KIẾN TRÚC MÁY TÍNH

KHOA HỌC & KỸ THUẬT MÁY TÍNH

Võ Tấn Phương

http://www.cse.hcmut.edu.vn/~vtphuong

Chapter 3.2

Hợp ngữ MIPS

(Assembly Language)

Nội dung

- Các phát biểu trong hợp ngữ MIPS
- Khung dạng chương trình hợp ngữ MIPS
- Định nghĩa/khai báo dữ liệu
- Địa chỉ bắt đầu (alignment) và thứ tự các byte trong bộ nhớ
- Các hàm hệ thống
- Thủ tục/hàm
- Truyền tham số và Runtime Stack

Các phát biểu trong hợp ngữ MIPS

- Có 3 loại phát biểu trong hợp ngữ MIPS
  - Bình thường một phát biểu là một dòng

1. Các lệnh thật
   - Là các lệnh trong tập lệnh của bộ xử lý MIPS
   - Tương ứng một lệnh dạng mã máy (số 32 bit)
2. Lệnh giả (Pseudo-Instructions) và Macros
   - Được chuyển sang các lệnh thật bởi assembler
   - Mục đích giảm công sức cho lập trình viên
3. Các chỉ thị (directive) của Assembler
   - Cung cấp thông tin để assembler dịch chương trình
   - Dùng để định nghĩa phân đoạn, cấp phát dữ liệu bộ nhớ
   - Không thực thi được: chỉ thị không phải là lệnh

Phát biểu loại lệnh

- Lệnh hợp ngữ có định dạng:

```
[label:]    mnemonic    [operands]    [#comment]
```

- Label: (optional)
  - Đánh dấu vị trí gợi nhớ của lệnh, phải có dấu '**:**'
  - Nhãn thường xuất hiện trong phân đoạn dữ liệu và mã
- Mnemonic
  - Xác định phép toán (vd: **add**, **sub**, vv.)
- Operands
  - Xác định toán hạn nguồn, đích của phép toán
  - Toán hạn có thể là thanh ghi, ô nhớ, hằng số
  - Thông thường một lệnh có 3 toán hạn

```
L1:    addiu $t0, $t0, 1          #increment $t0
```

Chú thích (Comments)

- Chú thích rất quan trọng!
  - Giải thích mục đích của chương trình
  - Giúp việc đọc hiểu chương trình dễ dàng khi viết và xem lại bởi chính mình và người khác
  - Giải thích dữ liệu vào, ra
  - Chú thích cần thiết ở đầu mỗi thủ tục/hàm
    - Chỉ ra đối số, kết quả của thủ tục/hàm
    - Mô tả mục đích của thủ tục/hàm
- Chú thích trên một dòng
  - Bắt đầu với ký tự #

Tiếp theo …

- Các phát biểu trong hợp ngữ MIPS
- Khung dạng chương trình hợp ngữ MIPS
- Định nghĩa/khai báo dữ liệu
- Địa chỉ bắt đầu (alignment) và thứ tự các byte trong bộ nhớ
- Các hàm hệ thống
- Thủ tục/hàm
- Truyền tham số và Runtime Stack

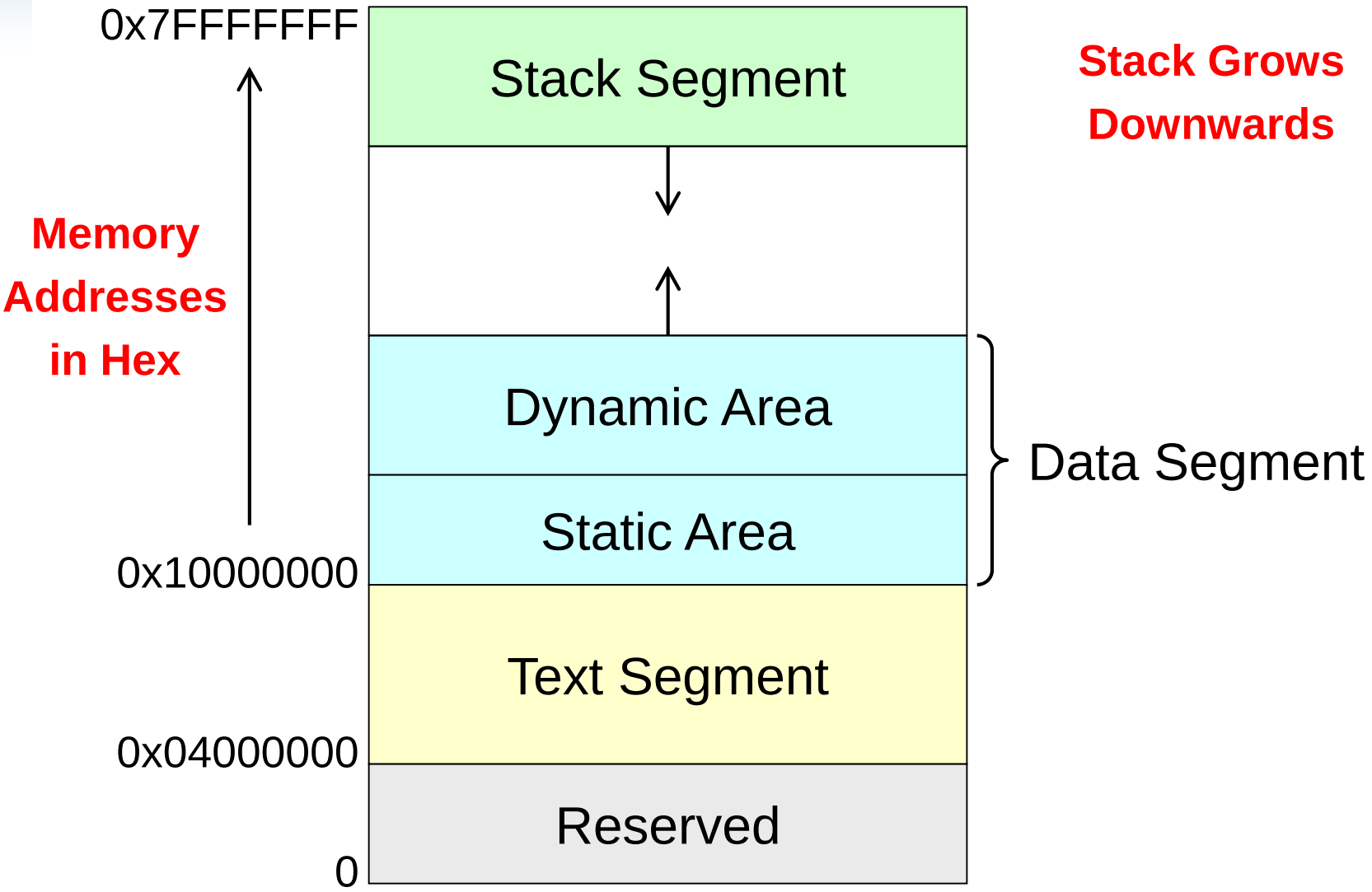
Khung dạng mẫu của chương trình hợp ngữ

```
# Title:                                    Filename:
# Author:                                   Date:
# Description:
# Input:
# Output:
################## Data segment ######################
.data
  . . .
################## Code segment ######################
.text
.globl main
main:                                       # main program entry
  . . .
li $v0, 10                                  # Exit program
syscall
```

Chỉ thị .DATA, .TEXT, & .GLOBL

- **Chỉ thị .DATA**
  - Định nghĩa phân đoạn dữ liệu (data segment)
  - Các biến của chương trình được định nghĩa tại vùng này
  - Assembler sẽ cấp phát và khởi tạo các biến này
- **Chỉ thị .TEXT**
  - Định nghĩa phân đọa mã (code segment) của một chương trình và chứa các lệnh
- **Chỉ thị .GLOBL**
  - Khai báo ký hiệu toàn cục (global)
  - Các ký hiệu toàn cục có thể kham khảo ở các file khác nhau
  - Ký hiệu hàm *main* dùng chỉ thị toàn cục

Phân chia phân đoạn của chương trình



Tiếp theo …

- Các phát biểu trong hợp ngữ MIPS
- Khung dạng chương trình hợp ngữ MIPS
- Định nghĩa/khai báo dữ liệu
- Địa chỉ bắt đầu (alignment) và thứ tự các byte trong bộ nhớ
- Các hàm hệ thống
- Thủ tục/hàm
- Truyền tham số và Runtime Stack

Phát biểu khai báo dữ liệu

- Nằm trong phân đọa dữ liệu .DATA
- Đánh dấu ô nhớ tương ứng với tên và dữ liệu khởi tạo
- Cú pháp:

[*name:*] *directive* *initializer* [, *initializer*] . . .

