ECE291

Lecture 6

Procedures and macros

Lecture outline

- Procedures
- Procedure call mechanism
- Passing parameters
- Local variable storage
- C-Style procedures
- Recursion
- Macros

Procedures defined

- Procedures are chunks of code that usually perform specific frequently used tasks
- They can be repeatedly called from someplace else in your programs
- These are essentially the same thing as functions or subroutines in high-level languages
- Procedures may have inputs/outputs, both, either, neither
- Essentially just labeled blocks of assembly language instructions with a special return instruction at the end

Procedure example

```
..start
;here's my main program
mov ax, 10h
mov bx, 20h
mov cx, 30h
mov dx, 40h
call AddRegs   ;this proc does AX + BX + CX + DX → AX
;here's the rest of my main program
call DosExit

;-------------------------------------------------------
AddRegs
   add ax, bx
   add ax, cx
   add ax, dx
ret
```

Proc prime directive

- Procs should never alter the contents of any register that the procedure isn't explicitly supposed to modify
- If for example a proc is supposed to return a value in AX, it is not only okay but required for it to modify AX
- No other registers should be left changed
- Push them at the beginning and pop them at the end

Call

- Call does two things
  - It pushes the address of the instruction immediately following the call statement onto the stack
  - It loads the instruction pointer with the value of the label marking the beginning of the procedure you're calling (this is essentially an unconditional jump to the beginning of the procedure

Two kinds of calls

- Near calls
  - Allow jumps to procedures within the same code segment
  - In this case, only the instruction pointer gets pushed onto the stack
- Far calls
  - Allow jumps to anywhere in the memory space
  - In this case, both the contents of the CS and IP registers get pushed onto the stack

Passing parameters

- Using registers
  - Registers are the ultimate global variables
  - Your proc can simply access information the calling program placed in specific registers
  - Also a simple way for your proc to send data back to the caller
  - The previous example used this method

Passing parameters

- Using global memory locations
  - Memory locations declared at the beginning of your program are global and can be accessed from any procedure
- Using a parameter block
  - You may pass a pointer to an array or other type of memory block instead of an individual data value

Passing parameters

- Using the stack
  - Caller pushes all arguments expected by the proc onto the stack
  - Proc can access these args directly from the stack by setting up the *stack frame*.

```
push bp                 ; save value of bp
mov bp, sp              ; mark start of stack frame
```

arg1 is at [ss:bp+4] assuming `call`

arg1 is at [ss:bp+6] assuming `call far`

Example

```
;call proc to calculate area of right triangle.
;proc expects two word sized args to be on the stack
push 3
push 4
call TriangleArea

;now we must remove the variables from the stack
;every push must be popped.
add sp, 4

; ax now contains 6
```

Example continued

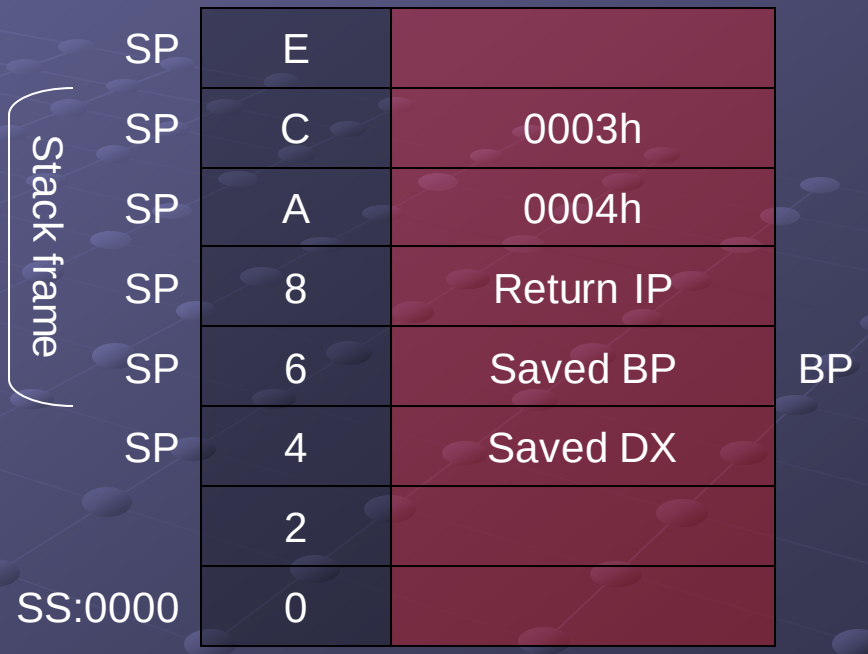
```
TriangleArea
  push bp
  mov bp, sp
  y equ bp+4     ;[bp+4] = y
  x equ bp+6     ;[bp+6] = x
  push dx
  mov ax, [y]    ;same as mov ax, [bp+4]
  mul word [x]   ;same as mul word, [bp+6]
  shr ax, 1      ;divide by two
  pop dx
  pop bp
  ret
```

