Advanced Systems Programming

Assembly

Maksym Planeta, Björn Döbel, Tobias Stumpf

24.09.2021

What the hell - Why should I learn assembly?

Understanding debugger output:

```
400d4e: 55                    push   %rbp
400d4f: 48 89 e5              mov    %rsp,%rbp
400d52: bf 84 79 48 00        mov    $0x487984,%edi
400d57: e8 54 6b 00 00        callq  4078b0 <_IO_puts>
400d5c: 5d                    pop    %rbp
400d5d: c3                    retq
```

get full controll over your hardware (using specific instructions)

system programming (e.g. kernel entry/exit)

We need to go deeper: Fibonacci

```
int fib(int n)
{
  int fcur = 0, fnext = 1, tmp;
  while(--n>0) {
    tmp = fcur + fnext;
    fcur = fnext;
    fnext = tmp;
  }
  return fnext;
}

int main(int argc, char ** argv)
{
  printf("Fib: %d\n", fib(atoi(argv[1])));
}
```

Fibonacci

fib.c

```
int fib(int n)
{
  int fcur = 0, fnext = 1, tmp;
  while(--n>0) {
    tmp = fcur + fnext;
    fcur = fnext;
    fnext = tmp;
  }
  return fnext;
}
```

Fibonacci

fib.c

```
int fib(int n)
{
  int fcur = 0, fnext = 1, tmp;
  while(--n>0) {
    tmp = fcur + fnext;
    fcur = fnext;
    fnext = tmp;
  }
  return fnext;
}
```

```
gcc -Wall -O2 -march=x86-64 -c -o fib.o fib.c
```

Sections of object file

```
$ objdump -h fib.o

fib.o:     file format elf64-x86-64

Sections:
Idx Name          Size      ...  File off  Algn
  0 .text         00000023  ...  00000040  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .data         00000000  ...  00000063  2**0
                  CONTENTS, ALLOC, LOAD, DATA

...
```

Sections of object file

```
$ objdump -h fib.o

fib.o:     file format elf64-x86-64

Sections:
Idx Name          Size      ...  File off  Algn
  0 .text         00000023  ...  00000040  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .data         00000000  ...  00000063  2**0
                  CONTENTS, ALLOC, LOAD, DATA

...
```

Looking into text section

```
 $ dd if=fib.o of=fib.o.hex bs=1 count=$((0x23)) skip=$((0x40))
35+0 records in
35+0 records out
35 bytes copied, 0.000799485 s, 43.8 kB/s
 $ xxd fib.o.hex
00000000: 83ef 0185 ff7e 16ba 0100 0000 31c9 6690  .....~......1.f.
00000010: 8d04 1189 d189 c283 ef01 75f4 c3b8 0100  ..........u.....
00000020: 0000 c3
```

What sees a processor

```
83ef0185ff7e16ba0100000031c966908d041189d189c283ef01
75f4c3b801000000c3
```

What sees a human

```
fib:
        sub     $0x1,%edi
        test    %edi,%edi
        jle     1d <fib+0x1d>
        mov     $0x1,%edx
        xor     %ecx,%ecx
        xchg    %ax,%ax
        lea     (%rcx,%rdx,1),%eax
        mov     %edx,%ecx
        mov     %eax,%edx
        sub     $0x1,%edi
        jne     10 <fib+0x10>
        retq
        mov     $0x1,%eax
        retq
```

What sees a processor

```
83ef0185ff7e16ba0100000031c966908d041189d189c283ef01
75f4c3b801000000c3
```

What sees a processor

```
83ef0185ff7e16ba0100000031c966908d041189d189c283ef01
75f4c3b801000000c3
```

A processor opens “Intel Software Developer’s Manual. Volume 2C. Appendix A. Table A-2”.

```
$ wget http://svn.inf.tu-dresden.de/repos/advsysprog/asm/opcodes.pdf
```

What sees a processor

```
83ef0185ff7e16ba0100000031c966908d041189d189c283ef01
75f4c3b801000000c3
```

A processor opens “Intel Software Developer’s Manual. Volume 2C. Appendix A. Table A-2”.

```
$ wget http://svn.inf.tu-dresden.de/repos/advsysprog/asm/opcodes.pdf
```

$$Table[0x8, 0x3] = \text{Immediate Grp 1 : Ev, Ib}$$

What 0x83 stands for?

Ev, Ib

- E A ModR/M byte follows the opcode. The operand is either a GPR or an address.
- v Word, doubleword or quadword
- I Immediate data
- b Byte

What 0x83 stands for?

Ev, Ib

- E A ModR/M byte follows the opcode. The operand is either a GPR or an address.
- v Word, doubleword or quadword
- I Immediate data
- b Byte

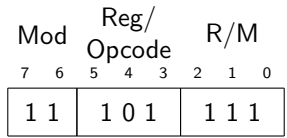
Need to look into next byte

ModR/M

```
83ef0185ff7e16ba0100000031c966908d041189d189c283ef01
75f4c3b801000000c3
```

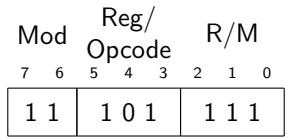
ModR/M

```
83ef0185ff7e16ba0100000031c966908d041189d189c283ef01
75f4c3b801000000c3
```



ModR/M

```
83ef0185ff7e16ba0100000031c966908d041189d189c283ef01
75f4c3b801000000c3
```



$\text{Mod}(11) + \text{R/M}(111) \rightarrow \textit{edi}$
$\text{Opcode}(101) \rightarrow \textit{sub}$

Immediate data

```
sub imm8, %edi
```

Immediate data

```
sub imm8, %edi
```

```
83ef0185ff7e16ba0100000031c966908d041189d189c283ef01
75f4c3b801000000c3
```

Look into next byte

Immediate data

```
sub imm8, %edi
```

```
83ef0185ff7e16ba0100000031c966908d041189d189c283ef01
75f4c3b801000000c3
```

Look into next byte

Immediate data

```
sub imm8, %edi
```

```
83ef0185ff7e16ba0100000031c966908d041189d189c283ef01
75f4c3b801000000c3
```

Look into next byte

```
sub $0x1, %edi
```

Next instruction

```
83ef0185ff7e16ba0100000031c966908d041189d189c283ef01
75f4c3b801000000c3
```

Next instruction

```
85ff7e16ba0100000031c966908d041189d189c283ef0175f4 c3b801000000c3
```