- Các giá trị khởi tạo chuyển thành dạng dữ liệu nhị phân trong vùn nhớ dữ liệu tương ứng

Các chỉ thị kiểu dữ liệu (Data Directives)

- **.BYTE**
  - Mỗi giá trị là 1 ô nhớ (8-bit, 1 byte)
- **.HALF**
  - Mỗi giá trị là 2 ô nhớ (16-bit, 2 byte), có địa chỉ bắt đầu chi hết cho 2 (half align)
- **.WORD**
  - Mỗi giá trị là 4 ô nhớ (32-bit, 4 byte), có địa chỉ bắt đầu chi hết cho 4 (word align)
- **.FLOAT**
  - Mỗi giá trị là 4 ô nhớ số thực dấu chấm động đơn
- **.DOUBLE**
  - Mỗi giá trị là 8 ô nhớ số thực dấu chấm động kép

Các chỉ thị về chuỗi (String Directives)

- **.ASCII**
  - Cấp phát các ô nhớ 1 byte cho chuỗi ASCII
- **.ASCIIZ**
  - Giống với chỉ thị **.ASCII**, nhưng thêm ký tự NULL tại vị trí kết thúc chuỗi
  - Ký tụ NULL có giá trị bằng 0, đánh dấu kết thúc chuỗi
- **.SPACE**
  - Cấp phát *n* ô nhớ 1 byte không khởi tạo giá trị trong vùng nhớ dữ liệu

Ví dụ khai báo biến

```
.DATA
var1:   .BYTE     'A', 'E', 127, -1, '\n'
var2:   .HALF     -10, 0xffff
var3:   .WORD     0x12345678:100
var4:   .FLOAT    12.3, -0.1
var5:   .DOUBLE   1.5e-10
str1:   .ASCII    "A String\n"
str2:   .ASCIIZ   "NULL Terminated String"
array:  .SPACE    100
```

Array of 100 words

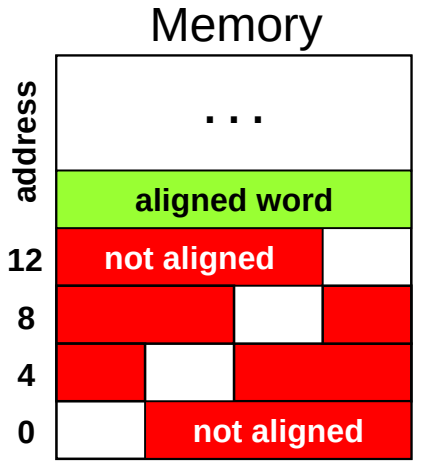
100 bytes (not initialized)

Tiếp theo …

- Các phát biểu trong hợp ngữ MIPS
- Khung dạng chương trình hợp ngữ MIPS
- Định nghĩa/khai báo dữ liệu
- Địa chỉ bắt đầu (alignment) và thứ tự các byte trong bộ nhớ
- Các hàm hệ thống
- Thủ tục/hàm
- Truyền tham số và Runtime Stack

Địa chỉ bắt đầu (alignment) trong bộ nhớ

- Bộ nhớ được xem là mảng các ô nhớ 1 byte
  - Định địa chỉ theo byte: mỗi địa chỉ tương ứng ô nhớ 1 byte
- Từ nhớ (word) chiếm 4 byte liên tiếp trong bộ nhớ
  - Mỗi lệnh MIPS là một số nhị phân 4 byte
- Alignment: địa chỉ bắt đầu phải chia hết cho kích thước
  - Địa chỉ một word là một số chia hết cho **4**
    - Hai bit thấp của địa chỉ là **00**
  - Địa chỉ một half word chia hết cho **2**



- **Chỉ thị `.ALIGN n`**
  - Quy định địa chỉ bắt đầu của biến khai báo kế tiếp có địa chỉ bắt đầu là một số chia hết cho 2^n

Bảng ký hiệu (Symbol Table)

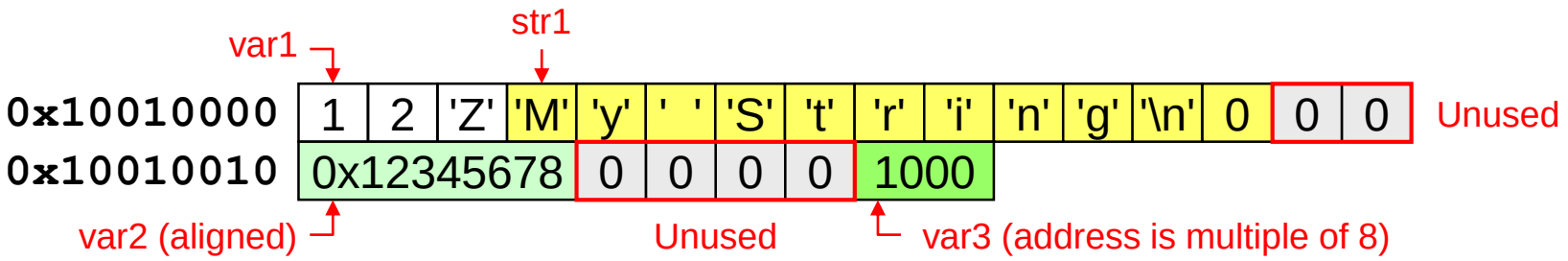
- Assembler tạo bảng ký hiệu cho các biến (label)
  - Assembler tính toán địa chỉ của các biến trong vùng nhớ dữ liệu và lưu vào bảng ký hiệu

- Ví dụ

```
.DATA
var1:   .BYTE   1, 2,'Z'
str1:   .ASCIIZ "My String\n"
var2:   .WORD   0x12345678
.ALIGN  3
var3:   .HALF   1000
```

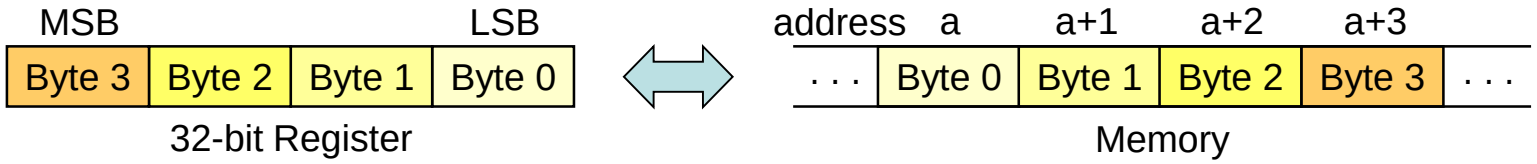
Symbol Table

Label	Address
var1	0x10010000
str1	0x10010003
var2	0x10010010
var3	0x10010018

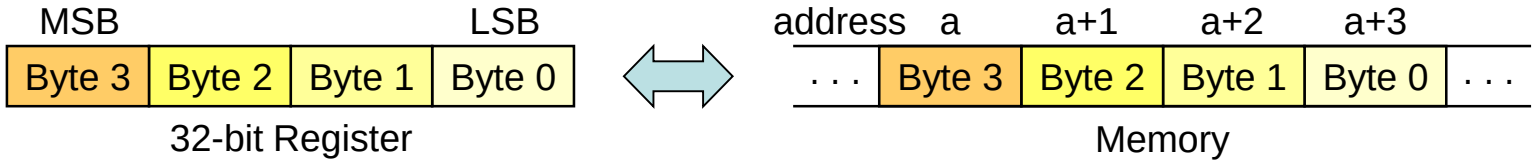


Thứ tự trọng số các ô nhớ (Endianness)

- Bộ xử lý xác định trọng số các ô nhớ trong một word theo:
- Little Endian
  - Địa chỉ bắt đầu = địa chỉ của **byte trọng số nhỏ LSB**
  - Ví dụ: Intel IA-32, Alpha



