CS323 Lab 15

Yepang Liu

liuyp1@sustech.edu.cn

Outline

- Introduction to Project Phase 4

MIPS32 Assembly Program Basics

- Typical program layout
  - SPIM simulator accepts a textual assembly file that ends with the .s or .asm extension and simulates its execution

```
        .text           #code section
        .globl main     #starting point: must be global
main:
        # user program code
        .data           #data section
        # user program data
```

Top-level directives:

- .text: code segement
- .data: data segment
- .globl sym: the symbol sym is global and can be referenced from other files

Data Types and Definitions

- Programmers can declare constants or global variables in data segments: `name: storage_type value(s)`

name: a label that locates the mem addr of the var
storage_type: data type
value: initial value

storage_type	description
`.ascii str`	store string `str` in memory, without null-terminate
`.asciiz str`	store string `str` in memory, with null-terminate
`.byte b1,b2,...,bn`	store `n` 8-bit values in successive bytes of memory
`.half h1,h2,...,hn`	store `n` 16-bit quantities in successive memory halfwords
`.word w1,w2,...,wn`	store `n` 32-bit quantities in successive memory words
`.space n`	allocate `n` bytes of space in the data segment

Example

```
#sample example 'add two numbers'

.text                       # text section
.globl main                 # call main by SPIM

main:   la $t0, value       # load address 'value' into $t0
        lw $t1, 0($t0)      # load word 0(value) into $t1
        lw $t2, 4($t0)      # load word 4(value) into $t2
        add $t3, $t1, $t2   # add two numbers into $t3
        sw $t3, 8($t0)      # store word $t3 into 8($t0)

.data                       # data section
value:  .word 10, 20, 0     # data for addition
```

Registers

Reserved for assembler and OS.

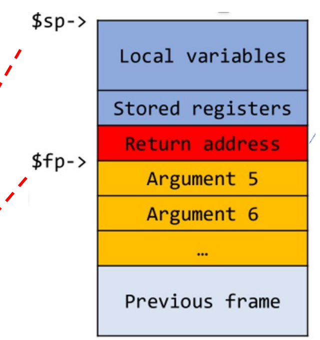
Cannot be used by user programs.

SPIM will raise syntax errors when user programs use them.

Point to the middle of a 64K block of memory in the static data segement

Remaining arugements can be passed using stack

Register Number	Mnemonic Name	Conventional Use		Register Number	Mnemonic Name	Conventional Use
$0	$zero	Permanently 0		$24, $25	$t8, $t9	Temporary
$1	$at	Assembler Temporary		$26, $27	$k0, $k1	Kernel
$2, $3	$v0, $v1	Value returned by a subroutine		$28	$gp	Global Pointer
$4-$7	$a0-$a3	Subroutine Arguments		$29	$sp	Stack Pointer
$8-$15	$t0-$t7	Temporary		$30	$fp	Frame Pointer
$16-$23	$s0-$s7	Saved registers		$31	$ra	Return Address



$t0-$t9: caller-saved registers (temporaries)

$s0-$s7: callee-saved registers (long-lived values)

Instruction Format

- Each instruction supported by SPIM is of the following general format:

```
Label: OpCode, Operand1, Operand2, Operand3
```

Checkout this webpage to see supported instructions:

https://cgi.cse.unsw.edu.au/~cs1521/17s2/docs/spim.php

Operands

The operands can be:

Operand	Description
R_n	a register; R_s and R_t are sources, and R_d is a destination
Imm	a constant value; a literal constant in decimal or hexadecimal format
Label	a symbolic name which is associated with a memory address
Addr	a memory address, in one of the formats described below

Format	Address
Label	the address associated with the label
(R_n)	the value stored in register R_n (indirect address)
Imm(R_n)	the sum of Imm and the value stored in register R_n
Label(R_n)	the sum of Label's address and the value stored in register R_n
Label ± Imm	the sum of Label's address and Imm
Label ± Imm(R_n)	the sum of Label's address, Imm and the value stored in register R_n

Addressing modes supported by SPIM

Instruction Mapping Between TAC & MIPS

three-address-code	MIPS32 instruction
x := #k	li reg(x), k
x := y	move reg(x), reg(y)
x := y + #k	addi reg(x), reg(y), k
x := y + z	add reg(x), reg(y), reg(z)
x := y - #k	addi reg(x), reg(y), -k
x := y - z	sub reg(x), reg(y), reg(z)
x := y * z	mul reg(x), reg(y), reg(z)
x := y / z	div reg(y), reg(z) mflo reg(x)
x := *y	lw reg(x), 0(reg(y))
*x := y	sw reg(y), 0(reg(x))
GOTO x	j x
x := CALL f	jal f move reg(x), $v0
RETURN x	move $v0, reg(x) jr $ra
IF x <y GOTO z	blt reg(x), reg(y), z
IF x <= y GOTO z	ble reg(x), reg(y), z
IF x >y GOTO z	bgt reg(x), reg(y), z
IF x >= y GOTO z	bge reg(x), reg(y), z
IF x != y GOTO z	bne reg(x), reg(y), z
IF x == y GOTO z	beq reg(x), reg(y), z

