Subprograms

ICS312 - Spring 2008
Machine-Level and Systems Programming

Henri Casanova (henric@hawaii.edu)

Subprograms

- Subprograms (functions, procedures, methods) are key to making programs easier to read and write (code reuse)
- We are going to see how to define and call subprograms in assembly
  - Useful to write large(r) assembly programs
  - More importantly, will allow us to understand how subprograms work in higher-level languages
- But first, let's just review the concept of indirection

Indirect Addressing

- So far we have seen one way to address the content of memory
  - Define a symbol in the .bss or .data segment

```
L    dd    FA123BDEh
```

  - Use that symbol as an address so that we can access content

```
mov    eax, [L]
```

- L is really an address in memory, and [L] is the content of whatever is stored at that address
  - The mov instruction above knows that eax is 32-bit, so it will read 32 bits starting at address L
- We have also used registers to store addresses

```
mov   eax, L        ; eax stores the address
mov   bx, [eax]     ; put the 2 bytes starting at
                    ; address eax into bx
```

Indirect Addressing

- So registers can hold "data" or "addresses"
  - Not keeping this straight leads to horrible bugs
  - e.g., if you think that a register contains an integer, but in fact it stores the address of the integer in memory, then your arithmetic operations on that integer will return very strange results
- Since addresses are 32-bit, only the EAX, EBX, ECX, EDX, ESI, and EDI registers can be used to store addresses in a program
- Storing addresses into a register makes it possible to implement our first subprogram

What is a subprogram?

- A subprogram is a piece of code that starts at some address in the text segment
- The program can jump to that address to "call" the subprogram
- When the subprogram is done executing it jumps back to the instruction after the call
- The subprogram can take parameters
- Let's see how we can implement this using only what we've seen so far in the course

Example Subprogram

- Say we want to write a subprogram that computes some numerical function of two operands and "returns" the result
  - e.g., because we need to compute that function often
- We will write the program so that when it is called, the first operand is in eax and the second in ebx, and when it returns the result is in eax
  - This is a convention that we make, and that should be documented in the code
- Calling the program can then be done via a simple jmp
- Let's look at the code

"By hand" subprogram

```
   . . .
   mov  eax, 12      ; first operand = 12
   mov  ebx, 14      ; second operand = 14
   jmp  func         ; "call" the function
ret:
   . . .
   . . .
func:
   add eax, ebx      ; do something with eax and ebx
                     ; put result in eax
   jmp ret           ; "return" to the instruction
                     ; after the call
```

Why isn't this really a valid implementation of a subprogram?

Multiple Calls?

- Typically we want to call a function from multiple places in a program
- The problem with the previous code is that the function always returns to a single label!

```
   . . .
   jmp  func      ; "call" the function
ret1:
   . . .
   jmp  func      ; "call" the function
ret2:
   . . .
func:
   . . .
   jmp  ???       ; where do we return???
```

A Better Function Call

- To fix our previous example, we simply need to remember the place where the function should return!
- This can be done by storing the address of the instruction after the call in a register, say, register ecx
- The code for the function then can just return to whatever instruction ecx points to
  - Again, this is a convention that we decide as a programmer and that we must remember

A Better Function Call

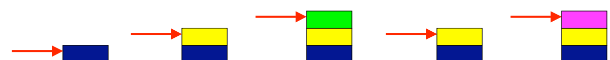
```
   . . .
   mov  ecx, ret1      ; store the return address
   jmp  func           ; "call" the function
ret1:
   . . .
   mov  ecx, ret2      ; store the return address
   jmp  func           ; "call" the function
ret2:
   . . .
func:
   . . .
   jmp  ecx       ; return
```

All Good, but ...

- So at this point, we can do any function call
- We just need to decide on convention about which registers hold
  - input parameters
  - return value
  - return address
- The problem is that this gets very cumbersome
  - It requires a bunch of "ret" labels
    - The book shows how the return address can be computed numerically as "$ + x", where x is the length in bytes of the address of the "jump func" instruction, which is very awkward
  - It forces the programmer to constantly keep track of registers and be careful to save and restore important values
- Solution:
  - A stack
  - Two new instructions: CALL and RET

The Stack

- A stack is a Last-In-Last-Out data structure
- Provides two operations
  - Push: puts something on the stack
  - Pop: removes something from the stack
- Defined by the address of the "element" at the top of the stack
  - Push: puts the element on top of the stack and increments the stack pointer
  - Pop: gets the element from the top of the stack and decrements the stack pointer
- Our stack only allows pushing/popping of elements that are double words (4-byte elements)
  - Note "quite" true, but a much safer approach



