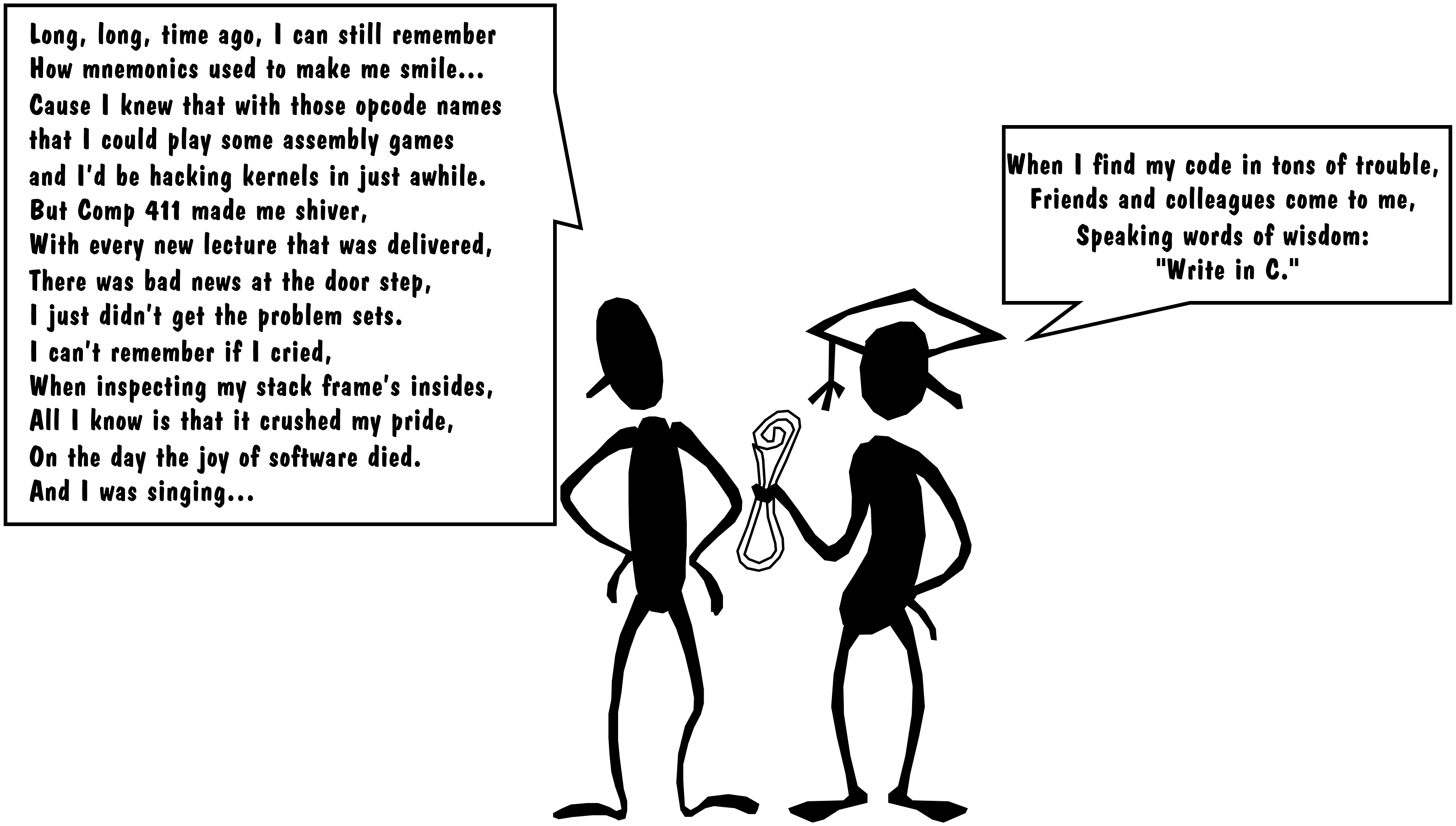
