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

Processor
- Control
- Datapath
  - PC
  - Registers
  - Arithmetic & Logic Unit (ALU)

Memory
- Program Instructions
- Bytes
- Data

Input
- Enable?
- Read/Write
- Address
- Write Data
- Read Data

Output
- Processor-Memory Interface
- I/O-Memory Interfaces

9/19/17
Fall 2017 - Lecture #8
### Review: RISC-V Instruction Formats

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-30</td>
<td>funct7</td>
<td>rs2</td>
<td>rd = rs1 OP rs2</td>
</tr>
<tr>
<td>25-24</td>
<td></td>
<td>rs1</td>
<td></td>
</tr>
<tr>
<td>21-20</td>
<td>funct3</td>
<td>rd</td>
<td></td>
</tr>
<tr>
<td>19-15</td>
<td></td>
<td>opcode</td>
<td></td>
</tr>
<tr>
<td>12-11</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8-7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6-0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11-10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### R-type

- **(rd = rs1 OP rs2)**
- **rd = rs1 OP Immediate**; Load rd from Memory(rs1 + Immediate);
- **JALR (rd = PC + 4, PC = rs1 + Immediate)**

#### I-type

- **rd = rs1 OP Immediate**; Load rd from Memory(rs1 + Immediate);
- **JAL to Memory (PC + Immediate*2), i.e., rd = PC + 4; PC = PC + immediate*2**

#### S-type

- Store rs2 to Memory(rs1 + Immediate)

#### B-type

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

#### U-type

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

#### J-type

- JAL to Memory (PC + Immediate*2), i.e., rd = PC + 4; PC = PC + immediate*2
J-Format for Jump Instructions (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 ±2^{19} locations, 2 bytes apart
  - ±2^{18} 32-bit instructions, ±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)

- **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

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

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

- \( m \text{ bits} \times n \text{ bits} = m + n \text{ bit product} \)
Integer Multiplication (1/3)

- Paper and pencil example (unsigned):
  
  **Multiplicand**  1000  8
  **Multiplier**  x 1001  9
  1000

  72

- \[ m \text{ bits} \times n \text{ bits} = m + n \text{ bit product} \]
Integer Multiplication (1/3)

• Paper and pencil example (unsigned):
  \[
  \begin{array}{c}
  \text{Multiplicand} \quad 1000 \quad 8 \\
  \text{Multiplier} \quad x \quad 1001 \quad 9 \\
  \hline
  1000 \\
  0000
  \end{array}
  \]

  \[
  72
  \]

• \(m\) bits \(\times\) \(n\) 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 $\times n$ bits $= m + n$ bit product
Integer Multiplication (1/3)

- Paper and pencil example (unsigned):
  Multiplicand 1000 8
  Multiplier x 1001 9
  \[
  \begin{array}{c}
  1000 \\
  0000 \\
  0000 \\
  +1000 \\
  \hline
  72
  \end{array}
  \]

- \( m \) bits \( \times n \) 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 \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
• 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: \[ a = b \times c; \] # a should be declared long long
  – in RISC-V:
    • Let \( b \) be \( t_2 \); let \( c \) be \( t_3 \); and let \( a \) be \( t_0 \) and \( t_1 \) (since it may be up to 64 bits)
      \[
      \begin{align*}
      & \# \text{ upper half of} \\
      & \# \text{ product into } s_0 \\
      & \# \text{ lower half of} \\
      & \# \text{ product into } s_1
      \end{align*}
      \]
Integer Multiplication (3/3)

• Example:
  – in C: \( a = b \times c; \) # 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)
    \[
    \text{mulh } t0, t2, t3
    \] # upper half of
    \[
    \text{# product into } s0
    \] # lower half of
    \[
    \text{# product into } s1
    \]
Example:

- in C: \( a = b \times c \); \# 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
  - `mul t1,t2,t3` \# lower half of
Integer Division (1/2)

• Paper and pencil example (unsigned):

\[
\begin{array}{c|c}
\text{Divisor} & 1000 \\
\hline
\text{Quotient} & 1001010 \\
\text{Dividend} & 1001010
\end{array}
\]

