

CALL

(Compiler/Assembler/Linker/ Loader)

Administrivia...

- Exam is on October 19th from 7-9 pm
 - Sorry for the disruption. What happened?
 - There was an error in room booking and our exam wasn't booked at the start of the semester...
 - So when we found out it was too late to get enough space to seat everyone!
 - This isn't helped by being at a University that hasn't built a large lecture hall in well, forever!
 - Final will clobber the midterm if you can't make the new exam date
- Remember, if you prefer a remote exam you can do so for any reason
 - Remote exams will be remote proctored
 - We will also have two alternate (remote only) exam slots: 9-11pm and 7-9am the next day for those with conflicts

Agenda

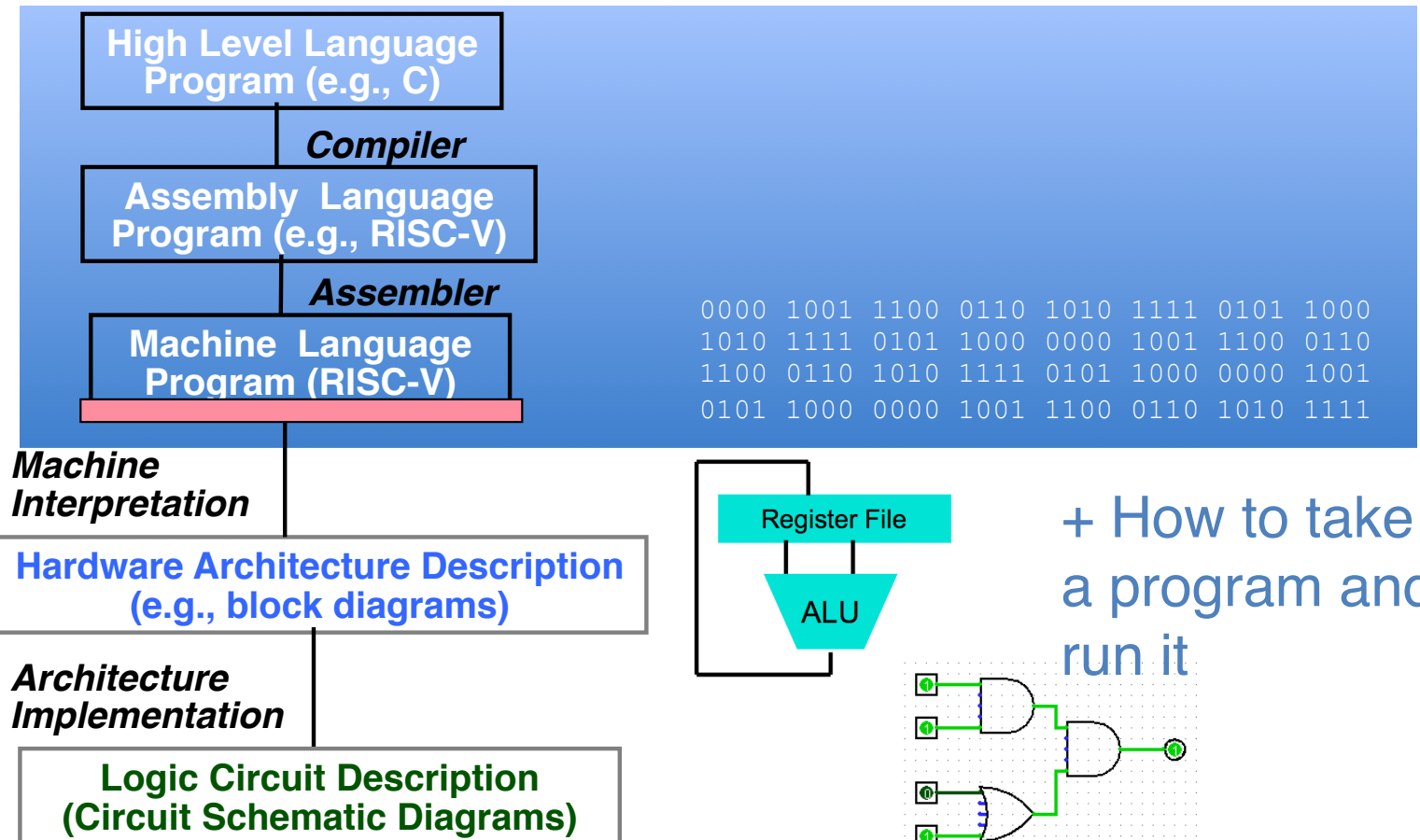
- Interpretation vs Compilation
- The CALL chain
- Producing Machine Language

Levels of Representation/Interpretation

```
temp = v[k];  
v[k] = v[k+1];  
v[k+1] = temp;
```

```
lw  t0, 0(a2)  
lw  t1, 4(a2)  
sw  t1, 0(a2)  
sw  t0, 4(a2)
```

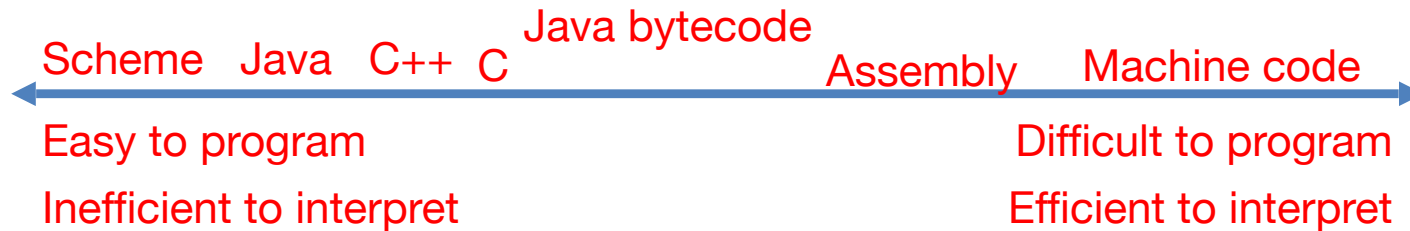
Anything can be represented
as a *number*,
i.e., data or instructions



Language Execution Continuum

解释器

- An **Interpreter** is a program that executes other programs.

