Z8000 MICROPROCESSOR FAMILY

PROGRAMMING

1st EDITION

JUNE 1990

CONTENTS

Instruction Set **1**
Introduction 1
Functional Summary 1
Load and Exchange Instructions 1
Arithmetic Instructions 2
Logical Instructions 3
Program Control Instructions 3
Bit Manipulation Instructions 4
Rotate and Shift Instructions 5
Block Transfer And String Manipulation 5
Input/Output Instructions 6
CPU Control Instructions 7
Extended Instructions 7
Processor Flags 8
Condition Codes 9
Instruction Interrupts and Traps 9
Notation and Binary Encoding 10
Z8000 Instruction Descriptions and Formats 13

EPA Instruction Templates **169**
Programmers Quick Reference **173**

Instruction Set

Introduction

This Manual describes the instruction set of the Z8000. An overview of the instruction set is presented first, in which the instructions are divided into ten functional groups. The instructions in each group are listed, followed by a summary description of the instructions. Significant characteristics shared by the instructions in the group, such as the available addressing modes, flags affected, or interruptibility, are described. Unusual instructions or features that are not typical of predecessor microprocessors are pointed out.

Following the functional summary of the instruction set, flags and condition codes are discussed in relation to the instruction set. This is followed by a section discussing interruptibility of instructions and a description of traps. The last part of this chapter consists of a detailed description of each Z8000 instruction, listed in alphabetical order. This section is intended to be used as a reference by Z8000 programmers. The entry for each instruction includes a description of the instruction, addressing modes, assembly language mnemonics, instruction formats, execution times and simple examples illustrating the use of the instruction.

Functional Summary

This section presents an overview of the Z8000 instructions. For this purpose, the instructions may be divided into ten functional groups:

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

The Load and Exchange group includes a variety of instructions that provide for movement of data between registers, memory, and the program itself (i.e., immediate data). These instructions are supported with the widest range of addressing modes, including the Base (BA) and the Base Index (BX) mode which are available here only. None of these instructions affect any of the CPU flags.

The Load and Load Relative instructions transfer a byte, word, or long word of data from the source operand to the destination operand. A special one-word instruction, LDK, is also included to handle the frequent requirement for loading a small constant (0 to 15) into a register.

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

These instructions basically provide one of the following three functions:

- Load a register with data from a register or a memory location.

Functional Summary (Continued)

- Load a memory location with data from a register.
- Load a register or a memory location with immediate data.

The memory location is specified using any of the addressing modes (IR, DA, X, BA, BX, RA).

The Clear and Clear Byte instructions can be used to clear a register or memory location to zero. While this is functionally equivalent to a Load Immediate where the immediate data is zero, this operation occurs frequently enough to justify a special instruction that is more compact and faster.

The Exchange instructions swap the contents of the source and destination operands.

The Load Multiple instruction provides for efficient saving and restoring of registers. This can significantly lower the overhead of procedure calls and context switches such as those that occur at interrupts. The instruction allows any contiguous group of 1 to 16 registers to be transferred to or from a memory area, which can be designated using the DA, IR or X addressing modes. (R0 is considered to follow R15, e.g., one may save R9-R15 and R0-R3 with a single instruction.)

Stack operations are supported by the PUSH, PUSHL, POP, and POPL instructions. Any general-purpose register (or register pair in segmented mode) may be used as the stack pointer except R0 and RR0. The source operand for the Push instructions and the destination operand for the Pop instructions may be a register or a memory location, specified by the DA, IR, or X addressing modes. Immediate data can also be pushed onto a stack one word at a time. Note that byte operations are not supported, and the stack pointer register must contain an even value when a stack instruction is executed. This is consistent with the general restriction of using even addresses for word and long word accesses.

The Load Address and Load Address Relative instructions compute the effective address for the DA, X, BA, BX and RA modes and return the value in a register. They are useful for management of complex data structures.

The Arithmetic group consists of instructions for performing integer arithmetic. The basic instructions use standard two's complement binary format and operations. Support is also provided for implementation of BCD arithmetic.

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
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

Most of the instructions in this group perform an operation between a register operand and a second operand designated by any of the five basic addressing modes, and load the result into the register.

The arithmetic instructions in general alter the C, Z, S and P/V flags, which can then be tested by subsequent conditional jump instructions. The P/V flag is used to indicate arithmetic overflow for these instructions and it is referred to as the V (overflow) flag. The byte version of these instructions generally alters the D and H flags as well.

The basic integer (binary) operations are performed on byte, word or long word operands, although not all operand sizes are supported by all instructions. Multiple precision operations can be implemented in software using the Add with Carry, (ADC, ADCB),

Functional Summary (Continued)

Subtract with Carry (SBC, SBCB) and Extend Sign (EXTS, EXTSB, EXTSL) instructions.

BCD operations are not provided directly, but can be implemented using a binary addition (ADC, ADCB) or subtraction (SUBB, SBCB) followed by a decimal adjust instruction (DAB).

The Multiply and Divide instructions perform signed two's complement arithmetic on word or long word operands. The Multiply instruction (MULT) mutliplies two 16-bit operands and produces a 32-bit result, which is loaded into the destination register pair. Similarly, Multiply Long (MULTL) multiplies two 32-bit operands and produces a 64-bit result, which is loaded into the destination register quadruple. An overflow condition is never generated by a multiply, nor can a true carry be generated. The carry flag is used instead to indicate where the product has too many significant bits to be contained entirely in the low-order half of the destination.

The Divide instruction (DIV) divides a 32-bit number in the destination register pair by a 16-bit source operand and loads a 16-bit quotient into the low-order half of the destination register. A 16-bit remainder is loaded into the high-order half. Divide Long (DIVL) operates similarly with a 64-bit destination register quadruple and a 32-bit source. The overflow flag is set if the quotient is bigger than the low-order half of the destination, or if the source is zero.

Logical Instructions.

Instruction	Operand(s)	Name of Instruction
AND ANDB	dst, src	And
COM COMB	dst	Complement
OR ORB	dst, src	Or
TEST TESTB TESTL	dst	Test
XOR XORB	dst, src	Exclusive Or

The instructions in this group perform logical operations on each of the bits of the operands. The operands may be bytes or words; logical operations on long word are not supported (except for TESTL) but are easily implemented with pairs of instructions.

The two-operand instructions, And (AND, ANDB), Or (OR, ORB) and Exclusive-Or (XOR,XORB) perform the appropriate logical operations on corresponding bits of the destination register and the source operand, which can be designated by any of four basic addressing modes (R, IR, DA, IM, X). The result is loaded into the destination register.

Complement (COM, COMB) complements the bits of the destination operand. Finally, Test (TEST, TESTB, TESTL) performs the OR operation between the destination operand and zero and sets the flags accordingly. The Complement and Test instructions can use four basic addressing modes to specify the destination.

The Logical instructions set the Z and S flags based on the result of the operation. The byte variants of these instructions also set the Parity Flag (P/V) if the parity of the result is even, while the word instructions leave this flag unchanged. The H and D flags are not affected by these instructions.

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

This group consists of the instructions that affect the Program Counter (PC) and thereby control program flow. General-purpose registers and memory are not altered except for the processor stack pointer and the processor stack, which play a significant role in procedures and interrupts. (An exception is Decrement and Jump if Not Zero (DJNZ), which uses a register as a loop counter.) The flags are also preserved except for IRET which reloads the program status, including the flags, from the processor stack.

The Jump (JP) and Jump Relative (JR) instructions provide a conditional transfer of control to a new location if the processor flags

statisfy the condition specified in the condition code field of the instruction. Jump Relative is a one-word instruction that will jump to any instruction within the range -254 to +256 bytes from the current location. Most conditional jumps in programs are made to locations only a few bytes away; the Jump Relative instruction exploits this fact to improve code compactness and efficiency.

Call and Call Relative are used for calling procedures; the current contents of the PC are pushed onto the processor stack, and the effective address indicated by the instruction is loaded into the PC. The use of a procedure address stack in this manner allows straightforward implementation of nested and recursive procedures. Like Jump Relative, Call Relative provides a one-word instruction for calling nearby subroutines. However, a much larger range, -4092 to +4098 bytes for CALR instruction, is provided since subroutine calls exhibit less locality than normal control transfers.

Both Jump and Call instructions are available with the indirect register, indexed and relative address modes in addition to the direct address mode. These can be useful for implementing complex control structures such as dispatch tables.

The Conditional Return instruction is a companion to the Call instruction; if the condition specified in the instruction is satisfied, it loads the PC from the stack and pops the stack.

A special instruction, Decrement and Jump if Not Zero (DJNZ, DBJNZ), implements the control part of the basic PASCAL FOR loop in a one-word instruction.

System Call (SC) is used for controlled access to facilities provided by the operating system. It is implemented identically to a trap or interrupt: the current program status is pushed onto the system processor stack followed by the instruction itself, and a new program status is loaded from a dedicated part of the Program Status Area. An 8-bit immediate source field in the instruction is ignored by the CPU hardware. It can be retrieved from the stack by the software which handles system calls and interpreted as desired, for example as an index into a dispatch table to implement a call to one of the services provided by the operating system.

Interrupt Return (IRET) is used for returning from interrupts and traps, including system calls, to the interrupted routines. This is a privileged instruction.

Bit Manipulation Instructions

Instruction	Operand(s)	Name of Instruction
BIT BITB	dst, src	Bit Test
RES RESB	dst, src	Reset Bit
SET SETB	dst, src	Set Bit
TSET TSETB	dst	Test and Set
TCC TCCB	cc, dst	Test condition code

The instructions in this group are useful for manipulating individual bits in registers or memory. In most computers, this has to be done using the logical instructions with suitable masks, which is neither natural nor efficient.

The Bit Set (SET, SETB) and Bit Reset (RES, RESB) instructions set or clear a single bit in the destination byte or word, which can be in a register or in a memory location specified by any of the five basic addressing modes. The particular bit to be manipulated may be specified statically by a value (0 to 7 for byte, 0 to 15 for word) in the instruction itself or it may be specified dynamically by the contents of a register, which could have been computed by previous instructions. In the latter case, the destination is restricted to a register. These instructions leave the flags unaffected. The companion Bit Test instruction (BIT, BITB) similarly tests a specified bit and sets the Z flag according to the state of the bit.

The Test and Set instruction (TSET, TSETB) is useful in multiprogramming and multiprocessing environments. It can be used for implementing synchronization mechanisms between processes on the same or different CPUs.

Another instruction in this group, Test Condition Code (TCC, TCCB) sets a bit in the destination register based on the state of the flags as specified by the condition code in the instruction. This may be used to control subsequent operation of the program after the flags have been changed by intervening

instructions. It may also be used by language compilers for generating boolean values.

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
SLL SLLB SLLL	dst, src	Shift Left Logical
SRA SRAB SRAL	dst, src	Shift Right Arithmetic
SRL SRLB SRLL	dst, src	Shift Right Logical

This group contains a rich repertoire of instructions for shifting and rotating data registers.

Instructions for shifting arithmetically or logically in either direction are available. Three operand lengths are supported: 8, 16 and 32 bits. The amount of the shift, which may be any value up to the operand length, can be specified statically by a field in the instruction or dynamically by the contents of a register. The ability to determine the shift amount dynamically is a useful feature, which is not available in most minicomputers.

The rotate instructions will rotate the contents of a byte or word register in either direction by one or two bits; the carry bit can be included in the rotation. A pair of digit rotation instructions (RLDB, RRDB) are especially useful in manipulating BCD data.

Block Transfer And String Manipulation Instructions.

Instruction	Operand(s)	Name of Instruction
CPD CPDB	dst, src, r, cc	Compare and Decrement
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
LDDR LDRB	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

This is an exceptionally powerful group of instructions that provides a full complement of string comparison, string translation and block transfer functions. Using these instructions, a byte or word block of any length up to 64K bytes can be moved in memory; a byte or word string can be searched until a given value is found; two byte or word strings can be compared; and a byte string can be translated by using the value of each byte as the address of

its own replacement in a translation table. The more complex Translate and Test instructions skip over a class of bytes specified by a translation table, detecting bytes with values of special interest.

All the operations can proceed through the data in either direction. Furthermore, the operations may be repeated automatically while decrementing a length counter until it is zero, or they may operate on one storage unit per execution with the length counter decremented by one and the source and destination pointer registers properly adjusted. The latter form is useful for implementing more complex operations in software by adding other instructions within a loop containing the block instructions.

Any word register can be used as a length counter in most cases. If the execution of the instruction causes this register to be decremented to zero, the P/V flag is set. The autorepeat forms of these instructions always leave this flag set.

The D and H flags are not affected by any of these instructions. The C and S flags are preserved by all but the compare instructions.

These instructions use the Indirect Register (IR) addressing mode: the source and destination operands are addressed by the contents of general-purpose registers (word registers in nonsegmented mode and register pairs in segmented mode). Note that in the segmented mode, only the low-order half of the register pair gets incremented or decremented as with all address arithmetic in the Z8000.

The repetitive forms of these instructions are interruptible. This is essential since the repetition count can be as high as 65,536 and the instructions can take 9 to 14 cycles for each iteration after the first one. The instruction can be interrupted after any iteration. The address of the instruction itself, rather than the next one, is saved on the stack, and the contents of the operand pointer registers, as well as the repetition counter, are such that the instruction can simply be reissued after returning from the interrrupt without any visible difference in its effect.

This group consists of instructions for transferring a byte, word or block of data between peripheral devices and the CPU registers or memory. Two separate I/O address spaces with 16-bit addresses are recognized, a Standard I/O address space and a Special I/O address space. The latter is intended for use with special Z8000 Family devices, typically the Z-MMU. Instructions that operate on the Special I/O address space are prefixed with the word "special." Standard I/O and Special I/O instructions generate different codes on the CPU status lines. Normal 8-bit peripherals

Input/Output Instructions.

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

are connected to bus lines AD_0-AD_7. Standard I/O byte instructions use odd addresses only. Special 8-bit peripherals such as the MMU, which are used with special I/O instructions, are connected to bus lines AD_8-AD_{15}. Special I/O byte instructions use even addresses only.

The instructions for transferring a single byte or word (IN, INB, OUT, OUTB, SIN, SINB, SOUT, SOUTB) can transfer data between any general-purpose register and any port in either address space. For the Standard I/O instructions, the port number may be specified statically in the instruction or dynamically by the contents of the CPU register. For the Special I/O instructions the port number is specified statically.

The remaining instructions in this group form a powerful and complete complement of instructions for transferring blocks of data between I/O ports and memory. The operation of these instructions is very similar to that of the block move instructions described earlier, with the exception that one operand is always an I/O port which remains unchanged as the address of the other operand (a memory location) is incremented or decremented. These instructions are also interruptible.

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

All I/O instructions are privileged, i.e. they can only be executed in system mode. The single byte/word I/O instructions don't alter any flags. The block I/O instructions, including the single iteration variants, alter the Z and P/V flags. The latter is set when the repetition counter is decremented to zero.

The instructions in this group relate to the CPU control and status registers (FCW, PSAP, REFRESH, etc.), or perform other unusual functions that do not fit into any of the other groups, such as instructions that support multi-microprocessor operation. Most of these instructions are privileged, with the exception of NOP and the instructions operating on the flags (SETFLG, RESFLG, COMFLG, LDCTLB).

Extended Instructions. The Z8000 architecture includes a powerful mechanism for extending the basic instruction set through the use of external devices known as Extended Processing Units (EPUs). A group of six opcodes, 0E, 0F, 4E, 4F, 8E and 8F (in hexadecimal), is dedicated for the implementation of extended instructions using this facility. The five basic addressing modes (R, IR, DA, IM and X) can be used by extended instructions for accessing data for the EPUs.

There are four types of extended instructions in the Z8000 CPU instruction repertoire: EPU internal operations; data transfers between memory and EPU; data transfers between EPU and CPU; and data transfers between EPU flag registers and CPU flag and control word. The last type is useful when the program must branch based on conditions determined by the EPU. The action taken by the CPU upon encountering extended instructions is dependent upon the EPA control bit in the CPU's FCW. When this bit is set, it indicates that the system configuration includes EPUs; therefore, the instruction is executed. If this bit is clear, the CPU traps (extended instruction trap) so that a trap handler in software can emulate the desired operation.

Processor Flags

The processor flags are a part of the program status. They provide a link between sequentially executed instructions in the sense that the result of executing one instruction may alter the flags, and the resulting value of the flags may be used to determine the operation of a subsequent instruction, typically a conditional jump instruction. An example is a Test followed by a Conditional Jump:

```
TEST R1          !sets Z flag if R1 = 0!
JR Z, DONE       !go to DONE if Z flag is
                  set!
    .
    .
    .
DONE:
```

The program branches to DONE if the TEST sets the Z flag, i.e., if R1 contains zero.

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)

The flags are modified by many instructions, including the arithmetic and logical instructions.

Appendix C lists the instructions and the flags they affect. In addition, there are Z8000 CPU control instructions which 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 by the Z8000 processor for BCD arithmetic corrections. They are not used explicitly by the programmer.

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 contents of the flag register may also be saved in a register or memory.

The Carry (C) flag, when set, generally indicates a carry out of or a borrow into the high-order bit position of a register being used as an accumulator. For example, adding the 8-bit numbers 225 and 64 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 Carry flag plays an important role in the implementation of multiple-precision arithmetic (see the ADC, SBC instructions). It is also involved in the Rotate Left Through Carry (RLC) and Rotate Right Through Carry (RRC) instructions. One of these instructions is used to implement rotation or shifting of long strings of bits.

The Zero (Z) flag is set when the result register's contents are zero following certain operations. This is often useful for determining when a counter reaches zero. In addition, the block compare instructions use the Z flag to indicate when the specified comparison condition is satisfied.

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 number 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	0	1	0	0	1
+105	0	1	1	0	1	0	0	1
225	┌1	1	1	0	0	0	0	1
	└1	= Overflow 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 on byte operands. 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 (P = 0), the flag is reset. This flag is often referred to as the P/V flag.

Processor Flags (Continued)

The Block Move and String instructions and the Block I/O instructions use the P/V flag to indicate the repetition counter has decremented to 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 record whether an add or subtract instruction was executed so that the subsequent Decimal Adjust (DAB) instruction can perform its function correctly (See the DAB instruction for further discussion on the use of this flag).

The Half-Carry (H) flag indicates a carry out of bit 3 or a borrow into bit 3 as the result of adding or subtracting bytes containing two BCD digits each. 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.

Condition Codes

The first four flags, C, Z, S, and P/V, are used to control the operation of certain "conditional" instructions such as the Conditional Jump. The operation of these instructions is a function of whether a specified boolean condition on the four flags is satisfied or not. It would take 16 bits to specify any of the 65,536 (2^{16}) boolean functions of the four flags. Since only a very small fraction of these are generally of interest, this procedure would be very wasteful. Sixteen functions of the flag settings found to be frequently useful are encoded in a 4-bit field called the condition code, which forms a part of all conditional instructions.

The condition codes and the flag settings they represent are listed in Section 6.6.

Although there are sixteen unique condition codes, the assembler recognizes more than sixteen mnemonics for the conditional codes. Some of the flag settings have more than one meaning for the programmer, depending on the context (PE & OV, Z & EQ, C & ULT, etc.). Program clarity is enhanced by having separate mnemonics for the same binary value of the condition codes in these cases.

Instruction Interrupts and Traps

This section looks at the relation-ship between instructions ans interrupts.

When the CPU receives an interrupt request, and it is enabled for interrupts of that class, the interrupt is normally processed at the end of the current instruction. However, certain instructions which might take a long time to complete are designed to be interruptible so as to minimize the length of time it takes the CPU to respond to an interrupt. These are the iterative versions of the String and Block instructions and the Block I/O instruction. If an interrupt request is received during one of these interruptible instructions, the instruction is suspended after the current iteration. The address of the instruction itself, rather than the address of the following instruction, is saved on the stack, so that the same instruction is executed again when the interrupt handler executes an IRET. The contents of the repetition counter and the registers which index into the block operands are such that after each iteration when the instruction is reissued upon returning from an interrupt, the effect is the same as if the instruction were not interrupted. This assumes, of course, the interrupt handler preserved the registers, which is a general requirement on interrupt handlers.

The longest noninterruptible instruction that can be used in normal mode is Divide Long (749 cycles in the worst case). Multi-Micro-Request, a privileged instruction, can take longer depending on the contents of the destination register.

Traps are synchronous events that result from the execution of an instruction. The action of the CPU in response to a trap condition is similar to the case of an interrupt (see Section 7). Traps are non-maskable.

Instruction Interrupts and Traps (Continued)

The Z8000 CPUs implement four kinds of traps:

- Extended Instruction
- Privileged Instruction in normal mode
- Segmentation violation
- System Call

The Extended Instruction trap occurs when an Extended Instruction is encountered, but the Extended Processor Architecture Facility is disabled, i.e., the EPA bit in the FCW is a zero. This allows the same software to be run on Z8000 system configurations with or without EPUs. On systems without EPUs, the desired extended instructions can be emulated by software which is invoked by the Extended Instruction trap.

The privileged instruction trap serves to protect the integrity of a system from erroneous or unauthorized actions of arbitrary processes. Certain instructions, called privileged instructions, can only be executed in system mode. An attempt to execute one of these instructions in normal mode causes a privileged instruction trap. All the I/O instructions and most of the instructions that operate on the FCW are privileged, as are instructions like HALT and IRET.

The System Call instruction always causes a trap. It is used to transfer control to system mode software in a controlled way, typically to request supervisor services.

Notation and Binary Encoding

The rest of this chapter consists of detailed descriptions of each instruction, listed in alphabetical order. This section describes the notational conventions used in the instruction descriptions and the binary encoding for some of the common instruction fields (e.g., register designation fields).

The description of an instruction begins with the instruction mnemonic and instruction name in the top part of the page. Privileged instructions are also identified at the top.

The assembler language syntax is then given in a single generic form that covers all the variants of the instruction, along with a list of applicable addressing modes.

Example:

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

The operation of the instruction is presented next, followed by a detailed discussion of the instruction.

The next part specifies the effect of the instruction on the processor flags. This is followed by a table that presents all the variants of the instruction for each applicable addressing mode and operand size. For each of these variants, the following information is provided:

A. Assembler Language Syntax. The syntax is shown for each applicable operand width (byte, word or long). The invariant part of the syntax is given in UPPER CASE and must appear as shown. Lower case characters represent the variable part of the syntax, for which suitable values are to be substituted. The syntax shown is for the most basic form of the instruction recognized by the assembler. For example,

ADD Rd,#data

represents a statement of the form ADD R3,#35. The assembler will also accept variations like ADD TOTAL, #NEW-DELTA where TOTAL, NEW and DELTA have been suitably defined.

The following notation is used for register operands:

Rd, Rs, etc.:	a word register in the range R0-R15
Rbd Rbs:	a byte register RHn or RLn where n = 0 – 7
RRd RRs:	a register pair RR0, RR2, ... RR14
RQd:	a register quadruple RQ0, RQ4, RQ8 or RQ12

The "s" or "d" represents a source or destination operand. Address registers used in Indirect, Base and Base Index addressing modes represent word registers in nonsegmented mode and register pairs in segmented mode. A one-word register used in segmented

Notation and Binary Encoding (Continued)

mode is flagged and a footnote explains the situation.

B. Instruction Format. The binary encoding of the instruction is given in each case for both the nonsegmented and segmented modes. Where applicable, both the short and long forms of the segmented version are given (SS and SL).

The instruction formats for byte and word versions of an instruction are usually combined. A single bit, labeled "w," distinguishes them: a one indicates a word instruction, while a zero indicates a byte instruction.

Fields specifying register operands are identified with the same symbols (Rs, RRd, etc.) as in Assembler Language Syntax. In some cases, only nonzero values are permitted for certain registers, such as index registers. This is indicated by a notation of the form "RS ≠ 0."

The binary encoding for register fields is as follows:

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
	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

For easy cross-references, the same symbols are used in the Assembler Language Syntax and the instruction format. In the case of addresses, the instruction format in segmented mode uses "segment" and "offset" to correspond to "address," while the instruction format contains "displacement," indicating that the assembler has computed the displacement and inserted it as indicated.

A condition code is indicated by "cc" in both the Assembler Language Syntax and the instruction formats. The condition codes, the flag settings they represent, and the binary encoding in the instruction are as follows:

Notation and Binary Encoding (Continued)

Code	Meaning	Flag Setting	Binary
F	Always false		0000
	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, NC-UGE, PE-OV, PO-NOV.

C. Cycles. This line gives the execution time of the instructions in CPU cycles.

D. Example. A short assembly language example is given showing the use of the instruction.

ADC
Add With Carry

ADC dst, src
ADCB

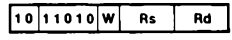
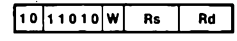
dst: R
src: R

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

Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
R:	ADC Rd, Rs ADCB Rbd, Rbs		5		5

Example: Long addition can 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
ADDB
ADDL

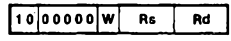
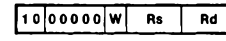
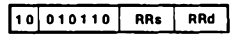
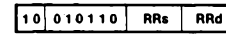
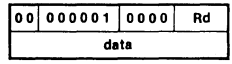
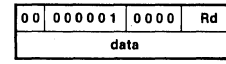
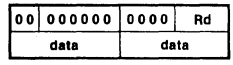
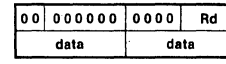
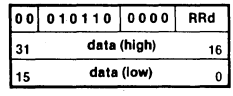
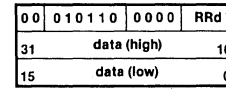
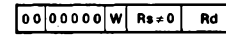
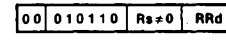
dst: R
src: R, IM, IR, DA, X

Operation: dst ← dst + src

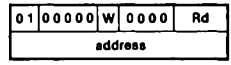
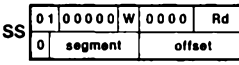
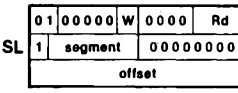
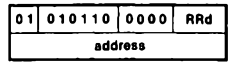
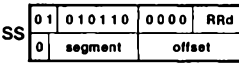
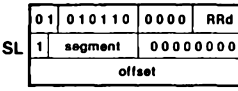
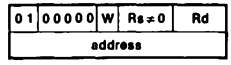
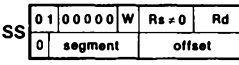
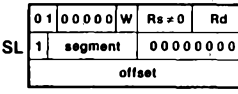
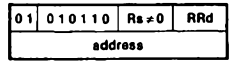
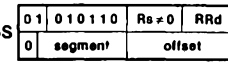
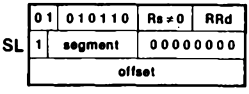
The source operand is added to the destination operand and the sum is stored in the destination. The contents of the source are not affected. Two's complement addition is performed.

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

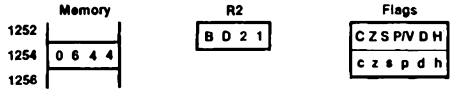
Source Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
R:	ADD Rd, Rs ADDB Rbd, Rbs		4		4
	ADDL RRd, RRs		8		8
IM:	ADD Rd, #data		7		7
	ADDB Rbd, #data		7		7
	ADDL RRd, #data		14		14
IR:	ADD Rd, @Rs[1] ADDB Rbd, @Rs[1]		7		7
	ADDL RRd, @Rs[1]		14		14

ADD
Add

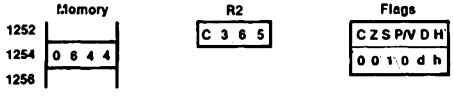
Source Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
DA:	ADD Rd, address ADDB Rbd, address		9		10
					12
	ADDL RRd, address		15		16
					18
X:	ADD Rd, addr(Rs) ADDB Rbd, addr(Rs)		10		10
					13
	ADDL RRd, addr(Rs)		16		16
					19

Example: ADD R2, AUGEND !augend A located at %1254!

Before instruction execution:



After instruction execution:



Note 1: Word register in nonsegmented mode, register pair in segmented mode.

AND
And

AND dst, src
ANDB

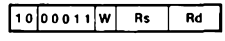
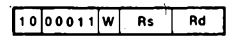
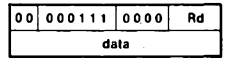
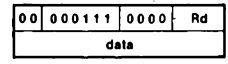
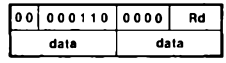
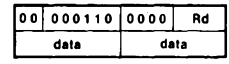
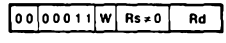
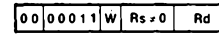
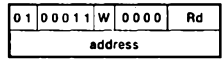
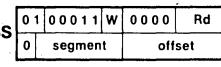
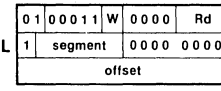
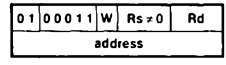
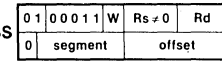
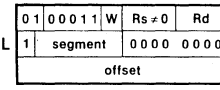
dst: R
src: R, IM, IR, DA, X

Operation: dst ← dst AND src

A logical AND operation is performed between the corresponding bits of the source and destination operands, and the result is stored in the destination. A one bit is stored wherever the corresponding bits in the two operands are both ones; otherwise a zero bit is stored. The source contents 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: AND — unaffected; ANDB — set if parity of the result is even; cleared otherwise
D: Unaffected
H: Unaffected

Source Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
R:	AND Rd, Rs ANDB Rbd, Rs		4		4
IM:	AND Rd, #data		7		7
	ANDB Rbd, #data		7		
IR:	AND Rd, @Rs! ANDB Rbd, @Rs!		7		7
DA:	AND Rd, address ANDB Rbd, address		9	SS 	10
				SL 	12
X:	AND Rd, addr(Rs) ANDB Rbd, addr(Rs)		10	SS 	10
				SL 	13

Example: ANDB RL3, # %CE

Before instruction execution:

RL3
1 1 1 0 0 1 1 1

Flags
C Z S P/V D H
c z s p d h

After instruction execution:

RL3
1 1 0 0 0 1 1 0

Flags
C Z S P/V D H
c 0 1 1 d h

Note 1: Word register in nonsegmented mode, register pair in segmented mode.

