信息科学与技术学院
School of Information Science and Technology

CS 110
Computer Architecture
CALL

Instructors:

Siting Liu & Chundong Wang

Course website: https://toast-lab.sist.shanghaitech.edu.cn/courses/CS110@ShanghaiTech/Spring-2024/index.html

School of Information Science and Technology (SIST)

ShanghaiTech University

2024/3/21

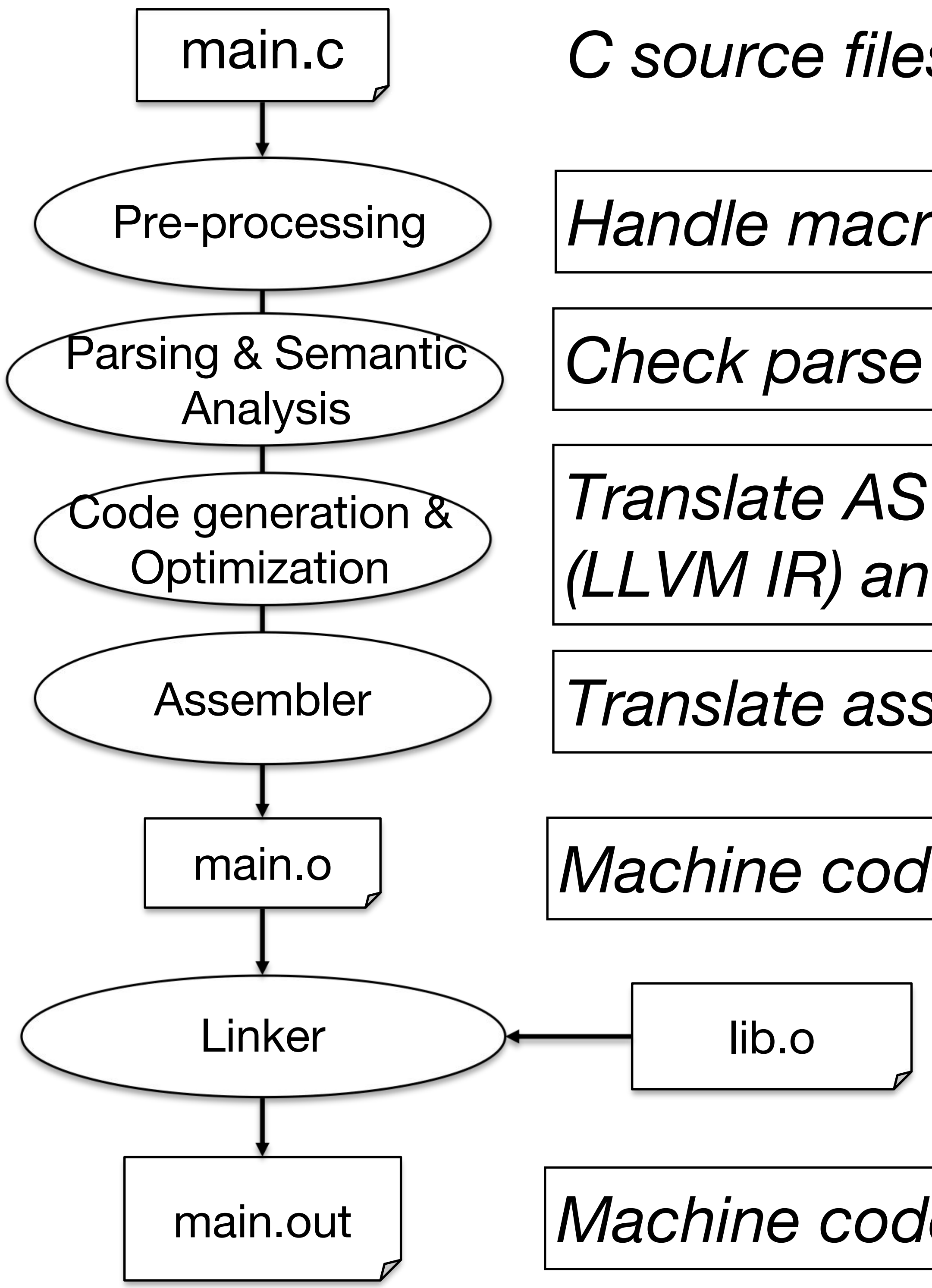
Administrative

- HW2, due Mar. 22nd
- Lab 4 available, please prepare in advance!
- Proj1.1 released, ddl April 8th
- Discussion next week on CALL, useful for Proj1.1, covered by TA Chen Suting at teaching center 301;

