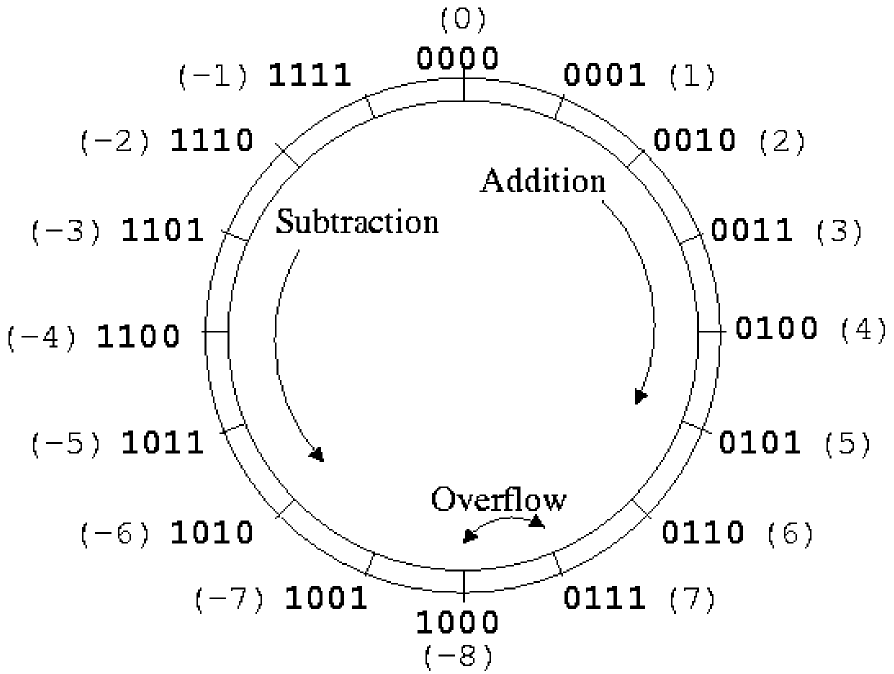
CSE351: Section 3

Number Representations
and x86 ISA

October 13, 2011

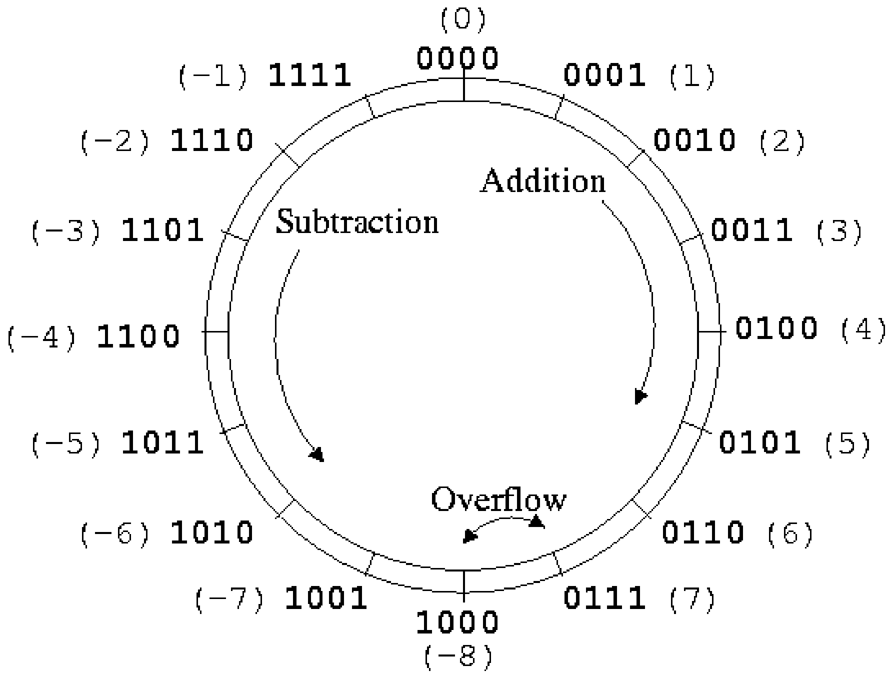
Review: Representing Integers

- Signed and unsigned values
  - Representing unsigned?
  - Representing signed?
- What is the two's complement representation?
  - Flip the bits and add 1
  - Ex: 4 is 0100, -4 is 1100:
    - Flip the bits: 1011
    - Add 1: 1100



