L7_1 Linux-ELF

EECS 370 – Introduction to Computer Organization – Fall 2020

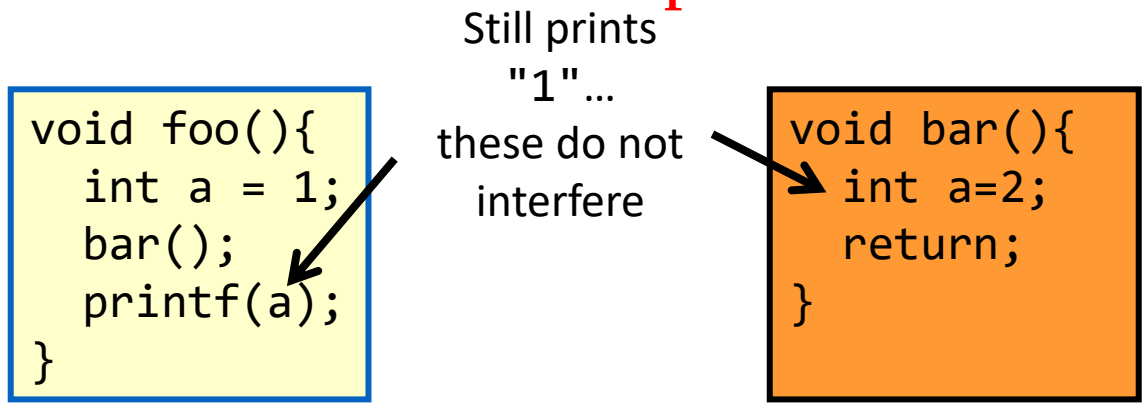
Learning Objectives

- To be able to identify the components of a Linux binary (assembled machine code) files.
- To understand the mapping of data and instructions to machine code files, including object files and executables.

Variable Scope – C/C++

Review Caller-Callee

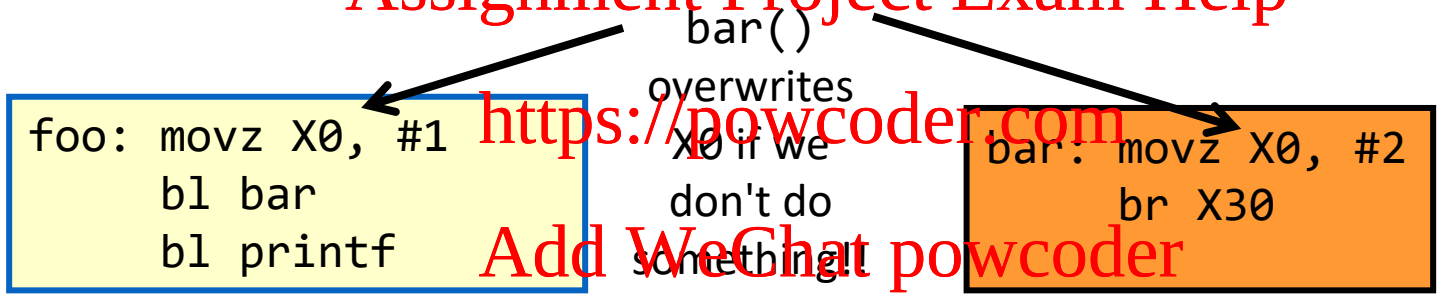
- Higher level languages (like C/C++) provide many abstractions that don't exist at the assembly level
- E.g. in C, each function has its own local variables
  - Even if different function have local variables with the same name, they are independent and guaranteed not to interfere with each other!



Saving / Restoring Registers

Review Caller-Callee

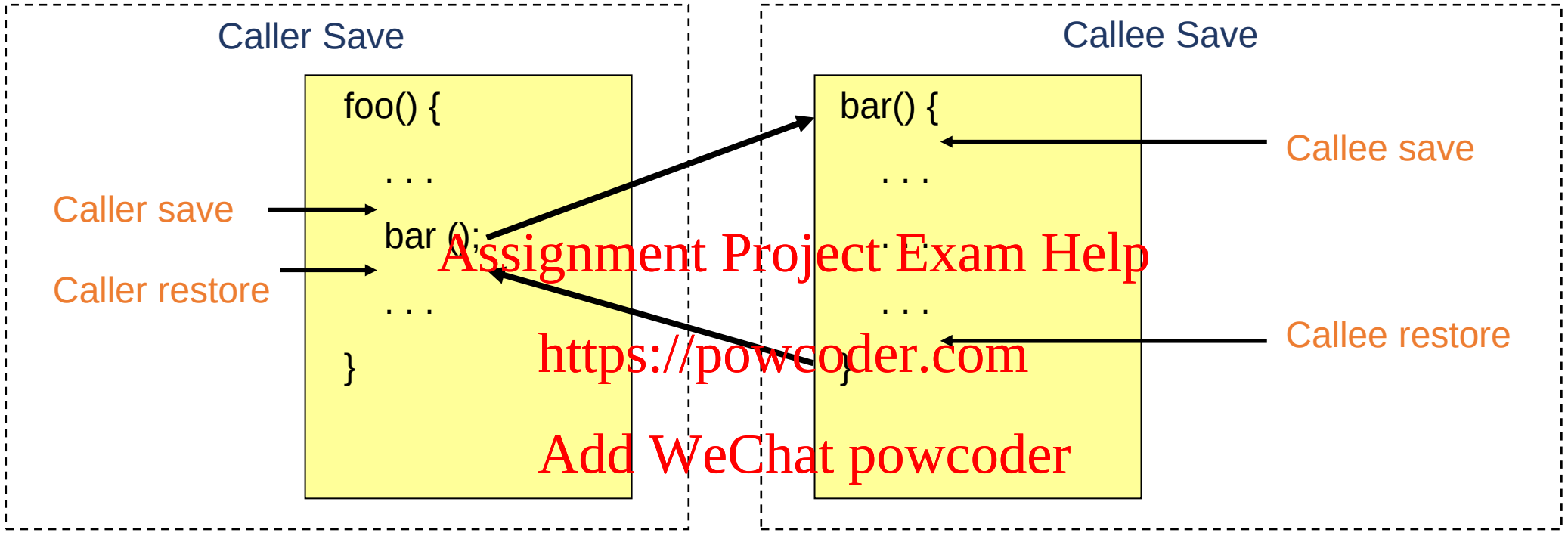
- But in assembly, all functions **share** a small set (e.g. 32) of registers
  - Called functions will overwrite registers needed by calling functions



