MIL-STD-1862B
3 January 1983
SUPERSEDING
MIL-STD-1862A
2 November 1981

MILITARY STANDARD

Nebula Instruction Set Architecture

FSC IPSC

Table of Contents

1. SCOPE AND PURPOSE 1
1.1. Scope 1
1.2. Purpose 1

2. REFERENCED DOCUMENTS 2
2.1. Issue of document 2

3. DEFINITIONS 3
3.1. Assembler Notation Conventions 4

4. OVERVIEW 5
4.1. Memory Organization 5
4.2. Instruction Format 5
4.3. Control Structure 5
4.4. Addressing 6
4.5. Data Types 6

5. OPERAND ADDRESSING MODES 7
5.1. Maximum Accessible Register 7
5.2. Compound Modes 7
5.3. Address Operands 8
5.4. Register Mode 9
5.5. Short Literal Mode 9
5.6. Literal Mode 10
5.7. Memory Addressing Modes 10
5.7.1. Indirect Register Mode 11
5.7.2. Register Indexed Modes 12
5.7.3. Absolute Mode 13
5.8. Parameter Addressing Modes 14
5.8.1. Short Parameter Mode 14
5.8.2. Extended Parameter Mode 14
5.8.3. General Parameter Mode 15
5.9. Unscaled Index Mode 16
5.10. Scaled Index Mode 17
5.11. Reserved Specifiers 18
5.12. Undefined Operand Sizes 18

6. PROCESSOR STATUS WORD 19
6.1. Kernel/Task Mode 19
6.2. Last Mode 19
6.3. Reserved Bits 19
6.4. Priority 19
6.5. Carry Condition Code 19
6.6. Truncate Condition Code 19

6.7. Negative Condition Code 20
6.8. Zero Condition Code 20
6.9. Debugging Control 20
6.10. Privilege 20
6.11. Base of Context 20
6.12. Supervisor Mode 20
6.13. Up/Down Level Exception Propagation (UDLE) 20
6.14. Enable Arithmetic Error 20
6.15. Maxreg Field 20
6.16. Number of Parameters 20

7. AUXILIARY STATUS REGISTER 21
7.1. Soft Memory Error Enable 21
7.2. Reserved Bits 21
7.3. Floating Point Mode Control 21
7.3.1. Infinity Control 21
7.3.2. Exception Event and Mask Bits 21
7.3.3. Rounding Control 21

8. PROCEDURE INTERFACE 22
8.1. The Context Stacks 22
8.1.1. Context Pointers 22
8.1.2. Structure of Context Stacks 23
8.1.3. Cacheing of the Context Stack 24
8.1.4. Changing of Context Stack Pointers 24
8.1.5. Alignment of Context Pointers 24
8.2. Procedure Descriptor 24
8.2.1. Fixed/Variable Number of Parameters 25
8.2.2. Reserved Bit 25
8.2.3. Exception Propagation Control 25
8.2.4. Arithmetic Error Control 25
8.2.5. Available Registers 25
8.3. Procedure Invocation 25
8.3.1. Determination of a New PSW 25
8.3.2. Register Set Allocation 26
8.3.3. Initialization of Exception Handler 26
8.3.4. Initialization of Parameter List 26
8.4. Parameter Lists 26
8.4.1. Parameter Specification 27
8.4.2. Parameter Access 27
8.4.3. Example of Parameter Linkage 27
8.4.4. Mechanism of Parameter Addressing 29
8.5. Vectored Calls: SVC and OPEX 31
8.6. Vector Format 33
8.7. Return from Procedures 33

9. EXCEPTIONS 34
9.1. Procedure Exception Handler 34
9.2. Definition of an Exception Handler 34
9.3. Raising an Exception 34

9.4. Exception Handling 35
9.5. Supervisor Exception Handler 37

10. DEBUGGING FACILITIES 38
10.1. Program Tracing 38

11. INTERRUPTS AND TRAPS 39
11.1. Interrupt Priority 39
11.2. I/O Interrupts 39
11.3. Software Interrupt Requests 39
11.4. Power Failure Interrupt 39
11.5. Power Restore Interrupt 40
11.6. Memory System Error Traps 40
11.6.1. Hard Memory Errors 40
11.6.2. Soft Memory Errors 41
11.7. Privileged Instruction Trap 41
11.8. Memory Management Traps 41
11.8.1. Instruction Execution 41
11.8.2. Task Context Stack 41
11.8.3. Kernel Context Stack 41
11.9. Reset and IPL 42
11.10. Built In Test Traps 42
11.11. Simultaneous Events 42

12. MEMORY MANAGEMENT SYSTEM 43
12.1. Virtual Address Space 43
12.2. Mapping of Virtual Addresses 44
12.2.1. Map Pointer Registers 44
12.2.2. Memory Map Structure 44
12.2.3. Segment Association 45
12.2.4. Relocation of Virtual Addresses 46
12.2.5. Access Protection 47
12.2.5.1. Self-Modifying Code 47
12.2.6. Crossing Segment Boundaries 47
12.2.7. Protection of the Supervisor 47
12.3. Implementation Considerations 48
12.3.1. Cacheing of Memory Maps 48
12.3.2. Aliasing of Physical Addresses 48
12.4. Memory Management Traps 48
12.5. I/O Space Selection 49
12.6. Subseting of Memory Management 49

13. I/O CONTROLLERS 50
13.1. Channel Configuration Registers 50
13.2. IOC Programs 50
13.2.1. Program Counter 51
13.2.2. Message Pointer Register 51
13.2.3. Accumulator 51
13.2.4. Channel Status 51
13.2.5. Channel Program Status 52

13.3. Virtual Addressing 52
13.3.1. Segment Specifiers 54
13.4. Physical Addressing 54
13.5. Instruction Execution 54
13.6. Operand Accessing 54
13.7. IOC Interrupts 54
13.7.1. IOC Error Interrupts 55
13.8. IOC Instructions 55
13.8.1. IOC Instruction Descriptions 55
13.8.2. Transfer Instructions 55
13.8.3. Control Instructions 59
13.8.4. Channel Specific Instructions 62
13.8.4.1. Parallel Point to Point Interface 62
13.8.4.2. Serial Point to Point Interface 64
13.8.4.3. MIL-STD-1553 Serial Interface 64
13.8.4.3.1. MIL-STD-1553 RT Mode Specific Instructions 64
13.8.4.3.2. MIL-STD-1553 BC Mode Specific Instructions 64
13.9. MIL-STD-1553B Remote Terminal Mode Operation 66
13.9.1. Message Pointer Register 66
13.9.2. Status Word 66
13.9.3. Vector Word 67
13.9.4. Transfer Commands 67
13.9.5. Mode Commands 67
13.9.6. RT Mode Interrupts 68
13.10. Interrupt Vector Assignments 68
13.11. IOC Control Register Assignments 69

14. TIMER SUPPORT 70
14.1. Time of Day 70
14.2. Interval Timers 70

15. ASSIGNED PHYSICAL ADDRESSES 71
15.1. Memory Space Assignments 71
15.2. I/O Space Assignments 72

16. CONCEPTUAL MODEL OF INSTRUCTION EXECUTION 73
16.1. The Serial Model 73
16.2. Effects of Parallelism 73
16.3. Instruction Fetch 75
16.4. Operand Address Calculation 76
16.5. Operand Fetch 76
16.6. Instruction Execution 76
16.7. Operand Storage 76
16.8. Memory Accesses 76
16.9. Interruptible Instructions 77
16.10. Serialization 77
16.11. IOC Serialization 77

17. INSTRUCTION DESCRIPTIONS 78
17.1. General Information 78

17.2. Operands 78
17.2.1. Signed Integers 78
17.2.2. Unsigned Integers 79
17.2.3. Logicals 79
17.2.4. Floating Point 79
17.2.5. Address Operands 79
17.3. Symbols and Functions 79

18. INTEGER ARITHMETIC 80
18.1. Integer Data Types 80
18.2. Integer Arithmetic Instructions 81

19. LOGICAL INSTRUCTIONS 95

20. SHIFT AND ROTATE INSTRUCTIONS 98

21. MOVE AND CLEAR INSTRUCTIONS 101

22. COMPARE AND TEST INSTRUCTIONS 105

23. CONTROL INSTRUCTIONS 109

24. PROCEDURE CALL AND RETURN INSTRUCTIONS 121

25. TASK CONTROL INSTRUCTIONS 126

26. EXCEPTION HANDLING INSTRUCTIONS 131

27. STRING INSTRUCTIONS 135

28. BIT FIELD INSTRUCTIONS 140

29. MISCELLANEOUS INSTRUCTIONS 145

30. FLOATING-POINT ARITHMETIC 155
30.1. Floating-Point Data Classes 155
30.2. Floating-Point Formats 155
30.3. Floating-Point Operands 156
30.4. Floating-Point Exceptions 156
30.4.1. Invalid.Operation 157
30.4.2. Divide.By.Zero 158
30.4.3. Floating.Overflow 158
30.4.4. Floating.Underflow 158
30.4.5. Floating.Inexact 158
30.5. Rounding 158
30.6. Infinity Arithmetic 159
30.7. Floating-Point Instructions 159

31. OPCODE ALLOCATION 177
31.1. Unimplemented Opcodes 177

32. NOTES 178

32.1. Prime Item Specification considerations 178
32.2. Implementation dependencies 179

List of Figures

Figure 4-1: Instruction Format 5
Figure 4-2: Integer Data Types 6
Figure 4-3: Floating Point Data Types 6
Figure 5-1: Operand Specifiers 8
Figure 6-1: Processor Status Word 19
Figure 7-1: Auxiliary Status Register 21
Figure 8-1: Stacked Procedure Contexts 23
Figure 8-2: Procedure Descriptor 24
Figure 8-3: Example of a Call Context 28
Figure 8-4: Normal Operand Processing 29
Figure 8-5: Parameter List Operand Processing 30
Figure 8-6: Parameter Access 30
Figure 8-7: SVC and OPEX Vector Registers 31
Figure 8-8: Vectored Call Address Calculation 32
Figure 8-9: Vector Format 33
Figure 9-1: States of an Exception Handler 34
Figure 9-2: Transfer to an Exception Handler 36
Figure 12-1: Map Data Structure 43
Figure 12-2: Memory Map Pointer Registers 45
Figure 12-3: Map Entry Format 45
Figure 12-4: Address Relocation 46
Figure 13-1: Channel Configuration Register 50
Figure 13-2: Channel Status Register 51
Figure 13-3: Channel Program Status 52
Figure 13-4: IOC Virtual Addressing 53
Figure 14-1: Timer Control Register 70
Figure 16-1: Conceptual model of instruction execution 74
Figure 30-1: Floating Point Number Representation 156
Figure 30-2: Floating Point Control Bits in ASR 157
Figure 30-3: The Compare Operation 171

1. Scope and Purpose

1.1. Scope. This standard defines the Nebula Instruction Set Architecture. The instruction set architecture includes all information required by a programmer in order to write any *time independent* program that will execute on computers conforming to this standard. This standard does not define any specific implementation details of a computer.

1.2. Purpose. The purpose of this document is to define the Nebula Architecture, independent of any specific implementation or vendor, with sufficient precision to permit independent implementations of this architecture that execute identical programs in the identical manner.

2. Referenced Documents

2.1. Issue of document. The following document, of the issue in effect on date of invitation for bid or request for proposal, forms a part of the standard to the extent specified herein. In the event of conflict between the documents referenced herein and the contents of this standard, the contents of this standard shall be considered a superseding requirement.

Standard

Military

MIL-STD-1553 - Aircraft Internal Time Division Command/Response Multiplex Data Bus

3. Definitions

The following terms are used in this document:

Cacheing
The term cacheing is used to refer to a variety of implementation techniques for enhancing performance by maintaining the active copy of some quantity in a place other than its assigned storage. The technique invoked may be as simple as the loading and storing of a register (file) or as complex as an associative memory scheme.

Implementation Dependent
The behavior of an action or feature described as implementation dependent is not defined by this document. The actual behavior seen by the programmer is dependent on the hardware he is using. This behavior will differ between machines.

Interruptible
An instruction designated as interruptible shall be restartable from the point of interruption.

Reserved
The use of reserved as a modifier indicates that the entity described is set aside for future definition. Unless otherwise specified, responsibility for definition of a reserved entity lies with the Nebula Control Board (NCB). Any assumptions about the behavior of such entities by anyone other than those responsible for the definition of the entity is forbidden.

Reserved (value) A reserved value is a bit pattern that, when written into a particular bit field, will cause unpredictable results. The use of such values by the software is forbidden.

Reserved (bit field) A reserved bit field is a sequence of bits that may produce a **Specification.Error** exception or unpredictable actions if set to a value other than zero. Reserved bits or reserved bit fields yield undefined results if read.

Undefined
An undefined result may produce arbitrary bit patterns in all fields that it was specified as modifying.

Unpredictable
An unpredictable action may produce any change in the state of the machine that is consistent with the rights of the program that caused it. For example, an unpredictable operation performed by a user task may destroy any of the locations it can access normally, but shall not destroy any state protected by the protection mechanisms. This restriction may be breached by a supervisor that gives a user access to critical memory or functions. For this reason, access to the CPU registers in the I/O space should be controlled by means of the memory management system. It should be noted that while the programmer cannot rely on any properties of an unpredictable action, it is considered desirable to make such actions as innocuous as is practical.

3.1. Assembler Notation Conventions. The notation used in example instructions in this document is a subset of the Nebula assembly language. A synopsis of the notation used is included below.

Types

Register numbers and parameter numbers are distinguished by prefix type specifiers. A number preceded by a % character is interpreted as a register number; a number preceded by a ? is interpreted as a parameter number. Examples:

```
%4              ;register 4
?3              ;parameter 3
```

Instruction Operands

Instruction operands are specified as a sequence of mode specifiers separated by commas. The syntax used to specify each mode is described below. Words enclosed in {} describe the contents of the field; the {} is not part of the assembler notation.

Syntax	Example	Addressing Mode
%{number}	%3	Register
#{value}	#52	Short Literal or Literal
@%{number}	@%4	Indirect Register
{index}(%{number})	16(%2)	Register Indexed
@({address})	@(64)	Absolute Address
?{number}	?4	Short or Extended Parameter
?({simple mode})	?(%4)	General Parameter
{base}({index})	@%3(%4)	Unscaled Index
{base}[{index}]	@%3[%4]	Scaled Index

Size Specifiers

Operands that address memory must also select the size of the data item referenced. Sizes are specified by following the mode specifier by a postfix size specifier. These specifiers are listed below:

↑B	Byte
↑H	Halfword
↑W	Word
↑D	Double Word

4. Overview

The Nebula architecture is a 32 bit general register architecture. Some of the characteristics of the architecture are:

- Variable length instructions with variable numbers of operands.
- Explicit addressing of all instruction operands.
- Multiple addressing modes for register, literal, memory and parameter access.
- Procedure based control structure with a local register set for each procedure.
- Support for several data types and representation lengths.
- Memory mapped I/O control.
- Vectored Interrupts and Exceptions.

4.1. Memory Organization. Main memory is byte addressable. The architecture provides for a virtual address space of 2^{32} 8-bit bytes. Bits are numbered from left to right; memory words are addressed by the address of their most significant byte with the following address locations (increasing addresses) containing the less significant bytes. Multi-byte data (halfwords, words, doublewords) may be aligned on arbitrary byte boundaries, although performance improvements may be expected with aligned data. The first 2^{20} bytes of the physical address space are called I/O space. These addresses are used to access device and channel control registers.

4.2. Instruction Format. Instructions are specified by a one byte opcode followed either by a list of operand specifiers or by a signed displacement. Each of these operand specifiers allows fully general addressing of the operand. The number of operand specifiers used by an instruction is determined solely by the nature of the operation it performs. No arbitrary restrictions are placed on the number of operands an instruction can address.

Figure 4-1: Instruction Format

4.3. Control Structure. The architecture provides a high level procedure based control structure. Procedures may be invoked by various types of calls, interrupts, traps, or as independent tasks. The procedure mechanism automatically provides for the passing of parameters and the maintenance of control linkages. Each procedure is provided with its own register set of up to 15 general registers for local use. The protection provided by the procedure structure allows privileged code to communicate with non-privileged code in a meaningful way.

4.4. Addressing. Addresses are calculated using 32-bit 2's complement arithmetic. Overflow in these calculations is ignored. Instructions can specify addresses using short displacements from registers or the instruction address. For address calculations, these short displacements are sign or zero extended as specified in the appropriate sections of the standard.

4.5. Data Types. The architecture provides support for integer, logical, and floating data types with multiple size representations for each. Integers are fully supported in 8, 16, and 32-bit lengths and some extended precision support is provided for 64-bit lengths. Integers are represented in 2's complement notation.

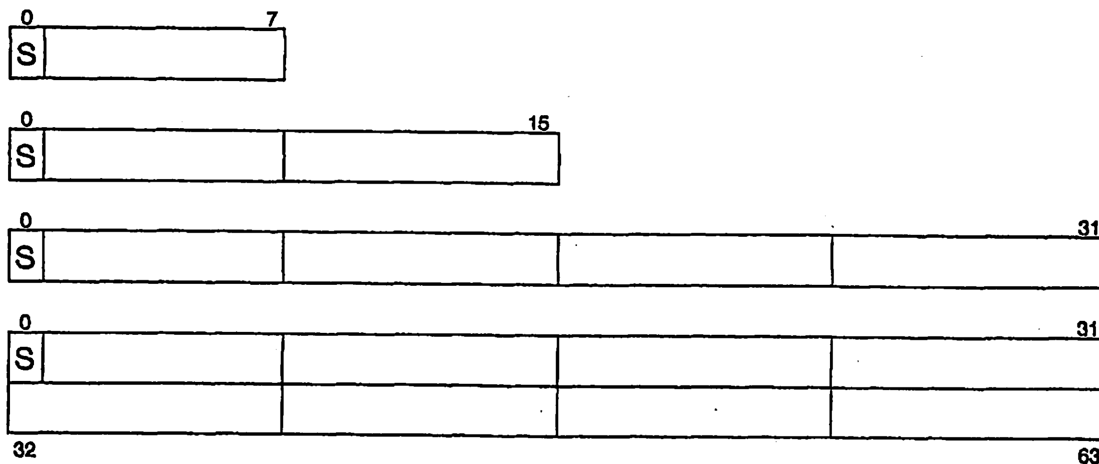


Figure 4-2: Integer Data Types

Logical quantities are supported in 1, 8, 16, 32, and 64-bit lengths. These data types may be used interchangeably as unsigned integers and bit patterns. The ability to access signed and unsigned bit fields is also provided.

Floating point numbers are provided in single (32-bit) and double (64-bit) length representations.

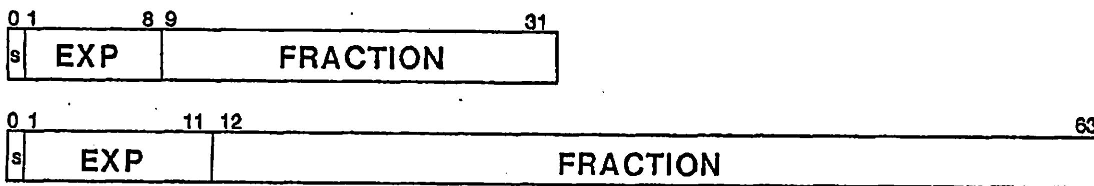


Figure 4-3: Floating Point Data Types

Conversion between integer and floating point representations is provided. Conversion between different lengths is provided implicitly; no conversion instructions are necessary.

5. Operand Addressing Modes

Operands are identified by the instructions through the decoding of multi-byte strings called **operand specifiers**. Operand specifiers contain the information needed to determine the location and the size of the operands to be accessed.

The location of each operand is fully general and may be:

- a literal constant in the code stream
- a register
- a memory location

The operand specifier selects the size of the operand from the supported primitive data types:

- 8-bit integer/logical
- 16-bit integer/logical
- 32-bit integer/logical
- 64-bit integer/logical
- 32-bit floating point
- 64-bit floating point

The overall structure of the Nebula architecture places a design restriction on the operand specifiers. The location (address) of all operands must be computable in the absence of any context information provided by the opcode. This permits operands to be "pre-evaluated" in the absence of such information as required by the procedure interface. Addressing modes are also required to be free of side effects. This eliminates any order dependencies in operand evaluation.

An overview of the addressing modes is shown in figure 5.

5.1. Maximum Accessible Register. Several of the addressing modes specify a four bit register field. This field is checked against the *Maxreg* field of the Processor Status Word to insure that the specified register is available in the current context. If the specified register number is greater than Maxreg, an **Illegal.Register** exception shall be initiated.

5.2. Compound Modes. Three addressing modes, General Parameter, Unscaled Index, and Scaled Index, use other operand specifiers as a "subpart" of their complete operand specification. These are referred to as **compound** modes. The subpart operand specifiers (that form a part of the full compound mode specifier) shall be restricted to non-compound modes. For example, as part of its specification the unscaled index mode calls for two other operands to be specified and used as part of the calculation of the unscaled indexed operand. These two subpart operands cannot be unscaled index or either of the other two compound operands. If a compound type operand is encountered when a subpart operand is being decoded, an **Illegal.Mode** exception shall be initiated.

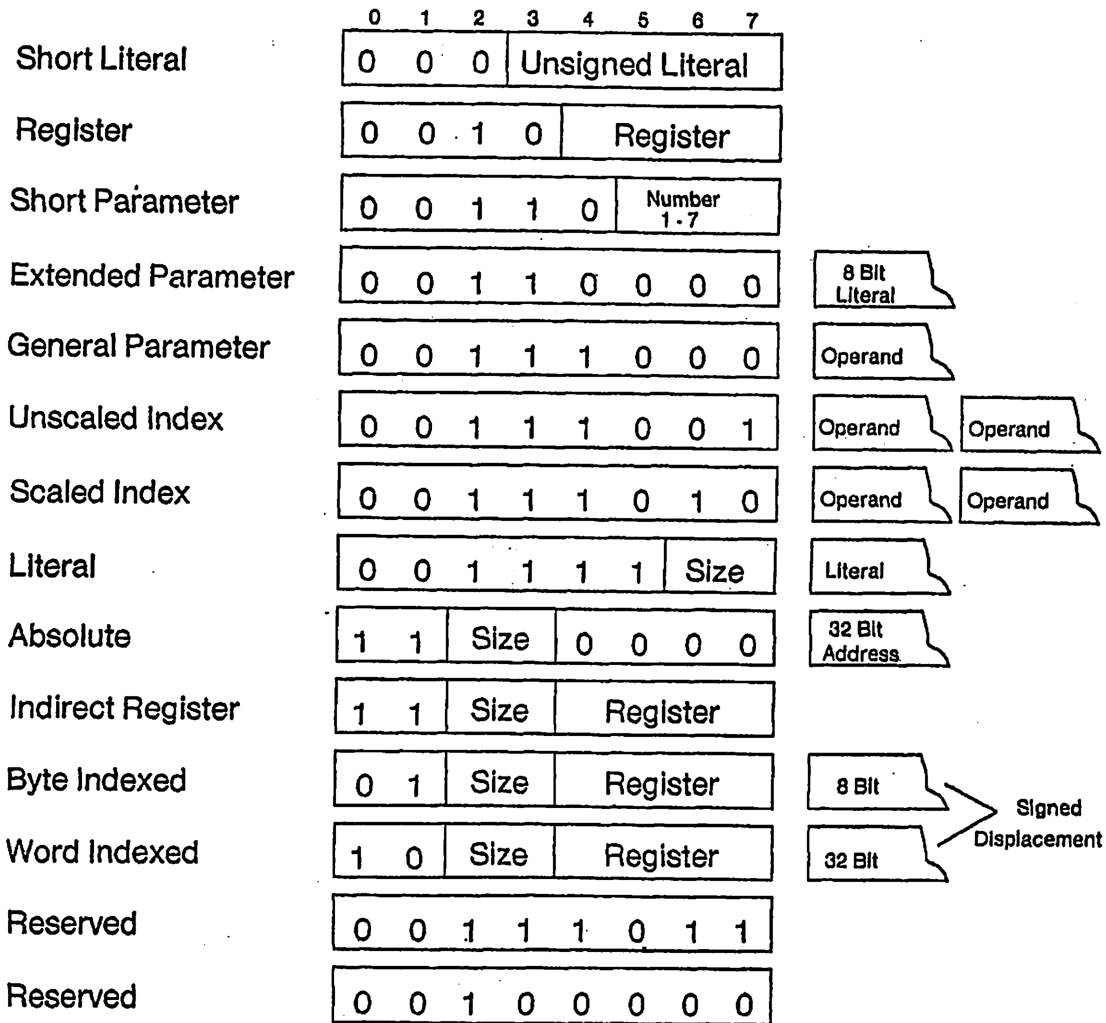
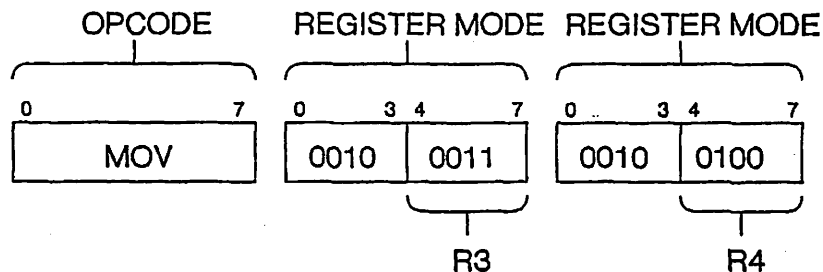


Figure 5-1: Operand Specifiers

5.3. Address Operands. Addressing modes that have their operands in memory may be used as **address operands.** When an instruction calls for an address operand, the operand specifier is evaluated until the address of the operand is obtained. That *address* is returned to the instruction rather than the value of the operand. If an operand specifier whose operand is NOT in memory (for example, register mode) is specified as an address operand, an **Illegal.Address** exception shall be initiated. The size field of an address type operand specifier is ignored unless otherwise specified by the individual instruction description.

5.4. Register Mode. In register mode the operand is located in the register (1 to 15) specified by the second half (bits 4:7) of the operand specifier byte. Register operands shall be word (32-bit) size and therefore require no size field specification. The Maxreg restriction applies to register mode.

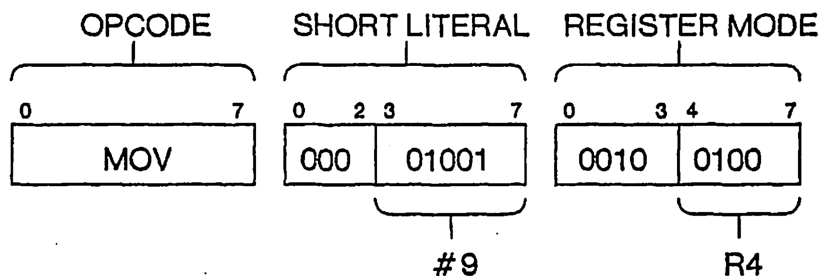
EXAMPLE: MOV %3, %4



In this example, the contents of register 3 are copied into register 4.

5.5. Short Literal Mode. In order to facilitate the use of short literals, a single byte specifier allowing unsigned literals of up to 5 bits is provided. Bits 3:7 of the specifier byte form the short literal. This short literal (0 - 31 decimal) is zero extended internally to the size necessary for further computation. An attempt to write into a short literal mode operand shall generate an **Illegal.Write** exception.

EXAMPLE: MOV #9, %4



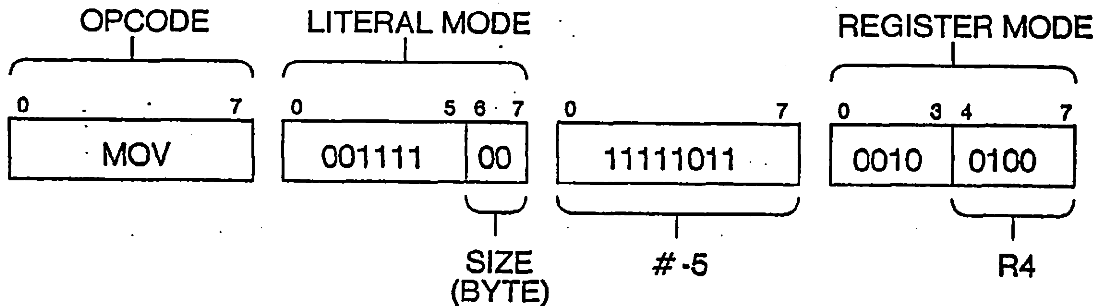
The literal 9 (encoded in the lower part of the short literal specifier) is zero extended and placed in register 4.

5.6. Literal Mode. When a literal that is negative or too large for short literal mode (larger than 31) must be specified, Literal Mode may be used. Literal mode addressing allows the specification of literals that are one, two, four, or eight bytes long. The literal follows the first specifier byte in the code stream. Bits 6:7 of the first specifier byte contain the encoded size of the literal. The encoding is:

- 00 - Byte (8 bits)
- 01 - Halfword (2 bytes, 16 bits)
- 10 - Word (4 bytes, 32 bits)
- 11 - Double word (8 bytes, 64 bits)

Once the literal is fetched, it may be extended internally, based on the requirements of the operation. An attempt to write into a literal mode operand shall generate an **Illegal.Write** exception.

EXAMPLE: MOV #-5, %4



The single byte literal -5 is sign extended and copied into register 4.

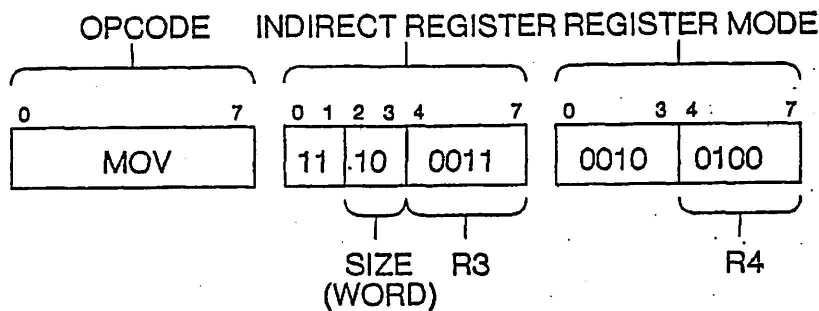
5.7. Memory Addressing Modes. The addressing modes described in this section all address memory. They may be used as address operands. Memory operands may be 1, 2, 4, or 8 bytes in length as specified by bits 2:3 of the first specifier byte.

Size field encoding:

- 00 - Byte (8 bits)
- 01 - Halfword (2 bytes, 16 bits)
- 10 - Word (4 bytes, 32 bits)
- 11 - Doubleword (8 bytes, 64 bits)

5.7.1. Indirect Register Mode. In indirect register mode, the second half of the operand specifier byte (bits 4:7) designates a register that contains the address of the operand. The Maxreg restriction applies to the register designation. Note that specification of 0 in the register field indicates absolute addressing described below.

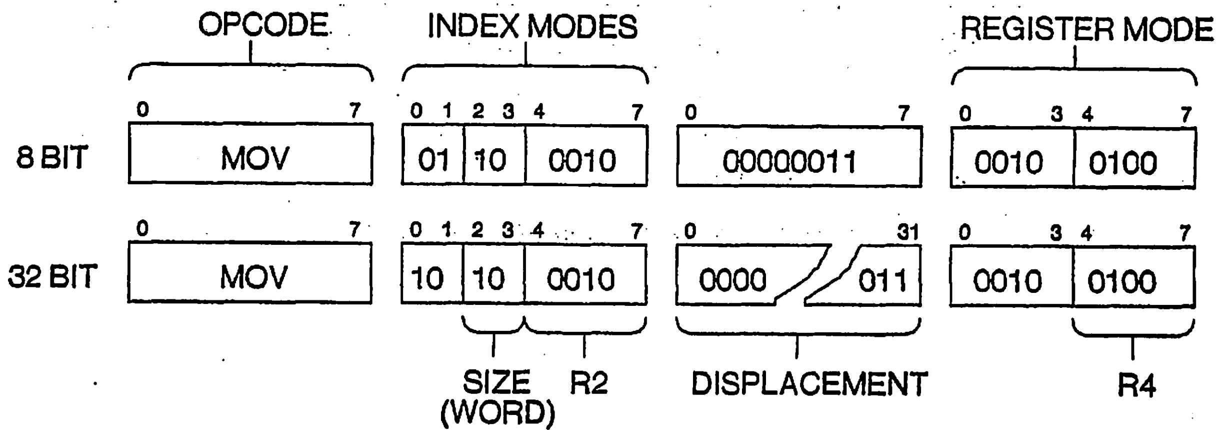
EXAMPLE: MOV @%3↑W, %4



The word operand located at the address specified in register 3 is copied into register 4.

5.7.2. Register Indexed Modes. There are two register indexed modes, Byte Indexed and Word Indexed. Each one specifies a register (bits 4:7) and a size (bits 2:3) in the first operand specifier byte followed by a signed displacement that is 8 or 32 bits long respectively. The signed displacement is added to the contents of the designated register to form the address of the operand. The Maxreg restriction applies to the designated register. The address of the displacement may be designated as the index in both register index modes by specifying a 0 in the register field of the first specifier byte. The specifier is then evaluated as though the index register were pointing to the displacement when calculating the memory address.

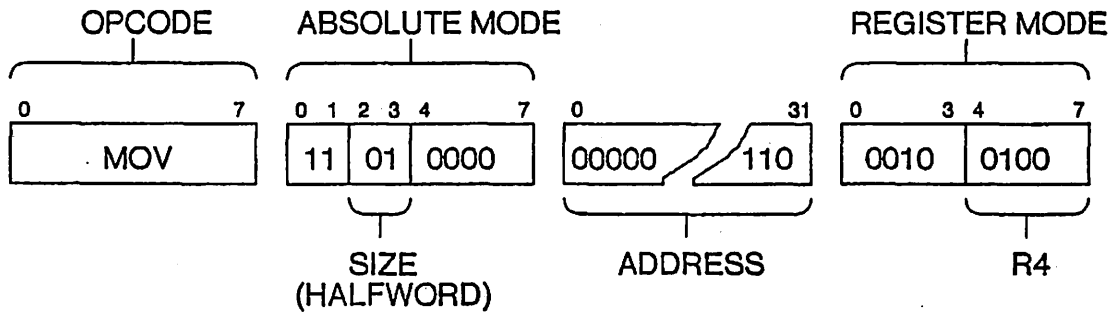
EXAMPLE: MOV 3(%2)W, %4



The displacement 3 is added to the contents of register 2 to form an address. The contents of the word at this address are copied into register 4. Note that this operation is the same for Byte Indexed and Word Indexed. The only difference is the size of the displacement.

5.7.3. Absolute Mode. In absolute mode, the first specifier byte is followed by the 32-bit address of the operand.

EXAMPLE: MOV @(6)H,%4



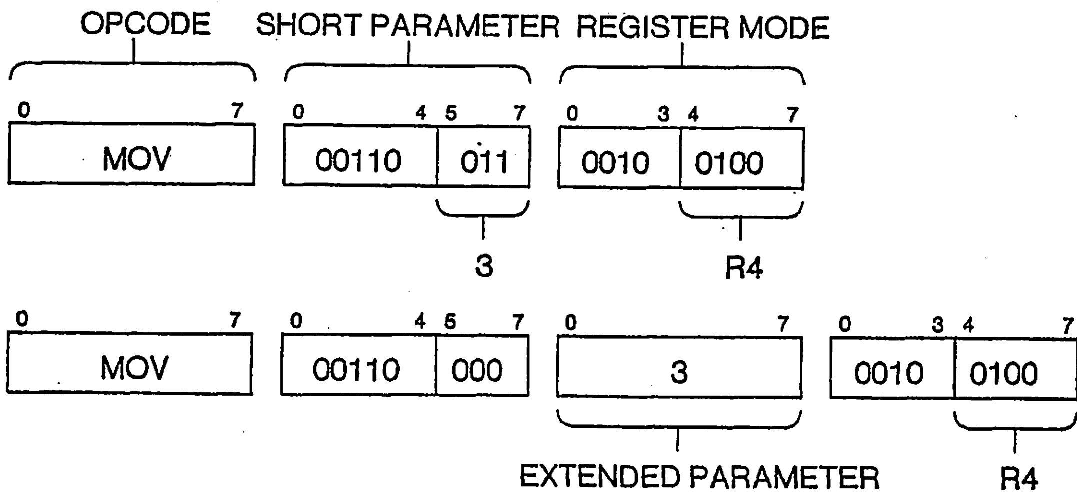
The halfword starting at location 6 in memory is copied into register 4.

5.8. Parameter Addressing Modes. Parameter addressing modes allow access to parameters defined by the caller of a procedure. The parameter modes specify an unsigned integer **parameter number**. If this number is greater than the *Number of Parameters* field in the processor status word an **Illegal.Parameter** exception is raised. Parameter numbers are bound to the actual parameters by the procedure call. Parameter modes may be used as address operands if the parameter referenced is memory type. See section 8.4 for more information on parameters, their generation and use.

5.8.1. Short Parameter Mode. Bits 5:7 of the specifier byte designate a parameter number in the range 1 to 7.

5.8.2. Extended Parameter Mode. Specifier 00110000_2 indicates Extended parameter mode. The byte following this initial specifier contains an unsigned parameter number in the range 0 to 255.

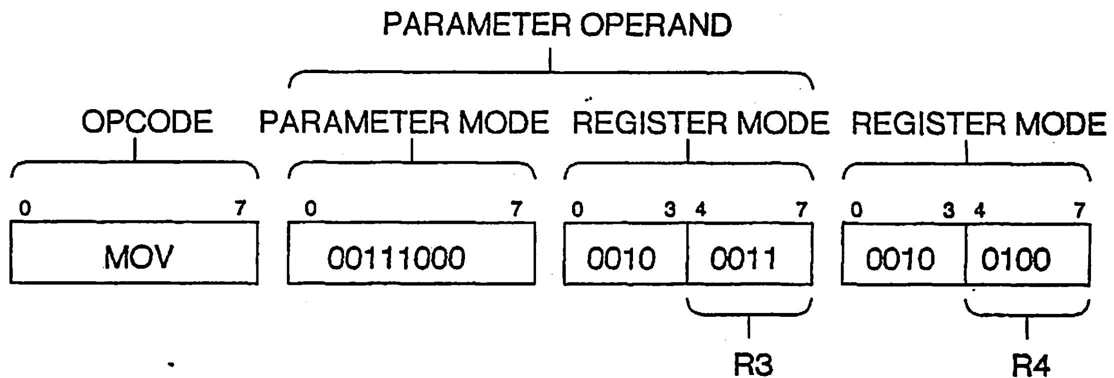
EXAMPLE: MOV ?3, %4



The contents of parameter 3 are copied into register 4.

