Review

• *Instruction formats* designed to be similar but still capable of handling all instructions

• Branches and Jumps move relative to current address

• Assembly/Disassembly: Use RISC-V Green Sheet to convert
Great Idea #1: Levels of Representation/Interpretation

- **Higher-Level Language Program (e.g. C)**
  - **Compiler**
- **Assembly Language Program (e.g. RISC-V)**
  - **Assembler**
- **Machine Language Program (RISC-V)**
  - **Machine Interpretation**
  - **Hardware Architecture Description (e.g. block diagrams)**
  - **Architecture Implementation**
  - **Logic Circuit Description (Circuit Schematic Diagrams)**

```
we are here
```
```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```
```
lw t0, 0(x2)
lw t1, 4(x2)
sw t1, 0(x2)
sw t0, 4(x2)
```
```
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
```
COMPILER

ASSEMBLER

LINKER

LOADER
• Review/Intro

• **Translation**

• Compiler

• Assembler

• Linker

• Loader

• Example
Translation vs. Interpretation (1/3)

• 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

• Directly interpret a high level language when efficiency is not critical

• Translate to a lower level language when increased performance is desired
Translation vs. Interpretation (2/3)

• Generally easier to write an interpreter
• Interpreter closer to high-level, so can give better error messages (e.g. Python, Venus)
• Interpreter is slower (~10x), but code is smaller (~2x)
• Interpreter provides instruction set independence: can run on any machine
Translation vs. Interpretation (3/3)

• 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
C Translation

• **Recall:** A key feature of C is that it allows you to compile files *separately*, later combining them into a single executable

• What can be accessed across files?
  — Functions
  — Global variables
Steps to Starting a Program:

1) **Compiler**
2) **Assembler**
3) **Linker**
4) **Loader**
• Review/Intro
• Translation

• **Compiler**
• Assembler
• Linker
• Loader
• Example

C program: `foo.c`

Assembly program: `foo.s`

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

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

Library: `lib.o`

Loader

Memory
Compiler

• **Input:** Higher-level language (HLL) code (e.g. C, Java in files such as foo.c)

• **Output:** Assembly Language Code (e.g. foo.s for RISC-V)

• Note that the output may contain pseudo-instructions

• In reality, there’s a preprocessor step before this to handle #directives but it’s not very exciting
Compilers Are Non-Trivial

• There’s a whole course about them – CS164
  — We won’t go into much detail in this course
  — For the very curious and highly motivated:

• Some examples of the task’s complexity:
  — Operator precedence: 2 + 3 * 4
  — Operator associativity: a = b = c;
  — Determining locally whether a program is valid
    • if (a) { if (b) { ... /*long distance*/ ... } } } //extra bracket
• Review/Intro
• Translation
• Compiler
  • **Assembler**
• Linker
• Loader
• Example

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
Assembler

• **Input:** Assembly language code (e.g. `foo.s` for RISC-V)

• **Output:** Object code (True Assembly), information tables (e.g. `foo.o` for RISC-V)
  — Object file

• Reads and uses **directives**

• Replaces pseudo-instructions

• Produces machine language
Assembler Directives

(For more info, see p.B-5 and B-7 in P&H)

• 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
  .asciiz str: Store the string str in memory and null-terminates it
  .word w₁…wₙ: Store the n 32-bit quantities in successive memory words
Pseudo-instruction Replacement

<table>
<thead>
<tr>
<th>Pseudo</th>
<th>Real</th>
</tr>
</thead>
<tbody>
<tr>
<td>mv t0, t1</td>
<td>addi t0,t1,0</td>
</tr>
<tr>
<td>neg t0, t1</td>
<td>sub t0, zero, t1</td>
</tr>
<tr>
<td>li t0, imm</td>
<td>addi t0, zero, imm</td>
</tr>
<tr>
<td>not t0, t1</td>
<td>xori t0, t1, -1</td>
</tr>
<tr>
<td>beqz t0, loop</td>
<td>beq t0, zero, loop</td>
</tr>
<tr>
<td>la t0, str</td>
<td>lui t0, str[31:12]</td>
</tr>
<tr>
<td></td>
<td>addi t0, t0, str[11:0]</td>
</tr>
<tr>
<td></td>
<td>OR</td>
</tr>
<tr>
<td></td>
<td>auipc t0, str[31:12]</td>
</tr>
<tr>
<td></td>
<td>addi t0, t0, str[11:0]</td>
</tr>
</tbody>
</table>
Producing Machine Language (1/3)

• Simple Cases
  — Arithmetic and logical instructions, shifts, etc.
  — All necessary info contained in the instruction

• What about Branches and Jumps?
  — Branches and Jumps require a relative address
  — Once pseudo-instructions are replaced by real ones, we know by how many instructions to branch, so no problem
• “Forward Reference” problem
  —Branch instructions can refer to labels that are “forward” in the program:

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

—Solution: Make two passes over the program
Two Passes Overview

• Pass 1:
  — Expands pseudo instructions encountered
  — Remember position of labels
  — Take out comments, empty lines, etc
  — Error checking

• Pass 2:
  — Use label positions to generate relative addresses (for branches and jumps)
  — Outputs the object file, a collection of instructions in binary code
Producing Machine Language (3/3)

• What about jumps to external labels?
