

Microprocessor Programming

Dr. Tin Ni Ni Kyaw

Ph. D (Kumamoto University, Japan)

Associate Professor

**Department of Computer Engineering and
Information Technology**

Yangon Technological University

Yangon, Myanmar

Microprocessor Programming

Lecture 6

Iteration Structures

Contents

- Introduction
- Pre-Test Loop Structure
- Post-Test Loop Structures
- Fixed-Iteration Loop Structures
- Loops and I/O
- Nested Loops
- Complete Program
- Summary
- Practical Works
- Assignments

Introduction

- There are many different types of iteration structures in a high-level programming language.
- Similarly, assembly language also has the corresponding iteration structures, such as the **pre-test, post-test, and fixed-iteration loop structures**.
- Depending on the circumstances, we should use the best structure for the task.

Pre-Test Loop Structure

- The count-controlled pre-test while loop is the most versatile loop.
- A number of tasks can be performed in the body of the loop.
- The basic structure of this loop can be found in C code segment below:

```
i=1;  
while(i<=3){  
    i++;  
}
```

Pre-Test Loop Structure (Cont.)

- In MASM, `.while` and `.endw` directives can simplify the implementation of the while loop structure as shown below:

```
mov i,1  
  
.while i<=3  
    inc i  
  
.endw
```

- For any number of statements in the body of the loop, the structure must end with the `.endw` directive.

Pre-Test Loop Structure (Cont.)

- In another way, the while structure can also be implemented by using **compare statement and jump statements**.

```
                                mov i,1
while01:                       cmp i,3
                                jg endw01
                                inc i
                                jmp while01
endw01:                         nop
```

- Notice the inclusion of the unconditional **jmp while01** at the bottom of the loop, because without it, the loop would execute the body of the loop only once.

Pre-Test Loop Structure (Cont.)

- The following C code is an example of the implementation of while loop structure together with if-then structure.

```
ans=0;
if(x!=0) {
    i=1;
    while(i<=y) {
        ans=ans+x;
        i++;
    }
}
```

Pre-Test Loop Structure (Cont.)

- The equivalent MASM code segment can be written as follow:

```
mov ans,0           ; initialize ans to 0
.if x != 0
mov i,1             ; initialize i to 1
mov eax,y           ; load eax with y for while
.while i<= eax
mov eax,ans         ; load eax with ans
add eax,x           ; add eax to ans
mov ans,eax         ; store eax in ans
mov eax,y           ; reload eax with y for while
inc i               ; increment i by 1
.endw
.endif
```

Pre-Test Loop Structure (Cont.)

- In the program, `i` is a loop control variable.
- So, alternatively, an `ecx register` which is a counter register can be used instead of `i`.
- Then, the value in `ecx` can simply be moved into the variable `i` at the end of the segment to get the final value.

Pre-Test Loop Structure (Cont.)

- The equivalent MASM code segment **using ecx register** is as follow:

```
mov ans,0           ; initialize ans to 0
.if x!= 0
mov ecx,1           ; initialize ecx to 1
.while ecx<=y
mov eax,ans         ; load eax with ans
add eax,x           ; add x to ans
mov ans,eax         ; store eax in ans
inc ecx             ; increment ecx by one
.endw
mov i,ecx           ; store ecx in i
.endif
```

Pre-Test Loop Structure (Cont.)

- To compare the three coding styles: the C code, MASM codes using high-level directives and without using high-level directives for pre-test loop structure, we can see as follows:

```
i=1;
while(i<=3){
    i++;
}
```

```
mov i,1
while i<=3
    inc i
endw
```

```
mov i,1
while01: cmp i,3
        jg endw01
        inc i
        jmp while01
endw01: nop
```

Post-Test Loop Structures

- In C programming language, it has a post-test loop structure called the `do-while`.
- The unique feature of post-test loops is that the body of the loop is executed at least one time.
- In MASM, the post-test loop structure is implemented by using the `.repeat` and `.until` directives.

Post-Test Loop Structures (Cont.)

- The C code for do-while loop is given on the left and the corresponding assembly language appears on the right:

```
i=1;  
do {  
    i++;  
} while (i<=3);
```

```
mov i,1  
.repeat  
inc i  
.until i>3
```

- Note that instead of $i \leq 3$, the `.until` has $i > 3$ in MASM.
- The relation of the jump is **reversed** from the one in the do-while.

Post-Test Loop Structures (Cont.)

- The implementation of this loop without MASM high-level directives using compares, jumps, and labels is shown below.
- The relation of the jump is the **same** as the one in the do-while loop in C code.

```
mov i,1
repeat01:  nop
           inc i
           cmp i,3
           jle repeat01
endrpt01:  nop
```

Post-Test Loop Structures (Cont.)

- For the next example, the C code and the equivalent MASM code segment are as follows:

```
ans=0;
if (y!=0) {
    i=1;
    do {
        ans=ans+x;
        i++;
    } while (i<=y);
}
```

```
mov ans,0
.if y!=0
mov ecx,1
.repeat
mov eax,ans
add eax,x
mov ans,eax
inc ecx
.until ecx>y
mov i,ecx
.endif
```

Post-Test Loop Structures (Cont.)

- To compare the three coding styles: the C code, MASM codes using high-level directives and without using high-level directives for post-test loop structure, we can see as follows:

```
i=1;  
do {  
    i++;  
} while (i<=3);
```

```
mov i,1  
.repeat  
inc i  
.until i>3
```

```
mov i,1  
repeat01: nop  
            inc i  
            cmp i,3  
            jle repeat01  
endrpt01: nop
```

Fixed-Iteration Loop Structures

- As in many high-level languages, there usually exists a fixed-iteration loop structure often called a **for loop** structure.
- It can be used when a loop needs to loop only a **fixed** number of times.
- In C, an example of such a loop is the for loop, where the braces are optional when there is only one statement:

```
for(i=1;i<=3;i++)  
{  
    // body of loop  
}
```

Fixed-Iteration Loop Structures (Cont.)

- In MASM, the directives that can be used for this task are the `.repeat` and `.untilcxz` directives.
- The `.repeat` and `.untilcxz` directives use the `ecx` register as a counter.
- The `.untilcxz` directive performs two tasks. It
 1. `decrements` the `ecx` register by 1 and then
 2. `jumps` to the `.repeat` directive when `ecx` is not equal to 0.
- In other words, it loops until the `ecx` register equals 0 (`cxz`).