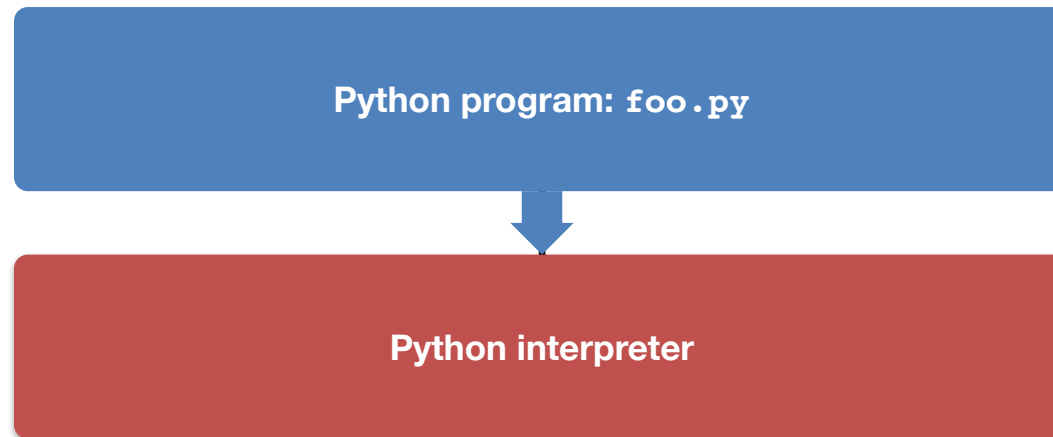


- Language **translation** gives us another option
- In general, we **interpret** a high-level language when efficiency is not critical and **translate** to a lower-level language to increase performance
- Although this is becoming a “distinction without a difference”
Many interpreters do a “just in time” runtime compilation to bytecode that either is emulated or directly compiled to machine code (e.g. the JVM)

Interpretation vs Translation

- How do we run a program written in a source language?
- **Interpreter**: Directly executes a program in the source language
- **Translator**: Converts a program from the source language to an equivalent program in another language
- For example, consider a Python program **foo.py**

Interpretation



- Python interpreter is just a program that reads a python program and performs the functions of that python program
- Well, that's an exaggeration, the interpreter converts to a simple bytecode that the interpreter runs... Saved copies end up in .pyc files

Interpretation

- Any good reason to interpret machine language in software?
- Simulators: Useful for learning / debugging
- Apple CPU Transitions
 - Switched from Motorola 680x0 instruction architecture to PowerPC and later from PowerPC to x86
 - We're seeing this again right now with the M1 chips (which use arm64)!
 - Could require all programs to be re-translated from high level language
 - Instead, let executables contain old and/or new machine code, interpret old code in software if necessary (emulation)

Interpretation vs. Translation? (1/2)

- Generally easier to write interpreter
- Interpreter closer to high-level, so can give better error messages
- Translator reaction: add extra information to help debugging (line numbers, names):
This is what `gcc -g` does, it tells the compiler to add all the debugging information
- Interpreter slower (10x?), code smaller (2x? or not?)
- Interpreter provides instruction set independence: run on any machine

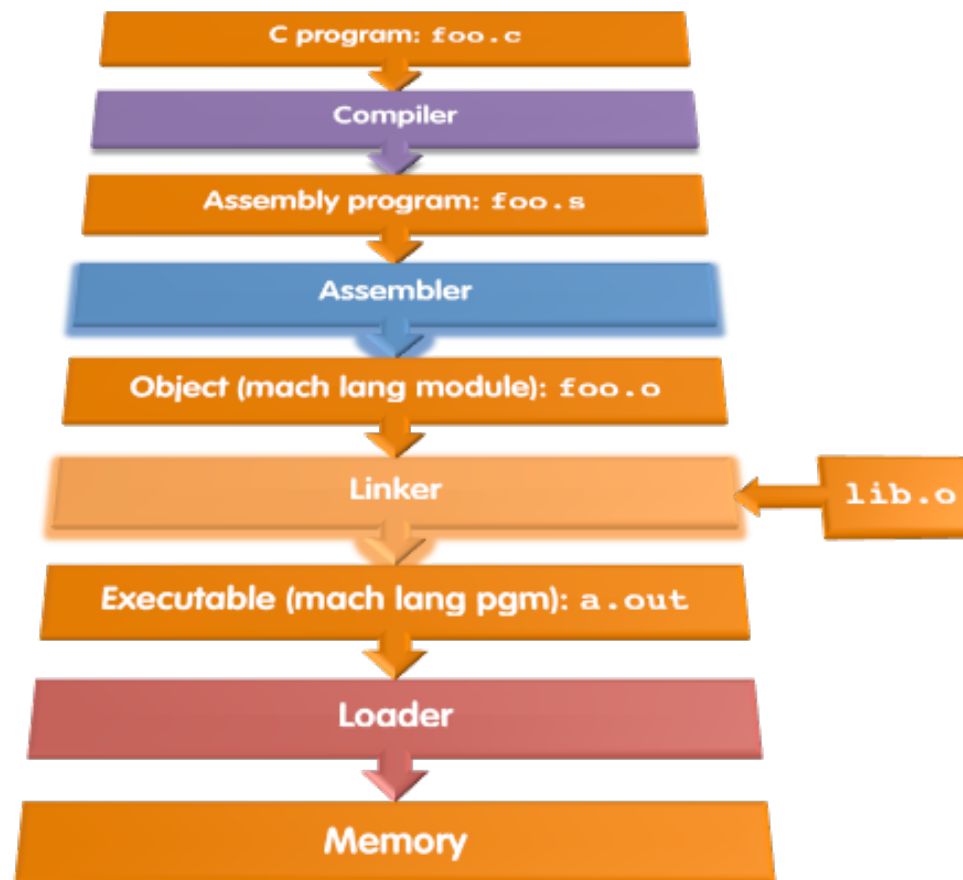
Interpretation vs. Translation? (2/2)

