Johns Hopkins Engineering

Module 6: MIPS Functions & Call Stack

EN605.204: Computer Organization

JOHNS HOPKINS
WHITING SCHOOL
of ENGINEERING

C Function

- When we call the add() function from our main method, we pass control to the add() function
  - We are "caller", add() is the "callee"
  - Must keep track of where we left off so we know where to return control
  - Go off, do some work, pick back up

```c
int add(int x, int y)
{
  int z = x + y;
  return z;
}

int main(void) {
  int sum = add(3, 4);
  printf("Sum: %i", sum);
  return 0;
}

// output: Sum: 7
```

C Function

- Our function does 4 things:
  - Declares an integer 'z'
  - Takes 2 parameters: x, y
  - Adds x and y, stores the sum in 'z'
  - Returns result and control to caller
- Question:
  - Where are the function and its data fields stored in memory?
- Answer: the "call stack"!

```c
int add(int x, int y)
{
  int z = x + y;
  return z;
}

int main(void) {
  int sum = add(3, 4);
  printf("Sum: %i", sum);
  return 0;
}

// output: Sum: 7
```

What is a "stack"?

A stack is a data structure that stores data and supports to operations:

- "push": add an element
- "pop" remove the top element

What is a "call stack"?

- An in-memory stack where we store:
  - Function parameters
  - Local variables
  - Return address

```c
int add(int x, int y)
{
    int z = x + y;
    return z;
}

int main(void) {
    int sum = add(3, 4);
    printf("Sum: %i", sum);
    return 0;
}

// output: Sum: 7
```

Stack Frames

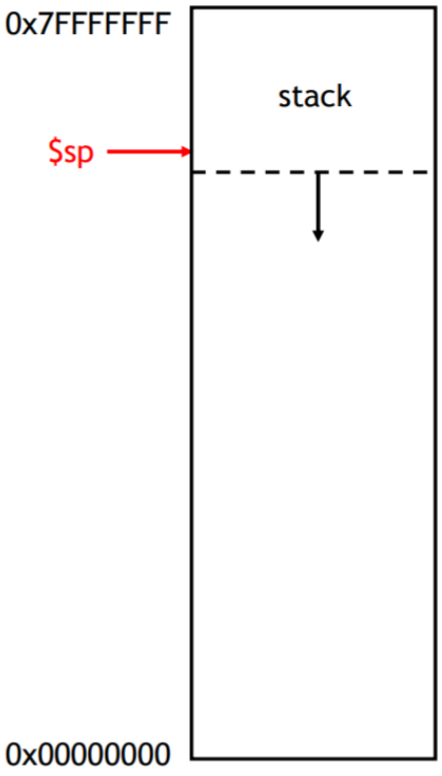
- Stack frame is the grouping of all data a function call puts on the stack
  - Sometimes called an "activation record"
  - Stack pointer ($sp) = top of stack
- Remember:
  - .text section stores code
    - begins at 0x00400000
  - .data section stores data
    - begins at 0x10010000

...
z (local variable)
y (argument)
x (argument)
Return address
...

Pushing to the Stack

- Stack memory addresses grow "down"
- To "push" a single, 4-byte element (integer) onto the stack, subtract 4 bytes from the value stored in $sp:

`subi $sp, $sp, 4` `sw $t1, 0($sp)`	`addi $sp, $sp, -4` `sw $t1, 0($sp)`



MIPS Function Conventions

- Who is responsible for saving values?
- "Convetion" says:
  - The "caller" is responsible for saving:
    - $v0, $v1, $a0-$a3, and $t0-$t9
  - The "callee" is responsible for saving:
    - $s0-$s7, $ra
- "Convention" not enforced by assembler!

PRESERVED ACROSS A CALL?
N.A.
No
No
No
No
Yes
No
No
Yes
Yes
Yes
No

MIPS Addition Function

```
.text
main:
    addi $t0, $zero, 3
    addi $t1, $zero, 4

    add $a0, $0, $t0   # copy t0 into a0
    add $a1, $0, $t1  # copy t1 into a1
    jal addition      # call function - stores return address in ra!!!
    add $t3, $0, $v0   # copy returned value into t3

    # print the value returned from the function
    add $a0, $zero, $t3
    addi $v0, $zero, 1
    syscall

    # exit the program
    addi $v0, $zero, 10
    syscall

addition:
    addi $sp, $sp, -4 # create 4 bytes on the stack
    sw $s0, 0($sp) # push t0 onto the stack

    add $s0, $a0, $a1     # Procedure Body
    add $v0, $0, $s0      # Result

    lw $s0, 0($sp) # restore $s0
    addi $sp, $sp, 4 # pop the element off the stack
    jr $ra # return execution to caller
```

Calling a Function: 'jal'

- 'jal': stores the address of the next instruction, PC + 4, in $ra before the jump
  - This is how me know where we left off
  - We can see that the 'jal' instruction is at 0x00400010, so 'jal' will store 0x00400014 in $ra

```
jal addition        # call function - stores return address in ra!!!
add $t3, $0, $v0     # copy returned value into t3
```

☐	0x00400010	0x0c10000b	jal 0x0040002c
☐	0x00400014	0x00025820	add $11,$0,$2

Preserving Register on the Stack

```
addition:
    addi $sp, $sp, -4 # create 4 bytes on the stack
    sw $s0, 0($sp) # push t0 onto the stack
```

- We want to use $s0 to store the result of the addition
  - By convention, the callee preserves $s0
    - Allocate 4 bytes on the stack by suubtracting 4 from the current stack pointer (stack grows down)
      - If top of stack was at $sp = 0x40, first element is stored at $sp = 0x3C
    - Use 'sw' to save current $s0 on the stack

Memory Contents	Memory address ($sp)
(None) top of stack	0x40
contents of s0	0x3C

$sp = $sp - 4

Function Body

```
add $s0, $a0, $a1
add $v0, $0, $s0
```

- $s0 is now safe to use (but we will have to restore it)
- Since $a0 and $a1 are our input registers, we add them
- $v0, $v1 commonly used to return values, move result there

Restore Register and Return Control to Caller

```
lw $s0, 0($sp) # restore $s0
addi $sp, $sp, 4 # pop the element off the stack
jr $ra # return execution to caller
```

- lw: "restore $s0" - retrieve the value at $sp and put it back into $s0
- addi: add 4 bytes to $sp to remove the element from stack
- jr: jump to the address stored in $ra
  - jal put the address of the next instruction into $ra for us!

Memory Contents	Memory address ($sp)
(None) top of stack	0x40
contents of s0	0x3C

$sp = $sp + 4 = 0x40

current $sp = 0x3C

$s0 = "contents of s0"

	0x00400010	0x0c10000b	jal 0x0040002c
	0x00400014	0x00025820	add $11,$0,$2

Completing the Program

```
jal addition      # call function - stores return address in ra!!!
add $t3, $0, $v0   # copy returned value into t3

# print the value returned from the function
add $a0, $zero, $t3
addi $v0, $zero, 1
syscall

# exit the program
addi $v0, $zero, 10
syscall
```

- We "return" to the add instruction with the result now in $v0
- Save the contents of $v0 into $t3
- Move the contents of $t3 into $a0 to make the syscall and print the result
- Exit the program with a syscall 10