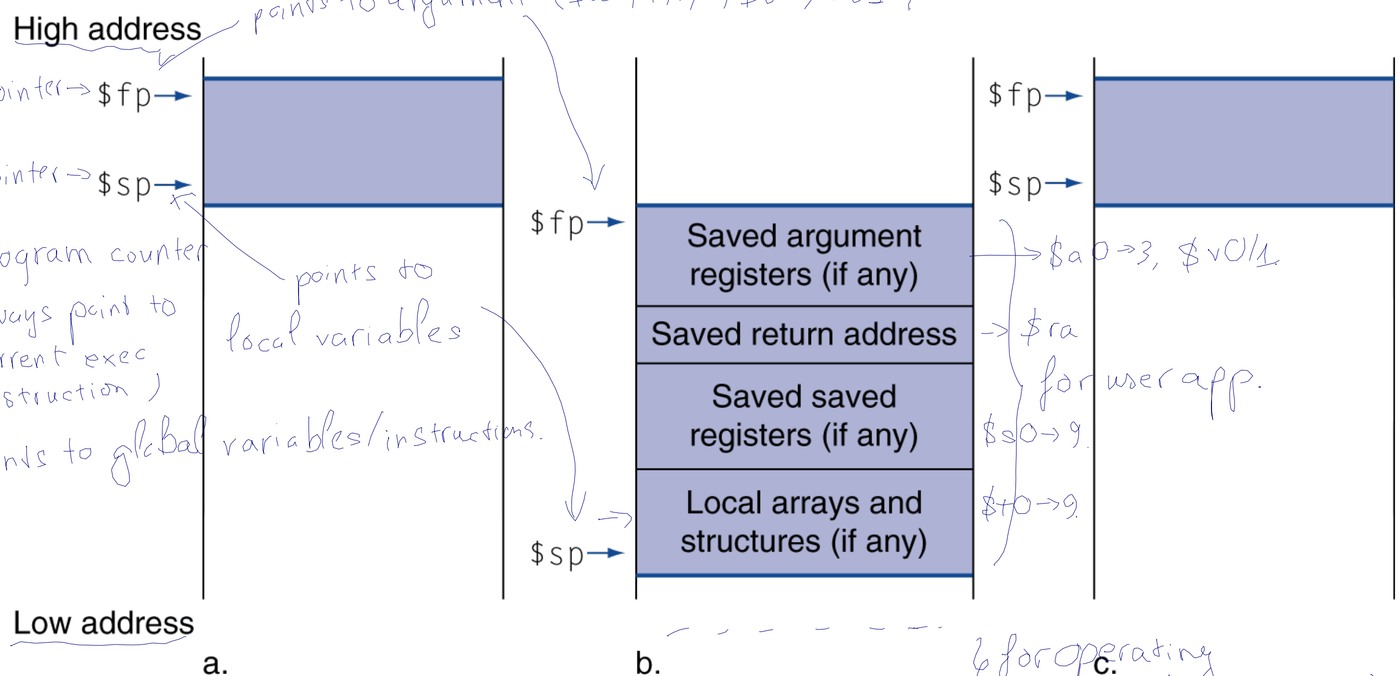
Week 5 : June 11th

Midterm on week 7th.

Unit 5 MIPS Addressing



points to arguments ($a0, 1, 2, 3, $v0, $v1)

frame pointer →

stack pointer →

pc: program counter (always point to current exec instruction)

gp: points to global variables/instructions.

points to local variables

$a0→3, $v0/1

$ra

for user app.