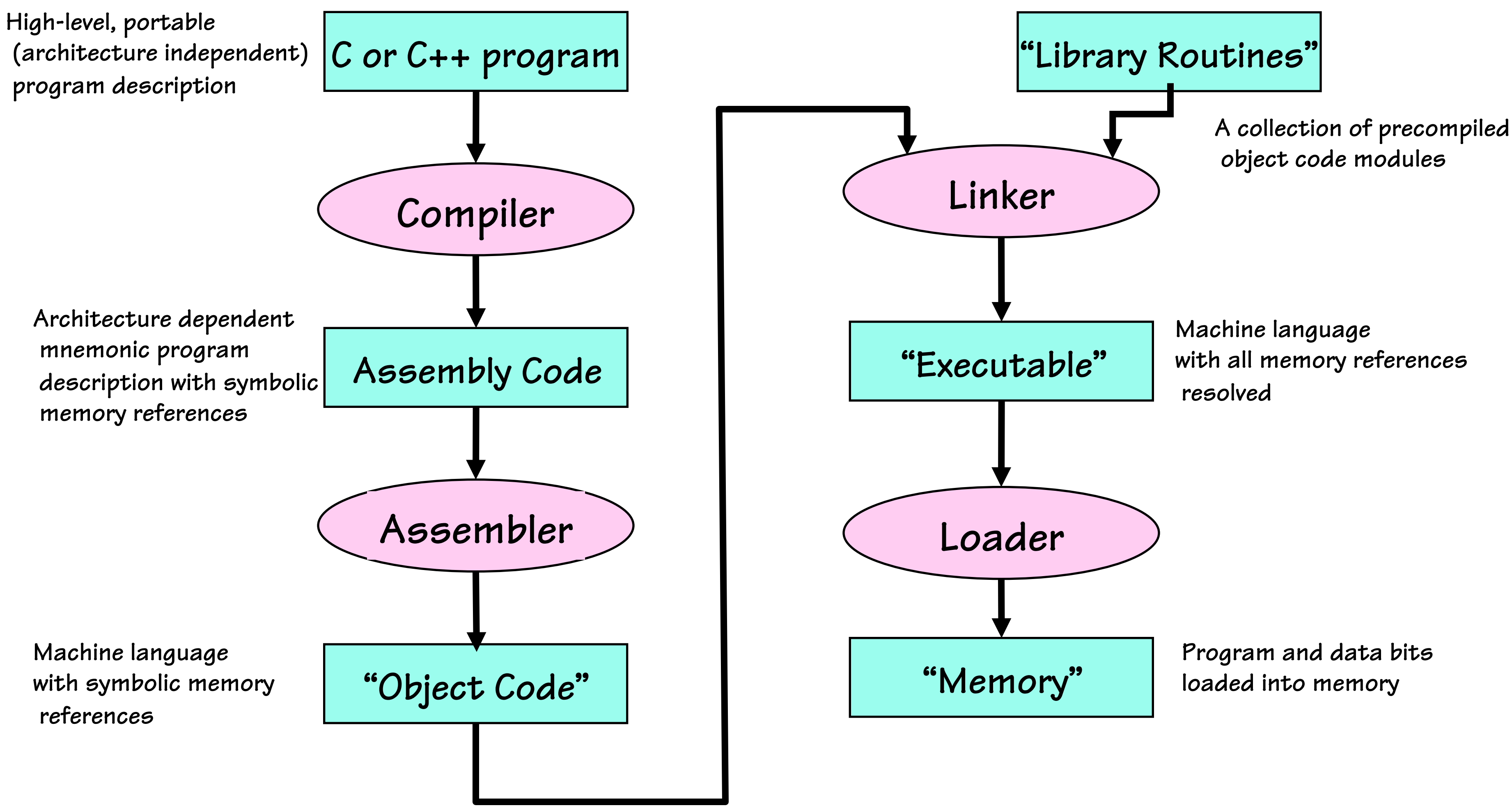
Assemblers and Compilers