The Stack and the ESP Register

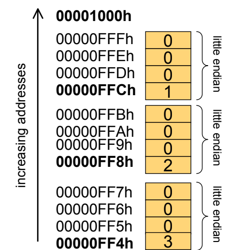
- Initially the stack is empty and the ESP register has some value
- Pushing an element:
  - Decrease ESP by 4
  - Write 4 bytes at address ESP
  - Examples:
    - push eax
    - push dword 42
- Popping an element:
  - Get the value from the top of the stack into a register
  - Increase ESP by 4
  - Examples:
    - pop eax
    - pop ebx
- Accessing an element:
  - Read the 4 bytes at address ESP
  - Example:
    - mov eax, [esp]

Example Stack Instructions

- Assuming that ESP=00001000h

```
push    dword   1       ; ESP = 00000FFCh
push    dword   2       ; ESP = 00000FF8h
push    dword   3       ; ESP = 00000FF4h

pop     eax             ; EAX = 3
pop     ebx             ; EBX = 2
pop     ecx             ; ECX = 1
```



The ESP Register

- The ESP register always contains the address of the element at the top of the stack
- Do not use it for anything else!
- Its value is typically updated by calls to push and pop
- Sometimes we'll update it by hand
  - See this in a few slides

PUSHA and POPA

- One use of the stack is to save/restore register values
- For instance, say your program uses eax and calls a function written by somebody else
- You have no idea (or don't care to know) whether that function uses eax also
  - If it does, your eax will be corrupted
- One easy solution:
  - push eax onto the stack
  - call the function
  - pop eax to restore its value
- The x86 offers two convenient instructions
  - PUSHA: pushes EAX, EBX, ECX, EDX, ESI, EDI, and EBP onto the stack
  - POPA: restores them and pops the stack
- It's now simple to say "save all my registers" and "restore my registers"

Recall the NASM Skeleton

```
    ; include directives

segment .data
    ; DX directives

segment .bss
    ; RESX directives

segment .text
        global asm_main
    asm_main:
        enter   0,0
        pusha
        ; Your program here
        popa
        mov     eax, 0
        leave
        ret
```

Save the registers since they may have been in use by the "driver" program

Restore the registers so that the "driver" program will not be disrupted by the call to function asm_main

The CALL and RET Instructions

- One of the annoying things with our previous subprogram was that we had to manage the return address
  - In our example we stored it into the ECX register
- Two convenient instructions can do this for us
- CALL:
  - Puts the address of the next instruction on the stack
  - Unconditionally jumps to a label (calling a function)
- RET:
  - Pops the stack and gets the return address
  - Unconditionally jumps to that address (returning from a function)

Without CALL and RET

```
        . . .
        mov  ecx, ret1        ; store the return address
        jmp  func             ; "call" the function
ret1:
        . . .
        mov  ecx, ret2        ; store the return address
        jmp  func             ; "call" the function
ret2:
        . . .
func:
        . . .
        jmp  ecx              ; return
```

With CALL and RET

```
        . . .
        call  func            ; call the function
        . . .

        call  func            ; call the function
        . . .

func:
        . . .
        ret                   ; return
```

Recall the NASM Skeleton

```
    ; include directives

segment .data
    ; DX directives

segment .bss
    ; RESX directives

segment .text
        global asm_main
    asm_main:
        enter    0,0
        pusha
        ; Your program here
        popa
        mov      eax, 0
        leave
        ret
```

Returns from function asm_main

Nested Calls

- The use of the stack enables nested calls
  - Return addresses are popped in the reverse order in which they were pushed (Last-In-First-Out)
- Warning: one must be extremely careful to pop everything that's pushed on the stack inside a function
- Example of erroneous use of the stack:

```
func:
        mov   eax, 12     ;
        push  eax         ; put eax on the stack
        ret               ; pop eax and interpret
                          ; it as a return address!!
```

Activation Records

- The stack is useful to store and retrieve return addresses, transparently managed via the CALL and RET instructions
- But it's much more useful than this
- In general, when calling a function, one puts all kinds of useful information on the stack
- When the function returns, this information is popped off the stack and the function's caller can safely resume execution
- The set of "useful information" is typically called an activation record (or a "stack frame")
- One very important component of an activation record is the parameters passed to the function
- Another is the return address, as we've already seen

