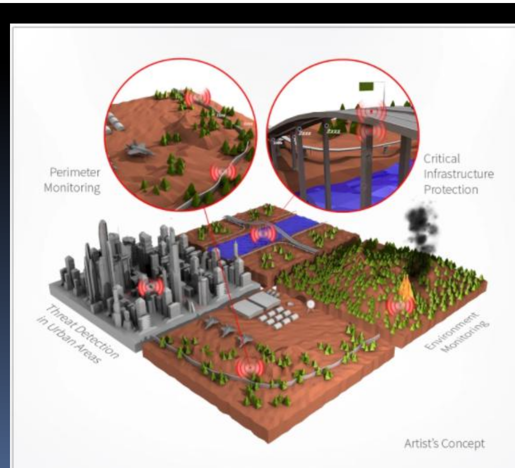
Computer Architecture
（计算机体系结构）

Lecturer Yuanqing

Lecture 13 – Running a Program I
(Compiling, Assembling, Linking, Loading)

2020.09.28

DARPA: Research Advances for Near-Zero-Power Sensors



(Source: DARPA)

Review

- Disassembly is simple and starts by decoding `opcode` field.
  - Be creative, efficient when authoring C
- Assembler expands real instruction set (TAL) with pseudoinstructions (MAL)
  - Only TAL can be converted to raw binary
  - Assembler's job to do conversion
  - Assembler uses reserved register `$at`
  - MAL makes it <u>much</u> easier to write MIPS

Overview

- Interpretation vs Translation
- Translating C Programs
  - Compiler
  - Assembler
  - Linker (next time)
  - Loader (next time)
- An Example (next time)

Language Execution Continuum

- An Interpreter is a program that executes other programs.

Java bytecode

Scheme Java C++ C Assembly machine language

Easy to program — Difficult to program

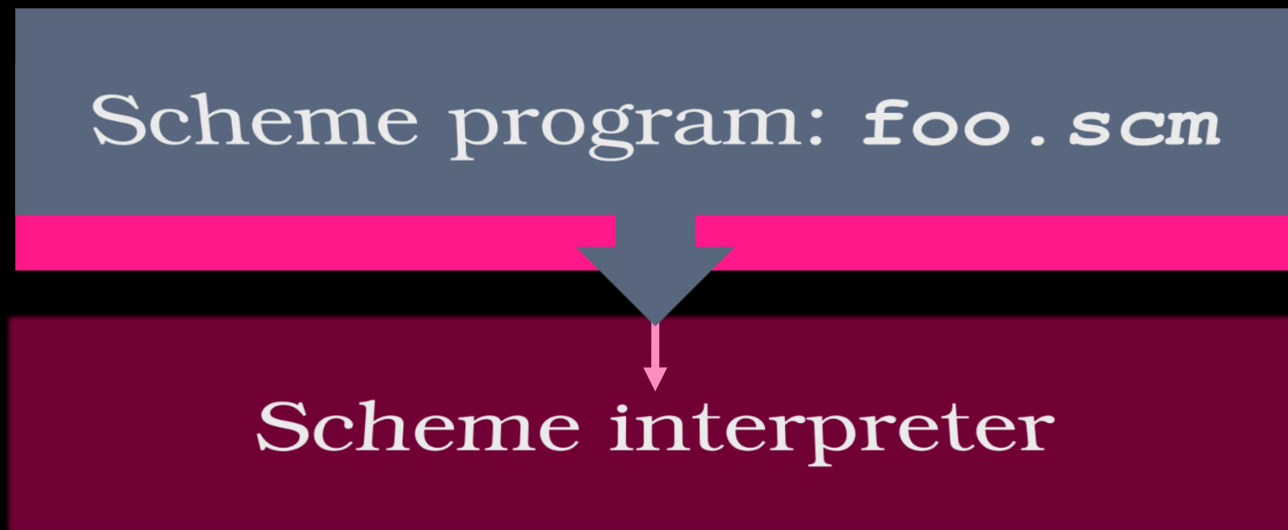
Inefficient to interpret — Efficient to interpret

- 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 for 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
- For example, consider a Scheme program **`foo.scm`**