- Big Endian
  - Địa chỉ bắt đầu = địa chỉ của **byte trọng số lớn MSB**
  - Ví dụ: SPARC, PA-RISC



- MIPS hỗ trợ cả hai dạng định thứ tự byte trên

Tiếp theo …

- Các phát biểu trong hợp ngữ MIPS
- Khung dạng chương trình hợp ngữ MIPS
- Định nghĩa/khai báo dữ liệu
- Địa chỉ bắt đầu (alignment) và thứ tự các byte trong bộ nhớ
- Các hàm hệ thống
- Thủ tục/hàm
- Truyền tham số và Runtime Stack

Các hàm hệ thống (System Calls)

- Chương trình thực hiện việc xuất/nhập thông qua các hàm hệ thống
- Hợp ngữ MIPS cung cấp lệnh **`syscall`**
  - Để gọi một dịch vụ từ hệ điều hành
  - SPIM và MARS hỗ trợ tương đối đầy đủ các hàm hệ thống
- Các sử dụng hàm **`syscall`** để gọi một dịch vụ
  - Gán mã số dịch vụ vào $v0
  - Gán giá trị các tham số (nếu có) vào các thanh ghi $a0, $a1, $a2 vv…
  - Gọi **`syscall`**
  - Lấy kết quả (nếu có) từ các thanh ghi kết quả

Các dịch vụ của Syscall

Service	$v0	Arguments / Result
Print Integer	1	$a0 = integer value to print
Print Float	2	$f12 = float value to print
Print Double	3	$f12 = double value to print
Print String	4	$a0 = address of null-terminated string
Read Integer	5	Return integer value in $v0
Read Float	6	Return float value in $f0
Read Double	7	Return double value in $f0
Read String	8	$a0 = address of input buffer $a1 = maximum number of characters to read
Allocate Heap memory	9	$a0 = number of bytes to allocate Return address of allocated memory in $v0
Exit Program	10	

Các dịch vụ của Syscall …

Print Char	11	$a0 = character to print
Read Char	12	Return character read in $v0
Open File	13	$a0 = address of null-terminated filename string $a1 = flags (0 = read-only, 1 = write-only) $a2 = mode (ignored) Return file descriptor in $v0 (negative if error)
Read from File	14	$a0 = File descriptor $a1 = address of input buffer $a2 = maximum number of characters to read Return number of characters read in $v0
Write to File	15	$a0 = File descriptor $a1 = address of buffer $a2 = number of characters to write Return number of characters written in $v0
Close File	16	$a0 = File descriptor

Ví dụ syscall - Đọc và In số nguyên

```
################### Code segment #######################
.text
.globl main
main:                                   # main program entry
   li    $v0, 5                         # Read integer
   syscall                              # $v0 = value read

   move  $a0, $v0                       # $a0 = value to print
   li    $v0, 1                         # Print integer
   syscall

   li    $v0, 10                        # Exit program
   syscall
```

Ví dụ syscall – Đọc và In chuỗi

```
################## Data segment ######################
.data
   str: .space  10          # array of 10 bytes
################## Code segment ######################
.text
.globl main
main:                       # main program entry
   la    $a0, str           # $a0 = address of str
   li    $a1, 10            # $a1 = max string length
   li    $v0, 8             # read string
   syscall
   li    $v0, 4             # Print string str
   syscall
   li    $v0, 10            # Exit program
   syscall
```

Ví dụ CT1: Tổng ba số nguyên

```
# Sum of three integers
#
# Objective: Computes the sum of three integers.
#     Input: Requests three numbers.
#    Output: Outputs the sum.
#################### Data segment ####################
.data
prompt:   .asciiz     "Please enter three numbers: \n"
sum_msg:  .asciiz     "The sum is: "
#################### Code segment ####################
.text
.globl main
main:
      la    $a0,prompt          # display prompt string
      li    $v0,4
      syscall
      li    $v0,5               # read 1st integer into $t0
      syscall
      move  $t0,$v0
```

Ví dụ CT1: Tổng ba số nguyên…

```
li     $v0,5                # read 2nd integer into $t1
syscall
move   $t1,$v0

li     $v0,5                # read 3rd integer into $t2
syscall
move   $t2,$v0

addu   $t0,$t0,$t1          # accumulate the sum
addu   $t0,$t0,$t2

la     $a0,sum_msg          # write sum message
li     $v0,4
syscall

move   $a0,$t0              # output sum
li     $v0,1
syscall

li     $v0,10               # exit
syscall
```

Ví dụ CT2: Đổi chữ thường sang hoa

```
# Objective: Convert lowercase letters to uppercase
#     Input: Requests a character string from the user.
#    Output: Prints the input string in uppercase.
##################### Data segment ######################
.data
name_prompt:  .asciiz          "Please type your name: "
out_msg:      .asciiz          "Your name in capitals is: "
in_name:      .space 31        # space for input string
##################### Code segment ######################
.text
.globl main
main:
      la    $a0,name_prompt   # print prompt string
      li    $v0,4
      syscall
      la    $a0,in_name       # read the input string
      li    $a1,31            # at most 30 chars + 1 null char
      li    $v0,8
      syscall
```

Ví dụ CT2: Đổi chữ thường sang hoa…

```
        la    $a0,out_msg       # write output message
        li    $v0,4
        syscall
        la    $t0,in_name
loop:
        lb    $t1,($t0)
        beqz  $t1,exit_loop     # if NULL, we are done
        blt   $t1,'a',no_change
        bgt   $t1,'z',no_change
        addiu $t1,$t1,-32       # convert to uppercase: 'A'-'a'=-32
        sb    $t1,($t0)
no_change:
        addiu $t0,$t0,1         # increment pointer
        j     loop
exit_loop:
        la    $a0,in_name       # output converted string
        li    $v0,4
        syscall
        li    $v0,10            # exit
        syscall
```

Ví dụ CT3: Truy xuất File

```
# Sample MIPS program that writes to a new text file
.data
file:   .asciiz "out.txt"       # output filename
buffer: .asciiz "Sample text to write"

.text
li   $v0, 13        # system call to open a file for writing
la   $a0, file      # output file name
li   $a1, 1         # Open for writing (flags 1 = write)
li   $a2, 0         # mode is ignored
syscall             # open a file (file descriptor returned in $v0)
move $s6, $v0       # save the file descriptor
li   $v0, 15        # Write to file just opened
move $a0, $s6       # file descriptor
la   $a1, buffer    # address of buffer from which to write
li   $a2, 20        # number of characters to write = 20
syscall             # write to file
li   $v0, 16        # system call to close file
move $a0, $s6       # file descriptor to close
syscall             # close file
```

Tiếp theo …

- Các phát biểu trong hợp ngữ MIPS
- Khung dạng chương trình hợp ngữ MIPS
- Định nghĩa/khai báo dữ liệu
- Địa chỉ bắt đầu (alignment) và thứ tự các byte trong bộ nhớ
- Các hàm hệ thống
- Thủ tục/hàm (Procedure)
- Truyền tham số và Runtime Stack

