Why do we look at machine code?

- understanding how the high-level programming language instructions are executed on a processor
- understanding how optimizing high-level program affects instructions executed in practice
- understanding security flaws of programs
- understanding things that are not handled at the high-level programming language

We will be working with the machine code for x86-64 processors.
A Bit of History
Intel x86 Processors

- Totally dominate laptop/desktop/server market

- Evolutionary design
  - Backwards compatible up until 8086, introduced in 1978
  - Added more features as time goes on

- Complex instruction set computer (CISC)
  - Many different instructions with many different formats
    - But, only small subset encountered with Linux programs
  - Hard to match performance of Reduced Instruction Set Computers (RISC)
  - But, Intel has done just that!
    - In terms of speed. Less so for low power.

We will just scratch the surface of the available instructions.
## Intel x86 Evolution

<table>
<thead>
<tr>
<th>Name</th>
<th>Date</th>
<th>Transistors</th>
<th>MHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>● 8086</td>
<td>1978</td>
<td>29K</td>
<td>5-10</td>
</tr>
<tr>
<td>○ First 16-bit processor. Basis for IBM PC &amp; DOS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>○ 1MB address space</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>● 386</td>
<td>1985</td>
<td>275K</td>
<td>16-33</td>
</tr>
<tr>
<td>○ First 32 bit processor, referred to as IA32</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>○ Added “flat addressing”</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>○ Capable of running Unix</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>○ 32-bit Linux/gcc uses no instructions introduced in later models</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>● Pentium 4F</td>
<td>2004</td>
<td>125M</td>
<td>2800-3800</td>
</tr>
<tr>
<td>○ First 64-bit processor, referred to as x86-64</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>● Core i7</td>
<td>2008</td>
<td>731M</td>
<td>2667-3333</td>
</tr>
</tbody>
</table>

Notice that the speed is not increasing as much any more.
Schematic of an Intel Processor
Another schematic of an Intel Processor

- Core i7 Broadwell 2015

- Desktop Model
  - 4 cores
  - Integrated graphics
  - 3.3-3.8 GHz
  - 65W

- Server Model
  - 8 cores
  - Integrated I/O
  - 2-2.6 GHz
  - 45W
X86 Close: s Advanced Micro Devices (AMD)

● Historically
  ○ AMD has followed just behind Intel
  ○ A little bit slower, a lot cheaper

● Then
  ○ Recruited top circuit designers from Digital Equipment Corp. and other downward trending companies
  ○ Built Opteron: tough competitor to Pentium 4
  ○ Developed x86-64, their own extension to 64 bits

● Recent Years
  ○ Intel got its act together ⇒ Leads the world in semiconductor technology
  ○ AMD has fallen behind ⇒ Relies on external semiconductor manufacturer
64-bit History

● 2001: Intel Attempts Radical Shift from IA32 to IA64
  ○ Totally different architecture (Itanium)
  ○ Executes IA32 code only as legacy
  ○ Performance disappointing

● 2003: AMD Steps in with Evolutionary Solution
  ○ x86-64 (now called “AMD64”)

● Intel Felt Obligated to Focus on IA64
  ○ Hard to admit mistake or that AMD is better

● 2004: Intel Announces EM64T extension to IA32
  ○ Extended Memory 64-bit Technology
  ○ Almost identical to x86-64!

● All but low-end x86 processors support x86-64
  ○ But, lots of code still runs in 32-bit mode
C, assembly, machine code
Definitions

- **Architecture**: (also ISA: instruction set architecture) The parts of a processor design that one needs to understand or write assembly/machine code.
  - Examples: instruction set specification, registers.
  - Target of the compiler

- **Microarchitecture**: Implementation of the architecture.
  - Examples: cache sizes and core frequency.

