

Zilog

Z80,000™ CPU PRELIMINARY TECHNICAL MANUAL

Z80,000™ CPU

**Preliminary
Technical Manual**

September 1984



Z80,000™ CPU
Preliminary
Technical Manual

**Z:1
Z:100
Z:1000
Zilog**

© 1984 Zilog, Inc. All rights reserved. No part of this publication may be reproduced without the written permission of Zilog, Inc.

Copyright 1984 by Zilog, Inc. All rights reserved. No part of this publication may be reproduced without the written permission of Zilog, Inc.
The information in this publication is subject to change without notice.

Table of Contents

Chapter 1. Z80,000 CPU Overview

1.1 Introduction	1-1
1.2 Architecture	1-1
1.2.1 Registers	1-1
1.2.2 Address Spaces	1-1
1.2.3 Memory Management	1-2
1.2.4 Addressing Modes	1-3
1.2.5 Instruction Set	1-3
1.2.6 Normal and System Modes of Operation	1-3
1.2.7 Exceptions	1-3
1.3 Extended Processing Architecture	1-4
1.4 Cache	1-4
1.5 External Interface	1-4
1.6 CPU Internal Organization	1-5
1.7 Z8000 Compatibility	1-6
1.8 Summary	1-6

Chapter 2. Data Formats and Registers

2.1 Introduction	2-1
2.2 Data Formats	2-1
2.3 General-Purpose Register File	2-1
2.4 Program Status Registers	2-2
2.5 Special-Purpose Control Registers	2-3
2.5.1 Program Status Area Pointer (PSAP)	2-3
2.5.2 Normal Stack Pointer (NSP)	2-3
2.5.3 Translation Table Descriptor Registers	2-4
2.5.4 Overflow Stack Pointer (OSP)	2-4
2.5.5 Hardware Interface Control Register (HICR)	2-4
2.5.6 System Configuration Control Longword (SCCL)	2-4
2.6 Reserved Control Bits	2-4

Chapter 3. Address Representation and Modes of Operation

3.1 Introduction	3-1
3.2 Address Representation	3-1
3.3 Normal and System Modes	3-2

Chapter 4. Address Spaces and Memory Management

4.1 Introduction	4-1
4.2 Address Spaces	4-1
4.2.1 Logical Memory Address Spaces	4-1
4.2.2 Logical I/O Address Space	4-3
4.2.3 Physical Address Spaces	4-3

1

2

3

4

iii

Table of Contents (Continued)

4.3 Memory Management	4-3	4
4.3.1 Address Translation	4-4	
4.3.2 Loading the TLB	4-5	
4.3.2.1 Translation Table Descriptor Registers	4-6	
4.3.2.2 Level-1 Table Entries	4-6	
4.3.2.3 Level-2 Table Entries	4-7	
4.3.2.4 Page Table Entries	4-7	
4.3.3 Access Protection	4-8	
4.3.4 Address Translation Algorithm	4-8	
4.3.5 Address Translation Exceptions	4-10	
4.3.6 Memory Management Instructions	4-10	
<hr/>		
Chapter 5. Addressing Modes and Address Calculations		5
5.1 Introduction	5-1	
5.2 Address Calculations	5-1	
5.2.1 Compact Address Calculations	5-1	
5.2.2 Segmented Address Calculations	5-1	
5.2.3 Linear Address Calculations	5-3	
5.3 Addressing Mode Descriptions	5-4	
5.3.1 Compact Mode Descriptions and Examples	5-4	
5.3.1.1 Register (R)	5-4	
5.3.1.2 Immediate (IM)	5-4	
5.3.1.3 Indirect Register (IR)	5-4	
5.3.1.4 Direct Address (DA)	5-5	
5.3.1.5 Index (X)	5-5	
5.3.1.6 Base Address (BA)	5-6	
5.3.1.7 Base Index (BX)	5-6	
5.3.1.8 Relative Address (RA)	5-7	
5.3.1.9 Relative Index (RX)	5-7	
5.3.2 Segmented and Linear Mode Descriptions and Examples	5-7	
5.3.2.1 Register (R)	5-7	
5.3.2.2 Immediate (IM)	5-8	
5.3.2.3 Indirect Register (IR)	5-8	
5.3.2.4 Direct Address (DA)	5-9	
5.3.2.5 Index (X)	5-10	
5.3.2.6 Base Address (BA)	5-11	
5.3.2.7 Base Index (BX)	5-12	
5.3.2.8 Relative Address (RA)	5-13	
5.3.2.9 Relative Index (RX)	5-14	
5.4 Extended Addressing Modes	5-15	

Chapter 6. Instruction Set**6**

6.1 Introduction	6-1
6.2 Functional Summary	6-1
6.2.1 Load and Exchange Instructions	6-1
6.2.2 Arithmetic Instructions	6-2
6.2.3 Logical Instructions	6-3
6.2.4 Program Control Instructions	6-4
6.2.5 Bit Manipulation Instructions	6-5
6.2.6 Bit Field Instructions	6-5
6.2.7 Rotate and Shift Instructions	6-6
6.2.8 Block Transfer and String Manipulation Instructions	6-7
6.2.9 Input/Output Instructions	6-8
6.2.10 CPU Control Instructions	6-8
6.2.11 Extended Instructions	6-9
6.3 Flags and Condition Codes	6-9
6.4 Notation and Binary Encoding	6-10
6.4.1 Assembler Language Syntax	6-11
6.4.2 Instruction Format	6-12
6.4.3 Extended Addressing Modes	6-13
6.4.3.1 Compact Mode	6-13
6.4.3.2 Segmented or Linear Mode	6-13
6.4.4 Unimplemented Instruction Encodings	6-15
6.5 Z80,000 Instruction Descriptions and Formats	6-16
6.6 EPA Instruction Templates	6-209

Chapter 7. Instruction Execution and Exceptions**7**

7.1 Introduction	7-1
7.2 Operating States	7-1
7.3 Instruction Execution	7-2
7.3.1 Instruction Ending	7-2
7.3.2 Effects of the Pipeline on Execution	7-3
7.4 Exceptions	7-3
7.4.1 Reset	7-3
7.4.2 Bus Error	7-4
7.4.3 Interrupts	7-4
7.4.4 Traps	7-4
7.4.4.1 Extended Instruction Trap	7-4
7.4.4.2 Privileged Instruction Trap	7-4
7.4.4.3 System Call Trap	7-4

Table of Contents (Continued)

7.4.4.4 Address Translation Trap	7-4	7
7.4.4.5 Breakpoint Trap	7-4	
7.4.4.6 Integer Arithmetic Error Trap	7-4	
7.4.4.7 Conditional Trap	7-4	
7.4.4.8 Unimplemented Instruction Trap	7-5	
7.4.4.9 Odd PC Trap	7-5	
7.4.4.10 Trace Trap	7-5	
7.4.5 Changing Program Status	7-5	
7.4.6 Exception Handlers	7-7	
7.4.7 Priority of Exceptions	7-8	
<hr/>		
Chapter 8. External Interface		8
8.1 Introduction	8-1	8
8.2 Bus Operations	8-1	
8.3 Multiprocessor Configurations	8-2	
8.4 Cache	8-3	
8.5 Pin Functions	8-3	
8.6 Hardware Interface Control Register	8-5	
8.7 Bus Timing	8-6	
8.8 Bus Transactions	8-7	
8.8.1 Response	8-8	
8.8.2 CPU-Memory Transactions	8-8	
8.8.2.1 Single Memory Read and Write Transactions	8-8	
8.8.2.2 Burst Memory Read and Write Transactions	8-10	
8.8.2.3 Interlocked Memory Transactions	8-11	
8.8.3 Input/Output Transactions	8-12	
8.8.4 EPU Transactions	8-13	
8.8.4.1 CPU-EPU Instruction Transactions	8-16	
8.8.4.2 CPU-EPU Data Transactions	8-17	
8.8.4.3 EPU-Memory Transactions	8-18	
8.8.5 Interrupt Request and Acknowledge	8-20	
8.8.6 Internal Operation and Halt Transactions	8-21	
8.8.7 Bus Retry	8-21	
8.8.8 Bus Error	8-21	
8.9 Bus Request and Acknowledge	8-22	
8.10 Reset	8-26	
<hr/>		
Appendix A. Z8000 Compatibility	A-1	
Appendix B. Memory-Mapped I/O	B-1	
Appendix C. Cache Control and Memory Transactions	C-1	
Appendix D. Programmer's Quick Reference Guide	D-1	

Appendix E. Timing Formulae for Performance Evaluation E-1

Glossary G-1

Index I-1

List of Illustrations

Figure 1-1. Memory Mapping	1-2
Figure 1-2. Functional Block Diagram	1-6
Figure 2-1. Data Formats	2-1
Figure 2-2. General-Purpose Registers	2-2
Figure 2-3. Program Status Registers	2-3
Figure 2-4. Special-Purpose Control Registers	2-3
Figure 3-1. Address Representation	3-2
Figure 4-1. Address Spaces	4-1
Figure 4-2. Logical Memory Addresses in Compact Mode	4-2
Figure 4-3. Memory Address Space in Segmented Mode	4-2
Figure 4-4. Bytes, Words, and Longwords in Memory	4-3
Figure 4-5. Address Translation Using the TLB	4-4
Figure 4-6. Logical Address Partition for Address Translation	4-5
Figure 4-7. Automatic Loading of the TLB Using Tables in Memory	4-5
Figure 4-8. Translation Table Descriptor	4-6
Figure 4-9. Translation Table Base Address	4-7
Figure 4-10. Table Entry Formats	4-7
Figure 4-11. Address Translation Trap Identifier Word	4-10
Figure 5-1. Addressing Modes	5-2
Figure 5-2. Segmented Addresses	5-3
Figure 6-1. Bit Field	6-6
Figure 7-1. Operating States	7-1
Figure 7-2. Program Status Saved on System Stack	7-5
Figure 7-3. Program Status Area	7-6
Figure 7-4. Program Status Saved on Overflow Stack	7-7
Figure 7-5. Exception Priority Flowchart	7-8
Figure 8-1. System Configuration	8-1
Figure 8-2. Multiprocessor Configurations	8-2
Figure 8-3. Z80,000 Pin Functions	8-4
Figure 8-4. Hardware Interface Control Register	8-5
Figure 8-5. Example of Memory Read Timing Showing Different Bus Scale Factors	8-6
Figure 8-6. Single Memory Read Timing	8-8
Figure 8-7. Single Memory Read Timing (One Wait State)	8-9
Figure 8-8. Single Memory Write Timing	8-10
Figure 8-9. Burst Transfer Protocol	8-11

Table of Contents (Continued)

Figure 8-10. Burst Memory Read Timing (One Wait State)	8-12
Figure 8-11. Burst Memory Write Timing	8-13
Figure 8-12. I/O Read Timing	8-14
Figure 8-13. EPA Instruction Processing	8-15
Figure 8-14. CPU-EPU Instruction Transfer Timing	8-16
Figure 8-15. CPU-EPU Data Read Timing	8-17
Figure 8-16. CPU-EPU Data Write Timing	8-18
Figure 8-17. EPU-Memory Single Write Timing	8-19
Figure 8-18. Interrupt Response/Acknowledge Timing	8-20
Figure 8-19. Internal Operation and Halt Timing	8-21
Figure 8-20. Bus Error Identifier Word	8-21
Figure 8-21. Local Bus Request Acknowledge Timing	8-22
Figure 8-22. Global Bus Request Timing	8-23
Figure 8-23. State Diagram for CPU Bus Request Protocol	8-24
Figure 8-24. Reset Timing	8-27
Figure C-1. Cache Organization	C-1
Figure E-1. Functional Block Diagram	E-1
Figure E-2. Instruction Pipeline	E-2

Chapter 1.

Z80,000 CPU Overview

1.1 INTRODUCTION

The Z80,000 CPU is an advanced 32-bit microprocessor that integrates the architecture of a mainframe computer into a single chip. A subset of the Z80,000 architecture was originally implemented in a 16-bit version, the Z8000™ microprocessor. The Z80,000 bus structure permits the use of Z8000 family peripherals, such as the Z8030 SCC and Z8036 CIO. While maintaining compatibility with Z8000 family software and hardware, the Z80,000 CPU offers greater power and flexibility in both its architecture and interface capability. Operating systems and compilers are easily developed in the Z80,000 CPU's sophisticated environment, and the hardware interface provides for connection in a wide variety of system configurations.

Memory management is integrated in the CPU, providing access to more than 4 billion bytes of logical address space without external support components. The Z80,000 CPU also includes a cache memory, which complements the pipelined design to achieve high performance with moderate memory speeds.

This chapter presents an overview of the features of the Z80,000 CPU that offer extraordinary flexibility to microprocessor system designers in tailoring the power of the CPU to their specialized applications. The chapters that follow describe these features in detail.

1.2 ARCHITECTURE

The CPU features a general-purpose register file with sixteen 32-bit registers. The instruction set offers a regular combination of nine general addressing modes with operations on numerous data types, including bits, bit fields, bytes (8 bits), words (16 bits), longwords (32 bits), and variable-length strings. The memory management, exception handling, and system and normal mode features support the development of reliable software systems.

1.2.1 Registers

The Z80,000 CPU includes sixteen 32-bit general-purpose registers. The registers can be used as data accumulators, index values, or memory pointers. Two of the registers, the Frame Pointer and Stack Pointer, are used for procedure linkage with the Call, Enter, Exit, and Return instructions.

The Z80,000 registers also include the 32-bit Program Counter and 16-bit Flag and Control Word. These two registers, together called the Program Status, are automatically saved during trap and interrupt processing. Nine other special-purpose registers are used for memory management, system configuration, and other CPU control.

1.2.2 Address Spaces

The CPU uses 32-bit logical addresses, permitting direct access to 4G bytes of memory. The logical addresses are translated by the memory management mechanism to the physical addresses used to access memory and peripherals.

The CPU supports three modes of address representation--compact, segmented, and linear--selected by two control bits in the Flag and Control Word register. Applications with an address space smaller than 64K bytes can take advantage of the dense code and efficient use of base registers with the 16-bit compact addresses. Although programs executing in compact mode can only manipulate 16-bit addresses, the logical address is extended to 32 bits by concatenating the 16 most-significant bits of the Program Counter register. Compact mode is equivalent to the Z8000 non-segmented mode.

Segmented mode supports two segment sizes--64K bytes and 16M bytes. Up to 32,768 of the small segments and 128 of the large segments are available. In segmented mode, address calculations do not affect the segment number, only the offset

Z80,000 CPU Overview

within the segment. Allocating individual objects such as program modules, stacks, or large data structures to separate segments allows applications to benefit from the logical structure of a segmented memory space.

The 32-bit addresses in linear mode provide uniform and unstructured access to 4G bytes of memory. Some applications benefit from the flexibility of linear addressing by allocating objects to arbitrary positions in the address space.

1.2.3 Memory Management

Memory management provides two valuable functions--address translation and access protection. Access protection ensures that proprietary portions of memory, or those portions concerned with operating system functions, are protected from tampering. Address translation, the process of mapping a program's logical addresses to the physical addresses used to access memory, streamlines system performance, since the operating system can relocate programs in memory, free from rigid constraints. By integrating memory management with the processor in a single chip, the Z80,000 CPU reduces parts-count and improves memory access time.

Another memory management function, demand-paged virtual memory, allows programs to execute even when only a portion of their memory requirements is available in primary storage. The rest of the program can be stored in secondary storage, typically on disk. Thus, virtual memory improves a

system's cost/performance by permitting programs to execute with varying amounts of memory.

The CPU implements a paged translation mechanism similar to that of most mainframe and super-mini-computers. The operating system creates translation tables in memory, then loads pointers to the tables in control registers. The CPU automatically refers to the tables to perform address translation and access protection.

To manage the large logical address space, the translation scheme divides it into fixed-size, 1K-byte pages. Similarly, the physical address space is divided into fixed-size frames, also 1K-bytes each. The memory management mechanism maps a logical page to an arbitrary physical frame (Figure 1-1). Since both the pages and frames are of fixed and equal size, the operating system's memory allocation problem is simplified.

The CPU implements a Translation Lookaside Buffer (TLB) to store the information needed to translate the sixteen most recently used pages. When the information needed to translate a page is missing from the TLB, the CPU automatically translates the address using the tables in memory, and then loads the information into the TLB.

The memory management mechanism can be used to map logical memory addresses to physical I/O addresses. The use of memory-mapped I/O permits protected access by application programs to selected peripheral devices.

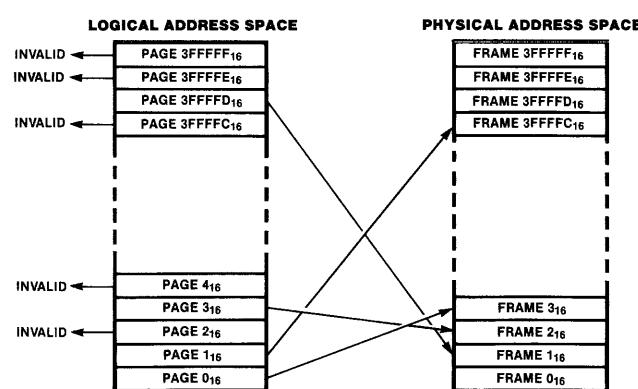


Figure 1-1. Memory Mapping

1.2.4 Addressing Modes

The CPU locates operands (the data manipulated by instructions) in registers, memory, peripheral ports, or in the instruction. The location of an operand is specified by one of nine general addressing modes: Register, Immediate, Indirect Register, Direct Address, Index, Base Address, Base Index, Relative Address, and Relative Index. Instruction formats provide compact encodings for the most frequently used addressing modes.

1.2.5 Instruction Set

The Z80,000 CPU supports operations on nine data types: bit, bit field, signed integer, unsigned integer, logical value, address, packed BCD integer, stack, and string. Integer and logical values can be byte, word, or longword in size. In addition, floating-point operations are implemented through the Extended Processing Architecture (EPA) facility by a coprocessor (Z8070 Arithmetic Processing Unit) or by software emulation.

Several instructions are provided for important control structures. Conditional branches and jumps support "if-then", "while", and "repeat" constructions. The Decrement and Branch if Non-Zero instruction can be used for loop control. Call, Enter, Exit, and Return instructions perform procedure linkage.

The regular combination of addressing modes, operations, and data types offers a powerful instruction set that is well-suited for compilation of high-level languages such as C, Pascal, and Ada.

1.2.6 Normal and System Modes of Operation

The CPU has two modes of operation--normal and system--used to isolate application programs from sensitive portions of the operating system. The mode is selected by a bit in the Flag and Control Word register.

Only programs in system mode are privileged to execute I/O instructions and access control registers. The memory management mechanism allows system mode programs to access regions of memory protected from normal mode access. Further protection is provided with separate stacks for system and normal modes. Application programs use the

System Call instruction and trap to request services from the operating system.

1.2.7 Exceptions

Exceptions are conditions or events that disrupt the usual sequence of instructions. The Z80,000 CPU supports four types of exceptions: reset, bus error, interrupts, and traps. A reset exception initializes the CPU state in response to an external request, typically part of a power-on sequence. A bus error exception occurs when external hardware indicates an irrecoverable error, such as an uncorrectable memory error, on a bus transaction. An interrupt is an asynchronous event signalled externally, typically when a peripheral device needs attention. A trap is a condition detected by the CPU synchronously with execution of an instruction.

When an exception occurs, the CPU saves the Program Status registers of the executing process on the system stack. Then new values for the Program Status registers are read from a table in memory (Program Status Area), thus passing control to an exception handler.

The CPU provides a flexible interrupt structure that includes three types of interrupts: nonmaskable, vectored, and nonvectored. The nonmaskable interrupt, which is of highest priority, is typically reserved for the most critical requirements, such as sudden power failure. Both vectored and nonvectored interrupts can be separately masked by bits in the Flag and Control Word register. Vectored interrupts allow the CPU to branch to a specific exception handler selected by a code read from the peripheral. Nonvectored interrupts use a common exception handler.

The CPU recognizes several trap conditions, all of which can be used to improve software reliability. The System Call trap provides controlled access for application programs to operating system functions. Traps for integer overflow, subrange out of bounds, and subscript out of bounds catch common run-time errors. The Address Translation trap allows the operating system to implement access protection and virtual memory. Traps for breakpoint and single instruction tracing are used during software development. The Conditional Trap instruction is used for software definition of exception conditions not recognized by the CPU hardware.

Z80,000 CPU Overview

1.3 EXTENDED PROCESSING ARCHITECTURE

The Extended Processing Architecture (EPA) facility allows the operations defined in the Z80,000 CPU architecture to be extended by software or hardware. For example, floating-point operations are supported by the Z8070 Arithmetic Processing Unit (APU) or by a software package that emulates the APU.

When the CPU encounters an EPA instruction, it checks a control bit in the Flag and Control Word register to determine whether the EPA facility is enabled. If disabled, the CPU traps for software emulation of the instruction. If enabled, the CPU sends the instruction across the external interface to an Extended Processing Unit (EPU). The CPU then transfers the operands for the instruction to the EPU.

The data processing operations performed by the EPU are transparent to the CPU. In general, the EPU executes complex operations such as floating-point arithmetic, decimal arithmetic, or signal processing with special-purpose hardware.

1.4 CACHE

The Z80,000 CPU contains an on-chip cache buffer to store copies of memory locations that were recently referred to. Most memory references are either to a location that was referred to recently (temporal locality) or to a nearby location (spatial locality). Therefore, on most memory fetches the CPU is able to find the required data in the cache (a hit), thus avoiding a slower access to external memory. When the required data is missing from the cache (a miss), the CPU fetches the data from external memory and loads a copy into the cache. The fetched data replaces the least recently used data in the cache.

The cache provides significant cost/performance advantages by allowing the CPU to execute instruc-

tions at a faster rate than permitted by external memory alone. The cache can be separately enabled to store both instructions and data. The effectiveness of the cache is enhanced by storing data along with instructions, but an application can cache instructions only. Cache replacement on a miss can also be inhibited. This option can be used to lock desired locations into the cache for fast, on-chip access.

1.5 EXTERNAL INTERFACE

The Z80,000 CPU offers a number of features for interfacing to systems that span a wide range of cost/performance requirements. The Hardware Interface Control Register (HICR) specifies certain characteristics of the hardware configuration surrounding the CPU, including bus speed, memory data path width, and number of automatic wait states.

The system designer can fine-tune performance by selecting not only the CPU clock rate and bus speed (1/2 or 1/4 the CPU clock), but also the access time and data path width for the memory. For two independent regions of memory the CPU can be programmed for both the number of wait states automatically inserted, and whether the data path is 16 or 32 bits wide. With these options, a system can easily accommodate a slow, 16-bit-wide bootstrap read-only memory (ROM) in one region and fast, 32-bit-wide random access memory (RAM) in the other. Furthermore, the CPU supports an optional burst transfer of several memory words from consecutive locations. Burst transfers can increase memory bandwidth for interleaved and "nibble-mode" memory systems.

The CPU provides support for four types of multiprocessor configurations: coprocessor, slave processor, tightly-coupled multiple CPUs, and loosely-coupled multiple CPUs. Coprocessors, such as the Z8070 Arithmetic Processing Unit, work synchronously with the CPU to execute a single instruction stream using the Extended Processing

Architecture facility. Slave processors, such as the Z8016 DMA Transfer Controller, perform dedicated functions asynchronously to the CPU. Tightly-coupled multiple CPUs execute independent instruction streams and generally communicate through shared memory on a common bus. Two separate bus request protocols support slave processing and tightly-coupled multiprocessors. Loosely-coupled multiple CPUs generally communicate through a multi-ported peripheral, such as the Z8038 FIFO I/O Interface Unit, using the interrupt and I/O facilities of the Z80,000 CPU.

1.6 CPU INTERNAL ORGANIZATION

Figure 1-2 shows a block diagram of the Z80,000 CPU internal organization, including the following major functional units and data paths:

- The external interface logic controls transactions on the bus. Addresses and data from the internal memory bus are transmitted through the interface to the Z-BUS. The Z-BUS is a time-multiplexed, address/data bus that connects the components of a microprocessor system.
- The cache stores copies of instruction and data memory locations. Instructions are read from the cache on the instruction bus. Data is read from or written to the cache on the memory bus.
- The Translation Lookaside Buffer (TLB) translates logical addresses calculated by the address arithmetic unit to physical addresses used to access the cache.
- The address arithmetic unit performs all address calculations. This unit has a path to the register file for reading base and index

registers and another path to the instruction bus for reading displacements and direct addresses. The result of the address calculation is transmitted to the TLB.

- The register file contains the sixteen general-purpose longword registers, Program Status registers, special-purpose control registers, and several registers used to store values temporarily during instruction execution. The register file has one path to the address arithmetic unit and two paths to the execution arithmetic and logic unit.
- The execution arithmetic and logical unit calculates the results of instruction execution, such as add, exclusive-or, and simple load. This unit has two paths to the register file on which two operands can be read simultaneously or one can be written. One of the paths to the register file is multiplexed with a path from the memory bus.
- The instruction decode and control unit decodes instructions and controls the operation of the other functional units. This unit has a path from the instruction bus and two programmable logic arrays for separate microcoded control of the two arithmetic units. This unit also controls the exception handling and loading of the TLB.

All of the functional units and data paths listed above are 32 bits wide.

The operation of the CPU is highly pipelined so that several instructions are simultaneously in different stages of execution. Thus, the functional units effectively operate in parallel with one instruction being fetched while an address is calculated for another instruction and results are stored for a third instruction.

Z80,000 CPU Overview

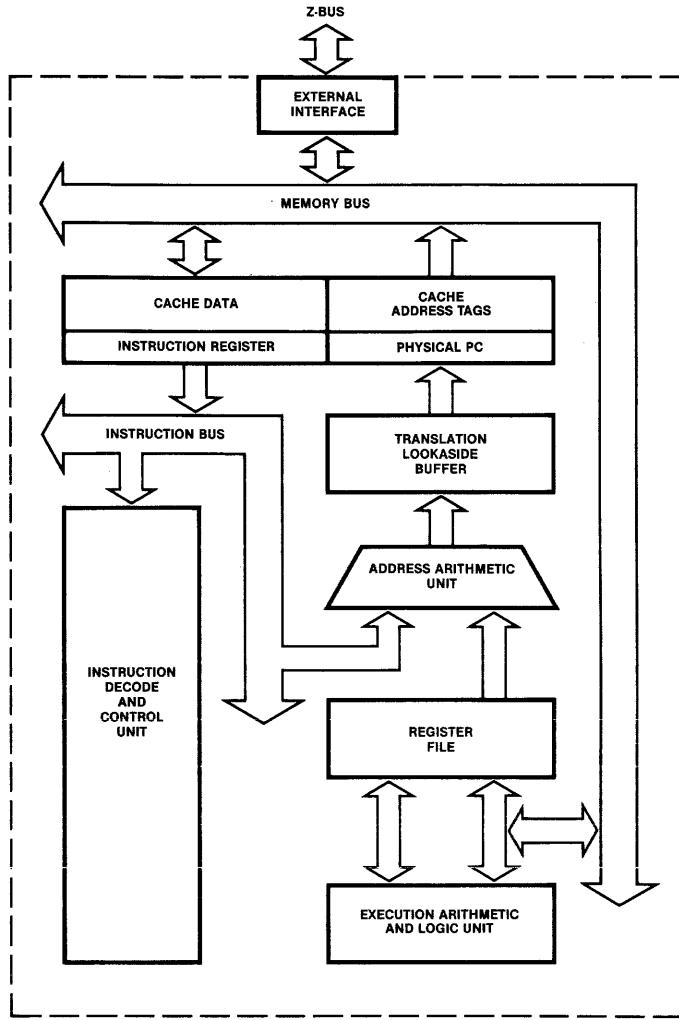


Figure 1-2. Functional Block Diagram

1.7 Z8000 COMPATIBILITY

The Z80,000 CPU's instruction set encoding allows it to directly execute Z8000 family software such as compilers and the ZRTS™ real-time operating system. Z8000 programs must not use the Z8000 privileged instructions, address, and control field encodings if they are to execute correctly on the Z80,000 CPU, since the Z80,000 CPU uses many of these reserved encodings to extend the register file, address range, and instruction functionality.

1.8 SUMMARY

The Z80,000 CPU meets and surpasses the requirements of medium and high-end microprocessor systems. Software program development is easily accomplished with the CPU's sophisticated architecture. The highly-pipelined design, on-chip cache, and external interface support systems ranging from dedicated controllers to mainframe computers.

Chapter 2. Data Formats and Registers

2.1 INTRODUCTION

The Z80,000 CPU manipulates data located in registers, memory, and peripherals. The Z80,000 register repertoire consists of the general-purpose register file, the Program Counter, the Flag and Control Word, and nine special-purpose control registers. This chapter describes the format for data and the use of registers. Chapter 4 describes the use of memory and peripherals.

2.2 DATA FORMATS

The CPU manipulates bits, bytes (8 bits), words (16 bits), longwords (32 bits), and quadwords (64 bits) of data. Within a byte, word, longword, or quadword, the bits are numbered from right to left, from least to most significant (Figure 2-1). This is consistent with the convention that bit n corresponds to position 2^n in the representation of binary numbers. (However, the bit numbering for bit field data, described in Section 6.2.6, is in the opposite direction from Figure 2-1.)

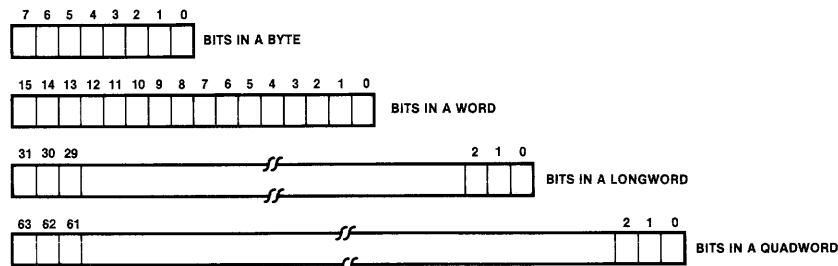


Figure 2-1. Data Formats

2.3 GENERAL-PURPOSE REGISTER FILE

The general-purpose register file contains 64 bytes of storage (Figure 2-2). The first 16 bytes (byte registers R_{L0}, R_{H0}, ..., R_{L7}, R_{H7}) can be used as accumulators for byte data. The first 16 words (word registers R₀, R₁, ..., R₁₅) can be used as accumulators for word data, as index registers (except R₀), or for memory addresses in compact mode (except R₀). Any longword register (R_{R0}, R_{R2}, ..., R_{R30}) can be used as an accumulator for longword data and, in segmented or linear mode, as an index register (except R_{R0}) or for memory addresses (except R_{R0}). Quadword registers (R_{Q0}, R_{Q4}, ..., R_{Q28}) can be used as accumulators

for Multiply, Divide, and Extend Sign instructions. Within quadword register R_{Qn}, R_{Rn} contains the more significant longword. A 4-bit field in instructions specifies which general-purpose register to access. The register size is determined by the instruction opcode.

The unique organization of the register file allows bytes and words of data to be manipulated conveniently while leaving most of the registers free to hold addresses, counters, or other values. For example, four bytes in R_{H0}, R_{L0}, R_{H1}, and R_{L1} can be packed into the single longword register R_{R0} and manipulated independently with the extensive byte-oriented instructions.

Data Formats and Registers

Two registers are dedicated for the Stack Pointer and Frame Pointer used by Call, Enter, Exit, and Return instructions. The Stack Pointer is also used in processing exceptions and by the Interrupt Return instruction. There are separate Stack Pointers for system and normal modes of operation.

The registers used for the Stack Pointer and Frame Pointer depend on the address representation mode. In compact mode, R15 is the Stack Pointer and R14 is the Frame Pointer. In segmented or linear mode, RR14 is the Stack Pointer and RR12 is the Frame Pointer. See Section 3.3 for more details on modes of operation.

RQ0 {	RR0	7 RH0 0	7 RL0 0	7 RH1 0	7 RL1 0	R0, R1
	RR2	7 RH2 0	7 RL2 0	7 RH3 0	7 RL3 0	R2, R3
RQ4 {	RR4	7 RH4 0	7 RL4 0	7 RH5 0	7 RL5 0	R4, R5
	RR6	7 RH6 0	7 RL6 0	7 RH7 0	7 RL7 0	R6, R7
RQ8 {	RR8	15 R8 0	15 R9 0			
	RR10	15 R10 0	15 R11 0			
RQ12 {	RR12	15 R12 0	15 R13 0			
	RR14	15 R14 0	15 R15 0			
RQ16 {	RR16	31	0			
	RR18	31	0			
RQ20 {	RR20	31	0			
	RR22	31	0			
RQ24 {	RR24	31	0			
	RR26	31	0			
RQ28 {	RR28	31	0			
	RR30	31	0			

Figure 2-2. General-Purpose Registers

2.4 PROGRAM STATUS REGISTERS

The Program Status registers are the Program Counter (PC) and the Flag and Control Word (FCW) (Figure 2-3). The PC contains the 32-bit address of the instruction being executed. The 16-bit FCW indicates operating modes, masks for traps and interrupts, and flags set according to the result of instructions.

The low-order byte of the FCW contains six flags, described below, and the integer overflow mask. Many instructions modify or use the flags.

Carry (C) indicates a carry out of the high-order bit position during an operation.

Zero (Z) indicates that the result of an operation is zero.

Sign (S) indicates whether the result of an operation is negative or positive.

Parity/Overflow (P/V) indicates that the result of a logical operation has even parity or that overflow has occurred for arithmetic operations.

Decimal-Adjust (D) is used in BCD arithmetic to indicate whether an addition or subtraction was last executed.

Half Carry (H) is used in BCD arithmetic to convert the result of a previous binary addition or subtraction to a decimal result.

The C, Z, S, and P/V flags can be manipulated using the Complement Flag and Set Flag instructions. Section 6.3 provides more information about the flags.

The Integer Overflow Enable (IV) bit is the mask for an Integer Overflow trap. While this bit is 1, the Integer Overflow trap is enabled; while 0, the integer overflow trap is disabled (see Section 7.4.4.6).

The low-order byte of the FCW can be accessed in normal mode using the Load Control Byte instruction.

The high-order byte of the FCW contains eight control bits:

Extended/Compact Mode (E/C) and **Linear/Segmented Mode (L/S)** controls the mode of address representation. While E/C is 0, addresses are compact (16 bits). While E/C is 1, addresses are extended (32 bits) and are either segmented (L/S is 0) or linear (L/S is 1).

System/Normal Mode (S/N) controls the operating mode. While this bit is 1, the CPU is operating in system mode; while 0, the CPU is operating in normal mode.

Extended Processor Architecture Mode (EPA) controls the Extended Processing Architecture facility. While this bit is 1, the CPU processes extended processing instructions as if the system contains Extended Processing Units, which serve as co-processors to assist the CPU in executing extended processor instructions. While this bit is 0, the CPU traps extended processor instructions.

Vectored Interrupt Enable (VIE) and Nonvectored Interrupt Enable (NVIE) determine when the CPU recognizes vectored and nonvectored interrupts. Vectored interrupts are enabled when VIE is 1; nonvectored interrupts are enabled when NVIE is 1. These bits can be manipulated using the Enable Interrupt and Disable Interrupt instructions.

Trace Pending (TP) and Trace Enable (T) are used for instruction tracing. While T is 1, instruction tracing is enabled; while 0, instruction

tracing is disabled. TP is used with T to ensure that exactly one trace trap occurs after each instruction executed when tracing is enabled (see Section 7.4.4.10).

During exception processing, the Program Status registers are saved on the system stack and new values for the registers are loaded from the Program Status Area. The Program Status registers

can also be loaded using the Interrupt Return and Load Program Status instructions. The FCW can be accessed using the Load Control instruction.

2.5 SPECIAL-PURPOSE CONTROL REGISTERS

The CPU includes nine special-purpose longword registers (Figure 2-4). These are accessed using the Load Control Long instruction.

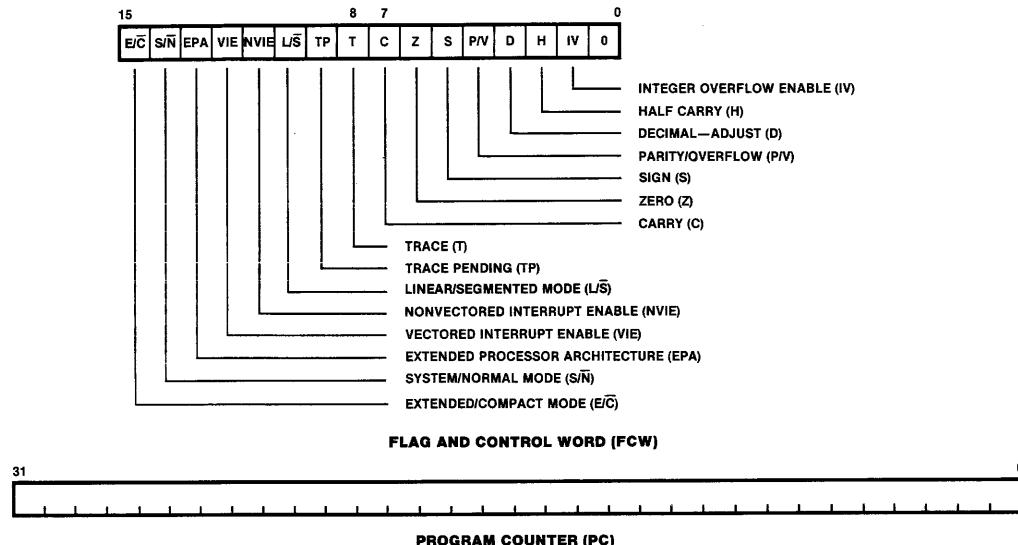


Figure 2-3. Program Status Registers

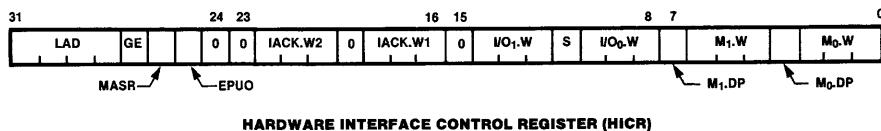


Figure 2-4. Special-Purpose Control Registers

2.5.1 Program Status Area Pointer (PSAP)

The Program Status Area Pointer contains the physical, base address of the Program Status Area. The Program Status Area contains the Program Status information (PC and FCW) fetched during exception processing. Refer to Chapter 7 for more information about the Program Status Area. The longword PSAP can be accessed using the Load Control Long instruction; both the low-order word and high-order word of the PSAP can be accessed using the Load Control instruction.

2.5.2 Normal Stack Pointer (NSP)

The Normal Stack Pointer contains the Stack Pointer used in normal mode. System mode programs can access normal mode register RR14 using the Load Control Long instruction and normal mode registers R14 and R15 using the Load Control instruction.

Data Formats and Registers

2.5.3 Translation Table Descriptor Registers

The translation table descriptor registers--System Instruction Translation Table Descriptor (SITTD), System Data Translation Table Descriptor (SDTTD), Normal Instruction Translation Table Descriptor (NITTD), and Normal Data Translation Table Descriptor (NDTTD)--contain the physical addresses of the translation tables used by the memory management mechanism. These registers also contain other fields that control the memory management mechanism (see Section 4.3.2.1).

2.5.4 Overflow Stack Pointer (OSP)

The Overflow Stack Pointer (OSP) contains the physical address of the Stack Overflow Area. The Stack Overflow Area is used when an address translation error occurs during exception processing (see Section 7.4.5).

2.5.5 Hardware Interface Control Register (HICR)

The Hardware Interface Control register contains fields controlling the external interface of the CPU, including bus speed, data path width, and automatic wait states. (See Section 8.6).

2.5.6 System Configuration Control Longword (SCCL)

The System Configuration Control Longword contains control bits for the address translation mechanism, cache mechanism, and exception processing. These bits are as follows:

System Address Translation (SX) and Normal Address Translation (NX) control the address translation mechanism for system space and normal space references. While either of these bits is 1, the

translation mechanism is enabled for references in the corresponding space; while either bit is 0, the translation mechanism is disabled for references in the corresponding space.

Cache Replacement (CR) controls the cache replacement algorithm. While this bit is 1, the cache replacement algorithm is enabled; while 0, the cache replacement algorithm is disabled. Most applications leave the replacement algorithm enabled. Some applications, however, selectively enable and disable the replacement algorithm to lock specific locations into the cache. Refer to Appendix C for more information.

Cache Instruction (CI) and Cache Data (CD) control the cache mechanism for instruction and data references. While either of these bits is 1, the cache mechanism is enabled for the corresponding references; while either bit is 0, the cache mechanism is disabled for the corresponding references. Refer to Appendix C for more information.

Exception Linear/Segmented mode (XL/S) controls whether linear or segmented mode of address representation is used during exception processing. While this bit is 1, linear mode is used; while 0, segmented mode is used (see Section 7.4.5.)

2.6 RESERVED CONTROL BITS

Some of the bits in the FCW and control register formats shown in Figures 2-3 and 2-4 are marked "0". These bits are reserved for future definition. When the control register is read, these bits return 0. When the control register is written, these bits must be 0. Although the CPU does not check that the reserved bits written to the control register are 0, functions may be defined for these bits in the future.

Chapter 3.

Address Representation and Modes of Operation

3.1 INTRODUCTION

The CPU has three modes of address representation--compact, segmented and linear--and two modes of operation--normal and system.

3.2 ADDRESS REPRESENTATION

As shown in Figure 3-1, the CPU has three modes of address representation: compact, segmented, and linear. The mode is selected by two control bits in the Flag and Control Word register (see Table 3-1). The Extended/Compact (E/\bar{C}) bit selects whether compact addresses (16 bits) or extended addresses (32 bits) are used. For extended addresses, the Linear/Segmented (L/\bar{S}) bit selects whether linear or segmented addresses are used. These modes affect only the representation for logical memory addresses, not logical I/O addresses.

The Load Address instruction can be used to manipulate addresses in any mode of representation. The address calculation performed by this instruction is the same as the addressing used to access an operand.

In compact mode, addresses are 16 bits. Address calculations using compact addresses involve all 16 bits. Compact mode is more efficient and consumes less program space for applications requiring less than 64K bytes of program and less than 64K bytes of data. This efficiency is due to shorter instructions in compact mode, and the fact that addresses in the register file use word rather than longword registers. Applications requiring more than 64K bytes of either program or data should use segmented or linear mode.

Segmented mode supports two segment sizes--64K bytes and 16M bytes. The most-significant bit of the 32-bit address selects either a 15-bit segment number with a 16-bit segment offset (MSB = 0) or a 7-bit segment number with 24-bit segment offset (MSB = 1). Thus, the address space includes 32,768 of the smaller segments and 128 of the larger segments. In segmented mode, address calculations involve only the segment offset; the segment number is unaffected.

Many applications benefit from the logical structure of segmentation by allocating individual objects, such as program modules, stacks, or large data structures, to separate segments.

In linear mode, addresses are 32 bits. Address calculations using linear addresses involve all 32 bits. In linear mode, the address space of 4G bytes is uniform and unstructured. Some applications benefit from the flexibility of linear addressing by allocating objects to arbitrary positions in the address space.

In compact mode, addresses stored in the register file use word registers; in segmented or linear mode, addresses use longword registers. When an address is specified in a register for Indirect, Base Address, and Base Index addressing modes, or for the destination of a Load Address instruction, the address register specified by the instruction is a word register in compact mode and a longword register in segmented or linear mode. Similarly, references to the Program Counter in compact mode use only the low-order word of the PC, while in segmented or linear mode, the entire longword PC is used. In compact mode, the Stack Pointer is R15 and the Frame Pointer is R14. In segmented or linear mode the Stack Pointer is RR14 and the Frame Pointer is RR12.

Some addressing modes generally available in segmented or linear mode are restricted in compact mode. Refer to Chapter 5 for more information about the effect of the address representation mode on addressing modes and address calculation.

In compact mode, addresses encoded in instructions occupy one word; in segmented or linear mode, addresses in instructions occupy one or two

Table 3-1. Address Representation

Control Bits in FCW		
E/\bar{C}	L/\bar{S}	Representation
0	0	Compact
0	1	Reserved
1	0	Segmented
1	1	Linear

Address Representation and Modes of Operation

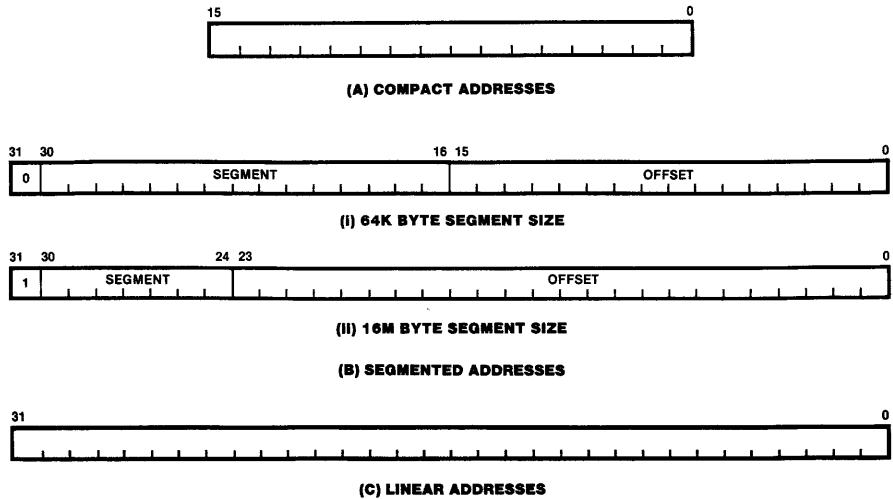


Figure 3-1. Address Representations

words. Refer to Chapter 6 for more information about the effect of the segmentation mode on instruction representation and execution.

3.3 NORMAL AND SYSTEM MODES

The CPU has two modes of operation, normal and system, selected by the S/N bit in the Flag and Control Word register. System mode ($S/N = 1$) is more privileged than normal mode ($S/N = 0$). These modes affect CPU operation in three areas: privileged instructions, Stack Pointers, and memory management.

All instructions can be executed in system mode. Some instructions, such as those performing I/O

operations or accessing control registers, can only be executed in system mode, and are called privileged instructions. When a program operating in normal mode attempts to execute a privileged instruction, an exception occurs. The privileged instructions are identified in the instruction set description in Chapter 6.

The Stack Pointer registers are distinct for normal and system modes. In normal mode, a reference to the Stack Pointer register accesses the Normal Stack Pointer. In system mode, a reference to the Stack Pointer register references the System Stack Pointer. In compact system mode, references to R14 use normal mode R14. Table 3-2 shows the registers accessed in the different modes.

Table 3-2. Registers Referenced by Access to R14 and R15

Register Referenced by Instruction	System Mode		Normal Mode	
	Segmented or Linear	Compact	Segmented or Linear	Compact
R14	System R14	Normal R14	Normal R14	Normal R14
R15	System R15	System R15	Normal R15	Normal R15
RR14	System R14	Normal R14	Normal R14	Normal R14
	System R15	System R15	Normal R15	Normal R15

In normal mode, the System Stack Pointer is not accessible. In system mode, the Normal Stack Pointer is accessed using the Load Control or Load Control Long instruction.

Memory address spaces are distinct for normal and system modes. Different translation tables are used for translating normal and system mode addresses, although the tables can optionally be merged. The access protection performed by the memory management mechanism allows access by system programs to memory locations that are prohibited from access by normal mode programs.

The CPU can change its operating mode whenever the FCW is loaded by a Load Control instruction, Load

Program Status instruction, Interrupt Return instruction, or during exception processing. The distinction between normal and system modes allows the construction of a protected operating system. The operating system kernel runs in system mode to manage the computer system resources--CPU, memory, and peripherals. Application programs run in normal mode, where they are prohibited from interfering with other application programs or the operating system. When application programs require a service that only the operating system can perform, the System Call instruction is executed. System Call causes a trap to the operating system, passing an identifier for the particular service requested.

Chapter 4. **Address Spaces and Memory Management**

4.1 INTRODUCTION

The CPU refers to memory and peripherals to fetch instructions, fetch and store operands, process exceptions, and perform memory management. The CPU uses addresses to specify the location for memory and peripheral references. Logical addresses, which are the addresses manipulated by programs, are distinguished from physical addresses, which are the addresses the CPU presents to memory and peripherals. This chapter describes the types of logical addresses and the procedure for mapping logical to physical addresses. Chapter 8 describes the way the CPU refers to memory and peripherals using physical addresses.

4.2 ADDRESS SPACES

The CPU supports several distinct spaces for logical and physical addresses (Figure 4-1). Logical

addresses are in one of four memory address spaces or in I/O address space. Physical addresses are in memory or I/O address space.

4.2.1 Logical Memory Address Spaces

Logical memory addresses are in system instruction space, system data space, normal instruction space, or normal data space. When the CPU is in system mode, one of the two system address spaces is used for a memory reference. In normal mode, one of the two normal address spaces is used. Instruction address space is used for instruction fetches, immediate mode operand fetches, and fetches or stores of operands specified using Relative Address or Relative Index addressing modes. Data address space is used for references to fetch or store operands in memory, other than those specified using Immediate, Relative, or Relative Index addressing modes. Refer to Chapter 5 for a description of addressing modes.

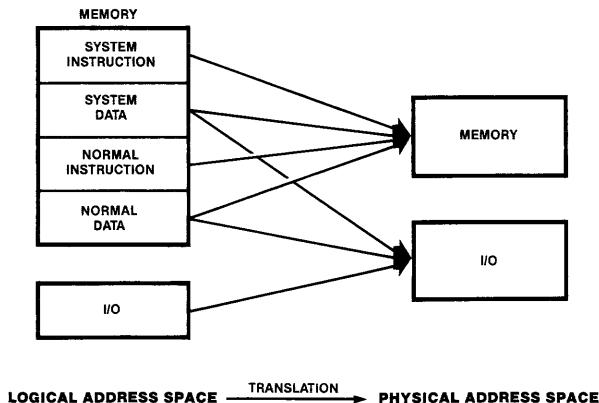


Figure 4-1. Address Spaces

Address Spaces and Memory Management

Logical addresses in the memory spaces are 32 bits. Each address specifies the location of a byte in memory. In compact mode, only the low-order 16 bits of the logical address can be directly manipulated; the high-order 16 bits of the logical address are the high-order 16 bits of the PC (Figure 4-2). In segmented mode, the lower half of each address space contains 32,768 small segments of maximum size 64K bytes, and the upper half contains 128 large segments of maximum size 16M bytes (Figure 4-3). Each segment can be viewed as a contiguous string of bytes at consecutive offsets. In linear mode, the entire address space is a contiguous string of bytes at consecutive addresses.

Words and longwords in memory are addressed using the lowest address of any byte in the word or longword. This is the left-most, highest-order, most-significant byte of the word or longword (Figure 4-4).

Word and longword operands located in memory can be at even or odd addresses. Performance is improved when word operands are located at even addresses and longword operands are located at addresses that are a multiple of four. Instruction words must be located at even addresses. When an attempt is made to execute an instruction at an odd address, an odd PC trap occurs.

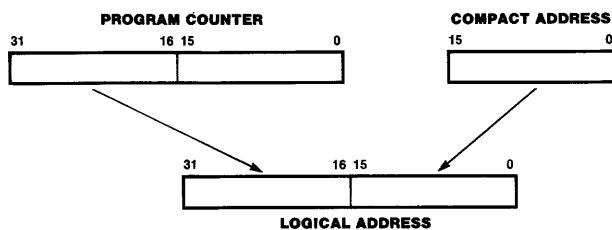


Figure 4-2.
Logical Memory Addresses in Compact Mode

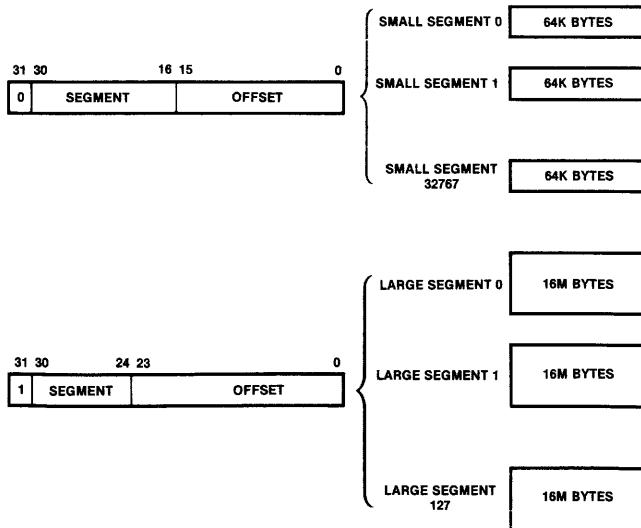


Figure 4-3.
Memory Address Space in Segmented Mode

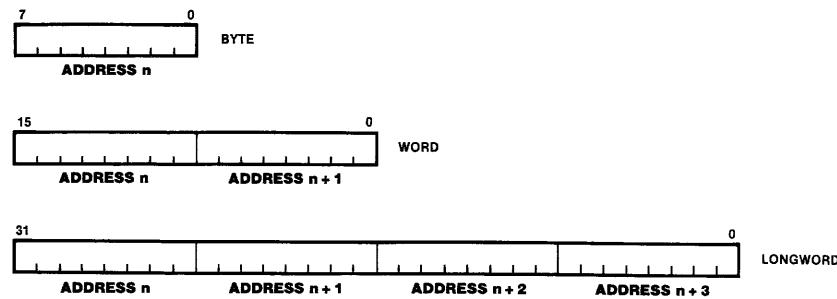


Figure 4-4.
Bytes, Words, and Longwords in Memory

4.2.2 Logical I/O Address Space

Although logical I/O addresses are 32 bits, only the 16 low-order bits of a logical I/O address can be manipulated; the CPU always forces the 16 high-order bits to 0.

Unlike logical memory address spaces, logical I/O address space is not viewed as a string of bytes at consecutive addresses. Rather, the address is simply used to locate a byte, word, or longword peripheral port. The byte port located at address n does not have to be contiguous with the byte port located at address n+1, nor must it be the more significant byte of the word port located at address n. Logical I/O addresses can be either even or odd.

4.2.3 Physical Address Spaces

Physical addresses are in physical memory space or physical I/O space. The two physical address spaces are distinguished by different status and timing on the external interface (see Chapter 8). Also, copies of physical memory locations can be stored in the cache, but copies of physical I/O locations cannot. Physical addresses in both spaces are 32 bits. (Note that the external interface provides information distinguishing between memory references for instructions and data, and between system and normal modes. This information should not be used, however, to separate physical memory addresses into different spaces when the cache mechanism is enabled, because the cache does not distinguish separate physical memory address spaces.)

The CPU maps logical addresses to physical addresses. Addresses in logical I/O space map to identical addresses in physical I/O space. Addresses in logical memory spaces map to addresses in physical memory space or physical I/O

space. The process of translating logical memory addresses is described in the following section.

4.3 MEMORY MANAGEMENT

The CPU features a memory management mechanism that translates logical memory addresses to physical addresses and protects for execute, read, and write accesses. The memory management mechanism serves four functions: relocation, protection, sharing, and virtual memory.

Relocation maps a logical address to a potentially different physical address. This allows multiple processes to use the same logical addresses for distinct physical memory locations. Paged address translation divides the logical address spaces into fixed-size units, called pages, and the physical address spaces into fixed-size units, called frames. A logical page can be mapped to an arbitrary physical frame. Because the pages and frames are of fixed and equal size, memory allocation is simplified.

Protection limits the type of access a process can make to a logical address. A segment or individual page can be protected against instruction fetches, operand fetches, or operand stores in either normal or system mode. The protection features of the CPU provide security for sensitive data or programs, such as proprietary code modules, that should not be copied or modified. The CPU also allows protected access by application programs to selected peripherals (memory-mapped I/O).

Sharing of physical memory by multiple processes is supported by relocation and protection. Logical addresses for several processes can map to the same physical address. The access protection attributes for each process may differ.

Address Spaces and Memory Management

Virtual memory means that the range of logical addresses used by a process can be larger than the allocated physical memory. When a reference is made to a logical address that is not mapped to a physical address, an exception occurs. After the missing page is transferred from secondary storage to main memory, the process can simply be restarted. The CPU provides information about pages that have been referred to or modified, thus helping the operating system allocate memory efficiently.

The memory management mechanism is selectively controlled for references in system or normal spaces by two bits in the System Configuration Control Longword register (SX and NX). When the memory management mechanism is disabled, the physical address used for the reference, which is in physical memory space, is identical to the logical address and all accesses are permitted. The following sections describe address translation and access protection when the memory management mechanism is enabled.

4.3.1 Address Translation

The page size used by the CPU is 1K bytes. The translation process involves mapping a logical page, which is specified by the 22 most-significant bits of the logical address, to a physical frame, which is specified by the 22 most-significant bits of the physical address. The 10 least-significant address bits, which specify the byte within a page or frame, are identical for the logical and physical address. A logical page can generally map to an arbitrary physical frame, except for a restriction that applies only when physical memory modules with different data path widths are used and operands can be located across consecutive logical pages. Refer to section 8.6 for more information.

The CPU contains a Translation Lookaside Buffer

(TLB) that stores the translation information for the 16 most recently used pages in a fully associative memory. For each memory reference, the logical page address is compared with the address tags in the TLB (Figure 4-5). If a matching address tag is found, the corresponding frame address is read from the TLB and used to complete the translation. When information needed to translate the page is missing from the TLB, the CPU automatically refers to tables in memory to perform the translation. The CPU then loads the missing translation information into the TLB, replacing the TLB entry of the least recently referenced page.

Thus, the TLB acts as a buffer for the most recently used page descriptors. This buffer is automatically maintained by on-chip hardware.*

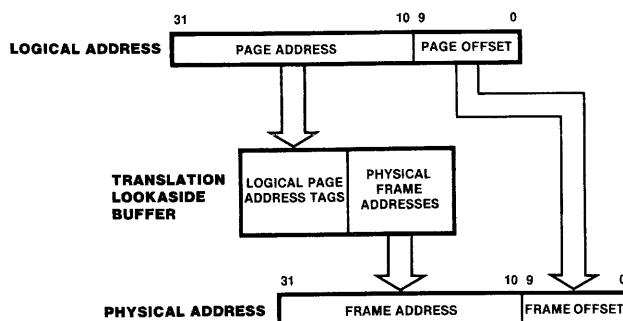


Figure 4-5. Address Translation Using the TLB

The address tags in the TLB are extended from 22 to 24 bits. The extra bits identify the memory address space for the page. Thus, references to pages with the same page number but in different address spaces are translated differently. The frame addresses in the TLBs are also augmented with the access protection code and the Non-Cacheable and Modification bits from the page table entry.

*The number of entries, degree of associativity, and replacement algorithm described for the TLB design in this section are specific to the first implementation of the Z80,000 CPU architecture and may differ in future products implementing the same architecture. Differences in the characteristics can impact systems performance, but have no effect on the function of software or the external interface.

4.3.2 Loading the TLB

To load the TLB with the information needed to translate a page address, the CPU automatically fetches entries from up to three levels of tables in physical memory. Figure 4-6 shows the partition of a logical address into an 8-bit

level-1 field (L1), an 8-bit level-2 field (L2), a 6-bit page number field (P), and a 10-bit page offset field (P-OFFSET). When loading the TLB, the L1, L2, and P fields are used as indexes into the different translation table levels. (Figure 4-7).

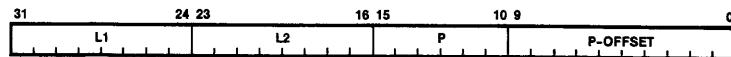


Figure 4-6.
Logical Address Partition for Address Translation

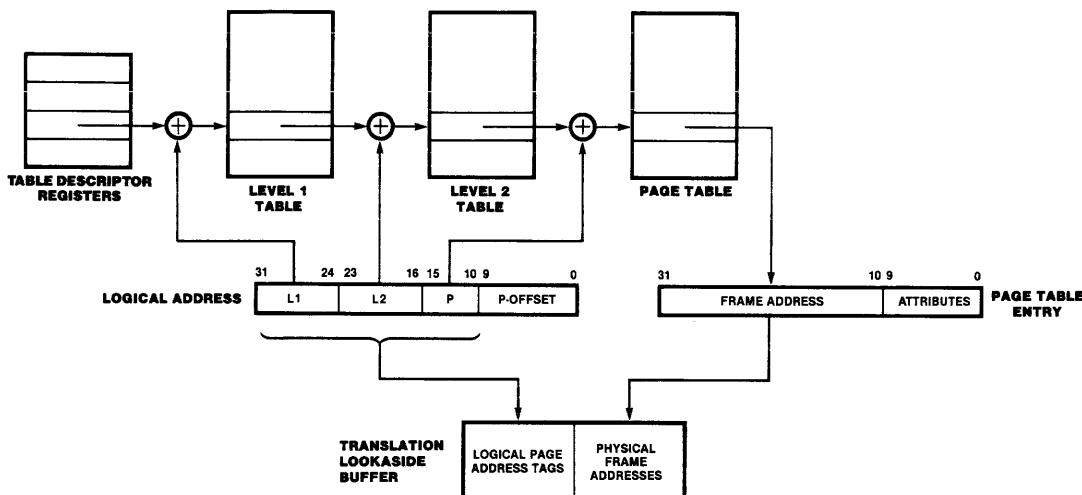


Figure 4-7. Automatic Loading of the TLB
Using Tables in Memory

When the address space is not fully used, the first-level and second-level translation tables can be selectively skipped to reduce the storage for tables and the number of memory references required to autoload the TLB. The level-1 tables can be skipped when an address space of 16M bytes is sufficient. The level-2 tables can be skipped for compatibility with Z8000 segmented addresses. Both level-1 and level-2 tables can be skipped for compact addresses. When a level of tables is skipped, the corresponding field of the logical address is ignored.

When the address spaces are not separated, it is also possible to reduce storage for tables by loading identical values into the translation table descriptor registers. The same tables would then be used to translate addresses in different spaces. The following sections describe the formats of the translation table descriptors and entries and explain the translation algorithm.

Address Spaces and Memory Management

4.3.2.1 Translation Table Descriptor Registers.

There is a translation table descriptor register for each of the four logical memory address spaces: System Instruction Translation Table Descriptor (SITTD), System Data Translation Table Descriptor (SDTTD), Normal Instruction

Translation Table Descriptor (NITTD), and Normal Data Translation Table Descriptor (NDTTD). The translation table descriptor registers are accessed using the Load Control Long instruction. Figure 4-8 shows the format of a translation table descriptor.

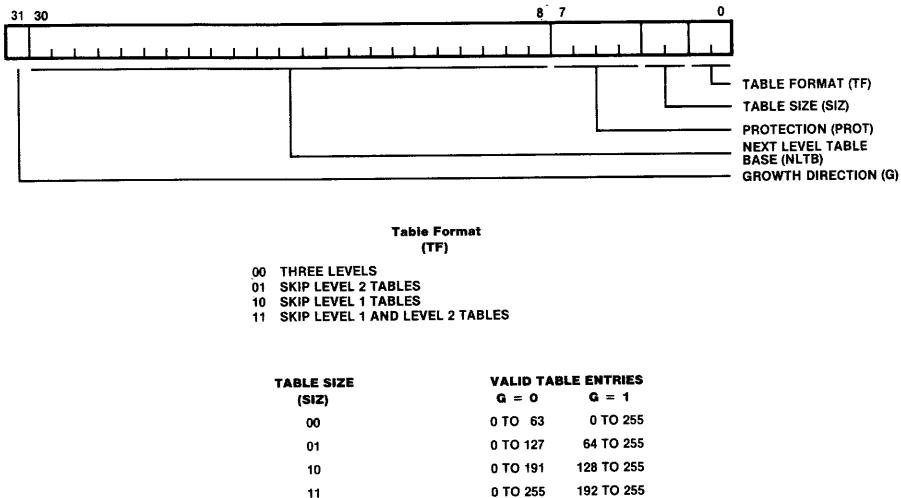


Figure 4-8. Translation Table Descriptor

The Table Format field (TF) specifies the structure of the translation tables. The table format can be a full three levels, two levels with either level-1 tables or level-2 tables skipped, or one level with both level-1 and level-2 tables skipped.

Next Level Table Base (NLTB) specifies 23 bits of the base address in physical memory of the next level table. The full 32-bit address is formed by extending NLTB with one high-order 0 and eight low-order 0s (Figure 4-9).

Growth Direction (G) specifies the growth direction of the next level table from low address to high address ($G=0$) or from high address to low address ($G=1$). The reverse growth direction ($G=1$) is used for downward-growing stacks.

The Table Size field (SIZ), in conjunction with the Growth Direction field, specifies the valid portion of the next level table in increments of 256 bytes. When only part of a table contains valid entries, storage for many invalid entries can be eliminated through use of the SIZ field.

When the next level table is a page table, then the G and SIZ fields must be 0 because a page table always has 64 entries.

Protection (PROT) specifies the access protection code (see Table 4-1).

4.3.2.2 Level-1 Table Entries. The L1 field of the logical address selects one of up to 256 entries in the level-1 table. Figure 4-10 shows the format of a level-1 table entry.

Valid (V) determines the validity of the G, NLTB, and SIZ fields. If the V bit is 1, the fields are valid; otherwise, the fields are invalid. The PROT field is always valid.

Growth direction (G), Next Level Table Base (NLTB), Table Size (SIZ), and Protection (PROT) have the same meaning as in the translation table descriptor registers.

Bit 0 of the level-1 table entry is reserved and must be 0. This bit is ignored by the translation mechanism.

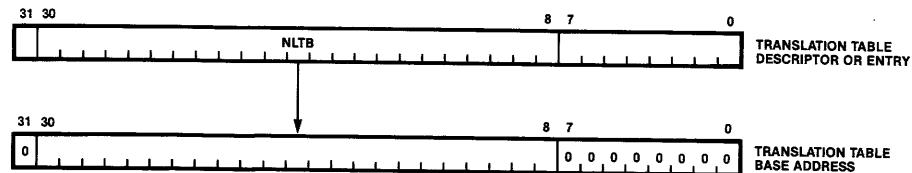


Figure 4-9. Translation Table Base Address

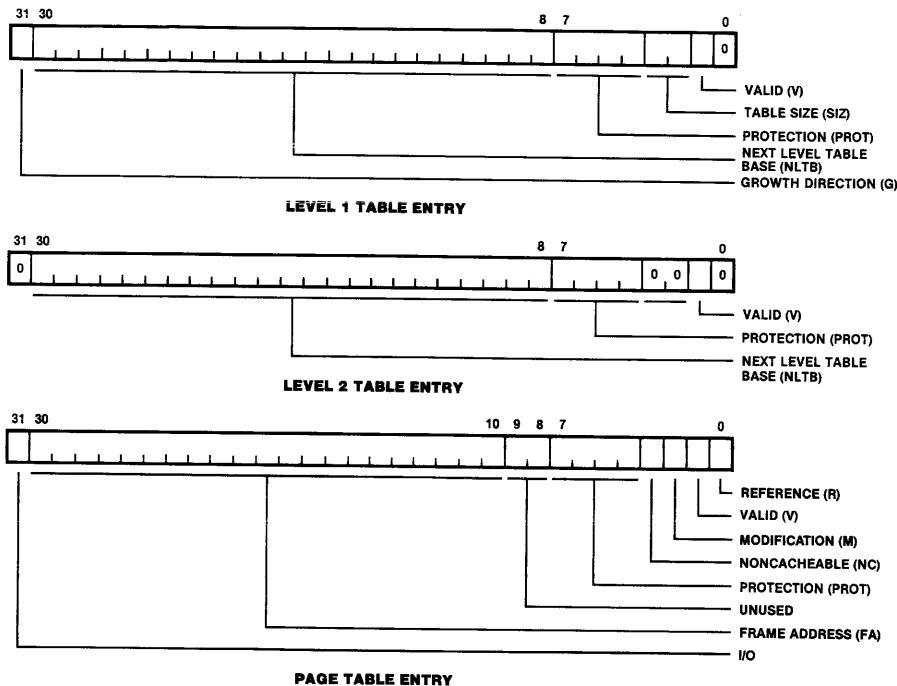


Figure 4-10. Table Entry Formats

4.3.2.3 Level-2 Table Entries. The L2 field of the logical address selects one of up to 256 entries in the level-2 table. Figure 4-10 shows the format of a level-2 segment table entry.

Valid (V) determines the validity of the NLTB field. If the V bit is 1, the field is valid; otherwise the field is invalid. The PROT field is always valid.

Next Level Table Base (NLTB) and Protection (PROT) have the same meaning as in the translation table descriptor registers.

Bits 0, 2, 3 and 31 of the level-2 table entry are reserved and must be 0.

4.3.2.4 Page Table Entries. The P field of the logical address selects one of 64 entries in the page table. Figure 4-10 shows the format of a page table entry.

Valid (V) determines the validity of the I/O, FA, NC, M, and R fields. If the V bit is 1, the fields are valid; otherwise, the fields are invalid. The PROT field is always valid.

I/O determines whether the address of the frame is in physical memory space or physical I/O space. When I/O is 0, the frame is in memory space; when 1, the frame is in I/O space.

Address Spaces and Memory Management

Frame Address (FA) specifies the physical address of the frame corresponding to the logical page. The address is formed by appending ten low-order Os to the I/O bit and the FA field.

Non-Cacheable (NC) is used to maintain the integrity of the cache. If the NC bit is 1, copies of memory locations in this frame cannot be stored in the cache; otherwise, copies of memory locations in this page can be stored in the cache. For example, the NC bit can be set for a page shared by multiple processes with write access in a system containing multiple CPUs. The NC bit has meaning only when the frame is in physical memory; I/O locations are never stored in the cache. See Appendix C for more information.

Modification (M) and Reference (R) bits are used by software to implement virtual memory replacement algorithms. The CPU sets the R bit of the page table entry when the page is first referred to, either for fetching or storing information. The CPU sets the M bit of the page table entry when an operand is first stored to the page. The CPU refers to translation tables in memory to set the M bit on the first store to the page, even if the translation information for the page is present in the TLB because of a previous fetch from the page. The CPU uses interlocked memory references (see Section 8.8.2.3) to set the R and M bits in the page table entry, allowing page tables to be shared between tightly-coupled multiprocessors.

Protection (PROT) specifies the access protection code described below.

Bits 8 and 9 of the page table entry are available for use by software; the bits are ignored by the translation mechanism.

4.3.3 Access Protection

The memory management mechanism enforces access protection for segments and pages using information encoded in the PROT field of translation table descriptors and table entries. The CPU checks three types of access operations: execute, read and write. Execute access is required for instruction fetches, including Immediate mode operand fetches. Read access is required for operand fetches other than Immediate mode. Write access is required for operand stores. The CPU allows different access rights for normal and system mode programs. Table 4-1 shows the interpretation for the PROT code.

Table 4-1.
Protection Field Encoding

Encoding	System	Normal
0000	NA	NA
0001	RE	NA
0010	RE	E
0011	RE	RE
0100	E	NA
0101	E	E
0110	R	NA
0111	R	R
1000	Next	Next
1001	RW	NA
1010	RW	R
1011	RW	RW
1100	RWE	NA
1101	RWE	E
1110	RWE	RE
1111	RWE	RWE

NA - no access is permitted

R - read access is permitted

W - write access is permitted

E - execute access is permitted

Next - Use the protection field of the next level translation table; for page table entries, a PROT field of 1000 indicates no access is permitted.

During the translation process, a PROT field is encountered at each level. The first PROT field with value other than 1000 is selected; the other PROT fields are ignored. If all PROT fields up to and including the page table entry are 1000, no access is permitted.

4.3.4 Address Translation Algorithm

The CPU executes the following algorithm to translate a logical address using the tables in memory when loading a missing entry into the TLB or setting the M bit on the first store to a page.

Step 1. Translation Table Descriptor Processing. One of the four translation table descriptor registers is selected according to the logical address space.

If the PROT field of the segment table descriptor is 1000, the intended access operation is not checked. Otherwise, if the intended access operation is not permitted by the PROT field, an Address Translation trap (access protection violation) occurs.

The G, NLTB, and SIZ fields are passed to the next step of the address translation algorithm.

If the TF field is 00 or 01, then go to Step 2; if the TF field is 10, then go to Step 3; otherwise, go to Step 4.

Step 2. Level-1 Table Entry Processing. The L1 field of the logical address is checked with the G and SIZ fields from Step 1. If G is 0 and L1 is greater than $64 \times (\text{SIZ}+1) - 1$ or if G is 1 and L1 is less than $64 \times \text{SIZ}$, an Address Translation trap (invalid table entry) occurs.

The address of the level-1 table is formed by extending the NLTB field from Step 1 with one high-order 0 and eight low-order 0s. The physical address of the level-1 table entry is calculated by adding $4 \times L1$ to the address of the level-1 table. The addition is a 32-bit unsigned arithmetic operation, ignoring the carry from the most-significant bit position.

The selected level-1 table entry is fetched from memory. If the intended access operation was checked at Step 1 or the PROT field of the table entry is 1000, the intended access operation is not checked at this step. Otherwise, if the intended access operation is not permitted by the PROT field, an Address Translation trap (access protection violation) occurs.

If the V bit of the table entry is 0, an Address Translation trap (invalid table entry) occurs.

The G, NLTB and SIZ fields of the table entry are passed to the next step of the address translation process.

If the TF field of the segment table descriptor is 00, then go to Step 3; otherwise go to Step 4.

Step 3. Level-2 Table Processing. The L2 field of the logical address is checked with the G and SIZ field from the previous step. If G is 0 and L2 is greater than $64 \times (\text{SIZ}+1) - 1$ or if G is 1 and L2 is less than $64 \times \text{SIZ}$, an Address Translation trap (invalid table entry) occurs.

The address of the level-2 table is formed by extending the NLTB field from the previous step with one high-order 0 and eight low-order 0s. The physical address of the level-2 table entry is calculated by adding $4 \times L2$ to the address of the level-2 table. The addition is a 32-bit unsigned arithmetic operation, ignoring the carry from the most-significant bit position.

The selected level-2 table entry is fetched from memory. If the intended access operation was checked at a previous step or the PROT field of the table entry is 1000, the intended access operation is not checked. Otherwise, if the intended access operation is not permitted by the PROT field, an Address Translation trap (access protection violation) occurs.

If the V bit of the table entry is 0, an Address Translation trap (invalid table entry) occurs.

The NLTB field of the table entry is passed to Step 4.

Step 4. Page Table Entry Processing. The address of the page table is formed by extending the NLTB field from the previous step with one high-order 0 and eight low-order 0s. The physical address of the page table entry is calculated by adding $4 \times P$ to the address of the page table. The addition is a 32-bit unsigned arithmetic operation, ignoring the carry from the most-significant bit position.

The selected page table entry is fetched from memory. If the intended access operation was not checked at a previous step, and the intended access operation is not permitted by the PROT field, an Address Translation trap (access protection violation) occurs.

If the V bit of the table entry is 0, an Address Translation trap (invalid table entry) occurs.

If the R bit of the table entry is 0, the CPU sets R to 1. If the M bit is 0 and the access operation is write, the CPU sets M to 1. If either the R or M bit changes, the CPU writes the low-order byte of the table entry back to memory; otherwise, the table entry is unchanged.

Finally, the I/O, FA, NC, M, and selected PROT fields are loaded into the TLB, along with the associated logical page address.

4.3.5 Address Translation Exceptions

The CPU detects two types of address translation exception conditions: access protection violation and invalid table entry. When either of the exception conditions is detected, the CPU suspends the instruction being executed and processes an Address Translation trap. During trap processing the CPU saves on the system stack the PC, the FCW, an identifier word, and the logical address that caused the trap. The saved PC value is the address of the first word of the instruction that caused the trap. The identifier word (Figure 4-11) indicates the type of exception and the

address space that caused the trap. When both types of address translation exception are detected, an access protection violation is indicated.

When an Address Translation trap occurs, the CPU saves the state of registers and memory so the instruction can simply be restarted. The instruction can be successfully completed by eliminating the exception condition, popping the violation address from the system stack, and executing the Interrupt Return instruction. Refer to Chapter 7 for more information about exception processing.

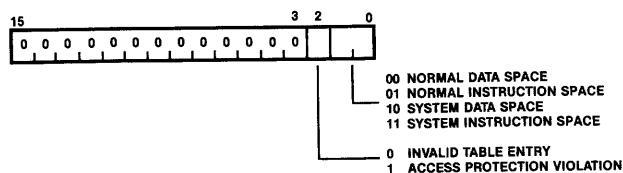


Figure 4-11.
Address Translation Trap Identifier Word

4.3.6 Memory Management Instructions

The CPU provides several privileged instructions directly concerned with memory management. The Load Normal instructions permit system mode programs to refer to normal address spaces. These instructions check access rights using system mode privilege.

The Load Physical address instructions translate a logical address in any of the memory address spaces and load the corresponding physical address into a register. The CPU sets the flag bits in the FCW to indicate the access rights and whether the translation is valid. Although the CPU does not refer to the location of the translated address, the R bit in the page table entry is set by this instruction.

Three types of instructions allow outdated information to be eliminated from the TLB when the memory map is changed by altering one of the translation table descriptor registers or translation table entries. When a page table entry is altered (other than setting the R, M, or V bits), then one of the Purge TLB Entry instructions can be used to remove the translation information for the page from the TLB. The Purge TLB Normal instruction removes all normal space entries from the TLB. This instruction is used when the normal space memory map is changed, but the system space memory map remains the same. For example, the operating system executes the Purge TLB Normal instruction when a process switch occurs as long as system and normal address spaces are separate. The Purge TLB instruction removes all entries from the TLB.

Chapter 5. Addressing Modes and Address Calculations

5.1 INTRODUCTION

The CPU locates operands (the data manipulated by instructions) in registers, memory, peripheral ports, or in the instruction. Figure 5-1 shows the nine addressing modes used to specify the location of operands. Although most operations can use any of the addressing modes, certain operations, such as Load Control, allow only a restricted set of addressing modes.

This chapter describes the addressing modes and the way operand addresses are calculated. Examples are given for compact, segmented, and linear modes of address representation. Chapter 6 provides details about the encoding of addressing modes and the addressing modes allowed for each operation.

5.2 ADDRESS CALCULATIONS

When an operand is in a logical memory address space, the "effective address" of the operand is calculated using a base address, an optional index value, and an optional displacement. The base address is located in a general-purpose register, the Program Counter (PC), or in the instruction. The index value is located in a word or longword register. The displacement is located in the instruction. The following sections describe the calculations of effective addresses in compact, segmented and linear modes.

When an operand is in logical I/O space, no address calculation is necessary. The 16-bit address of the I/O port is located in a word register or in the instruction.

5.2.1 Compact Address Calculations

In compact mode, addresses are 16 bits. The base address for the effective address calculation is located in either a word register other than R0, the low-order word of the PC, or a word of the instruction. When an index value is used, it is located in a word register other than R0. The displacement is encoded in 16 or fewer bits of the instruction. When the displacement is encoded in fewer than 16 bits, it is extended to 16 bits for

effective address calculation. Displacements are generally extended by replicating the sign (most-significant) bit in the high-order bit positions, but for the Decrement and Jump if Not Zero (DJNZ) instruction, the displacement is extended with 0s. In compact mode, it is not possible to specify both an index value and a displacement for effective address calculation.

The effective address is generally calculated by adding the base address to the optional index value or displacement, but for the Call Relative (CALR) and DJNZ instructions, the displacement is subtracted from the base address. Addresses are calculated using 16-bit arithmetic. Carry and overflow from the most-significant bit position are ignored. Thus, addresses wraparound with address 0 appearing to follow address 65,535.

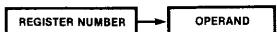
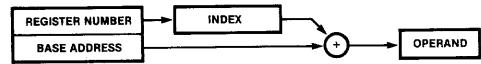
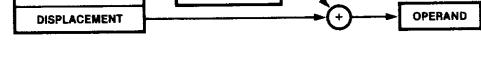
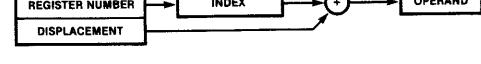
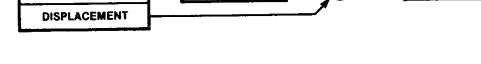
The following example shows an effective address calculation with base address 1234₁₆ and index value or displacement FEDC₁₆. The effective address is 1110₁₆

base address	1234
+ index value or displacement	FEDC
= effective address	1110

5.2.2 Segmented Address Calculations

In segmented mode, addresses are 32 bits. The base address for the effective address calculation is located in either a longword register other than RRO, in the PC, or in one or two words of the instruction. (A concise representation of the 32-bit base address using a single instruction word is available for some addresses. Refer to Section 6.4.3.2 for more information.) When an index value is used, it is located in a word register other than R0 or a longword register other than RRO. An index value located in a word register is extended to 32 bits for effective address calculation by replicating the sign (most-significant) bit in the high-order bit positions. The displacement in an instruction is encoded in 32 or fewer bits. When the displacement is encoded in fewer than 32 bits, it is extended to 32 bits for effective address calculation. Displacements are

Addressing Modes and Address Calculations

Addressing Mode	Operand Addressing		Operand Value
	In the Instruction	In a Register	
R	Register		The contents of the register
IM	Immediate		In the instruction
*IR	Indirect Register		The contents of the location whose address is in the register
DA	Direct Address		The contents of the location whose address is in the instruction
*X	Index		The contents of the location whose address is the address in the instruction, plus the contents of the Index Register
*BA	Base Address		The contents of the location whose address is the contents of the Base register, plus the displacement in the instruction
*BX	Base Index		The contents of the location whose address is the contents of the Base register, plus the contents of the Index register, plus the displacement in the instruction
RA	Relative Address		The contents of the location whose address is the contents of the Program Counter, plus the displacement in the instruction
*RX	Relative Index		The contents of the location whose address is the contents of the Program Counter, plus the contents of the Index register, plus the displacement in the instruction

*R0 and RR0 cannot be used for Indirect, Base, or Index registers

Figure 5-1. Addressing Modes

generally extended by replicating the sign (most-significant) bit in the high-order bit positions, but for the Decrement and Jump if Not Zero (DJNZ) instruction, the displacement is extended with 0s.

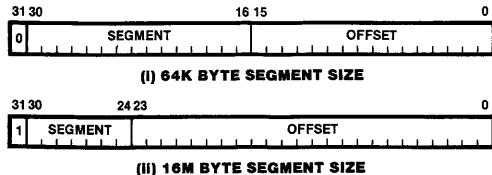


Figure 5-2. Segmented Addresses

In segmented mode, the base address is composed of a segment number and segment offset. Bit 31 of an address distinguishes between two segment sizes (Figure 5-2). When bit 31 of the address is 0, the segment number is 15 bits and the segment offset is 16 bits, providing a maximum segment size of 64K bytes. Addresses for these small segments are written using the notation $\langle\langle ss \# \rangle\rangle$ segment_offset. For example, small segment number five at offset 231A₁₆ would be written $\langle\langle ss \# 5 \rangle\rangle$ 231A₁₆. When bit 31 of the address is 1, the segment number is 7 bits and the segment offset is 24 bits, providing a maximum segment size of 16M bytes. Addresses for these large segments are written using the notation $\langle\langle ls \# \rangle\rangle$ segment_offset.

The effective address is generally calculated by adding the base address to the optional index value and optional displacement, but for CALR and DJNZ instructions, the displacement is subtracted from the base address. Only the segment offset is involved in address arithmetic. The segment size and segment number of the effective address are the same as the base address. The offset calculation uses 16-bit arithmetic for the small segments and 24-bit arithmetic for the large segments. Carry and overflow from the most-significant bit position are ignored. Thus, addresses wraparound within a segment. This means that, for the small segments, offset 0 appears to follow offset 65,535. For the large segments, offset 0 appears to follow offset 16,777,215.

The following example shows an effective address calculation for a small segment with base address $\langle\langle ss \# 2 \rangle\rangle$ 5678₁₆, index value 0000BA98₁₆, and displacement FFFFFFFF₁₆. The effective address is $\langle\langle ss \# 2 \rangle\rangle$ 110F₁₆.

	Segment Number	Segment Offset
base address	$\langle\langle ss \# 2 \rangle\rangle$	5678
+ index value		0000 BA98
+ displacement		FFFF FFFF
= effective address	$\langle\langle ss \# 2 \rangle\rangle$	110F

Another example shows an effective address calculation for a large segment with base address $\langle\langle ls \# 3 \rangle\rangle$ 13579B₁₆, index value FFFFFFFE0, and displacement 00000002. The effective address is $\langle\langle ls \# 3 \rangle\rangle$ 13577D₁₆.

	segment number	segment offset
base address	$\langle\langle ls \# 3 \rangle\rangle$	13579B
+ index value		FF FFFF E0
+ displacement		00 000002
= effective address	$\langle\langle ls \# 3 \rangle\rangle$	13577D

5.2.3 Linear Address Calculations

In linear mode, addresses are 32 bits. The base address for the effective address calculation is located in either a longword register other than R0, in the PC, or in one or two words of the instruction. (A concise representation of the 32-bit base address using a single instruction word is available for some addresses. Refer to Section 6.4.3.2 for more information.) When an index value is used, it is located in a word register other than R0 or a longword register other than R0. An index value located in a word register is extended to 32 bits for effective address calculation by replicating the sign (most-significant) bit in the high-order bit positions. The displacement in an instruction is encoded in 32 or fewer bits. When the displacement is encoded in fewer than 32 bits, it is extended to 32 bits for effective address calculation. Displacements are generally extended by replicating the sign (most-significant) bit in the high-order bit positions, but for the Decrement and Jump if Not Zero (DJNZ) instruction, the displacement is extended with 0s.

The effective address is generally calculated by adding the base address to the optional index value and optional displacement, but for CALR and DJNZ instructions the displacement is subtracted from the base address. Addresses are calculated using 32-bit arithmetic. Carry and overflow from the most-significant bit position are ignored. Thus, addresses wraparound with address 0 appearing to follow address $2^{32}-1$.

The following example shows an effective address calculation with base address 01000000₁₆, index value 00000064₁₆, and displacement FFFFFFF9B₁₆. The effective address is 00FFFFFF₁₆.

	base address	0100 0000
+ index value		0000 0064
+ displacement		FFFF FF9B
= effective address		00FF FFFF

Addressing Modes and Address Calculations

5.3 ADDRESSING MODE DESCRIPTIONS

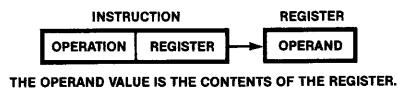
The following sections describe the nine addressing modes. Each description explains how the operand is located, shows the assembler language syntax used, and works through an example. The descriptions are grouped into two sections--one for compact mode and the other for segmented and linear modes. In the examples, hexadecimal notation is used for memory addresses and the contents of register and memory locations. The % symbol

precedes hexadecimal numbers in assembler language text. When the examples refer to memory locations, logical addresses are used; the logical addresses are translated to physical addresses if memory management is enabled.

5.3.1 Compact Mode Descriptions and Examples

This section describes the addressing modes used in the compact mode of operation.

5.3.1.1 Register (R). For Register addressing mode, the operand is located in the specified general-purpose register. Storing data in a register allows shorter instructions and faster execution than storing data in memory. The register size (byte, word, longword, or quadword) is specified by the instruction opcode.



THE OPERAND VALUE IS THE CONTENTS OF THE REGISTER.

Assembler language syntax:

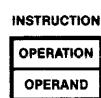
RHn, RLn	Byte register
Rn	Word register
RRn	Longword register
RQn	Quadword register

Example of R mode:

LDL RR20,RR22 //load the contents //of RR22 into RR20

Before Execution	After Execution
RR20 [01234567]	RR20 [A6B89A20]
RR22 [A6B89A20]	RR22 [A6B89A20]

5.3.1.2 Immediate (IM). For Immediate addressing mode, the operand is located in the instruction. Because an immediate operand is part of an instruction, it is located in one of the instruction memory address spaces. Small immediate values are used frequently, so the instruction set provides several concise encodings for these cases.



THE OPERAND VALUE IS IN THE INSTRUCTION.

