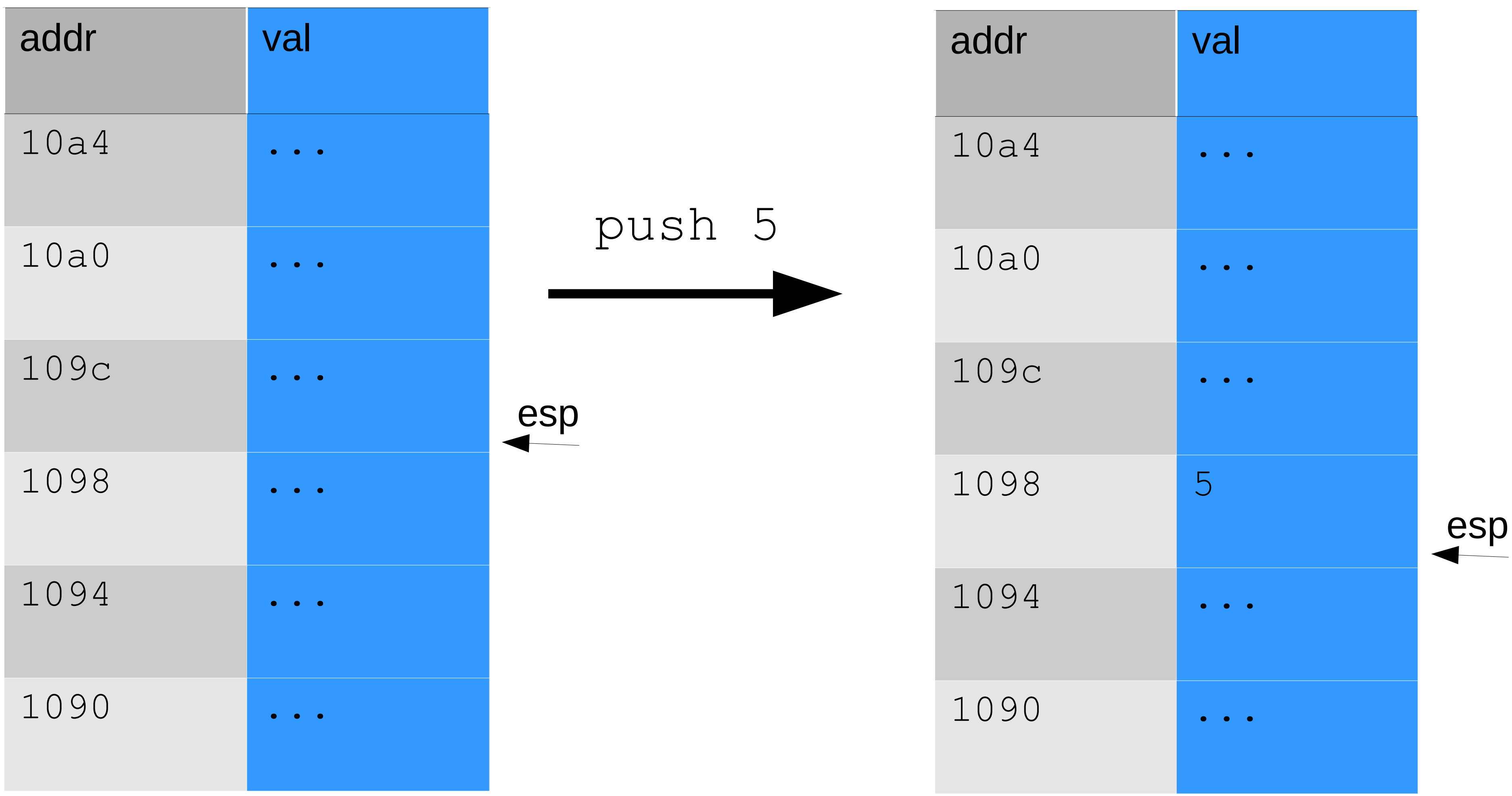
Intel stack

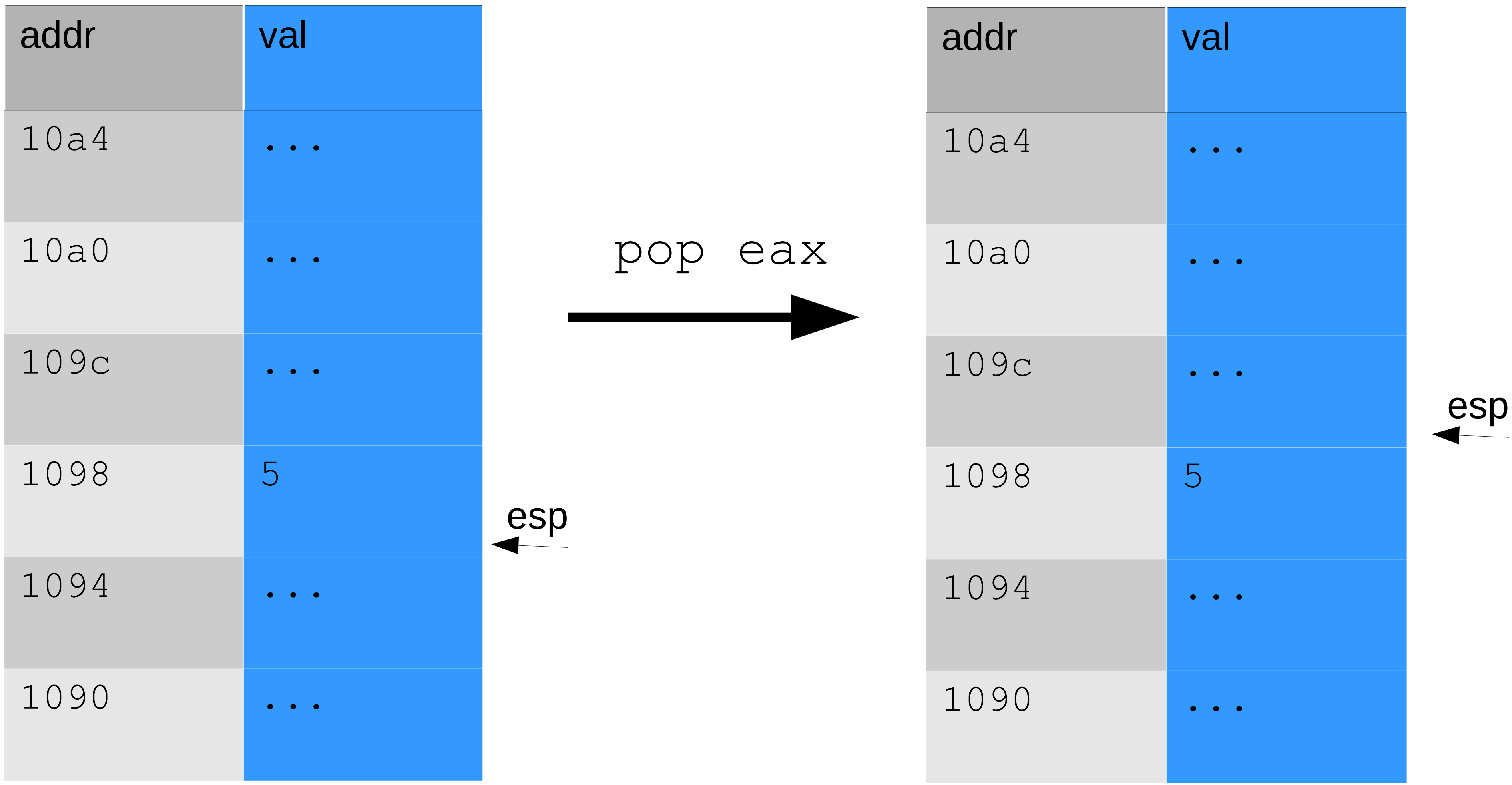
Stack instructions

Instruction	What it does (conceptually)
`JMP addr`	`MOV EIP, addr`
`PUSH reg`	`SUB ESP, 4` `MOV [ESP], reg`
`POP reg`	`MOV reg, [ESP]` `ADD ESP, 4`
`CALL addr`	`PUSH EIP` `JMP addr`
`RET`	`POP EIP`
`RET n`	`POP EIP` `ADD ESP, n`



