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 control over your hardware (using specific instructions)

system programming (e.g. kernel entry/exit)
We need to go deeper: Fibonacci

```c
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

```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
Fibonacci

fib.c

```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
$ 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

```bash
$ objdump -h fib.o

fib.o: file format elf64-x86-64

Sections:
<table>
<thead>
<tr>
<th>Idx</th>
<th>Name</th>
<th>Size</th>
<th>...</th>
<th>File off</th>
<th>Algn</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>.text</td>
<td>00000023</td>
<td>...</td>
<td>00000040</td>
<td>2**4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CONTENTS, ALLOC, LOAD, READONLY, CODE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>.data</td>
<td>00000000</td>
<td>...</td>
<td>00000063</td>
<td>2**0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CONTENTS, ALLOC, LOAD, DATA</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

...
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

83ef0185ff7e16ba0100000031c966908d041189d189c283ef0175f4c3b801000000c3
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

83ef0185ff7e16ba0100000031c966908d041189d189c283ef0175f4c3b8010000000c3
What sees a processor

83ef0185ff7e16ba0100000031c966908d041189d189c283ef0175f4c3b801000000c3


$ wget http://svn.inf.tu-dresden.de/repos/advsysprog/asm/opcodes.pdf
What sees a processor

83ef0185ff7e16ba0100000031c966908d041189d189c283ef0175f4c3b801000000c3


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

Table[0x8, 0x3] = Immediate Grp 1 : Ev, Ib
What 0x83 stands for?

Ev, Ib

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

l  Immediate data

b  Byte

Need to look into next byte
ModR/M

83ef0185ff7e16ba0100000031c966908d041189d189c283ef0175f4c3b801000000c3
ModR/M

83ef0185ff7e16ba0100000031c966908d041189d189c283ef0175f4c3b801000000c3

$$\begin{array}{|c|c|c|}
\hline
\text{Mod} & \text{Reg/Opcode} & \text{R/M} \\
\hline
11 & 101 & 111 \\
\hline
\end{array}$$
ModR/M

\[ 83\text{ef}01\text{85ff}7\text{e}16\text{ba}0100000031\text{c}966908\text{d}041189\text{d}189\text{c}283\text{ef}0175\text{f}4\text{c}3\text{b}801000000\text{c}3 \]

\[
\begin{array}{ccc}
\text{Mod} & \text{Reg/Opcode} & \text{R/M} \\
7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
1 & 1 & 1 & 0 & 1 & 1 & 1 \\
\end{array}
\]

\[
\text{Mod}(11) + \text{R/M}(111) \rightarrow \text{edi}
\]
\[
\text{Opcode}(101) \rightarrow \text{sub}
\]
Immediate data

```
sub imm8, %edi
```
Immediate data

\textbf{sub imm8, } \%edi

\texttt{83ef0185ff7e16ba0100000031c966908d041189d189c283ef0175f4c3b801000000c3}

Look into next byte
Immediate data

```
  sub  imm8, %edi
```

```
83ef0185ff7e16ba0100000031c966908d041189d189c283ef0175f4c3b801000000c3
```

Look into next byte
Immediate data

\[
\text{sub } \text{imm8}, \%\text{edi}
\]

83ef0185ff7e16ba0100000031c966908d041189d189c283ef0175f4c3b801000000c3

Look into next byte

\[
\text{sub } \$0x1, \%\text{edi}
\]
Next instruction

83ef0185ff7e16ba0100000031c966908d041189d189c283ef0175f4c3b801000000c3
Next instruction

85ff7e16ba0100000031c966908d041189d189c283ef0175f4c3b801000000c3
Next instruction

85ff7e16ba0100000031c966908d041189d189c283ef0175f4c3b801000000c3

Table[8, 5] = Test : Ev, Gv
Next instruction

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

$G$ ModR/M byte selects a general register
Next introduction (ModR/M)

85ff7e16ba0100000031c966908d041189d189c283ef0175f4c3b801000000c3
Next introduction (ModR/M)

```
85ff7e16ba0100000031c966908d041189d189c283ef0175f4
c3b801000000c3
```

<table>
<thead>
<tr>
<th>Mod</th>
<th>Reg/Opcode</th>
<th>R/M</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>111</td>
<td>111</td>
</tr>
</tbody>
</table>
Next introduction (ModR/M)

\[ \text{ModR/M} \]

\[ 85\text{ff7e16ba0100000031c966908d041189d189c283ef0175f4c3b801000000c3} \]

