SIEMENS

C500
Microcontroller Family

Architecture and Instruction Set

User's Manual   04.98

C500 Architecture and Instruction User's Manual	
Revision History :	**04.98** Original Version
Previous Releases:	07.96, 01.97, 09.97

Page	**Subjects (changes since last revision)**
4-15, 4-59, 4-79	Correction: first paragraph of "Description"; P is affected if <dest-byte> = A
4-16	Encoding for ANL direct,A corrected
Page	**Subjects (changes from 01.97 to 09.97)**
4-88	Table 4-4: Mnemonics for Opcode 83_H, 86_H and 87_H corrected
Page	**Subjects (changes from 07.96 to 01.97)**
Table of content	Reference to pages of each instruction added
4-89	Table header corrected

Table of Contents **Page**

1 Fundamental Structure

1.1 Introduction

The members of the C500 Siemens microcontroller family are basically fully compatible in architecture and software to the standard 8051 microcontroller family. Especially, they are functionally upward compatible to the SAB 80C52/80C32 microcontroller. While maintaining all architectural and operational characteristics of the SAB 80C52/80C32, the C500 microcontrollers differ in number and complexity of their peripheral units which have been adapted to the specific application areas.

The goal of this "Architecture and Instruction Set Manual" is to summarize the basic architecture and functional characteristics of all members of the C500 microcontroller family. This includes the description of the architecture and the description of the complete instruction set. Detailed information about the different versions of the C500 microcontrollers are given in the specific User Manuals.

1.2   Memory Organization

The memory resources of the C500 family microcontrollers are organized in different types of memories (data and program memory), which further can be located internally on the microcontroller chip or outside of the microcontroller. The memory partitioning of the C500 microcontrollers is typical for a Harvard architecture where data and program areas are held in separate memory areas. The on-chip peripheral units are accessed using an internal special function register memory area.

The available memory areas have different sizes and are located in the following five address spaces:

Table 1-1
C500 Address Spaces

Type of Memory	Location	Size
Program Memory	External	max. 64 KByte
	Internal (ROM, EEPROM)	Depending on C500 version 2K up to 64KByte
Data Memory	External	max. 64 KByte
	Internal XRAM	Depending on C500 version 256 Byte up to 3 KByte
	Internal	128 or 256 Byte
Special Function Register	Internal	128/256 Bytes

1.2.1 Program Memory

The program memory of the C500 family microcontrollers can be composed of either completely external program memory, of only internal program memory (on-chip ROM / EEPROM), or of a mixture of internal and external program memory. If the $\overline{EA}$ pin ($\overline{EA}$=**E**xternal **A**ccess) is held at low level, the C500 microcontrollers execute the program code always out of the external program memory. Romless C500 derivatives can use this type of program memory only. C500 derivatives with on-chip program memory typically use their internal program memory only. If the internal program memory is used the $\overline{EA}$ pin must be put to high level. With $\overline{EA}$ high, the microcontroller executes instructions internally unless the address exceeds the upper limit of the internal program memory. If the program counter is set to an address (e.g. by a jump instruction) which is higher than the internal program memory, instructions are executed out of an external program memory. When the instruction address again is below the internal program memory size limit, internal program memory is accessed again.

Figure 1-1 shows the typical C500 family microcontroller program memory configuration for the two cases $\overline{EA}$=0 and $\overline{EA}$=1. The ROM boundary shown in **figure 1-1**, applies to the C501 which has 8K byte of internal ROM. Other C500 family microcontrollers with different ROM size have different ROM boundaries.

The location of the ROM boundary depends on the specific C500 devices.

MCD02766
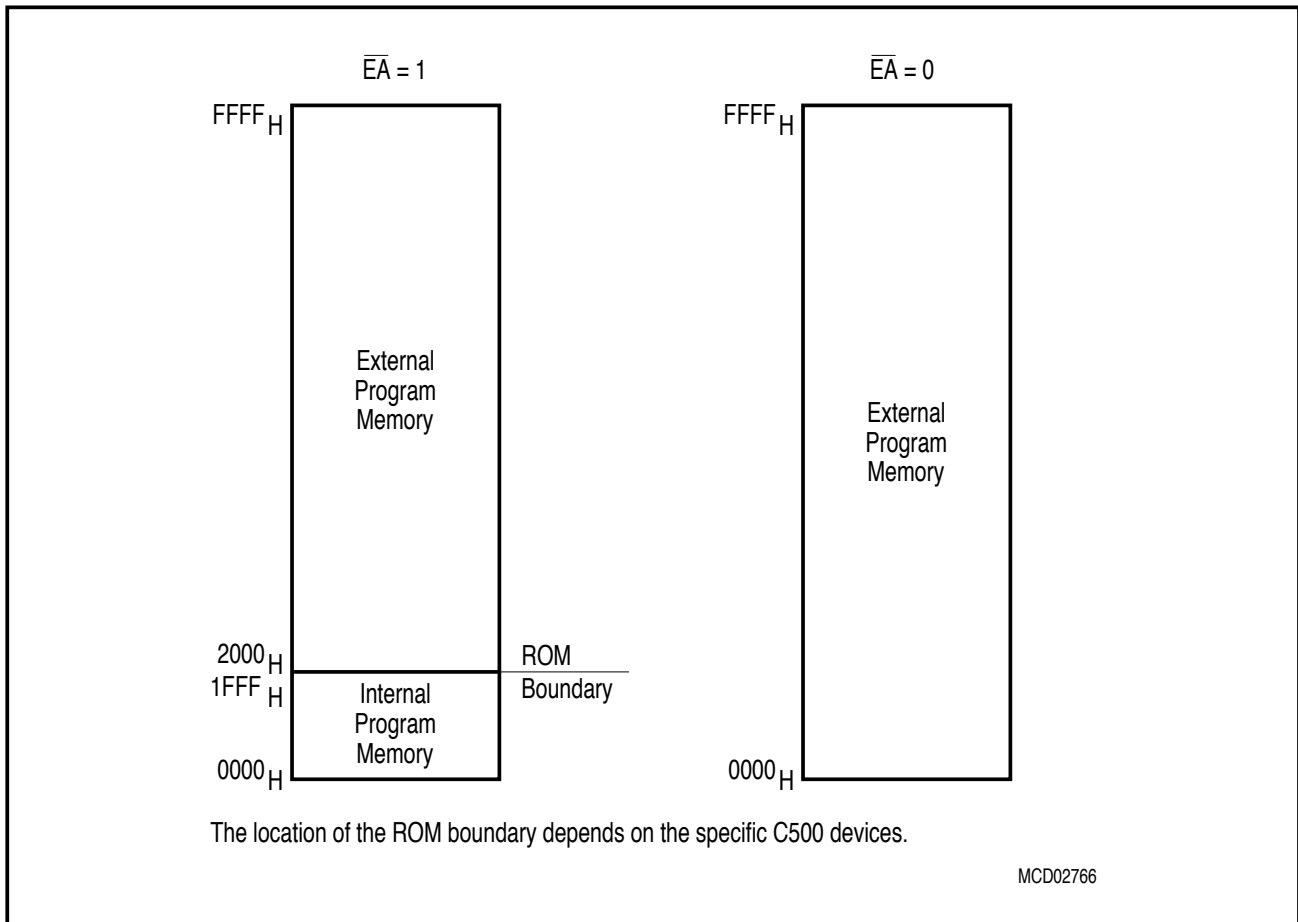
Figure 1-1
Program Memory Configuration (Example of the C501)

1.2.2 Data Memory

The data memory area of the C500 family microcontrollers consists of internal and external data memory portions. The internal data memory area is addressed using 8-bit addresses. The external data memory and the internal XRAM data memory are addressed by 8-bit or16-bit addresses.

The content of the internal data memory (also XRAM) is not affected by a reset operation. After power-up the content is undefined, while it remains unchanged during and after a reset as long as the power supply is not turned off. The XRAM content is also maintained when the C500 microcontrollers are in power saving modes.

1.2.2.1 Internal Data Memory

The internal data memory address space is divided into three basic, physically separate and distinct blocks: the lower 128 byte of internal data RAM, the upper 128 byte of internal data RAM, and the 128 byte special function register (SFR) area. The lower internal data RAM and the SFR area further include 128 bit locations each. These bits can be handled by specific bit manipulation instructions.

Figure 1-2 shows the configuration of the three basic internal RAM areas. The lower data RAM is located in the address range 00_H - $7F_H$ and can be addressed directly (e.g. MOV A,direct) or indirectly (e.g. MOV A,@R0 with address in R0). A bit-addressable area of 128 free programmable, direct addressable bits is located at byte addresses 20_H - $2F_H$ of the lower data RAM. Bit 0 of the internal data byte at 20_H has the bit address 00_H while bit 7 of the internal data byte at $2F_H$ has the bit address $7F_H$. The lower 32 locations of the internal lower data RAM are assigned to four banks with eight general purpose registers (GPRs) each. Only one of these banks can be enabled at a time to be used as general purpose registers.
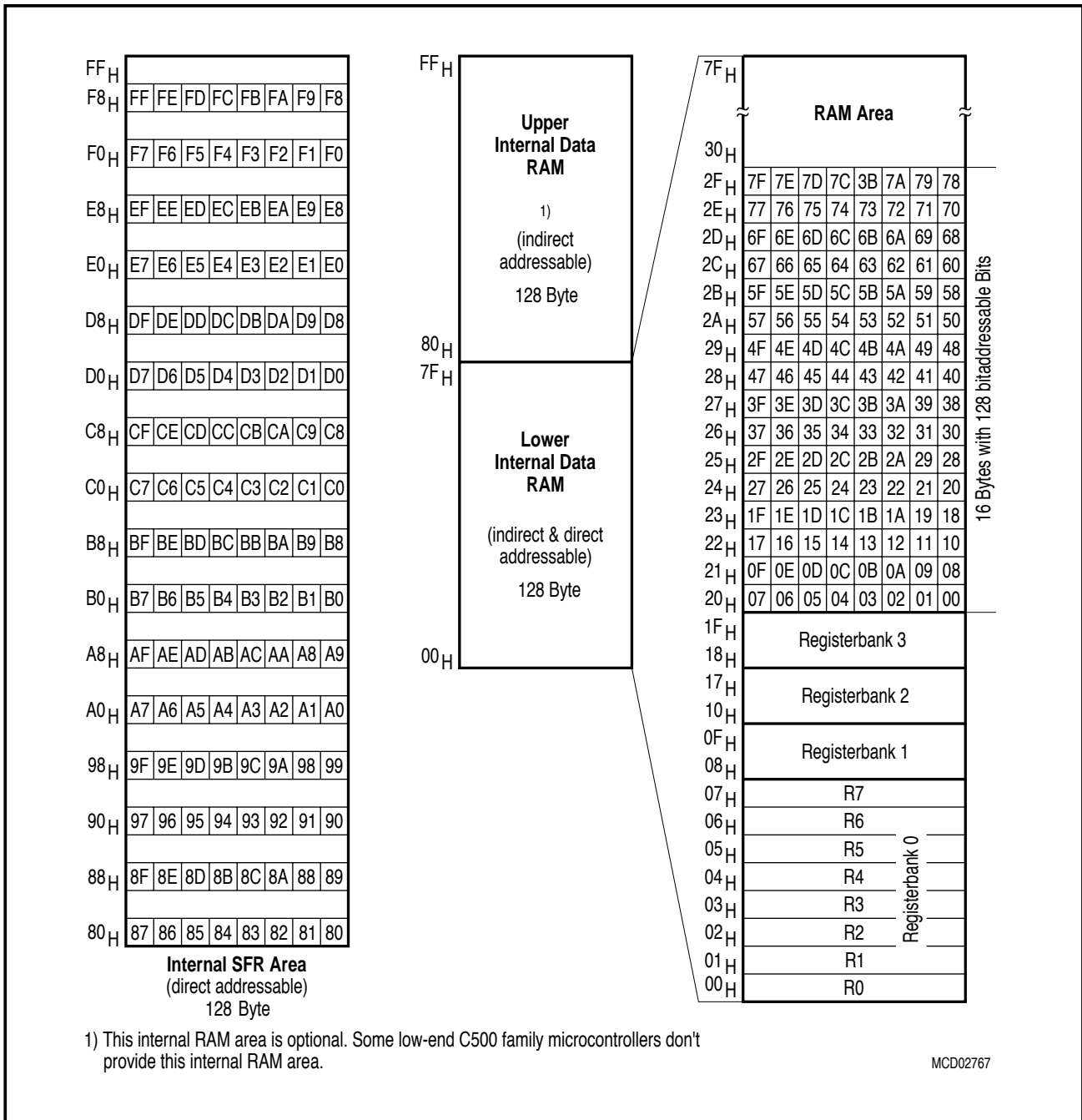
Figure 1-2
Internal Data Memory Organization

While the SFR area and the upper internal RAM area share the same address locations (80_H - FF_H), they must be accessed through different addressing modes. The upper internal RAM can only be accessed through indirect addressing while the special function registers (SFRs) are accessible only by direct addressing instructions. The SFRs which are located at addresses with address bit 0-2 equal 0 (addresses 80_H, 88_H, 90_H, ...$F0_H$, FF_H) are bitaddressable SFRs.

1.2.2.2 Internal Data Memory XRAM

Some members of the C500 family microcontrollers provide an additional internal data memory area, called the XRAM. This data memory area is logically located at the upper end of the external data memory space (except C502), but it is integrated on the chip. Because the XRAM is used in the same way as external data memory the same instruction types must be used for accessing the XRAM.

Figure 1-3 shows a typical 256 byte XRAM address mapping of the C500 microcontrollers.
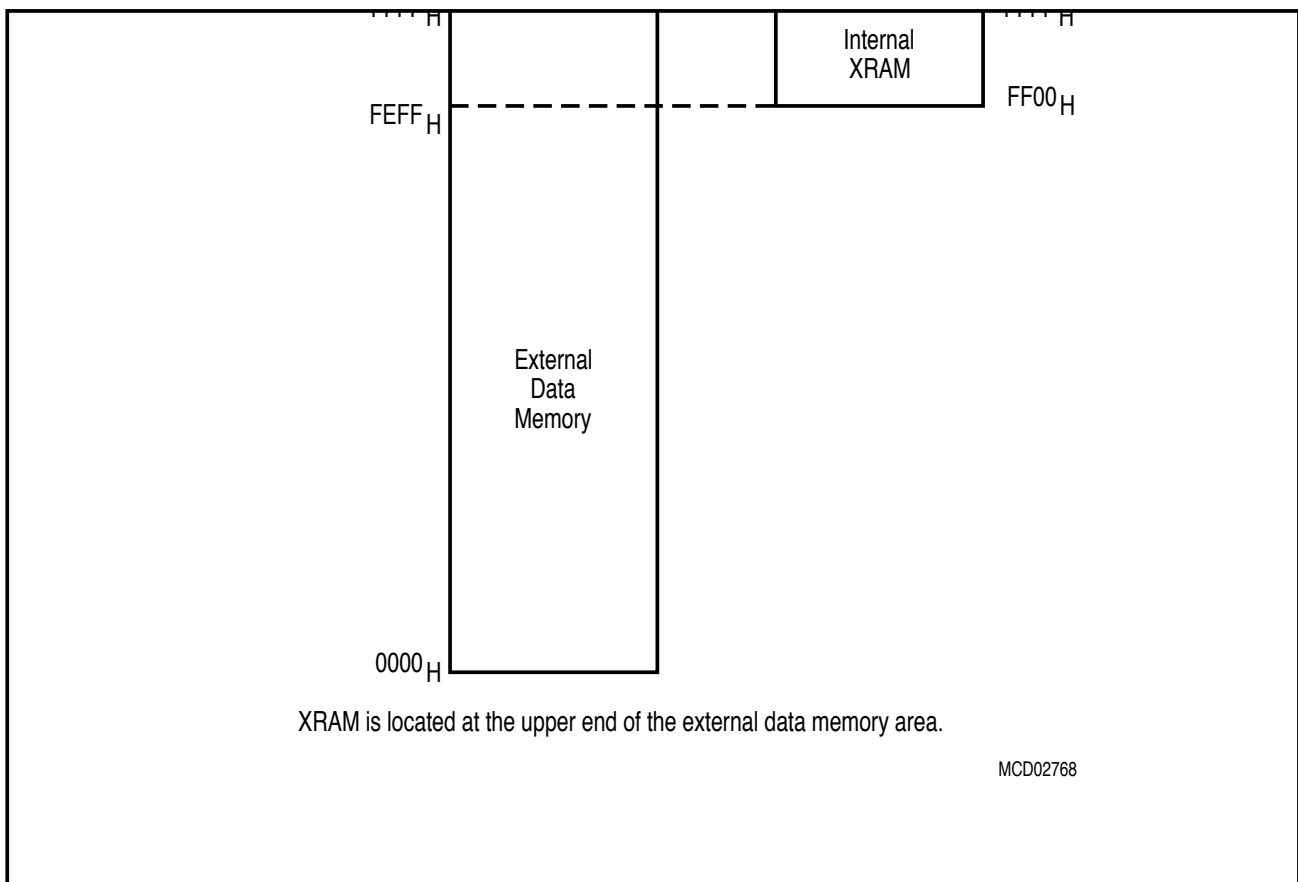
Figure 1-3
XRAM Memory Mapping (256 Byte)

Depending on the C500 derivative, the size of the XRAM area differs from 128 upto 3K byte. Further, the XRAM can be enabled or disabled. If an internal XRAM area is disabled, external data memory can be accessed in the address range of the internal XRAM.

1.2.2.3 External Data Memory

The 64 Kbyte external data memory can be addressed by instructions that use 8-bit or 16-bit indirect addressing. A 16-bit external memory addressing mode is supported by the MOVX instructions using the 16-bit datapointer DPTR for addressing. For 8-bit addressing MOVX instructions with the general purpose registers R0/R1 are used.

1.2.3 Special Function Register Area

The registers of a C500 microcontroller, except the program counter and the four general purpose register banks, reside in the special function register (SFR) area. The special function register area typically provides 128 bytes of direct addressable SFRs. The SFRs which are located at addresses with address bit 0-2 equal 0 (addresses 80_H, 88_H, 90_H, ...$F0_H$, FF_H) are bitaddressable SFRs (see also figure 1-1). For example, the SFR with byte address 80_H provides the bit locations with bit addresses 80_H to 87_H. The bit addresses of the SFR bits reach from 80_H to FF_H.

Due to the limited number of 128 standard SFRs, some derivatives of the C500 microcontroller family provide an additional 128 byte SFR area, called the mapped SFR area. The mapped SFR area provides the same addressing capabilities (direct addresses, bit addressing) as the standard SFR area.

Special Function Register SYSCON (Address $B1_H$)

Bit No.	MSB							LSB	
	7	6	5	4	3	2	1	0	
$B1_H$	–	–	–	RMAP	–	–	–	–	SYSCON

The functions of the shaded bits are not described in this section.

Bit	Function
RMAP	Special function register map bit RMAP = 0 : The access to the non-mapped (standard) special function register area is enabled (default after reset). RMAP = 1 : The access to the mapped special function register area is enabled.

As long as bit RMAP is set, mapped special function registers can be accessed. This bit is not cleared by hardware automatically. Thus, when non-mapped/mapped registers are to be accessed, the bit RMAP must be cleared/set by software, respectively each. Some registers (e.g. ACC) are accessed independently of bit RMAP.

Two bits in the program status word, RS0 (PSW.3) and RS1 (PSW.4), select the active register bank. This allows fast context switching, which is useful when entering subroutines or interrupt service routines. The 8 general purpose registers of the selected register bank may be accessed by register addressing. For indirect addressing modes, the registers R0 and R1 are used as pointer or index register to address internal or external memory (e.g. MOV @R0).

2 CPU Architecture

The typical architecture of a C500 family microcontroller is shown in **figure 2-1**. This block diagram includes all main functional blocks of the C500 microcontrollers. The shaded blocks are basic functional units which are mandatory for each C500 microcontroller. The other functional blocks such as XRAM, peripheral units, and ROM/RAM sizes are specific to each C500 microcontroller derivative.
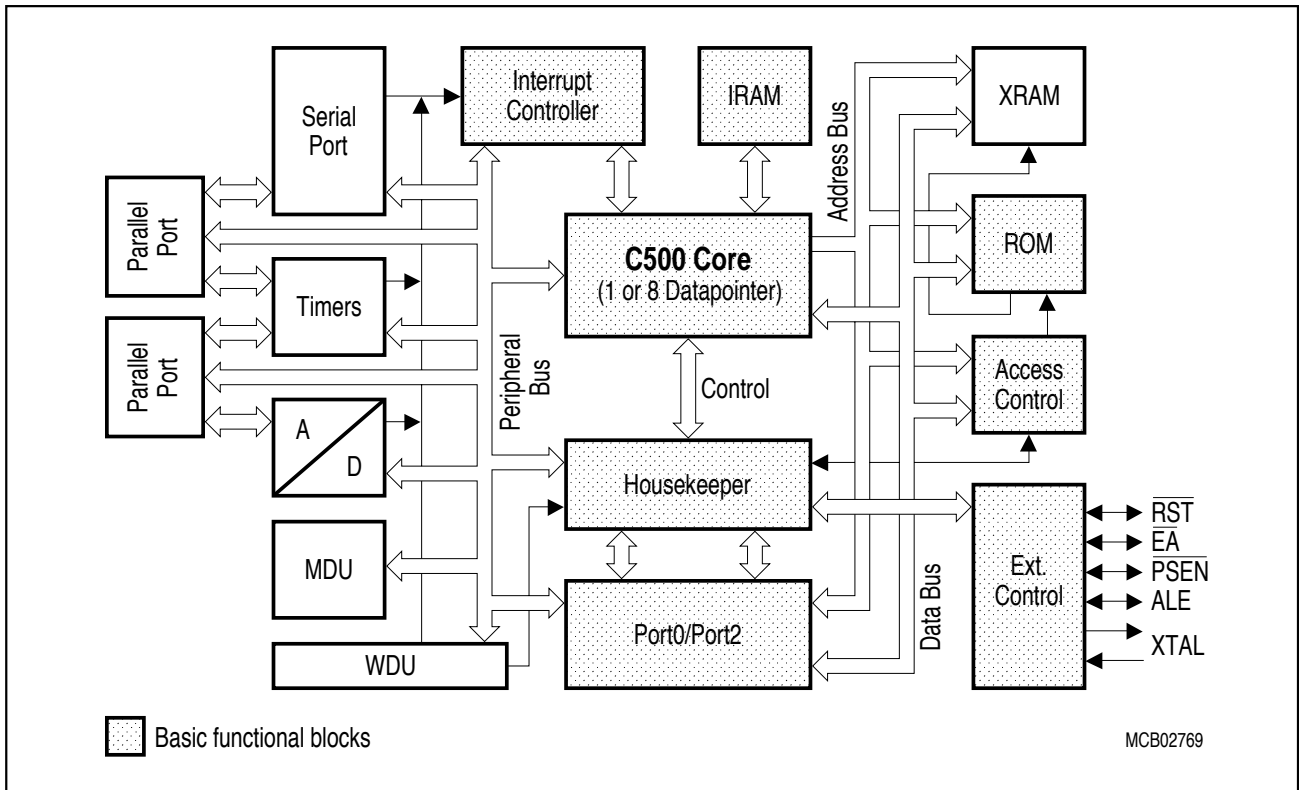
Figure 2-1
C500 Microcontroller Architecture Block Diagram

The core block represents the CPU (Central Processing Unit) of the C500 family microcontrollers. The CPU consists of the instruction decoder, the arithmetic section, the CPU registers, and the program control section. The housekeeper unit generates internal signals for controlling the functions of the individual internal units within the microcontroller. Port 0 and port 2 are required for accessing external code and data memory and for emulation purposes. The external control signals and the clock generation are handled in the external control block. The access control unit is responsible for the selection of the on-chip memory resources. The IRAM provides the internal RAM which includes the general purpose registers. The interrupt requests from the peripheral units are handled by an interrupt controller unit.