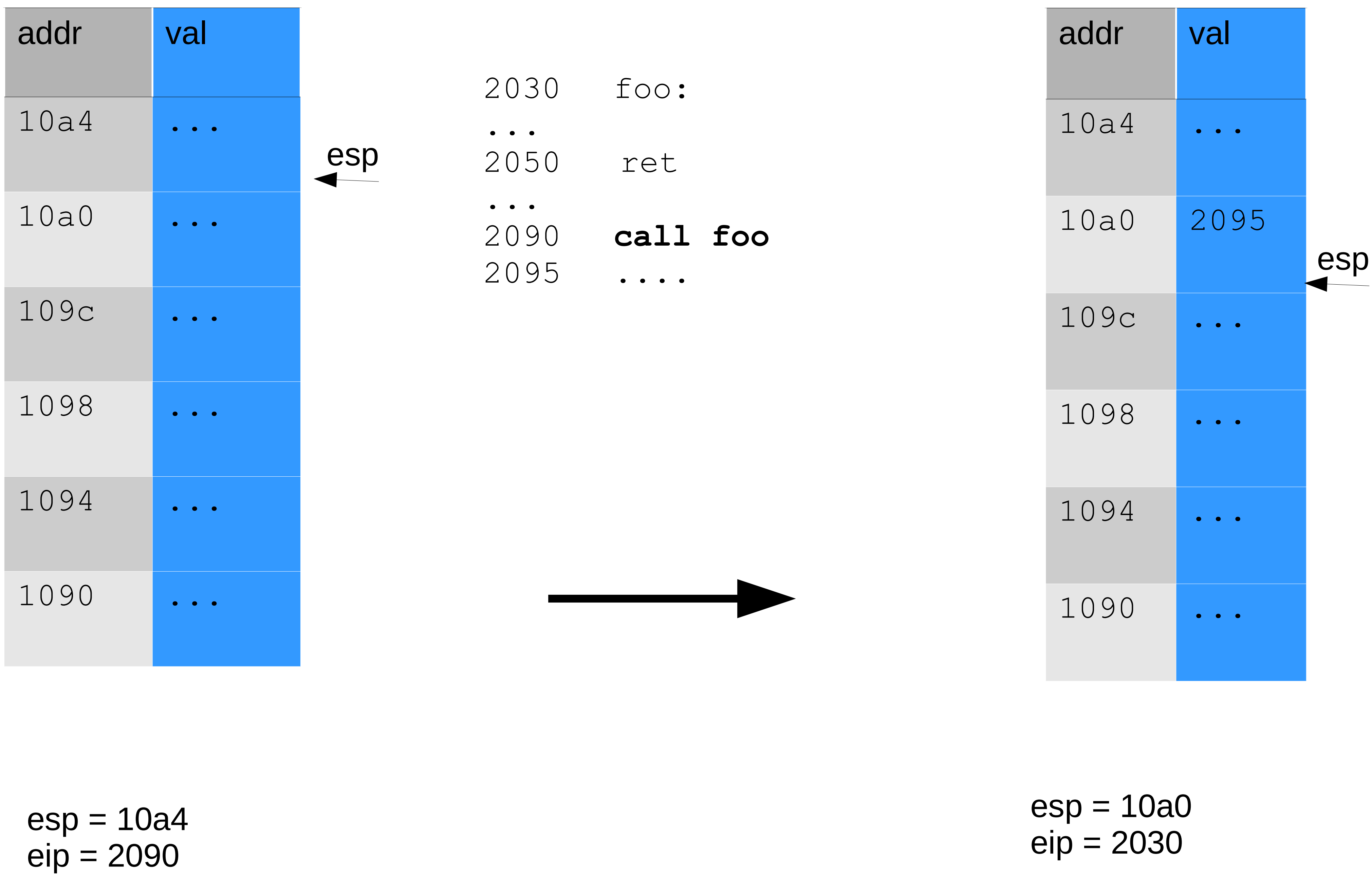
esp = 109c

esp = 1098



esp = 1098

esp = 109c
eax = 5


addr
val
10a4
...
10a0
...
109c
...
1098
...
1094
...
1090
...
esp
2030 foo:
...
2050 ret
...
2090 call foo
2095 ....
addr
val
10a4
...
10a0
2095
109c
...
1098
...
1094
...
1090
...
esp
esp = 10a4
eip = 2090
esp = 10a0
eip = 2030

addr	val
10a4	...
10a0	2095
109c	...
1098	...
1094	...
1090	...

esp

```
2030   foo:
...
2050   ret
...
2090   call foo
2095   ....
```

esp = 10a0
eip = 2050

addr	val
10a4	...
10a0	2095
109c	...
1098	...
1094	...
1090	...

esp

esp = 10a4
eip = 2095

A function

```
section .data
message: db "Hello",10

section .text
global _start


print_message:
    mov eax, 4
    mov ebx, 1
    mov ecx, message
    mov edx, 6
    int 80h
    ret

_start:
    call print_message
    call print_message

    mov eax, 1
    mov ebx, 0
    int 80h
```

A function

```
section .text
global _start


times_two:
    add eax, eax
    ret

_start:
    mov eax, 4
    call times_two
    call times_two

    mov eax, 1
    mov ebx, 0
    int 80h
```