$s0→9

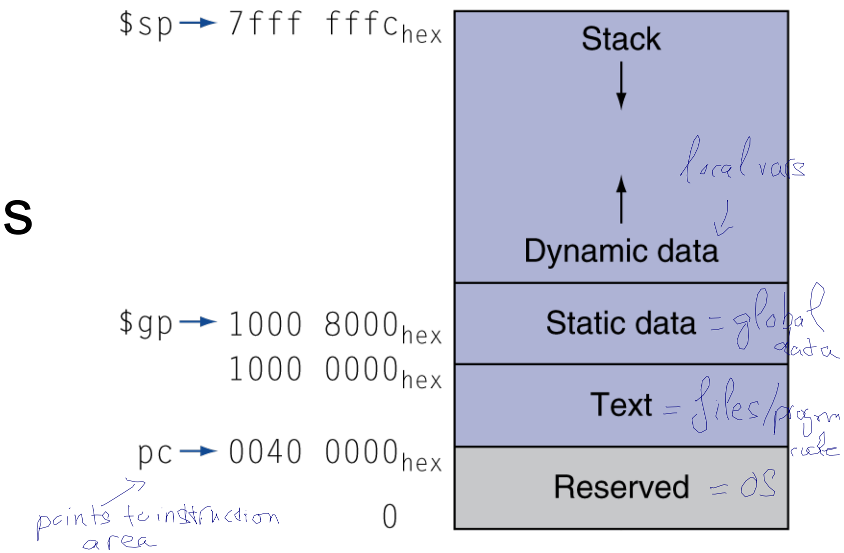
$t0→9

for operating system (very low mem)

- Local data allocated by callee
  - e.g., C automatic variables
- Procedure frame (activation record)
  - Used by some compilers to manage stack storage

Memory Layout

- Text: program code
- Static data: global variables
  - e.g., static variables in C, constant arrays and strings
  - $gp initialized to address allowing ±offsets into this segment
- Dynamic data: heap
  - E.g., malloc in C, new in Java
- Stack: automatic storage



Test: Prg A has n line, each line has 4 bytes. Data starts from adds...
Prg A calls prog B (subroutine), where is A, B in the memory.

Character Data

- Byte-encoded character sets
  - ASCII: 128 characters → 1 byte/char
    - 95 graphic, 33 control
  - Latin-1: 256 characters
    - ASCII, +96 more graphic characters
- Unicode: 32-bit character set ← 4 bytes/char
  - Used in Java, C++ wide characters, …
  - Most of the world's alphabets, plus symbols
  - UTF-8, UTF-16: variable-length encodings

C uses Ascii

Byte/Halfword Operations

- Could use bitwise operations
- MIPS byte/halfword load/store
  - String processing is a common case

similar to lw but load byte only 1 byte

load half(word) = 2 bytes

```
lb rt, offset(rs)      lh rt, offset(rs)
```

  - Sign extend to 32 bits in rt

load byte unsigned.

load halfword unsigned.

```
lbu rt, offset(rs)     lhu rt, offset(rs)
```

  - Zero extend to 32 bits in rt

```
sb rt, offset(rs)      sh rt, offset(rs)
```

  - Store just rightmost byte/halfword

String Copy Example

- C code (naïve):
  - Null-terminated string

copy from y to x

$a0 $a1

```
void strcpy (char x[], char y[])
{ int i;
  i = 0;
  while ((x[i]=y[i])!='\0')
    i += 1;
}
```

$s0

ASCII code '\0' means end of array (C code)

  - Addresses of x, y in $a0, $a1
  - i in $s0

String Copy Example

- MIPS code:

```
strcpy:
     addi $sp, $sp, -4         # adjust stack for 1 item
     sw   $s0, 0($sp)          # save $s0 to $sp
     add  $s0, $zero, $zero # i = 0 = 0+0
L1:  add  $t1, $s0, $a1        # addr of y[i] in $t1
     lbu  $t2, 0($t1)          # $t2 = y[i]
     add  $t3, $s0, $a0        # addr of x[i] in $t3
     sb   $t2, 0($t3)          # x[i] = y[i]
     beq  $t2, $zero, L2       # exit loop if y[i] == 0
     addi $s0, $s0, 1  ← 1 byte for char # i = i + 1
     j    L1                   # next iteration of loop
L2:  lw   $s0, 0($sp)          # restore saved $s0
     addi $sp, $sp, 4          # pop 1 item from stack
     jr   $ra                  # and return
```

character need to use lbu (each char is 1 byte)

do not need to save unless calling subroutine (recursive)

be careful with data type 1 byte = 1 char

32-bit Constants

- Most constants are small
  - 16-bit immediate is sufficient
- For the occasional 32-bit constant

for constant

`lui rt, constant`

lui: load upper immediate
ori:

  - Copies 16-bit constant to left 16 bits of rt
  - Clears right 16 bits of rt to 0

