Chapter 2

Instructions: Language of the Computer

Instruction Set

- The collection of instructions of a computer
- Different computers have different instruction sets
  - But with many aspects in common
- Early computers had very simple instruction sets
  - Simplified implementation
- Many modern computers also have simple instruction sets

The MIPS Instruction Set

- Used as the example throughout the course
- Stanford MIPS commercialized by MIPS Technologies (www.mips.com)
- Large share of embedded core market
  - Applications in consumer electronics, network/storage equipment, cameras, printers, …
- Typical of many modern ISAs
  - See MIPS Reference Data tear-out card, and Appendices B and E

Arithmetic Operations

- Add and subtract, three operands
  - Two sources and one destination

```
add a,b,c # a gets b + c
```

- All arithmetic operations have this form
- *Design Principle 1:* Simplicity favors regularity
  - Regularity makes implementation simpler
  - Simplicity enables higher performance at lower cost

MIPS operands

Name	Example	Comments
32 registers	\$s0–\$s7, \$t0–\$t9, \$zero, \$a0–\$a3, \$v0–\$v1, \$gp, \$fp, \$sp, \$ra, \$at	Fast locations for data. In MIPS, data must be in registers to perform arithmetic, register \$zero always equals 0, and register \$at is reserved by the assembler to handle large constants.
2^{30} memory words	Memory[0], Memory[4], . . . , Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, arrays, and spilled registers.

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1,$s2,$s3`	\$s1 = \$s2 + \$s3	Three register operands
	subtract	`sub $s1,$s2,$s3`	\$s1 = \$s2 – \$s3	Three register operands
	add immediate	`addi $s1,$s2,20`	\$s1 = \$s2 + 20	Used to add constants
Data transfer	load word	`lw $s1,20($s2)`	\$s1 = Memory[\$s2 + 20]	Word from memory to register
	store word	`sw $s1,20($s2)`	Memory[\$s2 + 20] = \$s1	Word from register to memory
	load half	`lh $s1,20($s2)`	\$s1 = Memory[\$s2 + 20]	Halfword memory to register
	load half unsigned	`lhu $s1,20($s2)`	\$s1 = Memory[\$s2 + 20]	Halfword memory to register
	store half	`sh $s1,20($s2)`	Memory[\$s2 + 20] = \$s1	Halfword register to memory
	load byte	`lb $s1,20($s2)`	\$s1 = Memory[\$s2 + 20]	Byte from memory to register
	load byte unsigned	`lbu $s1,20($s2)`	\$s1 = Memory[\$s2 + 20]	Byte from memory to register
	store byte	`sb $s1,20($s2)`	Memory[\$s2 + 20] = \$s1	Byte from register to memory
	load linked word	`ll $s1,20($s2)`	\$s1 = Memory[\$s2 + 20]	Load word as 1st half of atomic swap
	store condition. word	`sc $s1,20($s2)`	Memory[\$s2+20]=\$s1;\$s1=0 or 1	Store word as 2nd half of atomic swap
	load upper immed.	`lui $s1,20`	\$s1 = 20 * 2^{16}	Loads constant in upper 16 bits
Logical	and	`and $s1,$s2,$s3`	\$s1 = \$s2 & \$s3	Three reg. operands; bit-by-bit AND
	or	`or $s1,$s2,$s3`	\$s1 = \$s2 | \$s3	Three reg. operands; bit-by-bit OR
	nor	`nor $s1,$s2,$s3`	\$s1 = ~ (\$s2 | \$s3)	Three reg. operands; bit-by-bit NOR
	and immediate	`andi $s1,$s2,20`	\$s1 = \$s2 & 20	Bit-by-bit AND reg with constant
	or immediate	`ori $s1,$s2,20`	\$s1 = \$s2 | 20	Bit-by-bit OR reg with constant
	shift left logical	`sll $s1,$s2,10`	\$s1 = \$s2 << 10	Shift left by constant
	shift right logical	`srl $s1,$s2,10`	\$s1 = \$s2 >> 10	Shift right by constant
Conditional branch	branch on equal	`beq $s1,$s2,25`	if (\$s1 == \$s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	`bne $s1,$s2,25`	if (\$s1!= \$s2) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	`slt $s1,$s2,$s3`	if (\$s2 < \$s3) \$s1 = 1; else \$s1 = 0	Compare less than; for beq, bne
	set on less than unsigned	`sltu $s1,$s2,$s3`	if (\$s2 < \$s3) \$s1 = 1; else \$s1 = 0	Compare less than unsigned
	set less than immediate	`slti $s1,$s2,20`	if (\$s2 < 20) \$s1 = 1; else \$s1 = 0	Compare less than constant
	set less than immediate unsigned	`sltiu $s1,$s2,20`	if (\$s2 < 20) \$s1 = 1; else \$s1 = 0	Compare less than constant unsigned
Unconditional jump	jump	`j 2500`	go to 10000	Jump to target address
	jump register	`jr $ra`	go to \$ra	For switch, procedure return
	jump and link	`jal 2500`	\$ra = PC + 4; go to 10000	For procedure call

