Stack

Several slides taken from your textbook (Harris and Harris)

Functions

Recall that we know how to write functions:

- Pass control to elsewhere in code (`jal`)
- Pass arguments (`$a0 – $a3`)
- Return back to original place in code (`jr $ra`)
- Return result of function (`$v0 – $v1`)

One other important aspect of functions is that they do not change values outside their scope. If caller has important value in, e.g., `$s0`, it should not need to worry that callee will overwrite `$s0`

Functions

How is a function supposed to do any work without changing registers?

It can’t!

Instead, functions will “clean up after themselves”

If function changes a register, function will change it back before returning

- How does it know what to change register back to? (What was original value in register?)

Functions

If function changes a register, function will change it back before returning

Only places to store values in assembly are registers and memory

Big idea:

- If function needs to use register, it first writes original value to memory
- Before returning, function loads value from memory back into register

Special section of memory used for this purpose -- **stack**

Input Arguments & Return Value

MIPS assembly code

```
# $s0 = y

main:
   ...
   addi $a0, $0, 2     # argument 0 = 2
   addi $a1, $0, 3     # argument 1 = 3
   addi $a2, $0, 4     # argument 2 = 4
   addi $a3, $0, 5     # argument 3 = 5
   jal  diffofsums     # call Function
   add  $s0, $v0, $0   # y = returned value
   ...

# $s0 = result
diffofsums:
   add $t0, $a0, $a1   # $t0 = f + g
   add $t1, $a2, $a3   # $t1 = h + i
   sub $s0, $t0, $t1   # result = (f + g) - (h + i)
   add $v0, $s0, $0    # put return value in $v0
   jr  $ra             # return to caller
```

Input Arguments & Return Value

MIPS assembly code

```
# $s0 = result
diffofsums:
  add $t0, $a0, $a1   # $t0 = f + g
  add $t1, $a2, $a3   # $t1 = h + i
  sub $s0, $t0, $t1   # result = (f + g) - (h + i)
  add $v0, $s0, $0    # put return value in $v0
  jr  $ra             # return to caller
```

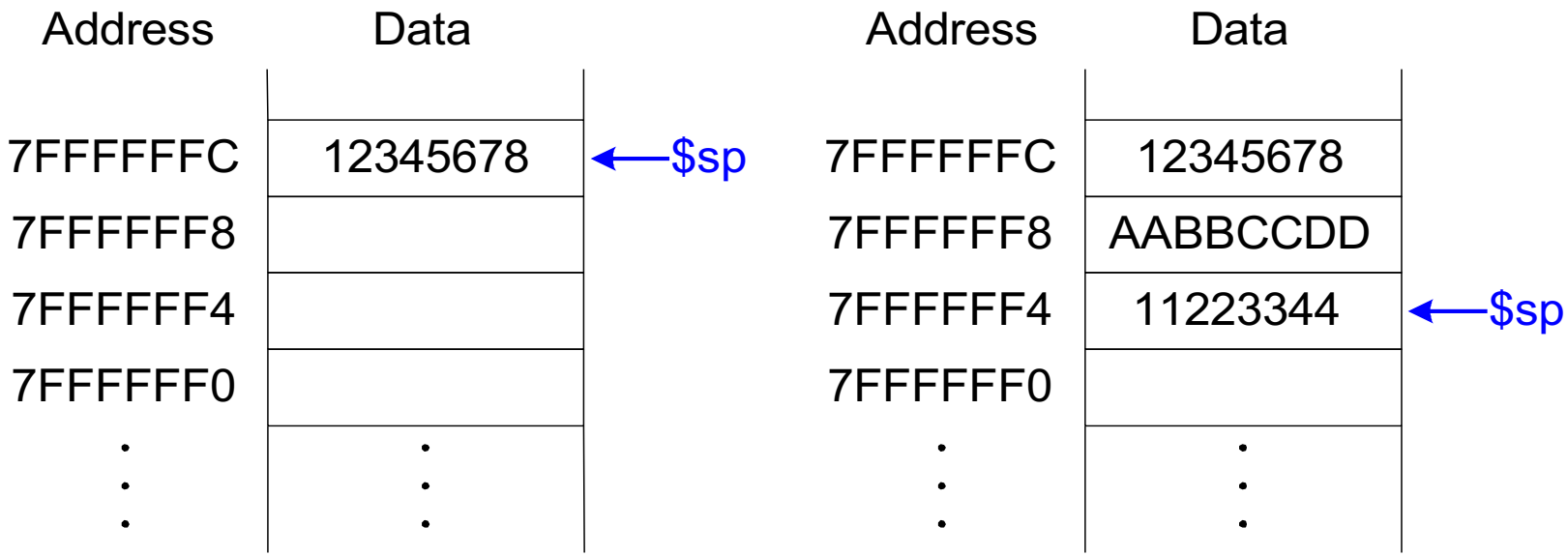
- `diffofsums` overwrote 3 registers: `$t0`, `$t1`, `$s0`
- `diffofsums` can use *stack* to temporarily store registers

