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

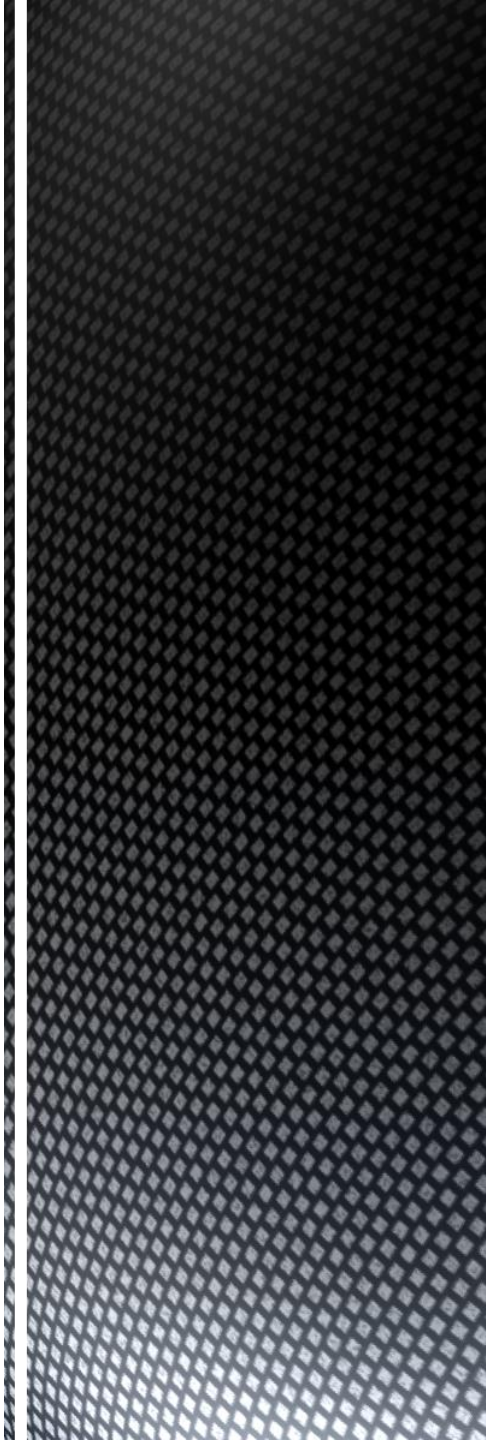
Microprocessor Programming

Lecture 9

Procedures

Contents

- Introduction
- Procedures
- Layout of Procedures
- Call Instruction
- Nested Procedure Calls
- Complete Program
- Summary
- Practical Works
- Assignments



Introduction

- A procedure is an important part of any computer system's architecture.
- It is a group of instructions that usually performs one task.
- Procedures allow programmers to save time by not having to rekey the same code over and over again.
- This saves memory space and makes it easier to develop software.

Introduction (Cont.)

- Procedures are designed to return anywhere from zero to many values.
- Although there are ways to make it possible to utilize parameters, the simplest way to communicate between a program and a procedure is to use either global variables or registers.

Procedures

- In assembly language, subprograms are called procedures.
- The actual procedure can be placed in a number of locations in the program.
- The most convenient place is after the ENDP statement in the main program and prior to the END statement.

Layout of Procedures

- With the assembler, there are specific rules for storing procedures.
- A procedure begins with the PROC directive and ends with the ENDP directive.
- Each directive appears with the name of the procedure (e.g., pname).

```
pname   proc
        .
        .
pname   endp
```

Layout of Procedures (Cont.)

- Next comes the body of the procedure.
- Then, it is followed by the return instruction (RET) as shown below.
- The procedure uses a RET (return from procedure) instruction to bring the processor back to the point in the program where the procedure was called.

```
pname   proc
        ; body of the procedure
        ret
pname   endp
```

Layout of Procedures (Cont.)

- The PROC and ENDP directives indicate to the assembler the beginning and the end of the procedure respectively.
- During the execution of the procedure, the RET instruction forces the CPU to return to the location from where the procedure was called.

Layout of Procedures (Cont.)

- In a procedure, it is better to include only one return statement to keep the program structured with only one entry point and one exit point.
- Also, it is usually best to be sure that the RET instruction is the last statement in a procedure prior to the ENDP directive.

Layout of Procedures (Cont.)

- Here is an example procedure using only one RET instruction.

```
pro1    proc
        .if eax == 0
        mov edx,1
        .else
        mov edx, 0
        .endif
        ret
pro1    endp
```

Call Instruction

- The instruction used to invoke a procedure is the CALL instruction.
- The CALL instruction has one operand that specifies the name of the procedure to be invoked.
- The CALL instruction calls a procedure by directing the processor to begin execution at a new memory location.
- Upon return from the procedure, execution will continue with the instruction after the CALL instruction.

```
call        pname
```

Call Instruction (Cont.)

- If we want to implement one algorithm in two different locations of the same program, it would be much easier to call the procedure from two different locations in the main program as follow.

```
main    proc
        .
        call mult
        .
        .
        call mult
        .
        ret
main    endp
```

Call Instruction (Cont.)

