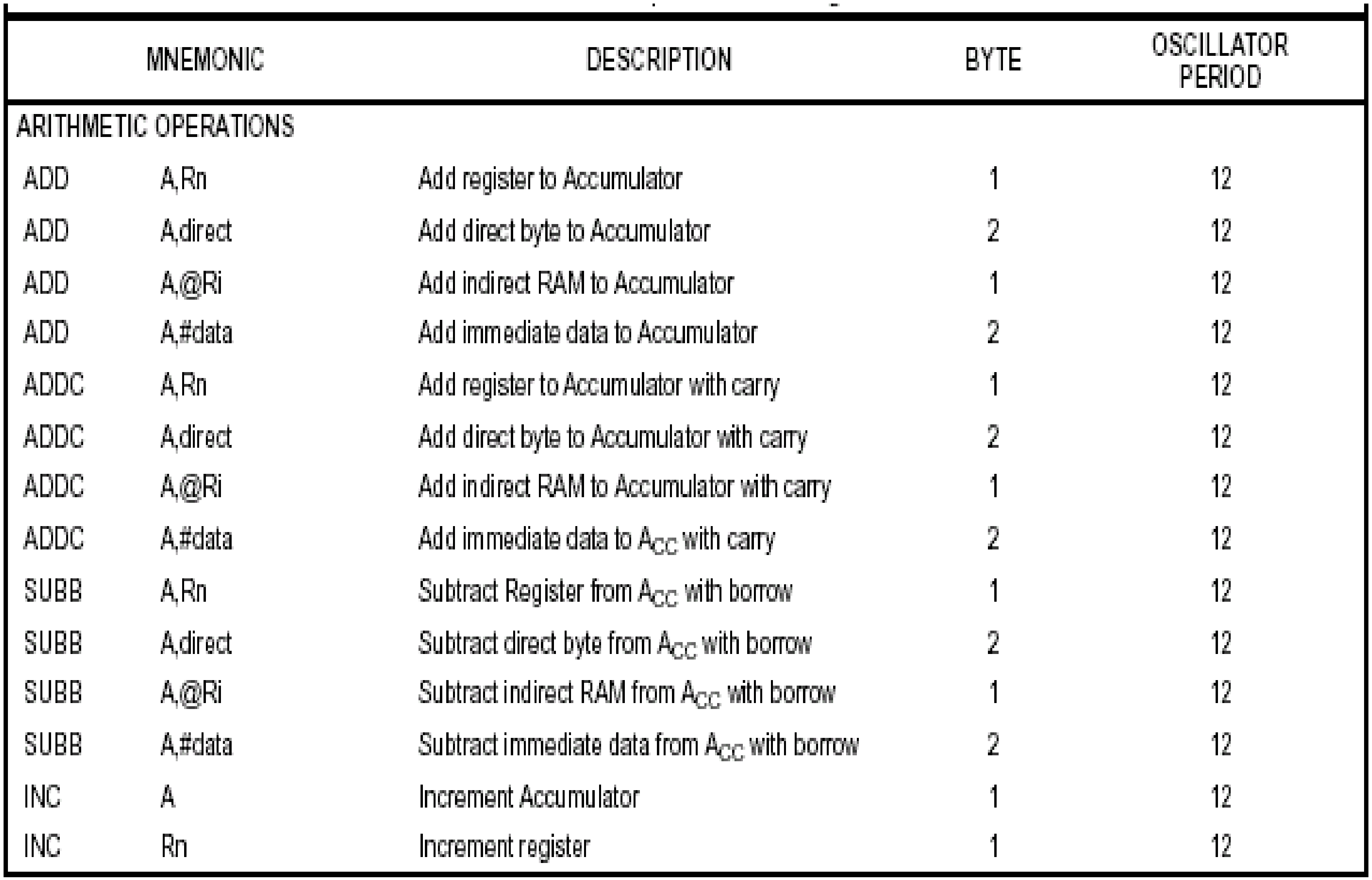
8051 Programming

- The 8051 may be programmed using a low-level or a high-level programming language.

- Low-Level Programming
  - Assembly language programming writes statements that the microcontroller directly executes
  - Advantages
    - 8051 assemblers are free
    - Produces the fastest and most compact code
  - Disadvantages
    - Difficult to learn (8051 assembler has 111 instructions)
    - Slow to program
    - Not portable to other microcontrollers

Assembly Language Instruction Set



MNEMONIC		DESCRIPTION	BYTE	OSCILLATOR PERIOD
ARITHMETIC OPERATIONS				
ADD	A,Rn	Add register to Accumulator	1	12
ADD	A,direct	Add direct byte to Accumulator	2	12
ADD	A,@Ri	Add indirect RAM to Accumulator	1	12
ADD	A,#data	Add immediate data to Accumulator	2	12
ADDC	A,Rn	Add register to Accumulator with carry	1	12
ADDC	A,direct	Add direct byte to Accumulator with carry	2	12
ADDC	A,@Ri	Add indirect RAM to Accumulator with carry	1	12
ADDC	A,#data	Add immediate data to A_{CC} with carry	2	12
SUBB	A,Rn	Subtract Register from A_{CC} with borrow	1	12
SUBB	A,direct	Subtract direct byte from A_{CC} with borrow	2	12
SUBB	A,@Ri	Subtract indirect RAM from A_{CC} with borrow	1	12
SUBB	A,#data	Subtract immediate data from A_{CC} with borrow	2	12
INC	A	Increment Accumulator	1	12
INC	Rn	Increment register	1	12

