南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

CS323 Lab 14

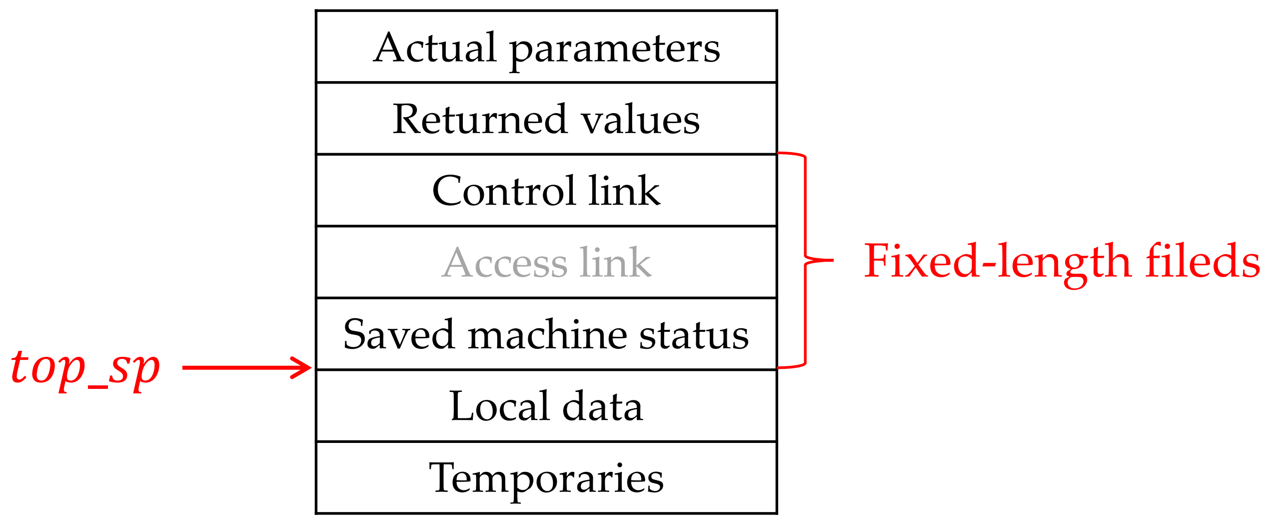
Yepang Liu
liuyp1@sustech.edu.cn

Outline

- Project Phase 3 Tutorial
- Introduction to Project Phase 4

Function Invocation

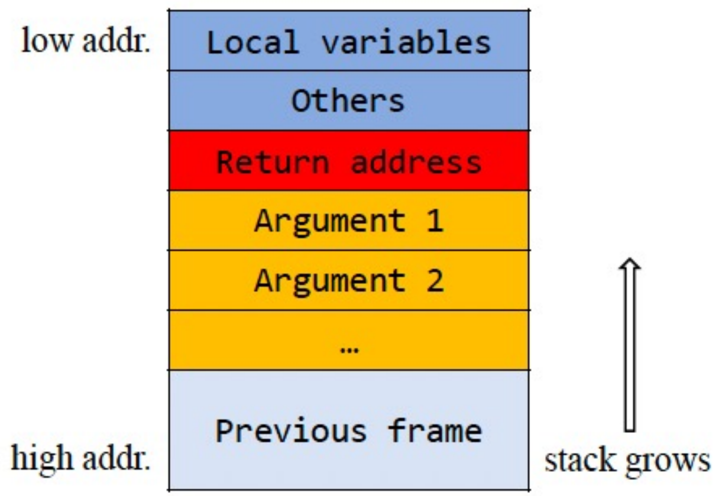
- Each active function has its own activation record, which stores the key information related to the function invocation
  - Actual parameters, local variables, saved register values, return address, etc.



Theoretical model

Function Invocation

- In most architectures, the activation records are managed using a stack. For this reason, activation records are often called stack frames.



