Introduction to the 8XC196KC/KD

2

CHAPTER 2
INTRODUCTION TO THE 8XC196KC/KD

The 8XC196KC and 8XC196KD are 16-bit CHMOS microcontrollers designed to handle high-speed calculations and fast input/output (I/O) operations. They share a common architecture and instruction set with other members of the MCS-96 family. This chapter provides a high-level overview of both the architecture and software.

Typical applications using the MCS-96 products include closed-loop control and mid-range digital signal processing. Modems, motor-control systems, printers, engine-control systems, photocopiers, anti-lock brakes, air conditioner control systems, disk drives, and medical instrumentation all use MCS-96 products.

Figure 2-1 is a block diagram of the 8XC196KC and 8XC196KD. Each device has a 16-bit-wide Central Processing Unit (CPU) that connects to both an interrupt controller and a memory controller via a 16-bit CPU bus. An extension of this bus connects the CPU to the internal peripheral modules. In addition, an 8-bit CPU bus transfers instruction bytes from the memory controller to the instruction register in the Register Arithmetic-Logic Unit (RALU).

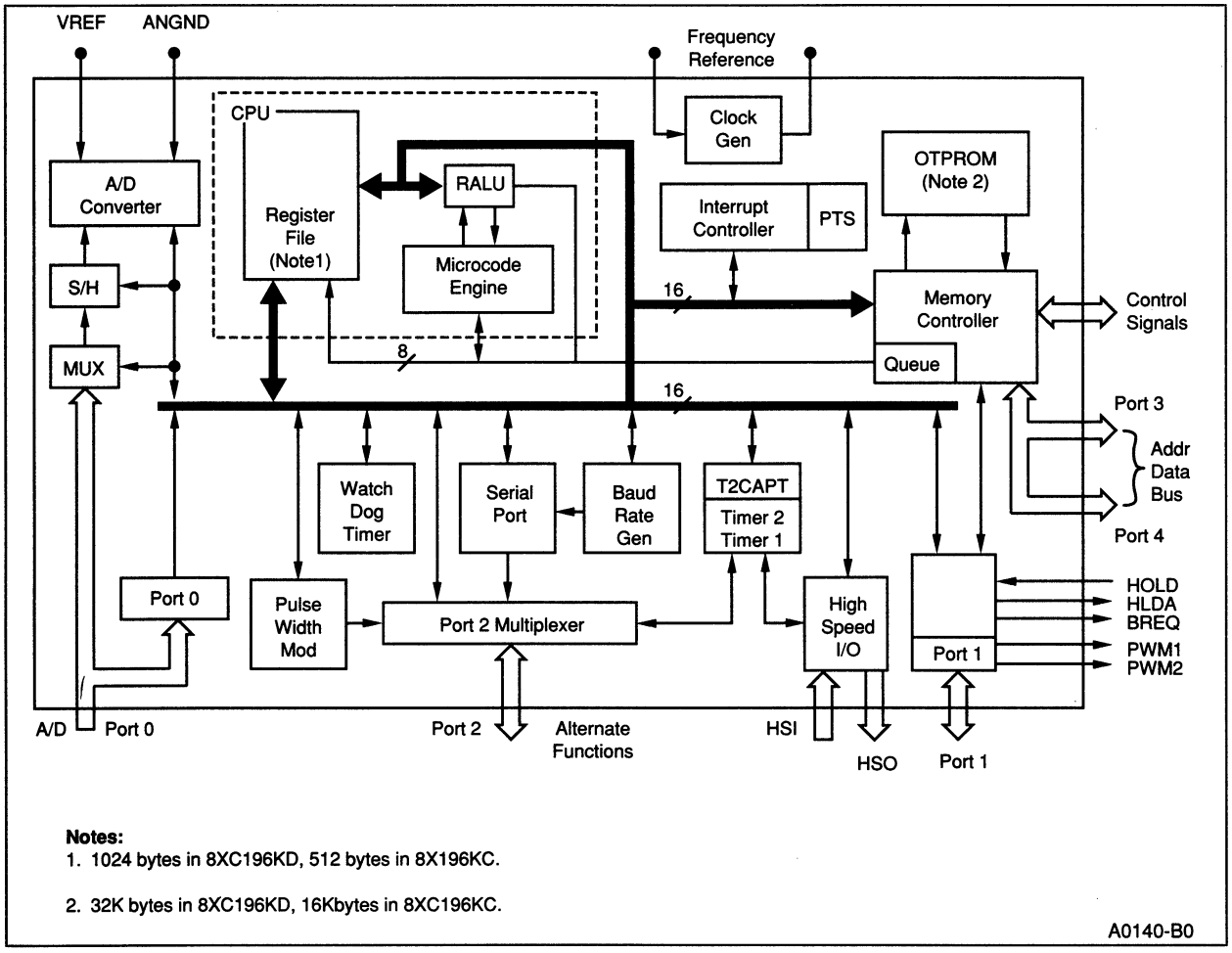

Figure 2-1. 8XC196KC/KD Block Diagram

2.1. COMPARISON OF THE 8XC196KC AND 8XC196KD

The 8XC196KD is a high-speed version of the 8XC196KC with twice as much internal One-Time-Programmable ROM (OTPROM) and RAM (see Table 2-1). The 8XC196KC and 8XC196KD have the same number and types of peripheral modules.

Table 2-1. 8XC196KC and 8XC196KD Comparisons

Feature	8XC196KC	8XC196KD
Addressable Memory Space	64 Kbytes	64 Kbytes
Internal RAM (including SFRs)	512 bytes	1024 bytes
One-Time-Programmable ROM	16 Kbytes	32 Kbytes
Maximum Operating Frequency	16 MHz	20 MHz

2.2. 8XC196KC/KD CORE

The core of the 8XC196KC/KD consists of Central Processing Unit (CPU), a memory controller, and an interrupt controller (see Figure 2-2). The CPU contains a Register/Arithmetic Logic Unit (RALU) and a Register File.

