TSC 80251

Programmer’s Guide

Rev E – 2000

Atmel Wireless & Microcontrollers reserves the right to make changes in the specifications contained in this document in order to improve design or performance and to supply the best possible products.Atmel also assumes no responsibility for the use of any circuits described herein, conveys no license under any patents or other rights, and makes no representations that the circuits are free from patent infringement. Applications for any integrated circuits contained in this publication are for illustration purposes only andAtmel makes no representation or warranty that such applications will be suitable for the use specified without further testing or modification. Reproduction of any portion hereof without the prior written consent of Atmel is prohibited.

On line information

World Wide Web: http://www.atmel–wm.com

Factory Technical Support

Email: micro@atmel–wm.com

Publisher

Atmel Nantes S.A.
La Chantrerie – Route de Gachet,
BP 70602
44306 NANTES Cedex 03
France
Phone: 33 2 40 18 18 18
Fax: +33 2 40 18 19 60

Table of Contents

Conventions

Chapter 1: Introduction

Chapter 2: Architectural Overview

Chapter 3: Address Spaces

Chapter 4: Programming

Chapter 5: Instruction Set

Glossary

List of Figures

Chapter 2: Architectural Overview 2.1

Chapter 3: Address Spaces 3.1

Chapter 4: Programming 4.1

List of Tables

Chapter 3: Address Spaces

Chapter 4: Programming

Chapter 5: Instruction Set

Conventions

The following notations and terminology are used in this manual. The Glossary defines all terms with special meanings.

#

The pound symbol (#) has either of two meanings, depending on the context. When used with a signal name, the symbol means that the signal is active low. When used in an instruction, the symbol prefixes an immediate value in immediate addressing mode.

italics

Italics identify variables and introduce new terminology. The context in which italics are used distinguishes between two possible meanings.
Variables in registers and signal names are commonly represented by x and y, where x represents the first variable and y represents the second variable. For example, in register Px.y, x represents the variable that identifies the specific port, and y represents the register bit variable [7:0]. Variables must be replaced with the correct values when configuring or programming registers or identifying signals.

XXXX

Uppercase X (no italics) represents an unknown value or a “don’t care” state or condition. The value may be either binary or hexadecimal, depending on the context. For example, 2XAFh (hex) indicates that bits 11:8 are unknown; 10XXb in binary context indicates that the two Least Significant Bits are unknown.

Assert and Deassert

The terms Assert and Deassert refer to the act of making a signal active (enabled) and inactive (disabled), respectively. The active polarity (high/low) is defined by the signal name. Active–low signals are designated by a pound symbol (#) suffix; active–high signals have no suffix. To assert RD# is to drive it low; to assert ALE is to drive it high; to deassert RD# is to drive it high; to deassert ALE is to drive it low.

Instructions

Instruction mnemonics are shown in upper case to avoid confusion. You may use either upper case or lower case.

Logic 0 (Low)

An input voltage level equal to or less than the maximum value of V_{IL} or an output voltage level equal to or less than the maximum value of V_{OL}. See Product Datasheet for values.

Logic 1 (High)

An input voltage level equal to or greater than the minimum value of V_{IH} or an output voltage level equal to or greater than the minimum value of V_{OH}. See Product Datasheet for values.

Numbers

Hexadecimal numbers are represented by a string of hexadecimal digits followed by the letter h. Decimal and binary numbers are represented by their customary notations: i.e. 255 is a decimal number and 1111 1111 is a binary number. In most cases of binary numbers, the letter b is added for clarity.

Register Bits

Bit locations are indexed by 7:0 for byte registers, 15:0 for word registers, and 31:0 for double word (dword) registers. Bit 0 is the least significant bit and 7, 15 or 31 are the most significant bits. An individual bit is represented by the register name, followed by a period and the bit number. For example, PCON.4 is bit 4 of the Power Control register. In some discussions, bit names are used. For example, the name of PCON.4 is POF, the Power Off flag.

Register Names

Register names are shown in upper case. For example, PCON is the Power Control register. If a register name contains a lowercase character, it represents more than one register. For example, CCAPMx (x = 0, 1, 2, 3, 4) represents the five registers: CCAPM0 through CCAPM4.

Reserved Bits

Some registers contain reserved bits. These bits are not used in this device but they may be used in future implementations. Pay attention to the recommendations when manipulating theses bits.

Set and Clear

The terms Set and Clear refer to the value of a bit or the act of giving it a value. If a bit is Set, its value is “1”; setting a bit gives it a “1” value. If a bit is Clear, its value is “0”; clearing a bit gives it a “0” value.

Introduction

1.1. 8/16–bit microcontroller

In the world of 8/16–bit microcontrollers, the C51 Architecture has become an industry standard for embedded applications. For over 15 years, Atmel Wireless & Microcontrollers has been a leading provider of this microcontroller family. This unsurpassed experience is the driving force as Atmel takes this proven family to the next level of performance: the TSC80251 family!

This new C251 Architecture at its lowest performance level (binary mode), is binary code compatible with the 80C51 microcontrollers, hence, attaining an increase in performance has never been easier.

Due to a 3–stage pipeline, the CPU–performance is increased by a factor 5, using existing C51 code without modifications.
Using the new C251 instruction set, which you will find in this document (See Chapter 5), the performance will increase up to 15 times at the same clock rate. This performance enhancement is based on the 16–bit instruction bus, allowing for more powerful instructions and additional internal instruction bus, 8–bit and 16–bit data busses.
The 24–bit address bus will allow to access up to 16 Mbytes in a single linear memory space. Please see each individual TSC80251 Product Design Guide for the effective addressable memory range.

Programming flexibility and C–code efficiency are both increased through a Register–based Architecture, the 64–Kbyte extended stack space combining with the new instruction set.
C251 C–compilers are some of the most efficient available (nearly no overhead), coupled with the final codesize which could be a factor of 3 down when compared with the C51 C–compilers.

All technical information in this document about core features are related to the core revision A and core revision D.

1.2. TSC80251 Derivatives

Atmel Wireless & Microcontrollers is developing a full family of application specific TSC80251 derivatives. Please see the Design Guide of each product for further information.

These products are designed to help you getting high–performance products to market faster.
Due to the high instruction throughput, the TSC80251 derivatives are focussing on all high–end 8–bit to 16–bit applications.

TSC80251 derivatives are also used in mid–range and lower–end microcontroller applications, where a very low operation frequency is needed, without decreasing the level of CPU–power.
This feature is ideal for today portable applications and EMC sensitive systems.

Typical applications for this family are:

- Automotive:
  - Airbag
  - ABS
  - Gearbox
  - Climate control
  - Car radio
  - Car navigation
- Communication:
  - Cordless phones
  - Cellular phones
  - High speed modems
  - High–end feature phones
  - ISDN phones
  - Line cards
  - Network termination
- Computer:
  - High–end monitors
  - DVD–ROM
  - Magtape card & smart card readers
  - Barcodes readers
  - Computer telephony
  - Force feedback joysticks
- Industrial:
  - Process monitoring control & readouts
  - Air conditioning systems
  - Automation

Atmel’s TSC80251 derivatives are designed around the C251 core, using standard peripherals dedicated to a targetted range of applications.

Here is a selection of peripheral blocks:

- Serial interfaces:
  - UART (Universal Asynchronous Receiver Transmitter)
  - I2C (Inter–Integrated Circuit)
  - SPI (Serial Protocol Interface)
  - µWire (Synchronous Serial Interface)
- Special Functions:
  - PCA: Programmable Counter Array (5 × 16–bit modules)
  - High–speed output
  - Compare/Capture I/O
  - 8–bit Pulse Width Modulator (PWM)
  - ADC (Analog to Digital Converter)
  - Smart sensor interfaces with PMU (Pulse Measurement Unit)
- Control functions:
  - Watchdog Timer
  - Timers/Counters
  - Power monitoring and management
  - Interrupt handler
- Memories:
  - RAM
  - ROM
  - EPROM/OTPROM

Most of TSC80251 derivatives are available as ROMless, OTPROM, EPROM and Mask ROM version. For any special request, refer to sales representative.

1.3. TSC80251 Documentation

The following documentation and starter tools are available to allow the full evaluation of the Atmel's TSC80251 derivatives:

- "TSC80251 Programmer's Guide"
  Contains all information for the programmer (Architecture, Instruction Set, Programming).
- "TSC80251 Design Guide"
  Contains all product specific data and a summary of available application notes.
- Application Notes
- "TSC80251 Product Starter Kit"
  This kit enables the product to be evaluated by the designer.
  Its contents is:
  - C–Compiler (limited to 2 Kbytes of code)
  - Assembler
  - Linker
  - Product Simulator
  - TSC80251 Product Evaluation Board with ROM–Monitor
  - EPROM and ROMless samples of the available derivatives
  - Please visit our WWW for updated versions in ZIP format.
- World Wide Web
  Please contact our WWW for possible updated information at http://www.atmel–wm.com
- Technical support: micro@atmel–wm.com

Architectural Overview

2.1. Microcontroller Architecture

The TSC80251 family of 8/16–bit microcontrollers is a high performance upgrade of the widely used 80C51 microcontrollers. It extends features and performance while maintaining binary code compatibility, so the impact on existing hardware and software is minimal.

The C251 Architecture core contains:

- 24–bit linear addressing and up to 16 Mbytes of memory
- a register file based CPU with registers accessible as bytes, words, and double words
- a page mode for accelerating external instruction fetches
- an instruction pipeline
- an enriched instruction set, including 16–bit arithmetic and logic instructions
- a 64–Kbyte extended stack space
- a minimum instruction–execution time of two clocks (vs. 12 clocks for 80C51 microcontrollers)
- binary–code compatibility with 80C51 microcontrollers

Several benefits are derived from these features :

- preservation of code written for 80C51 microcontrollers
- a significant increase in core execution speed in comparison with 80C51 microcontrollers at the same clock rate
- support for larger programs and more data
- increased efficiency for code written in C language

Figure 2.1. is a functional block diagram of TSC80251 microcontrollers. The core, which is common to all TSC80251 microcontrollers, is described in the next paragraph. Each derivative in the family has its own on–chip peripherals, I/O Ports, external bus, size of on–chip RAM, type and size of on–chip ROM.

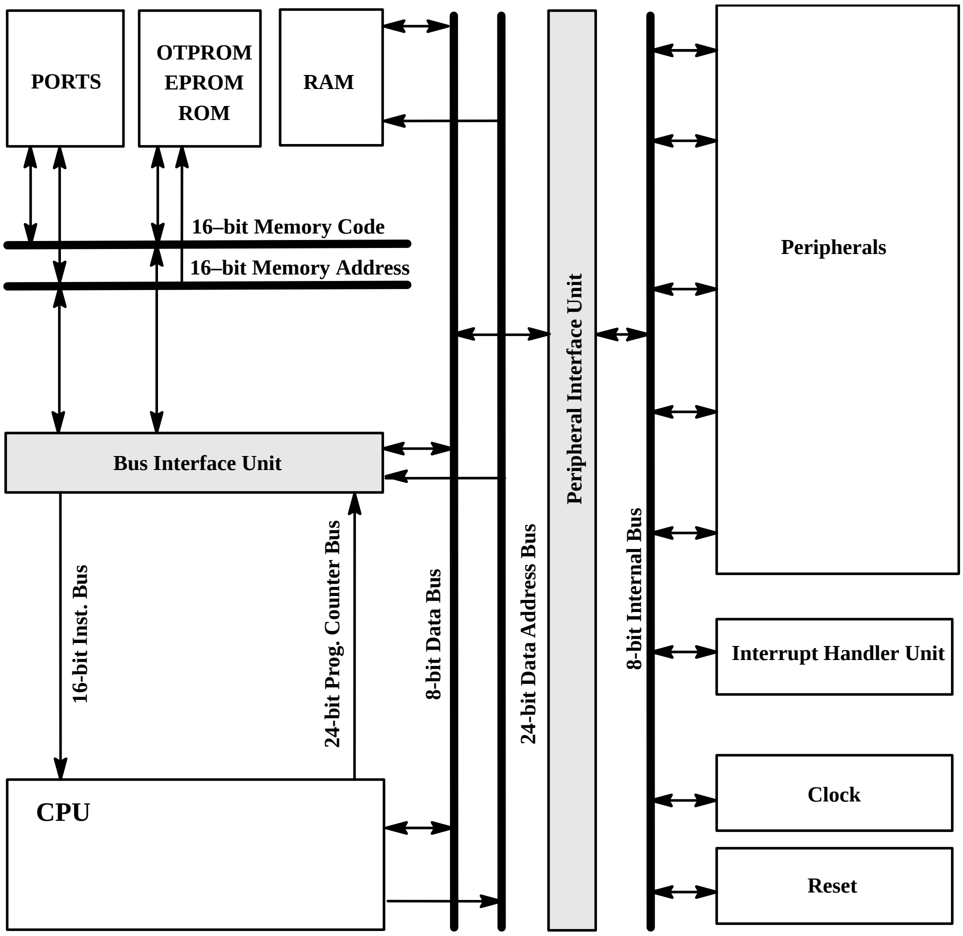


Figure 2.1. TSC80251 Product Block Diagram

2.2. Microcontroller Core

The TSC80251 microcontroller core contains the CPU, the clock and reset unit, the interrupt handler, the bus interface and the peripheral interface (See Figure 2.1.). The CPU contains the instruction sequencer, ALU, register file and data memory interface (See Figure 2.2.).

2.2.1. CPU

The TSC80251 fetches instructions from on–chip code memory two bytes at a time or from external memory one byte at a time. The instructions are sent over the 16–bit instruction bus to the CPU. You can configure the TSC80251 to operate in page mode for accelerated instruction fetches from external memory. In page mode, if an instruction fetch is to the same 256–byte “page” as the previous fetch, the fetch requires one state (two clocks) rather than two states (four clocks). For information regarding the page or non–page mode selection, see Product Design Guide.

The TSC80251 register file has 40 registers, which can be accessed as bytes (8–bit data), words (16–bit data) and double words (32–bit data). As in the C51 Architecture, registers 0-7 consist of four banks of eight registers each, where the active bank is selected by the Program Status Word (PSW) for fast context switches (See “Programming” chapter).

The TSC80251 CPU is a pipeline machine. When the pipeline is full and code is executing from on–chip code memory, an instruction can be completed every state time. When the pipeline is full and code is executing from external memory (with no wait states and no extension of the ALE signal) an instruction can be completed every two state times.

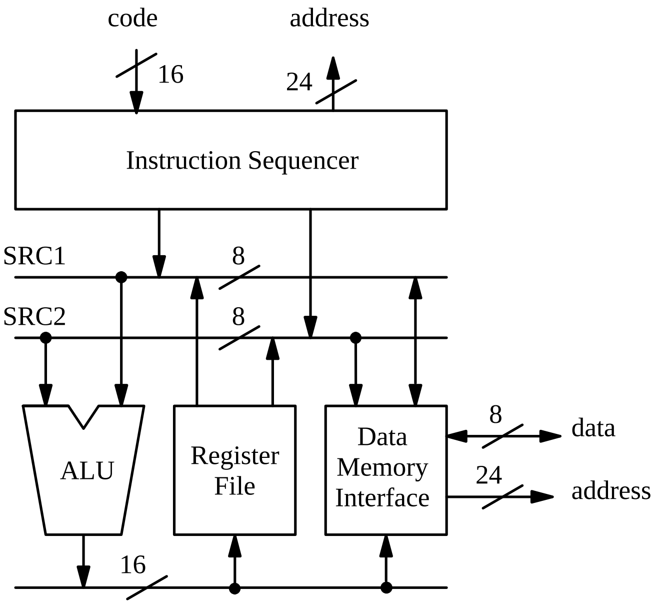


Figure 2.2. Central Processor Unit Block Diagram

2.2.2. Clock and Reset Unit

The timing source for the TSC80251 microcontroller can be an external oscillator or an internal oscillator with an external crystal/resonator. The basic unit of time in TSC80251 is the state time (or state), which is two oscillator periods. The state time is divided into phase P1 and phase P2 (See Figure 2.3.).

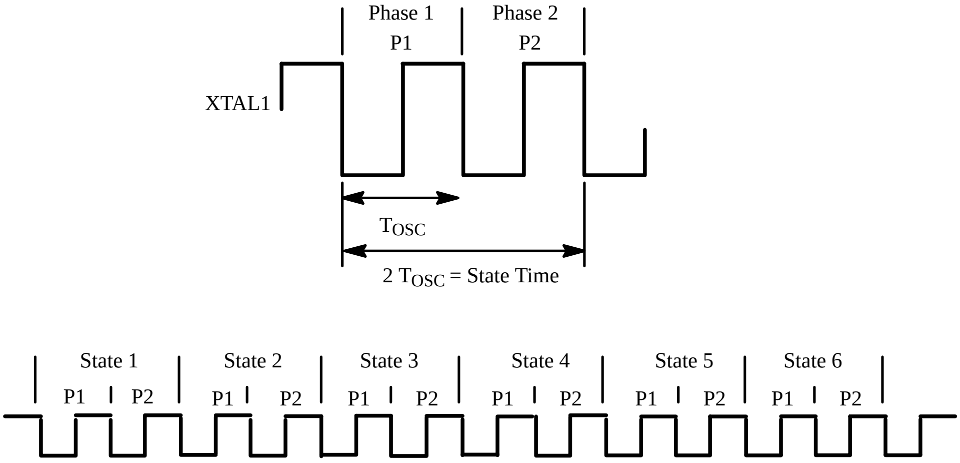


Figure 2.3. Clocking Definitions

The TSC80251 peripherals operate on a peripheral cycle, which is six state times (this peripheral cycle is not a characteristic of the C251 Architecture). A one–clock interval in a peripheral cycle is denoted by its state and phase (SxPy). *For simplicity purpose, XTAL1 signal has been used in this figure. In fact this is the prescaler output that drives the core. The clock prescaler being a software programmable device, the effective core clock can be dynamically adapted to the application speed and power consumption needs.*

The reset unit places the TSC80251 into a known state. A chip reset is initiated by asserting the RST pin or allowing the Watchdog Timer to time out when the TSC80251 has one.

2.2.3. Interrupt Handler Unit

The Interrupt Handler Unit can receive interrupt requests from many sources: internal peripheral sources, external sources and TRAP instruction. When the interrupt handler grants an interrupt request, the CPU discontinues the normal flow of instructions and branches to a routine that services the source that requested the interrupt. You can enable or disable the interrupts individually (except for TRAP and NMI which cannot be disabled) and you can chose among one to four priority levels for each interrupt.

Address Spaces

TSC80251 microcontrollers have three address spaces: a memory space, a Special Function Register (SFR) space and a register file. This chapter describes these address spaces as they apply to all TSC80251 microcontrollers. It also discusses the compatibility of the C251 Architecture and the C51 Architecture in terms of their address spaces.

1.1. C251 Architecture Address Spaces

Figure 3.1. shows the three address spaces: i.e. memory space, SFR space and register file for TSC80251 microcontrollers. The address spaces are depicted as being 8–byte wide with addresses increasing from left to right and from bottom to top (See Figure 3.1.).

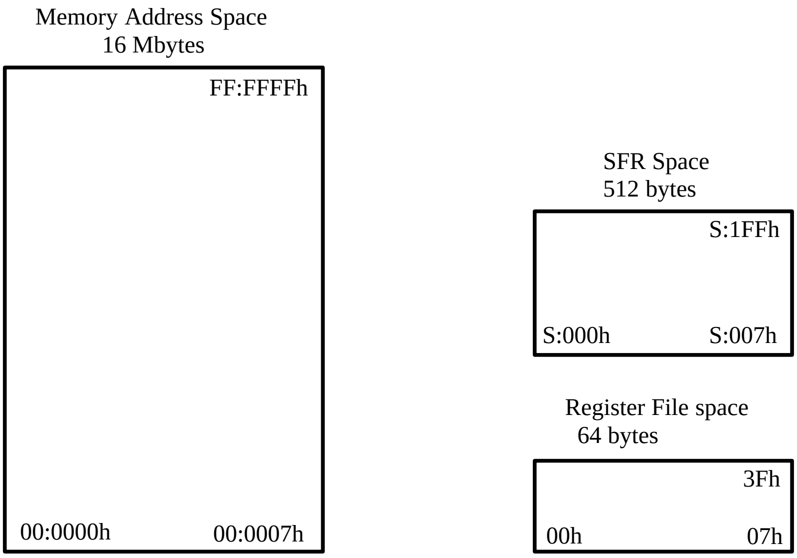


Figure 3.1. Address Spaces for TSC80251 Microcontrollers

It is convenient to view the unsegmented, 16–Mbyte memory space as consisting of 256 64–Kbyte regions, numbered 00: to FF:.

Note :
The memory space in the C251 Architecture is unsegmented. The 64– Kbyte “region” 00:, 01:, ..., FF: are introduced only as a convenience for discussions. Addressing in the C251 Architecture is linear; there are no segment registers.

TSC80251 microcontrollers can have up to 64–Kbytes of on–chip code memory in region FF:. On–chip data RAM begins at location 00:0000h. The first 32 bytes (00:0000h-00:001Fh) provide storage for a part of the register file.The sizes of the on–chip code memory and on–chip RAM depend on the particular device.

The register file has its own address space (See Figure 3.1.). The 64 locations in the register file are numbered decimally from 0 to 63. Locations 0-7 represent one of four, switchable register banks, each having 8 registers. The 32 bytes required for these banks occupy locations 00:0000h-00:001Fh in the memory space. Register file locations 8-63 do not appear in the memory space and are new hardware resources of the C251 Architecture.

The SFR space can accommodate up to 512 8–bit Special Function Registers with addresses S:000h-S:1FFh. Some of these locations may be unimplemented in a particular device. In the C251 Architecture, the prefix “S:” is used with SFR addresses to distinguish them from the memory space addresses 00:0000h-00:01FFh.

1.2. C51 Architecture Address Spaces

Figure 3.2. shows the address spaces of the C51 Architecture. Internal data memory locations 00h-7Fh can be addressed directly, indirectly by register addressing mode and bit addressing mode for data locations 20h–2Fh. Internal data locations 80h-FFh can only be addressed indirectly. Directly addressing these locations accesses the SFRs. The 64–Kbyte code memory has a separate memory space. Data in the code memory can be accessed only with the MOVC instruction. Similarly, the 64–Kbyte external data memory can be accessed only with the MOVX instruction.

The register file (registers R0-R7) comprises four, switchable register banks, each having 8 registers. The 32 bytes required for the four banks occupy locations 00h-1Fh in the on–chip data memory.

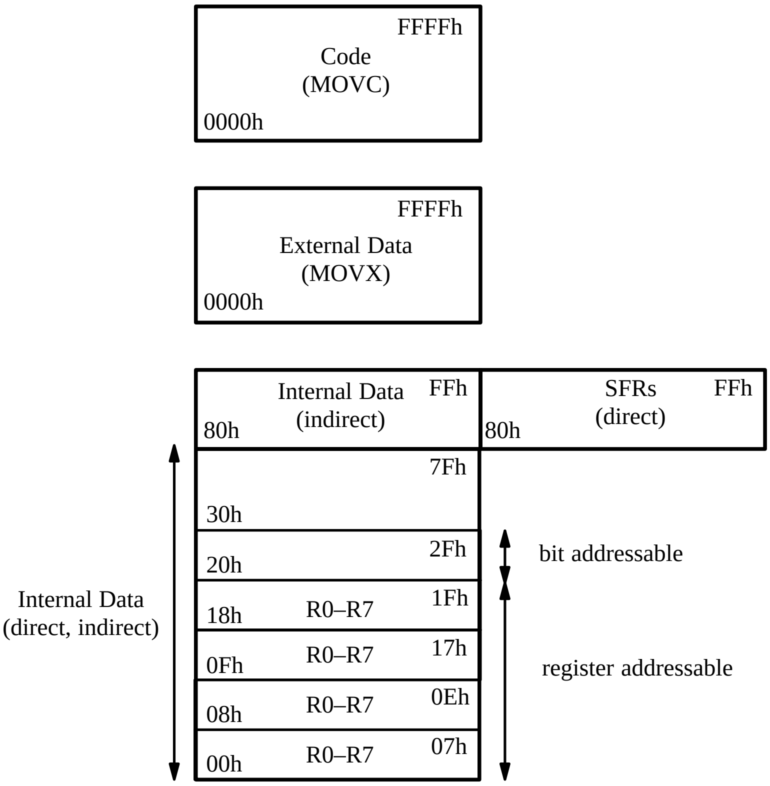


Figure 3.2. Address Spaces for the C51 Architecture

1.3. C51 Architecture mapping to C251 Architecture Address Spaces

The 64–Kbyte code memory for 80C51 microcontrollers maps into region FF: of the memory space for TSC80251 microcontrollers. Assemblers for TSC80251 microcontrollers assemble code for 80C51 microcontrollers into region FF:, and data accesses to code memory (MOVC) are directed to this region. The assembler also maps the interrupt vectors to region FF:. This mapping is transparent to the user; code executes just as with a 80C51 micro without modification.

Table 3.1. Address Mappings

Memory Type	C51 Architecture			C251 Architecture
	Size	Location	Data Addressing	Location
Code	64 Kbytes	0000h-FFFFh	Indirect using MOVC	FF:0000h-FF:FFFFh
External Data	64 Kbytes	0000h-FFFFh	Indirect using MOVX	01:0000h-01:FFFFh
Internal Data	128 bytes	00h-7Fh	Direct, Indirect	00:0000h-00:007Fh
	128 bytes	80h-FFh	Indirect	00:0080h-00:00FFh
SFRs	128 bytes	S:80h-S:FFh	Direct	S:0080h-S:0FFh
Register	8 bytes	R0-R7	Register	00:0000h–00:001Fh

The 64–Kbyte external data memory for 80C51 microcontrollers is mapped into the memory region specified by bits 16–23 of the data pointer DPX, i.e., DPXL, which is accessible as register file location 57 and also as SFR at S:084h. The reset value of DPXL is 01h, which maps the external memory to region 01: as shown in Figure 3.3. You can change this mapping by writing a different value to DPXL. A mapping of the C51 Architecture external data memory into any 64–Kbyte memory region in the C251 Architecture provides complete runtime compatibility because the lower 16 address bits are identical in both architectures.

The 256 bytes of on–chip data memory for 80C51 microcontrollers (00h–FFh) are mapped to addresses 00:0000h–00:00FFh to ensure complete runtime compatibility. In the C51 Architecture, the lower 128 bytes (00h–7Fh) are directly and indirectly addressable; however the upper 128 bytes are accessible by indirect addressing only. In the C251 Architecture, all locations in region 00: are accessible by direct, indirect, and displacement addressing.

The 128–byte SFR space for 80C51 microcontrollers is mapped into the 512–byte SFR space of the C251 Architecture starting at address S:080h, as shown in Figure 3.3. This provides complete compatibility with direct addressing of 80C51 microcontroller SFRs (including bit addressing). The SFR addresses are unchanged in the new Architecture. In the C251 Architecture, SFRs, A, B, DPL, DPH and SP, as well as the new DPXL and SPH, reside in the register file for high performance. However, to maintain compatibility, they are also mapped into the SFR space at the same addresses as in the C51 Architecture.

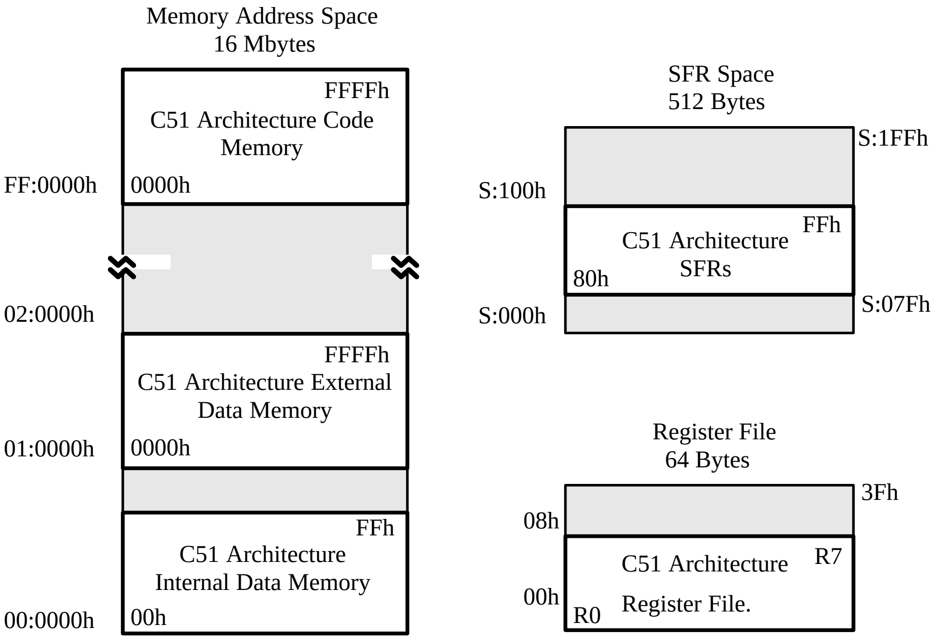


Figure 3.3. Mappings C51 Architecture to C251 Architecture Address Spaces

Figure 3.4. TSC80251 Memory Space

1.4. TSC80251 Register File

The TSC80251 register file consists of 40 byte locations: 0-31 and 56-63, as shown in Figure 3.5. These locations are accessible as bits, bytes, words and dwords. Several locations are dedicated to special registers; the others are general–purpose registers.

Register file locations 0-7 actually consist of four switchable banks of eight registers each, as illustrated in Figure 3.6. The four banks are implemented as the first 32 bytes of on–chip RAM and are always accessible as locations 00:0000h-00:001Fh in the memory address space. Only one of the four banks is accessible via the register file at a given time. The accessible, or “active”, bank is selected by bits RS1 and RS0 in the PSW register, as shown in Table 3.2. This bank selection can be used for fast context switches.

Register file locations 8-31 and 56-63 are always accessible. These locations are implemented as registers in the CPU. Register file locations 32-55 are reserved and cannot be accessed.

Table 3.2. Register Bank Selection

Bank	Address Range	PSW Selection Bits	
		RS1	RS0
Bank 0	00h-07h	0	0
Bank 1	08h-0Fh	0	1
Bank 2	10h-17h	1	0
Bank 3	18h-1Fh	1	1

1.4.1. Byte, Word and Dword Registers

Depending on its location in the register file, a register is addressable as a byte, a word, or a dword, as shown in the right side of Figure 3.5. A register is named for its lowest numbered byte location. For instance:

- R4 is the byte register consisting of location 4.
- WR4 is the word register consisting of registers 4 and 5.
- DR4 is the dword register consisting of registers 4, 5, 6, and 7.

Locations R0-R15 are addressable as bytes, words or dwords. Locations 16-31 are addressable only as words or dwords. Locations 56-63 are addressable only as dwords. Registers are addressed only by the names shown in Figure 3.5. , except for the 32 registers that comprise the four banks of registers R0-R7, which can also be accessed as locations 00:0000h-00:001Fh in the memory space (see Figure 3.6.).