Outline

- CALL (compiler, assembler, linker & loader)
    - Compiler
    - Assembler
    - Linker
    - Loader

C Compilation Simplified Review



C source files (text)

Handle macro expansion, #include, etc.

Check parse errors, generate “Abstract Syntax Tree” (AST)

Translate AST to lower level intermediate representation (LLVM IR) and then generate assembly (.s)

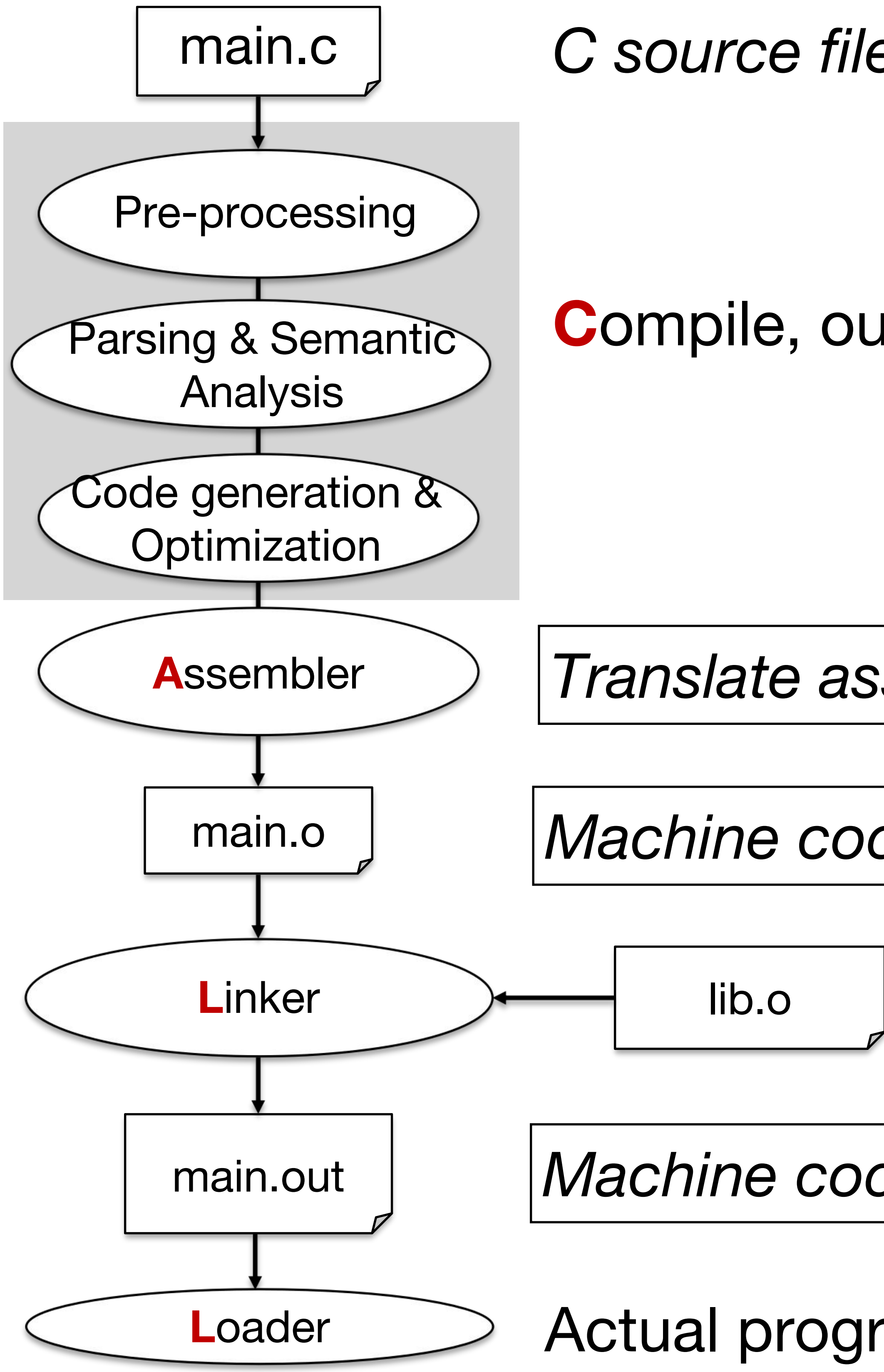
Translate assembly to machine code (binary)

