7. Linker and Floating Point Representation

EECS 370 – Introduction to Computer Organization – Winter 2023

EECS Department
University of Michigan in Ann Arbor, USA

Announcements—Reminders

- **Project 1.a due today!**

- Project 1.s and 1.m due Thursday 2/2

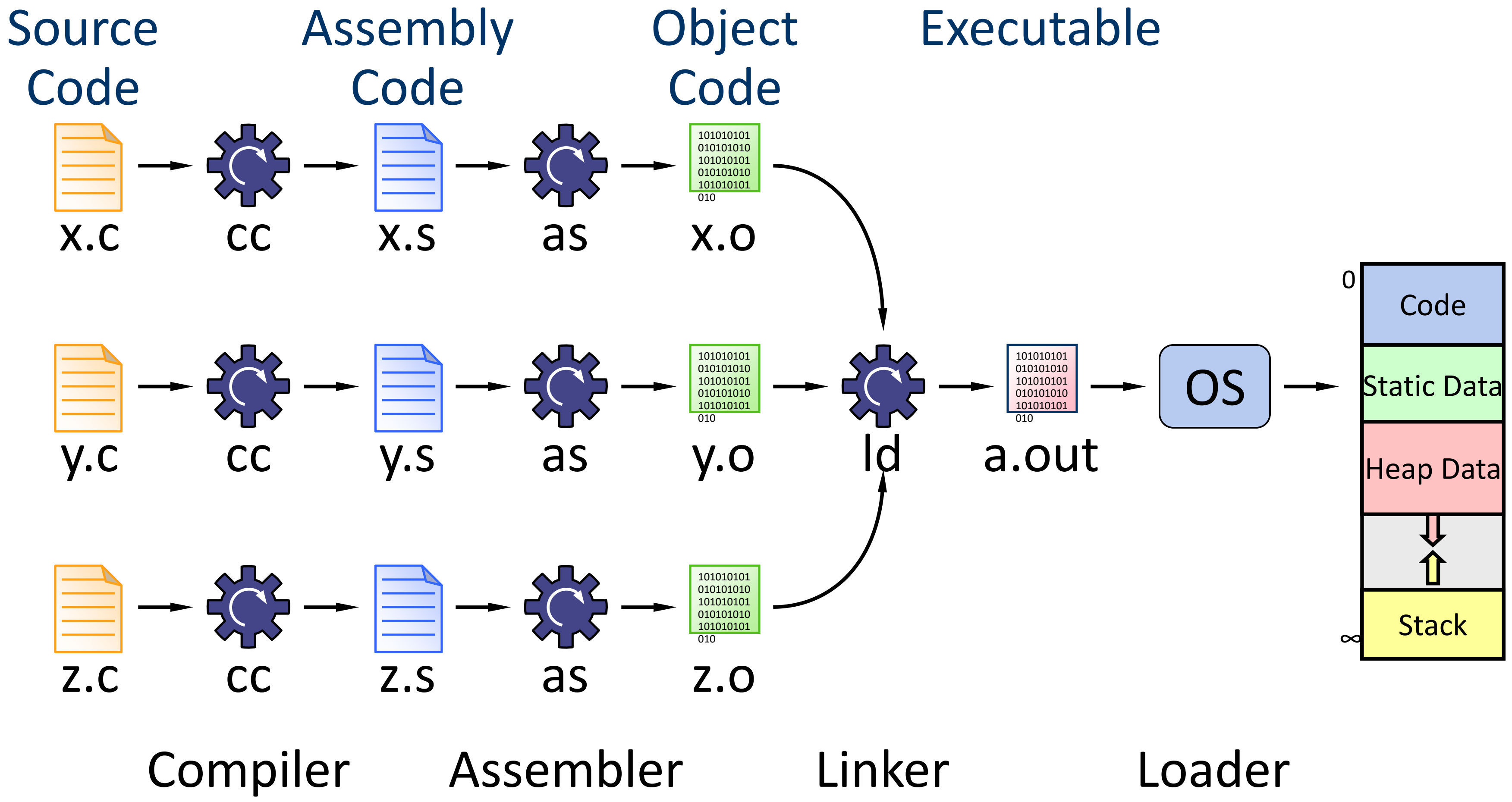
- Homework 2 due Monday 2/6
  - Group and individual (turned in separately)
  - For group part, if you don’t have a team, consider using the “search for teammates” message on Piazza.
    - Or talk to someone in this room.

- All due dates on calendar on web page.

Instruction Set Architecture (ISA) Design Lectures

- Lecture 2: Storage types
- Lecture 3 : Addressing modes and LC2K
- Lecture 4 : ARM Assembly
- Lecture 5 : C to Assembly
- Lecture 6 : Function calls
- **Lecture 7: Linker and Floating Point**

Source to Process Translation



Linux (ELF—executable and linkable format) object file format

Object files contain more than just machine code instructions!