Source Philips 80C51 Family Programmer's Guide and Instruction Set

8051 Programming

- High-Level Programming
  - Uses a general purpose programming language such as C
  - Advantages
    - Easier to learn
    - Faster to program
    - More portable than assembly language
  - Disadvantages
    - Code may not be as compact or as fast as assembly language
    - Good quality compilers are expensive

8051 Programming Examples

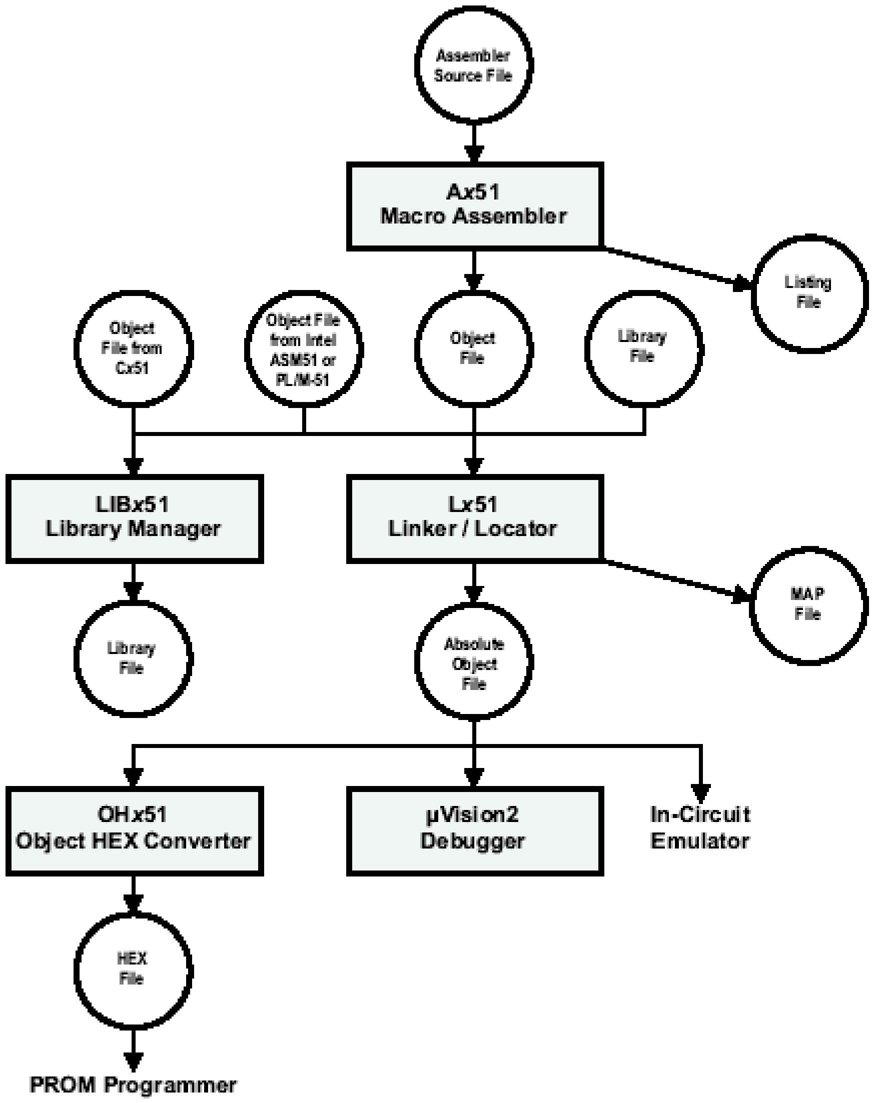
•C program example to add 2 numbers

```
void main()
{
        unsigned char x=5,y=6,z;
        z = x + y;
}
```

•Same code written using assembly language

```
MOV    A,#05H
ADD    A,#06H
MOV    R0,A                    ;result stored in R0
```

