# 7. Instruction Set Architecture –
# - translation software
# - floating point representation

EECS 370 – Introduction to Computer Organization - Winter 2016

Profs. Valeria Bertacco & Reetu Das

EECS Department
University of Michigan in Ann Arbor, USA

# Announcements

- Exam conflicts / special accommodations: DEADLINE is this Sunday
- REMEMBER to do your reading assignments!
- Homework 2 is due NEXT Tuesday 2/2
- Project 1 is due NEXT Thursday 2/4

- Next Tuesday instructor: Prof. DAS!

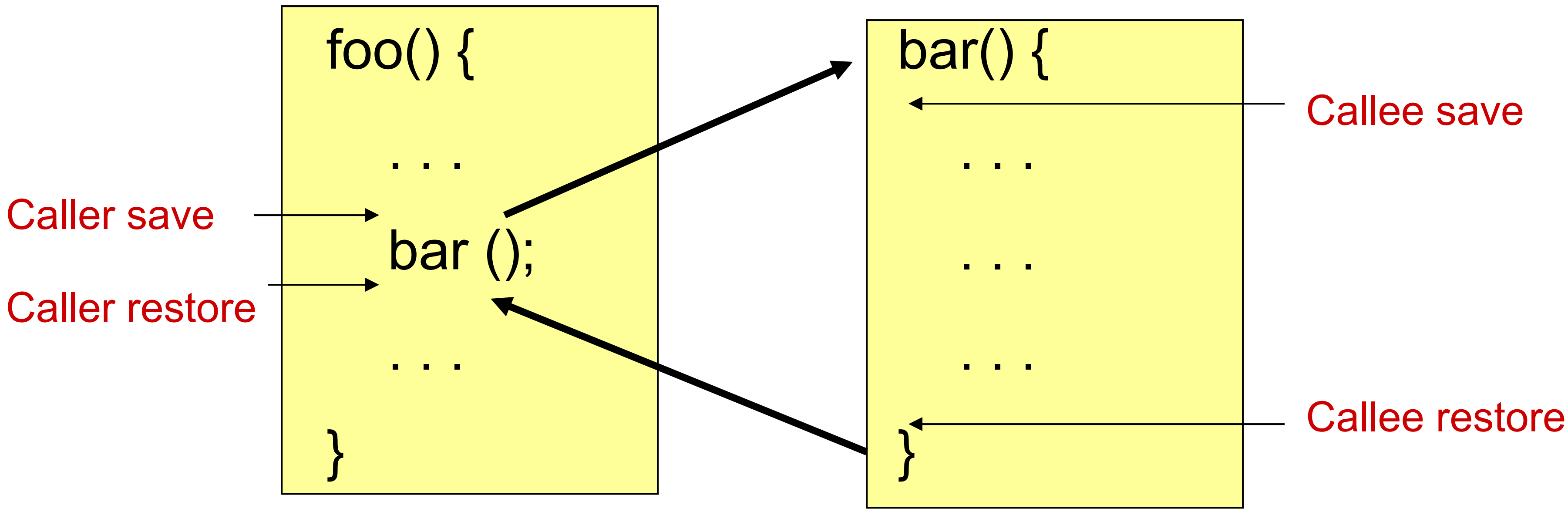
# Recap: last Tuesday

… we mostly talked about:

- Caller-saved registers and callee-saved registers
  - Most functions are both callers and a callee. Exceptions?
  - ISAs declare some registers “caller-saved registers” and other “callee-saved registers”
  - The compiler’s job is to map variables to registers so to minimize the number of memory accesses needed to execute the problem (memory accesses are SLOOOW, avoid them at all costs!)
  - In this type of problems you are a compiler

# Caller-Callee save/restore

CALLER-CALLEE




Caller save: Callee may change, so caller responsible for saving immediately before call and restoring immediately after call

Callee save: Callee may not change, so callee (called function) must leave these unchanged. Can be ensured by inserting saves at the start of the function and restores at the end