- **Code Forms**:
  - Machine Code: The byte-level programs that a processor executes
  - Assembly Code: A text representation of machine code

- **Example ISAs**:
  - Intel: x86, IA32, Itanium, x86-64
  - ARM: Used in almost all mobile phones
Assembly/Machine Code View of a Computer

Programmer-Visible State

- **PC**: Program counter
  - Address of next instruction
  - Called “RIP” (x86-64)
- **Register file**
  - Heavily used program data
- **Condition codes**
  - Store status information about most recent arithmetic or logical operation
  - Used for conditional branching

- **Memory**
  - Byte addressable array
  - Code and user data
  - Stack to support procedures
Turning C into Object Code

- Code in files p1.c p2.c
- Compile with command: `gcc -Og p1.c p2.c -o p`
  - Use basic optimizations (-Og) [New to recent versions of GCC]
  - Put resulting binary in file p
Compiling into Assembly

```c
long plus(long x, long y) {
    return x + y;
}

void sumstore(long x, long y, long *dest) {
    long t = plus(x, y);
    *dest = t;
}
```

Assembly code generated using

```
gcc -Og -S sum.c
```

output written to `sum.s` file.

**Note1:** the assembly code will be different for different versions of gcc and different compiler settings. The generated code should be equivalent in terms of what it does, though.

**Note2:** for now we ignore all instructions in the `.s` file that start with a dot - they are not really part of the assembly.
Assembly Characteristics: Data Types

- “Integer” data of 1, 2, 4, or 8 bytes
  - Data values (it does not matter if it is signed or not at the level of assembly)
  - Addresses (untyped pointers)

- Floating point data of 4, 8, or 10 bytes
  - We will not really go into floating point numbers at the level of assembly

- Code: Byte sequences encoding series of instructions

- No aggregate types such as arrays or structures, just contiguously allocated bytes in memory
Assembly Operations

- Perform arithmetic function on register or memory data
- Transfer data between memory and register
  - Load data from memory into register
  - Store register data into memory
- Transfer control
  - Unconditional jumps to/from procedures
  - Conditional branches

Very limited in what can be done in one instruction - does only one thing: move data, single simple arithmetic operation, memory dereference.
Object Code

- **Assembler**
  - Translates .s into .o
  - Binary encoding of each instruction
  - Nearly-complete image of executable code
  - Missing linkages between code in different files

- **Linker**
  - Resolves references between files
  - Combines with static run-time libraries
    - E.g., code for malloc, printf
  - Some libraries are dynamically linked
    - Linking occurs when program begins execution

```
0x4005a2:
  0x53
  0x48
  0x89
  0xd3
  0xe8
  0xf2
  0xff
  0xff
  0xff
  0xf2
  0xff
  0xff
  0x48
  0x89
  0x03
  0x5b
  0xc3
```

Total of 14 bytes. Each instruction can use a different number of bytes. Stats at location 0x4005a2.
Machine Instructions - Example

- **C Code**
  - Store value t where designated by dest

- **Assembly**
  - Move 8-byte value to memory
    - Quad words in x86-64 parlance
  - Operands:
    - t: Register %rax
    - dest: Register %rbx
    - *dest: Memory M[rbx]

- **Object Code**
  - 3-byte instruction
  - Stored at address 0x40059e
Disassembling Object Code

**Disassembler:** `objdump -d sum`

- Useful tool for examining object code
- Analyzes bit pattern of series of instructions
- Produces approximate rendition of assembly code
- Can be run on either a.out (complete executable) or .o file

Listing:

```
00000000004005a2 <sumstore>:
  4005a2:  53  push  %rbx
  4005a3:  48 89 d3  mov  %rdx,%rbx
  4005a6:  e8 f2 ff ff ff  callq 40059d <plus>
  4005ab:  48 89 03  mov  %rax, (%rbx)
  4005ae:  5b  pop  %rbx
  4005af:  c3  retq
```

```
sumstore:
  pushq  %rbx
  movq  %rdx, %rbx
  callq  plus
  movq  %rax, (%rbx)
  popq  %rbx
  ret
```
Alternate Disassembly

Within gdb Debugger

gdb sum

- disassemble sumstore
  Disassemble procedure

- x/14xb sumstore
  Examine the 14 bytes starting at sumstore

Dump of assembler code for function sumstore:
  0x00000000004005a2 <+0>: push %rbx
  0x00000000004005a3 <+1>: mov %rdx,%rbx
  0x00000000004005a6 <+4>: callq 0x40059d <plus>
  0x00000000004005ab <+9>: mov %rax,(%rbx)
  0x00000000004005ae <+12>: pop %rbx
  0x00000000004005af <+13>: retq

End of assembler dump.
What Can be Disassembled?

