# Introduction to MIPS and SPIM

CENG 444 Fall 2007

Hande Çelikkanat

Slides adapted from:

- "Assemblers, Linkers, and the SPIM Simulator", James R. Larus
- "SPIM S20: A MIPS R2000 Simulator", James R. Larus
- "http://chortle.ccsu.edu/AssemblyTutorial/index.html", Bradley Kjell

# MIPS Architecture

RISC architecture
32 general-purpose 32-bit registers

The basic MIPS architecture is difficult to program
The assembler reorganizes instructions to fill the delay slots
Also provides pseudoinstructions
a richer instruction set than the actual hardware
extracted into a number of real machine instructions

# SPIM

a simulator that runs programs for MIPS arch.
can read and immediately execute assembly files
contains a debugger
has an x version

- xspim for Linux
- PCspim for Windows

all the same functionality, but you can see all the register values all the time etc.

# Spim Commands (1/4)

*exit*
Exit the simulator

*read "file"* ≡ *load "file"*
Read file of assembly language into SPIM. If the file has already been read into SPIM, the system must be *reinitialized* or global labels will be multiply defined

*run <addr>*
Start running a program

Optional address --> the program starts at that address

Otherwise --> starts at the global label __start

--> usually the default start-up code that calls the routine at the global label *main*

# Spim Commands (2/4)

*step <N>*
Step the program for N instructions

default = 1

Print instructions as they execute

*continue*
Continue program execution without stepping

# Spim Commands (3/4)

*print $N / $fN*
Print register N / floating-point register N

*print_all_regs / print_all_regs hex*
Print all registers in hexadecimal

*print addr*
Print the contents of memory at address addr

*print_sym*
Print the names and addresses of the global labels known to SPIM
Labels are local by default and become global only when declared in a .globl assembler directive

# Spim Commands (4/4)

*reinitialize*
Clear the memory and registers

*breakpoint addr*
Set a breakpoint at address addr
    --> addr can be either a memory address or symbolic label

*delete addr*
Delete all breakpoints at address addr

*list*
List all breakpoints

# On Pseudoinstructions

i.e, MIPS hardware only provides
    branch if a register is equal to / not equal to 0

Other conditional branches
    such as *branch if greater or equal*
    are pseudoinstructions that are expanded into several
        machine instructions

When single-stepping or examining memory
    the instructions that you see are machine instructions

# xspim

xspim

**Register Display**

```
PC     = 00000000  EPC    = 00000000  Cause    = 0000000  BadVaddr = 00000000
Status= 00000000  HI     = 00000000  LO       = 0000000
                         General Registers
R0  (r0)  = 00000000  R8   (t0)  = 00000000  R16  (s0)  = 0000000  R24  (t8)  = 00000000
R1  (at)  = 00000000  R9   (t1)  = 00000000  R17  (s1)  = 0000000  R25  (s9)  = 00000000
R2  (v0)  = 00000000  R10  (t2)  = 00000000  R18  (s2)  = 0000000  R26  (k0)  = 00000000
R3  (v1)  = 00000000  R11  (t3)  = 00000000  R19  (s3)  = 0000000  R27  (k1)  = 00000000
R4  (a0)  = 00000000  R12  (t4)  = 00000000  R20  (s4)  = 0000000  R28  (gp)  = 00000000
R5  (a1)  = 00000000  R13  (t5)  = 00000000  R21  (s5)  = 0000000  R29  (gp)  = 00000000
R6  (a2)  = 00000000  R14  (t6)  = 00000000  R22  (s6)  = 0000000  R30  (s8)  = 00000000
R7  (a3)  = 00000000  R15  (t7)  = 00000000  R23  (s7)  = 0000000  R31  (ra)  = 00000000
                         Double Floating Point Registers
FP0       = 0.000000  FP8        = 0.000000  FP16       = 0.00000  FP24       = 0.000000
FP2       = 0.000000  FP10       = 0.000000  FP18       = 0.00000  FP26       = 0.000000
FP4       = 0.000000  FP12       = 0.000000  FP20       = 0.00000  FP28       = 0.000000
FP6       = 0.000000  FP14       = 0.000000  FP22       = 0.00000  FP30       = 0.000000
                         Single Floating Point Registers
```

**Control Buttons**

quit | load | run | step | clear | set value
print | breakpt | help | terminal | mode