Next instruction

85ff7e16ba0100000031c966908d041189d189c283ef0175f4 c3b801000000c3

$$\text{Table}[8, 5] = \textit{Test} : \text{Ev, Gv}$$

Next instruction

85ff7e16ba0100000031c966908d041189d189c283ef0175f4 c3b801000000c3

$$\text{Table}[8, 5] = \textit{Test} : \text{Ev, Gv}$$

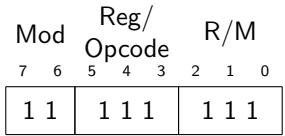
G ModR/M byte selects a general register

Next introduction (ModR/M)

```
85ff7e16ba0100000031c966908d041189d189c283ef0175f4 c3b801000000c3
```

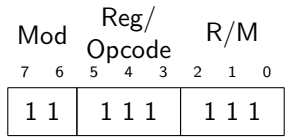
Next introduction (ModR/M)

```
85ff7e16ba0100000031c966908d041189d189c283ef0175f4 c3b801000000c3
```



Next introduction (ModR/M)

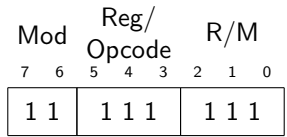
```
85ff7e16ba0100000031c966908d041189d189c283ef0175f4 c3b801000000c3
```



$$\text{Mod}(11) + \text{R/M}(111) \rightarrow \textit{edi}$$
$$\text{Reg}(111) \rightarrow \textit{edi}$$

Next introduction (ModR/M)

```
85ff7e16ba0100000031c966908d041189d189c283ef0175f4 c3b801000000c3
```



$$\text{Mod}(11) + \text{R/M}(111) \rightarrow \textit{edi}$$
$$\text{Reg}(111) \rightarrow \textit{edi}$$

```
test %edi, %edi
```

Continue decoding

```
85ff7e16ba0100000031c966908d041189d189c283ef0175f4 c3b801000000c3
```

Continue decoding

```
7e16ba0100000031c966908d041189d189c283ef0175f4c3b8 01000000c3
```

Continue decoding

`7e16ba0100000031c966908d041189d189c283ef0175f4c3b8 01000000c3`

$$\text{Table}[7, \text{e}] = \textit{jle}$$

Short jump is followed by single byte immediate offset.
Near jump has prefix `0x0f`, e. g. `0x0f7e` – near jle.

Continue decoding

`7e16ba0100000031c966908d041189d189c283ef0175f4c3b8 01000000c3`

$$\text{Table}[7, \text{e}] = \textit{jle}$$

Short jump is followed by single byte immediate offset.
Near jump has prefix `0x0f`, e. g. `0x0f7e` – near jle.

Continue decoding

7e16ba0100000031c966908d041189d189c283ef0175f4c3b8 01000000c3

$$\text{Table}[7, \text{e}] = \textit{jle}$$

Short jump is followed by single byte immediate offset.
Near jump has prefix `0x0f`, e. g. `0x0f7e` – near jle.

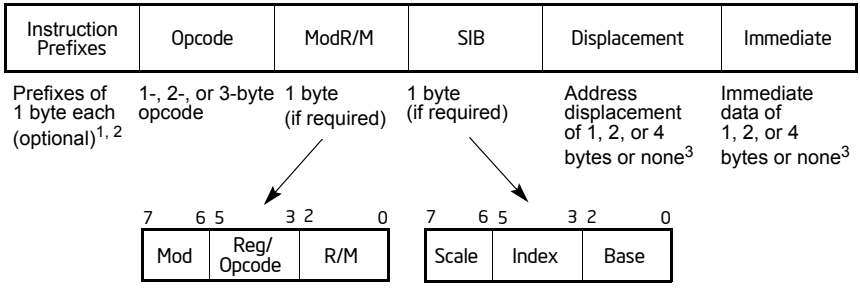
```
jle  $0x16
```

Jump is relative to the address of next instruction

Check

```
0000000000000000 <fib>:
   0:   83 ef 01              sub    $0x1,%edi
   3:   85 ff                 test   %edi,%edi
   5:   7e 16                 jle    1d <fib+0x1d>
...
  1c:   c3                    retq
  1d:   b8 01 00 00 00        mov    $0x1,%eax
  22:   c3                    retq
```

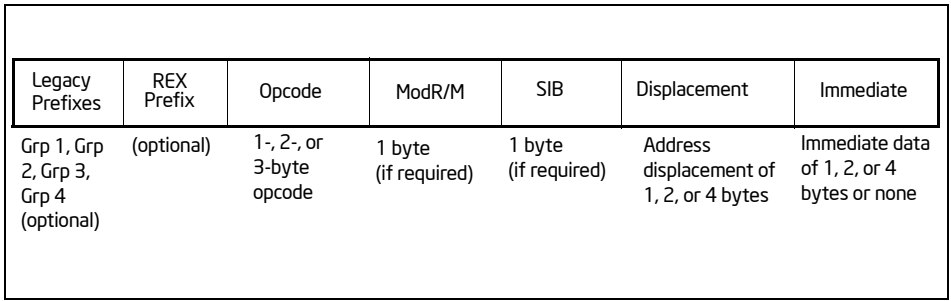
Generic instruction type



1. The REX prefix is optional, but if used must be immediately before the opcode; see Section 2.2.1, “REX Prefixes” for additional information.
2. For VEX encoding information, see Section 2.3, “Intel® Advanced Vector Extensions (Intel® AVX)”.
3. Some rare instructions can take an 8B immediate or 8B displacement.

Figure 2-1. Intel 64 and IA-32 Architectures Instruction Format

Generic instruction type



Legacy Prefixes	REX Prefix	Opcode	ModR/M	SIB	Displacement	Immediate
Grp 1, Grp 2, Grp 3, Grp 4 (optional)	(optional)	1-, 2-, or 3-byte opcode	1 byte (if required)	1 byte (if required)	Address displacement of 1, 2, or 4 bytes	Immediate data of 1, 2, or 4 bytes or none

Figure 2-3. Prefix Ordering in 64-bit Mode

Try it yourself

Pick one of those:

- `8d 04 11`
- `75 f4`
- `b8 01 ae 04 bc`

Complete disassembly

```
0000000000000000 <fib>:
 0: 83 ef 01                sub    $0x1,%edi
 3: 85 ff                   test   %edi,%edi
 5: 7e 16                   jle    1d <fib+0x1d>
 7: ba 01 00 00 00          mov    $0x1,%edx
 c: 31 c9                   xor    %ecx,%ecx
 e: 66 90                   xchg   %ax,%ax
10: 8d 04 11                lea    (%rcx,%rdx,1),%eax
13: 89 d1                   mov    %edx,%ecx
15: 89 c2                   mov    %eax,%edx
17: 83 ef 01                sub    $0x1,%edi
1a: 75 f4                   jne    10 <fib+0x10>
1c: c3                      retq
1d: b8 01 00 00 00          mov    $0x1,%eax
22: c3                      retq
```

What do you think?

