Objectives

• To introduce the inline assembly feature within C

• Reading
  – Irvine Chapter 12: High Level Language Interface

1. Explain the meaning of the following concepts
   – Inline assembly
   – usage of registers BP, SP, SS and DS registers in C
   – assumption about register contents upon entry to asm
   – differences between C and assembler expressions
   – assembler expression types and operators
   – entry & exit codes (also known as prolog/epilog in C)
   – Assembler vs HLL procedures and functions

2. Know how to perform the following operations:
   – defining and using variables within assembly language
   – using labels within an assembly program
   – incorporating assembly language instructions using the ASM statement into C programs
Combining assembly language and C programs

- Assembly language is the fastest language available, but the same assembly language program cannot run on different computer architectures.
- Eg, 80x86 assembly language program cannot run on Motorola 68000 series computers since the mnemonics (opcodes), register names, and size are different.
- The language C is portable but not as fast as assembly language.

Combining C and assembly language can take advantage of C’s portability and assembly’s speed.

- It is common to see a software project written using 70 to 80% C and the rest assembly language.
- There are two ways to mix C with assembly language:

1. By inserting the assembly language code in C programs, called **in-line assembly**;

2. By calling an external assembly language procedure from a C program.

Inline Assembly Code with Visual C++

- In Visual C++, the `__asm` directive is used to indicate the beginning of a block of assembly language statements (**asm block**).
- Note there are TWO underline characters before “asm”
The syntax of in-line assembly statement is:

```
asm opcode operands ; or newline
```

where
- `opcode` is a valid 80x86 assembly instruction
- `operands` contain the operand(s) acceptable to the `opcode`, and can reference C constants, variables, and labels
- a semicolon or a newline is used to signal the end of the asm statement.

Each line of in-line code is prefixed by the keyword `asm`.
A new `asm` statement can be placed on the same line, following a semicolon, but no `asm` statement can continue to the next line.

A block of in-line code (i.e., multiple `asm` statements) can be prefixed by the keyword `asm` by placing `asm` at the beginning of the block and then surrounding the statements with braces (`{ }`).

The initial brace must be on the same line as the `asm` keyword. Placing it on the following line will generate a syntax error.

**Inline Assembly Syntax**

A.) `__asm` statement

B.) `__asm` statement 1; statement 2; ...

C.) `__asm` {
    statement 1
    statement 2
    ............
}
• Any comments must be in the correct form for C (e.g., /* ... */ or //...).
• Semicolons are not used to start comments (unlike stand-alone assembly programs).

Features of MSVC inline assembly

• Intel Instruction Set only
• Use registers as operands
• Reference code labels and variables that were declared OUTSIDE the asm block
• Use PTR operator
• Use EVEN and ALIGN directives

Limitations

• Cannot use data definition directives such as DB and DW
• Cannot use assembler operators
• Cannot use STRUCT, RECORD, WIDTH, and MASK
• Cannot use macro operators (<>, !, &, %)

Example

/* Purpose: to input two integers and determine their sum. Uses MSVC++ in-line assembly

Note that this example was compiled as Win32 Console application. The unsigned integer data type is 32 bits

*/

#include <stdio.h>
```c
int main() {
    int n1=0, n2=0, sum=0;
    printf("This program determines the sum of two numbers: \n");
    printf("Please enter 2 numbers:\n");
    scanf("%d%d", &n1, &n2);

    __asm mov eax, n1; // in-line code in MSVC
    __asm add eax, n2; // semicolon is optional
    __asm mov sum, eax;

    printf("The sum of %d and %d is %d
", n1, n2, sum);
    return 0;
}
```

This program determines the sum of two numbers:
Please enter 2 numbers:
19 28

The sum of 19 and 28 is 47

This program determines the sum of two numbers:
Please enter 2 numbers:
-9 80

The sum of -9 and 80 is 71

---

**Register Usage and Preservation**

- It is free to modify EAX, EBX, ECX, and EDX in your inline code
- However, modifying too many registers makes it difficult for code optimisation because optimisation requires the use of registers