Fixed-Iteration Loop Structures (Cont.)

```
mov ecx,3  
.repeat  
; body of the loop  
.untilcxz
```

- In this MASM code segment, first, `ecx` register is loaded with the `number of times` the body of the loop should be executed.
- Then, each time the `.untilcxz` directive is executed, the value of the `ecx` register is `decremented by 1` and then `compared to 0`.
- If the value is not equal to 0, the loop `repeats`.
- If the value is 0, the flow of control is `passed` onto the instruction immediately following the `.untilcxz` directive.

Fixed-Iteration Loop Structures (Cont.)

- To implement the MASM code segment without using high-level directives, the `loop` instruction works the same as `.repeat` and `.untilcxz` directives.

```
mov ecx,3
for01:    nop
          ; body of the loop
          loop for01
endfor01: nop
```

- The `loop` instruction decrements `ecx` register by 1, branches to the label indicated in the operand field when `ecx` is not equal to 0, and falls through otherwise.

Fixed-Iteration Loop Structures (Cont.)

- To compare the three coding styles: the C code, MASM codes using high-level directives and without using high-level directives for fixed-iteration loop structures, we can see as follows:

```
for(i=1;i<=3;i++)  
{  
  // body of loop  
}
```

```
mov ecx,3  
.repeat  
; body of loop  
.untilcxz
```

```
mov ecx,3  
for01:    nop  
          ;body of loop  
          loop for01  
endfor01: nop
```

Post-Test Loop Structure (Cont.)

- As another example, the C code and the equivalent MASM code segment are as follow:

```
ans =0;  
if (y != 0)  
for(i=1; i<=y;i++)  
    ans = ans + x;
```

```
mov ans,0  
.if y != 0  
mov ecx,y  
.repeat  
mov eax,ans  
add eax, x  
mov ans,eax  
.untilcxz  
.endif
```

Loops and Input/Output

- This C code segment is for entering **exactly** 10 integers and calculating the sum of the integers and then displaying the result.

```
sum=0;
for(i=1; i<=10; i++) {
    printf("%s", "Enter an integer: ");
    scanf("%d",&num);
    sum=sum+num;
}
printf("\n%s%d\n\n", "The sum is ",sum);
return 0;
```

Loops and Input/Output (Cont.)

- The equivalent segment in assembly is shown below.

```
.data
msg1 byte "Enter an integer: ",0
msg2 byte "The sum is ",0
.code
mov sum,0
mov ecx,10
.repeat
mov temp,ecx
INVOKE printf, ADDR msg1fmt, ADDR msg1
INVOKE scanf, ADDR in1fmt, ADDR num
mov eax,sum
add eax,num
mov sum,eax
mov ecx,temp
.untilcxz
INVOKE printf, ADDR msg2fmt, ADDR msg2, sum
```

Loops and Input/Output (Cont.)

- Remember that the **INVOKE** directive can **destroy** the `eax`, `ecx`, and `edx` registers.
- Since the `.repeat-.untilcxz` directive uses the `ecx` register, it needs to save and restore its value.
- For this reason, the value of `ecx` is stored in a memory location called **temp** at the top of the loop and then the value of `ecx` is restored at the bottom of the loop.

Loops and Input/Output (Cont.)

- However, if the user wants to decide how many integers to be summed, the number of integers to be input may not be predefined in the program.
- In such cases, we need to add some statements such as a prompt for asking the user for the number of integers.
- To do this, we can add those statements before the loop.

Loops and Input/Output (Cont.)

- The following implementation includes the required actions mentioned in the previous section.

```
.data
msg0 byte "Enter the number of integers to input: ",0
msg1 byte "Enter an integer: ",0
msg2 byte "The sum is ",0
.code
mov sum,0
INVOKE printf, ADDR msg1fmt, ADDR msg0
INVOKE scanf, ADDR in1fmt, ADDR count
mov ecx,1
.while ecx<=count
mov temp,ecx
INVOKE printf, ADDR msg1fmt, ADDR msg1
INVOKE scanf, ADDR in1fmt, ADDR num
mov eax,sum
add eax,num
mov sum,eax
mov ecx,temp
inc ecx
.endw
INVOKE printf, ADDR msg2fmt, ADDR msg2, sum
```

Nested Loops

- An example for nested loops by the C code segment is below on the left and by the assembly code using high-level directives is on the right:

```
i = 1;
while(i<=2) {
    j=1;
    while(j<=3) {
        j++;
    }
    i++;
}
```

```
mov i,1
while i<=2
    mov j,1
    while j<=3
        inc j
    .endw
    inc i
.endw
```

Nested Loops (Cont.)

- The equivalent assembly code segment without using high-level directives is as follow.

```
mov i,1
while01:      cmp i,2
              jg endwhile01
              mov j,1
              while02:  cmp j,3
                      jg endwhile02
                      inc j
                      jmp while02
              endwhile02:  nop
                      inc i
                      jmp while01
              endwhile01:  nop
```

Complete Program: Implementing Power Function

- Consider the implementation of the power function (x^n), where an iterative definition of the power function is as follows:

x^n = If $x < 0$ or $n < 0$, then negative message
Else if $x = 0$ and $n = 0$, then undefined message
Else if $n = 0$, then 1
Otherwise $1 * x * x * \dots * x$ (n times)

Complete Program: Implementing Power Function

- The following C program implements the above definition:

```
#include <stdio.h>
int main() {
    int x,n,i,ans;
    printf("%s","Enter x: ");
    scanf("%d",&x);
    printf("%s","Enter n: ");
    scanf("%d",&n);
    if(x<0 || n<0)
        printf("\n%s\n\n","Error: Negative x and/or y");
    else
        if(x==0 && n==0)
            printf("\n%s\n\n","Error: Undefined answer");
        else {
            i=1;
            ans=1;
            while(i<=n) {
                ans=ans*x;
                i++;
            }
            printf("\n%s%d\n\n","The answer is: ",ans);
        }
    return 0;
}
```

