Subprograms: Arguments

ICS312

Machine-Level and Systems Programming

Henri Casanova (henric@hawaii.edu)

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

- When writing assembly, you could 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, follows a standard method to generate assembly for all function calls
- 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 must use this convention when generating assembly code from C code
  - We'll also use this convention when writing assembly by hand

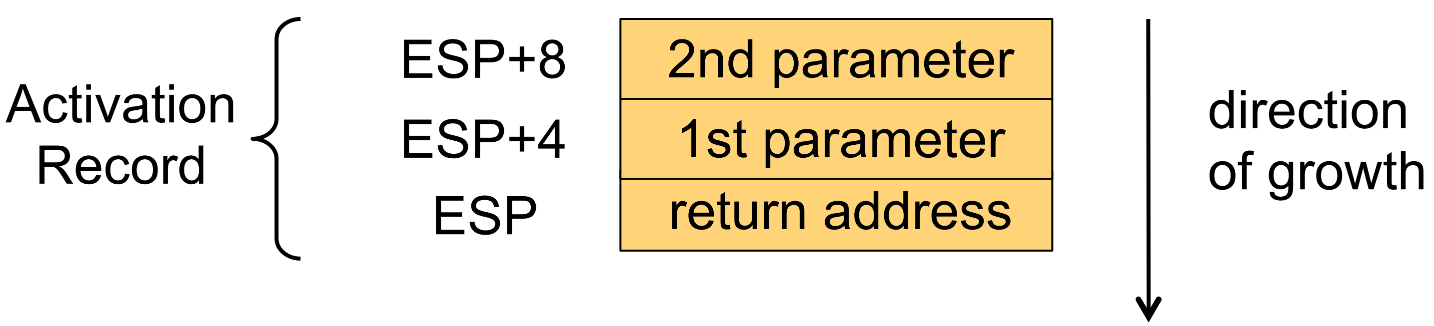
A Simple Activation Record

- To call a function you have to follow these 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
  - a is pushed onto the stack third
  - Makes sense: the first pop should get the first parameter