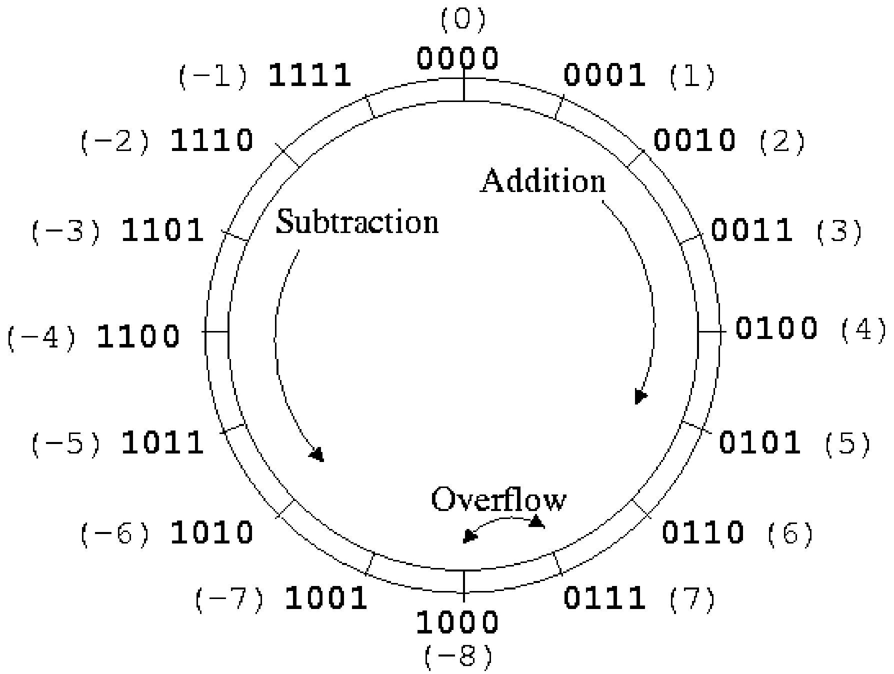
Review: Representing Integers

- Why Two's Compliment?



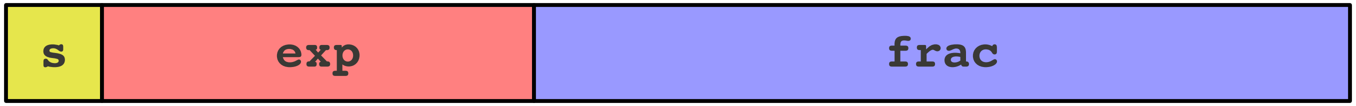
Review: Representing Integers

- Why Two's Compliment?
  - One value for 0 (zero)
    - Sign/magnitude has 0 and -0; leads to a lot of special cases
  - Works with existing adders
    - We don't need special signed/unsigned machinery



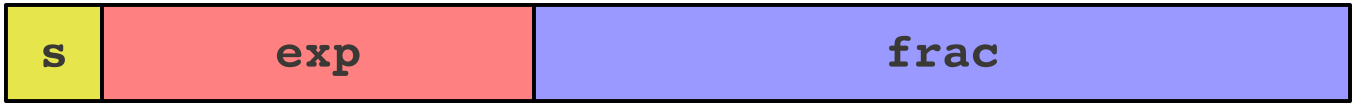
Review: Representing Floating-Point Values

- **Numerical Form:**
  - $(-1)^{s} * M * 2^{E}$
  - Sign bit **s** determines whether number is positive or negative
  - Mantissa **M** normally a fractional value between [1.0,2.0)
  - Exponent **E** weights value by power of two
- **Encoding**:
  - MSB **`s`** is sign bit **s**
  - **`frac`** field encodes **M** (but is not exactly **M**)
  - **`exp`** field encodes **E** (but is not exactly **E**)

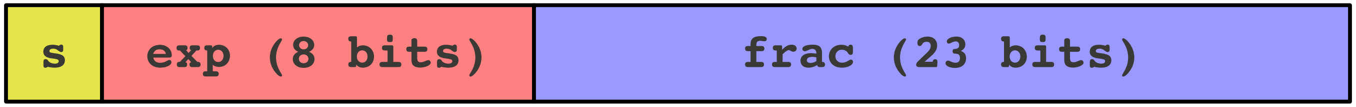


Review: Representing Floating-Point Values

- **Numerical Form:** $(-1)^{s} * M * 2^{E}$
- **Encoding**:
  - MSB **`s`** is sign bit **s**
  - **`frac`** field encodes **M** (but is not exactly **M**)
    - When M is represented as 1.xxxxxxxx in binary, M contains xxxxxxx
  - **`exp`** field encodes **E** (but is not exactly **E**)
    - **`exp`** = **E** + Bias
    - Bias = $2^{|exp|-1} - 1$ (e.g., 127 for 8 bit **`exp`**)



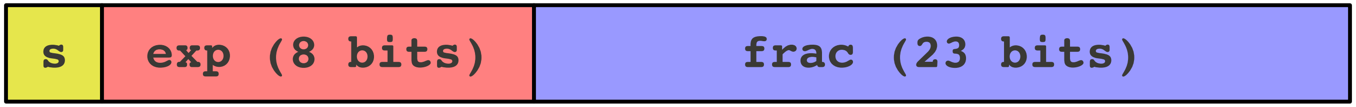
Review: Normalized Floating-Point Example



s	exp (8 bits)	frac (23 bits)

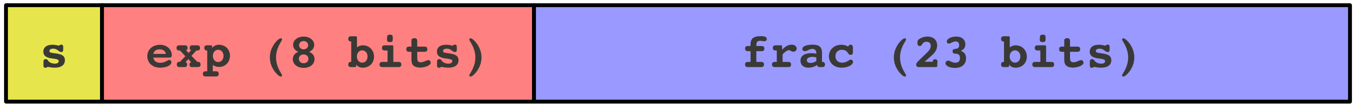
- How is float 12345.0 represented?
- Value:
  - $12345.0_{10} = 11000000111001_2$
    $= 1.1000000111001_2 \text{ x } 2^{13}$

