CSE 210: Computer Architecture

Lecture 13: Pointers

Stephen Checkoway

Slides from Cynthia Taylor

Announcements

- Problem Set 4 Due Friday
- Lab 3 available

Today's Class

- Finish the stack
- Discuss working with arrays
  - Needed for Lab 4
- Discuss pointers
  - We'll see how far we get!

CS History: Rózsa Péter

- Born in Budapest in 1905
- Almost quit mathematics to be a poet when she discovered a math result she was working on had already been discovered
- Was persuaded to return to math and started working on recursion
- Received a PhD in 1935
- Wasn't allowed to teach between 1939 and 1945 because of Jewish laws in Hungary
  - During this time she wrote a book titled "Playing with Infinity: Mathematical Explorations and Excursions" for lay readers – it has been translated into a dozen languages
- Helped found the field of recursive function theory
- Began applying recursion to computers in the 1950s

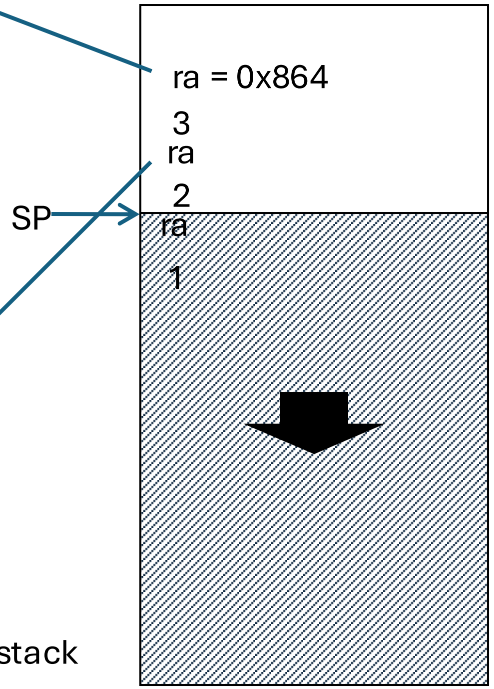
Non-Leaf Procedure Example

```
fact:
    addi $sp, $sp, -8     # adjust stack for 2 items
    sw   $ra, 4($sp)      # save return address
    sw   $a0, 0($sp)      # save argument
    slti $t0, $a0, 2      # test for n < 2
    beq  $t0, $zero, L1
    addi $v0, $zero, 1    # if so, result is 1
    addi $sp, $sp, 8      #   pop 2 items from stack
    jr   $ra              #   and return
L1: addi $a0, $a0, -1     # else decrement n
    jal  fact             # recursive call
    lw   $a0, 0($sp)      # restore original n
    lw   $ra, 4($sp)      #   and return address
    addi $sp, $sp, 8      # pop 2 items from stack
    mul  $v0, $a0, $v0    # multiply to get result
    jr   $ra              # and return
```

We will return to

$ra = L1 + 8
$a0 = 1
$v0 = 1
$t0 = 1

```
fact:
        addi $sp, $sp, -8     # adjust stack for 2 items
        sw   $ra, 4($sp)      # save return address
        sw   $a0, 0($sp)      # save argument
        slti $t0, $a0, 2      # test for n < 2
        beq  $t0, $zero, L1
        addi $v0, $zero, 1    # if so, result is 1
        addi $sp, $sp, 8      #   pop 2 items from stack
PC →    jr   $ra              #   and return
    L1: addi $a0, $a0, -1     # else decrement n
        jal  fact             # recursive call
        lw   $a0, 0($sp)      # restore original n
        lw   $ra, 4($sp)      #   and return address
        addi $sp, $sp, 8      # pop 2 items from stack
        mul  $v0, $a0, $v0    # multiply to get result
        jr   $ra              # and return
```

