CS107, Lecture 11
Assembly Continued + Privacy/Trust

Reading: B&O 3.1-3.4

Lecture Plan

- **Recap: mov so far**
- Data and Register Sizes
- The **lea** Instruction
- Logical and Arithmetic Operations
- Practice: Reverse Engineering
- Privacy and Trust

Reference Sheet:
`https://web.stanford.edu/class/archive/cs/cs107/cs107.1248/guide/x86-64.html`
See more guides on Resources page of course website!

Central Processing Units (CPUs)

Intel 8086, 16-bit microprocessor ($86.65, 1978)

Raspberry Pi BCM2836 32-bit **ARM** microprocessor ($35 for everything, 2015)

Intel Core i9-9900K 64-bit 8-core multi-core processor ($449, 2018)

mov

The **`mov`** instruction copies bytes from one place to another;
it is similar to the assignment operator (=) in C.

```
mov     src,dst
```

The **src** and **dst** can each be one of:

- Immediate (constant value, like a number) (*only **src**)*
- Register
- Memory Location
  (*at most one of **src, dst**)*

Memory Location Syntax

Syntax	Meaning
0x104	What's at addr: 0x104 (no $)
(%rax)	What's at the addr in %rax
4(%rax)	What's at the addr: (%rax + 4)
(%rax, %rdx)	What's at the addr: (%rax + %rdx)
4(%rax, %rdx)	What's at the addr: (%rax + %rdx + 4)
(, %rcx, 4)	What's at the addr: (%rcx * 4) (multiplier can be 1, 2, 4, 8)
(%rax, %rcx, 2)	What's at the addr: (%rax + 2 * %rcx)
8(%rax, %rcx, 2)	What's at the addr: (%rax + 2*%rcx + 8)

Operand Forms

Type	Form	Operand Value	Name
Immediate	$\$Imm$	Imm	Immediate
Register	$r_!$	$\mathrm{R}[r_!]$	Register
Memory	Imm	$\mathrm{M}[Imm]$	Absolute
Memory	$(r_!)$	$\mathrm{M}[\mathrm{R}[r_!]]$	Indirect
Memory	$Imm(r_{"})$	$\mathrm{M}[Imm + \mathrm{R}[r_{"}]]$	Base + displacement
Memory	$(r_{"}, r_{\#})$	$\mathrm{M}[\mathrm{R}[r_{"}] + \mathrm{R}[r_{\#}]]$	Indexed
Memory	$Imm(r_{"}, r_{\#})$	$\mathrm{M}[Imm + \mathrm{R}[r_{"}] + \mathrm{R}[r_{\#}]]$	Indexed
Memory	$(, r_{\#}, s)$	$\mathrm{M}[\mathrm{R}[r_{\#}] \ . \ s]$	Scaled indexed
Memory	$Imm(, r_{\#}, s)$	$\mathrm{M}[Imm + \mathrm{R}[r_{\#}] \ . \ s]$	Scaled indexed
Memory	$(r_{"}, r_{\#}, s)$	$\mathrm{M}[\mathrm{R}[r_{"}] + \mathrm{R}[r_{\#}] \ . \ s]$	Scaled indexed
Memory	$Imm(r_{"}, r_{\#}, s)$	$\mathrm{M}[Imm + \mathrm{R}[r_{"}] + \mathrm{R}[r_{\#}] \ . \ s]$	Scaled indexed

Figure 3.3 from the book: "Operand forms. Operands can denote immediate (constant) values, register values, or values from memory. The scaling factor *s* must be either, 1, 2, 4, or 8."

Lecture Plan

- **Recap: mov** so far
- **Data and Register Sizes**
- The **lea** Instruction
- Logical and Arithmetic Operations
- Practice: Reverse Engineering
- Privacy and Trust

Reference Sheet:
`https://web.stanford.edu/class/archive/cs/cs107/cs107.1248/guide/x86-64.html`
See more guides on Resources page of course website!

Data Sizes

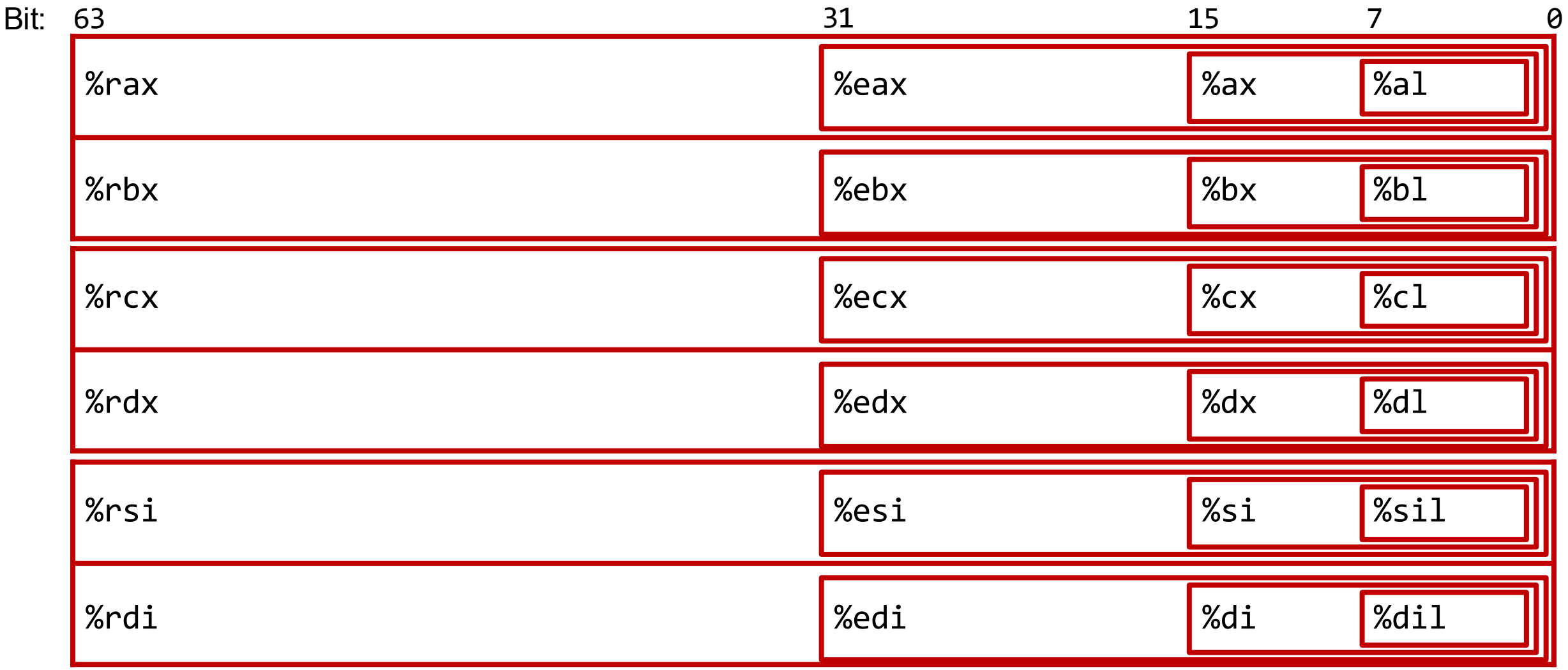
Data sizes in assembly have slightly different terminology to get used to:

- A **byte** is 1 byte.
- A **word** is 2 bytes.
- A **double word** is 4 bytes.
- A **quad word** is 8 bytes.

