Instructions of Assembly Language Programming

Instructor

Zhizheng Wu

吴智政

School of Mechatronic Engineering and Automation

OUTLINE

- Structure of assembly language
- Instruction set of assembly language

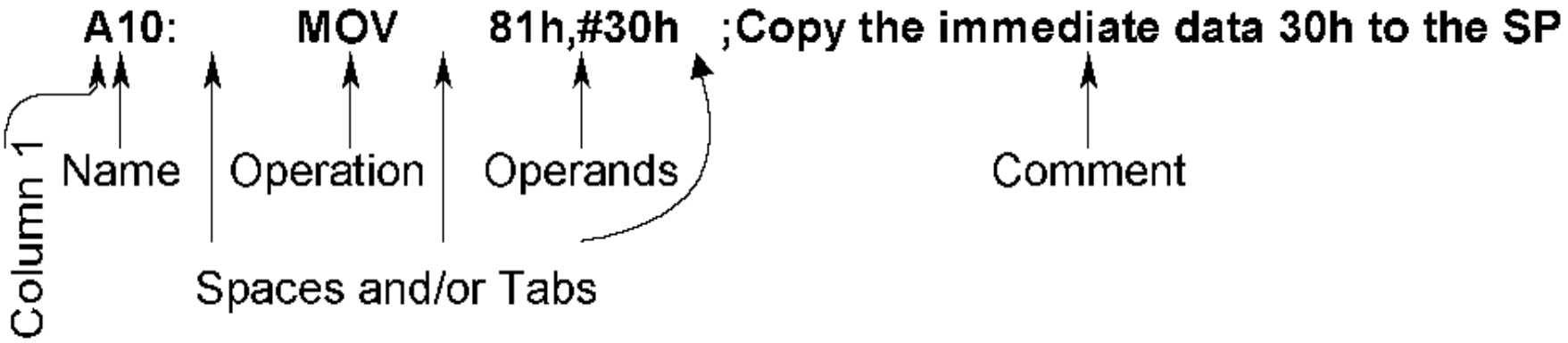
STRUCTURE OF ASSEMBLY LANGUAGE

☞ Structure of assembly language

- In the early days of the computer, programmers coded in machine language, consisting of 0s and 1s
  - Tedious, slow and prone to error
- Assembly languages, which provided mnemonics for the machine code instructions, plus other features, were developed
  - An Assembly language program consist of a series of lines of Assembly language instructions
  - Assembly language is referred to as a low level language. It deals directly with the internal structure of the CPU
- High level language, i.e. C++, C, Java, …

STRUCTURE OF ASSEMBLY LANGUAGE

- Assembly Language Syntax
  - Syntax = format/rule
  - **[label:] mnemonic [operands] [;comment]**
  - Items in square brackets are optional



STRUCTURE OF ASSEMBLY LANGUAGE

- Mnemonic/Operation Code (Opcode)
  - Program = a set of instructions
  - All computers work according to the program
  - All instructions contain a “verb” called mnemonic/operation code, which tells the computer what to do
  - e.g. MOV R0, #12h — “**MOV**” is the opcode

STRUCTURE OF ASSEMBLY LANGUAGE

- Operand

  - Apart from Opcode, an instruction also includes object to act on.
  - The object is called an operand.
  - Operand is optional. Instructions can have one, two or no operands.
  - e.g. MOV **R0, #12h** --- R0 and #12h are two operands

    INC **A** --- A is the only one operand

    NOP --- no operand follows

STRUCTURE OF ASSEMBLY LANGUAGE

- Binary nature of machine instruction
  - Unlike human, computers do not know verbal instructions; they only know **0**s and **1**s
  - Binary data: program should be in a stream of 0s and 1s
  - It is also called ***"Machine Language"***
  - For convenient purpose, machine instructions are usually expressed in hexadecimal (i.e. base-16) format, called machine codes.

    e.g. Mnemonic : ADD A, #10H

    Equivalent Machine codes: 24h 10h (hexadecimal)

- How 8051 Interprets Binary Data
  - Machine instructions can be 3 bytes (24 bits), 2 bytes (16 bits) or 1 byte (8 bits) long
  - The 1st byte (8 bits) is the operation code (opcode)
  - The remaining byte(s) is/are the supplement data for the operation code
  - 1-byte instruction: Contain the opcode only. Actions do not need supplement data.

e.g.

Mnemonic	Equivalent Machine codes	
NOP	00h	(hexadecimal)
ADD A, R0	28h	
INC A	04h	

- How 8051 Interprets Binary Data
  - 2-byte instruction: The 1st byte is the opcode. The 2nd byte may be either an immediate data (a number) or the low-order byte of an address

    e.g.

Mnemonic	Equivalent Machine codes
ADD A, #30h	24h 30h
ADD A, 30h	25h 30h

  - 3-byte instruction: The 1st byte is the opcode. The 2nd and the 3rd byte are the low-order byte and the high-order byte of an 16-bit memory address

    e.g.

Mnemonic	Equivalent Machine codes
LJMP #0130h	02h 30h 01h

STRUCTURE OF ASSEMBLY LANGUAGE

- Pseudo-instructions/Directives

Beside mnemonics, directives are used to define variables and memory locations where the machine codes are stored. These directives are interpreted by assembler during the conversion of the assembly language program into machine codes.