A broken Fibonacci

```
section .text
global _start


fib:
    cmp eax, 2
    jae recursive_case
    mov eax, 1
    jmp done

recursive_case:
    mov ebx, eax

    dec eax         ; call fib(n-1)
    call fib        ; result in eax
    mov ecx, eax

    mov eax, ebx
    sub eax, 2
    call fib        ; call fib(n-2)

    add eax, ecx    ; fib(n-1) + fib(n-2)
done:
    ret
```

```
_start:
    mov eax, 5
    call fib

    mov ebx, eax
    mov eax, 1
    int 80h
```

Typical prologue/epilogue

Typical function prologue:

```
push ebp        ; save the caller's base pointer
mov ebp, esp    ; set our base pointer
sub esp, X      ; allocate X bytes for local vars
```

Typical function epilogue:

```
mov esp, ebp    ; de-allocate local vars
pop ebp         ; restore caller's base pointer
ret N           ; return, and pop N bytes of params
```

Function call with two parameters

addr	val
10a4	0
10a0	99
109c	42
1098	...
1094	...
1090	...

ebp

esp

esp = 109c
ebp = 10a4
eip = 2090

```
2030   foo:
       push ebp
       mov ebp, esp
2038   ...
       mov esp, ebp
       pop ebp
2050   ret 8
...
208a   push 99h
208c   push 42h
2090   call foo
2095   ....
```

Within `foo` (around address 2038), the first parameter is accessible as `[ebp+8]` and the second is `[ebp+12]`

The `ret 8` instruction will clear both parameters off the stack before returning

addr	val
10a4	0
10a0	99
109c	42
1098	2095
1094	10a4
1090	...

esp,ebp

esp = 1094
ebp = 1094
eip = 2038

Function call with two parameters and a local variable

addr	val	
10a4	0	ebp
10a0	99	
109c	42	esp
1098	...	
1094	...	
1090	...	

esp = 109c
ebp = 10a4
eip = 2090

```
2030   foo:
       push ebp
       mov ebp, esp
       sub esp, 4
2040   ...
       mov esp, ebp
       pop ebp
2050   ret 8
...
208a   push 99h
208c   push 42h
2090   call foo
2095   ....
```

addr	val	
10a4	0	
10a0	99	
109c	42	
1098	2095	
1094	10a4	ebp
1090	...	esp

esp = 1090
ebp = 1094
eip = 2040

Parameters available as before, and now the local variable is accessible as `[ebp-4]`. The `sub` instruction allocates space on the stack for local variables; allocate as much space as you need. Local variables are removed from stack by the `mov esp, ebp` instruction.

A recursive call

```
8060   foof:
       push ebp
8061   mov ebp, esp
8063   mov eax, [ebp+8]
8066   cmp eax, 0
8069   je .done
806b   dec eax
806c   push eax
806d   call foof
8072   .done:
       pop ebp
8073   ret 4

8076   _start:
       push 9
8078   call foof
808d   ...
```

addr	val	
10a4	...	
10a0	9	
109c	808d	
1098	0	original EBP
1094	8	
1090	8072	
108c	1098	
1088	7	
1084	8072	
1080	108c	
107c	6	
1078	8072	
1074	1080	
1070	5	
106c	8072	
1068	...	