**User and Kernel Text Segments**

Text Segments

```
[0x00400000]  0x8fa40000 lw R4, 0(R29)   []
[0x00400004]  0x27a50004 addiu R5, R29, 4 []
[0x00400008]  0x24a60004 addiu R6, R5, 4 []
[0x0040000c]  0x00041090 sll R2, R4, 2
[0x00400010]  0x00c23021 addu R6, R6, R2
[0x00400014]  0x0c000000 jal 0x00000000 []
[0x00400018]  0x3402000a ori R0, R0, 10 []
[0x0040001c]  0x0000000c syscall
```

**Data and Stack Segments**

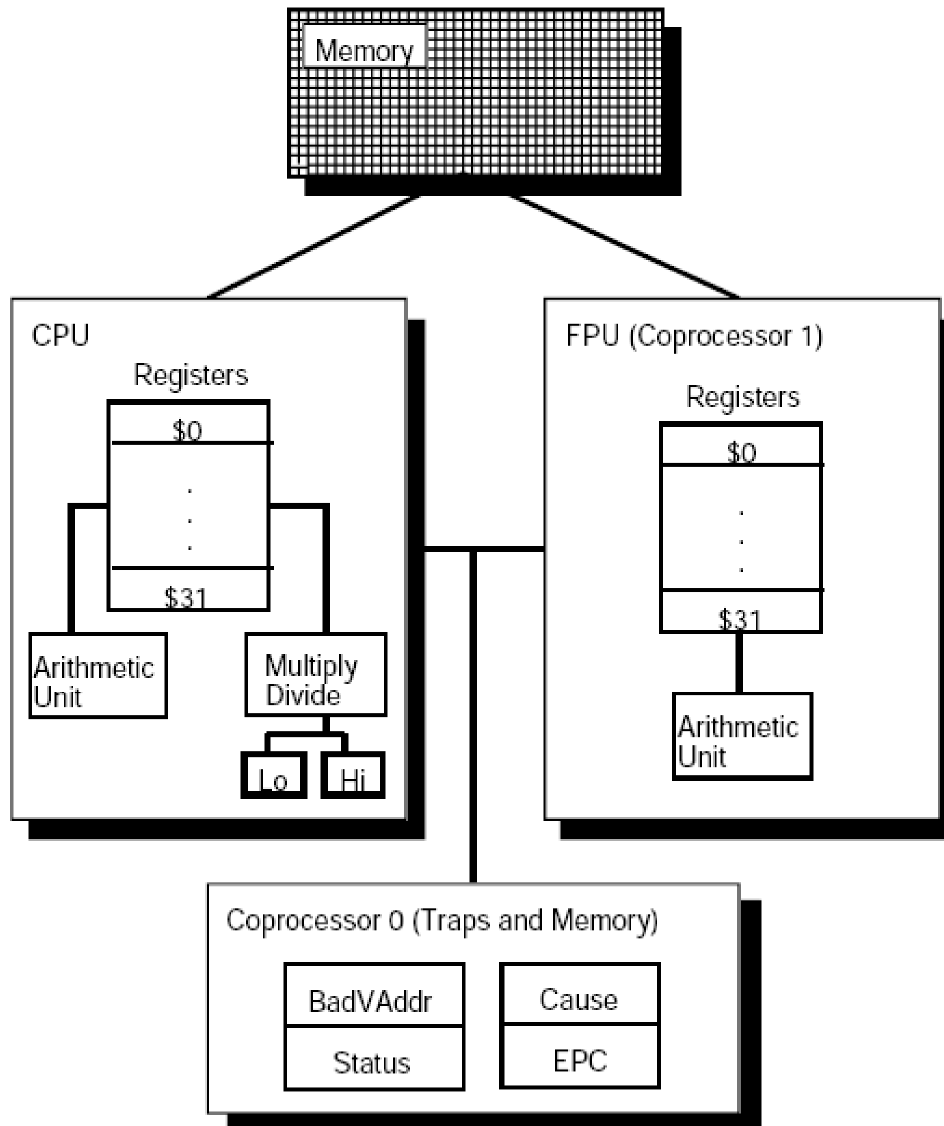
Data Segments

```
[0x10000000]...[0x10010000]  0x00000000
[0x10010004]   0x74706563   0x206e6f69   0x636f2000
[0x10010010]   0x72727563   0x61206465   0x6920646e   0x726f6e67
[0x10010020]   0x000a6465   0x495b2020   0x7265746e   0x74707572
[0x10010030]   0x0000205d   0x20200000   0x616e555b   0x6e67696c
[0x10010040]   0x61206465   0x65726464   0x69207373   0x6e69206e
[0x10010050]   0x642f7473   0x20617461   0x63746566   0x00205d68
[0x10010060]   0x555b2020   0x696c616e   0x64656e67   0x64646120
[0x10010070]   0x73736572   0x206e6920   0x726f7473   0x00205d65
```