- Translated/compiled code almost always more efficient and therefore **higher performance**:
 - Important for many applications, particularly operating systems.
- Compiled code does the hard work once: during compilation
 - Which is why most “interpreters” these days are really “**just in time compilers**”: **don’t throw away the work processing the program when you reexecute a function**
 - This is especially true of web browsers:
They **dynamically compile JavaScript using first a quick & dirty compiler and, if the code is repeatedly called, a more aggressive compiler**

Agenda

- Interpretation vs Compilation
- The CALL chain
- Producing Machine Language

Steps Compiling a C program



Compiler

- Input: High-Level Language Code
(e.g., **foo.c**)
- Output: Assembly Language Code
(e.g., **foo.s** for RISC-V)
 - Code matches the ***calling convention*** for the architecture
- Note: Output *may* contain pseudo-instructions
- Pseudo-instructions: instructions that assembler understands but not in machine
For example:

- **j label** \Rightarrow **jal x0 label**

Steps In The Compiler

Computer Science 61C Fall 2021

- **Lexer:** *Input* \Rightarrow *Tokens*
 - Turns the **input** into **"tokens"**, recognizes problems with the tokens
- **Parser:** *Tokens* \Rightarrow *Syntax Tree*
 - Turns the tokens into an "Abstract Syntax Tree", recognizes problems in the program structure
- **Semantic Analysis and Optimization:**
 - Checks for semantic errors, may reorganize the code to make it better
- **Code generation:**
 - Output the assembly code

下面的三个例子演示了什么是语法错误、什么是语义错误以及什么是运行时错误。

语法错误(Syntactic Error)

如果一个程序包含语法错误,则不能通过编译。

```
1 public static int returnNull(){
2     System.out.println("haha");
3 }
```

语义错误(Semantic Error)

如果程序包含了语义错误,则可以通过编译,但是得到的结果是错误的,或者不是所期望的功能。

```
1 public static int calSquareArea(int sideLength){
2     return sideLength * 2;
3 }
```

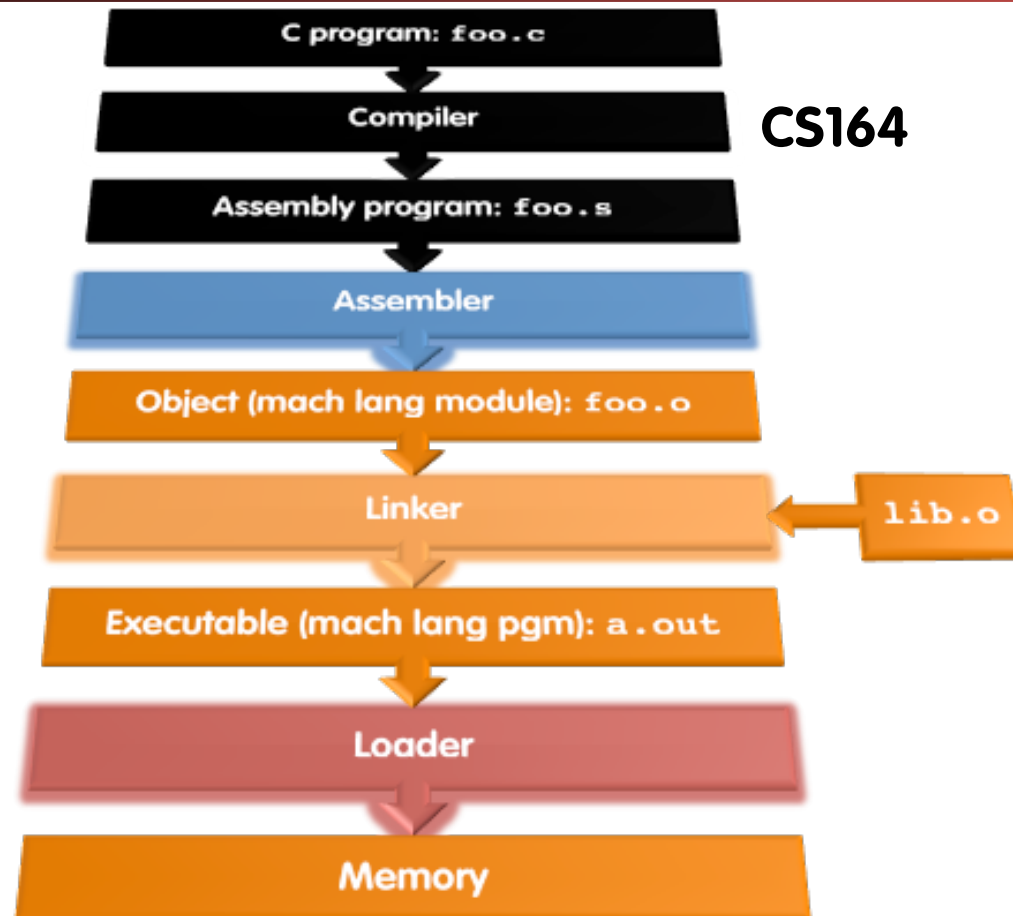
运行时错误(Runtime Error)