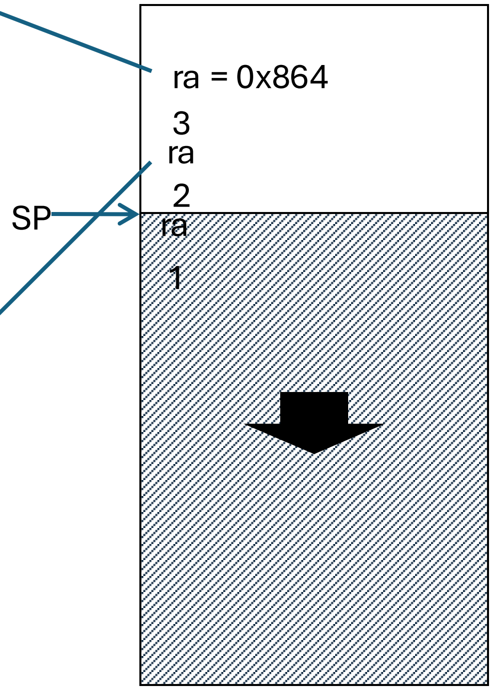


A. L1 + 8, because it in $ra
B. L1 + 8, because it's the most recent value on the stack
C. 0x864, because it's the top value on the stack
D. fact, because it's the procedure call
E. None of the above

fact

$ra = L1 + 8
$a0 = 2
$v0 = 1
$t0 = 1

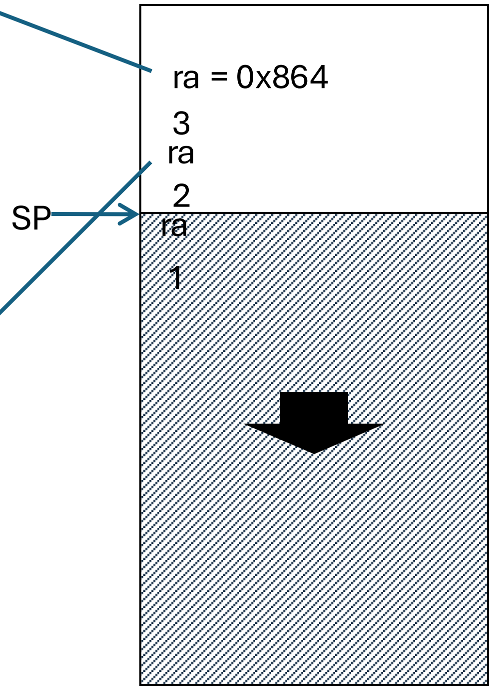
```
fact:
        addi $sp, $sp, -8     # adjust stack for 2 items
        sw   $ra, 4($sp)      # save return address
        sw   $a0, 0($sp)      # save argument
        slti $t0, $a0, 2      # test for n < 2
        beq  $t0, $zero, L1
        addi $v0, $zero, 1    # if so, result is 1
        addi $sp, $sp, 8      #   pop 2 items from stack
        jr   $ra              #   and return
    L1: addi $a0, $a0, -1     # else decrement n
        jal  fact             # recursive call
PC ->   lw   $a0, 0($sp)      # restore original n
        lw   $ra, 4($sp)      #   and return address
        addi $sp, $sp, 8      # pop 2 items from stack
        mul  $v0, $a0, $v0    # multiply to get result
        jr   $ra              # and return
```



