Recursion

Several slides taken (with modifications) from your textbook (Harris and Harris)

Recursion

You already know everything you need to write a recursive function

The same rules as any other function apply – as long as you follow them carefully, nothing is different

Function Call Summary

- **Caller**
  - Put arguments in `$a0-$a3`
  - Save any needed registers (`$ra`, maybe `$t0-t9`)
  - `jal callee`
  - Restore registers
  - Look for result in `$v0`
- **Callee**
  - Save registers that might be disturbed (`$s0-$s7`)
  - Perform function
  - Put result in `$v0`
  - Restore registers
  - `jr $ra`

Registers

Preserved *Callee-Saved*	Nonpreserved *Caller-Saved*
`$s0-$s7`	`$t0-$t9`
`$ra`	`$a0-$a3`
`$sp`	`$v0-$v1`
stack above `$sp`	**stack below `$sp`**

Recursive Function Call

High-level code

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

Recursive Function Call

MIPS assembly code

```
0x90 factorial:
0x94
0x98
0x9C            addi $t0, $0, 2
0xA0            slt  $t0, $a0, $t0 # a <= 1 ?
0xA4            beq  $t0, $0, else # no: go to else
0xA8            addi $v0, $0, 1    # yes: return 1
0xAC
0xB0            jr   $ra           # return
0xB4      else: addi $a0, $a0, -1  # n = n - 1
0xB8            jal  factorial     # recursive call
0xBC
0xC0
0xC4
0xC8            mul  $v0, $a0, $v0 # n * factorial(n-1)
0xCC            jr   $ra           # return
```

Recursive function call

This code is example of “simple” translation of function to assembly – just following our rules about conditionals and function calls

However, we failed to bear in mind our preserved registers

To see how this comes back to haunt us, consider what happens in previous code when calling `factorial(3)`. In assembly, this is

```
addi $a0, $0, 3
jal factorial
```

Factorial(3)

MIPS assembly code

```
0x90 factorial:
0x94
0x98
0x9C            addi $t0, $0, 2
0xA0            slt  $t0, $a0, $t0 # a <= 1 ?
0xA4            beq  $t0, $0, else # no: go to else
0xA8            addi $v0, $0, 1    # yes: return 1
0xAC
0xB0            jr   $ra           # return
0xB4      else: addi $a0, $a0, -1  # n = n - 1
0xB8            jal  factorial     # recursive call
0xBC
0xC0
0xC4
0xC8            mul  $v0, $a0, $v0 # n * factorial(n-1)
0xCC            jr   $ra           # return
```

$t0 = 0

$a0 = 2
factorial(2)

Factorial(2)

MIPS assembly code

```
0x90 factorial:
0x94
0x98
0x9C            addi $t0, $0, 2
0xA0            slt  $t0, $a0, $t0 # a <= 1 ?
0xA4            beq  $t0, $0, else # no: go to else
0xA8            addi $v0, $0, 1    # yes: return 1
0xAC
0xB0            jr   $ra           # return
0xB4      else: addi $a0, $a0, -1  # n = n - 1
0xB8            jal  factorial     # recursive call
0xBC
0xC0
0xC4
0xC8            mul  $v0, $a0, $v0 # n * factorial(n-1)
0xCC            jr   $ra           # return
```

$t0 = 0

$a0 = 1
factorial(1)

Factorial(1)

MIPS assembly code

```
0x90 factorial:
0x94
0x98
0x9C            addi $t0, $0, 2
0xA0            slt  $t0, $a0, $t0 # a <= 1 ?
0xA4            beq  $t0, $0, else # no: go to else
0xA8            addi $v0, $0, 1    # yes: return 1
0xAC
0xB0            jr   $ra           # return
0xB4      else: addi $a0, $a0, -1  # n = n - 1
0xB8            jal  factorial     # recursive call
0xBC
0xC0
0xC4
0xC8            mul  $v0, $a0, $v0 # n * factorial(n-1)
0xCC            jr   $ra           # return
```

```
$t0 = 1
$v0 = 1
return 1
```

Factorial(2) – after return

MIPS assembly code

```
0x90 factorial:
0x94
0x98
0x9C            addi $t0, $0, 2
0xA0            slt  $t0, $a0, $t0 # a <= 1 ?
0xA4            beq  $t0, $0, else # no: go to else
0xA8            addi $v0, $0, 1    # yes: return 1
0xAC
0xB0            jr   $ra           # return
0xB4      else: addi $a0, $a0, -1  # n = n - 1
0xB8            jal  factorial     # recursive call
0xBC
0xC0
0xC4
0xC8            mul  $v0, $a0, $v0 # n * factorial(n-1)
0xCC            jr   $ra           # return
```

```
$t0 = 0

$a0 = 1
factorial(1)
    # returns 1

$v0 = 1*1
return 1
```

