CS 61C:
Great Ideas in Computer Architecture
Running a Program - CALL (Compiling, Assembling, Linking, and Loading)

Instructors:

Krste Asanović & Randy H. Katz

http://inst.eecs.Berkeley.edu/~cs61c/fa17

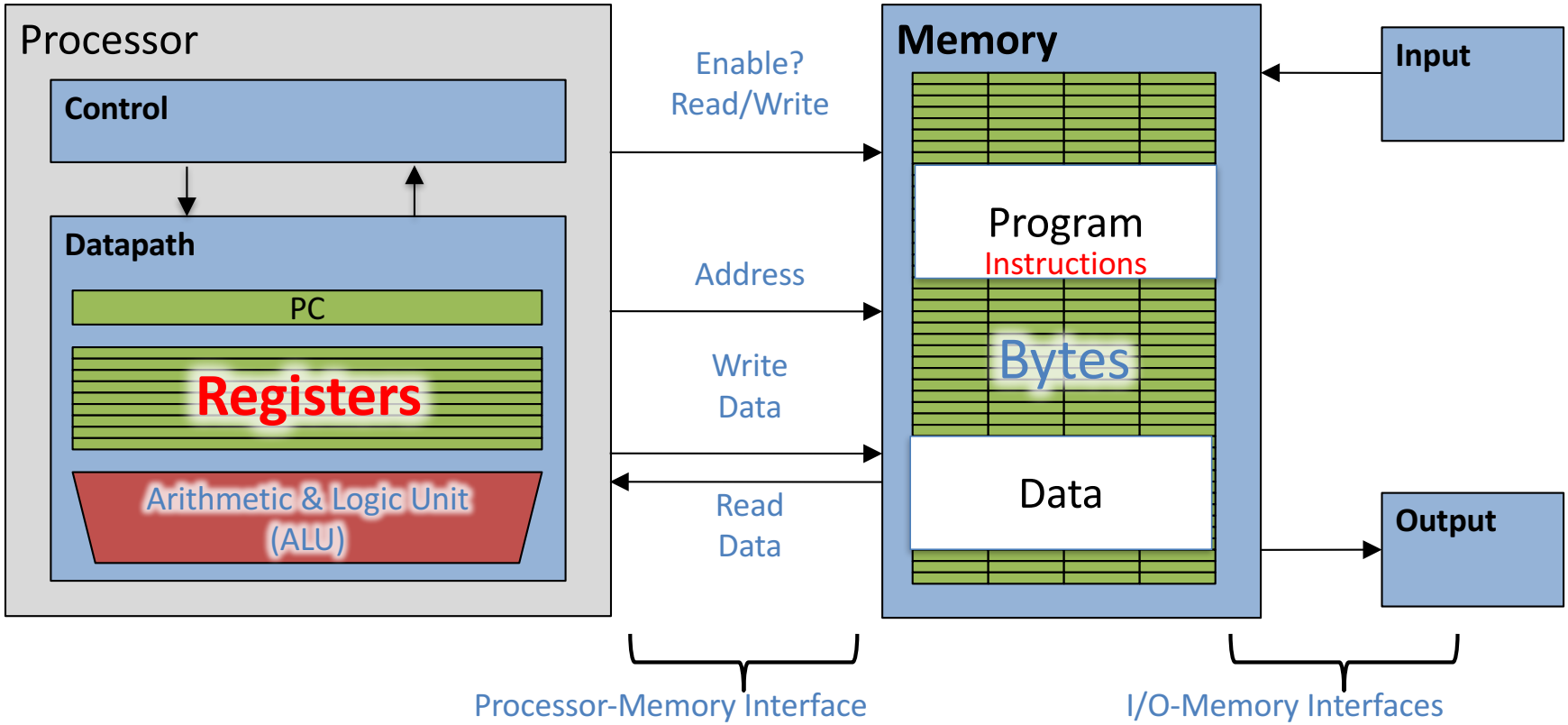
Outline

- Review Instruction Formats + J/JAL
- Multiply and Divide
- Interpretation vs. Translation
- Assembler
- Linker
- Loader
- And in Conclusion …

Outline

- Review Instruction Formats + J/JAL
- Multiply and Divide
- Interpretation vs. Translation
- Assembler
- Linker
- Loader
- And in Conclusion …

Review: Components of a Computer



Review: RISC-V Instruction Formats

31 30 25	24 21 20	19 15	14 12	11 8	7 6 0	
funct7	rs2	rs1	funct3	rd	opcode	R-type

rd = rs1 OP rs2

imm[11:0]		rs1	funct3	rd	opcode	I-type

rd = rs1 OP Immediate; Load rd from Memory(rs1 + Immediate); *JALR (rd = PC + 4, PC = rs1 + Immediate)*

imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type

Store rs2 to Memory(rs1 + Immediate)

imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode	B-type

Branch if (rs1 condition rs2) is true to Memory(PC + Immediate*2), i.e., PC = PC + Immediate*2

imm[31:12]	rd	opcode	U-type

Upper "Long" Immediate (AUIPC, LUI): PC or rd = {imm, 12b'0}

imm[20]	imm[10:1]	imm[11]	imm[19:12]	rd	opcode	J-type

JAL to Memory (PC + Immediate*2), i.e., rd = PC + 4; PC = PC + immediate*2

J-Format for Jump Instructions (JAL)

31	30 … 21	20	19 … 12	11 … 7	6 … 0
imm[20]	imm[10:1]	imm[11]	imm[19:12]	rd	opcode
1	10	1	8	5	7
	offset[20:1]			dest	JAL

- JAL saves PC+4 in register rd (the return address)
  - Assembler "`j`" jump is pseudo-instruction, uses JAL but sets rd=x0 to discard return address
- Set PC = PC + offset (PC-relative jump: offset = signed immediate * 2)
- Target somewhere within $\pm 2^{19}$ locations, 2 bytes apart
  - $\pm 2^{18}$ 32-bit instructions, $\pm 2^{20}$ bytes
- Immediate encoded optimized similarly to branch instruction to reduce hardware cost

Uses of JAL

```
# j pseudo-instruction
j Label = jal x0, Label # Discard return address

# Call function within 2^18 instructions of PC
jal ra, FuncName
```

JALR Instruction (I-Format)

31 – 20	19 – 15	14 – 12	11 – 7	6 – 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
offset[11:0]	base	0	dest	JALR