What just happened…

```
Push 3
Push 4
Call TriangleArea

TriangleArea
   push bp
   mov bp, sp
   push dx
   mov ax, [bp+4]
   mul word [bp+6]
   shr ax, 1
   pop dx
   pop bp
   ret

Add sp, 4
```



Local variable storage

- You may allocate stack space for use as local variables within procedures
- Subtract from the stack pointer the number of bytes you need to allocate after setting up the stack frame
- At the end of your procedure, just add the same amount you subtracted to free the memory you allocated just before you pop bp
- In the procedure, use the stack frame to address local variable storage

Local variable storage

```
MyProc
   ;setup stack frame
   push bp
   mov bp, sp        ;allocate space for two words
   sub sp, 4         ;access words at [bp-2] and [bp-4]
…
   add sp, 4         ;destroy local variables
   pop bp            ;restore original bp
   ret
```

Things to remember

- Always make sure that your stack is consistent (a proc doesn't leave information on the stack that wasn't there before the procedure ran)
- Always remember to save registers you modify that don't contain return values

C style procedures

- Assembly procedures that can be called by C programs
- Must follow the same calling procedure that C compilers use when building C programs
- You can also call C functions from assembly programs using the same protocol

C style procedures

- Suppose a C program calls an assembly procedure as follows
  - Proc1(a, b, c)
- Assume each argument is word-sized
- Further, assume the C compiler generates code to push a, b, and c onto the stack
- The assembly language procedure must be assembled with the correct ret (near or far) and stack handling of its procedures matching the corresponding values in the high-level module

C style procedures

- Assume C program always makes far call
  - So far return (retf) must end the C style procedure
  - Return CS gets pushed onto the stack as well as return IP
  - Makes a difference when accessing arguments using the stack frame
- C compilers push arguments in reverse order, from right to left.
  - C is pushed first, then B, then A
  - Lowest index from BP corresponds to first argument

BP+10	c
BP+8	b
BP+6	a
BP+4	Return CS
BP+2	Return IP
BP	Saved BP

Proc1(a, b, c)

C style procedures

- If the returned value needs four or fewer bytes, it is by default returned in registers
  - one or two bytes - returned in AX
  - three or four bytes - returned in AX (low word) and in DX (high byte or word), (in EAX in 32-bit mode)
- More than four bytes
  - the called procedure stores data in some address and returns the offset and segment parts of that address in AX and DX, respectively

C style procedures

- Caller is responsible for clearing the arguments from the stack as soon as it regains control after the call
  - this done by the compiler that generates the appropriate code
  - a far procedure called from C should end with RETF instead of RET

Example

- Calling and ASM proc from a C program – the proc lets you display a string at a given row and column

```
# include   <stdio.h>
extern    void  placeStr (char *, unsigned, unsigned);

void  main (void)
      {
            int         n;
            for  (n = 10; n < 20; ++n)
                  placeStr (“This is the string”, n, 45);
      }
```

Example

```
GLOBAL _placeStr

SEGMENT code
_placeStr
; setup stack frame and save
   state
   PUSH         BP
   MOV          BP, SP
   PUSH         AX
   PUSH         BX
   PUSH         DX

; get current page - returns in
   BH
   MOV          AH, 0fh
   INT          10h
; read unsigned args 2 and 3
   MOV          DL, [BP+10]
   MOV          DH, [BP+8]
```

```
;set cursor position
   MOV    AH, 02h
   INT    10h
;point to string
   MOV     BX, [BP+6]
;call outAsc to disp string
   call outAsc

;restore state
   POP    DX
   POP    BX
   POP    AX
   POP    BP

   RETF
```

Putting the two together

- The C module must be compiled
- The assembly language module assembled
- The pair must be linked together into an executable
- Extern in C is exactly the same as Extern in assembly programs
- Notice that the procedure is named _placeStr, because C compilers preface all external variables with an underscore

Complete calling procedure

Program writes function parameters to stack (C is right-pusher)
CALL saves program's return address on the stack [PUSH CS (Far Proc); PUSH IP]
Routine marks stack frame (PUSH BP; MOV BP, SP)
Routine allocates stack memory for local variables (SUB SP, n)
Routine saves registers it modifies (push SI, push BX, push CX)
Subroutine Code
Additional CALLs, PUSHs, POPs)
Routine restores registers it modifies (pop CX, pop BX, pop SI)
Routine deallocates stack memory for local variables (ADD SP, n)
Routine restores original value of BP (POP BP)
Subroutine Returns (RETF)
Program clears parameters from stack (ADD SP,p)

Recursion

- Recursion: procedure calls itself

```
RecursiveProc
                DEC     AX
                JZ      .QuitRecursion
                CALL    RecursiveProc
.QuitRecursion:
                RET
```

- Requires a termination condition in order to stop infinite recursion
- Many recursively implemented algorithms are more efficient than their iterative counterparts

