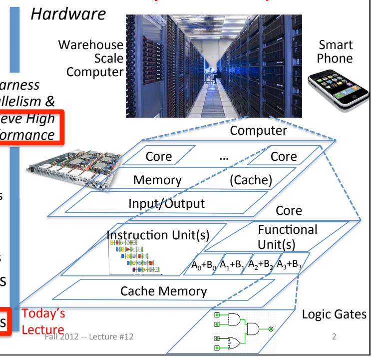
CS 61C:
Great Ideas in Computer Architecture
Assemblers, Linkers, Compilers

Instructors:
Krste Asanovic, Randy H. Katz
http://inst.eecs.Berkeley.edu/~cs61c/fa12

New-School Machine Structures (It's a bit more complicated!)

Software

- Parallel Requests
  Assigned to computer
  e.g., Search "Katz"
- Parallel Threads
  Assigned to core
  e.g., Lookup, Ads
- Parallel Instructions
  >1 instruction @ one time
  e.g., 5 pipelined instructions
- Parallel Data
  >1 data item @ one time
  e.g., Add of 4 pairs of words
- Hardware descriptions
  All gates @ one time
- Programming Languages



Big Idea #1: Levels of Representation/ Interpretation

Today's Lecture

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

Compiler

Assembly Language Program (e.g., MIPS)

Assembler

Machine Language Program (MIPS)

Machine Interpretation

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

Architecture Implementation

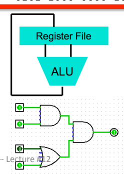
Logic Circuit Description (Circuit Schematic Diagrams)

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

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



Agenda

- Review: Performance
- Assemblers
- Administrivia
- Linkers
- Compilers vs. Interpreters
- And in Conclusion, …

Review: Defining Relative CPU Performance

- $\text{Performance}_X = 1/\text{Program Execution Time}_X$
- $\text{Performance}_X > \text{Performance}_Y \Rightarrow$
  $1/\text{Execution Time}_X > 1/\text{Execution Time}_Y \Rightarrow$
  $\text{Execution Time}_Y > \text{Execution Time}_X$
- *Computer X is N times faster than Computer Y*
  $\text{Performance}_X / \text{Performance}_Y = N$ or
  $\text{Execution Time}_Y / \text{Execution Time}_X = N$

Review: Performance Equation

- Time = $\dfrac{\text{Seconds}}{\text{Program}}$

$$= \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock Cycle}}$$

Performance Comparison

MacAir 3.1: 1.6 Ghz
MacAir 5.1: 2.0 GHz


	Late 2010 11"	Mid-2011 11"
Processor Speed:	1.4 GHz*	*1.6 GHz***
Processor Type:	Core 2 Duo (SU9400)*	*Core i5 (I5-2467M)***
L2/L3 Cache:	3 MB	*256k x2, 3 MB*
System Bus/DMI:	800 MHz	*5 GT/s*
Standard RAM:	2 GB	2 GB, *4 GB*
Maximum RAM:	4 GB†	4 GB†
Onboard RAM Type:	1066 MHz DDR3	*1333 MHz DDR3*
Internal Storage:	64 GB, 128 GB	64 GB, 128 GB
Storage Type:	SSD (Mini SATA)	SSD (Mini SATA)
Video Processor:	GeForce 320M	HD Graphics 3000
Shared VRAM:	256 MB	*256 MB, 384 MB*
Thunderbolt:	No	*Yes*
SD Card Slot:	No	No
Bluetooth:	2.1+EDR	*4.0*
Display Size:	11.6" Widescreen	11.6" Widescreen
Display Resolution:	1366x768	1366x768
Backlit Keyboard:	No	*Yes*
Battery Life:	5 Hours	5 Hours
Dimensions:	.11-.68 x 11.8 x 7.56	.11-.68 x 11.8 x 7.56
Weight:	*2.3 Pounds*	2.38 Pounds
Order Numbers:	MC505LL/A, MC506LL/A	MC968LL/A, MC969LL/A
EMC Number:	2393	2471
Model Identifier:	MacBookAir3,1	MacBookAir4,1

Performance Comparison

MacAir 3.1: 1.6 Ghz
MacAir 5.1: 2.0 GHz


Hardware Overview:

Model Name:	MacBook Air
Model Identifier:	MacBookAir3,1
Processor Name:	Intel Core 2 Duo
Processor Speed:	1.6 GHz
Number of Processors:	1
Total Number of Cores:	2
L2 Cache:	3 MB
Memory:	4 GB
Bus Speed:	800 MHz
Boot ROM Version:	MBA31.0061.B01
SMC Version (system):	1.67f4
Serial Number (system):	C02DW245DDR0
Hardware UUID:	35273131-8D15-5DEB-A919-701640168CB1

Hardware Overview:

Model Name:	MacBook Air
Model Identifier:	MacBookAir5,1
Processor Name:	Intel Core i7
Processor Speed:	2 GHz
Number of Processors:	1
Total Number of Cores:	2
L2 Cache (per Core):	256 KB
L3 Cache:	4 MB
Memory:	8 GB
Boot ROM Version:	MBA51.00EF.B00
SMC Version (system):	2.4f17
Serial Number (system):	C02J4248DRV9
Hardware UUID:	7CEA9B1F-172E-59ED-ACCA-F892BD0E0A98

Review: Performance Equation

- Time = $\frac{\text{Seconds}}{\text{Program}}$

$$= \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock Cycle}}$$

$\text{Time}_{3.1}$ = 1/1.6Ghz = 625 ns

$\text{Time}_{5.1}$ = 1/2.0Ghz = 500 ns

$$\frac{\text{Performance}_{5.1}}{\text{Performance}_{3.1}} = \frac{\text{Time}_{3.1}}{\text{Time}_{5.1}} = \frac{625}{500} = 1.25$$

gcc Optimization Experiment

	BubbleSort.c	Dhrystone.c
MacAir 3.1 No Opt		
-O2		
MacAir 5.1 No Opt		
-O2		

gcc Optimization Experiment

	BubbleSort.c	Dhrystone.c
MacAir 3.1 No Opt	3.2 s	7.4 s 3125000 dhrys/s
-O2	1.5 s	2.7 s 8333333 dhrys/s
MacAir 5.1 No Opt	1.9 s (1.7x)	3.0 s (2.4x) 8333333 dhrys/s (2.7x)
-O2	0.8 s (1.9x)	0.7 s (3.8x) 25000000 dhrys/s (3x)

Agenda

- Review: Performance
- Assemblers
- Administrivia
- Linkers
- Compilers vs. Interpreters
- And in Conclusion, …

Converting C to MIPS Machine code

\$t0 (reg 8), &A in \$t1 (reg 9), h=\$s2 (reg 18)

A[300] = h + A[300];

Format?

Student Roulette?

	Format		
lw \$t0,1200(\$t1)	_		
addu \$t0,\$s2,\$t0	_		
sw \$t0,1200(\$t1)	_		

Instruction	Format	op	rs	rt	rd	shamt	funct	address
addu	R	0	reg	reg	reg	0	33_{ten}	n.a.
lw (load word)	I	35_{ten}	reg	reg	n.a.	n.a.	n.a.	address
sw (store word)	I	43_{ten}	reg	reg	n.a.	n.a.	n.a.	address

R-type	op	rs	rt	rd	shamt	funct
I-type	op	rs	rt	address or constant		
J-type	op	address				

Converting C to MIPS Machine code

\$t0 (reg 8), &A in \$t1 (reg 9), h=\$s2 (reg 18)

A[300] = h + A[300];

Format?

	Format						
lw \$t0,1200(\$t1)	_	35	9	8	1200		
addu \$t0,\$s2,\$t0	_	0	18	8	8	0	33
sw \$t0,1200(\$t1)	_	43	9	8	1200		

Instruction	Format	op	rs	rt	rd	shamt	funct	address
addu	R	0	reg	reg	reg	0	33_{ten}	n.a.
lw (load word)	I	35_{ten}	reg	reg	n.a.	n.a.	n.a.	address
sw (store word)	I	43_{ten}	reg	reg	n.a.	n.a.	n.a.	address

R-type	op	rs	rt	rd	shamt	funct
I-type	op	rs	rt	address or constant		
J-type	op	address				

Converting to MIPS Machine code

Address Loop: Format?