- `JALR rd, rs, immediate`
  - Writes `PC+4` to `rd` (return address)
  - Sets `PC` = `rs` + `immediate` (12 bit, sign extended)
  - Uses same immediates as arithmetic and loads
    - *Unlike* branches, ***no*** multiplication by two before adding to rs to form the new PC
    - *Byte offset NOT halfword offset as in branches and JAL*

Uses of JALR

```
# ret and jr pseudo-instructions
ret = jr ra = jalr x0, ra, 0

# Call function at any 32-bit absolute address
lui x1, <hi20bits>
jalr ra, x1, <lo12bits>

# Jump PC-relative with 32-bit offset
auipc x1, <hi20bits>
jalr x0, x1, <lo12bits>
```

Pseudo Instructions

MNEMONIC	NAME	DESCRIPTION	USES
beqz	Branch = zero	if(R[rs1]==0) PC=PC+{imm,1b'0}	beq
bnez	Branch ≠ zero	if(R[rs1]!=0) PC=PC+{imm,1b'0}	bne
fabs.s,fabs.d	Absolute Value	F[rd] = (F[rs1]< 0) ? –F[rs1] : F[rs1]	fsgnx
fmv.s,fmv.d	FP Move	F[rd] = F[rs1]	fsgnj
fneg.s,fneg.d	FP negate	F[rd] = –F[rs1]	fsgnjn
j	Jump	PC = {imm,1b'0}	jal
jr	Jump register	PC = R[rs1]	jalr
la	Load address	R[rd] = address	auipc
li	Load imm	R[rd] = imm	addi
mv	Move	R[rd] = R[rs1]	addi
neg	Negate	R[rd] = –R[rs1]	sub
nop	No operation	R[0] = R[0]	addi
not	Not	R[rd] = ~R[rs1]	xori
ret	Return	PC = R[1]	jalr
seqz	Set = zero	R[rd] = (R[rs1]== 0) ? 1 : 0	sltiu
snez	Set ≠ zero	R[rd] = (R[rs1]!= 0) ? 1 : 0	sltu

Valid in assembly language but not in machine language
(i.e., maps to multiple machine language instructions)

Outline

- Review Instruction Formats + J/JAL
- Multiply and Divide
- Interpretation vs. Translation
- Assembler
- Linker
- Loader
- And in Conclusion …

Integer Multiplication (1/3)

- Paper and pencil example (unsigned):

```
Multiplicand   1000    8
Multiplier   x 1001    9
```

```
                  72
```

- m bits x n bits = m + n bit product

Integer Multiplication (1/3)

- Paper and pencil example (unsigned):

```
Multiplicand   1000    8
Multiplier   x 1001    9
             ------
               1000

                      72
```

- m bits x n bits = m + n bit product

Integer Multiplication (1/3)

- Paper and pencil example (unsigned):

```
Multiplicand   1000    8
Multiplier   x 1001    9
             ------
               1000
              0000

                      72
```

- m bits x n bits = m + n bit product

Integer Multiplication (1/3)

- Paper and pencil example (unsigned):

```
Multiplicand   1000    8
Multiplier   x 1001    9
             ------
               1000
              0000
             0000

                      72
```

- m bits x n bits = m + n bit product

Integer Multiplication (1/3)

- Paper and pencil example (unsigned):

```
Multiplicand   1000    8
Multiplier   x 1001    9
               1000
              0000
             0000
           + 1000
                      72
```

- m bits x n bits = m + n bit product

Integer Multiplication (1/3)

- Paper and pencil example (unsigned):

```
Multiplicand   1000    8
Multiplier   x 1001    9
               1000
              0000
             0000
           + 1000
            01001000   72
```

- m bits x 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
- Syntax of Multiplication (signed):
  - `MUL` performs an 32-bit×32-bit multiplication and places the lower 32 bits in the destination register
  - `MULH` performs the same multiplication but returns the upper 32 bits of the full 2×32-bit product
  - If 64-bit product is required, then the recommended code sequence is:

```
MULH rdh, rs1, rs2
MUL  rdl, rs1, rs2
```

    (source register specifiers must be in same order and `rdh` cannot be the same as `rs1` or `rs2`)

Integer Multiplication (3/3)

- Example:
  - in C: `a = b * c;  # a should be declared long long`
  - in RISC-V:
    - Let `b` be `t2`; let `c` be `t3`; and let `a` be `t0` and `t1` (since it may be up to 64 bits)

```
# upper half of
# product into $s0
# lower half of
# product into $s1
```

Integer Multiplication (3/3)

- Example:
  - in C: `a = b * c;` `# a should be declared long long`
  - in RISC-V:
    - Let `b` be `t2`; let `c` be `t3`; and let `a` be `t0` and `t1` (since it may be up to 64 bits)

```
mulh t0,t2,t3          # upper half of
                       # product into $s0
                       # lower half of
                       # product into $s1
```

Integer Multiplication (3/3)

- Example:
  - in C: `a = b * c;` `# a should be declared long long`
  - in RISC-V:
    - Let `b` be `t2`; let `c` be `t3`; and let `a` be `t0` and `t1` (since it may be up to 64 bits)

```
mulh t0,t2,t3        # upper half of
                     # product into $s0
mul  t1,t2,t3        # lower half of
                     # product into $s1
```

Integer Division (1/2)

- Paper and pencil example (unsigned):

```
                  ___________ Quotient
Divisor 1000|1001010    Dividend
```

- Dividend = Quotient x Divisor + Remainder

Integer Division (1/2)

- Paper and pencil example (unsigned):

```
                    1        Quotient
Divisor 1000|1001010         Dividend
           -1000
               1
```

- Dividend = Quotient x Divisor + Remainder

Integer Division (1/2)

- Paper and pencil example (unsigned):

```
                    10        Quotient
Divisor 1000|1001010          Dividend
           -1000
               10
```

- Dividend = Quotient x Divisor + Remainder

Integer Division (1/2)

- Paper and pencil example (unsigned):

```
                    100      Quotient
Divisor 1000|1001010         Dividend
           -1000
               10
               101
```

- Dividend = Quotient x Divisor + Remainder

Integer Division (1/2)

- Paper and pencil example (unsigned):

```
                    1001    Quotient
Divisor 1000|1001010    Dividend
           -1000
               10
               101
               1010
```

- Dividend = Quotient x Divisor + Remainder

Integer Division (1/2)

- Paper and pencil example (unsigned):

```
                    1001    Quotient
Divisor 1000|1001010    Dividend
           -1000
               10
               101
               1010
              -1000
```

- Dividend = Quotient x Divisor + Remainder

Integer Division (1/2)