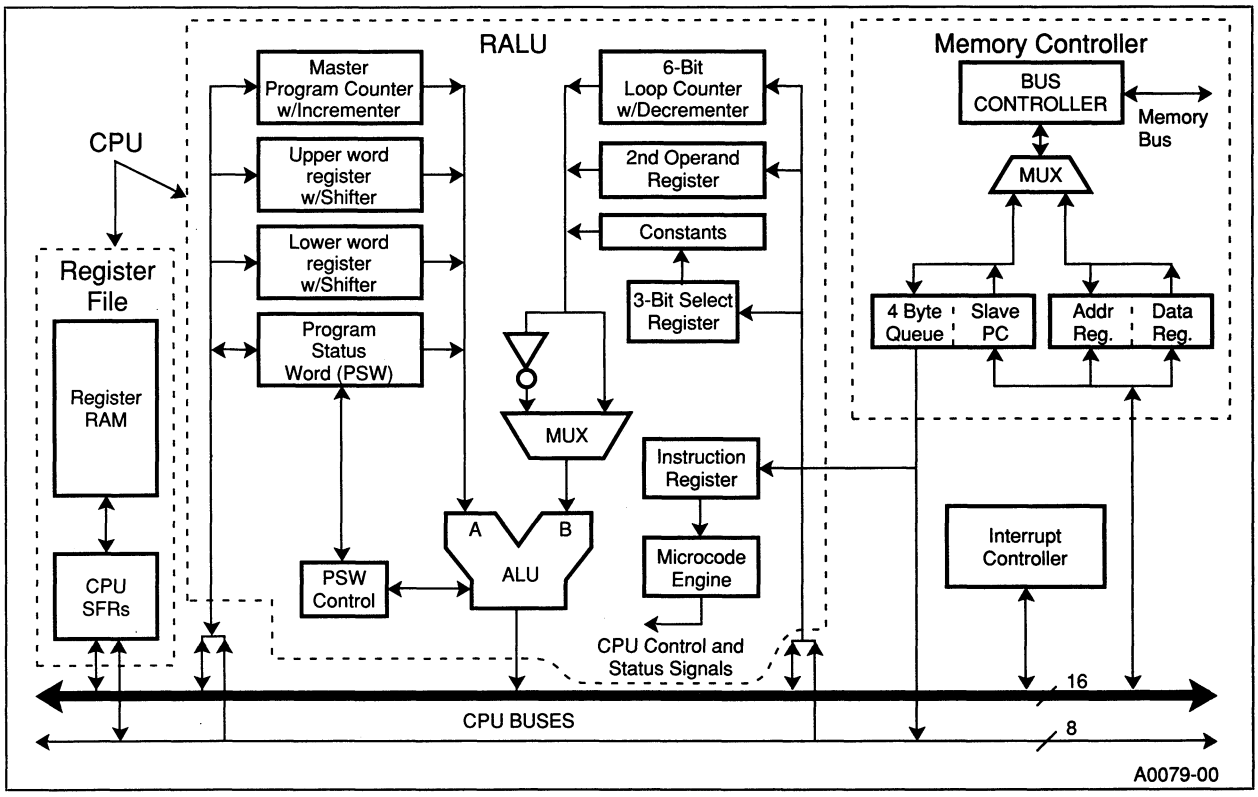

Figure 2-2. Block Diagram of the 8XC196KC/KD Core

2.2.1. CPU Control

The CPU is controlled by the microcode engine, which instructs the RALU to perform operations using bytes, words, or double words from either the 256-byte lower Register File or through a window that directly accesses the upper Register File. CPU instructions move from the four-byte queue in the memory controller into the RALU's instruction register. The microcode engine decodes the instructions and then generates the sequence of events that cause desired functions to occur.

2.2.2. Register File

The Register File is divided into an upper and lower file. The lower Register File contains 24 bytes of Special Function Register (SFR) space and 232 bytes of general-purpose register RAM. The upper Register File contains only general-purpose register RAM (256 bytes in the 8XC196KC and 768 bytes in the 8XC196KD). The general-purpose register RAM can be accessed as bytes, words, or double-words.

The RALU accesses the upper and lower Register Files differently. The lower Register File is always directly accessible via the Register-Direct address mode (see "Addressing Modes" in Chapter 3). The upper Register File is accessible via the Register-Direct address mode only when *vertical windowing* is enabled. Vertical windowing is a technique that maps blocks of the upper Register File into a *window* in the lower Register File. See Chapter 4, "Memory Partitions," for more information about the Register File and windowing.

2.2.3. Register Arithmetic-Logic Unit (RALU)

The RALU contains a 17-bit Arithmetic Logic Unit (ALU), the Program Status Word (PSW), the master program counter (PC), the instruction register, the microcode engine, a constants register, a 3-bit select register, a loop counter, and three temporary registers (the upper-word, lower-word, and second operand registers). All registers, except the three-bit select register, are either 16 or 17 bits (16 bits plus a sign extension) wide. Some of these registers can reduce the ALU's workload by performing simple operations. Words enter the ALU through the A and B inputs.

The RALU speeds up calculations by storing constants (i.e., 0, 1, and 2) in the constants register so that they are readily available when complementing, incrementing, or decrementing bytes or words.

The PSW contains one bit (PSW.1) that globally enables or disables servicing of all maskable interrupts, one bit (PSW.2) that enables or disables the Peripheral Transaction Server (PTS), and six Boolean flags that reflect the state of the user's program. Appendix C provides a detailed description of the PSW.