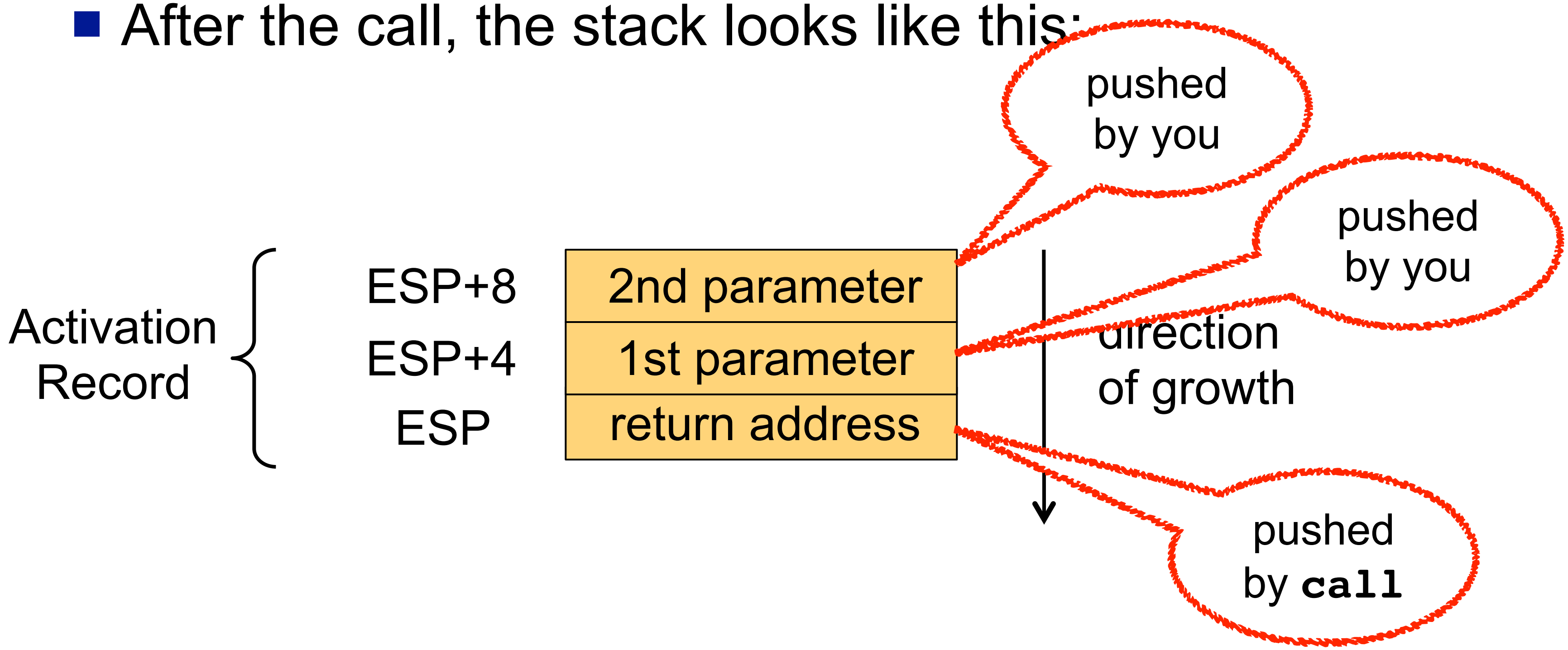
A Simple Activation Record

- Say you want to call a function with two 32-bit parameters
  - If parameters are < 32 bits, they need to be extended to 32-bit values, at least in this course
- After the call, the stack looks like this:



A Simple Activation Record

- Say you want to call a function with two 32-bit parameters
  - If parameters are < 32 bits, they need to be extended to 32-bit values, at least in this course
- After the call, the stack looks like this:



Using the Parameters

- Inside the code of the subprogram, parameters can be accessed via indirection from the stack pointer
- In our previous example:
  - mov eax, [ESP + 4] ; puts 1st parameter into eax
  - mov ebx, [ESP + 8] ; puts 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
    - They may be copied into registers for performance reasons
    - But we can always get their original values from the stack

Accessing the stack in C

```
void main(int x) {
  x++;   // Would be translated: inc [esp + 4]
}
```

- The activation record on the stack is the subprogram's little play pen
  - And yes, you can add one to the parameter as seen above, just as if it were a local variable
- The subprogram can do whatever, and eventually its activation record is wiped out anyway

- But, turns out, there is still a problem…

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 as an anchor 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

ESP+8	2nd parameter
ESP+4	1st parameter
ESP	return address

ESP and EBP

- Stack as it is when the subprogram begins

ESP+8	2nd parameter
ESP+4	1st parameter
ESP	return address

- EBP = ESP

EBP+8	2nd parameter
EBP+4	1st parameter
EBP = ESP	return address

ESP and EBP

- Stack as it is when the subprogram begins

ESP+8	2nd parameter
ESP+4	1st parameter
ESP	return address

- EBP = ESP

EBP+8	2nd parameter
EBP+4	1st parameter
EBP = ESP	return address

- Further use of the stack

ESP+16	EBP+8	2nd parameter
ESP+12	EBP+4	1st parameter
ESP+8	EBP	return address
	ESP+4	stuff
	ESP	stuff

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

ESP and EBP Mayhem

- **Big problem:** The **caller** may have been using EBP!
  - Typically to access its own parameters!!!
- So you can’t just overwrite EBP with what you need in it (you: a subprogram being called)
- Because when you return, the caller will have a wrong EBP and will access its own parameters erroneously
- How do we deal with having to save stuff?
- We use the stack!!

Saving EBP on the Stack

- 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:

ESP+12	2nd parameter
ESP+8	1st parameter
ESP+4	return address
EBP = ESP	old value of EBP

- 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 my caller's EBP
    mov     ebp, esp    ; set EBP = ESP

    . . .               ; subprogram code

    pop     ebp         ; restore my caller's EBP
    ret                 ; returns
```

Returning from a Subprogram

- After the subprogram finishes, one must “clean up” the stack
- The stack has on it:
  - The old EBP value, the return address, the parameters
- The old EBP value is 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
- The parameters need to be removed from the stack
- The C convention specifies that the caller code must remove the parameters from the stack
  - 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*
  - *Let’s go into a bit of a detour…. if you’re confused already, you can safely skip the next 2 slides when you study this content*

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 always 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 #3 = %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!
  - This is one case in which we do modify ESP directly
- The two arguments stay there in memory but will be overwritten next time a function is called or next time the stack is used
  - We don't zero out "old" value, we just lazily overwrite them later

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

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

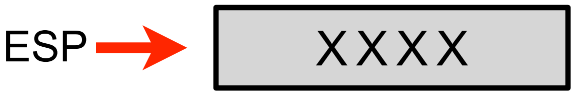
Recall the NASM Skeleton

```
        ; 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)

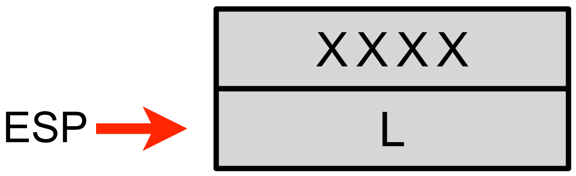
A Full Example

```
L        dd  42, 43, 44, 45, 56
...
push        dword L
call        func
add         esp, 4
call        print_int
...
func:
        push      ebp
        mov       ebp, esp
        push      [ebp+8]
        push      8
        call      reference
        add       esp, 8
        add       eax, 10
        pop       ebp
        ret
reference:
        push      ebp
        mov       ebp, esp
        mov       eax, [ebp+12]
        add       eax, [ebp+8]
        mov       eax, [eax]
        pop       ebp
        ret
```