**SPIM Messages**

```
SPIM Version 3.2 of January 14, 1990
```

# A MIPS Processor




Consists of :

- an integer processing unit (CPU)
- a collection of coprocessors that perform ancillary tasks or operate on other types of data i.e, foating points
- 

SPIM simulates two coprocessors:

- Coprocessor 0 handles
  - integer operations
  - traps and exceptions
  - the virtual memory system
  - SPIM simulates most of 1-2, entirely omits memory system
- Coprocessor 1 is the foating point unit

# CPU Registers (1/2)

| Register Name | Number | Usage |
|---|---|---|
| zero | 0 | Constant 0 |
| at | 1 | Reserved for assembler |
| v0 | 2 | Expression evaluation and |
| v1 | 3 | results of a function |
| a0 | 4 | Argument 1 |
| a1 | 5 | Argument 2 |
| a2 | 6 | Argument 3 |
| a3 | 7 | Argument 4 |
| t0 | 8 | Temporary (not preserved across call) |
| t1 | 9 | Temporary (not preserved across call) |
| t2 | 10 | Temporary (not preserved across call) |
| t3 | 11 | Temporary (not preserved across call) |
| t4 | 12 | Temporary (not preserved across call) |
| t5 | 13 | Temporary (not preserved across call) |
| t6 | 14 | Temporary (not preserved across call) |
| t7 | 15 | Temporary (not preserved across call) |

# CPU Registers (2/2)

| Register Name | Number | Usage |
|---|---|---|
| s0 | 16 | Saved temporary (preserved across call) |
| s1 | 17 | Saved temporary (preserved across call) |
| s2 | 18 | Saved temporary (preserved across call) |
| s3 | 19 | Saved temporary (preserved across call) |
| s4 | 20 | Saved temporary (preserved across call) |
| s5 | 21 | Saved temporary (preserved across call) |
| s6 | 22 | Saved temporary (preserved across call) |
| s7 | 23 | Saved temporary (preserved across call) |
| t8 | 24 | Temporary (not preserved across call) |
| t9 | 25 | Temporary (not preserved across call) |
| k0 | 26 | Reserved for OS kernel |
| k1 | 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) |

# Floating Point Registers

Floating point operations can only be held by Coprocessor 1!

It can operate on single precision (32-bit) and double precision (64bit) floats

This coprocessor has its own register set $f0-$f31(again 32 bit!) which you will use for float arithmetic

Because these registers are only 32-bits wide, *two* are required to hold doubles

- For simplicity: Even single precision operations only use even-numbered registers

For passing a value from a register in one coprocessor to the other, you must use an *explicit* move operation!

- Values are moved in or out of these registers a word (32-bits) at a time

# Register Usage Conventions

- For integers:
  $a0–$a3 are used to pass the first four arguments to routines
  Remaining arguments are passed on the stack
  $v0 is used to return a value