Thủ tục/hàm - Procedure

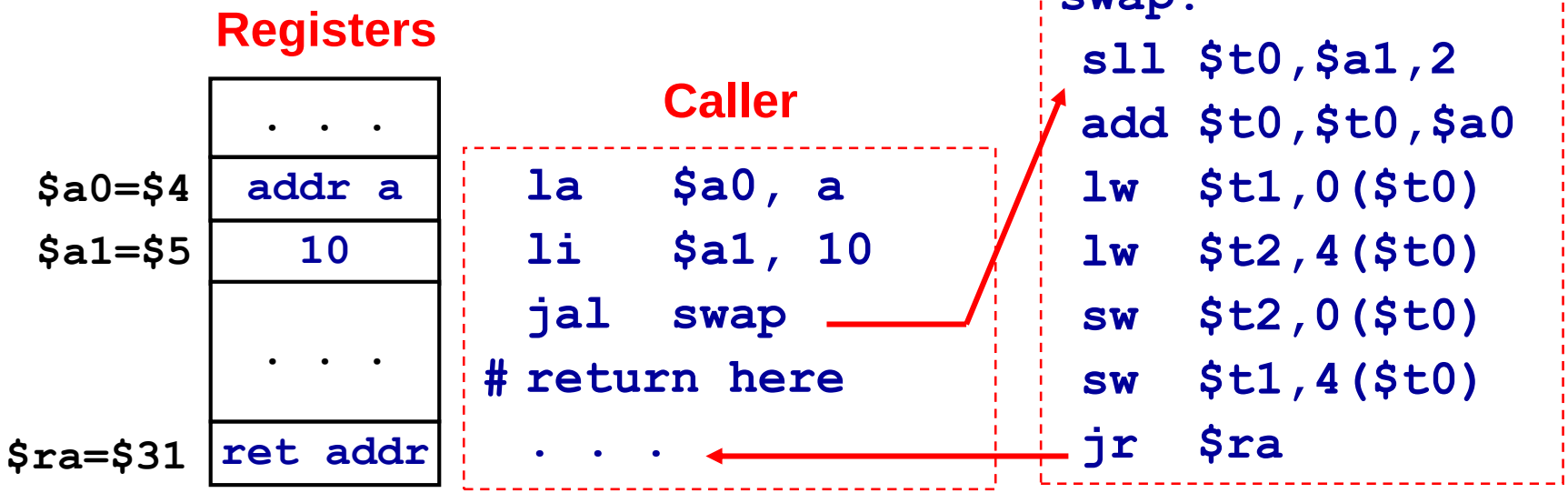
- Hàm giúp lập trình viên phân chia chương trình thành các khối chức năng để:
  - Dễ dàng cho việc viết và sửa lỗi
  - Cho phép dùng lại code
- Hàm giúp lập trình viên tập trung hiện thực một khối chức năng đơn giản tại một thời điểm
  - Các tham số là giao diện giữa hàm và phần còn lại của chương trình, các tham số gồm có đối số truyền cho hàm (arguments) và giá trị trả về (results)

6 bước trong quá trình sử dụng hàm

1. Hàm chính (caller) khởi tạo các đối số cho hàm được gọi (callee) và các vùng dữ liệu mà hàm được gọi có thể truy xuất
   - `$a0` – `$a3`: 4 thanh ghi đối số (argument)
   - `Stack`
2. Caller chuyển điều khiển đến callee
3. Callee thực hiện lấy dữ liệu cần thiết ở vùng nhớ mà hàm chính đã khởi tạo
4. Callee thực hiện công việc
5. Callee đặt kết quả trả về ở nơi mà caller có thể truy xuất
   - `$v0` – `$v1`: 2 thanh ghi chứa giá trị trả về
6. Callee trả điều khiển cho caller
   - `$ra`: thanh ghi chứa địa chỉ trở về return address

Ví dụ trình tự gọi hàm/trở về

- Giả sử muốn gọi hàm: **`swap(a,10)`**
  - **1**.Khởi tạo đối số `$a0` = **địa chỉ** của mảng **`a`** và `$a1=10`
  - **2**.Lưa **địa chỉ trở về** vào thanh ghi **$31 = $ra,** và nhảy đến hàm **`swap`**
  - **3,4,5**.Hàm **`swap`** thực thi công việc
  - **6**.Trở về hàm gọi bằng cách nhảy đến địa chỉ đã lư trong thanh ghi **$ra** (return address)



Các lệnh cho việc gọi hàm và trở về

- JAL (Jump-and-Link) sử dụng để gọi hàm, thực thi 2 việc:
  - Lưu địa chỉ trở về (**return address**) vào thanh ghi **$ra = PC+4** và
  - Nhảy tới hàm được gọi
- JR (Jump Register) được sử dụng để trở về hàm gọi
  - Nhảy tới vị trí lệnh trong thanh ghi Rs (PC = Rs)
- JALR (Jump-and-Link Register)
  - Lưu địa chỉ trở về vào thanh ghi Rd = PC+4, và
  - Nhảy tới vị trí lệnh trong thanh ghi Rs (PC = Rs)
  - Có thể dùng để gọi hàm (địa chỉ được biết trong lúc chạy)

Instruction		Meaning	Format					
jal	label	$31=PC+4, jump	$op^6 = 3$	imm^{26}				
jr	Rs	PC = Rs	$op^6 = 0$	rs^5	0	0	0	8
jalr	Rd, Rs	Rd=PC+4, PC=Rs	$op^6 = 0$	rs^5	0	rd^5	0	9

Ví dụ viết thủ tục/hàm

- Xét hàm swap (ngôn ngữ C)
- Chuyển sang hợp ngữ MIPS

```
void swap(int v[], int k)
{   int temp;
    temp = v[k]
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

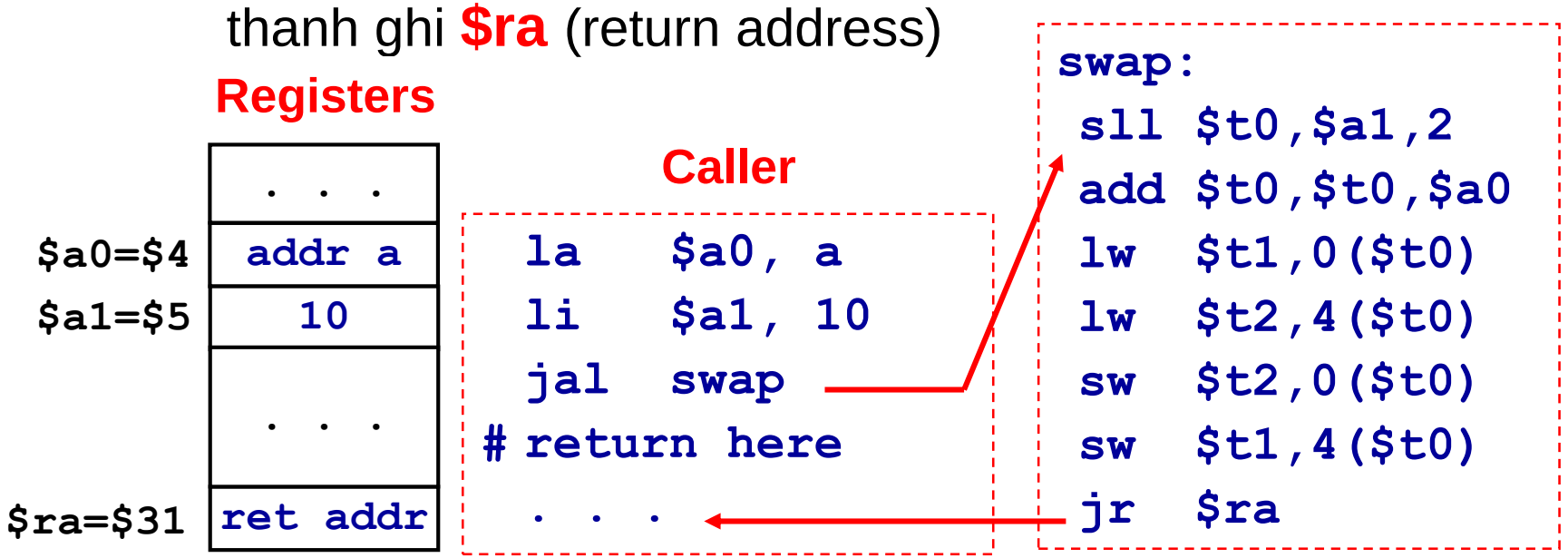
Parameters:

`$a0` = Address of `v[]`
`$a1` = `k`, and
Return address is in `$ra`

