CS107, Lecture 17
Assembly: Arithmetic and Logic, Continued

Reading: B&O 3.5-3.6

CS107 Topic 5

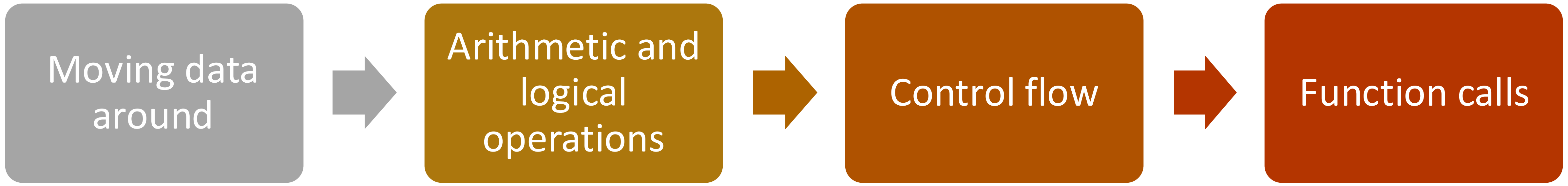
How does a computer interpret and execute C programs?

Why is answering this question important?

- Learning how our code is really translated and executed helps us write better code
- We can learn how to reverse engineer and exploit programs at the assembly level

assign5: find and exploit vulnerabilities in an ATM program, reverse engineer a program without seeing its code, and de-anonymize users given a data leak.

Learning Assembly



This Lecture

Reference Sheet: `cs107.stanford.edu/resources/x86-64-reference.pdf`
See more guides on Resources page of course website!

Helpful Assembly Resources

- **Course textbook** (reminder: see relevant readings for each lecture on the Calendar page, http://cs107.stanford.edu/calendar.html)
- **CS107 Assembly Reference Sheet:** http://cs107.stanford.edu/resources/x86-64-reference.pdf
- **CS107 Guide to x86-64**: http://cs107.stanford.edu/guide/x86-64.html

Learning Goals

- Learn how to perform arithmetic and logical operations in assembly
- Learn how to read assembly and understand the C code that generated it

Lecture Plan

- **Recap:** Assembly Instructions so far
- Arithmetic and logical operations
- Practice: Reverse Engineering

Reference Sheet: `cs107.stanford.edu/resources/x86-64-reference.pdf`
See more guides on Resources page of course website!

Lecture Plan

- **Recap: Assembly instructions so far**
- Arithmetic and logical operations
- Practice: Reverse Engineering

Reference Sheet: `cs107.stanford.edu/resources/x86-64-reference.pdf`
See more guides on Resources page of course website!

A Note About Operand Forms

- Many instructions share the address operand forms from **mov**. Eg. 7(%rax, %rcx, 2). They work the same as **mov**, except **lea:** it just does the calculation, *not the dereferencing.*
  - lea 8(%rax,%rdx),%rcx -> Calculate 8 + %rax + %rdx, put it in %rcx

Summary: parentheses means "dereference", <u>except for with **lea**.</u>

$\texttt{Imm}(r_b,\ r_i,\ s)$ is equivalent to (address) $\texttt{Imm} + R[r_b] + R[r_i]*s$

Displacement: pos/neg constant (if missing, = 0)

Base: register (if missing, = 0)

Index: register (if missing, = 0)

Scale must be 1, 2, 4, or 8 (if missing, = 1)

Data Sizes

Registers are 64 bits, but the lower [32, 16 or 8] bits of each register have separate names. E.g. %eax is the 32-bit sub-register within %rax.

Instructions may use suffixes to specify data sizes:

- `b` means **byte**
- `w` means **word** (2 bytes)
- `l` means **double word** (4 bytes)
- `q` means **quad word** (8 bytes)

One example: **movz and movs** use suffixes to specify sizes of source and dest.

Operand forms with parentheses (e.g. **mov (%rax)**) require that registers in parentheses be 64 bits. (can use e.g. **movs** to enlarge smaller register first).

lea

The **lea** instruction copies an “effective address” from one place to another.

```
lea        src,dst
```

Unlike **mov**, which copies data at the address src to the destination, **lea** copies the value of src *itself* to the destination.