- What is the most common number of operands for x86 assembly?

What do you think?

- What is the most common number of operands for x86 assembly?
- Why there is no three operand assembly instruction?

What do you think?

- What is the most common number of operands for x86 assembly?
- Why there is no three operand assembly instruction?
- Fixed length instructions. What are advantages and disadvantages?

What do you think?

- What is the most common number of operands for x86 assembly?
- Why there is no three operand assembly instruction?
- Fixed length instructions. What are advantages and disadvantages?
- What is one operand instruction?

What do you think?

- What is the most common number of operands for x86 assembly?
- Why there is no three operand assembly instruction?
- Fixed length instructions. What are advantages and disadvantages?
- What is one operand instruction?
- What is zero operand instruction?

Move Instructions

mov

move data between registers or to/from memory

```
movl   $1,%eax
movl   $0xff,%ebx
movl   (%ebx),%eax
movl   3(%ebx),%eax
```

Arithmetic Instructions

add/sub

addition / substraction

```
add   $1,%eax
add   %eax,%ebx
sub   $1,%eax
sub   %eax,%ebx
```

Logical Instructions

and/or/xor/test

logical operations

```
and   %eax,%ebx
or    %eax,%ebx
xor   %eax,%ebx
test  %eax,%ebx
```

Stack Instructions

push/pop

push or pop register content to or from the stack

```
push %eax
pop %eax
pusha
popa
```

Function-related Instructions

call

call a function

```
call 0xC0FFEE
call 0xBADA55
ret
```

x86_32 (Linux)

Arguments are passed on the stack.
Integer values and memory addresses are returned in the EAX register.
Registers EAX, ECX, and EDX are caller-saved, and the rest are callee-saved.

https://en.wikipedia.org/wiki/X86_calling_conventions

	Par. Reg	Par. Stack	Cleanup
Microsoft	RCX, RDX, R8, R9	RTL(C)	Caller
System V	RDI, RSI, RDX, RCX R8, R9	RTL(C)	Caller

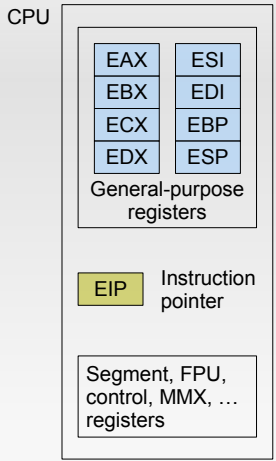
	Par. Reg	Par. Stack	Cleanup
Microsoft	RCX, RDX, R8, R9	RTL(C)	Caller
System V	RDI, RSI, RDX, RCX R8, R9	RTL(C)	Caller

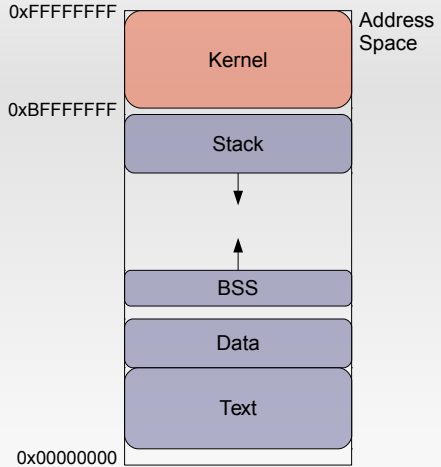
	Return	Callee Saved
Microsoft	RAX	RBX, RBP, RDI, RSI, R12 - R15
System V	RAX	RBX, RBP, R12-R15

Buffers on the stack

Stolen from DOS...

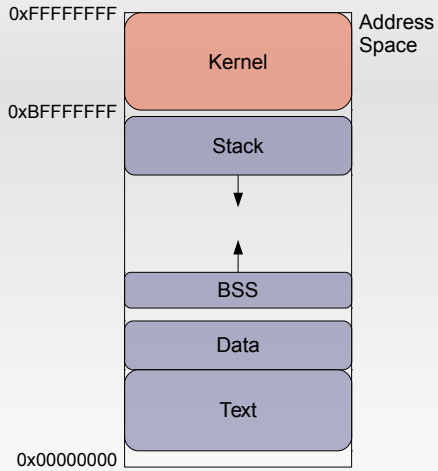
The Battlefield: x86/32





The Stack

- Stack frame per function
  - Set up by compiler-generated code
- Used to store
  - Function parameters
  - If not in registers – GCC: `__attribute__ ((regparm((<num>))))`
  - Local variables
  - Control information
    - Function return address



Calling a function

```
int sum(int a, int b)
{
   return a+b;
}
```

```
sum:
  pushl %ebp
  movl %esp, %ebp
  movl 12(%ebp), %eax
  addl 8(%ebp), %eax
  popl %ebp
  ret
```

```
int main()
{
   return sum(1,3);
}
```

```
main:
  pushl %ebp
  movl %esp, %ebp
  subl $8, %esp
  movl $3, 4(%esp)
  movl $1, (%esp)
  call sum
  ret
```

Assembly recap'd

%<reg> refers to register content

Offset notation: X(%reg) == memory Location pointed to by reg + X

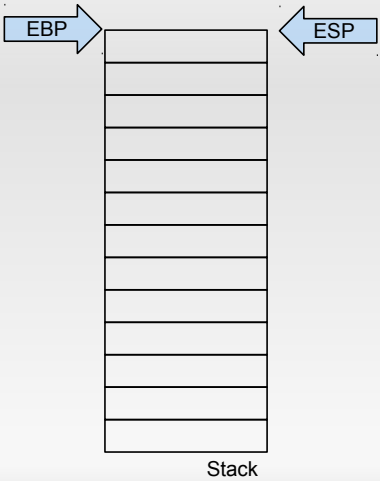
```
sum:
  pushl %ebp
  movl %esp, %ebp
  movl 12(%ebp), %eax
  addl 8(%ebp), %eax
  popl %ebp
  ret
```

Constants prefixed with $ sign

(%<reg>) refers to memory location pointed to by <reg>

```
main:
  pushl %ebp
  movl %esp, %ebp
  subl $8, %esp
  movl $3, 4(%esp)
  movl $1, (%esp)
  call sum
  ret
```

So what happens on a call?

