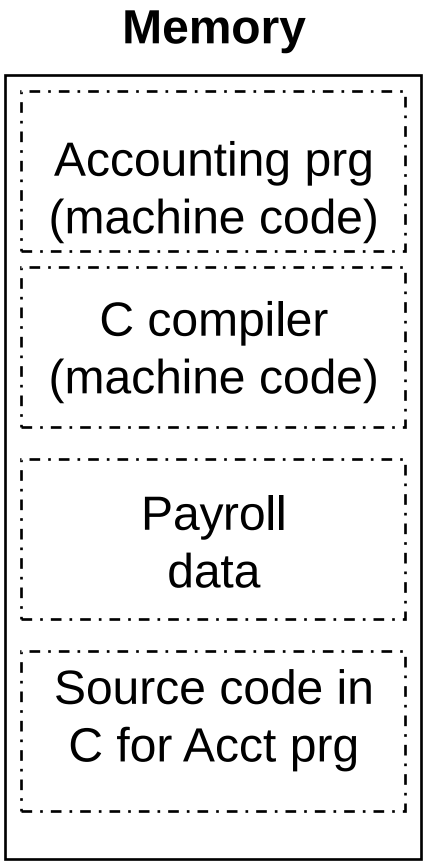
Two Key Principles of Machine Design

1. Instructions are represented as numbers and, as such, are indistinguishable from data

2. Programs are stored in alterable memory (that can be read or written to) just like data

- Stored-program concept
  - Programs can be shipped as files of binary numbers – binary compatibility
  - Computers can inherit ready-made software provided they are compatible with an existing ISA – leads industry to align around a small number of ISAs



Memory

Accounting prg
(machine code)

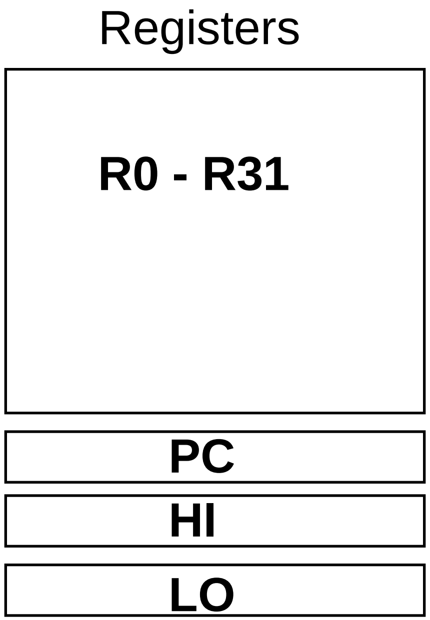
C compiler
(machine code)

Payroll
data

Source code in
C for Acct prg

MIPS-32 ISA

- Instruction Categories
  - Computational
  - Load/Store
  - Jump and Branch
  - Floating Point
    - coprocessor
  - Memory Management
  - Special