Review: Normalized Floating-Point Example



- How is float 12345.0 represented?
- Value:
  - $12345.0_{10} = 11000000111001_2$
    $= 1.1000000111001_2 \times 2^{13}$
- Mantissa:
  - M $= 1.\underline{1000000111001}_2$
    frac = $\underline{1000000111001}0000000000_2$ (Need to extend to fill all 23 bits)

Review: Normalized Floating-Point Example



s | exp (8 bits) | frac (23 bits)

- How is float 12345.0 represented?
- Value:
  - $12345.0_{10} = 11000000111001_2$
    $= 1.1000000111001_2 \times 2^{13}$
- Mantissa:
  - M $= 1.\underline{1000000111001}_2$
    frac = $\underline{1000000111001}0000000000_2$ (Need to extend to fill all 23 bits)
- Exponent:
  - E = 13
    Bias = $2^7 - 1 = 127$
    exp = $140_{10} = 10001100_2$

Review: Normalized Floating-Point Example

s	exp (8 bits)	frac (23 bits)

- How is float 12345.0 represented?
- Value:
  - $12345.0_{10} = 11000000111001_2$
    $= 1.1000000111001_2 \times 2^{13}$
- Mantissa:
  - $M \quad = 1.\underline{1000000111001}_2$
    $\text{frac} = \quad \underline{1000000111001}0000000000_2$ (Need to extend to fill all 23 bits)
- Exponent:
  - $E = 13$
    $\text{Bias} = 2^7 – 1 = 127$
    $\exp = 140_{10} = 10001100_2$

0	10001100	10000001110010000000000

Normalization and Special Values

- “Normalized” means mantissa is of form 1.xxxxxxxxxx
  - Leading 1 is implied, don't need to store it
- Special values:
  - 000...00 represents zero
  - exp = 111...11, frac = 000...00 represents INFINITY
    - Sign bit determines if it is +INF or -INF
    - E.g., 10.0 / 0.0 = INF
  - exp = 111...11, frac != 000...00 represents NaN
    - E.g., 0 * INF = NaN

Properties of Floating-Point Values

- Not really associative or distributive. Why?
  - Let a = 1.52342, b = 6.2342342, c = 2.2523555
  - (a + b) + c = 10.010009**700000001**
    a + (b + c) = 10.010009**699999999**
  - a * (b + c) = 12.92864048077 4**000**
    a * b + a * c = 12.928640480774**002**
- Infinities and NaNs have issues
  - Additive inverses?
- Overflow and infinity
  - Only have so many bits of exponent; if it overflows, we get INF

Floating-Point Values and the Programmer

```
#include <stdio.h>

int main(int argc, char* argv[]) {

  float f1 = 1.0;
  float f2 = 0.0;
  int i;
  for ( i=0; i<10; i++ ) {
    f2 += 1.0/10.0;
  }

  printf("0x%08x  0x%08x\n", *(int*)&f1, *(int*)&f2);
  printf(“f1 == f2? %s\n”, f1 == f2 ? “yes” : “no”);
  printf("f1 = %10.8f\n", f1);
  printf("f2 = %10.8f\n\n", f2);

  f1 = 1E30;
  f2 = 1E-30;
  float f3 = f1 + f2;
  printf ("f1 == f3? %s\n", f1 == f3 ? "yes" : "no" );

  return 0;
}
```

Floating-Point Values and the Programmer

```
#include <stdio.h>

int main(int argc, char* argv[]) {

  float f1 = 1.0;
  float f2 = 0.0;
  int i;
  for ( i=0; i<10; i++ ) {
    f2 += 1.0/10.0;
  }

  printf("0x%08x  0x%08x\n", *(int*)&f1, *(int*)&f2);
  printf(“f1 == f2? %s\n”, f1 == f2 ? “yes” : “no”);
  printf("f1 = %10.8f\n", f1);
  printf("f2 = %10.8f\n\n", f2);

  f1 = 1E30;
  f2 = 1E-30;
  float f3 = f1 + f2;
  printf ("f1 == f3? %s\n", f1 == f3 ? "yes" : "no" );

  return 0;
}
```

```
$ ./a.out
0x3f800000  0x3f800001
f1 == f2? no
f1 = 1.000000000
f2 = 1.000000119

f1 == f3? yes
```

Memory Referencing Bug

```
double fun(int i)
{
  volatile double d[1] = {3.14};
  volatile long int a[2];
  a[i] = 1073741824;
  return d[0];
}
```

Memory Referencing Bug

```
double fun(int i)
{
  volatile double d[1] = {3.14};
  volatile long int a[2];
  a[i] = 1073741824;
  return d[0];
}
```

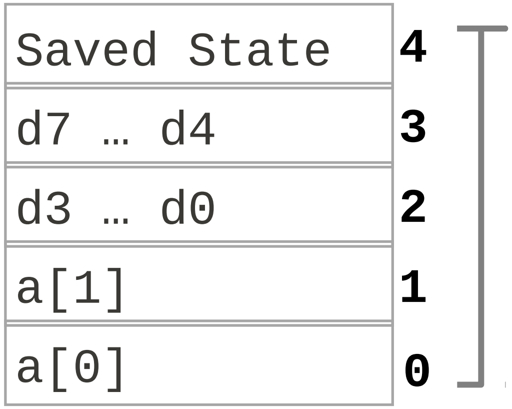
- What is the result of... ?
  - `fun(0)`
  - `fun(1)`
  - `fun(2)`
  - `fun(3)`
  - `fun(4)`