5.8.3. General Parameter Mode. General Parameter Mode is one of the *compound* modes described above. The first byte of the operand specifier is followed by another complete operand specifier. This *second* operand specifier is evaluated as an unsigned integer parameter number.

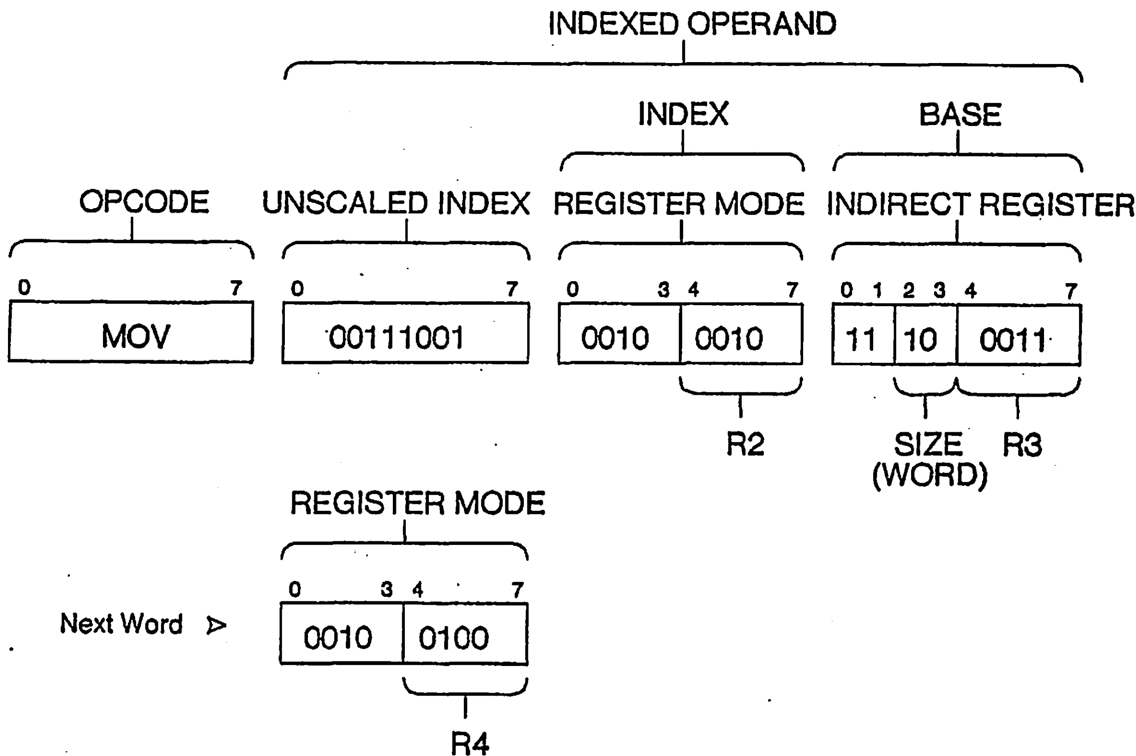
EXAMPLE: MOV ?(%3), %4



The parameter whose number is in register 3 is accessed and its value is copied into register 4.

5.9. Unscaled Index Mode. Unscaled Index is another of the *compound* modes. The first specifier byte is followed by *two* complete operand specifiers, the first is the **index** and the second is the **base.** The base operand specifier is an *address* type operand, and must evaluate to a memory address as described in section 5.3 above. In order to access the Unscaled Index Operand, the index specifier is evaluated as a signed integer operand and its **value** is added to the **address** obtained from the base specifier. This forms the full address of the operand. If the index specifier's size is 64 bits, an **Operand.Size** exception is raised. The size of the Unscaled Index Operand is determined by the size field in the base (second) specifier. Unscaled index mode may be used as an address operand.

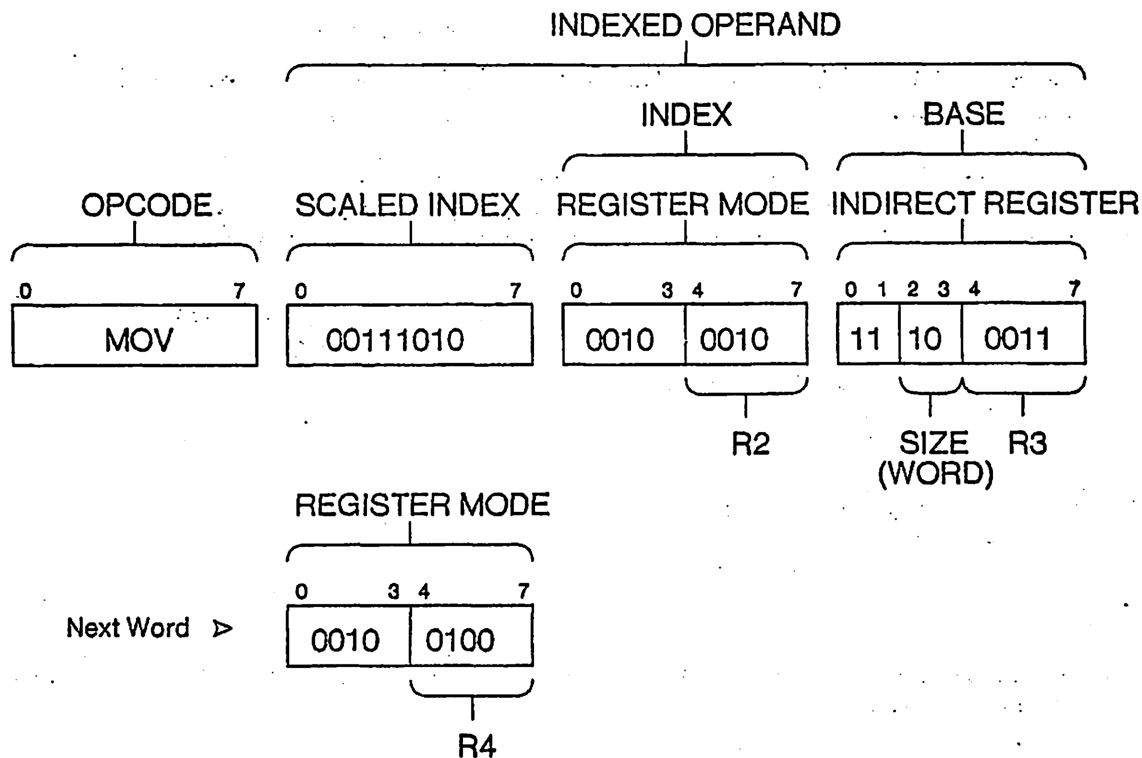
EXAMPLE: MOV @%3(%2)W, %4



The index in register 2 is added to the base address in register 3. The value of the word at this location is copied into register 4.

5.10. Scaled Index Mode. Scaled Index is the third *compound* mode. It is very similar to unscaled index described above. The first specifier byte is followed by two complete operand specifiers, the **index** and the **base**. The base operand specifier is an *address* type operand, and must evaluate to a memory address as described in section 5.3 above. The address of the Scaled index operand is found by evaluating the index specifier as a signed integer operand, scaling it, and adding it to the address obtained from the base specifier. The scaling is based on the size field in the base specifier. The scale factor is 1, 2, 4, or 8 based on the number of bytes in the operand specified by the base specifier size field. Scaling is simply multiplying the index value by the scale factor (since all of the scale factors are powers of 2 the multiplication can be accomplished by shifting). If the index overflows during scaling, the low order 32 bits are added to the base to form the operand address. If the index specifier's size is 64 bits, an **Operand.Size** exception is raised. Scaled index mode may be used as an address operand.

EXAMPLE: MOV @%3[%2]W, %4



The index value in register 2 is scaled by 4 (size = word) and added to the base address found in register 3. The value of the word found at the resulting address is copied into register 4.

5.11. Reserved Specifiers. Specifiers 00111011_2 and 00100000_2 are reserved for future definition. If they are encountered during operand specifier evaluation, an **Illegal.Mode** exception shall be initiated.

5.12. Undefined Operand Sizes. The individual instruction descriptions indicate the defined sizes for each operand. If, during execution, an instruction encounters an operand whose size is not defined in the instruction description, an **Operand.Size** exception shall be initiated.

6. Processor Status Word

Control information related to the current execution context is maintained in the 32-bit Processor Status Word. Bits 0:12 contain information global to the currently executing task. Bits 13:31 contain information related to the current procedure's context. The procedure call and return facilities alter only bits 13:31, while initiation or completion of interrupts or tasks alters the entire PSW.

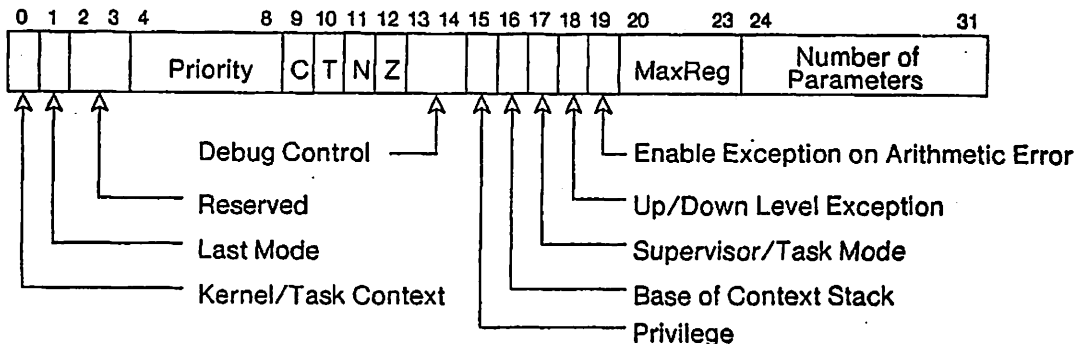


Figure 6-1: Processor Status Word

The form of the PSW is shown in figure 6-1. The fields of the PSW are defined below.

6.1. Kernel/Task Mode. Bit 0 specifies whether the processor is in kernel or task mode. If bit 0 is clear, the processor is in kernel mode. The Kernel Context Stack shall be used for the context of calls. If bit 0 of the PSW is set, the processor is in task mode. The task context stack shall be used for procedure calls.

6.2. Last Mode. Bit 1 shall be set on interrupt and trap operations to reflect the state of the Kernel/Task mode bit prior to the interrupt or trap. This information is used by the RET instruction to determine which context stack is to be reactivated.

6.3. Reserved Bits. Bits 2:3 of the PSW are reserved for implementation dependent functions. A PSW saved as part of an interrupt or trap context change may have an implementation dependent bit pattern in this field. This PSW should be restored with this pattern unaltered to restart the interrupted context. The results of altering this bit pattern in such a case are unpredictable.

6.4. Priority. Bits 4:8 of the PSW shall define the current priority of the processor. Hardware or software interrupt requests at an equal or lower priority shall be masked.

6.5. Carry Condition Code. Bit 9 of the PSW is the carry bit. The C bit shall be set by integer add and subtract instructions to reflect the carry from the most significant bit of the internal result.

6.6. Truncate Condition Code. Bit 10 of the PSW is the truncation flag. The T bit shall be set if the integer result of an operation cannot be correctly represented in the specified result operand. The setting of truncate indicates that significant bits (bits not equal to the stored sign bit) were lost. The truncate bit shall also be set on the occurrence of certain floating point errors, when the corresponding floating point exception is disabled. Refer to section 30.4.

6.7. Negative Condition Code. Bit 11 of the PSW is the negative condition code. The N bit shall be set as specified in the instruction descriptions. Generally, setting of the N bit implies a negative or less than condition.

6.8. Zero Condition Code. Bit 12 of the PSW is the zero condition code. The Z bit shall be set as specified in the instruction descriptions. Generally, setting of the Z bit implies a zero or equal condition.

6.9. Debugging Control. Bits 13:14 of the PSW control the instruction and procedure level debugging facilities. Refer to section 10 for a description of their function.

6.10. Privilege. Bit 15 of the PSW specifies whether the current context is privileged. If this bit is set, privileged instructions may be executed. If this bit is clear, an attempt to execute a privileged instruction shall cause a trap.

6.11. Base of Context. Bit 16 of the PSW indicates the execution context base. If this bit is set, the currently executing context has no caller. Execution of a RET instruction with the Base bit set shall cause an interrupt return operation as defined in the RET instruction description. Bit 16 shall be set in the PSW of the new context produced by an interrupt, trap, TINIT instruction, or PINIT instruction.

6.12. Supervisor Mode. Bit 17 shall indicate whether the current context is in Supervisor mode. If this bit is set accesses through the supervisor map may be made.

6.13. Up/Down Level Exception Propagation (UDLE). Bit 18 shall indicate the direction of exception propagation. If bit 18 is clear, the exception shall be directed to the current exception handler or to the caller. If bit 18 is set, the exception shall be directed to the Supervisor Exception Handler. Refer to section 9.4 for a detailed explanation of exception propagation.

6.14. Enable Arithmetic Error. EAE, bit 19 of the PSW, shall control the generation of exceptions related to arithmetic errors. If EAE is set and the T bit (bit 10 of the PSW) is set at the end of an instruction, a **Truncation** exception shall occur. If EAE is clear, truncation exceptions shall be suppressed. EAE is also used as a mask for floating point errors. Refer to section 30.4.

6.15. Maxreg Field. Bits 20:23 of the PSW define the unsigned maximum accessible register number in the current context. An attempt to access a register number greater than that specified in Maxreg shall cause an **Illegal.Register** exception.

6.16. Number of Parameters. Bits 24:31 define the unsigned number of parameters accessible in the current context. An attempt to access a parameter number greater than that specified in bits 24:31 shall cause an **Illegal.Parameter** exception.

7. Auxiliary Status Register

The Auxiliary Status Register is a 32-bit register located in the I/O space. It contains various processor control and status bits that need be accessed only infrequently. It is not saved and restored on context changes. The format of the ASR is shown in figure 7-1. The fields of the ASR are described below.

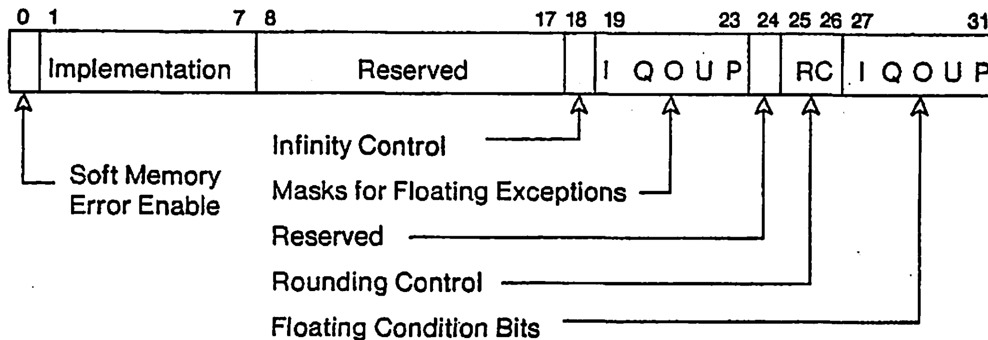


Figure 7-1: Auxiliary Status Register

7.1. Soft Memory Error Enable. Bit 0 shall control the generation of traps on soft memory error conditions. If this bit is clear, no traps shall be generated. Refer to section 11.6 for a description of the memory error handling facility.

7.2. Reserved Bits. Bits 8:17 and 24 are reserved for future definition. Bits 1:7 are reserved for implementation dependent functions. The ASR may have an implementation dependent bit pattern in this field.

7.3. Floating Point Mode Control. Bits 18:23 and 25:31 are dedicated to control and status information for the various modes supported by the floating point system. Refer to section 30.4 for a detailed discussion of each.

7.3.1. Infinity Control. Bit 18 shall control the ordering of the special floating point infinity symbols. If set, negative infinity shall be considered less than positive infinity. If clear, negative and positive infinity shall be considered equal.

7.3.2. Exception Event and Mask Bits. For each of five different exception conditions supported by the floating point system there shall be an event and a mask bit in the ASR. The mask bit shall control generation of an exception on occurrence of the corresponding condition. Refer to section 30.4 for a description of floating exception processing. The assigned bits and exceptions for each condition are listed below.

Condition	Mask Bit	Event Bit	Exception
Invalid Operation	19	27	Invalid.Operation
Divide By 0	20	28	Divide.By.Zero
Overflow	21	29	Floating.Overflow
Underflow	22	30	Floating.Underflow
Inexact Result	23	31	Floating.Inexact

7.3.3. Rounding Control. Bits 25:26 select the rounding mode used by floating point operations. Refer to section 30.5 for a complete description.

8. Procedure Interface

The fundamental unit of execution in the Nebula architecture is the procedure. A procedure is a code sequence with an associated context. The context of a procedure includes:

PSW
: The processor status word contains information about the current procedure. Included is the number of registers accessible, the number of parameters, the method of dealing with exceptions, and certain privilege information. The context for each procedure includes the PSW that describes its capabilities.

Registers
: Each procedure is provided with its own set of registers that are separate from those of any other procedure.

Parameters
: Procedures may have parameters. If a procedure has parameters, then the list of parameters for the procedure is included in the procedure's context.

Exception Handler
: Each procedure may define a sequence of code to be executed when an error or abnormal condition is detected. This error code is called an exception handler. The location and state of a procedure's exception handler is recorded in its context.

This collection of information, called the **Procedure Context**, defines the current state of execution of a procedure.

Procedures may call other procedures. This implies that the current procedure context should be preserved and a new procedure context created for the called procedure. Completion of the called procedure should cause its context to be eliminated and the calling procedure's context to be restarted. Thus the call/return behavior of procedures produces a stacking of procedure contexts. In the Nebula architecture, the stacked procedure contexts are maintained on a **Context Stack.** Figure 8-1 shows the general form of the context stack.

In each task there is a procedure (usually called the "main program") that has the unique property that it has no caller. The context for this procedure was created by the initiation of execution of the task. A return from this "main program" implies termination of the task. The context of the "main program" forms the BASE of the stack of procedure contexts for this task. This unique procedure context is identified by the fact that its PSW has bit 16, the BASE bit, set. The collection of procedure contexts (the currently active routine plus its caller plus the caller's caller... down to the "main program") is called an **Execution Context.**

8.1. The Context Stacks. There are two active context stacks in the Nebula architecture: the Kernel Context Stack and the Task Context Stack. These stacks are accessed by using one of two context pointers. The context stack used by the executing code is determined by the current PSW. Normal call and return operations do not change the context stack in use. Interrupts, traps, and task completions may cause a context stack switch.

8.1.1. Context Pointers. There shall be two 32-bit registers, called the Kernel Context Pointer and the Task Context Pointer, that contain the virtual addresses of the currently active procedure contexts on the Kernel and Task context stacks, respectively. The Context Pointer currently in use shall be selected by bit 0 of the current PSW. If this bit is clear, the Kernel Context pointer shall be used, if this bit is set the Task Context pointer shall be used. These registers shall be accessible in the I/O space and by the privileged LTASK and STASK instructions. The operation of these instructions nsure that the address contained in the context pointer is aligned on a 32-bit word boundary.

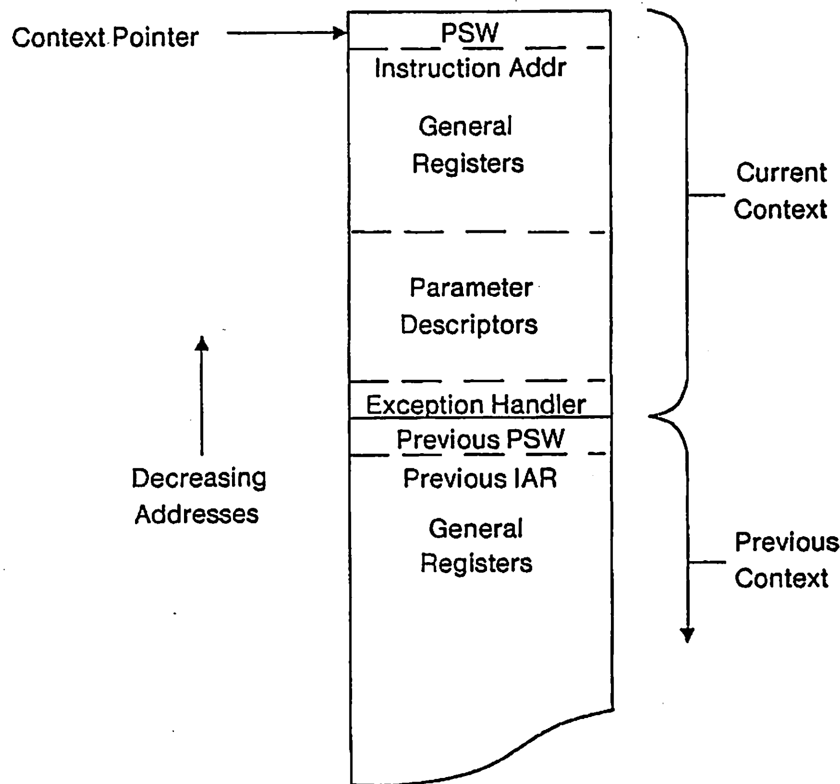


Figure 8-1: Stacked Procedure Contexts

8.1.2. Structure of Context Stacks. The context stack selected by the current context pointer shall be used by the executing code. The context of each executing procedure shall be stored in the following order (increasing addresses):

1. The PSW for this procedure.
2. The registers for this procedure, starting with the instruction address register, up to the maximum accessible register as defined in this procedure's PSW.
3. The parameter descriptors for this procedure (defined in section 8.4.4).
4. The state of the exception handler for this procedure (defined in section 9).

The context stack shall expand downward. A call by the currently executing procedure shall cause the context pointer to be decremented by the size of the context of the called procedure. The context of the called procedure shall then be placed in the area defined by the decremented context pointer. Note that contexts are not position independent in the virtual address space.

8.1.3. Cacheing of the Context Stack. The context of a procedure contains areas reserved for a variety of frequently accessed quantities such as the instruction address register, the PSW, the general registers and so on. In many implementations it will be desirable to maintain such information in fast registers. This is particularly true for the currently executing procedure's instruction address register and general registers. The Nebula architecture does *not* define the properties of any such cacheing mechanism. The representation of the context stored on the Kernel and Task context stacks is IMPLEMENTATION DEPENDENT. The value of such memory locations is undefined. The effect of storing into such memory locations is unpredictable. The memory management system provides a mechanism for protecting against such invalid software actions.

8.1.4. Changing of Context Stack Pointers. The Nebula architecture provides two **active context stacks**, corresponding to the Kernel Context Pointer and Task Context Pointer. Since each task in the system has associated with it a unique execution context, a mechanism for changing the active Task Context Pointer is provided. The LTASK and STASK instructions provide this facility. Additionally, these instructions shall force consistency between the context in memory and any cached state. Subsequent to the execution of STASK all context information necessary to restart the task context shall be stored in addressable memory in the context stack. For a complete description of their functions, refer to the individual instruction descriptions.

8.1.5. Alignment of Context Pointers. The knowledge of which item in the current procedure context is pointed to by the context pointer is implementation dependent. The address in the context pointer shall be greater than or equal to the smallest address occupied by the current context. When a new procedure context is created, the context pointer (prior to being decremented) shall be greater than the address of any byte of the newly created context. These restrictions imply that a context stack may be initialized by setting the context pointer to the greatest word address in the context segment plus 4.

8.2. Procedure Descriptor. The characteristics of a procedure are specified by a procedure descriptor located at the entry point(s) of the procedure. This 16-bit descriptor will be aligned on a word boundary. The format of the procedure descriptor is shown in figure 8-2. Upon invocation of the procedure, the information contained in bits 2:15 is recorded in bits 18:31 of the PSW. Execution of the procedure shall begin at the location immediately following the procedure descriptor located at the addressed entry point. The function of each field of the procedure descriptor is described below.

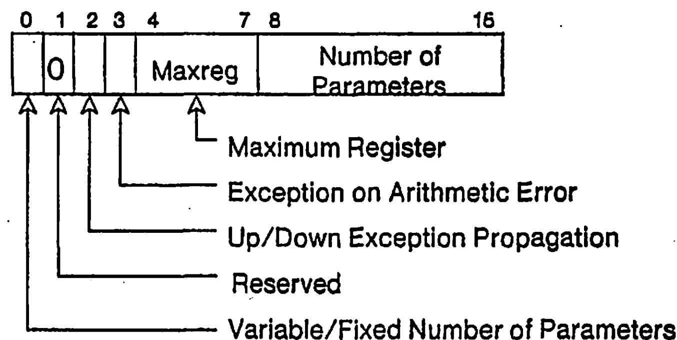


Figure 8-2: Procedure Descriptor

8.2.1. Fixed/Variable Number of Parameters. Bit 0 of the procedure descriptor shall specify whether the number of parameters for this procedure is fixed or variable. If bit 0 is clear, bits 8:15 will contain the number of parameters. If bit 0 is set, bits 8:15 are reserved and the first byte of the parameter list shall be evaluated as the unsigned integer number of parameters. In either event, the number of parameters for the procedure shall be placed in bits 24:31 of the PSW.

Some procedure entries, such as interrupt and trap handlers, have their number of parameters fixed by the hardware. In such procedure descriptors, bit 0 is reserved and bits 8:15 are ignored.

8.2.2. Reserved Bit. Bit 1 of the procedure descriptor shall be reserved for future definition.

8.2.3. Exception Propagation Control. Bit 2 of the procedure descriptor shall be copied to bit 18 of the PSW. This bit defines the direction of exception propagation. See section 9.4.

8.2.4. Arithmetic Error Control. Bit 3 of the procedure descriptor shall be copied to bit 19 of the PSW. This bit defines the action to be taken on arithmetic errors. See section 6.14.

8.2.5. Available Registers. Bits 4:7 shall determine the maximum numbered register accessible to this procedure. This value shall be copied to bits 20:23 of the PSW. This number of registers (and an instruction address register) shall be allocated for use by this procedure.

8.3. Procedure Invocation. In the Nebula architecture calls, supervisor calls (SVCs), unimplemented opcodes (OPEXs), supervisor handled exceptions, task initiations, interrupts, and traps are all handled as procedure invocations using a common mechanism. They differ primarily in the manner in which they determine the entry address of the procedure to be invoked and the parameters to be passed.

The procedure call mechanism is driven by the procedure descriptor located at the entry address of the procedure. Invocation of a procedure shall cause the following actions to occur:

- The current PSW is saved in the context stack and a new PSW is created.
- The register set of the invoked procedure is allocated.
- The exception handler for the procedure is initialized.
- The parameter descriptors for the procedure are initialized.

The current context pointer shall be updated to reflect the new procedure context added to the context stack. Each of these actions is detailed below.

8.3.1. Determination of a New PSW. The contents of the new PSW are determined by the type of invocation (call, trap, etc.) and by the procedure descriptor. Bits 18:31 of the PSW shall be determined by the procedure descriptor as described in section 8.2. Bit 16 (BASE) shall be set by interrupts, traps, the TINIT and PINIT instructions (task initiation), and by invocation of the supervisor exception handler with a **Task.Failure** exception. It shall be cleared otherwise. Bit 17 (Supervisor) shall be set equal to the most significant bit of the procedure entry address. Note that the call instructions will produce a trap if a non-supervisor program attempts to set this bit.Bits 2:3 are set/reset in an implementation dependent manner. Bits 0:1 and 4:14 of the PSW shall be set in the following manner:

CALL, CALLU	Unchanged
SVC	Bits 13:14 are cleared. Bits 0:1 and 4:12 are Unchanged

Supervisor Exception Handler
Bits 13:14 are cleared. Bits 0:1 and 4:12 are Unchanged

Unimplemented Opcode (OPEX)
Bits 13:14 are cleared. Bits 0:1 and 4:12 are Unchanged

Task Initiation — The TINIT and PINIT instructions specify the contents of bits 0:1 and 4:15 of the PSW as one of their operands. Bits 2:3 are implementation dependent.

Traps — Last mode (bit 1) shall be set to the previous value of bit 0. Bit 0 (Kernel/Task) shall be clear. This forces the trap to use the Kernel context stack. Bits 4:8 shall be set to 1F (Hex). Bits 9:14 (the condition codes and debug control) shall be clear. Bits 2:3 are implementation dependent.

Interrupts — The new PSW shall be determined as for traps except the priority (bits 4:8) shall be set to the priority of the interrupting device.

The privilege bit of the new PSW (bit 15) shall be set equal to bit 31 of the vector for trap and interrupt entries. For SVC, Supervisor Exception Handler, unimplemented opcode (OPEX) entries, the privilege bit of the new PSW shall be formed by OR'ing the privilege bit of the old PSW with Bit 31 of the vector. Task initiation specifies this bit explicitly. Call instructions leave the bit unchanged or clear it, depending on the opcode.

8.3.2. Register Set Allocation. The registers of a called procedure are separate from those of the caller. The number of registers available to the invoked procedure shall be determined by the procedure descriptor. Space shall be allocated on the context stack for each register starting with the instruction address register followed by the number of registers specified in the procedure descriptor. The contents of the newly created general registers are undefined with the exception of register 1. If this is not a task initiation, interrupt or trap invocation, and register 1 exists in both the caller and called procedure's registers, then the value of the caller's register 1 shall be copied to the called procedure's register 1. Otherwise, the contents of register 1 are also undefined. With this single exception, there is no inheritance of registers in the Nebula architecture. Note that this inheritance of register 1 applies to the supervisor exception handler. If register 1 is defined for the Supervisor Exception Handler and for the *procedure that invoked it*, the contents of the invoking procedure's register 1 are copied into register 1 of the Supervisor Exception Handler. If the Supervisor Exception Handler is invoked due to a **Task.Failure** exception, no invoking procedure is presumed and register 1 of the Supervisor Exception Handler is undefined.

8.3.3. Initialization of Exception Handler. Space in the procedure context shall be provided to contain the state of the exception handler for this procedure. The exception handler shall be set in the Disabled state (see section 9).

8.3.4. Initialization of Parameter List. The parameters for a procedure invoked by a CALL, CALLU, SVC, or unimplemented opcode (OPEX) are specified explicitly in a parameter list. The parameters of an interrupt, trap, or supervisor exception handler invocation are implicitly defined by the architecture. Task initiations have no parameters. An area of the procedure context shall be reserved for the parameter descriptors of the invoked procedure. There shall be one such descriptor for each parameter of the procedure. Parameter descriptors are defined functionally in section 8.4.

8.4. Parameter Lists. The passing of arguments or parameters to a called procedure is accomplished by specifying a parameter list. In calls, supervisor calls, and unimplemented opcode exceptions (OPEXs) the parameter list is explicitly specified while in interrupts, traps and the supervisor exception handler it is implied by the architecture. The specified parameters are accessed using the parameter addressing modes that determine the number of the parameter to be accessed

(1st, 2nd, 3rd, etc.). The representation of these addressing modes is described in section 5.8. The semantics of parameter access are defined below.

8.4.1. Parameter Specification. Explicit specification of parameters is accomplished by a parameter list that immediately follows the call opcode (for OPEX) or the index specifier (for SVC) or the entry address specifier (for CALL and CALLU). The parameter list shall consist of a sequence of operand specifiers or a byte literal followed by a sequence of operand specifiers. These operand specifiers may use any of the addressing modes described in section 5. The number of operand specifiers shall be determined in the following manner:

1. The calling instruction (CALL, CALLU, SVC, or OPEX) will specify the address of the procedure to be called. At this address there will be a procedure descriptor, as described in section 8.2.

2. If bit 0 of the procedure descriptor is clear, bits 8:15 of the procedure descriptor shall specify the number of parameters. If bit 0 is set, the first byte of the parameter list shall be interprefed as the unsigned integer number of parameters. The remaining operands of the reduced parameter list shall be interpreted as the actual parameter list.

Each operand specifier in the parameter list specifies the location and size of one operand. The location of each operand shall be determined using the state of the CALLER at the time of the call. Any registers specified in the parameter list shall refer to the caller's registers; any addresses in the parameter list shall be computed using the caller's registers as they exist when the call is executed.

8.4.2. Parameter Access. The parameters specified in the parameter list are conceptually numbered in increasing order, starting at 1. The parameter addressing modes specify a parameter number. The appearance of a parameter addressing mode as an operand shall be interpreted as an access to the correspondingly numbered parameter of the current context. Parameter 0 shall be interpreted as the number of parameters accessible in the current context; this information is recorded in bits 24:31 of the PSW. Parameter 0 cannot be written; an attempt to do so shall cause a **Illegal.Write** exception. Bits 24:31 of the PSW specify the maximum parameter number in the current context. An attempt to access a parameter number greater than this shall cause an **Illegal.Parameter** exception.

8.4.3. Example of Parameter Linkage. The context stack generated by this example (assuming 2 registers for both the caller and callee) is shown in figure 8-3. Assume X is a procedure with a fixed number of parameters =3 and that both X and its caller have access to a register 2 (%2). The following call:

```
Call    X, #5,A↑W, %2
```

would set up the following correspondence between parameter numbers (written as ?n) and parameters in the parameter list:

```
?0        =     #3      ;Number of parameters
?1        =     #5      ;A literal 5
?2        =     A↑W     ;A 32-bit word in memory named A
?3        =     %2      ;The caller's register 2
?4,?5,...       illegal
```

The use of parameter addressing modes in this context would have the following effects:

```
MOV    ?1,%2 <=>     MOV    #5,%2
MOV    ?2,%2 <=>     MOV    A↑W,%2
MOV    ?3,%2 <=>     MOV    %2*,%2
```

NOTE: The last example moves the contents of the CALLER's register 2 to the called routine's register 2.

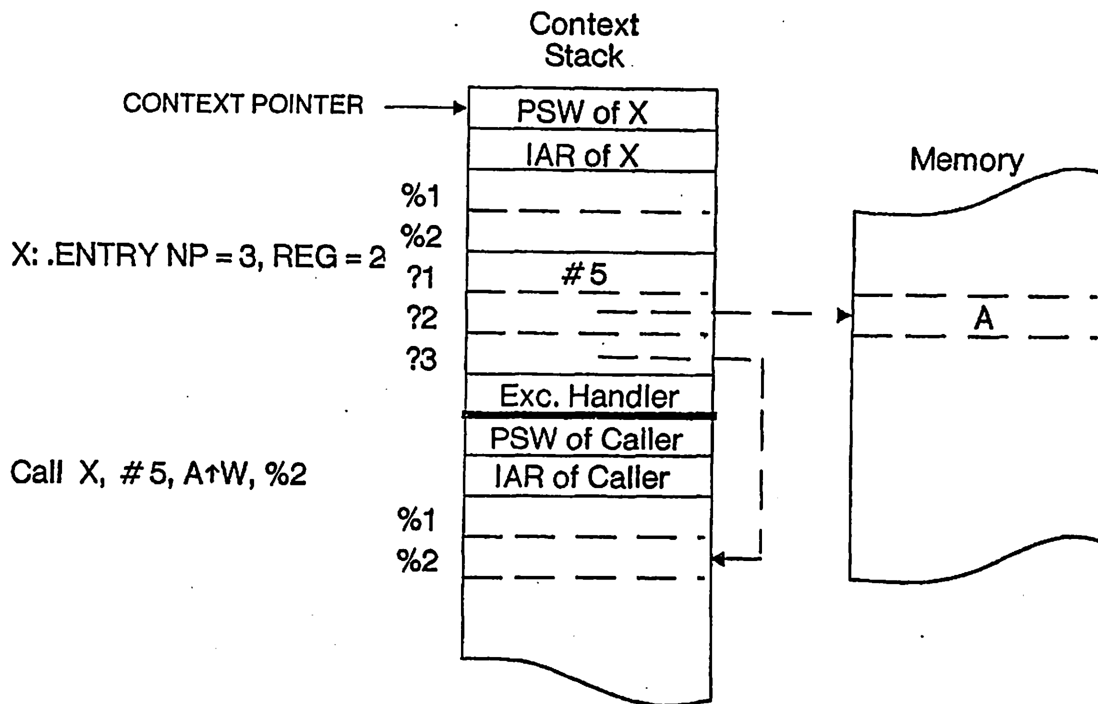


Figure 8-3: Example of a Call Context

8.4.4. Mechanism of Parameter Addressing. The parameter access mechanism can be viewed as a splitting of the operand access mechanism used by all instructions. Figure 8-4 is a conceptual picture of the Nebula microengine. Operand specifiers are reduced to location and size information by operand pre-evaluation. This primitive information is used for operand access during operand evaluation.

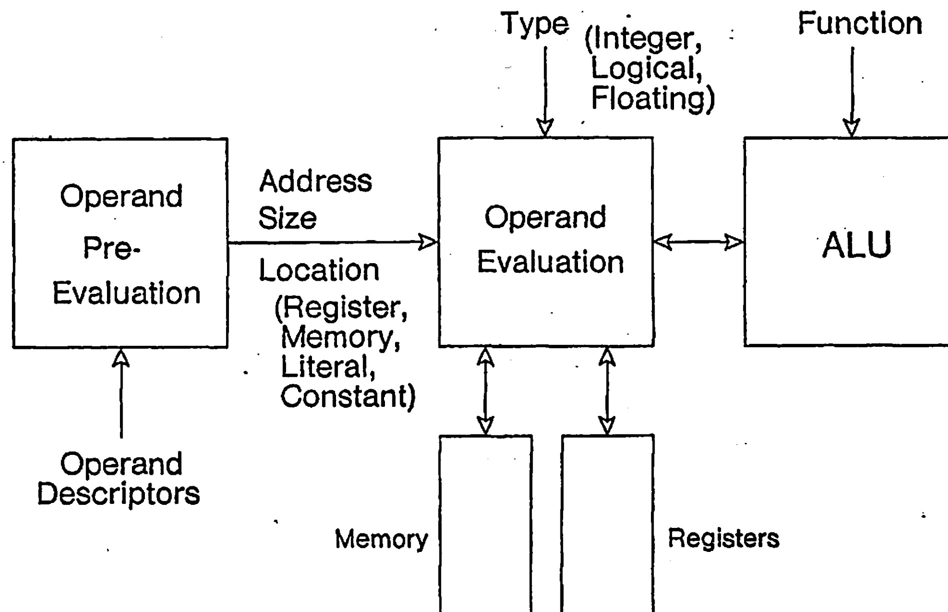


Figure 8-4: Normal Operand Processing

The operands of parameter lists, however, are not fully evaluated in this manner. Instead, the location and size information from pre-evaluation is diverted to the appropriate parameter descriptor in the context stack as shown in figure 8-5. This reduces the parameter list to a linear descriptor array.

Access to parameters is accomplished by parameter addressing modes. These operand specifiers evaluate to an unsigned integer parameter number. This index is used to recover the position and size information corresponding to the specified parameter. The mechanism is diagrammed in figure 8-6.