Student Roulette?

Address	Instruction	Format						
800	sll \$t1,\$s3,2	R						
804	addu \$t1,\$t1,\$s6	R						
808	lw \$t0,0(\$t1)	I						
812	bne \$t0,\$s5, Exit	I						
816	addiu \$s3,\$s3,1	I						
820	j Loop	J						

Exit:

R-type	op	rs	rt	rd	shamt	funct
I-type	op	rs	rt	address or constant		
J-type	op	address				

Converting to MIPS Machine code

Address Loop: Format?

Address	Instruction	Format						
800	sll \$t1,\$s3,2	R	0	0	19	9	2	0
804	addu \$t1,\$t1,\$s6	R	0	9	22	9	0	33
808	lw \$t0,0(\$t1)	I	35	9	8	0		
812	bne \$t0,\$s5, Exit	I	5	8	21	2		
816	addiu \$s3,\$s3,1	I	8	19	19	1		
820	j Loop	J	2	200				

Exit:

R-type	op	rs	rt	rd	shamt	funct
I-type	op	rs	rt	address or constant		
J-type	op	address				

32-bit Constants in MIPS

- Can create a 32-bit constant from two 32-bit MIPS instructions
- *Load Upper Immediate (*`lui` *or* "Louie") puts 16 bits into upper 16 bits of destination register
- MIPS to load 32-bit constant into register \$s0? $0000\ 0000\ 0011\ 1101\ 0000\ 1001\ 0000\ 0000_{two}$

lui \$s0, 61 # 61 = $0000\ 0000\ 0011\ 1101_{two}$

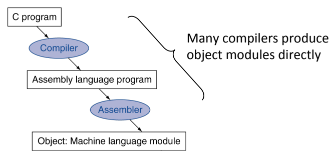
ori \$s0, \$s0, 2304 # 2304 = $0000\ 1001\ 0000\ 0000_{two}$

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 Pseudo-instructions
- Produce Machine Language
- Creates Object File

Translation

§2.12 Translating and Starting a Program



Assembly and Pseudo-instructions

- Turning textual MIPS instructions into machine code called *assembly,* program called *assembler*
  - Calculates addresses, maps register names to numbers, produces binary machine language
  - Textual language called *assembly language*
- Can also accept instructions convenient for programmer but not in hardware
  - *Load immediate* (*li*) allows 32-bit constants, assembler turns into lui + ori (if needed)
  - *Load double* (*ld*) uses two lwc1 instructions to load a pair of 32-bit floating point registers
  - Called *Pseudo-Instructions*

Assembler Directives (P&H B-5 to B-7)

- Give directions to assembler, but do not produce machine instructions
  - `.text`: Subsequent items put in user text segment
  - `.data`: Subsequent items put in user data segment
  - `.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` $w_1 \dots w_n$: Store the *n* 32-bit quantities in successive memory words

Assembler Pseudo-instructions

- Most assembler instructions represent machine instructions one-to-one
- Pseudo-instructions: figments of the assembler's imagination

```
move $t0, $t1      →  add $t0, $zero, $t1
blt $t0, $t1, L    →  slt $at, $t0, $t1
                      bne $at, $zero, L
```

  - `$at` (register 1): assembler temporary

More Pseudo-instructions

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

Pseudo:	Real:
`addu $t0,$t6,1`	______________
`subu $sp,$sp,32`	______________
`sd $a0,32($sp)`	______________
	______________
`la $a0,str`	______________

More Pseudoinstructions

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

Pseudo:	Real:
`addu $t0,$t6,1`	`addiu $t0,$t6,1`
`subu $sp,$sp,32`	______________
`sd $a0,32($sp)`	______________
	______________
`la $a0,str`	______________

More Pseudoinstructions

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

Pseudo: / Real:

```
addu $t0,$t6,1        addiu $t0,$t6,1
subu $sp,$sp,32       addiu $sp,$sp,-32
sd $a0,32($sp)        _______________
                      _______________
la $a0,str            _______________
```