C500 device specific is the configuration of the on-chip peripheral units. Serial interfaces, timers, capture/compare units, A/D converters, watchdog units, or a multiply/divide unit are typical examples for on-chip peripheral units. The external signals of these peripheral units are available at multifunctional parallel I/O ports or at dedicated pins.

The arithmetic section of the core performs extensive data manipulation and is comprised of the arithmetic/logic unit (ALU), an A register, B register and PSW register. Further, it has extensive facilities for binary and BCD arithmetic and excels in its bit-handling capabilities. Efficient use of program memory results from an instruction set consisting of 44% one-byte, 41% two-byte, and 15% three-byte instructions. The ALU accepts 8-bit data words from one or two sources and generates an 8-bit result under the control of the instruction decoder. The ALU performs the arithmetic operations add, substract, multiply, divide, increment, decrement, BDC-decimal-add-adjust and compare, and the logic operations AND, OR, Exclusive OR, complement and rotate (right, left or swap nibble (left four)). Also included is a Boolean processor performing the bit operations as set, clear, complement, jump-if-not-set, jump-if-set-and-clear and move to/from carry. Between any addressable bit (or its complement) and the carry flag, it can perform the bit operations of logical AND or logical OR with the result returned to the carry flag.

The program control section of the core controls the sequence in which the instructions stored in program memory are executed. The 16-bit program counter (PC) holds the address of the next instruction to be executed. The conditional branch logic enables internal and external events to the processor to cause a change in the program execution sequence.

2.1   Accumulator

ACC is the symbol for the accumulator register. The mnemonics for accumulator-specific instructions, however, refer to the accumulator simply as A.

2.2   B Register

The B register is used during multiply and divide and serves as both source and destination. For other instructions it can be treated as another scratch pad register.

2.3   Program Status Word

The Program Status Word (PSW) contains several status bits that reflect the current state of the CPU. The bits of the PSW are used for different functions which are: two register bank selection bits, two carry flags and an overflow flag for arithmetic instructions, a parity bit for the content of the ACC, and two general purpose flags.

The bit definitions of the PSW are shown on the next page.

Special Function Register PSW (Address D0$_H$)　　　　　　　　**Reset Value : 00$_H$**

Bit No.	MSB							LSB	
	7	6	5	4	3	2	1	0	
D0$_H$	CY	AC	F0	RS1	RS0	OV	F1	P	PSW

Bit	Function
CY	Carry Flag Used by arithmetic and conditional branch instruction.
AC	Auxiliary Carry Flag Used by instructions which execute BCD operations.
F0	General Purpose Flag
RS1 RS0	Register Bank select control bits These bits are used to select one of the four register banks.

RS1	RS0	Function
0	0	Registerbank 0 at data address 00$_H$-07$_H$ selected
0	1	Registerbank 1 at data address 08$_H$-0F$_H$ selected
1	0	Registerbank 2 at data address 10$_H$-17$_H$ selected
1	1	Registerbank 3 at data address 18$_H$-1F$_H$ selected

Bit	Function
OV	Overflow Flag Used by arithmetic instruction.
F1	General Purpose Flag
P	Parity Flag Always set/cleared by hardware to indicate an odd/even number of "one" bits in the accumulator, i.e. even parity.

2.4　Stack Pointer

The stack pointer (SP) register is 8 bits wide. It is incremented before data is stored during PUSH and CALL executions and decremented after data is popped during a POP and RET (RETI) execution, i.e. it always points to the last valid stack byte. While the stack may reside anywhere in the on-chip RAM, the stack pointer is initialized to 07$_H$ after a reset. This causes the stack to begin a location = 08$_H$ above register bank zero. The SP can be read or written under software control.

2.5 Data Pointer

8-bit accesses to the internal XRAM data memory or the external data memory are executed using the data pointer DPTR as an 16-bit address register. Normally, the C500 family microcontrollers have one data pointer. But some members of the C500 family provide eight data pointers. The availability of eight data pointers especially supports the programming in high level languages which have a demand to store data in large external data memory portions.

Special Function Register DPL (Address 82$_H$)   **Reset Value : 00$_H$**
Special Function Register DPH (Address 83$_H$)   **Reset Value : 00$_H$**
Special Function Register DPSEL (Address D0$_H$)   **Reset Value : 00$_H$**

	MSB							LSB	
Bit No.	7	6	5	4	3	2	1	0	
82$_H$	.7	.6	.5	.4	.3	.2	.1	LSB	DPL
83$_H$	MSB	.6	.5	.4	.3	.2	.1	.0	DPH
92$_H$	–	–	–	–	–	.2	.1	.0	DPSEL

Bit	Function
–	Reserved bits for future use
DPSEL.2 - 0	Data pointer select bits DPSEL.2-0 defines the number of the actual active data pointer.DPTR0-7.

DPSEL2	DPSEL1	DPSEL0	Function
0	0	0	Data pointer 0 selected
0	0	1	Data pointer 1 selected
0	1	0	Data pointer 2 selected
0	1	1	Data pointer 3 selected
1	0	0	Data pointer 4 selected
1	0	1	Data pointer 5 selected
1	1	0	Data pointer 6 selected
1	1	1	Data pointer 7 selected

2.5.1 The Importance of Additional Datapointers

The standard 8051 architecture provides just one 16-bit pointer for indirect addressing of external devices (memories, peripherals, latches, etc.). Except for a 16-bit "move immediate" to this datapointer and an increment instruction, any other pointer handling is to be done byte by byte. For complex applications with peripherals located in the external data memory space (e.g. CAN controller) or extended data storage capacity this turned out to be a "bottle neck" for the 8051's communication to the external world. Especially programming in high-level languages (PLM51, C51, PASCAL51) requires extended RAM capacity and at the same time a fast access to this additional RAM because of the reduced code efficiency of these languages.

2.5.2 How the eight Datapointers of the C500 are realized

Simply adding more datapointers is not suitable because of the need to keep up 100% compatibility to the 8051 instruction set. This instruction set, however, allows the handling of only one single 16-bit datapointer (DPTR, consisting of the two 8-bit SFRs DPH and DPL).

To meet both of the above requirements (speed up external accesses, 100% compatibility to 8051 architecture) the C500 contains a set of eight 16-bit registers from which the actual datapointer can be selected.

This means that the user's program may keep up to eight 16-bit addresses resident in these registers, but only one register at a time is selected to be the datapointer. Thus the datapointer in turn is accessed (or selected) via indirect addressing. This indirect addressing is done through a special function register called DPSEL (data pointer select register). All instructions of the C500 which handle the datapointer therefore affect only one of the eight pointers which is addressed by DPSEL at that very moment.

Figure 5-1 illustrates the addressing mechanism: a 3-bit field in register DPSEL points to the currently used DPTRx. Any standard 8051 instruction (e.g. MOVX @DPTR, A - transfer a byte from accumulator to an external location addressed by DPTR) now uses this activated DPTRx.
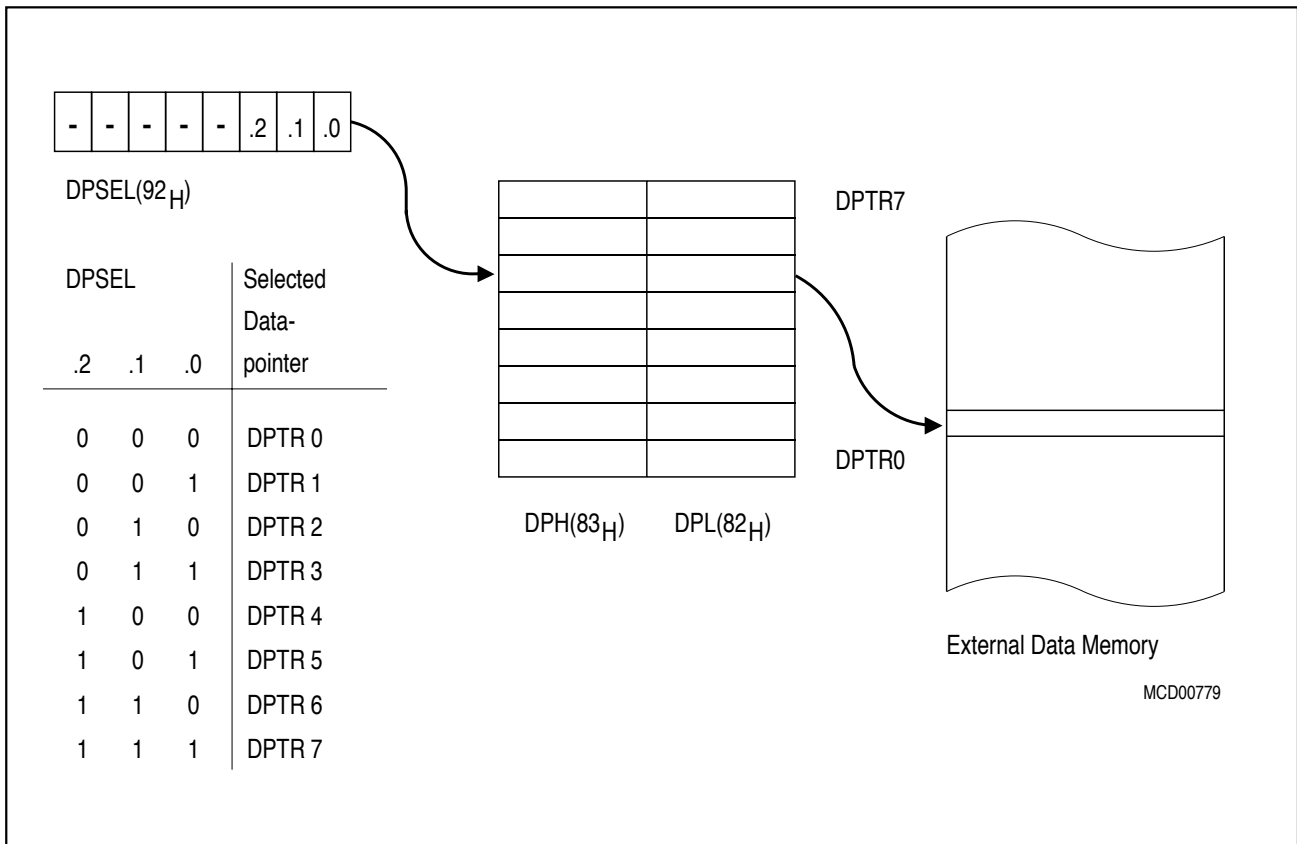
Figure 2-2
Accessing of External Data Memory via Multiple Datapointers

2.5.3 Advantages of Multiple Datapointers

Using the above addressing mechanism for external data memory results in less code and faster execution of external accesses. Whenever the contents of the datapointer must be altered between two or more 16-bit addresses, one single instruction, which selects a new datapointer, does this job. If the program uses just one datapointer, then it has to save the old value (with two 8-bit instructions) and load the new address, byte by byte. This not only takes more time, it also requires additional space in the internal RAM.

2.5.4 Application Example and Performance Analysis

The following example shall demonstrate the involvement of multiple data pointers in a table transfer from the code memory to external data memory.

Start address of ROM source table: $1FFF_H$
Start address of table in external RAM: $2FA0_H$

Example 1 : Using only One Datapointer (Code for a C501)

Initialization Routine

```
MOV     LOW(SRC_PTR), #0FFH        ;Initialize shadow_variables with source_pointer
MOV     HIGH(SRC_PTR), #1FH
MOV     LOW(DES_PTR), #0A0H        ;Initialize shadow_variables with destination_pointer
MOV     HIGH(DES_PTR), #2FH
```

Table Look-up Routine under Real Time Conditions

```
;                                                            Number of cycles
PUSH    DPL              ;Save old datapointer                     2
PUSH    DPH              ;                                         2
MOV     DPL, LOW(SRC_PTR)    ;Load Source Pointer                 2
MOV     DPH, HIGH(SRC_PTR)   ;                                    2
;INC    DPTR             Increment and check for end of table (execution time
;CJNE   …                not relevant for this consideration)     –
MOVC    A,@DPTR          ;Fetch source data byte from ROM table   2
MOV     LOW(SRC_PTR), DPL    ;Save source_pointer and             2
MOV     HIGH(SRC_PTR), DPH   ;load destination_pointer            2
MOV     DPL, LOW(DES_PTR)    ;                                    2
MOV     DPH, HIGH(DES_PTR)   ;                                    2
INC     DPTR             ;Increment destination_pointer
                         ;(ex. time not relevant)                 –
MOVX    @DPTR, A         ;Transfer byte to destination address    2
MOV     LOW(DES_PTR), DPL    ;Save destination_pointer            2
MOV     HIGH(DES_PTR),DPH    ;                                    2
POP     DPH              ;Restore old datapointer                 2
POP     DPL              ;                                         2

;                        Total execution time (machine cycles) :  28
```

Example 2 : Using Two Datapointers (Code for a C509)

Initialization Routine

```
MOV     DPSEL, #06H                  ;Initialize DPTR6 with source pointer
MOV     DPTR, #1FFFH
MOV     DPSEL, #07H                  ;Initialize DPTR7 with destination pointer
MOV     DPTR, #2FA0H
```

Table Look-up Routine under Real Time Conditions

		;	Number of cycles
PUSH	DPSEL	;Save old source pointer	2
MOV	DPSEL, #06H	;Load source pointer	2
;INC	DPTR	Increment and check for end of table (execution time	
;CJNE	…	not relevant for this consideration)	–
MOVC	A,@DPTR	;Fetch source data byte from ROM table	2
MOV	DPSEL, #07H	;Save source_pointer and	
		;load destination_pointer	2
MOVX	@DPTR, A	;Transfer byte to destination address	2
POP	DPSEL	;Save destination pointer and	
		;restore old datapointer	2
;		Total execution time (machine cycles) :	12

The above example shows that utilization of the C500's multiple datapointers can make external bus accesses two times as fast as with a standard 8051 or 8051 derivative. Here, four data variables in the internal RAM and two additional stack bytes were spared, too. This means for some applications where all eight datapointers are employed that an C500 program has up to 24 byte (16 variables and 8 stack bytes) of the internal RAM free for other use.

2.6 Enhanced Hooks Emulation Concept

The Enhanced Hooks Emulation Concept of the C500 microcontroller family is a new, innovative way to control the execution of C500 MCUs and to gain extensive information on the internal operation of the controllers. Emulation of on-chip ROM based programs is possible, too.
Each production chip has built-in logic for the support of the Enhanced Hooks Emulation Concept. Therefore, no costly bond-out chips are necessary for emulation. This also ensure that emulation and production chips are identical.

The Enhanced Hooks Technology^TM, which requires embedded logic in the C500, allows the C500 together with an EH-IC to function similar to a bond-out chip. This simplifies the design and reduces costs of an ICE-system. ICE-systems using an EH-IC and a compatible C500 are able to emulate all operating modes of the different versions of the C500. This includes emulation of ROM, ROM with code rollover and ROMless modes of operation. It is also able to operate in single step mode and to read the SFRs after a break.
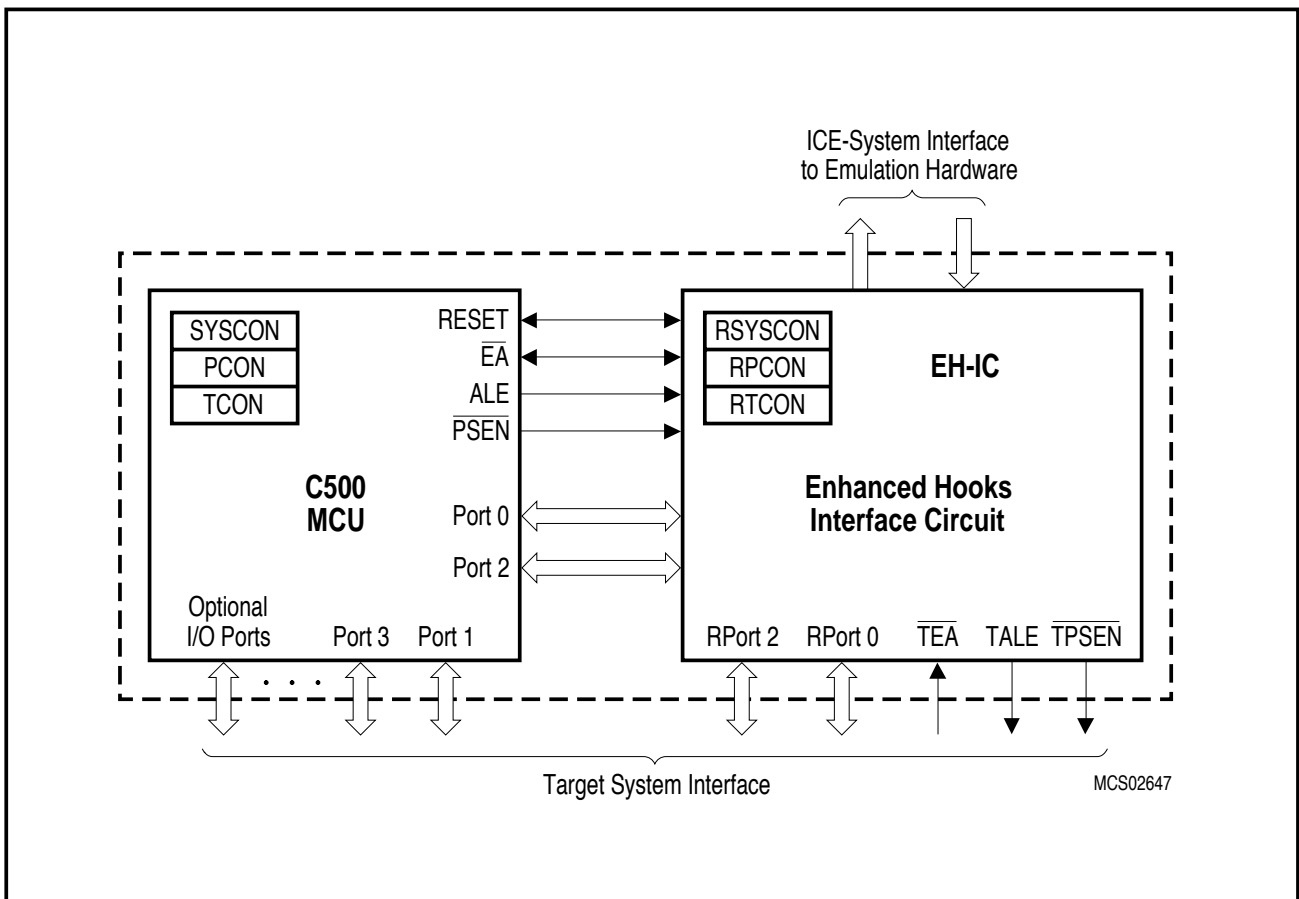
**Figure 2-3
Basic C500 MCU Enhanced Hooks Concept Configuration**

Port 0, port 2 and some of the control lines of the C500 based MCU are used by Enhanced Hooks Emulation Concept to control the operation of the device during emulation and to transfer informations about the program execution and data transfer between the external emulation hardware (ICE-system) and the C500 MCU.

2.7 Basic Interrupt Handling

Each member of the C500 microcontroller family provides several interrupt sources. These interrupts are generated typically by external events or by the internal peripheral units. If an interrupt is accepted by the CPU, the microcontroller interrupts a running program and proceeds the program execution at an interrupt source specific vector address where the interrupt service routine is located. After the execution of a RETI (return from interrupt) instruction the program is continued at the point where it has been interrupted. **Figure 2-4** shows an example for the interrupt vector addresses of a C500 microcontroller (C501). Generally, interrupt vector addresses are located in the code memory area starting at address 0003_H. The minimum distance between two consecutive vector addresses is always 8 bytes. Therefore, interrupt vectors can be assigned to the following addresses: 0003_H, $000B_H$, 0013_H, $001B_H$, 0023_H, $002B_H$, 0033_H ...... $00FB_H$.
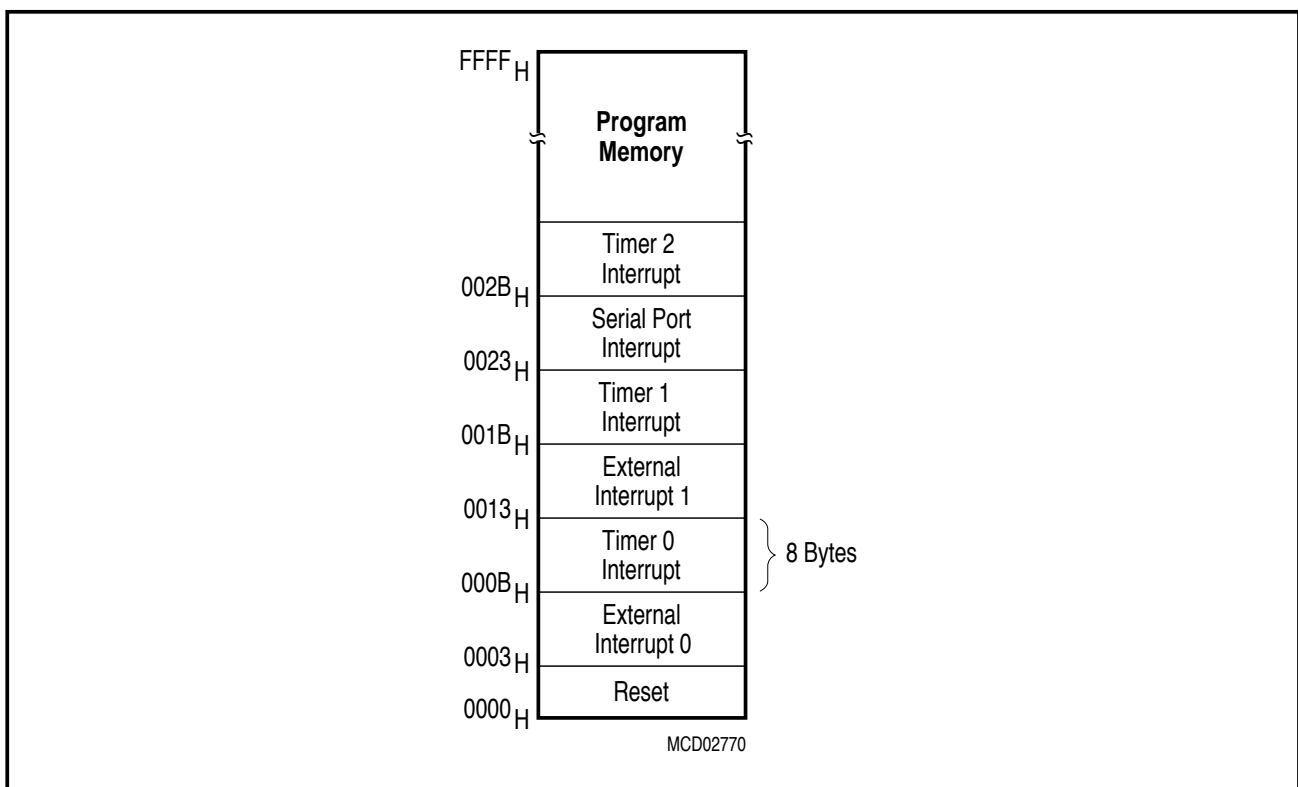
Figure 2-4
Interrupt Vector Addresses (Example of the C501)

An interrupt source indicates to the interrupt controller an interrupt condition by setting an interrupt request flag. The interrupt request flags are sampled in each machine cycle. The sampled flags are polled during the following machine cycle. If one of the flags was in a set condition in the preceeding cycle, the polling cycle will find it and the interrupt controller will cause the CPU to branch to the vector address of the appropriate service routine by generating an internal LCALL. This hardware-generated LCALL is blocked by any of the following conditions:

1. An interrupt of equal or higher priority is already in progress.
2. The current (polling) cycle is not in the final cycle of the instruction in progress.
3. The instruction in progress is RETI or any write access to interrupt enable or priority registers.

Any of these three conditions will block the generation of the LCALL to the interrupt service routine. Condition 2 ensures that the instruction in progress is completed before vectoring to any service routine. Condition 3 ensures that if the instruction in progress is RETI or any write access to interrupt enable or interrupt priority registers, then at least one more instruction will be executed before any interrupt is vectored too; this delay guarantees that changes of the interrupt status can be observed by the interrupt controller.

