signetics

TWIN Technical Note Software

Number : SW005

Subject: 8048 Assembler on TWIN

Date : 1979-02-14

Philips Elcoma welcomes you as a new user of the 8048 family assembler.

The first part of this guide describes the use on TWIN and some points that need your special attention.
The latter part describes the assembly language.

This guide only provides you with an insight in the 8048 assembly language. For information and explanation of the TWIN system you are assumed to be familiar with the manuals describing the TWIN system.

J.C. Cranendonk

PART ONE

1. INTRODUCTION

With the 8048 assembler you are now able to generate from an 8048 source code an object code (8048 instructions) that can be loaded into the TWIN memory.

2. EXECUTION

The assembler is executed by the standard command on TWIN to execute 2650 programs on the slave side called: XEQ.
Total format: XEQ ASM8048/x ①,②,③

where:

ASM8048	is the program name of the 8048 assembler to be executed
x	is the drive number of the diskette on which the 8048 assembler is residing
①	is the file identification for the 8048 source code file (name + drive)
②	is the file identification for the file that will be used to contain the listing file (name + drive, e.g.: LPT1 or OUTLIST/1 or ... etc.)
③	is the file identification for the file that will be used to save the generated object program in a format that is defined as TWIN-HEX file format (name + drive)

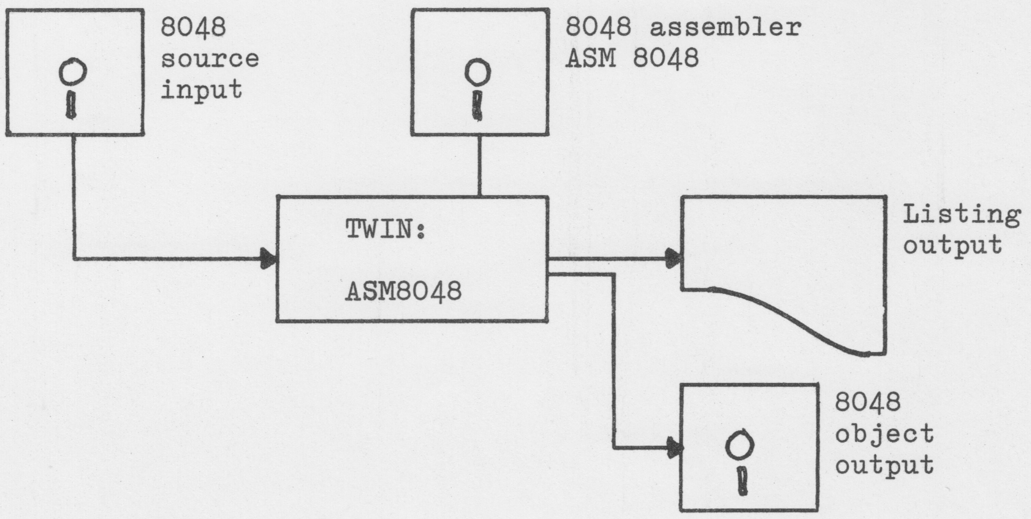


Fig. 1

Note: The output of the 8048 assembler on TWIN is in the TWIN-HEX format and can therefore be loaded into the TWIN slave memory with the RHEX command. (for description of RHEX command see TWIN documentation)

3. INPUTS

Instruction format.

The 8048 assembly language instructions and assembler directives consist of up to four fields as follows:

```
Ø....5....Ø....5....Ø...............................
LABEL     OPCODE     OPERAND     COMMENT
```

The label and comment field are always optional.

The operand field may contain zero, one or two operands depending on the opcode specified. Any number of blanks can seperate fields. The entire instruction must be entered on one line terminated by a carriage return and line feed. No continuation lines are possible, though you may have lines consisting entirely of comments.

Note 1: label field.

A label can be one to four alpha numeric characters, with the first character alphabetic. The label should not be followed by a colon.

Note 2: comment field.

The comment field must start with a semicolon (;) when it is the only field used, when other fields are preceeding the comment field the semicolon is optional.

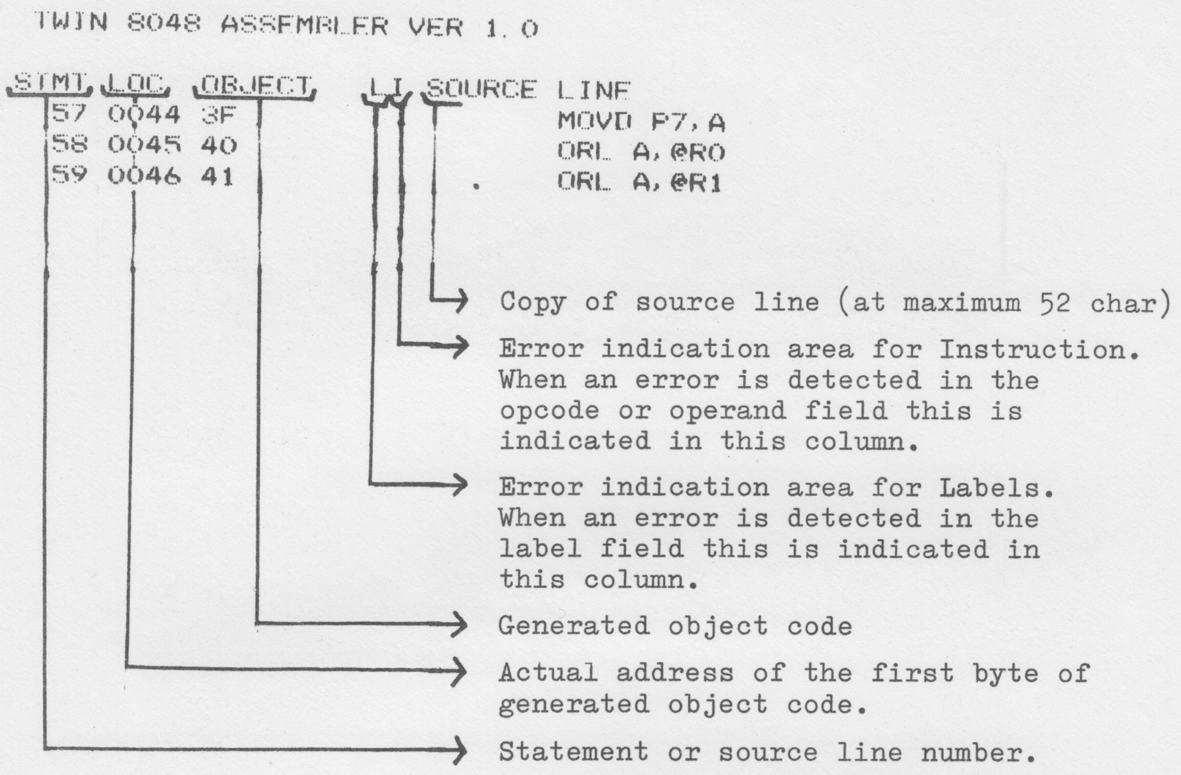
Note 3: operand field.

In some cases the operand field is not completely checked. When the leading part is unique this part is used to identify the operation the latter part is not checked. It remains, however, advisable to write syntactically right programs to be sure that future conversion of programs remains possible without large modifications.

4. LISTING OUTPUT

The listing output is generated in a printable format. It can be written to a disk file for later print out by a PRINT or COPY statement.

Layout:



```
TWIN 8048 ASSEMBLER VER 1.0

STMT LOC  OBJECT   LI SOURCE LINE
 57 0044 3F               MOVD P7,A
 58 0045 40               ORL  A,@R0
 59 0046 41               ORL  A,@R1
```

Copy of source line (at maximum 52 char)

Error indication area for Instruction. When an error is detected in the opcode or operand field this is indicated in this column.

Error indication area for Labels. When an error is detected in the label field this is indicated in this column.

Generated object code

Actual address of the first byte of generated object code.

Statement or source line number.

5. OBJECT CODE OUTPUT

The object code output is generated in the TWIN HEX format. (see TWIN Operator Guide).

6. ERROR INDICATIONS

Errors indicated in label area (errors concerning the label definitions).

S : Syntactical error in label definition (too long, illegal characters used, not starting with alphabetic. reserved word used).

D : Duplication. The label has been defined before.

T : Too many label definitions used (the maximum number of labels to be used in 255!).

Errors indicated in the instruction area

P : Page error. A reference is tried to be made outside the current page in an instruction that does not allow for a page cross.

I : Inconnu. The label referred to is not defined.

V : Value violation. This error is given when a numeric value is too large, also for register numbers, port numbers etc.

C : Code error. Wrong operation code used.

PART TWO

1. ASSEMBLER CONCEPTS

Assembly language

If you have ever written a computer program in a machine-recognizable form such as binary code, you will be particularly appreciative of the advantages of programming in a symbolic assembly language. Assembly-language operation codes (opcodes) are easily remembered (for example, MOV for a 'move' instruction, JMP for a 'jump'). You can also express symbolically the addresses and values referenced in the operand field of assembly language instructions. The names for these operands can be selected to suggest their purpose, making them as mnemonic as the opcodes.

The program consisting of assembly language instructions is called a source program. This program is passed through an assembler, which performs the clerical task of translating symbolic code into object code recognizable by the 8048 microcomputers.

The source file passed to the assembler actually includes more than source program instructions. It also includes assembler directives and (possibly) assembler controls. Only source program instructions are converted into executable object code, however. The assembler directives and controls initiate various functions that assist and direct the assembler in its translation operation.

Assembler output consists of three possible files: the object file containing your program code in machine-executable form, the list file printout of your source code and object code.

Instruction format

8048 assembly-language instructions and assembler directives consist of up to four fields as follows:

```
Label      Opcode     Operand     ;Comment
```

The label and comment fields are always optional. The operand field may contain zero, one, or two operands depending on the opcode specified. Any number of blanks can separate fields. The entire instruction must be entered on one line, terminated by a carriage return and line feed. No continuation lines are possible, though you may have lines consisting entirely of comments.

Label Field.

An instruction label is a symbol name whose value is the specific memory location where the instruction resides. It is optional. A label can be one to four alphanumeric characters, with the first character alphabetic. A symbol used as a label cannot be redefined elsewhere in your program. (See 'Symbols and Symbol Tables' later in this chapter.)

Opcode Field.

This field contains the mnemonic operation code for the 8048 instruction or assembler directive to be performed. It is terminated by a blank or nonalphanumeric character, or by a carriage return and line feed if no operand or comment field is present.

Operand field.