Machine code object files (.o)

Pre-built object file libraries

Machine code executable file (.out)

CALL Overview



C source files (text)

Compile, output assembly (.s)

Translate assembly to machine code (binary)

Machine code object files (.o)

Pre-built object file libraries

Machine code executable file (.out)

Actual program execution

Where are we?

High Level Language Program (e.g., C)

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

Compiler

Assembly Language Program (e.g., RISC-V)

```
lw   t0, 0(s2)
lw   t1, 4(s2)
sw   t1, 0(s2)
sw   t0, 4(s2)
```

Assembler

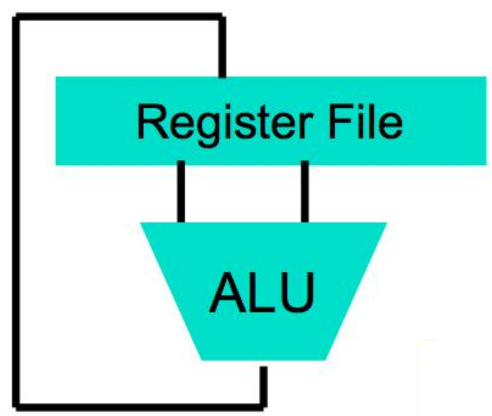
We are here!

Machine Language Program (RISC-V)

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

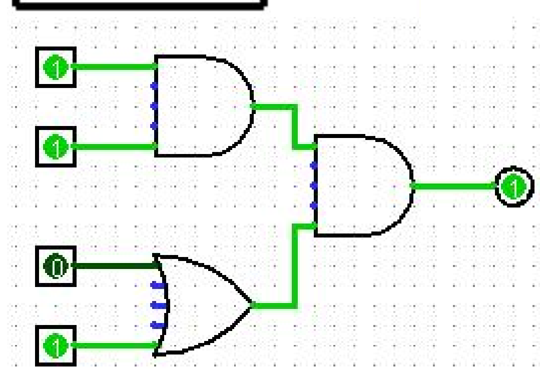
Machine Interpretation

Hardware Architecture Description (e.g., block diagrams)



Architecture Implementation

Logic Circuit Description (Circuit Schematic Diagrams)



Summary

31	30 25	24 21	20	19 15	14 12	11 8	7	6 0	
funct7		rs2		rs1	funct3	rd		opcode	R-type
imm[11:0]				rs1	funct3	rd		opcode	I-type
imm[11:5]		rs2		rs1	funct3	imm[4:0]		opcode	S-type
imm[12]	imm[10:5]	rs2		rs1	funct3	imm[4:1]	imm[11]	opcode	B-type
imm[31:12]						rd		opcode	U-type
imm[20]	imm[10:1]		imm[11]	imm[19:12]		rd		opcode	J-type

imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI

0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND
0000 pred	succ	00000	000	00000	0001111	FENCE
0000 0000	0000	00000	001	00000	0001111	FENCE.I
000000000000		00000	000	00000	1110011	ECALL
000000000001		00000	000	00000	1110011	EBREAK
csr		rs	[illegible]	rd	1110011	CSRRW
csr		rs	[illegible]	rd	1110011	CSRRS
csr		rs	[illegible]	rd	1110011	CSRRC
csr		zimm	101	rd	1110011	CSRRWI
csr		zimm	110	rd	1110011	CSRRSI
csr		zimm	111	rd	1110011	CSRRCI