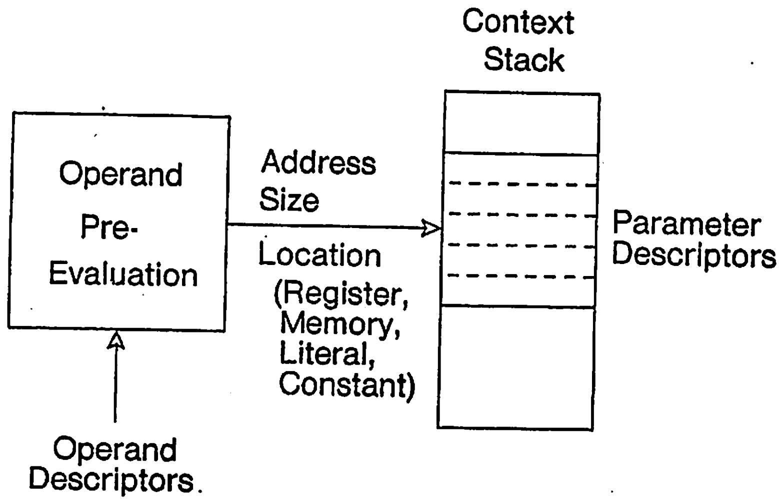


Figure 8-5: Parameter List Operand Processing

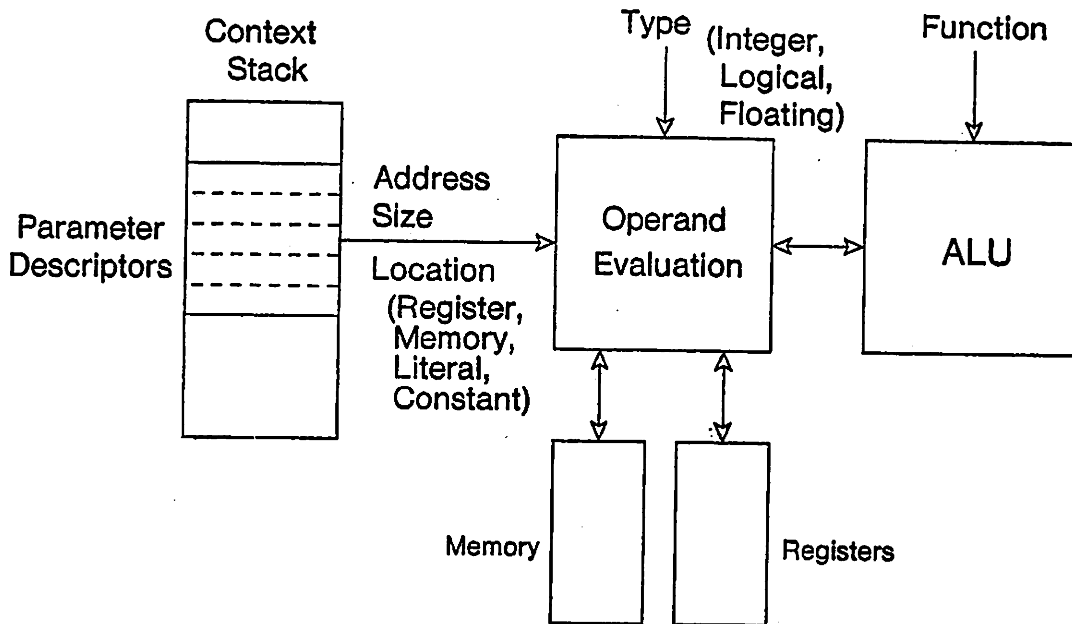


Figure 8-6: Parameter Access

The size and format of the parameter descriptors is implementation dependent. The encoding of these descriptors is constrained only by the functional requirements placed on the parameter passing and access facilities. The following is an example of an encoding sufficient to meet these requirements. In this example the parameter descriptors specify one of four possible locations for the parameter:

Constant — The parameter is a value contained in the parameter descriptor. Such a parameter is read-only. An attempt to write a constant parameter will cause an **Illegal.Write** exception.

Literal — The parameter is a literal contained in the instruction stream. The address and size of the literal are encoded in this descriptor. This parameter is read-only. An attempt to write this parameter will cause an **Illegal.Write** exception.

Register — The parameter is a register in the context stack. The size of the parameter is 32 bits. A pointer to the register in the context stack is encoded in this descriptor.

Memory — The parameter is a memory location. The size and address of this location are encoded in this descriptor.

8.5. Vectored Calls: SVC and OPEX. The supervisor call (SVC) and unimplemented opcode (OPcode EXception or OPEX) facilities utilize a vectoring mechanism controlled by the supervisor. Vectoring shall be controlled by a pair of processor registers for SVCs and a pair for OPEXs. The first register specifies the limits for the vectoring index. The second register specifies the word address of the vector table. The two low order bits (bits 30:31) are ignored. The format of these registers is shown in figure 8-7.

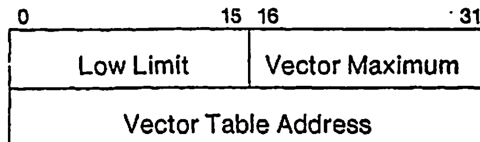


Figure 8-7: SVC and OPEX Vector Registers

The vectored instruction shall select the appropriate register pair (SVC or OPEX) and specify an unsigned index. This index shall be compared with the unsigned low limit specified in the first register of the pair. If the index is less than the low limit, or if (index - low limit) is greater than vector maximum (unsigned), the 32-bit word located at the word address before the virtual address in the second register (Vector Table Address - 4) shall be used as the vector. If the index is within the specified range, the quantity Vector Table Address + (index - low limit)*4 shall be used as the virtual address of the vector. The contents of the 32-bit vector shall be used as the entry address of the procedure. Access to this vector and to the procedure descriptor shall be allowed regardless of the state of the privilege or supervisor bits of the PSW. The vectoring operation is shown in figure 8-8.

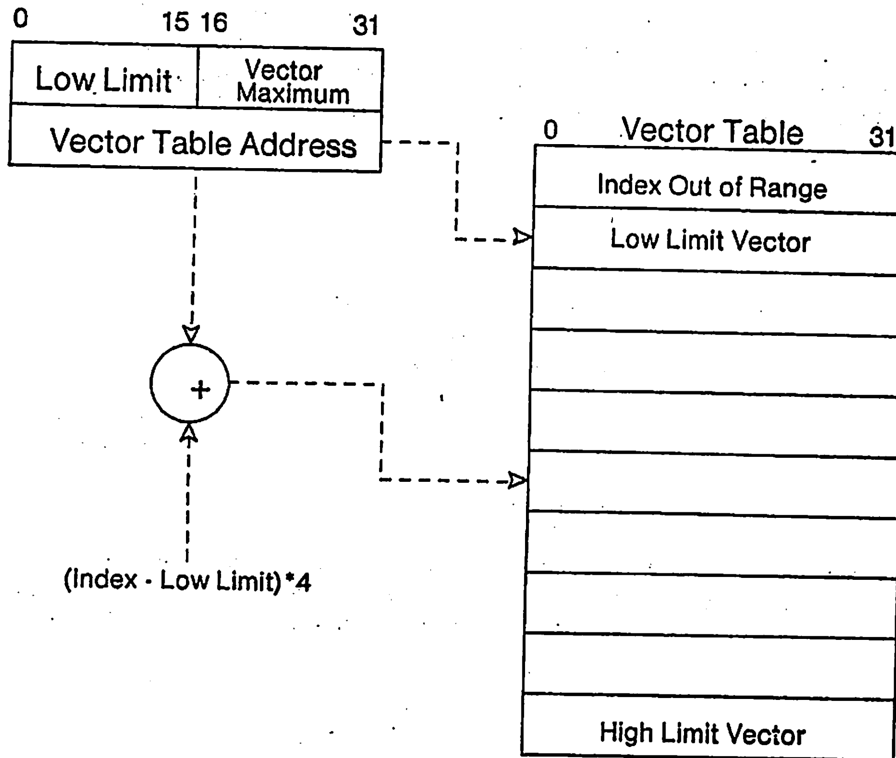


Figure 8-8: Vectored Call Address Calculation

8.6. Vector Format. All vectored operations in the Nebula architecture (interrupts, traps, SVCs, OPEXs and Supervisor Exception Handler entries) use a common vector format. Vectors are 32 bit words with the format shown in figure 8-9. Bits 0:29 of the vector specify bits 0:29 of the entry address. Bits 30:31 of the entry address are assumed to be zero. Bit 0 is copied into bit 17 of the new PSW. Bit 31 of the vector is used in determining the privilege of the called procedure. For interrupts and traps bit 31 of the vector is copied to bit 15 of the new PSW (privilege). For SVCs, OPEXs and supervisor exception handler calls bit 31 of the vector is OR'ed with bit 15 of the caller's PSW to form bit 15 for the new PSW.

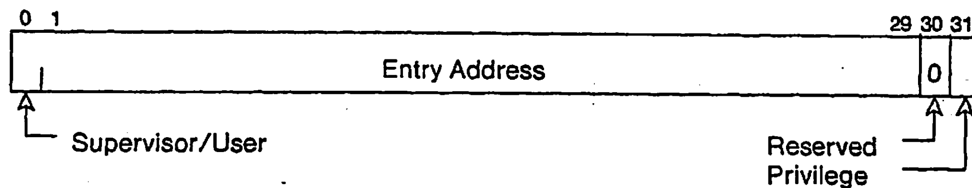


Figure 8-9: Vector Format

8.7. Return from Procedures. The return from a procedure is accomplished through the use of a return instruction. Execution of a RET instruction causes a normal return. Execution of an ERET or ERP instruction causes a return with exception information. See the individual instruction descriptions for details.

9. Exceptions

Program errors are handled by the exception facility. Exceptions may be handled by the procedure in which they occur, or by the supervisor, or by the caller of the procedure in which they occur.

9.1. Procedure Exception Handler. There shall be an entry in the context of each procedure defining the state of the current exception handler associated with that procedure. The exception handler of a procedure shall be in one of three states:

Disabled — No exception handler is currently defined for this procedure.

Handler Defined — The starting address of the exception handler for this procedure is recorded in the context stack.

Exception Code Available — No exception handler is currently defined for this procedure. The exception code for the last exception is recorded in the context stack.

The states of the exception handler for a procedure, and the typical transitions between states, are shown in figure 9-1. Encodings of the states in the context area shall be implementation dependent.

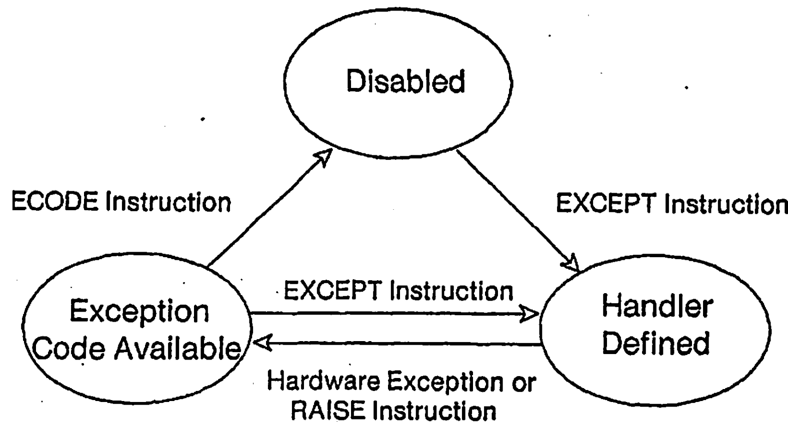


Figure 9-1: States of an Exception Handler

9.2. Definition of an Exception Handler. Upon invocation, a procedure's exception handler shall be set in the disabled state. The exception handler associated with a procedure can be defined by use of the EXCEPT instruction. When EXCEPT is executed, the exception handler for the current procedure shall be defined to start at the address specified by the EXCEPT instruction.

9.3. Raising an Exception. An exception may be raised by the RAISE, ERET, ERP, TRAISE, or PRAISE instructions, or by the detection of an abnormal condition by the hardware. An exception code is associated with each exception raised. The exception code is explicitly specified by the RAISE, ERET, ERP, TRAISE, and PRAISE instructions as an operand. The exception code specified shall be treated as an unsigned integer of 16 bits. Hardware generated exceptions shall have associated with them the following fixed codes:

Exception Name	Code (Decimal)	Described in Section
Specification.Error	1	25
Illegal.Mode	2	5.2, 5.11
Illegal.Parameter	3	8.4.2
Illegal.Register	4	5.1, 6.15
Illegal.Write	5	5.5, 5.6, 8.4.2
Bit.Field.Size	6	28
Illegal.Address	7	5.3
Operand.Size	8	5.9, 5.10, 5.12, 27
Context.Alignment	9	25
Context.Base	10	29
Segment.Specifier	11	29
Supervisor.Check	12	29
Task.Load.Error	13	25
IOC.Busy	14	29
Illegal.Divisor	16	18.2
Truncation	17	18.2
Range.Error	18	22
Invalid.Operation	19	30.4.1
Divide.By.Zero	20	30.4.2
Floating.Overflow	21	30.4.3
Floating.Underflow	22	30.4.4
Floating.Inexact	23	30.4.5
Unordered	24	30.7
Task.Failure	32	9.4, 26
Break	33	29
Instruction.Break	34	10
Call.Break	35	10

NOTE: Software should avoid using exception codes 0 to 63. These should be reserved to indicate the conditions listed above and others that may be added in the future.

9.4. Exception Handling. The control transfer on an exception is diagrammed in figure 9-2. When an exception is raised, the action that occurs shall depend upon the Up/Down Level Exception bit, bit 18 of the PSW. If UDLE is set, the state of the current procedure's exception handler shall be unaffected, and the Supervisor Exception Handler shall be invoked (described below). If UDLE is clear, the action taken shall depend on the state of the current context's exception handler. If the current state is Handler Defined, the instruction address register shall be set to the starting address specified, the T bit shall be reset, the state shall be changed to Exception Code Available, and the exception code shall be recorded in the current context area. If the current state is not Handler Defined, the action taken shall depend on the Base bit (bit 16) of the PSW. If this bit is clear, the current context shall be removed from the context stack and the exception shall be re-raised in the caller's context. If the Base bit is set, there is no caller. The current context shall be removed from the context stack and the Supervisor Exception Handler shall be invoked with the **Task.Failure** exception. The BASE bit in the PSW of the Supervisor Exception Handler shall be set.

NOTE: The exception control transfer described above may lock out interrupts Until an exception handler is found. In time critical applications that are deeply nested, an exception handler should be inserted every few levels to insure proper response to interrupts. This handler may just pass the

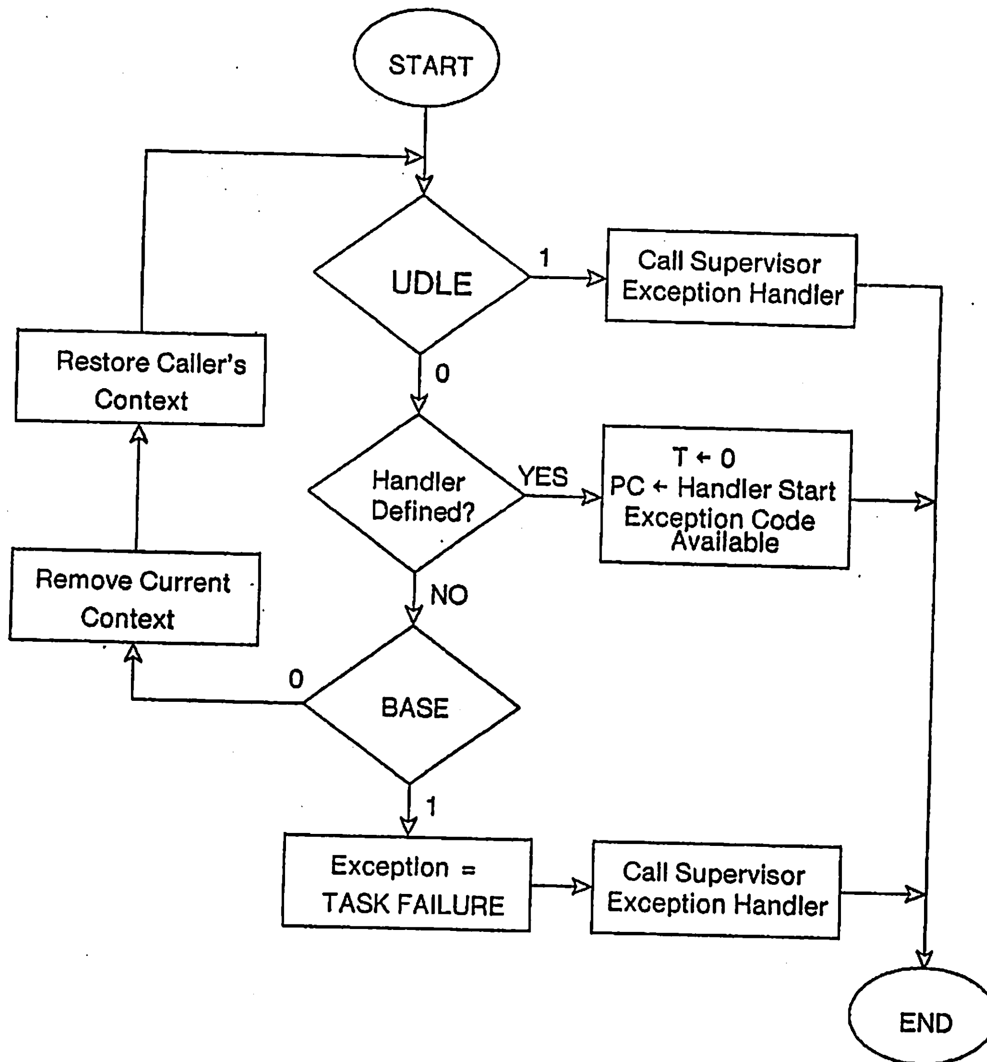


Figure 9-2: Transfer to an Exception Handler

exception on to the caller, but it will allow interrupts to be processed.

9.5. Supervisor Exception Handler. The Supervisor Exception Handler is a procedure defined by the supervisor to which control may be transferred on occurrence of an exception. This facility is enabled at the procedure level by the UDLE bit in the procedure descriptor. The entry address of the Supervisor Exception Handler is specified in a single word vector at physical address 100024 (Hex). The Supervisor Exception Handler shall be invoked as a procedure using the current context stack as described in section 8.3. There shall be three parameters to this procedure implicitly defined by the architecture:

?1 — The exception code associated with the exception. This parameter shall be a literal constant; it cannot be written. The size of this parameter is one halfword.

?2 — A reference to the opcode of the instruction causing the exception. The size of this parameter is one byte. The address of the offending instruction can be obtained by: MOVA ?2, X (where X is a local variable).

?3 — The instruction address register of the context that invoked the supervisor exception handler, as a register. This instruction address register contains the address of the instruction to be executed if the supervisor exception handler does a RET with the base bit clear. If the exception is not of type **Truncation** or break (**Break**, **Instruction.Break**, or **Call.Break**) and no contexts were removed in the propagation of the exception, then the address of the item referenced by ?2 is equal to this parameter when the supervisor exception handler is initiated. If the exception is of type **Truncation** or break and no contexts were removed in the propagation of the exception then this parameter is equal to the address of the instruction to be executed after the one that caused the exception. If any contexts were removed during the propagation of the exception, this parameter initially contains the return address from the context that invoked the Supervisor Exception Handler. This parameter may be read or written. Writing this parameter is equivalent to altering the invoking context's instruction address register by reference. If the Supervisor Exception Handler is invoked by a **Task.Failure** exception, this parameter is not defined. The size of this parameter is one word.

Thus transfer to the Supervisor Exception Handler looks like a procedure call with the exception code and offending instruction as parameters. The Supervisor Exception Handler may choose to take corrective action, abort the task, return the exception to the offending procedure (using ERP), or simply continue (using RET).

10. Debugging Facilities

The Nebula architecture provides the ability to monitor the execution of a program through tracing and break facilities at the instruction and procedure levels. These facilities operate through the supervisor exception handler mechanism that serves as the interface between the debugger and the rest of the system. Control may be transferred to the Supervisor Exception Handler either by explicit use of the BREAK instruction or implicitly by the program trace facility.

10.1. Program Tracing. Bits 13:14 of the PSW control the program trace facility. The defined settings of these bits are:

00 Disabled

01 Instruction Break. A break shall be generated after each instruction execution with exception code **Instruction.Break.**

10 Call Break. A break shall be generated after each CALL, CALLU or RET instruction execution with exception code **Call.Break.**

11 Reserved

The breaks specified by the above codes shall occur after the execution of the specified instruction and before a check for pending interrupts. An instruction that causes bits 13:14 of the PSW to be set shall not cause a break at the end of its execution. If bits 13:14 are nonzero, a break shall occur after an instruction that clears these bits.

The generation of a program break shall cause invocation of the Supervisor Exception Handler as described in section 9.5. The exception code parameter shall be set as described above.

The semantics of the procedure call mechanism imply that the setting of the trace bits (13:14 of the PSW) are propagated into called routines. Thus enabling trace in a procedure will also cause tracing of all procedures it explicitly calls. The trace setting is not propagated into SVCs, OPEXs, interrupts or traps. Thus the tracing is confined to the task in which it is initiated. Execution of an RET (or ERET or ERP) causes the trace bits to be restored to the value defined by the saved PSW. Return from a procedure will therefore cause tracing to resume as specified regardless of the conditions existing in the returning procedure.

11. Interrupts and Traps

Interrupts and traps are events requiring a change in the execution environment in the processor. Interrupts are asynchronous events generated externally or independently of the executing instructions; traps are conditions caused by the executing instructions. In the Nebula architecture interrupts and traps are treated as parameterized calls. The entry address for such a call is determined by a vector in physical memory. The interrupts and traps supported in the Nebula architecture are described below.

11.1. Interrupt Priority. Acceptance of interrupts shall be on a priority basis. The current priority of the processor shall be specified by bits 4:8 of the PSW. Priority shall be encoded as an unsigned integer with 0 being the lowest priority and 31 the highest priority. Power Fail/Restore Interrupts have the highest priority and *cannot be locked out*. I/O and software interrupt requests may be made at any priority between 1 and 31. The highest priority interrupt request with priority greater than the processor's shall be accepted. Interrupts with priority less than or equal to the processor's shall not be accepted. In the event of an I/O and a software interrupt request with equal priority, the I/O request shall be accepted first. The order of acceptance of I/O interrupt requests of equal priority is implementation dependent.

11.2. I/O Interrupts. I/O interrupts are requested by a device or controller independent of the processor. The interrupting device will specify a priority and a physical vector address. Acceptance of an interrupt shall cause the 32-bit vector located at this address to be used as the entry address of the procedure to be invoked. This procedure shall be invoked on the Kernel Context Stack as described in section 8.3. Device interrupt procedures shall be invoked with a single parameter: the physical vector address as a reference to a word in memory. The address of the physical vector may be obtained by: MOVA ?1, X (where X is a local variable). IOC interrupt procedures have a second parameter (see section 13.7). The priority of the processor (bits 4:8 of the PSW) shall be set equal to the priority of the interrupt request.

11.3. Software Interrupt Requests. An interrupt can be requested by the software using the Software Interrupt Request Register. This 32-bit register contains one bit for each priority level. Setting bit n shall cause a software interrupt to be requested at priority level n. Bit 0 is reserved. Acceptance of a software interrupt request at a given priority level shall cause the corresponding bit in the request register to be cleared. The 32-bit vector at physical address 100004 (Hex) shall specify the entry address of the procedure to be invoked. This procedure shall be invoked on the Kernel Context Stack at the requested priority level as described in section 8.3. The procedure shall have one parameter: a constant equal to the priority level of the software interrupt.

11.4. Power Failure Interrupt. Detection of a loss of power condition shall cause the following actions to occur:

1. The 32-bit word at physical address 100020 (Hex) will contain the physical address of a two word power fail save area. The hardware shall store in this save area the current contents of the Kernel Context Pointer, the Supervisor Map Pointer, and the Kernel bit (bit 0) of the current PSW.

2. The context stacks in memory shall be updated to reflect the contents of any implementation dependent context caches.

3. The procedure whose entry address is contained in the 32-bit vector at physical address 100018 (Hex) shall be invoked with no parameters on the Kernel context stack at priority 1F (Hex).

The power fail routine invoked by this mechanism may then save any other volatile information

needed to restart execution on power up. Note that the Kernel context was saved before the power fail routine was invoked. Therefore, the context of the power fail routine will be lost in the power fail.

11.5. Power Restore Interrupt. If power is applied to the processor with the contents of memory intact, the following actions shall occur as if performed in the following order:

1. The Software Interrupt Request Register and the Auxiliary Status Register shall be cleared.

2. The Kernel Context Pointer and the Supervisor Map Pointer shall be restored from the power fail save area specified above.

3. Any implementation dependent caches of the Kernel Context Stack or supervisor memory map shall be made consistent with those stored in memory.

4. The Kernel bit (bit 0 of the PSW) shall be restored from the power fail save area specified above. Next the procedure whose entry address is contained in the 32-bit vector at physical address 10001C (Hex) shall be invoked. This invocation shall use the Kernel Context Stack with no parameters at priority 1F (Hex) as a trap. Note that the Kernel bit restored from the power fail save area is used to set the last mode bit in this procedure's PSW. Return from this procedure will therefore cause the appropriate context stack to be reactivated. See Section 8.3.1 for PSW contents on procedure invocation.

The contents of all other processor registers are undefined. Restoration of these registers is the responsibility of the power restore routine.

If power is applied to the processor and the contents of memory are not intact, the IPL sequence shall be initiated.

11.6. Memory System Error Traps. The Nebula architecture provides support for detection of memory failures such as parity errors. There are two types of memory errors defined:

Soft Errors — This is an informational signal indicating that an error occurred within the memory system that was probably corrected by the memory system. The data transfer was assumed completed correctly. For example, an incorrect bit read from memory might be corrected by a Hamming code mechanism.

Hard Errors — An uncorrectable error was detected in a memory data transfer. The information transferred is probably wrong.

The ability to detect either type of error is implementation dependent. If this ability is provided, it shall function as specified below.

11.6.1. Hard Memory Errors. CPU hard memory errors shall cause a trap using the Kernel context stack. The address of the entry point for the procedure to be invoked shall be specified by the vector at physical address 100014 (Hex). The procedure shall be invoked with a single parameter that is a reference to the byte in memory whose attempted access caused the trap. The size of this parameter is one byte. The address of this byte can be obtained by: MOVA ?1, X. The execution of a return (RET) instruction within a hard memory error trap handler shall produce unpredictable behavior. A return should not be executed within a hard memory error trap handler. If a hard memory error is detected during the building of a context on the Kernel Context Stack, the RESET function described in section 11.9 shall be invoked.

11.6.2. Soft Memory Errors. CPU soft memory errors may be masked. This allows the processor to be informed of such errors without being saturated by repeated reports. If bit 0 of the auxiliary status register (figure 7-1) is clear, soft errors shall be ignored. If this bit is set, a soft memory error shall cause it to be cleared and a trap shall occur using the Kernel context stack. The address of the entry point for the procedure to be invoked shall be specified by the vector at physical address 100010 (Hex). The procedure shall be invoked with a single parameter that is a reference to the byte in memory whose attempted access caused the trap. The size of this parameter is one byte. The address of this byte can be obtained by: MOVA ?1, X. The execution of a return (RET) instruction within a soft memory error trap handler shall return control, in a transparent fashion, to the context that was executing prior to the trap.

11.7. Privileged Instruction Trap. An attempt to execute a privileged instruction with bit 15 of the PSW clear shall cause a trap using the Kernel Context Stack. The address of the entry point for the procedure to be invoked shall be specified by the vector at physical address 100028 (Hex). The procedure shall be invoked with a single parameter. The parameter shall be a byte size reference to the opcode of the offending instruction. The address of this instruction can be obtained by: MOVA ?1, X. The execution of a return (RET) instruction within a privileged instruction trap handler shall result in the resumption of the context that was executing prior to the trap beginning with the instruction referenced by parameter 1.

11.8. Memory Management Traps. A memory management trap is generated by an attempt to perform a memory access that is specified as invalid by the memory management facility (see section 12). Such an access shall cause a trap using the Kernel Context Stack. The address of the entry point for the procedure to be invoked shall be specified by the vector at physical address 10000C (Hex). The procedure shall be invoked with four parameters as described in section 12.4.

A memory access that could potentially result in a memory management trap may be made in the following cases:

- During instruction execution
- While building or unwinding contexts on the Task Context Stack
- While building or unwinding contexts on the Kernel Context Stack.

11.8.1. Instruction Execution. An invalid memory access detected by the memory management system during instruction execution shall suspend the instruction and cause a trap on the Kernel Context Stack. The execution of a return (RET) instruction within the memory management trap handler shall result in the resumption or restarting of the suspended instruction.

11.8.2. Task Context Stack. An invalid memory access detected by the memory management system during the building or unwinding of contexts on the Task Context Stack shall suspend the context activity and cause a trap on the Kernel Context Stack. The execution of a return (RET) instruction within the memory management trap handler shall result in the resumption of the suspended context in a transparent fashion.

11.8.3. Kernel Context Stack. An invalid memory access detected by the memory management system during the building or unwinding of contexts on the Kernel Context Stack shall suspend the context activity and cause a trap on the Kernel Context Stack.

If the invalid memory access is detected during the building of a context for an interrupt invocation, the context stack is cleared of context information relating to the interrupt. The execution of a return (RET) instruction within the memory management trap handler invoked under these circumstances shall resume execution of the context that was executing prior to the interrupt.

If the invalid memory access is detected during the building of a context for a trap other than a memory management trap, the execution of a return (RET) instruction within the memory management trap handler shall resume execution of the context that was executing prior to the trap.

If the invalid memory access is detected during the building of a context for a memory management trap, the RESET function described in section 11.9 shall be invoked.

11.9. Reset and IPL. When the reset switch is activated or the RESET instruction is executed, the contents of the 32-bit word at physical address 100044 (Hex) shall be interpreted as the address of a two word block containing values for the Kernel Context Pointer and supervisor map pointer, in the same format as the Power Fail save area. These values shall be loaded into the respective hardware registers. The supervisor memory map shall be made consistent with the new map specified. The ASR and Software Interrupt Request Registers shall be cleared. The contents of the 32-bit vector at physical location 100040 (Hex) shall be interpreted as the entry address of the procedure to be invoked. This procedure shall be invoked on the Kernel Context Stack as an interrupt with priority 1F (Hex) with one parameter. The size of this parameter is one byte. This parameter shall indicate the source of the reset action. The currently defined values for this parameter are:

Cause	Code
Panel Switch	1
RESET Instruction	2
Invalid Memory Access	3
Hard Memory Error	4
IPL Sequence	5

The last mode bit (bit 1 of the PSW) shall be clear. During the reset sequence, no interrupts shall be accepted. The IOCs shall be placed in the state defined when power is applied.

When the IPL switch is activated, the identical sequence of operations shall take place except that the entry address shall be specified in the data loaded by the IPL sequence.

If, during a RESET or IPL sequence, the value contained in the relocation and protection bits of a map pointer register is reserved, or a hard memory error is encountered then the system shall halt in an implementation dependent manner. See section 12.2.1 for details about map pointer registers.

Note that the RESET and IPL sequences are identical to the power restore sequence except that no prior Kernel context is assumed.

11.10. Built In Test Traps. The implementation dependent Built In Test (BIT) facilities associated with the CPU and IOCs will have the option of generating a trap on the Kernel Context Stack. The address of the entry point for the procedure to be invoked shall be specified by the vector at either physical address 100008 (Hex) or physical address 10002C (Hex). The circumstances under which these traps are generated shall be implementation dependent. The execution of a return (RET) instruction within a BIT trap handler shall produce an implementation dependent action.

11.11. Simultaneous Events. The simultaneous occurrence of the following events shall be handled in the priority shown: Trap (1st), Exception (2nd), Break (3rd), Interrupt (4th).

12. Memory Management System

The memory management system performs the following functions:

- Translation of virtual addresses to physical addresses
- Separation of areas of the virtual address space according to allowed accesses
- Separation of physical addresses into memory and I/O space addresses
- Maintenance of access information

These functions are controlled by memory maps stored in physical memory that are pointed to by map pointer registers in the processor. This relationship is shown in figure 12-1.

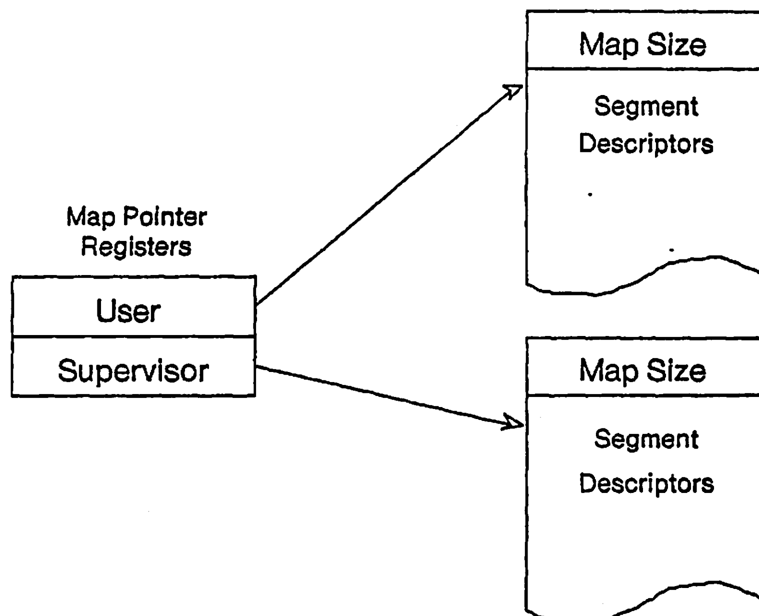


Figure 12-1: Map Data Structure

12.1. Virtual Address Space. The size of the virtual address space is defined as the number of distinct byte virtual addresses that can be generated by the processor. In the Nebula architecture two distinct measures of the virtual address space are defined:

- The Architectural Virtual Address Space is the number of distinct byte virtual addresses that can be generated by the addressing modes of the architecture. These addressing calculations are visible to the programmer through the MOVA (move address) instruction. The architectural virtual address space shall be 2^{32} bytes.
- The Implementation Virtual Address Space is the minimum of the number of distinct byte

virtual addresses that can be generated and mapped using an addressing mode with memory mapping enabled. In the Nebula architecture this may be less than the Architectural Virtual Address Space due to limitations in the parameter descriptor and mapping mechanisms. The implementation virtual address space shall be a minimum of 2^{24} bytes or 8 times the maximum implemented physical address space, whichever is greater.

These restrictions are designed to allow reasonable optimization of the parameter and memory map implementation while avoiding any software dependencies on these implementation dependencies. The manner in which the software manipulates addresses is driven by the architectural virtual address space, that has been specified as the full 32 bits. The implementation virtual address space impacts the allocation of address space. This is specified to exceed the point at which other software efficiency considerations conspire to limit allocation.

If an implementation virtual address must participate in an address calculation, it is converted into an architectural virtual address by placing a zero bit in each bit position defined for the architectural virtual address but not defined for the implementation virtual address.

12.2. Mapping of Virtual Addresses. The virtual address space is divided into two halves, one of which is accessible only to supervisor programs. Virtual addresses with the most significant bit (bit 0) set are mapped through the supervisor memory map. Virtual addresses with the most significant bit clear are mapped through the User memory map.

12.2.1. Map Pointer Registers. There shall be two map pointer registers; one for Supervisor and one for User state. The map address register used to translate a virtual address shall depend on bit 0 of the virtual address. If bit 0 is clear, the User Map Pointer register shall be used. If bit 0 is set, the Supervisor Map Pointer register shall be used. The format of these registers is shown in figure 12-2. These registers contain the physical address in memory of the memory maps. Bits 29:31 of these addresses shall be assumed 0. The address in these registers shall be the address of the first map entry in the memory map. Bits 30:31 control the translation of addresses using the selected map pointer register. These bits shall function as follows:

00	No relocation or protection. Bits 1:31 of the virtual address shall be used as a physical address. No segment association shall be performed. I/O space accesses shall be segregated as described in section 12.5.
01	Protection enabled. Accesses shall be checked for validity as described below. No relocation is performed. Physical addresses are determined as above.
10	Reserved. If this code is specified in the map pointer to be loaded by a LTASK instruction, a **Specification.Error** exception shall be raised. Section 11.9 describes the action taken when this code is encountered during a RESET or IPL sequence.
11	Relocation and Protection. Addresses shall be checked and relocated as described below.

12.2.2. Memory Map Structure. The memory map pointed to by a map pointer register shall be assumed to have the following format. The mapping information for each segment is stored in a 64-bit double word with the format shown in figure 12-3. These map entries are stored in consecutive double words, starting with the entry for segment zero, and aligned on double word boundaries. The number of entries in the map is stored in a 32-bit word preceding the segment zero entry. The maximum number of segments allowed is at least 16. The actual number allowed is implementation dependent: If the number of entries indicated by the word preceding the segment zero entry is

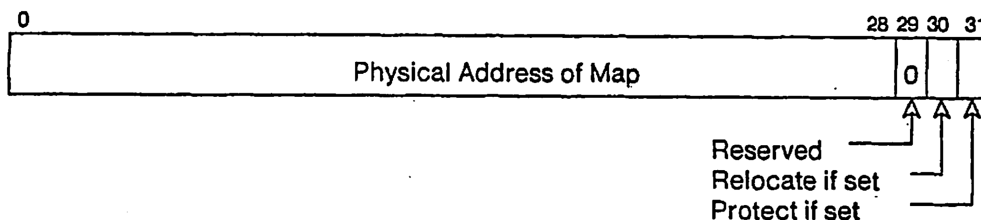


Figure 12-2: Memory Map Pointer Registers

greater than the number of segments implemented, entries beginning with the segment zero entry up to the number implemented are used for segment association. The address in the map pointer register is the address of the segment zero entry.

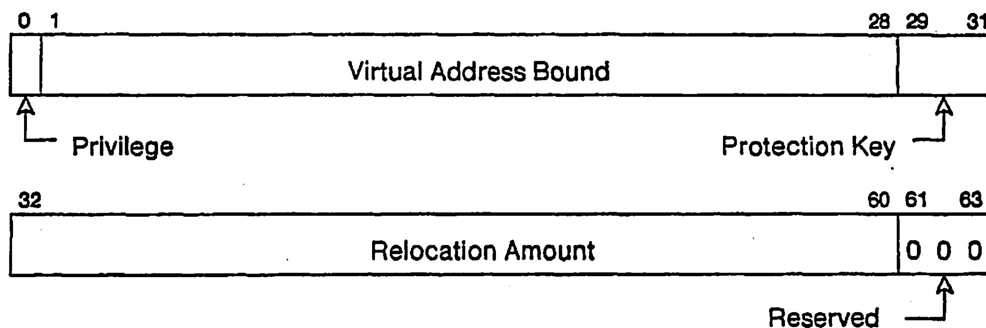


Figure 12-3: Map Entry Format

12.2.3. Segment Association. The relocation and protection functions of the memory management system operate by associating one of the segments specified by the map entries with a virtual address. This association is based on the virtual address bound field (bits 1:28) of the map entries. A virtual address is contained in a particular segment if:

1. Bits 1:28 of the virtual address are less than or equal to the virtual address bound of the segment.

2. Bits 1:28 of the virtual address are greater than the virtual address bound contained in the previous map entry. This condition shall be satisfied vacuously for map entry 0.