- ORG (origin)

Indicates the beginning of the address of the instructions. The number that comes after ORG can be either hex or decimal.

Eg. ORG 0030H

- END

Indicates to the assembler the end of the source assembly instructions.

- Pseudo-instructions/Directives

- EQU (equate)

Used to define a constant without occupying a memory location. It does not set aside storage for a data item but associates a constant value with a data label so that when the label appears in the program. Its constant value will be substituted for the label.

```
COUNT  EQU   25H
```

- DB (define byte)

Used to define 8-bit data and store them in assigned memory locations. Define data can be in decimal, binary, hex, or ASCII formats.

```
MYDATA      DB    23H
ASCIICODE:  DB    "APPLE"
```

- Pseudo-instructions/Directives

- EQU (equate)

Used to define a constant without occupying a memory location. It does not set aside storage for a data item but associates a constant value with a data label so that when the label appears in the program. Its constant value will be substituted for the label.

```
COUNT  EQU   25H
```

- DB (define byte)

Used to define 8-bit data and store them in assigned memory locations. Define data can be in decimal, binary, hex, or ASCII formats.

```
MYDATA      DB    23H
ASCIICODE:  DB    "APPLE"
```

- Pseudo-instructions/Directives
  - **BIT** Directive

Used to define bit addressable I/O and RAM locations.

Example:

Sensor1: BIT P3.1 ;Define Sensor1 to be the status of pin P3.1

Sensor2: BIT 4CH ;Define Sensor2 to be the contents of RAM bit with address 4CH

STRUCTURE OF ASSEMBLY LANGUAGE

Content of the List file of an assembly language program

```
LOC  OBJ       LINE        SOURCE
0025           1     COUNT EQU  25H          ;COUNT = 25H
0000           2           ORG    0H         ;start (origin) at location 0
0000  AD25     3           MOV    R5, COUNT  ;load 25H into R5
0002  7F34     4           MOV    R7, #34H   ;load 34H into R7
0004  7400     5           MOV    A, #0      ;load 0 into A
0006  2D       6           ADD    A, R5      ;add contents of R5 to A, now A = A + R5
0007  2F       7           ADD    A, R7      ;add contents of R7 to A, now A = A + R7
0008  2412     8           ADD    A, #12H    ;add to A value 12H, now A = A + 12H
000A  00       9           NOP               ;no operation
000B  80FE     10    HERE: SJMP HERE         ;stay in this loop
0020           11          ORG    20H
0020  39       12    DATA1: DB  39H          ;
0021  416D6572 13    DATA2: DB "America“     ;
0025  696361
               14          END               ;end of assembly source file
```

Directives

Opcode

Comments

Label

Operands

Source program in Assembly Language (Mnemonics)

Machine codes stored in memory

- **ASCII Codes**

➢ ASCII stands for American Standard Code for Information Interchange. Computers can only understand numbers, so an ASCII code is the numerical representation of a character.

➢ ASCII uses 7 bits to represent a character. As a result only 127 characters are defined as standard ASCII characters. Characters 128-255 are called extended ASCII characters

Dec	Hx	Oct	Char		Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL	(null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH	(start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX	(start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX	(end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT	(end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ	(enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK	(acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL	(bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS	(backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB	(horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF	(NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT	(vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF	(NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR	(carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO	(shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI	(shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE	(data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1	(device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2	(device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3	(device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4	(device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK	(negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN	(synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB	(end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN	(cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM	(end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB	(substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC	(escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS	(file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS	(group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS	(record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US	(unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.asciitable.com

ASCII table and description

128	Ç	144	É	160	á	176	░	193	┴	209	╤	225	ß	241	±
129	ü	145	æ	161	í	177	▒	194	┬	210	╥	226	Γ	242	≥
130	é	146	Æ	162	ó	178	▓	195	├	211	╙	227	π	243	≤
131	â	147	ô	163	ú	179	│	196	─	212	╘	228	Σ	244	⌠
132	ä	148	ö	164	ñ	180	┤	197	┼	213	╒	229	σ	245	⌡
133	à	149	ò	165	Ñ	181	╡	198	╞	214	╓	230	µ	246	÷
134	å	150	û	166	ª	182	╢	199	╟	215	╫	231	τ	247	≈
135	ç	151	ù	167	º	183	╖	200	╚	216	╪	232	Φ	248	°
136	ê	152	_	168	¿	184	╕	201	╔	217	┘	233	Θ	249	·
137	ë	153	Ö	169	_	185	╣	202	╩	218	┌	234	Ω	250	·
138	è	154	Ü	170	¬	186	║	203	╦	219	█	235	δ	251	√
139	ï	156	£	171	½	187	╗	204	╠	220	▄	236	∞	252	_
140	î	157	¥	172	¼	188	╝	205	═	221	▌	237	φ	253	²
141	ì	158	_	173	¡	189	╜	206	╬	222	▐	238	ε	254	■
142	Ä	159	ƒ	174	«	190	╛	207	╧	223	▀	239	∩	255	
143	Å	192	└	175	»	191	┐	208	╨	224	α	240	≡		

Source: www.asciitable.com

Extended ASCII codes

- **BCD Codes**
  - **Binary-coded decimal** (**BCD**) (sometimes called **natural binary-coded decimal**, **NBCD**) or, in its most common modern implementation, **packed decimal**, is an encoding for decimal numbers in which each digit is represented by its own binary sequence.
  - To encode a decimal number using the common BCD encoding, each decimal digit is stored in a 4-bit nibble.