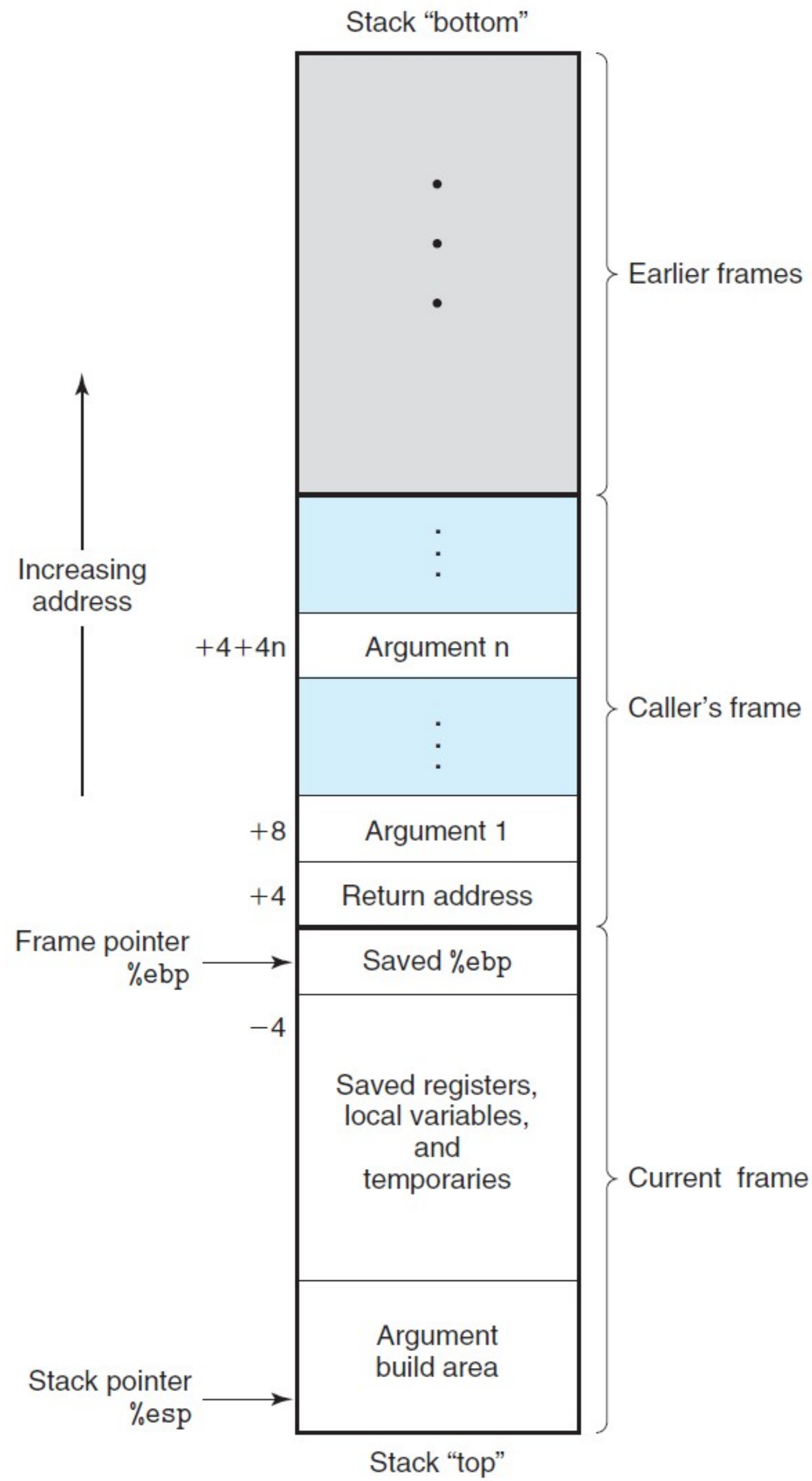
A better Fibonacci

```
section .text
global _start

fib:
    push ebp       ; prologue
    mov ebp, esp

    mov eax, [ebp+8] ; parameter
    cmp eax, 2
    jae recursive_case
    mov eax, 1
    jmp done

recursive_case:
    dec eax
    push eax       ; call fib(n-1)
    call fib       ; result in eax
    push eax

    mov eax, [ebp+8]
    sub eax, 2
    push eax       ; call fib(n-2)
    call fib       ; result in eax

    pop ebx
    add eax, ebx   ; fib(n-1)+fib(n-2)
done:
    pop ebp        ; epilogue
    ret 4
```

```
_start:
    push 10
    call fib

    mov ebx, eax
    mov eax, 1
    int 80h
```

Figure 3.21

Stack frame structure. The stack is used for passing arguments, for storing return information, for saving registers, and for local storage.