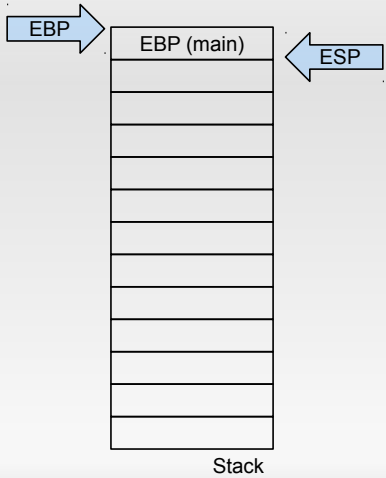



```
sum:
  pushl %ebp
  movl %esp, %ebp
  movl 12(%ebp), %eax
  addl 8(%ebp), %eax
  leave
  ret

main:
  pushl %ebp
  movl %esp, %ebp
  subl $8, %esp
  movl $3, 4(%esp)
  movl $1, (%esp)
  call sum
  ret
```

EIP

So what happens on a call?

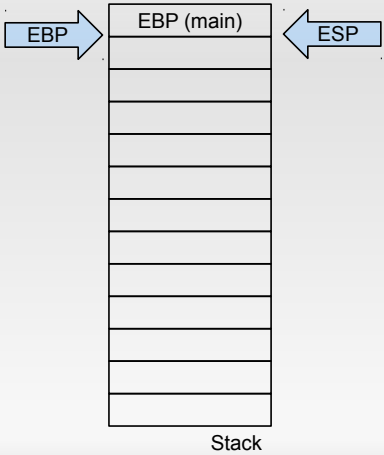



```
sum:
  pushl %ebp
  movl %esp, %ebp
  movl 12(%ebp), %eax
  addl 8(%ebp), %eax
  leave
  ret

main:
  pushl %ebp
  movl %esp, %ebp
  subl $8, %esp
  movl $3, 4(%esp)
  movl $1, (%esp)
  call sum
  ret
```

EIP

So what happens on a call?

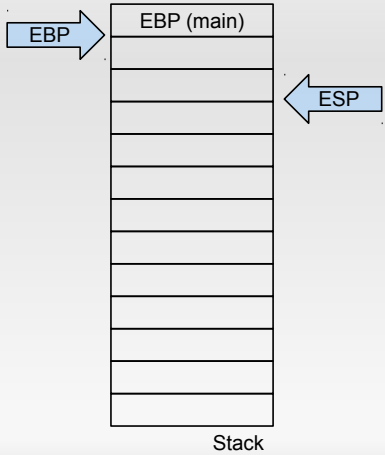



```
sum:
  pushl %ebp
  movl %esp, %ebp
  movl 12(%ebp), %eax
  addl 8(%ebp), %eax
  leave
  ret

main:
  pushl %ebp
  movl %esp, %ebp
  subl $8, %esp
  movl $3, 4(%esp)
  movl $1, (%esp)
  call sum
  ret
```

EIP

So what happens on a call?

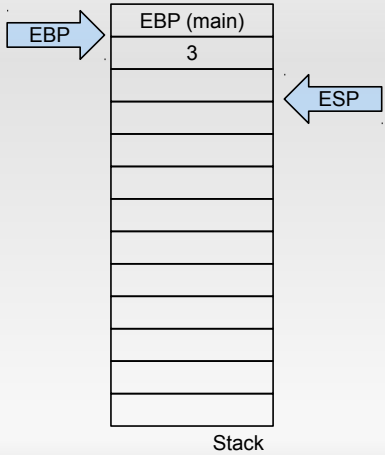



```
sum:
  pushl %ebp
  movl %esp, %ebp
  movl 12(%ebp), %eax
  addl 8(%ebp), %eax
  leave
  ret

main:
  pushl %ebp
  movl %esp, %ebp
  subl $8, %esp
  movl $3, 4(%esp)
  movl $1, (%esp)
  call sum
  ret
```

EIP

So what happens on a call?

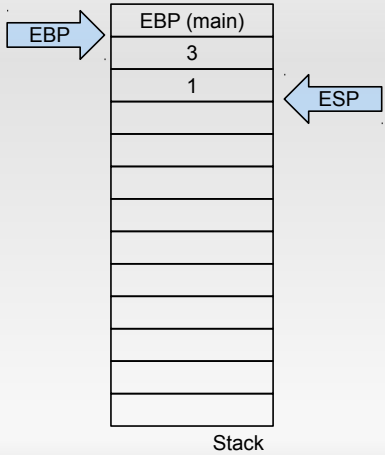



```
sum:
  pushl %ebp
  movl %esp, %ebp
  movl 12(%ebp), %eax
  addl 8(%ebp), %eax
  leave
  ret

main:
  pushl %ebp
  movl %esp, %ebp
  subl $8, %esp
  movl $3, 4(%esp)
  movl $1, (%esp)
  call sum
  ret
```

EIP

So what happens on a call?

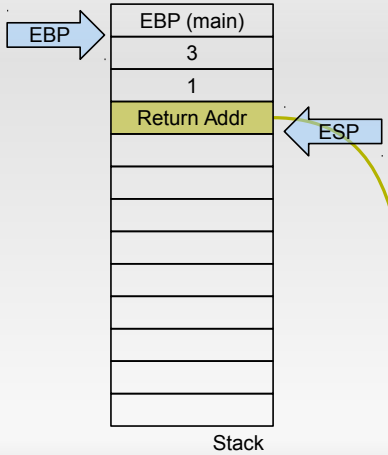



```
sum:
  pushl %ebp
  movl %esp, %ebp
  movl 12(%ebp), %eax
  addl 8(%ebp), %eax
  leave
  ret

main:
  pushl %ebp
  movl %esp, %ebp
  subl $8, %esp
  movl $3, 4(%esp)
  movl $1, (%esp)
  call sum
  ret
```

EIP

So what happens on a call?

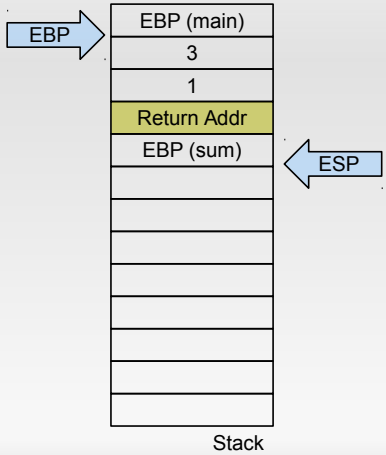



```
sum:
  pushl %ebp
  movl %esp, %ebp
  movl 12(%ebp), %eax
  addl 8(%ebp), %eax
  leave
  ret
```

EIP

```
main:
  pushl %ebp
  movl %esp, %ebp
  subl $8, %esp
  movl $3, 4(%esp)
  movl $1, (%esp)
  call sum
  ret
```

So what happens on a call?