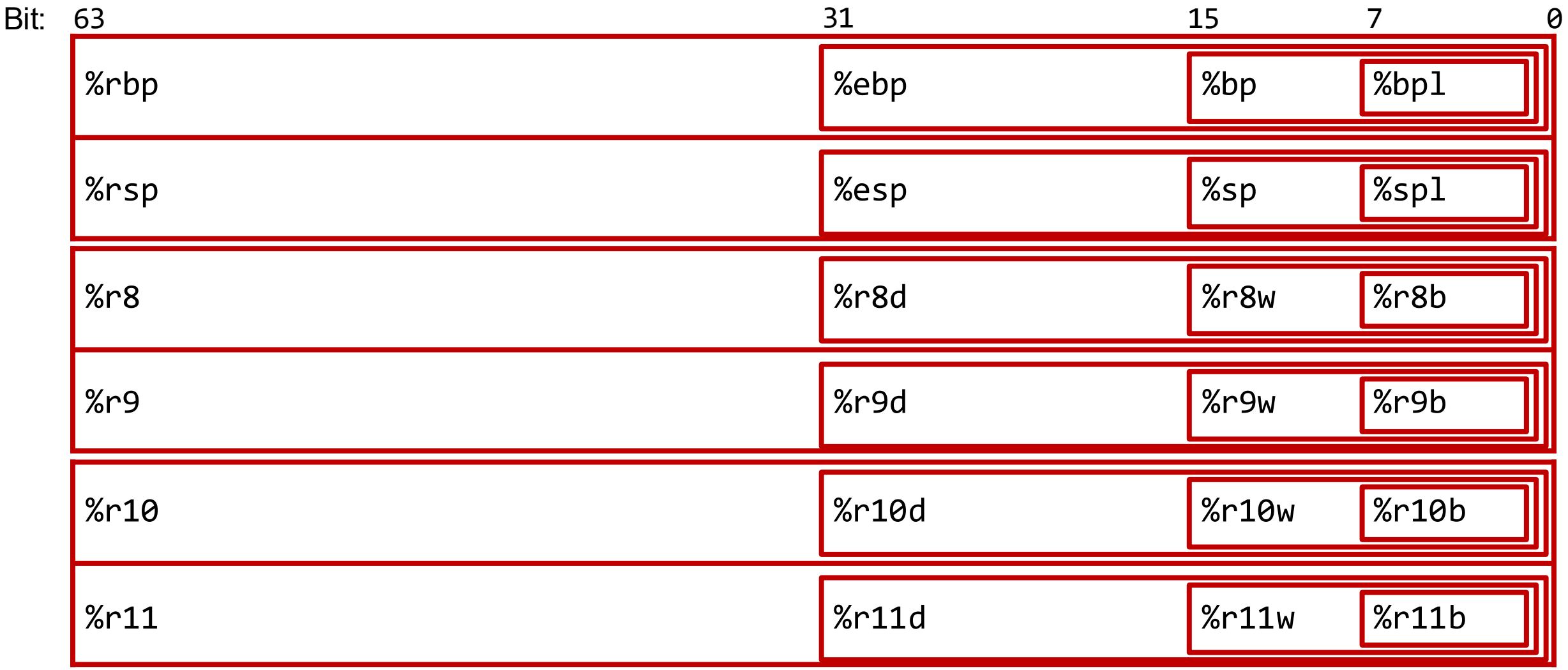
Assembly instructions can have suffixes to refer to these sizes:

- `b` means **byte**
- `w` means **word**
- `l` means **double word**
- `q` means **quad word**

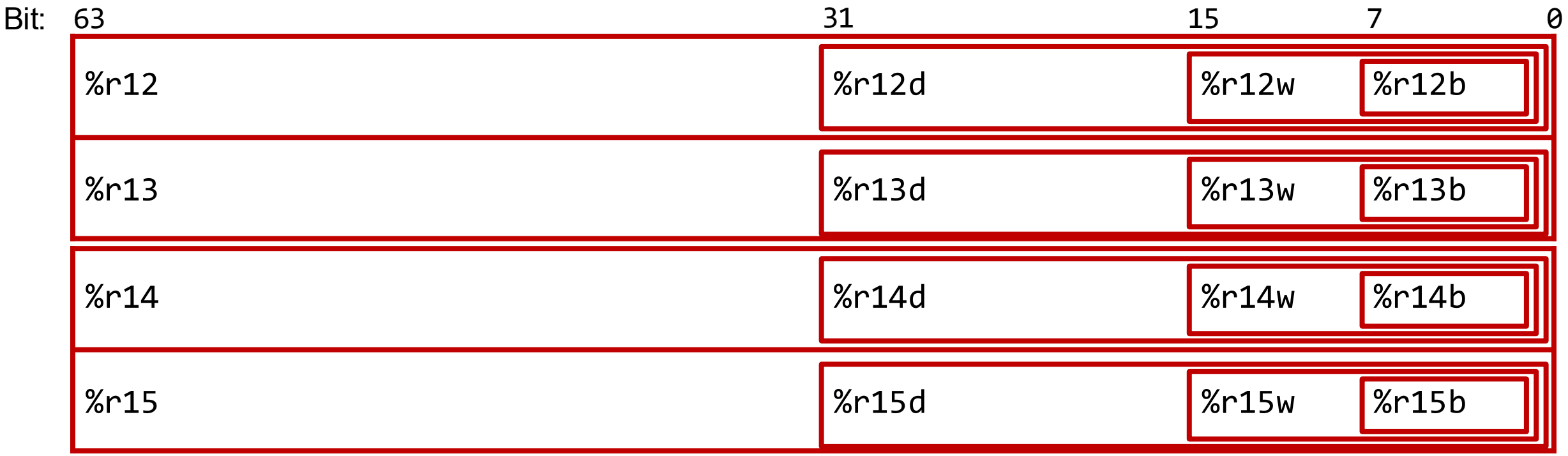
Register Sizes



Register Sizes



Register Sizes



Register Responsibilities

Some registers take on special responsibilities during program execution.

- **%rax** stores the return value
- **%rdi** stores the first parameter to a function
- **%rsi** stores the second parameter to a function
- **%rdx** stores the third parameter to a function
- **%rip** stores the address of the next instruction to execute
- **%rsp** stores the address of the current top of the stack

Reference Sheet:
`https://web.stanford.edu/class/archive/cs/cs107/cs107.1248/guide/x86-64.html`
See more guides on Resources page of course website!

mov Variants

- **mov** can take an optional suffix (b,w,l,q) that specifies the size of data to move: `movb, movw, movl, movq`
- **mov** only updates the specific register bytes or memory locations indicated.
  - **Exception: movl** writing to a register will also set high order 4 bytes to 0.