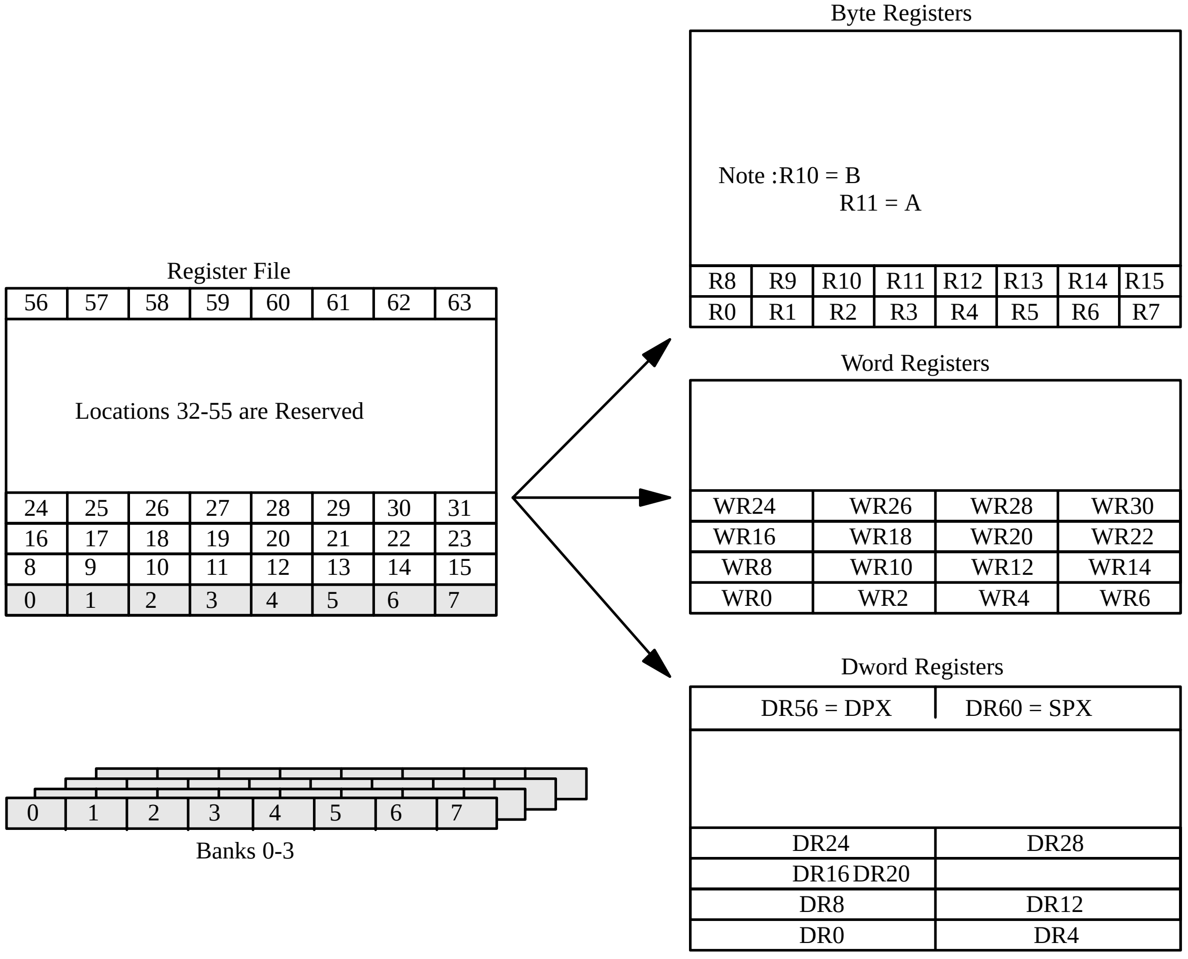


Figure 3.5. Register File in Byte, Word, and Dword Register Views

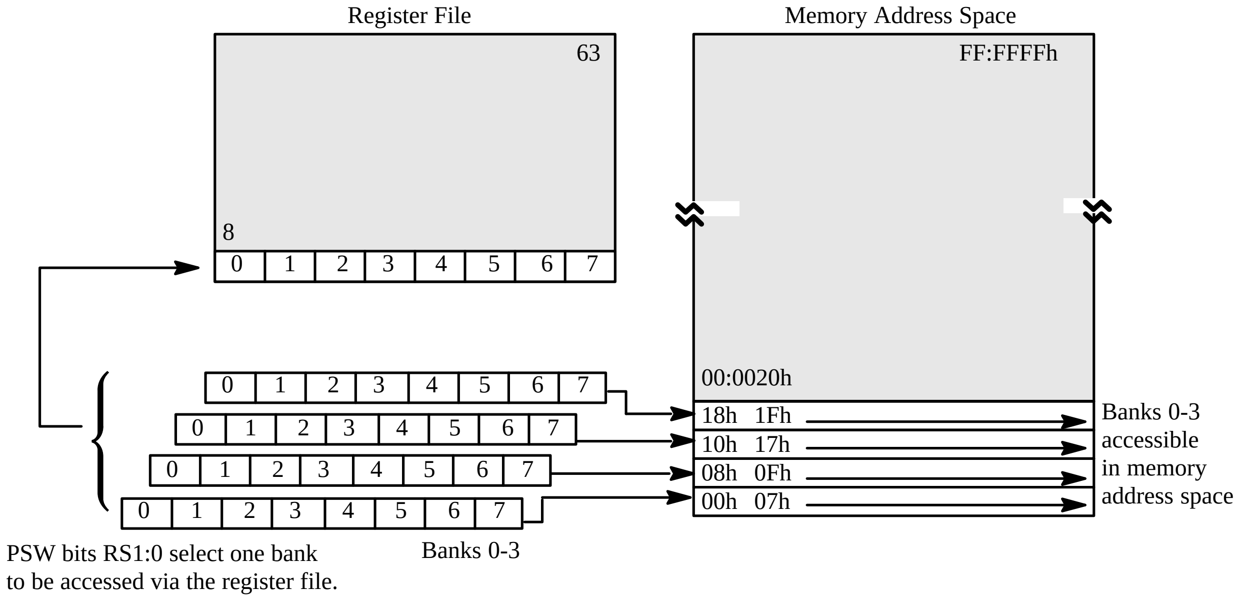

Register File
63
8
0 1 2 3 4 5 6 7
Memory Address Space
FF:FFFFh
00:0020h
18h 1Fh
10h 17h
08h 0Fh
00h 07h
Banks 0-3 accessible in memory address space
PSW bits RS1:0 select one bank to be accessed via the register file.
Banks 0-3

Figure 3.6. Register File Locations 0-7

1.4.2. Dedicated Registers

The register file has four dedicated registers :

- R10 is the B–register.
- R11 is the accumulator (A).
- DR56 is the extended data pointer, DPX.
- DR60 is the extended stack pointer, SPX.

These registers are located in the register file; however, R10, R11 and some bytes of DR56 and DR60 are also accessible as SFRs. The bytes of DPX and SPX can be accessed in the register file only by addressing the dword registers. The dedicated registers in the register file and their corresponding SFRs are illustrated in Figure 3.7. and listed in Table 3.3.

Table 3.3. Dedicated Registers in the Register File and their Corresponding SFRs

Register File						SFRs	
Name			Mnemonic	Reg.	Location	Mnemonic	Address
Stack Pointer (SPX)	–		–	DR60	60	–	–
	–		–		61	–	–
	Stack Pointer, High		SPH		62	SPH	S:BEh
	Stack Pointer, Low		SP		63	SP	S:81h
Data Pointer (DPX)	–		–	DR56	56	–	–
	Data Pointer, Extended Low		DPXL		57	DPXL	S:84h
	DPTR	Data Pointer, High	DPH		58	DPH	S:83h
		Data Pointer, Low	DPL		59	DPL	S:82h
Accumulator (A Register)			A	R11	11	A	S:E0h
B Register			B	R10	10	B	S:F0h

1.4.2.1. Accumulator and B Register

The 8–bit accumulator (A) is byte register R11, which is also accessible in the SFR space as A at S:0E0h (See Figure 3.7.). The B register, used in multiplies and divides, is register R10, which is also accessible in the SFR space as B at S:0F0h. Accessing A or B as a register is one state faster than accessing them as SFRs.

Instructions in the C51 Architecture use the accumulator as the primary register for data moves and calculations. however, in the C251 Architecture, any of registers R1-R15 can serve for these tasks. As a result, the accumulator does not play the central role that it has in 80C51 microcontrollers.

1.4.2.2. Extended Data Pointer, DPX

Dword register DR56 is the extended data pointer, DPX (See Figure 3.7.). The lower three bytes of DPX (DPL, DPH and DPXL) are accessible as SFRs. DPL and DPH comprise the 16–bit data pointer DPTR. While instructions in the C51 Architecture always use DPTR as the data pointer, instructions in the C251 Architecture can use any word or dword register as a data pointer.

DPXL, the byte in location 57, specifies the region of memory (00:-FF:) that maps into the 64–Kbyte external data memory space in the C51 Architecture. In other words, the MOVX instruction addresses the region specified by DPXL when it moves data to and from external memory. The reset value of DPXL is 01h.

1.4.2.3. Extended Stack Pointer, SPX

Dword register DR60 is the stack pointer, SPX (See Figure 3.7.). The byte at (location 63) is the 8–bit stack pointer, SP, in the C51 Architecture. The byte at location 62 is the stack pointer high, SPH. The two bytes allow the stack to extend to the top of memory region 00:. SP and SPH can be accessed as SFRs.

Two instructions, PUSH and POP directly address the stack pointer. Subroutine calls (ACALL, ECALL, LCALL) and returns (ERET, RET, RETI) also use the stack pointer. To preserve the stack, do not use DR60 as a general–purpose register.

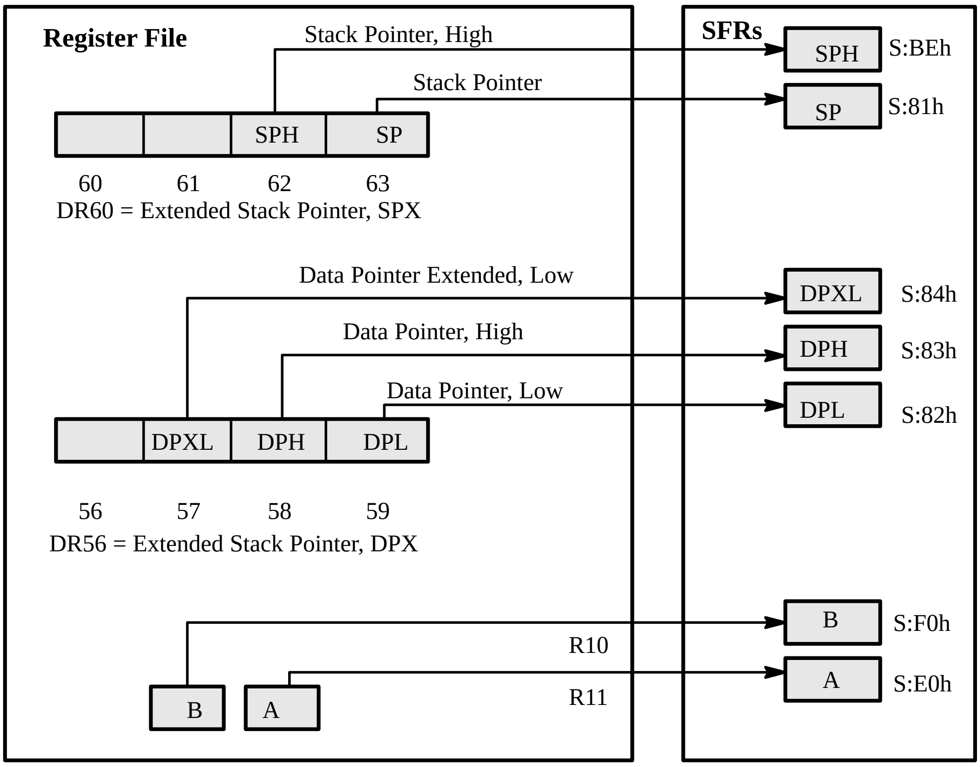


Figure 3.7. Dedicated Registers in the Register File and their Corresponding SFRs

1.5. Special Function Registers (SFRs)

The Special Function Registers (SFRs) reside in their associated on–chip peripherals or in the core. SFR addresses are preceded by “S:” to differentiate them from addresses in the memory space. Unoccupied locations in the SFR space are unimplemented, i.e., no register exists. If an unimplemented SFR location is read, it returns an unspecified value.

Note :
SFRs may be accessed only as bytes; they may not be accessed as words or dwords.

Table 3.4. Core SFRs

Mnemonic	Name	Address
A ★	Accumulator	S:E0h
B ★	B register	S:F0h
PW	Program Status Word	S:D0h
PSW1	Program Status Word 1	S:D1h
SP	Stack Pointer - LSB of SPX	S:81h
SPH ★	Stack Pointer high - MSB of SPX	S:BEh
DPTR ★	Data Pointer (2 bytes)	–
DPL ★	Low Byte of DPTR	S:82h
DPH ★	high Byte of DPTR	S:83h
DPXL ★	Data Pointer, Extended Low	S:84h
IE0	Interrupt Enable Control 0	S:A8h
IE1	Interrupt Enable Control 1	S:B1h
IPL0	Interrupt Priority Control Low 0	S:B8h
IPL1	Interrupt Priority Control Low 1	S:B3h
IPH0	Interrupt Priority Control High 0	S:B7h
IPH1	Interrupt Priority Control High 1	S:B2h

Note:
★ These SFRs can also be accessed by their corresponding registers in the register file (See Table 3.3.

Programming

The instruction set for the C251 Architecture is a superset of the instruction set for the C51 Architecture. This chapter describes the addressing modes and summarizes the instruction set, which is divided into data instructions, bit instructions, and control instructions. (Chapter 5, “Instruction Set Reference” contains an opcode map and the detailed description of each instruction.)

Notes:

1 The instruction execution times given in Chapter 5 are for code executing from on–chip code memory and for data that is read from and written to on–chip RAM. Execution times are increased by executing code from external memory, accessing peripheral SFRs, accessing data in external memory, using a wait state, or extending the ALE pulse.

2 For some instructions, accessing the port SFRs, Px (x = 0, 1, 2, 3) increases the execution time. These cases are noted individually in the tables in Chapter 5.

1. Source Mode or Binary Mode Opcodes

Source mode and Binary mode refer to the two ways of assigning opcodes to the instruction set of the C251 Architecture. Depending on the application, one mode or the other may produce more efficient code. The mode is established during device reset based on the value of the SRC bit in configuration byte CONFIG0. For information regarding the configuration bytes, see the Product Design Guide.

Binary mode and source mode refer to two ways of assigning opcodes to the instruction set for the C251 Architecture. One of these modes must be selected when the chip is configured. Depending on the application, binary mode or source mode may produce more efficient code. This section describes the binary and source modes and provides some guidelines for selecting the mode for your application.

The C251 Architecture has two types of instructions:

- Instructions that originate in the C51 Architecture
- Instructions that are unique to the C251 Architecture

Figure 4.1. shows the opcode map for the binary mode. Area I and area II make up the opcode map for the instructions that are unique to the C251 Architecture. Note that some of these opcodes are reserved for future instructions. The opcode values for areas II and III are identical (06H–FFH). To distinguish between the two areas in binary mode, the opcodes in area III are given the prefix A5H (the A5H instruction is not implemented in the native C51 Architecture). The area III opcodes are thus A506H–A5FFH.

Figure 4.2. shows the opcode map for source mode. Areas II and III have switched places (compare with Figure 4.1.). In source mode, opcodes for instructions in area II require the A5F escape prefix while opcodes for instructions in area III (C251 Architecture) do not.

To illustrate the difference between the binary–mode and source–mode opcodes, Table 4.1. shows the opcode assignments for three sample instructions.

Table 4.1. Examples of Opcodes in Binary and Source Modes

Instruction	Opcode	
	Binary Mode	**Source Mode**
DEC A	14H	14CH
SUBB A, R4	9CH	A59CH
SUB R4, R4	A59CH	9CH

1.1. Selecting Binary Mode or Source Mode

If you have code that was written for a C51 microcontroller and you want to run it unmodified on a C251 microcontroller, choose binary mode. You can use the object code without reassembling the source code. You can also assemble the source code with an assembler for the C251 Architecture and have it produce object code that is binary–compatible with C51 microcontrollers. The remainder of this section discusses the selection of binary mode or source mode for code that may contain instructions from both architectures.

An instruction with a prefixed opcode requires one more byte for code storage, and if an additional fetch is required for the extra byte, the execution time is increased by one state. This means that using fewer prefixed opcodes produces more efficient code.

If a program uses only instructions from the C51 Architecture, the binary–mode code is more efficient because it uses no prefixes. On the other hand, if a program uses many more new instructions than instructions from the C51 Architecture , source mode is likely to produce more efficient code. For a program where the choice is not clear, the better mode can be found by experimenting with a simulator.

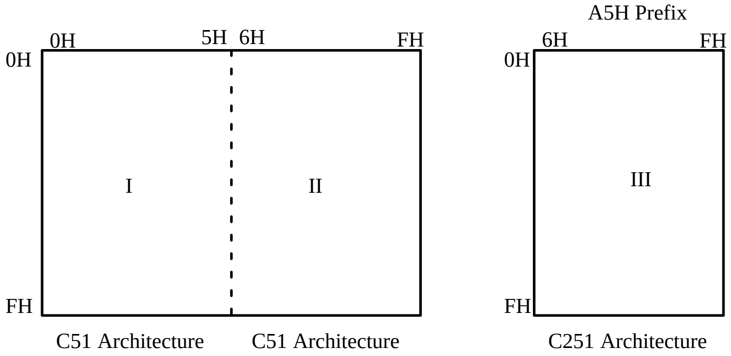


Figure 4.1. Binary Mode Opcode Map

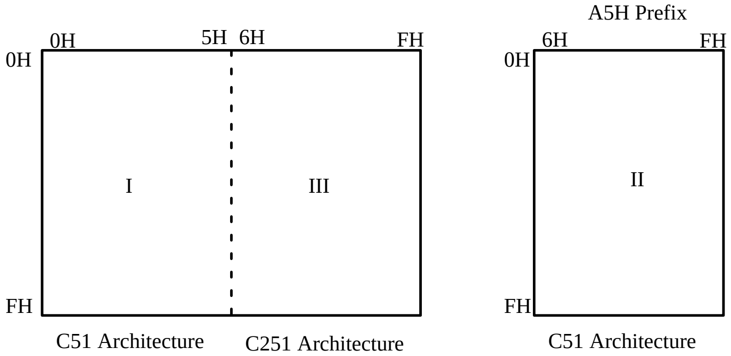


Figure 4.2. Source Mode Opcode Map

2. 4.1. Programming Features of the C251 Architecture

The instruction set for TSC80251 microcontrollers provides the user with new instructions that exploit the features of the C251 Architecture while maintaining compatibility with the instruction set for 80C51 microcontrollers. Many of the new instructions can operate on either 8–bit (byte), 16–bit (word) or 32–bit (dword) operands (In comparison with 8–bit and 16–bit operands, 32–bit operands are accessed with fewer addressing modes.). This capability increases the ease and efficiency of programming TSC80251 microcontrollers in a high–level language such as C.

The instruction set is divided into “Data Instructions”, “Bit Instructions” and “Control Instructions”. Data instructions process 8–bit, 16–bit and 32–bit data; bit instructions manipulate bits; and control instructions manage program flow.

2.1. Data Types

Table 4.2. lists the data types that are addressed by the instruction set. Words or dwords (double words) can be stored in memory starting at any byte address; alignment on two–byte or four–byte boundaries is not required. Words and dwords are stored in memory and the register file in big endian form.

Table 4.2. Data Types

Data Type	Number of Bits
Bit	1
Byte	8
Word	16
Dword (Double Word)	32

2.1.1. Order of Byte Storage for Words and Double Words

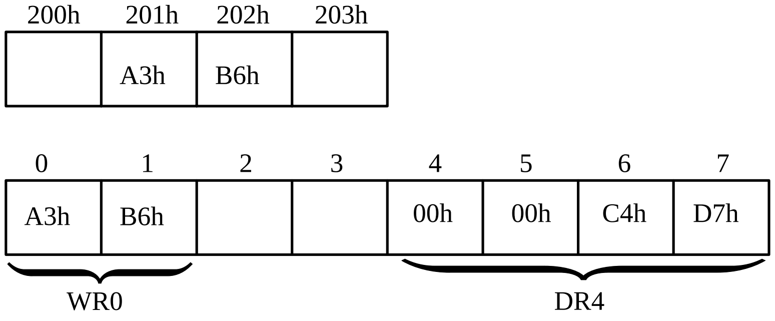
TSC80251 microcontrollers store words (2 bytes) and double words (4 bytes) in memory and in the register file in big endian form. In memory storage, the most significant byte (MSB) of the word or double word is stored in the memory byte specified in the instruction; the remaining bytes are stored at higher addresses, with the least significant byte (LSB) at the highest address. Words and double words can be stored in memory starting at any byte address. In the register file, the MSB is stored in the lowest byte of the register specified in the instruction. The code fragment in Figure 4.3. illustrates the storage of words and double words in big endian form.

2.2. Register Notations

In register–addressing instructions, specific indices denote the registers that can be used in that instruction. For example, the instruction ADD A,Rn uses“Rn” to denote any one of R0, R1, ..., R7; i.e., the range of n is 0-7. The instruction ADD Rm,#data uses “Rm” to denote R0, R1, ..., R15; i.e., the range of m is 0-15. Table 4.3. summarizes the notation used for the register indices. When an instruction contains two registers of the same type (e.g., MOV Rmd,Rms) the first index “d” denotes “destination” and the second index “s” denotes “source”.

2.3. Address Notations

In the C251 Architecture, memory addresses include a region number (00:, 01:, ..., FF:). SFR addresses have a prefix “S:” (S:000h-S:1FFh). The distinction between memory addresses and SFR addresses is necessary, because memory locations 00:0000h-00:01FFh and SFR locations S:000h-S:1FFh can both be directly addressed in an instruction.



Contents of register file and memory after execution:

```
MOV WR0, #A3B6h
MOV 00:0201h, WR0
MOV DR4, #0000C4D7h
```

Figure 4.3. Word and Double-word Storage in Big Endian Form

Table 4.3. Notation for Byte Registers, Word Registers, and Dword Registers

Register Type	Register Symbol	Destination Register	Source Register	Register Range
Byte	Ri	–	–	R0, R1
	Rn	–	–	R0-R7
	Rm	Rmd	Rms	R0-R15
Word	WRj	WRjd	WRjs	WR0, WR2, WR4, ..., WR30
Dword	DRk	DRkd	DRks	DR0, DR4, DR8, ..., DR28,DR56, DR60

Instructions in the C51 Architecture use 80h-FFh as addresses for both memory locations and SFRs, because memory locations are addressed only indirectly and SFR locations are addressed only directly. For compatibility, software tools for TSC80251 controllers recognize this notation for instructions in the C51 Architecture. No change is necessary in any code written for 80C51 microcontrollers.

For new instructions in the C251 Architecture, the memory region prefixes (00:, 01:, ..., FF:) and the SFR prefix (S:) are required. Also, software tools for the C251 Architecture permit 00: to be used for memory addresses 00h-FFh and permit the prefix S: to be used for SFR addresses in instructions in the C51 Architecture.

2.4. Addressing Modes

The C251 Architecture supports the following addressing modes:

- Register addressing
  The instruction specifies the register that contains the operand.
- Immediate addressing
  The instruction contains the operand.
- Direct addressing
  The instruction contains the operand address.
- Indirect addressing
  The instruction specifies the register that contains the operand address.
- Displacement addressing
  The instruction specifies a register and an offset. The operand address is the sum of the register contents (the base address) and the offset.
- Relative addressing
  The instruction contains the signed offset from the next instruction to the target address (the address for transfer of control, e.g., the jump address).
- Bit addressing
  The instruction contains the bit address.

3. Program Status Words

The Program Status Word (PSW) register and the Program Status Word 1 (PSW1) register contain four types of bits (see Figure 4.8. and Figure 4.9.):

- CY, AC, OV, N and Z are flags set by hardware to indicate the result of an operation.
- The P bit indicates the parity of the accumulator.
- Bits RS0 and RS1 are programmed by software to select the active register bank for registers R0-R7.
- F0 and UD are available to the user as general–purpose flags.

The PSW and PSW1 registers are read/write registers; however, the parity bit in the PSW is not affected by a write. Individual bits can be addressed with the bit instructions (“Bit Instructions”). The PSW and PSW1 bits are used implicitly in the conditional jump instructions (“Conditional Jumps”).

The PSW register is identical to the PSW register in 80C51 microcontrollers. The PSW1 register exists only in TSC80251 microcontrollers. Bits CY, AC, RS0, RS1, and OV in PSW1 are identical to the corresponding bits in PSW, i.e., the same bit can be accessed in either register. Table 4.4. lists the instructions that affect the CY, AC, OV, N and Z bits.

Table 4.4. The Efffects of Instructions on the PSW and PSW1 Flags

Instruction Type	Instruction	Flags Affected [1]				
		CY	OV	AC [2]	N	Z
Arithmetic	ADD, ADDC, SUB, CMP	X	X	X	X	X
	INC, DEC				X	X
	MUL, DIV [3]	0	X		X	X
	DA	X			X	X
Logical	ANL, ORL, XRL, CLR A, CPL A, RL, RR, SWAP				X	X
	RLC, RRC, SRL, SLL, SRA [4]	X			X	X
Program Control	CJNE	X			X	X
	DJNE				X	X

Notes :

1. X = the flag can be affected by the instruction. 0 = the flag is cleared by the instruction.

2. The AC flag is affected only by operations on 8–bit operands.

3. If the divisor is zero, the OV flag is set, and the other bits are meaningless.

4. For SRL, SLL and SRA instructions, the last bit shifted out is stored in the CY bit.

4. Data Instructions

Data instructions consist of arithmetic, logical, and data–transfer instructions for 8–bit, 16–bit and 32–bit data. This section describes the data addressing modes and the set of data instructions.

4.1. Data Addressing Modes

This section describes the data addressing modes, which are summarized in two tables: Table 4.6. for the instructions that are native to the C51 Architecture and Table 4.6. for the data instructions unique to the C251 Architecture.

Notes:

- References to registers R0-R7, WR0-WR6, DR0 and DR4 always refer to the register bank that is currently selected by the PSW and PSW1 registers. Registers in all banks (active and inactive) can be accessed as memory locations in the range 00h-1Fh.
- Instructions from the C51 Architecture access external memory through the region of memory specified by byte DPXL in the extended data pointer register, DPX (DR56). Following reset, DPXL contains 01h, which maps the external memory to region 01:. You can specify a different region by writing to DR56 or the DPXL SFR.

4.1.1. Addressable Registers

Both Architectures address registers directly.

- C251 Architecture
  In the register addressing mode, the operand(s) in a data instruction are in byte registers (R0-R15), word registers (WR0, WR2, ..., WR30) or dword registers (DR0, DR4, ..., DR28, DR56, DR60).
- C51 Architecture
  Instructions address registers R0-R7 only.

4.1.2. Immediate Addressing

- C251 Architecture
  In the immediate addressing mode, the instruction contains the data operand itself. Byte operations use 8–bit immediate data (#data); word operations use 16–bit immediate data (#data16). Dword operations use 16–bit immediate data in the lower word and either zeros in the upper word (denoted by #0data16) or ones in the upper word (denoted by #1data16). MOV instructions that place 16–bit immediate data into a dword register (DRk), place the data either into the upper word while leaving the lower word unchanged, or into the lower word with a sign extension or a zero extension.
  The increment and decrement instructions contain immediate data (#short = 1, 2, or 4), which specifies the amount of the increment/decrement.
- C51 Architecture
  Instructions use only 8–bit immediate data (#data).

4.1.3. Direct Addressing

- C251 Architecture
  In the direct addressing mode, the instruction contains the address of the data operand. The 8–bit direct mode addresses on–chip RAM (dir8 = 00:0000h-00:007Fh) as both bytes and words, and addresses the SFRs (dir8 = S:080h-S:1FFh) as bytes only. The 16–bit direct mode addresses both bytes and words in memory (dir16 = 00:0000h-00:FFFFh).
- C51 Architecture
  The 8–bit direct mode addresses 256 bytes of on–chip RAM (dir8 = 00h-7Fh) as bytes only and the SFRs (dir8 = 80h-FFh) as bytes only.

Table 4.5. Addressing Modes for Data Instruction in the C51 Architecture

Mode	Address Range of Operand	Assembly Language Reference	Comments
Register	00h-1Fh	R0-R7 (Bank selected by PSW)	
Immediate	Operand in Instruction	#data = #00h-#FFh	
Direct	00h-7Fh	dir8 = 00h-7Fh	On-chip RAM
	SFRs	dir8 = 80h-FFh or SFR mnemonic	SFR address
Indirect	00h-FFh	@R0, @R1	Accesses on-chip RAM or the lowest 256 bytes of external data memory (MOVX)
	0000h-FFFFh	@DPTR, @A+DPTR	Accesses external data memory (MOVX)
	0000h-FFFFh	@A+DPTR, @A+PC	Accesses region FF : of code memory (MOVC)

4.1.4. Indirect Addressing

In arithmetic and logical instructions that use indirect addressing, the source operand is always a byte, and the destination is either the accumulator or a byte register (R0-R15). The source address is a byte, word or dword. The two architectures do indirect addressing via different registers:

- C251 Architecture
  Memory is indirectly addressed via word and dword registers :
  - Word register (@WRj, j = 0, 2, 4, ..., 30)
    The 16–bit address in WRj can access locations 00:0000h-00:FFFFh.
  - Dword register (@DRk, k = 0, 4, 8, ..., 28, 56, and 60)
    The 24 least significant bits can access the entire 16–Mbyte address space. The upper eight bits of DRk must be 0. (If you use DR60 as a general data pointer, be aware that DR60 is the extended stack pointer register SPX.)
- C51 Architecture
  Instructions use indirect addressing to access on–chip RAM, code memory, and external data RAM.
  - Byte register (@Ri, i = 0, 1)
    Registers R0 and R1 indirectly address on–chip memory locations 00h-FFh and the lowest 256 bytes of external data RAM.
  - 16–bit data pointer (@DPTR or @A+DPTR)
    The MOVC and MOVX instructions use these indirect modes to access code memory and external data RAM.
  - 16–bit program counter (@A+PC)
    The MOVC instruction uses this indirect mode to access code memory.

Table 4.6. Addressing Modes for Data Instruction in the C251 Architecture

Mode	Address Range of Operand	Assembly Language Reference	Comments
Register	00:0000h-00:001Fh	R0-R15, WR0-WR30, DR0-DR28, DR56, DR60	R0-R7, WR0-WR6, and DR4 are in the register bank currently selected by the PSW and PSW1
Immediate 2 bits	N.A. (Operand is in the instruction)	#short = 1, 2, or 4	Used only in increment and decrement instructions
Immediate 8 bits	N.A. (Operand is in the instruction)	#data8 = #00h-#FFh	
Immediate 16 bits	N.A. (Operand is in the instruction)	#data16 = #0000h-#FFFFh	
Direct, 8 address bits	00:0000h-00:007Fh	dir8 = 00:0000h-00:007Fh	On-chip RAM
	SFRs	dir8 = S:080h-S:1FFh (2) or SFR mnemonic	SFR address
Direct, 16 address bits	00:0000h-00:FFFFh	dir16 = 00:0000h-00:FFFFh	
Indirect, 16 address bits	00:0000h-00:FFFFh	@WR0-@WR30	
Indirect, 24 address bits	00:0000h-FF:FFFFh	@DR0-@DR30, @DR56, @DR60	Upper 8 bits of DRk must be 00h
Displacement, 16 address bits	00:0000h-00:FFFFh	@WRj +dis16 = @WR0 +0h through @WR30 +FFFFh	Offset is signed; address wraps around in region 00:
Displacement, 24 address bits	00:0000h-FF:FFFFh	@DRk +dis24 = @DR0 +0h through @DR28 +FFFFh, @DR56 +(0h-FFFFh), @DR60 +(0h-FFFFh)	Offset is signed, upper 8 bits of DRk must 00h

Notes:

1. These registers are accessible in the memory space as well as in the register file.
2. The C251 Architecture supports SFRs in locations S:000h-S:1FFh.