Assembler language syntax:

#data

Example of IM mode:

LDB RH2,#%55 //load 55₁₆ into RH2

Before Execution	After Execution
RR2 [67891234]	RR2 [55891234]

5.3.1.3 Indirect Register (IR). For Indirect Register addressing mode, the operand is located at the address contained in the specified general-purpose word register. Any word register other than R0 can be used. Depending on the instruction opcode, the operand is located in one of the data memory address spaces or in I/O address space. Indirect Register mode has a short encoding and can be used to simulate more complex addressing modes by computing the address into a register.



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

Assembler language syntax:

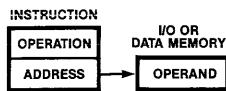
@Rn

Example of IR mode:

LD R2,@R5 //load R2 with the //data addressed //by the contents //of R5

Before Execution	Data Memory
RR2 [030F0005]	⋮
RR4 [2000170A]	1708 [A0230B0E]
	170C [10DC2345]
	⋮
RR2 [0B0E0005]	
RR4 [2000170A]	

5.3.1.4 Direct Address (DA). For Direct Address addressing mode, the operand is located at the address specified in the instruction. Depending on the instruction opcode, the operand is located in one of the data memory address spaces or in I/O address space.



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

Example of DA mode:

LDL RR30,%5E23 //load RR30 with the //longword whose //address is 5E23₁₆

Before Execution

RR30 6789A438

Data Memory

5E20	01	06	C1	02
5E24	03	04	05	00

After Execution

RR30 02030405

Assembler language syntax:

address Either memory or I/O

5.3.1.5 Index (X). For Index addressing mode the operand is located at the address calculated by adding the address specified in the instruction to the index value contained in the specified general-purpose word register. Any word register other than R0 can be used. The operand is located in one of the data memory address spaces. Index addressing mode can be used for random access to tables or other complex data structures where the address of the base of the table is known, but the particular element index must be computed by the program.

Assembler language syntax:

address(Rn)

Example of X mode:

LDL RR8,%231A(R7) //load RR8 with the //longword whose //address is 231A + //the value in R7

Before Execution

RR6 00 00 01 FE

Data Memory

2514	F3	C2	57	1E
2518	3D	0E	7A	DA

Address Calculation

$$\begin{array}{r} 231A \\ + 01FE \\ \hline 2518 \end{array}$$

After Execution

RR6 00 00 01 FE

RR8 3D0E 7ADA



THE OPERAND VALUE IS THE CONTENTS OF THE LOCATION WHOSE ADDRESS IS THE ADDRESS IN THE INSTRUCTION PLUS THE CONTENTS OF THE REGISTER.

Addressing Modes and Address Calculations

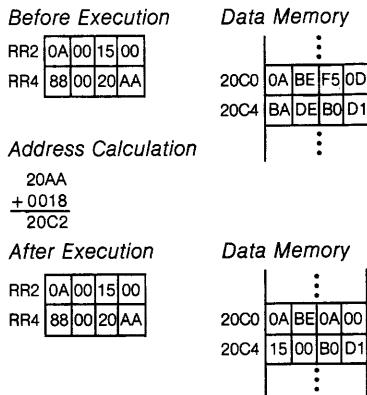
5.3.1.6 Base Address (BA). For Base Address addressing mode, the operand is located at the address calculated by adding the displacement contained in the instruction to the address contained in the specified general-purpose word register. Any word register other than R0 can be used. The operand is located in one of the data memory address spaces. In compact mode, Base Address addressing mode can only be used with Load and Load Address instructions. This restriction is not significant, however, because Index and Base Address addressing modes perform equivalent functions in compact mode.

Assembler language syntax:

Rn (disp)

Example of BA mode:

LDL R5(%18),RR2 //load RR2 into the
//longword whose
//address is the base
//address in
//R5 + 1816



THE OPERAND VALUE IS THE CONTENTS OF THE LOCATION WHOSE ADDRESS IS THE CONTENTS OF THE REGISTER PLUS THE DISPLACEMENT IN THE INSTRUCTION.

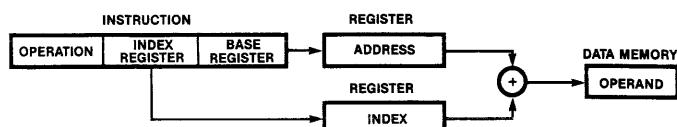
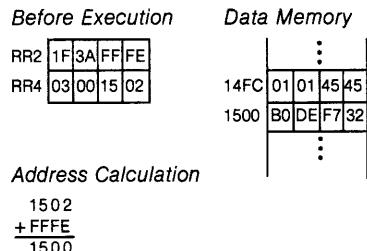
5.3.1.7 Base Index (BX). For Base Index addressing mode, the operand is located at the address calculated by adding the index value contained in the specified general-purpose word index register to the base address contained in the specified general-purpose word base register. Any word register other than R0 can be used for the index register or base register. The operand is located in one of the data memory address spaces. Base Index addressing mode can be used to access tables or other complex data structures when the base of the table and particular element index are not known until the program is executed. In compact mode, Base Index addressing mode can only be used with Load and Load Address instructions.

Assembler language syntax:

Rn (Rm)

Example of BX mode:

LDL RR2,R5(R3) //load RR2 with the
//longword whose
//address is the base
//address in R5 + the
//value in R3



THE OPERAND VALUE IS THE CONTENTS OF THE LOCATION WHOSE ADDRESS IS THE CONTENTS OF THE BASE REGISTER PLUS THE CONTENTS OF THE INDEX REGISTER.

5.3.1.8 Relative Address (RA). For Relative Address addressing mode, the operand is located at the address calculated by adding the displacement contained in the instruction to the low-order word of the Program Counter. The value used for the PC is the address of the instruction word following the displacement. The operand is located in one of the instruction memory address spaces. In compact mode, Relative Address addressing mode can only be used with Load, Load Address, Call, Jump, and DJNZ instructions.

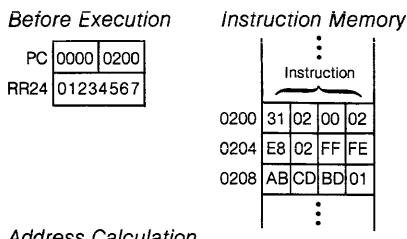
Assembler language syntax:

address

Example of RA mode: (Note that the symbol "\$" is used for the address of the first word of the current instruction.)

```
LDRL RR24,$ + %6 //load RR24 with the
//longword whose
//address is the
//address of the
//first word of
//the current
//instruction + 6
```

Because the Program Counter will be advanced to point to the next instruction when the address calculation is performed, the displacement in the instruction is actually +2 (four less than the offset given by the assembler language syntax).

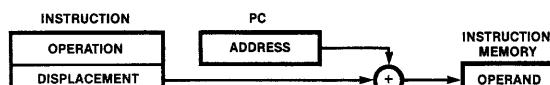


Address Calculation

$$\begin{array}{r} 0204 \\ +0002 \\ \hline 0206 \end{array}$$

After Execution

PC	0000 0204
RR24	FFFEABCD



THE OPERAND VALUE IS THE CONTENTS OF THE LOCATION WHOSE ADDRESS IS THE CONTENTS OF PC PLUS THE DISPLACEMENT IN THE INSTRUCTION.

5.3.1.9 Relative Index (RX). Relative Index addressing mode cannot be used in compact mode of operation.

5.3.2 Segmented and Linear Mode Descriptions and Examples

This section describes the addressing modes used in segmented and linear modes of operation. The description is identical for the two modes of address representation except that separate examples are given for address calculations.

5.3.2.1 Register (R). For Register addressing mode, the operand is located in the specified general-purpose register. Storing data in a register allows shorter instructions and faster execution than storing data in memory. The register size (byte, word, longword, or quadword) is specified by the instruction opcode.

INSTRUCTION REGISTER



THE OPERAND VALUE IS THE CONTENTS OF THE REGISTER.

Assembler language syntax:

RHn, RLn	Byte register
Rn	Word register
RRn	Longword register
RQn	Quadword register

Example of R mode:

```
LDL RR20, RR22 //load the contents of
//RR22 into RR20
```

Before Execution

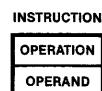
RR20	01234567
RR22	A6B89A20

After Execution

RR20	A6B89A20
RR22	A6B89A20

Addressing Modes and Address Calculations

5.3.2.2 Immediate (IM). For Immediate addressing mode, the operand is located in the instruction. Because an immediate operand is part of an instruction, it is located in one of the instruction memory address spaces. Small immediate values are used frequently, so the instruction set provides several concise encodings for these cases.



THE OPERAND VALUE IS IN THE INSTRUCTION

5.3.2.3 Indirect Register (IR). For Indirect Register addressing mode, the operand is located at the address contained in the specified general-purpose register. Depending on the instruction opcode, the operand is located in one of the data memory address spaces or in I/O address space.

Assembler language syntax:

#data

Example of IM mode:

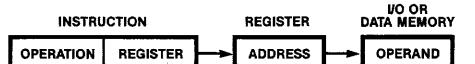
LDB RH2 #55 //load 55₁₆ into RH2

Before Execution

After Execution

BB2 55891234

For memory addresses, any longword register other than R0 can be specified; for I/O addresses any word register other than R0 can be specified. Indirect Register mode has a short encoding and can be used to simulate more complex addressing modes by computing the address into a register.



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

Example of segmented IR mode:

LD R2,@RR4 //load R2 with the
//word whose address
//is in RR4

Before Execution

RR2	03	0F	00	05
RR4	00	02	17	0C

Data Memory

◀ ss#2 ▶ 170E 10 D3 23 45

After Execution

RR2	0B	0E	00	05
RR4	00	02	17	0C

Assembler language syntax:

@Rn I/O address
@RRn Memory address

Example of linear IR mode:

LD R2,@RR4 //load R2 with the word whose address is in RR4

Before Execution

BB2 030E000E

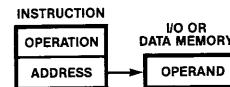
RR4 00 02 17 0C 0002 170A A0 23 0B 0E
0002 170E 10 D3 23 45

After Execution

RR2	0B	0E	00	05
RR4	00	02	17	0C

5.3.2.4 Direct Address (DA). For Direct Address addressing mode, the operand is located at the address specified in the instruction. Depending

on the instruction opcode, the operand is located in one of the data memory address spaces or in I/O address space.



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

Example of segmented DA mode:

LDL RR30,`<<ls#5>>%23`

//load RR30 with the
//longword in large
//segment #5
//at offset 23₁₆

Before Execution

RR30 [6789A438]

Data Memory

<<ls#5>> 000020	02	06	C1	02
<<ls#5>> 000024	03	04	05	00

After Execution

RR30 [02030405]

Assembler language syntax:

address Either memory or I/O

Example of linear DA mode:

LDL RR30, %85000023

//load RR30 with the
//longword whose
//address is
//85000023₁₆

Before Execution

RR30 [6789A438]

Data Memory

8500 0020	02	06	C1	02
8500 0024	03	04	05	00

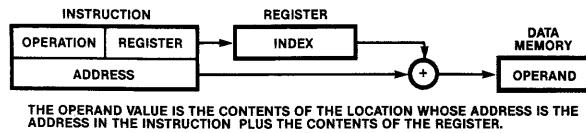
After Execution

RR30 [02030405]

Addressing Modes and Address Calculations

5.3.2.5 Index (X). For Index addressing mode, the operand is located at the address calculated by adding the address specified in the instruction to the index value contained in the specified general-purpose register. Any word register other than R0 or any longword register other than RRO

can be used. The operand is located in one of the data memory address spaces. Index addressing mode can be used for random access to tables or other complex data structures where the address of the base of the table is known, but the particular element index must be computed by the program.



Assembler language syntax:

address(Rn) Word index register
address(RRn) Longword index register

Example of segmented X mode:

```
LDL RR8, <ss#5>%231A(R7)
    //load RR8 with the
    //longword whose
    //address is small
    //segment 5 at
    //offset 231A +
    //the value in R7
```

Example of linear X mode:

LDL RR8, %0005231A(R7)
 //load RR8 with the
 //longword whose
 //address is
 //0005231A₁₆ +
 //the value in R7

Before Execution				Data Memory			
RR6 00 00 01 FE							
RR8 203A 4579							
				◀ ss#5 ▶ 2514	F3	C2	57 1E
				◀ ss#5 ▶ 2518	3D	0E	7A DA

Address Calculation

**←SS#5→ 231A
+ 01FE
←SS#5→ 2518**

After Execution

RR6	00	00	01	FE
RR8	3D0E	7ADA		

Before Execution		Data Memory	
RR6	000001FE		
RR8	203A4579		
		0005 2514	F3 C2 57 1E
		0005 2518	3D 0E 7A DA

Address Calculation

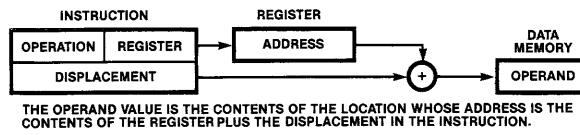
0005 231A
+ 0000 01FE
<hr/>
0005 2518

After Execution

RR6	00	00	01	FE
RR8	3D0E	7ADA		

5.3.2.6 Base Address (BA). For Base Address addressing mode, the operand is located at the address calculated by adding the displacement contained in the instruction to the address contained in the specified general-purpose longword register. Any longword register other than RR0 can be used. The operand is located in

one of the data memory address spaces. Base Address addressing mode can be used to access records or other data structures where the displacement of an element within the structure is known before the program is executed, but the base address of the particular structure is not known until the program is executed.



Assembler language syntax:

RRn(displ)

Example of segmented BA mode:

LDL RR4(%18),RR2 //load RR2 into the
//longword whose
//address is the
//base address in
//RR4 + 18₁₆

Before Execution

RR2	0A	00	15	00
RR4	88	00	20	AA

Data Memory

...
«ls#8» 20C0
0A BE F5 0D
«ls#8» 20C4
BA DE B0 D1
...

Address Calculation

$$\begin{array}{r} \ll Is\#8\gg 0020AA \\ + 000018 \\ \hline \ll Is\#8\gg 0020C2 \end{array}$$

After Execution

RR2	0A	00	15	00
RR4	88	00	20	AA

Data Memory

...
«ls#8» 20C0
0A BE 0A 00
«ls#8» 20C4
15 00 B0 D1
...

Example of linear BA mode:

LDL RR4(%18),RR2 //load RR2 into the
//longword whose
//address is the
//base address in
//RR4 + 18₁₆

Before Execution

RR2	0A	00	15	00
RR4	88	00	20	AA

Data Memory

...
8800 20C0
0A BE F5 0D
8800 20C4
BA DE B0 D1
...

Address Calculation

$$\begin{array}{r} 8800 \text{ 20AA} \\ + 0000 \text{ 0018} \\ \hline 8800 \text{ 20C2} \end{array}$$

After Execution

RR2	0A	00	15	00
RR4	88	00	20	AA

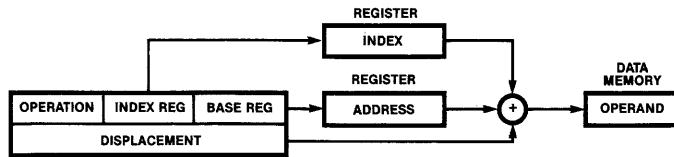
Data Memory

...
8800 20C0
0A BE 0A 00
8800 20C4
15 00 B0 D1
...

Addressing Modes and Address Calculations

5.3.2.7 Base Index (BX). For Base Index addressing mode, the operand is located at the address calculated by adding the displacement contained in the instruction to both the index value contained in the specified general-purpose index register and the address contained in the specified general-purpose base register. Any word or longword register other than R0 or RR0 can be

used for the index register; any longword register other than RR0 can be used for the base register. The operand is located in one of the data memory address spaces. Base Index addressing mode can be used to access tables or other complex data structures when the base of the table and particular element index are not known until the program is executed.



THE OPERAND VALUE IS THE CONTENTS OF THE LOCATION WHOSE ADDRESS IS THE CONTENTS OF THE BASE REGISTER, PLUS THE CONTENTS OF THE INDEX REGISTER, PLUS THE DISPLACEMENT IN THE INSTRUCTION.

Assembler language syntax:

RRn (Rm)(disp)	Word index register
RRn (RRm)(disp)	Longword index register
	The displacement can be omitted when it is zero.

Example of segmented BX mode:

LDL RR2,RR4 (R3)(1) //load RR2 with the
//longword whose
//address is the base
//address in RR4 + the
//index value in
//R3 + 1

Before Execution

RR2	35	35	FF	FD
RR4	00	01	15	02

Data Memory

Address Calculation

$$\begin{array}{r} \ll\#1>1502 \\ + \quad \quad \quad FFFF \\ + \quad \quad \quad 0001 \\ \hline \ll\#1>1500 \end{array}$$

After Execution

RR2	B0	DE	F7	32
RR4	00	01	15	02

Data Memory

Example of linear BX mode:

LDL RR2,RR4(R3)(1) //load RR2 with the
//longword whose
//address is the base
//address in RR4 plus
//the index value in
//R3 + 1

Before Execution

RR2	35	35	FF	FD
RR4	00	01	15	02

Data Memory

Address Calculation

$$\begin{array}{r} 0001 \ 1502 \\ + \quad \quad \quad FFFF \ FFFD \\ + \quad \quad \quad 0000 \ 0001 \\ \hline 0001 \ 1500 \end{array}$$

After Execution

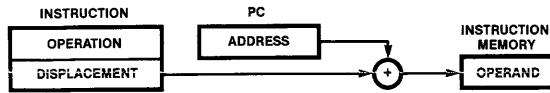
RR2	B0	DE	F7	32
RR4	00	01	15	02

Data Memory

NOTE: The index value in R3 has been sign-extended to 32 bits.

5.3.2.8 Relative Address (RA). For Relative Address addressing mode, the operand is located at the address calculated by adding the displacement contained in the instruction to the Program

Counter. The value used for PC is the address of the instruction word following the displacement. The operand is located in one of the instruction memory address spaces.



THE OPERAND VALUE IS THE CONTENTS OF THE LOCATION WHOSE ADDRESS IS THE CONTENTS OF PC PLUS THE DISPLACEMENT IN THE INSTRUCTION.

Assembler language syntax:

<address>

Example of segmented RA mode:

```
LDL RR24,<$ + 6> //load RR24 with the
//longword whose
//address is the
//address of the
//first word of the
//current instruction
// + 6
```

Because the Program Counter will be advanced to point to the next instruction when the address calculation is performed, the displacement in the instruction is actually +2 (four less than the offset given by the assembler language syntax).

Before Execution

PC	0000	0200
RR24	01234567	

Instruction Memory

			Instruction
31	02	00	02
E8	02	FF	FE
AB	CD	BD	01

Address Calculation

$$\begin{array}{r} \ll s\$0\gg 0204 \\ + 0002 \\ \hline \ll s\$0\gg 0206 \end{array}$$

After Execution

PC	0000	0204
RR24	FFFEABCD	

Example of linear RA mode:

```
LDL RR24,<$ + 6> //load RR24 with the
//longword whose
//address is the
//address of the
//first word of
//the current
//instruction + 6
```

Because the Program Counter will be advanced to point to the next instruction when the address calculation is performed, the displacement in the instruction is actually +2 (four less than the offset given by the assembler language syntax).

Before Execution

PC	0000	0200
RR24	01234567	

Instruction Memory

			Instruction
31	02	00	02
E8	02	FF	FE
AB	CD	BD	01

Address Calculation

$$\begin{array}{r} 0000\ 0204 \\ + 0000\ 0002 \\ \hline 0000\ 0206 \end{array}$$

After Execution

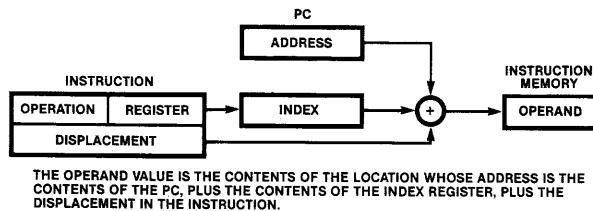
PC	0000	0204
RR24	FFEEABCD	

Note: Brackets (<>) enclosing the address can be omitted for CALR, DJNZ, JR, and LDR instructions.

Addressing Modes and Address Calculations

5.3.2.9 Relative Index (RX). For Relative Index addressing mode, the operand is located at the address calculated by adding the displacement contained in the instruction to both the index value contained in the specified general-purpose register and the Program Counter. Any word or longword register other than R0 or RR0 can be used

for the index register. The value used for PC is the address of the instruction word following the displacement. The operand is located in one of the program memory address spaces. Relative Index addressing mode can be used to access tables of constants.



Assembler language syntax:

<address>(Rn) Word index register
<address>(RRn) Longword index register

Example of segmented RX mode:

```
LDRL RR26, TABLE(RR28)
    //load RR26 with the
    //longword whose
    //address is TABLE plus
    //the index value in
    //RR28. TABLE is a
    //symbol for the begin-
    //ning of a table of
    //constants at offset
    //10016 in the same
    //segment as the
    //instruction
```

Before Execution

RR26	B101ABCD
RR28	00000002

Instruction Memory

59	27	27	18
01	23	28	18

After Execution

RR26	27180123
RR28	00000002

Example of linear RX mode:

LDL RR26, TABLE(RR28)

```
//load RR26 with the
//longword whose
//address is TABLE plus
//the index value in
//RR4. TABLE is a
//symbol for the begin-
//ning of a table of
//constants beginning
//at address 00010100
```

Before Execution

RR26	B101ABCD
RR428	00000002

Instruction Memory

59	27	27	18
01	23	28	18

After Execution

RR26	27180123
RR28	00000002

Note: Brackets enclosing the address (<>) can be omitted for CALR, DJNZ, JR, and LDR instructions.

5.4 EXTENDED ADDRESSING MODES

The instruction encodings for several of the addressing modes use one or more extension words following the opcode. Because the encoding of this group of addressing modes is similar, they are collectively given the name Extended Addressing Modes (EAM). The Extended Addressing Modes for compact and segmented or linear mode are shown in Table 5-1 below. Refer to Section 6.4.3 for more information about Extended Addressing Modes.

Table 5-1. Extended Addressing Modes

Compact	Segmented or Linear
Direct Address	Direct Address
Index	Index
	Base Address
	Base Index
	Relative Address
	Relative Index

Chapter 6. Instruction Set

6.1 INTRODUCTION	EX EXB EXL	dst,src	Exchange
This chapter describes the instruction set of the Z80,000 CPU. An overview of the instruction set, separated into functional groups, is presented first. Next, flags and condition codes are discussed. Finally, a description is provided for each instruction, including a summary of the operation, addressing modes, effect on flags, possible exceptions, assembler language syntax, instruction formats, and simple examples. The bit patterns used to encode various instruction fields are also described.	LD LDB LDL	dst,src	Load
	LDA LDAR	dst,src	Load Address
	LDK LDKL	dst,src	Load Constant
6.2 FUNCTIONAL SUMMARY	LDM	dst,src,num	Load Multiple
This section presents a functional overview of the instruction set. The instructions are separated by function into eleven groups. Within each group, the salient features are described, such as available addressing modes, effect on flags, and possible exceptions. The eleven functional groups are:	LDML mask,src dst,mask		Load Multiple Longwords
<ul style="list-style-type: none"> ● Load and Exchange ● Arithmetic ● Logical ● Program Control ● Bit Manipulation ● Bit Field ● Rotate and Shift ● Block Transfer and String Manipulation ● Input/Output ● CPU Control ● Extended Instructions 	LDR LDRB LDRL	dst,src	Load Relative
	POP POPL		Pop
	PUSH PUSHL	dst,src	Push
	The load and exchange instructions move data between registers and memory. Among these instructions, only Convert and Convert Unsigned affect the flags.		
	The Load instructions transfer a byte, word, or longword of data from the source operand to the destination operand. A register can either be loaded with an operand using any of the addressing modes or a register or immediate value can be loaded to a memory location. The Load Relative instructions load a register to or from a memory location specified with the Relative addressing mode. Special compact encodings are provided for the following frequent operations: (1) loading any constant byte to a register; (2) loading a small constant (0 to 15) word or longword to a register (Load Constant); and (3) loading an immediate value zero to a register or memory location (Clear).		
6.2.1 Load and Exchange Instructions			
Instruction	Operand(s)	Name of Instruction	
CLR	dst	Clear	
CLRB			
CLRL			
CVT	dst,src	Convert	
CVTU	dst,src	Convert Unsigned	

280,000 Instruction Descriptions and Formats

The Exchange instructions swap the byte, word, or longword contents of the source and destination operands. The contents of a register can be swapped with the contents of another register or memory location.	ADD	dst,src	Add
	ADDB		
	ADDL		
	CHK	dst,src	Check
	CHKB		
	CHKL		
	CP	dst,src	Compare
	CPB		
	CPL		
	DAB	dst	Decimal Adjust
The Convert and Convert Unsigned instructions are used to move the byte, word, or longword source operand to a different-sized destination operand. The data can be moved in either direction between a register and another register or memory location. When the destination is longer than the source, Convert performs sign extension and Convert Unsigned performs zero extension. If the destination is shorter than the source, the instructions set the V flag when the lost information is significant. The Integer Overflow trap occurs when the IV bit in FCW is 1 and the Convert instruction sets the V flag.	DEC	dst,src	Decrement
	DECB		
	DECI	dst,src	Decrement
	DECIB		Interlocked
	DECL		
	DIV	dst,src	Divide
	DIVL		
	DIVU	dst,src	Divide Unsigned
	DIVUL		
	EXTS	dst	Extend Sign
The Load Multiple and Load Multiple Longwords instructions provide efficient saving and restoring of registers. They are most useful for moving simple data types that are more than four bytes long and for changing the process context at interrupts. The Load Multiple instruction allows any contiguous group of 1 to 16 word registers to be loaded to or from consecutive memory locations. The Load Multiple Longwords instruction allows up to 16 longword registers selected by a bit mask to be loaded to or from consecutive memory locations.	EXTSB		
	EXTSL		
	INC	dst,src	Increment
	INC B		
	INCL		
	INCI	dst,src	Increment
	INCIL		Interlocked
	INDEX	dst,sub,src	Index
	INDEXL		
	MULT	dst,src	Multiply
Stack operations for words and longwords are supported by the Push and Pop instructions. Any general-purpose register other than R0 or RRO can be used as a stack pointer. The stack pointer is automatically decremented for Push and incremented for Pop. The source operand for Push and the destination operand for Pop can be specified using any of the addressing modes.	MULTL		
	MULTU	dst,src	Multiply Unsigned
	MULTUL		
	NEG	dst	Negate
	NEGB		
	NEGL		
	SBC	dst,src	Subtract with Carry
	SBCB		
	SBCL		
	SUB	dst,src	Subtract
6.2.2 Arithmetic Instructions	SUBB		
	SUBL		
	TESTA	dst	Test Arithmetic
	TESTAB		
	TESTAL		

The arithmetic group consists of instructions for performing integer arithmetic. The basic instructions operate on unsigned binary integers or signed twos complement binary integers. Support is provided for Binary Coded Decimal (BCD) arithmetic and multiple precision arithmetic.

The arithmetic instructions generally affect the C, Z, S, and V flags. The byte versions of these instructions generally affect the D and H flags as well. The V flag indicates arithmetic overflow. The Integer Overflow Trap occurs when the IV bit in the FCW is 1 and the V flag is set after execution of an Add, Decrement, Decrement Interlocked, Divide, Divide Unsigned, Increment, Increment Interlocked, Negate, or Subtract instruction.

Add, Subtract, Multiply, Multiply Unsigned, Divide, and Divide Unsigned instructions operate on a destination operand in a register and a source operand specified by any addressing mode. The result of the operation is stored in the destination. Add and Subtract operate on bytes, words, or longwords. The Multiply instructions operate on words or longwords and compute a double-precision product. The Divide instructions operate on words or longwords, using a double-precision dividend.

The Increment and Decrement instructions add or subtract a small constant (1 to 16) to or from the destination operand. The result is stored in the destination. The operand may be a byte, word, or longword specified in a register or memory location. Increment Interlocked and Decrement Interlocked instructions are similar to Increment and Decrement, but interlock protection is used to fetch and store the destination operand in memory. Interlock protection is important for implementing critical counters referred to by multiple processors.

The Negate instructions perform twos complement on the destination operand in a register or memory location.

The Compare instructions compare (subtract) the source and destination operands and set the flags to reflect the result. The contents of a register can be compared with an operand specified using any addressing mode, and the contents of a memory location can be compared with an immediate value. The Test Arithmetic instructions are special, compact encodings for comparing a register or memory location with zero.

BCD operations are supported with the Decimal Adjust instruction. The DAB instruction is used following the binary addition or subtraction of bytes to adjust the destination operand, specified in a register, for correct BCD representation.

Multiple precision arithmetic is supported with the Add with Carry, Subtract with Carry, and Extend Sign instructions. These instructions operate on byte, word, or longword operands stored only in registers. The Extend Sign instructions compute a double-precision result.

The Check instructions are used to compare the signed byte, word, or longword source operand against lower and upper bounds. The source operand is specified in a register, and the bounds are specified as immediate values or in consecutive memory locations. If the source is out of bounds, a Bounds Check trap occurs.

The Index instruction is used either to compute an index into a one-dimensional array, or as one step in computing the index into a multiple-dimensional array. The signed subscript is compared against lower and upper bounds. If the subscript is out of bounds, an Index Error Trap occurs; otherwise, the lower bound is subtracted from the subscript, and the difference is added to the destination. The sum is then multiplied by the scale factor, and the product is stored back into the destination, which is the calculated array offset. The source and destination operands are specified in registers. The bounds and scale factor are specified as immediate values or in consecutive memory locations. All operands are the same size, either word or longword.

6.2.3 Logical Instructions

Instruction	Operand(s)	Name of Instruction
AND	dst,src	And
ANDB		
ANDL		
COM	dst	Complement
COMB		
COML		
OR	dst,src	Or
ORB		
ORL		
TEST	dst	Test
TESTB		
TESTL		

Z80,000 Instruction Descriptions and Formats

XOR	dst,src	Exclusive Or
XORB		
XORL		

The logical group consists of instructions for performing logical operations on all bits of byte, word, or longword operands; the instructions set the Z and S flags according to the result. The byte versions affect the P flag as well, setting the P flag if the parity of the result is even.

The instructions And, Or, and Exclusive Or operate on a destination operand in a register and a source operand specified with any addressing mode. The appropriate logical operation is performed on bits of the operands, and the result is stored back into the destination.

The Complement instruction complements the bits of the destination operand; the result is stored back into the destination. The operand is a byte, word or longword specified in a register or memory location.

The Test instruction performs a logical Or of the destination operand and zero, and sets the flags according to the result. The operand is a byte, word, or longword specified in a register or memory location.

6.2.4 Program Control Instructions

Instruction	Operand(s)	Name of Instruction
BRKPT		Breakpoint
CALL	dst	Call
CALR		Call Relative
DJNZ	r,dst	Decrement and Jump if Not Zero
DBJNZ		
DLJNZ		
ENTER	mask,siz	Enter
EXIT		Exit
JP	cc,dst	Jump
JR	cc,dst	Jump Relative
RET	cc	Return
SC	src	System Call
TRAP	cc,src	Conditional Trap

This group consists of instructions that control program flow for jumps, loops, procedure calls, and exceptions. The instructions generally do not affect the flags, except when new Program Status is loaded for traps.

The Jump instruction loads the Program Counter (PC) with the effective address of the destination operand if the flags satisfy the specified condition. The destination is specified using any of the memory addressing modes. The Jump Relative instruction is a special, compact encoding used when the destination is within -254 to 256 bytes of the instruction location.

The Call instruction is used for calling procedures. The contents of the PC are pushed onto the processor stack, and the effective address of the destination operand is loaded into the PC. The destination operand is specified using any of the memory addressing modes. The Call Relative instruction is a special, compact encoding used when the destination operand is within -4092 to 4098 bytes of the instruction location.

The Enter instruction is executed at the beginning of a procedure to establish the procedure's environment. Enter adjusts the Frame Pointer and Stack Pointer registers to allocate a new activation record, which contains saved general-purpose registers, the Frame Pointer, the exception handler address, and local data. The instruction contains a bit mask indicating which general-purpose registers to save. The mask and the value of the Integer Overflow Enable bit in FCW are also saved in the activation record. The Call and Enter instructions provide the essential functions for linking procedures in high-level languages such as C and Pascal.

Corresponding to Call and Enter instructions are Return and Exit. Exit releases the activation record by adjusting the Stack Pointer and restoring the Frame Pointer. Exit also uses the mask saved by Enter to restore the saved general-purpose registers and Integer Overflow Enable bit. The Return instruction pops a value from the processor stack into the PC if the flags satisfy the specified condition.

The Decrement and Jump If Not Zero instructions are used to control loops, such as those implementing multiple-precision or decimal-string arithmetic. The specified byte, word, or longword register is decremented by one, and the result is stored back into the register. If the result is not zero, the PC is loaded with the effective address of the destination. The destination may

be specified using Relative Address addressing mode, at a location no more than 252 bytes (DJNZ, DBJNZ) or 250 bytes (DLJNZ) before the instruction.

The Breakpoint, System Call, and Conditional Trap instructions are all used to generate traps. The Breakpoint instruction is generally placed by a debugger at the first word of an instruction where a breakpoint is desired. The System Call instruction is used by programs operating in normal mode to request service from the operating system; the low-order byte of the instruction can be used to indicate the particular service desired. The Conditional Trap instruction generates a trap if the flags satisfy the specified condition. This instruction can be used for software detection of run-time errors or other exceptions; a 4-bit field in the instruction word can be used to identify the cause of the trap. When one of these traps occurs, the CPU pushes the Program Status registers and instruction word onto the system stack, and loads new values into the Program Status registers from the Program Status Area. See Chapter 7 for more details about trap processing.

6.2.5 Bit Manipulation Instructions

Instruction	Operand(s)	Name of Instruction
BIT	dst,src	Bit Test
BITB		
BITL		
RES	dst,src	Reset Bit
RESB		
RESL		
SET	dst,src	Set Bit
SETB		
SETL		
TSET	dst	Test and Set
TSETB		
TSETL		
TCC	cc,dst	Test Condition Code
TCCB		
TCCL		

The instructions in this group are used to manipulate an individual bit in a byte, word, or longword destination operand. Set Bit is used to set a bit to 1; Reset Bit clears a bit to 0. The bit of the destination operand specified by the source operand is set or cleared, and the result is stored back into the destination. The Bit Test

instruction tests the bit of the destination specified by the source operand, and sets the Z flag to indicate the result. For "static"^{*} bit operations, the source operand is specified by an immediate value and the destination operand may be in a register or memory location. For "dynamic" bit operations, the source and destination operands are in registers.

The Test Condition Code instruction sets the least-significant bit of the byte, word, or longword destination register if the flags satisfy the specified condition. This instruction is useful for evaluating Boolean expressions.

The Test and Set instruction tests whether the destination is negative, then sets all bits in the destination to 1. Interlock protection is used to fetch and store the destination operand in memory. Test and Set is used to access semaphores protecting critical shared data structures in a tightly-coupled multiprocessor system.

6.2.6 Bit Field Instructions

Instruction	Operand(s)	Name of Instruction
EXTR	dst, src, pos, siz	Extract Field
EXTRU		
INSRT	dst, src, pos, siz	Insert Field

The instructions in this group are used to insert and extract bit fields. A bit field is 1 to 32 contiguous bits that can cross byte boundaries. One version of Extract (EXTR) is used to extract and sign-extend a field into the destination longword register. Another version of Extract (EXTRU) extracts and zero-extends the field. Insert is used to insert a field from the source longword register.

A bit field is specified by three operands as follows: (Figure 6-1).

- The origin of the bit string is the most-significant bit of a memory location or longword register. The origin is specified by the source operand for Extract and the destination operand for Insert.

* The term "static" is used because the bit number is an immediate value that cannot change. "Dynamic" means the bit number is specified in a register and can change.

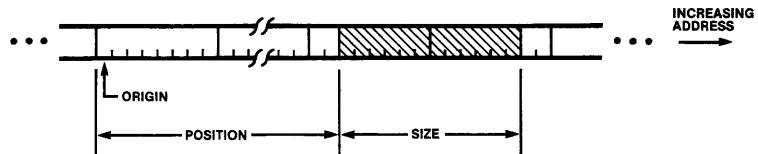


Figure 6-1. Bit Field

- The position of the field is the unsigned number of bits from the origin to the most-significant bit of the field. Position is measured in the direction of decreasing significance from the origin. The position of the origin is zero. The position is specified by an immediate value (0 to 31) or in a word or longword register. In the latter case the position may be any positive value.

- The size of the field is the number of bits in the field, between 0 and 31 inclusive, and represents fields of 1 to 32 bits. The size is specified by an immediate value or in a word or longword register.

A bit field in memory must be contained entirely within four consecutive bytes (i.e., the position modulo 8 plus the size operand must be less than or equal to 31). A bit field in a longword register must be entirely contained within the register (i.e., the position plus the size operand must be less than or equal to 31).

Note that the direction of increasing bit number for field position is opposite to Figure 2-1.

RRDB	dst,src	Rotate Right Digit
SDA	dst,src	Shift Dynamic
SDAB		Arithmetic
SDAL		
SDL	dst,src	Shift Dynamic Logical
SDLB		
SDLL		
SLA	dst,src	Shift Left Arithmetic
SLAB		
SLAL		
SLL	dst,src	Shift Left Logical
SLLB		
SLLL		
SRA	dst,src	Shift Right Arithmetic
SRAB		
SRAL		
SRL	dst,src	Shift Right Logical
SRLB		
SRLL		

This group of instructions provides for rotating and shifting of bytes, words, and longwords of data located in general-purpose registers. The Rotate and Shift instructions affect the C, Z, S, and P/V flags.

The Rotate instructions rotate the contents of the destination register left or right by an amount specified by the source operand. The source is an immediate value of one or two. Rotation is performed on the destination alone or, for multiple precision arithmetic, on both the destination and Carry bit. The digit rotation instructions RLDB and RRDB are useful for manipulating BCD data.

The Shift instructions shift the contents of the destination register left or right by an amount specified by the source operand. The value of the source operand can be any amount between zero and the number of bits in the destination. For "static" shift operations, the source is specified

6.2.7 Rotate and Shift Instructions

Instruction	Operand(s)	Name of Instruction
RL	dst,src	Rotate Left
RLB		
RLL		
RLC	dst,src	Rotate Left through Carry
RLCB		
RLCL		
RLDB	dst,src	Rotate Left Digit
RR	dst,src	Rotate Right
RRB		
RRL		
RCR	dst,src	Rotate Right through Carry
RCRB		
RCRL		

280,000 Instruction Descriptions and Formats

by an immediate value; for "dynamic" shift operations the source is specified in a register. Both logical and arithmetic shifts are supported. An Integer Overflow Trap occurs when the IV bit of FCW is 1 and the V flag is set after execution of an arithmetic shift instruction.			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
6.2.8 Block Transfer and String Manipulation Instructions			TRTDB	src1,src2,r	Translate, Test and Decrement
			TRTDRB	src1,src2,r	Translate, Test, Decrement, and Repeat
CPD	dst,src,r,cc	Compare and Decrement	TRTIB	src1,src2,r	Translate, Test and Increment
CPDB			TRTIRB	src1,src2,r	Translate, Test, Increment and Repeat
CPDL					
CPDR	dst,src,r,cc	Compare, Decrement and Repeat	This group of instructions provides a full complement of string comparison, string translation, and block transfer operations. A block can be moved in memory, a string can be searched for a given value, and two strings can be compared. These instructions manipulate blocks or strings containing up to 65,536 bytes, words, or longwords. In addition, a string containing up to 65,536 bytes can be translated according to a table in memory, or searched for a set of values specified by a table in memory.		
CPDRB					
CPDRL					
CPI	dst,src,r,cc	Compare and Increment	The block and string operands are specified using Indirect Register addressing mode. When a string is searched for a value, the value is located in a register. The length of the block or string is also located in a register.		
CPIB					
CPIL					
CPIR	dst,src,r,cc	Compare, Increment and Repeat	All the block transfer and string manipulation operations can proceed through the data in either direction. Furthermore, the operations can be repeated automatically while decrementing the length register until it is zero, or they can operate on a single element with the length register decremented by one and the pointer registers properly adjusted. The second form can be used with other instructions in a loop to implement more complex string operations.		
CPIRB					
CPIRL					
CPSD	dst,src,r,cc	Compare String and Decrement	These instructions set the P/V flag to indicate whether the length register was decremented to zero. The string Search and Compare instructions set the C, Z, and S flags to indicate the result of the comparison. The Translate and Test instructions set the Z flag when one of the specified set of values is found. Otherwise, the flags are unaffected.		
CPSDB					
CPSDL					
CPSDR	dst,src,r,cc	Compare String, Decrement and Repeat			
CPSDRB					
CPSDRL					
CPSI	dst,src,r,cc	Compare String and Increment			
CPSIB					
CPSIL					
CPSIR	dst,src,r,cc	Compare String, Increment and Repeat			
CPSIRB					
CPSIRL					
LDD	dst,src,r	Load and Decrement	The repetitive forms of these instructions are interruptible after each iteration. Section 7.3.1 provides more information about interruptible instructions.		
LDDB					
LDL					
LDDB	dst,src,r	Load, Decrement and Repeat			
LDDBR					
LDRL					
LDI	dst,src,r	Load and Increment			
LDIB					
LDIL					
LDIR	dst,src,r	Load, Increment and Repeat			
LDIRB					
LDIRL					

Z80,000 Instruction Descriptions and Formats

6.2.9 Input/Output Instructions

Instruction	Operand(s)	Name of Instruction	The other instructions in the group are used to transfer a block (up to 65,536 bytes, words, or longwords of data) between a peripheral port and memory. The port address and memory address are specified using Indirect Register addressing mode. The length of the block is located in a register. These instructions are similar to the block move instructions described in Section 6.2.7 except that the port address remains unchanged while the memory address is adjusted. The P/V flag is set when the length register is decremented to zero. The repetitive forms of these instructions are interruptible after each iteration.
IN	dst,src	Input	
INB			
INL			
IND	dst,src,r	Input and Decrement	
INDB			
INDL			
INDR	dst,src,r	Input,Decrement and Repeat	
INRB			
INRL			
INI	dst,src,r	Input and Increment	
INIB			
INIL			

6.2.10 CPU Control Instructions

Instruction	Operand(s)	Name of Instruction
INIR	dst,src,r	Input, Increment and Repeat
INIRB		
INIRL		
DI		Disable Interrupt
OTDR	dst,src,r	Output, Decrement and Repeat
OTDRB		
OTDRL		
EI		Enable Interrupt
HALT		Halt
OTIR	dst,src,r	Output, Increment and Repeat
OTIRB		
OTIRL		
IRET		Interrupt Return
LDCTL		Load Control Register
LDCTLB		
LDCTLL		
OUT	dst,src	Output
OUTB		
OUTL		
LDND		Load Normal Data
LDNDB		
LDNDL		
OUTD	dst,src,r	Output and Decrement
OUTDB		
OUTDL		
LDNI		Load Normal Instruction
LDNIB		
LDNIL		
OUTI	dst,src,r	Output and Increment
OUTIB		
OUTIL		
LDPND		Load Physical Address
LDPNI		
LDPSD		
LDPSI		
LDPS	src	Load Program Status
NOP		No Operation
PCACHE		Purge Cache
PTLB		Purge TLB

The instructions in this group transfer data between a peripheral port and a CPU register or memory. All of these instructions are privileged.

A single byte, word, or longword of data can be transferred between a peripheral port and a CPU register with the Input and Output instructions. The port address is specified using the Direct Address or Indirect Register addressing modes. The single transfer instructions do not affect the flags.

PTLBEND			TLB. Individual TLB entries can be invalidated using the Purge TLB Entry instructions. All the normal mode TLB entries can be invalidated using the Purge TLB Normal instruction.
PTLBENI	Purge TLB Entry		
PTLBESD			
PTLBESI			
PTLBN	Purge TLB Normal		
RESFLG	flag	Reset Flag	
SETFLG	flag	Set Flag	

The instructions in this group perform privileged operations necessary for the operating system to control the CPU; only the No Operation and flag manipulation (COMFLG, LDCTLB, RESFLG, SETFLG) instructions can be executed in normal mode. The only instructions that affect the flags are the flag manipulation instructions, the instructions that load the FCW (IRET, LDCTL, LDPS), and the Load Physical Address instructions.

The Disable Interrupt and Enable Interrupt instructions control the Vectored Interrupt and Non-Vectored Interrupt enable bits in FCW. The enable bits can be separately cleared or set.

The Halt instruction halts the CPU.

The Interrupt Return instruction is used to return from an interrupt or trap handler. The Program Status registers are loaded with values popped from the system stack.

The Load Control instructions move data between a control register and a general-purpose register. The Load Program Status instruction loads the Program Status registers (PC, FCW) from memory. The memory location is specified using the IR or EAM addressing modes.

Load Normal Data and Load Normal Instruction are used in system mode to move data between a register and a memory location in either of the normal mode memory address spaces. The memory location is specified using the IR or EAM addressing modes.

The Load Physical Address instructions load the physical address of the source operand to the destination register. The source operand is specified using the IR or EAM addressing modes. These instructions set the flags to indicate the access protection of the logical address and whether the address translation was valid.

The Purge Cache instruction invalidates the cache contents. The Purge TLB instruction invalidates all address translation table entries in the

6.2.11 Extended Instructions

The Z80,000 architecture includes a powerful mechanism for extending the basic instruction set through the use of coprocessors known as Extended Processing Units (EPUs). For example, floating-point arithmetic is supported by the Z8070 Arithmetic Processing Unit. When an extended instruction is executed and the EPA bit in the FCW is 1, the CPU transfers the instruction to the EPU. The CPU also controls the transfer of data between the EPU and either memory or the CPU. If the EPA bit is 0, an Extended Instruction trap occurs to allow software emulation in systems that lack an EPU.

The CPU supports four types of extended instructions: EPU internal operations that do not require any data transfer; transfer of one to sixteen words of data between the EPU and consecutive word or longword general-purpose registers; transfer of one byte of data between the EPU and the flag byte of the FCW; and the transfer of one to sixteen bytes or words of data between the EPU and memory. The flags are affected only when the flag byte is loaded.

6.3 FLAGS AND CONDITION CODES

The Program Status includes six processor flags as follows: Carry (C), Zero (Z), Sign (S), Parity/Overflow (P/V), Decimal Adjust (D), and Half Carry (H). These flags are affected or tested by most instructions. Arithmetic, logical, and other instructions previously described modify the flags to indicate the result of the operation. Among the instructions that test whether or not the flags indicate a specified condition are Jump, Return, and Test Condition Code. For example, a Test instruction may be followed by a 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.

Z80,000 Instruction Descriptions and Formats

The Carry (C) flag is set to 1 following certain operations when there is a carry from or a borrow into the high-order bit position of the result.

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	1
+ 64	0	1	0	0	0	0	0
289	0	0	1	0	0	0	1
	1	= Carry flag					

The Carry flag is important for implementing 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. These instructions are used to implement rotation or shifting of data.

The Zero (Z) flag is set to 1 when the result of certain operations is zero. This flag is useful to determine 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 1 when the result of certain operations is negative (i.e., the most-significant bit is 1).

The Overflow (V) flag is set to 1 when the result of certain operations cannot be represented as a two's complement number in the same precision as the destination. In the example below for 8-bit numbers, 120 is added to 105. The result, 225, cannot be represented in 8 bits; it appears to be -31. In such a case, the Overflow flag is set and only the low-order bits of the result are stored into the destination.

Bit							
7	6	5	4	3	2	1	0
120	0	1	1	1	1	0	0
+ 105	0	1	1	0	1	0	0
225	1	1	1	0	0	0	1
	1	= Overflow flag set					

The Parity (P) flag is set to 1 when the result of logical operations on bytes has even parity (i.e., the number of 1 bits is even). The Overflow and Parity flags share the same bit in the FCW, hence the bit is named P/V.

The Decimal Adjust (D) and Half-Carry (H) flags are used for BCD arithmetic. Following the binary addition of two bytes, the D flag is set and the H flag indicates the carry from bit 3. Following the binary subtraction of two bytes the D flag is cleared and the H flag indicates the borrow from bit 3. Decimal arithmetic on BCD bytes is performed by first adding or subtracting the operands using binary arithmetic. Afterwards, the Decimal Adjust instruction adjusts the result for correct BCD representation.

The C, Z, S, and P/V flags are also used to control the operation of conditional instructions such as Jump. The operation of these instructions depends on whether the four flags satisfy a specified condition. Conditional instructions contain a 4-bit field, called the condition code, that specifies one of sixteen flag conditions to test. Table 6-1 lists the flag condition tested and the binary encodings for the condition codes.

6.4 NOTATION AND BINARY ENCODING

The rest of this chapter contains detailed descriptions for each instruction, listed in alphabetical order. This section describes the notational conventions used in the instruction descriptions and the binary encoding for some common instruction fields (e.g., register designation fields). The bit patterns for other instruction fields are shown explicitly in the instruction format.

An instruction's description begins with the instruction mnemonic and instruction name in the top part of the page. Privileged instructions are also identified as such at the top of the page.

The assembler language syntax is then given in a general form that covers all the variants of the instruction and the order of source, destination and other operands, along with a list of applicable addressing modes.

Example:

```
AND dst, src      dst: R
ANDB              src: R, IM, IR, EAM
ANDL
```

Table 6-1. Condition Codes

Code	Meaning	Flag Setting	Binary
F	Always false	-	0000
T	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

Some condition codes correspond to identical flag settings: Z-EQ, NZ-NE, C-ULT, NC-UGE, PE-OV, and PO-NOV. If no condition is specified, the default condition is T (always true).

The operation of the instruction is presented next, followed by a detailed discussion of the instruction, including the effect of the instruction on the processor flags. Exceptions that can occur for the instruction are listed next. Some exceptions, such as the Address Translation trap, can occur for any instruction. Only exceptions specific to the instruction are listed.

Finally, a table is presented showing the assembler language syntax and instruction format for each addressing mode and operand size. An assembler language example showing the use of the instruction is also given.

6.4.1 Assembler Language Syntax

The syntax is shown for each operand size (byte, word or longword). 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 substituted. The syntax is shown 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,#\$5. The assembler also accepts variations such as ADD TOTAL, #NEW-DELTA where TOTAL, NEW and DELTA have been previously defined.

When the assembler syntax can be encoded in more than one format (e.g., LDB RH0, #1), the assembler generally uses the shortest encoding.

The following notation is used for registers:

Rbd,Rbs	a byte register (RH0,RH1,...,RH7,RL0, RL1,...,RL7)
Rd,Rs	a word register (R0,R1,...,R15)
RRd,RRs	a longword register (R0,R1,...,R15)
RQd	a quadword register (R0,R1,...,R15)

The ending "s" or "d" for the register notation indicates either a source or destination operand, respectively. Address registers must be word registers in compact mode and longword registers in segmented or linear mode, as explained in footnotes to applicable instructions.

Several addressing modes are combined together in a group called Extended Addressing Modes (EAM).

Z80,000 Instruction Descriptions and Formats

The instruction encoding for these addressing modes requires one or more extension words following the opcode. In compact mode, the EAMs are Direct Address and Index (Base Address and Index addressing modes are equivalent in compact mode.) In segmented or linear mode, the EAMs are Direct Address, Index, Base Address, Base Index, Relative Address, and Relative Index. Where the symbol "eam" is found in the assembler syntax, any EAM can be used. Refer to Section 5.3 for the assembler syntax for particular addressing modes.

Conditional instructions specify a condition code, indicated by "cc" in the assembler syntax. Table 6-1 lists the assembler mnemonics for condition codes.

The assembler recognizes comments beginning with "//" and continuing to the end of the line.

6.4.2 Instruction Format

The binary encoding of each instruction is given as part of the instruction description. Some fields in the instruction contain symbols whose values are described below.

The symbol "W" is used for a single bit that distinguishes between the byte and word versions of the instruction. The bit takes the value 0 for byte versions and 1 for word versions.

Fields specifying registers are identified with the same symbol (Rs, RRd, etc.) used in the assembler language syntax. When the field cannot take the value 0, a notation of the form "Rs#0" is used. Table 6-2 shows the binary encoding for register fields.

Table 6-2. Register Field Encoding

Code	Byte	Word	Long	Quad
0000	RH0	R0	RR0	RQ0
0001	RH1	R1	RR16	RQ16
0010	RH2	R2	RR2	Unimplemented
0011	RH3	R3	RR18	Unimplemented
0100	RH4	R4	RR4	RQ4
0101	RH5	R5	RR20	RQ20
0110	RH6	R6	RR6	Unimplemented
0111	RH7	R7	RR22	Unimplemented
1000	RL0	R8	RR8	RQ8
1001	RL1	R9	RR24	RQ24
1010	RL2	R10	RR10	Unimplemented
1011	RL3	R11	RR26	Unimplemented
1100	RL4	R12	RR12	RQ12
1101	RL5	R13	RR28	RQ28
1110	RL6	R14	RR14	Unimplemented
1111	RL7	R15	RR30	Unimplemented

For bit field instructions, the position and size operands are specified by a 6-bit field. The operands can be immediate values or located in a word or longword register. The format of the field is shown below.

0	n	n	n	n	n	5-bit immediate value (0 to 31)
1	0	r	r	r	r	word register
1	1	r	r	r	r	longword register

6.4.3 Extended Addressing Modes (EAM)

The format for instructions using an EAM includes an opcode word containing a 4-bit field indicated by "eam", followed by one, two, or three extension words. An example is shown below.

The following sections describe the various encoding possibilities for EAM. An EAM format specifies the three components of an effective address calculation: base address, index value, and displacement. Refer to Section 5.2 for more information about effective address calculations.

Assembler Language Syntax

ADDL RRd,eam

instruction Format

01	010110	eam	RRd
1, 2 or 3 extension words			

6.4.3.1 Compact Mode. In compact mode, the EAM format is used for Direct Address or Index addressing modes. The opcode is followed by a single extension word containing the base address

used in effective address calculation. The eam field specifies a word index register ($\text{eam} \neq 0$) or no index register ($\text{eam}=0$).

Addressing Modes

eam	Mode
0	DA
$\neq 0$	X (word index)

Instruction Format

01		eam	
address			

6.4.3.2 Segmented or Linear Mode. In segmented or linear mode, there are six EAM formats used for Direct Address, Index, Base Address, Base Index, Relative, and Relative Index addressing modes. The six formats are distinguished by the encoding of the most-significant bit and the four least-significant bits of the first extension word. The most frequently used formats require only a single extension word, but formats with two and three extension words are provided to access the entire

address space. The formats are described below.

The first format uses a single extension word to specify Base Address or Relative Address addressing modes. The eam field specifies the base address for the effective address calculation in a longword register ($\text{eam} \neq 0$) or the Program Counter ($\text{eam}=0$). The extension word encodes a displacement in the range -8192 to 8191 inclusive.

Addressing Modes

eam	Mode
0	RA
$\neq 0$	BA

Instruction Format

01		eam	
1	displacement		1

Z80,000 Instruction Descriptions and Formats

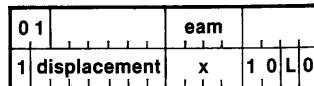
The second format uses a single extension word to specify Base Address, Base Index, Relative Address, or Relative Index addressing modes. The eam field specifies the base address for the effective address calculation in a longword register (eam \neq 0) or the Program Counter (eam=0).

The x field specifies an index register (x \neq 0) or no index register. When an index register is specified, the L field determines whether a longword (L=1) or word (L=0) register is used. The extension word encodes a displacement in the range -64 to 63 inclusive.

Addressing Modes

eam	x	L	Mode
0	0	0	RA
0	0	1	unimplemented
0	\neq 0	0	RX (word index)
0	\neq 0	1	RX (long index)
\neq 0	0	0	BA
\neq 0	0	1	unimplemented
\neq 0	\neq 0	0	BX (word index)
\neq 0	\neq 0	1	BX (long index)

Instruction Format



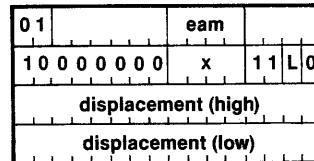
The third format uses three extension words to specify Base Address, Base Index, Relative Address, or Relative Index addressing modes. The encoding of the eam, x, and L fields is the same

as the previous format, but a 32-bit displacement is contained in the second and third extension words.

Addressing Modes

eam	x	L	Mode
0	0	0	RA
0	0	1	unimplemented
0	\neq 0	0	RX (word index)
0	\neq 0	1	RX (long index)
\neq 0	0	0	BA
\neq 0	0	1	unimplemented
\neq 0	\neq 0	0	BX (word index)
\neq 0	\neq 0	1	BX (long index)

Instruction Format



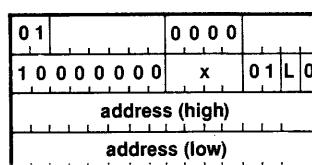
The fourth format uses three extension words to specify Direct Address or Index addressing modes. The base address used in the effective address calculation is contained in the second and third extension words. This format can be used to specify any address. The x field specifies an

index register (x \neq 0) or no index register (x = 0). When an index register is specified, the L field determines whether a longword (L = 1) or word (L = 0) register is used. Note that the eam field must be all 0s in this format.

Addressing Modes

x	L	Mode
0	0	DA
0	1	unimplemented
\neq 0	0	X (word index)
\neq 0	1	X (long index)

Instruction Format



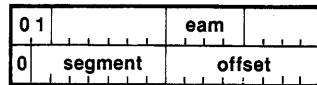
The fifth format uses a single extension word to specify Direct Address or Index addressing modes. The base address used in the effective address calculation is encoded in the extension word. In segmented mode, this format can be used to specify addresses in a 64K-byte segment with the eight

least-significant bits of the segment number and eight most-significant bits of the offset equal to 0. In linear mode, the CPU similarly decodes the address in the extension word, but this format is less useful. The eam field specifies a word index register ($\text{eam} \neq 0$) or no index register ($\text{eam} = 0$).

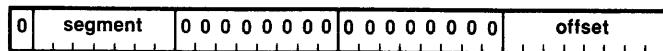
Addressing Modes

eam	Mode
0	DA
$\neq 0$	X (word index)

Instruction Format



Encoded Address



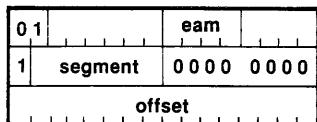
The sixth format uses two extension words to specify Direct Address or Index addressing modes. The base address used in the effective address calculation is encoded in the extension words. In segmented mode, this format can be used to specify addresses in a 64K byte segment with the eight

least-significant bits of the segment number equal to 0. In linear mode, the CPU similarly decodes the address in the extension words, but this format is less often used. The eam field specifies a word index register ($\text{eam} \neq 0$) or no index register ($\text{eam} = 0$).

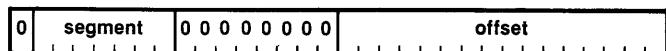
Addressing Modes

eam	Mode
0	DA
$\neq 0$	X (word index)

Instruction Format



Encoded Address



6.4.4 Unimplemented Instruction Encodings

Section 6.5 lists all of the instruction encodings for which the CPU's operation is defined. Any instruction encodings not listed are unimplemented and must not be used. For most of the unimplemented instruction encodings, including all those with first byte 3616 or BF16 and certain Z8000 opcodes described in Appendix A, an attempt to execute the instruction causes an Unimplemented

Instruction trap to occur. If a program erroneously uses an unimplemented instruction that does not trap, the CPU's operation is not specified; however, the CPU never performs an operation that could not otherwise be performed by executing a sequence of defined instructions. For example, a program executing in normal mode cannot gain access to privileged control registers or memory locations by executing an instruction with an unimplemented encoding.

ADC

Add With Carry

ADC

Operation: $\text{dst} \leftarrow \text{dst} + \text{src} + c$

The source operand, along with the setting of the C flag, is added to the destination operand and the sum is stored in the destination. The contents of the source are not affected. Twos 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, ADCL—unaffected; ADCB—cleared
	H: ADC, ADCL—unaffected; ADCB—set if there is a carry from the most-significant bit of the low-order four bits of the result; cleared otherwise

Exceptions: None

Addressing Mode	Assembler Language Syntax	Instruction Format													
R:	ADC Rd, Rs ADCB Rbd, Rbs ADCL RRs, RRs	<table border="1" data-bbox="876 1127 1191 1148"> <tr> <td>10</td> <td>11010</td> <td>W</td> <td>Rs</td> <td>Rd</td> </tr> </table> <table border="1" data-bbox="876 1182 1191 1205"> <tr> <td colspan="2">01111010</td> <td colspan="2">0000 0010</td> </tr> <tr> <td>10</td> <td>110101</td> <td>RRs</td> <td>RRd</td> </tr> </table>	10	11010	W	Rs	Rd	01111010		0000 0010		10	110101	RRs	RRd
10	11010	W	Rs	Rd											
01111010		0000 0010													
10	110101	RRs	RRd												

Example: Quadword addition can be done with the following instruction sequence, assuming RQ0 contains one operand and RQ4 contains the other operand:

ADDL RR2,RR6 //add low-order longwords
ADCL RR0,RR4 //add carry and high-order longwords

If RR0 contains %00000000, RR2 contains %FFFFFFF, RR4 contains %00004320 and RR6 contains %00000001, then executing the two instructions above leaves the value %00004321 in RR0 and %00000000 in RR2.

ADD

Add

ADD dst, src ADDB ADDL	dst: R src: R, IM, IR, EAM
---	-------------------------------

Operation: $dst \leftarrow dst + src$

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
---------------	--

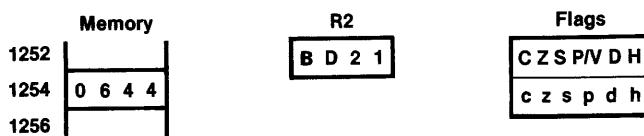
Exceptions: Integer Overflow trap

Source Addressing Mode	Assembler Language Syntax	Instruction Format																															
R:	ADD Rd, Rs ADDB Rbd, Rbs	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>W</td><td>Rs</td><td>Rd</td> </tr> </table>	1	0	0	0	0	0	W	Rs	Rd																						
1	0	0	0	0	0	W	Rs	Rd																									
ADDL RRd, RRs	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td><td>RRs</td><td>RRd</td> </tr> </table>	1	0	0	1	0	1	1	RRs	RRd																							
1	0	0	1	0	1	1	RRs	RRd																									
IM:	ADD Rd, #data	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>Rd</td> </tr> <tr> <td colspan="10">data</td> </tr> </table>	0	0	0	0	0	0	1	0	0	Rd	data																				
0	0	0	0	0	0	1	0	0	Rd																								
data																																	
ADDB Rbd, #data	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>Rbd</td> </tr> <tr> <td colspan="5">data</td> <td colspan="4">data</td> </tr> </table>	0	0	0	0	0	0	0	0	Rbd	data					data																	
0	0	0	0	0	0	0	0	Rbd																									
data					data																												
IR:	ADDL RRd, #data	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>RRd</td> </tr> <tr> <td colspan="10">data (high)</td> </tr> <tr> <td colspan="10">data (low)</td> </tr> </table>	0	0	0	1	0	1	1	0	0	0	RRd	data (high)										data (low)									
0	0	0	1	0	1	1	0	0	0	RRd																							
data (high)																																	
data (low)																																	
ADD Rd, @Rs ¹ ADDB Rbd, @Rs ¹	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>W</td><td>Rs ≠ 0</td><td>Rd</td> </tr> </table>	0	0	0	0	0	0	0	W	Rs ≠ 0	Rd																						
0	0	0	0	0	0	0	W	Rs ≠ 0	Rd																								
	ADDL RRd, @Rs ¹	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>Rs ≠ 0</td><td>RRd</td> </tr> </table>	0	0	0	1	0	1	1	0	Rs ≠ 0	RRd																					
0	0	0	1	0	1	1	0	Rs ≠ 0	RRd																								

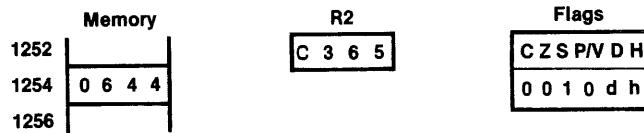
Source Addressing Mode	Assembler Language Syntax	Instruction Format										
EAM:	ADD Rd, eam ADDB Rbd, eam	<table border="1"> <tr> <td>01</td> <td>00000</td> <td>W</td> <td>eam</td> <td>Rd</td> </tr> <tr> <td colspan="5">1, 2, or 3 extension words</td> </tr> </table>	01	00000	W	eam	Rd	1, 2, or 3 extension words				
01	00000	W	eam	Rd								
1, 2, or 3 extension words												
ADDL RRd, eam	<table border="1"> <tr> <td>01</td> <td>010110</td> <td>eam</td> <td>RRd</td> </tr> <tr> <td colspan="4">1, 2, or 3 extension words</td> </tr> </table>	01	010110	eam	RRd	1, 2, or 3 extension words						
01	010110	eam	RRd									
1, 2, or 3 extension words												

Example: ADD R2, %1254 //add the word at %1254 to R2 in compact mode

Before instruction execution:



After instruction execution:



Note 1: Word register in compact mode, longword register in segmented or linear modes.

AND

And

AND dst, src	dst: R
ANDB	src: R, IM, IR, EAM
ANDL	

Operation: dst \leftarrow 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 1 bit is stored wherever the corresponding bits in the two operands are both 1s; otherwise a 0 bit is stored. The contents of the source 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, ANDL— unaffected; ANDB — set if parity of the result is even; cleared otherwise
- D:** Unaffected
- H:** Unaffected

Exceptions: None

Source Addressing Mode	Assembler Language Syntax	Instruction Format														
R:	AND Rd, Rs ANDB Rbd, Rbs	<table border="1"> <tr> <td>10</td> <td>00011</td> <td>W</td> <td>Rs</td> <td>Rd</td> </tr> </table>	10	00011	W	Rs	Rd									
10	00011	W	Rs	Rd												
ANDL RRd, RRs	<table border="1"> <tr> <td>01111010</td> <td>0000 0010</td> </tr> <tr> <td>10</td> <td>000111</td> <td>RRs</td> <td>RRd</td> </tr> </table>	01111010	0000 0010	10	000111	RRs	RRd									
01111010	0000 0010															
10	000111	RRs	RRd													
IM:	AND Rd, #data	<table border="1"> <tr> <td>00</td> <td>000111</td> <td>0000</td> <td>Rd</td> </tr> <tr> <td colspan="4">data</td> </tr> </table>	00	000111	0000	Rd	data									
00	000111	0000	Rd													
data																
ANDB Rbd, #data	<table border="1"> <tr> <td>00</td> <td>000110</td> <td>0000</td> <td>Rbd</td> </tr> <tr> <td colspan="2">data</td> <td colspan="2">data</td> </tr> </table>	00	000110	0000	Rbd	data		data								
00	000110	0000	Rbd													
data		data														
	ANDL RRd, #data	<table border="1"> <tr> <td>01111010</td> <td>0000 0010</td> </tr> <tr> <td>00</td> <td>000111</td> <td>0000</td> <td>RRd</td> </tr> <tr> <td colspan="4">data (high)</td> </tr> <tr> <td colspan="4">data (low)</td> </tr> </table>	01111010	0000 0010	00	000111	0000	RRd	data (high)				data (low)			
01111010	0000 0010															
00	000111	0000	RRd													
data (high)																
data (low)																

Source Addressing Mode	Assembler Language Syntax	Instruction Format										
IR:	AND Rd, @Rs ¹ ANDB Rbd, @Rs ¹	<table border="1"> <tr> <td>00</td> <td>00011</td> <td>W</td> <td>Rs≠0</td> <td>Rd</td> </tr> </table>	00	00011	W	Rs≠0	Rd					
00	00011	W	Rs≠0	Rd								
ANDL RRd, @Rs ¹	<table border="1"> <tr> <td>01111010</td> <td>0000 0010</td> </tr> <tr> <td>00</td> <td>000111</td> <td>Rs≠0</td> <td>RRd</td> </tr> </table>	01111010	0000 0010	00	000111	Rs≠0	RRd					
01111010	0000 0010											
00	000111	Rs≠0	RRd									
EAM:	AND Rd, eam ANDB Rbd, eam	<table border="1"> <tr> <td>01</td> <td>00011</td> <td>W</td> <td>eam</td> <td>Rd</td> </tr> <tr> <td colspan="5">1, 2, or 3 extension words</td></tr> </table>	01	00011	W	eam	Rd	1, 2, or 3 extension words				
01	00011	W	eam	Rd								
1, 2, or 3 extension words												
ANDL RRd, eam	<table border="1"> <tr> <td>01111010</td> <td>0000 0010</td> </tr> <tr> <td>01</td> <td>000111</td> <td>eam</td> <td>RRd</td> </tr> <tr> <td colspan="5">1, 2, or 3 extension words</td></tr> </table>	01111010	0000 0010	01	000111	eam	RRd	1, 2, or 3 extension words				
01111010	0000 0010											
01	000111	eam	RRd									
1, 2, or 3 extension words												

Example: ANDB RL3, # %CE

Before instruction execution:

RL3	Flags
11100111	C Z S P/V D H c z s p d h

After instruction execution:

RL3	Flags
11000110	C Z S P/V D H c 0 1 1 d h

Note 1: Word register in compact mode, longword register in segmented or linear modes.

BIT

Bit Test

BIT dst, src	dst: R, IR, EAM
BITB	src: IM
BITL	or dst: R src: R

Operation: $Z \leftarrow \text{NOT dst} <\text{src}>$

The specified bit within the destination operand is tested, and the Z flag is set to 1 if the specified bit is 0; otherwise the Z flag is cleared to 0. The contents of the destination are not affected. The bit number (the source) can be specified either as an immediate value (static), or as a word register that contains the value (dynamic). In the dynamic case, the destination operand must be in a register, and the source operand must be in a word register.

The bit number is a value from 0 to 7 for BITB, 0 to 15 for BIT, or 0 to 31 for BITL with 0 indicating the least-significant bit. Only the lower three bits of the source operand are used to specify the bit number for BITB, only the lower four bits are used for BIT, and only the lower five bits are used for BITL.

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

Exceptions: None

Bit Test Static

Destination Addressing Mode	Assembler Language Syntax	Instruction Format																							
R:	BIT Rd, #b BITB Rbd, #b	<table border="1" style="margin-left: auto; margin-right: auto;"><tr><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>W</td><td>Rd</td><td>b</td></tr></table>	1	0	1	0	0	1	1	W	Rd	b													
1	0	1	0	0	1	1	W	Rd	b																
BITL RRRd, #b	<table border="1" style="margin-left: auto; margin-right: auto;"><tr><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>b</td><td>RRd</td><td>b</td></tr></table>	0	1	1	1	1	0	1	0	0	0	0	0	1	0	1	0	1	0	0	1	1	b	RRd	b
0	1	1	1	1	0	1	0	0	0	0	0	1	0												
1	0	1	0	0	1	1	b	RRd	b																
IR:	BIT @Rd ¹ , #b BITB @Rd ¹ , #b	<table border="1" style="margin-left: auto; margin-right: auto;"><tr><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>W</td><td>Rd≠0</td><td>b</td></tr></table>	0	0	1	0	0	1	1	W	Rd≠0	b													
0	0	1	0	0	1	1	W	Rd≠0	b																
BITL @Rd ¹ , #b	<table border="1" style="margin-left: auto; margin-right: auto;"><tr><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td></tr><tr><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>b</td><td>Rd≠0</td><td>b</td></tr></table>	0	1	1	1	1	0	1	0	0	0	0	0	1	0	0	0	1	0	0	1	1	b	Rd≠0	b
0	1	1	1	1	0	1	0	0	0	0	0	1	0												
0	0	1	0	0	1	1	b	Rd≠0	b																
EAM:	BIT eam, #b BITB eam, #b	<table border="1" style="margin-left: auto; margin-right: auto;"><tr><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>W</td><td>eam</td><td>b</td></tr><tr><td colspan="10">1, 2, or 3 extension words</td></tr></table>	0	1	1	0	0	1	1	W	eam	b	1, 2, or 3 extension words												
0	1	1	0	0	1	1	W	eam	b																
1, 2, or 3 extension words																									

Bit Test Static (Continued)

Addressing Mode	Assembler Language Syntax	Instruction Format						
	BITL eam, #b	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr><td>01111010</td><td>0000 0010</td></tr> <tr><td>01100111</td><td>b eam b</td></tr> <tr><td colspan="2">1, 2, or 3 extension words</td></tr> </table>	01111010	0000 0010	01100111	b eam b	1, 2, or 3 extension words	
01111010	0000 0010							
01100111	b eam b							
1, 2, or 3 extension words								

Bit Test Dynamic

Addressing Mode	Assembler Language Syntax	Instruction Format										
R:	BIT Rd, Rs BITB Rbd, Rs	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr><td>00</td><td>10011</td><td>W</td><td>0000</td><td>Rs</td></tr> <tr><td>0000</td><td>Rd</td><td>0000</td><td>0000</td><td></td></tr> </table>	00	10011	W	0000	Rs	0000	Rd	0000	0000	
00	10011	W	0000	Rs								
0000	Rd	0000	0000									
BITL RRd, Rs	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr><td>01111010</td><td>0000</td><td>0010</td></tr> <tr><td>00100111</td><td>0000</td><td>Rs</td></tr> <tr><td>0000</td><td>RRd</td><td>0000</td><td>0000</td></tr> </table>	01111010	0000	0010	00100111	0000	Rs	0000	RRd	0000	0000	
01111010	0000	0010										
00100111	0000	Rs										
0000	RRd	0000	0000									

Example: If register RH2 contains %B2 (10110010), executing the instruction
 BITB RH2, #0
 leaves the Z flag set to 1.

Note 1: Word register in compact mode, longword register in segmented or linear modes.

BRKPT

Breakpoint

BRKPT

Operation: SP \leftarrow SP - 6
@ SP \leftarrow PS
SP \leftarrow SP - 2
@ SP \leftarrow instruction
PS \leftarrow Breakpoint trap PS

This is a one word instruction that causes a Breakpoint trap. This instruction can be used by a software debugger to replace the first word of the instruction where a breakpoint is set.

Flags: Flags loaded from Program Status Area

Exceptions: Breakpoint trap

Addressing Mode	Assembler Language Syntax	Instruction Format		
	BRKPT	<table border="1"><tr><td>01111010</td><td>0000 0001</td></tr></table>	01111010	0000 0001
01111010	0000 0001			

CALL

Call

CALL dst	dst: IR, EAM
-----------------	--------------

Operation:	<i>Compact</i> tmp \leftarrow EFFECTIVE_ADDRESS (dst) SP \leftarrow SP - 2 @SP \leftarrow PC	<i>Segmented or Linear</i> tmp \leftarrow EFFECTIVE_ADDRESS (dst) SP \leftarrow SP - 4 @SP \leftarrow PC
-------------------	---	---

This instruction transfers control to a procedure or subroutine. The current contents of the Program Counter (PC) are pushed onto the top of the processor stack. The Stack Pointer (SP) pushed is R15 in compact mode, or RR14 in segmented or linear mode. (The PC value used is the address of the first instruction word following the CALL instruction.) The destination address, which points to the first instruction of the called procedure, is then loaded into the PC. At the end of the called procedure, a RET instruction can be used to return control to the instruction following CALL. RET pops the top of the processor stack back into the PC.

Flags:	No flags affected
---------------	-------------------

Exceptions:	None
--------------------	------

Destination Addressing Mode	Assembler Language Syntax	Instruction Format								
IR:	CALL @Rd ¹	<table border="1"><tr><td>00</td><td>011111</td><td>Rd≠0</td><td>0000</td></tr></table>	00	011111	Rd≠0	0000				
00	011111	Rd≠0	0000							
EAM:	CALL eam	<table border="1"><tr><td>01</td><td>011111</td><td>eam</td><td>0000</td></tr><tr><td colspan="4">1, 2, or 3 extension words</td></tr></table>	01	011111	eam	0000	1, 2, or 3 extension words			
01	011111	eam	0000							
1, 2, or 3 extension words										

Example: In compact mode, if the contents of the PC are %1000 and the contents of the Stack Pointer (R15) are %3002, executing the instruction

CALL %2520

causes the SP 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 PC to be loaded with the value %2520. The PC now points to the address of the first instruction in the procedure to be executed.

Note 1: Word register in compact mode, longword register in segmented or linear modes.

CALR

Call Relative

CALR dst	dst: RA								
Operation:	<table> <tr> <td><i>Compact</i></td><td><i>Segmented or Linear</i></td></tr> <tr> <td>SP \leftarrow SP - 2</td><td>SP \leftarrow SP - 4</td></tr> <tr> <td>@SP \leftarrow PC</td><td>@SP \leftarrow PC</td></tr> <tr> <td>PC \leftarrow PC - (2 \times displacement)</td><td>PC \leftarrow PC - (2 \times displacement)</td></tr> </table>	<i>Compact</i>	<i>Segmented or Linear</i>	SP \leftarrow SP - 2	SP \leftarrow SP - 4	@SP \leftarrow PC	@SP \leftarrow PC	PC \leftarrow PC - (2 \times displacement)	PC \leftarrow PC - (2 \times displacement)
<i>Compact</i>	<i>Segmented or Linear</i>								
SP \leftarrow SP - 2	SP \leftarrow SP - 4								
@SP \leftarrow PC	@SP \leftarrow PC								
PC \leftarrow PC - (2 \times displacement)	PC \leftarrow PC - (2 \times displacement)								
<p>The current contents of the Program Counter (PC) are pushed onto the top of the processor stack. The Stack Pointer (SP) used is R15 in compact mode, or RR14 in segmented or linear mode. (The PC value used is the address of the first instruction word following the CALR instruction.) The destination address, which points to the first instruction of the called procedure, is calculated and then loaded into the PC.</p> <p>At the end of the called procedure, a RET instruction can be used to return control of the instruction following CALR. RET pops the top of the processor stack back into the PC.</p> <p>The destination address is calculated by subtracting twice the displacement in the instruction from the current value of the PC. 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. The assembler automatically calculates the displacement by subtracting the address given by the programmer from the PC value of the following instruction and dividing the result by two.</p>									
Flags: No flags affected									
Exceptions: None									
Destination Addressing Mode	Assembler Language Syntax	Instruction Format							
	CALR address	<table border="1"> <tr> <td>1101</td> <td>displacement</td> </tr> </table>	1101	displacement					
1101	displacement								

Example: In linear mode, if the contents of the PC are %00001000 and the contents of the SP (RR14) are %FFFF3002, executing the instruction
CALR PROC
causes the SP to be decremented to %FFFF3000, the value %00001002 (the address following the CALR instruction) to be loaded into the longword location %FFFF3000, and the PC to be loaded with the address of the first instruction in procedure PROC.

CHK

Check

CHK dst, src CHKB CHKL	dst: R src: IM, IR, EAM
---	----------------------------

Operation:

```

tmp ← EFFECTIVE_ADDRESS (src)
lower ← @tmp
if dst < lower then Bounds Check trap
tmp ← tmp + (1 if CHKB; 2 if CHK; 4 if CHKL)
upper ← @tmp
if dst > upper then Bounds Check trap

```

The destination is compared against the bounds specified by the source operand. If the destination is less than the lower bound or greater than the upper bound, a Bounds Check trap occurs. The destination and bounds are compared as signed integers. The contents of the source and destination are not affected.

The source specifies the lower bound. The upper bound is located at the next consecutive byte, word, or longword.

Flags: No flags affected.

Exceptions: Bounds Check trap

Source Addressing Mode	Assembler Language Syntax	Instruction Format																
IM:	CHK Rd, #lower, #upper	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr><td>00</td><td>001101</td><td>0000</td><td>1010</td></tr> <tr><td>0000</td><td>Rd</td><td>0000</td><td>0000</td></tr> <tr><td colspan="4">lower</td></tr> <tr><td colspan="4">upper</td></tr> </table>	00	001101	0000	1010	0000	Rd	0000	0000	lower				upper			
00	001101	0000	1010															
0000	Rd	0000	0000															
lower																		
upper																		
CHKB Rbd, #lower, #upper	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr><td>00</td><td>001100</td><td>0000</td><td>1010</td></tr> <tr><td>0000</td><td>Rbd</td><td>0000</td><td>0000</td></tr> <tr><td colspan="2">lower</td><td colspan="2">upper</td></tr> </table>	00	001100	0000	1010	0000	Rbd	0000	0000	lower		upper						
00	001100	0000	1010															
0000	Rbd	0000	0000															
lower		upper																
CHKL RRd, #lower, #upper	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr><td>00</td><td>001101</td><td>0000</td><td>1011</td></tr> <tr><td>0000</td><td>RRd</td><td>0000</td><td>0000</td></tr> <tr><td colspan="2">lower (high)</td><td colspan="2">lower (low)</td></tr> <tr><td colspan="2">upper (high)</td><td colspan="2">upper (low)</td></tr> </table>	00	001101	0000	1011	0000	RRd	0000	0000	lower (high)		lower (low)		upper (high)		upper (low)		
00	001101	0000	1011															
0000	RRd	0000	0000															
lower (high)		lower (low)																
upper (high)		upper (low)																

Source Addressing Mode	Assembler Language Syntax	Instruction Format															
IR:	CHK Rd, @Rs ¹ CHKB Rbd, @Rs ¹	<table border="1"> <tr> <td>00</td><td>00110</td><td>W</td><td>Rs ≠ 0</td><td>1010</td></tr> <tr> <td>0000</td><td>Rd</td><td>0000</td><td>0000</td><td></td></tr> </table>	00	00110	W	Rs ≠ 0	1010	0000	Rd	0000	0000						
00	00110	W	Rs ≠ 0	1010													
0000	Rd	0000	0000														
CHKL RRD, @Rs ¹	<table border="1"> <tr> <td>00</td><td>001101</td><td>Rs ≠ 0</td><td>1011</td></tr> <tr> <td>0000</td><td>RRD</td><td>0000</td><td>0000</td><td></td></tr> </table>	00	001101	Rs ≠ 0	1011	0000	RRD	0000	0000								
00	001101	Rs ≠ 0	1011														
0000	RRD	0000	0000														
EAM:	CHK Rd, eam CHKB Rbd, eam	<table border="1"> <tr> <td>01</td><td>00110</td><td>W</td><td>eam</td><td>1010</td></tr> <tr> <td>0000</td><td>Rd</td><td>0000</td><td>0000</td><td></td></tr> <tr> <td colspan="5">1, 2, or 3 extension words</td></tr> </table>	01	00110	W	eam	1010	0000	Rd	0000	0000		1, 2, or 3 extension words				
01	00110	W	eam	1010													
0000	Rd	0000	0000														
1, 2, or 3 extension words																	
CHKL RRD, eam	<table border="1"> <tr> <td>01</td><td>001101</td><td>eam</td><td>1011</td></tr> <tr> <td>0000</td><td>RRD</td><td>0000</td><td>0000</td><td></td></tr> <tr> <td colspan="5">1, 2, or 3 extension words</td></tr> </table>	01	001101	eam	1011	0000	RRD	0000	0000		1, 2, or 3 extension words						
01	001101	eam	1011														
0000	RRD	0000	0000														
1, 2, or 3 extension words																	

Example: If RR2 contains 11, executing the instruction

CHKL RR2, #0, #10

causes a Bounds Check trap because the value in RR2 is greater than the upper bound of 10.

Note 1: Word register in compact mode, longword register in segmented or linear modes.

CLR

Clear

CLR dst	dst: R, IR, EAM
CLRB	
CLRL	

Operation: dst \leftarrow 0

The destination is cleared to 0.

Flags: No flags affected.

Exceptions: None

Destination Addressing Mode	Assembler Language Syntax	Instruction Format										
R:	CLR Rd CLRB Rbd	<table border="1" style="margin-left: auto; margin-right: auto;"><tr><td>10</td><td>00110</td><td>W</td><td>Rd</td><td>1000</td></tr></table>	10	00110	W	Rd	1000					
10	00110	W	Rd	1000								
CLRL RRd	<table border="1" style="margin-left: auto; margin-right: auto;"><tr><td>10</td><td>011100</td><td>RRd</td><td>0100</td></tr></table>	10	011100	RRd	0100							
10	011100	RRd	0100									
IR:	CLR @Rd ¹ CLRB @Rd ¹	<table border="1" style="margin-left: auto; margin-right: auto;"><tr><td>00</td><td>00110</td><td>W</td><td>Rd ≠ 0</td><td>1000</td></tr></table>	00	00110	W	Rd ≠ 0	1000					
00	00110	W	Rd ≠ 0	1000								
CLRL @Rd ¹	<table border="1" style="margin-left: auto; margin-right: auto;"><tr><td>00</td><td>011100</td><td>Rd ≠ 0</td><td>0100</td></tr></table>	00	011100	Rd ≠ 0	0100							
00	011100	Rd ≠ 0	0100									
EAM:	CLR eam CLRB eam	<table border="1" style="margin-left: auto; margin-right: auto;"><tr><td>01</td><td>00110</td><td>W</td><td>eam</td><td>1000</td></tr><tr><td colspan="5">1, 2, or 3 extension words</td></tr></table>	01	00110	W	eam	1000	1, 2, or 3 extension words				
01	00110	W	eam	1000								
1, 2, or 3 extension words												
CLRL eam	<table border="1" style="margin-left: auto; margin-right: auto;"><tr><td>01</td><td>011100</td><td>eam</td><td>0100</td></tr><tr><td colspan="4">1, 2, or 3 extension words</td></tr></table>	01	011100	eam	0100	1, 2, or 3 extension words						
01	011100	eam	0100									
1, 2, or 3 extension words												

Example: In linear mode, if the longword at location %0000ABBA contains 13, executing the instruction
 CLRL %ABBA
 leaves the value 0 in the longword at location %0000ABBA.

Note 1: Word register in compact mode, longword register in segmented or linear modes.

COM Complement

COM dst COMB COML	dst: R, IR, EAM
--	-----------------

Operation: $\text{dst} \leftarrow \text{NOT dst}$

The contents of the destination are complemented (ones complement); all 1 bits are changed to 0, 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, COML—unaffected; COMB—set if parity of the result is even; cleared otherwise D: Unaffected H: Unaffected
---------------	--

Exceptions: None

Destination Addressing Mode	Assembler Language Syntax	Instruction Format
R:	COM Rd COMB Rbd	10 00110 W Rd 0000
	COML RRd	10 011100 RRd 0000
IR:	COM @Rd ¹ COMB @Rd ¹	00 00110 W Rd ≠ 0 0000
	COML @Rd ¹	00 011100 Rd≠0 0000
EAM:	COM eam COMB eam	01 00110 W eam 0000 1, 2, or 3 extension words
	COML eam	01 011100 eam 0000 1, 2, or 3 extension words

Example: If register R1 contains %2552 (0010010101010010), executing the instruction
COM R1
leaves the value %DAAD (1101101010101101) in R1.

Note 1: Word register in compact mode, longword register in segmented or linear modes.

COMFLG

Complement Flag

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

Operation: Any combination of the C, Z, S, P or V flags can be complemented (each 1 bit is changed to 0, and vice-versa). If the bit in the instruction corresponding to a flag is 1, the flag is complemented; if the bit is 0, the flag is unchanged. All other bits in the Flags register are unaffected. Note that the P and V flags are represented by the same bit. There can 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:** Unaffected

Exceptions: None

	Assembler Language Syntax	Instruction Format			
	COMFLG flags	<table border="1"><tr><td>10001101</td><td>CZSP/V</td><td>0101</td></tr></table>	10001101	CZSP/V	0101
10001101	CZSP/V	0101			

Example: If the C, Z, and S flags are all clear (= 0), and the P flag is set (= 1), executing the instruction
COMFLG P, S, Z, C
leaves the C, Z, and S flags set , and the P flag clear.

CP

Compare

CP dst, src	dst: R
CPB	src: R, IM, IR, EAM
CPL	or dst: IR, EAM src: IM

Operation: dst – src

The source operand is compared to (subtracted from) the destination operand, and the flags are set accordingly. The flags can then be used for arithmetic and logical conditional jumps. Both operands are unaffected; the only action is 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 basic addressing modes; Compare Immediate performs a comparison between an operand in memory and an immediate value.

