Recursion

Module 17

Dr. Tim McGuire

COSC 2329

Sam Houston State University

Goals for this Lecture

- IDEAL Objectives
  - Gaining factual knowledge (terminology, classifications, methods, trends)
  - Learning fundamental principles, generalizations, or theories
- ABET Student Outcomes

  (a) An ability to apply knowledge of computing and mathematics appropriate to the program’s student outcomes and to the discipline

  (b) An ability to analyze a problem, and identify and define the computing requirements appropriate to its solution

  (i) An ability to use current techniques, skills, and tools necessary for computing practice.

First, a Digression

- Mathematicians and Telephones

Overview

- A recursive procedure is a procedure that calls itself
- In HLLs, the way the system uses the stack is hidden from the programmer
- In assembly language, the programmer must actually program the stack operations

Recursive Procedures

- The factorial function n! may be defined recursively
  - 0! = 1
  - n! = n * (n-1)! for n > 0
- In C, this would be coded as:

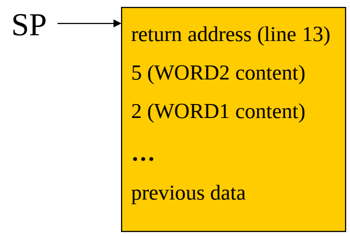
```
int fact(int n)
{
  if (n == 0)
   return 1;
  else
   return n * fact(n-1);
}
```


Passing Parameters on the Stack

- Recursive procedures are implemented by passing parameters on the stack
- This is the way that ***all*** parameter passing is done in HLLs
- An example is found in **`add_word.asm`**
- This program places the contents of two memory words on the stack, and calls a procedure ADD_WORDS that returns their sum in AX

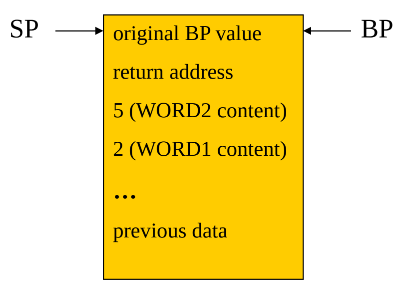
Explanation of `add_word.asm`

- The program pushes the contents of WORD1 and WORD2 on the stack, and calls ADD_WORDS
- On entry to the procedure, the stack looks like this:



Stack

- At lines 20-21, the procedure first saves the original contents of BP on the stack and sets BP to point to the stack top
- The result is:



Stack

- Now the data can be accessed by indirect addressing
- BP is used for two reasons:
  - when BP is used in indirect addressing, SS is the assumed segment register, and
  - SP itself may not be used in indirect addressing in the 8086
- At line 22, the effective address of the source in the instruction

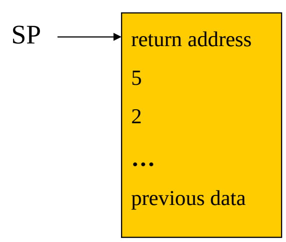
  ```
  mov ax,[bp+6]
  ```

  is the stack top plus 6, which is the location of WORD1 content (2)
- Similarly, in **`add ax, [bp+4]`**, the source is the location of WORD2 content (5)

- After restoring BP to its original value at line 24, the stack becomes:
- To exit the procedure and restore the stack to its original condition, we use

  ```
  ret 4
  ```

- This causes the return address to be popped into IP and four additional bytes (two words) to be removed from the stack

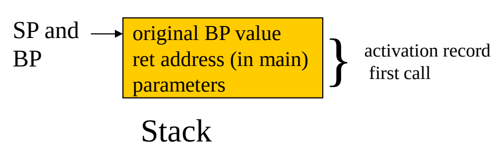


Stack

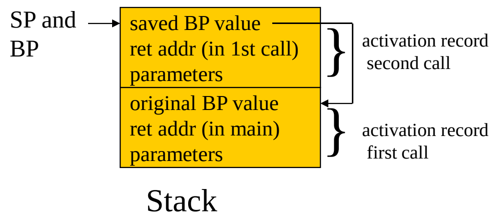
The Activation Record

- When a call has been completed, the procedure resumes the previous call at the point it left off
- It must remember that point, as well as the values of the parameters and local variables
- These values are called the ***activation record*** of the call