More Pseudoinstructions

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

Pseudo: / Real:

```
addu $t0,$t6,1        addiu $t0,$t6,1
subu $sp,$sp,32       addiu $sp,$sp,-32
sd $a0,32($sp)        sw $a0,32($sp)
                      sw $a1,36($sp)
la $a0,str            lui $at,left(str)
                      ori $a0,$at,right(str)
```


Producing an Object Module

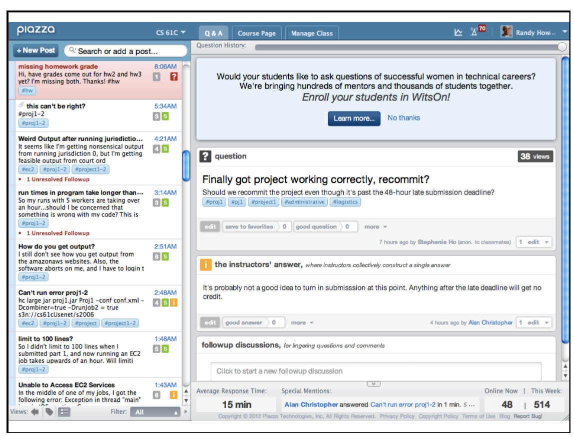
- Assembler (or compiler) translates program into machine instructions
- Provides information for building a complete program from the pieces
  - Header: described contents of object module
  - Text segment: translated instructions
  - Static data segment: data allocated for the life of the program
  - Relocation info: for contents that depend on absolute location of loaded program
  - Symbol table: global definitions and external refs
  - Debug info: for associating with source code

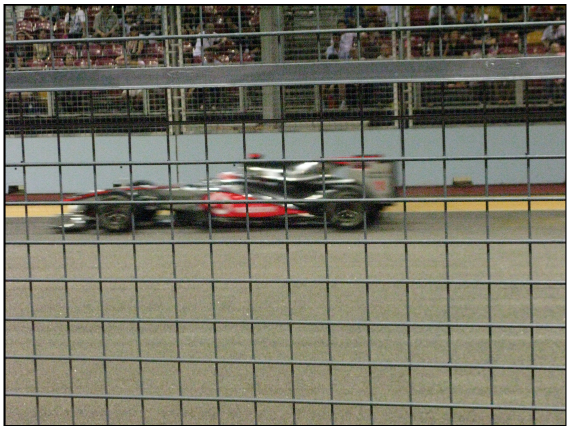
Agenda

- Review
- Assemblers
- Administrivia
- Linkers
- Compilers vs. Interpreters
- And in Conclusion, …

Administrivia

- Midterm! October 9th, Evening, 8 PM – 10 PM
  - 1 hour exam with 2 hours to complete it
  - Closed notes, book; one 8.5x11” crib sheet, MIPS green card provided
  - Comprehensive from course start
    - Lectures, Labs, Projects
  - There will be a TA-led review session
  - Special accommodations, contact instructors





Agenda

- Review
- Assemblers
- Linkers
- Administrivia
- Compilers vs. Interpreters
- And in conclusion, …

Separate Compilation and Assembly

- No need to compile all code at once
- How to put pieces together?

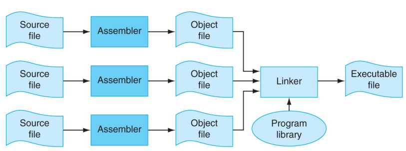
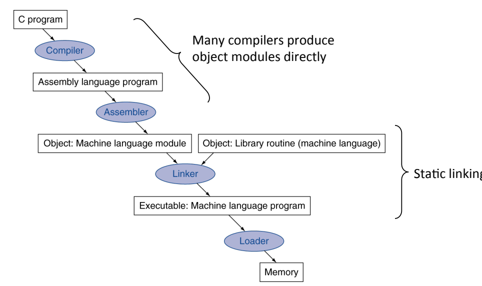


FIGURE B.1.1 The process that produces an executable file. An assembler translates a file of assembly language into an object file, which is linked with other files and libraries into an executable file.

