EECS 322
The SPIM simulator

Instructor: Francis G. Wolff  wolff@eecs.cwru.edu  Case Western Reserve University
This presentation uses powerpoint animation: please viewshow
### MIPS instructions

<table>
<thead>
<tr>
<th>Category</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ALU</strong></td>
<td>alu $rd,$rs,$rt</td>
<td>$rd = $rs &lt;alu&gt; $rt</td>
</tr>
<tr>
<td><strong>JR</strong></td>
<td>jr $rs</td>
<td>$pc = $rs</td>
</tr>
<tr>
<td><strong>ALUi</strong></td>
<td>alui $rd,$rs,value16</td>
<td>$rd = $rs &lt;alu&gt; $δ^{32}(value16)</td>
</tr>
<tr>
<td><strong>Data Transfer</strong></td>
<td>lw $rt,offset16($rs)</td>
<td>$rt = Mem[$rs + $δ^{32}(offset16)]</td>
</tr>
<tr>
<td></td>
<td>sw $rt,offset16($rs)</td>
<td>Mem[$rs + $δ^{32}(offset16)] = $rt</td>
</tr>
<tr>
<td><strong>Branch</strong></td>
<td>beq $rs,$rt,offset16</td>
<td>$pc = ($rt == $rs)? ($pc+4+$δ^{32}(offset16)&lt;&lt;2)):($pc+4);</td>
</tr>
<tr>
<td><strong>Jump</strong></td>
<td>j address</td>
<td>$pc = ($pc &amp; 0xFC000000)</td>
</tr>
<tr>
<td><strong>Jump&amp;Link</strong></td>
<td>jal address</td>
<td>$ra = $pc+4; $pc = ($pc &amp; 0xFC000000)</td>
</tr>
</tbody>
</table>
# MIPS fixed sized instruction formats

## R - Format

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>func</th>
</tr>
</thead>
</table>

### ALU

- `alu $rd,$rs,$rt`

### JR

- `jr $rs`

## I - Format

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>value or offset</th>
</tr>
</thead>
</table>

### ALUI

- `alui $rt,$rs,value`

### Data Transfer

- `lw $rt,offset($rs)`
- `sw $rt,offset($rs)`

### Branch

- `beq $rs,$rt,offset`

## J - Format

<table>
<thead>
<tr>
<th>op</th>
<th>absolute address</th>
</tr>
</thead>
</table>

### Jump

- `j address`

### Jump & Link

- `jal address`
Assembling MIPS Instructions

![Diagram showing MIPS instruction assembly]

- **alu $rd, $rs, $rt**
  - 0x00400020
  - 000000:00000:11111:10111:00000:100001
  - addu $23, $0, $31

- **alui $rt, $rs, value**
  - 0x00400024
  - 001000:00000:10001:000000000000000101
  - addi $17, $0, 5
MIPS instruction formats

Arithmetic
- addi $rt, $rs, value
- add $rd,$rs,$rt
- jr $rs

Data Transfer
- lw $rt,offset($rs)
- sw $rt,offset($rs)

Conditional branch
- beq $rs,$rt,offset

Unconditional jump
- j address
- jal address
The Spim Simulator


Spim documentation
  Appendix A.9 SPIM Patterson & Hennessy pages A-38 to A75

Spim runnable code samples (Hello_World.s, simplecalc.s, ...)
  http://vip.cs.utsa.edu/classes/cs2734s98/overview.html

Other useful links
  http://www.cs.wisc.edu/~larus/spim.html
## MIPS registers and conventions

<table>
<thead>
<tr>
<th>Name</th>
<th>Number</th>
<th>Conventional usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0</td>
<td>0</td>
<td>Constant 0</td>
</tr>
<tr>
<td>$v0-$v1</td>
<td>2-3</td>
<td>Expression evaluation &amp; function results</td>
</tr>
<tr>
<td>$a0-$a3</td>
<td>4-7</td>
<td>Arguments 1 to 4</td>
</tr>
<tr>
<td>$t1-$t9</td>
<td>8-15,24,35</td>
<td>Temporary (not preserved across call)</td>
</tr>
<tr>
<td>$s0-$s7</td>
<td>16-23</td>
<td>Saved Temporary (preserved across call)</td>
</tr>
<tr>
<td>$k0-$k1</td>
<td>26-27</td>
<td>Reserved for OS kernel</td>
</tr>
<tr>
<td>$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>
</tbody>
</table>
# MIPS Register Name translation

# calculate \( f = (g + h) - (i + j) \) (PH p. 109, file: simplecalc.s)

<table>
<thead>
<tr>
<th>Assembler .s</th>
<th>Translated (1 to 1 mapping)</th>
</tr>
</thead>
<tbody>
<tr>
<td>addi $s1, $0, 5</td>
<td>addi $17, $0, 5 #g = 5</td>
</tr>
<tr>
<td>addi $s2, $0, -20</td>
<td>addi $18, $0, -20 #h = -20</td>
</tr>
<tr>
<td>addi $s3, $0, 13</td>
<td>addi $19, $0, -20 #i = 13</td>
</tr>
<tr>
<td>addi $s4, $0, 3</td>
<td>addi $20, $0, 3 #j = 3</td>
</tr>
<tr>
<td>add $t0, $s1, $s2</td>
<td>add $8, $17, $18 #t0=g + h</td>
</tr>
<tr>
<td>add $t1, $s3, $s4</td>
<td>add $9, $19, $20 #t1=i + j</td>
</tr>
<tr>
<td>sub $s0, $t0, $t1</td>
<td>sub $16, $8, $9 #f=(g+h)-(i+j)</td>
</tr>
</tbody>
</table>
System call 1: print_int $a0

- System calls are used to interface with the operating system to provide **device independent** services.

- System call 1 converts the binary value in register $a0 into ascii and displays it on the console.

- This is equivalent in the C Language: `printf("%d", $a0)`

**Assembler .s**

<table>
<thead>
<tr>
<th>Assembly</th>
<th>Translated (1 to 1 mapping)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>li $v0, 1</code></td>
<td><code>ori $2, $0, 1  #print_int (system call 1)</code></td>
</tr>
<tr>
<td><code>add $a0,$0,$s0</code></td>
<td><code>add $4,$0,$16  #put value to print in $a0</code></td>
</tr>
<tr>
<td>syscall</td>
<td>syscall</td>
</tr>
</tbody>
</table>
# System Services

<table>
<thead>
<tr>
<th>Service</th>
<th>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>$v0=integer</td>
<td></td>
</tr>
<tr>
<td>read_float</td>
<td>6</td>
<td>$f0=float</td>
<td></td>
</tr>
<tr>
<td>read_double</td>
<td>7</td>
<td>$f0=double</td>
<td></td>
</tr>
<tr>
<td>read_string</td>
<td>8</td>
<td>$a0=buf, $a1=len</td>
<td></td>
</tr>
<tr>
<td>sbrk</td>
<td>9</td>
<td>$a0=amount</td>
<td>$v0=address</td>
</tr>
<tr>
<td>exit</td>
<td>10</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
System call 4: print_string $a0

- System call 4 copies the contents of memory located at $a0 to the console until a zero is encountered.
- This is equivalent in the C Language: printf("%s", $a0)

Assembler .s
.data
.globl msg3
msg3: .asciiz "\nThe value of f is: 
.text
li $v0, 4 ori $2, $0, 4 #print_string
la $a0, msg3 lui $4,4097 #address of string
syscall syscall
syscall

Translated (1 to 1 mapping)

msg3 is just a label but must match
Note the “z” in asciiz
.asciiz data representations

.data: items are place in the data segment

    which is not the same the same as the .text segment!

Assembler.s

msg3:     .asciiz \nThe va"

Same as in assembler.s

msg3:     .byte  '\n','T','h','e',' ','v','a', 0

Same as in assembler.s

msg3:     .byte  0x0a, 0x54, 0x68, 0x65
          .byte  0x20, 0x76, 0x61, 0x00

Same as in assembler.s

msg3:     .word 0x6568540a, 0x00617620

Translated in the .data segment: 0x6568540a 0x00617620

Big endian format
Memory layout: segments

- Segments allow the operating system to protect memory.
- Like Unix file systems: .text Execute only, .data R/W only.

```
addi $17,0,5
addi $18,0,-20
...
.asciiz "The value of f is "
```

```
0x00400000
Reserved
.text segment
.data segment
.stack segment

$sp = top of stack
0x7fffffff
```

- 0x00400000
- 0x7fffffff
Hello, World: hello.s

```c
# main( ) {
#     printf("\nHello World\n");
# }

.globl main
main:
    addu $s7, $0, $ra  #save the return address in a global reg.
.data
.globl hello
hello: .asciiz "\nHello World\n"  #string to print
.text
li $v0, 4  # print_str (system call 4)
l$sa0, hello  # $a0=address of hello string
syscall

# Usual stuff at the end of the main
addu $ra, $0, $s7  #restore the return address
jr $ra  #return to the main program
add $0, $0, $0  #nop
```

Note: alternating .text, .data, .text
Simplecalc.s (PH p. 109)

.globl main
main:  addu $s7, $0, $ra #save the return address
       addi $s1, $0, 5  #g = 5
       addi $s2, $0, -20 #h = -20
       addi $s3, $0, 13 #i = 13
       addi $s4, $0, 3  #j = 3
       add $t0, $s1, $s2 #register $t0 contains g + h
       add $t1, $s3, $s4 #register $t1 contains i + j
       sub $s0, $t0, $t1 #f = (g + h) - (i + j)
       li $v0, 4     #print_str (system call 4)
       la $a0, message # address of string
       syscall
       li $v0, 1     #print_int (system call 1)
       add $a0, $0, $s0 #put value to print in $a0
       syscall
       addu $ra, $0, $s7 #restore the return address
       jr $ra  #return to the main program
       add $0, $0, $0  #nop

.data
.globl message
message:  .asciiz \nThe value of f is: " #string to print
Simplecalc.s without symbols  (PH p. 109)

<table>
<thead>
<tr>
<th>Address</th>
<th>Assembly Instruction</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00400020</td>
<td>addu $23, $0, $31</td>
<td># addu $s7, $0, $ra</td>
</tr>
<tr>
<td>0x00400024</td>
<td>addi $17, $0, 5</td>
<td># addi $s1, $0, 5</td>
</tr>
<tr>
<td>0x00400028</td>
<td>addi $18, $0, -20</td>
<td># addi $s2, $0, -20</td>
</tr>
<tr>
<td>0x0040002c</td>
<td>addi $19, $0, 13</td>
<td># addi $s3, $0, 13</td>
</tr>
<tr>
<td>0x00400030</td>
<td>addi $20, $0, 3</td>
<td># addi $s4, $0, 3</td>
</tr>
<tr>
<td>0x00400034</td>
<td>add $8, $17, $18</td>
<td># add $t0, $s1, $s2</td>
</tr>
<tr>
<td>0x00400038</td>
<td>add $9, $19, $20</td>
<td># add $t1, $s3, $s4</td>
</tr>
<tr>
<td>0x0040003c</td>
<td>sub $16, $8, $9</td>
<td># sub $s0, $t0, $t1</td>
</tr>
<tr>
<td>0x00400040</td>
<td>ori $2, 0, 4</td>
<td>#print_str (system call 4)</td>
</tr>
<tr>
<td>0x00400044</td>
<td>lui $4, 0x10010000</td>
<td># address of string</td>
</tr>
<tr>
<td>0x00400048</td>
<td>syscall</td>
<td></td>
</tr>
<tr>
<td>0x0040004c</td>
<td>ori $2, 1</td>
<td>#print_int (system call 1)</td>
</tr>
<tr>
<td>0x00400050</td>
<td>add $4, $0, $16</td>
<td>#put value to print in $a0</td>
</tr>
<tr>
<td>0x00400054</td>
<td>syscall</td>
<td></td>
</tr>
<tr>
<td>0x00400058</td>
<td>addu $31, $0, $23</td>
<td>#restore the return address</td>
</tr>
<tr>
<td>0x0040005c</td>
<td>jr $31</td>
<td>#return to the main program</td>
</tr>
<tr>
<td>0x00400060</td>
<td>add $0, $0, $0</td>
<td>#nop</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>.data</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x10010000</td>
<td>.word 0x6568540a</td>
<td></td>
</tr>
<tr>
<td></td>
<td>.word 0x6c617620</td>
<td></td>
</tr>
<tr>
<td></td>
<td>.word 0xf206575</td>
<td></td>
</tr>
<tr>
<td></td>
<td>.word 0x20662066</td>
<td></td>
</tr>
<tr>
<td></td>
<td>.word 0x203a7369</td>
<td></td>
</tr>
<tr>
<td></td>
<td>.word 0x00000000</td>
<td></td>
</tr>
</tbody>
</table>
### Single Stepping

<table>
<thead>
<tr>
<th>$pc</th>
<th>$t0</th>
<th>$t1</th>
<th>$s0</th>
<th>$s1</th>
<th>$s2</th>
<th>$s3</th>
<th>$s4</th>
<th>$s7</th>
<th>$ra</th>
</tr>
</thead>
<tbody>
<tr>
<td>$8</td>
<td>$9</td>
<td>$16</td>
<td>$17</td>
<td>$18</td>
<td>$19</td>
<td>$20</td>
<td>$23</td>
<td>$31</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>$t0</th>
<th>$t1</th>
<th>$s0</th>
<th>$s1</th>
<th>$s2</th>
<th>$s3</th>
<th>$s4</th>
<th>$s7</th>
<th>$ra</th>
</tr>
</thead>
<tbody>
<tr>
<td>00400034</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>5</td>
<td>fffffffec</td>
<td>0d</td>
<td>3</td>
<td>400018</td>
<td>400018</td>
</tr>
<tr>
<td>00400038</td>
<td>fffffff1</td>
<td>?</td>
<td>?</td>
<td>5</td>
<td>fffffffec</td>
<td>0d</td>
<td>?</td>
<td>?</td>
<td>400018</td>
</tr>
<tr>
<td>0040003c</td>
<td>?</td>
<td>10</td>
<td>?</td>
<td>5</td>
<td>fffffffec</td>
<td>0d</td>
<td>?</td>
<td>?</td>
<td>400018</td>
</tr>
<tr>
<td>00400040</td>
<td>?</td>
<td>?</td>
<td>fffffffe1</td>
<td>5</td>
<td>fffffffec</td>
<td>0d</td>
<td>?</td>
<td>?</td>
<td>400018</td>
</tr>
</tbody>
</table>

Values changes after the instruction!
Sun Microsystems SPARC Architecture

• In 1987, Sun Microsystems introduced a 32-bit RISC architecture called SPARC.

• Sun’s UltraSparc workstations use this architecture.

• The general purpose registers are 32 bits, as are memory addresses.

• Thus $2^{32}$ bytes can be addressed.

• In addition, instructions are all 32 bits long.

• SPARC instructions support a variety of integer data types from single bytes to double words (eight bytes) and a variety of different precision floating-point types.
SPARC Registers

• The SPARC provides access to 32 registers
  • regs 0  %g0  ! global constant 0 (MIPS $zero, $0)
  • regs 1-7  %g1-%g7  ! global registers
  • regs 8-15  %o0-%o7  ! out  (MIPS $a0-$a3,$v0-$v1,$ra)
  • regs 16-23  %L0-%L7  ! local  (MIPS $s0-$s7)
  • regs 24-31  %i0-%i7  ! in registers (caller’s out regs)

• The global registers refer to the same set of physical registers in all procedures.

• Register 15 (%o7) is used by the call instruction to hold the return address during procedure calls (MIPS ($ra)).

• The other registers are stored in a register stack that provides the ability to manipulate register windows.

• The local registers are only accessible to the current procedure.
• When a procedure is called, parameters are passed in the out registers and the register window is shifted 16 registers further into the register stack.

• This makes the in registers of the called procedure the same as the out registers of the calling procedure.

• in registers: arguments from caller (MIPS %a0-%a3)

• out registers: When the procedure returns the caller can access the returned values in its out registers (MIPS $v0-%v1).
SPARC instructions

Arithmetic

add %l1, %i2, %l4  ! local %l4 = %l1 + i2
add %l4, 4, %l4   ! Increment %l4 by four.
mov 5, %l1        ! %l1 = 5

Data Transfer

ld [%l0], %l1     ! %l1 = Mem[%l0]
ld [%l0+4], %l1   ! %l1 = Mem[%l0+4]
st %l1, [%l0+12] ! Mem[%l0+12] = %l1

Conditional

cmp %l1, %l4     ! Compare and set condition codes.
bg  L2            ! Branch to label L2 if %l1 > %l14
nop               ! Do nothing in the delay slot.
SPARC functions

Calling functions

mov %l1, %o0        ! first parameter = %l1
mov %l2, %o1        ! second parameter = %l2
call fib            ! %o0=.fib(%o0,%o1,…%o7)
nop                 ! delay slot: no op
mov %o0, %l3        ! %i3 = return value

Assembler

gcc hello.s          ! executable file=a.out
gcc hello.s -o hello ! executable file=hello
gdb hello           ! GNU debugger
SPARC Hello, World.

.data
hmes:.asciz Hello, World\n"

.text
.global main ! visible outside

main:
add %r0,1,%%o0 ! %r8 is %o0, first arg
sethi %hi(hmes),%o1 ! %r9, (%o1) second arg
or %o1, %lo(hmes),%o1
or %r0,14,%o2 ! count in third arg
add %r0,4,%g1 ! system call number 4
ta 0 ! call the kernal

add %r0,%r0,%o0
add %r0,1,%g1 ! %r1, system call
ta 0 ! call the system exit
gdb: GNU debugger basics

This is the symbolic debugger for the gcc compiler. So keep all your source files and executables in the same current working directory.

**gcc hello.s**  
Assemble the program hello.s and put the executable in a.out (all files that end in “.s” are assembly files).

**gdb a.out**  
Start the debugger and read the a.out file.

**h**  
gdb Help command: lists all the command groups.

**info files**  
shows the program memory layout (.text, .data, …)

**info var**  
shows global and static variables ( _start )

**b _start**  
set the first breakpoint at beginning of program

**info break**  
displays your current breakpoints

**r**  
Start running your program and it will stop at _start
gdb: register & memory contents

info reg          displays the registers

set $L1=0x123    set the register %L1 to 0x123

display $L1     display register %L1 after every single step

info display    show all display numbers

undisplay <number>  stop displaying item <number>

diss 0x120 0x200 dissasemble memory location 0x120 to 0x200

x/b   0x120     display memory location 0x120 as a byte
x/4b  0x120     display memory location 0x120 as four bytes
x/4c  0x120     display memory location 0x120 as four characters
x/s   0x120     display memory location 0x120 as a ascii string
x/h   0x120     display memory location 0x120 as a halfword
x/w   0x120     display memory location 0x120 as a word
gdb: single stepping

si  single step exactly one instruction

b *0x2064  This sets a Breakpoint in your program at address 0x2064. Set as many as you need.

info break  Display all the breakpoints

c  Continue running the program until the next breakpoint. Set more breakpoints or do more “si” or restart program “r”

d  Delete all break points.

q  Quit debugging.
gdb a at Glance

b *0x2064 This sets a Breakpoint in your program at address 0x2064.

info break Display all the breakpoints

r Start Running your program and stop at any breakpoints.

c Continue running the program until the next breakpoint.

n Single step a single source line but do NOT enter the subroutine.

s Single step a single source line but enter the subroutine

disp <variable_name> DISPLAY the contents of a variable in your program.

und <display number> UN-Display a debugging variable (use disp line number)

cl Delete all breakpoints