CS 61C:
Great Ideas in Computer Architecture
More Machine Language, Compilers

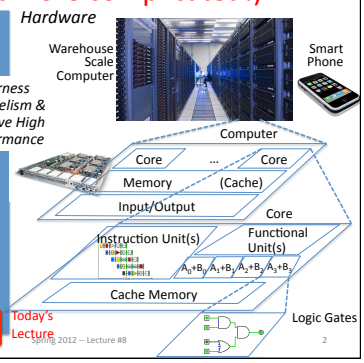
Instructor:
David A. Patterson
http://inst.eecs.Berkeley.edu/~cs61c/sp12

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

Software | *Hardware*

- 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

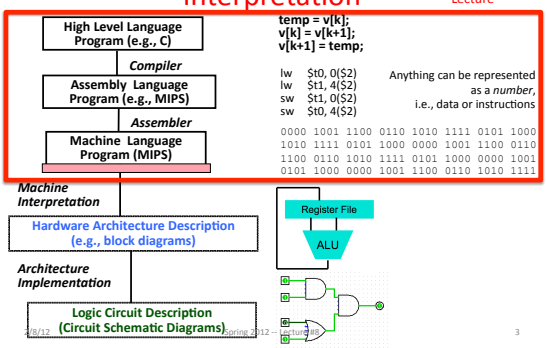
Harness Parallelism & Achieve High Performance



Today's Lecture

Big Idea #1: Levels of Representation/ Interpretation

Today's Lecture



Agenda

- Review
- Instructions as Numbers
- Administrivia
- Assemblers
- Compilers and Linkers
- Technology Break
- Compilers vs. Interpreters
- Compiler Optimization?

Review

- Program can interpret binary number as unsigned integer, two's complement signed integer, floating point number, ASCII characters, Unicode characters, …
- Integers have largest positive and largest negative numbers, but represent all in between
  - Two's comp. weirdness is one extra negative numInteger and floating point operations can lead to results too big to store within their representations: overflow/underflow
- Floating point is an *approximation* of reals
  - 2^{32} patterns to represent reals from –∞ to +∞
- Everything is a (binary) number in a computer

Instructions as Numbers

- Instructions are also kept as binary numbers in memory
  - Stored program concept
  - As easy to change programs as it is to change data
- Register names mapped to numbers
- Need to map instruction operation to a part of number

Names of MIPS fields

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- *op*: Basic operation of instruction, or *opcode*
- *rs*: 1st register source operand
- *rt*: 2nd register source operand.
- *rd*: register destination operand (result of operation)
- *shamt*: Shift amount.
- *funct*: Function. This field, often called *function code*, selects the specific variant of the operation in the op field

What about Load, Store, Immediate, Branches, Jumps?

- Fields for constants only 5 bits (-16 to +15)
  - Too small for many common cases
- #1 Simplicity favors regularity (all instructions use one format) vs. #3 Make common case fast (multiple instruction formats)?
- 4th Design Principle: *Good design demands good compromises*
- Better to have multiple instruction formats and keep all MIPS instructions same *size*
  - All MIPS instructions are 32 bits or 4 bytes

Names of MIPS Fields in I-type

op	rs	rt	address or constant
6 bits	5 bits	5 bits	16 bits

- *op*: Basic operation of instruction, or *opcode*
- *rs*: 1st register source operand
- *rt*: 2nd register source operand for branches but register destination operand for lw, sw, and immediate operations
- *Address/constant*: 16-bit two's complement number
  - Note: equal in size of rd, shamt, funct fields

Register (R), Immediate (I), Jump (J) Instruction Formats

	op	rs	rt	rd	shamt	funct
R-type	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

	op	rs	rt	address or constant
I-type	6 bits	5 bits	5 bits	16 bits

- Now loads, stores, branches, and immediates can have 16-bit two's complement address or constant: -32,768 (-2^{15}) to +32,767 ($2^{15}-1$)
- What about jump, jump and link?

	op	address
J-type	6 bits	26 bits

Addressing in Branches

	op	rs	rt	address or constant
I-type	6 bits	5 bits	5 bits	16 bits

- Programs much bigger than 2^{16} bytes, but branch address must fit in 16-bit field
  - Must specify a register for branch addresses for big programs: PC = Register + Branch address
  - Which register?
- Conditional branching for IF-statement, loops
  - Tend to be near branches; ½ within 16 instructions
- Idea: *PC-relative branching*