Assume %rax contains 0x100, %rcx contains 0x4, 0xc5 is in memory at address 0x108.

```
mov (%rax, %rcx, 2), %rdx            0xc5 copied into %rdx
```

vs.

```
lea (%rax, %rcx, 2), %rdx            0x108 copied into %rdx
```

Unary Instructions

The following instructions operate on a single operand (register or memory):

Instruction	Effect	Description
`inc D`	`D ← D + 1`	Increment
`dec D`	`D ← D - 1`	Decrement
`neg D`	`D ← -D`	Negate
`not D`	`D ← ~D`	Complement

Examples:

```
incq 16(%rax)
dec %rdx
not %rcx
```

Binary Instructions

The following instructions operate on two operands (both can be register or memory, source can also be immediate). Both cannot be memory locations. Read it as, e.g. "Subtract S from D":

Instruction	Effect	Description
`add S, D`	`D ← D + S`	Add
`sub S, D`	`D ← D - S`	Subtract
`imul S, D`	`D ← D * S`	Multiply
`xor S, D`	`D ← D ^ S`	Exclusive-or
`or S, D`	`D ← D | S`	Or
`and S, D`	`D ← D & S`	And

Examples:

```
addq %rcx,(%rax)
xorq $16,(%rax, %rdx, 8)
subq %rdx,8(%rax)
```

Lecture Plan

- **Recap:** Assembly Instructions so far
- **Arithmetic and logical operations**
- Practice: Reverse Engineering

Reference Sheet: `cs107.stanford.edu/resources/x86-64-reference.pdf`
See more guides on Resources page of course website!

Shift Instructions

The following instructions have two operands: the shift amount **k** and the destination to shift, **D**. **k** can be either an immediate value, or the byte register **%cl** (and only that register!)

Instruction	Effect	Description
`sal k, D`	D ← D << k	Left shift
`shl k, D`	D ← D << k	Left shift (same as `sal`)
`sar k, D`	D ← D $>>_A$ k	Arithmetic right shift
`shr k, D`	D ← D $>>_L$ k	Logical right shift

Examples:

```
shll $3,(%rax)
shrl %cl,(%rax,%rdx,8)
sarl $4,8(%rax)
```

Shift Amount

Instruction	Effect	Description
`sal k, D`	$D \leftarrow D \ll k$	Left shift
`shl k, D`	$D \leftarrow D \ll k$	Left shift (same as `sal`)
`sar k, D`	$D \leftarrow D \gg_A k$	Arithmetic right shift
`shr k, D`	$D \leftarrow D \gg_L k$	Logical right shift

shift uses only the number of bits in **%cl** that make sense for what is being shifted.

- E.g. when shifting 1 byte, it looks only at the lower 3 bits (storing at most 7)
- E.g. when shifting 2 bytes, it looks only at the lower 4 bits (storing at most 15)
- When shifting **w** bits, it looks at the low-order **log2(w)** bits of **%cl** for the shift amount.
- Why is this useful? Can specify shift amount as all 1s, but it will shift by the appropriate amount.

Assembly Exploration

Let’s pull these commands together and see how some C code might be translated to assembly.

- Compiler Explorer is a handy website that lets you quickly write C code and see its assembly translation. Let’s check it out! https://godbolt.org/z/Ecbde99e3

Code Reference: calculate

```
int calculate(int x, int arr[]) {
    int sum = x;
    sum += arr[0];
    sum <<= x;
    sum &= 512;
    return sum;
}

----------

calculate:
  movl %edi, %ecx
  movl %edi, %eax
  addl (%rsi), %eax
  sall %cl, %eax
  andl $512, %eax
  ret
```

Large Multiplication

Multiplying 64-bit numbers can produce a 128-bit result. How does x86-64 support this with only 64-bit registers?

- If you specify two operands to **imul**, it multiplies them together and truncates until it fits in a 64-bit register.

```
imul S, D        D ← D * S
```

- If you specify one operand, it multiplies that by **%rax** and splits the product across **2** registers. It puts the high-order 64 bits in **%rdx** and the low-order 64 bits in **%rax**.

Instruction	Effect	Description
`imulq S`	`R[%rdx]:R[%rax] ← S x R[%rax]`	Signed full multiply
`mulq S`	`R[%rdx]:R[%rax] ← S x R[%rax]`	Unsigned full multiply

Division and Remainder

Instruction	Effect	Description

`idivq S`	`R[%rdx] ← R[%rdx]:R[%rax] mod S;`
	`R[%rax] ← R[%rdx]:R[%rax] ÷ S`
		Signed divide

`divq S`	`R[%rdx] ← R[%rdx]:R[%rax] mod S;`
	`R[%rax] ← R[%rdx]:R[%rax] ÷ S`
		Unsigned divide

- **x86-64** supports dividing up to a 128-bit value by a 64-bit value.
- Terminology: **dividend / divisor = quotient with remainder**
- The high-order 64 bits of the dividend are in **%rdx**, and the low-order 64 bits are in **%rax**. The divisor is the operand to the instruction.
- The quotient is stored in **%rax**, and the remainder in **%rdx**.