- "Someone" needs to save/restore values when a function is called to ensure this doesn't happen
  - Convention: implementation scheme detailing design choices to be followed by everyone

Caller-Callee Save/Restore

Review Caller-Callee



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

Callee save registers: Must be the same value as when called. May do this by either not changing the value in a register ***or*** by inserting saves at the start of the function and restores at the end

Caller-Callee Convention

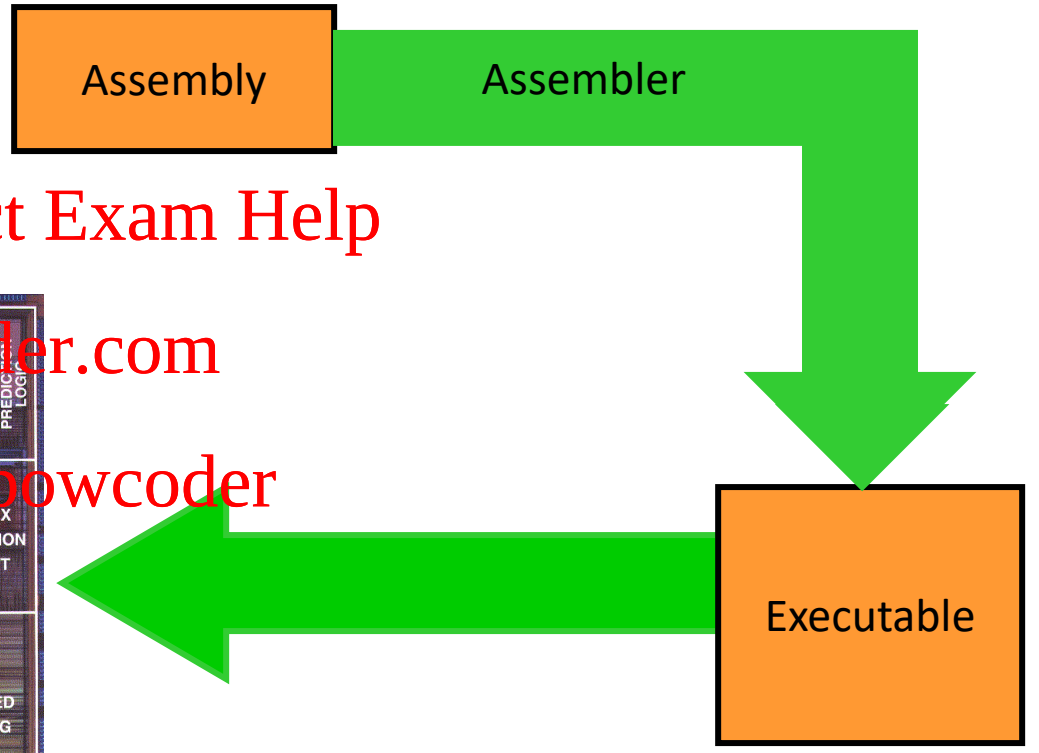
Review Caller-Callee

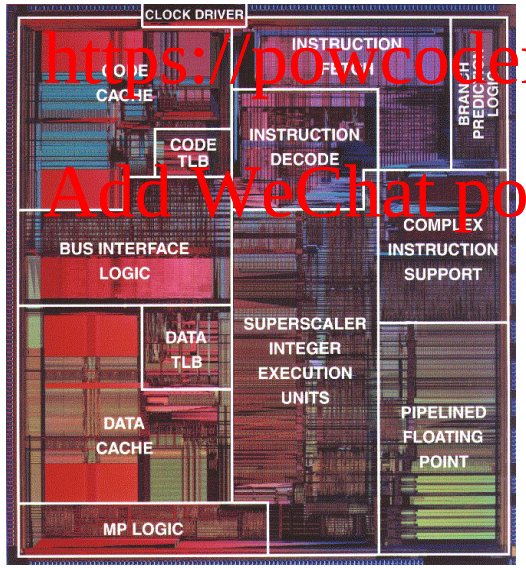
- This is probably in the top #3 for concepts 370 students have difficulty "getting"
  - But once it "clicks", it is really not that complicated!
  - Spend some time on your own thinking through it
  - Watch the supplemental video we have online
    - https://www.youtube.com/watch?v=SMH5uL3HjiU
  - Come to office hours to chat about it

Source Code to Execution

Linux ELF

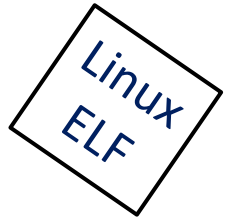
- In project 1a, our view is this:





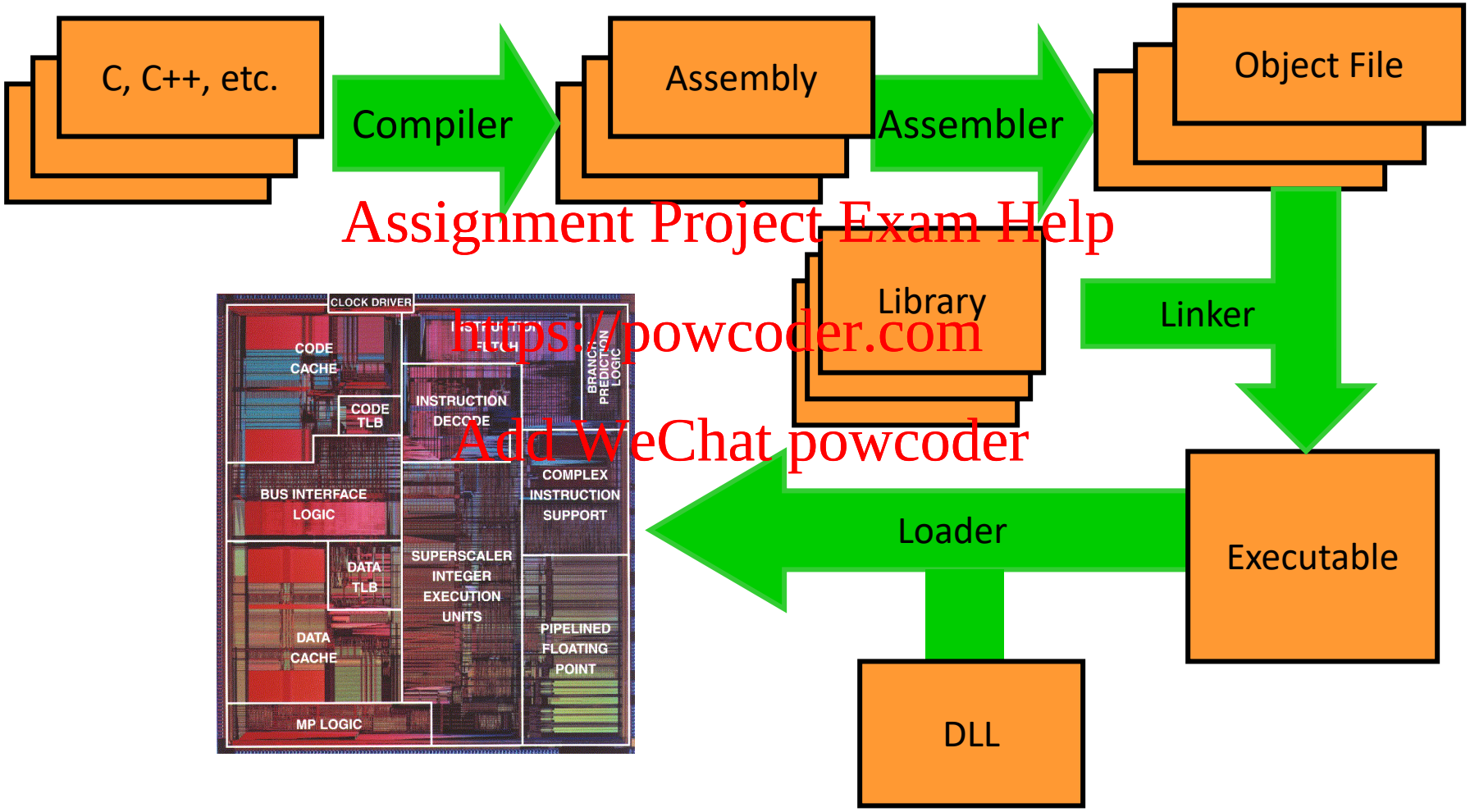
Not very accurate… why? Because it reality, we have multiple files

Multi-file programs

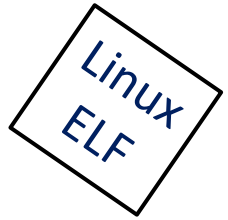


- In practice, programs are made from thousands or millions of lines of code
- If we change one line, do we need to recompile the whole thing?
  - No! If we compile each file into a separate **object file**, then we only need to recompile that one file and **link** it to the other, unchanged object files

Source Code to Execution



What Happens When You Invoke `gcc`?

