3 Data Types and Addresses

CHAPTER 3
DATA TYPES AND ADDRESSES

This chapter defines the operand types and addressing modes supported by the MCS-96 architecture.

Chapter 2, "Introduction to the 8XC196KC/KD," provides an overview of the MCS-96 instruction set. It discusses differences between the 8XC196KC/KD instruction set and that of the 8096BH and offers guidelines for program development. Appendix A provides reference information for the 8XC196KC/KD instruction set. It includes descriptions of the instructions, hexadecimal opcodes, instruction lengths, execution times, and the relationships between Program Status Word (PSW) flags and the instructions. Appendix C provides a detailed description of the PSW.

3.1. OPERAND TYPES

The MCS-96 architecture supports a variety of data types likely to be useful in a control application. Where appropriate, this discussion uses the names adopted by the PLM-96 programming language. The name of an operand type is shown in all capitals, to avoid confusion. For example, a *BYTE* is an unsigned eight-bit variable, while a *byte* is an eight-bit unit of data of any type.

The following data types are available on the 8XC196KC/KD:

- BIT
- BYTE
- SHORT-INTEGER
- WORD
- INTEGER
- DOUBLE-WORD
- LONG-INTEGER

Table 3-1 provides an overview of these operand types. The remainder of this section discusses each one in detail.

Table 3-1. Operand Type Definitions

Operand Type	No. of Bits	Signed	Possible Values	Addressing Restrictions
BIT	1	No	True or False	As components of bytes
BYTE	8	No	0 through 255	None
SHORT-INTEGER	8	Yes	–128 through +127	None
WORD	16	No	0 through 65535	Even byte address
INTEGER	16	Yes	–32,768 through +32,767	Even byte address
DOUBLE-WORD *	32	No	0 through 4,294,967,295	Even byte address in on-chip Register File; evenly divisible by four **
LONG-INTEGER *	32	Yes	–2,147,483,648 through +2,147,483,647	Even byte address in on-chip Register File; evenly divisible by four **

* The 32-bit operands are supported only as the operand in shift operations, as the dividend in 32-by-16 divide operations, and as the product of 16-by-16 multiply operations.

** For consistency with Intel-provided software, you should adopt the PLM-96 conventions for addressing 32-bit operands. For more information, refer to "Software Standards and Conventions" in Chapter 2.

3.1.1. BIT Operand

A BIT is a single-bit operand that can take on the Boolean values, "true" and "false." In addition to the normal support for bits as components of BYTE and WORD operands, the 8XC196KC/KD provides a means for directly testing any bit in the internal Register File. The MCS-96 architecture requires that bits be addressed as components of BYTEs or WORDs. It does not support the direct addressing of bits that can occur in the MCS-51 architecture.

3.1.2. BYTE Operand

A BYTE is an unsigned, 8-bit variable that can take on values from 0 through 255. Arithmetic and relational operators can be applied to BYTE operands, but the result must be interpreted in modulo 256 arithmetic. Logical operations on BYTES are applied bitwise. Bits within BYTEs are labeled from 0 to 7; bit 0 is the least-significant bit. There are no alignment restrictions for BYTEs, so they may be placed anywhere in the MCS-96 address space.

3.1.3. SHORT-INTEGER Operand

A SHORT-INTEGER is an 8-bit, signed variable that can take on values from –128 through +127. Arithmetic operations that generate results outside the range of a SHORT-INTEGER set the overflow flags in the PSW. The numeric result is the same as the result of the equivalent operation on BYTE variables. There are no alignment restrictions on SHORT-INTEGERs, so they may be placed anywhere in the MCS-96 address space.

3.1.4. WORD Operand

A WORD is an unsigned, 16-bit variable that can take on values from 0 through 65535. Arithmetic and relational operators can be applied to WORD operands, but the result must be interpreted in modulo 65536 arithmetic. Logical operations on WORDs are applied bitwise. Bits within WORDs are labeled from 0 to 15; bit 0 is the least-significant bit.

WORDs must be aligned at even byte boundaries in the MCS-96 address space. The least-significant byte of the WORD is in the even byte address, and the most-significant byte is in the next higher (odd) address. The address of a WORD is that of its least-significant byte (the even byte address). WORD operations to odd addresses are not guaranteed to operate in a consistent manner.

3.1.5. INTEGER Operand

An INTEGER is a 16-bit, signed variable that can take on values from –32,768 through +32,767. Arithmetic operations that generate results outside the range of an INTEGER set the overflow flags in the PSW. The numeric result is the same as the result of the equivalent operation on WORD variables.

INTEGERs must be aligned at even byte boundaries in the MCS-96 address space. The least-significant byte of the INTEGER is in the even byte address, and the most-significant byte is in the next higher (odd) address. The address of an INTEGER is that of its least-significant byte (the even byte address). INTEGER operations to odd addresses are not guaranteed to operate in a consistent manner.