Addressing in Branches

	op	rs	rt	address or constant
I-type	6 bits	5 bits	5 bits	16 bits

- Hardware increments PC early, so relative address is PC = (PC + 4) + Branch address
- Another optimization since all MIPS instructions 4 bytes long?
- Multiply value in branch address field by 4!
- MIPS PC-relative branching
  PC = (PC + 4) + (Branch address*4)

Addressing in Jumps

J-type	op	address
	6 bits	26 bits

- Same trick for Jumps, Jump and Link
  PC = Jump address * 4
- Since PC = 32 bits, and Jump address * 4 = 28 bits, what about other 4 bits?
- Jump and Jump and Link only changes bottom 28 bits of PC

Encoding of MIPS Instructions: Must Be Unique!

Instruction	Format	op	rs	rt	rd	shamt	funct	address
addu	R	0	reg	reg	reg	0	33_{ten}	n.a.
subu	R	0	reg	reg	reg	0	35_{ten}	n.a.
sltu	R	0	reg	reg	reg	0	43_{ten}	n.a.
sll	R	0	n.a.	reg	reg	constant	0_{ten}	n.a.
addi unsigned	I	9_{ten}	reg	reg	n.a.	n.a.	n.a.	constant
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
beq	I	4_{ten}	reg	reg	n.a.	n.a.	n.a.	address
bne	I	5_{ten}	reg	reg	n.a.	n.a.	n.a.	address
j (jump)	J	2_{ten}	n.a.	n.a.	n.a.	n.a.	n.a.	address
jal	J	3_{ten}	n.a.	n.a.	n.a.	n.a.	n.a.	address
jr (jump reg)	R	0	reg	reg	reg	0	8_{ten}	n.a.

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

Converting C to MIPS Machine code

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

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

Student Roulette?

Format?

lw $t0,1200($t1) _

addu $t0,$s2,$t0 _

sw $t0,1200($t1) _

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

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

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

Student Roulette?

Format?

lw $t0,1200($t1) I

addu $t0,$s2,$t0 _

sw $t0,1200($t1) _

		9	8	1200		

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

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

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

Student Roulette?

Format?

lw $t0,1200($t1) I

addu $t0,$s2,$t0 R

sw $t0,1200($t1) _

		9	8	1200		
		18	8	8	0	

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

Administrivia

- Lab #4 posted
- Project #1 Due Sunday @ 11:59:59
- This week should be easier – given jar file of MR
  - Run at scale, and compare to your code
- Midterm is now on the horizon:
  - No discussion during exam week
  - TA Review: Su, Mar 4, starting 2 PM, 2050 VLSB
  - Exam: Tu, Mar 6, 6:40-9:40 PM, 2050 VLSB (room change)
  - Small number of special consideration cases, due to class conflicts, etc.—contact me

My Life Beyond Computing

- 1935 Ford Station Wagon
- 221 cubic inch (85 HP) original Ford Flathead V8 Engine

My Life Beyond Computing

- 1938 Ford Cabriolet
- 454 cubic inch (7500 cc) "Big Block" Chevy V8 Engine

Converting C to MIPS Machine code

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

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

Student Roulette?

Format?

lw $t0,1200($t1)		9	8	1200			
addu $t0,$s2,$t0		18	8	8	0		
sw $t0,1200($t1)		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 C to MIPS Machine code

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

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

Student Roulette?

Format?

lw $t0,1200($t1)		9	8	1200			
addu $t0,$s2,$t0		18	8	8	0		
sw $t0,1200($t1)		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 C to MIPS Machine code

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

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

Student Roulette?

Format?

lw $t0,1200($t1)		9	8	1200			
addu $t0,$s2,$t0		18	8	0	0		
sw $t0,1200($t1)		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?
800	sll $t1,$s3,2	_
804	addu $t1,$t1,$s6	_
808	lw $t0,0($t1)	_
812	bne $t0,$s5, Exit	_
816	addiu $s3,$s3,1	_
820	j Loop	_
	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

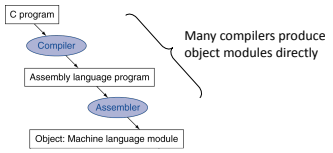
- Can create a 32-bit constant from two 32-bit MIPS instructions
- *Load Upper Immediate (*`lui` *or* "Louie") puts 16 bits into upper 16 bits of destination register
- MIPS to load 32-bit constant into register $s0? $0000\ 0000\ 0011\ 1101\ 0000\ 1001\ 0000\ 0000_{two}$

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

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

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

Assembler Pseudoinstructions

- Most assembler instructions represent machine instructions one-to-one
- Pseudoinstructions: 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 Pseudoinstructions

Student Roulette?