Decimal:	0	1	2	3	4	5	6	7	8	9
BCD:	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001

Example:

Decimal: 127

BCD: 0001 0010 0111

Binary: 0111 1111

- **BCD Codes**

  - **Virtue:** It allows easy conversion to decimal digits for printing or display, and allows faster decimal calculations

  - **Drawbacks:** A small increase in the complexity of circuits needed to implement mathematical operations. Uncompressed BCD is also a relatively inefficient encoding — it occupies more space than a purely binary representation

- **Unpacked BCD**
  - The lower 4 bits of the number represent the BCD number.
  - The rest of the bits are 0.
  - For example, "0000 1001" and "0000 0101" are unpacked BCD for 9 and 5, respectively.
  - Unpacked BCD requires 1 byte of memory or an 8-bit register to contain it.

- **Packed BCD**
  - A single byte has two BCD numbers in it, one in the lower 4 bits, and one in the upper 4 bits.
  - For example, "0101 1001" is packed BCD for 59H.
  - It takes only 1 byte of memory to store the packed BCD operands.
  - Its more efficient than unpacked BCD.

- There is a problem with adding BCD numbers.
- Adding two BCD numbers must give a BCD result.
- After adding packed BCD numbers, the result is no longer BCD.

```
MOV A, #17BCD
ADD  A, #28BCD          ;A = 3F which is not BCD
                        ;should be 17 + 28 = 45BCD
```

"DA A" is designed to correct the BCD addition problem.

- **DA** instruction

```
MOV A,#47H      ;A=47H first BCD operand
MOV B,#25H      ;B=25H second BCD operand
ADD A,B         ;hex (binary) addition (A=6CH)
DA  A           ;adjust for BCD addition (A=72H)BCD
```

- DA A must be used after the addition of BCD operands.
- Important to note that DA A works only after an ADD instruction, it will not work after the INC instruction.

STRUCTURE OF ASSEMBLY LANGUAGE

Key	ASCII (hex)	Binary	BCD (unpacked)
0	30	011 0000	0000 0000
1	31	011 0001	0000 0001
2	32	011 0010	0000 0010
3	33	011 0011	0000 0011
4	34	011 0100	0000 0100
5	35	011 0101	0000 0101
6	36	011 0110	0000 0110
7	37	011 0111	0000 0111
8	38	011 1000	0000 1000
9	39	011 1001	0000 1001

ASCII Code and BCD code for Digits 0–9

☞ Instruction set of assembly language

- There are 139 1-byte instructions, 92 2-byte instructions and 24 3-byte instructions.
- 8051 instructions are divided among five groups
  - Arithmetic
  - Logic
  - Data transfer
  - Boolean variable
  - Program branching

- Arithmetic Operations

  - With arithmetic instructions, the C8051 CPU has no special knowledge of the data format (e.g. signed/unsigned binary, binary coded decimal, ASCII, etc.)
  - The appropriate status bits in the PSW are set when specific conditions are met, which allows the user software to manage the different data formats (carry, overflow etc…)
  - [@Ri] implies contents of memory location pointed to by R0 or R1
  - Rn refers to registers R0-R7 of the currently selected register bank

INTRUCTION SET OF ASSEMBLY LANGUAGE

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

INTRUCTION SET OF ASSEMBLY LANGUAGE

- *ADD A,<source-byte>* *ADDC A,<source-byte>*
  - ADD adds the data byte specified by the source operand to the accumulator, leaving the result in the accumulator
  - ADDC adds the data byte specified by the source operand, the carry flag and the accumulator contents, leaving the result in the accumulator
  - Operation of both the instructions, ADD and ADDC, can affect the carry flag (CY), auxiliary carry flag (AC) and the overflow flag (OV)
    - CY=1 If there is a carryout from bit 7; cleared otherwise
    - AC=1 If there is a carryout from the lower 4-bit of A i.e. from bit 3; cleared otherwise
    - OV=1 If the signed result cannot be expressed within the number of bits in the destination operand; cleared otherwise

- *SUBB A,<source-byte>*
  - SUBB subtracts the specified data byte and the carry flag together from the accumulator, leaving the result in the accumulator
    - ✓CY=1 If a borrow is needed for bit 7; cleared otherwise

Example 6-7

Analyze the following program:

```
CLR  C          ;CY = 0
MOV  A,#62H     ;A = 62H
SUBB A,#96H     ;62H - 96H = CCH with CY = 1
MOV  R7,A       ;save the result
MOV  A,#27H     ;A=27H
SUBB A,#12H     ;27H - 12H - 1 = 14H
MOV  R6,A       ;save the result
```

Solution:

After the SUBB, A = 62H – 96H = CCH and the carry flag is set high indicating there is a borrow. Since CY = 1, when SUBB is executed the second time A = 27H – 12H – 1 = 14H. Therefore, we have 2762H – 1296H = 14CCH.