A virtual address for which the above conditions are not satisfied by any map entry shall cause a memory management trap with fault code **Invalid.Segment.**

The map entries will be ordered such that the virtual address bound in each map entry is greater than the bound in the previous map entry. Violation of this restriction by the operating system may produce unpredictable results.

12.2.4. Relocation of Virtual Addresses. If bit 30 of the selected map pointer register is set, relocation is enabled. The virtual address shall be associated with a map entry as described above. Bits 1:28 of the virtual address shall be zero extended and then added to the signed integer relocation field of the map entry. The result of the addition is truncated to 29 bits (modulo 2^{29}) to form bits 0:28 of the physical address. Bits 29:31 of the physical address shall be equal to bits 29:31 of the virtual address. This operation is shown in figure 12-4. No two distinct physical addresses generated shall correspond to the same implemented byte of memory (or I/O space). An attempt to access a physical address that does not correspond to implemented memory or I/O registers shall cause a hard memory error trap (see section 11.6.1).

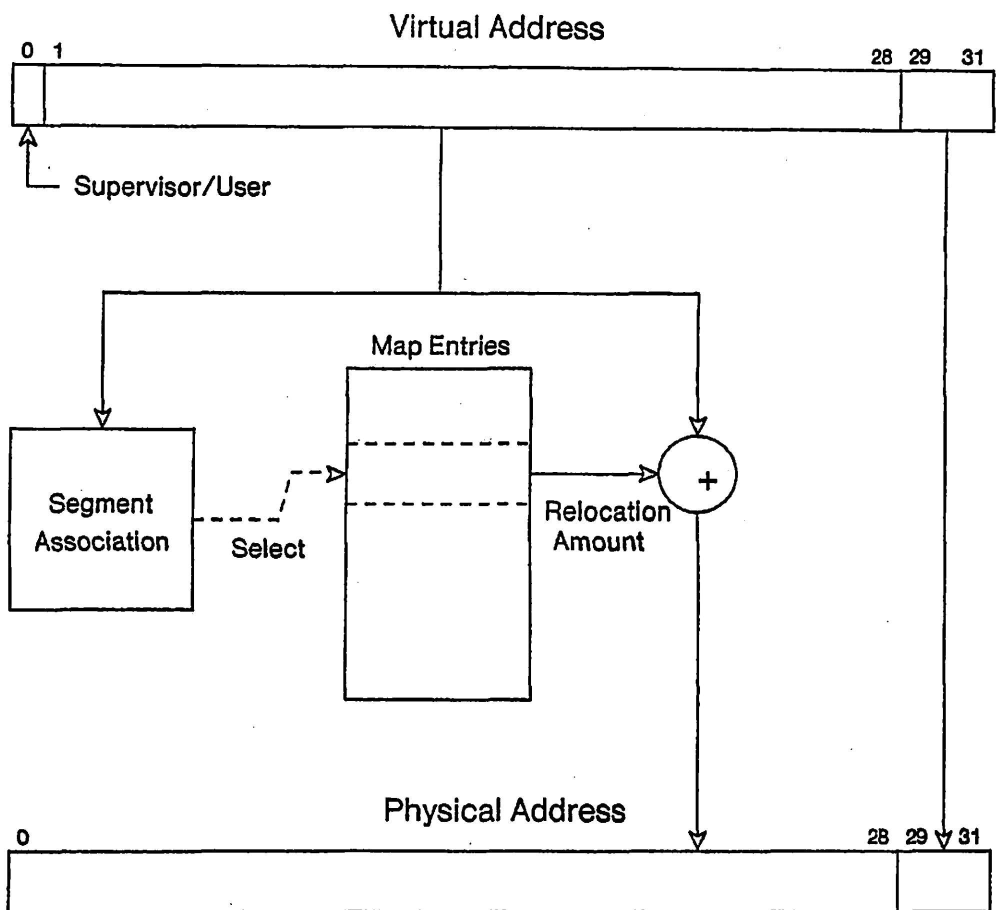


Figure 12-4: Address Relocation

12.2.5. Access Protection. If bit 31 of the selected map pointer register is set, protection is enabled. The virtual address shall be associated with a map entry as described above. Bit 0 of the map entry specifies whether access to this segment is a privileged operation. If this bit is set, and bit 15 of the PSW is clear, a memory management trap shall be generated with fault code **Privilege.Violation**. Bits 29:31 of the map entry specify the access types allowed for this segment. The encoding of this field is:

000	No Access. Any access shall cause a trap.
100	Instruction Access Only. Instruction fetch and reading of literals in short literal or literal operands is allowed.
001	Data Read Only. Reading of operands is allowed.
101	Instruction or Read Access. Instruction fetch, literal fetch and reading of operands is allowed.
010	Data Read/Write. Reading or writing of operands is allowed.
110	Reserved. Any access shall cause a trap.
011	Context Only. Access as part of a context stack is allowed. Any other access shall cause a trap.
111	Reserved. Any access shall cause a trap.

In all cases accesses other than those explicitly allowed shall cause an **Invalid.Access** trap (see section 12.4 below). Note in particular that context stacks can only be allocated in segments with access code 011. Such segments cannot be manipulated explicitly by any instruction. This is vital to the security of the system and the transportability of the software.

12.2.5.1. Self-Modifying Code. If access protection is disabled, it is possible to execute instructions that write their operands into the instruction stream in the immediate vicinity of the currently executing instruction. This type of code sequence is called self-modifying code. Since modern implementation techniques usually require some type of instruction pre-fetch, the action of such self-modifying code is unpredictable. Modifications (or data writes) to the instruction stream are guaranteed to be interpreted as stored only if a REPENT or LTASK instruction is executed before execution of the modified instruction stream is begun.

12.2.6. Crossing Segment Boundaries. When a 2, 4, or 8 byte primitive data object (see section 4.5) is being accessed, segment association, relocation and protection checks function as if the object were referenced one byte at a time. When a value is being stored into such a multi-byte object, if a transfer of any byte is blocked by the memory management system, then no bytes shall be stored.

While accessing instructions during execution or while accessing the context stack, segment association, relocation and protection checks function as if the instruction or context stack were being accessed one byte at a time. If the allocation of a procedure context is blocked by the memory management system, then none of the context shall be allocated.

12.2.7. Protection of the Supervisor. A virtual address with bit 0 set shall cause an **Invalid.Supervisor** trap if bit 17 of the PSW is clear.

12.3. Implementation Considerations.

12.3.1. Cacheing of Memory Maps. In many implementations it will be desirable to cache parts of the memory maps, such as the map size and a few recently used map entries. The properties of any such cacheing mechanism are implementation dependent. However, the following consistency requirements shall be met:

- Setting a Map Pointer Register using the LTASK instruction shall force the cache to be consistent with the specified map.
- Alteration of a map entry using the REPENT instruction shall force the cache to be consistent with the new entry and the map size of the altered map, as specified in memory.

12.3.2. Aliasing of Physical Addresses. Using the relocation facility of the memory management system, it is possible to map two distinct virtual addresses onto a single physical address. This is known as aliasing of a physical address. In pipelined implementations, it may be desirable to use virtual addresses for data access coordination. In this case, the order of multiple accesses to the same physical address through different virtual addresses in the same map is unpredictable. The practice of aliasing physical addresses within a single map should be avoided.

12.4. Memory Management Traps. A memory management trap is generated by an attempt to perform a memory access that is specified as invalid by the memory management facility. Such an access shall suspend the current execution context and cause a trap using the Kernel Context Stack. The address of the entry point for the procedure to be invoked shall be specified by the vector at physical address 10000C (Hex). The procedure shall be invoked with four parameters:

?1 — A reference to the byte in memory whose attempted access caused the trap. The size of this parameter is one byte. The address of this byte can be obtained by: MOVA ?1, X.

?2 — A reference to the opcode of the instruction causing the trap. The size of this parameter is one byte. The address of the offending instruction can be obtained by: MOVA ?2, X.

?3 — The segment number of the segment containing the invalid virtual address as a constant. The value of this parameter is undefined in the case of an **Invalid.Supervisor** or **Invalid.Segment** fault. The size of this parameter is one byte.

?4 — The memory management fault code as a constant. The size of this parameter is one byte.

The following memory management fault codes are defined:

Name	Code
Invalid.Supervisor	1
Invalid.Segment	2
Invalid.Access	3
Privilege.Violation	4

12.5. I/O Space Selection. Communication with I/O devices in the Nebula architecture is accomplished with memory access. A portion of the physical address space is reserved for I/O device registers. An access to one of these addresses transfers information to an associated device control register. The first 2^{20} physical addresses (00000000_{16} to $000FFFFF_{16}$) shall be used to select I/O space.

12.6. Subseting of Memory Management. A particular implementation may subset the functionality of the memory management system in three ways:

1. The size of the virtual addresses translated may be reduced as specified above under implementation virtual address space.

2. The size of the physical addresses generated may be reduced to correspond to the physical memory accessible to the implementation. However, the requirements of section 12.5 shall be met.

3. The segment association function may be reduced. The full architecture allows segment bounds to be specified on any doubleword boundary. The alignment requirement may be increased up to 256 byte boundaries. If this is done, the unused low order bits of the virtual address bound field of the map entries shall be reserved. The relocation amount field shall be reduced by the same number of bits; these bits shall also be reserved. The number of low order bits of the virtual address that are not changed by the relocation process shall be increased by the same number of bits.

13. I/O Controllers

Interaction with I/O devices in the Nebula architecture is provided by the ability to access control registers as locations in the I/O portion of the physical address space. Some implementations will provide I/O through a fixed set of ports or channels. This section defines a standard I/O controller architecture for handling transactions on such a channel. The I/O controller (IOC) is a processor with a limited instruction set (described below). This instruction set allows the IOC to perform predefined transfer, sequencing, and control functions without processor intervention. I/O transactions between the processor and IOC are structured as messages containing the specifics of the transfer requested. The IOC instruction set is designed to process such messages in a general manner and to provide additional intelligence to devices connected to its I/O channel. Most of the IOC instruction set is general and operates with any type of interface. However, some interfaces require that specific functions be provided for their proper operation. Channel specific information is provided below for the following interface types:

- Serial Point-to-Point interface (compatible with RS449)
- 16-bit Parallel Point-to-Point interface
- MIL-STD-1553 Serial Multiplex Data Bus interface

13.1. Channel Configuration Registers. Associated with each channel shall be a channel configuration register. This register shall be addressable by the processor as a 32-bit word in the I/O address space. The bits in this register shall be used to specify channel dependent parameters (such as parity or bit rate) for the associated interface. Bits 22:26 of this register shall specify the maximum priority allowed for programmed interrupt requests. Bits 27:31 shall specify the interrupt priority for IOC error interrupts from this channel. The assignment of the other bits in this register is implementation dependent. The contents of this register are undefined when power is applied to the IOC.

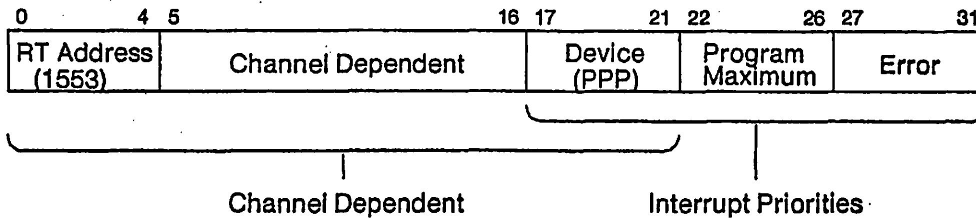


Figure 13-1: Channel Configuration Register

13.2. IOC Programs. The operation of each channel is controlled by one or more channel programs. The execution of channel programs shall appear parallel. Each parallel point to point (PPP) and MIL-STD-1553 channel shall support a single channel program. Each serial point to point (SPP) channel shall support 2 channel programs; one for input and one for output. Execution of a read operation in a SPP output program or a write operation in a SPP input program shall cause an **Illegal.Operation** fault.

Each channel program has a 32-bit channel program counter, a 32-bit message pointer register, a 16-bit accumulator, a 16-bit channel status, and a 16-bit channel program status register associated with it. These registers are described below. The channel program may perform transfer operations using the data in the record specified by the message pointer register as parameters of the transfer. The channel program may interrupt the processor as required, and return information about the requested transfer in the message.

13.2.1. Program Counter. The channel program counter (CHPC) is a 32-bit register that contains the virtual address of the next instruction in the channel program. This register is visible in the processor I/O address space. A channel program is specified by loading an address into this register. Channel program counters are restricted to even addresses. An odd address in a channel program counter shall cause an IOC error interrupt to be generated with fault code **Program.Alignment**. Writing of this register by the CPU while the channel program is running (bit 1 of the channel status set) shall cause an IOC error interrupt with fault code **IOC.Active**. In this event the contents of the register are undefined. Recognition of this error condition may be delayed until the IOC next examines the register. Section 13.7.1 describes the error interrupt mechanism. When power is applied to the IOC, bit 31 of the CHPC shall be 0 and the contents of bits 0:30 are undefined.

13.2.2. Message Pointer Register. The message pointer register (MP) is a 32-bit register that contains the virtual address of the current message being processed. This register is visible in the processor I/O address space. The value in this register must be even. An odd value in a message pointer register shall cause an IOC error interrupt with fault code **Message.Alignment**. Writing of this register by the CPU while the channel program is running (bit 1 of the channel status set) shall cause an IOC error interrupt with fault code **IOC.Active**. In this event the contents of the register are undefined. Recognition of this error condition may be delayed until the IOC next examines the register. Section 13.7.1 describes the error interrupt mechanism. When power is applied to the IOC, bit 31 of the MP shall be 0 and the contents of bits 0:30 are undefined.

13.2.3. Accumulator. The accumulator is a 16-bit register used by IOC instructions as an implicit operand. It is used as a computational temporary and a unit count register for transfer instructions. When power is applied to the IOC, the contents of the accumulator are undefined.

13.2.4. Channel Status. The channel status is a 16-bit register containing control information related to the channel. Bit 0 of this register is the run control bit. Setting the run bit shall signal the channel to begin execution of the channel program. The IOC shall respond by setting bit 1 (run status) and beginning execution of the instruction indicated by the channel program counter. Clearing bit 0 shall signal the channel to halt. The IOC shall perform an orderly termination of the operation currently in progress and then stop execution of the channel program and clear bit 1 of the channel status. Bits 0 and 1 are also cleared by channel program initiated halts. Bit 15 of the channel status register is reserved for reporting of Built In Test (BIT) detected problems. This bit is set to 1 by the implementation dependent BIT facilities to indicate a BIT detected problem. Bits 2:14 of the channel status register are implementation dependent. When power is applied to the IOC, bits 0, 1, and 15 of the channel status register are cleared and bits 2:14 are undefined. When power is applied to the IOC, the IOC shall be placed in the halt state.

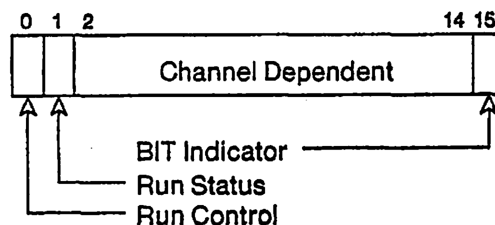


Figure 13-2: Channel Status Register

13.2.5. Channel Program Status. The channel program status is a 16-bit quantity that contains information about the last transfer operation requested by the channel program. This register is visible in the processor I/O address space. The format of this halfword is shown in figure 13-3.

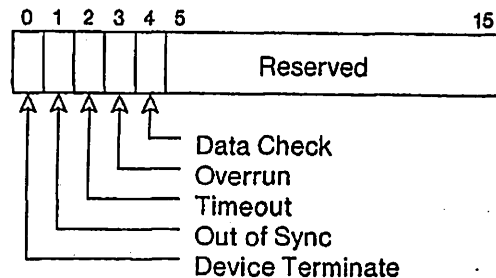


Figure 13-3: Channel Program Status

The bits of the channel program status halfword indicate which of several exceptional conditions may have caused termination of the transfer. Some of these error conditions are not detectable by some channels. The assigned fields are:

Device Terminate The device signaled transfer completion prior to exhaustion of the transfer count.

Out of Sync The transfer type or direction specified by an external device does not match the transfer instruction executed (PPP).

Timeout A transfer was not acknowledged within a channel dependent time.

Data Overrun Data was lost on a read due to the inability of the IOC to complete the transfer prior to the arrival of the next data.

Data Check An error was detected in the data received.

The associated bit is set to indicate the presence of the exceptional condition. A zero (all bits cleared) in the channel program status register shall indicate that no exceptional conditions were detected in the preceding transfer. When power is applied to the IOC, the contents of the Channel Program Status register are undefined.

13.3. Virtual Addressing. There shall be 3 virtual segments associated with each channel program. All memory accesses made by a channel program shall be made through one of these segments. There is one segment for instruction access, one for message access, and one for data access. The physical memory associated with each of these segments can be defined by the **Set I/O Segment** instruction. This instruction selects a segment from one of the CPU maps for use as one of the channel program's segments and verifies that the access key for the segment corresponds to the use intended. Segments defined as instruction segments are checked for instruction access; message and data segments are checked for read/write access. All instruction and literal accesses shall be made through the instruction space. The space used for data accesses is specified in the individual instruction descriptions. The IOC shall check that program specified addresses lie within the bounds of the corresponding segment. Valid addresses shall be relocated as specified in the selected segment. Invalid addresses shall cause an **Addressing.Error** fault. The associations for IOC virtual addressing are depicted in figure 13-4.

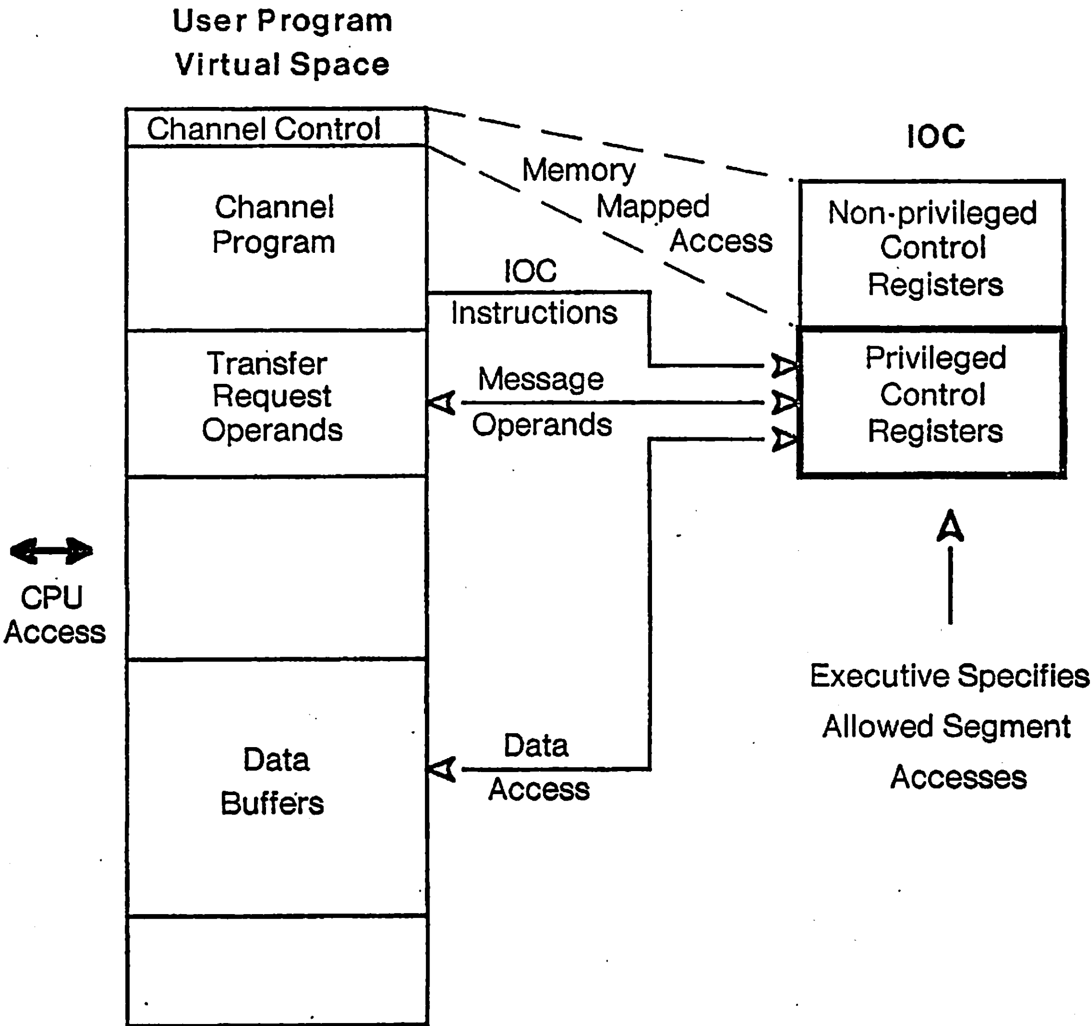


Figure 13-4: IOC Virtual Addressing

13.3.1. Segment Specifiers. IOC virtual segments are defined by using the **Set I/O Segment** (SETSEG) instruction to define an IOC segment specifier. The segment specifier to be defined is referenced by a memory mapped address in the IOC control registers. The control register addresses reserved for this purpose should not be accessed using instructions other than SETSEG. The effect of referencing these addresses with other instructions is unpredictable. The reference address for each segment specifier, relative to the base of the IOC control registers, is defined in the table below.

Relative Address (Hex)	Segment Specifier
10	Channel Program
20	Message
30	Data Buffer

When power is applied to the IOC, all three segments shall be set to prohibit all accesses.

13.4. Physical Addressing. IOC program, message, and data buffer references shall address only physical memory locations with addresses greater than or equal to 00100000 (hex). References to addresses less than 00100000, or to addresses greater than the implemented memory shall cause an IOC error interrupt with fault code **Memory.Error.** This fault shall also be generated if a hard memory error occurs on a transfer to a valid physical address.

13.5. Instruction Execution. IOC instructions are represented as one or more halfwords aligned on halfword boundaries. On an active channel, the channel program counter shall contain the virtual address of the next instruction to be executed. The channel program counter shall be incremented by the number of bytes in each instruction fetched.

13.6. Operand Accessing. Instruction operands are specified either implicitly in the accumulator or as an explicit literal or displacement in the instruction stream. Access to a literal in the instruction stream shall cause the channel program counter to be incremented by the number of bytes in the operand. Operands in the message are selected by a displacement field in the instruction. Twice this displacement added to the contents of the message pointer register shall be the virtual address of the operand in the message.

13.7. IOC Interrupts. The IOC shall have the ability to interrupt the processor. Processor interrupts may be initiated by IOC detected errors or by execution of an IOC interrupt instruction. An IOC interrupt shall invoke a service routine at the address specified by the contents of the IOC error or program interrupt vector, respectively. Each channel program is assigned one error vector and one program vector. Interrupts shall be held pending by the IOC until acknowledged by the CPU.

The service routine shall be invoked with two parameters:

?1 — The physical address of the interrupt vector used, as a reference to a word in memory. The address of the interrupt vector can be obtained by: MOVA ?1, X (where X is a local variable).

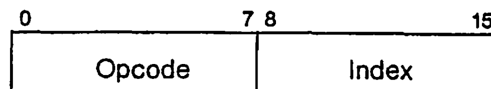
?2 — An interruption code identifying the nature of the interrupt. This is an operand specified by the IOC interrupt instruction or is implicitly determined by the nature of the IOC error. This parameter is read-only.

The priority of a programmed interrupt request is specified by the IOC interrupt instruction. The interrupt priority for IOC errors is defined by the channel configuration register for the interrupting channel.

13.7.1. IOC Error Interrupts. Detection of certain errors by the IOC shall cause an IOC Error Interrupt to be generated. The channel program executing at the time the error is detected shall be halted and bits 0:1 of the channel status register shall be cleared. The interruption code shall be fixed by the type of error:

Fault	Fault Code (Hex)	Described in Section
Program.Alignment	1	13.2.1
Message.Alignment	2	13.2.2
Illegal.Opcode	3	13.8
Addressing.Error	4	13.3
IOC.Active	5	13.2
Illegal.Operation	6	13.2, 13.8.4
Interrupt.Priority	7	13.8.3
Memory.Error	8	13.4

13.8. IOC Instructions. IOC instructions have the general form shown below:



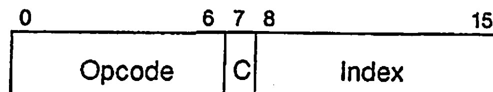
The index field is used as a halfword displacement from the message pointer register or the channel program counter, depending upon the instruction (parentheses indicate a "contents of" operation). The instruction descriptions use the notation (MP + index*2) to refer to the halfword in the message addressed by this index field. The index field and 16-bit displacements in the instruction stream shall be interpreted as unsigned quantities unless otherwise specified. The notation CHPC is used to refer to the channel program counter for the executing channel program.

The opcode (the value of bits 0:7 in Hex) and function of each instruction is described below. Fetch of any instruction other than those described below shall cause an IOC error interrupt with fault code **Illegal.Opcode.**

13.8.1. IOC Instruction Descriptions. The IOC instruction descriptions below have two parts. The first part after the name of the instruction is unlabeled and lists the assembler mnemonic, a short symbolic description of the instruction's action if possible, and the opcode of the instruction in Hex. The second part of the instruction description contains an English-language verbal description of the action of the instruction. This second part is preceded by the label *Description.* The verbal description and the symbolic description for each instruction are intended to complement each other. Omissions in one are resolved in the other. The verbal description has priority in resolving ambiguities.

13.8.2. Transfer Instructions. The transfer instructions read or write a block of data to or from the channel. These instructions have the form shown below: Bit 7 of the opcode is used to distinguish between control transfers (command and status information) and data transfers. A zero in this bit shall specify a data transfer; a one shall specify a control transfer. The ability to distinguish between command and data transactions is channel dependent. On SPP channels this bit shall be ignored.

The information to be transferred is specified in general by a buffer address and unit count. This *unit count* is the number of memory data units to be transferred. The size of a memory data unit is channel dependent. The MIL-STD-1553 and parallel channels shall transfer 16-bit halfwords while the



SPP channels shall transfer 8-bit bytes.

In general, the number of units to be transferred is specified by a unit count in the accumulator. After a transfer, the number of units not transferred is placed in the accumulator. Transfer instructions executed by MIL-STD-1553 channels interpret the accumulator slightly differently. Refer to section 13.8.4.3.

In the event a transfer is terminated prior to completion of the specified number of data transfers, the cause of the termination shall be indicated in the channel program status. Refer to section 13.2.5.

Read

READ	02
READS	03

Description:
A read operation is initiated on the channel. The data is read from the channel into the data buffer. The location of the data buffer is specified by a 32-bit virtual address in the message. Bits 8:15 of the instruction specify an unsigned displacement (in halfwords) from the message pointer contents. The buffer virtual address is obtained from this location in the message. This address is checked to insure that it is a valid address in the data segment. The accumulator contains the maximum number of units to be transferred. The type of the transfer is determined by the instruction (data for READ, status for READS). When the transfer has terminated, the accumulator shall contain the initial unit count minus the actual number of units transferred.

Read to Message

RDTMSG	04
RDTMSGS	05

Description:
A read operation is initiated on the channel. The data is read from the channel into the message. Bits 8:15 of the instruction specify an unsigned displacement (in halfwords) from the message pointer contents. The data is stored at this location in the message. The accumulator contains the maximum number of units to be transferred. The type of the transfer is determined by the instruction (data for RDTMSG, status for RDTMSGS). When the transfer has terminated, the accumulator shall contain the initial unit count minus the actual number of units transferred.

Write

WRITE	06
WRITEC	07

Description:
A write operation is initiated on the channel. The data is written from the data buffer to the channel. The location of the data buffer is specified by a 32-bit virtual address in the message. Bits 8:15 of the instruction specify an unsigned displacement (in halfwords) from the message pointer contents. The buffer virtual address is obtained from this location in the message. This address is checked to insure that it is a valid address in the data segment. The accumulator contains the maximum number of units to be transferred. The type of the transfer is determined by the instruction (data for WRITE, command for WRITEC). When the transfer has terminated, the accumulator shall contain the initial unit count minus the actual number of units transferred.

Write from Message

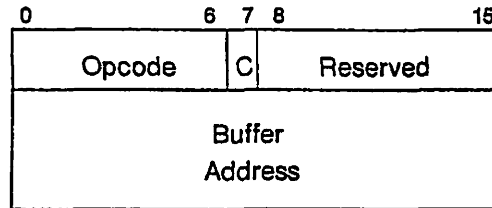
WRFMSG	08
WRFMSGC	09

Description:
A write operation is initiated on the channel. The data is written from the message to the channel. Bits 8:15 of the instruction specify an unsigned displacement (in halfwords) from the message pointer contents. The data stored at this location in the message is output. The accumulator contains the maximum number of units to be transferred. The type of the transfer is determined by the instruction (data for WRFMSG, command for WRFMSGC). When the transfer has terminated, the accumulator shall contain the initial unit count minus the actual number of units transferred.

Write Literal

WRLIT	0A , 32-bit address
WRLITC	0B , 32-bit address

Description:
A write operation is initiated on the channel. The data is written from the instruction segment to the channel. Bits 8:15 of the instruction are reserved. The instruction is followed by a 32-bit virtual address of the data to be written. This address is interpreted as being in the instruction segment. The data stored at this location is output. The accumulator contains the maximum number of units to be transferred. The type of the transfer is determined by the instruction (data for WRLIT, command for WRLITC). When the transfer has terminated, the accumulator shall contain the initial unit count minus the actual number of units transferred. The format of the WRLIT instruction is shown below.



13.8.3. Control Instructions. The following instructions provide the general data manipulation facilities needed for control functions.

LOAD

LOAD	AC <- (MP + index*2)	0C
LOADL	AC <- Literal	0D , 16-bit literal

Description:
The specified halfword in the message (for LOAD) or the 16-bit literal following the instruction (for LOADL) is placed in the accumulator. Bits 8:15 of the LOADL instruction are reserved.

Load Status

LOADST	AC <- CCR[offset]	14

Description:
Bits 8:15 of the instruction specify one of the memory mapped channel control registers. The value of bits 8:15 of the instruction, with bit 15 forced to 0, shall be interpreted as an address offset from the beginning of the upper half of the channel control registers. The contents of the 16-bit halfword at this location are placed in the accumulator.

Store

STORE	(MP + Index*2) <- AC	15

Description:
The contents of the accumulator are stored at the specified location in the message.

ADD

IADD	AC <- AC + (MP + index*2)	0E
IADDL	AC <- AC + Literal	0F, 16-bit literal

Description:
The indexed halfword in the message (for IADD) or the 16-bit literal following the instruction (for IADDL) is added to the accumulator. Bits 8:15 of the IADDL instruction are reserved.

Subtract

ISUB	AC <- AC - (MP + index*2)	17

Description:
The indexed halfword in the message is subtracted from the accumulator.

Logical AND

IAND	AC <- AC AND (MP + index*2)	10
IANDL	AC <- AC AND Literal	11, 16-bit literal

Description:
The logical AND of the accumulator contents and the indexed halfword (for IAND) or the 16-bit literal following the instruction (for IANDL) is placed in the accumulator. Bits 8:15 of the IANDL instruction are reserved.

Logical OR

IOR	AC <- AC OR (MP + index*2)	12
IORL	AC <- AC OR Literal	13, 16-bit literal

Description:
The logical OR of the accumulator contents and the indexed halfword (for IOR) or the 16-bit literal following the instruction (for IORL) is placed in the accumulator. Bits 8:15 of the IORL instruction are reserved.

Add to Address

ADDTA	(MP + index*2) <- (MP + index*2) + AC	1C

Description:
Bits 8:15 of the instruction specify the location of a 32-bit word in the message. The contents of the accumulator are interpreted as a signed quantity and added to this word.

Logical Shift

LSHFT	AC <- AC ↑ index	16

Description:
The contents of the accumulator are shifted left or right by the amount specified by bits 8:15 of the instruction. Bits 8:15 are interpreted as a signed shift count. A positive number causes a left shift, a negative number causes a right shift. A shift count greater than 15 or less than -15 shall cause the accumulator to be cleared.

Load Message Pointer

LMP	MP <- (MP + Index*2)	1D

Description:
The current message pointer is replaced by the 32-bit word at the specified location in the message. The new message pointer is checked to insure that it is an even address within the message space. If the value loaded into the message pointer is zero, bits 0:1 of the channel status register shall be cleared, and the channel program shall halt. LMP can be used to chain messages.

Branch

BRIO	CHPC <- CHPC + index*2	18
BNEQIO	IF AC NEQ 0 THEN CHPC <- CHPC + index*2	19
BLSSIO	IF AC LSS 0 THEN CHPC <- CHPC + index*2	1A

Description:
Bits 8:15 of the instruction are interpreted as a signed halfword displacement from the next instruction address. The displacement is added to the incremented channel program counter if the specified condition is met. BRIO always branches. BNEQIO branches if the accumulator is nonzero. BLSSIO branches if the most significant bit of the accumulator is nonzero.

Case

CASEIO	1B, list count, {list of 16-bit displacements}

Description:
The contents of the accumulator is used as a selector for the case. Bits 8:15 of the instruction contains the unsigned number of 16-bit displacements in the following list. The value in the AC is used as a zero-based index into the array of 16-bit literals that follow the instruction. If the selector is less than the list count then the selected signed displacement times 2 is added to the address of the first halfword in the list to form the next instruction address. Otherwise the CHPC is loaded with the address of the first halfword following the list (no branch is taken).

Bit Case

BCASE 1E, list count, {list of 16-bit displacements}

Description:
Bits 8:15 of the instruction contains the unsigned number of 16-bit displacements in the following list (the list count). The bit position number of the first set bit from the left in the accumulator is determined. If this position number is less than the list count then the position number selects one of the 16-bit displacements following the instruction. The selected signed displacement times 2 is added to the address of the first halfword in the list to form the next instruction address. Otherwise the CHPC is loaded with the address of the first halfword following the list (no branch is taken). If no bits were set in the accumulator no branch is taken.

Interrupt

INT 01

Description:
A processor interrupt is generated. The priority of this interrupt is specified by bits 11:15 of the instruction. If this priority exceeds the maximum defined in the channel configuration register an **Interrupt.Priority** fault shall occur. Otherwise, a processor interrupt of the specified priority shall be requested. The contents of the accumulator shall specify the interrupt code for this request.

A sequence of INT instructions shall cause the specified processor interrupts to be requested in the order of execution of the INT instructions. Each request shall be held pending until it is accepted by the processor. Execution of an INT instruction may cause the channel program to pause until the requested interrupt can be posted.

Halt

HALT 00

Description:
Bits 0:1 of the channel status register are cleared and the channel program is halted.

13.8.4. Channel Specific Instructions. The peculiarities of each interface require that some interface specific facilities be provided by the individual IOCs. This section describes those instructions whose use is restricted to a particular interface type. Fetch of these opcodes by interface types for which they are not defined shall cause an **Illegal.Opcode** fault.

13.8.4.1. Parallel Point to Point Interface. The parallel point to point interface can be configured in one of three modes: computer, peripheral or undefined. In computer mode, the interface initiates all command and data transactions. In peripheral mode, the interface responds to externally requested transfers. In undefined mode, all interface drivers and receivers except reset are in the "off" state. The mode of operation of the interface is controlled by a mode field in the channel configuration register.

In peripheral mode the interface shall respond to external transfer requests only during the execution of one of the transfer instructions described above. The instruction specifies the expected

direction and type (command/status or data) of transfer. Actual movement of data across the interface is controlled by the external device. If the direction and type of the external request does not match those specified by the instruction, an *Out of Sync* error shall be indicated in the channel program status word and the transfer instruction shall terminate. In the absence of transfer errors the instruction shall terminate when the specified number of units has been transferred.

The interface provides the ability for an external device to request a processor interrupt. Bits 17:21 of the channel configuration register shall specify the priority of this request. The entry address shall be specified by the contents of the device interrupt vector. See section 13.10. The device interrupt request shall be acknowledged when the processor interrupt is accepted.

An interface control instruction is provided to manipulate the channel specific control functions available. Some of these functions are restricted to a particular operating mode, as described below. An attempt to execute one of these functions in an improper mode shall cause an **Illegal.Operation** fault.

Interface Control

CONTROL 1F

Description:

Bits 8:15 of the instruction specify the control function to be performed. The defined functions are:

0000000X	Interrupt. An interrupt transaction is initiated on the interface. Bit 15 specifies the type of interrupt: 0 specifies normal end; 1 specifies unusual end. This function is restricted to peripheral mode.
00000010	Reset. A reset signal is generated.
10XXXXXX	Clear discrete. Bits 10:14 are reserved. Bit 15 is associated with the IPL Ready signal. If bit 15 is set, IPL ready is cleared. If bit 15 is clear, this is a no-op. This function is restricted to peripheral mode.
11XXXXXX	Set discrete. Bits 10:14 are reserved. Bit 15 controls IPL ready. If bit 15 is set, IPL Ready is set. If bit 15 is clear, this is a no-op. This function is restricted to peripheral mode.

Specification of any other code in bits 8:15 shall cause an **Illegal.Operation** fault.

13.8.4.2. Serial Point to Point Interface. Each SPP interface has an independent input and output channel. An interface control instruction is provided to operate the control signals that can be generated by the output channel. Use of this instruction on the input channel shall cause an **Illegal.Operation** fault.

Interface Control

CONTROL 1F

Description:
Bits 8:15 of the instruction specify the control function to be performed. The defined functions are:

00000000	Break. A break is sent on the output channel.
10XXXXXX	Clear discrete. Bits 10:12 are reserved. Bits 13:15 are associated with the control 1, control 2, and IPL Ready discrete signals, respectively. A one bit in any of these positions causes the corresponding signal to be cleared. A zero leaves the corresponding signal unchanged.
11XXXXXX	Set discrete. Bits 10:12 are reserved. Bits 13:15 are associated with the control 1, control 2, and IPL Ready discrete signals, respectively. A one bit in any of these positions causes the corresponding signal to be set. A zero leaves the corresponding signal unchanged.