- Paper and pencil example (unsigned):

```
                    1001    Quotient
Divisor 1000|1001010    Dividend
           -1000
               10
               101
               1010
              -1000
                 10 Remainder
              (or Modulo result)
```

- Dividend = Quotient x Divisor + Remainder

Integer Division (2/2)

- Syntax of Division (signed):
  - `DIV` performs signed integer division of 32 bits by 32 bits. REM provides the remainder of the corresponding division operation
  - If the quotient and remainder are required from the same division, the recommended code sequence is:
    ```
    DIV rdq, rs1, rs2
    REM rdr, rs1, rs2
    ```
    (rdq cannot be the same as rs1 or rs2)
- Implements C division (`/`) and modulo (`%`)
- Example in C: `a = c / d;` `b = c % d;`
- In RISC-V: `a↔t0; b↔t1; c↔t2; d↔t3`

```
div t0,t2,t3    # a=c/d
rem t1,t2,t3    # b=c%d
```

Peer Question

Which of the following places the address of LOOP in t1?

```
1) la t2, LOOP
   lw t1, 0(t2)

2)        jal LOOP
   LOOP: add t1, ra, x0

3) la t1, LOOP
```

1	2	3
T,	T,	T
T,	T,	F
F,	T,	T
F,	T,	F

Peer Question

Which of the following places the address of LOOP in t1?

```
1) la t2, LOOP
   lw t1, 0(t2)

2)        jal LOOP
   LOOP: add t1, ra, x0

3) la t1, LOOP
```

1	2	3
T,	T,	T
T,	T,	F
F,	T,	T
F,	T,	F


SCHOOL IS HELL
OR
3 CREDITS SHY OF GRADUATIN'
THE 9 TYPES OF COLLEGE TEACHERS
LOOKING FOR LEISURE?
ROCKS FOR JOCKS 101
FOLLOW THE ATHLETES.
THE STEADY DRONER
NONSTOP NASAL MONOTONE
ADVANTAGES: ORGANIZED, PREPARED.
DRAWBACKS: IS ANNOYED BY SNORING.
WARNING: THIS IS LIFE.
THE DISDAINFUL TEACHING ASSISTANT
HUMPH
ADVANTAGES: NONE.
DRAWBACKS: VINDICTIVE, MERCILESS.
WARNING: IF YOU HANG AROUND LONG ENOUGH, YOU COULD TURN INTO ONE.
THE MIGHTY FAMOUS BIGSHOT
?
ADVANTAGES: EXCITING, IMPRESSIVE.
DRAWBACKS: DISDAINFUL TEACHING ASSISTANT DOES THE TEACHING.
WARNING: BEWARE OF BAIT-AND-SWITCH.
THE BELOVED BABBLING GRANDPA WITH TENURE
SO THEN... SO THEN... WHERE ARE MY GLASSES?
ADVANTAGES: EASY, RELAXING.
DRAWBACKS: CRANKY, OLD-FASHIONED.
WARNING: MAY FLUNK YOU IF HIS BOWELS ARE ACTING UP THAT DAY.
THE GENIUS FROM ANOTHER DIMENSION
THIS THEORY GOES AGAINST ALL LOGIC, HISTORY, AND EXPERT OPINION, BUT I TELL YOU IT IS TRUE.
ADVANTAGES: LOONY, ENTERTAINING.
DRAWBACKS: LOONY, SCARY.
WARNING: MAY TURN YOU INTO BELIEVER.
OL' GLOOM & DOOM
THE THOUGHT OF SUICIDE IS A POWERFUL SOLACE: BY MEANS OF IT ONE GETS THROUGH MANY A BAD NIGHT.
ADVANTAGES: A CLOSE-UP GLIMPSE OF A SOUL IN TORMENT.
DRAWBACKS: MAKES YOU THINK ABOUT CREEPY STUFF.
WARNING: THE JOKE GETS OLD AFTER AWHILE.
THE SINGLE-THEORY-TO-EXPLAIN-EVERYTHING MANIAC
THE NATION THAT CONTROLS MAGNESIUM CONTROLS THE UNIVERSE!!!
ADVANTAGES: EASY TO PLEASE.
DRAWBACKS: PARROTING ISN'T LEARNING.
WARNING: THEORY MAY BE CORRECT.
THE INCOMPREHENSIBLE BRILLIANT FOREIGNER
DE VONSTIG IN DER SCHLONDO VEN--HOW YOU SAY--"GROOVY"--KREBBLER OSTEN VON---
ADVANTAGES: HAS A GREAT REPUTATION.
DRAWBACKS: NO TRANSLATORS AVAILABLE.
WARNING: WILL DRIVE YOU INSANE.
NICE LITTLE NOBODY
I SEEM TO HAVE FORGOTTEN MY LECTURE NOTES AGAIN.
ADVANTAGES: EASY TO IGNORE.
DRAWBACKS: MAKES YOU WONDER WHY YOU'RE IN COLLEGE.
WARNING: THIS IS LIFE.

Outline

- Review Instruction Formats + J/JAL
- Multiply and Divide
- Interpretation vs. Translation
- Assembler
- Linker
- Loader
- And in Conclusion …

Levels of Representation/Interpretation

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

Compiler

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

Assembler

Machine Language Program (RISC-V)

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

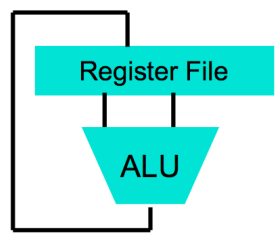
```
lw   $t0, 0($2)
lw   $t1, 4($2)
sw   $t1, 0($2)
sw   $t0, 4($2)
```

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

```
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
```

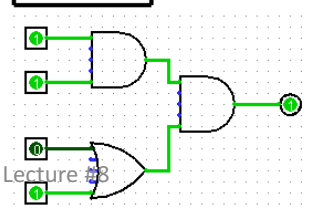
Machine Interpretation

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



Architecture Implementation

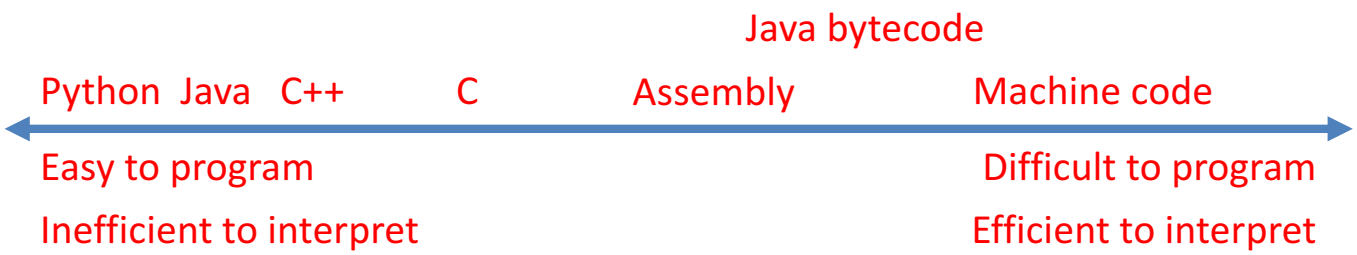
Logic Circuit Description (Circuit Schematic Diagrams)