Memory Referencing Bug

```
double fun(int i)
{
  volatile double d[1] = {3.14};
  volatile long int a[2];
  a[i] = 1073741824;
  return d[0];
}
```

- What is the result of... ?
  - fun(0) → 3.14
  - fun(1) → 3.14
  - fun(2) → 3.1399998664856
  - fun(3) → 2.00000061035156
  - fun(4) → 3.14, then a segfault

Location accessed by fun(i)



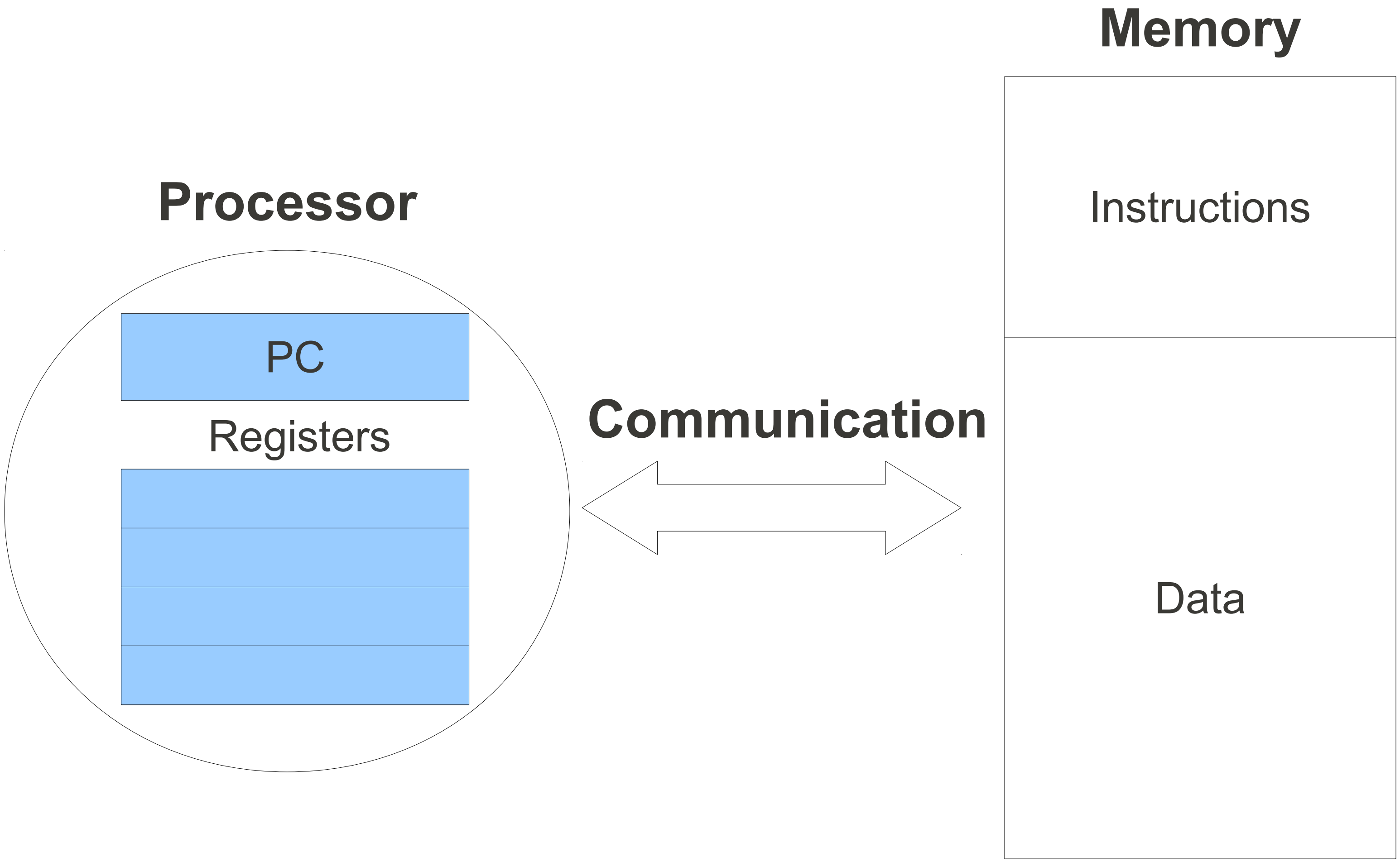
Saved State	4
d7 … d4	3
d3 … d0	2
a[1]	1
a[0]	0

