EECS 370 - Lecture 7

Linking

Announcements

- P1
  - Project 1 s + m due Thu
  - Instructor assembler available on the AG
- HW 1
  - Due Monday (9/22)
- Lab 4 meets Fr/M
  - Pre-Lab 4 quiz due Thursday
- Get exam conflicts sent to us **ASAP**
  - Forms listed on Ed

Instruction Set Architecture (ISA) Design Lectures

- Lecture 2: ISA - storage types, binary and addressing modes
- Lecture 3 : LC2K
- Lecture 4 : ARM
- Lecture 5 : Converting C to assembly – basic blocks
- Lecture 6 : Converting C to assembly – functions
- **Lecture 7 : Translation software; libraries, memory layout**

Problem 3: Reusing registers

- 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!

```
int main(){
  int a=1;
  foo();
  printf(a);
}
```

Still prints "1"... these don't interfere

```
void foo(){
  int a=2;
  return;
}
```

What about registers?

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

```
main: movz X0, #1
      bl foo
      bl printf
```

foo() overwrites
X0 if we don't
do something!!

```
foo: movz X0, #2
     br X30
```

- "Someone" needs to save/restore values when a function is called to ensure this doesn't happen

Two Possible Solutions

- Either the **called** function saves register values before it overwrites them and restores them before the function returns (**callee** saved)…

```
main: movz X0, #1
      bl foo
      bl printf
```

```
foo:  stur X0, [stack]
      movz X0, #2
      ldur X0, [stack]
      br X30
```

- Or the **calling** function saves register values before the function call and restores them after the function call (**caller** saved)…

```
main: movz X0, #1
      stur X0, [stack]
      bl foo
      ldur X0, [stack]
      bl printf
```

```
foo:  movz X0, #2
      br X30
```

a->r1, b->r2, etc.

Another example

Original C Code

```
void foo(){
  int a,b,c,d;

  a = 5; b = 6;
  c = a+1; d=c-1;


  bar();


  d = a+d;
  return();
}
```

No need to save r2/r3. Why?

Additions for Caller-save

```
void foo(){
  int a,b,c,d;

  a = 5; b = 6;
  c = a+1; d=c-1;
  save r1 to stack
  save r4 to stack
  bar();
  restore r4
  restore r1
  d = a+d;
  return();
}
```

Assume bar() will overwrite registers holding a,d

Additions for Callee-save

```
void foo(){
  int a,b,c,d;
  save r1
  save r2
  save r3
  save r4
  a = 5; b = 6;
  c = a+1; d=c-1;
  bar();
  d = a+d;
  restore r4
  restore r3
  restore r2
  restore r1
  return();
}
```

bar() will save a,b, but now foo() must save main's variables

"caller-save" vs. "callee-save"

- Caller-save
  - What if bar() doesn't use r1/r4?
  - No harm done, but wasted work

```
void foo(){
  int a,b,c,d;

  a = 5; b = 6;
  c = a+1; d=c-1;
  save r1 to stack
  save r4 to stack
  bar();
  restore r1
  restore r4
  d = a+d;
  return();
}
```

- Callee-save
  - What if main() doesn't use r1-r4?
  - No harm done, but wasted work

```
void foo(){
  int a,b,c,d;
  save r1
  save r2
  save r3
  save r4
  a = 5; b = 6;
  c = a+1; d=c-1;
  bar();
  d = a+d;
  restore r1
  restore r2
  restore r3
  restore r4
  return();
}
```

Saving/Restoring Optimizations

CALLER-CALLEE

- Where can we avoid loads/stores?
- Caller-saved
  - Only needs saving if value is "live" across function call
  - Live = contains a useful value: Assign value before function call, use that value after the function call
  - In a leaf function (a function that calls no other function), caller saves can be used without saving/restoring

a, d are live

b, c are NOT live

```
void foo(){
  int a,b,c,d;

  a = 5; b = 6;
  c = a+1; d=c-1;

  bar();

  d = a+d;
  return();
}
```