The polling cycle is repeated with each machine cycle, and the values polled are the values that were present at the previous machine cycle. Note that if any interrupt flag is active but not being responded to for one of the conditions already mentioned, or if the flag is no longer active when the blocking condition is removed, the denied interrupt will not be serviced. In other words, the fact that the interrupt flag was once active but not serviced is not remembered. Every polling cycle interrogates only the pending interrupt requests.

The polling cycle/LCALL sequence is illustrated in **figure 2-1**.
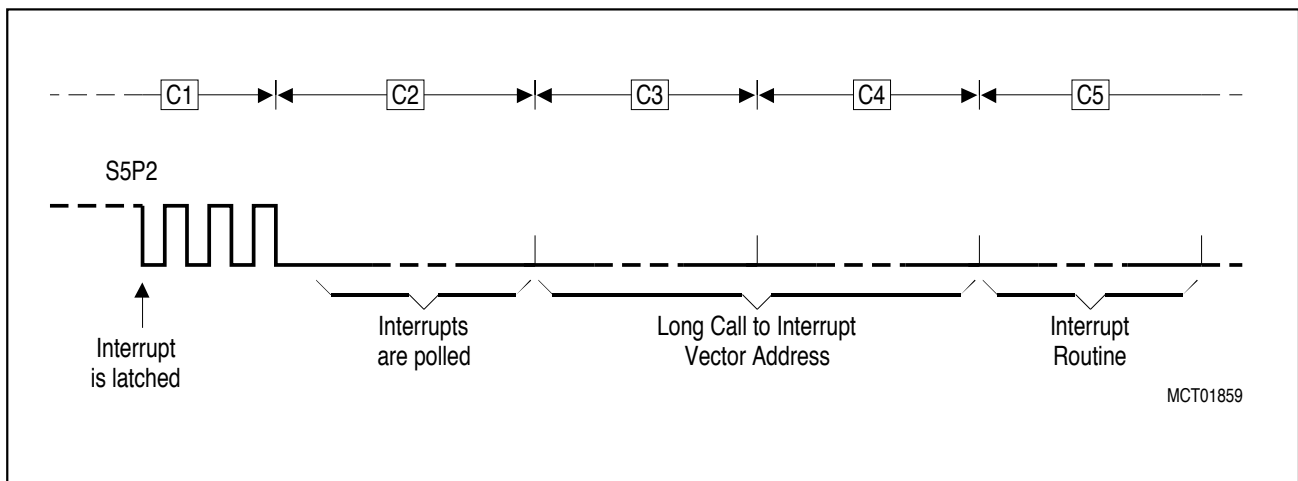
Figure 2-5
Interrupt Detection/Entry Diagram

Note that if an interrupt of a higher priority level goes active prior to S5P2 in the machine cycle labeled C3 in **figure 2-5** then, in accordance with the above rules, it will be vectored to during C5 and C6 without any instruction for the lower priority routine to be executed.

Thus, the processor acknowledges an interrupt request by executing a hardware-generated LCALL to the appropriate servicing routine. In some cases it also clears the flag that generated the interrupt, while in other cases it does not; then this has to be done by the user's software.

The program execution proceeds from that location until the RETI instruction is encountered. The RETI instruction informs the processor that the interrupt routine is no longer in progress, then pops the two top bytes from the stack and reloads the program counter. Execution of the interrupted program continues from the point where it was stopped. Note that the RETI instruction is very important because it informs the processor that the program left the current interrupt priority level. A simple RET instruction would also have returned execution to the interrupted program, but it would have left the interrupt control system thinking an interrupt was still in progress. In this case no interrupt of the same or lower priority level would be acknowledged.

2.8 Interrupt Response Time

If an external interrupt is recognized, its corresponding request flag is set at S5P2 in every machine cycle. The value is not polled by the circuitry until the next machine cycle. If the request is active and conditions are right for it to be acknowledged, a hardware subroutine call to the requested service routine will be next instruction to be executed. The call itself takes two cycles. Thus a minimum of three complete machine cycles will elapse between activation and external interrupt request and the beginning of execution of the first instruction of the service routine.

A longer response time would be obtained if the request was blocked by one of the three previously listed conditions. If an interrupt of equal or higher priority is already in progress, the additional wait time obviously depends on the nature of the other interrupt's service routine. If the instruction in progress is not in its final cycle, the additional wait time cannot be more than 3 cycles since the longest instructions (MUL and DIV) are only 4 cycles long; and, if the instruction in progress is RETI or a write access to interrupt enable or interrupt priority registers the additional wait time cannot be more than 5 cycles (a maximum of one more cycle to complete the instruction in progress, plus 4 cycles to complete the next instruction, if the instruction is MUL or DIV).

Thus a single interrupt system, the response time is always more than 3 cycles and less than 9 cycles.

3   CPU Timing

3.1   Basic Timing

A machine cycle consists of 6 states. Each state is divided into a phase 1 half, during which the phase 1 clock is active, and a phase 2 half, during which the phase 2 clock is active. Thus, a machine cycle consists of the states S1P1 (state 1, phase 1) through S6P2 (state 6, phase 2). Depending on the C500 type of microcontroller, each state lasts either one or two periods of the oscillator clock. Typically, arithmetic and logical operations take place during phase 1 and internal register-to-register transfers take place during phase 2.

The diagrams in **figure 3-1** show the fetch/execute timing related to the internal states and phases. Since these internal clock signals are not user-accessible, the ALE (address latch enable) signal is shown for external reference. ALE is normally activated twice during each machine cycle: once during S1P2 and S2P1, and again during S4P2 and S5P1.

The execution of a one-cycle instruction begins at S1P2, when the opcode is latched into the instruction register. If it is a two-byte instruction, the second reading takes place during S4 of the same machine cycle. If it is a one-byte instruction, there is still a fetch at S4, but the byte read (which would be the next op-code) is ignored (discarded fetch), and the program counter is not incremented. In any case, execution is completed at the end of S6P2.

Figures 3-1 (a) and **(b)** show the timing of a 1-byte, 1-cycle instruction and for a 2-byte, 1-cycle instruction.

Most C500 instructions are executed in one cycle. MUL (multiply) and DIV (divide) are the only instructions that take more than two cycles to complete; they take four cycles. Normally two code bytes are fetched from the program memory during every machine cycle. The only exception to this is when a MOVX instruction is executed. MOVX is a one-byte, 2-cycle instruction that accesses external data memory. During a MOVX, the two fetches in the second cycle are skipped while the external data memory is being addressed and strobed. **Figure 3-1 (c)** and **(d)** show the timing for a normal 1-byte, 2-cycle instruction and for a MOVX instruction.
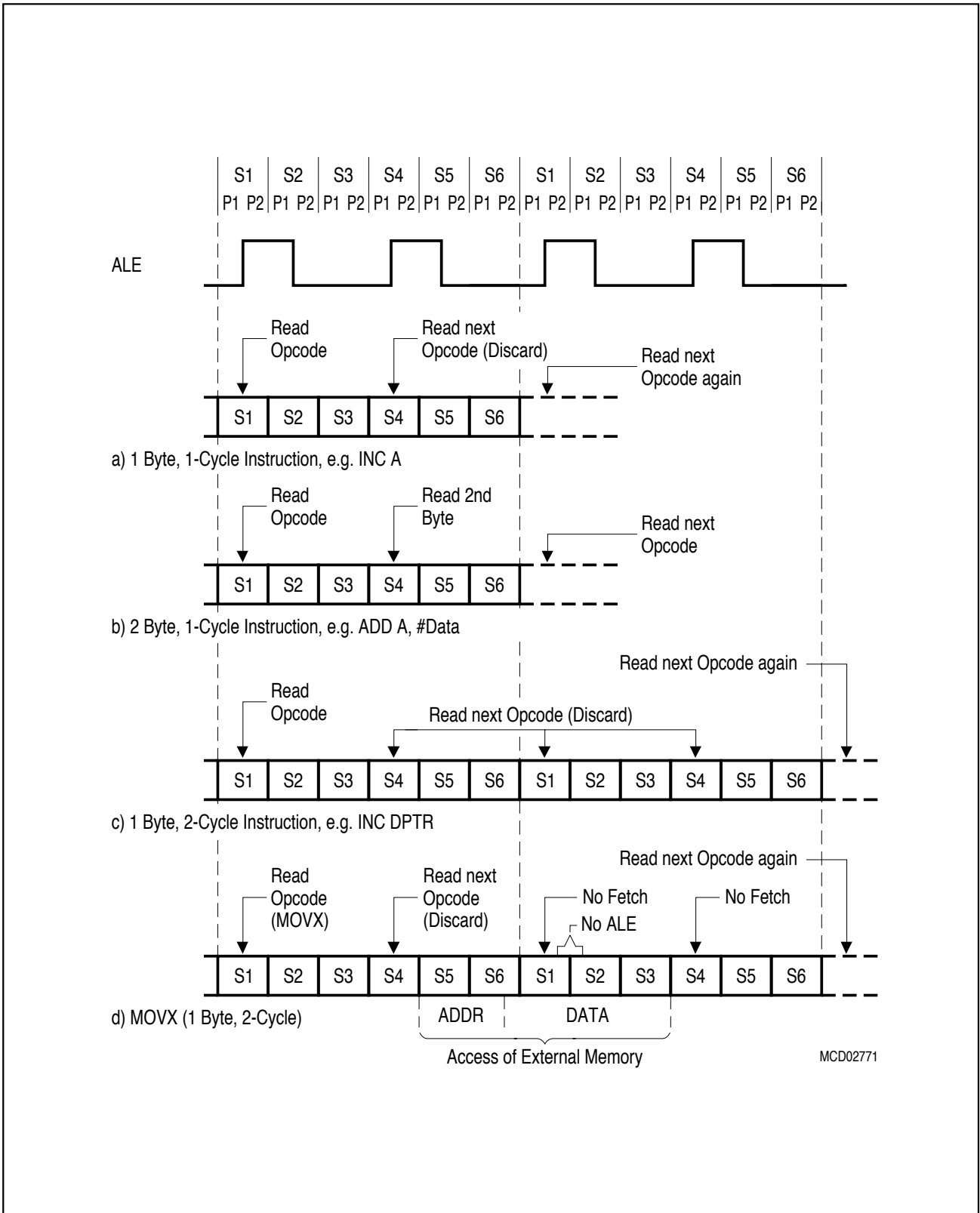
Figure 3-1
Fetch Execute Sequence

3.2   Accessing External Memory

There are two types of external memory accesses: accesses to external program memory and accesses to external data memory. Accesses to external program memory use the signal $\overline{\text{PSEN}}$ (program store enable) as the read strobe. Accesses to external data memory use the $\overline{\text{RD}}$ or $\overline{\text{WR}}$ (alternate functions of P3.7 and P3.6) to access the memory.

Fetches from external program memory always use a 16-bit address. Accesses to external data memory can use either a 16-bit address (MOVX @DPTR) or an 8-bit address (MOVX @Ri). Whenever a 16-bit address is used, the high byte of the address comes out on port 2, where it is held for the duration of the read, write, or code fetch cycle.

If an 8-bit address is being used (MOVX @Ri), the contents of the port 2 SFR remain at the port 2 pins throughout the whole external memory cycle. In this case, port 2 pins can be used to page the external data memory.

In either case, the low byte of the address is time-multiplexed with the data byte on port 0. The ADDRESS/DATA signal drives both FETS in the port 0 output buffers. Thus, in external bus mode the port 0 pins are not open-drain outputs and do not require external pullups. The ALE (address latch enable) signal should be used to latch the address byte into an external latch. The address byte is valid at the negative transition of ALE. Then, in a write cycle, the data byte to be written appears on port 0 just before $\overline{\text{WR}}$ is activated, and remains there until $\overline{\text{WR}}$ is deactivated. In a read cycle, the incoming byte is accepted at port 0 just before the read strobe ($\overline{\text{RD}}$) is deactivated.

During any access to external memory, the CPU writes FF_H to the port 0 latch (the special function register), thus obliterating the information in the port 0 SFR. Also, a MOV P0 instruction must not take place during external memory accesses. If the user writes to port 0 during an external memory fetch, the incoming code byte may be corrupted. Therefore, do not write to port 0 if external memory is used.

3.2.1 Accessing External Program Memory

External program memory is accessed under two conditions:

1. Whenever signal $\overline{\text{EA}}$ is active (low), or
2. Whenever signal $\overline{\text{EA}}$ is inactive (high) and the program counter (PC) contains an address greater than the internal ROM size (e.g. $1FFFF_H$ for an 8K internal ROM or $3FFF_H$ for an 16K internal ROM).

This requires that the ROMless versions have always $\overline{\text{EA}}$ wired to V_{SS} to enable the lower 8K, 16K, or 32K program bytes to be fetched from external memory.

When the CPU is executing out from external program memory (see timing diagram in **figure 3-2**), all 8 bits of port 2 are dedicated to an output function and may not be used for general purpose I/O. During external program fetches they output the high byte of the PC with the port 2 drivers using the strong pullups to emit bits that are 1´s.
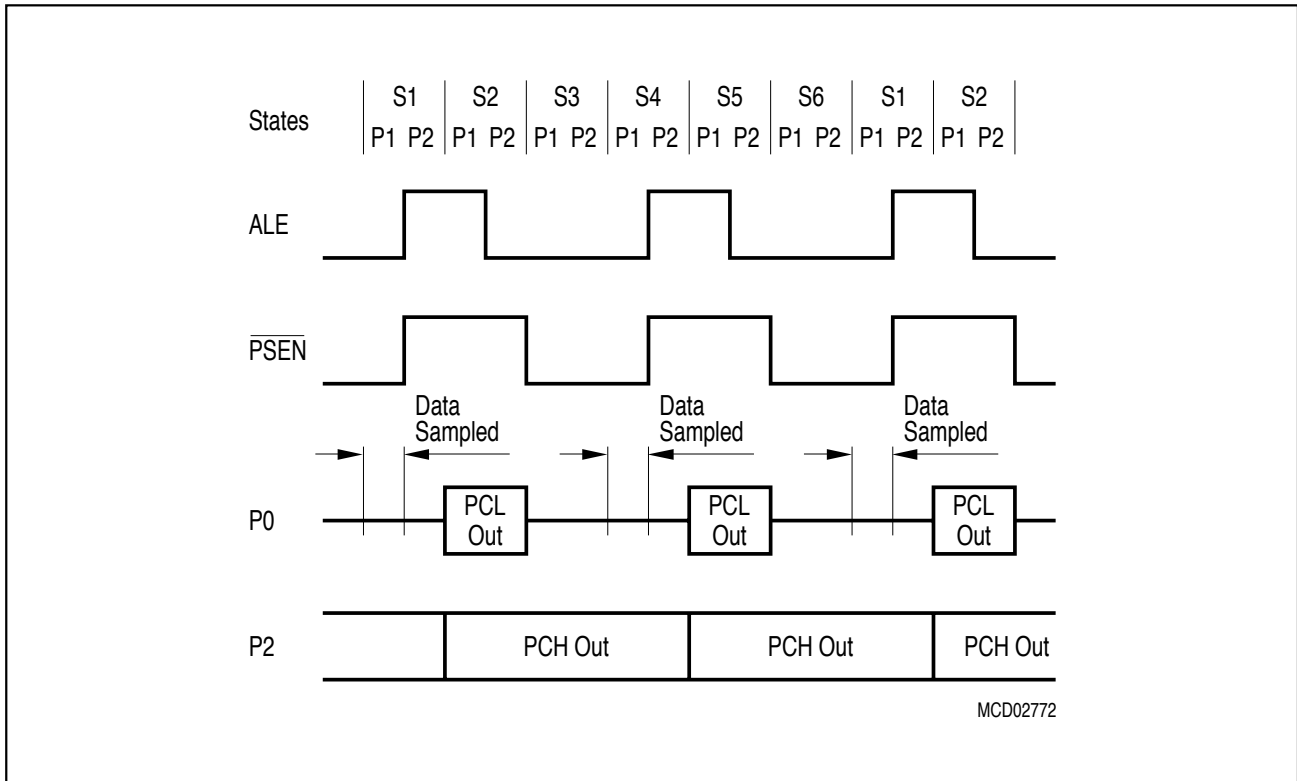
Figure 3-2
External Program Memory Fetches

3.2.2 Accessing External Data Memory

The port 2 drivers use the strong pullups during the entire time that they are emitting address bits that are 1´s. This occurs when the MOVX @DPTR instruction is executed and when external program fetches are executed. During this time the port 2 latch (the special function register) does not have to contain 1´s, and the contents of the port 2 SFR are not modified. If the external memory cycle is not immediately followed by another external memory cycle, the undisturbed contents of the port 2 SFR will reappear in the next cycle.

Figure 3-3 and **3-4** show in detail the timings of the external data memory read and write cycles.
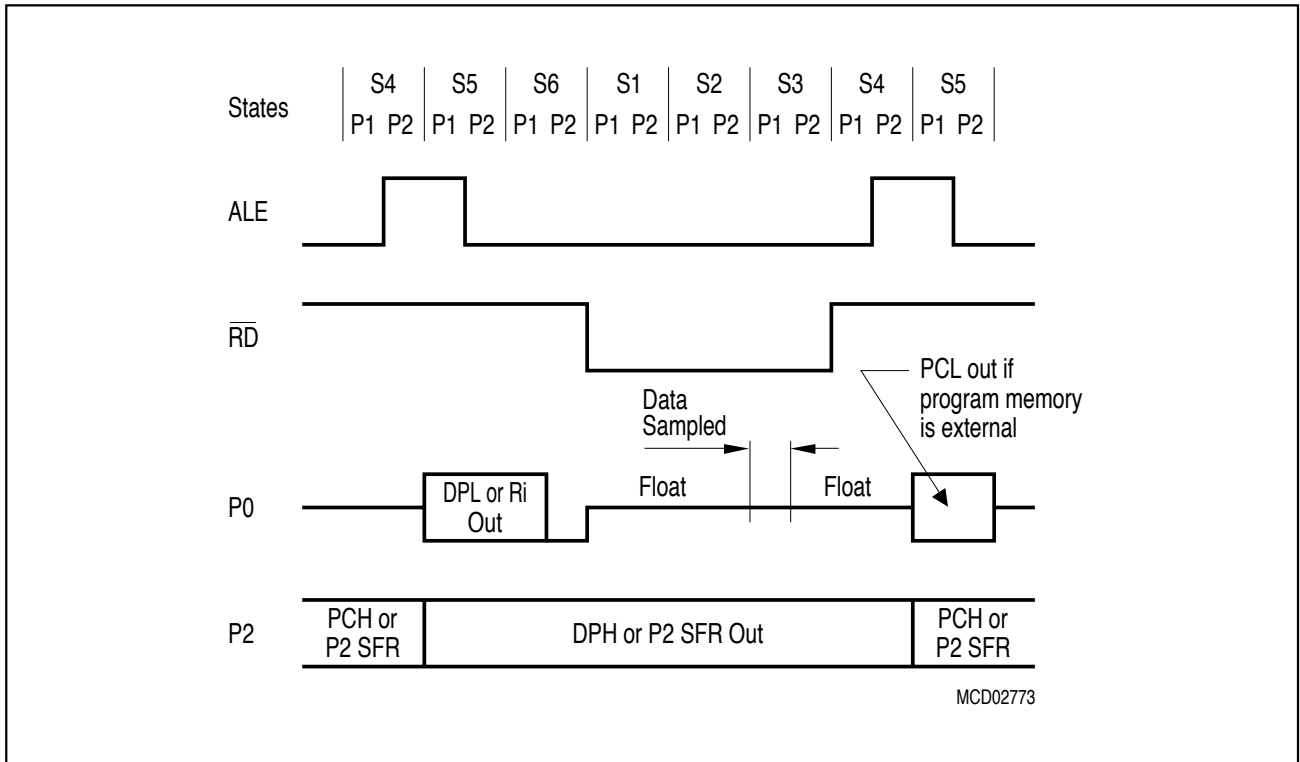
Figure 3-3
External Data Memory Read Cycle
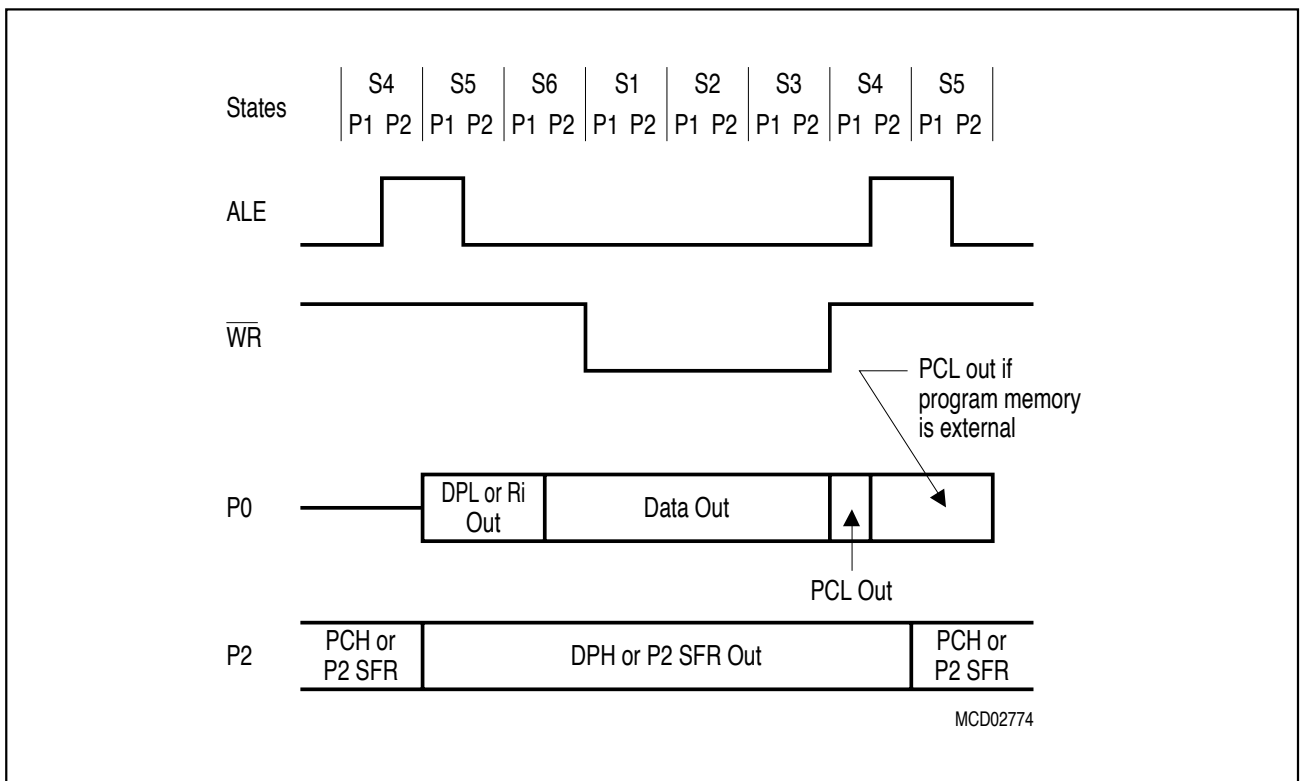
Figure 3-4
External Data Memory Write Cycle

4 Instruction Set

The C500 8-bit microcontroller family instruction set includes 111 instructions, 49 of which are single-byte, 45 two-byte and 17 three-byte instructions. The instruction opcode format consists of a function mnemonic followed by a "destination, source" operand field. This field specifies the data type and addressing method(s) to be used.

Like all other members of the 8051-family, the C500 microcontrollers can be programmed with the same instruction set common to the basic member, the SAB 8051.
Thus, the C500 family microcontrollers are 100% software compatible to the SAB 8051 and may be programmed with 8051 assembler or high-level languages.

4.1 Addressing Modes

The C500 uses five addressing modes:

- register
- direct
- immediate
- register indirect
- base register plus index-register indirect

Table 4-1 summarizes the memory spaces which may be accessed by each of the addressing modes.

Register Addressing

Register addressing accesses the eight working registers (R0 - R7) of the selected register bank. The least significant bit of the instruction opcode indicates which register is to be used. ACC, B, DPTR and CY, the Boolean processor accumulator, can also be addressed as registers.