Subprogram Conventions

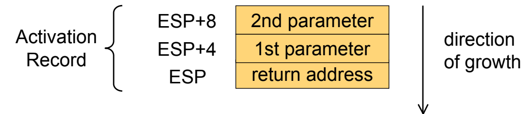
- Note that when writing assembly, you can do whatever you want
- For instance, you could devise a clever scheme that reuses register values in creative ways instead of the stack
- Such solutions are typically error prone, making the code difficult to debug/extend/maintain, but can enhance performance
- Typically, one uses a consistent calling convention, so that there is a generic way to call a subprogram
- Of course compilers use calling conventions
  - The compiler, when generating assembly code, must follow a standard process to generate assembly corresponding to function calls and returns
- Some languages specify which calling convention should be used
- What we describe in all that follows is mostly the convention used by the C language
  - i.e., C compilers should use this convention when generating assembly code from C code

A Simple Activation Record

- To call a function you have to follow the following steps:
  - Push the parameters onto the stack
  - Execute the CALL instruction, which pushes the return address onto the stack
- Warning: In the C calling convention parameters are pushed onto the stack in reverse order!
  - Say the function is f(a,b,c)
  - c is pushed onto the stack first
  - b is pushed onto the stack second
  - c is pushed onto the stack third

A Simple Activation Record

- Say you want to call a function with 2 32-bit parameters
  - If parameters are < 32 bits, they need to be converted to 32-bit values
- After the call, the stack looks like this:



Using the Parameters

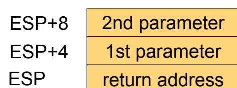
- Inside the code of the subprogram, parameters can be simply accessed via indirection from the stack pointer
- In our previous example:
  - mov eax, [ESP + 4] ; put 1st parameter into eax
  - mov ebx, [ESP + 8] ; put 2nd parameter into ebx
- Typically the subprogram does not pop the parameters off the stack before using them
  - It would be annoying to have to pop the return address first, and then push it back
  - It's convenient to have the parameters always stored in memory as opposed to being careful to constantly preserve them in registers

ESP and EBP

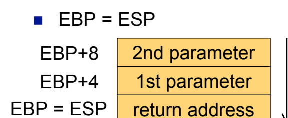
- There is one problem with referencing parameters using ESP, as in [ESP+8]
- If the subprogram uses the stack for something else, ESP will be modified!
  - So at some point in the program, the 2nd parameter should be accessed as [ESP+8]
  - And at some other point, it may be accessed as [ESP+12], [ESP+16], etc., depending on how the stack grows
- So the convention is to use the EBP register to save the value of ESP as soon as the subprogram starts
- Afterwards, the 2nd parameter is **always** accessed as [EBP+8] and the 1st parameter is **always** accessed as [EBP+4]

ESP and EBP

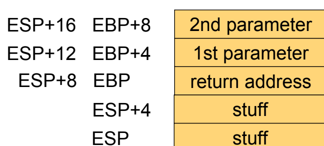
- Stack as it is when the subprogram begins



- EBP = ESP



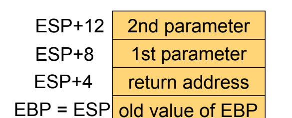
- Further use of the stack



Parameters still referred to as EBP+4 and EBP+8

ESP and EBP

- So far so good, but the **caller** may have been using EBP!
  - Typically to access its own parameters
- So the convention is to first **save the value of EBP** onto the stack and then set **EBP = ESP**, as soon as the program starts
- So, the stack right before the subprogram truly begins is:



- Parameter accesses:
  - 1st parameter: [EBP+8]
  - 2nd parameter: [EBP+12]
- At the end of the subprogram, the value of EBP is popped and restored with a simple POP instruction

Subprogram Skeleton

```
func:
    push    ebp         ; save original EBP
    mov     ebp, esp    ; set EBP = ESP

    . . .               ; subprogram code

    pop     ebp         ; restore original EBP
    ret                 ; returns
```

Returning from a Subprogram

- After the subprogram returns, one must “clean up” the stack
- The stack has on it:
  - The return address
  - The parameters
  - The old EBP value
- The old EBP value must be popped in the subprogram (at the end)
- The return address is removed by the RET instruction
  - You don’t see the POP, but it’s there
- So the only thing that must be removed from the stack are the parameters
- The C convention specifies that the caller code must do this
  - Other languages specify that the callee must do it
- In fact, it is well known that it’s a little bit more efficient to have the subprogram (i.e., the callee) do it!
- So one may wonder why C opts for the slower approach
- Turns out, it’s all because of *varargs*

Variable Number of Arguments

- C allows or the declaration of functions with variable number of arguments
- A well-known example: printf()
  - printf(“%d”, 2);
  - printf (“%d %d”, 2, 3);
  - printf(“%s %d %c %f”, “foo”, 1, ‘f’, 3.14);
- So sometimes there will be 1 argument to remove from the stack, sometimes 2, sometimes 3, etc.
- Having the subprogram (in this case printf) remove the arguments from the stack requires some complexity
  - e.g., pass an extra (shadow) parameter that specifies how many arguments should be removed
- Instead, the convention is that the caller removes the arguments, because it knows how many there are
  - e.g., it’s easy for a compiler to generate code that does this