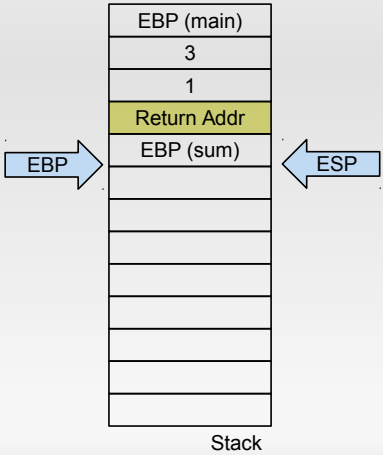



```
sum:
  pushl %ebp
  movl %esp, %ebp
  movl 12(%ebp), %eax
  addl 8(%ebp), %eax
  leave
  ret
```

EIP

```
main:
  pushl %ebp
  movl %esp, %ebp
  subl $8, %esp
  movl $3, 4(%esp)
  movl $1, (%esp)
  call sum
  ret
```

So what happens on a call?

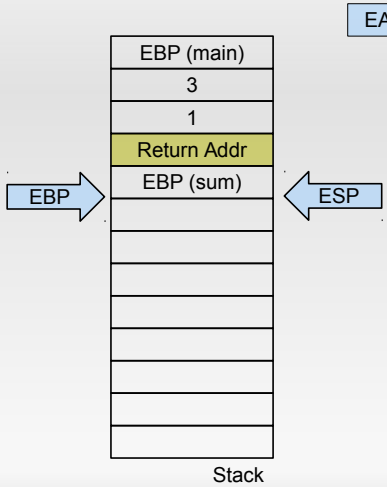



```
sum:
  pushl %ebp
  movl %esp, %ebp
  movl 12(%ebp), %eax
  addl 8(%ebp), %eax
  leave
  ret
```

EIP

```
main:
  pushl %ebp
  movl %esp, %ebp
  subl $8, %esp
  movl $3, 4(%esp)
  movl $1, (%esp)
  call sum
  ret
```

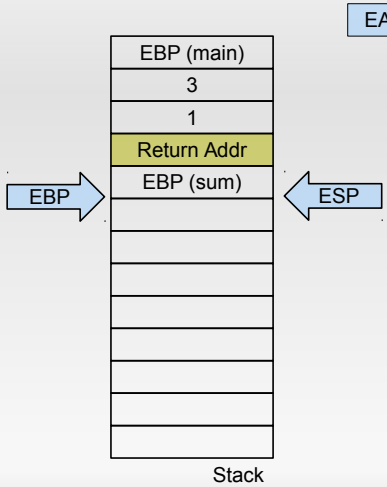
So what happens on a call?




```
sum:
  pushl %ebp
  movl %esp, %ebp
  movl 12(%ebp), %eax
  addl 8(%ebp), %eax
  leave
  ret

main:
  pushl %ebp
  movl %esp, %ebp
  subl $8, %esp
  movl $3, 4(%esp)
  movl $1, (%esp)
  call sum
  ret
```

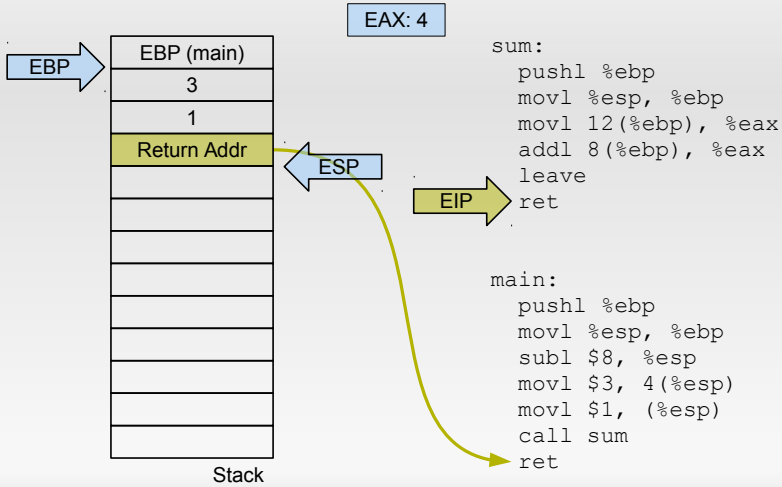
So what happens on a call?



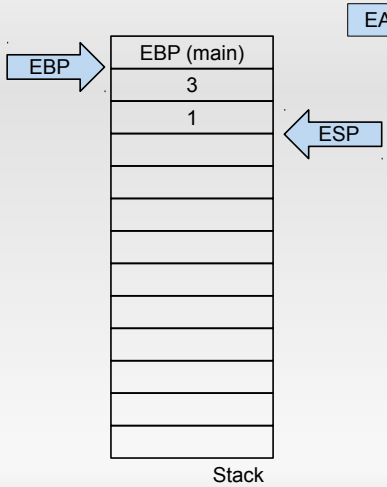

```
sum:
  pushl %ebp
  movl %esp, %ebp
  movl 12(%ebp), %eax
  addl 8(%ebp), %eax
  leave
  ret

main:
  pushl %ebp
  movl %esp, %ebp
  subl $8, %esp
  movl $3, 4(%esp)
  movl $1, (%esp)
  call sum
  ret
```

So what happens on a call?



So what happens on a call?




```
sum:
  pushl %ebp
  movl %esp, %ebp
  movl 12(%ebp), %eax
  addl 8(%ebp), %eax
  leave
  ret


main:
  pushl %ebp
  movl %esp, %ebp
  subl $8, %esp
  movl $3, 4(%esp)
  movl $1, (%esp)
  call sum
  ret
```

Exploitz

Now let's add a buffer

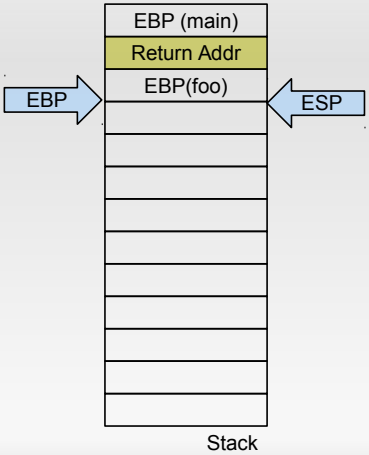
```
int foo()
{
   char buf[20];
   return 0;
}

int main()
{
   return foo();
}
```

```
foo:
   pushl %ebp
   movl %esp, %ebp
   subl $32, %esp
   movl $0, %eax
   leave
   ret

main:
   pushl %ebp
   movl %esp, %ebp
   call foo
   popl %ebp
   ret
```