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

Text: machine code

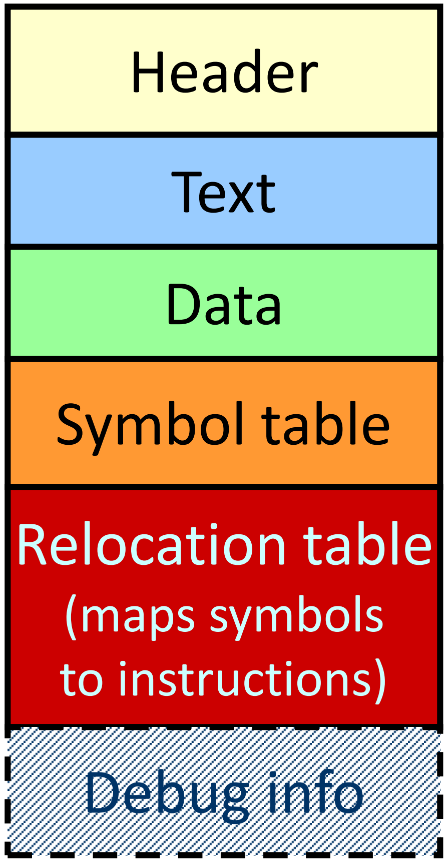
Data: global and static data

Symbol table: symbols and values

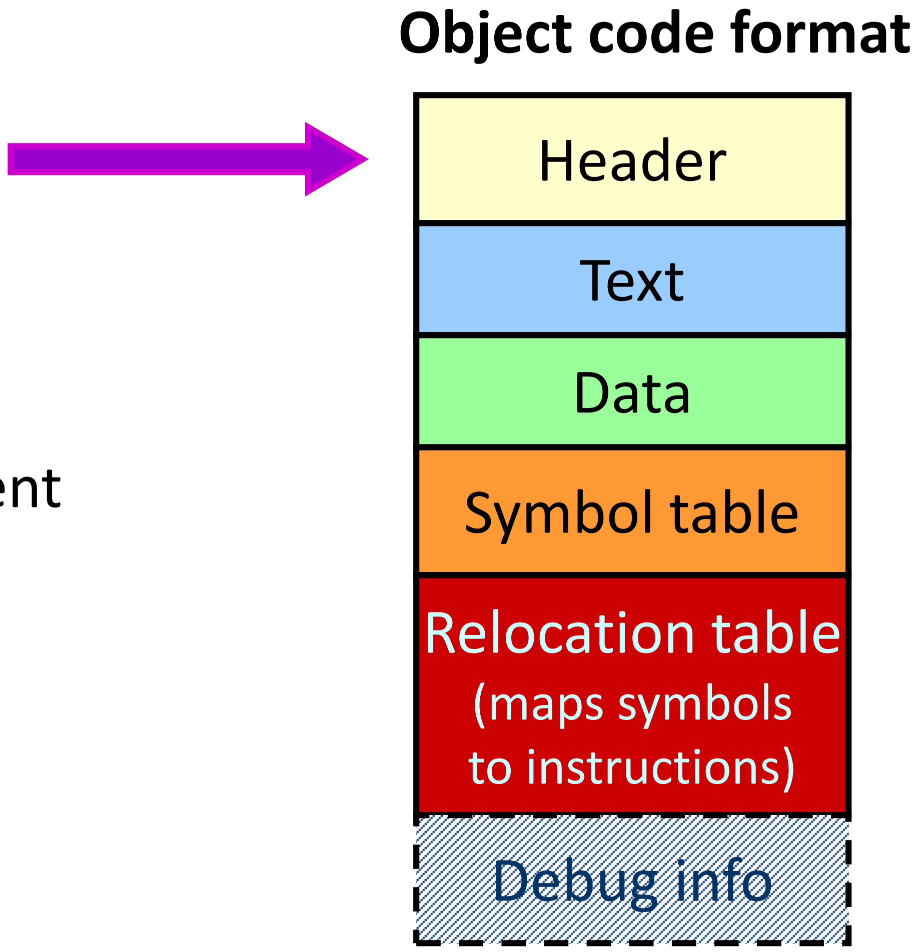
Relocation table: references to addresses that may change

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)