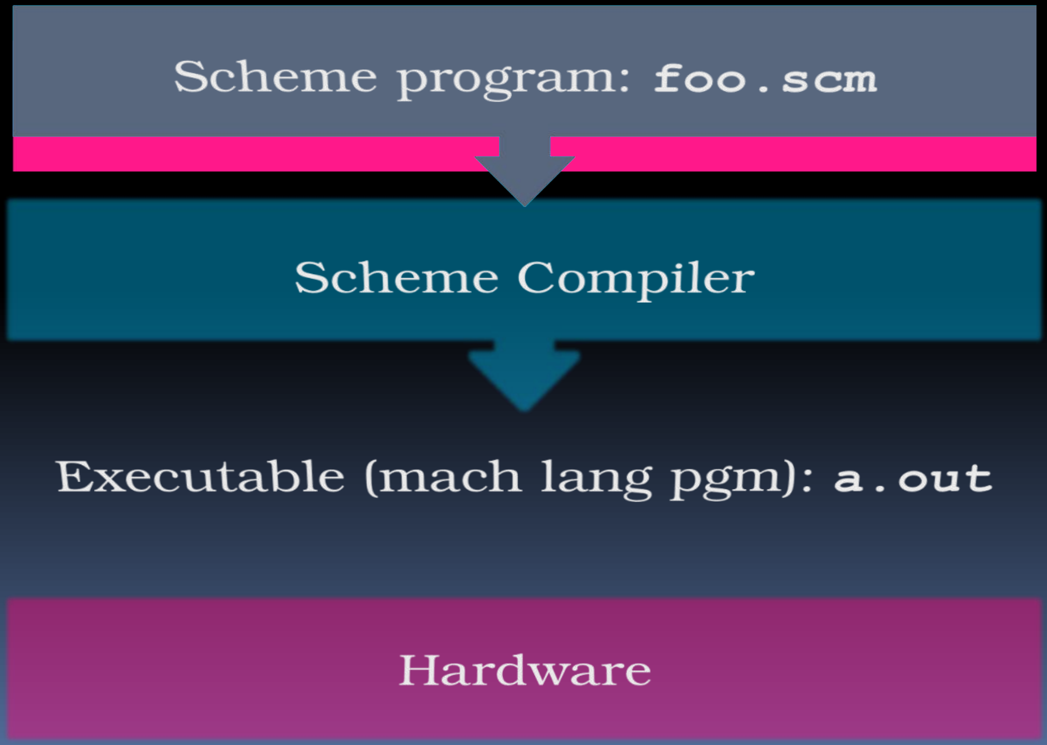
Interpretation



- Scheme Interpreter is just a program that reads a scheme program and performs the functions of that scheme program.

Translation

- Scheme Compiler is a translator from Scheme to machine language.
- The processor is a hardware interpeter of machine language.



Interpretation

- Any good reason to interpret machine language in software?
- SPIM – useful for learning / debugging
- Apple Macintosh conversion
  - Switched from Motorola 680x0 instruction architecture to PowerPC.
    - Similar issue with switch to 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., MARS, stk)
  - 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 (eg. 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 to S
(translation)

Compiler

Assembly program: **`foo.s`**

Assembler

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

Linker ← **`lib.o`**

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

Loader

Memory

Compiler

- Input: High-Level Language Code (e.g., C, Java such as `foo.c`)
- Output: Assembly Language Code (e.g., `foo.s` for MIPS)
- Note: Output *may* contain pseudoinstructions
- Pseudoinstructions: instructions that assembler understands but not in machine (last lecture)
  For example:
  - `mov $s1,$s2` ⇒ `or $s1,$s2,$zero`

Administrivia…

- Deadline of assignment: 10.9
- Deadline of lab 1 & lab 2: 10.9

Where A

Compiler

Assembly program: `foo.s`

CS164

Assembler

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

Linker ← `lib.o`

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

Loader

Memory

Assembler

- Input: Assembly Language Code (e.g., `foo.s` for MIPS)
- Output: Object Code, information tables (e.g., `foo.o` for MIPS)
- Reads and Uses Directives
- Replace Pseudoinstructions
- Produce Machine Language
- Creates Object File

Assembler Directives (p. A-51 to A-53)

- 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-terminate it
  - **.word w1…wn:** Store the *n* 32-bit quantities in successive memory words

Pseudoinstruction Replacement

- Asm. treats convenient variations of machine language instructions as if real instructions

Pseudo:	Real:
`subu $sp,$sp,32`	`addiu $sp,$sp,-32`
`sd $a0, 32($sp)`	`sw $a0, 32($sp)` `sw $a1, 36($sp)`
`mul $t7,$t6,$t5`	`mul $t6,$t5` `mflo $t7`
`addu $t0,$t6,1`	`addiu $t0,$t6,1`
`ble $t0,100,loop`	`slti $at,$t0,101` `bne $at,$0,loop`
`la $a0, str`	`lui $at,left(str)` `ori $a0,$at,right(str)`