- Then, after the main program, the code for the algorithm (e.g., mult) could be written as a procedure.

```
mult    proc
        mov eax,0          ; initialize eax to 0
        .if x != 0
        mov ecx,1          ; initialize i to 1
        .while ecx<=y
        add eax,x          ; add x to eax
        inc ecx            ; increment i by 1
        .endw
        .endif
        ret
mult    endp
```

Saving and Restoring Registers

- Not only in the main program but also in the procedures, registers are often needed to be saved and restored to backup their original contents.
- The following multiplication procedure includes saving and restoring the ecx register using PUSH and POP instructions.

```
mult    proc
        push ecx        ; save ecx
        mov eax,0       ; initialize eax to 0
        .if x != 0
        mov ecx,1       ; initialize i to 1
        .while ecx<=y
        add eax,x       ; add x to eax
        inc ecx         ; increment i by 1
        .endw
        .endif
        pop ecx         ; restore ecx
        ret
mult    endp
```

Saving and Restoring Registers (Cont.)

- Although it is possible to save and restore all the registers whether they were altered or not, it does not help other programmers understand what is happening in the procedure.
- By saving and restoring only the registers that are altered, it helps others understand which registers are being altered and also helps makes the code more self-documenting.
- However, there could be a chance for a logic error if the POP instructions were accidently written in the wrong order.

Saving and Restoring Registers (Cont.)

- Luckily, there is a way to save the four general purpose registers (eax, ebx, ecx, and edx) along with the esi, edi, ebp, and esp registers with only one instruction.
- The solution is to use PUSHAD and POPAD instructions.
- If a number of 32-bit registers are modified in a procedure, use PUSHAD at the beginning of the procedure and POPAD at the end to save and restore the registers.

Saving and Restoring Registers (Cont.)

PUSHAD and POPAD Instructions

- The PUSHAD instruction pushes the contents of all of the 32-bit general-purpose registers on the stack in the following order: eax, ecx, edx, ebx, esp (value before executing PUSHAD), ebp, esi, and edi.
- The POPAD instruction pops the same registers off the stack in reverse order.

Saving and Restoring Registers (Cont.)

Example

- Consider a procedure that outputs blank lines a variable number of times.
- In the following code, each of the four general purpose registers are saved and restored individually.

```
blankln   proc
          push eax
          push ebx
          push ecx
          push edx
          .repeat
          INVOKE printf, ADDR blnkfmt
          dec ebx
          .until ebx<=0
          pop edx
          pop ecx
          pop ebx
          pop eax
          ret
blankln   endp
```

Saving and Restoring Registers (Cont.)

Example

- In the code below, the four general purpose registers are saved all at once using the PUSHAD and POPAD instructions.

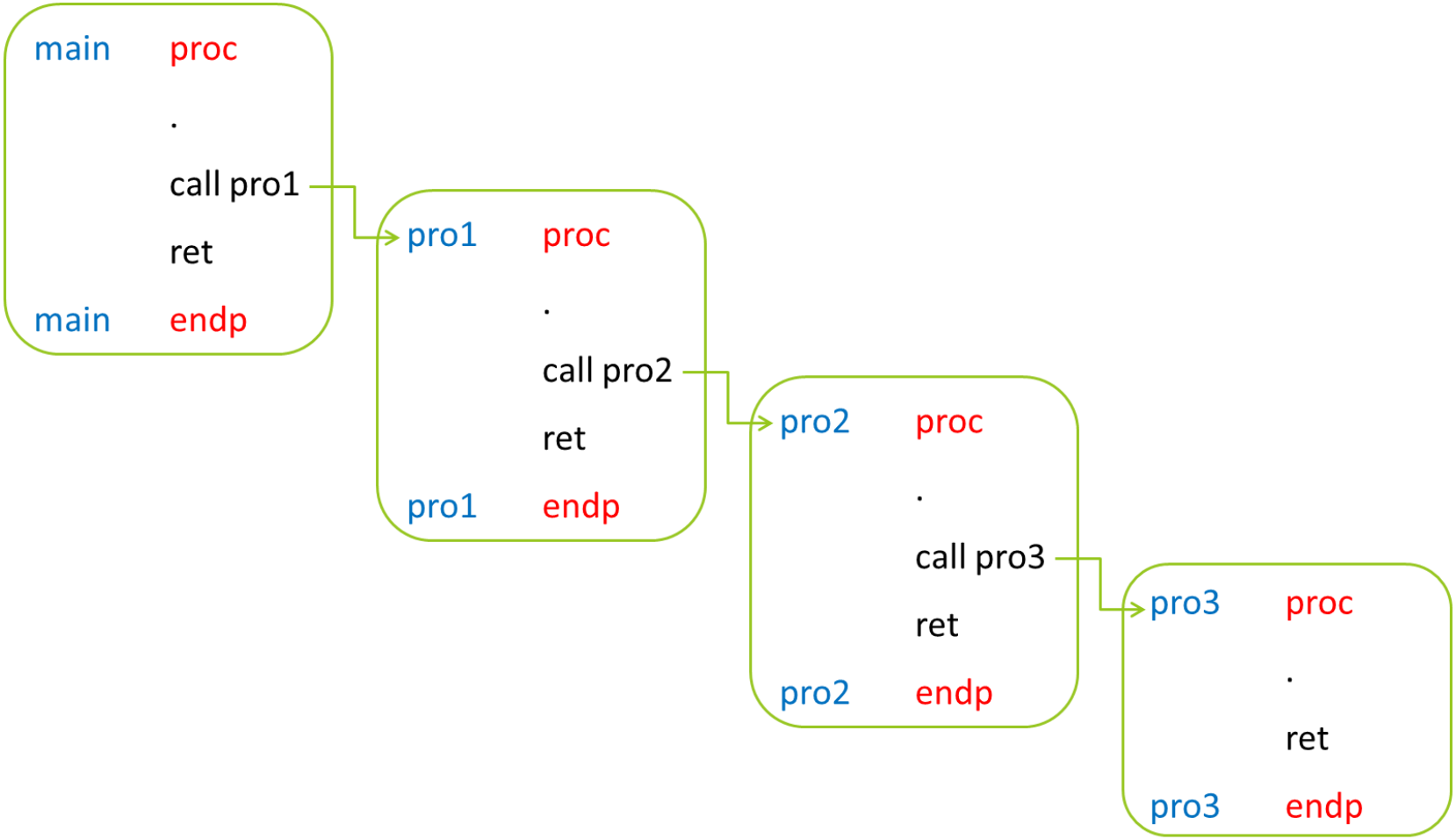
```
blankln  proc
         pushad
         .repeat
         INVOKE printf, ADDR blnkfmt
         dec ebx
         .until ebx<=0
         popad
         ret
blankln endp
```