BIT
Bit Test

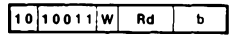
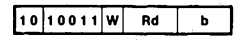
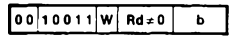
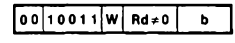
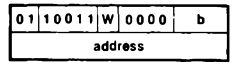
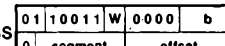
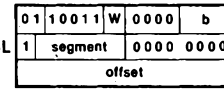
BIT dst, src	dst: R, IR, DA, X
BITB	src: IM
	or
	dst: R
	src: R

Operation: Z ← NOT dst (src)

The specified bit within the destination operand is tested, and the Z flag is set to one if the specified bit is zero; otherwise the Z flag is cleared to zero. The contents of the destination are not affected. The bit number (the source) can be specified statically as an immediate value, or dynamically as a word register whose contents are the bit number. In the dynamic 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. Note that 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 for BITB.

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

Bit Test Static

Destination Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
R:	BIT Rd, b BITB Rbd, b		4		4
IR:	BIT @Rd[1], b BITB @Rd[1], b		8		8
DA:	BIT address, b BITB address, b		10	SS 	11
				SL 	13

Bit Test Static (Continued)

Source Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
X:	BIT addr(Rd), b BITB addr(Rd), b	01 10011 W Rd≠0 b address	11	SS: 01 10011 W Rd≠0 b 0 segment offset	11
				SL: 01 10011 W Rd≠0 b 1 segment 0000 0000 offset	14
R:	BIT Rd, Rs BITB Rbd, Rs	00 10011 W 0000 Rs 0000 Rd 0000 0000	10	00 10011 W 0000 Rs 0000 Rd 0000 0000	10

Example: If register RH2 contains %B2 (10110010), the instruction

BITB RH2, #0

will leave the Z flag set to 1.

Note 1: Word register in nonsegmented mode, register pair in segmented mode.

CALL
Call

CALL dst dst: IR, DA, X

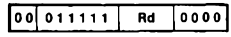
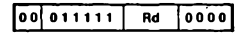
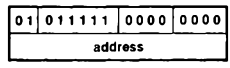
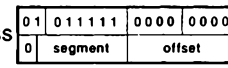
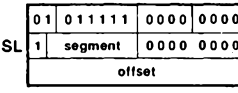
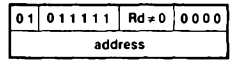
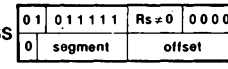
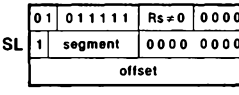
Operation:

Nonsegmented
SP ← SP − 2
C → SP ← PC
PC ← dst

Segmented
SP ← SP − 4
@SP ← PC
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 in nonsegmented mode, or RR14 in segmented mode. (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 the called procedure.
At the end of the procedure a RET instruction can be used to return to original program. RET pops the top of the processor stack back into the PC.

Flags: No flags affected

Destination Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
IR:	CALL @Rd		10		15
DA:	CALL address		12	SS 	18
				SL 	20
X:	CALL addr(Rd)		13	SS 	18
				SL 	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 instruction

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.

Note 1: Word register in nonsegmented mode, register pair in segmented mode.

CALR
Call Relative

CALR dst dst: RA

Operation:

Nonsegmented	Segmented
SP ← SP − 2	SP ← SP − 4
@SP ← PC	@SP ← PC
PC ← PC − (2 × displacement)	PC ← PC − (2 × displacement)

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.

The destination address is calculated by doubling the displacement in the instruction, then subtracting this value from the current value of the PC to derive the destination address. 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 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.

Flags: No flags affected

Destination Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
RA:	CALR address		10		15

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

```
CALR  PROC
```

causes the stack pointer to be decremented to %3000, the value %1004 (the address following the CALR instruction) to be loaded into the word location %3000, and the program counter to be loaded with the address of the first instruction in procedure PROC.

CLR
Clear

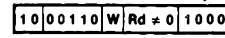
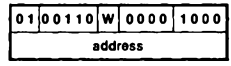
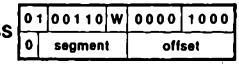
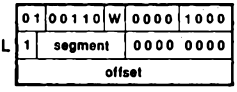
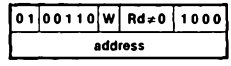
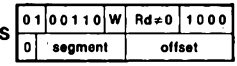
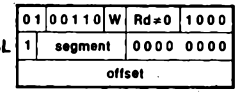
CLR dst
CLRB

dst: R, IR, DA, X

Operation: dst ← 0

The destination is cleared to zero.

Flags: No flags affected.

Destination Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
R:	CLR Rd CLRB Rbd		7		7
IR:	CLR @Rd CLRB @Rd		8		8
DA:	CLR address CLRB address		11	SS 	12
				SL 	14
X:	CLR addr(Rd) CLRB addr(Rd)		12	SS 	12
				SL 	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.

Note 1: Word register in nonsegmented mode, register pair in segmented mode.

COM
Complement

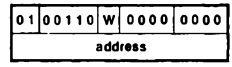
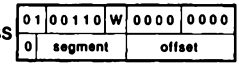
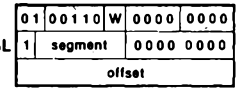
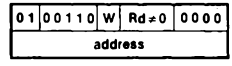
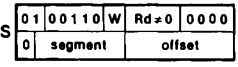
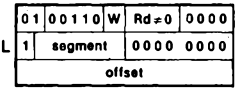
COM dst
COMB

dst: R, IR, DA, X

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

Destination Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
R:	COM Rd COMB Rbd		7		7
IR:	COM @Rd[1] COMB @Rd[1]		12		12
DA:	COM address COMB address		15	SS 	16
				SL 	18
X:	COM addr(Rd) COMB addr(Rd)		16	SS 	16
				SL 	19

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

```
COM  R1
```

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

Note 1: Word register in nonsegmented mode, register pair in segmented mode.

COMFLG
Complement Flag

COMFLG flag Flag: C, Z, S, P, V
FLAGS (4:7) ← FLAGS (4:7) XOR instruction (4:7)

Operation: Any combination of the C, Z, S, P or V flags is complemented (each one bit is changed to zero, and vice-versa). The flags to be complemented are encoded in a field in the instruction. If the bit in the field is one, the corresponding flag is complemented; if the bit is zero, the flag is left unchanged. 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

	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
	COMFLG flags	10001101 | CZSP/V | 0101	7	10001101 | CZSP/V | 0101	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
CPB
CPL

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

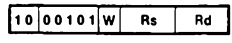
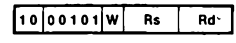
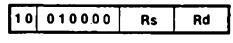
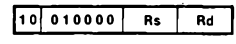
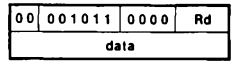
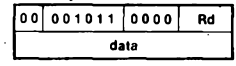
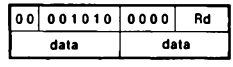
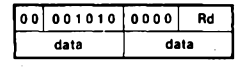
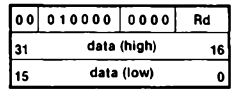
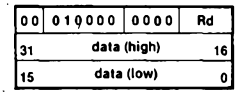
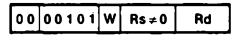
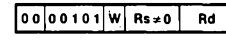
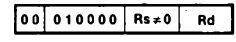
Operation: dst – src

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. There are two variants of this instruction: Compare Register compares the contents of a register against an operand specified by any of the five basic addressing modes; Compare Immediate performs a comparison between an operand in memory and an immediate value.

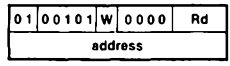
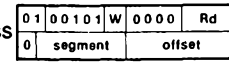
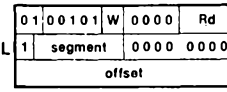
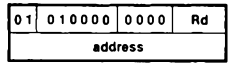
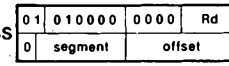
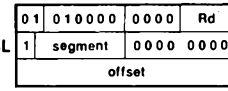
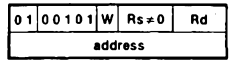
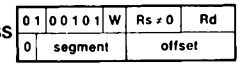
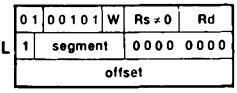
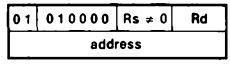
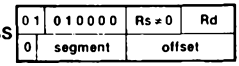
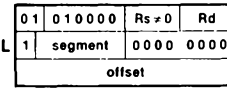
Flags:

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

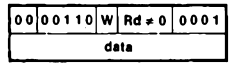
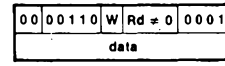
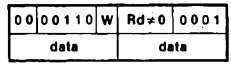
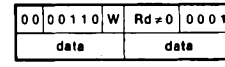
Compare Register

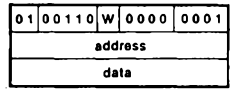
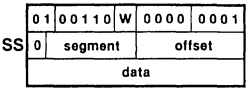
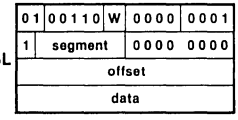
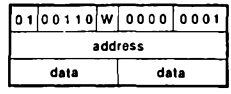
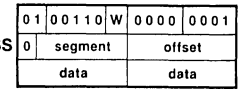
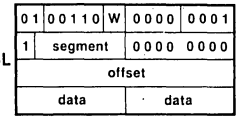
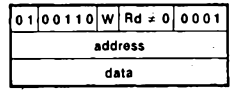
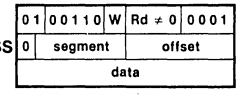
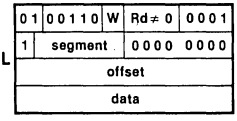
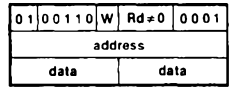
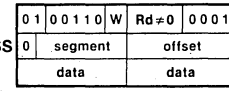
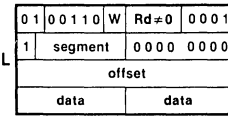
Source Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
R:	CP Rd, Rs CPB Rbd, Rbs		4		4
	CPL RRd, RRs		8		8
IM:	CP Rd, #data		7		7
	CPB Rbd, #data		7		7
	CPL RRd, #data		14		14
IR:	CP Rd, @Rs CPB Rbd, @Rs		7		7
	CPL RRd, @Rs		14		14

CP
Compare

Source Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
DA:	CP Rd, address CPB Rbd, address		9	SS 	10
				SL 	12
	CPL RRd, address		15	SS 	16
				SL 	18
X:	CP Rd, addr(Rs) CPB Rbd, addr(Rbs)		10	SS 	10
				SL 	13
	CPL RRd, addr(Rs)		16	SS 	16
				SL 	19

Compare Immediate

Destination Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
IR:	CP @Rd!, #data		11		11
	CPB @Rd!, #data		11		11

Destination Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
DA:	CP address, #data		14		15
					17
	CPB address, #data		14		15
					17
X:	CP addr(Rd), #data		15		15
					18
	CPB addr(Rd), #data		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.

Note 1: Word register in nonsegmented mode, register pair in segmented mode.

CPD
Compare and Decrement

CPD dst, src, r, cc — dst: IR
CPDB — src: IR

Operation:

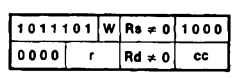
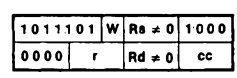
dst − src
AUTODECREMENT src (by 1 if byte, by 2 if word)
r ← r − 1

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 are 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 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

Source Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
IR:	CPD Rd, @Rs[1], r, cc CPDB Rbd, @Rs[1], r, cc		20		20

Example:

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

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. For segmented mode, R1 must be replaced by a register pair.

Note 1: Word register in nonsegmented mode, register pair in segmented mode.

CPDR
Compare Decrement and Repeat

CPDR dst, src, r, cc — dst: IR
CPDRB — src: IR

Operation:

dst − src
AUTODECREMENT src (by 1 if byte; by 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 are 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 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 "r" (used as a counter) is 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).

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.

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

Source Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles[2]	Instruction Format	Cycles[2]
IR:	CPDR Rd, @Rs[1], r, cc CPDRB Rbd, @Rs[1], r, cc	1011101 | W | Rs ≠ 0 | 1100 0000 | r | Rd ≠ 0 | cc	11 + 9n	1011101 | W | Rs ≠ 0 | 1100 0000 | r | Rd ≠ 0 | cc	11 + 9n

Example:

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

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

will leave the Z flag set indicating the condition was met. Register R2 will contain the value %2002, R3 will still contain 5, and R8 will contain 5. For segmented mode, a register pair would be used instead of R2.

Note 1: Word register in nonsegmented mode, register pair in segmented mode.
Note 2: n = number of data elements compared.

CPI
Compare and Increment

CPI dst, src, r, cc — dst: IR
CPIB — src: IR

Operation:

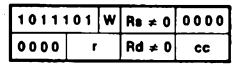
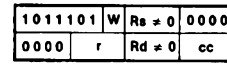
dst – src
AUTOINCREMENT src (by 1 if byte; by 2 if word)
r ← r – 1

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 are compared to (subtracted from) the destination operand and the Z flag is set if the condition code is specified by "cc" would be set by the comparison; otherwise the Z flag is cleared. See 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 source, destination, and counter registers must be separate and non-overlapping registers. 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

Source Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
IR:	CPI Rd, @Rs[1], r, cc CPIB Rbd, @Rs[1], r, cc		20		20

Example: This instruction can be used in a "loop" of instructions that 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 of instructions (to be executed in non-segmented mode) "scans while numeric," that is, a string is searched until either an ASCII character not in the range "0" to "9" (see Appendix C) is found, or the end of the string is reached. This involves a range check on each character (byte) in the string. For segmented mode, R1 must be changed to a register pair.

```
        LD      R3, #STRLEN          !initialize counter!
        LDA     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 > '0'!
        JR      Z, NONNUMERIC
        JR      NOV, LOOP            !repeat until counter = 0!
DONE:
        .
        .
        .
NONNUMERIC:                          !handle non-numeric char!
```

Note 1: Word register in nonsegmented mode, register pair in segmented mode.

CPIR
Compare, Increment and Repeat

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

Operation:

dst − src
AUTOINCREMENT src (by 1 if byte; by 2 if word)
r ← r − 1
repeat until cc is true or R = 0

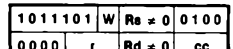
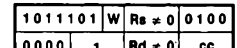
This instruction is used to search a string of data for an element meeting the specified condition. The contents of the location addressed by the source register are compared to (subtracted from) the destination operand, and the Z flag is set if the condition code specified by "cc" would be set by the comparison; otherwise the Z flag is cleared. See 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). The source, destination, and counter registers must be separate and non-overlapping registers.

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.

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

Source Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles[2]	Instruction Format	Cycles[2]
IR:	CPIR Rd, @Rs!, r, cc CPIRB Rbd,@Rs!, r, cc		11 + 9n		11 + 9n

Example: The following sequence of instructions (to be executed in nonsegmented mode) 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. For segmented mode, R1 must be changed to a register pair.

```
LDA     R1, STRSTART
LD      R3, #STRLEN
LDB     RL0, #% D              !hex code for return is D!
CPIRB   RL0, @R1, R3, EQ
JR      Z, FOUND
```

Note 1: Word register in nonsegmented mode, register pair in segmented mode.
Note 2: n = number of data elements compared.

CPSD
Compare String and Decrement

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

Operation:

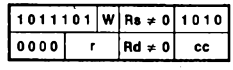
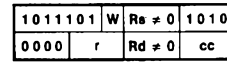
dst − src
AUTODECREMENT dst and src (by 1 if byte; by 2 if word)
r ← r − 1

This instruction can be used to compare two strings of data until the specified condition is true. The contents of the location addressed by the source register are 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 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: Cleard if there is a carry from the most significant bit of the result of the comparison; set otherwise, indicating a "borrow". Thus this flag will be set if the destination is less than the source when viewed as unsigned integers.
Z: Set if the condition code generated by the comparison matches cc; cleared otherwise
S: Set is the result of the comparison is negative; cleared otherwise
V: Set if the result of decrementing r is zero; cleared otherwise
D: Unaffected
H: Unaffected

Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
IR:	CPSD @Rd[1], @Rs[1], r, cc CPSDB @Rd[1],@Rs[1],r,cc		25		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 instruction (executed in nonsegmented mode)

CPSDB @R2, @R3, R4, UGE

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

Note 1: Word register in nonsegmented mode, register pair in segmented mode.

CPSDR
Compare String, Decrement and Repeat

CPSDR dst, src,r, cc — dst: IR
CPSDRB — src: IR

Operation:

dst − src
AUTODECREMENT dst and src (by 1 if byte; by 2 if word)
r ← r − 1
repeat until cc is true or r = 0

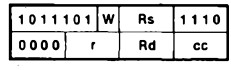
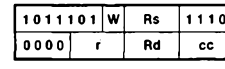
This instruction is used to compare two strings of data until the specified condition is true. The contents of the location addressed by the source register are compared to (subtracted from) the contents of the location addressed by the destination register. The Z flag is set if the condition code specified by "cc" would be set by the comparison; otherwise the Z flag is cleared. See 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 from 1 to 32768 words long (the value of r must not be greater than 32768 for CPSDR).

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.

Flags:

C: Cleared if there is a carry from the most significant bit of the result of the comparison; set otherwise, indicating a "borrow". Thus this flag will be set if the destination is less than the source when viewed as unsigned integers
Z: Set if the conditon code generated by the comparison matches cc; cleared otherwise
S: Set if the result of the comparison is negative; cleared otherwise
V: Set if the result of decrementing r is zero; cleared otherwise
D: Unaffected
H: Unaffected

Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
IR:	CPSDR@Rd^{l},@Rs^{l},r,cc CPSDRB@Rd^{l},@Rs^{l},r,cc		11 + 14n		11 + 14n

CPSDR
Compare String, Decrement and Repeat

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 instruction (executed in nonsegmented mode)

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

leaves the Z flag set to 1 since the condition code would have been "equal" (locations %1000 and %2000 both contain the value 0). The V flag will be set to 1 indicating r was decremented to 0. R13 will contain %0FFE, R14 will contain %1FFE, and R0 will contain 0. For segmented mode, R13 and R14 must be changed to register pairs.

Note 1: Word register in nonsegmented mode, register pair in segmented mode.

Note 2: n = number of data elements compared.

CPSI
Compare String and Increment

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

Operation:

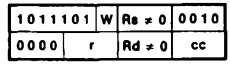
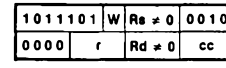
dst – src
AUTOINCREMENT dst and src (by 1 if byte, by 2 if word)
r ← r – 1

This instruction can be used to compare two strings of data until the specified condition is true. The contents of the location addressed by the source register are 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 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

Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
IR:	CPSI @Rd¹,@Rs¹,r,cc CPSIB @Rd¹,@Rs¹,r,cc		25		25

CPSI
Compare String and Increment

Example: This instruction can be 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 of instructions, to be executed in nonsegmented mode, 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 C) by resetting bit 5 of each character (byte) to 0 before comparison.

```
        LDA     R1, SRCSTART        !load start addresses!
        LDA     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!
```

In segmented mode, R1 and R2 must both be register pairs.

Note 1: Word register in nonsegmented mode, register pair in segmented mode.

CPSIR
Compare String, Increment and Repeat

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

Operation:

dst – src
AUTOINCREMENT dst and src (by 1 if byte, by 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 are 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 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 from 1 to 32768 words long (the value of r must not be greater than 32768 for CPSIR).

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 cycles should be added to this instruction's execution time for each interrupt request that is accepted. The source, destination, and counter registers must be separate and non-overlapping registers.

Flags:

C: Cleared if there is a carry from the most significant bit of the result of the last comparison made; set otherwise, indicating a "borrow". Thus this flag will be set if the last destination element is less than the last source element when viewed as unsigned integers.
Z: Set if the condition code generated by the comparison matches cc; cleared otherwise
S: Set if the result of the last comparison made is negative; cleared otherwise
V: Set if the result of decrementing r is zero; cleared otherwise
D: Unaffected
H: Unaffected

Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles[2]	Instruction Format	Cycles[2]
IR:	CPSIR @Rd[1],@Rs[1],r,cc CPSIRB @Rd[1],@Rs[1],r,cc	1011101 | W | Rs ≠ 0 | 0110 0000 | r | Rd ≠ 0 | cc	11 + 14n	1011101 | W | Rs ≠ 0 | 0110 0000 | r | Rd ≠ 0 | cc	11 + 14n

CPSIR
Compare String, Increment and Repeat

Example: The CPSIR instruction can be used to compare test strings for lexicographic order. (For most common character encoding — for example, ASCII and EBCDIC — lexicographic order is the same as alphabetic order for alphabetic test strings that do not contain blanks.)

Let S1 and S2 be text strings of lengths L1 and L2. According to lexicographic ordering, S1 is said to be "less than" or "before" S2 if either of the following is true:

- At the first character position at which S1 and S2 contain different characters, the character code for the S1 character is less than the character code for the S2 character.
- S1 is shorter than S2 and is equal, character for character, to an initial substring of S2.

For example, using the ASCII character code, the following strings are ascending lexicographic order:

A
A ␣ A
A B C
A B CD
A B D

Let us assume that the address of S1 is in RR2, the address of S2 is in RR4, the lengths L1 and L2 of S1 and S2 are in R0 and R1, and the shorter of L1 and L2 is in R6. The the following sequence of instructions will determine whether S1 is less than S2 in lexicographic order:

```
CPSIRB   @RR2, @RR4, R6, NE     !Scan to first unequal character!
                                 !The following flags settings are possible:
                                  Z = 0, V = 1: Strings are equal through L1
                                  character. (Z = 0, V = 0 cannot occur).
                                  Z = 1, V = 0 or 1: A character position was
                                  found at which the strings are unequal.
                                  C = 1 (S = 0 or 1): The character in the RR2
                                  string was less (viewed as numbers from 0 to
                                  255, not as numbers from -128 to +127).
                                  C = 0 (S = 0 or 1): The character in the RR2
                                  string was not less!
JR Z,CHAR_COMPARE                !If Z=1, compare the characters!
CP R0,R1                         !Otherwise, compare string lengths!
JR LT, S1_IS_LESS
JR S1_NOT_Less
CHAR_COMPARE:
JR ULT, S1_IS_LESS               !ULT is another name for C=1!
S1_NOT LESS:
          .
          .
          .
S1_IS_LESS:
```

DAB
Decimal Adjust

DAB dst dst: R

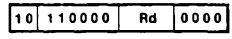
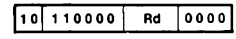
Operation: dst ← DA dst

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
	0	0-9	0	0-9	00	0
	0	0-8	0	A-F	06	0
ADDB	0	0-9	1	0-3	06	0
ADCB	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	0	0-9	0	0-9	00	0
SBCB	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

Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
R:	DAB Rbd		5		5

DAB
Decimal Adjust

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

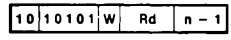
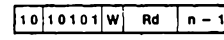
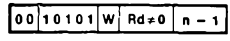
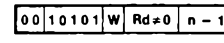
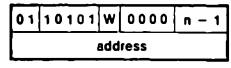
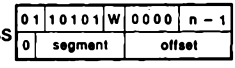
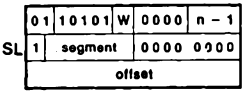
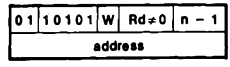
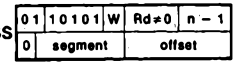
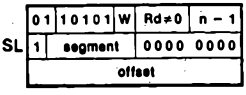
Operation: dst ← dst − src (where src = 1 to 16)

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 defaults to the value 1.

The value of the source field in the instruction is one less than the actual value of the source operand. Thus, the coding in the instruction for the source ranges from 0 to 15, which corresponds to the source values 1 to 16.

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, 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: Unaffected
H: Unaffected

Destination Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
R:	DEC Rd, #n DECB Rbd, #n		4		4
IR:	DEC @Rd[1], #n DECB @Rd[1], #n		11		11
DA:	DEC address, #n DECB address, #n		13	SS 	14
				SL 	16
X:	DEC addr(Rd), #n DECB addr(Rd), #n		14	SS 	14
				SL 	17

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

```
DEC  R10
```

will leave the value %0029 in R10.

Note 1: Word register in nonsegmented mode, register pair in segmented mode.

DI Privileged

Disable Interrupt

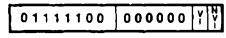
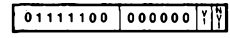
DI Int Int: VI, NVI

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.

	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
	DI int		7		7

Example: If the NVI and VI control bits are set (1) in the FCW, the instruction:

```
DI VI
```

will leave the NVI control bit in the FCW set (1) and will leave the VI control bit in the FCW cleared (0).

DIV
Divide

DIV dst, src
DIVL

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

Operation:

Word: (dst is register pair, src is word):
dst (0:31) is divided by src (0:15)
(dst (0:31) = quotient × src (0:15) + remainder)
dst (16:31) ← quotient
dst (0:15) ← remainder

Long: (dst register quadruple, src is long word or register pair):
dst (0:63) is divided by src (0:31)
(dst (0:63) = quotient × src (0:31) + remainder)
dst (32:63) ← quotient
dst (0:31) ← remainder

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.

There a four possible outcomes of the Divide instruction, depending on the division, and the resulting quotient:

CASE 1. If the quotient is within the range -2^{15} to $2^{15} - 1$ inclusive for DIV or -2^{31} to $2^{31} - 1$ inclusive for DIVL, then the quotient and remainder are left in the destination register as defined above, the overflow and carry flags are cleared to zero, and the sign and zero flags are set according to the value of the quotient.

CASE 2. If the divisor is zero, the destination register remains unchanged, the overflow and zero flags are set to one and the carry and sign flags are cleared to zero.

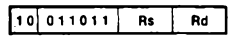
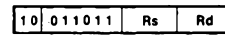
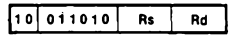
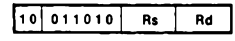
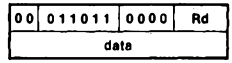
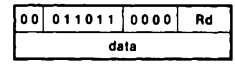
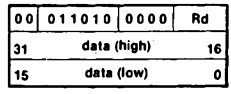
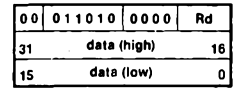
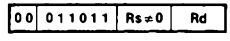
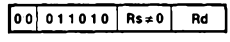
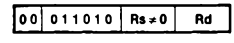
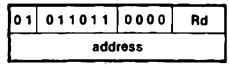
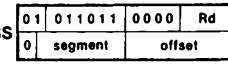
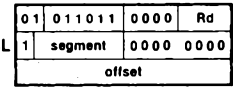
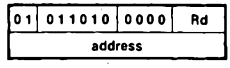
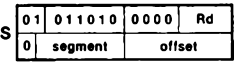
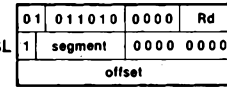
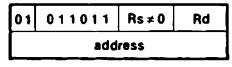
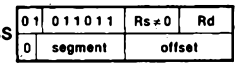
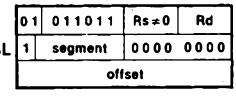
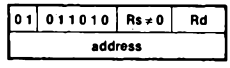
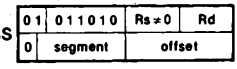
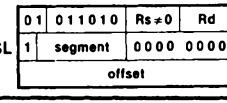
CASE 3. If the quotient is outside the range -2^{16} to $2^{16} - 1$ inclusive for DIV or -2^{32} to $2^{32} - 1$ inclusive for DIVL, the destination register contains an undefined value, the overflow flag is set to one, the carry and zero flags are cleared to zero, and the sign flag is undefined.

CASE 4. If the quotient is inside the range of case 3 but outside the range of case 1, then all but the sign bit of the quotient and all of the remainder are left in the destination register, the overflow and carry flags are set to one, and the sign and zero flags are set according to the value of the quotient. In this case, the sign flag can be replicated by subsequent instruction into the high-order half of the destination to produce the two's complement representation of the quotient in the same precision as the original dividend.