Producing Machine Language (1/3)

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

Producing Machine Language (2/3)

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

```
      or   $v0, $0,  $0
L1:   slt  $t0, $0,  $a1
      beq  $t0, $0,  L2
      addi $a1, $a1, -1
      j    L1
L2:   add  $t1, $a0, $a1
```

  - Solved by taking 2 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 jumps (`j` and `jal`)?
  - Jumps require absolute address.
  - So, forward or not, still 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 `ori`
  - These will require the full 32-bit address of the data.
- These can't be determined yet, so we create two tables…

Symbol Table

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

Relocation Table

- List of "items" this file needs the address later.
- What are they?
  - Any label jumped to: **`j`** or **`jal`**
    - internal
    - external (including lib files)
  - Any piece of data
    - such as the **`la`** instruction

Object File Format

- object file header: size and position of the other pieces of the object file
- text segment: the machine code
- data segment: binary representation of the data in the source file
- relocation information: identifies lines of code that need to be "handled"
- symbol table: list of this file's labels and data that can be referenced
- debugging information
- A standard format is ELF (except MS)

  http://www.skyfree.org/linux/references/ELF_Format.pdf

Peer Instruction

1) Assembler will ignore the instruction `Loop:nop` because it does nothing.

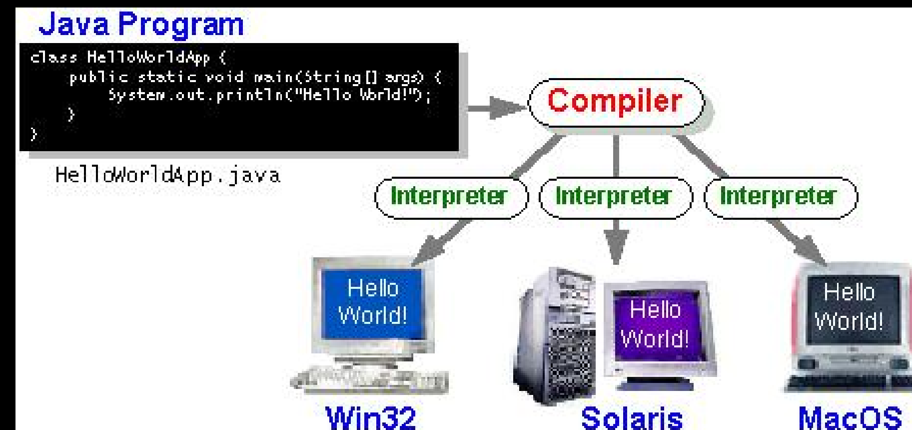
2) Java designers used a translater AND interpreter (rather than just a translater) mainly because of (at least 1 of): ease of writing, better error msgs, smaller object code.

	12
a)	FF
b)	FT
c)	TF
d)	TT

Peer Instruction Answer

1) Assembler keeps track of all labels in symbol table…F!

2) Java designers used both mainly because of code portability…F!