The PC contains the address of the next instruction and has a built-in incrementer that automatically loads the next sequential address. If a jump, interrupt, call, or return changes the address sequence, the ALU loads the appropriate address into the PC.

The upper and lower word registers are used together for 32-bit instructions and as temporary registers for many instructions. Since they have their own shift logic, the RALU also uses them for operations that require logical shifts, (e.g., normalize, multiply, and divide). The lower-word register is used only when double-word quantities are being shifted, the upper-word register is used whenever a shift is performed. Repetitive shifts are counted by the 6-bit loop counter.

The second operand register stores the second operand when the microcode engine executes a two-operand instruction. This includes the multiplier during multiply instructions and the divisor during divide instructions. During subtractions, the output of this register is complemented before it is moved into the B input of the ALU.

2.2.3.1. CODE EXECUTION

The RALU performs most calculations for the 8XC196KC/KD, but it does not use an accumulator. Instead it operates directly on the lower Register File, which essentially provides 256 *accumulators*. Because data does not flow through a single accumulator, the 8XC196KC/KD's code executes faster and more efficiently.

For example, the following 80C186 code multiplies two 16-bit variables (FACTOR_1 and FACTOR_2) and stores the 32-bit result in a third variable (RESULT).

```
MOV AX, FACTOR_1                    ;move factor_1 into a register
MUL AX, FACTOR_2                    ;multiply factor_2 and contents
                                    ;of AX register and store in AX
MOV RESULT, AX                      ;move lower byte into "result"
MOV RESULT+2,DX                     ;move upper byte into "result+2"
```

The following example shows the equivalent code for the 8XC196KC/KD.

```
MUL RESULT, FACTOR_1, FACTOR_2      ;multiply factor_1 & factor_2
                                    ;store answer in result
```

The 8XC196KC/KD can perform this operation in one instruction because it combines a large set of general-purpose registers with a 3-operand instruction format. This format allows a single instruction to specify two source registers and a separate destination register.