The Stack

- Memory used to temporarily save variables
- Like stack of dishes, last-in-first-out (LIFO) queue
- ***Expands*:** uses more memory when more space needed
- ***Contracts*:** uses less memory when the space is no longer needed

The Stack

- Grows down (from higher to lower memory addresses)
- Stack pointer: `$sp` points to top of the stack



The Stack

Notice that stack pointer stores *address*, not value

Stack pointer tells us which address is currently the “top” (actually the lowest address) of the stack

Like many other things in MIPS, there is nothing special about the area of memory that we call the stack – by convention, we agree to access that memory relative to `$sp` rather than directly by address

How Functions use the Stack

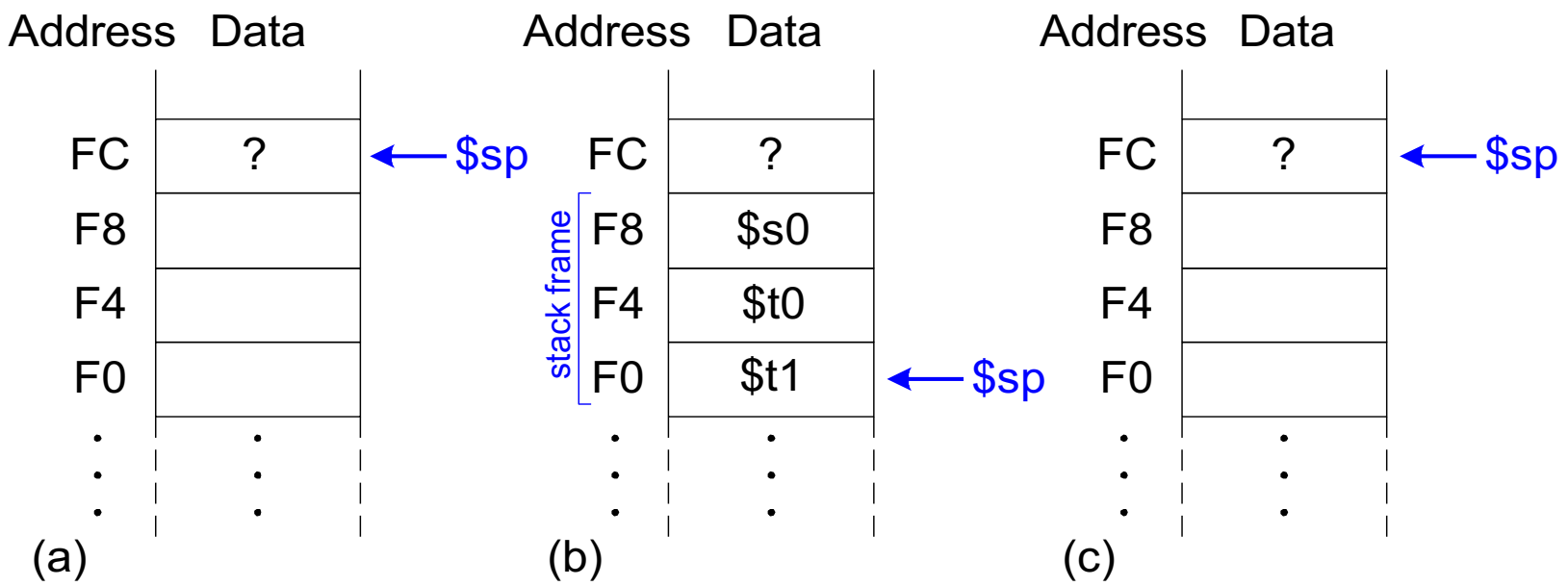
- Called functions must have no unintended side effects
- But `diffofsums` overwrites 3 registers: `$t0`, `$t1`, `$s0`

```
# MIPS assembly
# $s0 = result
diffofsums:
  add $t0, $a0, $a1   # $t0 = f + g
  add $t1, $a2, $a3   # $t1 = h + i
  sub $s0, $t0, $t1   # result = (f + g) - (h + i)
  add $v0, $s0, $0    # put return value in $v0
  jr  $ra             # return to caller
```