Factorial(3) – after return

MIPS assembly code

```
0x90 factorial:
0x94
0x98
0x9C            addi $t0, $0, 2
0xA0            slt  $t0, $a0, $t0 # a <= 1 ?
0xA4            beq  $t0, $0, else # no: go to else
0xA8            addi $v0, $0, 1    # yes: return 1
0xAC
0xB0            jr   $ra           # return
0xB4      else: addi $a0, $a0, -1  # n = n - 1
0xB8            jal  factorial     # recursive call
0xBC
0xC0
0xC4
0xC8            mul  $v0, $a0, $v0 # n * factorial(n-1)
0xCC            jr   $ra           # return
```

```
$t0 = 0

$a0 = 2
factorial(2)
   # sets $a0=1
   # returns 1

$v0 = 1
jump back to
mul statement –
infinite loop
```

Follow the rules

We ended up with the wrong answer and stuck in an infinite loop

All is not lost! If we use the stack to preserve registers, the rest of the code can stay the same

Follow the rules

First, any time a function calls another function, it needs to store/restore `$ra` using the stack

We did not use any preserved (`$s*`) registers yet, so no need to worry about those

Looking at the code, we made the implicit assumption that $a0 would not change during function calls – but this is wrong!

Follow the rules

We have two options to preserve `$a0`

1. Move it to `$s0`, which will be preserved. We can use `$s0` in the `mul` line without worrying that it has changed. However, this means we *must* add/restore `$s0` on the stack at start and end of function
2. We can add `$a0` to stack directly and restore it from stack after function calls

I tend to prefer the first way – move important things to `$s*` registers and only bother with setup and cleanup once at the very start and end of functions

Your book demonstrates the other way, which is also perfectly fine

Recursive Function Call

MIPS assembly code

```
0x90 factorial:
0x94
0x98
0x9C            addi $t0, $0, 2
0xA0            slt  $t0, $a0, $t0 # a <= 1 ?
0xA4            beq  $t0, $0, else # no: go to else
0xA8            addi $v0, $0, 1    # yes: return 1
0xAC
0xB0            jr   $ra           # return
0xB4      else: addi $a0, $a0, -1  # n = n - 1
0xB8            jal  factorial     # recursive call
0xBC
0xC0
0xC4
0xC8            mul  $v0, $a0, $v0 # n * factorial(n-1)
0xCC            jr   $ra           # return
```