# WARM UP
# Class problem – Caller-saved vs. callee saved

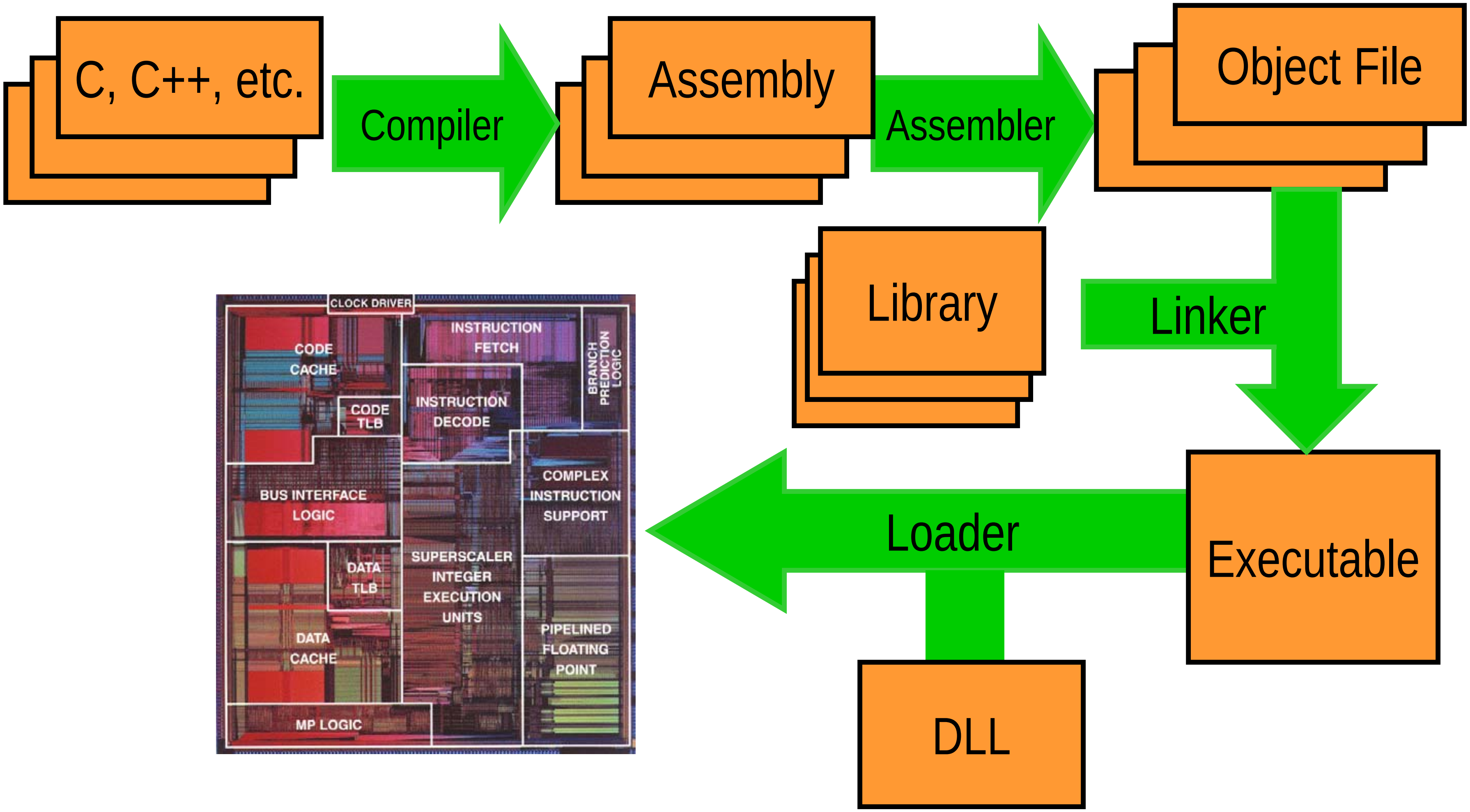
```
void main(){
  int a,b,c,d;
  .
  c = 5; d = 6;
  a = 2; b = 3;
  foo();
  d = a+b+c+d;
  .
  .
  .
}
```

```
void foo(x,y){
  int a,b,c,d,e;
  .
  .
  c = 1; d = 1;
  e = 1;
  a = 2; b = 3;
  foo(e-1,b-1);
  a = a + e;
  .
  .
  foo(b, a+b);
  b = a - b;
  .
  c++; d++;
}
```

# Caller-saved vs. callee saved problem

- Assume the function foo() is executed 15 times: it is called once from main, and then 7 times from the first call point, 7 times from the second call point.
- When the program is executed, how many regs need to be stored/loaded in total for the following scenarios:
  - Use a **caller-save** convention ?
  - Use a **callee-save** convention ?
  - Use a mixed **caller/callee**-save convention with 3 callee-s. and 3 caller-s. registers ?

# Source Code to Execution



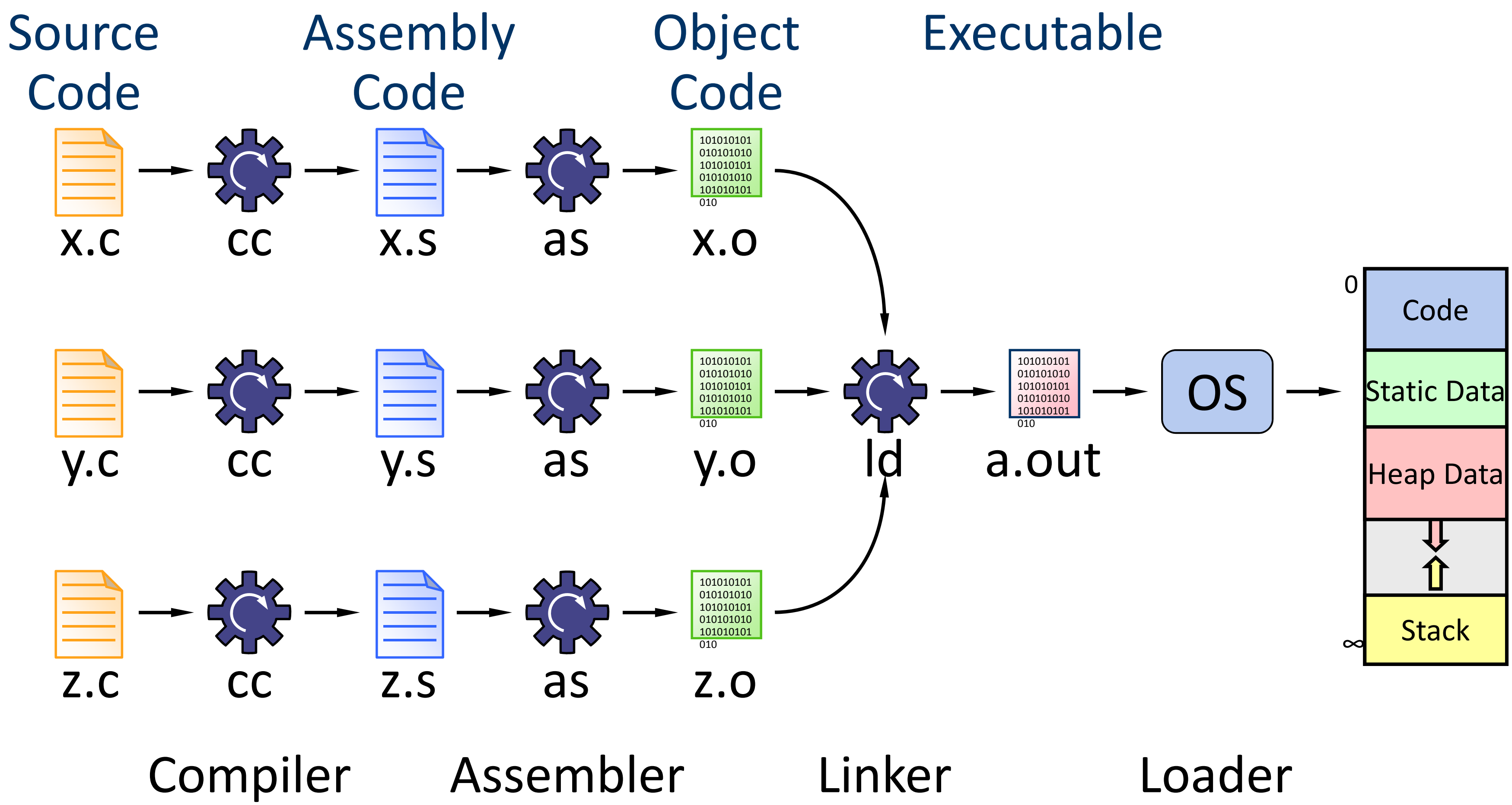
# What happens when you call gcc?