`lhi $s0, 61`	0000 0000 0111 1101 0000 0000 0000 0000
`ori $s0, $s0, 2304`	0000 0000 0111 1101 0000 1001 0000 0000

Branch Addressing

- Branch instructions specify
  - Opcode, two registers, target address
- Most branch targets are near branch
  - Forward or backward

example: beq $s1, $s2, L1

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

- PC-relative addressing
  - Target address = PC + offset × 4
  - PC already incremented by 4 by this time

Jump Addressing

- Jump (`j` and `jal`) targets could be anywhere in text segment
  - Encode full address in instruction

op	address
6 bits	26 bits

main different v/s branch only 16 bits

- (Pseudo)Direct jump addressing
  - Target address = $PC_{31\ldots28}$: (address × 4)

Target Addressing Example

* Most important

- Loop code from earlier example
  - Assume Loop at location 80000 * Immediate field contains the distance in words between PC+4 and the branch target.

$\$t1 = \$s3 \times 2^2$ times 2^2

Base 10

Loop:	sll	$t1, $s3, 2	80000	0	0	19	9	4	0
	add	$t1, $t1, $s6	80004	0	9	22	9	0	32
	lw	$t0, 0($t1)	80008	35	9	8	0		
PC	bne	$t0, $s5, Exit	80012	5	8	21	2		
(PC+4)	addi	$s3, $s3, 1	80016	8	19	19	1		
distance = 2 words	j	Loop	80020	2	20000 ?				
Exit:	…		80024						

$\$t0 = \$s6[\$s3]$ +1

* Goto address : 20000 × 4 = 80000

Notes

bne $s1, $s2, 25 ← jumps to PC+4 + 100
beq $s1, $s2, 25 ←
j 2500 ← jumps to 10000 (do not add 4)
jal 2500 ← { $ra = PC+4; jump to 10000 }

Branching Far Away

- If branch target is too far to encode with 16-bit offset, assembler rewrites the code
- Example

```
beq $s0,$s1, L1
      ↓
bne $s0,$s1, L2
j L1
L2: …
```

(: if the addr is > 16 bits have to use j)

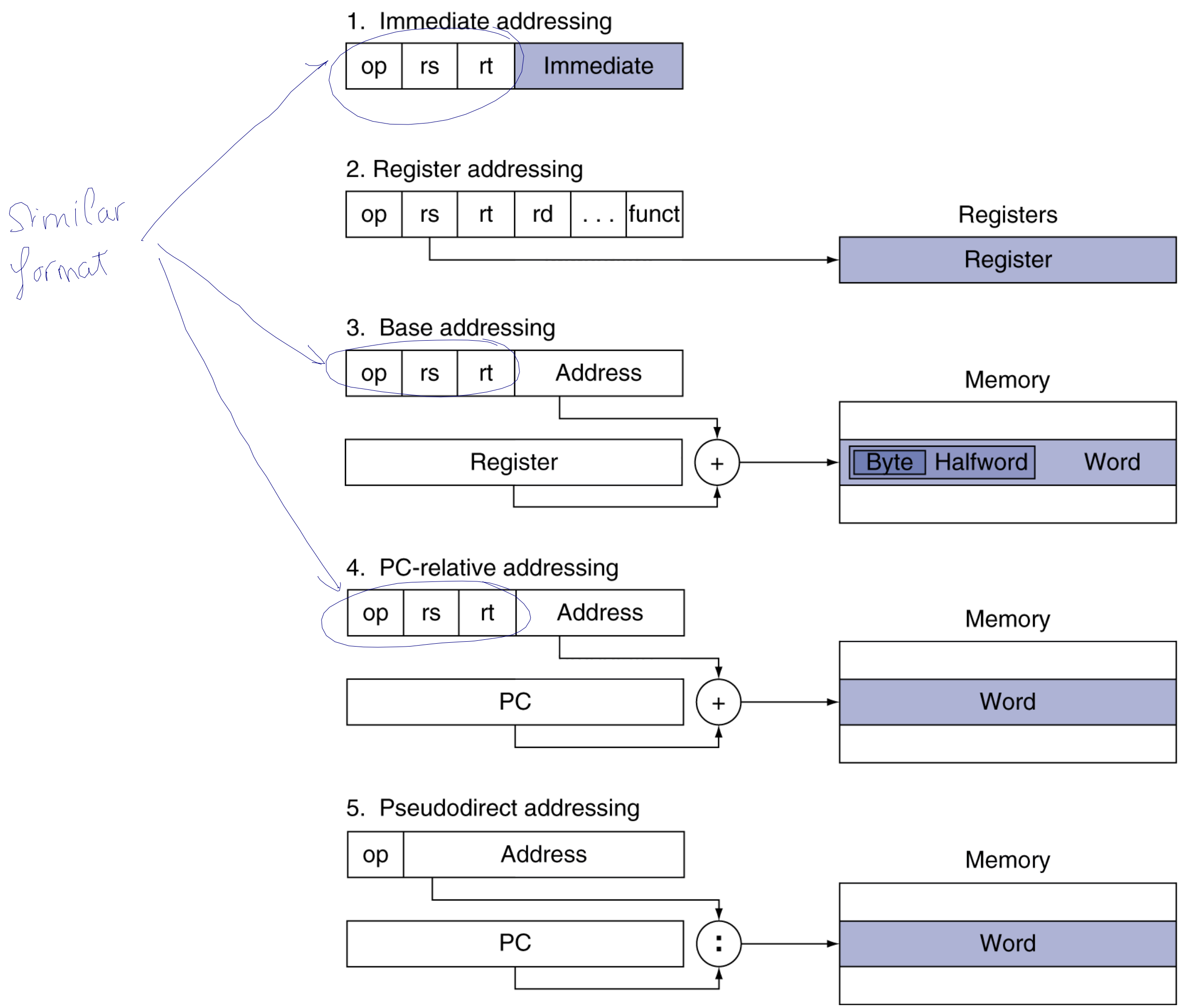
jumps to PC + 4 + L1 × 4.