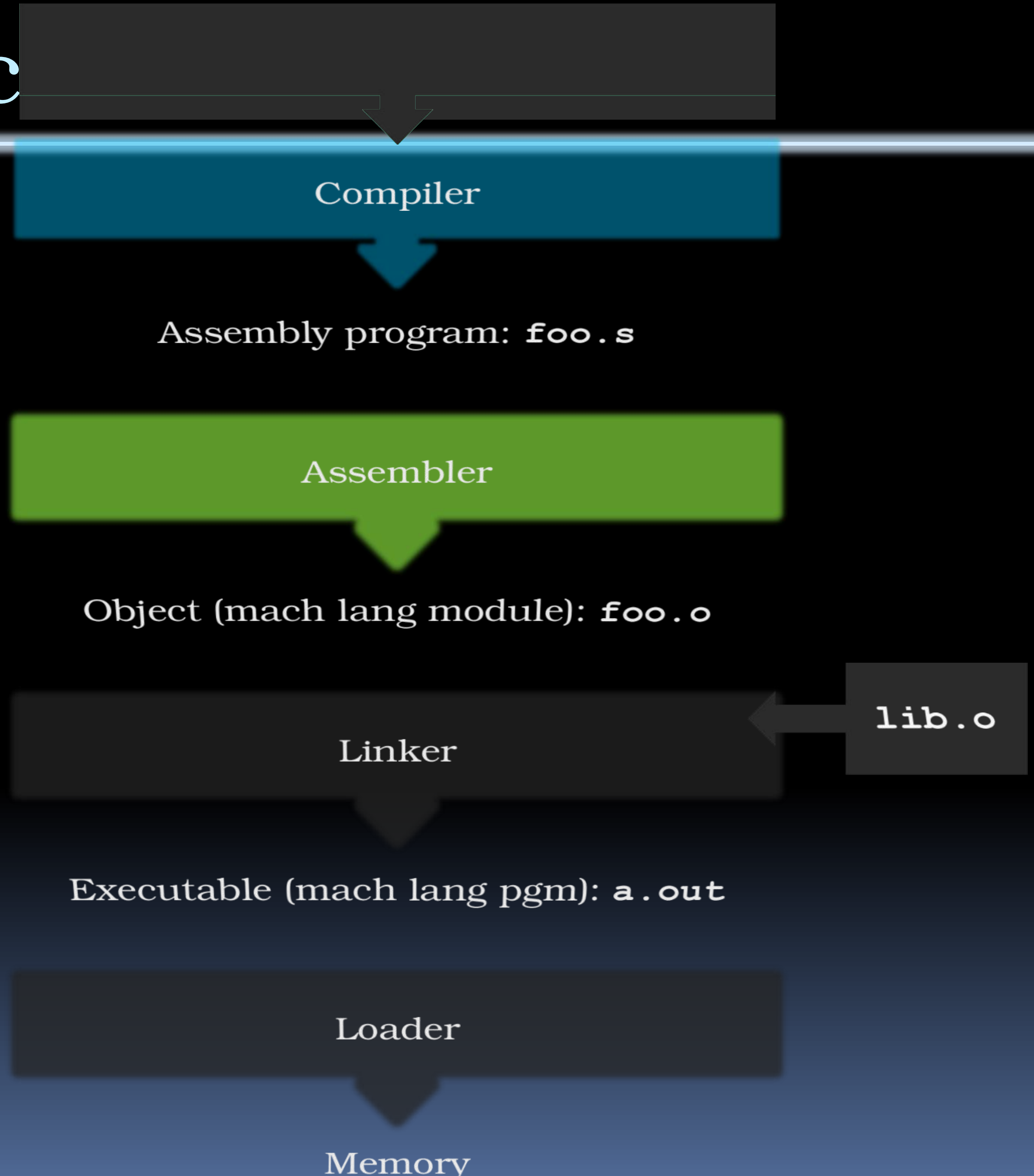


1) Assembler will ignore the instruction `Loop:nop` because it does nothing.

2) Java designers used a translater AND interpreter (rather than just a translater) mainly because of (at least 1 of): ease of writing, better error msgs, smaller object code.

	12
a)	FF
b)	FT
c)	TF
d)	TT

And in c


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

Bonus slides

- These are extra slides that used to be included in lecture notes, but have been moved to this, the "bonus" area to serve as a supplement.
- The slides will appear in the order they would have in the normal presentation

Bonus

Integer Multiplication (1/3)

- Paper and pencil example (unsigned):

```
  Multiplicand    1000        8
   Multiplier    x1001        9
                  1000
                 0000
                0000
             +1000
             01001000
```

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

Integer Multiplication (2/3)

- In MIPS, we multiply registers, so:
  - 32-bit value x 32-bit value = 64-bit value
- Syntax of Multiplication (signed):
  - `mult` register1, register2
  - Multiplies 32-bit values in those registers & puts 64-bit product in special result regs:
    - puts product upper half in hi, lower half in lo
  - hi and lo are 2 registers separate from the 32 general purpose registers
  - Use `mfhi` register & `mflo` register to move from hi, lo to another register

Integer Multiplication (3/3)

- Example:
  - in C: `a = b * c;`
  - in MIPS:
    - let `b` be `$s2`; let `c` be `$s3`; and let `a` be `$s0` and `$s1` (since it may be up to 64 bits)

```
mult $s2,$s3    # b*c
mfhi $s0        # upper half of
                # product into $s0
mflo $s1        # lower half of
                # product into $s1
```

- Note: Often, we only care about the lower half of the product.

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` register1, register2
  - Divides 32-bit register 1 by 32-bit register 2:
  - puts remainder of division in `hi`, quotient in `lo`
- Implements C division (/) and modulo (%)
- Example in C: `a = c / d;` `b = c % d;`
- in MIPS: `a↔$s0;b↔$s1;c↔$s2;d↔$s3`

```
div  $s2,$s3    # lo=c/d, hi=c%d
mflo $s0        # get quotient
```