Specification of any other code in bits 8:15 shall cause an **Illegal.Operation fault.**

13.8.4.3. MIL-STD-1553 Serial Interface. This serial interface shall operate in accordance with the latest release of MIL-STD-1553B. The MIL-STD-1553B interface can operate in two modes: bus controller (BC) or remote terminal (RT). The current mode of operation is specified by a mode field in the channel dependent section of the Channel Configuration Register.

13.8.4.3.1. MIL-STD-1553 RT Mode Specific Instructions. When the channel configuration register specifies RT mode operation, no instructions shall be executed. See section 13.9 for details.

13.8.4.3.2. MIL-STD-1553 BC Mode Specific Instructions. When the channel configuration register specifies bus controller mode, the standard IOC instruction set, as described above, is supported. However, the function of the transfer instructions described in section 13.8.2 is redefined slightly for compatibility with the MIL-STD-1553B interface. At the beginning of these instructions, the accumulator is assumed to contain a MIL-STD-1553B command word.

The standard READ, READ STATUS, WRITE, WRITE COMMAND classes of instructions perform the following functions:

READ	The contents of the accumulator are presumed to be a transfer command (no check for mode commands is made) with the direction specified by the T/R bit in the accumulator.
WRITE	The contents of the accumulator are presumed to be a transfer command (no check for mode commands is made) with the T/R bit forced to zero.
WRITE COMMAND	The contents of the accumulator are assumed to be a mode command with no

associated data word.

READ STATUS — The contents of the accumulator are assumed to be a mode command with one associated data word. The direction of data transfer is indicated by the T/R bit in the accumulator.

For transfer commands, the command word in the accumulator contains a word count field as well as an RT address, and transmit/receive bit. The READ and RDTMSG instructions interpret this word to determine the function to be performed. In particular, this means that a READ instruction can in fact cause a write operation, depending upon the command word supplied. The WRITE, WRFMSG, and WRLIT instructions operate in the same manner, except that they force the transmit/receive bit of the command word (bit 5 of the accumulator) to zero. The data required for the I/O transaction is obtained from the buffer specified by the instruction definition. When the instruction terminates, the status word received on the MIL-STD-1553 bus is placed in the accumulator. The Channel Program Status reflects the outcome of the transfer. A zero (all bits clear) in the Channel Program Status register indicates a correct transfer.

In multi-redundant bus systems, the bus select register is loaded by the CPU and indicates the bus to be used for transfers while in BC mode.

In addition to the standard transfer and control instructions, two additional instructions are provided for the MIL-STD-1553 interface. An interface control instruction and a special transfer instruction to initiate RT to RT transfers are described below.

Interface Control

CONTROL 1F

Description:
Bits 8:15 of the instruction specify the control function to be performed. The defined functions are:

00000000 — Specify transmission bus. The bus on which the bus controller communicates is specified by the contents of the accumulator. This value is placed in the bus select register. If the contents of the accumulator specify a bus that is not implemented, an **Illegal.Operation** fault shall result..

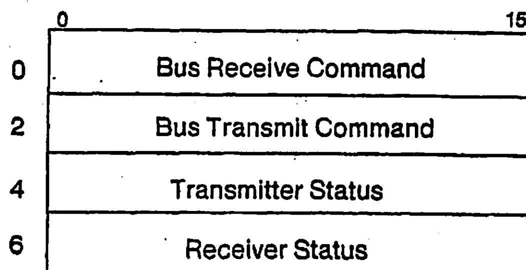
Specification of any other code in bits 8:15 shall cause an **Illegal.Operation** fault.

Initiate RT to RT transfer

RT2RT 20

Description:
Bits 8:15 of the instruction specify an unsigned displacement (in halfwords) from the message pointer contents. A four halfword operand block is located at this position in the message. The function of each halfword is:

- A 16-bit bus receive command to be sent to the destination RT.
- A 16-bit bus transmit command to be sent to the source RT.
- The 16-bit status word generated by the source RT is stored in this location at the completion of the transfer.
- The 16-bit status word generated by the destination RT is stored in this location at the completion of the transfer.



13.9. MIL-STD-1553B Remote Terminal Mode Operation. If the channel configuration register of a MIL-STD-1553B channel specifies remote terminal mode operation, execution of the channel program shall be disabled. The RT address of the IOC shall be specified by bits 0:4 of the channel configuration register. Operation of the channel shall be initiated or terminated by setting or clearing bit 0 of the channel status register, respectively. The following registers in the processor I/O space are interpreted to control the response to bus commands.

13.9.1. Message Pointer Register. The contents of the message pointer register shall be interpreted as the virtual address of an array of 32-bit buffer pointers. The array contains an input and an output entry for each subaddress. Accesses to this array shall be restricted as accesses to the message segment. An addressing error fault may occur when RT mode operation is initiated if all 64 entries of the array do not lie within the message segment.

13.9.2. Status Word. The status word is a 16-bit register that is loaded by the CPU. The contents of this register are used to determine the software controlled bits in the status reply to a MIL-STD-1553 bus command.

- The contents of bits 7, 13, and 15 are OR'ed with information from the IOC to produce the service request, subsystem flag, and terminal flag status bits, respectively.
- Bit 14 defines the dynamic bus control acceptance bit used to respond to a dynamic bus

control mode command.

- Bits 0:4 and 8:10 are reserved.

The remaining bits of the status reply are determined by the channel in accordance with MIL-STD-1553B.

13.9.3. Vector Word. The vector word is a 16-bit register whose contents are defined by the CPU. The contents of this register are transmitted in response to a Transmit Vector Word mode command. A non-zero value also causes the service request bit in the MIL-STD-1553 bus status word to be set.

13.9.4. Transfer Commands. When a transfer command is received on the bus in RT mode, the following actions shall occur in the order shown:

1. Bits 5:10 of the command word (the Transmit/Receive and Subaddress fields) shall be used as an index into the buffer address array. The base address of this array is defined by the contents of the message pointer register. The index times 4 plus the base address selects a 32-bit word from this array. Bit 30 of this selected word shall be set to zero.

2. Bits 30:31 of the value obtained from the selected word shall be forced to zero. The result shall be interpreted as the virtual address of a data buffer in the data segment. The transfer specified by the bus command shall be performed.

3. If the requested transfer is correctly completed, bit 30 of the buffer pointer selected in step 1 shall be set. If an error was detected in data received, this bit shall not be altered. If a data transmission cannot be properly performed, this bit shall not be altered.

4. Bit 31 of the selected word shall be examined. If this bit is set, and no errors were detected in the transfer, the IOC shall request an RT mode interrupt as described in section 13.9.6. Note: bit 31 is a software switch that causes an interrupt when a transfer is performed using this buffer.

13.9.5. Mode Commands. The optional RT mode commands shall perform the functions described in MIL-STD-1553B. The mode commands defined by MIL-STD-1553B shall be separated by the IOC into two classes. Mode commands in the first class shall be processed without notification of the CPU. The commands in this class are:

- Transmit Status Word
- Transmit Vector Word
- Transmit Last Command
- Transmit BIT Word

The mode commands in the second class shall perform the function described in MIL-STD-1553B and then cause the IOC to request an RT mode interrupt as described in section 13.9.6. The mode commands in this class are:

- Dynamic Bus Control
- Synchronize Without Data

- Initiate Self Test
- Transmitter Shutdown
- Override Transmitter Shutdown
- Inhibit Terminal Flag Bit
- Override Inhibit Terminal Flag Bit
- Reset Remote Terminal
- Synchronize With Data
- Selected Transmitter Shutdown
- Override Selected Transmitter Shutdown

In addition, receipt of one of the last three mode commands shall cause the received 16-bit data word to be placed in the channel program status register.

13.9.6. RT Mode Interrupts. Interrupts specified as RT mode interrupts shall be vectored through the IOC program interrupt vector. The priority of these interrupts shall be specified by bits 22:26 of the channel configuration register. The mechanism of these interrupts shall be the same as that described in section 13.7.

The interrupt code for an RT mode interrupt shall be a 16-bit value. This value shall be the 16-bit bus command that initiated the interrupt operation.

Interrupts shall be requested in the order in which the commands that generated them are received. If an interrupt cannot be posted due to a previously pending interrupt, the IOC shall respond with busy to the bus request.

RT mode operation can also cause **Message.Alignment**, **Addressing.Error**, and **IOC.Active** faults. These shall operate as previously defined in section 13.7.1.

13.10. Interrupt Vector Assignments. Each IOC shall be assigned a four word block of vectors in the device interrupt vector area (refer to section 15). The function of some of these vectors is interface specific. The table below lists the relative position and function of the vectors for each interface type described within this document.

Interrupt Vector Assignments

Location	Parallel	MIL-STD-1553	SPP
001000X0	Program	Program	Input Program
001000X4	Error	Error	Input Error
001000X8	Device	Reserved	Output Program
001000XC	Reserved	Reserved	Output Error

13.11. IOC Control Register Assignments. Each IOC program is assigned a 512 byte block of control registers in the I/O address space. This block is divided into two 256-byte halves. Control registers located in the low half are used for functions that may impact the security of the system. The assigned register offsets in this area are:

Address	Function
000XXY00 : 000XXY03	Channel Configuration Register
000XXY04 : 000XXY0F	Implementation Reserved
000XXY10 : 000XXY1F	Program Segment Specifying Location
000XXY20 : 000XXY2F	Message Segment Specifying Location
000XXY30 : 000XXY3F	Data Segment Specifying Location
000XXY40 : 000XXYDF	Implementation Reserved
000XXYE0 : 000XXYFF	Reserved

Where "X" is any Hex digit and "Y" is an EVEN Hex digit.

The upper half of the IOC control registers is used for those control registers that are available to an untrusted user. Implementation dependent registers in this area shall not perform any functions that may cause memory accesses outside the defined IOC segments or that may generate interrupts using vectors or priorities other than those defined above.

Address	Function
000XXY00 : 000XXY01	Channel Status
000XXY02 : 000XXY03	Channel Program Status
000XXY04 : 000XXY07	Channel Program Counter
000XXY08 : 000XXY0B	Message Pointer
000XXY0C : 000XXY0D	Status Word (MIL-STD-1553)
000XXY0E : 000XXY0F	Vector Word (MIL-STD-1553)
000XXY10 : 000XXYDF	Implementation Reserved
000XXYE0 : 000XXYE1	Bus Select (MIL-STD-1553)
000XXYE2 : 000XXYFF	Reserved

Where "X" is any Hex digit and "Y" is an ODD Hex digit.

14. Timer Support

The Nebula architecture provides two types of timer support: time of day timer and interval timers. These timers are accessible as registers in the I/O address space. Access to these registers is controlled by the memory management system.

14.1. Time of Day. There shall be a 32-bit register called the time of day clock. This register shall be a counter that is incremented every 10 ms. The contents of this register can be read or written by the software.

14.2. Interval Timers. There shall be four 32-bit registers called interval timers. These registers shall be counters that are decremented on the occurrence of particular events. These registers may be read or written by the software. Associated with each interval timer is a one byte field of the 32-bit Timer Control Register. The format of this register is shown in figure 14-1. Each byte of this register contains two fields: a mode field and a priority field. The mode field shall determine which events shall cause the corresponding interval timer to be decremented. The defined values of the mode field are:

000 Disabled. The corresponding interval timer shall not be decremented.

001 Time Base. The corresponding interval timer shall be decremented every 1 microsecond.

010 Instruction. The corresponding interval timer shall be decremented by each execution of an instruction while bit 0 (Kernel/Task) of the PSW is set. The timer shall also be decremented on each check for pending interrupts on interruptible instructions (such as MOVBK) while bit 0 of the PSW is set.

All other values of the mode field are reserved. Each time an interval timer is decremented to zero an interrupt shall be requested. The timer shall continue to be decremented as specified by the mode field. The priority of the interrupt request shall be determined by the priority field in the Timer Control Register corresponding to the interval timer. The entry address of the procedure to be invoked shall be determined by the contents of the interval timer's interrupt vector. See section 15. This interrupt shall have no parameters. Timer interrupts from each interval timer will be queued one deep.

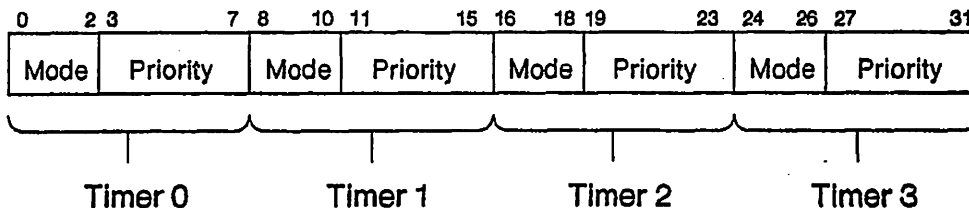


Figure 14-1: Timer Control Register

15. Assigned Physical Addresses

Access to the assigned physical addresses should be controlled by means of the memory management system.

15.1. Memory Space Assignments. The first 1024 bytes of physical memory are allocated for special uses by the architecture. The following table gives a list of interrupt and trap vector locations defined by the architecture.

VECTOR NAME	ADDRESS (in hex)
Reserved	00100000
Software interrupt	00100004
BIT trap (# 1)	00100008
Memory management errors	0010000C
Memory system errors (Soft)	00100010
Memory system errors (Hard)	00100014
Power failure	00100018
Power restore	0010001C
Kernel save area pointer	00100020
Supervisor Exception Handler	00100024
Privileged instruction trap	00100028
BIT trap (# 2)	0010002C
Timer 0 interrupt	00100030
Timer 1 interrupt	00100034
Timer 2 interrupt	00100038
Timer 3 interrupt	0010003C
Reset and IPL entry	00100040
Reset/IPL save area pointer	00100044
Reserved	00100048 to 0010005C
Reserved for Device Vectors	00100060 to 001000FC

The addresses from 00100100 to 00100400 (hex) are available for assignment to hardware specific uses such as additional device vectors, IPL and BIT functions. Addresses within this area are assigned by the hardware specification. Software that uses these assigned locations should be aware of the implications of the hardware use of these locations. In particular, each type of assignment has differing properties:

Device Vectors — These are accessed during interrupt processing.

IPL areas — During the IPL process these areas may be modified in an arbitrary manner. The IPL sequence cannot reliably initialize these areas. Once IPL is complete, these areas shall function as ordinary memory.

BIT areas — Memory assigned to the built-in test function may be modified at any time during execution.

Other uses — The properties of other assigned areas will be defined by the hardware specification.

Unassigned addresses within this area function as ordinary memory. However, if software is to be transportable between implementations with differing assignments, use of this area by the software should be avoided.

15.2. I/O Space Assignments. The top 2K bytes in the I/O space are dedicated to processor control registers. The following table lists the registers assigned in this area.

REGISTER NAME	ADDRESS (in hex)
Kernel Context Pointer	000FF800
Task Context Pointer	000FF804
Software interrupt request	000FF808
OPEX limit	000FF80C
OPEX table address	000FF810
SVC limit	000FF814
SVC table address	000FF818
Auxiliary Status Register	000FF81C
User Map Pointer	000FF820
Supervisor Map Pointer	000FF824
Timer control	000FF828
Interval timer 0	000FF82C
Interval timer 1	000FF830
Interval timer 2	000FF834
Interval timer 3	000FF838
Time of day clock	000FF83C
Reserved	000FF840 to 000FFFFF

Accesses to ALL registers in I/O space are restricted. First, accesses may not cross register boundaries. Second, accesses within registers must be made on item boundaries. In other words, halfword accesses must be on halfword boundaries within word or doubleword registers. Word accesses must be on word boundaries within doubleword registers. Bit field accesses may start on any bit as long as the field does not cross a register boundary. Accesses that do not meet these restrictions shall produce one of two outcomes; either the access shall complete as requested or the access shall produce a hard memory trap. The choice is implementation dependent.

The Kernel Context Pointer, the Task Context Pointer, the User map pointer, and the Supervisor Map Pointer are special registers that determine the control flow of the computer. As such, reading them through the I/O space may yield old or undefined values. Writing these registers through the I/O space will produce implementation dependent results.

16. Conceptual Model of Instruction Execution

The conceptual model of instruction execution provides information needed to coordinate the interaction between multiple processors and IOCs. It also details the ordering of traps and exceptions and their interaction with the normal instruction stream. For most purposes, the serial model described below is a sufficient description of instruction execution. The remainder of this section describes interactions that should be considered when writing code to synchronize multiple processors and IOCs.

16.1. The Serial Model. Conceptually, the CPU processes instructions one at a time, with the execution of one instruction preceding the execution of the instruction that follows in the instruction stream, and the execution of the instruction specified by a successful jump (including exception handler jumps) or branch follows the execution of the jump or branch instruction. Similarly, an interruption takes place between executions of instructions (except when the instruction is noted to be interruptable).

The sequence of events implied by the processing just described is sometimes called the *conceptual* sequence or *conceptual* order. Even though the actual processing may be quite different due to variations in physical storage width, instruction pre-fetch, pipelining, and so on, the conceptual order of execution is maintained, *as observed by the CPU itself*.

Figure 16-1 is a pictorial representation of the conceptual model. Each box in the figure has one or more arrows that point to it. The action or actions listed within a box take place (conceptually) after the action or actions of all boxes with arrows that point to the box.

16.2. Effects of Parallelism. In very simple machines in which operations are not overlapped, the conceptual order and the actual order are essentially the same. However, in more complex machines, overlapped operation, buffering of operands and results, and execution times that are comparable to the propagation delays between units can cause the actual order of execution to differ considerably from the conceptual order. In these machines, special circuitry is employed to detect dependencies between operations and ensure that the results obtained are those that would have been obtained if the operations had been performed in the conceptual order. However, as observed by IOCs and other CPUs, the sequence may appear to differ from the conceptual order.

When only a single CPU is involved, it can normally be assumed that the execution of each non-interruptable instruction occurs as an indivisible event. However, in actual operation, the execution of an instruction may consist of a series of discrete steps. Depending on the instruction, operands may be fetched and stored in a piecemeal fashion, and some delay may occur between fetching and storing a result. As a consequence, another CPU or IOC may be able to observe intermediate, or partially completed, results. When the program on one CPU interacts with a program on another CPU or IOC, the programs have to take into consideration that a single operation may consist of a series of accesses, and that the conceptual and actual sequences of these accesses may differ. If another CPU or IOC is to depend on the results of instruction execution up to a certain point, it is necessary to perform a serialization function at that point.

Traps for hard memory errors and soft memory errors may occur in a completely asynchronous fashion. The memory byte(s) in error may be completely unrelated to the instruction being executed when the error is detected. Ideally, the implementation will be so arranged as to provide a high probability that when an error is detected it will be related to the currently executing task. It is intended that the handler for a hard memory error should terminate the executing task, and that the handler for a soft memory error should resume the currently executing task in a completely transparent manner, without trying to draw conclusions about the currently executing instruction. In the remainder of this discussion, the word "traps" will refer to all traps *except* those for hard and soft memory errors.

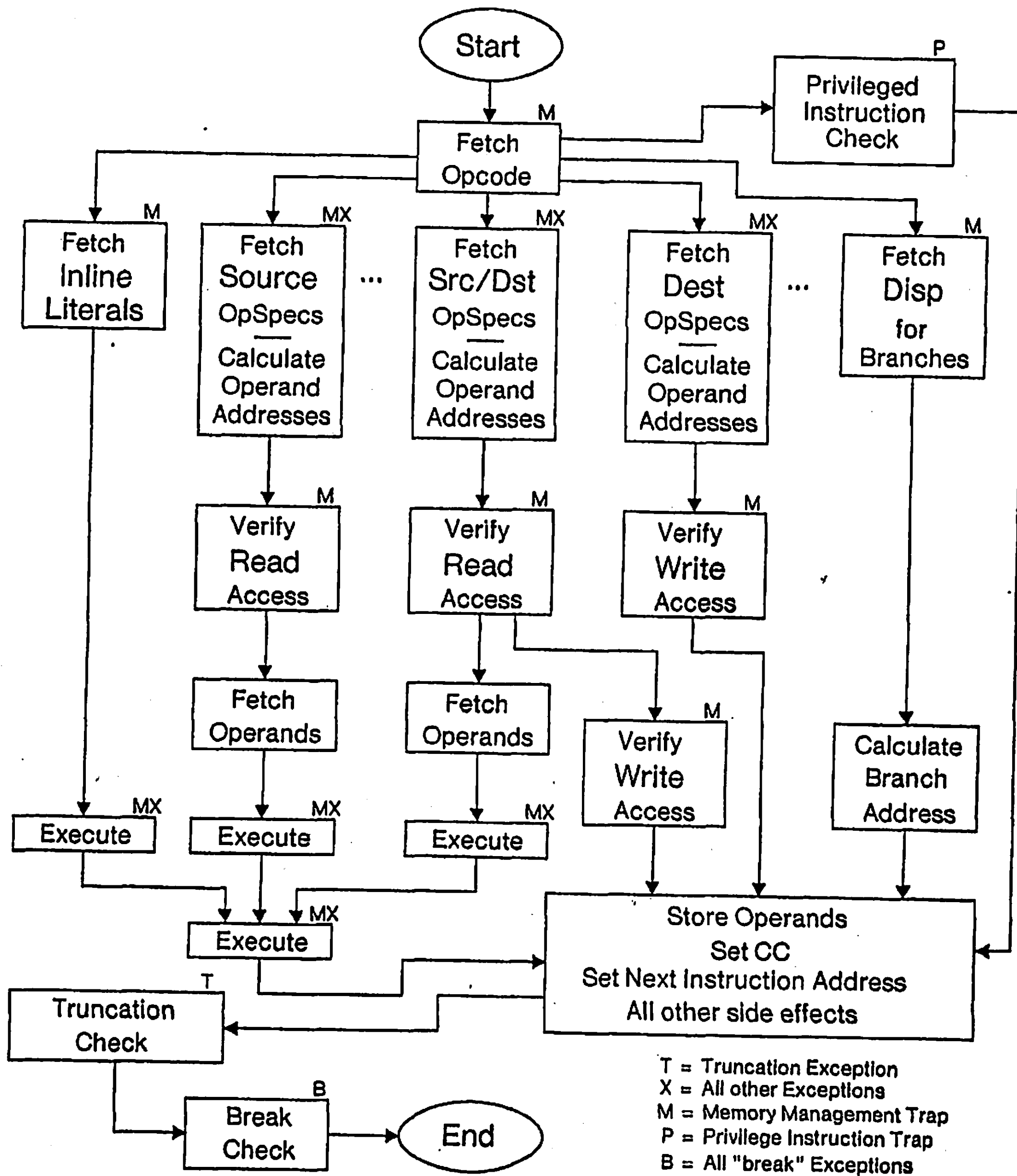


Figure 16-1: Conceptual model of instruction execution

The processing of an instruction may be divided into several stages: instruction fetch, operand address calculation, operand fetch, instruction execution, and operand storage. The following general rules hold for instruction execution:

- Within an instruction, all operand fetches occur before any operands are stored. Instructions with overlapping operands work as though the source operands are all fetched before the destination operands are stored. For instructions that store more than one operand, the possible effects of overlapping destination operands are specified on case-by-case basis (see the individual instruction descriptions). The consequences of overlap of the memory areas specified by the operands of string instructions are described in the individual string instruction descriptions.

- If an operand specifier is used to specify an operand that serves as both a source and a destination, it is as if the operand were specified separately as a source and as a destination; conceptually, it must be fetched before any operand stores occur, and stored after all operand fetches have occurred.

- When an exception or trap other than the **Truncate** exception causes premature instruction termination, the operands and condition codes are unaffected.

- If a branch instruction causes an exception that normally causes the Supervisor Exception Handler to receive the address of the following instruction as its third parameter, then the instruction must make its usual determination as to whether or not to take the branch in order to pass the correct next instruction address (that of the following instruction or that of the branch target).

- If execution of an instruction could cause more than one exception or trap to occur, then exactly one such exception or trap is chosen to occur in an implementation-dependent manner. For example, consider a three-operand division instruction where the operand specifier for the dividend is malformed, using a compound addressing mode within another compound addressing mode; the divisor is specified as a literal with value zero; and the quotient is to be stored into a location with read-only access. This instruction has the potential to signal an **Illegal.Mode** exception, an **Illegal.Divisor** exception, or a memory management trap. Any one of these may be signalled in a given situation; it is possible that the same implementation might choose to signal a different one under different circumstances. (The justification for this ambiguity is that one exception is as good an another, and traps are meant to result either in task termination or transparent task resumption.)

- A successfully completed instruction shall fetch all source operands and store all destination operands unless otherwise specified. Failure to complete an instruction as a result of an exception or trap may result in failure to access some or all of its operands.

16.3. Instruction Fetch. Instruction fetching consists of fetching the bytes of an instruction, including the opcode and all operand specifiers. The bytes of an instruction may be fetched piecemeal and are not necessarily accessed in order of increasing byte addresses. It is permissible to fetch an instruction byte more than once.

As observed by another CPU or an IOC, instructions are not necessarily fetched in the order in which they are conceptually executed and are not necessarily fetched each time they are executed. For example, storing by another CPU or IOC does not necessarily modify the copies of pre-fetched

instructions. Also, the fetching of an instruction may precede the operand fetch or storage for an instruction that is conceptually earlier.

However, as observed by the executing CPU, it must be as if one instruction is not fetched until the preceding one has been completed. For example, if fetching the next instruction would cause a memory-management trap, that trap may not be taken until the current instruction has successfully completed execution, and any trap or exception encountered by the current instruction has priority.

16.4. Operand Address Calculation. Operand addresses must conceptually be calculated as if all previous instructions had first stored their results. Operand address calculations need not be performed in left-to-right order; they may be performed in some other order, or in parallel.

16.5. Operand Fetch. Unless otherwise specified for an individual instruction, during the execution of an instruction, all or part of the storage operands for that instruction may be fetched, intermediate results may be maintained for subsequent modification, and final results may be temporarily held prior to placing them in main memory. (Here "main memory" refers to memory as seen by another CPU or IOC; it may, for example, include caches that are visible to such other CPU or IOC.) Stores by another CPU or IOC do not necessarily affect these intermediate results.

All bits within a single byte of a fetch are accessed concurrently. When an operand consists of more than one byte, the bytes may be fetched piecemeal and in any order (possibly concurrently) unless otherwise specified.

16.6. Instruction Execution. Individual instructions, such as the string instructions, may explicitly perform operand fetches and stores during their execution. Such memory accesses may cause memory management traps.

16.7. Operand Storage. All bits within a single byte of a store are accessed concurrently. When an operand consists of more than one byte, the bytes may be stored piecemeal and in any order (possibly concurrently) unless otherwise specified.

Conceptually, the results of an instruction are stored after the results of all previous instructions are stored, and before the operands of any subsequent instructions are fetched or stored. The actual sequence of fetches and stores may differ from the conceptual sequence; however, the results of each instruction executed by a single CPU are guaranteed to be identical to those obtained by the conceptually ordered execution.

There is no defined limit on the length of time that a CPU may delay before storing a result. Storage of all pending results may be forced by the execution of a serialization function, described below.

16.8. Memory Accesses. A processor is permitted to fetch and retain bytes from main memory that are extraneous to the conceptual course of execution, as long as such fetching is transparent to the executing program (for example, such extraneous fetching must not cause an extraneous memory management trap). However, performing a serialization function will force such pre-fetched bytes to be discarded; they must be re-fetched from main memory if they are subsequently required.

A processor is also permitted to perform extraneous re-storage of such extraneously fetched bytes under the following circumstances: if the conceptual course of execution requires storage of a byte, then the processor is permitted to physically store back into main memory any or all bytes of the doubleword-address-aligned doubleword containing the byte; the processor must of course have previously fetched any bytes not conceptually modified, so that they may be stored back with the same value.

These provisions effectively imply that, as a software convention, two processors should not attempt to access the same doubleword-address-aligned doubleword of physical memory without first performing explicit serialization and synchronization operations. (For example, suppose that two

processors A and B were to attempt to use two adjacent words, say at addresses 1000004 and 1000006. It would be legitimate for processor A to process an incrementation of the word at address 1000004 by fetching the entire doubleword, incrementing the first word, and storing the entire doubleword back into main memory, without any kind of memory interlock. Any modification of the word at address 1000006 by processor B could be lost. However, processor A would not be permitted to store unnecessarily into address 1000008, and so that word could safely be used by processor B.)

16.9. Interruptible Instructions. Interruptible instructions follow a slightly different model. Conceptually, interruptible instructions can be considered to contain a set of interruption points within their execution. Interrupts may be handled either by waiting until the instruction execution reaches the next interruption point or by beginning interrupt execution immediately after resetting the instruction state to the previous interruption point. Traps are expected to reset the instruction to the previous interruption point and enter the trap mechanism. A trap that allows a return (using the RET instruction within the handler) to the instruction is expected to continue from the point of interruption.

16.10. Serialization. An instruction is said to perform a serialization when execution of that instruction causes all storage of results into memory by conceptually previous instructions to be completed, as seen by another CPU or IOC, and all modifications of memory by other CPUs and IOCs to be visible in fetches by subsequent instructions.

Nebula further divides serialization functions into two classes:

Instruction — An instruction serialization causes all *instruction* fetches that follow the serialization to reflect memory state that existed at the time of or subsequent to the serialization.

Data — A data serialization causes all memory operand fetches to reflect memory state that existed at the time of or subsequent to the serialization. Further, it guarantees that all stores to memory operands of conceptually previous instructions have been completed.

Note that the issue of consistency of the context stack between the CPU and memory is not addressed by either of these serializations. The context stack abstraction does not define a conceptual seriality of context accesses to main memory. Rather, consistency of the context stack is guaranteed by execution of the LTASK and STASK instructions. Refer to the instruction descriptions for details.

The following instructions perform serializations of the specified types:

- LTASK -- Instruction (task context only)
- STASK -- Data
- REPENT -- Data, and Instruction within the replaced segment
- CMPS -- Data
- STOBIT -- Data

16.11. IOC Serialization. An IOC instruction serialization occurs when an IOC is started. IOC operand and data fetches and stores occur serially.

17. Instruction Descriptions

The following sections contain descriptions of the instructions for the Nebula computer architecture. This section contains information about these descriptions.

17.1. General Information. Each instruction description has five parts. After the name of the instruction is an unlabeled part that shows the instruction assembler mnemonic, a short symbolic description of the instruction's action if possible, and an ordered list of the items (separated by commas) expected in the instruction stream. This ordered list contains the opcode (in hex), the symbols used in the instruction description for the operands, and/or any displacements the instruction expects. There are four other labeled parts. The *operand types* part describes the characteristics of the operands. The *condition codes* part shows the instruction's action on the condition code bits. The *program exceptions* part lists any exceptions or traps specific to that instruction. The *description* part contains an English-language verbal description of the action of the instruction. The verbal description and the symbolic description for each instruction are intended to complement each other. Omissions in one are resolved in the other. The verbal description has priority in resolving ambiguities. The order of operations between each "next" in the symbolic description is implementation dependent.

17.2. Operands. The *operand types* section lists the type and supported sizes for each symbolic operand. There are five possible types:

- signed integer
- unsigned integer
- logical
- floating point
- address operand

Additionally, a few instructions specify an **inline literal** as one of their operands. This literal is located at the specified point in the instruction stream and is interpreted differently from normal operand specifiers. An inline literal is taken immediately from the instruction stream and treated as an unsigned integer. It typically has no effect upon the condition code bits.

The following subsections provide details about the typical use of the operand types. Variations on the use described below will be detailed in the description of the instruction where the variation occurs.

17.2.1. Signed Integers. Signed integer source operands are fetched and sign extended internally to include enough bits so that all operations can be performed without any loss of information. The least significant portion of a signed integer's internal representation is stored in a signed integer destination operand. If the size of the destination operand is too small to correctly represent the internal representation of the signed integer, the truncation condition code (T) is set. Otherwise, the T bit is cleared. If arithmetic exceptions are enabled (EAE is set) and the T bit is set a **Truncation** exception is initiated at the end of the instruction. The negative condition code (N) and the zero condition code (Z) are set to reflect the value of the destination operand except where otherwise noted. The carry condition code (C) is set by add and subtract instructions where noted, and signifies a carry from the most significant bit of the internal result.

17.2.2. Unsigned Integers. Unsigned integer source operands are fetched and zero extended internally. Unsigned integer destination operands are stored from the least significant portion of the internal representation. If the destination operand cannot correctly represent the internal representation, the T bit is set. In particular, if the internal representation is negative, the T bit is set. The destination operand is treated as a signed integer for the purposes of setting the Z and N bits. The C bit is set by unsigned add and subtract by a carry from the internal result.

17.2.3. Logicals. Logical source operands are treated in the same manner as unsigned integer source operands. They are fetched and zero extended internally. Logical destination operands are stored from the least significant portion of the internal representation. The T bit is unaffected by the storage of logical operands. The destination operand is treated as a signed integer for the purposes of setting the Z and N bits. The C bit is unaffected by the storage of a logical operand.

17.2.4. Floating Point. Floating point operands are fetched and converted to an internal format. See section 30.3.

17.2.5. Address Operands. Address operands evaluate to a virtual address (see section 5.3). Address operands typically have no effect upon the condition code bits.

17.3. Symbols and Functions. To aid in the description of the instructions certain special symbols and functions are used. The following is a list of relational symbols and their meaning:

Symbol	Meaning
Eql	Equal to
Neq	Not equal to
Lss	Less than
Leq	Less than or equal to
Gtr	Greater than
Geq	Greater than or equal to

If a relational has a "u" appended to it (for example, Lssu) then the relation treats its operands as unsigned.

The conventional logical operators AND, OR, NOT, and XOR are used with their conventional meaning. Other mathematical symbols used in the descriptions retain their conventional meaning also.

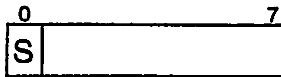
Most special functions are defined where they are used. Two that are not are **size** and **remainder.** The function size(x) (where x is an operand) refers to the size in bytes of the operand. The function remainder(B/A) refers to the remainder of the division of operand B by operand A. The result of remainder is defined as k such that $|k| < |A|$ and $kB \geq 0$ and $mA + k = B$ for some integer m.

The symbol NI is used in some of the symbolic descriptions of certain instructions. In this use, NI represents the address of the next instruction to be executed. The abbreviation "displ" is used for "displacement" in the symbolic descriptions.

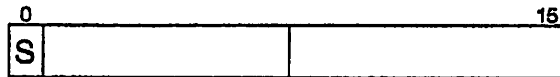
18. Integer Arithmetic

18.1. Integer Data Types. The Nebula instruction set architecture supports integer data types of 8-bit, 16-bit, 32-bit and 64-bit sizes. These are termed byte, halfword, word and doubleword integers, respectively. Integers can be treated as either signed or unsigned. Signed integers are represented in 2's complement form. When n bits are used in the representation, signed integers range from -2^{n-1} to $2^{n-1}-1$ and unsigned integers range from 0 to $2^{n}-1$.

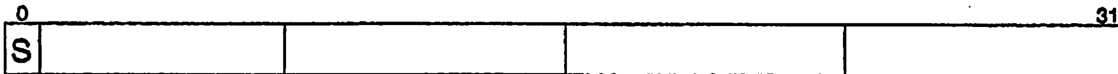
A **byte** is 8 contiguous bits starting on a byte boundary. The bits of decreasing significance are numbered from 0 to 7. A byte is specified by its address Addr. When the byte is interpreted as a signed integer, the most significant bit (bit 0) is the sign bit. The value of the signed integer is in the range -128 through 127.



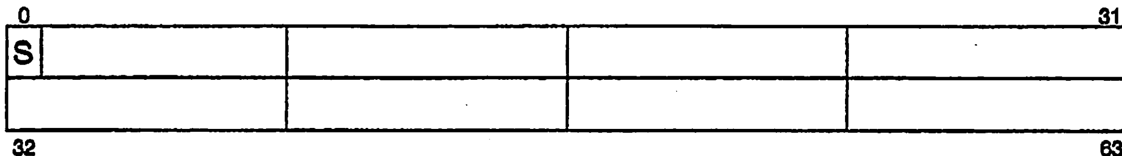
A **halfword** is 2 contiguous bytes starting on a byte boundary. The bits of decreasing significance are numbered from 0 to 15. A halfword is specified by its address Addr, the address of the byte containing bit 0. When the halfword is interpreted as a signed integer, the most significant bit (bit 0) is the sign bit. The value of the signed integer is in the range -32768 through 32767.



A **word** is 4 contiguous bytes starting on a byte boundary. The bits of decreasing significance are numbered from 0 to 31. A word is specified by its address Addr, the address of the byte containing bit 0. When the word is interpreted as a signed integer, the most significant bit (bit 0) is the sign bit. The value of the signed integer is in the range -2^{31} through $2^{31}-1$.



A **doubleword** is 8 contiguous bytes starting on a byte boundary. The bits of decreasing significance are numbered from 0 to 63. A doubleword is specified by its address Addr, the address of the byte containing bit 0. When the doubleword is interpreted as a signed integer, the most significant bit (bit 0) is the sign bit. The value of the signed integer is in the range -2^{63} through $2^{63}-1$. The doubleword integer data type is only partly supported by the Nebula instruction set architecture.



18.2. Integer Arithmetic Instructions. All operands involved in signed integer arithmetic instructions shall be sign extended internally to include enough bits so that all operations can be performed without any loss of information. If the instruction specifies a destination with fewer bits than the internal operation, the least significant bits of the result are stored in the destination. The truncate condition code (T) shall be set if the result cannot be correctly represented (as a signed integer) in the specified destination operand. The negative condition code (N) and the zero condition code (Z) are set to reflect the value of the destination operand except where otherwise noted. The carry condition code (C) is set by add and subtract instructions, and signifies carry from most significant (i.e. sign) bit of the internal result. If EAE is set (see section 6.14) and T is set at the completion of an instruction, a **Truncation** exception shall be raised.

All operands involved in unsigned integer arithmetic instructions shall be zero extended internally to 32 bits. If the instruction specifies a destination with fewer bits than the internal operation, the least significant bits of the result are stored in the destination. The truncate condition code (T) shall be set if the result cannot be correctly represented (as an unsigned integer) in the specified destination operand. The N and Z condition codes are set by treating the destination operand as a signed integer. The C condition code is set by a carry from the 32-bit internal result. If EAE is set and T is set at the completion of the instruction, a truncation exception shall be raised.

Integer Addition

ADD	R <- A + B	30, A, B, R
	R <- R + A	31, A, R

Operand types:
A, B, R   8-bit, 16-bit, 32-bit signed integer

Condition codes:
Z <- R Eql 0
N <- R Lss 0
T <- Set if the result is not representable in R, otherwise cleared
C <- Carry generated during 32-bit internal addition

Program exceptions:
Truncation

Description:
In the 3 operand format, the first operand is added to the second operand and the third operand is replaced by the sum. In the 2 operand format, the first operand is added to the second operand and the second operand is replaced by the sum.