jumps to PC + 4 + L2 × 4

jump to L1 × 4.

jal L1 ← $ra = PC + 4
jump to L1 × 4.

Addressing Mode Summary



Synchronization

use ll + sc to avoid data corruption in multiprocessors

- Two processors sharing an area of memory
  - P1 writes, then P2 reads
  - Data race if P1 and P2 don't synchronize
    - Result depends of order of accesses
- Hardware support required
  - Atomic read/write memory operation
  - No other access to the location allowed between the read and write
- Could be a single instruction
  - E.g., atomic swap of register ↔ memory
  - Or an atomic pair of instructions

Synchronization in MIPS

- "Load linked" record the value of memmory block at location offset(rs)
- User can execute the code in between
- "Store cond" write the content of rt to the location offset(rs) ONLY
* value of offset(rs) did not change (v/s recorded in load linked) => after update successful it updates rt = 1.
* if value of offset(rs) changed, it updates rt = 0 & stop update offset(rs).

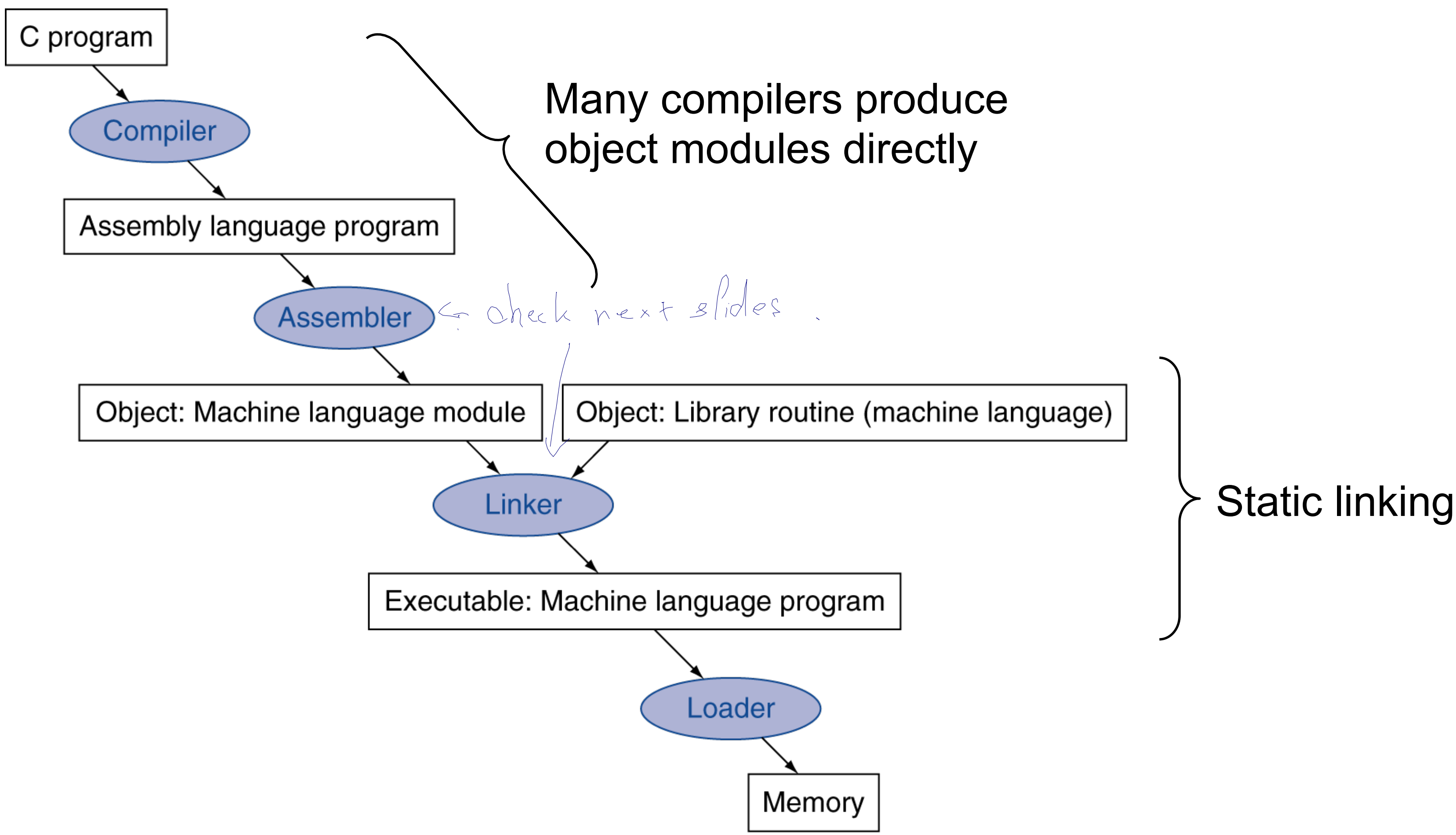
- Load linked: `ll rt, offset(rs)`
- Store conditional: `sc rt, offset(rs)`

Avoid data corruption when read/write at the same time.

- Succeeds if location not changed since the `ll`
  - Returns 1 in rt
- Fails if location is changed
  - Returns 0 in rt
- Example: atomic swap (to test/set lock variable)

```
try: add $t0,$zero,$s4 ;copy exchange value
     ll  $t1,0($s1)    ;load linked
     sc  $t0,0($s1)    ;store conditional
     beq $t0,$zero,try ;branch store fails
     add $s4,$zero,$t1 ;put load value in $s4
```

Translation and Startup



Assembler Pseudoinstructions

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

copy value Real MIPS commands.

not a real MIPS command, but can be understood & translated by the assembler.

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

  - $at (register 1): assembler temporary

Producing an Object Module

- Assembler (or compiler) translates program into machine instructions assembly code to binary obj.
- Provides information for building a complete program from the pieces
  - Header: described contents of object module
  - Text segment: translated instructions ← all commands here.
  - Static data segment: data allocated for the life of the program → global vars
  - Relocation info: for contents that depend on absolute location of loaded program ← which subroutines it needs
  - Symbol table: global definitions and external refs
  - Debug info: for associating with source code

Linking Object Modules

(Stitch all separate obj code into one.)

