The 8051 Assembler A51

Introduction

A51 is a cross macro-assembler intended to perform the clerical task of converting a user program written as a series of instructions into executable code that can run in the 8051 family of microcontrollers. A51 is part of the CALL51 software develop package for the 8051. CALL51 includes: a C compiler (C51), an assembler (A51), a linker/locator (L51), and a librarian (Lib51). This manual describes the assembler A51.

Disclaimer

Invocation

A51 is a command prompt program. It is called by typing a51 followed by the assembly source file to compile:

a51 [options] source.asm

A51 recognizes several optional options as listed below:

Option	Description
-l	Create list file
-s	Send output to stdout
-i	Case sensitive symbols
-c	Compile only. Generates object file.

Constants and Operators

A51 can recognize different constant radixes as well as perform unary and binary operations with labels, operands, and constants. Constants must start with a digit (0 to 9) regardless of radix. These are the radixes recognized by A51:

Radix	Prefix Operator	Suffix Operator	Examples
Binary	0B or 0b	B or b	`10011000B, 0b10011000`
Octal	0q, 0Q, 0o, 0O	Q or q	`3452Q, 0o3452,`
Decimal	0d, 0D, or empty	D, d, or empty	`2011D, 0d2011 or 2011`
Hexadecimal	0x or 0X	H or h	`0x3AFF or 3AFFH`

Assembly operators are evaluated either at compile time for absolute symbols and constants or at linking time for relocatable symbols. In either case, **these operators are not evaluated at run time** in the microcontroller. It is the responsibility of the programmer to perform operations at run time by means of assembly instructions. The following operators are available in A51:

Operator(s)	Meaning	Example
OR, |	Logic OR	MOV a, #(0x01 | 0x02)
AND, &	Logic AND	MOV R1, #(myvar and 0x0f)
NOT, ~	Complement	MOV a, #(~1)
XOR, ^	Exclusive OR	MOV a, #(myvar ^ 0F0H)
HIGH	Upper 8-bits	MOV R0, #HIGH(my_address)
LOW	Lower 8-bits	MOV R1, #LOW(my_address)
*	Multiplication	MOV R1, # (my_address*2)
/	Division	MOV R0, # (my_address/0x100)
%	Modulo	MOV R0, # (my_address%0x100)
- (unary form)	Two's complement	MOV DPTR, #(-1)
- (binary form)	subtraction	MOV DPTR, #(8000H-1)
+	Addition	LCALL mysub+3
SHR, >>	Shift right	MOV R0, # (my_address>>8)
SHL, <<	Shift left	MOV R1, # (my_address<<1)

Operator Precedence

The precedence of the operators listed above is as follows: Parenthesized expression ( ), unary -, HIGH, LOW, NOT, OR, AND, XOR, *, /, %, -, +, SHR, and SHL.

Symbols Names

A symbol must begin with a letter (A-Z, a-z) or the special characters underscore ('_') or question mark ('?'), and it can be followed by letters, decimal digits (0-9), and the special characters underscore ('_') and question mark ('?'). There is no restriction in the number of valid characters in a label name. The instruction mnemonics, assembly-time operators, some predefined bit and data addresses, segment attributes, and assembler directives may not be used as user defined symbol names. For a complete list of these reserved words, refer to the table below.

a	***ab***	***acall***	***add***	***addc***	***ajmp***
and	***anl***	***ar0***	***ar1***	***ar2***	***ar3***
ar4	***ar5***	***ar6***	***ar7***	***at***	***bit***
bseg	***byte***	***c***	***cjne***	***clr***	***code***
cpl	***cseg***	***da***	***data***	***db***	***dbit***
dec	***div***	***djnz***	***dptr***	***ds***	***dseg***
dw	***end***	***equ***	***extern***	***extrn***	***high***
idata	***inc***	***iseg***	***jb***	***jbc***	***jc***
jmp	***jnb***	***jnc***	***jnz***	***jz***	***lcall***
ljmp	***low***	***mac***	***mov***	***movc***	***movx***
mul	***nop***	***not***	***or***	***org***	***orl***
overlay	***pc***	***pop***	***public***	***push***	***r0***
r1	***r2***	***r3***	***r4***	***r5***	***r6***
r7	***ret***	***reti***	***rl***	***rlc***	***rr***
rrc	***rseg***	***seg***	***segment***	***set***	***setb***
shl	***shr***	***sjmp***	***skip***	***subb***	***swap***
using	***word***	***xch***	***xchd***	***xdata***	***xor***
xrl	***xseg***	***endmac***			

Reserved Assembler Symbols

Assembly Language Program Structure

An assembly language program consists of a sequence of statements that, after compiled into machine code, instruct the microcontroller to perform the desired operations. Most lines of an assembly program are comprised of up to four distinct fields. Some of the fields may be empty. The order of these fields is

```
[label:]  Operation        Operand          [;Comment]
```

By default A51 is not case sensitive. You can use uppercase or lowercase letters or even mix the use of cases to represent any of the four fields. The assembler can be made case sensitive by using the appropriate command line switch as described in Invocation section above. This switch is necessary when assembling files generated by C51, as the C program language is case sensitive.

The Label Field

Labels are symbols defined by the user to identify memory locations in the program or data areas of the assembly source code. For most instructions and directives, the label is optional. Labels must follow these rules:

- A label must be the first field in a source statement and must be terminated by a colon (:). Some assembler directives use symbols that do not represent a memory location and do not require a colon.
- Only one label can be defined per line. When labels are defined in consecutive lines without assembler operators, they represent the same memory location.

These are examples of instructions with valid labels:

```
sum:          ADD    A, B
out:          LCALL  _printf
loop1:        ; Labels can be the only field in a line
```

```
loop2:       MOV    R0, #10H
This_is_a_very_long_label_do_you_like_it?:
             INC A
```

These are examples of instructions with invalid labels:

```
add 10:      MOV    A,#10        ; a blank is included in the label
l1           ADD    A,RO         ; the label is not terminated with a colon
7UP:         DA     A            ; the label starts with a number
asf[]:       PUSH   AR5          ; label uses invalid characters [ and ]
```

The Operation Field

The operation field specifies the action to be performed. It may consist of an instruction mnemonic (opcode) or an assembler directive. Here are some examples of operation fields:

```
       INC    R1     ; INC is the operation field
       SUB    A,#10  ; SUB is the operation field
zero   EQU    0      ; directive EQU is the operation field
```

The Operand Field

If an operand field is present, it follows the operation field and is separated from the operation field by at least one space or tab character. The operand field specifies operands for instructions or arguments for assembler directives. The following examples show some operand fields:

```
       MOV A, R0           ; A and R0 are operands
forever:
       SJMP forever        ; forever is the operand
```

The Comment Field

The comment field starts with a semicolon and extends up to the end of the line. This field is optional and may contain any printable ASCII character. Comments do not affect assembly processing or program execution. Comments are used mainly for documentation purposes. These are some examples of comments:

```
       DEC    R0           ; decrement the contents at location 30H
                           ; The whole line is a comment
       ; MOV a, #10H ← this instruction was commented out!
```

Assembler Directives

Assembler directives are similar to instructions in an assembly language program, but instead of generating machine code they tell the assembler how to behave and how to process subsequent assembly language instructions. Directives are also used to define program constants and reserve space for variables. The A51 assembler directives are listed in the table below.

Category	**Directives**
Segment management	SEGMENT, RSEG, CSEG, DSEG, BSEG, ISEG, XSEG
Symbol definition	EQU, SET, BIT, CODE, DATA. IDATA, XDATA
Memory initialization	DB, DW
Memory reservation	DBIT, DS
Program linkage	PUBLIC, EXTERN
Address management	ORG, USING
Others	END

Segment Control Directives

A segment is a block memory (either code or data) created by the assembler as instructed in an 8051 assembly source file. Only one segment can be active at a time. A51 works with two types of segments: ***relocatable*** and ***absolute.***

Location Counter

A51 keeps a location counter for each segment. The location counter is a pointer to the address space of the active segment and represents an offset for relocatable segments or the current absolute address for absolute segments. When a segment is first selected, its associated location counter is reset to 0. The location counter is changed after each instruction according to the length of the instruction. The memory initialization and allocation directives (i.e., DS, DB, DW, or DBIT) change the value of the location counter as these directives reserve or preset memory. Using the ORG directive it is possible to set the value of the location counter for absolute segments. Every time a segment is activated, the location counter of the segment is restored to its previous value. Whenever the assembler encounters a label definition, it assigns the value of the location counter as well as the type of the current segment to that label.

The value of the location counter of the active segment can be accessed programmatically by using the dollar sign ($). When $ is as an operand for a multi-byte instruction or directive, the value used is the address of the first byte of that instruction. WARNING: do not use $ in relocatable segments, as the linker doesn't keep track of the number of bytes used per instruction.