Recursive Function Call

MIPS assembly code

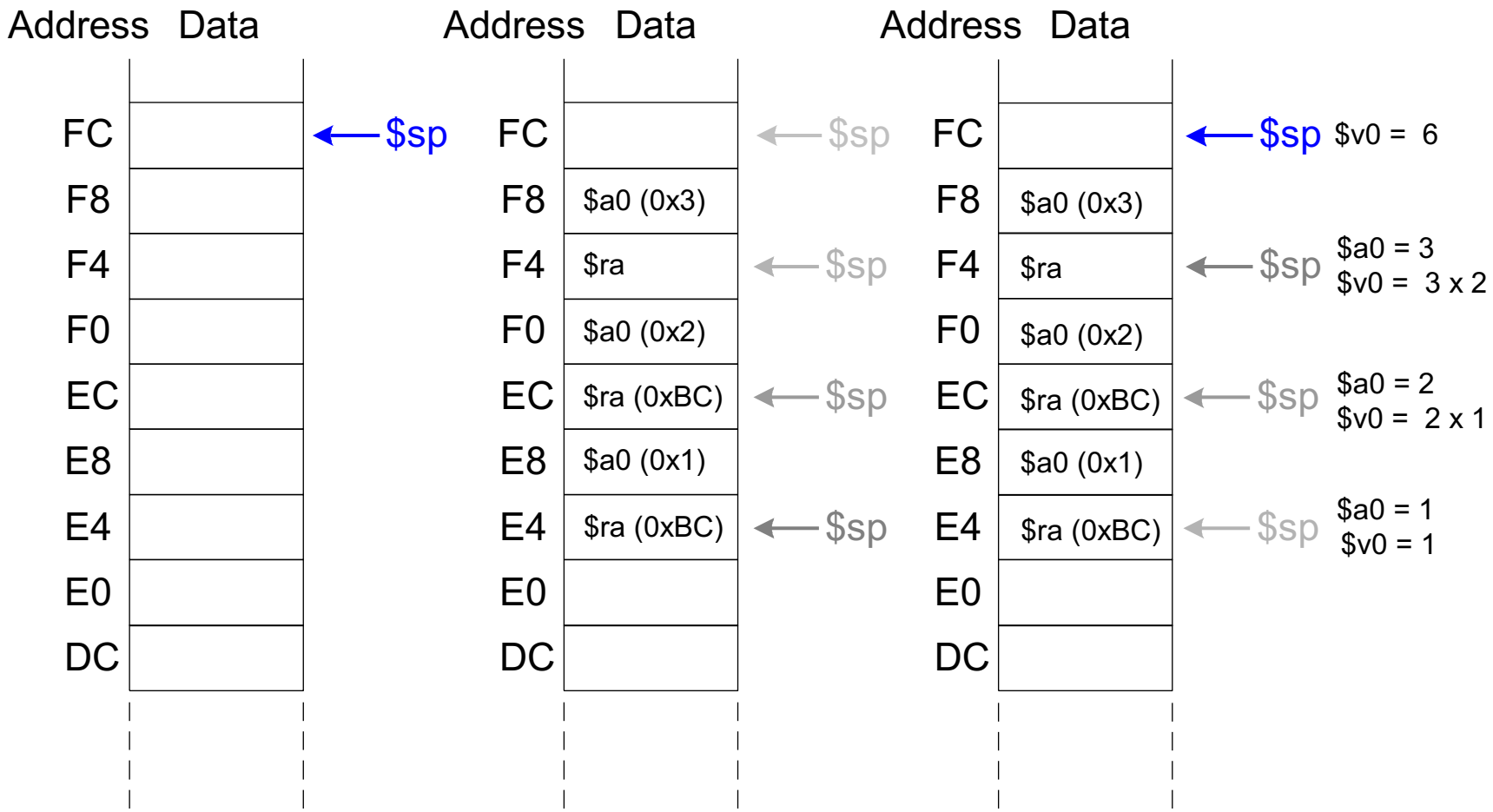
```
0x90 factorial: addi $sp, $sp, -8   # make room
0x94            sw   $a0, 4($sp)    # store $a0
0x98            sw   $ra, 0($sp)    # store $ra
0x9C            addi $t0, $0, 2
0xA0            slt  $t0, $a0, $t0  # a <= 1 ?
0xA4            beq  $t0, $0, else  # no: go to else
0xA8            addi $v0, $0, 1     # yes: return 1
0xAC            addi $sp, $sp, 8    # restore $sp
0xB0            jr   $ra            # return
0xB4       else: addi $a0, $a0, -1  # n = n - 1
0xB8            jal  factorial      # recursive call
0xBC            lw   $ra, 0($sp)    # restore $ra
0xC0            lw   $a0, 4($sp)    # restore $a0
0xC4            addi $sp, $sp, 8    # restore $sp
0xC8            mul  $v0, $a0, $v0  # n * factorial(n-1)
0xCC            jr   $ra            # return
```

Recursive Function Call

MIPS assembly code

```
0x90 factorial: addi $sp, $sp, -8   # make room
0x94            sw   $a0, 4($sp)    # store $a0
0x98            sw   $ra, 0($sp)    # store $ra
0x9C            addi $t0, $0, 2
0xA0            slt  $t0, $a0, $t0  # a <= 1 ?
0xA4            beq  $t0, $0, else  # no: go to else
0xA8            addi $v0, $0, 1     # yes: return 1
0xAC            addi $sp, $sp, 8    # restore $sp
0xB0            jr   $ra            # return
0xB4       else: addi $a0, $a0, -1  # n = n - 1
0xB8            jal  factorial      # recursive call
0xBC            lw   $ra, 0($sp)    # restore $ra
0xC0            lw   $a0, 4($sp)    # restore $a0
0xC4            addi $sp, $sp, 8    # restore $sp
0xC8            mul  $v0, $a0, $v0  # n * factorial(n-1)
0xCC            jr   $ra            # return
```

Stack During Recursive Call



Recursion and stack

Remember – anything not in a preserved register should be assumed to change during function call

If you need something after the function call, put it in preserved register or store it on your stack frame and restore it after the call

Note about preserved registers

Why are preserved registers preserved?

Once again, convention!

There is nothing physically different about `$s*` registers that makes them automatically restore after function calls

They preserve their values from a caller's perspective because you make them do so – you restore their values from the stack before returning

If you modify an `$s*` register and do not put it back, you are violating your contract with calling function, and that function will not work