How Software Executes

CS-576 Systems Security
Instructor: Georgios Portokalidis
Fall 2018
Overview

Introduction

Anatomy of a program

Basic assembly

Anatomy of function calls (and returns)

Memory Safety
Intel x86 Processors

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.
Intel x86 Processors

Machine Evolution

- 386  1985  0.3M
- Pentium  1993  3.1M
- Pentium/MMX  1997  4.5M
- PentiumPro  1995  6.5M
- Pentium III  1999  8.2M
- Pentium 4  2001  42M
- Core 2 Duo  2006  291M
- Core i7  2008  731M

Added Features

- Instructions to support multimedia operations
- Instructions to enable more efficient conditional operations
- Transition from 32 bits to 64 bits
- More cores
x86 Integer Registers

General purpose registers
- On 32-bit architectures
  EAX, EBX, ECX, EDX, EDI, ESI, ESP, EBP

The instruction pointer (IP)
- Also referred to as program counter (PC)
- EIP on 32-bit

FLAGS register
- Used for control flow operations, etc.
- EFLAGS
# 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>
<tr>
<td>%r8</td>
<td>%r8d</td>
</tr>
<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>

---

Stevens Institute of Technology

Fall 2018
### x86-64 Integer Registers

Can reference low-order bytes too:
- d suffix for lower 32-bits (r8d)
- w suffix for lower 16-bits (r8w)
- b suffix for lower 8-bits (r8b)

<table>
<thead>
<tr>
<th>Register</th>
<th>Low-Order Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>%r8</td>
<td>%r8d</td>
</tr>
<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>
Typical Register Uses

EAX: accumulator
EBX: Pointer to data
ECX: Counter for string operations and loops
EDX: I/O Operations
EDI: Destination for string operations
ESP: Stack pointer
EBP: Frame pointer
Assembly Syntax

Intel: OP dest, src

AT&T: OP src, dest

Unix systems prefer AT&T
  - We are going to use the same as the GNU assembler (gas syntax)
Assembly Instructions

**pushq**: push quad word to stack

**movq**: Move quad word

**imull**: Signed multiply long

**addl**: Add long

---

```
pushq  %rbp
movq   %rsp, %rbp
movl   %edi, -20(%rbp)
movl   %esi, -24(%rbp)
movl   %edx, -28(%rbp)
movl   -20(%rbp), %eax
imull  -28(%rbp), %eax
movl   %eax, %edx
movl   -24(%rbp), %eax
addl   %edx, %eax
imull  -28(%rbp), %eax
```
Operand Sizes

Instructions include a suffix that indicates the size of the operand(s)

Register operands are prefixed with a %

Register operands must match size
For example,
- quad → rax
- long → eax
- word → ax
- byte → ah or al

Intel syntax does not include a suffix, size depends on the size of the operand

pushq
%rax
Parentheses indicate a memory operand

Each memory address can be defined as:
Base + Index * Scale + Disp

- In AT&T syntax:
  disp(base, index, scale)
- disp, index, and scale are optional

```
pushq  %rbp
movq   %rsp, %rbp
movl   %edi, -20(%rbp)
movl   %esi, -24(%rbp)
movl   %edx, -28(%rbp)
movl   -20(%rbp), %eax
imull  -28(%rbp), %eax
movl   %eax, %edx
movl   -24(%rbp), %eax
addl   %edx, %eax
imull  -28(%rbp), %eax
```
Memory Addressing Modes

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

\[
\text{movq } (%r cx), \% r ax
\]

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

\[
\text{movq } 8(\% r b p), \% r dx
\]
Memory Addressing Modes

Most General Form

\[ D(B,I,S) \quad \text{Mem[Reg[R_b]+S*Reg[R_i]+ D]} \]

- **D**: Constant “displacement” 1, 2, or 4 bytes
- **Rb**: Base register
- **Ri**: Index register: Any, except for \%rsp
- **S**: Scale: 1, 2, 4, or 8

\[
\text{movq 8(\%rbp, \%rax, 4),\%rdx}
\]
Immediates

Constants or immediates are defined using $.