- Anything that can be interpreted as executable code
- Disassembler examines bytes and reconstructs assembly source

BUT:

The end user license agreement for some software forbids reverse engineering of code.
Assembly Basics: Registers, Operands, Move
**x86-64 Integer Registers**

<table>
<thead>
<tr>
<th>%rax</th>
<th>%eax</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rbx</td>
<td>%ebx</td>
</tr>
<tr>
<td>%rcx</td>
<td>%ecx</td>
</tr>
<tr>
<td>%rdx</td>
<td>%edx</td>
</tr>
<tr>
<td>%rsi</td>
<td>%esi</td>
</tr>
<tr>
<td>%rdi</td>
<td>%edi</td>
</tr>
<tr>
<td>%rsp</td>
<td>%esp</td>
</tr>
<tr>
<td>%rbp</td>
<td>%ebp</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>%r8</th>
<th>%r8d</th>
</tr>
</thead>
<tbody>
<tr>
<td>%r9</td>
<td>%r9d</td>
</tr>
<tr>
<td>%r10</td>
<td>%r10d</td>
</tr>
<tr>
<td>%r11</td>
<td>%r11d</td>
</tr>
<tr>
<td>%r12</td>
<td>%r12d</td>
</tr>
<tr>
<td>%r13</td>
<td>%r13d</td>
</tr>
<tr>
<td>%r14</td>
<td>%r14d</td>
</tr>
<tr>
<td>%r15</td>
<td>%r15d</td>
</tr>
</tbody>
</table>

Can reference low-order 4 bytes (also low-order 1 & 2 bytes)
IA32 Registers (History)

<table>
<thead>
<tr>
<th>Register</th>
<th>Origin (mostly obsolete)</th>
</tr>
</thead>
<tbody>
<tr>
<td>eax</td>
<td>accumulate</td>
</tr>
<tr>
<td>ecx</td>
<td>counter</td>
</tr>
<tr>
<td>edx</td>
<td>data</td>
</tr>
<tr>
<td>ebx</td>
<td>base</td>
</tr>
<tr>
<td>esi</td>
<td>source index</td>
</tr>
<tr>
<td>edi</td>
<td>destination index</td>
</tr>
<tr>
<td>esp</td>
<td>stack pointer</td>
</tr>
<tr>
<td>ebp</td>
<td>base pointer</td>
</tr>
</tbody>
</table>

16-bit virtual registers (backwards compatibility)
## Moving Data

- **Moving Data**

\[
\text{movq Source, Dest}
\]

## Operand Types

- **Immediate**: Constant integer data
  - Example: \$0\text{x}400, \$-533
  - Like C constant, but prefixed with ‘$’
  - Encoded with 1, 2, or 4 bytes
- **Register**: One of 16 integer registers
  - Example: \%rax, \%r13
  - But \%rsp reserved for special use
  - Others have special uses for particular instructions
- **Memory**: 8 consecutive bytes of memory at address given by register
  - Simplest example: (\%rax)
  - Various other “address modes”
### movq Operand Combinations

<table>
<thead>
<tr>
<th>Source</th>
<th>Dest</th>
<th>Src,Dest</th>
<th>C Analog</th>
</tr>
</thead>
<tbody>
<tr>
<td>Imm</td>
<td>Reg</td>
<td>movq $0x4, %rax</td>
<td>temp = 0x4;</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movq $-147, (%rax)</td>
<td>*p = -147;</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movq %rax, %rdx</td>
<td>temp2 = temp1;</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movq (%rax), %rdx</td>
<td>*p = temp;</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movq (%rax), %rdx</td>
<td>temp = *p;</td>
</tr>
</tbody>
</table>

Cannot do memory-memory transfer with a single instruction
Simple Memory Addressing Modes

- **Normal (R) Mem[Reg[R]]**
  - Register R specifies memory address

  ```
  movq (%rcx), %rax
  ```

- **Displacement D(R) Mem[Reg[R]+D]**
  - Register R specifies start of memory region
  - Constant displacement D specifies offset

  ```
  movq 8(%rbp), %rdx
  ```

Note: the normal mode is a special case of displacement mode in which D = 0
Simple Addressing Modes - \texttt{swap()} Examples

```c
void swap (long *xp, long *yp) {
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

```assembly
swap:
    movq (%rdi), %rax
    movq (%rsi), %rdx
    movq %rdx, (%rdi)
    movq %rax, (%rsi)
    ret
```