Flags:	C: Cleared if there is 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
---------------	--

Exceptions: None

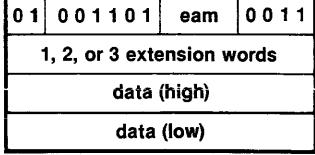
Compare Register

Source Addressing Mode	Assembler Language Syntax	Instruction Format											
R:	CP Rd, Rs CPB Rbd, Rbs	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>10</td><td>00101</td><td>W</td><td>Rs</td><td>Rd</td></tr> </table>	10	00101	W	Rs	Rd						
10	00101	W	Rs	Rd									
CPL RRD, RRS	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>10</td><td>010000</td><td>RRs</td><td>RRd</td></tr> </table>	10	010000	RRs	RRd								
10	010000	RRs	RRd										
IM:	CP Rd, #data	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>00</td><td>001011</td><td>0000</td><td>Rd</td></tr> <tr> <td colspan="4">data</td></tr> </table>	00	001011	0000	Rd	data						
00	001011	0000	Rd										
data													
CPB Rbd, #data	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>00</td><td>001010</td><td>0000</td><td>Rbd</td></tr> <tr> <td colspan="2">data</td><td colspan="2">data</td></tr> </table>	00	001010	0000	Rbd	data		data					
00	001010	0000	Rbd										
data		data											
CPL RRD, #data	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>00</td><td>010000</td><td>0000</td><td>RRd</td></tr> <tr> <td colspan="4">data (high)</td></tr> <tr> <td colspan="4">data (low)</td></tr> </table>	00	010000	0000	RRd	data (high)				data (low)			
00	010000	0000	RRd										
data (high)													
data (low)													

Source Addressing Mode	Assembler Language Syntax	Instruction Format									
IR:	CP Rd, @Rs ¹ CPB Rbd, @Rs ¹	<table border="1"><tr><td>00</td><td>00101</td><td>W</td><td>Rs ≠ 0</td><td>Rd</td></tr></table>	00	00101	W	Rs ≠ 0	Rd				
00	00101	W	Rs ≠ 0	Rd							
CPL RRd, @Rs ¹	<table border="1"><tr><td>00</td><td>01000</td><td>Rs ≠ 0</td><td>RRd</td></tr></table>	00	01000	Rs ≠ 0	RRd						
00	01000	Rs ≠ 0	RRd								
CP Rd, eam CPB Rbd, eam	<table border="1"><tr><td>01</td><td>00101</td><td>W</td><td>eam</td><td>Rd</td></tr><tr><td colspan="5">1, 2, or 3 extension words</td></tr></table>	01	00101	W	eam	Rd	1, 2, or 3 extension words				
01	00101	W	eam	Rd							
1, 2, or 3 extension words											
CPL RRd, eam	<table border="1"><tr><td>01</td><td>01000</td><td>eam</td><td>RRd</td></tr><tr><td colspan="4">1, 2, or 3 extension words</td></tr></table>	01	01000	eam	RRd	1, 2, or 3 extension words					
01	01000	eam	RRd								
1, 2, or 3 extension words											

Compare Immediate

Destination Addressing Mode	Assembler Language Syntax	Instruction Format											
IR:	CP @Rd ¹ , #data	<table border="1"><tr><td>00</td><td>001101</td><td>Rd ≠ 0</td><td>0001</td></tr><tr><td colspan="4">data</td></tr></table>	00	001101	Rd ≠ 0	0001	data						
00	001101	Rd ≠ 0	0001										
data													
CPB @Rd ¹ , #data	<table border="1"><tr><td>00</td><td>001100</td><td>Rd ≠ 0</td><td>0001</td></tr><tr><td>data</td><td></td><td>data</td><td></td></tr></table>	00	001100	Rd ≠ 0	0001	data		data					
00	001100	Rd ≠ 0	0001										
data		data											
CPL @Rd ¹ , #data	<table border="1"><tr><td>00</td><td>001101</td><td>Rd ≠ 0</td><td>0011</td></tr><tr><td colspan="4">data (high)</td></tr><tr><td colspan="4">data (low)</td></tr></table>	00	001101	Rd ≠ 0	0011	data (high)				data (low)			
00	001101	Rd ≠ 0	0011										
data (high)													
data (low)													
CP eam, #data	<table border="1"><tr><td>01</td><td>001101</td><td>eam</td><td>0001</td></tr><tr><td colspan="4">1, 2, or 3 extension words</td></tr><tr><td colspan="4">data</td></tr></table>	01	001101	eam	0001	1, 2, or 3 extension words				data			
01	001101	eam	0001										
1, 2, or 3 extension words													
data													
CPB eam, #data	<table border="1"><tr><td>01</td><td>001100</td><td>eam</td><td>0001</td></tr><tr><td colspan="4">1, 2, or 3 extension words</td></tr><tr><td>data</td><td></td><td>data</td><td></td></tr></table>	01	001100	eam	0001	1, 2, or 3 extension words				data		data	
01	001100	eam	0001										
1, 2, or 3 extension words													
data		data											

Destination Addressing Mode	Assembler Language Syntax	Instruction Format
	CPL eam, #data	 <pre> 01 001101 eam 0011 1, 2, or 3 extension words data (high) data (low) </pre>

Example: In linear mode, if register RR4 contains %00000400, the byte at location %00000400 contains 2, and the source operand is the immediate value 3, executing the instruction
 CPB @RR4,#3
 leaves the C flag set, indicating a borrow, the S flag set, and the Z and V flags cleared.

Note 1: Word register in compact mode, longword register in segmented or linear modes.

CPD

Compare and Decrement

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

Operation: dst – src
 AUTODECREMENT src (by 1 if CPDB; by 2 if CPD; by 4 if CPDL)
 $r \leftarrow r - 1$

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 to 1 if the condition code specified by "cc" is satisfied by the comparison; otherwise the Z flag is cleared to 0. See Section 6.3 for a list of condition codes. Both operands are unaffected.

The source register is then decremented by one if CPDB, by two if CPD, or by four if CPDL, 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 source, destination and count registers must be distinct and non-overlapping registers.

Flags: C: Cleared if there is a carry from the most-significant bit of the result of the comparison; set otherwise, indicating a borrow
 Z: Set if the condition code specified by cc is satisfied by the comparison; 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

Exceptions: None

Source Addressing Mode	Assembler Language Syntax	Instruction Format															
IR:	CPD Rd, @Rs ¹ , r, cc CPDB Rbd, @Rs ¹ , r, cc CPDL RRd, @Rs ¹ , r, cc	<table border="1" style="margin-bottom: 10px;"> <tr><td>1011101</td><td>W</td><td>Rs ≠ 0</td><td>1000</td></tr> <tr><td>0000</td><td>r</td><td>Rd</td><td>cc</td></tr> </table> <table border="1"> <tr><td>10111001</td><td>Rs ≠ 0</td><td>1000</td></tr> <tr><td>0000</td><td>r</td><td>RRd</td><td>cc</td></tr> </table>	1011101	W	Rs ≠ 0	1000	0000	r	Rd	cc	10111001	Rs ≠ 0	1000	0000	r	RRd	cc
1011101	W	Rs ≠ 0	1000														
0000	r	Rd	cc														
10111001	Rs ≠ 0	1000															
0000	r	RRd	cc														

Example: In linear mode, if register RH0 contains %FF, register RR4 contains %00004001, the byte at location %4001 contains %00, and register R3 contains 5, executing the instruction
 CPDB RH0, @RR4, R3, EQ
 leaves the Z flag cleared since the result of the comparison was not "equal."
 Register RR4 contains the value %00004000 and R3 contains 4. In compact mode, a word register must be used instead of RR4.

Note 1: Word register in compact mode, longword register in segmented or linear modes.

CPDR

Compare, Decrement and Repeat

	CPDR dst, src, r, cc CPDRB CPDRL	dst: R src: IR															
Operation:	<p>repeat dst – src AUTODECREMENT src (by 1 if CPDRB; by 2 if CPDR; by 4 if CPDRL) $r \leftarrow r - 1$ until cc is satisfied or $r = 0$</p>																
		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 to 1 if the condition code specified by "cc" is satisfied by the comparison; otherwise the Z flag is cleared to 0. See Section 6.3 for a list of condition codes. Both operands are unaffected.															
		The source register is then decremented by one if CPDRB, by two if CPDR, or by four if CPDRL, thus moving the pointer to the previous element in the string. The word register specified by "r" (used as a counter) is then decremented by one. The entire operation is repeated until either the condition is satisfied or the result of decrementing r is zero. This instruction can search a string of length 1 to 65,536 data elements. The source, destination, and counter registers must be distinct and non-overlapping registers.															
		This instruction can be interrupted after each execution of the basic operation.															
Flags:	C: Cleared if there is a carry from the most-significant bit of the result of the last comparison; set otherwise, indicating a borrow Z: Set if the condition code specified by cc is satisfied by the last comparison; cleared otherwise S: Set if the result of the last comparison is negative; cleared otherwise V: Set if the result of decrementing r is zero; cleared otherwise D: Unaffected H: Unaffected																
Exceptions:	None																
Source Addressing Mode	Assembler Language Syntax	Instruction Format															
IR:	CPDR Rd, @Rs1, r, cc CPDRB Rbd, @Rs1, r, cc CPDRL RRd, @Rs1, r, cc	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>1011101</td><td>W</td><td>Rs ≠ 0</td><td>1100</td></tr> <tr> <td>0000</td><td>r</td><td>Rd</td><td>cc</td></tr> </table> <table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>10111001</td><td>Rs ≠ 0</td><td>1100</td></tr> <tr> <td>0000</td><td>r</td><td>RRd</td><td>cc</td></tr> </table>	1011101	W	Rs ≠ 0	1100	0000	r	Rd	cc	10111001	Rs ≠ 0	1100	0000	r	RRd	cc
1011101	W	Rs ≠ 0	1100														
0000	r	Rd	cc														
10111001	Rs ≠ 0	1100															
0000	r	RRd	cc														

Example:	In compact mode, if the string of words starting at location %2000 contains the values 0, 2, 4, 6 and 8, register R2 contains %2008, R3 contains 5, and R8 contains 5, executing the instruction <code>CPDR R3, @R2, R8, GT</code> leaves the Z flag set, indicating the condition was satisfied. Register R2 contains the value %2002, R3 still contains 5, and R8 contains 2. In segmented or linear mode, a longword register must be used instead of R2.
-----------------	--

Note 1: Word register in compact mode, longword register in segmented or linear modes.

CPI

Compare and Increment

CPI dst, src, r, cc CPIB C PIL	dst: R src: IR																
Operation:	dst – src AUTOINCREMENT src (by 1 if CPIB; by 2 if CPI; by 4 if C PIL) $r \leftarrow r - 1$																
<p>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 to 1 if the condition code specified by "cc" is satisfied by the comparison; otherwise the Z flag is cleared to 0. See Section 6.3 for a list of condition codes. Both operands are unaffected.</p> <p>The source register is then incremented by one if CPIB, by two if CPI or by four if C PIL, 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 source, destination, and counter registers must be distinct and non-overlapping registers.</p>																	
Flags: <ul style="list-style-type: none"> C: Cleared if there is a carry from the most-significant bit of the result of the comparison; set otherwise, indicating a borrow Z: Set if the condition code specified by cc is satisfied by the comparison; 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 																	
Exceptions: None																	
Source Addressing Mode	Assembler Language Syntax	Instruction Format															
IR:	CPI Rd, @Rs ¹ , r, cc CPIB Rbd, @Rs ¹ , r, cc C PIL RRd, @Rs ¹ , r, cc	<table border="1" style="margin-left: auto; margin-right: auto; border-collapse: collapse; text-align: center;"> <tr> <td style="padding: 2px;">1011101</td><td style="padding: 2px;">W</td><td style="padding: 2px;">Rs ≠ 0</td><td style="padding: 2px;">0000</td></tr> <tr> <td style="padding: 2px;">0000</td><td style="padding: 2px;">r</td><td style="padding: 2px;">Rd</td><td style="padding: 2px;">cc</td></tr> </table> <table border="1" style="margin-left: auto; margin-right: auto; border-collapse: collapse; text-align: center;"> <tr> <td style="padding: 2px;">10111001</td><td style="padding: 2px;">Rs ≠ 0</td><td style="padding: 2px;">0000</td></tr> <tr> <td style="padding: 2px;">0000</td><td style="padding: 2px;">r</td><td style="padding: 2px;">RRd</td><td style="padding: 2px;">cc</td></tr> </table>	1011101	W	Rs ≠ 0	0000	0000	r	Rd	cc	10111001	Rs ≠ 0	0000	0000	r	RRd	cc
1011101	W	Rs ≠ 0	0000														
0000	r	Rd	cc														
10111001	Rs ≠ 0	0000															
0000	r	RRd	cc														

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. In compact mode, executing the following sequence of instructions “scans while numeric,” that is, a string is searched until either an ASCII character outside the range “0” to “9” is found, or the end of the string is reached. This involves a range check on each character (byte) in the string. In segmented or linear mode, a longword register must be used instead of R1.

```

LD      R3, #STRLEN           //initialize counter
LDA      R1, STRSTART         //load start address
LDB      RL0, #'9'            //largest numeric char

LOOP:
CPB      @R1, #'0'             //test char < '0'
JR      ULT, NONNUMERIC       //
CPIB     RL0, @R1, R3, ULE    //test char ≤ '9'
JR      NZ, NONNUMERIC        //
JR      NOV, LOOP              //repeat until counter = 0

DONE:
.

.

NONNUMERIC:                      //handle non-numeric char

```

Note 1: Word register in compact mode, longword register in segmented or linear modes.

CPIR

Compare, Increment and Repeat

	CPIR dst, src, r, cc CPIRB CPIRL	dst: R src: IR																																																				
Operation:	<pre>repeat dst - src AUTOINCREMENT src (by 1 if CPIRB; by 2 if CPIR; by 4 if CPIRL) r ← r - 1 until cc is satisfied or r = 0</pre>																																																					
<p>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 to 1 if the condition code specified by "cc" is satisfied by the comparison; otherwise the Z flag is cleared to 0. See Section 6.3 for a list of condition codes. Both operands are unaffected.</p> <p>The source register is then incremented by one if CPIRB, by two if CPIR, or by four if CPIRL, 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 satisfied or the result of decrementing r is zero. This instruction can search a string of length 1 to 65,536 data elements. The source, destination, and counter registers must be distinct and non-overlapping registers.</p> <p>This instruction can be interrupted after each execution of the basic operation.</p>																																																						
Flags: C: Cleared if there is a carry from the most-significant bit of the result of the last comparison; set otherwise, indicating a borrow Z: Set if the condition code specified by cc is satisfied by the last comparison; cleared otherwise S: Set if the result of the last comparison is negative; cleared otherwise V: Set if the result of decrementing r is zero; cleared otherwise D: Unaffected H: Unaffected																																																						
Exceptions:	None																																																					
Source Addressing Mode	Assembler Language Syntax	Instruction Format																																																				
IR:	CPIR Rd, @Rs ¹ , r, cc CPIRB Rbd, @Rs ¹ , r, cc	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>1</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>W</td><td>Rs ≠ 0</td><td>0</td><td>1</td><td>0</td><td>0</td> </tr> <tr> <td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>r</td><td>Rd</td><td>cc</td><td></td><td></td><td></td> </tr> </table> <table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>1</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>Rs ≠ 0</td><td>0</td><td>1</td><td>0</td><td>0</td> </tr> <tr> <td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>r</td><td>RRd</td><td>cc</td><td></td><td></td><td></td> </tr> </table>	1	0	1	1	1	0	1	W	Rs ≠ 0	0	1	0	0	0	0	0	0	0	0	0	r	Rd	cc				1	0	1	1	1	0	0	1	Rs ≠ 0	0	1	0	0	0	0	0	0	0	0	0	r	RRd	cc			
	1	0	1	1	1	0	1	W	Rs ≠ 0	0	1	0	0																																									
0	0	0	0	0	0	0	r	Rd	cc																																													
1	0	1	1	1	0	0	1	Rs ≠ 0	0	1	0	0																																										
0	0	0	0	0	0	0	r	RRd	cc																																													
CPIRL RRd, @Rs ¹ , r, cc																																																						

Example: The following sequence of instructions (to be executed in compact 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. In segmented or linear mode, a longword register must be used instead of R1.

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

Note 1: Word register in compact mode, longword register in segmented or linear modes.

CPSD

Compare String and Decrement

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

Operation: $dst - src$
AUTODECREMENT dst and src (by 1 if CPSDB; by 2 if CPSD; by 4 if CPSDL)
 $r \leftarrow r - 1$

This instruction is used to compare two strings of data in order to test the specified condition. 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 to 1 if the condition code specified by "cc" is satisfied by the comparison; otherwise the Z flag is cleared to 0. See Section 6.3 for a list of condition codes. Both operands are unaffected.

The source and destination registers are then decremented by one if CPSDB, by two if CPSD or by four if CPSDL, 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 source, destination and count register must be distinct, non-overlapping registers.

Flags:

- C:** Cleared if there is a carry from the most-significant bit of the result of the comparison; set otherwise, indicating a borrow.
- Z:** Set if the condition code specified by cc is satisfied by the comparison; 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

Exceptions: None

Addressing Mode	Assembler Language Syntax	Instruction Format								
IR:	CPSD @Rd ¹ , @Rs ¹ , r, cc CPSDB @Rd ¹ , @Rs ¹ , r, cc	<table border="1" style="margin: auto;"> <tr> <td style="padding: 2px;">1011101</td> <td style="padding: 2px; text-align: center;">W</td> <td style="padding: 2px; text-align: center;">Rs ≠ 0</td> <td style="padding: 2px;">1010</td> </tr> <tr> <td style="padding: 2px;">0000</td> <td style="padding: 2px; text-align: center;">r</td> <td style="padding: 2px; text-align: center;">Rd ≠ 0</td> <td style="padding: 2px;">cc</td> </tr> </table>	1011101	W	Rs ≠ 0	1010	0000	r	Rd ≠ 0	cc
1011101	W	Rs ≠ 0	1010							
0000	r	Rd ≠ 0	cc							
CPSDL @Rd ¹ , @Rs ¹ , r, cc	<table border="1" style="margin: auto;"> <tr> <td style="padding: 2px;">10111001</td> <td style="padding: 2px; text-align: center;">Rs ≠ 0</td> <td style="padding: 2px;">1010</td> </tr> <tr> <td style="padding: 2px;">0000</td> <td style="padding: 2px; text-align: center;">r</td> <td style="padding: 2px; text-align: center;">Rd ≠ 0</td> <td style="padding: 2px;">cc</td> </tr> </table>	10111001	Rs ≠ 0	1010	0000	r	Rd ≠ 0	cc		
10111001	Rs ≠ 0	1010								
0000	r	Rd ≠ 0	cc							

Example: In linear mode, if register RR24 contains %00002000, the byte at location %00002000 contains %FF, register RR26 contains %00003000, the byte at location %00003000 contains %00, and register R4 contains 1, executing the instruction

CPSDB @RR24, @RR26, R4, UGE

leaves the Z flag set to 1 since the result of the comparison was "unsigned greater than or equal", and the V flag set to 1 to indicate that the counter R4 now contains 0. RR24 contains %00001FFF, and RR26 contains %00002FFF. In compact mode, word registers must be used instead of RR24 and RR26.

Note 1: Word register in compact mode, longword register in segmented or linear modes.

CPSDR

Compare String, Decrement and Repeat

CPSDR dst, src,r, cc CPSDRB CPSDRL	dst: IR src: IR
---	--------------------

Operation:

```

repeat
    dst - src
    AUTODECREMENT dst and src (by 1 if CPSDRB; by 2 if CPSDR; by 4 if CPSDRL)
    r ← r - 1
until cc is satisfied 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 to 1 if the condition code specified by "cc" is satisfied by the comparison; otherwise the Z flag is cleared to 0. See Section 6.3 for a list of condition codes. Both operands are unaffected.

The source and destination registers are then decremented by one if CPSDRB, by two if CPSDR, or by four if CPSDRL, 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 satisfied or the result of decrementing r is zero. This instruction can compare strings of length 1 to 65,536 data elements. The source, destination, and counter registers must be distinct and non-overlapping registers.

This instruction can be interrupted after each execution of the basic operation.

Flags:

- C:** Cleared if there is a carry from the most-significant bit of the result of the last comparison; set otherwise, indicating a borrow.
- Z:** Set if the condition code specified by cc is satisfied by the last comparison; cleared otherwise
- S:** Set if the result of the last comparison is negative; cleared otherwise
- V:** Set if the result of decrementing r is zero; cleared otherwise
- D:** Unaffected
- H:** Unaffected

Exceptions: None

Addressing Mode	Assembler Language Syntax	Instruction Format																																																
IR:	CPSDR @Rd1, @Rs1, r,cc CPSDRB @Rd1, @Rs1, r, cc CPSDRL @Rd1, @Rs1, r, cc	<div style="margin-bottom: 20px;"> <table border="1" style="border-collapse: collapse; text-align: center;"> <tr> <td>1</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td> <td style="border: none;">W</td> <td style="border: none;">Rs≠0</td> <td style="border: none;">1</td> <td style="border: none;">1</td> <td style="border: none;">0</td> </tr> <tr> <td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td> <td>r</td> <td>Rd≠0</td> <td>cc</td> <td></td> <td></td> </tr> </table> </div> <div> <table border="1" style="border-collapse: collapse; text-align: center;"> <tr> <td>1</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td> <td style="border: none;">Rs≠0</td> <td style="border: none;">1</td> <td style="border: none;">1</td> <td style="border: none;">0</td> </tr> <tr> <td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td> <td>r</td> <td>Rd≠0</td> <td>cc</td> <td></td> <td></td> </tr> </table> </div>	1	0	1	1	1	0	1	W	Rs≠0	1	1	0	0	0	0	0	0	0	0	r	Rd≠0	cc			1	0	1	1	1	0	0	1	Rs≠0	1	1	0	0	0	0	0	0	0	0	r	Rd≠0	cc		
1	0	1	1	1	0	1	W	Rs≠0	1	1	0																																							
0	0	0	0	0	0	0	r	Rd≠0	cc																																									
1	0	1	1	1	0	0	1	Rs≠0	1	1	0																																							
0	0	0	0	0	0	0	r	Rd≠0	cc																																									

Example: In compact mode, 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, executing the instruction

CPSDR @R13, @R14, R0, EQ

leaves the Z flag set to 1 since the result of the comparison was “equal” (locations %1000 and %2000 both contain the value 0). The V flag is set to 1 indicating R0 was decremented to zero. R13 contains %OFFE, R14 contains %1FFE, and R0 contains 0. In segmented or linear mode, longword registers must be used instead of R13 and R14.

Note 1: Word register in compact mode, longword register in segmented or linear modes.

CPSI

Compare String and Increment

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

Operation: dst – src
 AUTOINCREMENT dst and src (by 1 if CPSIB; by 2 if CPSI; by 4 if CPSIL)
 r ← r – 1

This instruction is used to compare two strings of data, in order to test the specified condition. 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 to 1 if the condition code specified by "cc" is satisfied by the comparison; otherwise the Z flag is cleared to 0. See Section 6.3 for a list of condition codes. Both operands are unaffected.

The source and destination registers are then incremented by one if CPSIB, by two if CPSI or by four if CPSIL, 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 source, destination and count register must be distinct, non-overlapping registers.

Flags:

- C:** Cleared if there is a carry from the most-significant bit of the result of the comparison; set otherwise, indicating a borrow
- Z:** Set if the condition code specified by cc is satisfied by the comparison; 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

Exceptions: None

Addressing Mode	Assembler Language Syntax	Instruction Format														
IR:	CPSI @Rd1,@Rs1,r,cc CPSIB @Rd1,@Rs1,r, cc CPSIL @Rd1, @Rs1, r, cc	<table border="1" style="margin-left: auto; margin-right: auto; border-collapse: collapse;"> <tr> <td style="padding: 2px;">1011101</td> <td style="padding: 2px;">W</td> <td style="padding: 2px;">Rs ≠ 0</td> <td style="padding: 2px;">0010</td> </tr> <tr> <td style="padding: 2px;">0000</td> <td style="padding: 2px;">r</td> <td style="padding: 2px;">Rd ≠ 0</td> <td style="padding: 2px;">cc</td> </tr> </table> <table border="1" style="margin-left: auto; margin-right: auto; border-collapse: collapse;"> <tr> <td style="padding: 2px;">10111001</td> <td style="padding: 2px;">Rs ≠ 0</td> <td style="padding: 2px;">0010</td> </tr> <tr> <td style="padding: 2px;">0000</td> <td style="padding: 2px;">r</td> <td style="padding: 2px;">Rd ≠ 0</td> </tr> </table>	1011101	W	Rs ≠ 0	0010	0000	r	Rd ≠ 0	cc	10111001	Rs ≠ 0	0010	0000	r	Rd ≠ 0
1011101	W	Rs ≠ 0	0010													
0000	r	Rd ≠ 0	cc													
10111001	Rs ≠ 0	0010														
0000	r	Rd ≠ 0														

Example: This instruction can be used in a "loop" of instructions that 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 (executed in compact 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) 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:
.
.
.
NOTEQUAL: .           //match succeeds
```

In segmented or linear mode, longword registers must be used instead of R1 and R2.

Note 1: Word register in compact mode, longword register in segmented or linear modes.

CPSIR

Compare String, Increment and Repeat

CPSIR dst,src,r,cc CPSIRB CPSIRL	dst: IR src: IR
---	--------------------

Operation: repeat
 dst – src
 AUTOINCREMENT dst and src (by 1 if CPSIRB, by 2 if CPSIR; by 4 if CPSIRL)
 $r \leftarrow r - 1$
 until cc is satisfied 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 to 1 if the condition code specified by "cc" is satisfied by the comparison; otherwise the Z flag is cleared to 0. See Section 6.3 for a list of condition codes. Both operands are unaffected.

The source and destination registers are then incremented by one if CPSIRB, by two if CPSIR, or by four if CPSIRL, 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 satisfied or the result of decrementing r is zero. This instruction can compare strings of length 1 to 65,536 data elements. The source, destination, and counter registers must be distinct and non-overlapping registers.

This instruction can be interrupted after each execution of the basic operation.

Flags:	C: Cleared if there is a carry from the most-significant bit of the result of the last comparison; set otherwise, indicating a borrow. Z: Set if the condition code specified by cc is satisfied by the last comparison; cleared otherwise. S: Set if the result of the last comparison is negative; cleared otherwise V: Set if the result of decrementing r is zero; cleared otherwise D: Unaffected H: Unaffected
---------------	---

Exceptions: None

Addressing Mode	Assembler Language Syntax	Instruction Format															
IR:	CPSIR @Rd1,@Rs1,r,cc CPSIRB @Rd1,@Rs1,r,cc CPSIRL @Rd1,@Rs1,r,cc	<table border="1" style="margin-bottom: 10px;"> <tr> <td>1011101</td> <td>W</td> <td>Rs ≠ 0</td> <td>0110</td> </tr> <tr> <td>0000</td> <td>r</td> <td>Rd ≠ 0</td> <td>cc</td> </tr> </table> <table border="1"> <tr> <td>10111001</td> <td>Rs ≠ 0</td> <td>0110</td> </tr> <tr> <td>0000</td> <td>r</td> <td>Rd ≠ 0</td> <td>cc</td> </tr> </table>	1011101	W	Rs ≠ 0	0110	0000	r	Rd ≠ 0	cc	10111001	Rs ≠ 0	0110	0000	r	Rd ≠ 0	cc
1011101	W	Rs ≠ 0	0110														
0000	r	Rd ≠ 0	cc														
10111001	Rs ≠ 0	0110															
0000	r	Rd ≠ 0	cc														

Example:

The CPSIR instruction can be used to compare text strings for lexicographic order. (For most common character encodings — for example, ASCII and EBCDIC — lexicographic order is the same as alphabetic order for alphabetic text 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 C D
A B D

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 following sequence of instructions (executed in segmented or linear mode) 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:	

Note 1: Word register in compact mode, longword register in segmented or linear modes.

CVT Convert

CVTBW dst, src	dst: R
CVTBL	src: R, IR, EAM
CVTWB	or
CVTWL	dst: IR, EAM
CVTLB	src: R
CVTLW	

Operation: dst ← CONVERSION (src)

The contents of the source are converted to the size of the destination and then stored into the destination. The contents of the source are not affected.

The source and destination are treated as signed integers. The size of the destination operand is indicated by the fourth letter of the opcode mnemonic (B, W, or L); the size of the source operand is indicated by the last letter. For CVTWB, CVTLB, and CVTLW the source is sign-extended to the size of the destination before storing. For CVTBW, CVTBL, and CVTWL the source is truncated to the size of the destination, keeping the less-significant bits, before storing. If the source cannot be exactly represented in the destination because of truncation, then the V flag is set to 1; otherwise the V flag is cleared to 0.

Flags:	C: Cleared Z: Set if the result is zero; cleared otherwise S: Set if the most-significant bit of the result is set; cleared otherwise. V: CVTBW, CVTBL—set if the source is not in the range –128 to 127; cleared otherwise; CVTWL—set if the source is not in the range –32768 to 32767; cleared otherwise; CVTLB, CVTLW—cleared D: Unaffected H: Unaffected
---------------	--

Exceptions: Integer Overflow trap

Convert Register

Source Addressing Mode	Assembler Language Syntax	Instruction Format																								
R:	CVTBW Rbd, Rs	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td> </tr> <tr> <td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td></td><td>Rs</td><td></td><td>Rbd</td><td></td> </tr> </table>	0	1	1	1	0	0	0	0	0	0	0	1	1	0	1	0	0	0	1		Rs		Rbd	
0	1	1	1	0	0	0	0	0	0	0	1															
1	0	1	0	0	0	1		Rs		Rbd																
CVTBL Rbd, RR _s	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td> </tr> <tr> <td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>RR_s</td><td></td><td>Rbd</td><td></td><td></td> </tr> </table>	0	1	1	1	0	0	0	0	0	0	1	1	0	0	1	0	0	RR _s		Rbd					
0	1	1	1	0	0	0	0	0	0	1																
1	0	0	1	0	0	RR _s		Rbd																		
CVTWB Rd, Rbs	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td> </tr> <tr> <td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>Rbs</td><td></td><td>Rd</td><td></td><td></td> </tr> </table>	0	1	1	1	0	0	0	1	0	0	0	1	1	0	1	0	0	0	Rbs		Rd				
0	1	1	1	0	0	0	1	0	0	0	1															
1	0	1	0	0	0	Rbs		Rd																		
CVTWL Rd, RR _s	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td> </tr> <tr> <td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>RR_s</td><td></td><td>Rd</td><td></td><td></td> </tr> </table>	0	1	1	1	0	0	0	1	0	0	0	1	1	0	0	1	0	0	RR _s		Rd				
0	1	1	1	0	0	0	1	0	0	0	1															
1	0	0	1	0	0	RR _s		Rd																		

Source Addressing Mode	Assembler Language Syntax	Instruction Format						
IR:	CVTLB RRd, Rbs	<table border="1"> <tr><td>0 1 1 1 1 0 0 0</td><td>0 0 1 1 0 0 0 1</td></tr> <tr><td>1 0 1 0 0 0 0 0</td><td>Rbs RRd</td></tr> </table>	0 1 1 1 1 0 0 0	0 0 1 1 0 0 0 1	1 0 1 0 0 0 0 0	Rbs RRd		
0 1 1 1 1 0 0 0	0 0 1 1 0 0 0 1							
1 0 1 0 0 0 0 0	Rbs RRd							
CVTLW RRd, Rs	<table border="1"> <tr><td>0 1 1 1 1 0 0 0</td><td>0 0 1 1 0 0 0 1</td></tr> <tr><td>1 0 1 0 0 0 1 0</td><td>Rs RRd</td></tr> </table>	0 1 1 1 1 0 0 0	0 0 1 1 0 0 0 1	1 0 1 0 0 0 1 0	Rs RRd			
0 1 1 1 1 0 0 0	0 0 1 1 0 0 0 1							
1 0 1 0 0 0 1 0	Rs RRd							
CVTBW Rbd, @Rs ¹	<table border="1"> <tr><td>0 1 1 1 1 0 0 0</td><td>0 0 0 0 0 0 1 1</td></tr> <tr><td>0 0 1 0 0 0 1 0</td><td>Rs ≠ 0 Rbd</td></tr> </table>	0 1 1 1 1 0 0 0	0 0 0 0 0 0 1 1	0 0 1 0 0 0 1 0	Rs ≠ 0 Rbd			
0 1 1 1 1 0 0 0	0 0 0 0 0 0 1 1							
0 0 1 0 0 0 1 0	Rs ≠ 0 Rbd							
CVTBL Rbd, @Rs ¹	<table border="1"> <tr><td>0 1 1 1 1 0 0 0</td><td>0 0 0 0 0 0 1 1</td></tr> <tr><td>0 0 0 1 0 1 0 0</td><td>Rs ≠ 0 Rbd</td></tr> </table>	0 1 1 1 1 0 0 0	0 0 0 0 0 0 1 1	0 0 0 1 0 1 0 0	Rs ≠ 0 Rbd			
0 1 1 1 1 0 0 0	0 0 0 0 0 0 1 1							
0 0 0 1 0 1 0 0	Rs ≠ 0 Rbd							
CVTWB Rd, @Rs ¹	<table border="1"> <tr><td>0 1 1 1 1 0 0 0</td><td>0 0 1 0 0 0 1 1</td></tr> <tr><td>0 0 1 0 0 0 0 0</td><td>Rs ≠ 0 Rd</td></tr> </table>	0 1 1 1 1 0 0 0	0 0 1 0 0 0 1 1	0 0 1 0 0 0 0 0	Rs ≠ 0 Rd			
0 1 1 1 1 0 0 0	0 0 1 0 0 0 1 1							
0 0 1 0 0 0 0 0	Rs ≠ 0 Rd							
CVTWL Rd, @Rs ¹	<table border="1"> <tr><td>0 1 1 1 1 0 0 0</td><td>0 0 1 0 0 0 1 1</td></tr> <tr><td>0 0 0 1 0 1 0 0</td><td>Rs ≠ 0 Rd</td></tr> </table>	0 1 1 1 1 0 0 0	0 0 1 0 0 0 1 1	0 0 0 1 0 1 0 0	Rs ≠ 0 Rd			
0 1 1 1 1 0 0 0	0 0 1 0 0 0 1 1							
0 0 0 1 0 1 0 0	Rs ≠ 0 Rd							
CVTLB RRd, @Rs ¹	<table border="1"> <tr><td>0 1 1 1 1 0 0 0</td><td>0 0 1 1 0 0 0 1 1</td></tr> <tr><td>0 0 1 0 0 0 0 0</td><td>Rs ≠ 0 RRd</td></tr> </table>	0 1 1 1 1 0 0 0	0 0 1 1 0 0 0 1 1	0 0 1 0 0 0 0 0	Rs ≠ 0 RRd			
0 1 1 1 1 0 0 0	0 0 1 1 0 0 0 1 1							
0 0 1 0 0 0 0 0	Rs ≠ 0 RRd							
CVTLW RRd, @Rs ¹	<table border="1"> <tr><td>0 1 1 1 1 0 0 0</td><td>0 0 1 1 0 0 0 1 1</td></tr> <tr><td>0 0 1 0 0 0 1 0</td><td>Rs ≠ 0 RRd</td></tr> </table>	0 1 1 1 1 0 0 0	0 0 1 1 0 0 0 1 1	0 0 1 0 0 0 1 0	Rs ≠ 0 RRd			
0 1 1 1 1 0 0 0	0 0 1 1 0 0 0 1 1							
0 0 1 0 0 0 1 0	Rs ≠ 0 RRd							
CVTBW Rbd, eam	<table border="1"> <tr><td>0 1 1 1 1 0 0 0</td><td>0 0 0 0 0 0 1 1</td></tr> <tr><td>0 1 1 0 0 0 0 1</td><td>eam Rbd</td></tr> <tr><td colspan="2">1, 2, or 3 extension words</td></tr> </table>	0 1 1 1 1 0 0 0	0 0 0 0 0 0 1 1	0 1 1 0 0 0 0 1	eam Rbd	1, 2, or 3 extension words		
0 1 1 1 1 0 0 0	0 0 0 0 0 0 1 1							
0 1 1 0 0 0 0 1	eam Rbd							
1, 2, or 3 extension words								
CVTBL Rbd, eam	<table border="1"> <tr><td>0 1 1 1 1 0 0 0</td><td>0 0 0 0 0 0 1 1</td></tr> <tr><td>0 1 0 1 0 1 0 0</td><td>eam Rbd</td></tr> <tr><td colspan="2">1, 2, or 3 extension words</td></tr> </table>	0 1 1 1 1 0 0 0	0 0 0 0 0 0 1 1	0 1 0 1 0 1 0 0	eam Rbd	1, 2, or 3 extension words		
0 1 1 1 1 0 0 0	0 0 0 0 0 0 1 1							
0 1 0 1 0 1 0 0	eam Rbd							
1, 2, or 3 extension words								
EAM:	CVTWB Rd, eam	<table border="1"> <tr><td>0 1 1 1 1 0 0 0</td><td>0 0 1 0 0 0 1 1</td></tr> <tr><td>0 1 1 0 0 0 0 0</td><td>eam Rd</td></tr> <tr><td colspan="2">1, 2, or 3 extension words</td></tr> </table>	0 1 1 1 1 0 0 0	0 0 1 0 0 0 1 1	0 1 1 0 0 0 0 0	eam Rd	1, 2, or 3 extension words	
0 1 1 1 1 0 0 0	0 0 1 0 0 0 1 1							
0 1 1 0 0 0 0 0	eam Rd							
1, 2, or 3 extension words								

Source Addressing Mode	Assembler Language Syntax	Instruction Format										
	CVTWL Rd, eam	<table border="1"> <tr><td>0 1 1 1 1 0 0 0</td><td>0 0 1 0 0 0 1 1</td></tr> <tr><td>0 1</td><td>0 1 0 1 0 0</td><td>eam</td><td>Rd</td></tr> <tr><td colspan="4">1, 2, or 3 extension words</td></tr> </table>	0 1 1 1 1 0 0 0	0 0 1 0 0 0 1 1	0 1	0 1 0 1 0 0	eam	Rd	1, 2, or 3 extension words			
0 1 1 1 1 0 0 0	0 0 1 0 0 0 1 1											
0 1	0 1 0 1 0 0	eam	Rd									
1, 2, or 3 extension words												
	CVTLB RRs, eam	<table border="1"> <tr><td>0 1 1 1 1 0 0 0</td><td>0 0 1 1 0 0 1 1</td></tr> <tr><td>0 1</td><td>1 0 0 0 0 0</td><td>eam</td><td>RRs</td></tr> <tr><td colspan="4">1, 2, or 3 extension words</td></tr> </table>	0 1 1 1 1 0 0 0	0 0 1 1 0 0 1 1	0 1	1 0 0 0 0 0	eam	RRs	1, 2, or 3 extension words			
0 1 1 1 1 0 0 0	0 0 1 1 0 0 1 1											
0 1	1 0 0 0 0 0	eam	RRs									
1, 2, or 3 extension words												
	CVTLW RRs, eam	<table border="1"> <tr><td>0 1 1 1 1 0 0 0</td><td>0 0 1 1 0 0 1 1</td></tr> <tr><td>0 1</td><td>1 0 0 0 0 1</td><td>eam</td><td>RRs</td></tr> <tr><td colspan="4">1, 2, or 3 extension words</td></tr> </table>	0 1 1 1 1 0 0 0	0 0 1 1 0 0 1 1	0 1	1 0 0 0 0 1	eam	RRs	1, 2, or 3 extension words			
0 1 1 1 1 0 0 0	0 0 1 1 0 0 1 1											
0 1	1 0 0 0 0 1	eam	RRs									
1, 2, or 3 extension words												

Convert Memory

Destination Addressing Mode	Assembler Language Syntax	Instruction Format						
IR:	CVTBW @Rd1, Rs	<table border="1"> <tr><td>0 1 1 1 1 0 0 0</td><td>0 0 1 0 0 1 0 1</td></tr> <tr><td>0 0</td><td>1 0 1 1 1 0</td><td>Rd≠0</td><td>Rs</td></tr> </table>	0 1 1 1 1 0 0 0	0 0 1 0 0 1 0 1	0 0	1 0 1 1 1 0	Rd≠0	Rs
0 1 1 1 1 0 0 0	0 0 1 0 0 1 0 1							
0 0	1 0 1 1 1 0	Rd≠0	Rs					
	CVTBL @Rd1, RRs	<table border="1"> <tr><td>0 1 1 1 1 0 0 0</td><td>0 0 1 1 0 1 0 1</td></tr> <tr><td>0 0</td><td>1 0 1 1 1 0</td><td>Rd≠0</td><td>RRs</td></tr> </table>	0 1 1 1 1 0 0 0	0 0 1 1 0 1 0 1	0 0	1 0 1 1 1 0	Rd≠0	RRs
0 1 1 1 1 0 0 0	0 0 1 1 0 1 0 1							
0 0	1 0 1 1 1 0	Rd≠0	RRs					
	CVTWB @Rd1, Rbs	<table border="1"> <tr><td>0 1 1 1 1 0 0 0</td><td>0 0 0 0 0 1 0 1</td></tr> <tr><td>0 0</td><td>1 0 1 1 1 1</td><td>Rd≠0</td><td>Rbs</td></tr> </table>	0 1 1 1 1 0 0 0	0 0 0 0 0 1 0 1	0 0	1 0 1 1 1 1	Rd≠0	Rbs
0 1 1 1 1 0 0 0	0 0 0 0 0 1 0 1							
0 0	1 0 1 1 1 1	Rd≠0	Rbs					
	CVTWL @Rd1, RRs	<table border="1"> <tr><td>0 1 1 1 1 0 0 0</td><td>0 0 1 1 0 1 0 1</td></tr> <tr><td>0 0</td><td>1 0 1 1 1 1</td><td>Rd≠0</td><td>RRs</td></tr> </table>	0 1 1 1 1 0 0 0	0 0 1 1 0 1 0 1	0 0	1 0 1 1 1 1	Rd≠0	RRs
0 1 1 1 1 0 0 0	0 0 1 1 0 1 0 1							
0 0	1 0 1 1 1 1	Rd≠0	RRs					
	CVTLB @Rd1, Rbs	<table border="1"> <tr><td>0 1 1 1 1 0 0 0</td><td>0 0 0 0 0 1 0 1</td></tr> <tr><td>0 0</td><td>0 1 1 1 0 1</td><td>Rd≠0</td><td>Rbs</td></tr> </table>	0 1 1 1 1 0 0 0	0 0 0 0 0 1 0 1	0 0	0 1 1 1 0 1	Rd≠0	Rbs
0 1 1 1 1 0 0 0	0 0 0 0 0 1 0 1							
0 0	0 1 1 1 0 1	Rd≠0	Rbs					
	CVTLW @Rd1, Rs	<table border="1"> <tr><td>0 1 1 1 1 0 0 0</td><td>0 0 1 0 0 1 0 1</td></tr> <tr><td>0 0</td><td>0 1 1 1 0 1</td><td>Rd≠0</td><td>Rs</td></tr> </table>	0 1 1 1 1 0 0 0	0 0 1 0 0 1 0 1	0 0	0 1 1 1 0 1	Rd≠0	Rs
0 1 1 1 1 0 0 0	0 0 1 0 0 1 0 1							
0 0	0 1 1 1 0 1	Rd≠0	Rs					

Destination Addressing Mode	Assembler Language Syntax	Instruction Format				
EAM:	CVTBW eam, Rs	<table border="1" data-bbox="889 401 1199 475"> <tr> <td>0 1 1 1 1 0 0 0</td> <td>0 0 1 0 0 1 0 1</td> </tr> <tr> <td>0 1 1 0 1 1 1 0</td> <td>eam Rs</td> </tr> </table> <p data-bbox="922 485 1166 506">1, 2, or 3 extension words</p>	0 1 1 1 1 0 0 0	0 0 1 0 0 1 0 1	0 1 1 0 1 1 1 0	eam Rs
0 1 1 1 1 0 0 0	0 0 1 0 0 1 0 1					
0 1 1 0 1 1 1 0	eam Rs					
CVTBL eam, RR _S	<table border="1" data-bbox="889 559 1199 623"> <tr> <td>0 1 1 1 1 0 0 0</td> <td>0 0 1 1 0 1 0 1</td> </tr> <tr> <td>0 1 1 0 1 1 1 0</td> <td>eam RR_S</td> </tr> </table> <p data-bbox="922 644 1166 665">1, 2, or 3 extension words</p>	0 1 1 1 1 0 0 0	0 0 1 1 0 1 0 1	0 1 1 0 1 1 1 0	eam RR _S	
0 1 1 1 1 0 0 0	0 0 1 1 0 1 0 1					
0 1 1 0 1 1 1 0	eam RR _S					
CVTWB eam, Rbs	<table border="1" data-bbox="889 718 1199 781"> <tr> <td>0 1 1 1 1 0 0 0</td> <td>0 0 0 0 0 1 0 1</td> </tr> <tr> <td>0 1 1 0 1 1 1 1</td> <td>eam Rbs</td> </tr> </table> <p data-bbox="922 792 1166 813">1, 2, or 3 extension words</p>	0 1 1 1 1 0 0 0	0 0 0 0 0 1 0 1	0 1 1 0 1 1 1 1	eam Rbs	
0 1 1 1 1 0 0 0	0 0 0 0 0 1 0 1					
0 1 1 0 1 1 1 1	eam Rbs					
CVTWL eam, RR _S	<table border="1" data-bbox="889 876 1199 939"> <tr> <td>0 1 1 1 1 0 0 0</td> <td>0 0 1 1 0 1 0 1</td> </tr> <tr> <td>0 1 1 0 1 1 1 1</td> <td>eam RR_S</td> </tr> </table> <p data-bbox="922 960 1166 982">1, 2, or 3 extension words</p>	0 1 1 1 1 0 0 0	0 0 1 1 0 1 0 1	0 1 1 0 1 1 1 1	eam RR _S	
0 1 1 1 1 0 0 0	0 0 1 1 0 1 0 1					
0 1 1 0 1 1 1 1	eam RR _S					
CVTLB eam, Rbs	<table border="1" data-bbox="889 1034 1199 1098"> <tr> <td>0 1 1 1 1 0 0 0</td> <td>0 0 0 0 0 1 0 1</td> </tr> <tr> <td>0 1 0 1 1 1 0 1</td> <td>eam Rbs</td> </tr> </table> <p data-bbox="922 1108 1166 1129">1, 2, or 3 extension words</p>	0 1 1 1 1 0 0 0	0 0 0 0 0 1 0 1	0 1 0 1 1 1 0 1	eam Rbs	
0 1 1 1 1 0 0 0	0 0 0 0 0 1 0 1					
0 1 0 1 1 1 0 1	eam Rbs					
CVTLW eam, Rs	<table border="1" data-bbox="889 1193 1199 1256"> <tr> <td>0 1 1 1 1 0 0 0</td> <td>0 0 1 0 0 1 0 1</td> </tr> <tr> <td>0 1 0 1 1 1 0 1</td> <td>eam Rs</td> </tr> </table> <p data-bbox="922 1267 1166 1288">1, 2, or 3 extension words</p>	0 1 1 1 1 0 0 0	0 0 1 0 0 1 0 1	0 1 0 1 1 1 0 1	eam Rs	
0 1 1 1 1 0 0 0	0 0 1 0 0 1 0 1					
0 1 0 1 1 1 0 1	eam Rs					

Example: If byte register RH0 contains the value – 100, executing the instruction

CVTLB RR4, RH0

loads – 100 into longword register RR4. The S flag is set and the C, Z, and V flags are cleared.

Note 1: Word register in compact mode, longword register in segmented or linear modes.

CVTU Convert Unsigned

CVTUBW	dst, src	dst: R
CVTUBL		src: R, IR, EAM
CVTUWB		or
CVTUWL		dst: IR, EAM
CVTULB		src: R
CVTULW		

Operation: $\text{dst} \leftarrow \text{UNSIGNED_CONVERSION}(\text{src})$

The contents of the source are converted to the size of the destination and then stored into the destination. The contents of the source are not affected.

The source and destination are treated as unsigned integers. The size of the destination operand is indicated by the fifth letter of the opcode (B, W, or L); the size of the source operand is indicated by the last letter. For CVTUWB, CVTULB, and CVTULW the source is zero-extended to the size of the destination before storing. For CVTUBW, CVTUBL, and CVTUWL the source is truncated to the size of the destination, keeping the less significant bits, before storing. If the source cannot be exactly represented in the destination because of truncation then the V flag is set to 1; otherwise the V flag is cleared to 0.

Flags:	C: Cleared Z: Set if the result is zero; cleared otherwise S: Set if the most-significant bit of the result is set; cleared otherwise. V: CVTUBW, CVTUBL—set if the source is greater than 255; cleared otherwise CVTUWL—set if the source is greater than 65,535; cleared otherwise CVTULB,CVTULW—cleared D: Unaffected H: Unaffected
---------------	---

Exceptions: None

Convert Register Unsigned

Source Addressing Mode	Assembler Language Syntax	Instruction Format				
R:	CVTUBW Rbd, Rs	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>01111000</td><td>0000 0000</td></tr> <tr> <td>10 100001</td><td>Rs Rbd</td></tr> </table>	01111000	0000 0000	10 100001	Rs Rbd
01111000	0000 0000					
10 100001	Rs Rbd					
CVTUBL Rbd, RRs	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>01111000</td><td>0000 0000</td></tr> <tr> <td>10 010100</td><td>RRs Rbd</td></tr> </table>	01111000	0000 0000	10 010100	RRs Rbd	
01111000	0000 0000					
10 010100	RRs Rbd					
CVTUWB Rd, Rbs	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>01111000</td><td>0010 0000</td></tr> <tr> <td>10 100000</td><td>Rbs Rd</td></tr> </table>	01111000	0010 0000	10 100000	Rbs Rd	
01111000	0010 0000					
10 100000	Rbs Rd					
CVTUWL Rd, RRs	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>01111000</td><td>0010 0000</td></tr> <tr> <td>10 010100</td><td>RRs Rd</td></tr> </table>	01111000	0010 0000	10 010100	RRs Rd	
01111000	0010 0000					
10 010100	RRs Rd					

Source Addressing Mode	Assembler Language Syntax	Instruction Format												
IR:	CVTULB RRd, Rbs	<table border="1"> <tr><td>01111000</td><td>0011</td><td>0000</td></tr> <tr><td>10</td><td>100000</td><td>Rbs</td></tr> <tr><td>RRd</td><td></td><td></td></tr> </table>	01111000	0011	0000	10	100000	Rbs	RRd					
01111000	0011	0000												
10	100000	Rbs												
RRd														
CVTULW RRd, Rs	<table border="1"> <tr><td>01111000</td><td>0011</td><td>0000</td></tr> <tr><td>10</td><td>100001</td><td>Rs</td></tr> <tr><td>RRd</td><td></td><td></td></tr> </table>	01111000	0011	0000	10	100001	Rs	RRd						
01111000	0011	0000												
10	100001	Rs												
RRd														
CVTUBW Rbd, @Rs ¹	<table border="1"> <tr><td>01111000</td><td>0000</td><td>0010</td></tr> <tr><td>00</td><td>100001</td><td>Rs≠0</td></tr> <tr><td>Rbd</td><td></td><td></td></tr> </table>	01111000	0000	0010	00	100001	Rs≠0	Rbd						
01111000	0000	0010												
00	100001	Rs≠0												
Rbd														
CVTUBL Rbd, @Rs ¹	<table border="1"> <tr><td>01111000</td><td>0000</td><td>0010</td></tr> <tr><td>00</td><td>010100</td><td>Rs≠0</td></tr> <tr><td>Rbd</td><td></td><td></td></tr> </table>	01111000	0000	0010	00	010100	Rs≠0	Rbd						
01111000	0000	0010												
00	010100	Rs≠0												
Rbd														
CVTUWB Rd, @Rs ¹	<table border="1"> <tr><td>01111000</td><td>0010</td><td>0010</td></tr> <tr><td>00</td><td>100000</td><td>Rs≠0</td></tr> <tr><td>Rd</td><td></td><td></td></tr> </table>	01111000	0010	0010	00	100000	Rs≠0	Rd						
01111000	0010	0010												
00	100000	Rs≠0												
Rd														
CVTUWL Rd, @Rs ¹	<table border="1"> <tr><td>01111000</td><td>0010</td><td>0010</td></tr> <tr><td>00</td><td>010100</td><td>Rs≠0</td></tr> <tr><td>Rd</td><td></td><td></td></tr> </table>	01111000	0010	0010	00	010100	Rs≠0	Rd						
01111000	0010	0010												
00	010100	Rs≠0												
Rd														
CVTULB RRd, @Rs ¹	<table border="1"> <tr><td>01111000</td><td>0011</td><td>0010</td></tr> <tr><td>00</td><td>100000</td><td>Rs≠0</td></tr> <tr><td>RRd</td><td></td><td></td></tr> </table>	01111000	0011	0010	00	100000	Rs≠0	RRd						
01111000	0011	0010												
00	100000	Rs≠0												
RRd														
EAM:	CVTULW RRd, @Rs ¹	<table border="1"> <tr><td>01111000</td><td>0011</td><td>0010</td></tr> <tr><td>00</td><td>100001</td><td>Rs≠0</td></tr> <tr><td>RRd</td><td></td><td></td></tr> </table>	01111000	0011	0010	00	100001	Rs≠0	RRd					
01111000	0011	0010												
00	100001	Rs≠0												
RRd														
CVTUBW Rbd, eam	<table border="1"> <tr><td>01111000</td><td>0000</td><td>0010</td></tr> <tr><td>01</td><td>100001</td><td>eam</td></tr> <tr><td>Rbd</td><td></td><td></td></tr> <tr><td colspan="3">1, 2, or 3 extension words</td></tr> </table>	01111000	0000	0010	01	100001	eam	Rbd			1, 2, or 3 extension words			
01111000	0000	0010												
01	100001	eam												
Rbd														
1, 2, or 3 extension words														
CVTUBL Rbd, eam	<table border="1"> <tr><td>01111000</td><td>0000</td><td>0010</td></tr> <tr><td>01</td><td>010100</td><td>eam</td></tr> <tr><td>Rbd</td><td></td><td></td></tr> <tr><td colspan="3">1, 2, or 3 extension words</td></tr> </table>	01111000	0000	0010	01	010100	eam	Rbd			1, 2, or 3 extension words			
01111000	0000	0010												
01	010100	eam												
Rbd														
1, 2, or 3 extension words														
	CVTUWB Rd, eam	<table border="1"> <tr><td>01111000</td><td>0010</td><td>0010</td></tr> <tr><td>01</td><td>100000</td><td>eam</td></tr> <tr><td>Rd</td><td></td><td></td></tr> <tr><td colspan="3">1, 2, or 3 extension words</td></tr> </table>	01111000	0010	0010	01	100000	eam	Rd			1, 2, or 3 extension words		
01111000	0010	0010												
01	100000	eam												
Rd														
1, 2, or 3 extension words														

Source Addressing Mode	Assembler Language Syntax	Instruction Format										
	CVTUWL Rd, eam	<table border="1"> <tr><td>0 1 1 1 1 0 0 0</td><td>0 0 1 0 0 0 1 0</td></tr> <tr><td>0 1</td><td>0 1 0 1 0 0</td><td>eam</td><td>Rd</td></tr> <tr><td colspan="4">1, 2, or 3 extension words</td></tr> </table>	0 1 1 1 1 0 0 0	0 0 1 0 0 0 1 0	0 1	0 1 0 1 0 0	eam	Rd	1, 2, or 3 extension words			
0 1 1 1 1 0 0 0	0 0 1 0 0 0 1 0											
0 1	0 1 0 1 0 0	eam	Rd									
1, 2, or 3 extension words												
CVTULB RRd, eam	<table border="1"> <tr><td>0 1 1 1 1 0 0 0</td><td>0 0 1 1 0 0 1 0</td></tr> <tr><td>0 1</td><td>1 0 0 0 0 0</td><td>eam</td><td>RRd</td></tr> <tr><td colspan="4">1, 2, or 3 extension words</td></tr> </table>	0 1 1 1 1 0 0 0	0 0 1 1 0 0 1 0	0 1	1 0 0 0 0 0	eam	RRd	1, 2, or 3 extension words				
0 1 1 1 1 0 0 0	0 0 1 1 0 0 1 0											
0 1	1 0 0 0 0 0	eam	RRd									
1, 2, or 3 extension words												
CVTULW RRd, eam	<table border="1"> <tr><td>0 1 1 1 1 0 0 0</td><td>0 0 1 1 0 0 1 0</td></tr> <tr><td>0 1</td><td>1 0 0 0 0 1</td><td>eam</td><td>RRd</td></tr> <tr><td colspan="4">1, 2, or 3 extension words</td></tr> </table>	0 1 1 1 1 0 0 0	0 0 1 1 0 0 1 0	0 1	1 0 0 0 0 1	eam	RRd	1, 2, or 3 extension words				
0 1 1 1 1 0 0 0	0 0 1 1 0 0 1 0											
0 1	1 0 0 0 0 1	eam	RRd									
1, 2, or 3 extension words												

Convert Memory Unsigned

Destination Addressing Mode	Assembler Language Syntax	Instruction Format						
IR:	CVTUBW @Rd1, Rs	<table border="1"> <tr><td>0 1 1 1 1 0 0 0</td><td>0 0 1 0 0 1 0 0</td></tr> <tr><td>0 0</td><td>1 0 1 1 1 0</td><td>Rd ≠ 0</td><td>Rs</td></tr> </table>	0 1 1 1 1 0 0 0	0 0 1 0 0 1 0 0	0 0	1 0 1 1 1 0	Rd ≠ 0	Rs
0 1 1 1 1 0 0 0	0 0 1 0 0 1 0 0							
0 0	1 0 1 1 1 0	Rd ≠ 0	Rs					
CVTUBL @Rd1, RRs	<table border="1"> <tr><td>0 1 1 1 1 0 0 0</td><td>0 0 1 1 0 1 0 0</td></tr> <tr><td>0 0</td><td>1 0 1 1 1 0</td><td>Rd ≠ 0</td><td>RRs</td></tr> </table>	0 1 1 1 1 0 0 0	0 0 1 1 0 1 0 0	0 0	1 0 1 1 1 0	Rd ≠ 0	RRs	
0 1 1 1 1 0 0 0	0 0 1 1 0 1 0 0							
0 0	1 0 1 1 1 0	Rd ≠ 0	RRs					
CVTUWB @Rd1, Rbs	<table border="1"> <tr><td>0 1 1 1 1 0 0 0</td><td>0 0 0 0 0 1 0 0</td></tr> <tr><td>0 0</td><td>1 0 1 1 1 1</td><td>Rd ≠ 0</td><td>Rbs</td></tr> </table>	0 1 1 1 1 0 0 0	0 0 0 0 0 1 0 0	0 0	1 0 1 1 1 1	Rd ≠ 0	Rbs	
0 1 1 1 1 0 0 0	0 0 0 0 0 1 0 0							
0 0	1 0 1 1 1 1	Rd ≠ 0	Rbs					
CVTUWL @Rd1, RRs	<table border="1"> <tr><td>0 1 1 1 1 0 0 0</td><td>0 0 1 1 0 1 0 0</td></tr> <tr><td>0 0</td><td>1 0 1 1 1 1</td><td>Rd ≠ 0</td><td>RRs</td></tr> </table>	0 1 1 1 1 0 0 0	0 0 1 1 0 1 0 0	0 0	1 0 1 1 1 1	Rd ≠ 0	RRs	
0 1 1 1 1 0 0 0	0 0 1 1 0 1 0 0							
0 0	1 0 1 1 1 1	Rd ≠ 0	RRs					
CVTULB @Rd1, Rbs	<table border="1"> <tr><td>0 1 1 1 1 0 0 0</td><td>0 0 0 0 0 1 0 0</td></tr> <tr><td>0 0</td><td>0 1 1 1 0 1</td><td>Rd ≠ 0</td><td>Rbs</td></tr> </table>	0 1 1 1 1 0 0 0	0 0 0 0 0 1 0 0	0 0	0 1 1 1 0 1	Rd ≠ 0	Rbs	
0 1 1 1 1 0 0 0	0 0 0 0 0 1 0 0							
0 0	0 1 1 1 0 1	Rd ≠ 0	Rbs					
CVTULW @Rd1, Rs	<table border="1"> <tr><td>0 1 1 1 1 0 0 0</td><td>0 0 1 0 1 1 0 0</td></tr> <tr><td>0 0</td><td>0 1 1 1 0 1</td><td>Rd ≠ 0</td><td>Rs</td></tr> </table>	0 1 1 1 1 0 0 0	0 0 1 0 1 1 0 0	0 0	0 1 1 1 0 1	Rd ≠ 0	Rs	
0 1 1 1 1 0 0 0	0 0 1 0 1 1 0 0							
0 0	0 1 1 1 0 1	Rd ≠ 0	Rs					

Destination Addressing Mode	Assembler Language Syntax	Instruction Format										
EAM:	CVTUBW eam, Rs	<table border="1" data-bbox="882 403 1192 523"> <tr><td>0 1 1 1 1 0 0 0</td><td>0 0 1 0 0 1 0 0</td></tr> <tr><td>0 1</td><td>1 0 1 1 1 0</td><td>eam</td><td>Rs</td></tr> <tr><td colspan="4">1, 2, or 3 extension words</td></tr> </table>	0 1 1 1 1 0 0 0	0 0 1 0 0 1 0 0	0 1	1 0 1 1 1 0	eam	Rs	1, 2, or 3 extension words			
0 1 1 1 1 0 0 0	0 0 1 0 0 1 0 0											
0 1	1 0 1 1 1 0	eam	Rs									
1, 2, or 3 extension words												
CVTUBL eam, RRs	<table border="1" data-bbox="882 551 1192 671"> <tr><td>0 1 1 1 1 0 0 0</td><td>0 0 1 1 0 1 0 0</td></tr> <tr><td>0 1</td><td>1 0 1 1 1 0</td><td>eam</td><td>RRs</td></tr> <tr><td colspan="4">1, 2, or 3 extension words</td></tr> </table>	0 1 1 1 1 0 0 0	0 0 1 1 0 1 0 0	0 1	1 0 1 1 1 0	eam	RRs	1, 2, or 3 extension words				
0 1 1 1 1 0 0 0	0 0 1 1 0 1 0 0											
0 1	1 0 1 1 1 0	eam	RRs									
1, 2, or 3 extension words												
CVTUWB eam, Rbs	<table border="1" data-bbox="882 699 1192 819"> <tr><td>0 1 1 1 1 0 . 0 0</td><td>0 0 0 0 0 1 0 0</td></tr> <tr><td>0 1</td><td>1 0 1 1 1 1</td><td>eam</td><td>Rbs</td></tr> <tr><td colspan="4">1, 2, or 3 extension words</td></tr> </table>	0 1 1 1 1 0 . 0 0	0 0 0 0 0 1 0 0	0 1	1 0 1 1 1 1	eam	Rbs	1, 2, or 3 extension words				
0 1 1 1 1 0 . 0 0	0 0 0 0 0 1 0 0											
0 1	1 0 1 1 1 1	eam	Rbs									
1, 2, or 3 extension words												
CVTUWL eam, RRs	<table border="1" data-bbox="882 846 1192 967"> <tr><td>0 1 1 1 1 0 0 0</td><td>0 0 1 1 0 1 0 0</td></tr> <tr><td>0 1</td><td>1 0 1 1 1 1</td><td>eam</td><td>RRs</td></tr> <tr><td colspan="4">1, 2, or 3 extension words</td></tr> </table>	0 1 1 1 1 0 0 0	0 0 1 1 0 1 0 0	0 1	1 0 1 1 1 1	eam	RRs	1, 2, or 3 extension words				
0 1 1 1 1 0 0 0	0 0 1 1 0 1 0 0											
0 1	1 0 1 1 1 1	eam	RRs									
1, 2, or 3 extension words												
CVTULB eam, Rbs	<table border="1" data-bbox="882 994 1192 1115"> <tr><td>0 1 1 1 1 0 0 0</td><td>0 0 0 0 0 1 0 0</td></tr> <tr><td>0 1</td><td>0 1 1 1 0 1</td><td>eam</td><td>Rbs</td></tr> <tr><td colspan="4">1, 2, or 3 extension words</td></tr> </table>	0 1 1 1 1 0 0 0	0 0 0 0 0 1 0 0	0 1	0 1 1 1 0 1	eam	Rbs	1, 2, or 3 extension words				
0 1 1 1 1 0 0 0	0 0 0 0 0 1 0 0											
0 1	0 1 1 1 0 1	eam	Rbs									
1, 2, or 3 extension words												
CVTULW eam, Rs	<table border="1" data-bbox="882 1142 1192 1284"> <tr><td>0 1 1 1 1 0 0 0</td><td>0 0 1 0 0 1 0 0</td></tr> <tr><td>0 1</td><td>0 1 1 1 0 1</td><td>eam</td><td>Rs</td></tr> <tr><td colspan="4">1, 2, or 3 extension words</td></tr> </table>	0 1 1 1 1 0 0 0	0 0 1 0 0 1 0 0	0 1	0 1 1 1 0 1	eam	Rs	1, 2, or 3 extension words				
0 1 1 1 1 0 0 0	0 0 1 0 0 1 0 0											
0 1	0 1 1 1 0 1	eam	Rs									
1, 2, or 3 extension words												

Example: If word register R1 contains the value %0F12, executing the instruction

CVTUBW RL0, R1

loads %12 into byte register RL0. The V flag is set and the C, Z, and S flags are cleared.

Note 1: Word register in compact mode, longword register in segmented or linear modes.

DAB

Decimal Adjust

	DAB dst	dst: R															
Operation:	$\text{dst} \leftarrow \text{DECIMAL_ADJUST}(\text{dst})$																
The destination byte is adjusted to form two 4-bit BCD digits following a binary addition or subtraction operation on two BCD encoded bytes. Following addition (ADDB, ADCB) or subtraction (SUBB, SBCB), the table below indicates the operation performed:																	
<hr/>																	
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											
<hr/>																	
ADDB	0	0-9	0	0-9	00	0											
	0	0-8	0	A-F	06	0											
	0	0-9	1	0-3	06	0											
	0	A-F	0	0-9	60	1											
	0	9-F	0	A-F	66	1											
	0	A-F	1	0-3	66	1											
	1	0-2	0	0-9	60	1											
	1	0-2	0	A-F	66	1											
<hr/>																	
SUBB	0	0-9	0	0-9	00	0											
	0	0-8	1	6-F	FA	0											
	1	7-F	0	0-9	A0	1											
	1	6-F	1	6-F	9A	1											
<hr/>																	
The operation is undefined if the destination byte was not the result of a binary addition or subtraction of BCD digits.																	
<hr/>																	
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																
<hr/>																	
Exceptions:	None																
<hr/>																	
Addressing Mode	Assembler Language Syntax			Instruction Format													
<hr/>																	
R:	DAB Rbd			<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>Rbd</td><td>0</td><td>0</td><td>0</td> </tr> </table>			1	0	1	1	0	0	0	Rbd	0	0	0
1	0	1	1	0	0	0	Rbd	0	0	0							
<hr/>																	

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. As shown below, adding the two numbers using binary arithmetic gives a result of %3C, leaving the C and H flags clear.

$$\begin{array}{r} 0001 \quad 0101 \\ + \quad 0010 \quad 0111 \\ \hline 0011 \quad 1100 = \%3C \end{array}$$

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

$$\begin{array}{r} 0011 \quad 1100 \\ + \quad 0000 \quad 0110 \\ \hline 0100 \quad 0010 = 42 \end{array}$$

DEC

Decrement

DEC dst, src	dst: R, IR, EAM src: IM
DECB	
DECL	

Operation: $dst \leftarrow dst - src$ ($src = 1$ 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. If the source operand is omitted from the assembler language statement, the default value is 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 sign and the sign of the result is the same as the sign of the source; cleared otherwise.
	D: Unaffected
	H: Unaffected

Exceptions: Integer Overflow trap

Destination Addressing Mode	Assembler Language Syntax	Instruction Format										
R:	DEC Rd, #n DECB Rbd, #n	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr><td>10</td><td>10101</td><td>W</td><td>Rd</td><td>n - 1</td></tr> </table>	10	10101	W	Rd	n - 1					
10	10101	W	Rd	n - 1								
DECL RRd, #n	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr><td>01111010</td><td>0000 0010</td></tr> <tr><td>10</td><td>101011</td><td>RRd</td><td>n - 1</td></tr> </table>	01111010	0000 0010	10	101011	RRd	n - 1					
01111010	0000 0010											
10	101011	RRd	n - 1									
IR:	DEC @Rd ¹ , #n DECB @Rd ¹ , #n	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr><td>00</td><td>10101</td><td>W</td><td>Rd ≠ 0</td><td>n - 1</td></tr> </table>	00	10101	W	Rd ≠ 0	n - 1					
00	10101	W	Rd ≠ 0	n - 1								
DECL @Rd ¹ , #n	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr><td>01111010</td><td>0000 0010</td></tr> <tr><td>00</td><td>101011</td><td>Rd ≠ 0</td><td>n - 1</td></tr> </table>	01111010	0000 0010	00	101011	Rd ≠ 0	n - 1					
01111010	0000 0010											
00	101011	Rd ≠ 0	n - 1									
EAM:	DEC eam, #n DECB eam, #n	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr><td>01</td><td>10101</td><td>W</td><td>eam</td><td>n - 1</td></tr> <tr><td colspan="5">1, 2, or 3 extension words</td></tr> </table>	01	10101	W	eam	n - 1	1, 2, or 3 extension words				
01	10101	W	eam	n - 1								
1, 2, or 3 extension words												
DECL eam, #n	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr><td>01111010</td><td>0000 0010</td></tr> <tr><td>01</td><td>101011</td><td>eam</td><td>n - 1</td></tr> <tr><td colspan="5">1, 2, or 3 extension words</td></tr> </table>	01111010	0000 0010	01	101011	eam	n - 1	1, 2, or 3 extension words				
01111010	0000 0010											
01	101011	eam	n - 1									
1, 2, or 3 extension words												

Example: If register RR10 contains %0000002A, executing the instruction
 DECL RR10
leaves the value %00000029 in RR10.

Note 1: Word register in compact mode, longword register in segmented or linear modes.

DECI

Decrement Interlocked

DEC1 dst, src
DECIB

dst: IR, EAM
src: IM

Operation: $dst \leftarrow dst - src$ ($src = 1$ 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. If the source operand is omitted from the assembly language statement, the default value is 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.

This is an interlocked instruction. No other interlocked accesses are permitted to the destination memory location between fetching and storing the result.

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 sign and the sign of the result is the same as the sign of the source; cleared otherwise. D: Unaffected H: Unaffected
---------------	--

Exceptions: Integer Overflow trap

Destination Addressing Mode	Assembler Language Syntax	Instruction Format						
IR:	DEC1 @Rd1, #n DECIB @Rd1, #n	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>0 1 1 1 0 1 0</td> <td>0 0 0 0 0 1 0 0</td> </tr> <tr> <td>0 0 1 0 1 0 1 W</td> <td>Rd ≠ 0 n - 1</td> </tr> </table>	0 1 1 1 0 1 0	0 0 0 0 0 1 0 0	0 0 1 0 1 0 1 W	Rd ≠ 0 n - 1		
0 1 1 1 0 1 0	0 0 0 0 0 1 0 0							
0 0 1 0 1 0 1 W	Rd ≠ 0 n - 1							
EAM:	DEC1 eam, #n DECIB eam, #n	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>0 1 1 1 0 1 0</td> <td>0 0 0 0 0 1 0 0</td> </tr> <tr> <td>0 1 1 0 1 0 1 W</td> <td>eam n - 1</td> </tr> <tr> <td colspan="2">1, 2, or 3 extension words</td></tr> </table>	0 1 1 1 0 1 0	0 0 0 0 0 1 0 0	0 1 1 0 1 0 1 W	eam n - 1	1, 2, or 3 extension words	
0 1 1 1 0 1 0	0 0 0 0 0 1 0 0							
0 1 1 0 1 0 1 W	eam n - 1							
1, 2, or 3 extension words								

Example: This instruction can be used to allocate or release copies of a system resource in a multiprocessor environment. For example, several processes running on different processors can share use of a common page in memory. It is necessary to keep a reference counter for the number of active processes using the shared page. When one of these processes terminates, the reference counter is decremented. The DECI instruction should be used so that one processor completes the fetch and store of the counter in memory before any other processor accesses the counter.

```
DEC1 REFERENCE_COUNTER, #1           //decrement reference counter
                                         for shared page
```

Note 1: Word register in compact mode, longword register in segmented or linear modes.

DI **Disable Interrupt**

Privileged Instruction

DI Int	Int: VI, NVI
---------------	---------------------

Operation: If instruction<0> = 0 then NVI \leftarrow 0
If instruction<1> = 0 then VI \leftarrow 0

Any combination of the Vectored Interrupt (VI) or Non-Vectored Interrupt (NVI) control bits in the Flag and Control Word (FCW) are cleared to 0 if the corresponding bit in the instruction is 0, thus disabling the appropriate type of interrupt. If the corresponding bit in the instruction is 1, the control bit is not affected. All other bits in the FCW are not affected. There may be zero, one or two operands in the assembly language statement, in either order, specifying no source operand is equivalent to specifying both VI and NVI.

Flags: No flags affected.

Exceptions: Privileged Instruction trap

	Assembler Language Syntax	Instruction Format								
	DI int	<table><tr><td>01111100</td><td>000000</td><td>V</td><td>N</td></tr><tr><td></td><td></td><td>/</td><td>/</td></tr></table>	01111100	000000	V	N			/	/
01111100	000000	V	N							
		/	/							

Example: If the NVI and VI control bits are set (1) in the FCW, executing the instruction
DI VI
leaves the NVI control bit in the FCW set to 1 and the VI control bit in the FCW cleared to 0.

DIV

Divide

DIV dst, src	dst: R
DIVL	src: R, IM, IR, EAM

Operation:

Word: (dst is longword register, src is word):
 $\text{dst} < 31:0 >$ is divided by $\text{src} < 15:0 >$
 $(\text{dst} < 31:0 > = \text{quotient} \times \text{src} < 15:0 > + \text{remainder})$
 $\text{dst} < 15:0 > \leftarrow \text{quotient}$
 $\text{dst} < 31:16 > \leftarrow \text{remainder}$

Longword: (dst is quadword register, src is longword):
 $\text{dst} < 63:0 >$ is divided by $\text{src} < 31:0 >$
 $(\text{dst} < 63:0 > = \text{quotient} \times \text{src} < 31:0 > + \text{remainder})$
 $\text{dst} < 31:0 > \leftarrow \text{quotient}$
 $\text{dst} < 63:32 > \leftarrow \text{remainder}$

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. Division is performed so that the remainder is of the same sign as the dividend except when the remainder is 0 and the quotient sign is the exclusive OR of the signs of the dividend and divisor except when the quotient is 0. For DIV, the destination is a longword register and the source is a word value; for DIVL, the destination is a quadword register and the source is a longword value.

For proper instruction execution the "dst field" in the DIVL instruction encoding must specify a valid code for a quadword register.

There are four possible outcomes of the signed divide instruction.

CASE 1. If the divisor is 0, then the destination register is unmodified, the V and Z flags are set to 1, and the C and S flags are cleared to 0.

CASE 2. If the quotient is less than $-(2^{16} - 1)$ or greater than $(2^{16} - 1)$ for DIV or if the quotient is less than $-(2^{32} - 1)$ or greater than $(2^{32} - 1)$ for DIVL, then the destination register is unmodified. The V flag is set to 1, and the C, Z, and S flags are cleared to 0.

CASE 3. If the quotient is greater than $-(2^{15} + 1)$ and less than (2^{15}) for DIV or if the quotient is greater than $-(2^{31} + 1)$ and less than (2^{31}) for DIVL, then the quotient and remainder are left in the destination register as defined above. The V and C flags are cleared to 0 and the S and Z flags are set according to the value of the quotient.

CASE 4. If none of the above cases applies, then all of the remainder and all but the Sign bit of the quotient are left in the destination register. The V and C flags are set to 1, the Z flag is cleared to 0, and the S flag indicates the sign of the quotient. In this case, the S flag can be replicated into the high-order half of the destination to produce the two's complement representation of the quotient with the same precision as the original dividend.

Flags:

- C:** For CASE 4 set; cleared otherwise
- Z:** Set if the quotient or divisor is zero; cleared otherwise
- S:** For CASE 1 and CASE 2 cleared; for CASE 3 and CASE 4 set if the quotient is negative; cleared otherwise
- V:** For CASE 3 cleared; set otherwise
- D:** Unaffected
- H:** Unaffected

Exceptions: Integer Overflow trap

Source Addressing Mode	Assembler Language Syntax	Instruction Format											
R:	DIV RRd, Rs	<table border="1"><tr><td>10</td><td>011011</td><td>Rs</td><td>RRd</td></tr></table>	10	011011	Rs	RRd							
10	011011	Rs	RRd										
DIVL RQd, RRs	<table border="1"><tr><td>10</td><td>011010</td><td>RRs</td><td>RQd</td></tr></table>	10	011010	RRs	RQd								
10	011010	RRs	RQd										
DIV RRd, #data	<table border="1"><tr><td>00</td><td>011011</td><td>0000</td><td>RRd</td></tr><tr><td colspan="4">data</td></tr></table>	00	011011	0000	RRd	data							
00	011011	0000	RRd										
data													
DIVL RQd, #data	<table border="1"><tr><td>00</td><td>011010</td><td>0000</td><td>RQd</td></tr><tr><td colspan="4">data (high)</td></tr><tr><td colspan="4">data (low)</td></tr></table>	00	011010	0000	RQd	data (high)				data (low)			
00	011010	0000	RQd										
data (high)													
data (low)													
IR:	DIV RRd, @Rs ¹	<table border="1"><tr><td>00</td><td>011011</td><td>Rs≠0</td><td>RRd</td></tr></table>	00	011011	Rs≠0	RRd							
00	011011	Rs≠0	RRd										
DIVL RQd, @Rs ¹	<table border="1"><tr><td>00</td><td>011010</td><td>Rs≠0</td><td>RQd</td></tr></table>	00	011010	Rs≠0	RQd								
00	011010	Rs≠0	RQd										
EAM:	DiV RRd, eam	<table border="1"><tr><td>01</td><td>011011</td><td>eam</td><td>RRd</td></tr><tr><td colspan="4">1, 2, or 3 extension words</td></tr></table>	01	011011	eam	RRd	1, 2, or 3 extension words						
01	011011	eam	RRd										
1, 2, or 3 extension words													
DIVL RQd, eam	<table border="1"><tr><td>01</td><td>011010</td><td>eam</td><td>RQd</td></tr><tr><td colspan="4">1, 2, or 3 extension words</td></tr></table>	01	011010	eam	RQd	1, 2, or 3 extension words							
01	011010	eam	RQd										
1, 2, or 3 extension words													

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

DIV RR0,R3

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

Note 1: Word register in compact mode, longword register in segmented or linear modes.

DIVU

Divide Unsigned

DIVU dst, src DIVUL	dst: R src: R, IM, IR, EAM
--------------------------------------	-------------------------------

Operation:

Word: (dst is longword register, src is word):
 $dst_{<31:0>} \leftarrow \frac{dst_{<31:0>}}{src_{<15:0>}}$
 $(dst_{<31:0>} = \text{quotient} \times src_{<15:0>} + remainder)$
 $dst_{<15:0>} \leftarrow \text{quotient}$
 $dst_{<31:16>} \leftarrow \text{remainder}$

Longword: (dst is quadword register, src is longword):
 $dst_{<63:0>} \leftarrow \frac{dst_{<63:0>}}{src_{<31:0>}}$
 $(dst_{<63:0>} = \text{quotient} \times src_{<31:0>} + remainder)$
 $dst_{<31:0>} \leftarrow \text{quotient}$
 $dst_{<63:32>} \leftarrow \text{remainder}$

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 unsigned integers. For DIVU, the destination is a longword register and the source is a word value; for DIVUL, the destination is a quadword register and the source is a longword value.

For proper instruction execution the “dst field” in the DIVUL instruction encoding must specify a valid code for a quadword register.

There are three possible outcomes of the unsigned divide instruction.

CASE 1. If the divisor is 0, then the destination register is unmodified, the V and Z flags are set to 1, and the C and S flags are cleared to 0.

CASE 2. If the quotient is greater than $(2^{16} - 1)$ for DIVU or if the quotient is greater than $(2^{32} - 1)$ for DIVUL, then the destination register is unmodified. The V flag is set to 1, and the C, Z, and S flags are cleared to 0.

CASE 3. If the quotient is less than 2^{16} for DIVU, or if the quotient is less than 2^{32} for DIVUL, then the quotient and remainder are left in the destination register as defined above. The V and C flags are cleared to 0 and the S and Z flags are set according to the value of the quotient, as described below.

Flags:	C: Cleared Z: Set if the quotient or divisor is zero; cleared otherwise S: For CASE 1 and CASE 2 cleared; for CASE 3 set if the most-significant bit of the result is set; cleared otherwise V: For CASE 1 and CASE 2 set; cleared otherwise D: Unaffected H: Unaffected
---------------	---

Exceptions:	Integer Overflow trap
--------------------	-----------------------

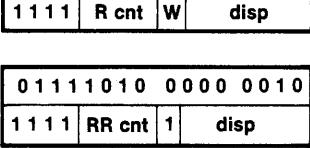
Source Addressing Mode	Assembler Language Syntax	Instruction Format							
R:	DIVU RRd, Rs	<table border="1"> <tr><td>01111010</td><td>0000 0011</td></tr> <tr><td>10 011011</td><td>Rs RRd</td></tr> </table>	01111010	0000 0011	10 011011	Rs RRd			
01111010	0000 0011								
10 011011	Rs RRd								
DIVUL RQd, RRs	<table border="1"> <tr><td>01111010</td><td>0000 0011</td></tr> <tr><td>10 011010</td><td>RRs RQd</td></tr> </table>	01111010	0000 0011	10 011010	RRs RQd				
01111010	0000 0011								
10 011010	RRs RQd								
IM:	DIVU RRd, #data	<table border="1"> <tr><td>01111010</td><td>0000 0011</td></tr> <tr><td>00 011011</td><td>0000 RRd</td></tr> <tr><td colspan="2">data</td></tr> </table>	01111010	0000 0011	00 011011	0000 RRd	data		
01111010	0000 0011								
00 011011	0000 RRd								
data									
DIVUL RQd, #data	<table border="1"> <tr><td>01111010</td><td>0000 0011</td></tr> <tr><td>00 011010</td><td>0000 RQd</td></tr> <tr><td colspan="2">data (high)</td></tr> <tr><td colspan="2">data (low)</td></tr> </table>	01111010	0000 0011	00 011010	0000 RQd	data (high)		data (low)	
01111010	0000 0011								
00 011010	0000 RQd								
data (high)									
data (low)									
IR:	DIVU RRd, @Rs1	<table border="1"> <tr><td>01111010</td><td>0000 0011</td></tr> <tr><td>00 011011</td><td>Rs≠0 RRd</td></tr> </table>	01111010	0000 0011	00 011011	Rs≠0 RRd			
01111010	0000 0011								
00 011011	Rs≠0 RRd								
DIVUL RQd, @Rs1	<table border="1"> <tr><td>01111010</td><td>0000 0011</td></tr> <tr><td>00 011010</td><td>Rs≠0 RQd</td></tr> </table>	01111010	0000 0011	00 011010	Rs≠0 RQd				
01111010	0000 0011								
00 011010	Rs≠0 RQd								
EAM:	DIVU RRd, eam	<table border="1"> <tr><td>01111010</td><td>0000 0011</td></tr> <tr><td>01 011011</td><td>eam RRd</td></tr> <tr><td colspan="2">1, 2, or 3 extension words</td></tr> </table>	01111010	0000 0011	01 011011	eam RRd	1, 2, or 3 extension words		
01111010	0000 0011								
01 011011	eam RRd								
1, 2, or 3 extension words									
DIVUL RQd, eam	<table border="1"> <tr><td>01111010</td><td>0000 0011</td></tr> <tr><td>01 011010</td><td>eam RQd</td></tr> <tr><td colspan="2">1, 2, or 3 extension words</td></tr> </table>	01111010	0000 0011	01 011010	eam RQd	1, 2, or 3 extension words			
01111010	0000 0011								
01 011010	eam RQd								
1, 2, or 3 extension words									

Example: If longword register RR0 (composed of word registers R0 and R1) contains the value %00000F00, executing the instruction
 DIVU RR0,#%81
 leaves the quotient %001D in R1 and the remainder %0063 in R0.

Note 1: Word register in compact mode, longword register in segmented or linear modes.

