

## Outline

- ISAs in General
- MIPS Assembly Programming
- Other Instruction Sets

## But first: SPIM

- SPIM is a program that simulates the behavior of MIPS32 computers
  - Can run MIPS32 assembly language programs
  - You will use SPIM to run/test the assembly language programs you write for homeworks in this class
- Two flavors of same thing:
  - spim: command line interface
  - xspim: xwindows interface

## MIPS Assembly Language

- One instruction per line
- **Numbers** are base-10 integers or Hex with leading **0x**
- **Identifiers**: alphanumeric, `_`, `.` string starting in a letter or `_`
- **Labels**: identifiers starting at the beginning of a line followed by `:"`
- **Comments**: everything following `#` until end-of-line
- Instruction format: Space and `,` separated fields
  - [Label:] `<op>` reg1, [reg2], [reg3] [# comment]
  - [Label:] `<op>` reg1, offset(reg2) [# comment]
  - `.Directive` [arg1], [arg2], ...

## MIPS Pseudo-Instructions

- Pseudo-instructions: extend the instruction set for convenience
- Examples

```
• move $2, $4           # $2 = $4, (copy $4 to $2)
  Translates to:
  add $2, $4, $0

• li $8, 40            # $8 = 40, (load 40 into $8)
  addi $8, $0, 40

• sd $4, 0($29)        # mem[$29] = $4; Mem[$29+4] = $5
  sw $4, 0($29)
  sw $5, 4($29)

• la $4, 0x1000056c     # Load address $4 = <address>
  lui $4, 0x1000
  ori $4, $4, 0x056c
```

## Assembly Language (cont.)

- **Directives:** tell the assembler what to do
- Format `".<string> [arg1], [arg2] . . .`

- **Examples**

```
.data [address]      # start a data segment
.text [address]      # start a code segment
.align n             # align segment on 2n byte boundary
.asciz <string>      # store a string in memory
.asciiz <string>     # store null-terminated string in memory
.word w1, w2, . . . , wn  # store n words in memory
```

Let's see how these get used in programs ...

## A Simple Program

- Add two numbers *x* and *y*:

```
.text                # declare text segment
.align 2             # align it on 4-byte (word) boundary
main:                # label for main
    la $3, x          # load address of x into R3 (pseudo-inst)
    lw $4, 0($3)      # load value of x into R4
    la $3, y          # load address of y into R3 (pseudo-inst)
    lw $5, 0($3)      # load value of y into R5
    add $6, $4, $5    # compute x+y
    jr $31            # return to calling routine

.data                # declare data segment
.align 2             # align it on 4-byte boundary
x:.word 10           # initialize x to 10
y:.word 3            # initialize y to 3
```

*Note: program  
doesn't obey register  
conventions*

## Another example: The C / C++ code

```
#include <iostream.h>

int main ( )
{
    int i;
    int sum = 0;
    for(i=0; i <= 100; i++)
        sum = sum + i*i ;
    cout << "The answer is " << sum << endl;
}
```

Let's write the assembly ...

## Assembly Language Example 1

```
.text
.align 2
main:
    move $14, $0 # i = 0
    move $15, $0 # tmp = 0
    move $16, $0 # sum = 0
loop:
    mul $15, $14, $14 # tmp = i*i
    add $16, $16, $15 # sum = sum + tmp
    addi $14, $14, 1 # i++
    ble $14, 100, loop # if i < 100, goto loop

# how are we going to print the answer here?
# and how are we going to exit the program?
```

## System Call Instruction

- System call is used to communicate with the operating system and request services (memory allocation, I/O)
    - `syscall` instruction in MIPS
  - SPIM supports "system-call-like"
- Load system call code into register `$v0`
    - Example: if `$v0==1`, then `syscall` will print an integer
  - Load arguments (if any) into registers `$a0`, `$a1`, or `$f12` (for floating point)
  - `syscall`
- Results returned in registers `$v0` or `$f0`

## SPIM System Call Support

<u>code</u>	<u>service</u>	<u>ArgType</u>	<u>Arg/Result</u>
1	print	int	<code>\$a0</code>
2	print	float	<code>\$f12</code>
3	print	double	<code>\$f12</code>
4	print	string	<code>\$a0</code> (string address)
5	read	integer	integer in <code>\$v0</code>
6	read	float	float in <code>\$f0</code>
7	read	double	double in <code>\$f0</code> & <code>\$f1</code>
8	read	string	<code>\$a0=buffer</code> , <code>\$a1=length</code>
9	sbrk	<code>\$a0=amount</code>	address in <code>\$v0</code>
10	exit		