Runtime errors 会在程序运行的时候发生。

```
1 public static void main(String[] args) {
2     divideInt(4,0);
3 }
4
5 public static int divideInt(int a, int b){
6     return a/b;
7 }
```

Wawrzynek and Weaver

Where Are We Now?



Assembler: A dumb compiler for assembly language

- Input: Assembly Language Code
(e.g., **foo.s**)
- Output: Object Code, information tables
(e.g., **foo.o**)
- Reads and Uses **Directives**
- Replace Pseudo-instructions
- Produce ***Machine Language*** rather than just ***Assembly Language***
- Creates **Object File**

Assembler Directives

汇编指令

- Give directions to assembler, but do not produce machine instructions
 - **.text**: Subsequent items put in user text segment (machine code)
 - **.data**: Subsequent items put in user data segment (binary rep of data in source file)
 - **.globl sym**: declares `sym` global and can be referenced from other files
 - **.string str**: Store the string `str` in memory and null-terminate it
 - **.word w1...wn**: Store the n 32-bit quantities in successive memory words

Pseudo-instruction Replacement

- Assembler treats convenient variations of machine language instructions as if real instructions

Pseudo	Real
<code>nop</code>	<code>addi x0, x0, 0</code>
<code>not rd, rs</code>	<code>xori rd, rs, -1</code>
<code>beqz rs, offset</code>	<code>beq rs, x0, offset</code>
<code>bgt rs, rt, offset</code>	<code>blt rt, rs, offset</code>
<code>j offset</code>	<code>jal x0, offset</code>
<code>ret</code>	<code>jalr x0, x1, offset</code>
<code>call offset</code> (if too big for just a <code>jal</code>)	<code>auipc x6, offset[31:12]</code> <code>jalr x1, x6, offset[11:0]</code>
<code>tail offset</code> (if too far for a <code>j</code>)	<code>auipc x6, offset[31:12]</code> <code>jalr x0, x6, offset[11:0]</code>

So what is "tail" about...

- Often times your code has a convention like this:
 - ```
{
 ...
 lots of code
 return foo(y);
}
```
  - It can be a recursive call to `foo()` if this is within `foo()`, or call to a different function...
- So for efficiency...
  - Evaluate the arguments for `foo()` and place them in `a0-a7`...
  - Restore `ra`, all callee saved registers, and `sp`
  - Then call `foo()` **with *j* or *tail***
- Then when `foo()` returns, it can return **directly** to where it needs to return to
  - Rather than returning to wherever `foo()` was called and returning from there
- **Tail Call Optimization**

# Agenda

- Interpretation vs Compilation
- The CALL chain
- Producing Machine Language

# Producing Machine Language (1/3)

- Simple Case
  - Arithmetic, Logical, Shifts, and so on
  - All necessary info is within the instruction already:  
Just convert into the binary representations we saw on last time
- What about Branches?
  - PC-Relative
  - So once pseudo-instructions are replaced by real ones, we know by how many instructions to branch
- So these can be handled

# Producing Machine Language (2/3)

- “Forward Reference” problem
  - Branch instructions can refer to labels that are “forward” in the program:

```
 or s0, x0, x0
L1: slt t0, x0, $a1
 beq t0, x0, L2
 addi a1, a1, -1
 jal x0, L1
L2: add $t1, $a0, $a1
```

- Solved by taking 2 passes over the program
  - First pass remembers position of labels
    - Can do this when we expand pseudo instructions
  - Second pass uses label positions to generate code

# Producing Machine Language (3/3)

- What about jumps (**j** and **jal**)?
  - Jumps within a file are PC relative (and we can easily compute)
  - Jumps to ***other*** files we can't
- What about references to static data?
  - **la** (Load Address, basically **li** but for a location) gets broken up into **lui** and **addi**
  - These will require the full 32-bit address of the data
- These can't be determined yet, so we create two tables...