- *INC <byte>*
  - Increments the data variable by 1. The instruction is used in register, direct or register direct addressing modes

***Example*:**

INC 6FH

If the internal RAM location 6FH contains 30H, then the instruction increments this value, leaving 31H in location 6FH

***Example*:**

MOV R1, #5E
INC R1
INC @R1

If R1=5E (01011110) and internal RAM location 5FH contains 20H, the instructions will result in R1=5FH and internal RAM location 5FH to increment by one to 21H

- *DEC <byte>*

  - The data variable is decremented by 1
  - The instruction is used in accumulator, register, direct or register direct addressing modes
  - A data of value 00H underflows to FFH after the operation
  - No flags are affected

- *INC DPTR*

  - Increments the 16-bit data pointer by 1
  - DPTR is the only 16-bit register that can be incremented
  - The instruction adds one to the contents of DPTR directly

- *MUL AB*
  - Multiplies A & B and the 16-bit result stored in [B15-8], [A7-0]
  - Multiplies the unsigned 8-bit integers in the accumulator and the B register
  - The **Low** order byte of the 16-bit product will go to the accumulator and the **High** order byte will go to the B register
  - If the product is greater than 255 (FFH), the overflow flag is set; otherwise it is cleared. The carry flag is always cleared.
  - E.g. If ACC=85 (55H) and B=23 (17H), the instruction gives the product 1955 (07A3H), so B is now 07H and the accumulator is A3H. The overflow flag is set and the carry flag is cleared.

- *DIV AB*
  - Divides A by B
  - The integer part of the quotient is stored in A and the remainder goes to the B register
  - E.g. If ACC=90 (5AH) and B=05(05H), the instruction leaves 18 (12H) in ACC and the value 00 (00H) in B, since 90/5 = 18 (quotient) and 00 (remainder)
  - Carry and OV are both cleared
  - *If B contains 00H before the division operation (divide by zero), then the values stored in ACC and B are undefined and an overflow flag is set. The carry flag is cleared.*

- **Logical Operations**

  - Logical instructions perform Boolean operations (AND, OR, XOR, and NOT) on data bytes on a *bit-by-bit* basis

  - Examples:

```
ANL  A,  #02H        ;Mask bit 1
ORL  TCON,  A        ;TCON=TCON OR A
```

INTRUCTION SET OF ASSEMBLY LANGUAGE

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

- *ANL <dest-byte>,<source-byte>*
  - This instruction performs the logical AND operation on the source and destination operands and stores the result in the destination variable
  - No flags are affected

***Example*:**

ANL A, R2

If ACC=D3H (11010011) and R2=75H (01110101), the result of the instruction is ACC=51H (01010001)

  - The following instruction is also useful when there is a need to mask a byte

***Example*:**

ANL P1, #10111001B

- *ORL <dest-byte>,<source-byte>*
  - This instruction performs the logical OR operation on the source and destination operands and stores the result in the destination variable
  - No flags are affected

***Example*:**

ORL A, R2

If ACC=D3H (11010011) and R2=75H (01110101), the result of the instruction is ACC=F7H (11110111)

***Example*:**

ORL P1,#11000010B

This instruction sets bits 7, 6, and 1 of output Port 1

- *XRL <dest-byte>,<source-byte>*
  - This instruction performs the logical XOR (Exclusive OR) operation on the source and destination operands and stores the result in the destination variable
  - No flags are affected

***Example*:**

XRL A, R0

If ACC=C3H (11000011) and R0=AAH (10101010), then the instruction results in ACC=69H (01101001)

***Example*:**

XRL P1, #00110001

This instruction complements bits 5, 4, and 0 of output Port 1

- *CLR A and CPL A*

CLR A

- This instruction clears the accumulator (all bits set to 0)
- No flags are affected
- If ACC=C3H, then the instruction results in ACC=00H

CPL A

- This instruction logically complements each bit of the accumulator (one's complement)
- No flags are affected
- If ACC=C3H (11000011), then the instruction results in ACC=3CH (00111100)

- *RL A*

  - The 8 bits in the accumulator are rotated one bit to the left. Bit 7 is rotated into the bit 0 position.

  - No flags are affected

  - If ACC=C3H (11000011), then the instruction results in ACC=87H (10000111) with the carry unaffected

- *RLC A*

  - The instruction rotates the accumulator contents one bit to the left through the carry flag
  - Bit 7 of the accumulator will move into carry flag and the original value of the carry flag will move into the Bit 0 position
  - No other flags are affected
  - If ACC=C3H (11000011), and the carry flag is 1, the instruction results in ACC=87H (10000111) with the carry flag set

- *RR A*

  - The 8 bits in the accumulator are rotated one bit to the right. Bit 0 is rotated into the bit 7 position.

  - No flags are affected

  - If ACC=C3H (11000011), then the instruction results in ACC=E1H (11100001) with the carry unaffected

- *RRC A*

➢ The instruction rotates the accumulator contents one bit to the right through the carry flag

➢ The original value of carry flag will move into Bit 7 of the accumulator and Bit 0 rotated into carry flag

➢ No other flags are affected

➢ If ACC=C3H (11000011), and the carry flag is 0, the instruction results in ACC=61H (01100001) with the carry flag set

- *SWAP A*

  - This instruction interchanges the low order 4-bit nibbles (A3-0) with the high order 4-bit nibbles (A7-4) of the ACC
  - The operation can also be thought of as a 4-bit rotate instruction
  - No flags are affected
  - If ACC=C3H (11000011), then the instruction leaves ACC=3CH (00111100)

- **Data transfer Operations**

  - Data transfer instructions can be used to transfer data between an internal RAM location and an SFR location without going through the accumulator