2.2.4. Memory Controller

The RALU communicates with all memory, except the Register File, through the memory controller. (It communicates with the upper Register File through the memory controller except when *vertical windowing* is used; see Chapter 4.) The memory controller contains address and data registers, a four-byte queue, a slave program counter (slave PC), and a bus controller.

The bus controller drives the memory bus, which consists of the internal OTPROM bus, the internal RAM bus, and the external address/data bus. The bus controller receives memory-access requests from either the RALU or the four-byte pre-fetch queue; queue requests have priority. This queue is transparent to the RALU and the user.

When the bus controller receives a request from the queue, it fetches the code from the address contained in the slave PC. This increases execution speed since the next instruction byte is available immediately and the processor need not wait for the master PC to send the address to the memory controller. If a jump, interrupt, call, or return changes the address sequence, the master PC loads the new address into the slave PC, the queue is flushed, and processing continues.

NOTE

When using a logic analyzer to debug code, remember that instructions are preloaded into the four-byte queue and are not necessarily executed immediately after they are fetched.

2.2.5. Interrupt Controller

The programmable Interrupt Controller has a hardware priority scheme that can be modified by user software. These interrupts are serviced by user-written interrupt service routines. In addition, the 8XC196KC/KD provides a microcoded hardware interrupt processor, the Peripheral Transaction Server (PTS). The PTS responds to interrupts with a fixed set of actions, such as transferring data, starting an A/D conversion, reading the High-Speed Input module's FIFO, and loading events into the High-Speed Output module. The PTS completes these tasks much more quickly than standard interrupt-driven software service routines can. The PTS can service all interrupts except NMI, Trap, and Unimplemented Opcode. PTS cycles have a higher priority than standard interrupts and may temporarily suspend interrupt service routines. See Chapter 5, "Interrupts" for more information.

2.3. INTERNAL TIMING

The clock generator halves the frequency of the signal on XTAL1 and produces the two internal timing signals, PH1 and PH2. These signals are active when high. The rising edges of PH1 and PH2 generate CLKOUT, the output of the internal clock generator (see Figure 2-3).

The combined period of PH1 and PH2 defines the basic time unit known as a *state time* or *state*. At the maximum 8XC196KD frequency of 20 MHz, one state time equals 100 ns. At the maximum 8XC196KC frequency of 16 MHz, one state time equals 125 ns. Because the 8XC196KC/KD can operate at many frequencies, this manual defines time requirements in terms of state times rather than specific times. Consult the latest data sheet for AC timing specifications.