**OFFSET Directive**

- MSVC does not support OFFSET operator to retrieve the offset of a variable (i.e. the address of the variable).
- Instead, you can use the LEA instruction.
  e.g. to move the offset of `buffer` to ESI

```asm
asm LEA ESI, buffer
```
**Length, Type, and Size**

- You are allowed to use the LENGTH, TYPE, and SIZE operators in your inline code.
- The LENGTH operator counts the number of elements in an array.
- The TYPE operator returns either:
  - The number of bytes for a scalar variable or structure
  - The size of a single array element for an array

```c
typedef struct {
    long originZip; // 4
    long destinationZip; //4
    float shippingPrice; //4
} Package;
```

```c
char myChar; //1
short myShort; //2
int myInt; //4
long myLong; //4
float myFloat; //4
double myDouble; //8
Package myPackage; //12
```

```c
long double myLongDouble; //8
long myLongArray[10]; //4
int _length, _type, _size;
```

- The SIZE operator returns LENGTH * TYPE.
- Note that the function of the SIZE operator is similar to that of the SIZEOF operator in C. However, you cannot use the SIZEOF operator in any MSVC inline code.
_asm mov eax, type myInt;  
_asm mov _type, eax;  
printf("The number of bytes used by myInt is %d\n", _type);  

_asm mov eax, type myLong;  
_asm mov _type, eax;  
printf("The number of bytes used by myLong is %d\n", _type);  

_asm mov eax, type myFloat;  
_asm mov _type, eax;  
printf("The number of bytes used by myFloat is %d\n", _type);  

_asm mov eax, type myDouble;  
_asm mov _type, eax;  
printf("The number of bytes used by myDouble is %d\n", _type);  

_asm mov eax, type myLongDouble;  
_asm mov _type, eax;  
printf("The number of bytes used by myLongDouble is %d\n", _type);  

_asm mov eax, type myPackage;  
_asm mov _type, eax;  
printf("The number of bytes used by myPackage is %d\n", _type);  

_asm mov eax, type myLongArray;  
_asm mov _type, eax;  
printf("The number of bytes used by an element in myLongArray is %d\n", _type);  

_asm mov eax, size myLong;  
_asm mov _size, eax;  
printf("The size of myLong is %d\n", _size);  

_asm mov eax, size myPackage;  
_asm mov _size, eax;  
printf("The size of myPackage is %d\n", _size);  

_asm mov eax, size myLongArray;  
_asm mov _size, eax;  
printf("The size of myLongArray is %d\n", _size);  

printf("The length of myInt is 1\n");  
printf("The length of myLongArray is 10\n");  

printf("The number of bytes used by myChar is 1\n");  
printf("The number of bytes used by myShort is 2\n");  
printf("The number of bytes used by myInt is 4\n");  
printf("The number of bytes used by myLong is 4\n");  
printf("The number of bytes used by myFloat is 4\n");  
printf("The number of bytes used by myDouble is 8\n");  
printf("The number of bytes used by myLongDouble is 8\n");  
printf("The number of bytes used by myPackage is 12\n");  
printf("The number of bytes used by an element in myLongArray is 4\n");  

printf("The size of myLong is 4\n");  
printf("The size of myPackage is 12\n");  
printf("The size of myLongArray is 40\n");  

printf("The size of myLong is 4\n");  
printf("The size of myPackage is 12\n");  
printf("The size of myLongArray is 40\n");
Example 2: Procedure Call

/* File: find_max_asm.c
Purpose: To implement a procedure to return the largest number of two input integers using inline assembly
*/

#include <stdio.h>

long findMax(long, long); // procedure prototype

int main() {
    long in1, in2, max;
    printf("This program returns the max of two numbers: \n");
    printf("Please enter 2 numbers: \n");
    scanf("%d%d", &in1, &in2);
    max = findMax(in1, in2); // call procedure
    printf("The max number is %d\n", max);
    return 0;
}