Nested Procedure Calls

- A nested procedure call occurs when a called procedure calls another procedure before the first procedure returns.
- For clearly understanding, we will consider a MASM code segments using nested procedure calls.
- Suppose that main program calls a procedure named pro1.
- While pro1 is executing, it calls the pro2 procedure.
- While pro2 is executing, it calls the pro3 procedure.

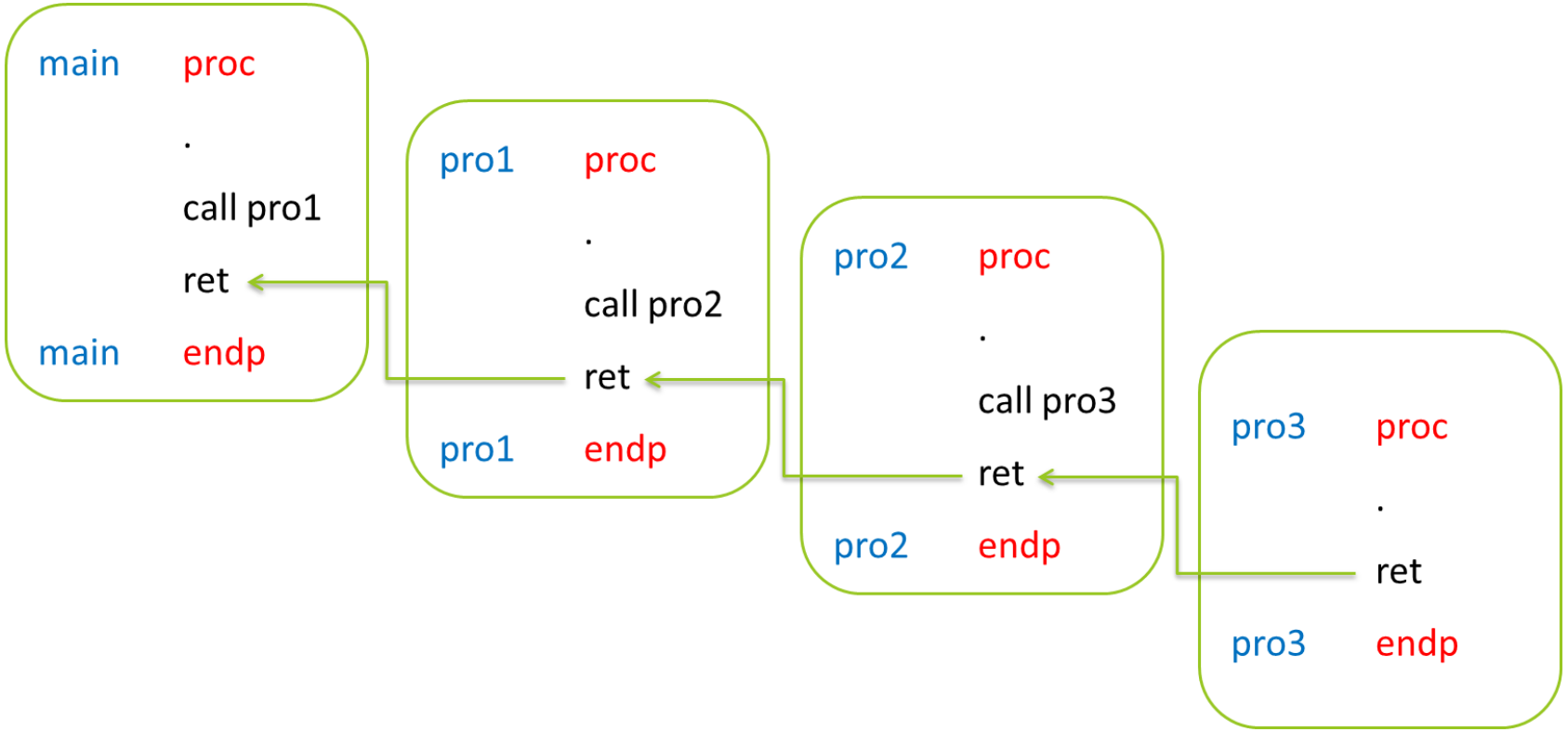
Nested Procedure Calls (Cont.)

- The invoking process is shown in figure.



Nested Procedure Calls (Cont.)

- The returning process is shown in figure.



Nested Procedure Calls (Cont.)

- When the RET instruction at the end of pro3 executes, this causes execution to resume at the instruction following the call pro3 instruction.
- When the RET instruction at the end of pro2 executes, execution resumes at the instruction following the call pro2 instruction.
- Finally, when pro1 returns, execution resumes in main.

Complete Program: Implementing the Power Function in a Procedure

- Consider the implementation of the power function (x^n) in a procedure instead of having the code to calculate x^n in the main program.
- 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 * … * x (n times)

Complete Program: Implementing the Power Function in a Procedure

- For the sake of simplicity both here in the C program and more importantly in the subsequent assembly language program, power is implemented as a procedure (void function) and x, n, and ans are implemented as global variables.
- In addition to outputting a message in the case of an error, the procedure also returns a -1 in the variable ans.
- The procedure can then be invoked more than one time from the main program without having to duplicate the code each time.