Integer Subtraction

SUB	R <- B + not(A) + 1	32, A, B, R
	R <- R + not(A) + 1	33, A, R

Operand types:
A, B, R   8-bit, 16-bit, 32-bit signed integer

Condition codes:
Z <- R Eql 0
N <- R Lss 0
T <- Set if the result is not representable in R, otherwise cleared
C <- Carry generated during 32-bit internal addition

Program exceptions:
Truncation

Description:
In the 3 operand format, the first operand is subtracted from the second operand and the third operand is replaced by the difference. In the 2 operand format, the first operand is subtracted from the second operand is replaced by the difference.

NOTE: Carry is NOT borrow. The condition codes are not set in the same manner as CMP.

Integer Multiplication

MUL	R <- A x B	34, A, B, R
	R <- R x A	35, A, R

Operand types:

A, B, R 8-bit, 16-bit, 32-bit signed integer

Condition codes:

Z <- R Eql 0
N <- R Lss 0
T <- Set if the result is not representable in R, otherwise cleared
C Unaffected

Program exceptions:

Truncation

Description:

In the 3 operand format, the first operand is multiplied by the second operand and the third operand is replaced by the product. In the 2 operand format, the first operand is multiplied by the second operand and the second operand is replaced by the product.

NOTE: If size(R) $\geq$ size(A) + size(B) then T cannot be set.

Integer Division

DIV	R <- B / A	36, A, B, R
	R <- R / A	37, A, R

Operand types:

A, B, R   8-bit, 16-bit, 32-bit signed integer

Condition codes:

Z <- R Eql 0
N <- R Lss 0
T <- Set if the result is not representable in R, otherwise cleared
C Unaffected

Program exceptions:

Truncation
Illegal.Divisor

Description:

In the 3 operand format, the second operand is divided by the first operand and the third operand is replaced by the quotient. In the 2 operand format, the second operand is divided by the first operand and the second operand is replaced by the quotient. An **Illegal.Divisor** exception shall be raised if A = 0. When this exception occurs the operands are unaffected.

The Identity:

$$(-B)/A = -(B/A) = B/(-A)$$

is satisfied.

Integer Negate

NEG	R <- not(A) + 1	3A, A, R
	R <- not(R) + 1	3B, R

Operand types:
A, R 8-bit, 16-bit, 32-bit signed integer

Condition codes:
Z <- R Eql 0
N <- R Lss 0
T <- Set if the result is not representable in R, otherwise cleared
C <- Carry generated by 0-A or R

Program exceptions:
Truncation

Description:
In the 2 operand format, the second operand is replaced by the negative (2's complement) value of the first operand. In the 1 operand format, the operand is replaced by the negative of the value.

Integer Modulus

MOD	R <- B MOD A	38, A, B, R

Operand types:
A, B, R 8-bit,16-bit,32-bit signed integers

Condition codes:
Z <- R Eql 0
N <- R Lss 0
T <- Set if the result is not representable in R, otherwise cleared
C Unaffected

Program exceptions:
Truncation
Illegal.Divisor

Description:
In MOD the third operand is replaced by B modulo A. If A neq 0, the result is that integer k between 0 (inclusive) and A (exclusive) such that mA + k = B for some integer m. If A = 0, then an **Illegal.Divisor** exception shall be initiated.

Remainder

REM	R <- remainder(B / A)	39, A, B, R

Operand types:

A, B, R	8-bit,16-bit,32-bit signed integers

Condition codes:

Z <- R Eql 0
N <- R Lss 0
T <- Set if the result is not representable in R, otherwise cleared
C Unaffected

Program exceptions:

Truncation
Illegal.Divisor

Description:

In REM the second operand is divided by the first operand and the third operand is replaced by the remainder. The result is defined as k such that $|k| < |A|$ and $kB \geq 0$ and $mA + k = B$ for some integer m. An **Illegal.Divisor** exception shall be raised if $A = 0$. When this exception occurs the operands are unaffected.

Extended Integer Multiplication

EMUL	R <- A x B	4E, A, B, R

Operand types:

A, B	8-bit, 16-bit, 32-bit signed integer
R	16-bit, 32-bit, 64-bit signed integers

Condition codes:

Z <- R Eql 0
N <- R Lss 0
T <- Set if the result is not representable in R, otherwise cleared
C Unaffected

Program exceptions:

Truncation

Description:

The first operand is multiplied by the second operand and the third operand is replaced by the product. Since the third operand can be specified as a 64-bit signed integer,the product of 32-bit signed integers can be stored without any loss of information.

NOTE: If $size(R) \geq size(A) + size(B)$ then T cannot be set.

Extended Integer Divide

EDIV	R2 <- B / A	4F, A, B, R1, R2
	R1 <- remainder(B / A)	

Operand types:

B, R2	16-bit, 32-bit, 64-bit signed integers
A, R1	8-bit, 16-bit, 32-bit signed integer

Condition codes:

Z <- R2 Eql 0
N <- R2 Lss 0
T <- Set if either the quotient is not representable in R2 or the
remainder is not representable in R1, otherwise cleared
C Unaffected

Program exceptions:

Truncation
Illegal.Divisor

Description:

The second operand is divided by the first operand, and the third and fourth operands are replaced by the remainder and quotient, respectively. Since the second and fourth operands can be specified as 64-bit signed integers, extended precision integer division can be carried out and the result stored without any loss of information. An **Illegal.Divisor** exception shall be raised if A = 0. When this exception occurs the operands are unaffected. If storage of either R1 or R2 is blocked by the memory management system, the storage is aborted and the operands are not affected. If R1 overlaps R2 the resulting values are undefined.

Multiply Fixed Point

MULFIX 50, A, B, S, R

R <- (|A x B| right shifted S places) x sign(A) x sign(B)
where sign(x) = 1 for $x \geq 0$, -1 for $x < 0$

Operand types:

A, B, R 8-bit, 16-bit, 32-bit signed integer with implied radix point
S 8-bit, 16-bit, 32-bit signed integer

Condition codes:

Z <- R Eql 0
N <- R Lss 0
T <- Set if the result is not representable in R, otherwise cleared
C Unaffected

Program exceptions:

Truncation

Description:

The first operand is multiplied by the second operand and the absolute value of the product is first shifted either |S| places to the right, if S is positive, or |S| places to the left, if S is negative, and then the fourth operand is replaced by the result.

Since A, B, and R are fixed point they can be represented as $A \times 2^{-X}$, $B \times 2^{-Y}$, and $R \times 2^{-Z}$ where

X = number of bits to the right of the radix point in A
Y = number of bits to the right of the radix point in B
Z = number of bits to the right of the radix point in R

Therefore $R = A \times B \times 2^{Z-Y-X}$ meaning that in order to have R scaled properly the product must be shifted (Z-X-Y) places. It follows that S = Z - X - Y. However, we define S = X + Y - Z (i.e. -S = Z - Y - X) so that short literal form may be used for most cases of S.

Divide Fixed Point

DIVFIX 51, A, B, S, R

R <- ((|B| left shifted S places) / |A|) × sign(A) × sign(B)
where sign(x) = 1 for $x \geq 0$, -1 for $x < 0$

Operand types:

A, B, R	8-bit, 16-bit, 32-bit signed integer with implied radix point
S	8-bit, 16-bit, 32-bit signed integer

Condition codes:

Z <- R Eql 0
N <- R Lss 0
T <- Set if the result is not representable in R, otherwise cleared
C Unaffected

Program exceptions:

Truncation
Illegal.Divisor

Description:

The absolute value of the second operand is first shifted either |S| places to the left, if S is positive, or |S| places to the right, if S is negative, and then divided by the absolute value of the first operand. Finally, the fourth operand is replaced by the result.

Since A, B, and R are fixed point they can be represented as $A \times 2^{-X}$, $B \times 2^{-Y}$, and $R \times 2^{-Z}$ where

X = number of bits to the right of the radix point in A
Y = number of bits to the right of the radix point in B
Z = number of bits to the right of the radix point in R

Therefore $R = (B \times 2^{Z+X-Y}) / A$ meaning that in order to have R scaled properly B must be shifted (Z + X - Y) places. It follows that $S = Z + X - Y$.

An **Illegal.Divisor** exception shall be raised if A = 0.

Increment

INC	R <- R + 1	55, R
INC2	R <- R + 2	52, R
INC4	R <- R + 4	53, R
INC8	R <- R + 8	54, R

Operand types:
R 8-bit, 16-bit, 32-bit signed integer

Condition codes:
Z <- R Eql 0
N <- R Lss 0
T <- Set if the result is not representable in R, otherwise cleared
C <- Carry generated during 32-bit internal addition

Program exceptions:
Truncation

Description:
In INC one is added to the operand and the operand is replaced by the sum. In INC2 two is added to the operand and the operand is replaced by the sum. In INC4 four is added to the operand and the operand is replaced by the sum. In INC8 eight is added to the operand and the operand is replaced by the sum.

Decrement

DEC	R <- R - 1	60, R

Operand types:
R 8-bit, 16-bit, 32-bit signed integer

Condition codes:
Z <- R Eql 0
N <- R Lss 0
T <- Set if the result is not representable in R, otherwise cleared
C <- Carry generated during 32-bit internal addition

Program exceptions:
Truncation

Description:
One is subtracted from the operand and the operand is replaced by the difference.

NOTE: Carry is NOT borrow.

Add with Carry

ADDC R <- A + B + C 3E, A, B, R

Operand types:
A, B, R 8-bit, 16-bit, 32-bit signed integer

Condition codes:
Z <- R Eql 0
N <- R Lss 0
T <- Set if the result is not representable in R, otherwise cleared
C <- Carry generated during 32-bit internal addition

Program exceptions:
Truncation

Description:
The first and second operands and the contents of the condition code bit C are added, and the third operand is replaced by the sum.

Subtract with Carry

SUBC R <- B + not(A) + C 3F, A, B, R

Operand types:
A, B, R 8-bit, 16-bit, 32-bit signed integer

Condition codes:
Z <- R Eql 0
N <- R Lss 0
T <- Set if the result is not representable in R, otherwise cleared
C <- Carry generated during 32-bit internal addition

Program exceptions:
Truncation

Description:
The 1's complement of the first operand, the second operand and the contents of the condition code bit C are added and the third operand is replaced by the sum.

Absolute Value

ABS   R <- |A| 3C, A, R

Operand types:

A, R   8-bit, 16-bit, 32-bit signed integer

Condition codes:

Z <- R Eql 0
N <- 0
T <- Set if the result is not representable in R, otherwise cleared
C Unaffected

Program exceptions:

Truncation

Description:

The second operand is replaced by the absolute value of the first operand.

Unsigned Addition

ADDU   R <- A + B   56, A, B, R

Operand types:

A, B, R   8-bit, 16-bit, 32-bit unsigned integer

Condition codes:

Z <- R Eql 0
N <- R Lss 0
T <- Set if the result is not representable in R, otherwise cleared
C <- Carry generated during 32-bit internal addition

Program exceptions:

Truncation

Description:

The operands are zero extended to (pseudo) 33-bits. The first operand is added to the second operand and the third operand is replaced by the sum.

Unsigned Subtraction

SUBU R <- B + not(A) + 1 57, A, B, R

Operand types:
A, B, R 8-bit, 16-bit, 32-bit unsigned integer

Condition codes:
Z <- R Eql 0
N <- R Lss 0
T <- Set if the result is not representable in R, otherwise cleared
C <- Carry generated during 32-bit internal addition

Program exceptions:
Truncation

Description:
The operands are zero extended to (pseudo) 33-bits. The 2's complement of the first operand is added to the second operand and the third operand is replaced by the sum. A negative result sets T.

Unsigned Multiplication

MULU R <- A x B 58, A, B, R

Operand types:
A, B, R 8-bit, 16-bit, 32-bit unsigned integer

Condition codes:
Z <- R Eql 0
N <- R Lss 0
T <- Set if the result is not representable in R, otherwise cleared
C Unaffected

Program exceptions:
Truncation

Description:
The source operands are zero extended to 32-bits. The first operand is multiplied by the second operand and the third operand is replaced by the product.

Unsigned Division

DIVU R <- B / A 59, A, B, R

Operand types:

A, B, R 8-bit, 16-bit, 32-bit unsigned integer

Condition codes:

Z <- R Eql 0
N <- R Lss 0
T <- Set if the result is not representable in R, otherwise cleared
C Unaffected

Program exceptions:

Truncation
Illegal.Divisor

Description:

The source operands are zero extended to 32-bits. The second operand is divided by the first operand and the third operand is replaced by the quotient. An **Illegal.Divisor** exception shall be raised if A = 0. When this exception occurs the operands are unaffected.

19. Logical Instructions

All operands involved in logical instructions shall be zero extended internally to include enough bits so that all operations can be performed without any loss of information. If the instruction specifies a destination with fewer bits than the internal operation, the least significant bits of the result are stored in the destination.

In general, for logical type operands the negative (N) and zero (Z) condition codes are set to reflect the value of the result stored (which is treated as a signed integer for this setting), except where otherwise noted. The truncate (T) condition code and the carry (C) condition code are unaffected, unless otherwise noted.

Logical Not

NOT	R <- not(A)	46, A, R
	R <- not(R)	47, R

Operand types:
A, R 8-bit, 16-bit, 32-bit logical

Condition codes:
Z <- R Eql 0
N <- R Lss 0
T Unaffected
C Unaffected

Program exceptions:
None

Description:
In the 2 operand format, the second operand is replaced by the logical complement of the first operand. In the 1 operand format, the operand is replaced by its logical complement.

Logical And

AND	R <- A and B	48, A, B, R
	R <- R and A	49, A, R

Operand types:

A, B, R   8-bit, 16-bit, 32-bit logical

Condition codes:

Z <- R Eql 0
N <- R Lss 0
T Unaffected
C Unaffected

Program exceptions:

None

Description:

In the 3 operand format, the logical AND function is performed on the first two operands, and the third operand is replaced by the result. In the 2 operand format, the logical AND function is performed on the first two operands, and the second operand is replaced by the result.

Logical Or

OR	R <- A or B	4A, A, B, R
	R <- R or A	4B, A, R

Operand types:

A, B, R   8-bit, 16-bit, 32-bit logical

Condition codes:

Z <- R Eql 0
N <- R Lss 0
T Unaffected
C Unaffected

Program exceptions:

None

Description:

In the 3 operand format, the logical OR function is performed on the first two operands, and the third operand is replaced by the result. In the 2 operand format, the logical OR function is performed on the first two operands, and the second operand is replaced by the result.

Logical Exclusive Or

XOR   R <- A xor B   4C, A, B, R

Operand types:
A, B, R   8-bit, 16-bit, 32-bit logical

Condition codes:
Z <- R Eql 0
N <- R Lss 0
T Unaffected
C Unaffected

Program exceptions:
None

Description:
The logical Exclusive OR function is performed on the first two operands, and the third operand is replaced by the result.

Count One Bits

COB   5F, A, R

R <- Number of bits set in A

Operand types:
A   8-bit, 16-bit, 32-bit logical
R   8-bit, 16-bit, 32-bit unsigned integer

Condition codes:
Z <- R Eql 0
N <- 0
T Unaffected
C Unaffected

Program exceptions:
None

Description:
In COB, the second operand is replaced by the value that represents the number of bits set in the first operand.

20. Shift and Rotate Instructions

Arithmetic Scale

SCALE	R <- B scaled by 2^A	5C, A, B, R

Operand types:

A, B, R   8-bit, 16-bit, 32-bit signed integer

Condition codes:

Z <- R Eql 0
N <- R Lss 0
T <- Set if the result is not representable in R, otherwise cleared
C Unaffected

Program exceptions:

Truncation

Description:

In SCALE, the second operand is arithmetically scaled by the power of 2 specified by the first operand, and the third operand is replaced by the result. When the first operand is positive, B is multiplied by $2^{|A|}$. When the first operand is negative, B is divided by $2^{|A|}$ (the result being rounded toward zero, if necessary).

Rotate

ROT 5D, A, B, R

R <- B left rotated A bit positions

Operand types:

A	8-bit, 16-bit, 32-bit signed integer
B, R	8-bit, 16-bit, 32-bit logical

Condition codes:

Z <- R Eql 0
N <- R Lss 0
T Unaffected
C Unaffected

Program exceptions:

None

Description:

In ROT, the second operand is logically rotated by the number of bits specified by the first operand, and the third operand is replaced by the result. When A is positive, a left rotate |A| positions is performed. When A is negative, a right rotate |A| positions is performed. Rotation is performed on the length of the second operand (B).

Logical Shift

LSH 5E, A, B, R

R <- B left shifted A bit positions

Operand types:

A	8-bit, 16-bit, 32-bit signed integer
B, R	8-bit, 16-bit, 32-bit logical

Condition codes:

Z <- R Eql 0
N <- R Lss 0
T Unaffected
C Unaffected

Program exceptions:

None

Description:

In LSH, the second operand is shifted logically by the number of bits specified by the first operand, and the third operand is replaced by the result. When A is positive, a left shift |A| positions is performed and 0's are inserted into the least significant bits. When A is negative, a right shift |A| positions is performed and 0's are inserted into the most significant bits.

21. Move and Clear Instructions

Move Arithmetic (Sign Extended)

MOV R <- A 40, A, R

Operand types:
A, R 8-bit, 16-bit, 32-bit signed integer

Condition codes:
Z <- R Eql 0
N <- R Lss 0
T <- Set if A is not representable in R, otherwise cleared
C Unaffected

Program exceptions:
Truncation

Description:
Operand A is copied (sign extended or truncated as necessary) into R.

Move Logical (Zero Extended)

MOVL R <- A 41, A, R

Operand types:
A, R 8-bit, 16-bit, 32-bit logical

Condition codes:
Z <- R Eql 0
N <- R Lss 0
T Unaffected
C Unaffected

Program exceptions:
None

Description:
The operand A is copied (Zero extended or truncated as necessary) into R.

Move Address

MOVA	R <- address of A	61, A, R

Operand types:

A	Address Operand
R	8-bit, 16-bit, 32-bit logical

Condition codes:

Z <- R Eql 0
N <- R Lss 0
T Unaffected
C Unaffected

Program exceptions:

None

Description:

The address operand A is copied into R. If R is smaller than 32-bits, the high order part of A is lost.

Clear

CLR	R <- 0	45, R

Operand types:

R	8-Bit, 16-bit, 32-bit logical

Condition codes:

Z <- 1
N <- 0
T Unaffected
C Unaffected

Program exceptions:

None

Description:

The operand R is set to 0 (all bits cleared).

Exchange

EXCH	tmp1 <- A tmp2 <- B next B <- tmp1 A <- tmp2	3D, A, B

Operand types:
A, B 8-bit, 16-bit, 32-bit logical

Condition codes:
Z <- B Eql 0 (after transfer)
N <- B Lss 0 (after transfer)
T Unaffected
C Unaffected

Program exceptions:
None

Description:
The contents of the two operands A and B are exchanged. The condition codes are set to reflect the final transfer into B. If either transfer is blocked by the memory management system, the transfer is aborted and the operands are not affected. If operands A and B overlap, the resulting values are undefined.

Push onto SP Stack

PUSH	tmp <- SP - 4 next (tmp) <- A next SP <- tmp	71, A

Operand types:
A 8-bit, 16-bit, 32-bit signed integer

Condition codes:
Unaffected

Program exceptions:
Illegal.Register

Description:
If operand A is less than 32-bits, it is sign extended to 32-bits. The 32-bit item is copied onto the push down stack defined by register 1 (the stack pointer SP). An **Illegal.Register** exception shall be initiated if Maxreg = 0.

Pop from SP Stack

POP	tmp <- SP next A <- (tmp) next SP <- tmp + 4	73, A

Operand types:

A 8-bit, 16-bit, 32-bit signed integer

Condition codes:

Z <- A Eql 0
N <- A Lss 0
T <- Set if item popped is not representable in A, otherwise cleared
C Unaffected

Program exceptions:

Truncation
Illegal.Register

Description:

The top 32-bit item is removed from the stack defined by register 1 (the stack pointer SP) and is placed in operand A. An Illegal.Register exception shall be initiated if Maxreg = 0.

22. Compare and Test Instructions

Compare (Sign Extended)

CMP 42, A, B

Operand types:
A, B 8-bit, 16-bit, 32-bit signed integer

Condition codes:
Z <- A Eql B
N <- A Lss B
T Unaffected
C Unaffected

Program exceptions:
None

Description:
Arithmetic comparison is carried out on signed integers A and B, and the condition code bits are set accordingly.

Compare Unsigned

CMPU 43, A, B

Operand types:
A, B 8-bit, 16-bit, 32-bit unsigned integer

Condition codes:
Z <- A Eql B
N <- A Lssu B
T Unaffected
C Unaffected

Program exceptions:
None

Description:
A comparison is made between unsigned integers A and B, and the condition code bits are set accordingly.

Compare and Swap

CMPS 4D, A, B, C

If B eql C then B <- A else C <- B

Operand types:

A, B, C 8-bit, 16-bit, 32-bit logical

Condition codes:

Z <- B Eql C (before transfer)
N <- B Lssu C (before transfer)
T Unaffected
C Unaffected

Program exceptions:

None

Description:

The second and third operands are compared. If the result of the comparison is equality, the second operand is replaced by the first operand, otherwise the third operand is replaced by the second operand. This is an interlocked operation. A serialization function precedes operand B fetch. No other memory data access to B is permitted until completion of the store of operand B, or the negative result of the condition (or a trap causes termination of the instruction). A fetch of operand A is always performed. Write access rights (for operands B and C) are only checked for the operand that is actually stored.

Compare within Bounds

CMPWB 5A, A, B1, B2

Operand types:
A, B1, B2 8-bit, 16-bit, 32-bit signed integer

Condition codes:
Z <- (A Geq B1) AND (A Leq B2)
N <- (A Lss B1)
T Unaffected
C Unaffected

Program exceptions:
None

Description:
A check whether the value of the first operand falls within the bounds specified by the second (lower bound) and third (upper bound) operands is performed.

If B1 < B2, the N and Z condition code bits specify the result of the check as follows:

Z = 1; N = 0 - value is within bounds;
Z = 0; N = 1 - value is less than the lower bound;
Z = 0; N = 0 - value is greater than the upper bound;

If B1 ≥ B2, the condition codes are still set as shown in the condition codes section above, but the relations shown in the table immediately above no longer hold.

Compare within Bounds and Take Exception (Range Check)

RANGE 5B, A, B1, B2

If (A Lss B1) OR (A Gtr B2) then **Range.Error** Exception

Operand types:
A, B1, B2 8-bit, 16-bit, 32-bit signed integer

Condition codes:
Unaffected

Program exceptions:
Range.Error

Description:
A check whether the value of the first operand falls within the range specified by the second (lower bound) and third (upper bound) operands is performed, and the **Range.Error** exception is raised if it is out of range.

Test Integer

TEST 44, A

Operand types:

A 8-bit, 16-bit, 32-bit signed integer

Condition codes:

Z <- A Eql 0
N <- A Lss 0
T Unaffected
C Unaffected

Program exceptions:

None

Description:

Arithmetic comparison is carried out on the operand (signed integer) and zero, and the condition code bits are set accordingly.

Set Based on Condition

EQL	A <- sign.extended (Z)	24, A
NEQ	A <- sign.extended (not Z)	26, A
LSS	A <- sign.extended (N and (not Z))	28, A
GTR	A <- sign.extended (not (N or Z))	2E, A
LEQ	A <- sign.extended (N or Z)	2C, A
GEQ	A <- sign.extended (Z or (not N))	2A, A

Operand types:

A 8-bit, 16-bit, 32-bit logical

Condition codes:

Unaffected

Program exceptions:

None

Description:

The operand is set (all 1's) if the condition is true and reset if false (all 0's). These instructions are intended to follow compare instructions that set the condition code bits.

23. Control Instructions

Control instructions use displacements to change the execution control. Displacements are 8-bit or 16-bit signed integers that follow the instruction in the code stream. The displacement is fetched, sign extended to 32-bits and added to the address of the displacement in the code stream. Depending on the branch condition, the result of this displacement calculation may determine the location of the next instruction to be executed.

Jump

JUMP	NI <- address of A	62, A

Operand types:
A Address Operand

Condition codes:
Unaffected

Program exceptions:
None

Description:
The address of the next instruction to be executed is determined by address operand A.

Branch

BR		0E, 8-bit-displacement
		0F, 16-bit-displacement
	NI <- displ address + displ	

Operand types:
None

Condition codes:
Unaffected

Program exceptions:
None

Description:
The address of the next instruction to be executed is calculated by adding the sign-extended branch displacement to the branch displacement's address.

Branch on Equal

BEQL | 14, 8-bit-displacement
15, 16-bit-displacement

If Z then NI <- displ address + displ

Operand types:
None

Condition codes:
Unaffected

Program exceptions:
None

Description:
If the Z bit is set then the address of the next instruction to be executed is calculated by adding the sign-extended branch displacement to the branch displacement's address. If the branch condition is not met then the instruction following the displacement is the next one to be executed.

Branch on Not Equal

BNEQ | 16, 8-bit-displacement
17, 16-bit-displacement

If not Z then NI <- displ address + displ

Operand types:
None

Condition codes:
Unaffected

Program exceptions:
None

Description:
If the Z bit is reset then the address of the next instruction to be executed is calculated by adding the sign-extended branch displacement to the branch displacement's address. If the branch condition is not met then the instruction following the displacement is the next one to be executed.

Branch on Less than or Equal

BLEQ — 1C, 8-bit-displacement
1D, 16-bit-displacement

If Z or N then NI <- displ address + displ

Operand types:
None

Condition codes:
Unaffected

Program exceptions:
None

Description:
If the Z bit is set or the N bit is set then the address of the next instruction to be executed is calculated by adding the sign-extended branch displacement to the branch displacement's address. If the branch condition is not met then the instruction following the displacement is the next one to be executed.

Branch on Less than

BLSS — 18, 8-bit-displacement
19, 16-bit-displacement

If N and (not Z) then NI <- displ address + displ

Operand types:
None

Condition codes:
Unaffected

Program exceptions:
None

Description:
If the N bit is set and the Z bit is clear then the address of the next instruction to be executed is calculated by adding the sign-extended branch displacement to the branch displacement's address. If the branch condition is not met then the instruction following the displacement is the next one to be executed.

Branch on Greater than or Equal

BGEQ 1A, 8-bit-displacement
1B, 16-bit-displacement

If Z or (not N) then NI <- displ address + displ

Operand types:
None

Condition codes:
Unaffected

Program exceptions:
None

Description:
If the Z bit is set or the N bit is reset then the address of the next instruction to be executed is calculated by adding the sign-extended branch displacement to the branch displacement's address. If the branch condition is not met then the instruction following the displacement is the next one to be executed.

Branch on Greater than

BGTR 1E, 8-bit-displacement
1F, 16-bit-displacement

If (not N) and (not Z) then NI <- displ address + displ

Operand types:
None

Condition codes:
Unaffected

Program exceptions:
None

Description:
If the Z bit is reset and the N bit is reset then the address of the next instruction to be executed is calculated by adding the sign-extended branch displacement to the branch displacement's address. If the branch condition is not met then the instruction following the displacement is the next one to be executed.

Branch on Carry Set

BCS 10, 8-bit-displacement
11, 16-bit-displacement

If C then NI <- displ address + displ

Operand types:
None

Condition codes:
Unaffected

Program exceptions:
None

Description:
If the C bit is set then the address of the next instruction to be executed is calculated by adding the sign-extended branch displacement to the branch displacement's address. If the branch condition is not met then the instruction following the displacement is the next one to be executed.

Branch on Carry Clear

BCC 12, 8-bit-displacement
13, 16-bit-displacement

If not C then NI <- displ address + displ

Operand types:
None

Condition codes:
Unaffected

Program exceptions:
None

Description:
If the C bit is reset then the address of the next instruction to be executed is calculated by adding the sign-extended branch displacement to the branch displacement's address. If the branch condition is not met then the instruction following the displacement is the next one to be executed.

Branch on Truncate Set

BTS 20, 8-bit-displacement
21, 16-bit-displacement

If T then NI <- displ address + displ

Operand types:
None

Condition codes:
Unaffected

Program exceptions:
None

Description:
If the T bit is set then the address of the next instruction to be executed is calculated by adding the sign-extended branch displacement to the branch displacement's address. If the branch condition is not met then the instruction following the displacement is the next one to be executed.

Branch on Truncate Clear

BTC 22, 8-bit-displacement
23, 16-bit-displacement

If not T then NI <- displ address + displ

Operand types:
None

Condition codes:
Unaffected

Program exceptions:
None

Description:
If the T bit is reset then the address of the next instruction to be executed is calculated by adding the sign-extended branch displacement to the branch displacement's address. If the branch condition is not met then the instruction following the displacement is the next one to be executed.

Case

CASE 7C, Sel, Base, Num, 16-bit-Displ[0],
... , 16-bit-Displ[Num-1]

If (Sel - Base) Geq 0 And (Sel - Base) Lssu Num Then NI <- displ[0] address + displ[Sel - Base]
Otherwise NI <- displ[0] address + (2 x Num)

Operand types:

Sel, Base	8-bit, 16-bit, 32-bit signed integer
Num	16-bit inline literal

Condition codes:
Unaffected

Program exceptions:
None

Description:
The base operand (Base) is subtracted from the selector operand (Sel). This value is used in an unsigned comparison with the number of cases Num. If the value is less than Num then the address of the next instruction to be executed is calculated by adding the address of the first displacement in the displacement list to the displacement selected by the value 2 x (Sel - Base). Otherwise the instruction following the last displacement is the next one to be executed.

Loop

LOOP 27, Inc, Ctr, Lim, 16-bit-displacement

```
count <- Ctr + Inc next
If (Inc Gtr 0 and count Leq Lim) or
  (Inc Lss 0 and count Geq Lim)
  Then NI <- displ address + displ next
Ctr <- count
```

Operand types:

Inc, Ctr, Lim 8-bit, 16-bit, 32-bit signed integer

Condition codes:

Z Unaffected
N Unaffected
T <- Set if count is not representable in Ctr, otherwise cleared
C Unaffected

Program exceptions:

Truncation
Illegal.Divisor

Description:

The signed value of Inc is added to Ctr to form count. If one of the comparisons with the limit (Lim) is within range, the address of the next instruction to be executed is calculated by adding the sign-extended branch displacement to the branch displacement's address. Otherwise the instruction following the displacement is the next one to be executed. Ctr is updated with count. An **Illegal.Divisor** exception is initiated if Inc = 0.

NOTE: (Lim - initial value of Ctr)/Inc is the number of times through the loop.

Increment and Branch on Less than or Equal to

IBLEQ 2D, Ctr, Lim, 16-bit-displacement

count <- Ctr + 1 next
If count Leq Lim Then NI <- displ address + displ next
Ctr <- count

Operand types:
Ctr, Lim 8-bit, 16-bit, 32-bit signed integer

Condition codes:
Z Unaffected
N Unaffected
T <- Set if count is not representable in Ctr, otherwise cleared
C Unaffected

Program exceptions:
Truncation

Description:
One is added to Ctr to form count. If count is less than or equal to the limit (Lim), the address of the next instruction to be executed is calculated by adding the sign-extended branch displacement to the branch displacement's address. Otherwise the instruction following the displacement is the next one to be executed. Ctr is updated from count.

Increment and Branch on Less than

IBLSS 29, Ctr, Lim, 16-bit-displacement

count <- Ctr + 1 next
If count Lss Lim Then NI <- displ address + displ next
Ctr <- count

Operand types:
Ctr, Lim 8-bit, 16-bit, 32-bit signed integer

Condition codes:
Z Unaffected
N Unaffected
T <- Set if count is not representable in Ctr, otherwise cleared
C Unaffected

Program exceptions:
Truncation

Description:
One is added to Ctr to form count. If count is less than the limit (Lim), the address of the next instruction to be executed is calculated by adding the sign-extended branch displacement to the branch displacement's address. Otherwise the instruction following the displacement is the next one to be executed. Ctr is updated from count.

Decrement and Branch on Greater than or Equal to

DBGEQ 2B, Ctr, Lim, 16-bit-displacement

count <- Ctr - 1 next
If count Geq Lim Then NI <- displ address + displ next
Ctr <- count

Operand types:
Ctr, Lim 8-bit, 16-bit, 32-bit signed integer

Condition codes:
Z Unaffected
N Unaffected
T <- Set if count is not representable in Ctr, otherwise cleared
C Unaffected

Program exceptions:
Truncation

Description:
One is subtracted from Ctr to form count. If count is greater than or equal to the limit (Lim), the address of the next instruction to be executed is calculated by adding the sign-extended branch displacement to the branch displacement's address. Otherwise the instruction following the displacement is the next one to be executed. Ctr is updated from count.

Decrement and Branch on Greater

DBGTR 2F, Ctr, Lim, 16-bit-displacement

count <- Ctr - 1 next
If count Gtr Lim Then NI <- displ address + displ next
Ctr <- count

Operand types:
Ctr, Lim 8-bit, 16-bit, 32-bit signed integer

Condition codes:
Z Unaffected
N Unaffected
T <- Set if count is not representable in Ctr, otherwise cleared
C Unaffected

Program exceptions:
Truncation

Description:
One is subtracted from Ctr to form count. If count is greater than the limit (Lim), the address of the next instruction to be executed is calculated by adding the sign-extended branch displacement to the branch displacement's address. Otherwise the instruction following the displacement is the next one to be executed. Ctr is updated from count.

24. Procedure Call and Return Instructions

Call Procedure

CALL 64, Proc, Nparms, P1, ..., Pn
64, Proc, P1, ..., Pn

Operand types:

Proc	Address Operand
Nparms	8-bit inline literal
P1, ..., Pn	All operand types allowed

Condition codes:
Unaffected

Program exceptions:
Invalid.Supervisor trap

Description:
The CALL instruction invokes a procedure with a parameter list. The first operand of the CALL instruction shall be interpreted as the address of the entry point of the procedure to be invoked. Bits 30:31 of this address shall be forced to 0 in determining the location of the procedure entry point. If the most significant bit of this address is set (bit 0), and the current PSW has bit 17 (Supervisor) clear, a memory management trap shall occur. The reported error shall be **Invalid.Supervisor** (refer to section 12.4). Otherwise, the specified procedure shall be invoked as described in section 8.3. The remaining operands of the call instruction shall form the parameter list for this invocation. The privilege bit (bit 15) of the PSW shall remain unchanged.

Call Unprivileged Procedure

CALLU 65, Proc, Nparms, P1, ..., Pn
65, Proc, P1, ..., Pn

Operand types:

Proc	Address Operand
Nparms	8-bit inline literal
P1, ..., Pn	All operand types allowed

Condition codes:
Unaffected

Program exceptions:
Invalid.Supervisor trap

Description:
The CALLU instruction operates in the same way as the CALL instruction Except the privilege bit in the PSW of the called context shall be cleared.

Return from Procedure

RET 67

Operand types:
None

Condition codes:
Loaded depending on BASE bit of PSW

Program exceptions:
None

Description:
The RET instruction returns from a procedure. The function of this instruction shall depend upon the BASE bit (bit 16) of the current PSW. If the base bit is clear the current procedure context has a caller; if the base bit is set the current context is the base of the execution context. Because of this distinction, the RET instruction shall function differently in these cases:

BASE = 0 — The current procedure context shall be removed from the context stack. The caller's context (which is now at the top of the stack) shall be restored. Bits 13:31 of the PSW shall be restored to the value specified in the caller's context. Bits 0:12 of the PSW shall be unchanged.

BASE = 1 — The current procedure context shall be removed from the context stack. The Last Mode bit of the current PSW (bit 1) shall select the context stack to become the current context stack. The procedure context at the top of this context stack shall be restored, including the full PSW contained in this context.

Call Supervisor

SVC 66, Index, Nparms, P1, ..., Pn
66, Index, P1, ..., Pn

Operand types:

Index	8-bit, 16-bit, 32-bit unsigned integer
Nparms	8-bit inline literal
P1, ..., Pn	All operand types allowed

Condition codes:
Unaffected

Program exceptions:
None

Description:
The supervisor call instruction provides a means of calling a supervisor specified procedure in a protected manner. The first operand of this instruction shall be evaluated as an unsigned integer. The result shall be used as an index for a vectored call operation using the SVC vector registers as described in section 8.5. The procedure determined by the vectoring operation shall be invoked using the current context stack. The remaining operands of the SVC shall be used as a parameter list for the call.

Jump to Subroutine

JSR SP <- SP - 4 next 63, A
(SP) <- address of instruction following JSR next
NI <- address of A

Operand types:

A	Address Operand

Condition codes:
Unaffected

Program exceptions:
Illegal.Register

Description:
The address of the instruction following the JSR is stored on the push down stack defined by register 1 (SP). The address of the next instruction to be executed is then set to A. An **Illegal.Register** exception shall be generated if Maxreg = 0.

Return from Subroutine

RSR	NI <- (SP) next SP <- SP + 4	68

Operand types:
None

Condition codes:
Unaffected

Program exceptions:
Illegal.Register

Description:
The top item on the register 1 (SP) stack is removed and provides the address of the next instruction to be executed. An Illegal.Register exception shall be generated if Maxreg = 0.

25. Task Control Instructions

Load Task

LTASK 00, A

Operand types:
A Address Operand

Condition codes:
Unaffected

Program exceptions:
Privileged Instruction Trap
Context.Alignment
Task.Load.Error
Specification.Error

Description:
The Load TASK instruction prepares a new task for execution by loading its memory map pointer and context pointer into the appropriate registers. The operand of this instruction shall specify the address of a two word block. The first word (lowest address) will contain the value of the context pointer. This value shall be placed in the Task Context Pointer register. The second word will contain the physical address of the task's memory map and the protection and relocation bits. This address with protection and relocation bits shall be placed in the User Map Pointer register. If the value of bits 30:31 of this map pointer is reserved, a **Specification.Error** exception shall be raised. The Context Pointer is required to be aligned on a word boundary. If the value specified for the context pointer has bits 30 or 31 set, a **Context.Alignment** exception shall be raised. If LTASK is executed by a procedure executing on the Task Context stack, the instruction is aborted and a **Task.Load.Error** exception shall be raised.

This instruction shall have the implementation dependent effect of forcing any context stack or memory map caches to be consistent with the specified map and context stack in memory.

The context pointer loaded by LTASK shall not be checked for context access rights by the memory management system. These access rights are determined when task execution is begun.

Store Task

STASK 01, A

Operand types:
A Address Operand

Condition codes:
Unaffected

Program exceptions:
Privileged Instruction Trap

Description:
The Store TASK instruction is used to prepare a task for suspension by saving its context in memory. The operand A shall specify the address of a two word block with the same format as that used by LTASK. The Task Context Pointer shall be stored in the first word and the User Map Pointer shall be stored in the second word.

