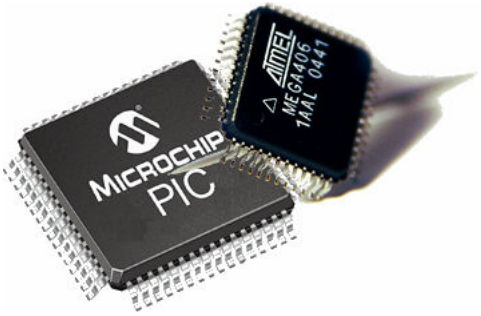
ASSEMBLY LANGUAGE PROGRAMMING



ASSEMBLY LANGUAGE PROGRAMMING

Number Representation for Different Bases

The following is an example showing the decimal number 46 represented in different number bases:

```
46d       ; 46 decimal
2Eh       ; 2Eh is 46 decimal represented as a hex number
56o       ; 56o is 46 decimal represented as an octal number
101110b   ; 101110b is 46 decimal represented as a binary number.
```

Note a number digit must be used in the first character of a hexadecimal number. For example the hexadecimal number A5h is illegally represented and should be represented as 0A5h.

The Arithmetic Operators

The arithmetic operators are:

+	add
-	subtract
*	multiply
/	divide
MOD	modulo (result is the remainder following division)

The Logical Operators

The logical operators are:

AND	Logical AND
OR	Logical OR
XOR	Logical XOR (exclusive OR)
NOT	Logical NOT

The Relational Operators

The result of a relational operation is either true (represented by *minus 1*), or false (represented by zero). The relational operators are:

Equal to	EQ	=
not equal to	NE	<>
greater than	GT	>
greater than or equal to	GE	>=
less than	LT	<

less than or equal to LE <=

(note 'EQ' symbol and '= ' symbol have the same meaning)

Operator Precedence
Like a high level language, assembly level programs define operator predence. Operators with same precedence are evaluated left to right. Note, brackets ( ) means to evaluate this first. HIGH indicates the high-byte and LOW indicates the low-byte. Later examples will clarify the use of such special operators. The precedence list, highest first, is as follows:

( )
HIGH LOW
* / MOD SHL SHR
+ -
= <> < <= > >=
NOT
AND
OR XOR

Some Assembler Directives
The assembler directives are special instruction to the assembler program to define some specific operations but these directives are not part of the executable program. Some of the most frequently assembler directives are listed as follows:

ORG OriGinate, defines the starting address for the program in program (code) memory

EQU EQUate, assigns a numeric value to a symbol identifier so as to make the program more readable.

DB Define a Byte, puts a byte (8-bit number) number constant at this memory location

DW Define a Word, puts a word (16-bit number) number constant at this memory location

DBIT Define a Bit, defines a bit constant, which is stored in the bit addressable section if the Internal RAM.

END This is the last statement in the source file to advise the assembler to stop the assembly process.

Types of Instructions

The assembly level instructions include: data transfer instructions, arithmetic instructions, logical instructions, program control instructions, and some special instructions such as the rotate instructions.

Data Transfer
Many computer operations are concerned with moving data from one location to another. The 8051 uses five different types of instruction to move data:

MOV	MOVX	MOVC
PUSH and POP	XCH	

MOV
In the 8051 the MOV instruction is concerned with moving data internally, i.e. between Internal RAM, SFR registers, general registers etc. MOVX and MOVC are used in accessing external memory data. The MOV instruction has the following format:

MOV *destination <- source*

The instruction copies *(copy* is a more accurate word than *move)* data from a defined source location to a destination location. Example MOV instructions are:

```
MOV R2, #80h          ; Move immediate data value 80h to register R2
MOV R4, A             ; Copy data from accumulator to register R4
MOV DPTR, #0F22Ch     ; Move immediate value F22Ch to the DPTR register
MOV R2, 80h           ; Copy data from 80h (Port 0 SFR) to R2
MOV 52h, #52h         ; Copy immediate data value 52h to RAM location 52h
MOV 52h, 53h          ; Copy data from RAM location 53h to RAM 52h
MOV A, @R0            ; Copy contents of location addressed in R0 to A
                        (indirect addressing)
```

MOVX
The 8051 the external memory can be addressed using *indirect* addressing only. The DPTR register is used to hold the address of the external data (since DPTR is a 16-bit register it can address 64KByte locations: 2^{16} = 64K). The 8 bit registers R0 or R1 can also be used for indirect addressing of external memory but the address range is limited to the lower 256 bytes of memory (2^8 = 256 bytes).