Arithmetic Example

- C code:

```
f = (g + h) - (i + j);
```

- Compiled MIPS code:

```
add t0, g, h   # temp t0 = g + h
add t1, i, j   # temp t1 = i + j
sub f, t0, t1  # f = t0 - t1
```

Register Operands

- Arithmetic instructions use register operands
- MIPS has a 32 by 32-bit register file
  - Used for frequently accessed data
  - Numbered 0 to 31
  - 32-bit data called a "word"
- Assembler names
  - $t0, $t1, …, $t9 for temporary values
  - $s0, $s1, …, $s7 for saved variables
- *Design Principle 2:* Smaller is faster
  - c.f. main memory: millions of locations

Register Operand Example

- C code:

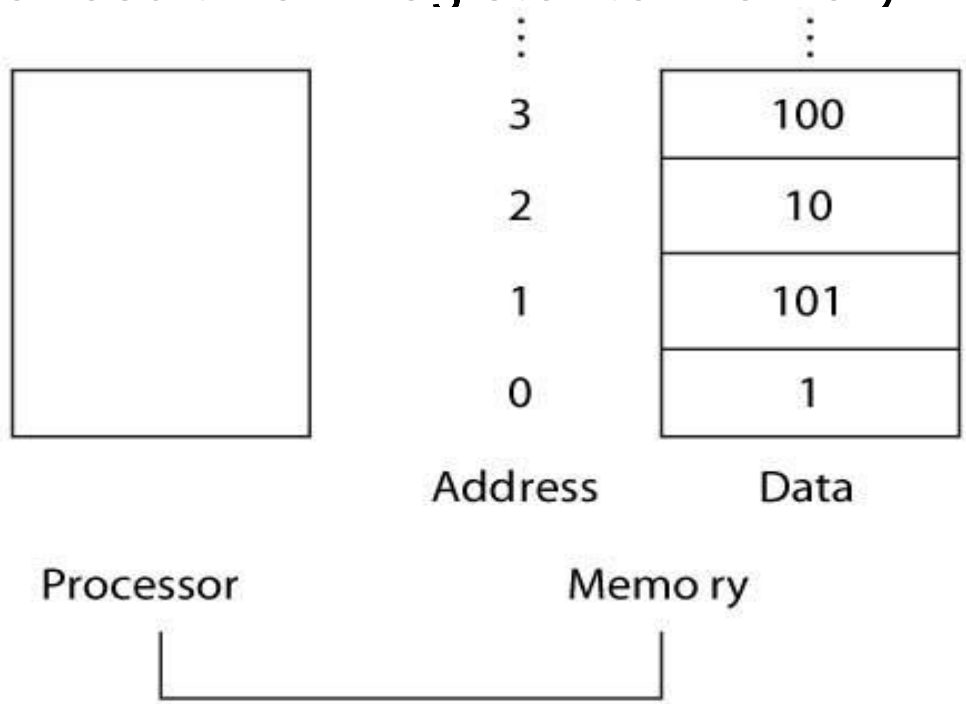
```
f = (g + h) - (i + j);
```

  - f, …, j in $s0, …, $s4
- Compiled MIPS code:

```
add $t0, $s1, $s2
add $t1, $s3, $s4
sub $s0, $t0, $t1
```