Practice: mov And Data Sizes

For each of the following mov instructions, determine the appropriate suffix based on the operands (e.g. **movb**, **movw**, **movl** or **movq**).

```
1. mov__ %eax, (%rsp)
2. mov__ (%rax), %dx
3. mov__ $0xff, %bl
4. mov__ (%rsp,%rdx,4),%dl
5. mov__ (%rdx), %rax
6. mov__ %dx, (%rax)
```

Practice: mov And Data Sizes

For each of the following mov instructions, determine the appropriate suffix based on the operands (e.g. **movb**, **movw**, **movl** or **movq**).

```
1. movl %eax, (%rsp)
2. movw (%rax), %dx
3. movb $0xff, %bl
4. movb (%rsp,%rdx,4),%dl
5. movq (%rdx), %rax
6. movw %dx, (%rax)
```

mov

- The **movabsq** instruction is used to write a 64-bit Immediate (constant) value.
- The regular **movq** instruction can only take 32-bit immediates.
- 64-bit immediate as source, only register as destination.

movabsq $0x0011223344556677, %rax

movz and movs

- There are two mov instructions that can be used to copy a smaller source to a larger destination: **movz** and **movs**.
- **movz** fills the remaining bytes with zeros
- **movs** fills the remaining bytes by sign-extending the most significant bit in the source.
- The source must be from memory or a register, and the destination is a register.

movz and movs

```
MOVZ S,R        R ← ZeroExtend(S)
```

Instruction	Description
`movzbw`	Move zero-extended byte to word
`movzbl`	Move zero-extended byte to double word
`movzwl`	Move zero-extended word to double word
`movzbq`	Move zero-extended byte to quad word
`movzwq`	Move zero-extended word to quad word

movz and movs

MOVS S,R R ← SignExtend(S)

Instruction	Description
movsbw	Move sign-extended byte to word
movsbl	Move sign-extended byte to double word
movswl	Move sign-extended word to double word
movsbq	Move sign-extended byte to quad word
movswq	Move sign-extended word to quad word
movslq	Move sign-extended double word to quad word
cltq	Sign-extend %eax to %rax %rax <- SignExtend(%eax)

Lecture Plan

- **Recap: mov** so far
- Data and Register Sizes
- **The lea Instruction**
- Logical and Arithmetic Operations
- Practice: Reverse Engineering
- Privacy and Trust

Reference Sheet:
`https://web.stanford.edu/class/archive/cs/cs107/cs107.1248/guide/x86-64.html`
See more guides on Resources page of course website!

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.

lea requires a memory-style expression for the src, and a register for dst.

The syntax for the destinations is the same as **mov**. The difference is how it handles the src.

lea vs. mov

Operands	mov Interpretation	lea Interpretation
`6(%rax), %rdx`	Go to the address (6 + what's in %rax), and copy data there into %rdx	Copy 6 + what's in %rax into %rdx.

lea vs. mov

Operands	mov Interpretation	lea Interpretation
`6(%rax), %rdx`	Go to the address (6 + what's in %rax), and copy data there into %rdx	Copy 6 + what's in %rax into %rdx.
`(%rax, %rcx), %rdx`	Go to the address (what's in %rax + what's in %rcx) and copy data there into %rdx	Copy (what's in %rax + what's in %rcx) into %rdx.

lea vs. mov

Operands	mov Interpretation	lea Interpretation
`6(%rax), %rdx`	Go to the address (6 + what's in %rax), and copy data there into %rdx	Copy 6 + what's in %rax into %rdx.
`(%rax, %rcx), %rdx`	Go to the address (what's in %rax + what's in %rcx) and copy data there into %rdx	Copy (what's in %rax + what's in %rcx) into %rdx.
`(%rax, %rcx, 4), %rdx`	Go to the address (%rax + 4 * %rcx) and copy data there into %rdx.	Copy (%rax + 4 * %rcx) into %rdx.

lea vs. mov

Operands	mov Interpretation	lea Interpretation
`6(%rax), %rdx`	Go to the address (6 + what's in %rax), and copy data there into %rdx	Copy 6 + what's in %rax into %rdx.
`(%rax, %rcx), %rdx`	Go to the address (what's in %rax + what's in %rcx) and copy data there into %rdx	Copy (what's in %rax + what's in %rcx) into %rdx.
`(%rax, %rcx, 4), %rdx`	Go to the address (%rax + 4 * %rcx) and copy data there into %rdx.	Copy (%rax + 4 * %rcx) into %rdx.
`7(%rax, %rax, 8), %rdx`	Go to the address (7 + %rax + 8 * %rax) and copy data there into %rdx.	Copy (7 + %rax + 8 * %rax) into %rdx.

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

Lecture Plan

- **Recap: mov** so far
- Data and Register Sizes
- The **lea** Instruction
- **Logical and Arithmetic Operations**
- Practice: Reverse Engineering
- Privacy and Trust

Reference Sheet:
`https://web.stanford.edu/class/archive/cs/cs107/cs107.1248/guide/x86-64.html`
See more guides on Resources page of course website!

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)
```

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 (discards upper bits).

```
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

- Terminology: **dividend / divisor = quotient + remainder**
- **x86-64** supports dividing up to a 128-bit value by a 64-bit value.
- 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 + 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.

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 ← 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

- When using **%cl**, the width of what you are shifting determines what portion of **%cl** is used.
- For **x** bits of data, it looks at the low-order **log2(x)** bits of **%cl** to know how much to shift.
  - If **%cl** = 0xff, then: **shlb** shifts by 7 because it considers only the low-order log2(8) = 3 bits, which represent 7. **shlw** shifts by 15 because it considers only the low-order log2(16) = 4 bits, which represent 15.

Lecture Plan

- **Recap: mov** so far
- Data and Register Sizes
- The **lea** Instruction
- Logical and Arithmetic Operations
- **Practice: Reverse Engineering**
- Privacy and Trust

Reference Sheet:
`https://web.stanford.edu/class/archive/cs/cs107/cs107.1248/guide/x86-64.html`
See more guides on Resources page of course website!

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/WPzz6G4a9

Code Reference: add_to_first

```
// Returns the sum of x and the first element in
arr
int add_to_first(int x, int arr[]) {
    int sum = x;
    sum += arr[0];
    return sum;
}

----------

add_to_first:
  movl %edi, %eax      // x is in %eax
  addl (%rsi), %eax    // add (4 bytes) arr[0] to %eax
  ret
```

Argument order reminder:
%rdi, %rsi, %rdx, %rcx, %r8, %r9

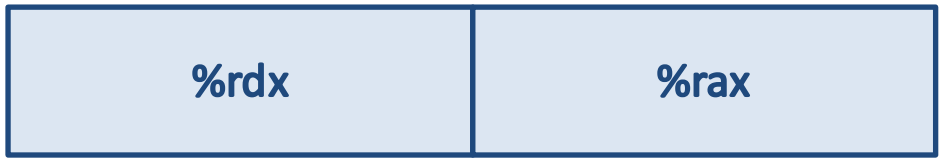
Code Reference: full_divide

```
// 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    // x is in %rax
  movq   %rdx, %rcx    // save remainder_ptr in %rcx
  cqto                 // sign extend %rax to %rdx:%rax
  idivq   %rsi         // %rdx:%rax ÷ %rsi
  movq   %rdx, (%rcx)  // *remainder_ptr = remainder
  ret
```

Argument order reminder:
%rdi, %rsi, %rdx, %rcx, %r8, %r9

Division "128-bit register" reminder:



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 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;
}
```

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

```
// B)
int sum_example1(int x, int y) {
    return 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?

