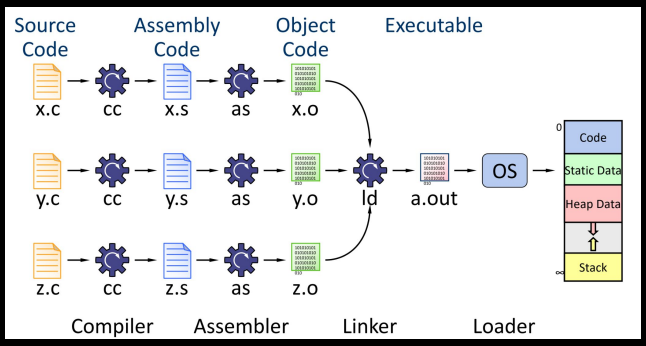
EECS 370

Lab 4:
Project 2 - Linking & Functions

Linking

How Source Code Becomes an Executable



Why Do We Compile Code This Way?

We want to split code across multiple files

- Makes code more reusable
- Reduces compile time (only compile what we need)
- Allows for standard libraries usable by all

The EECS 370 Makefile takes advantage of this - checks any obj files that need to be remade, and only reassembles those before linking.

How Do We Compile Code This Way?

We have to figure out how code can all fit together

- Make sure that code can branch to any function
- Be able to reference Globals / Statics that move around in memory
- Have the linked file to look similar to individual files

Memory Mapping: Data Can Be Stored in Many Different Locations

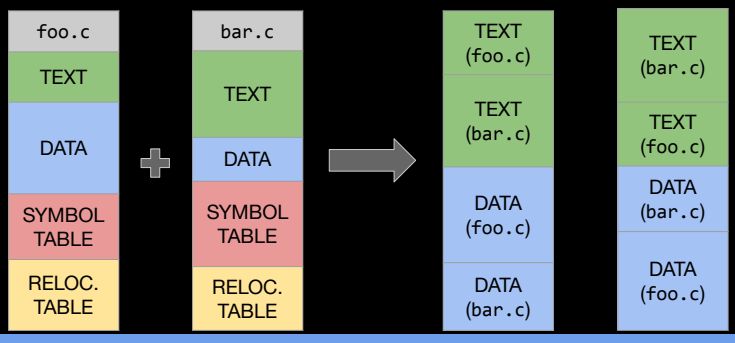
cheat sheet!

Data location	What is stored there	When this data is initialized
Text	Instructions	Compile-time
Static/Data	Globals, Static variables	Compile-time
Heap	Dynamic Memory (anything from `malloc()`)	Runtime
Stack	Local variables, some function parameters	Runtime

Executable and Linkable Format (ELF) Layout

Header	Metadata like the size of each section
Text	Machine code / Instructions
Data	Global variables, Static data (In reality, multiple sections)
Symbol Table	Mapping of symbols to address and references to symbols
Relocation Table	References to any address that may shift addresses
Debug Info	Source code to binary matching (line nums, var. names, etc.)

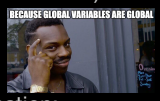
How Linking the Code Produces an Executable



The Symbol Table Keeps Track of Globals / Statics

The **Symbol Table** tracks the locations of many things, including:

- Global variables and exported functions (global functions)
- Unresolved variables / functions (references to globals in other files)
- Static variables (Things in the data section)



Each **Symbol Table Entry** contains three key pieces of information:

Label (Variable Name)	Type (Text / Data / Unknown)	Address (Section and Offset)

The Relocation Table Tracks Changes in Address

When linking files, sections in the code move around

The **Relocation Table** tracks what *instructions* need to be updated

These include:

- Calls to functions, both global and local
- References to anything in the Data section

A **Relocation Table Entry** contains the following three things:

Address of Instruction	Instr./Dir. Type (lw, sw, .fill)	Referenced Symbol

Functions

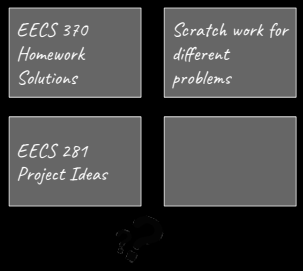
Analogy: Working on Whiteboards in a Classroom

Imagine you are working in a classroom with 4 whiteboards

You have written important things on some of the whiteboards

Then, you need to take a lunch break, but are worried about your work getting erased

...What do you do?