Translation and Startup

§2.12 Translating and Starting a Program



Linker Stitches Files Together

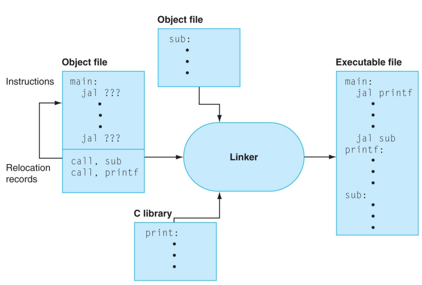


FIGURE B.3.1 The linker searches a collection of object fi les and program libraries to find nonlocal routines used in a program, combines them into a single executable file, and resolves references between routines in different files.

Linking Object Modules

- Produces an executable image
  1. Merges segments
  2. Resolve labels (determine their addresses)
  3. Patch location-dependent and external refs
- Often slower than compiling
  – All the machine code files must be read into memory and linked together

Loading a Program

- Load from image file on disk into memory
  1. Read header to determine segment sizes
  2. Create virtual address space (covered later in semester)
  3. Copy text and initialized data into memory
  4. Set up arguments on stack
  5. Initialize registers (including `$sp`, `$fp`, `$gp`)
  6. Jump to startup routine
     - Copies arguments to $a0, … and calls main
     - When main returns, do "exit" systems call

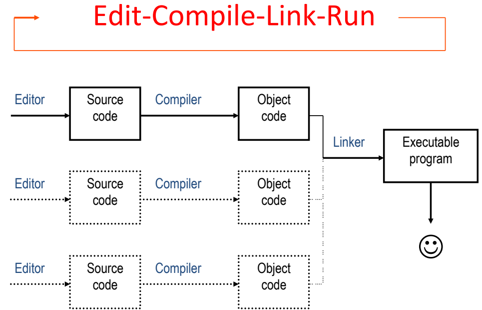
Agenda

- Review
- Assemblers
- Administrivia
- Linkers
- Compilers vs. Interpreters
- And in Conclusion, …

What's a Compiler?

- *Compiler*: a program that accepts as input a program text in a certain language and produces as output a program text in another language, *while preserving the meaning of that text*.
- Text must comply with the syntax rules of whichever programming language it is written in.
- Compiler's complexity depends on the syntax of the language and how much abstraction that programming language provides.
  – A C compiler is much simpler than C++ Compiler
- Compiler executes *before* compiled program runs

Compiled Languages: Edit-Compile-Link-Run



Compiler Optimization

- gcc compiler options

-O1: the compiler tries to reduce code size and execution time, without performing any optimizations that take a great deal of compilation time

-O2: Optimize even more. GCC performs nearly all supported optimizations that do not involve a space-speed tradeoff. As compared to -O, this option increases both compilation time and the performance of the generated code

-O3: Optimize yet more. All -O2 optimizations and also turns on the -finline-functions, …

What is Typical Benefit of Compiler Optimization?

- What is a typical program?
- For now, try a toy program: BubbleSort.c

```
#define ARRAY_SIZE 20000
int main() {
  int iarray[ARRAY_SIZE], x, y, holder;
for(x = 0; x < ARRAY_SIZE; x++)
    for(y = 0; y < ARRAY_SIZE-1; y++)
      if(iarray[y] > iarray[y+1]) {
        holder = iarray[y+1];
        iarray[y+1] = iarray[y];
        iarray[y] = holder;
      }
}
```