## Echo number and string

```
.text
main:
    li $v0, 5      # code to read an integer
    syscall        # do the read (invokes the OS)
    move $a0, $v0  # copy result from $v0 to $a0

    li $v0, 1      # code to print an integer
    syscall        # print the integer

    li $v0, 4      # code to print string
    la $a0, nln    # address of string (newline)
    syscall

# code continues on next slide ...
```

## Echo Continued

```
    li $v0, 8      # code to read a string
    la $a0, name   # address of buffer (name)
    li $a1, 8      # size of buffer (8 bytes)
    syscall

    la $a0, name   # address of string to print
    li $v0, 4      # code to print a string
    syscall

    jr $31         # return

.data
.align 2
name: .word 0,0
nln:  .asciiz "\n"
```

## Review: Procedure Call and Return

<code>int equal(int a1, int a2)</code>	0x10000	<code>addi \$1, \$0, 43</code>
<code>{</code>	0x10004	<code>addi \$2, \$0, 2</code>
<code>  int tsame;</code>	0x10008	<code>jal 0x30408</code>
<code>  tsame = 0;</code>	0x1000c	??
<code>  if (a1 == a2)</code>	0x30408	<code>addi \$3, \$0, 0</code>
<code>    tsame = 1;</code>	0x3040c	<code>bne \$1, \$2, 8</code>
<code>    return(tsame);</code>	0x30410	<code>addi \$3, \$0, 1</code>
<code>}</code>	0x30414	<code>jr \$31</code>
<code>main()</code>		
<code>{</code>	<b>PC</b>	<b>\$31</b>
<code>  int x,y,same;</code>	0x10000	??
<code>  x = 43;</code>	0x10004	??
<code>  y = 2;</code>	0x10008	??
<code>  same = equal(x,y);</code>	0x30408	0x1000c
<code>  // other computation</code>	0x3040c	0x1000c
<code>}</code>	0x30410	0x1000c
	0x30414	0x1000c
	0x1000c	0x1000c

© 2008 Daniel J. Sorin  
from Roth and Lebeck

54

## Procedure Call Gap

### ISA Level

- Call and return instructions

### C/C++ Level

- Local name scope
  - Change tsame to same
- Recursion
- Arguments and return value (functions)

### Assembly Level

- **Must bridge gap between HLL and ISA**
- Supporting local names
- Passing arguments (arbitrary number?)

© 2008 Daniel J. Sorin  
from Roth and Lebeck

55

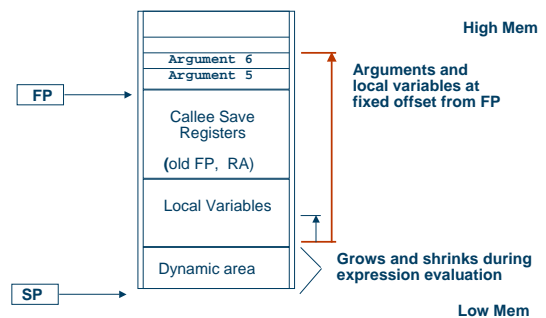
## Review: Procedure Call (Stack) Frame

- Procedures use a frame in the stack to:
  - Hold values passed to procedures as arguments
  - Save registers that a procedure may modify, but which the procedure's caller does not want changed
  - To provide space for local variables (variables with local scope)
  - To evaluate complex expressions

© 2008 Daniel J. Sorin  
from Roth and Lebeck

56

## MIPS Call-Return Linkage: Stack Frames



© 2008 Daniel J. Sorin  
from Roth and Lebeck

57

## MIPS Register Naming Conventions

0	zero constant	16	s0 callee saves
1	at reserved for assembler	...	
2	v0 expression evaluation &	23	s7
3	v1 function results	24	t8 temporary (cont'd)
4	a0 arguments	25	t9
5	a1	26	k0 reserved for OS kernel
6	a2	27	k1
7	a3	28	gp pointer to global area
8	t0 temporary: caller saves	29	sp Stack pointer
...		30	fp frame pointer
15	t7	31	ra return address