Now let's add a buffer

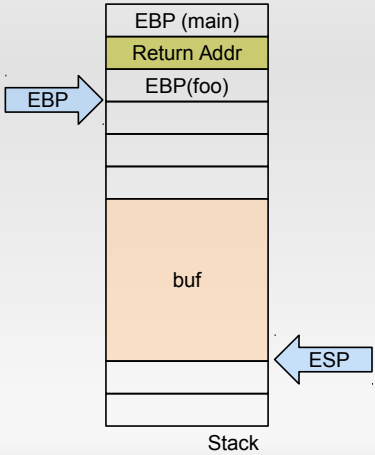



```
foo:
   pushl %ebp
   movl %esp, %ebp
   subl $32, %esp
   movl $0, %eax
   leave
   ret
```

EIP

```
main:
   pushl %ebp
   movl %esp, %ebp
   call foo
   popl %ebp
   ret
```

Now let's add a buffer




```
foo:
    pushl %ebp
    movl %esp, %ebp
    subl $32, %esp
    movl $0, %eax
    leave
    ret
```

EIP

```
main:
    pushl %ebp
    movl %esp, %ebp
    call foo
    popl %ebp
    ret
```

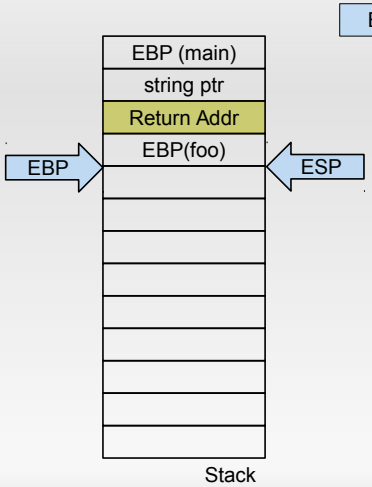
Calling a libC function

```
int foo(char *str)
{
   char buf[20];
   strcpy(buf, str);
   return 0;
}

int main(int argc,
         char *argv[])
{
   return foo(argv[1]);
}
```

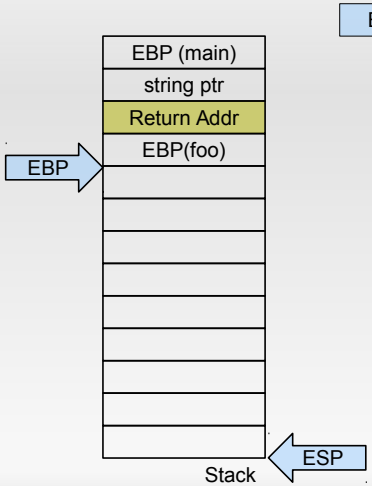
```
foo:
  pushl %ebp
  movl %esp, %ebp
  subl $36, %esp
  movl 8(%ebp), %eax
  movl %eax, 4(%esp)
  leal -28(%ebp), %eax
  movl %eax, (%esp)
  call strcpy
  xorl %eax, %eax
  leave
  ret
```

Calling a libC function



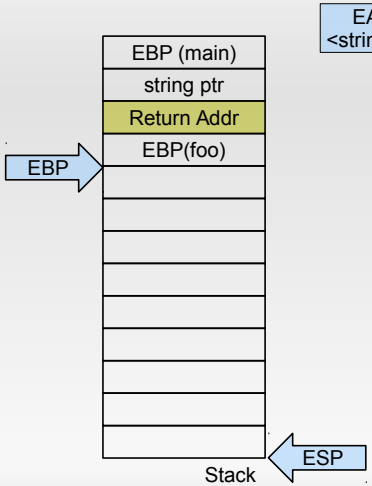

```
foo:
  pushl %ebp
  movl %esp, %ebp
  subl $36, %esp
  movl 8(%ebp), %eax
  movl %eax, 4(%esp)
  leal -28(%ebp), %eax
  movl %eax, (%esp)
  call strcpy
  xorl %eax, %eax
  leave
  ret
```

Calling a libC function




```
foo:
  pushl %ebp
  movl %esp, %ebp
  subl $36, %esp
  movl 8(%ebp), %eax
  movl %eax, 4(%esp)
  leal -28(%ebp), %eax
  movl %eax, (%esp)
  call strcpy
  xorl %eax, %eax
  leave
  ret
```

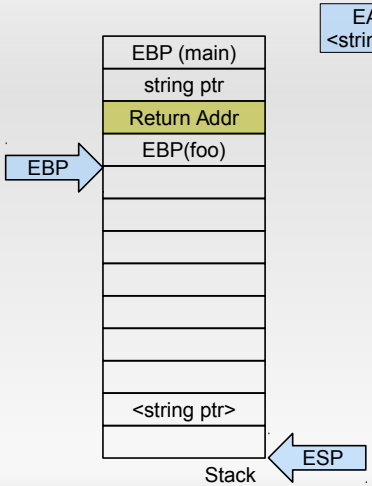
Calling a libC function




```
foo:
  pushl %ebp
  movl %esp, %ebp
  subl $36, %esp
  movl 8(%ebp), %eax
  movl %eax, 4(%esp)
  leal -28(%ebp), %eax
  movl %eax, (%esp)
  call strcpy
  xorl %eax, %eax
  leave
  ret
```

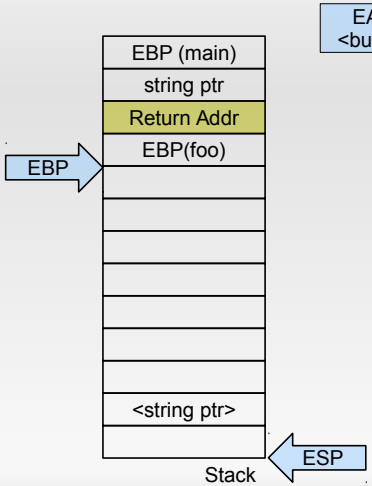
EIP

Calling a libC function



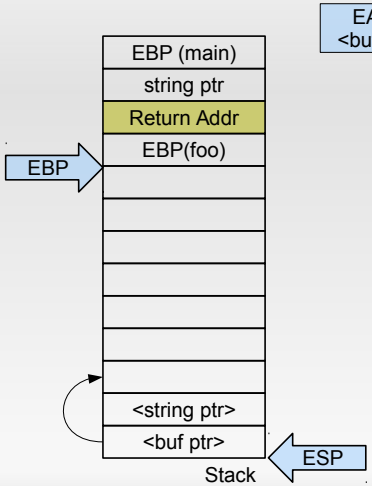

```
foo:
  pushl %ebp
  movl %esp, %ebp
  subl $36, %esp
  movl 8(%ebp), %eax
  movl %eax, 4(%esp)
EIP leal -28(%ebp), %eax
  movl %eax, (%esp)
  call strcpy
  xorl %eax, %eax
  leave
  ret
```