Not in CS110

Assembler

- Input: assembly language code (generated by compiler, usually contains pseudo-instructions)
- Output: object code, information tables
  - Object file header: size/position of other pieces of the object file
  - Text segment: machine code
  - Data segment: static data
  - Symbol table: list of files' labels, static data can be referenced by other programs
  - Relocation table: Code to be fixed later (by linker)
  - Debugging info.
- Reads and uses directives
- Replace pseudo-instructions
- Produce machine language
- Creates object file

Assembler

- Input: assembly language code (generated by compiler, usually contains pseudo-instructions)
- Output: object code, information tables
- Reads and uses directives
- Replace pseudo-instructions
- Produce machine language
- Creates object file

Directives

- Give directions to assembler, but do not produce machine instructions directly, e.g.,
  - `.text`: Subsequent items put in user text segment (instructions)
  - `.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
  - `.asciiz str`: Store the string str in memory and null-terminate it
  - `.word w1,...,wn`: Store the n 32-bit quantities in successive memory words
  - `.align [int]`: align to power of 2
  - `.option`: specify options such as arch, rvc

Directive Examples

```
.text
memcpy_general:
    add     a5,a1,a2
    beq     a1,a5,.L2
    add     a2,a0,a2
    mv      a5,a0
.L3:
    addi    a1,a1,1
    addi    a5,a5,1
    lbu     a4,-1(a1)
    sb      a4,-1(a5)
    bne     a5,a2,.L3
.L2:
    ret
```

More at https://github.com/riscv-non-isa/riscv-asm-manual/blob/master/riscv-asm.md
Note that not all directives are supported by venus

Assembler

- Input: assembly language code (generated by compiler, usually contains pseudo-instructions)
- Output: object code, information tables
- Reads and uses directives
- Replace pseudo-instructions
- Produce machine language
- Creates object file

Pseudo-instruction Examples

Assembler

Pseudo-instructions	Real instructions
`nop`	`addi x0, x0, 0`
`not rd, rs`	`xori rd, rs, -1`
`beqz rs, offset`	`beq rs, x0, offset`
`bgt rs1, rs2, offset`	`blt rs2, rs1, offset`
`j offset`	`jal x0, offset`
`ret`	`jalr x0, x1, offset`
`call offset (too big to jal)`	`auipc x6, offset[31:12]` `jalr x1, x6, offset[11:0]`
`tail offset (too far to j)`	`auipc x6, offset[31:12]` `jalr x0, x6, offset[11:0]`
`li/la rd imm/label`	`lui rd <hi20bits> (too large)` `addi rd, x0, <low12bits>`
`mv rs1, rs2`	`addi rs1, rs2, 0`

Tail Call

- Simple example

```
              x10/a0
int foo(int x){
  return bar(x * 2);
}
```

```
foo: slli x10,x10,1
     call bar
     [epilogue]        →    j bar
     ret
bar: ...
     ret
```

- Before optimization

```
      main: [prologue]
            call/jal foo
ra  →       [epilogue]
      foo:  slli x10,x10,1
            call bar
            [epilogue]
            ret
      bar:  ...
            ret
```

Tail Call

- Simple example

```
              x10/a0
int foo(int x){
  return bar(x * 2);
}

foo: slli x10,x10,1
     call bar
     [epilogue]    →    j bar
     ret
bar: ...
     ret
```

- After optimization

```
     main: [prologue]
           call/jal foo
ra →       [epilogue]
     foo:  slli x10,x10,1
           tail bar
     bar:  ...
           ret
```

Tail Call

- Nested procedures

```
              x11/a1    x10/a0

int fact (int n, int prod) {
    if (n>1) return fact(n-1, prod*n);
    else return (prod);
}
```

```
fact: addi   t0,x0,1
      ble    x11,t0,Exit
      mul    x10,x10,x11
      addi   x11,x11,-1
      jalr   x0,fact
Exit: add    x10,x0,x10
      jalr   x0,0(x1)
```

Assembler

- Input: assembly language code (generated by compiler, usually contains pseudo-instructions)
- Output: object code, information tables
- Reads and uses directives
- Replace pseudo-instructions
- Produce machine language
- Creates object file