Floating-Point Summary

- Floats have a finite number of bits
  - Overflow just like ints
- Some simple fractions have no exact representation
  - E.g. 0.1
- Calculations can lose precision, e.g., due to rounding
- Mathematically equivalent expressions can return different results

x86 ISA, C, and Assembly

The General ISA



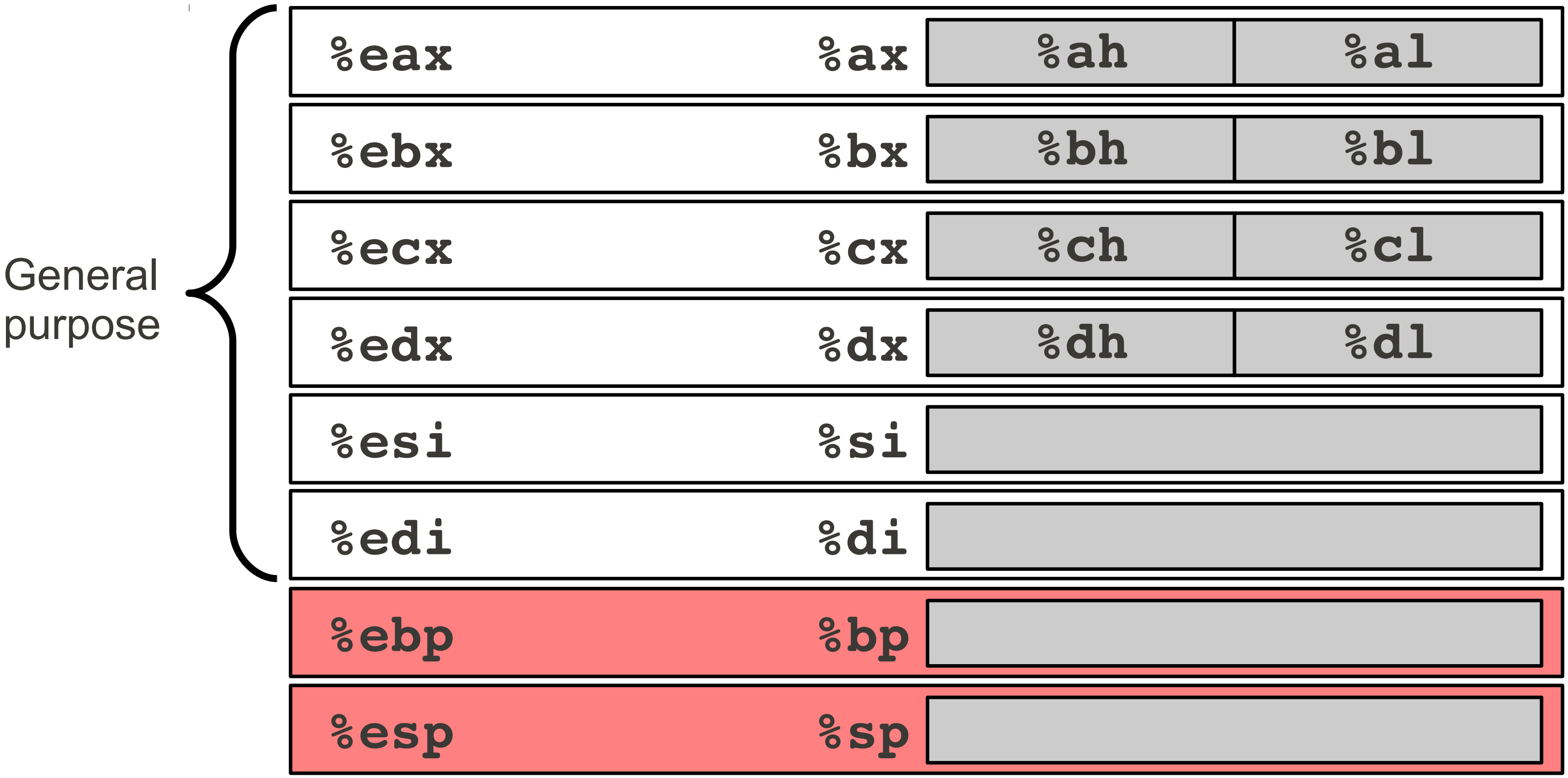
General ISA Design Questions

- What the programmer "sees"
- Defines HW/SW interface
  - What are the instructions?
    - What do they do?
    - How are they encoded?
  - How many registers? How wide are the registers?
  - How do you address memory?
- The ISA is an abstraction
  - Many different implementations by different manufacturers

Example: x86 and x86_64

- Complex Instruction Set Computers (CISC)
  - Some instructions do complex operations (e.g., copy strings)
  - Instructions are defined in detail in the manuals
- Registers are 32-bit for x86 and 64-bit for x86_64
- x86 has 8 registers for general use; x86_64 has 16
  - Convention dictates how these registers are used
- ISA also determines function calling conventions
  - x86 mostly uses stack to pass arguments to functions
  - X86_64 passes first six arguments directly in CPU registers