Relocatable Segments

Relocatable segments are created using the SEGMENT directive. When the programmer creates a relocatable segment, it must specify the name of the segment, the segment class, and an optional relocation type. Relocatable segments are assigned absolute addresses by the linker L51. For example,

```
mycode SEGMENT CODE
```

defines a segment named **mycode** with a memory class of **CODE.** This means that data in the **myprog** segment will be located in the code or program area of the MCS-51. In order to activate a previously defined relocatable segment, the **RSEG** directive is used. After RSEG is used to select a segment, that segment becomes the active segment and used for subsequent code and data until the segment is changed either with a new RSEG directive or with an absolute segment directive. For example,

```
RSEG mycode
```

selects the ***mycode*** segment, as defined previously, as the current active segment.

Absolute Segments

Absolute segments are allocated by the assembler at fixed memory locations as specified by the programmer. They are created using the **BSEG, CSEG, DSEG, ISEG,** and **XSEG** directives, which enable either to allocate code and data or reserve memory space at fixed locations. By default in A51, and until changed by the programmer, the absolute CODE segment is the active segment with and initial location counter of 0000H.

SEGMENT

The SEGMENT directive is used to declare a relocatable segment. An optional relocation type may be specified in the segment declaration. The format of SEGMENT directive is as follows:

Segment SEGMENT ***class reloctype***

Where

> ***Segment:*** The name to be assigned to the segment.
>
> ***Class:*** The memory space for the segment: BIT, CODE, DATA, IDATA, or XDATA.
>
> ***Reloctype*** Defines the relocation type that may be performed by the Linker/Locator.
> The valid relocation types are listed in Table 2.2.

For example,

```
DDS SEGMENT DATA
```

defines a segment with the name DDS and the memory class DATA.

```
XDS SEGMENT XDATA
```

defines a segment with the name **XDS** and the memory class XDATA.

BSEG, CSEG, DSEG, ISEG, and XSEG

Each of the BSEG, CSEG, DSEG, ISEG, or XSEG directives selects an absolute segment in the corresponding memory class. These directives use the following formats:

```
BSEG  AT address_8  ; selects the absolute BIT segment
CSEG  AT address_16 ; selects the absolute CODE segment
DSEG  AT address_8  ; selects the absolute DATA segment
ISEG  AT address_8  ; selects the absolute IDATA segment
XSEG  AT address_16 ; selects the absolute XDATA segment
```

where ***address_8*** and ***address_16*** are optional base address at which the segment starts.

```
BSEG AT 40H
```

selects the absolute BIT segment and sets its location counter to 40H.

```
CSEG AT 0x2000
```

selects the absolute CODE segment and sets its location counter to 2000H.

Symbol Definition Directives

The symbol definition directives permit the creation of symbols that can be used to represent segments, registers, numbers, and addresses. Symbols defined using these directives may not have been previously defined and may not be redefined by any means. The SET directive is the only exception to this rule.

EQU, SET

The EQU and SET directives assign a numeric value or register symbol to the specified symbol name. Symbols defined with EQU may not have been previously defined and may not be redefined by any means. The SET directive allows later redefinition of symbols. The formats of these two directives are as follows:

```
Symbol      EQU   expression
Symbol      EQU   register
Symbol      SET   expression
Symbol      SET   register
```

where

> ***Symbol:*** is the name of the symbol to be defined. The expression or register specified in the EQU or SET directive will be substituted for each occurrence of ***symbol*** that is used in your assembly program.
>
> ***expression*** :is a numeric expression that may include symbols.
>
> ***register:*** is one of the following register names: A, R0, R1, R2, R3, R4, R5, R6, R7, AR0, AR1, AR2, AR3, AR4, AR5, AR6, or AR7.

The following are examples of using EQU and SET directives:

```
ptr          SET R0       ; use R0 as ptr
true         EQU 1        ; use true to represent 1
counter      EQU R3       ; use R3 as counter
```

BIT, CODE, DATA, IDATA, and XDATA

The BIT, CODE, DATA, IDATA, and XDATA directives assign an address value to the specified symbol. Symbols defined with the BIT, CODE, DATA, IDATA, and XDATA directives may not be changed or redefined. The formats of these directives are

```
Symbol       BIT    bit_address        ; defines a BIT symbol
Symbol       CODE   code_address       ; defines a CODE symbol
Symbol       DATA   data_address       ; defines a DATA symbol
Symbol       IDATA  idata_address      ; defines an IDATA symbol
Symbol       XDATA  xdata_address      ; defines an XDATA symbol
```

where

bit_address is the address of a bit in internal data memory. Bit address cannot be greater than 255.
code_address is a code address in the range from 0000H to 0FFFFH.

data_address is a DATA memory address in the range from 0 to 127 or a special function register (SFR) address in the range from 128 to 255.

idata_address is an IDATA memory address in the range from 0 to 255.

xdata_address is a XDATA memory address in the range from 0000H to 0FFFFH

Examples of using these directives follow:

```
enable_flg   BIT 10H      ; use bit location at 10H as enable_flg
P1           DATA 90H     ; P1 is a special function register
buffer       IDATA 30H    ; use a location at IDATA as buffer
buffer       XDATA 8000H  ; use external data memory location 8000H
                          ; as variable buffer
reset        CODE 0000H   ; make ‘reset’ equal to 0000H
```

Memory Initialization Directives

The memory initialization directives are used to initialize code space in either byte or word units. The memory image starts at the point indicated by the current value of the location counter in the currently active segment.

DB

The DB directive initializes program memory with 8-bit byte values. It can be used only when a segment with a CODE storage class is the active segment. The DB directive has the following format:

```
label:      DB    expression, expression, . . .
```

where

label: is the symbol that is given the address of the initialized memory location.

expression: is a byte value. Each ***expression*** may be a symbol, a character string, or a numeric expression.

Examples of using the DB directive follow. If an optional label is used, its value will point to the first byte constant listed.

```
Hello:      DB    'Hello, world!' ; ASCII literal
Seven_seg:  DB    7EH,60H,6DH,79H,33H,5BH,5FH,70H,7FH,7BH
mixed:      DB    2*3,'date/time',2*8 ; can mix literals and numbers
```

DW

The DW directive initializes the program (code) memory with 16-bit constants. It can be used only when a segment with a CODE storage class is the active segment. The DW directive has the following format:

```
label:      DW    expression, expression, . . .
```

The following examples illustrate the use of this directive:

```
Jump_tab:   DW    Sun,Mon,Tue,Wed,Thu,Fri,Sat
misc:       DW    'A',1122H    ; first byte contains 0
                               ; second byte contains 41H
                               ; third byte contains 11H
                               ; fourth byte contains 22H
```

The symbols "Sun", "Mon", . . ., "Sat" are labels of some instructions in your program.

Memory Reservation Directives

The memory reservation directives are used to reserve space in bit, byte, or word units. The space reserved starts at the point indicated by the current value of the location counter in the currently active segment.

DBIT

The DBIT directive is used to reserve bits within a segment with the BIT storage class. The location counter of the segment is advanced by the value of the directive. The format of the DBIT directive is

```
label:      DBIT  expression
```

The following are examples of using the DBIT directive:

```
            DBIT  10    ; reserve 10 bits
```

```
status:      DBIT   5      ; reserve 5 bits to keep track of program status
```

DS

The DS directive is used to reserve space in the currently selected segment in byte units. It can be used only when currently active segment has storage classes IDATA, DATA, or XDATA. The location counter of the segment is advanced by the value of the directive.

The format of the DS directive is as follows:

label: DS ***expression***

The following are examples of using the DS directive in the external data segment.

```
             XDATA        ; select the external data segment
             DS     20    ; reserve 20 bytes (label is optional)
io_buf:      DS     40    ; reserve 40 bytes for I/O
```

Occasionally, it is desirable to tell the assembler where to reserve a block of memory. This can be achieved by combining the use of AT and DS directives. For example, the following directives reserve 100 bytes starting from address 2000H:

```
       XDATA AT 2000H      ; reserve a memory block starting at address 2000H
       DS     100          ; reserve 100 bytes
```

Here are other examples of using the storage reservation directives:

```
             BSEG AT 20H         ; absolute bit segment at 20H
dec_flag:    DBIT   1            ; absolute bit
inc_flag:    DBIT   1

             CSEG AT 100H        ; absolute code segment at 100H
; Seven-segment display patterns for 0-9
seg7-tab:    DB     7EH,30H,6DH,79H,33H
             DB     5BH,5FH,70H,7FH,7BH

             DSEG AT 40H         ; absolute data segment at 40H
i:           DS     1            ; reserve one byte for variable i
j:           DS     1            ; reserve one byte for variable j
sum:         DS     2            ; reserve two bytes for sum
float:       DS     4            ; reserve four bytes for float

             ISEG AT 0D0H        ; Absolute indirect data segment at D0H
ptr_1:       DS     2
ptr_2:       DS     2
out_buf:     DS     0x10         ; Reserve 16 bytes of IDATA starting at 0D4H

             XSEG AT 2000H       ; Absolute external data segment at 2000H
stringl:     DS     100          ; Reserve 100 bytes for string1
string2:     DS     10*10        ; Also reserve 100 bytes for string2
in_buf:      DS     80
```