Storing Register Values on the Stack

```
# $s0 = result
diffofsums:
  addi $sp, $sp, -12   # make space on stack
                       # to store 3 registers
  sw   $s0, 8($sp)     # save $s0 on stack
  sw   $t0, 4($sp)     # save $t0 on stack
  sw   $t1, 0($sp)     # save $t1 on stack
  add  $t0, $a0, $a1   # $t0 = f + g
  add  $t1, $a2, $a3   # $t1 = h + i
  sub  $s0, $t0, $t1   # result = (f + g) - (h + i)
  add  $v0, $s0, $0    # put return value in $v0
  lw   $t1, 0($sp)     # restore $t1 from stack
  lw   $t0, 4($sp)     # restore $t0 from stack
  lw   $s0, 8($sp)     # restore $s0 from stack
  addi $sp, $sp, 12    # deallocate stack space
  jr   $ra             # return to caller
```

The stack during `diffofsums` Call



Functions and the stack

Look at the pattern from two slides ago until it makes sense – this is the essence of how we use the stack

- Make space
- Store what we are going to change
- Do work
- Restore what we changed
- Free space

What would happen if we did not free space at the end?

Functions and the stack

Here is another common mistake:

```
addi $sp, $sp, -8
sw $s0, 4($sp)
sw $s1, 8($sp)
...
addi $sp, $sp, 8
```

Why is this wrong?

Preserved registers

Storing values on the stack takes time

That time is wasted if caller was not using those values

MIPS programmers have (yet another) agreement – anything in `$t*` registers can be overwritten by function

Caller must not have anything it needs stored in temporary registers – either move them to preserved (`$s*`) registers or to stack before calling function

Registers

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

Preserved registers

```
# put some values in $t0, $a0, and $s0
jal someFunction
# when control returns here:
#    $t0 and $a0 are “random”
#    $s0 has not changed
```