Variable of Arguments in C

- Just in case you are curious, here is an example of a C program with a vararg function

```
#include <stdarg.h>
#include <stdio.h>

int func(int first, ...) {
  va_list args;
  va_start(args, first);
  printf("arg #1 = %d\n",first);
  printf("arg #2 = %d\n",va_arg(args, int));
  printf("arg #2 = %s\n",va_arg(args, char*));
  va_end(args);
}
```

```
int main() {
  func(2,(void*)3,(void*)"foo");
}
```

Vararg functions are a bit dangerous. If you call va_arg() more times than there are arguments on the stack, you’ll just get bogus values!

Example: Calling a Subprogram

```
Caller:
    push    dword  2    ; second parameter
    push    dword  1    ; first parameter
    call    func        ; call the function
    add     esp,   8    ; pop the two arguments
```

- Note that to pop the two arguments we merely add 8 to the stack pointer ESP
  - Since we do not care to get the values of the arguments at this point, it’s quicker than to call pop twice!
  - For the case with one argument, calling pop may be better
- The two arguments stay there in memory but will be overwritten next time a function is called or next time the stack is used

Return Values?

- Often, one wants a subprogram to return a value
  - e.g., a function that computes some number
- There are several ways to do this
- One way is to pass as a parameter the address of a zone of memory in which some result should be written
  - As in: void foo(int *x);   foo(&a);
- This is not a *true* return value
  - As in: int foo();
- The C convention is that the return value is always stored in EAX when the function returns
  - It’s the responsibility of the caller to save the EAX value before the call (if needed) and to restore it later
  - In some of our previous example, we just didn’t use EAX to hold anything important so that this issue never arose
    - e.g., when calling read_int(), read_char(), etc.

Recall the NASM Skeleton

```
      ; include directives

segment .data
      ; DX directives

segment .bss
      ; RESX directives

segment .text
        global asm_main
      asm_main:
        enter   0,0
        pusha
        ; Your program here
        popa
        mov     eax, 0
        leave
        ret
```

Returns value 0! (annotation on `mov eax, 0`)

Recall the NASM Skeleton

```
      ; include directives

segment .data
      ; DX directives

segment .bss
      ; RESX directives

segment .text
        global asm_main
      asm_main:
        enter   0,0
        pusha
        ; Your program here
        popa
        mov     eax, 0
        leave
        ret
```

The last two remaining things that we haven't explained yet (but soon) (annotation on `enter 0,0` and `leave`)

In-class Exercise

- What things are wrong with the following program?

```
        push    ebx
        push    30
        call    func
        add     esp, 4
        call    print_int
        call    print_nl
        . . .

func:   push    ebp
        mov     ebp, esp
        mov     eax, [ebp+8]
        add     eax, [ebp+4]
        ret
```

In-class Exercise

- What things are wrong with the following program?

```
        push    ebx
        push    dword 30
        call    func
        add     esp, 8
        call    print_int
        call    print_nl
        . . .

func:   push    ebp
        mov     ebp, esp
        mov     eax, [ebp+12]
        add     eax, [ebp+8]
        pop     ebp
        ret
```

In-class Exercise

- What does the stack look like?

```
        push    ebx
        push    dword 30
        call    func
                <-------------------------------        HERE?
        add     esp, 8
        call    print_int
        call    print_nl
        . . .

func:   push    ebp
        mov     ebp, esp
                <--------------------------------       HERE?
        mov     eax, [ebp+12]
        add     eax, [ebp+8]
        pop     ebp
        ret
```

In-class Exercise

- What does the stack look like?

```
        push    ebx
        push    dword 30
        call    func
                <-----------------------------
        add     esp, 8
        call    print_int
        call    print_nl
        . . .

func:   push    ebp
        mov     ebp, esp
                <-----------------------------
      -
        mov     eax, [ebp+12]
        add     eax, [ebp+8]
        pop     ebp
        ret
```

Stack (first arrow)
EBX
30

Stack (second arrow)
EBX
30
Return @
EBP

A Full Example with Subprograms

- The book has a full example in Section 4.5.1
- Let's do another example here
- Say we want to write a program that first reads in a sequence of 10 integers and then prints the number of integers that are odd
- We will use four functions:
  - get_integers(): get the 10 integers from the user
  - count_odds(): count the number of odd integers
  - is_odd(): determines whether an integer is odd
- We could do this without functions, but
  - The code would most likely be less readable
    - But faster! (usual tradeoff)
- For now, we're writing the code is the most modular and "clean" fashion