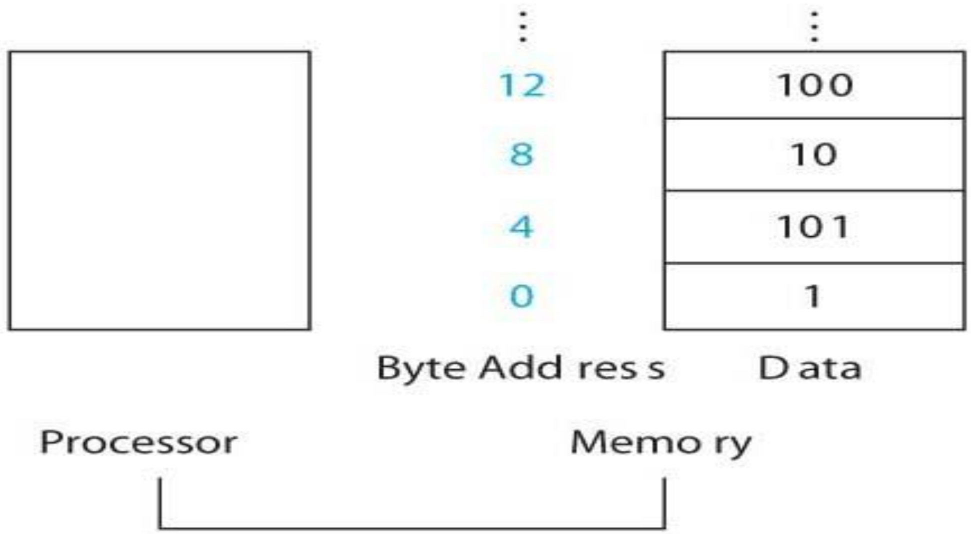
Memory Operands (1)

- Main memory used for composite data
  - Arrays, structures, dynamic data
- To apply arithmetic operations
  - Load values from memory into registers
  - Store result from register to memory



Memory Operands (2)

- Memory is byte addressed
  - Each address identifies an 8-bit byte
- Words are aligned in memory
  - Address must be a multiple of 4
- MIPS is Big Endian
  - Most-significant byte at least address of a word
  - *c.f.* Little Endian: least-significant byte at least address



Memory Operands (3)

- Data is transferred between memory and register using data transfer instructions: lw and sw

Category	Instruction	Example	Meaning	Comments
Data transfer	load word	`lw $s1,100($s2)`	`$s1 ← memory[$s2+100]`	Memory to Register
	store word	`sw $s1,100($s2)`	`memory[$s2+100]← $s1`	Register to memory

  - $s1 is receiving register
  - $s2 is base address of memory, 100 is called the offset, so ($s2+100) is the address of memory location

Memory Operand Example(1)

- C code:

  `g = h + A[8];`

  - g in \$s1, h in \$s2, base address of A in \$s3
- Compiled MIPS code:
  - Index 8 requires offset of 32
    - 4 bytes per word

```
lw  $t0, 32($s3)    # load word
add $s1, $s2, $t0
```

offset

base register

Memory Operand Example(2)

- C code:

```
A[12] = h + A[8];
```

  - h in $s2, base address of A in $s3
- Compiled MIPS code:
  - Index 8 requires offset of 32

```
lw  $t0, 32($s3)    # load word
add $t0, $s2, $t0
sw  $t0, 48($s3)    # store word
```

Registers vs. Memory

- Registers are faster to access than memory
- Operating on memory data requires loads and stores
  - More instructions to be executed
- Compiler must use registers for variables as much as possible
  - Only spill to memory for less frequently used variables
  - Register optimization is important!

Immediate Operands

- Constant data specified in an instruction

```
addi $s3, $s3, 4
```

- No subtract immediate instruction
  - Just use a negative constant

  ```
  addi $s2, $s1, -1
  ```