Preserved registers

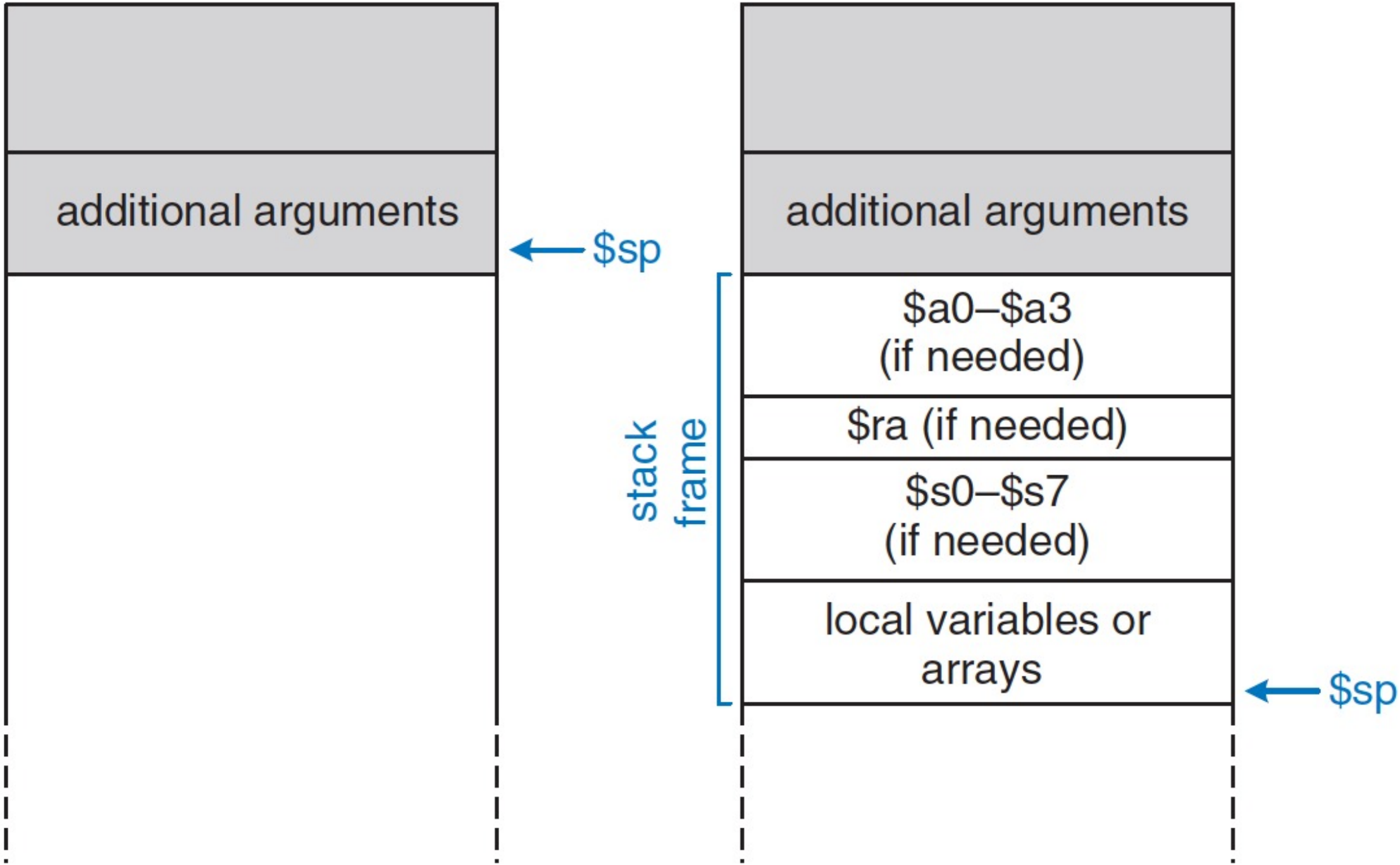
In addition to `$t*`, don't forget that calling `jal` will also change `$ra`

If I am a function that calls another function, I need my original `$ra` to return to whoever called me

Any function that plans to call another function *must* put `$ra` in its stack frame first

Multiple Function Calls

```
proc1:
   addi $sp, $sp, -4     # make space on stack
   sw   $ra, 0($sp)      # save $ra on stack
   jal  proc2
   ...
   lw   $ra, 0($sp)      # restore $s0 from stack
   addi $sp, $sp, 4      # deallocate stack space
   jr  $ra               # return to caller
```


additional arguments
$sp
additional arguments
$a0–$a3
(if needed)
$ra (if needed)
$s0–$s7
(if needed)
local variables or
arrays
$sp
stack
frame

Stack – why bother?

Already discussed why we need to preserve register values somewhere, but why is the stack special?

Consider how all of our memory accesses are relative to stack pointer

Stack is self-regulating – as long as everyone follows the rules, keeping track of who owns which memory is simple

- Consider: how would we keep track of who owns which memory otherwise? Who or what would know that information?

Each function worries only about its own stack frame and ignores the rest of the stack

Storing Saved Registers on the Stack

```
# $s0 = result
diffofsums:
  addi $sp, $sp, -4   # make space on stack to
                       # store one register
  sw  $s0, 0($sp)     # save $s0 on stack
                      # no need to save $t0 or $t1
  add $t0, $a0, $a1   # $t0 = f + g
  add $t1, $a2, $a3   # $t1 = h + i
  sub $s0, $t0, $t1   # result = (f + g) - (h + i)
  add $v0, $s0, $0    # put return value in $v0
  lw  $s0, 0($sp)     # restore $s0 from stack
  addi $sp, $sp, 4    # deallocate stack space
  jr  $ra             # return to caller
```

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`