INTRODUCTION
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)
What is the result of the following instruction?

```
mov %eax, %ebx
```
**INSTRUCTIONS**

**mov**
move data between registers or to/from memory

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

add/sub
addition / substraction

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

and/or/xor

logical operations

\textbf{and} \hspace{0.5cm} \%eax,\%ebx

\textbf{or} \hspace{0.5cm} \%eax,\%ebx

\textbf{xor} \hspace{0.5cm} \%eax,\%ebx
INSTRUCTIONS

push/pop
push or pop register content to or from the stack

push %eax
pop %eax
pusha
popa
INSTRUCTIONS

call
  call a function
  call 0xC0FFEE
  call 0xBADA55
  ret
THE STACK

shamelessly steal DOS slides...
CALLING CONVENTIONS
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
# Calling Conventions

## 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>
# Calling Conventions

## x86_64

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

<table>
<thead>
<tr>
<th></th>
<th>Return</th>
<th>Callee Saved</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Microsoft</strong></td>
<td>RAX</td>
<td>RBX, RBP, RDI, RSI, R12 - R15</td>
</tr>
<tr>
<td><strong>System V</strong></td>
<td>RAX</td>
<td>RBX, RBP, R12-R15</td>
</tr>
</tbody>
</table>
EXERCISE
functions

Let’s write some code:

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

System calls

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

Max. 6 Arguments for syscalls.
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).
## Syntax

<table>
<thead>
<tr>
<th></th>
<th>Intel</th>
<th>AT&amp;T</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>order</strong></td>
<td>instr dest, src</td>
<td>instr src, dest</td>
</tr>
<tr>
<td><strong>size</strong></td>
<td>implicit (by reg. name)</td>
<td>explicit (by instr)</td>
</tr>
<tr>
<td><strong>Sigils</strong></td>
<td>automatic</td>
<td>prefixes ($, %)</td>
</tr>
<tr>
<td><strong>mem access</strong></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>
<tr>
<td><strong>Example</strong></td>
<td>mov eax, 1</td>
<td>movl $1, %eax</td>
</tr>
<tr>
<td></td>
<td>mov ebx, 0ffh</td>
<td>movl $0xff, %ebx</td>
</tr>
<tr>
<td></td>
<td>mov eax, [ebx]</td>
<td>movl (%ebx), %eax</td>
</tr>
<tr>
<td></td>
<td>mov eax, [ebx+3]</td>
<td>movl 3(%ebx), %eax</td>
</tr>
</tbody>
</table>
INSTRUCTIONS
loops

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

cmp
compare two values

cmp $0, %eax

cmp %eax, %ebx
INSTRUCTIONS

cmp
compare two values

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

Where to store the result?
FLAGS

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
INSTRUCTIONS

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

EXERCISE
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
**INLINE ASSEMBLY**

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

```asm
c volatile
("add %0, %0;");
```

<table>
<thead>
<tr>
<th>Constraints</th>
<th>Modifiers</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>r</code></td>
<td>any general purpose register</td>
</tr>
<tr>
<td><code>a</code></td>
<td>al, ax, eax, rax</td>
</tr>
<tr>
<td><code>c</code></td>
<td>cl, cx, ecx, rcx</td>
</tr>
<tr>
<td><code>D</code></td>
<td>edi, rdi</td>
</tr>
<tr>
<td><code>m</code></td>
<td>memory operand</td>
</tr>
<tr>
<td></td>
<td><code>=</code> write only operand</td>
</tr>
<tr>
<td></td>
<td><code>+</code> read / write</td>
</tr>
<tr>
<td></td>
<td><code>&amp;</code> early clobber</td>
</tr>
</tbody>
</table>
**INLINE ASSEMBLY**

```c
int add(int a, int b) {
    asm volatile("add %0, %1;
                 : \"+r\"(a) : \"+r(b)\")
    return b;
}
```
SSE
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).
EXERCISE
functions

Let’s write some code:

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

Intel Software Developer Manual

X86 Calling Conventions
https://en.wikipedia.org/wiki/X86_calling_conventions

FLAGS register
https://en.wikipedia.org/wiki/FLAGS_register
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()`
BUILTINS

Obtaining EIP

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

Counting bits

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

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

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
BENCHMARKING
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
Benchmarking
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 $\rightarrow \text{gettimeofday()}$
  - Short running code $\rightarrow$ CPU cycles
BENCHMARKING
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?
BENCHMARKING

Reading the TSC

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

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;
}
```
BENCHMARKING
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 execution

Before a measurement:

```c
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)
        :
        : "eax", "ebx", "ecx", "edx";

    return (hi << 32) | lo;
}
```
Catching out-of-order execution

After a measurement:

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

    asm volatile (
        "rdtscp;\n        cpuid\n        mov %edx, %0\n        mov %eax, %1\n        : "=r" (hi), "=r" (lo)
        : "eax", "ebx", "ecx", "edx";

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

Live coding
More ASM Stuff

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
MORE ASM STUFF

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
MORE ASM STUFF
Better Loops

loop <LBL>

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

```assembly
mov $10, %ecx
.L1:
  add %eax, %ebx
loop .L1
```
MORE ASM STUFF
Buffers on the stack

Again, stolen from DOS...
Exercise

Implement the following function in assembly:

```c
unsigned reverse_buf(char *buf, size_t size);
```
MORE ASM STUFF
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.

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