Complete Program: Implementing Power Function

- This MASM program also implements the power function.
- Note that high-level directives are used in this program.

```

                                .listall
                                .386
                                .model flat,c
                                .stack 100h
scanf                             PROTO arg2:Ptr Byte, inputlist:VARARG
printf                            PROTO arg1:Ptr Byte, printlist:VARARG

                                .data
inlfmt                             byte "%d",0
msg1fmt                             byte "%s",0
msg3fmt                             byte "%s%d",0Ah,0Ah,0
errfmt                             byte "%s",0Ah,0Ah,0
errmsg1                             byte 0Ah,"Error: Negative x and/or y",0
errmsg2                             byte 0Ah,"Error: Undefined answer",0
msg1                                 byte "Enter x: ",0
msg2                                 byte "Enter n: ",0
msg3                                 byte 0Ah,"The answer is: ",0
x                                   sdword ?
n                                   sdword ?
ans                                 sdword ?
i                                   sdword ?

                                .code
main                               proc
    INVOKE printf, ADDR msg1fmt, ADDR msg1
    INVOKE scanf, ADDR inlfmt, ADDR x
    INVOKE printf, ADDR msg1fmt, ADDR msg2
    INVOKE scanf, ADDR inlfmt, ADDR n
    .if x<0 || n<0
    INVOKE printf, ADDR errfmt, ADDR errmsg1
    .else
    .if x==0 && n==0
    INVOKE printf, ADDR errfmt, ADDR errmsg2
    .else
    mov ecx,1
    mov ans,1
    .while ecx <= n
    mov eax,ans
    imul x
    mov ans,eax
    inc ecx
    .endw
    mov i,ecx
    INVOKE printf, ADDR msg3fmt, ADDR msg3, ans
    .endif
    .endif
    ret
main                               endp
                                end
```

Summary

- For pre-test **while** loop structure, **.while** and **.endw** directives can be used to implement.
- For post-test **do-while** loop structures, the **.repeat** and **.until** directives can be used.
- For fixed iteration **for** loop structures, the **.repeat** and **.untilcxz** directives can be used.
- It is important **to include the unconditional jump (jmp) or loop** at the bottom of the loop to repeat the corresponding loop.

Microprocessor Programming

Practical Works

Implementing Loops and I/O and Power Function

Implementing Loops and I/O

- The given MASM code segment is implemented as a complete program described below:

```
.stack 100h
printf PROTO arg1: Ptr Byte, printlist: VARARG
scanf PROTO arg2: Ptr Byte, inputlist: VARARG
.data
msg1fmt byte "%a", 0
msg2fmt byte "%a", "%a", 0Ah, 0Ah, 0
inifmt byte "%d", 0
msg1 byte "Enter an integer: ", 0
msg2 byte "The sum is ", 0
num sdword ?
sum sdword ?
temp sdword ?
.code
main proc
mov sum, 0
mov ecx, 10
mov sum, 0
mov ecx, 10
.repeat
mov temp, ecx
INVOKE printf, ADDR msg1fmt, ADDR msg1
INVOKE scanf, ADDR inifmt, ADDR num
mov eax, sum
add eax, num
mov sum, eax
mov ecx, temp
.untilx2
INVOKE printf, ADDR msg2fmt, ADDR msg2, sum
ret
main endp
end
```

Output

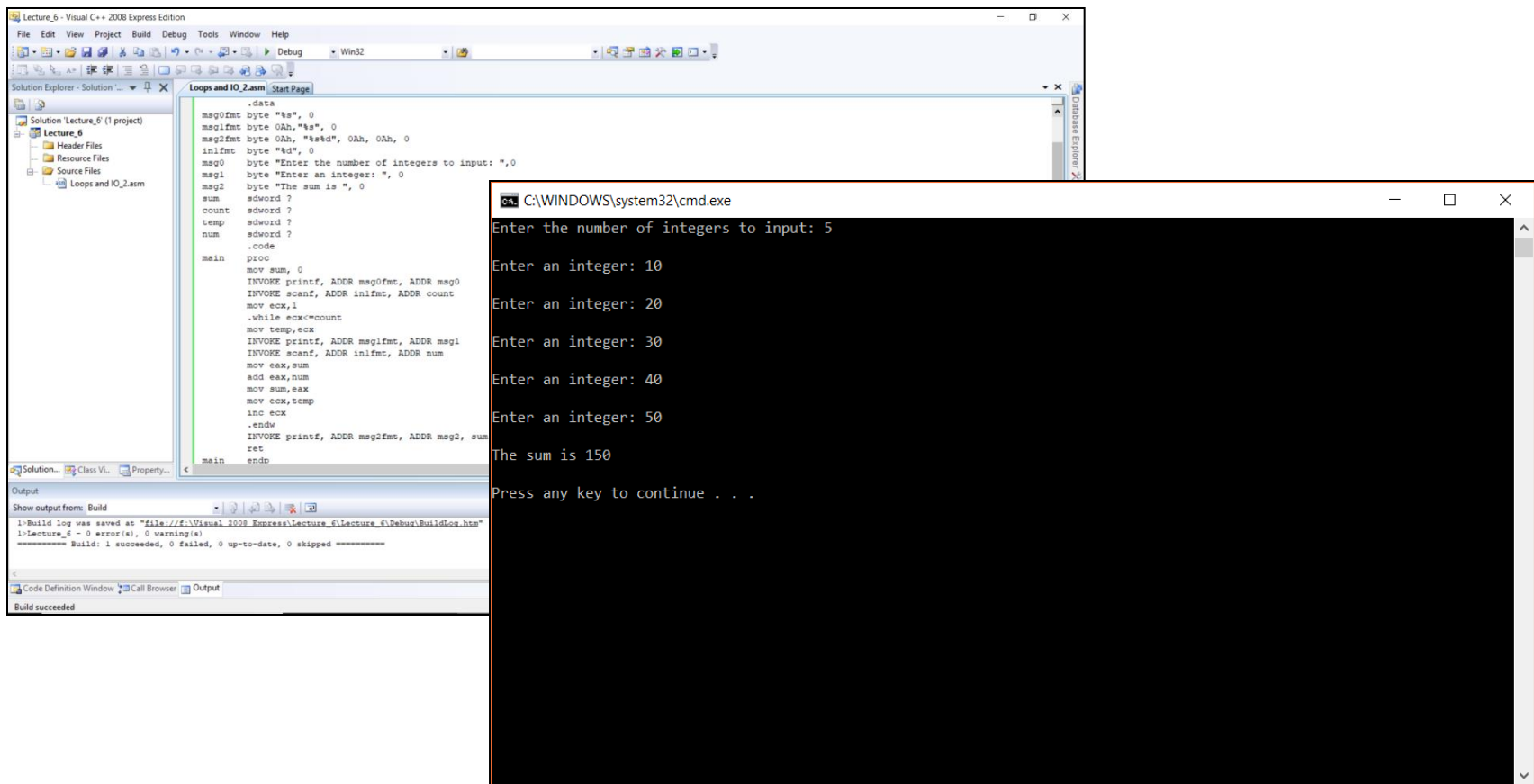
```
1>Build log was saved at "file:///f:/Visual_2008_Express/Lecture_6/Lecture_6/Debug/BuildLog.htm"
1>Lecture_6 - 0 error(s), 0 warning(s)
***** Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped *****
```