+ How to take a program and run it

Language Execution Continuum

- Interpreter is a program that executes other programs



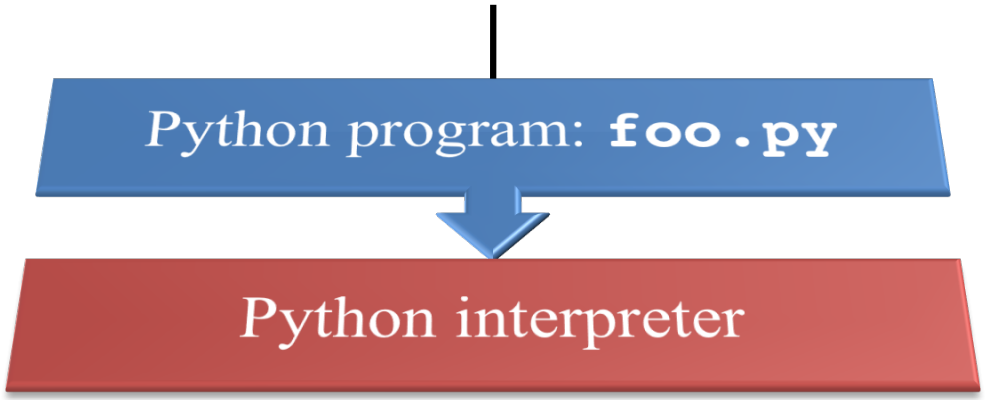
- Language translation gives us another option
- In general, we interpret a high-level language when efficiency is not critical and translate to a lower-level language to increase performance

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

Interpretation

- For example, consider a Python program **`foo.py`**



- Python interpreter is just a program that reads a python program and performs the functions of that python program

Interpretation

- WHY interpret machine language in software?
- VENUS (Lab #3), Project #1: RISC-V simulator useful for learning/debugging
- E.g., Apple Macintosh conversion
  - Switched from Motorola 680x0 instruction architecture to PowerPC (before x86)
  - 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)

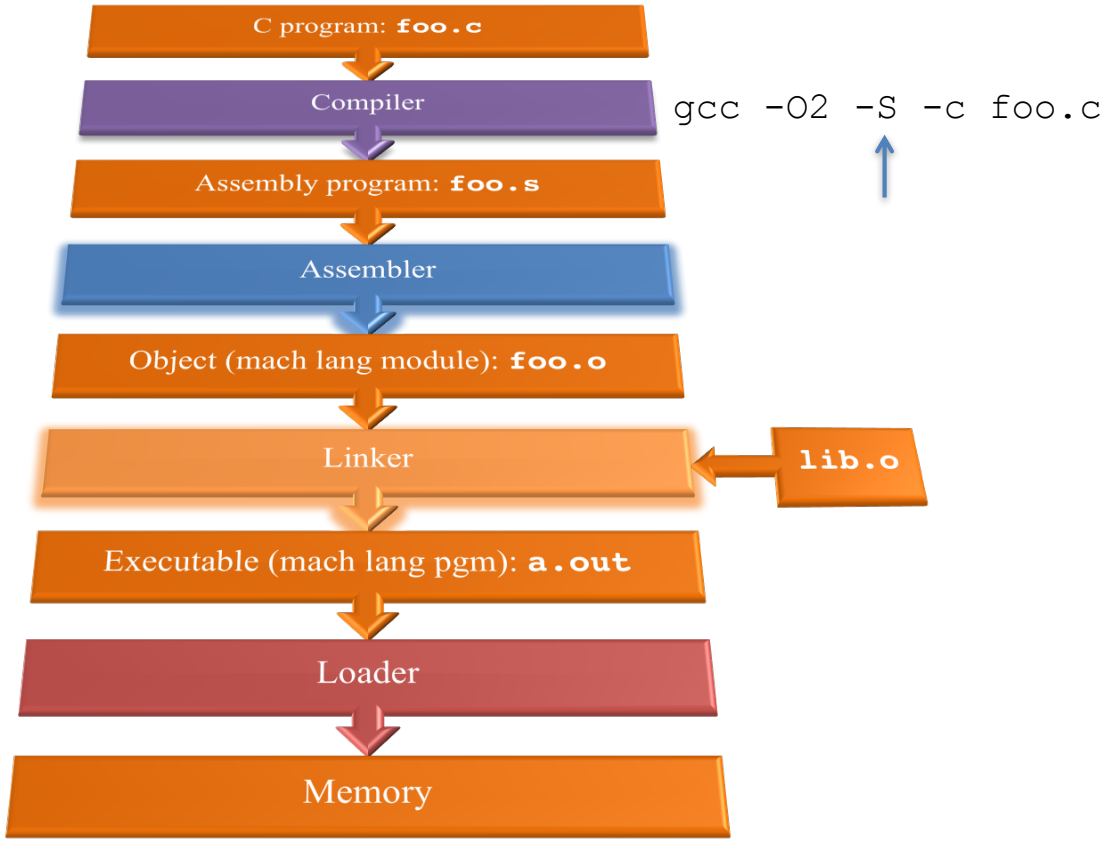
Interpretation vs. Translation? (1/2)

- Generally easier to write interpreter
- Interpreter closer to high-level, so can give better error messages (e.g., VENUS, Project #1)
  - Translator reaction: add extra information to help debugging (line numbers, names)
- Interpreter slower (10x?), code smaller (2x?)
- 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
- Translation/compilation helps "hide" the program "source" from the users:
  - One model for creating value in the marketplace (e.g., Microsoft keeps all their source code secret)
  - Alternative model, "open source", creates value by publishing the source code and fostering a community of developers.