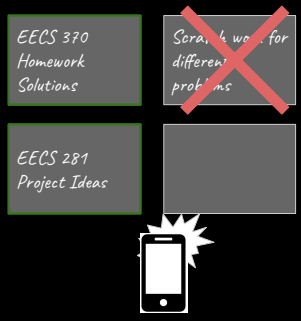
Option A: Take Pictures of Your Whiteboards

You take out your phone and snap a picture of the important boards

Since you don't care about the scratch work, don't take a picture

Now, after coming back, you can rewrite things from your phone

This is similar to a ***caller save***

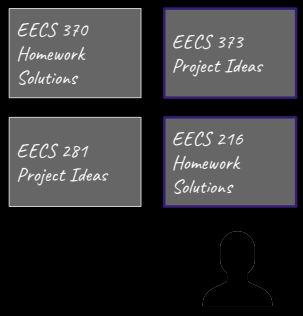


Option B: Assume Nobody Will Change It

In a perfect world, anyone who used the room while you were gone would save your work and put everything back how it was

They don't have to fix blank boards or boards they didn't change

This is similar to a ***callee save***



Making the Comparison to Caller / Callee Saves

Caller-Saved Register

Similar to Option A, where you take responsibility for your boards

Calling function is responsible for saving registers

Only need to save a register if it is ***live*** (needed after returning)

Callee-Saved Register

Similar to Option B, where anyone in the room must save your work

Called function is responsible for saving registers

Only need to save a register if it is ***overwritten*** (changed)

LC2K ABI

r0: 0 (Global)
r1: Argument #1 (Typically caller, can be Callee - up to you.)
r2: Argument #2 (Typically caller, can be Callee - up to you.)
r3: Return value of function (Caller save. Why?)
r4: Scratch register (Caller or Callee - Up to you**)
r5: Stack offset pointer (Global)
r6: Scratch register (Caller or Callee - Up to you**)
r7: Return address (Depends on how you interpret jalr. Either works.)

**You need 1 non-return caller saved register for a junk jalr regB target.

Functions in LC2K

Convert this function to LC2K using the LC2K ABI, with the following parameters:

First, with reg 1 caller-saved

Then, with reg 1 callee-saved

Assume register 1 maps to a, and r6 is caller-save for jalr's junk value

```
void start() {
    int a = 1;
    a += mystery(a);
}
```

```
Start   //callee stores
        lw   0   1   one //init a
        //caller stores
        lw   0   6   fcall
        jalr     6   7
        //caller loads
        add  1   3   1
        //callee loads
        jalr     7   6
one     .fill    1
fcall   .fill    Mystery
```


Example: Caller-Saves in LC2K

Convert this function to LC2K using the LC2K ABI, with the following parameters:

Reg 1 is caller-saved

Assume register 1 maps to a, and r6 is caller-save for jalr's junk value

```
void start() {
    int a = 1;
    a += mystery(a);
}
```

```
Start   lw   0   1   one  //init a
        lw   0   6   one
        sw   5   7   Stack
        add  5   6   5    //sp++
        sw   5   1   Stack
        add  5   6   5    //sp++
        lw   0   6   fcall
        jalr     6   7
        lw   0   6   neg1
        add  5   6   5    //sp--
        lw   5   1   Stack
        add  5   6   5    //sp--
        lw   5   7   Stack
        add  1   3   1
        jalr     7   6
one     .fill    1
fcall   .fill    Mystery
neg1    .fill    -1
```


Example: Callee-Saves in LC2K

Convert this function to LC2K using the LC2K ABI, with the following parameters:

Reg 1 is callee-saved

Assume register 1 maps to a, and r6 is caller-save for jalr's junk value

```
void start() {
    int a = 1;
    a += mystery(a);
}
```

```
Start   lw   0   6   one
        sw   5   7   Stack
        add  5   6   5    //sp++
        sw   5   1   Stack
        add  5   6   5    //sp++
        lw   0   1   one  //init a
        lw   0   6   fcall
        jalr     6   7
        add  1   3   1
        lw   0   6   neg1
        add  5   6   5    //sp--
        lw   5   1   Stack
        add  5   6   5    //sp--
        lw   5   7   Stack
        jalr     7   6
one     .fill    1
fcall   .fill    Mystery
neg1    .fill    -1
```