4.1.5. Displacement Addressing

Several move instructions use displacement addressing to move bytes or words from a source to a destination. Sixteen–bit displacement addressing (@WRj+dis16) accesses indirectly the lowest 64 Kbytes in memory. The base address can be in any word register WRj. The instruction contains a 16–bit signed offset which is added to the base address. Only the lowest 16 bits of the sum are used to compute the operand address. If the sum of the base address and a positive offset exceeds FFFFh, the computed address wraps around within region 00: (e.g. F000h + 2005h becomes 1005h). Similarly, if the sum of the base address and a negative offset is less than zero, the computed address wraps around the top of region 00: (e.g., 2005h + F000h becomes 1005h).

24–bit displacement addressing (@DRk+dis24) accesses indirectly the entire 16–Mbyte address space. The base address must be in DR0, DR4, ..., DR24, DR28, DR56, or DR60. The upper byte in the dword register must be zero. The instruction contains a 16–bit signed offset which is added to the base address.

4.2. Arithmetic Instructions

The set of arithmetic instructions is greatly expanded in the C251 Architecture. The ADD and SUB instructions (see Table 5.19) operate on byte and word data that is accessed in several ways :

- as the contents of the accumulator, a byte register (Rn), or a word register (WRj)
- in the instruction itself (immediate data)
- in memory via direct or indirect addressing

The ADDC and SUBB instructions are the same as those for 80C51 microcontrollers.

The CMP (compare) instruction (see Table 5.20) calculates the difference of two bytes or words and then writes to flags CY, OV, AC, N, and Z in the PSW and PSW1 registers. The difference is not stored. The operands can be addressed in a variety of modes. The most frequent use of CMP is to compare data or addresses preceding a conditional jump instruction.

Table 5.21 lists the INC (increment) and DEC (decrement) instructions. The instructions for 80C51 microcontrollers are supplemented by instructions that can address byte, word, and dword registers and increment or decrement them by 1, 2, or 4 (denoted by #short). These instructions are supplied primarily for register–based address pointers and loop counters.

The C251 Architecture provides the MUL (multiply) and DIV (divide) instructions for unsigned 8–bit and 16–bit data (Table 5.22). Signed multiply and divide are left for the user to manage through a conversion process. The following operations are implemented :

- eight–bit multiplication: 8 bits x 8 bits → 16 bits
- sixteen–bit multiplication: 16 bits x 16 bits → 32 bits
- eight–bit division: 8 bits / 8 bits → 16 bits (8–bit quotient, 8–bit remainder)
- sixteen–bit division: 16 bits / 16 bits → 32 bits (16–bit quotient, 16–bit remainder)

These instructions operate on pairs of byte registers (Rmd,Rms), word registers (WRjd,WRjs), or the accumulator and B register (A, B). For 8–bit register multiplies, the result is stored in the word register that contains the first operand register. For example, the product from an instruction MUL R3,R8 is stored in WR2. Similarly, for 16–bit multiplies, the result is stored in the dword register that contains the first operand register. For example, the product from the instruction MUL WR6,WR18 is stored in DR4.

For 8–bit divides, the operands are byte registers. The result is stored in the word register that contains the first operand register. The quotient is stored in the lower byte, and the remainder is stored in the higher byte. A 16–bit divide is similar. The first operand is a word register, and the result is stored in the double word register that contains that word register. If the second operand (the divisor) is zero, the overflow flag (OV) is set and the other bits in PSW and PSW1 are meaningless.

4.3. Logical Instructions

The C251 Architecture provides a set of instructions that perform logical operations. The ANL, ORL, and XRL (logical AND, logical OR, and logical exclusive OR) instructions operate on bytes and words that are accessed via several addressing modes (see Table 5.23). A byte register, word register, or the accumulator can be logically combined with a register, im–mediate data, or data that is addressed directly or indirectly. These instructions affect the Z and N flags.

In addition to the CLR (clear), CPL (complement), SWAP (swap), and four rotate instructions that operate on the accumulator, TSC80251 microcontrollers have three shift commands for byte and word registers :

- SLL (Shift Left Logical) shifts the register one bit left and replaces the LSB with 0.
- SRL (Shift Right Logical) shifts the register one bit right and replaces the MSB with 0.
- SRA (Shift Right Arithmetic) shifts the register one bit right; the MSB is unchanged.

4.4. Data Transfer Instructions

Data transfer instructions copy data from one register or memory location to another. These instructions include the move instructions (see Table 5.24) and the exchange, PUSH, and pop instructions (see Table 5.24). Instructions that move only a single bit are listed with the other bit instructions in Table 5.26.

MOV (Move) is the most versatile instruction, and its addressing modes are expanded in the C251 Architecture. MOV can transfer a byte, word or dword between any two registers or between a register and any location in the address space.

The MOVX (Move External) instruction moves a byte from external memory to the accumulator or from the accumulator to memory. The external memory is in the region specified by DPXL, whose reset value is 01h.

The MOVC (Move Code) instruction moves a byte from code memory (region FF:) to the accumulator.

MOVS (Move with Sign Extension) and MOVZ (Move with Zero Extension) move the contents of an 8–bit register to the lower byte of a 16–bit register. The upper byte is filled with the sign bit (MOVS) or zeros (MOVZ). The MOVH (Move to high Word) instruction places 16–bit immediate data into the high word of a dword register.

The XCH (Exchange) instruction interchanges the contents of the accumulator with a register or memory location. The XCHD (Exchange Digit) instruction interchanges the lower nibble of the accumulator with the lower nibble of a byte in on–chip RAM. XCHD is useful for BCD (binary coded decimal) operations.

The PUSH and POP instructions facilitate storing information (PUSH) and then retrieving it (POP) in reverse order. PUSH can push a byte, a word or a dword onto the stack, using the immediate, direct or register addressing modes. POP can pop a byte or a word from the stack to a register or to memory.

5. Bit Instructions

A bit instruction addresses a specific bit in a memory location or SFR. There are four categories of bit instructions:

- SETB (Set Bit), CLR (Clear Bit), CPL (Complement Bit). These instructions can set, clear or complement any addressable bit.
- ANL (And Logical), ANL/ (And Logical Complement), ORL (OR Logical), ORL/ (Or Logical Complement). These instructions allow anding and oring of any addressable bit or its complement with the CY flag.
- MOV (Move) instructions transfer any addressable bit to the carry (CY) bit or vice versa.
- Bit–conditional jump instructions execute a jump if the bit has a specified state. The bit–conditional jump instructions are classified with the control instructions.

5.1. Bit Addressing

The bits that can be individually addressed are in the on–chip RAM and the SFRs (see Table 4.7.). The bit instructions that are unique to the C251 Architecture can address a wider range of bits than the instructions from the C51 Architecture.

There are some differences in the way the instructions from the two Architectures address bits. In the C51 Architecture, a bit (denoted by bit51) can be specified in terms of its location within a certain register, or it can be specified by a bit address in the range 00h-7Fh. The C251 Architecture does not have bit addresses as such. A bit can be addressed by name or by its location within a certain register, but not by a bit address.

Table 4.8. illustrates bit addressing in the two Architectures by using two sample bits:

- RAMBIT is bit 5 in RAMREG, which is location 23h. (“RAMBIT” and “RAMREG” are assumed to be defined in user code.)
- IT1 is bit 2 in TCON, which is an SFR at location 88h.

Table 4.7. Bit-addressable Locations

Architecture	Bit-addressable Locations	
	On-chip RAM	SFRs
C251 Architecture	20h-7Fh	All defined SFRs
C51 Architecture	20h-2Fh	SFRs with addresses ending in 0h or 8h: 80h, 88h, 90h, 98h, ..., F8h

Table 4.9. lists the addressing modes for bit insructions, and Table 5.26 summarizes the bit instructions. “bit” denotes a bit that is addressed by a new instruction in the C251 Architecture, and “bit51” denotes a bit that is addressed by an instruction in the C51 Architecture.

Table 4.8. Two Samples of Bits Addressing

Location	Addressing Mode	C51 Architecture	C251 Architecture
On-chip RAM	Register Name	RAMREG.5	RAMREG.5
	Register Address	23h.5	23h.5
	Bit Name	RAMBIT	RAMBIT
	Bit Address	1Dh	NA
SFR	Register Name	TCON.2	TCON.2
	Register Address	88.2h	S:88.2h
	Bit Name	IT1	IT1
	Bit Address	8A	NA

Table 4.9. Addressing Modes for Bit Instructions

Architecture	Variants	Bit Address	Memory/SFR Address	Comments
C251 (bit)	Memory	NA	20h.0-7Fh.7	
	SFR	NA	All defined SFRs	
C51 (bit)	Memory	00h-7Fh	20h.0-7Fh.7	
	SFR	80h-F8h	XXh.0-XXh.7, where XX = 80, 88, 90, 98, ..., F0, F8	SFRs are not defined at all bit-addressable locations

6. Control Instructions

Control instructions “instructions that change program flow” include calls, returns, and conditional and unconditional jumps (see Table 5.27). Instead of executing the next instruction in the queue, the processor executes a target instruction. The control instruction provides the address of a target instruction either implicitly, as in a return from a subroutine, or explicitly, in the form of a relative, direct, or indirect address.

TSC80251 microcontrollers have a 24–bit program counter (PC), which allows a target instruction to be anywhere in the 16–Mbyte address space. however, as discussed in this section, some control instructions restrict the target address to the current 2–Kbyte or 64–Kbyte address range by allowing only the lowest 11 or lowest 16 bits of the program counter to change.

6.1. Addressing Modes for Control Instructions

Table 4.10. lists the addressing modes for the control instructions.

- Relative addressing:
  The control instruction provides the target address as an 8–bit signed offset (rel) from the address of the next instruction.
- Direct addressing:
  The control instruction provides a target address, which can have 11 bits (addr11), 16 bits (addr16), or 24 bits (addr24). The target address is written to the PC.
  - addr11: Only the lower 11 bits of the PC are changed; i.e., the target address must be in the current 2–Kbyte block (the 2–Kbyte block that includes the first byte of the next instruction).
  - addr16: Only the lower 16 bits of the PC are changed; i.e., the target address must be in the current 64–Kbyte region (the 64–Kbyte region that includes the first byte of the next instruction).
  - addr24: The target address can be anywhere in the 16–Mbyte address space.
- Indirect addressing:
  There are two types of indirect addressing for control instructions:
  - For the instructions LCALL @WRj and LJMP @WRj, the target address is in the current 64–Kbyte region. The 16–bit address in WRj is placed in the lower 16 bits of the PC. The upper eight bits of the PC remain unchanged from the address of the next instruction.
  - For the instruction JMP @A+DPTR, the sum of the accumulator and DPTR is placed in the lower 16 bits of the PC, and the upper eight bits of the PC are FF:, which restricts the target address to the code memory space of the C51 Architecture.

Table 4.10. Addressing Modes for Control Instructions

Description	Address Bits Provided	Address Range
Relative, 8-bit relative address (rel)	8	–128 to +127 from first byte of next instruction
Direct, 11-bit target address (addr11)	11	Current 2 Kbytes
Direct, 16-bit target address (addr16)	16	Current 64 Kbytes
Direct, 24-bit target address (addr24) ★	24	00:0000h-FF:FFFFh
Indirect (@WRj) ★	16	Current 64 Kbytes
Indirect (@A +DPTR)	16	64-Kbyte region specified by DPXL (reset value = 01h)

Note:
★ *These modes are not used by instructions in the C51 Architecture.*

6.2. Conditional Jumps

The C251 Architecture supports bit–conditional jumps, compare–conditional jumps, and jumps based on the value of the accumulator. A bit–conditional jump is based on the state of a bit. In a compare–conditional jump, the jump is based on a comparison of two operands. All conditional jumps are relative, and the target address (rel) must be in the current 256–byte block of code. The instruction set includes three kinds of bit–conditional jumps :

- JB (Jump on Bit): Jump if the bit is set.
- JNB (Jump on Not Bit): Jump if the bit is clear.
- JBC (Jump on Bit then Clear it): Jump if the bit is set; then clear it.

Compare–conditional jumps test a condition resulting from a compare (CMP) instruction that is assumed to precede the jump instruction. The jump instruction examines the PSW and PSW1 registers and interprets their flags as though they were set or cleared by a compare (CMP) instruction. Actually, the state of each flag is determined by the last instruction that could have affected that flag.

The condition flags are used to test one of the following six relations between the operands :

- equal (=), not equal (≠)
- greater than (>), less than (<)
- greater than or equal (≥), less than or equal (≤)

For each relation there are two instructions, one for signed operands and one for unsigned operands (see Table 4.11.).

Table 4.11. Compare-conditional Jump Instructions

Operand Type	Relation					
	=	≠	>	<	≥	≤
Unsigned	JE	JNE	JG	JL	JGE	JLE
Signed			JSG	JSL	JSGE	JSLE

6.3. Unconditional Jumps

There are five unconditional jumps. NOP and SJMP jump to addresses relative to the program counter. AJMP, LJMP, and EJMP jump to direct or indirect addresses.

- NOP (No Operation) is an unconditional jump to the next instruction.
- SJMP (Short Jump) jumps to any instruction within –128 to 127 of the next instruction.
- AJMP (Absolute Jump) changes the lowest 11 bits of the PC to jump anywhere within the current 2–Kbyte block of memory. The address can be direct or indirect.
- LJMP (Long Jump) changes the lowest 16 bits of the PC to jump anywhere within the current 64–Kbyte region.
- EJMP (Extended Jump) changes all 24 bits of the PC to jump anywhere in the 16–Mbyte address space. The address can be direct or indirect.

6.4. Calls and Returns

The C251 Architecture provides relative, direct, and indirect calls and returns.

- ACALL (Absolute Call) pushes the lower 16 bits of the next instruction address onto the stack and then changes the lower 11 bits of the PC to the 11–bit address specified by the instruction. The call is to an address that is in the same 2–Kbyte block of memory as the address of the next instruction.
- LCALL (Long Call) pushes the lower 16 bits of the next–instruction address onto the stack and then changes the lower 16 bits of the PC to the 16–bit address specified by the instruction. The call is to an address in the same 64–Kbyte block of memory as the address of the next instruction.
- ECALL (Extended Call) pushes the 24 bits of the next instruction address onto the stack and then changes the 24 bits of the PC to the 24–bit address specified by the instruction. The call is to an address anywhere in the 16–Mbyte memory space.
- RET (Return) pops the top two bytes from the stack to return to the instruction following a subroutine call (ACALL or LCALL). The return address must be in the same 64–Kbyte region.
- ERET (Extended Return) pops the top three bytes from the stack to return to the address following a subroutine call (ECALL). The return address can be anywhere in the 16–Mbyte address space.
- RETI (Return from Interrupt) provides a return from an interrupt service routine. The operation of RETI depends on the INTR bit in the CONFIG1 configuration byte (see Product Design Guide):
  - For INTR = 0, an interrupt pushes the two lower bytes of the PC onto the stack in the following order : PC.7:0, PC.15:8. The RETI instruction pops these two bytes and uses them as the 16–bit return address in region FF:. RETI also restores the interrupt logic to accept additional interrupts at the same priority level as the one just processed.
  - For INTR = 1, an interrupt pushes the three PC bytes and PSW1 onto the stack in the following order: PSW1, PC.23:16, PC.7:0, PC.15:8. The RETI instruction pops these four bytes and then returns to the specified 24–bit address, which can be anywhere in the 16–Mbyte address space. RETI also clears the interrupt request line. (see the note in Table 4.10. regarding compatibility with code written for 80C51 microcontrollers.)

The TRAP instruction which causes a non maskable interrupt call is useful for the development of emulation of a TSC80251 microcontroller.

Note:
A simple RET instruction also returns execution to the interrupted program. In previous implementations this inappropriately allowed the system to operate as though an interrupt service routine is still in progress. The C251 Architecture allows use of both RETI and RET instructions for interrupt completion. However, for code expected to run properly on both 80C51 and TSC80C251 microcontrollers, only the execution of a RETI instruction is considered proper completion of the interrupt operation.

7. Interrupt Processing

7.1. Interrupt Request

Interrupt processing is a dynamic operation that begins when a source requests an interrupt and lasts until the execution of the first instruction in the interrupt service routine (see Figure 4.4.). Response time is the amount of time between the interrupt request and the resulting break in the current instruction stream. Latency is the amount of time between the interrupt request and the execution of the first instruction in the interrupt service routine. These periods are dynamic due to the presence of both fixed-time sequences and several variable conditions. These conditions contribute to total elapsed time.

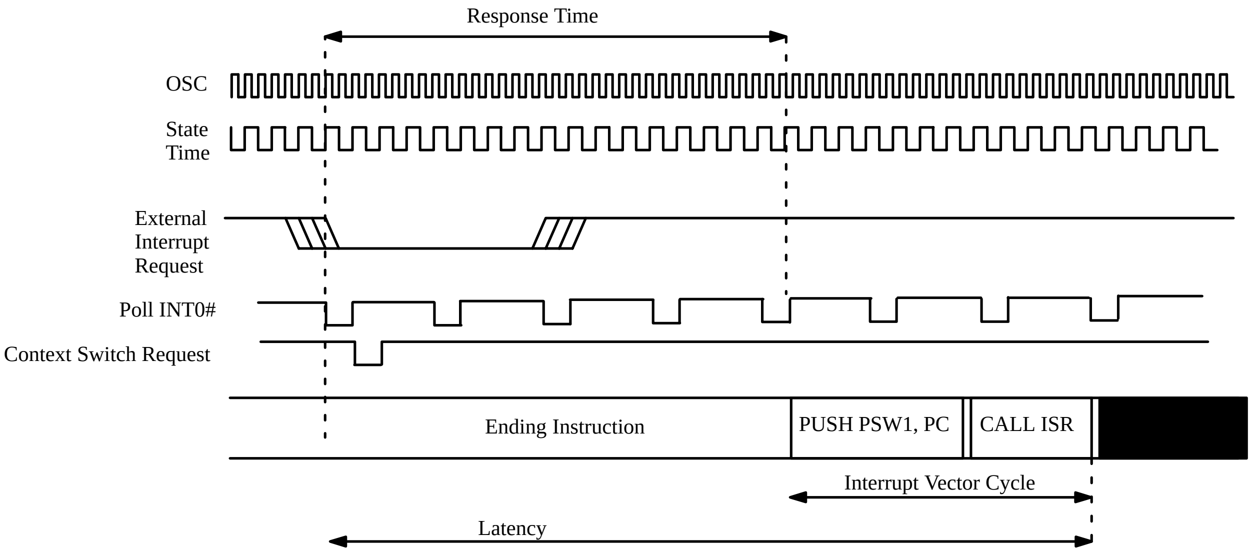


Figure 4.4. Interrupt Process

Both response time and latency begin with the interrupt request. The subsequent minimum fixed sequence comprises the interrupt sample, poll, context switch request and interrupt vector cycle operations. The variables consist of (but are not limited to): specific instruction in use at request time, internal versus external interrupt source requests, internal versus external program operation, stack location, presence of wait states, page-mode operation and call pointer length.

7.2. Blocking Conditions

If all enable and priority requirements have been met, a single prioritized interrupt request at a time generates a context switch and a vector cycle to an ISR. There are three causes of blocking conditions with hardware-generated vectors:

- An interrupt of equal or higher priority level is already in progress (defined as any point after the flag has been set and the RETI of the ISR has not executed).
- The current polling cycle is not the final cycle of the instruction in progress.
- The instruction in progress is RETI
- The instruction in progress is any write/read–modify–write to interrupt enable or interrupt priority level registers (see the Product Design Guide).

Any of these conditions blocks calls to ISR. Condition two ensures the instruction in progress completes before the system vectors to the ISR. Condition three ensures at least one more instruction of the interrupted routine executes before the system vectors to additional interrupts. Condition four insures interrupt requests are polled and prioritized consistently.

Note:
If the interrupt flag for a level-triggered external interrupt is set but denied for one of the above conditions and is clear when the blocking condition is removed, then the denied interrupt is ignored. In other words, blocked interrupt requests are not buffered for retention.
Furthermore, if several interrupts are pending, the interrupt actually served will be the one selected by the last polling cycle when the blocking condition disappears, hence blocking the other ones.

7.3. Interrupt Vector Cycle

When an interrupt vector cycle is initiated following a context switch request, the CPU breaks the instruction stream sequence, resolves all instruction pipeline decisions, and pushes multiple program counter (PC) bytes onto the stack. The CPU then reloads the PC with a start address for the appropriate ISR. The number of bytes pushed to the stack and the call pointer length depend upon the INTR bit in the configuration register (see CONFIG1 in the Product Design Guide). A processor status word (PSW1) may also be pushed to the stack according to the INTR bit.

7.4. Interrupt Service Routine

ISR execution proceeds until the RETI instruction is encountered. The RETI instruction informs the processor the interrupt routine is completed. It pops PC address bytes off the stack (as well as PSW1 for INTR = 1), and execution resumes at the suspended instruction stream.

With the exception of TRAP, the start addresses of consecutive interrupt service routines (ISR) are eight bytes apart. If consecutive interrupts are used (IE0 and TF0, for example, or TF0 and IE1), the first interrupt routine (if more than eight bytes long including RETI instruction) must execute a jump to some other memory location. This prevents overlap of the start address of the following interrupt routine but slightly increase the ISR overhead.

8. Interrupt Times

To have a system supporting heavy duty operation, the maximum latency has to be considered. Though the average performance rather depends on the average latency which is more difficult to predict. This section explains how to compute the maximum time and to estimate the average time.

8.1. Interrupt Response Time

Response time is defined as the start of a dynamic time period when a source requests an interrupt and lasts until a break in the current instruction execution stream occurs (see Figure 4.5.). Response time (and therefore latency) is affected by two primary factors: the incidence of the request relative to the four-state-time sample window and the completion time of instructions in the response period (i.e., shorter instructions complete earlier than longer instructions).

Note:
External interrupt signals require one additional state time in comparison to internal interrupts. This is necessary to sample and latch the pin value prior to a poll of the interrupts. The sample occurs in the first half of the state time and the poll/request occurs in the second half of the next state time. Therefore, this sample and poll/request portion of the minimum/maximum fixed response and latency time is two/five states for internal interrupts and three/six states for external interrupts. External interrupts should remain active for more than four state times to guarantee interrupt recognition when the request occurs immediately after a sample has been taken (i.e., requested in the second half of a sample state time).

If the external interrupt goes active one state after the poll state, the interrupt is not resampled and polled for another three states. After the second sample is taken and the interrupt request is recognized, the interrupt controller requests the interrupt vector cycle. The programmer must also consider the time to complete the instruction at the moment the context switch request is sent to the execution unit. If 9 states of a 10-state instruction have completed when the context switch is requested, the total response time is 6 states, with an interrupt vector cycle immediately after the final state of the 10-state instruction (see Figure 4.5.).

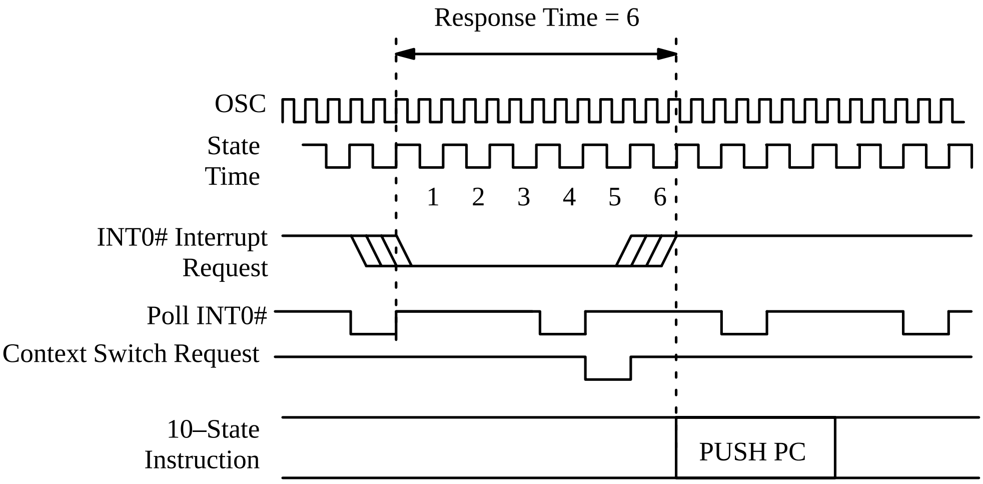


Figure 4.5. Response Time Example 1

Conversely, if the external interrupt requests service in the state just prior to the next sample, response is much quicker. One state asserts the interrupt request, one state samples, and one state requests the context switch. If at that point the same instruction conditions exist, one additional state time is needed to complete the 10-state instruction prior to the interrupt vector cycle (see Figure 4.6.). The total response time in this case is four state times. The programmer must evaluate all pertinent conditions for accurate predictability.

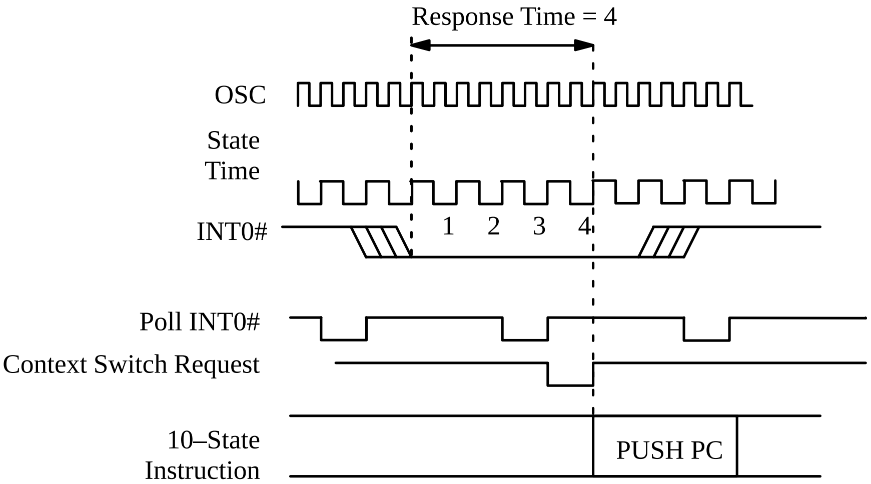


Figure 4.6. Response Time Example 2

8.2. Interrupt Latency Time

8.2.1. Minimum Fixed Interrupt Time

Each interrupt is sampled and polled every four state times (see Figure 4.5.). One additional state time is required for a context switch request. For code branches to jump locations in the current 64-Kbyte memory region (compatible with C51 Architecture), the interrupt vector cycle time is 11 states. Therefore, the minimum fixed poll and request time is 13 states (1 poll states + 1 request state + 11 states for the interrupt vector cycle = 13 state times).

Therefore, this minimum fixed period rests upon five assumptions:

- The interrupt request is coincident with its polling cycle.
- The source request is an internal interrupt with high enough priority to take precedence over other potential interrupts.
- The context switch request is coincident with internal execution and needs no instruction completion time before the interrupt vector cycle.
- The program uses an internal stack location.
- The ISR is in on-chip code memory.

8.2.2. Worst Case Latency Variables

Worst-case latency calculations assume that the longest C251 Architecture instruction used in the program must fully execute prior to a context switch. The delay from instruction completion time is reduced by one state with the given assumption that the first instruction state overlaps the context switch request state (therefore, 16-bit DIV is 21–1 = 20 states for latency calculations). The calculations add fixed and variable interrupt times (see Table 4.12.) to this instruction time to predict latency. The worst-case latency (both fixed and variable times included) is expressed by a pseudo-formula:

FIXED_TIME + VARIABLES + LONGEST_INSTRUCTION = MAXIMUM LATENCY PREDICTION

Table 4.12. Interrupt Latency Variables

Variable	Polling Time	External Interrupt	>64K Jump to ISR (1	External Execution (2)	External Stack 2–byte push (3)	External Stack 4–byte push (3)
Number of States Added	0 to 3	1	8	N1+2	2 × (N2+2)	4 × (N2+2)

Notes:
1. <64K / >64K means inside/outside the 64-Kbyte memory region where code is executing.
2. N1 is the number of wait states for external code fetch, add the number states to fetch possible additional bytes and complete the first instruction according to the information provided in SECTION 5.2.
3. N2 is the number of wait states for external stack accesses.

8.2.3. Latency Calculations

Assume the use of a zero-wait-state external memory where current instructions, the ISR and the stack are located within the same 64-Kbyte memory region (compatible with memory maps for 80C51 Microcontrollers). Further assume INT0# has made the request one state prior to the poll state. Also assume there are seven states yet to complete in the current 21-state DIV instruction when INT0# requests service. As shown in Figure 4.7. , the completion of the current instruction is the limiting factor for this assumption. The actual response time is seven states while the best case response time is two states for internal interrupts with one more state for external interrupts.

Latency calculations begin with the minimum fixed latency of 13 states: two states best case response time and 11 states best case context switch time. From Table 4.12. , one state is added for an INT0# request from external hardware; two states are added for external execution; and four states for an external stack with 2–byte push (64–Kbyte pointers). Three states are further added for the current instruction to complete. Finally one state is added for the interrupt request has been made one state before the poll state. The actual latency is 24 states. Maximum latency calculations predict 43 states for this example due to inclusion of total DIV instruction time (less one state). Average latency calculations estimate 23 states, assuming an average execution time of three states per instruction in the interruptible routines: the average completion time is half of the average execution time. Minimum latency calculations predict 20 states when there is no delay for polling or instruction completion.

Table 4.13. Actual vs. Predicted Latency Calculations

Latency Factors	Actual	Minimum	Maximum	Average
Base Case Minimum Fixed Time	13	13	13	13
INT0# External Request	1	1	1	1
External Execution	2	2	2	2
External Stack 2–byte Push	4	4	4	4
Completion Time for Current (DIV instruction)	3	0	20	1.5
Polling Time	1	0	3	1.5
TOTAL	24	20	43	23

Note:
This computation does not include the possible additional states to actually complete the first instruction of the ISR.
It further assumes the average execution time is three states per instruction for the interrupted routines.