Steps in Compiling a C Program



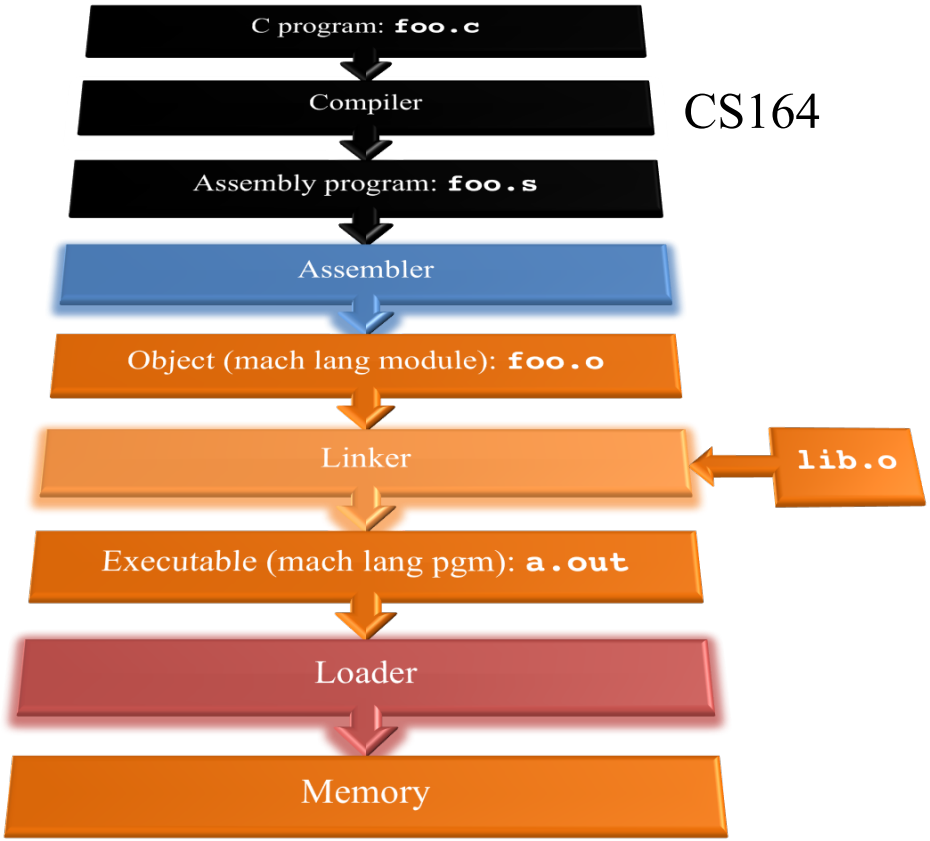
Compiler

- Input: High-Level Language Code (e.g., **`foo.c`**)
- Output: Assembly Language Code
  (e.g., **`foo.s`** for RISC-V)
- Note: Output *may* contain pseudo-instructions
- Pseudo-instructions: instructions that assembler understands but not in machine
  For example (move t2 to t1):
  - **`mv t1,t2`** ⇒ **`addi t1,t2,0`**

Outline

- Review Instruction Formats + J/JAL
- Multiply and Divide
- Interpretation vs. Translation
- Assembler
- Linker
- Loader
- And in Conclusion …

Where Are We Now?



Assembler

- Input: Assembly Language Code (includes pseudo ops) (e.g., **`foo.s`** for RISC-V)
- Output: Object Code, information tables (true assembly only) (e.g., **`foo.o`** for RISC-V)
- Reads and Uses Directives
- Replace Pseudo-instructions
- Produce Machine Language
- Creates Object File

Assembler Directives
(See *RISCV Reader, Chapter 3*)

- 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:

  ```
  mv t0, t1
  ```

Pseudo-instruction Replacement

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

Pseudo:	Real:
`mv t0, t1`	`addi t0,t1,0`
`neg t0, t1`	`sub t0, zero, t1`
`li t0, imm`	`addi t0, zero, imm`
`not t0, t1`	`xori t0, t1, -1`
`beqz t0, loop`	`beq t0, zero, loop`

`la t0, str`	`lui t0, str[31:12]`
	`addi t0, t0, str[11:0]` OR
	`auipc t0, str[31:12]`
	`addi  t0, t0, str[11:0]`

DON'T FORGET: sign extended immediates + Branch immediates count halfwords)

STATIC Addressing

PC-Relative Addressing

Peer Instruction

Which of the following is a correct assembly language sequence sequence for the pseudoinstruction: la t1, FOO?*

A:
```
ori   t1, 0xABCD0
addi  t1, 0x124
```

B:
```
ori   t1, 0x124
lui   t1, 0xABCD0
```

C:
```
lui   t1, 0xD0124
ori   t1, 0xABC
```

D:
```
lui   t1, 0xABCD0
addi  t1, 0x124
```

*Assume the address of FOO is 0xABCD0124

Peer Instruction

Which of the following is a correct assembly language sequence sequence for the pseudoinstruction: la t1, FOO?*

```
A: ori   t1, 0xABCD0
   addi  t1, 0x124

B: ori   t1, 0x124
   lui   t1, 0xABCD0

C: lui   t1, 0xD0124
   ori   t1, 0xABC

D: lui   t1, 0xABCD0
   addi  t1, 0x124
```

*Assume the address of FOO is 0xABCD0124

Producing Machine Language (1/3)

- Simple Case
  - Arithmetic, Logical, Shifts, and so on
  - All necessary info is within the instruction already
- What about Branches and Jumps?
  - PC-Relative (e.g., **`beq/bne`** and **`jal`**)
  - So once pseudo-instructions are replaced by real ones, we know by how many instructions to branch/jump over
- 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:

2 words forward (4 halfwords)
3 words forward (6 halfwords)
2 words **back** (4 halfwords)
3 words **back** (6 halfwords)

```
        addi t2,  zero,  9   # t2 = 9
L1:     bge  zero, t2, L2    # 0 >= t2? Exit!
        addi t2,  t2, -1     # 0 < t2; t2--
        j  L1                # Go to L1
L2:
```

  - Solved by taking two passes over the program
    - First pass remembers position of labels
    - Second pass uses label positions to generate code

Producing Machine Language (3/3)

- What about PC-relative jumps (**`jal`**) and branches (**`beq`**, **`bne`**)?
  - **`j offset`** *pseudo instruction* expands to **`JAL zero, offset`**
  - Just count the number of instruction *halfwords* between target and jump to determine the offset: *position-independent code (PIC)*
- What about references to static data?
  - **`la`** gets broken up into **`lui`** and **`addi`** (use **auipc/addi** for PIC)
  - These 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" whose address this file needs
  What are they?
  - Any absolute label jumped to: **`jal, jalr`**
    - Internal
    - External (including lib files)
    - Such as the **`la`** instruction
      E.g., for **`jalr`** base register
  - Any piece of data in static section
    - Such as the **`la`** instruction
      E.g., for **`lw/sw`** base register