Header

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

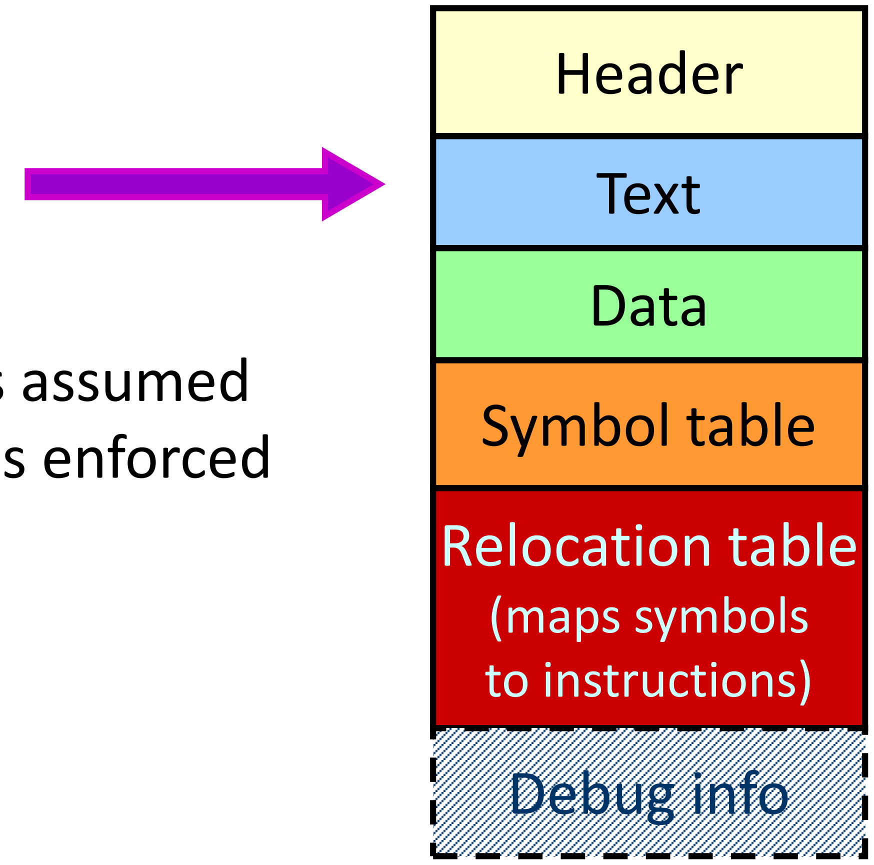
Linux (ELF) object file format (3)

Text segment

- machine code

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

Object code format



Linux (ELF) object file format (4)

Data segment (Initialized static segment)

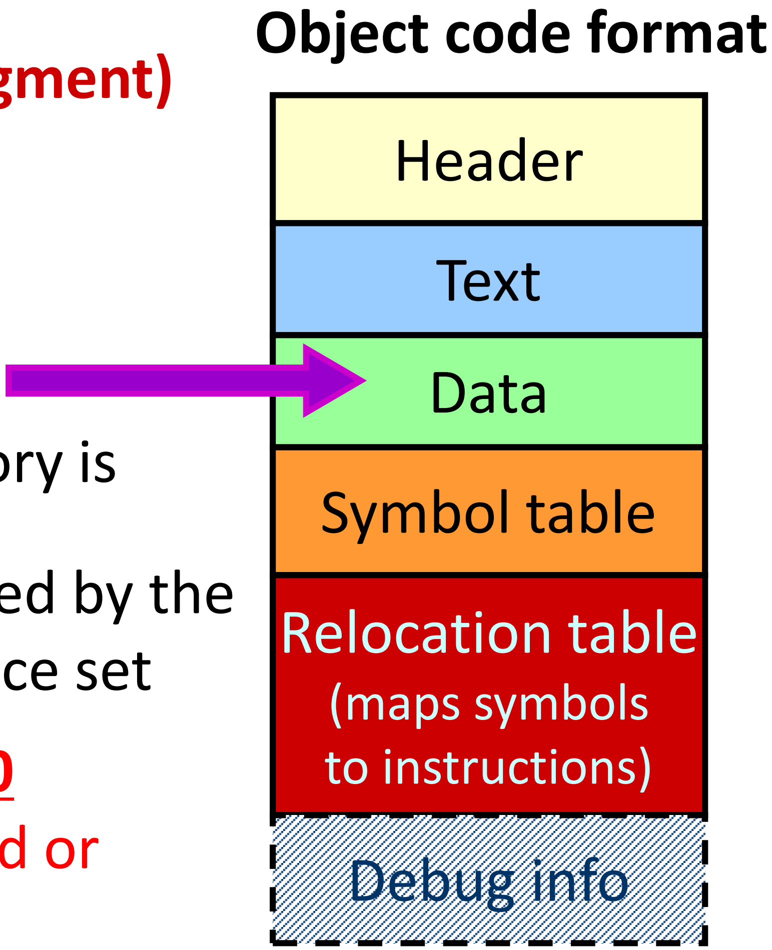
- values of initialized globals
- values of initialized static locals

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

This goes in its own space allocated by the loader called the **bss**—basic service set