  - It is also possible to transfer data between the internal and external RAM by using indirect addressing

  - The upper 128 bytes of data RAM are accessed only by indirect addressing and the SFRs are accessed only by direct addressing

INTRUCTION SET OF ASSEMBLY LANGUAGE

Instruction Set: Data Transfer *) MOV A,ACC is not a valid instruction			
Mnemonic	Description	Byte	Cycle
MOV A,Rn	Move register to accumulator	1	1
MOV A,direct	*) Move direct byte to accumulator	2	1
MOV A,@Ri	Move indirect RAM to accumulator	1	1
MOV A,#data	Move immediate data to accumulator	2	1
MOV Rn,A	Move accumulator to register	1	1
MOV Rn,direct	Move direct byte to register	2	2
MOV Rn,#data	Move immediate data to register	2	1
MOV direct,A	Move accumulator to direct byte	2	1
MOV direct,Rn	Move register to direct byte	2	2
MOV direct,direct	Move direct byte to direct byte	3	2
MOV direct,@Ri	Move indirect RAM to direct byte	2	2
MOV direct,#data	Move immediate data to direct byte	3	2
MOV @Ri,A	Move accumulator to indirect RAM	1	1
MOV @Ri,direct	Move direct byte to indirect RAM	2	2
MOV @Ri, #data	Move immediate data to indirect RAM	2	1
MOV DPTR, #data16	Load data pointer with a 16-bit constant	3	2
MOVC A,@A + DPTR	Move code byte relative to DPTR to accumulator	1	2
MOVC A,@A + PC	Move code byte relative to PC to accumulator	1	2
MOVX A,@Ri	Move external RAM (8-bit addr.) to A	1	2
MOVX A,@DPTR	Move external RAM (16-bit addr.) to A	1	2
MOVX @Ri,A	Move A to external RAM (8-bit addr.)	1	2
MOVX @DPTR,A	Move A to external RAM (16-bit addr.)	1	2
PUSH direct	Push direct byte onto stack	2	2
POP direct	Pop direct byte from stack	2	2
XCH A,Rn	Exchange register with accumulator	1	1
XCH A,direct	Exchange direct byte with accumulator	2	1
XCH A,@Ri	Exchange indirect RAM with accumulator	1	1
XCHD A,@Ri	Exchange low-order nibble indir. RAM with A	1	1

- *MOV <dest-byte>,<source-byte>*
  - This instruction moves the source byte into the destination location
  - The source byte is not affected, neither are any other registers or flags

***Example*:**

```
MOV  R1, #60H      ;R1=60H
MOV  A, @R1        ;A=[60H]
MOV  R2, #61H      ;R2=61H
ADD  A, @R2        ;A=A+[61H]
MOV  R7, A         ;R7=A
```

If internal RAM locations 60H=10H, and 61H=20H, then after the operations of the above instructions R7=A=30H. The data contents of memory locations 60H and 61H remain intact.

- *MOV DPTR, #data 16*

➢ This instruction loads the data pointer with the 16-bit constant and no flags are affected

Example:

MOV DPTR, #1032H

This instruction loads the value 1032H into the data pointer, i.e. DPH=10H and DPL=32H.

- *MOVC A, @A + <base-reg>*

➢ This instruction moves a code byte from program memory into ACC

➢ The effective address of the byte fetched is formed by adding the original 8-bit accumulator contents and the contents of the base register, which is either the data pointer (DPTR) or program counter (PC) 16-bit addition is performed and no flags are affected

➢ The instruction is useful in reading the look-up tables in the program memory

➢ If the PC is used, it is incremented to the address of the following instruction before being added to the ACC

- *MOVX <dest-byte>,<source-byte>*
  - ➢ This instruction transfers data between ACC and a byte of external data memory
  - ➢ There are two forms of this instruction, the only difference between them is whether to use an 8-bit or 16-bit indirect addressing mode to access the external data RAM
  - ☞ The 8-bit form of the MOVX instruction uses the P2 to determine the upper 8 bits of the effective address to be accessed and the contents of R0 or R1 to determine the lower 8 bits of the effective address to be accessed

***Example*:**

```
MOV  P2, #10H      ;Load high byte of address into P2
MOV  R0, #34H      ;Load low byte of address into R0(or R1)
MOVX  A, @R0       ;Load contents of 1034H into ACC
```

- *MOVX <dest-byte>,<source-byte>*

☞ The 16-bit form of the MOVX instruction accesses the memory location pointed to by the contents of the DPTR register

***Example*:**

```
MOV   DPTR, #1034H      ;Load DPTR with 16 bit address
                          to read (1034H).
MOVX  A, @DPTR          ;Load contents of 1034H into
                          ACC.
```

- *XCH A,<byte>*

  - This instruction swaps the contents of ACC with the contents of the indicated data byte

***Example*:**

XCH A, @R0

Suppose R0=2EH, ACC=F3H (11110011) and internal RAM location 2EH=76H (01110110). The result of the above instruction leaves RAM location [2EH]=F3H and ACC=76H.