In decimal, unless:
- 0x prefix is used → hexadecimal
- 0 prefix is used → octal

```
addl $1, %eax
```

Immediates can help you identify the syntax
Endianness

Memory representation of multi-byte integers
For example the integer: 0x0A0B0C0Dh

Big-endian $\leftrightarrow$ highest order byte first
  0A 0B 0C 0D

Little-endian $\leftrightarrow$ lowest order byte first (X86)
  0D 0C 0B 0A
Load Effective Address

\textbf{leaq} \text{ Src, Dst}

- \text{Src} is address mode expression
- \text{Set Dst} to address denoted by expression

Computing addresses without a memory reference
- E.g., translation of $p = &x[i]$;

Computing arithmetic expressions of the form $x + k*y$
- $k = 1, 2, 4, \text{ or } 8$

Example

\texttt{leaq (%rdi,%rdi,2), %rax}
Control Flow

```plaintext
if (a > b) {
    c = d;
} else {
    d = c;
}
```

13: `cmp -0x8(%rbp),%eax`
16: `jle 0xe`
18: `mov -0x10(%rbp),%eax`
1b: `mov %eax,-0xc(%rbp)`
1e: `jmp 0x6`
20: `mov -0xc(%rbp),%eax`
23: `mov %eax,-0x10(%rbp)`
26: `mov -0xc(%rbp),%eax`
EFLAGS Register

- ID Flag (ID)
- Virtual Interrupt Pending (VIP)
- Virtual Interrupt Flag (VIF)
- Alignment Check (AC)
- Virtual-8086 Mode (VM)
- Resume Flag (RF)
- Nested Task (NT)
- I/O Privilege Level (IOPL)
- Overflow Flag (OF)
- Direction Flag (DF)
- Interrupt Enable Flag (IF)
- Trap Flag (TF)
- Sign Flag (SF)
- Zero Flag (ZF)
- Auxiliary Carry Flag (AF)
- Parity Flag (PF)
- Carry Flag (CF)

S Indicates a Status Flag
C Indicates a Control Flag
X Indicates a System Flag

Reserved bit positions. DO NOT USE.
Always set to values previously read.
Condition Codes Set by Compare

Explicit Setting by Compare Instruction

- `cmpq Src2, Src1`
- `cmpq b,a` like computing `a - b` without setting destination

- **CF set** if carry out from most significant bit (used for unsigned comparisons)
- **ZF set** if `a == b`
- **SF set** if `(a - b) < 0` (as signed)
- **OF set** if two’s-complement (signed) overflow
  
  `(a>0 && b<0 && (a-b)<0) || (a<0 && b>0 && (a-b)>0)`
Condition Codes (Explicit Setting: Test)

Explicit Setting by Test instruction

- `testq` Src2, Src1
  - `testq b,a` like computing `a&b` without setting destination

- Sets condition codes based on value of Src1 & Src2
- Useful to have one of the operands be a mask

- ZF set when `a&b == 0`
- SF set when `a&b < 0`
# Common Conditional Jumps

<table>
<thead>
<tr>
<th>jX</th>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jmp</td>
<td>1</td>
<td>Unconditional</td>
</tr>
<tr>
<td>je</td>
<td>ZF</td>
<td>Equal / Zero</td>
</tr>
<tr>
<td>jne</td>
<td>~ZF</td>
<td>Not Equal / Not Zero</td>
</tr>
<tr>
<td>jg</td>
<td>~ (SF^OF) &amp; ~ZF</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td>jge</td>
<td>~ (SF^OF)</td>
<td>Greater or Equal (Signed)</td>
</tr>
<tr>
<td>jl</td>
<td>(SF^OF)</td>
<td>Less (Signed)</td>
</tr>
<tr>
<td>jle</td>
<td>(SF^OF)</td>
<td>Less or Equal (Signed)</td>
</tr>
</tbody>
</table>
Overview

Introduction

Anatomy of a program

Basic assembly

Anatomy of function calls (and returns)

Memory Safety
The Stack Pointer (SP)

The stack pointer points to the first element in the stack.

Usually the RSP/ESP is used to store the SP.
Simple Function Call

```c
void F1()
{
    ....
    F2();
    ...
}
```
Simple Function Call

```c
void F1()
{
    ....
    F2();
    ...
}

void F2()
{
    ...
}
```
Simple Function Call

```c
void F1()
{
    ....
    F2();
    ....
}
```

```c
void F2()
{
    ...
}
```
Simple Function Call

void F1()
{
    ....
    F2();
    ...
}

void F2()
{
    ...
}

...
Function Calls

Function calls transfer control and use the stack to keep track of callees

- `call <address>` Transfer control to address and save the address of the next instruction on the stack
- `ret` Pop the address from the stack and transfer control to it

`call` and `ret` implicitly use the RSP register

- `push/pop`
Simple Function Call

```c
void F1()
{
    ....
    call F2
    ins
    ret
}
```

```c
void F2()
{
    ...
}
ret
```

- call F1
- ins
- call F2
- ins
- ret
Simple Function Call

call F1
ins

void F1()
{
    ...
    call F2
    ins
ret
}

void F2()
{
    ...
}
ret

Fall 2018
Stevens Institute of Technology
Simple Function Call

call F1
ins

void F1()
{
    ....
    call F2
    ins
    ret

} ret

void F2()
{
    ...