Saving/Restoring Optimizations

CALLER-CALLEE

- Where can we avoid loads/stores?
- Callee-saved
  - Only needs saving at beginning of function and restoring at end of function
  - Only save/restore it if function overwrites the register

Only use r1-r4

No need to save other registers

```
void foo(){
  int a,b,c,d;

  a = 5; b = 6;
  c = a+1; d=c-1;

  bar();

  d = a+d;
  return();
}
```

Agenda

- Branching far away
- Function calls and the call stack
- Assigning variables to memory locations
- Saving registers
- **Caller/callee example**

Caller versus Callee

- Which is better??
- Let's look at some examples...

- Simplifying assumptions:
  - A function can be invoked by many different call sites in different functions.
  - Assume no inter-procedural analysis (hard problem)
    - A function has no knowledge about which registers are used in either its caller or callee
    - Assume main() is not invoked by another function
  - Implication
    - Any register allocation optimization is done using function local information

Caller-saved vs. callee saved – Multiple function case

```
void main(){
  int a,b,c,d;

  c = 5; d = 6;
  a = 2; b = 3;
  foo();
  d = a+b+c+d;
}
```

```
void foo(){
  int e,f;


  e = 2; f = 3;
  bar();
  e = e + f;
}
```

```
void bar(){
  int g,h,i,j;

  g = 0; h = 1;
  i = 2; j = 3;
  final();
  j = g+h+i;
}
```

```
void final(){
  int y,z;


  y = 2; z = 3;

  z = y+z;
}
```

Note: assume main does not have to save any callee registers

Caller-saved vs. callee saved – Multiple function case

- Questions:

1. How many registers need to be saved/restored if we use a **caller-save** convention?
2. How many registers need to be saved/restored if we use a **callee-save** convention?
3. How many registers need to be saved/restored if we use a mix of **caller-save** and **callee-save**?

Question 1: Caller-save

```
void main(){
  int a,b,c,d;
  c = 5; d = 6;
  a = 2; b = 3;
  [4 STUR]
  foo();
  [4 LDUR]
  d = a+b+c+d;
}
```

```
void foo(){
  int e,f;

  e = 2; f = 3;
  [2 STUR]
  bar();
  [2 LDUR]
  e = e + f;
}
```

```
void bar(){
  int g,h,i,j;
  g = 0; h = 1;
  i = 2; j = 3;
  [3 STUR]
  final();
  [3 LDUR]
  j = g+h+i;
}
```

```
void final(){
  int y,z;


  y = 2; z = 3;

  z = y+z;
}
```

Total: 9 STUR / 9 LDUR

Question 2: Callee-save

```
void main(){
  int a,b,c,d;

  c = 5; d = 6;
  a = 2; b = 3;
  foo();
  d = a+b+c+d;



}
```

```
void foo(){
  [2 STUR]
  int e,f;

  e = 2; f = 3;
  bar();
  e = e + f;


  [2 LDUR]
}
```

```
void bar(){
  [4 STUR]
  int g,h,i,j;
  g = 0; h = 1;
  i = 2; j = 3;
  final();
  j = g+h+i;


  [4 LDUR]
}
```

```
void final(){
  [2 STUR]
  int y,z;

  y = 2; z = 3;

  z = y+z;


  [2 LDUR]
}
```

Total: 8 STUR / 8 LDUR

Is one better?