```
swap:
 sll $t0,$a1,2     # $t0=k*4
 add $t0,$t0,$a0   # $t0=v+k*4
 lw  $t1,0($t0)    # $t1=v[k]
 lw  $t2,4($t0)    # $t2=v[k+1]
 sw  $t2,0($t0)    # v[k]=$t2
 sw  $t1,4($t0)    # v[k+1]=$t1
 jr  $ra           # return
```

Trình tự gọi hàm/trở về

- Gọi hàm: **`swap(a,10)`**
  - **1**.Khởi tạo đối số `$a0` = **địa chỉ** của mảng **`a`** và `$a1=10`
  - **2**.Lưa **địa chỉ trở về** vào thanh ghi **$31 = $ra,** và nhảy đến hàm **`swap`**
  - **3,4,5**.Hàm **`swap`** thực thi công việc
  - **6**.Trở về hàm gọi bằng cách nhảy đến địa chỉ đã lư trong thanh ghi **$ra** (return address)



Chi tiết lệnh JAL và JR

Address	Instructions	Assembly Language
00400020	lui $1, 0x1001	la $a0, a
00400024	ori $4, $1, 0	
00400028	ori $5, $0, 10	ori $a1,$0,10
0040002C	jal 0x10000f	jal swap
00400030	. . .	# return here
		swap:
0040003C	sll $8, $5, 2	sll $t0,$a1,2
00400040	add $8, $8, $4	add $t0,$t0,$a0
00400044	lw $9, 0($8)	lw $t1,0($t0)
00400048	lw $10,4($8)	lw $t2,4($t0)
0040004C	sw $10,0($8)	sw $t2,0($t0)
00400050	sw $9, 4($8)	sw $t1,4($t0)
00400054	jr $31	jr $ra

Pseudo-Direct Addressing

PC = imm26<<2

0x10000f << 2

= 0x0040003C

$31 0x00400030

Register $31 is the return address register

Tiếp theo …

- Các phát biểu trong hợp ngữ MIPS
- Khung dạng chương trình hợp ngữ MIPS
- Định nghĩa/khai báo dữ liệu
- Địa chỉ bắt đầu (alignment) và thứ tự các byte trong bộ nhớ
- Các hàm hệ thống
- Thủ tục/hàm
- Truyền tham số và Runtime Stack

Truyền tham số

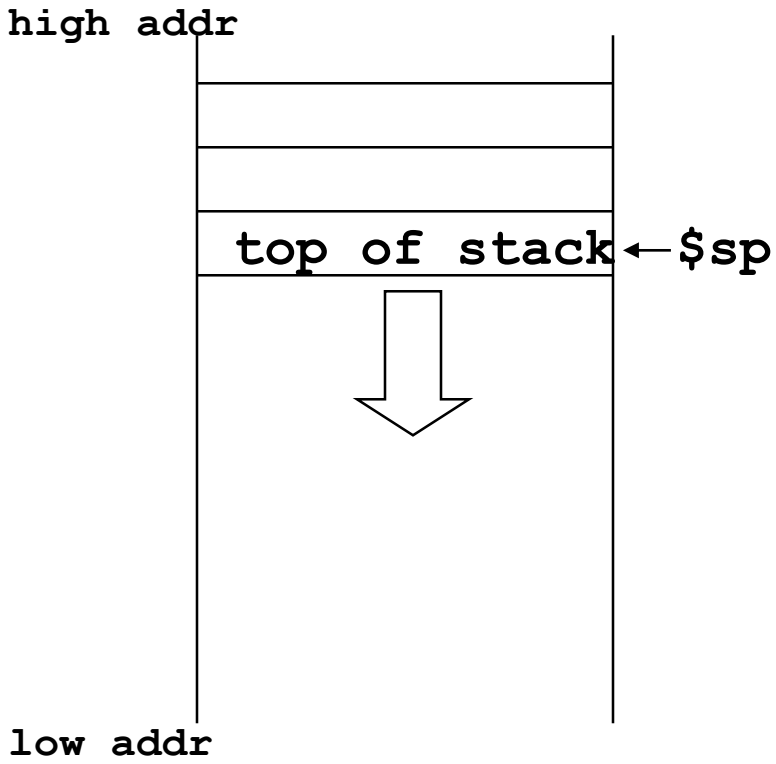
- Truyền tham số cho thủ tục/hàm
  - Đưa tấc cả các tham số cần thiết vào vùng lưu trữ mà thủ tục/hàm được gọi có thể truy xuất được
  - Sau đó gọi hàm (jal)
- Hai loại vùng lưu trữ
  - Thanh ghi: các thanh ghi đa mục đích được sử dụng (phương pháp truyền tham số bằng thanh ghi)
  - Bộ nhớ: vùng nhớ stack (phương pháp truyền tham số bằng stack)
- Hai cơ chế phổ biến của việc truyền tham số
  - Pass-by-value: **giá trị** của tham số được truyền
  - Pass-by-reference: **địa chỉ** của tham số được truyền

Truyền tham số….

- Theo quy ước, thanh ghi được sử dụng để truyền
  - `$a0 = $4 .. $a3 = $7` **truyền tham số**
  - `$v0 = $2 .. $v1 = $3` **kết quả trả về**
- Các tham số/kết quả trả về khác có thể dùng stack
- Trong khi thực thi stack (runtime stack) cũng cần thiết để…
  - Lưu biến/ cấu trúc dữ liệu lớn không chứa vừa trong các thanh ghi
  - Lưu và phục hồi các thanh ghi trong quá trình gọi hàm
  - Hiện thực đệ quy
- Runtime stack sử dụng hai thanh ghi để quản lý
  - **stack pointer** `$sp = $29` (points to top of stack)
  - **frame pointer** `$fp = $30` (points to a procedure frame)

Truyền tham số….

- Khi hàm được gọi (callee) cần nhiều hơn số lượng thanh ghi mặc định cho tham số và giá trị trả về?
  - callee sử dụng stack – hàng đợi last-in-first-out



- Thanh ghi **$sp** (**29**) được sử dụng để giữ địa chỉ của stack (stack "lớn" theo chiều đi xuống)
  - thêm dữ liệu vào stack– **push**

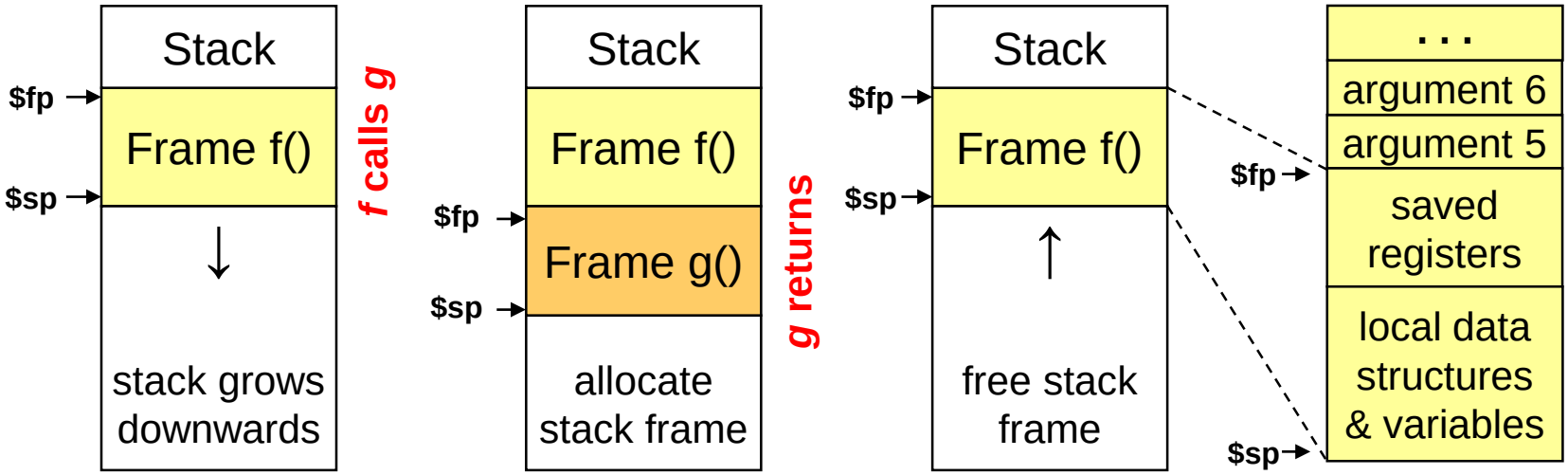
```
$sp = $sp – 4  #mở rộng stack
sw data,0($sp) #lưu vào tos
```

  - lấy dữ liệu từ stack– **pop**