} ret

call F2
ins

pop val
jmp val
Local Variables

```c
void F1()
{
    int a;
    char buf[16];
    unsigned long l;
    ....
    F2();
    ...
}
```
Local Variables

```c
void F1()
{
    int a;
    char buf[16];
    unsigned long l;
    ...
    F2();
    ...
}
```
Local Variables

```c
void F1()
{
    int a;
    char buf[16];
    unsigned long l;
    ...
    F2();
    ...
}

void F2()
{
    int i;
    ...
}
```
Stack Frames

void F1()
{
    int a;
    char buf[16];
    unsigned long l;
    ...
    F2();
    ...
}

void F2()
{
    int i;
    ...
}
Frame Pointer (FP)

The frame pointer points to the bottom of the function’s frame.

Usually the RBP/EBP is used to store the FP.

void F1() {
    int a;
    char buf[16];
    unsigned long l;
    ....
    F2();
    ...
}

void F2() {
    int i;
    ...
}

24 bytes (32b)

28 bytes (64b)
When F2() returns how does the system update the frame pointer?
Saved Frame Pointer

void F1()
{
    int a;
    char buf[16];
    unsigned long l;
    ....
    F2();
    ...
}

The **callee** saves (preserves) the old value of the frame pointer.
Function Prologue/Epilogue

void F1()
{
    int a;
    char buf[16];
    unsigned long l;
    ....
    F2();
    ...
}

Function entry (ENTER)
pushq %rbp
movq %rsp, %rbp

Function exit (LEAVE)
movq %rbp, %rsp
pop %rbp
Frame pointers are optional

gcc -fomit-frame-pointer test.c
void F1()
{
    int a;
    char buf[16];
    unsigned long l;
    ....
    F2(a, buf, l);
    ...
}

void F2(int A, char *BUF unsigned long L)
{
    int i;
    ...
}
Calling Conventions

Defines the standard for passing arguments

Caller and callee need to agree

Enforced by compiler

Important when using 3rd party libraries

Different styles ↔ different advantages
cdecl (mostly 32-bit)
Arguments are passed on the stack
  ▪ Pushed right to left
eax, edx, ecx are caller saved
  ▪ callee can overwrite without saving
ebx, esi, edi are callee saved
  ▪ callee must ensure they have same value on return
eax used for function return value

System V AMD64 ABI
Arguments are passed using registers
  ▪ First 6 integer or pointer arguments are passed in registers RDI, RSI, RDX, RCX, R8, and R9
RBP, RBX, and R12–R15 are callee saved
RAX used for function return

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

```c
void F1()
{
    int a;
    char buf[16];
    unsigned long l;
    ....
    F2(a, buf, l);
    ...
}
```

```c
void F2(int A, char *BUF unsigned long L)
{
    int i;
    ...
}
```
cdecl Example

void F1()
{
    int a;
    char buf[16];
    unsigned long l;
    ....
    F2(a, buf, l);
    ...
}

void F2(int A, char *BUF unsigned long L)
{
    int i;
    ...
}
System V AMD64 ABI Example

```c
void F1()
{
    int a;
    char buf[16];
    unsigned long l;
    ...
    F2(a, buf, l);
    ...
}

void F2(int A, char *BUF unsigned long L)
{
    int i;
    ...
}
```
void F1()
{
    int a;
    char buf[16];
    unsigned long l;
    ....
    F2(a, buf, l);
    ...
}

void F2(int A, char *BUF unsigned long L)
{
    int i;
    ...
}
Variable Number of Arguments

Used in variadic functions, like

```c
int printf(const char *format, ...);
```

Arguments passed in the stack
- Order right-to-left

Only caller knows exact number of arguments
- Caller responsible for cleaning
Example on 64-bit

```c
void F1()
{
    int a;
    char buf[16];
    unsigned long l;
    ....
    printf("%d %s\n", a, buf);
    ...
}
```

```c
void printf(const char *fmt, ...)
{
    ...
}
```
Example on 32-bit

```c
void F1()
{
    int a;
    char buf[16];
    unsigned long l;
    ....
    printf("%d %s\n", a, buf);
    ...
}

void printf(const char *fmt, ...)
{
    ...
}
```

```
void printf(const char *fmt, ...)
{
    ...
}
```
Alignment

