Implementing Recursion in Assembly

Credits: original slides by Ms. Noshaba Nasir

Recursion

- A recursive subroutine is one that calls itself, either directly or indirectly.
- Recursion, the practice of calling recursive subroutines, can be a powerful tool when working with data structures that have repeating patterns.
- Examples are linked lists and various types of connected graphs where a program must retrace its path.

Infinite Recursion

- If you are not careful when creating recursive subroutines, it is easy to fall into infinite recursion trap.

```
endless:
   mov ax, bx
   call endless
   ret

start:
   call endless
   mov ax, 0x4c00
   int 21h
```

Useful recursive subroutines

- Useful recursive subroutines always contain a **terminating condition**.
- When the terminating condition becomes true, the stack unwinds when the program executes all pending RET instructions.
- There are two cases in recursive code: base case and recursive case
- To illustrate, consider the recursive procedure which sums the integers 1 to n

```
int calc_sum (int n)
{
   if (n==0) // termination condition
      return 0;

   return n + calc_sum(n-1); // recursive call
}
```

Code loosely translated in assembly

```
; Input parameter n is passed in CX
; CalcSum returns the sum in AX

CalcSum:
    cmp cx, 0   ; check  termination condition
    je L2
    add ax, cx
    dec cx
    call CalcSum
L2: ret

start:
    mov cx, 5   ; suppose n=5
    mov ax, 0
    call CalcSum
L1: mov ax, 4c00h
    int 21h
```

Stack and Registers on Calls

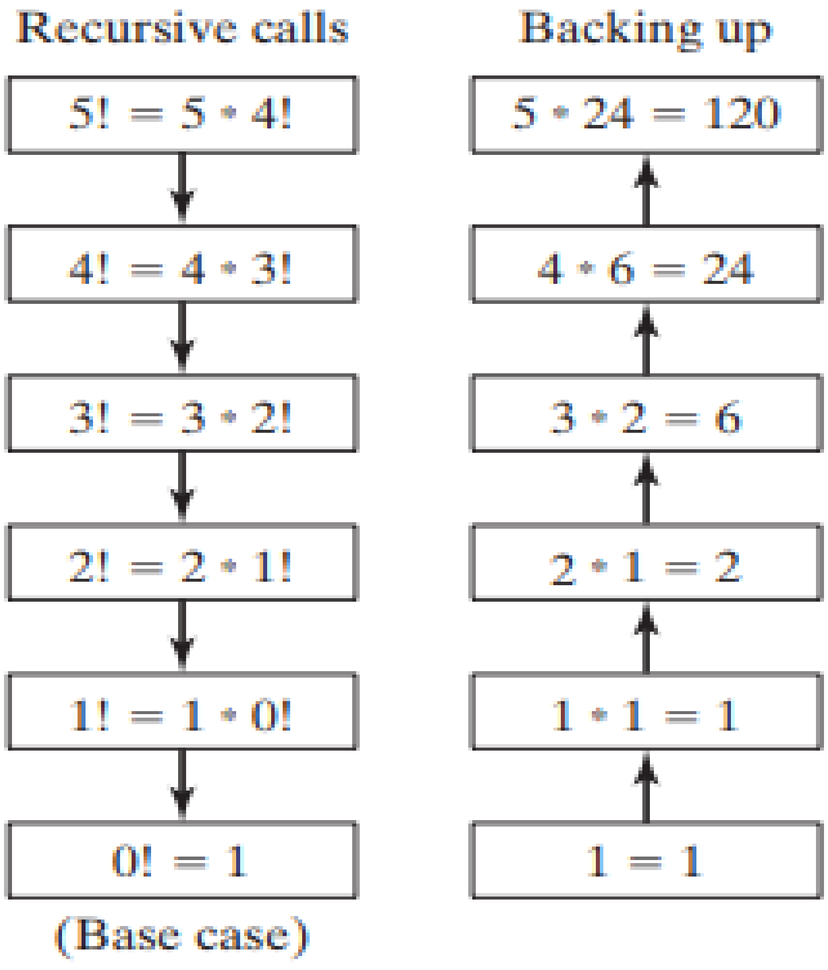
Calls for n =5

- ax=0, cx=5, L1 is pushed on stack (first call to calcSum(5))
- ax=5, cx=4, L2 is pushed on stack (recursive call to calcSum(4))
- ax=9, cx=3, L2 is pushed on stack (recursive call to calcSum(3))
- ax=12, cx=2, L2 is pushed on stack (recursive call to calcSum(2))
- ax=14, cx=1, L2 is pushed on stack (recursive call to calcSum(1))
- ax=15, cx=0, L2 is pushed on stack (recursive call to calcSum(0))
- Ret by calcSum(0), ax=15, cx=0
- Ret by calcSum(1), ax=15, cx=0
- Ret by calcSum(2), ax=15, cx=0
- Ret by calcSum(3), ax=15, cx=0
- Ret by calcSum(4), ax=15, cx=0
- Ret by calcSum(5), ax=15, cx=0
- Final answer is in ax=15