Example: Main program

```
%include "asm_io.inc"

segment .data
      msg_odd db      "The number of odd numbers is: ",0

segment .bss
      integers resd   10              ; space to store 10 32-bit integers

segment .text
      global  asm_main
asm_main:
      enter     0,0                   ; set up
      pusha                           ; set up

      push    integers                ; we pass integers to get_integers
      push    dword 10                ; we pass the number of integers to get_integers
      call    get_integers            ; call get_integers
      add     esp, 8                  ; clean up the stack (also doable as "pop ecx" twice)
      mov     eax, msg_odd            ; store the address of the message to print into eax
      call    print_string            ; print the message
      push    integers                ; we pass integers to count_odds
      push    10                      ; we pass the number of integers to count_odds
      call    count_odds              ; call count_odds
      add     esp, 8                  ; clean up the stack
      call    print_int               ; print the content of eax as an integer (this is what count_odds returned)
      call    print_nl                ; print a new line

      popa                            ; clean up
      mov       eax, 0                ; clean up
      leave                           ; clean up
      ret                             ; clean up
```

Piecemeal segment declarations

- The NASM assembler allows for the declaration of multiple .data, .bss, and .text segments
- This makes it possible to declare subprograms in their own region of the .asm file, with parts of .data and .bss segments that are relevant for the subprograms
- Let's look at the get_integers() subprogram

Example: get_integers

```
;       FUNCTION: Get_Integers
;       Takes two parameters:  an address in memory in which to store integers, and a number of integers to store (>0)
;       Destroys values of eax, ebx, and ecx!!

segment .data
      msg_int        db      "Enter an integer: ",0

segment .text
get_integers:
      push    ebp                     ; save the value of EBP of the caller
      mov     ebp, esp                ; update the value of EBP for this subprogram

      mov     ecx, [ebp + 12]         ; ECX = address at which to store the integers (parameter #2)
      mov     ebx, [ebp + 8]          ; EBX = number of integers to read (parameter #1)
      shl     ebx, 2                  ; EBX = EBX * 4  (unsigned)
      add     ebx, ecx                ; EBX = ECX + EBX = address beyond that of the last integer to be stored
loop1:
      mov     eax, msg_int            ; EAX = address of the message to print
      call    print_string;           ; print the message
      call    read_int                ; read an integer from the keyboard (which will be stored in EAX)
      mov     [ecx], eax              ; store the integer in memory at the correct address
      add     ecx, 4                  ; ECX = ECX + 4
      cmp     ecx, ebx                ; compare ECX, EBX
      jb      loop1                   ; if ECX < EBX, jump to loop1   (unsigned)

      pop     ebp                     ; restore the value of EBP
      ret                             ; clean up
```

Example: count_odds

```
;       FUNCTION: count_odds
;       Takes two parameters:  an address in memory in which integers are stored, and the number of integers (>0)
;       Destroys values of eax, ebx, and edx!!  (eax = returned value)

segment .text
count_odds:
      push    ebp                     ; save the value of EBP of the caller
      mov     ebp, esp                ; update the value of EBP for this subprogram

      mov     eax, [ebp + 12]         ; EAX = address at which integers are stored (parameter #2)
      mov     ebx, [ebp + 8]          ; EBX = number of integers (parameter #1)
      shl     ebx, 2                  ; EBX = EBX * 4  (unsigned)
      add     ebx, eax                ; EBX = EAX + EBX = address beyond that of the last integer
      sub     ebx, 4                  ; EBX = EBX - 4 = address of the last integer
      xor     edx, edx                ; EDX = 0 = number of odd integers
loop2:

      push    dword [ebx]             ; store the current integer on the stack
      call    is_odd                  ; call is_odd
      add     esp, 4                  ; clean up the stack
      add     edx, eax                ; EDX += EAX  (EAX = 0 if even, EAX = 1 if odd)
      sub     ebx, 4                  ; EBX = EBX - 4
      cmp     ebx, [ebp+12]           ; compare EBX and the address of the first integer
      jnb     loop2                   ; if EBX >= [EBP+12]  jump to loop2   (unsigned)

      mov     eax, edx                ; EAX = EDX (= number of odd integers)

      pop     ebp                     ; restore the value of EBP
      ret                             ; clean up
```

Example: is_odd

```
;       FUNCTION: is_odd
;       Takes one parameter:  an integers (>0)
;       Destroys values of eax and ecx (eax = returned value)

segment .text
is_odd:
      push    ebp                     ; save the value of EBP of the caller
      mov     ebp, esp                ; update the value of EBP for this subprogram

      mov     eax, 0                  ; EAX = 0
      mov     ecx, [ebp+8]            ; EBX = integer (parameter #1)
      shr     ecx, 1                  ; Right logical shift
      adc     eax, 0                  ; EAX = EAX + carry  (if even: EAX = 0, if odd: EAX = 1)

      pop     ebp                     ; restore the value of EBP
      ret                             ; clean up
```

Destroyed Registers?