CPUs like aligned data

- Better performance

Compilers try to align data

```
0x00  0x04  0x08  0x10

int  long
```
Accessing Stack Variables

With frame pointer

```
mov  -0x18(%rbp),%eax
```

With stack pointer

```
mov  0xc(%rsp),%eax
```
Overview

Introduction

Anatomy of a program

Basic assembly

Anatomy of function calls (and returns)

Memory Safety
Variables and Memory

**C, C++**

```c
int n = 0xdeadbeef;
char str[16] = "Hello";
```

**Python**

```python
n = 0xdeadbeef
str = "Hello"
```

**Java**

```java
int n = 0xdeadbeef;
String str = "Hello";
```
Variables and Memory

C, C++

```c
int n = 0xdeadbeef;
char str[16] = “Hello”;
```

Python

```python
n = 0xdeadbeef
str = “Hello”
```

Java

```java
int n = 0xdeadbeef;
String str = “Hello”;
```
Variables and Memory

The location and actual storage characteristics are hidden by the programmer.

Python

n = 0xdeadbeef
str = "Hello"

Java

int n = 0xdeadbeef;
String str = "Hello";

Variables are stored in memory.
**Variables and Memory**

C, C++

```c
int n = 0xdeadbeef;
char str[16] = "Hello";
```

The location and storage characteristics are transparent to the programmer

Process Memory

```
variables are stored in memory
```

n
str
Variables in C and C++

```c
int n = 0xdeadbeef;

char str[16] = "Hello";
```

10 uninitialized bytes
Pointers in C and C++

```c
int n = 0xdeadbeef;
int *n_p = &n;

char str[16] = "Hello";
char *str_p = str;
```

10 uninitialized bytes
Pointers in C and C++

```c
int n = 0xdeadbeef;
int n_p = &n;
n_p++;

char str[16] = "Hello";
char *str_p = str;
str_p++;
```

```c
0xef 0xbe 0xed 0xde
'H' 'e' 'l' 'l' 'o' 0x00
```

10 uninitialized bytes
Pointers in C and C++

```c
int n = 0xdeadbeef;
int *n_p = &n;
n_p++;  // 0xef
long n_addr = (long)&n;

char str[16] = "Hello";
char *str_p = str;
str_p++;  // 'H'
long str_addr = (long)str;
```

10 uninitialized bytes
The languages support pointers and pointer arithmetic

A pointer can be easily cast to a number

10 uninitialized bytes

simple_function(year);

int simple_function(char *str)
{
    char *c;

    for (c = str; c != '\0'; c++) {
        if (*c == '0')
            *c = '1';
    }
}

simple_function(year);

int simple_function(char *str)
{
    char *c;

    for (c = str; c != '\0'; c++) {
        if (*c == '0')
            *c = '1';
    }
}
C/C++ does not perform bounds checking to ensure accesses through remain within range or are performed on the correct memory object. This task falls on the programmer.

C/C++ are memory unsafe!

```c
{  
    char *c;

    for (c = str; c != '\0'; c++) {
        if (*c == '0')
            *c = '1';
    }
}
```
Can the Hardware Help?

Modern hardware has limited support for isolated memory coarsely

Memory is organized into pages
  - Contiguous 4KB or 2MB chunks of memory

Each page can be configured as being readable or readable and writable
  - All readable pages used to be executable
  - Now they can be also marked as non-executable
    - We’ll get back to this later
Page permissions: readable (implicit, writable)

The hardware enforces these permissions

Processes can change (their) page permissions

Process
Segment Permissions

Kernel space
User code CANNOT read from nor write to these addresses, doing so results in a Segmentation Fault

Stack (grows down)

Memory Mapping Segment
File mappings (including dynamic libraries) and anonymous mappings. Example: /lib/libc.so

Heap

BSS segment
Uninitialized static variables, filled with zeros. Example: static char *userName;

Data segment
Static variables initialized by the programmer. Example: static char *gonzo = “God’s own prototype”;

Text segment (ELF)
Stores the binary image of the process (e.g., /bin/gonzo)

0x00000000 == TASK_SIZE
Random stack offset
RLIMIT_STACK (e.g., 8MB)
Random mmap offset
program break brk
start_brk
Random brk offset
end_data
start_data
end_code
0x08048000
0