Example: Factorial

```
int factorial (int n)
{
    if (n==0 || n==1) // Base condition
        return 1;
    else
        // Non base condition
        return n * factorial(n-1);
}
```

Example: Factorial



Steps: How to **call** the recursive subroutine

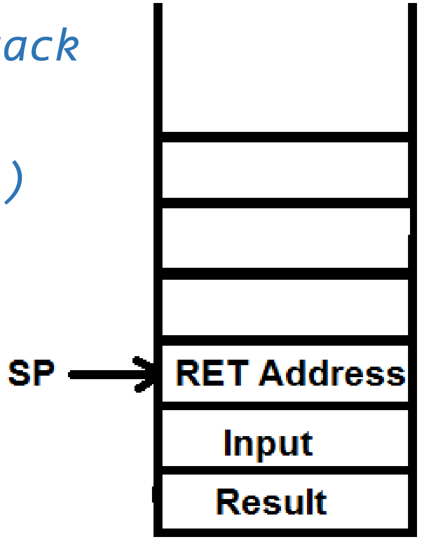
1. Create space for result on stack (result-space)
2. Prepare and push input parameters on the stack
3. Call your recursive procedure (subroutine)
4. Use the results (pop from stack)

First write your main

```
main:

sub sp, 2 ; Step 1: Create result-space on stack
push 3    ; Step 2: Push input parameters on the stack
call fact ; Step 3: Call your recursive subroutine
pop ax    ; Step 4: Use the results (Pop from stack)
; now ax will have the factorial result

mov ax, 0x4c00 ; Terminate
int 21h
```



Stack after Step 3

Steps: How to **write** the recursive subroutine

a) Start creating the subroutine using the standard template
   - Push BP, copy SP to BP
   - Save all important registers on the stack
   - At the end, restore all registers back from the stack
   - Return from subroutine and clear the parameters from stack

b) Write your base condition

c) Place the result of base condition on stack (in result-space created by caller)

d) Write your non base condition in which you call the same subroutine again (again follow steps 1-3 on previous slide)

e) Use the results from this call (pop the result from stack and do all the necessary calculations)

f) Place the result of non-base condition on stack

Do all this while keeping in mind the position of important data on stack, so that you can access it when needed

Next, write your subroutine

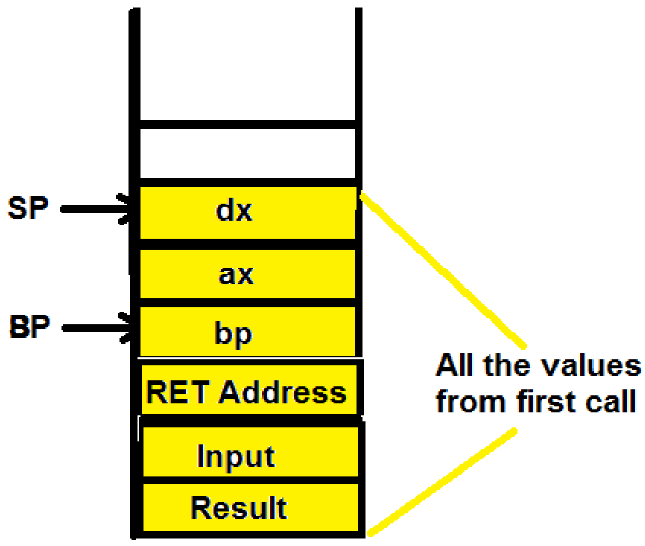
Step a. Start creating the subroutine using the standard template

```
fact: push bp        ; save previous BP
      mov bp, sp     ; take snapshot of SP
      push ax        ; save registers
      push dx

      ; --- factorial calculation code
      ; --- to be added here

exit: pop dx      ; restore registers
      pop ax
      pop bp

      ret 2       ; one parameter only (free 2 bytes on stack)
```



Note: You can modify 'save-registers' part at the end, once you finalize which registers will be used in your subroutine.

Step b. Write your base condition
Step c. Place the result of base condition on stack (in result-space created by caller)