Registers

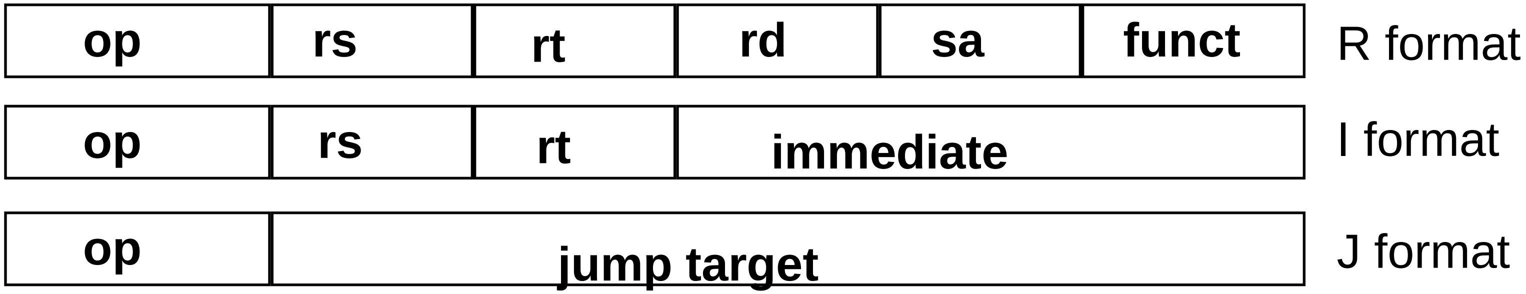
R0 - R31

PC

HI

LO

3 Instruction Formats: **all 32 bits wide**


op | **rs** | **rt** | **rd** | **sa** | **funct** — R format

op | **rs** | **rt** | **immediate** — I format

op | **jump target** — J format

MIPS Instruction Fields

- MIPS fields are given names to make them easier to refer to

op	rs	rt	rd	shamt	funct

op	6-bits	opcode that specifies the operation
rs	5-bits	register file address of the first source operand
rt	5-bits	register file address of the second source operand
rd	5-bits	register file address of the result’s destination
shamt	5-bits	shift amount (for shift instructions)
funct	6-bits	function code augmenting the opcode

Byte Addresses

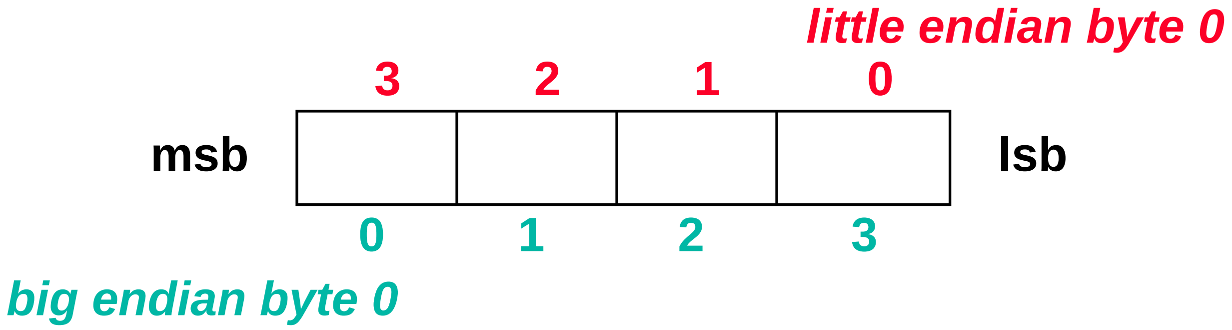
- Since 8-bit bytes are so useful, most architectures address individual bytes in memory
  - Alignment restriction - the memory address of a word must be on natural word boundaries (a multiple of 4 in MIPS-32)

- Big Endian: leftmost byte is word address

  IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA

- Little Endian: rightmost byte is word address

  Intel 80x86, DEC Vax, DEC Alpha (Windows NT)



Review: Unsigned Binary Representation

Hex	Binary	Decimal
0x00000000	0…0000	0
0x00000001	0…0001	1
0x00000002	0…0010	2
0x00000003	0…0011	3
0x00000004	0…0100	4
0x00000005	0…0101	5
0x00000006	0…0110	6
0x00000007	0…0111	7
0x00000008	0…1000	8
0x00000009	0…1001	9
	…	
0xFFFFFFFC	1…1100	$2^{32} - 4$
0xFFFFFFFD	1…1101	$2^{32} - 3$
0xFFFFFFFE	1…1110	$2^{32} - 2$
0xFFFFFFFF	1…1111	$2^{32} - 1$

2^{31}	2^{30}	2^{29}	. . .	2^{3}	2^{2}	2^{1}	2^{0}	bit weight
31	30	29	. . .	3	2	1	0	bit position
1	**1**	**1**	**. . .**	**1**	**1**	**1**	**1**	**bit**