Program Linkage Directives

A program may need to access variables or call subroutines that reside in different files in order to perform its intended operations. To make this type of cross-reference possible, the programmer needs to use assembler directives to tell the assembler that some variables or subroutines used/called in one module reside in a different module. The programmer also needs to inform the assembler about variables and subroutines that will be accessed by programs in different modules. At some point, all these modules must be combined into a single executable by 'linking' them. This is the function of the linker L51. These are the program linkage directives:

PUBLIC

The PUBLIC directive lists symbols that may be used in other object modules. A symbol declared PUBLIC must be defined in the current module. The format of this directive is as follows:

```
EXTERN symbol, symbol, . . .
```

The following example illustrates the use of this directive:

```
PUBLIC getchar, putchar, gets, puts
```

EXTRN

The EXTRN directive tells the assembler of a list of symbols to be referenced in the current module but that are defined in other modules. The list of external symbols must have a segment associated with each symbol in the list. The format of this directive is as follows:

```
EXTERN class (symbol, symbol, . . . )
```

where

> ***class:*** is the memory class where the symbol has been defined and may be one of the following: BIT, CODE, DATA, IDATA, or XDATA.
>
> ***symbol:*** is an external symbol name.

As indicated earlier, the PUBLIC and EXTRN directives work together. When one symbol is declared EXTRN in one module, it must be declared PUBLIC in another module. Symbols can be declared public only once in all the modules to be linked. The use of the PUBLIC and EXTRN directives is illustrated in the following example:

In module main.asm:

```
EXTRN CODE (putchar, getchar)
.
.
.
LCALL putchar
```

```
.
.
LCALL getchar
```

In module util.asm:

```
PUBLIC putchar, getchar
.
.
.
putchar: ;begin of subroutine
.
.
.
RET
getchar: ;begin of subroutine
.
.
.
RET
```

Address Control Directives

The ORG directive allow the control of the address location counter for the current absolute segment, while the USING directive is use to select the bank for absolute symbols R0 to R7.

ORG

The ORG directive is used to specify a value for the currently active segment's location counter. The ORG directive works only for absolute segments CSEG, DSEG, ISEG, BSEG, and XSEG.

The format for the ORG directive is as follows:

```
ORG expression
```

The following examples illustrate the use of this directive:

```
ORG    1000H              ; set active location counter to 4096
ORG    RESET              ; RESET is a previously defined symbol
ORG    jump_tab+200H      ; arithmetic expression
```

Using

The format for the USING directive is shown below. Note that a label is not permitted.

```
USING expression
```

This directive notifies the assembler of the register bank that is used by the subsequent code. The expression is the number (between 0 and 3 inclusive) which refers to one of four register banks. The USING directive allows the programmer to use the predefined symbolic register addresses (AR0 through AR7) instead of their absolute addresses.

NOTE: the ‘USING’ directive does not select the register bank when using register R0 to R7. To perform such task, the programmer must update the value of the Program Status Word Special Function Register (PSW SFR) using assembly instructions.

Examples:

```
USING  3
PUSH   AR2    ; Push register 2 of bank 3

USING  1
PUSH   AR2    ; Push register 2 of bank 1
```

Other Directives

END

The END directive signals the end of the assembly module. Any text in the assembly file that appears after the END directive is ignored.

Macro Definition Directives

A macro is a name assigned to one or more assembly statements or directives. Macros are used to include the same sequence of instructions in several places. This sequence of instructions may operate with different parameters, as indicated by the programmer.

MAC

The MAC directive is used to define the start of a macro. A macro is a segment of instructions that is enclosed between the directives MAC and ENDMAC. The format of a macro is as follows:

```
name MAC ; comment
.
.
.
ENDMAC
```

ENDMAC

This directive terminates a macro definition.

```
average MAC; a macro that computes the average of three byte arguments
       mov    A, %0
       add    A, %1
       add    A, %2
       mov B, #3
       div AB        ; divide B into A and A gets the average
ENDMAC               ; terminate the macro definition
```

To invoke the above defined macro, enter the macro name and its parameters. The statement

```
average (R1 ,R2, R3)
```

will enable the assembler to generate the following instructions, starting from the current location counter:

```
mov A, Rl
add A, R2
add A, R3
mov B, #3
div AB
```

In order to place local labels within macros, use '%M' in the name of the label symbol. '%M' will be replaced with a consecutive and unique integer for each macro usage. For example:

```
MYMACRO MAC
       ; MYMACRO begins here
       mov P0, %0
loop%M:
       djnz R0, loop%M
       mov P0, %1
       ; MYMACRO ends here
ENDMAC
```

Assembler Controls

Control statements are used to tell the assembler how it interacts with the host computer regarding data input and output. A51 recognizes the following controls:

Control	Description
$INCLUDE	Includes a file into the current assembly source file
$MOD	Include register definition for a particular 8051 variant
$NOMOD51	Do not define default 8051 special function register definition
$LIST	Enable output to list file
$NOLIST	Disable output to list file.

$INCLUDE

The $INCLUDE control is used to include a file into the user source code. The format of this control is as follows:

$INCLUDE(filename)

For example:

$include(c:\source\asm\bpm2.asm)

$MOD

The $MOD control is used to include a Special Function Register definition file for an specific 8051 processor. The format of this control is as follows:

$MODproccesor

For example:
$MODDE2

The line above includes the SFR declarations for the DE2-8052 processor. In order for this control to work, a file named 'MODDE2' must exist at least in one of these three locations:

1. Where the assembly source file resides.

2. Where the assembler compiler resides.

3. In directory 'Define' at the parent directory where the assembler compiler resides. For example, if a51.exe resides in C:\Call51\bin, then the file can be placed in folder C:\Call51\Define.

$NOMOD51

By default A51 will define the Special Function Registers for the standard 8051 microcontroller if no $MOD control is found. This behaviour can be prevented by using the $NOMOD51 control.

$LIST

Enables output to the listing file.

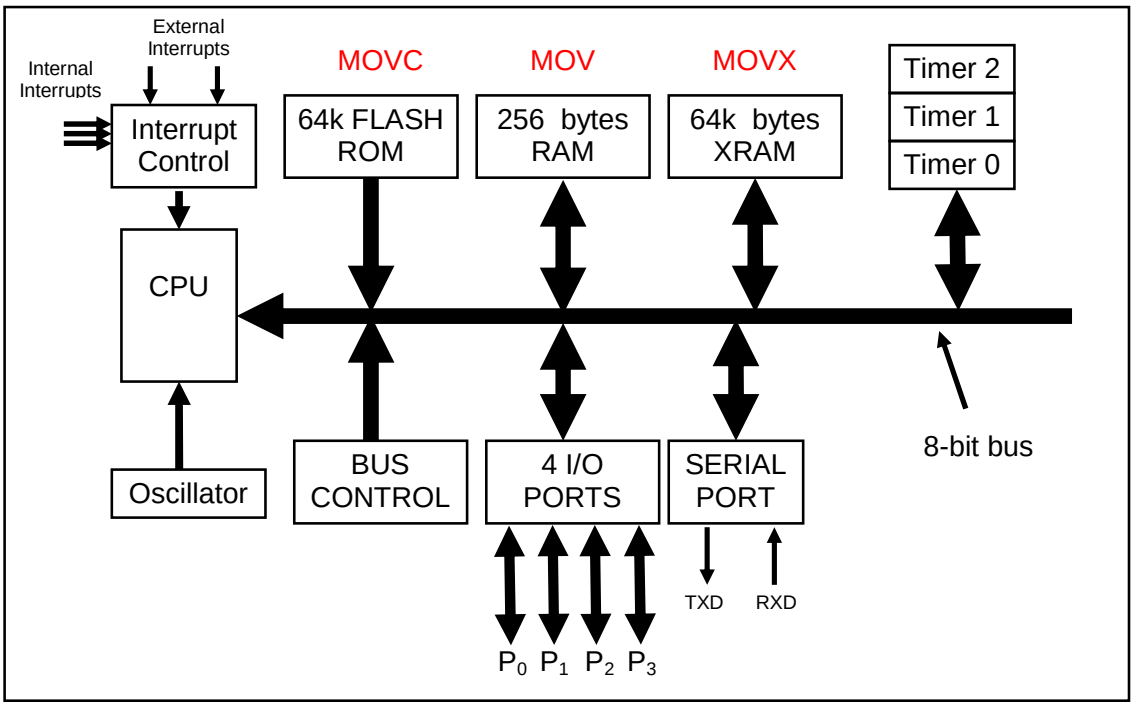
$NOLIST

Disables output to the listing file.