- Caller-save works best when we don't have many live values across function call
- Callee-save works best when we don't use many registers overall
- We probably see functions of both kinds across an entire program
- Solution:
  - Use both!
  - E.g. if we have 6 registers, use some (say r0-r2) as caller-save and others (say r3-r5) as callee-save
  - Now each function can optimize for each situation to reduce saving/restoring
  - Not discussed further in this class

LEGv8 ABI- Application Binary Interface

- The ABI is an agreement about how to use the various registers
- Not enforced by hardware, just a convention by programmers / compilers
- If you want your code to work with other functions / libraries, **follow these**
- Some register conventions in ARMv8
  - X30 is the **link register** – used to hold return address
  - X28 is **stack pointer** – holds address of top of stack
  - X19-X27 are **callee-saved** – function must save these before writing to them
  - X0-15 are **caller-saved** –function must save live values before call
  - X0-X7 used for **arguments** (memory used if more space is needed)
  - X0 used for **return value**

Caller/Callee

- Still not clicking?
- Don't worry, this is a tricky concept for students to get
- Check out supplemental video
    - https://www.youtube.com/watch?v=SMH5uL3HiiU
    - Don’t worry about mixed caller/callee save case at the end
- Come to office hours to go over examples

Today we’ll finish up software

- Introduce linkers and loaders
  - Basic relationship of complier, assembler, linker and loader.
  - Object files
    - Symbol tables and relocation tables

Source Code to Execution

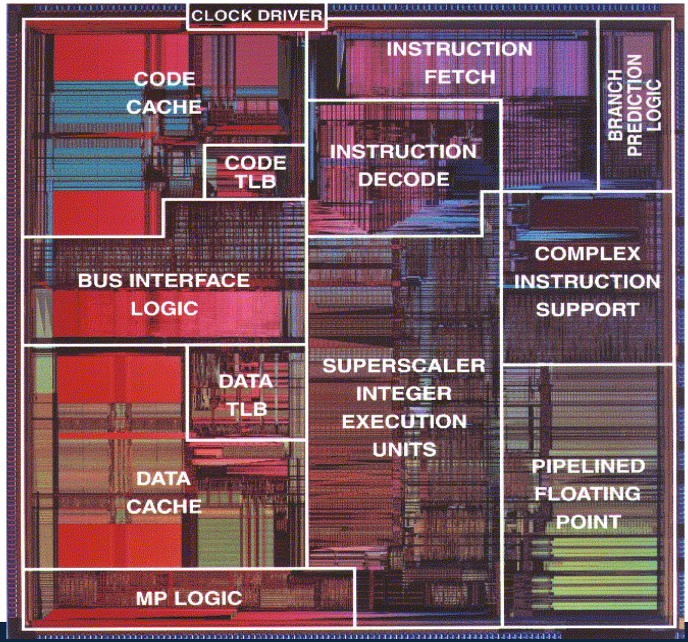
- In project 1a, our view is this:

Assembly

Assembler

Executable

Why do we write programs in multiple files?