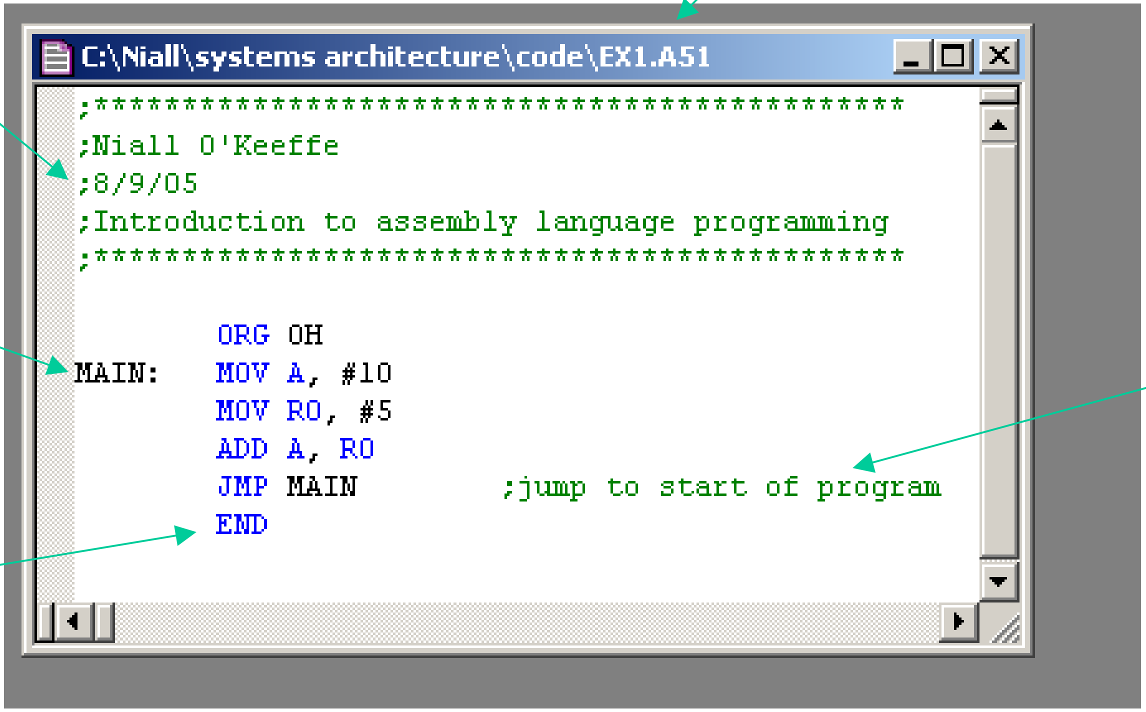
Assembly Language Development Cycle



Rules/Syntax

- All code is normally typed in upper case
- All comments are typed in lower case
  - All comments must be preceded with a semicolon
- All symbols and labels must begin with a letter
- All labels must be followed by a colon
  - Labels must be the first field in a line of assembler code
- The last line of any program must be the END directive

Assembly Programme Example



File is saved with extension .A51
Title Section
label
Code comment
directive
C:\Niall\systems architecture\code\EX1.A51

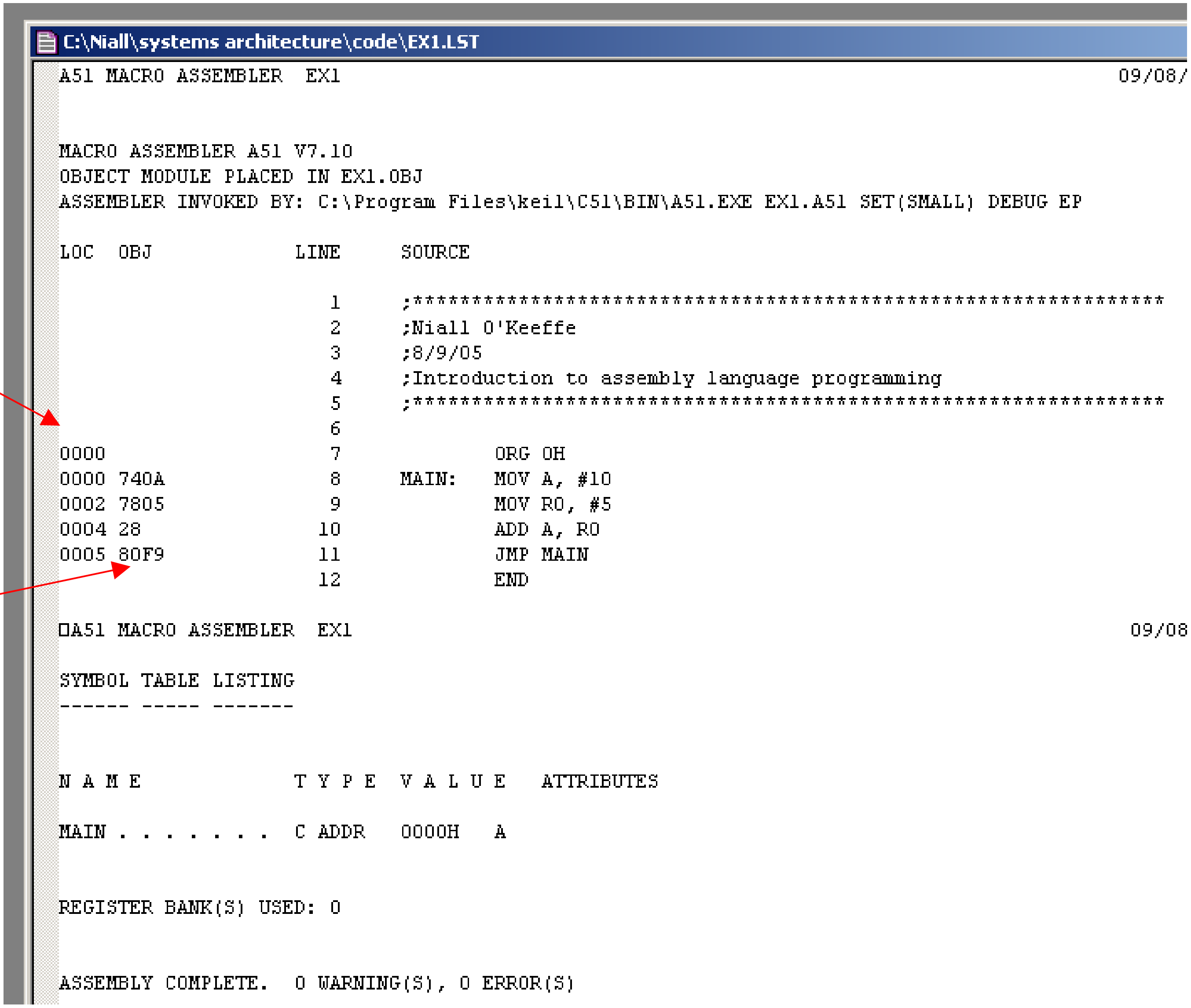
```
;**********************************************
;Niall O'Keeffe
;8/9/05
;Introduction to assembly language programming
;**********************************************

        ORG 0H
MAIN:   MOV A, #10
        MOV R0, #5
        ADD A, R0
        JMP MAIN        ;jump to start of program
        END
```