8051 Design Overview

The 8051 family of microcontrollers has been around since 1980. Originally designed and manufactured by Intel, it has become a de facto industry standard with more than 20 companies manufacturing significantly improved versions of this processor. Over more than three decades, many resources had become available for the 8051 family of microcontrollers, including books, source code, tutorials, debuggers, compilers, etc. Modern 8051 microcontrollers are among the cheapest processor available which in turn encourages manufactures to come out with newer and improved versions every year. The 8051 has been also very popular as a soft-processor with VHDL and Verilog implementations available, both free and commercial.

The figure below shows the block diagram of a modern 8051 microcontroller. It is important to notice that the 8051 microcontroller uses separated memory spaces for code and variables. A program running in the 8051 can access code memory using only the MOVC instruction. Variables in the 8051 can reside into two different memory spaces: internal and extended RAM. The internal RAM is a fast access memory space with many dedicated MOV instructions to utilize it. A program running in the 8051 can access expanded RAM using only the MOVX instruction.



8051 Registers

The 8051 microcontroller has many registers. All of them are 8-bit registers with the exception of the DPTR which is a 16-bit register. These registers are arranged as:

- 32 general purpose registers arranged in 4 banks of 8 register each: R0 to R7. R0 and R1 can be used as “index” registers for indirect access.

- Several (depends on the derivative, but usually 25 or more) Special Function Registers or SFRs. All the hardware resources of the 8051 are accessed through SFRs.

A few SFRs are related to ALU operations. That is, there are instructions in the 8051 microcontroller that use these registers implicitly. These registers are:

- Accumulator (A).
- Register B.
- Data Pointer (DPTR, a 16-bit register made using two eight bit registers put together: DPH and DPL)
- Stack Pointer (SP).
- Program Counter (PC).
- Program Status Word (PSW).

Accumulator

In the 8051 this is the most versatile of all the registers. Most of the ALU operations place the result in the accumulator. Additionally, many opcodes accept only the accumulator as operand. The accumulator is bit addressable.

Example usage:

```
MOV a, #20H ; Load accumulator with 20H
INC a ; Add one to accumulator
SWAP A ; Swap bits 0 to 3 with bits 4 to 7
```

Register B

This register is used together with the Accumulator to perform 8-bit multiplication and division operations. This SFR is bit addressable.

Example usage:

```
MOV A, #140
MOV B, #150
MUL AB ; After this inst. A=08H, B=52H
```

Data Pointer Register (DPTR)

Formed by two 8-bit SFRs (DPH and DPL) this is the only 16-bit register in the 8051. Together with some especial opcodes, it is used to access CODE and XRAM memory. There is a dedicated opcode to increment it! Unfortunately there is no opcode to decrement it!

Example usage:

```
MOV DPTR, #0AAAAH
INC DPTR ; DPL=0ABH, DPH=0AAH
MOVX A, @DPTR
```

Stack Pointer (SP)

The stack pointer SFR is used as an index register for instructions that use the stack. These instructions are CALL, LCALL, RET, IRET, PUSH, and POP. The stack pointer is initialized to 07H after reset. This SP SFR is incremented before a stack write operation and decremented before a stack read operation. The SP pointer should be initialized by the user program to a suitable memory location at the beginning of the main program in order to prevent unintentional memory accesses.

Example usage:

```
MOV SP, #7FH ; Set the stack after direct memory
```

Program Counter (PC)

This register points to the location of the code memory under access when running a program. This register not be accessed directly. This register is incremented by one, two, or three bytes (one for most instructions), depending on the opcode length. The jump, call, or return instructions modify the value of the program counter.

Example usage:

```
LJMP 34AAH ; Set PC to 34AAH
```

Program Status Word (PSW)

This bit addressable SFR is used to store the ALU flags as well as to select the active register bank. It contains the following bits:

CY	AC	F0	RS1	RS0	OV	--	P

CY	Carry flag
AC	Auxiliary Carry flag (For BCD Operations)
FO	Flag 0 (Available to the user for General Purpose)
RS1, RS0	Register bank select bits.
OV	Overflow flag
P	Parity flag

The carry, overflow, and parity flag are affected by the instructions as shown in the table below.

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

Example usage:

```
JC 8 ; If the carry is set jump 8 bytes ahead
```

8051 Addressing Modes

The 8051 supports the following ten different types of addressing:

1. Register Inherent
2. Direct
3. Immediate
4. Indirect
5. Indexed
6. Relative
7. Absolute
8. Long
9. Bit Inherent
10. Bit Direct

Register Inherent Addressing

For this mode the opcode is already associated with the register for speed and efficiency.

Examples:

```
MOV R1, #10 ;  opcode: 01111001B + 00001010B
INC A ; opcode: 00000100B
INC R3 ; opcode: 00001011B
```

Direct Addressing

Direct addressing is used to access the first 128 bytes of internal ram (addresses 0 to 127) or the SFRs (addresses 128 to 255).

Examples:

```
MOV 20H, A ; 11110101B + 00100000B
MOV 50H, 25H ; 10000101B + 00100101B + 01010000B
MOV 50H, #25H ; 01110101B + 01010000B + 00100101B
MOV 80H, A ; 80H > 127 therefore is a SFR access!
MOV P0, A ; SFR P0 is at address 80H
```

Immediate Addressing

Immediate addressing is used to initialize a register or memory with a constant. The constant to load MUST be preceded with '#'.

Examples:

```
     MOV R0, #10
L1:  MOV P1, #01H
     NOP ; Does nothing just wastes time!
     MOV P1, #00H
     DJNZ R0, L1 ; Decrement an jump if no zero
```

Indirect Addressing

Indirect addressing can be used to access ALL the internal RAM of the microcontroller. It is the only means of accessing the internal RAM from address 128 to 255! This kind of addressing is the only means of accessing XRAM memory using the MOVX instruction.

Examples:

```
MOV R0, #80H
MOV A, @R0 ; Copy content of RAM loc. 80H into the Accumulator
MOV A, 80H ; Copy SFR 80H (P0) into the accumulator
MOV DPTR, #200H
MOVX A, @DPTR ; Copy content of XRAM loc. 200H into the Acc.
MOV R6, A
INC DPTR
MOVX A, @DPTR ; Copy content of XRAM loc. 201H into the Acc.
MOV R7, A
```

Indexed Addressing

Indexed addressing uses a base register (DPTR) and a offset register (Accumulator)

Examples:

```
JMP @a+DPTR
MOVC a, @a+DPTR
MOVC a, @a+PC
```

Relative Addressing

Relative addressing is used to jump (conditionally or unconditionally) to other parts of program. The offset is an 8-bit SIGNED number. The 8051 can jump in the range -128 to +127 bytes:

Examples:

```
                1    $mod52
0000            2    L1:
0000 00         3            nop
0001 80FD       4            sjmp L1
0003 8001       5            sjmp L2
0005 00         6            nop
0006 00         7    L2:     nop
                8    end
```

To jump conditionally any bit variable available can be used. Some conditional relative jumps implicitly use some bits in the program status word (PSW):

```
JZ L1
JNZ L2
JC L3
JNC L4
JB P1.3, L5
```

Absolute Addressing

Absolute addressing works on jumps or calls using a combination of the 11 bits in the destination and 5 upper bits of the program counter. It allows to jump or call within the same 2K page were the program counter is. There are only two instructions that support this addressing mode: ACALL and AJMP

Examples:

```
ACALL myroutine
AJMP DONE
```

Long Addressing

Long addressing works on jumps or calls to any destination in the 16-bit address range of the microcontroller.

Examples:

```
LCALL my_other_routine
LJMP DONE
```

Bit Inherent Addressing

This kind of addressing works only with the carry flag.

Examples:

```
SETB C
CLR C
CPL C
```

Bit Direct Addressing

The 8051 provides direct bit addressing of certain parts of its internal memory or SFRs. The first 128 bits are mapped in internal ram from bytes 20H to 2FH. The second 128 bits are mapped to SFRs, only if the SFR address location is divisible by 8.

