Data Types and Addresses 8

CHAPTER 8
DATA TYPES AND ADDRESSES

This chapter describes the data types that the 80960SA/SB processor recognizes and the addressing modes that are available for accessing memory locations.

DATA TYPES

The processor defines and operates on the following data types:

- Integer (8, 16, 32, and 64 bits)
- Ordinal (8, 16, 32, and 64 bits)
- Real (32, 64, and 80 bits)
- Bit Field
- Decimal (ASCII digits)
- Triple-Word (96 bit)
- Quad-Word (128 bit)

NOTE

The real and decimal data types are not defined in the i960 architecture. They are supported in the 80960SB processor, but not in the 80960SA processor.

The integer, ordinal, real, and decimal data types can be thought of as numeric data types because some operations on these data types produce numeric results (e.g., add, subtract).

The remaining data types (bit field, triple word, and quad word) represent groupings of bits or bytes that the processor can operate on as a whole, regardless of the nature of the data contained in the group. These data types facilitate moving and operating on blocks of bits or bytes.

Integers

Integers are signed whole numbers, which are stored and operated on in two's complement format. The processor recognizes four sizes of integers: 8 bit (byte integers), 16 bit (short integers), 32 bit (integers), and 64 bit (long integers). Figure 8-1 shows the formats for the four integer sizes and Table 8-1 shows the ranges of values allowed for each size.

Ordinals

Ordinals are a general-purpose data type. The processor recognizes four sizes of ordinals: 8 bit (byte ordinals), 16 bit (short ordinals), 32 bit (ordinals), and 64 bit (long ordinals). Figure 8-1 shows the formats for the four ordinal sizes; Table 8-1 shows the ranges of numeric values allowed for each size.

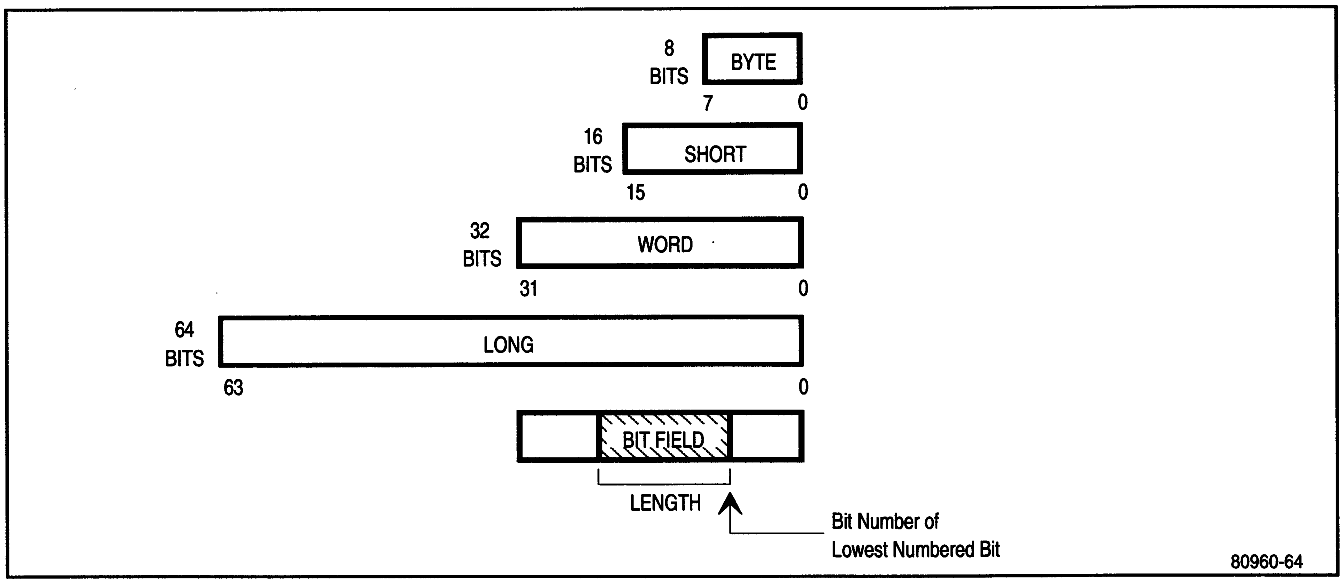

Figure 8-1: Integer, Ordinal, and Bit Field Format

Table 8-1: Integer, Ordinal, and Bit Field Range

Data Type	Size	Range	Decimal Equivalent
Integer	Byte Integer Short Integer Integer Long Integer	-2^{7} to $2^{7} - 1$ -2^{15} to $2^{15} - 1$ -2^{31} to $2^{31} - 1$ -2^{63} to $2^{63} - 1$	-128 to 127 -32,768 to 32,767 -2.14×10^{8} to 2.14×10^{8} -9.22×10^{16} to 9.22×10^{16}
Ordinal	Byte Ordinal Short Ordinal Ordinal Long Ordinal	0 to $2^{8} - 1$ 0 to $2^{16} - 1$ 0 to $2^{32} - 1$ 0 to $2^{64} - 1$	0 to 255 0 to 65,535 0 to 4.29×10^{8} 0 to 1.84×10^{16}
Bit and Bit Field	Bit Bit Field	0 to 1 0 - 31	0 to 1 Not Applicable

The processor uses ordinals for both numeric and non-numeric operations. For numeric operations, ordinals are treated as unsigned whole numbers. The processor provides several arithmetic instructions that operate on ordinals. For non-numeric operations, ordinals contain bit fields, byte strings, and Boolean values.