Complete Program: Implementing the Power Function in a Procedure

- The following is the implementation of the power function in a procedure in C code.

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

Complete Program: Implementing the Power Function in a Procedure

- The following is the implementation of the power function in a procedure in MASM code.

```
            .listall
            .386
            .model flat,c
            .stack 100h
scanf       PROTO arg2:Ptr Byte, inputlist:VARARG
printf      PROTO arg1:Ptr Byte, printlist:VARARG
            .data
in1fmt      byte "%d",0
msg1fmt     byte "%s",0
msg3fmt     byte "%s%d",0Ah,0Ah,0
errfmt      byte "%s",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 ?
            .code
main        proc
            INVOKE printf, ADDR msg1fmt, ADDR msg1
            INVOKE scanf, ADDR in1fmt, ADDR x
            INVOKE printf, ADDR msg1fmt, ADDR msg2
            INVOKE scanf, ADDR in1fmt, ADDR n
            call power
            INVOKE printf, ADDR msg3fmt, ADDR msg3, ans
            ret
main        endp
```

```
power       proc
            push eax    ; save registers
            push ecx
            push edx
            mov ans,-1        ; default value for ans
            .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         ; initialize ecx loop counter
            mov ans,1         ; initialize ans
            .while ecx <= n
            mov eax,ans       ; load eax with ans
            imul x            ; multiply eax by x
            mov ans,eax       ; sotre eax in ans
            inc ecx           ; increment eax loop countere
            .endw
            .endif
            .endif
            pop edx           ; restore registers
            pop ecx
            pop eax
            ret
power       endp
            end
```

Summary

- To invoke a procedure, use the CALL instruction followed by the name of the procedure.
- Always include a RET instruction in a procedure.
- Although more than one RET instruction can be included in a procedure, it is best to have only one and include it as the last instruction in a procedure.