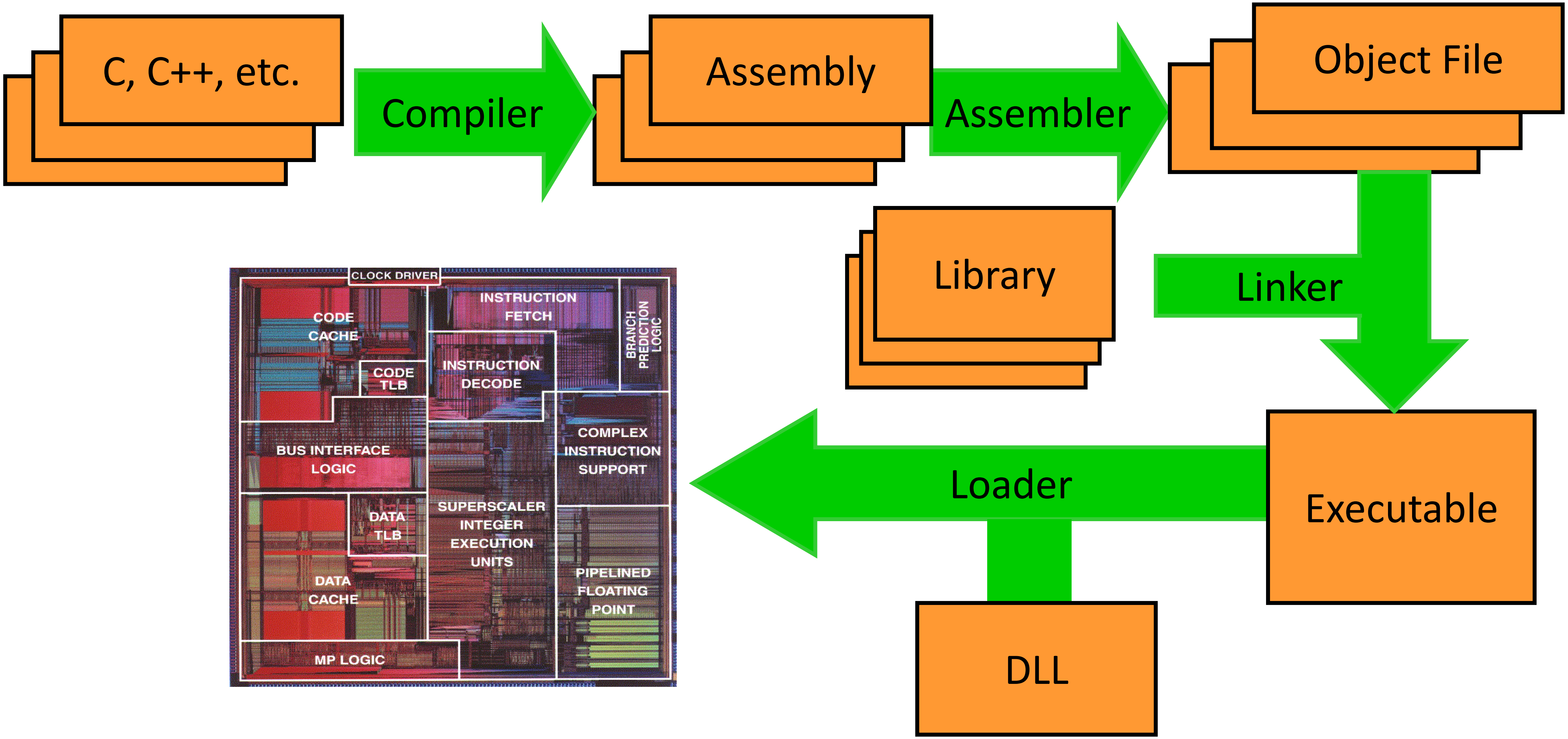
Not very accurate… why?
Because in reality, we have multiple files

Multi-file programs

- In practice, programs are made from thousands or millions of lines of code
  - Use pre-existing libraries like stdlib
- 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 do object files look like?



What do object files look like?

```
extern int X;
extern void foo();
int Y;

void main() {
  Y = X + 1;
  foo();
}
```

```
extern int Y;
int X;

void foo() {
  Y *= 2;
}
```

"extern" means defined in another file

Compile

```
.main:
LDUR    X1, [XZR, X]
ADDI    X9, X1, #1
STUR    X9, [XZR, Y]
BL      foo
HALT
```

Compile

```
.foo:
LDUR    X1, [XZR, Y]
LSL     X9, X1, #1
STUR    X9, [XZR, Y]
BR      X30
```

Uh-oh! Don't know address of X, Y, or foo!

Linking

```
.main:
LDUR    X1, [XZR, X]
ADDI    X9, X1, #1
STUR    X9, [XZR, Y]
BL      foo
HALT
```

Assemble

???

```
.foo:
LDUR    X1, [XZR, Y]
LSL     X9, X1, #1
STUR    X9, [XZR, Y]
BR      X30
```

Assemble

???

What needs to go in this intermediate "object file"?

LINK

LINK

NOTE: this will actually be in machine code, not assembly

```
LDUR    X1, [XZR, #40]
ADDI    X9, X1, #1
STUR    X9, [XZR, #36]
BL      #2
HALT
LDUR    X1, [XZR, #36]
LSL     X9, X1, #1
STUR    X9, [XZR, #36]
BR      X30
// Addr #36 starts here
```

Linking

```
.main:
LDUR    X1, [XZR, X]
ADDI    X9, X1, #1
STUR    X9, [XZR, Y]
BL      foo
HALT
```

Assemble

???

