Introduction to SPIM

- RISC (MIPS 2000) architecture
- SPIM instructions
- Examples

Target Architecture

- **1960's - stack architectures, Burroughs 5000**
- **1970's - general purpose register architectures**
  - **Complex instruction sets, VAX, IBM 360**
  - **Temporary storage in registers**
  - **Explicit operands in all instructions**
  - **Result destination is register or memory**
- **1980's - better optimizing compilers, back to simpler architectures, RISC**

RISC

- MIPS R2000, about 1986, uses PC relative addressing so code can run independent of where it is loaded
- Hennessey and Patterson studied this in Ch4 as DLX generic architecture
  - Empirical study (gcc, Spice, TeX, US Steel)
    - ~75% references to registers
    - Conditional branch, add, load, store dominate
- Related to Intel i860, Motorola 88000, Sparc

DLX Machine

- **Simple load/store instruction set**
- **Designed for pipelining**
- **Easily decoded instructions**
- **32 - 32bit registers**
- **R0 always 0 for comparisons**
- **ALU instructions are register to register**
- **Jumps to PC or address in a register**
- **Can compare instructions between 2 registers, set a destination register to 0 or 1**

Instruction Types

- **Register to register instructions**
  - **Fixed length instruction encoding**
  - **Simple code generation**
  - **Leads to more instructions than other styles**
- **Register to memory**
  - **Easy to encode**
  - **Operands not equivalent because first is overwritten**

Instruction Types, cont.

- **Memory to memory**
  - **Most compact**
  - **Large variation in instruction size and work per instruction**
  - **Possible memory bottleneck**
- **Memory is byte addressable**
  - **SPIM is big-endian [0,1,2,3] on our SPARC Hw**

Kinds of Instructions

• **Arithmetic/logical**	**integer add**
• **Data transfer**	**load, store**
• **Control**	**jump**
• **System**	**O/S call (e.g. print)**
• **Floating point**	**add**
• **String**	**search, compare**

SPIM Instructions

- **Only need the integer instructions**

  <instruct> Rdest, Rsrc1, Src2 **where *Src2* is immediate or a register (addu, subu, divu, multu)**

- **Load/store**

la Rdest, address	**load address into register *Rdest***
lw Rdest, address	**load value at address into *Rdest***
sw Rsrc, address	**store word from register *Rsrc* into address**

- **Move**

move Rdest, Rsrc	**move contents of *Rsrc* into *Rdest***

SPIM Instructions

- **Control**

  b label **branch to label**

  beqz Rsrc, label **branch to label if value in Rsrc equals 0**

  bgeu Rsrc1, Src2, label **branch to label if contents of register *Rsrc1* is greater than or equal to *Src2***

  similarly *bgtu, bleu, bltu*

  j label **unconditionally jump to label**

  jal label **unconditionally jump to labe and save address of next instruction in $ra**

SPIM Instructions

- **System**

  syscall **register $v0 contains number of system call**

print_int	**1**	**$a0 has integer**
print_string	**4**	**$a0 has string**
read_int	**5**	**integer result in $v0**
read_string	**8**	**$a0 buffer, $a1 length**

- **Pseudoinstructions expand into several machine instructions**

Addressing Modes in SPIM

- **Memory indirect (register) : *Add R1, (R3)* where memory address is in register R3**
- **Immediate imm: *Add R4, 3***
- **Based imm(register) : *Add R4,100(R1)*   where fetch from memory location Mem[100+ contents of register R1]**
- **Register: *Add R1, R2***
- **Symbolic symbol: *X***
- **Symbolic symbol +/- imm: *X* + *3***
- **Symbolic symbol+/- imm(register): *X - 4 (R1)***

Target Machine

- **Has difficult target instruction set**
  - **Delayed branches and loads (2 cycles)**
  - **Restricted addressing modes**
- **Pipelining - exploiting parallelism among the instructions (overlapping execution)**

instruction execution(in machine cycles)

fetch - decode - effective address - memory access - write back

- **Can overlap execution of 2 instructions when possible**

More SPIM

- **Data layout directives: *asciiz <string>***
- **Data segment *.data***
- **Instruction segment *.text***
- **Command line option *-file* specifies input file**
- **Debugging commands are available (step, continue, prints, breakpoint settings, (see manual ppA-44 ff**

“Hello World” in SPIM

```
# This is a simple program to print hello world
# a comment starts with a # till the end of the line
   .data                # start putting stuff in the data segment
greet: .asciiz "Hello world\n” # declare greet to be the string
   .text                #start putting stuff in the text segment
main:                   # main is a label here. Names a function
   li $v0, 4            # system call code for print_str(sect1.5)
   la $a0, greet        # address of string to print
   syscall              # print the string
   jr $ra               # return from main

# here li is load immediate into an integer register
#       la is load computed address into an integer register
#       jr is standard return from function call...$ra contains
#      return address
```