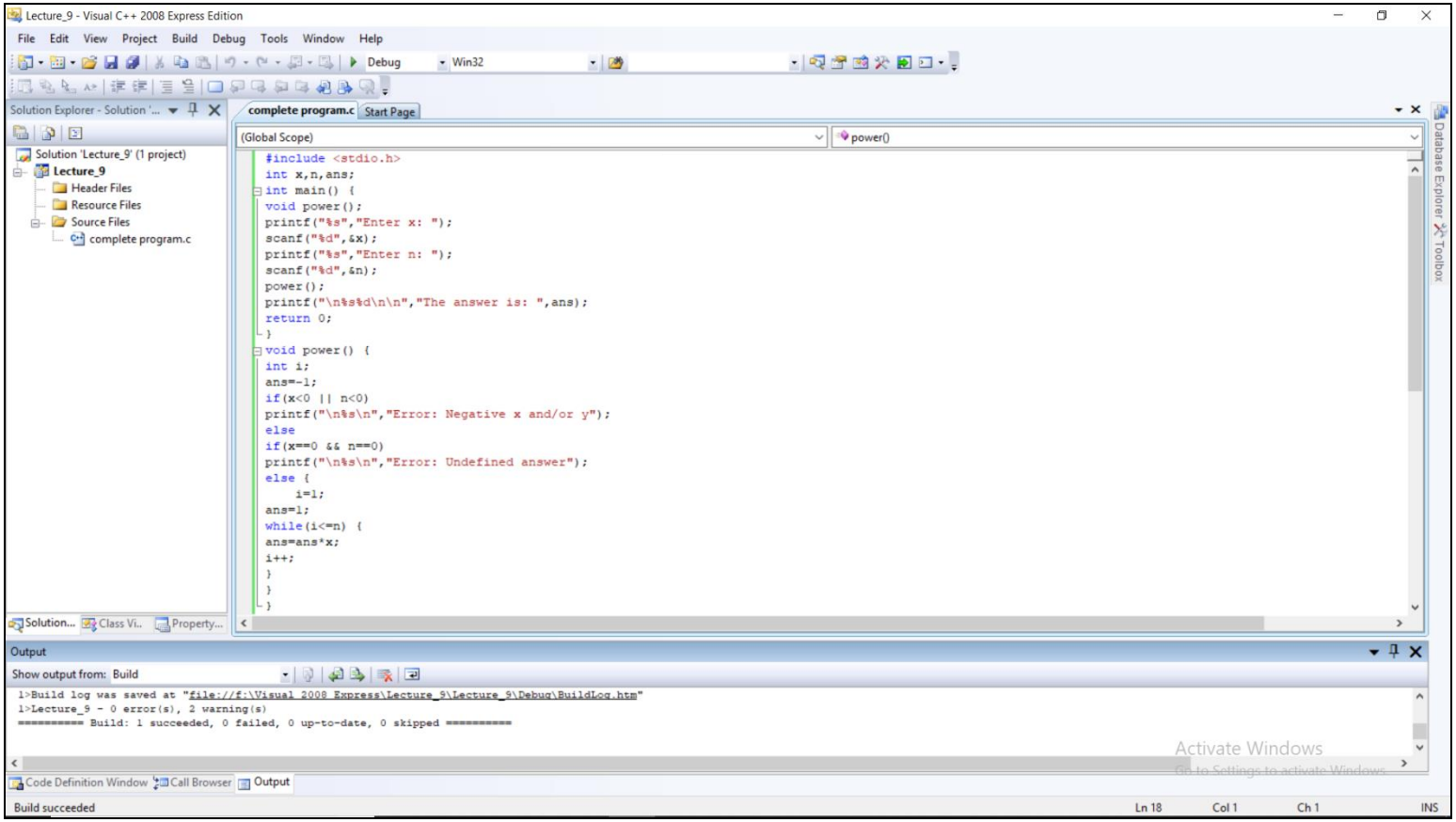
Microprocessor Programming

Practical Works

Implementing the Power Function in a Procedure

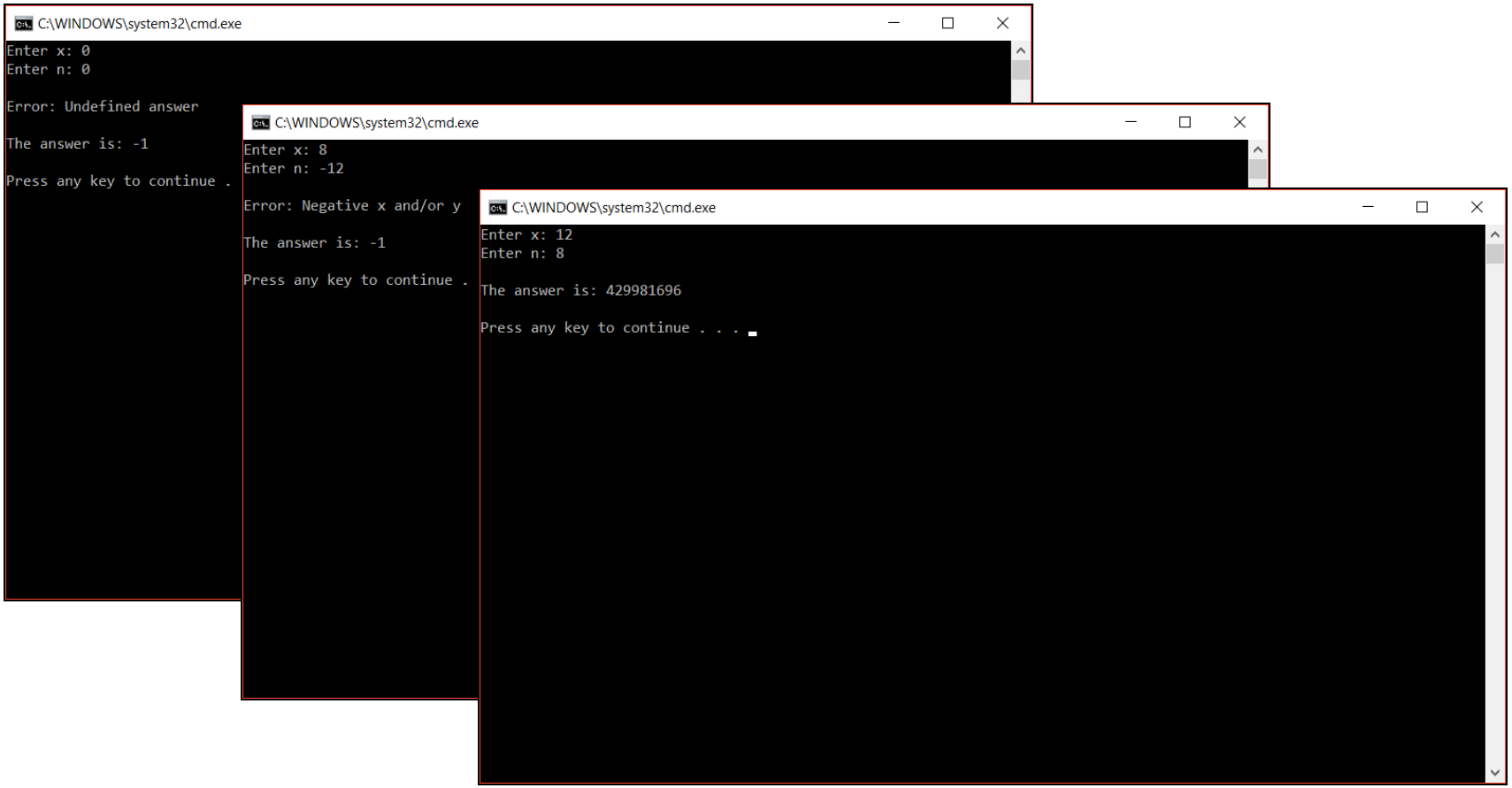
Implementing the Power Function in a Procedure

- The implementation of the given C program is described below:



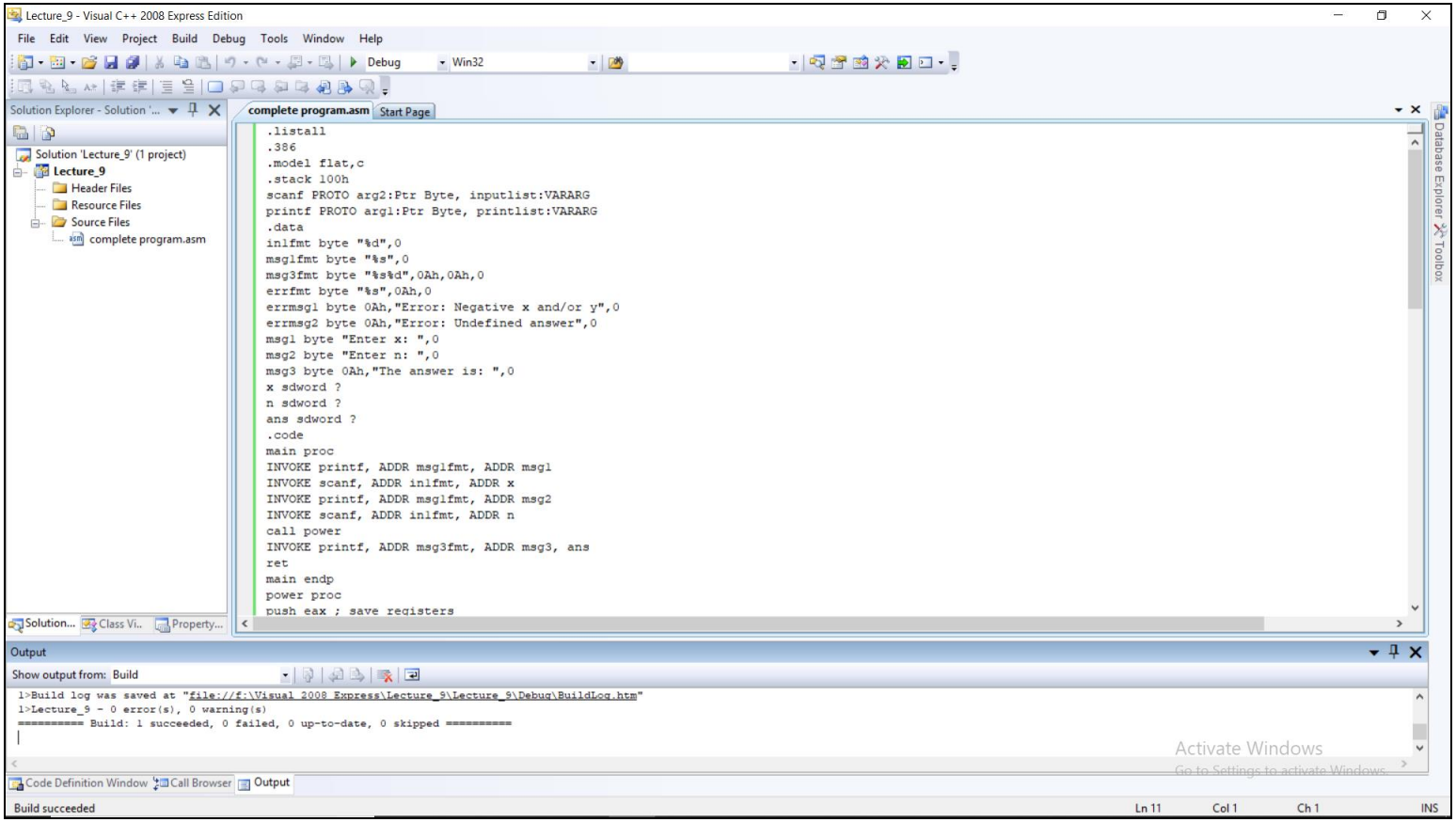
Implementing the Power Function in a Procedure

- The outputs of the given C program are described below:



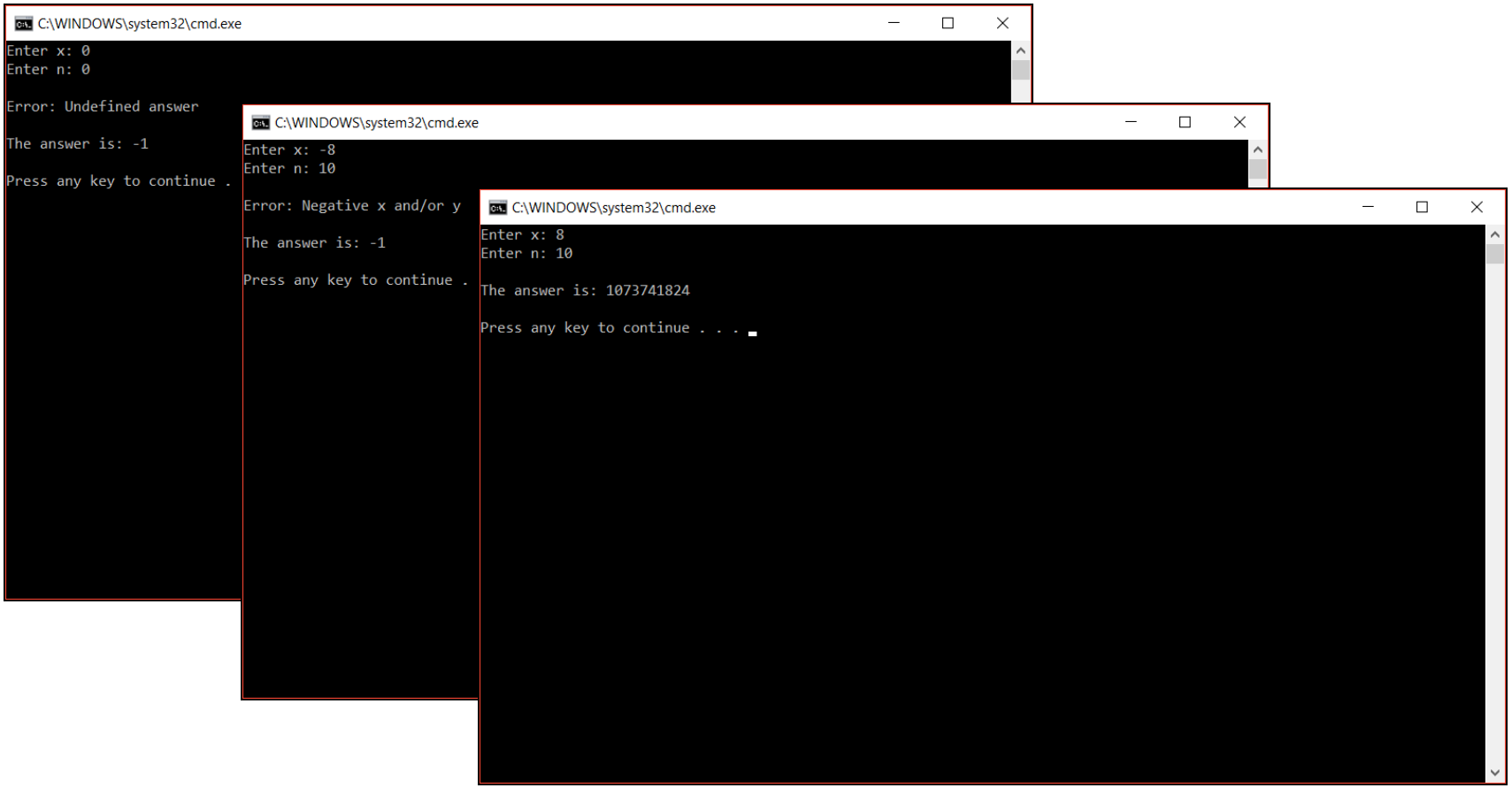
Implementing the Power Function in a Procedure