Calling a libC function



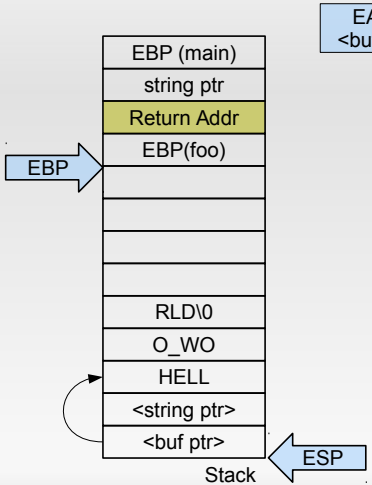

```
foo:
  pushl %ebp
  movl %esp, %ebp
  subl $36, %esp
  movl 8(%ebp), %eax
  movl %eax, 4(%esp)
  leal -28(%ebp), %eax
EIP movl %eax, (%esp)
  call strcpy
  xorl %eax, %eax
  leave
  ret
```

Calling a libC function




```
foo:
  pushl %ebp
  movl %esp, %ebp
  subl $36, %esp
  movl 8(%ebp), %eax
  movl %eax, 4(%esp)
  leal -28(%ebp), %eax
  movl %eax, (%esp)
EIP call strcpy
  xorl %eax, %eax
  leave
  ret
```

Calling a libC function




```
foo:
  pushl %ebp
  movl %esp, %ebp
  subl $36, %esp
  movl 8(%ebp), %eax
  movl %eax, 4(%esp)
  leal -28(%ebp), %eax
  movl %eax, (%esp)
  call strcpy
  xorl %eax, %eax
  leave
  ret
```

EIP

string = ”Hello world”

Assignment: functions

Let's write some code:

1. add two values
2. return the current instruction pointer (rip)
3. return the current stack pointer (rsp)

Functions in assembly

1. How do you create local variables?
2. How do you ensure that control flow of a function does not go into another function?
3. Can address on a stack be one or two bytes, like with jmp?
4. Is it possible to use pop and jmp instead of ret? How?

System calls

	Return	Syscall Number	Args
Linux	RAX	RAX	RDI, RSI, RDX, R10, R8, R9

Max. 6 Arguments for syscalls.

Assignment: functions

Let's write some code:

1. get the process id from the operating system

You will need the `getpid()` system call – number 39 (x86_64).

	Intel	**AT&T**
order	instr dest, src	instr src, dest
size	implicit (by reg. name)	explicit (by instr)
Sigils	automatic	prefixes ($, %)
mem access	[base+index*scale+disp]	disp(base,index,scale)
	[base + disp]	disp(base)

Example	`mov eax,1`
	`mov ebx,0ffh`
	`mov eax,[ebx]`
	`mov eax,[ebx+3]`
		`movl $1,%eax`
		`movl $0xff,%ebx`
		`movl (%ebx),%eax`
		`movl 3(%ebx),%eax`

loops

How would you implement a loop?
Which instructions do you need?

cmp

compare two values

```
cmp $0, %eax
cmp %eax, %ebx
```

cmp

compare two values

```
cmp $0, %eax
cmp %eax, %ebx
```

Where to store the result?

Special purpose register that contains several bits to indicate the result of certain instructions – like cmp.

- 0 CF Carry Flag
- 2 PF Parity Flag
- 6 ZF Zero Flag
- 7 SF Sign Flag
- 8 TF Trap Flag (single step)
- 9 IF Interrupt Enable Flag

https://en.wikipedia.org/wiki/FLAGS_register

jmp

(Conditionally) jump to an address

```
jmp 0xC0FFEE
jmp %eax
ja 0xC0FFEE
jae 0xC0FFEE
jb[e] 0xC0FFEE
jg[e] 0xC0FFEE
jl[e] 0xC0FFEE
jne 0xC0FFEE
jz 0xC0FFEE
```

and lots of others, see the Intel manual:

http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf

Assignment: Hello world

A function which prints "Hello world!" N times.

1. Use directives *.data* and *.text*
2. Make a syscall from within the assembly function
3. Call your function from c code and test it

Assignment: Bitcount

Count the bits in a given integer.

1. write a function bitcount in x86_64 assembly
2. call your function from c code and test it

```
asm [volatile] ( AssemblerTemplate
                      : OutputOperands
                      [ : InputOperands
                      [ : Clobbers ] ])
```

```
int i = 42;
asm volatile ("add␣%0,␣%0;"
              : "+r"(i)
              : // no other input, just i
              : // no clobber
              );
```

Register Constraints and Modifiers

```
asm volatile ("add %0, %0;" : "+r"(i) );
```

	Constraints		Modifiers
r	any general purpose register	=	write only operand
a	al, ax, eax, rax	+	read / write
c	cl, cx, ecx, rcx	&	early clobber
D	edi, rdi		
m	memory operand		

add

```
int add(int a, int b) {
  asm volatile ("add %1, %0;"
                : "+r"(a) : "r"(b) );
  return b;
}
```

Additional Registers

- SSE adds 16 new 128bit registers – xmm0 - xmm15.
- Must be explicitly enabled by the OS.
- Eases and accelerates vector computations.

For a full description see Intel Manual (Volume 1, Chapter 10).