⇩

1 0 0 0 . . . 0 0 0 0 - 1

⇩

2^{32} - 1

Review: Signed Binary Representation

	2'sc binary	decimal
-2^3 =	1000	-8
$-(2^3 - 1)$ =	1001	-7
	1010	-6
	1011	-5
	1100	-4
	1101	-3
	1110	-2
	1111	-1
	0000	0
	0001	1
	0010	2
	0011	3
	0100	4
	0101	5
	0110	6
$2^3 - 1$ =	0111	7

complement all the bits

0101

and add a 1

0110

1011

and add a 1

1010

complement all the bits

MIPS Shift Operations

- Need operations to pack and unpack 8-bit characters into 32-bit words

- Shifts move all the bits in a word left or right

```
sll $t2, $s0, 8     #$t2 = $s0 << 8 bits

srl $t2, $s0, 8     #$t2 = $s0 >> 8 bits
```

- Instruction Format (R format)

0		16	10	8	0x00

- Such shifts are called logical because they fill with zeros
  - Notice that a 5-bit shamt field is enough to shift a 32-bit value $2^5 - 1$ or 31 bit positions

MIPS Logical Operations

- There are a number of bit-wise logical operations in the MIPS ISA

```
and $t0, $t1, $t2  #$t0 = $t1 & $t2
or  $t0, $t1, $t2  #$t0 = $t1 | $t2
nor $t0, $t1, $t2  #$t0 = not($t1 | $t2)
```

- Instruction Format (R format)

0	9	10	8	0	0x24

```
andi $t0, $t1, 0xFF00   #$t0 = $t1 & ff00
ori  $t0, $t1, 0xFF00   #$t0 = $t1 | ff00
```

- Instruction Format (I format)

0x0D	9	8	0xFF00

Instructions for Accessing Procedures

- MIPS procedure call instruction:

```
jal   ProcedureAddress     #jump and link
```

- Saves PC+4 in register $ra to have a link to the next instruction for the procedure return
- Machine format (J format):

0x03	26 bit address

- Then can do procedure return with a

```
jr    $ra                  #return
```

- Instruction format (R format):