The operand field identifies the data to be operated on by the specified instruction opcode. Some instructions require no operand. Others require one or two operands. In the latter case, the operands are separated by a comma. As a general rule, when two operands are required (data transfer, addition, and logical operations), the first operand specifies the destination (or target) of the operation's result and the second operand specifies the source data.

```
ADD A,R3            ;ADD CONTENTS OF REG 3 TO ACC
ANL A,R3            ;LOGICAL 'AND' CONTENTS OF ACC
                    ;WITH MASK CONTAINED IN REG 3
MOV R1, #OFFH       ;MOVE 'FF' HEX (ONES) INTO REG 1
```

Operands can reference directly data contained in 8048 registers such as the PSW, accumulator, or data memory working registers 0-7.

```
MOV A,PSW           ;MOVE PSW CONTENTS TO ACC
XCH A,R4            ;EXCHANGE ACC DATA WITH
                    ;REG 4 DATA
```

All data memory locations can be accessed indirectly by prefacing a reference to Register 0 or 1 with a 'commercial at' sign (@).

```
MOV @RO,A           ;MOVE ACC CONTENTS TO DATA MEMORY
                    ;LOCATION WHOSE ADDRESS IS
                    ;SPECIFIED IN REG O
```

The JMPP instruction allows program memory locations to be accessed indirectly by prefacing an accumulator reference with

```
JMPP   A               ;CONTENTS OF PROGRAM MEMORY LOCATION
                        POINTED TO BY
                       ;ACC ARE SUBSTITUTED FOR BITS 0-7 OF
                        PROGRAM COUNTER.
```

Operands can contain 'immediate' data. The desired value is inserted directly into the operand field. All immediate data must be prefixed with a pound sign ( ) to distinguish it from register data and must evaluate to eight bits.

Immediate data can be in the form of an ASCII constant (a character enclosed in single quotes), a number, an expression to be evaluated at assembly time, or a symbol name. To indicate a quote as an ASCII constant, show the quote as two consecutive single quotes ("). Any symbol appearing in the operand field must be previously defined.

```
MOV A,  A              ;MOVE THE VALUE OF ASCII
                       ;CONSTANT 'A' (01000001)
                       ;INTO ACC
ADD A,  OAH            ;ADD HEX 'OA' (00001010)
                       ;TO ACC
```

Finally, the operand field of a jump instruction (that is, the address to be jumped to) can be expressed as a symbolic label, as an absolute 12-bit program memory address, or as an expression that can be evaluated to such an address. In no case is this operand preceded by a pound sign.

```
JMP START              ;JUMP TO THE LOCATION LABELED 'START'
JMP 200H               ;JUMP TO LOCATION 200 HEX (512 DECIMAL)
```

Expression evaluation and symbols are discussed in more detail in the next two sections of this chapter.

Comment Field.

The comment field can contain any information you deem useful for annotating your program. The only stipulation is that this field be preceded by a semicolon.

Arithmetic operations

When discussing arithmetic operations, we must distinguish between operations performed by your program when it is executed (such as ADD A,R5) and expression evaluation performed by the assembler at assembly time (such as

MOV A, P+3x(X/2). Numbers are represented indentically in both cases, but your program has considerably more flexibility than the assembler in determining the range of numbers, internal notation, and whether numbers are to be considered signed or unsigned. The characteristics of both modes of arithmetic are summarized in Figure 2 and discussed in more detail in the following subsections.

Number Characteristic	Assembly–Time Expression Evaluation	Program Execution Arithmetic
Base Representation	Binary, Octal, Decimal, or Hexadecimal	Binary, Octal, Decimal, or Hexadecimal
Range	0-(64K-1)	User Controlled
Evaluates To:	16 Bits	User Interpretation
Internal Notation	Two's Complement	Two's Complement
Signed/Unsigned Arithmetic	Unsigned	Unsigned Unless User Manipulates

Figure 2 Number Representation

Number Base Representation.

Numbers can be expressed in decimal, hexadecimal, octal, or binary form. A hexadecimal number must begin with a decimal digit and have the suffix 'H' (for example: 3AH, OFFH, 12H). Octal values must have one of the suffixes 'O' or 'Q' (for example: 760,53Q). Binary numbers must have the suffix 'B' (for example: 10111010B). Decimal numbers can be suffixed optionally by 'D' (for example: 512, 512D). Where no suffix is present, decimal is assumed.

Permissible Range of Numbers.

In general, numbers can range between 0 and 65,535 (OFFFFH). Numbers outside this range are evaluated 'modulo' 64K (that is, a number greater than 64K is divided by 64K and the remainder substituted for the original number). All expressions can be evaluated to 16 bits.

Certain limitations must be applied within this general range, however. For example, most program execution arithmetic is done using the 8-bit accumulator or 8-bit registers and most results evaluate to 8 bits. To work with large numbers would require manipulation of register pairs.

If you are doing signed arithmetic, the high-order bit of each number is used to indicate the sign of that number (0 if positive, 1 if negative). Consequently, the remaining bits can only express a number in the range -32,768 to +32,767 for 16-bit arithmetic. For 8-bit arithmetic, the range is -128 to +127.

If a number is too large for its intended use, either an error results or modulo arithmetic is performed. For example:

o Program memory addresses must be in the range 0-4095 (12 bits). In some cases, an address reference must be 'within page! that is, within the range 0-255 (8 bits).

o Data memory addresses must be in the range 0-255 (8bits).

o Operands containing 8-bit immediate data must evaluate to an 8-bit number.

o Expressions in a DB assembler directive (except strings) must evaluate to 8 bits.

<u>Two's Complement Arithmetic.</u>

Two's complement notation allows subtraction to be performed by a series of bit complementations and additions (thus reducing the circuitry requirements of a processor). A number is converted to two's complement form by complementing all its bits and adding a binary one to the result.

When a number is interpreted as a signed two's complement number, the low-order bits supply the magnitude of the number and the high-order bit is interpreted as the sign of the number. As was mentioned above, the range of a signed two's complement value is -32,768 to +32,767 (for 16 bits) and -128 to +127 (for 8 bits).

When a 16-bit value is interpreted as an unsigned two's complement number, it is considered to be positive and in the range 0-76,535. An 8-bit value is in the range 0-255.

The assemblers perform all expression evaluation assuming unsigned two's complement numbers. Similarly, execution-time arithmetic normally assumes unsigned two's complement notation, but you can perform signed arithmetic by isolating and inspecting the high-order bit with the instruction:

```
JB7 MINUS          ;IF ACC BIT 7=1 GO TO 'MINUS' ROUTINE
```

The MCS-48 instruction set does not include a subtraction instruction. Subtraction is done by complementing the accumulator and proceeding as in a normal two's complement addition operation. The CPL A (complement accumulator) instruction performs a straight binary one's complement. You must perform the binary addition of one, necessary to convert the number to two's complement notation, yourself.

Example: Subtract 1AH from 63H using signed two's complement notation.

```
    MOV A,  1AH          ;MOVE '1AH' INTO ACC (00011010)
    CPL A                ;ONE'S COMPLEMENT ACC (11100101)
    INC A                ;CONVERT TO TWO'S COMPLEMENT
                         ;(11100110)
    ADD A,  63H          ;ADD '63' TO VALUE IN ACC (01001001)
    JB7 MINUS            ;IF ACC BIT 7=1 GO TO 'MINUS' ROUTINE
```

The result is +49H.

Assembly - Time Expression Evaluation

An expression is a combination of numbers, symbols, and operators. The latter can be arithmetic, relational, and logical operators or specially-defined operators. Any symbol appearing in an expression must have a previously-defined absolute value.

The ASCII characters 'null' and 'rubout' are ignored on input, but the null string can be represented by two consecutive quotes or by a missing operand. The null string is illegal in any context that requires numerical evaluation.

Operators.

The assembler includes five groups of operators that permit the following assembly-time operations: arithmetic, bit shifting operations, logical evaluation, value comparison, and byte isolation. These are all assembly-time operations. Once the assembler has evaluated an expression, it becomes a permanent part of your program.

Arithmetic Operators.

The arithmetic operators are as follows:

Operator	Meaning
+	Unary or binary addition
-	Unary or binary subtraction
x	Multiplication
/	Division. Any remainder is discarded (7/3=2)

Example:

The following expressions generate the bit pattern for the ASCII character A:

```
5+30x2
(25/5)+30x2
5+(-30x-2)
```

Precedence of Operators

Expressions are evaluated left to right. Operators with higher precedence are evaluated before other operators that immediately precede or follow them. When two operators have equal precedence, the leftmost is evaluated first.

Parentheses can be used to override normal rules of precedence. The part of an expression enclosed in parentheses is evaluated first. If parentheses are nested, the innermost are evaluated first.

```
15/3+18/9    = 5 + 2 = 7
15/(3+18/9) = 15/3+2) = 15/5 = 3
```

The following list describes the classes of operators in order of precedence:

o Parenthesized expressions
o Multiplication/Division:x,/
o Addition/Subtraction:+,- (Unary and binary)

Symbols and symbol tables

Symbolic Addressing

If you have never done symbolic programming before, the following analogy may help clarify the distinction between a 'symbolic' and an 'absolute' address.

The locations in program memory can be compared to a cluster of post office boxes. Suppose Richard Roe rents box 500 for two months. He can then ask for his letters by saying 'Give me the mail in box 500', or 'Give me the mail for Roe'. If Donald Doe later rents box 500, he too can ask for his mail by either box number 500 or by his name.
The content of the post office box can be accessed by a fixed, absolute address (500) or by a symbolic , variable name. The postal clerk correlates the symbolic names and their absolute values in his log book.

The assembler, performs the same function, keeping track of symbols and their values in a symbol table. Note that you do not have to assign values to symbolic addresses. The assembler references its location counter during the assembly process to calculate these addresses for you. (The location counter does for the assembler what the program counter does for the microcomputer. It tells the assembler where the next instruction or operand is to be placed in memory.)

Symbol Characteristics

A symbol can contain one to four alphabetic (A-Z) or numeric (0-9) characters (with the first character alphabetic). A dollar sign can be used as a symbol to denote the value currently in the location counter. For example, the command

```
JMP $+6
```

forces a jump to the instruction residing six memory locations higher than the JMP instruction.

The assemblers regard symbols as being reserved or user-defined, global or limited, permanent or redefinable. All symbols are absolute, that is, fixed to some absolute memory address or fixed-value expression unaffected by program loading.

Reserved and User-Defined symbols.