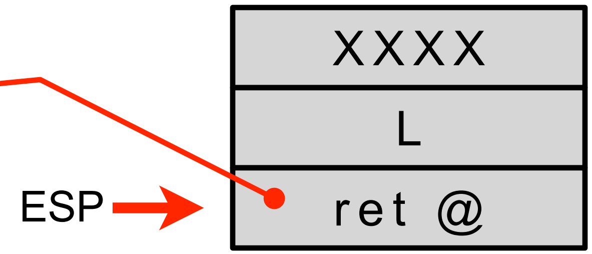
A Full Example

```
L       dd  42, 43, 44, 45, 56
...
push        dword L
call        func
add         esp, 4
call        print_int
...
func:
        push      ebp
        mov       ebp, esp
        push      [ebp+8]
        push      8
        call      reference
        add       esp, 8
        add       eax, 10
        pop       ebp
        ret
reference:
        push      ebp
        mov       ebp, esp
        mov       eax, [ebp+12]
        add       eax, [ebp+8]
        mov       eax, [eax]
        pop       ebp
        ret
```



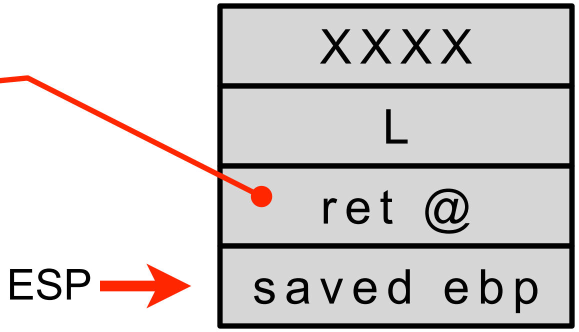
A Full Example

```
L       dd  42, 43, 44, 45, 56
...
push        dword L
call        func
add         esp, 4
call        print_int
...
func:
        push      ebp
        mov       ebp, esp
        push      [ebp+8]
        push      8
        call      reference
        add       esp, 8
        add       eax, 10
        pop       ebp
        ret
reference:
        push      ebp
        mov       ebp, esp
        mov       eax, [ebp+12]
        add       eax, [ebp+8]
        mov       eax, [eax]
        pop       ebp
        ret
```



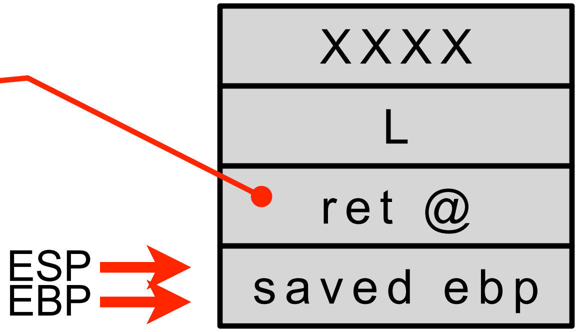
A Full Example

```
L        dd  42, 43, 44, 45, 56
...
push         dword L
call         func
add          esp, 4
call         print_int
...
func:
         push       ebp
         mov        ebp, esp
         push       [ebp+8]
         push       8
         call       reference
         add        esp, 8
         add        eax, 10
         pop        ebp
         ret
reference:
         push       ebp
         mov        ebp, esp
         mov        eax, [ebp+12]
         add        eax, [ebp+8]
         mov        eax, [eax]
         pop        ebp
         ret
```



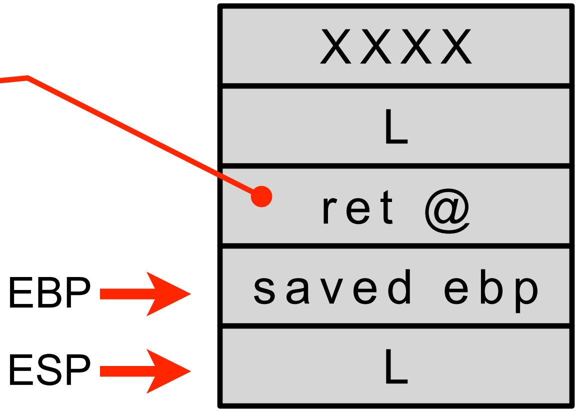
A Full Example

```
L       dd  42, 43, 44, 45, 56
...
push        dword L
call        func
add         esp, 4
call        print_int
...
func:
        push     ebp
        mov      ebp, esp
        push     [ebp+8]
        push     8
        call     reference
        add      esp, 8
        add      eax, 10
        pop      ebp
        ret
reference:
        push     ebp
        mov      ebp, esp
        mov      eax, [ebp+12]
        add      eax, [ebp+8]
        mov      eax, [eax]
        pop      ebp
        ret
```