Assignment: functions

Let's write some code:

1. add two vectors using SSE
2. multiply two vectors using SSE

Intel Software Developer Manual

http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf

X86 Calling Conventions

https://en.wikipedia.org/wiki/X86_calling_conventions

FLAGS register

https://en.wikipedia.org/wiki/FLAGS_register

Compiler Builtins

GCC (and others) come with special intrinsics that map to optimized code. Examples:

Compiler Builtins

GCC (and others) come with special intrinsics that map to optimized code. Examples:

- Common libC functions (`__builtin_memcpy`)
- `__builtin_expect()`
- `__builtin_popcount()`
- `__builtin_prefetch()`
- `__builtin_bswap32()`
- `__builtin_return_address()`
- `__builtin_ia32_addps()`

Obtaining EIP

```
unsigned long long
__attribute__((noinline))
eip()
{
  return __builtin_return_address(0);
}
```

Counting bits

```
unsigned count_bits(unsigned x)
{
  return __builtin_popcount(x);
}
```

SSE

```
typedef float v4sf
    __attribute__((vector_size(16))); // Hah!

void sse() {
  v4sf v1 = {1,2,3,4};
  v4sf v2 = {1,2,3,4};
  v4sf v3 = {2,2,2,2};
  v4sf res;

  res = __builtin_ia32_mulps(v3,
          __builtin_ia32_addps(v1, v2));

  printf("res = [%f,%f,%f,%f]\n", res[0],
         res[1], res[2], res[3]);
}
```

... or

```
typedef float v4sf
    __attribute__((vector_size(16))); // Hah!

void sse() {
  v4sf v1 = {1,2,3,4};
  v4sf v2 = {1,2,3,4};
  v4sf v3 = {2,2,2,2};
  v4sf res;

  res = v3 * (v1 + v2);

  printf("res = [%f,%f,%f,%f]\n", res[0],
         res[1], res[2], res[3]);
}
```

How much is my code?

You will always need to understand the cost of your code:

- Memory / resource consumption
  - Memory consumption in GiB?
  - Binary size
  - Energy consumption

How much is my code?

You will always need to understand the cost of your code:

- Memory / resource consumption
  - Memory consumption in GiB?
  - Binary size
  - Energy consumption
- Implementation cost
  - Source Lines of Code
  - Cyclomatic Complexity

How much is my code?

You will always need to understand the cost of your code:

- Memory / resource consumption
  - Memory consumption in GiB?
  - Binary size
  - Energy consumption
- Implementation cost
  - Source Lines of Code
  - Cyclomatic Complexity
- Execution time
  - Execution time in seconds → `gettimeofday()`
  - Short running code → CPU cycles

CPU Time Stamp Counter

64 bit register counting the clocks since system startup.

- Pentium*, early Xeon CPUs: increment with every CPU cycle.
- Newer Xeons and Core*: increment at a constant rate.
- AMD up to K8: per CPU, increment with every CPU cycle

Spot the problem, anyone?

Reading the TSC

Instruction: `rdtsc` stores TSC in
EAX (lower 32 bits) and EDX (higher 32 bits).

Reading the TSC

Instruction: `rdtsc` stores TSC in
EAX (lower 32 bits) and EDX (higher 32 bits).

```
unsigned long long rdtsc() {
unsigned long long hi, lo;

 asm volatile("rdtsc\n\t"
              "mov␣%edx,␣%0\n\t"
              "mov␣%eax,␣%1\n\t"
     : "=r" (hi), "=r" (lo));

 return (hi << 32) | lo;
}
```

Clobbering matters!

```
unsigned long long rdtsc() {
unsigned long long hi, lo;

 asm volatile("rdtsc"
              "mov␣%edx,␣%0\n\t"
              "mov␣%eax,␣%1\n\t"
     : "=r" (hi), "=r" (lo)
     :
     : "eax", "edx");

 return (hi << 32) | lo;
}
```

Catching out-of-order execution[1]

Before a measurement:

```
unsigned long long rdtsc_pre() {
unsigned long long hi, lo;

 asm volatile("cpuid; rdtsc"
              "mov %edx, %0\n\t"
              "mov %eax, %1\n\t"
    : "=r" (hi), "=r" (lo)
    :
    : "rax", "rbx", "rcx", "rdx");

  return (hi << 32) | lo;
}
```

[1] How to Benchmark Code Execution Times on Intel ® IA-32 and IA-64 Instruction SetArchitectures. Gabriele Paoloni

Catching out-of-order execution

After a measurement:

```
unsigned long long rdtsc_post() {
unsigned long long hi, lo;

 asm volatile("rdtscp\n\t"
              "mov %edx, %0\n\t"
              "mov %eax, %1\n\t"
              "cpuid\n\t"
    : "=r" (hi), "=r" (lo)
    :
    : "rax", "rbx", "rcx", "rdx");

  return (hi << 32) | lo;
}
```

Benchmarking Considerations

- RTSC is not for free.

Benchmarking Considerations

- RTSC is not for free.
- Interruption by other programs, migration.
  - Own OS: measure in kernel and disable IRQs.
  - Linux user space: difficult
    - Set CPU affinity
    - Collect 1000s of samples and ignore outliers

Register Names

Did you know register names are there for a reason?

- (R/E)SP – stack pointer
- (R/E)BP – base pointer
- (R/E)IP – instruction pointer

Register Names

Did you know register names are there for a reason?

- (R/E)SP – stack pointer
- (R/E)BP – base pointer
- (R/E)IP – instruction pointer
- (R/E)AX – accumulator
- (R/E)BX – base register
- (R/E)CX – counter register
- (R/E)DX – extenDed accumulator
- (R/E)SI – source index
- (R/E)DI – destination index

Better Loops

`loop <LBL>`

- Decrement the counter register (ECX)
- If ECX is not zero, jump to LBL (conditional jmp)

```
    mov $10, %ecx
.L1:
    add %eax, %ebx
    loop .L1
```

Assignment

Implement the following function in assembly:

```
/*
 * Takes the argument <buf> if length size,
 * reverses it and stores the result in the
 * location of the original <buf>.
 *
 * Returns the number of bytes reversed.
 */
unsigned reverse_buf(char *buf, size_t size);
```

Source and Destination Index?

`movs, movsb, movsw, movsl, movsq`

- Move one byte/word/dword/quadword from DS:ESI to ES:EDI
- Linux sets all segments to whole AS, so we can ignore them here
- Advance ESI and EDI by number of bytes copied
  - The direction flag (DF) decides, whether they are incremented or decremented

Copying multiple bytes?

String instructions (INS, MOVS, OUTS, LODS, STOS, CMPS, SCAS) can be prefixed with a **REP** prefix.

This repeats the string instruction for the number of times specified in ECX.

```
mov $10, %ecx
mov $0xCOFFEE, %esi
mov $0xF00BA4, %edi
rep movsb // memcpy(0xc0ffee, 0xf00ba4, 10)
```

Counting Lines

Implement the following function in assembly:

```
/*
 * Gets a file descriptor to an open file and
 * iterates over the file's content to count the
 * number of lines in the file. (A.k.a an ASM
 * equivalent of 'wc -l' on the shell.
 */
unsigned count_lines(int fd);
```

What we've learned

- Assembly instruction format
- Decoding rules
- Some of assembly instructions
- Calling conventions
- How to program in assembly