LINK

```
LDUR    X1, [XZR, #40]
ADDI    X9, X1, #1
STUR    X9, [XZR, #36]
BL      #2
HALT
LDUR    X1, [XZR, #36]
LSL     X9, X1, #1
STUR    X9, [XZR, #36]
BR      X30
```

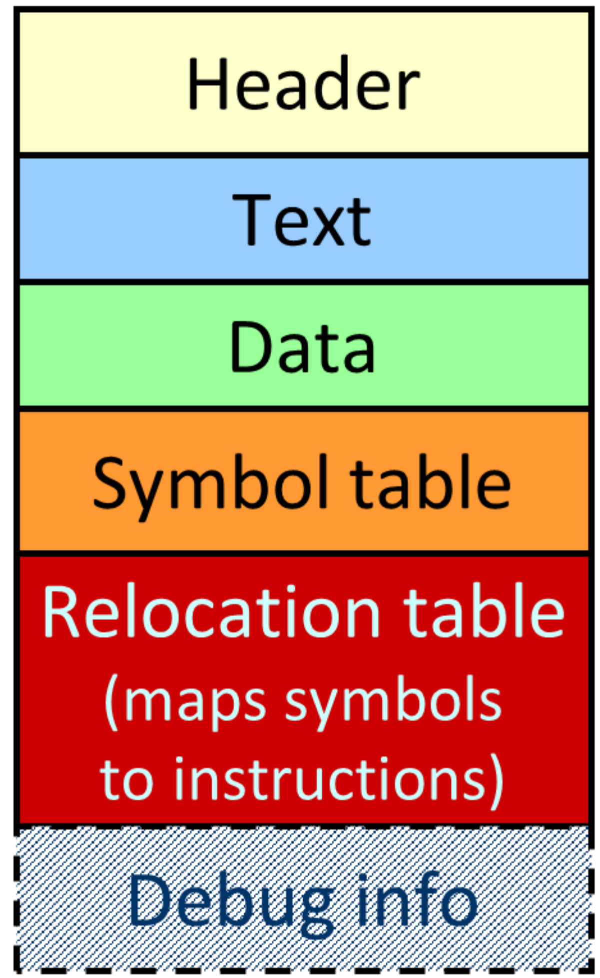
We need:

- **the assembled machine code:**
- **list of instructions that need to be updated once addresses are resolved**
- **list of symbols for cross-ref**

What do object files look like?

- Since we can't make executable, we make an object file
- Basically, includes the machine code that will go in the executable
  - Plus extra information on what we need to modify once we stitch all the other object files together
- Looks like this ->

Object code format



We won't discuss "Debug" much. Gets included when you compile with "-g" in gcc

Assembly → Object file - example

```c
extern int G;
extern void B();
int X = 3;
main() {
  int Y = G + 1;
  B();
}
```

LDUR	X1, [XZR, G]
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, [XZR, G]	
	4	ADDI X9, X1, #1	
	8	BL B	
Data	0	X	3
	Label	Address	
Symbol table	X	0	
	B	-	
	main	0	
	G	-	
Reloc table	Addr	Instruction type	Dependency
	0	LDUR	G
	8	BL	B

Assembly → Object file - example

Header: keeps track of size of each section

```
extern in
extern vo
int X = 3;
main() {
   int Y = G + 1;
   B();
}
```

LDUR	X1, [XZR, G]
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, [XZR, G]	
	4	ADDI X9, X1, #1	
	8	BL B	
Data	0	X	3
Symbol table	Label	Address	
	X	0	
	B	-	
	main	0	
	G	-	
Reloc table	Addr	Instruction type	Dependency
	0	LDUR	G
	8	BL	B