The '$' symbol and following 8048 instruction-set opcodes are reserved and should not be specified as user-defined symbols

ADD	ENT0	JNI	MOVD	RL
ADDC	IN	JNIBF	MOVP	RLC
ANL	INC	JNT0	MOVP3	RR
ANLD	INS	JNT1	MOVX	RRC
CALL	JBn	JNZ	NOP	SEL
CLR	JC	JOBF	ORL	STOP
CPL	JF0	JTF	ORLD	STRT
DA	JF1	JT0	OUT	SWAP
DEC	JMP	JT1	OUTL	XCH
DIS	JMPP	JZ	RET	XCHD
DJNZ	JNC	MOV	RETR	XRL
EN				

The following instruction operand symbols and symbols required by the assembler are also reserved:

Symbol	Meaning
A	Accumulator
R0	Register 0
R1	Register 1
R2	Register 2
R3	Register 3
R4	Register 4
R5	Register 5
R6	Register 6
R7	Register 7
PSW	Program Status Word
BUS	BUS Port
P0	I/O Port 0 (8021)
P1	I/O Port 1
P2	I/O Port 2
P4	I/O Port 4
P5	I/O Port 5
P6	I/O Port 6
P7	I/O Port 7
C	Carry Flag
T	Timer Register
CNT	Counter Register
TCNT	Timer/Counter
RB0	Register Bank 0
RB1	Register Bank 1
MB0	Memory Bank 0
MB1	Memory Bank 1
I	Interrupt
TCNTI	Timer/Counter Interrupt
F0	Flag 0
F1	Flag 1

Finally, the following directives vannot be used as symbols except in a limited context:

```
DB      END     EQU     ORG
DS
DW                      SET
                        EJE
                        SPC
```

User-defined symbols are symbols you create to reference instruction addresses and data. These symbols are defined when they appear in the label field of an instruction or in the name field of EQU or SET assembler directives. Values for these symbols are determined modulo 64K although specific environments may limit the value even further. (See the subsection 'Permissible Range of Numbers, 'earlier in this chapter.) Values outside these ranges cause an error.

NOTE: Only instructions that allow registers as operands may have register-type operands. Expressions containing register-type operands are flagged as errors. The only assembler directives that may contain register-type operands are EQU and SET. Registers can be assigned alternate names only by EQU or SET.

Permanent and Redefinable Symbols.

Most symbols are permanent, that is, their values cannot change during the assembly operation. Only symbols defined with the SET assembler directive are redefinable.

2. 8048 ASSEMBLY LANGUAGE INSTRUCTIONS

SYMBOLS AND ABBREVIATIONS USED

A	Accumulator
AC	Auxillary Carry
addr	12-Bit Program Memory Address
Bb	Bit Designator (b=0-7)
BS	Bank Switch
BUS	BUS Port
C	Carry
CLK	Clock
CNT	Event Counter
D	Mnemonic for 4-Bit Digit (Nibble)
data	8-Bit Number or Expression
DBF	Memory Bank Flip-Flop
FO, F1	Flag 0, Flag 1
I	Interrupt
P	Mnemonic for in-page Operation
PC	Program Counter
Pp	Port Designator (p=1, 2 or 4-7)
PSW	Program Status Word
Rr	Register Designator (r=0, 1 or 0-7)
SP	Stack Pointer
T	Timer
TF	Timer Flag
TO, T1	Test 0, Test 1
X	Mnemonic for External RAM
	Immediate Data Prefix
	Indirect Address Prefix
$	Current Value of Program Counter
(X)	Contents of X
((X))	Contents of Location Addressed by X
	Is Replaced by

NOTE: In the examples label definitions do end with a colon, the assembler described in this guide will not accept these definitions. Also the max nr. of characters for a label definition is <u>four</u>.

ADD A,R_r Add Register Contents to Accumulator

| 0110 | 1rrr |

The contents of register 'r' are added to the accumulator. Carry is affected.

(A) ← (A) + (Rr)   r=0-7

Example:
```
ADDREG: ADD A,R6      ;ADD REG 6 CONTENTS
                      ;TO ACC
```

ADD A,@R_r Add Data Memory Contents to Accumulator

| 0110 | 000r |

The contents of the resident data memory location addressed by register 'r' bits 0-5 are added to the accumulator. Carry is affected.

(A)←(A) + ((Rr))   r=0-1

Example:
```
ADDM: MOV R0, #0AFH   ;MOVE 'AF' HEX TO REG 0
      ADD A, @R0      ;ADD VALUE OF LOCATION
                      ;47 TO ACC
```

ADD A,#data Add Immediate Data to Accumulator

| 0000 | 0011 | | $d_7 d_6 d_5 d_4$ | $d_3 d_2 d_1 d_0$ |

This is a 2-cycle instruction. The specified data is added to the accumulator. Carry is affected

(A) ← (A) + data

Example:
```
ADDID: ADD A,#ADDER:  ;ADD VALUE OF SYMBOL
                      ;'ADDER' TO ACC
```

ADDC A,R_r Add Carry and Register Contents to Accumulator

| 0111 | 1rrr |

The content of the carry bit is added to accumulator location 0 and the carry bit cleared. The contents of register 'r' are then added to the accumulator. Carry is affected.

(A) ← (A)+(Rr)+(C)   r=0-7

Example:
```
ADDRGC: ADDC A,R4     ;ADD CARRY AND REG 4
                      ;CONTENTS TO ACC
```

ADDC A,@R_r Add Carry and Data Memory Contents to Accumulator

| 0111 | 000r |

The content of the carry bit is added to accumulator location 0 and the carry bit cleared. Then the contents of the resident data memory location addressed by register 'r' bits 0-5 are added to the accumulator. Carry is affected.

(A) ← (A)+((Rr))+(C)   r 0-1

Example:
```
ADDMC: MOV R1,#40     ;MOVE '40' DEC TO REG 1
       ADDC A,@R1     ;ADD CARRY AND LOCATION 40
                      ;CONTENTS TO ACC
```

ADDC A,#data Add Carry and Immediate Data to Accumulator

| 0001 | 0011 | | $d_7 d_6 d_5 d_4$ | $d_3 d_2 d_1 d_0$ |

This is a 2-cycle instruction. The content of the carry bit is added to accumulator location 0 and the carry bit cleared. Then the specified data is added to the accumulator. Carry is affected.

(A) ← (A)+data+(C)

Example:
```
       ADDC A,#225    ;ADD CARRY AND '225' DEC
                      ;TO ACC
```

ANL A,R_r Logical AND Accumulator With Register Mask

| 0101 | 1rrr |

Data in the accumulator is logically ANDed with the mask contained in working register 'r'.

(A) ← (A) AND (Rr)   r=0-7

Example:
```
ANDREG: ANL A,R3      ;'AND' ACC CONTENTS WITH MASK
                      ;IN REG 3
```

ANL A,@R_r Logical AND Accumulator With Memory Mask

| 0101 | 000r |

Data in the accumulator is logically ANDed with the mask contained in the data memory location referenced by register 'r', bits 0-5.

(A) ← (A) AND ((Rr))   r=0-1

Example:
```
ANDDM: MOV R0,#0FFH   ;MOVE 'FF' HEX TO REG 0
       ANL A, @R0     ;'AND' ACC CONTENTS WITH
                      ;MASK IN LOCATION 63
```

ANL A,#data Logical AND Accumulator With Immediate Mask

| 0101 | 0011 | | $d_7 d_6 d_5 d_4$ | $d_3 d_2 d_1 d_0$ |

This is a 2-cycle instruction. Data in the accumulator is logically ANDed with an immediately-specified mask.

(A) ← (A) AND data

Examples:
```
ANDID: ANL A,#OAFH    ;'AND' ACC CONTENTS
                      ;WITH MASK 10101111
       ANL A,#3+X/Y   ;'AND' ACC CONTENTS
                      ;WITH VALUE OF EXP
                      ;'3+X/Y'
```

ANL BUS,#data Logical AND BUS With Immediate Mask

| 1001 | 1000 | | $d_7 d_6 d_5 d_4$ | $d_3 d_2 d_1 d_0$ |

This is a 2-cycle instruction. Data on the BUS port is logically ANDed with an immediately-specified mask. This instruction assumes prior specification of an 'OUTL BUS, A' instruction.

(BUS) ← (BUS) AND data

Example:
```
ANDBUS: ANL BUS, #MASK ;'AND' BUS CONTENTS
                       ;WITH MASK EQUAL VALUE
                       ;OF SYMBOL 'MASK'
```

ANL Pp,#data Logical AND Port 1-2 With Immediate Mask

| 1001 | 10pp | | $d_7 d_6 d_5 d_4$ | $d_3 d_2 d_1 d_0$ |

This is a 2-cycle instruction. Data on port 'p' is logically ANDed with an immediately-specified mask.

(Pp) ← (Pp) AND data   p 1-2

Example:
```
ANDP2: ANL P2,#0F0H   ;'AND' PORT 2 CONTENTS
                      ;WITH MASK 'F0' HEX
                      ;(CLEAR P20-23)
```

ANLD Pp,A Logical AND Port 4-7 With Accumulator Mask

| 1001 | 11pp |

This is a 2-cycle instruction. Data on port 'p' is logically ANDed with the digit mask contained in accumulator bits 0-3.

(Pp) ← (Pp) AND (A0-3) p=4-7

Note: The mapping of port 'p' to opcode bits 0-1 is as follows:

1 0	Port
0 0	4
0 1	5
1 0	6
1 1	7

Example:
```
ANDP4: ANLD P4,A      ;'AND' PORT 4 CONTENTS
                      ;WITH ACC BITS 0-3
```

CALL address Subroutine Call

$a_{10}\ a_9\ a_8\ 1$ | 0 1 0 0 $a_7\ a_6\ a_5\ a_4$ | $a_3\ a_2\ a_1\ a_0$

This is a 2-cycle instruction. The program counter and PSW bits 4-7 are saved in the stack. The stack pointer (PSW bits 0-2) is updated. Program control is then passed to the location specified by 'address'. PC bit 11 is determined by the most recent SEL MB instruction.

Execution continues at the instruction following the CALL upon return from the subroutine.

$((SP)) \leftarrow (PC),\ (PSW_{4-7})$
$(SP) \leftarrow (SP)+1$
$(PC_{8-10}) \leftarrow addr_{8-10}$
$(PC_{0-7}) \leftarrow addr_{0-7}$
$(PC_{11}) \leftarrow (DBF)$

Example: Add three groups of two numbers. Put subtotals in locations 50, 51 and total in location 52.

```
        MOV R0,#50    ;MOVE '50' DEC T0 ADDRESS
                      ;REG 0
BEGADD: MOV A,R1      ;MOVE CONTENTS OF REG 1
                      ;TO ACC
        ADD A,R2      ;ADD REG 2 TO ACC
        CALL SUBTOT   ;CALL SUBROUTINE 'SUBTOT'
        ADD A R3      ;ADD REG 3 TO ACC
        ADD A,R4      ;ADD REG 4 TO ACC
        CALL SUBTOT   ;CALL SUBROUTINE 'SUBTOT'
        ADD A,R5      ;ADD REG 5 TO ACC
        ADD A,R6      ;ADD REG 6 TO ACC
        CALL SUBTOT   ;CALL SUBROUTINE 'SUBTOT'

SUBTOT: MOV @R0,A     ;MOVE CONTENTS OF ACC TO
                      ;LOCATION ADDRESSED BY
                      ;REG 0
        INC R0        ;INCREMENT REG 0
        RET           ;RETURN TO MAIN PROGRAM
```

CLR A Clear Accumulator

0 0 1 0 | 0 1 1 1

The contents of the accumulator are cleared to zero.