When ordinals are used to represent Boolean values, a 1_2 represents a TRUE and a 0_2 represents a FALSE.

Reals

Reals are floating-point numbers. The processor recognizes three sizes of reals: 32 bit (reals), 64 bit (long reals), and 80 bit (extended reals). The real-number format conforms to ANSI/IEEE Std. 754-1985, the IEEE Standard For Binary Floating-Point Arithmetic. Real numbers are discussed in greater detail in Chapter 10.

Bits and Bit Fields

The processor provides several instructions that perform operations on individual bits or fields of bits within an ordinal (32 bit) operand. Figure 8-1 and Table 8-1 illustrate and describe these data types.

An individual bit is specified for a bit operation by giving its bit number in the ordinal in which it resides. The least-significant bit of a 32-bit ordinal is bit 0; the most-significant bit is bit 31.

A bit field is a contiguous sequence of bits of from 0 to 32 bits in length within a 32-bit ordinal. A bit field is defined by giving its length in bits and the bit number of its lowest-numbered bit.

A bit field cannot span a register boundary.

Decimals

The processor provides three instructions that perform operations on decimal values when the values are presented in ASCII format. Figure 8-2 shows the ASCII format for decimal digits. Each decimal digit is contained in the least-significant byte of an ordinal (32 bits). The decimal digit must be of the form $0011dddd_2$, where $dddd_2$ is a binary-coded decimal value from 0 to 9. For decimal operations, bits 8 through 31 of the ordinal containing the decimal digit are ignored.

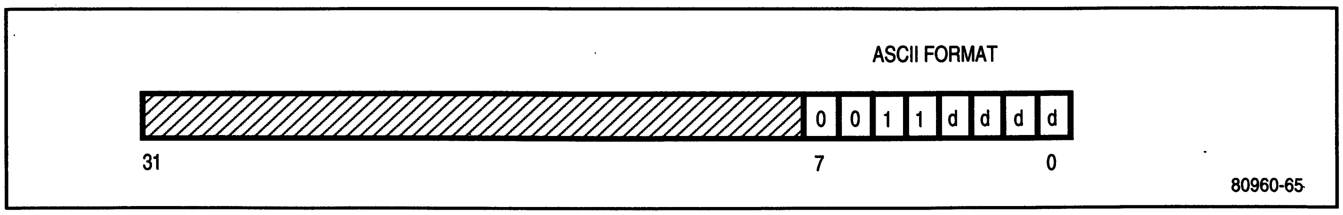

Figure 8-2: Decimal Format

Triple and Quad Words

Triple and quad words refer to consecutive bytes in memory or in registers: a triple word is 12 bytes and a quad word is 16 bytes. These data types facilitate the moving of blocks of bytes. The triple-word data type is useful for moving extended-real numbers (80 bits).

The quad-word instructions (**ldq**, **stq**, and **movq**) offer the most efficient way to move large blocks of data.

BYTE, WORD, AND BIT ADDRESSING

The processor provides instructions for moving blocks of data values of various lengths from memory to registers (load) and from registers to memory (store). The allowable sizes for blocks are bytes, half-words (2 bytes), words (4 bytes), double words, triple words, and quad words. For example, the **stl** (store long) instruction stores an 8-byte (double word) block of data in memory.

When a block of data is stored in memory, the least-significant byte of the block is stored at a base memory address and the more significant bytes are stored at successively higher addresses.

When loading a byte, half-word, or word from memory to a register, the least-significant bit of the block is always loaded in bit 0 of the register. When loading double words, triple words, and quad words, the least-significant word is stored in the base register. The more significant words are then stored at successively higher numbered registers. Double words, triple words, and quad words must also be aligned in registers to natural boundaries as described in Chapter 2 in the section titled "Register Alignment."

Bits can only be addressed in data that resides in a register. Bit 0 in a register is the least-significant bit and bit 31 is the most-significant bit.

LITERALS

The processor recognizes two types of literals (ordinal literals and floating-point literals), which can be used as operands in some instructions. An ordinal literal can range from 0 to 31 (5 bits). When an ordinal literal is used as an operand, the processor expands it to 32 bits by adding leading zeros. If the instruction defines an operand larger than 32 bits, the processor zero-extends the value to the operand size. If an ordinal literal is used in an instruction that requires integer operands, the processor treats the literal as a positive integer value.