Direct Addressing

Direct addressing is the only method of accessing the special function registers. The lower 128 bytes of internal RAM are also directly addressable.

Immediate Addressing

Immediate addressing allows constants to be part of the instruction in program memory.

Table 4-1
Addressing Modes and Associated Memory Spaces

Addressing Modes	Associated Memory Spaces
Register addressing	R0 through R7 of selected register bank, ACC, B, CY (Bit), DPTR
Direct addressing	Lower 128 bytes of internal RAM, special function registers
Immediate addressing	Program memory
Register indirect addressing	Internal RAM (@R1, @R0, SP), external data memory (@R1, @R0, @DPTR)
Base register plus index register addressing	Program memory (@A + DPTR, @A + PC)

Register Indirect Addressing

Register indirect addressing uses the contents of either R0 or R1 (in the selected register bank) as a pointer to locations in a 256-byte block: the 256 bytes of internal RAM or the lower 256 bytes of external data memory. Note that the special function registers are not accessible by this method. The upper half of the internal RAM can be accessed by indirect addressing only. Access to the full 64 Kbytes of external data memory address space is accomplished by using the 16-bit data pointer. Execution of PUSH and POP instructions also uses register indirect addressing. The stack may reside anywhere in the internal RAM.

Base Register plus Index Register Addressing

Base register plus index register addressing allows a byte to be accessed from program memory via an indirect move from the location whose address is the sum of a base register (DPTR or PC) and index register, ACC. This mode facilitates look-up table accesses.

Boolean Processor

The Boolean processor is a bit processor integrated into the C500 family microcontrollers. It has its own instruction set, accumulator (the carry flag), bit-addressable RAM and I/O.

The bit manipulation instructions allow:

– set bit
– clear bit
– complement bit
– jump if bit is set
– jump if bit is not set
– jump if bit is set and clear bit
– move bit from / to carry

Addressable bits, or their complements, may be logically AND-ed or OR-ed with the contents of the carry flag. The result is returned to the carry register.

4.2 Introduction to the Instruction Set

The instruction set is divided into four functional groups:

– data transfer
– arithmetic
– logic
– control transfer

4.2.1 Data Transfer Instructions

Data transfer operations are divided into three classes:

– general-purpose
– accumulator-specific
– address-object

None of these operations affects the PSW flag settings except a POP or MOV directly to the PSW.

General-Purpose Transfers

– MOV performs a bit or byte transfer from the source operand to the destination operand.
– PUSH increments the SP register and then transfers a byte from the source operand to the stack location currently addressed by SP.
– POP transfers a byte operand from the stack location addressed by the SP to the destination operand and then decrements SP.

Accumulator-Specific Transfers

– XCH exchanges the byte source operand with register A (accumulator).
– XCHD exchanges the low-order nibble of the source operand byte with the low-order nibble of A.
– MOVX performs a byte move between the external data memory and the accumulator. The external address can be specified by the DPTR register (16 bit) or the R1 or R0 register (8 bit).
– MOVC moves a byte from program memory to the accumulator. The operand in A is used as an index into a 256-byte table pointed to by the base register (DPTR or PC). The byte operand accessed is transferred to the accumulator.

Address-Object Transfer

– MOV DPTR, #data loads 16 bits of immediate data into a pair of destination registers, DPH and DPL.

4.2.2   Arithmetic Instructions

The C500 family microcontrollers have four basic mathematical operations. Only 8-bit operations using unsigned arithmetic are supported directly. The overflow flag, however, permits the addition and subtraction operation to serve for both unsigned and signed binary integers. Arithmetic can also be performed directly on packed BCD representations.

Addition

– INC (increment) adds one to the source operand and puts the result in the operand (flags in PSW are not affected).
– ADD adds A to the source operand and returns the result to A.
– ADDC (add with carry) adds A and the source operand, then adds one (1) if CY is set, and puts the result in A.
– DA (decimal-add-adjust for BCD addition) corrects the sum which results from the binary addition of two-digit decimal operands. The packed decimal sum formed by DA is returned to A. CY is set if the BCD result is greater than 99; otherwise, it is cleared.

Subtraction

– SUBB (subtract with borrow) subtracts the second source operand from the first operand (the accumulator), subtracts one (1) if CY is set and returns the result to A.
– DEC (decrement) subtracts one (1) from the source operand and returns the result to the operand (flags in PSW are not affected).

Multiplication

– MUL performs an unsigned multiplication of the A register by the B register, returning a double byte result. A receives the low-order byte, B receives the high-order byte. OV is cleared if the top half of the result is zero and is set if it is not zero. CY is cleared. AC is unaffected.

Division

– DIV performs an unsigned division of the A register by the B register; it returns the integer quotient to the A register and returns the fractional remainder to the B register. Division by zero leaves indeterminate data in registers A and B and sets OV; otherwise, OV is cleared. CY is cleared. AC remains unaffected.

Flags

Unless otherwise stated in the previous descriptions, the flags of PSW are affected as follows:

– CY is set if the operation causes a carry to or a borrow from the resulting high-order bit; otherwise CY is cleared.
– AC is set if the operation results in a carry from the low-order four bits of the result (during addition), or a borrow from the high-order bits to the low-order bits (during subtraction); otherwise AC is cleared.
– OV is set if the operation results in a carry to the high-order bit of the result but not a carry from the bit, or vice versa; otherwise OV is cleared. OV is used in two's-complement arithmetic, because it is set when the signal result cannot be represented in 8 bits.
– P is set if the modulo-2 sum of the eight bits in the accumulator is 1 (odd parity); otherwise P is cleared (even parity). When a value is written to the PSW register, the P bit remains unchanged, as it always reflects the parity of A.

4.2.3 Logic Instructions

The C500 family microcontrollers perform basic logic operations on both bit and byte operands.

Single-Operand Operations

– CLR sets A or any directly addressable bit to zero (0).
– SETB sets any directly bit-addressable bit to one (1).
– CPL is used to complement the contents of the A register without affecting any flag, or any directly addressable bit location.
– RL, RLC, RR, RRC, SWAP are the five operations that can be performed on A. RL, rotate left, RR, rotate right, RLC, rotate left through carry, RRC, rotate right through carry, and SWAP, rotate left four. For RLC and RRC the CY flag becomes equal to the last bit rotated out. SWAP rotates A left four places to exchange bits 3 through 0 with bits 7 through 4.

Two-Operand Operations

– ANL performs bitwise logical AND of two operands (for both bit and byte operands) and returns the result to the location of the first operand.
– ORL performs bitwise logical OR of two source operands (for both bit and byte operands) and returns the result to the location of the first operand.
– XRL performs logical Exclusive OR of two source operands (byte operands) and returns the result to the location of the first operand.

4.2.4 Control Transfer Instructions

There are three classes of control transfer operations: unconditional calls, returns, jumps, conditional jumps, and interrupts. All control transfer operations, some upon a specific condition, cause the program execution to continue a non-sequential location in program memory.

Unconditional Calls, Returns and Jumps

Unconditional calls, returns and jumps transfer control from the current value of the program counter to the target address. Both direct and indirect transfers are supported.

– ACALL and LCALL push the address of the next instruction onto the stack and then transfer control to the target address. ACALL is a 2-byte instruction used when the target address is in the current 2K page. LCALL is a 3-byte instruction that addresses the full 64K program space. In ACALL, immediate data (i.e. an 11-bit address field) is concatenated to the five most significant bits of the PC (which is pointing to the next instruction). If ACALL is in the last 2 bytes of a 2K page then the call will be made to the next page since the PC will have been incremented to the next instruction prior to execution.
– RET transfers control to the return address saved on the stack by a previous call operation and decrements the SP register by two (2) to adjust the SP for the popped address.
– AJMP, LJMP and SJMP transfer control to the target operand. The operation of AJMP and LJMP are analogous to ACALL and LCALL. The SJMP (short jump) instruction provides for transfers within a 256-byte range centered about the starting address of the next instruction (– 128 to + 127).
– JMP @A + DPTR performs a jump relative to the DPTR register. The operand in A is used as the offset (0 - 255) to the address in the DPTR register. Thus, the effective destination for a jump can be anywhere in the program memory space.

Conditional Jumps

Conditional jumps perform a jump contingent upon a specific condition. The destination will be within a 256-byte range centered about the starting address of the next instruction (– 128 to + 127).

– JZ performs a jump if the accumulator is zero.
– JNZ performs a jump if the accumulator is not zero.
– JC performs a jump if the carry flag is set.
– JNC performs a jump if the carry flag is not set.
– JB performs a jump if the directly addressed bit is set.
– JNB performs a jump if the directly addressed bit is not set.
– JBC performs a jump if the directly addressed bit is set and then clears the directly addressed bit.
– CJNE compares the first operand to the second operand and performs a jump if they are not equal. CY is set if the first operand is less than the second operand; otherwise it is cleared. Comparisons can be made between A and directly addressable bytes in internal data memory or an immediate value and either A, a register in the selected register bank, or a register indirectly addressable byte of the internal RAM.
– DJNZ decrements the source operand and returns the result to the operand. A jump is performed if the result is not zero. The source operand of the DJNZ instruction may be any directly addressable byte in the internal data memory. Either direct or register addressing may be used to address the source operand.

Interrupt Returns

– RETI transfers control as RET does, but additionally enables interrupts of the current priority level.

4.3 Instruction Definitions

All 111 instructions of the C500 family microcontrollers can essentially be condensed to 53 basic operations, in the following alphabetically ordered according to the operation mnemonic section.

**Table 4-2
PSW Flag Modification (CY,OV,AC)**

Instruction	Flag			Instruction	Flag		
	CY	**OV**	**AC**		**CY**	**OV**	**AC**
ADD	X	X	X	SETB C	1		
ADDC	X	X	X	CLR C	0		
SUBB	X	X	X	CPL C	X		
MUL	0	X		ANL C,bit	X		
DIV	0	X		ANL C,/bit	X		
DA	X			ORL C,bit	X		
RRC	X			ORL C,/bit	X		
RLC	X			MOV C,bit	X		
CJNE	X						

A brief example of how the instruction might be used is given as well as its effect on the PSW flags. The number of bytes and machine cycles required, the binary machine language encoding, and a symbolic description or restatement of the function is also provided.

Note:

Only the carry, auxiliary carry, and overflow flags are discussed. The parity bit is always computed from the actual content of the accumulator.

Similarly, instructions which alter directly addressed registers could affect the other status flags if the instruction is applied to the PSW. Status flags can also be modified by bit manipulation.

Notes on Data Addressing Modes:

Rn	-	Working register R0-R7
direct	-	128 internal RAM locations, any I/O port, control or status register
@Ri	-	Indirect internal or external RAM location addressed by register R0 or R1
#data	-	8-bit constant included in instruction
#data 16	-	16-bit constant included as bytes 2 and 3 of instruction
bit	-	128 software flags, any bit-addressable I/O pin, control or status bit
A	-	Accumulator

Notes on Program Addressing Modes:

addr16	-	Destination address for LCALL and LJMP may be anywhere within the 64-Kbyte program memory address space.
addr11	-	Destination address for ACALL and AJMP will be within the same 2-Kbyte page of program memory as the first byte of the following instruction.
rel	-	SJMP and all conditional jumps include an 8-bit offset byte. Range is + 127/– 128 bytes relative to the first byte of the following instruction.

All mnemonics copyrighted: © Intel Corporation 1980

ACALL   addr11

Function: Absolute call

Description: ACALL unconditionally calls a subroutine located at the indicated address. The instruction increments the PC twice to obtain the address of the following instruction, then pushes the 16-bit result onto the stack (low-order byte first) and increments the stack pointer twice. The destination address is obtained by successively concatenating the five high-order bits of the incremented PC, op code bits 7-5, and the second byte of the instruction. The subroutine called must therefore start within the same 2K block of program memory as the first byte of the instruction following ACALL. No flags are affected.

Example: Initially SP equals 07_H. The label "SUBRTN" is at program memory location 0345_H. After executing the instruction

ACALL   SUBRTN

at location 0123_H, SP will contain 09_H, internal RAM location 08_H and 09_H will contain 25_H and 01_H, respectively, and the PC will contain 0345_H.

Operation: ACALL
$(PC) \leftarrow (PC) + 2$
$(SP) \leftarrow (SP) + 1$
$((SP)) \leftarrow (PC7-0)$
$(SP) \leftarrow (SP) + 1$
$((SP)) \leftarrow (PC15-8)$
$(PC10-0) \leftarrow$ page address

Encoding:

a10 a9 a8 1	0 0 0 1		a7 a6 a5 a4	a3 a2 a1 a0

Bytes: 2

Cycles: 2

ADD   A, <src-byte>

Function: Add

Description: ADD adds the byte variable indicated to the accumulator, leaving the result in the accumulator. The carry and auxiliary carry flags are set, respectively, if there is a carry out of bit 7 or bit 3, and cleared otherwise. When adding unsigned integers, the carry flag indicates an overflow occurred.

OV is set if there is a carry out of bit 6 but not out of bit 7, or a carry out of bit 7 but not out of bit 6; otherwise OV is cleared. When adding signed integers, OV indicates a negative number produced as the sum of two positive operands, or a positive sum from two negative operands.

Four source operand addressing modes are allowed: register, direct, register-indirect, or immediate.

Example: The accumulator holds $0C3_H$ (11000011_B) and register 0 holds $0AA_H$ (10101010_B).
The instruction

ADD   A,R0

will leave $6D_H$ (01101101_B) in the accumulator with the AC flag cleared and both the carry flag and OV set to 1.

ADD   A,Rn

Operation: ADD
             $(A) \leftarrow (A) + (Rn)$

Encoding:

0	0	1	0	1	r	r	r

Bytes: 1

Cycles: 1

ADD   A,direct

Operation: ADD
             $(A) \leftarrow (A) + (direct)$

Encoding:

0 0 1 0	0 1 0 1	direct address

Bytes: 2

Cycles: 1

ADD   A, @Ri

Operation: ADD
                (A) ← (A) + ((Ri))

Encoding: | 0 0 1 0 | 0 1 1 i |

Bytes: 1

Cycles: 1

ADD   A, #data

Operation: ADD
                (A) ← (A) + #data

Encoding: | 0 0 1 0 | 0 1 0 0 | | immediate data |

Bytes: 2

Cycles: 1

ADDC   A, < src-byte>

Function: Add with carry

Description: ADDC simultaneously adds the byte variable indicated, the carry flag and the accumulator contents, leaving the result in the accumulator. The carry and auxiliary carry flags are set, respectively, if there is a carry out of bit 7 or bit 3, and cleared otherwise. When adding unsigned integers, the carry flag indicates an overflow occurred.

OV is set if there is a carry out of bit 6 but not out of bit 7, or a carry out of bit 7 but not out of bit 6; otherwise OV is cleared. When adding signed integers, OV indicates a negative number produced as the sum of two positive operands or a positive sum from two negative operands.

Four source operand addressing modes are allowed: register, direct, register-indirect, or immediate.

Example: The accumulator holds $0C3_H$ (11000011_B) and register 0 holds $0AA_H$ (10101010_B) with the carry flag set. The instruction

ADDC   A,R0

will leave $6E_H$ (01101110_B) in the accumulator with AC cleared and both the carry flag and OV set to 1.

ADDC   A,Rn

Operation: ADDC
$(A) \leftarrow (A) + (C) + (Rn)$

Encoding:

0 0 1 1	1 r r r

Bytes: 1

Cycles: 1

ADDC   A,direct

Operation: ADDC
$(A) \leftarrow (A) + (C) + (direct)$

Encoding:

0 0 1 1	0 1 0 1		direct address

Bytes: 2

Cycles: 1

ADDC   A, @Ri

Operation: ADDC
                $(A) \leftarrow (A) + (C) + ((Ri))$

Encoding: | 0 0 1 1 | 0 1 1 i |

Bytes: 1

Cycles: 1

ADDC   A, #data

Operation: ADDC
                $(A) \leftarrow (A) + (C) + \#data$

Encoding: | 0 0 1 1 | 0 1 0 0 | | immediate data |

Bytes: 2

Cycles: 1

AJMP   addr11

Function: Absolute jump

Description: AJMP transfers program execution to the indicated address, which is formed at run-
time by concatenating the high-order five bits of the PC (after incrementing the PC
twice), op code bits 7-5, and the second byte of the instruction. The destination must
therefore be within the same 2K block of program memory as the first byte of the
instruction following AJMP.

Example: The label "JMPADR" is at program memory location 0123_H. The instruction

AJMP   JMPADR

is at location 0345_H and will load the PC with 0123_H.

Operation: AJM P
$(PC) \leftarrow (PC) + 2$
$(PC10-0) \leftarrow$ page address

Encoding:

a10 a9 a8 0	0 0 0 1		a7 a6 a5 a4	a3 a2 a1 a0

Bytes: 2

Cycles: 2

ANL   <dest-byte>, <src-byte>

Function: Logical AND for byte variables

Description: ANL performs the bitwise logical AND operation between the variables indicated and stores the results in the destination variable. No flags are affected (except P, if <dest-byte> = A).

The two operands allow six addressing mode combinations. When the destination is a accumulator, the source can use register, direct, register-indirect, or immediate addressing; when the destination is a direct address, the source can be the accumulator or immediate data.

Note:

When this instruction is used to modify an output port, the value used as the original port data will be read from the output data latch, not the input pins.

Example: If the accumulator holds $0C3_H$ (11000011_B) and register 0 holds $0AA_H$ (10101010_B) then the instruction

ANL   A,R0

will leave 81_H (10000001_B) in the accumulator.

When the destination is a directly addressed byte, this instruction will clear combinations of bits in any RAM location or hardware register. The mask byte determining the pattern of bits to be cleared would either be a constant contained in the instruction or a value computed in the accumulator at run-time.
The instruction

ANL   P1, #01110011_B

will clear bits 7, 3, and 2 of output port 1.

ANL   A,Rn

Operation: ANL
                (A) ← (A) ∧ (Rn)

Encoding:

0 1 0 1	1 r r r

Bytes: 1

Cycles: 1

ANL   A,direct

Operation: ANL
                (A) ← (A) ∧ (direct)

Encoding:

0 1 0 1	0 1 0 1		direct address

Bytes: 2

Cycles: 1

ANL   A, @Ri

Operation: ANL
               $(A) \leftarrow (A) \wedge ((Ri))$

Encoding: | 0 1 0 1 | 0 1 1 i |

Bytes: 1

Cycles: 1

ANL   A, #data

Operation: ANL
               $(A) \leftarrow (A) \wedge$ #data

Encoding: | 0 1 0 1 | 0 1 0 0 | | immediate data |

Bytes: 2

Cycles: 1

ANL   direct,A

Operation: ANL
               $(direct) \leftarrow (direct) \wedge (A)$

Encoding: | 0 1 0 1 | 0 0 1 0 | | direct address |

Bytes: 2

Cycles: 1

ANL   direct, #data

Operation: ANL
                (direct) ← (direct) ∧ #data

Encoding: | 0 1 0 1 | 0 0 1 1 | | direct address | | immediate data |

Bytes: 3

Cycles: 2

ANL   C, <src-bit>

Function: Logical AND for bit variables

Description: If the Boolean value of the source bit is a logic 0 then clear the carry flag; otherwise leave the carry flag in its current state. A slash ("/") preceding the operand in the assembly language indicates that the logical complement of the addressed bit is used as the source value, but the source bit itself is not affected. No other flags are affected.

Only direct bit addressing is allowed for the source operand.

Example: Set the carry flag if, and only if, P1.0 = 1, ACC.7 = 1, and OV = 0:

MOV	C,P1.0	; Load carry with input pin state
ANL	C,ACC.7	; AND carry with accumulator bit 7
ANL	C,/OV	; AND with inverse of overflow flag

ANL   C,bit

Operation: ANL
               (C) ← (C) ∧ (bit)

Encoding:

1 0 0 0	0 0 1 0		bit address

Bytes: 2

Cycles: 2

ANL   C,/bit

Operation: ANL
               (C) ← (C) ∧ / (bit)

Encoding:

1 0 1 1	0 0 0 0		bit address

Bytes: 2

Cycles: 2

CJNE   **<dest-byte >, < src-byte >, rel**

Function: Compare and jump if not equal

Description: CJNE compares the magnitudes of the tirst two operands, and branches if their values are not equal. The branch destination is computed by adding the signed relative displacement in the last instruction byte to the PC, after incrementing the PC to the start of the next instruction. The carry flag is set if the unsigned integer value of <dest-byte> is less than the unsigned integer value of <src-byte>; otherwise, the carry is cleared. Neither operand is affected.

The first two operands allow four addressing mode combinations: the accumulator may be compared with any directly addressed byte or immediate data, and any indirect RAM location or working register can be compared with an immediate constant.

Example: The accumulator contains 34_H. Register 7 contains 56_H. The first instruction in the sequence

	CJNE	R7, # 60H, NOT_EQ	
;	. . .	. . . . .	; R7 = 60_H
NOT_EQ	JC	REQ_LOW	; If R7 < 60_H
;	. . .	. . . . .	; R7 > 60_H

sets the carry flag and branches to the instruction at label NOT_EQ. By testing the carry flag, this instruction determines whether R7 is greater or less than 60_H.

If the data being presented to port 1 is also 34_H, then the instruction

WAIT: CJNE   A,P1,WAIT

clears the carry flag and continues with the next instruction in sequence, since the accumulator does equal the data read from P1. (If some other value was input on P1, the program will loop at this point until the P1 data changes to 34_H).

CJNE   **A,direct,rel**

Operation: (PC) ← (PC) + 3
if (A) < > (direct)
then (PC) ← (PC) + relative offset
if (A) < (direct)
then (C) ←1
else (C) ← 0

Encoding:

1 0 1 1	0 1 0 1		direct address		rel. address

Bytes: 3

Cycles: 2

CJNE   **A, #data,rel**

Operation: (PC) ← (PC) + 3
if (A) < > data
then (PC) ← (PC) + relative offset
if (A) ← data
then (C) ←1
else (C) ← 0

Encoding:

1 0 1 1	0 1 0 0		immediate data		rel. address

Bytes: 3

Cycles: 2

CJNE   **RN, #data, rel**

Operation: (PC) ← (PC) + 3
if (Rn) < > data
then (PC) ← (PC) + relative offset
if (Rn) < data
then (C) ← 1
else (C) ← 0

Encoding:

1 0 1 1	1 r r r		immediate data		rel. address

Bytes: 3

Cycles: 2

CJNE   **@Ri, #data, rel**

Operation: $(PC) \leftarrow (PC) + 3$
if $((Ri)) <> data$
then $(PC) \leftarrow (PC) + relative\ offset$
if $((Ri)) < data$
then $(C) \leftarrow 1$
else $(C) \leftarrow 0$

Encoding:

1 0 1 1	0 1 1 i		immediate data		rel. address

Bytes: 3

Cycles: 2

CLR   A

Function:	Clear accumulator
Description:	The accumulator is cleared (all bits set to zero). No flags are affected.
Example:	The accumulator contains $5C_H$ (01011100_B). The instruction
	CLR A
	will leave the accumulator set to 00_H (00000000_B).
Operation:	CLR
	$(A) \leftarrow 0$

Encoding:

1 1 1 0	0 1 0 0

Bytes:	1
Cycles:	1

CLR   bit

Function: Clear bit

Description: The indicated bit is cleared (reset to zero). No other flags are affected. CLR can operate on the carry flag or any directly addressable bit.

Example: Port 1 has previously been written with $5D_H$ (01011101_B). The instruction

CLR   P1.2

will leave the port set to 59_H (01011001_B).

CLR   C

Operation: CLR
                $(C) \leftarrow 0$

Encoding: | 1 1 0 0 | 0 0 1 1 |
                |---------|---------|

Bytes: 1

Cycles: 1

CLR   bit

Operation: CLR
                $(bit) \leftarrow 0$

Encoding: | 1 1 0 0 | 0 0 1 0 | | bit address |
                |---------|---------| |-------------|

Bytes: 2

Cycles: 1

CPL   A

Function: Complement accumulator

Description: Each bit of the accumulator is logically complemented (one's complement). Bits which previously contained a one are changed to zero and vice versa. No flags are affected.