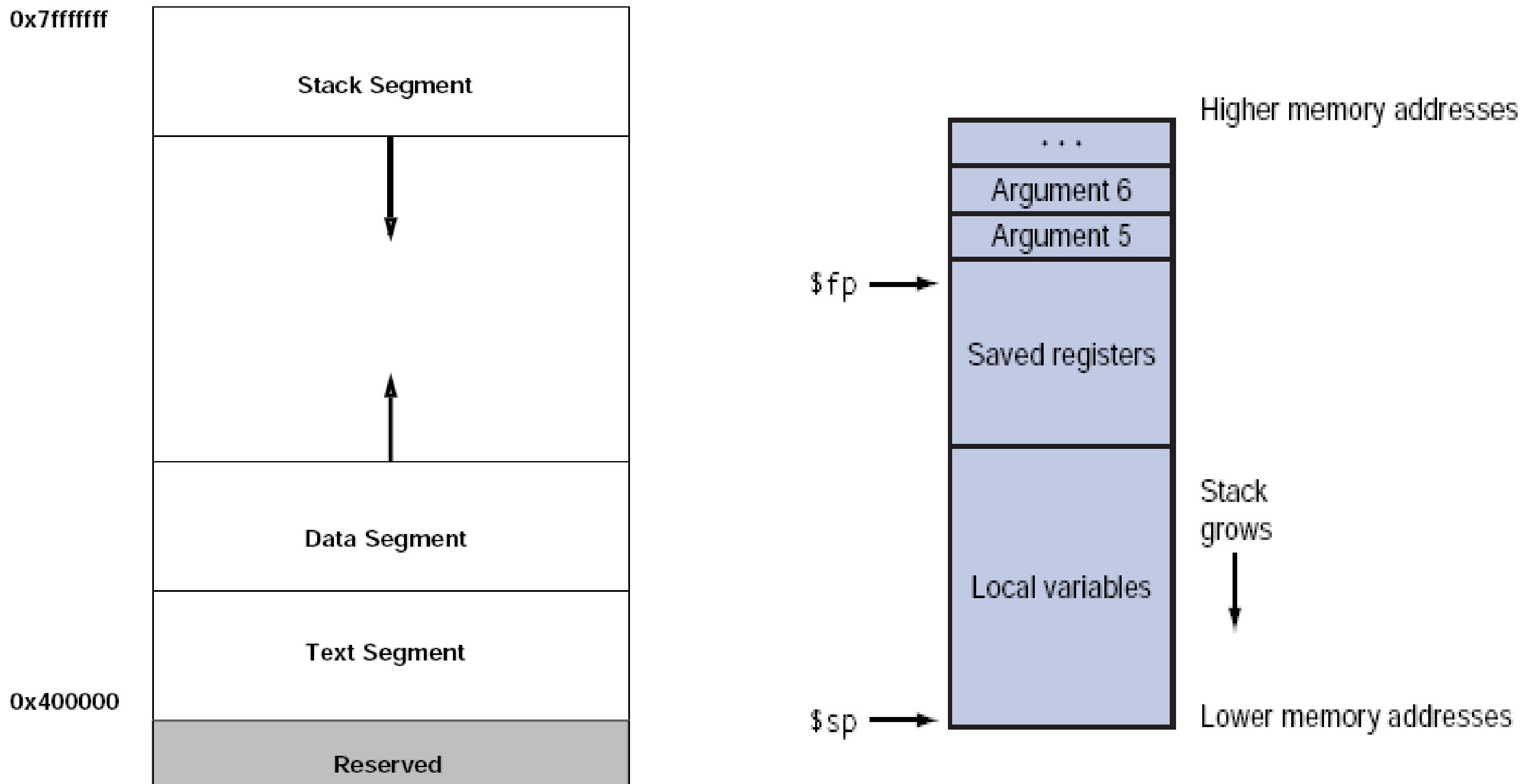
- For floats:
  Single precision arguments passed in $f12 and $f14
  Double precision arguments passed in $f12-13 pair and $f14-$f15 pair
  $f0 is used to return a float value
  Why don't we need a hard wired 0 in $f0? Bec. it's the same with $0.

- $t0–$t9 are caller-saved, $s0–$s7 are callee-saved

- $gp is a global pointer to the middle of a 64K block of memory in the static data segment (16 bit immediate vs. data segment starting from 10000000hex)

- $ra holds the return address from a procedure call, set by jal instruction

# Virtual Memory Organization




- $sp is the stack pointer which points to the last location on the stack
- $fp is the frame pointer which points to the beginning of the stack
  - i.e, you can access arguments using *immediate($fp)*

# Procedure Call Conventions

Conventions comes into play at three points during a procedure call:

- immediately before the caller invokes the callee
- just as the callee starts executing
- immediately before the callee returns to the caller

# Before a procedure calls another procedure:

It puts the arguments in standard places and then invokes the callee:

1. Pass arguments
    The first four/two arguments are passed in registers $a0–$a3 / $f12:$f13, $f14:$f15
    Remaining arguments are pushed on the stack
    The appear at the beginning of the called procedure's stack frame