This instruction shall have the implementation dependent effect of forcing all cached Task context into memory. This insures that the task can be restarted by a subsequent LTASK. In order to guarantee that the memory image stored is correct, ANY alterations of the task's memory (required by the supervisor, for instance) must be performed before the STASK instruction that stores the task.

Start Task

TSTART	02, A
PSTART	03, A

Operand types:
A 8-bit, 16-bit, 32-bit unsigned integer

Condition codes:
Unaffected

Program exceptions:
Privileged Instruction Trap
Specification.Error

Description:
The TSTART and PSTART instructions cause the processor to resume execution of the topmost procedure context on the context stack specified by the operand. If PSTART is executed, the top execution context shall be removed from the current context stack. This means that procedure contexts are removed until one with the BASE bit set in its PSW is discarded. If TSTART is executed, the current context shall simply be suspended. Next both TSTART and PSTART shall cause execution to resume with the topmost context on the specified context stack after checking the context pointer for proper access rights. The operand is decoded as follows:

Operand Value	Specification
0	Kernel Context Stack
1	Task Context Stack
Otherwise	Specification.Error

A **Specification.Error** shall be raised if the value of the operand is not zero or one.

NOTE: Execution contexts should be built on context segment boundaries to insure that errors in the use of PSTART are detected by the memory management system.

Start Task Setting Exception

TRAISE	04, A, B
PRAISE	05, A, B

Operand types:
A, B 8-bit, 16-bit, 32-bit unsigned integer

Condition codes:
Unaffected

Program exceptions:
Privileged Instruction Trap
Specification.Error

Description:
The TRAISE and PRAISE instructions cause an exception to be raised in the topmost procedure on the context stack specified by the second operand. The exception code shall be specified by the first operand of the instruction. The exception code is truncated to 16 bits. PRAISE shall cause the top execution context to be removed from the current context stack. TRAISE shall simply suspend execution of the current context. Next both TRAISE and PRAISE shall cause an exception to be raised in the topmost procedure context on the specified context stack after checking the context pointer for proper access rights. Execution shall begin with the processing of this exception. The second operand is decoded as follows:

Operand Value	Specification
0	Kernel Context Stack
1	Task Context Stack
Otherwise	Specification.Error

A **Specification.Error** shall be raised if the value of the operand is not zero or one.

NOTE: Execution contexts should be built on context segment boundaries to insure that errors in the use of PRAISE are detected by the memory management system.

Initiate Task

TINIT	06, A, B
PINIT	07, A, B

Operand types:

A   Address Operand
B   8-bits, 16-bits, 32-bits logical

Condition codes:

Unaffected

Program exceptions:

Privileged Instruction Trap

Description:

The TINIT and PINIT instructions invoke a task. Specifically, they create an execution context on the specified context stack and begin its execution. The first operand shall specify the entry address of the procedure to be entered as the "main program". The second operand shall specify bits 0:15 of the PSW for the new execution context. Operand B is zero extended to 32 bits. Bits 16:31 of the zero extended B are used to specify bits 0:15 of the new PSW. Note that bit zero of the new PSW selects the context stack on which the execution context will be built. PINIT shall remove the top execution context from the current context stack. TINIT shall simply suspend execution of the current context. Both TINIT and PINIT shall then use the specified entry address and PSW to invoke the procedure as described in section 8.3.

NOTE: Execution contexts should be built on context segment boundaries to insure that errors in the use of PINIT are detected by the memory management system.

26. Exception Handling Instructions

Raise Exception

RAISE 69, A

Operand types:
A 8-bit, 16-bit, 32-bit unsigned integer

Condition codes:
Unaffected

Program exceptions:
None

Description:
The RAISE instruction shall cause an exception with the specified exception code A. Exception codes passed by RAISE are truncated to 16 bits.

Store Exception Code

ECODE 6A, A

Operand types:
A 8-bit, 16-bit, 32-bit unsigned integer

Condition codes:
Z <- A Eql 0
N <- A Lss 0
T <- Set if Exception code is not representable in A, otherwise cleared
C Unaffected

Program exceptions:
Truncation

Description:
The ECODE instruction allows an exception handler to obtain the exception code corresponding to the last exception. If the state of the procedure's exception handler is *exception code available*, the exception code shall be stored in the location specified by the operand A. Execution of the ECODE instruction shall place the procedure's exception handler in the *disabled* state. If ECODE is executed while the exception handler is in the *disabled* or *handler defined* states, A is set to 0.

Store Exception Handler Address

STOREH 78, A

Operand types:

A 8-bit, 16-bit, 32-bit unsigned integer

Condition codes:

Z <- A Eql 0
N <- A Lss 0
T <- Set if Handler address is not representable in A, otherwise cleared
C Unaffected

Program exceptions:

Truncation

Description:

If the exception handler is in the *handler defined* state, the STOREH instruction stores the handler entry address from the context stack into A. If the exception handler is in the *disabled* or *exception code available* states, A is set to 0.

Set Exception Handler Entry Address

EXCEPT 6B, A

Operand types:

A Address Operand

Condition codes:

Unaffected

Program exceptions:

None

Description:

The EXCEPT instruction allows a procedure to redefine the starting address of its exception handler. Execution of the EXCEPT instruction shall place the procedure's exception handler in the *handler defined* state. The handler start address shall be set to the address specified by the operand A. If A evaluates to virtual address 0, the exception handler shall be placed in the *disabled* state.

Exception Return

ERET 6C, A

Operand types:

A 8-bit, 16-bit, 32-bit unsigned integer

Condition codes:

Unaffected

Program exceptions:

Task.Failure

Description:

The ERET instruction returns an exception to the caller of a procedure. The exception code shall be specified by the operand of the instruction. The exception code is truncated to 16 bits. This instruction functions similarly to the RET instruction, except that if there is no caller (BASE = 1) an error condition exists:

BASE = 0 — The current procedure context shall be removed from the context stack. The caller's context (which is now at the top of the stack) shall be restored. Bits 13:31 of the PSW shall be restored to the value specified in the caller's context. Bits 0:12 of the PSW shall be unchanged. An exception shall be raised in the restored procedure context with the exception code specified by the ERET instruction.

BASE = 1 — The current procedure context shall be removed from the context stack. The Supervisor Exception Handler (section 9.5) shall be invoked with the exception Task.Failure. The BASE bit in the supervisor exception handler's context shall be set.

Exception Return and Propagate

ERP 6D, A

Operand types:
A 8-bit, 16-bit, 32-bit unsigned integer

Condition codes:
Unaffected

Program exceptions:
Task.Failure

Description:
The ERP instruction allows the Supervisor Exception Handler to force an exception back to the handler of the procedure in which it occurs. This is done by suppressing the examination of the UDLE bit when the exception is first raised. The ERP instruction shall function identically with the ERET instruction except the UDLE bit shall be treated as 0 in the procedure to which the return is made. Should the exception be propagated to the procedure's caller, the UDLE bit of the caller shall function normally.

27. String Instructions

All the string instructions are required to be interruptible. If an interrupt or trap occurs during the execution of such an instruction at a point where processing has begun but not yet completed, the intermediate state of the instruction is preserved (in the context stack, in an implementation--dependent form). When the interrupt or trap handler returns and the instruction is resumed, the instruction shall be correctly completed, provided that certain operands of the instruction have not been altered by means other than the interrupted string instruction.

To be more precise, the string instructions operate on ordinary operands and also regions of memory. Such regions are variable size arrays of bytes in contiguous virtual addresses. The first byte address in such a region is specified by an address operand. The size of the region is determined in an instruction dependent manner.

If a string instruction is interrupted before completion, the entire contents of its destination region, as well as any condition codes set by the instruction, are undefined unless and until the instruction is resumed and completed. Moreover, if any source or destination region of a string instruction is altered after the instruction processing has begun but has not yet completed because of an interrupt or trap, or is altered because of any other memory writes not performed by the CPU (such as an I/O transfer) after the instruction processing has begun but has not yet completed, then when the instruction is resumed it shall complete and correctly transfer control to the next instruction, but the contents of any destination region are undefined, and any condition codes or ordinary destination operands set by the instruction are undefined.

Compare Block

CMPBK 95, Cnt, S1, S2

Operand types:

S1, S2 Address Operands
Cnt 8-bit, 16-bit, 32-bit unsigned integer

Condition codes:

Z <- S1 Eql S2
N <- S1(failed) Lssu S2(failed)
T Unaffected
C Unaffected

Program exceptions:

Operand.Size

Description:

Strings S1 and S2, each with Cnt items, are compared as unsigned integers item by item beginning with the low addresses and proceeding to higher addresses. The size of the items compared is derived from the size fields of the operand specifiers for S1 and S2. If size(S1) is different from size(S2) then an Operand.Size exception shall be raised. If all of the items in S1 are identical to the items in S2, the Z bit shall be set. Otherwise the Z bit shall be cleared. Upon encountering two nonequal items, the instruction may terminate. The N bit shall be set based on the first nonequal items encountered. If all items are identical, the N bit shall be cleared. If Cnt = 0, Z shall be set and N shall be cleared. This instruction shall be interruptible.

NOTE: If the two blocks are not equal, the implementation is not required to check or perform memory accesses beyond the first nonequal items.

Move Block

MOVBK 94, Cnt, Src, Dest

Operand types:

Src, Dest	Address Operands
Cnt	8-bit, 16-bit, 32-bit unsigned integer

Condition codes:
Unaffected

Program exceptions:
Operand.Size

Description:
Cnt items are copied from (Src) to (Dest). If size(Src) = size(Dest) then overlapping source and destination fields do not affect the results. If size(Src) is different from size(Dest), an Operand.Size exception shall be raised. If Src = Dest, no move is required. This instruction shall be interruptible.

NOTE: If Src = Dest, the implementation is not required to check or perform memory accesses.

Move Multiple (Fill)

MOVM 93, Cnt, Src, Dest

Operand types:

Cnt	8-bit, 16-bit, 32-bit unsigned integer
Src	8-bit, 16-bit, 32-bit logical
Dest	Address Operand

Condition codes:
Unaffected

Program exceptions:
None

Description:
Replace Cnt items, beginning at Dest with Src. If size(Src) is larger than size(Dest), Src is truncated and no indication is given (as with other logicals). If size(Src) is smaller than size(Dest), Src is zero extended. This instruction shall be interruptible.

Move Translated

MOVTR 96, Table, Cnt, Src, Dest

Operand types:

Table, Src, Dest — Address Operands

Cnt — 8-bit, 16-bit, 32-bit unsigned integer

Condition codes:

Unaffected

Program exceptions:

Operand.Size

Description:

Cnt *bytes* are copied from the source string (Src), translated, and placed in the destination string (Dest). Table is the address of a 256 byte vector. Each byte is read from the source string, used as an unsigned index into the table, and the corresponding entry in the table is copied into the destination string. If the source and destination strings overlap, the destination string will contain the correct result. If either size(Src) or size(Dest) is NOT byte, an **Operand.Size** exception shall be raised. If Dest overlaps the translation table, the result is undefined. This instruction shall be interruptible.

Scan and Break

SCANB	97, Btable, Src, Slen, Blen

Operand types:

Btable, Src	Address Operands
Slen, Blen	8-bit, 16-bit, 32-bit unsigned integer

Condition codes:

Z <- Blen Eql Slen
N <- Blen Lssu Slen
T Unaffected
C Unaffected

Program exceptions:

Operand.Size

Description:

SCANB scans a source string byte by byte in a forward direction (increasing addresses) until it finds a break byte or until the source string is exhausted (Slen bytes have been examined). Btable is the address of a 256-bit vector called the break table. This bit vector is similar to the bit fields handled by the bit field instructions. Btable is the base pointer for this bit field. Bytes taken from the source string are used to index the break table. The byte is zero extended forming a positive integer indicating the position of a bit in the bit vector (just as the position combines with the base to indicate a bit in a bit field). If the bit referenced by the source string byte is reset (0), the next byte in the source string is checked. If the bit in the vector is set (1) the number of bytes checked BEFORE the set bit was encountered is placed in the operand Blen, and the instruction is terminated. If no break bit is found, Blen is set to Slen. If Slen is 0, Blen is set to 0. If size(Src) is NOT byte, an **Operand.Size** exception shall be initiated. This instruction shall be interruptible.

28. Bit Field Instructions

A bit field is defined by three parameters, the base (Base), the position (Pos) and the size (Size). The base is a byte address, the position is the count in bits from the beginning of the byte referenced by the base to the start of the bit field, and the size is the size of the field. A bit field is formed by concatenating bytes with successively higher addresses. Bits within bytes are numbered from left to right. The base can be thought of as pointing to bit 0 of the byte it addresses. The position indicates the starting bit of the bit field, relative to the base. The position may be positive or negative. The beginning of the bit field is bit number {position MOD 8} in byte address {base + $\lfloor$(position/8)$\rfloor$}. For example, if position is -10, the bit field begins at bit 6 of byte base - 2. If position is 12, the bit field begins at bit 4 of byte base + 1. The protection check is done after the position is added to the base. If the specified size is greater than 32 a **Bit.Field.Size** exception shall be raised. Instructions SETBIT, CLRBIT, STOBIT, and TSTBIT implicitly reference bit fields of size 1.

Store Bit Field

SBF 7D, Src, Pos, Size, Base

field(Pos,Size,Base) <- Src

Operand types:

Src, Size	8-bit, 16-bit, 32-bit unsigned integer
Pos	8-bit, 16-bit, 32-bit signed integer
Base	Address Operand

Condition codes:
Unaffected

Program exceptions:
Bit.Field.Size

Description:
The bit field defined by Pos, Size and Base is replaced by the operand Src. If the size of Src in bits is smaller than Size, the high order part of the bit field is filled with zeros. If the size of Src in bits is larger than Size, the high order bits of Src are not stored. If Size is 0, no bits are stored.

Load Bit Field (Sign Extended)

LBFS 7E, Pos, Size, Base, R

R <- field(Pos,Size,Base) ; sign extended

Operand types:

Size	8-bit, 16-bit, 32-bit unsigned integer
Pos, R	8-bit, 16-bit, 32-bit signed integer
Base	Address Operand

Condition codes:

Z <- R Eql 0
N <- R Lss 0
T <- Set if the field value is not representable in R, otherwise cleared
C Unaffected

Program exceptions:

Truncation
Bit.Field.Size

Description:

The contents of the bit field defined by Pos, Size and Base are copied into R. If the size of R in bits is larger than Size, the field is sign extended. If the size of R in bits is smaller than Size, the field is truncated from the most significant part and T is set if the value contained in the field is not representable in R. If Size is 0, R is set to 0.

Load Bit Field (Zero Extended)

LBF 7F, Pos, Size, Base, R

R <- field(Pos,Size,Base) ; zero extended

Operand types:

Size	8-bit, 16-bit, 32-bit unsigned integer
Pos	8-bit, 16-bit, 32-bit signed integer
Base	Address Operand
R	8-bit, 16-bit, 32-bit logical

Condition codes:

Z <- R Eql 0
N <- R Lss 0
T Unaffected
C Unaffected

Program exceptions:

Bit.Field.Size

Description:

The contents of the bit field defined by Pos, Size and Base are copied into R. If the size of R in bits is larger than Size, the field is zero extended. If the size of R in bits is smaller than Size, the field is truncated from the most significant part. If Size is 0, R is set to 0.

Set Bit

SETBIT 74, Pos, Base

N <- Bit(Pos,Base) next
bit(Pos,Base) <- 1

Operand types:

Pos 8-bit, 16-bit, 32-bit signed integer
Base Address Operand

Condition codes:

Z <- Not original contents of Bit(Pos,Base)
N <- Original contents of Bit(Pos,Base)
T Unaffected
C Unaffected

Program exceptions:

None

Description:

The bit specified by Pos and Base is interrogated and set to One.

Clear Bit

CLRBIT 75, Pos, Base

N <- Bit(Pos,Base) next
Bit(Pos,Base) <- 0

Operand types:

Pos 8-bit, 16-bit, 32-bit signed integer
Base Address Operand

Condition codes:

Z <- Not original contents of Bit(Pos,Base)
N <- Original contents of Bit(Pos,Base)
T Unaffected
C Unaffected

Program exceptions:

None

Description:

The bit selected by Pos and Base is interrogated and set to zero.

Test and Store Bit

STOBIT 76, Src, Pos, Base

N <- Bit(Pos,Base) next
Bit(Pos,Base) <- LSB(Src)

Operand types:

Src	8-bit, 16-bit, 32-bit unsigned integer
Pos	8-bit, 16-bit, 32-bit signed integer
Base	Address Operand

Condition codes:

Z <- Not original contents of Bit(Pos,Base)
N <- Original contents of Bit(Pos,Base)
T Unaffected
C Unaffected

Program exceptions:

None

Description:

The bit selected by Pos and Base is interrogated. Next, the least significant bit of operand Src is placed in the bit selected by Pos and Base. This is an interlocked operation. A serialization function precedes the fetch of the bit selected by Pos and Base. No other memory access to this bit is permitted until completion of the store into this bit (or a trap causes termination of the instruction).

Test Bit

TSTBIT 77, Pos, Base

Operand types:

Pos	8-bit, 16-bit, 32-bit signed integer
Base	Address Operand

Condition codes:

Z <- not Bit(Pos,Base)
N <- Bit(Pos,Base)
T Unaffected
C Unaffected

Program exceptions:

None

Description:

The bit selected by Pos and Base is interrogated to set the condition codes.

29. Miscellaneous Instructions

No Operation

NOP 6E

Operand types:
None

Condition codes:
Unaffected

Program exceptions:
None

Description:
No operation is performed. The instruction following the NOP is the next instruction to be executed.

Break

BREAK 6F

Operand types:
None

Condition codes:
Unaffected

Program exceptions:
None

Description:
The Supervisor Exception Handler is invoked with exception code **Break**.

Load PSW

LPSW 08, A

Operand types:

A 8-bit, 16-bit, 32-bit logical

Condition codes:

Unaffected

Program exceptions:

Privileged Instruction Trap
Context.Base

Description:

The LPSW instruction replaces the *caller's* PSW contents by the value of its operand. This shall be a privileged instruction. The effect of altering the BASE (PSW<16>), Maxreg (PSW<20:23>), or number of parameters (PSW<24:31>) fields is unpredictable. If the BASE bit of the current PSW = 1 a Context.Base exception shall be raised.

Store PSW

SPSW 09, A

Operand types:

A 8-bit, 16-bit, 32-bit logical

Condition codes:

Unaffected

Program exceptions:

Privileged Instruction Trap
Context.Base

Description:

The SPSW instruction shall store the contents of the *caller's* PSW in the location specified by its operand. If the BASE bit of the current PSW = 1 a Context.Base exception shall be raised.

Size

SIZE	R <- Number of bytes in A	7B, A, R

Operand types:

A	8-bit, 16-bit, 32-bit, 64-bit logical
R	8-bit, 16-bit, 32-bit logical

Condition codes:

Z <- 0
N <- 0
T Unaffected
C Unaffected

Program exceptions:

None

Description:

The size, in bytes, of the operand A is returned in R. The size of a short literal is one byte. The size of parameter 0 (?0) is one byte.

Set Condition Codes

SETCC	CC <- A	7A, A

Operand types:

A	8-bit, 16-bit, 32-bit logical

Condition codes:

The condition codes EAE,C,T,N,Z are set based on the low order bits of A respectively.

Program exceptions:

Truncation

Description:

The condition codes are set based on A. The lowest order bit of A is placed in Z. The second lowest order bit of A is placed in N. The third lowest order bit of A is placed in T. The fourth lowest order bit of A is placed in C. The fifth lowest order bit of A is placed in EAE.

Set Priority Level

SETPRI priority <- A 98, A

Operand types:
A 8-bit, 16-bit, 32-bit logical

Condition codes:
Unaffected

Program exceptions:
Privileged Instruction Trap

Description:
The five least significant bits of the operand are placed in the priority field of the current PSW. Note that, when the PSW base bit is clear, this new priority propagates back to the caller upon execution of a RET instruction.

Window into the Micromachine

WINDOW 0D, Info

Operand types:
Info 8-bit inline literal

Condition codes:
Implementation dependent

Program exceptions:
Privileged Instruction Trap

Description:
The byte of information (Info) following the opcode in the instruction stream is passed into the micromachine in an implementation dependent manner. The action of this instruction is implementation dependent.

Replace Entry in Map

REPENT 0A, A, Map, Seg

Operand types:

Map, Seg	8-bit, 16-bit, 32-bit logical
A	16-bit, 32-bit, 64-bit logical

Condition codes:
Unaffected

Program exceptions:
Privileged Instruction Trap
Specification.Error

Description:
The replace entry in map instruction provides a guaranteed way of altering an entry in a memory map. Execution of this instruction shall cause the value of the operand A to be stored in the map entry specified by the segment number (Seg) and map pointer number (Map) operands. The Supervisor Map Pointer shall be map number 1; the User Map Pointer shall be map number 0. If Map specifies a value other than 0 or 1, a Specification.Error exception shall be raised. This instruction shall have the implementation dependent effect of forcing any translation cache to be consistent with this new entry and the current map length stored in memory.

Map Virtual Address

MAP OB, A, Ptr, Phy, Seg

Operand types:
A, Ptr, Phy, Seg 8-bit, 16-bit, 32-bit logical

Condition codes:
Z <- Set if segment found, otherwise cleared
N <- Set if the map is disabled, otherwise cleared
T Unaffected
C Unaffected

Program exceptions:
Privileged Instruction Trap

Description:
The first operand (A) is zero extended to 32 bits and treated as a virtual address. The virtual address is translated using the map specified by Ptr. Ptr is zero extended to 32 bits and interpreted in the same way that the contents of a map pointer register is interpreted (see figure 12-2). The MAP instruction's segment association shall be functionally equivalent to the hardware segment association (see section 12.2.3) except that the total number of segments specified by the map size (32-bit word immediately preceding the first segment descriptor in the map) shall be used during the segment association. If either of the two least significant bits of Ptr are set, segment association is performed by MAP. If a segment containing the virtual address A is found, the corresponding physical address is placed in Phy, the associated segment number is placed in Seg, the N bit is cleared, and the Z bit is set. If a segment containing A is not found, the Z and N bits are cleared and Phy and Seg are set to 0. If both relocation and protection are disabled based on bits in the pointer operand, Seg is set to 0, the most significant bit of the extended A is cleared and the resulting value is placed in Phy, the Z bit is cleared, and the N bit is set. Access and privilege information contained in the map specified by PTR is ignored by the MAP instruction. If storage of either Phy or Seg is blocked by the memory management system, the storage is aborted and the operands are not affected. If operands Phy and Seg overlap their resulting values are undefined.

Set I/O Segment

SETSEG OC, Seg, Adr

Operand types:

Seg, Adr Address Operands

Condition codes:

Z <- Set if Adr is not within virtual space, otherwise cleared
N <- Set if Access code is incorrect, otherwise cleared
T Unaffected
C Unaffected

Program exceptions:

Privileged Instruction Trap
Segment.Specifier
IOC.Busy

Description:

The privileged Set I/O Segment instruction transmits CPU segment information to an IOC. Operand Seg is interpreted as the virtual address of one of the IOC segment specifiers in an assigned IOC control register block. See sections 13.3 and 13.3.1. This virtual address is translated to a physical address that should be one of the following forms:

Address (Hex)	Segment Specifier	Access Code Required
000XXX10	Channel Program	Instruction
000XXX20	Message	Read/Write
000XXX30	Data Buffer	Read/Write

If the physical address specified by the operand Seg is not of one of these forms, a **Segment.Specifier** exception shall be initiated. If the physical address (in particular, bits 12 to 27) does not specify a segment specifier location within an implemented IOC control register block, a **Segment.Specifier** exception or a hard memory error trap shall be initiated, depending upon the implementation. If the run status bit (bit 1) in the channel status register of the IOC selected by the operand Seg is set (indicating that the IOC is running) then an **IOC.Busy** exception shall be initiated. Read/write access rights are required for the 16 bytes that are addressed by Seg.

Operand Adr is interpreted as a virtual address in the currently active supervisor or task memory maps. No access is made to the item addressed by Adr. The segment containing this address is to be made accessible to an IOC. If Adr is not contained in the currently active virtual memory space, the selected segment specifier shall be set to prohibit all accesses, the Z condition code shall be set, the N condition code shall be cleared, and the instruction shall terminate. The virtual address FFFFFFFF (hex) shall always be interpreted as such an invalid address by this instruction.

If no errors have been detected, the segment indicated by the Adr operand shall be checked for the proper access code, as listed in the table above. In particular, if Seg references a Channel Program Segment specifier, the indicated segment shall be checked for an Instruction access key. If Seg references a Message or Data Buffer Segment specifier, the indicated segment shall be checked for data Read/Write access. If proper access is not found the selected segment specifier shall be set to prohibit all accesses, the Z condition code shall be cleared, the N condition code shall be set, and the instruction shall be terminated. The access code check shall be omitted if the map containing the CPU segment has protection disabled.

If the I/O segment referenced is valid and the CPU segment has the proper access, the N and Z condition codes shall be cleared, and implementation dependent mapping information shall be transmitted to the selected IOC segment specifier. This information shall be sufficient to enable the IOC to distinguish valid virtual addresses within the selected CPU segment and to relocate them to the physical memory assigned to the segment, as specified by the CPU map entry. If the map containing the CPU segment has relocation disabled, the I/O segment referenced is valid, and the protection checks (if any) are satisfied, the IOC shall be given access to all of physical memory for IOC accesses through the I/O segment specified and the N and Z bits shall be cleared. Note that three proper SETSEG instructions must be executed before an IOC can run with separate program, message and data areas even if mapping in the CPU is disabled.

Wait for Interrupt

WAIT 25

Operand types:
None

Condition codes:
Unaffected

Program exceptions:
Privileged Instruction Trap

Description:
The processor shall enter a wait state until an interrupt of higher priority than the priority of the procedure executing the WAIT instruction is posted. Upon the receipt of such an interrupt, the processor shall service the interrupt normally and continue with the instruction after the WAIT when the interrupt procedure executes a RET.

Check Privilege Rights

PCHECK 70, A

Operand types:
A 8-bit, 16-bit, 32-bit, 64-bit logical

Condition codes:
Unaffected

Program exceptions:
Privileged Instruction Trap
Context.Base

Description:
The PCHECK instruction is used to check the privilege of an operand (usually a parameter) in the privilege context of the caller. Bit 15 of the *caller's* PSW is copied into bit 15 of the current PSW. A data read access of operand A is performed. If the access is successful (no traps are taken) then bit 15 of the current PSW is set and the instruction terminates. If a trap is initiated due to the attempted access, bit 15 of the PSW that is stored when the trap is initiated will reflect the privilege of the caller. If the BASE bit of the PSW = 1 a **Context.Base** exception shall be raised.

Check Supervisor Rights

SCHECK 72, A

Operand types:
A 8-bit, 16-bit, 32-bit, 64-bit logical

Condition codes:
Unaffected

Program exceptions:
Supervisor.Check
Context.Base

Description:
The SCHECK instruction is used to check the supervisor rights of an operand (usually a parameter) in the context of the caller. If operand A is memory type (in other words, the operand can be used as an address type operand, this includes memory type parameters only) then the supervisor bit of the virtual address of each byte of the operand is examined. If bit 0 of any of these virtual addresses is set and bit 17 (the supervisor bit) of the *caller's* PSW is clear then a **Supervisor.Check** exception shall be raised. If bit 0 of all of the virtual addresses is clear or if bit 17 of the caller's PSW is set or if operand A is not memory type, then the instruction shall terminate. If the BASE bit of the PSW = 1 a **Context.Base** exception shall be raised.

Reset

RESET 79

Operand types:
None

Condition codes:
reset

Program exceptions:
Privileged Instruction Trap
Context.Alignment

Description:
The RESET instruction initiates the reset operation described in section 11.9.

30. Floating-Point Arithmetic

The floating-point instructions are used to perform calculations on operands with a wide range of magnitude.

30.1. Floating-Point Data Classes. There are five floating-point data classes: normalized numbers, denormalized numbers, normal zeroes, infinities, and NaN.

Normalized — Any number whose exponent is not equal to the minimum or maximum exponent allowed by the format. Normalized numbers have an assumed 1 at the left of the fraction. This bit is implied in the stored number.

Denormalized — Any number with a zero exponent field and a non-zero fraction is a denormalized number.

Normal zero — A normal zero has a zero exponent and a zero fraction. Both plus and minus zero are allowed.

Infinity — A number with a maximum exponent and a zero fraction is interpreted as infinity. Plus and minus infinity are allowed.

NaN — A number with the maximum exponent and a non-zero fraction is a NaN (Not a Number). There are two types of NaNs, trapping and non-trapping. A trapping NaN will create an **Invalid.Operation** condition when used as a source operand. Non-trapping NaNs will propagate through arithmetic operations without raising exceptions. Trapping NaNs are distinguished by having their most significant fraction bit set. If the number is a NaN and bit 12 in double precision or bit 9 in single precision is set then the number is a trapping NaN.

30.2. Floating-Point Formats. Floating-point data shall occupy a fixed-length format which may be either a 32-bit format (single) or a 64-bit format (double). Figure 30-1 shows the number representation for the two formats.

In the single precision format the leftmost bit (bit 0) is the sign bit. Bits 1:8 are occupied by the exponent. The exponent is a biased exponent with a bias of 127. The remaining 23 bits contain the fraction.

In the double precision format the leftmost bit (bit 0) is the sign bit. Bits 1:11 contain the exponent. The exponent is a biased exponent with a bias of 1023. The remaining 52 bits contain the fraction.

In both formats the fraction has an implicit binary point to the left of the most significant bit.

	SINGLE	DOUBLE
Fields and widths in bits:		
S = Sign	1	1
E = Exponent	8	11
F = Fraction	23	52
L = Leading bit (implicit)	(1)	(1)
Total Width	(1) + 32	(1) + 64
Sign: +/- represented by 0/1 respectively		
Exponent: biased integer		
Max E	255	2047
Min E	0	0
Bias of E	127	1023
Normalized numbers:		
Range of E	(Min E + 1) to (Max E - 1)	
Represented number	$(-1)^{S} \times 2^{E-Bias} \times (1.F)$	
Signed zeros:		
E	Min E	Min E
F	0	0
Reserved Operands:		
Denormalized numbers:		
E	Min E	Min E
F	nonzero	nonzero
Represented number	$(-1)^{S} \times 2^{E-Bias+1} \times (0.F)$	
Signed Infinities:		
E	Max E	Max E
F	0	0
NaN's:		
E	Max E	Max E
F	nonzero	nonzero

Figure 30-1: Floating Point Number Representation

30.3. Floating-Point Operands. Operands to floating point instructions may specify 32-bit or 64-bit operands in registers or memory. Literal operands may specify 16-bit, 32-bit, or 64-bit literals (literals include parameter literals). The format of 32-bit and 64-bit operands shall be those described above. 16-bit literal floating point operands shall be extended with 16 least significant bits of zero and interpreted as a 32-bit format number. An addressing mode that specifies any operand representation other than these shall cause an exception as described in section 5.12.

30.4. Floating-Point Exceptions. There are five maskable floating-point exception conditions. These conditions are **Invalid.Operation**, **Divide.By.Zero**, **Floating.Underflow**, **Floating.Overflow** and **Floating.Inexact**. They are explained further below.

One non-maskable exception condition can occur during execution of the floating-point compare (CMPF) instruction. This exception condition, named **Unordered**, shall occur when the operands in

the compare cannot be related using conventional relational operators (= ,<,>...). An example of this would be a comparison between NaN and a finite number.

Associated with each maskable exception is a "sticky" flag and a mask bit. The sticky flags are bits 27:31 (See Figure 30-2) in the Auxiliary Status Register (ASR). The mask bits are bits 19:23 in the ASR. The action taken when an exception condition occurs depends upon the setting of the mask bit associated with the exception and the EAE bit in the PSW. The table below describes the action to be taken when such an exception condition occurs.

Mask Bit	EAE	Action Taken
0	0	Set T condition code in PSW (sticky bit unaffected)
0	1	Raise exception with corresponding code (T bit and sticky bit unaffected)
1	0 or 1	Set associated sticky flag (T bit unaffected)

Sticky flags can be cleared by moving zeros into the ASR.

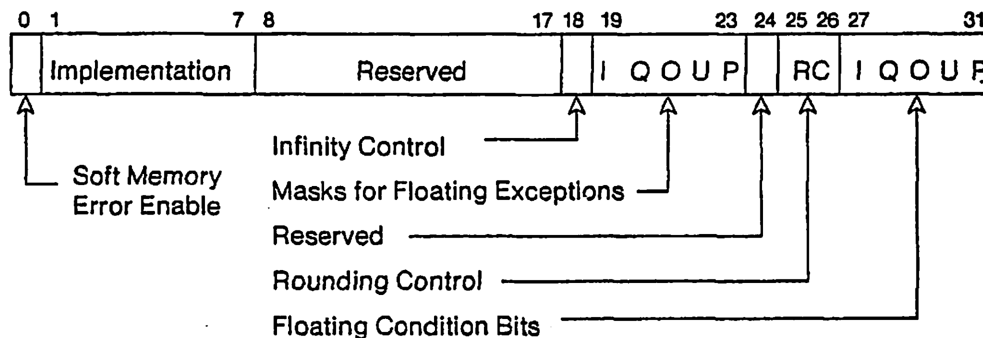


Figure 30-2: Floating Point Control Bits in ASR

30.4.1. Invalid.Operation. The Invalid.Operation condition exists in floating-point arithmetic operations on errors that are not frequent enough or important enough to merit their own fault conditions. An **Invalid.Operation** exception shall be raised if arithmetic exceptions are enabled and the mask bit (bit 19 in the ASR) is zero. If the exception is masked then the sticky flag for this exception (bit 27 in the ASR) shall be set and the result shall be a non-trapping NaN.

There are two cases for Invalid.Operation conditions. The first arises if an operand is illegal for the operation to be performed. The other arises if the result is illegal for the destination.

Some examples of Invalid Operations are:

SQRT(-5)
$\infty - \infty$
$0 \times \infty$

30.4.2. Divide.By.Zero. The Divide.By.Zero condition exists in a division operation when the divisor is normal zero and the dividend is a finite nonzero number. The **Divide.By.Zero** exception shall be raised if arithmetic exceptions are enabled and the mask bit (bit 20 in the ASR) is zero. If the exception is masked then the sticky flag for this condition (bit 28 in the ASR) shall be set and the result shall be infinity with sign according to convention.

30.4.3. Floating.Overflow. If the exponent of a rounded result of an arithmetic operation overflows the range of the destination, then the Floating.Overflow condition exists. The **Floating.Overflow** exception shall be raised if arithmetic exceptions are enabled and the mask bit (bit 21 in the ASR) is zero. If the exception is masked then the sticky flag for this condition (bit 29 in the ASR) shall be set and the result depends on the rounding mode and the sign of the result. The result is set to infinity with the sign of the result if the rounding mode is RN, RZ, (RP and the result is positive) or (RM and the result is negative). Otherwise the result is the largest magnitude normalized number representable in the destination.

30.4.4. Floating.Underflow. A Floating.Underflow condition exists if the exponent of a result lies below the exponent range of the destination field. The test for underflow may be performed either after temporarily rounding the result toward zero or after rounding based on the rounding mode at the implementor's option. The **Floating.Underflow** exception shall be raised if arithmetic exceptions are enabled and the mask bit (bit 22 in the ASR) is zero. If the exception is masked then the sticky flag for this condition (bit 30 in the ASR) shall be set and the unrounded result shall be denormalized by shifting the fraction right while incrementing the exponent until the exponent reaches its minimum allowable value.

30.4.5. Floating.Inexact. In the absence of an Invalid.Operation condition, if the result cannot be exactly specified in the destination format and no other exceptions are raised a Floating.Inexact condition exists. If arithmetic exceptions are enabled and the mask bit for this exception (bit 23 in the ASR) is zero then the **Floating.Inexact** exception shall be raised. If the exception is masked then the sticky flag for this condition (bit 31 in the ASR) shall be set and the result shall be the correctly rounded number.

30.5. Rounding. Four rounding modes are supported. The mode currently in effect is determined by bits 25:26 of the ASR as described in the table below.

Bits 25:26	**Rounding Mode**	**Action**
00	RN	Round to Nearest
01	RZ	Round toward Zero
10	RP	Round toward + infinity
11	RM	Round toward - infinity

Preliminary results can be viewed as having been computed to infinite precision. From the preliminary result (R), determine the two numbers (R1 and R2) in the desired format that most closely bracket R.

If R1 = R = R2, there is no rounding error and RN(R) = RZ(R) = RP(R) = RM(R). Otherwise Floating.Inexact is signaled and the value returned is determined from the table below, assuming that R1 < R < R2.

Rounding Mode	Result
RN	the nearer of R1 and R2. If equidistant then the value whose least significant bit is Zero.
RZ	the smaller of R1 and R2 in magnitude.
RP	R2
RM	R1

30.6. Infinity Arithmetic. The Nebula floating point system supports two modes of infinity arithmetic. The first mode, affine, specifies $-\infty < +\infty$. In projective mode $-\infty = +\infty$. The mode for infinity arithmetic is specified by bit 18 in the ASR. Zero specifies projective mode; One specifies affine mode.

30.7. Floating-Point Instructions. All floating-point instructions follow the same steps during execution. These steps are:

1. Fetch source operands - if either operand is a trapping NaN then raise Invalid.Operation exception.
2. Compute preliminary result.
3. Round and Check Floating.Underflow (order optional).
4. Check Floating.Inexact and Floating.Overflow.
5. Return result.

The compare instruction (CMPF) is the only floating point instruction that does not return a result so it only goes through Step 1 above.

Floating-Point Addition

ADDF	R <- A + B	80, A, B, R
	R <- R + A	81, A, R

Operand types:

A, B, R 32-bit, 64-bit floating point

Condition codes:

Z <- R Eql 0
N <- Sign bit of R
T <- Set as described in section 30.4
C Unaffected

Program exceptions:

Invalid.Operation
Floating.Underflow
Floating.Overflow
Floating.Inexact

Description:

The first operand is added to the second operand. In two operand addition the result is placed in the second operand. In three operand addition the result is placed in the third operand. If either operand was normalized the result will be also normalized unless Floating.Underflow occurs.

An Invalid.Operation condition exists whenever either operand is a trapping NaN, when the operation is $(+\infty) + (-\infty)$ or $(-\infty) + (+\infty)$ in Affine mode, or when both operands are infinity in Projective mode. If the exception is not enabled, the result shall be a non-trapping NaN.

A Floating.Underflow condition exists when the intermediate result's exponent is less than the destination format's minimum. If the exception is not enabled, then the intermediate result is denormalized (fraction shifted right and exponent incremented by one for each bit shift) until the exponent is at the destination format's minimum.