Example: The accumulator contains $5C_H$ (01011100_B). The instruction

CPL   A

will leave the accumulator set to $0A3_H$ (10100011_B).

Operation: CPL
$(A) \leftarrow / (A)$

Encoding:

1 1 1 1	0 1 0 0

Bytes: 1

Cycles: 1

CPL   bit

Function: Complement bit

Description: The bit variable specified is complemented. A bit which had been a one is changed to zero and vice versa. No other flags are affected. CPL can operate on the carry or any directly addressable bit.

Note:

When this instruction is used to modify an output pin, the value used as the original data will be read from the output data latch, not the input pin.

Example: Port 1 has previously been written with $5D_H$ (01011101_B). The instruction sequence

CPL   P1.1
CPL   P1.2

will leave the port set to $5B_H$ (01011011_B).

CPL   C

Operation: CPL
                (bit) ← / (C)

Encoding:

1	0	1	1	0	0	1	1

Bytes: 1

Cycles: 1

CPL   bit

Operation: CPL
                (C) ← (bit)

Encoding:

1	0	1	1	0	0	1	0		bit address

Bytes: 2

Cycles: 1

DA   A

Function: Decimal adjust accumulator for addition

Description: DA A adjusts the eight-bit value in the accumulator resulting from the earlier addition of two variables (each in packed BCD format), producing two four-bit digits. Any ADD or ADDC instruction may have been used to perform the addition.

If accumulator bits 3-0 are greater than nine (xxxx1010-xxxx1111), or if the AC flag is one, six is added to the accumulator producing the proper BCD digit in the low-order nibble. This internal addition would set the carry flag if a carry-out of the low-order four-bit field propagated through all high-order bits, but it would not clear the carry flag otherwise.

If the carry flag is now set, or if the four high-order bits now exceed nine (1010xxxx-1111xxxx), these high-order bits are incremented by six, producing the proper BCD digit in the high-order nibble. Again, this would set the carry flag if there was a carry-out of the high-order bits, but wouldn't clear the carry. The carry flag thus indicates if the sum of the original two BCD variables is greater than 100, allowing multiple precision decimal addition. OV is not affected.

All of this occurs during the one instruction cycle. Essentially; this instruction performs the decimal conversion by adding 00_H, 06_H, 60_H, or 66_H to the accumulator, depending on initial accumulator and PSW conditions.

Note:

DA A cannot simply convert a hexadecimal number in the accumulator to BCD notation, nor does DA A apply to decimal subtraction.

Example: The accumulator holds the value 56_H (01010110_B) representing the packed BCD digits of the decimal number 56. Register 3 contains the value 67_H (01100111_B) representing the packed BCD digits of the decimal number 67. The carry flag is set. The instruction sequence

ADDC   A,R3
DA   A

will first perform a standard two's-complement binary addition, resulting in the value $0BE_H$ (10111110_B) in the accumulator. The carry and auxiliary carry flags will be cleared.

The decimal adjust instruction will then alter the accumulator to the value 24_H (00100100_B), indicating the packed BCD digits of the decimal number 24, the low-order two digits of the decimal sum of 56, 67, and the carry-in. The carry flag will be set by the decimal adjust instruction, indicating that a decimal overflow occurred. The true sum 56, 67, and 1 is 124.

BCD variables can be incremented or decremented by adding 01_H or 99_H. If the accumulator initially holds 30_H (representing the digits of 30 decimal), then the instruction sequence

ADD   A, #99H
DA   A

will leave the carry set and 29_H in the accumulator, since 30 + 99 = 129. The low-order byte of the sum can be interpreted to mean 30 − 1 = 29.

Operation: DA
contents of accumulator are BCD
if [[(A3-0) > 9] ∨ [(AC) = 1]]
then (A3-0) ← (A3-0) + 6
and
if [[(A7-4) > 9] ∨ [(C) = 1]]
then (A7-4) ← (A7-4) + 6

Encoding:

1 1 0 1	0 1 0 0

Bytes: 1

Cycles: 1

DEC   byte

Function: Decrement

Description: The variable indicated is decremented by 1. An original value of 00_H will underflow to $0FF_H$. No flags are affected. Four operand addressing modes are allowed: accumulator, register, direct, or register-indirect.

Note:

When this instruction is used to modify an output port, the value used as the original port data will be read from the output data latch, not the input pins.

Example: Register 0 contains $7F_H$ (01111111_B). Internal RAM locations $7E_H$ and $7F_H$ contain 00_H and 40_H, respectively. The instruction sequence

DEC   @R0
DEC   R0
DEC   @R0

will leave register 0 set to $7E_H$ and internal RAM locations $7E_H$ and $7F_H$ set to $0FF_H$ and $3F_H$.

DEC   A

Operation: DEC
              $(A) \leftarrow (A) - 1$

Encoding: | 0 0 0 1 | 0 1 0 0 |

Bytes: 1

Cycles: 1

DEC   Rn

Operation: DEC
              $(Rn) \leftarrow (Rn) - 1$

Encoding: | 0 0 0 1 | 1 r r r |

Bytes: 1

Cycles: 1

DEC   direct

Operation: DEC
                (direct) ← (direct) – 1

Encoding: | 0 0 0 1 | 0 1 0 1 | | direct address |

Bytes: 2

Cycles: 1

DEC   @Ri

Operation: DEC
                ((Ri)) ← ((Ri)) – 1

Encoding: | 0 0 0 1 | 0 1 1 i |

Bytes: 1

Cycles: 1

DIV   AB

Function: Divide

Description: DIV AB divides the unsigned eight-bit integer in the accumulator by the unsigned eight-bit integer in register B. The accumulator receives the integer part of the quotient; register B receives the integer remainder. The carry and OV flags will be cleared.

Exception: If B had originally contained 00_H, the values returned in the accumulator and B register will be undefined and the overflow flag will be set. The carry flag is cleared in any case.

Example: The accumulator contains 251 ($0FB_H$ or 11111011_B) and B contains 18 (12_H or 00010010_B). The instruction

DIV   AB

will leave 13 in the accumulator ($0D_H$ or 00001101_B) and the value 17 (11_H or 00010001_B) in B, since 251 = (13x18) + 17. Carry and OV will both be cleared.

Operation: DIV

$$\begin{matrix} (A15\text{-}8) \\ (B7\text{-}0) \end{matrix} \leftarrow (A) / (B)$$

Encoding: | 1 0 0 0 | 0 1 0 0 |

Bytes: 1

Cycles: 4

DJNZ <byte>, <rel-addr>

Function: Decrement and jump if not zero

Description: DJNZ decrements the location indicated by 1, and branches to the address indicated by the second operand if the resulting value is not zero. An original value of 00_H will underflow to $0FF_H$. No flags are affected. The branch destination would be computed by adding the signed relative-displacement value in the last instruction byte to the PC, after incrementing the PC to the first byte of the following instruction.

The location decremented may be a register or directly addressed byte.

Note:

When this instruction is used to modify an output port, the value used as the original port data will be read from the output data latch, not the input pins.

Example: Internal RAM locations 40_H, 50_H, and 60_H contain the values, 01_H, 70_H, and 15_H, respectively. The instruction sequence

DJNZ 40H,LABEL_1
DJNZ 50H,LABEL_2
DJNZ 60H,LABEL_3

will cause a jump to the instruction at label LABEL_2 with the values 00_H, $6F_H$, and 15_H in the three RAM locations. The first jump was not taken because the result was zero.

This instruction provides a simple way of executing a program loop a given number of times, or for adding a moderate time delay (from 2 to 512 machine cycles) with a single instruction. The instruction sequence

```
        MOV     R2, #8
TOGGLE: CPL     P1.7
        DJNZ    R2,TOGGLE
```

will toggle P1.7 eight times, causing four output pulses to appear at bit 7 of output port 1. Each pulse will last three machine cycles; two for DJNZ and one to alter the pin.

DJNZ   Rn,rel

Operation: DJNZ
                $(PC) \leftarrow (PC) + 2$
                $(Rn) \leftarrow (Rn) - 1$
                if $(Rn) > 0$ or $(Rn) < 0$
                then $(PC) \leftarrow (PC) + rel$

Encoding:

1 1 0 1	1 r r r		rel. address

Bytes: 2

Cycles: 2

DJNZ   direct,rel

Operation: DJNZ
                $(PC) \leftarrow (PC) + 2$
                $(direct) \leftarrow (direct) - 1$
                if $(direct) > 0$ or $(direct) < 0$
                then $(PC) \leftarrow (PC) + rel$

Encoding:

1 1 0 1	0 1 0 1		direct address		rel. address

Bytes: 3

Cycles: 2

INC   <byte>

Function: Increment

Description: INC increments the indicated variable by 1. An original value of $0FF_H$ will overflow
                to 00_H. No flags are affected. Three addressing modes are allowed: register, direct,
                or register-indirect.

Note:

When this instruction is used to modify an output port, the value used as the original
port data will be read from the output data latch, not the input pins.

Example: Register 0 contains $7E_H$ (01111110_B). Internal RAM locations $7E_H$ and $7F_H$
                contain $0FF_H$ and 40_H, respectively. The instruction sequence

                INC   @R0
                INC   R0
                INC   @R0

                will leave register 0 set to $7F_H$ and internal RAM locations $7E_H$ and $7F_H$ holding
                (respectively) 00_H and 41_H.

INC   A

Operation: INC
                $(A) \leftarrow (A) + 1$

Encoding: | 0 0 0 0 | 0 1 0 0 |

Bytes: 1

Cycles: 1

INC   Rn

Operation: INC
                $(Rn) \leftarrow (Rn) + 1$

Encoding: | 0 0 0 0 | 1 r r r |

Bytes: 1

Cycles: 1

INC   direct

Operation: INC
               (direct) ← (direct) + 1

Encoding: | 0 0 0 0 | 0 1 0 1 | | direct address |

Bytes: 2

Cycles: 1

INC   @Ri

Operation: INC
               ((Ri)) ← ((Ri)) + 1

Encoding: | 0 0 0 0 | 0 1 1 i |

Bytes: 1

Cycles: 1

INC   DPTR

Function: Increment data pointer

Description: Increment the 16-bit data pointer by 1. A 16-bit increment (modulo 2^{16}) is performed; an overflow of the low-order byte of the data pointer (DPL) from 0FF$_H$ to 00$_H$ will increment the high-order byte (DPH). No flags are affected.

This is the only 16-bit register which can be incremented.

Example: Registers DPH and DPL contain 12$_H$ and 0FE$_H$, respectively. The instruction sequence

INC   DPTR
INC   DPTR
INC   DPTR

will change DPH and DPL to 13$_H$ and 01$_H$.

Operation: INC
(DPTR) ← (DPTR) + 1

Encoding:

1 0 1 0	0 0 1 1

Bytes: 1

Cycles: 2

JB   bit,rel

Function: Jump if bit is set

Description: If the indicated bit is a one, jump to the address indicated; otherwise proceed with the next instruction. The branch destination is computed by adding the signed relative-displacement in the third instruction byte to the PC, after incrementing the PC to the first byte of the next instruction. The bit tested is not modified. No flags are affected.

Example: The data present at input port 1 is 11001010_B. The accumulator holds 56 (01010110_B). The instruction sequence

JB   P1.2,LABEL1
JB   ACC.2,LABEL2

will cause program execution to branch to the instruction at label LABEL2.

Operation: JB
                $(PC) \leftarrow (PC) + 3$
                if (bit) = 1
                then $(PC) \leftarrow (PC) + rel$

Encoding:

0 0 1 0	0 0 0 0		bit address		rel. address

Bytes: 3

Cycles: 2

JBC   bit,rel

Function: Jump if bit is set and clear bit

Description: If the indicated bit is one, branch to the address indicated; otherwise proceed with the next instruction. In either case, clear the designated bit. The branch destination is computed by adding the signed relative displacement in the third instruction byte to the PC, after incrementing the PC to the first byte of the next instruction. No flags are affected.

Note:

When this instruction is used to test an output pin, the value used as the original data will be read from the output data latch, not the input pin.

Example: The accumulator holds 56_H (01010110_B). The instruction sequence

JBC   ACC.3,LABEL1
JBC   ACC.2,LABEL2

will cause program execution to continue at the instruction identified by the label LABEL2, with the accumulator modified to 52_H (01010010_B).

Operation: JBC
                  (PC) ← (PC) + 3
                  if (bit) = 1
                  then (bit) ← 0
                        (PC) ← (PC) + rel

Encoding:

0 0 0 1	0 0 0 0		bit address		rel. address

Bytes: 3

Cycles: 2

JC   rel

Function: Jump if carry is set

Description: If the carry flag is set, branch to the address indicated; otherwise proceed with the next instruction. The branch destination is computed by adding the signed relative-displacement in the second instruction byte to the PC, after incrementing the PC twice. No flags are affected.

Example: The carry flag is cleared. The instruction sequence

                 JC   LABEL1
                 CPL   C
                 JC   LABEL2

                 will set the carry and cause program execution to continue at the instruction identified by the label LABEL2.

Operation: JC
                 (PC) ← (PC) + 2
                 if (C) = 1
                 then (PC) ← (PC) + rel

Encoding: | 0 1 0 0 | 0 0 0 0 | | rel. address |

Bytes: 2

Cycles: 2

JMP   @A + DPTR

Function: Jump indirect

Description: Add the eight-bit unsigned contents of the accumulator with the sixteen-bit data
                 pointer, and load the resulting sum to the program counter. This will be the address
                 for subsequent instruction fetches. Sixteen-bit addition is performed (modulo 2^{16}): a
                 carry-out from the low-order eight bits propagates through the higher-order bits.
                 Neither the accumulator nor the data pointer is altered. No flags are affected.

Example: An even number from 0 to 6 is in the accumulator. The following sequence of
                 instructions will branch to one of four AJMP instructions in a jump table starting at
                 JMP_TBL:

	MOV	DPTR, #JMP_TBL
	JMP	@A + DPTR
JMP_TBL:	AJMP	LABEL0
	AJMP	LABEL1
	AJMP	LABEL2
	AJMP	LABEL3

If the accumulator equals 04_H when starting this sequence, execution will jump to
label LABEL2. Remember that AJMP is a two-byte instruction, so the jump
instructions start at every other address.

Operation: JMP
                 (PC) ← (A) + (DPTR)

Encoding: | 0 1 1 1 | 0 0 1 1 |

Bytes: 1

Cycles: 2

JNB   bit,rel

Function: Jump if bit is not set

Description: If the indicated bit is a zero, branch to the indicated address; otherwise proceed with the next instruction. The branch destination is computed by adding the signed relative-displacement in the third instruction byte to the PC, after incrementing the PC to the first byte of the next instruction. The bit tested is not modified. No flags are affected.

Example: The data present at input port 1 is 11001010_B. The accumulator holds 56_H (01010110_B). The instruction sequence

               JNB   P1.3,LABEL1
               JNB   ACC.3,LABEL2

               will cause program execution to continue at the instruction at label LABEL2.

Operation: JNB
               $(PC) \leftarrow (PC) + 3$
               if $(bit) = 0$
               then $(PC) \leftarrow (PC) + rel.$

Encoding: | 0 0 1 1 | 0 0 0 0 | | bit address | | rel. address |

Bytes: 3
Cycles: 2

JNC   rel

Function: Jump if carry is not set

Description: If the carry flag is a zero, branch to the address indicated; otherwise proceed with
the next instruction. The branch destination is computed by adding the signed
relative-displacement in the second instruction byte to the PC, after incrementing
the PC twice to point to the next instruction. The carry flag is not modified.

Example: The carry flag is set. The instruction sequence

JNC   LABEL1
CPL   C
JNC   LABEL2

will clear the carry and cause program execution to continue at the instruction
identified by the label LABEL2.

Operation: JNC
(PC) ← (PC) + 2
if (C) = 0
then (PC) ← (PC) + rel

Encoding: | 0 1 0 1 | 0 0 0 0 | | rel. address |

Bytes: 2

Cycles: 2

JNZ   rel

Function: Jump if accumulator is not zero

Description: If any bit of the accumulator is a one, branch to the indicated address; otherwise
                proceed with the next instruction. The branch destination is computed by adding the
                signed relative-displacement in the second instruction byte to the PC, after
                incrementing the PC twice. The accumulator is not modified. No flags are affected.

Example: The accumulator originally holds 00_H. The instruction sequence

                JNZ   LABEL1
                INC   A
                JNZ   LABEL2

                will set the accumulator to 01_H and continue at label LABEL2.

Operation: JNZ
                $(PC) \leftarrow (PC) + 2$
                if $(A) \neq 0$
                then $(PC) \leftarrow (PC) + rel.$

Encoding: | 0 1 1 1 | 0 0 0 0 | | rel. address |
                |---|---|

Bytes: 2

Cycles: 2

JZ   rel

Function: Jump if accumulator is zero

Description: If all bits of the accumulator are zero, branch to the address indicated; otherwise
                 proceed with the next instruction. The branch destination is computed by adding the
                 signed relative-displacement in the second instruction byte to the PC, after
                 incrementing the PC twice. The accumulator is not modified. No flags are affected.

Example: The accumulator originally contains 01_H. The instruction sequence

                 JZ   LABEL1
                 DEC   A
                 JZ   LABEL2

                 will change the accumulator to 00_H and cause program execution to continue at the
                 instruction identified by the label LABEL2.

Operation: JZ
                 (PC) ← (PC) + 2
                 if (A) = 0
                 then (PC) ← (PC) + rel

Encoding: | 0 1 1 0 | 0 0 0 0 | | rel. address |

Bytes: 2

Cycles: 2

LCALL   addr16

Function: Long call

Description: LCALL calls a subroutine located at the indicated address. The instruction adds
three to the program counter to generate the address of the next instruction and
then pushes the 16-bit result onto the stack (low byte first), incrementing the stack
pointer by two. The high-order and low-order bytes of the PC are then loaded,
respectively, with the second and third bytes of the LCALL instruction. Program
execution continues with the instruction at this address. The subroutine may
therefore begin anywhere in the full 64 Kbyte program memory address space. No
flags are affected.

Example: Initially the stack pointer equals 07_H. The label "SUBRTN" is assigned to program
memory location 1234_H. After executing the instruction

LCALL   SUBRTN

at location 0123_H, the stack pointer will contain 09_H, internal RAM locations 08_H
and 09_H will contain 26_H and 01_H, and the PC will contain 1234_H.

Operation: LCALL
$(PC) \leftarrow (PC) + 3$
$(SP) \leftarrow (SP) + 1$
$((SP)) \leftarrow (PC7\text{-}0)$
$(SP) \leftarrow (SP) + 1$
$((SP)) \leftarrow (PC15\text{-}8)$
$(PC) \leftarrow addr15\text{-}0$

Encoding:

0 0 0 1	0 0 1 0		addr15 . . addr8		addr7 . . addr0

Bytes: 3

Cycles: 2

LJMP   addr16

Function: Long jump

Description: LJMP causes an unconditional branch to the indicated address, by loading the high-order and low-order bytes of the PC (respectively) with the second and third instruction bytes. The destination may therefore be anywhere in the full 64K program memory address space. No flags are affected.

Example: The label "JMPADR" is assigned to the instruction at program memory location 1234_H. The instruction

LJMP   JMPADR

at location 0123_H will load the program counter with 1234_H.

Operation: LJMP
              (PC) ← addr15-0

Encoding:

0 0 0 0	0 0 1 0		addr15 . . . addr8		addr7 . . . addr0

Bytes: 3

Cycles: 2

MOV **<dest-byte>, <src-byte>**

Function: Move byte variable

Description: The byte variable indicated by the second operand is copied into the location specified by the first operand. The source byte is not affected. No other register or flag is affected.

This is by far the most flexible operation. Fifteen combinations of source and destination addressing modes are allowed.

Example: Internal RAM location 30_H holds 40_H. The value of RAM location 40_H is 10_H. The data present at input port 1 is 11001010B ($0CA_H$).

MOV	R0, #30H	; R0 < = 30H
MOV	A, @R0	; A < = 40H
MOV	R1,A	; R1 < = 40H
MOV	B, @R1	; B < = 10_H
MOV	@R1,P1	; RAM (40H) < = 0CAH
MOV	P2,P1	; P2 < = 0CAH

leaves the value 30_H in register 0, 40_H in both the accumulator and register 1, 10_H in register B, and $0CA_H$ (11001010_B) both in RAM location 40_H and output on port 2.

MOV **A,Rn**

Operation: MOV
(A) ← (Rn)

Encoding:

1 1 1 0	1 r r r

Bytes: 1

Cycles: 1

MOV **A,direct *)**

Operation: MOV
(A) ← (direct)

Encoding:

1 1 1 0	0 1 0 1	direct address

Bytes: 2

Cycles: 1

*) MOV A,ACC is not a valid instruction. The content of the accumulator after the execution of this instruction is undefined.

MOV   A,@Ri

Operation: MOV
                (A) ← ((Ri))

Encoding: | 1 1 1 0 | 0 1 1 i |

Bytes: 1

Cycles: 1

MOV   A, #data

Operation: MOV
                (A) ← #data

Encoding: | 0 1 1 1 | 0 1 0 0 | | immediate data |

Bytes: 2

Cycles: 1

MOV   Rn,A

Operation: MOV
                (Rn) ← (A)

Encoding: | 1 1 1 1 | 1 r r r |

Bytes: 1

Cycles: 1

MOV   Rn,direct

Operation: MOV
                (Rn) ← (direct)

Encoding: | 1 0 1 0 | 1 r r r | | direct address |

Bytes: 2

Cycles: 2

MOV   Rn, #data

Operation: MOV
              (Rn) ← #data

Encoding: | 0 1 1 1 | 1 r r r | | immediate data |

Bytes: 2

Cycles: 1

MOV   direct,A

Operation: MOV
              (direct) ← (A)

Encoding: | 1 1 1 1 | 0 1 0 1 | | direct address |

Bytes: 2

Cycles: 1

MOV   direct,Rn

Operation: MOV
              (direct) ← (Rn)

Encoding: | 1 0 0 0 | 1 r r r | | direct address |

Bytes: 2

Cycles: 2

MOV   direct,direct

Operation: MOV
              (direct) ← (direct)

Encoding: | 1 0 0 0 | 0 1 0 1 | | dir.addr. (src) | | dir.addr. (dest) |

Bytes: 3

Cycles: 2

MOV   direct, @ Ri

Operation: MOV
              (direct) ← ((Ri))

Encoding: | 1 0 0 0 | 0 1 1 i | | direct address |

Bytes: 2

Cycles: 2

MOV   direct, #data

Operation: MOV
              (direct) ← #data

Encoding: | 0 1 1 1 | 0 1 0 1 | | direct address | | immediate data |

Bytes: 3

Cycles: 2

MOV   @ Ri,A

Operation: MOV
              ((Ri)) ← (A)

Encoding: | 1 1 1 1 | 0 1 1 i |

Bytes: 1

Cycles: 1

MOV   @ Ri,direct

Ooeration: MOV
              ((Ri)) ← (direct)

Encoding: | 1 0 1 0 | 0 1 1 i | | direct address |

Bytes: 2

Cycles: 2

MOV   @ Ri,#data

Operation: MOV
              $((Ri)) \leftarrow \#data$

Encoding: | 0 1 1 1 | 0 1 1 i | | immediate data |

Bytes: 2

Cycles: 1

MOV   <dest-bit>, <src-bit>

Function: Move bit data

Description: The Boolean variable indicated by the second operand is copied into the location
specified by the first operand. One of the operands must be the carry flag; the other
may be any directly addressable bit. No other register or flag is affected.

Example: The carry flag is originally set. The data present at input port 3 is 11000101_B. The
data previously written to output port 1 is 35_H (00110101_B).

                MOV   P1.3,C
                MOV   C,P3.3
                MOV   P1.2,C

                will leave the carry cleared and change port 1 to 39_H (00111001_B).