DJNZ

Decrement and Jump if Not Zero

DJNZ cnt, dst DBJNZ DLJNZ	cnt: R dst: RA	
Operation:	cnt \leftarrow cnt - 1 If cnt \neq 0 then PC \leftarrow PC - (2 \times displacement)	
	<p>The counter ('cnt') is decremented. If the contents of the counter are not zero after decrementing, the destination address is loaded into the Program Counter (PC). Otherwise, when the counter reaches zero, control falls through to the instruction following DJNZ, DBJNZ, or DLJNZ. This instruction provides a simple method of loop control.</p> <p>The destination address is calculated by subtracting twice the displacement in the instruction from the updated value of the PC. The updated PC value is the address of the instruction word following the DJNZ, DBJNZ, or DLJNZ instruction. 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 or -250 to 4 bytes from the start of the DLJNZ instruction. The assembler automatically calculates the displacement by subtracting the PC value of the following instruction from the address given by the programmer and dividing the result by two.</p>	
Flags:	No flags affected	
Exceptions:	None	
Destination Addressing Mode	Assembler Language Syntax	Instruction Format
RA:	DJNZ Rcnt, address DBJNZ Rbcnt, address DLJNZ RRcnt, address	

Example: DJNZ, DBJNZ and DLJNZ are typically used to control a "loop" of instructions. In this example for compact mode, 100 bytes are moved from one buffer area to another, and the Sign bit of each byte is cleared to 0. Register RH0 is used as the counter.

```
LDB      RH0,#100          //initialize counter
LD       R1,#SRCBUF        //load start address
LD       R2,#DSTBUF        //load destination address

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:

In segmented or linear mode, longword registers must be used instead of R1 and R2.

Privileged Instruction

EI Enable Interrupts

EI int	Int: VI, NVI				
Operation:	If instruction <0> = 0 then NVI \leftarrow 1 If instruction <1> = 0 then VI \leftarrow 1				
	Any combination of the Vectored Interrupt (VI) or Non-Vectored Interrupt (NVI) control bits in the Flag and Control Word (FCW) are set to 1 if the corresponding bit in the instruction is 0, thus enabling the appropriate type of interrupt. If the corresponding bit in the instruction is 1, the control bit is not affected. No other bits in the FCW are affected. There may be zero, one or two operands in the assembly language statement, in either order, specifying no source operand is equivalent to specifying both VI and NVI.				
Flags:	No flags affected				
Exceptions:	Privileged Instruction trap				
Assembler Language Syntax	Instruction Format				
EI int	<table border="1"><tr><td>01111100</td><td>000001</td><td>V</td><td>N</td></tr></table>	01111100	000001	V	N
01111100	000001	V	N		

Example: If the NVI control bit is set to 1 in the FCW, and the VI control bit is cleared 0, executing the instruction
EI VI
leaves both the NVI and VI control bits in the FCW set to 1.

ENTER

Enter

	ENTER mask, siz	mask: IM siz: IM
Operation:	tmp1 \leftarrow mask if FCW.E/C then n \leftarrow 13 else n \leftarrow 14 for i = n down to 8 do if tmp1 <i> = 1 then push RR [2 \times i - 16] for i = 7 down to 0 do if tmp1 <i> = 1 then push RR [2 \times i + 16] tmp2 \leftarrow tmp1 tmp2 <15> \leftarrow FCW.IV if FCW.E/C then push RR12 push tmp2 push 0 else push R14 push tmp2 push 0 FP \leftarrow SP SP \leftarrow SP + siz FCW.IV \leftarrow tmp1 <15>	//segmented or linear mode //compact mode //save registers //segmented or linear mode //save FP //save mask word //initialize exception handler address //(longword) //compact mode //save FP //save mask word //initialize exception handler address //(word) //allocate activation record //reserve local storage

This instruction is executed upon entering a procedure to allocate and initialize an activation record on the processor stack. The operation involves saving the specified general-purpose registers, saving and adjusting the Frame Pointer (FP), initializing the pointer to the procedure's exception handler, saving the current setting of the Integer Overflow trap enable bit, initializing the Integer Overflow trap enable bit, and reserving the local storage area.

The bits in the mask word operand (called the Enter Mask) correspond to general-purpose longword registers, as shown in Figure 6-2. When a mask bit is set to 1, the corresponding register is saved on the stack. Bit 15 of the Enter Mask corresponds to the setting of FCW.IV, the Integer Overflow trap enable bit, after the Enter instruction is executed. The Enter Mask is used to construct the Exit Mask, which is saved on the stack. The bits in the Exit Mask correspond to the longword registers that have been saved and the setting of FCW.IV before the Enter instruction is executed.

The activation record format in compact mode is shown in Figure 6-3a. After the saved PC, which has been pushed by the previous CALL or CALR instruction, the specified general-purpose longword registers are pushed on the stack. Next, the Frame Pointer (R14) is pushed on the stack, followed by the Exit Mask. Then a word containing 0 is pushed on the stack to initialize the pointer to the exception handler for the entered procedure. Finally, the size operand word is added to SP (R15), and FP is left pointing to the exception handler address.

The activation record format in segmented or linear mode, shown in Figure 6-3b, is similar. After the specified general-purpose longword registers are pushed onto the stack, the Frame Pointer (RR12) is pushed, followed by the Exit Mask. Then a longword containing 0 is pushed on the stack to initialize the exception handler pointer. Finally, the sign-extended size operand word is added to SP (RR14), and FP is left pointing to the exception handler address.

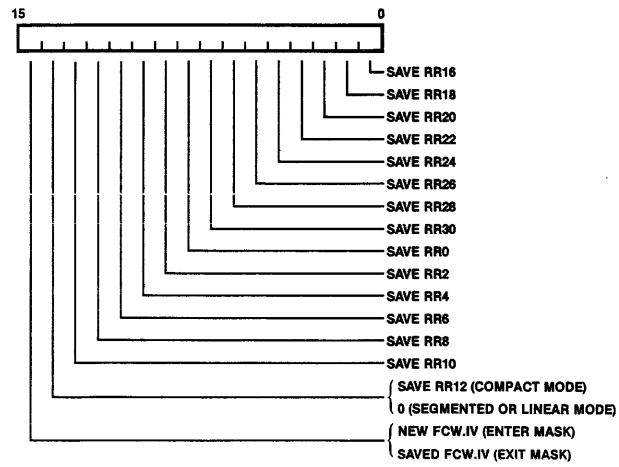


Figure 6-2. Enter Mask and Exit Mask Formats

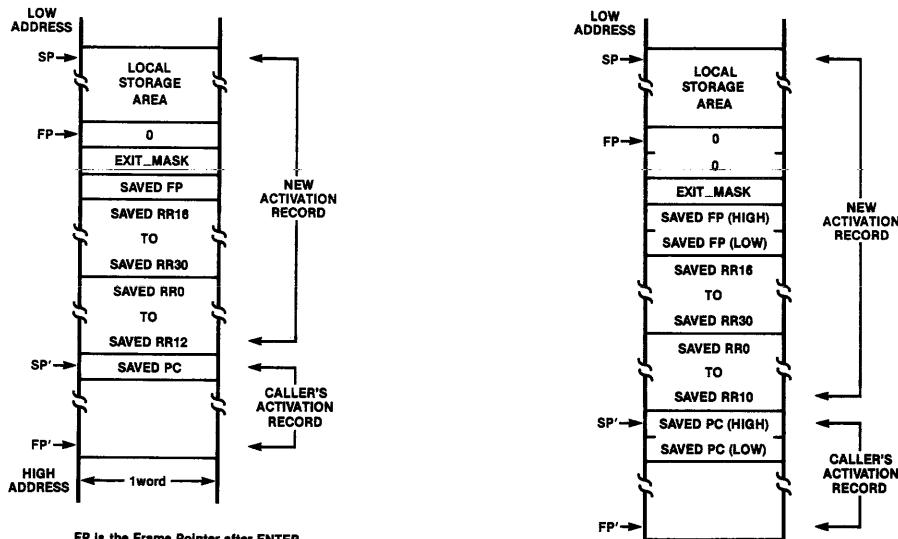


Figure 6-3a. Activation Record Format
(Compact Mode)

Figure 6-3b. Activation Record Format
(Segmented or Linear Mode)

Flags: No flags affected

Exceptions: None

Addressing Mode	Assembler Language Syntax	Instruction Format									
IM:	ENTER# enter__mask,#siz	<table border="1"><tr><td>0 1 1 1 1 0 1 0</td><td>0 0 0 0</td><td>0 1 0 1</td></tr><tr><td colspan="3">enter__mask</td></tr><tr><td colspan="3">siz</td></tr></table>	0 1 1 1 1 0 1 0	0 0 0 0	0 1 0 1	enter__mask			siz		
0 1 1 1 1 0 1 0	0 0 0 0	0 1 0 1									
enter__mask											
siz											

Example: Executing the instruction

ENTER #%05, #100

saves registers RR16 and RR20 on the stack, clears FCW.IV, and allocates an activation record with 100 bytes of local storage.

EX

Exchange

	EX dst, src EXB EXL	dst: R src: R, IR, EAM										
Operation:	tmp \leftarrow src src \leftarrow dst dst \leftarrow tmp	The contents of the source operand are exchanged with the contents of the destination operand.										
Flags:	No flags affected											
Exceptions:	None											
Source Addressing Mode	Assembler Language Syntax	Instruction Format										
R:	EX Rd, Rs EXB Rbd, Rbs	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr><td>10</td><td>10110</td><td>W</td><td>Rs</td><td>Rd</td></tr> </table>	10	10110	W	Rs	Rd					
	10	10110	W	Rs	Rd							
EXL RRd, RRs	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr><td>01111010</td><td>0000 0010</td></tr> <tr><td>10</td><td>101101</td><td>RRs</td><td>RRd</td></tr> </table>	01111010	0000 0010	10	101101	RRs	RRd					
01111010	0000 0010											
10	101101	RRs	RRd									
IR:	EX Rd, @Rs ¹ EXB Rbd, @Rs ¹	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr><td>00</td><td>10110</td><td>W</td><td>Rs\neq0</td><td>Rd</td></tr> </table>	00	10110	W	Rs \neq 0	Rd					
	00	10110	W	Rs \neq 0	Rd							
EXL RRd, @Rs ¹	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr><td>01111010</td><td>0000 0010</td></tr> <tr><td>00</td><td>101101</td><td>Rs\neq0</td><td>RRd</td></tr> </table>	01111010	0000 0010	00	101101	Rs \neq 0	RRd					
01111010	0000 0010											
00	101101	Rs \neq 0	RRd									
EAM:	EX Rd, eam EXB Rbd, eam	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr><td>01</td><td>10110</td><td>W</td><td>Rs\neq0</td><td>Rd</td></tr> <tr><td colspan="5" style="text-align: center;">address</td></tr> </table>	01	10110	W	Rs \neq 0	Rd	address				
	01	10110	W	Rs \neq 0	Rd							
address												
EXL RRd, eam	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr><td>01111010</td><td>0000 0010</td></tr> <tr><td>01</td><td>101101</td><td>Rs</td><td>RRd</td></tr> <tr><td colspan="4" style="text-align: center;">1, 2, or 3 extension words</td></tr> </table>	01111010	0000 0010	01	101101	Rs	RRd	1, 2, or 3 extension words				
01111010	0000 0010											
01	101101	Rs	RRd									
1, 2, or 3 extension words												

Example: If register R0 contains 8 and register R5 contains 9, executing the instruction
EX R0,R5
leaves the values 9 in R0 and 8 in R5.

Note 1: Word register in compact mode, longword register in segmented or linear modes.

EXIT

Exit

EXIT

Operation:

```
if FCW.E/C then           //segmented or linear mode
    SP ← FP + 4          //skip over exception handler
    pop tmp1               //Exit Mask
    pop RR12                //restore FP
    n ← 13
else                         //compact mode
    SP ← FP + 2          //skip over exception handler
    pop tmp1               //Exit Mask
    pop R14                //restore FP
    n ← 14
for i=0 to 7 do
    if tmp1<i> = 1 then pop RR [2 × i + 16]
for i=8 to n do
    if tmp1<i> = 1 then pop RR [2 × i - 16]
FCW.IV ← tmp1<15>
```

This instruction removes an activation record created with the ENTER instruction. (See the description of the ENTER instruction for more detailed information about the activation record and Exit Mask formats.)

In compact mode, first the value of the Frame Pointer (R14) is incremented by two and loaded into SP (R15), removing the local storage area and exception handler pointer from the processor stack. Next, the Exit Mask and Frame Pointer are popped from the stack. Then, the longword registers specified by the Exit Mask are popped from the stack, and FCW.IV is loaded from bit 15 of the Exit Mask.

In segmented or linear mode, first the value of the Frame Pointer (RR12) is incremented by four and loaded into SP (RR14), removing the local storage area and exception handler pointer from the processor stack. Next, the Exit Mask and Frame Pointer are popped from the stack. Then, the longword registers specified by the Exit Mask are popped from the stack, and FCW.IV is loaded from bit 15 of the Exit Mask.

Flags: No flags affected

Exceptions: None

	Assembler Language Syntax	Instruction Format		
	EXIT	<table border="1"><tr><td>01111010</td><td>0000 0110</td></tr></table>	01111010	0000 0110
01111010	0000 0110			

Example: At the end of a procedure that has been called using CALL or CALR instructions and that has been entered using the ENTER instruction, executing the instruction sequence

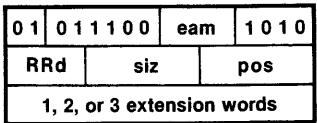
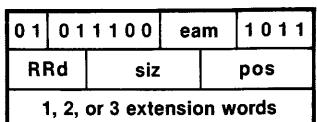
EXIT
RET

returns control to the caller at the instruction following the CALL and leaves the caller's activation record on top of the stack.

EXTR

Extract Field

EXTR	dst, src, pos, siz	dst: R src: R, IR, EAM pos: IM, R siz: IM, R																																																
Operation:	dst \leftarrow src (pos,siz)																																																	
This instruction is used to extract a bit field from memory or a longword register and load it into a longword register. For a description of bit fields see Section 6.2.6.																																																		
The bits in the source field are loaded, right-justified, into the least-significant bits of the destination longword register. For EXTR the remaining bits in the destination are loaded with the most-significant bit of the field. For EXTRU the remaining bits in the destination are cleared to 0.																																																		
The position and size operands can be specified as immediate values in the range 0 to 31 or in a word or longword register. The assembler encodes each operand in a 6-bit field of the instruction with the following format:																																																		
<table> <tr> <td>0 n n n n n</td> <td>5-bit unsigned immediate value</td> </tr> <tr> <td>1 0 r r r r</td> <td>word register contains value</td> </tr> <tr> <td>1 1 r r r r</td> <td>longword register contains value</td> </tr> </table>		0 n n n n n	5-bit unsigned immediate value	1 0 r r r r	word register contains value	1 1 r r r r	longword register contains value																																											
0 n n n n n	5-bit unsigned immediate value																																																	
1 0 r r r r	word register contains value																																																	
1 1 r r r r	longword register contains value																																																	
Flags:	C: Cleared Z: Set if the result is zero; cleared otherwise S: Set if the most-significant bit of the result is set; cleared otherwise V: Cleared D: Unaffected H: Unaffected																																																	
Exceptions:	None																																																	
Source Addressing Mode	Assembler Language Syntax	Instruction Format																																																
R:	EXTR RRd,RRs,pos,siz EXTRU RRd,RRs,pos,siz	<table border="1"> <tr> <td>1</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>RRs</td><td>1</td><td>0</td><td>1</td><td>0</td> </tr> <tr> <td>RRd</td><td></td><td>siz</td><td></td><td>pos</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td> </tr> </table> <table border="1"> <tr> <td>1</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>RRs</td><td>1</td><td>0</td><td>1</td><td>1</td> </tr> <tr> <td>RRd</td><td></td><td>siz</td><td></td><td>pos</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td> </tr> </table>	1	0	1	1	1	0	0	RRs	1	0	1	0	RRd		siz		pos								1	0	1	1	1	0	0	RRs	1	0	1	1	RRd		siz		pos							
1	0	1	1	1	0	0	RRs	1	0	1	0																																							
RRd		siz		pos																																														
1	0	1	1	1	0	0	RRs	1	0	1	1																																							
RRd		siz		pos																																														
IR:	EXTR RRd,@Rs ¹ ,pos,siz EXTRU RRd, @Rs ¹ ,pos,siz	<table border="1"> <tr> <td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>Rs≠0</td><td>1</td><td>0</td><td>1</td><td>0</td> </tr> <tr> <td>RRd</td><td></td><td>siz</td><td></td><td>pos</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td> </tr> </table> <table border="1"> <tr> <td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>Rs≠0</td><td>1</td><td>0</td><td>1</td><td>1</td> </tr> <tr> <td>RRd</td><td></td><td>siz</td><td></td><td>pos</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td> </tr> </table>	0	0	1	1	1	0	0	Rs≠0	1	0	1	0	RRd		siz		pos								0	0	1	1	1	0	0	Rs≠0	1	0	1	1	RRd		siz		pos							
0	0	1	1	1	0	0	Rs≠0	1	0	1	0																																							
RRd		siz		pos																																														
0	0	1	1	1	0	0	Rs≠0	1	0	1	1																																							
RRd		siz		pos																																														

Source Addressing Mode	Assembler Language Syntax	Instruction Format
EAM:	EXTR RRd,eam,pos,siz	 RRd siz pos 1, 2, or 3 extension words
	EXTRU RRd,eam,pos,siz	 RRd siz pos 1, 2, or 3 extension words

Example: If register RR4 contains %01200000 (0000 0001 0010 0000 0000 0000 0000 0000), executing the instruction

EXTR RR6,RR4,#7,#3

extracts the 4-bit field 1001 beginning at the 7th bit from the most-significant bit of RR4 and leaves the sign-extended value %FFFFFFF9 in RR6. Note that the size operand (#3) has a value one less than the number of bits in the field (4).

Note 1: Word register in compact mode, longword register in segmented or linear modes.

EXTS

Extend Sign

EXTSB dst EXTS EXTSL	dst: R	
Operation:		
Byte		
if $dst < 7 > = 0$	then $dst < 15:8 > \leftarrow 000...000$	
	else $dst < 15:8 > \leftarrow 111...111$	
Word		
if $dst < 15 > = 0$ then	$dst < 31:16 > \leftarrow 000...000$	
	else	
	$dst < 31:16 > \leftarrow 111...111$	
Longword		
if $dst < 31 > = 0$ then	$dst < 63:32 > \leftarrow 000...000$	
	else	
	$dst < 63:32 > \leftarrow 111...111$	
The Sign bit of the low-order half of the destination operand is copied into all bit positions of the high-order half of the destination. For EXTSB the destination is a word; for EXTS and EXTSL, the destination is a longword register.		
This instruction is useful in multiple precision arithmetic or for conversion of small signed operands to larger signed operands (for example, before a divide).		
Flags:	No flags affected	
Exceptions:	None	
Destination Addressing Mode	Assembler Language Syntax	Instruction Format
R:	EXTSB Rd EXTS RRd EXTSL RQd	  

Example: If longword register RR2 (composed of word registers R2 and R3) contains %12345678, executing the instruction

EXTS RR2

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

HALT

Halt

Privileged Instruction

HALT

Operation: The CPU enters halted state (see Section 7.2), in which instruction execution ceases. Only the occurrence of reset or an enabled interrupt causes the CPU to leave halted state. After HALT is executed, the address of the instruction following HALT is in the PC, which will be saved on the system stack during interrupt processing.

Flags: No flags affected

Exceptions: Privileged Instruction trap

	Assembler Language Syntax	Instruction Format		
	HALT	<table border="1"><tr><td>01111010</td><td>00000000</td></tr></table>	01111010	00000000
01111010	00000000			

Privileged Instruction

IN
Input

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

Operation $dst \leftarrow src$

The contents of the source operand, an input port, are loaded into the destination register. I/O port addresses are 16 bits.

Flags: No flags affected

Exceptions: Privileged Instruction trap

Source Addressing Mode	Assembler Language Syntax	Instruction Format										
IR:	IN Rd, @Rs INB Rbd, @Rs	<table border="1"> <tr><td>00</td><td>11110</td><td>W</td><td>Rs ≠ 0</td><td>Rd</td></tr> </table>	00	11110	W	Rs ≠ 0	Rd					
00	11110	W	Rs ≠ 0	Rd								
INL RRd, @Rs	<table border="1"> <tr><td>01111010</td><td>0000 0010</td></tr> <tr><td>00</td><td>111101</td><td>Rs ≠ 0</td><td>RRd</td></tr> </table>	01111010	0000 0010	00	111101	Rs ≠ 0	RRd					
01111010	0000 0010											
00	111101	Rs ≠ 0	RRd									
DA:	IN Rd, port INB Rbd, port	<table border="1"> <tr><td>00</td><td>11101</td><td>W</td><td>Rd</td><td>0100</td></tr> <tr><td colspan="5">port</td></tr> </table>	00	11101	W	Rd	0100	port				
00	11101	W	Rd	0100								
port												
INL RRd, port	<table border="1"> <tr><td>01111010</td><td>0000 0010</td></tr> <tr><td>00</td><td>111011</td><td>RRd</td><td>0100</td></tr> <tr><td colspan="4">port</td></tr> </table>	01111010	0000 0010	00	111011	RRd	0100	port				
01111010	0000 0010											
00	111011	RRd	0100									
port												

Example: If register R6 contains the I/O port address %0123 and the port %0123 contains %FF, executing the instruction
INB RH2, @R6
leaves the value %FF in register RH2.

INC

Increment

INC dst, src	dst: R, IR, EAM
INCB	src: IM
INCL	

Operation: $\text{dst} \leftarrow \text{dst} + \text{src}$ ($\text{src} = 1$ 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. Twos complement addition is performed. If the source operand is omitted from the assembler language statement, the default value is 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
--------	--

Exceptions: Integer Overflow trap

Destination Addressing Mode	Assembler Language Syntax	Instruction Format											
R:	INC Rd, #n INCB Rbd, #n	<table border="1" style="margin-bottom: 10px;"> <tr><td>10</td><td>10100</td><td>W</td><td>Rd</td><td>n - 1</td></tr> </table> <table border="1"> <tr><td>01111010</td><td>0000 0010</td></tr> <tr><td>10</td><td>101001</td><td>RRd</td><td>n - 1</td></tr> </table>	10	10100	W	Rd	n - 1	01111010	0000 0010	10	101001	RRd	n - 1
10	10100	W	Rd	n - 1									
01111010	0000 0010												
10	101001	RRd	n - 1										
INCL RRd, #n													
IR:	INC @Rd ¹ , #n INCB @Rd ¹ , #n	<table border="1" style="margin-bottom: 10px;"> <tr><td>00</td><td>10100</td><td>W</td><td>Rd ≠ 0</td><td>n - 1</td></tr> </table> <table border="1"> <tr><td>01111010</td><td>0000 0010</td></tr> <tr><td>00</td><td>101001</td><td>Rd ≠ 0</td><td>n - 1</td></tr> </table>	00	10100	W	Rd ≠ 0	n - 1	01111010	0000 0010	00	101001	Rd ≠ 0	n - 1
00	10100	W	Rd ≠ 0	n - 1									
01111010	0000 0010												
00	101001	Rd ≠ 0	n - 1										
INCL @Rd ¹ , #n													
EAM:	INC eam, #n INCB eam, #n	<table border="1" style="margin-bottom: 10px;"> <tr><td>01</td><td>10100</td><td>W</td><td>eam</td><td>n - 1</td></tr> </table> <p>1, 2, or 3 extension words</p>	01	10100	W	eam	n - 1						
01	10100	W	eam	n - 1									
INCL eam, #n	<table border="1" style="margin-bottom: 10px;"> <tr><td>01111010</td><td>0000 0010</td></tr> <tr><td>01</td><td>101001</td><td>eam</td><td>n - 1</td></tr> </table> <p>1, 2, or 3 extension words</p>	01111010	0000 0010	01	101001	eam	n - 1						
01111010	0000 0010												
01	101001	eam	n - 1										

Example: If register RH2 contains %21, executing the instruction
INCB RH2,#6
leaves the value %27 in RH2.

Note 1: Word register in compact mode, longword register in segmented or linear modes.

INCI

Increment Interlocked

INCI dst, src	dst: IR, EAM
INCIB	src: IM

Operation: $dst \leftarrow dst + src$ ($src = 1$ to 16)

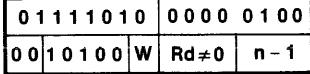
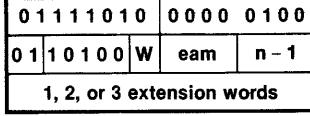
The source operand (a value from 1 to 16) is added to the destination operand and the sum is stored in the destination. Twos complement addition is performed. If the source operand is missing from the assembler language statement, the default value is 1.

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.

This is an interlocked instruction. No other interlocked accesses are permitted to the destination memory location between fetching and storing the result.

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
---------------	---

Exceptions: Integer Overflow trap

Destination Addressing Mode	Assembler Language Syntax	Instruction Format
IR:	INCI @Rd ¹ , #n INCIB @Rd ¹ , #n	
EAM:	INCI eam, #n INCIB eam, #n	 1, 2, or 3 extension words

Example: This instruction can be used to allocate or release copies of a system resource in a multiprocessor environment. For example, several processes running on different processors can share use of a common page in memory. It is necessary to keep a reference counter for the number of active processes using the shared page. When a new process requires use of the page the reference counter is incremented. The INCI instruction should be used so that one processor completes the fetch and store of the counter in memory before any other processor accesses the counter.

```
INCI  REFERENCE_COUNTER, #1           //increment reference counter
                                         //for shared page
```

Note 1: Word register in compact mode, longword register in segmented or linear modes.

Privileged Instruction

IND Input and Decrement

IND dst, src, r INDB INDL	dst: IR src: IR
--	--------------------

Operation: $dst \leftarrow src$
AUTODECREMENT dst (by 1 if INDB; by 2 if IND; by 4 if INDL)
 $r \leftarrow r - 1$

This instruction is used for block input of strings of data. 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 INDB, by two if IND, or by four if INDL, 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 source, destination, and counter registers must be distinct and non-overlapping registers.

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

Exceptions: Privileged Instruction trap

Addressing Mode	Assembler Language Syntax	Instruction Format														
IR:	IND @Rd ¹ , @Rs, r INDB @Rd ¹ , @Rs, r	<div style="border: 1px solid black; padding: 5px; margin-bottom: 10px;"> <table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 10%;">0011101</td> <td style="width: 10%;">W</td> <td style="width: 10%;">Rs ≠ 0</td> <td style="width: 10%;">1000</td> </tr> <tr> <td>0000</td> <td>r</td> <td>Rd ≠ 0</td> <td>1000</td> </tr> </table> </div> <div style="border: 1px solid black; padding: 5px;"> <table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 10%;">01111010</td> <td style="width: 10%;">0000 0010</td> </tr> <tr> <td>00111011</td> <td>Rs ≠ 0 1000</td> </tr> <tr> <td>0000</td> <td>r Rd ≠ 0 1000</td> </tr> </table> </div>	0011101	W	Rs ≠ 0	1000	0000	r	Rd ≠ 0	1000	01111010	0000 0010	00111011	Rs ≠ 0 1000	0000	r Rd ≠ 0 1000
0011101	W	Rs ≠ 0	1000													
0000	r	Rd ≠ 0	1000													
01111010	0000 0010															
00111011	Rs ≠ 0 1000															
0000	r Rd ≠ 0 1000															
INDL @Rd ¹ , @Rs, r																

Example: In linear mode, if register RR24 contains %00004000, register R6 contains the I/O port address %0228, the port %0228 contains %05B9, and register R0 contains %0016, executing the instruction

IND @RR24, @R6, R0

leaves the value %05B9 in location %00004000, the value %00003FFE in RR24, and the value %0015 in R0. The V flag is cleared. Register R6 still contains the value %0228. In compact mode, a word register must be used instead of RR24.

Note 1: Word register in compact mode, longword register in segmented or linear modes.

INDEX

Index

INDEX dst, sub, src	dst: R
INDEXL	sub: R
	src: IM,IR,EAM

Operation:

```
tmp ← EFFECTIVE_ADDRESS (src)
lower ← @tmp
if sub < lower then Index Error trap
tmp ← tmp + (2 if INDEX; 4 if INDEXL)
upper ← @tmp
if sub > upper then Index Error trap
tmp ← tmp + (2 if INDEX; 4 if INDEXL)
scale ← @tmp
dst ← (dst + (sub - lower)) × scale
```

This instruction is used to check an array subscript and calculate the corresponding index value. For arrays with multiple dimensions, the instruction performs one step of the index calculation, accumulating the index value in the destination.

The subscript is compared against the bounds specified by the source operand. If the subscript is less than the lower bound or greater than the upper bound, then the destination and flags are unaffected and an Index trap occurs. If the subscript is in bounds, then the lower bound is subtracted from the subscript, the difference is added to the destination, the sum is multiplied by the scale factor, and the product is stored into the destination. The subscript, lower bound, upper bound, scale factor, and destination are all the same size, either word or longword. The operands are treated as signed integers. The contents of the subscript and source are not affected.

The source operand specifies the lower bound. The upper bound and scale factor are located at the next two consecutive words or longwords.

When the instruction is used appropriately, an Index trap occurs if the calculated index is outside the array. Hence, overflow is not detected during the index calculation. If overflow does occur during addition, only the less-significant word or longword of the sum is stored into the destination. If overflow does occur during multiplication, only the less-significant word or longword of the product is stored.

Flags:

- C:** Unaffected if Index Error trap; cleared otherwise
- Z:** Unaffected if Index Error trap; else set if the result is zero; cleared otherwise
- S:** Unaffected if Index Error trap; else set if the most-significant bit of the result is set; cleared otherwise
- V:** Unaffected if Index Error trap; cleared otherwise
- D:** Unaffected
- H:** Unaffected

Exceptions: Index Error trap

Source Addressing Mode	Assembler Language Syntax	Instruction Format																															
IM:	INDEX Rd, Rsub, #lower,#upper, #scale	<table border="1" data-bbox="959 371 1269 561"> <tr><td>0 0</td><td>0 0 1 1 0 1</td><td>0 0 0 0</td><td>1 1 1 0</td></tr> <tr><td>0 0 0 0</td><td>Rsub</td><td>Rd ≠ 0</td><td>0 0 0 0</td></tr> <tr><td colspan="4">lower</td></tr> <tr><td colspan="4">upper</td></tr> <tr><td colspan="4">scale</td></tr> </table>	0 0	0 0 1 1 0 1	0 0 0 0	1 1 1 0	0 0 0 0	Rsub	Rd ≠ 0	0 0 0 0	lower				upper				scale														
0 0	0 0 1 1 0 1	0 0 0 0	1 1 1 0																														
0 0 0 0	Rsub	Rd ≠ 0	0 0 0 0																														
lower																																	
upper																																	
scale																																	
INDEXL RRd, RRsub, #lower,#upper, #scale	<table border="1" data-bbox="959 604 1269 910"> <tr><td>0 0</td><td>0 0 1 1 0 1</td><td>0 0 0 0</td><td>1 1 1 1</td></tr> <tr><td>0 0 0 0</td><td>RRsub</td><td>RRd ≠ 0</td><td>0 0 0 0</td></tr> <tr><td colspan="4">lower (high)</td></tr> <tr><td colspan="4">lower (low)</td></tr> <tr><td colspan="4">upper (high)</td></tr> <tr><td colspan="4">upper (low)</td></tr> <tr><td colspan="4">scale (high)</td></tr> <tr><td colspan="4">scale (low)</td></tr> </table>	0 0	0 0 1 1 0 1	0 0 0 0	1 1 1 1	0 0 0 0	RRsub	RRd ≠ 0	0 0 0 0	lower (high)				lower (low)				upper (high)				upper (low)				scale (high)				scale (low)			
0 0	0 0 1 1 0 1	0 0 0 0	1 1 1 1																														
0 0 0 0	RRsub	RRd ≠ 0	0 0 0 0																														
lower (high)																																	
lower (low)																																	
upper (high)																																	
upper (low)																																	
scale (high)																																	
scale (low)																																	
IR:	INDEX Rd, Rsub, @Rs ¹	<table border="1" data-bbox="959 952 1269 1036"> <tr><td>0 0</td><td>0 0 1 1 0 1</td><td>Rs ≠ 0</td><td>1 1 1 0</td></tr> <tr><td>0 0 0 0</td><td>Rsub</td><td>Rd ≠ 0</td><td>0 0 0 0</td></tr> </table>	0 0	0 0 1 1 0 1	Rs ≠ 0	1 1 1 0	0 0 0 0	Rsub	Rd ≠ 0	0 0 0 0																							
0 0	0 0 1 1 0 1	Rs ≠ 0	1 1 1 0																														
0 0 0 0	Rsub	Rd ≠ 0	0 0 0 0																														
INDEXL RRd, RRsub, @Rs ¹	<table border="1" data-bbox="959 1079 1269 1163"> <tr><td>0 0</td><td>0 0 1 1 0 1</td><td>Rs ≠ 0</td><td>1 1 1 1</td></tr> <tr><td>0 0 0 0</td><td>RRsub</td><td>RRd ≠ 0</td><td>0 0 0 0</td></tr> </table>	0 0	0 0 1 1 0 1	Rs ≠ 0	1 1 1 1	0 0 0 0	RRsub	RRd ≠ 0	0 0 0 0																								
0 0	0 0 1 1 0 1	Rs ≠ 0	1 1 1 1																														
0 0 0 0	RRsub	RRd ≠ 0	0 0 0 0																														
EAM:	INDEX Rd, Rsub, eam	<table border="1" data-bbox="959 1184 1269 1290"> <tr><td>0 1</td><td>0 0 1 1 0 1</td><td>eam</td><td>1 1 1 0</td></tr> <tr><td>0 0 0 0</td><td>Rsub</td><td>Rd ≠ 0</td><td>0 0 0 0</td></tr> <tr><td colspan="4">1, 2, or 3 extension words</td></tr> </table>	0 1	0 0 1 1 0 1	eam	1 1 1 0	0 0 0 0	Rsub	Rd ≠ 0	0 0 0 0	1, 2, or 3 extension words																						
0 1	0 0 1 1 0 1	eam	1 1 1 0																														
0 0 0 0	Rsub	Rd ≠ 0	0 0 0 0																														
1, 2, or 3 extension words																																	
INDEXL RRd, RRsub, eam	<table border="1" data-bbox="959 1364 1269 1448"> <tr><td>0 1</td><td>0 0 1 1 0 1</td><td>eam</td><td>1 1 1 1</td></tr> <tr><td>0 0 0 0</td><td>RRsub</td><td>RRd ≠ 0</td><td>0 0 0 0</td></tr> <tr><td colspan="4">1, 2, or 3 extension words</td></tr> </table>	0 1	0 0 1 1 0 1	eam	1 1 1 1	0 0 0 0	RRsub	RRd ≠ 0	0 0 0 0	1, 2, or 3 extension words																							
0 1	0 0 1 1 0 1	eam	1 1 1 1																														
0 0 0 0	RRsub	RRd ≠ 0	0 0 0 0																														
1, 2, or 3 extension words																																	

Example: The subscript values for a two-dimensional array of records range from 10 to 20 and from 1 to 100. Each record in the array is 12 bytes. The base address of the array is contained in RR2, the first subscript value is contained in RR6, and the second subscript value is in RR8. Executing the instruction sequence (in segmented or linear mode)

```
CLRL RR4                                //initialize index register  
INDEXL RR4,RR6,#10,#20,#100               //check and accumulate first  
                                         //subscript  
INDEXL RR4,RR8,#1,#100,#12                 //calculate array index  
LDB RH0,RR2(RR4)                          //load first byte of record
```

loads the first byte of the indexed record into RH0.

Note 1: Word register in compact mode, longword register in segmented or linear modes.

Privileged Instruction

INDR

Input, Decrement and Repeat

INDR dst, src, r
INDRB
INDRL

dst: IR
src: IR

Operation: repeat
dst \leftarrow src
AUTODECREMENT dst (by 1 if INDRB; by 2 if INDR; by 4 if INDRL)
r \leftarrow r - 1
until r = 0

This instruction is used for block input of strings of data. 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 INDRB, by two if INDR, or by four if INDRL, 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 65,536 data elements. The source, destination, and counter registers must be distinct, non-overlapping registers.

This instruction can be interrupted after each execution of the basic operation.

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

Exceptions: Privileged Instruction trap

Addressing Mode	Assembler Language Syntax	Instruction Format								
IR:	INDR @Rd ¹ , @Rs, r INDRB @Rd ¹ , @Rs, r	<table border="1"> <tr> <td>0011101</td> <td>W</td> <td>Rs ≠ 0</td> <td>1000</td> </tr> <tr> <td>0000</td> <td>r</td> <td>Rd ≠ 0</td> <td>0000</td> </tr> </table>	0011101	W	Rs ≠ 0	1000	0000	r	Rd ≠ 0	0000
0011101	W	Rs ≠ 0	1000							
0000	r	Rd ≠ 0	0000							
INDRL @Rd ¹ , @Rs, r	<table border="1"> <tr> <td>01111010</td> <td>0000 0010</td> </tr> <tr> <td>00111011</td> <td>Rs ≠ 0 1000</td> </tr> <tr> <td>0000</td> <td>r Rd ≠ 0 0000</td> </tr> </table>	01111010	0000 0010	00111011	Rs ≠ 0 1000	0000	r Rd ≠ 0 0000			
01111010	0000 0010									
00111011	Rs ≠ 0 1000									
0000	r Rd ≠ 0 0000									

Example: In compact mode, if register R1 contains %202A, register R2 contains the I/O address %0AFC, and register R3 contains 8, executing the instruction

INDRB @R1, @R2, R3

inputs 8 bytes from the I/O port %0AFC and leaves them in descending order from %202A to %2023. Register R1 contains %2022, and R3 contains 0. R2 is not affected. The V flag is set. In segmented or linear mode, a longword register must be used instead of R1.

Note 1: Word register in compact mode, longword register in segmented or linear modes.

Privileged Instruction

INI

Input and Increment

Operation: $\text{dst} \leftarrow \text{src}$
AUTOINCREMENT dst (by 1 if INIB; by 2 if INI; by 4 if INIL)
 $r \leftarrow r - 1$

This instruction is used for block input of strings of data. 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 INIB, by two if INI, or by four if INIL, 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. The source, destination, and counter registers must be distinct, non-overlapping registers.

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

Exceptions: Privileged Instruction trap

Addressing Mode	Assembler Language Syntax	Instruction Format											
IR:	INI @Rd ¹ , @Rs, r INIB @Rd ¹ , @Rs, r	<table border="1" data-bbox="944 1157 1261 1216"> <tr> <td>0011101</td> <td>W</td> <td>Rs ≠ 0</td> <td>0000</td> </tr> <tr> <td>0000</td> <td>r</td> <td>Rd ≠ 0</td> <td>1000</td> </tr> </table>	0011101	W	Rs ≠ 0	0000	0000	r	Rd ≠ 0	1000			
0011101	W	Rs ≠ 0	0000										
0000	r	Rd ≠ 0	1000										
INIL @Rd ¹ , @Rs, r	<table border="1" data-bbox="944 1256 1261 1315"> <tr> <td>01111010</td> <td>0000</td> <td>0010</td> </tr> <tr> <td>00111011</td> <td>Rs ≠ 0</td> <td>0000</td> </tr> <tr> <td>0000</td> <td>r</td> <td>Rd ≠ 0</td> </tr> <tr> <td>1000</td> <td></td> <td></td> </tr> </table>	01111010	0000	0010	00111011	Rs ≠ 0	0000	0000	r	Rd ≠ 0	1000		
01111010	0000	0010											
00111011	Rs ≠ 0	0000											
0000	r	Rd ≠ 0											
1000													

Example: In compact 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, executing the instruction

INIB @R4, @R6, R0

leaves 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 or linear mode, a longword register must be used instead of R4.

Note 1: Word register in compact mode, longword register in segmented or linear modes.

INIR

Privileged Instruction

Input, Increment and Repeat

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

Operation:

```

repeat
    dst ← src
    AUTOINCREMENT dst (by 1 if INIRB; by 2 if INIR; by 4 if INIRL)
    r ← r - 1
until r = 0

```

This instruction is used for block input of strings of data. 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 INIRB, by two if INIR, or by four if INIRL, thus moving the pointer to the next element in 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 65,536 data elements. The source, destination, and counter registers must be distinct, non-overlapping registers.

This instruction can be interrupted after each execution of the basic operation.

Flags:

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

Exceptions: Privileged Instruction trap

Addressing Mode	Assembler Language Syntax	Instruction Format														
IR:	INIR @Rd ¹ , @Rs, r INIRB @Rd ¹ , @Rs, r INIRL @Rd ¹ , @Rs, r	<table border="1" style="margin-bottom: 10px;"> <tr> <td>0011101</td> <td>W</td> <td>Rs ≠ 0</td> <td>0000</td> </tr> <tr> <td>0000</td> <td>r</td> <td>Rd ≠ 0</td> <td>0000</td> </tr> </table> <table border="1"> <tr> <td>01111010</td> <td>0000 0010</td> </tr> <tr> <td>00111011</td> <td>Rs ≠ 0</td> </tr> <tr> <td>0000</td> <td>Rd ≠ 0 0000</td> </tr> </table>	0011101	W	Rs ≠ 0	0000	0000	r	Rd ≠ 0	0000	01111010	0000 0010	00111011	Rs ≠ 0	0000	Rd ≠ 0 0000
0011101	W	Rs ≠ 0	0000													
0000	r	Rd ≠ 0	0000													
01111010	0000 0010															
00111011	Rs ≠ 0															
0000	Rd ≠ 0 0000															

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

INIRB @R1, @R2, R3

inputs 8 bytes from port %0551 and leave them in ascending order from %2023 to %202A. Register R1 contains %202B, and R3 contains 0. R2 is not affected. The V flag is set. In segmented or linear mode, a longword register must be used instead of R1.

Note 1: Word register in compact mode, longword register in segmented or linear modes.

INSRT

Insert Field

INSRT dst, src, pos, siz	dst: R, IR, EAM src: R pos: R, IM siz: R, IM
---------------------------------	---

Operation: dst (pos, siz) \leftarrow src

This instruction is used to insert a bit field from a longword register into memory or a longword register. For a description of bit fields, see Section 6.2.6.

The bits in the destination field are loaded from the least-significant bits of the source register.

The position and size operands can be specified as immediate values in the range 0 to 31 or in a word or longword register. The assembler encodes each operand in a 6-bit field of the instruction with the following format:

0 n n n n n	5-bit unsigned immediate value
1 0 r r r r r	word register contains value
1 1 r r r r r	longword register contains value

Flags: No flags affected

Exceptions: None

Destination Addressing Mode	Assembler Language Syntax	Instruction Format												
R:	INSRT RRd,RRs,pos,siz	<table border="1"> <tr> <td>1 0</td> <td>0 1 1 1 0 0</td> <td>RRd</td> <td>0 1 1 0</td> </tr> <tr> <td>RRs</td> <td>siz</td> <td>pos</td> <td></td> </tr> </table>	1 0	0 1 1 1 0 0	RRd	0 1 1 0	RRs	siz	pos					
1 0	0 1 1 1 0 0	RRd	0 1 1 0											
RRs	siz	pos												
IR:	INSRT @Rd ¹ ,RRs,pos,siz	<table border="1"> <tr> <td>0 0</td> <td>0 1 1 1 0 0</td> <td>Rd ≠ 0</td> <td>0 1 1 0</td> </tr> <tr> <td>RRs</td> <td>siz</td> <td>pos</td> <td></td> </tr> </table>	0 0	0 1 1 1 0 0	Rd ≠ 0	0 1 1 0	RRs	siz	pos					
0 0	0 1 1 1 0 0	Rd ≠ 0	0 1 1 0											
RRs	siz	pos												
EAM:	INSRT eam,RRs,pos,siz	<table border="1"> <tr> <td>0 1</td> <td>0 1 1 1 0 0</td> <td>eam</td> <td>0 1 1 0</td> </tr> <tr> <td>RRs</td> <td>siz</td> <td>pos</td> <td></td> </tr> <tr> <td colspan="4">1, 2, or 3 extension words</td></tr> </table>	0 1	0 1 1 1 0 0	eam	0 1 1 0	RRs	siz	pos		1, 2, or 3 extension words			
0 1	0 1 1 1 0 0	eam	0 1 1 0											
RRs	siz	pos												
1, 2, or 3 extension words														

Example: If register RR2 contains %0101012A (0000 0001 0000 0001 0000 0001 0010 1010) and register RR4 contains %FFFF FFFF, executing the instruction

INSRT RR4,RR2,#4,#6

inserts the 7-bit field 0101010 from the least-significant bits of RR2 into RR4 beginning at the 4th from the most-significant bit, leaving %F55FFFFF (1111 0101 0101 1111 1111 1111 1111 1111 in RR4. Note that the size operand (#6) has a value one less than the number of bits in the field (7).

Note 1: Word register in compact mode, longword register in segmented or linear modes.

Privileged Instruction

IRET Interrupt Return

IRET

Operation: $SP \leftarrow SP + 2$ //pop "identifier"
 pop tmp //pop FCW
 pop PC
 if FCW.T then tmp<9> $\leftarrow 1$
 FCW \leftarrow tmp

This instruction is used at the end of an exception handler routine to return to the program at the point where the exception occurred. First, an "identifier" word associated with the exception is popped from the stack. Then, the FCW and PC are popped from the stack.

After IRET is executed, the Trace Pending bit (FCW.TP) is set if bit 9 is set in the popped FCW or if the Trace Enable bit (FCW.T) was set before the instruction was executed. This allows tracing of exception handler routines for single-step debugging. This instruction may be executed in segmented or linear mode only; in compact mode, execution of this instruction is undefined.

Flags: **C:** Loaded from system stack
 Z: Loaded from system stack
 S: Loaded from system stack
 P/V: Loaded from system stack
 D: Loaded from system stack
 H: Loaded from system stack

Exceptions: Privileged Instruction trap

	Assembler Language Syntax	Instruction Format		
	IRET	<table border="1"><tr><td>01111011</td><td>00000000</td></tr></table>	01111011	00000000
01111011	00000000			

JP
Jump

Operation: If cc is satisfied, then $PC \leftarrow$ EFFECTIVE ADDRESS (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 Section 6.3 for a list of condition codes. If the condition is satisfied, the Program Counter (PC) is loaded with the destination address; otherwise, the instruction following the JP instruction is executed. If no condition is specified, the jump is taken regardless of the flag settings.

Flags: No flags affected

Exceptions: None

Destination Addressing Mode	Assembler Language Syntax	Instruction Format				
IR:	JP cc, @Rd ¹	<table border="1" data-bbox="856 882 1160 906"> <tr> <td>00</td> <td>011110</td> <td>Rd ≠ 0</td> <td>cc</td> </tr> </table>	00	011110	Rd ≠ 0	cc
00	011110	Rd ≠ 0	cc			
EAM:	JP cc, eam	<table border="1" data-bbox="856 941 1160 963"> <tr> <td>01</td> <td>011110</td> <td>eam</td> <td>cc</td> </tr> </table> <p data-bbox="878 973 1139 982">1, 2, or 3 extension words</p>	01	011110	eam	cc
01	011110	eam	cc			

Example: If the C flag is set, executing the instruction (in compact mode)

JP C. % 1520

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

Note 1: Word register in compact mode, longword register in segmented or linear modes

JR Jump Relative

JR cc, dst

dst: RA

Operation: if cc is satisfied then $PC \leftarrow PC + (2 \times \text{displacement})$

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 Section 6.3 for a list of condition codes. If the condition is satisfied, the Program Counter (PC) is loaded with the destination address; otherwise, the instruction following the JR instruction is executed. If no condition is specified, the jump is taken regardless of the flag settings.

The destination address is calculated by adding twice the displacement in the instruction to the updated value of the PC. The updated PC value is the address of the instruction word following the JR instruction. 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. The assembler automatically calculates the displacement by subtracting the PC value of the following instruction from the address given by the programmer and dividing the result by two.

Flags: No flags affected

Exceptions: None

Destination Addressing Mode	Assembler Language Syntax	Instruction Format						
RA:	JR cc, address	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td style="padding: 2px;">1</td> <td style="padding: 2px;">1</td> <td style="padding: 2px;">1</td> <td style="padding: 2px;">0</td> <td style="padding: 2px;">cc</td> <td style="padding: 2px;">displacement</td> </tr> </table>	1	1	1	0	cc	displacement
1	1	1	0	cc	displacement			

Example: If the result of the last arithmetic operation executed is negative, the next 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 omitted in this case, indicating that the jump is always taken.

LD Load

LD dst, src	dst: R
LDB	src: R, IR, BA, BX, EAM
LDL	or
	dst: IR, BA, BX, EAM
	src: R
	or
	dst: R, IR, EAM
	src: IM

Operation: $dst \leftarrow src$

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

Exceptions: None

Load Register

Source Addressing Mode	Assembler Language Syntax	Instruction Format										
R:	LD Rd, Rs LDB Rbd, Rbs	<table border="1"><tr><td>10</td><td>10000</td><td>W</td><td>Rs</td><td>Rd</td></tr></table>	10	10000	W	Rs	Rd					
10	10000	W	Rs	Rd								
LDL RRD, RRs	<table border="1"><tr><td>10</td><td>010100</td><td>RRs</td><td>RRd</td></tr></table>	10	010100	RRs	RRd							
10	010100	RRs	RRd									
IR:	LD Rd, @Rs ¹ LDB Rbd, @Rs ¹	<table border="1"><tr><td>00</td><td>10000</td><td>W</td><td>Rs ≠ 0</td><td>Rd</td></tr></table>	00	10000	W	Rs ≠ 0	Rd					
00	10000	W	Rs ≠ 0	Rd								
LDL RRD, @Rs ¹	<table border="1"><tr><td>00</td><td>010100</td><td>Rs ≠ 0</td><td>RRd</td></tr></table>	00	010100	Rs ≠ 0	RRd							
00	010100	Rs ≠ 0	RRd									
BA:	LD Rd, Rs ¹ (disp) LDB Rbd, Rs ¹ (disp)	<table border="1"><tr><td>00</td><td>11000</td><td>W</td><td>Rs ≠ 0</td><td>Rd</td></tr><tr><td colspan="5">displacement</td></tr></table>	00	11000	W	Rs ≠ 0	Rd	displacement				
00	11000	W	Rs ≠ 0	Rd								
displacement												
LDL RRD, Rs ¹ (disp)	<table border="1"><tr><td>00</td><td>110101</td><td>Rs ≠ 0</td><td>RRd</td></tr><tr><td colspan="5">displacement</td></tr></table>	00	110101	Rs ≠ 0	RRd	displacement						
00	110101	Rs ≠ 0	RRd									
displacement												
BX:	LD Rd, Rs ¹ (Rx) LDB Rbd, Rs ¹ (Rx)	<table border="1"><tr><td>01</td><td>11000</td><td>W</td><td>Rs ≠ 0</td><td>Rd</td></tr><tr><td>0000</td><td>Rx ≠ 0</td><td>0000</td><td>0000</td><td>0000</td></tr></table>	01	11000	W	Rs ≠ 0	Rd	0000	Rx ≠ 0	0000	0000	0000
01	11000	W	Rs ≠ 0	Rd								
0000	Rx ≠ 0	0000	0000	0000								
LDL RRD, Rs ¹ (Rx)	<table border="1"><tr><td>01</td><td>110101</td><td>Rs ≠ 0</td><td>RRd</td></tr><tr><td>0000</td><td>Rx ≠ 0</td><td>0000</td><td>0000</td></tr></table>	01	110101	Rs ≠ 0	RRd	0000	Rx ≠ 0	0000	0000			
01	110101	Rs ≠ 0	RRd									
0000	Rx ≠ 0	0000	0000									

Load Register (Continued)

Source Addressing Mode	Assembler Language Syntax	Instruction Format										
EAM:	LD Rd, eam LDB Rbd, eam	<table border="1"> <tr><td>01</td><td>10000</td><td>W</td><td>eam</td><td>Rd</td></tr> <tr><td colspan="5">1, 2, or 3 extension words</td></tr> </table>	01	10000	W	eam	Rd	1, 2, or 3 extension words				
01	10000	W	eam	Rd								
1, 2, or 3 extension words												
LDL RRd, eam	<table border="1"> <tr><td>01</td><td>010100</td><td>eam</td><td>RRd</td></tr> <tr><td colspan="4">1, 2, or 3 extension words</td></tr> </table>	01	010100	eam	RRd	1, 2, or 3 extension words						
01	010100	eam	RRd									
1, 2, or 3 extension words												

Load Memory

Destination Addressing Mode	Assembler Language Syntax	Instruction Format										
IR:	LD @Rd1, Rs LDB @Rd1, Rbs	<table border="1"> <tr><td>00</td><td>10111</td><td>W</td><td>Rd≠0</td><td>Rs</td></tr> </table>	00	10111	W	Rd≠0	Rs					
00	10111	W	Rd≠0	Rs								
LDL @Rd1, RRs	<table border="1"> <tr><td>00</td><td>011101</td><td>Rd≠0</td><td>RRs</td></tr> </table>	00	011101	Rd≠0	RRs							
00	011101	Rd≠0	RRs									
BA:	LD Rd1(disp), Rs LDB Rd1(disp), Rbs	<table border="1"> <tr><td>00</td><td>11001</td><td>W</td><td>Rd≠0</td><td>Rs</td></tr> <tr><td colspan="5">displacement</td></tr> </table>	00	11001	W	Rd≠0	Rs	displacement				
00	11001	W	Rd≠0	Rs								
displacement												
LDL Rd1(disp), RRs	<table border="1"> <tr><td>00</td><td>110111</td><td>Rd≠0</td><td>RRs</td></tr> <tr><td colspan="4">displacement</td></tr> </table>	00	110111	Rd≠0	RRs	displacement						
00	110111	Rd≠0	RRs									
displacement												
BX:	LD Rd1(Rx), Rs LDB Rd1(Rx), Rbs	<table border="1"> <tr><td>01</td><td>11001</td><td>W</td><td>Rd≠0</td><td>Rs</td></tr> <tr><td>0000</td><td>Rx≠0</td><td colspan="3">0000 0000</td></tr> </table>	01	11001	W	Rd≠0	Rs	0000	Rx≠0	0000 0000		
01	11001	W	Rd≠0	Rs								
0000	Rx≠0	0000 0000										
LDL Rd1(Rx), RRs	<table border="1"> <tr><td>01</td><td>110111</td><td>Rd≠0</td><td>RRs</td></tr> <tr><td>0000</td><td>Rx≠0</td><td colspan="3">0000 0000</td></tr> </table>	01	110111	Rd≠0	RRs	0000	Rx≠0	0000 0000				
01	110111	Rd≠0	RRs									
0000	Rx≠0	0000 0000										
EAM:	LD eam, Rs LDB eam, Rbs	<table border="1"> <tr><td>01</td><td>10111</td><td>W</td><td>eam</td><td>Rs</td></tr> <tr><td colspan="5">1, 2, or 3 extension words</td></tr> </table>	01	10111	W	eam	Rs	1, 2, or 3 extension words				
01	10111	W	eam	Rs								
1, 2, or 3 extension words												
LDL eam, RRs	<table border="1"> <tr><td>01</td><td>011101</td><td>eam</td><td>RRs</td></tr> <tr><td colspan="4">1, 2, or 3 extension words</td></tr> </table>	01	011101	eam	RRs	1, 2, or 3 extension words						
01	011101	eam	RRs									
1, 2, or 3 extension words												

Load Immediate Value

Destination Addressing Mode	Assembler Language Syntax	Instruction Format															
R:	LD Rd, #data	<table border="1"> <tr><td>00</td><td>100001</td><td>0000</td><td>Rd</td></tr> <tr><td colspan="4">data</td></tr> </table>	00	100001	0000	Rd	data										
00	100001	0000	Rd														
data																	
LDB Rbd, #data ²	<table border="1"> <tr><td>00</td><td>100000</td><td>0000</td><td>Rbd</td></tr> <tr><td colspan="2">data</td><td colspan="2">data</td></tr> </table>	00	100000	0000	Rbd	data		data									
00	100000	0000	Rbd														
data		data															
	<table border="1"> <tr><td>1100</td><td>Rd</td><td colspan="2">data</td></tr> </table>	1100	Rd	data													
1100	Rd	data															
LDL RRd, #data	<table border="1"> <tr><td>00</td><td>010100</td><td>0000</td><td>RRd</td></tr> <tr><td colspan="2">data (high)</td><td colspan="2"></td></tr> <tr><td colspan="2">data (low)</td><td colspan="2"></td></tr> </table>	00	010100	0000	RRd	data (high)				data (low)							
00	010100	0000	RRd														
data (high)																	
data (low)																	
LD @Rd1, #data	<table border="1"> <tr><td>00</td><td>001101</td><td>Rd≠0</td><td>0101</td></tr> <tr><td colspan="4">data</td></tr> </table>	00	001101	Rd≠0	0101	data											
00	001101	Rd≠0	0101														
data																	
LDB @Rd1, #data	<table border="1"> <tr><td>00</td><td>001100</td><td>Rd≠0</td><td>0101</td></tr> <tr><td colspan="2">data</td><td colspan="2">data</td></tr> </table>	00	001100	Rd≠0	0101	data		data									
00	001100	Rd≠0	0101														
data		data															
LDL @Rd1, #data	<table border="1"> <tr><td>00</td><td>001101</td><td>Rd≠0</td><td>0111</td></tr> <tr><td colspan="2">data (high)</td><td colspan="2"></td></tr> <tr><td colspan="2">data (low)</td><td colspan="2"></td></tr> </table>	00	001101	Rd≠0	0111	data (high)				data (low)							
00	001101	Rd≠0	0111														
data (high)																	
data (low)																	
LD eam, #data	<table border="1"> <tr><td>01</td><td>001101</td><td>eam</td><td>0101</td></tr> <tr><td colspan="4">1, 2, or 3 extension words</td></tr> <tr><td colspan="4">data</td></tr> </table>	01	001101	eam	0101	1, 2, or 3 extension words				data							
01	001101	eam	0101														
1, 2, or 3 extension words																	
data																	
LDB eam, #data	<table border="1"> <tr><td>01</td><td>001100</td><td>eam</td><td>0101</td></tr> <tr><td colspan="4">1, 2, or 3 extension words</td></tr> <tr><td colspan="2">data</td><td colspan="2">data</td></tr> </table>	01	001100	eam	0101	1, 2, or 3 extension words				data		data					
01	001100	eam	0101														
1, 2, or 3 extension words																	
data		data															
LDL eam, #data	<table border="1"> <tr><td>01</td><td>001101</td><td>eam</td><td>0111</td></tr> <tr><td colspan="4">1, 2, or 3 extension words</td></tr> <tr><td colspan="2">data (high)</td><td colspan="2"></td></tr> <tr><td colspan="2">data (low)</td><td colspan="2"></td></tr> </table>	01	001101	eam	0111	1, 2, or 3 extension words				data (high)				data (low)			
01	001101	eam	0111														
1, 2, or 3 extension words																	
data (high)																	
data (low)																	

Example: If register RH0 contains %AB, executing the instruction
LD RL7, RH0
loads %AB into RL7.

Note 1: Word register in compact mode, longword register in segmented or linear modes.

Note 2: As shown, the instruction set includes two formats for loading an immediate value into a byte register. The assembler uses the format with one word.

LDA

Load Address

LDA dst, src

dst: R
src: BA, BX, EAM

Operation: dst ← EFFECTIVE ADDRESS (src)

The effective address of the source operand is calculated and loaded into the destination. The contents of the source are not affected. The address calculation follows the rules for address arithmetic in the current mode of address representation: compact, segmented or linear. The destination is a word register in compact mode, and a longword register in segmented or linear mode.

Flags: No flags affected

Exceptions: None

Source Addressing Mode	Assembler Language Syntax	Instruction Format								
BA:	LDA Rd ¹ , Rs ¹ (disp)	<table border="1"> <tr> <td>0 0 1 1 0 1 0 0</td> <td>Rs ≠ 0</td> <td>Rd</td> </tr> <tr> <td colspan="3">displacement</td> </tr> </table>	0 0 1 1 0 1 0 0	Rs ≠ 0	Rd	displacement				
0 0 1 1 0 1 0 0	Rs ≠ 0	Rd								
displacement										
BX:	LDA Rd ¹ , Rs ¹ (Rx)	<table border="1"> <tr> <td>0 1 1 1 0 1 0 0</td> <td>Rs ≠ 0</td> <td>Rd</td> </tr> <tr> <td>0 0 0 0</td> <td>Rx ≠ 0</td> <td>0 0 0 0 0 0 0 0</td> </tr> </table>	0 1 1 1 0 1 0 0	Rs ≠ 0	Rd	0 0 0 0	Rx ≠ 0	0 0 0 0 0 0 0 0		
0 1 1 1 0 1 0 0	Rs ≠ 0	Rd								
0 0 0 0	Rx ≠ 0	0 0 0 0 0 0 0 0								
EAM:	LDA Rd ¹ , eam	<table border="1"> <tr> <td>0 1</td> <td>1 1 0 1 1 0</td> <td>eam</td> <td>Rd</td> </tr> <tr> <td colspan="4">1, 2, or 3 extension words</td> </tr> </table>	0 1	1 1 0 1 1 0	eam	Rd	1, 2, or 3 extension words			
0 1	1 1 0 1 1 0	eam	Rd							
1, 2, or 3 extension words										

Examples:

LDA R4,STRUCT	//in compact mode, register R4 is loaded //with the compact address of the location //named STRUCT
LDA RR2,RR4(8)	//in linear mode, if base register RR4 //contains %01000020, then register RR2 is loaded //with the address %01000028

Note 1: Word register in compact mode, longword register in segmented or linear modes.

LDAR

Load Address Relative

LDAR dst, src	dst: R src: RA
----------------------	-------------------

Operation: $\text{dst} \leftarrow \text{EFFECTIVE_ADDRESS}(\text{src})$

The effective address of the source operand is calculated and loaded into the destination. The contents of the source are not affected. The destination is a word register in compact mode, and a longword register in segmented or linear mode.

The destination address is calculated by adding the displacement in the instruction to the updated value of the Program Counter (PC). The updated PC value is the address of the instruction word following the LDAR instruction. The displacement is a 16-bit signed value in the range -32768 to 32767 in the second word of the instruction. The addition is performed following the rules of address arithmetic in the current mode of address representation: compact, segmented, or linear.

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

Exceptions: None

Source Addressing Mode	Assembler Language Syntax	Instruction Format						
RA:	LDAR Rd1, address	<table border="1" data-bbox="928 1015 1245 1049"> <tr> <td data-bbox="933 1015 1047 1026">00110100</td> <td data-bbox="1047 1015 1157 1026">0000</td> <td data-bbox="1157 1015 1240 1026">Rd</td> </tr> <tr> <td data-bbox="933 1026 1245 1049" style="text-align: center;">displacement</td><td data-bbox="933 1026 1245 1049"></td><td data-bbox="933 1026 1245 1049"></td></tr> </table>	00110100	0000	Rd	displacement		
00110100	0000	Rd						
displacement								

Example: LDAR RR4, TABLE

//in segmented mode, register RR4 is
//loaded with the segmented address of TABLE

Note 1: Word register in compact mode, longword register in segmented or linear modes.

LDCTL Load Control

Privileged Instruction

LDCTL dst, src

dst: CTRL
src: R
or
dst: R
src: CTRL

Operation: dst ← src

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

FCW	Flag and Control Word
PSAPSEG	Program Status Area Pointer—high word
PSAOFF	Program Status Area Pointer—low word
NSPSEG	Normal Stack Pointer—high word
NSPOFF	Normal Stack Pointer—low word

When the destination register is FCW, the Trace Pending bit (FCW.TP) is set if bit 9 of the source operand is set or if the Trace Enable bit (FCW.T) is set before the instruction is executed. This allows tracing of system programs that may load the FCW mistakenly.

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.

Exceptions: Privileged Instruction trap

Load Into Control Register

	Assembler Language Syntax	Instruction Format			
	LDCTL FCW, Rs	<table><tr><td>01111101</td><td>Rs</td><td>1010</td></tr></table>	01111101	Rs	1010
01111101	Rs	1010			
	LDCTL PSAPSEG, Rs	<table><tr><td>01111101</td><td>Rs</td><td>1100</td></tr></table>	01111101	Rs	1100
01111101	Rs	1100			
	LDCTL PSAOFF, Rs	<table><tr><td>01111101</td><td>Rs</td><td>1101</td></tr></table>	01111101	Rs	1101
01111101	Rs	1101			
	LDCTL NSPSEG, Rs	<table><tr><td>01111101</td><td>Rs</td><td>1110</td></tr></table>	01111101	Rs	1110
01111101	Rs	1110			
	LDCTL NSPOFF, Rs	<table><tr><td>01111101</td><td>Rs</td><td>1111</td></tr></table>	01111101	Rs	1111
01111101	Rs	1111			

Load From Control Register

	Assembler Language Syntax	Instruction Format			
	LDCTL Rd, FCW	<table border="1"><tr><td>01111101</td><td>Rd</td><td>0010</td></tr></table>	01111101	Rd	0010
01111101	Rd	0010			
	LDCTL Rd, PSAPSEG	<table border="1"><tr><td>01111101</td><td>Rd</td><td>0100</td></tr></table>	01111101	Rd	0100
01111101	Rd	0100			
	LDCTL Rd, PSAPOFF	<table border="1"><tr><td>01111101</td><td>Rd</td><td>0101</td></tr></table>	01111101	Rd	0101
01111101	Rd	0101			
	LDCTL Rd, NSPSEG	<table border="1"><tr><td>01111101</td><td>Rd</td><td>0110</td></tr></table>	01111101	Rd	0110
01111101	Rd	0110			
	LDCTL Rd, NSPOFF	<table border="1"><tr><td>01111101</td><td>Rd</td><td>0111</td></tr></table>	01111101	Rd	0111
01111101	Rd	0111			

LDCTLB

Load Control Byte

LDCTLB dst, src	dst: FLAGS src: R or dst: R src: FLAGS
------------------------	--

Operation: dst \leftarrow src

This instruction loads the contents of a general-purpose byte register into the Flags register, or loads the contents of the Flags register into a general-purpose byte register. (The Flags register is the low-order byte of the Flag and Control Word register.) Note that this is not a privileged instruction.

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.

Exceptions: None

	Assembler Language Syntax	Instruction Format			
	LDCTLB FLAGS, Rbs	<table><tr><td>10001100</td><td>Rbs</td><td>1001</td></tr></table>	10001100	Rbs	1001
10001100	Rbs	1001			
	LDCTLB Rbd, FLAGS	<table><tr><td>10001100</td><td>Rbd</td><td>0001</td></tr></table>	10001100	Rbd	0001
10001100	Rbd	0001			

Privileged Instruction

LDCTLL Load Control Longword

LDCTLL dst, src	dst: CTLLR src: R or dst: R src: CTLLR				
Operation:	dst \leftarrow src				
This instruction loads the contents of a general-purpose longword register into a control register, or loads the contents of a control register into a general-purpose longword register. The control register must be one of the following:					
SITTD	System Instruction Translation Table Descriptor				
SDTTD	System Data Translation Table Descriptor				
NITTD	Normal Instruction Translation Table Descriptor				
NDTTD	Normal Data Translation Table Descriptor				
SCCL	System Configuration Control Longword				
OSP	Overflow Stack Pointer				
HICR	Hardware Interface Control Register				
PSAP	Program Status Area Pointer				
NSP	Normal Stack Pointer				
Flags:	No flags affected				
Exceptions:	Privileged Instruction trap				
Load Into Control Register					
	Assembler Language Syntax	Instruction Format			
	LDCTLL SITTD, RRs	<table border="1"><tr><td>10011101</td><td>RRs</td><td>0000</td></tr></table>	10011101	RRs	0000
10011101	RRs	0000			
	LDCTLL SDTTD, RRs	<table border="1"><tr><td>10011101</td><td>RRs</td><td>0001</td></tr></table>	10011101	RRs	0001
10011101	RRs	0001			
	LDCTLL NITTD, RRs	<table border="1"><tr><td>10011101</td><td>RRs</td><td>0010</td></tr></table>	10011101	RRs	0010
10011101	RRs	0010			
	LDCTLL NDTTD, RRs	<table border="1"><tr><td>10011101</td><td>RRs</td><td>0011</td></tr></table>	10011101	RRs	0011
10011101	RRs	0011			
	LDCTLL SCCL, RRs	<table border="1"><tr><td>10011101</td><td>RRs</td><td>0100</td></tr></table>	10011101	RRs	0100
10011101	RRs	0100			
	LDCTLL OSP, RRs	<table border="1"><tr><td>10011101</td><td>RRs</td><td>1110</td></tr></table>	10011101	RRs	1110
10011101	RRs	1110			
	LDCTLL HICR, RRs	<table border="1"><tr><td>10011101</td><td>RRs</td><td>0111</td></tr></table>	10011101	RRs	0111
10011101	RRs	0111			

Load Into Control Register (Continued)

	Assembler Language Syntax	Instruction Format			
	LDCTLL PSAP, RRs	<table border="1"><tr><td>10011101</td><td>RRs</td><td>1100</td></tr></table>	10011101	RRs	1100
10011101	RRs	1100			
	LDCTLL NSP, RRs	<table border="1"><tr><td>10011101</td><td>RRs</td><td>0110</td></tr></table>	10011101	RRs	0110
10011101	RRs	0110			

Load From Control Register

	LDCTLL RRd, SITTD	<table border="1"><tr><td>10011111</td><td>RRd</td><td>0000</td></tr></table>	10011111	RRd	0000
10011111	RRd	0000			
	LDCTLL RRd, SDTTD	<table border="1"><tr><td>10011111</td><td>RRd</td><td>0001</td></tr></table>	10011111	RRd	0001
10011111	RRd	0001			
	LDCTLL RRd, NITTD	<table border="1"><tr><td>10011111</td><td>RRd</td><td>0010</td></tr></table>	10011111	RRd	0010
10011111	RRd	0010			
	LDCTLL RRd, NDTTD	<table border="1"><tr><td>10011111</td><td>RRd</td><td>0011</td></tr></table>	10011111	RRd	0011
10011111	RRd	0011			
	LDCTLL RRd, SCCL	<table border="1"><tr><td>10011111</td><td>RRd</td><td>0100</td></tr></table>	10011111	RRd	0100
10011111	RRd	0100			
	LDCTLL RRd, OSP	<table border="1"><tr><td>10011111</td><td>RRd</td><td>1110</td></tr></table>	10011111	RRd	1110
10011111	RRd	1110			
	LDCTLL RRd, HICR	<table border="1"><tr><td>10011111</td><td>RRd</td><td>0111</td></tr></table>	10011111	RRd	0111
10011111	RRd	0111			
	LDCTLL RRd, PSAP	<table border="1"><tr><td>10011111</td><td>RRd</td><td>1100</td></tr></table>	10011111	RRd	1100
10011111	RRd	1100			
	LDCTLL RRd, NSP	<table border="1"><tr><td>10011111</td><td>RRd</td><td>0110</td></tr></table>	10011111	RRd	0110
10011111	RRd	0110			

LDD

Load and Decrement

LDD dst, src, r LDDB LDL	dst: IR src: IR																																																	
Operation:	$dst \leftarrow src$ AUTODECREMENT dst and src (by 1 if LDDB; by 2 if LDD; by 4 if LDL) $r \leftarrow r - 1$																																																	
	<p>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, by two if LDD or by four if LDL, 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 source destination, and counter registers must be distinct and non-overlapping registers.</p> <p>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 correctly copied including the overlapping area. However, the destination address must not exceed the source address by one for LDD, and by one, two, or three for LDL; otherwise, the CPU may not recover correctly from address translation exceptions.</p>																																																	
Flags:	C: Unaffected Z: Unaffected S: Unaffected V: Set if the result of decrementing r is zero; cleared otherwise D: Unaffected H: Unaffected																																																	
Exceptions:	None																																																	
Addressing Mode	Assembler Language Syntax	Instruction Format																																																
IR:	LDD @Rs ¹ , @Rd ¹ , r LDDB @Rs ¹ , @Rd ¹ , r LDL @Rs ¹ , @Rd ¹ , r	<table border="1" style="margin-bottom: 10px; border-collapse: collapse;"> <tr> <td style="padding: 2px;">1</td><td style="padding: 2px;">0</td><td style="padding: 2px;">1</td><td style="padding: 2px;">1</td><td style="padding: 2px;">1</td><td style="padding: 2px;">0</td><td style="padding: 2px;">1</td><td style="padding: 2px;">W</td><td style="padding: 2px;">Rs ≠ 0</td><td style="padding: 2px;">1</td><td style="padding: 2px;">0</td><td style="padding: 2px;">1</td> </tr> <tr> <td style="padding: 2px;">0</td><td style="padding: 2px;">r</td><td style="padding: 2px;">Rd ≠ 0</td><td style="padding: 2px;">1</td><td style="padding: 2px;">0</td><td style="padding: 2px;">0</td> </tr> </table> <table border="1" style="border-collapse: collapse;"> <tr> <td style="padding: 2px;">1</td><td style="padding: 2px;">0</td><td style="padding: 2px;">1</td><td style="padding: 2px;">1</td><td style="padding: 2px;">1</td><td style="padding: 2px;">0</td><td style="padding: 2px;">0</td><td style="padding: 2px;">1</td><td style="padding: 2px;">Rs ≠ 0</td><td style="padding: 2px;">1</td><td style="padding: 2px;">0</td><td style="padding: 2px;">1</td> </tr> <tr> <td style="padding: 2px;">0</td><td style="padding: 2px;">r</td><td style="padding: 2px;">Rd ≠ 0</td><td style="padding: 2px;">1</td><td style="padding: 2px;">0</td><td style="padding: 2px;">0</td> </tr> </table>	1	0	1	1	1	0	1	W	Rs ≠ 0	1	0	1	0	0	0	0	0	0	0	r	Rd ≠ 0	1	0	0	1	0	1	1	1	0	0	1	Rs ≠ 0	1	0	1	0	0	0	0	0	0	0	r	Rd ≠ 0	1	0	0
1	0	1	1	1	0	1	W	Rs ≠ 0	1	0	1																																							
0	0	0	0	0	0	0	r	Rd ≠ 0	1	0	0																																							
1	0	1	1	1	0	0	1	Rs ≠ 0	1	0	1																																							
0	0	0	0	0	0	0	r	Rd ≠ 0	1	0	0																																							

Example: In linear mode, if register RR20 contains %0000202A, register RR22 contains %0000404A, the word at location %0000404A contains %FFFF, and register R3 contains 5, executing the instruction

LDI @RR20, @RR22, R3

leaves the value %FFFF at location %0000202A, the value %00002028 in RR20, the value %00004048 in RR22, and the value 4 in R3. The V flag is cleared. In compact mode, word registers must be used instead of RR20 and RR22.

Note 1: Word register in compact mode, longword register in segmented or linear modes.

LDDR

Load, Decrement and Repeat

LDDR dst, src, r LDDRB LDDRL	dst: IR src: IR																							
<hr/>																								
Operation: repeat dst \leftarrow src AUTODECREMENT dst and src (by 1 if LDDRB; by 2 if LDDR; by 4 if LDDRL) r \leftarrow r - 1 until r = 0																								
<p>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, by two if LDDR, or by four if LDDRL, thus moving the pointers to the previous elements in the strings. The word register specified by "r" (used as a counter) is then decremented by one. The entire operation is repeated until the result of decrementing r is zero. This instruction can move from 1 to 65,536 data elements. The source, destination, and counter registers must be distinct and non-overlapping registers.</p> <p>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 correctly copied including the overlapping area. However, the destination address must not exceed the source address by one for LDDR, and by one, two, or three for LDDRL; otherwise, the CPU may not recover correctly from address translation exceptions.</p> <p>This instruction can be interrupted after each execution of the basic operation.</p>																								
Flags: C: Unaffected Z: Unaffected S: Unaffected V: Set D: Unaffected H: Unaffected																								
Exceptions: None																								
<hr/> <table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th style="width: 25%;">Addressing Mode</th> <th style="width: 25%;">Assembler Language Syntax</th> <th style="width: 50%;">Instruction Format</th> </tr> </thead> <tbody> <tr> <td style="text-align: center;">IR:</td> <td> LDDR @Rd¹, @Rs¹, r LDDRB @Rd¹, @Rs¹, r LDDRL @Rd¹, @Rs¹, r </td> <td style="text-align: center;"> <table border="1" style="margin: auto;"> <tr> <td>1011101</td> <td>W</td> <td>Rs ≠ 0</td> <td>1001</td> </tr> <tr> <td>0000</td> <td>r</td> <td>Rd ≠ 0</td> <td>0000</td> </tr> </table> <table border="1" style="margin: auto;"> <tr> <td>10111001</td> <td>Rs ≠ 0</td> <td>1001</td> </tr> <tr> <td>0000</td> <td>r</td> <td>Rd ≠ 0</td> <tr> <td colspan="3">0000</td> </tr> </tr></table> </td> </tr> </tbody> </table> <hr/>		Addressing Mode	Assembler Language Syntax	Instruction Format	IR:	LDDR @Rd ¹ , @Rs ¹ , r LDDRB @Rd ¹ , @Rs ¹ , r LDDRL @Rd ¹ , @Rs ¹ , r	<table border="1" style="margin: auto;"> <tr> <td>1011101</td> <td>W</td> <td>Rs ≠ 0</td> <td>1001</td> </tr> <tr> <td>0000</td> <td>r</td> <td>Rd ≠ 0</td> <td>0000</td> </tr> </table> <table border="1" style="margin: auto;"> <tr> <td>10111001</td> <td>Rs ≠ 0</td> <td>1001</td> </tr> <tr> <td>0000</td> <td>r</td> <td>Rd ≠ 0</td> <tr> <td colspan="3">0000</td> </tr> </tr></table>	1011101	W	Rs ≠ 0	1001	0000	r	Rd ≠ 0	0000	10111001	Rs ≠ 0	1001	0000	r	Rd ≠ 0	0000		
Addressing Mode	Assembler Language Syntax	Instruction Format																						
IR:	LDDR @Rd ¹ , @Rs ¹ , r LDDRB @Rd ¹ , @Rs ¹ , r LDDRL @Rd ¹ , @Rs ¹ , r	<table border="1" style="margin: auto;"> <tr> <td>1011101</td> <td>W</td> <td>Rs ≠ 0</td> <td>1001</td> </tr> <tr> <td>0000</td> <td>r</td> <td>Rd ≠ 0</td> <td>0000</td> </tr> </table> <table border="1" style="margin: auto;"> <tr> <td>10111001</td> <td>Rs ≠ 0</td> <td>1001</td> </tr> <tr> <td>0000</td> <td>r</td> <td>Rd ≠ 0</td> <tr> <td colspan="3">0000</td> </tr> </tr></table>	1011101	W	Rs ≠ 0	1001	0000	r	Rd ≠ 0	0000	10111001	Rs ≠ 0	1001	0000	r	Rd ≠ 0	0000							
1011101	W	Rs ≠ 0	1001																					
0000	r	Rd ≠ 0	0000																					
10111001	Rs ≠ 0	1001																						
0000	r	Rd ≠ 0	0000																					
0000																								

LDI

Load and Increment

LDI dst, src, r LDIB LDIL	dst: IR src: IR																																																	
Operation:	dst \leftarrow src AUTOINCREMENT dst and src (by 1 if LDIB; by 2 if LDI; by 4 if LDIL) $r \leftarrow r - 1$																																																	
<p>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, by two if LDI, or by four if LDIL, 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 source, destination, and counter registers must be distinct, non-overlapping registers.</p> <p>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 correctly copied including the overlapping area. However, the destination address must not exceed the source address by one for LDI; and by one, two, or three for LDIL; otherwise, the CPU may not recover correctly from address translation exceptions.</p>																																																		
Flags: C: Unaffected Z: Unaffected S: Unaffected V: Set if the result of decrementing r is zero, cleared otherwise D: Unaffected H: Unaffected																																																		
Exceptions:	None																																																	
Addressing Mode	Assembler Language Syntax	Instruction Format																																																
IR:	LDI @Rd ¹ , @Rs ¹ , r LDIB @Rd ¹ , @Rs ¹ , r LDIL @Rd ¹ , @Rs ¹ , r	<table border="1" style="margin-left: auto; margin-right: auto; border-collapse: collapse;"> <tr> <td style="padding: 2px;">1</td><td style="padding: 2px;">0</td><td style="padding: 2px;">1</td><td style="padding: 2px;">1</td><td style="padding: 2px;">0</td><td style="padding: 2px;">1</td><td style="padding: 2px; text-align: center;">W</td><td style="padding: 2px;">R</td><td style="padding: 2px;">s ≠ 0</td><td style="padding: 2px;">0</td><td style="padding: 2px;">0</td><td style="padding: 2px;">0</td> </tr> <tr> <td style="padding: 2px;">0</td><td style="padding: 2px;">0</td><td style="padding: 2px;">0</td><td style="padding: 2px;">0</td><td style="padding: 2px;">r</td><td style="padding: 2px;"></td><td style="padding: 2px;">R</td><td style="padding: 2px;">d ≠ 0</td><td style="padding: 2px;">1</td><td style="padding: 2px;">0</td><td style="padding: 2px;">0</td><td style="padding: 2px;">0</td> </tr> </table> <table border="1" style="margin-left: auto; margin-right: auto; border-collapse: collapse;"> <tr> <td style="padding: 2px;">1</td><td style="padding: 2px;">0</td><td style="padding: 2px;">1</td><td style="padding: 2px;">1</td><td style="padding: 2px;">0</td><td style="padding: 2px;">0</td><td style="padding: 2px; text-align: center;">R</td><td style="padding: 2px;">s ≠ 0</td><td style="padding: 2px;">0</td><td style="padding: 2px;">0</td><td style="padding: 2px;">0</td><td style="padding: 2px;">1</td> </tr> <tr> <td style="padding: 2px;">0</td><td style="padding: 2px;">0</td><td style="padding: 2px;">0</td><td style="padding: 2px;">0</td><td style="padding: 2px;">r</td><td style="padding: 2px;"></td><td style="padding: 2px;">R</td><td style="padding: 2px;">d ≠ 0</td><td style="padding: 2px;">1</td><td style="padding: 2px;">0</td><td style="padding: 2px;">0</td><td style="padding: 2px;">0</td> </tr> </table>	1	0	1	1	0	1	W	R	s ≠ 0	0	0	0	0	0	0	0	r		R	d ≠ 0	1	0	0	0	1	0	1	1	0	0	R	s ≠ 0	0	0	0	1	0	0	0	0	r		R	d ≠ 0	1	0	0	0
1	0	1	1	0	1	W	R	s ≠ 0	0	0	0																																							
0	0	0	0	r		R	d ≠ 0	1	0	0	0																																							
1	0	1	1	0	0	R	s ≠ 0	0	0	0	1																																							
0	0	0	0	r		R	d ≠ 0	1	0	0	0																																							

Example: This instruction can be used in a "loop" of instructions which transfers a string of data from one location to another, but where 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 compact mode. In segmented or linear mode, longword registers must 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 compact mode, longword register in segmented or linear modes.

LDIR

Load, Increment and Repeat

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

Operation: repeat
 $dst \leftarrow src$
 AUTOINCREMENT dst and src (by 1 if LDIRB; by 2 if LDIR; by 4 if LDIRL)
 $r \leftarrow r - 1$
 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 LDI, or by four if LDIL, thus moving the pointers to the next elements in the strings. The word register specified by "r" (used as a counter) is then decremented by one. The entire operation is repeated until the result of decrementing r is zero. This instruction can move from 1 to 65,536 data elements. The source, destination, and counter registers must be distinct, non-overlapping registers.

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 correctly copied including the overlapping area. However, the destination address must not exceed the source address by one for LDIR, and by one, two, or three for LDIL; otherwise, the CPU may not recover correctly from address translation exceptions.

This instruction can be interrupted after each execution of the basic operation.

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

Exceptions: None

Addressing Mode	Assembler Language Syntax	Instruction Format								
IR:	LDIR @Rd ¹ , @Rs ¹ , r LDIRB @Rd ¹ , @Rs ¹ , r	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>1011101</td><td>W</td><td>Rs ≠ 0</td><td>0001</td></tr> <tr> <td>0000</td><td>r</td><td>Rd ≠ 0</td><td>0000</td></tr> </table>	1011101	W	Rs ≠ 0	0001	0000	r	Rd ≠ 0	0000
1011101	W	Rs ≠ 0	0001							
0000	r	Rd ≠ 0	0000							
LDIRL @Rd ¹ , @Rs ¹ , r	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>10111001</td><td>Rs ≠ 0</td><td>0001</td></tr> <tr> <td>0000</td><td>r</td><td>Rd ≠ 0</td><td>0000</td></tr> </table>	10111001	Rs ≠ 0	0001	0000	r	Rd ≠ 0	0000		
10111001	Rs ≠ 0	0001								
0000	r	Rd ≠ 0	0000							

Example: The following sequence of instructions can be used in compact 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 or linear mode, longword registers must be used instead of R1 and R2.

Note 1: Word register in compact mode, longword register in segmented or linear modes.

LDK

Load Constant

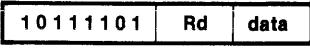
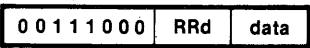
LDK dst, src LDKL	dst: R src: IM
----------------------	-------------------

Operation: dst \leftarrow src (src = 0 to 15)

The source operand, a value from 0 to 15, is loaded into the destination register.

Flags: No flags affected

Exceptions: None

Destination Addressing Mode	Assembler Language Syntax	Instruction Format
R:	LDK Rd, #data	
	LDKL RRd, #data	

Example: To load register R3 with the constant 9, execute the instruction

LDK R3,#9

LDM Load Multiple

LDM dst, src, n	dst: R src: IR, EAM or dst: IR, EAM src: R
LDM dst, src	dst: R src: IM

Operation: $\text{dst} \leftarrow \text{src}(n \text{ words})$

The contents of n (a value from 1 to 16) consecutive source words are loaded into the destination. The contents of the source are not affected. The instruction can be used to load multiple word registers either into or from memory. Registers are accessed in increasing order starting with the specified register; R0 follows R15.

The value in the instruction field for the number of words loaded ("n") is one less than the actual number of words. Thus, the coding in the instruction field ranges from 0 to 15, which corresponds to loading 1 to 16 words.

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

Flags: No flags affected

Exceptions: None

Load Multiple—Registers From Memory

Source Addressing Mode	Assembler Language Syntax	Instruction Format												
IM:	LDM Rd, #data ₀ , #data ₁ , ..., #data _{n-1}	<table border="1"> <tr> <td>00</td><td>011100</td><td>0000</td><td>0001</td></tr> <tr> <td>0000</td><td>Rd</td><td>0000</td><td>n - 1</td></tr> <tr> <td colspan="4">n words data</td></tr> </table>	00	011100	0000	0001	0000	Rd	0000	n - 1	n words data			
00	011100	0000	0001											
0000	Rd	0000	n - 1											
n words data														
IR:	LDM Rd, @Rs ¹ , #n	<table border="1"> <tr> <td>00</td><td>011100</td><td>Rs ≠ 0</td><td>0001</td></tr> <tr> <td>0000</td><td>Rd</td><td>0000</td><td>n - 1</td></tr> </table>	00	011100	Rs ≠ 0	0001	0000	Rd	0000	n - 1				
00	011100	Rs ≠ 0	0001											
0000	Rd	0000	n - 1											
EAM:	LDM Rd, eam, #n	<table border="1"> <tr> <td>01</td><td>011100</td><td>eam</td><td>0001</td></tr> <tr> <td>0000</td><td>Rd</td><td>0000</td><td>n - 1</td></tr> <tr> <td colspan="4">1, 2, or 3 extension words</td></tr> </table>	01	011100	eam	0001	0000	Rd	0000	n - 1	1, 2, or 3 extension words			
01	011100	eam	0001											
0000	Rd	0000	n - 1											
1, 2, or 3 extension words														

Load Multiple—Memory From Registers

Destination Addressing Mode	Assembler Language Syntax	Instruction Format								
IR:	LDM @Rd ¹ , Rs, #n	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>00</td> <td>011100</td> <td>Rd ≠ 0</td> <td>1001</td> </tr> <tr> <td>0000</td> <td>Rs</td> <td>0000</td> <td>n-1</td> </tr> </table>	00	011100	Rd ≠ 0	1001	0000	Rs	0000	n-1
00	011100	Rd ≠ 0	1001							
0000	Rs	0000	n-1							
EAM:	LDM eam, Rs, #n	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>01</td> <td>011100</td> <td>eam</td> <td>1001</td> </tr> <tr> <td>0000</td> <td>Rs</td> <td>0000</td> <td>n-1</td> </tr> </table> <p>1, 2, or 3 extension words</p>	01	011100	eam	1001	0000	Rs	0000	n-1
01	011100	eam	1001							
0000	Rs	0000	n-1							

Example: In compact mode, if register R5 contains 5, R6 contains %0100, and R7 contains 7, executing the instruction

LDM @R6, R5, #3

leaves the values 5, %0100, and 7 at word locations %0100, %0102, and %0104, respectively; none of the registers is affected. In segmented or linear mode, a longword register must be used instead of R6.

Note 1: Word register in compact mode, longword register in segmented or linear modes.

LDML

Load Multiple Longwords

LDML mask, src	src: IM, IR, EAM mask: IM or
LDML dst, mask	dst: IR, EAM mask: IM

Operation: Load Multiple Longwords—Registers from Memory

```

tsrc ← EFFECTIVE_ADDRESS (src)
for i = 0 to 7 do
    if mask<i> = 1 then
        RR [2 × i + 16] ← @tsrc
        tsrc ← tsrc + 4
for i = 8 to 15 do
    if mask<i> = 1 then
        RR [2 × i - 16] ← @tsrc
        tsrc ← tsrc + 4

```

Load Multiple Longwords—Memory from Registers