x86 Registers

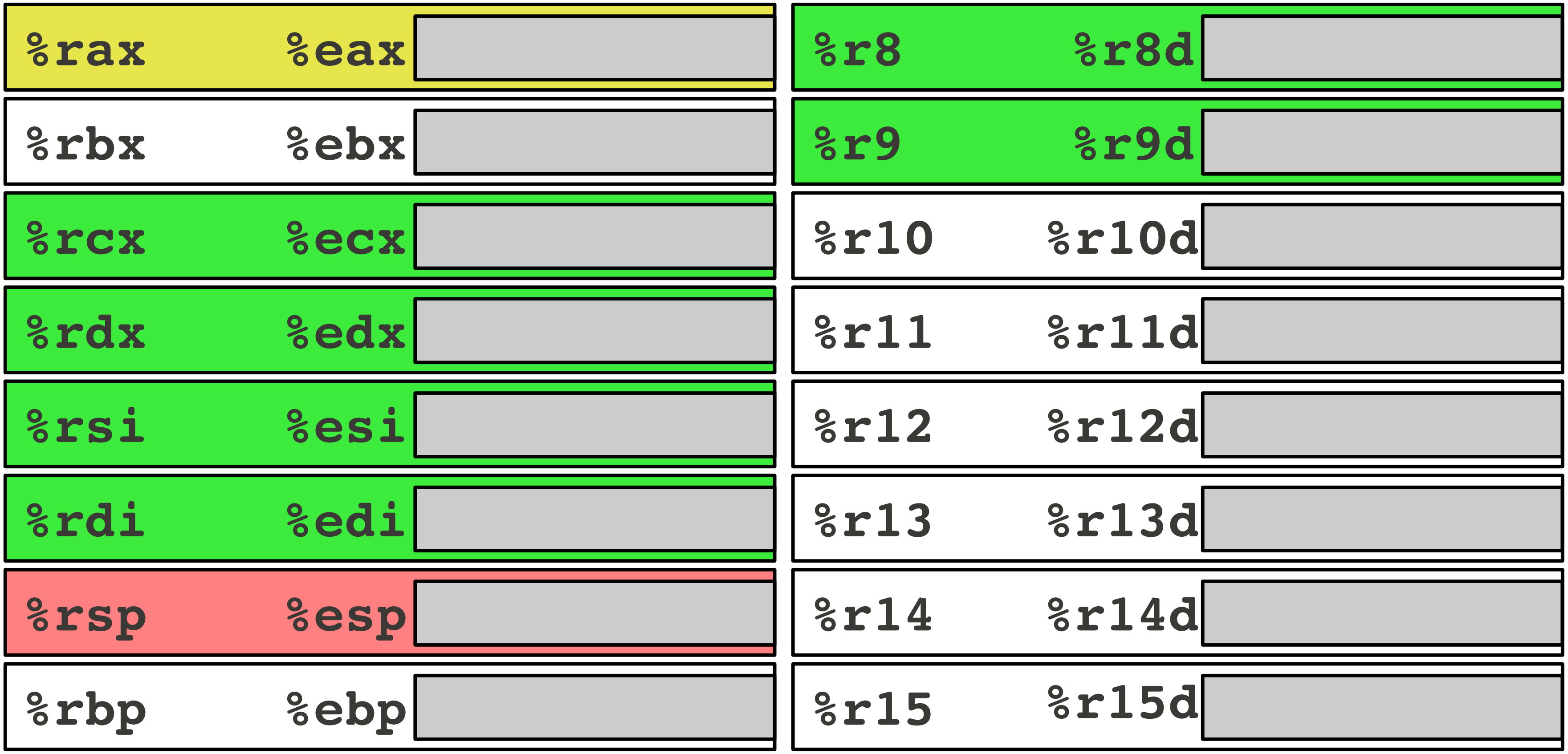


General purpose

%eax	%ax	%ah	%al
%ebx	%bx	%bh	%bl
%ecx	%cx	%ch	%cl
%edx	%dx	%dh	%dl
%esi	%si		
%edi	%di		
%ebp	%bp		
%esp	%sp		

- There are also other registers that can't be accessed directly: %eip, %eflags, %cs, %ds, etc.

x86_64 Registers



%rax	%eax	%r8	%r8d
%rbx	%ebx	%r9	%r9d
%rcx	%ecx	%r10	%r10d
%rdx	%edx	%r11	%r11d
%rsi	%esi	%r12	%r12d
%rdi	%edi	%r13	%r13d
%rsp	%esp	%r14	%r14d
%rbp	%ebp	%r15	%r15d

- More registers, different conventions
  - Some function arguments are now passed in %rdi, %rsi, %rdx, %rcx, %r8 and %r9
- There are also other registers that can't be accessed directly: %eip, %eflags, %cs, %ds, etc.

X86 Basics - Instructions

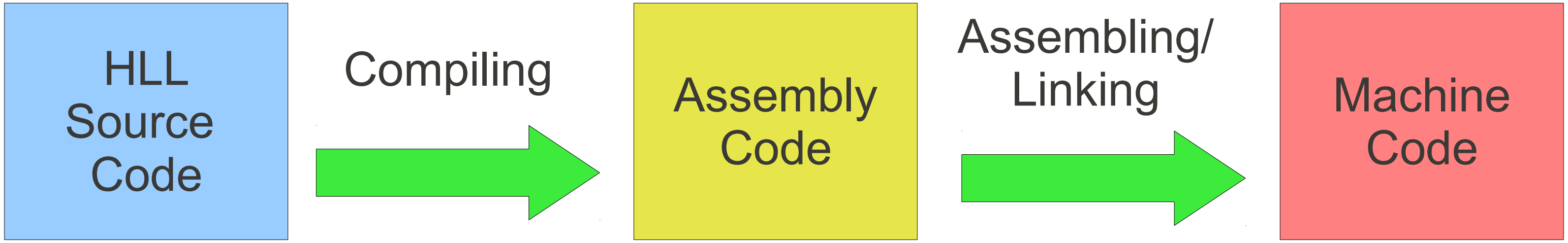
- Arithmetic
  - `add, sub, mul, idiv`
- Logical / Bitwise
  - `and, or, xor, neg, sal/shl, sar/shr`
- Control
  - `jmp, je, jne, jg, jl, jle, jge`
  - Use after test or cmp instruction
    - `test` – bitwise AND which sets flags
    - `cmp` – subtraction which sets flags
  - `ret` – used to return from a function
- Other
  - Stack insns: `push, pop`
  - Data manipulating: `mov, enter, leave`