/* This function returns the max integer. The function is to demonstrate the use of inline assembly for accessing function parameters and local variable */
long findMax(long num1, long num2) {
    long max;
    __asm {
        mov eax, num1;               // Load parameters into EAX
        mov ebx, num2;               // and EBX registers. Then
        // we perform the comparison
        cmp eax, ebx; // using CMP opcode.
        jna L1; // If EAX is smaller than EBX,
        mov dword ptr max, eax;  // jump to L1 label, where
        jmp L2; // it stores the content of EBX
        L1:
        mov dword ptr max, ebx; // Otherwise, store the content
        L2:
        // to the local variable max;
    }
    return max; // return the max integer
}

Passing information (parameters) to a subroutine from the calling routine

- MSVC uses the stack for passing information to the calling routine
- Normally, the first parameter is stored at location [EBP+8] and the first local variable is stored at location [EBP-4]
```c
long findMax(long num1, long num2) {
    long max;

    __asm {
        mov eax, dword ptr [ebp+8]; // parameter num1
        mov ebx, dword ptr [ebp+0ch]; // parameter num2
        cmp eax, ebx;
        jna L1;
        mov dword ptr [ebp-4], eax; // local variable max
        jmp L2;
    
    L1:
        mov dword ptr [ebp-4], ebx; // local variable max
    
    L2:
    }

    return max;
}
```

**Function Return Values**

- In MSVC, any scalar value to be returned to the calling routine is mainly kept under the EAX register:
  - AL register: `char`;
  - AX register: `short int`;
  - EAX register: `int` and `long int`, pointers;
- **Float, double, and long double** values are pushed on the floating-point stack
- For larger data structures, a pointer to the static location of the data is returned in EAX

**Borland C++ Data Types in 16-bit Applications**

<table>
<thead>
<tr>
<th>Type</th>
<th>Storage Bytes</th>
<th>ASM Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>char, unsigned char</td>
<td>1</td>
<td>byte</td>
</tr>
<tr>
<td>int, unsigned int, short int</td>
<td>2</td>
<td>word</td>
</tr>
<tr>
<td>long, unsigned long</td>
<td>4</td>
<td>dword</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
<td>dword</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
<td>qword</td>
</tr>
<tr>
<td>long double</td>
<td>10</td>
<td>tbyte</td>
</tr>
</tbody>
</table>
Example

/* File: test0-asm.c I CT106 test program
   Purpose: To input two integers and determine
   their sum. Uses in-line assembly.
   NOTE: Unless specified otherwise, for all examples using in-
   line assembly on BC++ V5, use
either
   Target Type: EasyWin ; Target Model: Small
or,
   Target Type: Application; Platform: DOS (standard)
   Target Model: Small. Then use TD to debug.
*/

#include <stdio.h>
main ( )
{
    int n1, n2, sum;
    printf("This program determines the sum of two numbers.\n");
    printf("Please enter 2 numbers: \n");
    scanf("%d%d", &n1, &n2);
    asm MOV   AX, n1; // in-line code
    asm ADD   AX, n2; // semicolon is optional
    asm MOV   sum, AX
    printf("The sum of %d and %d is %d\n", n1, n2, sum);
} /* end main */

/* Output:
This program determines the sum of two numbers.
Please enter 2 numbers:
30   45
The sum of 30 and 45 is 75
*/

Example 1 (Borland C):

/* Example to demonstrate the use of inline assembly
   - each assembly statement is prefixed by asm
*/
#include <stdio.h>
main ( )
{
    int n1, n2, sum;
    printf("This program makes the system beep\n");
    scanf("%d%d", &n1, &n2);
    asm MOV   AX, n1; // in-line code
    asm ADD   AX, n2; // semicolon is optional
    asm MOV   sum, AX
    printf("The sum of %d and %d is %d\n", n1, n2, sum);
} /* end main */
Example 2:

/* Example to demonstrate the use of inline assembly
 - grouping several assembly statements into one block */
#include <stdio.h>
main ()
{
    printf("This program makes the system beep\n");
    asm {
        mov ax, 0e07h
        xor bx, bx
        int 10h // makes the system beep
    }
    printf ("End of program\n");
}