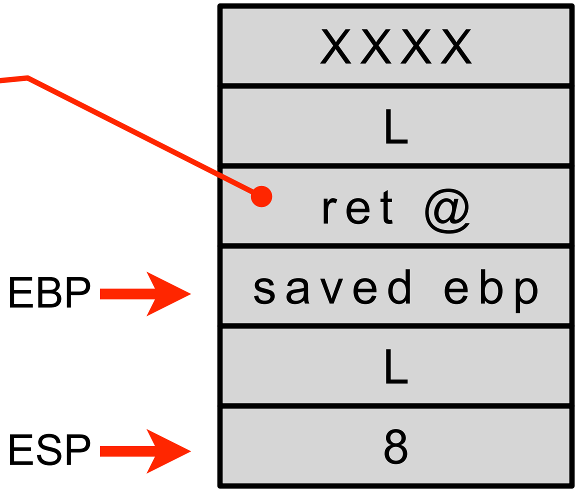
A Full Example

```
L       dd  42, 43, 44, 45, 56
...
push        dword L
call        func
add         esp, 4
call        print_int
...
func:
        push      ebp
        mov       ebp, esp
        push      [ebp+8]
        push      8
        call      reference
        add       esp, 8
        add       eax, 10
        pop       ebp
        ret
reference:
        push      ebp
        mov       ebp, esp
        mov       eax, [ebp+12]
        add       eax, [ebp+8]
        mov       eax, [eax]
        pop       ebp
        ret
```



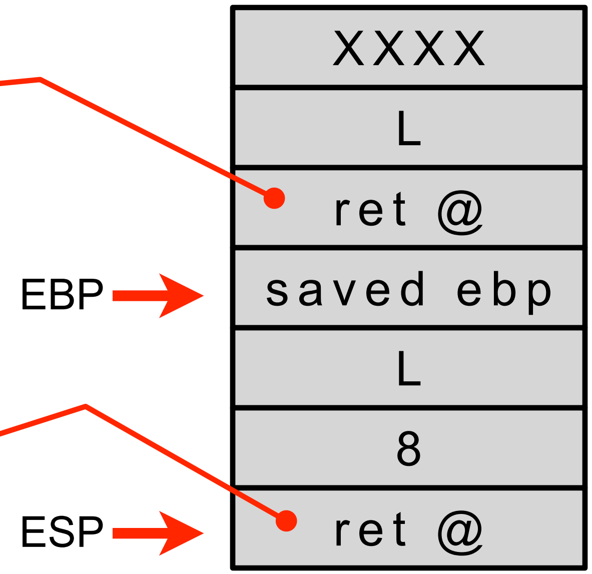
A Full Example

```
L       dd  42, 43, 44, 45, 56
...
push        dword L
call        func
add         esp, 4
call        print_int
...
func:
        push        ebp
        mov         ebp, esp
        push        [ebp+8]
        push        8
        call        reference
        add         esp, 8
        add         eax, 10
        pop         ebp
        ret
reference:
        push        ebp
        mov         ebp, esp
        mov         eax, [ebp+12]
        add         eax, [ebp+8]
        mov         eax, [eax]
        pop         ebp
        ret
```



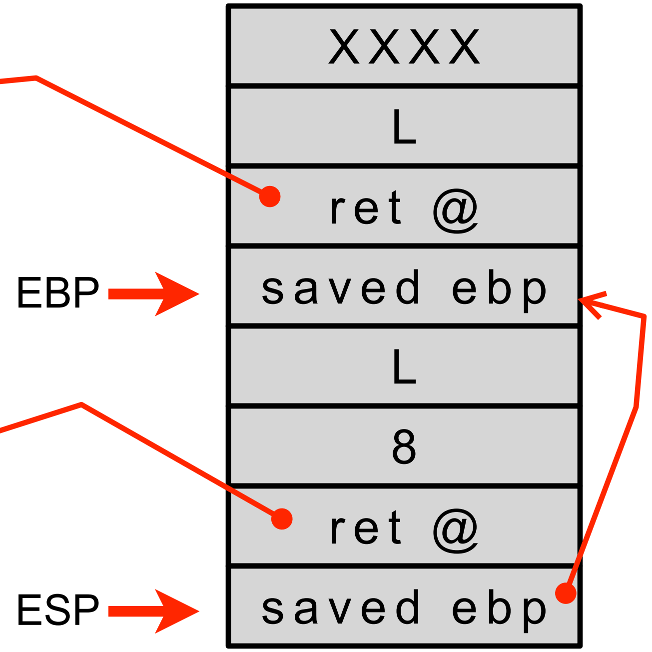
A Full Example

```
L       dd  42, 43, 44, 45, 56
...
push        dword L
call        func
add         esp, 4
call        print_int
...
func:
        push      ebp
        mov       ebp, esp
        push      [ebp+8]
        push      8
        call      reference
        add       esp, 8
        add       eax, 10
        pop       ebp
        ret
reference:
        push      ebp
        mov       ebp, esp
        mov       eax, [ebp+12]
        add       eax, [ebp+8]
        mov       eax, [eax]
        pop       ebp
        ret
```



