# COMP2611: Computer Organization

# Recursion in MIPS

# Overview

- You will learn the following in this lab:
  - the concept of recursion,
  - the reason for using a stack to implement a recursion in MIPS,
  - how a recursion can be implemented correctly in MIPS.

# Reminder about recursion

- The idea of recursion is to solve a big problem by dividing it into a number of **smaller problems** that are **identical to the original problem**, and then further divide the smaller problems to even smaller problems until we reach the base case.

# Reminder about recursion

factorial(n) = factorial(0) x 1 x 2 x 3… x n

factorial(0) = 1
factorial(1) = factorial(0) x 1
factorial(2) = factorial(1) x 2
factorial(3) = factorial(2) x 3

: :

: :

factorial(n-1) = factorial(n-2) x (n-1)
factorial(n) = factorial(n-1) x n

- The base case is usually simple enough to be solved immediately.
- After solving the base case, we return to one level up in the recursion tree. With the result from the base case we can solve the problem at this level easily. Then we return with the result and solve the problem in the next level (for example n=3 for factorial(n)).
- Eventually we return to the original problem, and with the result returned from the immediate lower level, the original problem is solved (i.e. factorial(n) = n*factorial(n-1)).

# Reminder about recursion

- To implement the recursion in MIPS isn't so straight forward.
- As you will see, you need to take care of the returning addresses of the recursion in MIPS.
- You may also need to store some intermediate results for further uses.
- You will find the data structure known as "stack" useful for keeping returning addresses and storing the intermediate results.
- In this lab., we will go through a MIPS recursion program with you and illustrate how we use the stack to implement a recursion.

# Recursion example (factorial)

- **The following is a piece of sample recursive C/C++ code for calculating the factorial:**

```
1. int factorial (int n){ // n is assumed to be +ve
2.         if  (n == 0)
3.           return 1;     // base case reached
4.  else
5.         return n*factorial ( n - 1 ); // non-base case
6.  }
```

# Recursion example (factorial)

- Assume the **argument *n*** is stored in $a0 and the **return value** is stored in $v0.

- The argument *n* in $a0 will be modified for multiple times as we run the recursion, because we are calling the factorial function with smaller and smaller values of argument.

- In order not to lose the current value of *n* when the factorial(n) is being called, we need to **push (store) the value of $a0 onto the stack** (why? Because you need *n* in order to calculate the value n*factorial(n-1)! ).

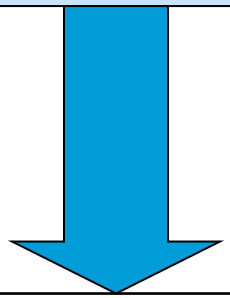
- Moreover, since factorial(n) is a recursion, it acts both as a caller and as a callee. We must store $ra properly so that the function can return correctly when it is a callee.

- Therefore we also need to **store the value of $ra onto the stack**.

# Recursion example (factorial)

- The base case of the recursion is simple.
- If *n* ($a0) is equal to 0 just return with 1 as the result.
- This part of the code is simple because it does not involve further function calls, so there is no need to push (store) the register values onto the stack.

```
1. int factorial (int n){
2.          if  (n == 0)
3.        return 1;
4.    else
5.        return n*factorial ( n - 1 );
6. }
```



```
base_case:
    bne $a0, $zero, recursive_case    # if N>=1, f(N) = N * f(N-1)
    addi $v0, $zero, 1 # return f(0) = 1
    j factorial_exit
     :     :
```

# Recursion example (factorial)

- For the recursive case, we need to do four major things:
  - Push (store) register values (of $a0 and $ra) onto the stack,
  - Call the factorial part of the code for the value of f(n-1),
  - Calculate for the value of n*f(n-1),
  - Pop (retrieve) the register values, and return back to the caller.

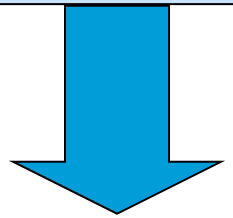
- MIPS will not store the values of $a0 and $ra for you, even if they are to be overwritten.

- In this program, we let the caller function preserve the values of the registers.