- The implementation of the given MASM program is described below:



Implementing the Power Function in a Procedure

- The outputs of the given MASM program are described below:



Microprocessor Programming

Practical Assignments (Instructions)

Assignment 1

- Rewrite a complete assembly program for the given implementation of power function in a procedure without using high-level directives with only compare and jump instructions.

Microprocessor Programming

Practical Assignments (Report)

Assignment 1

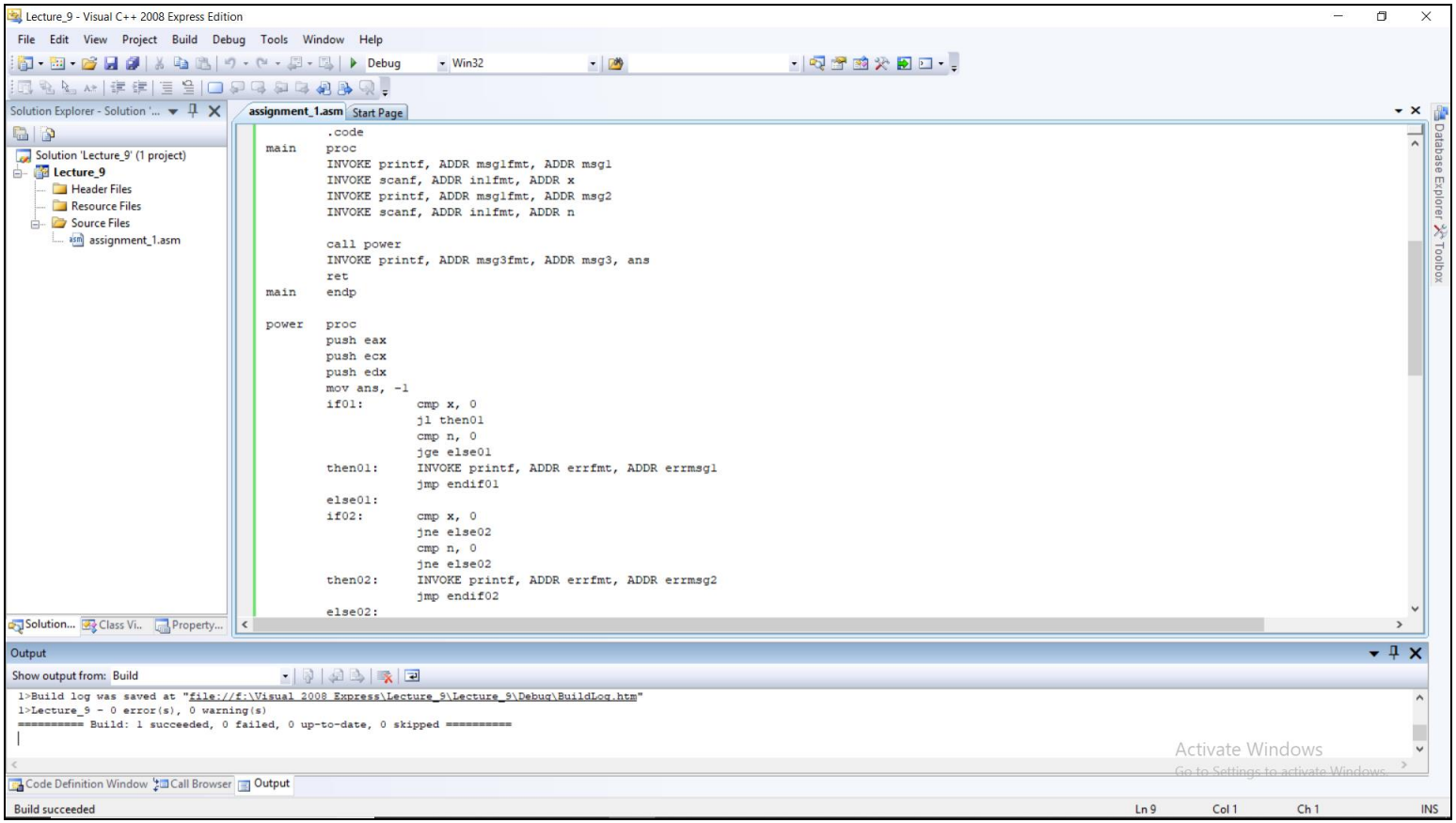
- A complete assembly program is described below:

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

            .data
in1fmt      byte "%d", 0
msg1fmt     byte "%s", 0
msg3fmt     byte "%s%d", 0Ah, 0Ah, 0
errfmt      byte "%s", 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 ?