- 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`	_____

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

Separate Compilation and Assembly

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

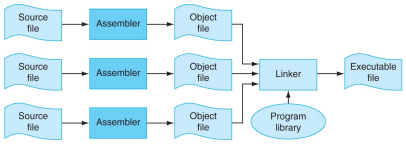
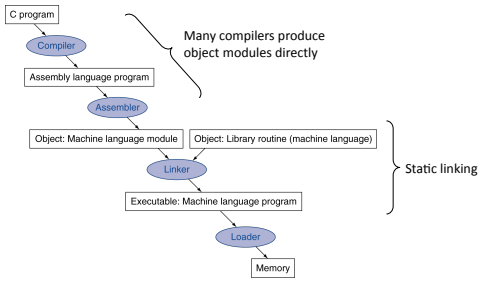


FIGURE B.1.1 The process that produces an executable file. An assembler translates a file of assembly language into an object file, which is linked with other files and libraries into an executable file. Copyright © 2009 Elsevier, Inc. All rights reserved.

Translation and Startup

§2.12 Translating and Starting a Program



Linker Stitches Files Together

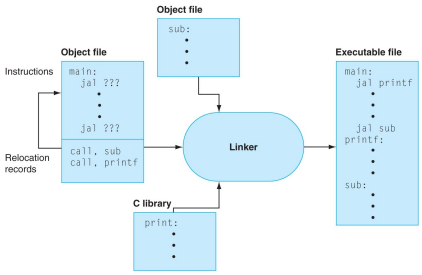


FIGURE B.3.1 The linker searches a collection of object fi les and program libraries to find nonlocal routines used in a program, combines them into a single executable file, and resolves references between routines in different files. Copyright © 2009 Elsevier, Inc. All rights reserved.

Linking Object Modules

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

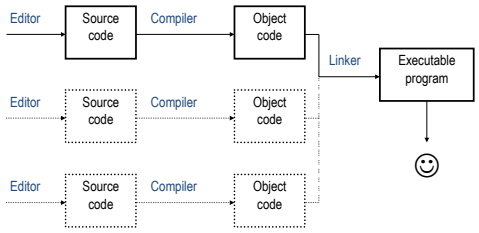
Loading a Program

- Load from image file on disk into memory
  1. Read header to determine segment sizes
  2. Create virtual address space (cover 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

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*.
- The text must comply with the syntax rules of whichever programming language it is written in.
- A 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
```

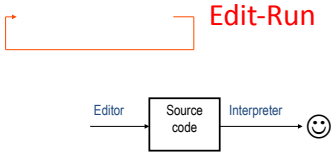
-O2 optimized MIPS Code

```
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
      .
```


What's an Interpreter?

- It reads and executes source statements executed one at a time
  - No linking
  - No machine code generation, so more portable
- Start executing quicker, but run much more slowly than compiled code
- Performing the actions straight from the text allows better error checking and reporting to be done
- The 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

Interpreted Languages: Edit-Run



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

Compiler vs. Interpreter Disadvantages

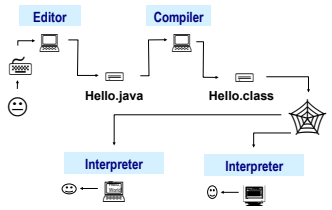
Compilation:
- Harder to debug program
- Takes longer to change source code, recompile, and relink

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

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

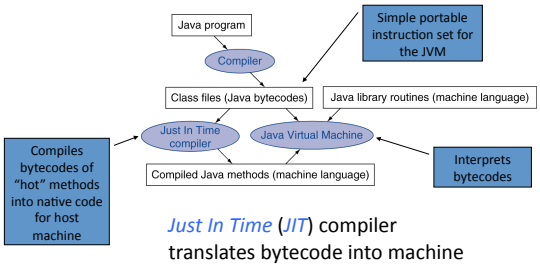
JVM uses Stack vs. Registers

```
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

Review

- Everything is a (binary) number in a computer
  - Instructions and data; stored program concept
- 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