## MIPS/GCC Procedure Calling Conventions

### Calling Procedure

- Step-1: Pass the arguments
  - First four arguments (arg0-arg3) are passed in registers \$a0-\$a3
  - Remaining arguments are pushed onto the stack (in reverse order, arg5 is at the top of the stack)
- Step-2: Save caller-saved registers
  - Save registers \$t0-\$t9 if they contain live values at the call site
- Step-3: Execute a `jal` instruction

## MIPS/GCC Procedure Calling Conventions (cont.)

### Called Routine

- Step-1: Establish stack frame
  - Subtract the frame size from the stack pointer  
`subiu $sp, $sp, <frame-size>`
  - Typically, minimum frame size is 32 bytes (8 words)
- Step-2: Save callee saved registers in the frame
  - Register \$fp is always saved
  - Register \$ra is saved if routine makes a call
  - Registers \$s0-\$s7 are saved if they are used
- Step-3: Establish frame pointer
  - Add the stack <frame size> - 4 to the address in \$sp  
`addiu $fp, $sp, <frame-size> - 4`

## MIPS/GCC Procedure Calling Conventions (cont.)

### On return from a call

- Step-1: Put returned values in registers \$v0 and \$v1 (if values are returned)
- Step-2: Restore callee-saved registers
  - Restore \$fp and other saved registers: \$ra, \$s0 - \$s7
- Step-3: Pop the stack
  - Add the frame size to \$sp  
`addiu $sp, $sp, <frame-size>`
- Step-4: Return
  - Jump to the address in \$ra  
`jr $ra`

## Example2

```
# Program to add together list of 9 numbers
.text                # Code
.align 2
.globl main
main:                # MAIN procedure Entrance
    subu   $sp, 40      #\ Push the stack
    sw     $ra, 36($sp) # \ Save return address
    sw     $s3, 32($sp) # \
    sw     $s2, 28($sp) # > Entry Housekeeping
    sw     $s1, 24($sp) # / save registers on stack
    sw     $s0, 20($sp) # /
    move   $v0, $0      #/ initialize exit code to 0
    move   $s1, $0      #\
    la     $s0, list    # \ Initialization
    la     $s2, msg     # /
    la     $s3, list+36 #/
```

© 2008 Daniel J. Sorin  
from Roth and Lebeck

62

## Example2 (cont.)

```
#                                Main code segment
again:                            # Begin main loop
    lw     $t6, 0($s0)            #\
    addu   $s1, $s1, $t6         #/ Actual "work"
                                           # SPIM I/O
    li     $v0, 4                #\
    move   $a0, $s2              # > Print a string
    syscall                               #/
    li     $v0, 1                #\
    move   $a0, $s1              # > Print a number
    syscall                               #/
    li     $v0, 4                #\
    la     $a0, nlN              # > Print a string (eol)
    syscall                               #/
    addu   $s0, $s0, 4           #\ index update and
    bne   $s0, $s3, again       #/ end of loop
```

© 2008 Daniel J. Sorin  
from Roth and Lebeck

63

## Example2 (cont.)

```
#                                Exit Code
    move   $v0, $0              #\
    lw     $s0, 20($sp)         # \
    lw     $s1, 24($sp)         # \
    lw     $s2, 28($sp)         # \ Closing Housekeeping
    lw     $s3, 32($sp)         # / restore registers
    lw     $ra, 36($sp)         # / load return address
    addu   $sp, 40              #/ Pop the stack
    jr     $ra                  #/ exit(0) ;
    .end   main                 # end of program

#                                Data Segment
    .data                        # Start of data segment
list:    .word 35, 16, 42, 19, 55, 91, 24, 61, 53
msg:     .ascii "The sum is "
nlN:     .ascii "\n"
```

© 2008 Daniel J. Sorin  
from Roth and Lebeck

64

## Some Details of the MIPS instruction set

- Register zero always has the value zero
  - Even if you try to write it!
- jal puts the return address PC+4 into the link register (\$ra)
- All instructions change all 32 bits of the destination register (lui, lb, lh) and read all 32 bits of sources (add, sub, and, or, ...)
- Immediate arithmetic and logical instructions are extended as follows:
  - logical immediates are zero extended to 32 bits
  - arithmetic immediates are sign extended to 32 bits
- lb and lh extend data as follows:
  - lbu, lhu are zero extended
  - lb, lh are sign extended

© 2008 Daniel J. Sorin  
from Roth and Lebeck

65