X86 Basics – Data Sizes

- Instructions take a data size specifier as their last character
  - **L – operate on 4 bytes**
    - **<u>Ex:</u> addl, pushl, movl, cmpl**
  - **B – operate on least significant byte**
    - **<u>Ex:</u> movb, cmpb, testb**
- Need to be combined with appropriately named operands!
  - **<u>Ex:</u> addl %edx, %eax → valid!**

    cmpb %eax, %cl → invalid!

C-to-Assembly Example



HLL Source Code

Compiling

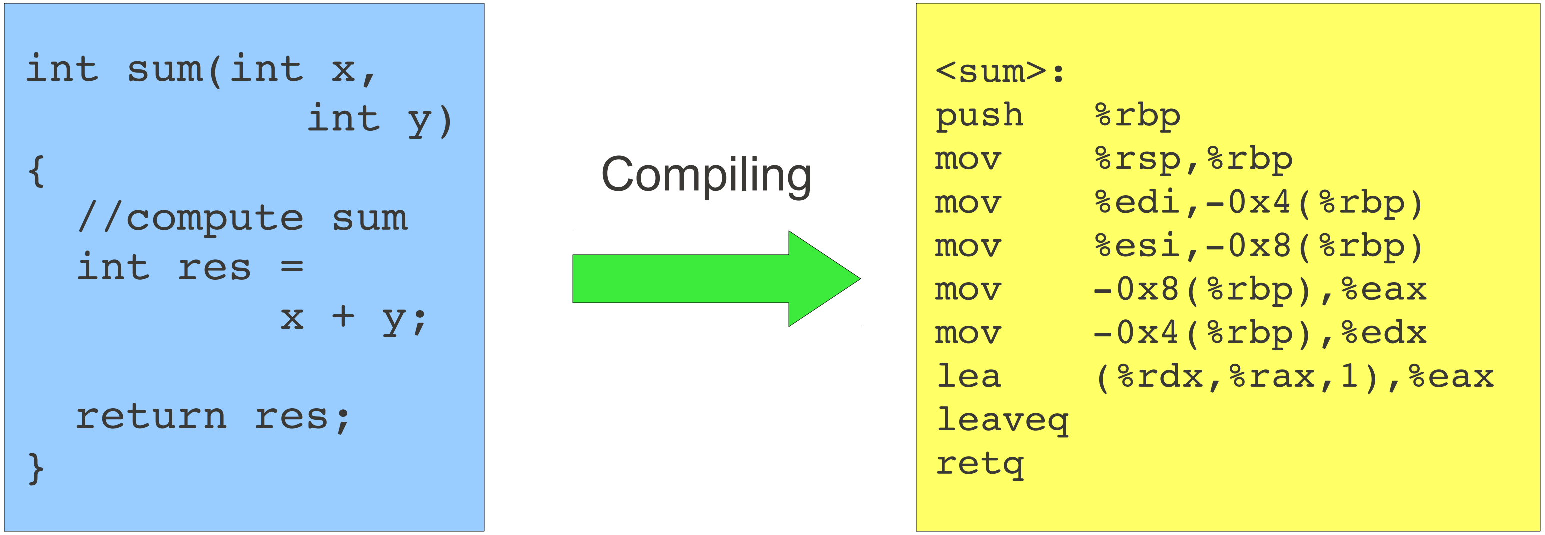
Assembly Code

Assembling/ Linking

Machine Code

Compiling

- Turning high level code (e.g., C) to intermediate assembly **for the target ISA**
    - Must compile the HL code multiple times if targeting multiple ISAs
- Can produce with: `gcc foo.c -S -o foo.s`