3.1.6. DOUBLE-WORD Operand

A DOUBLE-WORD is an unsigned, 32-bit variable that can take on values from 0 through 4,294,967,295. The MCS-96 architecture directly supports DOUBLE-WORD operands only as the operand in shift operations, as the dividend in 32-by-16 divide operations, and as the product of 16-by-16 multiply operations. For these operations, a DOUBLE-WORD variable must reside in the on-chip Register File and must be aligned at an address that is evenly divisible by four. The address of a DOUBLE-WORD is that of its least-significant byte (the even byte address). The least-significant word of the DOUBLE-WORD is always in the lower address, even when the data is in the stack. This means that the most-significant word must be pushed into the stack first.

DOUBLE-WORD operations that are not directly supported can be easily implemented with two WORD operations.

For consistency with Intel-provided software, you should adopt the PLM-96 conventions for addressing DOUBLE-WORD operands. (The PLM-96 conventions are discussed in "Software Standards and Conventions" in Chapter 2.)

3.1.7. LONG-INTEGER Operand

A LONG-INTEGER is a 32-bit, signed variable that can take on values from –2,147,483,648 through +2,147,483,647. The MCS-96 architecture directly supports DOUBLE-WORD operands only as the operand in shift operations, as the dividend in 32-by-16 divide operations, and as the product of 16-by-16 multiply operations. For these operations, a LONG-INTEGER variable must reside in the on-chip Register File and must be aligned at an address that is evenly divisible by four. The address of a LONG-INTEGER is that of its least-significant byte (the even byte address).

LONG-INTEGER operations that are not directly supported can be easily implemented with two INTEGER operations.

For consistency with Intel-provided software, you should adopt the PLM-96 conventions for addressing LONG-INTEGER operands. (The PLM-96 conventions are discussed in "Software Standards and Conventions" in Chapter 2.)

3.2. ADDRESSING MODES

Six basic addressing modes are used to access operands within the address space of the 80C196KC/KD:

- Register-Direct
- Indirect
- Indirect with Auto-Increment
- Immediate
- Short-Indexed
- Long-Indexed

Two other useful modes are Zero Register addressing and Stack Pointer Register addressing. Zero Register addressing combines the ZERO_REG with Long-Indexed addressing, allowing direct access to any location in memory. Stack Pointer Register addressing combines the SP with Indirect addressing to access the top of the stack and with Short-Indexed addressing to access data within the stack.

This section describes the addressing modes as they are handled by the hardware. An understanding of these details will help programmers to take full advantage of the architecture. The assembly language hides some of the details of how these addressing modes work. The "Assembly Language Addressing Mode Selections" section (see page 3-8) describes how the assembly language handles direct and indexed addressing modes.

3.2.1. Register-Direct Addressing

The Register-Direct addressing mode directly accesses a register from the 256-byte on-chip lower Register File. With windowing, this mode can also directly access the additional SFRs or the upper Register File (see Chapter 4, "Memory Partitions"). The register is selected by an 8-bit field within the instruction, and the register address must conform to the alignment rules for the operand type. Depending on the instruction, up to three registers can take part in the calculation.

Examples of Register-Direct Addressing:

```
ADD AX,BX,CX ; AX <-- BX + CX (ADD_3op)
MUL AX,BX    ; AX <-- AX * BX (MUL_2op)
INCB CL      ; CL <-- CL + 1
```

Definition of Temporary Registers:

AX, BX, and CX are 16-bit registers. CL is the low byte of CX.

3.2.2. Indirect Addressing

The Indirect addressing mode accesses an operand by placing its address in a WORD variable in the Register File. The calculated address must conform to the alignment rules for the operand type. Note that the indirect address can refer to an operand anywhere within the MCS-96 address space, including the Register File. An 8-bit field within the instruction selects the register that contains the indirect address. An instruction can contain only one indirect reference; any additional operands must be Register-Direct references.

Examples of Indirect Addressing:

```
LD AX,[AX]       ; AX <-- MEM_WORD(AX)
ADDB AL,BL,[CX] ; AL <-- BL + MEM_BYTE(CX) (ADDB_3op)
POP [AX]         ; MEM_WORD(AX) <-- MEM_WORD(SP)
                 ; SP <-- SP + 2
```

Definition of Temporary Registers:

AX, CX are 16-bit registers. AL is the low byte of AX. CL is the low byte of CX.

3.2.3. Indirect with Auto-Increment Addressing

Indirect with Auto-Increment addressing mode is the same as the Indirect mode, except that the WORD variable that contains the indirect address is incremented after it is used to address the operand. The least-significant bit of a WORD register distinguishes between indirect addressing with or without auto-increment. If the instruction operates on a BYTE or

SHORT-INTEGER, the indirect address variable is incremented by one. If the instruction operates on a WORD or INTEGER, the indirect address variable is incremented by two.

Examples of Indirect with Auto-Increment Addressing:

```
LD AX,[BX]+       ; AX <-- MEM_WORD(BX)
                  ; BX <-- BX + 2
ADDB AL,BL,[CX]+ ; AL <-- BL + MEM_BYTE(CX)
                  ; CX <-- CX + 1 (ADDB_3op)
PUSH [AX]+        ; SP <-- SP - 2
                  ; MEM_WORD(SP) <-- MEM_WORD(AX)
                  ; AX <-- AX + 2
```

Definition of Temporary Registers:

AX, BX, CX are 16-bit registers. AL is the low byte of AX. BL is the low byte of BX.