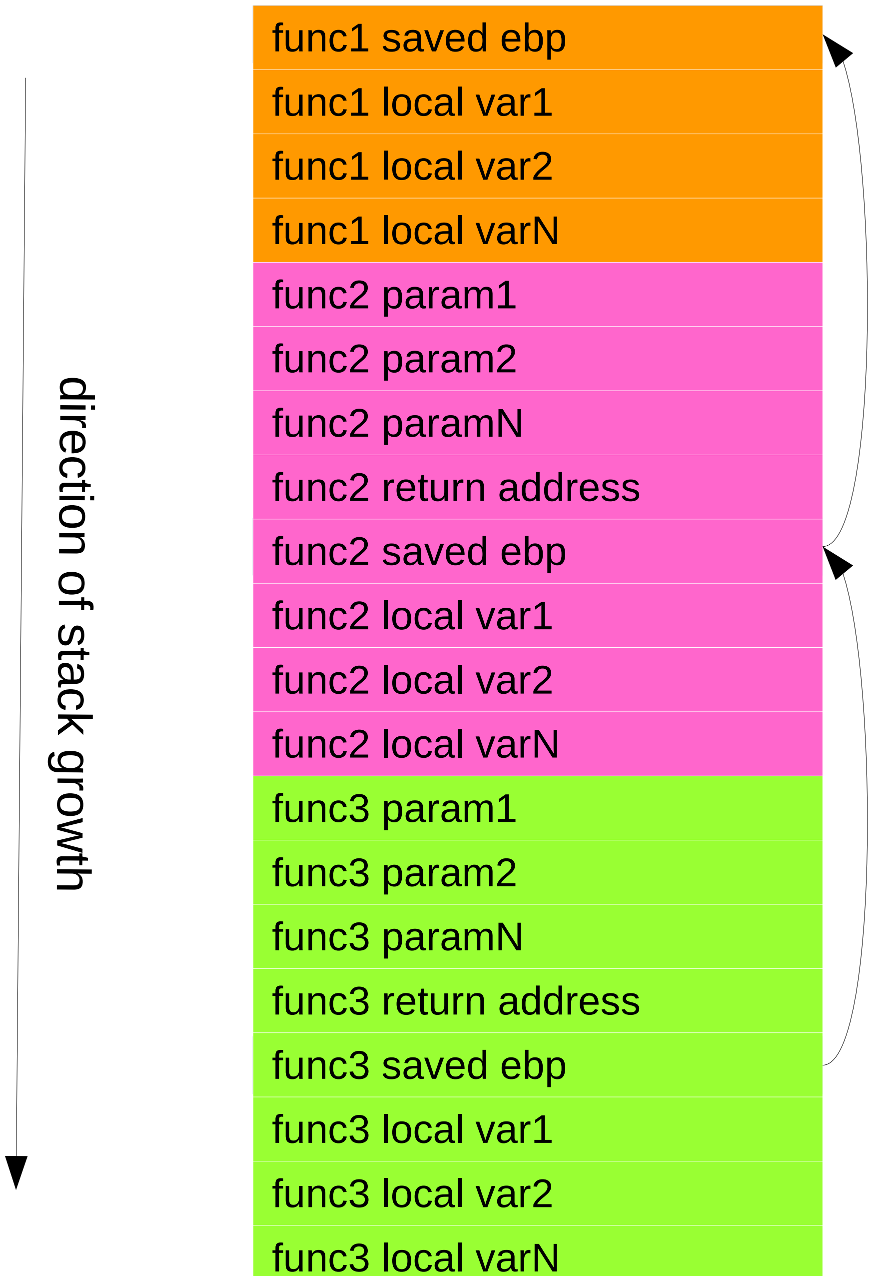
How to access params and local vars

- Given the stack layout on the previous slide, we know this:
  - EBP is the address of the caller's base pointer
  - EBP+4 is the address of this function's return address
  - EBP+8, EBP+12, EBP+18, etc... are the addresses of this function's parameters, in the opposite order that they were pushed (assuming all parameters are dwords)
  - EBP-4, EBP-8, EBP-12, etc.... are the addresses of this function's local variables (assuming that all variables are dwords)

How to access params and local vars

- In practice:
  - **`mov eax, [ebp+8]     ; load first parameter`**
  - **`mov ebx, [ebp-4]     ; load first local variable`**
- Parameters larger than a dword may be passed:
  - by *reference*: the (dword-sized) pointer to the actual value is pushed on the stack; the full value is not placed on the stack
  - by *value*: the full value is placed on the stack, in a sequence of pushes

Chain of stack frames




```
def func1():
    var1=1
    var2=2
    varN=3
    func2(4, 5, 6)

def func2(param1, param2, paramN):
    var1=7
    var2=8
    var3=9
    func3(10, 11, 12)

def func3(param1, param2, paramN):
    var1=13
    var2=14
    var3=15
    print_stack_trace()
```

Timeline of a function call

Caller	Callee
Push parameters on stack `push a`, `push b` (or however many params)	
Push return address and jump to callee `call the_function`	
	Saver caller's ebp `push ebp`
	Set up ebp (for accessing params and local vars) `mov ebp, esp`
	Allocate space for local variables `sub esp, 16` (or however many bytes are needed)
	Main function body! (set return value in reg if necessary)
	Restore stack to initial state (freeing allocated space) `mov esp, ebp`
	Restore caller's ebp `pop ebp`
	Remove args from stack and jump to return address `ret 8` (here we remove 8 bytes, because we had two dword args)
Back in caller, function call complete	

A note on a common point of confusion

`ret`

This instruction pops the return address and jumps to it.

`ret` *n*

This instruction pops the return address, jumps to it, *and removes an additional n bytes from the stack.*

`ret n`, despite its appearance, does *not* specify the return value of a function! That is typically given in the `eax` register.