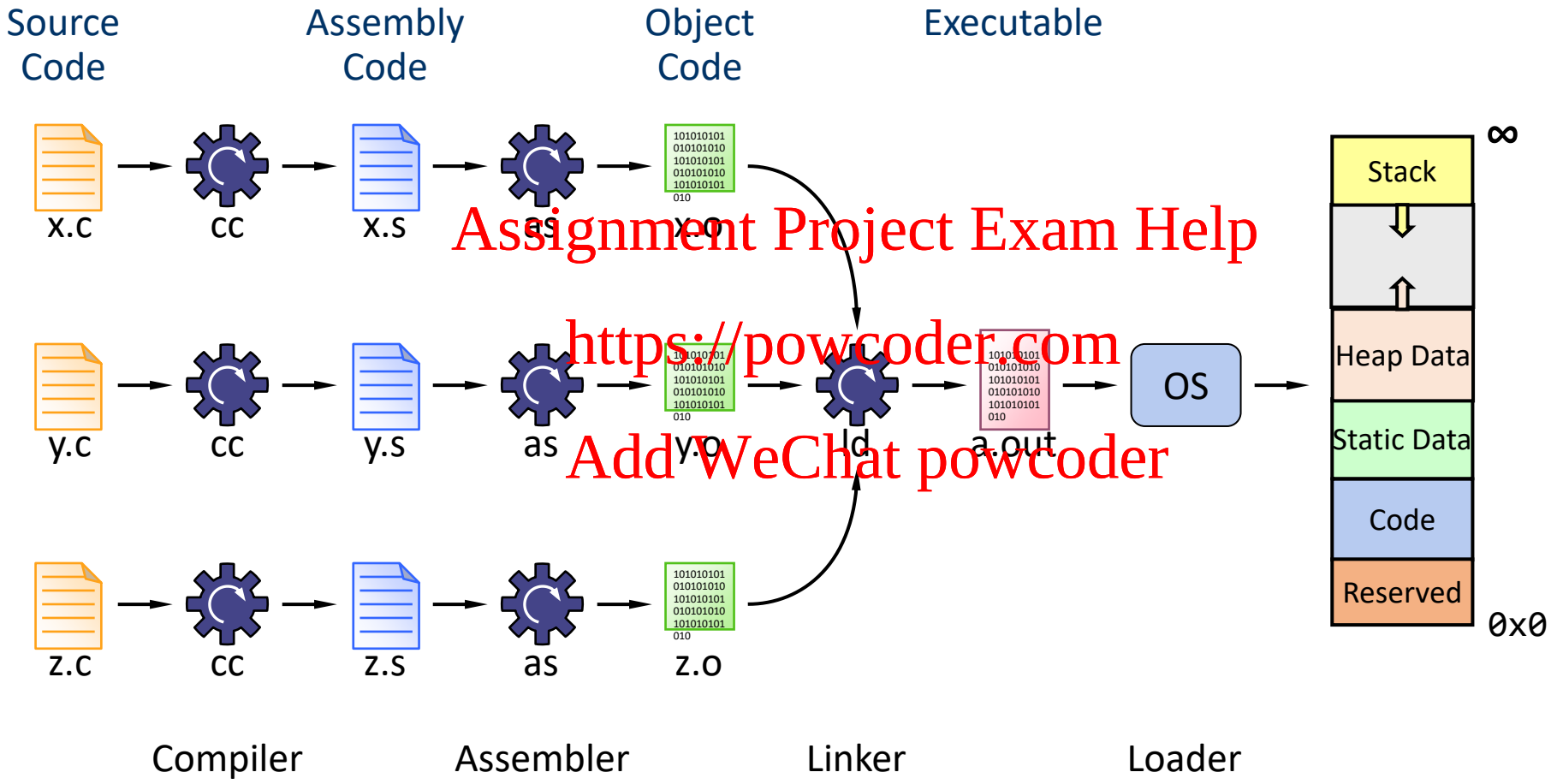


1. C preprocessor
   - Handles macros, `#define`, `#ifdef`, `#if`
   - `gcc -E foo.c > foo.i` (foo.i contains preprocessed source code)
2. Compiler
   - `gcc -S foo.c` (foo.s contains textual assembly)
3. Assembler
   - `as foo.s -o foo.o` OR `gcc -c foo.s`
4. Linker
   - `ld foo.o bar.o bunch_of_other_stuff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

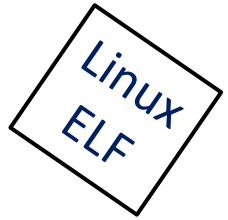
You can run `gcc -v` to see all the commands that it is running
- Note gcc does not call `ld`, it calls `collect2`, which is a wrapper that calls `ld`

Source to Process Translation

Linux ELF



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

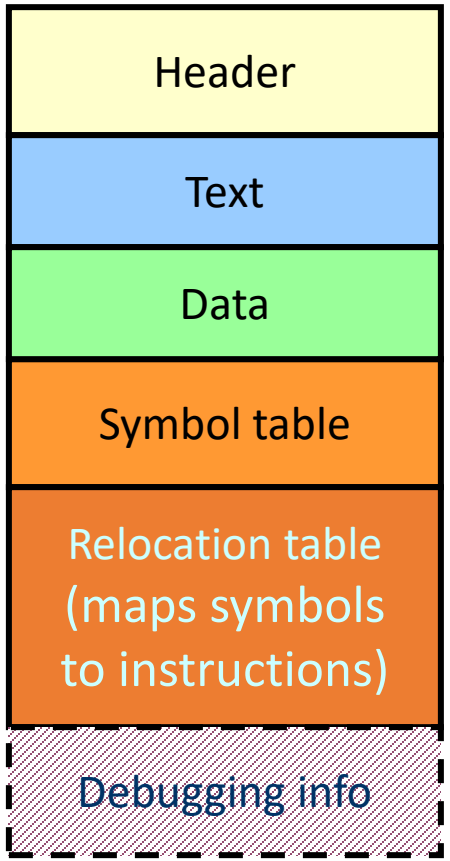
Data: global and static data

Symbol table: symbols and values

Relocation table: references to addresses that may change when application is loaded