• Dividend = Quotient x Divisor + Remainder
Integer Division (1/2)

• Paper and pencil example (unsigned):

\[
\begin{array}{c|c}
\text{Divisor} & 1001010 \\
\hline
\text{Quotient} & 1 \\
\text{Dividend} & 1000 \\
\hline
\text{Remainder} & 1
\end{array}
\]

• Dividend = Quotient \times \text{Divisor} + \text{Remainder}
Integer Division (1/2)

• Paper and pencil example (unsigned):

\[
\begin{array}{c|c}
\text{Divisor} & 1000 \mid 1001010 \\
\hline
\text{Dividend} & -1000 \\
\hline
\text{Quotient} & 10 \\
\end{array}
\]

• Dividend = Quotient \times \text{Divisor} + \text{Remainder}
Integer Division (1/2)

- Paper and pencil example (unsigned):

  \[
  \begin{array}{c|c}
  \text{Divisor} & 1001010 \\
  \hline
  1000 & 1001010 \\
  -1000 & \\
  \hline
  10 & \\
  101 & \\
  \end{array}
  \]

  Quotient

- Dividend = Quotient \times Divisor + Remainder
Integer Division (1/2)

• Paper and pencil example (unsigned):

\[
\begin{array}{c|c}
\text{Divisor} & 1001 \\
\hline
\text{Dividend} & 1001010 \\
\hline
-1000 & \\
\hline
10 & \\
101 & \\
1010 & \\
\end{array}
\]

• Dividend = Quotient x Divisor + Remainder
Integer Division (1/2)

• Paper and pencil example (unsigned):

\[
\begin{array}{c|c}
\text{Divisor} & 1001010 \\
\hline
\text{Dividend} & 1001010 \\
\hline
-1000 & \\
10 & \\
101 & \\
1010 & \\
-1000 & \\
\end{array}
\]

Quotient

• Dividend = Quotient \times \text{Divisor} + \text{Remainder}
Integer Division (1/2)

• Paper and pencil example (unsigned):

\[
\begin{array}{c|c}
\text{Quotient} & \underline{1001} \\
\text{Dividend} & \underline{1001010} \\
\text{Divisor} & 1000 \\
\end{array}
\]

\[
\begin{array}{c}
-1000 \\
10 \\
101 \\
1010 \\
-1000 \\
\end{array}
\]

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: \[ a = c / d; \quad b = c \% d; \]

• In RISC-V: \[ a\leftrightarrow t0; \quad b\leftrightarrow t1; \quad c\leftrightarrow t2; \quad d\leftrightarrow t3 \]
  \[ \text{div t0,t2,t3} \quad \# a = c/d \]
  \[ \text{rem t1,t2,t3} \quad \# b = c \% d \]
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
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`
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)

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

Machine Language Program (RISC-V)

Compiler

Assembler

Machine Interpretation

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

Architecture Implementation

Logic Circuit Description (Circuit Schematic Diagrams)

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

lw $t0, 0($2)
lw $t1, 4($2)    Anything can be represented
sw $t1, 0($2)    as a number, i.e., data or instructions
sw $t0, 4($2)

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

+ How to take a program and run it
Language Execution Continuum

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

  - Easy to program
  - Inefficient to interpret

  - Difficult to program
  - Efficient to interpret

- 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

C program: `foo.c`

Compiler

Assembly program: `foo.s`

Assembler

Object (mach lang module): `foo.o`

Linker

Executable (mach lang pgm): `a.out`

Loader

Memory

Commands:

```
gcc -O2 -S -c foo.c
```
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):
  \[ \text{mv t1,t2} \Rightarrow \text{addi t1,t2,0} \]
Outline

- Review Instruction Formats + J/JAL
- Multiply and Divide
- Interpretation vs. Translation
- Assembler
- Linker
- Loader
- And in Conclusion ...
Where Are We Now?