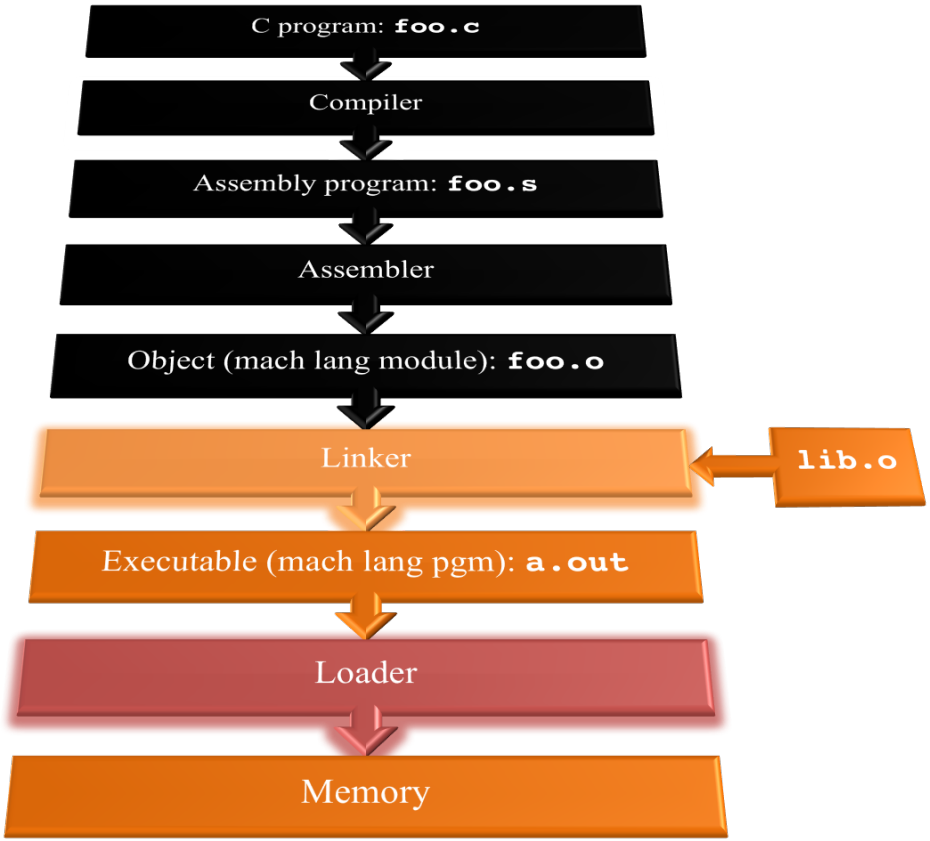
Object File Format

- 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 MS)
  `http://www.skyfree.org/linux/references/ELF_Format.pdf`

Outline

- Review Instruction Formats + J/JAL
- Multiply and Divide
- Interpretation vs. Translation
- Assembler
- Linker
- Loader
- And in Conclusion …

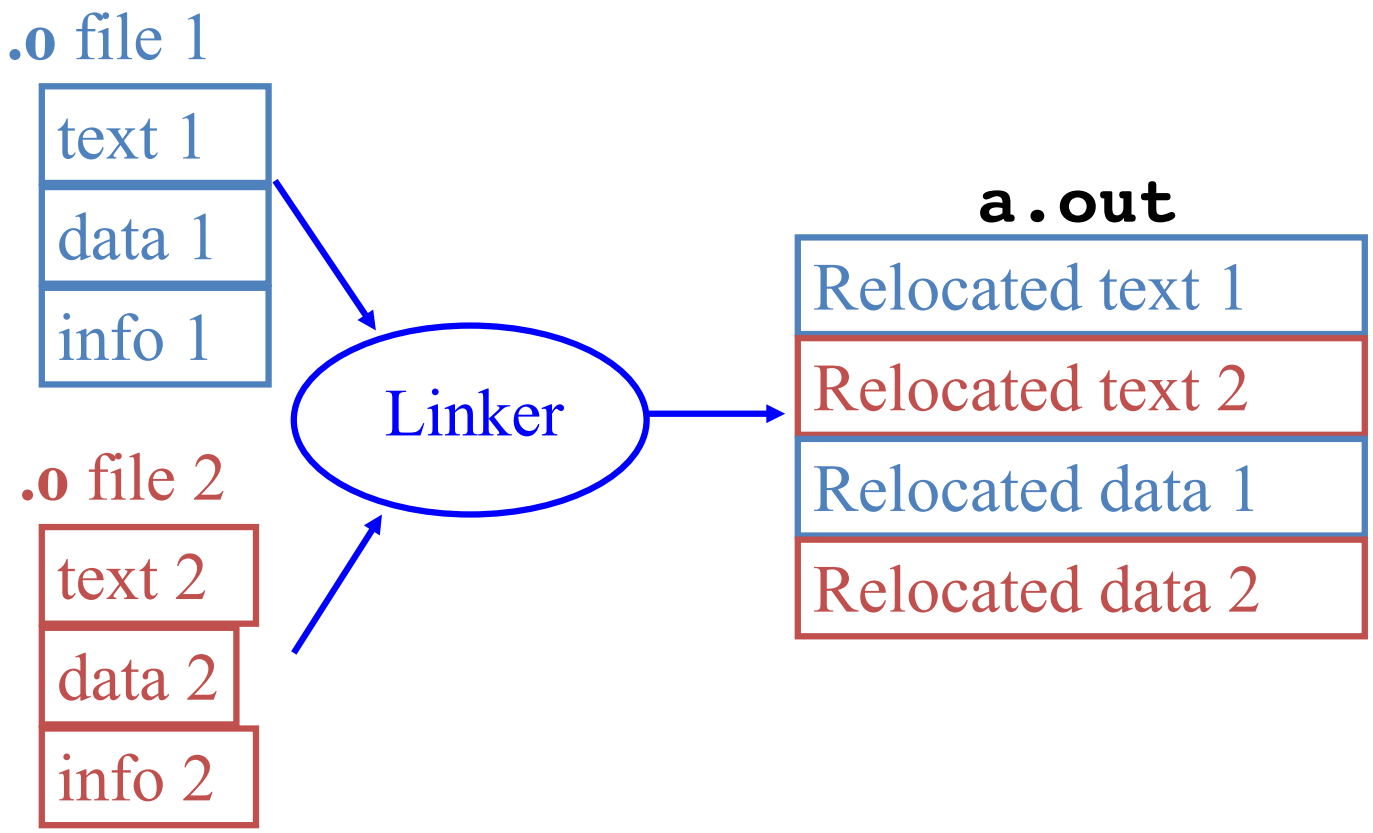
Where Are We Now?