Simplifying Assumption for EECS370

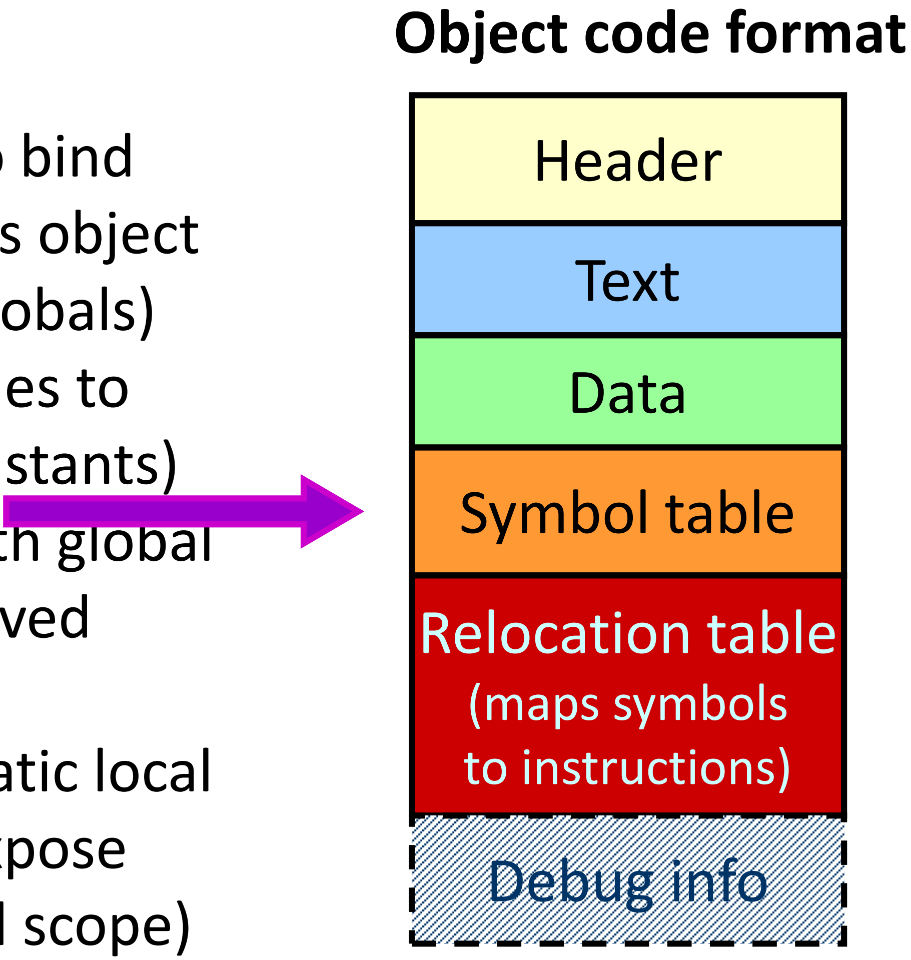
All globals and static locals (initialized or not) go in the data segment



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
- Includes addresses of static local variables, but doesn't expose them to other files (local scope)

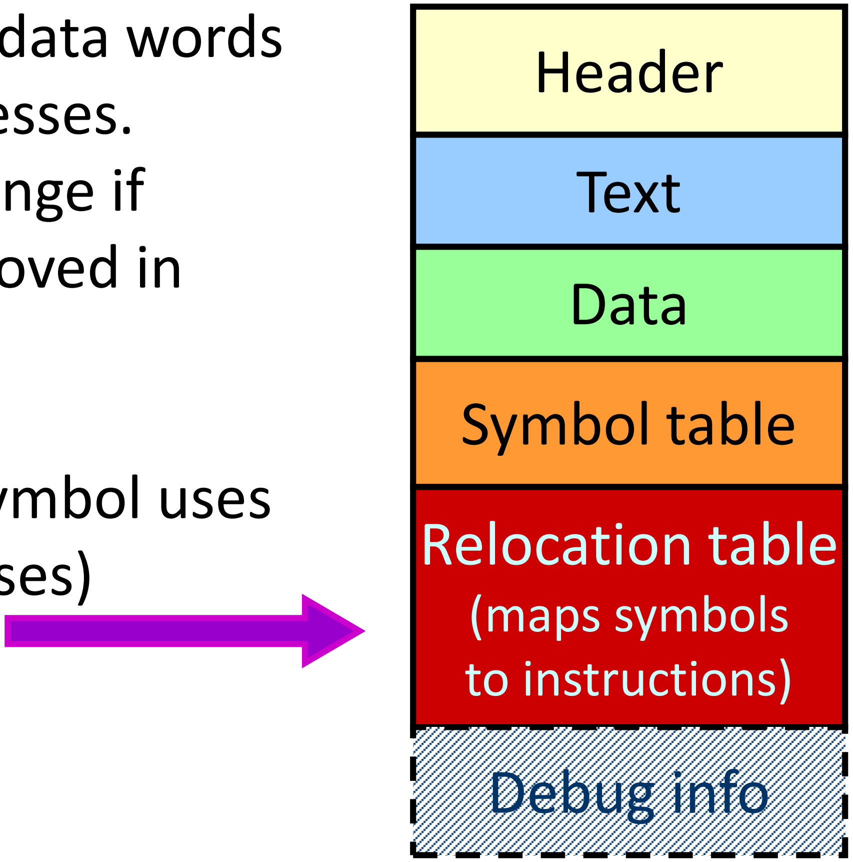


Linux (ELF) object file format (6)

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)