```
lw data,0($sp) #lấy dữ liệu
$sp = $sp + 4  #chỉnh lại tos
```

Stack Frame

- Stack frame là một vùng nhớ stack…
  - Chứa các thanh ghi cần bảo toàn giá trị (saved registers), và các cấu trúc dữ liệu và các biến cục bộ (nếu có)
- Được gọi là activation frame
- Frame được push hay pop bằng…
  - Stack pointer **`$sp = $29`** và Frame pointer **`$fp = $30`**
  - Giảm **`$sp`** để cấp phát, tăng **`$sp`** để giải phóng



Quy ước sử dụng các thanh ghi

- $a0 – $a3: arguments (reg's 4 – 7)
- $v0, $v1: result values (reg's 2 and 3)
- $t0 – $t9: temporaries
  - Có thể thay đổi giá trị tùy ý trong hàm
- $s0 – $s7: saved
  - Phải bảo toàn giá trị khi hàm thực thi lệnh trở về hàm gọi (lưu/phục hồi bằng stack)
- $gp: global pointer cho dữ liệu tĩnh(reg 28)
- $sp: stack pointer (reg 29)
- $fp: frame pointer (reg 30)
- $ra: return address (reg 31)

Các quy ước về gọi hàm

❖ Hàm gọi Caller phải thực hiện:

1. Truyền tham số (arguments)
   - ✧ Nhỏ hơn 5 tham số: sử dụng thanh ghi $a0 đến $a3
   - ✧ Tham số thứ 5 trở đi: đưa vào đỉnh stack
2. Lưu các thanh ghi $a0 - $a3 và $t0 - $t9 nếu cần
   - ✧ Thanh ghi $a0 - $a3 và $t0 - $t9 được thay đổi giá trị tùy ý trong hàm được gọi callee
   - ✧ Lưu vào stack trước khi gọi hàm
   - ✧ Phục hồi giá trị sau khi hàm được gọi trở về
3. Gọi hàm, thực thi lệnh JAL
   - ✧ Nhảy đến hàm được gọi
   - ✧ Lưu địa chỉ trả về trong thanh ghi $ra

Các quy ước về gọi hàm - 2

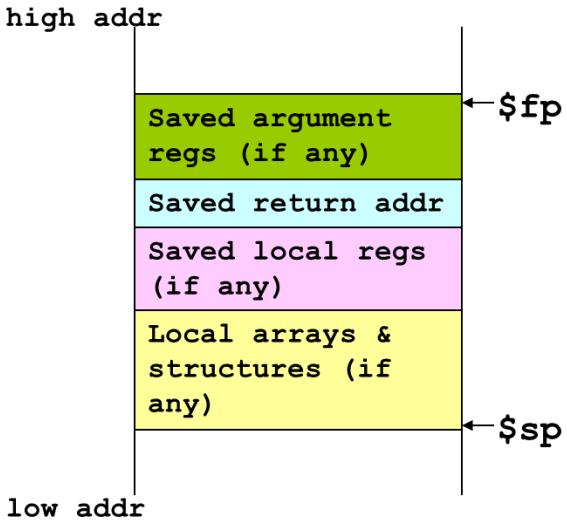
❖ Hàm được gọi Callee phải làm các việc sau:

1. Cấp phát vùng nhớ cho stack frame
   - ✧ `$sp = $sp – n` (n byte cấp phát cho stack frame)
   - ✧ Lập trình viên cần xác định n
   - ✧ Hàm lá không cần stack frame (n = 0)
2. Lưu các thanh ghi $ra, $fp, $s0 - $s7 vào stack frame
   - ✧ $ra, $fp, $s0 - $s7 phải được bảo toàn giá trị trước khi thực hiện việc trở về (return)
   - ✧ Trước khi thay đổi giá trị (nếu cần)
   - ✧ Thanh ghi $ra cần phải lưu nếu trong hàm gọi một hàm khác (hàm lồng)
3. Cập nhật frame pointer $fp (nếu cần)
   - ✧ Các hàm đơn giản, thanh ghi không cần sử dụng $fp

Các quy ước lúc trở về (return)

❖ Trước khi return, hàm được gọi phải đảm bảo:

1. Đưa giá trị trả về vào $v0 và $v1 (nếu có)
2. Phục hồi giá trị các thanh ghi đã lưu vào stack lúc đầu
   - ✧ pop giá trị thanh ghi $ra, $fp, $s0 - $s7 nếu đã lưu trong stack frame
3. Giải phóng stack frame
   - ✧ `$sp = $sp + n`
4. Trở về hàm đã gọi
   - ✧ Nhảy đến địa chỉ trở về trong thanh ghi $ra: `jr $ra`



Ví dụ hàm lá 1

- Hàm lá là hàm không gọi các hàm khác
- C code:

```
int leaf_example (int g, h, i, j)
{ int f;
  f = (g + h) - (i + j);
  return f;
}
```

  - Arguments `g`, …, `j` in `$a0`, …, `$a3`
  - Result in `$v0`

leaf_example 2

- MIPS code:

```
leaf_example:
    add  $t0, $a0, $a1
    add  $t1, $a2, $a3
    sub  $v0, $t0, $t1
    jr   $ra
```

Procedure body

Result

Return

Ví dụ hàm lá 2

- C code:

```
int leaf_example (int g, h, i, j)
{ int f;
  f = (g + h) - (i + j);
  return f;
}
```

  - Arguments `g`, …, `j` in `$a0`, …, `$a3`
  - `f` in `$s0` (cần phải bảo toàn giá trị trước khi trở về)
  - Result in `$v0`

leaf_example 2

- MIPS code:

```
leaf_example:
    addi $sp, $sp, -4
    sw   $s0, 0($sp)
    add  $t0, $a0, $a1
    add  $t1, $a2, $a3
    sub  $s0, $t0, $t1
    add  $v0, $s0, $zero
    lw   $s0, 0($sp)
    addi $sp, $sp, 4
    jr   $ra
```

Save $s0 on stack

Procedure body

Result

Restore $s0

Return

Hàm lồng

- Hàm lồng là hàm có thực hiện gọi hàm khác
- Địa chỉ trở về trong thanh ghi $ra trong hàm lồng?

```
int rt_1 (int i) {
    if (i == 0) return 0;
    else return i + rt_2(i-1); }

caller: jal  rt_1
next:   . . .

rt_1:   bne  $a0, $zero, to_2
        add  $v0, $zero, $zero
        jr   $ra
to_2:   addu $t0, $zero, $a0
        addi $a0, $a0, -1
        jal  rt_2
        addu $v0, $t0, $v0
        jr   $ra
rt_2:   . . .
```

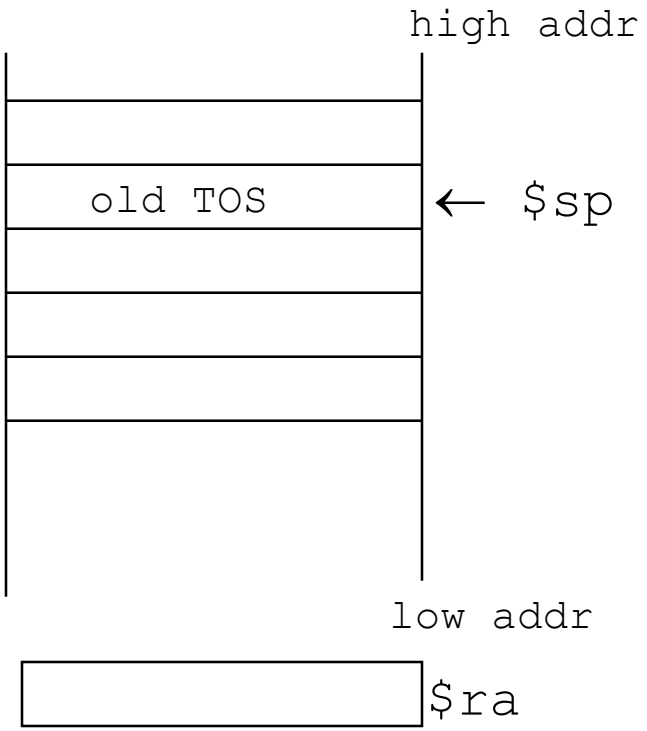
Giá trị thanh ghi $ra

```
caller: jal  rt_1
next:   . . .