Assembly → Object file - example

```
extern int G;
extern void B();
int X = 3
main() {
  int Y =
  B();
}
```

Text: machine code

```
LDUR    X1, [XZR, G]
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, [XZR, G]	
	4	ADDI X9, X1, #1	
	8	BL B	
Data	0	X	3
	Label	Address	
Symbol table	X	0	
	B	-	
	main	0	
	G	-	
Reloc table	Addr	Instruction type	Dependency
	0	LDUR	G
	8	BL	B

Simplifying Assumption for EECS370

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

Assembly → Object file - example

```
extern int G;
extern void B();
int X = 3;
main() {
   int Y = G + 1;
   B();
}
```

Data: initialized globals and static locals

LDUR	X1, [XZR, G]
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, [XZR, G]	
	4	ADDI X9, X1, #1	
	8	BL B	
Data	0	X	3
Symbol table	Label	Address	
	X	0	
	B	-	
	main	0	
	G	-	
Reloc table	Addr	Instruction type	Dependency
	0	LDUR	G
	8	BL	B

Assembly → Object file - example

```
extern int G;
extern void B();
int X = 3;
main() {
   int Y = G + 1;
   B();
}
```

LDUR
ADDI
BL

Symbol table:
Lists all labels visible outside this file (i.e. function names and global variables)

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

Assembly → Object file - example

```
extern int G;
extern void B();
int X = 3;
main() {
  int Y = G + 1;
  B();
}
```

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

Relocation Table: list of instructions and data words that must be updated if things are moved in memory

Class Problem 1

Which symbols will be put in 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;
}
```

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	-

Which lines / instructions are in the relocation table? (i.e. which "things" need to be updated after linking?)

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 }
```

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.

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, #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> these just used by assembler/linker

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
    - Initializes registers (PC and SP most important)
- Take operating systems class (EECS 482) to learn more!

Summary

- 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

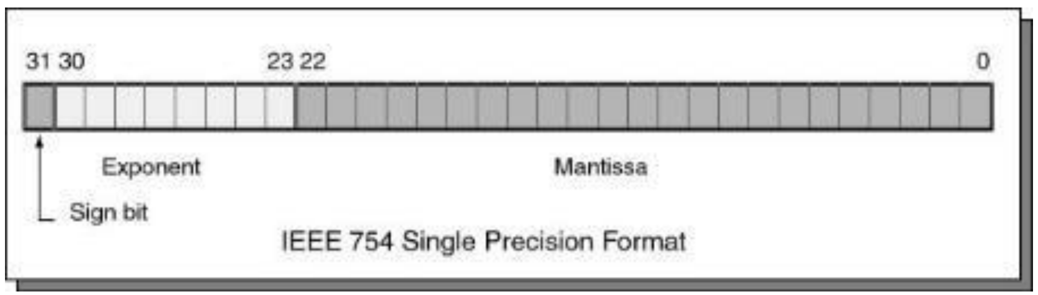
Floating Point Arithmetic

Why floating point

- Have to represent real numbers somehow
- Rational numbers
  - Ok, but can be cumbersome to work with
- 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: this is good!**
  - Exponential notation allows HUGE dynamic range
  - Constant (approximately) relative precision across the whole range

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

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

- Step 1: convert from decimal to binary
  - 1st bit after "binary" point represents 0.5 (i.e. 2^{-1})
  - 2nd bit represents 0.25 (i.e. 2^{-2})
  - etc.

Floating Point Representation

$$10.625_{10} \Rightarrow 1010.101_2$$

$$1.010101 \times 2^3$$

- Step 2: normalize number by shifting binary point until you get 1.XXX * 2^Y

Floating Point Representation

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

$$1.010101 \times 2^{3}$$

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

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

- Step 3: store relevant numbers in proper location (ignoring initial 1 of significand)

Floating Point Representation

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

$\mathbf{1}.010101 \times 2^{3}$

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

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

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

Next Time

- Wrap up Floating Point
- And… hardware time!