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

Assembly Exercise 3

```
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]**)?

Assembly Exercise 3

```
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

Our First Assembly

```
int sum_array(int arr[], int nelems) {
    int sum = 0;
    for (int i = 0; i < nelems; i++) {
        sum += arr[i];
    }
    return sum;
}
```

We're 1/2 of the way to understanding assembly!
What looks understandable right now?

```
0000000000401136 <sum_array>:
  401136:    b8 00 00 00 00      mov    $0x0,%eax
  40113b:    ba 00 00 00 00      mov    $0x0,%edx
  401140:    39 f0               cmp    %esi,%eax
  401142:    7d 0b               jge    40114f <sum_array+0x19>
  401144:    48 63 c8            movslq %eax,%rcx
  401147:    03 14 8f            add    (%rdi,%rcx,4),%edx
  40114a:    83 c0 01            add    $0x1,%eax
  40114d:    eb f1               jmp    401140 <sum_array+0xa>
  40114f:    89 d0               mov    %edx,%eax
  401151:    c3                  retq
```

A Note About Operand Forms

- Many instructions share the same address operand forms that **mov** uses.
  - Eg. 7(%rax, %rcx, 2).
- These forms work the same way for other instructions, e.g. sub:
  - sub 8(%rax,%rdx),%rcx -> Go to 8 + %rax + %rdx, subtract what's there from %rcx
- The exception is **lea:**
  - It interprets this form as just the calculation, *not the dereferencing*
  - lea 8(%rax,%rdx),%rcx -> Calculate 8 + %rax + %rdx, put it in %rcx

Extra Practice

https://godbolt.org/z/hGKPWszq4

Executing Instructions

So far:

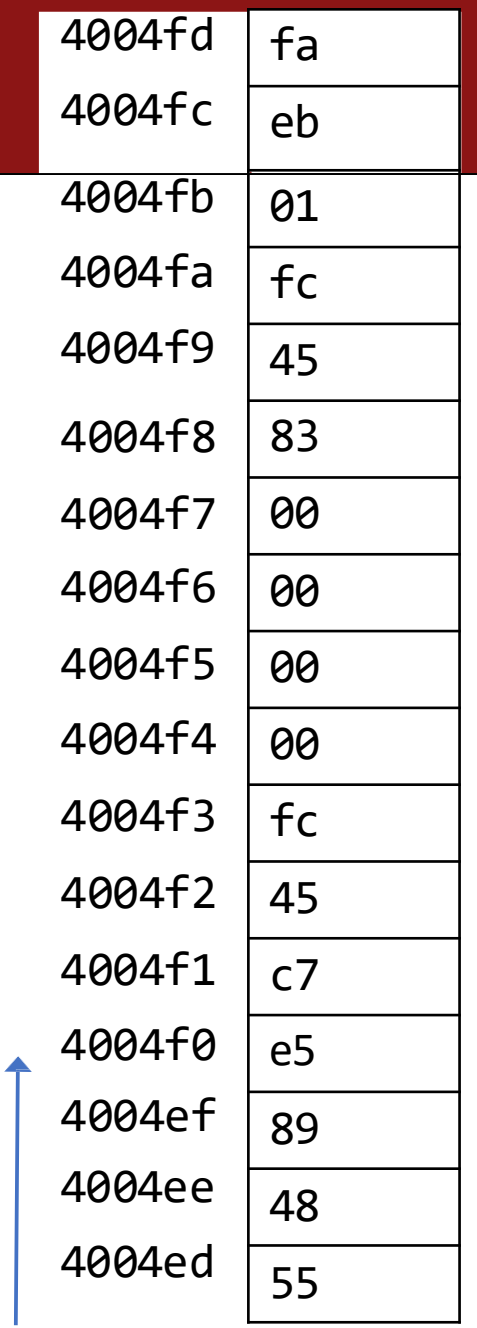
- Program values can be stored in memory or registers.
- Assembly instructions read/write values back and forth between registers (on the CPU) and memory.
- Assembly instructions are also stored in memory.

Now:

- **Who controls the instructions**?
  How do we know what to do now or next?

Answer:

- The **program counter** (PC), %rip.



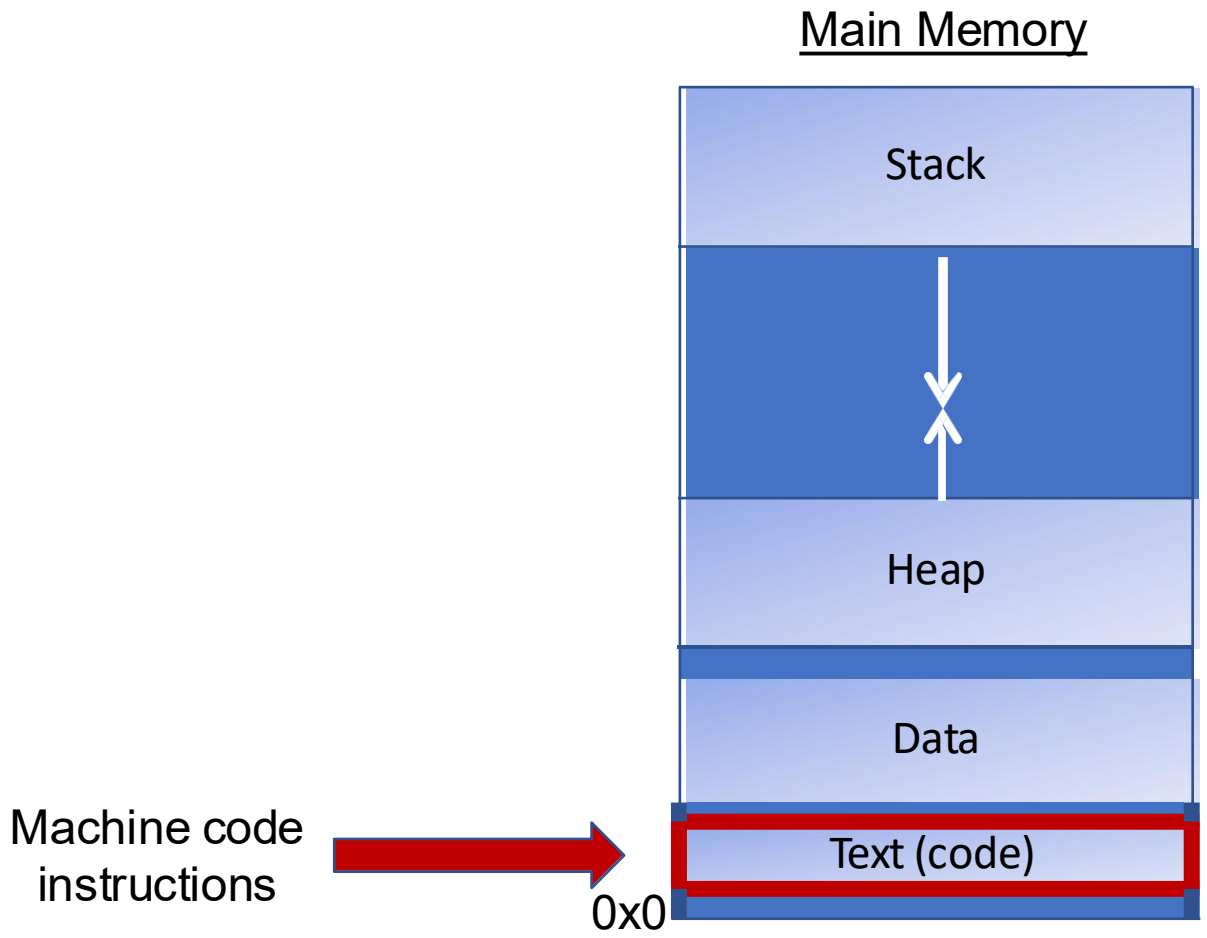
Register Responsibilities

Some registers take on special responsibilities during program execution.