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

Object code format

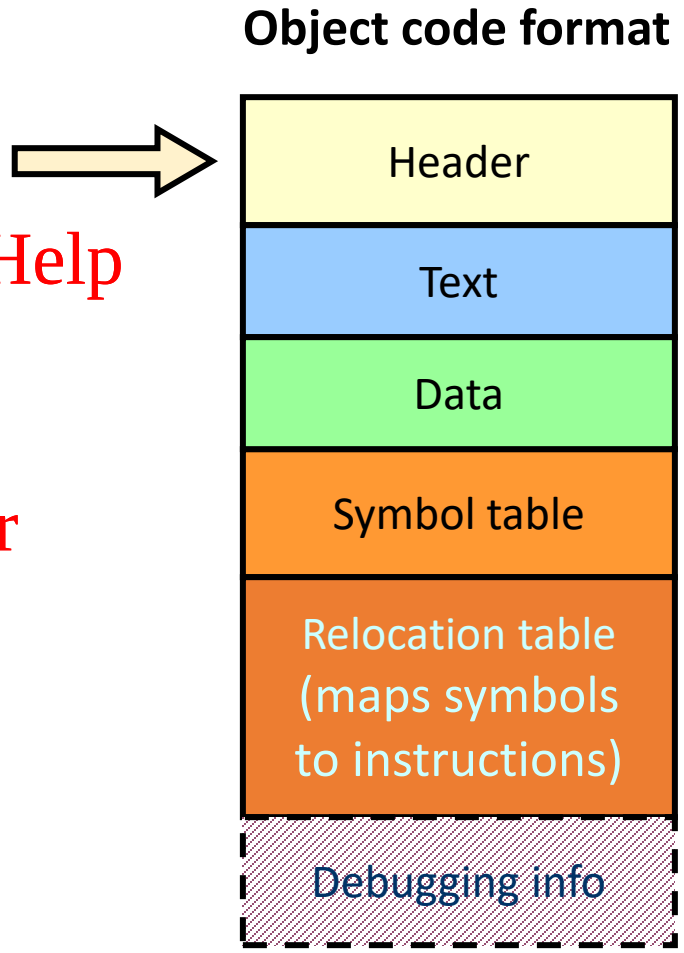


Linux (ELF) Object File Format- Header

Linux ELF

Header

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



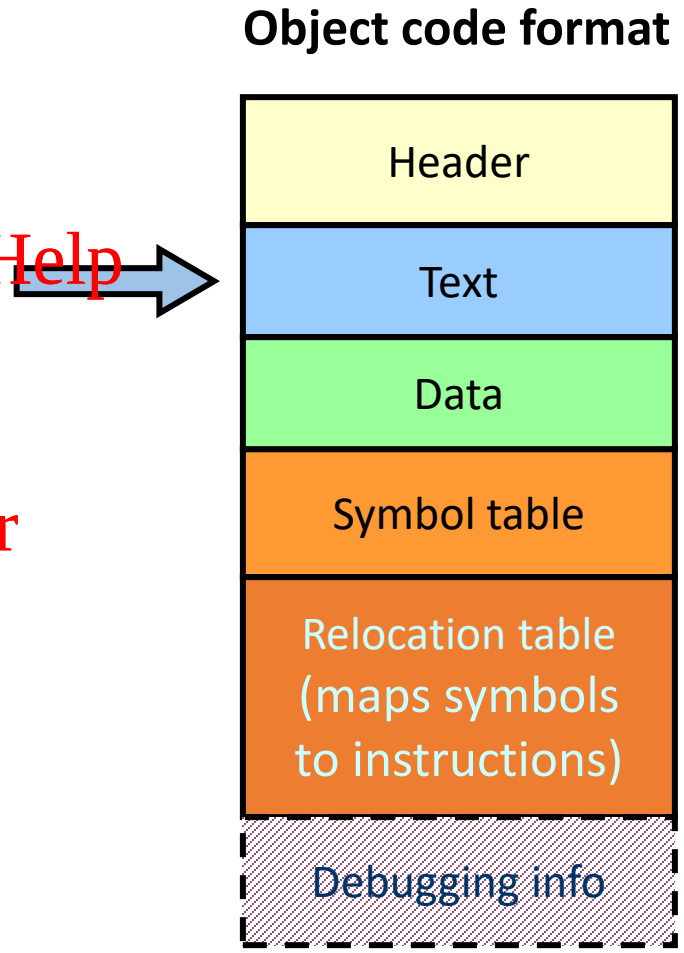
Linux (ELF) Object File Format- Text

Linux ELF

Text segment

- machine code
  i.e., executable code statements

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



Linux (ELF) Object File Format- Data

Linux ELF

Data segment (Initialized static segment)

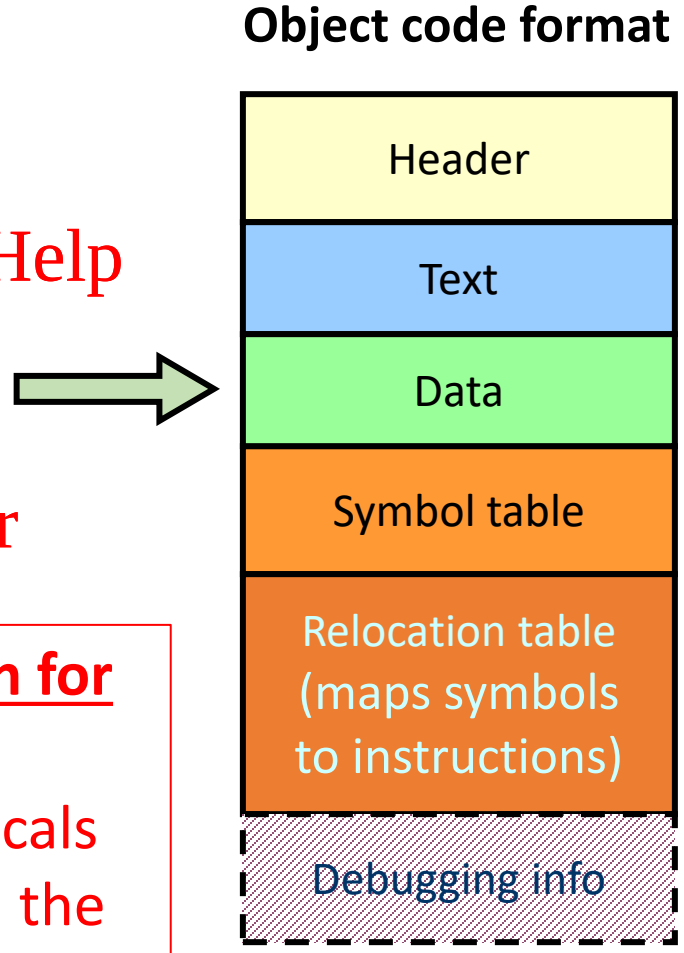
- values of initialized globals
- values of initialized static locals