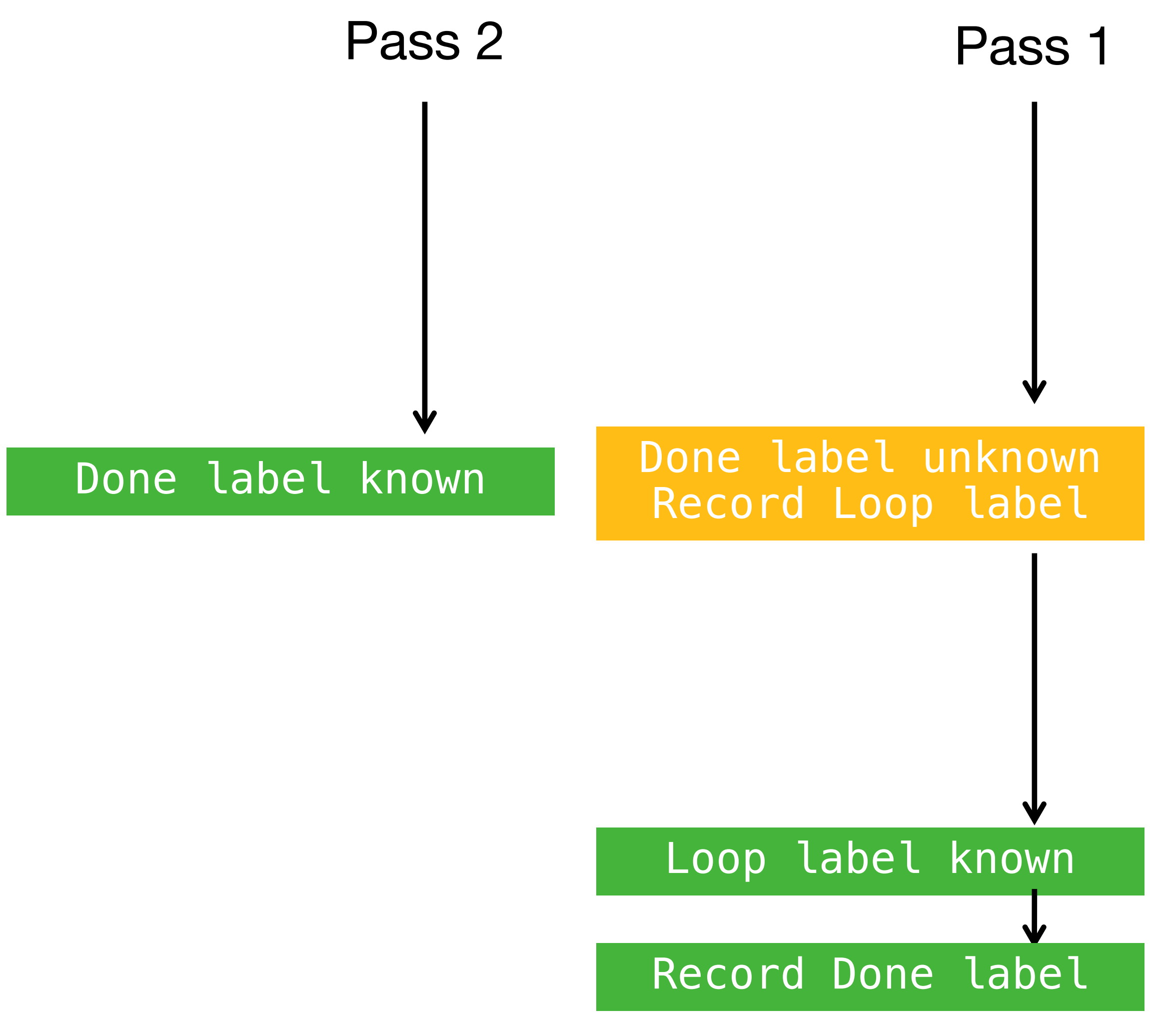
Produce machine language

- Instruction encoding
  - Simple cases: all the logic and arithmetic operations
  - PC-relative branches and jumps:
    - e.g. `beq`/`bne`/etc. and `jal`
    - Position-independent code (PIC): PC-relative addressing can be computed

```
  add x9, x8, x0   # x9=&A[0]
  add x10, x0, x0 # sum=0
  add x11, x0, x0 # i=0
  addi x13,x0, 20 # x13=20
Loop:
  bge x11,x13,Done
  lw x12, 0(x9)    # x12=A[i]
  add x10,x10,x12 # sum+=
  addi x9, x9,4    # &A[i+1]
  addi x11,x11,1   # i++
  j Loop
Done:
  ret
```