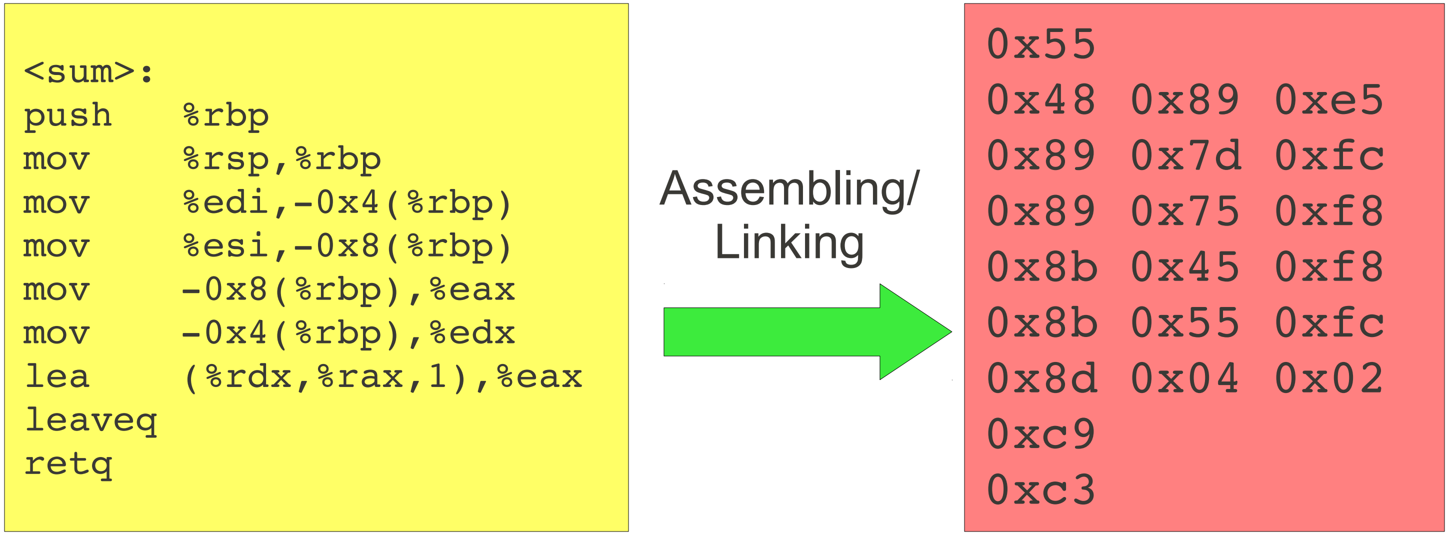
```
int sum(int x,
            int y)
{
  //compute sum
  int res =
          x + y;

  return res;
}
```

Compiling

```
<sum>:
push    %rbp
mov     %rsp,%rbp
mov     %edi,-0x4(%rbp)
mov     %esi,-0x8(%rbp)
mov     -0x8(%rbp),%eax
mov     -0x4(%rbp),%edx
lea     (%rdx,%rax,1),%eax
leaveq
retq
```

Assembling/Linking

- Transform human-readable assembly to machine-readable binary
- Can produce directly with `gcc`, or with `as`
- Linking additionally includes code from libraries
  - `printf, strlen, etc.`



```
<sum>:
push    %rbp
mov     %rsp,%rbp
mov     %edi,-0x4(%rbp)
mov     %esi,-0x8(%rbp)
mov     -0x8(%rbp),%eax
mov     -0x4(%rbp),%edx
lea     (%rdx,%rax,1),%eax
leaveq
retq
```

```
0x55
0x48 0x89 0xe5
0x89 0x7d 0xfc
0x89 0x75 0xf8
0x8b 0x45 0xf8
0x8b 0x55 0xfc
0x8d 0x04 0x02
0xc9
0xc3
```

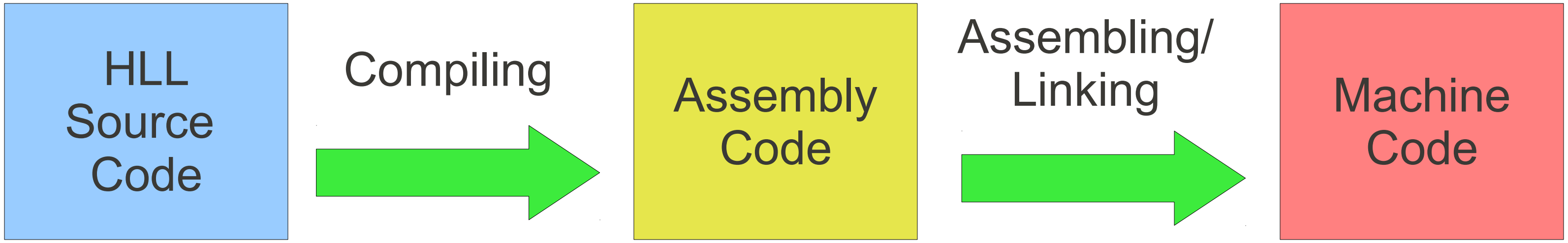
Going from Binary to Assembly

- Sometimes you want to go the other way
  - E.g., converting an executable binary back to assembly
  - Usually hard/impossible to go back to HLL
- Useful for debugging, reverse engineering, and **Lab 2**
- Two possible ways to do this:
  - Use GDB and the `disas' command

    ```
    – $ gdb foo
      > disas main
    ```
  - Objdump program from the command line

    ```
    – $ objdump -D foo
    ```
  - See man pages for specifics

C-to-Assembly Example



```
int sum(int x,
           int y)
{
  //compute sum
  int res =
          x + y;

  return res;
}
```

```
<sum>:
push    %rbp
mov     %rsp,%rbp
mov     %edi,-0x4(%rbp)
mov     %esi,-0x8(%rbp)
mov     -0x8(%rbp),%eax
mov     -0x4(%rbp),%edx
lea     (%rdx,%rax,1),%eax
leaveq
retq
```

```
0x55
0x48 0x89 0xe5
0x89 0x7d 0xfc
0x89 0x75 0xf8
0x8b 0x45 0xf8
0x8b 0x55 0xfc
0x8d 0x04 0x02
0xc9
0xc3
```