Flags:

C: Set if V is set and the quotient lies in the range from -2^{16} to $2^{16} - 1$ inclusive for DIV or in the range from -2^{32} to $2^{32} - 1$ inclusive for DIVL; cleared otherwise
Z: Set if the quotient or divisor is zero; cleared otherwise
S: Undefined if V is set and C is clear (overflow); otherwise set if the quotient is negative, cleared if the quotient is non-negative.
V: Set if the divisor is zero or if the computed quotient lies outside the range from -2^{15} to $2^{15} - 1$ inclusive for DIV or outside range from -2^{31} to $2^{31} - 1$ inclusive for DIVL; cleared otherwise
D: Unaffected
H: Unaffected

DIV
Divide

Source Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles[2]	Instruction Format	Cycles[2]
R:	DIV RRd, Rs				
	DIVL RQd, RRs				
IM:	DIV RRd, #data				
	DIVL RQd, #data				
IR:	DIV RRd, @Rs[1]				
	DIVL RQd, @Rs[1]				
DA:	DIV RRd, address			SS  SL 	
	DIVL RQD, address			SS  SL 	
X:	DIV RRd, addr(Rs)			SS  SL 	
	DIVL RQd, addr(Rs)			SS  SL 	

Note 1: Word register in nonsegmented mode, register pair in segmented mode.
Note 2: Execution times for each instruction are given in the table under Example.

Example: The following table gives the DIV instruction execution times for word and long word operands in all possible addressing modes.

src	Word			Long Word		
	NS	SS	SL	NS	SS	SL
R	107	—	—	744	—	—
IM	107	—	—	744	—	—
IR	107	107	107	744	744	744
DA	108	108	111	745	746	748
X	109	109	112	746	746	749
(Divisor is zero)						
R	13	13	13	30	30	30
IM	13	13	13	30	30	30
IR	13	13	13	30	30	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	25	25	51	51	51
IM	25	25	25	51	51	51
IR	25	25	25	51	51	51
DA	26	27	29	52	53	55
X	27	27	30	53	53	56

Note that for proper 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.

If register RR0 (composed of word register 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).

Note 1: Word register in nonsegmented mode, register pair in segmented mode.

Note 2: The execution time for the instruction will be lower than indicated for divide by zero and certain overflow conditions.

DJNZ
Decrement and Jump if Not Zero

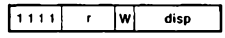
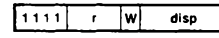
DJNZ R, dst
DBJNZ dst: RA

Operation: R ← R − 1
If R ≠ 0 then PC ← PC − (2 × displacement)

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 method of loop control.

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 cannot be used to transfer control in the forward direction, nor to another segment in segmented mode operation.

Flags: No flags affected

Destination Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
RA:	DJNZ R, displacement DBJNZ Rb, displacement		11		11

Example: DJNZ and DBJNZ are typically used to control a "loop" of instructions. In this example for nonsegmented mode, 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        !initalize counter!
        LDA     R1, SRCBUF      !load start address!
        LDA     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:
```

For segmented mode, R1 and R2 must be changed for register pairs.

EI
Enable Interrupts

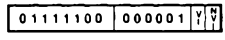
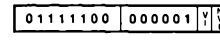
EI int Int: VI, NVI

Operation:

If instruction (0) = 0 then NVI ← 1
If instruction (1) = 0 then VI ← 1

Any combination of the Vectored Interrupt (VI) or Non-Vetored 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

	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
	EI int		7		7

Example: If the NVI contol bit is set (1) in the FCW, and the VI control bit is clear (0), the instruction

EI VI

will leave both the NVI and VI control bits in the FCW set (1)

EX
Exchange

EX dst, src
EXB

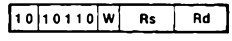
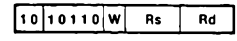
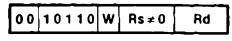
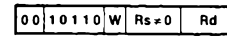
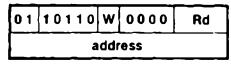
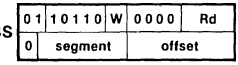
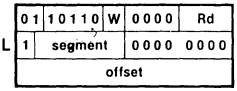
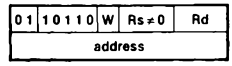
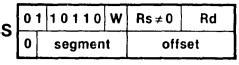
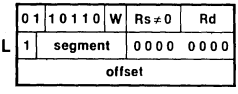
dst: R
src: R, IR, DA, X

Operation:

tmp ← src (tmp is a temporary 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

Source Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
R:	EX Rd, Rs EXB Rbd, Rbs		6		6
IR:	EX Rd, @Rs[1] EXB Rbd, @Rs[1]		12		12
DA:	EX Rd, address EXB Rbd, address		15	SS 	16
				SL 	18
X:	EX Rd, addr(Rs) EXB Rbd, addr(Rs)		16	SS 	16
				SL 	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. The flags will be left unchanged.

Note 1: Word register in nonsegmented mode, register pair in segmented mode.

EXTS
Extend Sign

EXTSB dst
EXTS
EXTSL

dst: R

Operation:

Byte
if dst (7) = 0 then dst (8:15) ← 000...000
else dst (8:15) ← 111...111

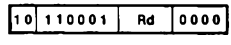
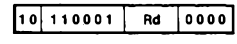
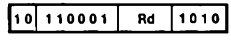
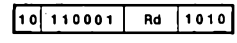
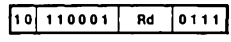
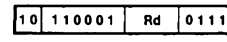
Word
if dst (15) = 0 then dst (16:31) ← 000...000
else dst (16:31) ← 111...111

Long
if dst (31) = 0 then dst (32:63) ← 000...000
else dst (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 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 (as, for example, before a divide).

Flags: No flags affected

Destination Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
R:	EXTSB Rd		11		11
	EXTS RRD		11		11
	EXTSL RQd		11		11

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

Halt

Privileged

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, the instruction following HALT is executed. While halted, memory refresh cycles will still occur, and $\overline{\text{BUSREQ}}$ will be honored.

Flags: No flags affected

	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles[1]	Instruction Format	Cycles[1]
	HALT	01111010 | 00000000	8+3n	01111010 | 00000000	8+3n

Note 1: Interrupts are recognized at the end of each 3-cycle period; thus n = number of periods without interruption.

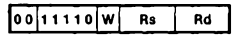
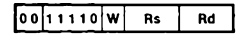
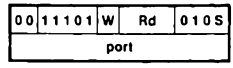
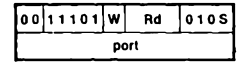
IN (SIN)
(Special) Input

IN dst, src	dst: R
INB	src: IR, DA
SIN dst, src	dst: R
SINB	src: DA

Operation dst ← src

The contents of the source operand, an Input or Special Input port, are loaded into the destination register. IN and INB are used for normal I/O operation; SIN and SINB are used for Special I/O operation.

Flags: No flags affected

Source Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
IR:	IN Rd[1], @Rs INB Rbd[1], @Rs		10		10
DA:	IN Rd, port INB Rbd, port SIN Rd, port SINB Rbd, port		12		12

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

```
INB  RH2, @R6
```

will leave the value %FF in register RH2.

Note 1. Word register in nonsegmented mode; register pair in segmented mode.

INC
Increment

INC dst, src
INCB

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

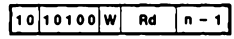
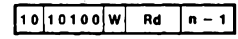
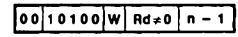
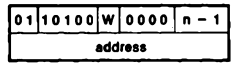
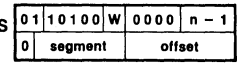
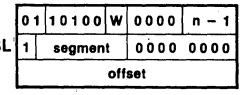
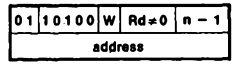
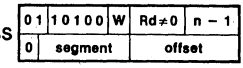
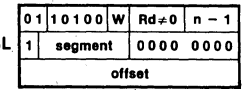
Operation: dst ← dst + src (src = 1 to 16)

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 defaults to the value 1.

The value of the source field in the instruction is one less than the actual value of the source operand. Thus, the coding in the instruction for the source ranges from 0 to 15, which corresponds to the source values 1 to 16.

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, if both operands were of the same sign and the result is of the opposite sign; cleared otherwise
D: Unaffected
H: Unaffected

Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
R:	INC Rd, #n INCB Rbd, #n		4		4
IR:	INC @Rd[1], #n INCB @Rd[1], #n		11		11
DA:	INC address, #n INCB address, #n		13	SS 	14
				SL 	16
X:	INC addr(Rd), #n INCB addr(Rd), #n		14	SS 	14
				SL 	17

Example: If register RH2 contains %21, the statement

```
INCB  RH2,#6
```

will leave the value %27 in RH2.

Note 1: Word register in nonsegmented mode, register pair in segmented mode.

Privileged IND (SIND)
(Special) Input and Decrement

IND dst, src, r
INDB
SIND
SINDB

dst: IR
src: IR

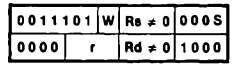
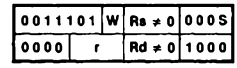
Operation:

dst ← src
AUTODECREMENT dst (by 1 byte, by 2 if word)
r ← r − 1

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

Flags:

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

Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
IR:	IND @Rd[1], @Rs, r INDB @Rd[1], @Rs, r SIND @Rd[1], @Rs, r SINDB @Rd[1], @Rs, r		21		21

Example:

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

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

will leave the value %05B9 in location %02004000, the value %02003FFE in RR4, and the value %0015 in R0. The V flag will be cleared. Register R6 still contains the value %0228. In nonsegmented mode, a word register would be used instead of RR4.

Note 1: Word register in nonsegmented mode, register pair in segmented mode.

INDR (SINDR) Privileged
(Special) Input, Decrement and Repeat

INDR dst, src, r
INDRB
SINDR
SINDRB

dst: IR
src: IR

Operation:

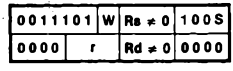
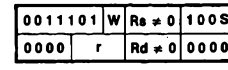
dst ← src
AUTODECREMENT dst (by 1 if byte, by 2 if word)
r ← r − 1
repeat until r = 0

This instruction is used for block input of strings of data. INDR and INDRB are used for normal I/O operation; SINDR and SINDRB are used for special I/O operation. The contents of the I/O port addressed by the source word register are loaded into the memory location addressed by the destination register. I/O port addresses are 16 bits. The destination register is then decremented by one if a byte instruction, or by two if a word instruction, thus moving the pointer to the previous element of the string in memory. The word register specified by "r" (used as a counter) is then decremented by one. The address of the I/O port in the source register is unchanged. 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 or SINDR).

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.

Flags:

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

Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles[2]	Instruction Format	Cycles[2]
IR:	INDR @Rd[1], @Rs, r INDRB @Rd[1], @Rs, r SINDR @Rd[1], @Rs, r SINDRB @Rd[1], @Rs, r		11 + 10n		11 + 10n

Example: If register R1 contains %202A, register R2 contains the Special I/O address %0AFC, and register R3 contains 8, the instruction

```
SINDRB  @R1, @R2, R3
```

will input 8 bytes from the special I/O port 0AFC 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. The V flag will be set. This example assumes nonsegmented mode; in segmented mode, R1 would be replaced by a register pair.

Note 1: Word register in nonsegmented mode, register pair in segmented mode.
Note 2: n = number of data elements transferred.

INI (SINI) Privileged
(Special) Input and Increment

INI dst, src, r
INIB
SINI
SINIB

dst: IR
src: IR

Operation:

dst ← src
AUTOINCREMENT dst (by 1 if byte, by 2 if word)
r ← r − 1

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

Flags:

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

Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
IR:	INI @Rd[1], @Rs, r INIB @Rd[1], @Rs, r SINI @Rd[1], @Rs, r SINIB @Rd[1], @Rs, r	0011101 | W | Rs ≠ 0 | 100S 0000 | r | Rd ≠ 0 | 1000	21	0011101 | W | Rs ≠ 0 | 100S 0000 | r | Rd ≠ 0 | 1000	21

Example:

In nonsegmented mode, if register R4 contains %4000, register R6 contains the I/O port address %0229, the port %0229 contains %B9, and register R0 contains %0016, the instruction

```
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 %0229. The V flag is cleared. In segmented mode, R4 would be replaced by a register pair.

Note 1: Word register in nonsegmented mode, register pair in segmented mode.

INIR (SINIR)

(Special) Input, Increment and Repeat

INIR dst, src, r
INIRB
SINIR
SINIRB

dst: IR
src: IR

Operation:

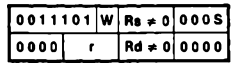
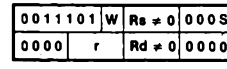
dst ← src
AUTOINCREMENT dst (by 1 if byte, by 2 if word)
r ← r − 1
repeat until r = 0

This instruction is used for block input of strings of data. INIR and INIRB are used for normal I/O operation; SINIR and SINIRB are used for special I/O operation. The contents of the I/O port addressed by the source word register are loaded into the memory location addressed by the destination register. I/O port addresses are 16 bits. The destination register is then incremented by one if a byte instruction, or by two if a word instruction, 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 address of the I/O port in the source register is unchanged. 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 or SINIR).

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.

Flags:

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

Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
IR:	INIR @Rd[1], @Rs, r INIRB @Rd[1], @Rs, r SINIR @Rd[1], @Rs, r SINIRB @Rd[1], @Rs, r		11 + 10n		11 + 10n

INIR (SINIR) Privileged
(Special) Input, Increment and Repeat

Example: In nonsegmented mode, if register R1 contains %2023, register R2 contains the I/O port address %0551, and register R3 contains 8, the statement

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

will input 8 bytes from port %0051 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. The V flag will be set. In segmented mode, a register pair must be used instead of R1.

Note 1: Word register in nonsegmented mode, register pair in segmented mode.
Note 2: n = number of data elements transferred.

IRET
Interrupt Return

IRET

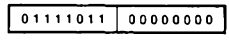
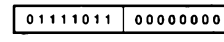
Operation:

Nonsegmented	Segmented
SP ← SP + 2 (Pop "identifier")	SP ← SP + 2 (Pop "identifier")
PS ← @SP	PS ← @SP
SP ← SP + 4	SP ← SP + 6

This instruction is used to return to a previously executed 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 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 to access memory. When using a Z8001, the operation of IRET in nonsegmented mode is undefined. A Z8001 must be in segmented mode when an IRET instruction is performed.

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

Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
	IRET		13		16

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 instruction

```
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 will be determined by the saved FCW value.

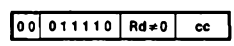
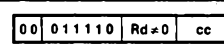
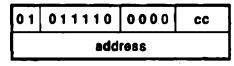
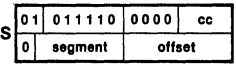
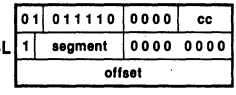
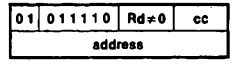
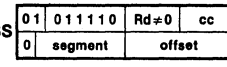
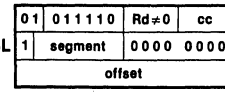
JP
Jump

JP cc, dst dst: IR, DA, X

Operation: If cc is satisfied, then PC ← dst

A conditional jump transfers program control to the destination address if the condition specified by "cc" is satisfied by the flags in the FCW. See list of condition codes. If the condition is satisfied, the program counter (PC) is loaded with the designated address; otherwise, the instruction following the JP instruction is executed.

Flags: No flags affected

Destination Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles[2]	Instruction Format	Cycles[2]
IR:	JP cc, @Rd[1]		10/7		15/7
DA:	JP cc, address		7/7	SS 	8/8
				SL 	10/10
X:	JP cc, addr(Rd)		8/8	SS 	11/11
				SL 	11/11

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.

Note 1: Word register in nonsegmented mode, register pair in segmented mode.
Note 2: The two values correspond to jump taken and jump not taken.

JR
Jump Relative

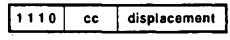
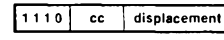
JR cc, dst dst: RA

Operation: if cc is satisfied then PC ← PC + (2 × displacement)

A conditional jump transfers program control to the destination address if the condition specified by "cc" is satisfied by the flags in the FCW. See list of condition codes. If the condition is satisfied, the program counter (PC) is loaded with the designated address; otherwise, the instruction following the JR instruction is executed. The destination address 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 an 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.

Flags: No flags affected

Destination Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
RA:	JR cc, address		6		6

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

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

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

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
Load

LD dst, src
LDB
LDL

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

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

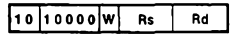
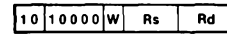
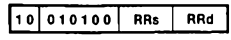
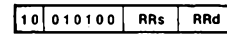
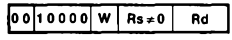
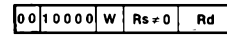
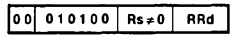
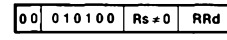
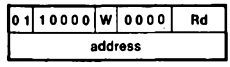
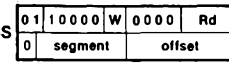
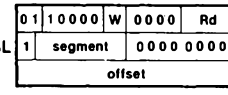
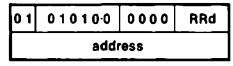
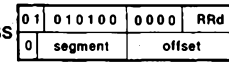
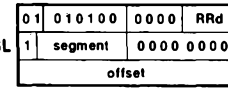
Operation: dst ← src

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

There are three versions of the Load instruction: Load into a register, load into memory and load an immediate value.

Flags: No flags affected

Load Register

Source Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
R:	LD Rd, Rs LDB Rbd, Rbs		3		3
	LDL RRd, RRs		5		5
IR:	LD Rd, @Rs[1] LDB Rbd, @Rs[1]		7		7
	LDL RRd, @Rs[1]		11		11
DA:	LD Rd, address LDB Rbd, address		9	SS 	10
				SL 	12
	LDL RRd, address		12	SS 	13
				SL 	15

Note 1: Word register in nonsegmented mode, register pair in segmented mode.

Load Register (Continued)

Source Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
X:	LD Rd, addr(Rs) LDB Rbd, addr(Rs)	01 | 10000 | W | Rs≠0 | Rd address	10	SS: 01 | 10000 | W | Rs≠0 | Rd 0 | segment | offset	10
				SL: 01 | 10000 | W | Rs≠0 | Rd 1 | segment | 0000 0000 offset	13
	LDL RRd, addr(Rs)	01 | 010100 | Rs≠0 | RRd address	13	SS: 01 | 010100 | Rs≠0 | RRd 0 | segment | offset	13
				SL: 01 | 010100 | Rs≠0 | RRd 1 | segment | 0000 0000 offset	16
BA:	LD Rd, Rs[1](#disp) LDB Rbd, Rs[1](#disp)	00 | 11000 | W | Rs≠0 | Rd displacement	14	00 | 11000 | W | Rs≠0 | Rd displacement	14
	LDL RRd, Rs[1](#disp)	00 | 110101 | Rs≠0 | Rd displacement	17	00 | 110101 | Rs≠0 | Rd displacement	17
BX:	LD Rd, Rs[1](Rx) LDB Rd, Rs[1](Rx)	01 | 11000 | W | Rs≠0 | Rd 0000 | Rx | 0000 0000	14	01 | 11000 | W | Rs≠0 | Rd 0000 | Rx | 0000 0000	14
	LDL RRd, Rs[1](Rx)	01 | 11010 | 1 | Rs≠0 | Rd 0000 | Rx | 0000 0000	17	01 | 11010 | 1 | Rs≠0 | Rd 0000 | Rx | 0000 0000	17

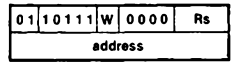
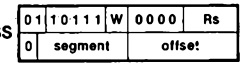
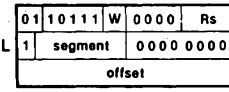
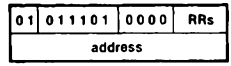
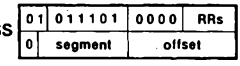
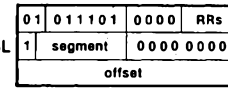
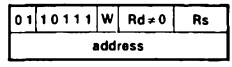
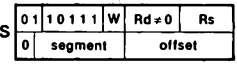
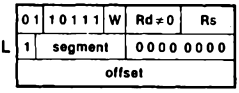
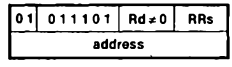
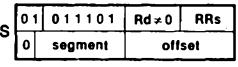
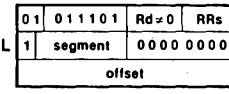
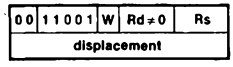
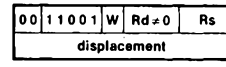
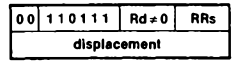
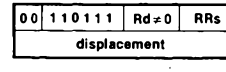
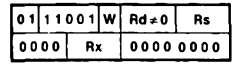
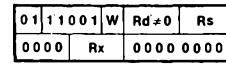
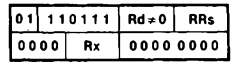
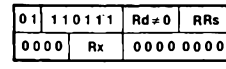
Load Memory

Destination Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
IR:	LD @Rd[1], Rs LDB @Rd[1], Rbs	00 | 10111 | W | Rd≠0 | Rs	8	00 | 10111 | W | Rd≠0 | Rs	8
	LDL @Rd[1], RRs	00 | 011101 | Rd≠0 | RRs	11	00 | 011101 | Rd≠0 | RRs	11

Note 1: Word register in nonsegmented mode, register pair in segmented mode.

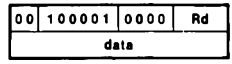
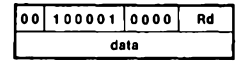
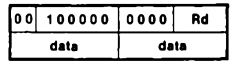
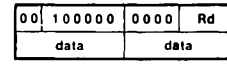
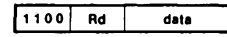
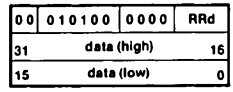
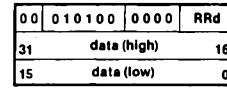
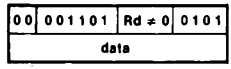
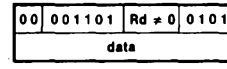
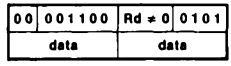
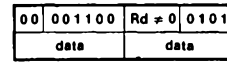
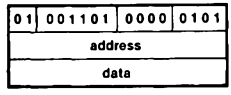
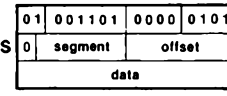
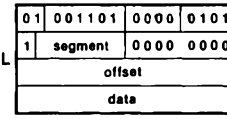
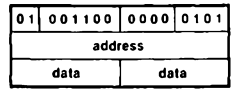
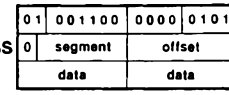
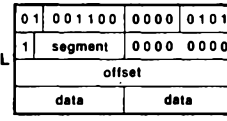
LD
Load

Load Memory (Continued)

Destination Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
DA:	LD address, Rs LDB address, Rbs		11	SS 	12
				SL 	14
DA:	LDL address, RRs		14	SS 	15
				SL 	17
X:	LD addr(Rd), Rs LDB addr(Rd), Rbs		12	SS 	12
				SL 	15
	LDR addr(Rd), RRs		15	SS 	15
				SL 	18
BA:	LD Rd[1](#disp), Rs LDB Rd[1](#disp), Rbs		14		14
	LDL Rd[1](#disp), RRs		17		17
BX:	LD Rd[1](Rx), Rs LDB Rd[1](Rx), Rbs		14		14
	LDL Rd[1](Rx), RRs		17		17

Note 1: Word register in nonsegmented mode, register pair in segmented mode.

Load Immediate Value

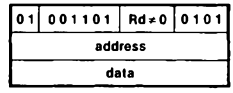
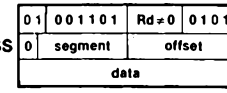
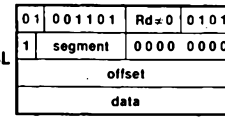
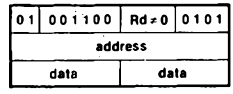
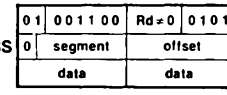
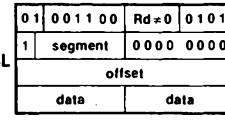
Destination Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
R:	LD Rd, #data		7		7
	LDB Rbd, #data[2]		7		7
			5		5
	LDL RRd, #data		11		11
IR:	LD @Rd[1], #data		11		11
	LDB @Rd[1], #data		11		11
DA:	LD address, #data		14	SS 	15
				SL 	17
	LDB address, #data		14	SS 	15
				SL 	17

Note 1: Word register in nonsegmented mode, register pair in segmented mode.

Note 2: Although two formats exist for "LDB R, IM", the assembler always uses the short format. In this case, the "src field" in the instruction format encoding contains the source operand.

LD
Load

Load Immediate Value (Continued)

Destination Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
X:	LD addr(Rd), #data		15	SS 	15
				SL 	18
	LDB addr(Rd), #data		15	SS 	15
				SL 	18

Example: Several examples of the use of the Load instruction are treated in detail in Chapter 4 under addressing modes.

LDA
Load Address

LDA dst, src

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

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 address arithmetic. The destination is a word register in nonsegmented mode, and a register pair in segmented mode.

In segmented mode, the address loaded into the destination has an undefined value in all reserved bits (bits 16-23 and bit 31). However, this address may be used by subsequent instructions in the indirect based or base-index addressing modes without any modification to the reserved bits.

Flags: No flags affected

Source Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
DA:	LDA Rdˡ, address	01 | 110110 | 0000 | Rd address	12	SS: 01 | 110110 | 0000 | RRd 0 | segment | offset	13
				SL: 01 | 110110 | 0000 | RRd 1 | segment | 0000 0000 offset	15
X:	LDA Rdˡ, addr(Rs)	01 | 110110 | Rs≠0 | Rd address	13	SS: 01 | 110110 | Rs≠0 | RRd 0 | segment | offset	13
				SL: 01 | 110110 | Rs≠0 | RRd 1 | segment | 0000 0000 offset	16
BA:	LDA Rdˡ, Rsˡ (#disp)	00110100 | Rs≠0 | Rd displacement	15	00110100 | Rs≠0 | Rd displacement	15
BX:	LDA Rdˡ, Rsˡ (Rx)	01110100 | Rs≠0 | Rd 0000 | Rx | 0000 0000	15	01110100 | Rs≠0 | Rd 0000 | Rx | 0000 0000	15

LDA
Load Address

Examples:

```
LDA   R4,STRUCT          !in nonsegmented mode, register R4 is loaded!
                         !with the nonsegmented address of the location!
                         !named STRUCT!
LDA   RR2, <<3>> 8(R4)   !in segmented mode, if index register R4!
                         !contains %20, then register RR2 is loaded!
                         !with the segmented address (<<3>>, offset %28)!
LDA   RR2,RR4(#8)        !in segmented mode, if base register RR4!
                         !contains %01000020, then register RR2 is loaded!
                         !with the segment address << 1 >> %28!
                         !(segment 1, offset %28)!
```

Note 1: Word register in nonsegmented mode, register pair in segmented mode.

LDAR
Load Address Relative

LDAR dst, src

dst: R
src: 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. In segmented mode, the address loaded into the destination has all "reserved" bits (bits 16-23 and bit 31) cleared to zero.

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 addition is performed following the rules of address arithmetic, with no modifications to the segment number in segmented mode. Thus in segmented mode, the source operand must be in the same segment as the LDAR instruction.

The assembler automatically calculates the displacement by subtracting the PC value of the following instruction from the address given by the programmer.

Flags: No flags affected

Source Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
RA:	LDAR Rd[1], address	00110100 | 0000 | Rd displacement	15	00110100 | 0000 | Rd displacement	15

Example:

```
LDAR    R2, TABLE       !in nonsegmented mode, register R2 is loaded!
                        !with the address of TABLE!
LDAR    RR4, TABLE      !in segmented mode, register pair RR4 is!
                        !loaded with the segmented address of TABLE,!
                        !which must be in the same segment as the program!
```

Note 1: Word register in nonsegmented mode, register pair in segmented mode.

LDCTL
Load Control

Privileged

LDCTL dst, src

dst: CTLR
src: R
or
dst: R
src: CTLR

Operation: dst ← src

This instruction loads the contents of a general purpose register into a control register, or loads the contents of a control register into a general-purpose register. The control register may be one of the following CPU registers:

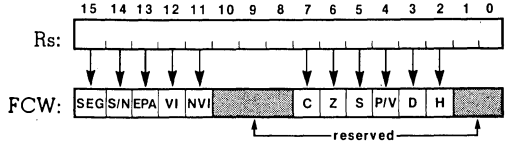
FCW	Flag and Control Word
REFRESH	Refresh Control
PSAPSEG	Program Status Area Pointer - segment number
PSAPOFF	Program Status Area Pointer - offset
NSPSEG	Normal Stack Pointer - segment number
NSPOFF	Normal Stack Pointer - offset

The operation of each of the variants of the instruction is detailed below. The ones which load data into a control register are described first, followed by the variants which load data from a control register into a general purpose register. Whenever bits are marked reserved, the corresponding bit in the source register must be either 0 or the value returned by a previous load from the same control register. For compatibility with future CPUs, programs should not assume that memory copies of control registers contain 0s, nor should they store data in reserved fields of memory copies of control registers.

Load Into Control Register

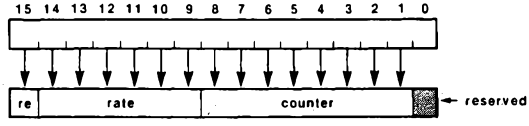
LDCTL FCW, Rs

Operation: FCW (2:7) ← Rs (2:7)
FCW (11:15) ← Rs (11:15)



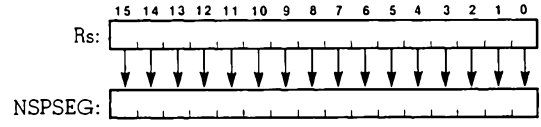
LDCTL REFRESH, Rs

Operation: REFRESH (1:15) ← Rs (1:15)
Rs:
REFRESH:



LDCTL NSPSEG, Rs

Operation: NSPSEG (0:15) ← Rs (0:15)

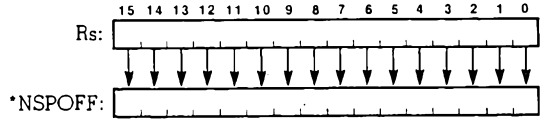


In segmented mode, the NSPSEG register is the normal mode R14 and contains the segment number of the normal mode processor stack pointer which is otherwise inaccessible for system mode.

In nonsegmented mode, R14 is not used as part of the normal processor stack pointer. This instruction may not be used in nonsegmented mode.

LDCTL NSPOFF, Rs
NSP, Rs

Operation: NSPOFF (0:15) ← Rs (0:15)



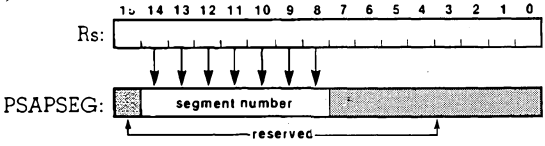
***NSP in nonsegmented mode**

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.

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

LDCTL PSAPSEG, Rs

Operation: PSAPSEG (8:14) ← Rs (8:14)



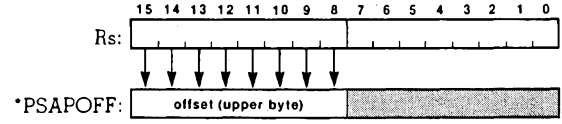
The PSAPSEG register may not be used in the nonsegmented Z8002. In the segmented Z8001, care must be exercised when changing the two PSAP register values so that an interrupt occurring between the changing of PSAPSEG and PSAPOFF is handled correctly. This is typically accomplished by first disabling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

LDCTL
Load Control

Privileged

LDCTL PSAPOFF, Rs
PSAP, Rs

Operation: PSAPOFF (8:15) ← Rs (8:15)



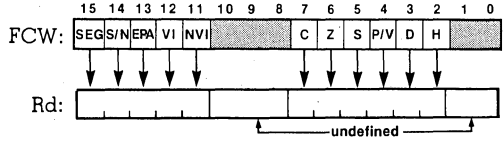
***PSAP in nonsegmented mode**

In the nonsegmented Z8002, 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, 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. This is typically accomplished by first disabling interrupts before changing PSAPSEG and PSAPOFF. The low order byte of PSAPOFF should be 0.

Load From Control Register

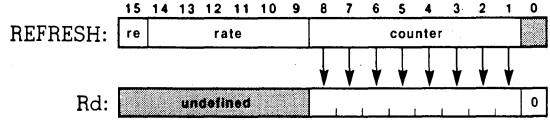
LDCTL Rd, FCW

Operation:
Rd (2:7) ← FCW (2:7)
Rd (11:15) ← FCW (11:15) (Z8001 only)
Rd (11:14) ← FCW (11:14) (Z8002 only)
Rd (0:1) ← UNDEFINED
Rd (8:10) ← UNDEFINED
Rd (15) ← 0 (Z8002 only)



LDCTL Rd, REFRESH

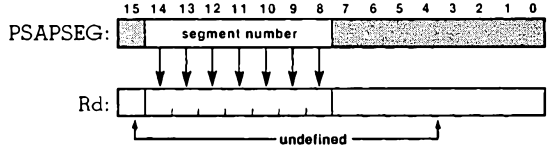
Operation:
Rd (1:8) ← REFRESH (1:8)
Rd (0) ← UNDEFINED
Rd (9:15) ← UNDEFINED



LDCTL Rd, PSAPSEG

Operation:

Rd (8:14) ← PSAPSEG (8:14)
Rd (0:7) ← UNDEFINED
Rd (15) ← UNDEFINED

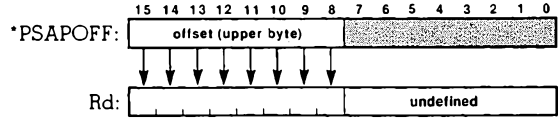


This instruction may not be used in the nonsegmented version.

LDCTL Rd, PSAPOFF
Rd, PSAP

Operation:

Rd (8:15) ← PSAPOFF (8:15)
Rd (0:7) ← UNDEFINED



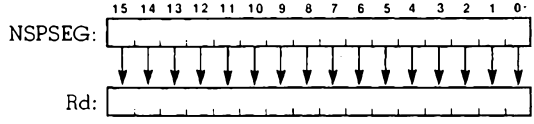
***PSAP in nonsegmented mode**

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

LDCTL Rd, NSPSEG

Operation:

Rd (0:15) ← NSPSEG (0:15)



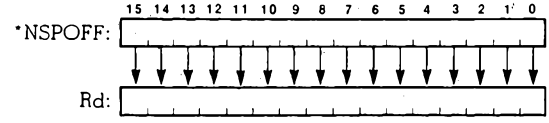
This instruction is not available in nonsegmented mode.

LDCTL
Load Control

Privileged

LDCTL Rd, NSPOFF
Rd, NSP

Operation: Rd (0:15) ← NSPOFF (0:15)



***NSP in nonsegmented mode**

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

Flags: No flags affected, except when the destination is the Flag and Control Word (LDCTL FCW, Rs), in which case all the flags are loaded from the source register.

Source Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
	LDCTL FCW, Rs	01111101 | Rs | 1010	7	01111101 | Rs | 1010	7
	LDCTL REFRESH, Rs	01111101 | Rs | 1011	7	01111101 | Rs | 1011	7
	LDCTL PSAPSEG, Rs			01111101 | Rs | 1100	7
	LDCTL PSAPOFF, Rs PSAP, Rs	01111101 | Rs | 1101	7	01111101 | Rs | 1101	7
	LDCTL NSPSEG, Rs			01111101 | Rs | 1110	7
	LDCTL NSPOFF, Rs NSP, Rs	01111101 | Rs | 1111	7	01111101 | Rs | 1111	7

Destination Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
	LDCTL Rd, FCW	01111101 | Rd | 0010	7	01111101 | Rd | 0010	7
	LDCTL Rd, REFRESH	01111101 | Rd | 0011	7	01111101 | Rd | 0011	7
	LDCTL Rd, PSAPSEG			01111101 | Rd | 0100	7
	LDCTL Rd, PSAPOFF LDCTL Rd, PSAP LDCTL Rd, NSPSEG	01111101 | Rd | 0101	7	01111101 | Rd | 0101	7
				01111101 | Rd | 0110	7
	LDCTL Rd, NSPOFF Rd, NSP	01111101 | Rd | 0111	7	01111101 | Rd | 0111	7

LDCTLB
Load Control Byte

LDCTLB dst, src

dst: FLAGS
src: R
or
dst: R
src: FLAGS

Operation: dst ← src

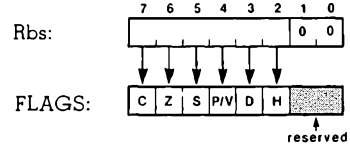
This instruction is used to load the FLAGS register or to transfer its contents into a general-purpose register. Note that this is not a privileged instruction.

Load Into FLAGS Register

LDCTLB FLAGS, Rbs

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.

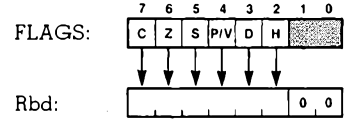


Load From FLAGS Register

LDCTLB Rbd, FLAGS

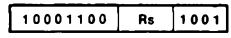
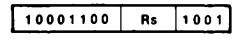
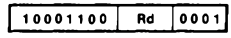
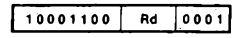
dst (2:7) ← FLAGS (2:7)
dst (0:1) ← 0

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.



Flags: When the FLAGS register is the destination, all the flags are loaded from the source. When the FLAGS register is the source, none of the flags are affected.

LDCTLB
Load Control Byte

	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
	LDCTLB FLAGS, Rbs		7		7
	LDCTLB Rbd, FLAGS		7		7

LDD
Load and Decrement

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

Operation:

dst ← src
AUTODECREMENT dst and src (by 1 if byte, by 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 source destination, and counter registers must be separate and non-overlapping registers. 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

Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
IR:	LDD @Rs[1], @Rd[1], r LDDB @Rs[1], @Rd[1], r	1011101 | W | Rs ≠ 0 | 1001 0000 | r | Rd ≠ 0 | 1000	20	1011101 | W | Rs ≠ 0 | 1001 0000 | r | Rd ≠ 0 | 1000	20

Example:

In nonsegmented mode, if register R1 contains %202A, register R2 contains %404A, the word at location %404A contains %FFFF, and register R3 contains 5, the instruction

LDD @R1, @R2, R3

will leave the value %FFFF at location %202A, the value %2028 in R1, the value %4048 in R2, and the value 4 in R3. The V flag will be cleared. In segmented mode, register pairs would be used instead of R1 and R2.

Note 1: Word register in nonsegmented mode, register pair in segmented mode.

LDDR
Load, Decrement and Repeat

LDDR dst, src, r — dst: IR
LDDRB — src: IR

Operation:

dst ← src
AUTODECREMENT dst and src (by 1 if byte, by 2 if word)
r ← r − 1
repeat until r = 0

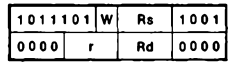
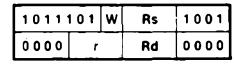
This instruction is used for block transfers of strings of data. The contents of the location addressed by the source register are loaded into the location addressed by the destination register. The source and destination registers are then decremented by one if LDDRB, or by two if LDDR, thus moving 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. The source, destination, and counter registers must be separate and non-overlapping registers. This instruction can transfer from 1 to 65536 bytes or from 1 to 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.

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 cycles should be added to this instruction's execution time for each interrupt request that is accepted.

Flags:

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

Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles[2]	Instruction Format	Cycles[2]
IR:	LDDR @Rd[1], @Rs[1], r LDDRB @Rd[1], @Rs[1], r		11 + 9n		11 + 9n

Example: In nonsegmented mode, if register R1 contains %202A, register R2 contains %404A, the words at locations %4040 through %404A all contain %FFFF, and register R3 contains 6, the instruction

```
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. The V flag will be set. In segmented mode, register pairs would be used instead of R1 and R2.

Note 1: Word register in nonsegmented mode, register pair in segmented mode.
Note 2: n = number of data elements transferred.

LDI
Load and Increment

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

Operation:

dst ← src
AUTOINCREMENT dst and src (by 1 if byte, by 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 source, destination, and counter registers must be separate and non-overlapping registers. 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

Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
IR:	LDI @Rd[1], @Rs[1], r LDIB @Rd[1], @Rs[1], r	1011101 W Rs ≠ 0 0001 0000 r Rd ≠ 0 1000	20	1011101 W Rs ≠ 0 0001 0000 r Rd ≠ 0 1000	20

Example:

This instruction can be 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. This example assumes nonsegmented mode. In segmented mode, register pairs would be used instead of R1 and R2.

```
        LD      R3, #80          !initialize counter!
        LDA     R1, DSTBUF       !load start addresses!
        LDA     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:
```

Note 1: Word register in nonsegmented mode, register pair in segmented mode.

LDIR
Load, Increment and Repeat

LDIR dst, src, r
LDIRB

dst: IR
src: IR

Operation:

dst ← src
AUTOINCREMENT dst and src (by 1 if byte; by two if word)
r ← r − 1
repeat until R = 0

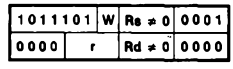
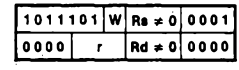
This instruction is used for block transfers of strings of data. The contents of the location addressed by the source register are loaded into the location addressed by the destination register. The source and destination registers are then incremented by one if LDIRB, or by two if LDIR, thus moving 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. The source, destination, and counter registers must be separate and non-overlapping registers. This instruction can transfer from 1 to 65536 bytes or from 1 to 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.

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 cycles should be added to this instruction's execution time for each interrupt request that is accepted.

Flags:

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

Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles[2]	Instruction Format	Cycles[2]
IR:	LDIR @Rd[1], @Rs[1], r LDIRB @Rd[1], @Rs[1], r		11 + 9n		11 + 9n

LDIR
Load, Increment and Repeat

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

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

In segmented mode, R1 and R2 must be replaced by register pairs.

Note 1: Word register in nonsegmented mode, register pair in segmented mode.
Note 2: n = number of data elements transferred.

LDK
Load Constant

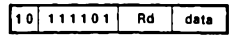
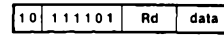
LDK dst, src

dst: R
src: IM

Operation: dst ← src (src = 0 to 15)

The source operand (a constant value specified in the src field) is loaded into the destination register. The source operand is a value from 0 to 15. It is loaded into the four low-order bits of the destination register, while the high-order 12 bits are cleared to zero.

Flags: No flags affected

Destination Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
R:	LDK Rd, #data		5		5

Example: To load register R3 with the constant 9:

```
LDK  R3,#9
```

LDM
Load Multiple

LDM dst, src, n

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

Operation: dst ← src(n words)

The contents of n source words are loaded into the destination. The contents of the source are not affected. The value of n lies between 1 and 16, inclusive. This instruction moves information between memory and registers; registers are accessed in increasing order starting with the specified register; R0 follows R15. The instruction can be used either to load multiple registers into memory (e.g. to save the contents of registers upon subroutine entry) or to load multiple registers from memory (e.g. to restore the contents of registers upon subroutine exit).

The instruction encoding contains values from 0 to 15 in the "num" field corresponding to values of 1 to 16 for n, the number of registers to be loaded or saved.

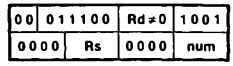
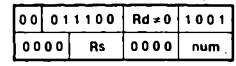
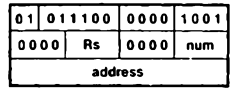
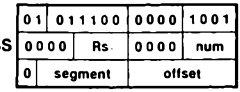
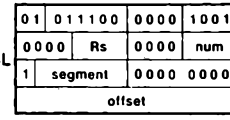
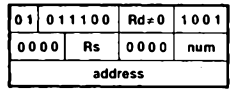
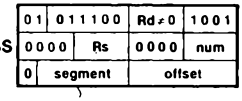
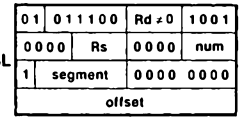
The starting address is computed once at the start of execution, and incremented by two for each register loaded. If the original address computation involved a register, the register's value will not be affected by the address incrementation during execution. Similarly, modifying that register during a load from memory will not affect the address used by this instruction.

Flags: No flags affected

Load Multiple - Registers From Memory

Source Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
IR:	LDM Rd, @Rs[1], #n	00 011100 Rs≠0 0001 0000 Rd 0000 num	11+3n	00 011100 Rs≠0 0001 0000 Rd 0000 num	11+3n
DA:	LDM Rd, address, #n	01 011100 0000 0001 0000 Rd 0000 num address	14+3n	SS: 01 011100 0000 0001 0000 Rd 0000 num 0 segment offset	15+3n
				SL: 01 011100 0000 0001 0000 Rd 0000 num 1 segment 0000 0000 offset	17+3n
X:	LDM Rd, addr(Rs), #n	01 011100 Rs≠0 0001 0000 Rd 0000 num address	15+3n	SS: 01 011100 Rs≠0 0001 0000 Rd 0000 num 0 segment offset	15+3n
				SL: 01 011100 Rs≠0 0001 0000 Rd 0000 num 1 segment 0000 0000 offset	18+3n

Load Multiple - Memory From Registers

Destination Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
IR:	LDM@Rd[1], Rs, #n		11 + 3n		11 + 3n
DA:	LDM address, Rs, #n		14 + 3n	SS 	15 + 3n
				SL 	17 + 3n
X:	LDM addr(Rd), Rs, #n		15 + 3n	SS 	15 + 3n
				SL 	18 + 3n

Example: In nonsegmented mode, 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. In segmented mode, a register pair would be used instead of R6.

Note 1: Word register in nonsegmented mode, register pair in segmented mode.
Note 2: n = number of registers.

LDPS — Privileged

Load Program Status

LDPS src — src: IR, DA, X

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)

Flags: All flags are loaded from the source operand.

Source Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
IR:	LDPS @Rs!	00 | 111001 | Rs≠0 | 0000	12	00 | 111001 | Rs≠0 | 0000	16
DA:	LDPS address	01 | 111001 | 0000 | 0000 address	16	SS: 01 | 111001 | 0000 | 0000 0 | segment | offset	20
				SL: 01 | 111001 | 0000 | 0000 1 | segment | 0000 0000 offset	22
X:	LDPS addr(Rs)	01 | 111001 | Rs≠0 | 0000 address	17	SS: 01 | 111001 | Rs≠0 | 0000 0 | segment | offset	20
				SL: 01 | 111001 | Rs≠0 | 0000 1 | segment | 0000 0000 offset	23

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 instruction

```
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.) In the segmented mode, a register pair is used instead of R3. Note: Word register is used in nonsegmented mode, register pair in segmented mode.

LDR
Load Relative

LDR dst, src **LDRB** **LDRL**	dst: R src: RA or dst: RA src: R

Operation: dst ← src

The contents of the source operand are loaded into the destination. The contents of the source are not affected. The relative address is calculated by adding the displacement in the instruction to the updated value of the program counter (PC) to derive the operand's address. In segmented mode, the segmented number of the computed address is the same as the segment number of the PC. 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.

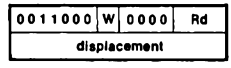
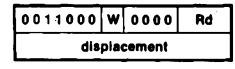
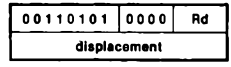
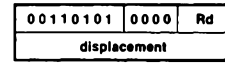
Status pin information during the access to memory for the data operand will be Program Reference, (1100) instead of Data Memory request (1000).

The assembler automatically calculates the displacement by subtracting the PC value of the following instruction from the address given by the programmer.

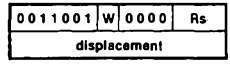
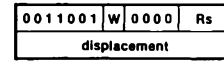
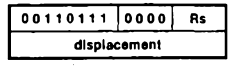
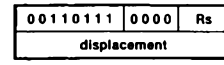
This instruction must be used to modify memory locations containing program information, such as the Program Status Area, if program and data space are allocated to different segments.

Flags: No flags affected

Load Relative Register

Source Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
RA:	LDR Rd, address LDRB Rbd, address		14		14
	LDRL RRd, address		17		17

Load Relative Memory

Destination Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
RA:	LDR address, Rs LDRB address, Rbs		14		14
	LDRL address, RRs		17		17

Example: LDR R2, DATA !register R2 is loaded with the value in the!
!location named DATA!

MBIT — Privileged

Multi-Micro Bit Test

MBIT

Operation: S ← 1 if $\overline{MI}$ high (inactive); 0 otherwise

This instruction is used to synchronize multiple processors' exclusive access to shared hardware resources. The multi-micro input pin ($\overline{MI}$) is tested, and the S flag is cleared if the pin is low (active); otherwise, the S flag is set, indicating that the pin is high (inactive).

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 $\overline{MI}$ is high; cleared otherwise
V: Unaffected
D: Unaffected
H: Unaffected

	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
	MBIT	0111101100001010	7	0111101100001010	7

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

Operation:

```
Z ← 0
if MI low (active)  then S ← 0
                         MO forced high (inactive)
                    else MO forced low (active)
                         repeat dst ← dst − 1 until dst = 0
                         if MI low (active)  then S ← 1
                                             else S ← 0
                                                  MO forced high (inactive)
                         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 ($\overline{MI}$ and $\overline{MO}$), 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 $\overline{MI}$ pin is tested. If the $\overline{MI}$ pin is low (active), the S flag is cleared and the $\overline{MO}$ pin is forced high (inactive),thus indicating that the resource is not available and removing any previous request by the CPU from the $\overline{MO}$ line.

If the $\overline{MI}$ pin is high (inactive), indicating that the resource may be available, a sequence of machine operations occurs. First, the $\overline{MO}$ pin is forced low (active), 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 $\overline{MI}$ pin is tested to determine whether the request for the resource was acknowledged. If the $\overline{MI}$ pin is low (active), the S flag is set to one, indicating that the resource is available and access is granted. If the $\overline{MI}$ pin is still high (inactive), the S flag is cleared to zero, and the $\overline{MO}$ pin is forced high (inactive), indicating that the request was not granted and removing the request signal for the $\overline{MO}$. Finally, in either case, the Z flag is set to one, indicating that the original test of the $\overline{MI}$ pin caused a request to be made.

S flag	Z flag	$\overline{MO}$	Indicates
0	0	high	Request not signalled (resource not available)
0	1	high	Request not granted (resource not available)
1	1	low	Request granted (resource available)

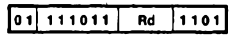
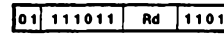
Flags:

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

MREQ

Multi-Micro Request

Privileged

Destination Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles[1]	Instruction Format	Cycles[1]
R:	MREQ Rd		12+7n		12+7n

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!
```

Note 1: If the request is made, n = number of times the destination is decremented. If the request is not made, n = 0.

MRES
Multi-Micro Reset

MRES

Operation: $\overline{MO}$ is forced high (inactive)

This instruction is used to synchronize multiple processors' exclusive access to shared hardware resources. The multi-micro output pin $\overline{MO}$ is forced high (inactive). Forcing $\overline{MO}$ high (inactive) indicates that a resource controlled by the CPU is available for use by other processors.

Flags: No flags affected.

	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
	MRES	01111011 | 00001001	5	01111011 | 00001001	5

Example:
```
MRES        !signal that resource controlled by this CPU!
            !is available to other processors!
```

MSET

Multi-Micro Set

Privileged

MSET

Operation: $\overline{MO}$ is forced low (active)

This instruction is used to synchronize multiple processors' exclusive access to shared hardware resources. The multi-micro output pin $\overline{MO}$ is forced low (active). Forcing $\overline{MO}$ low (active) 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.

	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
	MSET	01111011 | 00001000	5	01111011 | 00001000	5

Example:

```
MSET            !CPU controlled resource not available!
```

MULT
Multiply

MULT dst, src
MULTL

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

Operation:

Word
dst (0:31) ← dst (0:15) × src (0:15)
Long
dst (0:63) ← dst (0:31) × src (0:31)

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.

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.

The initial contents of the high-order half of the destination register do not affect the operation of this instruction and are overwritten by the result. The carry flag is set to indicate that the upper half of the destination register is required to represent the result; if the carry flag is clear, the product can be correctly represented in the same precision as the multiplicand and the upper half of the destination merely holds a sign extension.

The following table gives execution times for word and long word operands in each possible addressing mode.

src	Word			Long Word		
	NS	SS	SL	NS	SS	SL
R	70	70	70	282+7*n	282+7*n	282+7*n
IM	70	70	70	282+7*n	282+7*n	282+7*n
IR	70	70	70	282+7*n	282+7*n	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 table 32 bits of the destination operand)

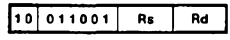
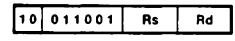
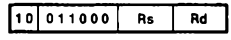
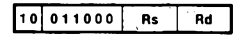
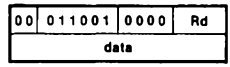
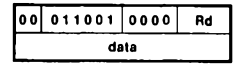
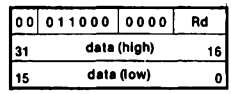
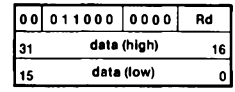
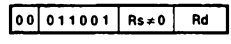
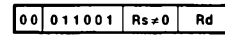
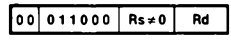
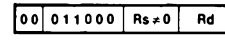
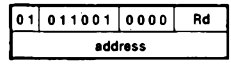
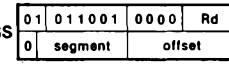
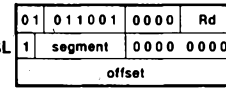
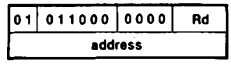
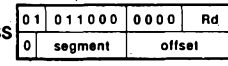
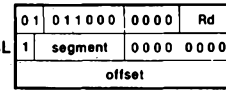
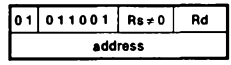
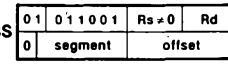
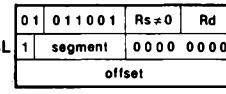
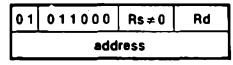
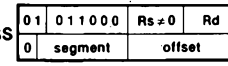
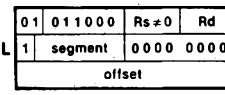
When the multiplier is zero, the execution time of Multiply is reduced to the following times:

src	Word			Long Word		
	NS	SS	SL	NS	SS	SL
R	18	18	18	30	30	30
IM	18	18	18	30	30	30
IR	18	18	18	30	30	30
DA	19	20	22	31	32	34
X	20	20	23	32	32	35

Flags:

C: MULT—set if product is less than -2^{31} 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

MULT
Multiply

Source Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles[2]	Instruction Format	Cycles[2]
R:	MULT RRd, Rs				
	MULTL RQd, RRs				
IM:	MULT RRd, #data				
	MULTL RQd, #data				
IR:	MULT RRd, @Rs[1]				
	MULTL RQd, @Rs[1]				
DA:	MULT RRd, address			SS  SL 	
	MULTL RQd, address			SS  SL 	
X:	MULT RRd, addr(Rs)			SS  SL 	
	MULTL RQd, addr(Rs)			SS  SL 	

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.

Note 1: Word register in nonsegmented mode, register pair in segmented mode.
Note 2: Execution times for each instruction are given in the preceding tables.

NEG
Negate

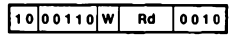
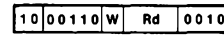
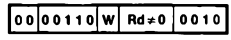
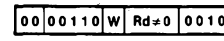
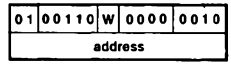
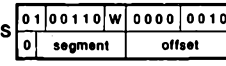
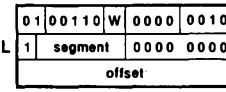
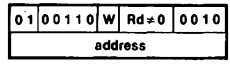
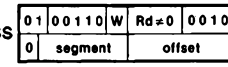
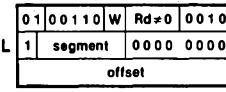
NEG dst
NEGB

dst: R, IR, DA, X

Operation: dst ← dst

The contents of the destination are 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 negative number with greatest magnitude has no positive counterpart; for these two cases, the V flag is set.

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

Destination Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
R:	NEG Rd NEGB Rbd		7		7
IR:	NEG @Rd[1] NEGB @Rd[1]		12		12
DA:	NEG address NEGB address		15	SS 	16
				SL 	18
X:	NEG addr(Rd) NEGB addr(Rd)		16	SS 	16
				SL 	19

Example: If register R8 contains %051F, the statement

```
NEG  R8
```

will leave the value %FAE1 in R8.

Note 1: Word register in nonsegmented mode, register pair in segmented mode.

NOP
No Operation

NOP

Operation: No operation is performed.

Flags: No flags affected

	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
	NOP	10001101 | 00000111	7	10001101 | 00000111	7

OR
Or

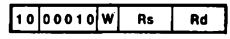
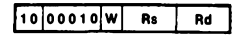
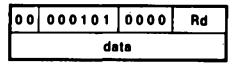
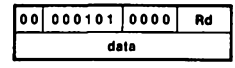
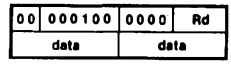
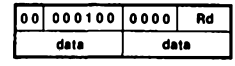
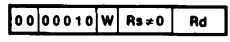
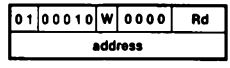
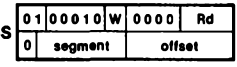
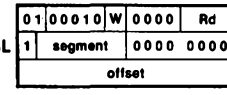
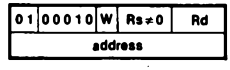
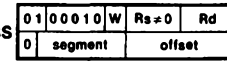
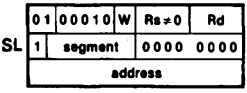
OR dst, src
ORB

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

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

Source Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
R:	OR Rd, Rs ORB Rbd, Rbs		4		4
IM:	OR Rd, #data		7		7
	ORB Rbd, #data		7		7
IR:	OR Rd, @Rs! ORB Rbd, @Rs!		7		7
DA:	OR Rd, address ORB Rbd, address		9	SS 	10
				SL 	12
X:	OR Rd, addr(Rs) ORB Rbd, addr(Rs)		10	SS 	10
				SL 	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.

Note 1: Word register in nonsegmented mode, register pair in segmented mode.

OTDR (SOTDR) Privileged
(Special), Output, Decrement and Repeat

OTDR dst, src, r
OTDRB
SOTDR
SOTDRB

dst: IR
src: IR

Operation:

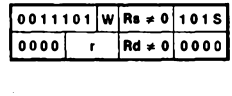
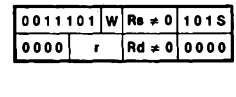
dst ← src
AUTODECREMENT src (by 1 if byte, by 2 if word)
r ← r - 1
repeat until r = 0

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

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.

Flags:

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

Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles[2]	Instruction Format	Cycles[2]
IR:	OTDR @Rd,@Rs[1], r OTDRB @Rd,@Rs[1], r SOTDR @Rd,@Rs[1], r SOTDRB @Rd,@Rs[1], r		11 + 10n		11 + 10n

Example: In nonsegmented mode, if register R11 contains %0FFF, register R12 contains %B006, and R13 contains 6, the instruction

```
OTDR  @R11, @R12, R13
```

will output the string of words from locations %B006 to %AFFC (in descending order of address) to port %0FFF. R12 will contain %AFFA, and R13 will contain 0. R11 will not be affected. The V flag will be set. In segmented mode, R12 would be replaced by a register pair.

Note 1: Word register in nonsegmented mode, register pair in segmented mode.
Note 2: n = number of data elements transferred.

OTIR (SOTIR) Privileged
(Special) Output, Increment and Repeat

OTIR dst, src, r
OTIRB
SOTIR
SOTIRB

dst: IR
src: IR

Operation:

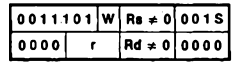
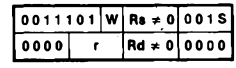
dst ← src
AUTOINCREMENT src (by 1 if byte, by 2 if word)
r ← r − 1
repeat until r = 0

This instruction is used for block output of strings of data. OTIR and OTIRB are used for normal I/O operation; SOTIR and SOTIRB are used for special I/O operation. The contents of the memory location addressed by the source register are loaded into the I/O port addressed by the destination word register. I/O port addresses are 16 bits. The source register is then incremented by one if a byte instruction, or by two if a word instruction, thus moving the pointer to the next element of the string in memory. The word register specified by "r" (used as a counter) is then decremented by one. The address of I/O port in the destination register is unchanged. 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 or SOTIR).

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.

Flags:

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

Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
IR:	OTIR @Rd, @Rs[l], r OTIRB @Rd, @Rs[l], r SOTIR @Rd, @Rs[l], r SOTIRB @Rd, @Rs[l], r		11 + 10n		11 + 10n

Example: In nonsegmented mode, 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
LDA     R2, SRCBUF
LD      R3, #LENGTH
OTIRB   @R1, @R2, R3
```

In segmented mode, a register pair would be used instead of R2.