Does not contain uninitialized data.
Just keep track of how much memory is needed for uninitialized data

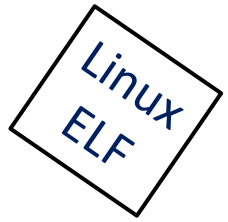
This goes in its own space allocated by the loader called the **bss**—basic service set

Simplifying Assumption for EECS370

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

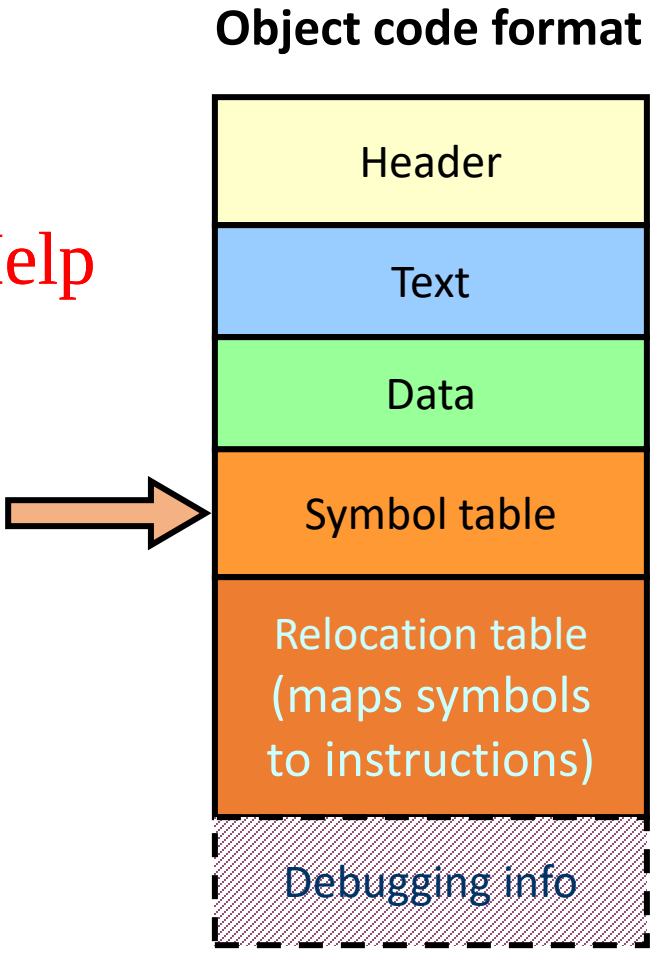


Linux (ELF) Object File Format- Symbol Table



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 does not expose them to other files (local scope)



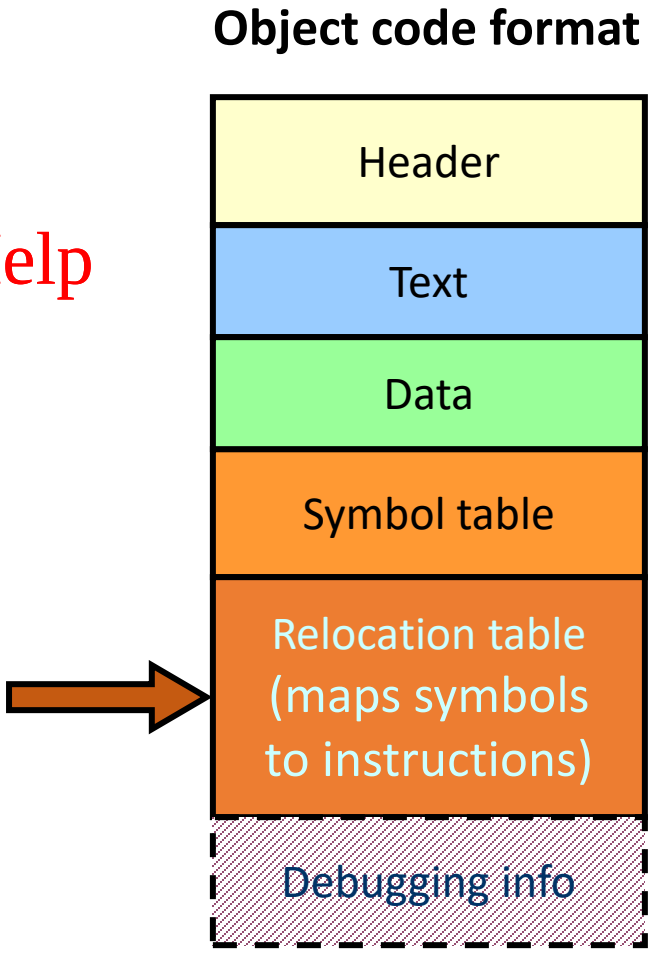
Linux (ELF) Object File Format- Relocation Table