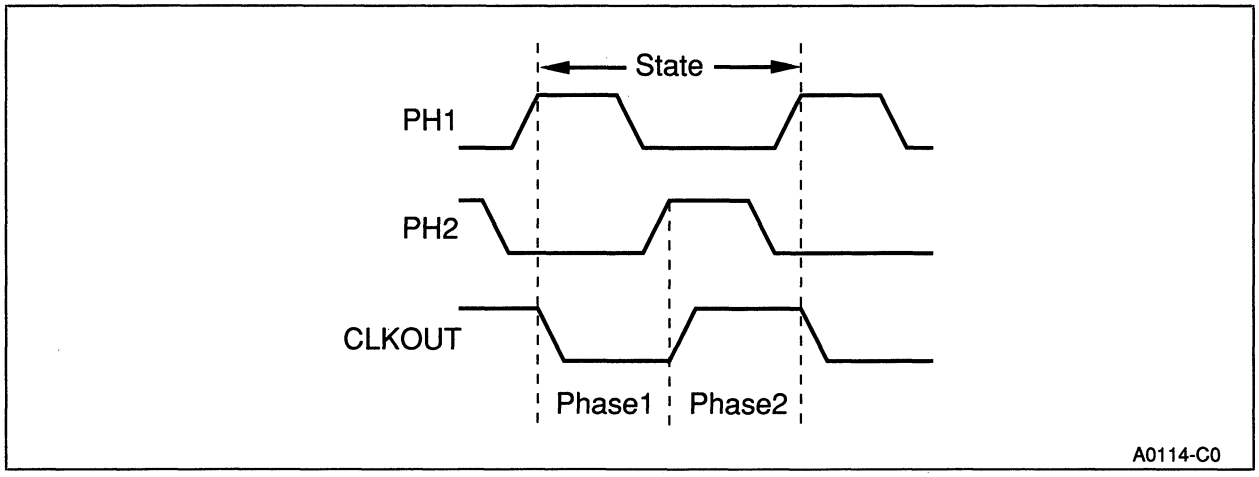

Figure 2-3. Internal Clock Phases

NOTE

A CLKOUT disable bit was added to the IOC3 register in the 8XC196KD and the 8XC196KC (C-Step). Setting this bit disables the CLKOUT signal, which can reduce system noise. This feature is not available in earlier versions of the 8XC196KC. See Appendix C for a description of the IOC3 register.

2.4. INTERNAL PERIPHERALS

The 8XC196KC/KD's internal peripheral modules provide special functions for a variety of applications.

2.4.1. Standard I/O Ports

The 8XC196KC/KD has five 8-bit I/O ports. Some are input-only, some are output-only, some are bidirectional, and some support multiple functions. Port 0 is an input port that is also the analog input for the A/D converter. Port 1 is a quasi-bidirectional port. Port 1 pins are multiplexed with bus control signals and two outputs from the Pulse Width Modulator.

Port 2 contains three types of port lines: quasi-bidirectional, input, and output. Other functions on the 8XC196KC/KD share the Port 2 input and output lines. Ports 3 and 4 are open-drain bidirectional ports that share their pins with the address/data bus. See Chapter 6, "I/O Ports," for more information.

2.4.2. Serial I/O Port

The serial I/O port is an asynchronous/synchronous port that includes a Universal Asynchronous Receiver and Transmitter (UART). The UART has one synchronous mode (Mode 0) and three asynchronous modes (Modes 1, 2 and 3). The asynchronous modes are full duplex, meaning that they can transmit and receive data simultaneously. The receiver on the 8XC196KC/KD is double buffered, so the reception of a second byte may begin before the first byte is read. The transmitter is also double buffered and can generate continual transmissions. See Chapter 7, "Serial I/O Port," for more information.

2.4.3. The High-Speed Input/Output (HSIO) Unit

The HSIO unit contains four individual peripheral modules: Timer 1, Timer 2, High-Speed Input, High-Speed Output. Together, these modules form a flexible timer/counter-based I/O system. See Chapter 8, "High-Speed Input/Output Unit," for more information.

2.4.3.1. TIMER 1 AND TIMER 2

Timer 1 is a free-running timer that is incremented every eight state times. It is the time base for the High-Speed Input module and optionally for the High-Speed Output module.

Timer 2 counts both positive and negative input transitions. It can be used as the time base for the High-Speed Output module, as an up/down counter, or as an extra timer.

2.4.3.2. HIGH-SPEED INPUT (HSI)

The HSI module can record times of external events with an eight-state-time resolution. It can monitor four independently configurable inputs and capture the value of Timer 1 when an event takes place. The four types of events that can trigger captures include rising edges, falling edges, rising or falling edges, or every eighth rising edge. The HSI module can store up to eight entries (Timer 1 values): seven in the seven-level FIFO and one in the HSI holding register.

2.4.3.3. HIGH-SPEED OUTPUT (HSO)

The HSO module can trigger events at specified times based on Timer 1 or Timer 2. These programmable events include starting an A/D conversion, resetting Timer 2, generating up to four software timers, and setting or clearing one or more of the six HSO output lines. The HSO unit stores pending events and the specified times in a Content-Addressable Memory (CAM) file. This file stores up to eight commands. Each command specifies the action time,

the nature of the action, whether an interrupt is to occur, and whether Timer 1 or Timer 2 is the reference timer.

2.4.4. Analog-to-Digital Converter

The analog-to-digital (A/D) converter converts an analog input voltage to a digital equivalent. Resolution is either 8 or 10 bits; sample and convert times are programmable. Automated A/D conversions and result storage are facilitated by the A/D Scan Mode of the PTS. The main components of the A/D Converter are a sample and hold, an 8-channel multiplexer, and an 8-bit or 10-bit *successive approximation* analog-to-digital converter. See Chapter 9, "Analog-to-Digital Converter," for more information.