Note 1: Word register in nonsegmented mode, register pair in segmented mode.
Note 2: n = number of data elements transferred.

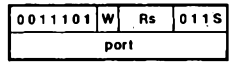
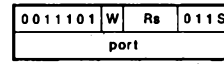
OUT (SOUT) Privileged
(Special) Output

OUT dst, src	dst: IR, DA
OUTB	src: R
SOUT dst, src	dst: DA
SOUTB	src: R

Operation: dst ← src

The contents of the source register are loaded into the destination, an Output or Special Output port. OUT and OUTB are used for normal I/O operation; SOUT and SOUTB are used for special I/O operation.

Flags: No flags affected.

Destination Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
IR:	OUT @Rd, Rs OUTB @Rd, Rbs		10		10
DA:	OUT port, Rs OUTB port, Rbs SOUT port, Rs SOUTB port, Rbs		12		12

Example: If register R6 contains %5252, the instruction

```
OUT  %1120, R6
```

will output the value %5252 to the port %1120.

Privileged OUTD (SOUTD)
(Special) Output and Decrement

OUTD dst, src, r
OUTDB
SOUTD
SOUTDB

dst: IR
src: IR

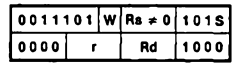
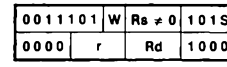
Operation:

dst ← src
AUTODECREMENT src (by 1 if byte, by 2 if word)
r ← r − 1

This instruction is used for block output of strings of data. OUTD and OUTDB are used for normal I/O operation; SOUTD and SOUTDB are used for special I/O operation. The contents of the memory location addressed by the source register are loaded into the I/O port addressed by the destination word register. I/O port addresses are 16 bits. The source register is then decremented by one if a byte instruction, or by two if a word instruction, thus moving the pointer to the previous element of the string in memory. The word register specified by "r" (used as a counter) is then decremented by one. The address of the I/O port in the destination register is unchanged.

Flags:

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

Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
IR:	OUTD @Rd, @Rs[1], r OUTDB @Rd, @Rs[1], r SOUTD @Rd, @Rs[1], r SOUTDB @Rd, @Rs[1], r		21		21

Example:

In segmented mode, if register R2 contains the I/O port address %0030, register RR6 contains %12005552 (segment %12, offset %5552), the word at memory location %12005552 contains %1234, and register R8 contains %1001, the instruction

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. The V flag will be cleared. In nonsegmented mode, a word register would be used instead of RR6.

Note 1: Word register in nonsegmented mode, register pair in segmented mode.

OUTI (SOUTI) Privileged
(Special) Output and Increment

OUTI dst, src, r
OUTIB
SOUTI
SOUTIB

dst: IR
src: IR

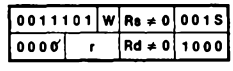
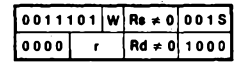
Operation:

dst ← src
AUTOINCREMENT src (by 1 if byte, by 2 if word)
r ← r − 1

This instruction is used for block output of strings of data. OUTI and OUTIB are used for normal I/O operation; SOUTI and SOUTIB are used for special I/O operation. The contents of the memory location addressed by the source register are loaded into the I/O port addressed by the destination word register. I/O port addresses are 16-bit. The source register is then incremented by one if a byte instruction, or by two if a word instruction, thus moving the pointer to the next element of the string in memory. The word register specified by "r" (used as a counter) is then decremented by one. The address of the I/O port in the destination register is unchanged.

Flags:

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

Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
IR:	OUTI @Rd, @Rs[1], r OUTIB @Rd, @Rs[1], r SOUTI @Rd, @Rs[1], r SOUTIB @Rd, @Rs[1], r		21		21

Example: This instruction can be 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. Bit 7 of each character is initially zero. This example assumes nonsegmented mode. In segmented mode, R2 would be replaced with a register pair.

```
        LD      R1, #PORT       !load I/O address!
        LDA     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:
```

Note 1: Word register in nonsegmented mode, register pair in segmented mode.

POP
Pop

POP dst, src — dst: R, IR, DA, X
POPL — src: IR

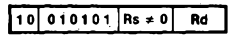
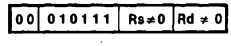
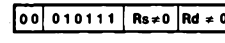
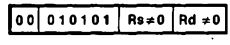
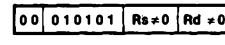
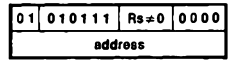
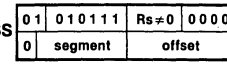
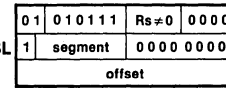
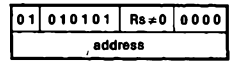
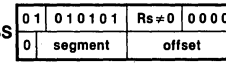
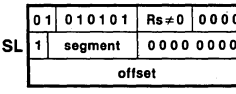
Operation:

dst ← src
AUTOINCREMENT src (by 2 if word, by 4 if long)

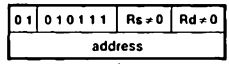
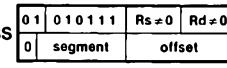
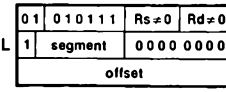
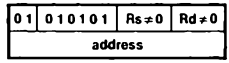
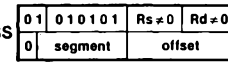
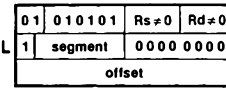
The contents of the location addressed by the source register (a stack pointer) are loaded into the destination. The source register is then incremented by a value which equals the size in bytes of the destination operand, thus removing the top element of the stack by changing the stack pointer. Any register except R0 (or RR0 in segmented mode) can be used as a stack pointer.

With the POPL instruction, the same register cannot be used in both the source and destination addressing fields.

Flags: No flags affected

Destination Addressing Mode	Assembler Language Syntax	Nonsegmented Mode: Instruction Format	Cycles	Segmented Mode: Instruction Format	Cycles
R:	POP Rd, @Rs[1]		8		8
	POPL RRd, @Rs[1]		12		12
IR:	POP @Rd[1], @Rs[1]		12		12
	POPL @Rd[1], @Rs[1]		19		19
DA:	POP address, @Rs[1]		16	SS 	16
				SL 	19
	POPL address, @Rs[1]		23	SS 	23
				SL 	26

POP
Pop

Destination Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
X:	POP addr(Rd), @Rs[1]		16	SS 	16
				SL 	19
	POPL addr(Rd), @Rs[1]		23	SS 	23
				SL 	26

Example: In nonsegmented mode, if register R12 (a stack pointer) contains %1000, the word at location %1000 contains %0055, and register R3 contains %0022, the instruction

```
POP  R3, @R12
```

will leave the value %0055 in R3 and the value %1002 in R12. In segmented mode, a register pair must be used as the stack pointer instead of R12.

Note 1: Word register in nonsegmented mode, register pair in segmented mode.

PUSH
Push

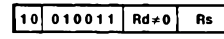
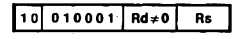
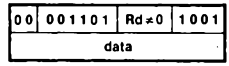
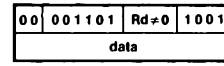
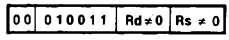
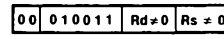
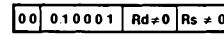
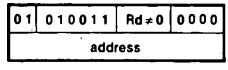
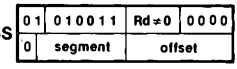
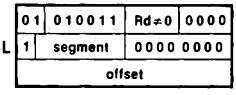
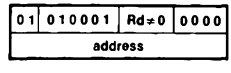
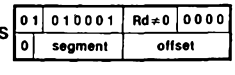
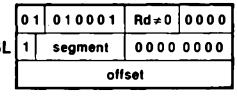
PUSH dst, src — dst: IR
PUSHL — src: R, IM, IR, DA, X

Operation: AUTODECREMENT dst (by 2 if word, by 4 if long)
dst ← src

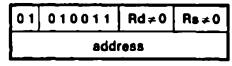
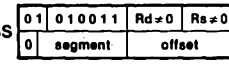
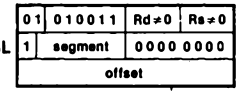
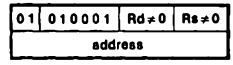
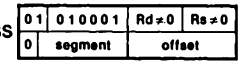
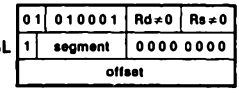
The contents of the destination register (a stack pointer) are decremented by a value which equals the size in bytes of the source operand. Then the source operand is loaded into the location addressed by the updated destination register, thus adding a new element to the top of the stack by changing the stack pointer. Any register except R0 (or RR0 in segmented mode) can be used as a stack pointer.

With PUSHL, the same register cannot be used for both the source and destination addressing fields.

Flags: No flags affected

Source Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
R:	PUSH @Rd, Rs		9		9
	PUSHL @Rd, RRs		12		12
IM:	PUSH @Rd, #data		12		12
IR:	PUSH @Rd, @Rs		13		13
	PUSHL @Rd, @Rs		20		20
DA:	PUSH @Rd, address		14	SS 	14
				SL 	17
	PUSHL@Rd, address		21	SS 	13
				SL 	24

PUSH
Push

Source Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
X:	PUSH @Rd[1], addr(Rs)		14	SS 	14
				SL 	17
	PUSHL @Rd[1], addr(Rs)		21	SS 	21
				SL 	24

Example: In nonsegmented mode, if register R12 (a stack pointer) contains %1002, the word at location %1000 contains %0055, and register R3 contains %0022, the instruction

```
PUSH @R12, R3
```

will leave the value %0022 in location %1000 and the value %1000 in R12. In segmented mode, a register pair must be used as the stack pointer instead of R12.

Note 1: Word register is used in nonsegmented mode, register pair in segmented mode.

RES
Reset Bit

RES dst, src **RESB**	dst: R, IR, DA, X src: IM or dst: R src: R	

Operation: dst(src) ← 0

This instruction 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 (Static), or as a word register which contains the value (Dynamic). In the second case, the destination 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.

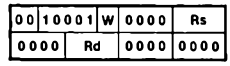
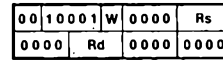
Only the lower four bits of the source operand are used to specify the bit number for RES, while only the lower three bits of the source operand are used with RESB. When the source operand is an immediate value, the "src field" in the instruction format encoding contains the bit number in the lowest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

Flags: No flags affected

Reset Bit Static

Destination Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
R:	RES Rd, #b RESB Rbd, #b	10 | 10001 | W | Rd | b	4	10 | 10001 | W | Rd | b	4
IR:	RES @Rd[1], #b RESB @Rd[1], #b	00 | 10001 | W | Rd≠0 | b	11	00 | 10001 | W | Rd≠0 | b	11
DA:	RES address, #b RESB address, #b	01 | 10001 | W | 0000 | b address	13	SS: 01 | 10001 | W | 0000 | b 0 | segment | offset	14
				SL: 01 | 10001 | W | 0000 | b 1 | segment | 0000 0000 offset	16
X:	RES addr(Rd), #b RESB addr(Rd), #b	01 | 10001 | W | Rd≠0 | b address	14	SS: 01 | 10001 | W | Rd≠0 | b 0 | segment | offset	14
				SL: 01 | 10001 | W | Rd≠0 | b 1 | segment | 0000 0000 offset	17

Reset Bit Dynamic

Source Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
R:	RES Rd, Rs RESB Rbd, Rs		10		10

Example: If register RL3 contains %B2 (10110010), the instruction

```
RESB  RL3, #1
```

will leave the value %B0 (10110000) in RL3.

Note 1: Word register in nonsegmented mode, register pair in segmented mode.

RESFLG
Reset Flag

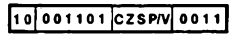
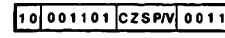
RESFLG flag flag: C, Z, S, P, V

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 bits in the instruction are one. If the bit in the instruction corresponding to a flag 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

	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
	RESFLG flags		7		7

Example: If the C, S, and V flags are 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

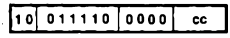
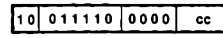
RET cc

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 executed procedure at the end of a procedure entered by a CALL or CALR instruction. If the condition specified by "cc" is satisfied by the flags in the FCW, 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 list of condition codes. The stack pointer used is R15 in nonsegmented mode, or RR14 in segmented mode. If the condition is not satisfied, then the instruction following the RET instruction is executed. If no condition is specified, the return is taken regardless of the flag settings.

Flags: No flags affected

Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles[1]	Instruction Format	Cycles[1]
	RET cc		10/7		13/7

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 instruction

```
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).

Note 1: The two values correspond to return taken and return not taken.

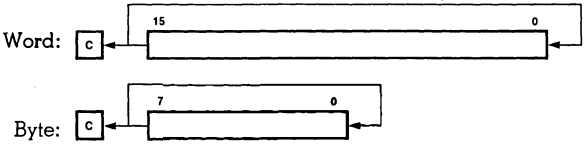
RL
Rotate Left

RL dst, src
RLB

dst: R
src: IM

Operation:

Do src times: (src = 1 or 2)
tmp ← dst
c ← tmp (msb)
dst(0) ← tmp (msb)
dst (n + 1) ← tmp (n) (for n = 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 most significant bit (msb) of the destination operand is moved to the bit 0 position 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 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

Destination Addressing Mode	Assembler Language Syntax[1]	Nonsegmented Mode		Segmented Mode	
		Instruction Format[2]	Cycles[3]	Instruction Format[2]	Cycles[3]
R:	RL Rd, #n RLB Rbd, #n	10 | 11001 | W | Rd | 00 | S | 0	6/7	10 | 11001 | W | Rd | 00 | S | 0	6/7

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.

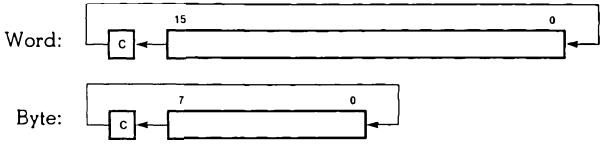
Note 1: n = source operand.
Note 2: s = 0 for rotation by 1 bit; s = 1 for rotation by 2 bits.
Note 3: The given execution times are for rotation by 1 and 2 bits respectively.

RLC
Rotate Left through Carry

RLC dst: R
RLCB src: IM

Operation:

Do src times: (src = 1 or 2)
tmp ← c
c ← dst (msb)
dst (n + 1) ← dst (n) (for n = msb − 1 to 0)
dst (0) ← tmp

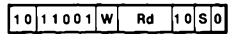
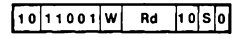


The contents of the destination operand with the C flag are rotated left one bit position if the source operand is 1, or two bit positions if the source operand is 2. The most significant bit (msb) of the destination operand replaces the C flag and the previous value of the C flag is moved to the bit 0 position of the destination during each rotation.

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 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

Destination Addressing Mode	Assembler Language Syntax[1]	Nonsegmented Mode		Segmented Mode	
		Instruction Format[2]	Cycles[3]	Instruction Format[2]	Cycles[3]
R:	RLC Rd, #n RLCB Rbd, #n		6/7		6/7

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.

Note 1: n = source operand.
Note 2: s = 0 for rotation by 1 bit; s = 1 for rotation by 2 bits.
Note 3: The given execution times are for rotation by 1 and 2 bits respectively.

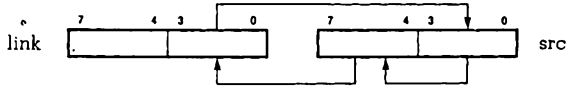
RLDB
Rotate Left Digit

RLDB link, src

src: R
link: R

Operation:

tmp (0:3) ← link (0:3)
link (0:3) ← src (4:7)
src (4:7) ← src (0:3)
src (0:3) ← tmp (0:3)



The low digit of the link byte register is logically concatenated to the source byte register. The resulting three-digit quantity is rotated to the left by one BCD digit (four bits). The lower digit of the source is moved to the upper digit of the source; the upper digit of the source is moved to the lower digit of the link, and the lower digit of the link is moved to the lower digit of the source. The upper digit of the link 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. The link serves to transfer digits between successive bytes of the string. This is analogous to the use of the Carry flag in multiple precision shifting using the RLC instruction.

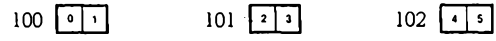
The same byte register must not be used as both the source and the link.

Flags:

C: Unaffected
Z: Set if the link is zero after the operation; cleared otherwise
S: Undefined
V: Unaffected
D: Unaffected
H: Unaffected

Destination Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
R:	RLDB Rbl, Rbs	10 111110 Rbs Rbl	9	10 111110 Rbs Rbl	9

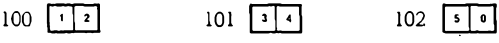
Example: If location 100 contains the BCD digits 0,1 (00000001), location 101 contains 2,3 (00100011), and location 102 contains 4,5 (01000101)



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.



In segmented mode, R2 would be replaced by a register pair.

RR
Rotate Right

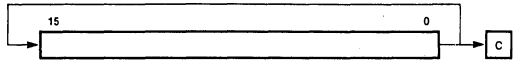
RR dst, src
RRB

dst: R
src: IM

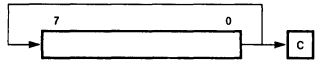
Operation:

Do src times: (src = 1 or 2)
tmp ← dst
c ← tmp (0)
dot (msb) ← tmp (0)
dst (n − 1) ← tmp (n) (for n = 1 to msb)

Word:



Byte:

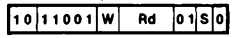


The contents of the destination operand are rotated right one bit position if the source operand is 1, or two bit positions if the source operand is 2. The least significant bit of the destination operand is moved to the most significant bit (msb) and also replaces the C flag.

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 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

Destination Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format[1]	Cycles[2]	Instruction Format[1]	Cycles[2]
R:	RR Rd, #n RRB Rbd, #n		6/7		6/7

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.

Note 1: s = 0 for rotation by 1 bit; s = 1 for rotation by 2 bits.
Note 2: The given execution times are for rotation by 1 and 2 bits respectively.

RRC
Rotate Right through Carry

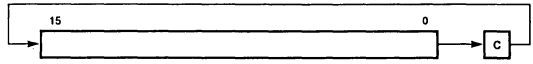
RRC dst, src
RRCB

dst: R
src: IM

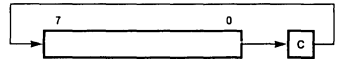
Operation:

Do src times: (src = 1 or 2)
tmp ← c
c ← dst (0)
dst (n) ← dst (n + 1) (for n = 0 to msb − 1)
dst (msb) ← tmp

Word:



Byte:

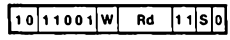
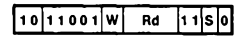


The contents of the destination operand with the C flag are rotated one bit position if the source operand is 1, or two bit positions if the source operand is 2. The least significant bit of the destination operand replaces the C flag and the previous value of the C flag is moved to the most significant bit (msb) position of the destination during each rotation.

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

Destination Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format[1]	Cycles[2]	Instruction Format[1]	Cycles[2]
	RRC Rd, #n RRCB Rbd, #n		6/7		6/7

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 (1000000011011l) in R0 and clear the Carry flag.

Note 1: s = 0 for rotation by 1 bit; s = 1 for rotation by 2 bits
Note 2: The given execution times are for rotation by 1 and 2 bits respectively.

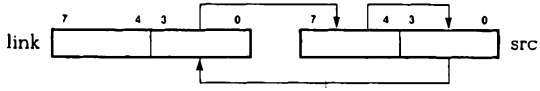
RRDB
Rotate Right Digit

RRDB link, src

src: R
link: R

Operation:

tmp (0:3) ← link (0:3)
link (0:3) ← src (0:3)
src (0:3) ← src (4:7)
src (4:7) ← tmp (0:3)



The low digit of the link byte register is logically concatenated to the source byte register. The resulting three-digit quantity is rotated to the right by one BCD digit (four bits).

The lower digit of the source is moved to the lower digit of the link; the upper digit of the source is moved to the lower digit of the source and the lower digit of the link is moved to the upper digit of the source.

The upper digit of the link is unaffected. In multiple-digit BCD arithmetic, this instruction can be used to shift to the right a string of BCD digits, thus dividing it by a power of ten. The link serves to transfer digits between successive bytes of the string. This is analogous to the use of the carry flag in multiple precision shifting using the RRC instruction.

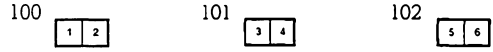
The same byte register must not be used as both the source and the link.

Flags:

C: Unaffected
Z: Set if the link is zero after the operation; cleared otherwise
S: Undefined
V: Unaffected
D: Unaffected
H: Unaffected

Destination Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
R:	RRDB Rbl, Rbs	10 | 111100 | Rbs | Rbl	9	10 | 111100 | Rbs | Rbl	9

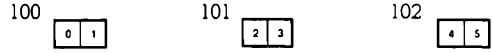
Example: If location 100 contains the BCD digits 1,2 (00010010), location 101 contains 3,4 (00110100), and location 102 contains 5,6 (01010110)



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. RH1 will contain 6, the remainder from dividing the string by 10.



In segmented mode, R2 would be replaced by a register pair.

SBC
Subtract with Carry

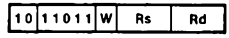
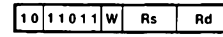
SBC dst, src — dst: R
SBCB — src: R

Operation: dst ← dst − src − C

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"

Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
R:	SBC Rd, Rs SBCB Rbd, Rbs		5		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 src src: IM

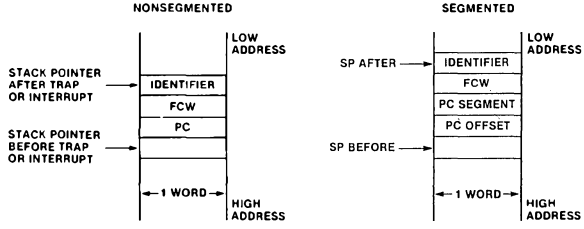
Operation:

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, which includes the source operand (an 8-bit value) is pushed. The PS includes the Flag 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 in nonsegmented mode, or RR14 in segmented mode), 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 6.2.4), 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:



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 onto 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.

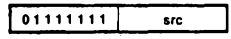
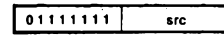
Flags:

No flags affected
Flags loaded from Program Status Area

SC

System Call

Privileged

Source Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
IM:	SC #src		33		39

Example: In the nonsegmented Z8002, if the contents of the program counter are %1000, the contents of the system stack pointer (R15) are %3006, and the Program Counter and FCW values associated with the System Call trap in the Program Status Area are %2000 and %1000, respectively, the instruction