rt_1:   bne  $a0, $zero, to_2
        add  $v0, $zero, $zero
        jr   $ra
to_2:   addu $t0, $zero, $a0
        addi $a0, $a0, -1
        jal  rt_2
        addu $v0, $t0, $v0
        jr   $ra
rt_2:   . . .
```

- Khi gọi hàm **rt_1**, địa chỉ trở về (**next** ở hàm caller) lưu vào **$ra**. Giá trị thanh ghi **$ra** (i != 0) khi **rt_1** gọi hàm **rt_2**?

Lưu địa chỉ trở về

- Hiện thực đúng (`i` là `$a0`, giá trị trả về `$v0`)




```
rt_1: bne   $a0, $zero, to_2
      add   $v0, $zero, $zero
      jr    $ra
to_2: addi  $sp, $sp, -8
      sw    $ra, 4($sp)
      sw    $a0, 0($sp)
      addi  $a0, $a0, -1
      jal   rt_2
bk_2: lw    $a0, 0($sp)
      lw    $ra, 4($sp)
      addi  $sp, $sp, 8
      addu  $v0, $v0, $a0
      jr    $ra
```

- Lưu địa chỉ trở về (và tham số) trên stack

Ví dụ hàm lồng – tính giai thừa

- C code:

```
int fact (int n)
{
  if (n < 1) return 1;
  else return n * fact(n - 1);
}
```

  - Argument **n** in **$a0**
  - Result in **$v0**

Ví dụ hàm lồng – tính giai thừa …

- MIPS code:

```
fact:
    addi $sp, $sp, -8       # adjust stack for 2 items
    sw   $ra, 4($sp)        # save return address
    sw   $a0, 0($sp)        # save argument
    slti $t0, $a0, 1        # test for n < 1
    beq  $t0, $zero, L1
    addi $v0, $zero, 1      # if so, result is 1
    addi $sp, $sp, 8        #   pop 2 items from stack
    jr   $ra                #   and return
L1: addi $a0, $a0, -1       # else decrement n
    jal  fact               # recursive call
    lw   $a0, 0($sp)        # restore original n
    lw   $ra, 4($sp)        #   and return address
    addi $sp, $sp, 8        # pop 2 items from stack
    mul  $v0, $a0, $v0      # multiply to get result
    jr   $ra                # and return
```

Chuyển một hàm đệ quy

- Ví dụ hàm tính giai thừa

```
int fact (int n) {
    if (n < 1) return 1;
    else return (n * fact (n-1)); }
```

- Thực thi hàm đệ quy (hàm gọi chính nó!)

  fact (0) = 1

  fact (1) = 1 * 1 = 1

  fact (2) = 2 * 1 * 1 = 2

  fact (3) = 3 * 2 * 1 * 1 = 6

  fact (4) = 4 * 3 * 2 * 1 * 1 = 24

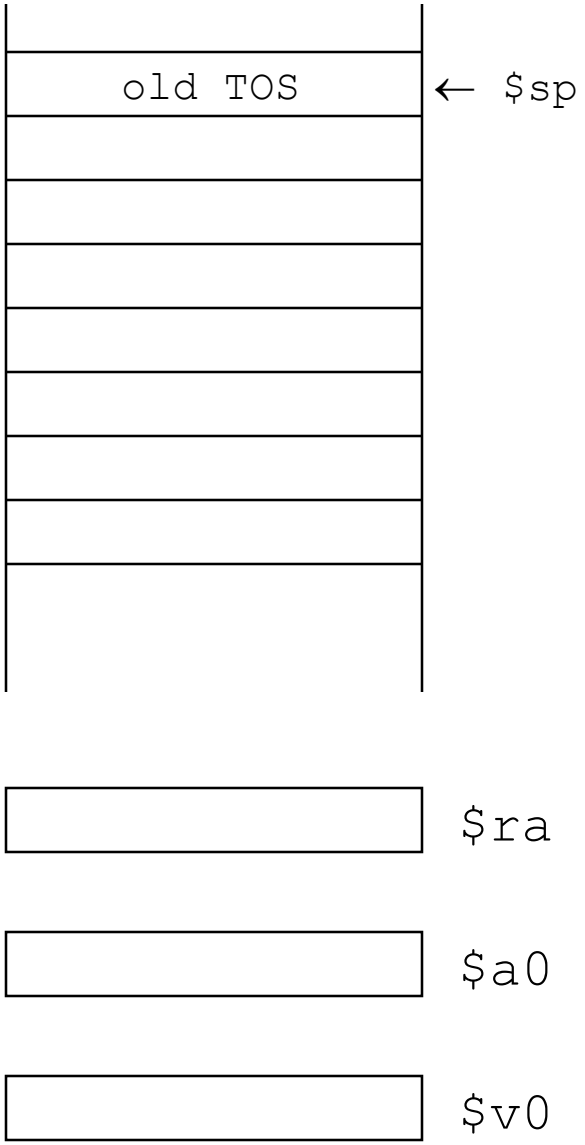
  . . .

- Giả sử **n** là thanh ghi **$a0**; kết quả trở về trong **$v0**

Chuyển một hàm đệ quy…

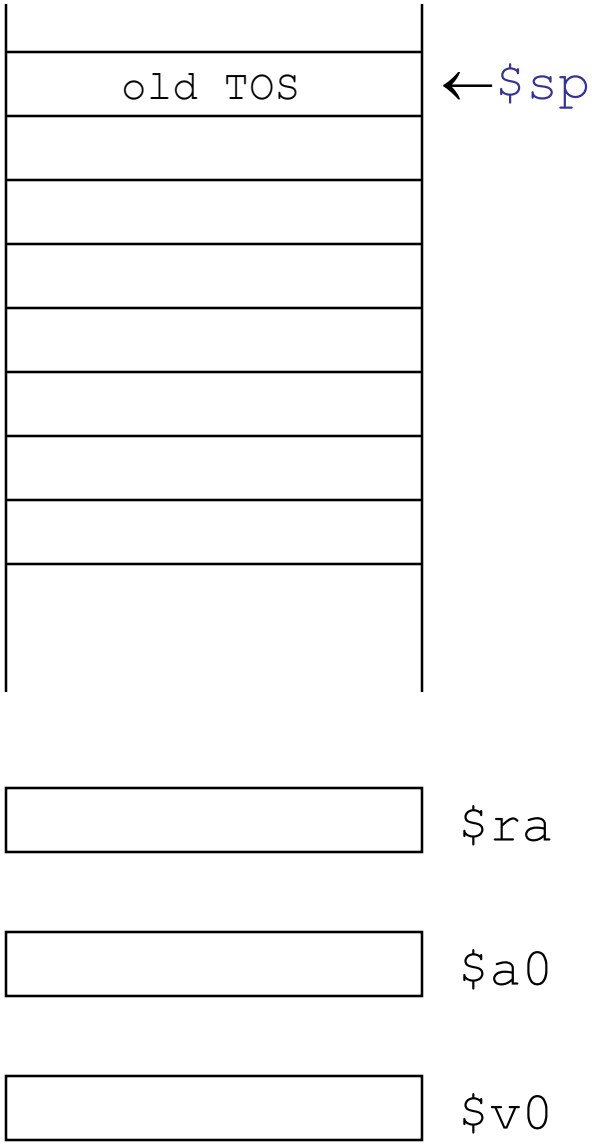
```
fact: addi  $sp, $sp, -8       #adjust stack pointer
      sw    $ra, 4($sp) #save return address
      sw    $a0, 0($sp) #save argument n
      slti  $t0, $a0, 1 #test for n < 1
      beq   $t0, $zero, L1     #if n >=1, go to L1
      addi  $v0, $zero, 1      #else return 1 in $v0
      addi  $sp, $sp, 8 #adjust stack pointer
      jr    $ra                #return to caller (1st)