Listing File Produced by Assembler

Program Memory Address

Machine code


C:\Niall\systems architecture\code\EX1.LST

```
A51 MACRO ASSEMBLER  EX1                                                                      09/08/


MACRO ASSEMBLER A51 V7.10
OBJECT MODULE PLACED IN EX1.OBJ
ASSEMBLER INVOKED BY: C:\Program Files\keil\C51\BIN\A51.EXE EX1.A51 SET(SMALL) DEBUG EP

LOC  OBJ            LINE     SOURCE

                       1     ;*****************************************************************
                       2     ;Niall O'Keeffe
                       3     ;8/9/05
                       4     ;Introduction to assembly language programming
                       5     ;*****************************************************************
                       6
0000                   7             ORG 0H
0000 740A              8     MAIN:   MOV A, #10
0002 7805              9             MOV R0, #5
0004 28               10             ADD A, R0
0005 80F9             11             JMP MAIN
                      12             END

□A51 MACRO ASSEMBLER  EX1                                                                     09/08

SYMBOL TABLE LISTING
------ ----- -------


N A M E             T Y P E  V A L U E   ATTRIBUTES

MAIN . . . . . . .  C ADDR   0000H   A


REGISTER BANK(S) USED: 0


ASSEMBLY COMPLETE.  0 WARNING(S), 0 ERROR(S)
```

8051 Assembly Language

- An assembler program is made up of 3 elements
  - Instructions
  - Assembler Directives
    - Instructions used by the assembler to generate an object file
    - The instructions are not translated to machine code
    - e.g. ORG, END
  - Assembler Controls
    - Define the mode of the assembler
    - e.g. produce a listing file

8051 Instruction Set

The 8051 instruction set can be divided into 5 subgroups: -

- Data Transfer
  - MOV instructions used to transfer data internal and external to the 8051
- Arithmetic
  - Add, subtract, multiply, divide
- Logical
  - AND, OR, XOR, NOT and rotate operations
- Boolean variable manipulation
  - Operations on bit variables
- Program Branching
  - Conditional and unconditional jump instructions

8051 Instruction Set

8051 assembly code contains the following fields:-

<label:> MNEMONIC <DESTINATION>, <SOURCE> <;comment>

- The label and comment fields are optional.
- The mnemonic is the assembler instruction e.g. MOV, ADD
- The destination and source fields are optional
  - It is important to remember that the destination comes first

- The 8051 uses 4 addressing modes: -
  - Immediate Addressing
  - Register Addressing
  - Direct Addressing
  - Register Indirect Addressing

Immediate Addressing

- In immediate addressing the data source is always a number and is specified by a '#'.
  - The number specified is copied into the destination

```
MOV A, #10          ;moves number 10 into Accumulator
MOV R0, #0AH        ;moves number 10 into R0
```

- Assembler Number Representation
  - Default numbering system is decimal
  - Hexadecimal numbers must be followed by the letter H and must begin with a number i.e. the number FA Hex is written as 0FAH.
  - Binary numbers must be followed by the letter B.

  - The following instructions have the same effect

    MOV R0, #255

    MOV R0, #0FFH

    MOV R0, #11111111B

Register Addressing

- Internal registers A, R0 to R7 and DPTR may be used as the source or the destination.

```
MOV A, R0    ;copies contents of R0 to A
```

- Note: - Data may not be copied from Rn to Rn
  - MOV R0, R1 will generate an assembler error
- The source remains unchanged.

Direct Addressing

- Direct Addressing is used in instructions that affect internal data memory locations or the SFR's.
  - The internal data memory address range is 0 to 127 (0 to 7FH)

```
MOV A, 20H          ;copies contents of address 20H into the Accumulator

MOV 30H, 40H        ;copies contents of address 40H to address 30H

MOV P1, A           ;move the contents of the Accumulator to Port 1
```

Indirect Addressing

- The most powerful addressing mode.
- A register is used to store the address of the destination or source of data
  - Similar to the use of pointers in high level languages
  - The @ symbol is used before the register to specify indirect addressing
  - SFRs may not be indirectly addressed
  - Internal data memory may be directly or indirectly addressed

```
MOV R0, #20H        ;Load R0 with the number 20H
MOV @R0, #55H       ;Move 55H to the address contained in R0 (20H)
                    ;R0 acts as a pointer to address 20H
MOV A, @R0          ;Copy the contents of address 20H to the Accumulator
```

- Only registers R0 and R1 may be used for moving data to/from internal data memory when using indirect addressing
- Registers R0, R1 and DPTR may be used when indirectly addressing external memory (more later)

Addressing Modes Exercise

- What are the contents of registers A, R0, R7 and memory locations 30H and 31H after the following code runs: -

```
MOV A, #5
MOV R7, #40H
MOV R0, #30H
MOV 31H, #14H
MOV @RO, A
INC R0
MOV R7, @R0
```

- How long does the code take to execute if the 8051 is operating off a 12MHz crystal?