fact

$ra = L1 + 8
$a0 = 2
$v0 = 1
$t0 = 1

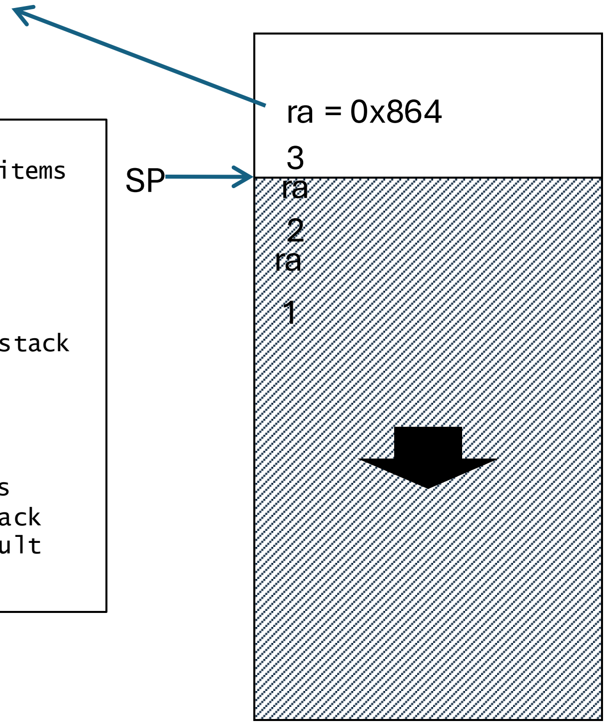
```
fact:
        addi $sp, $sp, -8     # adjust stack for 2 items
        sw   $ra, 4($sp)      # save return address
        sw   $a0, 0($sp)      # save argument
        slti $t0, $a0, 2      # test for n < 2
        beq  $t0, $zero, L1
        addi $v0, $zero, 1    # if so, result is 1
        addi $sp, $sp, 8      #   pop 2 items from stack
        jr   $ra              #   and return
    L1: addi $a0, $a0, -1     # else decrement n
        jal  fact             # recursive call
        lw   $a0, 0($sp)      # restore original n
PC →    lw   $ra, 4($sp)      #   and return address
        addi $sp, $sp, 8      # pop 2 items from stack
        mul  $v0, $a0, $v0    # multiply to get result
        jr   $ra              # and return
```



fact

$ra = L1 + 8
$a0 = 2
$v0 = 1
$t0 = 1

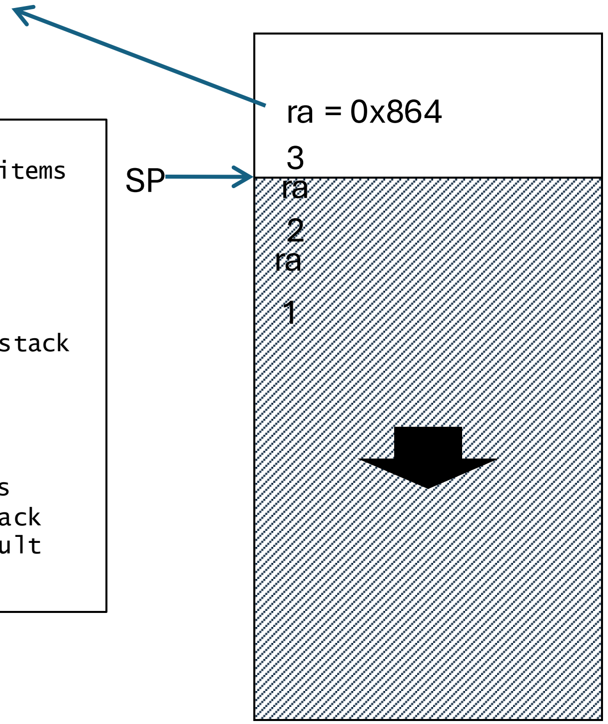
```
fact:
        addi $sp, $sp, -8     # adjust stack for 2 items
        sw   $ra, 4($sp)      # save return address
        sw   $a0, 0($sp)      # save argument
        slti $t0, $a0, 2      # test for n < 2
        beq  $t0, $zero, L1
        addi $v0, $zero, 1    # if so, result is 1
        addi $sp, $sp, 8      #   pop 2 items from stack
        jr   $ra              #   and return
    L1: addi $a0, $a0, -1     # else decrement n
        jal  fact             # recursive call
        lw   $a0, 0($sp)      # restore original n
        lw   $ra, 4($sp)      #   and return address
PC ---> addi $sp, $sp, 8      # pop 2 items from stack
        mul  $v0, $a0, $v0    # multiply to get result
        jr   $ra              # and return
```