- **%rax** stores the return value
- **%rdi** stores the first parameter to a function
- **%rsi** stores the second parameter to a function
- **%rdx** stores the third parameter to a function
- **%rip** stores the address of the next instruction to execute
- **%rsp** stores the address of the current top of the stack

See the x86-64 Guide and Reference Sheet on the Resources webpage for more!

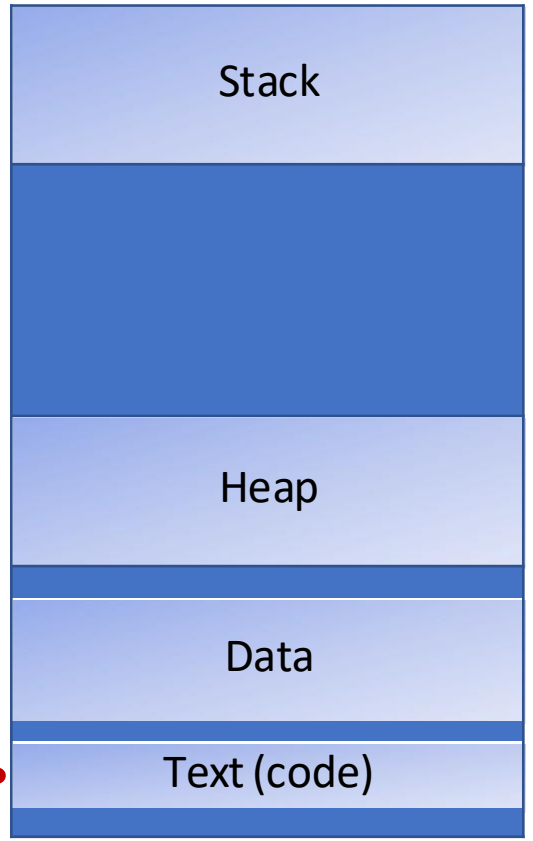
Instructions Are Just Bytes!



%rip

```
00000000004004ed <loop>:
4004ed: 55                              push   %rbp
4004ee: 48 89 e5                        mov    %rsp,%rbp
4004f1: c7 45 fc 00 00 00 00            movl   $0x0,-0x4(%rbp)
4004f8: 83 45 fc 01                     addl   $0x1,-0x4(%rbp)
4004fc: eb fa                           jmp    4004f8 <loop+0xb>
```

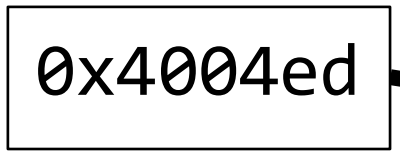
	fa
4004fc	eb
4004fb	01
4004fa	fc
4004f9	45
4004f8	83
4004f7	00
4004f6	00
4004f5	00
4004f4	00
4004f3	fc
4004f2	45
4004f1	c7
4004f0	e5
4004ef	89
4004ee	48
4004ed	55



%rip

```
0000000004004ed <loop>:
4004ed: 55                      push   %rbp
4004ee: 48 89 e5                mov    %rsp,%rbp
4004f1: c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%rbp)
4004f8: 83 45 fc 01             addl   $0x1,-0x4(%rbp)
4004fc: eb fa                   jmp    4004f8 <loop+0xb>
```

The **program counter** (PC), known as %rip in x86-64, stores the address in memory of the ***next instruction*** to be executed.

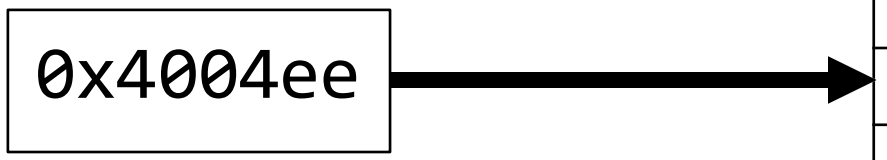


4004fd	fa
4004fc	eb
4004fb	01
4004fa	fc
4004f9	45
4004f8	83
4004f7	00
4004f6	00
4004f5	00
4004f4	00
4004f3	fc
4004f2	45
4004f1	c7
4004f0	e5
4004ef	89
4004ee	48
4004ed	55

%rip

```
0000000004004ed <loop>:
4004ed: 55                              push   %rbp
4004ee: 48 89 e5                        mov    %rsp,%rbp
4004f1: c7 45 fc 00 00 00 00            movl   $0x0,-0x4(%rbp)
4004f8: 83 45 fc 01                     addl   $0x1,-0x4(%rbp)
4004fc: eb fa                           jmp    4004f8 <loop+0xb>
```

The **program counter** (PC), known as %rip in x86-64, stores the address in memory of the ***next instruction*** to be executed.

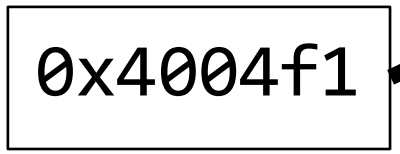


4004fd	fa
4004fc	eb
4004fb	01
4004fa	fc
4004f9	45
4004f8	83
4004f7	00
4004f6	00
4004f5	00
4004f4	00
4004f3	fc
4004f2	45
4004f1	c7
4004f0	e5
4004ef	89
4004ee	48
4004ed	55

%rip

```
0000000004004ed <loop>:
4004ed: 55                              push   %rbp
4004ee: 48 89 e5                        mov    %rsp,%rbp
4004f1: c7 45 fc 00 00 00 00            movl   $0x0,-0x4(%rbp)
4004f8: 83 45 fc 01                     addl   $0x1,-0x4(%rbp)
4004fc: eb fa                           jmp    4004f8 <loop+0xb>
```

The **program counter** (PC), known as %rip in x86-64, stores the address in memory of the ***next instruction*** to be executed.

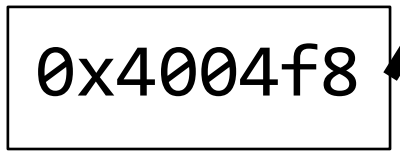


4004fd	fa
4004fc	eb
4004fb	01
4004fa	fc
4004f9	45
4004f8	83
4004f7	00
4004f6	00
4004f5	00
4004f4	00
4004f3	fc
4004f2	45
4004f1	c7
4004f0	e5
4004ef	89
4004ee	48
4004ed	55

%rip

```
0000000004004ed <loop>:
4004ed: 55                              push   %rbp
4004ee: 48 89 e5                        mov    %rsp,%rbp
4004f1: c7 45 fc 00 00 00 00            movl   $0x0,-0x4(%rbp)
4004f8: 83 45 fc 01                     addl   $0x1,-0x4(%rbp)
4004fc: eb fa                           jmp    4004f8 <loop+0xb>
```

The **program counter** (PC), known as %rip in x86-64, stores the address in memory of the ***next instruction*** to be executed.