Study sections 2.10-2.15

Path from Programs to Bits

- Traditional Compilation



How an Assembler Works

Three major components of assembly

1) Allocating and initialing data storage

2) Conversion of mnemonics to binary instructions

3) Resolving addresses

```
.data
array:   .space 40
total:   .word 0
```

```
.text
.globl main
main:    la      $t1,array
         move    $t2,$0
         move    $t3,$0
         beq     $0,$0,test
loop:    sll     $t0,$t3,2
         add     $t0,$t1,$t0
         sw      $t3,($t0)
         add     $t2,$t2,$t3
         addi    $t3,$t3,1
test:    slti    $t0,$t3,10
         bne     $t0,$0,loop
         sw      $t2,total
         jr      $ra
```

→

```
lui    $9, arrayhi
ori    $9,$9,arraylo
```

→

```
0x3c09????
0x3529????
```

Resolving Addresses- 1st Pass

- "Old-style" 2-pass assembler approach

Pass 1

Segment offset	Code	Instruction
0 4	0x3c090000 0x35290000	la $t1,array
8 12	0x00005021 0x00005821	move $t2,$ move $t3,$0
16	0x10000000	beq $0,$0,test
20	0x000b4080	loop: sll $t0,$t3,2

24
28
32
36
	0x01284020
	0xad0b0000
	0x014b5020
	0x216b0001
		add $t0,$t1,$t0
		sw $t0,($t0)
		add $t0,$t1,$t0
		addi $t3,$t3,1

40	0x2968000a	test: slti $t0,$t3,10
44	0x15000000	bne $t0,$0,loop
48 52	0x3c010000 0xac2a0000	sw $t2,total
56	0x03e00008	j $ra

- In the first pass, data and instructions are encoded and assigned offsets within their segment, while the symbol table is constructed.
- Unresolved address references are set to 0

Symbol table after Pass 1

Symbol	Segment	Location pointer offset
array	data	0
total	data	40
main	text	0
loop	text	20
test	text	40

Resolving Addresses in 2 Passes

- "Old-style" 2-pass assembler approach

Pass 2

Segment offset	Code	Instruction
0 4	0x3c091001 0x35290000	la $t1,array
8 12	0x00005021 0x00005821	move $t2,$ move $t3,$0
16	0x10000006	beq $0,$0,test
20	0x000b4080	loop: sll $t0,$t3,2

24
28
32
36
	0x01284020
	0xad0b0000
	0x014b5020
	0x216b0001
		add $t0,$t1,$t0
		sw $t0,($t0)
		add $t0,$t1,$t0
		addi $t3,$t3,1

40	0x2968000a	test: slti $t0,$t3,10
44	0x1500fffa	bne $t0,$0,loop
48 52	0x3c011001 0xac2a0028	sw $t2,total
56	0x03e00008	j $ra

– In the second pass, the appropriate fields of those instructions that reference memory are filled in with the correct values if possible.

Symbol table after Pass 1

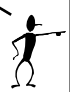
Symbol	Segment	Location pointer offset
array	data	0
total	data	40
main	text	0
loop	text	20
test	text	40

Modern Way – 1-Pass Assemblers

Modern assemblers keep more information in their symbol table which allows them to resolve addresses in a single pass.

- Known addresses (backward references) are immediately resolved.
- Unknown addresses (forward references) are "back-filled" once they are resolved.