3.2.4. Immediate Addressing

Immediate addressing mode allows an operand to be taken directly from a field in the instruction. For operations on BYTE or SHORT-INTEGER operands, this is an 8-bit field. For operations on WORD or INTEGER operands, it is a 16-bit field. An instruction can contain only one Immediate reference; any additional operands must be Register-Direct references.

Examples of Immediate Addressing:

```
ADD AX,#340     ; AX <-- AX + 340 (ADD_2op)
PUSH #1234H     ; SP <-- SP - 2
                ; MEM_WORD(SP) <-- 1234H
DIVB AX,#10     ; AL <-- AX/10
                ; AH <-- AX MOD 10
```

Definition of Temporary Registers:

AX is a 16-bit register. AL is the low byte and AH is the high byte of AX.

3.2.5. Short-Indexed Addressing

In Short-Indexed addressing mode, the address of one of the operands is calculated from two 8-bit fields. One 8-bit field in the instruction selects a WORD variable in the Register File, which contains an address. The second 8-bit field in the instruction stream is sign-extended and summed with the WORD variable to form the effective address of the operand. The effective address can be up to 128 bytes before the address in the WORD variable and up to 127 bytes after it. An instruction can contain only one Short-Indexed reference; any remaining operands must be Register-Direct references.

Examples of Short-Indexed Addressing:

```
LD AX,12[BX]        ; AX <-- MEM_WORD(BX+12)
MULB AX,BL,3[CX]    ; AX <-- BL * MEM_BYTE(CX+3)
                    ; (MULB_3op)
```

Definition of Temporary Registers:

AX, BX, CX are 16-bit registers. BL is the low byte of BX.

3.2.6. Long-Indexed Addressing

The Long-Indexed addressing mode is like the Short-Indexed mode, except that a 16-bit field is taken from the instruction and added to the WORD variable to form the address of the operand. No sign extension is necessary. An instruction can contain only one Long-Indexed reference; any remaining operands must be Register-Direct references.

Examples of Long-Indexed Addressing:

```
AND AX,BX,TABLE[CX]     ; AX <-- BX AND MEM_WORD(TABLE+CX)
                        ; (AND_3op)
ST AX,TABLE[BX]         ; MEM_WORD(TABLE+BX) <-- AX
ADDB AL,BL,LOOKUP[CX]   ; AL <-- BL + MEM_BYTE(LOOKUP+CX)
                        ; (ADDB_3op)
```

Definition of Temporary Registers:

AX, BX, CX are 16-bit registers. AL is the low byte of AX. BL is the low byte of BX.

3.2.7. Zero Register Addressing

The first two bytes in the Register File constitute the Zero Register (ZERO_REG). These bytes are fixed at zero by the 80C196KC/KD hardware. In addition to providing a fixed source of the constant zero for calculations and comparisons, the Zero Register can be used as the WORD variable in a Long-Indexed reference. This combination of register selection and addressing mode allows any location in memory to be addressed directly. Since this mode uses indexed addressing, accesses are slower than register-direct accesses.

Examples of Zero Register Addressing:

```
ADD AX,1234[ZER0_REG]    ; AX <-- AX + MEM_WORD(1234)  (ADD_2op)
POP 5678[ZERO_REG]       ; MEM_WORD(5678) <-- MEM_WORD(SP)
                         ; SP <-- SP + 2
```

Definition of Temporary Registers:

AX is a 16-bit register.

3.2.8. Stack Pointer Register Addressing

Bytes 18H and 19H of the lower Register File contain the system Stack Pointer (SP), which is addressed at 18H. Besides providing for convenient manipulation of the Stack Pointer, SP can also be used as the WORD variable in an Indirect reference to access the top of the stack or in a Short-Indexed reference to access data within the stack.

Examples of Stack Pointer Register Addressing:

```
PUSH [SP]    ; Duplicate TOP_OF_STACK
LD AX,2[SP] ; AX <-- NEXT_TO_TOP
```

Definition of Temporary Registers:

AX is a 16-bit register.

3.3. ASSEMBLY LANGUAGE ADDRESSING MODE SELECTIONS

The MCS-96 assembly language simplifies the choice of addressing modes. These features simplify the programming task and should be used wherever possible.

3.3.1. Direct Addressing

The assembly language chooses between Register-Direct and Zero Register addressing depending on the memory location of the operand. The programmer can simply refer to the operand by its symbolic name. If the operand is in the lower Register File, the assembly language chooses a Register-Direct reference. If the operand is elsewhere in memory, it chooses a Long-Indexed reference.

3.3.2. Indexed Addressing

The assembly language chooses between Short-Indexed and Long-Indexed addressing depending on the value of the index expression. If the value can be expressed in eight bits, the assembly language chooses a Short-Indexed reference. If the value is greater than eight bits, it chooses a Long-Indexed reference.