• Note that the above method (Example 2) of writing in-line code can cause problems if one or more of the statements are labeled.
• It is best simply to have each assembly instruction prefixed with the ASM keyword as in Example 1 above

Example 3:

/* Example to demonstrate the use of inline assembly
 - displays a message a certain no. of times,
   uses the CX register for counter */
#include <stdio.h>
main ()
{
    unsigned int counter = 5; // word size data
    asm MOV CX, counter // CX is counter
    AGAIN:
    asm PUSH CX // save the counter
    printf("This is a test!\n");
    asm POP CX // restore the counter
    asm DEC CX // decrement the counter
    asm JNZ AGAIN
}

/* OUTPUT
This is a test!
This is a test!
This is a test!
This is a test!
This is a test! */
Example 4:

/* Example to demonstrate the use of inline assembly
   - displays a message a certain no. of times, starting at a
     specified cursor position;
   - should be compiled as DOS application because it uses the
     DOS function INT 10H
*/
#include <stdio.h>
main()
{
    unsigned char row = 10;       // byte size data
    unsigned char column = 20;    // byte size data
    unsigned int counter = 5;      // word size data
    asm mov cx, counter // cx is counter
    AGAIN:
    asm mov ah, 2 // ah = 02 of INT 10H to
    // set cursor
    asm mov bh, 0 // page 0
    asm mov dh, row // load the row value
    asm mov dl, column // load the column value
    asm int 10h // call DOS function
    // INT 10H to set cursor
    asm PUSH cx // save the counter
    printf("This is a test!\n");
    row++;
    column++;
    asm pop cx // restore the counter
    asm DEC cx // decrement the counter
    asm jnz AGAIN
}

/* OUTPUT:
   This is a test!
   This is a test!
   This is a test!
   This is a test!
   This is a test!
*/

• In example 4, the in-line code sets the
  cursor at row=10 and column=20, and
  then displays a string using a combination
  of C and assembly statements.
• Note that INT 10H is a DOS function – so
  the above program needs to be compiled
  as a DOS application (i.e., select
  platform:DOS).
Register Usage and Preservation 2

- One must be extremely careful inter-mixing HLL and assembler. Nothing in general can then be assumed regarding the register contents.
- If in doubt, you can always go into the debugger and look at the disassembled code and/or single step to check where the registers are being modified.

Keyboard Input and Screen Output

- When using Windows C, it is best to make use of the ‘normal’ I/O facilities.
- The first example uses a fairly simple method, whereby something is done in assembler that directly changes the contents of some variables, but the actual input and output is still done directly by C.

Example 5:

```c
/*Something Example
  Demonstration of how to read & write values using normal variables in C, and then within assembler the contents of these variables may be accessed and modified
*/
#include <stdio.h>
void near DoSomething(); // function prototype
Global variables
int Number; // These provide interface
char Letter; // between C & assembly language
```
void main()
{
    printf("Input a number: ");
    scanf("%d", &Number);
    printf("Input a character: ");
    scanf("%c", &Letter);
    DoSomething();
    printf("Number added to itself: %d\n", Number);
    printf("Next character after one given is: %c\n", Letter);
}

void near DoSomething()
// Get the contents of the global variables
// NUMBER & LETTER
// modify them and put them back into the
// variables
{
    asm mov ax, word ptr Number
    asm add ax, ax
    asm mov word ptr Number, ax
    asm mov al, byte ptr Letter
    asm add al, 1
    asm mov byte ptr Letter, al
}

/* OUTPUT
Input a number: 20
Input a character: A
Number added to itself: 40
Next character after one given is: B */

Gcc inline assembly (option)
• It is NOT compulsory to understand gcc-inline-assembly because gcc uses AT&T assembly syntax. The direction of the operands in AT&T syntax is opposite to that of Intel. Thus,

Op-code dst src in Intel syntax changes to Op-code src dst in AT&T syntax.
Registers and Immediate operand

- AT&T immediate operands are preceded by 'S' and AT&T Register names are prefixed by %
- E.g.

\[
\begin{align*}
\text{mov} & \quad %ax, %bx; \\
\text{mov} & \quad %al, %ah; \\
\text{mov} & \quad %bx, $10; \\
\text{mov} & \quad %al, $0x2A; \quad \text{// hexadecimal constant}
\end{align*}
\]

Operand Size

- In AT&T syntax the size of memory operands is determined from the last character of the op-code name. Op-code suffixes of 'b', 'w', and 'l' specify byte(8-bit), word(16-bit), and long(32-bit) memory references.
- Intel syntax accomplishes this by prefixing memory operands (not the op-codes) with 'byte ptr', 'word ptr', and 'dword ptr'.