Summation in SPIM

```
# this program adds the numbers 1 to 10
   .text
main:
   move $t0, $zero # initialize sum (t0) to 0
   move $t1, $zero # initialize counter (t1) to 0
loop:
   addi        $t1, $t1, 1  # increment counter by 1
   add         $t0, $t0, $t1 # add counter to sum
   blt         $t1, 10, loop # if counter < 10, goto loop

# this is outside the loop
# code to print the sum
   li $v0, 1          # system call for print_int
   move $a0, $t0      # the sum to print
   syscall            # print the sum
```

Summation, cont

```
                    # code to print a newline
   li $v0, 4        # system call for print_str
   la $a0, newline  # address of str to print
   syscall          # wind up the program
   jr $ra           # return from main
   .data
newline:
   .asciiz "\n"     # declare the string newline
                    # note, the decl follows the use.
                    # it may also be within the code as
                    # long as we toggle between .text
                    # and .data correctly
```

Longer Example

- Factorial program we have seen before augmented with debugging prints
- Also shows context switching in SPIM, use of temporary registers to store info over calls

Factorial

1.

```
#this is the factorial program
#main(){printf("the factorial of 10 is %d\n"),fact(10));}
#int fact (int n)
# {if (n<1) return (1)
# else return (n*fact(n-1));
   .text
   .globl main
main:
   subu $sp,$sp,32    #stack frame is 32 bytes long
   sw $ra,20($sp)     #save return address in $ra
   sw $fp,16($sp)     #save old frame pointer
   addu $fp,$sp,28    #setup frame pointer
                      #main calls fact
   li $a0,4           #put arg in $a0
   move $s1,$a0       #save chosen n value for output in $s1
   jal fact           #call factorial function
```

2.

```
#BGR using syscalls for debugging
   move $s0,$v0        #save return value in $s0
   li $v0,4            #syscall to print a string
   la $a0,str          #put address of string str in $a0
   syscall             #prints the string str
   li $v0,1            #syscall to print an integer
   move $a0,$s1        #print out chosen n value
   syscall             #prints n value
   li $v0,4            #syscall to print a string
   la $a0,str2         #put address of string str2 in $a0
   syscall             #prints the string str2
   li $v0,1            #syscall to print an integer
   move $a0,$s0        #move return value into $a0
   syscall             #prints the integer return value
   li $v0,4            #syscall to print a string
   la $a0,new          #put address of string new in $a0
   syscall             #prints the string new
#BGR end debugging syscalls
      lw $ra,20($sp)   #epilogue to exit
      lw $fp,16($sp)   #restore frame pointer
      addu $sp,$sp,32  #reset stack pointer
      jr $ra           #return to caller
```

3.

```
      .rdata
    str2: .asciiz " equals ”
    str: .asciiz "The factorial for argument ”
    new: .asciiz "\n”
   .text
fact: subu $sp,$sp,32         #stack frame is 32 bytes
   sw $ra,20($sp)             #save return address
   sw $fp,16($sp)             #save old frame pointer
   addu $fp,$sp,28            #setup frame pointer
   sw $a0,0($fp)              #save argument - n
   lw $v0,0($fp)              #load n
   bgtz $v0,$L2               #branch if n>0
   li $v0,1                   #return 1
   j $L1                      #jump to code to return(base case)
$L2: lw $v1,0($fp)            #load n
   subu $v0,$v1,1             #compute n-1
   move $a0,$v0               #move value to $a0
   jal fact                   #call recursively
   lw $v1,0($fp)              #load n
   mul $v0,$v0,$v1            #compute n*fact(n-1) in $v0
```

```
#BGR more debugging                                      4.
   move $s0,$v0       #save return value
   li $v0,1           #syscall to print an integer
   move $a0,$s0       #move return value into $a0
   syscall            #prints the integer return value
   li $v0,4
   la $a0,new         #prints newline
   syscall
   move $v0,$s0       #restore return value into $v0
#BGR end debugging code
$L1: lw $ra,20($sp) #restore $ra
   lw $fp,16($sp)     #return $fp
   addu $sp,$sp,32    #pop stack
   j $ra              #return to caller
```

Factorial Output

```
106 remus!spim> spim -file fact.s
SPIM Version 6.0 of July 21, 1997
Copyright 1990-1997 by James R. Larus (larus@cs.wisc.edu).
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /usr/local/lib/spim/trap.handler
1
2
6
24
The factorial for argument 4 equals 24
```