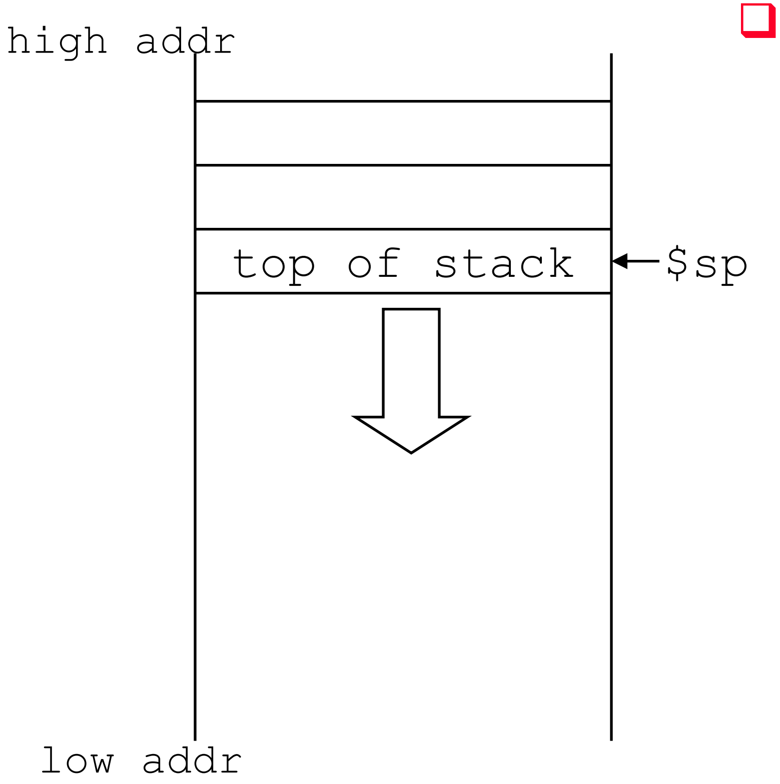
0	31				0x08

Six Steps in Execution of a Procedure

1. Main routine (caller) places parameters in a place where the procedure (callee) can access them
    - `$a0` - `$a3`: four argument registers
2. Caller transfers control to the callee
3. Callee acquires the storage resources needed
4. Callee performs the desired task
5. Callee places the result value in a place where the caller can access it
    - `$v0` - `$v1`: two value registers for result values
6. Callee returns control to the caller
    - `$ra`: one return address register to return to the point of origin

Aside: Spilling Registers

- What if the callee needs to use more registers than allocated to argument and return values?
  - callee uses a stack – a last-in-first-out queue



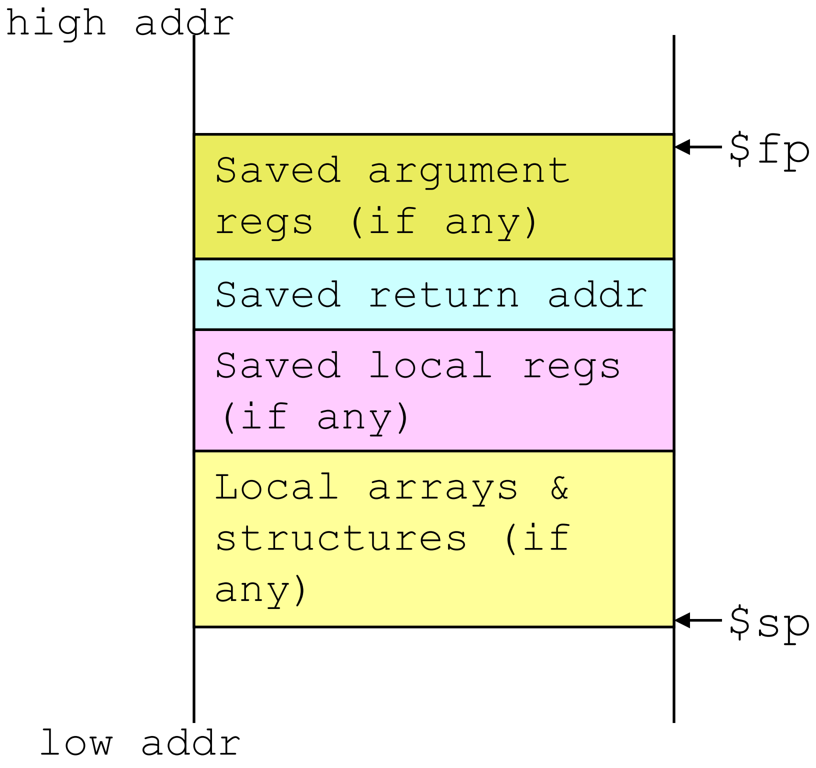
- One of the general registers, `$sp` (`$29`), is used to address the stack (which “grows” from high address to low address)
  - add data onto the stack – push