The MOVX instruction is used to access the external memory (X indicates eXternal memory access). All external moves must work through the A register (accumulator). Examples of MOVX instructions are:

```
MOVX @DPTR, A    ; Copy data from A to the address specified in DPTR
MOVX  A, @DPTR   ; Copy data from address specified in DPTR to A
```

MOVC

MOVX instructions operate on RAM, which is (normally) a volatile memory. Program tables often need to be stored in ROM since ROM is non volatile memory. The MOVC instruction is used to read data from the external code memory (ROM). Like the MOVX instruction the DPTR register is used as the indirect address register. The indirect addressing is enhanced to realise an indexed addressing mode where register A can be used to provide an offset in the address specification. Like the MOVX instruction all moves must be done through register A. The following sequence of instructions provides an example:

```
MOV DPTR, # 2000h       ; Copy the data value 2000h to the DPTR register
MOV A, #80h             ; Copy the data value 80h to register A
MOVC A, @A+DPTR         ; Copy the contents of the address 2080h (2000h + 80h)
                        ; to register A
```

Note, for the MOVC the program counter, PC, can also be used to form the address.

PUSH and POP

PUSH and POP instructions are used with the stack only. The SFR register SP contains the current stack address. Direct addressing is used as shown in the following examples:

```
PUSH 4Ch    ; Contents of RAM location 4Ch is saved to the stack. SP is
              incremented.
PUSH 00h    ; The content of R0 (which is at 00h in RAM) is saved to the stack and
              SP is incremented.
POP 80h     ; The data from current SP address is copied to 80h and SP is
              decremented.
```

XCH

The above move instructions copy data from a source location to a destination location, leaving the source data unaffected. A special XCH (eXCHange) instruction will actually swap the data between source and destination, effectively changing the source data. Immediate addressing may not be used with XCH. XCH instructions must use register A. XCHD is a special case of the exchange instruction where just the lower nibbles are exchanged. Examples using the XCH instruction are:

```
XCH A, R3    ; Exchange bytes between A and R3
XCH A, @R0   ; Exchange bytes between A and RAM location whose address is in R0
XCH A, A0h   ; Exchange bytes between A and RAM location A0h (SFR port 2)
```

Arithmetic

Some key flags within the PSW, i.e. C, AC, OV, P, are utilised in many of the arithmetic instructions. The arithmetic instructions can be grouped as follows:

Addition

Subtraction
Increment/decrement
Multiply/divide
Decimal adjust

Addition
Register A (the accumulator) is used to hold the result of any addition operation. Some simple addition examples are:

```
ADD A, #25h  ; Adds the number 25h to A, putting sum in A
ADD A, R3    ; Adds the register R3 value to A, putting sum in A
```

The flags in the PSW register are affected by the various addition operations, as follows:

The C (carry) flag is set to 1 if the addition resulted in a carry out of the accumulator's MSB bit, otherwise it is cleared.

The AC (auxiliary) flag is set to 1 if there is a carry out of bit position 3 of the accumulator, otherwise it is cleared.

For signed numbers the OV flag is set to 1 if there is an arithmetic overflow (described elsewhere in these notes)

Simple addition is done within the 8051 based on 8 bit numbers, but it is often required to add 16 bit numbers, or 24 bit numbers etc. This leads to the use of multiple byte (multi-precision) arithmetic. The least significant bytes are first added, and if a carry results, this carry is carried over in the addition of the next significant byte etc. This addition process is done at 8-bit precision steps to achieve multi-precision arithmetic. The ADDC instruction is used to include the carry bit in the addition process. Example instructions using ADDC are:

```
ADDC A, #55h    ; Add contents of A, the number 55h, the carry bit; and put the
                  sum in A

ADDC A, R4      ; Add the contents of A, the register R4, the carry bit; and put
                  the sum in A.
```

Subtraction
Computer subtraction can be achieved using 2's complement arithmetic. Most computers also provide instructions to directly subtract signed or unsigned numbers. The accumulator, register A, will contain the result (difference) of the subtraction operation. The C (carry) flag is treated as a borrow flag, which is always subtracted from the minuend during a subtraction operation. Some examples of subtraction instructions are:

```
SUBB A, #55d    ; Subtract the number 55 (decimal) and the C flag from A; and
```

put the result in A.

```
SUBB A, R6          ; Subtract R6 the C flag from A; and put the result in A.

SUBB A, 58h         ; Subtract the number in RAM location 58h and the C flag
                      From A; and put the result in A.
```