fact

$ra = L1 + 8
$a0 = 2
$v0 = 2
$t0 = 1

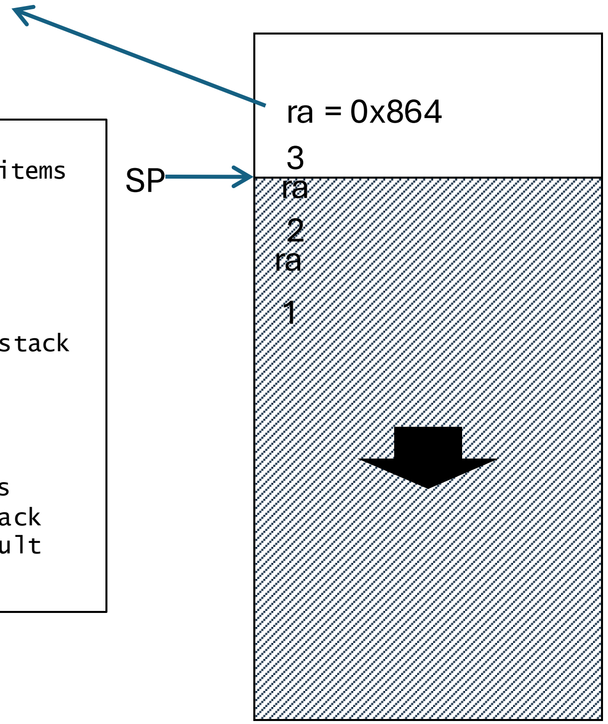
```
fact:
        addi $sp, $sp, -8     # adjust stack for 2 items
        sw   $ra, 4($sp)      # save return address
        sw   $a0, 0($sp)      # save argument
        slti $t0, $a0, 2      # test for n < 2
        beq  $t0, $zero, L1
        addi $v0, $zero, 1    # if so, result is 1
        addi $sp, $sp, 8      #   pop 2 items from stack
        jr   $ra              #   and return
    L1: addi $a0, $a0, -1     # else decrement n
        jal  fact             # recursive call
        lw   $a0, 0($sp)      # restore original n
        lw   $ra, 4($sp)      #   and return address
        addi $sp, $sp, 8      # pop 2 items from stack
PC      mul  $v0, $a0, $v0    # multiply to get result
        jr   $ra              # and return
```



fact

$ra = L1 + 8
$a0 = 2
$v0 = 2
$t0 = 1

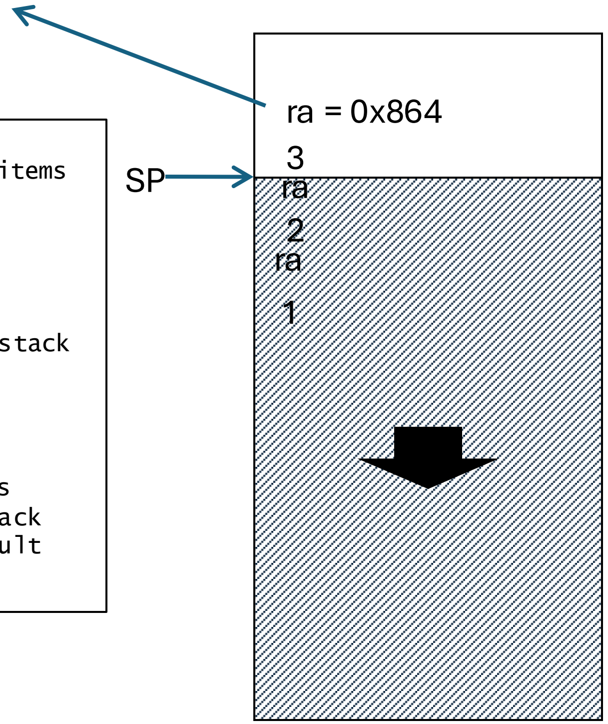
```
fact:
       addi $sp, $sp, -8     # adjust stack for 2 items
       sw   $ra, 4($sp)      # save return address
       sw   $a0, 0($sp)      # save argument
       slti $t0, $a0, 2      # test for n < 2
       beq  $t0, $zero, L1
       addi $v0, $zero, 1    # if so, result is 1
       addi $sp, $sp, 8      #   pop 2 items from stack
       jr   $ra              #   and return
   L1: addi $a0, $a0, -1     # else decrement n
       jal  fact             # recursive call
       lw   $a0, 0($sp)      # restore original n
       lw   $ra, 4($sp)      #   and return address
       addi $sp, $sp, 8      # pop 2 items from stack
       mul  $v0, $a0, $v0    # multiply to get result
PC →   jr   $ra              # and return
```