```
$ gcc -S -Og swap.c
```

\texttt{swap.c} \quad \rightarrow \quad \texttt{swap.s}
Understanding \texttt{swap()} \\

void swap (long *xp, long *yp) 
{ 
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
} \\

\begin{tabular}{|ll|}
  \hline
  Register Value & Value  \\
  %rdi & xp  \\
  %rsi & yp  \\
  %rax & t0  \\
  %rdx & t1  \\
  \hline
\end{tabular}

\begin{verbatim}
swap:
  movq   (%rdi), %rax
  movq   (%rsi), %rdx
  movq   %rdx, (%rdi)
  movq   %rax, (%rsi)
  ret
\end{verbatim}
Understanding `swap()`

**Registers**

<table>
<thead>
<tr>
<th>Register</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>0x120</td>
</tr>
<tr>
<td>%rsi</td>
<td>0x100</td>
</tr>
<tr>
<td>%rax</td>
<td></td>
</tr>
<tr>
<td>%rdx</td>
<td></td>
</tr>
</tbody>
</table>

**Memory**

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x120</td>
<td>123</td>
</tr>
<tr>
<td>0x118</td>
<td></td>
</tr>
<tr>
<td>0x110</td>
<td></td>
</tr>
<tr>
<td>0x108</td>
<td></td>
</tr>
<tr>
<td>0x100</td>
<td>456</td>
</tr>
</tbody>
</table>

`swap:`

```
    movq   (%rdi), %rax    # t0 = *xp
    movq   (%rsi), %rdx    # t1 = *yp
    movq   %rdx, (%rdi)    # *xp = t1
    movq   %rax, (%rsi)    # *yp = t0
    ret
```
Understanding `swap()`

```
swap:
    movq (%rdi), %rax    # t0 = *xp
    movq (%rsi), %rdx    # t1 = *yp
    movq %rdx, (%rdi)    # *xp = t1
    movq %rax, (%rsi)    # *yp = t0
    ret
```
Understanding `swap()`

```assembly
movq    (%rdi), %rax    # t0 = *xp
movq    (%rsi), %rdx    # t1 = *yp
movq    %rdx, (%rdi)    # *xp = t1
movq    %rax, (%rsi)    # *yp = t0
ret
```
Understanding `swap()`

```assembly
swap:
    movq (%rdi), %rax    # t0 = *xp
    movq (%rsi), %rdx    # t1 = *yp
    movq %rdx, (%rdi)    # *xp = t1
    movq %rax, (%rsi)    # *yp = t0
    ret
```
Understanding `swap()`

```assembly
swap:
    movq (%rdi), %rax    # t0 = *xp
    movq (%rsi), %rdx    # t1 = *yp
    movq %rdx, (%rdi)    # *xp = t1
    movq %rax, (%rsi)    # *yp = t0
    ret
```
Simple Memory Addressing Modes

- **Normal (R) Mem[Reg[R]]**
  - Register R specifies memory address
  - **Pointer dereferencing in C**

  \[
  \text{movq } (%rcx),%rax
  \]

- **Displacement D(R) Mem[Reg[R]+D]**
  - Register R specifies start of memory region
  - Constant displacement D specifies offset

  \[
  \text{movq } 8(%rbp),%rdx
  \]

Note: the normal mode is a special case of displacement mode in which D = 0
Complete Memory Addressing Modes

● Most General Form

\[ D(Rb, Ri, S) \quad \text{Mem}[\text{Reg}[Rb] + S \times \text{Reg}[Ri] + D] \]

- **D**: Constant “displacement” 1, 2, or 4 bytes
- **Rb**: Base register: Any of 16 integer registers
- **Ri**: Index register: Any, except for %rsp
- **S**: Scale: 1, 2, 4, or 8 (why these numbers?)

● Special Cases

- \((Rb, Ri)\) \quad \text{Mem}[\text{Reg}[Rb] + \text{Reg}[Ri]]
- \(D(Rb, Ri)\) \quad \text{Mem}[\text{Reg}[Rb] + \text{Reg}[Ri] + D]
- \((Rb, Ri, S)\) \quad \text{Mem}[\text{Reg}[Rb] + S \times \text{Reg}[Ri]]

## Address Computation Examples

<table>
<thead>
<tr>
<th>Expression</th>
<th>Address Computation</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8(%rdx)</td>
<td>0xf000 + 0x8</td>
<td>0xf008</td>
</tr>
<tr>
<td>(%rdx,%rcx)</td>
<td>0xf000 + 0x100</td>
<td>0xf100</td>
</tr>
<tr>
<td>(%rdx,%rcx,4)</td>
<td>0xf000 + 4*0x100</td>
<td>0xf400</td>
</tr>
<tr>
<td>0x80(%rdx,2)</td>
<td>2*0xf000 + 0x80</td>
<td>0x1e080</td>
</tr>
</tbody>
</table>
Logical and Arithmetic Operations
Address Computation Instruction

- **leaq Src, Dst**
  - load effective address
  - Src is address mode expression
  - Set Dst to address denoted by expression

- **Uses**
  - Computing addresses without a memory reference
    - E.g., translation of p = &x[i];
  - Computing arithmetic expressions of the form x + k*y
    - k = 1, 2, 4, or 8

- **Example**