State of the symbol table after the instruction sw $t0, ($t0) is assembled



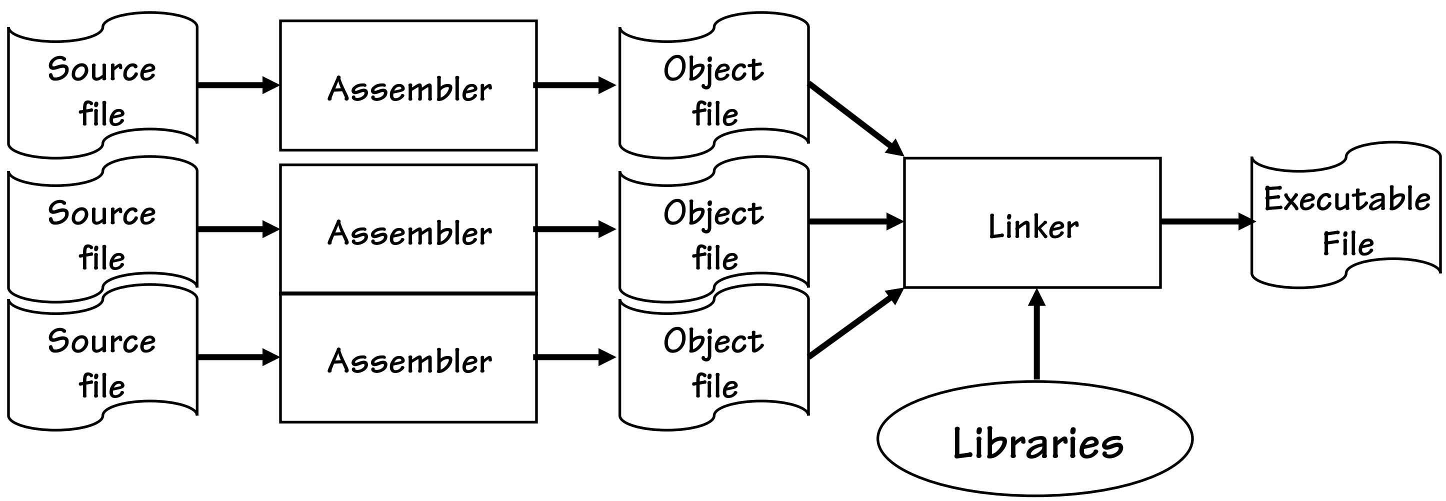
SYMBOL	SEGMENT	Location pointer offset	Resolved?	Reference list
array	data	0	y	null
total	data	40	y	null
main	text	0	y	null
loop	text	20	y	null
test	text	?	n	16

The Role of a Linker

Some aspects of address resolution cannot be handled by the assembler alone.

1) References to data or routines in other object modules
2) The layout of all segments (.text, .data) in memory
3) Support for REUSABLE code modules
4) Support for RELOCATABLE code modules

This final step of resolution is the job of a LINKER



Static and Dynamic Libraries

- LIBRARIES are commonly used routines stored as a concatenation of "Object files". A global symbol table is maintained for the entire library with entry points for each routine.
- When a routine in a LIBRARY is referenced by an assembly module, the routine's address is resolved by the LINKER, and the appropriate code is added to the executable. This sort of linking is called STATIC linking.
- Many programs use common libraries. It is wasteful of both memory and disk space to include the same code in multiple executables. The modern alternative to STATIC linking is to allow the LOADER and THE PROGRAM ITSELF to resolve the addresses of libraries routines. This form of lining is called DYNAMIC linking (e.x. .dll).

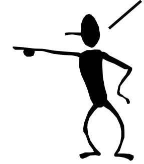
Dynamically Linked Libraries

- C call to library function:

```
printf("sqr[%d] = %d\n", x, y);
```

- Assembly code

```
addi    $a0,$0,1
la      $a1,ctrlstring
lw      $a2,x
lw      $a3,y
call    fprintf
```

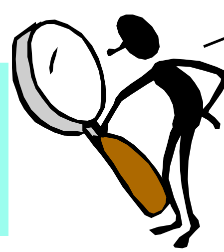


Yet another pseudoinstruction

How does dynamic linking work?

- Maps to:

```
addi    $a0,$0,1
lui     $a1,ctrlstringHi
ori     $a1,ctrlstringLo
lui     $at,globaldata
lw      $a2,x($at)
lw      $a3,y($at)
lui     $at,fprintfHi
lw      $at,fprintfLo($at)
jalr    $at,$31
```



Why are we loading the function's address into a register first, and then calling it?

