Embedded C programming basics :

Factors for Selecting the Programming Language

•**Size:** The memory that the program occupies is very important as Embedded Processors like Microcontrollers have a very limited amount of ROM.
•**Speed:** The programs must be very fast i.e. they must run as fast as possible. The hardware should not be slowed down due to a slow running software.
Portability: The same program can be compiled for different processors.
•**Ease of Implementation**
•**Ease of Maintenance**
•**Readability**

Embedded systems programming

•Embedded devices have resource constraints

•embedded systems typically uses smaller, less power consuming components.

•Embedded systems are more tied to the hardware.

Embedded systems programming

- Machine Code
- Low level language, i.e., assembly
- High level language like C, C++, Java, Ada, etc.
- Application level language like Visual Basic, scripts, Access, etc

Assembly Language Programming

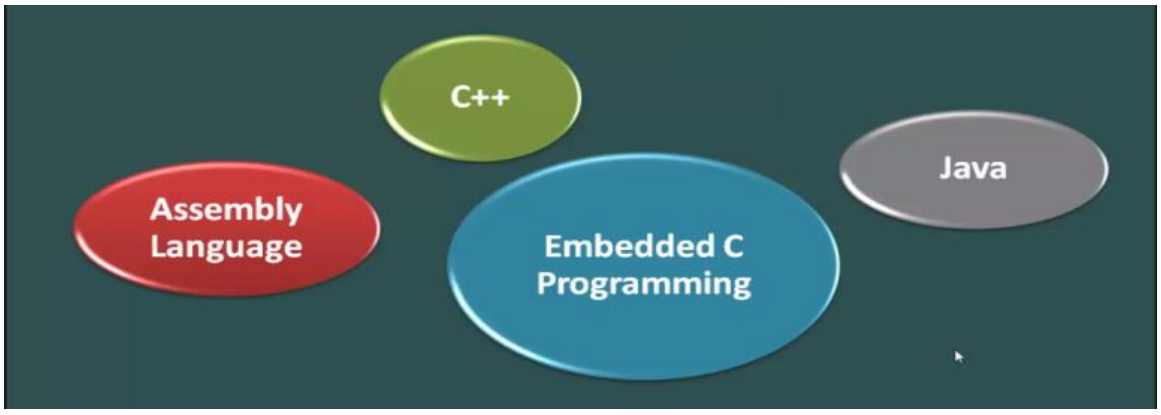
```
HERE:   MOV   R0,#01H
        MOV   R1,#02H
        MOV   A,R0
        ADD   A,R1
        MOV   P0,A
        SJMP  HERE
```

Address	Opcode	Operand
0000	78	01
0002	79	02
0004	E8	
0005	29	
0006	F5	80
0008	80	00

Use of C in embedded systems is driven by following advantages

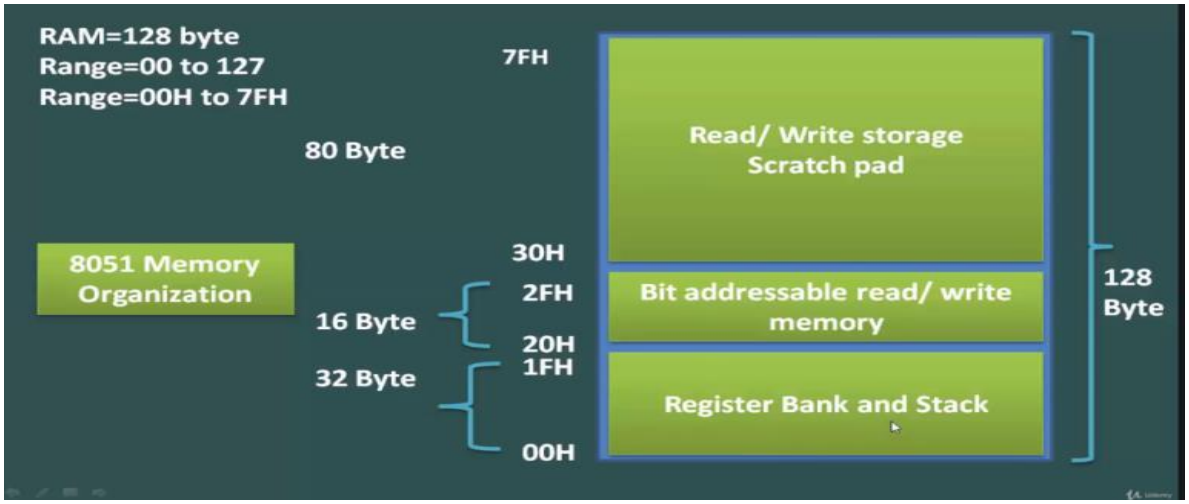
- it is small and reasonably simpler to learn, understand, program and debug.
- C Compilers are available for almost all embedded devices
- C has advantage of processor-independence

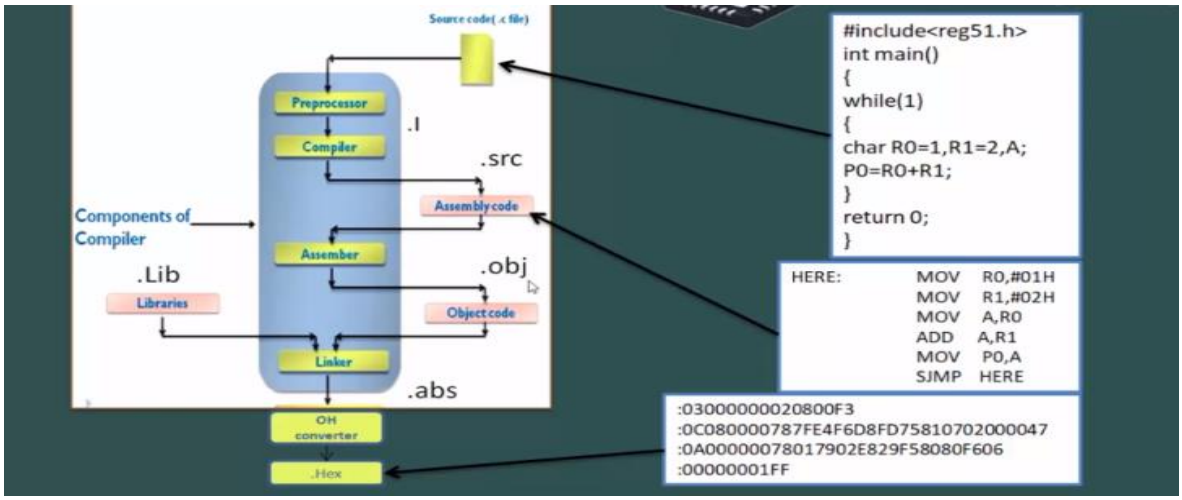
- C combines functionality of assembly language and features of high level languages
- it is fairly efficient

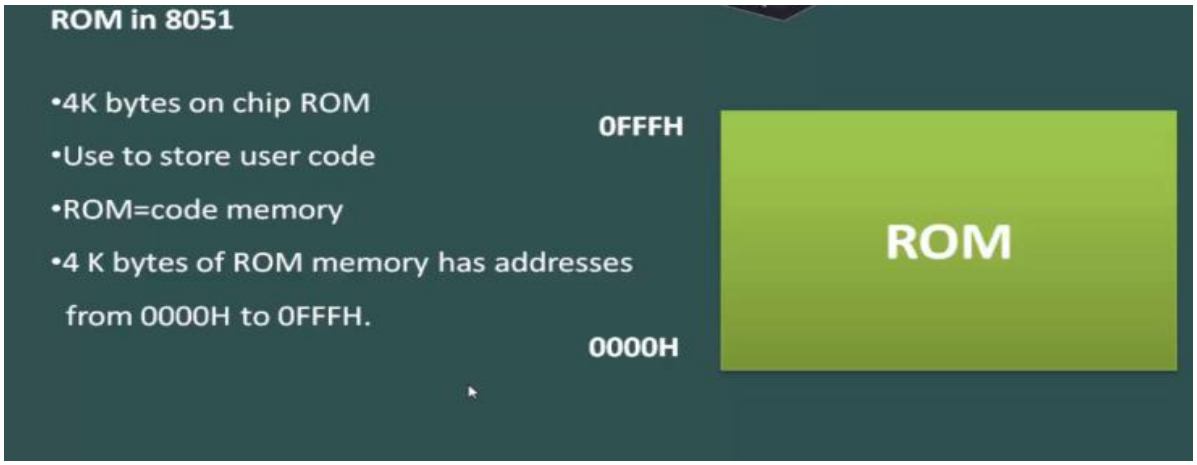


Difference between C and Embedded C

Though **C and embedded C** appear different and are used in different contexts, they have more similarities than the differences. Most of the constructs are same; the difference lies in their applications.


RAM=128 byte
Range=00 to 127
Range=00H to 7FH
7FH
80 Byte
Read/ Write storage
Scratch pad
8051 Memory
Organization
30H
2FH
16 Byte
Bit addressable read/ write
memory
20H
1FH
32 Byte
Register Bank and Stack
00H
128
Byte


Source code(.c file)
Preprocessor
.I
Compiler
.src
Assembly code
Components of
Compiler
Assembler
.obj
.Lib
Libraries
Object code
Linker
.abs
OH
converter
.Hex
#include<reg51.h>
int main()
{
while(1)
{
char R0=1,R1=2,A;
P0=R0+R1;
}
return 0;
}
HERE: MOV R0,#01H
MOV R1,#02H
MOV A,R0
ADD A,R1
MOV P0,A
SJMP HERE
:03000000020800F3
:0C080000787FE4F6D8FD75810702000047
:0A000000780179 02E829F58080F606
:00000001FF


ROM in 8051
•4K bytes on chip ROM
•Use to store user code
•ROM=code memory
•4 K bytes of ROM memory has addresses
from 0000H to 0FFFH.
0FFFH
0000H
ROM

Automatic variables

Declare within a function/procedure

Variable is visible (has scope) only within that function

Space for the variable is allocated on the system stack when the procedure is entered
De-allocated, to be re-used, when the procedure is exited

If only 1 or 2 variables, the compiler may allocate them to registers within that procedure, instead of allocating memory

Values are not retained between procedure calls

Program variables

```
Int x,y,z;     //declares 3 variables of type "int"
char a,b;      //declares 2 variables of type "char"
```

Space for variables may be allocated in registers, RAM, or ROM/Flash

Variables can be automatic or static

```
void delay ( )
 {
Int i,j;                          //automatic variables –visible only within delay( )

for (i=0; i<100; i++)             //outer loop
  {
for (j=0; j<20000; j++)           //inner loop
     {
     }                            //do nothing
  }
 }
```

Static variables

Retained for use throughout the program in RAM locations that are not reallocated during program execution.

Declare either within or outside of a function

If declared outside a function, the variable is global in scope, i.e. known to all functions of the program

Use “normal” declarations.

Example: int count;

If declared within a function, insert key word static before the variable definition. The variable is local in scope, i.e. known only within this function.

Example: static int count;

C control structures

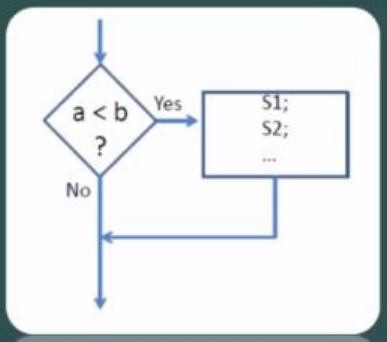
Control order in which instructions are executed

- Conditional execution
  - Execute a set of statements if some condition is met
  - Select one set of statements to be executed from several options, depending on one or more conditions
- Iterative execution
  - Repeated execution of a set of statements
    - A specified number of times, or
    - Until some condition is met, or
    - While some condition is true

IF-THEN structure

Execute a set of statements if and only if some condition is met

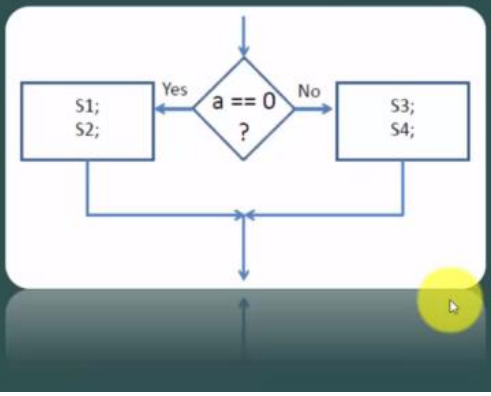
```
if (a < b)
{
statement s1;
statement s2;
....
}
```



IF-THEN-ELSE structure

Execute one set of statements if a condition is met and an alternate set if the condition is not met.

```
if (a == 0)
{
statement s1;
statement s2;
}
else
{
statement s3;
statement s4:
}
```



Multi-way decision, with expressions evaluated in a specified order

```
if (n == 1)
{
statement1;          //do if n == 1
else if (n == 2)
statement2;          //do if n == 2
else if (n == 3)
statement3;          //do if n == 3
else
statement4;          //do if any other value of n
}
```

Test	TRUE condition
(m == b)	m equal to b
(m != b)	m not equal to b
(m < b)	m less than b
(m <= b)	m less than or equal to b
(m > b)	m greater than b
(m >= b)	m greater than or equal to b
(m)	m non-zero
(1)	always TRUE
(0)	always FALSE

Boolean operators &&(AND) and ||(OR) produce TRUE/FALSE results when testing multiple TRUE/FALSE conditions

```
if ((n > 1) && (n < 5))     //test for n between 1 and 5
if ((c = 'q') || (c = 'Q'))  //test c = lower or upper case Q
```

Note the difference between Boolean operators &&, || and bitwise logical operators &, |

Note that ==is a relational operator, whereas =is an assignment operator

SWITCH statement

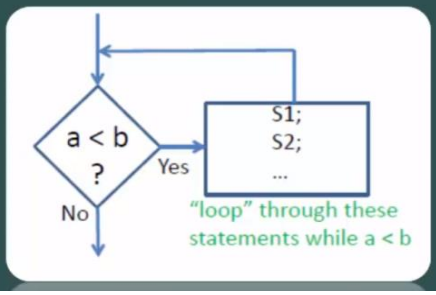
Compact alternative to ELSE-IF structure, for multiway decision that tests one variable or expression for a number of constant values.

```
switch ( n)                 //n is the variable to be tested
{
case 0: statement1;         //do if n == 0
case 1: statement2;         // do if n == 1
case 2: statement3;         // do if n == 2
default: statement4;        //if for any other n value
 }
```

WHILE loop structure

Repeat a set of statements (a "loop") as long as some condition is met

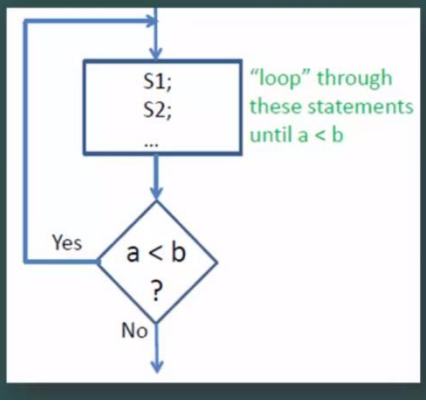
```
while (a < b)
{
statement s1;
statement s2;
....
}
```



DO-WHILE loop structure

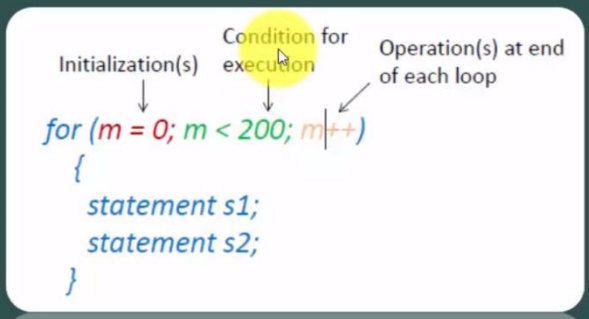
Repeat a set of statements (one "loop") **until some condition is met**

```
do
{
statement s1;
statement s2;
 ....
}
while (a < b);
```



FOR loop structure

FOR loop is a more compact form of the WHILE loop structure



```
/* Nested FOR loops to create a time delay */

for (i = 0; i < 100; i++)
{
                //do outer loop 100 times

    for (j = 0; j < 1000; j++)

        {
                 //do inner loop 1000 times
                //do "nothing" in inner loop
        }
}
```

```
/* Nested FOR loops to create a time delay */

for (i = 0; i < 100; i++)
{
                //do outer loop 100 times

    for (j = 0; j < 1000; j++)

        {
                 //do inner loop 1000 times
                //do "nothing" in inner loop
        }
}
```

Total no. of iteration
= 1000 x100

C function

Functions partition large programs into a set of smaller tasks

- Helps manage program complexity
- Smaller tasks are easier to design and debug
- Functions can often be reused instead of starting over
- Can use of "libraries" of functions developed by 3rdparties, instead of designing your own
- The function may return a result to the caller
- One or more arguments may be passed to the function/procedure

3. Function in Embedded C

Embedded C Programming

```
#include<reg51.h>

Int  math_func( int k; int  n)          Function Declaration

Void main()
{
Int a,b,c;
a = 10; b =20;
c=math_func (a,b);                      Function call
}

Int  math_func( int k; int  n)
{
Int  j;          //local variable
j = n + k -5;    //function body        Function definition
return(j);       //return the result
}
```

Constant/literal

Constants in C programming language, as the name suggests are the data that doesn't change. Constants are also known as literals.

Integer constants

```
123    /* decimal constant*/
0x9b   /* hexadecimal constant*/
O456   /* octal constant*/
```

For decimal literals :	no prefix is used.
Prefix used for hexadecimal:	0x / 0X
Prefix used for octal:	0

Character constants

Character constants hold a single character enclosed in single quotations marks

```
m = 'a';      //ASCII value 0x61
m= 0x01
```

String Constants/Literals

String constants consist of any number of consecutive characters in enclosed quotation marks (").

String(array) of characters:

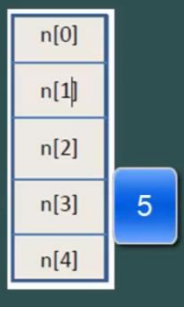
```
char my_string[] = "My String";

// Compiler will interpret the above statement as

char my_string[10] = {'M', 'y', ' ', 'S', 't', 'r', 'i', 'n', 'g', '\0'};
```

Null Character

Variable arrays

- An array is a set of data, stored in consecutive memory locations, beginning at a named address
  - Declare array name and number of data elements, N
  - Elements are "indexed", with indices [0 .. N-1]

```
Int n[5];    //declare array of 5 "int" values
n[3] = 5;    //set value of 4th array element
```



Base	Prefix
Binary:	None
Decimal:	None
Hexadecimal:	**0x** or **0X**
Octal:	**0** (zero)

```
unsigned int n;
n = 0x64;        //HexaDecimal
n = 100;         //Decimal
n = 0144         //Octal
```

Basic data types in C51 compiler

Data Type	Bits	Bytes	Value Range
bit	1		0 to 1
signed char	8	1	-128 to +127
unsigned char	8	1	0 to 255
enum	16	2	-32768 to +32767
signed short	16	2	-32768 to +32767
unsigned short	16	2	0 to 65535
signed int	16	2	-32768 to +32767
unsigned int	16	2	0 to 65535
signed long	32	4	-2147483648 to 2147483647
unsigned long	32	4	0 to 4294967295
float	32	4	+/-1.175494E-38 to +/-3.402823E+38
sbit	1		0 to 1
sfr	8	1	0 to 255
sfr16	16	2	0 to 65535

Arithmetic operations

```
Int  i, j, k;              // 32-bit signed integers
uint8_t m,n,p;             // 8-bit unsigned numbers

i= j + k;                  // add 32-bit integers
m = n -5;                  // subtract 8-bit numbers
j = i* k;                  // multiply 32-bit integers
m = n / p;                 // quotient of 8-bit divide
m = n % p;                 // remainder of 8-bit divide
i= (j + k) * (i–2);        //arithmetic expression
```

*, /, % are higher in precedence than +, -(higher precedence applied 1st)

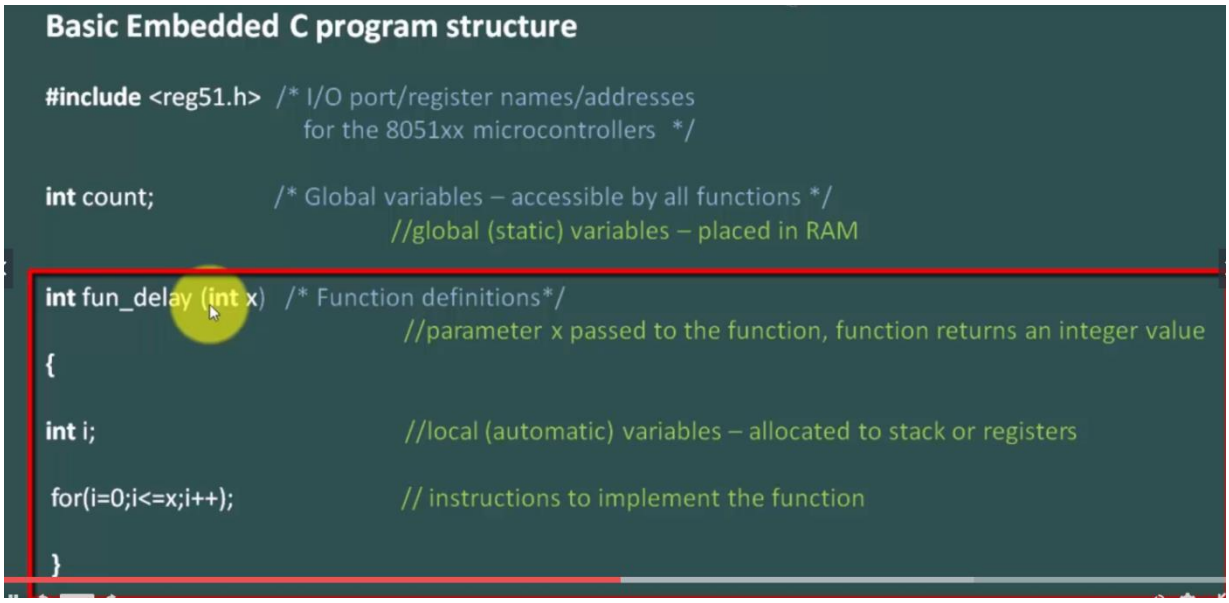
Example: j * k + m / n = (j * k) + (m / n)

Basic Embedded C program structure

```
#include <reg51.h>     /* I/O port/register names/addresses
                          for the 8051xx microcontrollers  */

int count;             /* Global variables – accessible by all functions */
                          //global (static) variables – placed in RAM

int fun_delay (int x)  /* Function definitions*/
                          //parameter x passed to the function, function returns an integer value
{

int i;                    //local (automatic) variables – allocated to stack or registers

 for(i=0;i<=x;i++);       // instructions to implement the function

 }
```

Basic Embedded C program structure

```
#include <reg51.h>  /* I/O port/register names/addresses
                       for the 8051xx microcontrollers  */

int count;          /* Global variables – accessible by all functions */
                       //global (static) variables – placed in RAM

int fun_delay (int x)   /* Function definitions*/
                       //parameter x passed to the function, function returns an integer value
{

int i;                 //local (automatic) variables – allocated to stack or registers

 for(i=0;i<=x;i++);    // instructions to implement the function

 }
```



```
void main(void)   /* Main program */
{

int k;                              //local (automatic) variable (stack or registers)
P1=0x00;          /* Initialization section */ // instructions to initialize
k = 10;                             //variables, I/O ports, devices, function registers

while (1)         /* Endless loop */
                                    //Can also use: for(;;)
{                 /* repeat forever */
P1=0x0FF;
Fun_delay( k );                     // function call
P1=0x00;
Fun_delay( k );                     // instructions to be repeated
}

}
```

Bit level Operations in C

1. Bitwise OR operator denoted by '**|**'
2. Bitwise AND operator denoted by '**&**'
3. Bitwise Complement or Negation Operator denoted by '**~**'
4. Bitwise Right Shift & Left Shift denoted by '**>>**' and '**<<**' respectively
5. Bitwise XOR operator denoted by '**^**'

AND Truth Table

Inputs		Output
A	B	Y = A.B
0	0	0
0	1	0
1	0	0
1	1	1

```
C = A & B;      A   0 1 1 0 0 1 1 0
(AND)           B   1 0 1 1 0 0 1 1
                C   0 0 1 0 0 0 1 0
```

```
unsigned char A,B,C;   //we can declare an 8-bit number as a char
A = 0x66;                // binary A =  01100110;
B = 0xB3;                // binary  B =   10110011;
C = A & B;              // binary C =   00100010; i.e 0x22;
```

OR Truth Table

Inputs		Output
A	B	Y = A+B
0	0	0
0	1	1
1	0	1
1	1	1

```
C = A | B;        A   0 1 1 0 0 1 0 0
(OR)              B   0 0 0 1 0 0 0 0
                  C   0 1 1 1 0 1 0 0
```

```
unsigned int A,B,C;
A = 0x64;              //binary A = 01100100
B = 0x10;              //binary B = 00010000
C = A| B;             // C=0x74 which is binary 01110100
```

XOR Truth Table

Inputs		Output
A	B	Y = $A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

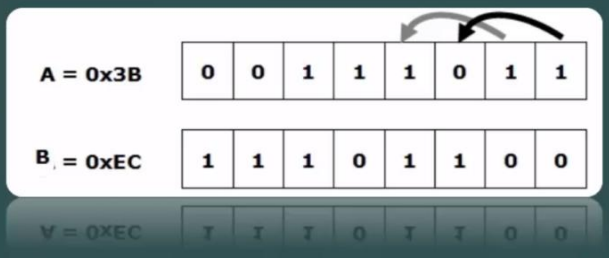
```
C = A ^ B;        A   0 1 1 0 0 1 0 0
(XOR)             B   1 0 1 1 0 0 1 1
                  C   1 1 0 1 0 1 1 1
```

```
unsigned int A,B,C;
A = 0x64;              //binary A = 01100100
B = 0xB3;              //binary B = 10110011
C = A^B;              // C = 0xD7 which is binary 11010111
```

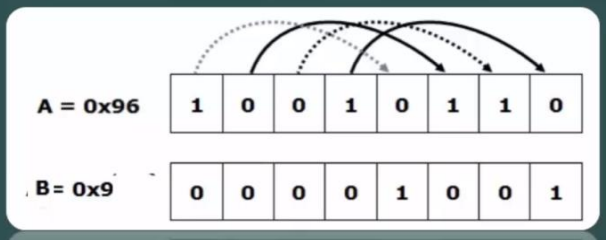
```
B = ~A;           A   0 1 1 0 0 1 0 0
(COMPLEMENT)      B   1 0 0 1 1 0 1 1
```

```
unsigned int A,B;
A= 0x64;               //binary A = 0b01100100
B = ~ A;              //  B= 0x9B which is binary 0b10011011
```

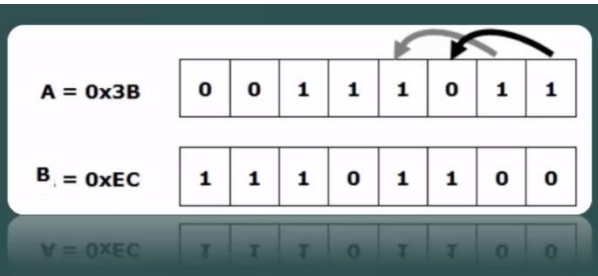
B = A << 2; // left shift A by 2



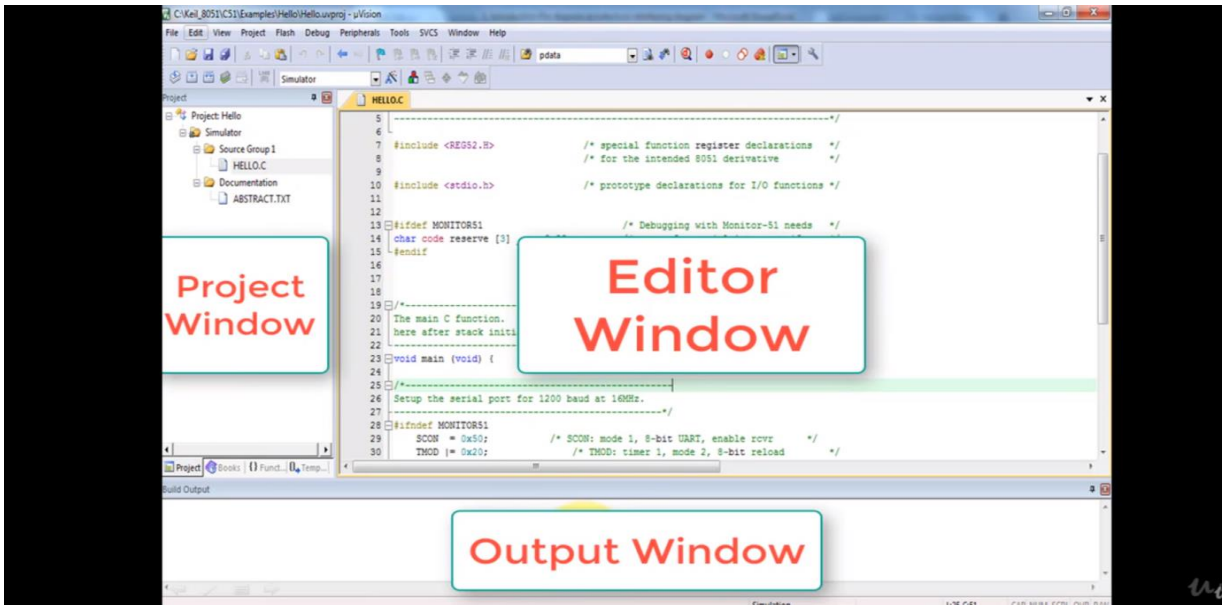
B = A >> 4; // right shift B by 4



B = A << 2; // left shift A by 2



B = A >> 4; // right shift B by 4


Project Window
Editor Window
Output Window