Division and Remainder

Instruction	Effect	Description

`idivq S`	`R[%rdx] ← R[%rdx]:R[%rax] mod S;`
	`R[%rax] ← R[%rdx]:R[%rax] ÷ S`
		Signed divide

`divq S`	`R[%rdx] ← R[%rdx]:R[%rax] mod S;`
	`R[%rax] ← R[%rdx]:R[%rax] ÷ S`
		Unsigned divide

`cqto`	`R[%rdx]:R[%rax] ← SignExtend(R[%rax])`	Convert to oct word

- Terminology: **dividend / divisor = quotient with remainder**
- The high-order 64 bits of the dividend are in **%rdx**, and the low-order 64 bits are in **%rax**. The divisor is the operand to the instruction.
- Most division uses only 64-bit dividends. The **cqto** instruction sign-extends the 64-bit value in **%rax** into **%rdx** to fill both registers with the dividend, as the division instruction expects.

Compiler Explorer Demo

https://godbolt.org/z/4cT75M4nd

Code Reference: full_divide

```
// Returns x/y, stores remainder in location stored in remainder_ptr
long full_divide(long x, long y, long *remainder_ptr) {
    long quotient = x / y;
    long remainder = x % y;
    *remainder_ptr = remainder;
    return quotient;
}

-------

full_divide:
  movq %rdi, %rax
  movq %rdx, %rcx
  cqto
  idivq %rsi
  movq %rdx, (%rcx)
  ret
```

Lecture Plan

- **Recap:** Assembly Instructions so far
- Arithmetic and logical operations
- **Practice: Reverse Engineering**

Reference Sheet: `cs107.stanford.edu/resources/x86-64-reference.pdf`
See more guides on Resources page of course website!

Assembly Exercise

```
0000000000401172 <sum_example2>:
      401172: 8b 47 0c        mov   0xc(%rdi),%eax
      401175: 03 07           add   (%rdi),%eax
      401177: 2b 47 18        sub   0x18(%rdi),%eax
      40117a: c3              retq
```

Respond with your thoughts on PollEv:
pollev.com/cs107

```
int sum_example2(int arr[]) {
    int sum = 0;
    sum += arr[0];
    sum += arr[3];
    sum -= arr[6];
    return sum;
}
```

What location or value in the assembly code above represents the C code's **6** (as in **arr[6]**)?

What location or value in the assembly code above represents the C code's 6 (as in arr[6])?

0xc
0%
0x18
0%
%rdi
0%
%eax
0%

Assembly Exercise

```
0000000000401172 <sum_example2>:
      401172: 8b 47 0c         mov   0xc(%rdi),%eax
      401175: 03 07            add   (%rdi),%eax
      401177: 2b 47 18         sub   0x18(%rdi),%eax
      40117a: c3               retq
```

```
int sum_example2(int arr[]) {
    int sum = 0;
    sum += arr[0];
    sum += arr[3];
    sum -= arr[6];
    return sum;
}
```

What location or value in the assembly code above represents the C code's **6** (as in **arr[6]**)?

0x18

Recap

- **Recap:** Assembly instructions so far
- Arithmetic and logical operations
- Practice: Reverse Engineering

Lecture 17 takeaway: There are assembly instructions for arithmetic and logical operations. They share the same operand form as mov, but lea interprets them differently.

Next Time: control flow in assembly (while loops, if statements, and more)

Extra Practice

Assembly Exercise 1

```
000000000040116e <sum_example1>:
  40116e: 8d 04 37            lea   (%rdi,%rsi,1),%eax
  401171: c3                  retq
```

Which of the following is most likely to have generated the above assembly?

```
// A)
void sum_example1() {
    int x;
    int y;
    int sum = x + y;
}
```

```
// [✓ B)]
int sum_example1(int x, int y) {
    return x + y;
}
```

```
// C)
void sum_example1(int x, int y) {
    int sum = x + y;
}
```

Assembly Exercise 2

```
0000000000401172 <sum_example2>:
      401172: 8b 47 0c        mov   0xc(%rdi),%eax
      401175: 03 07           add   (%rdi),%eax
      401177: 2b 47 18        sub   0x18(%rdi),%eax
      40117a: c3              retq
```

```
int sum_example2(int arr[]) {
    int sum = 0;
    sum += arr[0];
    sum += arr[3];
    sum -= arr[6];
    return sum;
}
```

What location or value in the assembly above represents the C code's **sum** variable?