$A \leftarrow 0$

CLR C Clear Carry Bit

1 0 0 1 | 0 1 1 1

During normal program execution, the carry bit can be set to one by the ADD, ADDC, RLC, CPL C, RRC, and DAA instructions. This instruction resets the carry bit to zero.

$C \leftarrow 0$

CLR F1 Clear Flag 1

1 0 1 0 | 0 1 0 1

Flag 1 is cleared to zero.

$(F1) \leftarrow 0$

CLR F0 Clear Flag 0

1 0 0 0 | 0 1 0 1

Flag 0 is cleared to zero.

$(F0) \leftarrow 0$

CPL A Complement Accumulator

0 0 1 1 | 0 1 1 1

The contents of the accumulator are complemented. This is strictly a one's complement. Each one is changed to zero and vice-versa.

$(A) \leftarrow NOT\ (A)$

Example: Assume accumulator contains 01101010.

```
CPLA: CPL A          ;ACC CONTENTS ARE COMPLE-
                     ;MENTED TO 10010101
```

CPL C Complement Carry Bit

1 0 1 0 | 0 1 1 1

The setting of the carry bit is complemented; one is changed to zero, and zero is changed to one.

$(C) \leftarrow NOT\ (C)$

Example: Set C to one; current setting is unknown.

```
CTO1: CLR C          ;C IS CLEARED TO ZERO
      CPL C          ;C IS SET TO ONE
```

CPL F0 Complement Flag 0

1 0 0 1 | 0 1 0 1

The setting of flag 0 is complemented; one is changed to zero, and zero is changed to one.

$F0 \leftarrow NOT\ (F0)$

CPL F1 Complement Flag 1

1 0 1 1 | 0 1 0 1

The setting of flag 1 is complemented; one is changed to zero, and zero is changed to one.

$(F1) \leftarrow NOT\ (F1)$

DA A Decimal Adjust Accumulator

0 1 0 1 | 0 1 1 1

The 8-bit accumulator value is adjusted to form two 4-bit Binary Coded Decimal (BCD) digits following the binary addition of BCD numbers. The carry bit C is affected. If the contents of bits 0-3 are greater than nine, or if AC is one, the accumulator is incremented by six.

The four high-order bits are then checked. If bits 4-7 exceed nine, or if C is one, these bits are increased by six. If an overflow occurs, C is set to one; otherwise, it is cleared to zero.

Example: Assume accumulator contains 10011011.

```
DA A                          ;ACC ADJUSTED TO 00000001
                              ;WITH C SET

C  AC   7     4 3     0
0  0    1 0 0 1  1 0 1 1
                 0 1 1 0      ADD SIX TO BITS 0-5
0 0     1 0 1 0  0 0 0 1
        0 1 1 0               ADD SIX TO BITS 4-7
1 0     0 0 0 0  0 0 0 1       OVERFLOW TO C
```

DEC A Decrement Accumulator

0 0 0 0 | 0 1 1 1

The contents of the accumulator are decremented by one.

$(A) \leftarrow (A)-1$

Example: Decrement contents of external data memory location 63.

```
MOV R0,#3FH        ;MOVE '3F' HEX TO REG 0
MOVX A,@R0         ;MOVE CONTENTS OF LOCATION 63
                   ;TO ACC
DEC A              ;DECREMENT ACC
MOVX @R0,A         ;MOVE CONTENTS OF ACC TO
                   ;LOCATION 63 IN EXPANDED
                   ;MEMORY
```

DEC R_r Decrement Register

1 1 0 0 | 1 r r r

The contents of working register 'r' are decremented by one.

$(Rr) \leftarrow (Rr)-1$ $r=0-7$

Example:

```
DECR1: DEC R1              ;DECREMENT CONTENTS OF REG 1
```

DIS I Disable External Interrupt

0 0 0 1 | 0 1 0 1

External interrupts are disabled. A low signal on the interrupt input pin has no effect.

DIS TCNTI Disable Timer/Counter Interrupt

0 0 1 1 | 0 1 0 1

Timer/counter interrupts are disabled. Any pending timer interrupt request is cleared. The interrupt sequence is not initiated by an overflow, but the timer flag is set and time accumulation continues.

DJNZ R_r, address   Decrement Register and Test

[1 1 1 0 | 1 r r r] [a_7 a_6 a_5 a_4 | a_3 a_2 a_1 a_0]

This is a 2-cycle instruction. Register 'r' is decremented and tested for zero. If the register contains all zeros, program control falls through to the next instruction. If the register contents are not zero, control jumps to the specified 'address'.

The address in this case must evaluate to 8-bits, that is, the jump must be to a location within the current 256-location page.

(Rr) ← (Rr)-1   r=0-7
If Rr not 0
(PC_{0-7}) ← addr

Note: A 12-bit address specification does not cause an error if the DJNZ instruction and the jump target are on the same page. If the DJNZ instruction begins in location 255 of a page, it must jump to a target address on the following page.

Example: Increment values in data memory locations 50-54.

```
        MOV R0,#50       ;MOVE '50' DEC TO ADDRESS
                         ;REG 0
        MOV R3,#5        ;MOVE '5' DEC TO COUNTER
                         ;REG 3
INCRT:  INC @R0          ;INCREMENT CONTENTS OF
                         ;LOCATION ADDRESSED BY
                         ;REG 0
        INC R0           ;INCREMENT ADDRESS IN REG 0
        DJNZ R3, INCRT   ;DECREMENT REG 3 — JUMP TO
                         ;'INCRT' IF REG 3 NONZERO
        NEXT —           ;'NEXT' ROUTINE EXECUTED
                         ;IF R3 IS ZERO
```

EN I   Enable External Interrupt

[0 0 0 0 | 0 1 0 1]

External interrupts are enabled. A low signal on the interrupt input pin initiates the interrupt sequence.

EN TCNTI   Enable Timer/Counter Interrupt

[0 0 1 0 | 0 1 0 1]

Timer/counter interrupts are enabled. An overflow of this register initiates the interrupt sequence.

ENT0 CLK   Enable Clock Output

[0 1 1 1 | 0 1 0 1]

The test 0 pin is enabled to act as the clock output. This function is disabled by a system reset.

Example:
```
EMTST0: ENT0 CLK        ;ENABLE T0 AS CLOCK OUTPUT
```

IN A,Pp   Input Port or Data to Accumulator

[0 0 0 0 | 1 0 p p]

This is a 2-cycle instruction. Data present on port 'p' is transferred (read) to the accumulator.

(A) ← (Pp)   p=1-2

Example:
```
INP12:  IN A,P1         ;INPUT PORT 1 CONTENTS
                        ;TO ACC
        MOV R6,A        ;MOVE ACC CONTENTS TO
                        ;REG 6
        IN A,P2         ;INPUT PORT 2 CONTENTS
                        ;TO ACC
        MOV R7,A        ;MOVE ACC CONTENTS TO REG 7
```

INC A   Increment Accumulator

[0 0 0 1 | 0 1 1 1]

The contents of the accumulator are incremented by one.

(A) ← (A)+1

Example: Increment contents of location 100 in external data memory.

```
INCA:   MOV R0,#100     ;MOVE '100' DEC TO ADDRESS
                        ;REG 0
        MOVX A,@R0      ;MOVE CONTENTS OF LOCATION
                        ;100 TO ACC
        INC A           ;INCREMENT A
        MOVX @R0,A      ;MOVE ACC CONTENTS TO
                        ;LOCATION 100
```

INC R_r Increment Register

[0 0 0 1 | 1 r r r]

The contents of working register 'r' are incremented by one.

(Rr) ← (Rr)+1   r=0-7

Example:
```
INCR0: INC R0           ;INCREMENT ADDRESS REG 0
```

INC @R_r Increment Data Memory Location

[0 0 0 1 | 0 0 0 r]

The contents of the resident data memory location addressed by register 'r' bits 0-5 are incremented by one.

((Rr)) ← ((Rr))+1   r=0-1

Example:
```
INCDM:  MOV R1,#OFFH    ;MOVE ONES TO REG 1
        INC @R1         ;INCREMENT LOCATION 63
```

INS A,BUS   Strobed Input of BUS Data to Accumulator

[0 0 0 0 | 1 0 0 0]

This is a 2-cycle instruction. Data present on the BUS port is transferred (read) to the accumulator when the RD pulse is dropped. (Refer to section on programming memory expansion for details).

(A) ← (BUS)

Example:
```
INPBUS: INS A,BUS       ;INPUT BUS CONTENTS
                        ;TO ACC
```

JBb address   Jump If Accumulator Bit is Set

[b_2 b_1 b_0 1 | 0 0 1 0] [a_7 a_6 a_5 a_4 | a_3 a_2 a_1 a_0]

This is a 2-cycle instruction. Control passes to the specified address if accumulator bit 'b' is set to one.

(PC_{0-7}) ← addr   If Bb=1
(PC) = (PC)+2   If Bb=0

Example:
```
JB4IS1: JB4 NEXT        ;JUMP TO 'NEXT' ROUTINE
                        ;IF ACC BIT 4=1
```

JC address   Jump If Carry Is Set

[1 1 1 1 | 0 1 1 0] [a_7 a_6 a_5 a_4 | a_3 a_2 a_1 a_0]

This is a 2-cycle instruction. Control passes to the specified address if the carry bit is set to one.

(PC_{0-7}) ← addr   If C=1
(PC) = (PC)+2   If C=0

Example:
```
JC1: JC OVFLOW          ;JUMP TO 'OVFLOW' ROUTINE
                        ;IF C=1
```

JF0 address   Jump If Flag 0 Is Set

[1 0 1 1 | 0 1 1 0] [a_7 a_6 a_5 a_4 | a_3 a_2 a_1 a_0]

This is a 2-cycle instruction. Control passes to the specified address if flag 0 is set to one.

(PC_{0-7}) ← addr   If F0=1
(PC) = (PC)+2   If F0=0

Example:
```
JF0IS1: JF0 TOTAL       ;JUMP TO 'TOTAL' ROUTINE
                        ;IF F0=1
```

JF1 address   Jump If Flag 1 Is Set

[0 1 1 1 | 0 1 1 0] [a_7 a_6 a_5 a_4 | a_3 a_2 a_1 a_0]

This is a 2-cycle instruction. Control passes to the specified address if flag 1 is set to one.

(PC_{0-7}) ← addr   If F1=1
(PC) = (PC)+2   IF F1=0

Example:
```
JF1IS1: JF1 FILBUF      ;JUMP TO 'FILBUF'
                        ;ROUTINE IF F1=1
```

JMP address   Direct Jump Within 2K Block

$\boxed{a_{10}\,a_9\,a_8\,0 \mid 0\,1\,0\,0}$ $\boxed{a_7\,a_6\,a_5\,a_4 \mid a_3\,a_2\,a_1\,a_0}$

This is a 2-cycle instruction. Bits 0-10 of the program counter are replaced with the directly-specified address. The setting of PC bit 11 is determined by the most recent SELECT MB instruction.