- Produces an executable image
  1. Merges segments
  2. Resolve labels (determine their addresses)
  3. Patch 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

Loading a Program

- Load from image file on disk into memory
  1. Read header to determine segment sizes
  2. Create virtual address space
  3. Copy text and initialized data into memory
     - Or set page table entries so they can be faulted in
  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 bloat caused by static linking of all (transitively) referenced libraries
  - Automatically picks up new library versions

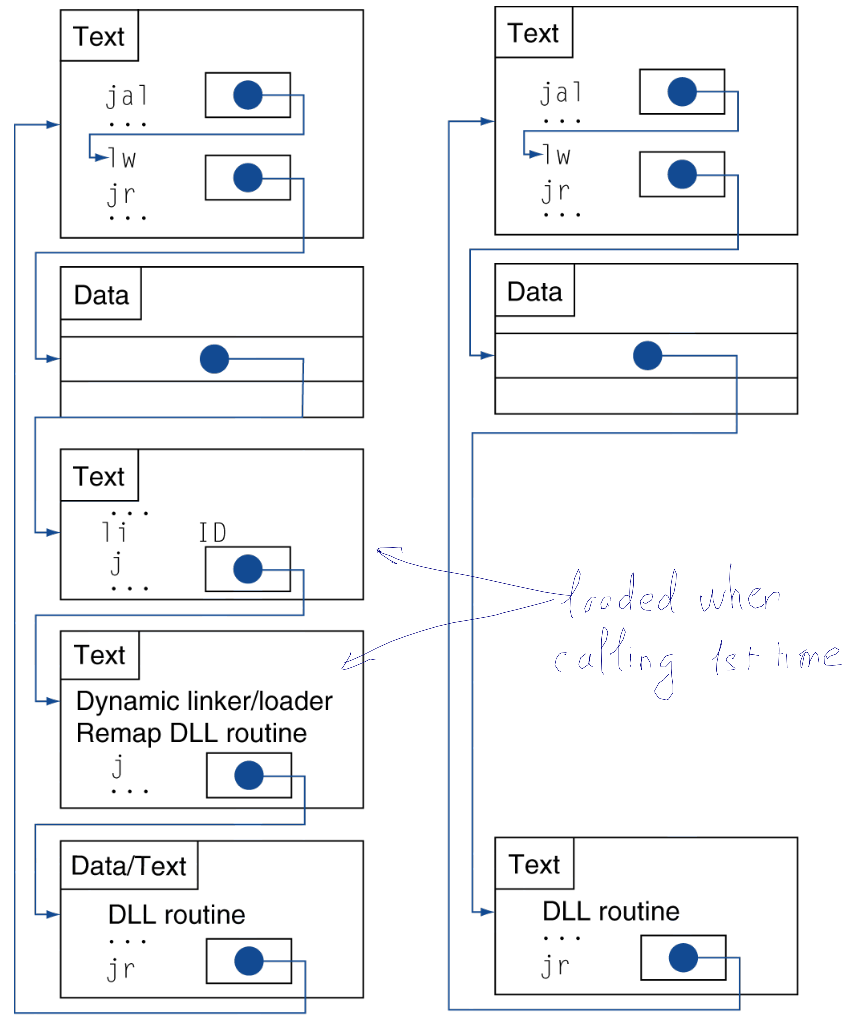
Lazy Linkage

Indirection table

Stub: Loads routine ID, Jump to linker/loader

Linker/loader code

Dynamically mapped code

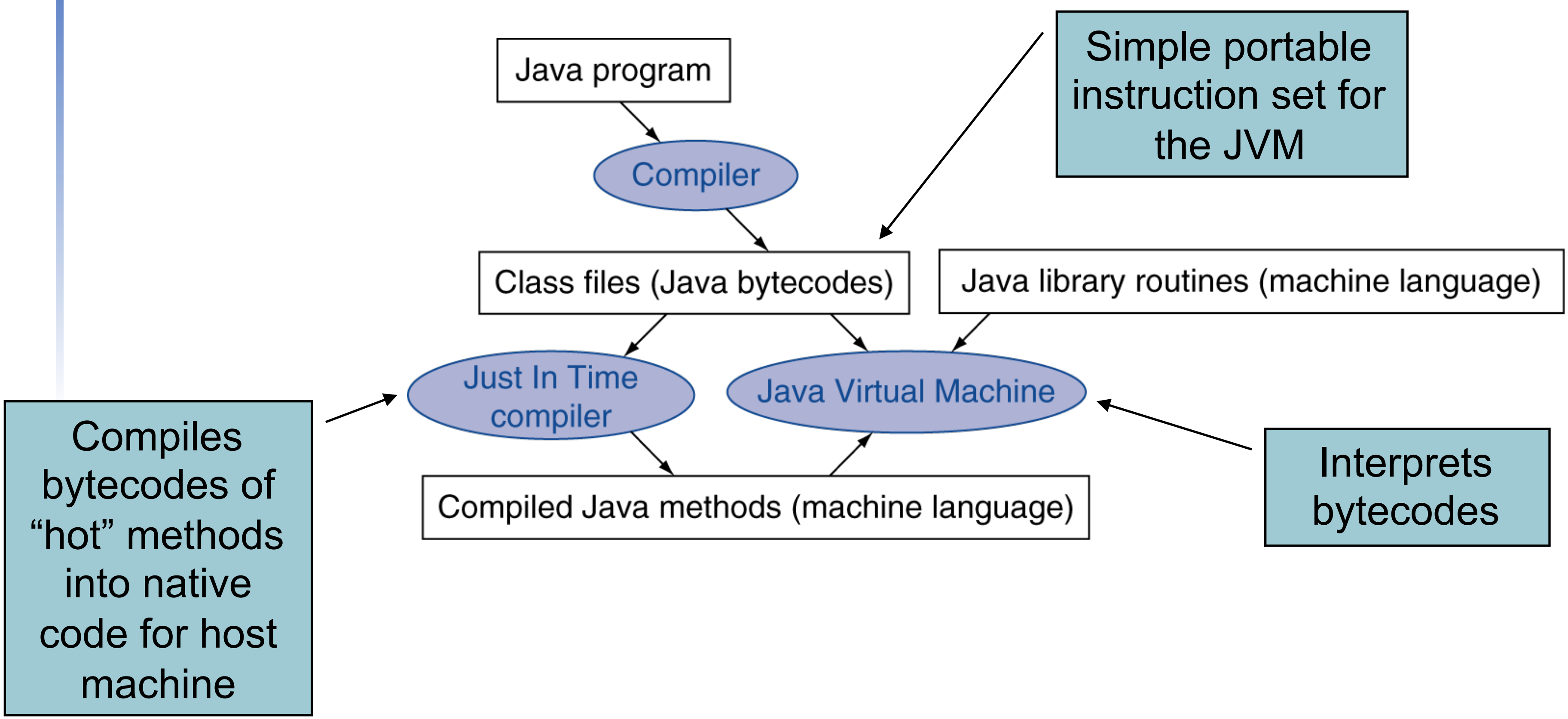


a. First call to DLL routine

b. Subsequent calls to DLL routine

Starting Java Applications

lesser steps than C compiler



C Sort Example

- Illustrates use of assembly instructions for a C bubble sort function
- Swap procedure (leaf)

$a0 $a1

```
void swap(int v[], int k)
{
  int temp;
  temp = v[k];
  v[k] = v[k+1];
  v[k+1] = temp;
}
```

$t0

  - v in $a0, k in $a1, temp in $t0

The Procedure Swap

```
swap: sll $t1, $a1, 2   # $t1 = k * 4
      add $t1, $a0, $t1 # $t1 = v+(k*4)
                        #   (address of v[k])
      lw $t0, 0($t1)    # $t0 (temp) = v[k]
      lw $t2, 4($t1)    # $t2 = v[k+1]
      sw $t2, 0($t1)    # v[k] = $t2 (v[k+1])
      sw $t0, 4($t1)    # v[k+1] = $t0 (temp)
      jr $ra            # return to calling routine
```

The Sort Procedure in C

- Non-leaf (calls swap) $a0 $a1

```
void sort (int v[], int n)
{        $s0  $s1
   int i, j;
   for (i = 0; i < n; i += 1) {
      for (j = i – 1;
           j >= 0 && v[j] > v[j + 1];
           j -= 1) {
         swap(v,j);
      }
   }
}
```