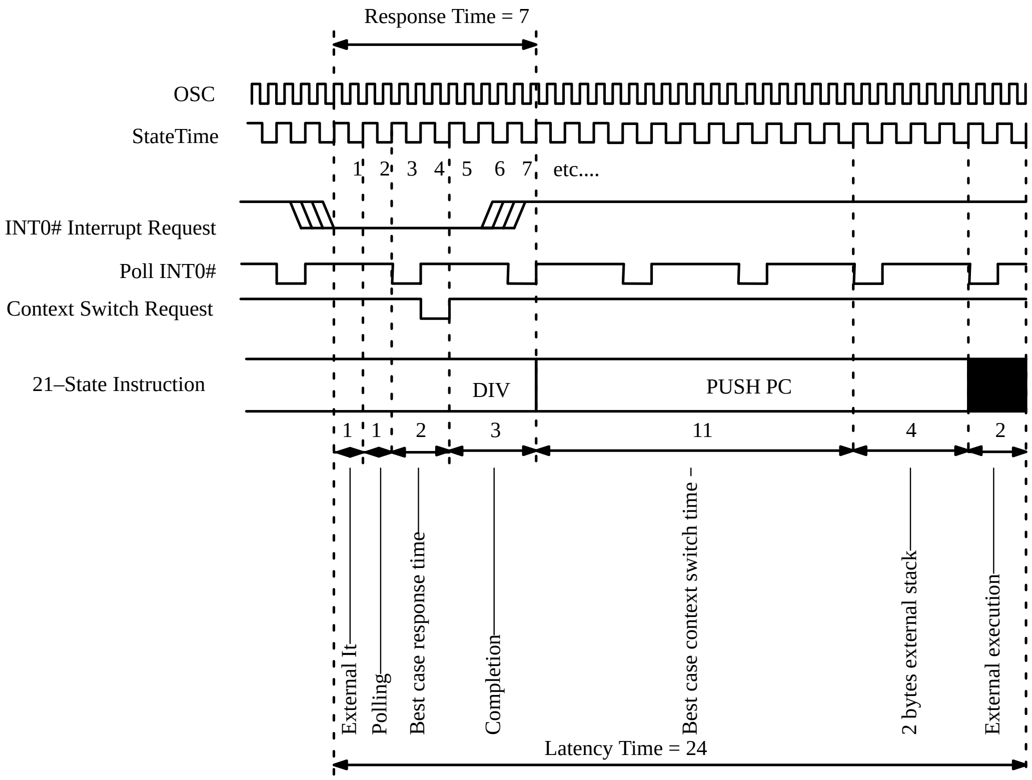


Figure 4.7. Latency Time Example

PSW (S:D0h)
Program Status Word register

CY	AC	FO	RS1	RS0	OV	UD	P
7	**6**	**5**	**4**	**3**	**2**	**1**	**0**

Bit Number	Bit Mnemonic	Description
7	CY	**Carry flag** The carry flag is set by an addition (ADD, ADDC) if there is a carry out of the MSB. It is set by a subtraction (SUB, SUBB) or compare (CMP) if a borrow is needed for the MSB. The carry flag is also affected by some rotate and shift instructions, logical bit instructions and bit move instructions, and the multiply (MUL) and decimal adjust (DA) instructions (see Table 4.4.).
6	AC	**Auxiliary Carry flag** The auxiliary flag is affected only by instructions that address 8-bit operands. The AC flag is set if an arithmetic instruction with an 8-bit operand produces a carry out of bit 3 (from addition) or a borrow into bit 3 (from subtraction). Otherwise it is cleared. This flag is useful for BCD arithmetic (see Table 4.4.).
5	FO	**Flag 0** This general-purpose flag is available to the user.
4	RS1	**Register Bank Select bit 1** This bit selects the memory locations that comprise the active bank of the register file (registers R0-R7). RS1 Bank Address 0 0 00h-07h 0 1 08h-0Fh 1 2 10h-17h 1 3 18h-1Fh
3	RS0	**Register Bank Select bit 0** This bit selects the memory locations that comprise the active bank of the register file (registers R0-R7). RS0 Bank Address 0 0 00h-07h 1 1 08h-0Fh 0 2 10h-17h 1 3 18h-1Fh
2	OV	**Overflow flag** This bit is set if an addition or subtraction of signed variables results in an overflow error (i.e., if the magnitude of the sum or differnecce is too great for the seven LSBs in 2's-complement representation). The overflow flag is also set if a multiplication product overflows one byte or if a division by zero is attempted.
1	UD	**User-definable flag** This general-purpose flag is available to the user.
0	P	**Parity bit** This bit indicates the parity of the accumulator. It is set if an odd number of bits in the accumulator are set. Otherwise, it is cleared. Not all instructions update the parity bit.

Reset Value = 0000 0000b

Figure 4.8. Program Status Word register (PSW)

PSW1 (S:D1h)
Program Status Word 1 register

CY	AC	N	RS1	RS0	OV	Z	–
7	**6**	**5**	**4**	**3**	**2**	**1**	**0**

Bit Number	Bit Mnemonic	Description
7	CY	**Carry flag** Identical to the CY bit in the PSW register (see Figure 4.8.).
6	AC	**Auxiliary Carry flag** Identical to the AC bit in the PSW register (see Figure 4.8.).
5	N	**Negative flag** This bit is set if the result of the last logical or arithmetic operation was negative, i.e., bit15 = 1. Otherwise it is cleared.
4	RS1	**Register Bank Select bit 1** Identical to the RS1 bit in the PSW register (see Figure 4.8.).
3	RS0	**Register Bank Select bit 0** Identical to the RS0 bit in the PSW register (see Figure 4.8.).
2	OV	**Overflow flag** Identical to the OV bit in the PSW register (see Figure 4.8.).
1	Z	**Zero flag** This flag is set if the result of the last logical or arithmetic operation is zero. Otherwise it is cleared.
0	–	**Reserved** The value read from this bit is indeterminate. Do not set this bit.

Reset Value = 0000 0000b

Figure 4.9. Program Status Word 1 register (PSW1)

Instruction Set Reference

This chapter contains reference material for the instructions in the C251 Architecture. It includes an opcode map, a summary of the instructions –with instruction lengths and execution times– and a detailed description of each instruction. It contains the following tables:

Table 1through Table 5describe the notation used for the instruction operands.

Table 6bounds the minimum number of states per instruction.

Table 22and Table 23comprise the opcode map for the instruction set.

The following tables list the instructions with their lengths in bytes and their execution times:

- Add and Subtract Instructions, Table 7
- Increment and Decrement Instructions, Table 8
- Compare Instructions, Table 9
- Logical Instructions, Table 10and Table 11
- Multiply, Divide and Decimal-adjust Instructions, Table 12
- Move Instructions, Table 13to Table 15
- Bit Instructions, Table 16
- Exchange, Push and Pop Instructions, Table 17
- Control Instructions, Table 18to Table 21

Table 24through Table 33contain supporting material for the opcode map.

Notes:

1. The instruction execution times given in this appendix are for code executing from on-chip code memory and for data that is read from and written to on-chip RAM. Execution times are increased by executing code from external memory, accessing peripheral SFRs, accessing data in external memory, using a wait state, or extending the ALE pulse.

2. For some instructions, accessing the Port SFRs, Px, x = 0-3, increases the execution time.

9. Notation for Instruction Operands

Table 1 to Table 5 provide Notation for Instruction Operands.

Table 1Notation for Direct Addressing

Direct Address	Description	C251	C51
dir8	A direct 8-bit address. This can be a memory address (00h-7Fh) or a SFR address (80h-FFh). It is a byte (default), word or double word depending on the other operand.	✓	✓
dir16	A 16-bit memory address (00:0000h-00:FFFFh) used in direct addressing.	✓	

Table 2Notation for Immediate Addressing

Immediate Address	Description	C251	C51
#data	An 8-bit constant that is immediately addressed in an instruction.	✓	✓
#data16	A 16-bit constant that is immediately addressed in an instruction.	✓	
#0data16 #1data16	A 32-bit constant that is immediately addressed in an instruction. The upper word is filled with zeros (#0data16) or ones (#1data16).	✓	
#short vv	A constant, equal to 1, 2, or 4, that is immediately addressed in an instruction. Binary representation of #short (’00’ is 1, ’01’ is 2, ’10’ is 4 and ’11’ is reserved).	✓	

Table 3Notation for Bit Addressing

Direct Address	Description	C251	C51
bit51	A directly addressed bit (bit number= 00h-FFh) in memory or an SFR. Bits 00h-7Fh are the 128 bits in byte locations 20h-2Fh in the on-chip RAM. Bits 80h-FFh are the 128 bits in the 16 SFRs with addresses that end in 0h or 8h, S:80h, S:88h, S:90h,..., S:F0h, S:F8h.		✓
bit yyy	A directly addressed bit in memory locations 00:0020h-00:007Fh or in any defined SFR. Binary representation of a bit number (0–7) whitin a byte.	✓	

Table 4Notation for Destination in Control Instructions

Direct Address	Description	C251	C51
rel	A signed (two’s complement) 8-bit relative address. The destination is –128 to +127 bytes relative to the next instruction’s first byte.	✓	✓
addr11	An 11-bit target address. The target is in the same 2-Kbyte block of memory as the next instruction’s first byte.		✓
addr16	A 16-bit target address. The target can be anywhere within the same 64-Kbyte region as the next instruction’s first byte.		✓
addr24	A 24-bit target address. The target can be anywhere within the 16–Mbyte address space.	✓	

Table 5Notation for Register Operands

Register	Description	C251	C51
@Ri	A memory location (00h-FFh) addressed indirectly via byte registers R0 or R1.		✓
Rn n rrr	Byte register R0-R7 of the currently selected register bank. Byte register index: n= 0-7. Binary representation of byte register index n.		✓
Rm Rmd Rms m, md, ms ssss SSSS	Byte register R0-R15 of the currently selected register file. Destination byte register. Source byte register. Byte register index: m, md, ms= 0-15. Binary representation of byte register index m or md. Binary representation of byte register index ms.	✓	
WRj WRjd WRjs @WRj @WRj +dis16 j, jd, js tttt TTTT	Word register WR0, WR2, ..., WR30 of the currently selected register file. Destination word register. Source word register. A memory location (00:0000h-00:FFFFh) addressed indirectly through word register WR0-WR30, is the target address for jump instructions. A memory location (00:0000h-00:FFFFh) addressed indirectly through word register (WR0-WR30) + 16–bit signed (two’s complement) displacement value. Word register index: j, jd, js= 0-30. Binary representation of word register index j/2 or jd/2. Binary representation of word register index js/2.	✓	
DRk DRkd DRks @DRk @DRk +dis16 k, kd, ks uuuu UUUU	Dword register DR0, DR4, ..., DR28, DR56, DR60 of the currently selected register file. Destination dword register. Source dword register. A memory location (00:0000h-FF:FFFFh) addressed indirectly through dword register DR0-DR28, DR56 and DR60, is the target address for jump instruction. A memory location (00:0000h-FF:FFFFh) addressed indirectly through dword register (DR0-DR28, DR56, DR60) + 16–bit (two’s complement) signed displacement value. Dword register index: k, kd, ks= 0, 4, 8..., 28, 56, 60. Binary representation of dword register index k/2 or kd/2. Binary representation of dword register index ks/2.	✓	

Table 5.1. defines the symbols (–, ✓,1, 0, ?) used to indicate the effect of the instruction on the flags in the PSW and PSW1 registers. For a conditional jump instruction, “!” indicates that a flag influences the decision to jump.

Table 5.1. Flag Symbols

Symbol	Description
–	The instruction does not modify the flag.
✓	The instruction sets or clears the flag, as appropriate.
1	The instruction sets the flag.
0	The instruction clears the flag.
?	The instruction leaves the flag in an indeterminate state.
!	For a conditional jump instruction: the state of the flag before the instruction executes influences the decision to jump or not jump.

10. Instruction Set Summary

This section contains tables that summarize the instruction set. For each instruction there is a short description, its length in bytes, and its execution time in states (one state time is equal to two system clock cycles). There are two concurrent processes limiting the effective instruction throughput:

- Instruction Fetch
- Instruction Execution

Table 7to Table 21assume code executing from on–chip memory, then the CPU is fetching 16–bit at a time and this is never limiting the execution speed.

If the code is fetched from external memory, a pre–fetch queue will store instructions ahead of execution to optimize the memory bandwidth usage when slower instructions are executed. However, the effective speed may be limited depending on the average size of instructions (for the considered section of the program flow). The maximum average instruction throughput is provided by Table 6depending on the external memory configuration (from Page Mode without wait state to Non–Page Mode with the maximum number of wait states). If the average size of instructions is not an integer, the maximum effective throughput is found by pondering the number of states for the neighbor integer values.

Note:

For instructions addressing an I/O Port (Px, x= 0-3), the pre–fetch process is disturbed and some wait states are added as highlighted by footnotes in Table 7to Table 21Adding the corresponding number of wait states to the actual lenght of each of these instructions provides the equivalent average instruction sizes to account for the pre–fetch disturbance.

Table 6Minimum Number of States per Instruction for given Average Sizes

Average size of Instructions (bytes)	Page Mode (states)	Non–Page Mode (states)				
		0 Wait State	1 Wait State	2 Wait States	3 Wait States	4 Wait States
1	1	2	3	4	5	6
2	2	4	6	8	10	12
3	3	6	9	12	15	18
4	4	8	12	16	20	24
5	5	10	15	20	25	30

If the average execution time of the considered instructions is larger than the number of states given by Table 6, this larger value will prevail as the limiting factor. Otherwise, the value from Table 6 must be taken. This is providing a fair estimation of the execution speed but only the actual code execution can provide the final value.

10.1. Size and Execution Time for Instruction Families

Table 7Summary of Add and Subtract Instructions

Add	ADD <dest>, <src>	dest opnd ← dest opnd + src opnd
Subtract	SUB <dest>, <src>	dest opnd ← dest opnd – src opnd
Add with Carry	ADDC <dest>, <src>	(A) ← (A) + src opnd + (CY)
Subtract with Borrow	SUBB <dest>, <src>	(A) ← (A) – src opnd – (CY)

Mnemonic	<dest>, <src>[1]	Comments	Binary Mode		Source Mode	
			Bytes	States	Bytes	States
ADD	A, Rn	Register to ACC	1	1	2	2
	A, dir8	Direct address to ACC	2	1[2]	2	1[2]
	A, @Ri	Indirect address to ACC	1	2	2	3
	A, #data	Immediate data to ACC	2	1	2	1
ADD / SUB	Rmd, Rms	Byte register to/from byte register	3	2	2	1
	WRjd, WRjs	Word register to/from word register	3	3	2	2
	DRkd, DRks	Dword register to/from dword register	3	5	2	4
	Rm, #data	Immediate 8-bit data to/from byte register	4	3	3	2
	WRj, #data16	Immediate 16-bit data to/from word register	5	4	4	3
	DRk, #0data16	16-bit unsigned immediate data to/from dword register	5	6	4	5
	Rm, dir8	Direct address (on–chip RAM or SFR) to/from byte register	4	3[2]	3	2[2]
	WRj, dir8	Direct address (on–chip RAM or SFR) to/from word register	4	4	3	3
	Rm, dir16	Direct address (64K) to/from byte register	5	3[3]	4	2[3]
	WRj, dir16	Direct address (64K) to/from word register	5	4[4]	4	3[4]
	Rm, @WRj	Indirect address (64K) to/from byte register	4	3[3]	3	2[3]
	Rm, @DRk	Indirect address (16M) to/from byte register	4	4[3]	3	3[3]
ADDC / SUBB	A, Rn	Register to/from ACC with carry	1	1	2	2
	A, dir8	Direct address (on–chip RAM or SFR) to/from ACC with carry	2	1[2]	2	1[2]
	A, @Ri	Indirect address to/from ACC with carry	1	2	2	3
	A, #data	Immediate data to/from ACC with carry	2	1	2	1

Notes:

1. *A shaded cell denotes an instruction in the C51 Architecture.*
2. *If this instruction addresses an I/O Port (Px, x= 0-3), add 1 to the number of states. Add 2 if it addresses a Peripheral SFR.*
3. *If this instruction addresses external memory location, add N+2 to the number of states (N: number of wait states).*
4. *If this instruction addresses external memory location, add 2(N+2) to the number of states (N: number of wait states).*

Table 8Summary of Increment and Decrement Instructions

Increment	INC <dest>	dest opnd ← dest opnd + 1
Increment	INC <dest>, <src>	dest opnd ← dest opnd + src opnd
Decrement	DEC <dest>	dest opnd ← dest opnd – 1
Decrement	DEC <dest>, <src>	dest opnd ← dest opnd – src opnd

Mnemonic	<dest>, <src>(1)	Comments	Binary Mode		Source Mode	
			Bytes	States	Bytes	States
INC DEC	A	ACC by 1	1	1	1	1
	Rn	Register by 1	1	1	2	2
	dir8	Direct address (on–chip RAM or SFR) by 1	2	2(2)	2	2(2)
	@Ri	Indirect address by 1	1	3	2	4
INC DEC	Rm, #short	Byte register by 1, 2, or 4	3	2	2	1
	WRj, #short	Word register by 1, 2, or 4	3	2	2	1
INC	DRk, #short	Double word register by 1, 2, or 4	3	4	2	3
DEC	DRk, #short	Double word register by 1, 2, or 4	3	5	2	4
INC	DPTR	Data pointer by 1	1	1	1	1

Notes:

1. A shaded cell denotes an instruction in the C51 Architecture.

2. If this instruction addresses an I/O Port (Px, x= 0-3), add 2 to the number of states. Add 3 if it addresses a Peripheral SFR.

Table 9Summary of Compare Instructions

Compare	CMP <dest>, <src>	dest opnd – src opnd

Mnemonic	<dest>, <src>(1)	Comments	Binary Mode		Source Mode	
			Bytes	States	Bytes	States
CMP	Rmd, Rms	Register with register	3	2	2	1
	WRjd, WRjs	Word register with word register	3	3	2	2
	DRkd, DRks	Dword register with dword register	3	5	2	4
	Rm, #data	Register with immediate data	4	3	3	2
	WRj, #data16	Word register with immediate 16-bit data	5	4	4	3
	DRk, #0data16	Dword register with zero-extended 16-bit immediate data	5	6	4	5
	DRk, #1data16	Dword register with one-extended 16-bit immediate data	5	6	4	5
	Rm, dir8	Direct address (on–chip RAM or SFR) with byte register	4	3(1)	3	2(1)
	WRj, dir8	Direct address (on–chip RAM or SFR) with word register	4	4	3	3
	Rm, dir16	Direct address (64K) with byte register	5	3(2)	4	2(2)
	WRj, dir16	Direct address (64K) with word register	5	4(3)	4	3(3)
	Rm, @WRj	Indirect address (64K) with byte register	4	3(2)	3	2(2)
	Rm, @DRk	Indirect address (16M) with byte register	4	4(2)	3	3(2)

Notes:

1. If this instruction addresses an I/O Port (Px, x= 0-3), add 1 to the number of states. Add 2 if it addresses a Peripheral SFR.

2. If this instruction addresses external memory location, add N+2 to the number of states (N: number of wait states).

3. If this instruction addresses external memory location, add 2(N+2) to the number of states (N: number of wait states).

Table 10Summary of Logical Instructions (1/2)

Logical AND[1]	ANL <dest>, <src>	dest opnd ← dest opnd Λ src opnd
Logical OR[1]	ORL <dest>, <src>	dest opnd ← dest opnd V src opnd
Logical Exclusive OR[1]	XRL <dest>, <src>	dest opnd ← dest opnd ∀ src opnd
Clear[1]	CLR A	(A) ← 0
Complement[1]	CPL A	(A) ← ∅ (A)
Rotate Left	RL A	$(A)_{n+1} \leftarrow (A)_n$, n= 0..6 $(A)_0 \leftarrow (A)_7$
Rotate Left Carry	RLC A	$(A)_{n+1} \leftarrow (A)_n$, n= 0..6 $(CY) \leftarrow (A)_7$ $(A)_0 \leftarrow (CY)$
Rotate Right	RR A	$(A)_{n-1} \leftarrow (A)_n$, n= 7..1 $(A)_7 \leftarrow (A)_0$
Rotate Right Carry	RRC A	$(A)_{n-1} \leftarrow (A)_n$, n= 7..1 $(CY) \leftarrow (A)_0$ $(A)_7 \leftarrow (CY)$

Mnemonic	<dest>, <src>[2]	Comments	Binary Mode		Source Mode	
			Bytes	States	Bytes	States
ANL ORL XRL	A, Rn	register to ACC	1	1	2	2
	A, dir8	Direct address (on–chip RAM or SFR) to ACC	2	1[3]	2	1[3]
	A, @Ri	Indirect address to ACC	1	2	2	3
	A, #data	Immediate data to ACC	2	1	2	1
	dir8, A	ACC to direct address	2	2[4]	2	2[4]
	dir8, #data	Immediate 8–bit data to direct address	3	3[4]	3	3[4]
	Rmd, Rms	Byte register to byte register	3	2	2	1
	WRjd, WRjs	Word register to word register	3	3	2	2
	Rm, #data	Immediate 8-bit data to byte register	4	3	3	2
	WRj, #data16	Immediate 16-bit data to word register	5	4	4	3
	Rm, dir8	Direct address to byte register	4	3[3]	3	2[3]
	WRj, dir8	Direct address to word register	4	4	3	3
	Rm, dir16	Direct address (64K) to byte register	5	3[5]	4	2[5]
	WRj, dir16	Direct address (64K) to word register	5	4[6]	4	3[6]
	Rm, @WRj	Indirect address (64K) to byte register	4	3[5]	3	2[5]
	Rm, @DRk	Indirect address (16M) to byte register	4	4[5]	3	3[5]
CLR	A	Clear ACC	1	1	1	1
CPL	A	Complement ACC	1	1	1	1
RL	A	Rotate ACC left	1	1	1	1
RLC	A	Rotate ACC left through CY	1	1	1	1
RR	A	Rotate ACC right	1	1	1	1
RRC	A	Rotate ACC right through CY	1	1	1	1

Notes:

1. *Logical instructions that affect a bit are in Table 16.*
2. *A shaded cell denotes an instruction in the C51 Architecture.*
3. *If this instruction addresses an I/O Port (Px, x= 0–3), add 1 to the number of states. Add 2 if it addresses a Peripheral SFR.*
4. *If this instruction addresses an I/O Port (Px, x= 0–3), add 2 to the number of states. Add 3 if it addresses a Peripheral SFR.*
5. *If this instruction addresses external memory location, add N+2 to the number of states (N: number of wait states).*
6. *If this instruction addresses external memory location, add 2(N+2) to the number of states (N: number of wait states).*

Table 11Summary of Logical Instructions (2/2)

Shift Left Logical	SLL <dest>	<dest>$_{0}$ ← 0 <dest>$_{n+1}$ ← <dest>$_{n}$, n= 0..msb–1 (CY) ← <dest>$_{msb}$
Shift Right Arithmetic	SRA <dest>	<dest>$_{msb}$ ← <dest>$_{msb}$ <dest>$_{n-1}$ ← <dest>$_{n}$, n= msb..1 (CY) ← <dest>$_{0}$
Shift Right Logical	SRL <dest>	<dest>$_{msb}$ ← 0 <dest>$_{n-1}$ ← <dest>$_{n}$, n= msb..1 (CY) ← <dest>$_{0}$
Swap	SWAP A	A$_{3:0}$ ↔ A$_{7:4}$

Mnemonic	<dest>, <src>[(1)]	Comments	Binary Mode		Source Mode	
			Bytes	States	Bytes	States
SLL	Rm	Shift byte register left through the MSB	3	2	2	1
	WRj	Shift word register left through the MSB	3	2	2	1
SRA	Rm	Shift byte register right	3	2	2	1
	WRj	Shift word register right	3	2	2	1
SRL	Rm	Shift byte register left	3	2	2	1
	WRj	Shift word register left	3	2	2	1
SWAP	A	Swap nibbles within ACC	1	2	1	2

Note:

1. *A shaded cell denotes an instruction in the C51 Architecture.*

Table 12Summary of Multiply, Divide and Decimal-adjust Instructions

Multiply	MUL AB MUL <dest>, <src>	(B:A) ← (A)×(B) extended dest opnd ← dest opnd × src opnd
Divide	DIV AB	(A) ← Quotient ((A)/(B)) (B) ← Remainder ((A)/(B))
Divide	DIV <dest>, <src>	ext. dest opnd high ← Quotient (dest opnd/src opnd) ext. dest opnd low ← Remainder (dest opnd/src opnd)

Decimal-adjust ACC
for Addition (BCD)
	DA A	IF [[(A)$_{3:0}$ > 9] ∨ [(AC)= 1]]
		THEN (A)$_{3:0}$ ← (A)$_{3:0}$ + 6 !affects CY;
		IF [[(A)$_{7:4}$ > 9] ∨ [(CY)= 1]]
		THEN (A)$_{7:4}$ ← (A)$_{7:4}$ + 6

Mnemonic	<dest>, <src>[(1)]	Comments	Binary Mode		Source Mode	
			Bytes	States	Bytes	States
MUL	AB	Multiply A and B	1	5	1	5
	Rmd, Rms	Multiply byte register and byte register	3	6	2	5
	WRjd, WRjs	Multiply word register and word register	3	12	2	11
DIV	AB	Divide A and B	1	10	1	10
	Rmd, Rms	Divide byte register and byte register	3	11	2	10
	WRjd, WRjs	Divide word register and word register	3	21	2	20
DA	A	Decimal adjust ACC	1	1	1	1

Note:

1. *A shaded cell denotes an instruction in the C51 Architecture.*

Table 13Summary of Move Instructions (1/3)

Move to High word	MOVH <dest>, <src>	dest $\text{opnd}_{31:16}$ ← src opnd
Move with Sign extension	MOVS <dest>, <src>	dest opnd ← src opnd with sign extend
Move with Zero extension	MOVZ <dest>, <src>	dest opnd ← src opnd with zero extend
Move Code	MOVC A, <src>	(A) ← src opnd
Move eXtended	MOVX <dest>, <src>	dest opnd ← src opnd

Mnemonic	<dest>, <src>(1)	Comments	Binary Mode		Source Mode	
			Bytes	States	Bytes	States
MOVH	DRk, #data16	16-bit immediate data into upper word of dword register	5	3	4	2
MOVS	WRj, Rm	Byte register to word register with sign extension	3	2	2	1
MOVZ	WRj, Rm	Byte register to word register with zeros extension	3	2	2	1
MOVC	A, @A +DPTR	Code byte relative to DPTR to ACC	1	6(3)	1	6(3)
	A, @A +PC	Code byte relative to PC to ACC	1	6(3)	1	6(3)
MOVX	A, @Ri	Extended memory (8-bit address) to ACC(2)	1	4	1	5
	A, @DPTR	Extended memory (16-bit address) to ACC(2)	1	3(4)	1	3(4)
	@Ri, A	ACC to extended memory (8-bit address)(2)	1	4	1	4
	@DPTR, A	ACC to extended memory (16-bit address)(2)	1	4(3)	1	4(3)

Notes:

1. *A shaded cell denotes an instruction in the C51 Architecture.*
2. *Extended memory addressed is in the region specified by DPXL (reset value= 01h).*
3. *If this instruction addresses external memory location, add N+1 to the number of states (N: number of wait states).*
4. *If this instruction addresses external memory location, add N+2 to the number of states (N: number of wait states).*

Table 14Summary of Move Instructions (2/3)

Move(1)	MOV <dest>, <src>	dest opnd ← src opnd

Mnemonic	<dest>, <src>(2)	Comments	Binary Mode		Source Mode	
			Bytes	States	Bytes	States
MOV	A, Rn	Register to ACC	1	1	2	2
	A, dir8	Direct address (on–chip RAM or SFR) to ACC	2	1(3)	2	1(3)
	A, @Ri	Indirect address to ACC	1	2	2	3
	A, #data	Immediate data to ACC	2	1	2	1
	Rn, A	ACC to register	1	1	2	2
	Rn, dir8	Direct address (on–chip RAM or SFR) to register	2	1(3)	3	2(3)
	Rn, #data	Immediate data to register	2	1	3	2
	dir8, A	ACC to direct address	2	2(3)	2	2(3)
	dir8, Rn	Register to direct address	2	2(3)	3	3(3)
	dir8, dir8	Direct address to direct address	3	3(4)	3	3(4)
	dir8, @Ri	Indirect address to direct address	2	3(3)	3	4(3)
	dir8, #data	Immediate data to direct address	3	3(3)	3	3(3)
	@Ri, A	ACC to indirect address	1	3	2	4
	@Ri, dir8	Direct address to indirect address	2	3(3)	3	4(3)
	@Ri, #data	Immediate data to indirect address	2	3	3	4
	DPTR, #data16	Load Data Pointer with a 16-bit constant	3	2	3	2

Notes:

1. *Instructions that move bits are in Table 16.*
2. *Move instructions from the C51 Architecture.*
3. *If this instruction addresses an I/O Port (Px, x= 0-3), add 1 to the number of states. Add 2 if it addresses a Peripheral SFR.*
4. *Apply note 3 for each dir8 operand.*

Table 15Summary of Move Instructions (3/3)

Move[1]	MOV <dest>, <src>	dest opnd ← src opnd				
Mnemonic	**<dest>, <src>[2]**	**Comments**	**Binary Mode**		**Source Mode**	
			Bytes	**States**	**Bytes**	**States**
MOV	Rmd, Rms	Byte register to byte register	3	2	2	1
	WRjd, WRjs	Word register to word register	3	2	2	1
	DRkd, DRks	Dword register to dword register	3	3	2	2
	Rm, #data	Immediate 8-bit data to byte register	4	3	3	2
	WRj, #data16	Immediate 16-bit data to word register	5	3	4	2
	DRk, #0data16	zero-ext 16bit immediate data to dword register	5	5	4	4
	DRk, #1data16	one-ext 16bit immediate data to dword register	5	5	4	4
	Rm, dir8	Direct address to byte register	4	3[3]	3	2[3]
	WRj, dir8	Direct address to word register	4	4	3	3
	DRk, dir8	Direct address to dword register	4	6	3	5
	Rm, dir16	Direct address (64K) to byte register	5	3[4]	4	2[4]
	WRj, dir16	Direct address (64K) to word register	5	4[5]	4	3[5]
	DRk, dir16	Direct address (64K) to dword register	5	6[6]	4	5[6]
	Rm, @WRj	Indirect address (64K) to byte register	4	3[4]	3	2[4]
	Rm, @DRk	Indirect address (16M) to byte register	4	4[4]	3	3[4]
	WRjd, @WRjs	Indirect address (64K) to word register	4	4[5]	3	3[5]
	WRj, @DRk	Indirect address (16M) to word register	4	5[5]	3	4[5]
	dir8, Rm	Byte register to direct address	4	4[3]	3	3[3]
	dir8, WRj	Word register to direct address	4	5	3	4
	dir8, DRk	Dword register to direct address	4	7	3	6
	dir16, Rm	Byte register to direct address (64K)	5	4[4]	4	3[4]
	dir16, WRj	Word register to direct address (64K)	5	5[5]	4	4[5]
	dir16, DRk	Dword register to direct address (64K)	5	7[6]	4	6[6]
	@WRj, Rm	Byte register to indirect address (64K)	4	4[4]	3	3[4]
	@DRk, Rm	Byte register to indirect address (16M)	4	5[4]	3	4[4]
	@WRjd, WRjs	Word register to indirect address (64K)	4	5[5]	3	4[5]
	@DRk, WRj	Word register to indirect address (16M)	4	6[5]	3	5[5]
	Rm, @WRj+dis16	Indirect with 16–bit dis (64K) to byte register	5	6[4]	4	5[4]
	WRj, @WRj+dis16	Indirect with 16–bit dis (64K) to word register	5	7[5]	4	6[5]
	Rm, @DRk+dis16	Indirect with 16–bit dis (16M) to byte register	5	7[4]	4	6[4]
	WRj, @DRk+dis16	Indirect with 16–bit dis (16M) to word register	5	8[5]	4	7[5]
	@WRj+dis16, Rm	Byte register to indirect with 16–bit dis (64K)	5	6[4]	4	5[4]
	@WRj+dis16, WRj	Word register to indirect with 16–bit dis (64K)	5	7[5]	4	6[5]
	@DRk+dis16, Rm	Byte register to indirect with 16–bit dis (16M)	5	7[4]	4	6[4]
	@DRk+dis16, WRj	Word register to indirect with 16–bit dis (16M)	5	8[5]	4	7[5]