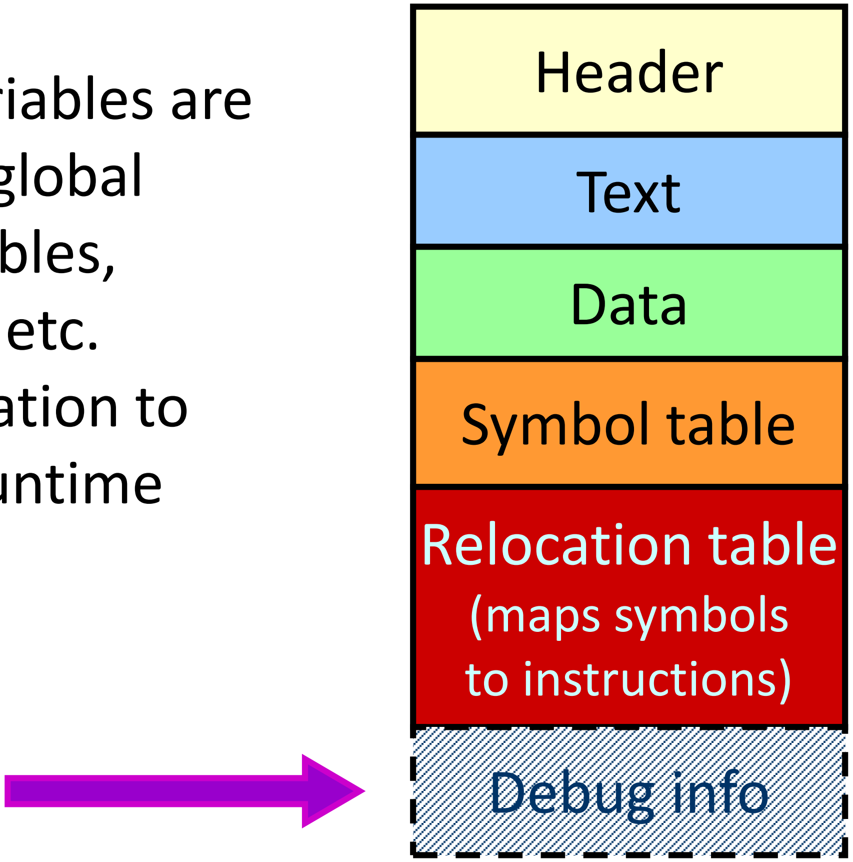
Object code format



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

Object code format



Assembly → Object file - example

Snippet of C

```
int X = 3;
main() {
  Y = X + 1;
  B();
  …
```

Snippet of assembly code

```
LDUR      X1, [X27, #0]
ADDI      X9, X1, #1
BL        B
```

Header	Name	foo	
	Text size	0x0C //probably bigger	
	Data size	0x04 //probably bigger	
Text	Address	Instruction	
	0	LDUR X1, [X27, #0] //X27 global reg	
	4	ADDI X9, X1, #1 //X9 local variable Y	
	8	BL B	
Data	0	X	3
	…		
Symbol table	Label	Address	
	X	0	
	B	-	
	main	0	
Reloc table	Addr	Instruction type	Dependency
	0	LDUR	X
	8	BL	B

Linker, or Link Editor

- 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 - continued

- Determine the 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 relocate
    - Absolute Address (mov 27, #X): always relocate
    - External Reference (usually bl): always relocate
    - Data Reference (often movz/movk): always relocate

- Executable file contains <u>no relocation info or symbol table</u> they are just used by assembler/linker

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

Class Problem 1

- In the following files, which symbols will be put in the object file's symbol table? If its location is defined in this file, indicate if it would be in the data or text section.

Recall local variables are not in tables:
b in file 1 and
*e in file 2

```
file1.c
extern void bar(int);
extern char c[];
int a;
int foo (int x) {
   int b;
   a = c[3] + 1;
   bar(x);
   b = 27;
}
```

file 1 – symbol table

sym	**loc**
a	data
foo	text
c	-
bar	-

```
file2.c
extern int a;
char c[100];
void bar (int y) {
   char e[100];
   a = y;
   c[20] = e[7];
}
```

file 2 – symbol table

sym	**loc**
c	data
bar	text
a	-

Class Problem 2

Which lines / instructions are in the relocation table for each file?

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  }
```

file 1 - relocation table

line	type	dep
6	ldur	c
6	stur	a
7	bl	bar

file2.c

```
1  extern int a;
2  char c[100];
3  void bar (int y) {
4     char e[100];
5     a = y;
6     c[20] = e[7];
7  }
```

file 2 - relocation table

line	type	dep
5	stur	a
6	stur	c

Note: in a real relocation table, the "line" would really be the address in "text" section of the instruction we need to update.

Some additional questions

```
file1.c
extern void bar(int);
extern char c[];
int a;
int foo (int x) {
    int b;
    a = c[3] + 1;
    bar(x);
    b = 27;
}
```