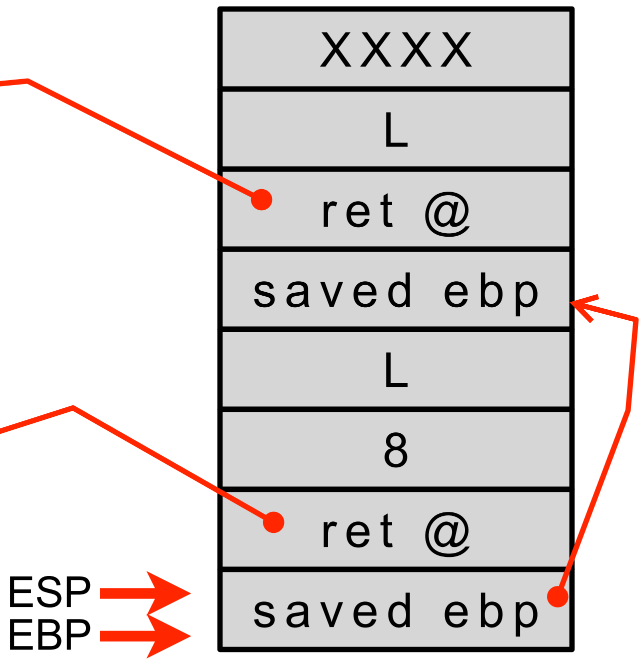
A Full Example

```
L       dd  42, 43, 44, 45, 56
...
push        dword L
call        func
add         esp, 4
call        print_int
...
func:
        push     ebp
        mov      ebp, esp
        push     [ebp+8]
        push     8
        call     reference
        add      esp, 8
        add      eax, 10
        pop      ebp
        ret
reference:
        push     ebp
        mov      ebp, esp
        mov      eax, [ebp+12]
        add      eax, [ebp+8]
        mov      eax, [eax]
        pop      ebp
        ret
```



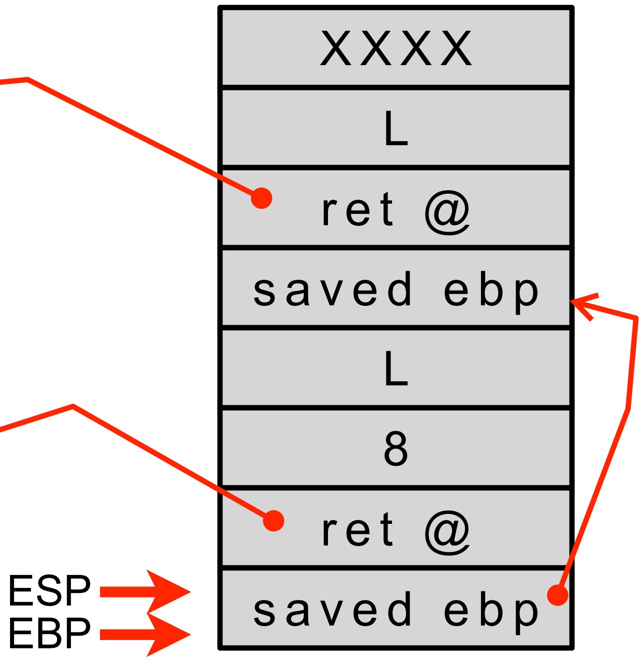
A Full Example

```
L        dd  42, 43, 44, 45, 56
...
push        dword L
call        func
add         esp, 4
call        print_int
...
func:
        push      ebp
        mov       ebp, esp
        push      [ebp+8]
        push      8
        call      reference
        add       esp, 8
        add       eax, 10
        pop       ebp
        ret
reference:
        push      ebp
        mov       ebp, esp
        mov       eax, [ebp+12]
        add       eax, [ebp+8]
        mov       eax, [eax]
        pop       ebp
        ret
```