Some Useful Directives

- END
  - Last line of code. Assembler will not compile after this line
- ORG
  - Origin directive. Sets the location counter address for the following instructions
- EQU
  - Equate directive. Used to equate a name with an address or a data value. Useful for constant assignments.
- DATA
  - Used to assign a symbol name to an address in internal data memory

    AVERAGE DATA 30H

    MOV AVERAGE, A
- BIT
  - Used to assign a symbol name to a bit in bit-addressable data memory

Programme Sequencing

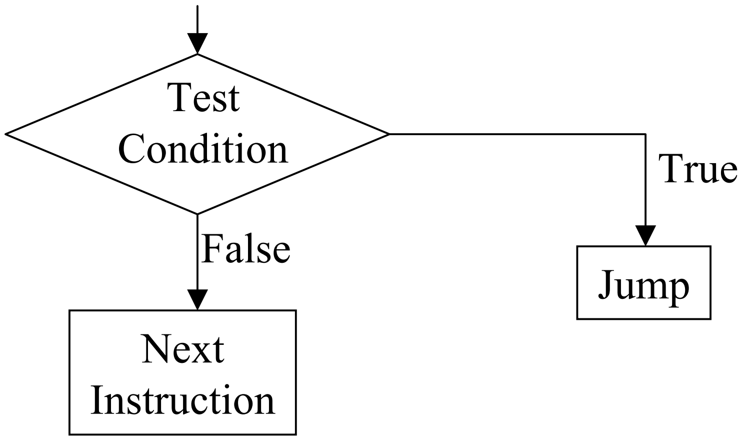
- Normal program execution is sequential
  - The PC is loaded with the address of instruction N+1 while instruction N is being executed
- The program branching instructions allow the programmer to alter the program execution sequence
  - These instructions allow the address contained in the PC to be changed
  - Program branching is used for jumps, function calls and interrupt service routines.

Program Branching Instructions

PROGRAM BRANCHING				
ACALL	addr11	Absolute subroutine call	2	24
LCALL	addr16	Long subroutine call	3	24
RET		Return from subroutine	1	24
RETI		Return from interrupt	1	24
AJMP	addr11	Absolute jump	2	24
LJMP	addr16	Long jump	3	24
SJMP	rel	Short jump (relative addr)	2	24
JMP	@A+DPTR	Jump indirect relative to the DPTR	1	24
JZ	rel	Jump if Accumulator is zero	2	24
JNZ	rel	Jump if Accumulator is not zero	2	24
CJNE	A,direct,rel	Compare direct byte to A_{CC} and jump if not equal	3	24
CJNE	A,#data,rel	Compare immediate to A_{CC} and jump if not equal	3	24
CJNE	RN,#data,rel	Compare immediate to register and jump if not equal	3	24
CJNE	@Ri,#data,rel	Compare immediate to indirect and jump if not equal	3	24
DJNZ	Rn,rel	Decrement register and jump if not zero	2	24
DJNZ	direct,rel	Decrement direct byte and jump if not zero	3	24
NOP		No operation	1	12

Jump Instructions

- The 8051 has 2 types of JUMP instructions
- Unconditional Jump
  - This instruction type will load the PC with a new address and will automatically jump to the instruction at that address
- Conditional Jump
  - This instruction type will only jump if a certain condition is true
  - Similar to an "if" statement in C.



Unconditional Jumps

The 8051 has 3 unconditional jump instructions with a different range: -

- SJMP (Short Jump)
  - Allows a jump of –128 to +127 bytes relative to the current PC value
  - Instruction is 2 bytes long
- AJMP (Absolute Jump)
  - Allows a jump with the same 2KByte page that the PC is currently located in
  - Instruction is 2 bytes long
- LJMP (Long Jump)
  - Allows a jump anywhere within the 64KByte program memory range of the 8051
- If unsure which of the 3 instructions to use, simply use the JMP instruction and let the assembler decide which instruction to use.

Conditional Jumps

- The 8051 can test conditions at the bit and byte level

- Bit Conditional Jump Instructions
  - These instructions will jump if a bit is in a certain state
  - e.g. JC label ;jump to label if carry bit is set
  - JNC label2 ;jump to label2 if the carry bit is clear
  - These instructions are commonly used for arithmetic instructions and for the testing of flags

- Byte Conditional Jump Instructions
  - DJNZ – Decrement and Jump if Not Zero
  - CJNE – Compare and Jump if Not Equal

DJNZ Instruction

- Decrement and Jump if Not Zero
  - DJNZ Rn, label
  - DJNZ direct address, label

- DJNZ is used to execute a block of code N times
  - Similar to a for or while loop in C
  - Very useful for generating delays

```
        MOV R0, #10       ;R0 = loop counter
LOOP:   DJNZ R0, LOOP     ;DJNZ instruction executed 10 times
        MOV A, R1
```

DJNZ for Generating Delays

```
        MOV R0, #10        ;R0 = loop counter
LOOP:   DJNZ R0, LOOP      ;DJNZ instruction executed 10 times
        MOV A, R1
```

- The DJNZ instruction takes 2 machine cycles to execute (24 clocks)
- If the 8051 is operating from a 12MHz crystal, the loop execution time is
  (10 * 24)/12000000 = 20usec

- The maximum delay for a single loop occurs when the loop counter is initialised to 0
  - This will cause 256 loop iterations
  - Delay = (256 * 24)/12000000 = 512usec

- How do we generate delays longer than 512usec?