Examples:

```
CLR P0.0 ; Clear bit 0 of port 0 (SFR address 80H)
SETB P0.1 ; Set bit 1 of port 0 (SFR address 81H)
SETB 00H ;  This is bit 0 of byte 20 in internal RAM
```

8051 Instruction Set

Notes on the Addressing Modes

Rn	Working register R0-R7
direct	128 internal RAM locations, any I/O port, control or status register
@Ri	Indirect internal or external RAM location addressed by register R0 or R1
#data	8-bit immediate constant
#data 16	16-bit immediate constant included as bytes 2 and 3 of instruction
bit	128 software flags, any bit addressable I/O pin, control or status bit
A	Accumulator
addr16	Destination address for LCALL and LJMP may be anywhere within the 64-Kbyte program memory address space
addr11	Destination address for ACALL and AJMP will be within the same 2-Kbyte page of program memory as the first byte of the following instruction
rel	SJMP and all conditional jumps include an 8 bit offset byte. Range is +127/-128 bytes relative to the first byte of the following instruction

Copyright

ACALL addr11

Function: Absolute call

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

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

```
ACALL  SUBRTN
```

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

Operation: ACALL

(PC) ← (PC) + 2
(SP) ← (SP) + 1
((SP)) ← (PC_{7-0})
(SP) ← (SP) + 1
((SP)) ← (PC_{15-8})
(PC10-0) ← page address

Encoding:

a_{10}	a_9	a_8	**1**	**0**	**0**	**0**	**1**

a_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0

ADD A, <src-byte>

Function: Add without carry

Description: ADD adds the byte variable indicated to the accumulator, leaving the result in the accumulator. The carry and auxiliary carry flags are set, respectively, if there is a carry out of bit 7 or bit 3, and cleared otherwise. When adding unsigned integers, the carry flag indicates an overflow occurred. OV is set if there is a carry out of bit 6 but not out of bit 7, or a carry out of bit 7 but not out of bit 6; otherwise OV is cleared. When adding signed integers, OV indicates a negative number produced as the sum of two positive operands, or a positive sum from two negative operands. Four source operand addressing modes are allowed: register, direct, register indirect, or immediate.

Example: The accumulator holds 0C3H (11000011B) and register 0 holds 0AAH (10101010B). The instruction

```
ADD A,R0
```

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

Operation: **ADD A, Rn**

(A) ← (A) + (direct)

Encoding:

0	0	1	0	1	r	r	r

Operation: **ADD A, direct**

(A) ← (A) + (direct)

Encoding:

0	0	1	0	0	1	0	1	direct address

Operation: **ADD A, @Ri**

(A) ← (A) + ((Ri))

Encoding:

0	0	1	0	0	1	1	i

Operation: **ADD A, #data**

(A) ← (A) + #data

Encoding:

0	0	1	0	0	1	0	0	immediate data

ADDC A, < src-byte>

Function: Add with carry

Description: ADDC simultaneously adds the byte variable indicated, the carry flag and the accumulator contents, leaving the result in the accumulator. The carry and auxiliary carry flags are set, respectively, if there is a carry out of bit 7 or bit 3, and cleared otherwise. When adding unsigned integers, the carry flag indicates an overflow occurred. OV is set if there is a carry out of bit 6 but not out of bit 7, or a carry out of bit 7 but not out of bit 6; otherwise OV is cleared. When adding signed integers, OV indicates a negative number produced as the sum of two positive operands or a positive sum from two negative operands. Four source operand addressing modes are allowed: register, direct, register indirect, or immediate.

Example: The accumulator holds 0C3H (11000011B) and register 0 holds 0AAH (10101010B) with the carry flag set. The instruction

```
ADDC A, R0
```

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

Operation: **ADDC A, Rn**

(A) ← (A) + (C) + (Rn)

Encoding:

0	0	1	1	1	r	r	r

Operation: **ADDC A, direct**

(A) ← (A) + (C) + (direct)

Encoding:

0	0	1	1	0	1	0	1	direct address

Operation: **ADDC A, @Ri**

(A) ← (A) + (C) + ((Ri))

Encoding:

0	0	1	1	0	1	1	i

Operation: **ADDC A, #data**

(A) ← (A) + (C) + #data

Encoding:

0	0	1	1	0	1	0	0	immediate data

AJMP addr11

Function: Absolute jump

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

Example: The label ”JMPADR” is at program memory location 0123H. The instruction

```
AJMP JMPADR
```

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

Operation: **AJMP addr11**

(PC) ← (PC) + 2

(PC_{10-0}) ← page address

Encoding:

a_{10}	a_9	a_8	**0**	**0**	**0**	**0**	**1**

a_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0

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

Function: Logical AND for byte variables

Description: ANL performs the bitwise logical AND operation between the variables indicated and stores the results in the destination variable. No flags are affected. The two operands allow six addressing mode combinations. When the destination is a accumulator, the source can use register, direct, register-indirect, or immediate addressing; when the destination is a direct address, the source can be the accumulator or immediate data.

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

Example: If the accumulator holds 0C3H (11000011B) and register 0 holds 0AAH (10101010B) then the instruction

```
ANL A,R0
```

will leave 81H (10000001B) in the accumulator. When the destination is a directly addressed byte, this instruction will clear combinations of bits in

any RAM location or hardware register. The mask byte determining the pattern of bits to be cleared would either be a constant contained in the instruction or a value computed in the accumulator at run-time. The instruction

```
ANL P1, #01110011B
```

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

Operation: **ANL A, Rn**

(A) ← (A) ∧ (Rn)

Encoding:

0	1	0	1	1	r	r	r

Operation: **ANL A, direct**

(A) ← (A) ∧ (direct)

Encoding:

0	1	0	1	0	1	0	1	direct address

Operation: **ANL A, @Ri**

(A) ← (A) ∧ ((Ri))

Encoding:

0	1	0	1	0	1	1	i

Operation: **ANL A, @Ri**

(A) ← (A) ∧ ((Ri))

Encoding:

0	1	0	1	0	1	1	i

Operation: **ANL A, #data**

(A) ← (A) ∧ #data

Encoding:

0	1	0	1	0	1	0	0	immediate data

Operation: **ANL direct, A**

(direct) ← (direct) ∧ (A)

Encoding:

0	1	0	1	0	0	1	0	direct address

Operation: **ANL direct, #data**

(direct) ← (direct) ∧ #data

Encoding:

0	1	0	1	0	0	1	1	direct address	immediate data

ANL C, <src-bit>

Function: Logical AND for bit variables

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

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

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

Operation: **ANL C, bit**

(C) ← (C) ∧ (bit)

Encoding:

1	0	0	0	0	0	1	0	bit address

Operation: **ANL C, /bit**

(C) ← (C) ∧ ¬ (bit)

Encoding:

1	0	1	1	0	0	0	0	bit address

CALL addr16

Function: Call subroutine.

Description: This is a generic call instruction which will be replaced by the assembler with a ACALL or LCALL instruction. The assembler will try to pick the most economical replacement.

Example: The instruction sequence

CALL MYSUB

will cause program execution to continue at the instruction at label MYSUB after pushing the return address into the stack.

Operation: **Please check the instructions ACALL and LCALL.**

Encoding: **Please check the instructions ACALL and LCALL.**

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

Function: Compare and jump if not equal

Description: CJNE 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. The carry flag is set if the unsigned integer value of <dest-byte> is less than the unsigned integer value of <src-byte>; otherwise, the carry is cleared. Neither operand is affected.

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

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

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

sets the carry flag and branches to the instruction at label NOT_EQ. By testing the carry flag, this instruction determines whether R7 is greater or less than 60H. If the data being presented to port 1 is also 34H, then the instruction

```
WAIT:   CJNE A,P1,WAIT
```

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

Operation: **CJNE A, direct, rel**

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

Encoding:

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

Operation: **CJNE A, #data, rel**

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

Encoding:

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

Operation: **CJNE Rn, #data, rel**

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

Encoding:

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

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

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

Encoding:

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

CLR A

Function: Clear accumulator

Description: The accumulator is cleared (all bits set to zero). No flags are affected.

Example: The accumulator contains 5CH (01011100B). The instruction

```
CLR A
```

will leave the accumulator set to 00H (00000000B).

Operation: **CLR**

(A) ← 0

Encoding:

1	1	1	0	0	1	0	0

CLR bit

Function: Clear bit

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

Example: Port 1 has previously been written with 5DH (01011101B). The instruction

```
CLR P1.2
```

will leave the port set to 59H (01011001B).

Operation: **CLR C**

(C) ←0

Encoding:

1	1	0	0	0	0	1	1

Operation: **CLR bit**

(bit) ← 0

Encoding:

1	1	0	0	0	0	1	0	bit address

CPL A

Function: Complement accumulator

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

Example: The accumulator contains 5CH (01011100B). The instruction

```
CPL  A
```

will leave the accumulator set to 0A3H (10100011B).

Operation: **CPL A**

$(A) \leftarrow \neg (A)$

Encoding:

1	**1**	**1**	**1**	**0**	**1**	**0**	**0**

CPL bit

Function: Complement bit

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

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

Example: Port 1 has previously been written with 5DH (01011101B). The instruction sequence

```
CPL P1.1
CPL P1.2
```

will leave the port set to 5BH (01011011B).

Operation: **CPL C**

(C)← ¬ (C)

Encoding:

1	1	0	0	0	0	1	1

Operation: **CPL bit**

(bit) ← ¬ (bit)

Encoding:

1	0	1	1	0	0	1	0	bit address

DA A

Function: Decimal adjust accumulator for addition

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

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

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

All of this occurs during the one instruction cycle. Essentially; this instruction performs the decimal conversion by adding 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.