```

dst ← EFFECTIVE_ADDRESS (dst)
for i = 0 to 7 do
    if mask<i> = 1 then
        @tdst ← RR [2 × i + 16]
        tdst ← tdst + 4
for i = 8 to 15 do
    if mask<i> = 1 then
        @tdst ← RR [2 × i - 16];
        tdst ← tdst + 4

```

This instruction can be used to load multiple longword registers either into or from memory. Each bit in the mask operand that is set to 1 corresponds to a longword register to be loaded. Bits 0 to 7 of the mask operand designate the longword registers RR16 to RR30 respectively. Bits 8 to 15 of the mask operand designate the longword registers RR0 to RR14 respectively. The format of the mask operand is shown in Figure 6-4.

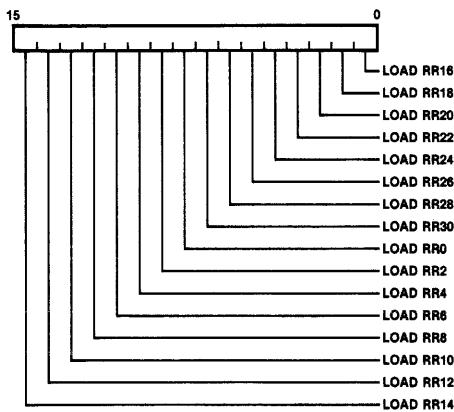


Figure 6-4. Mask Operand Format

The starting memory address is calculated once at the start of execution and incremented by four for each register loaded. If the original address calculation involved a register, the register's value is not affected by incrementing the address during execution. Similarly, modifying that register during a load from memory does not affect the address used by this instruction.

Flags: No flags affected

Exceptions: None

Load Multiple Longwords—Registers From Memory

Source Addressing Mode	Assembler Language Syntax	Instruction Format												
IM:	LDML #mask, #data ₀ , #data ₁ , ..., #data _{n - 1}	<table border="1"> <tr><td>00</td><td>011100</td><td>0000</td><td>0101</td></tr> <tr><td colspan="4">mask</td></tr> <tr><td colspan="4">n longwords data</td></tr> </table>	00	011100	0000	0101	mask				n longwords data			
00	011100	0000	0101											
mask														
n longwords data														
IR:	LDML #mask, @Rs ¹	<table border="1"> <tr><td>00</td><td>011100</td><td>Rs ≠ 0</td><td>0101</td></tr> <tr><td colspan="4">mask</td></tr> </table>	00	011100	Rs ≠ 0	0101	mask							
00	011100	Rs ≠ 0	0101											
mask														
EAM:	LDML, #mask, eam	<table border="1"> <tr><td>01</td><td>011100</td><td>eam</td><td>0101</td></tr> <tr><td colspan="4">mask</td></tr> <tr><td colspan="4">1, 2, or 3 extension words</td></tr> </table>	01	011100	eam	0101	mask				1, 2, or 3 extension words			
01	011100	eam	0101											
mask														
1, 2, or 3 extension words														

Load Multiple Longwords—Memory from Registers

Destination Addressing Mode	Assembler Language Syntax	Instruction Format												
IR:	LDML @Rd ¹ , #mask	<table border="1"> <tr><td>00</td><td>011100</td><td>Rd ≠ 0</td><td>1101</td></tr> <tr><td colspan="4">mask</td></tr> </table>	00	011100	Rd ≠ 0	1101	mask							
00	011100	Rd ≠ 0	1101											
mask														
EAM:	LDML eam, #mask	<table border="1"> <tr><td>01</td><td>011100</td><td>eam</td><td>1101</td></tr> <tr><td colspan="4">mask</td></tr> <tr><td colspan="4">1, 2, or 3 extension words</td></tr> </table>	01	011100	eam	1101	mask				1, 2, or 3 extension words			
01	011100	eam	1101											
mask														
1, 2, or 3 extension words														

Example: In linear mode, if base register RR2 contains %1000 and the longwords at location %1000 and %1002 contain 100 and 150 respectively, executing the instruction
 LDML, #5, @RR2
 loads 100 into RR16 and 150 into RR20.

Note 1: Word register in compact mode, longword register in segmented or linear modes.

LDN

Load Normal

Privileged Instruction

LDND dst, src, n	dst: R
LDNDB	src: IR, EAM
LDNDL	or
LDNI	dst: IR, EAM
LDNIB	src: R
LDNIL	

Operation: dst ← src

These instructions allow programs executing in system mode to reference information in normal mode data and instruction memory address spaces. This is useful for accessing system call parameters when system and normal mode address spaces are separated. The LDND instructions reference normal data space and the LDNI instructions reference normal instruction space. There are versions of the instructions to load from memory to a register and from a register to memory. When performing the memory reference, the address translation mechanism uses the translation tables for normal data or instruction space, and checks the access permission for system mode.

Flags: No flags affected

Exceptions: Privileged Instruction trap

Load Register from Normal Space

Source Addressing Mode	Assembler Language Syntax	Instruction Format						
IR:	LDND Rd, @Rs ¹ LDNDB Rbd, @Rs ¹	<table border="1"> <tr> <td>01111010</td> <td>0011</td> <td>0111</td> </tr> <tr> <td>0010000W</td> <td>Rs≠0</td> <td>Rd</td> </tr> </table>	01111010	0011	0111	0010000W	Rs≠0	Rd
01111010	0011	0111						
0010000W	Rs≠0	Rd						
LDNDL RRd, @Rs ¹	<table border="1"> <tr> <td>01111010</td> <td>0011</td> <td>0111</td> </tr> <tr> <td>00010100</td> <td>Rs≠0</td> <td>RRd</td> </tr> </table>	01111010	0011	0111	00010100	Rs≠0	RRd	
01111010	0011	0111						
00010100	Rs≠0	RRd						
LDNI Rd, @Rs ¹ LDNIB Rbd, @Rs ¹	<table border="1"> <tr> <td>01111010</td> <td>0010</td> <td>0111</td> </tr> <tr> <td>0010000W</td> <td>Rs≠0</td> <td>Rd</td> </tr> </table>	01111010	0010	0111	0010000W	Rs≠0	Rd	
01111010	0010	0111						
0010000W	Rs≠0	Rd						
LDNIL RRd, @Rs ¹	<table border="1"> <tr> <td>01111010</td> <td>0010</td> <td>0111</td> </tr> <tr> <td>00010100</td> <td>Rs≠0</td> <td>RRd</td> </tr> </table>	01111010	0010	0111	00010100	Rs≠0	RRd	
01111010	0010	0111						
00010100	Rs≠0	RRd						

Load Register from Normal Space (Continued)

Source Addressing Mode	Assembler Language Syntax	Instruction Format												
EAM:	LDND Rd, eam LDNDB Rbd, eam	<table border="1"> <tr><td>0 1 1 1 1 0 1 0</td><td>0 0 1 1</td><td>0 1 1 1</td></tr> <tr><td>0 1 1 0 0 0 0</td><td>W</td><td>eam</td></tr> <tr><td colspan="3">Rd</td></tr> <tr><td colspan="3">1, 2, or 3 extension words</td></tr> </table>	0 1 1 1 1 0 1 0	0 0 1 1	0 1 1 1	0 1 1 0 0 0 0	W	eam	Rd			1, 2, or 3 extension words		
0 1 1 1 1 0 1 0	0 0 1 1	0 1 1 1												
0 1 1 0 0 0 0	W	eam												
Rd														
1, 2, or 3 extension words														
LDNDL RRd, eam	<table border="1"> <tr><td>0 1 1 1 1 0 1 0</td><td>0 0 1 1</td><td>0 1 1 1</td></tr> <tr><td>0 1 0 1 0 1 0</td><td>0</td><td>eam</td></tr> <tr><td colspan="3">RRd</td></tr> <tr><td colspan="3">1, 2, or 3 extension words</td></tr> </table>	0 1 1 1 1 0 1 0	0 0 1 1	0 1 1 1	0 1 0 1 0 1 0	0	eam	RRd			1, 2, or 3 extension words			
0 1 1 1 1 0 1 0	0 0 1 1	0 1 1 1												
0 1 0 1 0 1 0	0	eam												
RRd														
1, 2, or 3 extension words														
LDNI Rd, eam LDNIB Rbd, eam	<table border="1"> <tr><td>0 1 1 1 1 0 1 0</td><td>0 0 1 0</td><td>0 1 1 1</td></tr> <tr><td>0 1 1 0 0 0 0</td><td>W</td><td>eam</td></tr> <tr><td colspan="3">Rd</td></tr> <tr><td colspan="3">1, 2, or 3 extension words</td></tr> </table>	0 1 1 1 1 0 1 0	0 0 1 0	0 1 1 1	0 1 1 0 0 0 0	W	eam	Rd			1, 2, or 3 extension words			
0 1 1 1 1 0 1 0	0 0 1 0	0 1 1 1												
0 1 1 0 0 0 0	W	eam												
Rd														
1, 2, or 3 extension words														
LDNIL RRd, eam	<table border="1"> <tr><td>0 1 1 1 1 0 1 0</td><td>0 0 1 0</td><td>0 1 1 1</td></tr> <tr><td>0 1 0 1 0 1 0</td><td>0</td><td>eam</td></tr> <tr><td colspan="3">RRd</td></tr> <tr><td colspan="3">1, 2, or 3 extension words</td></tr> </table>	0 1 1 1 1 0 1 0	0 0 1 0	0 1 1 1	0 1 0 1 0 1 0	0	eam	RRd			1, 2, or 3 extension words			
0 1 1 1 1 0 1 0	0 0 1 0	0 1 1 1												
0 1 0 1 0 1 0	0	eam												
RRd														
1, 2, or 3 extension words														

Load Normal Space from Register

Destination Addressing Mode	Assembler Language Syntax	Instruction Format									
IR:	LDND @Rd1, Rs LDNDB @Rd, Rbs	<table border="1"> <tr><td>0 1 1 1 1 0 1 0</td><td>0 0 1 1</td><td>0 1 1 1</td></tr> <tr><td>0 0 1 0 1 1 1</td><td>W</td><td>Rd ≠ 0</td></tr> <tr><td colspan="3">Rs</td></tr> </table>	0 1 1 1 1 0 1 0	0 0 1 1	0 1 1 1	0 0 1 0 1 1 1	W	Rd ≠ 0	Rs		
0 1 1 1 1 0 1 0	0 0 1 1	0 1 1 1									
0 0 1 0 1 1 1	W	Rd ≠ 0									
Rs											
LDNDL @Rd1, RRs	<table border="1"> <tr><td>0 1 1 1 1 0 1 0</td><td>0 0 1 1</td><td>0 1 1 1</td></tr> <tr><td>0 0 0 1 1 1 0</td><td>1</td><td>Rd ≠ 0</td></tr> <tr><td colspan="3">Rs</td></tr> </table>	0 1 1 1 1 0 1 0	0 0 1 1	0 1 1 1	0 0 0 1 1 1 0	1	Rd ≠ 0	Rs			
0 1 1 1 1 0 1 0	0 0 1 1	0 1 1 1									
0 0 0 1 1 1 0	1	Rd ≠ 0									
Rs											
LDNI @Rd1, Rs LDNIB @Rd1, Rbs	<table border="1"> <tr><td>0 1 1 1 1 0 1 0</td><td>0 0 1 0</td><td>0 1 1 1</td></tr> <tr><td>0 0 1 0 1 1 1</td><td>W</td><td>Rd ≠ 0</td></tr> <tr><td colspan="3">Rs</td></tr> </table>	0 1 1 1 1 0 1 0	0 0 1 0	0 1 1 1	0 0 1 0 1 1 1	W	Rd ≠ 0	Rs			
0 1 1 1 1 0 1 0	0 0 1 0	0 1 1 1									
0 0 1 0 1 1 1	W	Rd ≠ 0									
Rs											
LDNIL @Rd1, RRs	<table border="1"> <tr><td>0 1 1 1 1 0 1 0</td><td>0 0 1 0</td><td>0 1 1 1</td></tr> <tr><td>0 0 0 1 1 1 0</td><td>1</td><td>Rd ≠ 0</td></tr> <tr><td colspan="3">Rs</td></tr> </table>	0 1 1 1 1 0 1 0	0 0 1 0	0 1 1 1	0 0 0 1 1 1 0	1	Rd ≠ 0	Rs			
0 1 1 1 1 0 1 0	0 0 1 0	0 1 1 1									
0 0 0 1 1 1 0	1	Rd ≠ 0									
Rs											

Load Normal Space from Register (Continued)

Destination Addressing Mode	Assembler Language Syntax	Instruction Format											
EAM:	LDND eam, Rs LDNDB eam, Rbs	<table border="1" data-bbox="878 451 1188 572"> <tr><td>01111010</td><td>0011</td><td>0111</td></tr> <tr><td>01</td><td>10111W</td><td>eam</td><td>Rs</td></tr> <tr><td colspan="4">1, 2, or 3 extension words</td></tr> </table>	01111010	0011	0111	01	10111W	eam	Rs	1, 2, or 3 extension words			
01111010	0011	0111											
01	10111W	eam	Rs										
1, 2, or 3 extension words													
LDNDL eam, RRs	<table border="1" data-bbox="878 606 1188 726"> <tr><td>01111010</td><td>0011</td><td>0111</td></tr> <tr><td>01</td><td>011101</td><td>eam</td><td>RRs</td></tr> <tr><td colspan="4">1, 2, or 3 extension words</td></tr> </table>	01111010	0011	0111	01	011101	eam	RRs	1, 2, or 3 extension words				
01111010	0011	0111											
01	011101	eam	RRs										
1, 2, or 3 extension words													
LDNI eam, Rs LDNIB eam, Rbs	<table border="1" data-bbox="878 768 1188 889"> <tr><td>01111010</td><td>0010</td><td>0111</td></tr> <tr><td>01</td><td>10111W</td><td>eam</td><td>Rs</td></tr> <tr><td colspan="4">1, 2, or 3 extension words</td></tr> </table>	01111010	0010	0111	01	10111W	eam	Rs	1, 2, or 3 extension words				
01111010	0010	0111											
01	10111W	eam	Rs										
1, 2, or 3 extension words													
LDNIL eam, RRs	<table border="1" data-bbox="878 910 1188 1030"> <tr><td>01111010</td><td>0010</td><td>0111</td></tr> <tr><td>01</td><td>011101</td><td>eam</td><td>RRs</td></tr> <tr><td colspan="4">1, 2, or 3 extension words</td></tr> </table>	01111010	0010	0111	01	011101	eam	RRs	1, 2, or 3 extension words				
01111010	0010	0111											
01	011101	eam	RRs										
1, 2, or 3 extension words													

Note 1: Word register in compact mode, longword register in segmented or linear modes.

Privileged Instruction

LDP Load Physical Address

LDPND dst, src	dst: R
LDPNI	src: IR, EAM
LDPSI	
LDPSD	

Operation: dst ← PHYSICAL_ADDRESS (src)

These instructions translate the logical address of the source operand to a physical address, and store the result into the destination. Four versions of the instruction are provided, one for each of the logical memory address spaces: normal mode instruction space (LDPNI), normal mode data space (LDPND), system mode instruction space (LDPSI), and system mode data space (LDPSD). The Z flag is set when the translation is valid, and cleared otherwise.

The V and C flag settings indicate whether or not read and write accesses are permitted to the source byte address. This feature is useful for verifying access rights for addresses passed as system call parameters from normal to system mode. The S flag is set when the access information reported in the V and C flags is valid, and cleared otherwise. (During address translation, the PROT field specifying the access rights may be valid although one of the translation table entries is invalid.) When address translation is disabled, read and write accesses are permitted to all addresses.

Flags:	C: LDPND, LDPNI—set if write access is permitted for the source operand in normal mode; cleared otherwise; LDPSI, LDPSD—set if write access is permitted for the source operand in system mode; cleared otherwise Z: Set if the translation is valid; cleared otherwise S: Set if the protection information in flags C and V is valid; cleared otherwise V: LDPND, LDPNI—set if read access is permitted for the source operand in normal mode; cleared otherwise; LDPSI, LDPSD—set if read access is permitted for the source operand in system mode; cleared otherwise D: Unaffected H: Unaffected
---------------	--

Exceptions: Privileged Instruction trap

Source Addressing Mode	Assembler Language Syntax	Instruction Format						
IR:	LDPND RRd, @Rs ¹	<table border="1"> <tr> <td>01111010</td><td>0011</td><td>1101</td></tr> <tr> <td>00</td><td>110110</td><td>Rs≠0 RRd</td></tr> </table>	01111010	0011	1101	00	110110	Rs≠0 RRd
01111010	0011	1101						
00	110110	Rs≠0 RRd						
LDPNI RRd, @Rs ¹	<table border="1"> <tr> <td>01111010</td><td>0010</td><td>1101</td></tr> <tr> <td>00</td><td>110110</td><td>Rs≠0 RRd</td></tr> </table>	01111010	0010	1101	00	110110	Rs≠0 RRd	
01111010	0010	1101						
00	110110	Rs≠0 RRd						
LDPSD RRd, @Rs ¹	<table border="1"> <tr> <td>01111010</td><td>0001</td><td>1101</td></tr> <tr> <td>00</td><td>110110</td><td>Rs≠0 RRd</td></tr> </table>	01111010	0001	1101	00	110110	Rs≠0 RRd	
01111010	0001	1101						
00	110110	Rs≠0 RRd						
LDPSI RRd, @Rs ¹	<table border="1"> <tr> <td>01111010</td><td>0000</td><td>1101</td></tr> <tr> <td>00</td><td>110110</td><td>Rs≠0 RRd</td></tr> </table>	01111010	0000	1101	00	110110	Rs≠0 RRd	
01111010	0000	1101						
00	110110	Rs≠0 RRd						

Source Addressing Mode	Assembler Language Syntax	Instruction Format																																																
EAM:	LDPND RRd, eam	<table border="1" data-bbox="882 403 1192 519"> <tr><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td></tr> <tr><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td></tr> <tr><td colspan="12">eam</td></tr> <tr><td colspan="12">RRd</td></tr> </table> <p data-bbox="915 487 1160 508">1, 2, or 3 extension words</p>	0	1	1	1	1	0	1	0	0	1	1	1	0	1	1	1	0	1	1	0	1	1	0	1	eam												RRd											
0	1	1	1	1	0	1	0	0	1	1	1																																							
0	1	1	1	0	1	1	0	1	1	0	1																																							
eam																																																		
RRd																																																		
LDPNI RRd, eam	<table border="1" data-bbox="882 551 1192 667"> <tr><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td></tr> <tr><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td></tr> <tr><td colspan="12">eam</td></tr> <tr><td colspan="12">RRd</td></tr> </table> <p data-bbox="915 635 1160 656">1, 2, or 3 extension words</p>	0	1	1	1	1	0	1	0	0	1	1	0	0	1	1	1	0	1	1	0	1	1	0	1	eam												RRd												
0	1	1	1	1	0	1	0	0	1	1	0																																							
0	1	1	1	0	1	1	0	1	1	0	1																																							
eam																																																		
RRd																																																		
LDPSD RRd, eam	<table border="1" data-bbox="882 699 1192 815"> <tr><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td></tr> <tr><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td></tr> <tr><td colspan="12">eam</td></tr> <tr><td colspan="12">RRd</td></tr> </table> <p data-bbox="915 783 1160 804">1, 2, or 3 extension words</p>	0	1	1	1	1	0	1	0	0	0	1	1	0	1	1	1	0	1	1	0	1	1	0	1	eam												RRd												
0	1	1	1	1	0	1	0	0	0	1	1																																							
0	1	1	1	0	1	1	0	1	1	0	1																																							
eam																																																		
RRd																																																		
LDPSI RRd, eam	<table border="1" data-bbox="882 846 1192 963"> <tr><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td></tr> <tr><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td></tr> <tr><td colspan="12">eam</td></tr> <tr><td colspan="12">RRd</td></tr> </table> <p data-bbox="915 931 1160 952">1, 2, or 3 extension words</p>	0	1	1	1	1	0	1	0	0	0	1	1	0	1	1	1	0	1	1	0	1	1	0	1	eam												RRd												
0	1	1	1	1	0	1	0	0	0	1	1																																							
0	1	1	1	0	1	1	0	1	1	0	1																																							
eam																																																		
RRd																																																		

Note 1: Word register in compact mode, longword register in segmented or linear modes.

Privileged Instruction

LDPS Load Program Status

LDPS src

src: IR, EAM

Operation:

```

tmp ← EFFECTIVE_ADDRESS (src)
if FCW.E/C then           //segmented or linear mode
    tmp 2 ← @(tmp1 + 2)   //fetch FCW
    PC ← @(tmp1 + 4)      //fetch PC (longword)
else
    tmp 2 ← @tmp1        //fetch FCW
    PC ← @(tmp1 + 2)      //fetch PC (low-order word)
if FCW.T then tmp2<9> ← 1
FCW ← tmp2

```

The contents of the source operand are loaded into the Program Status (PS) registers, both the Flag and Control Word (FCW) and the Program Counter (PC). In compact mode the source operand includes two words: the new FCW and the new low-order word of PC. The high-order word of PC is unaffected. In segmented or linear mode, the source operand includes four words: a reserved word (which must contain 0), the new FCW, and the new PC longword.

After LDPS is executed, the Trace Pending bit (FCW.TP) is set if bit 9 is set in the source operand FCW or if the Trace Enable (FCW.T) bit was set before the instruction was executed. This allows the LDPS instruction to be traced for single-step debugging.



Flags: All flags are loaded from the source operand.

Exceptions: Privileged Instruction trap

Source Addressing Mode	Assembler Language Syntax	Instruction Format								
IR:	LDPS @Rs ¹	<table border="1"> <tr> <td>00</td> <td>111001</td> <td>Rs≠0</td> <td>0000</td> </tr> </table>	00	111001	Rs≠0	0000				
00	111001	Rs≠0	0000							
EAM:	LDPS eam	<table border="1"> <tr> <td>01</td> <td>111001</td> <td>eam</td> <td>0000</td> </tr> <tr> <td colspan="4">1, 2, or 3 extension words</td> </tr> </table>	01	111001	eam	0000	1, 2, or 3 extension words			
01	111001	eam	0000							
1, 2, or 3 extension words										

Example: In compact mode, if register R3 contains %5000, location %5000 contains %1800, and location %5002 contains %A000, executing the instruction
LDPS @R3
leaves the value %A000 in the PC, and the FCW value is %1800.

Note 1: Word register in compact mode, longword register in segmented or linear modes.

LDR

Load Relative

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

Operation: $dst \leftarrow src$

The contents of the source operand are loaded into the destination. The contents of the source are not affected. The effective address is calculated by adding the displacement in the instruction to the updated value of the program counter (PC). The updated PC value is the address of the instruction word following the LDR, LDRB, or LDRL instruction. The displacement is a 16-bit signed value in the range -32768 to 32767.

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

Flags: No flags affected

Exceptions: None

Load Relative Register

Destination Addressing Mode	Assembler Language Syntax	Instruction Format								
RA:	LDR Rd, address LDRB Rbd, address	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>0011000</td> <td>W</td> <td>0000</td> <td>Rd</td> </tr> <tr> <td colspan="4" style="text-align: center;">displacement</td> </tr> </table>	0011000	W	0000	Rd	displacement			
0011000	W	0000	Rd							
displacement										
LDRL RRd, address	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>00110101</td> <td>0000</td> <td>RRd</td> </tr> <tr> <td colspan="3" style="text-align: center;">displacement</td> </tr> </table>	00110101	0000	RRd	displacement					
00110101	0000	RRd								
displacement										

Load Relative Memory

Destination Addressing Mode	Assembler Language Syntax	Instruction Format								
RA:	LDR address, Rs LDRB address, Rbs	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>0011001</td> <td>W</td> <td>0000</td> <td>Rs</td> </tr> <tr> <td colspan="4" style="text-align: center;">displacement</td> </tr> </table>	0011001	W	0000	Rs	displacement			
0011001	W	0000	Rs							
displacement										
LDRL address, RRs	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>00110111</td> <td>0000</td> <td>RRs</td> </tr> <tr> <td colspan="3" style="text-align: center;">displacement</td> </tr> </table>	00110111	0000	RRs	displacement					
00110111	0000	RRs								
displacement										

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

MULT

Multiply

MULT dst, src	dst: R
MULTL	src: R, IM, IR, EAM

Operation: Word (dst is longword register, src is word)
 $dst_{<31:0>} \leftarrow dst_{<15:0>} \times src_{<15:0>}$
 Longword (dst is quadword register, src is longword)
 $dst_{<63:0>} \leftarrow dst_{<31:0>} \times src_{<31:0>}$

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 longword register and the source is a word value; for MULTL, the destination is a quadword register and the source is a longword value.

For proper instruction execution, the "dst field" in the MULTL instruction format encoding must specify a valid code for a quadword register. Otherwise, the result is undefined.

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 C flag is set to indicate that the upper half of the destination register is required to represent the result; if the C 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.

Flags:

- C:** MULT—set if product is less than -2^{15} or greater than or equal to 2^{15} ; cleared otherwise; MULTL—set if product is less than -2^{31} or greater than or equal to -2^{31} ; cleared otherwise
- Z:** Set if the result is zero; cleared otherwise
- S:** Set if the result is negative; cleared otherwise
- V:** Cleared
- D:** Unaffected
- H:** Unaffected

Exceptions: None

Source Addressing Mode	Assembler Language Syntax	Instruction Format											
R:	MULT RRd, Rs	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>10</td><td>011001</td><td>Rs</td><td>RRd</td></tr> </table>	10	011001	Rs	RRd							
10	011001	Rs	RRd										
MULTL RQd, RRs	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>10</td><td>011000</td><td>RRs</td><td>RQd</td></tr> </table>	10	011000	RRs	RQd								
10	011000	RRs	RQd										
IM:	MULT RRd, #data	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>00</td><td>011001</td><td>0000</td><td>RRd</td></tr> <tr> <td colspan="4" style="text-align: center;">data</td></tr> </table>	00	011001	0000	RRd	data						
00	011001	0000	RRd										
data													
MULTL RQd, #data	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>00</td><td>011000</td><td>0000</td><td>RQd</td></tr> <tr> <td colspan="4" style="text-align: center;">data (high)</td></tr> <tr> <td colspan="4" style="text-align: center;">data (low)</td></tr> </table>	00	011000	0000	RQd	data (high)				data (low)			
00	011000	0000	RQd										
data (high)													
data (low)													

Source Addressing Mode	Assembler Language Syntax	Instruction Format								
IR:	MULT RRd, @Rs ¹	<table border="1"> <tr> <td>00</td> <td>011001</td> <td>Rs≠0</td> <td>RRd</td> </tr> </table>	00	011001	Rs≠0	RRd				
00	011001	Rs≠0	RRd							
MULTL RQd, @Rs ¹	<table border="1"> <tr> <td>00</td> <td>011000</td> <td>Rs≠0</td> <td>RQd</td> </tr> </table>	00	011000	Rs≠0	RQd					
00	011000	Rs≠0	RQd							
EAM:	MULT RRd, eam-	<table border="1"> <tr> <td>01</td> <td>011001</td> <td>eam</td> <td>RRd</td> </tr> <tr> <td colspan="4">1, 2, or 3 extension words</td> </tr> </table>	01	011001	eam	RRd	1, 2, or 3 extension words			
01	011001	eam	RRd							
1, 2, or 3 extension words										
MULTL RQd, eam	<table border="1"> <tr> <td>01</td> <td>011000</td> <td>eam</td> <td>RQd</td> </tr> <tr> <td colspan="4">1, 2, or 3 extension words</td> </tr> </table>	01	011000	eam	RQd	1, 2, or 3 extension words				
01	011000	eam	RQd							
1, 2, or 3 extension words										

Example: If register RQ0 (composed of longword registers RR0 and RR2) contains %222222200000031 (RR2 contains decimal 49), executing the instruction

MULT RQ0,#10

leaves the value %000000000000001EA (decimal 490) in RQ0. The C, Z, S, and V flags are cleared.

Note 1: Word register in compact mode, longword register in segmented or linear modes.

MULTU

Multiply Unsigned

MULTU dst,src

dst: R
src: R, IM, IR, EAM

Operation: Word (dst is longword register, src is word)

$dst <31:0> \leftarrow dst <15:0> \times src <15:0>$

Longword (dst is quadword register, src is longword)

$dst <63:0> \leftarrow dst <31:0> \times src <31:0>$

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 unsigned integers. For MULTU, the destination is a longword register and the source is a word value; for MULTUL, the destination is a quadword register and the source is a longword value.

For proper instruction execution the "dst field" in the MULTUL instruction encoding must specify a valid code for a quadword register. Otherwise, the result is undefined.

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 C flag is set to indicate that the upper half of the destination register is required to represent the result; if the C 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 0.

Flags:	C: MULTU—set if product is greater than or equal to 2^{16} ; cleared otherwise; MULTUL—set if product is greater than or equal to 2^{32} ; 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: Cleared D: Unaffected H: Unaffected
---------------	--

Exceptions: None

Source Addressing Mode	Assembler Language Syntax	Instruction Format																												
R:	MULTU RRd, Rs	<table border="1"> <tr> <td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td> </tr> <tr> <td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>Rs</td><td>RRd</td><td></td><td></td><td></td><td></td><td></td> </tr> </table>	0	1	1	1	1	0	1	0	0	0	0	0	1	1	1	0	1	1	0	0	1	Rs	RRd					
0	1	1	1	1	0	1	0	0	0	0	0	1	1																	
1	0	1	1	0	0	1	Rs	RRd																						
MULTUL RQd, RRs	<table border="1"> <tr> <td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td> </tr> <tr> <td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>RRs</td><td>RQd</td><td></td><td></td><td></td><td></td><td></td> </tr> </table>	0	1	1	1	1	0	1	0	0	0	0	0	1	1	1	0	1	1	0	0	0	RRs	RQd						
0	1	1	1	1	0	1	0	0	0	0	0	1	1																	
1	0	1	1	0	0	0	RRs	RQd																						

Source Addressing Mode	Assembler Language Syntax	Instruction Format							
IM:	MULTU RRd, #data	<table border="1"> <tr><td>01111010</td><td>0000 0011</td></tr> <tr><td>00</td><td>011001 0000 RRd</td></tr> <tr><td colspan="2">data</td></tr> </table>	01111010	0000 0011	00	011001 0000 RRd	data		
01111010	0000 0011								
00	011001 0000 RRd								
data									
MULTUL RQd, #data	<table border="1"> <tr><td>01111010</td><td>0000 0011</td></tr> <tr><td>00</td><td>011000 0000 RQd</td></tr> <tr><td colspan="2">data(high)</td></tr> <tr><td colspan="2">data(low)</td></tr> </table>	01111010	0000 0011	00	011000 0000 RQd	data(high)		data(low)	
01111010	0000 0011								
00	011000 0000 RQd								
data(high)									
data(low)									
MULTU RRd, @Rs ¹	<table border="1"> <tr><td>01111010</td><td>0000 0011</td></tr> <tr><td>00</td><td>011001 Rs≠0 RRd</td></tr> </table>	01111010	0000 0011	00	011001 Rs≠0 RRd				
01111010	0000 0011								
00	011001 Rs≠0 RRd								
MULTUL RQd, @Rs ¹	<table border="1"> <tr><td>01111010</td><td>0000 0011</td></tr> <tr><td>00</td><td>011000 Rs≠0 RQd</td></tr> </table>	01111010	0000 0011	00	011000 Rs≠0 RQd				
01111010	0000 0011								
00	011000 Rs≠0 RQd								
EAM:	MULTU RRd, eam	<table border="1"> <tr><td>01111010</td><td>0000 0011</td></tr> <tr><td>01</td><td>011001 eam RRd</td></tr> <tr><td colspan="2">1, 2, or 3 extension words</td></tr> </table>	01111010	0000 0011	01	011001 eam RRd	1, 2, or 3 extension words		
01111010	0000 0011								
01	011001 eam RRd								
1, 2, or 3 extension words									
MULTUL RQd, eam	<table border="1"> <tr><td>01111010</td><td>0000 0011</td></tr> <tr><td>01</td><td>011000 eam RQd</td></tr> <tr><td colspan="2">1, 2, or 3 extension words</td></tr> </table>	01111010	0000 0011	01	011000 eam RQd	1, 2, or 3 extension words			
01111010	0000 0011								
01	011000 eam RQd								
1, 2, or 3 extension words									

Example: If register RR0 (composed of R0 and R1) contains % ABCD FFFF (R1 contains decimal 65,535), executing the instruction

MULTU RR0,#16

leaves the value % 000FFFF0 (decimal 1,048,560) in RR0. The C flag is set and the Z, S, and V flags are cleared.

Note 1: Word register in compact mode, longword register in segmented or linear modes.

NEG

Negate

NEG dst	dst: R, IR, EAM
NEGB	
NEGL	

Operation: dst \leftarrow -dst

The contents of the destination are negated, that is, replaced by two's complement values. Note that %8000 for NEG, %80 for NEGB, and %80000000 for NEGL are replaced by themselves since in two's complement representation the negative number with greatest magnitude has no positive counterpart; for these three 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, %80 for NEGB, or %80000000 for NEGL cleared otherwise
D:	Unaffected
H:	Unaffected

Exceptions: Integer Overflow trap

Destination Addressing Mode	Assembler Language Syntax	Instruction Format										
R:	NEG Rd NEGB Rbd	<table border="1" style="margin-left: auto; margin-right: auto;"><tr><td>10</td><td>00110</td><td>W</td><td>Rd</td><td>0010</td></tr></table>	10	00110	W	Rd	0010					
10	00110	W	Rd	0010								
NEGL RRd	<table border="1" style="margin-left: auto; margin-right: auto;"><tr><td>10</td><td>011100</td><td>RRd</td><td>0010</td></tr></table>	10	011100	RRd	0010							
10	011100	RRd	0010									
IR:	NEG @Rd ¹ NEGB @Rd ¹	<table border="1" style="margin-left: auto; margin-right: auto;"><tr><td>00</td><td>00110</td><td>W</td><td>Rd≠0</td><td>0010</td></tr></table>	00	00110	W	Rd≠0	0010					
00	00110	W	Rd≠0	0010								
NEGL @Rd ¹	<table border="1" style="margin-left: auto; margin-right: auto;"><tr><td>00</td><td>011100</td><td>Rd≠0</td><td>0010</td></tr></table>	00	011100	Rd≠0	0010							
00	011100	Rd≠0	0010									
EAM:	NEG eam NEGB eam	<table border="1" style="margin-left: auto; margin-right: auto;"><tr><td>01</td><td>00110</td><td>W</td><td>eam</td><td>0010</td></tr><tr><td colspan="5">1, 2, or 3 extension words</td></tr></table>	01	00110	W	eam	0010	1, 2, or 3 extension words				
01	00110	W	eam	0010								
1, 2, or 3 extension words												
NEGL eam	<table border="1" style="margin-left: auto; margin-right: auto;"><tr><td>01</td><td>011100</td><td>eam</td><td>0010</td></tr><tr><td colspan="4">1, 2, or 3 extension words</td></tr></table>	01	011100	eam	0010	1, 2, or 3 extension words						
01	011100	eam	0010									
1, 2, or 3 extension words												

Example: If register RR8 contains %0000051F, executing the instruction

NEGL RR8

leaves the value %FFFFFAE1 in RR8.

Note 1: Word register in compact mode, longword register in segmented or linear modes.

NOP

No Operation

NOP				
Operation: No operation is performed.				
Flags: No flags affected				
Exceptions: None				
Destination Addressing Mode	Assembler Language Syntax	Instruction Format		
	NOP	<table border="1"><tr><td>10001101</td><td>00000111</td></tr></table>	10001101	00000111
10001101	00000111			

OR

Or

OR dst, src	dst: R
ORB	src: R, IM, IR, EAM
ORL	

Operation: dst \leftarrow dst OR src

The source operand is logically ORed with the destination operand and the result is stored in the destination. A 1 bit is stored whenever either of the corresponding bits in the two operands is 1; otherwise a 0 bit is stored. The contents of the source 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:	OR, ORL—unaffected; ORB—set if parity of the result is even; cleared otherwise
D:	Unaffected
H:	Unaffected

Exceptions: None

Source Addressing Mode	Assembler Language Syntax	Instruction Format														
R:	OR Rd, Rs	<table border="1" style="margin-left: auto; margin-right: auto;"><tr><td>10</td><td>00010</td><td>W</td><td>Rs</td><td>Rd</td></tr></table>	10	00010	W	Rs	Rd									
10	00010	W	Rs	Rd												
ORB Rbd, Rbs	<table border="1" style="margin-left: auto; margin-right: auto;"><tr><td>01111010</td><td>0000 0010</td></tr><tr><td>10</td><td>000101</td><td>RRs</td><td>RRd</td></tr></table>	01111010	0000 0010	10	000101	RRs	RRd									
01111010	0000 0010															
10	000101	RRs	RRd													
OR Rd, #data	<table border="1" style="margin-left: auto; margin-right: auto;"><tr><td>00</td><td>000101</td><td>0000</td><td>Rd</td></tr><tr><td colspan="4">data</td></tr></table>	00	000101	0000	Rd	data										
00	000101	0000	Rd													
data																
ORB Rbd, #data	<table border="1" style="margin-left: auto; margin-right: auto;"><tr><td>00</td><td>000100</td><td>0000</td><td>Rd</td></tr><tr><td colspan="4">data</td></tr></table>	00	000100	0000	Rd	data										
00	000100	0000	Rd													
data																
ORL RRD, #data	<table border="1" style="margin-left: auto; margin-right: auto;"><tr><td>01111010</td><td>0000</td><td>0010</td></tr><tr><td>00</td><td>000101</td><td>0000</td><td>RRd</td></tr><tr><td colspan="4">data (high)</td></tr><tr><td colspan="4">data (low)</td></tr></table>	01111010	0000	0010	00	000101	0000	RRd	data (high)				data (low)			
01111010	0000	0010														
00	000101	0000	RRd													
data (high)																
data (low)																
OR Rd, @Rs ¹	<table border="1" style="margin-left: auto; margin-right: auto;"><tr><td>00</td><td>00010</td><td>W</td><td>Rs ≠ 0</td><td>Rd</td></tr></table>	00	00010	W	Rs ≠ 0	Rd										
00	00010	W	Rs ≠ 0	Rd												
IR:	ORB Rbd, @Rs ¹	<table border="1" style="margin-left: auto; margin-right: auto;"><tr><td>01111010</td><td>0000</td><td>0010</td></tr><tr><td>00</td><td>000101</td><td>Rs ≠ 0</td><td>RRd</td></tr></table>	01111010	0000	0010	00	000101	Rs ≠ 0	RRd							
01111010	0000	0010														
00	000101	Rs ≠ 0	RRd													

Source Addressing Mode	Assembler Language Syntax	Instruction Format											
EAM:	OR Rd, eam ORB Rbd, eam	<table border="1" data-bbox="948 361 1258 439"> <tr> <td>01</td> <td>00010</td> <td>W</td> <td>eam</td> <td>Rd</td> </tr> <tr> <td colspan="5">1, 2, or 3 extension words</td> </tr> </table>	01	00010	W	eam	Rd	1, 2, or 3 extension words					
01	00010	W	eam	Rd									
1, 2, or 3 extension words													
ORL RRd, eam	<table border="1" data-bbox="948 477 1258 593"> <tr> <td colspan="2">01111010</td> <td colspan="2">0000 0010</td> </tr> <tr> <td>01</td> <td>000101</td> <td>eam</td> <td>RRd</td> </tr> <tr> <td colspan="4">1, 2, or 3 extension words</td> </tr> </table>	01111010		0000 0010		01	000101	eam	RRd	1, 2, or 3 extension words			
01111010		0000 0010											
01	000101	eam	RRd										
1, 2, or 3 extension words													

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

ORB RL3,#%7B

leaves the value %FB (11111011) in RL3.

Note 1: Word register in compact mode, longword register in segmented or linear modes.

OTDR Privileged Instruction **Output, Decrement and Repeat**

Operation: repeat
 $dst \leftarrow src$
 AUTODECREMENT src (by 1 if OTDRB; by 2 if OTDR; by 4 if OTDRL)
 $r \leftarrow r - 1$
until $r = 0$

This instruction is used for block output of strings of data. The contents of the memory location addressed by the source register are loaded into the I/O port addressed by the destination word register. I/O port addresses are 16 bits. The source register is then decremented by one if OTDRB, by two if OTDR, or by four if OTDRL, 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. The entire operation is repeated until the result of decrementing r is zero. This instruction can output from 1 to 65,536 data elements. The source, destination, and counter registers must be distinct, non-overlapping registers.

This instruction can be interrupted after each execution of the basic operation.

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

Exceptions: Privileged Instruction trap

Addressing Mode	Assembler Language Syntax	Instruction Format																				
IR:	OTDR @Rd,@Rs ¹ , r OTDRB @Rd,@Rs ¹ , r OTDRL@Rd,@Rs ¹ , r	<table border="1" data-bbox="871 1328 1183 1383"> <tr> <td data-bbox="871 1328 956 1343">0011101</td> <td data-bbox="956 1328 992 1343">W</td> <td data-bbox="992 1328 1052 1343">Rs ≠ 0</td> <td data-bbox="1052 1328 1183 1343">1010</td> </tr> <tr> <td data-bbox="871 1343 956 1383">0000</td> <td data-bbox="956 1343 992 1383">r</td> <td data-bbox="992 1343 1052 1383">Rd ≠ 0</td> <td data-bbox="1052 1343 1183 1383">0000</td> </tr> </table> <table border="1" data-bbox="871 1415 1183 1469"> <tr> <td data-bbox="871 1415 956 1429">01111010</td> <td data-bbox="956 1415 992 1429">0000</td> <td data-bbox="992 1415 1183 1429">0010</td> </tr> <tr> <td data-bbox="871 1429 956 1444">00111011</td> <td data-bbox="956 1429 992 1444">Rs ≠ 0</td> <td data-bbox="992 1429 1183 1444">1010</td> </tr> <tr> <td data-bbox="871 1444 956 1469">0000</td> <td data-bbox="956 1444 992 1469">r</td> <td data-bbox="992 1444 1183 1469">Rd ≠ 0</td> </tr> <tr> <td data-bbox="871 1469 956 1484">0000</td> <td data-bbox="956 1469 992 1484"></td> <td data-bbox="992 1469 1183 1484"></td> </tr> </table>	0011101	W	Rs ≠ 0	1010	0000	r	Rd ≠ 0	0000	01111010	0000	0010	00111011	Rs ≠ 0	1010	0000	r	Rd ≠ 0	0000		
0011101	W	Rs ≠ 0	1010																			
0000	r	Rd ≠ 0	0000																			
01111010	0000	0010																				
00111011	Rs ≠ 0	1010																				
0000	r	Rd ≠ 0																				
0000																						

Example: In linear mode, if register R11 contains %0FFF, register RR22 contains %0000B006, and R13 contains 6, executing the instruction

OTDR @R11, @RR22, R13

outputs the string of words from locations %0000B006 to %0000AFFC (in descending order of address) to port %0FFF. RR22 contains %0000AFFA, and R13 contains 0. R11 is not affected. The V flag is set. In compact mode, a word register must be used instead of RR22.

Note 1: Word register in compact mode, longword register in segmented or linear modes.

OTIR

Privileged Instruction Output, Increment and Repeat

OTIR dst, src, r	dst: IR
OTIRB	src: IR
OTIRL	

Operation:

```

repeat
    dst ← src
    AUTOINCREMENT src (by 1 if OTIRB; by 2 if OTIR; by 4 if OTIRL)
    r ← r - 1
until r = 0

```

This instruction is used for block output of strings of data. The contents of the memory location addressed by the source register are loaded into the I/O port addressed by the destination word register. I/O port addresses are 16 bits. The source register is then incremented by one if OTIRB, by two if OTIR, or by four if OTIRL, 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. The entire operation is repeated until the result of decrementing r is zero. This instruction can output from 1 to 65,536 data elements. The source, destination, and counter registers must be distinct, non-overlapping registers.

This instruction can be interrupted after each execution of the basic operation.

Flags:

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

Exceptions: Privileged Instruction trap

Addressing Mode	Assembler Language Syntax	Instruction Format																
IR:	OTIR @Rd, @Rs ¹ , r OTIRB @Rd, @Rs ¹ , r OTIRL @Rd, @Rs ¹ , r	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>0011101</td> <td>W</td> <td>Rs ≠ 0</td> <td>0010</td> </tr> <tr> <td>0000</td> <td>r</td> <td>Rd ≠ 0</td> <td>0000</td> </tr> </table> <table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>01111010</td> <td>0000 0010</td> </tr> <tr> <td>00111011</td> <td>Rs ≠ 0</td> </tr> <tr> <td>0000</td> <td>Rd ≠ 0</td> </tr> <tr> <td>r</td> <td>0000</td> </tr> </table>	0011101	W	Rs ≠ 0	0010	0000	r	Rd ≠ 0	0000	01111010	0000 0010	00111011	Rs ≠ 0	0000	Rd ≠ 0	r	0000
0011101	W	Rs ≠ 0	0010															
0000	r	Rd ≠ 0	0000															
01111010	0000 0010																	
00111011	Rs ≠ 0																	
0000	Rd ≠ 0																	
r	0000																	

Example: In compact 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 or linear mode, a longword register must be used instead of R2.

Note 1: Word register in compact mode, longword register in segmented or linear modes.

OUT

Output

Privileged Instruction

	OUT dst, src OUTB OUTL	dst: IR, DA src: R									
Operation:	dst \leftarrow src	The contents of the source register are loaded into the destination, an output port. I/O port addresses are 16 bits.									
Flags:	No flags affected.										
Exceptions:	Privileged Instruction trap										
Destination Addressing Mode	Assembler Language Syntax	Instruction Format									
IR:	OUT @Rd, Rs OUTB @Rd, Rbs	<table border="1"> <tr> <td>0011111</td> <td>W</td> <td>Rd \neq 0</td> <td>Rs</td> </tr> </table>	0011111	W	Rd \neq 0	Rs					
0011111	W	Rd \neq 0	Rs								
OUTL @Rd,RRs	<table border="1"> <tr> <td>01111010</td> <td>0000 0010</td> </tr> <tr> <td>00</td> <td>111111</td> <td>Rd \neq 0</td> <td>RRs</td> </tr> </table>	01111010	0000 0010	00	111111	Rd \neq 0	RRs				
01111010	0000 0010										
00	111111	Rd \neq 0	RRs								
DA:	OUT port, Rs OUTB port, Rbs	<table border="1"> <tr> <td>0011101</td> <td>W</td> <td>Rs</td> <td>0110</td> </tr> <tr> <td colspan="4">port</td> </tr> </table>	0011101	W	Rs	0110	port				
0011101	W	Rs	0110								
port											
OUTL port, RRs	<table border="1"> <tr> <td>01111010</td> <td>0000 0010</td> </tr> <tr> <td>00</td> <td>111011</td> <td>RRs</td> <td>0110</td> </tr> <tr> <td colspan="4">port</td> </tr> </table>	01111010	0000 0010	00	111011	RRs	0110	port			
01111010	0000 0010										
00	111011	RRs	0110								
port											

Example: If register R6 contains %5252, executing the instruction

OUT %1120, R6

outputs the value %5252 to the port %1120.

Privileged Instruction

OUTD Output and Decrement

OUTD dst, src, r	dst: IR
OUTDB	src: IR
OUTDL	

Operation: $dst \leftarrow src$
 AUTODECREMENT src (by 1 if OUTDB; by 2 if OUTD; or by 4 if OUTDL)
 $r \leftarrow r - 1$

This instruction is used for block output of strings of data. The contents of the memory location addressed by the source register are loaded into the I/O port addressed by the destination word register. I/O port addresses are 16 bits. The source register is then decremented by one if OUTDB, by two if OUTD, or by four if OUTDL, 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. The source, destination, and counter registers must be distinct, non-overlapping registers.

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

Exceptions: Privileged Instruction trap

Addressing Mode	Assembler Language Syntax	Instruction Format																								
IR:	OUTD @Rd, @Rs ¹ , r OUTDB @Rd, @Rs ¹ , r OUTDL @Rd, @Rs ¹ , r	<table border="1"> <tr> <td>0011101</td> <td>W</td> <td>Rs ≠ 0</td> <td>1010</td> </tr> <tr> <td>0000</td> <td>r</td> <td>Rd ≠ 0</td> <td>1000</td> </tr> <tr> <td colspan="2"> </td> <td colspan="2"> </td> </tr> <tr> <td>01111010</td> <td>0000 0010</td> <td></td> <td></td> </tr> <tr> <td>00111011</td> <td>Rs ≠ 0</td> <td>1010</td> <td></td> </tr> <tr> <td>0000</td> <td>r</td> <td>Rd ≠ 0</td> <td>1000</td> </tr> </table>	0011101	W	Rs ≠ 0	1010	0000	r	Rd ≠ 0	1000					01111010	0000 0010			00111011	Rs ≠ 0	1010		0000	r	Rd ≠ 0	1000
0011101	W	Rs ≠ 0	1010																							
0000	r	Rd ≠ 0	1000																							
01111010	0000 0010																									
00111011	Rs ≠ 0	1010																								
0000	r	Rd ≠ 0	1000																							

Example: In linear mode, if register R2 contains the I/O port address %0030, register RR6 contains %12005552, the word at memory location %12005552 contains %1234, and register R8 contains %1001, executing the instruction

OUTD @R2, @RR6, R8

outputs the value %1234 to port %0030 and leaves the value %12005550 in RR6, and %1000 in R8. Register R2 is not affected. The V flag is cleared. In compact mode, a word register must be used instead of RR6.

Note 1: Word register in compact mode, longword register in segmented or linear modes.

OUTI

Privileged Instruction

Output and Increment

OUTI dst, src, r	dst: IR
OUTIB	src: IR
OUTIL	

Operation: $dst \leftarrow src$
 AUTOINCREMENT src (by 1 if OUTIB; by 2 if OUTI; by 4 if OUTIL)
 $r \leftarrow r - 1$

This instruction is used for block output of strings of data. The contents of the memory location addressed by the source register are loaded into the I/O port addressed by the destination word register. I/O port addresses are 16 bits. The source register is then incremented by one if OUTIB, by two if OUTI, or by four if OUTIL, 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. The source, destination, and counter registers must be distinct, non-overlapping registers.

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

Exceptions: Privileged Instruction trap

Addressing Mode	Assembler Language Syntax	Instruction Format																								
IR:	OUTI @Rd, @Rs ¹ , r OUTIB @Rd, @Rs ¹ , r OUTIL @Rd, @Rs ¹ , r	<table><tr><td>0011101</td><td>W</td><td>Rs ≠ 0</td><td>0010</td></tr><tr><td>0000</td><td>r</td><td>Rd ≠ 0</td><td>1000</td></tr><tr><td colspan="4"> </td></tr><tr><td>01111010</td><td>0000 0010</td><td></td><td></td></tr><tr><td>00111011</td><td>Rs ≠ 0</td><td>0010</td><td></td></tr><tr><td>0000</td><td>r</td><td>Rd ≠ 0</td><td>1000</td></tr></table>	0011101	W	Rs ≠ 0	0010	0000	r	Rd ≠ 0	1000					01111010	0000 0010			00111011	Rs ≠ 0	0010		0000	r	Rd ≠ 0	1000
0011101	W	Rs ≠ 0	0010																							
0000	r	Rd ≠ 0	1000																							
01111010	0000 0010																									
00111011	Rs ≠ 0	0010																								
0000	r	Rd ≠ 0	1000																							

Example: This instruction can be used in a “loop” of instructions that 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 0. This example assumes compact mode. In segmented or linear mode, a longword register must be used instead of R2.

```
        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            //branch if parity 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 compact mode, longword register in segmented or linear modes.

PCACHE

Purge Cache

Privileged Instruction

PCACHE

Operation: Purge all cache entries

All cache entries are invalidated. This instruction is executed when a memory location that may have been copied into the cache has been modified by another processor. For example, if a slave processor reads from a peripheral port to a memory location that may be copied in the cache, the cache must be purged.

Flags: No flags affected

Exceptions: Privileged Instruction trap

	Assembler Language Syntax	Instruction Format		
	PCACHE	<table border="1"><tr><td>01111010</td><td>00001000</td></tr></table>	01111010	00001000
01111010	00001000			

POP Pop

POP dst, src POPL	dst: R, IR, EAM src: IR									
Operation:	dst \leftarrow src AUTOINCREMENT src (by 2 if POP, by 4 if POPL)									
	<p>The contents of the location addressed by the source register (used as a stack pointer) are loaded into the destination. The source register is then incremented by two if POP or by four if POPL, thus removing the top element from the stack by changing the stack pointer. Any register except R0 in compact mode or RR0 in segmented or linear mode can be used as a stack pointer.</p> <p>If the destination is a register, the source and destination registers must be distinct and non-overlapping. Similarly, if the destination is in memory, then the source and destination operands must not overlap. Otherwise, the result of executing the instruction is undefined.</p>									
Flags:	No flags affected									
Exceptions:	None									
Destination Addressing Mode	Assembler Language Syntax	Instruction Format								
R:	POP Rd, @Rs ¹ POPL RRd, @Rs ¹	<table border="1" style="margin-left: auto; margin-right: auto; border-collapse: collapse; width: fit-content;"> <tr> <td style="padding: 2px;">10</td><td style="padding: 2px;">0 1 0 1 1 1</td><td style="padding: 2px;">Rs \neq 0</td><td style="padding: 2px;">Rd</td></tr> </table> <table border="1" style="margin-left: auto; margin-right: auto; border-collapse: collapse; width: fit-content;"> <tr> <td style="padding: 2px;">10</td><td style="padding: 2px;">0 1 0 1 0 1</td><td style="padding: 2px;">Rs \neq 0</td><td style="padding: 2px;">RRd</td></tr> </table>	10	0 1 0 1 1 1	Rs \neq 0	Rd	10	0 1 0 1 0 1	Rs \neq 0	RRd
10	0 1 0 1 1 1	Rs \neq 0	Rd							
10	0 1 0 1 0 1	Rs \neq 0	RRd							
IR:	POP @Rd ¹ , @Rs ¹ POPL @Rd ¹ , @Rs ¹	<table border="1" style="margin-left: auto; margin-right: auto; border-collapse: collapse; width: fit-content;"> <tr> <td style="padding: 2px;">00</td><td style="padding: 2px;">0 1 0 1 1 1</td><td style="padding: 2px;">Rs \neq 0</td><td style="padding: 2px;">Rd \neq 0</td></tr> </table> <table border="1" style="margin-left: auto; margin-right: auto; border-collapse: collapse; width: fit-content;"> <tr> <td style="padding: 2px;">00</td><td style="padding: 2px;">0 1 0 1 0 1</td><td style="padding: 2px;">Rs \neq 0</td><td style="padding: 2px;">Rd \neq 0</td></tr> </table>	00	0 1 0 1 1 1	Rs \neq 0	Rd \neq 0	00	0 1 0 1 0 1	Rs \neq 0	Rd \neq 0
00	0 1 0 1 1 1	Rs \neq 0	Rd \neq 0							
00	0 1 0 1 0 1	Rs \neq 0	Rd \neq 0							
EAM:	POP eam, @Rs ¹ POPL eam, @Rs ¹	<table border="1" style="margin-left: auto; margin-right: auto; border-collapse: collapse; width: fit-content;"> <tr> <td style="padding: 2px;">01</td><td style="padding: 2px;">0 1 0 1 1 1</td><td style="padding: 2px;">Rs \neq 0</td><td style="padding: 2px;">eam</td></tr> </table> <p style="text-align: center;">1, 2, or 3 extension words</p> <table border="1" style="margin-left: auto; margin-right: auto; border-collapse: collapse; width: fit-content;"> <tr> <td style="padding: 2px;">01</td><td style="padding: 2px;">0 1 0 1 0 1</td><td style="padding: 2px;">Rs \neq 0</td><td style="padding: 2px;">eam</td></tr> </table> <p style="text-align: center;">1, 2, or 3 extension words</p>	01	0 1 0 1 1 1	Rs \neq 0	eam	01	0 1 0 1 0 1	Rs \neq 0	eam
01	0 1 0 1 1 1	Rs \neq 0	eam							
01	0 1 0 1 0 1	Rs \neq 0	eam							
Example:		<p>In compact mode, if register R12 (used as a stack pointer) contains %1000, the word at location %1000 contains %0055, and register R3 contains %0022, executing the instruction</p> <pre style="margin-left: 40px;">POP R3, @R12</pre> <p>leaves the value %0055 in R3 and the value %1002 in R12. In segmented or linear mode, a longword register must be used instead of R12.</p>								
<p>Note 1: Word register in compact mode, longword register in segmented or linear modes.</p>										

PTLB

Purge TLB

Privileged Instruction

PTLB

Operation: Purge all TLB entries

All TLB entries are invalidated. This instruction is executed when system and normal mode address spaces are merged and the operating system changes from executing one user process to another.

Flags: No flags affected

Exceptions: Privileged Instruction trap

	Assembler Language Syntax	Instruction Format
	PTLB	01111010 0000 1010

Privileged Instruction

PTLBE

Operation: Purge the TLB entry for the effective address of src

If any TLB entry corresponds to the logical address of the source operand, that entry is invalidated. Four versions of the instruction are provided, one for each of the logical memory address spaces: normal data space (PTLBEND), normal instruction space (PTLBENI), system data space (PTLBESD), and system instruction space (PTLBESI).

This instruction is executed when information is changed in the translation tables for a page in one of the current address spaces. If the page is shared by current address spaces (for example, instruction and data spaces are merged), the page must be purged in each of the address spaces.

Flags: No flags affected

Exceptions: Privileged Instruction trap

Source Addressing Mode	Assembler Language Syntax	Instruction Format																																																
IR:	PTLBEND @Rs ¹	<table border="1" data-bbox="940 1039 1248 1104"> <tr> <td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td> </tr> <tr> <td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td> </tr> <tr> <td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td> </tr> <tr> <td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td> </tr> </table>	0	1	1	1	1	0	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0																								
0	1	1	1	1	0	1	0	1	0	1	0																																							
0	0	0	0	0	0	0	0	0	0	0	0																																							
PTLBENI @Rs ¹	<table border="1" data-bbox="940 1127 1248 1193"> <tr> <td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td> </tr> <tr> <td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td> </tr> <tr> <td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td> </tr> <tr> <td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td> </tr> </table>	0	1	1	1	1	0	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0																									
0	1	1	1	1	0	1	0	1	0	1	0																																							
0	0	0	0	0	0	0	0	0	0	0	0																																							
PTLBESD @Rs ¹	<table border="1" data-bbox="940 1216 1248 1281"> <tr> <td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td> </tr> <tr> <td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td> </tr> <tr> <td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td> </tr> <tr> <td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td> </tr> </table>	0	1	1	1	1	0	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0																									
0	1	1	1	1	0	1	0	1	0	1	0																																							
0	0	0	0	0	0	0	0	0	0	0	0																																							
PTLBESI @Rs ¹	<table border="1" data-bbox="940 1307 1248 1370"> <tr> <td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td> </tr> <tr> <td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td> </tr> <tr> <td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td> </tr> <tr> <td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td> </tr> </table>	0	1	1	1	1	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0																									
0	1	1	1	1	0	1	0	0	0	1	0																																							
0	0	0	0	0	0	0	0	0	0	0	0																																							

Source Addressing Mode	Assembler Language Syntax	Instruction Format																																
EAM:	PTLBEND eam	<table border="1" data-bbox="894 397 1204 508"> <tr><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td></tr> <tr><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr> <tr><td colspan="8">eam</td></tr> <tr><td colspan="8">0 0 0 0</td></tr> </table> <p data-bbox="926 481 1176 508">1, 2, or 3 extension words</p>	0	1	1	1	1	0	1	0	0	1	0	0	0	0	0	0	eam								0 0 0 0							
0	1	1	1	1	0	1	0																											
0	1	0	0	0	0	0	0																											
eam																																		
0 0 0 0																																		
PTLBENI eam	<table border="1" data-bbox="894 544 1204 656"> <tr><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td></tr> <tr><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr> <tr><td colspan="8">eam</td></tr> <tr><td colspan="8">0 0 0 0</td></tr> </table> <p data-bbox="926 629 1176 656">1, 2, or 3 extension words</p>	0	1	1	1	1	0	1	0	0	1	0	0	0	0	0	0	eam								0 0 0 0								
0	1	1	1	1	0	1	0																											
0	1	0	0	0	0	0	0																											
eam																																		
0 0 0 0																																		
PTLBESD eam	<table border="1" data-bbox="894 692 1204 804"> <tr><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td></tr> <tr><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr> <tr><td colspan="8">eam</td></tr> <tr><td colspan="8">0 0 0 0</td></tr> </table> <p data-bbox="926 777 1176 804">1, 2, or 3 extension words</p>	0	1	1	1	1	0	1	0	0	1	0	0	0	0	0	0	eam								0 0 0 0								
0	1	1	1	1	0	1	0																											
0	1	0	0	0	0	0	0																											
eam																																		
0 0 0 0																																		
PTLBESI eam	<table border="1" data-bbox="894 840 1204 952"> <tr><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr> <tr><td colspan="8">eam</td></tr> <tr><td colspan="8">0 0 0 0</td></tr> </table> <p data-bbox="926 925 1176 952">1, 2, or 3 extension words</p>	0	1	1	1	1	0	0	0	0	1	0	0	0	0	0	0	eam								0 0 0 0								
0	1	1	1	1	0	0	0																											
0	1	0	0	0	0	0	0																											
eam																																		
0 0 0 0																																		

Note 1: Word register in compact mode, longword register in segmented or linear modes.

Privileged Instruction**PTLBN****Purge TLB Normal Space**

PTLBN**Operation:** Purge Normal Space TLB entries

All TLB entries corresponding to pages in normal data or normal instruction address spaces are invalidated. This instruction is executed when system and normal mode address spaces are separated and the user operating system changes from one process executing in normal mode to another.

Flags: No flags affected

Exceptions: Privileged Instruction trap

	Assembler Language Syntax	Instruction Format		
	PTLBN	<table border="1"><tr><td>01111010</td><td>0000 1011</td></tr></table>	01111010	0000 1011
01111010	0000 1011			

PUSH

Push

PUSH dst, src	dst: IR
PUSHL	src: R, IM, IR, EAM

Operation: AUTODECREMENT dst (by 2 if PUSH, by 4 if PUSHL)
 $dst \leftarrow src$

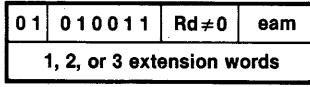
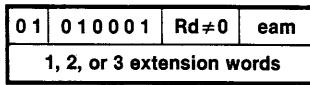
The contents of the destination register (used as a stack pointer) are decremented by two if PUSH or by four if PUSHL. Then the source operand is loaded into the location addressed by the updated destination register, thus adding a new element to the top of the stack by changing the stack pointer. Any register except R0 in compact mode or RR0 in segmented or linear mode can be used as a stack pointer.

If the source is a register, then the source and destination registers must be distinct and non-overlapping. Similarly, if the source is in memory, the source and destination operands must not overlap. Otherwise, the result of executing the instruction is undefined.

Flags: No flags affected

Exceptions: None

Source Addressing Mode	Assembler Language Syntax	Instruction Format											
R:	PUSH @Rd1, Rs	<table border="1"><tr><td>10</td><td>010011</td><td>Rd≠0</td><td>Rs</td></tr></table>	10	010011	Rd≠0	Rs							
10	010011	Rd≠0	Rs										
PUSHL @Rd1, RRs	<table border="1"><tr><td>10</td><td>010001</td><td>Rd≠0</td><td>RRs</td></tr></table>	10	010001	Rd≠0	RRs								
10	010001	Rd≠0	RRs										
IM:	PUSH @Rd1, #data	<table border="1"><tr><td>00</td><td>001101</td><td>Rd≠0</td><td>1001</td></tr><tr><td colspan="4">data</td></tr></table>	00	001101	Rd≠0	1001	data						
00	001101	Rd≠0	1001										
data													
PUSHL @Rd1, #data	<table border="1"><tr><td>00</td><td>010001</td><td>Rd≠0</td><td>0000</td></tr><tr><td colspan="4">data (high)</td></tr><tr><td colspan="4">data (low)</td></tr></table>	00	010001	Rd≠0	0000	data (high)				data (low)			
00	010001	Rd≠0	0000										
data (high)													
data (low)													
IR:	PUSH @Rd1, @Rs1	<table border="1"><tr><td>00</td><td>010011</td><td>Rd≠0</td><td>Rs≠0</td></tr></table>	00	010011	Rd≠0	Rs≠0							
00	010011	Rd≠0	Rs≠0										
PUSHL @Rd1, @Rs1	<table border="1"><tr><td>00</td><td>010001</td><td>Rd≠0</td><td>Rs≠0</td></tr></table>	00	010001	Rd≠0	Rs≠0								
00	010001	Rd≠0	Rs≠0										

Source Addressing Mode	Assembler Language Syntax	Instruction Format
EAM:	PUSH @Rd ¹ , eam	
	PUSHL @Rd ¹ , eam	

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

PUSH @R12, R3

leaves the value %0022 in location %1000 and the value %1000 in R12. In segmented or linear mode, a longword register must be used instead of R12.

Note 1: Word register in compact mode, longword register in segmented or linear modes.

RES

Reset Bit

RES dst, src	dst: R, IR, EAM
RESB	src: IM
RESL	or dst: R src: R

Operation: dst<src> $\leftarrow 0$

This instruction clears the specified bit within the destination operand to 0 without affecting any other bits in the destination. The bit number (the source) can be specified either as an immediate value (static), or as a word register that contains the value (dynamic). In the dynamic case, the destination operand must be in a register, and the source operand must be in a word register.

The bit number is a value from 0 to 7 for RESB, 0 to 15 for RES, or 0 to 31 for RESL, with 0 indicating the least-significant bit. Only the lower three bits of the source operand are used to specify the bit number for RESB, only the lower four bits are used for RES, and only the lower five bits are used for RESL.

Flags: No flags affected

Exceptions: None

Reset Bit Static

Destination Addressing Mode	Assembler Language Syntax	Instruction Format									
R:	RES Rd, #b RESB Rbd, #b	<table border="1"> <tr><td>10</td><td>10001</td><td>W</td><td>Rd</td><td>b</td></tr> </table> <table border="1"> <tr><td>01111010</td><td>0000 0010</td></tr> <tr><td>1010001</td><td>b RRd b</td></tr> </table>	10	10001	W	Rd	b	01111010	0000 0010	1010001	b RRd b
10	10001	W	Rd	b							
01111010	0000 0010										
1010001	b RRd b										
RESL RRd, #b	<table border="1"> <tr><td>01111010</td><td>0000 0010</td></tr> <tr><td>0010001</td><td>b Rd≠0 b</td></tr> </table>	01111010	0000 0010	0010001	b Rd≠0 b						
01111010	0000 0010										
0010001	b Rd≠0 b										
IR:	RES @Rd1, #b RESB @Rd1, #b	<table border="1"> <tr><td>0010001</td><td>W Rd≠0 b</td></tr> </table>	0010001	W Rd≠0 b							
0010001	W Rd≠0 b										
RESL @Rd1, #b	<table border="1"> <tr><td>01111010</td><td>0000 0010</td></tr> <tr><td>0010001</td><td>b Rd≠0 b</td></tr> </table>	01111010	0000 0010	0010001	b Rd≠0 b						
01111010	0000 0010										
0010001	b Rd≠0 b										
EAM:	RES eam, #b RESB eam, #b	<table border="1"> <tr><td>0110001</td><td>W eam b</td></tr> <tr><td colspan="4">1, 2, or 3 extension words</td></tr> </table>	0110001	W eam b	1, 2, or 3 extension words						
0110001	W eam b										
1, 2, or 3 extension words											
RESL eam, #b	<table border="1"> <tr><td>01111010</td><td>0000 0010</td></tr> <tr><td>0110001</td><td>b eam b</td></tr> <tr><td colspan="4">1, 2, or 3 extension words</td></tr> </table>	01111010	0000 0010	0110001	b eam b	1, 2, or 3 extension words					
01111010	0000 0010										
0110001	b eam b										
1, 2, or 3 extension words											

Reset Bit Dynamic

	Assembler Language Syntax	Instruction Format										
R:	RES Rd, Rs RESB Rb, Rs	<table border="1"> <tr> <td>00</td> <td>10001</td> <td>W</td> <td>0000</td> <td>Rs</td> </tr> <tr> <td>0000</td> <td>Rd</td> <td>0000</td> <td>0000</td> <td></td> </tr> </table>	00	10001	W	0000	Rs	0000	Rd	0000	0000	
00	10001	W	0000	Rs								
0000	Rd	0000	0000									
RESL RRd, Rs	<table border="1"> <tr> <td>01111010</td> <td>0000 0010</td> </tr> <tr> <td>00</td> <td>100011</td> <td>0000</td> <td>Rs</td> </tr> <tr> <td>0000</td> <td>RRd</td> <td>0000</td> <td>0000</td> </tr> </table>	01111010	0000 0010	00	100011	0000	Rs	0000	RRd	0000	0000	
01111010	0000 0010											
00	100011	0000	Rs									
0000	RRd	0000	0000									

Example: If register RL3 contains %B2 (10110010), executing the instruction
 RESB RL3, #1
 leaves the value %B0 (10110000) in RL3.

Note 1: Word register in compact mode, longword register in segmented or linear modes.

RESFLG

Reset Flag

RESFLG flag

flag: C, Z, S, P, V

Operation: $\text{FLAGS}_{<7:4>} \leftarrow \text{FLAGS}_{<7:4>} \text{ AND NOT instruction}_{<7:4>}$

Any combination of the C, Z, S, P or V flags can be cleared to 0. If the bit in the instruction corresponding to a flag is 1, the flag is cleared; if the bit is 0, the flag is unchanged. All other bits in the FLAGS register are unaffected. Note that the P and V flags are represented by the same bit. There can 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

Exceptions: None

	Assembler Language Syntax	Instruction Format					
	RESFLG flags	<table border="1"><tr><td>1</td><td>0</td><td>001101</td><td>CZSP/V</td><td>0011</td></tr></table>	1	0	001101	CZSP/V	0011
1	0	001101	CZSP/V	0011			

Example: If the C, S, and V flags are set (1) and the Z flag is clear (0), executing the statement
RESFLG C, V
leaves the S flag set (1), and the C, Z, and V flags clear (0).

RET

Return

RET cc

Operation:	<i>Compact</i> if cc is satisfied then PC ← @SP SP ← SP + 2	<i>Segmented or linear</i> if cc is satisfied then PC ← @SP SP ← SP + 4
-------------------	--	--

This instruction is used to return at the end of a procedure called by executing either a CALL or CALR instruction. If the condition specified by "cc" is satisfied by the flags in the FCW, then the contents of the top of the processor Stack Pointer are popped into the Program Counter (PC), thus returning control to the caller. See Section 6.3 for a list of condition codes. The Stack Pointer used is R15 in compact mode, or RR14 in segmented or linear 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

Exceptions: None

	Assembler Language Syntax	Instruction Format				
	RET cc	<table border="1"><tr><td>10</td><td>011110</td><td>0000</td><td>cc</td></tr></table>	10	011110	0000	cc
10	011110	0000	cc			

Example: In compact mode, if the Program Counter contains %2550, the Stack Pointer (R15) contains %3000, location %3000 contains %1004, and the Z flag is clear, executing the instruction

RET NZ

leaves the value %3002 in the Stack Pointer, and the Program Counter contains %1004 (the address of the next instruction to be executed).

RL

Rotate Left

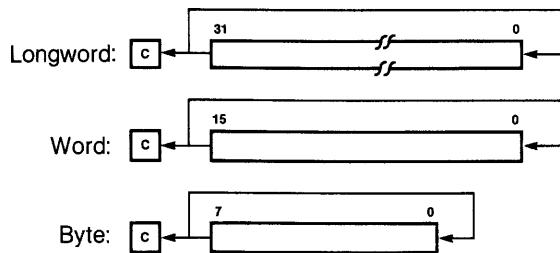
RL dst, src	dst: R
RLB	src: IM
RLL	

Operation:

```

for i ← 1 to src do
    C ← dst <msb>
    for j ← msb down to 1 do
        dst<j> ← dst<j -1>
    dst <0> ← C

```



The contents of the destination operand are rotated left one or two bit positions as specified by the source operand. During rotation, the most-significant bit (msb) of the destination operand is moved to the bit 0 position and also replaces the C flag.

If the source operand is omitted from the assembler language statement, the default value is one.

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

Exceptions: None

Destination Addressing Mode	Assembler Language Syntax	Instruction Format ¹
R:	RL Rd, #n RLB Rbd, #n RLL RRd, #n	<div style="display: flex; align-items: center; justify-content: space-between;"> <div style="flex-grow: 1; text-align: right; margin-right: 10px;"> 10 11001 W Rd 00 S 0 </div> <div style="border: 1px solid black; padding: 2px; font-family: monospace;"> 01111010 00000010 10 110011 RRd 00 S 0 </div> </div>

Example: If register RH5 contains %88 (10001000), executing the instruction
RLB RH5
leaves the value %11 (00010001) in RH5 and sets the C flag to 1.

Note 1: S = 0 for rotation by 1 bit; S = 1 for rotation by 2 bits.

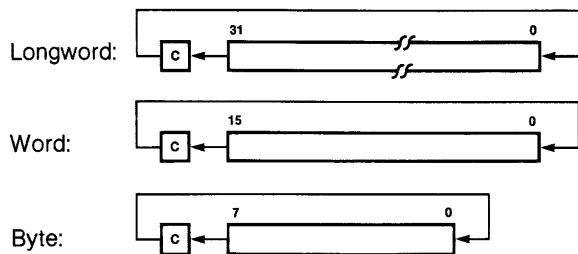
RLC

Rotate Left through Carry

RLC dst, src
RLCB
RLCL

dst: R
src: IM

Operation: for $i \leftarrow 1$ to src do
temp $\leftarrow C$
 $C \leftarrow dst < msb >$
for $j \leftarrow msb$ down to 1 do
 $dst < j > \leftarrow dst < j - 1 >$
 $dst < 0 > \leftarrow temp$



The contents of the destination operand concatenated with the C flag are rotated left one or two bit positions as specified by the source operand. During rotation, 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.

If the source operand is omitted from the assembler language statement, the default value is one.

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

Exceptions: None

Destination Addressing Mode	Assembler Language Syntax	Instruction Format ¹																					
R:	RLC Rd, #n RLCB Rbd, #n RLCL RRd, #n	<table border="1"> <tr> <td>10</td><td>11001</td><td>W</td><td>Rd</td><td>10</td><td>S</td><td>0</td> </tr> <tr> <td>01111010</td><td colspan="3">00000010</td><td colspan="3"></td> </tr> <tr> <td>10</td><td>110011</td><td>RRd</td><td>10</td><td>S</td><td>0</td><td></td> </tr> </table>	10	11001	W	Rd	10	S	0	01111010	00000010						10	110011	RRd	10	S	0	
10	11001	W	Rd	10	S	0																	
01111010	00000010																						
10	110011	RRd	10	S	0																		

Example: If the C flag is clear (0) and register R0 contains %800F (1000000000001111), executing the instruction

RLC R0,#2

leaves the value %003D (000000000111101) in R0 and clears the C flag.

Note 1: S = 0 for rotation by 1 bit; S = 1 for rotation by 2 bits

RLDB

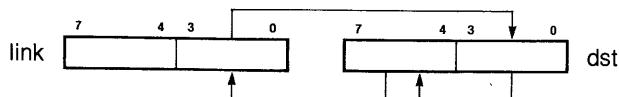
Rotate Left Digit

RLDB link, dst

link: R
dst: R

Operation:

```
temp <3:0> ← link <3:0>
link <3:0> ← dst <7:4>
dst <7:4> ← dst <3:0>
dst <3:0> ← temp <3:0>
```



The low digit of the link byte register is concatenated to the destination byte register. The resulting three-digit quantity is rotated to the left by one BCD digit (four bits). The lower digit of the destination is moved to the upper digit of the destination; the upper digit of the destination is moved to the lower digit of the link, and the lower digit of the link is moved to the lower digit of the destination. The upper digit of the link is unaffected.

In multiple-digit BCD arithmetic, this instruction can be used to shift a string of BCD digits to the left, 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 C flag in multiple precision shifting using the RLC instruction.

The destination and link registers must be distinct.

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

Exceptions:	None
--------------------	------

Destination Addressing Mode	Assembler Language Syntax	Instruction Format				
R:	RLDB Rbl, Rbd	<table border="1" style="display: inline-table;"><tr><td>10</td><td>111110</td><td>Rbd</td><td>Rbl</td></tr></table>	10	111110	Rbd	Rbl
10	111110	Rbd	Rbl			

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

100

0	1
---	---

 101

2	3
---	---

 102

4	5
---	---

executing the sequence of instructions in compact mode

LD	R3,#3	//set loop counter for 3 bytes //(6 digits)
LDA	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

leaves 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.

100

1	2
---	---

 101

3	4
---	---

 102

5	0
---	---

In segmented or linear mode, a longword register must be used instead of R2.

RR

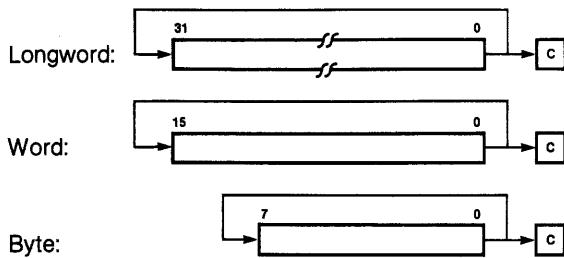
Rotate Right

RR dst, src	dst: R
RRB	src: IM
RRL	

Operation:

```

for i ← 1 to src do
    C ← dst<0>
    for j ← 1 to msb do
        dst<j-1> ← dst<j>
    dst <msb> ← C
  
```



The contents of the destination operand are rotated right one or two bit positions as specified by the source operand. During rotation, the least-significant bit of the destination operand is moved to the most-significant bit (msb) and also replaces the C flag.

If the source operand is omitted from the assembly language statement, the default value is one.

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

Exceptions: None

Destination Addressing Mode	Assembler Language Syntax	Instruction Format ¹
R:	RR Rd, #n RRB Rbd, #n RRL RRs, #n	<div style="display: flex; justify-content: space-around; align-items: center;"> 10 11001 W Rd 01 S 0 </div> <div style="display: flex; justify-content: space-around; align-items: center;"> 01111010 00000010 </div> <div style="display: flex; justify-content: space-around; align-items: center;"> 10 110011 RRd 01 S 0 </div>

Example: If register RL6 contains %31 (00110001), executing the instruction
RRB RL6
leaves the value %98 (10011000) in RL6 and sets the C flag to 1.

Note 1: S = 0 for rotation by 1 bit; S = 1 for rotation by 2 bits.

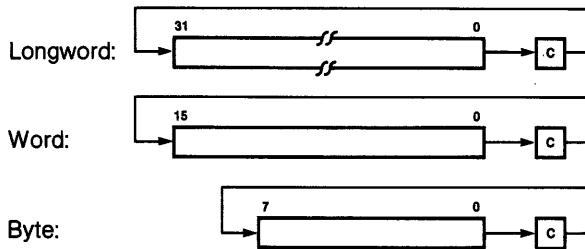
RRC

Rotate Right through Carry

RRC dst, src
RRCB
RRCL

dst: R
src: IM

Operation: for $i \leftarrow 1$ to src do
temp $\leftarrow C$
 $C \leftarrow dst < 0 >$
for $j \leftarrow 1$ to msb do
 $dst < j-1 > \leftarrow dst < j >$
 $dst < msb > \leftarrow temp$



The contents of the destination operand concatenated with the C flag are rotated right one or two bit positions as specified by the source operand. During rotation, 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.

If the source operand is omitted from the assembly language statement, the default value is one.

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
---------------	--

Exceptions:	None
--------------------	------

Destination Addressing Mode	Assembler Language Syntax	Instruction Format ¹																	
R:	RRC Rd, #n RRCB Rbd, #n RRCL RRd, #n	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>10</td> <td>11001</td> <td>W</td> <td>Rd</td> <td>11</td> <td>S</td> <td>0</td> </tr> </table> <table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td colspan="2">01111010</td> <td colspan="2">00000010</td> </tr> <tr> <td>10</td> <td>110011</td> <td>RRd</td> <td>11</td> <td>S</td> <td>0</td> </tr> </table>	10	11001	W	Rd	11	S	0	01111010		00000010		10	110011	RRd	11	S	0
10	11001	W	Rd	11	S	0													
01111010		00000010																	
10	110011	RRd	11	S	0														

Example: If the C flag is clear (0) and the register R0 contains %00DD (0000000011011101), executing the instruction

RRC R0,#2

leaves the value %8037 (1000000000110111) in R0 and clears the C flag.

Note 1: S = 0 for rotation by 1 bit; S = 1 for rotation by 2 bits

RRDB

Rotate Right Digit

RRDB link, dst

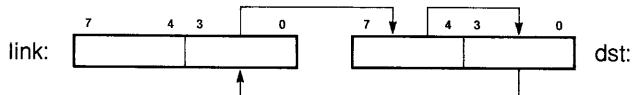
link: R
dst: R

Operation:

```

temp <3:0> ← link <3:0>
link <3:0> ← dst <3:0>
dst <3:0> ← dst <7:4>
dst <7:4> ← temp <3:0>

```



The low digit of the link byte register is concatenated to the destination byte register. The resulting three-digit quantity is rotated to the right by one BCD digit (four bits). The lower digit of the destination is moved to the lower digit of the link, the upper digit of the destination is moved to the lower digit of the destination, and the lower digit of the link is moved to the upper digit of the destination. The upper digit of the link is unaffected.

In multiple-digit BCD arithmetic, this instruction can be used to shift a string of BCD digits to the right, 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 C flag in multiple precision shifting using the RRC instruction.

The destination and link registers must be distinct.

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

Exceptions: None

Destination Addressing Mode	Assembler Language Syntax	Instruction Format				
R:	RRDB Rbl, Rbd	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>10</td> <td>111100</td> <td>Rbd</td> <td>Rbl</td> </tr> </table>	10	111100	Rbd	Rbl
10	111100	Rbd	Rbl			

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

100

1	2
---	---

 101

3	4
---	---

 102

5	6
---	---

executing the sequence of instructions in compact mode

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

leaves 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 contains 6, the remainder from dividing the string by 10.

100

0	1
---	---

 101

2	3
---	---

 102

4	5
---	---

In segmented or linear mode, a longword register must be used instead of R2.

SBC

Subtract with Carry

SBC dst, src SBCB SBCL	dst: R src: R
---	------------------

Operation: $\text{dst} \leftarrow \text{dst} - \text{src} - C$

The source operand, along with the setting of the C 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 "borrow" from the subtraction of low-order operands to be borrowed 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, SBCL—unaffected; SBCB—set H: SBC, SBCL—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
---------------	--

Exceptions: None

Source Addressing Mode	Assembler Language Syntax	Instruction Format																													
R:	SBC Rd, Rs SBCB Rbd, Rbs SBCL RRd, RRs	<table border="1" style="margin-bottom: 10px; width: 100%; border-collapse: collapse;"> <tr> <td style="width: 10%;">1</td> <td style="width: 10%;">0</td> <td style="width: 10%;">1</td> <td style="width: 10%;">0</td> <td style="width: 10%;">1</td> <td style="width: 10%;">1</td> <td style="width: 10%;">W</td> <td style="width: 10%;">R_s</td> <td style="width: 10%;">R_d</td> </tr> </table> <table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 50%;">0</td> <td style="width: 50%;">1</td> <td style="width: 50%;">1</td> <td style="width: 50%;">1</td> <td style="width: 50%;">0</td> <td style="width: 50%;">0</td> <td style="width: 50%;">0</td> <td style="width: 50%;">0</td> <td style="width: 50%;">1</td> <td style="width: 50%;">0</td> </tr> <tr> <td>1</td> <td>0</td> <td>1</td> <td>1</td> <td>0</td> <td>1</td> <td>1</td> <td>R_rs</td> <td>R_rd</td> <td></td> </tr> </table>	1	0	1	0	1	1	W	R _s	R _d	0	1	1	1	0	0	0	0	1	0	1	0	1	1	0	1	1	R _r s	R _r d	
1	0	1	0	1	1	W	R _s	R _d																							
0	1	1	1	0	0	0	0	1	0																						
1	0	1	1	0	1	1	R _r s	R _r d																							

Example: Quadword subtraction can be done with the following instruction sequence, assuming RQ0 contains one operand and RQ4 contains the other operand:

```
SUBL RR2,RR6      //subtract low-order longwords
SBCL RR0,RR4      //subtract borrow and high-order longwords
```

If RR0 contains %00000038, RR2 contains %00004000, RR4 contains %0000000A and RR6 contains %FFFFF000, executing the two instructions above leaves the value %0000002D in RR0 and %00005000 in RR2.

SC System Call

SC src

src: IM

Operation: $SP \leftarrow SP - 6$
 $@SP \leftarrow PS$
 $SP \leftarrow SP - 2$
 $@SP \leftarrow \text{instruction}$
 $PS \leftarrow \text{System Call PS}$

This instruction causes a System Call trap for controlled access to operating system software. The instruction word and the contents of the Program Status registers are pushed onto the system stack. The source operand, which is contained in the second byte of the instruction, identifies the particular service requested from the operating system. The source operand must be in the range from 0 to 255.

Flags: Flags loaded from Program Status Area

Exceptions: System Call trap

Source Addressing Mode	Assembler Language Syntax	Instruction Format		
IM:	SC #n	<table border="1"><tr><td>01111111</td><td>n</td></tr></table>	01111111	n
01111111	n			

SDA

Shift Dynamic Arithmetic

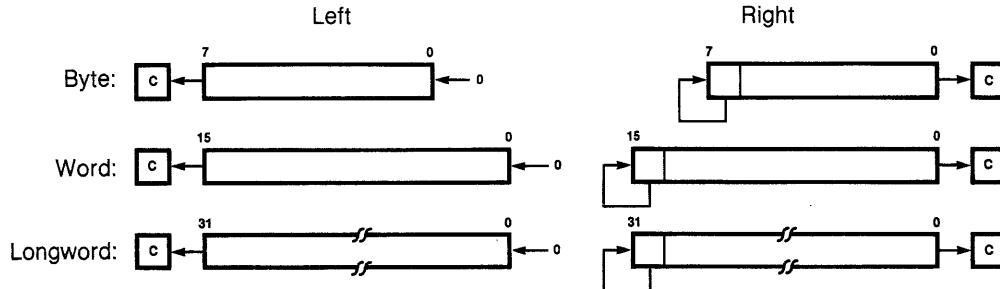
SDA dst, src	dst: R
SDAB	src: R
SDAL	

Operation:

```

if src ≥ 0           // left shift
    for i ← 1 to src do
        C ← dst<msb>
        for j ← msb down to 1 do
            dst<j> ← dst<j-1>
        dst<0> ← 0
    else for i ← 1 to -src do   // right shift
        C ← dst<0>
        for j ← 1 to msb do
            dst<j-1> ← dst<j>

```



The destination operand is shifted left or right arithmetically the number of bit positions specified by the source operand, a word register. For right shifts, the most-significant bit is replicated, and the C flag is loaded from the least-significant bit of the destination. For left shifts, the least-significant bit is filled with 0 and the C flag is loaded from the most-significant bit of the destination. A shift of zero positions does not affect the destination; however, the flags are set according to the destination value.

The source operand must be in the range from -8 to 8 for SDAB, from -16 to 16 for SDA or from -32 to 32 for SDAL. If its 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.

Flags:	C: Set if the last bit shifted from the destination was 1; cleared if the last bit shifted from the destination was 0 or zero shift was specified 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
Exceptions:	Integer Overflow trap

Destination Addressing Mode	Assembler Language Syntax	Instruction Format								
R:	SDA Rd, Rs	<table border="1"> <tr> <td>10</td> <td>110011</td> <td>Rd</td> <td>1011</td> </tr> <tr> <td>0000</td> <td>Rs</td> <td>0000 0000</td> <td></td> </tr> </table>	10	110011	Rd	1011	0000	Rs	0000 0000	
10	110011	Rd	1011							
0000	Rs	0000 0000								
SDAB Rbd, Rs	<table border="1"> <tr> <td>10</td> <td>110010</td> <td>Rbd</td> <td>1011</td> </tr> <tr> <td>0000</td> <td>Rs</td> <td>0000 0000</td> <td></td> </tr> </table>	10	110010	Rbd	1011	0000	Rs	0000 0000		
10	110010	Rbd	1011							
0000	Rs	0000 0000								
SDAL RRd, Rs	<table border="1"> <tr> <td>10</td> <td>110011</td> <td>RRd</td> <td>1111</td> </tr> <tr> <td>0000</td> <td>Rs</td> <td>0000 0000</td> <td></td> </tr> </table>	10	110011	RRd	1111	0000	Rs	0000 0000		
10	110011	RRd	1111							
0000	Rs	0000 0000								

Example: If register R5 contains %C705 (110001100000101) and register R1 contains – 2 (%FFE or 111111111111110), executing the instruction

SDA R5,R1

performs an arithmetic right shift of two bit positions, leaves the value %F1C1 (111100011000001) in R5, and clears the C flag.