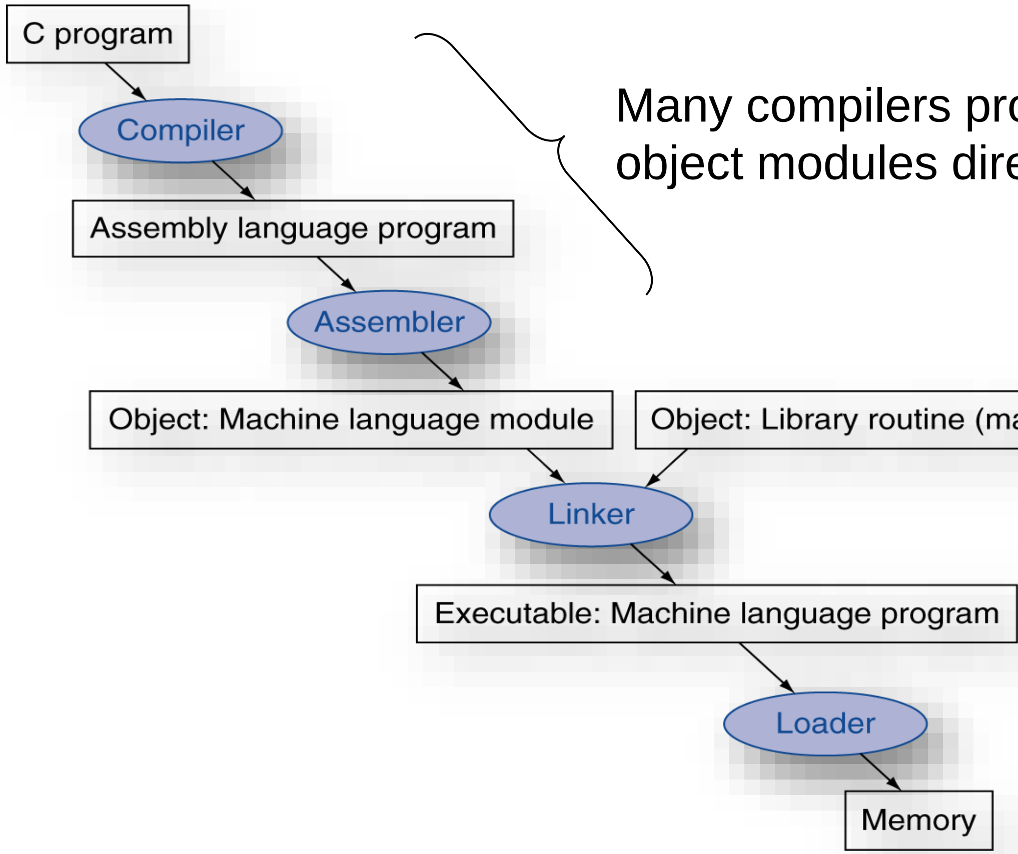
- *Design Principle 3:* Make the common case fast
  - Small constants are common
  - Immediate operand avoids a load instruction

The Constant Zero

- MIPS register 0 ($zero) is the constant 0
  - Cannot be overwritten
- Useful for common operations
  - E.g., move between registers

```
add $t2, $s1, $zero
```

Translation and Startup



UNIX: C source files are named x.c, assembly files are x.s, object files are named x.o, statically linked library routines are x.a, dynamically linked library routes are x.so, and executable fi les by default are called a.out.
MS-DOS uses the .C, .ASM, .OBJ, .LIB, .DLL, and .EXE to the same effect.

Translation

- Assembler (or compiler) translates program into machine instructions
- Linker produces an executable image
- Loader loads from image file on disk into memory

SPIM Simulator

- SPIM is a software simulator that runs assembly language programs
- SPIM is just MIPS spelled backwards
- SPIM can read and immediately execute assembly language files
- Two versions for different machines
  - Unix xspim(used in lab), spim
  - PC/Mac: QtSpim
- Resources and Download
  - http://spimsimulator.sourceforge.net

System Calls in SPIM

- SPIM provides a small set of system-like services through the system call (syscall) instruction.
- Format for system calls
  - Place value of input argument in $a0
  - Place value of system-call-code in $v0
  - Syscall

System Calls

Example: print a string

```
    .data
str:
    .asciiz "answer is:"

    .text
    addi $v0,$zero,4
    la $a0, str
    syscall
```

Service	System Call Code	Arguments	Result
print_int	1	$a0 = integer	
print_float	2	$f12 = float	
print_double	3	$f12 = double	
print_string	4	$a0 = string	
read_int	5		integer (in $v0)
read_float	6		float (in $f0)
read_double	7		double (in $f0)
read_string	8	$a0 = buffer, $a1 = length	
sbrk	9	$a0 = amount	address (in $v0)
exit	10		
print_character	11	$a0 = character	
read_character	12		character (in $v0)
open	13	$a0 = filename,	file descriptor (in $v0)
		$a1 = flags, $a2 = mode	
read	14	$a0 = file descriptor,	bytes read (in $v0)
		$a1 = buffer, $a2 = count	
write	15	$a0 = file descriptor,	bytes written (in $v0)
		$a1 = buffer, $a2 = count	
close	16	$a0 = file descriptor	0 (in $v0)
exit2	17	$a0 = value	