%eax

Reverse Engineering 1

```
int add_to(int x, int arr[], int i) {
    int sum = ___?___;
    sum += arr[___?___];
    return ___?___;
}

----------
// x in %edi, arr in %rsi, i in %edx
add_to:
  movslq %edx, %rdx
  movl %edi, %eax
  addl (%rsi,%rdx,4), %eax
  ret
```

Reverse Engineering 1

```
int add_to(int x, int arr[], int i) {
    int sum = ___?___;
    sum += arr[___?___];
    return ___?___;
}

----------
// x in %edi, arr in %rsi, i in %edx
add_to:
  movslq %edx, %rdx              // sign-extend i into full register
  movl %edi, %eax                // copy x into %eax
  addl (%rsi,%rdx,4), %eax       // add arr[i] to %eax
  ret
```

Reverse Engineering 1

```
int add_to(int x, int arr[], int i) {
    int sum = x;
    sum += arr[i];
    return sum;
}

----------
// x in %edi, arr in %rsi, i in %edx
add_to:
  movslq %edx, %rdx               // sign-extend i into full register
  movl %edi, %eax                 // copy x into %eax
  addl (%rsi,%rdx,4), %eax        // add arr[i] to %eax
  ret
```

Reverse Engineering 2

```
int elem_arithmetic(int nums[], int y) {
    int z = nums[___?___] * ___?___;
    z -= ___?___;
    z >>= ___?___;
    return ___?___;
}
----------
// nums in %rdi, y in %esi
elem_arithmetic:
  movl %esi, %eax
  imull (%rdi), %eax
  subl 4(%rdi), %eax
  sarl $2, %eax
  addl $2, %eax
  ret
```

Reverse Engineering 2

```
int elem_arithmetic(int nums[], int y) {
    int z = nums[___?___] * ___?___;
    z -= ___?___;
    z >>= ___?___;
    return ___?___;
}
----------
// nums in %rdi, y in %esi
elem_arithmetic:
  movl %esi, %eax           // copy y into %eax
  imull (%rdi), %eax        // multiply %eax by nums[0]
  subl 4(%rdi), %eax        // subtract nums[1] from %eax
  sarl $2, %eax             // shift %eax right by 2
  addl $2, %eax             // add 2 to %eax
  ret
```

Reverse Engineering 2

```
int elem_arithmetic(int nums[], int y) {
    int z = nums[0] * y;
    z -= nums[1];
    z >>= 2;
    return z + 2;
}
----------
// nums in %rdi, y in %esi
elem_arithmetic:
  movl %esi, %eax           // copy y into %eax
  imull (%rdi), %eax        // multiply %eax by nums[0]
  subl 4(%rdi), %eax        // subtract nums[1] from %eax
  sarl $2, %eax             // shift %eax right by 2
  addl $2, %eax             // add 2 to %eax
  ret
```

Reverse Engineering 3

```
long func(long x, long *ptr) {
    *ptr = ___?___ + 1;
    long result = x % ___?___;
    return ___?___;
}
----------
// x in %rdi, ptr in %rsi
func:
  movq %rdi, %rax
  leaq 1(%rdi), %rcx
  movq %rcx, (%rsi)
  cqto
  idivq %rcx
  movq %rdx, %rax
  ret
```

https://godbolt.org/z/hGKPWszq4

Reverse Engineering 3

```
long func(long x, long *ptr) {
    *ptr = ___?___ + 1;
    long result = x % ___?___;
    return ___?___;
}
----------
// x in %rdi, ptr in %rsi
func:
  movq %rdi, %rax          // copy x into %rax
  leaq 1(%rdi), %rcx       // put x + 1 into %rcx
  movq %rcx, (%rsi)        // copy %rcx into *ptr
  cqto                     // sign-extend x into %rdx
  idivq %rcx               // calculate x / (x + 1)
  movq %rdx, %rax          // copy the remainder into %rax
  ret
```

Reverse Engineering 3

```
long func(long x, long *ptr) {
    *ptr = x + 1;
    long result = x % *ptr; // or x + 1
    return result;
}
----------
// x in %rdi, ptr in %rsi
func:
  movq %rdi, %rax            // copy x into %rax
  leaq 1(%rdi), %rcx         // put x + 1 into %rcx
  movq %rcx, (%rsi)          // copy %rcx into *ptr
  cqto                       // sign-extend x into %rdx
  idivq %rcx                 // calculate x / (x + 1)
  movq %rdx, %rax            // copy the remainder into %rax
  ret
```

Side Note: Old GCC Output

```
long func(long x, long *ptr) {
    *ptr = x + 1;
    long result = x % *ptr; // or x + 1
    return result;
}
----------
// x in %rdi, ptr in %rsi
func:
  leaq 1(%rdi), %rcx       // put x + 1 into %rcx
  movq %rcx, (%rsi)        // copy %rcx into *ptr
  movq %rdi, %rax          // copy x into %rax
  cqto                     // sign-extend x into %rdx
  idivq %rcx               // calculate x / (x + 1)
  movq %rdx, %rax          // copy the remainder into %rax
  ret
```