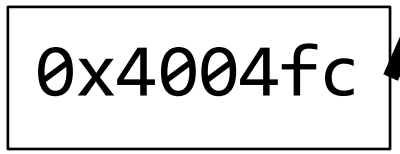


4004fd	fa
4004fc	eb
4004fb	01
4004fa	fc
4004f9	45
4004f8	83
4004f7	00
4004f6	00
4004f5	00
4004f4	00
4004f3	fc
4004f2	45
4004f1	c7
4004f0	e5
4004ef	89
4004ee	48
4004ed	55

%rip

```
0000000004004ed <loop>:
4004ed: 55                              push   %rbp
4004ee: 48 89 e5                        mov    %rsp,%rbp
4004f1: c7 45 fc 00 00 00 00            movl   $0x0,-0x4(%rbp)
4004f8: 83 45 fc 01                     addl   $0x1,-0x4(%rbp)
4004fc: eb fa                           jmp    4004f8 <loop+0xb>
```

The **program counter** (PC), known as %rip in x86-64, stores the address in memory of the ***next instruction*** to be executed.



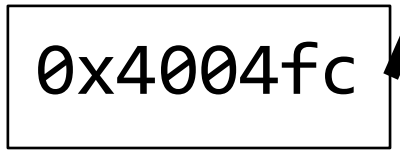
4004fd	fa
4004fc	eb
4004fb	01
4004fa	fc
4004f9	45
4004f8	83
4004f7	00
4004f6	00
4004f5	00
4004f4	00
4004f3	fc
4004f2	45
4004f1	c7
4004f0	e5
4004ef	89
4004ee	48
4004ed	55

%rip

```
0000000004004ed <loop>:
4004ed: 55                              push   %rbp
4004ee: 48 89 e5                        mov    %rsp,%rbp
4004f1: c7 45 fc 00 00 00 00            movl   $0x0,-0x4(%rbp)
4004f8: 83 45 fc 01                     addl   $0x1,-0x4(%rbp)
4004fc: eb fa                           jmp    4004f8 <loop+0xb>
```

Special hardware sets the program counter to the next instruction:

%rip += size of bytes of current instruction



4004fd	fa
4004fc	eb
4004fb	01
4004fa	fc
4004f9	45
4004f8	83
4004f7	00
4004f6	00
4004f5	00
4004f4	00
4004f3	fc
4004f2	45
4004f1	c7
4004f0	e5
4004ef	89
4004ee	48
4004ed	55