The processor also recognizes two floating-point literals (+0.0 and +1.0). These floating-point literals can only be used with floating-point instructions. As with the ordinal literals, the processor converts the floating-point literals to the operand size specified by the instruction.

A few of the floating-point instructions use both floating-point and non-floating-point operands (e.g., the convert integer-to-real instructions). Ordinal literals can be used in these instructions for non-floating-point operands.

REGISTER ADDRESSING

A register may be used as an operand in an instruction by giving the register's number (e.g., g0, r5, fp3). Both floating-point and non-floating-point instructions can reference global and local registers in this way. However, floating-point registers can only be referenced in conjunction with a floating-point instruction.

MEMORY-ADDRESSING MODES

The processor offers 9 modes for addressing operands in memory. These modes are grouped as follows:

- Absolute
- Register Indirect
- Register Indirect with Index
- Index with Displacement
- IP with Displacement

Each addressing mode is used to reference a byte address in the processor's address space. Table 8-2 shows all the memory-addressing modes, a brief description of the elements of the address in each mode, and the assembly-code syntax for each mode.

Table 8-2: Addressing Modes

Mode	Description	Assembler Syntax
Absolute Offset	offset	exp
Register Indirect	abase	(reg)
Register Indirect with offset	abase + Offset	exp (reg)
Register Indirect with index	abase + (index*scale)	(reg) [reg*scale]
Register Indirect with index and displacement	abase + (index*scale) + displacement	exp (reg) [reg*scale]
Index with displacement	(index*scale) + displacement	exp [reg*scale]
IP with displacement	IP + displacement + 8	exp (IP)

where: *reg* is register and *exp* is expression

Absolute

Absolute addressing is used to reference a memory location directly as an offset from address 0 of the address space, ranging from -2^{31} to $2^{31} - 1$. Typically, an assembler will allow absolute addresses to be specified through arithmetic expressions (e.g., x + 44), symbolic labels, and absolute values.

At the machine-level, two absolute-addressing modes are provided, depending on the instruction format (i.e., MEMA or MEMB). For the MEMA format, the offset is an ordinal number ranging from 0 to 4096; for the MEMB format, the offset is an integer (called a displacement) ranging from -2^{31} to $2^{31} -1$. After evaluating an absolute address, the assembler will convert the address into an offset and select the appropriate machine-level instruction type and addressing mode. (The machine-level addressing modes and instruction formats are described in Appendix B.)

Register Indirect

The register indirect addressing modes allow an address to be specified with an ordinal value (32 bits) in a register or with an offset or a displacement added to a value in a register. Here, the value in the register is referred to as the address base (abase).

Again, an assembler will allow the offset and displacement to be specified with an expression or symbolic label, then evaluate the address to determine whether an offset or a displacement is appropriate.

Register Indirect with Index

The register indirect with index addressing modes allow a scaled index to be added to the value in a register. The index is specified by means of a value placed in a register. This index value is then multiplied by the scale factor. The allowable scale factors are 1, 2, 4, 8, and 16.

A displacement may also be added to the abase value and scaled index.

Index with Displacement

A scaled index can also be used with a displacement alone. Again, the index is contained in a register and is multiplied by a scaling constant before the displacement is added to it.

IP with Displacement

The IP with displacement addressing mode is often used with load and store instructions to make them IP relative.

Note that with this mode the displacement plus a constant of 8 is added to the IP of the instruction.

Examples;

The following examples show how these addressing modes are encoded in assembly language.

```
st   g4,xyz             # absolute; word from g4 stored at
                          memory location designated with
                          label xyz.

ldob (r3),r4            # register indirect; ordinal byte
                          from memory location given in
                          register r3 loaded into register
                          r4.
```

```
stl  g6,xyz(g5)       # register indirect with
                        displacement; double word from
                        g6,g7 stored at memory location
                        xyz + g5.

ldq  (r8)[r9*4],r4    # register indirect with index;
                        quad-word beginning at memory
                        location r8 + (r9 scaled by 4)
                        loaded into registers r4
                        through r7.

st  g3,xyz(g4)[g5*2]  # register indirect with index
                        and displacement; word in g3
                        loaded into memory location g4 +
                        xyz + (g5 scaled by 2).

ldis xyz[r12*1],r13   # index with displacement; load
                        short integer at memory location
                        xyz + r12 into r13.

st   r4,xyz(IP)       # IP with displacement; store word
                        in r4 at memory location IP + xyz.
```

Unaligned Memory Accesses

The i960 architecture requires that operands in memory be aligned on natural boundaries (i.e., half-words on half-word boundaries, word operands on word boundaries, double-word operands on double-word boundaries, and triple-word and quad-word operands on 64-byte boundaries). When an unaligned memory access is made, the architecture gives the processor two options: (1) raise an operation-unimplemented fault or (2) relax the requirement.

The 80960SA/SB processor uses the second option, and allow unaligned memory accesses. However, for compatibility with other processors based on the 80960 core architecture, unaligned memory accesses should be avoided.