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

```
ADDC A, R3
DA A
```

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

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

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

```
ADD A, #99H
DA A
```

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

Operation: **DA A**

contents of accumulator are BCD
if [[(A3-0) > 9] $\vee$ [(AC) = 1]]
then (A3-0) $\leftarrow$ (A3-0) + 6
and
if [[(A7-4) > 9] $\vee$ [(C) = 1]]
then (A7-4) $\leftarrow$ (A7-4) + 6

Encoding:

1	**1**	**0**	**1**	**0**	**1**	**0**	**0**

DEC byte

Function: Decrement

Description: The variable indicated is decremented by 1. An original value of 00H will underflow to 0FFH. No flags are affected. Four operand addressing modes are allowed: accumulator, register, direct, or register-indirect. *Note***:** When this instruction is used to modify an output port, the value used as the original port data will be read from the output data latch, not the input pins.

Example: Register 0 contains 7FH (01111111B). Internal RAM locations 7EH and 7FH contain 00H and 40H, respectively. The instruction sequence

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

will leave register 0 set to 7EH and internal RAM locations 7EH and 7FH set to 0FFH and 3FH.

Operation: **DEC A**

(A) ← (A) – 1

Encoding:

0	**0**	**0**	**1**	**0**	**1**	**0**	**0**

Operation: **DEC Rn**

(Rn) ← (Rn) – 1

Encoding:

0	**0**	**0**	**1**	**1**	**r**	**r**	**r**

Operation: **DEC direct**

(direct) ← (direct) – 1

Encoding:

0	**0**	**0**	**1**	**0**	**1**	**0**	**1**	direct address

Operation: **DEC @Ri**

((Ri)) ← ((Ri)) – 1

Encoding:

0	**0**	**0**	**1**	**0**	**1**	**1**	**i**

DIV AB

Function: Divide

Description: DIV AB divides the unsigned eight-bit integer in the accumulator by the unsigned eight-bit integer in register B. The accumulator receives the integer part of the quotient; register B receives the integer remainder. The carry and OV flags will be cleared. *Exception*: If B had originally contained 00H, the values returned in the accumulator and B register will be undefined and the overflow flag will be set. The carry flag is cleared in any case.

Example: The accumulator contains 251 (0FBH or 11111011B) and B contains 18 (12H or 00010010B). The instruction

DIV AB

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

Operation: **DIV AB**

(A), (B) ← (A) / (B)

Encoding:

1	**0**	**0**	**0**	**0**	**1**	**0**	**0**

DJNZ <byte>, < rel-addr>

Function: Decrement and jump if not zero

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

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

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

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

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

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

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

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

Operation: **DJNZ Rn, rel**

(PC) ← (PC) + 2
(Rn) ← (Rn) – 1
if (Rn) > 0 or (Rn) < 0
then (PC) ← (PC) + rel

Encoding:

1	**1**	**0**	**1**	**1**	**r**	**r**	**r**	rel. address

Operation: **DJNZ direct, rel**

(PC) ← (PC) + 2
(direct) ← (direct) – 1
if (direct) > 0 or (direct) < 0
then (PC) ← (PC) + rel

Encoding:

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

INC <byte>

Function: Increment

Description: INC increments the indicated variable by 1. An original value of 0FFH will overflow to 00H. No flags are affected. Three addressing modes are allowed: register, direct, or register-indirect. *Note***:** When this instruction is used to modify an output port, the value used as the original port data will be read from the output data latch, not the input pins.

Example: Register 0 contains 7EH (01111110B). Internal RAM locations 7EH and 7FH contain 0FFH and 40H, respectively. The instruction sequence

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

will leave register 0 set to 7FH and internal RAM locations 7EH and 7FH holding (respectively) 00H and 41H.

Operation: **INC A**

(A) ← (A) + 1

Encoding:

0	**0**	**0**	**0**	**0**	**1**	**0**	**0**

Operation: **INC Rn**

(Rn) ← (Rn) + 1

Encoding:

0	**0**	**0**	**0**	**1**	**r**	**r**	**r**

Operation: **INC direct**

(direct) ← (direct) + 1

Encoding:

0	**0**	**0**	**0**	**0**	**1**	**0**	**1**	direct address

Operation: **INC @Ri**

((Ri)) ← ((Ri)) + 1

Encoding:

0	**0**	**0**	**0**	**0**	**1**	**1**	**i**

INC DPTR

Function: Increment data pointer

Description: Increment the 16-bit data pointer by 1. A 16-bit increment (modulo 2^{16}) is performed; an overflow of the low-order byte of the data pointer (DPL) from 0FFH to 00H will increment the high-order byte (DPH). No flags are affected. This is the only 16-bit register which can be incremented.

Example: Registers DPH and DPL contain 12H and 0FEH, respectively. The instruction sequence

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

will change DPH and DPL to 13H and 01H.

Operation: **INC DPTR**

(DPTR) ← (DPTR) + 1

Encoding:

1	0	1	0	0	0	1	1

JB bit, rel

Function: Jump if bit is set

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

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

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

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

Operation: **JB bit, rel**

(PC) ← (PC) + 3
if (bit) = 1
then (PC) ← (PC) + rel

Encoding:

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

JBC bit, rel

Function: Jump if bit is set and clear bit

Description: If the indicated bit is one, branch to the address indicated; otherwise proceed with the next instruction. In either case, clear the designated bit. The branch destination is computed by adding the signed relative displacement in the third instruction byte to the PC, after incrementing the PC to the first byte of the next instruction. No flags are affected. *Note***:** When this instruction is used to test an output pin, the value used as the original data will be read from the output data latch, not the input pin.

Example: The accumulator holds 56H (01010110B). The instruction sequence

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

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

Operation: **JBC bit, rel**

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

Encoding:

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

JC rel

Function: Jump if carry is set

Description: If the carry flag is set, branch to the address indicated; otherwise proceed with the next instruction. The branch destination is computed by adding the signed relative displacement in the second instruction byte to the PC,

after incrementing the PC twice. No flags are affected.

Example: The carry flag is cleared. The instruction sequence

```
JC LABEL1
CPL C
JC LABEL2
```

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

Operation: **JC rel**

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

Encoding:

0	1	0	0	0	0	0	0	rel. address

JMP addr16

Function: Unconditional jump.

Description: This is a generic jump instruction which will be replaced by the assembler with a SJMP, AJMP, or LJMP instruction. The assembler will try to pick the most economical replacement.

Example: The instruction sequence

```
JMP MYLABEL
```

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

Operation: **Please check the instructions SJMP, AJMP, and LJMP.**

Encoding: **Please check the instructions SJMP, AJMP, and LJMP.**

JMP @A + DPTR

Function: Jump indirect

Description: Add the eight-bit unsigned contents of the accumulator with the sixteen-bit data pointer, and load the resulting sum to the program counter. This

will be the address for subsequent instruction fetches. Sixteen-bit addition is performed (modulo 2^{16}): a carry-out from the low-order eight bits propagates through the higher-order bits. Neither the accumulator nor the data pointer is altered. No flags are affected.

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

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

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

Operation: **JMP @A + DPTR**

(PC) ← (A) + (DPTR)

Encoding:

0	**1**	**1**	**1**	**0**	**0**	**1**	**1**

JNB bit, rel

Function: Jump if bit is not set

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

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

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

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

Operation: **JNB bit, rel**

(PC) ← (PC) + 3

if (bit) = 0
then (PC) ← (PC) + rel

Encoding:

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

JNC rel

Function: Jump if carry is not set

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

Example: The carry flag is set. The instruction sequence

```
JNC LABEL1
CPL C
JNC LABEL2
```

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

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

Encoding:

0	1	0	1	0	0	0	0	rel. address

JNZ rel

Function: Jump if accumulator is not zero

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

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

```
JNZ LABEL1
```

```
INC A
JNZ LABEL2
```

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

Operation: **JNZ rel**

(PC) ← (PC) + 2
if (A) ≠ 0
then (PC) ← (PC) + rel.

Encoding:

0	1	1	1	0	0	0	0	rel. address

JZ rel

Function: Jump if accumulator is zero

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

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

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

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

Operation: **JZ rel**

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

Encoding:

0	1	1	0	0	0	0	0	rel. address

LCALL addr16

Function: Long call

Description: LCALL calls a subroutine located at the indicated address. The

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

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

```
LCALL SUBRTN
```

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

Operation: **LCALL addr16**

(PC) ← (PC) + 3
(SP) ← (SP) + 1
((SP)) ← (PC7-0)
(SP) ← (SP) + 1
((SP)) ← (PC15-8)
(PC) ← addr15-0

Encoding:

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

LJMP addr16

Function: Long jump

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

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

```
LJMP JMPADR
```

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

Operation: **LJMP addr16**

(PC) ← addr15-0

Encoding:

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

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

Function: Move byte variable

Description: The byte variable indicated by the second operand is copied into the location specified by the first operand. The source byte is not affected. No other register or flag is affected. This is by far the most flexible operation. Fifteen combinations of source and destination addressing modes are allowed.