- Note that in the previous program we have added comments specifying which registers are destroyed
- The caller is then responsible for making sure that its registers are not corrupted
- One way to ensure this is to save them somewhere in memory, for instance on the stack
- However, in a program that has many functions it becomes really annoying to constantly have to pay attention to what needs to be saved and what doesn't
- The typical approach is to have the subprogram save what it knows needs to be saved
  - And comment that the caller doesn't need to worry about anything

Saving Registers in Subprograms

- Just saving EBP

```
func:
    push    ebp          ; save original EBP
    mov     ebp, esp     ; set EBP = ESP

    . . .                ; subprogram code

    mov     eax, ...     ; set return value

    pop     ebp          ; restore original EBP
    ret                  ; returns
```

Saving Registers in Subprograms

- Saving EBX and ECX in addition to EBP

```
func:
    push    ebp          ; save original EBP
    mov     ebp, esp     ; set EBP = ESP
    push    ebx          ; save EBX
    push    ecx          ; save ECX

    . . .                ; subprogram code

    mov     eax, ...     ; set return value

    pop     ecx          ; restore ECX
    pop     ebx          ; restore EBX
    pop     ebp          ; restore ebp
    ret                  ; returns
```

Saving Registers in Subprograms

- Saving "all" registers using PUSHA and POPA

```
func:
    push    ebp          ; save original EBP
    mov     ebp, esp     ; set EBP = ESP
    pusha                ; save all (including new EBP)

    . . .                ; subprogram code

    mov     eax, ...     ; set return value

    popa                 ; restore all (including new EBP)
    pop     ebp          ; restore original ebp
    ret                  ; returns
```

Problem?

Overwrites the return value that's stored in eax!

Saving Registers in Subprograms

- Saving "all" registers using PUSHA and POPA, a good option

```
.bss:
    returnvalue  resd   1        ; place in memory for the return value
func:
    push    ebp                  ; save original EBP
    mov     ebp, esp             ; set EBP = ESP
    pusha                        ; save all (including new EBP)

    . . .                        ; subprogram code

    mov     [returnvalue], eax   ; save return value in memory

    popa                         ; restore all (including new EBP)
    mov     eax, [returnvalue]   ; retrieve the saves return value
                                 ; (as done in our skeleton)
    pop     ebp                  ; restore original ebp
    ret                          ; returns
```

Recursion

- The subprogram calling conventions we have just described enable recursion
- Let's see this on an example program that computes the sum of the first n integers
  - Yes, it's n(n+1)/2, and even if we didn't know that an iterative program would be more efficient, but for the sake of this example let's just write a recursive program to compute it

Example: Recursive Program

```
. . .
segment .data
    msg1        db          'Enter n: ', 0
    msg2        db          'The sum is: ', 0
segment .text
    . . .                               ; declaration of asm_main and setup

    mov         eax, msg1               ; eax = address of msg1
    call        print_string            ; print msg1
    call        read_int                ; get an integer from the keyboard (in EAX)
    push        eax                     ; put the integer on the stack (parameter #1)
    call        recursive_sum           ; call recursive_sum
    pop         ebx                     ; remove the parameter from the stack
    mov         ebx, eax                ; save the value returned by recursive_sum
    mov         eax, msg2               ; eax = address of msg2
    call        print_string            ; print msg2
    mov         eax, ebx                ; eax = sum
    call        print_int               ; print the sum
    call        print_nl                ; print a new line

    . . .                               ; cleanup
```

Example: recursive_sum()

```
;
segment .bss
    value           resd, 1             ; to store the return value temporarily
segment .text
recursive_sum
    push            ebp                 ; save ebp
    mov             ebp, esp            ; set EBP = ESP
    pusha                               ; save all registers (probably overkill)
    mov             ebx, [ebp+8]        ; ebx = integer (parameter #1)
    cmp             ebx, 0              ; ebx = 0 ?
    jnz             next                ; if (ebx != 0) go to next
    xor             ecx, ecx            ; ECX = 0
    jmp             end                 ; Jump to end
next:
    mov             ecx, ebx            ; ECX = EBX
    dec             ecx                 ; ECX = ECX - 1
    push            ecx                 ; put ECX on the stack
    call            recursive_sum       ; recursive call to recursive_sum!
    pop             edx                 ; pop the parameter from the stack
    add             ebx, eax            ; EBX = EBX + recursive_sum(EBX -1)
    mov             ecx, ebx            ; ECX = EBX
end:                                    ; at this point, ECX contains the result
    mov             [value], ecx        ; save ECX, the return value, in memory
    popa                                ; restore registers
    mov             eax, [value]        ; put the saved returned value into eax
    pop             ebp                 ; restore EBP
    ret                                 ; return
```