DJNZ for Generating Delays

- Longer delays may be generated by using nested DJNZ instructions

```
        MOV R0, #0          ;12 clocks
        MOV R1, #200        ;12 clocks
LOOP:   DJNZ R0, LOOP       ;256 * 24 clocks
        DJNZ R1, LOOP       ;executes inner loop + DJNZ 200 times
```

- *Execution time is (12 + 12 + 200((256*24) + 24))/12000000 = 0.102802 sec*

- *Rewrite the code to generate a delay of 0.1usec accurate to 10usec*

DJNZ Exercise

```
        MOV R0, #0
        MOV R1, #0
        MOV R2, #10
LOOP:   DJNZ R0, LOOP
        DJNZ R1, LOOP
        DJNZ R2, LOOP
```

1. *How long does the above code take to execute if the 8051 is operating off a 12MHz crystal?*
2. *Repeat part 1 for a 16MHz crystal*
3. *Rewrite the code to generate a delay of 1 second accurate to 10usec (assume a 12MHz crystal)*

CJNE Instruction

- Compare and Jump if Not Equal to Zero

  CJNE destination, source, label

- The destination and source bytes are compared and a jump takes place if they are not equal.
  - The carry flag is set if the destination byte is less than the source byte
- Often used to validate characters received via the serial port
- May be used for delays but code is not as efficient as DJNZ

```
        MOV R0, #10
LOOP:   CJNE R0, #0, LOOP1
        JMP DONE
LOOP1:  DEC R0
        JMP LOOP
DONE:
```

Subroutines

- A subroutine is a block of code that can be used many times in the execution of a larger program (similar to functions in higher level languages)
- Subroutines allow the program to branch to a section of code and to remember where it branched from.
- When the subroutine is complete program execution will continue from the line of code following the subroutine call.

- Subroutines have the following advantages: -
  - Code savings
    - The same subroutine may be called over and over again
  - Program structuring
    - A large program may be divided into a number of small subroutines
    - This makes the program easer to maintain and debug

Subroutine Example

```
        ORG 0000H
MAIN:   ……….
        ……….
        CALL SUB1       ;subroutine call (store PC on stack)
        CALL SUB2
        ……….
        JMP MAIN

SUB1:   ……….            ;label defines start of subroutine
        ……….
        RET             ;return to line after CALL

SUB2:   ……….
        ……….
        RET
        END
```

Subroutines

- A subroutine is always executed with the CALL instruction
  - When the CALL instruction is executed the PC register contains the address of the next instruction to be executed (this is known as the return address)
  - The PC is saved onto the stack low byte first
  - The PC is then loaded with the address of the subroutine
  - The subroutine is then executed

- The last line of a subroutine is always the RET instruction
  - The RET instruction will cause the return address to be popped off the stack and loaded into the PC
  - The instruction at the return address is then executed

Subroutine Parameter Passing

1. Place the parameter into an address in internal data memory
   Most popular method used

2. Push the parameter onto the stack
   This method is limited by the size of the stack

3. Place the parameter into external data memory
   Used when internal data memory has been used up.
   Slower execution speed than when using internal data memory

Subroutine Call With No Parameter Passing

```
;code to output a waveform on P1.0 that is high for 5 seconds and low for 2.5 seconds
          ORG 0
MAIN:     CALL ON
          CALL OFF
          JMP MAIN

ON:       SETB P1.0
          MOV R0, #40
          MOV R1, #0
          MOV R2, #0
DELAY1:   DJNZ R2, DELAY1
          DJNZ R1, DELAY1
          DJNZ R0, DELAY1
          RET

OFF:      CLR P1.0
          MOV R0, #20
          MOV R1, #0
          MOV R2, #0
DELAY2:   DJNZ R2, DELAY2
          DJNZ R1, DELAY2
          DJNZ R0, DELAY2
          RET

          END
```

Subroutine Call With Parameter Passing

```
;code to output a waveform on P1.0 that is high for 5 seconds and low for 2.5 seconds
        ORG 0
MAIN:   CALL ON
        CALL OFF
        JMP MAIN

ON:     SETB P1.0
        MOV R0, #40
        CALL DELAY
        RET

OFF:    CLR P1.0
        MOV R0, #20
        CALL DELAY
        RET

DELAY:  MOV R1, #0
        MOV R2, #0
LOOP:   DJNZ R2, LOOP
        DJNZ R1, LOOP
        DJNZ R0, LOOP
        RET

        END
```

8051 Arithmetic Operations

- All arithmetic operations are carried out in the ALU
- The 8051 has 4 arithmetic flags that are stored in the PSW register

1. C Carry Flag
   Set if there is a carry out after addition or a borrow after subtraction.
   Used for unsigned arithmetic.

2. AC Auxiliary Carry Flag
   Set if there is a carry out of bit 3 during addition or a borrow during subtraction.
   Used for BCD arithmetic.