MOV   C,bit

Operation: MOV
                (C) ← (bit)

Encoding:

1 0 1 0	0 0 1 0		bit address

Bytes: 2

Cycles: 1

MOV   bit,C

Operation: MOV
                (bit) ← (C)

Encoding:

1 0 0 1	0 0 1 0		bit address

Bytes: 2

Cycles: 2

MOV   DPTR, #data16

Function: Load data pointer with a 16-bit constant

Description: The data pointer is loaded with the 16-bit constant indicated. The 16 bit constant is
                 loaded into the second and third bytes of the instruction. The second byte (DPH) is
                 the high-order byte, while the third byte (DPL) holds the low-order byte. No flags are
                 affected.

                 This is the only instruction which moves 16 bits of data at once.

Example: The instruction

                 MOV   DPTR, #1234H

                 will load the value 1234_H into the data pointer: DPH will hold 12_H and DPL will hold
                 34_H.

Operation: MOV
                 (DPTR) ← #data15-0
                 DPH □ DPL ← #data15-8 □ #data7-0

Encoding: | 1 0 0 1 | 0 0 0 0 | | immed. data 15 . . . 8 | | immed. data 7 . . . 0 |

Bytes: 3

Cycles: 2

MOVC   A, @A + <base-reg>

Function: Move code byte

Description: The MOVC instructions load the accumulator with a code byte, or constant from program memory. The address of the byte fetched is the sum of the original unsigned eight-bit accumulator contents and the contents of a sixteen-bit base register, which may be either the data pointer or the PC. In the latter case, the PC is incremented to the address of the following instruction before being added to the accumulator; otherwise the base register is not altered. Sixteen-bit addition is performed so a carry-out from the low-order eight bits may propagate through higher-order bits. No flags are affected.

Example: A value between 0 and 3 is in the accumulator. The following instructions will translate the value in the accumulator to one of four values defined by the DB (define byte) directive.

```
REL_PC: INC        A
        MOVC       A, @A + PC
        RET
        DB         66H
        DB         77H
        DB         88H
        DB         99H
```

If the subroutine is called with the accumulator equal to 01_H, it will return with 77_H in the accumulator. The INC A before the MOVC instruction is needed to "get around" the RET instruction above the table. If several bytes of code separated the MOVC from the table, the corresponding number would be added to the accumulator instead.

MOVC   A, @A + DPTR

Operation: MOVC
           $(A) \leftarrow ((A) + (DPTR))$

Encoding:

1 0 0 1	0 0 1 1

Bytes: 1

Cycles: 2

MOVC   A, @A + PC

Operation: MOVC
               (PC) ← (PC) + 1
               (A) ← ((A) + (PC))

Encoding: | 1 0 0 0 | 0 0 1 1 |

Bytes: 1

Cycles: 2

MOVX   <dest-byte>, <src-byte>

Function: Move external

Description: The MOVX instructions transfer data between the accumulator and a byte of external data memory, hence the "X" appended to MOV. There are two types of instructions, differing in whether they provide an eight bit or sixteen-bit indirect address to the external data RAM.

In the first type, the contents of R0 or R1 in the current register bank provide an eight-bit address multiplexed with data on P0. Eight bits are sufficient for external I/O expansion decoding or a relatively small RAM array. For somewhat larger arrays, any output port pins can be used to output higher-order address bits. These pins would be controlled by an output instruction preceding the MOVX.

In the second type of MOVX instructions, the data pointer generates a sixteen-bit address. P2 outputs the high-order eight address bits (the contents of DPH) while P0 multiplexes the low-order eight bits (DPL) with data. The P2 special function register retains its previous contents while the P2 output buffers are emining the contents of DPH. This form is faster and more efficient when accessing very large data arrays (up to 64 Kbyte), since no additional instructions are needed to set up the output ports.

It is possible in some situations to mix the two MOVX types. A large RAM array with its high-order address lines driven by P2 can be addressed via the data pointer, or with code to output high-order address bits to P2 followed by a MOVX instruction using R0 or R1.

Example: An external 256-byte RAM using multiplexed address/data lines is connected to the C500 port 0. Port 3 provides control lines for the external RAM. Ports 1 and 2 are used for normal I/O. Registers 0 and 1 contain 12_H and 34_H. Location 34_H of the external RAM holds the value 56_H. The instruction sequence

MOVX   A, @R1
MOVX   @R0,A

copies the value 56_H into both the accumulator and external RAM location 12_H.

MOVX   A,@Ri

Operation: MOVX
               $(A) \leftarrow ((Ri))$

Encoding: | 1 1 1 0 | 0 0 1 i |

Bytes: 1

Cycles: 2

MOVX   A,@DPTR

Operation: MOVX
               $(A) \leftarrow ((DPTR))$

Encoding: | 1 1 1 0 | 0 0 0 0 |

Bytes: 1

Cycles: 2

MOVX   @Ri,A

Operation: MOVX
               $((Ri)) \leftarrow (A)$

Encoding: | 1 1 1 1 | 0 0 1 i |

Bytes: 1

Cycles: 2

MOVX   @DPTR,A

Operation: MOVX
               $((DPTR))$ (A)

Encoding: | 1 1 1 1 | 0 0 0 0 |

Bytes: 1

Cycles: 2

MUL   AB

Function: Multiply

Description: MUL AB multiplies the unsigned eight-bit integers in the accumulator and register
B. The low-order byte of the sixteen-bit product is left in the accumulator, and the
high-order byte in B. If the product is greater than 255 ($0FF_H$) the overflow flag is
set; otherwise it is cleared. The carry flag is always cleared.

Example: Originally the accumulator holds the value 80 (50_H). Register B holds the value 160
($0A0_H$). The instruction

MUL   AB

will give the product 12,800 (3200_H), so B is changed to 32_H (00110010_B) and the
accumulator is cleared. The overflow flag is set, carry is cleared.

Operation: MUL

$$\begin{matrix}(A7\text{-}0)\\(B15\text{-}8)\end{matrix} \leftarrow (A) \times (B)$$

Encoding: | 1 0 1 0 | 0 1 0 0 |

Bytes: 1

Cycles: 4

NOP

Function:	No operation
Description:	Execution continues at the following instruction. Other than the PC, no registers or flags are affected.
Example:	It is desired to produce a low-going output pulse on bit 7 of port 2 lasting exactly 5 cycles. A simple SETB/CLR sequence would generate a one-cycle pulse, so four additional cycles must be inserted. This may be done (assuming no interrupts are enabled) with the instruction sequence

```
CLR    P2.7
NOP
NOP
NOP
NOP
SETB  P2.7
```

Operation:	NOP
Encoding:	0 0 0 0 0 0 0 0
Bytes:	1
Cycles:	1

ORL **<dest-byte>, <src-byte>**

Function: Logical OR for byte variables

Description: ORL performs the bitwise logical OR operation between the indicated variables, storing the results in the destination byte. No flags are affected (except P, if <dest-byte> = A).

The two operands allow six addressing mode combinations. When the destination is the accumulator, the source can use register, direct, register-indirect, or immediate addressing; when the destination is a direct address, the source can be the accumulator or immediate data.

Note:

When this instruction is used to modify an output port, the value used as the original port data will be read from the output data latch, not the input pins.

Example: If the accumulator holds $0C3_H$ (11000011_B) and R0 holds 55_H (01010101_B) then the instruction

ORL A,R0

will leave the accumulator holding the value $0D7_H$ (11010111_B).

When the destination is a directly addressed byte, the instruction can set combinations of bits in any RAM location or hardware register. The pattern of bits to be set is determined by a mask byte, which may be either a constant data value in the instruction or a variable computed in the accumulator at run-time. The instruction

ORL P1,#00110010$_B$

will set bits 5, 4, and 1 of output port 1.

ORL **A,Rn**

Operation: ORL
$(A) \leftarrow (A) \vee (Rn)$

Encoding: | 0 1 0 0 | 1 r r r |

Bytes: 1

Cycles: 1

ORL   A,direct

Operation: ORL
              $(A) \leftarrow (A) \vee (direct)$

Encoding: | 0 1 0 0 | 0 1 0 1 | | direct address |

Bytes: 2

Cycles: 1

ORL   A,@Ri

Operation: ORL
              $(A) \leftarrow (A) \vee ((Ri))$

Encoding: | 0 1 0 0 | 0 1 1 i |

Bytes: 1

Cycles: 1

ORL   A,#data

Operation: ORL
              $(A) \leftarrow (A) \vee \#data$

Encoding: | 0 1 0 0 | 0 1 0 0 | | immediate data |

Bytes: 2

Cycles: 1

ORL   direct,A

Operation: ORL
              $(direct) \leftarrow (direct) \vee (A)$

Encoding: | 0 1 0 0 | 0 0 1 0 | | direct address |

Bytes: 2

Cycles: 1

ORL   direct, #data

Operation: ORL
                (direct) ← (direct) ∨ #data

Encoding: | 0 1 0 0 | 0 0 1 1 | | direct address | | immediate data |

Bytes: 3

Cycles: 2

ORL   C, <src-bit>

Function: Logical OR for bit variables

Description: Set the carry flag if the Boolean value is a logic 1; leave the carry in its current state
               otherwise. A slash ("/") preceding the operand in the assembly language indicates
               that the logical complement of the addressed bit is used as the source value, but
               the source bit itself is not affected. No other flags are affected.

Example: Set the carry flag if, and only if, P1.0 = 1, ACC.7 = 1, or OV = 0:

MOV	C,P1.0	; Load carry with input pin P1.0
ORL	C,ACC.7	; OR carry with the accumulator bit 7
ORL	C,/OV	; OR carry with the inverse of OV

ORL   C,bit

Operation: ORL
               $(C) \leftarrow (C) \vee (bit)$

Encoding:

0 1 1 1	0 0 1 0		bit address

Bytes: 2

Cycles: 2

ORL   C,/bit

Operation: ORL
               $(C) \leftarrow (C) \vee / (bit)$

Encoding:

1 0 1 0	0 0 0 0		bit address

Bytes: 2

Cycles: 2

POP   direct

Function: Pop from stack

Description: The contents of the internal RAM location addressed by the stack pointer is read, and the stack pointer is decremented by one. The value read is the transfer to the directly addressed byte indicated. No flags are affected.

Example: The stack pointer originally contains the value 32_H, and internal RAM locations 30_H through 32_H contain the values 20_H, 23_H, and 01_H, respectively. The instruction sequence

POP   DPH
POP   DPL

will leave the stack pointer equal to the value 30_H and the data pointer set to 0123_H. At this point the instruction

POP   SP

will leave the stack pointer set to 20_H. Note that in this special case the stack pointer was decremented to $2F_H$ before being loaded with the value popped (20_H).

Operation: POP
(direct) ← ((SP))
(SP) ← (SP) − 1

Encoding: | 1 1 0 1 | 0 0 0 0 | | direct address |

Bytes: 2

Cycles: 2

PUSH   direct

Function: Push onto stack

Description: The stack pointer is incremented by one. The contents of the indicated variable is
                then copied into the internal RAM location addressed by the stack pointer.
                Otherwise no flags are affected.

Example: On entering an interrupt routine the stack pointer contains 09_H. The data pointer
                holds the value 0123_H. The instruction sequence

                PUSH   DPL
                PUSH   DPH

                will leave the stack pointer set to $0B_H$ and store 23_H and 01_H in internal RAM
                locations $0A_H$ and $0B_H$, respectively.

Operation: PUSH
                $(SP) \leftarrow (SP) + 1$
                $((SP)) \leftarrow (direct)$

Encoding: | 1 1 0 0 | 0 0 0 0 | | direct address |

Bytes: 2

Cycles: 2

RET

Function: Return from subroutine

Description: RET pops the high and low-order bytes of the PC successively from the stack, decrementing the stack pointer by two. Program execution continues at the resulting address, generally the instruction immediately following an ACALL or LCALL. No flags are affected.

Example: The stack pointer originally contains the value $0B_H$. Internal RAM locations $0A_H$ and $0B_H$ contain the values 23_H and 01_H, respectively. The instruction

RET

will leave the stack pointer equal to the value 09_H. Program execution will continue at location 0123_H.

Operation: RET
$(PC15\text{-}8) \leftarrow ((SP))$
$(SP) \leftarrow (SP) - 1$
$(PC7\text{-}0) \leftarrow ((SP))$
$(SP) \leftarrow (SP) - 1$

Encoding: | 0 0 1 0 | 0 0 1 0 |

Bytes: 1

Cycles: 2

RETI

Function: Return from interrupt

Description: RETI pops the high and low-order bytes of the PC successively from the stack, and restores the interrupt logic to accept additional interrupts at the same priority level as the one just processed. The stack pointer is left decremented by two. No other registers are affected; the PSW is not automatically restored to its pre-interrupt status. Program execution continues at the resulting address, which is generally the instruction immediately after the point at which the interrupt request was detected. If a lower or same-level interrupt is pending when the RETI instruction is executed, that one instruction will be executed before the pending interrupt is processed.

Example: The stack pointer originally contains the value $0B_H$. An interrupt was detected during the instruction ending at location 0122_H. Internal RAM locations $0A_H$ and $0B_H$ contain the values 23_H and 01_H, respectively. The instruction

              RETI

              will leave the stack pointer equal to 09_H and return program execution to location 0123_H.

Operation: RETI
              $(PC15\text{-}8) \leftarrow ((SP))$
              $(SP) \leftarrow (SP) - 1$
              $(PC7\text{-}0) \leftarrow ((SP))$
              $(SP) \leftarrow (SP) - 1$

Encoding: | 0 0 1 1 | 0 0 1 0 |

Bytes: 1

Cycles: 2

RL   A

Function: Rotate accumulator left

Description: The eight bits in the accumulator are rotated one bit to the left. Bit 7 is rotated into the bit 0 position. No flags are affected.

Example: The accumulator holds the value $0C5_H$ (11000101_B). The instruction

                RL   A

                leaves the accumulator holding the value $8B_H$ (10001011_B) with the carry unaffected.

Operation: RL
                $(An + 1) \leftarrow (An)$ n = 0-6
                $(A0) \leftarrow (A7)$

Encoding: | 0 0 1 0 | 0 0 1 1 |

Bytes: 1

Cycles: 1

RLC   A

Function: Rotate accumulator left through carry flag

Description: The eight bits in the accumulator and the carry flag are together rotated one bit to the left. Bit 7 moves into the carry flag; the original state of the carry flag moves into the bit 0 position. No other flags are affected.

Example: The accumulator holds the value $0C5_H$ (11000101_B), and the carry is zero. The instruction

RLC   A

leaves the accumulator holding the value $8A_H$ (10001010B) with the carry set.

Operation: RLC
$(An + 1) \leftarrow (An)$ n = 0-6
$(A0) \leftarrow (C)$
$(C) \leftarrow (A7)$

Encoding: | 0 0 1 1 | 0 0 1 1 |

Bytes: 1

Cycles: 1

RR   **A**

Function: Rotate accumulator right

Description: The eight bits in the accumulator are rotated one bit to the right. Bit 0 is rotated into the bit 7 position. No flags are affected.

Example: The accumulator holds the value $0C5_H$ (11000101_B). The instruction

     RR   A

     leaves the accumulator holding the value $0E2_H$ (11100010_B) with the carry unaffected.

Operation: RR
$(An) \leftarrow (An + 1)$ n = 0-6
$(A7) \leftarrow (A0)$

Encoding: | 0 0 0 0 | 0 0 1 1 |

Bytes: 1

Cycles: 1

RRC   A

Function: Rotate accumulator right through carry flag

Description: The eight bits in the accumulator and the carry flag are together rotated one bit to the right. Bit 0 moves into the carry flag; the original value of the carry flag moves into the bit 7 position. No other flags are affected.

Example: The accumulator holds the value $0C5_H$ (11000101_B), the carry is zero. The instruction

RRC   A

leaves the accumulator holding the value 62_H (01100010_B) with the carry set.

Operation: RRC
$(An) \leftarrow (An + 1)$ n=0-6
$(A7) \leftarrow (C)$
$(C) \leftarrow (A0)$

Encoding: | 0 0 0 1 | 0 0 1 1 |

Bytes: 1

Cycles: 1

SETB   **<bit>**

Function: Set bit

Description: SETB sets the indicated bit to one. SETB can operate on the carry flag or any directiy addressable bit. No other flags are affected.

Example: The carry flag is cleared. Output port 1 has been written with the value 34_H (00110100_B). The instructions

    SETB   C
    SETB   P1.0

    will leave the carry flag set to 1 and change the data output on port 1 to 35_H (00110101_B).

SETB   **C**

Operation: SETB
    (C) ← 1

Encoding:

1 1 0 1	0 0 1 1

Bytes: 1

Cycles: 1

SETB   **bit**

Operation: SETB
    (bit) ← 1

Encoding:

1 1 0 1	0 0 1 0		bit address

Bytes: 2

Cycles: 1

SJMP   rel

Function: Short jump

Description: Program control branches unconditionally to the address indicated. The branch
               destination is computed by adding the signed displacement in the second
               instruction byte to the PC, after incrementing the PC twice. Therefore, the range of
               destinations allowed is from 128 bytes preceding this instruction to 127 bytes
               following it.

Example: The label "RELADR" is assigned to an instruction at program memory location
               0123_H. The instruction

               SJMP   RELADR

               will assemble into location 0100_H. After the instruction is executed, the PC will
               contain the value 0123_H.

               Note:

               Under the above conditions the instruction following SJMP will be at 102_H.
               Therefore, the displacement byte of the instruction will be the relative offset (0123_H-
               0102_H) = 21_H. In other words, an SJMP with a displacement of $0FE_H$ would be a
               one-instruction infinite loop.

Operation: SJMP
               $(PC) \leftarrow (PC) + 2$
               $(PC) \leftarrow (PC) + rel$

Encoding: | 1 0 0 0 | 0 0 0 0 | | rel. address |

Bytes: 2

Cycles: 2

SUBB   A, \<src-byte>

Function: Subtract with borrow

Description: SUBB subtracts the indicated variable and the carry flag together from the accumulator, leaving the result in the accumulator. SUBB sets the carry (borrow) flag if a borrow is needed for bit 7, and clears C otherwise. (If C was set before executing a SUBB instruction, this indicates that a borrow was needed for the previous step in a multiple precision subtraction, so the carry is subtracted from the accumulator along with the source operand). AC is set if a borrow is needed for bit 3, and cleared otherwise. OV is set if a borrow is needed into bit 6 but not into bit 7, or into bit 7 but not bit 6.

When subtracting signed integers OV indicates a negative number produced when a negative value is subtracted from a positive value, or a positive result when a positive number is subtracted from a negative number.

The source operand allows four addressing modes: register, direct, register-indirect, or immediate.

Example: The accumulator holds $0C9_H$ (11001001_B), register 2 holds 54_H (01010100_B), and the carry flag is set. The instruction

SUBB   A,R2

will leave the value 74_H (01110100_B) in the accumulator, with the carry flag and AC cleared but OV set.

Notice that $0C9_H$ minus 54_H is 75_H. The difference between this and the above result is due to the (borrow) flag being set before the operation. If the state of the carry is not known before starting a single or multiple-precision subtraction, it should be explicitly cleared by a CLR C instruction.

SUBB   A,Rn

Operation: SUBB
$$(A) \leftarrow (A) - (C) - (Rn)$$

Encoding:

1 0 0 1	1 r r r

Bytes: 1

Cycles: 1

SUBB   A,direct

Operation: SUBB
               (A) ← (A) − (C) − (direct)

Encoding: | 1 0 0 1 | 0 1 0 1 | | direct address |

Bytes: 2

Cycles: 1

SUBB   A, @ Ri

Operation: SUBB
               (A) ← (A) − (C) − ((Ri))

Encoding: | 1 0 0 1 | 0 1 1 i |

Bytes: 1

Cycles: 1

SUBB   A, #data

Operation: SUBB
               (A) ← (A) − (C) − #data

Encoding: | 1 0 0 1 | 0 1 0 0 | | immediate data |

Bytes: 2

Cycles: 1

SWAP   A

Function: Swap nibbles within the accumulator

Description: SWAP A interchanges the low and high-order nibbles (four-bit fields) of the accumulator (bits 3-0 and bits 7-4). The operation can also be thought of as a four-bit rotate instruction. No flags are affected.

Example: The accumulator holds the value $0C5_H$ (11000101_B). The instruction

SWAP   A

leaves the accumulator holding the value $5C_H$ (01011100_B).

Operation: SWAP
(A3-0) $\leftrightarrows$ (A7-4), (A7-4) $\leftarrow$ (A3-0)

Encoding: | 1 1 0 0 | 0 1 0 0 |

Bytes: 1

Cycles: 1

XCH   A, \<byte\>

Function: Exchange accumulator with byte variable

Description: XCH loads the accumulator with the contents of the indicated variable, at the same time writing the original accumulator contents to the indicated variable. The source/ destination operand can use register, direct, or register-indirect addressing.

Example: R0 contains the address 20_H. The accumulator holds the value $3F_H$ (00111111_B). Internal RAM location 20_H holds the value 75_H (01110101_B). The instruction

XCH   A, @R0

will leave RAM location 20_H holding the value $3F_H$ (00111111_B) and 75_H (01110101_B) in the accumulator.

XCH   A,Rn

Operation: XCH
               $(A) \leftrightarrows (Rn)$

Encoding:

1 1 0 0	1 r r r

Bytes: 1

Cycles: 1

XCH   A,direct

Operation: XCH
               $(A) \leftrightarrows (direct)$

Encoding:

| 1 1 0 0 | 0 1 0 1 | | direct address |
|---------|---------| |----------------|

Bytes: 2

Cycles: 1

XCH   A, @ Ri

Operation: XCH
                     (A) $\leftrightarrows$ ((Ri))

Encoding: | 1 1 0 0 | 0 1 1 i |

Bytes: 1

Cycles: 1

XCHD   A,@Ri

Function: Exchange digit

Description: XCHD exchanges the low-order nibble of the accumulator (bits 3-0, generally
representing a hexadecimal or BCD digit), with that of the internal RAM location
indirectly addressed by the specified register. The high-order nibbles (bits 7-4) of
each register are not affected. No flags are affected.

Example: R0 contains the address 20_H. The accumulator holds the value 36_H (00110110_B).
Internal RAM location 20_H holds the value 75_H (01110101_B). The instruction

XCHD   A, @ R0

will leave RAM location 20_H holding the value 76_H (01110110_B) and 35_H
(00110101_B) in the accumulator.

Operation: XCHD
$(A3\text{-}0) \leftrightarrows ((Ri)3\text{-}0)$

Encoding:

1 1 0 1	0 1 1 i

Bytes: 1

Cycles: 1

XRL   <dest-byte>, <src-byte>

Function: Logical Exclusive OR for byte variables

Description: XRL performs the bitwise logical Exclusive OR operation between the indicated variables, storing the results in the destination. No flags are affected (except P, if <dest-byte> = A).

The two operands allow six addressing mode combinations. When the destination is the accumulator, the source can use register, direct, register-indirect, or immediate addressing; when the destination is a direct address, the source can be accumulator or immediate data.

Note:

When this instruction is used to modify an output port, the value used as the original port data will be read from the output data latch, not the input pins.

Example: If the accumulator holds $0C3_H$ (11000011_B) and register 0 holds $0AA_H$ (10101010B) then the instruction

XRL   A,R0

will leave the accumulator holding the value 69_H (01101001_B).

When the destination is a directly addressed byte, this instruction can complement combinations of bits in any RAM location or hardware register. The pattern of bits to be complemented is then determined by a mask byte, either a constant contained in the instruction or a variable computed in the accumulator at run-time. The instruction

XRL   P1,#00110001B

will complement bits 5, 4, and 0 of output port 1.

