PA5

- Kanga to MIPS
- Due next Friday

- Use the provided code.
- Fill in the visit methods for the Kanga instructions.
Today

- MIPS
- SPIM
- Instruction scheduling
MIPS

- RISC architecture
- developed by MIPS Computer Systems
  - founded by John Hennessy
  - R2000 – 1985
  - bought by SGI 1992

- Used in SGI computers
- Big influence on DEC Alpha, PowerPC
- Used in PS2, PSP
Text vs Data

• Assembly programs have two sections:
  • **Text**
    • Instructions go here
    • Larger of the two sections
    • Contains the beginning of the program
  • **Data**
    • Where the variables are declared
    • A limited number of data types in ASM
Assembler directives

• Other Directives
  • .text – Indicates that following items are stored in the user text segment
  • .data – Indicates that following data items are stored in the data segment
  • .globl sym – Declare that symbol sym is global and can be referenced from other files
Declaring Data

• Done in the .data section

<name>: <type> <value>
str: .asciiz "Hello World\0"
str2: .ascii "Hello"
number: .byte 10
bigNum: .word 10000
Assembler directives

• Data Types
  • .word, .half, .byte – 32/16/8 bit integer
  • .ascii, .asciiz – string (asciiz is null terminated)
    • Strings are enclosed in double-quotas ("")
    • Special characters in strings follow the C convention
      • newline(\n), tab(\t), quote(\")
  • .double, .float – floating point
Mnemonics

• add, sub, addu, subu
  • R format: opcode followed by rd, rs, rt
  • Example: addu $s4, $t1, $s3  ; $s4 = $t1 + $s3
• addi, addui
  • I format: opcode is followed by rd, rs, number
  • Example: addui $s4, $t1, -4   ; $s4 = $t1 + (-4)

• Comments
  • Remember the adjectives – u=ignore overflow, i=immediate
  • Order of arguments is usually destination first
  • There is no subi, to save opcodes, addi with negative number
  • Normally you should use the u version
Instructions

addu $s4, $t1, $s3 ; $s4 = $t1 + $s3

• Each instruction is a single line
• Opcode field is basic tells whether this is machine instruction or pseudoinstruction
• Whitespace separates location, opcode, and operand fields
• Commas separate operands
## Register conventions

<table>
<thead>
<tr>
<th>Name</th>
<th>Number</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>0</td>
<td>constant 0</td>
</tr>
<tr>
<td>$at</td>
<td>1</td>
<td>reserved for assembler</td>
</tr>
<tr>
<td>$v0-1</td>
<td>2-3</td>
<td>expression evaluation and results of a function</td>
</tr>
<tr>
<td>$a0-3</td>
<td>4-7</td>
<td>argument 1..4</td>
</tr>
<tr>
<td>$t0-9</td>
<td>8-15, 24-25</td>
<td>temporary (not preserved across call)</td>
</tr>
<tr>
<td>$s0-7</td>
<td>16</td>
<td>saved temporary (preserved across call)</td>
</tr>
<tr>
<td>$k0-1</td>
<td>26-27</td>
<td>reserved for OS kernel</td>
</tr>
<tr>
<td>$gp</td>
<td>28</td>
<td>pointer to global area</td>
</tr>
<tr>
<td>$sp</td>
<td>29</td>
<td>stack pointer</td>
</tr>
<tr>
<td>$fp</td>
<td>30</td>
<td>frame pointer</td>
</tr>
<tr>
<td>$ra</td>
<td>31</td>
<td>return address (used by function call)</td>
</tr>
<tr>
<td>$f0-15</td>
<td>f0-15</td>
<td>floating point registers (may contain only floats)</td>
</tr>
</tbody>
</table>
add $r0, $r2, $r3  # r0 = r2 + r3
addi $t0, $t1, 15 # t0 = t1 + 15 (from instruction)
sub $r0, $r0, $r3  # r0 = r0 – r3
mul $t0, $t1, $t2  # t0 = t1 * t2
div $t0, $t1, $t2  # t0 = t1 / t2
la $a0, str  # load address of str into $a0
li $t1, 3   # t1 = 3 (from instruction)
Multiplication/Division

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>add $t1, $t2, $t3 ; $t1 = $t2 + $t3</td>
</tr>
<tr>
<td>add immediate</td>
<td>addi $t1, $t2, 50 ; $t1 = $t2 + 50</td>
</tr>
<tr>
<td>subtract</td>
<td>sub $t1, $t2, $t3 ; $t1 = $t2 - $t3</td>
</tr>
<tr>
<td>multiply</td>
<td>mult $t2, $t3 ; (hi,lo) = $t2 * $t3</td>
</tr>
<tr>
<td>divide</td>
<td>div $t2, $t3 ; (hi,lo) = $t2 / $t3</td>
</tr>
<tr>
<td>move from lo</td>
<td>mfhi $t1 ; $t1 = hi</td>
</tr>
<tr>
<td>move from hi</td>
<td>mflo $t1 ; $t1 = lo</td>
</tr>
</tbody>
</table>

Because multiplication/division can result in a larger number, there are two special registers named lo and hi:

- \( \text{li \ $t0, 5} \)
- \( \text{li \ $t1, 2} \)
- \( \text{mult \ $t0, $t1} \)
- \( \text{mflo \ $t3} \)
Load/store

lw   $t0, <n>($t1)  # load word $t1+n into $t0
sw   $t0, <n>($t1)  # store word $t0 into $t1+n

• Spills:
  • Load/store relative to frame pointer $fp
Control Structures

j <addr>  #update PC to be addr
beq $t0, $t1, <addr>  #if t0==t1, go to addr
beqz $t0, <addr>  #if t0 == 0, go to addr
bne $t0, $t1, <addr>  #if t0 != t1, go to addr
blt $t0, $t1, <addr>  #if t0 < t1, go to addr
Labels

• Anywhere in our code, we can put a label
• Labels are names in the code
• Keep us from having to know the exact memory address!
• Labels are followed by a colon

myLabel:
Calling conventions

- Copy first four arguments into a0 to a3
- Put remaining arguments onto stack
- Call with `jal` (“jump and link”)

**Caller saves registers**
- Before call: save t0-9 to stack if live across the call
- After call: restore caller saves registers

**Callee saves registers**
- On entry: save s0-7 if used in procedure body (can delay saves until actual use)
- On return: restore callee saves registers
Call/return

jal <addr>  # call (jump and link)
jalr $t0     # call (jump and link)

Prologue:
    sw $fp, -8($sp)
    move $fp, $sp
    subu $sp, $sp, 20  ; frame size = 4* (#spills+2)
    sw $ra, -4($fp)

Epilogue:
    lw $ra, -4($fp)
    lw $fp, 12($sp)
    addu $sp, $sp, 20
    j $ra
SPIM

• A simulator that runs programs for the MIPS R2000/R3000 RISC computers
• Developed by James R. Larus, University of Wisconsin

• Reads and executes MIPS assembly language file immediately
• Works as a debugger
• Provides some OS like services

• Download and follow instructions at:
  • http://pages.cs.wisc.edu/~larus/spim.html
System calls

<table>
<thead>
<tr>
<th>Service</th>
<th>Call code</th>
<th>Arguments</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>print_int</td>
<td>1</td>
<td>$a0 = integer</td>
<td></td>
</tr>
<tr>
<td>print_float</td>
<td>2</td>
<td>$f12 = float</td>
<td></td>
</tr>
<tr>
<td>print_double</td>
<td>3</td>
<td>$f12 = double</td>
<td></td>
</tr>
<tr>
<td>print_string</td>
<td>4</td>
<td>$a0 = string</td>
<td></td>
</tr>
<tr>
<td>read_int</td>
<td>5</td>
<td></td>
<td>integer in $v0</td>
</tr>
<tr>
<td>read_float</td>
<td>6</td>
<td></td>
<td>float in $f0</td>
</tr>
<tr>
<td>read_double</td>
<td>7</td>
<td></td>
<td>double in $f0</td>
</tr>
<tr>
<td>read_string</td>
<td>8</td>
<td>$a0 = buffer, $a1 = length</td>
<td></td>
</tr>
<tr>
<td>sbrk</td>
<td>9</td>
<td>$a0 = amount</td>
<td>address in $v0</td>
</tr>
<tr>
<td>exit</td>
<td>10</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Load code into $v0 and arguments into registers $a0…$a3. Perform **syscall**
Hello World

.text  # instruction segment starts here
.globl __start  # tell where to start
__start:  # execution starts here
    la $a0, str  # load the address of str into $a0
    li $v0, 4  # syscall 4 = print_string
    syscall  # perform syscall
    li $v0, 10  # syscall 10 = exit
    syscall  # perform syscall
.data  # data segment starts here
    str: .asciiz "hello world\n"  # define a null terminated string called str
.text
.globl __start
__start:
    la $a0, prompt
    li $v0, 4
    syscall
    li $v0, 5
    syscall

    mul $t0, $v0, 9
    div $t0, $t0, 5
    add $t0, $t0, 32

    la $a0, ans1
    li $v0, 4
    syscall

    move $a0, $t0
    li $v0, 1
    syscall
    la $a0, endl
    li $v0, 4
    syscall
    li $v0, 10
    syscall

.data
.globl prompt
prompt: .asciiz "Enter temperature (Celsius): 
.globl ans1
ans1: .asciiz "The temperature in Fahrenheit is: 
.globl endl
endl: .asciiz "\n"
Count to 10

.text
.globl __start

__start:
    li $t0, 0  # $t0 will be our counter, starting at 0

loop:       # here is our label to jump to later
    add $t0, 1  # $t0 = $t0 + 1
    move $a0, $t0  # $a0 = $t0, necessary because $v0 works with $a0
    li $v0, 1  # what kind of syscall? Print int! Go back & look at chart
    syscall

    la $a0, endl  # print a return (\n)
    li $v0, 4
    syscall

    bne $t0, 10, loop  # if $t0 != 10, then branch to the loop label

li $v0, 4  # print "Thank you"
la $a0, str
syscall

.data       # data section
.endl:    .asciiz "\n"
.str:        .asciiz "Thank you!\n"
Reading in Strings

• First, need to open up space for the string
  
  # open up 64 bytes
  
  mySpace: .space64

• String goes into $a0, and length into $a1
  
  la $a0, mySpace
  li $a1, 64

• Do system call
  
  li $v0, 8

• Now, mySpace is an array of characters!
.text
.globl   __start
__start:
    la      $a0, gimme       # print gimme a cookie
    li      $v0, 4
    syscall
    li      $v0, 8       # read string $a0 = buffer, $a1 = 1
    la      $a0, mySpace   # read into mySpace
    li      $a1, 64      # read max of 64
    syscall
    li      $t0, 0       # this is our counter
    li      $t5, 6      # number of letter in cookie
loop:
    lb      $t2, mySpace($t0)   # copy char of mySpace into $t2
    lb      $t3, cookie($t0)# copy char of cookie into $t3
    bne    $t2, $t3, __start   # if not equal, jump to start
    addi   $t0, 1       # $t0 = $t0 + 1
    bne    $t0, $t5, loop    # if $t0 isn't 6, jump back to loop
end:
    la      $a0, rock       # print you rock!
    li      $v0, 4
    syscall
.data
    gimme:   .asciiz "Gimme a cookie: 
    rock:     .asciiz "You rock!\n"
    mySpace:  .space    64
    cookie:   .asciiz "cookie"