Dynamically Linked Libraries

- Lazy address resolution:

```
sysload:   addui $sp,$sp,16
           .
           .
          # check if stdio module
          # is loaded, if not load it
          .
          .
          # backpatch jump table
          la      $t1,stdio
          la      $t0,$dfopen
          sw      $t0,($t1)
          la      $t0,$dfclose
          sw      $t0,4($t1)
          la      $t0,$dfputc
          sw      $t0,8($t1)
          la      $t0,$dfgetc
          sw      $t0,12($t1)
          la      $t0,$dfprintf
          sw      $t0,16($t1)
```

Because, the entry points to dynamic library routines are stored in a TABLE. And the contents of this table are loaded on an “as needed” basis!

- Before any call is made to a procedure in “stdio.dll”

```
.globl stdio:
stdio:
fopen:    .word sysload
fclose:   .word sysload
fgetc:    .word sysload
fputc:    .word sysload
fprintf:  .word sysload
```

- After first call is made to any procedure in “stdio.dll”

```
.globl stdio:
stdio:
fopen:    dfopen
fclose:   dclose
fgetc:    dfgetc
fputc:    dfputc
fprintf:  dprintf
```

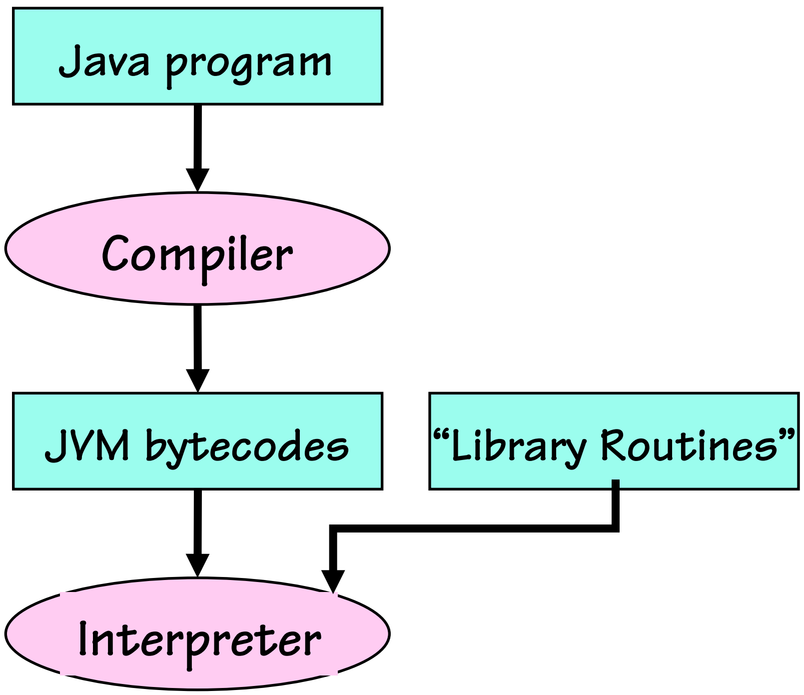
Modern Languages

- Intermediate “object code language”

High-level, portable (architecture independent) program description

PORTABLE mnemonic program description with symbolic memory references

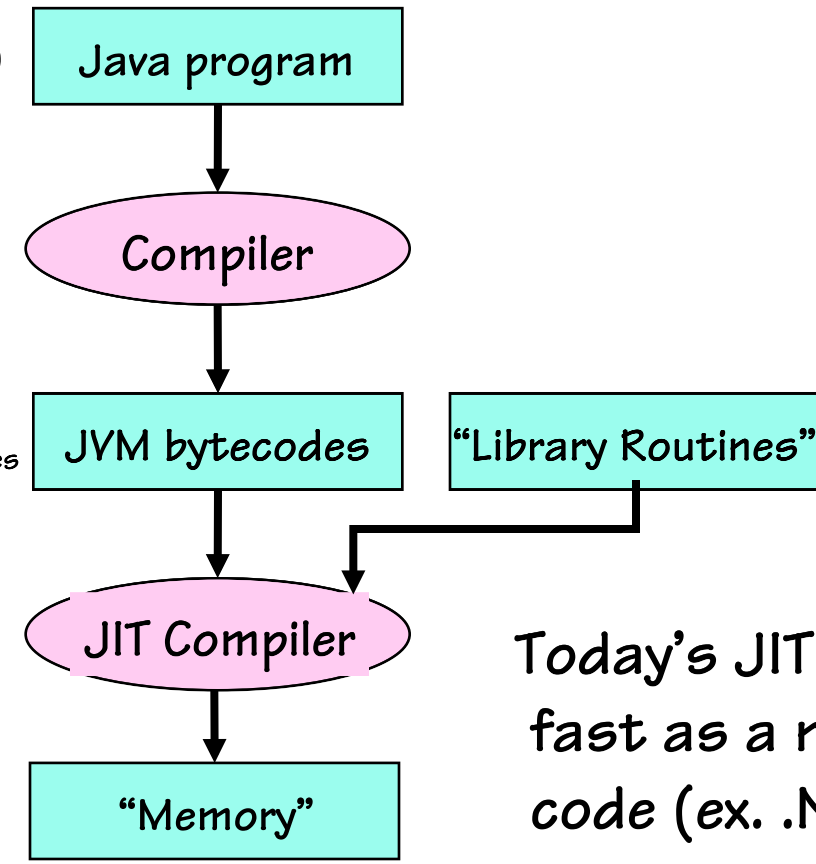
An application that EMULATES a virtual machine. Can be written for any Instruction Set Architecture. In the end, machine language instructions must be executed for each JVM bytecode



Modern Languages

- Intermediate "object code language"

High-level, portable (architecture independent) program description



PORTABLE mnemonic program description with symbolic memory references

While interpreting on the first pass it keeps a copy of the machine language instructions used. Future references access machine language code, avoiding further interpretation

Today's JITs are nearly as fast as a native compiled code (ex. .NET).

Assembly? Really?

- In the early days compilers were dumb
  - literal line-by-line generation of assembly code of "C" source
  - This was efficient in terms of S/W development time
    - C is portable, ISA independent, write once– run anywhere
    - C is easier to read and understand
    - Details of stack allocation and memory management are hidden
  - However, a savvy programmer could nearly always generate code that would execute faster
- Enter the modern era of Compilers
  - Focused on optimized code-generation
  - Captured the common tricks that low-level programmers used
  - Meticulous bookkeeping (i.e. will I ever use this variable again?)
  - It is hard for even the best hacker to improve on code generated by good optimizing compilers

Example Compiler Optimizations

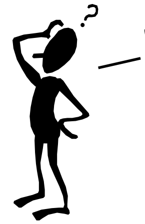
- Example "C" Code:

```
int array[10];
int total;

int main( ) {
    int i;

    total = 0;
    for (i = 0; i < 10; i++) {
        array[i] = i;
        total = total + i;
    }
}
```

Unoptimized Assembly Output



Why does turning on debugging generate the worse code?

Ans: Because the complier reverts back to line-by-line translation.

- With debug flags set:

```
.globl main
.text
main:
        addiu $sp,$sp,-8          # allocates space for ra and i
        sw $0,total               # total = 0
        sw $0,0($sp)              # i = 0
        lw $8,0($sp)              # copy i to $t0
        b  L.3                    # goto test
L.2:                              # for(...) {
        sll $24,$8,2              #  make i a word offset
        sw $8,array($24)          #  array[i] = i
        lw $24,total              #  total = total + i
        addu $24,$24,$8
        sw $24,total
        addi $8,$8,1              #  i = i + 1
L.3:
        sw $8,0($sp)              #  update i in memory
        slti $1,$8,10             #  (i < 10)?
        bne $1,$0,L.2             #} if TRUE loop
        addiu $sp,$sp,8
        jr $31
```