Notes:

1. *Instructions that move bits are in Table 16.*
2. *Move instructions unique to the C251 Architecture.*
3. *If this instruction addresses an I/O Port (Px, x= 0-3), add 1 to the number of states. Add 2 if it addresses a Peripheral SFR.*
4. *If this instruction addresses external memory location, add N+2 to the number of states (N: number of wait states).*
5. *If this instruction addresses external memory location, add 2(N+1) to the number of states (N: number of wait states).*
6. *If this instruction addresses external memory location, add 4(N+2) to the number of states (N: number of wait states).*

Table 16Summary of Bit Instructions

Clear Bit	CLR <dest>	dest opnd ← 0
Set Bit	SETB <dest>	dest opnd ← 1
Complement Bit	CPL <dest>	dest opnd ← ∅ bit
AND Carry with Bit	ANL CY, <src>	(CY) ← (CY) ∧ src opnd
AND Carry with Complement of Bit	ANL CY, /<src>	(CY) ← (CY) ∧ ∅ src opnd
OR Carry with Bit	ORL CY, <src>	(CY) ← (CY) ∨ src opnd
OR Carry with Complement of Bit	ORL CY, /<src>	(CY) ← (CY) ∨ ∅ src opnd
Move Bit to Carry	MOV CY, <src>	(CY) ← src opnd
Move Bit from Carry	MOV <dest>, CY	dest opnd ← (CY)

Mnemonic	<dest>, <src>[1]	Comments	Binary Mode		Source Mode	
			Bytes	States	Bytes	States
CLR	CY	Clear carry	1	1	1	1
	bit51	Clear direct bit	2	2[3]	2	2[3]
	bit	Clear direct bit	4	4[3]	3	3[3]
SETB	CY	Set carry	1	1	1	1
	bit51	Set direct bit	2	2[3]	2	2[3]
	bit	Set direct bit	4	4[3]	3	3[3]
CPL	CY	Complement carry	1	1	1	1
	bit51	Complement direct bit	2	2[3]	2	2[3]
	bit	Complement direct bit	4	4[3]	3	3[3]
ANL	CY, bit51	And direct bit to carry	2	1[2]	2	1[2]
	CY, bit	And direct bit to carry	4	3[2]	3	2[2]
	CY, /bit51	And complemented direct bit to carry	2	1[2]	2	1[2]
	CY, /bit	And complemented direct bit to carry	4	3[2]	3	2[2]
ORL	CY, bit51	Or direct bit to carry	2	1[2]	2	1[2]
	CY, bit	Or direct bit to carry	4	3[2]	3	2[2]
	CY, /bit51	Or complemented direct bit to carry	2	1[2]	2	1[2]
	CY, /bit	Or complemented direct bit to carry	4	3[2]	3	2[2]
MOV	CY, bit51	Move direct bit to carry	2	1[2]	2	1[2]
	CY, bit	Move direct bit to carry	4	3[2]	3	2[2]
	bit51, CY	Move carry to direct bit	2	2[3]	2	2[3]
	bit, CY	Move carry to direct bit	4	4[3]	3	3[3]

Notes:

1. *A shaded cell denotes an instruction in the C51 Architecture.*
2. *If this instruction addresses an I/O Port (Px, x= 0–3), add 1 to the number of states. Add 2 if it addresses a Peripheral SFR.*
3. *If this instruction addresses an I/O Port (Px, x= 0–3), add 2 to the number of states. Add 3 if it addresses a Peripheral SFR.*

Table 17Summary of Exchange, Push and Pop Instructions

Exchange bytes	XCH A, <src>	(A) ↔ src opnd
Exchange Digit	XCHD A, <src>	$(A)_{3:0}$ ↔ src $\text{opnd}_{3:0}$
Push	PUSH <src>	(SP) ← (SP) +1; ((SP)) ← src opnd; (SP) ← (SP) + size (src opnd) – 1
Pop	POP <dest>	(SP) ← (SP) – size (dest opnd) + 1; dest opnd ← ((SP)); (SP) ← (SP) –1

Mnemonic	<dest>, <src>[1]	Comments	Binary Mode		Source Mode	
			Bytes	States	Bytes	States
XCH	A, Rn	ACC and register	1	3	2	4
	A, dir8	ACC and direct address (on–chip RAM or SFR)	2	3[3]	2	3[3]
	A, @Ri	ACC and indirect address	1	4	2	5
XCHD	A, @Ri	ACC low nibble and indirect address (256 bytes)	1	4	2	5
PUSH	dir8	Push direct address onto stack	2	2[2]	2	2[2]
	#data	Push immediate data onto stack	4	4	3	3
	#data16	Push 16-bit immediate data onto stack	5	5	4	5
	Rm	Push byte register onto stack	3	4	2	3
	WRj	Push word register onto stack	3	5	2	4
	DRk	Push double word register onto stack	3	9	2	8
POP	dir8	Pop direct address (on–chip RAM or SFR) from stack	2	3[2]	2	3[2]
	Rm	Pop byte register from stack	3	3	2	2
	WRj	Pop word register from stack	3	5	2	4
	DRk	Pop double word register from stack	3	9	2	8

Notes:

1. *A shaded cell denotes an instruction in the C51 Architecture.*
2. *If this instruction addresses an I/O Port (Px, x= 0-3), add 1 to the number of states. Add 2 if it addresses a Peripheral SFR.*
3. *If this instruction addresses an I/O Port (Px, x= 0-3), add 2 to the number of states. Add 3 if it addresses a Peripheral SFR.*

Table 18Summary of Conditional Jump Instructions (1/2)

Jump conditional on status	Jcc rel	(PC) ← (PC) + size (instr);
		IF [cc] THEN (PC) ← (PC) + rel

Mnemonic	<dest>, <src>[1]	Comments	Binary Mode[2]		Source Mode[2]	
			Bytes	States	Bytes	States
JC	rel	Jump if carry	2	1/4[3]	2	1/4[3]
JNC	rel	Jump if not carry	2	1/4[3]	2	1/4[3]
JE	rel	Jump if equal	3	2/5[3]	2	1/4[3]
JNE	rel	Jump if not equal	3	2/5[3]	2	1/4[3]
JG	rel	Jump if greater than	3	2/5[3]	2	1/4[3]
JLE	rel	Jump if less than, or equal	3	2/5[3]	2	1/4[3]
JSL	rel	Jump if less than (signed)	3	2/5[3]	2	1/4[3]
JSLE	rel	Jump if less than, or equal (signed)	3	2/5[3]	2	1/4[3]
JSG	rel	Jump if greater than (signed)	3	2/5[3]	2	1/4[3]
JSGE	rel	Jump if greater than or equal (signed)	3	2/5[3]	2	1/4[3]

Notes:

1. *A shaded cell denotes an instruction in the C51 Architecture.*
2. *States are given as jump not-taken/taken.*
3. *In internal execution only, add 1 to the number of states of the 'jump taken' if the destination address is internal and odd.*

Table 19Summary of Conditional Jump Instructions (2/2)

Jump if bit	JB <src>, rel	(PC) ← (PC) + size (instr);
		IF [src opnd= 1] THEN (PC) ← (PC) + rel

Jump if not bit	JNB <src>, rel	(PC) ← (PC) + size (instr);
		IF [src opnd= 0] THEN (PC) ← (PC) + rel

Jump if bit and clear	JBC <dest>, rel	(PC) ← (PC) + size (instr);
		IF [dest opnd= 1] THEN
		dest opnd ← 0
		(PC) ← (PC) + rel

Jump if accumulator is zero	JZ rel	(PC) ← (PC) + size (instr);
		IF [(A)= 0] THEN (PC) ← (PC) + rel

Jump if accumulator is not zero	JNZ rel	(PC) ← (PC) + size (instr);
		IF [(A) ≠ 0] THEN (PC) ← (PC) + rel

Compare and jump if not equal	CJNE <src1>, <src2>, rel	(PC) ← (PC) + size (instr);
		IF [src opnd1 < src opnd2] THEN (CY) ← 1
		IF [src opnd1 ≥ src opnd2] THEN (CY) ← 0
		IF [src opnd1 ≠ src opnd2] THEN (PC) ← (PC) + rel

Decrement and jump if not zero	DJNZ <dest>, rel	(PC) ← (PC) + size (instr); dest opnd ← dest opnd –1;
		IF [∅ (Z)] THEN (PC) ← (PC) + rel

Mnemonic	<dest>, <src>[1]	Comments	Binary Mode[2]		Source Mode[2]	
			Bytes	States	Bytes	States
JB	bit51, rel	Jump if direct bit is set	3	2/5[3][6]	3	2/5[3][6]
	bit, rel	Jump if direct bit of 8-bit address location is set	5	4/7[3][6]	4	3/6[3][6]
JNB	bit51, rel	Jump if direct bit is not set	3	2/5[3][6]	3	2/5[3][6]
	bit, rel	Jump if direct bit of 8-bit address location is not set	5	4/7[3][6]	4	3/6[3]
JBC	bit51, rel	Jump if direct bit is set & clear bit	3	4/7[5][6]	3	4/7[5][6]
	bit, rel	Jump if direct bit of 8-bit address location is set and clear	5	7/10[5][6]	4	6/9[5][6]
JZ	rel	Jump if ACC is zero	2	2/5[6]	2	2/5[6]
JNZ	rel	Jump if ACC is not zero	2	2/5[6]	2	2/5[6]
CJNE	A, dir8, rel	Compare direct address to ACC and jump if not equal	3	2/5[3][6]	3	2/5[3][6]
	A, #data, rel	Compare immediate to ACC and jump if not equal	3	2/5[6]	3	2/5[6]
	Rn, #data, rel	Compare immediate to register and jump if not equal	3	2/5[6]	4	3/6[6]
	@Ri, #data, rel	Compare immediate to indirect and jump if not equal	3	3/6[6]	4	4/7[6]
DJNZ	Rn, rel	Decrement register and jump if not zero	2	2/5[6]	3	3/6[6]
	dir8, rel	Decrement direct address and jump if not zero	3	3/6[4][6]	3	3/6[4][6]

Notes:

1. *A shaded cell denotes an instruction in the C51 Architecture.*
2. *States are given as jump not-taken/taken.*
3. *If this instruction addresses an I/O Port (Px, x= 0-3), add 1 to the number of states. Add 2 if it addresses a Peripheral SFR.*
4. *If this instruction addresses an I/O Port (Px, x= 0-3), add 2 to the number of states. Add 3 if it addresses a Peripheral SFR.*
5. *If this instruction addresses an I/O Port (Px, x= 0-3), add 3 to the number of states. Add 5 if it addresses a Peripheral SFR.*
6. *In internal execution only, add 1 to the number of states of the 'jump taken' if the destination address is internal and odd.*

Table 20Summary of unconditional Jump Instructions

Absolute jump	AJMP <src>	$(PC) \leftarrow (PC) + 2; (PC)_{10:0} \leftarrow$ src opnd
Extended jump	EJMP <src>	$(PC) \leftarrow (PC) +$ size (instr); $(PC)_{23:0} \leftarrow$ src opnd
Long jump	LJMP <src>	$(PC) \leftarrow (PC) +$ size (instr); $(PC)_{15:0} \leftarrow$ src opnd
Short jump	SJMP rel	$(PC) \leftarrow (PC) + 2; (PC) \leftarrow (PC) +$rel
Jump indirect	JMP @A +DPTR	$(PC)_{23:16} \leftarrow$ FFh; $(PC)_{15:0} \leftarrow (A) + (DPTR)$
No operation	NOP	$(PC) \leftarrow (PC) + 1$

Mnemonic	<dest>, <src>(1)	Comments	Binary Mode		Source Mode	
			Bytes	States	Bytes	States
AJMP	addr11	Absolute jump	2	3(2)(3)	2	3(2)(3)
EJMP	addr24	Extended jump	5	6(2)(4)	4	5(2)(4)
	@DRk	Extended jump (indirect)	3	7(2)(4)	2	6(2)(4)
LJMP	@WRj	Long jump (indirect)	3	6(2)(4)	2	5(2)(4)
	addr16	Long jump (direct address)	3	5(2)(4)	3	5(2)(4)
SJMP	rel	Short jump (relative address)	2	4(2)(4)	2	4(2)(4)
JMP	@A +DPTR	Jump indirect relative to the DPTR	1	5(2)(4)	1	5(2)(4)
NOP		No operation (Jump never)	1	1	1	1

Notes:

1. *A shaded cell denotes an instruction in the C51 Architecture.*
2. *In internal execution only, add 1 to the number of states if the destination address is internal and odd.*
3. *Add 2 to the number of states if the destination address is external.*
4. *Add 3 to the number of states if the destination address is external.*

Table 21Summary of Call and Return Instructions

Absolute call	ACALL <src>	$(PC) \leftarrow (PC) + 2$; push $(PC)_{15:0}$; $(PC)_{10:0} \leftarrow$ src opnd
Extended call	ECALL <src>	$(PC) \leftarrow (PC) +$ size (instr); push $(PC)_{23:0}$; $(PC)_{23:0} \leftarrow$ src opnd
Long call	LCALL <src>	$(PC) \leftarrow (PC) +$ size (instr); push $(PC)_{15:0}$; $(PC)_{15:0} \leftarrow$ src opnd
Return from subroutine	RET	pop $(PC)_{15:0}$
Extended return from subroutine	ERET	pop $(PC)_{23:0}$

Return from interrupt	RETI	IF [INTR= 0] THEN pop $(PC)_{15:0}$
		IF [INTR= 1] THEN pop $(PC)_{23:0}$; pop (PSW1)

Trap interrupt	TRAP	$(PC) \leftarrow (PC) +$ size (instr);
		IF [INTR= 0] THEN push $(PC)_{15:0}$
		IF [INTR= 1] THEN push (PSW1); push $(PC)_{23:0}$

Mnemonic	<dest>, <src>(1)	Comments	Binary Mode		Source Mode	
			Bytes	States	Bytes	States
ACALL	addr11	Absolute subroutine call	2	9(2)(3)	2	9(2)(3)
ECALL	@DRk	Extended subroutine call (indirect)	3	14(2)(3)	2	13(2)(3)
	addr24	Extended subroutine call	5	14(2)(3)	4	13(2)(3)
LCALL	@WRj	Long subroutine call (indirect)	3	10(2)(3)	2	9(2)(3)
	addr16	Long subroutine call	3	9(2)(3)	3	9(2)(3)
RET		Return from subroutine	1	7(2)	1	7(2)
ERET		Extended subroutine return	3	9(2)	2	8(2)
RETI		Return from interrupt	1	7(2)(4)	1	7(2)(4)
TRAP		Jump to the trap interrupt vector	2	12(4)	1	11(4)

Notes:

1. *A shaded cell denotes an instruction in the C51 Architecture.*
2. *In internal execution only, add 1 to the number of states if the destination/return address is internal and odd.*
3. *Add 2 to the number of states if the destination address is external.*
4. *Add 5 to the number of states if INTR= 1.*

10.2. Opcode Map and Supporting Tables

Table 22Instructions for 80C51 Microcontrollers

Bin	x0	x1	x2	x3
Src	**x0**	**x1**	**x2**	**x3**
0	NOP	AJMP addr11	LJMP addr16	RR A
1	JBC bit51, rel	ACALL addr11	LCALL addr16	RRC A
2	JB bit51, rel	AJMP addr11	RET	RL A
3	JNB bit51, rel	ACALL addr11	RETI	RLC A
4	JC rel	AJMP addr11	ORL dir8, A	ORL dir8,#data
5	JNC rel	ACALL addr11	ANL dir8, A	ANL dir8,#data
6	JZ rel	AJMP addr11	XRL dir8, A	XRL dir8,#data
7	JNZ rel	ACALL addr11	ORL CY,bit51	JMP @A+DPTR
8	SJMP rel	AJMP addr11	ANL CY,bit51	MOVC A,@A+PC
9	MOV DPTR,#data16	ACALL addr11	MOV bit51,CY	MOVC A,@A+DPTR
A	ORL CY,bit51	AJMP addr11	MOV CY,bit51	INC DPTR
B	ANL CY,bit51	ACALL addr11	CPL bit51	CPL CY
C	PUSH dir8	AJMP addr11	CLR bit51	CLR CY
D	POP dir8	ACALL addr11	SETB bit51	SETB CY
E	MOVX A, @DPTR	AJMP addr11	MOVX A,@RI	
F	MOV @DPTR,A	ACALL addr11	MOVX @RI,A	

Bin	x4	x5	x6–x7	x8–xF
Src	**x4**	**x5**	**A5x6–A5x7⋆**	**A5x8–A5xF⋆**
0	INC A	INC dir8	INC @Ri	INC Rn
1	DEC A	DEC dir8	DEC @Ri	DEC Rn
2	ADD A,#data	ADD A,dir8	ADD A,@Ri	ADD A,Rn
3	ADDC A,#data	ADDC A,dir8	ADDC A,@Ri	ADDC A,Rn
4	ORL A,#data	ORL A,dir8	ORL A,@Ri	ORL A,Rn
5	ANL A,#data	ANL A,dir8	ANL A,@Ri	ANL A,Rn
6	XRL A,#data	XRL A,dir8	XRL A,@Ri	XRL A,Rn
7	MOV A,#data	MOV dir8,#data	MOVX @Ri,data	MOV Rn,#data
8	DIV AB	MOV dir8,dir8	MOV dir8,@Ri	MOV dir8,Rn
9	SUBB A,#data	SUBB A,dir8	SUBB A,@Ri	SUBB A,Rn
A	MUL AB	*Escape*	MOV @Ri,dir8	MOV Rn,dir8
B	CJNE A,#data,rel	CJNE A,dir8,rel	CJNE @Ri,#data,rel	CJNE Rn,#data,rel
C	SWAP A	XCH A,dir8	XCH A,@Ri	XCH A,Rn
D	DA A	DJNZ dir8,rel	XCHD A,@Ri	DJNZ Rn,rel
E	CLR A	MOV A,dir8	MOV A,@Ri	MOV A,Rn
F	CPL A	MOV dir8,A	MOV @Ri,A	MOV Rn,A

⋆ x takes the values found in Bin and Src column.

Table 23New Instructions for the C251 Architecture

Bin	A5x8★	A5x9★	A5xA★	A5xB★
Src	**x8★**	**x9★**	**xA★**	**xB★**
0	JSLE rel	MOV Rm, @WRj +dis16	MOVZ WRj, Rm	INC R, #short (1) MOV reg, ind
1	JSG rel	MOV @WRj +dis16, Rm	MOVS WRj, Rm	DEC R, #short (1) MOV ind, reg
2	JLE rel	MOV Rm, @DRk +dis24		
3	JG rel	MOV @DRk +dis24, Rm		
4	JSL rel	MOV WRj, @WRj +dis16		
5	JSGE rel	MOV @WRj +dis16, WRj		
6	JE rel	MOV WRj, @DRk +dis24		
7	JNE rel	MOV @DRk +dis16, WRj	MOV op1, reg (2)	
8		LJMP @WRj EJMP @DRk	EJMP addr24	
9		LCALL @WRj ECALL @DRk	ECALL addr24	
A		*Escape* Bit Instructions (3)	ERET	
B		TRAP		
C			PUSH op1 (4) MOV DRk, PC	
D			POP op1 (4)	

Bin	A5xC★	A5xD★	A5xE★	A5xF★
Src	**xC★**	**xD★**	**xE★**	**xF★**
0			SRA reg	
1			SRL reg	
2	ADD Rmd, Rms	ADD WRjd, WRjs	ADD reg, op2*	ADD DRkd, DRks
3			SLL reg	
4	ORL Rmd, Rms	ORL WRjd, WRjs	ORL reg, op2*	
5	ANL Rmd, Rms	ANL WRjd, WRjs	ANL reg, op2*	
6	XRL Rmd, Rms	XRL WRjd, WRjs	XRL reg, op2*	
7	MOV Rmd, Rms	MOV WRjd, WRjs	MOV reg, op2*	MOV DRkd, DRks
8	DIV Rmd, Rms	DIV WRjd, WRjs		
9	SUB Rmd, Rms	SUB WRjd, WRjs	SUB reg, op2*	SUB DRkd, DRkd
A	MUL Rmd, Rms	MUL WRjd, WRjs		
B	CMP Rmd, Rms	CMP WRjd, WRjs	CMP reg, op2*	CMP DRkd, DRks

Notes :

★ *x takes the values found in Bin and Src column.*

1. *R = Rm/WRj/DRk.*
2. *op1, op2 are defined in Table 24*
3. *See Table 26and Table 27*
4. *See Table 28*

Table 24Data Instructions

Instruction	Byte 0		Byte 1		Byte 2		Byte 3
Oper Rmd, Rms	x	C	md	ms			
Oper WRjd, WRjs	x	D	jd/2	js/2			
Oper DRkd, DRks	x	F	kd/4	ks/4			
Oper Rm, #data	x	E	m	0	#data		
Oper WRj, #data16	x	E	j/2	4	#data (high)		#data (low)
Oper DRk, #data16	x	E	k/4	8	#data (high)		#data (low)
MOVH DRk(h), #data16 MOV DRk, #1data16 CMP DRk,#1data16	7 7 B	A E E	k/4	C	#data (high)		#data (low)
Oper Rm, dir8	x	E	m	1	dir8 addr		
Oper WRj, dir8	x	E	j/2	5	dir8 addr		
Oper DRk, dir8	x	E	k/4	D	dir8 addr		
Oper Rm, dir16	x	E	m	3	dir16 addr (high)		dir16 addr (low)
Oper WRj, dir16	x	E	j/2	7	dir16 addr (high)		dir16 addr (low)
Oper DRk, dir16 [1]	x	E	k/4	F	dir16 addr (high)		dir16 addr (low)
Oper Rm, @WRj	x	E	j/2	9	m	0	
Oper Rm, @DRk	x	E	k/4	B	m	0	

Note :

1. For this instruction, the only valid operation is MOV.

Table 25High Nibble, Byte 0 of Data Instructions

x	Operation	Notes
2	ADD reg, op2	All data addressing modes are supported.
9	SUB reg, op2	
B	CMP reg, op2 [1]	
4	ORL reg, op2 [2]	
5	ANL reg, op2 [2]	
6	XRL reg, op2 [2]	
7	MOV reg, op2	
8	DIV reg, op2	Two modes only: reg, op2 = Rmd, Rms reg, op2 = Wjd, Wjs
A	MUL reg, op2	

Notes :

1. The CMP operation does not support DRk, direct16.

2. For the ORL, ANL and XRL operations, neither reg nor op2 can be DRk.

All of the bit instructions in the C251 Architecture (See Table 23) have opcode A9, which serves as an escape byte (similar to A5). The high nibble of byte 1 specifies the bit instruction, as given in Table 26

Table 26Bit Instructions

Instruction	Byte 0(x)		Byte 1		Byte2	Byte 3
BitInstr (dir8)	A	9	x	0 bit	dir8_addr	rel_addr

Table 27Byte 1 (High Nibble) for Bit Instructions

x	Bit Instruction
1	JBC bit
2	JB bit
3	JNB bit
7	ORL CY, bit
8	ANL CY, bit
9	MOV bit, CY
A	MOV CY, bit
B	CPL bit
C	CLR bit
D	SETB bit
E	ORL CY, /bit
F	ANL CY, /bit

Table 28PUSH/POP Instructions

Instruction	Byte 0(x)		Byte 1		Byte 2	Byte 3
PUSH #data16	C	A	0	6	#data16 (high)	#data16 (low)
PUSH #data	C	A	0	2	#data	
PUSH Rm	C	A	m	8		
PUSH WRj	C	A	j/2	9		
PUSH DRk	C	A	k/4	B		
MOV DRk, PC	C	A	k/4	1		
POP Rm	D	A	m	8		
POP WRj	D	A	j/2	9		
POP DRk	D	A	k/4	B		

Table 29Control Instructions

Instruction	Byte 0		Byte 1		Byte 2	Byte 3
ACALL addr11	addr[10:9] 1	1	addr[7:0]			
AJMP addr11	addr[10:8] 0	1	addr[7:0]			
EJMP addr24	8	A	addr[23:16]		addr[15:8]	addr[7:0]
ECALL addr24	9	A	addr[23:16]		addr[15:8]	addr[7:0]
LJMP @WRj	8	9	j/2	4		
LCALL @WRj	9	9	j/2	4		
EJMP @DRk	8	9	k/4	8		
ECALL @DRk	9	9	k/4	8		
ERET	A	A				
JE rel	8	8	rel			
JNE rel	7	8	rel			
JLE rel	2	8	rel			
JG rel	3	8	rel			
JSL rel	4	8	rel			
JSGE rel	5	8	rel			
JSLE rel	0	8	rel			
JSG rel	1	8	rel			
TRAP	B	9				

Table 30Displacement/Extended MOVs Instructions

Instruction	Byte 0		Byte 1		Byte 2		Byte 3
MOV Rm, @WRj +dis16	0	9	m	j/2	dis[15:8]		dis[7:0]
MOVWRk, @WRj +dis16	4	9	j/2	k/2	dis[15:8]		dis[7:0]
MOV Rm, @DRk +dis24	2	9	m	k/4	dis[15:8]		dis[7:0]
MOV WRj, @DRk +dis24	6	9	j/2	k/4	dis[15:8]		dis[7:0]
MOV @WRj +dis16, Rm	1	9	m	j/2	dis[15:8]		dis[7:0]
MOV @WRj +dis16, WRk	5	9	j/2	k/2	dis[15:8]		dis[7:0]
MOV @DRk +dis24, Rm	3	9	m	k/4	dis[15:8]		dis[7:0]
MOV @DRk +dis24, WRj	7	9	j/2	k/4	dis[15:8]		dis[7:0]
MOVS WRj, Rm	1	A	j/2	m			
MOVZ WRj, Rm	0	A	j/2	m			
MOV WRjd, @WRjs	0	B	js/2	8	jd/2	0	
MOV WRj, @DRk	0	B	k/4	A	j/2	0	
MOV @WRjd, WRjs	1	B	js/2	8	jd/2	0	
MOV @DRk, WRj	1	B	k/4	A	j/2	0	
MOV dir8, Rm	7	A	m	3	dir8 addr		
MOV dir8, WRj	7	A	j/2	5	dir8 addr		
MOV dir8, DRk	7	A	k/4	D	dir8 addr		
MOV dir16, Rm	7	A	m	1	dir16 addr (high)		dir16 addr (low)
MOV dir16, WRj	7	A	j/2	7	dir16 addr (high)		dir16 addr (low)
MOV dir16, DRk	7	A	k/4	F	dir16 addr (high)		dir16 addr (low)
MOV @WRj, Rm	7	A	j/2	9	m	0	
MOV @DRk, Rm	7	A	k/4	B	m	0	

Table 31Shift Instructions

	Instruction	Byte 0(x)		Byte 1	
1	SRA Rm	0	E	m	0
2	SRA WRj	0	E	j/2	4
3	SRL Rm	1	E	m	0
4	SRl WRj	1	E	j/2	4
5	SLL Rm	3	E	m	0
6	SLL WRj	3	E	j/2	4

Table 32INC/DEC Instructions

	Instruction	Byte 0(x)		Byte 1	
1	INC Rm, #short	0	B	m	00 vv
2	INC WRj, #short	0	B	j/2	01 vv
3	INC DRk, #short	0	B	k/4	11 vv
4	DEC Rm, #short	1	B	m	00 vv
5	DEC WRj, #short	1	B	j/2	01 vv
6	DEC DRk, #short	1	B	k/4	11 vv

Table 33Encoding for INC/DEC Instructions

vv	#short
00	1
01	2
10	4

11. Instruction Descriptions

This section describes each instruction in the C251 Architecture.

ACALL <addr11>

Function:

Absolute call

Description:

Unconditionally calls a subroutine at the specified address. The instruction increments the 3–byte PC twice to obtain the address of the following instruction, then pushes bytes 0 and 1 of the result onto the stack (byte 0 first) and increments the stack pointer twice. The destination address is obtained by successively concatenating bits 15-11 of the incremented PC, opcode bits 7-5, and the second byte of the instruction. The subroutine called must therefore start within the same 2–Kbyte “page” of the program memory as the first byte of the instruction following ACALL.

FLAGS :

CY	AC	OV	N	Z
–	–	–	–	–

Example :

The stack pointer (SP) contains 07h and the label ”SUBRTN” is at program memory location 0345h. After executing the instruction ACALL SUBRTN at location 0123h, SP contains 09h; on–chip RAM locations 09h and 08h contain 01h and 25h, respectively; and the PC contains 0345h.

[Encoding]

a10 | a9 | a8 | 1	**1**	addr7–addr4	addr3–addr0

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [Encoding]

Operation:

ACALL
(PC) ← (PC) + 2
(SP) ← (SP) + 1
((SP)) ← (PC.7:0)
(SP) ← (SP) + 1
((SP)) ← (PC.15:8)
(PC.10:0) ← page address

ADD <dest>,<src>

Function:

Add

Description:

Adds the source operand to the destination operand, which can be a register or the accumulator, leaving the result in the register or accumulator. If there is a carry out of bit 7 (CY), the CY flag is set. If byte variables are added, and if there is a carry out of bit 3 (AC), the AC flag is set. For addition of unsigned integers, the CY flag indicates that an overflow occurred.

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

Bit 6 and bit 7 in this description refer to the most significant byte of the operand (8, 16 or 32 bit)

Four source operand addressing modes are allowed: register, direct, register–indirect and immediate.

FLAGS :

CY	AC	OV	N	Z
✓	✓	✓	✓	✓

Example:

Register 1 contains 0C3h (11000011B) and register 0 contains 0AAh (10101010B). After executing the instruction ADD R1,R0 register 1 contains 6Dh (01101101B), the AC flag is clear, and the CY and OV flags are set.

ADD A,#data

[Encoding]

2	4	immed. data

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [Encoding]

Operation:

ADD
(A) ← (A) + #data

ADD A,dir8

[Encoding]

2	5	addr7–addr0

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [Encoding]

Operation:

ADD
(A) ← (A) + (dir8)

ADD A,@Ri

[Encoding]

2	0 1 1 i

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [A5][Encoding]

Operation:

ADD
(A) ← (A) + ((Ri))

ADD A,Rn

[Encoding]

2	1 rrr

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [A5][Encoding]

Operation:

ADD
(A) ← (A) + (Rn)

ADD Rmd,Rms

[Encoding]

2	**C**	ssss	SSSS

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

ADD
(Rmd) ← (Rmd) + (Rms)

ADD WRjd, WRjs

[Encoding]

2	**D**	tttt	TTTT

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

ADD
(WRjd) ← (WRjd) + (WRjs)

ADD DRkd,DRks

[Encoding]

2	**F**	uuuu	UUUU

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

ADD
(DRkd) ← (DRkd) + (DRks)

ADD Rm,#data

[Encoding]

2	**E**	ssss	**0**	immed data

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

ADD
(Rm) ← (Rm) + #data

ADD WRj,#data16

[Encoding]

2	**E**	tttt	**4**	immed data hi	immed data low

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

ADD
(WRj) ← (WRj) + #data16

ADD DRk,#0data16

[Encoding]

2	**E**	uuuu	**8**	immed data hi	immed data low

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

ADD
(DRk) ← (DRk) + #data16

ADD Rm,dir8

[Encoding]

2	**E**	ssss	**1**	addr7–addr4	addr3–addr0

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

ADD
(Rm) ← (Rm) + (dir8)

ADD WRj,dir8

[Encoding]

2	**E**	tttt	**5**	addr7–addr4	addr3–addr0

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

ADD
(WRj) ← (WRj) + (dir8)

ADD Rm,dir16

[Encoding]

2	**E**	ssss	**3**	addr15–addr12	addr11–addr8	addr7–addr4	addr3–addr0

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

ADD
(Rm) ← (Rm) + (dir16)

ADD WRj,dir16

[Encoding]

2	**E**	tttt	**7**	addr15–addr12	addr11–addr8	addr7–addr4	addr3–addr0

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

ADD
(WRj) ← (WRj) + (dir16)

ADD Rm,@WRj

[Encoding]

2	**E**	tttt	**9**	ssss	**0**

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

ADD
(Rm) ← (Rm) + ((WRj))

ADD Rm,@DRk

[Encoding]

2	**E**	uuuu	**B**	ssss	**0**

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

ADD
(Rm) ← (Rm) + ((DRk))

ADDC A,<src>

Function:

Add with carry

Description:

Simultaneously adds the specified byte variable, the CY flag and the accumulator contents, leaving the result in the accumulator. If there is a carry out of bit 7 (CY), the CY flag is set; if there is a carry out of bit 3 (AC), the AC flag is set. When adding unsigned integers, the CY flag indicates that an overflow occurred.