2.4.5. Pulse Width Modulator (PWM)

The 8XC196KC/KD has three PWM modules. The output waveform from each is a variable duty cycle pulse that occurs every 256 or 512 state times as programmed. Several types of motors require a PWM waveform for most efficient operation. When filtered, the PWM waveform will produce a DC level that can change in 256 steps by varying the duty cycle. See Chapter 10, "Pulse Width Modulator," for more information.

2.4.6. Watchdog Timer

The Watchdog Timer is an internal timer that resets the device if the software fails to operate properly. See Chapter 11, "Minimum Hardware Considerations," for more information.

2.5. SPECIAL OPERATING MODES

In addition to the normal execution mode, the 8XC196KC/KD operates in several special-purpose modes. Idle mode and Powerdown mode conserve power when the device is inactive, ONCE mode electrically isolates the 8XC196KC/KD from the system, and several other modes provide programming options for nonvolatile memory. See Chapter 12, "Special Operating Modes," for more information about Idle, Powerdown, and ONCE modes and Chapter 13, "Programming the Nonvolatile Memory," for details about programming options.

2.5.1. Reducing Power Consumption

In Idle mode, the CPU stops executing instructions, but the peripheral clocks remain active. Power consumption drops to about 40% of normal execution mode consumption. Either a hardware reset or any enabled interrupt source will bring the device out of Idle mode.

In Powerdown mode, all internal clocks are frozen at logic state zero and the oscillator is shut off. Internal RAM and most peripherals retain their data if V_{CC} is maintained. Power consumption drops into the µW range.

2.5.2. Testing the Printed Circuit Board

ONCE mode electrically isolates the 8XC196KC/KD from the system. By invoking ONCE mode, you can test the printed circuit board while the 8XC196KC/KD is soldered onto the board.

2.5.3. Programming the 8XC196KC/KD

The 8XC196KC/KD supports Auto Programming Mode, Slave Programming Mode, and Run-Time Programming.

- Auto Programming Mode enables the 8XC196KC/KD to program itself from an external EPROM, without an EPROM programmer.

- Slave Programming Mode supports programming with an EPROM programmer. While using this programming mode, you can program and verify any single word in the OTPROM.

- Run-Time Programming allows you to program individual OTPROM locations during normal code execution, while under complete software control.

2.6. INTRODUCTION TO THE 8XC196KC/KD SOFTWARE

This section provides an overview of the MCS-96 instruction set, 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 and SFRs. Chapter 3 describes data types and addressing modes.

2.6.1. Overview of the MCS®-96 Instruction Set

The MCS-96 instruction set contains a full set of arithmetic and logical operations for the 8- and 16-bit data types (BYTE and SHORT-INTEGER, WORD and INTEGER). It supports the 32-bit data types (DOUBLE-WORD and LONG-INTEGER) only as operands in shift operations, as the dividends of 32-by-16 divide operations, and as products of 16-by-16 multiply operations. The remaining operations on 32-bit variables can be implemented by combinations of 16-bit operations.

For example, the following sequences of 16-bit operations perform a 32-bit addition and a 32-bit subtraction, respectively.

```
ADD AX,CX ; (ADD_2op)
ADDC BX,DX

SUB AX,CX ; (SUB_2op)
SUBC BX,DX
```

The instruction set also supports conversions between the data types. The LDBZE (load byte, zero extended) instruction converts a BYTE to a WORD. CLR (clear) can convert a WORD to a DOUBLE-WORD by clearing (writing zeros to) the upper WORD of the DOUBLE-WORD. LDBSE (load byte, sign extended) converts a SHORT-INTEGER into an INTEGER. EXT (sign extend) converts an INTEGER to a LONG-INTEGER.

The MCS-96 instructions for addition, subtraction, and comparison do not distinguish between unsigned WORDs and signed INTEGERs. However, the conditional jump instructions allow you to treat the results of these operations as signed or unsigned quantities. For example, the CMPB (compare byte) instruction is used to compare both signed and unsigned eight-bit quantities. Following a compare operation, you can use the JH (jump if higher) instruction for unsigned operands or the JGT (jump if greater than) instruction for signed operands.