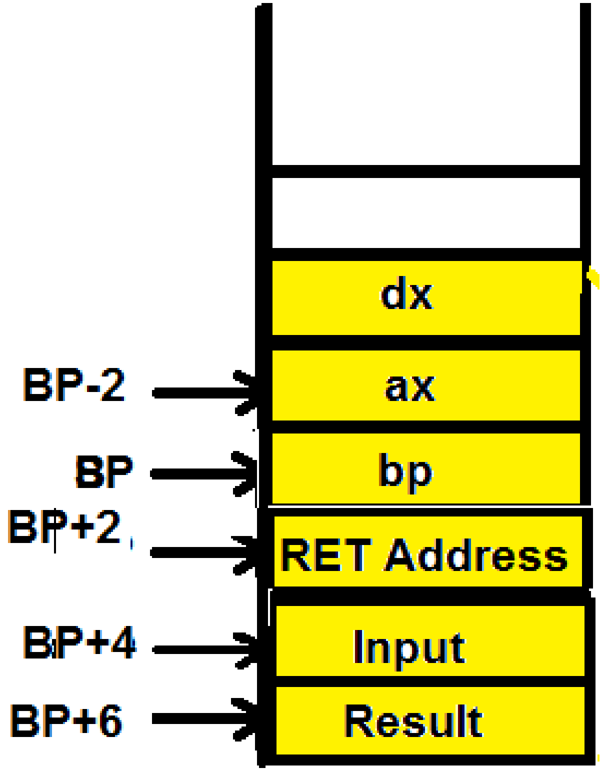
C++

```
if (n==0 || n==1)
    return 1;
```

Assembly

```
cmp word [bp+4], 0   ; if n==0
je baseExit
cmp word [bp+4], 1   ; if n==1
je baseExit

baseExit:
mov word [bp+6], 1   ; Return 1 (Place 1 at result space)
```



C++

```
return n * factorial(n-1);
```

Step d. Write your non base condition in which you call the same subroutine again

```
sub sp, 2          ; Step 1. Create result-space on stack
mov ax, [bp+4]
dec ax
push ax            ; Step 2. Push input param on the stack (n-1)
call fact          ; Step 3: Make the recursive call
```

C++

```
return n * factorial(n-1);
```

Step e. Use the results from this call (Pop the result, do all the necessary calculations)

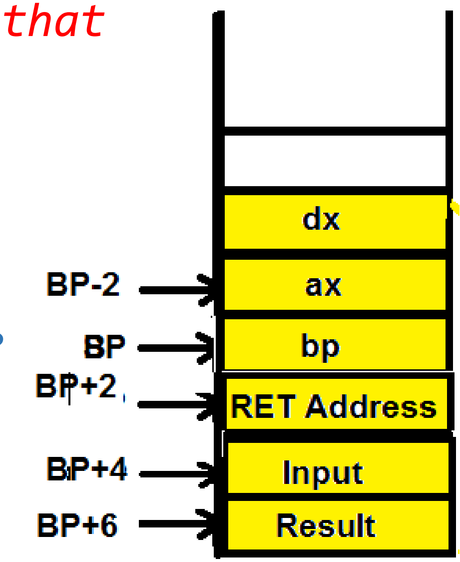
```
pop ax              ; returned result
mul word [bp+4]     ; dx-ax = n * f(n-1) i.e. [bp+4] * ax
```

Note: BP+4 was the input for this call

mul result (32 bits) goes in dx-ax. Let's assume that it fits in the lower order 16 bits only (ax).

Step f. Place the result of non-base condition on stack (in result space)

```
mov [bp+6], ax    ; put result in result-space
```



Summing Up all the code

factorial subroutine

```
fact:
  push bp       ; save previous BP
  mov bp, sp    ; take snapshot of SP
  push ax       ; save registers
  push dx

  ; Base condition
  cmp word [bp+4], 1  ; if n==1
  je baseExit
  cmp word [bp+4], 0  ; if n==0
  je baseExit

  ; Non base condition - prepare for next call
  sub sp, 2         ; create result-space on stack
  mov ax, [bp+4]
  dec ax
  push ax           ; place input parameter (n-1)
  call fact         ; make the recursive call
```

```
  pop ax              ; returned result
  mul word [bp+4]     ; ax = n * f(n-1)

  mov [bp+6], ax      ; place result in result-space
  jmp exit

baseExit:
  mov word [bp+6], 1  ; place 1 in result-space

exit:
  pop dx    ; restore registers
  pop ax
  pop bp

  ret 2     ; return and release 2 bytes
```

Summing Up all the code

main program

```
    [org 100h]
    jmp main

    ; ----------------------
    ; complete fact subroutine here
    ; ----------------------

main:

    sub sp, 2   ; create result-space on stack
    push 8      ; push input parameter (n)
    call fact   ; call recursive subroutine
    pop ax      ; retrieve the results from stack
                ; now ax will have the factorial result

finish:
    mov ax, 0x4c00
    int 21h
```

Exercise 1: Fibonacci

- Go through all these steps to create a recursive FIB subroutine

```
int fibo (int x) {

    if (x==1 || x==0)
        return x;

    else
        return fib(x-1) + fib(x-2);
}
```

Exercise 2: Palindrome

- Write this recursive function in Assembly language following all the steps discussed previously

```
int palindrome (char* str, int length) {
    if (length <= 1)
        return 1;
    else
        return (palindrome(str+1, length-2) &&
                                     (*str == *(str+length-1)));
}
```

Exercise 3: Tower of Hanoi

- Follow the link to understand the problem of tower of Hanoi and using the given pseudo code on link, write an assembly language code to solve this problem

https://www.cs.cmu.edu/~cburch/survey/recurse/hanoiimpl.html