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

Assembler Pseudoinstructions (2)

- Pseudoinstructions give MIPS a richer set of assembly language instructions than those implemented by the hardware.
- Register, $at (assembler temporary), reserved for use by the assembler.
- For productivity, use pseudoinstructions to write assembly programs.
- For performance, use real MIPS instructions

Reading

- Read Appendix A.9 for SPIM
- List of Pseudoinstructions can be found on page 235

Producing an Object Module

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

Linking Object Modules

- Produces an executable file
  1. Merges segments
  2. Resolves labels (determine their addresses)
  3. Patches location-dependent and external refs
- Could leave location dependencies for fixing by a relocating loader
  - But with virtual memory, no need to do this
  - Program can be loaded into absolute location in virtual memory space

Object file header			
	Name	Procedure A	
	Text size	100_{hex}	
	Data size	20_{hex}	
Text segment	Address	Instruction	
	0	`lw $a0, 0($gp)`	
	4	`jal 0`	
	...	...	
Data segment	0	(X)	
	...	...	
Relocation information	Address	Instruction type	Dependency
	0	`lw`	X
	4	`jal`	B
Symbol table	Label	Address	
	X	–	
	B	–	
Object file header			
	Name	Procedure B	
	Text size	200_{hex}	
	Data size	30_{hex}	
Text segment	Address	Instruction	
	0	`sw $a1, 0($gp)`	
	4	`jal 0`	
	...	...	
Data segment	0	(Y)	
	...	...	
Relocation information	Address	Instruction type	Dependency
	0	`sw`	Y
	4	`jal`	A
Symbol table	Label	Address	
	Y	–	
	A	–	

Linking Object Modules

Executable file header		
	Text size	300_{hex}
	Data size	50_{hex}
Text segment	Address	Instruction
	$0040\ 0000_{hex}$	lw \$a0, 8000_{hex}(\$gp)
	$0040\ 0004_{hex}$	jal $40\ 0100_{hex}$
	...	...
	$0040\ 0100_{hex}$	sw \$a1, 8020_{hex}(\$gp)
	$0040\ 0104_{hex}$	jal $40\ 0000_{hex}$
	...	...
Data segment	Address	
	$1000\ 0000_{hex}$	(X)
	...	...
	$1000\ 0020_{hex}$	(Y)
	...	...