```
SC  #3                    !system call, request code = 3!
```

causes the system stack pointer to be decremented to %3000. Location %3000 contains %7F03 (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, and the FCW contains %1000.

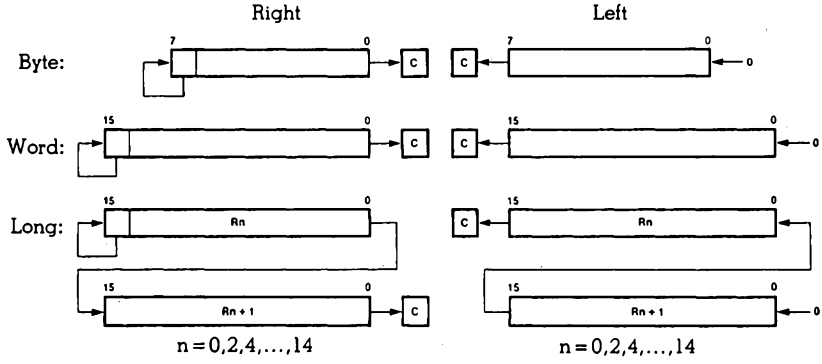
SDA
Shift Dynamic Arithmetic

SDA dst, src
SDAB
SDAL

dst: R
src: R

Operation:

Right (src negative)
Do src times:
  c ← dst (0)
  dst (n) ← dst (n + 1) (for n = 0 to msb − 1)
  dst (msb) ← dst (msb)
Left (src positive)
Do src times:
  c ← dst (msb)
  dst (n + 1) ← dst (n) (for n = msb − 1 to 0)
  dst (0) ← 0



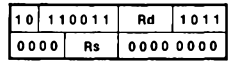
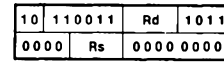
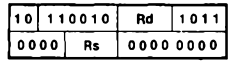
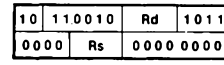
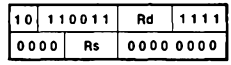
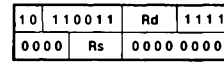
The destination operand is shifted arithmetically left or right by the number of bit positions specified by the contents of the source operand, a word register.

The shift count ranges from -8 to +8 for SDAB, from -16 to +16 for SDA and from -32 to +32 for SDAL. If the value is outside the specified range, the operation is undefined. The source operand is represented as a 16-bit two's complement value. Positive values specify a left shift, while negative values specify a right 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 most significant bit (msb) of the destination. The setting of the carry bit is undefined for zero shift.

Flags:

C: Set if the last bit shifted from the destination was 1, undefined for zero shift; 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

SDA
Shift Dynamic Arithmetic

Destination Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles[1]	Instruction Format	Cycles[1]
R:	SDA Rd, Rs		15 + 3n		15 + 3n
	SDAB Rbd, Rs		15 + 3n		15 + 3n
	SDAL RRd, Rs		15 + 3n		15 + 3n

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.

Note 1: n = number of bit positions; the execution time for n = 0 is the same as for n = 1.

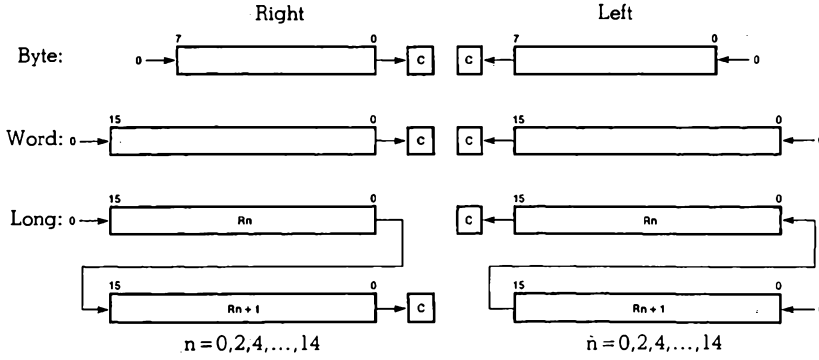
SDL
Shift Dynamic Logical

SDL dst, src — dst: R
SDLB — src: R
SDLL

Operation:

Right
Do src times
  c ← dst (0)
  dst (n) ← dst (n + 1) (for n = 0 to msb − 1)
  dst (msb) ← 0

Left
Do src times
  c ← dst (msb)
  dst (n + 1) ← dst (n) (for n = msb − 1 to 0)
  dst (0) ←

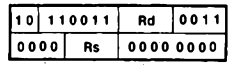
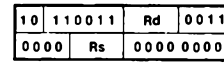
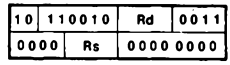
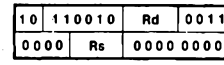
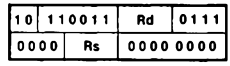
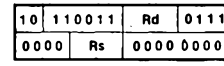


The destination operand is shifted logically left or right by the number of bit positions specified by the contents of the source operand, a word register. The shift count ranges from −8 to +8 for SDL, from −16 to +16 for SDLB and from −32 to +32 for SDLL. If the value is outside the specified range, the operation is undefined. The source operand is represented as a 16-bit two's complement value. Positive values specify a left shift, while negative values specify a right 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 (msb) 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. The setting of the carry bit is undefined for zero shift.

Flags:

C: Set if the last bit shifted from the destination was 1, undefined for zero shift; 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

SDL
Shift Dynamic Logical

Destination Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
R:	SDL Rd, Rs		15 + 3n		15 + 3n
	SDLB Rbd, Rs		15 + 3n		15 + 3n
	SDLL RRd, Rs		15 + 3n		15 + 3n

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.

Note 1: n = number of bit positions; the execution time for n = 0 is the same as for n = 1.

SET dst, src **SETB**	dst: R, IR, DA, X src: IM or dst: R src: 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 (Static), or as a word register which contains the value (Dynamic). 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.

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.

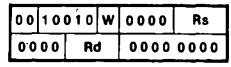
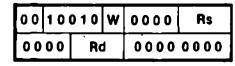
Flags: No flags affected

Set Bit Static

Destination Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
R:	SET Rd, #b SETB Rbd, #b	10 | 10010 | W | Rd | b	4	10 | 10010 | W | Rd | b	4
IR:	SET @Rd[1], #b SETB @Rd[1], #b	00 | 10010 | W | Rd≠0 | b	11	00 | 10010 | W | Rd≠0 | b	11
DA:	SET address, #b SETB address, #b	01 | 10010 | W | 0000 | b address	13	SS: 01 | 10010 | W | 0000 | b 0 | segment | offset	14
				SL: 01 | 10010 | W | 0000 | b 1 | segment | 0000 0000 offset	16
X:	SET addr(Rd), #b SETB addr(Rd), #b	01 | 10010 | W | Rd≠0 | b address	14	SS: 01 | 10010 | W | Rd ≠ 0 | b 0 | segment | offset	14
				SL: 01 | 10010 | W | Rd ≠ 0 | b 1 | segment | 0000 0000 offset	17

SET
Set Bit

Set Bit Dynamic

Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
R:	SET Rd, Rs SETB Rbd, Rs		10		10

Example: If register RL3 contains %B2 (10110010) and register R2 contains the value 6, the instruction

SETB RL3, R2

will leave the value %F2 (11110010) in RL3.

Note 1: Word register in nonsegmented mode, register pair in segmented mode.

SETFLG
Set Flag

SETFLG flag Flag: C, Z, S, P, V

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 bits in the instruction are one. If the bit in the instruction corresponding to a flag 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

	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
	SETFLG flags	10001101 | CZSP/V | 0001	7	10001101 | CZSP/V | 0001	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).

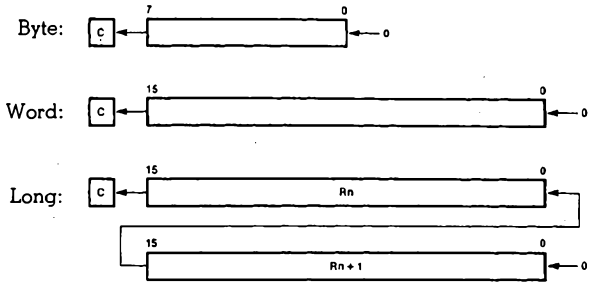
SLA
Shift Left Arithmetic

SLA dst, src
SLAB
SLAL

dst: R
src: IM

Operation:

Do src times:
c ← dst (msb)
dst (n + 1) ← dst (n) (for n = msb − 1 to 0)
dst (0) ← 0



n = 0, 2, 4, ..., 14

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. The operation is the equivalent of a multiplication of the destination by a power of two with overflow indication.

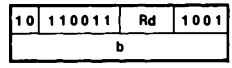
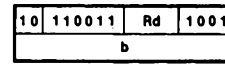
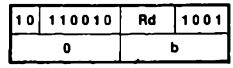
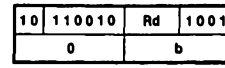
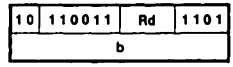
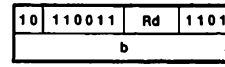
The src field is encoded in the instruction format as the 8- or 16-bit two's complement positive value of the source operand. For each operand size, the operation is undefined if the source op- and is not in the specified range.

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, undefined for zero shift; 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

SLA
Shift Left Arithmetic

Destination Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles[1]	Instruction Format	Cycles[1]
R:	SLA Rd, #b		13 + 3b		13 + 3b
	SLAB Rbd, #b		13 + 3b		13 + 3b
	SLAL RRd, #b		13 + 3b		13 + 3b

Example: If register pair RR2 contains %1234ABCD, the statement

SLAL RR2,#8

will leave the value %34ABCD00 in RR2 and clear the Carry flag.

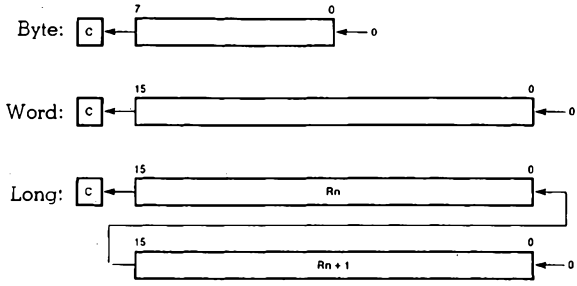
Note 1: b = number of bit positions; the execution time for b = 0 is the same as for b = 1.

SLL
Shift Left Logical

SLL dst, src — dst: R
SLB — src: IM
SLLL

Operation:

Do src times:
c ← dst (msb)
dst (n + 1) ← dst (n) (for n = msb − 1 to 0)
dst (0) ← 0



n = 0, 2, 4, ..., 14

The destination operand is shifted logically left by 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 setting of the carry bit is undefined for zero shift. The least significant bit of the destination is filled with 0, and the C flag is loaded from the most significant bit (msb) of the destination. This instruction performs an unsigned multiplication of the destination by a power of two.

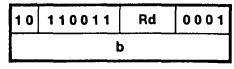
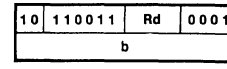
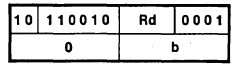
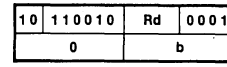
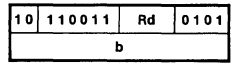
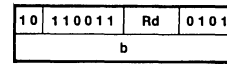
The src field is encoded in the instruction format as the 8- or 16-bit positive value of the source operand. For each operand size, the operation is undefined if the source operand is not in the specified range.

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, undefined for zero shift; 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

SLL
Shift Left Logical

Destination Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles[1]	Instruction Format	Cycles[1]
R:	SLL Rd, #b		13+3b		13+3b
	SLLB Rbd, #b		13+3b		13+3b
	SLLL RRd, #b		13+3b		13+3b

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.

Note 1: b = number of bit positions; the execution time for b = 0 is the same as for b = 1.

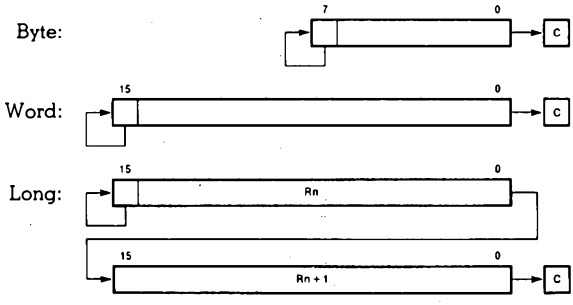
SRA
Shift Right Arithmetic

SRA dst, src
SRAB
SRAL

dst: R
src: IM

Operation:

Do src times:
c ← dst (0)
dst (n) ← dst (n + 1)(for n = 0 to msb − 1)
dst (msb) ← dst (msb)



n = 0, 2, 4, ..., 14

The destination operand is shifted arithmetically right by the number of bit positions specified by the source operands. 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 right shift of zero for SRA is not possible. The most significant bit (msb) of the destination is replicated, and the C flag is loaded from bit 0 of the destination, this instruction performs a signed division of the destination by a power of two.

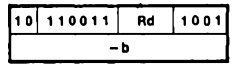
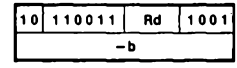
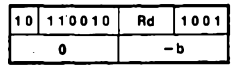
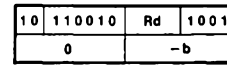
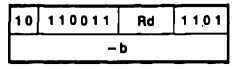
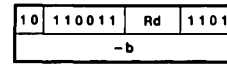
The src field is encoded in the instruction format as the 8- or 16-bit two's complement negative of the source operand. For each operand size, the operation is undefined if the source operand is not in the specified range.

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

SRA
Shift Right Arithmetic

Destination Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles[1]	Instruction Format	Cycles[1]
R:	SRA Rd, #b		13 + 3b		13 + 3b
	SRAB Rbd, #b		13 + 3b		13 + 3b
	SRAL RRd, #b		13 + 3b		13 + 3b

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.

Note 1: b = number of bit positions; the execution time for b = 0 is the same as for b = 1.

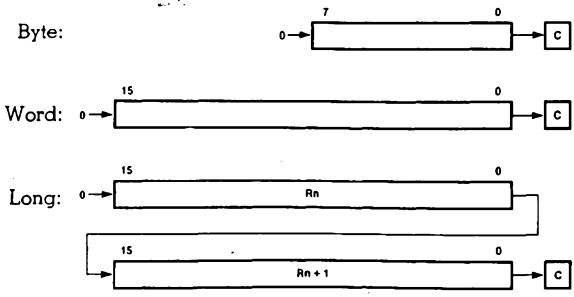
SRL
Shift Right Logical

SRL dst, src
SRLB
SRLL

dst: R
src: IM

Operation:

Do src times:
c ← dst (0)
dst (n) ← dst (n + 1)(for n = 0 to msb − 1)
dst (msb) ← 0



n = 0, 2, 4, ..., 14

The destination operand is shifted logically right by 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 right shift of zero for SRL is not possible. The most significant bit (msb) 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.

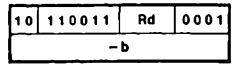
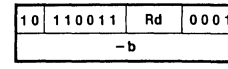
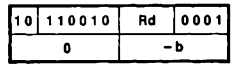
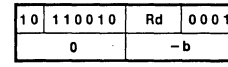
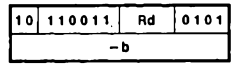
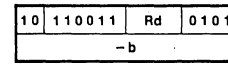
The src field is encoded in the instruction format as the 8- or 16-bit negative value of the source operand in two's complement rotation. For each operand size, the operation is undefined if the source operand is not in the range specified above.

The source operand may be omitted from the assembly language statement and thus defaults to the value of 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 most significant bit of the result is one; cleared otherwise
V: Undefined
D: Unaffected
H: Unaffected

SRL
Shift Right Logical

Destination Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles[1]	Instruction Format	Cycles[1]
R:	SRL Rd, #b		13 + 3b		13 + 3b
	SRLB Rbd, #b		13 + 3b		13 + 3b
	SRLL RRd, #b		13 + 3b		13 + 3b

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.

Note 1: b = number of bit positions; the execution time for b = 0 is the same as for b = 1.

SUB
Subtract

SUB dst, src
SUBB
SUBL

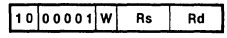
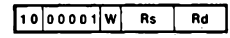
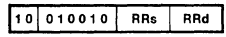
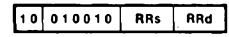
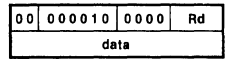
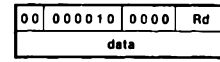
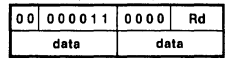
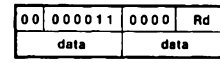
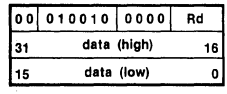
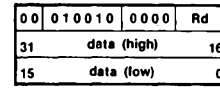
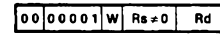
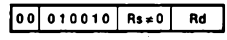
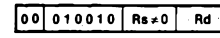
dst: R
src: R, IM, IR, DA, X

Operation: dst ← dst − src

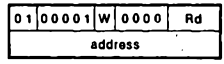
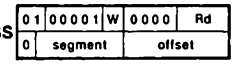
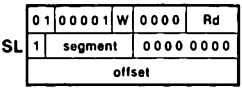
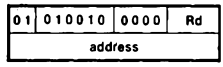
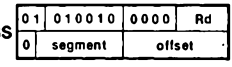
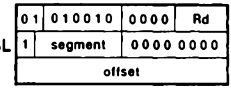
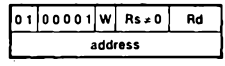
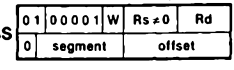
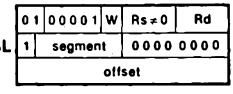
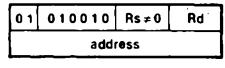
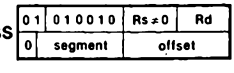
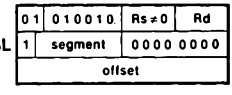
The source operand is subtracted from the destination operand and the result is stored in the destination. The contents of the source are not affected. Subtraction is performed by adding the two's complement of the source operand to the destination operand.

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"

Source Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
R:	SUB Rd, Rs SUBB Rbd, Rbs		4		4
	SUBL RRd, RRs		8		8
IM:	SUB Rd, #data		7		7
	SUBB Rbd, #data		7		7
	SUBL RRd, #data		14		14
IR:	SUB Rd, @Rs[1] SUBB Rbd, @Rs[1]		7		7
	SUBL RRd, @Rs[1]		14		14

SUB
Subtract

Source Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
DA:	SUB Rd, address SUBB Rbd, address		9		10
					12
	SUBL RRd, address		15		16
					18
X:	SUB Rd, addr(Rs) SUBB Rbd, addr(Rs)		10		10
					13
	SUBL RRD, addr(Rs)		16		16
					19

Example: If register R0 contains %0344, the statement

```
SUB   R0,#%AA
```

will leave the value %029A in R0.

Note 1: Word register in nonsegmented mode, register pair in segmented mode.

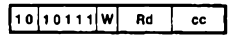
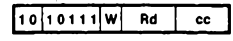
TCC
Test Condition Code

TCC cc, dst dst: R
TCCB

Operation: if cc is satisfied then
dst (0) ← 1

This instruction is used to create a Boolean data value based on the flags set by a previous operation. The flags in the FCW are tested to see if the condition specified by "cc" is satisfied. If the condition is satisfied, then the least significant bit of the destination is set. If the condition is not satisfied, bit zero of the destination is not cleared but retains its previous value. All other bits in the destination are unaffected by this instruction.

Flags: No flags affected

Destination Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
R:	TCC cc, Rd TCCB cc, Rbd		5		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.

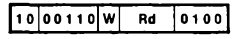
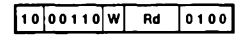
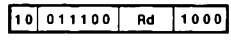
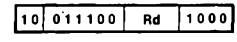
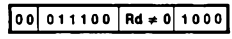
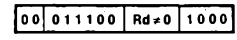
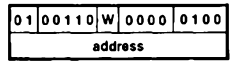
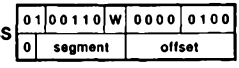
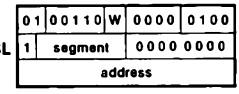
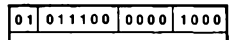
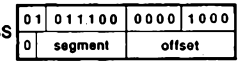
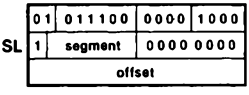
TEST dst
TESTB
TESTL

dst: R, IR, DA, X

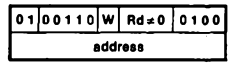
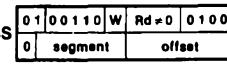
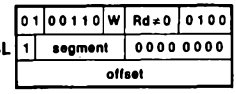
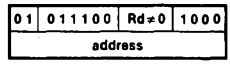
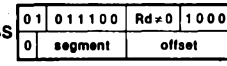
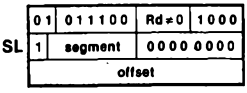
Operation: dst OR 0

The destination operand is tested (logically ORed with zero), and the Z, S and P flags are set to reflect the attributes of the result. The flags may then be used for 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

Destination Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
R:	TEST Rd TESTB Rbd		7		7
	TESTL RRd		13		13
IR:	TEST @Rd TESTB @Rd		8		8
	TESTL @Rd		13		13
DA:	TEST address TESTB address		11	SS 	12
				SL 	14
	TESTL address		16	SS 	17
				SL 	19

TEST
Test

Destination Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
X:	TEST addr(Rd) TESTB addr(Rd)		12	SS 	12
				SL 	15
			17	SS 	17
				SL 	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.

Note 1: Word register in nonsegmented mode, register pair in segmented mode.

TRDB
Translate and Decrement

TRDB dst, src, r

dst: IR
src: IR

Operation:

dst ← src[dst]
AUTODECREMENT dst by 1
r ← r − 1

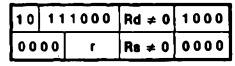
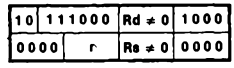
This instruction is used to translate a string of bytes from one code to another code. The contents of the location addressed by the destination register (the "target byte") are used as an index into a table of translation values whose lowest address is contained in the source register. The index is computed by adding the target byte to the address contained in the source register. The addition is performed following the rule for address 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 original contents of register RH1 are lost and are replaced by an undefined value. R1 in nonsegmented mode, or RR0 in segmented mode, must not be used as a source or destination pointer, and R1 should not be used as a counter. The source, destination, and counter registers must be separate and non-overlapping registers.

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. The source register is unchanged.

Flags:

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

Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
IR:	TRDB @Rd[1], @Rs[1], r		25		25

Example:

In nonsegmented mode, 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 instruction

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. The V flag will be cleared. RH1 will be set to an undefined value. In segmented mode, R6 and R9 would be replaced with register pairs.

Note 1: Word register in nonsegmented mode, register pair in segmented mode.

TRDRB
Translate Decrement and Repeat

TRDRB dst, src, R

dst: IR
src: IR

Operation:

dst ← src [dst]
AUTODECREMENT dst by 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") are 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 address 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 that 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. The original contents of register RH1 are lost and are replaced by an undefined value. The source register is unchanged. The source, destination, and counter registers must be separate and non-overlapping registers.

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.

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

Flags:

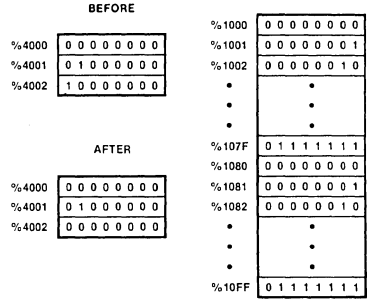
C: Unaffected
Z: Undefined
S: Unaffected
V: Set
D: Unaffected
H: Unaffected

Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles[2]	Instruction Format	Cycles[2]
IR:	TRDRB @Rbd[1], @Rbs[1], r	10 | 111000 | Rd ≠ 0 | 1100 0000 | r | Rs ≠ 0 | 0000	11 + 14n	10 | 111000 | Rd ≠ 0 | 1100 0000 | r | Rs ≠ 0 | 0000	11 + 14n

Example: In nonsegmented mode, 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 instruction

```
TRDRB   @R6, @R9, R12
```

will leave the values %00, %40, %00 in byte locations %4000 through %4002, respectively. Register R6 will contain %3FFF, and R12 will contain 0. R9 will not be affected. The V flag will be set, and the contents of RH1 will be replaced by an undefined value. In segmented mode, R6 and R9 would be replaced by register pairs.



Note 1: Word register in nonsegmented mode, register pair in segmented mode.
Note 2: n = number of data elements translated.

TRIB
Translate and Increment

TRIB dst, src, R

dst: IR
src: IR

Operation:

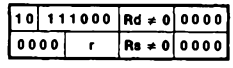
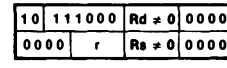
dst ← src[dst]
AUTOINCREMENT dst by 1
r ← r − 1

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") are 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 address 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 original contents of register RH1 are lost and are replaced by an undefined value. The source register is unchanged. The source, destination, and counter registers must be separate and non-overlapping registers.

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.

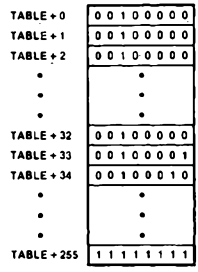
Flags:

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

Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
	TRIB @Rd¹, @Rs¹, r		25		25

Example: This instruction can be 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" 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. This example assumes nonsegmented mode. In segmented mode, R4 and R5 would be replaced by register pairs.

```
        LD      R3, #1000       !initialize counter!
        LDA     R4, STRING      !load start addresses!
        LDA     R5, TABLE
LOOP:
        CPB     @R4, #13        !check for return character!
        JR      EQ, DONE        !exit loop if found!
        TRIB    @R4, @R5, R3    !translate next byte!
        JR      NOV, LOOP       !repeat until counter = 0!
DONE:
```



Note 1: Word register in nonsegmented mode, register pair in segmented mode.

TRIRB
Translate, Increment and Repeat

TRIRB dst, src, R

dst: IR
src: IR

Operation:

dst ← src[dst]
AUTOINCREMENT dst by 1
r ← r − 1
repeat until r = 0

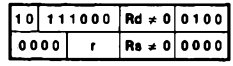
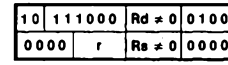
This instruction is used to translate a string of bytes from one code to another code. The contents of the location addressed by the destination register (the "target byte") are used as an index into a table of translation values whose lowest address is contained in the source register. The index is computed by adding the target byte to the address contained in the source register. The addition is performed following the rules for address 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. The original contents of register RH1 are lost and are replaced by an undefined value. The source register is unaffected. The source, destination, and counter registers must be separate and non-overlapping registers.

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.

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 cycles should be added to this instruction's execution time for each interrupt request that is accepted.

Flags:

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

Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles[2]	Instruction Format	Cycles[2]
IR:	TRIRB @Rd[1], @Rs[1], r		11 + 14n		11 + 14n

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. After executing the last instruction, the V flag is set and the contents of RH1 are lost. The example assumes nonsegmented mode. In segmented mode, R4 and R5 would be replaced by register pairs.

```
LDA    R4, STRING
LDA    R5, TABLE
LD     R3, #80
TRIRB  @R4, @R5, R3
```

Note 1: Word register in nonsegmented mode, register pair in segmented mode.
Note 2: n = number of data elements translated.

TRTDB
Translate, Test and Decrement

TRTDB src1, src2, R

src 1: IR
src 2: IR

Operation:

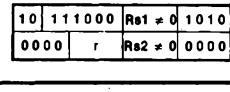
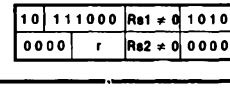
RH1 ← src2[src1]
AUTODECREMENT src1 by 1
r ← r − 1

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") are 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 address 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 second source register is unaffected. The source, destination, and counter registers must be separate and non-overlapping registers.

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.

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

Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
IR:	TRTDB @Rs1[1], @Rs2[1], r		25		25

Example:

In nonsegmented mode, 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 instruction

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. The Z and V flags will be cleared. In segmented mode, register pairs must be used instead of R6 and R9.

Note 1: Word register in nonsegmented mode, register pair in segmented mode.

TRTDRB
Translate, Test, Decrement and Repeat

TRTDRB src 1, src 2, R

src 1: IR
src 2: IR

Operation:

RH1 ← src 2[src1]
AUTODECREMENT src1 by 1
r ← r − 1
repeat until RH1 = 0 or r = 0

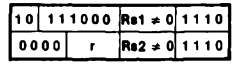
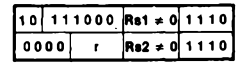
This instruction is used to scan a string of bytes testing for bytes with special meaning. The contents of the location addressed by the first source register (the "target byte") are used as an index into a table of translation values whose lowest address is contained in the second source register. The 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 address 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. The source, destination, and counter registers must be separate and non-overlapping registers.

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.

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 cycles should be added to this instruction's execution time for each interrupt request that is accepted.

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

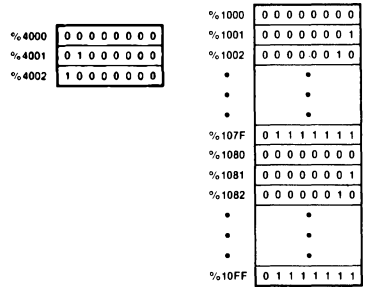
Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles[2]	Instruction Format	Cycles[2]
IR:	TRTDRB@Rs1[1],@Rs2[1],r		11 + 14n		11 + 14n

TRTDRB
Translate, Test, Decrement and Repeat

Example: In nonsegmented mode, if register R6 contains %4002, the bytes at locations %4000 through %4002 contain the values %00, %40, %80, repectively, 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 instruction

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. The Z and V flags will be cleared. In segmented mode, register pairs are used instead of R6 and R9.



Note 1: Word register in nonsegmented mode, register pair in segmented mode.
Note 2: n = number of data elements translated.

TRTIB
Translate, Test and Increment

TRTIB src 1, src 2, R

src 1: IR
src 2: IR

Operation:

RH1 ← src2[src1]
AUTOINCREMENT src1 by 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") are 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 address 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 second source register is unaffected. The source, destination, and counter registers must be separate and non-overlapping registers.

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.

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

Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
IR:	TRTIB @Rs1[1], @Rs2[1], r	10 | 111000 | Rs1 ≠ 0 | 0010 0000 | r | Rs2 ≠ 0 | 0000	25	10 | 111000 | Rs1 ≠ 0 | 0010 0000 | r | Rs2 ≠ 0 | 0000	25

TRTIB
Translate, Test and Increment

Example: This instruction can be 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 to an 8-bit value with odd parity. Lower case characters are translated to upper case, and any embedded control characters are skipped over. The translation table contains 128 bytes, which assumes that the most significant bit of each byte in the string to be translated is always zero. The first 32 entries and the 128th entry are zero, so that ASCII control characters and the "delete" character (%7F) are suppressed. The given instruction sequence is for nonsegmented mode. In segmented mode, register pairs would be used instead of R3 and R4.

```
        LD      R5, #72          !initialize counter!
        LDA     R3, STRING       !load start address!
        LDA     R4, TABLE
LOOP:
        TRTIB   @R3, @R4, R5     !translate and test next byte!
        JR      Z, LOOP          !skip control character!
        OUTB    PORTn, RH1       !output characters!
        JR      NOV, LOOP        !repeat until counter = 0!
DONE:
```

Note 1: Word register in nonsegmented mode, register pair in segmented mode.

TRTIRB
Test, Increment and Repeat

TRTIRB src 1, src 2, R

src 1: IR
src 2: IR

Operation:

RH1 ← src2[src1]
AUTOINCREMENT src1 by 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") are 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 address 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.

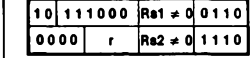
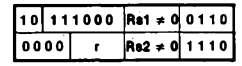
The first source register is then incremented by one, thus moving the pointer to the next element in the string. The word register specified by "r" (used as a counter) is then decremented by one. The entire operation is repeated until either the Z flag is clear, indicating that a non-zero translation 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. The source, destination, and counter registers must be separate and non-overlapping registers.

Target byte values which have corresponding zero translation table entry values are scanned over, while target byte values which have corresponding non-zero translation table entry values are detected and terminate the scan. 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.

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 cycles should be added to this instruction's execution time for each interrupt request that is accepted.

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

Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles[2]	Instruction Format	Cycles[2]
IR:	TRTIRB @Rs1[1], @Rs2[1], r		11 + 14n		11 + 14n

TRTIRB
Test, Increment and Repeat

Example: The following sequence of instructions can be used in nonsegmented mode 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 done. 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. In segmented mode, R4 and R5 must be replaced with register pairs.