```
file2.c
extern int a;
char c[100];
void bar (int y) {
   char e[100];
   a = y;
   c[20] = e[7];
}
```

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

A smart compiler would not put j in symbol table. A dumb one would. It's a benign issue, no harm done if j is in symbol table uselessly.

B) What if variable 'e' is static?

It is in the data section (no longer stack) and any reference to it is in relocation table. It is also in the symbol table so the address can be calculated, but it will be flagged not to be exported to other files, since the scope is local.

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

You should get a compile error

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 straight forward, 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

Things to remember

- Compiler converts a single source code file into a single assembly language file
- Assembler handles directives (.fill), 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: Thus unchanged files, including libraries need not be recompiled.
- Linker resolves remaining addresses.
- Loader loads executable into memory and begins execution

Floating Point Arithmetic

Why floating point

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

Floating point before IEEE-754 standard

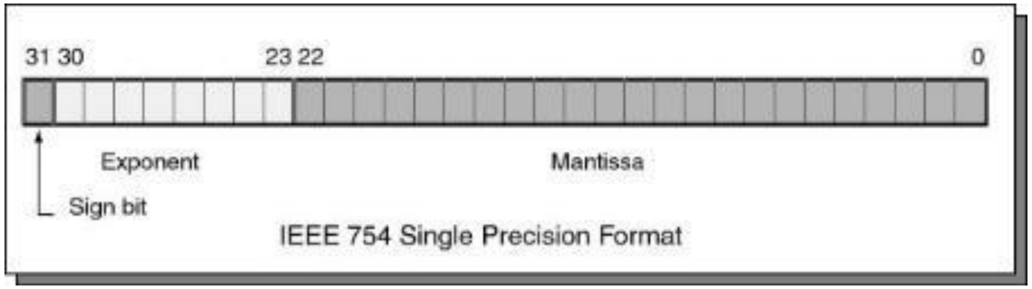
- Late 1970s formats
  - About two dozen different, incompatible floating point number formats
  - Precisions from about 4 to about 17 decimal digits
  - Ranges from about 10^{19} to 10^{322}
- Sloppy arithmetic
  - Last few bits were often wrong, and in different ways
  - Overflow sometimes detected, 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

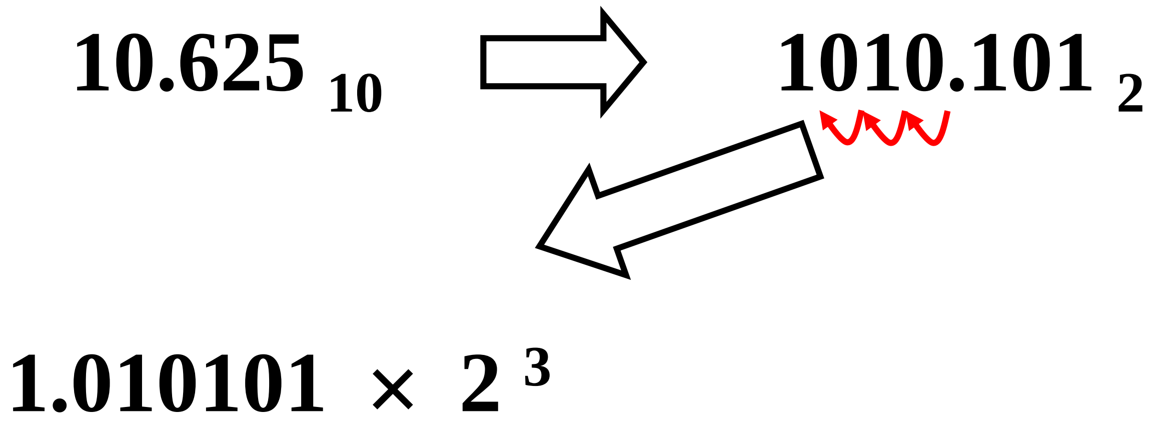
- Standard set by IEEE
  - Intel took the lead in 1976 for a good standard
  - First working implementation: Intel 8087 floating point coprocessor, 1980
  - Full formal adoption: 1985
  - Updated in 2008
- Rigorous specification for high accuracy computation
  - Made every bit count
  - Dependable accuracy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

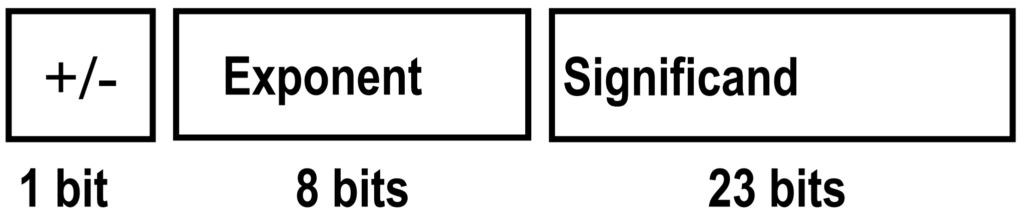
IEEE Floating point format (single precision)

- 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 1 $\rightarrow$ 10000000
  - -126 $\rightarrow$ 00000001 2 $\rightarrow$ 10000001
  - … …
  - 0 $\rightarrow$ 01111111 128 $\rightarrow$ 11111111
- How do you represent zero ? Special convention:
  - Exponent: -127 (all zeroes), Significand 0 (all zeroes), Sign + or -



Floating Point Representation