A runtime stack of a Linux-x86 process

Main Tasks in Project Phase 3

- We do not need to manage the stack frames, which is machine-dependent (our IR simulator will do the job)
- What we need to do mainly includes the following steps:
  - Prepare the arguments
  - Pass arguments using the `ARG` instruction (e.g., `ARG x`)
  - Invoke the function using the `CALL` instruction (e.g., `x := CALL f`)

Argument Passing

C++ supports both approaches:

- `void foo(type arg)`, `arg` is passed by value regardless of whether type is a simple type, a pointer type or a class type
- `void foo(type& arg)`, `arg` is passed by reference

- **Two main approaches:**
  - **Pass by value**: A copy of the actual argument's value is made in memory, i.e., the caller and callee have two independent copies. If the callee modifies the parameter variable, the effect is not visible to the caller.
    - Typical languages: C, Java
  - **Pass by reference (a.k.a., pass by address)**: Pass the reference of the actual argument in the caller to the corresponding formal parameter of the callee so that the parameter variable becomes an alias of the argument variable (it cannot be alias of other variables, which is different from pass-by-value for reference types in Java). If the callee modifies the parameter variable, the effect is visible to the caller.
    - Languages: C++ (using the `&` operator, see above example), C# (using the `ref` keyword), etc.

Argument Passing in SPL

- For primitive types, arguments are passed by value
  - The callee's stack frame will contain copies of these values

- For derived types, your compiler should make sure that the callee gets the starting address of each argument (like Java's treatment*)
  - To pass a `struct` variable `s1` as an argument to a called function, we should push the argument onto stack using `ARG &S1` rather than `ARG S1`

* In C, we will explicitly pass a struct pointer to avoid copying the whole structure.

Translation Schemes

- First, we should add two pre-defined functions that simulate I/O to the symbol table
  - `read`: takes no parameter and returns an integer value
  - `write`: accepts an integer argument and outputs it

translate_Exp(Exp, place) = case Exp of	
read LP RP	return [READ place]
write LP Exp RP	tp = new_place() return translate_Exp(Exp, tp) + [WRITE tp]

Translated into `read` and `write` instructions

* place is the address to store the evaluation result of the expression

Translation Schemes

- Invoking functions without parameters

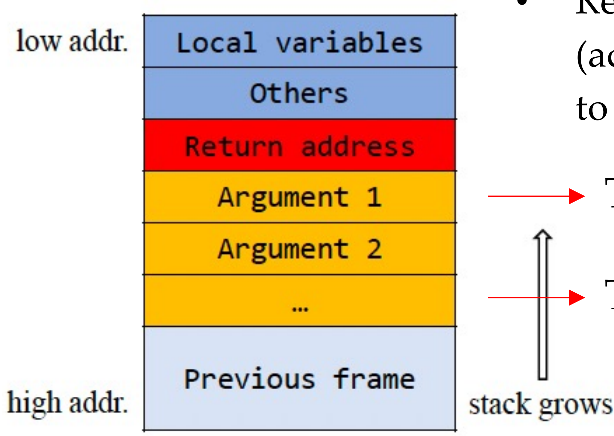
translate_Exp(Exp, place) = case Exp of	
ID LP RP	function = symtab_lookup(ID) return [place := CALL function.name]

Translation Schemes

- Invoking functions with parameters

Arguments should be pused in reverse order of declaration

- Reverse-order pushing makes the argument fetching easier (adding offsets to the stack top pointer, which typically points to the end of fixed-length fields)



The last pushed argument (using the `ARG` instruction)

The first pushed argument (using the `ARG` instruction)