$(PC_{8-10}) \leftarrow$ addr 8-10
$(PC_{0-7}) \leftarrow$ addr 0-7
$(PC_{11}) \leftarrow$ (DBF)

Example:
```
JMP SUBTOT          ;JUMP TO SUBROUTINE 'SUBTOT'
JMP $-6             ;JUMP TO INSTRUCTION SIX LOCATIONS
                    ;BEFORE CURRENT LOCATION
JMP 2FH             ;JUMP TO ADDRESS '2F' HEX
```

JMPP @A   Indirect Jump Within Page

$\boxed{1\,0\,1\,1 \mid 0\,0\,1\,1}$

This is a 2-cycle instruction. The contents of the program memory location pointed to by the accumulator are substituted for the 'page' portion of the program counter (PC bits 0-7).

$(PC_{0-7}) \leftarrow ((A))$

Example: Assume accumulator contains OFH.
```
JMPPAG: JMPP @A     ;JUMP TO ADDRESS STORED IN
                    ;LOCATION 15 IN CURRENT PAGE
```

JNC address   Jump If Carry Is Not Set

$\boxed{1\,1\,1\,0 \mid 0\,1\,1\,0}$ $\boxed{a_7\,a_6\,a_5\,a_4 \mid a_3\,a_2\,a_1\,a_0}$

This is a 2-cycle instruction. Control passes to the specified address if the carry bit is not set, that is, equals zero.

$(PC_{0-7}) \leftarrow$ addr   If C=0
$(PC) = (PC)+2$ IF C=1

Example:
```
JC0: JNC NOVFLO     ;JUMP TO 'NOVFLO' ROUTINE
                    ;If C=0
```

JNI address   Jump If Interrupt Input is Low

$\boxed{1\,0\,0\,0 \mid 0\,1\,1\,0}$ $\boxed{a_7\,a_6\,a_5\,a_4 \mid a_3\,a_2\,a_1\,a_0}$

This is a 2-cycle instruction. Control passes to the specified address if the interrupt input signal is low (=0), that is, an external interrupt has been signaled. (This signal initiates an interrupt service sequence if the external interrupt is enabled.)

$(PC_{0-7}) \leftarrow$ addr   If I=0
$(PC) = (PC)+2$ If I=1

Example:
```
LOC 3: JNI EXTINT   ;JUMP TO 'EXTINT' ROUTINE
                    ;If I=0
```

JNT0 address   Jump If Test 0 Is Low

$\boxed{0\,0\,1\,0 \mid 0\,1\,1\,0}$ $\boxed{a_7\,a_6\,a_5\,a_4 \mid a_3\,a_2\,a_1\,a_0}$

This is a 2-cycle instruction. Control passes to the specified address, if the test 0 signal is low

$(PC_{0-7}) \leftarrow$ addr   If T0=0
$(PC) = (PC)+2$ If T0=1

Example:
```
JTOLOW: JNTO 60     ;JUMP TO LOCATION 60 DEC
                    ;IF T0=0
```

JNT1 address   Jump If Test 1 Is Low

$\boxed{0\,1\,0\,0 \mid 0\,1\,1\,0}$ $\boxed{a_7\,a_6\,a_5\,a_4 \mid a_3\,a_2\,a_1\,a_0}$

This is a 2-cycle instruction. Control passes to the specified address, if the test 1 signal is low.

$(PC_{0-7}) \leftarrow$ addr   If T1=0
$(PC) = (PC)+2$ If T1=1

JNZ address   Jump If Accumulator Is Not Zero

$\boxed{1\,0\,0\,1 \mid 0\,1\,1\,0}$ $\boxed{a_7\,a_6\,a_5\,a_4 \mid a_3\,a_2\,a_1\,a_0}$

This is a 2-cycle instruction. Control pases to the specified address if the accumulator contents are nonzero at the time this instruction is executed.

$(PC_{0-7}) \leftarrow$ addr   If A≠0
$(PC) = (PC)+2$ If A=0

Example:
```
JACCN0: JNZ 0ABH    ;JUMP TO LOCATION 'AB' HEX
                    ;IF ACC VALUE IS NONZERO
```

JTF address   Jump If Timer Flag Is Set

$\boxed{0\,0\,0\,1 \mid 0\,1\,1\,0}$ $\boxed{a_7\,a_6\,a_5\,a_4 \mid a_3\,a_2\,a_1\,a_0}$

This is a 2-cycle instruction. Control passes to the specified address if the timer flag is set to one, that is, the timer/counter register has overflowed. Testing the timer flag resets it to zero. (This overflow initiates an interrupt service sequence if the timer-overflow interrupt is enabled.)

$(PC_{0-7}) \leftarrow$ addr   If TF=1
$(PC) = (PC)+2$ If TF=0

Example:
```
JTF1: JTF TIMER     ;JUMP TO 'TIMER' ROUTINE
                    ;IF TF=1
```

JT0 address   Jump If Test 0 Is High

$\boxed{0\,0\,1\,1 \mid 0\,1\,1\,0}$ $\boxed{a_7\,a_6\,a_5\,a_4 \mid a_3\,a_2\,a_1\,a_0}$

This is a 2-cycle instruction. Control passes to the specified address if the test 0 signal is high (=1)

$(PC_{0-7}) \leftarrow$ addr   If T0=1
$(PC) = (PC)+2$ If T0=0

Example:
```
JT0HI: JT0 53       ;JUMP TO LOCATION 53 DEC
                    ;IF T0=1
```

JT1 address   Jump If Test 1 Is High

$\boxed{0\,1\,0\,1 \mid 0\,1\,1\,0}$ $\boxed{a_7\,a_6\,a_5\,a_4 \mid a_3\,a_2\,a_1\,a_0}$

This is a 2-cycle instruction. Control passes to the specified address if the test 1 signal is high (=1)

$(PC_{0-7}) \leftarrow$ addr   If T1=1
$(PC) = (PC)+2$ If T1=0

Example:
```
JT1HI: JT1 COUNT    ;JUMP TO 'COUNT' ROUTINE
                    ;IF T1=1
```

JZ address   Jump If Accumulator Is Zero

$\boxed{1\,1\,0\,0 \mid 0\,1\,1\,0}$ $\boxed{a_7\,a_6\,a_5\,a_4 \mid a_3\,a_2\,a_1\,a_0}$

This is a 2-cycle instruction. Control passes to the specified address if the accumulator contains all zeros at the time this instruction is executed.

$(PC_{0-7}) \leftarrow$ addr   If A=0
$(PC) = (PC)+2$ If A≠0

Example:
```
JACCO: JZ OA3H      ;JUMP TO LOCATION 'A3' HEX
                    ;IF ACC VALUE IS ZERO
```

MOV A, #data   Move Immediate Data to Accumulator

$\boxed{0\,0\,1\,0 \mid 0\,0\,1\,1}$ $\boxed{d_7\,d_6\,d_5\,d_4 \mid d_3\,d_2\,d_1\,d_0}$

This is a 2-cycle instruction. The 8-bit value specified by 'data' is loaded in the accumulator.

$(A) \leftarrow$ data

Example:
```
MOV A,#0A3H         ;MOVE 'A3' HEX TO ACC
```

MOV A,PSW   Move PSW Contents to Accumulator

$\boxed{1\,1\,0\,0 \mid 0\,1\,1\,1}$

The contents of the program status word are moved to the accumulator.

$(A) \leftarrow$ (PSW)

Example: Jump to 'RB1SET' routine if PSW bank switch, bit 4, is set.
```
BSCHK: MOV A,PSW    ;MOVE PSW CONTENTS TO ACC
       JB4 RB1SET   ;JUMP TO 'RB1SET' IF ACC
                    ;BIT 4 1
```

MOV A,R_r Move Register Contents to Accumulator

$\boxed{1\,1\,1\,1 \mid 1\,r\,r\,r}$

8-bits of data are moved from working register r into the accumulator.

$(A) \leftarrow (R_r)$ r 0-7

Example:
```
MAR: MOV A,R3       ;MOVE CONTENTS OF REG 3
                    ;TO ACC
```

MOV A,@R$_r$ Move Data Memory Contents to Accumulator

1111	000r

The contents of the resident data memory location addressed by bits 0-5 of register 'r' are moved to the accumulator. Register 'r' contents are unaffected.

(A) ← ((Rr)) r=0-1

Example: Assume R1 contains 01110110.

```
MADM: MOV A,@R1      ;MOVE CONTENTS OF DATA MEM
                     ;LOCATION 54 TO ACC
```

MOV A,T Move Timer/Counter Contents to Accumulator

0100	0010

The contents of the timer/event-counter register are moved to the accumulator.

(A) ← (T)

Example: Jump to "EXIT" routine when timer reaches '64', that is, when bit 6 set — assuming initialization 64,

```
TIMCHK: MOV A,T      ;MOVE TIMER CONTENTS TO
                     ;ACC
        JB6 EXIT     ;JUMP TO 'EXIT' IF ACC BIT
                     ;6=1
```

MOV PSW,A Move Accumulator Contents to PSW

1101	0111

The contents of the accumulator are moved into the program status word. All condition bits and the stack pointer are affected by this move.

(PSW) ← (A)

Example: Move up stack pointer by two memory locations, that is, increment the pointer by one.

```
INCPTR: MOV A,PSW    ;MOVE PSW CONTENTS TO ACC
        INC A        ;INCREMENT ACC BY ONE
        MOV PSW,A    ;MOVE ACC CONTENTS TO PSW
```

MOV R$_r$,A Move Accumulator Contents to Register

1010	1rrr

The contents of the accumulator are moved to register 'r'.

(Rr) ← (A) r=0-7

Example:

```
MRA: MOV R0,A        ;MOVE CONTENTS OF ACC TO
                     ;REG 0
```

MOV R$_r$,#data Move Immediate Data to Register

1011	1 r_2 r_1 r_0	d_7 d_6 d_5 d_4	d_3 d_2 d_1 d_0

This is a 2-cycle instruction. The 8-bit value specified by 'data' is moved to register 'r'.

(Rr) ← data r=0-7

Examples:

```
MIR4: MOV R4,#HEXTEN ;THE VALUE OF THE SYMBOL
                     ;'HEXTEN' IS MOVED INTO
                     ;REG 4
MIR 5: MOV R5,#PI*(R*R) ;THE VALUE OF THE
                     ;EXPRESSION 'PI*(R*R)
                     ;IS MOVED INTO REG 5
MIR 6: MOV R6, #0ADH ;'AD' HEX IS MOVED INTO
                     ;REG 6
```

MOV @R$_r$,A Move Accumulator Contents to Data Memory

1010	000r

The contents of the accumulator are moved to the resident data memory location whose address is specified by bits 0-5 of register 'r'. Register 'r' contents are unaffected.