- *XCHD A, @Ri*
  - This instruction exchanges the low order nibble of ACC (bits 0-3), with that of the internal RAM location pointed to by Ri register
  - The high order nibbles (bits 7-4) of both the registers remain the same
  - No flags are affected

Example:

XCHD A, @R0

If R0=2EH, ACC=76H (01110110) and internal RAM location [2EH]=F3H (11110011), the result of the instruction leaves RAM location [2EH]=F6H (11110110) and ACC=73H (01110011)

- **Boolean Variable Operations**

  - The C8051 processor can perform single bit operations
  - The operations include *set*, *clear*, *and*, *or* and *complement* instructions
  - Also included are bit–level moves or conditional jump instructions
  - All bit accesses use direct addressing
  - Examples:

```
        SETB TR0          ;Start Timer0
POLL:   JNB TR0, POLL     ;Wait until timer overflows
```

INTRUCTION SET OF ASSEMBLY LANGUAGE

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

- *CLR <bit>*

  - This operation clears (reset to 0) the specified bit indicated in the instruction
  - No other flags are affected
  - CLR instruction can operate on the carry flag or any directly addressable bit

Example:

CLR P2.7

If Port 2 has been previously written with DCH (11011100), then the operation leaves the port set to 5CH (01011100)

- *SETB <bit>*

  - This operation sets the specified bit to 1
  - SETB instruction can operate on the carry flag or any directly-addressable bit
  - No other flags are affected

Example:

SETB C

SETB P2.0

The carry flag is cleared and the output Port 2 has the value of 24H (00100100), then the result of the instructions sets the carry flag to 1 and changes the Port 2 value to 25H (00100101)

- *CPL <bit>*

  - This operation complements the bit indicated by the operand
  - No other flags are affected
  - CPL instruction can operate on the carry flag or any directly addressable bit

Example:

CPL P2.1

CPL P2.2

If Port 2 has the value of 53H (01010011) before the start of the instructions, then after the execution of the instructions it leaves the port set to 55H (01010101)

- *ANL C, <source-bit>*
  - This instruction ANDs the bit addressed with the carry bit and stores the result in the carry bit itself
  - If the source bit is a logical 0, then the instruction clears the carry flag; else the carry flag is left in its original value
  - If a slash (/) is used in the source operand bit, it means that the logical complement of the addressed source bit is used, **but the source bit itself is not affected**
  - No other flags are affected

Example:

```
MOV  C, P2.0      ;Load C with input pin state of P2.0
ANL  C, P2.7      ;AND carry flag with bit 7 of P2
MOV  P2.1, C      ;Move C to bit 1 of Port 2
ANL  C, /OV       ;AND with inverse of OV flag
```

If P2.0=1, P2.7=0 and OV=0 initially, then after the above instructions, P2.1=0, CY=0 and the OV remains unchanged, i.e. OV=0

- *ORL C, <source-bit>*
  - This instruction ORs the bit addressed with the carry bit and stores the result in the carry bit itself
  - It sets the carry flag if the source bit is a logical 1; else the carry is left in its original value
  - If a slash (/) is used in the source operand bit, it means that the logical complement of the addressed source bit is used, ***but the source bit itself is not affected***
  - No other flags are affected

Example:

```
MOV  C, P2.0      ;Load C with input pin state of P2.0
ORL  C, P2.7      ;OR carry flag with bit 7 of P2
MOV  P2.1, C      ;Move C to bit 1 of port 2
ORL  C, / OV      ;OR with inverse of OV flag
```

- *MOV <dest-bit>,<source-bit>*
  - The instruction loads the value of source operand bit into the destination operand bit
  - 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:

MOV P2.3, C
MOV C, P3.3
MOV P2.0, C

If P2=C5H (11000101), P3.3=0 and CY=1 initially, then after the above instructions, P2=CCH (11001100) and CY=0

- *JC rel*
  - This instruction branches to the address, indicated by the label, if the carry flag is set, otherwise the program continues to the next instruction
  - No flags are affected

Example:

```
CLR  C
SUBB  A, R0
JC  ARRAY1
MOV  A, #20H
```

The carry flag is cleared initially. After the SUBB instruction, if the value of A is smaller than R0, then the instruction sets the carry flag and causes program execution to branch to ARRAY1 address, otherwise it continues to the MOV instruction.

- *JNC rel*
  - This instruction branches to the address, indicated by the label, if the carry flag is **not** set, otherwise the program continues to the next instruction
  - No flags are affected. **The carry flag is not modified.**

Example:

```
CLR  C
SUBB  A, R0
JNC  ARRAY2
MOV  A, #20H
```

The above sequence of instructions will cause the jump to be taken if the value of A is greater than or equal to R0. Otherwise the program will continue to the MOV instruction.

- *JB <bit>, rel*

➢ This instruction jumps to the address indicated if the destination bit is 1, otherwise the program continues to the next instruction

➢ No flags are affected. **The bit tested is not modified**.

Example:

JB ACC.7, ARRAY1

JB P1.2, ARRAY2

If the accumulator value is 01001010 and Port 1=57H (01010111), then the above instruction sequence will cause the program to branch to the instruction at ARRAY2

- *JNB <bit>, rel*

➢ This instruction jumps to the address indicated if the destination bit is 0, otherwise the program continues to the next instruction

➢ No flags are affected. **The bit tested is not modified**.

Example:

JNB ACC.6, ARRAY1

JNB P1.3, ARRAY2

If the accumulator value is 01001010 and Port 1=57H (01010111), then the above instruction sequence will cause the program to branch to the instruction at ARRAY2

- *JBC <bit>, rel*

➢ If the source bit is 1, this instruction clears it and branches to the address indicated; else it proceeds with the next instruction

➢ **The bit is not cleared if it is already a 0**. No flags are affected.