3. OV Overflow Flag
   Set if there is a carry from bit 6 XOR bit 7
   Used for signed arithmetic

4. P Parity Flag
   Contains the parity of the accumulator. 1 if odd, 0 if even. Useful for some serial port operations.

Increment/Decrement Instructions

- INC Source
  - Adds 1 to the source
- DEC Source
  - Subtract 1 from the source
- Source may be a register or a direct or indirect address
  - INC A
  - DEC R1
  - INC 30H
  - DEC @R0
- No flags are affected by the INC and DEC instructions

Multiply/Divide Instructions

- MUL AB
  - Note no comma between source and destination
  - Multiplies the A register by the B register. The low order byte of the result is placed in A, the high order byte in B.
  - The OV flag is set if the result exceeds 255

- DIV AB
  - Divides A register by B register.
  - The integer part of the quotient is placed in A, the integer part of the remainder is placed in B.
  - OV flag is set if a divide by 0 is attempted

Addition

ADD A, SOURCE ;A = A + Source

ADDC A, SOURCE ;A = A + Source + C

- The accumulator is always used to store the result
- All addressing modes may be used.

- Affect on flags
  - Carry bit C is set if there is a carry out of bit 7, cleared otherwise.
    - Used for unsigned addition
  - AC flag set if there is a carry from bit 3, cleared otherwise
    - Used for BCD addition
  - OV flag is set if C7 XOR C6, cleared otherwise
    - Used for signed addition

Unsigned Addition

- The carry bit C should always be tested after an addition to see if the result has exceeded 255 (FFH).
  - *JC Label ;jump if carry bit is set*
  - *JNC Label ;jump if carry bit is clear*

Examples: -

```
25      00011001
47      00101111
72      01001000     No carry (result <=255)

65        01000001
208       11010000
273     1 00010001   Carry (result > 255)
```

Unsigned Addition

- Write a program to add the contents of internal data memory locations 30H and 31H If a carry occurs, set the pin P1.0

```
        ERROR BIT P1.0
MAIN:   CLR ERROR
LOOP:   MOV A, 30H
        ADD A, 31H
        JNC MAIN            ;no carry
        SETB ERROR          ;carry, set error pin
        JMP LOOP
        END
```

Signed Addition

- For signed arithmetic bit 7 of the number is used as a sign bit.
  - 1 for a negative number and 0 for a positive number
  - Number range is restricted to –128 to +127
- The OV flag should always be tested after adding 2 signed numbers
  - The OV flag will only change when adding numbers of the same sign yields a result outside of the range –128 to +127

Examples: -

+25	00011001	
-45	11010011	
-20	11101100	;OV = 0, take no action
+120	01111000	
+48	00110000	
+168	10101000	;OV = 1, result is –88? Need to adjust result

Signed Addition

+120	01111000	
+48	00110000	
+168	10101000	;OV = 1, result is –88? Need to adjust result

- How do we adjust the result to get the correct value (+168)?
  - Remember that the result range is –128 to +127
  - If there is an overflow this range has been exceeded
  - Invert bit 7 to get the correct polarity for the result
  - The result then needs to be adjusted by +/-128 depending on whether we are adding positive or negative numbers

- For the above example, inverting the result bit 7 yields 00101000 (+40)
  - Because of the overflow the real result is 40 +128 = 168.

Adding 2-Byte Numbers

- Write a program to add 2 integers.
  - Integer 1 is stored at addresses 30H and 31H (low byte at address 30H)
  - Integer 2 is stored at addresses 32H and 33H (low byte at address 32H)
  - The result should be stored at addresses 34H to 36H (low byte at address 34H)

- Hint
  - Use the ADDC instruction instead of ADD

BCD Addition

- The 8051 performs addition in pure binary – this may lead to errors when performing BCD addition

Example

```
49 BCD     01001001 BCD
38  BCD    00111000 BCD
-----------------------
87 BCD     10000001 (81BCD)
```

- The result must be adjusted to yield the correct BCD result
  - DA A (decimal adjust instruction)
  - The carry flag is set if the adjusted number exceeds 99 BCD

```
MOV A, #9
ADD A, #11      ;A = 1AH (expecting 20H if these are BCD numbers)
DA A            ;A = 20H
```

Subtraction

- *SUBB A, SOURCE*
  - Subtracts source and carry flags from A
  - Result placed in A

- For unsigned subtraction the carry flag C is set if there is a borrow needed for bit 7

- For signed subtraction the OV flag is set if the subtraction of a negative number from a positive number yields a negative result or if the subtraction of a positive number from a negative number yields a positive result.

8051 Logical Operations

- All 4 addressing modes may be used for the 8051 logical instructions

- AND
  - *ANL A,SOURCE*
  - May be used to selectively clear bits in the destination operand
  - e.g. *ANL A, #11111100B ;will clear lower 2 bits of A register*

- OR
  - *ORL A, SOURCE*
  - May be used to selectively set bits in the destination operand
  - *ORL A, #00000001B ;will set bit 0 of A register*

- XOR
  - *XRL A, SOURCE*
  - May be used to selectively complement bits in the destination operand
  - *XRL P1, #00001111B ;will complement lower 4 bits of port 1*