1. C preprocessor
   - Handles macros, #define, #ifdef, #if
   - gcc –E foo.c > foo.i    (foo.i contains preprocessed source code)
2. Compiler
   - gcc –S foo.c    (foo.s contains textual assembly)
3. Assembler
   - as foo.s –o foo.o or gcc –c foo.s
4. Linker
   - ld foo.o bar.o _bunch_of_other_stuff –o a.out

You can run gcc –v to see all the commands that it is running

- Note gcc does not call ld, it calls collect2, which is a wrapper that calls ld

# Source to Process Translation






Ack: Ousterhout

# Linux (ELF) object file format

*Object files contain more than just machine code instructions!*

Header: (this is an object file) contains sizes of other parts

Text: machine code

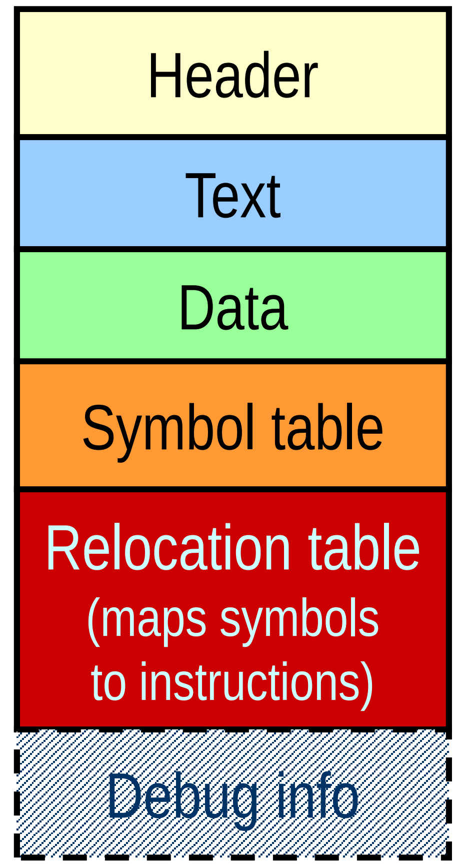
Data: global and static data

Symbol table: symbols and values

Relocation table: references to addresses that may change

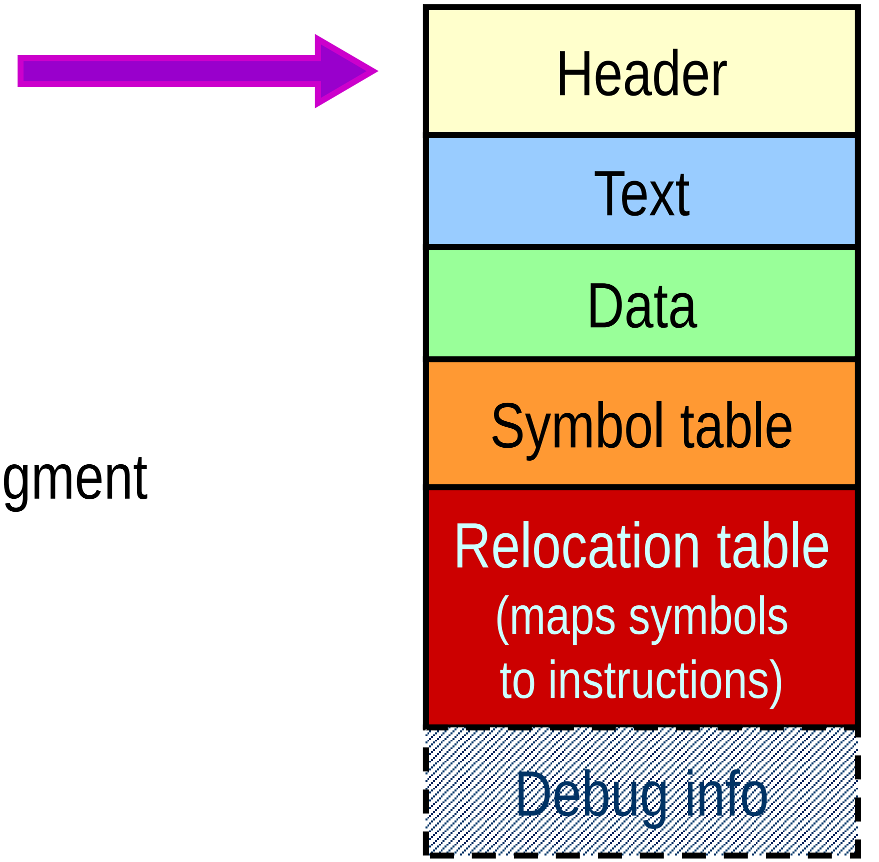
Debug info: mapping of object back to source (only exists when debugging options are turned on)