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

```
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
```

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

Operation: **MOV A, Rn**

(A) ← (Rn)

Encoding:

1	1	1	0	1	r	r	r

Operation: **MOV A, direct** [1]

(A) ← (direct)

Encoding:

1	1	1	0	0	1	0	1	direct address

Operation: **MOV A, @Ri**

(A) ← ((Ri))

Encoding:

1	1	1	0	0	1	1	i

[1] MOV A, ACC is not a valid instruction.

Operation: **MOV A, #data**

(A) ← #data

Encoding:

0	1	1	1	0	1	0	0	immediate data

Operation: **MOV Rn, A**

(Rn) ← (A)

Encoding:

1	1	1	1	1	r	r	r

Operation: **MOV Rn, direct**

(Rn) ← (direct)

Encoding:

1	0	1	0	1	r	r	r	direct address

Operation: **MOV Rn, #data**

(Rn) ← #data

Encoding:

0	1	1	1	1	r	r	r	immediate data

Operation: **MOV direct, A**

(direct) ← (A)

Encoding:

1	1	1	1	0	1	0	1	direct address

Operation: **MOV direct, Rn**

(direct) ← (Rn)

Encoding:

1	0	0	0	1	r	r	r	direct address

Operation: **MOV direct, direct**

(direct) ← (direct)

Encoding:

1	0	0	0	0	1	0	1	dir.addr. (src)	dir.addr. (dest)

Operation: **MOV direct, @Ri**

(direct) ← ((Ri))

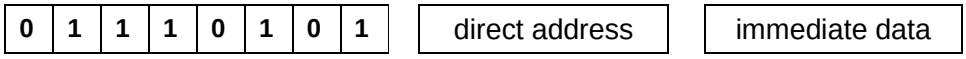
Encoding:

1	0	0	0	0	1	1	i	direct address

Operation: **MOV direct, #data**

(direct) ← #data

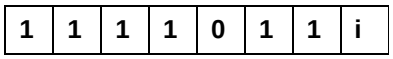
Encoding:



Operation: **MOV @ Ri, A**

((Ri)) ← (A)

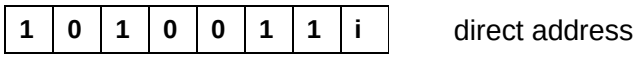
Encoding:



Operation: **MOV @ Ri, direct**

((Ri)) ← (direct)

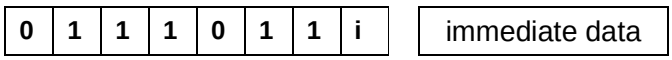
Encoding:



Operation: **MOV @ Ri, #data**

((Ri)) ← #data

Encoding:



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

Function: Move bit data

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

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

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

will leave the carry cleared and change port 1 to 39H (00111001 B).

Operation: **MOV C, bit**

(C) ← (bit)

Encoding:

1	0	1	0	0	0	1	0	bit address

Operation: **MOV bit, C**

(bit) ← (C)

Encoding:

1	0	0	1	0	0	1	0	bit address

MOV DPTR, #data16

Function: Load data pointer with a 16-bit constant

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

Example: The instruction

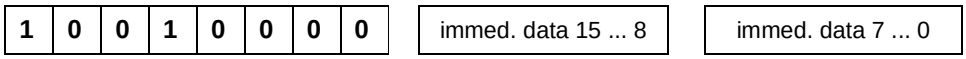
```
MOV DPTR, #1234H
```

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

Operation: **MOV DPTR, #data16**

(DPTR) ← #data15-0

Encoding:



MOVC A, @A + <base-reg>

Function: Move code byte

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

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

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

```
        DB 77H
        DB 88H
        DB 99H
```

If the subroutine is called with the accumulator equal to 01H, it will return 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.

Operation: **MOVC A, @A + DPTR**

(A) ← ((A) + (DPTR))

Encoding:

1	**0**	**0**	**1**	**0**	**0**	**1**	**1**

Operation: **MOVC A, @A + PC**

(PC) ← (PC) + 1

(A) ← ((A) + (PC))

Encoding:

1	**0**	**0**	**0**	**0**	**0**	**1**	**1**

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

Function: Move expanded

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

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

In the second type of MOVX instructions, the data pointer generates a sixteen-bit address. P2 outputs the high-order eight address bits (the contents of DPH) while P0 multiplexes the low-order eight bits (DPL) with data. The P2 special function register retains its previous contents while the P2 output buffers are emitting the contents of DPH. This form is faster and more efficient when accessing very large data arrays (up to 64 Kbyte), since no additional instructions are needed to set up the output ports. It is possible in some situations to mix the two MOVX types. A large RAM array with its high-order address lines driven by P2 can be addressed via the data pointer, or with code to output high-order address

bits to P2 followed by a MOVX instruction using R0 or R1.

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

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

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

Operation: **MOVX A, @Ri**

(A) ← ((Ri))

Encoding:

1	**1**	**1**	**0**	**0**	**0**	**1**	**i**

Operation: **MOVX A, @DPTR**

(A) ← ((DPTR))

Encoding:

1	**1**	**1**	**0**	**0**	**0**	**0**	**0**

Operation: **MOVX @Ri, A**

((Ri)) ← (A)

Encoding:

1	**1**	**1**	**1**	**0**	**0**	**1**	**i**

Operation: **MOVX @DPTR, A**

((DPTR)) ← (A)

Encoding:

1	**1**	**1**	**1**	**0**	**0**	**0**	**0**

MUL AB

Function: Multiply

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

Example: Originally the accumulator holds the value 80 (50H). Register B holds the value 160 (0A0H). The instruction

```
MUL AB
```

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

Operation: **MUL AB**

(A7-0), (B15-8) ← (A) x (B)

Encoding:

1	0	1	0	0	1	0	0

NOP

Function: No operation

Description: Execution continues at the following instruction. Other than the PC, no registers or flags are affected.

Example: It is desired to produce a low-going output pulse on bit 7 of port 2 lasting exactly 5 cycles. A simple SETB/CLR sequence would generate a one-cycle pulse[2], so four additional cycles must be inserted. This may be done (assuming no interrupts are enabled) with the instruction sequence

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

Operation: **NOP**

Encoding:

0	0	0	0	0	0	0	0

[2] This is for the original 8051 from Intel, where one cycle takes 12 clock periods. Modern variants of the 8051 microcontroller, in particular those running at one cycle per clock period, may take more than one cycle to execute the SETB/CLR instructions. Please refer the datasheet of the particular microcontroller your are using for more details.

ORL <dest-byte> <src-byte>

Function: Logical OR for byte variables

Description: ORL performs the bitwise logical OR operation between the indicated variables, storing the results in the destination byte. No flags are affected . The two operands allow six addressing mode combinations. When the destination is the accumulator, the source can use register, direct, register-indirect, or immediate addressing; when the destination is a direct address, the source can be the accumulator or immediate data. *Note***:** When this instruction is used to modify an output port, the value used as the original port data will be read from the output data latch, not the input pins.

Example: If the accumulator holds 0C3H (11000011B) and R0 holds 55H (01010101B) then the instruction

```
ORL A, R0
```

will leave the accumulator holding the value 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 either a constant data value in the instruction or a variable computed in the accumulator at run-time. The instruction

```
ORL P1, #00110010B
```

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

Operation: **ORL A, Rn**

(A) ← (A) ∨ (Rn)

Encoding:

0	**1**	**0**	**0**	**1**	**r**	**r**	**r**

Operation: **ORL A, direct**

(A) ← (A) ∨ (direct)

Encoding:

0	**1**	**0**	**0**	**0**	**1**	**0**	**1**	direct address

Operation: **ORL A, @Ri**

(A) ← (A) ∨ ((Ri))

Encoding:

0	**1**	**0**	**0**	**0**	**1**	**1**	**i**

Operation: **ORL A, #data**

(A) ← (A) ∨ #data

Encoding:

0	1	0	0	0	1	0	0	immediate data

Operation: **ORL direct, A**

(direct) ← (direct) ∨ (A)

Encoding:

0	1	0	0	0	0	1	0	direct address

Operation: **ORL direct, #data**

(direct) ← (direct) ∨ #data

Encoding:

0	1	0	0	0	0	1	1	direct address	immediate data

ORL C, <src-bit>

Function: Logical OR for bit variables

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

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

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

Operation: **ORL C, bit**

(C) ← (C) ∨ (bit)

Encoding:

0	1	1	1	0	0	1	0	bit address

Operation: **ORL C, /bit**

(C) ← (C) ∨ ¬ (bit)

Encoding:

1	0	1	0	0	0	0	0	bit address

POP direct

Function: Pop from stack

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

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

```
POP  DPH
POP  DPL
```

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

```
POP  SP
```

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

Operation: **POP direct**

(direct) ← ((SP))
(SP) ← (SP) – 1

Encoding:

1	**1**	**0**	**1**	**0**	**0**	**0**	**0**	direct address

PUSH direct

Function: Push onto stack

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

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

```
PUSH  DPL
PUSH  DPH
```

will leave the stack pointer set to 0BH and store 23H and 01H in internal RAM locations 0AH and 0BH, respectively.