# Symbol Table

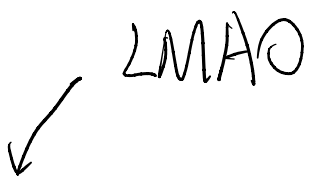
- List of “items” in this file that may be used by other files
- What are they?
  - Labels: function calling
  - Data: anything in the `.data` section; variables which may be accessed across files



# Relocation Table

- List of “items” this file needs the address of later
- What are they?
  - Any external label jumped to: **jal**
    - external (including lib files)
  - Any piece of data in static section
    - such as the **la** instruction

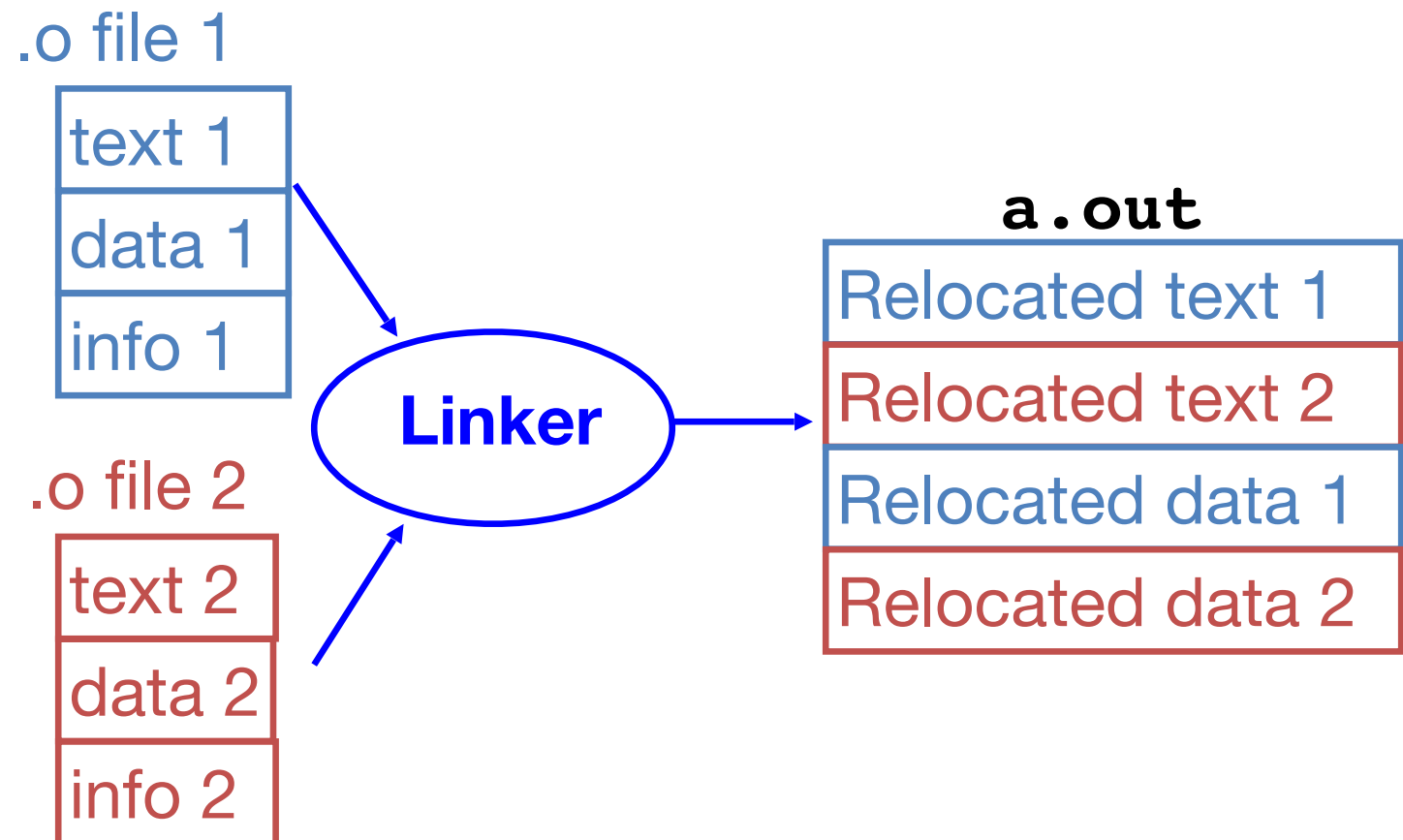
# Object File Format

- object file header: size and position of the other pieces of the object file
  - text segment: the machine code
  - data segment: binary representation of the static data in the source file
  - relocation information: identifies lines of code that need to be fixed up later
  - symbol table: list of this file's labels and static data that can be referenced
  - debugging information
  - A standard format is ELF (except Microsoft)
- [http://www.skyfree.org/linux/references/ELF\\_Format.pdf](http://www.skyfree.org/linux/references/ELF_Format.pdf)
- LAO
- 

# Linker (1/3)

- Input: Object code files with information tables (e.g., `foo.o`, `libc.o`)
- Output: Executable code (e.g., `a.out`)
- Combines several object (`.o`) files into a single executable (“[linking](#)”)
- Enable separate compilation of files
  - Changes to one file do not require recompilation of the whole program
    - Windows 7 source was > 40 M lines of code!
  - Old name “Link Editor” from editing the “links” in jump and link instructions

# Linker (2/3)



# Linker (3/3)

- 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
- Step 3: **Resolve references**
  - Go through Relocation Table; handle each entry
  - That is, fill in all **absolute addresses**

# Three Types of Addresses

- PC-Relative Addressing (**beq**, **bne**, **jal**)
  - **never relocate**
- External Function Reference (usually **jal**)
  - always relocate
- Static Data Reference (often **auipc** and **addi**)
  - always relocate
- RISC-V often uses **auipc** rather than **lui** so that a big block of stuff can be further relocated as long as it is fixed relative to the pc
  - load upper immediate*

# Absolute Addresses in RISC-V

- Which instructions need relocation editing?
  - Jump and link: ONLY for external jumps

|            |           |               |
|------------|-----------|---------------|
| <b>jal</b> | <b>rd</b> | <b>xxxxxx</b> |
|------------|-----------|---------------|