Object code format



# Linux (ELF) object file format (2)

Object code format

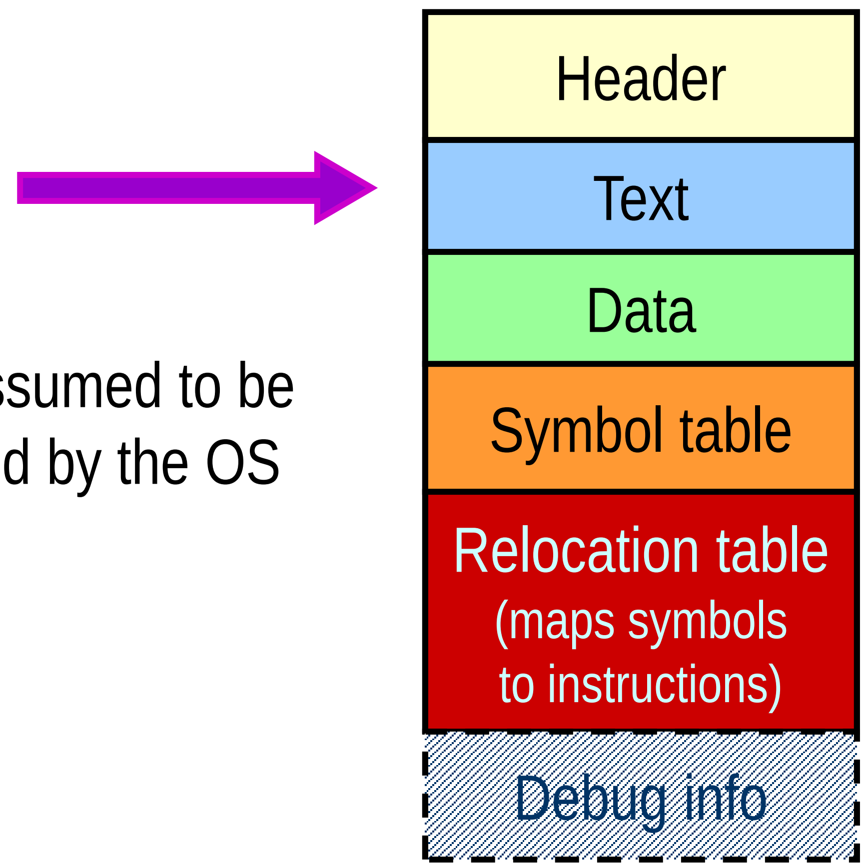



Header

- size of other pieces in file
- size of text segment
- size of static data segment
- size of uninitialized data segment
- size of symbol table
- size of relocation table

# Linux (ELF) object file format (3)

Object code format




Text segment

- machine code

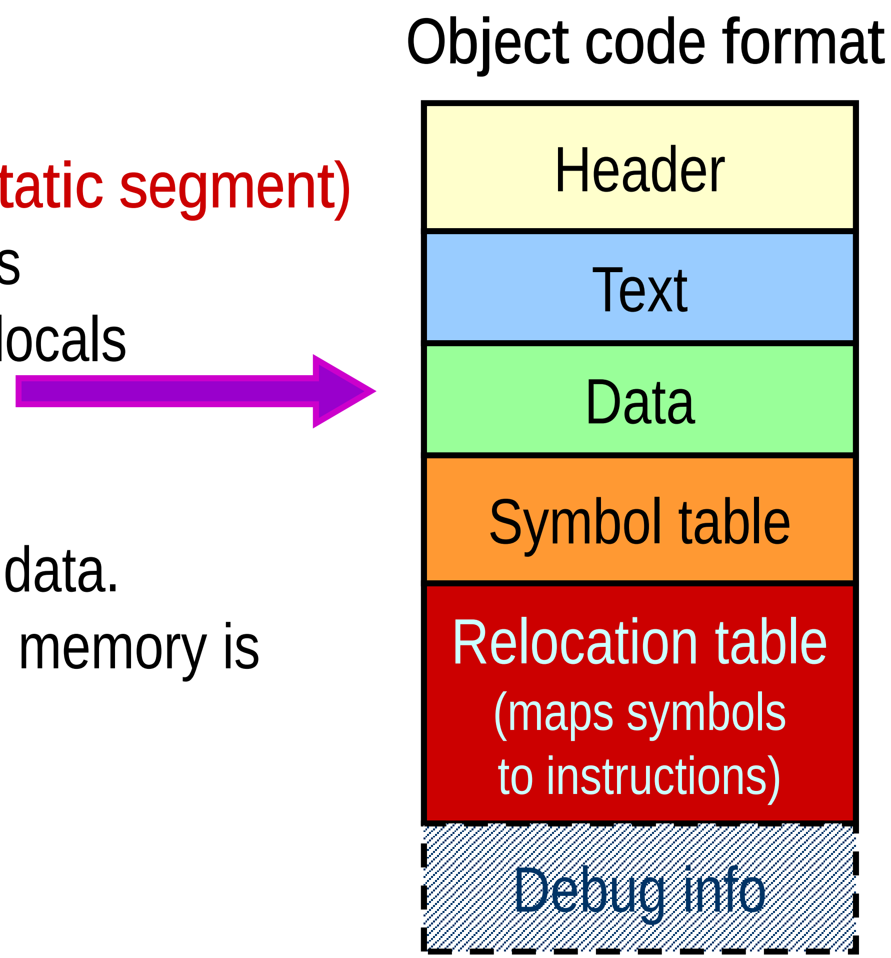
By default this segment is assumed to be read-only and that is enforced by the OS

# Linux (ELF) object file format (4)

Data segment (Initialized static segment)