((Rr)) ← (A) r=0-1

Example: Assume R0 contains 11000111.

```
MDMA: MOV @R0,A      ;MOVE CONTENTS OF ACC TO
                     ;LOCATION 7 (REG 7)
```

MOV @R$_r$,#data Move Immediate Data to Data Memory

1011	000r	d_7 d_6 d_5 d_4	d_3 d_2 d_1 d_0

This is a 2-cycle instruction. The 8-bit value specified by 'data' is moved to the resident data memory location addressed by register 'r', bits 0-5.

((Rr)) ← data r=0-1

Examples: Move the hexadecimal value AC3F to locations 62-63.

```
MIDM: MOV R0,#62     ;MOVE '62' DEC TO ADDR REG 0
      MOV @R0,#OACH  ;MOVE 'AC' HEX TO LOCATION 62
      INC R0         ;INCREMENT REG 0 TO '63'
      MOV @R0,#3FH   ;MOVE '3F' HEX TO LOCATION 63
```

MOV T,A Move Accumulator Contents to Timer/Counter

0110	0010

The contents of the accumulator are moved to the timer/event-counter register.

(T) ← (A)

Example: Initialize and start event counter.

```
INITEC: CLR A        ;CLEAR ACC TO ZEROS
        MOV T,A      ;MOVE ZEROS TO EVENT COUNTER
        STRT CNT     ;START COUNTER
```

MOVD A,Pp Move Port 4-7 Data to Accumulator

0000	11pp

This is a 2-cycle instruction. Data on 8243 port 'p' is moved (read) to accumulator bits 0-3. Accumulator bits 4-7 are zeroed.

($A_{0\text{-}3}$) ← (Pp) p=4-7
($A_{4\text{-}7}$) ← 0

Note: Bits 0-1 of the opcode are used to represent ports 4-7. If you are coding in binary rather than assembly language, the mapping is as follows:

Bits 1 0	Port
0 0	4
0 1	5
1 0	6
1 1	7

Example:

```
INPPT5: MOVD A,P5    ;MOVE PORT 5 DATA TO ACC
                     ;BITS 0-3, ZERO ACC BITS 4-7
```

MOVD Pp,A Move Accumulator Data to Port 4-7

0011	11pp

Data in accumulator bits 0-3 is moved (written) to 8243 port 'p'. Accumulator bits 4-7 are unaffected. (See NOTE above regarding port mapping.)

(Pp) ← ($A_{0\text{-}3}$) p=4-7

Example: Move data in accumulator to ports 4 and 5.

```
OUTP45: MOVD P4,A    ;MOVE ACC BITS 0-3 TO PORT 4
        SWAP A       ;EXCHANGE ACC BITS 0-3 AND 4-7
        MOVD P5,A    ;MOVE ACC BITS 0-3 TO PORT 5
```

MOVP A,@A Move Current Page Data to Accumulator

1010	0011

The contents of the program memory location addressed by the accumulator are moved to the accumulator. Only bits 0-7 of the program counter are affected, limiting the program memory reference to the current page. The program counter is restored following this operation

($PC_{0\text{-}7}$) ← (A)
(A) ← ((PC))

Note: This is a 1-byte, 2-cycle instruction. If it appears in location 255 of a program memory page, @A addresses a location in the following page.

Example:

```
MOV128: MOV A,#128   ;MOVE '128' DEC TO ACC
        MOVP A,@A    ;CONTENTS OF 129th LOCATION
                     ;IN CURRENT PAGE ARE MOVED TO
                     ;ACC
```

MOVP3 A,@A Move Page 3 Data to Accumulator

```
| 1110 | 0011 |
```

This is a 2-cycle instruction. The contents of the program memory location (within page 3) addressed by the accumulator are moved to the accumulator. The program counter is restored following this operation.

$(PC_{0-7}) \leftarrow (A)$
$(PC_{8-10}) \leftarrow 011$
$(A) \leftarrow ((PC))$

Example: Look up ASCII equivalent of hexadecimal code in table contained at the beginning of page 3. Note that ASCII characters are designated by a 7-bit code; the eighth bit is always reset.

```
TABSCH: MOV A,#0B8H   ;MOVE 'B8' HEX TO ACC (10111000)
        ANL A,#7FH    ;LOGICAL AND ACC TO MASK BIT
                      ;7 (00111000)
        MOVP3 A,@A    ;MOVE CONTENTS OF LOCATION
                      ;'38' HEX IN PAGE 3 TO ACC
                      ;(ASCII '8')
```

Access contents of location in page 3 labelled TAB1. Assume current program location is not in page 3.

```
TABSCH: MOV A,#LOW TAB1 ;ISOLATE BITS 0-7 OF LABEL
                        ;ADDRESS VALUE
        MOVP3 A,@A      ;MOVE CONTENTS OF PAGE 3
                        ;LOCATION LABELED 'TAB1'
                        ;TO ACC
```

MOVX A,@R$_r$ Move External-Data-Memory Contents to Accumulator

```
| 1000 | 000r |
```

This is a 2-cycle instruction. The contents of the external data memory location addressed by register 'r' are moved to the accumulator. Register 'r' contents are unaffected.

$(A) \leftarrow ((Rr))$ r=0-1

Example: Assume P1 contains 01110110.

```
MAXDM: MOVX A,@R1       ;MOVE CONTENTS OF LOCATION
                        ;118 TO ACC
```

MOVX @R$_r$,A Move Accumulator Contents to External Data Memory

```
| 1001 | 000r |
```

This is a 2-cycle instruction. The contents of the accumulator are moved to the external data memory location addressed by register 'r'. Register 'r' contents are unaffected.

$((Rr)) \leftarrow A$

Example: Assume R0 contains 11000111.

```
MXDMA: MOVX @R0,A       ;MOVE CONTENTS OF ACC TO
                        ;LOCATION 199 IN EXPANDED
                        ;DATA MEMORY
```

NOP The NOP Instruction

```
| 0000 | 0000 |
```

No operation is performed. Execution continues with the following instruction.

ORL A,R$_r$ Logical OR Accumulator With Register Mask

```
| 0100 | 1rrr |
```

Data in the accumulator is logically ORed with the mask contained in working register 'r'.

$(A) \leftarrow (A)$ OR (Rr) r=0-7

Example:

```
ORREG: ORL A,R4         ;'OR' ACC CONTENTS WITH
                        ;MASK IN REG 4
```

ORL A,@R$_r$ Logical OR Accumulator With Memory Mask

```
| 0100 | 000r |
```

Data in the accumulator is logically ORed with the mask contained in the resident data memory location referenced by register 'r', bits 0-5.

$(A) \leftarrow (A)$ OR $((Rr))$ r=0-1

Example:

```
ORDM: MOV R0,#3FH       ;MOVE '3F' HEX TO REG 0
      ORL A,@R0         ;'OR' ACC CONTENTS WITH MASK
                        ;IN LOCATION 63
```

ORL A,#data Logical OR Accumulator With Immediate Mask

```
| 0100 | 0011 |   | d7 d6 d5 d4 | d3 d2 d1 d0 |
```

This is a 2-cycle instruction. Data in the accumulator is logically ORed with an immediately-specified mask.

$(A) \leftarrow (A)$ OR data

Example:

```
ORID: ORL A,#'X'        ;'OR' ACC CONTENTS WITH MASK
                        ;01011000 (ASCII VALUE OF 'X'
```

ORL BUS,#data Logical OR BUS With Immediate Mask

```
| 1000 | 1000 |   | d7 d6 d5 d4 | d3 d2 d1 d0 |
```

This is a 2-cycle instruction. Data on the BUS port is logically ORed with an immediately-specified mask. This instruction assumes prior specification of an 'OUTL BUS,A' instruction.

$(BUS) \leftarrow (BUS)$ OR data

Example:

```
ORBUS: ORL BUS,#HEXMSK  ;'OR' BUS CONTENTS WITH
                        ;MASK EQUAL VALUE OF SYMBOL
                        ;'HEXMSK'
```

ORL Pp, #data Logical OR Port 1 or 2 With Immediate Mask

```
| 1000 | 10pp |   | d7 d6 d5 d4 | d3 d2 d1 d0 |
```

This is a 2-cycle instruction. Data on port 'p' is logically ORed with an immediately-specified mask.

$(Pp) \leftarrow (Pp)$ OR data p=1-2

Example:

```
ORP1: ORL P1, #0FFH     ;'OR' PORT 1 CONTENTS WITH
                        ;MASK 'FF' HEX ( SET PORT 1
                        ;TO ALL ONES)
```

ORLD Pp,A Logical OR Port 4-7 With Accumulator Mask

```
| 1000 | 11pp |
```

Data on port 'p' is logically ORed with the digit mask contained in accumulator bits 0-3.

$(Pp) \leftarrow (Pp)$ OR (A_{0-3}) p=4-7

Example:

```
ORP7: ORLD P7,A         ;'OR' PORT 7 CONTENTS
                        ;WITH ACC BITS 0-3
```

OUTL BUS,A Output Accumulator Data to BUS

```
| 0000 | 0010 |
```

Data residing in the accumulator is transferred (written) to the BUS port and latched. The latched data remains valid until altered by another OUTL instruction. Any other instruction requiring use of the BUS port (except INS) destroys the contents of the BUS latch. This includes expanded memory operations (such as the MOVX instruction). Logical operations on BUS data (AND, OR) assume the OUTL BUS,A instruction has been issued previously.

$(BUS) \leftarrow (A)$

Example:

```
OUTLBP: OUTL BUS,A      ;OUTPUT ACC CONTENTS TO BUS
```

OUTL Pp,A Output Accumulator Data to Port 1 or 2

```
| 0011 | 10pp |
```

Data residing in the accumulator is transferred (written) to port 'p' and latched.

$(Pp) \leftarrow (A)$ p=1-2

Example:

```
OUTLP: MOV A,R7         ;MOVE REG 7 CONTENTS TO ACC
       OUTL P2,A        ;OUTPUT ACC CONTENTS TO PORT 2
       MOV A,R6         ;MOVE REG 6 CONTENTS TO ACC
       OUTL P1,A        ;OUTPUT ACC CONTENTS TO PORT 1
```

RET Return Without PSW Restore

```
| 1000 | 0011 |
```

This is a 2-cycle instruction. The stack pointer (PSW bits 0-2) is decremented. The program counter is then restored from the stack. PSW bits 4-7 are not restored.

$(SP) \leftarrow (SP)-1$
$(PC) \leftarrow ((SP))$

RETR Return With PSW Restore

| 1001 | 0011 |

This is a 2-cycle instruction. The stack pointer is decremented. The program counter and bits 4-7 of the PSW are then restored from the stack. Note that RETR should be used to return from an interrupt, but should not be used within the interrupt service routine as it signals the end of an interrupt routine.

(SP) ← (SP)-1
(PC) ← ((SP))
(PSW 4-7) ← ((SP))

RL A Rotate Left Without Carry

| 1110 | 0111 |

The contents of the accumulator are rotated left one bit. Bit 7 is rotated into the bit 0 position.