1. C program: foo.c
2. Compiler
3. Assembly program: foo.s
4. Assembler
5. Object (mach lang module): foo.o
6. Linker
7. Executable (mach lang pgm): a.out
8. Loader
9. Memory

CS164
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 \texttt{sym} global and can be referenced from other files
  
  **.string str:** Store the string \texttt{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: \texttt{mv t0, t1}  
Real:  
\texttt{mv t0, t1}
Pseudo-instruction Replacement

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

Pseudo:

\[
\begin{align*}
\text{mv} & \ t0, \ t1 \\
\text{neg} & \ t0, \ t1 \\
\text{li} & \ t0, \ \text{imm} \\
\text{not} & \ t0, \ t1 \\
\text{beqz} & \ t0, \ \text{loop} \\
\text{la} & \ t0, \ \text{str}
\end{align*}
\]

Real:

\[
\begin{align*}
\text{addi} & \ t0, t1, 0 \\
\text{sub} & \ t0, \text{zero}, \ t1 \\
\text{addi} & \ t0, \text{zero}, \ \text{imm} \\
\text{xori} & \ t0, \ t1, -1 \\
\text{beq} & \ t0, \text{zero}, \ \text{loop} \\
\text{lui} & \ t0, \text{str[31:12]} \\
\text{addi} & \ t0, t0, \text{str[11:0]} \ \text{OR} \\
\text{auipc} & \ t0, \text{str[31:12]} \\
\text{addi} & \ t0, t0, \text{str[11:0]}
\end{align*}
\]

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 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., \texttt{beq/bne} and \texttt{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:

```
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: \texttt{jal}, \texttt{jalr}
    • Internal
    • External (including lib files)
    • Such as the \texttt{la} instruction
      E.g., for \texttt{jalr} base register
  – Any piece of data in static section
    • Such as the \texttt{la} instruction
      E.g., for \texttt{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?

C program: foo.c
Compiler
Assembly program: foo.s
Assembler
Object (mach lang module): foo.o
Linker
Executable (mach lang pgm): a.out
Loader
Memory
lib.o
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)

<Linker>
  .o file 1
  text 1
  data 1
  info 1

  .o file 2
  text 2
  data 2
  info 2

  a.out
  Relocated text 1
  Relocated text 2
  Relocated data 1
  Relocated data 2
</Linker>
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 (\texttt{beq}, \texttt{bne}, \texttt{jal}; \texttt{auipc}/\texttt{addi})
  – Never need to relocate (PIC: position independent code)

• Absolute Function Address (\texttt{auipc}/\texttt{jalr})
  – Always relocate

• External Function Reference (\texttt{auipc}/\texttt{jalr})
  – Always relocate

• Static Data Reference (often \texttt{lui}/\texttt{addi})
  – Always relocate
Absolute Addresses in RISC-V

- Which instructions need relocation editing?
  - J-format: jump/jump and link
    
    | xxxxxx | 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 \textbf{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)
Outline

• Review Instruction Formats + J/JAL
• Multiply and Divide
• Interpretation vs. Translation
• Assembler
• Linker
• Loader
• And in Conclusion ...
Where Are We Now?

C program: `foo.c` → Compiler → Assembly program: `foo.s` → Assembler → Object (mach lang module): `foo.o` → Linker → Executable (mach lang pgm): `a.out` → Loader → Memory

Compiler
Assembly program: `foo.s`
Assembler
Object (mach lang module): `foo.o`
Linker
Executable (mach lang pgm): `a.out`
Loader
Memory
Loader Basics

• Input: Executable Code (e.g., \texttt{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
.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:  000000097 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)!

\[
0x00020 000 + 0xFFFFFFFF A10 = 0x0001F A10 \text{ (Off by 0x00001\,000)}
\]

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

\[
(0x00020 000 + 0x00001 000) + 0xFFFFFFFF A10 = 0x00020 A10
\]

What is **0xFFFFFFFF A10**?

Twos complement of 0xFFFFFFFF A10 = 0x00000 5EF + 1 = 0x00000 5F0 = 1520\text{ten}

So 0xFFFFFFFF A10 = -1520\text{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
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