Floating Point Representation

$10.625_{10} \Rightarrow 1010.101_2$

1.010101×2^3

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

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

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

Class Problem

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

1	**10000101**	**01011001000000000000000**

Consider numbers in *scientific* notation

- Consider a representation of the form $\pm$A.AA x 10 $^{\pm B}$ where the value in front of the decimal must be non-zero.
  - E.g. 7.68 x 10^2 but not 0.10 x 10^2
- What is the largest number you can represent?
  - At that value, what is the gap between representations?
- What is the smallest number you can represent that is greater than 0?
  - At that value, what is the gap between representations?

What matters to a CS person?

- When you are adding small numbers to big numbers, the result may not change
  - E.g. $1.00 \times 10^3 + 1.00 \times 10^1 = 1000+1=1.00\mathrm{x}10^3$.
- This can be a real problem when writing scientific code.
  - For the above example, imagine you did that addition a million times.
    - You'd still have 1000 when the answer should be 1,001,000.
    - That's a problem.
- So you need to be aware of the issue.
  - This is why most people use "double" instead of "float"
  - The problem can still exist, it's just less likely.

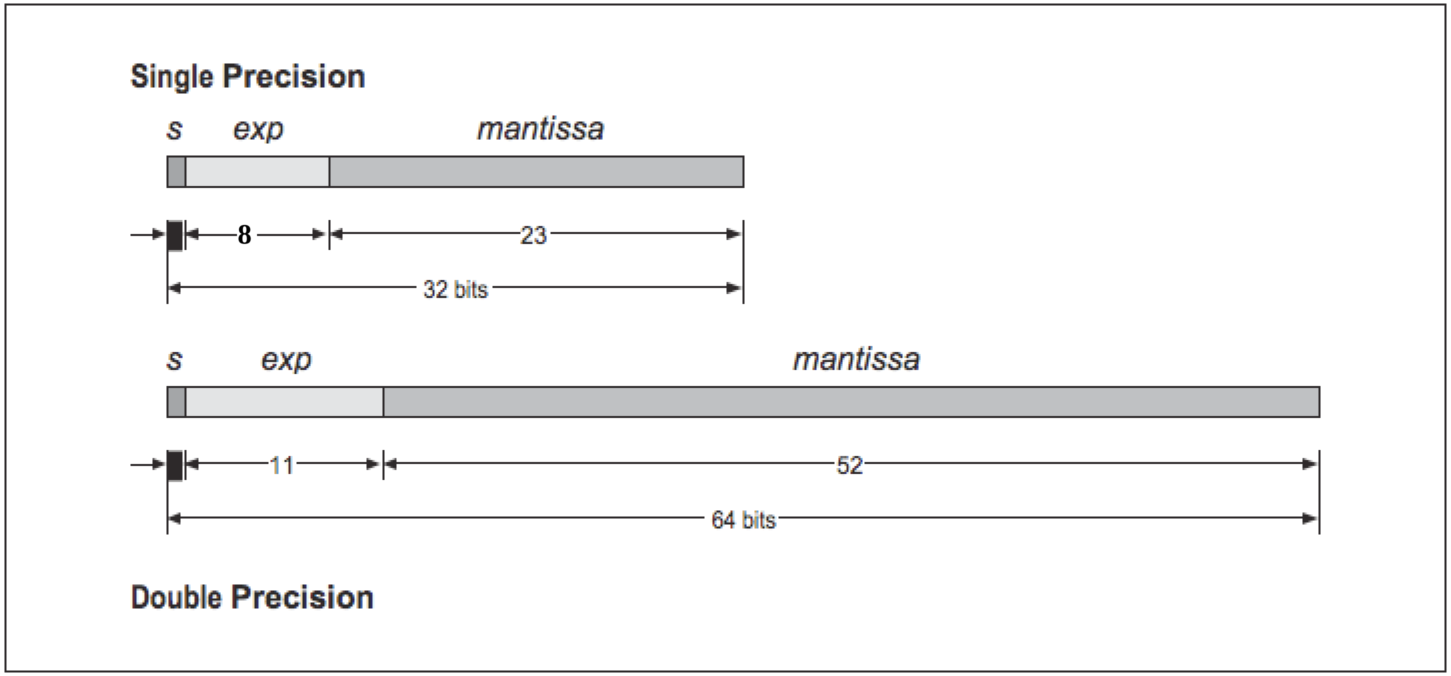
More precision and range

- 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

Extreme floating point

- binary128, “quad precision”; recent addition to standard:
  - 113 bits precision, about 34 decimal digits
  - $1.2*10^{4932}$ maximum value
  - Very rarely used, but some computations require extreme accuracy to limit cumulative roundoff error
- Another recent addition was binary16, “half precision”
  - 11 bits precision, about 3.3 decimal digits
  - 65504 maximum value
  - Used in graphics processors for accurate rendering of scenes with a large dynamic range in lighting levels.
  - Minimizes storage per pixel

Single (“float”) percision



Next up: Hardware!

Software done!

- We've covered the software side of things (L2-L7)
  - Two ISAs, programming in assembly, linkers, loaders, floating point, etc.

Next:

- The basics of digital logic (L8, L9)
  - Gates, flip-flops, state machines, memory
- How to implement a processor
  - Single-cycle (L10), multi-cycle (L11) and pipelined (L12-L15)
- Caches and memory systems (L17-24)
- Hot topics in computers (between L21 and L22)

Not testable