A Full Example

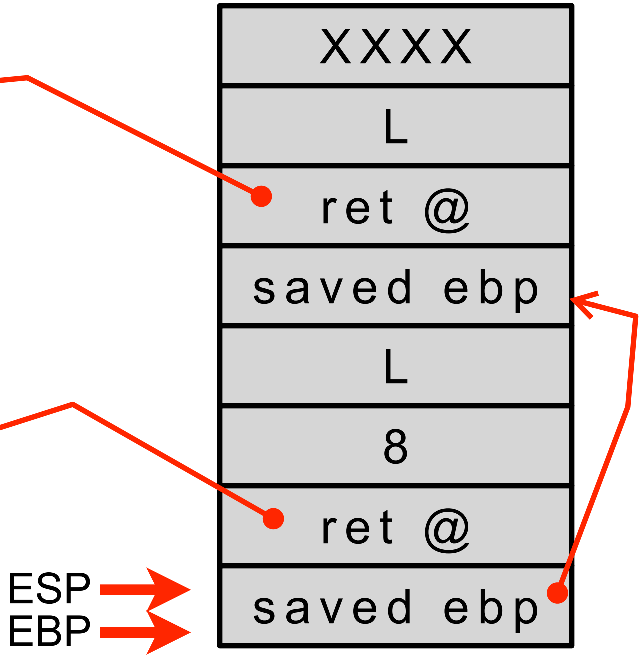
```
L       dd  42, 43, 44, 45, 56
...
push        dword L
call        func
add         esp, 4
call        print_int
...
func:
        push      ebp
        mov       ebp, esp
        push      [ebp+8]
        push      8
        call      reference
        add       esp, 8
        add       eax, 10
        pop       ebp
        ret
reference:
        push      ebp
        mov       ebp, esp
        mov       eax, [ebp+12]
        add       eax, [ebp+8]
        mov       eax, [eax]
        pop       ebp
        ret
```



EAX = L

A Full Example

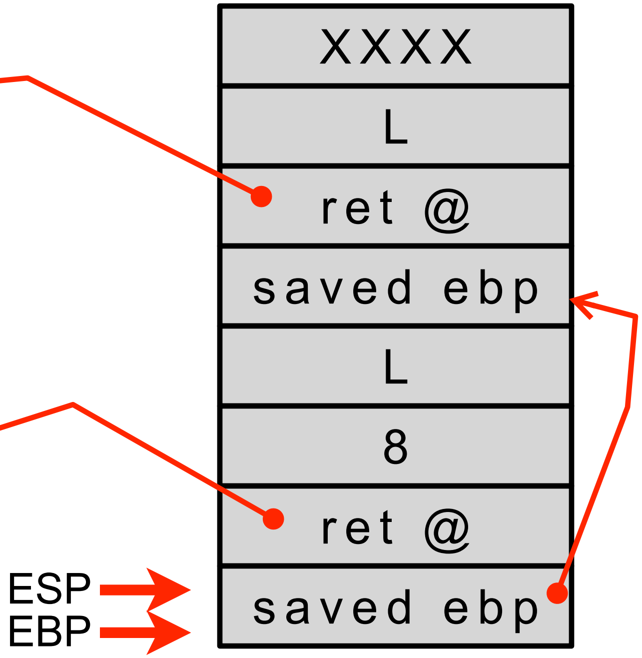
```
L       dd  42, 43, 44, 45, 56
...
push        dword L
call        func
add         esp, 4
call        print_int
...
func:
        push      ebp
        mov       ebp, esp
        push      [ebp+8]
        push      8
        call      reference
        add       esp, 8
        add       eax, 10
        pop       ebp
        ret
reference:
        push      ebp
        mov       ebp, esp
        mov       eax, [ebp+12]
        add       eax, [ebp+8]
        mov       eax, [eax]
        pop       ebp
        ret
```



EAX = L + 8

A Full Example

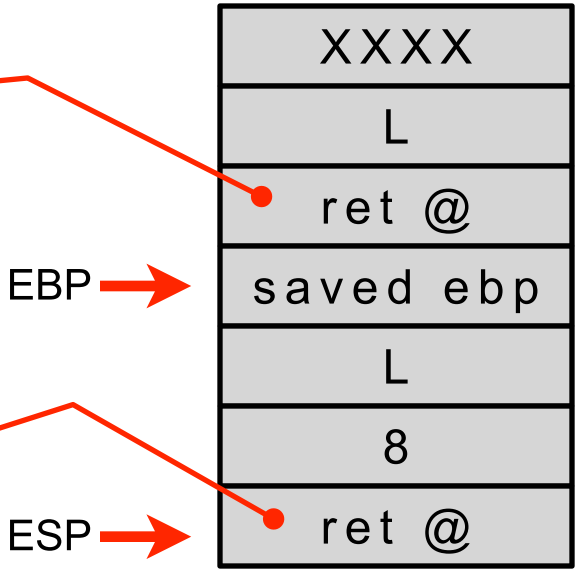
```
L       dd  42, 43, 44, 45, 56
...
push        dword L
call        func
add         esp, 4
call        print_int
...
func:
        push      ebp
        mov       ebp, esp
        push      [ebp+8]
        push      8
        call      reference
        add       esp, 8
        add       eax, 10
        pop       ebp
        ret
reference:
        push      ebp
        mov       ebp, esp
        mov       eax, [ebp+12]
        add       eax, [ebp+8]
        mov       eax, [eax]
        pop       ebp
        ret
```