  — Requiring knowing a final address
  — Forward or not, can’t generate machine instruction without knowing the position of instructions in memory

• What about references to data?
  — `la` 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” that may be used by other files
  - Each file has its own symbol table
- What are they?
  - Labels: function calling
  - Data: anything in the `.data` section; variables may be accessed across files
- Keeping track of the labels fixes the forward reference problem
Relocation Table

• List of “items” this file will need the address of later (currently undetermined)

• What are they?
  — Any external label jumped to: jal or jalr
    • internal
    • external (including library files)
  — Any piece of data
    • such as anything referenced in the data section
Object File Format

1) object file header: size and position of the other pieces of the object file
2) text segment: the machine code
3) data segment: data in the source file (binary)
4) relocation table: identifies lines of code that need to be "handled"
5) symbol table: list of this file’s labels and data that can be referenced
6) debugging information

- A standard format is ELF (except MS)
  
Assembler

• **Input:** Assembly language code (e.g. `foo.s` for RISC-V)

• **Output:** Object code (True Assembly), information tables (e.g. `foo.o` for RISC-V)
  —Object file

• Reads and uses **directives**

• Replaces pseudo-instructions

• Produces machine language
• Review/Intro
• Translation
• Compiler
• Assembler
• **Linker**
• Loader
• Example
Linker (1/3)

• **Input:** Object Code files, information tables (e.g. foo.o, lib.o for RISC-V)

• **Output:** Executable Code (e.g. a.out for RISC-V)

• Combines several object (.o) files into a single executable (“linking”)

• Enables separate compilation of files
  — Changes to one file do not require recompilation of whole program
  — Old name “Link Editor” from editing the “links” in jump and link instructions
Linker (2/3)

Object file 1

- text 1
- data 1
- info 1

Object 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)

1) Take text segment from each .o file and put them together

2) Take data segment from each .o file, put them together, and concatenate this onto end of text segments

3) Resolve References
   — Go through Relocation Table; handle each entry
   — i.e. 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
Absolute Addresses in RISC-V

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

<table>
<thead>
<tr>
<th>xxxxx</th>
<th>jal</th>
</tr>
</thead>
</table>

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

<table>
<thead>
<tr>
<th>xxx</th>
<th>gp</th>
<th>rd</th>
<th>lw</th>
</tr>
</thead>
<tbody>
<tr>
<td>xx</td>
<td>rs1</td>
<td>gp</td>
<td>sw</td>
</tr>
</tbody>
</table>

• What about conditional branches?

| rs1 | rs2 | x | beq | bne |

• PC-relative addressing preserved even if code moves
Resolving References (1/2)

- Linker assumes the first word of the first text segment is at 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:
  1) Search for reference (data or label) in all “user” symbol tables
  2) If not found, search library files (e.g. `printf`)
  3) Once absolute address is determined, fill in the machine code appropriately

• Output of linker: executable file containing text and data (plus header)
• Review/Intro
• Translation
• Compiler
• Assembler
• Linker
• **Loader**
• Example
Loader

• **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

1) Reads executable file’s header to determine size of text and data segments

2) Creates new address space for program large enough to hold text and data segments, along with a stack segment

<more on this later>

3) Copies instructions and data from executable file into the new address space
Loader

4) Copies arguments passed to the program onto the stack

5) Initializes machine registers
   — Most registers cleared, but stack pointer assigned address of 1st free stack location

6) Jumps to start-up routine that copies program’s arguments from stack to registers and sets the PC
   — If main routine returns, start-up routine terminates program with the exit system call
• Review/Intro
• Translation
• Compiler
• Assembler
• Linker
• Loader
• Example
#include <stdio.h>

int main()
{
    printf("Hello, %s\n", "world");
    return 0;
}
.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: 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: 000021537 lui a0,0x21
  101bc: a1050513 addi a0,a0,-1520 # 20a10
<string1>
  101c0: 0000215b7 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: 000008067 jalr ra
```
Summary

• **Compiler** converts a single HLL file into a single assembly file \( .c \rightarrow .s \)

• **Assembler** removes pseudo-instructions, converts what it can to machine language, and creates a checklist for linker (relocation table) \( .s \rightarrow .o \)
  — Resolves addresses by making 2 passes (for internal forward references)

• **Linker** combines several object files and resolves absolute addresses \( .o \rightarrow .out \)
  — Enable separate compilation and use of libraries

• **Loader** loads executable into memory and begins execution