# CENG3420 Computer Organization and Design
## Lab 1-2: System calls and recursions

**Wen Zong**

Department of Computer Science and Engineering
The Chinese University of Hong Kong

*wzong@cse.cuhk.edu.hk*

香港中文大學
The Chinese University of Hong Kong

# Overview

# Save register to stack I

A subroutine *fun_a* can also call another subroutine *fun_b*. When *fun_a* calls *fun_b* using *jal fun_b*, $ra is overwritten to the address returning to *fun_a*.

## $ra overwritten example

```
fun_a: ins 0
       ins 1
       jal fun_b   # $ra will store addess of ins 2
       ins 2
       jr $ra      # hope to return to fun_a's caller

fun_b: ins 3
       jr $ra      # return to fun_a
```

$ra needs to be saved to stack, and becomes like this:

# Save register to stack II

## save $ra

```
fun_a: addi $sp, -4      # allocate  space in stack
       sw 0($sp), $ra    # save $ra value
       ins 0
       ins 1
       jal fun_b         # $ra will store addess of ins 2
       ins 2
       lw $ra, 0($sp)    # restore $ra value
       add $sp, 4        # free stack space
       jr $ra            # return to fun_a's caller

fun_b: ins 3
       jr $ra            # return to fun_a
```

# Save register to stack III

Similarly we can save other registers to stack by allocating more spaces in stack.
Register are divided to saved register group \$s0 - \$s7 and temporary register group \$t0 - \$t7. By convention, caller expects callee to save the saved register group but does not expect temporary register to be saved.

# Recursion

Recursion is a function call to itself which is a special case of mentioned nested function call. It usually works in a divide and conquer manner.
The same with nested function call, in recursive functions, register contents that is useful after a subroutine call should be saved to stack since they may be modified by subroutines.

# Recurion example I

Factorial can be computed recursively $fact(n) = n * fact(n-1)$. When the sub-problem $f(n-1)$ is solved, we can multiply it by $n$ to get the result of current problem $f(n)$. When writting the assembly code we should save the register that contians this $n$ which is $\$a0$ in this example.

## Factorials

# Recurion example II

```
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
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
```

# Stack behavior during fact() call I

Suppose we call *fact*(3) in *main* function, the stack will grow like this after recursive function calls.

| | |
|---|---|
| ret_add in main | |
| \$a0 (=3) | in fact(3) |
| bk_f | |
| \$a0 (=2) | in fact(2) |
| bk_f | |
| \$a0 (=1) | in fact(1) |
| bk_f | |
| \$a0 (=0) | in fact(0) |

# Stack behavior during fact() call II

In *fact*(0), it won't invoke *fact*() anymore and sets $\$v0 = 1$ and returns. The stack will shrink backwards to merge the solutions. When *fact*(0) returns, the stack looks like this,

| |
|---|
| ret_add in main |
| $a0 (=3) |
| bk_f |
| $a0 (=2) |
| bk_f |
| $a0 (=1) |

at *bk_f*, $v0 = 1

The program continues to execute code at label *bk_f*, which mulitplies $\$a0$ with $\$v0$ and then return to caller.

# Stack behavior during fact() call III

After *fact*(1) returns to *fact*(2), the stack content is:

| |
|---|
| ret_add in main |
| \$a0 (=3) |
| bk_f |
| \$a0 (=2) |

at $bk\_f$, \$v0 = 1

After *fact*(2) returns to *fact*(3), the stack content is:

| |
|---|
| ret_add in main |
| \$a0 (=3) |

at $bk\_f$, \$v0 = 2

*fact*(3) will return to the its caller the main function and the result is stored in $\$v0$ register.

# Exercises

Write a Quicksort function using MIPS assembly language. We provide a c++ version of the Quicksort function, and you need to translate it to MIPS assembly. The array is declared in the data segment with name *array*. The starting code looks like this:

```
        .data
array:  .word  -1 22 8 35 5 4 11 2 1 78

        .text
main:   la $a0, array
        li $a1, 0
        li $a2, 9
        jal qsort
        li $v0, 10
        syscall
qsort:  ...
```

# Overview

# The ideal of Quicksort

Quicksort is a divide and conquer algorithm. Quicksort first divides a large array into two smaller sub-arrays: the low elements and the high elements. Quicksort can then recursively sort the sub-arrays. The steps are:

1. Pick an element, called a pivot, from the array.
2. Partitioning: reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation.
3. Recursively apply the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values. The base case of the recursion is arrays of size zero or one, which never need to be sorted.

# C++ implementation of Quicksort I

```
void quickSort(int arr[], int left, int right) {
      int i = left, j = right;
      int tmp;
      int pivot = arr[(left + right) / 2];

      /* partition */
      while (i <= j) {
            while (arr[i] < pivot)
                  i++;
            while (arr[j] > pivot)
                  j--;
            if (i <= j) {
                  tmp = arr[i];
                  arr[i] = arr[j];
                  arr[j] = tmp;
```

# C++ implementation of Quicksort II

```
                        i++;
                        j--;
                }
        };

        /* recursion */
        if (left < j)
                quickSort(arr, left, j);
        if (i < right)
                quickSort(arr, i, right);
}
```