```
file: leaq_example.c

long m12 (long x) {
    return x*12;
}
```

create object code using
```
gcc -Og -S leaq_example.c
```
```
leaq (%rdi,%rdi,2), %rax  # t = x+x*2
salq$2, %rax              # return t<<2
```
# Arithmetic Operations

- **Two Operand Instructions:**

<table>
<thead>
<tr>
<th>Format</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>addq</td>
<td>Dest = Dest + Src</td>
</tr>
<tr>
<td>subq</td>
<td>Dest = Dest - Src</td>
</tr>
<tr>
<td>imulq</td>
<td>Dest = Dest * Src</td>
</tr>
<tr>
<td>salq</td>
<td>Dest = Dest &lt;&lt; Src</td>
</tr>
<tr>
<td>sarq</td>
<td>Dest = Dest &gt;&gt; Src</td>
</tr>
<tr>
<td>shrq</td>
<td>Dest = Dest &gt;&gt; Src</td>
</tr>
<tr>
<td>xorq</td>
<td>Dest = Dest ^ Src</td>
</tr>
<tr>
<td>andq</td>
<td>Dest = Dest &amp; Src</td>
</tr>
<tr>
<td>orq</td>
<td>Dest = Dest</td>
</tr>
</tbody>
</table>

- Also called shlq
- Arithmetic
- Logical

- Watch out for argument order!
- No distinction between signed and unsigned int (why?)
Arithmetic Operations

- One Operand Instructions:

<table>
<thead>
<tr>
<th>Format</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>incq</td>
<td>Dest = Dest + 1</td>
</tr>
<tr>
<td>decq</td>
<td>Dest = Dest - 1</td>
</tr>
<tr>
<td>negq</td>
<td>Dest = ~Dest</td>
</tr>
</tbody>
</table>

- Watch out for argument order!
- No distinction between signed and unsigned int
Example: arithmetic expression

```c
long arith (long x, long y, long z)
{
    long t1 = x+y;
    long t2 = z+t1;
    long t3 = x+4;
    long t4 = y * 48;
    long t5 = t3 + t4;
    long rval = t2 * t5;
    return rval;
}
```

```assembly
arith:
    leaq (%rdi,%rsi), %rax
    addq %rdx, %rax
    leaq (%rsi,%rsi,2), %rcx
    salq $4, %rcx
    leaq4 (%rdi,%rcx), %rcx
    imulq %rcx, %rax
    ret
```
Example: arithmetic expression

```c
long arith (long x, long y, long z)
{
    long t1 = x+y;
    long t2 = z+t1;
    long t3 = x+4;
    long t4 = y * 48;
    long t5 = t3 + t4;
    long rval = t2 * t5;
    return rval;
}
```

arith:
```
leaq (%rdi,%rsi), %rax #t1
addq %rdx, %rax #t2
leaq (%rsi,%rsi,2), %rcx #t4
salq $4, %rcx #t5
leaq 4(%rdi,%rcx), %rcx
imulq %rcx, %rax #rval
ret
```

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>Argument x</td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument y</td>
</tr>
<tr>
<td>%rdx</td>
<td>Argument z</td>
</tr>
<tr>
<td>%rax</td>
<td>t1, t2, rval</td>
</tr>
<tr>
<td>%rdx</td>
<td>t4</td>
</tr>
<tr>
<td>%rcx</td>
<td>t5</td>
</tr>
</tbody>
</table>