fact

$ra = L1 + 8
$a0 = 3
$v0 = 2
$t0 = 1

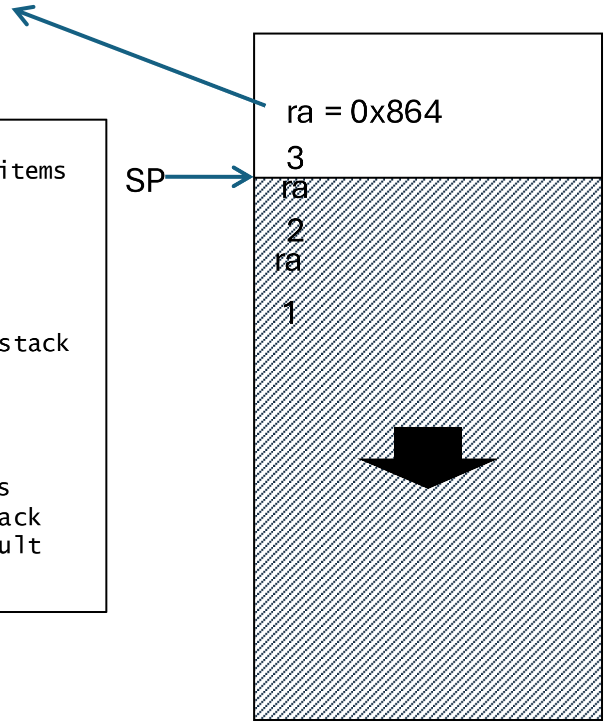
```
fact:
        addi $sp, $sp, -8     # adjust stack for 2 items
        sw   $ra, 4($sp)      # save return address
        sw   $a0, 0($sp)      # save argument
        slti $t0, $a0, 2      # test for n < 2
        beq  $t0, $zero, L1
        addi $v0, $zero, 1    # if so, result is 1
        addi $sp, $sp, 8      #   pop 2 items from stack
        jr   $ra              #   and return
    L1: addi $a0, $a0, -1     # else decrement n
        jal  fact             # recursive call
PC →    lw   $a0, 0($sp)      # restore original n
        lw   $ra, 4($sp)      #   and return address
        addi $sp, $sp, 8      # pop 2 items from stack
        mul  $v0, $a0, $v0    # multiply to get result
        jr   $ra              # and return
```



fact

$ra = 0x864
$a0 = 3
$v0 = 2
$t0 = 1

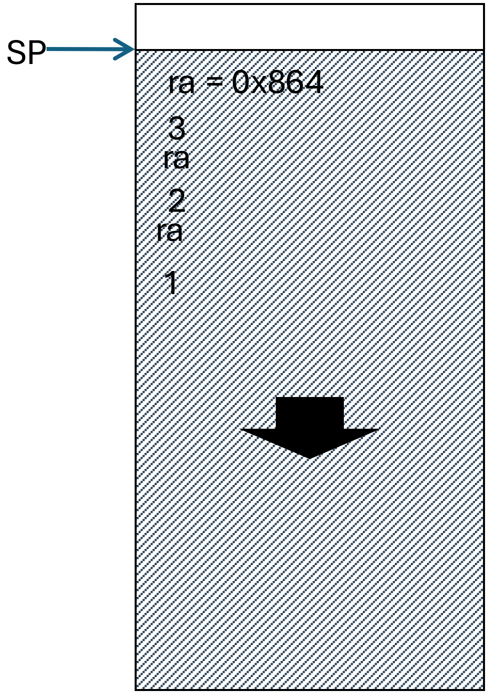
```
fact:
        addi $sp, $sp, -8      # adjust stack for 2 items
        sw   $ra, 4($sp)       # save return address
        sw   $a0, 0($sp)       # save argument
        slti $t0, $a0, 2       # test for n < 2
        beq  $t0, $zero, L1
        addi $v0, $zero, 1     # if so, result is 1
        addi $sp, $sp, 8       #   pop 2 items from stack
        jr   $ra               #   and return
    L1: addi $a0, $a0, -1      # else decrement n
        jal  fact              # recursive call
        lw   $a0, 0($sp)       # restore original n
PC ---> lw   $ra, 4($sp)       #   and return address
        addi $sp, $sp, 8       # pop 2 items from stack
        mul  $v0, $a0, $v0     # multiply to get result
        jr   $ra               # and return
```