Unoptimized MIPS Code

```
$L3:
    lw     $2,80016($sp)
    slt    $3,$2,20000
    bne    $3,$0,$L6
    j      $L4
$L6:
    .set   noreorder
    nop
    .set   reorder
    sw     $0,80020($sp)
$L7:
    lw     $2,80020($sp)
    slt    $3,$2,19999
    bne    $3,$0,$L10
    j      $L5
$L10:
    lw     $2,80020($sp)
    move   $3,$2
    sll    $2,$3,2
    addu   $3,$sp,16
    addu   $2,$3,$2
    lw     $4,80020($sp)
    addu   $3,$4,1
    move   $4,$3
    sll    $3,$4,2
    addu   $4,$sp,16
    addu   $3,$4,$3
    lw     $2,0($2)
    lw     $3,0($3)
    slt    $2,$3,$2
    beq    $2,$0,$L9
    lw     $3,80020($sp)
    addu   $2,$3,1
    move   $3,$2
    sll    $2,$3,2
    addu   $3,$sp,16
    addu   $2,$3,$2
    lw     $3,0($2)
    sw     $3,80024($sp
    lw     $3,80020($sp)
    addu   $2,$3,1
    move   $3,$2
    sll    $2,$3,2
    addu   $3,$sp,16
    addu   $2,$3,$2
    lw     $3,80020($sp)
    move   $4,$3
    sll    $3,$4,2
    addu   $4,$sp,16
    addu   $3,$4,$3
    lw     $4,0($3)
    sw     $4,0($2)
    lw     $2,80020($sp)
    move   $3,$2
    sll    $2,$3,2
    addu   $3,$sp,16
    addu   $2,$3,$2
    lw     $3,80024($sp)
    sw     $3,0($2)
$L11:
$L9:
    lw     $2,80020($sp)
    addu   $3,$2,1
    sw     $3,80020
    ($sp)
    j      $L7
$L8:
$L5:
    lw     $2,80016($sp)
    addu   $3,$2,1
    sw     $3,80016
    ($sp)
    j      $L3
$L4:
$L2:
    li     $12,65536
    ori
    $12,$12,0x38b0
    addu   $13,$12,$sp
    addu   $sp,$sp,$12
    j      $31
```

9/21/12 Fall 2012 -- Lecture #12 43

-O2 optimized MIPS Code

```
    li     $13,65536
    ori    $13,$13,0x3890
    addu   $13,$13,$sp
    sw     $28,0($13)
    move   $4,$0
    addu   $8,$sp,16
$L6:
    move   $3,$0
    addu   $9,$4,1
    .p2align 3
$L10:
    sll    $2,$3,2
    addu   $6,$8,$2
    addu   $7,$3,1
    sll    $2,$7,2
    addu   $5,$8,$2
    lw     $3,0($6)
    lw     $4,0($5)
    slt    $2,$4,$3
    beq    $2,$0,$L9
    sw     $3,0($5)
    sw     $4,0($6)
$L9:
    move   $3,$7
    slt    $2,$3,19999
    bne    $2,$0,$L10
    move   $4,$9
    slt    $2,$4,20000
    bne    $2,$0,$L6
    li     $12,65536
    ori    $12,$12,0x38a0
    addu   $13,$12,$sp
    addu   $sp,$sp,$12
    j      $31
    .
```

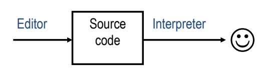
9/21/12 Fall 2012 -- Lecture #12 44

What's an Interpreter?

- Reads and executes source statements executed one at a time
  - No linking
  - No machine code generation, so more portable
- Starts executing quicker, but runs much more slowly than compiled code
- Performing the actions straight from the text allows better error checking and reporting to be done
- Interpreter stays around during execution
  - Unlike compiler, some work is done *after* program starts
- Writing an interpreter is much less work than writing a compiler

9/21/12 Fall 2012 -- Lecture #12 45

Interpreted Languages: Edit-Run



9/21/12 Fall 2012 -- Lecture #12 2-46

Compiler vs. Interpreter Advantages

Compilation:
- Faster Execution
- Single file to execute
- Compiler can do better diagnosis of syntax and semantic errors, since it has more info than an interpreter (Interpreter only sees one line at a time)
- Can find syntax errors *before* run program
- Compiler can optimize code

Interpreter:
- Easier to debug program
- Faster development time

9/21/12 Fall 2012 -- Lecture #12 47

Compiler vs. Interpreter Disadvantages

Compilation:
- Harder to debug program
- Takes longer to change source code, recompile, and relink

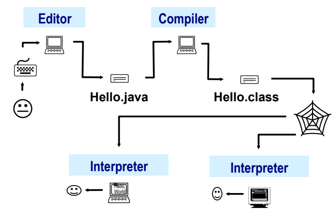
Interpreter:
- Slower execution times
- No optimization
- Need all of source code available
- Source code larger than executable for large systems
- Interpreter must remain installed while the program is interpreted

9/21/12 Fall 2012 -- Lecture #12 48

Java's Hybrid Approach: Compiler + Interpreter

- A Java compiler converts Java source code into instructions for the *Java Virtual Machine (JVM)*
- These instructions, called *bytecodes*, are same for any computer / OS
- A CPU-specific Java interpreter interprets bytecodes on a particular computer

Java's Compiler + Interpreter



Why Bytecodes?