Build succeeded

```
C:\WINDOWS\system32\cmd.exe
Enter an integer: 2
Enter an integer: 4
Enter an integer: 6
Enter an integer: 8
Enter an integer: 10
Enter an integer: 12
Enter an integer: 14
Enter an integer: 16
Enter an integer: 18
Enter an integer: 20
The sum is 110
Press any key to continue . . .
```

Implementing Loops and I/O (Cont.)

- Given MASM code segment is implemented as a complete program as follow:



The screenshot displays the Visual Studio 2008 Express Edition interface. The main window shows the assembly code for a program named 'Loops and IO_2.asm'. The code includes data definitions for messages and a main procedure that uses a loop to read integers and calculate their sum.

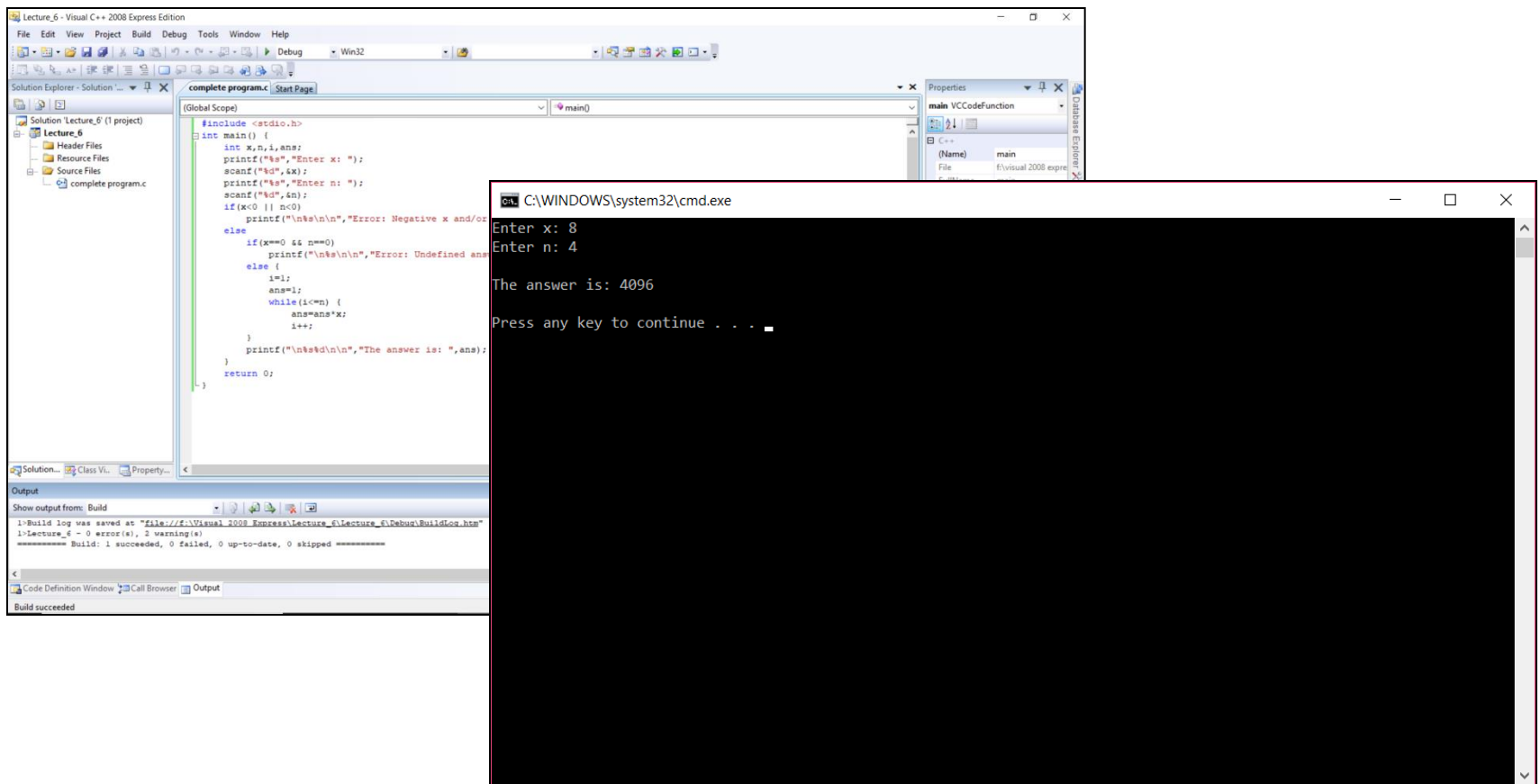
```
.data
msg0fmt byte "%s", 0
msg1fmt byte 0Ah, "%s", 0
msg2fmt byte 0Ah, "%s%d", 0Ah, 0Ah, 0
inifmt byte "%d", 0
msg0 byte "Enter the number of integers to input: ", 0
msg1 byte "Enter an integer: ", 0
msg2 byte "The sum is ", 0
sum sdword ?
count sdword ?
temp sdword ?
num sdword ?
.code
main
proc
mov sum, 0
INVOKE printf, ADDR msg0fmt, ADDR msg0
INVOKE scanf, ADDR inifmt, ADDR count
mov ecx, 1
    .while ecx <= count
    mov temp, ecx
    INVOKE printf, ADDR msg1fmt, ADDR msg1
    INVOKE scanf, ADDR inifmt, ADDR num
    mov eax, sum
    add eax, num
    mov sum, eax
    mov ecx, temp
    inc ecx
    .endw
INVOKE printf, ADDR msg2fmt, ADDR msg2, sum
ret
endp
```

The output window shows the program's execution. It prompts the user to enter the number of integers (5), then iteratively prompts for each integer (10, 20, 30, 40, 50), and finally displays the sum (150).

```
C:\WINDOWS\system32\cmd.exe
Enter the number of integers to input: 5
Enter an integer: 10
Enter an integer: 20
Enter an integer: 30
Enter an integer: 40
Enter an integer: 50
The sum is 150
Press any key to continue . . .
```

Implementing Power Function

- For a given complete C program, we implement and describe its output below.