fact

$ra = 0x864
$a0 = 3
$v0 = 2
$t0 = 1

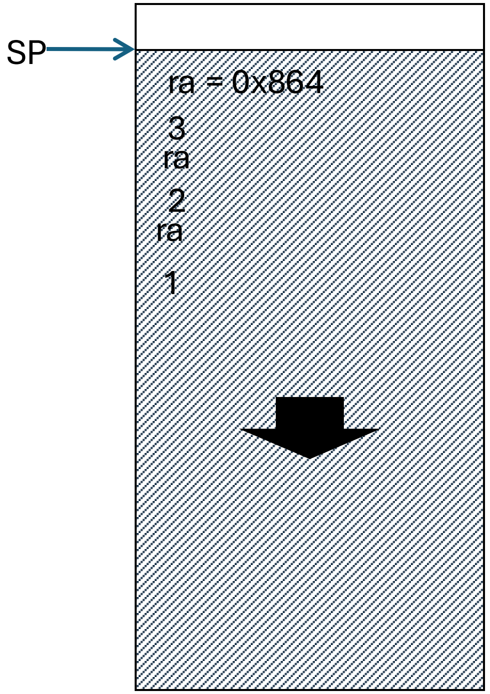
```
fact:
        addi $sp, $sp, -8     # adjust stack for 2 items
        sw   $ra, 4($sp)      # save return address
        sw   $a0, 0($sp)      # save argument
        slti $t0, $a0, 2      # test for n < 2
        beq  $t0, $zero, L1
        addi $v0, $zero, 1    # if so, result is 1
        addi $sp, $sp, 8      #   pop 2 items from stack
        jr   $ra              #   and return
    L1: addi $a0, $a0, -1     # else decrement n
        jal  fact             # recursive call
        lw   $a0, 0($sp)      # restore original n
        lw   $ra, 4($sp)      #   and return address
PC----> addi $sp, $sp, 8      # pop 2 items from stack
        mul  $v0, $a0, $v0    # multiply to get result
        jr   $ra              # and return
```



fact

$ra = 0x864
$a0 = 3
$v0 = 6
$t0 = 1

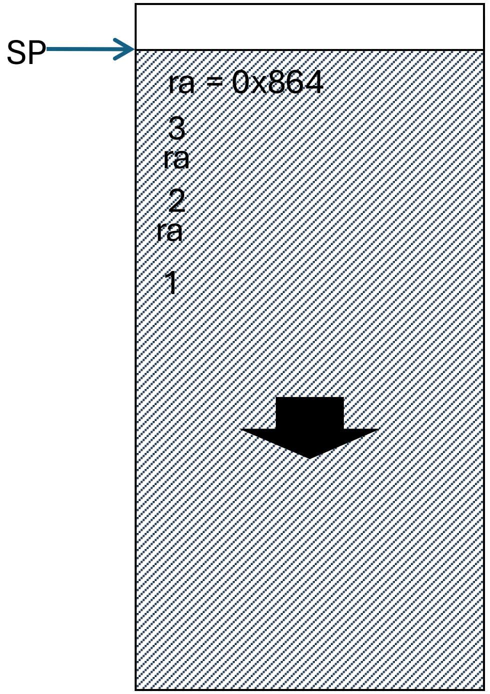
```
fact:
        addi $sp, $sp, -8     # adjust stack for 2 items
        sw   $ra, 4($sp)      # save return address
        sw   $a0, 0($sp)      # save argument
        slti $t0, $a0, 2      # test for n < 2
        beq  $t0, $zero, L1
        addi $v0, $zero, 1    # if so, result is 1
        addi $sp, $sp, 8      #   pop 2 items from stack
        jr   $ra              #   and return
    L1: addi $a0, $a0, -1     # else decrement n
        jal  fact             # recursive call
        lw   $a0, 0($sp)      # restore original n
        lw   $ra, 4($sp)      #   and return address
        addi $sp, $sp, 8      # pop 2 items from stack
PC →    mul  $v0, $a0, $v0    # multiply to get result
        jr   $ra              # and return
```