Example:

JBC P1.3, ARRAY1
JBC P1.2, ARRAY2

If P1=56H (01010110), the above instruction sequence will cause the program to branch to the instruction at ARRAY2, modifying P1 to 52H (01010010)

- **Program Branching Operations**

  - Program branching instructions are used to control the flow of program execution

  - Some instructions provide decision making capabilities before transferring control to other parts of the program (conditional branches).

Instruction Set - Program and Machine Control			
Mnemonic	Description	Byte	Cycle
ACALL addr11	Absolute subroutine call	2	2
LCALL addr16	Long subroutine call	3	2
RET	Return from subroutine	1	2
RETI	Return from interrupt	1	2
AJMP addr11	Absolute jump	2	2
LJMP addr16	Long jump	3	2
SJMP rel	Short jump (relative addr.)	2	2
JMP @A + DPTR	Jump indirect relative to the DPTR	1	2
JZ rel	Jump if accumulator is zero	2	2
JNZ rel	Jump if accumulator is not zero	2	2
JC rel	Jump if carry flag is set	2	2
JNC rel	Jump if carry flag is not set	2	2
JB bit, rel	Jump if direct bit is set	3	2
JNB bit, rel	Jump if direct bit is not set	3	2
JBC bit, rel	Jump if direct bit is set and clear bit	3	2
CJNE A, direct, rel	Compare direct byte to A and jump if not equal	3	2
CJNE A, #data, rel	Compare immediate to A and jump if not equal	3	2
CJNE Rn, #data rel	Compare immed. to reg. and jump if not equal	3	2
CJNE @Ri, #data, rel	Compare immed. to ind. and jump if not equal	3	2
DJNZ Rn, rel	Decrement register and jump if not zero	2	2
DJNZ direct, rel	Decrement direct byte and jump if not zero	3	2
NOP	No operation	1	1

- *ACALL addr11*
  - This instruction **unconditionally** calls a subroutine indicated by the address
  - The operation will cause the PC to increase by 2, then it pushes the 16-bit PC value onto the stack (low order byte first) and increments the stack pointer twice
  - The PC is now loaded with the value *addr11* and the program execution continues from this new location
  - The subroutine called must therefore start within the same 2 kB block of the program memory
  - No flags are affected

Example: **ACALL LOC_SUB**

If SP=07H initially and the label "LOC_SUB" is at program memory location 0567H, after executing the instruction at location 0230H, SP = ☐ internal RAM locations 08H = ☐ 09H = ☐ and PC = ☐

- *LCALL addr16*
  - This instruction **unconditionally** calls a subroutine located at the indicated address
  - The operation will cause the PC to increase by 3, then it pushes the 16-bit PC value onto the stack (low order byte first) and increments the stack pointer twice
  - The PC is then loaded with the value *addr16* and the program execution continues from this new location
  - Since it is a Long call, the subroutine may therefore begin anywhere in the full 64 kB program memory address space
  - No flags are affected

Example: **LCALL LOC_SUB**

If SP=07H initially and the label "LOC_SUB" is at program memory location 0567H, after executing the instruction at location 0230H, SP = ☐ internal RAM locations 08H = ☐ 09H = ☐ and PC = ☐

- *RET*

➢ This instruction returns the program from a subroutine
➢ RET pops the high byte and low byte address of PC from the stack and decrements the SP by 2
➢ The execution of the instruction will result in the program to resume from the location just after the "call" instruction
➢ No flags are affected

Suppose SP=0BH originally and internal RAM locations 0AH and 0BH contain the values 30H and 02H respectively. The instruction leaves SP = ____ and program execution will continue at location ____

- *RETI*

  - This instruction returns the program from an interrupt subroutine
  - RETI pops the high byte and low byte address of PC from the stack
  - After the RETI, program execution will resume immediately after the point at which the interrupt is detected

- *AJMP addr11*

➢ The AJMP instruction transfers program execution to the destination address which is located at the absolute short range distance (short range means 11-bit address)

➢The destination must therefore be within the same 2 kB block of program memory

***Example*:**

AJMP NEAR

If the label NEAR is at program memory location 0120H, the AJMP instruction at location 0234H loads the PC with 0120H

- *LJMP addr16*
  - The LJMP instruction transfers program execution to the destination address which is located at the absolute long range distance (long range means 16-bit address)
  - The destination may therefore be anywhere in the full 64 kB program memory address space
  - No flags are affected

***Example*:**

LJMP FAR_ADR

If the label FAR_ADR is at program memory location 3456H, the LJMP instruction at location 0120H loads the PC with 3456H

- *SJMP rel*
  - This is a short jump instruction, which increments the PC by 2 and then adds the relative value '*rel*' (signed 8-bit) to the PC
  - This will be the new address where the program would branch to unconditionally
  - Therefore, the range of destination allowed is from -128 to +127 bytes from the instruction

***Example*:**

SJMP RELSRT

If the label RELSRT is at program memory location 0120H and the SJMP instruction is located at address 0100H, after executing the instruction, PC=0120H.