  - v in \$a0, ~~k~~ n in \$a1, i in \$s0, j in \$s1

The Procedure Body

	move	$s2, $a0	# save $a0 into $s2 // $s2 = $a0.	Move params
	move	$s3, $a1	# save $a1 into $s3 // $s3 = $a1	
	move	$s0, $zero	# i = 0 $t0=1 if i<n	Outer loop
for1tst:	slt	$t0, $s0, $s3	# ~~$t0 = 0 if $s0 ≥ $s3 (i ≥ n)~~	
	beq	$t0, $zero, exit1	# go to exit1 if $s0 ≥ $s3 (i ≥ n)	
	addi	$s1, $s0, –1	# j = i – 1	
for2tst:	slti	$t0, $s1, 0	# $t0 = 1 if $s1 < 0 (j < 0)	
	bne	$t0, $zero, exit2	# go to exit2 if $s1 < 0 (j < 0)	
	sll	$t1, $s1, 2	# $t1 = j * 4	Inner loop
	add	$t2, $s2, $t1	# $t2 = v + (j * 4)	
	lw	$t3, 0($t2)	# $t3 = v[j]	
	lw	$t4, 4($t2)	# $t4 = v[j + 1]	
	slt	$t0, $t4, $t3	# $t0 = 0 if $t4 ≥ $t3	
	beq	$t0, $zero, exit2	# go to exit2 if $t4 ≥ $t3	
	move	$a0, $s2	# 1st param of swap is v (old $a0)	Pass params & call
	move	$a1, $s1	# 2nd param of swap is j	
	jal	swap	# call swap procedure	
	addi	$s1, $s1, –1	# j –= 1	Inner loop
	j	for2tst	# jump to test of inner loop	
exit2:	addi	$s0, $s0, 1	# i += 1	Outer loop
	j	for1tst	# jump to test of outer loop	

The Full Procedure

```
sort:    addi $sp,$sp, -20       # make room on stack for 5 registers
         sw $ra, 16($sp)         # save $ra on stack
         sw $s3,12($sp)          # save $s3 on stack
         sw $s2, 8($sp)          # save $s2 on stack
         sw $s1, 4($sp)          # save $s1 on stack
         sw $s0, 0($sp)          # save $s0 on stack
         …                       # procedure body
         …
         exit1: lw $s0, 0($sp)   # restore $s0 from stack
         lw $s1, 4($sp)          # restore $s1 from stack
         lw $s2, 8($sp)          # restore $s2 from stack
         lw $s3,12($sp)          # restore $s3 from stack
         lw $ra,16($sp)          # restore $ra from stack
         addi $sp,$sp, 20        # restore stack pointer
         jr $ra                  # return to calling routine
```

Unit 5 Homework

- 1. Convert this MIPS instruction to base 10 and base2 machine code : `bne   $s1,  $s2,  L1`“
- 2. What are the information data contained in the object file ?
- 3. What are the different steps that the loader will do ?

Q1) * As L1 is not defined, assume it is the next PC from the executed instruction ⇒ imm16 = 0

MIPS instructions

```
bne   $s1, $s2, L1
L1: ...
```

Base 10.

5	17	18	0

Base 2.

000101	10001	10010	0000 0000 0000 0000

→ 00010110 00110010 00000000 00000000.

* If L1 is not found any where and this is the only instruction, the PC return to the current instruction.

⇒ imm16 = −1

⇒ Base 10 : 5 | 17 | 18 | −1

Base 2 : | 000101 | 10001 | 10010 | 1111 1111 1111 1111 |

→ 00010110 00110010 11111111 11111111

Q2) A general object file can contain these:

- Header : describes content of the object file
- Code segment (text segment): translated instructions
- Data segment : initialized static variables allocated for the life of the program
- Read-only segment : initialized static constants
- BSS segment : uninitialized static data
- External definitions & references for linking
- Relocation info: for content that depends on the absolute location of the loaded program
- Dynamic linking info
- Debugging info : for associating with source code.

Q3) Loader loads from image file on disk into memory:

1. Read header to determine segment sizes
2. Create virtual address space
3. Copy text & initialized data into memory or set page table entries so they can be faulted in.
4. Setup arguments on stack
5. Initialize registers (including `$sp`, `$fp`, `$gp`)
6. Jump to start up routine
   - Copy arguments to `$a0`, ... and calls main.
   - When main returns, do exit syscall.