(An+1) ← (An)
(A0) ← (A7)   n=0-6

Example: Assume accumulator contains 10110001

```
RLNC: RL A          ;NEW ACC CONTENTS ARE 01100011
```

RLC A Rotate Left Through Carry

| 1111 | 0111 |

The contents of the accumulator are rotated left one bit. Bit 7 replaces the carry bit; the carry bit is rotated into the bit 0 position.

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

Example: Assume accumulator contains a 'signed' number; isolate sign without changing value.

```
RLTC: CLR C                     ;CLEAR CARRY TO ZERO
      RLC A                     ;ROTATE ACC LEFT, SIGN
                                ;BIT (7) IS PLACED IN CARRY
      RR A                      ;ROTATE ACC RIGHT — VALUE
                                (BITS 0-6) IS RESTORED,
                                ;CARRY UNCHANGED, BIT 7
                                ;IS ZERO
```

RR A Rotate Right Without Carry

| 0111 | 0111 |

The contents of the accumulator are rotated right one bit. Bit 0 is rotated into the bit 7 position

(An) ← (An+1)   n=0-6
(A7) ← (A0)

Example: Assume accumulator contains 10110001.

```
RRNC: RR A          ;NEW ACC CONTENTS ARE 11011000
```

RRC A Rotate Right Through Carry

| 0110 | 0111 |

The contents of the accumulator are rotated right one bit. Bit 0 replaces the carry bit; the carry bit is rotated into the bit 7 position.

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

Example: Assume carry is not set and accumulator contains 10110001.

```
RRTC: RRC A                     ;CARRY IS SET AND ACC
                                ;CONTAINS 01011000
```

SEL MBO Select Memory Bank 0

| 1110 | 0101 |

PC bit 11 is set to zero on next branch instruction. All references to program memory addresses fall within the range 0-2047.

(DBF) ← 0

Example: Assume program counter contains 834 Hex and the carry bit is set.

```
      SEL MBO                   ;SELECT MEMORY BANK 0
      JC $+20                   ;IF C=1, JUMP TO LOCATION
                                ;48 HEX
```

SEL MB1 Select Memory Bank 1

| 1111 | 0101 |

PC bit 11 is set to one on next branch instruction. All references to program memory addresses fall within the range 2048-4095.

(DBF) ← 1

SEL RB0 Select Register Bank 0

| 1100 | 0101 |

PSW bit 4 is set to zero. References to working registers 0-7 address data memory locations 0-7. This is the recommended setting for normal program execution.

(BS) ← 0

SEL RB1 Select Register Bank 1

| 1101 | 0101 |

PSW bit 4 is set to one. References to working registers 0-7 address data memory locations 24-31. This is the recommended setting for interrupt service routines, since locations 0-7 are left intact. The setting of PSW bit 4 in effect at the time of an interrupt is restored by the RETR instruction when the interrupt service routine is completed.

(BS) ← 1

Example: Assume an external interrupt has occurred, control has passed to program memory location 3, and PSW bit 4 was zero before the interrupt.

```
LOC3: JNI INIT                  ;JUMP TO ROUTINE 'INIT' IF
                                ;INTERRUPT INPUT IS ZERO

 INIT: MOV R7,A                 ;MOVE ACC CONTENTS TO
                                ;LOCATION 7
       SEL RB1                  ;SELECT REG BANK 1
       MOV R7,#0FAH             ;MOVE 'FA' HEX TO LOCATION 31
       .
       .
       .
       SEL RB0                  ;SELECT REG BANK 0
       MOV A,R7                 ;RESTORE ACC FROM LOCATION 7
       RETR                     ;RETURN — RESTORE PC AND PSW
```

STOP TCNT Stop Timer/Event-Counter

| 0110 | 0101 |

This instruction is used to stop both time accumulation and event counting.

Example: Disable interrupt, but jump to interrupt routine after eight overflows and stop timer. Count overflows in register 7.

```
START:  DIS TCNTI               ;DISABLE TIMER INTERRUPT
        CLR A                   ;CLEAR ACC TO ZEROS
        MOV T,A                 ;MOVE ZEROS TO TIMER
        MOV R7,A                ;MOVE ZEROS TO REG 7
        STRT T                  ;START TIMER
MAIN:   JTF COUNT               ;JUMP TO ROUTINE 'COUNT'
                                ;IF TF=1 AND CLEAR TIMER FLAG
        JMP MAIN                ;CLOSE LOOP
COUNT:  INC R7                  ;INCREMENT REG 7
        MOV A,R7                ;MOVE REG 7 CONTENTS TO ACC
        JB3 INT                 ;JUMP TO ROUTINE 'INT' IF ACC
                                ;BIT 3 IS SET (REG 7=8)
        JMP MAIN                ;OTHERWISE RETURN TO ROUTINE
                                ;MAIN
        .
        .
        .
INT:    STOP TCNT               ;STOP TIMER
         JMP 7H                 ;JUMP TO LOCATION 7 (TIMER)
                                ;INTERRUPT ROUTINE
```

STRT CNT Start Event Counter

0100	0101

The test 1 (T1) pin is enabled as the event-counter input and the counter is started. The event-counter register is incremented with each high-to-low transition on the T1 pin.

Example: Initialize and start event counter. Assume overflow is desired with first T1 input.

```
STARTC: EN TCNTI      ;ENABLE COUNTER INTERRUPT
        MOV A,#0FFH   ;MOVE 'FF' HEX (ONES) TO
                      ;ACC
        MOV T,A       ;MOVE ONES TO COUNTER
        STRT CNT      ;ENABLE TIAS COUNTER
                      ;INPUT AND START
```

STRT T Start Timer

0101	0101

Timer accumulation is initiated in the timer register. The register is incremented every 32 instruction cycles. The prescaler which counts the 32 cycles is cleared but the timer register is not.

Example: Initialize and start timer.

```
STARTT: CLR A         ;CLEAR ACC TO ZEROS
        MOV T,A       ;MOVE ZEROS TO TIMER
        EN TCNTI      ;ENABLE TIMER INTERRUPT
        STRT T        ;START TIMER
```

SWAP A Swap Nibbles Within Accumulator

0100	0111

Bits 0-3 of the accumulator are swapped with bits 4-7 of the accumulator.

$(A_{4-7}) \leftrightarrows (A_{0-3})$

Example: Pack bits 0-3 of locations 50-51 into location 50.

```
PCKDIG: MOV R0, #50   ;MOVE '50' DEC TO REG 0
        MOV R1, #51   ;MOVE '51' DEC TO REG 1
        XCHD A,@R0    ;EXCHANGE BITS 0-3 OF ACC
                      ;AND LOCATION 50
        SWAP A        ;SWAP BITS 0-3 AND 4-7 OF ACC
        XCHD A,@R1    ;EXCHANGE BITS 0-3 OF ACC AND
                      ;LOCATION 51
        MOV @R0,A     ;MOVE CONTENTS OF ACC TO
                      ;LOCATION 50
```

XCH A,R$_r$ Exchange Accumulator-Register Contents

0010	1rrr

The contents of the accumulator and the contents of working register 'r' are exchanged.

$(A) \leftrightarrows (Rr)$ r=0-7

Example: Move PSW contents to Reg 7 without losing accumulator contents.

```
XCHAR7: XCH A,R7      ;EXCHANGE CONTENTS OF REG 7
                      ;AND ACC
        MOV A, PSW    ;MOVE PSW CONTENTS TO ACC
        XCH A,R7      ;EXCHANGE CONTENTS OF REG 7
                      ;AND ACC AGAIN
```

XCH A,@R$_r$ Exchange Accumulator and Data Memory Contents

0010	000r

The contents of the accumulator and the contents of the resident data memory location addressed by bits 0-5 of register 'r' are exchanged. Register 'r' contents are unaffected.

$(A) \leftrightarrows -((Rr))$ r=0-1

Example: Decrement contents of location 52.

```
DEC52: MOV R0,#52     ;MOVE '52' DEC TO ADDRESS
                      ;REG 0
       XCH A,@R0      ;EXCHANGE CONTENTS OF ACC
                      ;AND LOCATION 52
       DEC A          ;DECREMENT ACC CONTENTS
       XCH A,@R0      ;EXCHANGE CONTENTS OF ACC
                      ;AND LOCATION 52 AGAIN
```

XCHD A,@R$_r$ Exchange Accumulator and Data Memory 4-Bit Data

0011	000r

This instruction exchanges bits 0-3 of the accumulator with bits 0-3 of the data memory location addressed by bits 0-5 of register 'r'. Bits 4-7 of the accumulator, bits 4-7 of the data memory location, and the contents of register 'r' are unaffected.

$(A_{0-3}) \leftrightarrows ((Rr0-3))$ r=0-1

Example: Assume program counter contents have been stacked in locations 22-23.

```
XCHNIB: MOV R0,#23    ;MOVE '23' DEC TO REG 0
        CLR A         ;CLEAR ACC TO ZEROS
        XCHD A,@R0    ;EXCHANGE BITS 0-3 OF ACC
                      ;AND LOCATION 23 (BITS 8-11
                      ;OF PC ARE ZEROED. ADDRESS
                      ;REFERS TO PAGE 0)
```

XRL A,R$_r$ Logical XOR Accumulator With Register Mask

1101	1rrr

Data in the accumulator in EXCLUSIVE ORed with the mask contained in working register 'r'.

$(A) \leftarrow (A)$ XOR (Rr) r=0-7

Example:

```
XORREG: XRL A,R5      ;'XOR' ACC CONTENTS WITH
                      ;MASK IN REG 5
```

XRL A,@R$_r$ Logical XOR Accumulator With Memory Mask

1101	000r

Data in the accumulator is EXCLUSIVE ORed with the mask contained in the data memory location addressed by register 'r', bits 0-5.

$(A) \leftarrow (A)$ XOR $((Rr))$ r=0-1

Example:

```
XORDM: MOV R1, #20H   ;MOVE '20' HEX TO REG 1
       XRL A,@R1      ;'XOR' ACC CONTENTS WITH MASK
                      ;IN LOCATION 32
```

XRL A,#data Logical XOR Accumulator With Immediate Mask

1101	0011	$d_7 d_6 d_5 d_4$	$d_3 d_2 d_1 d_0$

This is a 2-cycle instruction. Data in the accumulator is EXCLUSIVE ORed with an immediately-specified mask.