Local Variables in Subprograms

- In all the examples we have seen so far, the subprograms were able to do their work using only registers
- But sometimes, a subprogram's needs are beyond the set of available registers and some data must be kept in memory
  - Just think of all subprograms you wrote that used more than 6 local variables (EAX, EBX, ECX, EDX, ESI, EDI)
- One possibility could be to declare a small .bss segment for each subprogram, to reserve memory space for all local variables
- Drawback #1: memory waste
  - This reserved memory consumes memory space for the entire duration of the execution even if the subprogram is only active for a tiny fraction of the execution time
- Drawback #2: subprogram are not reentrant

Re-entrant subprogram

- A subprogram is active if it has been called and if its RET instruction hasn't been executed yet
- A subprogram is reentrant if it can be called from anywhere
- This implies that the program can call itself, directly or indirectly, which enables recursion
  - e.g., f calls g, which calls h, which calls f
- This means that at a given point in time, two or more instances of a subprogram can be active
  - Two or more activation records for this subprogram on the stack
- If we store the local variables of a subprogram in the .bss segment, then there can only be one activation!
  - Otherwise the second activation could corrupt the local variables of the first activation
- Therefore, with our current scheme for storing local variables, programs are not reentrant and one cannot have recursive calls when subprograms have local variables!
  - In our previous example the recursive program had no local variables
- Having reentrant programs is typically a very useful thing and we don't want to live without it

Local variables on the stack

- Since activation records on the stack are used to store relevant information pertaining to a subprogram, why not use it for storing the subprogram local variables?
- The standard approach is to store local variables right after the saved EBP value on the stack
  - This is simply done by subtracting some amount to the ESP pointer
- The local variables are then accessed as [EBP - 4], [EBP - 8], etc.
- Let's see this on an example

Local Variable Examples

- Say we have a subprogram that takes 2 parameters, uses 3 local variables, and doesn't return any value
- The code of the subprogram is as follows:

```
func:
    push    ebp             ; save old EBP value
    mov     ebp, esp        ; set EBP
    sub     esp, 12         ; add space for 3 local variables
    ; subprogram body
    mov     esp, ebp        ; deallocate local variables
    pop     ebp             ; restore old EBP value
    ret
```

- Let's look at the content of the stack when the subprogram body begins

Local Variables Example

- Inside the body of the subprogram, parameters are referenced as:
  - [EBP+12]: 2nd parameter
  - [EBP+8]: 1st parameter
- Inside the body of the subprogram, local variables are referenced as:
  - [EBP-4]: 1st local variable
  - [EBP-8]: 2nd local variable
  - [EBP-12]: 3rd local variable

Address	Content
EBP+12	2nd parameter
EBP+8	1st parameter
EBP+4	return address
EBP	saved EBP
EBP-4	1st local var
EBP-8	2nd local var
EBP-12	3rd local var

ENTER and LEAVE

- We always have the same *prologue* and the same *epilogue*

```
push    ebp         ; save old EBP value
mov     ebp, esp    ; set EBP
sub     esp, X      ; reserve X=4*N bytes for N local vars
```

```
mov     esp, ebp    ; remove space for local vars
pop     ebp         ; restore old EBP value
ret                 ; return
```

ENTER and LEAVE

- There are two convenient functions: ENTER and LEAVE

```
push    ebb         ; save old EBP value
mov     ebp, esp    ; set EBP
sub     esp, X      ; reserve X=4*N bytes for N local vars
```

equivalent to

```
enter   X, 0
```

```
mov     esp, ebp    ; remove space for local vars
pop     ebp         ; restore old EBP value
ret                 ; return
```

equivalent to

```
leave
ret
```

Recall the NASM Skeleton

```
    ; include directives

segment .data
    ; DX directives

segment .bss
    ; RESX directives

segment .text
        global asm_main
    asm_main:
        enter   0,0
        pusha
        ; Your program here
        popa
        mov     eax, 0
        leave
        ret
```

Prologue and epilogue of asm_main

We Finally Understand the Skeleton

```
    ; include directives

segment .data
    ; DX directives

segment .bss
    ; RESX directives

segment .text
        global asm_main
    asm_main:
        enter   0,0         ; Save EBP, reserve 0 bytes for local variables
        pusha               ; Save ALL registers
        ; Your program here
        popa                ; Restore ALL registers
        mov     eax, 0      ; Set the return value to 0
        leave               ; Restore EBP, remove space for local variables
        ret                 ; Pop the return address and jump to it
```