```
                LDA         R4, STRING
                LDA         R5, TABLE
                LD          R6, #80
                TRTIRB      @R4, @R5, R6
                JR          NZ, SPECIAL
END_OF_STRING:              .
                            .
                            .
SPECIAL:
                JR          OV,LAST_CHAR_SPECIAL
                            .
                            .
                            .
LAST_CHAR_SPECIAL:
```

Note 1: Word register in nonsegmented mode, register pair in segmented mode.
Note 2: n = number of data elements translated.

TSET
Test and Set

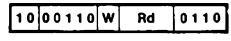
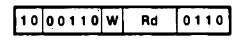
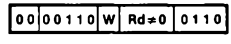
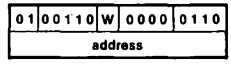
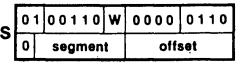
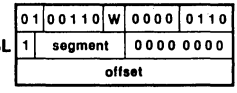
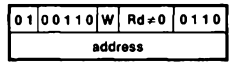
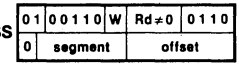
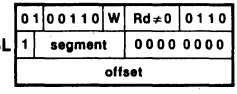
TSET dst — dst: R, IR, DA, X
TSETB

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.

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, unless this locking mechanism can be guaranteed to function correctly with multi-processor accesses.

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

Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
R:	TSET Rd TSETB Rbd		7		7
IR:	TSET @Rd[1] TSETB @Rd[1]		11		11
DA:	TSET address TSETB address		14	SS 	15
				SL 	17
X:	TSET addr(Rd) TSETB addr(Rd)		15	SS 	15
				SL 	18

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 con-!
                            !trolled by SEMAPHORE!
                            !is available!
        .
        .
        .
!Critical Region—only one software process!
!executes this code at a time!
        .
        .
        .
CLR     SEMAPHORE           !release resource controlled!
                            !by SEMAPHORE!
```

XOR
Exclusive Or

XOR dst, src
XORB

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

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

Source Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
R:	XOR Rd, Rs XORB Rbd, Rbs	10 | 00100 | W | Rs | Rd	4	10 | 00100 | W | Rs | Rd	4
IM:	XOR Rd, #data	00 | 001001 | 0000 | Rd data	7	00 | 001001 | 0000 | Rd data	7
	XORB Rbd, #data	00 | 001000 | 0000 | Rd data | data	7	00 | 001000 | 0000 | Rd data | data	7
IR:	XOR Rd, @Rs XORB Rbd, @Rs	00 | 00100 | W | Rs≠0 | Rd	7	00 | 00100 | W | Rs≠0 | Rd	7
DA:	XOR Rd, address XORB Rbd, address	01 | 00100 | W | 0000 | Rd address	9	SS: 01 | 00100 | W | 0000 | Rd 0 | segment | offset	10
				SL: 01 | 00100 | W | 0000 | Rd 1 | segment | 0000 0000 offset	12
X:	XOR Rd, addr(Rs) XORB Rbd, addr(Rs)	01 | 00100 | W | Rs≠0 | Rd address	10	SS: 01 | 00100 | W | Rs≠0 | Rd 0 | segment | offset	10
				SL: 01 | 00100 | W | Rs≠0 | Rd 1 | segment | 0000 0000 offset	13

XOR
Exclusive Or

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.

Note 1: Word register in nonsegmented mode, register pair in segmented mode.

EPA Instruction Templates

There are seven "templates" for EPA instructions. These templates correspond to EPA instructions, which combine EPU operations with possible transfers between memory and an EPU, between CPU registers and EPU registers, and between the Flag byte of the CPU's FCW and the EPU. Each of these templates is described on the following pages. The description assumes that the EPA control bit in the CPU's FCW has been set to 1. In addition, the description is from the point of view of the CPU—that is, only CPU activities are described; the operation of the EPU is implied, but the full specification of the instruction depends upon the implementation of the EPU and is beyond the scope of this manual.

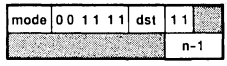
Fields ignored by the CPU are shaded in the diagrams of the templates. The 2-bit field in bit positions 0 and 1 of the first word of each template would normally be used as an identification field for selecting one of up to four EPUs in a multiple EPU system configuration. Other shaded fields would typically contain opcodes for instructing an EPU as to the operation it is to perform in addition to the data transfer specified by the template.

Extended Instruction
Load Memory from EPU

Operation: Memory ← EPU

The CPU performs the indicated address calculation and generates *n* EPU memory write transactions. The *n* words are supplied by an EPU and are stored in *n* consecutive memory locations starting with the effective address.

Flags/Registers: No flags or CPU registers are affected by this instruction.



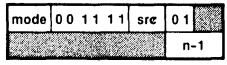
mode	dst	Clock Cycles NS	SS	SL
0 0	IR (dst ≠ 0)	11 + 3n		
0 1	X (dst ≠ 0)	15 + 3n	15 + 3n	18 + 3n
0 1	DA (dst = 0)	14 + 3n	15 + 3n	17 + 3n

Extended Instruction
Load EPU from Memory

Operation: EPU ← Memory

The CPU performs the indicated address calculation and generates *n* EPU memory read transactions. The *n* consecutive words are fetched from the memory locations starting with the effective address. The data is read by an EPU and operated upon according to the EPA instruction encoded into the shaded fields.

Flags/Registers: No flags or CPU registers are affected by this instruction.



mode	src	NS	Clock Cycles SS	SL
0 0	IR (src ≠ 0)	11 + 3n		
0 1	X (src ≠ 0)	15 + 3n	15 + 3n	18 + 3n
0 1	DA (src = 0)	14 + 3n	15 + 3n	17 + 3n

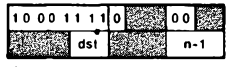
Extended Instruction
Load CPU from EPU

Operation: CPU ← EPU registers

The contents of *n* words are transferred from an EPU to consecutive CPU registers starting with register dst. CPU registers are transferred consecutively, with register 0 following register 15.

Flags/Registers: No flags are affected by this instruction.

Execution Time: 11 + 3n cycles.



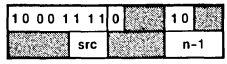
Extended Instruction
Load EPU from CPU

Operation: EPU ← CPU registers

The contents of *n* words are transferred to an EPU from consecutive CPU registers starting with register src. CPU registers are transferred consecutively, with register 0 following register 15.

Flags/Registers: No flags are affected by this instruction.

Execution Time: 11 + 3n cycles.



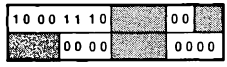
Extended Instruction
Load FCW from EPU

Operation: Flags ← EPU

The Flags in the CPU's Flag and Control Word are loaded with information from an EPU on AD lines AD_0-AD_7.

Flags/Registers: The contents of CPU register 0 are undefined after the execution of this instruction.

Execution Time: 14 cycles.



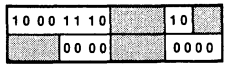
Extended Instruction
Load EPU from FCW

Operation: EPU ← Flags

The Flags in the CPU's Flag and Control Word are transferred to an EPU on AD lines AD_0-AD_7.

Flags/Registers: The flags in the FCW are unaffected by this instruction.

Execution Time: 14 cycles.



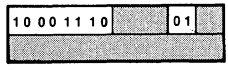
Extended Instruction
Internal EPU Operation

Operation: Internal EPU Operation

The CPU treats this template as a No Op. It is typically used to initiate an internal EPU operation.

Flags/Registers: The flags in the FCW are unaffected by this instruction.

Execution Time: 14 cycles.



Programmers Quick Reference

Mnemonics	Operands	Addr. Modes	Clock Cycles* Word, Byte NS	SS	SL	Long Word NS	SS	SL	Operation
ADC **ADCB**	R,src	R	5						**Add with Carry** R ← R + src + carry
ADD **ADDB** **ADDL**	R,src	R	4	4	4	8	8	8	**Add** R ← R + src
		IM	7	7	7	14	14	14	
		IR	7			14			
		DA	9	10	12	15	16	18	
		X	10	10	13	16	16	19	
AND **ANDB**	R,src	R	4	4	4				**AND** R ← R AND src
		IM	7	7	7				
		IR	7						
		DA	9	10	12				
		X	10	10	13				
BIT **BITB**	dst,b	R	4	4	4				**Test Bit Static** Z flag ← NOT dst bit specified by b
		IR	8						
		DA	10	11	13				
		X	11	11	14				
BIT **BITB**	dst,R	R	10	10	10				**Test Bit Dynamic** Z flag ← NOT dst bit specified by contents of R
CALL	dst	IR	10	10	15				**Call Subroutine** Autodecrement SP @ SP ← PC PC ← dst
		DA	12	18	20				
		X	13	18	21				
CALR	dst	RA	10	10	15				**Call Relative** Autodecrement SP @ SP ← PC PC ← PC + dst(range -4094 to +4096)
CLR **CLRB**	dst	R	7	7	7				**Clear** dst ← 0
		IR	8						
		DA	11	12	14				
		X	12	12	15				
COM **COMB**	dst	R	7	7	7				**Complement** dst ← NOT dst
		IR	12						
		DA	15	16	18				
		X	16	16	19				
COMFLG	flags		7	7	7				**Complement Flag** (Any combination of C, Z, S, P/V)
CP **CPB** **CPL**	R,src	R	4	4	4	8	8	8	**Compare with Register** R - src
		IM	7	7	7	14	14	14	
		IR	7			14			
		DA	9	10	12	15	16	18	
		X	10	10	13	16	16	19	
CP **CPB**	dst,IM	IR	11						**Compare with Immediate** dst - IM
		DA	14	15	17				
		X	15	15	18				

*** NS = Non-Segmented, SS = Short Segmented Offset, SL =Segmented Long Offset, Blank = Not Implemented.**

Mnemonics	Operands	Addr. Modes	Clock Cycles: Word, Byte NS	SS	SL	Long Word NS	SS	SL	Operation
CPD **CPDB**	R_X,src,R_Y,cc	IR	20						**Compare and Decrement** R_X – src Autodecrement src address $R_Y \leftarrow R_Y - 1$
CPDR **CPDRB**	R_X,src,R_Y,cc	IR	(11 + 9n)						**Compare, Decrement and Repeat** R_Y – src Autodecrement src address $R_X \leftarrow R_Y - 1$ Repeat until cc is true or $R_Y = 0$
CPI **CPIB**	R_X,src,R_Y,cc	IR	20						**Compare and Increment** R_X – src Autoincrement src address $R_Y \leftarrow R_Y - 1$
CPIR **CPIRB**	R_X,src,R_Y,cc	IR	(11 + 9n)						**Compare, Increment and Repeat** R_X – src Autoincrement src address $R_Y \leftarrow R_Y - 1$ Repeat until cc is true or $R_Y = 0$
CPSD **CPSDB**	dst,src,R,cc	IR	25						**Compare String and Decrement** dst – src Autodecrement dst and src addresses R ← R – 1
CPSDR **CPSDRB**	dst,src,R,cc	IR	(11 + 14n)						**Compare String, Decr. and Repeat** dst – src Autodecrement dst and src addresses R ← R – 1 Repeat until cc is true or R = 0
CPSI **CPSIB**	dst,src,R,cc	IR	25						**Compare String and Increment** dst – src Autoincrement dst and src addresses R ← R – 1
CPSIR **CPSIRB**	dst,src,R,cc	IR	(11 + 14n)						**Compare String, Incr. and Repeat** dst – src Autoincrement dst and src addresses R ← R – 1 Repeat until cc is true or R = 0
DAB	dst	R	5	5	5				**Decimal Adjust**
DEC **DECB**	dst,n	R IR DA X	4 11 13 14	4 14 14	4 16 17				**Decrement by n** dst ← dst – n (n = 1...16)
DI*	int		7	7	7				**Disable Interrupt** (Any combination of NVI, VI)
DIV **DIVL**	R,src	R IM IR DA X	107 107 107 108 109	 107 109 109	 107 111 112	744 744 744 745 746	 744 746 746	 744 748 749	**Divide** (signed) Word: $R_{n+1} \leftarrow R_{n,n+1}$ + src R_n ← remainder Long Word: $R_{n+2,n+3} \leftarrow R_{n...n+3}$ + src $R_{n,n+1}$ ← remainder

*Privileged instruction. Executed in system mode only.

Mnemonics	Operands	Addr. Modes	Clock Cycles: Word, Byte NS	SS	SL	Long Word NS	SS	SL	Operation
DJNZ **DBJNZ**	R,dst	RA	11	11	11				**Decrement and Jump if Non-Zero** R ← R − 1 If R ≠ 0: PC ← PC + dst(range -254 to 0)
EI*	int		7	7	7				**Enable Interrupt** (Any combination of NVI, VI)
EX **EXB**	R,src	R IR DA X	6 12 15 16	6 16 16	6 18 19				**Exchange** R ← src
EXTS **EXTSB** **EXTSL**	dst	R	11	11	11	11	11	11	**Extend Sign** Extend sign of low order half of dst through high order half of dst
HALT*			(8 + 3n)						**HALT**
IN* **INB***	R,src	IR DA	10 12	 12	 12				**Input** R ← src
INC **INCB**	dst,n	R IR DA X	4 11 13 14	4 14 14	4 16 17				**Increment by n** dst ← dst + n (n = 1...16)
IND* **INDB***	dst,src,R	IR	21						**Input and Decrement** dst ← src Autodecrement dst addresed R ← R − 1
INDR* **INDRB***	dst,src,R	IR	(11 + 10n)						**Input, Decrement and Repeat** dst ← src Autodecrement dst address R ← R − 1 Repeat until R = 0
INI* **INIB***	dst,src,R	IR	21						**Input and Increment** dst ← src Autoincrement dst address R ← R − 1
INIR* **INIRB***	dst,src,R	IR	(11 + 10n)						**Input, Increment and Repeat** dst ← src Autoincrement dst address R ← R − 1 Repeat until R = 0
IRET*			13	13	16				**Interrupt Return** PS ← @ SP Autoincrement SP
JP	cc,dst	IR IR DA X	10 7 7 8	 8 8	15 7 10 11	(taken) (not taken)			**Jump Conditional** If cc is true: PC ← dst
JR	cc,dst	RA	6	6	6				**Jump Conditional Relative** If cc is true: PC ← PC + dst (range -256 to +254)

***Privileged instruction. Executed in system mode only.**

Mnemonics	Operands	Addr. Modes	Clock Cycles: Word, Byte NS	SS	SL	Long Word NS	SS	SL	Operation
LD **LDB** **LDL**	R,src	R	3	3	3	5	5	5	**Load into Register** R ← src
		IM	7	7	7	11	11	11	
		IM	5	(byte only)					
		IR	7			11			
		DA	9	10	12	12	13	15	
		X	10	10	13	13	13	16	
		BA	14		14	17		17	
		BX	14		14	17		17	
LD **LDB** **LDL**	dst,R	IR	8			11			**Load into Memory** (Store) dst ← R
		DA	11	12	14	14	15	17	
		X	12	12	15	15	15	18	
		BA	14	14	14	17	17	17	
		BX	14	14	14	17	17	17	
LD **LDB**	dst,IM	IR	11						**Load Immediate into Memory** dst ← IM
		DA	14	15	17				
		X	15	15	18				
LDA	R,src	DA	12	13	15				**Load Address** R ← source address
		X	13	13	16				
		BA	15	15	15				
		BX	15	15	15				
LDAR	R,src	RA	15	15	15				**Load Address Relative**
LDCTL*	CTLR,src	R	7	7	7				**Load into Control Register** CTLR ← src
LDCTL*	dst,CLTR	R	7	7	7				**Load from Control Register** dst ← CTLR
LDCTLB	FLGR,src	R	7	7	7				**Load into Flag Byte Register** FLGR ← src
LDCTLB	dst,FLGR	R	7	7	7				**Load from Flag Byte Register** dst ← FLGR
LDD **LDDB**	dst,src,R	IR	20						**Load and Decrement** dst ← src Autodecrement dst and src addresses R ← R + 1
LDDR **LDDRB**	dst,src,R	IR	(11 + 9 n)						**Load, Decrement and Repeat** dst ← src Autodecrement dst and src addresses R ← R − 1 Repeat until R = 0
LDI **LDIB**	dst,src,R	IR	20						**Load and Increment** dst ← src Autoincrement dst and src addresses R ← R − 1
LDIR **LDIRB**	dst,src,R	IR	(11 + 9 n)						**Load, Increment and Repeat** dst ← src Autoincrement dst and src addresses R ← R − 1 Repeat until R = 0

Mnemonics	Operands	Addr. Modes	Clock Cycles: Word, Byte NS	SS	SL	Long Word NS	SS	SL	Operation
LDK	R,src	IM	5	5	5				**Load Constant** R ← n (n = 0...15)
LDM	R,src,n	IR	11						**Load Multiple** dst ← src (n consecutive words) (n = 1...16)
		DA	14	15	17	+ 3n			
		X	15	15	18				
LDM	dst,R,n	IR	11						**Load Multiple** (Store Multiple) dst ← R (n consecutive words) (n = 1...16)
		DA	14	15	17	+ 3n			
		X	15	15	18				
LDPS*	src	IR	12						**Load Program Status** PS ← src
		DA	16	20	22				
		X	17	20	23				
LDR **LDRB**	R,src	RA	14	14	14	17	17	17	**Load Relative** R ← src (range -32768... + 32767)
LDR **LDRB** **LDRL**	dst,R	RA	14	14	14	17	17	17	**Load Relative** (Store Relative) dst ← R (range -32768... + 32767)
MBIT*			7	7	7				**Test Multi-Micro Bit** Set if M_I is Low; reset S if M_I is High.
MREQ*	dst	R	(12 + 7n)						**Multi-Mircre Request**
MRES*			5	5	5				**Multi-Micro Reset**
MSET*			5	5	5				**Multi-Micro Set**
MULT **MULTL**	R,src	R	70	70	70	282 +	282 +	282 +	**Multiply** (signed) Word: $R_{n,n+1} - R_{n+1} \cdot$ src Long Word: $R_{n...n+3} - R_{n+2,\ n+3} \cdot$ src + Plus seven cycles for each 1 in the absolute value of the low order 16 bits of the multiplicand.
		IM	70	70	70	282 +	282 +	282 +	
		IR	70			282 +			
		DA	71	72	74	283 +	283 +	286 +	
		X	72	72	75	284 +	284 +	287 +	
NEG **NEGB**	dst	R	7	7	7				**Negate** dst ← 0 − dst
		IR	12						
		DA	15	16	18				
		X	16	16	19				
NOP			7	7	7				**No Operation**
OR **ORB**	R,src	R	4	4	4				**OR** R ← R OR src
		IM	7	7	7				
		IR	7						
		DA	9	10	12				
		X	10	10	13				
OTDR* **OTDRB***	dst,src,r	IR	(11 + 10 n)						**Output, Decrement and Repeat** dst ← src Autodecrement src address R ← R − 1 Repeat until R = 0

***Privileged instructions. Executed in system mode only.**

Mnemonics	Operands	Addr. Modes	Clock Cycles: Word, Byte NS	SS	SL	Long Word NS	SS	SL	Operation
OTIR* **OTIRB***	dst,src,R	IR	(11 + 10 n)						**Output, Increment and Repeat** dst ← src Autoincrement scr address R ← R − 1 Repeat until R = 0
OUT* **OUTB***	dst,R	IR	10						**Output** dst ← R
		DA	12	12	12				
OUTD* **OUTDB***	dst,src,R	IR	21						**Output and Decrement** dst ← src Autodecrement src address R ← R − 1
OUTI* **OUTIB***	dst,src,R	IR	21						**Output and Increment** dst ← src Autoincrement src address R ← R − 1
POP **POPL**	dst,IR	R	8	8	8	12	12	12	**Pop** dst ← IR Autoincrement contents of R
		IR	12			19			
		DA	16	16	18	23	23	25	
		X	16	16	19	23	23	26	
PUSH **PUSHL**	IR,src	R	9	9	9	12	12	12	**Push** Autodecrement contents of R IR ← src
		IM	12	12	12				
		IR	13			20			
		DA	14	14	16	16	21	23	
		X	14	14	17	21	21	24	
RES **RESB**	dst,b	R	4	4	4				**Reset Bit Static** Reset dst bit specified by b
		IR	11						
		DA	13	14	16				
		X	14	14	17				
RES **RESB**	dst,R	R	10	10	10				**Reset Bit Dynamic** Reset dst bit specified by contents R
RESFLG	flag		7	7	7				**Reset Flag** (Any combination of C, Z, S, P/V)
RET	cc		10	10	13	(taken)			**Return Conditional** If cc is true: PC ← @ SP Autoincrement SP
			7	7	7	(not taken)			
RL **RLB**	dst,n	R	6 for n = 1						**Rotate Left** by n bits (n = 1, 2)
		R	7 for n = 2						
RLC **RLCB**	dst,n	R	6 for n = 1						**Rotate Left through Carry** by n bits (n = 1, 2)
		R	7 for n = 2						
RLDB	R,src	R	9	9	9				**Rotate Digit Left**
RR **RRb**	dst,n	R	6 for n = 1						**Rotate Right** by n bits (n = 1, 2)
		R	7 for n = 2						
RRC **RRCB**	dst,n	R	6 for n = 1						**Rotate Right through Carry** by n bits (n = 1, 2)
		R	7 for n = 2						

***Privileged instruction. Executed in system mode only.**

Mnemonics	Operands	Addr. Modes	Clock Cycles: Word, Byte NS	SS	SL	Long Word NS	SS	SL	Operation
RRDB	R,src	R	9	9	9				**Rotate Digit Right**
SBC **SBCB**	R,src	R	5	5	5				**Subtract with Carry** R ← R − src − carry
SC	src	IM	33		39				**System Call** Autodecrement SP @ SP ← old PS Push instruction PS ← System Call PS
SDA **SDAB** **SDAL**	dst,R	R	(15 + 3n)			(15 + 3n)			**Shift Dynamic Arithmetic** Shift dst left or right by contents of R
SDL **SDLB** **SDLL**	dst,R	R	(15 + 3n)			(15 + 3n)			**Shift Dynamic Logical** Shift dst left or right by contents of R
SET **SETB**	dst,b	R IR DA X	4 11 13 14	4 14 14	4 16 17				**Set Bit Static** Set dst bit specified by b
SET **SETB**	dst,R	R	10	10	10				**Set Bit Dynamic** Set dst bit specified by contents of R
SETFLG	flag	X	7	7	7				**Set Flag** (Any combination of C, Z, S, P/V)
SIN* **SINB***	R,src	DA	12	12	12				**Special Input** R ← src
SIND* **SINDB***	dst,src,R	IR	21						**Special Input and Decrement** dst ← src Autodecrement dst address R ← R − 1
SINDR* **SINDRB***	dst,src,R	IR	(11 + 10n)						**Special Input, Decrement and Repeat** dst ← src Autodecrement dst address R ← R − 1 Repeat until R = 0
SINI* **SINIB***	dst,src,R	IR	21						**Special Input and Increment** dst ← src Autoincrement dst address R ← R − 1
SINIR* **SINIRB***	dst,src,R	IR	(11 + 10n)						**Special Input, Increment and Repeat** dst ← src Autoincrement dst address R ← R − 1 Repeat until R = 0
SLA **SLAB** **SLAL**	dst,n	R	(13 + 3n)			(13 + 3n)			**Shift Left Arithmetic** by n bits
SLL **SLLB** **SLLL**	dst,n	R	(13 + 3n)			(13 + 3n)			**Shift Left Logical** by n bits

***Privileged instruction. Executed in system mode only.**

Mnemonics	Operands	Addr. Modes	Clock Cycles: Word, Byte NS	SS	SL	Long Word NS	SS	SL	Operation
SOTDR* **SOTDRB***	dst,src,R	IR	(11 + 10 n)						**Special Output, Decr. and Repeat** dst ← src Autodecrement src address R ← R − 1 Repeat until R = 0
SOTIR* **SOTIRB***	dst,src,R	R	(11 + 10 n)						**Special Output, Incr. and Repeat** dst ← src Autoincrement src address R ← R − 1 Repeat until R = 0
SOUT* **SOUTB***	dst,src	DA	12	12	12				**Special Output** dst ← src
SOUTD* **SOUTDB***	dst,src,R	IR	21						**Special Output and Decrement** dst ← src Autodecrement src address R ← R − 1
SOUTI* **SOUTIB***	dst,src,R	IR	21						**Special Output and Increment** dst ← src Autoincrement src address R ← R + 1
SRA **SRAB** **SRAL**	dst,n	R	(13 + 3 n)			(13 + 3 n)			**Shift Right Arithmetic** by n bits
SRL **SRLB** **SRLL**	dst,n	R	(13 + 3 n)			(13 + 3 n)			**Shift Right Logical** by n bits
SUB **SUBB** **SUBL**	R,src	R	4	4	4	8	8	8	**Subtract** R ← R − src
		IM	7	7	7	14	14	14	
		IR	7			14			
		DA	9	10	12	15	16	18	
		X	10	10	13	16	16	19	
TCC **TCCB**	cc,dst	R	5	5	5				**Test Condition Code** Set LSB if cc is true
TEST **TESTB**	dst	R	7	7	7	13	13	13	**Test** dst OR 0
		IR	8			13			
		DA	11	12	14	16	17	19	
		X	12	12	15	17	17	20	

***Privileged instructions. Executed in system mode only.**

Mnemonics	Operands	Addr. Modes	Clock Cycles: Word, Byte NS	SS	SL	Long Word NS	SS	SL	Operation
TRDB	dst,src,R	IR	25						**Translate and Decrement** dst ← src(dst) Autodecrement dst address R ← R − 1
TRDRB	dst,src,R	IR	(11 + 14n)						**Translate, Decrement and Repeat** dst ← src(dst) Autodecrement dst address R ← R − 1 Repeat until R = 0
TRIB	dst,src,R	IR	25						**Translate and Increment** dst ← src(dst) Autoincrement dst address R ← R − 1
TRIRB	dst,src,R	IR	(11 + 14n)						**Translate, Increment and Repeat** dst ← src(dst) Autoincrement dst address R ← R − 1 Repeat until R = 0
TRTDB	src1,src2,R	IR	25						**Translate and Test, Decrement** RH1 ← src2 (src1) Autodecrement src 1 address R ← R − 1
TRTDRB	src1,src2,R	IR	(11 + 14n)						**Translate and Test, Decr. and Repeat** RH1 ← src2 (src1) Autodecrement src1 address R ← R − 1 Repeat until R = 0 or RH1 ≠ 0
TRTIB	src1,src2,R	IR	25						**Translate and Test, Increment** RH1 ← src2 (src1) Autoincrement src address R ← R − 1
TRTIRB	src1,src2,R	IR	(11 + 14n)						**Translate and Test, Incr. and Repeat** RH1 ← src2 (src1) Autoincrement src 1 address R ← R1 Repeat until R = 0 or RH1 ≠ 0
TSET **TSETB**	dst	R IR DA X	7 11 14 15	7 15 15	7 17 18				**Test and Set** S flag ← MSB of dst dst ← all 1s
XOR **XORB**	R,src	R IM IR DA X	4 7 7 9 10	4 7 10 10	4 7 12 13				**Exclusive OR** R ← R XOR src

LOWER NIBBLE (HEX), UPPER INSTRUCTION BYTE

UPPER NIBBLE (HEX), UPPER INSTRUCTION BYTE	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	ADDB R ← IR R ← IM	ADD R ← IR R ← IM	SUBB R ← IR R ← IM	SUB R ← IR R ← IM	ORB R ← IR R ← IM	OR R ← IR R ← IM	ANDB R ← IR R ← IM	AND R ← IR R ← IM	XORB R ← IR R ← IM	XOR R ← IR R ← IM	CPB R ← IR R ← IM	CP R ← IR R ← IM	See Table 1	See Table 1	EXTEND INST	EXTEND INST
1	CPL R ← IR R ← IM	PUSHL IR ← IR	SUBL R ← IR R ← IM	PUSH IR ← IR	LDL R ← IR R ← IM	POPL IR ← IR	ADDL R ← IR R ← IM	POP IR ← IR	MULTL R ← IR R ← IM	MULT R ← IR R ← IM	DIVL R ← IR R ← IM	DIV R ← IR R ← IM	See Table 2	LDL IR ← R	JP PC ← IR	CALL PC ← IR
2	LDB R ← IR R ← IM	LD R ← IR R ← IM	RESB IR ← IM R ← R	RES IR ← IM R ← R	SETB IR ← IM R ← R	SET IR ← IM R ← R	BITB IR ← IM R ← R	BIT IR ← IM R ← R	INCB IR ← IM	INC IR ← IM	DECB IR ← IM	DEC IR ← IM	EXB R ← IR	EX R ← IR	LDB IR ← R	LD IR ← R
3	LDB R ← BA LDRB R ← RA	LD R ← BA LDR R ← RA	LDB BA ← R LDRB RA ← R	LD BA ← R LDR RA ← R	LDA R ← BA LDAR R ← RA	LDL R ← BA LDRL R ← RA	RSVD	LDL BA ← R LDRL RA ← R	RSVD	LDPS IR	See Table 3A	See Table 3B	INB R ← IR	IN R ← IR	OUTB IR ← R	OUT IR ← R
4	ADDB R ← X R ← DA	ADD R ← X R ← DA	SUBB R ← X R ← DA	SUB R ← X R ← DA	ORB R ← X R ← DA	OR R ← X R ← DA	ANDB R ← X R ← DA	AND R ← X R ← DA	XORB R ← X R ← DA	XOR R ← X R ← DA	CPB R ← X R ← DA	CP R ← X R ← DA	See Table 1	See Table 1	EXTEND INST	EXTEND INST
5	CPL R ← X R ← DA	PUSHL IR ← X IR ← DA	SUBL R ← X R ← DA	PUSH IR ← X IR ← DA	LDL R ← X R ← DA	POPL IR ← X IR ← DA	ADDL R ← X R ← DA	POP IR ← X IR ← DA	MULTL R ← X R ← DA	MULT R ← X R ← DA	DIVL R ← X R ← DA	DIV R ← X	See Table 2	LDL X ← R DA ← R	JP PC ← X PC ← DA	CALL PC ← X PC ← DA
6	LDB R ← X R ← DA	LD R ← X R ← DA	RESB X ← IM DA ← IM	RES X ← IM DA ← IM	SETB X ← IM DA ← IM	SET X ← IM DA ← IM	BITB X ← IM DA ← IM	BIT X ← IM DA ← IM	INCB X ← IM DA ← IM	INC X ← IM DA ← IM	DECB X ← IM DA ← IM	DEC X ← IM DA ← IM	EXB R ← X R ← DA	EX R ← X R ← DA	LDB X ← R DA ← R	LD X ← R DA ← R
7	LDB R ← BX	See Table 7	LDB BX ← R	LD BX ← R	LDA R ← BX	LDL R ← BX	LDA R ← X R ← DA	LDL BX ← R	RSVD	LDPS PS ← X PS ← DA	HALT	See Table 7	EI DI	See Table 7	RSVD	SC
8	ADDB R ← R	ADD R ← R	SUBB R ← R	SUB R ← R	ORB R ← R	OR R ← R	ANDB R ← R	AND R ← R	XORB R ← R	XOR R ← R	CPB R ← R	CP R ← R	See Table 1	See Table 1	EXTEND INST.	EXTEND INST.
9	CPL R ← R	PUSHL IR ← R	SUBL R ← R	PUSH IR ← R	LDL R ← R	POPL R ← IR	ADDL R ← R	POP R ← IR	MULTL R ← R	MULT R ← R	DIVL R ← R	DIV R ← R	See Table 2	RSVD	RET PC ← (SP)	RSVD
A	LDB R ← R	LD R ← R	RESB R ← IM	RES R ← IM	SETB R ← IM	SET R ← IM	BITB R ← IM	BIT R ← IM	INCB R ← IM	INC R ← IM	DECB R ← IM	DEC R ← IM	EXB R ← R	EX R ← R	TCCB R	TCC R
B	DAB R	EXTS EXTSB EXTSL R	See Table 4	See Table 4	ADCB R ← R	ADC R ← R	SBCB R ← R	SBC R ← R	See Table 5	RSVD	See Table 6	See Table 6	RRDB R	LDK R ← IM	RLDB R	RSVD
C	LDB R ← IM															
D	CALR PC ← RA															
E	JR PC ← RA															
F	DJNZ DBJNZ PC ← RA															

Op Code Map

Notes:

1) Reserved Instructions (RSVD) should not be used. The result of their execution is not defined.