Removing the additional *n* bytes from the stack is called *stack clean-up*: this removes the parameters that the caller pushed before calling the function. We need to do this, or else the height of the stack would be different after calling.

We've addressed a style called "callee clean-up," wherein the called function (through the `ret n` instruction) removes the parameters before returning. An alternative style, "caller clean-up" is similar, but the called function does a simple `ret` and then the caller must adjust the stack pointer manually (typically with `add esp, n`).

It doesn't matter which style is used, as long as caller and callee agree. For this class, we are only concerned with callee clean-up.

Uppercase a single letter

```
toupper:
push ebp
mov ebp, esp
mov eax, [ebp+8]    ; param is a single ASCII char
cmp al, 97          ; 'a'
jb done
cmp al, 122         ; 'z'
ja done
sub al, 32

done:
pop ebp
ret 4
```

Given a single character, return its uppercase form in eax. If the parameter is not a lowercase letter, return the character unchanged.

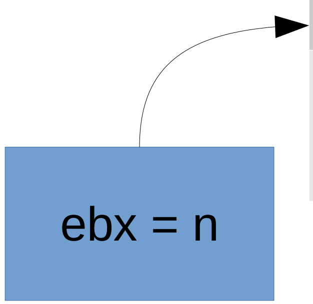
Strings

- What do strings look like in memory?
- We can't store them in a register: strings can be of arbitrary size, but registers are small
- Instead, strings are arrays of bytes, where each byte represents a character
- Then, we store the address of the first byte of the string in a register: this is a `char*`
- How do we know how long the string is?
  - Option 1: The byte after the last character in the string is 0. When we reach that byte, we know we've found the end of the string
  - Option 2: The length of the string is stored separately as a dword

Strings

Option 1: register stores address of first byte. String is terminated by NUL.

address	n	n+1	n+2	n+3	n+4	n+5
numeric	72	101	108	108	111	0
char	H	e	l	l	o	NUL



Strings

Option 2: no terminated NULL byte. Instead, length is carried separately.

address	n	n+1	n+2	n+3	n+4
numeric	72	101	108	108	111
char	H	e	l	l	o

ebx = n

ecx = 5

Uppercase a whole string

```
toupper_string:
push ebp
mov ebp, esp
mov ebx, [ebp+8]    ; param is address of a null-terminated string

loop:
xor ecx,ecx
mov cl, [ebx]   ; get one char from string
cmp cl, 0       ; if it's null, we're done
je done
push ecx
call toupper    ; returns output char in al
mov [ebx], al   ; store it back into string
inc ebx         ; go to next char
jmp loop

done:
pop ebp
ret 4
```

This is okay because we know that toupper doesn't modify ebx

Given a pointer to a null-terminated string, adjust it to be uppercase. We call the toupper function defined earlier. We do not return a useful value: instead the parameter is mutated.

$$\gcd(x, y) = \begin{cases} x & \text{if } y = 0 \\ \gcd(y, \text{remainder}(x, y)) & \text{if } y > 0 \end{cases}$$

```
int gcd(int x, int y)
{
  if (y == 0)
     return x;
  else
     return gcd(y, x % y);
}
```

Call it like this:

```
push 15
push 25
call gcd
```

```
gcd:
    push ebp
    mov ebp, esp
    ; assume x is at ebp+8, y is at ebp+12

   ;; YOUR CODE HERE
   ;; put return value in eax

    mov esp, ebp
    pop ebp
    ret 8
```

```
gcd:
    push ebp
    mov ebp, esp
    ; assume x is at ebp+8, y is at ebp+12

    cmp [ebp+12], 0
    je returnx

    xor edx, edx
    mov eax, [ebp+8]
    div [ebp+12]            ; divides edx:eax by [ebp+12]
                            ; leaves remainder in edx

    push edx                ; push args right-to-left
    push [ebp+12]
    call gcd
    jmp justreturn


returnx:
    mov eax, [ebp+8]

justreturn:
    mov esp, ebp
    pop ebp
    ret 8
```

Debugging with gdb: an example

- We want to sum an array of dwords

```
section .data
array_len:
    dd 10
array:
    dd 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
```

Debugging with gdb: an example

- Almost correct solution:

```
section .data
array_len:
    dd 10
array:
    dd 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
sum:
    dd 0
section .text
_start:
    mov ecx, array_len
    mov eax, array
loop:
    mov ebx, [eax]
    add [sum], ebx
    add eax, 4
    dec ecx     ; reduce length by 1 to keep track of # of iterations
    cmp ecx, 1  ; stop if we're at the end
    jne loop
```