PC-relative addressing: two-pass

- “Foward reference” problem
  - Branches, PC-relative jumps can refer to labels that are “forward” in the program
  - Pass 1: remember positions of labels (in symbol table)
  - Pass 2: Use label positions to generate machine code




```
0     add x9, x8, x0   # x9=&A[0]
4     add x10, x0, x0 # sum=0
8     add x11, x0, x0 # i=0
c     addi x13,x0, 20 # x13=20
10  Loop:
      bge x11,x13,Done
14    lw x12, 0(x9)    # x12=A[i]
18    add x10,x10,x12 # sum+=
1c    addi x9, x9,4    # &A[i+1]
20    addi x11,x11,1   # i++
24    j Loop
28  Done:
      ret
```

Other References Missing

- Function call (multiple files, library, etc.)
- Static data (global)
- Assembler jots them down in symbol table and relocation records
- Linker will handle these

Symbol table

- Labels
- `.globl` directive
- `.data` section
- Information for debugger
- etc.

Relocation record

"TODO items" whose address this file still needs

- Any external label jumped to:
  - External label (including lib)
- etc.

Compiler/Assembler Example

```
gcc -O3 -S main.c main.s
```

```c
#include <stdio.h>
#include "add.h"
#include "max.h"
int main(void)
{
    int d=add(1234,4321);
    int m=max(1234,4321);
    printf("Result is %d\n",d+m);
    return 0;
}

int add(int x,int y)
{return x+y;}

int max(int x,int y)
{return x>y?x:y;}
```

Compiler/Assembler Example

`gcc -O3 **-c** main.c add.c max.c`

`objdump **-d** main.o (-x for symbol table & relocation records)`

```
#include <stdio.h>
#include "add.h"
#include "max.h"
int main(void)
{
    int d=add(1234,4321);
    int m=max(1234,4321);
    printf("Result is
%d\n",d+m);
    return 0;
}

int add(int x,int y)
{return x+y;}

int max(int x,int y)
{return x>y?x:y;}
```

```
main.o:     file format elf64-littleriscv
Disassembly of section .text.startup:
0000000000000000 <main>:
   0:1101       add sp,sp,-32
   2:e426       sd s1,8(sp)
   4:6485       lui s1,0x1
   6:0e148593   add a1,s1,225 # 10e1 <main+0x10e1>
   a:4d200513   li a0,1234
   e:ec06       sd ra,24(sp)
  10:e822       sd s0,16(sp)
  12:00000097   auipc ra,0x0
  16:000080e7   jalr ra # 12 <main+0x12>
  1a:842a       mv s0,a0
  1c:0e148593   adda1,s1,225
  20:4d200513   li a0,1234
  24:00000097   auipc ra,0x0
  28:000080e7   jalr ra # 24 <main+0x24>
  2c:00a405bb   addw a1,s0,a0
  30:00000537   lui a0,0x0
  34:00050513   mv a0,a0
  38:00000097   auipc ra,0x0
  3c:000080e7   jalr ra # 38 <main+0x38>
  40:60e2       ld ra,24(sp)
  42:6442       ld s0,16(sp)
  44:64a2       ld s1,8(sp)
  46:4501       li a0,0
  48:6105       add sp,sp,32
  4a:8082       ret
```

Compiler/Assembler Example

`gcc -O3 -c main.c add.c max.c`

```
#include <stdio.h>
#include "add.h"
#include "max.h"
int main(void)
{
    int d=add(1234,4321);
    int m=max(1234,4321);
    printf("Result is
%d\n",d+m);
    return 0;
}
```

```
int add(int x,int y)
{return x+y;}
```

```
int max(int x,int y)
{return x>y?x:y;}
```

