COMP2611: Computer Organization

MIPS function and recursion

❑ You will learn the following in this lab:

    ❑ how to use MIPS functions in a program;

    ❑ the concept of recursion;

    ❑ how a recursion can be implemented correctly in MARS.

❑ Try the following example programs in order:

    ❑ <u>functionCall1.s</u> (returning through the address in $ra).

    ❑ <u>functionCall2.s</u> (can't return because $ra has been overwritten).

    ❑ <u>functionCall3.s</u> (preserving the registers).

❑ 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.

factorial(n) = factorial(1) x 2 x 3... x n

factorial(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)).

❑ 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.

❑ 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 <= 1)
3.          return 1;     // base case reached
4.      else
5.          return n*factorial ( n - 1 ); // non-base case
6.  }
```

❑ 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**.

- ❑ The base case of the recursion is simple.
- ❑ If *n* ($a0) is less than or equal to 1 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 <= 1)
3.          return 1;
4.    else
5.        return n*factorial ( n - 1 );
6.  }
```
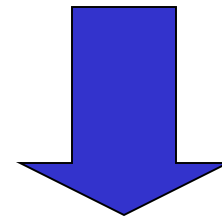
```
base_case:                      # Base Case
slti $t0, $a0, 2                # n<2? (if true n<=1)
addi $t1, $zero, 1              # copy 1 to $t1 for comparison
bne $t0, $t1, non_base_case # n >1, non-base case
addi $v0, $zero, 1             #  n<=1, cal factorial(1)
jr $ra                         # return to the caller
:   :
```

❑ For the non-base 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 over written.

❑ The MIPS codes on this slide

- ❑ update the stack pointer to store two additional 32-bit words to the memory,

- ❑ push/store register values (of $a0 and $ra) 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 <= 1)
3.          return 1;
4.    else
5.        return n*factorial ( n - 1 );
6.  }
```
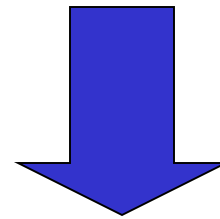
```
:  :
bne $t0, $t1 , non_base_case  # n >1
:  :
non_base_case:    # Non Base Case
addi $sp, $sp, -8   # Move the stack pointer down
sw $a0, 4($sp)      # push $a0=n to the stack
sw $ra, 0($sp)      # push $ra to the stack

addi $a0, $a0, -1  # reduce n by 1
jal factorial         # cal factorial(n-1)
:   :
```

❑ The MIPS codes on this slide

- ❑ pop/retrieve the register values of $a0 and $ra from the stack,
- ❑ update the stack pointer to free 2 words (occupied by the two registers) from the memory,
- ❑ calculate the value n*f(n-1)
- ❑ return back to the caller.

```
1.  int factorial (int n){
2.         if  (n <= 1)
3.             return 1;
4.     else
5.          return n*factorial ( n - 1 );
6.  }
```
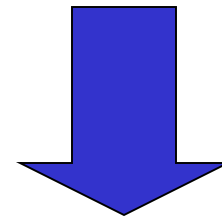
```
:  :
lw $a0, 4($sp)    #pop n after calling factorial
lw $ra, 0($sp)    #pop $ra after calling factorial
addi $sp, $sp, 8  #update the stack pointer
mult $a0, $v0     #calculate n*f(n-1) and
mflo $v0          #and store in $v0

jr $ra            #return back to the caller
:  :
```

❏ Putting everything together we have:

```
.data
msg: .asciiz "Enter the value n:"
.text
.globl main

main:
#print a string
li $v0, 4
la $a0, msg
syscall

#read an integer into $v0
li $v0, 5
syscall

#$v0 contains n, now
add $a0, $v0, $zero
#calculate the factorial(n)
jal factorial

#copy the result to $a0
#output the result and terminate
add $a0, $v0, $zero
li $v0,1
syscall
j end
```

```
factorial:

base_case:                         # Base Case
slti $t0, $a0, 2                   # n<2 (if true n<=1)
addi $t1, $zero , 1
bne $t0, $t1, non_base_case  # n >=2, jump to the part for the non-base case
addi $v0, $zero, 1                #  n<2,   do the calculation for the base case
jr $ra                            # return to the caller


non_base_case:                    # Non Base Case
addi $sp, $sp, -8                 # Move the stack pointer down by 2 words
sw $a0, 4($sp)                    # push $a0=n to the stack
sw $ra, 0($sp)                    # push $ra to the stack

addi $a0, $a0, -1                 # reduce n before calling factorial
jal factorial                     # recursively call factorial to cal factorial(n-1)

lw $a0, 4($sp)                    # pop n=$a0 after the call to factorial(n-1)
lw $ra, 0($sp)                    # pop $ra after the call to factorial(n-1)
addi $sp, $sp, 8                  # update the stack pointer (free 2 words)
multu $a0, $v0                    # calculate n*f(n-1) and store in $v0
mflo $v0                          #
jr $ra                            # return to the caller
end:
```

❑ You have learnt:

  ❑ how to use MIPS functions in a program;

  ❑ we introduced briefly the concept of recursion;
  ❑ we mentioned the reason for using a stack to implement a recursion;
  ❑ we show how a recursion can be correctly implemented.