            .code
main        proc
            INVOKE printf, ADDR msg1fmt, ADDR msg1
            INVOKE scanf, ADDR in1fmt, ADDR x
            INVOKE printf, ADDR msg1fmt, ADDR msg2
            INVOKE scanf, ADDR in1fmt, ADDR n

            call power
            INVOKE printf, ADDR msg3fmt, ADDR msg3, ans
            ret
main        endp
```

```
power       proc
            push eax
            push ecx
            push edx
            mov ans, -1
            if01:       cmp x, 0
                        jl then01
                        cmp n, 0
                        jge else01
            then01:     INVOKE printf, ADDR errfmt, ADDR errmsg1
                        jmp endif01
            else01:
            if02:       cmp x, 0
                        jne else02
                        cmp n, 0
                        jne else02
            then02:     INVOKE printf, ADDR errfmt, ADDR errmsg2
                        jmp endif02
            else02:
                        mov ecx, 1
                        mov ans, 1
            while02:    cmp ecx, n
                        jg endw02
                        mov eax, ans
                        imul x
                        mov ans, eax
                        inc ecx
                        jmp while02
            endw02:     nop
            endif02:    nop
            endif01:    nop
            pop edx
            pop ecx
            pop eax
            ret
power       endp
            end
```

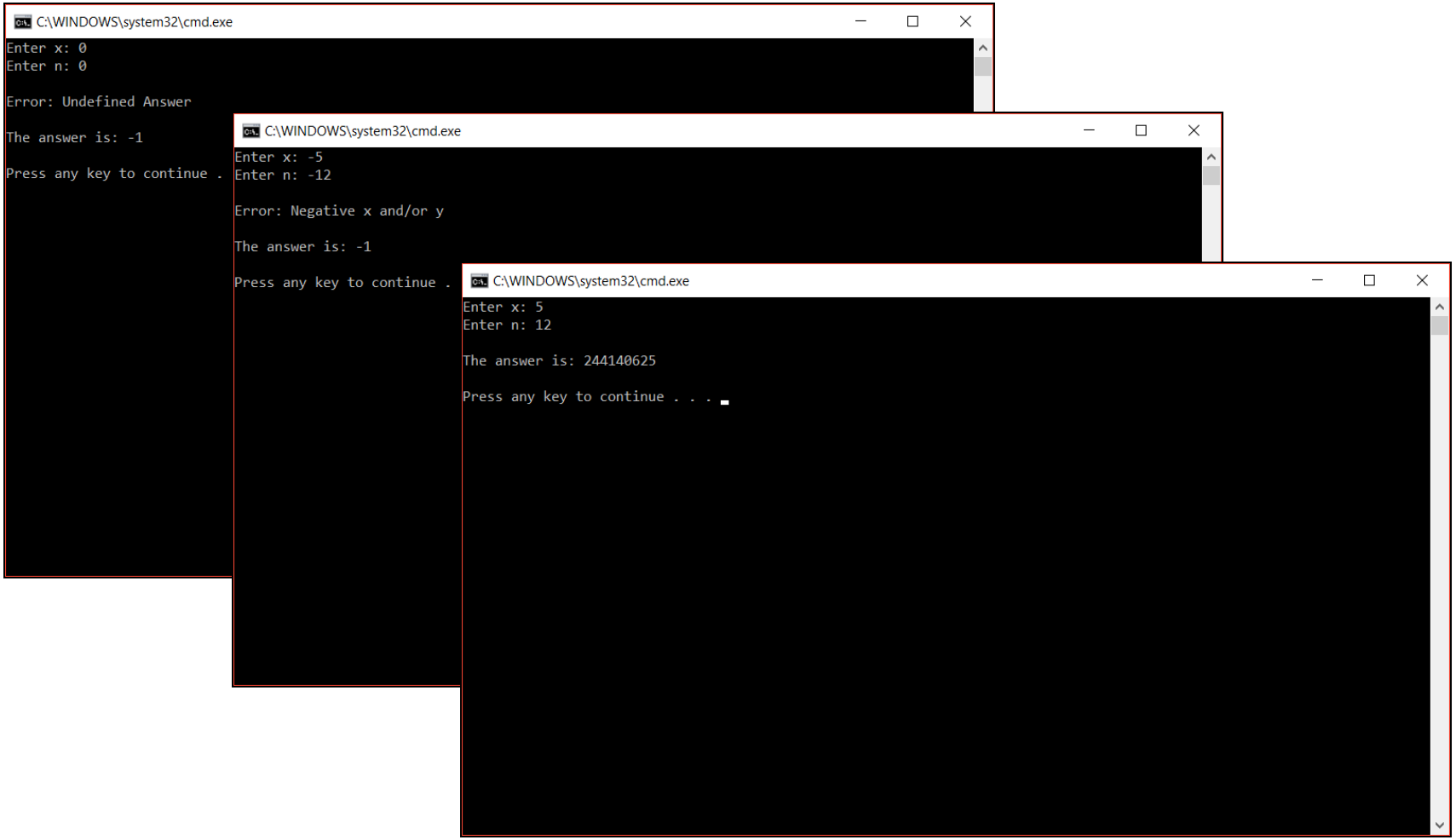
Assignment 1

- A complete assembly program is implemented as follow:



Assignment 1

- The corresponding outputs are as follow:



Next Lecture

- Macros
- Conditional Assembly

Thank You