Integer multiplication and division in mips: https://www.cim.mcgill.ca/~langer/273/12-notes.pdf

Register Allocation

- You are suggested to implement the following simple local register allocation strategy
  - Registers are only allocated inside basic blocks
  - When entering a basic block, all registers are labeled as idle. If there is a variable that should be loaded into a register, do the following:
    - If there is an idle register, use it.
    - If no registers are idle, pick a register and spill its content to the memory. It is preferable to choose a register, whose content is not going to be accessed in the near future or inside the basic block.
  - When exiting a basic block, all allocated registers' values should be saved into memory

- Global register allocation is challenging as it requires inter-block analysis to infer the liveness of variables

Procedure Calls

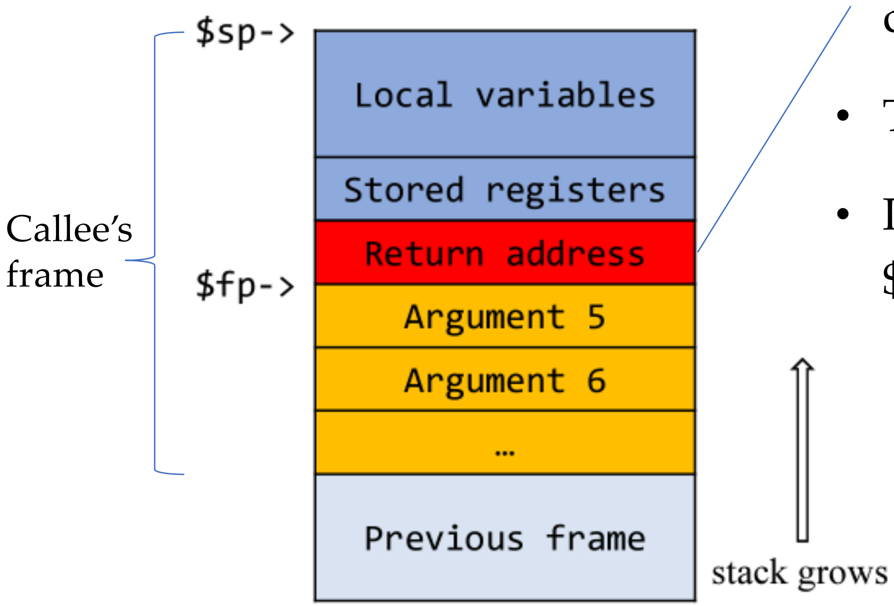


Figure 1: A typical stack frame layout

Context saving:

- The return address of the caller is stored in the callee's frame.
- The return address of callee is stored in register $ra.
- Register values: callee is responsible for saving $s0-$s8 while caller is responsible for saving $t0-$t9.

Checkout project notes for the templates of calling/return sequences.

System Calls

Service	Code	Arguments	Result
print_int	1	`$a0` = integer	
print_float	2	`$f12` = float	
print_double	3	`$f12` = double	
print_string	4	`$a0` = char *	
read_int	5		integer in `$v0`
read_float	6		float in `$v0`
read_double	7		double in `$v0`
read_string	8	`$a0` = buffer, `$a1` = length	string in buffer
sbrk	9	`$a0` = # bytes	extend data segment
exit	10		program exits
print_char	11	`$a0` = char	

```
.data
_prompt: .asciiz "Enter an integer:"
_ret: .asciiz "\n"
.globl main
.text
read:
    li $v0, 4
    la $a0, _prompt
    syscall
    li $v0, 5
    syscall
    jr $ra
```

Perform syscall print_string (code: 4) to print the prompt message

Perform syscall read_int (code: 5) to read in an integer

See more at https://cgi.cse.unsw.edu.au/~cs1521/17s2/docs/spim.php

Project Final Check

- Jan 6 & 7 morning (9:30 am to 11:30 am) at Room 650A CoE Building (South)

- Please select a time before Dec 31: https://wj.qq.com/edit/v2.html?sid=16976409

- Required task: generation of three-address code

- Optional task: generation of MIPS32 code