Translation Scheme

translate_Exp(Exp, place) = case Exp of	
ID LP Args RP	function = symtab_lookup(ID) arg_list = EMPTY_LIST 1: Create an empty list to hold arguments code1 = translate_Args(Args, arg_list) code2 = EMPTY_CODE for i = 1 to arg_list.length: 3: Traverse the list and generate ARG instructions code2 = code2 + [ARG arg_list[i]] return code1 + code2 + [place := CALL function.name] 4: Generate CALL instruction
translate_Args(Args, arg_list) = case Args of	
Single parameter: Exp	tp = new_place() code = translate_Exp(Exp, tp) arg_list = tp + arg_list 2: Adding each argument to the list head return code
Multiple parameters: Exp COMMA Args	tp = new_place() code1 = translate_Exp(Exp, tp) arg_list = tp + arg_list code2 = translate_Args(Args, arg_list) Handling the remaining parameters return code1 + code2

Example

```
int fact(int n)
{
    if (n == 1)
        return n;
    else
        return (n * fact(n - 1));
}
int main()
{
    int m, result;
    m = read();
    if (m > 1)
        result = fact(m);
    else
        result = 1;
    write(result);
    return 0;
}
```

```
FUNCTION fact :
PARAM v1
IF v1 == #1 GOTO label1
GOTO label2
LABEL label1 :
RETURN v1
LABEL label2 :
t1 := v1 - #1
ARG t1
t2 := CALL fact
t3 := v1 * t2
RETURN t3

FUNCTION main :
READ t4
v2 := t4
IF v2 > #1 GOTO label3
GOTO label4
LABEL label3 :
ARG v2
t5 := CALL fact
v3 := t5
GOTO label5
LABEL label4 :
v3 := #1
LABEL label5 :
WRITE v3
RETURN #0
```

Outline

- Project Phase 3 Tutorial
- 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.

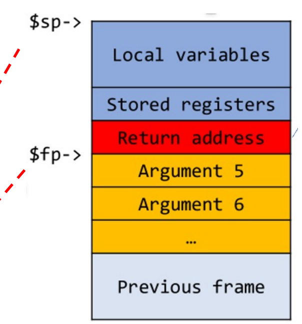
Cannot be used by user programs.

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`

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

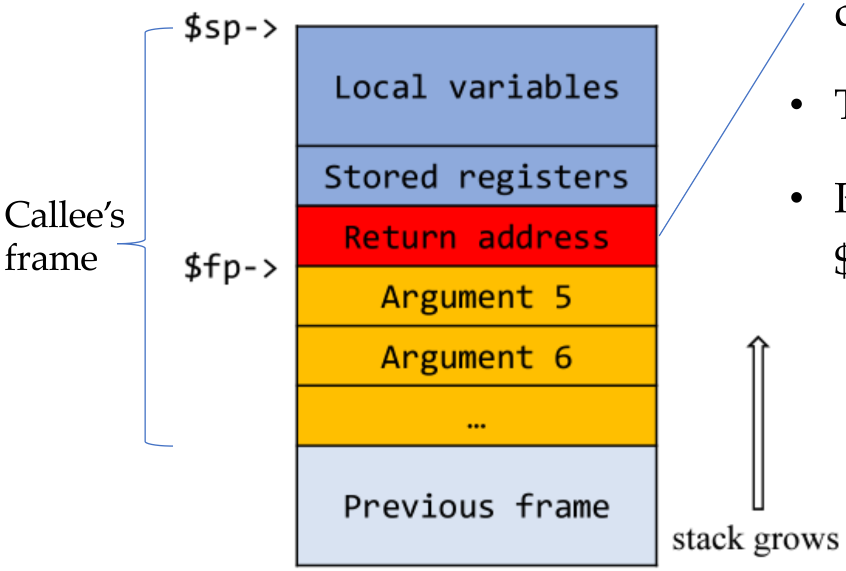


Figure 1: A typical stack frame layout

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

Deadline & Grading

- Deadline: 10:00 PM, Jan 14, 2024 (no grace period)
- Passing all our three test cases will get 80 points; The remaining 20 points depend on your design and implementation (you should submit a report to illustrate your algorithms)
- If your score is above 60, the extra points (your score – 60) can be used to redeem the points you lose for projects/labs