EAX = [L + 8] = 44

A Full Example

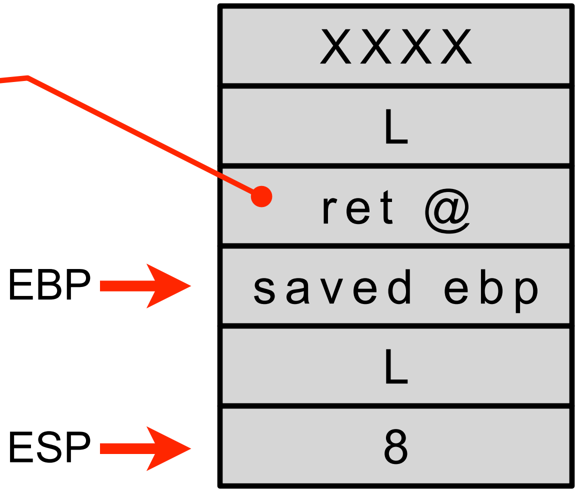
```
L         dd  42, 43, 44, 45, 56
...
push        dword L
call        func
add         esp, 4
call        print_int
...
func:
        push      ebp
        mov       ebp, esp
        push      [ebp+8]
        push      8
        call      reference
        add       esp, 8
        add       eax, 10
        pop       ebp
        ret
reference:
        push      ebp
        mov       ebp, esp
        mov       eax, [ebp+12]
        add       eax, [ebp+8]
        mov       eax, [eax]
        pop       ebp
        ret
```



EAX = 44

A Full Example

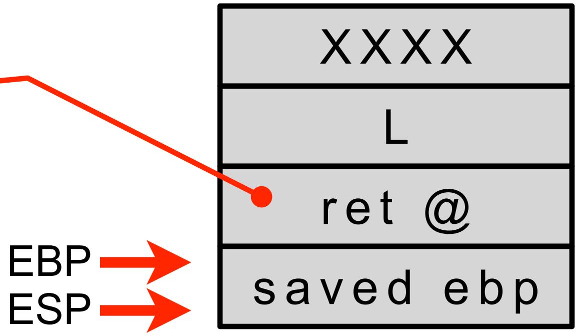
```
L         dd  42, 43, 44, 45, 56
...
push         dword L
call         func
add          esp, 4
call         print_int
...
func:
        push      ebp
        mov       ebp, esp
        push      [ebp+8]
        push      8
        call      reference
        add       esp, 8
        add       eax, 10
        pop       ebp
        ret
reference:
        push      ebp
        mov       ebp, esp
        mov       eax, [ebp+12]
        add       eax, [ebp+8]
        mov       eax, [eax]
        pop       ebp
        ret
```



EAX = 44

A Full Example

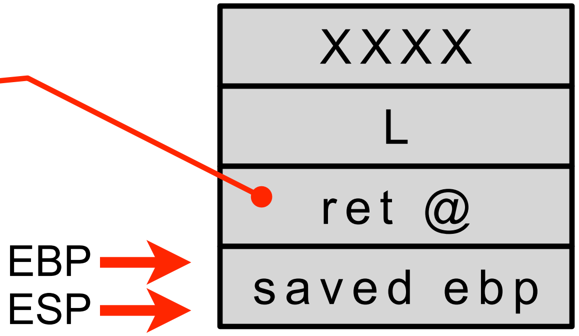
```
L        dd  42, 43, 44, 45, 56
...
push         dword L
call         func
add          esp, 4
call         print_int
...
func:
        push      ebp
        mov       ebp, esp
        push      [ebp+8]
        push      8
        call      reference
        add       esp, 8
        add       eax, 10
        pop       ebp
        ret
reference:
        push      ebp
        mov       ebp, esp
        mov       eax, [ebp+12]
        add       eax, [ebp+8]
        mov       eax, [eax]
        pop       ebp
        ret
```



EAX = 44

A Full Example

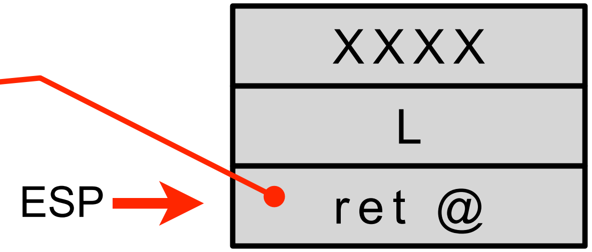
```
L       dd  42, 43, 44, 45, 56
...
push        dword L
call        func
add         esp, 4
call        print_int
...
func:
        push        ebp
        mov         ebp, esp
        push        [ebp+8]
        push        8
        call        reference
        add         esp, 8
        add         eax, 10
        pop         ebp
        ret
reference:
        push        ebp
        mov         ebp, esp
        mov         eax, [ebp+12]
        add         eax, [ebp+8]
        mov         eax, [eax]
        pop         ebp
        ret
```