The hardware does not directly support operations on REAL (floating point) variables. Those operations are supported by the floating point library for the 8XC196KC/KD (FPAL-96), which implements a single-precision subset of the proposed IEEE standard for floating point operations. The performance of this software is significantly improved by the NORML instruction and by the Sticky Bit (ST) flag in the PSW. The NORML instruction normalizes a 32-bit variable; the Sticky Bit (ST) flag can be used in conjunction with the Carry (C) flag to achieve finer resolution in rounding.

2.6.2. Additions to the MCS®-96 Instruction Set

For users already familiar with the 8096BH, this section briefly describes the instructions that have been added to the standard MCS-96 instruction set to form the 8XC196KC/KD instruction set. Please refer to Appendix A for detailed descriptions.

BMOV	BLOCK MOVE. Moves a block of word data from one location to another in memory. This instruction cannot be interrupted.
BMOVI	INTERRUPTABLE BLOCK MOVE. Moves a block of word data from one location to another in memory. This instruction is identical to BMOV, except that BMOVI can be interrupted.
CMPL	COMPARE LONG. Compares the magnitudes of two double-word operands.

DJNZW	DECREMENT AND JUMP IF NOT ZERO WORD. Decrements the value of the word operand and jumps if the result is other than zero.
DPTS	DISABLE PTS. Clears PSW.2, which disables the Peripheral Transaction Server (PTS).
EPTS	ENABLE PTS. Sets PSW.2, which enables the Peripheral Transaction Server (PTS).
IDLPD	IDLE/POWERDOWN. Causes the device either to enter Idle mode, to enter Powerdown mode, or to execute a reset sequence, depending on the value of the operand.
POPA	POPA. Used instead of POPF, to support the eight added interrupts. It pops two words off the stack, placing the first into the INT_MASK1/WSR register-pair and the second into the PSW/INT_MASK word.
PUSHA	PUSH ALL. Used instead of PUSHF, to support the eight added interrupts. It pushes two words onto the stack: the PSW/INT_MASK word and the word formed by the INT_MASK1/WSR register-pair. It clears the PSW, INT_MASK, and INT_MASK1 registers.
TIJMP	TABLE INDIRECT JUMP. Selects an address from a table of addresses, calculates the destination address, and jumps to that address. The TIJMP instruction can reduce the time required to access a look-up table.
XCH	EXCHANGE WORD. Exchanges the value of the source word operand with that of the destination word operand.
XCHB	EXCHANGE BYTE. Exchanges the value of the source byte operand with that of the destination byte operand.

2.6.3. Instruction Set Differences

For many instructions, execution times are shorter on the 8XC196KC/KD than on the 8096BH. The multiply instructions are nearly twice as fast. For example, a 16-by-16 unsigned multiply operation that took 25 state times on the 8096BH takes only 14 state times on the 8XC196KC/KD. Many zero- and one-operand instructions and most instructions that use external data take one or two fewer state times on the 8XC196KC/KD than on the 8096BH.

Indexed and indirect operations relative to the Stack Pointer (SP) work differently on the 8XC196KC/KD than on the 8096BH. On the 8096BH, the address is calculated based on the value of the SP before it is updated; on the 8XC196KC/KD the updated SP is used. The offset for PUSH [SP], POP [SP], PUSH *nn*[SP], and POP *nn*[SP] instructions may need to be changed by a count of two.

2.6.4. Software Standards and Conventions

For a software project of any size, it is a good idea to modularize the program and to establish standards that control communication between the modules. These standards vary with the needs of the final application. However, all standards must include some mechanism for passing parameters to procedures and returning results from procedures. We recommend that you use the conventions adopted by the PLM-96 programming language for procedure linkage. It is a very usable standard for both the assembly language and PLM-96 environment, and it offers compatibility between these environments. It also allows the programmer access to the floating-point arithmetic library (FPAL-96) that PLM-96 uses to operate on REAL variables.

2.6.4.1. USING REGISTERS

The MCS-96 architecture provides a 256-byte lower Register File. Some of these registers are used for register-mapped I/O devices and special functions such as the Zero Register and the Stack Pointer. The remaining bytes in the lower Register File, some 232 of them, are available for your use.

To use these registers effectively, you must have some overall strategy for allocating them. PLM-96 adopts a simple and effective strategy. PLM-96 allocates the eight bytes between addresses 1CH and 23H as temporary storage (calling the starting address of this region PLMREG), and treats the remaining area in the Register File as a segment of memory that is allocated as required.