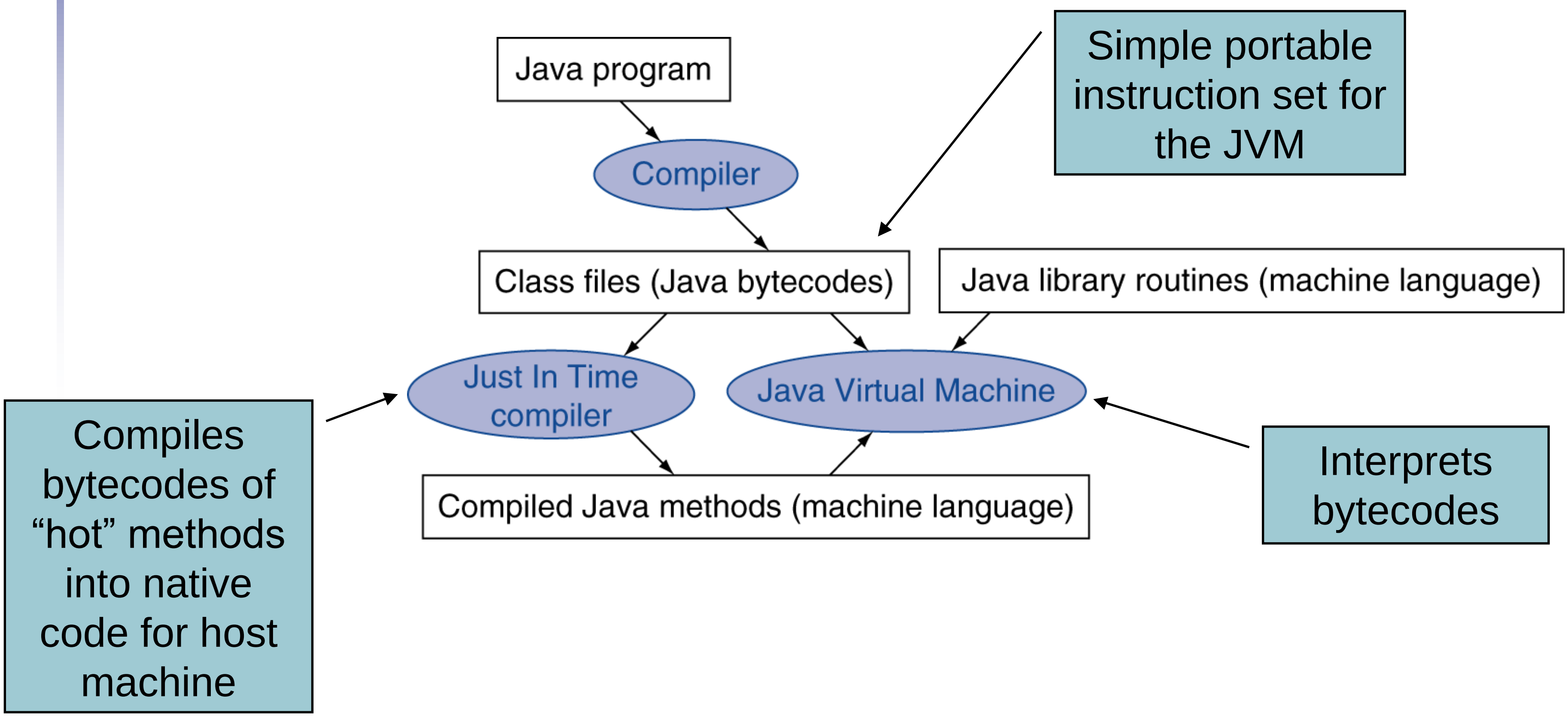
Loading a Program

- Load from file on disk into memory
  1. Read header to determine segment sizes
  2. Create address space for text and data
  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 syscall

Dynamic Linking

- Only link/load library procedure when it is called
  - Requires procedure code to be relocatable
  - Avoids image enlarge caused by static linking of all (transitively) referenced libraries
  - Automatically picks up new library versions

Starting Java Applications



An Example MIPS Program

```
#  Program: (descriptive name)                 Programmer: NAME
#  Due Date:                                   Course: CSE 2021
#  Functional Description:  Find the sum of the integers from 1 to N where
#  N is a value input from the keyboard.
##########################################################################
# Register Usage: $t0 is used to accumulate the sum
#                       $v0 the loop counter, counts down to zero
##########################################################################
# Algorithmic Description in Pseudocode:
# main:    v0 <<  value read from the keyboard (syscall 4)
#           if (v0 < = 0 ) stop
#          t0 = 0;                  # t0 is used to accumulate the sum
#          While (v0 > 0)  { t0 = t0 + v0;  v0 = v0 - 1}
#          Output to monitor syscall(1) << t0;     goto main
##########################################################################
                    .data
prompt:             .asciiz         "\n\n   Please Input a value for N = "
result:             .asciiz          " The sum of the integers from 1 to N is "
bye:                .asciiz          "\n  **** Have a good day **** "
                    .globl          main
```

An Example MIPS Program(2)

```
        .text
main:   li      $v0, 4              # system call code for print_str
        la      $a0, prompt         # load address of prompt into a0
        syscall                     # print the prompt message
        li      $v0, 5              # system call code for read int
        syscall                     # reads a value of N into v0
        blez    $v0,  done          # if ( v0  < = 0 ) go to done
        li      $t0, 0              # clear $t0 to zero
loop:   add     $t0, $t0, $v0       # sum of integers in register $t0
        addi    $v0, $v0, -1        # summing  in reverse order
        bnez    $v0, loop           # branch to loop if $v0 is != zero
        li      $v0, 4              # system call code for print_str
        la      $a0, result         # load address of message into $a0
        syscall                     # print the string
        li      $v0, 1              # system call code for print_int
        move    $a0, $t0            # a0 = $t0
        syscall                     # prints the value  in register $a0
        b       main
done:   li      $v0, 4              # system call code for print_str
        la      $a0, bye            # load address of msg. into $a0
        syscall                     # print the string
        li      $v0, 10             # terminate program
        syscall                     # return control to  system
```