EAX = 44 + 10 = 54

A Full Example

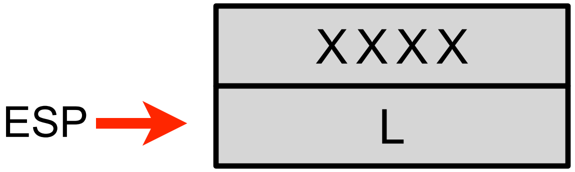
```
L       dd  42, 43, 44, 45, 56
...
push        dword L
call        func
add         esp, 4
call        print_int
...
func:
        push      ebp
        mov       ebp, esp
        push      [ebp+8]
        push      8
        call      reference
        add       esp, 8
        add       eax, 10
        pop       ebp
        ret
reference:
        push      ebp
        mov       ebp, esp
        mov       eax, [ebp+12]
        add       eax, [ebp+8]
        mov       eax, [eax]
        pop       ebp
        ret
```



EAX = 54

A Full Example

```
L       dd  42, 43, 44, 45, 56
...
push        dword L
call        func
add         esp, 4
call        print_int
...
func:
        push      ebp
        mov       ebp, esp
        push      [ebp+8]
        push      8
        call      reference
        add       esp, 8
        add       eax, 10
        pop       ebp
        ret
reference:
        push      ebp
        mov       ebp, esp
        mov       eax, [ebp+12]
        add       eax, [ebp+8]
        mov       eax, [eax]
        pop       ebp
        ret
```



EAX = 54

A Full Example

```
L       dd  42, 43, 44, 45, 56
...
push        dword L
call        func
add         esp, 4
call        print_int
...
func:
        push      ebp
        mov       ebp, esp
        push      [ebp+8]
        push      8
        call      reference
        add       esp, 8
        add       eax, 10
        pop       ebp
        ret
reference:
        push      ebp
        mov       ebp, esp
        mov       eax, [ebp+12]
        add       eax, [ebp+8]
        mov       eax, [eax]
        pop       ebp
        ret
```

ESP → XXXX

EAX = 54

A Full Example

```
L        dd  42, 43, 44, 45, 56
...
push        dword L
call        func
add         esp, 4
call        print_int
...
func:
        push     ebp
        mov      ebp, esp
        push     [ebp+8]
        push     8
        call     reference
        add      esp, 8
        add      eax, 10
        pop      ebp
        ret
reference:
        push     ebp
        mov      ebp, esp
        mov      eax, [ebp+12]
        add      eax, [ebp+8]
        mov      eax, [eax]
        pop      ebp
        ret
```

ESP → XXXX

prints “54”

C Translation of the previous program (reverse-engineering)

```
#include <stdio.h>
int     L[5] = {42, 43, 44, 45, 56};

int func(int *array);
int reference(int a, int *ptr);

int main(int argc, char **argv) {
 // ...
 printf("%d", func(L));
 // ...
}

int func(int *array) {
  return 10 + reference(8, array);
}
```

In-class Exercise

- What 4 things are wrong with the following program?

```
        push    ebx
        push    dword 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 4 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
                <------------------------------        THERE?
        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
```

Zoom poll…

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
```

xxxxxx
EBX
30

xxxxxx
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 odd integers
- We will use three 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
  - The code would most likely be less readable
    - But faster! (usual tradeoff)
- For now, we’re writing the code in the most modular and “clean” fashion
- Let’s first look at the easy main program

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 for 10 integers

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

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

```
push    integers          ; we pass integers (address) 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
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 (address) to count_odds
push    dword 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
```

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 test)

        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]        ; ECX = 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
- 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 onto the stack!
  - And comment that the caller doesn't need to worry about anything
- Let's look at examples

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, for instance, 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
```

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
```

Overwrites the return value that's stored in eax!

Dealing with Return Value

- Saving “all” registers using PUSHA and POPA + return value handling

```
.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 saved return value
                                       ; (as done in our skeleton)
    pop             ebp                ; restore original ebp
    ret                                ; returns
```

Dealing with Return Value

- Saving "all" registers using PUSHA and POPA + return value handling

```
.bss:
    returnvalue     resd        1           ; place in memory for the return value
func:
    push
    mov
    pusha

    . . .

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

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

A much better option is to put the return value in a local variable, which we'll see in the next set of lecture notes

Recursion

- The subprogram calling conventions we have just described enable recursion out of the box!

- Let’s live-code a 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 this, 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
    add             esp, 4              ; 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!
    add             esp, 4               ; 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
```

Conclusion

- You must absolutely make sure you fully understand all code examples in this set of slides
  - Not that this is not true for all code examples in this course ;)
- In the next set of lecture notes we'll talk about local variables in subprograms