A Floating.Overflow condition exists if the result's exponent value would be greater than or equal to the maximum exponent value of the destination format and the result is not infinity or NaN. If the exception is not enabled, then the final result depends on the rounding mode and the sign of the intermediate result. The final result is set to infinity with the sign of the intermediate result if the rounding mode is RN, RZ, (RP and the intermediate result is positive) or (RM and the intermediate result is negative). Otherwise the final result is the largest magnitude normalized number representable in the destination.

A Floating.Inexact condition exists if the delivered result is not exactly equal to the intermediate result and no other exceptions have been raised.

Operands may be either single or double floating-point numbers.

Special Cases:

When a result is to be returned and either operand is a NaN then the result will be the same NaN (converted to non-trapping if necessary). If both operands are NaNs then the result will be the first operand.

If both operands are zero then the result is +0 in rounding modes RN, RZ, RP, or if both operands are +0. The result is -0 in mode RM or if both operands are -0.

If one operand is infinity then the result is the same infinity.

In Affine mode, $(+\infty) + (+\infty) = (+\infty)$ and $(-\infty) + (-\infty) = (-\infty)$. An Invalid.Operation condition exists if the operation is $(+\infty) + (-\infty)$ or $(-\infty) + (+\infty)$.

In Projective mode if both operands are infinity then an Invalid.Operation condition exists.

Floating-Point Subtraction

SUBF	R <- B - A	82, A, B, R
	R <- R - A	83, A, R

Operand types:

A, B, R   32-bit, 64-bit floating point

Condition codes:

Z <- R Eql 0
N <- Sign bit of R
T <- Set as described in section 30.4
C Unaffected

Program exceptions:

Invalid.Operation
Floating.Underflow
Floating.Overflow
Floating.Inexact

Description:

The first operand is subtracted from the second operand. In the two operand version the result is placed in the second operand. In the three operand version the result is placed in the third operand.

The execution of the subtract operation is identical to that of the add operation except that the sign bit of the first operand is first inverted.

Operands may be either single or double floating-point numbers.

Special Cases:

See ADDF

Floating-Point Multiplication

MULF	R <- B x A	84, A, B, R
	R <- R x A	85, A, R

Operand types:
A, B, R 32-bit, 64-bit floating point

Condition codes:
Z <- R Eql 0
N <- Sign bit of R
T <- Set as described in section 30.4
C Unaffected

Program exceptions:
Invalid.Operation
Floating.Underflow
Floating.Overflow
Floating.Inexact

Description:
The second operand is multiplied by the first operand. In two operand multiplication the result is placed in the second operand. In the three operand operation the result is placed in the third operand.

An Invalid.Operation condition exists whenever either operand is a trapping NaN or when one operand is 0 and the other is infinity. If the exception is not enabled, then the result shall be a non-trapping NaN.

A Floating.Underflow condition exists when the intermediate result's exponent is less than the destination format's minimum. If the exception is not enabled, then the intermediate result is denormalized (fraction shifted right and exponent incremented by one for each bit shift) until the exponent is at the destination format's minimum.

A Floating.Overflow condition exists if the result's exponent value would be greater than or equal to the maximum exponent value of the destination format and the result is not infinity or NaN. If the exception is not enabled, then the final result depends on the rounding mode and the sign of the intermediate result. The final result is set to infinity with the sign of the intermediate result if the rounding mode is RN, RZ, (RP and the intermediate result is positive) or (RM and the intermediate result is negative). Otherwise the final result is the largest magnitude normalized number representable in the destination.

A Floating.Inexact condition exists if the delivered result is not exactly equal to the intermediate result and no other exceptions have been raised.

Operands may be either single or double floating-point numbers.

Special Cases:
When a result is to be returned and either operand is a NaN then the result will be the same NaN (converted to non-trapping if necessary). If both operands are NaNs then the result will be the first operand.

If one operand is zero and the other is a finite number or if both operands are zero then the result is zero with the proper sign.

If one operand is zero and the other is infinity then an Invalid.Operation condition exists. If the exception is not enabled, then the result shall be a non-trapping NaN.

If either operand is infinity and the other is a finite number or both operands are infinity, then the result is infinity with the sign bit equal to the exclusive OR of the operands' sign bits.

Floating-Point Division

DIVF	R<- B / A	86, A, B, R
	R<- R / A	87, A, R

Operand types:

A, B, R   32-bit, 64-bit floating point

Condition codes:

Z <- R Eql 0
N <- Sign bit of R
T <- Set as described in section 30.4
C Unaffected

Program exceptions:

Invalid.Operation
Divide.By.Zero
Floating.Underflow
Floating.Overflow
Floating.Inexact

Description:

The second operand (the dividend) is divided by the first operand (the divisor). In two operand division the result is placed in the second operand. In the three operand operation the result is placed in the third operand.

An Invalid.Operation condition exists when either operand is a trapping NaN, when both operands are zero, or when both operands are infinity. If the exception is not enabled, then the result shall be a non-trapping NaN.

A Divide.By.Zero condition exists if the divisor is zero and the dividend is a non-zero finite number. If the exception is not enabled, then the result is infinity with the proper sign.

A Floating.Underflow condition exists when the intermediate result's exponent is less than the destination format's minimum. If the exception is not enabled, then the intermediate result is denormalized (fraction shifted right and exponent incremented by one for each bit shift) until the exponent is at the destination format's minimum.

A Floating.Overflow condition exists if the result's exponent value would be greater than or equal to the maximum exponent value of the destination format and the result is not infinity or NaN. If the exception is not enabled, then the final result depends on the rounding mode and the sign of the intermediate result. The final result is set to infinity with the sign of the intermediate result if the rounding mode is RN, RZ, (RP and the intermediate result is positive) or (RM and the intermediate result is negative). Otherwise the final result is the largest magnitude normalized number representable in the destination.

A Floating.Inexact condition exists if the delivered result is not exactly equal to the intermediate result and no other exceptions have been raised.

Operands may be either single or double floating-point numbers.

Special Cases:

When a result is to be returned and either operand is a NaN then the result will be the same NaN (converted to non-trapping if necessary). If both operands are NaNs then the result will be the first operand.

If both operands are zero or both operands are infinity then an Invalid.Operation condition exists and the result is a non-trapping NaN.

If the dividend is zero and the divisor is not then the result is zero with the proper sign.

If the divisor is 0 and the dividend is a non-zero finite number then a Divide.By.Zero condition exists and the result is infinity with the proper sign.

If the dividend is infinity and the divisor is not infinity or NaN then the result is infinity with the proper sign.

If the divisor is infinity and the dividend is a non-zero finite number then the result is 0 with the proper sign.

Negate Floating

NEGF	R <- (-A)	8A, A, R
	R <- (-R)	8B, R

Operand types:

A, R 32-bit, 64-bit floating point

Condition codes:

Z <- R Eql 0
N <- Sign bit of R
T <- Set as described in section 30.4
C Unaffected

Program exceptions:

Invalid.Operation
Floating.Underflow
Floating.Overflow
Floating.Inexact

Description:

In the two operand version the first operand is placed in the second operand with the sign bit inverted. *In the single operand version the sign bit of the operand is inverted.*

An Invalid.Operation condition exists whenever the source operand is a trapping NaN. If the exception is not enabled, then the result shall be a non-trapping NaN.

A Floating.Underflow condition exists when the number to be put in the second operand has an exponent less than the destination format's minimum. This can only happen if the source operand is a double floating number and the destination is single floating. If the exception is not enabled, then the intermediate result is denormalized (fraction shifted right and exponent incremented by one for each bit shift) until the exponent is at the destination format's minimum.

A Floating.Overflow condition exists if the result's exponent value would be greater than or equal to the maximum exponent value of the destination format and the result is not infinity or NaN. This can only happen if the source operand is a double floating number and the destination is single floating. If the exception is not enabled, then the final result depends on the rounding mode and the sign of the intermediate result. The final result is set to infinity with the sign of the intermediate result if the rounding mode is RN, RZ, (RP and the intermediate result is positive) or (RM and the intermediate result is negative). Otherwise the final result is the largest magnitude normalized number representable in the destination.

A Floating.Inexact condition exists if the delivered result is not exactly equal to the intermediate result and no other exceptions have been raised. This can only happen if the source operand is a double floating number and the destination is single floating.

Operands may be either single or double floating-point numbers.

Special Cases:

If the first operand is a NaN the sign bit will be inverted.

Convert Integer to Floating

FLOAT R <- A 8C, A, R

Operand types:

A 8-bit, 16-bit, 32-bit signed integer
R 32-bit, 64-bit floating point

Condition codes:

Z <- R Eql 0
N <- Sign bit of R
T <- Set as described in section 30.4
C Unaffected

Program exceptions:

Floating.Inexact

Description:

The source operand is converted from a two's-complement integer format to a floating-point format and placed in the destination operand.

A Floating.Inexact condition exists if the destination does not have exactly the same value as the source. This can only happen when the source is a 32-bit integer and the destination is a 32-bit floating number.

The source operand is a two's-complement integer in byte, 16-bit or 32-bit size. The destination operand must be a 32-bit or 64-bit floating-point format.

Special Cases:

None

Convert Floating to Integer

FIX	R <- Integerized(A)	8D, A, R

Operand types:

A	32-bit, 64-bit floating point
R	8-bit, 16-bit, 32-bit signed integer

Condition codes:

Z <- R Eql 0
N <- R Lss 0
T <- Set if the result is not representable in R, otherwise cleared
C Unaffected

Program exceptions:

Invalid.Operation
Truncation

Description:

The source operand is converted from a floating-point format to an integer format and placed in the destination operand. The floating-point number is truncated to an integer value. The source operand is not affected.

An Invalid.Operation condition exists if the source operand is a NaN or infinity. The destination operand is left unchanged.

The source operand can be a single or double floating-point number. The destination operand is an 8-bit, 16-bit or 32-bit signed integer.

Special Cases:

If the source operand is a NaN or infinity an Invalid.Operation condition exists. The destination operand is not affected.

If the result overflows the destination field, then excessive high-order bits are truncated and the T condition code is set.

Move Floating

MOVF B<- A 8E, A, B

Operand types:

A, B 32-bit, 64-bit floating point

Condition codes:

Z <- B Eql 0
N <- Sign bit of B
T <- Set as described in section 30.4
C Unaffected

Program exceptions:

Invalid.Operation
Floating.Underflow
Floating.Overflow
Floating.Inexact

Description:

The first operand is placed in the second operand. The first operand is not affected. This instruction can perform size conversion.

An Invalid.Operation condition exists if the source operand is a trapping NaN. If the exception is not enabled, then a non-trapping NaN is placed in the destination operand.

A Floating.Underflow condition exists when the number to be put in the second operand has an exponent less than the destination format's minimum. This can only happen if the source operand is a double floating number and the destination is single floating. If the exception is not enabled, then the intermediate result is denormalized (fraction shifted right and exponent incremented by one for each bit shift) until the exponent is at the destination format's minimum.

A Floating.Overflow condition exists if the source operand is double floating, the destination operand is single floating and the source operand's exponent is greater than or equal to the destination format's maximum and the number is not infinity or NaN. If the exception is not enabled, then the final result depends on the rounding mode and the sign of the intermediate result. The final result is set to infinity with the sign of the intermediate result if the rounding mode is RN, RZ, (RP and the intermediate result is positive) or (RM and the intermediate result is negative). Otherwise the final result is the largest magnitude normalized number representable in the destination.

A Floating.Inexact condition exists if the destination is not exactly equal to the source and no other exceptions have been raised.

Both operands are either 32-bit or 64-bit floating-point numbers.

Special Cases:

If the source operand is a trapping NaN and no exception is raised then a non-trapping NaN is placed in the destination operand.

If the operands are of different sizes then size conversion is implied.

Clear Floating

CLRF	A <- +0	8F, A

Operand type:

A — 32-bit, 64-bit floating point

Condition codes:

Z <- 1
N <- 0
T <- Set as described in section 30.4
C Unaffected

Program exceptions:

None

Description:

The operand is replaced by positive zero.

The operand can be either a 32-bit or 64-bit floating-point number.

Special Cases:

None

Compare Floating

CMPF	90, A, B

Operand type:

A, B — 32-bit, 64-bit floating point

Condition codes:

T <- Set as described in section 30.4
C Unaffected
If A and B are unordered then
    the Unordered exception is raised.
Otherwise:
    Z <- A Eql B
    N <- A Lss B

Program exceptions:

Invalid.Operation
Unordered

Description:

The two operands are compared and the condition codes are set according to the comparison. If the operands cannot be related using conventional relational operators then the Unordered exception is raised. Figure 30-3 specifies the compare operation.

X vs Y	-infinity affine	finite	+infinity affine	infinity Projective	NaN
-infinity affine	=	<	<	N/A	a
finite	>	b	<	a	a
+infinity affine	>	>	=	N/A	a
infinity Projective	N/A	a	N/A	=	a
NaN	a	a	a	a	=

a: The Unordered exception is raised.

b: The result is based on the result of X - Y.
Possible exception conditions are suppressed.

Figure 30-3: The Compare Operation

An Invalid.Operation condition exists if the source operand is a trapping NaN. If the **Invalid.Operation** exception is not enabled, then the **Unordered** exception is raised.

The operands can be either 32-bit or 64-bit floating-point numbers.

Special Cases:
If either operand is a NaN and the **Invalid.Operation** exception is not enabled, then the **Unordered** exception shall be raised.

If the mode for infinity arithmetic is projective and one operand is infinity then the **Unordered** exception shall be raised. If both operands are infinity then the comparison is equal.

Floating-Point Square Root

SQRTF   R <- SQRT(A)   89, A, R

Operand types:
A, R   32-bit, 64-bit floating point

Condition codes:
Z <- R Eql 0
N <- Sign bit of R
T <- Set as described in section 30.4
C Unaffected

Program exceptions:
Invalid.Operation
Floating.Inexact
Floating.Overflow
Floating.Underflow

Description:
The square-root of the first operand is placed in the second operand. The first operand is not affected.

An Invalid.Operation condition exists if the source operand is a trapping NaN, if the source operand is less than zero, or if the source operand is +infinity and the mode is projective. If the exception is not enabled, then the result shall be a NaN.

A Floating.Underflow condition exists when the intermediate result's exponent is less than the destination format's minimum. If the exception is not enabled, then the intermediate result is denormalized (fraction shifted right and exponent incremented by one for each bit shift) until the exponent is at the destination format's minimum.

A Floating.Overflow condition exists if the result's exponent value would be greater than or equal to the maximum exponent value of the destination format and the result is not infinity or NaN. If the exception is not enabled, then the result depends on the rounding mode. The result is set to positive infinity if the rounding mode is RN, RZ, RP. Otherwise the result is the largest magnitude normalized number representable in the destination.

A Floating.Inexact condition exists if the delivered result is not exactly equal to the intermediate result and no other exceptions have been raised.

Operands may be either single or double floating-point numbers.

Special Cases:
When a result is to be returned and the source operand is a NaN then the result will be the same NaN (converted to non-trapping if necessary).

The square-root of $+\infty$ in affine mode is $+\infty$. The square root of ∞ in projective mode creates an Invalid.Operation condition.

The square root of -0 is -0.

Absolute Value Floating

ABSF R <- |A| 91, A, R

Operand types:

A, R 32-bit, 64-bit floating point

Condition codes:

Z <- R Eql 0
N <- Sign bit of R
T <- Set as described in section 30.4
C Unaffected

Program exceptions:

Invalid.Operation
Floating.Underflow
Floating.Overflow
Floating.Inexact

Description:

The first operand is placed in the second operand with the sign bit set to zero. The first operand is not affected.

An Invalid.Operation condition exists if the source operand is a trapping NaN. If the exception is not enabled, then a non-trapping NaN is placed in the destination operand.

A Floating.Underflow condition exists when the number to be put in the second operand has an exponent less than the destination format's minimum. This can only happen if the source operand is a double floating number and the destination is single floating. If the exception is not enabled, then the intermediate result is denormalized (fraction shifted right and exponent incremented by one for each bit shift) until the exponent is at the destination format's minimum.

A Floating.Overflow condition may occur if the source operand is double floating and the destination operand is single floating. This condition exists when the source operand's exponent is greater than or equal to the destination format's maximum and the number is not infinity or NaN. If the exception is not enabled, then the result depends on the rounding mode. The result is set to positive infinity if the rounding mode is RN, RZ, RP. Otherwise the result is the largest magnitude normalized number representable in the destination.

A Floating.Inexact condition exists if the delivered result is not exactly equal to the intermediate result and no other exceptions have been raised. This can only happen if the source operand is a double floating number and the destination is single floating.

Both operands are either 32-bit or 64-bit floating-point numbers.

Special Cases:

If the source operand is a trapping NaN and the **Invalid.Operation** exception is not enabled then a non-trapping NaN, with the sign bit zero, is placed in the destination operand.

If the operands are of different sizes then size conversion is implied.

Remainder Floating

REMF	R <- B REM A	88, A, B, R

Operand type:
A, B, R   32-bit, 64-bit floating point

Condition codes:
Z <- R Eql 0
N <- Sign bit of R
T <- Set as described in section 30.4
C Unaffected

Program exceptions:
Invalid.Operation
Floating.Underflow
Floating.Overflow
Floating.Inexact

Description:
The result of the operation B REM A is placed in the destination operand. The remainder is defined regardless of the rounding mode by the following relation when A NEQ 0:

$$R = B - (A \times n) \text{ where } n \text{ is the integer nearest } B / A. \text{ If } |n - B / A| = ½ \text{ then } n \text{ is even.}$$

An Invalid.Operation condition exists if:
1. both source operands are zero.
2. the dividend is a finite number and the divisor is zero.
3. the dividend is infinity and the divisor is zero.
4. the dividend is infinity and the divisor is a finite number.
5. both source operands are infinity.
6. either source operand is a trapping NaN.

If the exception is not enabled, then the result shall be a non-trapping NaN.

A Floating.Underflow condition exists when the intermediate result's exponent is less than the destination format's minimum. If the exception is not enabled, then the intermediate result is denormalized (fraction shifted right and exponent incremented by one for each bit shift) until the exponent is at the destination format's minimum.

A Floating.Overflow condition exists if the result's exponent value would be greater than or equal to the maximum exponent value of the destination format and the result is not infinity or NaN. If the exception is not enabled, then the final result depends on the rounding mode and the sign of the intermediate result. The final result is set to infinity with the sign of the intermediate result if the rounding mode is RN, RZ, (RP and the intermediate result is positive) or (RM and the intermediate result is negative). Otherwise the final result is the largest magnitude normalized number representable in the destination.

A Floating.Inexact condition exists if the delivered result is not exactly equal to the intermediate result and no other exceptions have been raised.

Operands may be either single or double floating-point numbers.

Special Cases:

If the divisor is zero and the dividend is NaN, or the dividend is infinity and the divisor is a NaN, then an Invalid.Operation condition exists.

If the dividend is zero and the divisor is not zero or NaN the result is zero.

If the divisor is infinity and the dividend is a finite number then the result is the dividend.

If either operand is a NaN and the Invalid.Operation exception is not enabled the result is the same NaN (converted to non-trapping if necessary). If both operands are NaN's then the result is the second operand.

NOTE: The precision required from the REMF instruction can lead to very long execution times for many implementations in those cases where B is extremely large and A is extremely small. Since this potentially long execution time may preclude the locking out of interrupts during REMF execution, and since REMF is NOT required to resume from the point of interruption, continued restarting of REMF in some programming environments could effectively block further execution of a task that encounters REMF with large B and small A.

Round to Integer

RNDI   R <- A   92, A, R

Operand types:
A, R   32-bit, 64-bit floating point

Condition codes:
Z <- R Eql 0
N <- Sign bit of R
T <- Set as described in section 30.4
C Unaffected

Program exceptions:
Invalid.Operation
Floating.Overflow
Floating.Inexact

Description:
The first operand is rounded to an integer value and then placed in the second operand. The first operand is not affected. If the source's exponents is so large that there are no (zero or non-zero) significant fractions bits, the result will be set to the source.

An Invalid.Operation condition exists if the source operand is a trapping NaN. If the exception is not enabled, then a non-trapping NaN is placed in the destination operand.

A Floating.Overflow condition exists if the source operand is double floating, the destination operand is single floating and the source operand's exponent is greater than or equal to the destination formats maximum and the number is not infinity or NaN. If the exception is not enabled, then the final result depends on the rounding mode and the sign of the intermediate result. The final result is set to infinity with the sign of the intermediate result if the rounding mode is RN, RZ, (RP and the intermediate result is positive) or (RM and the intermediate result is negative). Otherwise the final result is the largest magnitude normalized number representable in the destination.

A Floating.Inexact condition exists if the delivered result is not exactly equal to the intermediate result and no other exceptions have been raised. This can only happen if the source operand is a double floating number and the destination is single floating.

Both operands are either 32-bit or 64-bit floating-point numbers.

Special Cases:
If the source is zero, infinity, or NaN the destination will be set to the same value (trapping NaNs are converted to non-trapping).

31. Opcode Allocation

Nebula Opcode Allocation

	0X	1X	2X	3X	4X	5X	6X	7X	8X	9X
X0	LTASK	BCS	BTS	ADD	MOV	MULFIX	DEC	PCHECK	ADDF	CMPF
X1	STASK	BCS	BTS	ADD	MOVL	DIVFIX	MOVA	PUSH	ADDF	ABSF
X2	TSTART	BCC	BTC	SUB	CMP	INC2	JUMP	SCHECK	SUBF	RNDI
X3	PSTART	BCC	BTC	SUB	CMPU	INC4	JSR	POP	SUBF	MOVM
X4	TRAISE	BEQL	EQL	MUL	TEST	INC8	CALL	SETBIT	MULF	MOVBK
X5	PRAISE	BEQL	WAIT	MUL	CLR	INC	CALLU	CLRBIT	MULF	CMPBK
X6	TINIT	BNEQ	NEQ	DIV	NOT	ADDU	SVC	STOBIT	DIVF	MOVTR
X7	PINIT	BNEQ	LOOP	DIV	NOT	SUBU	RET	TSTBIT	DIVF	SCANB
X8	LPSW	BLSS	LSS	MOD	AND	MULU	RSR	STOREH	REMF	SETPRI
X9	SPSW	BLSS	IBLSS	REM	AND	DIVU	RAISE	RESET	SQRTF	
XA	REPENT	BGEQ	GEQ	NEG	OR	CMPWB	ECODE	SETCC	NEGF	
XB	MAP	BGEQ	DBGEQ	NEG	OR	RANGE	EXCEPT	SIZE	NEGF	
XC	SETSEG	BLEQ	LEQ	ABS	XOR	SCALE	ERET	CASE	FLOAT	
XD	WINDOW	BLEQ	IBLEQ	EXCH	CMPS	ROT	ERP	SBF	FIX	
XE	BR	BGTR	GTR	ADDC	EMUL	LSH	NOP	LBFS	MOVF	
XF	BR	BGTR	DBGTR	SUBC	EDIV	COB	BREAK	LBF	CLRF	

Reserved For:	Opcodes	Number Reserved
Nebula Control Board	99 : CF, F0 : F9	65
Applications Use	D0 : DF	16
Compiler Runtime	E0 : EF	16
Hardware Implementor	FA : FE	5
Extension to 2-byte opcodes	FF	1

Opcodes reserved for applications and Compiler runtime use shall initiate a vectored OPEX call (as described in section 8.5) in all future 1862 implementations. In other words, these opcodes shall be reserved for software systems use.

Opcodes reserved for hardware implementors shall produce implementation dependent unpredictable results.

Opcodes reserved to the Nebula Control Board shall be assigned by the control board at their discretion. The codes may be assigned for software and/or hardware use. Such assignments shall become part of the standard.

The opcode FF shall be reserved by the control board for possible future use in establishing two-byte opcodes.

31.1. Unimplemented Opcodes. Unimplemented opcodes (OPcode EXceptions or OPEXs) shall cause a vector call using the opcode as an index through the OPEX vector registers. Refer to section 8.5.

32. Notes

32.1. Prime Item Specification considerations. Several items have been identified that should be considered while preparing a prime item specification for a Nebula computer.

- Number of memory map segments - the standard specifies a minimum for the number of segments that must be supported by the hardware for a Nebula computer. However, depending on the use intended for a particular implementation and the package constraints, a larger number of segments may be desirable.

- IPL format - The standard specifies how an IPL mechanism will interact with the architecture, but no definition of the IPL format is given.

- Remotely signaled halt - for some multiprocessor and distributed processing applications, it may be desirable to have the ability for one processor to halt another. This provision must be made at the interface level.

- Use of WINDOW - the window instruction is designed to allow diagnostic access to implementation mechanisms that lie below the architecture level. It may be desirable to specify some specific functionality that is required for specific systems.

- Implementor's opcodes - a set of opcodes are reserved for the implementor. It may be desirable to specify specific functions required for a given implementation.

- I/O register blocks and vectors - space for IOC register blocks and IOC interrupt vectors has been reserved. It may be desirable to specify specific locations within that reserved area, especially if the new implementation must match an already existing implementation.

- Initial 1K of memory address space - the first 1k of physical memory space has been allocated for special uses. Part of that space is specified for the architecture's use. The rest is intended for additional device vectors and for implementation scratch areas. It may be desirable to specify the uses for that space.

- MIL-STD-1553 decisions - there are a number of decisions involved in the use of MIL-STD-1553.

  - The RT mode commands are optional. Implementation of these commands depends on the features supported in the particular MIL-STD-1553 bus configuration.

  - The number of redundant busses is implementation dependent.

  - Optional features include broadcast mode, dynamic bus control, and service request among others.

32.2. Implementation dependencies. The following table is a list of items that are allowed to differ from one implementation to another.

Implementation Dependencies in the Standard

Description	Section	Page
Use of PSW bits 2:3	6.3	19
Use of ASR bits 1:7	7.2	21
Representation of context stacks	8.1.3	24
Exact location referenced by context pointer	8.1.5	24
Setting of PSW bits 2:3 for new PSW	8.3.1	25
Size and format of parameter descriptors	8.4.4	31
Exception handler state encoding	9.1	34
Order of acceptance of I/O interrupts of equal priority	11.1	39
Ability to detect hard or soft memory errors	11.6	40
Halt required by Reset function	11.9	42
Definition of BIT traps	11.10	42
Implementation virtual address space	12.1	44
Number of hardware supported map segments	12.2.2	44
Effect of self-modifying code	12.2.5.1	47
Memory map cacheing mechanisms	12.3.1	48
Effect of aliasing of physical addresses	12.3.2	48
Subseting of memory management	12.6	49
Channel configuration register definition	13.1	50
Recognition of access to Program counter	13.2.1	51
Recognition of access to message pointer	13.2.2	51
Channel status register bits 2:14	13.2.4	51
Optional RT mode commands	13.9.5	67
Base address of IOC register blocks	13.11	69
Implementation reserved IOC registers	13.11	69
Device vector assignments and use of low memory space	15.1	71
Result of illegal access to I/O space registers	15.2	72
Result of writing context and map pointers in I/O space	15.2	72
Trap or exception chosen within instruction	16.2	75
Order of emulation between "next"s in instructions	17.1	78
LTASK method of forcing consistency	25	126
STASK method of forcing context cache to memory	25	127
Information stored for interruptible instruction	27	135
Access by MOVBK when Src = Dest	27	137
Definition of action by WINDOW instruction	29	148
REPENT method of forcing consistency	29	149
SETSEG action on illegal segment specifier address	29	151
SETSEG information transmitted to IOC for Virtual addressing	29	152
Exact time of floating point underflow check	30.4.4	158
Use of opcodes FA:FE	31	177

CUSTODIAN: ARMY-CR, AIR FORCE-10

PREPARING ACTIVITY: ARMY-CR

PROJECT: IPSC-0153-01

Index

ABS 92
ABSF 173
Absolute Value 92
Absolute Value Floating 173
Active context stack 24
ADD 59, 82
Add to Address 60
Add with Carry 91
ADDC 91
ADDF 160
Address operands 8
Addressing.Error 52, 55, 68
ADDTA 60
ADDU 92
AND 96
Architectural virtual address space 43
Arithmetic Scale 98
Assembler notation 4
Auxiliary Status Register (ASR) 21, 42, 157, 158, 159

BCASE 62
BCC 113
BCS 113
BEQL 110
BGEQ 112
BGTR 112
Bit Case 62
Bit.Field.Size 35, 140
BLEQ 111
BLSS 111
BLSSIO 61
BNEQ 110
BNEQIO 61
BR 109
Branch 61, 109
Branch on Carry Clear 113
Branch on Carry Set 113
Branch on Equal 110
Branch on Greater than 112
Branch on Greater than or Equal 112
Branch on Less than 111
Branch on Less than or Equal 111
Branch on Not Equal 110
Branch on Truncate Clear 114
Branch on Truncate Set 114
Break 35, 37, 145
BRIO 61
BTC 114
BTS 114

Cacheing 3, 24
CALL 121

Call Procedure 121
Call Supervisor 124
Call Unprivileged Procedure 122
Call.Break 35, 37, 38
CALLU 122
Case 61, 115
CASEIO 61
Channel control register 5, 50, 151
Check Privilege Rights 153
Check Supervisor Rights 153
Clear 102
Clear Bit 143
Clear Floating 170
CLR 102
CLRBIT 143
CLRF 170
CMP 105
CMPBK 136
CMPF 170
CMPS 77, 106
CMPU 105
CMPWB 107
COB 97
Compare (Sign Extended) 105
Compare and Swap 106
Compare Block 136
Compare Floating 170
Compare Unsigned 105
Compare within Bounds 107
Compare within Bounds and Take Exception (Range Check) 107
Compound modes 7
Conceptual order 73
Context pointer 22, 24, 39, 40, 42, 72, 127
Context stack 22
Context.Alignment 35, 126
Context.Base 35, 146, 153
CONTROL 63, 64, 65
Convert Floating to Integer 168
Convert Integer to Floating 167
Count One Bits 97

DBGEQ 119
DBGTR 120
DEC 90
Decrement 90
Decrement and Branch on Greater 120
Decrement and Branch on Greater than or Equal to 119
DIV 84
DIVF 164
DIVFIX 89
Divide Fixed Point 89
Divide.By.Zero 21, 35, 156, 158, 164
DIVU 94

EAE 20, 78, 81
ECODE 131

EDIV 87
EMUL 86
EQL 108
ERET 33, 34, 38, 133, 134
ERP 33, 34, 37, 38, 134
EXCEPT 34, 132
Exception 19, 20, 21, 22, 25, 34
Exception Return 133
Exception Return and Propagate 134
EXCH 103
Exchange 103
Extended Integer Divide 87
Extended Integer Multiplication 86

FIX 168
FLOAT 167
Floating-Point Addition 160
Floating-Point Division 164
Floating-Point Multiplication 162
Floating-Point Square Root 172
Floating-Point Subtraction 161
Floating.Inexact 21, 35, 156, 158, 159, 160, 161, 162, 164, 166, 167, 169, 172, 173, 174, 176
Floating.Overflow 21, 35, 156, 158, 159, 160, 161, 162, 164, 166, 169, 172, 173, 174, 176
Floating.Underflow 21, 35, 156, 158, 159, 160, 161, 162, 164, 166, 169, 172, 173, 174

GEQ 108
GTR 108

Halt 62

I/O 50
I/O space 3, 5, 21, 22, 43, 44, 46, 49, 50, 51, 52, 66, 69, 70, 72
IADD 59
IADDL 59
IAND 60
IANDL 60
IBLEQ 117
IBLSS 118
Illegal.Address 8, 35
Illegal.Divisor 35, 84, 85, 86, 87, 89, 94, 116
Illegal.Mode 7, 18, 35
Illegal.Opcode 55
Illegal.Operation 50, 55, 63, 64, 65
Illegal.Parameter 14, 20, 27, 35
Illegal.Register 7, 20, 35, 103, 104, 124, 125
Illegal.Write 9, 10, 27, 31, 35
Implementation dependent 3
Implementation virtual address space 43
INC 90
INC2 90
INC4 90
INC8 90
Increment 90
Increment and Branch on Less than 118
Increment and Branch on Less than or Equal to 117
Infinity 21, 155, 158, 159, 160, 162, 164, 166, 168, 169, 170, 172, 173, 174, 176

Inheritance of registers 26
Initiate RT to RT transfer 66
Initiate Task 130
Inline literal 78
Instruction.Break 35, 37, 38
INT 62
Integer Addition 82
Integer Division 84
Integer Modulus 85
Integer Multiplication 83
Integer Negate 85
Integer Subtraction 82
Interface Control 63, 64, 65
Interrupt 5, 19, 20, 22, 25, 26, 33, 35, 38, 39, 50, 51, 54, 55, 62, 63, 67, 68, 70, 71, 77, 135, 152, 174
Interrupt.Priority 55, 62
Interruptible 3
Invalid.Access 47, 48
Invalid.Operation 21, 35, 155, 156, 157, 158, 159, 160, 161, 162, 164, 166, 168, 169, 170, 172, 173, 174, 176
Invalid.Segment 45, 48
Invalid.Supervisor 47, 48, 121
IOC.Active 51, 55, 68
IOC.Busy 35, 151
IOR 60
IORL 60
IPL 40, 42, 44, 63, 64, 71
ISUB 60

JSR 124
Jump 109
Jump to Subroutine 124

Kernel 19, 26, 39
Kernel Context Pointer 22, 24, 39, 40, 42, 72
Kernel Context Stack 19, 22, 39, 40, 41, 42, 48, 128, 129

LBF 142
LBFS 141
LEQ 108
LMP 61
LOAD 59
Load Bit Field (Sign Extended) 141
Load Bit Field (Zero Extended) 142
Load Message Pointer 61
Load PSW 146
Load Status 59
Load Task 128
LOADL 59
LOADST 59
Logical AND 60, 96
Logical Exclusive Or 97
Logical Not 95
Logical OR 60, 96
Logical Shift 61, 100
Loop 116
LPSW 146
LSH 100

LSHFT 61
LSS 108
LTASK 22, 24, 44, 47, 48, 77, 126, 127

Main memory 5, 76, 77
MAP 150
Map Pointer 39, 40, 44, 72, 126, 127, 149
Map Virtual Address 150
Maxreg 7, 9, 11, 12, 20
Memory.Error 54, 55
Message.Alignment 51, 55, 68
MOD 85
MOV 101
MOVA 102
MOVBK 137
Move Address 102
Move Arithmetic (Sign Extended) 101
Move Block 137
Move Floating 169
Move Logical (Zero Extended) 101
Move Multiple (Fill) 137
Move Translated 138
MOVF 169
MOVL 101
MOVM 137
MOVTR 138
MUL 83
MULF 162
MULFIX 88
Multiply Fixed Point 88
MULU 93

NaN (Not a Number) 155, 157, 159, 160, 162, 164, 166, 168, 169, 170, 172, 173, 174, 176
NEG 85
Negate Floating 166
NEGF 166
NEQ 108
NI 79
No Operation 145
NOP 145
NOT 95

Operand specifier 5, 7, 27, 29, 75, 136
Operand.Size 16, 17, 18, 35, 136, 137, 138, 139
OPEX 25, 26, 31, 38
OR 96

PCHECK 153
Physical address 37, 39, 40, 41, 42, 43, 44, 46, 48, 49, 54, 71, 126, 150, 151
Physical address space 5, 44, 49, 50
PINIT 20, 25, 26, 130
POP 104
Pop from SP Stack 104
PRAISE 34, 129
Priority 19, 39
Privilege 20, 22, 26, 31, 33, 41, 47, 153

Privilege.Violation 47, 48
Procedures 22
Processor Status Word (PSW) 19, 20, 22, 23, 24, 25, 27, 33, 35, 38, 39, 41, 42, 47, 70, 121, 122, 123, 128, 130, 133, 146, 148, 153, 157
Program.Alignment 51, 55
PSTART 128
PUSH 103
Push onto SP Stack 103

RAISE 34, 131
Raise Exception 131
RANGE 107
Range.Error 35, 107
RDTMSG 56
RDTMSGS 56
Read 56
Read to Message 56
READS 56
Register set 26
REM 86
Remainder 86
Remainder Floating 174
REMF 174
REPENT 47, 48, 77, 149
Replace Entry in Map 149
Reserved 3
Reset 154
RET 19, 20, 33, 37, 38, 40, 41, 42, 77, 123, 133, 148, 152
Return from Procedure 123
Return from Subroutine 125
RNDI 176
ROT 99
Rotate 99
Round to Integer 176
RSR 125
RT2RT 66

SBF 140
SCALE 98
Scan and Break 139
SCANB 139
SCHECK 153
Segment.Specifier 35, 151
Set Based on Condition 108
Set Bit 143
Set Condition Codes 147
Set Exception Handler Entry Address 132
Set I/O Segment 151
Set Priority Level 148
SETBIT 143
SETCC 147
SETPRI 148
SETSEG 151
Size 147
Specification.Error 3, 44, 126, 128, 129, 149
SPSW 146

SQRTF 172
Start Task 128
Start Task Setting Exception 129
STASK 22, 24, 77, 127
STOBIT 77, 144
Store 59
Store Bit Field 140
Store Exception Code 131
Store Exception Handler Address 132
Store PSW 146
Store Task 127
STOREH 132
SUB 82
SUBC 91
SUBF 161
Subtract 60
Subtract with Carry 91
SUBU 93
Supervisor 20, 25, 31, 42, 153
Supervisor Exception Handler 20, 25, 26, 33, 35, 37, 38, 71, 75, 133, 134, 145
Supervisor Map Pointer 39, 40, 44, 72, 149
Supervisor.Check 35, 153
SVC 25, 26, 31, 38, 124

Task Context Pointer 22, 24, 72, 127
Task Context Stack 22, 41, 128, 129
Task.Failure 25, 26, 35, 37, 133
Task.Load.Error 35, 126
TEST 108
Test and Store Bit 144
Test Bit 144
Test Integer 108
Timers 70
TINIT 20, 25, 26, 130
TRAISE 34, 129
Truncation 20, 35, 37, 78, 81
TSTART 128
TSTBIT 144

UDLE 20, 35, 37
Undefined 3
Unordered 35, 156, 170
Unpredictable 3
Unsigned Addition 92
Unsigned Division 94
Unsigned Multiplication 93
Unsigned Subtraction 93
User Map Pointer 44, 72, 126, 127, 149

Virtual address 22, 31, 43, 44, 46, 47, 48, 49, 51, 52, 54, 56, 58, 66, 67, 79, 132, 135, 150, 151, 153
Virtual address space 5, 23, 43, 44, 151

WAIT 152
Wait for Interrupt 152
WINDOW 148
Window into the Micromachine 148

WRFMSG 57
WRFMSGC 57
Write 57
Write from Message 57
Write Literal 58
WRITEC 57
WRLIT 58
WRLITC 58

XOR 97