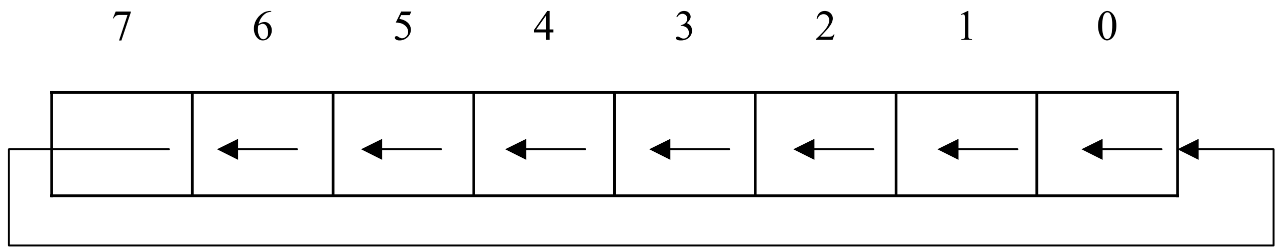
8051 Logical Operations

- Complement
  - *CPL A*
  - Complements each bit of the A register

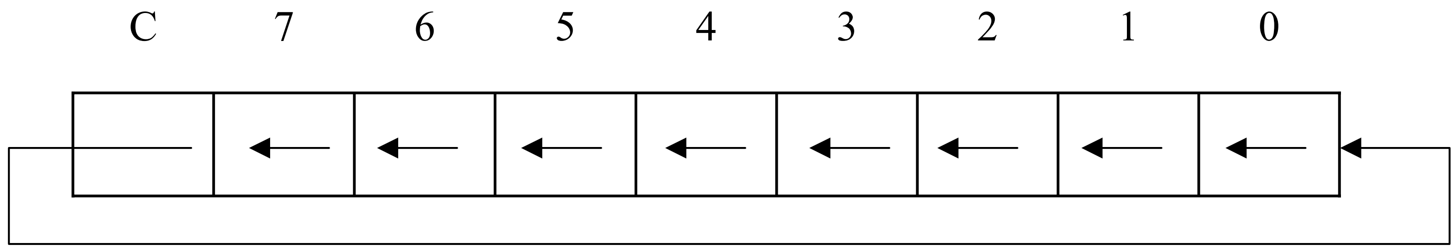
- Clear
  - *CLR A*
  - Clears each bit of the A register

Rotate Operations

- All rotate operations are carried out on the accumulator
- *RL A*
  - Rotate accumulator left, MSB becomes LSB



- *RLC A*
  - Rotate accumulator left, MSB becomes carry, carry becomes LSB



- Similar operations for rotate right: - RR A, RRC A

Bit Level Logical Operations

- These instructions allow a single bit to be altered without affecting the entire byte
  - Can be used to set/clear a bit in bit-addressable memory
  - Can be used to set/clear an I/O pin

- *SETB BIT ;sets bit high*
- *CLR BIT ;clears bit low*

- Example

```
SETB P1.0       ;P1.0 = '1'
CLR P1.0        ;P1.0 = '0'

or

LED BIT P1.0    ;use BIT directive to name pin
SETB LED
```

Bit-level Boolean Operations

BOOLEAN VARIABLE MANIPULATION				
CLR	C	Clear carry	1	12
CLR	bit	Clear direct bit	2	12
SETB	C	Set carry	1	12
SETB	bit	Set direct bit	2	12
CPL	C	Complement carry	1	12
CPL	bit	Complement direct bit	2	12
ANL	C,bit	AND direct bit to carry	2	24
ANL	C,/bit	AND complement of direct bit to carry	2	24
ORL	C,bit	OR direct bit to carry	2	24
ORL	C,/bit	OR complement of direct bit to carry	2	24
MOV	C,bit	Move direct bit to carry	2	12
MOV	bit,C	Move carry to direct bit	2	24
JC	rel	Jump if carry is set	2	24
JNC	rel	Jump if carry not set	2	24
JB	rel	Jump if direct bit is set	3	24
JNB	rel	Jump if direct bit is not set	3	24
JBC	bit,rel	Jump if direct bit is set and clear bit	3	24

Look-Up Tables

- A look-up table is a table of constants stored in program memory
  - Look-up tables can be used to speed up arithmetic operations
  - The look-up table may be accessed using the DPTR or PC as a pointer to the start of the table. The A register is used as an index to the table.

    MOVC A, @A+DPTR

    MOVC A, @A+PC

  - The look-up table is defined using the DB directive

    ORG 200H

    DB 1,2,4,9
  - This code will create a look-up table at address 200H. The value 1 will be stored at address 200H, 2 at address 201H etc
  - Ensure that the look-up table does not overlap with the address space used by code.

Look-up Table Example

- Write a program to read an 8-bit temperature in Celsius from Port 1 and to output the Farenheight temperature equivalent onto Port 2
  - F = ((C * 9)/5) + 32

- This temperature conversion could be coded in 2 ways: -
  - Use arithmetic operations to work out the formula
    - This would involve a multiply and a divide operation which are the 2 instructions with the longest execution time
    - The code would also have to deal with the 2-byte result of the multiply

  - Use a look-up table that stores the Farenheight equivalent of all possible Celsius readings
    - This would require more program memory – 1 byte for each temperature
    - The code is much simpler to implement

Temperature Conversion Program

```
        TABLE EQU 100H
        ORG 0
MAIN:   MOV DPTR, #TABLE
LOOP:   MOV A, P1
        MOVC A, @A+DPTR
        MOV P2, A
        JMP LOOP

        ;conversion look-up table for 0 to 40 degrees Celsius
        ORG TABLE
        DB 32,34,36,37,39,41,43,45,46,48,50,52,54,55,57,59,61,63,64,66
        DB 70,72,72,75,77,79,81,82,84,86,88,90,91,93,95,97,99,100,102,104
        END

;What happens if a value outside of the range 0 to 40 is read from Port 1?
;How do you deal with this scenario in code?
```