- values of initialized globals
- values of initialized static locals

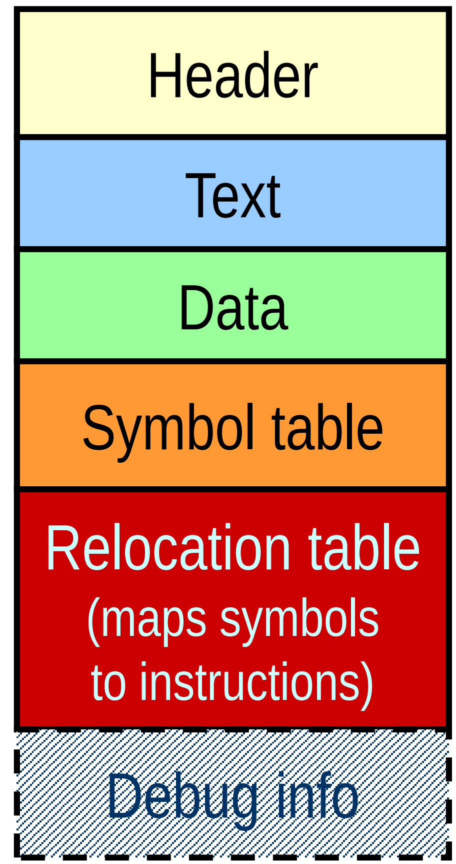
Doesn't contain uninitialized data.
Just keep track of how much memory is needed for uninitialized data



# Linux (ELF) object file format (5)

Symbol table:
- It is used by the linker to bind public entities within this object file (function calls and globals)
- Maps string symbol names to values (addresses or constants)
- Associates addresses with global labels.  Also lists unresolved labels

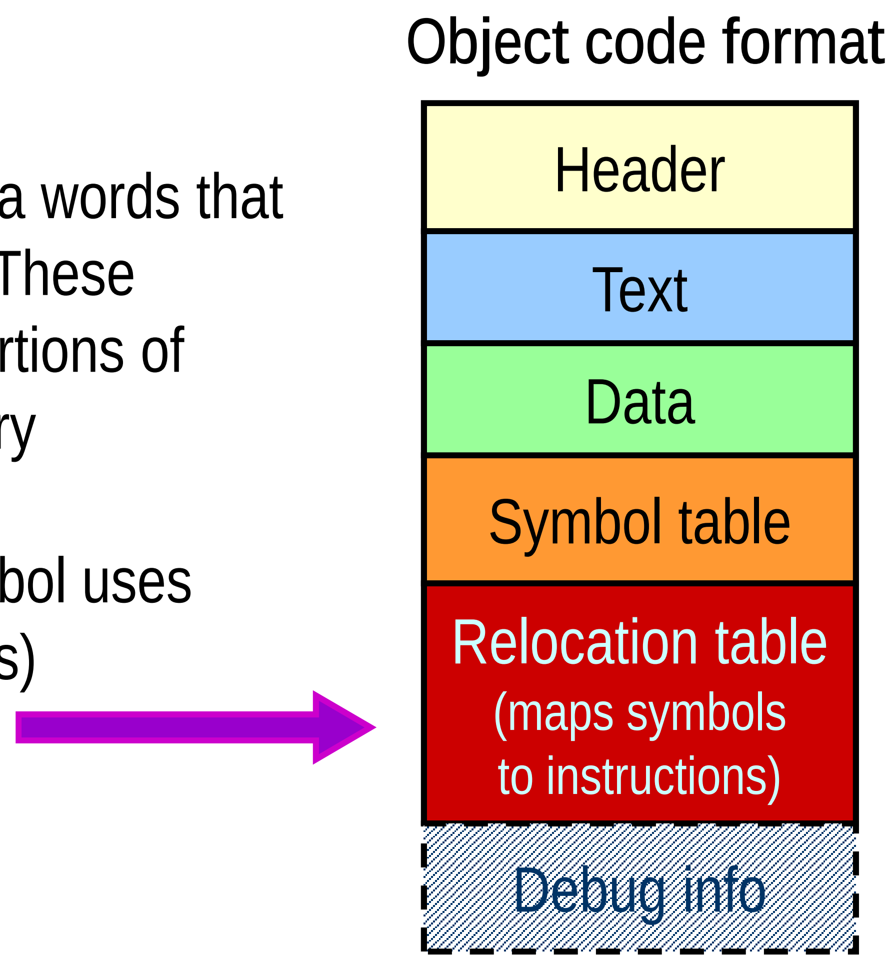
Object code format



# Linux (ELF) object file format (6)

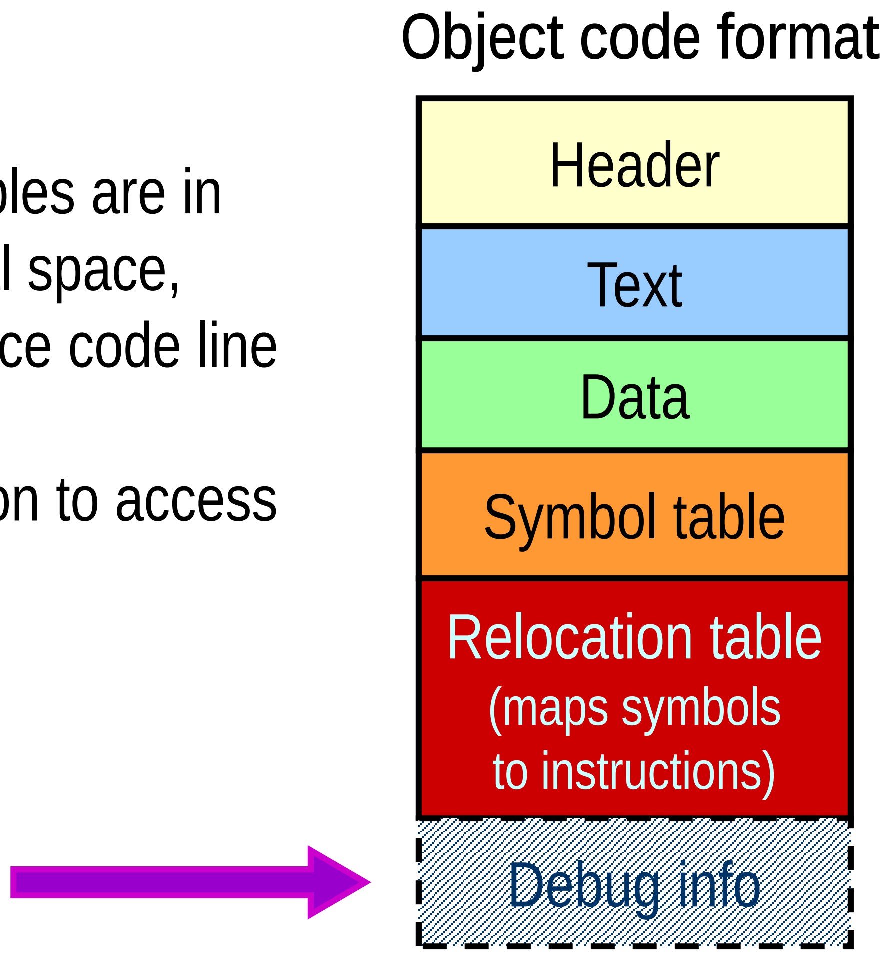
Relocation table :
identifies instructions and data words that rely on absolute addresses. These references must change if portions of program are moved in memory