Special Function Registers (SFRs) can be operated on as BYTEs or WORDs, unless otherwise specified. Use caution when using an SFR as the source of an operand or as the base or index register for indirect or indexed operations. Unexpected results can occur because external events can change SFRs and reading some SFRs clears them. Consider the potential for an SFR to change value, especially when using high-level languages, which do not always allow for SFR-type registers.

2.6.4.2. ADDRESSING 32-BIT OPERANDS

The 32-bit operands (DOUBLE-WORDs and LONG-INTEGERs) are formed by two adjacent 16-bit words in memory. The least significant word of a DOUBLE-WORD is always in the lower address, even when the data is in the stack (which means that the most significant word must be pushed into the stack first). The address of a 32-bit operand is that of its least significant byte.

The hardware supports the 32-bit data types as operands in shift operations, as the dividends of 32-by-16 divide operations, and as products of 16-by-16 multiply operations. For these operations, the 32-bit operand must reside in the internal Register File and must be aligned at an address that is evenly divisible by four.

2.6.4.3. LINKING SUBROUTINES

Parameters are passed to subroutines via the stack. Parameters are pushed into the stack in the order in which they are encountered in the scanning of the source text. The 8-bit parameters (BYTE and SHORT-INTEGER) are pushed into the stack with the high order byte undefined. The 32-bit parameters (LONG-INTEGER, DOUBLE-WORD, and REAL) are pushed onto the stack as two 16-bit values; the most significant half of the parameter is pushed into the stack first.

As an example, consider the following PLM-96 procedure:

```
example_procedure:PROCEDURE (param1,param2,param3);
DECLARE param1 BYTE,
param2 DWORD,
param3 WORD
```

When this procedure is entered at run-time, the stack will contain the parameters in the following order:

```
Stack Image
------------ | ?????? ; param1      |
             | high word of param2  |
             | low word of param2   |
             | param3               |
             | return address       | <-- Stack_Pointer
```

If a procedure returns a value to the calling code (as opposed to modifying more global variables) then the result is returned in the PLMREG variable. PLMREG is viewed as either an 8-, 16-, or 32-bit variable, depending on the type of the procedure.

The standard calling convention adopted by PLM-96 has several key features:

- Procedures can always assume that the eight bytes of Register File memory starting at PLMREG can be used as temporary storage within the body of the procedure.
- Code that calls a procedure must assume that the procedure modifies the eight bytes of Register File memory starting at PLMREG.
- Code that calls a procedure must assume that the procedure modifies the Program Status Word (PSW) condition flags (Z, N, V, VT, C, and ST), because procedures do not save and restore the PSW.
- Function results from procedures are always returned in the variable PLMREG.

PLM-96 allows the definition of INTERRUPT procedures, which are executed when a predefined interrupt occurs. INTERRUPT procedures do not conform to the rules of normal procedures. Parameters cannot be passed to these procedures and they cannot return results. Since INTERRUPT procedures can execute essentially at any time, they must save and restore the PSW and PLMREG.

2.6.5. Software Protection Features and Guidelines

The 8XC196KC/KD has several features to assist in recovering from hardware and software errors. The Unimplemented Opcode interrupt provides protection from executing unimplemented opcodes. The hardware reset instruction (RST) can cause a reset if the program counter goes out of bounds. The RST instruction opcode is 0FFH, so the processor will reset itself if it reads in bus lines that have been pulled high. The Watchdog Timer (WDT) can also reset the device in the event of a hardware or software error.

We recommend that you fill unused areas of code with NOPs and periodic jumps to an error routine or RST instruction. This is particularly important in the code surrounding lookup tables, since executing lookup tables will cause undesired results. Wherever space allows, each table should be surrounded by seven NOPs (because the longest 8XC196KC/KD instruction has seven bytes) and a RST or a jump to an error routine. Since RST is a one-byte instruction, the NOPs are unnecessary if RSTs are used instead of jumps to an error routine. This will help to ensure a speedy recovery should the processor have a glitch in the program flow.

When using the WDT for software protection, we recommend that you reset the WDT from only one place in code, reducing the chance of an undesired WDT reset. The section of code that resets the WDT should monitor the other code sections for proper operation. This can be done by checking variables to make sure they are within reasonable values. Simply using a software timer to reset the WDT every 10 milliseconds will provide protection only for catastrophic failures.