fact

$ra = 0x864
$a0 = 3
$v0 = 6
$t0 = 1

```
fact:
        addi $sp, $sp, -8     # adjust stack for 2 items
        sw   $ra, 4($sp)      # save return address
        sw   $a0, 0($sp)      # save argument
        slti $t0, $a0, 2      # test for n < 2
        beq  $t0, $zero, L1
        addi $v0, $zero, 1    # if so, result is 1
        addi $sp, $sp, 8      #   pop 2 items from stack
        jr   $ra              #   and return
    L1: addi $a0, $a0, -1     # else decrement n
        jal  fact             # recursive call
        lw   $a0, 0($sp)      # restore original n
        lw   $ra, 4($sp)      #   and return address
        addi $sp, $sp, 8      # pop 2 items from stack
        mul  $v0, $a0, $v0    # multiply to get result
PC→     jr   $ra              # and return
```



Why store registers relative to the stack pointer, rather than at some set memory location?

A. Saves space.

B. Easier to figure out where we stored things.

C. Functions won’t overwrite each other’s saves.

D. None of the above

Questions on the Stack, Spilling and Filling, etc?

Assembler directives

- Instructions to the assembler
  - .data / .text / .rodata / .bss are used to switch between global (mutable) data, executable code, read-only data, and uninitialized data in the output
  - .word x allocates space for 4 bytes with value x
  - .space n allocates n bytes of space
  - .asciiz “string” writes a 0-terminated string at that location

Review: Arrays!

- How do we declare a 10-word array in our data section?

- Could do

```
.data
  x1:           .word    0
  x2:           .word0
  x3:           .word0
  ...
  x10:          .word0
```

Review: Declaring an Array

- Instead, just declare a big chunk of memory

```
.data
arr: .space   40
```

```
.data
arr:    .space 40

.text
    li          $t0, 0
    addi        $t1, $t0, 10
    la          $s0, arr
loop:
    beq                 $t0,
$t1, end
    What goes here?
    addi        $t0, $t0, 1
    j                   loop
end:
```

```c
int i;
for (i = 0; i < 10; i++){
        arr[i] = i;
}
```

```
sw                      $t0,
$t1($s0)
```

A

```
add                     $t2, $s0,
$t1
sw                      $t0, 0($t2)
```

B

```
sw                      $t0,
0($s0)
addi        $s0, $s0, 4
```

C

D. More than one of the above

E. None of the above

But what if we don’t know how big the array will be before runtime?

sbrk system call

- Allocates memory and returns its address in $v0
- Amount of memory is specified in bytes in $a0
- Used by malloc, new

System Calls

- Syscalls (when we need OS intervention)
  - I/O (print/read stdout/file)
  - Exit (terminate)
  - Get system time
  - Random values