The image shows a screenshot of Visual Studio Express Edition with a C program for calculating power. The code is as follows:

```
#include <stdio.h>
int main() {
    int x,n,i,ans;
    printf("%s","Enter x: ");
    scanf("%d",&x);
    printf("%s","Enter n: ");
    scanf("%d",&n);
    if(x<0 || n<0)
        printf("\n\n","Error: Negative x and/or n");
    else
        if(x==0 && n==0)
            printf("\n\n","Error: Undefined ans");
        else {
            ans=1;
            while(i<=n) {
                ans=ans*x;
                i++;
            }
            printf("\n\n%d\n\n","The answer is: ",ans);
        }
    return 0;
}
```

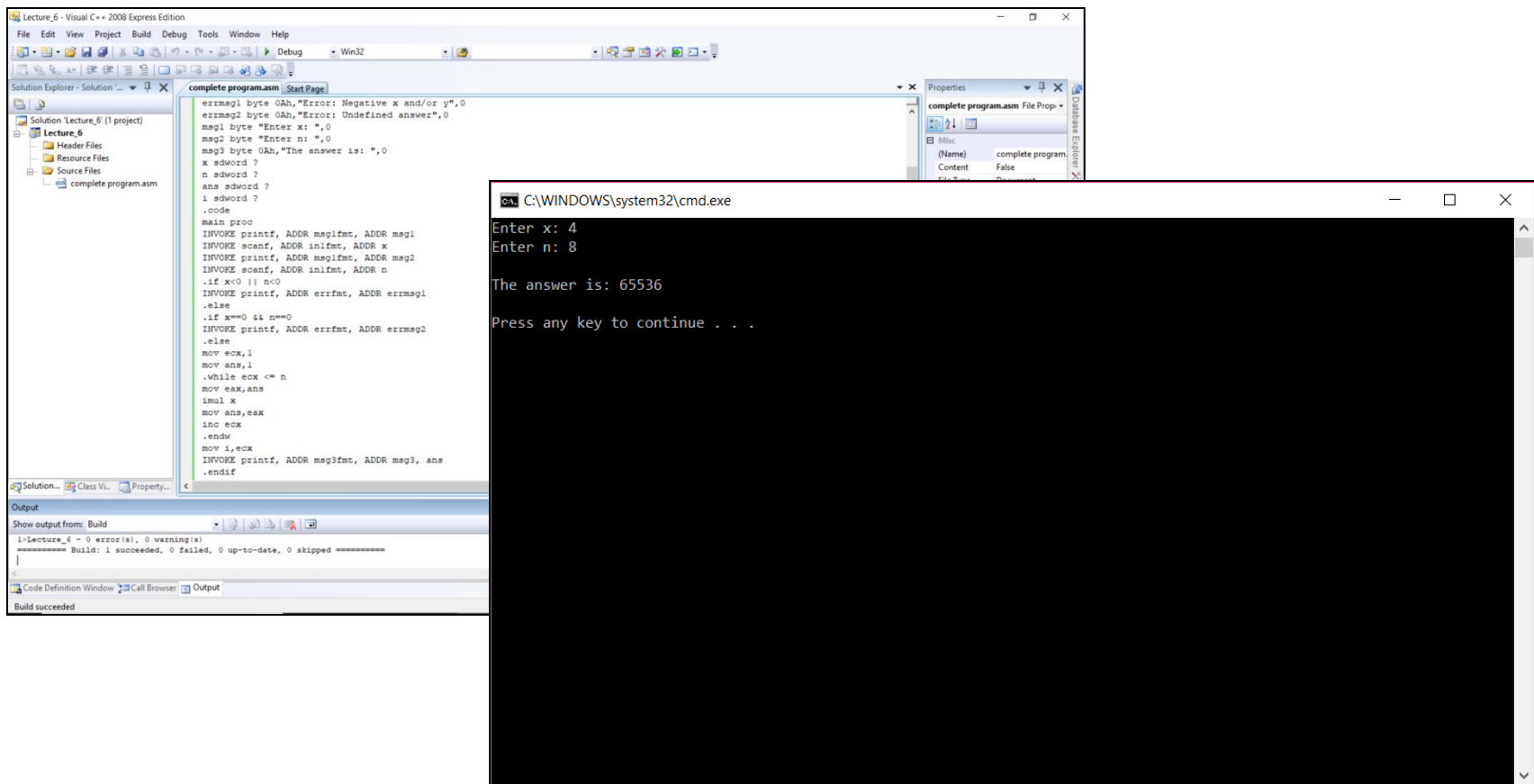
The output window shows the following execution:

```
Enter x: 8
Enter n: 4
The answer is: 4096
Press any key to continue . . .
```

The output window title is "C:\WINDOWS\system32\cmd.exe". The Visual Studio interface also shows the Solution Explorer with "Lecture_6" project, the Properties window for "main VCCodeFunction", and the Output window showing "Build succeeded".

Implementing Power Function (Cont.)