<table>
<thead>
<tr>
<th>Mod</th>
<th>Reg/Opcode</th>
<th>R/M</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>111</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>111</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>111</td>
</tr>
</tbody>
</table>

\[ \text{Mod}(11) + \text{R/M}(111) \rightarrow \text{edi} \]
\[ \text{Reg}(111) \rightarrow \text{edi} \]
Next introduction (ModR/M)

```
85ff7e16ba0100000031c966908d041189d189c283ef0175f4
c3b801000000c3
```

<table>
<thead>
<tr>
<th>Mod</th>
<th>Reg/Opcode</th>
<th>R/M</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>111</td>
<td>111</td>
</tr>
</tbody>
</table>

Mod(11) + R/M(111) → edi
Reg(111) → edi

```
test %edi, %edi
```
Continue decoding

85ff7e16ba0100000031c966908d041189d189c283ef0175f4c3b801000000c3
Continue decoding

7e16ba0100000031c966908d041189d189c283ef0175f4c3b8
010000000c3
Continue decoding

7e16ba0100000031c966908d041189d189c283ef0175f4c3b801000000c3

Table[7, e] = jle

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

`7e16ba0100000031c966908d041189d189c283ef0175f4c3b801000000c3`

Table[7, e] = jle

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

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

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

\[\text{jle } \$0x16\]

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 00 mov $0x1, %eax
  22:  c3     retq
This chapter describes the instruction format for all Intel 64 and IA-32 processors. The instruction format for protected mode, real-address mode and virtual-8086 mode is described in Section 2.1. Increments provided for IA-32e mode and its sub-modes are described in Section 2.2.

2.1 INSTRUCTION FORMAT FOR PROTECTED MODE, REAL-ADDRESS MODE, AND VIRTUAL-8086 MODE

The Intel 64 and IA-32 architectures instruction encodings are subsets of the format shown in Figure 2-1. Instructions consist of optional instruction prefixes (in any order), primary opcode bytes (up to three bytes), an addressing-form specifier (if required) consisting of the ModR/M byte and sometimes the SIB (Scale-Index-Base) byte, a displacement (if required), and an immediate data field (if required).

2.1.1 Instruction Prefixes

Instruction prefixes are divided into four groups, each with a set of allowable prefix codes. For each instruction, it is only useful to include up to one prefix code from each of the four groups (Groups 1, 2, 3, 4). Groups 1 through 4 may be placed in any order relative to each other.

- **Group 1**
  - **Lock and repeat prefixes:**
    - LOCK prefix is encoded using F0H.
    - REPNE/REPNZ prefix is encoded using F2H. Repeat-Not-Zero prefix applies only to string and input/output instructions. (F2H is also used as a mandatory prefix for some instructions.)
    - REP or REPE/REPZ is encoded using F3H. The repeat prefix applies only to string and input/output instructions. F3H is also used as a mandatory prefix for POPCNT, LZCNT and ADOX instructions.

- **Group 2**
  - Bound prefix is encoded using F2H if the following conditions are true:
    - CPUID.(EAX=07H, ECX=0):EBX.MPX[bit 14] is set.

![Figure 2-1. Intel 64 and IA-32 Architectures Instruction Format](image)

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

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.
2.2.1 REX Prefixes

REX prefixes are instruction-prefix bytes used in 64-bit mode. They do the following:

- Specify GPRs and SSE registers.
- Specify 64-bit operand size.
- Specify extended control registers.

Not all instructions require a REX prefix in 64-bit mode. A prefix is necessary only if an instruction references one of the extended registers or uses a 64-bit operand. If a REX prefix is used when it has no meaning, it is ignored.

Only one REX prefix is allowed per instruction. If used, the REX prefix byte must immediately precede the opcode byte or the escape opcode byte (0FH). When a REX prefix is used in conjunction with an instruction containing a mandatory prefix, the mandatory prefix must come before the REX so the REX prefix can be immediately preceding the opcode or the escape byte. For example, CVTDQ2PD with a REX prefix should have REX placed between F3 and 0F E6. Other placements are ignored. The instruction-size limit of 15 bytes still applies to instructions with a REX prefix. See Figure 2-3.

2.2.1.1 Encoding

Intel 64 and IA-32 instruction formats specify up to three registers by using 3-bit fields in the encoding, depending on the format:

- ModR/M: the reg and r/m fields of the ModR/M byte
- ModR/M with SIB: the reg field of the ModR/M byte, the base and index fields of the SIB (scale, index, base) byte
- Instructions without ModR/M: the reg field of the opcode

In 64-bit mode, these formats do not change. Bits needed to define fields in the 64-bit context are provided by the addition of REX prefixes.