Linker (1/3)

- Input: Object code files, information tables (e.g., `foo.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
- 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
  - I.e., fill in all absolute addresses

Four Types of Addresses

- PC-Relative Addressing (**`beq`**, **`bne`**, **`jal; auipc/addi`**)
  - Never need to relocate (PIC: position independent code)
- Absolute Function Address (**`auipc/jalr`**)
  - Always relocate
- External Function Reference (**`auipc/jalr`**)
  - Always relocate
- Static Data Reference (often **`lui/addi`**)
  - Always relocate

Absolute Addresses in RISC-V

- Which instructions need relocation editing?
  - J-format: jump/jump and link

xxxxx	rd	jal

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

xxx		gp		rd	lw
xx	rs1	gp		x	sw

  - What about conditional branches?

xx	rs1	rs2		x	beq bne

  - PC-relative addressing preserved even if code moves

Resolving References (1/2)

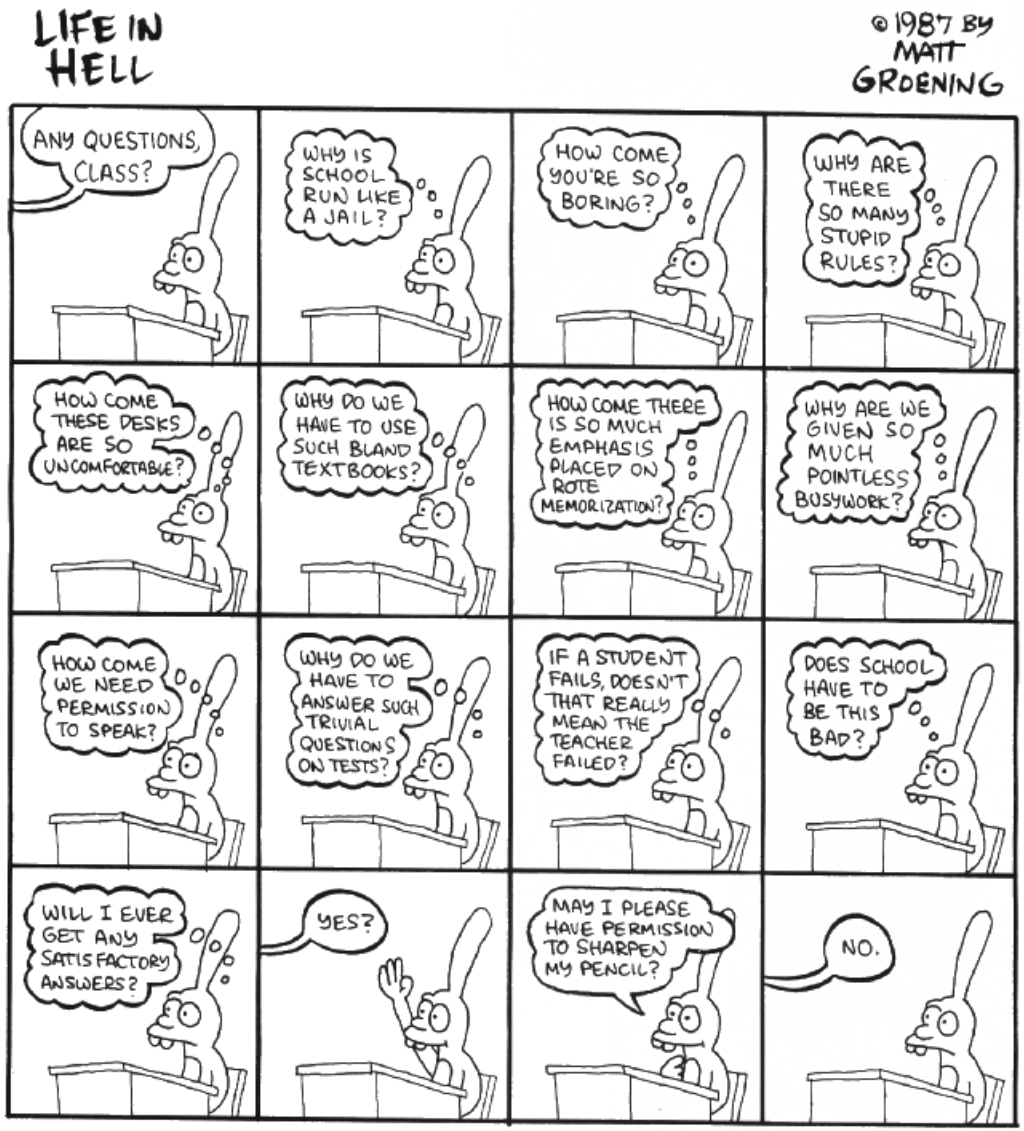
- Linker assumes first word of first text segment is at address **`0x10000`** for RV32.
  - (More later when we study "virtual memory")
- Linker knows:
  - Length of each text and data segment
  - Ordering of text and data segments
- Linker calculates:
  - Absolute address of each label to be jumped to (internal or external) 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 (e.g., for `printf`)
  - Once absolute address is determined, fill in the machine code appropriately
- Output of linker: executable file containing text and data (plus header)