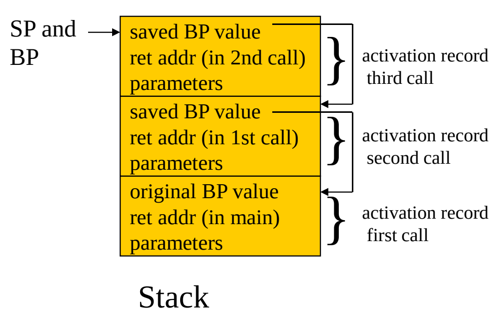
- Suppose we have a procedure that is called from the main procedure, and then calls itself twice more
- Before initiating the first call, the main procedure places the initial activation record on the stack and calls the procedure
- The procedure saves BP and sets BP to point to the stack top
- The stack now looks like this:



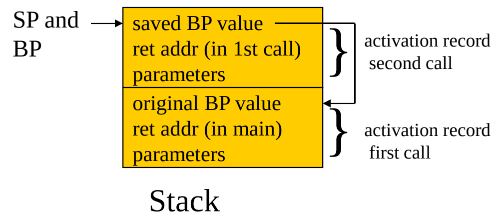
- Using BP to access the parameters and local variables, the procedure executes its instructions
- Before calling itself, it places the activation record for the next call on the stack
- The return address that is placed on the stack is that of the next instruction to be done in the procedure
- As the second call begins, BP is saved once again and BP is set to point to the stack top
- The result is:



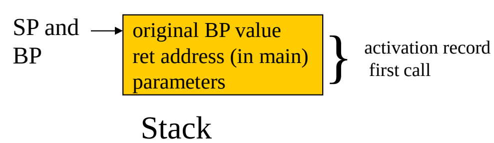
- Now, as in the first call, the procedure uses BP to access the data for the second call
- Before initiating the third call, its activation record is placed on the stack
- The third call saves BP and sets it to point to the stack top
- The stack becomes:



- If the third call is the escape case, the result it computes may be placed in a register so that it is available to the second call when the second call resumes
- After the third call is completed, the second call may be resumed by first popping BP to restore the previous value, and executing a return
- The return pops the third call's parameters and local variables off the stack and discards them
- The stack becomes:



- Now the second call resumes and picks up the result of the third call and executes to completion
- When it has finished and stored the result, the stack is once again popped into BP, and control returns to the first call
- As before, the second call's data are discarded, and the stack looks like the figure
- When the first call is done, the procedure restores BP to its original value, and control passes to the main procedure and the parameters are discarded



Implementation of Recursive Procedures

- Look at factor.asm
- It implements the factorial function discussed previously:

```
int fact(int n)
{
  if (n == 0)
   return 1;
  else
   return n * fact(n-1);
}
```


More Complex Recursion

- In the previous examples, the code for recursive procedures has only involved one recursive call
  - However, it is possible that the code for a recursive procedure may involve multiple recursive calls
- As an example, suppose we would like to write a procedure to compute the binomial coefficients C(n,k)
  - These are the coefficients that appear in the expansion of $(x+y)^n$

The Binomial Coefficients

- These coefficients are also used in the construction of Pascal's Triangle:

```
    1                         C(0,0)
   1 1                 C(1,0)      C(1,1)
  1 2 1          C(2,0)      C(2,1)      C(2,2)
 1 3 3 1   C(3,0)      C(3,1)      C(3,2)      C(3,3)
1 4 6 4 1                      etc.
```

- The coefficients satisfy the following relation:
  - $C(n,0) = C(n,n) = 1$ for all n
  - $C(n,k) = C(n-1,k) + C(n-1,k-1)$ for $n > k > 0$

- For example
  - C(2,1) = C(1,1) + C(1,0)
  - C(1,1) = 1
  - C(1,0) = 1
  - So, C(2,1) = 1 + 1 = 2
  - Similarly, C(3,2) = C(2,2) + C(2,1) = 1 + 2 = 3
- In C, this would be coded as:

```
int C(int n, int k)
{
  if ((k == 0)||(k == n))
   return 1;
  else
   return C(n-1,k) + C(n-1,k-1);
}
```

- The code is in **binomial.asm**
- The function differs from the previous examples in the following ways:
  - There are two trivial cases, $k = n$ or $k = 0$
  - In the general case, computation of $C(n, k)$ involves two recursive calls to compute $C(n - 1, k)$ and $C(n - 1, k - 1)$
- To fully understand how function BINOMIAL works, you are encouraged to trace the effect of the function on the stack