SDL

Shift Dynamic Logical

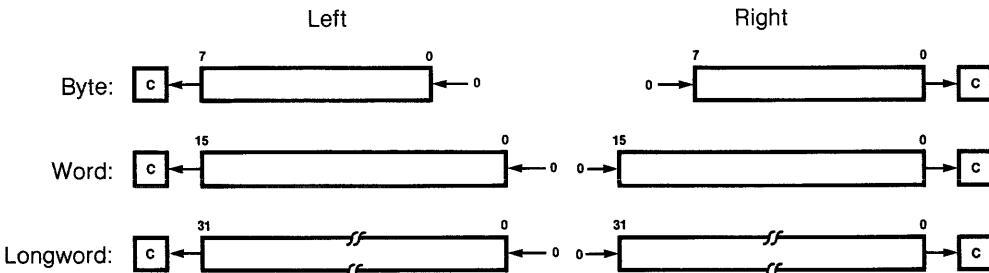
SDL dst, src	dst: R
SDLB	src: R
SDLL	

Operation:

```

if src ≥ 0                      // left shift
    for i ← 1 to src do
        C ← dst<msb>
        for j ← msb down to 1 do
            dst<j> ← dst<j-1>
        dst <0> ← 0
    else for i ← 1 to -src do      // right shift
        C ← dst<0>
        for j ← 1 to msb do
            dst<j-1> ← dst<j>
        dst<msb> ← 0

```



The destination operand is shifted left or right logically the number of bit positions specified by the source operand, a word register. For right shifts, the most-significant bit is filled with 0 and the C flag is loaded from the least-significant bit of the destination. For left shifts, the least-significant bit is filled with 0 and the C flag is loaded from the most-significant bit of the destination. A shift of zero positions does not affect the destination; however, the flags are set according to the destination value.

The source operand must be in the range from -8 to 8 for SDLB, from -16 to 16 for SDL or from -32 to 32 for SDLL. If its value is outside the specified range, the operation is undefined. The source operand is represented as a 16-bit twos complement value. Positive values specify a left shift, while negative values specify a right shift.

Flags:	C: Set if the last bit shifted from the destination was 1; cleared if the last bit shifted from the destination was 0 or zero shift was specified Z: Set if the result is zero; cleared otherwise S: Set if the most-significant bit of the result is set; cleared otherwise P: SDL, SDLL—unaffected; SDLB—set if parity of the result is even; cleared otherwise D: Unaffected H: Unaffected
---------------	--

Exceptions: None

Destination Addressing Mode	Assembler Language Syntax	Instruction Format								
R:	SDL Rd, Rs	<table border="1" data-bbox="954 369 1264 443"> <tr> <td>10</td> <td>110011</td> <td>Rd</td> <td>0011</td> </tr> <tr> <td>0000</td> <td>Rs</td> <td>0000 0000</td> <td></td> </tr> </table>	10	110011	Rd	0011	0000	Rs	0000 0000	
10	110011	Rd	0011							
0000	Rs	0000 0000								
SDLB Rbd, Rs	<table border="1" data-bbox="954 475 1264 549"> <tr> <td>10</td> <td>110010</td> <td>Rbd</td> <td>0011</td> </tr> <tr> <td>0000</td> <td>Rs</td> <td>0000 0000</td> <td></td> </tr> </table>	10	110010	Rbd	0011	0000	Rs	0000 0000		
10	110010	Rbd	0011							
0000	Rs	0000 0000								
SDLL RRd, Rs	<table border="1" data-bbox="954 580 1264 654"> <tr> <td>10</td> <td>110011</td> <td>RRd</td> <td>0111</td> </tr> <tr> <td>0000</td> <td>Rs</td> <td>0000 0000</td> <td></td> </tr> </table>	10	110011	RRd	0111	0000	Rs	0000 0000		
10	110011	RRd	0111							
0000	Rs	0000 0000								

Example: If register RL5 contains %B3 (10110011) and register R1 contains 4 (000000000000100), executing the instruction

SDLB RL5,R1

performs a logical left shift of four bit positions, leaves the value %30 (00110000) in RL5, and sets the C flag.

SET

Set Bit

SET dst, src	dst: R, IR, EAM
SETB	src: IM
SETL	or dst: R src: R

Operation: $\text{dst} < \text{src} > \leftarrow 1$

This instruction sets the specified bit within the destination operand to 1 without affecting any other bits in the destination. The bit number (the source) can be specified either as an immediate value (static), or as a word register that contains the value (dynamic). In the dynamic case, the destination operand must be in a register, and the source operand must be in a word register.

The bit number is a value from 0 to 7 for SETB, 0 to 15 for SET, or 0 to 31 for SETL with 0 indicating the least-significant bit. Only the lower three bits of the source operand are used to specify the bit number for SETB, only the lower four bits are used for SET, and only the lower five bits are used for SETL.

Flags: No flags affected

Exceptions: None

Set Bit Static

Destination Addressing Mode	Assembler Language Syntax	Instruction Format									
R:	SET Rd, #b SETB Rbd, #b	<table border="1"> <tr> <td>10</td> <td>10010</td> <td>W</td> <td>Rd</td> <td>b</td> </tr> </table> <table border="1"> <tr> <td>01111010</td> <td>0000 0010</td> </tr> <tr> <td>1010010</td> <td>b RRd b</td> </tr> </table>	10	10010	W	Rd	b	01111010	0000 0010	1010010	b RRd b
10	10010	W	Rd	b							
01111010	0000 0010										
1010010	b RRd b										
SETL RRd, #b	<table border="1"> <tr> <td>01111010</td> <td>0000 0010</td> </tr> <tr> <td>0010010</td> <td>b Rd b</td> </tr> </table>	01111010	0000 0010	0010010	b Rd b						
01111010	0000 0010										
0010010	b Rd b										
IR:	SET @Rd1, #b SETB @Rd1, #b	<table border="1"> <tr> <td>0010010</td> <td>W</td> <td>Rd≠0</td> <td>b</td> </tr> </table>	0010010	W	Rd≠0	b					
0010010	W	Rd≠0	b								
SETL @Rd1, #b	<table border="1"> <tr> <td>01111010</td> <td>0000 0010</td> </tr> <tr> <td>0010010</td> <td>b Rd b</td> </tr> </table>	01111010	0000 0010	0010010	b Rd b						
01111010	0000 0010										
0010010	b Rd b										
EAM:	SET eam, #b SETB eam, #b	<table border="1"> <tr> <td>0110010</td> <td>W</td> <td>eam</td> <td>b</td> </tr> </table> <p>1, 2, or 3 extension words</p>	0110010	W	eam	b					
0110010	W	eam	b								
SETL eam, #b	<table border="1"> <tr> <td>01111010</td> <td>0000 0010</td> </tr> <tr> <td>0110010</td> <td>b eam b</td> </tr> </table> <p>1, 2, or 3 extension words</p>	01111010	0000 0010	0110010	b eam b						
01111010	0000 0010										
0110010	b eam b										

Set Bit Dynamic

Addressing Mode	Assembler Language Syntax	Instruction Format																						
R:	SET Rd, Rs SETB Rbd, Rs SETL RRd, Rs	<table border="1" data-bbox="948 430 1258 515"> <tr> <td>00</td> <td>10010</td> <td>W</td> <td>0000</td> <td>Rs</td> </tr> <tr> <td>0000</td> <td>Rd</td> <td colspan="3">0000 0000</td> </tr> </table> <table border="1" data-bbox="948 544 1258 663"> <tr> <td>01111010</td> <td colspan="3">0000 0010</td> </tr> <tr> <td>00</td> <td>100101</td> <td>0000</td> <td>Rs</td> </tr> <tr> <td>0000</td> <td>RRd</td> <td colspan="2">0000 0000</td> </tr> </table>	00	10010	W	0000	Rs	0000	Rd	0000 0000			01111010	0000 0010			00	100101	0000	Rs	0000	RRd	0000 0000	
00	10010	W	0000	Rs																				
0000	Rd	0000 0000																						
01111010	0000 0010																							
00	100101	0000	Rs																					
0000	RRd	0000 0000																						

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

SETB RL3, R2

leaves the value %F2 (11110010) in RL3.

Note 1: Word register in compact mode, longword register in segmented or linear modes.

SETFLG

Set Flag

SETFLG flag

Flag: C, Z, S, P, V

Operation: $\text{FLAGS}_{<7:4>} \leftarrow \text{FLAGS}_{<7:4>} \text{ OR instruction}_{<7:4>}$

Any combination of the C, Z, S, P or V flags can be set to 1. If the bit in the instruction corresponding to a flag is 1, the flag is set; if the bit is 0, the flag is unchanged. All other bits in the Flags register are unaffected. Note that the P and V flags are represented by the same bit. There can 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

Exceptions: None

	Assembler Language Syntax	Instruction Format															
	SETFLG flags	<table border="1"><tr><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td><td>C</td><td>Z</td><td>S</td><td>P/V</td><td>0</td><td>0</td><td>1</td></tr></table>	1	0	0	0	1	1	0	1	C	Z	S	P/V	0	0	1
1	0	0	0	1	1	0	1	C	Z	S	P/V	0	0	1			

Example: If the C, Z, and S flags are all clear (0), and the P flag is set (1), executing the instruction

SETFLG C

leaves the C and P flags set (1), and the Z and S flags clear (0).

SLA

Shift Left Arithmetic

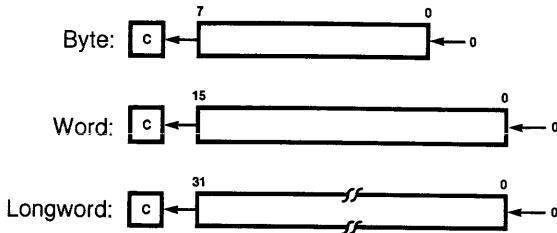
SLA dst, src	dst: R
SLAB	src: IM
SLAL	

Operation:

```

for i ← 1 to src do
    C ← dst <msb>
    for j ← msb down to 1 do
        dst<j> ← dst <j-1>
    dst <0> ← 0

```



The destination operand is shifted left arithmetically the number of bit positions specified by the source operand. The least-significant bit of the destination is filled with 0 and the C flag is loaded from the most-significant bit of the destination. A shift of zero position does not affect the destination; however, the flags are set according to the destination value. This operation differs from Shift Left Logical in the setting of the P/V flag and the detection of an Integer Overflow trap.

The source operand must be in the range from 0 to 8 for SLAB, from 0 to 16 for SLA, or from 0 to 32 for SLAL. If its value is outside the specified range, the operation is undefined. The source operand is encoded as an 8- or 16-bit two's complement number contained in the second word of the instruction. If the source operand is omitted from the assembly language statement, the default value is 1.

Flags:

- C:** Set if the last bit shifted from the destination was 1; cleared if the last bit shifted from the destination was 0 or zero shift was specified
- 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

Exceptions: Integer Overflow trap

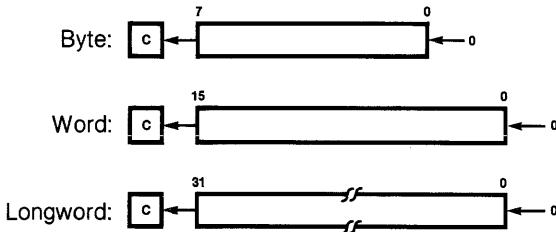
Destination Addressing Mode	Assembler Language Syntax	Instruction Format								
R:	SLA Rd, #b	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>10</td> <td>110011</td> <td>Rd</td> <td>1001</td> </tr> <tr> <td colspan="4" style="text-align: center;">b</td> </tr> </table>	10	110011	Rd	1001	b			
10	110011	Rd	1001							
b										
SLAB Rbd, #b	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>10</td> <td>110010</td> <td>Rbd</td> <td>1001</td> </tr> <tr> <td>0</td> <td colspan="3" style="text-align: center;">b</td> </tr> </table>	10	110010	Rbd	1001	0	b			
10	110010	Rbd	1001							
0	b									
SLAL RRd, #b	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>10</td> <td>110011</td> <td>RRd</td> <td>1101</td> </tr> <tr> <td colspan="4" style="text-align: center;">b</td> </tr> </table>	10	110011	RRd	1101	b				
10	110011	RRd	1101							
b										

Example: If longword register RR2 contains %1234ABCD, executing the instruction
SLAL RR2,#8
leaves the value %34ABCD00 in RR2 and clears the C flag.

SLL Shift Left Logical

SLL dst, src	dst: R
SLLB	src: IM
SLLL	

Operation: for $i \leftarrow 1$ to src do
 $C \leftarrow dst < msb >$
 for $j \leftarrow msb$ down to 1 do
 $dst < j > \leftarrow dst < j-1 >$
 $dst < 0 > \leftarrow 0$



The destination operand is shifted left logically the number of bit positions specified by the source operand. The least-significant bit of the destination is filled with 0 and the C flag is loaded from the most-significant bit of the destination. A shift of zero position does not affect the destination; however, the flags are set according to the destination value. This operation differs from Shift Left Arithmetic in the setting of the P/V flag and the detection of an Integer Overflow trap.

The source operand must be in the range from 0 to 8 for SLLB, from 0 to 16 for SLL, or from 0 to 32 for SLLL. If its value is outside the specified range, operation is undefined. The source operand is encoded as an 8- or 16-bit two's complement number contained in the second word of the instruction. If the source operand is omitted from the assembly language statement, the default value is one.

Flags:

- C:** Set if the last bit shifted from the destination was 1; cleared if the last bit shifted from the destination was 0 or zero shift was specified
- Z:** Set if the result is zero; cleared otherwise
- S:** Set if the most-significant bit of the result is set; cleared otherwise
- P:** SLL, SLLL—unaffected; SLLB—set if parity of the result is even; cleared otherwise
- D:** Unaffected
- H:** Unaffected

Exceptions: None

Destination Addressing Mode	Assembler Language Syntax	Instruction Format								
R:	SLL Rd, #b	<table border="1"> <tr><td>10</td><td>110011</td><td>Rd</td><td>0001</td></tr> <tr><td colspan="4">b</td></tr> </table>	10	110011	Rd	0001	b			
10	110011	Rd	0001							
b										
SLLB Rbd, #b	<table border="1"> <tr><td>10</td><td>110010</td><td>Rbd</td><td>0001</td></tr> <tr><td>0</td><td colspan="3">b</td></tr> </table>	10	110010	Rbd	0001	0	b			
10	110010	Rbd	0001							
0	b									
SLLL RRd, #b	<table border="1"> <tr><td>10</td><td>110011</td><td>RRd</td><td>0101</td></tr> <tr><td colspan="4">b</td></tr> </table>	10	110011	RRd	0101	b				
10	110011	RRd	0101							
b										

Example: If register R3 contains %4321 (0100001100100001), executing the instruction
 SLL R3,#1
 leaves the value %8642 (1000011001000010) in R3 and clears the C flag.

SRA

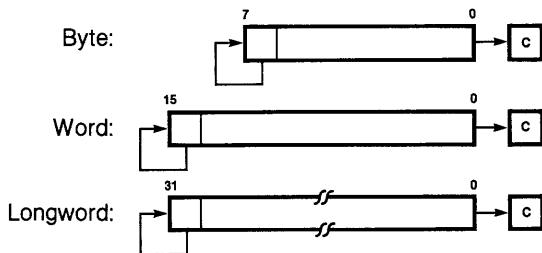
Shift Right Arithmetic

SRA dst, src SRAB SRAL	dst: R src: IM
---	-------------------

Operation:

```

for i ← 1 to src do
  C ← dst<0>
  for j ← 1 to msb do
    dst<j-1> ← dst<j>
  
```



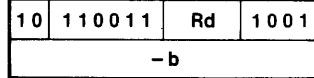
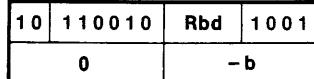
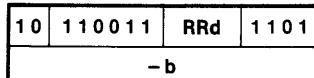
The destination operand is shifted right arithmetically the number of bit positions specified by the source operand. The most-significant bit of the destination is replicated, and the C flag is loaded from the least-significant bit of the destination.

The source operand must be in the range from 1 to 8 for SRAB, from 1 to 16 for SRA, or from 1 to 32 for SRAL. If its value is outside the specified range, the operation is undefined. The negative of the source operand is encoded as an 8- or 16-bit two's complement number contained in the second word of the instruction. If the source operand is omitted from the assembly language statement, the default value is one.

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

Exceptions: None

Destination Addressing Mode	Assembler Language Syntax	Instruction Format
R:	SRA Rd, #b SRAB Rbd, #b SRAL RRd, #b	  

Example: If register RH6 contains %3B (00111011), executing the instruction
SRAB RH6,#2
leaves the value %0E (00001110) in RH6 and sets the C flag.

SRL

Shift Right Logical

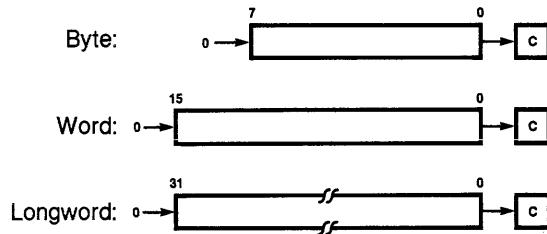
SRL dst, src SRLB SRLL	dst: R src: IM
---	-------------------

Operation:

```

for i ← 1 to src do
  C ← dst<0>
  for j ← 1 to msb do
    dst<j-1> ← dst<j>
  dst<msb> ← 0

```



The destination operand is shifted right logically the number of bit positions specified by the source operand. The most-significant bit of the destination is filled with 0 and the C flag is loaded from the least-significant bit of the destination.

The source operand must be in the range from 1 to 8 for SRLB, from 1 to 16 for SRL, or from 1 to 32 for SRL. If its value is outside the specified range, the operation is undefined. The negative of the source operand is encoded as an 8- or 16-bit two's complement number contained in the second word of the instruction. If the source operand is omitted from the assembly language statement, the default value is one.

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 1; cleared otherwise
- P:** SRL, SRLL—unaffected; SRLB—set if parity of the result is even; cleared otherwise
- D:** Unaffected
- H:** Unaffected

Exceptions: None

Destination Addressing Mode	Assembler Language Syntax	Instruction Format
R:	SRL Rd, #b	A 32-bit instruction format with four fields: - Bit 31 (10) is fixed at 1. - Bits 30-25 (110011) are fixed at 110011. - Bit 24 (Rd) is the register number. - Bits 23-20 (0001) are fixed at 0001. Below the format is the comment "- b".
	SRLB Rbd, #b	A 32-bit instruction format with four fields: - Bit 31 (10) is fixed at 1. - Bits 30-25 (110010) are fixed at 110010. - Bit 24 (Rbd) is the register number. - Bits 23-20 (0001) are fixed at 0001. Below the format is the comment "- b".
	SRLL RRd, #b	A 32-bit instruction format with four fields: - Bit 31 (10) is fixed at 1. - Bits 30-25 (110011) are fixed at 110011. - Bit 24 (RRd) is the register number. - Bits 23-20 (0101) are fixed at 0101. Below the format is the comment "- b".

Example: If register R0 contains %1111 (0001000100010001), executing the instruction
SRL R0,#6
 leaves the value %0044 (000000001000100) in R0 and clears the C flag.

SUB

Subtract

SUB dst, src SUBB SUBL	dst: R src: R, IM, IR, EAM
---	-------------------------------

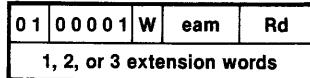
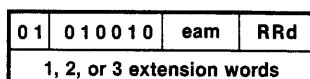
Operation: $dst \leftarrow dst - src$

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
---------------	--

Exceptions: Integer Overflow trap

Source Addressing Mode	Assembler Language Syntax	Instruction Format											
R:	SUB Rd, Rs SUBB Rbd, Rbs	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr><td>10</td><td>00001</td><td>W</td><td>Rs</td><td>Rd</td></tr> </table>	10	00001	W	Rs	Rd						
10	00001	W	Rs	Rd									
SUBL RRD, RRS	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr><td>10</td><td>010010</td><td>RRs</td><td>RRd</td></tr> </table>	10	010010	RRs	RRd								
10	010010	RRs	RRd										
IM:	SUB Rd, #data	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr><td>00</td><td>000011</td><td>0000</td><td>Rd</td></tr> <tr><td colspan="4">data</td></tr> </table>	00	000011	0000	Rd	data						
00	000011	0000	Rd										
data													
SUBB Rbd, #data	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr><td>00</td><td>000010</td><td>0000</td><td>Rbd</td></tr> <tr><td colspan="2">data</td><td colspan="2">data</td></tr> </table>	00	000010	0000	Rbd	data		data					
00	000010	0000	Rbd										
data		data											
SUBL RRD, #data	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr><td>00</td><td>010010</td><td>0000</td><td>RRd</td></tr> <tr><td colspan="2">data (high)</td><td colspan="2"></td></tr> <tr><td colspan="2">data (low)</td><td colspan="2"></td></tr> </table>	00	010010	0000	RRd	data (high)				data (low)			
00	010010	0000	RRd										
data (high)													
data (low)													
SUB Rd, @Rs ¹ SUBB Rbd, @Rs ¹	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr><td>00</td><td>00001</td><td>W</td><td>Rs ≠ 0</td><td>Rd</td></tr> </table>	00	00001	W	Rs ≠ 0	Rd							
00	00001	W	Rs ≠ 0	Rd									
IR:	SUBL RRD, @Rs ¹	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr><td>00</td><td>010010</td><td>Rs ≠ 0</td><td>RRd</td></tr> </table>	00	010010	Rs ≠ 0	RRd							
00	010010	Rs ≠ 0	RRd										

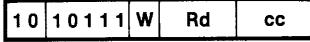
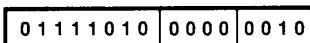
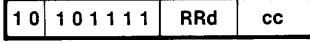
Source Addressing Mode	Assembler Language Syntax	Instruction Format
EAM:	SUB Rd, eam SUBB Rbd, eam	 01 00001 W eam Rd 1, 2, or 3 extension words
	SUBL RRd, eam	 01 010010 eam RRd 1, 2, or 3 extension words

Example: If register R0 contains %0344, executing the instruction
 SUB R0,#%AA
 leaves the value %029A in R0.

Note 1: Word register in compact mode, longword register in segmented or linear modes.

TCC

Test Condition Code

TCC cc, dst TCCB TCCL	dst: R	
Operation: if cc is satisfied then $dst <0> \leftarrow 1$		
	<p>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 specified condition is satisfied. If the condition is satisfied, then the least-significant bit of the destination is set. If the condition is not satisfied, bit 0 of the destination is unaffected. All other bits in the destination are unaffected by this instruction.</p>	
Flags: No flags affected		
Exceptions: None		
Destination Addressing Mode	Assembler Language Syntax	Instruction Format
R:	TCC cc, Rd TCCB cc, Rbd TCCL, cc, RRd	  

Example: If register R1 contains 0, and the Z flag is set, executing the instruction
TCC EQ,R1
leaves the value 1 in R1.

TEST

Test

TEST dst	dst: R, IR, EAM
TESTB	
TESTL	

Operation: dst OR 0

The destination operand is tested (logically ORed with zero), and the Z, S and P flags are set according to the result. This operation differs from Test Arithmetic in the setting of the C and P/V flags. 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, TESTL—unaffected; TESTB—set if parity of the result is even; cleared otherwise
- D:** Unaffected
- H:** Unaffected

Exceptions: None

Destination Addressing Mode	Assembler Language Syntax	Instruction Format										
R:	TEST Rd TESTB Rbd	<table border="1"> <tr><td>10</td><td>00110</td><td>W</td><td>Rd</td><td>0100</td></tr> </table>	10	00110	W	Rd	0100					
10	00110	W	Rd	0100								
TESTL RRd	<table border="1"> <tr><td>10</td><td>011100</td><td>RRd</td><td>1000</td></tr> </table>	10	011100	RRd	1000							
10	011100	RRd	1000									
IR:	TEST @Rd ¹ TESTB @Rd ¹	<table border="1"> <tr><td>00</td><td>00110</td><td>W</td><td>Rd≠0</td><td>0100</td></tr> </table>	00	00110	W	Rd≠0	0100					
00	00110	W	Rd≠0	0100								
TESTL @Rd ¹	<table border="1"> <tr><td>00</td><td>011100</td><td>Rd≠0</td><td>1000</td></tr> </table>	00	011100	Rd≠0	1000							
00	011100	Rd≠0	1000									
EAM:	TEST eam TESTB eam	<table border="1"> <tr><td>01</td><td>00110</td><td>W</td><td>eam</td><td>0100</td></tr> <tr><td colspan="5">1, 2, or 3 extension words</td></tr> </table>	01	00110	W	eam	0100	1, 2, or 3 extension words				
01	00110	W	eam	0100								
1, 2, or 3 extension words												
TESTL eam	<table border="1"> <tr><td>01</td><td>011100</td><td>eam</td><td>1000</td></tr> <tr><td colspan="5">1, 2, or 3 extension words</td></tr> </table>	01	011100	eam	1000	1, 2, or 3 extension words						
01	011100	eam	1000									
1, 2, or 3 extension words												

Example: If register R5 contains %FFFF (1111111111111111), executing the instruction
TEST R5
sets the S flag, clears the Z flag, and leaves the other flags unaffected.

Note 1: Word register in compact mode, longword register in segmented or linear modes.

TESTA

Test Arithmetic

TESTA dst TESTAB TESTAL	dst: R, IR, EAM																			
Operation:	dst – 0 Zero is compared to (subtracted from) the destination operand and the flags are set according to the result. The contents of the destination are not affected. This operation differs from Test in the setting of the C and P/V flags. Test Arithmetic must be used when an arithmetic condition (such as “greater than”) is required.																			
Flags:	C: Cleared Z: Set if the result is zero; cleared otherwise S: Set if the result is negative; cleared otherwise V: Cleared D: Unaffected H: Unaffected																			
Exceptions:	None																			
Destination Addressing Mode	Assembler Language Syntax	Instruction Format																		
R: TESTA Rd TESTAB Rbd TESTAL RRd	TESTA @Rd ¹ TESTAB @Rd ¹ TESTAL @Rd ¹	<table border="1" style="margin-bottom: 10px; width: 100%;"> <tr><td>10</td><td>00110</td><td>W</td><td>Rd</td><td>1100</td></tr> </table> <table border="1" style="width: 100%;"> <tr><td>10</td><td>01100</td><td>RRd</td><td>1100</td></tr> </table> <table border="1" style="margin-bottom: 10px; width: 100%;"> <tr><td>00</td><td>00110</td><td>W</td><td>Rd≠0</td><td>1100</td></tr> </table> <table border="1" style="width: 100%;"> <tr><td>00</td><td>01100</td><td>Rd≠0</td><td>1100</td></tr> </table>	10	00110	W	Rd	1100	10	01100	RRd	1100	00	00110	W	Rd≠0	1100	00	01100	Rd≠0	1100
10	00110	W	Rd	1100																
10	01100	RRd	1100																	
00	00110	W	Rd≠0	1100																
00	01100	Rd≠0	1100																	
IR:																				
EAM:	TESTA eam TESTAB eam TESTAL eam	<table border="1" style="margin-bottom: 10px; width: 100%;"> <tr><td>01</td><td>00110</td><td>W</td><td>eam</td><td>1100</td></tr> </table> <p style="text-align: center;">1, 2, or 3 extension words</p> <table border="1" style="margin-top: 10px; width: 100%;"> <tr><td>01</td><td>01100</td><td>eam</td><td>1100</td></tr> </table> <p style="text-align: center;">1, 2, or 3 extension words</p>	01	00110	W	eam	1100	01	01100	eam	1100									
01	00110	W	eam	1100																
01	01100	eam	1100																	

Example: If register R0 contains – 1 (%FFFF) executing the two instructions

```
TESTA R0
JR LE, NEG_OR_ZERO
```

transfers control to the instruction at label NEG_OR_ZERO. Note that using TEST instead of TESTA would require two JR instructions for equivalent effect because conditions involving the V flag cannot be used following TEST.

Note 1: Word register in compact mode, longword register in segmented or linear modes.

TRAP

Conditional Trap

TRAP cc, src

src: IM

Operation: if cc is satisfied then

SP \leftarrow SP-6
@ SP \leftarrow PS
SP \leftarrow SP-2
@ SP \leftarrow instruction
PS \leftarrow Conditional Trap PS

If the condition specified by "cc" is satisfied by the flags in the FCW, this instruction causes a Conditional trap. The instruction and the contents of the Program Status registers are pushed onto the system stack. The source operand, which is contained in bits 7 to 4 of the instruction, identifies the particular cause of the trap. The source operand must be in the range from 0 to 15. This instruction is used for the generation of exceptions detected by software, such as an overflow on decimal arithmetic.

Flags: Flags loaded from Program Status Area

Exceptions: Conditional trap

Source Addressing Mode	Assembler Language Syntax	Instruction Format			
IM:	TRAP cc, #n	<table border="1"><tr><td>01111110</td><td>n</td><td>cc</td></tr></table>	01111110	n	cc
01111110	n	cc			

TRDB

Translate and Decrement

TRDB dst, src, r

dst: IR
src: IR

Operation: $dst \leftarrow src[dst]$
 AUTODECREMENT dst by 1
 $r \leftarrow r - 1$

This instruction is used to translate a string of bytes from one code to another. The contents of the location addressed by the destination register (the "target byte") are used as an unsigned index into a translation table whose base address is contained in the source register. An effective address is calculated by adding the zero-extended target byte to the translation table base address using the rules for address arithmetic in the current mode of address representation: compact, segmented, or linear. The effective address is the location of the translated value used to replace the original contents of the target byte.

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 source register is unchanged. The source, destination, and counter registers must be distinct, non-overlapping registers. The translation table contains up to 256 bytes, one for each possible value of the target byte. The size of the translation table may be reduced when it is known that some target byte values will not occur.

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

Exceptions: None

Addressing Mode	Assembler Language Syntax	Instruction Format																				
IR:	TRDB @Rd1, @Rs1, r	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>Rd ≠ 0</td><td>1</td><td>0</td><td>0</td> </tr> <tr> <td>0</td><td>0</td><td>0</td><td>0</td><td>r</td><td>Rs ≠ 0</td><td>0</td><td>0</td><td>0</td><td>0</td> </tr> </table>	1	0	1	1	0	0	Rd ≠ 0	1	0	0	0	0	0	0	r	Rs ≠ 0	0	0	0	0
1	0	1	1	0	0	Rd ≠ 0	1	0	0													
0	0	0	0	r	Rs ≠ 0	0	0	0	0													

Example: In linear mode, if register RR6 contains %00004001, the byte at location %00004001 contains 3, register RR20 contains %00001000, the byte at location %00001003 contains %AA, and register R12 contains 2, executing the instruction
 TRDB @RR6, @RR20, R12

leaves the value %AA in location %00004001, the value %00004000 in RR6, and the value 1 in R12. RR20 is not affected. The V flag is cleared. In compact mode, word registers must be used instead of RR6 and RR20.

Note 1: Word register in compact mode, longword register in segmented or linear modes.

TRDRB

Translate, Decrement and Repeat

TRDRB dst, src, r

dst: IR

src: IR

Operation:

```
repeat
    dst ← src [dst]
    AUTODECREMENT dst by 1
    r ← r - 1
until r = 0
```

This instruction is used to translate a string of bytes from one code to another. The contents of the location addressed by the destination register (the "target byte") are used as an unsigned index into a translation table whose base address is contained in the source register. An effective address is calculated by adding the zero-extended target byte to the translation table base address using the rules for address arithmetic in the current mode of address representation: compact, segmented, or linear. The effective address is the location of the translated value used to replace the original contents of the target byte.

The destination register is the 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 source register is unchanged. The entire operation is repeated until the result of decrementing r is zero. This instruction can translate from 1 to 65,536 bytes. The source, destination, and counter registers must be distinct and non-overlapping registers. The translation table contains up to 256 bytes, one for each possible value of the target byte. The size of the translation table may be reduced when it is known that some target byte values will not occur.

This instruction can be interrupted after each execution of the basic operation.

Flags:

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

Exceptions: None

Addressing Mode	Assembler Language Syntax	Instruction Format								
IR:	TRDRB @Rd1, @Rs1, r	<table border="1"><tr><td>10</td><td>111000</td><td>Rd≠0</td><td>1100</td></tr><tr><td>0000</td><td>r</td><td>Rs≠0</td><td>0000</td></tr></table>	10	111000	Rd≠0	1100	0000	r	Rs≠0	0000
10	111000	Rd≠0	1100							
0000	r	Rs≠0	0000							

Example: In compact 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, executing the instruction

TRDRB @R6, @R9, R12

leaves the values %00, %40, %00 in byte locations %4000 through %4002, respectively. Register R6 contains %3FFF, and R12 contains 0. R9 is not affected. The V flag is set. In segmented or linear mode, longword registers must be used instead of R6 and R9.

BEFORE		%1000	0 0 0 0 0 0 0 0
%4000	0 0 0 0 0 0 0 0	%1001	0 0 0 0 0 0 0 1
%4001	0 1 0 0 0 0 0 0	%1002	0 0 0 0 0 0 1 0
%4002	1 0 0 0 0 0 0 0	•	•
		•	•
		•	•
AFTER		%107F	0 1 1 1 1 1 1 1
%4000	0 0 0 0 0 0 0 0	%1080	0 0 0 0 0 0 0 0
%4001	0 1 0 0 0 0 0 0	%1081	0 0 0 0 0 0 0 1
%4002	0 0 0 0 0 0 0 0	%1082	0 0 0 0 0 0 1 0
		•	•
		•	•
		•	•
		%10FF	0 1 1 1 1 1 1 1

Note 1: Word register in compact mode, longword register in segmented or linear modes.

TRIB

Translate and Increment

TRIB dst, src, r

dst: IR
src: IR

Operation: $dst \leftarrow src[dst]$
 AUTOINCREMENT dst by 1
 $r \leftarrow r - 1$

This instruction is used to translate a string of bytes from one code to another. The contents of the location addressed by the destination register (the "target byte") are used as an unsigned index into a translation table whose base address is contained in the source register. An effective address is calculated by adding the zero-extended target byte to the translation table base address using the rules for address arithmetic in the current mode of address representation: compact, segmented, or linear. The effective address is the location of the translated value used to replace the original contents of the target byte.

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 source register is unchanged. The source, destination, and counter registers must be distinct and non-overlapping registers. The translation table contains up to 256 bytes, one for each possible value of the target byte. The size of the translation table may be reduced when it is known that some target byte values will not occur.

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

Exceptions: None

Addressing Mode	Assembler Language Syntax	Instruction Format								
IR:	TRIB @Rd1, @Rs1, r	<table border="1"><tr><td>10</td><td>111000</td><td>Rd≠0</td><td>0000</td></tr><tr><td>0000</td><td>r</td><td>Rs≠0</td><td>0000</td></tr></table>	10	111000	Rd≠0	0000	0000	r	Rs≠0	0000
10	111000	Rd≠0	0000							
0000	r	Rs≠0	0000							

Example: This instruction can be used in a “loop” of instructions that translate a string of data from one code to another code, but an intermediate operation on each data element is required. The following sequence translates a string of 1000 bytes to the same string of bytes, with all ASCII “control characters” (values less than 32) 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 compact mode. In segmented or linear mode, longword registers must be used instead of R4 and R5.

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:		

TABLE + 0	0 0 1 0 0 0 0 0
TABLE + 1	0 0 1 0 0 0 0 0
TABLE + 2	0 0 1 0 0 0 0 0
•	•
•	•
•	•
TABLE + 32	0 0 1 0 0 0 0 0
TABLE + 33	0 0 1 0 0 0 0 1
TABLE + 34	0 0 1 0 0 0 1 0
•	•
•	•
•	•
TABLE + 255	1 1 1 1 1 1 1 1

Note 1: Word register in compact mode, longword register in segmented or linear modes.

TRIRB

Translate, Increment and Repeat

TRIRB dst, src, r

dst: IR

src: IR

Operation:

```

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

```

This instruction is used to translate a string of bytes from one code to another. The contents of the location addressed by the destination register (the "target byte") are used as an unsigned index into a translation table whose base address is contained in the source register. An effective address is calculated by adding the zero-extended target byte to the translation table base address using the rules for address arithmetic in the current mode of address representation: compact, segmented, or linear. The effective address is the location of the translated value used to replace the original contents of the target byte.

The destination register is then incremented by one, thus moving the pointer to the next byte in the string. The word register specified by "r" (used as a counter) is then decremented by one. The source register is unchanged. The entire operation is repeated until the result of decrementing r is zero. This instruction can translate from 1 to 65,536 bytes. The source, destination, and counter registers must be distinct and non-overlapping registers. The translation table contains up to 256 bytes, one for each possible value of the target byte. The size of the translation table may be reduced when it is known that some target byte values will not occur.

This instruction can be interrupted after each execution of the basic operation.

Flags:

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

Exceptions:

None

Addressing Mode	Assembler Language Syntax	Instruction Format								
IR:	TRIRB @Rd1, @Rs1, r	<table border="1" data-bbox="882 1501 1192 1571"> <tr> <td>10</td><td>111000</td><td>Rd≠0</td><td>0100</td></tr> <tr> <td>0000</td><td>r</td><td>Rs≠0</td><td>0000</td></tr> </table>	10	111000	Rd≠0	0100	0000	r	Rs≠0	0000
10	111000	Rd≠0	0100							
0000	r	Rs≠0	0000							

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. The example assumes compact mode. In segmented or linear mode, longword registers must be used instead of R4 and R5.

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

Note 1: Word register in compact mode, longword register in segmented or linear modes.

TRTDB

Translate, Test and Decrement

TRTDB src1, src2, r

src1: IR

src2: IR

Operation:

RH1 \leftarrow src2[src1]
AUTODECREMENT src1 by 1
r \leftarrow r - 1

This instruction is used to scan a string of bytes, testing for bytes with special values. The contents of the location addressed by the first source register (the "target byte") are used as an unsigned index into a translation table whose base address is contained in the second source register. An effective address is calculated by adding the zero-extended target byte to the base address using the current mode of address representation: compact, segmented, or linear. The effective address is the location of the translated value that is loaded into register RH1. The setting of the Z flag indicates whether or not the translated value is zero.

The first source register is then decremented by one, thus moving the pointer to the previous byte in the string. The word register specified by "r" (used as a counter) is then decremented by one. The second source register is unchanged. The source and counter registers must be distinct, non-overlapping registers. The translation table contains up to 256 bytes, one for each possible value of the target byte. The size of the translation table may be reduced when it is known that some target byte values will not occur.

Flags:

C: Unaffected
Z: Set if the translated 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

Exceptions: None

Addressing Mode	Assembler Language Syntax	Instruction Format								
IR:	TRTDB @Rs11, @Rs21, r	<table border="1"><tr><td>10</td><td>111000</td><td>Rs1≠0</td><td>1010</td></tr><tr><td>0000</td><td>r</td><td>Rs2≠0</td><td>0000</td></tr></table>	10	111000	Rs1≠0	1010	0000	r	Rs2≠0	0000
10	111000	Rs1≠0	1010							
0000	r	Rs2≠0	0000							

Example:

In compact 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, executing the instruction

TRTDB @R6, @R9, R12

leaves the value %AA in RH1, the value %4000 in R6, and the value 1 in R12. Location %4001 and register R9 are not affected. The Z and V flags are cleared. In segmented or linear mode, longword registers must be used instead of R6 and R9.

Note 1: Word register in compact mode, longword register in segmented or linear modes.

TRTDRB

Translate, Test, Decrement and Repeat

TRTDRB src1, src2, r	src1: IR src2: IR
-----------------------------	----------------------

Operation:

```

repeat
    RH1 ← src2[src1]
    AUTODECREMENT src1 by 1
    r ← r - 1
until RH1 ≠ 0 or r = 0

```

This instruction is used to scan a string of bytes, testing for bytes with special values. The contents of the location addressed by the first source register (the "target byte") are used as an unsigned index into a translation table whose base address is contained in the second source register. An effective address is calculated by adding the zero-extended target byte to the base address using the current mode of address representation: compact segmented, or linear. The effective address is the location of the translated value that is loaded into register RH1. The setting of the Z flag indicates whether or not the translated value is zero.

The first source register is then decremented by one, thus moving the pointer to the previous byte 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 a non-zero value is loaded into RH1 or the result of decrementing r is zero. This instruction can translate and test from 1 to 65,536 bytes. The second source register is unchanged. The source and counter registers must be distinct and non-overlapping registers. The translation table contains up to 256 bytes, one for each possible value of the target byte. The size of the translation table may be reduced when it is known that some target byte values will not occur.

This instruction can be interrupted after each execution of the basic operation.

Flags:	C: Unaffected Z: Set if the translated 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
---------------	--

Exceptions:	None
--------------------	------

Addressing Mode	Assembler Language Syntax	Instruction Format																							
IR:	TRTDRB @Rs1 ¹ ,@Rs2 ¹ ,r	<table border="1" style="margin-left: auto; margin-right: auto; border-collapse: collapse; text-align: center;"> <tr> <td style="padding: 2px;">1</td><td style="padding: 2px;">0</td><td style="padding: 2px;">1</td><td style="padding: 2px;">1</td><td style="padding: 2px;">0</td><td style="padding: 2px;">0</td><td style="padding: 2px;">0</td><td style="padding: 2px;">R</td><td style="padding: 2px;">s1≠0</td><td style="padding: 2px;">1</td><td style="padding: 2px;">1</td><td style="padding: 2px;">0</td> </tr> <tr> <td style="padding: 2px;">0</td><td style="padding: 2px;">0</td><td style="padding: 2px;">0</td><td style="padding: 2px;">0</td><td style="padding: 2px;">r</td><td style="padding: 2px;">R</td><td style="padding: 2px;">s2≠0</td><td style="padding: 2px;">1</td><td style="padding: 2px;">1</td><td style="padding: 2px;">1</td><td style="padding: 2px;">0</td> </tr> </table>	1	0	1	1	0	0	0	R	s1≠0	1	1	0	0	0	0	0	r	R	s2≠0	1	1	1	0
1	0	1	1	0	0	0	R	s1≠0	1	1	0														
0	0	0	0	r	R	s2≠0	1	1	1	0															

Example: In compact 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, executing the instruction

TRTDRB @R6, @R9, R12

leaves the value %40 in RH1 (which was loaded from location %1040). Register R6 contains %4000, and R12 contains 1. R9 is not affected. The Z and V flags are cleared. In segmented or linear mode, longword registers must be used instead of R6 and R9.

%4000	0 0 0 0 0 0 0 0	%1000	0 0 0 0 0 0 0 0
%4001	0 1 0 0 0 0 0 0	%1001	0 0 0 0 0 0 0 1
%4002	1 0 0 0 0 0 0 0	•	•
		•	•
		•	•
		•	•
		%107F	0 1 1 1 1 1 1 1
		%1080	0 0 0 0 0 0 0 0
		%1081	0 0 0 0 0 0 0 1
		%1082	0 0 0 0 0 0 1 0
		•	•
		•	•
		•	•
		%10FF	0 1 1 1 1 1 1 1

Note 1: Word register in compact mode, longword register in segmented or linear modes.

TRTIB

Translate, Test and Increment

TRTIB src1, src2, r

src1: IR
src2: 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 values. The contents of the location addressed by the first source register (the "target byte") are used as an unsigned index into a translation table whose base address is contained in the second source register. An effective address is calculated by adding the zero-extended target byte to the base address using the current mode of address representation: compact, segmented, or linear. The effective address is the location of the translated value that is loaded into register RH1. The setting of the Z flag indicates whether or not the translated value is zero.

The first source register is then incremented by one, thus moving the pointer to the next byte in the string. The word registers specified by "r" (used as a counter) is then decremented by one. The second source register is unchanged. The source and counter registers must be distinct and non-overlapping registers. The translation table contains up to 256 bytes, one for each possible value of the target byte. The size of the translation table may be reduced when it is known that some target byte values will not occur.

Flags:

- C:** Unaffected
- Z:** Set if the translated 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

Exceptions: None

Addressing Mode	Assembler Language Syntax	Instruction Format								
IR:	TRTIB @Rs11, @Rs21, r	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>10</td><td>111000</td><td>Rs1≠0</td><td>0010</td></tr> <tr> <td>0000</td><td>r</td><td>Rs2≠0</td><td>0000</td></tr> </table>	10	111000	Rs1≠0	0010	0000	r	Rs2≠0	0000
10	111000	Rs1≠0	0010							
0000	r	Rs2≠0	0000							

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 compact mode. In segmented or linear mode, longword registers must 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 compact mode, longword register in segmented or linear modes.

TRTIRB

Translate, Test, Increment and Repeat

TRTIRB src1, src2, r src1: IR
 src2: IR

Operation:

```

repeat
    RH1 ← src2[src1]
    AUTOINCREMENT src1 by 1
    r ← r - 1
until RH1 ≠ 0 or r = 0

```

This instruction is used to scan a string of bytes, testing for bytes with special values. The contents of the location addressed by the first source register (the "target byte") are used as an unsigned index into a translation table whose base address is contained in the second source register. An effective address is calculated by adding the zero-extended target byte to the base address using the current mode of address representation: compact, segmented, or linear. The effective address is the location of the translated value that is loaded into register RH1. The setting of the Z flag indicates whether or not the translated value is zero.

The first source register is then incremented by one, thus moving the pointer to the next byte 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 a non-zero value is loaded into RH1 or the result of decrementing r is zero. This instruction can translate and test from 1 to 65,536 bytes. The second source register is unchanged. The source and counter registers must be distinct and non-overlapping registers.

The translation table contains up to 256 bytes, one for each possible value of the target byte. The size of the translation table may be reduced when it is known that some target byte values will not occur.

This instruction can be interrupted after each execution of the basic operation.

Flags:

- C:** Unaffected
- Z:** Set if the translated 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

Exceptions: None

Addressing Mode	Assembler Language Syntax	Instruction Format																										
IR:	TRTIRB @Rs11, @Rs21, r	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>1</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>R</td><td>s1≠0</td><td>0</td><td>1</td><td>1</td><td>0</td> </tr> <tr> <td>0</td><td>0</td><td>0</td><td>0</td><td>r</td><td>R</td><td>s2≠0</td><td>1</td><td>1</td><td>1</td><td>0</td><td></td><td></td> </tr> </table>	1	0	1	1	1	0	0	R	s1≠0	0	1	1	0	0	0	0	0	r	R	s2≠0	1	1	1	0		
1	0	1	1	1	0	0	R	s1≠0	0	1	1	0																
0	0	0	0	r	R	s2≠0	1	1	1	0																		

Example:

The following sequence of instructions can be used in compact 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 can then be used to index another table, or to select one of a set of sequences of instructions to execute. In segmented or linear mode, longword registers must be used instead of R4 and R5.

```
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 compact mode, longword register in segmented or linear modes.

TSET

Test and Set

TSET dst TSETB TSETL	dst: R, IR, EAM																										
Operation:	$S \leftarrow dst < msb >$ $dst \leftarrow -1$																										
	This instruction 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. It provides a locking mechanism for synchronizing software processes that require exclusive access to certain data or instructions at one time. No other interlocked accesses are permitted to the destination memory location between fetching and storing the result.																										
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																										
Exceptions:	None																										
Destination Addressing Mode	Assembler Language Syntax	Instruction Format																									
R:	TSET Rd TSETB Rbd TSETL RRd	<table border="1" style="margin-left: auto; margin-right: auto; border-collapse: collapse;"> <tr><td>10</td><td>00110</td><td>W</td><td>Rd</td><td>0110</td></tr> <tr><td colspan="2">01111010</td><td>0000</td><td>0010</td><td></td></tr> <tr><td>10</td><td>001101</td><td>RRd</td><td>0110</td><td></td></tr> </table>	10	00110	W	Rd	0110	01111010		0000	0010		10	001101	RRd	0110											
10	00110	W	Rd	0110																							
01111010		0000	0010																								
10	001101	RRd	0110																								
IR:	TSET @Rd ¹ TSETB @Rd ¹ TSETL @Rd ¹	<table border="1" style="margin-left: auto; margin-right: auto; border-collapse: collapse;"> <tr><td>00</td><td>00110</td><td>W</td><td>Rd ≠ 0</td><td>0110</td></tr> <tr><td colspan="2">01111010</td><td>0000</td><td>0010</td><td></td></tr> <tr><td>00</td><td>001101</td><td>Rd ≠ 0</td><td>0110</td><td></td></tr> </table>	00	00110	W	Rd ≠ 0	0110	01111010		0000	0010		00	001101	Rd ≠ 0	0110											
00	00110	W	Rd ≠ 0	0110																							
01111010		0000	0010																								
00	001101	Rd ≠ 0	0110																								
EAM:	TSET eam TSETB eam TSETL eam	<table border="1" style="margin-left: auto; margin-right: auto; border-collapse: collapse;"> <tr><td>01</td><td>00110</td><td>W</td><td>eam</td><td>0110</td></tr> <tr><td colspan="4">1, 2, or 3 extension words</td><td></td></tr> <tr><td colspan="2">01111010</td><td>0000</td><td>0010</td><td></td></tr> <tr><td>01</td><td>001101</td><td>eam</td><td>0110</td><td></td></tr> <tr><td colspan="4">1, 2, or 3 extension words</td><td></td></tr> </table>	01	00110	W	eam	0110	1, 2, or 3 extension words					01111010		0000	0010		01	001101	eam	0110		1, 2, or 3 extension words				
01	00110	W	eam	0110																							
1, 2, or 3 extension words																											
01111010		0000	0010																								
01	001101	eam	0110																								
1, 2, or 3 extension words																											

Example: A simple mutually-exclusive critical region can 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

Note 1: Word register in compact mode, longword register in segmented or linear modes.

XOR

Exclusive Or

XOR dst, src XORB XORL	dst: R src: R, IM, IR, EAM
---	-------------------------------

Operation: dst \leftarrow dst XOR src

A logical XOR operation is performed between the corresponding bits of the source and destination operands, and the result is stored in the destination. A 1 bit is stored wherever the corresponding bits in the two operands differ; otherwise a 0 bit is stored. The contents of the source 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: XOR, XORL—unaffected; XORB—set if parity of the result is even; cleared otherwise D: Unaffected H: Unaffected
---------------	--

Source Addressing Mode	Assembler Language Syntax	Instruction Format																																											
R:	XOR Rd, Rs XORB Rbd, Rbs XORL RRd, RRs	<table border="1" style="margin-bottom: 10px; text-align: center;"> <tr><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>W</td><td>Rs</td><td>Rd</td></tr> </table> <table border="1" style="text-align: center;"> <tr><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td></tr> <tr><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>RRs</td><td>RRd</td></tr> </table>	1	0	0	1	0	0	W	Rs	Rd	0	1	1	1	0	1	0	0	0	0	0	1	0	1	0	0	1	0	0	1	RRs	RRd												
1	0	0	1	0	0	W	Rs	Rd																																					
0	1	1	1	0	1	0	0	0	0	0	1	0																																	
1	0	0	1	0	0	1	RRs	RRd																																					
IM:	XOR Rd, #data	<table border="1" style="margin-bottom: 10px; text-align: center;"> <tr><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>Rd</td></tr> <tr><td colspan="10">data</td></tr> </table>	0	0	0	1	0	0	1	0	0	0	Rd	data																															
0	0	0	1	0	0	1	0	0	0	Rd																																			
data																																													
XORB Rbd, #data	<table border="1" style="margin-bottom: 10px; text-align: center;"> <tr><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>Rbd</td></tr> <tr><td colspan="5">data</td><td colspan="5">data</td></tr> </table>	0	0	0	1	0	0	0	0	0	Rbd	data					data																												
0	0	0	1	0	0	0	0	0	Rbd																																				
data					data																																								
XORL RRd, #data	<table border="1" style="margin-bottom: 10px; text-align: center;"> <tr><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td></tr> <tr><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>RRd</td></tr> <tr><td colspan="10">data (high)</td></tr> <tr><td colspan="10">data (low)</td></tr> </table>	0	1	1	1	0	1	0	0	0	0	0	1	0	0	0	0	1	0	0	1	0	0	0	RRd	data (high)										data (low)									
0	1	1	1	0	1	0	0	0	0	0	1	0																																	
0	0	0	1	0	0	1	0	0	0	RRd																																			
data (high)																																													
data (low)																																													

Source Addressing Mode	Assembler Language Syntax	Instruction Format											
IR:	XOR Rd, @Rs ¹ XORB Rbd, @Rs ¹	<table border="1"> <tr> <td>00</td> <td>00100</td> <td>W</td> <td>Rs ≠ 0</td> <td>Rd</td> </tr> </table>	00	00100	W	Rs ≠ 0	Rd						
00	00100	W	Rs ≠ 0	Rd									
XORL RRd, @Rs ¹	<table border="1"> <tr> <td>01111010</td> <td>0000</td> <td>0010</td> </tr> <tr> <td>00</td> <td>001001</td> <td>Rs ≠ 0</td> <td>RRd</td> </tr> </table>	01111010	0000	0010	00	001001	Rs ≠ 0	RRd					
01111010	0000	0010											
00	001001	Rs ≠ 0	RRd										
EAM:	XOR Rd, eam XORB Rbd, eam	<table border="1"> <tr> <td>01</td> <td>00100</td> <td>W</td> <td>eam</td> <td>Rd</td> </tr> <tr> <td colspan="5">1, 2, or 3 extension words</td></tr> </table>	01	00100	W	eam	Rd	1, 2, or 3 extension words					
01	00100	W	eam	Rd									
1, 2, or 3 extension words													
XORL RRd, eam	<table border="1"> <tr> <td>01111010</td> <td>0000</td> <td>0010</td> </tr> <tr> <td>01</td> <td>001001</td> <td>eam</td> <td>RRd</td> </tr> <tr> <td colspan="5">1, 2, or 3 extension words</td></tr> </table>	01111010	0000	0010	01	001001	eam	RRd	1, 2, or 3 extension words				
01111010	0000	0010											
01	001001	eam	RRd										
1, 2, or 3 extension words													

Example: If register RL3 contains %C3 (11000011) and the source operand is the immediate value %7B (0111011), executing the instruction

XORB RL3,#%7B

leaves the value %B8 (10111000) in RL3.

Note 1: Word register in compact mode, longword register in segmented or linear modes.

6.6 EPA Instruction Templates

There are seven templates for EPA instructions. If the Extended Processing Architecture enable bit (EPA) in the Flag and Control Word is set when the CPU encounters one of the instruction templates, the CPU transfers the instruction and operands to the EPU. The CPU merely transfers the operands to the EPU, but does not process them in any way.

Each type of EPU has its own mnemonics, opcodes, and exceptions to represent its particular data processing operations. The shaded portions of the instruction template shown below are ignored by the CPU; they are used by an EPU to specify its particular operations. The two least-significant bits of the first word of the instruction templates are reserved to encode an identifier field

that selects one of up to four possible EPUs in the system. When an EPU detects an exception, it signals the CPU through one of the interrupt request pins. For examples of EPU mnemonics, opcodes, and exceptions, see the Z8070 Floating Point Processor Technical Manual (Zilog document 03-8226-01).

The instruction templates shown below correspond to the data transfer operations performed by the CPU. Data can be transferred between an EPU and memory, EPU and CPU general-purpose registers, or between an EPU and the CPU flags byte register. The last template is for EPU internal operations that require no data transfers.

Extended Instruction

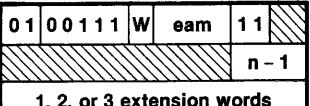
Load Memory from EPU

Operation: Memory \leftarrow EPU (n bytes or words)

The CPU calculates the effective address and generates transactions on the external interface for an EPU to write n words or bytes of data to memory. The value in the instruction field for the number of words or bytes loaded ("n") is one less than the actual value of the source operand. Thus, the coding in the instruction field ranges from 0 to 15, which corresponds to loading 1 to 16 words or bytes.

Flags: No flags affected.

Exceptions: Extended Instruction trap

Destination Addressing Mode	Operation	Instruction Format
IR:	$\text{@Rd} \leftarrow \text{EPU}$	 n - 1
EAM:	$\text{EPU} \leftarrow \text{eam}$	 1, 2, or 3 extension words

Note 1: Word register in compact mode, longword register in segmented or linear modes.

Extended Instruction

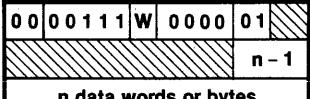
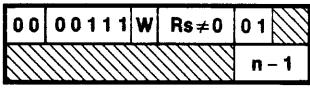
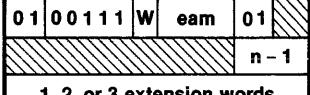
Load EPU from Memory

Operation: EPU \leftarrow Memory (n bytes or words)

The CPU calculates the effective address and generates transactions on the external interface to read n words or bytes of data from memory to an EPU. The value in the instruction field for the number of words or bytes loaded ('n') is one less than the actual value of the source operand. Thus, the coding in the instruction field ranges from 0 to 15, which corresponds to loading 1 to 16 words or bytes. When Immediate addressing mode is used for an odd number of bytes, an extra byte containing 0s is included at the end of the instruction, making the instruction length an integral number of words.

Flags: No flags affected.

Exceptions: Extended Instruction trap

Source Addressing Mode	Operation	Instruction Format
IM:	EPU \leftarrow # data	 n data words or bytes
IR:	EPU \leftarrow @Rs ¹	 n - 1
EAM:	EPU \leftarrow eam	 1, 2, or 3 extension words

Note 1: Word register in compact mode, longword register in segmented or linear modes.

Extended Instruction

Load CPU from EPU

Operation: CPU \leftarrow EPU registers (n words)

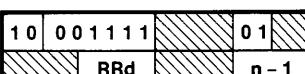
The contents of n words are transferred from an EPU to consecutive CPU registers starting with the specified destination register. The value in the instruction field for the number of words loaded ("n") is one less than the actual value of the source operand. Thus, the coding in the instruction field ranges from 0 to 15, which corresponds to loading 1 to 16 words.

For the word operand version, the CPU word registers (R0 – R15) are loaded. R0 follows R15 in consecutive order.

For the longword operand version, the CPU longword registers (RR0 – RR30) are loaded. RR0 follows RR30 in consecutive order. If the number of loaded words is odd, then the low-order half of the last longword register loaded is undefined after executing this instruction.

Flags: No flags affected.

Exceptions: Extended Instruction trap

Destination Addressing Mode	Operation	Instruction Format
R:	Rd \leftarrow EPU	
	RRd \leftarrow EPU	

Extended Instruction

Load EPU from CPU

Operation: EPU \leftarrow CPU registers (n words)

The contents of n words are transferred to an EPU from consecutive CPU registers starting with the specified source register. The value in the instruction field for the number of words loaded ("n") is one less than the actual value of the source operand. Thus, the coding in the instruction field ranges from 0 to 15, which corresponds to loading 1 to 16 words.

For the word operand version, the EPU is loaded from CPU word registers (R0 – R15). R0 follows R15 in consecutive order.

For the longword operand version, the EPU is loaded from CPU longword registers (RR0 – RR30). RR0 follows RR30 in consecutive order. If the number of loaded words is odd, then the low-order word of the last longword register is not involved in the loading.

Flags: No flags affected.

Exceptions: Extended Instruction trap

Source Addressing Mode	Operation	Instruction Format
R:	EPU \leftarrow Rs	
	EPU \leftarrow RRd	

Extended Instruction

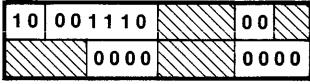
Load FCW from EPU

Operation: Flags \leftarrow EPU

The flags in the CPU's Flag and Control Word are loaded with information from an EPU. Only the flag bits are loaded; bits 0 and 1 of the Flag and Control Word are unaffected.

Flags: Flags loaded from EPU.

Exceptions: Extended Instruction trap

	Operation	Instruction Format
	FCW \leftarrow EPU	

Extended Instruction

Load EPU from FCW

Extended Instruction

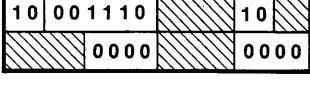
Load EPU from FCW

Operation: EPU \leftarrow Flags

The flag byte of the CPU's Flag and Control Word is transferred to an EPU.

Flags: No flags affected.

Exceptions: Extended Instruction trap

	Operation	Instruction Format
	EPU \leftarrow FCW	

Extended Instruction

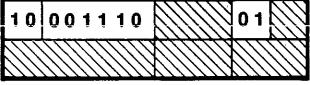
Internal EPU Operation

Operation: Internal EPU Operation

This template is for an EPU internal operation, one which requires no data transfers.

Flags: No flags affected.

Exceptions: Extended Instruction trap

Instruction Format		
		

Chapter 7. Instruction Execution and Exceptions

7.1 INTRODUCTION

To execute an instruction, the CPU fetches the instruction whose address is in the Program Counter (PC), increments the PC by the length of the instruction, and performs the operations specified in Chapter 6 for the particular instruction. Exceptions are conditions or events that alter the sequence of instruction execution. The CPU recognizes four types of exceptions: reset, bus error, interrupts, and traps.

A reset exception occurs when the RESET line is activated. Reset initializes the CPU. A bus error exception occurs when external hardware indicates an irrecoverable error during a data transfer on the external interface. An interrupt is an asynchronous event indicated when the NMI, VI, or NI line is activated. Interrupts are typically caused by peripheral devices that require attention. A trap occurs synchronously when a particular condition, such as integer overflow, is detected by the CPU during instruction execution.

When an exception occurs, the CPU stores the Program Status on the system stack, fetches the new Program Status from the Program Status Area, and resumes executing instructions. This chapter describes instruction execution and exception processing.

7.2 OPERATING STATES

The CPU is always in one of four possible operating states regarding instruction execution and exception processing: reset, exception processing, instruction executing, or halted. Figure 7-1 shows the four states and the transitions between them.

The CPU enters the reset state from any other state when a reset request is signalled on the RESET line. When RESET is released, the CPU enters exception processing state. The reset state is described in more detail in Section 8.10.

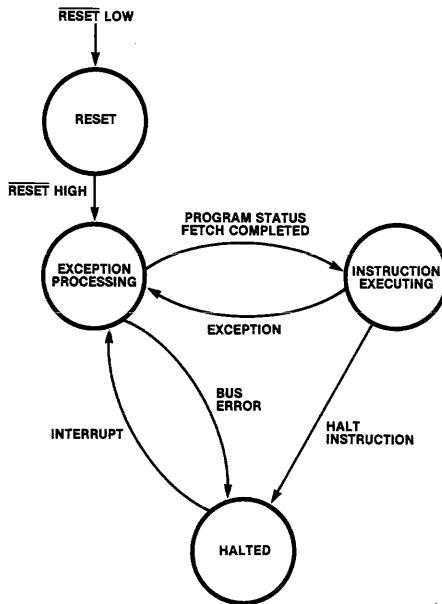


Figure 7-1. Operating States

In the exception processing state, the CPU is either storing values from the Program Status registers to memory or fetching values from memory for the Program Status registers. The storing and fetching of Program Status is described in Section 7.4.5. From the exception processing state the CPU normally enters the instruction executing state; however, a bus error exception causes a transition to the halted state.

In the instruction executing state, the CPU executes instructions. When the Halt instruction is executed, the CPU enters the halted state. If an exception other than reset occurs, the CPU enters the exception processing state.

In the halted state the CPU is halted; it is neither executing instructions nor processing exceptions. When an interrupt occurs, the CPU enters the exception processing state.

Instruction Execution and Exceptions

7.3 INSTRUCTION EXECUTION

Executing an instruction involves the following operations:

- Fetch the instruction
- Increment PC
- Fetch operands, if necessary
- Calculate results
- Store results and flags, if necessary

In concept, the CPU executes instructions by performing all the operations listed above in strict sequence for one instruction, and then beginning execution of the next instruction. However, the CPU checks for exceptions at several points during instruction execution. An exception can alter the operations for an instruction currently being executed, as well as the sequence from one instruction to the next. Also, the CPU overlaps the operations for executing several instructions in a multiple-stage pipeline. That is, while the CPU is calculating the results for one instruction, it can be storing the results for the previous instruction and fetching the operands for the next instruction. The use of an instruction pipeline, rather than completely executing each instruction in strict sequence, enhances the performance of the CPU.

This section describes the effects of exceptions and the pipeline on instruction execution. Section 7.3.1 explains how different exceptions affect instruction execution, and Section 7.3.2 explains how the pipeline affects instruction execution.

7.3.1 Instruction Ending

Instruction execution can end in any of five ways: completion, suspension, suspension with PC modification, termination, or partial completion. Generally, an instruction ends in completion; however, exceptions can cause a different conclusion. Section 7.4 explains each exception recognized by the CPU, and refers to the different types of instruction endings described here.

When an instruction ends in completion, the CPU has completely executed the instruction and all previous instructions. Any result operands and flags modified by the instruction have been stored, and the PC holds the address of the next instruction to execute. If an exception occurs after an instruction ends in completion, the Program Status saved on the system stack can be restored using the Interrupt Return (IRET) instruction. Execution will then resume with the

next instruction in sequence following the completed instruction.

When an instruction ends in suspension or suspension with PC modification, the CPU has not completely executed the instruction, but all previous instructions have been completed. Any flags and destination operands due to be stored by the instruction may be modified; however, only modifications that allow the instruction to be completed are possible. Also, an instruction that ends in suspension or suspension with PC modification will not have modified any control registers, memory locations, or peripheral ports that are protected from access in the current operating mode.

Examples:

1. An Add (ADDB) instruction modifies the flags, but does not examine the flags. If an ADDB instruction ends in suspension because of an address translation exception, the flags may be modified.
2. A Load (LD) instruction can store into a register whose contents are required for an effective address calculation, e.g., LDL RR2, @RR2. If the LD instruction ends in suspension because of an address translation exception, the register contents are unmodified.

When an instruction ends in suspension, the PC holds the address of the first word of the instruction. When an instruction ends in suspension with PC modification, the PC holds the address of the word following the first word of the instruction.

An instruction ends in suspension, or suspension with PC modification, when the CPU detects a trap condition, such as an address translation exception or unimplemented instruction, before completely executing the instruction. An instruction ending in suspension can be completed by eliminating the trap condition and restoring the Program Status saved on the system stack using the IRET instruction. An instruction ending in suspension with PC modification can be completed by eliminating the trap condition, decrementing the PC value stored on the system stack by two using the mode of address representation in effect for the suspended instruction, and restoring the Program Status using the IRET instruction.

When an instruction ends in termination, the CPU has not completely executed the instruction, but all previous instructions have been completed. Any flags and destination operands due to be stored by the instruction may be modified; the

contents of PC are undefined. A terminated instruction will not have modified any control registers, memory locations, or peripheral ports that are protected from access in the current operating mode. It is not possible to complete an instruction that ends in termination. Only reset and bus error cause instruction termination.

Only interruptible instructions can end in partial completion. Interruptible instructions are the "repeat" versions of block transfer, string manipulation, and input/output instructions (Sections 6.2.8 and 6.2.9). Interruptible instructions are repeatedly executed until a specified data value is found for one of the operands, or a counter held in a register is decremented to zero. While the CPU is executing an interruptible instruction, if an Address Translation trap or interrupt occurs; the instruction ends in partial completion. Any flags and destination operands due to be stored by the instruction may be modified; however, the values stored in the counter and address registers allow the instruction to be completed correctly when the instruction is re-executed. The PC holds the address of the first word of the instruction. An instruction ending in partial completion can be completed by eliminating the cause of the exception and restoring the Program Status saved on the system stack using the IRET instruction.

7.3.2 Effects of the Pipeline on Execution

The CPU executes several instructions simultaneously in a multiple-stage pipeline. In most circumstances, the differences between pipelined instruction execution and the complete execution of each instruction in strict sequence cannot be detected by software or hardware. However, the few cases in which the effects of the pipeline can be detected are described below.

The CPU can prefetch an instruction before completing all previous instructions. Consequently, if an instruction stores to a location from which a subsequent instruction is fetched (i.e., the program modifies itself), the CPU can prefetch the original contents of the memory location rather than the modified contents. Thus, self-modifying programs may not operate as intended. On the external interface, instruction prefetching can have the effect of fetching an instruction that is not executed (e.g., if the previous instruction causes a trap) or fetching an instruction before the operands for a previous instruction are fetched. Some privileged instructions (IRET, LDCTL, LDCTLL, LDPS, PCACHE, PTLB, PTLBE, and PTLBN) have the effect of serializing instruction

execution. The serializing instruction and all previous instructions are completely executed, including storing of all results and flags, before fetching the next instruction. Thus, when a new value is loaded into the FCW by a LDCTL instruction, the address representation mode and operating mode used to fetch and execute the next instruction are determined by the new FCW value.

The CPU can also prefetch an operand for an instruction before completing all previous instructions. The effects of operand prefetching cannot be detected by software because the CPU only fetches an operand from a location after completing all previous instructions that modify the location. On the external interface, operand prefetching can have the effect of fetching an operand for an instruction that is not executed, for example, if the previous instruction causes a trap. Operands in physical I/O space are not prefetched, ensuring that the CPU only fetches data from an input peripheral port for instructions that are executed.

7.4 EXCEPTIONS

The CPU recognizes four types of exceptions: reset, bus error, interrupts, and traps. In processing exceptions other than reset, the CPU saves the Program Status and an identifier word on the system stack. For some exceptions, the CPU saves an additional longword parameter. Then the CPU fetches a new Program Status from the Program Status Area. The sections below describe the cause of each exception, CPU response to exceptions, and priority among exceptions.

7.4.1 Reset

Reset occurs when the RESET line is Low. Reset causes any instruction in execution to end in termination.

At reset the Translation and Cache Enable bits of the System Configuration Control Longword register (NX, SX, CI, and CD) are cleared to 0. Some fields of the Hardware Interface Control register are initialized as described in Section 8.10. When the RESET line is driven High, the CPU fetches the FCW from physical memory address 2 and the PC from physical memory address 4. Reset also invalidates all entries in the cache and the Translation Lookaside Buffer. After reset, the contents of all CPU registers other than the FCW, the PC, and the specified fields of SCCL and HICR are undefined. Reset should be used to initialize the CPU at power-on.

Instruction Execution and Exceptions

7.4.2 Bus Error

Bus error is indicated by a device responding to a data transfer transaction on the external interface. A bus error causes any instruction in execution to end in termination. The identifier word saved during bus error exception processing reports the state of the CPU pins. The physical address for the transaction is saved as a parameter on the system stack. Refer to Section 8.8.8 for more details about the bus error exception.

7.4.3 Interrupts

The CPU recognizes three kinds of interrupt signalled on separate pins: non-maskable, vectored, and non-vectored. Non-maskable interrupts are always enabled. Vectored and non-vectored interrupts can be selectively enabled by bits VIE and NVIE in the FCW. Vectored interrupts are enabled when VIE is 1; non-vectored interrupts are enabled when NVIE is 1.

An interrupt occurs when an enabled interrupt request is signalled on a CPU pin. The CPU generates an interrupt acknowledge transaction on the external interface to fetch the identifier word, which is then saved on the system stack. For vectored interrupts, the low-order byte of the identifier word is used to select a pointer to a particular interrupt handler routine. Refer to Section 8.7.5 for more details about interrupt request and acknowledge.

7.4.4 Traps

The CPU recognizes ten traps, described below.

7.4.4.1 Extended Instruction Trap. This trap occurs when an Extended Processing Architecture instruction is executed and the EPA bit of the FCW is 0. The instruction ends in suspension with PC modification. The identifier is the first word of the instruction. This trap allows software to simulate execution of the EPA instruction when no EPU is in the system.

7.4.4.2 Privileged Instruction Trap. This trap occurs when a program attempts to execute a privileged instruction in normal mode; the instruction ends in suspension with PC modification. The identifier is the first word of the instruction.

7.4.4.3 System Call Trap. This trap occurs when a System Call instruction is executed. The

instruction ends in completion; the identifier is the instruction word. This trap is used by programs executing in normal mode to request services from the operating system. The low-order byte of the instruction word indicates the particular service requested.

7.4.4.4 Address Translation Trap. This trap occurs when an address translation error is detected, either an invalid table entry or an access protection violation. The instruction ends in suspension. The identifier word reports the address space for the logical address and the exception type (see Section 4.3.5 for more information). The logical address that caused the translation error is saved as a parameter on the system stack.

7.4.4.5 Breakpoint Trap. This trap occurs when the Breakpoint instruction is executed. The instruction ends in completion; the identifier is the instruction word.

7.4.4.6 Integer Arithmetic Error Trap. This trap occurs when any of three error conditions is detected during execution of integer arithmetic instructions. The error conditions are integer overflow, bounds check, and index error. Integer overflow error is enabled by the IV bit in the FCW. Integer overflow is detected when the IV bit is 1 and the V flag is set by execution of ADD, DEC, DECI, DIV, DIVU, INC, INCI, NEG, SUB, SDA, SRA, SLA, CVT, or CVTU instructions. For DIV and DIVU instructions, Integer Overflow error includes the case of zero divisor. A bounds check error is detected when a Check instruction is executed and the destination operand is out of bounds. An index error is detected when an Index instruction is executed and the subscript is out of bounds.

The instruction ends in completion. The identifier word indicates the type of error, as shown in the following table.

Identifier	Error
0	Integer Overflow
1	Bounds Check
2	Index Error

7.4.4.7 Conditional Trap. This trap occurs when a Trap instruction is executed and the tested condition is satisfied. The instruction ends in completion; the identifier is the instruction word. This trap can be used for software detection of run-time errors.

7.4.4.8 Unimplemented Instruction Trap. This trap occurs when a program attempts to execute an instruction with an unimplemented bit pattern. The detected bit patterns include certain Z8000 opcodes described in Appendix A and instructions with first byte 36₁₆, or BF₁₆. The instruction ends in suspension with PC modification; the identifier is the first word of the instruction.

7.4.4.9 Odd PC Trap. This trap occurs before execution of an instruction when the PC contains an odd address. The contents of the identifier word are undefined.

7.4.4.10 Trace Trap. This trap occurs before an instruction is executed when the TP bit in the FCW is 1. The contents of the identifier word are undefined.

Instruction tracing is enabled by the T bit in FCW. Before each instruction is executed, T is copied to TP. The use of two bits to control instruction tracing ensures that, while tracing is enabled, exactly one Trace trap is processed after each instruction's execution, and after the servicing of other traps and interrupts. Section 7.4.7 provides more information about the priority for handling Trace traps and other exceptions.

The Trace trap handler should set the T bit to 1 and clear the TP bit to 0 in the FCW on the system stack before executing IRET and returning to the traced program. Note that the T bit in the FCW on the system stack can be cleared when an IRET, LDCTL, or LDPS instruction is traced.

7.4.5 Changing Program Status

To process all exceptions other than reset, the CPU pushes the Program Status and an identifier word on the system stack. An Address Translation trap and bus error push an additional longword parameter onto the system stack. The saved value of the PC depends on the type of instruction ending. As selected by the XL/S bit in the System Configuration Control Longword (SCCL) register, the CPU operates in either segmented system mode (XL/S = 0) or linear system mode (XL/S = 1) while saving the Program Status and other information; but the saved value of the FCW indicates the mode of operation when the exception occurred. Figure 7-2 shows how the information is saved on the stack.

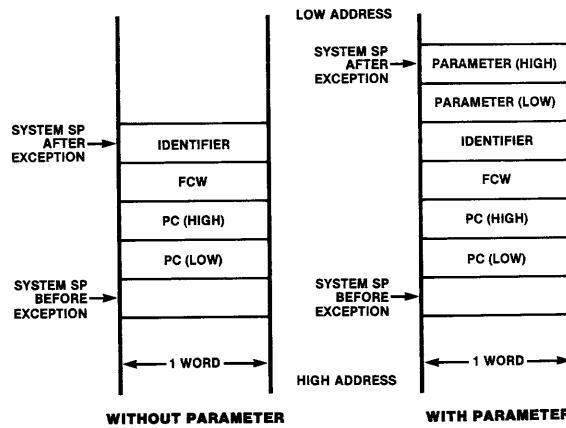


Figure 7-2.
Program Status Saved on System Stack

Instruction Execution and Exceptions

A new Program Status must be fetched from memory to process any exception. For reset, the FCW is fetched from physical address 2 and the PC is fetched from physical address 4. Other exceptions fetch the new Program Status from an entry in the Program Status Area (PSA) (Figure 7-3). Bus error, non-maskable interrupt, non-vectored interrupt, and all traps have unique entries in the PSA

from which the new Program Status is fetched. For vectored interrupts, the new value of the FCW is loaded from displacement 122 in the PSA. The low-order byte of the identifier word is used to select the new value of the PC by indexing into a table of 256 values beginning at displacement 124 in the PSA.

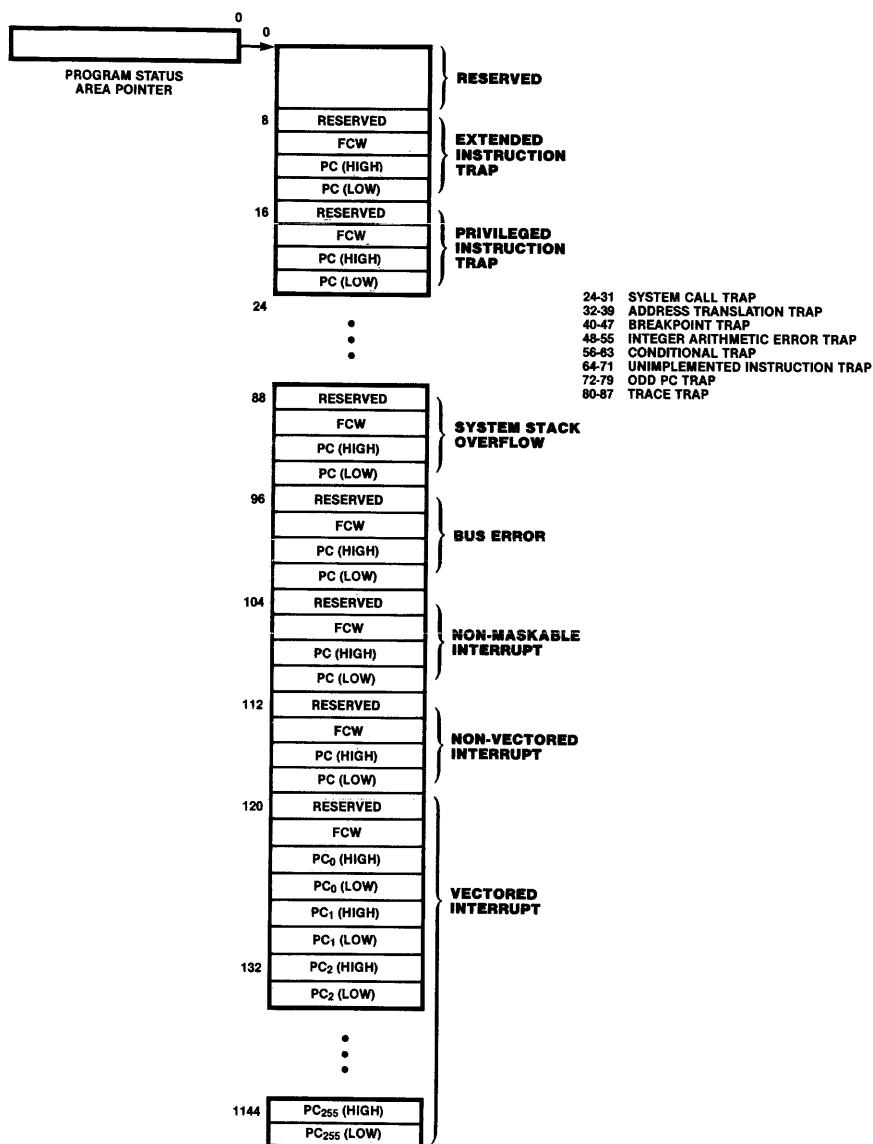


Figure 7-3. Program Status Area

The effective address of an entry in the Program Status Area is calculated by adding the displacement shown in Figure 7-3 to the physical base address held in the Program Status Area Pointer register. The effective address calculation is performed in segmented or linear mode, as selected by the XL/S bit in the SCCL register. The result is the physical address used to fetch the PSA entry.

During exception processing, if an address translation error is detected while information is being saved on the system stack, the System Stack Pointer is restored to its value before the exception occurred and the overflow stack is used instead. The top of the overflow stack is

addressed by the Overflow Stack Pointer register (OSP). The Program Status, identifier word, and exception parameter (or an undefined longword if there is no exception parameter) are pushed on the overflow stack. A word containing the displacement of the exception entry in the PSA is also pushed onto the overflow stack. The new Program Status is fetched from displacement 88 in the PSA. Since the OSP register contains a physical address, an Address Translation trap cannot occur when pushing information on the overflow stack. The effective address calculation for pushing onto the overflow stack is performed in segmented or linear mode, as selected by the XL/S bit in the SCCL register. Figure 7-4 shows how information is saved on the overflow stack.

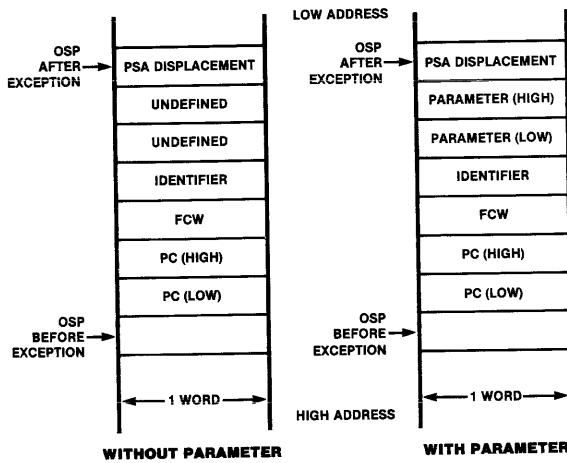


Figure 7-4.
Program Status Saved on Overflow Stack

7.4.6 Exception Handlers

After the new Program Status has been fetched, the CPU begins executing instructions of the exception handler routine whose address was loaded into the PC. The new value of the FCW determines the address representation mode (compact/segmented/linear), operating mode (system/normal), and the enabled interrupts and traps for the exception handler. An interrupt handler can execute with interrupts disabled until critical information has been stored. The interrupt handler can then enable interrupts, permitting nested interrupt servicing.

The exception handler can examine the identifier word and parameter (only bus error and Address Translation trap have a parameter) for information about the cause of the exception. After completing their service, handlers for traps and interrupts execute the Interrupt Return instruction. The Address Translation trap handler must pop the longword violation address from the stack before executing IRET. IRET restores the Program Status from the system stack so instruction execution can resume at the point where the exception occurred. The handlers for Extended Instruction trap, Privileged Instruction trap, and Unimplemented Instruction trap must modify the PC value stored on the stack before executing IRET.

Instruction Execution and Exceptions

7.4.7 Priority of Exceptions

It is possible for several exceptions to occur simultaneously. The CPU checks for particular

exceptions at specific points during instruction execution. (Figure 7-5.) If multiple exceptions are detected, the CPU responds to the one with highest priority.

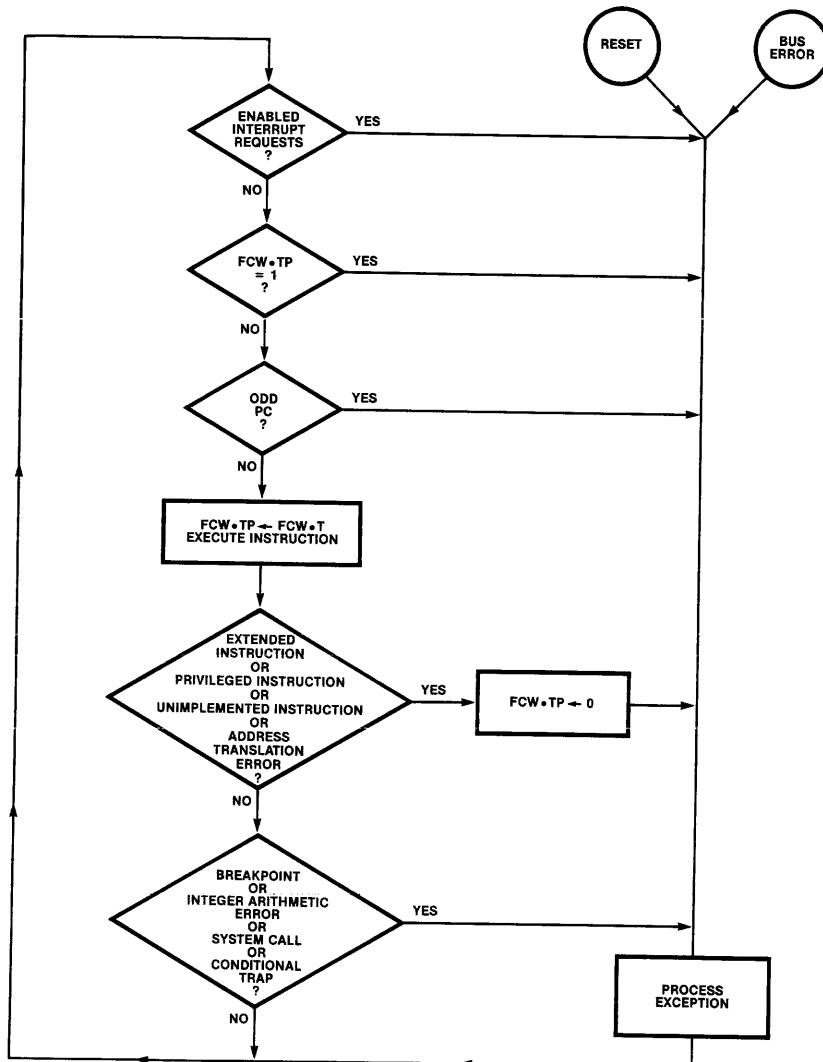


Figure 7-5. Exception Priority Flowchart

Whenever a reset exception is detected, the CPU responds immediately; any instruction being executed is terminated. Pending bus errors, traps, and internally latched non-maskable interrupt requests are eliminated.

If a bus error is detected and reset is not requested, the CPU responds to the bus error exception. Any instruction being executed is terminated, and pending traps are eliminated.

Before executing an instruction, the CPU checks for enabled interrupt requests. The CPU responds to the highest priority enabled interrupt request, if any. The priority of interrupts is, in descending order, nonmaskable, vectored, and nonvectored. If several devices are requesting the same interrupt, priority among the devices must be resolved externally, typically with a daisy chain or interrupt priority controller. After responding to an interrupt, the new value of FCW is used to check again for enabled interrupt requests before executing the first instruction of the service routine.

If there are no enabled interrupt requests, the CPU checks the TP bit in the FCW. If TP is set to 1, a Trace trap occurs. Otherwise, the CPU checks whether the PC contains an odd address. If the least-significant bit of PC is 1, an Odd PC trap occurs. Otherwise, the CPU copies T to TP and begins executing the instruction.

During instruction execution, one of the following trap conditions may be detected: Extended Instruction trap, Privileged Instruction trap, Unimplemented Instruction trap, or Address Translation trap. If one of the conditions is detected, instruction execution is suspended; TP is cleared to 0; and the trap is processed. Otherwise, instruction execution is completed.

After completion of the instruction, one of four trap conditions may be detected: System Call trap, Breakpoint trap, Integer Arithmetic Error trap, or Conditional trap. If one of these trap conditions is detected, the corresponding trap is processed.

For interruptible instructions, the CPU checks for address translation exceptions during each iteration. If an address translation exception is detected, instruction execution ends in partial completion, TP is cleared to 0, and the trap is processed. If no address translation error has been detected, the CPU checks for enabled interrupt requests at the end of each iteration except the last. If an interrupt request is pending, the CPU clears TP to 0 and responds to the highest priority request.

An interrupt can occur immediately after the Enable Interrupt instruction is executed and before the next instruction.



Chapter 8. External Interface

8.1 INTRODUCTION

The CPU is only one component in a computer system containing memory, peripherals, Extended Processing Units (EPUs), DMA controllers, and other CPUs (Figure 8-1). Zilog has established the Z-BUS as a convention for the signals and timing used to

interconnect components of a microprocessor system. The Z80,000 CPU is compatible with the Z-BUS, allowing the CPU to be easily connected into a wide variety of system configurations. This chapter describes the operation of the CPU interface with other system components.

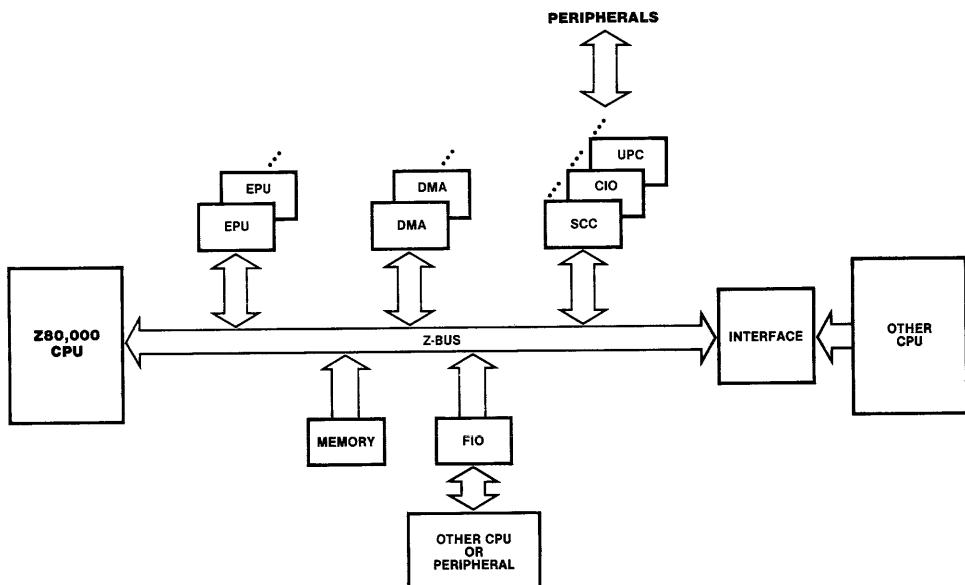


Figure 8-1. System Configuration

8.2 BUS OPERATIONS

Two kinds of bus operations are defined: transactions and requests. At any one time, only one device, known as the master, has control of the bus. The master can initiate transactions on the bus to transfer data to another device, known as the responder. In some transactions, called flyby, the master controls the transaction, but another device transfers data with the responder. The master can also initiate transactions that do not transfer data. The CPU performs transactions that transfer data to and from memory, peripherals, or EPUs.

The CPU controls flyby transactions that transfer data between an EPU and memory. The CPU also performs internal operation and halt transactions, which do not transfer data. Only the bus master can initiate transactions; however, other devices can initiate requests. The CPU responds to interrupt requests from peripherals by generating an interrupt acknowledge transaction. The CPU responds to bus requests from other potential bus masters, and can initiate bus requests of its own, as described in Section 8.9. In addition, the CPU responds to reset requests, which are used to initialize the CPU.

External Interface

8.3 MULTIPROCESSOR CONFIGURATIONS

The CPU provides support for interconnection in four types of multiprocessor configurations (Figure 8-2): coprocessor, slave processor, tightly-coupled multiple CPUs, and loosely-coupled multiple CPUs.

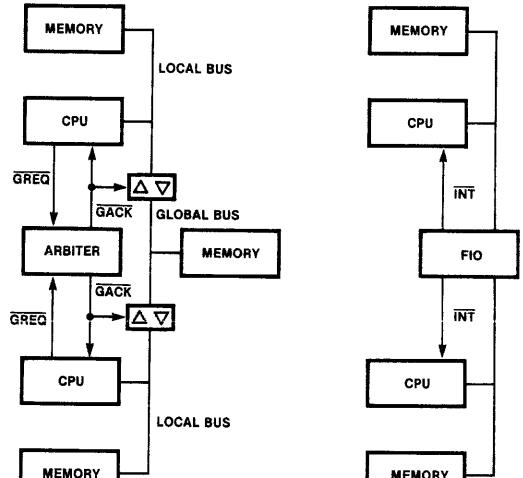
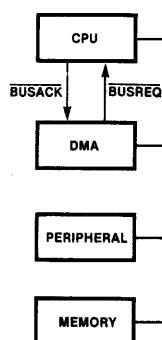
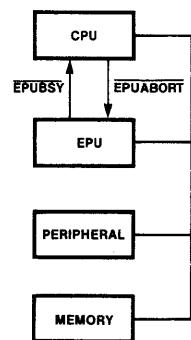
Coprocessors, such as the Z8070 Arithmetic Processing Unit, work synchronously with the CPU to execute a single instruction stream using the Extended Processing Architecture facility. The EPUBSY and EPUABORT signals are dedicated for connection with coprocessors, as described in Section 8.8.4.

Slave processors, such as the Z8016 DMA Transfer Controller, perform dedicated functions asynchronously to the CPU. The CPU and slave processor share a local bus, of which the CPU is the default

master, using the BUSREQ and BUSACK signals, as described in Section 8.9.

Tightly-coupled, multiple CPUs execute independent instruction streams and communicate through shared memory located on a common (global) bus using the GREQ and GACK signals, as described in Section 8.9. Each CPU is default master of its local bus, but the global bus master is chosen by an external arbiter. The CPU also provides special bus status information for interlocked memory references (Test and Set, Increment Interlocked, and Decrement Interlocked instructions), which can be used with multiple-ported memories.

Loosely-coupled, multiple CPUs generally communicate through a multiple-ported peripheral, such as the Z8038 FIO I/O Interface Unit. The Z80,000 CPU's I/O and interrupt facilities can support loosely-coupled multiprocessing.



(A) COPROCESSOR

(B) SLAVE PROCESSOR

(C) TIGHTLY-COUPLED MULTIPLE CPU

(D) LOOSELY-COUPLED MULTIPLE CPU

Figure 8-2.
Multiprocessor Configurations

8.4 CACHE

The CPU implements a cache mechanism that keeps a copy of recently used memory locations on-chip. These locations can contain both instructions and data. On memory fetches, the CPU examines the cache to determine if the addressed information is stored there. If the information is in the cache (a hit), then the CPU fetches the copy from the cache, and no transaction is necessary on the external interface. If the information is not in the cache (a miss), then the CPU performs a memory read transaction to fetch the missing information and stores a copy of the information into the cache, replacing the least recently used data in the cache. Thus, the cache serves to reduce the number of memory read transactions, providing a substantial boost to performance.

Software can control the cache mechanism in several ways. The System Configuration Control Longword register contains separate control bits (CI and CD) that enable the cache for instruction and data references and another bit (CR) that enables the cache replacement algorithm. In page table entries, the NC bit can be set to disable the use of the cache for selected pages. The Purge Cache instruction can be executed to invalidate the contents of the cache when a memory location that may have been copied into the cache has been modified by another processor. For example, if a slave processor reads from a peripheral port to a memory location that may be copied in the cache, the cache must be purged. Similarly, if two or more tightly-coupled CPUs can alternately execute one process, the cache must be purged when the operating system changes from executing one user-process to another. Appendix C describes the cache mechanism in more detail, including its control and interaction with the external interface.

8.5 PIN FUNCTIONS

The CPU interface includes 59 signal lines, and four power supply connections (Figure 8-3). A summary of the signal pin functions is given below.

AD₀-AD₃₁. **Address/Data (Bidirectional, active High, 3-state).** These 32 lines are time-multiplexed to transfer address and data. At the beginning of each transaction the lines are driven with the 32-bit address. After the address has been driven, the lines are used to transfer one or more bytes, words, or longwords of data.

AS. **Address Strobe (Output, active Low, 3-state).** The rising edge of AS indicates the beginning of a transaction and shows that the address, ST₀-ST₃, R/W, BL/W, BW/L, N/S, and BRST are valid.

BRST. **Burst (Output, active Low, 3-state).** A Low on this line indicates that the CPU is performing a burst transfer; that is, multiple Data Strobes following a single Address Strobe.

BRSTA. **Burst Acknowledge (Input, active Low).** A Low on this line indicates that the responding device can support burst transfers.

BUSREQ. **Bus Request (Input, active Low).** A Low on this line indicates that a bus requester has obtained or is trying to obtain control of the local bus.

BUSACK. **Bus Acknowledge (Output, active Low).** A Low on this line indicates that the CPU has relinquished control of the local bus in response to a bus request.

BL/W, BW/L. **(Output, 3-state).** These two lines specify the data transfer size.

BL/W	BW/L	Size
High	High	Byte
Low	High	Word
High	Low	Longword
Low	Low	Reserved

CLK. **Clock (Input).** This line is the clock used to generate all CPU timing.

DS. **Data Strobe (Output, active Low, 3-state).** DS is used for timing data transfers.

EPUBSY. **EPU Busy (Input, active Low).** A Low on this line indicates that an EPU is busy. This line is used to synchronize the operation of the CPU with an EPU during execution of an EPA instruction.

EPUABORT. **EPU Abort (Output, active Low).** A Low on this line indicates that the CPU is aborting execution of an EPA instruction, typically because an Address Translation trap has occurred.

GACK. **Global Acknowledge (Input, active Low).** A Low on this line indicates that the CPU has been granted control of a global bus.

GREQ. **Global Request (Output, active Low, 3-state).** A Low on this line indicates that the CPU has obtained or is trying to obtain control of a global bus.

External Interface

IE. Input Enable (Output, active Low, 3-state). A Low on this line can be used to enable buffers on the AD lines to drive toward the CPU.

NMI. Non-Maskable Interrupt (Input, edge activated). A High-to-Low transition on this line requests a non-maskable interrupt.

NVI. Non-Vectored Interrupt (Input, active Low). A Low on this line requests a non-vectored interrupt.

N/S. Normal/System Mode (Output, Low = System Mode, 3-state). This line indicates whether the CPU is operating in normal or system mode.

OE. Output Enable (Output, active Low, 3-state). A Low on this line can be used to enable buffers on the AD lines to drive away from the CPU.

R/W. Read/Write (Output, Low = Write, 3-state) This line indicates the direction of data transfer.

RESET. (Input, active Low). A Low on this line resets the CPU.

RSP₀-RSP₁. Response (Input). These lines encode the response to transactions initiated by the CPU. RSP₀ and RSP₁ can be connected together for Z-BUS WAIT timing.

RSP ₀	RSP ₁	Response
High	High	Ready
Low	High	Bus Error
High	Low	Bus Retry
Low	Low	Wait

ST₀-ST₃. Status (Output, active High, 3-state). These lines encode the kind of transaction occurring on the bus. (See Table 8-1.)

VI. Vectored Interrupt (Input, active Low). A Low on this line requests a vectored interrupt.

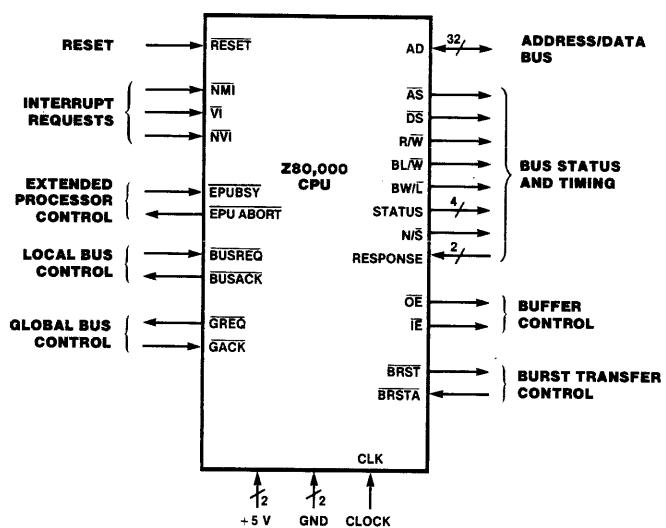


Figure 8-3. Z80,000 Pin Functions

8.6 HARDWARE INTERFACE CONTROL REGISTER

The Hardware Interface Control register (HICR) specifies certain characteristics of the hardware configuration surrounding the CPU, including bus speed, memory data path width, and number of automatic wait states. The physical memory address space is divided into two sections, M₀ and M₁,

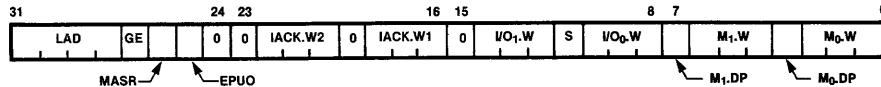


Figure 8-4. Hardware Interface Control Register

M₀ Wait Count (M₀.W) specifies the number of wait states automatically inserted by the CPU for references to M₀. If the value is 0, no wait states are inserted. If the value is n>0, n wait states are automatically inserted for memory read and n-1 wait states are inserted for memory write.

M₀ Data Path Width (M₀.DP) specifies the data path width for references to M₀. While this bit is 1, the data path width for M₀ is 16 bits; otherwise, the data path width for M₀ is 32 bits.

M₁ Wait Count (M₁.W) specifies the number of wait states automatically inserted by the CPU for references to M₁. If the value is 0, no wait states are inserted. If the value is n>0, then n wait states are automatically inserted for memory read and n-1 wait states are inserted for memory write.

M₁ Data Path Width (M₁.DP) specifies the data path width for references to M₁. While this bit is 1, the data path width for M₁ is 16 bits; otherwise, the data path width for M₁ is 32 bits.

I/O₀ Wait Count (I/O₀.W) specifies the number (0-7) of wait states automatically inserted by the CPU for references to I/O₀.

I/O₁ Wait Count (I/O₁.W) specifies the number (0-7) of wait states automatically inserted by the CPU for references to I/O₁.

Interrupt Acknowledge Wait Count 1 (IACK.W1) specifies the number (0-7) of wait states automatically inserted by the CPU before D_S falls during interrupt acknowledge transactions.

Interrupt Acknowledge Wait Count 2 (IACK.W2) specifies the number (0-7) of wait states automatically inserted by the CPU before D_S rises during interrupt acknowledge transactions.

selected by bit 30 of the memory address. A typical system would locate slow, 16-bit wide bootstrap ROM in M₀ and faster, 32-bit wide dynamic RAM in M₁. The physical I/O address space is similarly divided into two sections, I/O₀ and I/O₁, selected by bit 30 of the port address. The fields of HICR (Figure 8-4) are described below.

Speed (S) specifies the frequency of the bus clock relative to the processor clock. If this bit is 1, the bus clock frequency is 1/2 the processor clock frequency; otherwise, the bus clock frequency is 1/4 the processor clock frequency. The value of this bit is determined by hardware at reset, and cannot be altered by software (see Section 8.10).

EPU Overlap Mode (EPUO) and another field in an EPU control register control the degree of overlap for CPU and EPU operations. While this bit is 1, overlap is enabled; otherwise, overlap is disabled. While overlap is disabled, the EPU can use the signal EPUBSY to stop the CPU from processing instructions. There are several degrees of overlap that affect performance, system debugging and recovery from exceptions. Refer to Section 8.8.4 for more information.

Minimum Address Strobe Rate (MASR) controls an option that ensures an Address Strobe is generated at least once every 16 bus clock cycles. While this bit is 1, the option is enabled; otherwise, the option is disabled. While the MASR option is enabled and the CPU has neither performed any transactions, granted the local bus, nor requested a global bus for 16 bus cycles, the CPU performs an internal operation or halt transaction. If the CPU is in halted state, a halt transaction is performed; otherwise, an internal operation transaction is performed. This function can be used for refreshing pseudostatic RAMs. Also, some Z-BUS peripherals require Address Strobe to generate interrupt request timing.

Global Enable (GE) and Local Address (LAD) control the use of the global bus request protocol. While GE is 1, the protocol is enabled; otherwise, the protocol is disabled. The LAD field selects 1 of 16 sections of the physical address spaces used for references to the local bus; references to other sections use the global bus. See Section 8.9 for more information.

External Interface

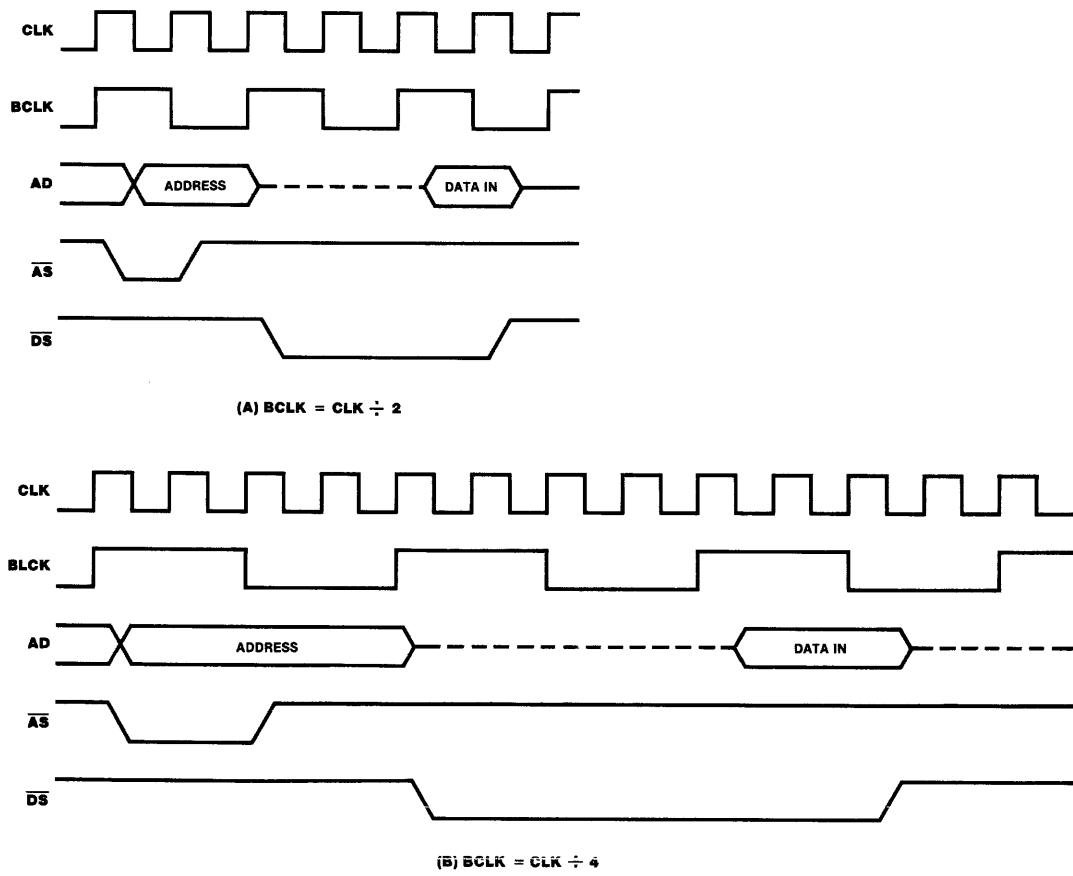
In systems that combine memories with different widths, an individual operand must be located entirely within physical memory modules of a single width. Thus if an operand is located across consecutive logical pages, including operands for ENTER, EXIT, LDM, LDML, and EPA instructions that may occupy several longwords, then the two physical frames containing the operand must both be in 16-bit memory modules or 32-bit memory modules.

8.7 BUS TIMING

The CPU performs transactions on the external interface to transfer data for fetching instructions, fetching and storing operands, processing exceptions, and performing memory management. In addition, the CPU performs internal operation and halt transactions, which do not transfer data. Each transaction occurs during

a sequence of bus clock cycles, named T_1 , T_2 , etc. The CPU has a single clock line, CLK, used to generate all timing. Internally, the CPU derives another clock for bus timing by dividing CLK by 2 or 4. The scale factor for bus timing (2 or 4) is selected at reset. In the AC timing characteristics for the CPU (available in a separate data sheet from Zilog), input setup and hold times and output delays are specified with respect to a rising edge of CLK. When CPU output transitions occur on different rising clock edges, the time between the transitions is specified in terms of a constant delay and a variable number of CLK cycles. The number of CLK cycles depends on the bus timing scale factor, type of transaction, and number of wait states.

In the logical timing diagrams that follow, the signal transitions on the bus are shown in relation to the bus clock, BCLK. The beginning of a transaction, signified by a falling edge of \overline{AS} ,



**Figure 8-5. Example of Memory Read Timing
Showing Different Bus Scale Factors**

always occurs on a rising edge of BCLK. The BCLK signal is derived internally to the CPU as described above, and is not available on the pins. BCLK can also be derived externally by dividing CLK by the selected bus timing scale factor. Section 8.10 discusses synchronization of the internal and external bus clocks. The timing diagrams in Figure 8-5 show example memory read transactions with one wait state using the different scale factors.

8.8 BUS TRANSACTIONS

All bus transactions begin with Address Strobe (\overline{AS}) first asserted* and then negated. On the rising edge of \overline{AS} , the lines for status (ST_0-ST_3), Read/Write (R/W), data transfer size (BW/L , BL/W), and Normal/System (N/S) are valid. The status lines indicate the type of transaction being initiated (Table 8-1). The R/W line indicates the direction of data transfer. The data transfer size indicates whether a byte, word, or longword of data is to be transferred. The N/S line indicates the CPU's operating mode. The following sections describe timing for the different transactions.

Table 8-1. Status Codes

ST_3-ST_0	Definition
0 0 0 0	Internal Operation
0 0 0 1	CPU-EPU (data)
0 0 1 0	I/O
0 0 1 1	Halt
0 1 0 0	CPU-EPU (Instruction)
0 1 0 1	NMI Acknowledge
0 1 1 0	NVT Acknowledge
0 1 1 1	VI Acknowledge
1 0 0 0	Cacheable CPU-Memory (Data)
1 0 0 1	Non-Cacheable CPU-Memory (Data)
1 0 1 0	Cacheable EPU-Memory
1 0 1 1	Non-Cacheable EPU-Memory
1 1 0 0	Cacheable CPU-Memory (Instruction)
1 1 0 1	Non-Cacheable CPU-Memory (Instruction)
1 1 1 0	Reserved
1 1 1 1	Interlocked CPU-Memory (Data)

*In the description of bus transactions, the term "asserted" means an active signal and "negated" means an inactive signal. A signal is either active when High or when Low, as specified in the pin function list.

On the rising edge of \overline{AS} , the address on the AD lines is also valid. Addresses are not required for internal operation, halt, interrupt acknowledge, and CPU-EPU data transactions; the AD lines are driven but the address is undefined for those transactions. The CPU uses Data Strobe (\overline{DS}) to time the data transfer. (Note that internal operation and halt transactions do not transfer data, and thus do not assert \overline{DS} .) For write operations ($R/W = \text{Low}$), the CPU asserts \overline{DS} when valid data is on the AD lines. For read operations ($R/W = \text{High}$), the CPU makes the AD lines 3-state before asserting \overline{DS} so the addressed device can put its data on the bus. The CPU samples the data in the middle of a bus cycle while negating \overline{DS} .

The AD lines can be used to transfer bytes, words, or longwords of data. When reading from memory, the CPU always reads a word or longword, depending on the memory data path width, regardless of the size of the information required. For read transactions the three cases are handled as follows:

- Byte transfers use AD_0-AD_7 ; AD_8-AD_{31} are ignored.
- Word transfers use AD_0-AD_{15} ; $AD_{16}-AD_{31}$ are ignored.
- Longword transfers use AD_0-AD_{31} .

For write transactions, the three cases are handled as follows:

- Byte transfers replicate the data on AD_0-AD_7 , AD_8-AD_{15} , $AD_{16}-AD_{23}$, and $AD_{24}-AD_{31}$.
- Word transfers replicate the data on AD_0-AD_{15} and $AD_{16}-AD_{31}$.
- Longword transfers use AD_0-AD_{31} .

The Input Enable (\overline{IE}) and Output Enable (\overline{OE}) signals can be used to enable buffers on the bidirectional AD lines. \overline{IE} is asserted when the buffers are to drive toward the CPU; \overline{OE} is asserted when the buffers are to drive away from the CPU. Whenever the direction for the AD lines changes, both \overline{IE} and \overline{OE} are negated for at least one CLK cycle.

To transfer more than one data item, the CPU can perform burst transactions. The data items are transferred in the same direction, and are equal in size. \overline{DS} is used to time each transfer. The CPU asserts Burst ($BRST$) to indicate a burst transfer. The responding device asserts Burst Acknowledge ($BRSTA$) if it is capable of supporting burst transfers. If $BRSTA$ is not asserted, the CPU transfers only a single data item.

External Interface

8.8.1 Response

Any time data is transferred, the responding device returns a code on the Response lines (RSP_0-RSP_1) to indicate ready, wait, bus error, or bus retry. The response is sampled at a time specific for each type of transaction, generally before the AD lines are sampled for reads or \overline{DS} is negated for writes, and after automatic wait states are inserted.

Ready indicates the completion of a successful transfer.

Wait indicates that the responding device needs more time to complete the transaction. The CPU waits one bus cycle before sampling the response again to accommodate slow memory or peripherals. A simple system using only Z-BUS WAIT can be implemented by connecting WAIT to both RSP_0 and RSP_1 .

Bus error indicates that a fatal error has occurred during the transaction, e.g., bus timeout for a nonexistent device. The CPU treats bus error as an exception.

Bus retry indicates that the transaction should be tried again, e.g., a transient parity error was detected. The CPU negates \overline{DS} and tries the transaction again.

The CPU can insert wait states automatically under control of several fields in the Hardware Interface Control register. If an automatic wait state is programmed for a bus cycle, the CPU ignores the response and wait is assumed. Thus, wait states can be inserted automatically by the CPU or upon request of the responding device. It must be emphasized that the RSP_0-RSP_1 lines are sampled synchronously. Thus, they must meet the specified setup and hold times for correct operation.

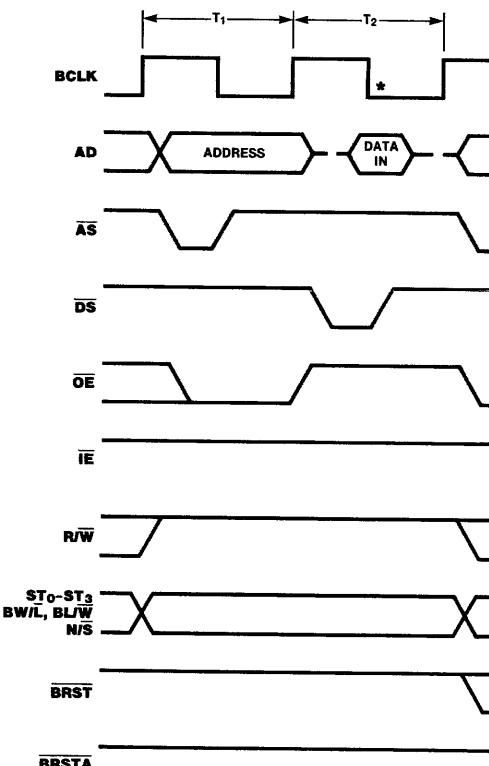
8.8.2 CPU-Memory Transactions

The CPU performs transactions with status 1000, 1001, 1100, 1101, or 1111 to read from and write to memory. See Appendix C for more information about the different status codes. The transactions involve either a single data transfer or multiple, burst data transfers.

8.8.2.1 Single Memory Read and Write Transactions.

Figure 8-6 shows timing for a single memory read transaction with no wait states. \overline{AS} is asserted during the first half of T1. The rising edge of \overline{AS} indicates that the address on AD_0-AD_{31} and control signals ST_0-ST_3 , R/W , BW/L , BL/W , and N/S are valid. The control signals remain valid for the duration of the transaction. $BRST$ is negated during the transaction because only a single data item is transferred. At the beginning of T2, the CPU stops driving the address, asserts \overline{DS} , and prepares to receive data from memory. In the middle of T2, RSP_0-RSP_1 are sampled ready, the input data is latched, and \overline{DS} is negated. The signal \overline{OE} is asserted during T1; however, for two-cycle read transactions, \overline{IE} is not asserted. \overline{IE} is unasserted because there is no bus clock transition between the negation of \overline{OE} at the end of T1 and the sampling of data in the middle of T2. The two-cycle read transaction is a compatible extension of the Z-BUS three-cycle read transaction. Two-cycle read transactions are intended for use with fast memories connected directly to the CPU pins without buffers, such as an external cache.

asserted during the first half of T1. The rising edge of \overline{AS} indicates that the address on AD_0-AD_{31} and control signals ST_0-ST_3 , R/W , BW/L , BL/W , and N/S are valid. The control signals remain valid for the duration of the transaction. $BRST$ is negated during the transaction because only a single data item is transferred. At the beginning of T2, the CPU stops driving the address, asserts \overline{DS} , and prepares to receive data from memory. In the middle of T2, RSP_0-RSP_1 are sampled ready, the input data is latched, and \overline{DS} is negated. The signal \overline{OE} is asserted during T1; however, for two-cycle read transactions, \overline{IE} is not asserted. \overline{IE} is unasserted because there is no bus clock transition between the negation of \overline{OE} at the end of T1 and the sampling of data in the middle of T2. The two-cycle read transaction is a compatible extension of the Z-BUS three-cycle read transaction. Two-cycle read transactions are intended for use with fast memories connected directly to the CPU pins without buffers, such as an external cache.



* RSP_0-RSP_1 and data sampled.

Figure 8-6. Single Memory Read Timing

The CPU can insert wait states in the middle of T2 if RSP₀-RSP₁ are sampled wait or if automatic wait states are programmed in the appropriate field of HICR. The duration of a wait state is one BCLK cycle.

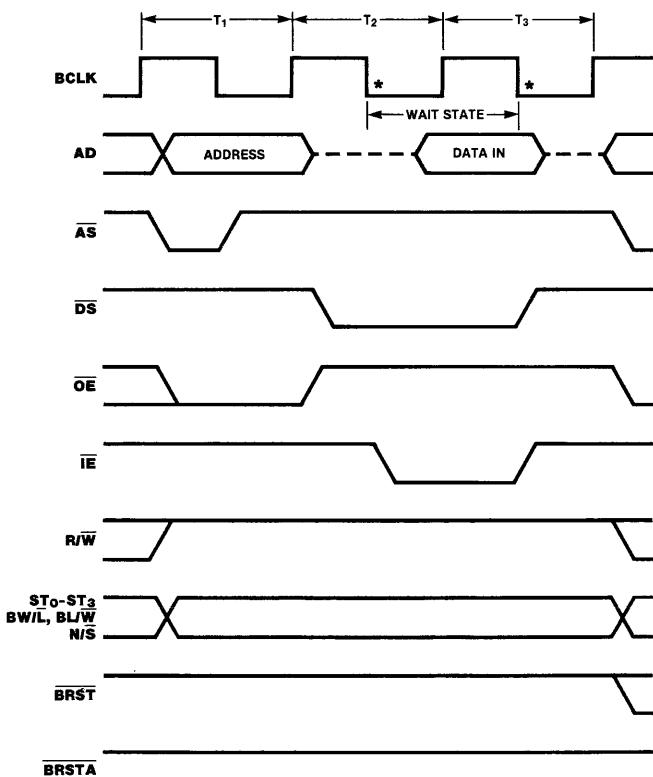
The timing for a single memory read transaction with one wait state is shown in Figure 8-7. This is not a true wait state because the CPU asserts IE in the middle of T2 and continues until the middle of T3. For memory read transactions longer than two bus cycles, either because of wait states or burst transfers, IE is asserted from the middle of T2 until the end of data transfer. The signals OE and IE can be used to control buffers on the AD lines.

For memory read transactions, the data transfer size is equal to the data path width specified in HICR. The memory should transfer the aligned longword addressed by AD₂-AD₃₁ (ignored AD₀-AD₁)

for a 32-bit data path, or the aligned word addressed by AD₁-AD₃₁ (ignoring AD₀) for a 16-bit data path. The CPU selects the required bytes from the transferred word or longword.

A single memory write transaction (Figure 8-8) begins with AS to indicate that address and control signals are valid. At the beginning of T2 the CPU stops driving the address and starts driving the data. In the middle of T2, DS is asserted. The CPU negates DS in the middle of T3. OE is asserted beginning at T1 and continues for the duration of the transaction. The CPU samples RSP₀-RSP₁ in the middle of T3.

For memory write transactions, the data transfer size is less than or equal to the data path width specified in HICR. Bytes and words can be written to a 16-bit memory; bytes, words, and longwords can be written to a 32-bit memory. The CPU writes bytes to any address, but words and longwords are



*RSP₀-RSP₁ and data sampled.

Figure 8-7.
Single Memory Read Timing (One Wait State)

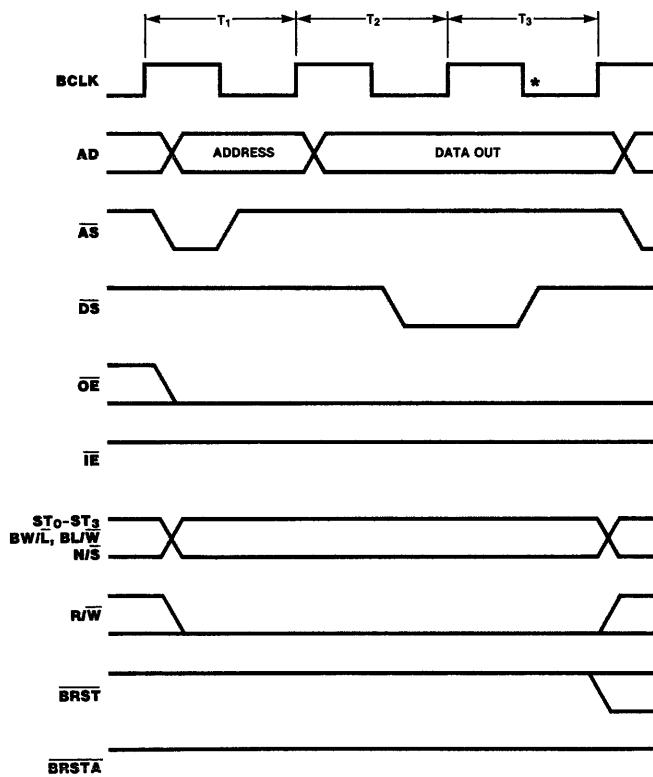
External Interface

always written to an aligned address; that is, words are always written to an even address and longwords are always written to an address that is a multiple of four. When a program writes a word or longword to an unaligned address, the CPU performs two or more write transactions to aligned addresses. For example, if the program writes a word to an odd address, the CPU first writes the more significant byte to the odd address, then it writes the less significant byte to the successive even address.

Single memory read and write timing are slightly different from Z-BUS specifications. The minimum read transaction is two bus cycles, and the response is sampled at the end of the data transfer. For the Z-BUS, the minimum read transaction is three cycles, and the response is sampled one cycle before the end of the data transfer. For strict Z-BUS compatibility it is possible to program one automatic wait state for memory read and to delay the response using an external flipflop.

8.8.2.2 Burst Memory Read and Write Transactions. Burst memory transactions use multiple Data Strobes following a single Address Strobe to transfer data at consecutive memory addresses. The BRST and BRSTA signals control the burst transaction. The CPU uses burst transactions to prefetch the cache block for a cache miss on an instruction fetch. The CPU also uses burst transactions to fetch or store operands when more than one transfer is necessary, as with unaligned operands, string instructions, Load Multiple instructions, and loading of Program Status.

If the memory does not support burst transfers, the burst transfer protocol described below (Figure 8-9) allows BRSTA to be tied High. The CPU then separates the burst transaction into a sequence of single transfers, but only a single transfer is performed for a cache miss on an instruction fetch.



*RSP₀-RSP₁ sampled.

Figure 8-8. Single Memory Write Timing

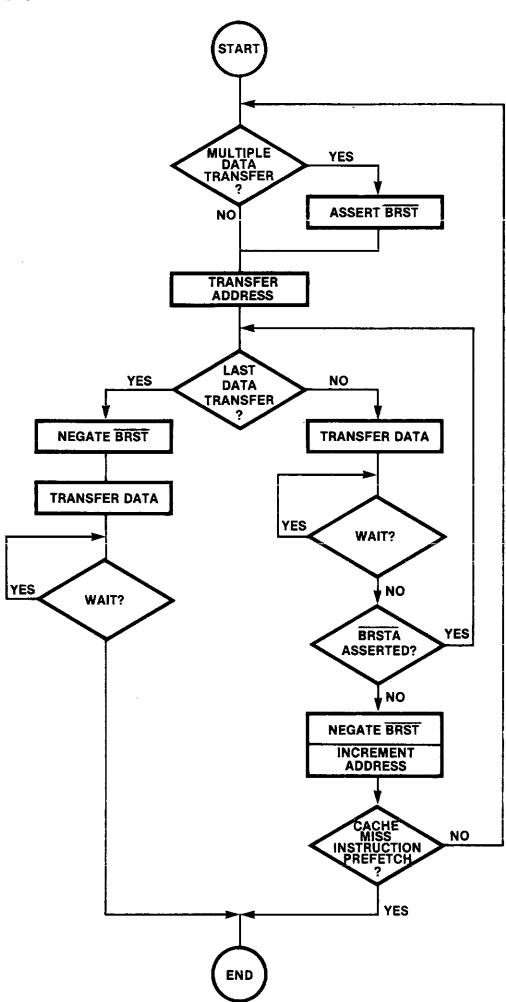


Figure 8-9. Burst Transfer Protocol

At the beginning of a burst transaction, the CPU asserts BRST along with other control signals. If the CPU continues to assert BRST when DS falls, this indicates to memory that the CPU can support another data transfer following the one in process. If the CPU negates BRST before DS falls, this indicates to memory that the current transfer is the last in the transaction.

When BRSTA is asserted at the time the RSP₀-RSP₁ lines are sampled ready, this indicates to the CPU that memory can support another data transfer following the one in process. When BRSTA is negated at the time the RSP₀-RSP₁ lines are sampled ready, this indicates to the CPU that the current data transfer is the last in the transaction. The burst transaction can be terminated by either the

CPU or memory. If memory terminates the transfer by negating BRSTA, the CPU responds by negating BRST when DS is negated. (See the example for burst memory read.) If the CPU terminates the transfer by negating BRST before the falling edge of DS, memory responds by negating BRSTA. (See the example for burst memory write.) The CPU terminates the burst transaction when all the required data items have been transferred or after reaching the end of an aligned, 16-byte block.

Figure 8-10 shows timing for a burst memory read transaction with one wait state. In this example, three data items are transferred, after which memory terminates the burst. BRST is asserted at the beginning of T1; otherwise, the timing for the first transfer is identical to a single memory read. In the middle of T3, the CPU samples RSP₀-RSP₁ ready, latches the data, and samples BRSTA active. During T4 the second data item is transferred, accompanied by DS. The time for the second and subsequent transfers can be extended with wait states if RSP₀-RSP₁ are sampled wait; the CPU inserts automatic wait states only for the first transfer. During T5 the third data item is transferred. At the same time RSP₀-RSP₁ are sampled ready, the data is latched and BRSTA is sampled inactive. Memory terminated the burst transfer, and the CPU responds by negating BRST.

Figure 8-11 shows timing for a burst memory write transaction with no wait states. In this example, two data items are transferred, and the CPU terminates the burst. BRST is asserted at the beginning of T1; otherwise, the timing for the first transfer is identical to a single memory write. In the middle of T3, the CPU samples RSP₀-RSP₁ ready and BRSTA active. At the beginning of T4, the CPU negates BRST, indicating that one more data transfer will follow. During T4, the second data item is transferred, accompanied by DS. The time for the second and subsequent transfers can be extended with wait states if RSP₀-RSP₁ are sampled wait; the CPU inserts automatic wait states only for the first transfer. Memory recognizes that the CPU has terminated the burst transfer, and responds by negating BRSTA before the end of T4. Note that a memory system can be designed to support burst transfers only for read transactions through selective enabling of BRSTA.

8.8.2.3 Interlocked Memory Transactions. In tightly-coupled multiprocessor configurations, the CPU must at certain times inhibit other bus masters from referring to shared memory while the CPU performs two or more interlocked transactions. The CPU uses interlock protection for data references associated with Test and Set, Decrement

External Interface

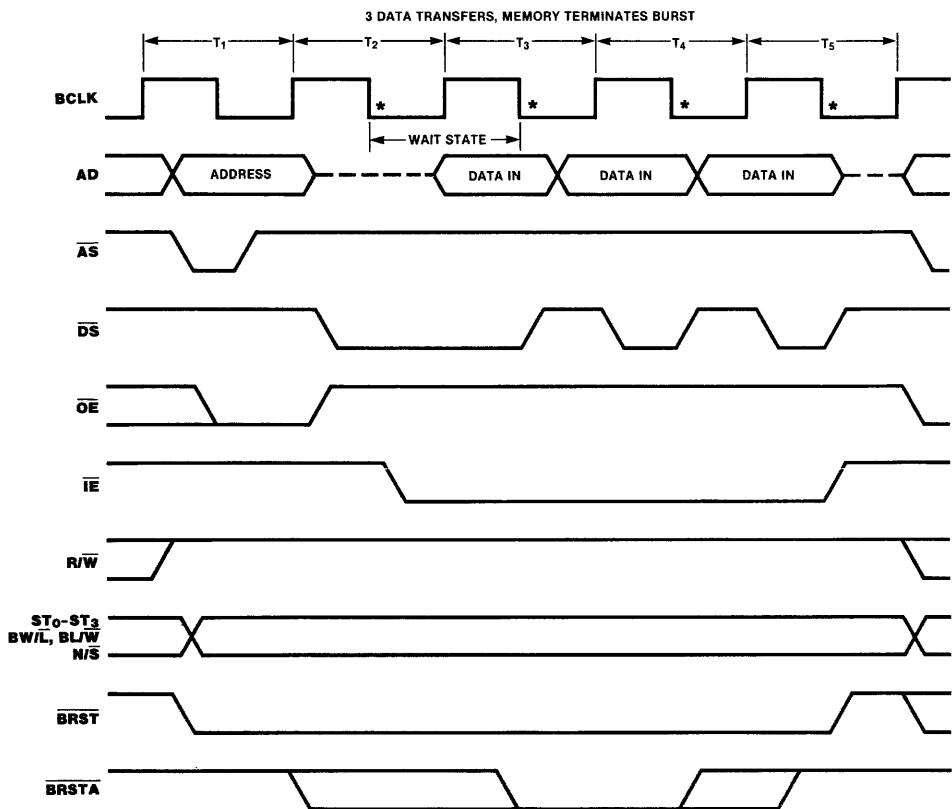


Figure 8-10. Burst Memory Read Timing (One Wait State)

Interlocked, and Increment Interlocked instructions. The CPU also uses interlock protection for references to address translation table entries when loading the Translation Lookaside Buffer. The CPU indicates interlocked protection for a sequence of memory references by using status 1111 for any of the memory transactions previously described. While the CPU indicates status 1111, the memory system must prevent interlocked references to shared memory by other processors. During a sequence of interlocked memory transactions, the CPU does not acknowledge local bus requests nor does the CPU generate any bus transactions with status other than 1111.

8.8.3 Input/Output Transactions

The CPU uses status 0010 to read from and write to I/O ports. I/O transactions are generated for I/O instructions and, when address translation is enabled, by data references to pages with bit 31 of the page table entry set to 1.

The timing for I/O and memory transactions is very similar. The major difference is that DS falls in the middle of T₂ for I/O read timing, compared to the beginning of T₂ for memory read timing. This allows peripheral devices more time for address decoding. Another difference is that the data

External Interface

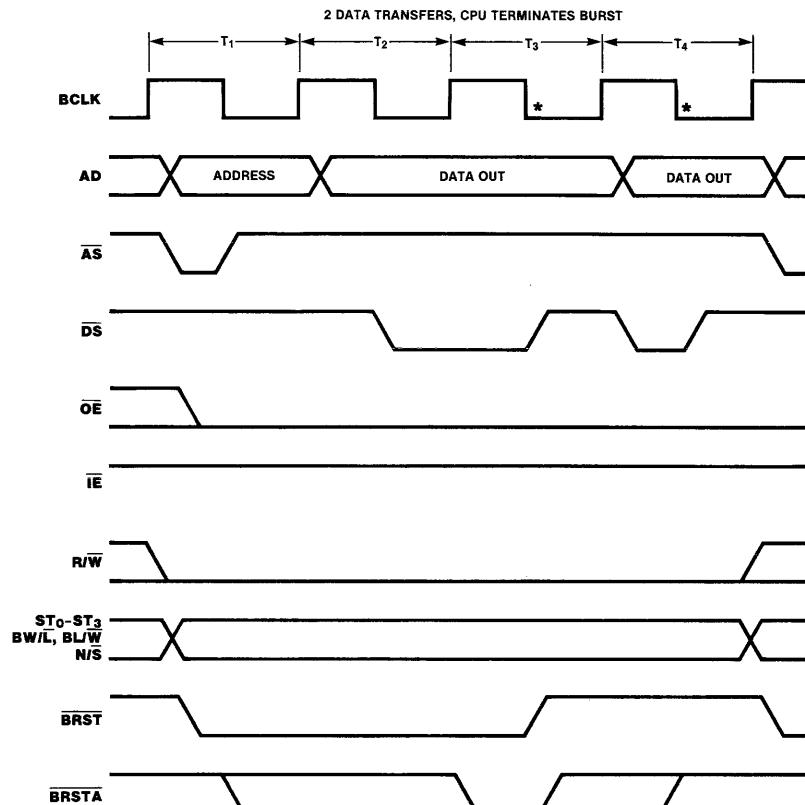


Figure 8-11. Burst Memory Write Timing

transfer size (byte, word, or longword) for I/O transactions is specified by the instruction, not by HICR. The final difference is that the CPU does not support burst I/O transactions. Figure 8-12 shows timing for an I/O read transaction. I/O write timing is the same as a single memory write (Figure 8-8).

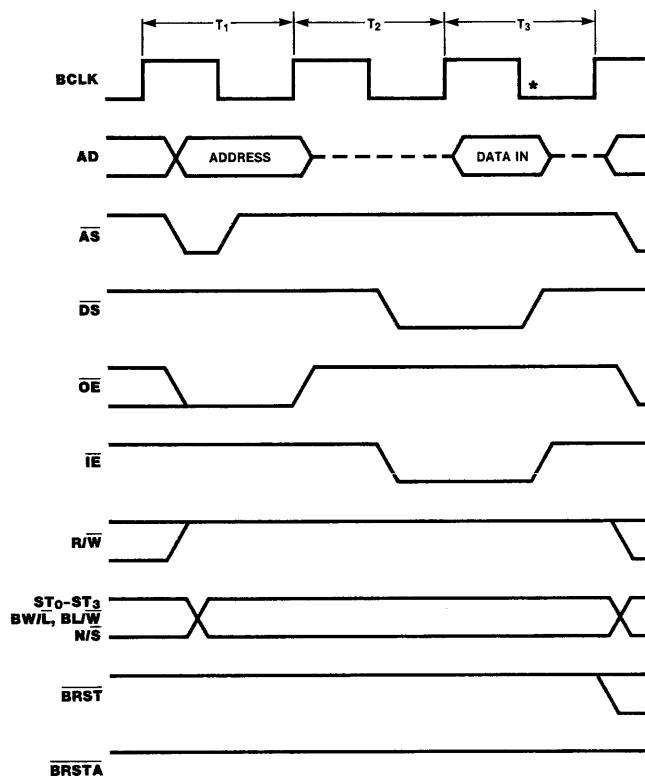
8.8.4 EPU Transactions

The CPU and EPU cooperate in the execution of EPA instructions (Figure 8-13). When the CPU encounters an EPA instruction and the EPA bit in FCW is 1, the CPU broadcasts the first two words of the

instruction to the EPUs in the system using the CPU-EPU instruction transfer transaction. All EPUs in the system recognize the transaction, but only one of four possible EPUs is selected by bits 16 and 17 of the EPU instruction. The CPU also transfers the PC value for the instruction, which the selected EPU saves for use in exception handling. If data transfers are required to complete the instruction, the CPU controls the data transfer transactions while the EPU drives or receives the data.

The EPUBSY signal, output from the EPU, is used to synchronize the CPU and EPU in executing EPA instructions. (When multiple EPUs are present in

External Interface



*RSP₀-RSP₁ and data sampled.

Figure 8-12. I/O Read Timing

a system, the EPUBSY input to the CPU must be driven by an external AND gate whose inputs are the EPUBSY signals from the EPUs. The CPU must sample EPUBSY inactive before initiating an EPU instruction transfer. If data transfers are required, the CPU must sample EPUBSY inactive before initiating the first transfer.

While the CPU samples EPUBSY active, no transactions are initiated; however, the CPU may grant the local bus.

EPUBSY is also used to control the degree of overlap between CPU and EPU instruction execution. Ordinarily, the CPU can continue processing other instructions after performing the data transfers associated with an EPA instruction and before the EPU has completed executing the instruction. To simplify debugging and recovery from exceptions, overlap can be disabled under control of the EPUO bit in HICR. When overlap is disabled (EPUO = 0), the CPU samples EPUBSY in the middle of the bus

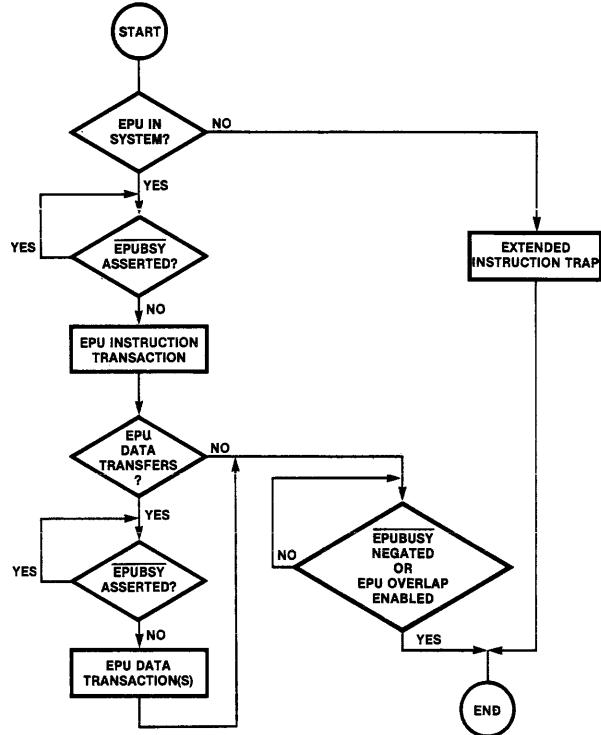


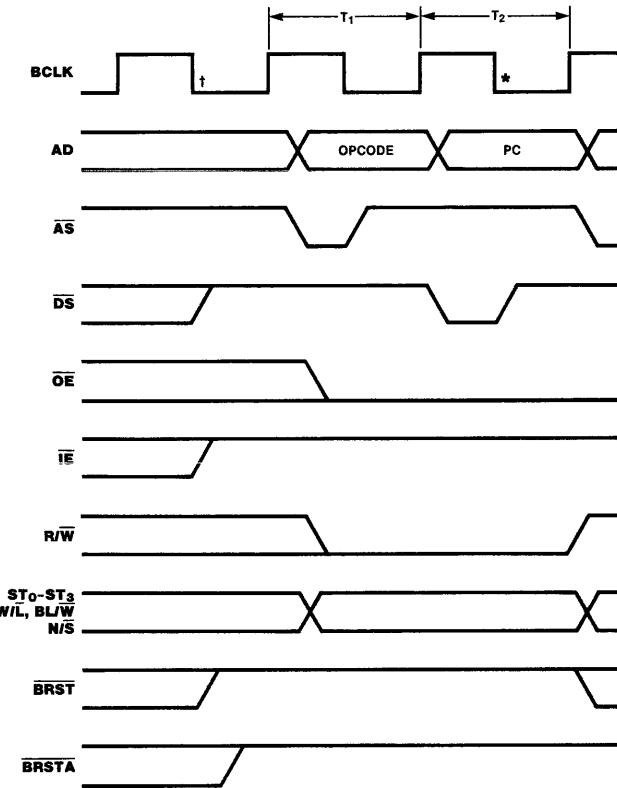
Figure 8-13. EPA Instruction Processing

cycle during which the last data transfer for an EPA instruction occurs. If EPUBSY is asserted, the CPU ceases processing instructions or interrupts until EPUBSY is sampled inactive in the middle of a bus cycle. When overlap is enabled (EPUO = 1), the CPU does not sample EPUBSY after the last data transfer, but only samples EPUBSY before initiating the next EPU instruction transfer.

While processing an EPA instruction and after the instruction has been transferred to the selected EPU, the CPU may detect an address translation exception. In such an event, the CPU asserts EPUABORT, informing the selected EPU to abort execution of the instruction; at all other times, the CPU negates EPUABORT. The CPU then saves the address of the suspended EPA instruction on the system stack during exception processing.

When CPU and EPU instruction processing overlap, the CPU may complete all data transfers for an EPA instruction (the queued instruction) before the EPU completes execution of a previous EPA instruction. If the EPU then detects an exception during execution of the previous instruction, the EPU does not execute the queued instruction. In such a case, the address of the queued instruction is in an EPU control register, and the CPU saves the address of a subsequent instruction on the system stack.

To simplify system hardware, the CPU and EPU AD lines should be wired together with no buffers between them. If the AD lines are separated by buffers, external circuitry must generate IE and OE timing for CPU-EPU data read and EPU-memory write transactions.



[†]EPUBSY sampled.
*RSP₀-RSP₁ sampled; EPUBSY sampled if EPU internal operation.

Figure 8-14.
CPU-EPU Instruction Transfer Timing

8.8.4.1 CPU-EPU Instruction Transactions.
Figure 8-14 shows timing for a CPU-EPU instruction transfer transaction with status 0100. The rising edge of AS indicates that the AD lines and status are valid. During T1, the AD lines are used to transfer the opcode, i.e., the first two words of the EPA instruction. At the beginning of T2 the CPU stops driving the opcode, asserts DS, and starts driving PC on the AD lines. In the middle of T2, the CPU samples RSP₀-RSP₁ ready and negates DS. The data transfer size for the transaction is longword.

The duration of a CPU-EPU instruction or data transfer can be extended with wait states if RSP₀-RSP₁ are sampled wait. The Z8070 APU, however, does not require wait states, nor does it drive RSP₀-RSP₁. Systems using the Z8070 APU must ensure that RSP₀-RSP₁ are both High, indicating ready, during CPU-EPU instruction and data transactions.

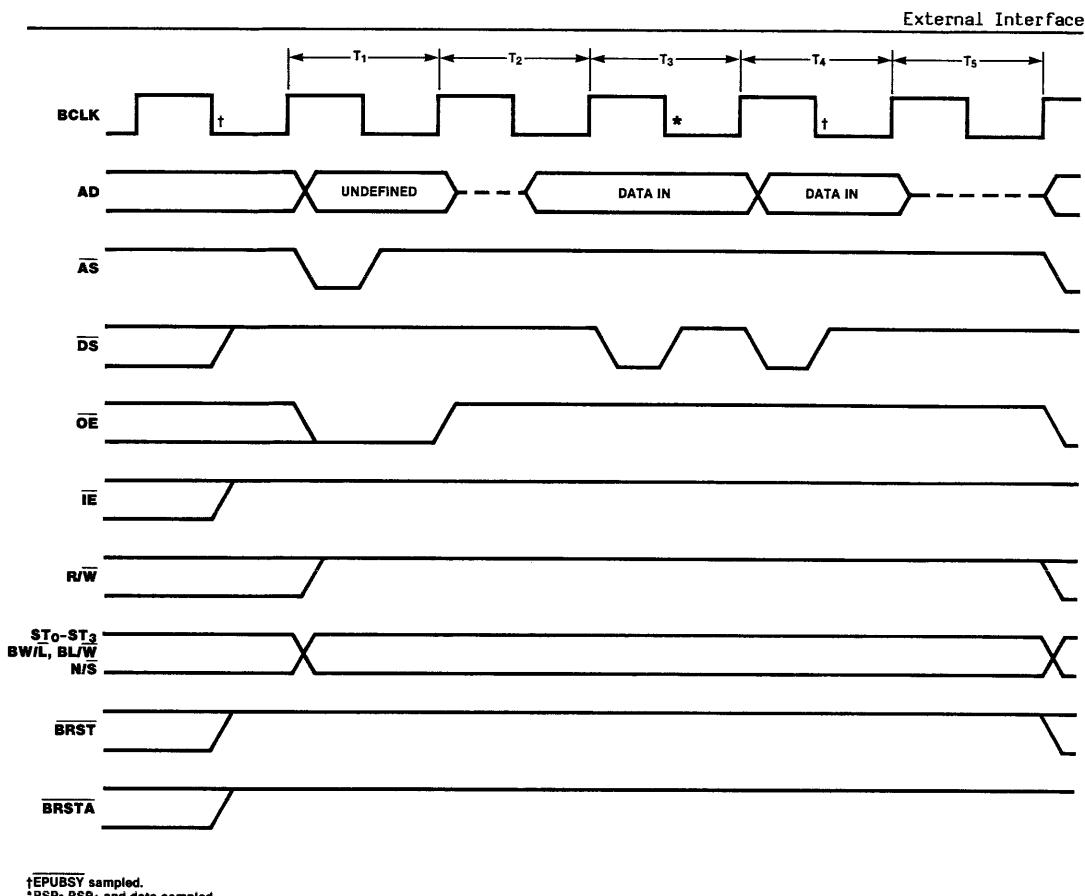


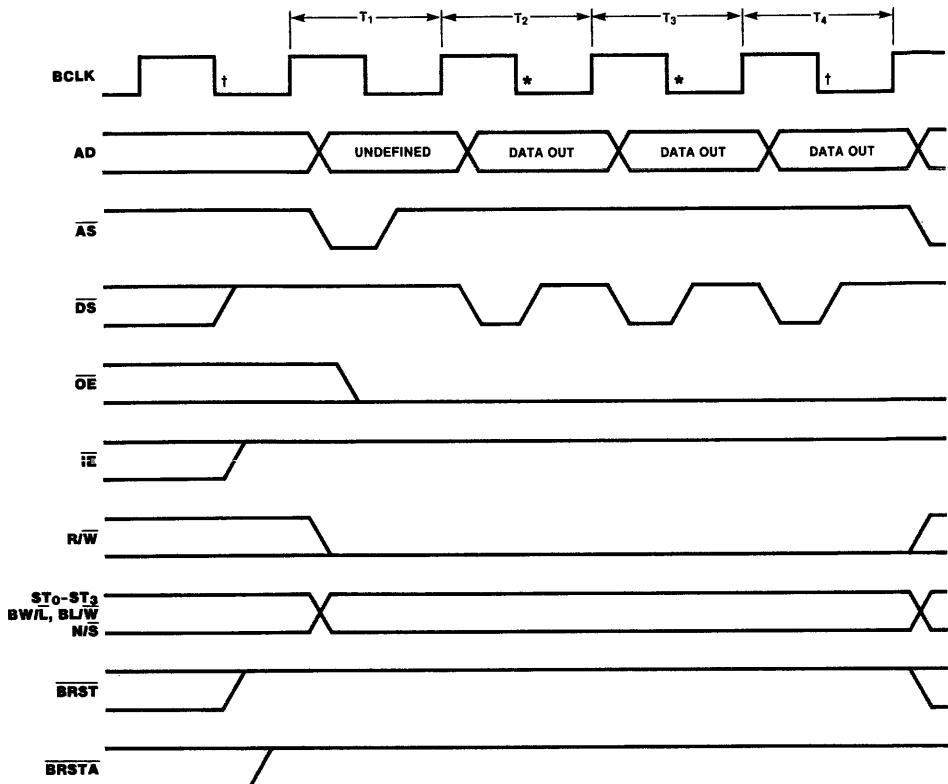
Figure 8-15. CPU-EPU Data Read Timing

8.8.4.2 CPU-EPU Data Transactions. Transactions to transfer data between the CPU and EPU use status 0001. The EPA instruction opcode indicates the number of words transferred. One or more longwords of data are transferred until all words have been transferred. If the last transfer contains a single word, the data is on AD₁₆-AD₃₁. The CPU does not assert BRST and ignores BRSTA.

Figure 8-15 shows timing for a CPU-EPU data read transaction. This example has two data transfers; any number of data transfers between one and eight

is possible. The rising edge of AS indicates that status and control signals are valid. The CPU stops driving the AD lines at the end of T₁; the EPU begins driving them in the middle of T₂. At the beginning of T₃, the CPU asserts DS. In the middle of T₃ the CPU samples RSP₀-RSP₁ ready, latches the data, and negates DS. The second longword of data is transferred during T₄. After the last data transfer the CPU inserts an idle bus cycle (T₅ in the example) during which neither the CPU nor EPU drive the AD lines.

External Interface



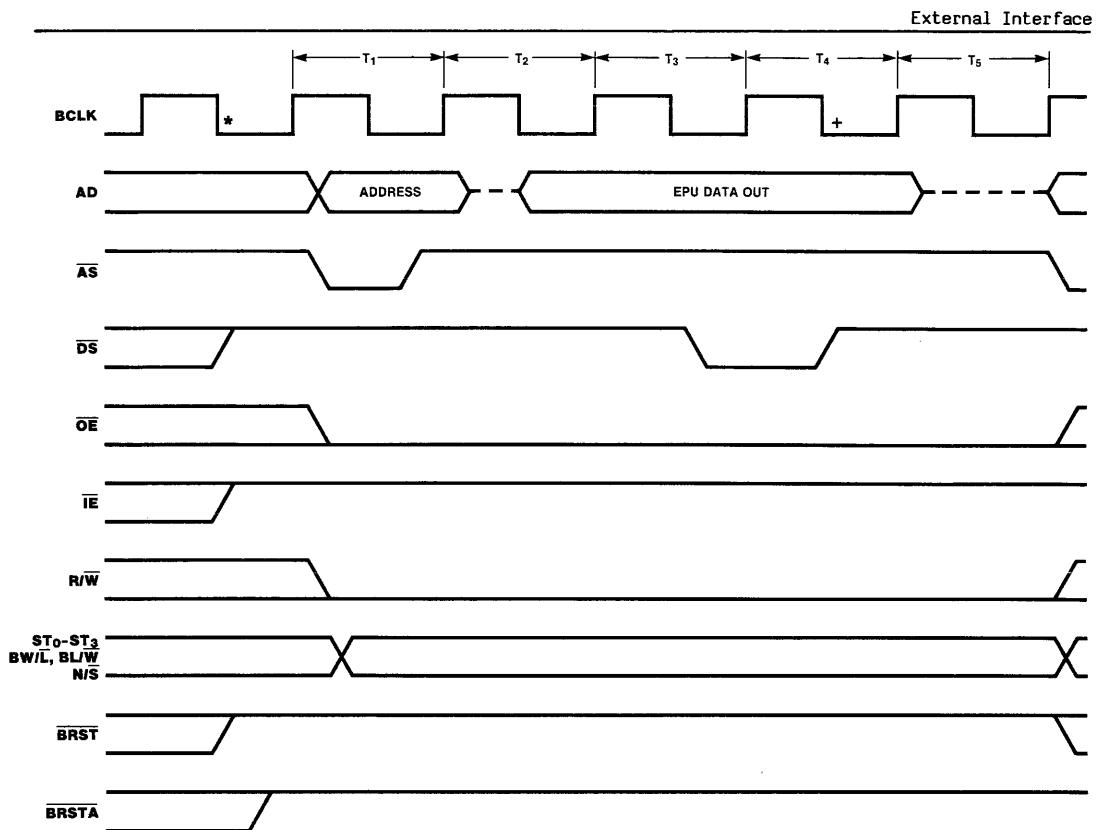
†EPUBSY sampled.

*RSP0-RSP1 sampled.

Figure 8-16. CPU-EPU Data Write Timing

Figure 8-16 shows timing for a CPU-EPU data write transaction. This example has three data transfers; any number of data transfers between one and eight is possible. Timing for the first transfer is identical to the CPU-EPU instruction transfer transaction. A second longword of data is transferred during T₃, and the third longword is transferred during T₄.

8.8.4.3 EPU-Memory Transactions. The CPU uses status 1010 or 1011 for the EPU to read from and write to memory using flyby transactions. The timing is identical for EPU-memory read and CPU-memory read. The EPU monitors the CPU timing on the bus, and uses the two least significant address bits on the first transfer, the data transfer size, and the length of the operand from the instruction to select the bytes it needs from the AD lines.



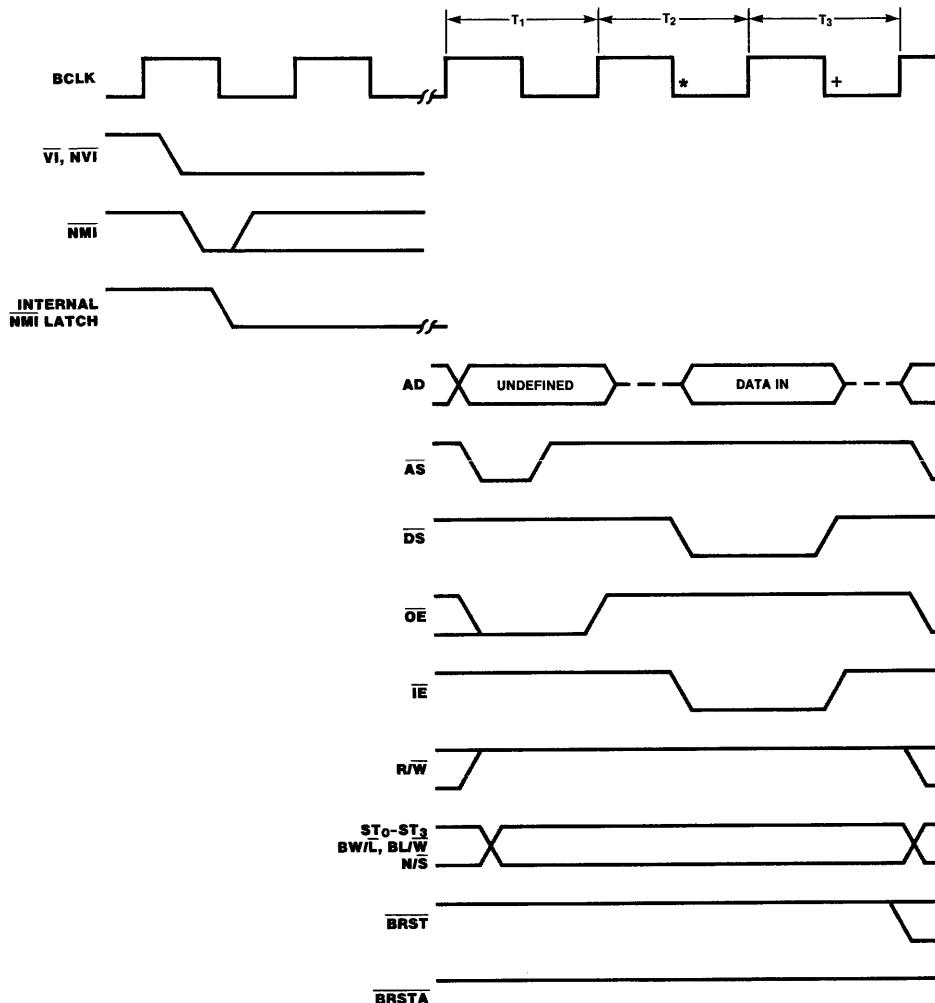
*EPUBSY sampled:
+ RSP₀-RSP₁ sampled; EPUBSY sampled if last transaction.

Figure 8-17. EPU-Memory Single Write Timing

The timing for an EPU-memroy write transaction differs slightly from a CPU-memroy write transaction. Two extra bus cycles are included to pass the AD lines from CPU to EPU after the address transfer and from EPU back to CPU after the last data transfer. Figure 8-17 shows an example for a single EPU-memroy write transaction with no wait states. The CPU stops driving the AD lines at the end of T₁; the EPU begins driving them in the middle of T₂. DS is asserted in the middle of T₃, one bus cycle later than for CPU-memroy write

timing. The CPU negates DS in the middle of T₄. The CPU can insert wait states in the middle of T₄. The EPU continues to drive the AD lines until the end of T₄. After the last data transfer the CPU inserts an idle bus cycle (T₅ in the example) during which neither the CPU nor EPU drive the AD lines. EPU-memroy burst write transactions are similarly extended by two bus cycles more than CPU-memroy burst write timing. One cycle is inserted before the first data transfer, and another after the last data transfer.

External Interface



* RSP₀-RSP₁ sampled.
+ RSP₀-RSP₁ and data sampled.

Figure 8-18. Interrupt Request/Acknowledge Timing

8.8.5 Interrupt Request and Acknowledge

The CPU recognizes vectored, nonvectored, and nonmaskable interrupt requests. The decreasing order of priority for interrupts is nonmaskable, vectored, and nonvectored. **NMI** is edge sensitive; when **NMI** is asserted, an internal latch is loaded. **VI** and **NVI** are level sensitive.

The CPU samples **VI**, **NVI**, and the internal **NMI** latch on the rising edge of **CLK**. The interrupt request signals can be asynchronous to **CLK**; the CPU synchronizes them internally.

After a request for an enabled interrupt is asserted, the CPU begins an interrupt acknowledge transaction. Figure 8-18 shows timing for an interrupt acknowledge transaction, indicated by status 0101, 0110, or 0111. The timing is similar to a single I/O read. Wait states (either programmed for automatic insertion or externally generated) can be inserted before **DS** falls in the middle of **T₂**, and before **DS** rises in the middle of **T₃**. Inserting wait states before **DS** falls allows for delay in the interrupt priority daisy chain.

A word of data is transferred on AD₀-AD₁₅. All of the interrupts save the transferred word on the system stack for processing the interrupt. Vectored interrupt uses the low-order byte of the word to select a unique PC value from the Program Status Area.

8.8.6 Internal Operation and Halt Transactions

Figure 8-19 shows timing for internal operation (status = 0000) and halt (status = 0011) transactions. Unlike other bus transactions, data is not transferred during these operations. Nevertheless, the data transfer size for the transaction indicates longword. The duration of the transaction is two bus cycles.

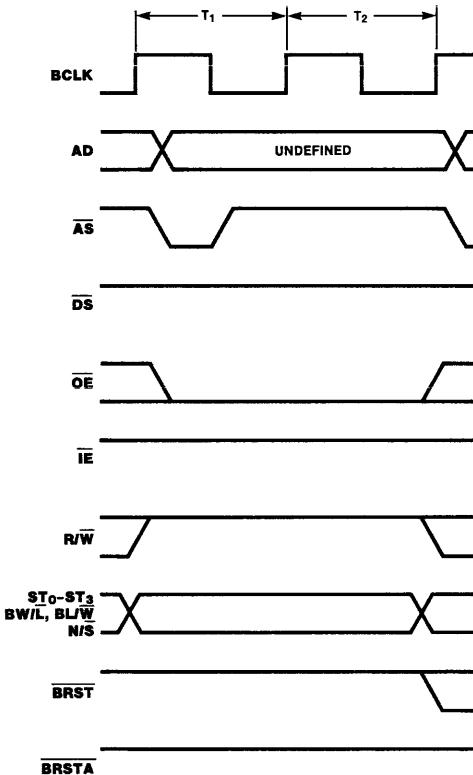


Figure 8-19.
Internal Operation and Halt Timing

The CPU generates an internal operation transaction after the end of a sequence of interlocked memory transactions. The CPU generates a halt transaction upon entering halted state (Section 7.2). When the Minimum Address Strobe Rate option is enabled (the MASR bit in HICR is 1), the CPU maintains a steady rate for Address Strobes by generating halt transactions in halted state or internal operation transactions otherwise.

8.8.7 Bus Retry

During transactions in which data is transferred, the responding device can indicate bus retry on RSP₀-RSP₁. When bus retry is sampled, the CPU terminates the transaction in progress, negating DS and BRST, then repeats the same transaction. If bus retry is indicated during a burst transfer, the retry transaction begins with the address for the data transfer where bus retry was indicated. The CPU does not acknowledge interrupts or bus requests between the retry response and the retry transaction.

8.8.8 Bus Error

During transactions in which data is transferred, the responding device can indicate a bus error exception on RSP₀-RSP₁. When bus error is sampled, the CPU terminates the transaction in progress, negating DS and BRST. A bus error exception also causes termination of the instruction in execution. In processing a bus error exception, the CPU saves the Program Status, physical address for the transaction, and a word identifying the status and control signals used for the transaction on the system stack, in that order (Figure 8-20). In the identifier word, High signals are 1, and Low signals are 0.

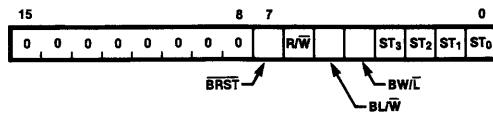


Figure 8-20. Bus Error Identifier Word

External Interface

8.9 BUS REQUEST AND ACKNOWLEDGE

The CPU supports two types of bus request/acknowledge sequences, local and global. Other bus masters request the local bus from the CPU using a handshake of **BUSREQ** and **BUSACK**. The CPU requests a global bus from an external arbiter using a handshake of **GREQ** and **GACK**.

To generate transactions on the local bus, a potential bus master (such as a DMA controller) must gain control of the bus by making a bus request (Figure 8-21). A local bus request is initiated by asserting **BUSREQ**. Several bus requestors may be wired to the **BUSREQ** signal; priorities are resolved externally to the CPU, usually by a priority daisy chain.

The CPU samples **BUSREQ** on the rising edge of **CLK**. **BUSREQ** can be asynchronous to **CLK**; the CPU synchronizes it internally. After **BUSREQ** is asserted, the CPU completes any transaction or sequence of interlocked transactions in progress, including possible retries. Next, the CPU responds by asserting **BUSACK** and placing its other output signals except **EPUABORT** in 3-state. The **EPUABORT** signal remains valid while the CPU has granted the local bus, and may be asserted if an EPA instruction is in progress. Later, when **BUSREQ** is negated, the CPU negates **BUSACK** and begins driving all other output signals.

The CPU can initiate transactions with devices located on a global bus shared with other CPUs. At any time, only one of the CPUs can initiate transactions on the global bus. Control of the global bus is arbitrated by external circuitry. Before initiating transactions on the global bus, the CPU requests control of the global bus from the arbiter using the protocol described below.

The CPU uses two fields of **HICR** to distinguish between local and global bus transactions. The **GE** bit enables use of the global bus. The 4-bit **LAD** field specifies one of sixteen sections of the physical address space used for local references.

Before every memory and I/O bus transaction (status codes 0010 and 1000 through 1111), the CPU compares the **LAD** field with bits 26 to 29 of the physical address. If the comparison is unequal and **GE** is 1, then the transaction is a global bus reference; otherwise the transaction is a local bus reference. In a tightly-coupled multi-processor system (Figure 8-2c), each of the local and global memory locations and peripheral ports can have a unique system address. Each CPU loads a distinct value into **LAD**, identifying its local addresses; the CPUs refer to global addresses and local addresses of other CPUs using the global bus request protocol.

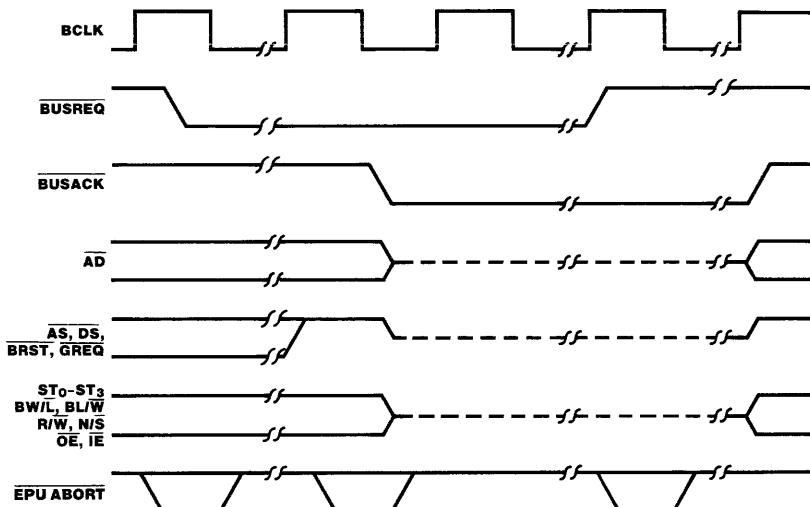


Figure 8-21.
Local Bus Request Acknowledge Timing

Figure 8-22 shows timing for the global bus request/acknowledge protocol. Before initiating a transaction on the global bus, the CPU drives the address, ST₀-ST₃, BRST, R/W, N/S, BL/W, and BW/L valid at the beginning of a bus cycle. Then, in the middle of the bus cycle, the CPU asserts GREQ. When the global bus selected by the address is available to the CPU, the arbiter asserts

GACK. The CPU samples GACK on the rising edge of CLK. GACK can be asynchronous to CLK; the CPU synchronizes it internally. The CPU performs one or more transactions on the global bus, then negates GREQ. The arbiter responds by negating GACK; the CPU can then initiate more transactions.

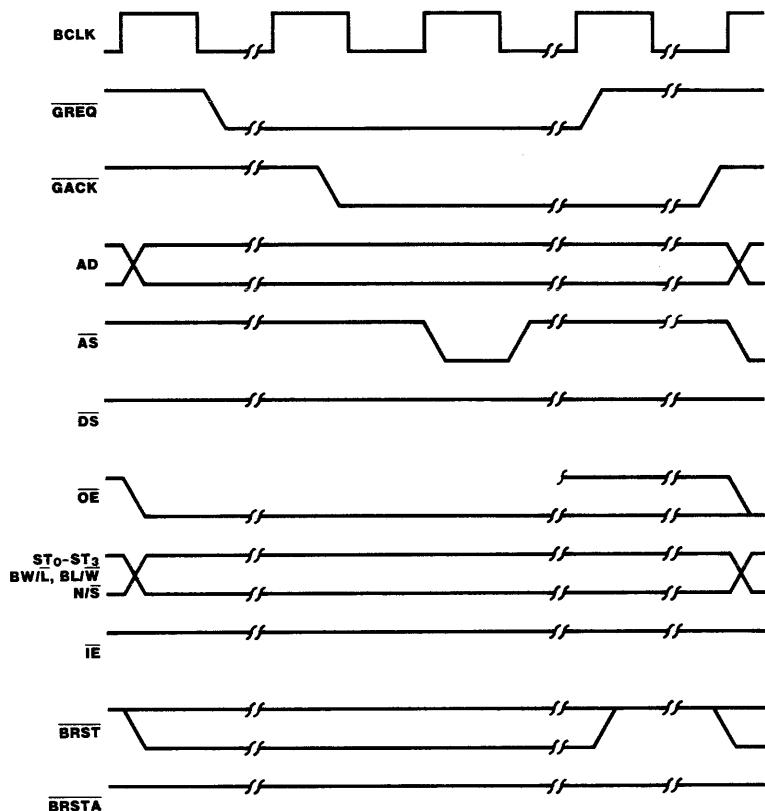
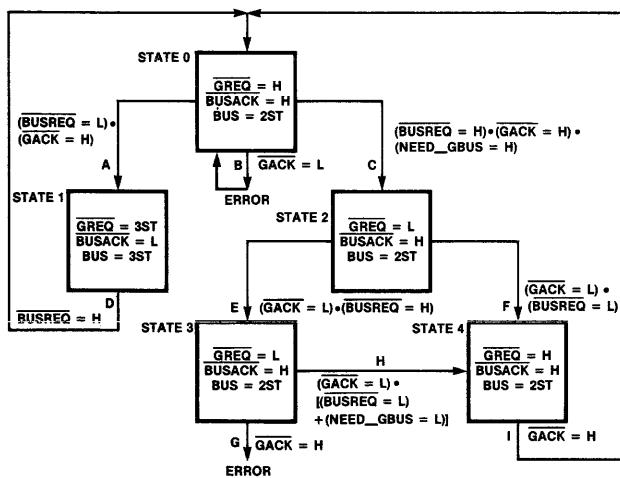


Figure 8-22. Global Bus Request Timing

External Interface

Figure 8-23 shows a state diagram for the local and global bus request protocols. To prevent deadlock between CPUs referring to each other's local memories, a CPU can be preempted while it is

waiting for GACK in State 2. If BUSREQ is asserted before GACK, the CPU relinquishes the global bus without performing any transactions.



NOTES: Interface signals are High (H), Low (L), High or Low (2ST), or 3-stated (3ST).

NEED_GBUS is an active High signal internal to the CPU.

Figure 8-23. State Diagram for CPU Bus Request Protocol

External Interface

State Legend		Transition Legend	
State 0	The CPU controls the local bus and is neither requesting nor controlling the global bus. The CPU can perform transactions on the local bus.	A B	A local bus request occurs. The global bus arbiter grants control of the global bus when no global bus request is pending. This is an error. The CPU remains in State 0.
State 1	The CPU has granted the local bus. The CPU cannot perform transactions.	C	The CPU requests the global bus in response to the internally generated signal NEED_GBUS.
State 2	The CPU controls the local bus and is requesting the global bus. The CPU cannot perform transactions.	D	The local bus master relinquishes the bus.
State 3	The CPU controls the local and global buses. The CPU can perform transactions on the global bus.	E F	The global bus arbiter grants the global bus to the CPU while no local bus request is pending. The global bus arbiter grants the global bus to the CPU while a local bus request is pending. The CPU is preempted.
State 4	The CPU controls the local bus and is relinquishing control of the global bus. The CPU cannot perform transactions.	G H I	The global bus arbiter reclaims the global bus before the CPU relinquishes the global bus. This is an error. The CPU's response to this error is undefined. The CPU relinquishes control of the global bus when it no longer needs the global bus or in response to a local bus request. The global bus arbiter reclaims the global bus.

External Interface

8.10 RESET

Figure 8-24 shows Reset timing. After **RESET** is asserted, the CPU responds as follows.

- AD lines are turned to input direction
- AS, BRST, BUSACK, DS, EPUABORT, GREQ, IE, and OE are negated
- ST₀-ST₃ are driven to 1111
- BW/C and BL/W are driven Low
- N/S and R/W are undefined

If **RESET** is asserted while the CPU is asserting **BUSACK**, the CPU first negates **BUSACK**, then the other CPU output lines are removed from 3-state and driven as described above. After **RESET** is asserted, external circuitry can detect that the CPU has responded to the reset request by sensing **BW/C** and **BL/W** Low. At power on, **RESET** should be asserted until after power has stabilized.

During reset, bits SX, NX, CI, and CD of the SCCL control register are cleared, disabling the address translation and cache mechanisms. Bit GE of HICR is also cleared, disabling the global bus request protocol.

At the rising edge of **RESET**, the relationship between bus timing, memory data path, and number of automatic wait states is determined. If **RSP₀** is High at the rising edge of **RESET**, **HICR** is initialized with M₀.DP = 1, M₀.W = 7, and S = 1. This corresponds to a default configuration of 16-bit memory path, seven automatic wait states, and bus clock scale factor 2. If **RSP₀** is Low at the rising edge of **RESET**, AD₀-AD₃ and AD₁₁ are latched into the corresponding bits of **HICR**, and AD₁₅ must be High.

RESET need not be synchronous with **CLK**; however, the CPU assumes that the last rising edge of **CLK** on which **RESET** is asserted corresponds to a rising edge of **BCLK**. Thus, if **RESET** is synchronized with the rising edge of the external bus clock, the internal and external bus clocks will be in phase with respect to **CLK**. After **RESET** is negated, the CPU reads FCW from memory address 2 and PC from address 4 using status 1101. If **BUSREQ** is asserted before **RESET** is negated, the CPU acknowledges the bus request before fetching the Program Status.

External Interface

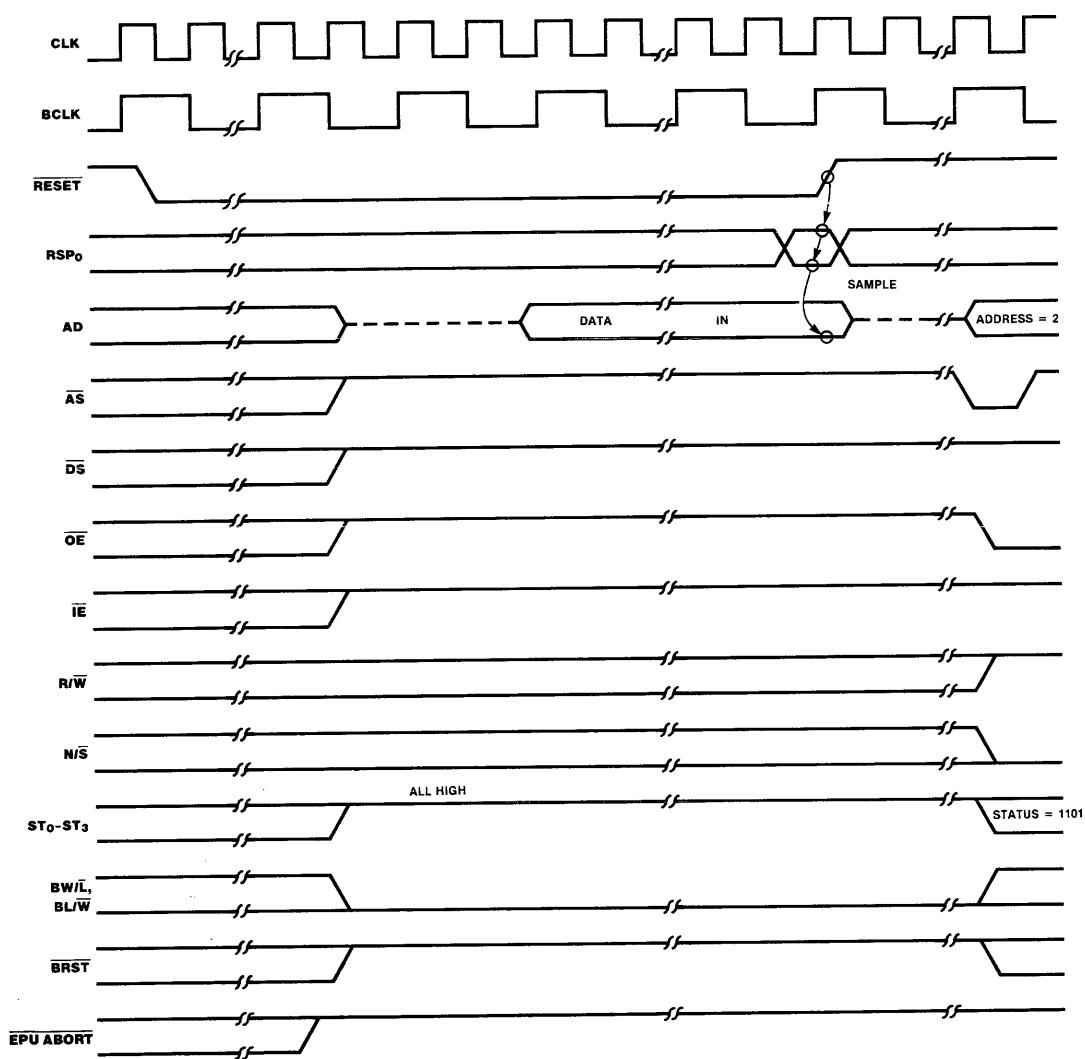


Figure 8-24. Reset Timing

A

Zilog

Appendix A. Z8000 Compatibility

The Z80,000 CPU is an upward-compatible extension of Z8000 architecture and bus interface. All Z8000 normal mode software and most Z8000 system mode software executes on the Z80,000 CPU, provided the software contains no timing dependencies, does not modify itself, and does not use any of the Z8000 reserved instruction, address, and control field encodings.

A few of the Z8000 privileged instructions are not implemented by the Z80,000 CPU. The instructions are LDCTL (refresh control register), the Multi-Micro set (MBIT, MREQ, MRES, MSET), and the Special I/O instruction set (SIN, SINB, SIND, SINDB, SINDR, SINDRB, SINI, SINIB, SINIR, SINIRB, SOTDR, SOTDRB, SOTIR, SOTIRB, SOUT, SOUTB, SOUTD, SOUTDB, SOUTI, and SOUTIB). An Unimplemented Instruction trap occurs when a program attempts to execute one of these instructions.

The portions of a Z8000 operating system concerning memory management and initialization of the Program Status Area (PSA) must be modified to execute on the Z80,000 CPU. The PSA for the Z80,000 CPU is an extension of the Z8000's PSA, with more entries for additional exceptions.

Memory management is integrated in the Z80,000 CPU, while the Z8000 CPU implements memory management in peripheral components (Z8010 Memory Management Unit and Z8015 Paged Memory Management Unit). In addition, the Z80,000 CPU does not separate stack and data address spaces as does the Z8000 CPU. Any inconveniences caused by these

differences can be minimized by following the guidelines in the application note "Memory Management and the Z80,000 32-bit Microprocessor" (Zilog document number 00-2329-01).

The Z80,000 CPU is compatible with the signals and timing of the 16-bit Z-BUS, except for the Multi-Micro resource request signals. The global bus request protocol of the Z80,000 CPU replaces the Multi-Micro protocol. The Z80,000 CPU also improves the Z-BUS sampling of WAIT and permits memory read transactions of two bus cycles duration, though strict Z-BUS compatibility can be maintained by programming appropriate fields in the Hardware Interface Control register. (For strict Z-BUS compatibility, HICR fields $M_0.DP$, $M_0.W$, $M_1.DP$, $M_1.W$, $I/O_0.W$, and $I/O_1.W$ are 1; $IACK.W1$ is 3; $IACK.W2$ is 2; and GE is 0.) For the Z80,000 CPU, EPU-to-memory write transaction timing includes one cycle more than the Z-BUS specification; the additional cycle prevents a bus clash between the CPU and EPU.

Aside from the Z-BUS signals and timing described above, there are only the following few differences between the Z80,000 CPU and Z8000 CPU pin signals. The Z80,000 CPU does not implement the Z8000 CPU signals MREQ, STOP, ABORT, (Z8003 and Z8004 only), SEGT (Z8001 only), and SAT (Z8003 only). Additionally, some of the status code definitions have been changed to accommodate the cache in the Z80,000 CPU. The Z80,000 CPU does not support refresh transactions.

B

Zilog

Appendix B. Memory-Mapped I/O

The CPU's memory management mechanism can map logical memory addresses to physical I/O addresses by setting bit 31 of a page table entry to 1. Memory-mapped I/O can be used only for references to the data memory logical address spaces with the following instructions.

ADD	DEC	RES
AND	EX	SET
BIT	INC	SUB
CLR	LD	TEST
COM	NEG	TESTA
CP (not Immediate)	OR	XOR

Memory-mapped I/O must not be used for instruction address space references or for data references with instructions other than those listed above. If memory-mapped I/O is used in this prohibited manner, the CPU may not be able to recover correctly from an address translation exception that is detected after the peripheral port has been accessed, because the state of the peripheral may have changed. In addition, instructions like Decrement Interlocked and those for the Extended Processing Architecture cannot use I/O status on bus transactions.

C

Zilog

Appendix C. Cache Control and Memory Transactions

The Z80,000 CPU implements a cache mechanism that keeps copies of frequently used memory locations on-chip for fast access. The cache mechanism is selectively enabled for instruction and data references by bits CI and CD in the SCCL register. The cache replacement algorithm is controlled by the CR bit in the SCCL register. When the replacement algorithm is enabled, (CR=1), the cache stores a copy of the most recently used memory locations; otherwise, the cache stores a copy of fixed memory locations.

The cache contains 16 blocks of storage (Figure C-1). Each block includes an address tag, which stores the 28 most-significant bits of the physical memory address corresponding to the block, and a bit specifying whether the address tag is valid. Associated with the tag, the block also stores eight data words and a bit for each word specifying whether or not the word contains a valid copy of the corresponding memory location. The cache is fully associative, so that any memory location can be assigned to any block. In all, the cache provides 256 bytes of data storage.

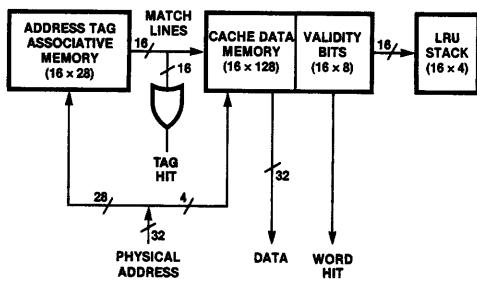


Figure C-1. Cache Organization

The Purge Cache (PCACHE) instruction invalidates all of the address tags and data words.

On memory references for which the cache is enabled, the cache is examined to determine whether a copy of the addressed location's contents is stored on-chip. If the cache is not enabled, the cache is bypassed. For instruction fetches (including fetches of operands specified by Immediate, Relative Address, or Relative Index addressing mode), the cache is enabled when CI is set to 1; if memory management is enabled, the NC bit of the page table entry must also be 0. For operand fetches, the cache is enabled when CD is set to 1 and the reference is not interlocked (i.e., not DECI, INCI, and TSET instructions); if memory management is enabled, the NC bit of the page table must also be 0. For operand stores, the cache is always enabled. When the CPU fetches from the Program Status Area during exception processing or from the translation tables during address translation, the cache is bypassed.

When the cache is enabled for a reference, bits 4 to 31 of the physical memory address are compared to the tags in each cache block. The reference is called either a "tag hit" if one of the valid tags matches the address, or a "tag miss" if none of the tags matches. When a tag hit occurs, bits 1 to 3 of the address select a data word in the block. If the data word is valid, the reference is called a "word hit"; otherwise, it is called a "word miss." For an aligned longword reference, both the high-order and low-order words, along with their validity bits, are accessed simultaneously.

For instruction fetches, if the reference is a word hit, the instruction word is simply read from the cache. If the reference misses and the cache is enabled for instructions, the instruction word is fetched from memory using a burst transaction. The CPU continues the burst transaction, reading successive words as long as memory acknowledges the burst or until the end of the block. If the cache is bypassed, the instruction is fetched using a single read operation.

Cache Control and Memory Transactions

For operand fetches, if the reference is a word hit, the data word is simply read from the cache. Otherwise, if the reference misses or the cache is bypassed, the data word is fetched from memory. Only data fetches that involve more than one transfer use burst transactions, such as those for the following instructions: CPI(R), CPSI(R), CHECK, EXIT, INDEX, IRET, LDI(R), LDM, LDML, LDPS, OUTI(R), TRTI(R)B, and EPA instructions. Similarly, burst transactions are used for fetching unaligned operands and longword operands on a 16-bit memory data path. When an operand is specified using Relative Address addressing mode, the instruction transfer status (1100 or 1101) is used except for EPA instructions, which use data transfer status (1010 or 1011).

For operand stores and saving Program Status during exception processing, if the reference is a word hit, the data byte or word is written to the cache; however, the data word is invalidated for an EPA instruction. If the reference is a tag miss or word miss, the cache is unaffected. The data is written to memory regardless of whether the cache hits or misses. This ensures that the current value for a location is always stored in memory. The CPU uses burst transactions only for stores with Load Multiple and Load Multiple longword registers, Enter, and EPA instructions.

Table C-1 summarizes the activity in the cache and external interface described above. The status codes distinguish cacheable and non-cacheable references for use with an external cache.

When the CPU fetches from the PSA during exception processing, a burst transaction with status 1101 is used. If the CPU stores to the overflow stack during exception processing, a transaction with status 1001 is used. When translation table entries are fetched or stored (to update the M and R bits) during address translation, the CPU uses status 1111.

In addition to the address tags, data, and validity bits, the cache contains a stack that orders the blocks according to how recently they have been used with the most recently used block on the top of the stack. Whenever a reference is a tag hit, the corresponding block moves to the top of the stack, and the blocks that previous to the reference had been more recently used move down the stack. The bottom of the stack identifies the least recently used (LRU) block.

If the cache replacement algorithm is enabled, the contents of the cache change when a cache miss occurs. For a tag miss, the CPU first replaces the tag of the LRU block with the missing block's address, and marks all the data words in the block invalid. For either a tag miss or word miss, the CPU loads the data fetched from memory into the selected cache block and marks the corresponding words valid.

When the cache replacement algorithm is disabled, copies of fixed memory locations can be locked into the cache for fast, on-chip access. To do this, the cache is first enabled for block replacement of data references only (CR=1, CD=1, CI=0). Then the cache is purged and selected blocks are read into the cache. Afterwards, the replacement algorithm is disabled, and the cache is enabled for instruction and data references (CR=0, CD=1, CI=1).

The number of data words per block, number of blocks, degree of associativity, and replacement algorithm described for the cache design in this appendix are specific to the first implementation of the Z80,000 CPU architecture and may differ in future products implementing the same architecture. Differences in these characteristics can impact on system performance, but have no effect on the function of software or the external interface.

Table C-1. Cache and Bus Activity

Reference	Hit/Miss	Cache Activity Data	LRU	Bus Transaction (status)
Instruction Fetch				
CI•NC	hit miss	no change update	update update	no yes (1100)
CI•NC	don't care	no change	no change	yes (1100)
NC	don't care	no change	no change	yes (1101)
Operand Fetch				
CD•NC•ILOK	hit miss	no change update	update update	no yes (1000, 1010, or 1100)
CD•NC•ILOK	don't care	no change	no change	yes (1000, 1010, or 1100)
NC•ILOK	don't care	no change	no change	yes (1001, 1011, or 1101)
ILOK	don't care	no change	no change	yes (1111)
Operand Store				
NC•ILOK	hit miss	update no change	update no change	yes (1000, 1010, or 1100) yes (1000, 1010, or 1100)
NC•ILOK	hit miss	update no change	update no change	yes (1001, 1011, or 1101) yes (1001, 1011, or 1101)
ILOK	hit miss	update no change	update no change	yes (1111) yes (1111)

Key: CD CD in SCCL
 CI CI in SCCL
 ILOK Interlocked reference required
 NC NC in Page Table Entry

D

Zilog

Appendix D. Programmer's Quick Reference Guide

LOWER NIBBLE (HEX), LOWER INSTRUCTION BYTE																
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	ADD _B 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 IR → IM	SUBL R → IR R → IM	PUSH IR → IR IR → IM	LDL R → IR R → IM	POPL IR → IR R → IM	ADDL R → IR R → IM	POP IR → IR R → IM	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 R → IM R → R	RES R → IM R → R	SETB R → IM R → R	BITB R → IM R → R	BIT R → IM R → R	INC R → IM R → R	INC R → IM R → R	INC R → IM R → R	DEC R → IM R → R	DEC R → IM R → R	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	UNIM	LDL BA → R LDRL RA → R	LDKL R → IM	LDPS IR	See Table 3	See Table 3	INB R → IR	IN R → IR	OUTB IR → R	OUT IR → R
4	ADD _B R → EAM	ADD R → EAM	SUBB R → EAM	SUB R → EAM	ORB R → EAM	OR R → EAM	ANDB R → EAM	AND R → EAM	XORB R → EAM	XOR R → EAM	CPB R → EAM	CP R → EAM	See Table 1	See Table 1	EXTEND INST	EXTEND INST
5	CPL R → EAM	PUSHL IR → EAM	SUBL R → EAM	PUSH IR → EAM	LDL R → EAM	POPL IR → EAM	ADDL R → EAM	POP IR → EAM	MULTL R → EAM	MULT R → EAM	DIVL R → EAM	DIV R → EAM	See Table 2	LDL EAM → R	JP PC → EAM	CALL PC → EAM
6	LDB R → EAM	LD R → EAM	RESB EAM → IM	RES EAM → IM	SETB EAM → IM	SET EAM → IM	BITB EAM → IM	BIT EAM → IM	INC EAM → IM	INC EAM → IM	DEC EAM → IM	DEC EAM → IM	EXB R → EAM	EX R → EAM	LDB EAM → R	LD EAM → R
7	LDB R → BX	See Table 7	LDB BX → R	LD BX → R	LDA R → BX	LDL R → EAM	LDA BX → R	LDL PS → EAM	CVT CVTU	LDPS PS → EAM	See Table 8	See Table 7	EI DI	See Table 7	TRAP	SC
8	ADD _B 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	LDCTLL CTLRL → R	RET PC → (SP)	LDCTLL R → CTLRL
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	INC R → IM	INC R → IM	DEC 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	ADC _B R → R	ADC R → R	SBCB R → R	SBC R → R	See Table 5	See Table 5	See Table 6	See Table 6	RRDB R	LDK R → IM	RLDB R	UNIM
C	LDB R → IM															
D	CALR PC → RA															
E	JR PC → RA															
F	DJNZ DBJNZ PC → RA															

Notes:

- 1) Opcodes marked UNIM are unimplemented and must not be used. Attempting to execute an unimplemented opcode causes an Unimplemented Instruction trap.
- 2) The execution of an extended instruction results in an Extended Instruction trap if the EPA bit in the FCW is 0; otherwise, the CPU sends the instruction to an EPU for execution.

Opcode Map

Programmer's Quick Reference Guide

	OC	OD
0	COMB IR	COM IR
1	CPB IR—IM	CP IR—IM
2	NEGB IR	NEG IR
3	UNIM	CPL IR—IM
4	TESTB IR	TEST IR
5	LDB IR—IM	LD IR—IM
6	TSETB IR	TSET IR
7	UNIM	LDL IR—IM
8	CLRB IR	CLR IR
9	UNIM	PUSH IR—IM
A	CHKB R—IM R—IR	CHK R—IM R—IR
B	UNIM	CHKL R—IM R—IR
C	TESTAB IR	TESTA IR
D	UNIM	UNIM
E	UNIM	INDEX R—IM R—IR
F	UNIM	INDEXL R—IM R—IR

LOWER NIBBLE (HEX), LOWER INSTRUCTION BYTE

	4C	4D
	COMB EAM	COM EAM
	CPB EAM—IM	CP EAM—IM
	NEGB EAM	NEG EAM
	UNIM	CPL EAM—IM
	TESTB EAM	TEST EAM
	LDB EAM—IM	LD EAM—IM
	TSETB EAM	TSET EAM
	UNIM	LDL EAM—IR
	CLRB EAM	CLR EAM
	UNIM	PUSH IR—IM
	CHKB R—EAM	CHK R—EAM
	UNIM	CHKL R—EAM
	TESTAB EAM	TESTA EAM
	UNIM	INDEX R—EAM
	UNIM	INDEXL R—EAM

	8C	8D
	COMB R	COM R
	LDCTLB R—FLGS	SETFLG
	NEGB R	NEG R
	UNIM	RESFLG
	TESTB R	TEST R
	UNIM	COMFLG
	TSETB R	TSET R
	UNIM	NOP
	CLRB R	CLR R
	UNIM	UNIM
	LDCTLB FLGS—R	UNIM
	UNIM	UNIM
	TESTAB R	TESTA R
	UNIM	UNIM
	UNIM	UNIM
	UNIM	UNIM

	1C	5C	9C
0	COML IR	COML EAM	COML R
1	LDM R—IR R—IM	LDM R—EAM	UNIM
2	NEGL IR	NEGL EAM	NEGL R
3	UNIM	UNIM	UNIM
4	CLRL IR	CLRL EAM	CLRL R
5	LDML IM—IR IM—IM	LDML IM—EAM	UNIM
6	INSRT IR—R	INSRT EAM—R	INSRT R—R
7	UNIM	UNIM	UNIM
8	TESTL IR	TESTL EAM	TESTL R
9	LDM IR—R	LDM EAM—R	UNIM
A	EXTR R—IR	EXTR R—EAM	EXTR R—R
B	EXTRU R—IR	EXTRU R—EAM	EXTRU R—R
C	TESTAL IR	TESTAL EAM	TESTAL R
D	LDML IR—IM	LDML EAM—IM	UNIM
E	UNIM	UNIM	UNIM
F	UNIM	UNIM	UNIM

Table 1. Upper Instruction Byte

Table 2. Upper Instruction Byte

LOWER NIBBLE (HEX), LOWER INSTRUCTION BYTE		LOWER NIBBLE (HEX), LOWER INSTRUCTION BYTE		LOWER NIBBLE (HEX), LOWER INSTRUCTION BYTE		LOWER NIBBLE (HEX), LOWER INSTRUCTION BYTE		LOWER NIBBLE (HEX), LOWER INSTRUCTION BYTE	
3A	3B	B2	B3	B8		BA	BB	BB	B9
0	INIB IR—IR INIRB IR—IR	INI IR—IR INIR IR—IR	0	RIB (1 bit) R—IM	RL (1 bit) R—IM	0	TRIB IR—IR	CPIB IR—IR	CPI IR—IR
1	UNIM	UNIM	1	SLLB R—IM SRLB R—IM	SLL R—IM SRL R—IM	1	UNIM	LDB IR—IR LDIRB IR—IR	LDI IR—IR LDIR IR—IR
2	OUTIB IR—IR OTIRB IR—IR	OUTI IR—IR OTIR IR—IR	2	RLB (2 bits) R—IM	RL (2 bits) R—IM	2	TRTIB IR—IR	CPSIB IR—IR	CPSI IR—IR
3	UNIM	UNIM	3	SDLB R—IM	SDL R—IM	3	UNIM	UNIM	UNIM
4	INB R—DA	IN R—DA	4	RRB (1 bit) R—IM	RR (1 bit) R—IM	4	TRIRB IR—IR	CPIRB R—IR	CPIR R—IR
5	UNIM	UNIM	5	UNIM	SLLL R—IM SRLL R—IM	5	UNIM	UNIM	UNIM
6	OUTB DA—R	OUT DA—R	6	RRB (2 bits) R—IM	RR (2 bits) R—IM	6	TRTIRB IR—IR	CPSIRB IR—IR	CPSIR IR—IR
7	UNIM	UNIM	7	UNIM	SDLL R—IM	7	UNIM	UNIM	UNIM
8	INDB IR—IR INDRB IR—IR	IND IR—IR INDR IR—IR	8	RLCB (1 bit) R—IM	RLC (1 bit) R—IM	8	TRDB IR—IR	CPD R—IR	CPD R—IR
9	UNIM	UNIM	9	SLAB R—IM SRAB R—IM	SLA R—IM SRA R—IM	9	UNIM	LDBB IR—IR LDDR IR—IR	LDD IR—IR LDDR IR—IR
A	OUTDB IR—IR OTDRB IR—IR	OUTD IR—IR OTDR IR—IR	A	RLCB (2 bits) R—IM	RLC (2 bits) R—IM	A	TRTDB IR—IR	CPSDB IR—IR	CPSD IR—IR
B	UNIM	UNIM	B	SDAB R—R	SDA R—R	B	UNIM	UNIM	UNIM
C			C	RRCB (1 bit) R—IM	RRC (1 bit) R—IM	C	TRDRB IR—IR	CPDRB R—IR	CPDR R—IR
D			D	UNIM	SLAL R—IM SRAL R—IM	D	UNIM	UNIM	UNIM
E			E	RRCB (2 bits) R—IM	RRC (2 bits) R—IM	E	TRTDRB IR—IR	CPSDRB IR—IR	CPSDR IR—IR
F			F	UNIM	SDAL R—R	F	UNIM	UNIM	UNIM

Table 3.
Upper Instruction Byte

Table 4.
Upper Instruction Byte

Table 5.
Upper Instruction Byte

Table 6.
Upper Instruction Byte

Programmer's Quick Reference Guide

	7B	7D
0	IRET	UNIM
1	UNIM	UNIM
2	UNIM	LDCTL R—FCW
3	UNIM	UNIM
4	UNIM	LDCTL R— PSAPSEG
5	UNIM	LDCTL R— PSAPOFF
6	UNIM	LDCTL R—NSPSEG
7	UNIM	LDCTL R—NSPOFF
8	UNIM	UNIM
9	UNIM	UNIM
A	UNIM	LDCTL FCW—R
B	UNIM	UNIM
C	UNIM	LDCTL PSAPSEG —R
D	UNIM	LDCTL PSAPOFF —R
E	UNIM	LDCTL NSPSEG—R
F	UNIM	LDCTL NSPOFF—R

LOWER NIBBLE (HEX), LOWER INSTRUCTION BYTE

	0	1	2	3
0	HALT			
1	BREAK- POINT			
2	ESC LONG'			
3	ESC UNSIGN'			
4	ESC INTERLOCK'			
5	ENTER			
6	EXIT			
7		LDNI	LDND	
8	PCACHE			
9	PTLBESI	PTLBESD	PTLBENI	PTLBEND
A	PTLB			
B	PTLBN			
C				
D	LDPSI	LDPSD	LDPNI	LDPNND
E				
F				

Table 8. Lower Instruction Byte

Compact	Segmented or Linear
Direct Address Index	Direct Address Index Base Address Base Index Relative Address Relative Index

Table 9. Extended Addressing Modes

E

Zilog

Appendix E. Timing Formulae for Performance Evaluation

INTRODUCTION

The Z80,000 CPU, unlike the Z8000 and other 16-bit microprocessors, integrates a highly pipelined design, cache memory, and memory management into a single component. With the earlier microprocessors it is relatively simple to calculate exact performance measurements for a benchmark program or program workload mixture, as follows. Each instruction (i) in the architecture is characterized by its execution cycle count (n_i) and number of memory references (r_i). From the program workload, the frequency of execution for each instruction (f_i) can be determined. If W is the number of wait states for the memory system, then the average number of cycles to process an instruction (T_I) can be determined from the following formula.

$$T_I = \sum_i f_i(n_i + r_i W)$$

And, if T_C is the cycle time of the processor, then the processor's performance (that is, the processor's rate of executing instructions) is given by the formula below.

$$\text{performance} = (T_I T_C)^{-1}$$

Calculating the performance of the Z80,000 processor involves a more complex formulation that accounts for dependencies between instructions in the pipeline and misses for the cache and Translation Lookaside Buffer (TLB). This appendix contains the timing formulae used to analyze the performance of the Z80,000 CPU, and also a sufficiently detailed description of the processor's implementation to calculate timing parameters for a program workload.

Theory of Operation

Figure E-1 shows a block diagram of the Z80,000 CPU's internal organization, including the following major functional units and data paths:

- The external interface logic controls transactions on the bus. Addresses and data from the internal memory bus are transmitted

through the interface to the Z-BUS. The Z-BUS is a time-multiplexed, address/data bus that connects the components of a microprocessor system.

- The cache stores copies of instruction and data memory locations. Instructions are read from the cache on the instruction bus. Data is read from or written to the cache on the memory bus. The cache also includes a copy of the physical Program Counter, so that the logical addresses of instructions are translated only for branches and when incrementing the Program Counter across a page boundary.
- The Translation Lookaside Buffer (TLB) translates logical addresses calculated by the address arithmetic unit to physical addresses used to access the cache.
- The address arithmetic unit performs all address calculations. This unit has a path to the register file for reading base and index registers and another path to the instruction bus for reading displacements and direct addresses. The result of the address calculation is transmitted to the TLB.
- The register file contains the sixteen general-purpose longword registers, Program Status registers, special-purpose control registers, and several registers used to store values temporarily during instruction execution. The register file has one path to the address arithmetic unit and two paths to the execution arithmetic and logic unit.
- The execution arithmetic and logic unit calculates the results of instruction execution, such as add, exclusive-or, and simple load. This unit has two paths to the register file on which two operands can be read simultaneously or one can be written. One of the paths to the register file is multiplexed with a path from the memory bus.
- The instruction decoding and control unit decodes instructions and controls the operation of the other functional units. This unit has a path from the instruction bus and two

Timing Formulae for Performance Evaluation

programmable logic arrays for separate microcoded control of the two arithmetic units. This unit also controls exception handling and TLB loading.

All of the functional units and data paths listed above are 32 bits wide.

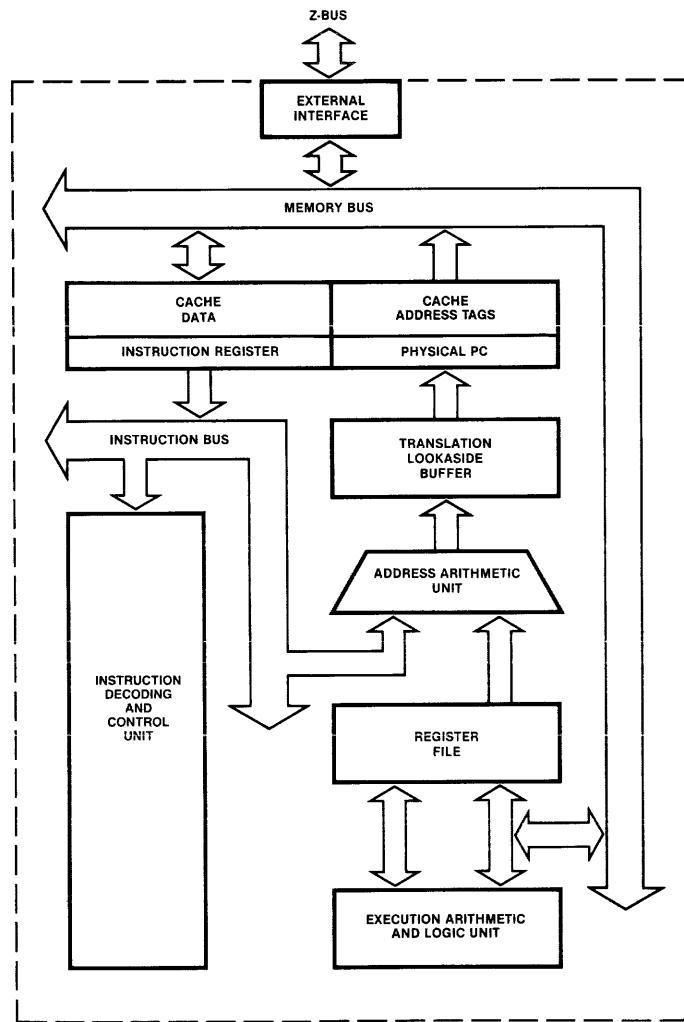


Figure E-1. Functional Block Diagram

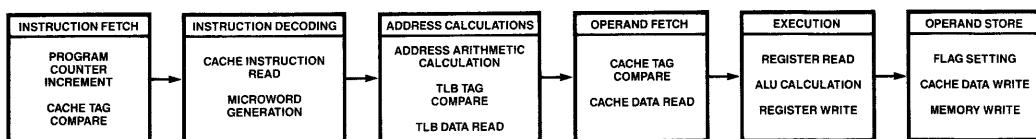


Figure E-2. Instruction Pipeline

The operation of the CPU is highly pipelined so that several instructions are simultaneously in different stages of execution. Thus, the functional units effectively operate in parallel with one instruction being fetched while an address is calculated for another instruction and results are stored for a third instruction.

Figure E-2 shows the six-stage, synchronous pipeline. Instructions flow through each stage of the pipeline in sequence. The various pipeline stages can be working simultaneously on separate instructions or on separate portions of a single complex instruction. Each pipeline stage operates in one processor cycle, which is composed of two clock cycles, called ϕ_1 and ϕ_2 . Thus, a processor cycle is 200 ns with a 10 MHz clock or 80 ns with a 25 MHz clock.

The instruction-fetch stage increments the Program Counter and initiates instructions fetched from the cache. The instruction-decoding stage receives and decodes instructions to set up control of the address-calculation stage.

The address-calculation stage can generally calculate a memory address in one processor cycle, except for Base Index, Relative, and Relative Index addressing modes, which require multiple cycles. After the logical effective address has been calculated, the corresponding physical address is provided by the TLB. The operand-fetch stage fetches the data from the cache and latches it into a holding register.

The execution stage performs data manipulations. Byte, word, and longword results are generally calculated in one processor cycle, but certain instructions, such as multiply and block-move operations, require multiple cycles. During the execution stage, results are stored to registers. Results are stored to the cache and external memory during the operand-store stage. The flags are also set during the operand-store stage.

The cache can handle two references during a processor cycle. Instruction fetches use the ϕ_2 , clock cycle for tag comparison and ϕ_1 for data access. Either an operand fetch or store can use ϕ_1 for tag comparison and ϕ_2 for data access.

The pipeline allows single instructions, like register-to-register load and memory-to-register add, to execute at a rate of one per processor cycle. Thus, the peak performance of the CPU is 12.5 million instructions per second (MIPS) with a 25 MHz clock. In practice, the actual performance is reduced to approximately one-third of the peak because of delays due to the execution of

multiple-cycle instructions, interference between instructions in the pipeline, and main memory accesses for cache and TLB misses.

In order to calculate the processor's performance it is helpful to separate the average processing time for an instruction into four components: execution delays, pipeline delays, addressing delays, and memory delays. The following sections describe the various delay components.

Execution Time

The first component of instruction processing time is the basic execution time: the time required to execute an instruction assuming that there is no interference from other instructions in the pipeline and that all memory references hit in the cache and TLB. An instruction's execution time is determined by its operation, data type, and addressing mode.

For most instructions, the execution delay can be calculated by adding the number of cycles from Table E-2 corresponding to the operation and data type to the number of cycles from Table E-1 corresponding to the addressing mode. Use either the source or destination addressing mode, as listed with the instruction's format in section 6.5. For the remaining instructions, Table E-2 gives the execution delays for specific combinations of operations, data types, and addressing modes. The following example shows how to use the tables.

An instruction that loads a longword from a register to a register (e.g., LDL RR4, RR2), has an execution time of 1 processor cycle: 1 for the operation and 0 for the addressing mode. An instruction that adds a longword immediate value to a register (e.g., ADDL RRO, #100), has an execution delay of 2 processor cycles: 1 for the operation and 1 for the addressing mode. An instruction that tests a bit of a byte in memory specified by IR addressing mode (e.g., BITB @RR2, #1), has an execution delay of 3 processor cycles: the delay is listed in Table E-2 for the specific operation and addressing mode.

Pipeline Delays

Pipeline delays result from interference between instructions at different stages of the pipeline. Pipeline delays occur when instructions contend for the use of a bus or functional unit, and one instruction must be delayed. There are two

Timing Formulae for Performance Evaluation

sources of pipeline delays: register interlocks and cache reference interlocks.

A register interlock occurs when an instruction modifies a register that is required for an address calculation by either of the two subsequent instructions. In addition, the following instructions, which may modify more than one register, cause an interlock for any registers used in subsequent address calculations: CPD(BL), CPI(BL), CPSD(BL), CPSI(BL), DIVL, DIVUL, EXIT, EXTSL, LDD(BL), LDI(BL), LDM registers from memory, LDML registers from memory, MULTL, MULTUL, and Load CPU from EPU. When the instruction that modifies the register is followed immediately by

the interlocked address calculation, then the pipeline delay is 2 processor cycles, otherwise the interlock causes a pipeline delay of 1 processor cycle. Register interlocks are detected for the use of longword registers. Thus, with the CPU's register file organization (see Figure 2-2), if a byte or word within a longword register is modified, then a subsequent address calculation can be interlocked by using the longword register itself or either of the word registers it contains.

For example, the following instruction sequences cause register interlock delays when executed (in linear mode).

INCL RR2, #4	//register interlock delay//
LDB RHO, @RR2	//for RR2 is 2 processor cycles//
MULT RR24, #1000	//register interlock delay//
LDL RRO, RR4	//for RR24//
ADDL RRO, RR12 (RR24)(16)	//is 1 processor cycle//

Table E-1. Execution Times for General Addressing Modes

		Address Representation	
		Addressing Mode	Compact Segmented or Linear
	R		0 0
IM	(byte or word) (longword)	0 1	0 1
	IR		0
	DA	0	0 for 1 extension word 1 for 2 or 3 extension words
	X	0	0 for 1 extension word 1 for 2 or 3 extension words
	BA	0	0 for 1 extension word 1 for 3 extension words
	BX	2	1 for 1 extension word 2 for 3 extension words
	RA	1	1 for 1 or 3 extension words
	RX	Not Available	2 for 1 or 3 extension words

Table E-2. Execution Time for Instruction Operations

Operation	Data Type	Addressing Modes	Execution Time	Notes
ADC	B,W	R	1	
	L	R	2	
ADD	B,W,L	See Table E-1	1	
AND	B,W	See Table E-1	1	
	L	See Table E-1	2	
Bit (Static)	B,W	R,EAM--See Table E-1	2	
		IR	3	
	L	R,EAM--See Table E-1	3	
		IR	4	
Bit (Dynamic)	B,W	R	4	
	L	R	5	
BRKPT			-	See Table E-5.
CALL		See Table E-1	5	
CALR		RA	4	
CHK	B,W,L	See Table E-1	8	Assumes trap not taken; see table E-5 if trap taken.
CLR	B,W,L	See Table E-1	1	
COM	B,W,L	R	1	
		IR,EAM--See Table E-1	2	
COMFLG			1	
CP (Register)	B,W,L	See Table E-1	1	
CP (Immediate)	B,W	See Table E-1	2	
	L	See Table E-1	3	
CPD	B,W,L	IR	7	
CPDR	B,W,L	IR	5+4n	n is number of iterations.
CPI	B,W,L	IR	7	
CPIR	B,W,L	IR	5+4n	n is number of iterations.

Timing Formulae for Performance Evaluation

Table E-2. Execution Time for Instruction Operations-Continued

Operation	Data Type	Addressing Modes	Execution Time	Notes
CPSD	B,W,L	IR	8	
CPSDR	B,W,L	IR	4+5n	n is number of iterations.
CPSI	B,W,L	IR	8	
CPSIR	B,W,L	IR	4+5n	n is number of iterations.
CVT (register)	All	See Table E-1	6	
CVT (memory)	All	See Table E-1	6	
CVTU (register)	All	See Table E-1	6	
CVTU (memory)	All	See Table E-1	6	
DEC	B,W	R	1	
		IR,EAM--See Table E-1	3	
	L	R	2	
		IR,EAM--See Table E-1	4	
DECI	B,W	See Table E-1	4	Cache bypassed for operand fetch, treat like cache miss.
DI			3	
DIV	W	See Table E-1	5	Case 1 Case 2 Case 3 or 4
			7	
	L	See Table E-1	25	
			4	Case 1 Case 2 Case 3 or 4
DIVU	W	See Table E-1	6	
			8	
	L	See Table E-1	26	
			5	Case 1 Case 2 Case 3
			7	
			39	

Table E-2. Execution Time for Instruction Operations-Continued

Operation	Data Type	Addressing Modes	Execution Time	Notes
DJNZ	B,W	R	2	Not taken
			5	Taken
	L	R	3	Not taken
			6	Taken
EI			3	
ENTER			15+4n	n is the number of registers specified in enter mask.
EX	B,W	See Table E-1	3	
	L	See Table E-1	4	
EXIT			10+n	n is the number of the registers specified in exit mask.
EXTR		R	6	
		IR,EAM--See Table E-1	11	
EXTRU		R	6	
		IR,EAM--See Table E-1	11	
EXTS	B	R	3	
	W,L	R	2	
HALT			1	
IN	B,W	IR	2	Add access time for input port.
		DA	1	
	L	IR	3	
		DA	2	
INC	B,W	R	1	
		IR,EAM--See Table E-1	3	
	L	R	2	
		IR,EAM--See Table E-1	4	
INCI	B,W	See Table E-1	4	Cache bypassed for operand fetch, treat like cache miss.

Timing Formulae for Performance Evaluation

Table E-2. Execution Time for Instruction Operations-Continued

Operation	Data Type	Addressing Modes	Execution Time	Notes
IND	B,W	IR	11	Assumes no I/O wait states--I/O wait states must be added.
	L	IR	12	
INDEX	W	See Table E-1	19	Assumes trap not taken; see Table E-5 if trap is taken.
	L	See Table E-1	27	
INDR	B,W	IR	3+8n	n is number of iterations. Assumes no I/O wait states--I/O wait states must be added for each iteration.
	L	IR	4+8n	
INI	B,W	IR	11	Assumes no I/O wait states--I/O wait states must be added.
	L	IR	12	
INIR	B,W	IR	3+8n	n is number of iterations. Assumes no I/O wait states--I/O wait states must be added for each iteration.
	L	IR	4+8n	
INSRT		R	17	
		IR,EAM--See Table E-1	18	
IRET			12	
JP		See Table E-1	1	Not taken
			4	Taken
JR		RA	1	Not taken
			4	Taken
LD (register)	B,W,L	See Table E-1	1	
LD (memory)	B,W,L	See Table E-1	1	
LD (immediate)	B,W,L	See Table E-1	3	
LDA		See Table E-1	1	
LDAR			1	
LDCTL (into Control register)			7	FCW
			3	NSP
			6	PSAP
LDCTL (from Control register)			1	

Table E-2. Execution Time for Instruction Operations-Continued

Operation	Data Type	Addressing Modes	Execution Time	Notes
LDCTLB			1	
LDCTLL (into Control register)			5 11 7	OSP, PSAP NSP SITTID, SDTTD, NITTD, NDTTD, SCCL, NSP
LDCTLL (from Control register)			1	
LDD	B,W,L	IR	9	
DDR	B,W,L	IR	4+5n	n is number of iterations.
LDI	B,W,L	IR	9	
LDIR	B,W,L	IR	4+5n	n is number of iterations.
LDK		R	1	
LDM (registers from memory)	W	See Table E-1	6+n/2	n is even number of registers.
			6+(n+1)/2	n is odd number of registers.
LDM (memory from registers)	W	See Table E-1	2n	n is even number of registers. See note 2.
			2 + 2n	n is odd number of registers.
LDML (registers from memory)	L	IM	9+n	n is number of registers specified in mask operand.
		IR,EAM--See Table E-1	7+n	
LDML (memory from registers)	L	See Table E-1	3+4n	n is number of registers specified in mask operand. See note 2.
LDN	B,W,L	See Table E-1	2	
LDP		See Table E-1	2	
LPDS		See Table E-1	11	
LDR	B,W,L		2	
MULT	W	See Table E-1	15	
	L	See Table E-1	24	

Timing Formulae for Performance Evaluation

Table E-2. Execution Time for Instruction Operations-Continued

Operation	Data Type	Addressing Modes	Execution Time	Notes
MULTU	W	See Table E-1	16	
	L	See Table E-1	25	
NEG	B,W,L	R	1	
		IR,EAM--See Table E-1	2	
NOP			1	
OR	B,W	See Table E-1	1	
	L	See Table E-1	2	
OTDR	B,W	IR	6	
	L	IR	7	
OTIR	B,W,	IR	6	
	L	IR	7	
OUT	B,W	IR	2	
		DA	1	
	L	IR	3	
		DA	2	
OUTD	B,W	IR	2+4n	n is number of iterations.
	L	IR	3+4n	
OUTI	B,W	IR	2+4n	n is number of iterations.
	L	IR	3+4n	
PCACHE			6	
POP	B,W,L	R	2	
		IR,EAM--See Table E-1	3	
PTLB			6	
PTLBE			6	
PTLBN			6	

Table E-2. Execution Time for Instruction Operations-Continued

Operation	Data Type	Addressing Modes	Execution Time	Notes
PUSH	B,W,L	R	2	
		IR,EAM--See Table E-1	3	
RES (Static)	B,W	R	2	
		IR	4	
		EAM--See Table E-1	3	
	L	R	3	
		IR	5	
		EAM--See Table E-1	4	
RES (Dynamic)	B,W	R	4	
	L	R	5	
RESFLG			1	
RET			6	Not taken
			7	Taken
RL	B,W	R	2+n	n = number of bits rotated.
	L	R	3+n	
RLC	B,W	R	2+n	n = number of bits rotated.
	L	R	3+n	
RLDB		R	6	
RR	B,W	R	2+n	n = number of bits rotated.
	L	R	3+n	
RRC	B,W	R	2+n	n = number of bits rotated.
	L	R	3+n	
RRDB		R	6	
SBC	B,W	R	1	
	L	R	2	
SC			-	See Table E-5.

Timing Formulae for Performance Evaluation

Table E-2. Execution Time for Instruction Operations-Continued

Operation	Data Type	Addressing Modes	Execution Time	Notes
SDA	B,W,L	R	8	Right shift
			9	Left shift
SDL	B,W,L	R	4	
SET (Static)	B,W	R	1	
		IR	3	
		EAM--See Table E-1	2	
	L	R	2	
		IR	4	
		EAM--See Table E-1	3	
SET (Dynamic)	B,W	R	3	
	L	R	3	
SETFLG			2	
SLA	B,W,L	R	9	
SLL	B,W,L	R	4	
SRA	B,W,L	R	8	
SRL	B,W,L	R	4	
SUB	B,W,L	See Table E-1	1	
TCC	B,W	R	1	
	L	R	2	
TEST	B,W,L	See Table E-1	1	
TESTA	B,W,L	See Table E-1	1	
TRAP			4	Assumes trap not taken; see Table E-5 if trap taken.
TRDB	B	IR	11	
TRDRB	B	IR	4+7n	n is number of iterations.
TRIB	B	IR	11	

Table E-2. Execution Time for Instruction Operations-Continued

Operation	Data Type	Addressing Modes	Execution Time	Notes
TRIRB	B	IR	$4+7n$	n is number of iterations.
TRTDB	B	IR	11	
TRTDRB	B	IR	$4+7n$	n is number of iterations.
TRTIB	B	IR	11	
TRTIRB	B	IR	$4+7n$	n is number of iterations.
TSET	B,W	See Table E-1	2	Cache bypassed for operand fetch, treat like cache miss.
	L	See Table E-1	3	
XOR	B,W	See Table E-1	1	
	L	See Table E-1	2	
Load EPU from Memory ¹	B,W	See Table E-1	4	Bus-timing scale factor is 2. Cache bypassed for operand fetch, treat like cache miss.
			7	Bus-timing scale factor is 4. Cache bypassed for operand fetch, treat like cache miss.
Load Memory from EPU ¹	B,W	See Table E-1	4	Bus-timing scale factor is 2. Add time to store operand, see memory delays section.
			7	Bus-timing scale factor is 4. Add time to store operand, see memory delays section.
Load CPU from EPU ¹	W,L	R	$9+(n/2)$	Bus-timing scale factor is 2. n is even number of words transferred.
			$9+(n+1)/2$	Bus-timing scale factor is 2. n is odd number of words transferred.
			15+n	Bus-timing scale factor is 4. n is even number of words transferred.
			16+n	Bus-timing scale factor is 4. n is odd number of words transferred.

Timing Formulae for Performance Evaluation

Table E-2. Execution Time for Instruction Operations-Continued

Operation	Data Type	Addressing Modes	Execution Time	Notes
Load EPU from CPU ¹	W,L	R	8+(n/2)	Bus-timing scale factor is 2. n is even number of words transferred.
			8+(n+1/2)	Bus-timing scale factor is 2. n is odd number of words transferred.
			12+n	Bus-timing scale factor is 4. n is even number of words transferred.
			13+n	Bus-timing scale factor is 4. n is odd number of words transferred.
Load FCW from EPU ¹			10	Bus-timing scale factor is 2.
			17	Bus-timing scale factor is 4.
Load EPU from FCW ¹			9	Bus-timing scale factor is 2.
			14	Bus-timing scale factor is 4.
Internal EPU operation ¹			1	Bus-timing scale factor is 2.
			2	Bus-timing scale factor is 4.

Note 1: The execution times reported for EPA instructions assume that the EPU does not force the CPU to wait by asserting EPUBSY. Refer to the Z8070 APU Technical Manual (Zilog document number 03-8226-01) for more information about execution delays for particular EPA instructions and consideration of instruction overlap between the CPU and EPU.

Note 2: Execution time for this instruction is less if burst transfers are supported for storing data into memory. See memory delays section.

A cache reference interlock occurs when an instruction modifies a memory location and either of the following two instructions fetches an operand from memory (including immediate mode operands other than those specified by special, compact encodings, like the source operands for BIT, DEC, and LDK instructions). This interlock is caused by contention for both the cache and memory bus. When the instruction that modifies memory is followed immediately by an instruction that fetches an operand, the pipeline delay is 2 processor cycles; otherwise, the pipeline delay is 1 processor cycle.

For example, the following instruction sequences cause cache reference interlocks when executed (in linear mode).