Reverse Engineering 1

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

----------

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 ___?___;
}
----------

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 ___?___;
}
----------

func:
  movq %rdi, %rax
  leaq 1(%rdi), %rcx
  movq %rcx, (%rsi)
  cqto
  idivq %rcx
  movq %rdx, %rax
  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          // 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
```

Helpful Assembly

- **Course textbook** (reminder: see relevant readings for each lecture on the Schedule page, http://cs107.stanford.edu/schedule.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

References + Advanced Reading

References:

- Stanford guide to x86-64: https://web.stanford.edu/class/cs107/guide/x86-64.html
  CS107 one-page of x86-64: https://web.stanford.edu/class/cs107/resources/onepage_x86-64.pdf
  gdbtui: https://beej.us/guide/bggdb/
  More gdbtui: https://sourceware.org/gdb/onlinedocs/gdb/TUI.html
  Compiler explorer: https://gcc.godbolt.org

Advanced Reading:

- x86-64 Intel Software Developer manual: https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf
  history of x86 instructions: https://en.wikipedia.org/wiki/X86_instruction_listings
  x86-64 Wikipedia: https://en.wikipedia.org/wiki/X86-64

Lecture Plan

- **Recap: mov** so far
- Data and Register Sizes
- The **lea** Instruction
- Logical and Arithmetic Operations
- Practice: Reverse Engineering
- **Privacy and Trust**

Reference Sheet:
`https://web.stanford.edu/class/archive/cs/cs107/cs107.1248/guide/x86-64.html`
See more guides on Resources page of course website!

Privacy and Trust

- Our learning about assembly and program execution helps us better understand computer security (the protection of data, devices, and networks from disruption, harm, theft, unauthorized access or modification).
- Computer security is important in part because it enables privacy.
- In understanding computer security, it's essential to understand the context in which it comes up (privacy and trust).

Data Breaches

Privacy/trust example: data breaches

- California list of data security breaches: link
- How does a data breach make a customer feel?

Privacy

What is privacy? 4 possible framings in two categories:

Individualist: the value of privacy as an individual right

- Privacy as **control of information** – controlling how our private information is shared with others.
- Privacy as **autonomy** – capacity to choose/decide for ourselves what is valuable.

Social: the value of privacy for a group

- Privacy as **social good** – social life would be unlivable without privacy.
- Privacy (protection) as based in **trust** – privacy enables trusting relationships

Privacy

Privacy as **control of information** – controlling how our information is communicated to others.

- Consent requires *free* choice with available alternatives and *informed* understanding of what is being offered.
- How many of you just skip past the terms of service for new online services you sign up for?
- Do you feel in control of your information with the services you choose to use? Why or why not? If you're working on a service, how can you respect privacy while achieving product goals?
- Control over personal data being collected (e.g. data exports from services you use, privacy dashboards, device privacy protections)

Privacy

Art. 1 GDPR

Subject-matter and objectives

1. This Regulation lays down rules relating to the protection of natural persons with regard to the processing of personal data and rules relating to the free movement of personal data.
2. This Regulation protects fundamental rights and freedoms of natural persons and in particular their right to the protection of personal data.
3. The free movement of personal data within the Union shall be neither restricted nor prohibited for reasons connected with the protection of natural persons with regard to the processing of personal data.

Media & Entertainment

Instagram launches "Data Download" tool to let you leave

Josh Constine / 9:44 AM PDT • April 24, 2018 Comment

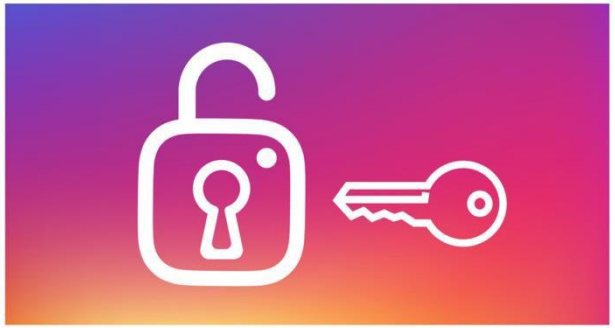


Image Credits: Bryce Durbin/TechCrunch /

TECH / APPLE / GOOGLE

Apple now lets you automatically transfer your iCloud Photo Library to Google Photos

/ Not everything can come along for the ride, though

By Mitchell Clark

Mar 3, 2021 at 1:10 PM PST

Google Report content on Google

Personal Data Removal Request Form

For privacy and data protection reasons (such as pursuant to the EU General Data Protection Regulation) you may have the right to ask for certain personal data relating to you to be removed.

This form is for requesting the removal of specific results for queries that include your name from Google Search. Google LLC is the controller responsible for the processing of personal data carried out in the context of determining the results shown by Google Search, as well as handling delisting requests sent through this form.

Privacy

Privacy as **autonomy** – capacity to choose/decide for ourselves what is valuable.

- Links to autonomy over our own lives and our ability to lead them as we choose.
- Do you feel that your autonomy is always respected when using products and services? Why or why not?

"[P]rivacy is valuable because it acknowledges our respect for persons as autonomous beings with the capacity to love, care and like—in other words, persons with the potential to freely develop close relationships" (Innes 1992)

Individualist Models of Privacy

Privacy as **autonomy** and privacy as **control over information** focus the value of privacy at an individual level.

- Individual privacy can conflict with interests of society or the state.
- Many debates over ”privacy vs. security” – whether one should be sacrificed for the other
  - Apple v. FBI case re: unlocking iPhones (link)
  - Debates around encryption (link)
- Where do your beliefs fall in balancing privacy and security? When (if at all) is it ok to sacrifice one, and how much?

Privacy

Privacy as **social good** – social life would be unlivable without privacy.

- Privacy has a social value in bringing about the kind of society we want to live in.
- What would society look like without privacy?

Privacy

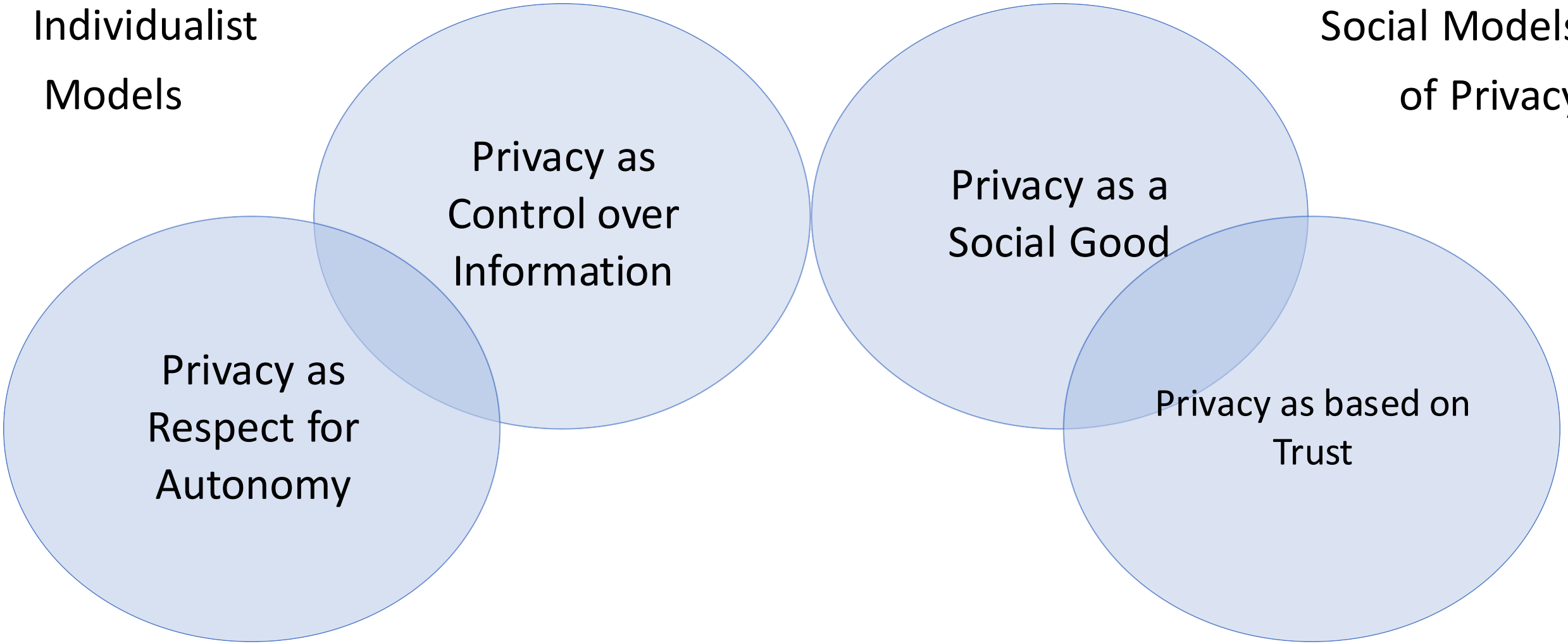
Privacy (protection) as based in **trust** – privacy enables trusting relationships

- Privacy may help enable trusting relationships essential for cooperation.
  - For instance, a *fiduciary*: someone who stands in a legal or ethical relationship of trust with another person (or group). The fiduciary must act for the benefit of and in the best interest of the other person. E.g. tax filer with access to your bank account
    - Should anyone who has access to personal info have a *fiduciary* responsibility? (Richards & Hartzog 2020).
- This model of privacy stresses the essential relationship of trust placed in any holder of personal data and the responsibilities that result from this trust.

Models of Privacy

Individualist Models

Social Models of Privacy



Who Should We Trust?

Both security and privacy rely on trusted people (who administer security, perform penetration tests, submit vulnerabilities to databases, or keep private information secret). The final piece of the security puzzle is understanding trust.