Used by linker to update symbol uses (e.g., branch target addresses)



# Linux (ELF) object file format (7)

Debug info (optional) :
Contains info on where variables are in stack frames and in the global space, types of those variables, source code line numbers, etc.
Debuggers use this information to access debugging info at runtime



# Assembly → Object file

Snippet of C

```
int X = 3;
main() {
  Y = X;
  B();
  ...
```

Snippet of assembly code

```
ldr r12, [pc, #0]
ldr r0, [r12,#0]
b #0
.word DataSegment
bl B
```

| | | | |
|---|---|---|---|
| **Header** | Name | foo | |
| | Text size | 0x100 | |
| | Data size | 0x20 | |
| **Text** | Address | Instruction | |
| | 0 | ldr r12, [pc, #0] // r12=start of data | |
| | 4 | ldr r0, [r12, #4] | |
| | 8 | b #0 | |
| | 12 | .word DataSegment | |
| | 16 | bl B | |
| **Data** | … | | |
| | 4 | X | 3 |
| **Symbol table** | Label | Address | |
| | X | 4 | |
| | B | - | |
| | main | 0 | |
| **Reloc table** | Addr | Instruction type | Dependency |
| | 12 | .word | X |
| | 16 | bl | B |

# Linker, or Link editor, as it was once known

- Stitches independently created object files into a single executable file (i.e., a.out)
  - Step 1: Take text segment from each .o file and put them together.
  - Step 2: Take data segment from each .o file, put them together, and concatenate this onto end of text segments.
- What about libraries?
  - Libraries are just special object files.
  - You create new libraries by making lots of object files (for the components of the library) and combining them (see ar and ranlib on Unix machines).

- Step 3: Resolve cross-file references to labels
  - Make sure there are no undefined labels

# Linker – step 3 - continued

- Determine which memory locations the code and data of each file will occupy
  - Each function could be assembled on its own
  - Thus the relative placement of code/data is not known up to this point
  - Must relocate absolute references to reflect placement by the linker
    - PC-Relative Addressing (beq, bne): never fixup
    - Absolute Address (mov r15, X): always fixup
    - External Reference (usually bl): always fixup
    - Data Reference (often .word or movw/movt): always fixup

- Executable file contains <u>no relocation info or symbol table</u> – these are just used by assembler/linker (exception: DLL)

# Linker - continued

- Linker assumes first word of first text segment is at fixed address
- Linker knows:
  - Length of each text and data segment
  - Ordering of text and data segments
- Linker calculates:
  - Absolute address of each label to be jumped to (internal or external) and each piece of data being referenced
- To resolve references:
  - Search for reference (data or label) in all symbol tables
  - If not found, search library files (for example, for printf)
  - Once absolute address is determined, fill in the machine code appropriately

# Example Executable File

| | | |
|---|---|---|
| Header | Text size | 0x200 |
| | Data size | 0x40 |
| Text | Address | Instruction |
| | 0x0040 0000 | ldr r12, [pc, #0] |
| | 0x0040 0004 | ldr r0, [r12, #4] |
| | 0x0040 0008 | b #0 |
| | 0x0040 000c | .word 0x1000 0000 |
| | 0x0040 0010 | str r0, [sp, #-16] |
| | 0x0040 0014 | bl 0x400100 |
| | … | |
| | 0x0040 0100 | sub r13, r13, #20 |
| | 0x0040 0104 | bl 0x400200 |
| Data | 0x1000 0000 | .. |
| | 0x1000 0004 | X |

# Class Problem 1

In the object file for file1 and file2, which symbols are in the symbol table?

file1.c

```
extern int bar(int);
extern char c[];
int a;
int foo (int x) {
    int b;
    a = c[3] + 1;
    bar(x);
    b = 27;
}
```

file2.c

```
extern double d[];
char c[100];
int bar (int y) {
   char *e[100];
   d[3] = (double)y;
   c[20] = *e[7];
}
```

# Class Problem 2

file1.c

```
1 extern void bar(int);
2 extern char c[];
3 int a;
4 int foo (int x) {
5     int b;
6     a = c[3] + 1;
7     bar(x);
8     b = 27;
9 }
```

file2.c

```
extern double d[];
char c[100];
void bar (int y) {
   char *e[100];
   d[3] = (double)y;
   c[20] = *e[7];
}
```

A) What if file2.c contains extern int j, but no reference to j?

B) What if variable ‘e’ is static?

C) What if the externs in file1.c are deleted?

D) Which source lines require relocation during linking?