Operation: **PUSH direct**

(SP) ← (SP) + 1
((SP)) ← (direct)

Encoding:

1	1	0	0	0	0	0	0	direct address

RET

Function: Return from subroutine

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

Example: The stack pointer originally contains the value 0BH. Internal RAM locations 0AH and 0BH contain the values 23H and 01H, respectively. The instruction

RET

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

Operation: **RET**

$(PC_{15-8}) \leftarrow ((SP))$
$(SP) \leftarrow (SP) - 1$
$(PC_{7-0}) \leftarrow ((SP))$
$(SP) \leftarrow (SP) - 1$

Encoding:

0	0	1	0	0	0	1	0

RETI

Function: Return from interrupt

Description: RETI pops the high and low-order bytes of the PC successively from the stack, and restores the interrupt logic to accept additional interrupts at the same priority level as the one just processed. The stack pointer is left decremented by two. No other registers are affected; the PSW is not automatically restored to its pre-interrupt status. Program execution

continues at the resulting address, which is generally the instruction immediately after the point at which the interrupt request was detected. If a lower or same-level interrupt is pending when the RETI instruction is executed, that one instruction will be executed before the pending interrupt is processed.

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

```
RETI
```

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

Operation: **RETI**

$(PC_{15-8}) \leftarrow ((SP))$
$(SP) \leftarrow (SP) - 1$
$(PC_{7-0}) \leftarrow ((SP))$
$(SP) \leftarrow (SP) - 1$

Encoding:

0	**0**	**1**	**1**	**0**	**0**	**1**	**0**

RL A

Function: Rotate accumulator left

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

Example: The accumulator holds the value 0C5H (11000101B). The instruction

```
RL  A
```

leaves the accumulator holding the value 8BH (10001011B) with the carry unaffected.

Operation: **RL A**

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

Encoding:

0	**0**	**1**	**0**	**0**	**0**	**1**	**1**

RLC A

Function: Rotate accumulator left through carry flag

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

Example: The accumulator holds the value 0C5H (11000101B), and the carry is zero. The instruction

RLC A

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

Operation: **RLC A**

(An + 1) ← (An) n = 0-6
(A0) ← (C)
(C) ← (A7)

Encoding:

0	**0**	**1**	**1**	**0**	**0**	**1**	**1**

RR A

Function: Rotate accumulator right

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

Example: The accumulator holds the value 0C5H (11000101B). The instruction

RR A

leaves the accumulator holding the value 0E2H (11100010B) with the carry unaffected.

Operation: **RR A**

(An) ← (An + 1) n = 0-6
(A7) ← (A0)

Encoding:

0	**0**	**0**	**0**	**0**	**0**	**1**	**1**

RRC A

Function: Rotate accumulator right through carry flag

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

Example: The accumulator holds the value 0C5H (11000101B), the carry is zero. The instruction

RRC A

leaves the accumulator holding the value 62H (01100010B) with the carry set.

Operation: **RRC A**

(An) ← (An + 1) n=0-6
(A7) ← (C)
(C) ← (A0)

Encoding:

0	**0**	**0**	**1**	**0**	**0**	**1**	**1**

SETB <bit>

Function: Set bit

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

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

```
SETB C
SETB P1.0
```

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

Operation: **SETB C**

(C) ←1

Encoding:

1	**1**	**0**	**1**	**0**	**0**	**1**	**1**

Operation: **SETB bit**

(bit) ← 1

Encoding:

1	1	0	1	0	0	1	0	bit address

SJMP rel

Function: Short jump

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

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

SJMP RELADR

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

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

Operation: **SJMP rel**

(PC) ← (PC) + 2

(PC) ← (PC) + rel

Encoding:

1	0	0	0	0	0	0	0	rel. address

SUBB A, <src-byte>

Function: Subtract with borrow

Description: SUBB subtracts the indicated variable and the carry flag together from the accumulator, leaving the result in the accumulator. SUBB sets the carry (borrow) flag if a borrow is needed for bit 7, and clears C otherwise. (If C was set before executing a SUBB instruction, this indicates that a borrow was needed for the previous step in a multiple precision

subtraction, so the carry is subtracted from the accumulator along with the source operand). AC is set if a borrow is needed for bit 3, and cleared otherwise. OV is set if a borrow is needed into bit 6 but not into bit 7, or into bit 7 but not bit 6.

When subtracting signed integers OV indicates a negative number produced when a negative value is subtracted from a positive value, or a positive result when a positive number is subtracted from a negative number. The source operand allows four addressing modes: register, direct, register indirect, or immediate.

Example: The accumulator holds 0C9H (11001001B), register 2 holds 54H (01010100B), and the carry flag is set. The instruction

```
SUBB  A, R2
```

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

Notice that 0C9H minus 54H is 75H. The difference between this and the above result is due to the (borrow) flag being set before the operation. If the state of the carry is not known before starting a single or multiple-precision subtraction, it should be explicitly cleared by a CLR C instruction.

Operation: **SUBB A, Rn**

(A) ← (A) – (C) – (Rn)

Encoding:

1	0	0	1	1	r	r	r

Operation: **SUBB A, direct**

(A) ← (A) – (C) – (direct)

Encoding:

1	0	0	1	0	1	0	1	direct address

Operation: **SUBB A, @ Ri**

(A) ← (A) – (C) – ((Ri))

Encoding:

1	0	0	1	0	1	1	i

Operation: **SUBB A, #data**

(A) ← (A) – (C) – #data

Encoding:

1	0	0	1	0	1	0	0	immediate data

SWAP A

Function: Swap nibbles within the accumulator

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

Example: The accumulator holds the value 0C5H (11000101B). The instruction

```
SWAP  A
```

leaves the accumulator holding the value 5CH (01011100B).

Operation: **SWAP A**

(A3-0) (A7-4), (A7-4) ← (A3-0)

Encoding:

1	**1**	**0**	**0**	**0**	**1**	**0**	**0**

XCH A, <byte>

Function: Exchange accumulator with byte variable

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

Example: R0 contains the address 20H. The accumulator holds the value 3FH (00111111B). Internal RAM location 20H holds the value 75H (01110101B). The instruction

```
XCH A, @R0
```

will leave RAM location 20H holding the value 3FH (00111111 B) and 75H (01110101B) in the accumulator.

Operation: **XCH A, Rn**

(A)↔ (Rn)

Encoding:

1	**1**	**0**	**0**	**1**	**r**	**r**	**r**

Operation: **XCH A, direct**
(A) ↔ (direct)

Encoding:

1	1	0	0	0	1	0	1	direct address

Operation: **XCH A, @Ri**
(A) ↔ ((Ri))

Encoding:

1	1	0	0	0	1	1	i

XCHD A, @Ri

Function: Exchange digit

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

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

```
XCHD A, @R0
```

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

Operation: **XCHD A, @Ri**
(A3-0) ↔ ((Ri)3-0)

Encoding:

1	1	0	1	0	1	1	i

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

Function: Logical Exclusive OR for byte variables

Description: XRL performs the bitwise logical Exclusive OR operation between the indicated variables, storing the results in the destination. No flags are affected.

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

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

Example: If the accumulator holds 0C3H (11000011B) and register 0 holds 0AAH (10101010B) then the instruction

```
XRL A, R0
```

will leave the accumulator holding the value 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
```

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

Operation: **XRL A, Rn**

(A) ← (A) $\veebar$ (Rn)

Encoding:

0	1	1	0	1	r	r	r

Operation: **XRL A, direct**

(A) ← (A) $\veebar$ (direct)

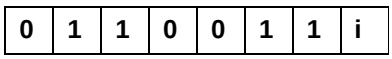
Encoding:

0	1	1	0	0	1	0	1	direct address

Operation: **XRL A, @Ri**

(A) ← (A) $\veebar$ ((Ri))

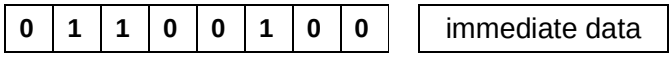
Encoding:



Operation: **XRL A, #data**

(A) ← (A) ⊻ #data

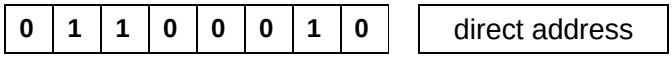
Encoding:



Operation: **XRL direct, A**

(direct) ← (direct) ⊻ (A)

Encoding:



Operation: **XRL direct, #data**

(direct) ← (direct) ⊻ #data

Encoding: