Instructor: Francis G. Wolff wolff@eecs.cwru.edu Case Western Reserve University

This presentation uses powerpoint animation: please viewshow

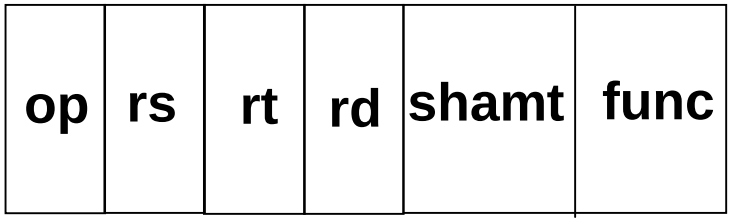
MIPS instructions

δ^{32}=sign extend 16 bit number to 32 bits

ALU	alu $rd,$rs,$rt	$rd = $rs <alu> $rt
JR	jr $rs	$pc = $rs
ALUi	alui $rd,$rs,value16	$rd = $rs <alu> δ^{32}(value16)
Data Transfer	lw $rt,offset16($rs)	$rt = Mem[$rs + δ^{32}(offset16)]
	sw $rt,offset16($rs)	Mem[$rs + δ^{32}(offset16)]=$rt
Branch	beq $rs,$rt,offset16	$pc = ($rt == $rs)? ($pc+4+(δ^{32}(offset16)<<2))):($pc+4);
Jump	j address	$pc=($pc & 0xFC00000)|(addr<<2)
Jump&Link	jal address	$ra = $pc+4; $pc=($pc & 0xFC00000)|(addr<<2)

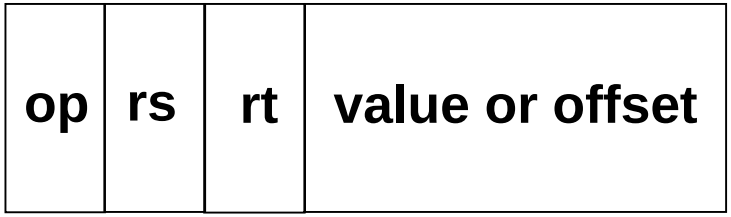
MIPS fixed sized instruction formats

R - Format



ALU	alu $rd,$rs,$rt
jr	jr $rs

I - Format

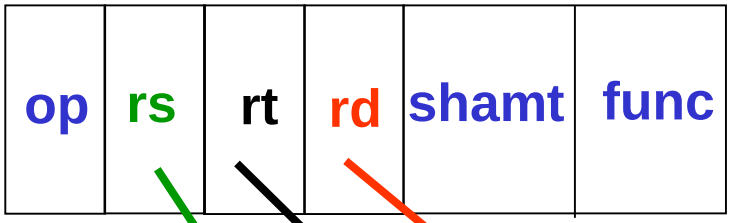


ALUi	alui $rt,$rs,value
Data Transfer	lw $rt,offset($rs) sw $rt,offset($rs)
Branch	beq $rs,$rt,offset

J - Format

Jump	j address
Jump&Link	jal address

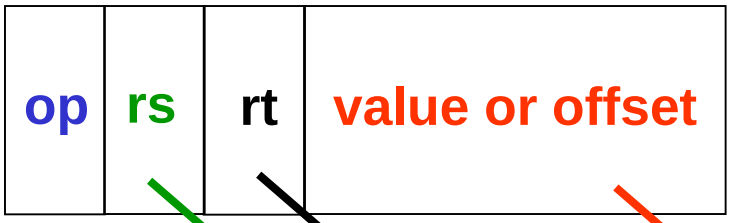
Assembling MIPS Instructions



ALU alu $rd,$rs,$rt

0x00400020 addu $23, $0, $31

000000:00000:11111:10111:00000:100001



ALUi alui $rt,$rs,value

0x00400024 addi $17, $0, 5

001000:00000:10001:0000000000000101

MIPS instruction formats

Arithmetic

addi $rt, $rs, value

add $rd,$rs,$rt

jr $rs

Data Transfer

lw $rt,offset($rs)

sw $rt,offset($rs)

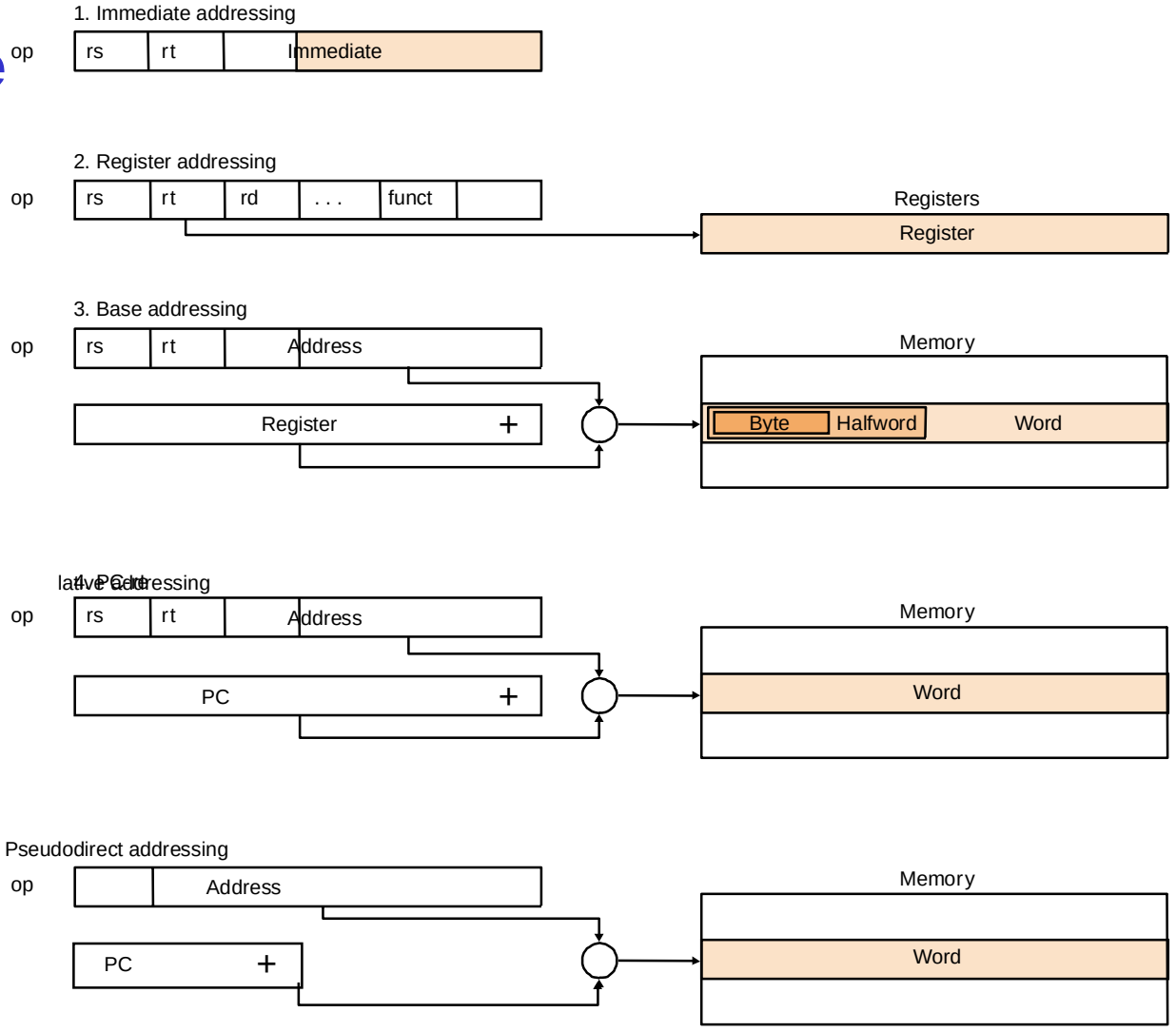
Conditional branch

beq $rs,$rt,offset

Unconditional jump

j address

jal address



The Spim Simulator

Spim download: ftp://ftp.cs.wisc.edu/pub/spim

unix: ftp://ftp.cs.wisc.edu/pub/spim/spim.tar.gz

win95: ftp://ftp.cs.wisc.edu/pub/spim/spimwin.exe

Spim documentation

Main document

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

ftp://ftp.cs.wisc.edu/pub/spim/spim_documentation.ps

ftp://ftp.cs.wisc.edu/pub/spim/spimwin.ps

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

http://www.cs.bilkent.edu.tr/~baray/cs224/howspim1.html

MIPS registers and conventions

Name	Number	Conventional usage
$0	0	Constant 0
$v0-$v1	2-3	Expression evaluation & function results
$a0-$a3	4-7	Arguments 1 to 4
$t1-$t9	8-15,24,35	Temporary (not preserved across call)
$s0-$s7	16-23	Saved Temporary (preserved across call)
$k0-$k1	26-27	Reserved for OS kernel
$gp	28	Pointer to global area
$sp	29	Stack pointer
$fp	30	Frame pointer
$ra	31	Return address (used by function call)

MIPS Register Name translation

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

Assember .s		Translated (1 to 1 mapping)		
addi	$s1, $0, 5	addi	$17, $0, 5	#g = 5
addi	$s2, $0, -20	addi	$18, $0, -20	#h = -20
addi	$s3, $0, 13	addi	$19, $0, -20	#i = 13
addi	$s4, $0, 3	addi	$20, $0, 3	#j = 3
add	$t0, $s1, $s2	add	$8, $17, $18	#$t0=g + h
add	$t1, $s3, $s4	add	$9, $19, $20	#$t1=i + j
sub	$s0, $t0, $t1	sub	$16, $8, $9	#f=(g+h)-(i+j)

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)

Assember .s		Translated (1 to 1 mapping)		
li	$v0, 1	ori	$2, $0, 1	#print_int (system call 1)
add	$a0,$0,$s0	add	$4,$0,$16	#put value to print in $a0
syscall		syscall		

System Services

Service	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		$v0=integer
read_float	6		$f0=float
read_double	7		$f0=double
read_string	8	$a0=buf, $a1=len	
sbrk	9	$a0=amount	$v0=address
exit	10		

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)

Assember .s

```
.data
.globl msg3
msg3:   .asciiz "\nThe value of f is: "
.text
li      $v0, 4
la      $a0,msg3
syscall
```

Translated (1 to 1 mapping)

```
ori    $2, $0, 4    #print_string
lui    $4,4097      #address of string
syscall
```

Note the "z" in asciiz

msg3 is just a label but must match

.asciiz data representations

.data: items are place in the data segment

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

Assember .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
```

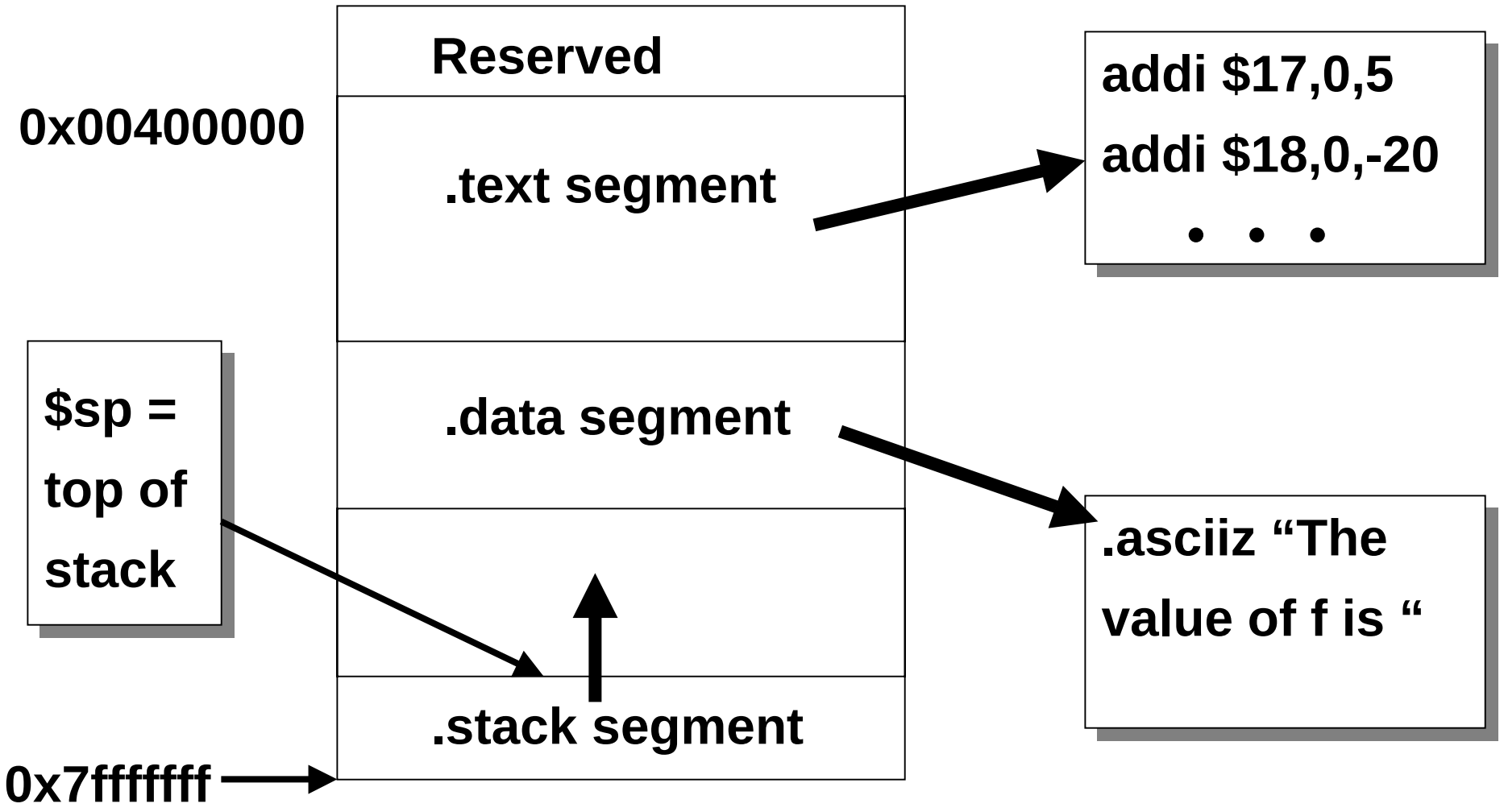
Same as in assembler.s

```
msg3:       .word 0x6568540a,  0x00617620
```

Big endian format

Translated in the .data segment: 0x6568540a 0x00617620

Memory layout: segments



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

Hello, World: hello.s

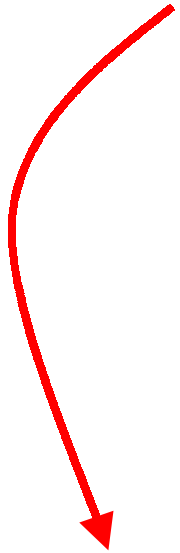
```
# main( ) {
#         printf("\nHello World"\n");
# }
        .globl main
main:                                   #main has to be a global label
        addu    $s7, $0, $ra            #save the return address in a global reg.
        .data
        .globl  hello
hello:  .asciiz "\nHello World\n"       #string to print
        .text
        li      $v0, 4                  # print_str (system call 4)
        la      $a0, 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)

Order of .text and .data not important

```
        .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)

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

```
                .data
0x10010000      .word   0x6568540a, 0x6c617620, 0x6f206575
                .word   0x20662066, 0x203a7369, 0x00000000
```

Single Stepping

Values changes after the instruction!

$pc	$t0 $8	$t1 $9	$s0 $16	$s1 $17	$s2 $18	$s3 $19	$s4 $20	$s7 $23	$ra $31
00400020	?	?	?	?	?	?	?	?	400018
00400024	?	?	?	?	?	?	?	400018	400018
00400028	?	?	?	5	?	?	?	400018	400018
0040002c	?	?	?	5	fffffec	?	?	400018	400018
00400030	?	?	?	5	fffffec	0d	?	400018	400018
00400034	?	?	?	5	fffffec	0d	3	400018	400018
00400038	fffff1	?	?	5	fffffec	0d	?	400018	400018
0040003c	?	10	?	5	fffffec	0d	?	400018	400018
00400040	?	?	ffffffe1	5	fffffec	0d	?	400018	400018

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)

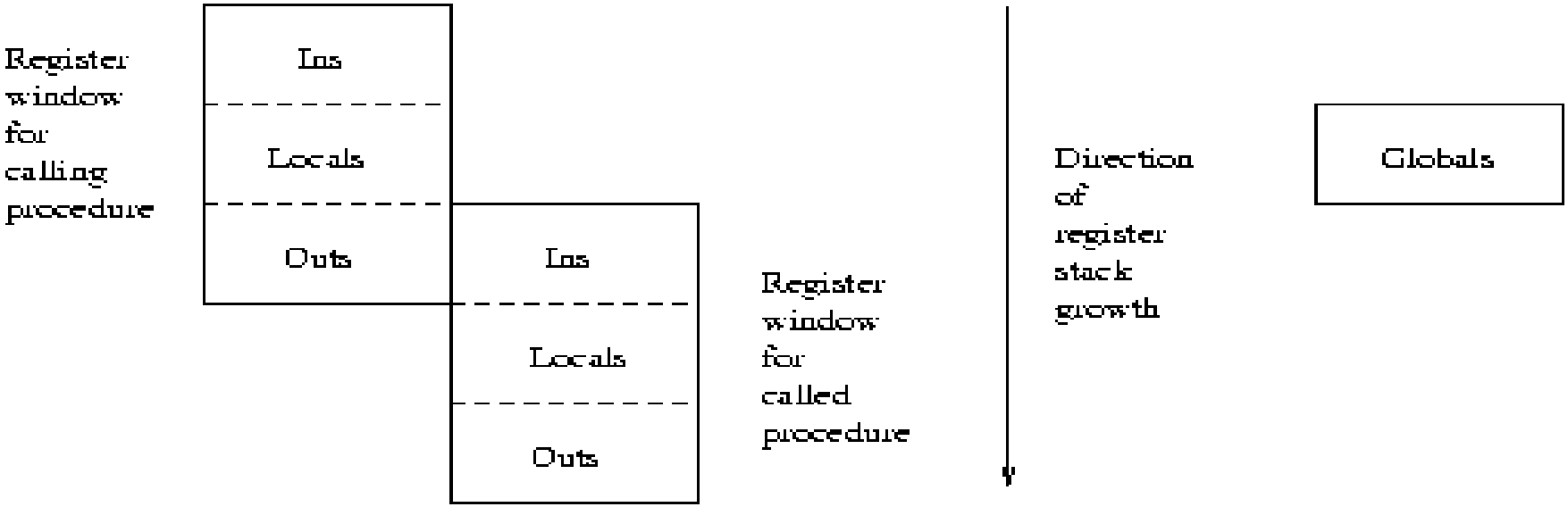
• The global registers refer to the same set of physical registers in all procedures.

• 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.

SPARC Register windows



- 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+l2]= %l1
```

Conditional

```
cmp %l1, %l4     ! Compare and set condition codes.
bg  L2           ! Branch to label L2 if %l1 > %l4
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	dissassemble 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 asciiz 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)