- For a given complete MASM program, we implemented as follow:



Microprocessor Programming

Practical Assignments (Instructions)

Assignment 1

- Implement a complete MASM program for the given segment with compares, jumps, and appropriate labels. Also, display its output.

```
.data
msg1 byte "Enter an integer: ",0
msg2 byte "The sum is ",0
.code
mov sum,0
mov ecx,10
.repeat
mov temp,ecx
INVOKE printf, ADDR msg1fmt, ADDR msg1
INVOKE scanf, ADDR in1fmt, ADDR num
mov eax,sum
add eax,num
mov sum,eax
mov ecx,temp
.untilcxz
INVOKE printf, ADDR msg2fmt, ADDR msg2, sum
```

Assignment 2

- Implement a complete MASM program for the given segment without using high-level directives. Also, display its output.

```
.data
msg0 byte "Enter the number of integers to input: ",0
msg1 byte "Enter an integer: ",0
msg2 byte "The sum is ",0
.code
mov sum,0
INVOKE printf, ADDR msg1fmt, ADDR msg0
INVOKE scanf, ADDR in1fmt, ADDR count
mov ecx,1
.while ecx<=count
mov temp,ecx
INVOKE printf, ADDR msg1fmt, ADDR msg1
INVOKE scanf, ADDR in1fmt, ADDR num
mov eax,sum
add eax,num
mov sum,eax
mov ecx,temp
inc ecx
.endw
INVOKE printf, ADDR msg2fmt, ADDR msg2, sum
```

Assignment 3

- Rewrite a complete MASM code without using high-level directives. Also, display its outputs.

```
.listall
.386
.model flat,c
.stack 100h
scanf      PROTO arg2:Ptr Byte, inputlist:VARARG
printf     PROTO arg1:Ptr Byte, printlist:VARARG

.data
infmt      byte "%d",0
msg1fmt    byte "%s",0
msg3fmt    byte "%s%d",0Ah,0Ah,0
errfmt     byte "%s",0Ah,0Ah,0
errmsg1    byte 0Ah,"Error: Negative x and/or y",0
errmsg2    byte 0Ah,"Error: Undefined answer",0
msg1       byte "Enter x: ",0
msg2       byte "Enter n: ",0
msg3       byte 0Ah,"The answer is: ",0
x          sdword ?
n          sdword ?
ans        sdword ?
i          sdword ?

.code
main       proc
           INVOKE printf, ADDR msg1fmt, ADDR msg1
           INVOKE scanf, ADDR infmt, ADDR x
           INVOKE printf, ADDR msg1fmt, ADDR msg2
           INVOKE scanf, ADDR infmt, ADDR n
           .if x<0 || n<0
           INVOKE printf, ADDR errfmt, ADDR errmsg1
           .else
           .if x==0 && n==0
           INVOKE printf, ADDR errfmt, ADDR errmsg2
           .else
           mov ecx,1
           mov ans,i
           .while ecx <= n
           mov eax,ans
           imul x
           mov ans,eax
           inc ecx
           .endw
           mov i,ecx
           INVOKE printf, ADDR msg3fmt, ADDR msg3, ans
           .endif
           .endif
           ret
main       endp
end
```

Microprocessor Programming

Practical Assignments (Report)

Assignment 1

- A complete assembly program without using high-level directives, with only compares, jumps and labels is implemented and its output is as follow:

The image shows a screenshot of Visual Studio 2008 Express Edition. The main window displays the assembly code for 'assignment_1.asm'. The code includes directives like .386, .model flat, c, .stack 100h, and various instructions for printing and scanning integers. A loop 'for01' is used to calculate the sum of integers from 1 to 10. The output window shows the program's execution, displaying the sum of integers from 1 to 10, which is 165.

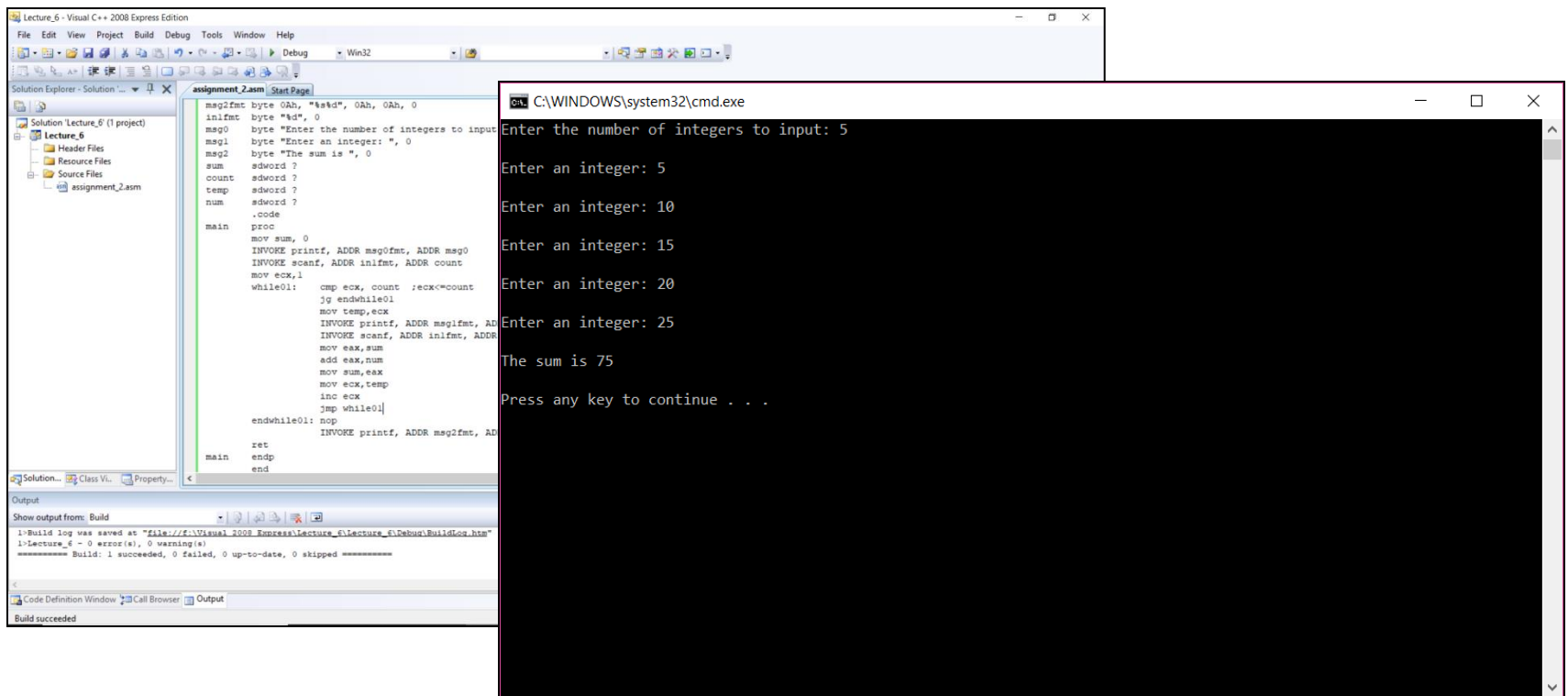
```
.386
.model flat, c
.stack 100h
printf PROTO arg1: Ptr Byte, printlist: VARARG
scanf PROTO arg2: Ptr Byte, inputlist: VARARG
.data
msg1fmt byte "%s", 0
msg2fmt byte "Ah, %kd", 0Ah, 0Ah, 0
in1fmt byte "%d", 0
msg1 byte "Enter an integer: ", 0
msg2 byte "The sum is ", 0
num sdword ?
sum sdword ?
temp sdword ?
.proc
.code
main
mov sum, 0
mov ecx, 10
for01: mov temp, ecx
        INVOKE printf, ADDR msg1fmt, ADDR in1fmt, ADDR sum
        mov eax, sum
        add eax, num
        mov sum, eax
        mov ecx, temp
        loop for01
endfor01: nop
INVOKE printf, ADDR msg2fmt, ADDR msg2, sum
ret
main
endp
end
```

```
C:\WINDOWS\system32\cmd.exe
Enter an integer: 3
Enter an integer: 6
Enter an integer: 9
Enter an integer: 12
Enter an integer: 15
Enter an integer: 18
Enter an integer: 21
Enter an integer: 24
Enter an integer: 27
Enter an integer: 30
The sum is 165
Press any key to continue . . .
```