```
LDL RR12(10), RRO //cache reference interlock//
ADDL RR2, @RR20    //delay is 2 processor cycles//  

LDL RR12(10), RRO //cache reference interlock//
ADDL RR2, RR4      //delay is 1 processor cycle//  

ADDL RR2, @RR20
```

Addressing Delays

Addressing delays can occur when instructions or operands are located across longword or page boundaries. Unlike memory delays due to cache and TLB misses, which are described in the next section, addressing delays can be calculated from knowledge of the CPU's operation alone, without considering the memory system's latency and bandwidth.

An addressing delay of 1 processor cycle occurs when an operand that crosses a longword boundary is fetched. That is, when a longword is fetched from an address for which the two least significant bits differ from 00 or a word is fetched from an address for which the two least significant bits are 11. This delay arises because the CPU must make two memory references on its 32-bit memory bus.

An addressing delay of 1 cycle also occurs when the CPU branches to a two-word instruction that is located at an odd-word address. Another addressing delay of 3 cycles occurs when the PC is incremented across a page boundary during sequential instruction processing. The former delay arises from a gap in filling the instruction buffer, while the latter delay is caused by the need to translate the new page address in the PC.

Memory Delays

Memory delays occur when the CPU must wait to access external memory to service a cache or TLB miss or to store an operand. The duration of such delays depends on the memory system's data path width (16 or 32 bits), its access time, and its support for burst transfers. Thus, a microprocessor system designer can trade cost for performance by specifying these memory parameters as well as the CPU's clock speed and the bus-timing scale factor. In the description that follows, the times for single memory-read and -write transactions are represented by T_R and T_W processor cycles, respectively; the bus-timing scale factor (2 or 4) is represented by S . Burst transfers are assumed to take the same times (T_R and T_W) for the initial transfer and 1 bus clock cycle for each subsequent transfer.

The memory delay for both instruction and operand cache fetch misses is T_R . For instruction cache misses, burst transactions are used as follows: The CPU reads the missing word or longword (depending on the memory's data path width) and

requests the words or longwords that follow in the 16-byte cache block by signaling a burst transfer. The burst transfer continues until either the end of the 16-byte block is reached or the memory system indicates that it cannot support further transfers.

For operand fetch cache misses, burst transactions are used when more than one transfer is anticipated within a 16-byte block. Specifically, burst transfers are used to fetch operands for the following instructions: CPI(R), CPSI(R), CHECK, EXIT (registers only), INDEX, IRET, LDI(R), LDM, LDML, LDPS, OUTI(R), TRTI(R)B, and EPA instructions. Burst transfers are also used to fetch longword and unaligned word operands from a 16-bit wide memory, plus unaligned word and longword operands that cross an aligned longword boundary for a 32-bit wide memory. The CPU issues bus transactions until the entire operand has been fetched. If more than one operand word (for 16-bit memory) or longword (for 32-bit memory) remains to be transferred, the CPU transfers the first word or longword and attempts to burst transfer the remaining words or longwords until either all transfers are complete, the end of a 16-byte block is reached, or the memory system indicates that it cannot support further burst transfers.

For example, assume that the CPU requires seven longwords from memory location 8 to execute an LDML instruction, that all the longwords are missing from the cache, and that the memory system is 32 bits and supports burst transfers of 16-byte blocks. The CPU performs three bus transactions to fetch the seven longwords. The first transaction is a burst transfer of the longwords at locations 8 and 12, the second transaction is a burst transfer of the four longwords beginning at location 16, and the final transaction is a single transfer of the longword at location 32.

For a burst transaction with a bus-timing scale factor of 2, no memory delay in addition to T_R is incurred for burst transactions except when other transactions are pending, as described below. With a bus-timing scale factor of 4, an additional memory delay of 1 processor cycle is incurred for each burst transfer.

The memory delay for a TLB miss depends on the time to fetch an aligned longword from memory and the number of translation table levels. The formulae in Table E-3 give the number of processor cycle delays for a TLB miss, where N represents the number of table levels.

Table E-3. TLB Miss Delay

Memory System	TLB Miss Delay
16-bit, no burst	$11 + (5 + 2T_R + S/2) \times N$
16-bit, burst	$11 + (5 + T_R + S/2) \times N$
32-bit	$11 + (5 + T_R) \times N$

For example, assume that the time for a single memory read transaction is 2 processor cycles, the memory data path is 32 bits, and 2 levels of translation tables are used. Then the memory delay for a TLB miss is 25 processor cycles ($25 = 11 + (5 + 2) \times 2$).

Besides cache and TLB misses, the CPU can also experience memory delays if one bus transaction is held pending while another is performed. In such cases of bus contention, the CPU completes the first transaction, then after 1 bus cycle delay, initiates the pending transaction. Thus, additional cycles of delay occur if the servicing of a cache miss must wait for the completion of a previous burst memory-read transaction or a memory-write transaction. (The servicing of a cache miss may also be delayed by an EPA instruction transfer for a previous EPU internal operation instruction.) Similarly, additional delay is incurred when the storing of an operand must wait for the completion of a previous burst-memory read transaction or a memory-write transaction. In general, the delays due to bus contention either between read transactions or between read and write transactions can be ignored in calculating the CPU's performance; these delays have in large part been counted by the cache misses and cache interlocks previously described. Delays caused by bus contention between write transactions, though, must be considered, as explained below.

Because the CPU buffers the data for only one write transaction at a time, when an instruction that stores an operand to memory is followed shortly by another instruction that stores to memory, the second instruction is delayed. If the two store instructions are separated by Δ instructions, where the value of Δ for consecutive instructions is 1, then the CPU is delayed by $\text{Max}(0, T_W + S/2 - \Delta)$ processor cycles. For instance, assume that the time for a single memory-write transaction is 3 processor cycles and the bus-timing scale factor is 2. Then the CPU is

delayed by 3 processor cycles when the second store instruction immediately follows the first or by 2 processor cycles if there is one non-store instruction intervening between the two store instructions. If the store instructions are separated by more than three instructions that do not store, then there is no delay.

Two or three consecutive memory-write transactions are required for an instruction that stores an unaligned word or longword and also for an instruction that stores an aligned longword to a 16-bit memory. The memory delay in processor cycles is shown for these cases in Table E-4.

Certain instructions, like LDIR and LDM, store more than one operand to memory. The memory delays for such instructions are included in their execution times listed in Table E-2 based on the following assumptions: the operands are aligned, the memory is 32 bits wide, and $T_W + S/2$ is four processor cycles. If $T_W + S/2$ exceeds four processor cycles, then the excess must be counted as a memory delay for every operand stored by the instruction. Similarly, if operands are unaligned or the memory is 16 bits wide, then an additional memory delay must be counted for every stored operand, as shown in Table E-4. For example, if an LDIR instruction stores 3 aligned longwords to a 16-bit memory, then the instruction is delayed by $T_W + S/2$ processor cycles for each of three operands, or $3T_W + 3S/2$ processor cycles.

The CPU attempts to use burst-write transactions to store operands for ENTER (registers only), LDM, LDML, and EPA instructions. In storing an operand for these instructions, if the starting address is not aligned to the size of the memory's width (either 16 or 32 bits), the CPU issues one or two single-write transactions to store the operand's initial bytes until an aligned address is reached. Then, while one or more operand words (for 16-bit memory) or longwords (for 32-bit memory) remain to be transferred, the CPU transfers the first word or longword and attempts to burst transfer the remaining words or longwords until all transfers are complete, the number of remaining bytes is smaller than the memory's width, the end of a 16-byte block is reached, or the memory indicates that it cannot support further burst transfers. If any bytes remain to be stored, the CPU issues one or two single-write transactions to store the final bytes.

Table E-4. Memory Delays for Storing Word and Longword Operands

Address Bits A ₁ A ₀	Data Type	Bus Width (bits)	Memory Delay (Processor Cycles)
00	W	16	0
		32	0
	L	16	$T_W + S/2$
		32	0
01	W	16	$T_W + S/2$
		32	$T_W + S/2$
	L	16	$2T_W + S$
		32	$2T_W + S$
10	W	16	0
		32	0
	L	16	$T_W + S/2$
		32	$T_W + S/2$
11	W	16	$T_W + S/2$
		32	$T_W + S/2$
	L	16	$2T_W + S$
		32	$2T_W + S$

For example, assume the CPU is storing seven longwords to memory location 13 to execute an ENTER instruction and that the memory system is 32 bits and supports burst-write transfers of 16-byte blocks. Then the CPU performs five transactions to store the seven longwords:

1. Store a single byte at location 13.
2. Store a word at location 14.
3. Burst transfer four longwords to store at location 16.
4. Burst transfer two longwords to store at location 32.
5. Store a single byte at location 40.

Thus, using memory systems that support burst-write transactions, the execution time for ENTER, LDM, LDML, and EPA instructions are less than the values shown in Table E-2. To calculate the appropriate instruction execution time for such systems, add the number of cycles to perform the memory references (for LDM, LDML, and EPA instructions if the last transaction is not a burst transfer, count only one cycle for it) to 15 for ENTER, 3 for LDM, 6 for LDML, and 4 for EPA instructions.

Timing Formulae for Performance Evaluation

Performance Calculation

In order to determine the CPU's performance for a program workload, the average number of processor cycles per instruction for execution (T_E), pipeline (T_p), addressing (T_A), and memory (T_M) delays can be calculated by measuring the frequency of occurrence for the various delay causes and using the formulae presented in previous sections. The average number of processor cycles per instruction (T_I) can be estimated by adding the individual delay components as shown below.

$$T_I = T_E + T_p + T_A + T_M$$

Since two clock cycles are in every processor cycle, the following formula gives the performance of a CPU whose clock cycle time is T_C .

$$\text{Performance} = (2T_I T_C)^{-1}$$

Because certain details of the CPU's operation have been omitted to simplify the description and analysis presented in this appendix, the formula above gives only an approximate prediction of the processor's actual performance. In general, the analysis is conservative; performance will typically be better than predicted because the simultaneous occurrence of two or more delay causes has been ignored. For example, the CPU can handle a cache miss for one instruction while executing another multiple-cycle instruction, like DIV. But, the time during which the delay causes are overlapping is counted twice because execution and memory delays are separately calculated. Nevertheless, the analysis described above is extremely useful, though inexact, because it is much simpler and faster than a register-transfer-level simulation necessary for exact performance calculations.

Table E-5. Exception Processing Times

Exception	Processing Delay	Notes
Bus Error	29	
Non-maskable interrupt	21	
Vectored interrupt	26	
Non-vectored interrupt	21	
Extended Instruction trap	23	
Privileged Instruction trap	23	
System Call trap	22	
Address Translation trap	24	Add 11 cycles if access protection violation detected for translation table descriptor register. Otherwise add number of cycles given in Table E-3 to access levels of translation table until exception detected.
Breakpoint	22	
Integer Overflow trap	20	
Bounds Check trap	26	Source operand below lower bound
	28	Source operand above upper bound
Index Error trap	26	Source operand below lower bound
	28	Source operand above upper bound
Conditional trap	23	
Unimplemented Instruction trap	23	
PC trap	23	
Trace trap	20	

Note 1: For all exceptions, add the time to store Program Status registers onto the System Stack and to load Program Status registers from the Program Status Area in external memory.

Note 2: For Bus Error and Address Translation exceptions, also add the time to store the violation longword address onto the System Stack.

Note 3: For interrupts, add the time for the Interrupt Acknowledge transaction.

Exception Processing Delays

In addition to processing instructions, the CPU must occasionally process exceptions. Table E-5 lists the delays incurred for processing various types of exception. Calculating the delays involves determining the time to store the Program Status registers to memory and fetching new values for the Program Status register from the Program Status Area. For example, assume that the time for a single memory-read transaction is 2 processor cycles and the time for a single memory-write transaction is 3 processor cycles, the memory data path is 32 bits, and the bus-timing scale factor is 2. Then the time to store and fetch the Program Status is 13 processor cycles: The 4 memory references require 3 processor cycles each, and an idle bus cycle follows each of the first 3 references. Thus, the delay for processing a System Call trap is 35 processor cycles.

Example

This section describes an example of performance evaluation for a workload containing fifteen programs representative of 16-bit microprocessor applications. The programs are all written in C and run in normal compact mode under Zilog's ZEUS version of the UNIX* operating system. Table E-6 lists the programs in the workload, which includes five million executed instructions.

Table E-6. Program Workload Used for 280,000 CPU Performance Evaluation

Program	Use
C1	C compiler parser
C2	C compiler code generator
C3	C compiler optimizer
C4	C compiler lister
CPP	C compiler preprocessor
DIFF	File comparison
ED	Line editor
GREP	Pattern searching
LS	File directory listing
NM	Load module name listing
OD	Octal dumping of core images
PR	Format for line printer
SED	Stream editor
SORT	Sorting
VI	Screen editor

*UNIX is a trademark of AT&T Bell Laboratories.
Zilog is licensed by AT&T Technologies, Inc.

In order to calculate the frequencies of the various delay components, the programs were interpreted by a software simulator for the CPU's instruction set. The performance was then determined for systems composed of a 12 MHz CPU and each of three different memories that varied in their data path size and support for burst transfers.

The execution delay for the workload was determined from the frequency distribution of instruction. Table E-7 shows the ten most commonly executed instructions and their frequencies as a percentage of total instructions. The average execution delay is 1.8 processor cycles per instruction.

Table E-7. Most Commonly Executed Instructions

Opcode	Instruction	Addressing Mode	Frequency (percent)
JR		RA	19.0
LD(register)		R	10.7
INC		R	7.9
CP(register)		IM	4.7
LD(register)		X	4.4
LDB(register)		IR	4.1
DEC		R	3.3
EXTSB		R	3.2
LD(memory)		X	2.1
LD(memory)		IR	2.0

The average pipeline delay per instruction is 0.3 processor cycle. A register interlock occurs for 11% of instructions, causing 0.19 processor cycle delay, and a cache reference interlock occurs for 6% of instructions, causing 0.11 processor cycle delay.

Addressing delays are 0.03 processor cycles per instruction. These delays result almost entirely from branches to unaligned two-word instructions, because the compiler positions operands at aligned addresses and page-crossings rarely occur during sequential instruction processing.

In calculating memory delays, three memory systems were considered. The first memory has a 16-bit data path, a cycle time of 2 processor cycles for read and 3 processor cycles for write and no burst transfers. The second and third memories have 32-bit data paths and cycle times of 2 processor cycles for read and 3 processor cycles for write, but the third supports burst transfers whereas the

Timing Formulae for Performance Evaluation

second does not. All three systems use a bus clock scaled by a factor of 2 from the CPU's clock.

To determine the average delay caused by cache misses it is useful to compute the average number of misses per instruction, μ . To calculate μ , it is necessary to know the cache hit ratio (h), which is the fraction of fetched words that are located in the cache, and the average number of fetched words per instruction. For this workload, an average of 1.4 instruction words and 0.3 operand word are fetched per instruction. Therefore, the average number of cache misses per instruction is given by $\mu = 1.7(1-h)$, and the average delay per instruction due to cache misses is 2μ . The values of cache hit ratio, misses per instruction, and delays per instruction are shown in Table E-8.

Table E-8. Cache and TLB Miss Delays

Memory System	16-Bit	32-Bit	32-Bit
	No Burst	No Burst	Burst
Cache Performance			
Hit ratio	0.62	0.75	0.88
Misses per instruction	0.65	0.42	0.21
Delay per instruction	1.3	0.84	0.42
TLB Performance			
Hit Ratio	0.99	0.99	0.99
Misses per instruction	0.02	0.02	0.02
Delay per instruction	0.57	0.46	0.46

Calculating the average delay caused by TLB misses is similar to cache misses, as described above, but operand stores as well as fetches can cause TLB misses. This is because the physical frame address in the page table entry is needed to store an operand. On an average, 0.15 operand word is stored per instruction. The delay to service a TLB miss for two-level translation tables can be derived from the formulae previously given in the section on memory delays: 31 processor cycles with the 16-bit memory and 25 processor cycles with the 32-bit memory. The values of TLB hit ratio, misses per instruction, and delays per instruction are shown in Table E-8.

In addition, the delay caused by bus contention amounts to 0.2 processor cycle per instruction for all of the memory systems. In general, a 32-bit memory would exhibit less bus contention than a 16-bit memory, but the memory systems show negligible difference in bus contention for this workload, which makes little use of longword operands. (Fewer than 2% of memory operands are longwords.)

The performance of a 25 MHz CPU with each of the three memory systems is calculated by adding the various delay components. The results, summarized in Table E-9 show the performance ranges from 3.1 to 5.0 million instructions per second (MIPS). For short sequences of instructions executed repeatedly, it is possible to approach the maximum performance of 12.5 MIPS.

Table E-9. Processing Performance

Memory System	Performance (MIPS)*	$T_I = T_E + T_p + T_A + T_M$	
		Processor	Cycles Per Instruction
16-bit no-burst	3.1	4.0	= 1.8 + 0.3 + 0.0 + 1.9
32-bit no burst	3.7	3.4	= 1.8 + 0.3 + 0.0 + 1.3
32-bit burst	4.2	3.0	= 1.8 + 0.3 + 0.0 + 0.90
32-bit burst, no translation	5.0	2.5	= 1.8 + 0.3 + 0.0 + 0.4

* The analysis used in calculating the performance is conservative; the delays are independently calculated, but in practice the delays may often overlap. Consequently the actual performance may be better than the values shown in the table.

G

Zilog

Glossary

access protection: A function of memory management that controls read, write and execute access to memory locations, protecting proprietary or operating system memory areas from tampering by unauthorized users. The CPU uses the protection (PROT) field to determine access rights for a page or segment.

access protection violation: An incorrect or forbidden attempt to access a memory location; for example, an attempt to write to a read-only page. An access violation causes the CPU to generate an Address Translation trap.

activation record: A data structure containing the local storage, saved register contents, and exception handler address associated with the invocation of a procedure. Activation records are stored on the processor stack in a linked list. An activation record is allocated when the Enter instruction is executed at the beginning of a procedure. The record is released when the Exit instruction is executed at the end of a procedure.

addressing mode: The way in which the location of an operand is specified. There are nine addressing modes: Register, Immediate, Indirect Register, Direct Address, Index, Base Address, Base Index, Relative Address, and Relative Index.

address tag: The portion of certain associative memories that is compared against a referenced address to determine whether the matching value is found. The address tag for a Translation Lookaside Buffer entry is the logical page address; the address tag for a cache block is the physical memory address.

address translation: The process of mapping logical addresses into physical addresses.

Address Translation trap: An exception that occurs during address translation when either an access protection violation or an invalid table entry is detected. The instruction being executed is suspended, and the PC, FCW, identifier word, and the logical address that caused the trap are saved on the system stack.

aligned address: An address that is a multiple of an operand's size in bytes. Aligned word addresses are a multiple of two; aligned longword addresses are a multiple of four.

associative memory: A memory in which data is accessed by specifying a value rather than a location. The Translation Lookaside Buffer and cache are associative memories.

autodecrement: The operation of decrementing an address in a register by the operand's size in bytes. The decrement amount is one for byte operands, two for word operands, and four for longword operands.

autoincrement: The operation of incrementing an address in a register by the operand's size in bytes. The increment amount is one for byte operands, two for word operands, and four for longword operands.

base address: The address used, along with an index and/or displacement value, to calculate the effective address of an operand. The base address is located in a general-purpose register, the Program Counter, or the instruction.

Base Address (BA) addressing mode: In this mode, the displacement in the instruction is added to the contents of the base register to obtain the effective address.

Base Index (BX) addressing mode: In this mode, the contents of the base register and index register are added to the displacement in the instruction to obtain the effective address.

bit field: One to thirty-two contiguous bits that can cross byte boundaries. A bit field is specified by its byte origin, its bit position from the origin, and its size in bits. The instruction set allows bit fields to be extracted from a longword and inserted into a longword.

burst transaction: The transfer of several consecutive items of data (either words or longwords) in one memory transaction.

Glossary

bus error: An exception that occurs when external hardware identifies an irrecoverable error during a data transfer on the external interface.

bus master: The device in control of the bus.

bus retry: A response to a data transfer transaction that indicates the transaction must be tried again because of some transient error condition.

byte: A data item containing 8 contiguous bits. A byte is the basic data unit for addressing memory and peripherals.

cache: An on-chip buffer that automatically stores copies of recently used memory locations (both instructions and data), allowing fast access on memory fetches.

compact mode: A mode of address representation, usually used for applications with small memory requirements, in which 16-bit addresses are manipulated; address calculations involve all 16 bits. The logical address is extended to 32 bits by concatenating the 16 most-significant bits of the Program Counter.

completion: An instruction ending in which the current instruction has been completely executed. This is the normal instruction ending, but exceptions can cause a different ending.

coprocessor: A processor, such as a Z8070 Arithmetic Processing Unit, that works synchronously with the CPU to execute a single instruction stream using the Extended Processing Architecture (EPA).

Direct Address (DA) addressing mode: In this mode, the effective address is contained in the instruction.

displacement: A constant value located in the instruction that is used for calculating the effective address of an operand.

dynamic operation: A bit manipulation operation in which the source operand is located in a register and therefore its value is changeable.

effective address: The logical memory address of an operand, calculated by adding the base address, an optional index value, and an optional displacement.

EPU internal operation: An EPU-handled operation that controls EPU operations but does not transfer data.

exception: A condition or event that alters the usual flow of instruction processing. The Z80,000 CPU supports four types of exception: reset, bus error, interrupts, and traps. When an exception occurs, the CPU saves the Program Status on the system stack and fetches a new Program Status from the Program Status Area.

exception processing state: A CPU operating state that results when an exception occurs, during which the CPU stores values from the Program Status registers to memory, and fetches values from memory for the Program Status registers.

execute access: The type of memory access used by the CPU for fetching instructions and immediate mode operands.

Extended Addressing Mode (EAM): An addressing mode in which one or more extension words follow the opcode. In compact mode, EAMs are Direct Address and Index. In segmented or linear mode, EAMs are Direct Address, Index, Base Address, Base Index, Relative Address and Relative Index.

Extended Processing Architecture (EPA): A CPU facility controlled by the EPA bit in the Flag and Control Word that allows the operations defined in the architecture to be extended by hardware or software. If enabled, the CPU transfers EPA instructions to an Extended Processing Unit (EPU) for execution; if disabled, the CPU traps EPA instructions for software emulation.

Extended Processing Unit (EPU): An external device, such as a Z8070 APU, that handles Extended Processing Architecture instructions (such as floating-point arithmetic).

Flag and Control Word (FCW) register: One of the two Program Status registers, a 16-bit register that contains the flags and bits that control the operation of the CPU.

flyby transaction: A transaction controlled by the bus master, but in which another device transfers data to the responding device.

frame: A 1K-byte physical memory unit used by the memory management mechanism to map 1K-byte logical memory pages. A frame is specified by the 22 most-significant bits of the physical address.

Frame Pointer (FP): The register that points to the current activation record on the stack. In compact mode, the FP is a word register, R14; in segmented or linear mode, a longword register, RR12.

general-purpose registers: The 16 versatile registers that can be used as data accumulators, index values, or memory pointers.	least recently used (LRU): The CPU records the order of use for Translation Lookaside Buffer entries and cache blocks. When a TLB miss or cache tag miss occurs, the CPU replaces the least recently used entry or block.
global bus: A bus shared by tightly-coupled, multiple CPUs; the bus master is chosen by an external arbiter device.	length counter: A register that contains the value that is the length of a block or string of data that is manipulated by instructions.
halted state: A CPU operating state that results when a Halt instruction is executed or a bus error exception occurs during exception processing.	linear mode: A mode of address representation in which 32-bit addresses are manipulated, providing uniform and unstructured access to the 4G bytes of memory. Address calculations involve all 32 bits.
Hardware Interface Control register (HICR): The 32-bit special-purpose register that specifies certain characteristics of the hardware configuration incorporating the CPU, such as bus speed, memory data path width, and number of wait states.	local bus: The bus controlled by the CPU and shared with slave processors.
hit: A hit occurs when an associative memory is searched for a value and a match is found.	logical address: The address manipulated by the program. The memory management mechanism translates logical addresses to physical addresses.
identifier word: A 16-bit code saved on the system stack during exception processing that provides information about the cause of the exception.	longword: A data item containing 32 contiguous bits.
Immediate (IM) addressing mode: In this mode, the operand is contained in the instruction.	loosely-coupled CPUs: CPUs that execute independent instruction streams and communicate through a multi-ported peripheral, such as a Z8038 FIO I/O interface unit.
index: A value located in a register used for calculating the effective address of an operand. The index value usually specifies the calculated offset of an operand from the origin of an array or other data structure.	memory management: The process of translating logical addresses into physical addresses, plus certain protection functions. In the Z80,000 CPU, memory management is integrated into the chip.
Index (X) addressing mode: In this mode, the contents of an index register are added to a base address contained in the instruction to obtain the effective address.	memory-mapped I/O: A memory management feature that allows logical memory addresses to be mapped to physical I/O addresses. Memory mapped I/O provides protected access by application programs to peripherals.
Indirect Register (IR) addressing mode: In this mode, the effective address is contained in a register.	miss: A miss occurs when an associative memory is searched for a value and no match is found.
instruction executing state: A CPU operating state in which the CPU executes instructions.	nonmaskable interrupt: The highest priority interrupt; cannot be disabled.
interrupt: An asynchronous exception that occurs when the <u>NMI</u> , <u>VI</u> , or <u>NVI</u> line is activated, usually when a peripheral device needs attention.	nonvectored interrupt: The lowest priority interrupt, which does not use an identifier word as a vector to an interrupt service routine; can be disabled.
invalid table entry: A cause of an Address Translation trap that is detected during address translation if the CPU fetches a translation table entry with a Valid bit of 0.	normal mode: A CPU mode of operation, generally used for application programs, in which the S/N flag in the FCW is 0. In this mode, the CPU cannot execute privileged instructions or access protected memory locations.
large segment: In the segmented mode, one of the 128 segments in the upper half of the memory address space. Segments are 16M bytes in size or smaller.	

Glossary

Normal Stack Pointer (NSP): The Stack Pointer used while the CPU is in normal mode. System mode programs can access the NSP with the Load Control instruction.

overflow stack: The stack used for saving the Program Status, identifier word, and exception parameters when an address translation exception occurs during exception processing.

Overflow Stack Pointer (OSP): The 32-bit register that contains the physical address of the overflow stack.

page: A 1K-byte logical memory unit mapped by the memory management mechanism to a 1K-byte physical memory frame. A page is specified by the 22 most-significant bits of the logical address.

page table: The third level of translation tables, containing the physical frame address used during address translation.

paged translation: A method of address translation in which the logical and physical address spaces are divided into fixed, equal-sized units called pages and frames, respectively. During address translation, a logical page is mapped to an arbitrary physical frame.

partial completion: An instruction ending in which the execution of an interruptible instruction is disrupted before completion by a trap or interrupt.

physical address: The 32-bit address required for accessing memory and peripherals, obtained by the CPU's address translation hardware.

pipeline: A computer design technique in which an instruction is executed in a sequence of stages by different functional units. The functional units can be operating on several different instructions simultaneously, similar to an automobile assembly line.

prefetching: Ability of the CPU to fetch an instruction or operand before the previous instructions have been completed.

privileged instruction: An instruction that performs I/O operations, accesses control registers, or performs some other operating system function. Privileged instructions execute in system mode only.

Program Counter (PC): One of the two Program Status registers, a 32-bit register that contains the address of the current instruction.

Program Status registers: The two registers (Program Counter and Flag and Control Word) that contain the Program Status. The Program Status is automatically saved during exception processing.

Program Status Area (PSA): The area in memory reserved for storing the Program Status of the interrupt and trap service routines.

Program Status Area Pointer (PSAP): The 32-bit register that contains the physical, base address of the Program Status Area.

protection: See access protection.

protection (PROT) field: A 4-bit field contained in the translation table descriptor registers and translation table entries that specifies access protection information for a logical address during address translation.

quadword: A data item containing 64 contiguous bits.

read access: The type of memory access used by the CPU for fetching data operands other than those specified by Immediate mode.

Register (R) addressing mode: In this mode, the operand is in a general-purpose register.

Relative Address (RA) addressing mode: In this mode, the displacement in the instruction is added to the contents of the Program Counter to obtain the effective address.

Relative Index (RX) addressing mode: In this mode, the contents of the Program Counter and index register are added to the displacement in the instruction to obtain the effective address.

relocation: The process of mapping a logical address to a different physical address, so that multiple processes can use the same logical address for distinct physical memory locations.

reset: A CPU operating state or exception that results when a reset request is signaled on the RESET line. A reset initializes the Program Status registers.

responder: The device to which bus transactions transfer data.

result register: The register that holds the result of an operation.

segmented mode: A mode of address representation that supports either 64K- or 16M-byte segments with 32-bit addresses. The most-significant address bit selects either a 15-bit segment number with 16-bit offset, or a 7-bit segment number with 24-bit offset. Calculations affect only the offset and not the segment number.

self-modifying program: A program that stores to a location from which a subsequent instruction is fetched.

slave processor: A processor, such as a Direct Memory Access transfer controller, that performs dedicated functions asynchronously to the CPU.

small segment: In the segmented mode, one of the 32,768 segments in the lower half of the memory address space. Segments are 64K bytes or smaller.

spatial locality: The characteristic of program behavior whereby consecutive memory references often apply to closely located addresses.

special-purpose control registers: Nine registers used for system configuration, memory management, Program Status, and CPU control.

Stack Pointer (SP): A general-purpose register indicating the top (lowest address) of the processor stack used by Call, Enter, Exit, and Return instructions for linking procedures. The SP is a word register, R15, in compact mode, and a longword register, RR14, in linear or segmented mode. Normal and system modes of operation use separate stack pointers, the Normal Stack Pointer (NSP) and System Stack Pointer (SSP).

static operation: A bit manipulation operation in which the source operand is an immediate value and is therefore fixed (static).

suspension: An instruction ending in which the current instruction has not been completed because a trap is detected during instruction execution. The instruction can be completed by eliminating the cause of the trap and starting the instruction again.

suspension with PC modification: An instruction ending similar to suspension, but the Program Counter saved on the system stack during exception processing must be decremented by two before starting the instruction again.

System Configuration Control Longword register (SCCL): The 32-bit special-purpose register that contains control bits for address translation, cache, and exception processing.

system mode: A CPU mode of operation, used for operating system functions, in which the S/N flag in the FCW is 1. In this mode, the CPU can execute privileged (and all other) instructions.

System Stack Pointer (SSP): The Stack Pointer used while the CPU is in system mode. Normal mode programs cannot access the SSP.

tag hit: On a memory reference, a tag hit occurs when the cache address tags are searched for the referenced address and a match is found.

tag miss: On a memory reference, a tag miss occurs when the cache address tags are searched for the referenced address and no match is found.

temporal locality: The characteristic of program behavior whereby memory references often apply to a location that has been referred to recently.

termination: An instruction ending in which the current instruction has not been completed and it is not possible to complete the instruction by starting it again.

tightly-coupled CPUs: CPUs that execute independent instruction streams and communicate through shared memory on a common (global) bus.

Translation Lookaside Buffer (TLB): An on-chip memory that automatically stores translation information for the most recently used memory pages.

translation table: One of three levels of tables selected by the page descriptor registers during address translation. Each level corresponds to a field in the logical page address.

translation table descriptor register: One of four registers that contain the physical addresses of the translation tables used by the memory management mechanism during address translation.

translation table entry: An entry in one of the three levels of translation tables. Entries in the first two levels point to another level table. Entries in the third level (page table) contain the physical frame address used during translation.

trap: An exception that occurs when certain conditions, such as an access protection violation, are detected during execution of an instruction.

Glossary

unaligned address: An address that is not a multiple of an operand's size in bytes. Odd addresses are unaligned for words and longwords; even addresses that are not multiples of four are unaligned for longwords.

vectored interrupt: An interrupt that uses the low-order byte of the identifier word as a vector to an interrupt service routine; can be disabled.

virtual memory: A memory management technique in which the system's logical memory address space is not necessarily the same as, and can be much larger than, the available physical memory.

word: A data item containing sixteen contiguous bits.

word hit: On a memory reference to the cache, a tag hit occurs and a valid copy of the word is stored in the cache.

word miss: On a memory reference to the cache, a tag hit occurs but a valid copy of the word is not stored in the cache.

write access: The type of memory access used by the CPU for storing data operands.

I

Zilog

Index

-A-

Access protection, 1:2, 4:8
Address calculations, 5:1,3
Addressing delays, E:15
Addressing modes, 4-14, 5:2
 Base Address (BA), 5:6,10
 Base Index (BX), 5:6,11
 Direct Address (DA), 5:5,8
 Extended, 5:14
 Immediate (IM), 5:4,8
 Index (X), 5:5,9
 Indirect Register (IR), 5:4,8
 Relative Address (RA), 5:7,12
 Relative Index (RX), 5:7,13
 used in compact mode, 5:4-7
 used in segmented and linear modes, 5:7-12
Address representation, 1:1, 3:1-2
 Compact mode, 1:1, 3:1-2
 Linear mode, 1:2, 3:1-2
 Segmented mode, 1:1-2, 3:1-2
Address spaces, 1:1, 4:1-3
Address translation, 1:2, 4:4
Address Translation trap, 7:4
Architecture, 1:1
Arithmetic instructions, 6:2-3
Assembler language syntax, 6:10-11

-B-

Base Address (BA) addressing mode, 5:6,10
Base Index (BX) addressing mode, 5:6,11
Bit Field instructions, 6:5-6
Bit Manipulation instructions, 6:5
Block diagram of Z80,000 CPU, 1:6
Block Transfer and String Manipulation
 instructions, 6:7
Breakpoint trap, 7:4
Burst Memory Read and Write transaction, 8:10-11
Burst Memory Read timing, 8:12
Burst Memory Write timing, 8:13
Burst transfer protocol, 8:11
Bus acknowledge, 8:22-24
Bus error, 7:4, 8:21
Bus operations, 8:1
Bus request, 8:22-24
Bus Request Acknowledge timing, 8:22

Bus request protocol, 8:24
Bus retry, 8:21
Bus timing, 8:6-7
Bus transaction response, 8:8
Bus transactions, 8:7-21
 Burst Memory Read and Write transaction, 8:10-11
 CPU-EPU Data transactions, 8:17-18
 CPU-EPU Instruction transactions, 8:16
 CPU-Memory transactions, 8:8-12
 EPU-Memory transactions, 8:18-19
 EPU transactions, 8:13-15
 Input/Output transactions, 8:12-13
 Interlocked Memory transactions, 8:11-12
 Single Memory Read and Write transactions,
 8:8-10

-C-

Cache, 1:4, 8:3, C:1-3
Compact mode, 1:1, 3:1-2, 6:13
Compatibility with Z8000 CPU, 1:6, A:1
Condition codes, 6:9-10
Conditional trap, 7:4
Coprocessor, 1:4, 8:2
CPU Bus Request Protocol, 8:24
CPU Control instructions, 6:8-9
CPU-EPU Data Read timing, 8:17
CPU-EPU Data transactions, 8:17-18
CPU-EPU Data Write timing, 8:18
CPU-EPU Instruction transactions, 8:16
CPU-EPU Instruction Transfer timing, 8:16
CPU internal organization, 1:5-6
CPU-Memory transactions, 8:8-12

-D-

Data formats, 2:1
Demand-paged virtual memory, 1:2
Direct Address (DA) addressing mode, 5:5,8

-E-

EPA, see Extended Processing Architecture
EPU-Memory Single Write timing, 8:19 EPU-Memory
 transactions, 8:18-19

Index**-E- (Continued)**

EPU transactions, 8:13-15
Exception handlers, 7:7
Exception processing delays, E:19-20
Exceptions, 1:3, 7:3-9
 Bus error, 7:4
 Interrupts, 7:4
 Priority of, 7:8-9
 Reset, 7:3
 Traps, 7:4, 5
Execution time, E:3
Extended addressing modes, 5:14, 6:13
Extended Instructions, 6:9
Extended Instruction trap, 7:4
Extended Processing Architecture (EPA), 1:3, 4
External interface, 1:4, 8:1-27

Descriptions and formats, 6:16-214
Extended Instructions, 6:9
Flags and condition codes, 6:9-10
Input/Output instructions, 6:8
Load and Exchange instructions, 6:1-2
Logical instructions, 6:3-4
Notation and binary encoding used in, 6:10-12
Program Control instructions, 6:4-5
Integer Arithmetic Error trap, 7:4
Interlocked Memory transactions, 8:11-12
Internal Operation and Halt timing, 8:21
Internal Operation and Halt transactions, 8:21
Interrupt Request/Acknowledge timing, 8:20
Interrupt Request and Acknowledge, 8:20-21
Interrupts, 1:3, 7:4
 Non-maskable, 1:3
 Non-vectored, 1:3
 Vectored, 1:3 I/O Read timing, 8:14

-F-

Flag and Control Word register, 1:1, 2:2-3
Flags, 6:9-10
Floating-point operations, 1:3
Flyby transactions, 8:1
Frame pointer, 1:1, 2:2

Linear mode, 1:2, 3:1-2, 6:13-15
Load and Exchange instructions, 6:1-2
Local Bus Request Acknowledge timing, 8:22
Logical instructions, 6:3-4
Logical I/O address spaces, 4:3
Logical memory address spaces, 4:1
Loosely-coupled multiple CPU, 1:5

-G-

General-purpose register file, 2:1-2
Global Bus Request timing, 8:23

-H-

Hardware Interface Control register (HICR), 2:4,
8:5-6

Memory delays, E:15-18
Memory management, 1:2, 4:3
Memory-mapped I/O, 8:1
Multiprocessor Configurations, 1:4, 8:2
 Coprocessor, 1:4 Slave processor, 1:5
 Tightly-coupled multiple CPUs, 1:5
 Loosely-coupled multiple CPUs, 1:5

-I-

Immediate (IM) addressing mode, 5:4, 8
Index (X) addressing mode, 5:5, 9
Indirect Register (IR) addressing mode, 5:4, 8
Input/Output instructions, 6:8
Instruction execution, 7:2-3
Instruction format, 6:12
Instruction Set, 6:16-214
 Arithmetic instructions, 6:2-3
 Bit Field instructions, 6:5-6
 Bit Manipulation instructions, 6:5
 Block Transfer and String Manipulation
 instructions, 6:7
 CPU Control instructions, 6:8-9

-N-

Non-maskable interrupts, 7:4
Non-vectored interrupts, 7:4
Normal mode, 1:3, 3:2-3
Normal Stack Pointer, 2:3, 3:2

-O-

Odd PC trap, 7:5
Operating states, 7:1
Overflow Stack Pointer (OSP), 2:4

-P-

Physical address space, 4:3
Pin Functions, 8:3-4
Pipeline delays, E:3-4
Pipelined instruction execution, 7:3
Privileged Instruction trap, 7:4
Program Control instructions, 6:4-5
Program Counter, 1:1, 2:3
Program Status, 7:5-7
Program Status Area Pointer (PSAP), 2:3
Program Status registers, 1:1, 2:2-3
Program Counter, 1:1, 2:3
Flag and Control Word register, 1:1, 2:2-3

-R-

Register (R) addressing mode, 5:4,7
Relative Address (RA) addressing mode, 5:7,12
Relative Index (RX) addressing mode, 5:7.13
Reserved control bits, 2:4
Reset, 8:26-27

-S-

Segmented mode, 1:1-2, 3:1-2, 6:13-15
Single Memory Read and Write transactions, 8:8-10
Single Memory Read timing, 8:8-9
Single Memory Write timing, 8:10
Slave processor, 1:5, 8:2
Special-purpose control registers, 2:3
Stack Pointer, 1:1, 2:2
System Call trap, 7:4
System Configuration Control Longword, 2:4
System mode, 1:3, 3:2-3
System Stack Pointer, 3:2

-T-

Table entry formats, 4:7
Tightly-coupled multiple CPU, 1:5
Timing formulae, Appendix E
TLB, see Translation Lookaside Buffer
Trace trap, 7:5
Translation Lookaside Buffer (TLB), 1:2, 4:4-5
Translation Table Descriptor registers, 2:4
Traps, 7:4-5
 Address Translation trap, 7:4
 Breakpoint trap, 7:4
 Conditional trap, 7:4
 Extended Instruction trap, 7:4
 Integer Arithmetic Error trap, 7:4
 Odd PC trap, 7:5
 Privileged Instruction trap, 7:4
 System Call trap, 7:4
 Trace trap, 7:5
 Unimplemented Instruction trap, 7:5

-U-

Unimplemented Instruction trap, 7:5

-V-

Vectorized interrupts, 7:4

-Z-

Z8000 CPU, compatibility with, 1:6, A:1
Z8070 Arithmetic Processing Unit, 1:3
Z80,000 CPU block diagram, 1:6



READER COMMENTS

Your comments concerning this publication are important to us.
Please take the time to complete this questionnaire and return it to
Zilog.

Title of Publication: _____

Document Number: _____

Your Hardware Model and Memory Size: _____

Describe your likes/dislikes concerning this document:

Technical Information: _____

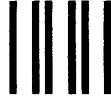
Supporting Diagrams: _____

Ease of Use: _____

Your Name: _____

Company and Address: _____

Your Position/Department: _____



No Postage
Necessary If
Mailed In The
United States

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 35, CAMPBELL, CA.

POSTAGE WILL BE PAID BY:

Zilog

1315 Dell Ave.
Campbell, California 95008
ATTENTION: Corporate Publications



Zilog Sales Offices and Technical Centers**West**

Sales & Technical Center
Zilog, Incorporated
10340 Bubb Road
Cupertino, CA 95014
Phone: (408) 370-8120
 (408) 370-8122 (G/S)
TWX: 910-338-2296
FAX: (408) 370-8016

Sales & Technical Center
Zilog, Incorporated
15643 Sherman Way
Suite 430
Van Nuys, CA 91406
Phone: (213) 989-7485
TWX: 910-495-1765

Sales & Technical Center
Zilog, Incorporated
125 Baker Ave
Suite 180
Costa Mesa, CA 92626
Phone: (714) 261-1281

Sales & Technical Center
Zilog, Incorporated
1750 112th Ave. N.E.
Suite D161
Bellevue, WA 98004
Phone: (206) 454-5597

Technical Center
Zilog, Incorporated
2885 Aurora Ave.
Suite 23
Boulder, CO 80303
Phone: (303) 440-3971

Midwest

Sales & Technical Center
Zilog, Incorporated
951 North Plum Grove Road
Suite F
Schaumburg, IL 60195
Phone: (312) 885-8080
TWX: 910-291-1064

Technical Center
Zilog, Incorporated
7101 York Ave., South
Edina, MN 55435
Phone: (612) 921-3369

Sales & Technical Center
Zilog, Incorporated
28349 Chagrin Blvd.
Suite 109
Woodmere, OH 44122
Phone: (216) 831-7040
FAX: 216-831-2957

South

Sales & Technical Center
Zilog, Incorporated
4851 Keller Springs Road.
Suite 211
Dallas, TX 75248
Phone: (214) 931-9090
TWX: 910-860-5850

Technical Center
Zilog, Incorporated
7113 Burnet Rd.
Suite 207
Austin, TX 78757
Phone: (512) 453-3216

East

Sales & Technical Center
Zilog, Incorporated
Corporate Place
99 South Bedford St.
Burlington, MA 01803
Phone: (617) 273-4222
TWX: 710-332-1726

Sales & Technical Center
Zilog, Incorporated
240 Cedar Knolls Rd.
Cedar Knolls, NJ 07927
Phone: (201) 540-1671

Technical Center
Plaza Office Center
Suite 412
Route 73 and Fellowship Rd.
Mt. Laurel, NJ 08054
Phone: (609) 778-8070

Technical Center
Zilog, Incorporated
3300 Buckeye Rd.
Suite 401
Atlanta, GA 30341
Phone: (404) 451-8425

Sales & Technical Center
Zilog, Incorporated
1301 Seminole Blvd.
Suite 103
Largo, FL 33540
Phone: (813) 585-2533
TWX: 810-866-9740

United Kingdom

Zilog (U.K.) Limited
Zilog House
43-53 Moorbridge Road
Maidenhead
Berkshire, SL6 8PL England
Phone: 0628-39200
Telex: 848609

France

Zilog, Incorporated
Cedex 31
92098 Paris La Defense
France
Phone: (1) 334-60-09
TWX: 611445F

West Germany

Zilog GmbH
Eschenstrasse 8
D-8028 TAUFKIRCHEN
Munich, West Germany
Phone: 89-612-6046
Telex: 529110 Zilog d.

Japan

Zilog, Japan K.K.
Konparu Bldg. 5F
2-8 Akasaka 4-Chome
Minato-Ku, Tokyo 107
Japan
Phone: (81) (03) 587-0528
Telex: 242202 A/B: Zilog J

Hong Kong

Zilog Asia Ltd.
22-26 Austin Ave.
Room 1009 Austin Tower
Tsimshatsui Kowloon
Hong Kong
Phone: (852) (3) 723-8979
Telex: 52102 ZILOG HK