XRL   A,Rn

Operation: XRL2
                 $(A) \leftarrow (A) \veebar (Rn)$

Encoding:

0 1 1 0	1 r r r

Bytes: 1

Cycles: 1

XRL   A,direct

Operation: XRL
              $(A) \leftarrow (A) \veebar (\text{direct})$

Encoding: | 0 1 1 0 | 0 1 0 1 | | direct address |

Bytes: 2

Cycles: 1

XRL   A, @ Ri

Operation: XRL
              $(A) \leftarrow (A) \veebar ((\text{Ri}))$

Encoding: | 0 1 1 0 | 0 1 1 i |

Bytes: 1

Cycles: 1

XRL   A, #data

Operation: XRL
              $(A) \leftarrow (A) \veebar \#\text{data}$

Encoding: | 0 1 1 0 | 0 1 0 0 | | immediate data |

Bytes: 2

Cycles: 1

XRL   direct,A

Operation: XRL
              $(\text{direct}) \leftarrow (\text{direct}) \veebar (A)$

Encoding: | 0 1 1 0 | 0 0 1 0 | | direct address |

Bytes: 2

Cycles: 1

XRL   direct, #data

Operation: XRL
             (direct) ← (direct) ⩡ #data

Encoding: | 0 1 1 0 | 0 0 1 1 | | direct address | | immediate data |

Bytes: 3
Cycles: 2

4.4 Instruction Set Summary Tables

The following two tables give a survey about the instruction set of the C500 family microcontrollers. In **table 4-3** the instructions are ordered in functional groups. In **table 4-4** the instructions are ordered in the hexadecimal order of their opcode.

4.4.1 Functional Groups of Instructions

**Table 4-3 :
Instruction Set Summary**

Mnemonic		Description	Byte	Cycle
Arithmetic Operations				
ADD	A,Rn	Add register to accumulator	1	1
ADD	A,direct	Add direct byte to accumulator	2	1
ADD	A @Ri	Add indirect RAM to accumulator	1	1
ADD	A,#data	Add immediate data to accumulator	2	1
ADDC	A,Rn	Add register to accumulator with carry flag	1	1
ADDC	A,direct	Add direct byte to A with carry flag	2	1
ADDC	A,@Ri	Add indirect RAM to A with carry flag	1	1
ADDC	A, #data	Add immediate data to A with carry flag	2	1
SUBB	A,Rn	Subtract register from A with borrow	1	1
SUBB	A,direct	Subtract direct byte from A with borrow	2	1
SUBB	A,@Ri	Subtract indirect RAM from A with borrow	1	1
SUBB	A,#data	Subtract immediate data from A with borrow	2	1
INC	A	Increment accumulator	1	1
INC	Rn	Increment register	1	1
INC	direct	Increment direct byte	2	1
INC	@Ri	Increment indirect RAM	1	1
DEC	A	Decrement accumulator	1	1
DEC	Rn	Decrement register	1	1
DEC	direct	Decrement direct byte	2	1
DEC	@Ri	Decrement indirect RAM	1	1
INC	DPTR	Increment data pointer	1	2
MUL	AB	Multiply A and B	1	4
DIV	AB	Divide A by B	1	4
DA	A	Decimal adjust accumulator	1	1

SIEMENS

**Table 4-3 :
Instruction Set Summary   (cont'd)**

Mnemonic		Description	Byte	Cycle
Logic Operations				
ANL	A,Rn	AND register to accumulator	1	1
ANL	A,direct	AND direct byte to accumulator	2	1
ANL	A,@Ri	AND indirect RAM to accumulator	1	1
ANL	A,#data	AND immediate data to accumulator	2	1
ANL	direct,A	AND accumulator to direct byte	2	1
ANL	direct,#data	AND immediate data to direct byte	3	2
ORL	A,Rn	OR register to accumulator	1	1
ORL	A,direct	OR direct byte to accumulator	2	1
ORL	A,@Ri	OR indirect RAM to accumulator	1	1
ORL	A,#data	OR immediate data to accumulator	2	1
ORL	direct,A	OR accumulator to direct byte	2	1
ORL	direct,#data	OR immediate data to direct byte	3	2
XRL	A,Rn	Exclusive OR register to accumulator	1	1
XRL	A direct	Exclusive OR direct byte to accumulator	2	1
XRL	A,@Ri	Exclusive OR indirect RAM to accumulator	1	1
XRL	A,#data	Exclusive OR immediate data to accumulator	2	1
XRL	direct,A	Exclusive OR accumulator to direct byte	2	1
XRL	direct,#data	Exclusive OR immediate data to direct byte	3	2
CLR	A	Clear accumulator	1	1
CPL	A	Complement accumulator	1	1
RL	A	Rotate accumulator left	1	1
RLC	A	Rotate accumulator left through carry	1	1
RR	A	Rotate accumulator right	1	1
RRC	A	Rotate accumulator right through carry	1	1
SWAP	A	Swap nibbles within the accumulator	1	1

**Table 4-3 :
Instruction Set Summary   (cont'd)**

Mnemonic	Description	Byte	Cycle
Data Transfer			
MOV A,Rn	Move register to accumulator	1	1
MOV A,direct	Move direct byte to accumulator	2	1
MOV A,@Ri	Move indirect RAM to accumulator	1	1
MOV A,#data	Move immediate data to accumulator	2	1
MOV Rn,A	Move accumulator to register	1	1
MOV Rn,direct	Move direct byte to register	2	2
MOV Rn,#data	Move immediate data to register	2	1
MOV direct,A	Move accumulator to direct byte	2	1
MOV direct,Rn	Move register to direct byte	2	2
MOV direct,direct	Move direct byte to direct byte	3	2
MOV direct,@Ri	Move indirect RAM to direct byte	2	2
MOV direct,#data	Move immediate data to direct byte	3	2
MOV @Ri,A	Move accumulator to indirect RAM	1	1
MOV @Ri,direct	Move direct byte to indirect RAM	2	2
MOV @Ri, #data	Move immediate data to indirect RAM	2	1
MOV DPTR, #data16	Load data pointer with a 16-bit constant	3	2
MOVC A,@A + DPTR	Move code byte relative to DPTR to accumulator	1	2
MOVC A,@A + PC	Move code byte relative to PC to accumulator	1	2
MOVX A,@Ri	Move external RAM (8-bit addr.) to A	1	2
MOVX A,@DPTR	Move external RAM (16-bit addr.) to A	1	2
MOVX @Ri,A	Move A to external RAM (8-bit addr.)	1	2
MOVX @DPTR,A	Move A to external RAM (16-bit addr.)	1	2
PUSH direct	Push direct byte onto stack	2	2
POP direct	Pop direct byte from stack	2	2
XCH A,Rn	Exchange register with accumulator	1	1
XCH A,direct	Exchange direct byte with accumulator	2	1
XCH A,@Ri	Exchange indirect RAM with accumulator	1	1
XCHD A,@Ri	Exchange low-order nibble indir. RAM with A	1	1

1)   MOV A,ACC is not a valid instruction

**Table 4-3 :
Instruction Set Summary   (cont'd)**

Mnemonic		Description	Byte	Cycle
Boolean Variable Manipulation				
CLR	C	Clear carry flag	1	1
CLR	bit	Clear direct bit	2	1
SETB	C	Set carry flag	1	1
SETB	bit	Set direct bit	2	1
CPL	C	Complement carry flag	1	1
CPL	bit	Complement direct bit	2	1
ANL	C,bit	AND direct bit to carry flag	2	2
ANL	C,/bit	AND complement of direct bit to carry	2	2
ORL	C,bit	OR direct bit to carry flag	2	2
ORL	C,/bit	OR complement of direct bit to carry	2	2
MOV	C,bit	Move direct bit to carry flag	2	1
MOV	bit,C	Move carry flag to direct bit	2	2

Program and Machine Control

Mnemonic		Description	Byte	Cycle
ACALL	addr11	Absolute subroutine call	2	2
LCALL	addr16	Long subroutine call	3	2
RET		Return from subroutine	1	2
RETI		Return from interrupt	1	2
AJMP	addr11	Absolute jump	2	2
LJMP	addr16	Long iump	3	2
SJMP	rel	Short jump (relative addr.)	2	2
JMP	@A + DPTR	Jump indirect relative to the DPTR	1	2
JZ	rel	Jump if accumulator is zero	2	2
JNZ	rel	Jump if accumulator is not zero	2	2
JC	rel	Jump if carry flag is set	2	2
JNC	rel	Jump if carry flag is not set	2	2
JB	bit,rel	Jump if direct bit is set	3	2
JNB	bit,rel	Jump if direct bit is not set	3	2
JBC	bit,rel	Jump if direct bit is set and clear bit	3	2
CJNE	A,direct,rel	Compare direct byte to A and jump if not equal	3	2

**Table 4-3 :
Instruction Set Summary   (cont'd)**

Mnemonic	Description	Byte	Cycle

Program and Machine Control (cont'd)

Mnemonic	Description	Byte	Cycle
CJNE A,#data,rel	Compare immediate to A and jump if not equal	3	2
CJNE Rn,#data rel	Compare immed. to reg. and jump if not equal	3	2
CJNE @Ri,#data,rel	Compare immed. to ind. and jump if not equal	3	2
DJNZ Rn,rel	Decrement register and jump if not zero	2	2
DJNZ direct,rel	Decrement direct byte and jump if not zero	3	2
NOP	No operation	1	1

4.4.2 Hexadecimal Ordered Instructions

Table 4-4 :
Instruction List in Hexadecimal Order

Op-Code	Mnemonic		Op-Code	Mnemonic		Op-Code	Mnemonic	
00_H	NOP		20_H	JB	bit.rel	40_H	JC	rel
01_H	AJMP	addr11	21_H	AJMP	addr11	41_H	AJMP	addr11
02_H	LJMP	addr16	22_H	RET		42_H	ORL	direct,A
03_H	RR	A	23_H	RL	A	43_H	ORL	direct,#data
04_H	INC	A	24_H	ADD	A,#data	44_H	ORL	A,#data
05_H	INC	direct	25_H	ADD	A,direct	45_H	ORL	A,direct
06_H	INC	@R0	26_H	ADD	A,@R0	46_H	ORL	A,@R0
07_H	INC	@R1	27_H	ADD	A,@R1	47_H	ORL	A,@R1
08_H	INC	R0	28_H	ADD	A,R0	48_H	ORL	A,R0
09_H	INC	R1	29_H	ADD	A,R1	49_H	ORL	A,R1
$0A_H$	INC	R2	$2A_H$	ADD	A,R2	$4A_H$	ORL	A,R2
$0B_H$	INC	R3	$2B_H$	ADD	A,R3	$4B_H$	ORL	A,R3
$0C_H$	INC	R4	$2C_H$	ADD	A,R4	$4C_H$	ORL	A,R4
$0D_H$	INC	R5	$2D_H$	ADD	A,R5	$4D_H$	ORL	A,R5
$0E_H$	INC	R6	$2E_H$	ADD	A,R6	$4E_H$	ORL	A,R6
$0F_H$	INC	R7	$2F_H$	ADD	A,R7	$4F_H$	ORL	A,R7
10_H	JBC	bit,rel	30_H	JNB	bit.rel	50_H	JNC	rel
11_H	ACALL	addr11	31_H	ACALL	addr11	51_H	ACALL	addr11
12_H	LCALL	addr16	32_H	RETI		52_H	ANL	direct,A
13_H	RRC	A	33_H	RLC	A	53_H	ANL	direct,#data
14_H	DEC	A	34_H	ADDC	A,#data	54_H	ANL	A,#data
15_H	DEC	direct	35_H	ADDC	A,direct	55_H	ANL	A,direct
16_H	DEC	@R0	36_H	ADDC	A,@R0	56_H	ANL	A,@R0
17_H	DEC	@R1	37_H	ADDC	A,@R1	57_H	ANL	A,@R1
18_H	DEC	R0	38_H	ADDC	A,R0	58_H	ANL	A,R0
19_H	DEC	R1	39_H	ADDC	A,R1	59_H	ANL	A,R1
$1A_H$	DEC	R2	$3A_H$	ADDC	A,R2	$5A_H$	ANL	A,R2
$1B_H$	DEC	R3	$3B_H$	ADDC	A,R3	$5B_H$	ANL	A,R3
$1C_H$	DEC	R4	$3C_H$	ADDC	A,R4	$5C_H$	ANL	A,R4
$1D_H$	DEC	R5	$3D_H$	ADDC	A,R5	$5D_H$	ANL	A,R5
$1E_H$	DEC	R6	$3E_H$	ADDC	A,R6	$5E_H$	ANL	A,R6
$1F_H$	DEC	R7	$3F_H$	ADDC	A,R7	$5F_H$	ANL	A,R7

**Table 4-4 :
Instruction List in Hexadecimal Order   (cont'd)**

Op-Code	Mnemonic		Op-Code	Mnemonic		Op-Code	Mnemonic	
60H	JZ	rel	80H	SJMP	rel	A0H	ORL	C,/bit
61H	AJMP	addr11	81H	AJMP	addr11	A1H	AJMP	addr11
62H	XRL	direct,A	82H	ANL	C,bit	A2H	MOV	C,bit
63H	XRL	direct,#data	83H	MOVC	A,@A+PC	A3H	INC	DPTR
64H	XRL	A,#data	84H	DIV	AB	A4H	MUL	AB
65H	XRL	A,direct	85H	MOV	direct,direct	A5H	-	
66H	XRL	A,@R0	86H	MOV	direct,@R0	A6H	MOV	@R0,direct
67H	XRL	A,@R1	87H	MOV	direct,@R1	A7H	MOV	@R1,direct
68H	XRL	A,R0	88H	MOV	direct,R0	A8H	MOV	R0,direct
69H	XRL	A,R1	89H	MOV	direct,R1	A9H	MOV	R1,direct
6AH	XRL	A,R2	8AH	MOV	direct,R2	AAH	MOV	R2,direct
6BH	XRL	A,R3	8BH	MOV	direct,R3	ABH	MOV	R3,direct
6CH	XRL	A,R4	8CH	MOV	direct,R4	ACH	MOV	R4,direct
6DH	XRL	A,R5	8DH	MOV	direct,R5	ADH	MOV	R5,direct
6EH	XRL	A,R6	8EH	MOV	direct,R6	AEH	MOV	R6,direct
6FH	XRL	A,R7	8FH	MOV	direct,R7	AFH	MOV	R7,direct
70H	JNZ	rel	90H	MOV	DPTR,#data16	B0H	ANL	C,/bit
71H	ACALL	addr11	91H	ACALL	addr11	B1H	ACALL	addr11
72H	ORL	C,direct	92H	MOV	bit,C	B2H	CPL	bit
73H	JMP	@A+DPTR	93H	MOVC	A,@A+DPTR	B3H	CPL	C
74H	MOV	A,#data	94H	SUBB	A,#data	B4H	CJNE	A,#data,rel
75H	MOV	direct,#data	95H	SUBB	A,direct	B5H	CJNE	A,direct,rel
76H	MOV	@R0,#data	96H	SUBB	A,@R0	B6H	CJNE	@R0,#data,rel
77H	MOV	@R1,#data	97H	SUBB	A,@R1	B7H	CJNE	@R1,#data,rel
78H	MOV	R0.#data	98H	SUBB	A,R0	B8H	CJNE	R0,#data,rel
79H	MOV	R1.#data	99H	SUBB	A,R1	B9H	CJNE	R1,#data,rel
7AH	MOV	R2.#data	9AH	SUBB	A,R2	BAH	CJNE	R2,#data,rel
7BH	MOV	R3.#data	9BH	SUBB	A,R3	BBH	CJNE	R3,#data,rel
7CH	MOV	R4.#data	9CH	SUBB	A,R4	BCH	CJNE	R4,#data,rel
7DH	MOV	R5.#data	9DH	SUBB	A,R5	BDH	CJNE	R5,#data,rel
7EH	MOV	R6.#data	9EH	SUBB	A,R6	BEH	CJNE	R6,#data,rel
7FH	MOV	R7.#data	9FH	SUBB	A,R7	BFH	CJNE	R7,#data,rel

SIEMENS

Table 4-4 :
Instruction List in Hexadecimal Order   (cont'd)

Op-Code	Mnemonic		Op-Code	Mnemonic	
C0$_H$	PUSH	direct	E0$_H$	MOVX	A,@DPTR
C1$_H$	AJMP	addr11	E1$_H$	AJMP	addr11
C2$_H$	CLR	bit	E2$_H$	MOVX	A,@R0
C3$_H$	CLR	C	E3$_H$	MOVX	A,@R1
C4$_H$	SWAP	A	E4$_H$	CLR	A
C5$_H$	XCH	A,direct	E5$_H$	MOV	A,direct
C6$_H$	XCH	A,@R0	E6$_H$	MOV	A,@R0
C7$_H$	XCH	A,@R1	E7$_H$	MOV	A,@R1
C8$_H$	XCH	A,R0	E8$_H$	MOV	A,R0
C9$_H$	XCH	A,R1	E9$_H$	MOV	A,R1
CA$_H$	XCH	A,R2	EA$_H$	MOV	A,R2
CB$_H$	XCH	A,R3	EB$_H$	MOV	A,R3
CC$_H$	XCH	A,R4	EC$_H$	MOV	A,R4
CD$_H$	XCH	A,R5	ED$_H$	MOV	A,R5
CE$_H$	XCH	A,R6	EE$_H$	MOV	A,R6
CF$_H$	XCH	A,R7	EF$_H$	MOV	A,R7
D0$_H$	POP	direct	F0$_H$	MOVX	@DPTR,A
D1$_H$	ACALL	addr11	F1$_H$	ACALL	addr11
D2$_H$	SETB	bit	F2$_H$	MOVX	@R0,A
D3$_H$	SETB	C	F3$_H$	MOVX	@R1,A
D4$_H$	DA	A	F4$_H$	CPL	A
D5$_H$	DJNZ	direct,rel	F5$_H$	MOV	direct,A
D6$_H$	XCHD	A,@R0	F6$_H$	MOV	@R0,A
D7$_H$	XCHD	A,@R1	F7$_H$	MOV	@R1,A
D8$_H$	DJNZ	R0,rel	F8$_H$	MOV	R0,A
D9$_H$	DJNZ	R1,rel	F9$_H$	MOV	R1,A
DA$_H$	DJNZ	R2,rel	FA$_H$	MOV	R2,A
DB$_H$	DJNZ	R3,rel	FB$_H$	MOV	R3,A
DC$_H$	DJNZ	R4,rel	FC$_H$	MOV	R4,A
DD$_H$	DJNZ	R5,rel	FD$_H$	MOV	R5,A
DE$_H$	DJNZ	R6,rel	FE$_H$	MOV	R6,A
DF$_H$	DJNZ	R7,rel	FF$_H$	MOV	R7,A

5 Package Information

This chapter shows typical package outlines of the packages which are actually used for the microcontrollers of the C500 family. The appropriate data sheet should always be regarded when the package of a specific C500 microcontroller has to be referenced.

5.1 P-DIP Package

P-DIP-40-3
(Plastic Dual In-line Package)

1) Does not include plastic or metal protrusion of 0.25 max. per side

GPD05883

SMD = Surface Mounted Device   Dimensions in mm
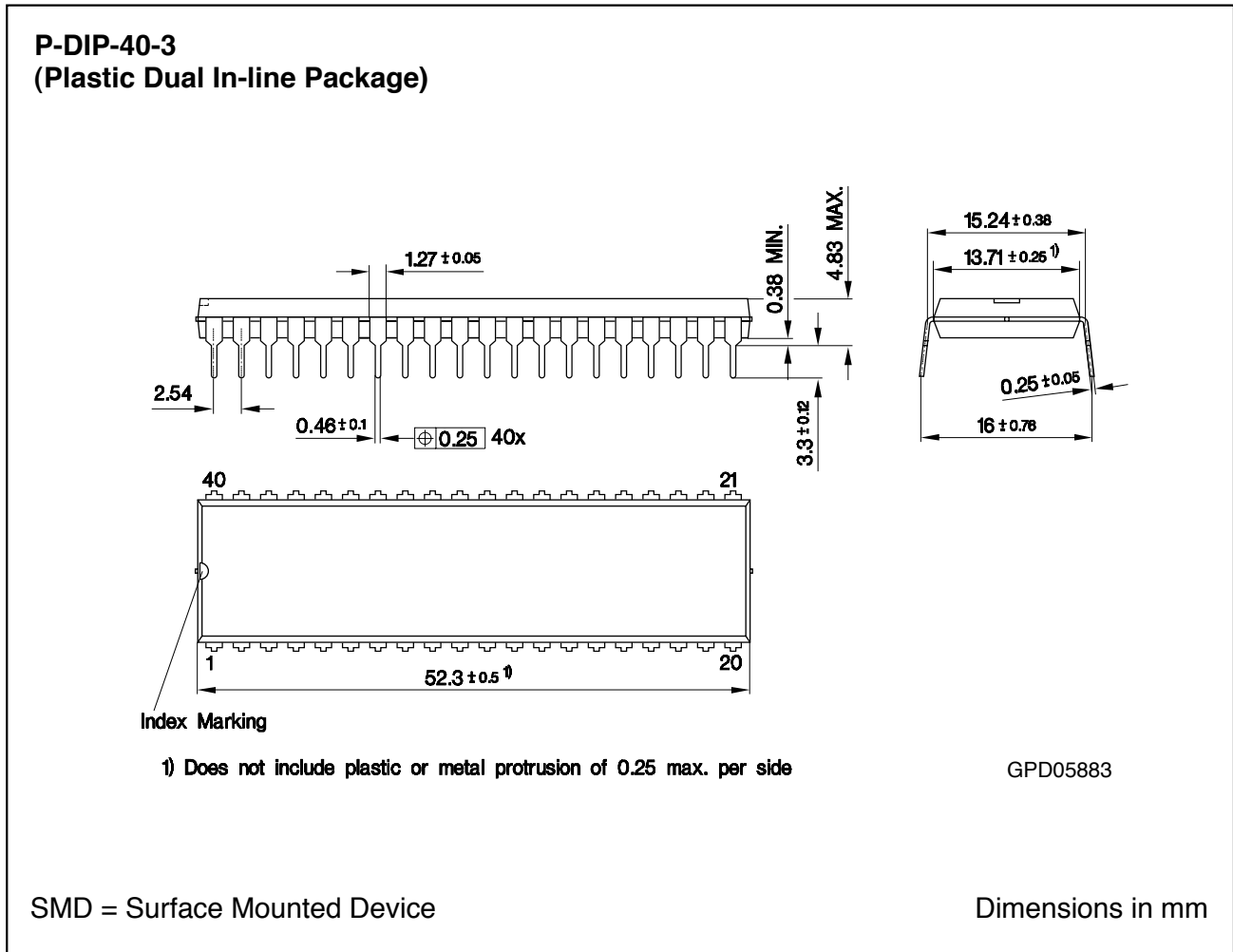
Figure 5-1
P-DIP-40-3 Package Outlines

5.2 PLCC Packages

P-LCC-44-2 (SMD)
(Plastic Leaded Chip Carrier Package)

0.74 ± 0.07

0.5 MIN.

3.55 ± 0.2

4.38 ± 0.19

0.25

1.2 ± 0.13 x 45°

1.27

C

0.43 ± 0.1

⊕ 0.18 A-B-D C 44x

⬜ 0.1

12.7

15.5 ± 0.5

17.53 ± 0.13

≡ 0.38 Ⓜ A-B D 22x

D

A

B

16.6 ± 0.07 1)

44 1

Index Marking

0.5 MAX. x45°
3x

1.1 ± 0.035 x 45°

16.6 ± 0.07 1)