Build succeeded

Assignment 2

- A complete assembly program without using high-level directives, with only compares, jumps and labels is implemented as follow:



The image shows a screenshot of Visual Studio 2008 Express Edition. The main window displays the assembly code for 'assignment_2.asm'. The code defines a program that prompts the user for the number of integers to input, then enters a loop where it repeatedly asks for integers and calculates their sum. The program uses only basic assembly instructions like 'mov', 'add', 'cmp', 'jg', and 'jmp', along with 'INVOKE printf' and 'INVOKE scanf' for I/O. The 'main' function is clearly marked.

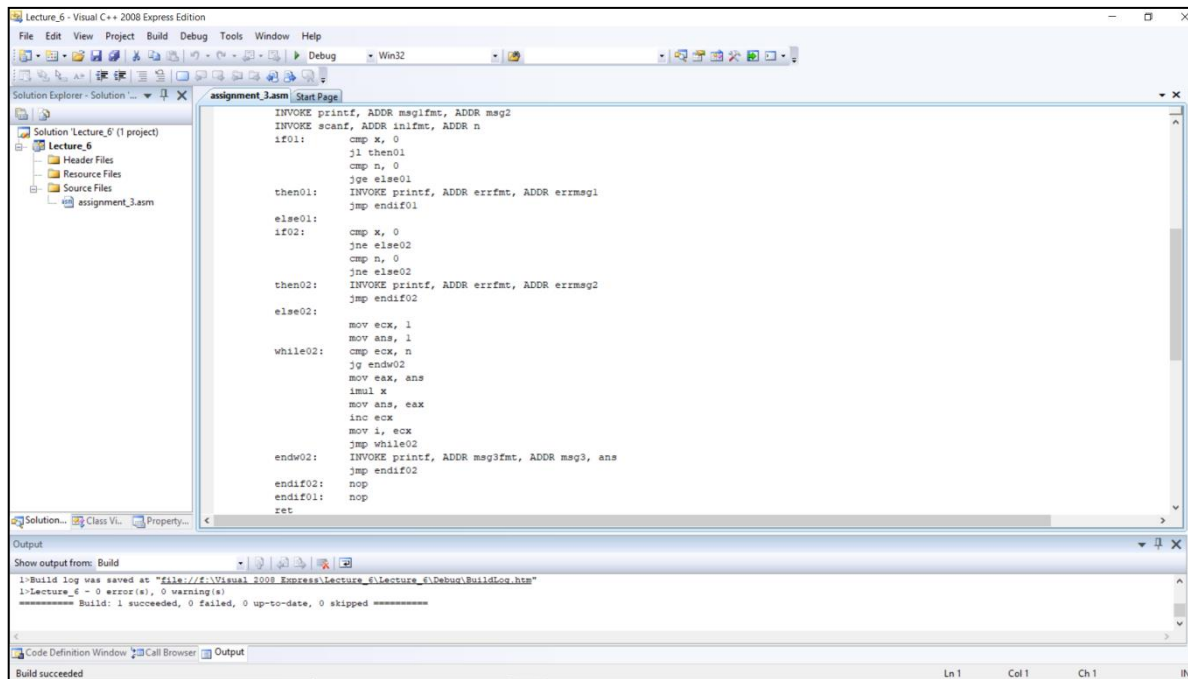
```
msg2fmt byte 0Ah, "%d", 0Ah, 0Ah, 0
in1fmt  byte "%d", 0
msg0    byte "Enter the number of integers to input: ", 0
msg1    byte "Enter an integer: ", 0
msg2    byte "The sum is ", 0
sum      dword ?
count    dword ?
temp     dword ?
num      dword ?
.code
main    proc
        mov sum, 0
        INVOKE printf, ADDR msg0fmt, ADDR msg0
        INVOKE scanf, ADDR in1fmt, ADDR count
        mov ecx, 1
        while01: cmp ecx, count ;ecx<=count
                jg  endwhile01
                mov temp, ecx
                INVOKE printf, ADDR msg1fmt, ADDR temp
                INVOKE scanf, ADDR in1fmt, ADDR num
                mov eax, sum
                add eax, num
                mov sum, eax
                mov ecx, temp
                inc ecx
                jmp while01
        endwhile01: mov
                INVOKE printf, ADDR msg2fmt, ADDR sum
        ret
main    endp
        end
```