# Loader

- Executable file is sitting on the disk
- Puts the executable file code image into memory and asks the operating system to schedule it as a new process
  - Creates new address space for program large enough to hold text and data segments, along with a stack segment
  - Copies instructions and data from executable file into the new address space (this may be anywhere in memory)
  - Initializes registers (PC and SP most important)
- Linking used to be straightforward, but times are changing; it is not simple anymore.
  - We now delay some of the linking to load time
  - Some systems even delay some code optimization (usually a compiler job) to load time
  - Loaders must deal with more sophisticated operating systems

# Trends in software systems

- Programmers are expensive
- Applications are more sophisticated
  - 3D graphics, streaming video, etc
- Application programmers rely more on library code to make high quality apps while reducing development time
  - This means that more of the executable is library code
  - Why not keep those shared library routines in memory and link an object file at load time? (DLLs)
    - Executable files are smaller (not very important)
    - Updating library routines is easy
- Porting code to a variety of platforms is costly/time-intensive
  - Utilize virtual instruction sets (e.g., Java bytecode, C# CLR) and VMs
  - “Write once, run everywhere”

# Things to remember

- Compiler converts a single source code file into a single assembly language file
- Assembler removes pseudos, converts what it can to machine language, and creates a checklist for the linker (relocation table). This changes each .s file into a .o file
- Assembler does 2 passes to resolve addresses, handling internal forward references
- Linker combines several .o files and resolves absolute addresses
- Linker enables separate compilation, libraries that need not be compiled, and resolves remaining addresses
- Loader loads executable into memory and begins execution

# Floating point arithmetic

# Why Floating Point

- Have to represent non-integer values somehow
- Rational numbers
  - Ok, but can be cumbersome to work with
  - Falls apart for sqrt(2) and other irrational numbers
- Fixed point
  - Do everything in thousandths (or millionths, etc.)
  - Not always easy to pick the right units
  - Different scaling factors for different stages of computation
- Scientific notation
  - Exponential notation allows HUGE dynamic range
  - Constant (approximately) relative precision across the whole range

# Lots of Ways to do Floating Point

Decimal: $2.99792458 \times 10^8$

Hexadecimal: $1.1de784a \times 16^7$

Binary: $1.0001110111100111100001001010 \times 2^{28}$

Wilder alternatives

Arbitrary precision arithmetic

- Software support for arbitrary number of digits (or bits)
- Powerful, but almost always slow

Represent numbers by their logarithms

- Used for centuries in slide rules
- Makes multiplication and division really fast and easy
- But addition and subtraction become quite painful

# Floating Point Before IEEE-754 Standard

## Late 1970s formats

- About two dozen different, incompatible floating point number formats
- Decimal, binary, octal, hexadecimal all in use
- Precisions from about 4 to about 17 decimal digits
- Ranges from about $10^{19}$ to $10^{322}$

## Sloppy arithmetic

- Last few bits were often wrong
- Overflow sometimes detected, sometimes ignored
- Arbitrary, almost random rounding modes
  - Truncate, round up, round to nearest
- Addition and multiplication not necessarily commutative
  - Small differences due to roundoff errors

# IEEE Floating Point

- Standard set by IEEE
  - John Palmer at Intel took the lead in 1976 for a good standard
  - First working implementation:  Intel 8087 floating point coprocessor, 1980
  - Full formal adoption:  1985
  - Updated in 2008
- Rigorous specification for high accuracy computation
  - Made every bit count
  - Dependable accuracy even in the lowest bits
  - Predictable, reasonable behavior for exceptional conditions
    - (divide by zero, overflow, etc.)

# IEEE Floating Point Format (single precision)

Sign bit: (0 is positive, 1 is negative)

Significand: (also called the *mantissa*; stores the 23 most significant bits after the decimal point)

Exponent: used biased base 127 encoding

- Add 127 to the value of the exponent to encode:
- $-127 \rightarrow 00000000$ $\quad$ $1 \rightarrow 10000000$
- $-126 \rightarrow 00000001$ $\quad$ $2 \rightarrow 10000001$
- … $\quad$ …
- $0 \rightarrow 01111111$ $\quad$ $128 \rightarrow 11111111$

How do you represent zero ? Special convention:

- Exponent: -127 (all zeroes ), Significand 0 (all zeroes), Sign + or -

# Floating Point Representation

$10.625_{10} \Rightarrow 1010.101_2$

$\color{red}{1}.010101 \times 2^{3}$

This must be a 1!
So don't store it.

| +/- | Significand (1010101) |
|---|---|
| 1 bit | 23 bits |

| Exponent (3) |
|---|
| 8 bits |

# Floating Point Representation

$$10.625_{10} \Rightarrow 1010.101_{2}$$

$$\textcolor{red}{1}.010101 \times 2^{3}$$

This must be a 1!
So don't store it.

| +/- | Significand (1010101) | Exponent (3) |
|---|---|---|
| 1 bit | 23 bits | 8 bits |

$$10.625_{10} = 0 \quad 10000010 \quad 01010100000000000000000$$

# Class Problem 1

What is the value (in decimal) of the following IEEE 754 floating point encoded number?

| | | |
|---|---|---|
| **1** | **10000101** | **01011001000000000000000** |

# Floating Point Multiplication

- Add exponents (don’t forget to account for the bias)
- Multiply significands (don’t forget the implicit 1 bits)
- Renormalize if necessary
- Compute sign bit (simple exclusive-or)