- Loads and stores to variables in static area, relative to the global pointer

|              |           |           |               |
|--------------|-----------|-----------|---------------|
| <b>lw/sw</b> | <b>gp</b> | <b>x?</b> | <b>xxxxxx</b> |
|--------------|-----------|-----------|---------------|

- What about conditional branches?

|            |           |           |                |
|------------|-----------|-----------|----------------|
| <b>beq</b> | <b>rs</b> | <b>rt</b> | <b>xxxxxxx</b> |
|------------|-----------|-----------|----------------|

- PC-relative addressing **preserved** even if code moves

# Resolving References (1/2)

- Linker **assumes** first word of first text segment is at address **0x04000000**.
  - (More later when we study “virtual memory”)
- 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 and each piece of data being referenced

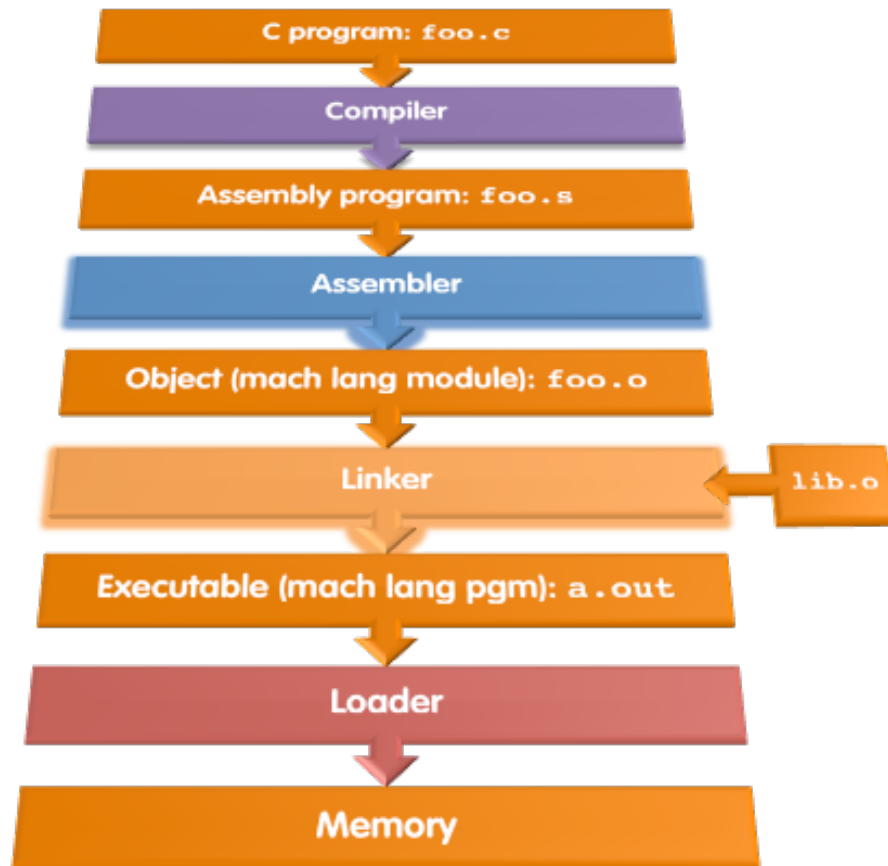


# Resolving References (2/2)

- To resolve references:
  - search for reference (data or label) in all “user” symbol tables
  - if not found, search library files (for example, for `printf`)
  - once absolute address is determined, fill in the machine code appropriately
- Output of linker: executable file containing text and data (plus header)

# To Summarize

- Compiler converts a single HLL file into a single assembly language file.
- Assembler removes pseudo-instructions, converts what it can to machine language, and creates a checklist for the linker (relocation table). A .s file becomes a .o file.
  - Does 2 passes to resolve addresses, handling internal forward references
- Linker combines several .o files and resolves absolute addresses.
  - Enables separate compilation, libraries that need not be compiled, and resolves remaining addresses
- Loader loads executable into memory and begins execution.



# Loader Basics

- Input: Executable Code (e.g., **a.out**)
- Output: (program is run)
- Executable files are stored on disk
- When one is run, loader's job is to load it into memory and start it running
- In reality, loader is the operating system (OS)
  - Loading is one of the OS's tasks
  - And these days, the loader actually does a lot of the linking: Linker's 'executable' is actually only partially linked, instead still having external references

# Loader ... what does it do?

- Reads executable file's header to determine size of text and data segments
- 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
- Copies arguments passed to the program onto the stack
- Initializes machine registers
  - Most registers cleared, but stack pointer assigned address of 1st free stack location
- Jumps to start-up routine that copies program's arguments from stack to registers & sets the PC
  - If main routine returns, start-up routine terminates program with the exit system call

Example: C  $\Rightarrow$  Asm  $\Rightarrow$  Obj  $\Rightarrow$  Exe  $\Rightarrow$  Run

### C Program Source Code: *prog.c*

```
#include <stdio.h>

int main (int argc, char *argv[]) {
 int i, sum = 0;
 for (i = 0; i <= 100; i++)
 sum = sum + i * i;
 printf ("The sum of sq from 0 .. 100 is %d\n", sum);
}
```

*“printf” lives in “libc”*

# Compilation: Assembly Language:

$i = t0, \text{sum} = a1$

```
.text
.align 2
.globl main
main:
 addi sp, sp, -4
 sw ra, 0(sp)
 mv t0, x0
 mv a1, x0
 li t1, 100
 j check
loop:
 mul t2, t0, t0
 add a1, a1, t2
 addi t0, t0, 1
```