Knowing your stack

- At this point it should be clear that it is very important to understand how the stack works and how to use is very clearly
- When programming you should always have a mental picture of the stack
  - Something you don't do when using a high-level programming language typically
    - which is actually not good and separates good programmers from others
- It's typically a good idea to be consistent
  - Compilers are consistent by design

A Full Example

- Let's write the assembly code equivalent to the following 2 C functions

```
int   f(int num) {   // computes Fibonacci numbers
    int x, sum;
    if (num == 0) return 0;
    if (num == 1) return 1;
    x = f(num-1);
    sum = x + f(num-2)
    return sum;
}
```

- Let's write a "straight" translation, without optimizing variables away, just for demonstration purposes

A Full Example

```
%include "asm_io.inc"

segment .data
      msg1        db    "Enter n: ", 0
      msg2        db    "The result is: ", 0

segment .text
      global  asm_main
asm_main:
      enter   0,0                   ; set up
      pusha                         ; set up

      mov     eax, msg1             ; eax = address of msg1
      call    print_string          ; print msg1
      call    read_int              ; get an integer from the keyboard (in EAX)
      push    eax                   ; put the integer on the stack (parameter #1)
      call    f                     ; call recursive_sum
      pop     ebx                   ; remove the parameter from the stack
      mov     ebx, eax              ; save the value returned by recursive_sum
      mov     eax, msg2             ; eax = address of msg2
      call    print_string          ; print msg2
      mov     eax, ebx              ; eax = sum
      call    print_int             ; print the sum
      call    print_nl              ; print a new line

      popa                          ; clean up
      mov     eax, 0                ; clean up
      leave                         ; clean up
      ret                           ; clean up
```

A Full Example

```
;     FUNCTION: f
;     Takes one parameter:  an integer
;     eax = return value
segment .data
      debug1  db    "Function f called with integer: ",0
segment .text
f:    enter     2,0               ; num in [ebp+8], local var x in [ebp-4], local var sum in [ebp-8]
      push      ebx               ; save ebx
      push      ecx               ; save ecx
      push      edx               ; save edx

      mov       eax, [ebp+8]      ; eax = num
      sub       eax, 2            ; eax -= 2
      jns       next              ; if not <0, goto next
      add       eax, 2            ; eax += 2
      jmp       end
next:
      mov       eax, [ebp+8]      ; eax = num
      add       eax, -1           ; eax -= 1
      push      eax               ; put (num -1) on the stack
      call      f                 ; call f (recursively)
      add       esp, 4            ; remove (num-1) from the stack
      mov       [ebp-4], eax      ; put the returned value in x
      mov       eax, [ebp+8]      ; eax = num
      add       eax, -2           ; eax -= 2
      push      eax               ; put (num -2) on the stack
      call      f                 ; call f (recursively), the return value is in eax
      add       esp, 4            ; remove (num-1) from the stack
      add       eax, [ebp-4]      ; eax += x

end:
      pop       edx               ; restore ebx
      pop       ecx               ; restore ecx
      pop       ebx               ; restore edx
      leave                       ; clean up the stack
      ret                         ; return
```

Interfacing Assembly and C

- Section 4.7 of the book talks about interfacing C and assembly
- We have seen most of this content already, but let's talk about the issue of saving registers on the stack
- By convention, C assumes that a subprogram (e.g., the one you're writing in assembly), will not destroy values in EBX, ESI, EDI, EBP, CS, DS, SS, and ES
- So, if you write an assembly subprogram, make sure you save these on the stack and restore them
  - We've already said we save EBP
- Example: I know my subprogram uses EBX (as on page 86)

```
enter    4,0            ; prologue (1 32-bit local var)
push     ebx            ; save EBX
. . .
pop ebx                 ; restore ECX
leave                   ; epilogue
ret                     ; return
```

In-Class Quiz

- We'll have an in-class quiz on this set of slides next ...

Conclusion

- When programming one always faces trade-offs between program readability and program performance
  - Choices must be made based on the task at hand
- With by-hand assembly programming, the programmer can make fine-tuned decisions for these trade-offs
  - e.g., for a particular function I decide to not save all registers because I _know_ that it won't corrupt them, thus saving a bit of time
  - e.g., I know that I can reuse some register value that was modified in a subprogram to do some clever optimization
- Some of these optimizations can only be done by a human who understands what the program does
- Some of these optimizations can sometimes be done by a compiler that generates assembly code from a program written in some high-level language