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

Bit 6 and bit 7 in this description refer to the most significant byte of the operand (8, 16 or 32 bit)

Four source operand addressing modes are allowed: register, direct, register–indirect and immediate.

FLAGS :

CY	AC	OV	N	Z
✓	✓	✓	✓	✓

Example :

The accumulator contains 0C3h (11000011B), register 0 contains 0AAh (10101010B) and the CY flag is set. After executing the instruction ADDC A,R0 the accumulator contains 6Eh (01101110B), the AC flag is clear and the CY and OV flags are set.

ADDC A,#data

[Encoding]

3	4	immed. data

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [Encoding]

Operation:

ADDC
(A) ← (A) + (CY) + #data

ADDC A,dir8

[Encoding]

3	5	addr7–addr0

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [Encoding]

Operation:

ADDC
(A) ← (A) + (CY) + (dir8)

ADDC A,@Ri

[Encoding]

3	011i

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [A5][Encoding]

Operation:

ADDC
(A) ← (A) + (CY) + ((Ri))

ADDC A,Rn

[Encoding]

3	1rrr

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [A5][Encoding]

Operation:

ADDC
(A) ← (A) + (CY) + (Rn)

AJMP addr11

Function:

Absolute jump

Description:

Transfers program execution to the specified address, which is formed at run time by concatenating the upper five bits of the PC (after incrementing the PC twice), opcode bits 7-5, and the second byte of the instruction. The destination must therefore be within the same 2–Kbyte “page” of program memory as the first byte of the instruction following AJMP.

FLAGS :

CY	AC	OV	N	Z
–	–	–	–	–

Example :

The label ”JMPADR” is at program memory location 0123h. After executing the instruction AJMP JMPADR at location 0345h the PC contains 0123h.

[Encoding]

<table>
<tr><td>a10 | a9 | a8 | 0</td><td><b>1</b></td><td>addr7–addr4</td><td>addr3–addr0</td></tr>
</table>

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [Encoding]

Operation:

AJMP
(PC) ← (PC) + 2
(PC.10:0) ← page address

ANL <dest>,<src>

Function:

Logical–AND

Description:

Performs the bitwise logical–AND (Λ) operation between the specified variables and stores the results in the destination variable.

The two operands allow 10 addressing mode combinations. When the destination is the register or 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 is read from the output data latch, not the input pins.

FLAGS :

CY	AC	OV	N	Z
–	–	–	✓	✓

Example :

Register 1 contains 0C3h (11000011B) and register 0 contains 55h (01010101B). After executing the instruction ANL R1, R0 register 1 contains 41h (01000001B).

When the destination is a directly addressed byte, this instruction clears combinations of bits in any RAM location or hardware register. The mask byte determining the pattern of bits to be cleared would either be an immediate constant contained in the instruction or a value computed in the register or accumulator at run time. The instruction ANL P1,#01110011B clears bits 7, 3, and 2 of output Port 1.

ANL dir8,A

[Encoding]

5	**2**	addr7–addr0

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [Encoding]

Operation:

ANL
(dir8) ← (dir8) Λ (A)

ANL dir8,#data

[Encoding]

5	**3**	addr7–addr0	immed data

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [Encoding]

Operation:

ANL
(dir8) ← (dir8) Λ #data

ANL A,#data

[Encoding]

5	**4**	immed data

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [Encoding]

Operation:

ANL
(A) ← (A) Λ #data

ANL A,dir8

[Encoding]

5	**5**	addr7–addr0

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [Encoding]

Operation:

ANL
(A) ← (A) Λ (dir8)

ANL A,@Ri

[Encoding]

5	011i

Hex Code in:

Binary Mode = [Encoding]
Source Mode =[A5][Encoding]

Operation:

ANL
(A) ← (A) Λ ((Ri))

ANL A,Rn

[Encoding]

5	1rrr

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [A5][Encoding]

Operation:

ANL
(A) ← (A) Λ (Rn)

ANL Rmd,Rms

[Encoding]

5	**C**	ssss	SSSS

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

ANL
(Rmd) ← (Rmd) Λ (Rms)

ANL WRjd,WRjs

[Encoding]

5	**D**	tttt	TTTT

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

ANL
(WRjd) ← (WRjd) Λ (WRjs)

ANL Rm,#data

[Encoding]

5	**F**	ssss	**0**	immed data

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

ANL
(Rm) ← (Rm) Λ #data

ANL WRj,#data16

[Encoding]

5	**E**	tttt	**4**	immed data hi	immed data low

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

ANL
(WRj) ← (WRj) Λ #data16

ANL Rm,dir8

[Encoding]

5	**E**	ssss	**1**	addr7–addr4	addr3–addr0

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

ANL
(Rm) ← (Rm) Λ (dir8)

ANL WRj,dir8

[Encoding]

5	**E**	tttt	**5**	addr7–addr4	addr3–addr0

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

ANL
(WRj) ← (WRj) Λ (dir8)

ANL Rm,dir16

[Encoding]

5	**E**	ssss	**3**	addr15-addr8	addr7-addr0

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

ANL
(Rm) ← (Rm) Λ (dir16)

ANL WRj,dir16

[Encoding]

5	**E**	tttt	**7**	addr15-addr8	addr7-addr0

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

ANL
(WRj) ← (WRj) Λ (dir16)

ANL Rm,@WRj

[Encoding]

5	**E**	tttt	**9**	ssss	**0**

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

ANL
(Rm) ← (Rm) Λ ((WRj))

ANL Rm,@DRk

[Encoding]

5	**E**	uuuu	**B**	ssss	**0**

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

ANL
(Rm) ← (Rm) Λ ((DRk))

ANL CY,<src-bit>

Function:

Logical–AND for bit variables

Description:

If the boolean value of the source bit is a logical 0, clear the CY flag; otherwise leave the CY 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.

Only direct addressing is allowed for the source operand.

FLAGS :

CY	AC	OV	N	Z
✓	–	–	–	–

Example :

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

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

ANL CY,bit51

[Encoding]

8	**2**	bit addr

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [Encoding]

Operation:

ANL
(CY) ← (CY) Λ (bit51)

ANL CY,/bit51

[Encoding]

B	**0**	bit addr

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [Encoding]

Operation:

ANL
(CY) ← (CY) Λ ∅ (bit51)

ANL CY,bit

[Encoding]

A	**9**	**8**	0yyy	bit addr

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

ANL
(CY) ← (CY) Λ (bit)

ANL CY,/bit

[Encoding]

A	**9**	**F**	0yyy	bit addr

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

ANL
(CY) ← (CY) Λ ∅ (bit)

CJNE <dest>,<src>,rel

Function:

Compare and jump if not equal.

Description:

Compares the magnitudes of the first 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. If the unsigned integer value of <dest–byte> is less than the unsigned integer value of <src–byte>, the CY flag is set. 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.

FLAGS :

CY	AC	OV	N	Z
✓	–	–	✓	✓

Example :

The accumulator contains 34h and R7 contains 56h. After executing the first instruction in the sequence

```
                    CJNE  R7,#60h,NOT_EQ ;R7 = 60h ; . . .            . . .
        NOT_EQ:     JC          REQ_LOW    ; IF R7 < 60h
                                ;R7 > 60h
                                ; . . .          . . .
```

the CY flag is set and program execution continues at label NOT_EQ. By testing the CY flag, this instruction determines whether R7 is greater or less than 60h.

If the data being presented to Port 1 is also 34h, then executing the instruction, WAIT: CJNE A,P1,WAIT clears the CY flag and continues with the next instruction in the sequence, since the accumulator does equal the data read from Port 1. (If some other value was being input on Port 1, the program loops at this point until the Port 1 data changes to 34h.)

CJNE A,#data,rel

[Encoding]

B	**4**	immed data	rel addr

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [Encoding]

Operation:

```
(PC) ← (PC) + 3
IF [(A) ≠ #data]
THEN
    (PC) ← (PC) + relative offset
IF [(A) < #data]
THEN
    (CY) ← 1
ELSE
    (CY) ← 0
```

CJNE A,dir8,rel

[Encoding]

B	**5**	addr7-addr0	rel addr

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [Encoding]

Operation:

```
(PC) ← (PC) + 3
IF [(A) ≠ (dir8)]
THEN
    (PC) ← (PC) + relative offset
IF [(A) < (dir8)]
THEN
    (CY) ← 1
ELSE
    (CY) ← 0
```

CJNE @Ri,#data,rel

[Encoding]

B	011i	immed data	rel addr

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [A5][Encoding]

Operation:

```
(PC) ← (PC) + 3
IF [((Ri)) ≠ #data]
THEN
    (PC) ← (PC) + relative offset
IF [((Ri)) < #data]
THEN
    (CY) ← 1
ELSE
    (CY) ← 0
```

CJNE Rn,#data,rel

[Encoding]

B	1rrr	immed data	rel addr

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [A5][Encoding]

Operation:

```
(PC) ← (PC) + 3
IF [(Rn) ≠ #data]
THEN
    (PC) ← (PC) + relative offset
IF [(Rn) < #data]
THEN
    (CY) ← 1
ELSE
    (CY) ← 0
```

CLR A

Function:

Clear accumulator

Description:

Clears the accumulator (i.e., resets all bits to zero).

FLAGS :

CY	AC	OV	N	Z
_	_	_	✓	✓

Example :

The accumulator contains 5Ch (01011100B). The instruction CLR A clears the accumulator to 00h (00000000B).

[Encoding]

E	**4**

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [Encoding]

Operation:

CLR
(A) ← 0

CLR bit

Function:

Clear bit

Description:

Clears the specified bit. CLR can operate on the CY flag or any directly addressable bit.

FLAGS : Only for instructions with CY as the operand.

CY	AC	OV	N	Z
✓	–	–	–	–

Example :

Port 1 contains 5Dh (01011101B). After executing the instruction CLR P1.2 Port 1 contains 59h (01011001B).

CLR bit51

[Encoding]

C	**2**	bit addr

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

CLR
(bit51) ← 0

CLR CY

[Encoding]

C	**3**

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [Encoding]

Operation:

CLR
(CY) ← 0

CLR bit

* If this instruction addresses a Port (Px, x = 0-3), add 2 states.

[Encoding]

A	**9**	**C**	0 yyy	bit addr

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

CLR
(bit) ← 0

CMP <dest>,<src>

Function:

Compare

Description:

Subtracts the source operand from the destination operand. The result is not stored in the destination operand. If a borrow is needed for bit 7, the CY (borrow) flag is set; otherwise it is clear.

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

Bit 7 in this description refers to the most significant byte of the operand (8, 16 or 32 bit)

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

FLAGS :

CY	AC	OV	N	Z
✓	✓	✓	✓	✓

Example :

Register 1 contains 0C9h (11001001B) and register 0 contains 54h (01010100B). The instruction CMP R1,R0 clears the CY and AC flags and sets the OV flag.

CMP Rmd,Rms

[Encoding]

B	**C**	ssss	SSSS

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

CMP
(Rmd) - (Rms)

CMP WRjd,WRjs

[Encoding]

B	**D**	tttt	TTTT

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

CMP
(WRjd) - (WRjs)

CMP DRkd,DRks

[Encoding]

B	**F**	uuuu	UUUU

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

CMP
(DRkd) - (DRks)

CMP Rm,#data

[Encoding]

B	**E**	ssss	**0**	immed data

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

CMP
(Rm) - #data

CMP WRj,#data16

[Encoding]

B	**E**	tttt	**4**	immed data hi	immed data low

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

CMP
(WRj) - #data16

CMP DRk, #0data16

[Encoding]

B	**E**	uuuu	**8**	immed data hi	immed data hi

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

CMP (DRk) - #0data16

CMP DRk,#1data16

[Encoding]

B	**E**	uuuu	**C**	immed data hi	immed data hi

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

CMP
(DRk) - #1data16

CMP Rm,dir8

[Encoding]

B	**E**	ssss	**1**	addr7-addr0

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

CMP
(Rm) - (dir8)

CMP WRj,dir8

[Encoding]

B	**E**	tttt	**5**	addr7-addr0

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

CMP
(WRj) - (dir8)

CMP Rm,dir16

[Encoding]

B	**E**	ssss	**3**	addr15-addr8	addr7-addr0

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

CMP
(Rm) - (dir8)

CMP WRj,dir16

[Encoding]

B	**E**	tttt	**7**	addr15-addr8	addr7-addr0

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

CMP
(WRj) - (dir16)

CMP Rm,@WRj

[Encoding]

B	**E**	tttt	**9**	ssss	0000

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

CMP
(Rm) - ((WRj))

CMP Rm,@DRk

[Encoding]

B	**E**	uuuu	**B**	ssss	**0**

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

CMP
(Rm) - ((DRk))

CPL A

Function:

Complement accumulator

Description:

Logically complements (∅) each bit of the accumulator (one’s complement). Clear bits which are set and set bits which are cleared.

FLAGS :

CY	AC	OV	N	Z
–	–	–	✓	✓

Example :

The accumulator contains 5Ch (01011100B). After executing the instruction CPL A the accumulator contains 0A3h (10100011B).

[Encoding]

F	4

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [Encoding]

Operation:

CPL
(A) ← ∅(A)

CPL bit

Function:

Complement bit

Description:

Complements (∅) the specified bit variable. A clear bit is set, and a set bit is cleared. CPL can operate on the CY or any directly addressable bit.

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

FLAGS : Only for instructions with CY as the operand.

CY	AC	OV	N	Z
✓	–	–	–	–

Example :

Port 1 contains 5Bh (01011101B). After executing the instruction sequence CPL P1.1 CPL P1.2 Port 1 contains 5Bh (01011011B).

CPL bit51

[Encoding]

B	**2**	bit addr

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [Encoding]

Operation:

CPL
(bit51) ← ∅(bit51)

CPL CY

[Encoding]

B	**3**

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [Encoding]

Operation:

CPL
(CY) ← ∅(CY)

CPL bit

[Encoding]

A	**9**	**B**	0yyy	bit addr

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

CPL
(bit) ← ∅(bit)

DA A

Function:

Decimal–adjust accumulator for addition

Description:

Adjusts the 8–bit value in the accumulator that resulted from the earlier addition of two variables (each in packed–BCD format), producing two 4–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 set, six is added to the accumulator, producing the proper BCD digit in the low nibble. This internal addition sets the CY flag if a carry out of the lowest 4 bits propagated through all higher bits, but it does not clear the CY flag otherwise.

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

All of this occurs during one instruction cycle. Essentially, this instruction performs the decimal conversion by adding 00h, 06h, 60h or 66h 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.

FLAGS :

CY	AC	OV	N	Z
✓	_	_	3	3

Example :

The accumulator contains 56h (01010110B), which represents the packed BCD digits of the decimal number 56. Register 3 contains 67h (01100111B), which represents the packed BCD digits of the decimal number 67. The CY flag is set. After executing the instruction sequence ADDC A,R3.

DA A the accumulator contains 0BEh (10111110B) and the CY and AC flags are clear. The Decimal Adjust instruction then alters the accumulator to the value 24h (00100100B), indicating the packed BCD digits of the decimal number 24, the lower two digits of the decimal sum of 56, 67, and the carry–in. The CY flag is set by the Decimal Adjust instruction, indicating that a decimal overflow occurred. The true sum of 56, 67 and 1 is 124. BCD variables can be incremented or decremented by adding 01h or 99h. If the accumulator contains 30h (representing the digits of 30 decimal), then the instruction sequence:

ADD A, #99h.

DA A

leaves the CY flag set and 29h in the accumulator, since 30 + 99 = 129. The low byte of the sum can be interpreted to mean 30 - 1 = 29.

DA A

[Encoding]

D	4

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [Encoding]

Operation:

```
DA
(Contents of accumulator are BCD)
IF  [[(A.3:0) > 9] V [(AC) = 1]]
    THEN (A.3:0) ← (A.3:0) + 6
        AND
    IF [[(A.7:4) > 9] V [(CY) = 1]]
    THEN (A.7:4) ← (A.7:4) + 6
```

DEC byte

Function:

Decrement

Description:

Decrements the specified byte variable by 1. An original value of 00h underflows to 0FFh. Four operands 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 is read from the output data latch, not the input pins.

FLAGS :

CY	AC	OV	N	Z
–	–	–	✓	✓

Example :

Register 0 contains 7Fh (01111111B). On–chip RAM locations 7Eh and 7Fh contain 00h and 40h, respectively. After executing the instruction sequence:

```
DEC @R0
DEC R0
DEC @R0
```

register 0 contains 7Eh and on–chip RAM locations 7Eh and 7Fh are set to 0FFh and 3Fh, respectively.

DEC A

[Encoding]

1	**4**

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [Encoding]

Operation:

DEC
(A) ← (A) - 1

DEC dir8

[Encoding]

1	**5**	addr7-addr0

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [Encoding]

Operation:

DEC
(dir8) ← (dir8) - 1

DEC @Ri

[Encoding]

1	011i

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [A5][Encoding]

Operation:

DEC
((Ri)) ← ((Ri)) - 1

DEC Rn

[Encoding]

1	1rrr

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [A5][Encoding]

Operation:

DEC
(Rn) ← (Rn) - 1

DEC <dest>,<src>

Function:

Decrement

Description:

Decrements the specified variable at the destination operand by 1, 2 or 4. An original value of 00h underflows to 0FFh.

FLAGS :

CY	AC	OV	N	Z
–	–	–	✓	✓

Example :

Register 0 contains 7Fh (01111111B). After executing the instruction sequence DEC R0,#1 register 0 contains 7Eh.

DEC Rm,#short

[Encoding]

1	**B**	ssss	00vv

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

DEC
(Rm) ← (Rm) - #short

DEC WRj,#short

[Encoding]

1	**B**	tttt	01vv

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

DEC
(WRj) ← (WRj) - #short

DEC DRk,#short

[Encoding]

1	**B**	uuuu	11vv

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

DEC
(DRk) ← (DRk) - #short

DIV <dest>,<src>

Function:

Divide

Description:

Divides the unsigned integer in the register by the unsigned integer operand in register addressing mode and clears the CY and OV flags.

For byte operands (<dest>,<src> = Rmd,Rms) the result is 16 bits. The 8–bit quotient is stored in the higher byte of the word where Rmd resides; the 8–bit remainder is stored in the lower byte of the word where Rmd resides. For example: register 1 contains 251 (0FBh or 11111011B) and register 5 contains 18 (12h or 00010010B). After executing the instruction DIV R1,R5 register 0 contains 13 (0Dh or 00001101B); register 1 contains 17 (11h or 00010001B), since 251 = (13 x 18) + 17; and the CY and OV bits are clear (See Flags).

FLAGS : The CY flag is cleared. The N flag is set if the MSB of the quotient is set. The Z flag is set if the quotient is zero.:

CY	AC	OV	N	Z
0	✓	✓	✓	✓

Exception: if <src> contains 00h, the values returned in both operands are undefined; the CY flag is cleared, OV flag is set, and the rest of the flags are undefined.

FLAGS :

CY	AC	OV	N	Z
0	?	1	?	?

DIV Rmd,Rms

[Encoding]

8	**C**	ssss	SSSS

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

DIV (8–bit operands)
(Rmd) ← remainder (Rmd) / (Rms)
if <dest>md = 0,2,4,..,14
(Rmd+1) ← quotient (Rmd) / (Rms)
(Rmd-1) ← remainder (Rmd) / (Rms)
if <dest> md = 1,3,5,..,15
(Rmd) ← quotient (Rmd) / (Rms)

DIV WRjd,WRjs

[Encoding]

8	D	tttt	TTTT

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

DIV (16–bit operands)
(WRjd) ← remainder (WRjd) / (WRjs)
if <dest> jd = 0, 4, 8, ... 28
(WRjd+2) ← quotient (WRjd) / (WRjs)
(WRjd-2) ← remainder (WRjd) / (WRjs))
if <dest> jd = 2, 6, 10, ... 30
(WRjd) ← quotient (WRjd) / (WRjs

For example, for a destination register WR4, assume the quotient is 1122h and the remainder is 3344h. Then, the results are stored in these register file locations:

Location	4	5	6	7
Contents	33h	44h	11h	22h

DIV AB

Function:

Divide

Description:

Divides the unsigned 8–bit integer in the accumulator by the unsigned 8–bit integer in register B. The accumulator receives the integer part of the quotient; register B receives the integer remainder. The CY and OV flags are cleared.

Exception: if register B contains 00h, the values returned in the accumulator and register B are undefined; the CY flag is cleared and the OV flag is set.

FLAGS :

CY	AC	OV	N	Z
0	✓	✓	✓	✓

For division by zero**:**

FLAGS :

CY	AC	OV	N	Z
0	?	1	?	?

Example :

The accumulator contains 251 (0FBh or 11111011B) and register B contains 18 (12h or 00010010B). After executing the instruction DIV AB the accumulator contains 13 (0Dh or 00001101B); register B contains 17 (11h or 00010001B), since 251 = (13 x 18) + 17; and the CY and OV flags are clear.

DIV A B

[Encoding]

8	**4**

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [Encoding]

Operation:

DIV
(A) ← quotient (A)/(B)
(B) ← remainder (A)/(B)

DJNZ <byte>,<rel-addr>

Function:

Decrement and jump if not zero

Description:

Decrements the specified location by 1 and branches to the address specified by the second operand if the resulting value is not zero. An original value of 00h underflows to 0FFh. The branch destination is 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 is read from the output data latch, not the input pins.

FLAGS :

CY	AC	OV	N	Z
–	–	–	✓	✓

Example :

The on–chip RAM locations 40h, 50h, and 60h contain 01h, 70h, and 15h, respectively. After executing the following instruction sequence:

```
DJNZ 40h,LABEL1
DJNZ 50h,LABEL2
DJNZ 60h,LABEL
```

on–chip RAM locations 40h, 50h, and 60h contain 00h, 6Fh, and 15h, respectively, and program execution continues at label LABEL2. (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,

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

toggles P1.7 eight times, causing four output pulses to appear at bit 7 of output Port 1. Each pulse lasts three states: two for DJNZ and one to alter the pin.

DJNZ dir8,rel

[Encoding]

D	5	addr7-addr0	rel addr

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [Encoding]

Operation:

```
DJNZ
(PC) ← (PC) + 3
(dir8) ← (dir8) - 1
IF [[(dir8) > 0] or [(dir8) < 0]]
THEN
(PC) ← (PC) + rel
```

DJNZ Rn,rel

[Encoding]

D	1rrr	rel addr

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

```
DJNZ
(PC) ← (PC) + size(instr)
(Rn) ← (Rn) - 1
IF [[(Rn) > 0] or [(Rn) < 0]]
THEN
(PC) ← (PC) + rel
```

ECALL <dest>

Function:

Extended call

Description:

Calls a subroutine located at the specified address. The instruction adds four to the program counter to generate the address of the next instruction and then PUSHes the 24–bit result onto the stack (high byte first), incrementing the stack pointer by three. The 8 bits of the high word and the 16 bits of the low word of the PC are then loaded, respectively, with the second, third and fourth bytes of the ECALL instruction. Program execution continues with the instruction at this address. The subroutine may therefore begin anywhere in the full 16–Mbyte memory space.

FLAGS :

CY	AC	OV	N	Z
–	–	–	–	–

Example :

The stack pointer contains 07h and the label ”SUBRTN” is assigned to program memory location 123456h. After executing the instruction ECALL SUBRTN at location 054321h, SP contains 0Ah; on–chip RAM locations 08h, 09h and 0Ah contain 05h, 43h and 21h, respectively; and the PC contains 123456h.

ECALL addr24

[Encoding]

9	**A**	addr23- addr16	addr15-addr8	addr7-addr0

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

ECALL
(PC) ← (PC) + size(instr)
(SP) ← (SP) + 1
((SP)) ← (PC.23:16)
(SP) ← (SP) + 1
((SP)) ← (PC.15:8)
(SP) ← (SP) + 1
((SP)) ← (PC.7:0)
(PC) ← (addr.23:0)

ECALL @DRk

[Encoding]

9	**9**	uuuu	**8**

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

ECALL
(PC) ← (PC) + size(instr)
(SP) ← (SP) + 1
((SP)) ← (PC.23:16)
(SP) ← (SP) + 1
((SP)) ← (PC.15:8)
(SP) ← (SP) + 1
((SP)) ← (PC.7:0)
(PC) ← ((DRk))

EJMP <dest>

Function:

Extended jump

Description:

Causes an unconditional branch to the specified address by loading the 8 bits of the high order and 16 bits of the low order words of the PC with the second, third, and fourth instruction bytes. The destination may be therefore be anywhere in the full 16–Mbyte memory space.

FLAGS :

CY	AC	OV	N	Z
–	–	–	–	–

Example :

The label ”JMPADR” is assigned to the instruction at program memory location 123456h. The instruction is EJMP JMPADR

EJMP addr24

[Encoding]

8	**A**	addr23- addr16	addr15-addr8	addr7-addr0

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

EJMP
(PC)← (addr.23:0)

EJMP @DRk

[Encoding]

8	**9**	uuuu	**8**

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

EJMP
(PC) ← ((DRk))

ERET

Function:

Extended return

Description:

POPs byte 2, byte 1, and byte 0 of the 3–byte PC successively from the stack and decrements the stack pointer by 3. Program execution continues at the resulting address, which normally is the instruction immediately following ECALL.

FLAGS :

CY	AC	OV	N	Z
–	–	–	–	–

Example :

The stack pointer contains 0Bh. On–chip RAM locations 08h, 09h and 0Ah contain 01h, 23h and 49h, respectively. After executing the instruction ERET the stack pointer contains 08h and program execution continues at location 012349h.

ERET

[Encoding]

A	A

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

ERET
(PC.7:0) ← ((SP))
(SP) ← (SP) - 1
(PC.15:8) ← ((SP))
(SP) ← (SP) - 1
(PC.23:16) ← ((SP))
(SP) ← (SP) - 1

INC <Byte>

Function:

Increment

Description:

Increments the specified byte variable by 1. An original value of 0FFh overflows to 00h. Three addressing modes are allowed for 8–bit operands: register, direct, or register–indirect.

Note :
When this instruction is used to modify an output Port, the value used as the original Port data is read from the output data latch, not the input pins.

FLAGS :

CY	AC	OV	N	Z
–	–	–	✓	✓

Example :

Register 0 contains 7Eh (011111110B) and on–chip RAM locations 7Eh and 7Fh contain 0FFh and 40h, respectively. After executing the instruction sequence:

```
INC @R0
INC R0
INC @R0
```

register 0 contains 7Fh and on–chip RAM locations 7Eh and 7Fh contain 00h and 41h, respectively.

INC A

[Encoding]

0	**4**

Hex Code in:	**Operation:**
Binary Mode = [Encoding] Source Mode = [Encoding]	INC (A) ← (A) + 1

INC dir8

[Encoding]

0	**5**	addr7-addr0

Hex Code in:	**Operation:**
Binary Mode = [Encoding] Source Mode = [Encoding]	INC (dir8) ← (dir8) + 1

INC @Ri

[Encoding]

0	011i

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [A5][Encoding]

Operation:

INC
((Ri) ← ((Ri)) + 1

INC Rn

[Encoding]

0	1rrr

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [A5][Encoding]

Operation:

INC
(Rn) ← (Rn) + 1

INC <dest>,<src>

Function:

Increment

Description:

Increments the specified variable by 1, 2 or 4. An original value of 0FFh overflows to 00h.

FLAGS :

CY	AC	OV	N	Z
–	–	–	✓	✓

Example :

Register 0 contains 7Eh (011111110B). After executing the instruction INC R0,#1 register 0 contains 7Fh.

INC Rm,#short

[Encoding]

0	**B**	ssss	00vv

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

INC
(Rm) ← (Rm) + #short

INC WRj,#short

[Encoding]

0	**B**	tttt	01vv

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

INC
(WRj) ← (WRj) + #short

INC DRk,#short

[Encoding]

0	**B**	uuuu	11vv

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

INC
(DRk) ← (DRk) + #shortdata pointer

INC DPTR

Function:

Increment data pointer

Description:

Increments the 16–bit data pointer by one. A 16–bit increment (modulo 2^{16}) is performed; an overflow of the low byte of the data pointer (DPL) from 0FFh to 00h increments the high byte of the data pointer (DPH) by one. An overflow of the high byte (DPH) does not increment the high word of the extended data pointer (DPX = DR56).

FLAGS :

CY	AC	OV	N	Z
–	–	–	✓	✓

Example :

Registers DPH and DPL contain 12h and 0FEh, respectively. After the instruction sequence:

```
INC DPTR
INC DPTR
INC DPTR
```

DPH and DPL contain 13h and 01h, respectively.

INC DPTR

[Encoding]

A	3

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [Encoding]

Operation:

INC
(DPTR) ← (DPTR) + 1

JB <bit>,rel

Function:

Jump if bit set

Description:

If the specified bit is a one, jump to the address specified; 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.

FLAGS :

CY	AC	OV	N	Z
–	–	–	–	–

Example :

Input Port 1 contains 11001010B and the accumulator contains 56h (01010110B). After the instruction sequence:

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

program execution continues at label LABEL2.

Variations

JB bit51,rel

2	**0**	bit addr	rel addr

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [Encoding]

Operation:

```
JB
(PC) ← (PC) + 3
IF (bit51) = 1
    THEN
    (PC) ← (PC) + rel
```

JB bit,rel

[Encoding]

A	**9**	**2**	0yyy	bit addr	rel addr

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

```
JB
(PC) ← (PC) + size(instr)
IF [(bit) = 1]
    THEN
    (PC) ← (PC) + rel
```

JBC <bit>,rel

Function:

Jump if bit is set and clear bit.

Description:

If the specified bit is one, branch to the specified address; otherwise proceed with the next instruction. The bit is not cleared if it is already a zero. 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.

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

FLAGS :

CY	AC	OV	N	Z
!	_	_	_	_

Example :

The accumulator contains 56h (01010110B). After the instruction sequence:

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

the accumulator contains 52h (01010010B) and program execution continues at label LABEL2.

JBC bit51,rel

[Encoding]

1	0	bit addr	rel addr

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [Encoding]

Operation:

```
JBC
(PC) ← (PC) + 3
IF [(bit51) = 1]
    THEN
    (bit51) ← 0
    (PC) ← (PC) + rel
```

JBC bit,rel

[Encoding]

A	**9**	**1**	0yyy	bit addr	rel addr

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

```
JBC
(PC) ← (PC) + size(instr)
IF [(bit51) = 1]
    THEN
    (bit51) ← 0
    (PC) ← (PC) + rel
```

JC rel

Function:

Jump if carry is set

Description:

If the CY flag is set, branch to the address specified; 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.