- Platform-independent
- Load from the Internet faster than source code
- Interpreter is faster and smaller than it would be for Java source
- Source code is not revealed to end users
- Interpreter performs additional security checks, screens out malicious code

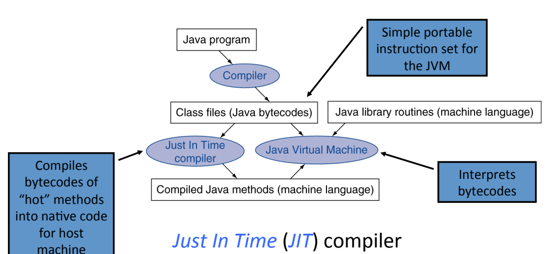
JVM uses Stack vs. Registers

```
a = b + c;
=>
iload b   ;  push b onto Top Of Stack (TOS)
iload c   ;  push c onto Top Of Stack (TOS)
iadd      ;  Next to top Of Stack (NOS) =
          ;  Top Of Stack (TOS) + NOS
istore a  ;  store TOS into a and pop stack
```


Java Bytecodes (Stack) vs. MIPS (Reg.)

Category	Operation	Java bytecode	Size (bits)	MIPS Instr.	Meaning
Arithmetic	add	iadd	8	add	NOS=TOS+NOS; pop
	subtract	isub	8	sub	NOS=TOS–NOS; pop
	increment	iinc I8a I8b	8	addi	Frame[I8a]= Frame[I8a] + I8b
Data transfer	load local integer/address	iload I8/aload I8	16	lw	TOS=Frame[I8]
	load local integer/address	iload_/aload_{0,1,2,3}	8	lw	TOS=Frame[{0,1,2,3}]
	store local integer/address	istore I8/astore I8	16	sw	Frame[I8]=TOS; pop
	load integer/address from array	iaload/aaload	8	lw	NOS=*NOS[TOS]; pop
	store integer/address into array	iastore/aastore	8	sw	*NNOS[NOS]=TOS; pop2
	load half from array	saload	8	lh	NOS=*NOS[TOS]; pop
	store half into array	sastore	8	sh	*NNOS[NOS]=TOS; pop2
	load byte from array	baload	8	lb	NOS=*NOS[TOS]; pop
	store byte into array	bastore	8	sb	*NNOS[NOS]=TOS; pop2
	load immediate	bipush I8, sipush I16	16, 24	addi	push; TOS=I8 or I16
	load immediate	iconst_{–1,0,1,2,3,4,5}	8	addi	push; TOS={–1,0,1,2,3,4,5}
Logical	and	iand	8	and	NOS=TOS&NOS; pop
	or	ior	8	or	NOS=TOS|NOS; pop
	shift left	ishl	8	sll	NOS=NOS << TOS; pop
	shift right	iushr	8	srl	NOS=NOS >> TOS; pop
Conditional branch	branch on equal	if_icompeq I16	24	beq	if TOS == NOS, go to I16; pop2
	branch on not equal	if_icompne I16	24	bne	if TOS != NOS, go to I16; pop2
	compare	if_icomp{lt,le,gt,ge} I16	24	slt	if TOS {<,<=,>,>=} NOS, go to I16; pop2
Unconditional jump	jump	goto I16	24	j	go to I16
	return	ret, ireturn	8	jr	
	jump to subroutine	jsr I16	24	jal	go to I16; push; TOS=PC+3

Starting Java Applications



Just In Time (*JIT*) compiler translates bytecode into machine language just before execution

And, in Conclusion, …

- Assemblers can enhance machine instruction set to help assembly-language programmer
- Translate from text that easy for programmers to understand into code that machine executes efficiently: Compilers, Assemblers
- Linkers allow separate translation of modules
- Interpreters for debugging, but slow execution
- Hybrid (Java): Compiler + Interpreter to try to get best of both
- Compiler Optimization to relieve programmer