Pseudo-  
Instructions?

```
check:
 blt t0, t1 loop:
 la $a0, str
 jal printf
 mv a0, x0
 lw ra, 0(sp)
 addi sp, sp, 4
 ret
.data
.align 0
str:
 .asciiz "The sum of sq from 0
 .. 100 is %d\n"
```

# Compilation: Assembly Language:

i = t0, sum = a1

```
.text
.align 2
.globl main
main:
 addi sp, sp, -4
 sw ra, 0(sp)
 mv t0, x0
 mv a1, x0
 li t1, 100
 j check
loop:
 mul t2, t0, t0
 add a1, a1, t2
 addi t0, t0, 1
```

Pseudo-  
Instructions?  
Underlined

```
check:
 blt t0, t1 loop:
 la $a0, str
 jal printf
 mv a0, x0
 lw ra, 0(sp)
 addi sp, sp, 4
 ret
.data
 .align 0
str:
 .asciiz "The sum of sq from 0
 .. 100 is %d\n"
```

# Assembly step 1: Remove Pseudo Instructions, assign jumps

```
.text
.align 2
.globl main
main:
 addi sp, sp, -4
 sw ra, 0(sp)
 addi t0, x0, 0
 addi a1, x0, 0
 addi t1, x0, 100
 jal x0, 12
loop:
 mul t2, t0, t0
 add a1, a1, t2
 addi t0, t0, 1
```

Pseudo-  
Instructions?  
Underlined

```
check:
 blt t0, t1 -16
 lui a0, l.str
 addi a0, a0, r.str
 jal printf
 addi a0, x0, 0
 lw ra, 0(sp)
 addi sp, sp, 4
 jalr x0, ra
.data
.align 0
str:
 .asciiz "The sum of sq from 0
 .. 100 is %d\n"
```



# Assembly step 2

## Create relocation table and symbol table

- Symbol Table

| Label        | address (in module) | type               |
|--------------|---------------------|--------------------|
| <b>main:</b> | <b>0x00000000</b>   | <b>global text</b> |
| <b>loop:</b> | <b>0x00000014</b>   | <b>local text</b>  |
| <b>str:</b>  | <b>0x00000000</b>   | <b>local data</b>  |

- Relocation Information

| Address           | Instr. type | Dependency    |
|-------------------|-------------|---------------|
| <b>0x0000002c</b> | <b>lui</b>  | <b>l.str</b>  |
| <b>0x00000030</b> | <b>addi</b> | <b>r.str</b>  |
| <b>0x00000034</b> | <b>jal</b>  | <b>printf</b> |

# Assembly step 3

- Generate object (.o) file:
  - Output binary representation for
    - text segment (instructions)
    - data segment (data)
    - symbol and relocation tables
  - Using dummy “placeholders” for unresolved absolute and external references
- And then... We link!

# Linking Just Resolves References...

- So take all the .o files
  - Squish the different segments together
- For each entry in the relocation table:
  - Replace it with the actual address for the symbol table of the item you are linking to
- Result is a single binary

# Static vs. Dynamically Linked Libraries

- What we've described is the traditional way: **statically-linked** approach
  - Library is now part of the executable, so if the library updates, we don't get the fix (have to recompile if we have source)
  - Includes the entire library even if not all of it will be used
  - Executable is self-contained
- Alternative is **dynamically linked libraries** (DLL), common on Windows & UNIX platforms

# Dynamically Linked Libraries

[en.wikipedia.org/wiki/Dynamic\\_linking](https://en.wikipedia.org/wiki/Dynamic_linking)

- Space/time issues
  - + Storing a program requires less disk space
  - + Sending a program requires less time
  - + Executing two programs requires less memory (if they share a library)
    - We will see how this is possible when we talk about virtual memory
  - – At runtime, there's time overhead to do link
- Upgrades
  - + Replacing one file (libXYZ.so) upgrades every program that uses library “XYZ”
  - – Having the executable isn't enough anymore
    - Can create a mess with dependencies, thus “Linux Containers”

*Overall, dynamic linking adds quite a bit of complexity to the compiler, linker, and operating system. However, it provides many benefits that often outweigh these*

# Final Review C Program: Hello.c

```
#include <stdio.h>

int main()
{
 printf("Hello, %s\n", "world");
 return 0;
}
```

# Compiled Hello.c: Hello.s

```
.text
 .align 2
 .globl main
main:
 addi sp,sp,-16
 sw ra,12(sp)
 lui a0,%hi(string1)
 addi a0,a0,%lo(string1)
 lui a1,%hi(string2)
 addi a1,a1,%lo(string2)
 call printf
 lw ra,12(sp)
 addi sp,sp,16
 li a0,0
 ret
 .section .rodata
 .balign 4
string1:
 .string "Hello, %s!\n"
string2:
 .string "world"

Directive: enter text section
Directive: align code to 2^2 bytes
Directive: declare global symbol main
label for start of main
allocate stack frame
save return address
compute address of
string1
compute address of
string2
call function printf
restore return address
deallocate stack frame
load return value 0
return
Directive: enter read-only data section
Directive: align data section to 4 bytes
label for first string
Directive: null-terminated string
label for second string
Directive: null-terminated string
```