2.2.1.2 More on REX Prefix Fields

REX prefixes are a set of 16 opcodes that span one row of the opcode map and occupy entries 40H to 4FH. These opcodes represent valid instructions (INC or DEC) in IA-32 operating modes and in compatibility mode. In 64-bit mode, the same opcodes represent the instruction prefix REX and are not treated as individual instructions.

The single-byte-opcode forms of the INC/DEC instructions are not available in 64-bit mode. INC/DEC functionality is still available using ModR/M forms of the same instructions (opcodes FF/0 and FF/1).

See Table 2-4 for a summary of the REX prefix format. Figure 2-4 though Figure 2-7 show examples of REX prefix fields in use. Some combinations of REX prefix fields are invalid. In such cases, the prefix is ignored. Some additional information follows:

- Setting REX.W can be used to determine the operand size but does not solely determine operand width. Like the 66H size prefix, 64-bit operand size override has no effect on byte-specific operations.
- For non-byte operations: if a 66H prefix is used with prefix (REX.W = 1), 66H is ignored.
- If a 66H override is used with REX and REX.W = 0, the operand size is 16 bits.

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?
mov
move data between registers or to/from memory

movl $1, %eax
movl $0xff, %ebx
movl (%ebx), %eax
movl 3(%ebx), %eax
add/sub

addition / substraction

add $1 , %eax
add %eax , %ebx
sub $1 , %eax
sub %eax , %ebx
and/or/xor/test

logical operations

and %eax,%ebx
or %eax,%ebx
xor %eax,%ebx
test %eax,%ebx
push/pop

push or pop register content to or from the stack

push %eax
pop %eax
pusha
popa
call

call a function

call 0xC0FFEE

call 0xBADA55

ret
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
<table>
<thead>
<tr>
<th></th>
<th>Par. Reg</th>
<th>Par. Stack</th>
<th>Cleanup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microsoft</td>
<td>RCX, RDX, R8, R9</td>
<td>RTL(C) Caller</td>
<td>Caller</td>
</tr>
<tr>
<td>System V</td>
<td>RDI, RSI, RDX, RCX</td>
<td>R8, R9</td>
<td></td>
</tr>
</tbody>
</table>
## x86_64

<table>
<thead>
<tr>
<th></th>
<th>Par. Reg</th>
<th>Par. Stack</th>
<th>Cleanup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microsoft</td>
<td>RCX, RDX, R8, R9</td>
<td>RTL(C)</td>
<td>Caller</td>
</tr>
<tr>
<td>System V</td>
<td>RDI, RSI, RDX, RCX</td>
<td>RTL(C)</td>
<td>Caller</td>
</tr>
<tr>
<td></td>
<td>R8, R9</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Return</th>
<th>Callee Saved</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microsoft</td>
<td>RAX</td>
<td>RBX, RBP, RDI, RSI, R12 - R15</td>
</tr>
<tr>
<td>System V</td>
<td>RAX</td>
<td>RBX, RBP, R12-R15</td>
</tr>
</tbody>
</table>
Buffers on the stack

Stolen from DOS...
The Battlefield: x86/32

- General-purpose registers: EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP
- Instruction pointer: EIP
- Segment, FPU, control, MMX, … registers

Address Space:
- Text
- Data
- BSS
- Stack
- Kernel

Exploitz
The Stack

- Stack frame per function
  - Set up by compiler-generated code
- Used to store
  - Function parameters
  - If not in registers – GCC:
    ```c
    __attribute__((regparm((<num>))))
    ```
  - Local variables
  - Control information
    - Function return address
```c
int sum(int a, int b)
{
    return a+b;
}

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

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

```
main:
    pushl %ebp
    movl %esp, %ebp
    subl $8, %esp
    movl $3, 4(%esp)
    movl $1, (%esp)
    call sum
    ret
```
%<reg> refers to register content

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

Constants prefixed with $ sign

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

Exploitz
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?

```assembly
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?

```c
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?

```assembly
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?

```
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
```

Stack

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBP (main)</td>
<td>3</td>
</tr>
<tr>
<td>EBP (main)</td>
<td>1</td>
</tr>
<tr>
<td>Return Addr</td>
<td></td>
</tr>
</tbody>
</table>

Exploitz

EIP

ESP

EBP (main)

EIP

EBP
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?

```assembly
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?