    `$sp` = `$sp` – 4
    data on stack at new $sp

  - remove data from the stack – pop

    data from stack at $sp
    `$sp` = `$sp` + 4

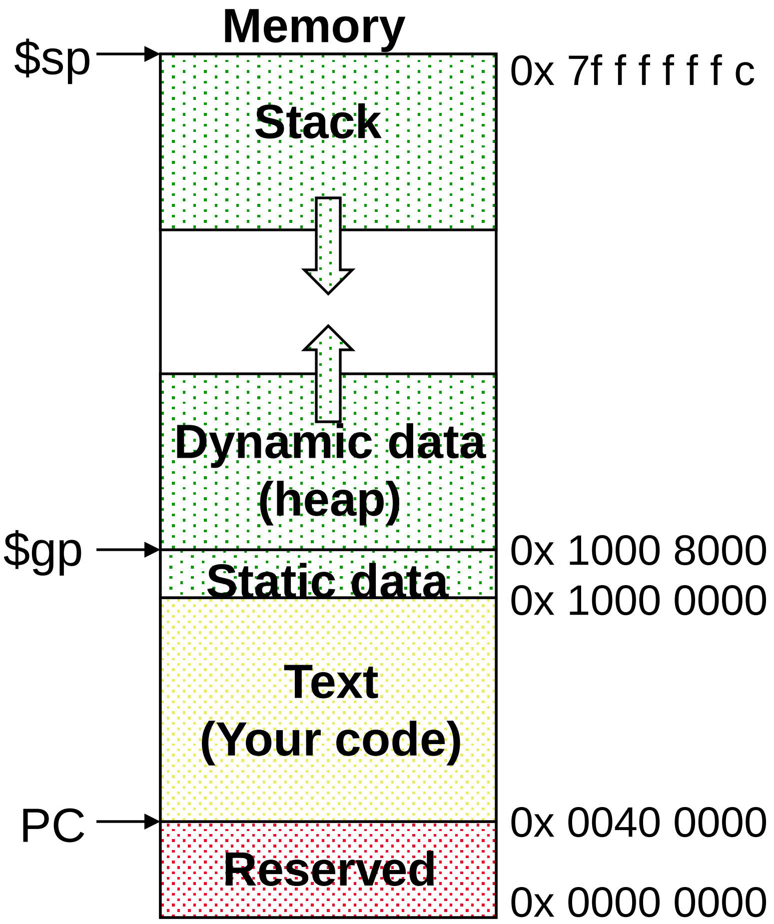
Aside: Allocating Space on the Stack



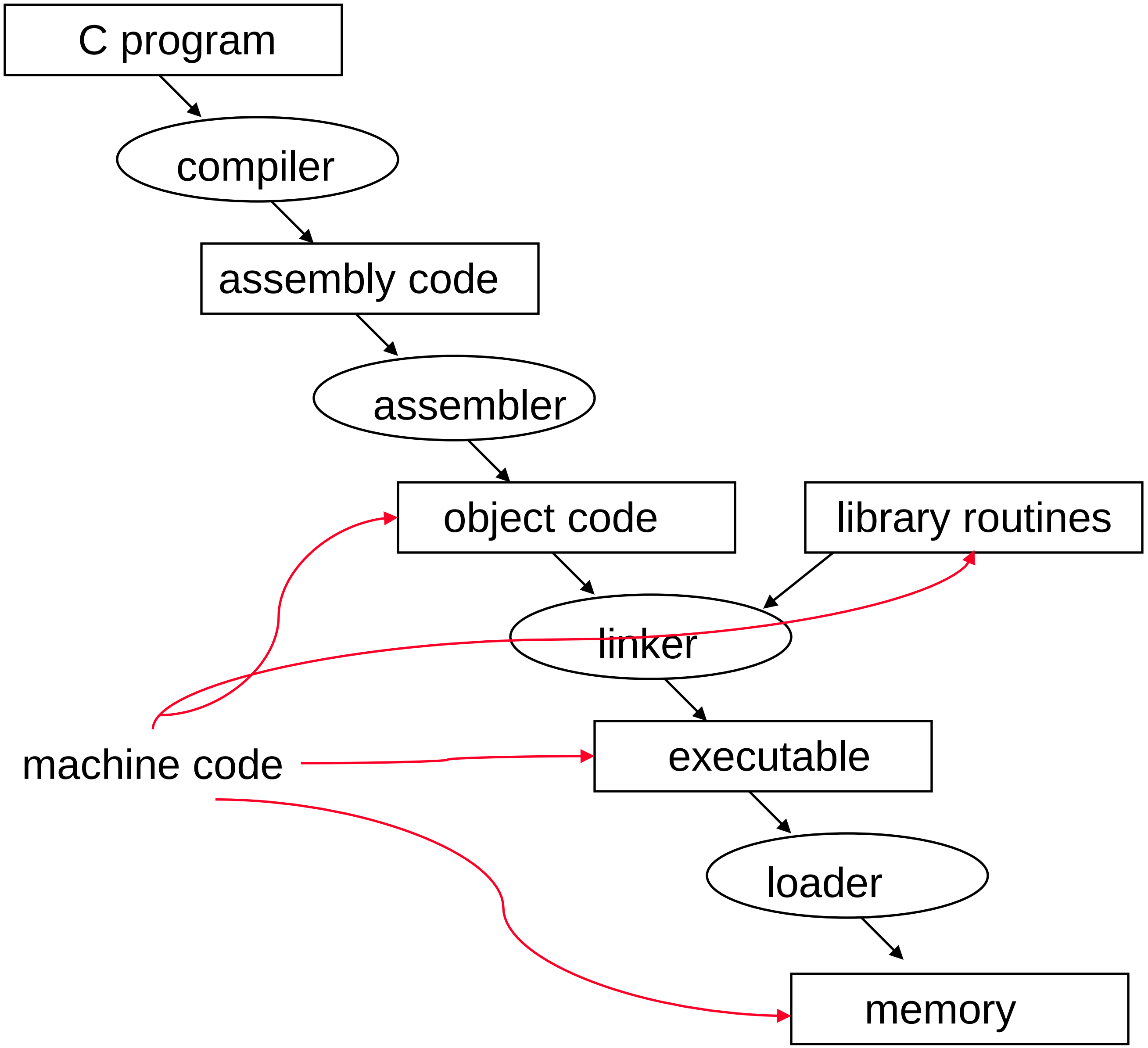
- The segment of the stack containing a procedure’s saved registers and local variables is its procedure frame (aka activation record)
  - The frame pointer (`$fp`) points to the first word of the frame of a procedure – providing a stable “base” register for the procedure
    - `$fp` is initialized using `$sp` on a call and `$sp` is restored using `$fp` on a return