`objdump -d max.o`

```
max.o:     file format elf64-littleriscv

Disassembly of section .text:

0000000000000000 <max>:
   0:87ae      mv     a5,a1
   2:00a5d363  bge    a1,a0,8 <.L2>
   6:87aa      mv     a5,a0
0000000000000008 <.L2>:
   8:0007851b  sext.w a0,a5
   c:8082      ret
```

Compiler/Assembler Example

```
gcc -O3 -c main.c add.c max.c
```

```
#include <stdio.h>
#include "add.h"
#include "max.h"
int main(void)
{
    int d=add(1234,4321);
    int m=max(1234,4321);
    printf("Result is
%d\n",d+m);
    return 0;
}


int add(int x,int y)
{return x+y;}


int max(int x,int y)
{return x>y?x:y;}
```

```
objdump -d add.o
```

```
add.o:     file format elf64-littleriscv

Disassembly of section .text:

0000000000000000 <add>:
   0:9d2d    addw a0,a0,a1
   2:8082    ret
```

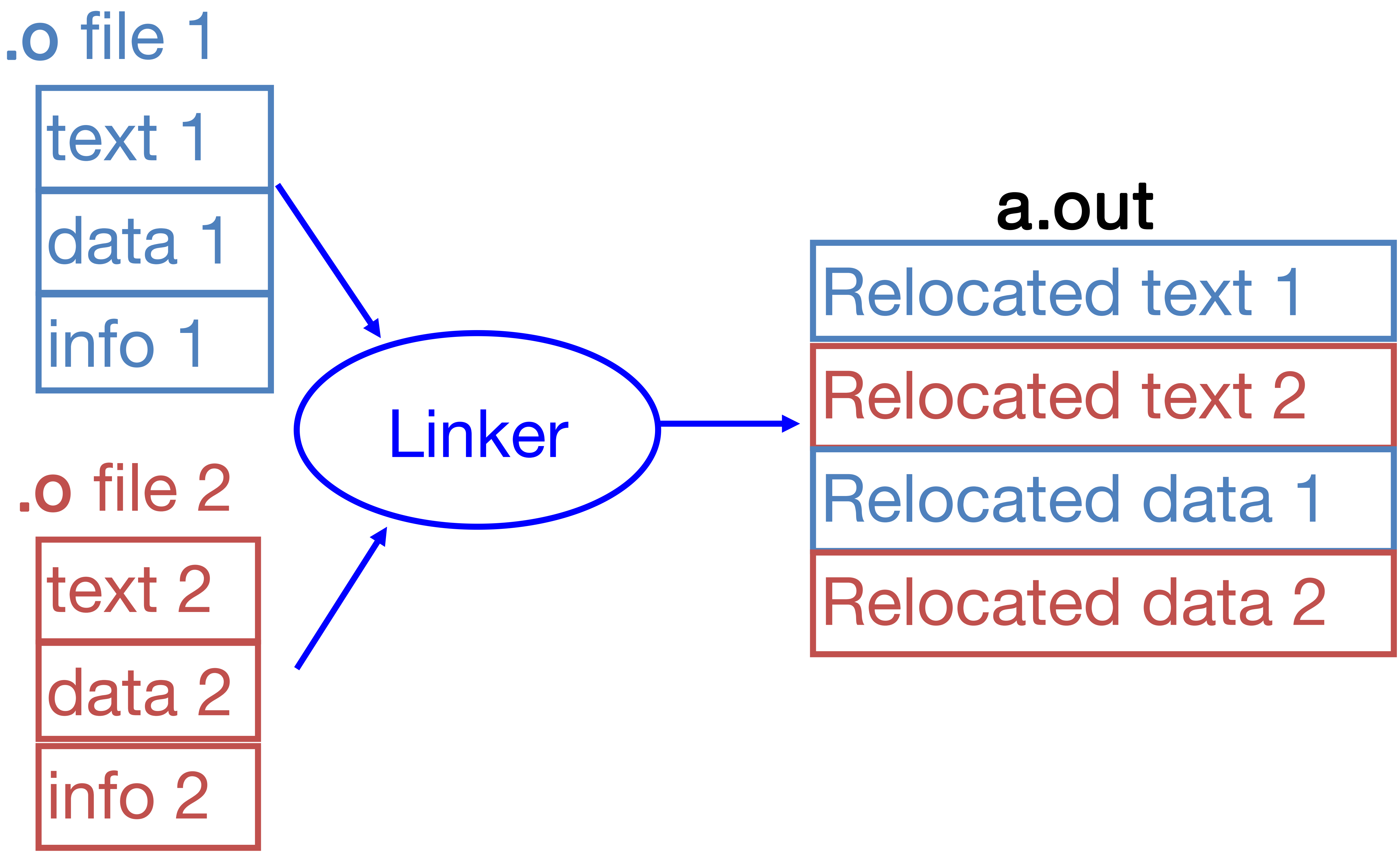
Summary: Object File Format

- 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 (by linker)
- symbol table: list of this file's labels and static data that can be referenced
- debugging information
- A standard format is ELF (except MS) http://www.skyfree.org/linux/references/ELF_Format.pdf

Linker (1/3)

- Input: object code files, information tables (e.g., <your C code>.o, libc.o for RISC-V)
- Output: executable code (e.g., a.out for RISC-V)
- 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
    - Linux source > 20 M lines of code!
  - Old name “Link Editor” from editing the “links” in jump and link instructions

Linker (2/3)


.o file 1
text 1
data 1
info 1
.o file 2
text 2
data 2
info 2
Linker
a.out
Relocated text 1
Relocated text 2
Relocated data 1
Relocated data 2

Linker (3/3)

- Step 1: Take text segment from each .o file and put them together; Take data segment from each .o file, put them together, and concatenate this onto end of text segments
- Step 2: Determine the addresses of data and instruction labels
- Step 3: Resolve references
  - Go through relocation records; handle each entry
  - That is, fill in all absolute addresses

Three Types of Addresses

- PC-Relative Addressing (`beq`, `bne`, `jal`)
  - Never need to relocate (PIC: position independent code)
- External Function Reference (usually `jal`)
  - Always relocate
- Static Data Reference (often `auipc/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`

Absolute Addresses in RISC-V