Administrivia

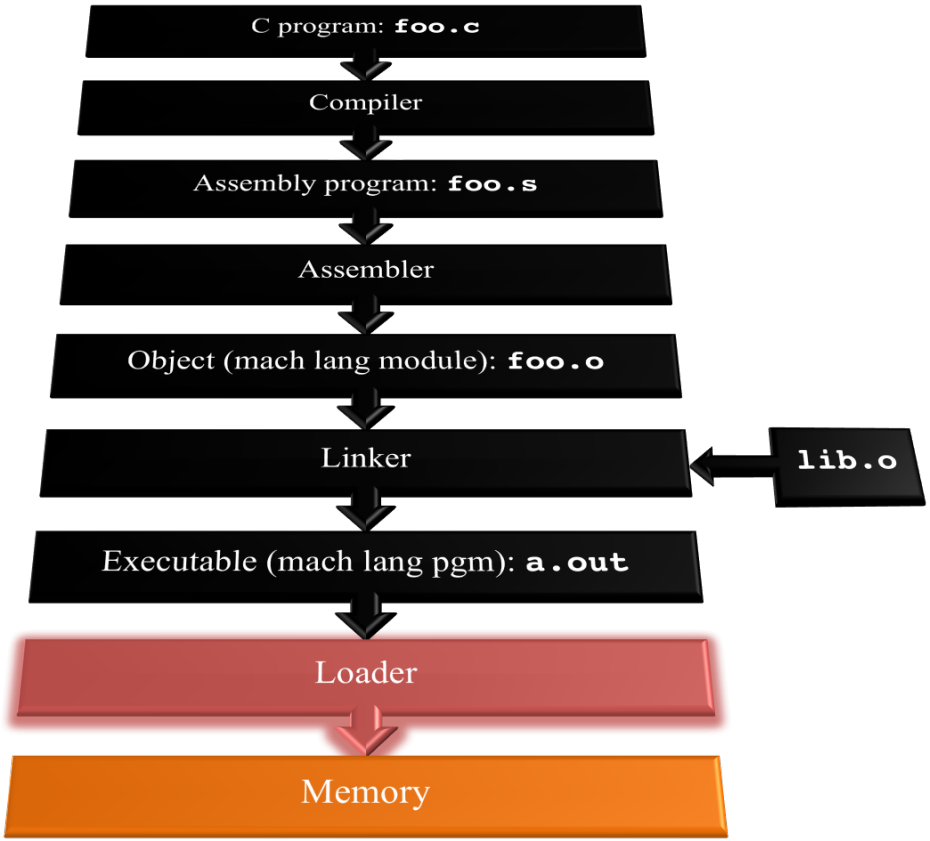
- Midterm #1 in 1 week: September 26!
  - IN CLASS! 8-9:30 AM Wheeler Auditorium + five Spill Rooms based on class login
  - Number Representations, C (pointers, arrays, strings), mapping C to RISC-V Assembly (and vice versa), NO MACHINE LANGUAGE ON THIS EXAM
  - Single double sided 8.5” x 11” cheat sheet + we give you the RISC-V Green Card
  - Review session Saturday, September 23: **Time and place TBD**
  - DSP students: please make sure we know about your special accommodations (contact TA Steven Ho)


LIFE IN HELL
© 1987 BY MATT GROENING
ANY QUESTIONS, CLASS?
WHY IS SCHOOL RUN LIKE A JAIL?
HOW COME YOU'RE SO BORING?
WHY ARE THERE SO MANY STUPID RULES?
HOW COME THESE DESKS ARE SO UNCOMFORTABLE?
WHY DO WE HAVE TO USE SUCH BLAND TEXTBOOKS?
HOW COME THERE IS SO MUCH EMPHASIS PLACED ON ROTE MEMORIZATION?
WHY ARE WE GIVEN SO MUCH POINTLESS BUSYWORK?
HOW COME WE NEED PERMISSION TO SPEAK?
WHY DO WE HAVE TO ANSWER SUCH TRIVIAL QUESTIONS ON TESTS?
IF A STUDENT FAILS, DOESN'T THAT REALLY MEAN THE TEACHER FAILED?
DOES SCHOOL HAVE TO BE THIS BAD?
WILL I EVER GET ANY SATISFACTORY ANSWERS?
YES?
MAY I PLEASE HAVE PERMISSION TO SHARPEN MY PENCIL?
NO.

Outline

- Review Instruction Formats + J/JAL
- Multiply and Divide
- Interpretation vs. Translation
- Assembler
- Linker
- Loader
- And in Conclusion …

Where Are We Now?



Loader Basics

- Input: Executable Code (e.g., **`a.out`** for RISC-V)
- 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 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 + 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

Peer Instruction

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`

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

Peer Instruction

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`

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

Example C Program: `Hello.c`

```
#include <stdio.h>
int main()
{
  printf("Hello, %s\n", "world");
  return 0;
}
```

Compiled `Hello.c`: `Hello.s`

```
.text                                   # Directive: enter text section
  .align 2                              # Directive: align code to 2^2 bytes
  .globl main                           # Directive: declare global symbol main
main:                                   # label for start of main
  addi sp,sp,-16                        # allocate stack frame
  sw   ra,12(sp)                        # save return address
  lui  a0,%hi(string1)                  # compute address of
  addi a0,a0,%lo(string1)               #   string1
  lui  a1,%hi(string2)                  # compute address of
  addi a1,a1,%lo(string2)               #   string2
  call printf                           # call function printf
  lw   ra,12(sp)                        # restore return address
  addi sp,sp,16                         # deallocate stack frame
  li   a0,0                             # load return value 0
  ret                                   # return
  .section .rodata                      # Directive: enter read-only data section
  .balign 4                             # Directive: align data section to 4 bytes
string1:                                # label for first string
  .string "Hello, %s!\n"                # Directive: null-terminated string
string2:                                # label for second string
  .string "world"                       # 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: 00000097 auipc ra,0x0      # addr placeholder
1c: 000080e7 jalr ra           # addr placeholder
20: 00c12083 lw ra,12(sp)
24: 01010113 addi sp,sp,16
28: 00000513 addi a0,a0,0
2c: 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
```

LUI/ADDI Address Calculation in RISC-V

Target address of <string1> is **`0x00020 A10`**

Instruction sequence `LUI 0x00020`, `ADDI 0xA10` does not quite work because immediates in RISC-V are sign extended (and `0xA10` has a 1 in the high order bit)!

`0x00020 000 + 0xFFFFF A10 = 0x0001F A10` (Off by 0x00001 000)

So we get the right address if we calculate it as follows:

`(0x00020 000 + 0x00001 000) + 0xFFFFF A10 = 0x00020 A10`

What is `0xFFFFF A10`?

Twos complement of `0xFFFFF A10` = `0x00000 5EF + 1` = `0x00000 5F0` = 1520_{ten}

So `0xFFFFF A10` = -1520_{ten}

Instruction sequence `LUI 0x00021`, `ADDI -1520` calculates **`0x00020 A10`**

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

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

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

Dynamically Linked Libraries

- Prevailing approach to dynamic linking uses machine code as the “lowest common denominator”
  - Linker does not use information about how the program or library was compiled (i.e., what compiler or language)
  - Can be described as “linking at the machine code level”
  - This isn’t the only way to do it ...

Outline

- Review Instruction Formats + J/JAL
- Multiply and Divide
- Interpretation vs. Translation
- Assembler
- Linker
- Loader
- And in Conclusion …

And 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