L1:   addi  $a0, $a0, -1       #n >=1, so decrement n
      jal   fact               #call fact with (n-1)
      #this is where fact returns

bk_f: lw    $a0, 0($sp) #restore argument n
      lw    $ra, 4($sp) #restore return address
      addi  $sp, $sp, 8 #adjust stack pointer
      mul   $v0, $a0, $v0      #$v0 = n * fact(n-1)
      jr    $ra                #return to caller (2nd)
```

Giá trị trên stack khi n=2 (thời điểm 1)



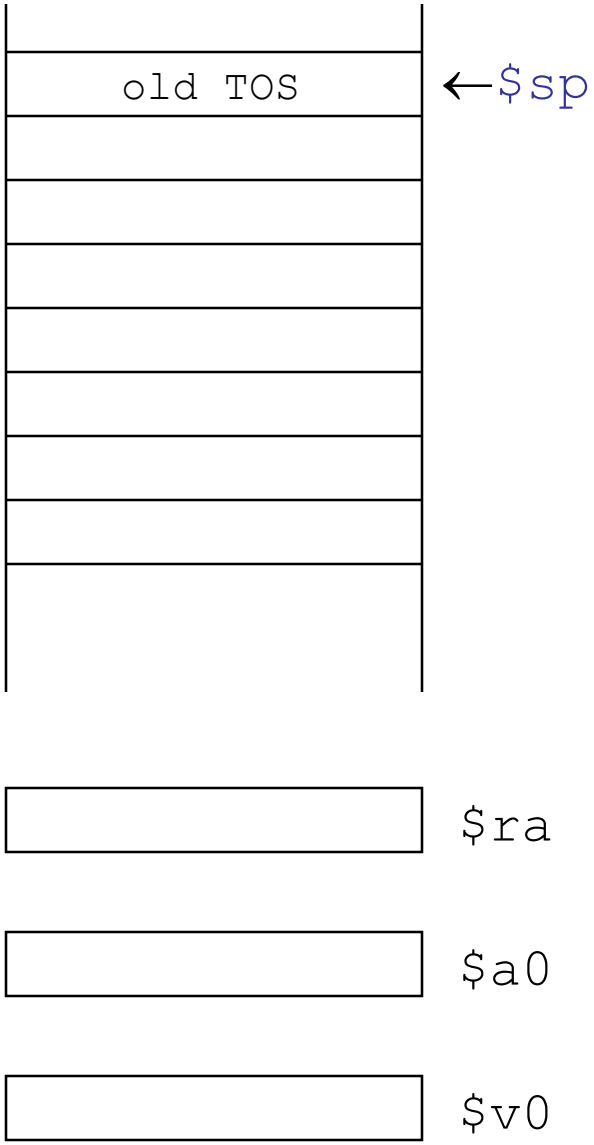
- Trạng thái Stack lúc thực thi lệnh **jal** (*gọi hàm fact với* $a0 lúc này bằng 1)
  - lưu địa chỉ trở về của hàm gọi caller (vd: một vị trí nào đó trong hàm main khi thực hiện gọi hàm fact *lần đầu tiên*) vào stack
  - lưu tham số $a0 ban đầu (n=2) vào stack

Giá trị trên stack khi n=2 (thời điểm 2)



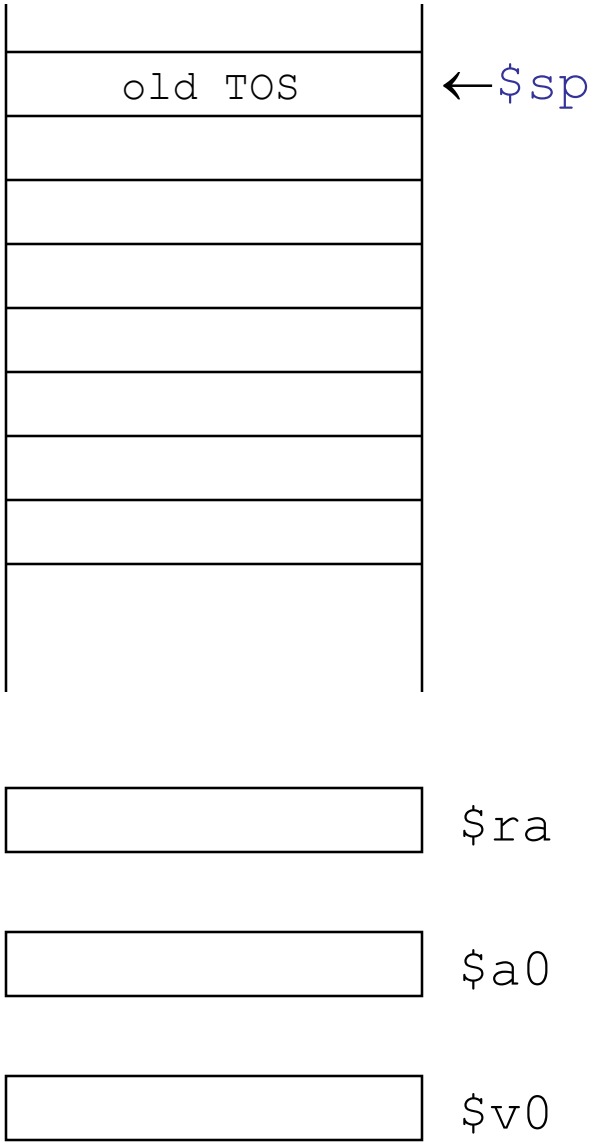
- Stack sau khi thực thi lệnh **jal** lần 2 (*gọi hàm fact với* **$a0** lúc này bằng 0)
  - Lưu địa chỉ trở về **bk_f** (vị trí sau lệnh **jal**) vào stack
  - Lưu giá trị **$a0** trước đó (n=1) vào stack

Giá trị trên stack khi n=2 (thời điểm 3)



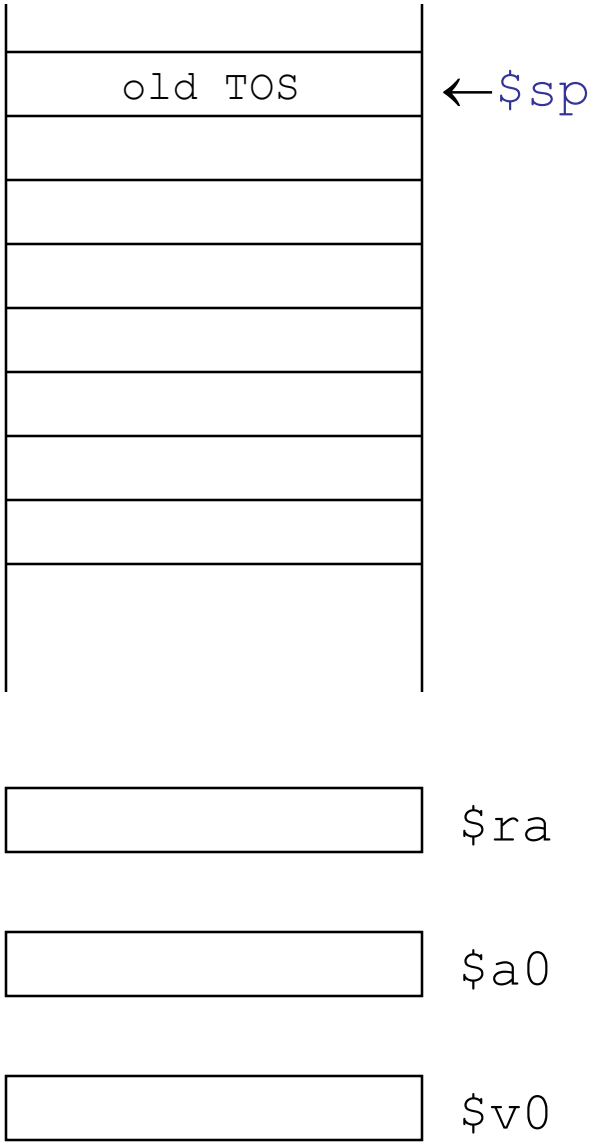
- Stack khi thực thi lệnh **jr** đầu tiên(**$v0** gán giá trị 1)
  - Nhảy tới nhãn **bk_f**
  - Cập nhật stack pointer

Giá trị trên stack khi n=2 (thời điểm 4)



- Stack khi thực thi lệnh **jr** vị trí thứ 2 (trở về sau khi gán **$v0** = 1 * 1)
  - trở về vị trí **$ra** = **bk_f** được phục hồi từ stack
  - giá trị **$a0 = 2** cũng được phục hồi từ stack
  - phục hồi stack pointer

Giá trị trên stack khi n=2 (thời điểm 5)



- Stack khi thực thi lệnh **jr** vị trí thứ 2 (trở về sau khi gán **$v0** = 2* 1 * 1)
  - trở về vị trí sau lệnh gọi trong hàm main
  - **$a0 = 2**
  - stack pointer phục hồi giá trị nguyên thủy