Aside: Allocating Space on the Heap

- Static data segment for constants and other static variables (e.g., arrays)
- Dynamic data segment (aka heap) for structures that grow and shrink (e.g., linked lists)
  - Allocate space on the heap with `malloc()` and free it with `free()` in C



The C Code Translation Hierarchy



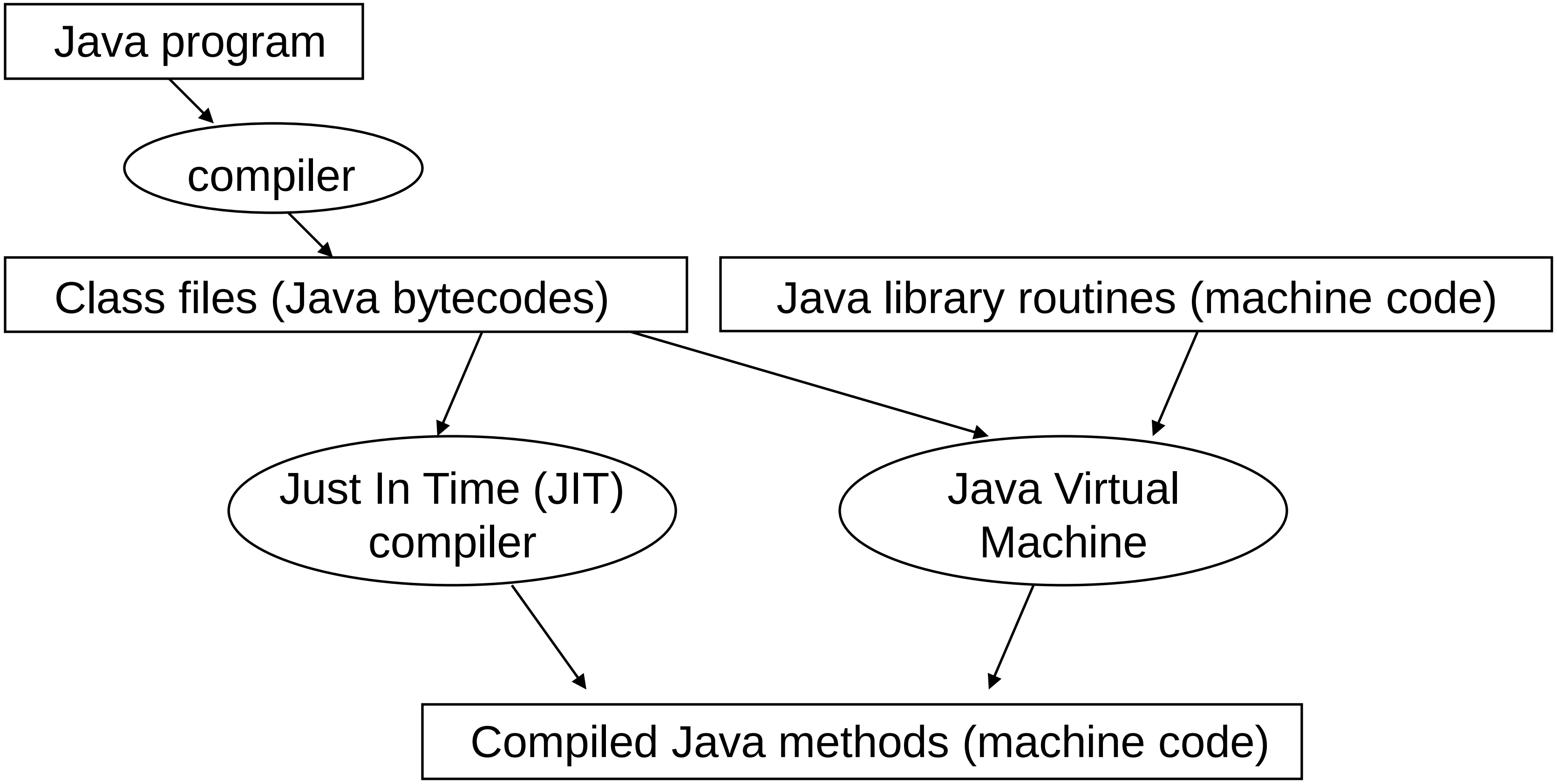
Compiler Benefits

- Comparing performance for bubble (exchange) sort
  - To sort 100,000 words with the array initialized to random values on a Pentium 4 with a 3.06 clock rate, a 533 MHz system bus, with 2 GB of DDR SDRAM, using Linux version 2.4.20

gcc opt	Relative performance	Clock cycles (M)	Instr count (M)	CPI
None	1.00	158,615	114,938	1.38
O1 (medium)	2.37	66,990	37,470	1.79
O2 (full)	2.38	66,521	39,993	1.66
O3 (proc mig)	2.41	65,747	44,993	1.46

- The unoptimized code has the best CPI, the O1 version has the lowest instruction count, but the O3 version is the fastest. Why?

The Java Code Translation Hierarchy



Java program

compiler

Class files (Java bytecodes)

Java library routines (machine code)

Just In Time (JIT) compiler

Java Virtual Machine

Compiled Java methods (machine code)

Sorting in C versus Java

- Comparing performance for two sort algorithms in C and Java
  - The JVM/JIT is Sun/Hotspot version 1.3.1/1.3.1

	Method	Opt	Bubble	Quick	Speedup quick vs bubble
			Relative performance		
C	Compiler	None	1.00	1.00	2468
C	Compiler	O1	2.37	1.50	1562
C	Compiler	O2	2.38	1.50	1555
C	Compiler	O3	2.41	1.91	1955
Java	Interpreted		0.12	0.05	1050
Java	JIT compiler		2.13	0.29	338

- Observations?