The output window shows the execution of the program. It prompts the user to enter the number of integers (5), then enters a loop where it repeatedly asks for integers (5, 10, 15, 20, 25) and calculates their sum (75). The program ends with the message "The sum is 75" and "Press any key to continue . . .".

```
C:\WINDOWS\system32\cmd.exe
Enter the number of integers to input: 5
Enter an integer: 5
Enter an integer: 10
Enter an integer: 15
Enter an integer: 20
Enter an integer: 25
The sum is 75
Press any key to continue . . .
```

Assignment 3

- A complete assembly program without using high-level directives, with only compares, jumps and labels is implemented as follow:



```
INVOKE printf, ADDR msg1fmt, ADDR msg2
INVOKE scanf, ADDR in1fmt, ADDR n
if01:  cmp x, 0
      j1 then01
      cmp n, 0
      jge else01
then01: INVOKE printf, ADDR err1fmt, ADDR errmsg1
      jmp endif01
else01:
if02:  cmp x, 0
      jne else02
      cmp n, 0
      jne else02
then02: INVOKE printf, ADDR err2fmt, ADDR errmsg2
      jmp endif02
else02:
while02: mov ecx, 1
        mov ans, 1
        cmp ecx, n
        jg endw02
        mov eax, ans
        imul x
        mov ans, eax
        inc ecx
        mov i, ecx
        jmp while02
endw02: INVOKE printf, ADDR msg3fmt, ADDR msg3, ans
      jmp endif02
endif02: nop
endif01: nop
ret
```

Output

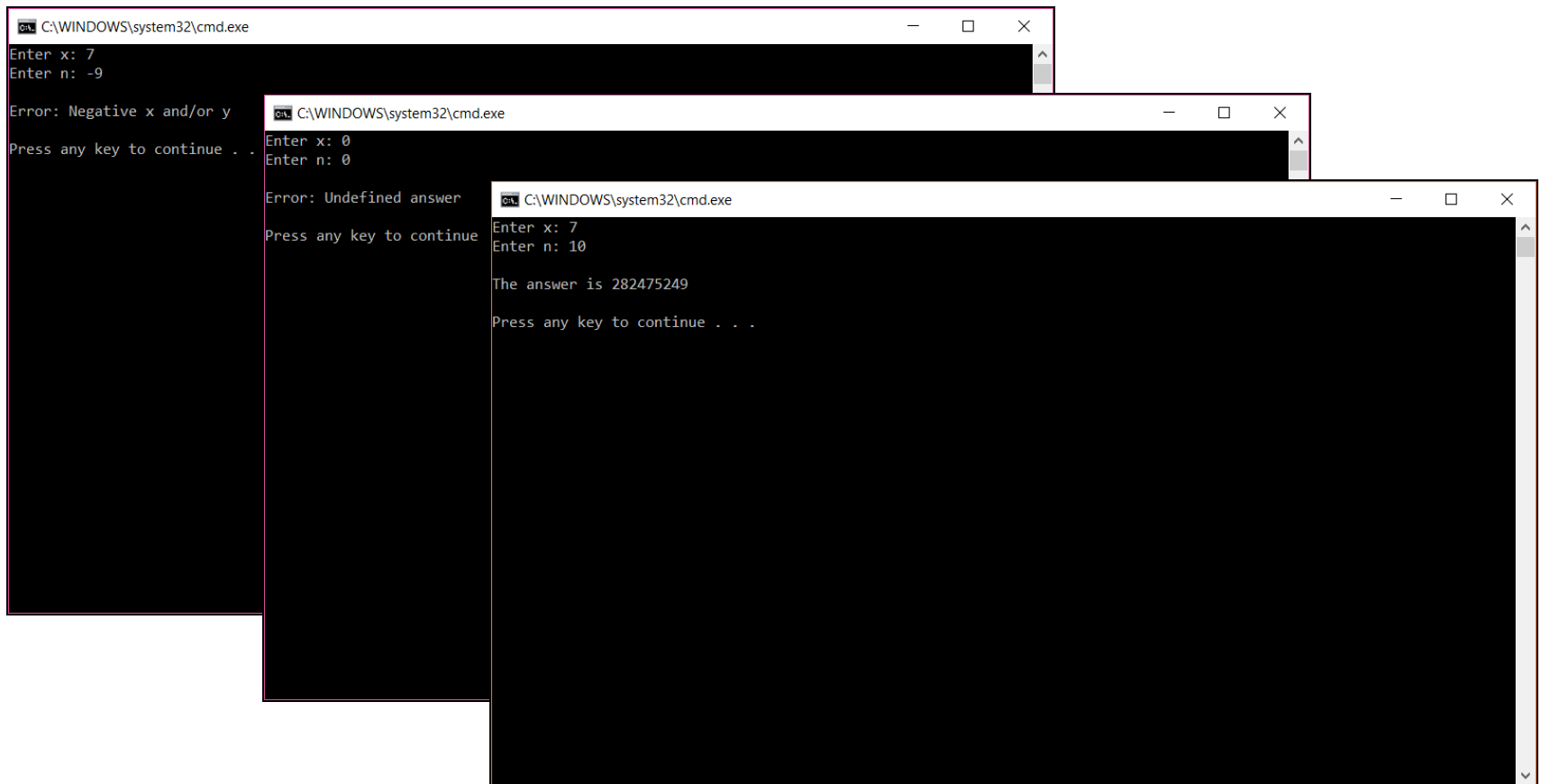
Show output from: Build

```
1-Build log was saved at "file:///F:/Visual 2008 Express/Lecture_6/Lecture_6_Debug_BuildLog.htm"
1-Lecture_6 - 0 error(s), 0 warning(s)
===== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped =====
```

Build succeeded

Assignment 3 (Cont.)

- The followings show some outputs of the implementation of power function:



```
C:\WINDOWS\system32\cmd.exe
Enter x: 7
Enter n: -9

Error: Negative x and/or y
Press any key to continue . . .

C:\WINDOWS\system32\cmd.exe
Enter x: 0
Enter n: 0

Error: Undefined answer
Press any key to continue

C:\WINDOWS\system32\cmd.exe
Enter x: 7
Enter n: 10

The answer is 282475249
Press any key to continue . . .
```

Next Lecture

- Logic Instructions
- Logical Shift Instructions
- Arithmetic Shift Instructions
- Rotate Instructions

Thank You