# Recursion example (factorial)

- **The MIPS codes on this slide**
  - update the stack pointer to store one additional 32-bit words to the memory,
  - push/store register values of $a0 onto the stack,
  - reduce n by 1,
  - call the factorial codes for the value of f(n-1).

```
1. int factorial (int n){
2.           if  (n == 0)
3.         return 1;
4.    else
5.        return n*factorial ( n - 1 );
6. }
```

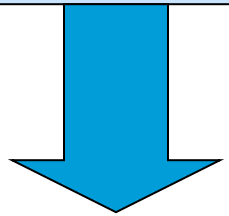


```
:    :
bne $a0, $zero, recursive_case    # if N>=1, f(N) = N * f(N-1)
:    :
recursive_case:
    addi $sp, $sp, -4  # push $a0 (to save value N), caller's role
    sw $a0, 0($sp)
    addi $a0, $a0, -1  # now $a0 = N-1
    jal factorial      # call f(N-1)
:    :
```

# Recursion example (factorial)

- **The MIPS codes on this slide**
  - pop/retrieve the register values of $a0 from the stack,
  - update the stack pointer to free 1 words (occupied by the one register) from the memory,
  - calculate the value n*f(n-1),
  - pop/retrieve the register values of $ra from the stack,

```
1. int factorial (int n){
2.           if  (n == 0 )
3.         return 1;
4.    else
5.        return n*factorial ( n - 1 );
6. }
```



```
:    :
lw $a0, 0($sp)          # pop value N from stack
addi $sp, $sp, 4
mult $a0, $v0           # f(N) = N * f(N-1)
mflo $v0                # now $v0 holds f(N)
:    :
factorial_exit:
lw $ra, 0($sp)          # pop return address $ra
addi $sp, $sp, 4
jr $ra
```

# Recursion example (factorial)

- Download the assembly file factorial.s.

- Load it to MARS and try running it by stepping into the codes. Watch the changes of $ra and $a0.

- Try to understand the flow of the program by challenging yourself. For example, you can remove the codes for storing and retrieving $ra, and challenge yourself with the question "why doesn't the program terminate now?"

- Do ask me questions if you have any doubts.

# Recursion example (factorial)

- **Factorial(N) (part 1)**

```
.data
    prompt: .asciiz "Input an integer N:\n"
    result: .asciiz "Factorial(N) = "

.text
.globl main

main:
    # show prompt
    li          $v0, 4
    la          $a0, prompt
    syscall
    # read x
    li          $v0, 5
    syscall
    # function call
    move        $a0, $v0        #  $a0 contains the read in integer (i.e. N)
    jal       factorial         # call factorial(N)
    move        $t0, $v0        # $t0 = $v0;$v0 contains the result of factorial(N)
    # show prompt
    li          $v0, 4
    la          $a0, result
    syscall
    # print the result
    li          $v0, 1          # system call #1 - print int
    move        $a0, $t0        # $a0 = $t0
    syscall                     # execute
    # return 0
    li          $v0, 10         # $v0 = 10
    syscall
```

# Recursion example (factorial)

- **Factorial(N) (part 2)**

```
factorial:
     addi $sp, $sp, -4   #push $ra, callee's role
     sw $ra, 0($sp)

base_case:
     bne $a0, $zero, recursive_case     # if N>=1, f(N) = N * f(N-1)
     addi $v0, $zero, 1  # return f(0) = 1
     j factorial_exit

recursive_case:
     addi $sp, $sp, -4   # push $a0 (to save value N), caller's role
     sw $a0, 0($sp)
     addi $a0, $a0, -1   # now $a0 = N-1
     jal factorial       # call f(N-1)
                         # when factorial returns, f(N-1) is in $v0
     lw $a0, 0($sp)      # pop value N from stack
     addi $sp, $sp, 4
     mult $a0, $v0       # f(N) = N * f(N-1)
     mflo $v0            # now $v0 holds f(N)

factorial_exit:
     lw $ra, 0($sp)      # pop return address $ra
     addi $sp, $sp, 4
     jr $ra
```

# Fibonacci Sequence

- **The Fibonacci sequence can be expressed recursively as:**

fib(n) = fib(n-1) + fib(n-2)

Where fib(0) = 0 and fib(1) = 1

0, 1, 1, 2, 3, 5, 8, 13, 21,…

- **The corresponding C/C++ codes**

```
1.  int fib(int n){          // n is assumed to be non-negative
2.      if (n<=1)
3.          return n;  // base case reached
4.      else
5.          return fib(n-1) + fib(n-2); // non-base case
6.  }
```

# Exercise (Fibonacci function)

- Using a similar approach as the factorial function to implement a recursive Fibonacci function in MIPS.
- You should prompt the user for the user input value *n*.
- You should store the value of *n* in $a0, and the final result of fib(n) in $v0.
- Whenever necessary, feel free to use additional registers (for storing temporary values).
- If you can't finish it in the lab., make sure you finish it at home!

# Exercise
# (Fibonacci function)

- Things to be aware of :
  - this is a recursive function so make sure you use the stack to preserve register values whenever necessary,
  - this function calls itself for two times in each iteration (fib(n-1) and fib(n-2)), which is not completely the same as the factorial function,
  - beware of the value of ‘n-2’, and make sure your program input will not use -ve n.

# Conclusions

- You have learnt:
  - ❑ the concept of recursion,
  - ❑ the reason for using a stack to implement a recursion in MIPS,
  - ❑ how a recursion can be implemented correctly in MIPS.