- Which instructions need relocation editing?
  - J-format: jump and link: ONLY for external jumps

`xxxxxxxxxxxxxxxxxxxx`	`rd`	`JAL`

  - I-,S- Format: Loads and stores to variables in static area, relative to global pointer

`xxxxxxx`	`rs2`	`gp`	`funct3`	`xxxxx`	`STORE`

`xxxxxxx`	`xxxxx`	`gp`	`funct3`	`rd`	`LOAD`

  - What about conditional branches?
    - Do not need editing
  - PC-relative addressing preserved even if code moves

Resolving References (1/2)

- 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, printf, malloc)
  - once absolute address is determined, fill in the machine code appropriately
- Output of linker: executable file containing text and data (plus header)

Static vs. Dynamic Linking

- What we've described is the traditional way: statically-linked approach
  - The 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)
  - It includes the entire library even if not all of it will be used
  - Executable is self-contained
- An alternative is dynamically linked libraries (DLL), common on Windows (.dll) & UNIX (.so) & MacOS (.dylib) platforms

en.wikipedia.org/wiki/Dynamic_linking

Dynamically linked libraries

- 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)
  - – 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
  - Thus "containers": We hate dependencies, so we are just going to ship around all the libraries and everything else as part of the 'application'

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

Loader Basics

- Input: Executable Code (e.g., a.out for RISC-V)
- Output: (program 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 tasks

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

Question

At what point in process are all the machine code bits generated for the following assembly instructions:

```
1) add x6, x7, x8
2) jal x1, fprintf
```

```
A: 1) & 2) After compilation
B: 1) After compilation,  2) After assembly
C: 1) After assembly,        2) After linking
D: 1) After assembly,       2) After loading
E: 1) After compilation,  2) After linking
```

Answer

At what point in process are all the machine code bits determined for the following assembly instructions:

```
1) add x6, x7, x8
2) jal x1, fprintf
```

```
C: (1) After assembly, (2) After linking
```

In Conclusion…

- 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.