$(A) \leftarrow (A)$ XOR data

Example:

```
XORID: XOR A,#HEXTEN;XOR CONTENTS OF ACC WITH
                    ;MASK EQUAL VALUE OF SYMBOL
                    ;'HEXTEN'
```

INSTRUCTION SET SUMMARY

	Mnemonic	Description	Bytes	Cycle
Accumulator	ADD A, R	Add register to A	1	1
	ADD A, @R	Add data memory to A	1	1
	ADD A, #data	Add immediate to A	2	2
	ADDC A, R	Add register with carry	1	1
	ADDC A, @R	Add data memory with carry	1	1
	ADDC A, #data	Add immediate with carry	2	2
	ANL A, R	And register to A	1	1
	ANL A, @R	And data memory to A	1	1
	ANL A, #data	And immediate to A	2	2
	ORL A, R	Or register to A	1	1
	ORL A, @R	Or data memory to A	1	1
	ORL A, #data	Or immediate to A	2	2
	XRL A, R	Exclusive Or register to A	1	1
	XRL A, @R	Exclusive or data memory to A	1	1
	XRL A, #data	Exclusive or immediate to A	2	2
	INC A	Increment A	1	1
	DEC A	Decrement A	1	1
	CLR A	Clear A	1	1
	CPL A	Complement A	1	1
	DA A	Decimal Adjust A	1	1
	SWAP A	Swap nibbles of A	1	1
	RL A	Rotate A left	1	1
	RLC A	Rotate A left through carry	1	1
	RR A	Rotate A right	1	1
	RRC A	Rotate A right through carry	1	1
Input/Output	IN A, P	Input port to A	1	2
	OUTL P, A	Output A to port	1	2
	ANL P, #data	And immediate to port	2	2
	ORL P, #data	Or immediate to port	2	2
	INS A, BUS	Input BUS to A	1	2
	OUTL BUS, A	Output A to BUS	1	2
	ANL BUS, #data	And immediate to BUS	2	2
	ORL BUS, #data	Or immediate to BUS	2	2
	MOVD A, P	Input Expander port to A	1	2
	MOVD P, A	Output A to Expander port	1	2
	ANLD P, A	And A to Expander port	1	2
	ORLD P, A	Or A to Expander port	1	2
Registers	INC R	Increment register	1	1
	INC @R	Increment data memory	1	1
	DEC R	Decrement register	1	1
Branch	JMP addr	Jump unconditional	2	2
	JMPP @A	Jump indirect	1	2
	DJNZ R, addr	Decrement register and test	2	2
	JC addr	Jump on Carry = 1	2	2
	JNC addr	Jump on Carry = 0	2	2
	J Z addr	Jump on A Zero	2	2
	JNZ addr	Jump on A not Zero	2	2
	JT0 addr	Jump on T0 = 1	2	2
	JNT0 addr	Jump on T0 = 0	2	2
	JT1 addr	Jump on T1 = 1	2	2
	JNT1 addr	Jump on T1 = 0	2	2
	JF0 addr	Jump on F0 = 1	2	2
	JF1 addr	Jump on F1 = 1	2	2
	JTF addr	Jump on timer flag	2	2
	JNI addr	Jump on $\overline{\text{INT}}$ = 0	2	2
	JBb addr	Jump on Accumulator Bit	2	2

	Mnemonic	Description	Bytes	Cycles
Subroutine	CALL	Jump to subroutine	2	2
	RET	Return	1	2
	RETR	Return and restore status	1	2
Flags	CLR C	Clear Carry	1	1
	CPL C	Complement Carry	1	1
	CLR F0	Clear Flag 0	1	1
	CPL F0	Complement Flag 0	1	1
	CLR F1	Clear Flag 1	1	1
	CPL F1	Complement Flag 1	1	1
Data Moves	MOV A, R	Move register to A	1	1
	MOV A, @R	Move data memory to A	1	1
	MOV A, #data	Move immediate to A	2	2
	MOV R, A	Move A to register	1	1
	MOV @R, A	Move A to data memory	1	1
	MOV R, #data	Move immediate to register	2	2
	MOV @R, #data	Move immediate to data memory	2	2
	MOV A, PSW	Move PSW to A	1	1
	MOV PSW, A	Move A to PSW	1	1
	XCH A, R	Exchange A and register	1	1
	XCH A,@R	Exchange A and data memory	1	1
	XCHD A, @R	Exchange nibble of A and register	1	1
	MOVX A, @R	Move external data memory to A	1	2
	MOVX @R, A	Move A to external data memory	1	2
	MOVP A, @A	Move to A from current page	1	2
	MOVP3 A, @A	Move to A from Page 3	1	2
Timer/Counter	MOV A, T	Read Timer/Counter	1	1
	MOV T, A	Load Timer/Counter	1	1
	STRT T	Start Timer	1	1
	STRT CNT	Start Counter	1	1
	STOP TCNT	Stop Timer/Counter	1	1
	EN TCNTI	Enable Timer/Counter Interrupt	1	1
	DIS TCNTI	Disable Timer/Counter Interrupt	1	1
Control	EN I	Enable external interrupt	1	1
	DIS I	Disable external interrupt	1	1
	SEL RB0	Select register bank 0	1	1
	SEL RB1	Select register bank 1	1	1
	SEL MB0	Select memory bank 0	1	1
	SEL MB1	Select memory bank 1	1	1
	ENT0 CLK	Enable Clock output on T0	1	1
	NOP	No Operation	1	1

3. ASSEMBLER DIRECTIVES

This chapter describes the assembler directives used to control the 8048 assembler in its generation of object code. These directives are written in the same format as 8048 instructions, in general, and can be interspersed throughout your assembly language program.

Unlike assembly language instructions, however, they produce no executable object code.

Assembler directives can be divided functionally as follows;

- Location counter control
  - ORG
- Symbol definition
  - EQU
  - SET
- Data definition
  - DB
  - DW
- Memory reservation
  - DS
- Assembler termination
  - END

One notable format difference between assembler directives and 8048 instructions involves the 'lable' field. This field is always optional in 8048 instructions. The same is generally true of assembler directives, but two directives (EQU, SET) require the name of the symbol to be present in the label field.

LOCATION COUNTER CONTROL

The location counter performs the same function for the assembler as the program counter performs during program execution. It tells the assembler the next memory location available for instruction or data assembly.

The location counter can be set by the ORG (origin) directive. See also the discussion of the END directive in the section 'Assembler Termination,' later in this chapter.

ORG Directive

Label	Opcode	Operand
optional	ORG	expression.

The location counter is set to the value of 'expression', which must evaluate to a valid 12-bit program memory address. If 'expression' is a symbol, the symbol must be previously defined. The next machine instruction or data item is assembled at the address specified. If no ORG is included before the first instruction or data byte in your program, assembly begins at location zero.

Your program can include any number of ORG statements. Multiple ORGs need not be listed in ascending order, but failure to do so creates potential memory overlap problems.

Example:

```
     PAG1     ORG     OFFH      ;ORG ASSEMBLER TO LOCATION
                                ;'FF'HEX (255 DEC)
```

SYMBOL DEFINITION

Symbol names appearing as labels in 8048 instructions are assigned values automatically during the assembly process. The value in this case is the value in the location counter when the labeled instruction is assembled.

Other symbols are defined and assigned arbitrary values using the EQU and SET directives. Symbols defined using EQU cannot be redefined during assembly; those defined by SET can be assigned new values by subsequent SET directives.

EQU Directive

<u>Label</u>	<u>Opcode</u>	<u>Operand</u>
name	EQU	expression

The symbol 'name' is created and assigned the value of 'expression'. This 'name' cannot appear in the label field of another EQU or SET directive, that is, it is not redefinable.

Example:

```
     ONES   EQU   OFFH      ;CREATE SYMBOL 'ONES' WITH
                            ;BINARY VALUE 11111111
```

SET Directive

<u>Label</u>	<u>Opcode</u>	<u>Operand</u>
name	SET	expression

The symbol 'name' is assigned the value of 'expression'. Wherever the symbol name is encountered in the assembly, this value is used unless 'name' is assigned a new value by another SET directive.

DATA DEFINITION

The DB (define byte) and DW (define word) directives enable you to specify data in your program. Data can be specified in the form of 8-bit or 16-bit values, or as a text string.

DB Directive

Label	Opcode	Operands
optional	DB	expression(s) or string (s)

The operand field of the DB directive can contain a list of expressions and/or text strings with the list items separated by commas. The list can contain up to eight total elements, but elements containing expressions can reduce this maximum allowance.

Expressions must evaluate to 1-byte (8-bit) numbers. This provides a range of -256 to +255 (all ones or all zeroes in the high-order byte of the internal representation). Strings can extend over an arbitrary number of bytes. The bytes assembled for the DB directive are stored consecutively in available memory starting at the address in the location counter.

Note: Do not mix expressions and strings in one DB or DW directive, this will cause errors.

Example:

```
DATA    DB    'STRING 1', 'STRING 2'
QUTE    DB    'THIS IS A QUOTE'''
NUM     DB    4, 5, 6
```

DW Directive

Label	Opcode	Operands
optional	DW	expression(s) or string constant(s)

The operand field of the DW directive can contain a list of expressions and/or 1-byte or 2-byte string constants. List items are separated by commas. The list can contain up to eight total elements, but elements containing expressions can reduce this maximum allowance.

Expressions must evaluate to 1-word (16-bit) numbers. The high-order eight bits of the 16-bit value are assembled at the address in the location counter; the low order eight bits are assembled at the next higher location.

Strings are limited to one or two characters. In the case of a single character string, the high-order eight bits are filled with zeros.

Examples:

```
     ADDR      DW       FIRST, LAST
     PAGE      DW       0,0100H,0200H,0300H
     STRS      DW       'AB', 'CD'
```

MEMORY RESERVATION

A block of program memory can be reserved using the DS (define storage) directive. No data is assembled into these locations and no assumptions can be made about their initial contents when your program is loaded.

DS Directive

Label	Opcode	Operand
optional	DS	expression

'Expression' specifies the number of locations to be reserved for data storage. This block of memory locations is reserved by incrementing the location counter by the value of 'expression'. This value must be absolute. Any symbol appearing in the operand field must be previously defined.

If the optional label is present, it is assigned the starting value of the location counter (before incrementing), and thus references the starting address of the rserved block.

If the value of 'expression' is zero, no memory is reserved, but the label is assigned the current value of the location counter.

Example:

```
     TTYB      DS      72      ;RESERVE 72 LOCATIONS AS A
                               ;TERMINAL OUTPUT BUFFER.
```

ASSEMBLER TERMINATION

The END directive terminates assembler execution.

END Directive

Label	Opcode	Operand
optional	END	expression

The END directive identifies the end of the source program and terminates each pass of the assembler. Only one END directive can appear in your program and it must be the last source line of the program.

If 'expression' is specified in the operand field, its value is used as the program execution starting address. If no 'expression' is given, the starting address is zero.

Example:

```
    END     STRT      ;EXECUTION BEGINS AT THE
                      ;ADDRESS LABELED 'START'
```

NOTE: After the END directive assembly line should be inserted to let the assembler print the END-directive-line. This is a known anomally in this assembler.

PRINTER CONTROL

EJECT Directive

Label	Opcode	Operand
-----	EJE	-------

The "EJECT" directive ejects a page of the listing, the remainder of the listing to be continued at the top of nextpage.

SPACE Directive

Label	Opcode	Operand
-----	SPC	expression

The "SPACE" directive causes a number of blank lines to be printed before the listing continues.
"Expression" must result in an absolute value less or equal to 57 but larger than zero.