Example (GCC-cygwin)

```c
/* Purpose: to input two integers and determine their sum. Uses gcc in-line assembly */
#include <stdio.h>

int n1, n2, sum;
```
int main() {
    printf("This program determines the sum of two numbers: \n");
    printf("Please enter 2 numbers:\n");
    scanf("%d%d", &n1, &n2);

    asm ("mov _n1, %ax"); // in-line code in cygwin
    asm ("add _n2, %ax"); // note that gcc uses AT&T syntax
    asm ("mov %ax, _sum"); // i.e. inst source, destination

    printf("The sum of %d and %d is %d\n", n1, n2, sum);
    return 0;
}

DOS Service Functions

- MS-DOS provides a number of functions for displaying text on the console or getting input from the keyboard. They are all part of a group called INT 21h MS-DOS Function Calls

Common Output Functions

- **INT 21h Function 2**
  - Description: to write a single character to standard output and advance the cursor one column forward
  - Receives: AH = 2
    DL = character value
  - No return

- **INT 21h Function 5**
  - Description: to write a single character to the printer
  - Receives: AH = 5
    DL = character value
  - No return
• **INT 21h Function 6**  
  Description: to write a character to standard output  
  Receives: AH = 6  
  DL = character value  
  No return  
  Notes: Unlike other 21h functions, this one does not filter ASCII control characters

• **INT 21h Function 9**  
  Description: to write a $-terminated string to standard output  
  Receives: AH = 9  
  DS:DX = segment/offset of the string  
  No return

• E.g.  
  .data  
  string BYTE “Hello, World!$”  
  .code  
  mov ah, 9  
  mov dx, OFFSET string  
  int 21h

• **INT 21h Function 40h**  
  Description: to write an array of bytes to a file or device  
  Receives: AH = 40h  
  BX = file or device handle (console=1)  
  CX = number of bytes to write  
  DS:DX = address of array  
  Return: AX = number of bytes written
• E.g.
  .data
  message db “Hello, world!”
  .code
  mov ah, 40h
  mov bx, 1
  mov cx, LENGTHOF message
  mov dx, OFFSET message
  int 21h

Common Input Functions

• **INT 21h Function 1**
  • Description: to read a single character from standard input
  • Receives: AH = 1
  • Return: AL = character (ASCII code)

• **INT 21h Function 6**
  • Description: to read a character from standard input without waiting
  • Receives: AH = 6
  • DL = FFh
  • Return: If ZF = 0, AL contains the character’s ASCII code

• E.g.
  mov ah, 6
  mov dl, 0FFH
  int 21h
  jz skip
  mov char, AL
  skip:
Notes

• With function 1, the program waits, if no character is present in the input buffer. This function echoes the character to standard output
• Function 6 only returns a character if one is already waiting in the input buffer. It does not echo the character

• \textbf{INT 21h Function 0Ah}
  • Description: to read an array of buffered characters from standard input
  • Receives: \texttt{AH = 0Ah}
  • Return: The structure is initialised with the input characters

• \textbf{INT 21h Function 0Bh}
  • Description: to get the status of the standard input buffer
  • Receives: \texttt{AH = 0Bh}
  • Return: If a character is waiting, \texttt{AL = 0FFh}; otherwise \texttt{AL = 0}

\begin{verbatim}
.data
kybdData KEYBOARD <>
.code
  mov  ah, 0Ah
  mov  dx, OFFSET kybdData
  int  21h
\end{verbatim}