Linux ELF

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)

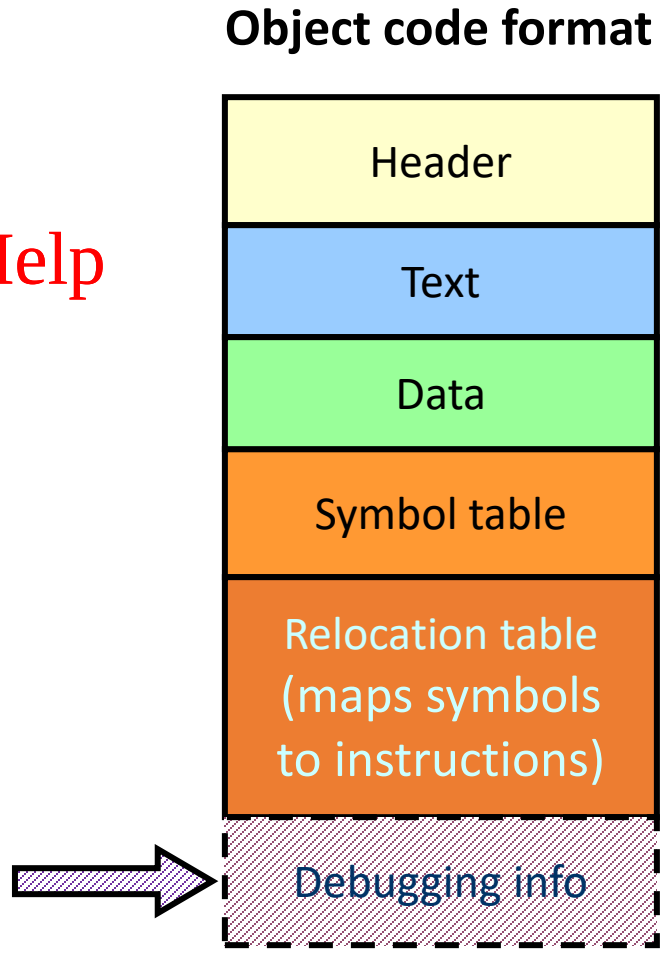


Linux (ELF) Object File Format- Debugging Info

Linux ELF

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 - Example

Linux ELF

Snippet of C

```
int x = 3;
main() {
  int y;
  y = x + 1;
  B();
  // more code
```

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

Logistics

- There are 3 videos for lecture 7
  - L7_1 – Linux-ELF
  - L7_2 – Linker
  - L7_3 – IEEE_Floating-Point
- There is one worksheet for lecture 7
  1. Linker and loader – wait until after L7_2

L7_2 Linker

EECS 370 – Introduction to Computer Organization – Fall 2020

Learning Objectives

- Describe operations for the linking and loading of object files (binary representations of programs intended to be directly executed on a processor).
- Describe symbol and relocation tables and contents for source code files.

Linker, or Link Editor

Linker

- 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, jal): always relocate
    - External Reference (usually bl): always relocate
    - Data Reference (often movz/movk): always relocate

- Executable file contains no relocation info or symbol table these just used by assembler/linker

Symbol Table – Example

Linker

Problem: Which symbols will be put into the symbol table? *i.e., which "things" should be visible to all files?*

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;
}
```

file1.c – symbol table

symbol	location

file2.c

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

file2.c – symbol table

symbol	location

Symbol Table – Example

Linker

Problem: Which symbols will be put into the symbol table? *i.e., which "things" should be visible to all files?*

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;
}
```

file1.c – symbol table

symbol	location
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];
}
```

file2.c – symbol table

symbol	location
c	data
bar	text
a	-

Local variables are not in tables:

- `b` in `file1.c`
- `*e` in `file2.c`

Relocation Table – Example

Linker

Problem: Which lines/instructions are in the relocation table? *i.e., which "things" need to be updated after linking?*

file3.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  }
```

file3.c - relocation table

line	type	dep

file4.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  }
```

file4.c - relocation table

line	type	dep

Relocation Table – Example

Linker

Problem: Which lines/instructions are in the relocation table? *i.e., which "things" need to be updated after linking?*

file3.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  }
```