Increment/Decrement

The increment (INC) instruction has the effect of simply adding a binary 1 to a number while a decrement (DEC) instruction has the effect of subtracting a binary 1 from a number. The increment and decrement instructions can use the addressing modes: direct, indirect and register. The flags C, AC, and OV are **not** affected by the increment or decrement instructions. If a value of FFh is increment it overflows to 00h. If a value of 00h is decrement it underflows to FFh. The DPTR can overflow from FFFFh to 0000h. The DPTR register cannot be decremented using a DEC instruction (unfortunately!). Some example INC and DEC instructions are as follows:

```
INC R7          ; Increment register R7
INC A           ; Increment A
INC @R1         ; Increment the number which is the content of the address in R1
DEC A           ; Decrement register A
DEC 43h         ; Decrement the number in RAM address 43h
INC DPTR        ; Increment the DPTR register
```

Multiply / Divide

The 8051 supports 8-bit multiplication and division. This is low precision (8 bit) arithmetic but is useful for many simple control applications. The arithmetic is relatively fast since multiplication and division are implemented as single instructions. If better precision, or indeed, if floating point arithmetic is required then special software routines need to be written. For the MUL or DIV instructions the A and B registers must be used and only unsigned numbers are supported.

Multiplication

The MUL instruction is used as follows (note absence of a comma between the A and B operands):

```
MUL AB          ; Multiply A by B.
```

The resulting product resides in registers A and B, the low-order byte is in A and the high order byte is in B.

Division

The DIV instruction is used as follows:

```
DIV AB          ; A is divided by B.
```

The remainder is put in register B and the integer part of the quotient is put in register A.

Decimal Adjust (Special)

The 8051 performs all arithmetic in binary numbers (i.e. it does not support BCD arithmetic). If two BCD numbers are added then the result can be adjusted by using the DA, decimal adjust, instruction:

```
DA A   ; Decimal adjust A following the addition of two BCD numbers.
```

Logical

Boolean Operations
Most control applications implement control logic using Boolean operators to act on the data. Most microcomputers provide a set of Boolean instructions that act on byte level data. However, the 8051 (somewhat uniquely) additionally provides Boolean instruction which can operate on bit level data.

The following Boolean operations can operate on byte level or bit level data:

ANL Logical AND
ORL Logical OR
CPL Complement (logical NOT)
XRL Logical XOR (exclusive OR)

Logical operations at the BYTE level
The destination address of the operartion can be the accumulator (register A), a general register, or a direct address. Status flags are not affected by these logical operations (unless PSW is directly manipulated). Example instructions are:

```
ANL A, #55h  ; AND each bit in A with corresponding bit in number 55h, leaving
               the result in A.

ANL 42h, R4  ; AND each bit in RAM location 42h with corresponding bit in R4,
               leaving the result in RAM location 42h.

ORL A,@R1    ; OR each bit in A with corresponding bit in the number whose address
               is contained in R1 leaving the result in A.

XRL R4, 80h  ; XOR each bit in R4 with corresponding bit in RAM location 80h
               (port 0), leaving result in A.
```

```
CPL R0          ; Complement each bit in R0
```

Logical operations at the BIT level

The C (carry) flag is the destination of most bit level logical operations. The carry flag can easily be tested using a branch (jump) instruction to quickly establish program flow control decisions following a bit level logical operation.

The following SFR registers only are addressable in bit level operations:

PSW IE IP TCON SCON

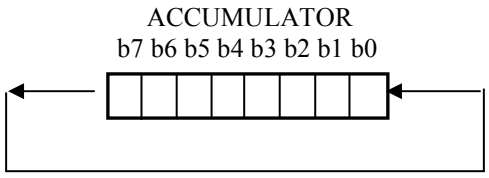
Examples of bit level logical operations are as follows:

```
SETB 2Fh      ; Bit 7 of Internal RAM location 25h is set
CLR  C        ; Clear the carry flag (flag =0)
CPL  20h      ; Complement bit 0 of Internal RAM location 24h
MOV  C, 87h   ; Move to carry flag the bit 7of Port 0 (SFR at 80h)
ANL  C,90h    ; AND C with the bit 0 of Port 1 (SFR at 90)
ORL  C, 91h   ; OR C with the bit 1 of Port 1 (SFR at 90)
```

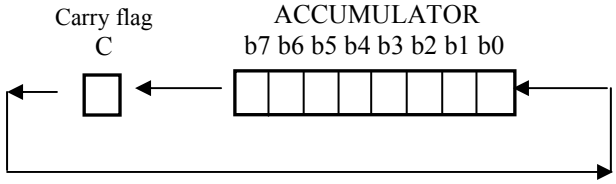
Rotate Instructions

The ability to rotate the A register (accumulator) data is useful to allow examination of individual bits. The options for such rotation are as follows:

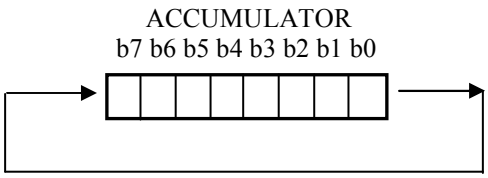
```
RL A          ; Rotate A one bit to the left. Bit 7 rotates to the bit 0 position
```



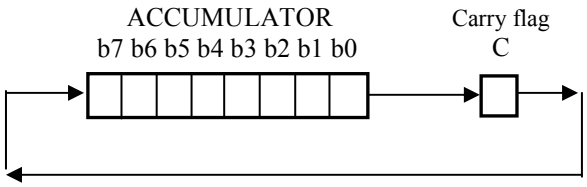

```
RLC A         ; The Carry flag is used as a ninth bit in the rotation loop
```



RR A ; Rotates A to the right (clockwise)

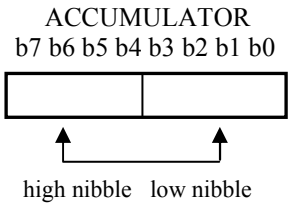


RRC A ; Rotates to the right and includes the carry bit as the 9th bit.



Swap = special
The Swap instruction swaps the accumulator's high order nibble with the low-order nibble using the instruction:

SWAP A



Program Control Instructions
The 8051 supports three kind of jump instructions:

LJMP SJMP AJMP

LJMP
LJMP (long jump) causes the program to branch to a destination address defined by the 16-bit operand in the jump instruction. Because a 16-bit address is used the instruction

can cause a jump to any location within the 64KByte program space (2^{16} = 64K). Some example instructions are:

```
LJMP  LABEL_X     ; Jump to the specified label
LJMP  0F200h      ; Jump to address 0F200h
LJMP  @A+DPTR     ; Jump to address which is the sum of DPTR and Reg. A
```

SJMP
SJMP (short jump) uses a singe byte address. This address is a signed 8-bit number and allows the program to branch to a distance –128 bytes back from the current PC address or +127 bytes forward from the current PC address. The address mode used with this form of jumping (or branching) is referred to as *relative addressing*, introduced earlier, as the jump is calculated relative to the current PC address.

AJMP
This is a special 8051 jump instruction, which allows a jump with a 2KByte address boundary (a 2K page)

There is also a generic JMP instruction supported by many 8051 assemblers. The assembler will decide which type of jump instruction to use, LJMP, SJMP or AJMP, so as to choose the most efficient instruction.

Subroutines and program flow control
A suboutine is called using the LCALL or the ACALL instruction.

LCALL
This instruction is used to call a subroutine at a specified address. The address is 16 bits long so the call can be made to any location within the 64KByte memory space. When a LCALL instruction is executed the current PC content is automatically pushed onto the stack of the PC. When the program returns from the subroutine the PC contents is returned from the stack so that the program can resume operation from the point where the LCALL was made

The return from subroutine is achieved using the RET instruction, which simply pops the PC back from the stack.

ACALL
The ACALL instruction is logically similar to the LCALL but has a limited address range similar to the AJMP instruction.

CALL is a generic call instruction supported by many 8051 assemblers. The assembler will decide which type of call instruction, LCALL or ACALL, to use so as to choose the most efficient instruction.

Program control using conditional jumps

Most 8051 jump instructions use an 8-bit destination address, based on relative addressing, i.e. addressing within the range –128 to +127 bytes.

When using a conditional jump instruction the programmer can simply specify a program label or a full 16-bit address for the conditional jump instruction's destination. The assembler will position the code and work out the correct 8-bit relative address for the instruction. Some example conditional jump instructions are:

```
JZ      LABEL_1     ; Jump to LABEL_1 if accumulator is equal to zero

JNZ     LABEL_X     ; Jump to LABEL_X if accumulator is not equal to zero

JNC     LABEL_Y     ; Jump to LABEL_Y if the carry flag is not set


DJNZ  R2, LABEL     ; Decrement R2 and jump to LABEL if the resulting value of
                      R2 is not zero.

CJNE  R1, #55h , LABEL_2
; Compare the magnitude of R1 and the number 55h and jump to LABEL_2 if the
  magnitudes are not equal.
```

Note, jump instructions such as DJNZ and CJNE are very powerful as they carry out a particular operation (e.g.: decrement, compare) and then make a decision based on the result of this operation. Some example code later will help to explain the context in which such instructions might be used.