FLAGS :

CY	AC	OV	N	Z
!	_	_	_	_

Example :

The CY flag is clear. After the instruction sequence:

```
JC      LABEL1
CPL     CY
JC      LABEL 2
```

the CY flag is set and program execution continues at label LABEL2.

JC rel

[Encoding]

4	**0**	rel addr

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [Encoding]

Operation:

```
JC
(PC) ← (PC) + 2
IF [(CY) = 1]
    THEN
    (PC) ← (PC) + rel
```

JE rel

Function:

Jump if equal

Description:

If the Z flag is set, branch to the address specified; 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.

FLAGS :

CY	AC	OV	N	Z
_	_	_	_	!

Example :

The Z flag is set. After executing the instruction JE LABEL1 program execution continues at label LABEL1.

JE rel

[Encoding]

6	8	rel addr

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

```
JE
(PC) ← (PC) + size(instr)
IF [(Z) = 1]
    THEN
    (PC) ← (PC) + rel
```

JG rel

Function:

Jump if greater than

Description:

If the Z flag and the CY flag are both clear, branch to the address specified; 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.

FLAGS :

CY	AC	OV	N	Z
_	_	_	!	_

Example :

The instruction JG LABEL1 causes program execution to continue at label LABEL1 if the Z flag and the CY flag are both clear.

JG rel

[Encoding]

3	**8**	rel addr

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

```
JG
(PC) ← (PC) + size(instr)
IF [[(Z) = 0] AND [(CY) = 0]]
    THEN
    (PC) ← (PC) + rel
```

JLE rel

Function:

Jump if less than or equal

Description:

If the Z flag or the CY flag is set, branch to the address specified; 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.

FLAGS :

CY	AC	OV	N	Z
–	–	–	!	!

Example :

The instruction JLE LABEL1 causes program execution to continue at LABEL1 if the Z flag or the CY flag is set.

JLE rel

[Encoding]

2	8	rel addr

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

```
JLE
(PC) ← (PC) + size(instr)
IF [[(Z) = 1] OR [(CY) = 1]]
    THEN
    (PC) ← (PC) + rel
```

JMP @A+DPTR

Function:

Jump indirect

Description:

Adds the 8–bit unsigned content of the accumulator with the 16–bit data pointer and load the resulting sum into the lower 16 bits of the program counter. This is the address for subsequent instruction fetches. The contents of the accumulator and the data pointer are not affected.

FLAGS :

CY	AC	OV	N	Z
–	–	–	–	–

Example :

The accumulator contains an even number from 0 to 6. The following sequence of instructions branchs to one of four AJMP instructions in a jump table starting at JMP_TBL :

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

If the accumulator contains 04h at the start this sequence, execution jumps to LABEL2. Remember that AJMP is a two–byte instruction, so the jump instructions start at every other address.

JMP @A+DPTR

[Encoding]

7	3

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [Encoding]

Operation:

JMP
(PC.15:0) ← (A) + (DPTR)

JNB bit51,rel JNB bit,rel

Function:

Jump if bit not set

Description:

If the specified bit is clear, branch to the specified 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.

FLAGS :

CY	AC	OV	N	Z
–	–	–	–	–

Example :

Input Port 1 contains 11001010B and the accumulator contains 56h (01010110B). After executing the instruction sequence:

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

program execution continues at label LABEL2.

JNB bit51,rel

[Encoding]

3	**0**	bit addr	rel addr

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [Encoding]

Operation:

```
JNB
(PC) ← (PC) + 3
IF [(bit51) = 0]
    THEN
    (PC) ← (PC) + rel
```

JNB bit,rel

[Encoding]

A	**9**	**3**	00yy	bit addr	rel addr

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

```
JNB
(PC) ← (PC) + size(instr)
IF [(bit) = 0]
    THEN
    (PC) ← (PC) + rel
```

JNC rel

Function: Jump if carry not set

Description:

If the CY flag is clear, branch to the address specified; 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 CY flag is not modified.

FLAGS :

CY	AC	OV	N	Z
!	_	_	_	_

Example :

The CY flag is set. The instruction sequence:

```
JNC   LABEL1
CPL   CY
JNC   LABEL2
```

clears the CY flag and causes program execution to continue at label LABEL2.

JNC rel

[Encoding]

5	0	rel addr

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [Encoding]

Operation:

```
JNC
(PC) ← (PC) + 2
IF [(CY) = 0]
    THEN
    (PC) ← (PC) + rel
```

JNE rel

Function:

Jump if not equal

Description:

If the Z flag is clear, branch to the address specified; 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.

FLAGS :

CY	AC	OV	N	Z
–	–	–	–	!

Example :

The instruction JNE LABEL1 causes program execution to continue at LABEL1 if the Z flag is clear.

JNE rel

[Encoding]

7	8	rel addr

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

```
JNE
(PC) ← (PC) + size(instr)
IF [(Z) = 0]
    THEN
    (PC) ← (PC) + rel
```

JNZ rel

Function:

Jump if accumulator not zero

Description:

If any bit of the accumulator is set, branch to the specified 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.

FLAGS :

CY	AC	OV	N	Z
–	–	–	–	!

Example :

The accumulator contains 00h. After executing the instruction sequence:

```
JNZ    LABEL1
INC    A
JNZ    LABEL2
```

the accumulator contains 01h and program execution continues at label LABEL2.

JNZ rel

[Encoding]

7	**0**	rel addr

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [Encoding]

Operation:

```
JNZ
(PC) ← (PC) + 2
IF [(A) ≠ 0]
    THEN
    (PC) ← (PC) + rel
```

JSG rel

Function:

Jump if greater than (signed)

Description:

If the Z flag is clear and the N flag and the OV flag have the same value, branch to the address specified; 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.

FLAGS :

CY	AC	OV	N	Z
_	_	!	!	!

Example :

The instruction JSG LABEL1 causes program execution to continue at LABEL1 if the Z flag is clear and the N flag and the OV flag have the same value.

JSG rel

[Encoding]

1	**8**	rel addr

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

```
JSG
(PC) ← (PC) + size(instr)
IF [(Z) = 0] AND [(N) = (OV)]
    THEN
    (PC) ← (PC) + rel
```

JSGE rel

Function:

Jump if greater than or equal (signed)

Description:

If the N flag and the OV flag have the same value, branch to the address specified; 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.

FLAGS :

CY	AC	OV	N	Z
_	_	!	!	!

Example :

The instruction JSGE LABEL1 causes program execution to continue at LABEL1 if the N flag and the OV flag have the same value.

JSGE rel

[Encoding]

5	**8**	rel addr

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

```
JSGE
(PC) ← (PC) + size(instr)
IF [(N) = (OV)]
    THEN
    (PC) ← (PC) + rel
```

JSL rel

Function:

Jump if less than (signed)

Description:

If the N flag and the OV flag have different values, branch to the address specified; 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.

FLAGS :

CY	AC	OV	N	Z
_	_	!	!	!

Example :

The instruction JSL LABEL1 causes program execution to continue at LABEL1 if the N flag and the OV flag have different values.

JSL rel

[Encoding]

4	8	rel addr

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

```
JSL
(PC) ← (PC) + size(instr)
IF [(N) ≠ (OV)]
    THEN
    (PC) ← (PC) + rel
```

JSLE rel

Function:

Jump if less than or equal (signed)

Description:

If the Z flag is set OR if the the N flag and the OV flag have different values, branch to the address specified; 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.

FLAGS :

CY	AC	OV	N	Z
_	_	!	!	!

Example :

The instruction JSLE LABEL1 causes program execution to continue at LABEL1 if the Z flag is set OR if the the N flag and the OV flag have different values.

JSLE rel

[Encoding]

0	8	rel addr

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

```
JSLE
(PC) ← (PC) + 2
IF [[(Z) = 1] OR [(N) ≠ (OV)]]
    THEN
    (PC) ← (PC) + rel
```

JZ rel

Function:

Jump if accumulator zero

Description:

If all bits of the accumulator are clear (zero), branch to the address specified; 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.

FLAGS :

CY	AC	OV	N	Z
–	–	–	–	!

Example :

The accumulator contains 01h. After executing the instruction sequence:

```
        JZ      LABEL1
        DEC     A
        JZ      LABEL2
```

the accumulator contains 00h and program execution continues at label LABEL2.

JZ rel

[Encoding]

6	0	rel. addr

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [Encoding]

Operation:

```
JZ
(PC) ← (PC) + 2
IF [(A) = 0]
    THEN
    (PC) ← (PC) + rel
```

LCALL <dest>

Function:

Long call

Description:

Calls a subroutine located at the specified 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). The stack pointer is incremented by two. The high and low 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 64–Kbyte region of memory where the next instruction is located.

FLAGS :

CY	AC	OV	N	Z
–	–	–	–	–

Example :

The stack pointer contains 07h and the label ”SUBRTN” is assigned to program memory location 1234h. After executing the instruction LCALL SUBRTN at location 0123h, the stack pointer contains 09h, on–chip RAM locations 08h and 09h contain 01h and 26h and the PC contains 1234h.

LCALL addr16

[Encoding]

1	**2**	addr15-addr8	addr7-addr0

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [Encoding]

Operation:

LCALL
(PC) ← (PC) + 3
(SP) ← (SP) + 1
((SP)) ← (PC.7:0)
(SP) ← (SP) + 1
((SP)) ← (PC.15:8)
(PC) ← (addr.15:0)

LCALL @WRj

[Encoding]

9	**9**	tttt	**4**

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

LCALL
(PC) ← (PC) + size(instr)
(SP) ← (SP) + 1
((SP)) ← (PC.7:0)
(SP) ← (SP) + 1
((SP)) ← (PC.15:8)
(PC) ← (WRj)

LJMP <dest>

Function:

Long Jump

Description:

Causes an unconditional branch to the specified address, by loading the high and low bytes of the PC (respectively) with the second and third instruction bytes. The destination may therefore be anywhere in the 64–Kbyte memory region where the next instruction is located.

FLAGS :

CY	AC	OV	N	Z
–	–	–	–	–

Example :

The label ”JMPADR” is assigned to the instruction at program memory location 1234h. After executing the instruction LJMP JMPADR at location 0123h, the program counter contains 1234h.

LJMP addr16

[Encoding]

0	**2**	addr15–addr8	addr7–addr0

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [Encoding]

Operation:

LJMP
(PC) ← (addr.15:0)

LJMP @WRj

[Encoding]

8	**9**	tttt	**4**

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

LJMP
(PC) ← (WRj)

MOV <dest>,<src>

Function:

Move byte variable

Description:

Copies the byte variable specified by the second operand into the location specified by the first operand. The source byte is not affected.

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

FLAGS :

CY	AC	OV	N	Z
–	–	–	–	–

Example :

On–chip RAM location 30h contains 40h, on–chip RAM location 40h contains 10h, and input Port 1 contains 11001010B (0CAh). After executing the instruction sequence:

```
MOV      R0,#30h     ;R0 < = 30h
MOV      A,@R0       ;A < = 40h
MOV      R1,A   ;R1  < = 40h
MOV      B,@R1       ;B < = 10h
MOV      @R1,P1      ;RAM (40h) < = 0CAh
MOV      P2,P1  ;P2 #0CAh
```

register 0 contains 30h, the accumulator and register 1 contain 40h, register B contains 10h and on–chip RAM location 40h and output Port 2 contain 0CAh (11001010B).

MOV A,#data

[Encoding]

7	**4**	immed data

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [Encoding]

Operation:

MOV
(A) ← #data

MOV dir8,#data

[Encoding]

7	**5**	addr7-addr0	immed data

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [Encoding]

Operation:

MOV
(dir8) ← #data

MOV @Ri,#data

[Encoding]

7	011i	immed data

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [A5][Encoding]

Operation:

MOV
((Ri)) ← #data

MOV Rn,#data

[Encoding]

7	1rrr	immed data

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [A5][Encoding]

Operation:

MOV
(Rn) ← #data

MOV dir8,dir8

[Encoding]

8	**5**	addr7-addr0$_s$	addr7-addr0$_d$

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [Encoding]

Operation:

MOV
(dir8) ← (dir8)

MOV dir8,@Ri

[Encoding]

8	011i	addr7-addr0

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [A5][Encoding]

Operation:

MOV
(dir8) ← ((Ri))

MOV dir8,Rn

[Encoding]

8	1rrr	addr7-addr0

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [A5][Encoding]

Operation:

MOV
(dir8) ← (Rn)

MOV @Ri,dir8

[Encoding]

A	011i	addr7-addr0

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [A5][Encoding]

Operation:

MOV
((Ri)) ← (dir8)

MOV Rn,dir8

[Encoding]

A	1rrr	addr7-addr0

Hex Code in:	**Operation:**
Binary Mode = [Encoding] Source Mode = [A5][Encoding]	MOV (Rn) ← (dir8)

MOV A,dir8

[Encoding]

E	**5**	addr7-addr0

Hex Code in:	**Operation:**
Binary Mode = [Encoding] Source Mode = [Encoding]	MOV (A) ← (dir8)

MOV A,@Ri

[Encoding]

E	011i

Hex Code in:	**Operation:**
Binary Mode = [Encoding] Source Mode = [A5][Encoding]	MOV (A) ← ((Ri))

MOV A,Rn

[Encoding]

E	1rrr

Hex Code in:	**Operation:**
Binary Mode = [Encoding] Source Mode = [A5][Encoding]	MOV (A) ← (Rn)

MOV dir8,A

[Encoding]

F	**5**	addr7-addr0

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [Encoding]

Operation:

MOV
(dir8) ← (A)

MOV @Ri,A

[Encoding]

F	011i

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [A5][Encoding]

Operation:

MOV
((Ri)) ← (A)

MOV Rn,A

[Encoding]

F	1rrr

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [A5][Encoding]

Operation:

MOV
(Rn) ← (A)

MOV Rmd,Rms

[Encoding]

7	**C**	ssss	SSSS

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

MOV
(Rmd) ← (Rms)

MOV WRjd,WRjs

[Encoding]

3	**D**	tttt	TTTT

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

MOV
(WRjd) ← (WRjs)

MOV DRkd,DRks

[Encoding]

7	**F**	uuuu	UUUU

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

MOV
(DRkd) ← (DRks)

MOV Rm,#data

[Encoding]

7	**E**	ssss	**0**	immed data

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

MOV
(Rm) ← #data

MOV WRj,#data16

[Encoding]

7	**E**	tttt	**4**	immed data hi	immed data low

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

MOV
(WRj) ← #data16

MOV DRk,#0data16

[Encoding]

3	**E**	uuuu	**8**	immed data hi	immed data low

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

MOV
(DRk) ← #0data16

MOV DRk,#1data16

[Encoding]

3	**E**	uuuu	**C**	immed data hi	immed data low

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

MOV
(DRk) ← #1data16

MOV Rm,dir8

[Encoding]

7	**E**	ssss	**1**	addr7-addr0

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

MOV
(Rm) ← (dir8)

MOV WRj,dir8

[Encoding]

7	**E**	tttt	**5**	addr7-addr0

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

MOV
(WRj) ← (dir8)

MOV DRk,dir8

[Encoding]

7	**E**	uuuu	**D**	addr7-addr0

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

MOV
(DRk) ← (dir8)

MOV Rm,dir16

[Encoding]

7	**E**	ssss	**3**	addr15-addr8	addr7-addr0

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

MOV
(Rm) ← (dir16)

MOV WRj, dir16

[Encoding]

7	**E**	tttt	**7**	addr15-addr8	addr7-addr0

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

MOV
(WRj ← (dir16)

MOV DRk,dir16

[Encoding]

7	**E**	uuuu	**F**	addr15-addr8	addr7-addr0

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

MOV
(DRk) ← (dir16)

MOV Rm,@WRj

[Encoding]

7	**E**	tttt	**9**	ssss	**0**

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

MOV
(Rm) ← ((WRj))

MOV Rm,@DRk

[Encoding]

7	**E**	uuuu	**B**	ssss	**0**

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

MOV
(Rm) ← ((DRk))

MOV WRjd,@WRjs

[Encoding]

0	**B**	TTTT	**8**	tttt	**0**

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

MOV
(WRjd) ← ((WRjs))

MOV WRj,@DRk

[Encoding]

0	**B**	uuuu	**A**	tttt	**0**

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

MOV
(WRj) ← ((DRk))

MOV dir8,Rm

[Encoding]

7	**A**	ssss	**1**	addr7-addr0

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

MOV
(dir8) ← (Rm)

MOV dir8,WRj

[Encoding]

7	**A**	tttt	**5**	addr7-addr0

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

MOV
(dir8) ← (WRj)

MOV dir8,DRk

[Encoding]

7	**A**	uuuu	**D**	addr7-addr0

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

MOV
(dir8) ← (DRk)

MOV dir16,Rm

[Encoding]

7	**A**	ssss	**3**	addr15-addr8	addr7-addr0

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

MOV
(dir16) ← (Rm)

MOV dir16,WRj

[Encoding]

7	**A**	tttt	**7**	addr15-addr8	addr7-addr0

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

MOV
(dir16) ← (WRj)

MOV dir16,DRk

[Encoding]

7	**A**	uuuu	**F**	addr15-addr8	addr7-addr0

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

MOV
(dir16) ← (DRk)

MOV @WRj,Rm

[Encoding]

7	**A**	tttt	**9**	ssss	**0**

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

MOV
((WRj)) ← (Rm)

MOV @DRk,Rm

[Encoding]

7	**A**	uuuu	**B**	ssss	**0**

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

MOV
((DRk)) ← (Rm)

MOV @WRjd,WRjs

[Encoding]

1	**B**	tttt	**8**	TTTT	**0**

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

MOV
((WRjd)) ← (WRjs)

MOV @DRk,WRj

[Encoding]

1	**B**	uuuu	**A**	tttt	**0**

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

MOV
((DRk)) ← (WRj)

MOV Rm,@WRj + dis16

[Encoding]

0	**9**	ssss	tttt	dis hi	dis low

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

MOV
(Rm) ← ((WRj) + dis16)

MOV WRj,@WRj + dis16

[Encoding]

4	**9**	tttt	TTTT	dis hi	dis low

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

MOV
(WRj) ← ((WRj) + dis16)

MOV Rm,@DRk + dis16

[Encoding]

2	**9**	ssss	uuuu	dis hi	dis low

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

MOV
(Rm) ← ((DRk) + dis24)

MOV WRj,@DRk + dis16

[Encoding]

6	**9**	tttt	uuuu	dis hi	dis low

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

MOV
(WRj) ← ((DRk) + dis24)

MOV @WRj + dis16,Rm

[Encoding]

1	**9**	ssss	tttt	dis hi	dis low

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

MOV
((WRj) + dis16) ← (Rm)

MOV @WRj + dis16,WRj

[Encoding]

5	**9**	TTTT	tttt	dis hi	dis low

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

MOV
((WRj) + dis16) ← (WRj)

MOV @DRk + dis16,Rm

[Encoding]

3	**9**	ssss	uuuu	dis hi	dis low

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

MOV
((DRk) + dis24) ← (Rm)

MOV @DRk + dis16,WRj

[Encoding]

7	**9**	tttt	uuuu	dis hi	dis low

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

MOV
((DRk) + dis24) ← (WRj)

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

Function

Move bit data

Description

Copies the boolean variable specified by the second operand into the location specified by the first operand. One of the operands must be the CY flag; the other may be any directly addressable bit. Does not affect any other register.

FLAGS :

CY	AC	OV	N	Z
✓	–	–	–	–

Example :

The CY flag is set, input Port 3 contains 11001001B and output Port 1 contains 35h (00110101B). After executing the instruction sequence:

```
MOV P1.3,CY
MOV CY,P3.3
MOV P1.2,CY
```

the CY flag is clear and Port 1 contains 39h (00111001B).

MOV bit51,CY

[Encoding]

9	**2**	bit addr

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [Encoding]

Operation:

MOV
(bit51) ← (CY)

MOV CY,bit51

[Encoding]

A	**2**	bit addr

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [Encoding]

Operation:

MOV
(CY) ← (bit51)

MOV bit,CY

[Encoding]

A	**9**	**9**	0yyy	bit addr

Hex Code in:	**Operation:**
Binary Mode = [A5][Encoding] Source Mode = [Encoding]	MOV (bit) ← (CY)

MOV CY,bit

[Encoding]

A	**9**	**A**	0yyy	bit addr

Hex Code in:	**Operation:**
Binary Mode = [A5][Encoding] Source Mode = [Encoding]	MOV (CY) ← (bit)

MOV DPTR,#data16

Function:

Load data pointer with a 16–bit constant

Description:

Loads the 16–bit data pointer (DPTR) with the specified 16–bit constant. The high byte of the constant is loaded into the high byte of the data pointer (DPH). The low byte of the constant is loaded into the low byte of the data pointer (DPL).

FLAGS :

CY	AC	OV	N	Z
–	–	–	–	–

Example :

After executing the instruction MOV DPTR,#1234h DPTR contains 1234h (DPH contains 12h and DPL contains 34h).

MOV DPTR,#data16

[Encoding]

9	0	immed data hi	immed data low

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [Encoding]

Operation:

MOV
(DPTR) ← #data16

MOVC A,@A+<base-reg>

Function:

Move code byte

Description:

Loads the accumulator with a code byte or constant from program memory. The address of the byte fetched is the sum of the original unsigned 8–bit accumulator contents and the contents of a 16–bit base register, which may be the 16 LSBs of the data pointer or PC. In the latter case, the PC is incremented to the address of the following instruction before being added with the accumulator; otherwise the base register is not altered. 16–bit addition is performed.

FLAGS :

CY	AC	OV	N	Z
–	–	–	–	–

Example :

The accumulator contains a number between 0 and 3. The following instruction sequence translates the value in the accumulator to one of four values defined by the DB (define byte) directive.

```
RELPC:    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 01h, it returns with 77h 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+PC

[Encoding]

8	3

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [Encoding]

Operation:

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

MOVC A,@A+DPTR

[Encoding]

9	3

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [Encoding]

Operation:

MOVC
(A) ← ((A) + (DPTR))

MOVH DRk(hi),#data16

Function:

Move immediate 16–bit data to the high word of a dword (double–word) register.

Description:

Moves 16–bit immediate data to the high word of a dword (32–bit) register. The low word of the dword register is unchanged.

FLAGS :

CY	AC	OV	N	Z
–	–	–	–	–

Example :

The dword register DRk contains 5566 7788h. After the instruction MOVH DRk,#1122h executes, DRk contains 1122 7788h.

MOVH DRk(hi),#data16

[Encoding]

7	A	uuuu	C	immed data hi	immed data low

Hex Code in:

Binary Mode =[A5] [Encoding]
Source Mode = [Encoding]

Operation:

MOVH
(DRk).31-16 ← #data16

MOVS WRj,Rm

Function:

Move 8–bit register to 16–bit register with sign extension

Description:

Moves the contents of an 8–bit register to the low byte of a 16–bit register. The high byte of the 16–bit register is filled with the sign extension, which is obtained from the MSB of the 8– bit source register.

FLAGS :

CY	AC	OV	N	Z
–	–	–	–	–

Example :

8–bit register Rm contains 055h (01010101B) and the 16–bit register WRj contains 0FFFFh (11111111 11111111B). The instruction MOVS WRj,Rm moves the contents of register Rm (01010101B) to register WRj (i.e., WRj contains 00000000 01010101B).

MOVS WRj, Rm

[Encoding]

1	A	tttt	ssss

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

MOVS
(WRj).7-0 ← (Rm).7-0
(WRj).15-8 ← MSB

MOVX <dest>,<src>

Function:

Move external

Description:

Transfers data between the accumulator and a byte in external data RAM. There are two types of instructions. One provides an 8–bit indirect address to external data RAM; the second provides a 16–bit indirect address to external data RAM.

In the first type of MOVX instruction, the contents of R0 or R1 in the current register bank provides an 8–bit address on Port 0. 8 bits are sufficient for external I/O expansion decoding or for a relatively small RAM array. For larger arrays, any Port pins can be used to output higher address bits. These pins would be controlled by an output instruction preceding the MOVX.

In the second type of MOVX instruction, the data pointer generates a 16–bit address. Port 2 outputs the upper 8 address bits (from DPH) while Port 0 outputs the lower 8 address bits (from DPL).

For both types of moves in nonpage mode, the data is multiplexed with the lower address bits on Port 0. In page mode, the data is multiplexed with the contents of P2 on Port 2 (8–bit address) or with the upper address bits on Port 2 (16–bit address).

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

FLAGS :

CY	AC	OV	N	Z
–	–	–	–	–

Example :

The TSC80251 Microcontroller is operating in nonpage mode. An external 256–byte RAM using multiplexed address/data lines (e.g., an Intel 8155 RAM/I/O/Timer) is connected to Port 0. Port 3 provides control lines for the external RAM. Ports 1 and 2 are used for normal I/O. R0 and R1 contain 12h and 34h. Location 34h of the external RAM contains 56h. After executing the instruction sequence:

```
MOVX A,@R1
MOVX @R0,A
```

the accumulator and external RAM location 12h contain 56h.

MOVX A,@DPTR

[Encoding]

E	0

Hex Code in:	**Operation:**
Binary Mode = [Encoding] Source Mode = [Encoding]	MOVX (A) ← ((DPTR))

MOVX A,@Ri

[Encoding]

E	001i

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [Encoding]

Operation:

MOVX
(A) ← ((Ri))

MOVX @DPTR,A

[Encoding]

F	**0**

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [Encoding]

Operation:

MOVX
((DPTR)) ← (A)

MOVX @Ri,A

[Encoding]

F	001i

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [Encoding]

Operation:

MOVX
((Ri)) ← (A)

MOVZ WRj,Rm

Function:

Move 8–bit register to 16–bit register with zero extension

Description:

Moves the contents of an 8–bit register to the low byte of a 16–bit register. The upper byte of the 16–bit register is filled with zeros.

FLAGS :

CY	AC	OV	N	Z
–	–	–	–	–

Example :

8–bit register Rm contains 055h (01010101B) and 16–bit register WRj contains 0FFFFh (11111111 11111111B). The instruction MOVZ WRj,Rm moves the contents of register Rm (01010101B) to register WRj. At the end of the operation, WRj contains 00000000 01010101B.

MOVZ WRj,Rm

[Encoding]

0	A	tttt	ssss

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

MOV
(WRj).7-0 ← (Rm).7-0
(WRj).15-8 ← 0

MUL <dest>,<src>

Function:

Multiply

Description:

Multiplies the unsigned integer in the source register with the unsigned integer in the destination register. Only register addressing is allowed.

For 8–bit operands, the result is 16 bits. The most significant byte of the result is stored in the low byte of the word where the destination register resides. The least significant byte is stored in the following byte register. The OV flag is set if the product is greater than 255 (0FFh); otherwise it is cleared.

For 16–bit operands, the result is 32 bits. The most significant word is stored in the low word of the the dword where the destination register resides. The least significant word is stored in the following word register. In this operation, the OV flag is set if the product is greater than 0FFFFh, otherwise it is cleared. The CY flag is always cleared. The N flag is set when the MSB of the result is set. The Z flag is set when the result is zero.

FLAGS :

CY	AC	OV	N	Z
0	_	✓	✓	✓

Example :

Register R1 contains 80 (50h or 10010000B) and register R0 contains 160 (0A0h or 10010000B). After executing the instruction MUL R1,R0 which gives the product 12800 (3200h), register R0 contains 32h (00110010B), register R1 contains 00h, the OV flag is set and the CY flag is clear.

MUL Rmd,Rms

[Encoding]

A	C	ssss	SSSS

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

MUL (8–bit operands)
if <dest> md = 0, 2, 4, .., 14
Rmd ← high byte of the Rmd x Rms
Rmd+1 ← low byte of the Rmd x Rms
if <dest> md = 1, 3, 5, .., 15
Rmd-1 ← high byte of the Rmd x Rms
Rmd ← low byte of the Rmd x Rms

MUL WRjd,WRjs

[Encoding]

A	D	tttt	TTTT

Hex Code in:

Binary Mode =[A5][Encoding]
Source Mode = [Encoding]

Operation:

MUL (16–bit operands)
if <dest> jd = 0, 4, 8, .., 28
WRjd ← high word of the WRjd x WRjs
WRjd+2 ← low word of the WRjd x WRjs
if <dest> jd = 2, 6, 10, .., 30
WRjd-2 ← high word of the WRjd x WRjs
WRjd← low word of the WRjd x WRjs

MUL AB

Function:

Multiply

Description:

Multiplies the unsigned 8–bit integers in the accumulator and register B. The low byte of the 16–bit product is left in the accumulator, and the high byte is left in register in B. If the product is greater than 255 (0FFh) the OV flag is set; otherwise it is clear. The CY flag is always clear.

FLAGS :

CY	AC	OV	N	Z
0	–	✓	✓	✓

Example :

The accumulator contains 80 (50h) and register B contains 160 (0A0h). After executing the instruction MUL AB which gives the product 12800 (3200h), register B contains 32h (00110010B), the accumulator contains 00h, the OV flag is set and the CY flag is clear.

MUL AB

[Encoding]

A	**4**

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [Encoding]

Operation:

MUL
(A) ← low byte of (A) X (B)
(B) ← high byte of (A) X (B)

NOP

Function:

No operation

Description:

Execution continues at the following instruction. Affects the PC register only.

FLAGS :

CY	AC	OV	N	Z
–	–	–	–	–

Example :

We assume we are executing an internal code and you want to produce a low–going output pulse on bit 7 of Port 2 that lasts exactly 11 states. A simple CLR–SETB sequence generates an eight–state pulse. (Each instruction requires four states to write to a Port SFR.) You can insert three additional states (if no interrupts are enabled) with the following instruction sequence :

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

NOP

[Encoding]

0	0

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [Encoding]

Operation:

NOP
(PC) ← (PC) + 1

ORL <dest> <src>

Function:

Logical–OR for byte variables

Description:

Performs the bitwise logical–OR operation (V) between the specified variables, storing the results in the destination operand.

The destination operand can be a register, an accumulator or direct address.

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

Note:
When this instruction is used to modify an output Port, the value used as the original Port data is read from the output data latch, not the input pins.

FLAGS :

CY	AC	OV	N	Z
–	–	–	✓	✓

Example :

The accumulator contains 0C3h (11000011B) and R0 contains 55h (01010101B). After executing the instruction, ORL A, R0 the accumulator contains 0D7h (11010111B).

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 a constant data value in the instruction or a variable computed in the accumulator at run time. After executing the instruction ORL P1, #00110010B sets bits 5, 4 and 1 of output Port 1.

ORL dir8,A

[Encoding]

4	**2**	addr7-addr0

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [Encoding]

Operation:

ORL
(dir8) ← (dir8) V (A)

ORL dir8,#data

[Encoding]

4	**3**	addr7-addr0	immed data

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [Encoding]

Operation:

ORL
(dir8) ← (dir8) V #data

ORL A,#data

[Encoding]

4	**4**	immed data

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [Encoding]

Operation:

ORL
(A) ← (A) V #data

ORL A,dir8

[Encoding]

4	**5**	addr7-addr0

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [Encoding]

Operation:

ORL
(A) ← (A) V (dir8)

ORL A,@Ri

[Encoding]

4	011i

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [A5][Encoding]

Operation:

ORL (A) ← (A) V ((Ri))

ORL A,Rn

[Encoding]

4	1rrr

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [A5][Encoding]

Operation:

ORL
(A) ← (A) V (Rn)

ORL Rmd,Rms

[Encoding]

4	**C**	ssss	SSSS

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

ORL
(Rmd) ← (Rmd) V (Rms)

ORL WRjd,WRjs

[Encoding]

4	**D**	tttt	TTTT

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

ORL
(WRjd)←(WRjd) V (WRjs)

ORL Rm,#data

[Encoding]

4	**E**	ssss	**0**	immed data

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

ORL
(Rm) ← (Rm) V #data

ORL WRj,#data16

[Encoding]

4	**E**	tttt	**4**	immed data hi	immed data low

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

ORL
(WRj) ← (WRj) V #data16

ORL Rm,dir8

[Encoding]

4	**E**	ssss	**1**	addr7-addr0

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

ORL
(Rm) ← (Rm) V (dir8)

ORL WRj,dir8

[Encoding]

4	**E**	tttt	**5**	addr7-addr0

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

ORL
(WRj) ← (WRj) V (dir8)

ORL Rm,dir16

[Encoding]

4	**E**	ssss	**3**	addr15-addr8	addr7-addr0

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

ORL
(Rm) ← (Rm) V (dir16)

ORL WRj,dir16

[Encoding]

4	**E**	tttt	**7**	addr15-addr8	addr7-addr0

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

ORL
(WRj) ← (WRj) V (dir16)

ORL Rm,@WRj

[Encoding]

4	**E**	tttt	**9**	ssss	**0**

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

ORL
(Rm) ← (Rm) V ((WRj))

ORL Rm,@DRk

[Encoding]

4	**E**	uuuu	**B**	ssss	**0**

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

ORL
(Rm) ← (Rm) V ((DRk))

ORL CY,<src–bit>

Function:

Logical–OR for bit variables

Description:

Sets the CY flag if the Boolean value is a logical 1; leaves the CY flag 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.

FLAGS :

CY	AC	OV	N	Z
✓	–	–	–	–

Example :

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

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

ORL CY,bit51

[Encoding]

7	**2**	bit addr

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [Encoding]

Operation:

ORL
(CY) ← (CY) V (bit51)

ORL CY,/bit51

★ If this instruction addresses a Port (Px, x = 0-3), add 1 state.

[Encoding]

A	**0**	bit addr

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [Encoding]

Operation:

ORL
(CY) ← (CY) V¬ (bit51)

ORL CY,bit

[Encoding]

A	9	7	0yyy	bit addr

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

ORL
(CY) ← (CY) V (bit)

ORL CY,/bit

[Encoding]

A	9	F	0yyy	bit addr

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

ORL
(CY) ← (CY) V 0 (bit)

POP <src>

Function:

Pop from stack.

Description:

Reads the contents of the on–chip RAM location addressed by the stack pointer, then decrements the stack pointer by one. The value read at the original RAM location is transferred to the newly addressed location, which can be 8–bit or 16–bit.

FLAGS :

CY	AC	OV	N	Z
–	–	–	–	–

Example :

The stack pointer contains 32h and on–chip RAM locations 30h through 32h contain 01h, 23h, and 20h, respectively. After executing the instruction sequence:

```
POP DPH
POP DPL
```

the stack pointer contains 30h and the data pointer contains 0123h. After executing the instruction POP SP the stack pointer contains 20h. Note that in this special case the stack pointer was decremented to 2Fh before it was loaded with the value popped (20h).

POP dir8

[Encoding]

D	0	addr7-addr0

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [Encoding]

Operation:

POP
(dir8) ← ((SP)
(SP) ← (SP) - 1

POP Rm

[Encoding]

D	A	ssss	8

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

POP
(Rm) ← ((SP))
(SP) ← (SP) - 1

POP WRj

[Encoding]

D	A	tttt	9

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

POP
(SP) ← (SP) - 1
(WRj) ← ((SP))
(SP) ← (SP) - 1

POP DRk

[Encoding]

D	A	uuuu	B

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

POP
(SP) ← (SP) - 3
(DRk) ← ((SP))
(SP) ← (SP) - 1

PUSH <dest>

Function:

PUSH onto stack

Description:

Increments the stack pointer by one. The contents of the specified variable are then copied into the on–chip RAM location addressed by the stack pointer.

FLAGS :

CY	AC	OV	N	Z
–	–	–	–	–

Example: On entering an interrupt routine, the stack pointer contains 09h and the data pointer contains 0123h. After executing the instruction sequence:

```
PUSH DPL
PUSH DPH
```

the stack pointer contains 0Bh and on–chip RAM locations 0Ah and 0Bh contain 01h and 23h, respectively.

PUSH dir8

[Encoding]

C	0	addr7-addr0

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [Encoding]

Operation:

PUSH
(SP) ← (SP) + 1
((SP)) ← (dir8)

PUSH #data

[Encoding]

C	A	0	2	immed data

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

PUSH
(SP) ← (SP) + 1
((SP)) ← #data

PUSH #data16

[Encoding]

C	**A**	**0**	**6**	immed data hi	immed data low

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

PUSH
(SP) ← (SP) + 1
((SP)) ← #data16
(SP) ← (SP) + 1

PUSH Rm

[Encoding]

C	**A**	ssss	**8**

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

PUSH
(SP) ← (SP) + 1
((SP)) ← (Rm)

PUSH WRj

[Encoding]

C	**A**	tttt	**9**

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

PUSH
(SP) ← (SP) + 1
((SP)) ← (WRj)
(SP) ← (SP) + 1

PUSH DRk

[Encoding]

C	**A**	uuuu	**B**

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

PUSH
(SP) ← (SP) + 1
((SP)) ← (DRk)
(SP) ← (SP) + 3

RET

Function:

Return from subroutine

Description:

Pops the high and low bytes of the PC successively from the stack, decrementing the stack pointer by two. Program execution continues at the resulting address, which normally is the instruction immediately following ACALL or LCALL.

FLAGS :

CY	AC	OV	N	Z
–	–	–	–	–

Example :

The stack pointer contains 0Bh and on–chip RAM locations 0Ah and 0Bh contain 01h and 23h, respectively. After executing the instruction, RET the stack pointer contains 09h and program execution continues at location 0123h.

RET

[Encoding]

2	2

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [Encoding]

Operation:

RET
(PC).15:8 ← ((SP))
(SP) ← (SP) - 1
(PC).7:0 ← ((SP))
(SP) ← (SP) - 1

RETI

Function:

Return from interrupt

Description:

This instruction pops two or four bytes from the stack, depending on the INTR bit in the CONFIG1 register .

If INTR = 0, RETI pops the high and low bytes of the PC successively from the stack and uses them as the 16–bit return address in region FF:.The stack pointer is decremented by two. No other registers are affected, and neither PSW nor PSW1 is automatically restored to its pre–interrupt status.

If INTR = 1, RETI pops four bytes from the stack: PSW1 and the three bytes of the PC. The three bytes of the PC are the return address, which can be anywhere in the 16–Mbyte memory space. The stack pointer is decremented by four. PSW1 is restored to its pre–interrupt status, but PSW is not restored to its pre–interrupt status. No other registers are affected.

For either value of INTR, hardware restores the interrupt logic to accept additional interrupts at the same priority level as the one just processed. Program execution continues at the return address, which normally is the instruction immediately after the point at which the interrupt request was detected. If an interrupt of the same or lower priority is pending when the RETI instruction is executed, that one instruction is executed before the pending interrupt is processed.

FLAGS :

CY	AC	OV	N	Z
–	–	–	–	–

Example :

INTR = 0. The stack pointer contains 0Bh. An interrupt was detected during the instruction ending at location 0122h. On–chip RAM locations 0Ah and 0Bh contain 01h and 23h, respectively. After executing the instruction, RETI the stack pointer contains 09h and program execution continues at location 0123h.

RETI

[Encoding]

3	2

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [Encoding]

Operation:

Operation for INTR = 0 :
RETI
(PC).15:8 ← ((SP))
(SP) ← (SP) - 1
(PC).7:0 ← ((SP))
(SP) ← (SP) - 1
Operation for INTR = 1 :
RETI
(PC).15:8 ← ((SP))
(SP) ← (SP) - 1
(PC).7:0 ← ((SP))
(SP) ← (SP) - 1
(PC).23:16 ← ((SP))
(SP) ← (SP) - 1
PSW1 ← ((SP))
(SP) ← (SP) - 1

RL A

Function:

Rotate accumulator left

Description:

Rotates the 8 bits in the accumulator one bit to the left. Bit 7 is rotated into the bit 0 position.

FLAGS :

CY	AC	OV	N	Z
–	–	–	✓	✓

Example :

The accumulator contains 0C5h (11000101B). After executing the instruction, RL A the accumulator contains 8Bh (10001011B); the CY flag is unaffected.

RL A

[Encoding]

2	3

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [Encoding]

Operation:

RL
(A).a+1 ← (A).a
(A).0 ← (A).7

RLC A

Function:

Rotate accumulator left through the carry flag

Description:

Rotates the 8 bits in the accumulator and the CY flag one bit to the left. Bit 7 moves into the CY flag position and the original state of the CY flag moves into bit 0 position.

Description:

FLAGS :

CY	AC	OV	N	Z
✓	_	_	✓	✓

Example :

The accumulator contains 0C5h (11000101B) and the CY flag is clear. After executing the instruction RLC A the accumulator contains 8Ah (10001010B) and the CY flag is set.

RLC A

[Encoding]

3	3

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [Encoding]

Operation:

RLC
(A).a+1 ← (A).a
(A).0 ← (CY)
(CY) ← (A) .7

RR A

Function:

Rotate accumulator right

Description:

Rotates the 8 bits in the accumulator one bit to the right. Bit 0 is moved into the bit 7 position.

FLAGS :

CY	AC	OV	N	Z
–	–	–	✓	✓

Example :

The accumulator contains 0C5h (11000101B). After executing the instruction, RR A the accumulator contains 0E2h (11100010B) and the CY flag is unaffected.

RR A

[Encoding]

0	**3**

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [Encoding]

Operation:

RR
(A).a ← (A).a+1
(A).7 ← (A) .0

RRC A

Function:

Rotate accumulator right through carry flag

Description:

Rotates the 8 bits in the accumulator and the CY flag one bit to the right. Bit 0 moves into the CY flag position; the original value of the CY flag moves into the bit 7 position.

FLAGS :

CY	AC	OV	N	Z
–	–	–	✓	✓

Example :

The accumulator contains 0C5h (11000101B) and the CY flag is clear. After executing the instruction RRC A the accumulator contains 62h (01100010B) and the CY flag is set.

RRC A

[Encoding]

1	**3**

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [Encoding]

Operation:

RRC
(A).a ← (A).a+1
(A).7 ← (CY)
(CY) ← (A).0

SETB <bit>

Function:

Set bit

Description:

Sets the specified bit to one. SETB can operate on the CY flag or any directly addressable bit.

FLAGS : No flags are affected except the CY flag for instruction with CY as the operand.

CY	AC	OV	N	Z
✓	–	–	–	–

Example:

The CY flag is clear and output Port 1 contains 34h (00110100B). After executing the instruction sequence:

SETB CY
SETB P1.0

the CY flag is set and output Port 1 contains 35h (00110101B).

SETB bit51

[Encoding]

D	**2**	bit addr

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [Encoding]

Operation:

SETB
(bit51) ← 1

SETB CY

[Encoding]

D	**3**

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [Encoding]

Operation:

SETB
(CY) ← 1

SETB bit

[Encoding]

A	**9**	**D**	0yyy	bit addr

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

SETB
(bit) ← 1

SJMP rel

Function:

Short jump

Description:

Program control branches unconditionally to the specified address. 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.

FLAGS :

CY	AC	OV	N	Z
–	–	–	–	–

Example :

The label ”RELADR” is assigned to an instruction at program memory location 0123h. The instruction SJMP RELADR assembles into location 0100h. After executing the instruction, the PC contains 0123h.

Note :
In the above example, the instruction following SJMP is located at 102h. Therefore, the displacement byte of the instruction is the relative offset (0123h-0102h) = 21h. Put another way, an SJMP with a displacement of 0FEh would be a one–instruction infinite loop.

SJMP rel

[Encoding]

8	0	rel addr

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [Encoding]

Operation:

SJMP
(PC) ← (PC) + 2
(PC) ← (PC) + rel

SLL <src>

Function:

Shift logical left by 1 bit

Description:

Shifts the specified variable to the left by 1 bit, replacing the LSB with zero. The bit shifted out (MSB) is stored in the CY bit.

FLAGS :

CY	AC	OV	N	Z
✓	_	_	✓	✓

Example :

Register 1 contains 0C5h (11000101B). After executing the instruction SLL R 1 register 1 contains 8Ah (10001010B) and CY = 1.

SLL Rm

[Encoding]

3	**E**	ssss	**0**

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

SLL
(Rm).a+1 ← (Rm).a
(Rm).0 ← 0
CY ← (Rm).7

SLL WRj

[Encoding]

3	**E**	tttt	**4**

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

SLL
(WRj).b+1 ← (WRj).b
(WRj).0 ← 0
CY← (WRj).15

SRA <src>

Function:

Shift arithmetic right by 1 bit

Description:

Shifts the specified variable to the arithmetic right by 1 bit. The MSB is unchanged. The bit shifted out (LSB) is stored in the CY bit.

FLAGS :

CY	AC	OV	N	Z
✓	_	_	✓	✓

Example :

Register 1 contains 0C5h (11000101B). After executing the instruction SRA R 1 register 1 contains 0E2h (11100010B) and CY = 1.

SRA Rm

[Encoding]

0	**E**	ssss	**0**

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

SRA
(Rm).7← (Rm).7
(Rm).a ← (Rm).a + 1
CY← (Rm).0

SRA WRj

[Encoding]

0	**E**	tttt	**4**

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

SRA
(WRj).15 ← (WRj).15
(WRj).b ← (WRj).b+1
CY← (WRj).0

SRL <src>

Function:

Shift logical right by 1 bit

Description:

SRL shifts the specified variable to the right by 1 bit, replacing the MSB with a zero. The bit shifted out (LSB) is stored in the CY bit.

FLAGS :

CY	AC	OV	N	Z
✓	_	_	✓	✓

Example :

Register 1 contains 0C5h (11000101B). After executing the instruction SRL R 1 register 1 contains 62h (01100010B) and CY = 1.

SRL Rm

[Encoding]

1	**E**	ssss	**0**

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

SRL
(Rm).7 ← 0
(Rm).a ← (Rm) a + 1
CY← (Rm).0

SRL WRj

[Encoding]

1	**E**	tttt	**4**

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

SRL
(WRj).15 ← 0
(WRj).b ← (WRj).b+1
CY← (WRj).0

SUB <dest>,<src>

Function:

Subtract

Description:

Subtracts the specified variable from the destination operand, leaving the result in the destination operand. SUB sets the CY (borrow) flag if a borrow is needed for bit 7. Otherwise, CY is clear.

When subtracting signed integers, the OV flag 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.

Bit 7 in this description refers to the most significant byte of the operand (8, 16, or 32 bit)

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

FLAGS :

CY	AC	OV	N	Z
✓	✓ ★	✓	✓	✓

★ For word and dword subtractions, AC is not affected.

Example :

Register 1 contains 0C9h (11001001B) and register 0 contains 54h (01010100B). After executing the instruction SUB R1,R0 register 1 contains 75h (01110101B), the CY and AC flags are clear, and the OV flag is set.

SUB Rmd,Rms

[Encoding]

9	**C**	ssss	SSSS

Hex Code in:

Binary Mode =[A5][Encoding]
Source Mode = [Encoding]

Operation:

SUB
(Rmd) ← (Rmd) - (Rms)

SUB WRjd,WRjs

[Encoding]

9	**D**	tttt	TTTT

Hex Code in:

Binary Mode =[A5][Encoding]
Source Mode = [Encoding]

Operation:

SUB
(WRjd) ← (WRjd) - (WRjs)

SUB DRkd,DRks

[Encoding]

9	**F**	uuuu	UUUU

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

SUB
(DRkd) ← (DRkd) - (DRks)

SUB Rm,#data

[Encoding]

9	**E**	ssss	**0**	immed data

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

SUB
(Rm) ← (Rm) - #data

SUB WRj,#data16

[Encoding]

9	**E**	tttt	**4**	immed data hi	immed data low

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

SUB
(WRj) ← (WRj) - #data16

SUB DRk,#data16

[Encoding]

9	**E**	uuuu	**8**	immed data hi	immed data low

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

SUB
(DRk) ← (DRk) - #data16

SUB Rm,dir8

[Encoding]

9	**E**	ssss	**1**	addr7-addr0

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

SUB
(Rm) ← (Rm) - (dir8)

SUB WRj,dir8

[Encoding]

9	**E**	tttt	**5**	addr7-addr0

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

SUB
(WRj) ← (WRj) - (dir8)

SUB Rm,dir16

[Encoding]

9	**E**	ssss	**3**	addr15-addr8	addr7-addr0

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

SUB
(Rm) ← (Rm) - (dir16)

SUB WRj,dir16

[Encoding]

9	**E**	tttt	**7**	addr15-addr8	addr7-addr0

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

SUB
(WRj) ← (WRj) - (dir16)

SUB Rm,@WRj

[Encoding]

9	**E**	tttt	**9**	ssss	**0**

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

SUB
(Rm) ← (Rm) - ((WRj))

SUB Rm,@DRk

[Encoding]

9	**E**	uuuu	**B**	ssss	**0**

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

SUB
(Rm) ← (Rm) - ((DRk))

SUBB A,<src-byte>

Function:

Subtract with borrow

Description:

SUBB subtracts the specified variable and the CY flag together from the accumulator, leaving the result in the accumulator. SUBB sets the CY (borrow) flag if a borrow is needed for bit 7 and clears CY otherwise. (If CY 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 CY flag 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 the OV flag 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.

Bit 6 and bit 7 in this description refer to the most significant byte of the operand (8, 16 or 32 bit)

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

FLAGS :

CY	AC	OV	N	Z
✓	✓	✓	✓	✓

Example :

The accumulator contains 0C9h (11001001B), register 2 contains 54h (01010100B), and the CY flag is set. After executing the instruction SUBB A,R2 the accumulator contains 74h (01110100B), the CY and AC flags are clear, and the OV flag is set.Notice that 0C9h minus 54h is 75h. The difference between this and the above result is due to the CY (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 CY instruction.

SUBB A,#data

[Encoding]

9	4	immed data

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [Encoding]

Operation:

SUBB
(A) ← (A) - (CY) - #data

SUBB A,dir8

[Encoding]

9	**5**	addr7-addr0

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [Encoding]

Operation:

SUBB
(A) ← (A) - (CY) - (dir8)

SUBB A,@Ri

[Encoding]

9	011i

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [A5][Encoding]

Operation:

SUBB
(A) ← (A) - (CY) - ((Ri))

SUBB A,Rn

[Encoding]

9	1rrr

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [A5][Encoding]

Operation:

SUBB
(A) ← (A) - (CY) - (Rn)

SWAP A

Function:

Swap nibbles within the accumulator

Description:

Interchanges the low and high nibbles (4–bit fields) of the accumulator (bits 3-0 and bits 7- 4). This operation can also be thought of as a 4–bit rotate instruction.

FLAGS :

CY	AC	OV	N	Z
–	–	–	–	–

Example :

The accumulator contains 0C5h (11000101B). After executing the instruction SWAP A the accumulator contains 5Ch (01011100B).

SWAP A

[Encoding]

C	4

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [Encoding]

Operation:

SWAP
(A).3:0 → ← (A).7:4

TRAP

Function:

Causes interrupt call

Description:

Causes an interrupt call that is vectored through location FF:007Bh. The operation of this instruction is not affected by the state of the interrupt enable flag in PSW0 and PSW1. Interrupt calls can not occur immediately following this instruction.

FLAGS :

CY	AC	OV	N	Z
–	–	–	–	–

Example :

The instruction TRAP causes an interrupt call to location 0FF007Bh during normal operation.

TRAP

[Encoding]

B	9

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

Operation for INTR = 0 :
TRAP
(SP) ← (SP) + 1
(SP) ← (PC).15:0
(SP) ← (SP) + 1
(PC) ← (FF007Bh)
Operation for INTR = 1 :
TRAP
(SP) ← (SP) + 1
(PSW1) ← ((SP))
(SP) ← (SP) + 1
(PC).23:16 ← ((SP))
(SP) ← (SP) + 1
(PC).15:8 ← ((SP))
(SP) ← (SP) + 1
(PC) ← (FF007Bh)

XCh A,<byte>

Function:

Exchange accumulator with byte variable

Description:

Loads the accumulator with the contents of the specified variable, at the same time writing the original accumulator contents to the specified variable. The source/destination operand can use register, direct or register–indirect addressing.

FLAGS :

CY	AC	OV	N	Z
–	–	–	–	–

Example :

R0 contains the address 20h, the accumulator contains 3Fh (00111111B) and on–chip RAM location 20h contains 75h (01110101B). After executing the instruction XCh A,@R0. RAM location 20h contains 3Fh (00111111B) and the accumulator contains 75h (01110101B).

XCh A,dir8

[Encoding]

C	**5**	addr7-addr0

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [Encoding]

Operation:

XCh
(A) → ← (dir8)

XCh A,@Ri

[Encoding]

C	011i

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [A5][Encoding]

Operation:

XCh
(A) → ← ((Ri))

XCh A,Rn

[Encoding]

C	1rrr

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [A5][Encoding]

Operation:

XCh
(A) → ← (Rn)

XCHD A,@Ri

Function:

Exchange digit

Description:

Exchanges the low nibble of the accumulator (bits 3–0) generally representing a Hexadecimal or BCD digit, with that of the on–chip RAM location indirectly addressed by the specified register. Does not affect the high nibble (bits 7–4) of either register.

FLAGS :

CY	AC	OV	N	Z
–	–	–	–	–

Example :

R0 contains the address 20h, the accumulator contains 36h (00110110B), and on–chip RAM location 20h contains 75h (01110101B). After executing the instruction, XCHD A,@R0 on–chip RAM location 20h contains 76h (01110110B) and 35h (00110101B) in the accumulator.

XCHD A,@Ri

[Encoding]

D	011i

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [A5][Encoding]

Operation:

XCHD
(A).3:0 → ← ((Ri)).3:0

XRL <dest>,<src>

Function:

Logical Exclusive–OR for byte variables

Description:

Performs the bitwise logical Exclusive–OR operation (∀) between the specified variables, storing the results in the destination. The destination operand can be the accumulator, a register or a direct address.

The two operands allow 12 addressing mode combinations. When the destination is the accumulator or a register, the source addressing can be register, direct, register–indirect or immediate; 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 is read from the output data latch, not the input pins.

FLAGS :

CY	AC	OV	N	Z
–	–	–	✓	✓

Example :

The contains 0C3h (11000011B) and R0 contains 0AAh (10101010B). After executing the instruction, XRL A,R0 the accumulator contains 69h (01101001B).

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 complements bits 5, 4, and 0 of output Port 1.

XRL dir8,A

[Encoding]

6	**2**	addr7-addr0

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [Encoding]

Operation:

XRL
(dir8) ← (dir8) ∀ (A)

XRL dir8,#data

[Encoding]

6	**3**	addr7-addr0	immed data

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [Encoding]

Operation:

XRL
(dir8) ← (dir8) ∀ #data

XRL A,#data

[Encoding]

6	**4**	immed data

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [Encoding]

Operation:

XRL
(A) ← (A) ∀ #data

XRL A,dir8

[Encoding]

6	**5**	addr7-addr0

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [Encoding]

Operation:

XRL
(A) ← (A) ∀ (dir8)

XRL A,@Ri

[Encoding]

6	011i

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [A5][Encoding]

Operation:

XRL
(A) ← (A) ∀ ((Ri))

XRL A,Rn

[Encoding]

6	1rrr

Hex Code in:

Binary Mode = [Encoding]
Source Mode = [A5][Encoding]

Operation:

XRL
(A) ← (A) ∀ (Rn)

XRL Rmd,Rms

[Encoding]

6	**C**	ssss	SSSS

Hex Code in:	**Operation:**
Binary Mode = [A5][Encoding] Source Mode = [Encoding]	XRL (Rmd) ← (Rmd) ∀ (Rms)

XRL WRjd,WRjs

[Encoding]

6	**D**	tttt	TTTT

Hex Code in:	**Operation:**
Binary Mode = [A5][Encoding] Source Mode = [Encoding]	XRL (WRds) ← (WRjd) ∀ (WRjs)

XRL Rm,#data

[Encoding]

6	**E**	ssss	**0**	immed data

Hex Code in:	**Operation:**
Binary Mode = [A5][Encoding] Source Mode = [Encoding]	XRL (Rm) ← (Rm) ∀ #data

XRL WRj,#data16

[Encoding]

6	**E**	tttt	**4**	immed data hi	immed data low

Hex Code in:	**Operation:**
Binary Mode = [A5][Encoding] Source Mode = [Encoding]	XRL (WRj) ← (WRj) ∀ #data16

XRL Rm,dir8

[Encoding]

6	**E**	ssss	**1**	addr7-addr0

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

XRL
(Rm) ← (Rm) ∀ (dir8)

XRL WRj,dir8

[Encoding]

6	**E**	tttt	**5**	addr7-addr0

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

XRL
(WRj) ← (WRj) ∀ (dir8)

XRL Rm,dir16

[Encoding]

6	**E**	ssss	**3**	addr15-addr8	addr7-addr0

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

XRL
(Rm) ← (Rm) ∀ (dir16)

XRL WRj,dir16

[Encoding]

6	**E**	tttt	**7**	addr15-addr8	addr7-addr0

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

XRL
(WRj) ← (WRj) ∀ (dir16)

XRL Rm,@Wrj

[Encoding]

6	**E**	tttt	**9**	ssss	**0**

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

XRL
(Rm) ← (Rm) ∀ ((WRj))

XRL Rm,@Drk

[Encoding]

6	**E**	uuuu	**B**	ssss	**0**

Hex Code in:

Binary Mode = [A5][Encoding]
Source Mode = [Encoding]

Operation:

XRL
(Rm) ← (Rm) ∀ ((DRk))

Glossary

This glossary defines acronyms, abbreviations, and terms that have special meaning in this manual.

#0data16 A 32–bit constant that is immediately addressed in an instruction. The upper 16–bit part is filled with zeros.

#1data16 A 32–bit constant that is immediately addressed in an instruction. The upper 16–bit part is filled with ones.

#data An 8–bit constant that is immediately addressed in an instruction.

#data16 A 16–bit constant that is immediately addressed in an instruction.

#short A constant, equal to 1, 2 or 4, that is immediately addressed in an instruction.

accumulator A register or storage location that forms the result of an arithmetic or logical operation.

addr11 An 11–bit destination address. The destination can be anywhere within the same 2–Kbyte block of memory as the first byte of the next instruction.

addr16 A 16–bit destination address. The destination can be anywhere within the same 64–Kbyte region as the first byte of the next instruction.

addr24 A 24–bit destination address. The destination can be anywhere within the 16–Mbyte address space.

ALU Arithmetic–logic unit. The part of the CPU that processes arithmetic and logical operations.

assert The term assert refers to the act of making a signal active (enabled). The polarity (high/low) is defined by the signal name. Active–low signals are designated by a pound symbol (#) suffix; active–high signals have no suffix. To assert RD# is to drive it low; to assert ALE is to drive it high.

binary–code compatibility The ability of a TSC80251 microcontroller to execute, without modification, binary code written for an 80C51 microcontroller.

binary mode An operating mode, selected by a configuration bit, that enables a TSC80251 microcontroller to execute, without modification, binary code written for a 80C51 microcontroller.

bit A binary digit.

bit (operand) An addressable bit in the C251 Architecture.

bit51 An addressable bit in the C251 Architecture.

byte Any 8–bit unit of data.

clear The term clear refers to the value of a bit or the act of giving it a value. If a bit is clear, its value is “0”; clearing a bit gives it a “0” value.

code memory See program memory.

configuration bytes Bytes that determine a set of operating parameters for the TSC80251 Product. For TSC80251 EPROM and OTPROM versions, these bytes are programmable in an EPROM area. For TSC83251 masked ROM versions, these bytes are additional information provided in a masked ROM area. For TSC80251 ROMless version, these bytes are configured in factory according to the part number.

dir8 An 8–bit direct address. This can be a memory address or an SFR address.

dir16 A 16–bit memory address (00:0000h-00:FFFFh) used in direct addressing.

DPTR The 16–bit data pointer. In TSC80251 microcontrollers, DPTR is the lower 16 bits of the 24–bit extended data pointer, DPX.

DPX The 24–bit extended data pointer in TSC80251 microcontrollers. See also DPTR.

deassert The term deassert refers to the act of making a signal inactive (disabled). The polarity (high/low) is defined by the signal name. Active–low signals are designated by a pound symbol (#) suffix; active–high signals have no suffix. To deassert RD# is to drive it high; to deassert ALE is to drive it low.

double word A 32–bit unit of data. In memory, a double word comprises four contiguous bytes.

dword See double word.

EPROM Erasable programmable read–only memory.

external address A 16–bit or 17–bit address presented on the device pins. The address decoded by an external device depends on how many of these address bits the external system uses. See also internal address.

integer Any member of the set consisting of the positive and negative whole numbers and zero.

internal address The 24–bit address that the device generates. See also external address.

interrupt handler The module responsible for handling interrupts that are to be serviced by user–written interrupt service routines.

interrupt latency The delay between an interrupt request and the time when the first instruction in the interrupt service routine begins execution.

interrupt response time The time delay between an interrupt request and the resulting break in the current instruction stream.

interrupt service routine The software routine that services an interrupt.

LSB Least–significant bit of a byte or a least–significant byte of a word.

MSB Most–significant bit of a byte or a most–significant byte of a word.

multiplexed bus A bus on which the data is time–multiplexed with (some of) the address bits.

OTPROM One–time–programmable read–only memory, a version of EPROM.

PC Program counter.

program memory A part of memory where instructions can be stored for fetching and execution.

RAM Random access memory

rel A signed (two’s complement) 8–bit, relative destination address. The destination is –128 to +127 bytes relative to the first byte of the next instruction.

reserved bits Register bits that are not used in this device but may be used in future implementations. Avoid any software dependence on these bits. In most cases: the value read from this bit is indeterminate; do not set this bit.

ROM Read only memory

set The term set refers to the value of a bit or the act of giving it a value. If a bit is set, its value is “1”; setting a bit gives it a “1” value.

SFR Special Function Register.

sign extension A method for converting data to a larger format by filling the extra bit positions with the value of the sign. This conversion preserves the positive or negative value of signed integers.

source–code compatibility The ability of an TSC80251 microcontroller to execute recompiled source code written for an 80C51 microcontroller.

source mode An operating mode that is selected by a configuration bit. In source mode, a TSC80251 microcontroller can execute recompiled source code written for a 80C51 microcontroller. In source mode, the TSC80251 microcontroller cannot execute unmodified binary code written for an 80C51 microcontroller. See binary mode.

SP Stack pointer.

SPX Extended stack pointer.

state time (or state) The basic time unit of the microcontroller; the combined period of the two internal timing signals, PH1 and PH2. (The internal clock generator produces PH1 and PH2 by halving the frequency of the signal on XTAL1.) With a 16–MHz crystal, one state time equals 125 ns. Because the device can operate at many frequencies, this manual defines time requirements in terms of state times rather than in specific units of time.

word A 16–bit unit of data. In memory, a word comprises two contiguous bytes.