Register Allocation

- Assign local variable “i” to a register

Two instructions outside the loop are replaced with one

```
.globl main
.text
main:
        addiu $sp,$sp,-4        #allocates space for ra
        sw $0,total             #total = 0
        move $8,$0              #i = 0
        b  L.3                  #goto test
L.2:                            #for(...) {
        sll $24,$8,2            #  make i a word offset
        sw $8,array($24)        #  array[i] = i
        lw $24,total            #  total = total + i
        addu $24,$24,$8
        sw $24,total
        addi $8,$8,1            #  i = i + 1
L.3:
        slti $1,$8,10           #  (i < 10)?
        bne $1,$0,L.2           #} if TRUE loop
        addiu $sp,$sp,4
        jr $31
```

Loop-Invariant Code Motion

- Temporarily allocate temp registers to hold global values to avoid loads inside the loop, yet mirroring changes

We've added an instruction here outside of the loop

and eliminated an lw inside of loop

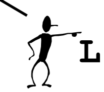
```
.globl main
.text
main:
        addiu $sp,$sp,-4          #allocates space for ra
        sw $0,total               #total = 0
        move $9,$0                #temp for total
        move $8,$0                #i = 0
        b  L.3                    #goto test
L.2:                              #for(...) {
        sll $24,$8,2              #  make i a word offset
        sw $8,array($24)          #  array[i] = i
        addu $9,$9,$8
        sw $9,total
        addi $8,$8,1              #  i = i + 1
L.3:
        slti $1,$8,10             #  (i < 10)?
        bne $1,$0,L.2             #} if TRUE loop
        addiu $sp,$sp,4
        jr $31
```

Remove Unnecessary Tests

- Since "i" is initially set to "0", we already know it is less than "10", so why bother testing it the first time?

```
.globl main
.text
main:
        addiu $sp,$sp,-4          #allocates space for ra
        sw $0,total               #total = 0
        move $9,$0                #temp for total
        move $8,$0                #i = 0
L.2:                              #for(...) {
        sll $24,$8,2              #  make i a word offset
        sw $8,array($24)          #  array[i] = i
        addu $9,$9,$8
        sw  $9,total
        addi $8,$8,1              #  i = i + 1
        slti $1,$8,10             #  loads const 10
        bne $1,$0,L.2             #} loops while i < 10
        addiu $sp,$sp,4
        jr $31
```

Eliminated a branch here and the label it referenced



Remove Unnecessary Stores

- All we care about it the value of total after the loop finishes, so there is no need to update it on each pass

```
.globl main
.text
main:
        addiu $sp,$sp,-4          #allocates space for ra and i
        sw $0,total               #total = 0
        move $9,$0                #temp for total
        move $8,$0                #i = 0
L.2:
        sll $24,$8,2              #for(...) {
        sw $8,array($24)          #   array[i] = i
        addu $9,$9,$8
        addi $8,$8,1              #   i = i + 1
        slti $1,$8,10             #   loads const 10
        bne $1,$0,L.2             #} loops while i < 10
        sw  $9,total
        addiu $sp,$sp,4
        jr $31
```

Moved this instruction outside the loop

Unrolling Loops

- By examining the function we can see it is always executed 10 times. Thus, we can make 2, 5, or 10 copies of the inner loop reduce the branching overhead.

```
.globl main
.text
main:
     addiu $sp,$sp,-4        #allocates space for ra and i
     sw $0,total             #total = 0
     move $9,$0              #temp for total
     move $8,$0              #i = 0
L.2:
     sll $24,$8,2            #for(...) {
     sw $8,array($24)        #   array[i] = i
     addu $9,$9,$8
     addi $8,$8,1            #   i = i + 1
     sll $24,$8,2            #
     sw $8,array($24)        #   array[i] = i
     addu $9,$9,$8
     addi $8,$8,1            #   i = i + 1
     slti $24,$8,10          #   loads const 10
     bne $24,$0,L.2          #} loops while i < 10
     sw  $9,total
     addiu $sp,$sp,4
     jr $31
```

Added a second copy of these four lines.

Next Time

- We go deeper into the rabbit hole…

- Quiz on Friday
  - Multiple Choice
  - Open book/open notes
  - No computers or calculators