Trust = Reliance + Risk of Betrayal

What makes trust unique to relationships between people is that trust exposes one to being *betrayed or being let down* (Baier 1986).

Penetration Testing & Trust

Penetration testing is the practice of encouraging or hiring security researchers / contractors to find vulnerabilities in one's own code or system.

- Position of trust – tester is given access to the system and encouraged to find exploitable vulnerabilities, expected to share what they have found with you.
- Means *relying on* their skill at finding vulnerabilities and *trusting* that their ethical compass will lead them to tell you and to act as a trustworthy *fiduciary* (guardian of your interests).

In Assignment 5, you have the opportunity to explore this further!

Loss of Privacy

Loss of privacy can cause us various harms, including:

- ***Aggregation*:** combining personal information from various sources to build a profile of someone
- ***Exclusion:*** not knowing how our information is being used, or being unable to access or modify it (Google removing personal info from search – link)
- ***Secondary Use***: using your information for purposes other than what was intended without permission.

Mitigation: Differential Privacy

Differential privacy is a formal measure of privacy for datasets to try and protect individuals from aggregation by making them harder to identify (Dwork 2008).

- Imagine a large database, e.g., a medical database, with personal information and records of past activity tied to a name.
- The records might be useful for research purposes, or to train a machine learning model to predict future health outcomes, but what if giving access to the records exposed the privacy of individual person's health records?
- Differential privacy adds inconsequential noise (e.g., changing a birthday from 2001 to 2002) or removes records to make individuals harder to identify while preserving the utility of the dataset overall.

Trust Models

In every evaluation of privacy, we can ask: who is trusted? Who is distrusted? Does this model concentrate trust (and therefore power) in a single individual or small group, or does it distribute trust?

Differential Privacy's Trust Model

Differential privacy assumes that the only threat to privacy is an *external user querying the database* who must be prevented from aggregating data that could identify a user.

- In other words, the *trust model* of differential privacy is that the database owners and maintainers are to be fully trusted, and no one else.
- But is that the only threat? Differential privacy does not protect against improper use by people with full access to data or against leaks of the whole database, which may be the primary data exposure risks.

Differential privacy also does not question the assumption that amassing & storing large amounts of personal data is worth the risk of inevitable leaks (Rogaway 2015).

What should someone do if they find a vulnerability? How can we incentivize responsible disclosure?

86

Disclosure

What's the best way to disclose vulnerabilities?

- **Full disclosure?** Make vulnerabilities public as soon as they are found? *Few people now endorse this approach due to its drawbacks.*
- **Responsible disclosure?** Privately alert software maker to fix in reasonable amount of time before publicizing? *Most common, and recommended by ACM code of ethics.*

87

Disclosure

- Various entities may want to financially reward people for finding and reporting vulnerabilities.
- The US Federal Government is one of the largest discoverers and purchasers of 0-day vulnerabilities. It follows a “Vulnerability Equities Process” (VEP) to determine which vulnerabilities to responsibly disclose and which to keep secret and use for espionage or intelligence gathering.

88

How do we weigh competing stakeholder interests here, such as country vs. individual?

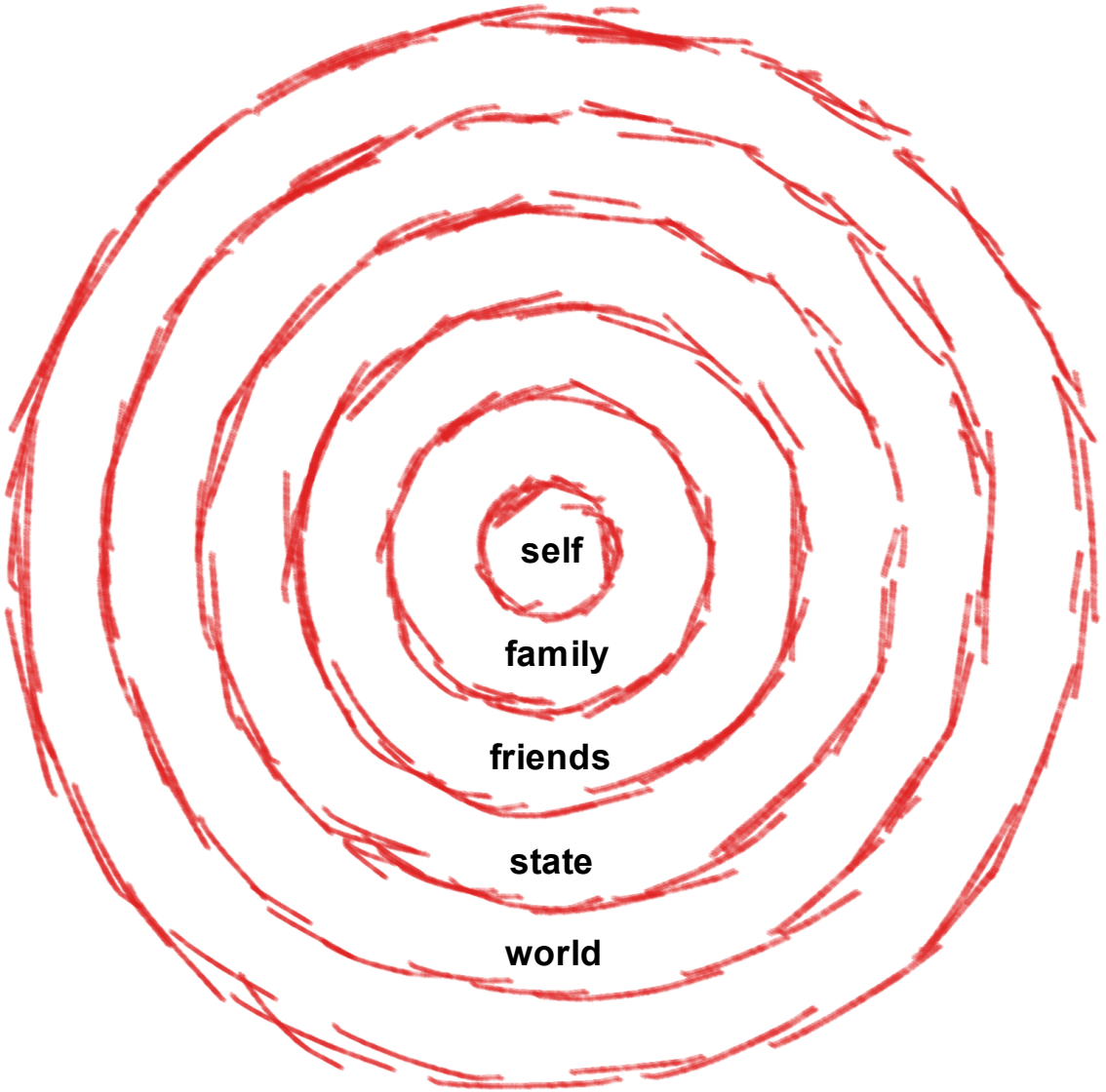
89

Partiality

- *Partiality* holds that it's acceptable to give preferential treatment to some people based on our relationships to them or shared group membership with them.
- *Impartiality* involves "acting from a position that acknowledges that all persons are ... equally entitled to fundamental conditions of well-being and respect."

90

Partiality



91

Degrees of Partiality

Partiality: preference towards own family, friends, and state is morally acceptable or even required

Partial Cosmpolitanism: limited preference towards own state acceptable

Universal Care: preference towards family acceptable but not towards state

Impartial Benevolence: same moral responsibilities towards all people

92

Case Study: EternalBlue

2012-2017: NSA secretly stores the EternalBlue Microsoft vulnerability and uses it to spy on both US and non-US citizens.

early 2017: EternalBlue stolen by hacker group the ShadowBrokers. NSA discloses EternalBlue to Microsoft.

March 14, 2017: Microsoft releases a patch for the vulnerability.

May 12, 2017: EternalBlue is the basis of the WannaCry and other ransomware attacks, leading to downtime in critical hospital and city systems and over $1 billion of damages.

93

Microsoft's Argument

"[T]his attack provides yet another example of why the **stockpiling of vulnerabilities** by governments is such a problem. …

We need governments to consider the **damage to civilians** that comes from hoarding these vulnerabilities and the use of these exploits.

This is one reason we called in February for a new "Digital Geneva Convention" to govern these issues, including a **new requirement for governments to report vulnerabilities to vendors**, rather than stockpile, sell, or exploit them.

And it's why we've pledged our support for **defending every customer everywhere** in the face of cyberattacks**, regardless of their nationality**."

Full post here

94

Critical Questions

- Do we have special obligations to our own country and to protect our people? If so, what would this mean?
- If intentionally exploiting a vulnerability is wrong when done by a private citizen, is it equally wrong when done by the government?
- Should I be loyal to my country, a citizen of the world, or both?
- When should I give preference to my family members and when should I strive to treat all equally?

What you choose matters – the moral obligations you take on constitute who you are.

95

Revisiting EternalBlue

Federal Government

Microsoft

Partiality: preference towards own family, friends, and state is morally acceptable or even required	Partial Cosmpolitanism: limited preference towards own state acceptable	Universal Care: preference towards family acceptable but not towards state	Impartial Benevolence: same moral responsibilities towards all people

96

Partiality Takeaways

- Understanding partiality helps us understand how we balance cases of competing interests and where we may personally fall on this spectrum.
- In order to evaluate situations, it's critical to understand the good and the bad that may come of it (e.g. EternalBlue). Better understanding privacy and privacy concerns is critical to this! (more later)

97