2. Save caller-saved registers ($t0–$t9) and argument registers if necessary
    The called procedure can use these registers without first saving their value
    If the caller expects to use one of these registers after a call, it must save its value before the call

3. Execute a jal instruction which
    saves the return address in register $ra
    and jumps to the callee's first instruction

# Before a called procedure starts running:

It sets up its stack frame:

1. Allocate memory for the frame by setting:
   $sp = $sp – calle's stack frame size (Min. stack size allowed to be 24 bytes.)
   (i.e, pull stack pointer down)

2. Save callee-saved registers in the frame
   Must save the values in $s0–$s7, $fp, and $ra if it will alter them
   Register $fp is saved by every procedure that allocates a new stack frame
   However, $ra only needs to be saved if the callee itself makes a call

3. Establish the frame pointer by setting
   $fp =  $sp + stack frame’s size - 4

# Before a called procedure returns:

It should:

1. Place the returned value in $v0 if it is returning a value
2. Restore all callee-saved registers that were saved upon procedure entry
3. Pop the stack frame by adding the frame size to $sp
4. Return by jumping to the address in register $ra

# System Calls

SPIM provides a small set of operating-system-like services through the syscall instruction

| Service | System Call 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 | | integer (in $v0) |
| read_float | 6 | | float (in $f0) |
| read_double | 7 | | double (in $f0) |
| read_string | 8 | $a0 = buffer, $a1 = length | |
| sbrk | 9 | $a0 = amount | address (in $v0) |
| exit | 10 | | |

```
.data
str: .asciiz "the answer = "
.text

li $v0, 4 # system call code for print_str
la $a0, str # address of string to print
syscall # print the string

li $v0, 1 # system call code for print_int
li $a0, 5 # integer to print
syscall # print it
```

# Byte Order

MIPS can operate both with big-endian or little-endian byte order

SPIM's byte order mode is determined by underlying hardware running the simulator:

- little-endian for Intel, big-endian for Mac

# Addressing Modes

Load/store architecture --> only load and store instructions can access memory
--> computation instructions *entirely* on registers

Bare machine provides only one memory addressing mode: c(rx)

- the sum of the immediate c and the contents of register rx as the address

Built on top of it are the following addressing modes:

| Format | Address Computation |
|---|---|
| (register) | contents of register |
| imm | immediate |
| imm (register) | immediate + contents of register |
| symbol | address of symbol |
| symbol ± imm | address of symbol + or − immediate |
| symbol ± imm (register) | address of symbol + or − (immediate + contents of register) |

Most load and store instructions operate only on aligned data
A quantity is aligned if its memory address is a multiple of its size in bytes

- a halfword object stored at an even address
- a full word object must be stored at address 4n

# Assembler Syntax

```
.data
item: .word 1
.text
.globl main # Must be global
main: lw $t0, item
```

Labels --> declared by putting them at the beginning of a line followed by a colon
Strings --> follow the C convention

.align n
Align the next datum on a 2n byte boundary. (i.e, 2 for word)

.ascii str / .asciiz str
Store the string in memory, do not null-terminate / do null terminate

.byte b1, ..., bn / .word w1, ..., wn / .float f1, ..., fn / .double d1, ..., dn
Store the n bytes / words / single precision floats / double precision floats in successive locations of memory.

# Assembler Syntax (2/2)

.data <addr> / .text <addr>
The following data items should be stored in the data / text segment
optional addr --> the items are stored beginning at address addr

.globl sym
Declare that symbol sym is global and can be referenced from other files.
main label *must* be declared global!

.space n
Allocate n bytes of space in the current segment (which must be the data segment in SPIM)

# Instructions

add Rdest, Rsrc1, Src2R
abs Rdest, Rsrc
move Rdest, Rsrcs
mfhi RdestR / mthi RdestR
mflo RdestR / mtlo RdestR
bgt Rsrc1, Src2, labelS
la Rdest, address #load computed address, not *content*, into Rdest
lw Rdest, address

# Floating Point Instructions

Registers $f0-$f31

lwcz Rdest, address (z --> coprocessor #)
swcz Rsrc, address
mtcz Rsrc, Cpdest
mfcz Rdest, Cpsrc

l.d FRdest, address / l.s FRdest, address
s.d FRdest, address / s.s FRdest, address

add.d FRdest, FRsrc1, Frsrc2F
add.s FRdest, FRsrc1, FRsrc2F