# Floating Point Multiply

$$10.625_{10} = 1010.101_2$$

$$10_{10} = 1010_2$$

| | | |
|---|---|---|
| 0 | 10000010 | 01010100000000000000000 |
| | + | × |
| 0 | 10000010 | 01000000000000000000000 |
| | -127 | |
| 0 | 10000101 | 10101001000000000000000 |

$$
\begin{array}{r}
1\,0\,1\,0\,1\,0\,1 \\
\times \qquad 1\,0\,1 \\
\hline
1\,0\,1\,0\,1\,0\,1 \\
1\,0\,1\,0\,1\,0\,1\,0\,0 \\
\hline
1\,1\,0\,1\,0\,1\,0\,0\,1
\end{array}
$$

$$1101010.01_2$$

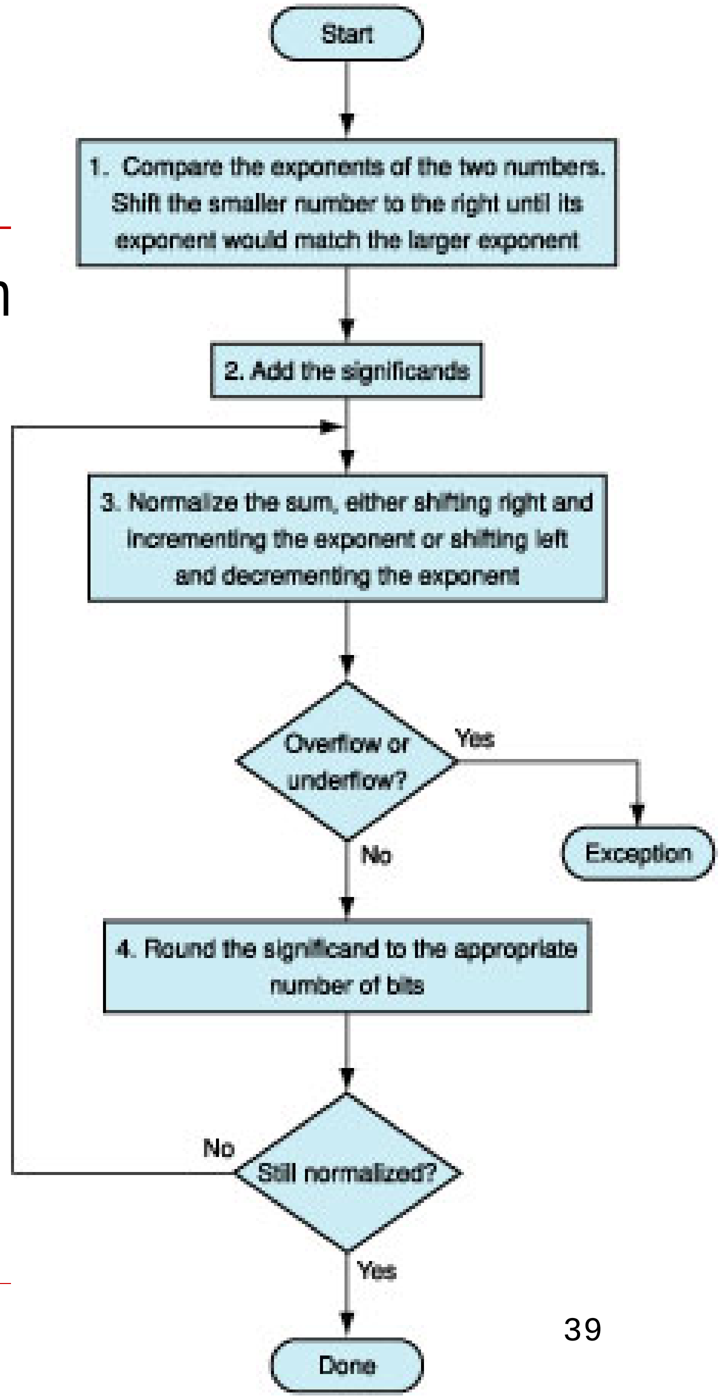
$$= 106.25_{10}$$

# Floating Point Addition

- More complicated than floating point multiplication!
- If exponents are unequal, must shift the significand of the smaller number to the right to align the corresponding place values
- Once numbers are aligned, simple addition (could be subtraction, if one of the numbers is negative)
- Renormalize (which could be messy if the numbers had opposite signs; for example, consider addition of +1.5000 and – 1.4999)
- Added complication:  rounding to the correct number of bits to store could denormalize the number, and require one more step

# Floating Point Addition

1. Shift smaller exponent number significan right to match larger.
1. Add significands.
2. Normalize and update exponent.
3. Check for “out of range”.



# Class Problem 2

Show how to add the following 2 numbers using IEEE floating point addition: 100.125 + 13.75

# Class Problem 2

101.125 | 0 | 10000101 | 10010100100000000000000

13.75 | 0 | 10000010 | 10111000000000000000000

Shift by 6-3 = 3

Shift mantissa by difference in exponent

00110111000000000000000

Note: When shifting to the right, the first shift should put the implicit 1, then 0's

Sum Mantissa's

```
  1 0 0 1 0 1 0 0 1
+ 0 0 1 1 0 1 1 1 0
-------------------
  1 1 0 0 1 0 1 1 1
```

Sum didn't overflow, so no re-normalization needed

0 | 10000101 | 11001011100000000000000

= 114.875

# Class Problem 3

**Show how to add the following 2 numbers using IEEE floating point addition: 117.125 + 13.75**

# Class Problem 3

117.125 | 0 | 10000101 | 11010100100000000000000

13.75 | 0 | 10000010 | 10111000000000000000000

Shift by 6-3 = 3

Shift mantissa by difference in exponent

00110111000000000000000

Note: When shifting to the right, the first shift should put the implicit 1, then 0's

Sum Mantissa's

```
  1 1 0 1 0 1 0 0 1
+ 0 0 1 1 0 1 1 1 0
-------------------
1 0 0 0 0 1 0 1 1 1
```

0 | 10000110 | 00000101110000000000000

= 114.875

Sum overflows, re-normalize by adding one to exponent and shifting mantissa by one

# More precision and range

We have described IEEE-754 binary32 floating point format, commonly known as “single precision” (“float” in C/C++)

24 bits precision; equivalent to about 7 decimal digits

$3.4 * 10^{38}$ maximum value

Good enough for most but not all calculations

IEEE-754 also defines a larger binary64 format, “double precision” (“double” in C/C++)

53 bits precision, equivalent to about 16 decimal digits

$1.8 * 10^{308}$ maximum value

Most accurate physical values currently known only to about 47 bits precision, about 14 decimal digits