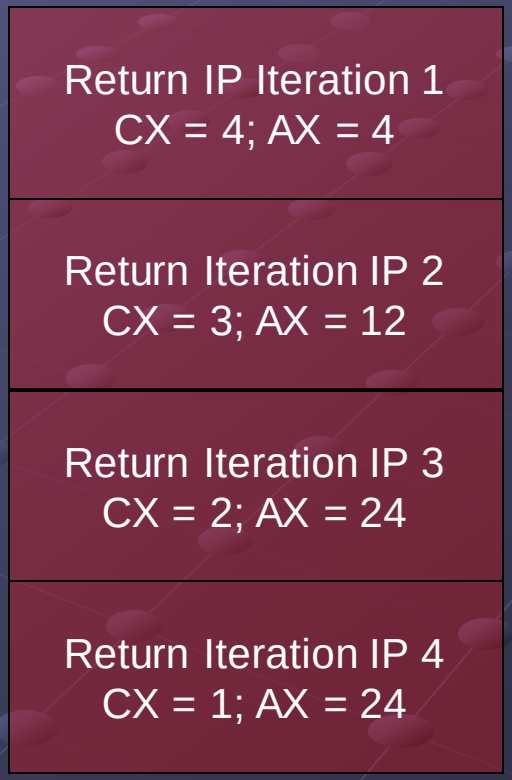
Recursion example

```
Factorial
   ; Input  AX = CX = Value
   ; Output AX = Value !

   DEC  CX
   ;Test for base case
   CMP  CX,0
   JE   .FactDone
   IMUL CX
   ; Recurs
   Call Factorial

   .FactDone:
   RET
```

Stack contents
Assume:
AX = CX = 4



Recursion

- Recursion must maintain separate copies of all pertinent information (parameter value, return address, local variables) for each active call
- Recursive routines can consume a considerable amount of stack space
- Remember to allocate sufficient memory in your stack segment when using recursion
- In general you will not know the depth to which recursion will take you
  - allocate a large block of memory for the stack

Macros

- A macro inserts a block of statements at various points in a program during assembly
- Substitutions made at compile time
  - **Not** a procedure—code is literally dumped into the program with each instantiation
  - Parameter names are substituted
  - Useful for tedious programming tasks

Macros

- Generic Format

```
%macro MACRO_NAME   numargs
  Your Code ...
  ... %{1} ...
  … %{2} ...
  Your Code ...
  JMP %%MyLabel
  Your Code ...
%%MyLabel:
  ... %{N} ...
  Your Code ...
  %endmacro
```

Local Variables in a Macro

- A local label is one that appears in the macro, but is not available outside the macro
- We use the **%%** prefix for defining a local label
  - If the label *MyLabel* in the previous example is not defined as local, the assembler will flag it with errors on the second and subsequent attempts to use the macro because there would be duplicate label names
- Macros can be placed in a separate file
  - use %include directive to include the file with external macro definitions into a program
  - no EXTERN statement is needed to access the macro statements that have been included

Macros

```
; Store into Result the signed result of X / Y
; Calculate Result = X / Y
; (all 16-bit signed integers)
; Destroys Registers AX,DX

%macro DIV16 3          ; Args: Result, X, Y
MOV   AX, %{2}          ; Load AX with Dividend
CWD                     ; Extend Sign into DX
IDIV  %{3}              ; Signed Division
MOV   %{1}, AX          ; Store Quotient
%endmacro
```

Macros

```
; Example: Using the macro in a program
; Variable Section
   varX1       DW     20
   varX2       DW     4
   varR        RESW
; Code Section
   DIV16 word [varR], word [varX1], word [varX2]

;Will actually generate the following code inline in your
   program for every instantiation of the DIV16 macro (You
   won't actually see this unless you debug the program).

   MOV  AX, word [varX1]
   CWD
   IDIV word [varX2]
   MOV  word [varR], AX
```

Macros vs. Procedures

```
Proc_1
        MOV     AX, 0
        MOV     BX, AX
        MOV     CX, 5
        RET

%macro Macro_1       0
         MOV    AX, 0
         MOV    BX, AX
         MOV    CX, 5
%endmacro
```

```
    CALL Proc_1
    …
    CALL Proc_1
    …
    Macro_1
    …
    Macro_1
```

Macros vs. Procedures

- In the example the macro and procedure produce the same result
- The procedure definition generates code in your executable
- The macro definition does not produce any code
- Upon encountering Macro_1 in your code, NASM assembles every statement between the %macro and %endmacro directives for Macro_1 and produces that code in the output file
- At run time, the processor executes these instructions without the call/ret overhead because the instructions are themselves inline in your code

Macros vs. Procedures

- Advantage of using macros
  - execution of macro expansion is faster (no call and ret) than the execution of the same code implemented with procedures
- Disadvantages
  - assembler copies the macro code into the program at each macro invocation
  - if the number of macro invocations within the program is large then the program will be much larger than when using procedures