⊥ 0.05 A-B-D

17.53 ± 0.13

1) Does not include plastic or metal protrusion of 0.15 max. per side

GPL05102

SMD = Surface Mounted Device   Dimensions in mm
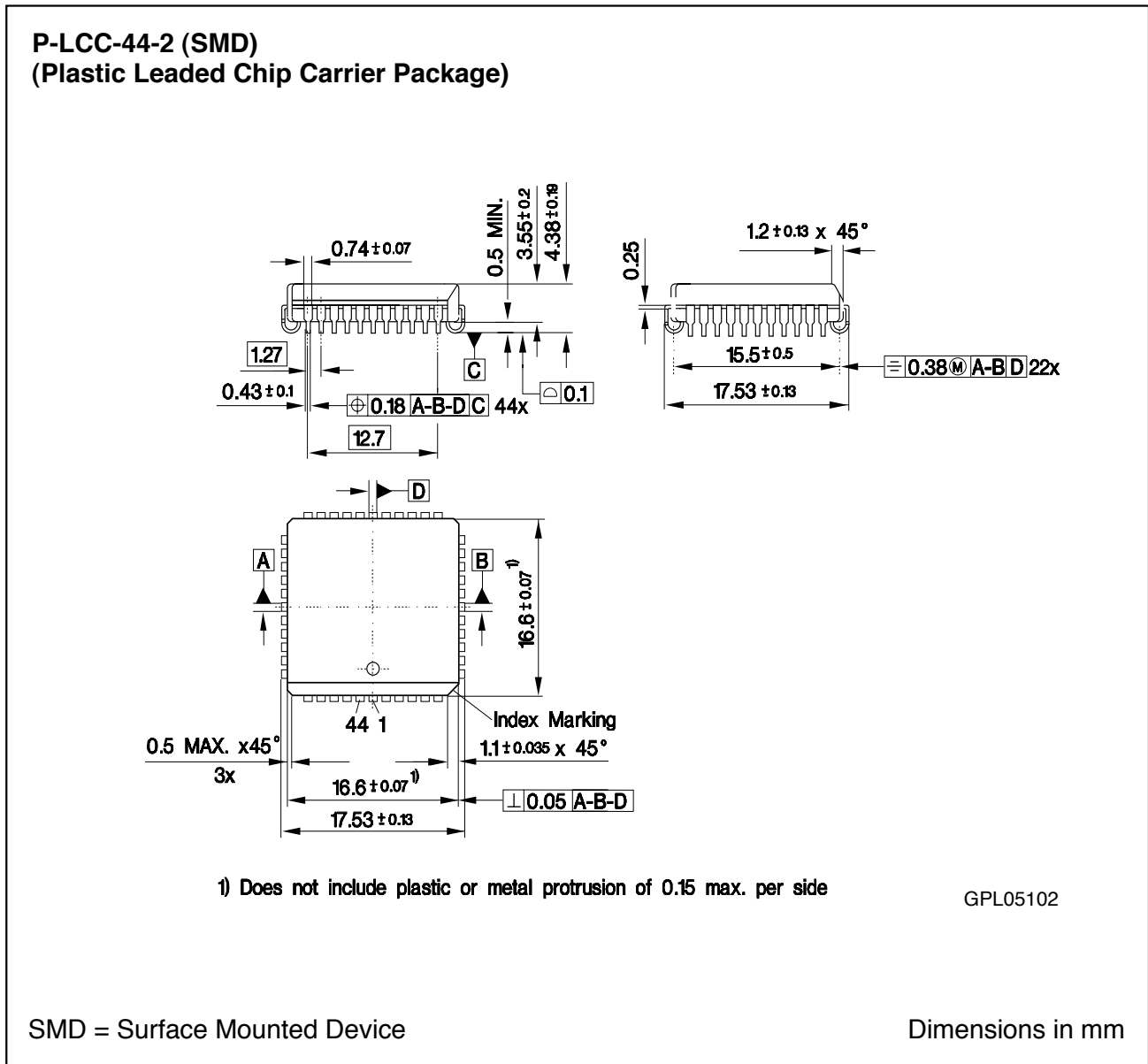
Figure 5-2
P-LCC-44-2 Package Outlines

P-LCC-68-4 (SMD)
(Plastic Leaded Chip Carrier Package)

0.5 min

3.5 ±0.2

5.08 max

1.27

0.81 max

0.43 ±0.1

⊕ 0.18 Ⓜ A-B D 68x

20.32

◻ 0.1

1.2 x 45°

0.2

23.3 ±0.3

24.21 ±0.07 [1]

25.28 -0.26

▤ 0.38 Ⓜ A-B D 34x

D

A

B

0.5 x 45°

3 x

68   1

1.1 x 45°

Index Marking

24.21 ±0.07 [1]

25.28 -0.26

1) Does not include plastic or metal protrusions of 0.15 max per side

GPL5099 GPL05099

SMD = Surface Mounted Device

Dimensions in mm
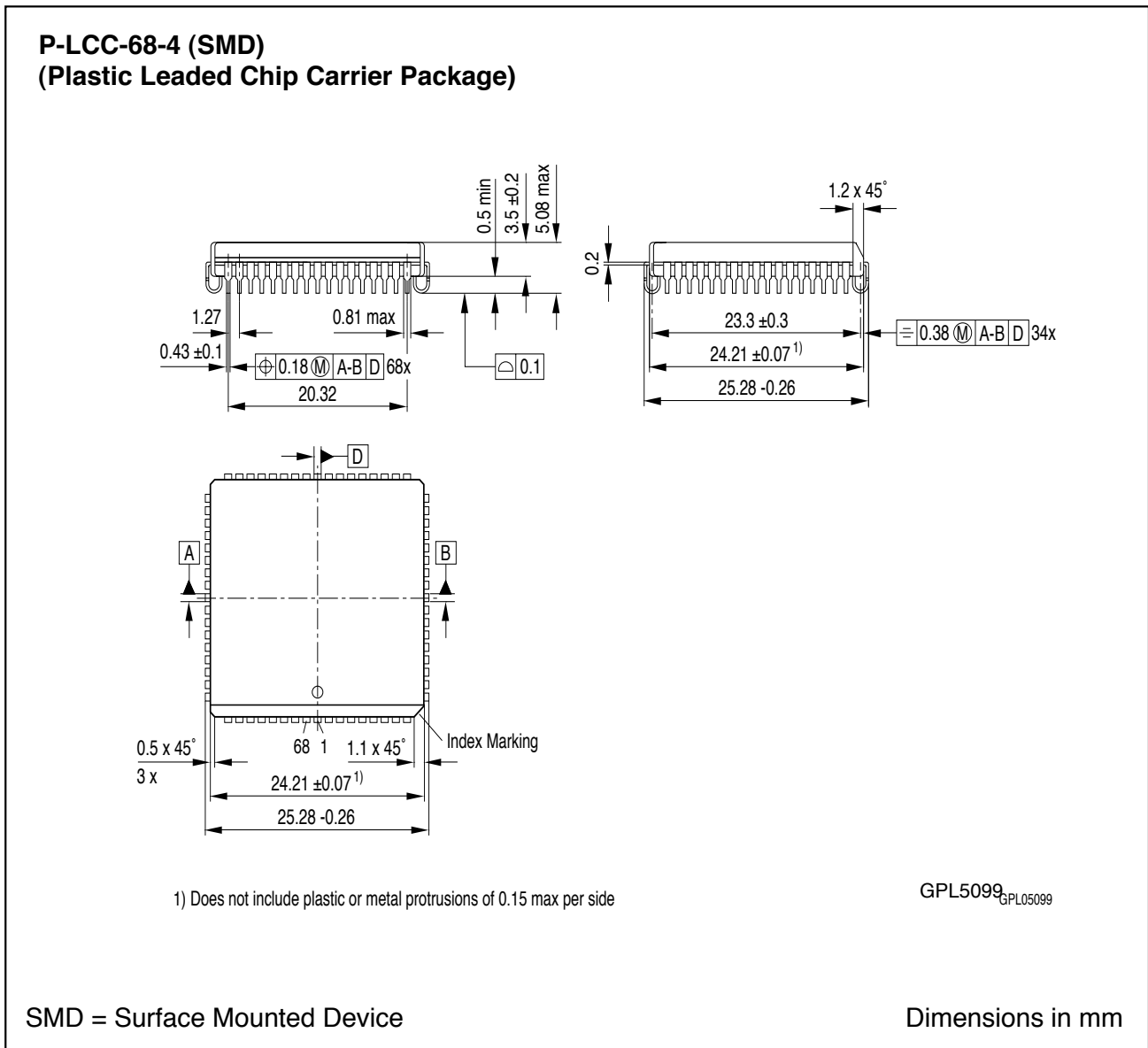
Figure 5-3
P-LCC-68-4 Package Outline

**P-LCC-84-2 (SMD)
(Plastic Leaded Chip Carrier Package)**

1) Does not include plastic or metal protrusion of 0.25 max. per side
2) Dimension from center to center

GPM05620

SMD = Surface Mounted Device   Dimensions in mm
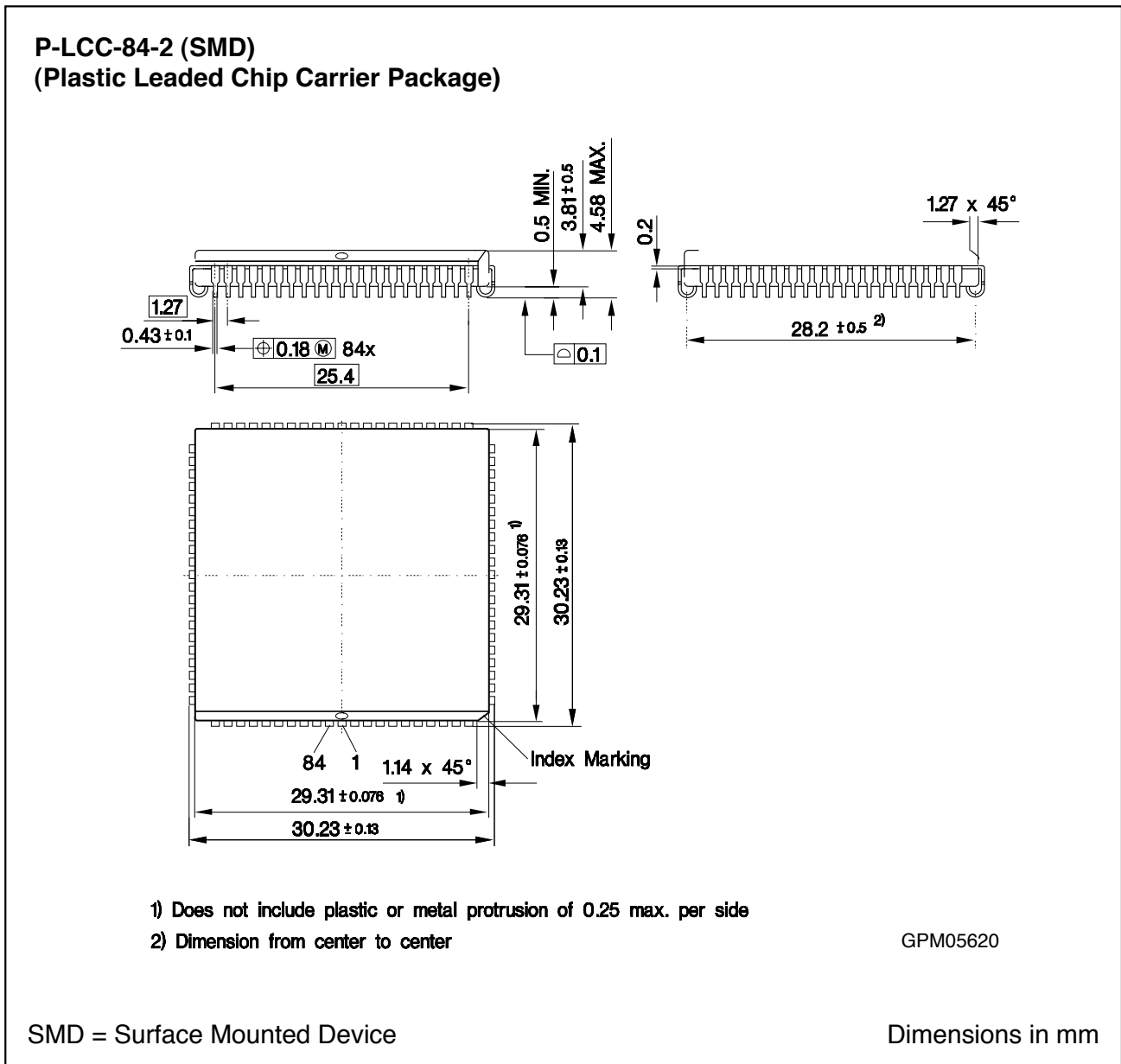
**Figure 5-4
P-LCC-84-2 Package Outline**

5.3 MQFP Packages

P-MQFP-44-2 (SMD)
(Plastic Metric Quad Flat Package)

1) Does not include plastic or metal protrusion of 0.25 max. per side

GPM05622

SMD = Surface Mounted Device   Dimensions in mm
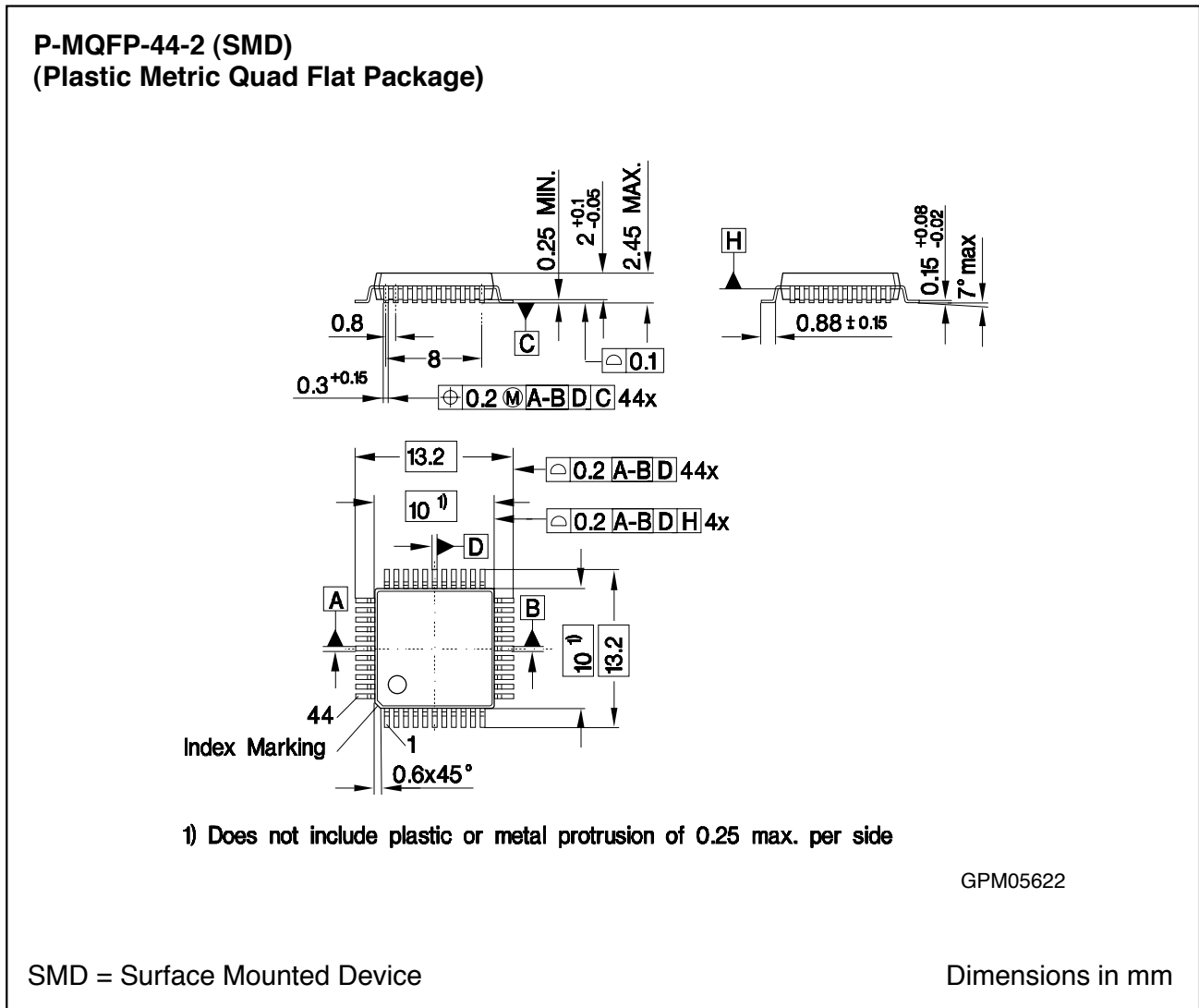
Figure 5-5
P-MQFP-44-2 Package Outline

SIEMENS

P-MQFP-80-1 (SMD)
(Plastic Metric Quad Flat Package)

0.25 min
2 +0.1 -0.05
2.45 max

0.15 +0.08 -0.02
7° max

0.65
0.3 ±0.08
12.35
0.88 ±0.15

17.2
14 1)

17.2
14 1)

80
1
Index Marking
0.6x45°

1) Does not include plastic or metal protrusions of 0.25 max per side

GPM05249

SMD = Surface Mounted Device　　　　Dimensions in mm
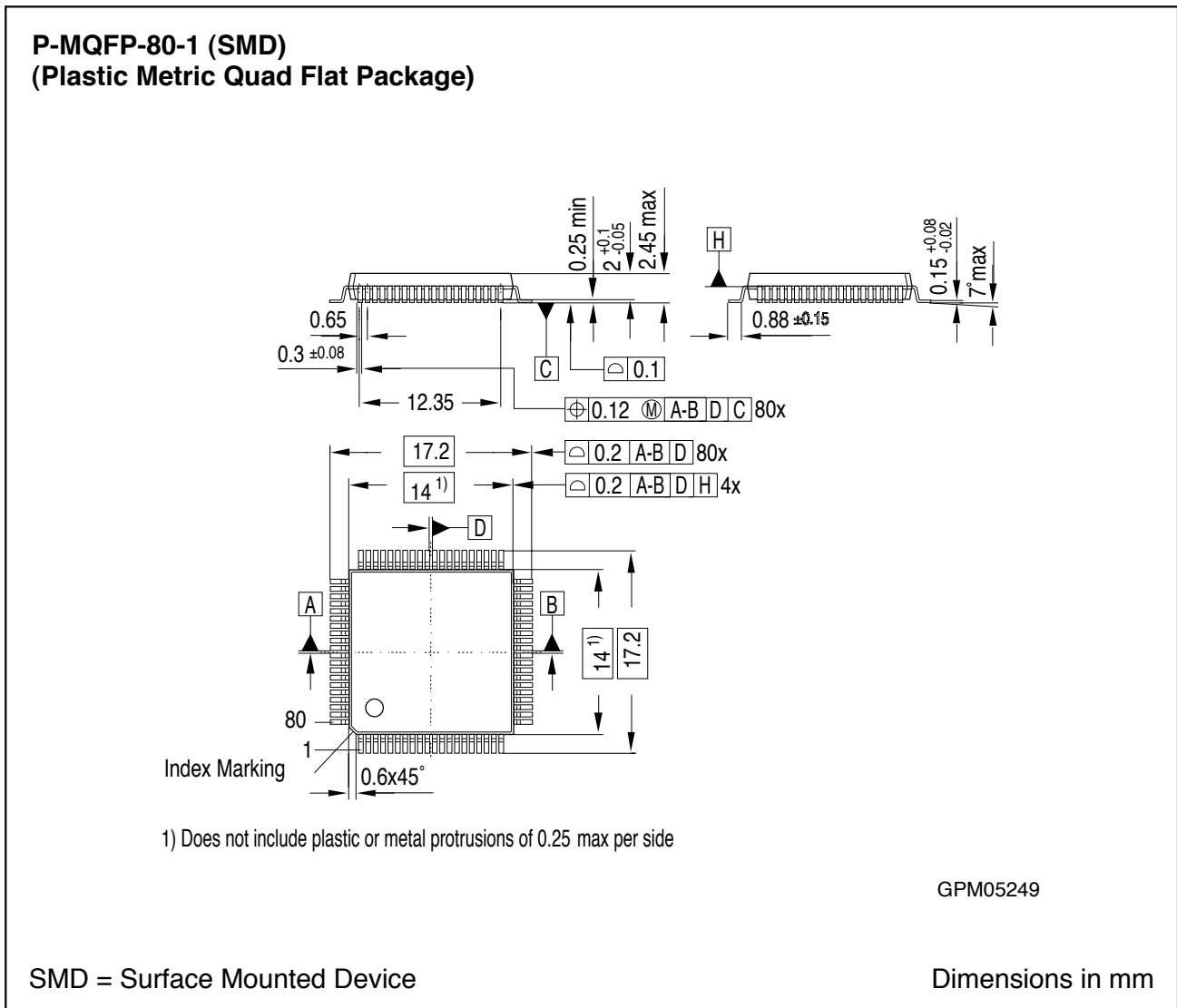
Figure 5-6
P-MQFP-80-1 Package Outline

P-MQFP-100-2 (SMD)
(Plastic Metric Quad Flat Package, rectangular)

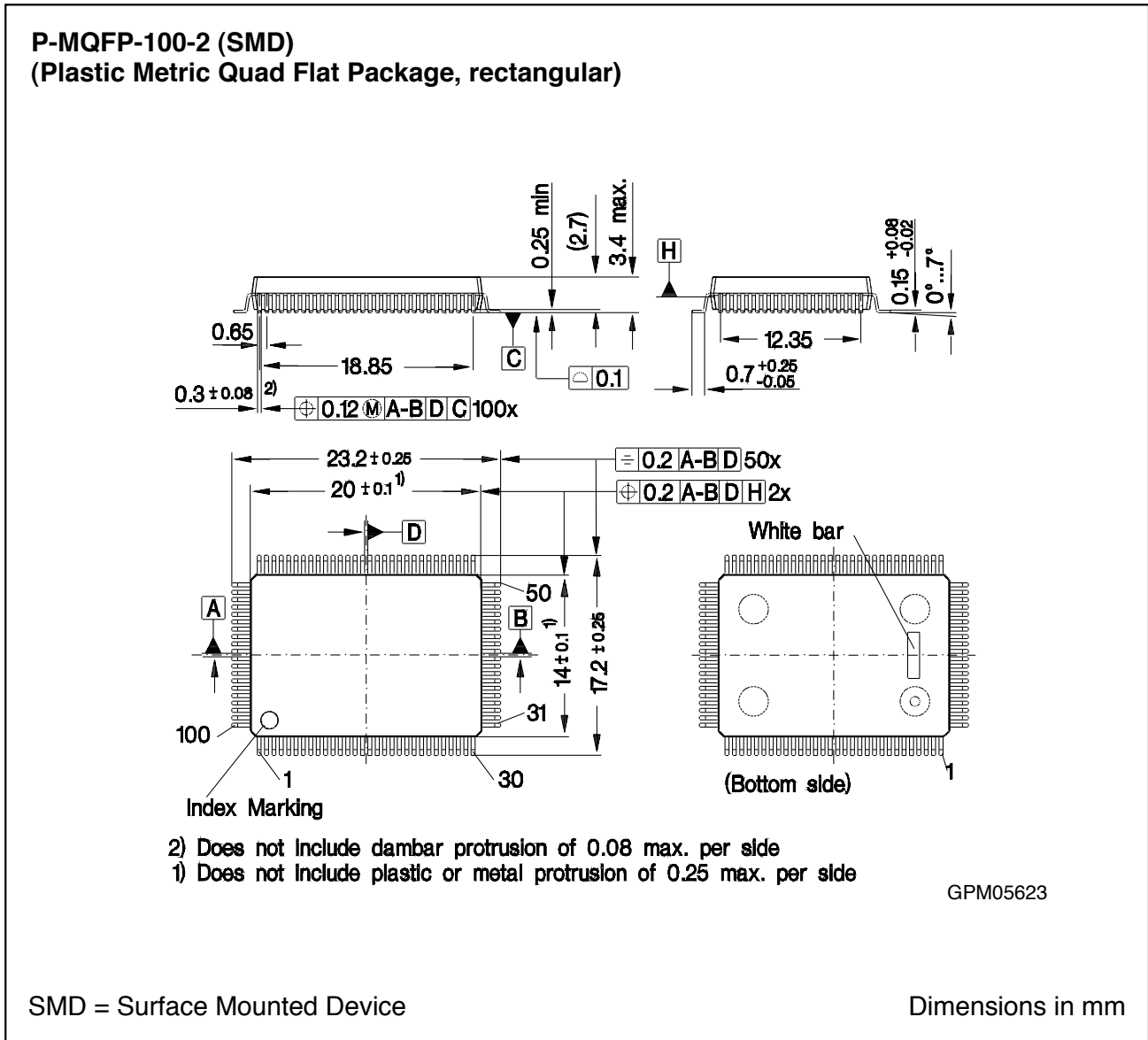SMD = Surface Mounted Device   Dimensions in mm

Figure 5-7
P-MQFP-100-2 Package Outline