Debugging with gdb: an example

- Problem:

Debugging with gdb: an example

- Let's run the program in gdb

```
─── Output/messages ─────────────────────────────────────────────

Program received signal SIGSEGV, Segmentation fault.
0x08049010 in loop ()
─── Assembly ────────────────────────────────────────────────────
0x08049010 loop+0  mov    ebx,DWORD PTR [eax]
0x08049012 loop+2  add    DWORD PTR ds:0x804a024,ebx
0x08049018 loop+8  add    eax,0x4
0x0804901b loop+11 dec    ecx
─── Expressions ─────────────────────────────────────────────────
─── History ─────────────────────────────────────────────────────
─── Memory ──────────────────────────────────────────────────────
─── Registers ───────────────────────────────────────────────────
   eax 0x0804b000      ecx 0x08049c01      edx 0x00000000      ebx 0x00000000
   esp 0xffffd290      ebp 0x00000000      esi 0x00000000      edi 0x00000000
   eip 0x08049010   eflags [ PF IF RF ]     cs 0x00000023       ss 0x0000002b
    ds 0x0000002b       es 0x0000002b       fs 0x00000000       gs 0x00000000
─── Source ──────────────────────────────────────────────────────
```

Debugging with gdb: an example

- Why is this instruction problematic?

```
── Output/messages ──────────────────────────────────────────

Program received signal SIGSEGV, Segmentation fault.
0x08049010 in loop ()
── Assembly ─────────────────────────────────────────────────
0x08049010 loop+0  mov    ebx,DWORD PTR [eax]
0x08049012 loop+2  add    DWORD PTR ds:0x804a024,ebx
0x08049018 loop+8  add    eax,0x4
0x0804901b loop+11 dec    ecx
── Expressions ──────────────────────────────────────────────
── History ──────────────────────────────────────────────────
── Memory ───────────────────────────────────────────────────
── Registers ────────────────────────────────────────────────
   eax 0x0804b000                          edx 0x00000000       ebx 0x00000000
   esp 0xffffd290                           si 0x00000000       edi 0x00000000
   eip 0x08049010                           cs 0x00000023        ss 0x0000002b
    ds 0x0000002b                           fs 0x00000000        gs 0x00000000
── Source ───────────────────────────────────────────────────
```

eax should store the address of an array element. Evidently, this address is invalid. But why?

Debugging with gdb: an example

- Why is this instruction problematic?

```
─── Output/messages ─────────────────────────────────────────────

Program received signal SIGSEGV, Segmentation fault.
0x08049010 in loop ()
─── Assembly ────────────────────────────────────────────────────
0x08049010 loop+0  mov    ebx,DWORD PTR [eax]
0x08049012 loop+2  add    DWORD PTR ds:0x804a024,ebx
0x08049018 loop+8  add    eax,0x4
0x0804901b loop+11 dec    ecx
─── Expressions ─────────────────────────────────────────────────
─── History ─────────────────────────────────────────────────────
─── Memory ──────────────────────────────────────────────────────
─── Registers ───────────────────────────────────────────────────
   eax 0x0804b000      ecx 0x08049c01      edx 0x00000000      ebx 0x00000000
   esp 0xffffd290      ebp 0x0000000                           edi 0x00000000
   eip 0x08049010   eflags [ PF IF RF                           ss 0x0000002b
    ds 0x0000002b       es 0x0000002b                           gs 0x00000000
─── Source ──────────────────────────────────────────────────────
```

ecx should store the length of the array. This value is clearly wrong. What happened?

Debugging with gdb: an example

- Almost correct solution:

```
section .data
array_len:
    dd 10
array:
    dd 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
sum:
    dd 0
section .text
_start:
    mov ecx, array_len
    mov eax, array
loop:
    mov ebx, [eax]
    add [sum], ebx
    add eax, 4
    dec ecx      ; reduce length by 1 to keep track of # of iterations
    cmp ecx, 1   ; stop if we're at the end
    jne loop
```

Debugging with gdb: an example

- Almost correct solution:

```
section .data
array_len:
    dd 10
array:
    dd 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
sum:
    dd 0
section .text
_start:
    mov ecx, [array_len]
    mov eax, array
loop:
    mov ebx, [eax]
    add [sum], ebx
    add eax, 4
    dec ecx      ; reduce length by 1 to keep track of # of iterations
    cmp ecx, 1   ; stop if we're at the end
    jne loop
```

We're storing the *address* of the array length in ecx, not the array length itself. Therefore ecx's value is too high, and the loop executes too many iterations, causing eax to be incremented past the range of valid memory.