Bonus slides – this material is not testable

- This material is here for those folks that may care.
  - **You *may* find it useful when considering the gap between representations**
  - But the material isn't directly testable.
- It is interesting if you are into that kind of thing.
- It can be useful if you are going to do scientific programming for a living.
- So it is provided as a reference, but isn't part of the class (we may cover a bit of it in lecture if we have time)

Floating point multiplication

Not testable

- Add exponents (don't forget to account for the bias of 127)
- Multiply significands (don't forget the implicit **1** bits)
- Renormalize if necessary
- Compute sign bit (simple exclusive-or)

Not testable

Floating point multiply

$10.625_{10} = 1010.101_2 \Rightarrow$ 0 | 10000010 | 01010100000000000000000

$+ \qquad \times$

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

$= 106.25_{10}$

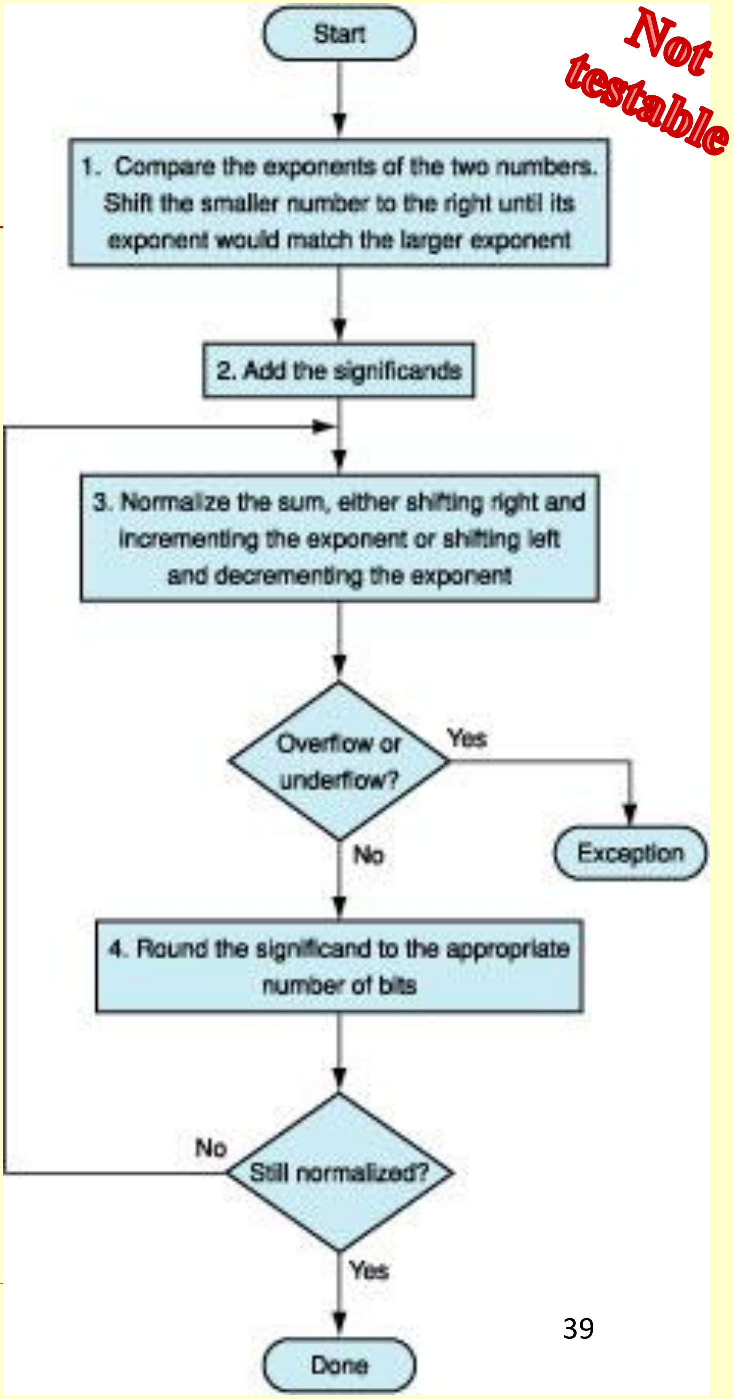
Not testable

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 right to match larger.
2. Add significands
3. Normalize and update exponent
4. Check for "out of range"



Not testable
Start
1. Compare the exponents of the two numbers. Shift the smaller number to the right until its exponent would match the larger exponent
2. Add the significands
3. Normalize the sum, either shifting right and incrementing the exponent or shifting left and decrementing the exponent
Overflow or underflow?
Yes
Exception
No
4. Round the significand to the appropriate number of bits
No
Still normalized?
Yes
Done

Not testable

Class Problem

Show how to add the following 2 numbers using IEEE floating point addition: 101.125 + 13.75

Not testable

Class Problem

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 Significands

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

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

0 | 10000101 | 11100101110000000000000

= 114.875

Not testable

Class Problem

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

Class Problem

Not testable

117.125 $\boxed{0}\ \boxed{10000101}\ \boxed{11010100100000000000000}$

13.75 $\boxed{0}\ \boxed{10000010}\ \boxed{10111000000000000000000}$

Shift by 6-3 = 3

Shift mantissa by difference in exponent

$\boxed{00110111000000000000000}$

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

Sum Significands

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

$\boxed{0}\ \boxed{10000110}\ \boxed{00000101110000000000000}$

= 130.875

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