# Assembled Hello.s: Linkable Hello.o

```
00000000 <main>:
0: ff010113 addi sp,sp,-16
4: 00112623 sw ra,12(sp)
8: 00000537 lui a0,0x0 # addr placeholder
c: 00050513 addi a0,a0,0 # addr placeholder
10: 000005b7 lui a1,0x0 # addr placeholder
14: 00058593 addi a1,a1,0 # addr placeholder
18: 000000ef jal ra # addr placeholder
1c: 00c12083 lw ra,12(sp)
20: 01010113 addi sp,sp,16
24: 00000513 addi a0,a0,0
28: 00008067 jalr ra
```



# Linked Hello.o: a.out

```
000101b0 <main>:
 101b0: ff010113 addi sp,sp,-16
 101b4: 00112623 sw ra,12(sp)
 101b8: 00021537 lui a0,0x21
 101bc: a1050513 addi a0,a0,-1520 # 20a10 <string1>
 101c0: 000215b7 lui a1,0x21
 101c4: a1c58593 addi a1,a1,-1508 # 20a1c <string2>
 101c8: 288000ef jal ra,10450 # <printf>
 101cc: 00c12083 lw ra,12(sp)
 101d0: 01010113 addi sp,sp,16
 101d4: 00000513 addi a0,0,0
 101d8: 00008067 jalr ra
```

# And the Class So Far...

- Lots of C
  - Including Structures, Functions, Pointers, Pointers to Pointers, Unions, etc...
- Binary numbers
  - Can you count to 31 on the fingers of one hand? Two's Complement?
- Assembly
  - How it works
- CALL

# Integer Multiplication (1/3)

- Paper and pencil example (unsigned):

$$\begin{array}{r} \text{Multiplicand} \quad 1000 \quad 8 \\ \text{Multiplier} \quad \quad \times 1001 \quad 9 \\ \hline \quad \quad \quad 1000 \\ \quad \quad \quad 0000 \\ \quad \quad 0000 \\ \quad + 1000 \\ \hline 01001000 \quad 72 \end{array}$$

- $m$  bits  $\times$   $n$  bits =  $m + n$  bit product

# Integer Multiplication (2/3)

- In RISC-V, we multiply registers, so:
  - 32-bit value x 32-bit value = 64-bit value
- Multiplication is **not** part of standard RISC-V...
  - Instead it is an **optional** extra:  
The compiler needs to produce a series of shifts and adds if the multiplier isn't present
    - Why on the exam we did the multiplication for you...
- Syntax of Multiplication (signed):
  - `mul rd, rs1, rs2`
  - `mulhhigh bits rd, rs1, rs2`
  - Multiplies 32-bit values in those registers and returns either the lower or upper 32b result
    - If you do mulh/mul back to back, the architecture can fuse them
  - Also unsigned versions of the above

# Integer Multiplication (3/3)

- Example:
  - in C: `a = b * c;`
    - `int64_t a; int32_t b, c;`
    - Aside, these types are defined in C99, in `stdint.h`
  - in RISC-V:
    - let `b` be `s2`; let `c` be `s3`; and let `a` be `s0` and `s1` (since it may be up to 64 bits)
    - `mulh as1, bs2, cs3`  
`mul as0, s2, s3`

# Integer Division (1/2)

- Paper and pencil example (unsigned):

$$\begin{array}{r} \text{Dividend} \quad \quad \quad \underline{1001} \quad \text{Quotient} \quad \text{Divisor} \quad 1000 \mid 1001010 \\ -1000 \\ \hline 10 \\ 101 \\ 1010 \\ -1000 \\ \hline 10 \text{ Remainder} \\ \text{(or Modulo result)} \end{array}$$

- Dividend = Quotient x Divisor + Remainder

被除数 商 除数 余数

$$1001010 = 1001 \times 1000 + 10$$

# Integer Division (2/2)

- Syntax of Division (signed):
  - `div rd, rs1, rs2`  
`rem rd, rs1, rs2`
  - Divides 32-bit rs1 by 32-bit rs2, returns the quotient (/) for div, remainder (%) for rem
  - Again, can fuse two adjacent instructions
- Example in C: `a = c / d;`    `b = c % d;`
- RISC-V:
  - `a↔s0; b↔s1; c↔s2; d↔s3`
  - `div s0, s2, s3`  
`rem s1, s2, s3`

# Note Optimization...

- A recommended convention
  - `mulh s1 s2 s3`  
`mul s0 s2 s3`
  - `div s0 s2 s3`  
`rem s1 s2 s3`
- Not a *requirement but...*
  - RISC-V says "if you do it this way, **and** the microarchitecture supports it, it can fuse the two operations into one"
  - Same logic behind much of the 16b ISA design:  
If you follow the convention you can get significant optimizations



# Optional Coolness (not on the test): RISC-V 16b compressed ISA

- Not a complete ISA
- Instead an optimization for the most **common** instruction motifs:
  - Small immediates
  - Common registers (x0, ra/x1, sp/x2)
  - rs2 == rd
- Humans and the compiler don't know **or care**
  - Every 16b instruction has a 32b counterpart
- So the assembler creates them in the same way they expand pseudo-instructions

# EG, some examples...

- Load word/store word relative to the stack pointer
  - 6 bit immediate is word aligned and zero-extended: only positive offsets
  - Cuts the size of prolog/epilog by effectively 50%
- Common register load/store (s0-s1, a0-a5)
  - 5 bit immediate, also word aligned and zero-extended
- J/JAL with a 11 bit immediate
  - Writing link into either x0 or ra
- Branch equal/not equal zero