file3.c - relocation table

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

file4.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  }
```

file4.c - 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 assembly instruction we need to update.

Loader

Linker

- 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 (starting address of program is random and may be anywhere in memory - ASLR)
  - Initializes registers (PC and SP most important)
- Loading is now complex
  - Dynamically linked libraries (DLLs on Windows, SOs on Linux)
    - Linking when program loaded, one copy of library in memory shared by all running applications
  - Some systems even delay some code optimization (usually a compiler job) to load time
  - Position Independent Code (PIC), Procedure Linkage Table (PLT), Global Offset Table (GOT)
  - Loaders must deal with sophisticated operating systems

Things to Remember

Linker

- 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

Logistics

- There are 3 videos for lecture 7
  - L7_1 – Linux-ELF
  - L7_2 – Linker
  - L7_3 – IEEE_Floating-Point
- There is one worksheet for lecture 7
  1. Linker and loader – you can do this now.

L7_3 IEEE_Floating-Point

EECS 370 – Introduction to Computer Organization – Fall 2020

Learning Objectives

- Ability to describe the representation and encoding used for real numbers.

Why Floating Point

Floating Point

- Need to represent real numbers
- Rational numbers *(can be represented by dividing two integers, e.g., 1/3)*
  - Ok, but can be cumbersome to work with
  - Falls apart for sqrt(2) and other irrational numbers
- Fixed point *(fixed number of digits before/after decimal 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! *(mantissa and exponent, e.g., 3 x 10^4)*
  - Exponential notation allows HUGE dynamic range
  - Constant (approximately) relative precision across the whole range

Floating Point Pre-Standardization

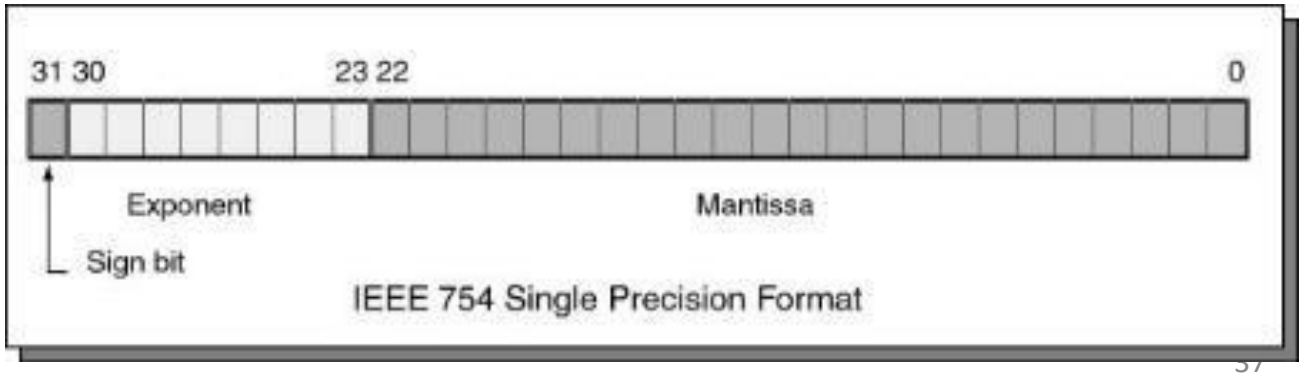
- Late 1970s formats
  - About two dozen different, incompatible floating point number formats
  - Precisions from about 4 to about 17 decimal digits
  - Ranges from about 10^{19} to 10^{322}
- Sloppy arithmetic
  - Last few bits were often wrong, and in different ways
  - Overflow sometimes detected, 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 754 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 → 00000000 1 → 10000000
  - -126 → 00000001 2 → 10000001
  - … …
  - 0 → 01111111 128 → 11111111
- How do you represent zero ? Special convention:
  - Exponent: -127 (all zeroes), Significand 0 (all zeroes), Sign + or -

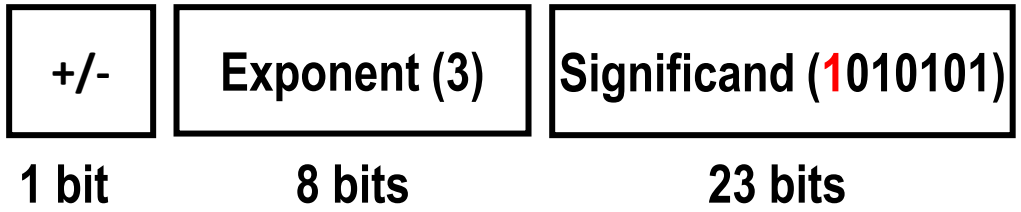


Floating Point Representation

10.625_{10} ⇒ 1010.101_{2}

1.010101×2^{3}

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

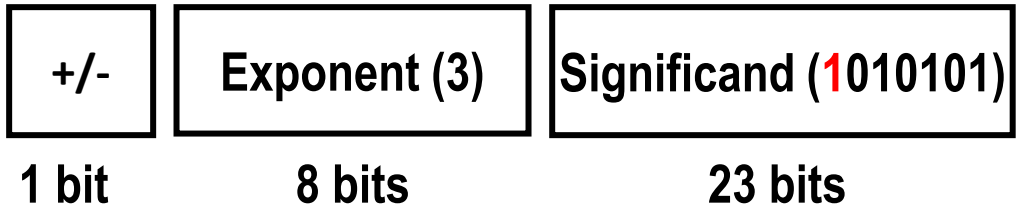


Floating Point Representation

10.625_{10} ⇒ 1010.101_{2}

1.010101×2^{3}

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



$10.625_{10} = 0\ 10000010\ 01010100000000000000000_{2}$

Floating Point - Example

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

1	10000101	01011001000000000000000

Floating Point - Example

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

1	10000101	01011001000000000000000

Floating Point - Example

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

1	10000101	01011001000000000000000

sign bit	1	- (negative)

exponent	10000101	133 – 127 = 6 (biased by 127)

significand	01011001000000000000000	add implicit 1

-1.01011001×2^6	shift radix point 6 places	-1010110.01

$-1010110.01 = -(2^6 + 2^4 + 2^2 + 2^1 + 2^{-2}) = -(64 + 16 + 4 + 2 + ¼) = -86.25_{10}$

Logistics

- There are 3 videos for lecture 7
  - L7_1 – Linux-ELF
  - L7_2 – Linker
  - L7_3 – IEEE_Floating-Point
- There is one worksheet for lecture 7
  1. Linker – do if you have not already.