2) The execution of an extended instruction will result in an Extended Instruction Trap if the EPA bit in the FCW is a zero. If the flag is a one the Extended Instruction will be executed by the EPU function.

LOWER NIBBLE (HEX), LOWER INSTRUCTION BYTE	OC	OD	4C	4D	8C	8D
0	**COMB** IR	**COM** IR	**COMB** X DA	**COM** X DA	**COMB** R	**COM** R
1	**CPB** IR,IM	**CP** IR,IM	**CPB** X,IM DA,IM	**CP** X,IM DA,IM	**LDCTLB** R←FLGS	**SETFLG**
2	**NEGB** IR	**NEG** IR	**NEGB** X DA	**NEG** X DA	**NEGB** R	**NEG** R
3	RSVD	RSVD	RSVD	RSVD	RSVD	**RESFLG**
4	**TESTB** IR	**TEST** IR	**TESTB** X DA	**TEST** X DA	**TESTB** R	**TEST** R
5	**LDB** IR←IM	**LD** IR←IM	**LDB** X←IM DA←IM	**LD** X←IM DA←IM	RSVD	**COMFLG**
6	**TSETB** IR	**TSET** IR	**TSETB** X DA	**TSET** X DA	**TSETB** R	**TSET** R
7	RSVD	RSVD	RSVD	RSVD	RSVD	**NOP**
8	**CLRB** IR	**CLR** IR	**CLRB** X DA	**CLR** X DA	**CLRB** R	**CLR** R
9		**PUSH** IM			**LDCTLB** FLGS←R	

Table 1. Upper Instruction Byte

LOWER NIBBLE (HEX), LOWER INSTRUCTION BYTE	1C	5C	9C
0	RSVD	RSVD	RSVD
1	**LDM** R←IR	**LDM** R←X R←DA	
8	**TESTL** IR	**TESTL** X DA	**TESTL** R
9	**LDM** IR←R	**LDM** X←R DA←R	

Table 2. Upper Instruction Byte

LOWER NIBBLE (HEX), LOWER INSTRUCTION BYTE	3A	3B
0	**INIB** IR←IR **INIRB** IR←IR	**INI** IR←IR **INIR** IR←IR
1	**SINIB** IR←IR **SINIRB** IR←IR	**SINI** IR←IR **SINIR** IR←IR
2	**OUTIB** IR←IR **OTIRB** IR←IR	**OUTI** IR←IR **OTIR** IR←IR
3	**SOUTIB** IR←IR **SOTIRB** IR←IR	**SOUTI** IR←IR **SOTIR** IR←IR
4	**INB** R←DA	**IN** R←DA
5	**SINB** R←DA	**SIN** R←DA
6	**OUTB** DA←R	**OUT** DA←R
7	**SOUTB** DA←R	**SOUT** DA←R
8	**INDB** IR←IR **INDRB** IR←IR	**IND** IR←IR **INDR** IR←IR
9	**SINDB** IR←IR **SINDRB** IR←IR	**SIND** IR←IR **SINDR** IR←IR
A	**OUTDB** IR←IR **OTDRB** IR←IR	**OUTD** IR←IR **OTDR** IR←IR
B	**SOUTDB** IR←IR **SOTDRB** IR←IR	**SOUTD** IR←IR **SOTDR** IR←IR

Table 3. Upper Instruction Byte

LOWER NIBBLE (HEX), LOWER INSTRUCTION BYTE

	B2	B3
0	**RLB** (1 bit) R	**RL** (1 bit) R
1	**SLLB** R **SRLB** R	**SLL** R **SRL** R
2	**RLB** (2 bits) R	**RL** (2 bits) R
3	**SDLB** R	**SDL** R
4	**RRB** (1 bit) R	**RR** (1 bit) R
5	RSVD	**SLLL** R **SRLL**
6	**RRB** (2 bits) R	**RR** (2 bits) R
7	RSVD	**SDLL** R
8	**RLCB** (1 bit) R	**RLC** (1 bit) R
9	**SLAB** R **SRAB** R	**SLA** R **SRA** R
A	**RLCB** (2 bits) R	**RLC** (2 bits) R
B	**SDAB** R	**SDA** R
C	**RRCB** (1 bit) R	**RRC** (1 bit) R
D	RSVD	**SLAL** R **SRAL**
E	**RRCB** (2 bits) R	**RRC** (2 bits) R
F	RSVD	**SDAL** R

Table 4.
Upper Instruction Byte

LOWER NIBBLE (HEX), LOWER INSTRUCTION BYTE

	B8
0	**TRIB** IR
1	RSVD
2	**TRTIB** IR
3	RSVD
4	**TRIRB** IR
5	RSVD
6	**TRTIRB** IR
7	RSVD
8	**TRDB** IR
9	RSVD
A	**TRTDB** IR
B	RSVD
C	**TRDRB** IR
D	RSVD
E	**TRTDRB** IR
F	RSVD

Table 5.
Upper Instruction Byte

	BA	BB
0	**CPIB** IR	**CPI** IR
1	**LDIB** IR→IR **LDIRB** IR→IR	**LDI** IR→IR **LDIR** IR→IR
2	**CPSIB** IR	**CPSI** IR
3	RSVD	RSVD
4	**CPRIB** IR	**CPIR** IR
5	RSVD	RSVD
6	**CPSIRB** IR	**CPSIR** IR
7	RSVD	RSVD
8	**CPDB** IR	**CPD** IR
9	**LDDB** IR→IR **LDDRB** IR→IR	**LDD** IR→IR **LDDR** IR→IR
A	**CPSDB** IR	**CPSD** IR
B	RSVD	RSVD
C	**CPDRB** IR	**CPDR** IR
D	RSVD	RSVD
E	**CPSDRB** IR	**CPSDR** IR
F	RSVD	RSVD

Table 6.
Upper Instruction Byte

	7B	7D
0	**IRET** PC←(SSP)	RSVD
1	RSVD	RSVD
2	RSVD	**LDCTL** R←FCW
3	RSVD	**LDCTL** R←RFRSH
4	RSVD	**LDCTL** R← PSAPSEG
5	RSVD	**LDCTL** R← PSAPOFF
6	RSVD	**LDCTL** R←NSPSEG
7	RSVD	**LDCTL** R←NSPOFF
8	**MSET**	RSVD
9	**MRES**	RSVD
A	**MBIT**	**LDCTL** FCW←R
B	RSVD	**LDCTL** RFRSH←R
C	↓	**LDCTL** PSAPSEG ←R
D	**MREQ** R	**LDCTL** PSAPOFF ←R
E	RSVD	**LDCTL** NSPSEG←R
F	RSVD	**LDCTL** NSPOFF←R

Table 7.
Upper Instruction Byte

Topical Index

Instruction Description	Mnemonic	Data Types	Addressing Modes	Flags Affected
Arithmetic				
Add with Carry	ADC	B, W	R	C, Z, S, V, D[1], H[1]
Add	ADD	B, W, L	R, IM, IR, DA, X	C, Z, S, V, D[1], H[1]
Compare (Immediate)	CP	B, W	IR, DA, X	C, Z, S, V
Compare (Register)	CP	B, W, L	R, IM, IR, DA, X	C, Z, S, V
Decimal Adjust Bit	DAB	B	IR	C, Z, S
Decrement	DEC	B, W	R, IR, DA, X	Z, S, V
Divide	DIV	W, L	R, IM, IR, DA, X	C, Z, S, V
Extend Sign	EXTS	B, W, L	R	C, Z, S, V
Increment	INC	B, W	R, IR, DA, X	Z, S, V
Multiply	MULT	W, L	R, IM, IR, DA, X	C, Z, S, V
Negate	NEG	B, W	R, IR, DA, X	C, Z, S, V
Subtract with Carry	SBC	B, W	R	C, Z, S, V, D[1], H[1]
Subtract	SUB	B, W, L	R, IM, IR, DA, X	C, Z, S, V, D[1], H[1]
Bit Manipulation				
Bit Test	BIT	B, W	R	Z
Bit Reset (Static)	RES	B, W	R, IR, DA, X	—
Bit Reset (Dynamic)	RES	B, W	R	—
Bit Set (Static)	SET	B, W	R, IR, DA, X	—
Bit Set (Dynamic)	SET	B, W	R	—
Bit Test and Set	TSET	B, W	R, IR, DA, X	S
Block Transfer and String Manipulation				
Compare and Decrement	CPD	B, W	IR	C, Z, S, V
Compare, Decrement, and Repeat	CPDR	B, W	IR	C, Z, S, V
Compare and Increment	CPI	B, W	IR	C, Z, S, V
Compare, Increment, and Repeat	CPIR	B, W	IR	C, Z, S, V
Compare String and Decrement	CPSD	B, W	IR	C, Z, S, V
Compare String, Decrement, and Repeat	CPSDR	B, W	IR	C, Z, S, V
Compare String and Increment	CPSI	B, W	IR	C, Z, S, V
Compare String, Increment, and Repeat	CPSIR	B, W	IR	C, Z, S, V
Load and Decrement	LDD	B, W	IR	V
Load, Decrement, and Repeat	LDDR	B, W	IR	V
Load and Increment	LDI	B, W	IR	V
Load, Increment, and Repeat	LDIR	B, W	IR	V
Translate and Decrement	TRDB	B	IR	Z, V
Translate, Decrement, and Repeat	TRDRB	B	IR	Z, V
Translate and Increment	TRIB	B	IR	Z, V
Translate, Increment, and Repeat	TRIRB	B	IR	Z, V
Translate, Test, and Decrement	TRTDB	B	IR	Z, V
Translate, Test, Decrement, Repeat	TRTDRB	B	IR	Z, V
Translate, Test, and Increment	TRTIB	B	IR	Z, V
Translate, Test, Increment, and Repeat	TRTIRB	B	IR	Z, V
CPU Control Instructions				
Complement Flag	COMFLG	—	—	C[2], Z[2], S[2], P[2], V[2]
Disable Interrupt	DI	—	—	—
Enable Interrupt	EI	—	—	—
Halt	HALT	—	—	—
Load Control Register (from register)	LDCTL	—	R	C[2], Z[2], S[2], P[2], D[2], H[2]
Load Control Register (to register)	LDCTL	—	—	—
Load Program Status	LDPS	—	IR, DA, X	C, Z, S, P, D, H
Multi-Bit Test	MBIT	—	—	S
Multi-Micro Request	MREQ	—	—	Z, S
Multi-Micro Reset	MRES	—	—	—
Multi-Micro Set	MSET	—	—	—
No Operation	NOP	—	—	—
Reset Flag	RESFLG	—	—	C[2], Z[2], S[2], P[2], V[2]
Set Flag	SETFLG	—	—	C[2], Z[2], S[2], P[2], V[2]

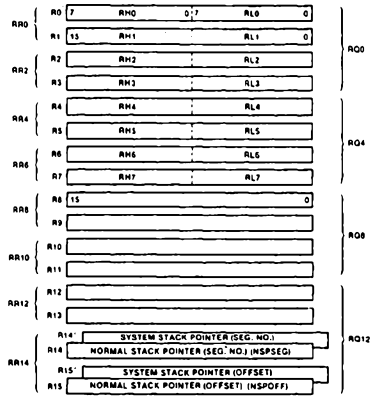
1. Flag affected only for byte operation.

2. Flag modified only if specified by the instruction.

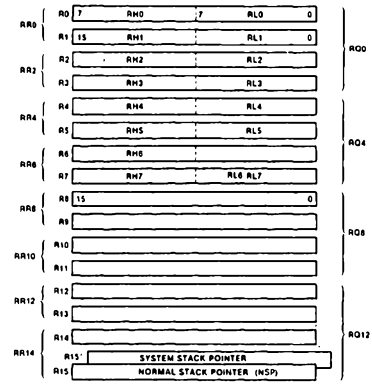
Topical Index (Continued)

Instruction Description	Mnemonic	Data Types	Addressing Modes		Flags Affected
Input/Output Instructions[3]			**Regular**	**Special**	
Input	(S)IN[3]	B, W	IR, DA	(DA)	—
Input and Decrement	(S)IND[3]	B, W	IR	(IR)	V
Input, Decrement and Repeat	(S)INDR[3]	B, W	IR	(IR)	V
Input and Increment	(S)INI[3]	B, W	IR	(IR)	V
Input, Increment, and Repeat	(S)INIR[3]	B, W	IR	(IR)	V
Output	(S)OUT[3]	B, W	IR, DA	(DA)	—
Output and Decrement	(S)OUTD[3]	B, W	IR	(IR)	V
Output, Decrement, and Repeat	(S)OUTDR[3]	B, W	IR	(IR)	V
Output and Increment	(S)OUTI[3]	B, W	IR	(IR)	V
Output, Increment, and Repeat	(S)OUTIR[3]	B, W	IR	(IR)	V
Logical Instructions					
And	AND	B, W	R, IM, IR, DA, X		Z, S, P
Complement	COM	B, W	R, IR, DA, X		Z, S, P
Or	OR	B, W	R, IM, IR, DA, X		Z, S, P
Test	TEST	B, W, L	R, IR, DA, X		Z, S, P
Test Condition Code	TCC	B, W	R		—
Exclusive Or	XOR	B, W	R, IM, IR, DA, X		Z, S, P
Program Control Instructions					
Call Procedure	CALL	—	IR, DA, X		—
Call Procedure Relative	CALR	—	RA		—
Decrement, Jump if Not Zero	DJNZ	B, W	RA		—
Interrupt Return	IRET	—	—		C, Z, S, P, D, H
Jump	JP	—	IR, DA, X		—
Jump Relative	JR	—	RA		—
Return From Procedure	RET	—	—		—
System Call	SC	—	—		—
Rotate and Shift Instructions					
Rotate Left	RL	B, W	R		—
Rotate Left Through Carry	RLC	B, W	R		C, Z, S, V
Rotate Left Digit	RLDB	B	R		Z, S
Rotate Right	RR	B, W	R		C, Z, S, V
Rotate Right Through Carry	RRC	B, W	R		C, Z, S, V
Rotate Right Digit	RRDB	B	R		Z, S
Shift Dynamic Arithmetic	SDA	B, W, L	R		C, Z, S, V
Shift Dynamic Logical	SDL	B, W, L	R		C, Z, S, V
Shift Left Arithmetic	SLA	B, W, L	R		C, Z, S, V
Shift Left Logical	SLL	B, W, L	R		C, Z, S, V
Shift Right Arithmetic	SRA	B, W, L	R		C, Z, S, V
Shift Right Logical	SRL	B, W, L	R		C, Z, S, V

3. Each I/O instruction has a Special counterpart used to alert other devices that a Special I/O transaction is occurring. The Special I/O mnemonic is S + Regular mnemonic. Refer to section 6.2.8 for further details.



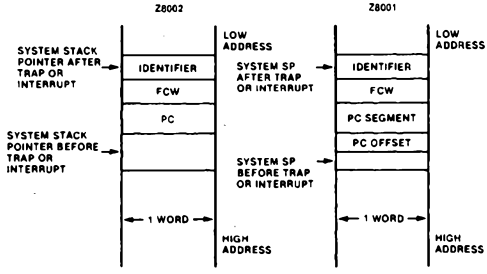
Z8001 General Purpose Registers



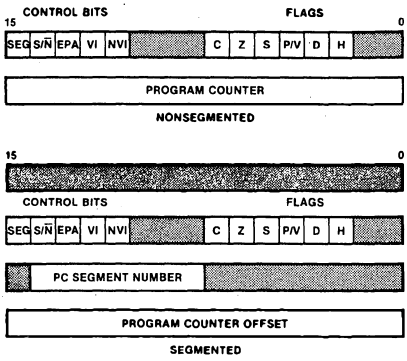
Z8002 General Purpose Registers

Register				Binary	Hex
RQ0	RR0	R0	RH0	0000	0
		R1	RH1	0001	1
	RR2	R2	RH2	0010	2
		R3	RH3	0011	3
RQ4	RR4	R4	RH4	0100	4
		R5	RH5	0101	5
	RR6	R6	RH6	0110	6
		R7	RH7	0111	7
RQ8	RR8	R8	RL0	1000	8
		R9	RL1	1001	9
	RR10	R10	RL2	1010	A
		R11	RL3	1011	B
RQ12	RR12	R12	RL4	1100	C
		R13	RL5	1101	D
	RR14	R14	RL6	1110	E
		R15	RL7	1111	F

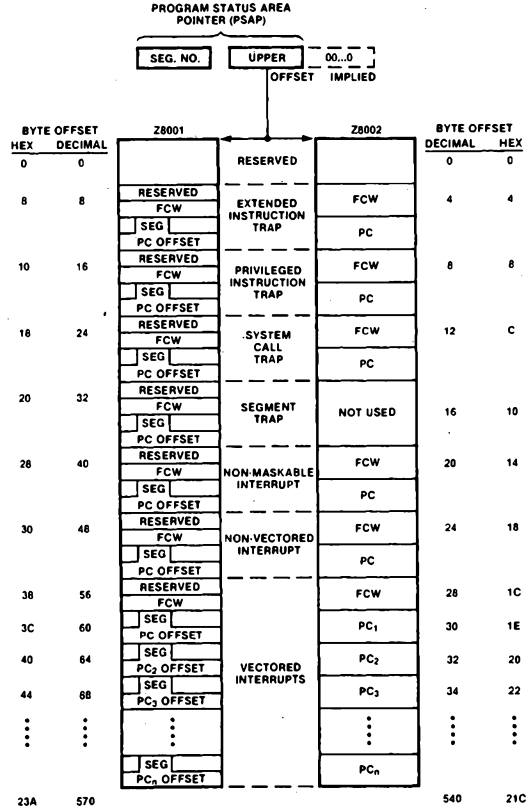
Binary Encoding for Register Fields



Format of Saved Program Status in the System Stack



Program Status Blocks



Program Status Area

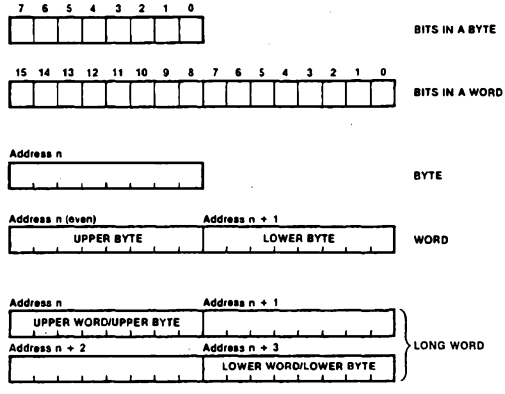
Condition Codes

Code	Meaning	Flag Setting	Binary
F	Always false*	-	0000
	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

This table provides the condition codes and the flag settings they represent.

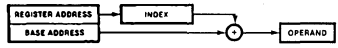
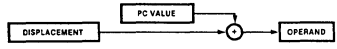
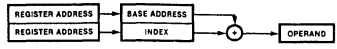
Note that some of the condition codes correspond to identical flag settings: i.e., Z-EQ, NZ-NE, NC-UGE, PE-OV, PO-NOV.

*Presently not implemented in PLZ/ASM Z8000 compiler.



Addressable Data Elements

Z8000 Addressing Modes

Addressing Mode	Operand Addressing			Operand Value
	In the Instruction	In a Register	In Memory	
R				
Register				The content of the register
IM				
Immediate				In the instruction
***IR**				
Indirect Register				The content of the location whose address is in the register
DA				
Direct Address				The content of the location whose address is in the instruction
***X**				
Index				The content of the location whose address is the address in the instruction plus the content of the working register.
RA				
Relative Address				The content of the location whose address is the content of the program counter, offset by the displacement in the instruction
***BA**				
Base Address				The content of the location whose address is the address in the register, offset by the displacement in the instruction
***BX**				
Base Index				The content of the location whose address is the address in a register plus the index value in another register.

*Do not use R0 or RR0 as indirect, index, or base registers.

Powers of 2 and 16

2^n	n
256	8
512	9
1 024	10
2 048	11
4 096	12
8 192	13
16 384	14
32 768	15
65 536	16
131 072	17
262 144	18
524 288	19
1 048 576	20
2 097 152	21
4 194 304	22
8 388 608	23
16 777 216	24

Powers of 2

2^0	$= 16^0$
2^4	$= 16^1$
2^8	$= 16^2$
2^{12}	$= 16^3$
2^{16}	$= 16^4$
2^{20}	$= 16^5$
2^{24}	$= 16^6$
2^{28}	$= 16^7$
2^{32}	$= 16^8$
2^{36}	$= 16^9$
2^{40}	$= 16^{10}$
2^{44}	$= 16^{11}$
2^{48}	$= 16^{12}$
2^{52}	$= 16^{13}$
2^{56}	$= 16^{14}$
2^{60}	$= 16^{15}$

16^n	n
1	0
16	1
256	2
4 096	3
65 536	4
1 048 576	5
16 777 216	6
268 435 456	7
4 294 967 296	8
68 719 476 736	9
1 099 511 627 776	10
17 592 186 044 416	11
281 474 976 710 656	12
4 503 599 627 370 496	13
72 057 594 037 927 936	14
1 152 921 504 606 846 976	15

Powers of 16

8		7		6		5		4		3		2		1	
Hex	Decimal	Hex	Decimal	Hex	Decimal	Hex	Decimal	Hex	Decimal	Hex	Decimal	Hex	Decimal	Hex	Decimal
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	268,435,456	1	16,777,216	1	1,048,576	1	65,536	1	4,096	1	256	1	16	1	1
2	536,870,912	2	33,554,432	2	2,097,152	2	131,072	2	8,192	2	512	2	32	2	2
3	805,306,368	3	50,331,648	3	3,145,728	3	196,608	3	12,288	3	768	3	48	3	3
4	1,073,741,824	4	67,108,864	4	4,194,304	4	262,144	4	16,384	4	1,024	4	64	4	4
5	1,342,177,280	5	83,886,080	5	5,242,880	5	327,680	5	20,480	5	1,280	5	80	5	5
6	1,610,612,736	6	100,663,296	6	6,291,456	6	393,216	6	24,576	6	1,536	6	96	6	6
7	1,879,048,192	7	117,440,512	7	7,340,032	7	458,752	7	28,672	7	1,792	7	112	7	7
8	2,147,483,648	8	134,217,728	8	8,388,608	8	524,288	8	32,768	8	2,048	8	128	8	8
9	2,415,919,104	9	150,994,944	9	9,437,184	9	589,824	9	36,864	9	2,304	9	144	9	9
A	2,684,354,560	A	167,772,160	A	10,485,760	A	655,360	A	40,960	A	2,560	A	160	A	10
B	2,952,790,016	B	184,549,376	B	11,534,336	B	720,896	B	45,056	B	2,816	B	176	B	11
C	3,221,225,472	C	201,326,592	C	12,582,912	C	786,432	C	49,152	C	3,072	C	192	C	12
D	3,489,660,928	D	218,103,808	D	13,631,488	D	851,968	D	53,248	D	3,328	D	208	D	13
E	3,758,096,384	E	234,881,024	E	14,680,064	E	917,504	E	57,344	E	3,584	E	224	E	14
F	4,026,531,840	F	251,658,240	F	15,728,640	F	983,040	F	61,440	F	3,840	F	240	F	15
	8		7		6		5		4		3		2		1

Hexadecimal and Decimal Interger Conversion Table

To Convert Hexadecimal to Decimal

1. Locate the column of decimal numbers corresponding to the left-most digit or letter of the hexadecimal: select from this column and record the number that corresponds to the position of the hexadecimal digit or letter.
2. Repeat step 1 for the units (second from the left) position.
3. Repeat step 1 for the units (third from the left) position.
4. Add the numbers selected from the table to form the decimal number.

To convert integer numbers greater than the capacity of the table, use the techniques below:

Hexadecimal to Decimal

Succesive cumulative mulitplication from left to right, adding units position.

Example: $D34_{16} = 3380_{10}$

D = 13
× 16
208
3 = + 13
211
× 16
3376
4 = + 4
3380

Example:

Conversion of Hexadecimal Value	D34
1. D	3328
2. 3	48
3. 4	6
4. Decimal	3380

To Convert Decimal to Hexadecimal

1. (a) Select from the tabel the highest decimal number that is equal to or less than the number to be converted.
   (b) Record the hexadecimal of the column containing the selected number.
   (c) Subtract the selected decimal from the number to be converted.
2. Using the remainder from step 1(c) repeat all of step 1 to develop the second position of the hexadecimal (and a remainder).
3. Using the remainder from step 2 repeat all of step 1 to develop the units position of the hexadecimal.
4. Combine terms to form the hexadecimal number.

Decimal to Hexadecimal

Divide and collect the remainder in reverse order.

Example: $3380_{10} = D34_{16}$

16|3380 remainder
16| 211 4
16| 13 3
D

Example:

Conversion of Decimal Value	3380
1. D	- 3328
	52
2. 3	- 48
	4
3. 4	- 4
4. Hexadecimal	D34

ASCII Characters

Hexadecimal	Character	Meaning	Hexadecimal	Character
00	NUL	NULL Character	40	@
01	SOH	Start of Heading	41	A
02	STX	Start of Text	42	B
03	ETX	End of Text	43	C
04	EOT	End of Transmission	44	D
05	ENQ	Enquiry	45	E
06	ACK	Acknowledge	46	F
07	BEL	Bell	47	G
08	BS	Backspace	48	H
09	HT	Horizontal Tabulation	49	I
0A	LF	Line Feed	4A	J
0B	VT	Vertical Tabulation	4B	K
0C	FF	Form Feed	4C	L
0D	CR	Carriage Return	4D	M
0E	SO	Shift Out	4E	N
0F	SI	Shift In	4F	O
10	DLE	Data Link Escape	50	P
11	DC1	Device Control 1	51	Q
12	DC2	Device Control 2	52	R
13	DC3	Device Control 3	53	S
14	DC4	Device Control 4	54	T
15	NAK	Negative Acknowledge	55	U
16	SYN	Synchronous Idle	56	V
17	ETB	End of Transmission Block	57	W
18	CAN	Cancel	58	X
19	EM	End of Medium	59	Y
1A	SUB	Substitute	5A	Z
1B	ESC	Escape	5B	[
1C	FS	File Separator	5C	\
1D	GS	Group Separator	5D	]
1E	RS	Record Separator	5E	^
1F	US	Unit Separator	5F	_
20	SP	Space	60	`
21	!		61	a
22	"		62	b
23	#		63	c
24	$		64	d
25	%		65	e
26	&		66	f
27	'		67	g
28	(		68	h
29	)		69	i
2A	*		6A	j
2B	+		6B	k
2C	,		6C	l
2D	-		6D	m
2E	.		6E	n
2F	/		6F	o
30	0		70	p
31	1		71	q
32	2		72	r
33	3		73	s
34	4		74	t
35	5		75	u
36	6		76	v
37	7		77	w
38	8		78	x
39	9		79	y
3A	:		7A	z
3B	;		7B	{
3C	<		7C	|
3D	=		7D	}
3E	>		7E	~
3F	?		7F	DEL Delete