- *JMP @A + DPTR*
  - This instruction adds the 8-bit unsigned value of the ACC to the 16-bit data pointer and the resulting sum is returned to the PC
  - Neither ACC nor DPTR is altered
  - No flags are affected

***Example*:**

```
                MOV  DPTR, #LOOK_TBL
                JMP  @A + DPTR
LOOK_TBL:   AJMP  LOC0
                AJMP  LOC1
                AJMP  LOC2
```

If the ACC=02H, execution jumps to

Hint: AJMP is a two byte instruction

- *JZ rel*
  - This instruction branches to the destination address if ACC=0; else the program continues to the next instruction
  - The ACC is not modified and no flags are affected

***Example*:**

```
SUBB  A, #20H
JZ  LABEL1
DEC  A
```

If ACC originally holds 20H and CY=0, then the SUBB instruction changes ACC to 00H and causes the program execution to continue at the instruction identified by LABEL1; otherwise the program continues to the DEC instruction

- *JNZ rel*
  - This instruction branches to the destination address if any bit of ACC is a 1; else the program continues to the next instruction
  - The ACC is not modified and no flags are affected

***Example*:**

```
DEC  A
JNZ  LABEL2
MOV  RO, A
```

If ACC originally holds 00H, then the instructions change ACC to FFH and cause the program execution to continue at the instruction identified by LABEL2; otherwise the program continues to MOV instruction

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

➢ This instruction compares the magnitude of the *dest-byte* and the *source-byte* and branches if their values are not equal

➢ The carry flag is set if the unsigned *dest-byte* is less than the unsigned integer *source-byte*; otherwise, the carry flag is cleared

➢ Neither operand is affected

***Example*:**

```
        CJNE  R3, #50H, NEQU
              … …                ;R3 = 50H
NEQU:   JC  LOC1                 ;If R3 < 50H
              … …                ;R3 > 50H
LOC1:         … …                ;R3 < 50H
```

- *DJNZ <byte>,<rel-addr>*
  - This instruction is ”decrement jump not zero”
  - It decrements the contents of the destination location and if the resulting value is not 0, branches to the address indicated by the source operand
  - An original value of 00H underflows to FFH
  - No flags are affected

***Example*:**

DJNZ 20H, LOC1
DJNZ 30H, LOC2

The first instruction will not branch to LOC1 because the [20H] = 00H, hence the program continues to the second instruction. Only after the execution of the second instruction (where the location [30H] = 5FH), then the branching takes place

- *NOP*
  - This is the no operation instruction
  - The instruction takes one machine cycle operation time
  - Hence it is useful to time the ON/OFF bit of an output port

***Example*:**

```
CLR  P1.2
NOP
NOP
NOP
NOP
SETB  P1.2
```

The above sequence of instructions outputs a low-going output pulse on bit 2 of Port 1 lasting exactly cycles

•Note a simple SETB/CLR generates a 1 cycle pulse, so four additional cycles must be inserted in order to have a 5-clock pulse width

- Example:

  Write a program to copy a block of 10 bytes from RAM location starting at 37h to RAM location starting at 59h.

Solution:

```
        MOV   R0,#37h        ; source pointer
        MOV   R1,#59h        ; destination pointer
        MOV   R2,#10         ; counter
L1:     MOV   A,@R0
        MOV   @R1,A
        INC   R0
        INC   R1
        DJNZ  R2, L1
```

- Example: Performing the Addition

	65536's	256's	1's
.		**R6**	**R7**
+		**R4**	**R5**
=	R1	R2	R3

1.Add the low bytes R7 and R5, leave the answer in R3.

2.Add the high bytes R6 and R4, adding any carry from step 1, and leave the answer in R2.

3.Put any carry from step 2 in the final byte, R1.

INTRUCTION SET OF ASSEMBLY LANGUAGE

```
MOV  A, R7 ;Move the low-byte into the accumulator

ADD  A, R5 ;Add the second low-byte to the accumulator

MOV  R3, A ;Move the answer to the low-byte of the result
```

```
MOV  A, R6 ;Move the high-byte into the accumulator

ADDC A, R4 ;Add the second high-byte to the accumulator, plus carry

MOV  R2, A ;Move the answer to the high-byte of the result
```

```
MOV  A, #00h ;By default, the highest byte will be zero

ADDC  A, #00h ;Add zero, plus carry from step 2

MOV  R1, A ;Move the answer to the highest byte of the result
```

Do the calculation using subroutine

```
ORG 0000H

MOV  R6, #1Ah
MOV  R7, #44h          ;Load the first value into R6 and R7

MOV  R4, #22h
MOV  R5, #0DBh         ;Load the first value into R4 and R5
LCALL  ADD16_16        ;Call the 16-bit addition routine

ADD16_16:

MOV  A, R7             ;Move the low-byte into the accumulator
ADD  A, R5             ;Add the second low-byte to the accumulator
MOV  R3, A             ;Move the answer to the low-byte of the result

MOV  A, R6             ;Move the high-byte into the accumulator
ADDC  A, R4            ;Add the second high-byte to the accumulator, plus carry.
MOV  R2, A             ;Move the answer to the high-byte of the result


MOV  A, #00h           ;By default, the highest byte will be zero.
ADDC  A, #00h          ;Add zero, plus carry from step 2.
MOV  R1, A             ;Move the answer to the highest byte of the result

RET                    ;Return - answer now resides in R1, R2, and R3. RET
```