```
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
```

Stack

EBP (main)
3
1

Return Addr
EBP (sum)

EBP
ESP

EIP

EAX: 4

Exploitz
So what happens on a call?

Exploitz

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
Now let's add a buffer

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

int main()
{
    return foo();
}
```
Now let's add a buffer

```
foo:
   pushl %ebp
   movl %esp, %ebp
   subl $32, %esp
   movl $0, %eax
   ... %ebp
   movl %esp, %ebp
   call foo
   popl %ebp
   ret
```

```
main:
   pushl %ebp
   movl %esp, %ebp
   call foo
   popl %ebp
   ret
```
Now let's add a buffer

Exploitz

```asm
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
```

Stack

- EBP (main)
- EBP(foo)
- Return Addr
- buf
- EIP
- ESP

Exploitz
**Calling a libC function**

```c
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
```

Stack

EBP (main)
string ptr
Return Addr
EBP(foo)

EBP
ESP
EIP
EAX

Exploitz
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

```
Exploitz
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
```

Stack

<table>
<thead>
<tr>
<th>EBP (main)</th>
</tr>
</thead>
<tbody>
<tr>
<td>string ptr</td>
</tr>
<tr>
<td>Return Addr</td>
</tr>
<tr>
<td>EBP(foo)</td>
</tr>
</tbody>
</table>

EAX: `<string ptr>`

EIP

```
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
```

Stack

- EBP (main)
- string ptr
- Return Addr
- EBP(foo)

EAX: <buf ptr>

- <string ptr>

Exploitz
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
```

Stack

- EBP (main)
- string ptr
- EBP(foo)
- <string ptr>
- <buf ptr>

EBP

EIP

<buf ptr>
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
```

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

<table>
<thead>
<tr>
<th>Linux</th>
<th>Return</th>
<th>Syscall Number</th>
<th>Args</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RAX</td>
<td>RAX</td>
<td>RDI, RSI, RDX, R10, R8, R9</td>
</tr>
</tbody>
</table>

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).
<table>
<thead>
<tr>
<th>Order</th>
<th>Intel</th>
<th>AT&amp;T</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>instr dest, src</td>
<td>instr src, dest</td>
</tr>
<tr>
<td>Sigils</td>
<td>implicit (by reg. name)</td>
<td>explicit (by instr)</td>
</tr>
<tr>
<td>Mem access</td>
<td>[base+index*scale+disp]</td>
<td>disp(base,index, scale)</td>
</tr>
<tr>
<td></td>
<td>[base + disp]</td>
<td>disp(base)</td>
</tr>
</tbody>
</table>

**Example**

<table>
<thead>
<tr>
<th>Intel</th>
<th>AT&amp;T</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov eax, 1</td>
<td>movl $1, %eax</td>
</tr>
<tr>
<td>mov ebx, 0ffh</td>
<td>movl $0xff, %ebx</td>
</tr>
<tr>
<td>mov eax, [ebx]</td>
<td>movl (%ebx), %eax</td>
</tr>
<tr>
<td>mov eax, [ebx+3]</td>
<td>movl 3(%ebx), %eax</td>
</tr>
</tbody>
</table>
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
- **4** AF Adjust 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

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
int i = 42;
asm volatile ("add %0, %0;
    : "+r"(i)
    : // no other input, just i
    : // no clobber
);
Register Constraints and Modifiers

\[
\text{asm volatile } ( "\text{add } \%0, \%0;" : "+r\text{ }^\text{(i)} \) );
\]

<table>
<thead>
<tr>
<th>Constraints</th>
<th>Modifiers</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>any general purpose register</td>
</tr>
<tr>
<td>a</td>
<td>write only operand</td>
</tr>
<tr>
<td>c</td>
<td>read / write</td>
</tr>
<tr>
<td>D</td>
<td>early clobber</td>
</tr>
<tr>
<td>m</td>
<td>memory operand</td>
</tr>
</tbody>
</table>
```c
int add(int a, int b) {
    asm volatile ("add %0, %1;"
                  : "+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

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_memcpry)
- __builtin_expect()
- __builtin_popcount()
- __builtin_prefetch()
- __builtin_bswap32()
- __builtin_return_address()
- __builtin_ia32_addps()
Obtaining EIP

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

```c
unsigned count_bits(unsigned x)
{
    return __builtin_popcount(x);
}
```
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]);
}
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).

```c
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!

```c
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 execution¹

Before a measurement:

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

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

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

¹How to Benchmark Code Execution Times on Intel ® IA-32 and IA-64 Instruction Set Architectures. Gabriele Paoloni
Catching out-of-order execution

After a measurement:

```c
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:

```c
/*
 * 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 $0xC0FFEE, %esi
mov $0xF00BA4, %edi
rep movsb  // memcpy(0xc0ffee, 0xf00ba4, 10)
```
Counting Lines

Implement the following function in assembly:

```c
/*
 * 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