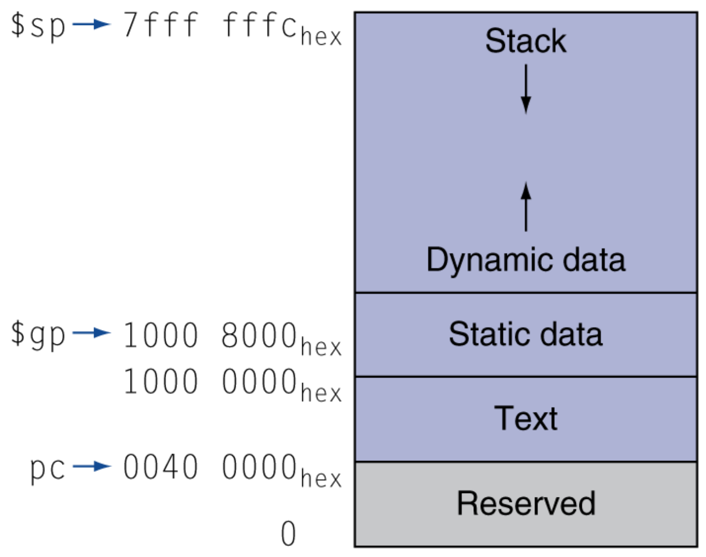
System Calls Review

- How to use:
  - Put syscall number into register $v0
  - Load arguments into argument registers
  - Issue syscall instruction
  - Retrieve return values
- Example (allocate $t4 bytes of memory with sbrk):

```
li       $v0, 9   # sbrk system call number
move     $a0, $t4 # allocate $t4 bytes of mem
syscall
move     $s0, $v0 # $s0 holds a pointer to mem
```

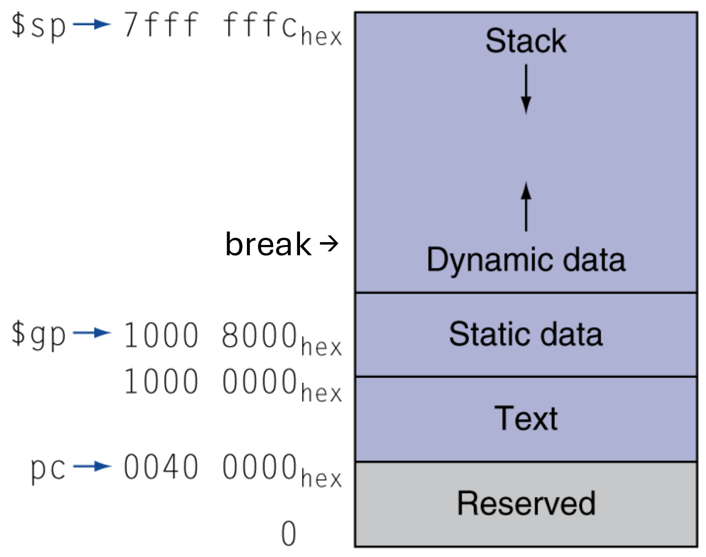
sbrk allocates memory from which region?

A. Stack
B. Dynamic data
C. Static data
D. Text
E. Reserved



What about freeing memory?

- Some operating systems maintain a “program break” which controls the size of the dynamic data
- sbrk requests the OS increment/decrement the break
- malloc()/free() carve the dynamic data up into chunks which the application can use and maintain lists of free chunks
- Freeing memory adds the chunk to a “free list”
- When more memory is needed, the break is changed



High Level Concepts, Low Level Language

- So far we have looked at basic MIPS instructions, control flow, and memory addressing

- But how do we build things like objects and structs in MIPS?

Java Parameter Passing

In main:

```
int i = 10;
increase_i(i);
System.out.print(i);
```

What gets printed?

A. 10
B. 20
C. Runtime error
D. None of the above

```
public static void increase_i(int val) {
    val = val + 10;
}
```

Java Parameter Passing

```
class wrapper{
  int i=0;
}
```

What gets printed?

A. 10
B. 20
C. Runtime error
D. None of the above

In main:

```
wrapper w = new wrapper();
w.i = 10;
add_wrapper(w);
System.out.print(w.i);
```

```
static void add_wrapper(wrapper w){
      w.i = w.i+10;
}
```

Java Parameter Passing

- Java is “Call By Value”
  - Passes a copy of the value, not a pointer/reference to it
  - Explains behavior in first question

- When the argument is an Object, it copies *pointers* (references) to the variables inside the object
  - Explains behavior in second question

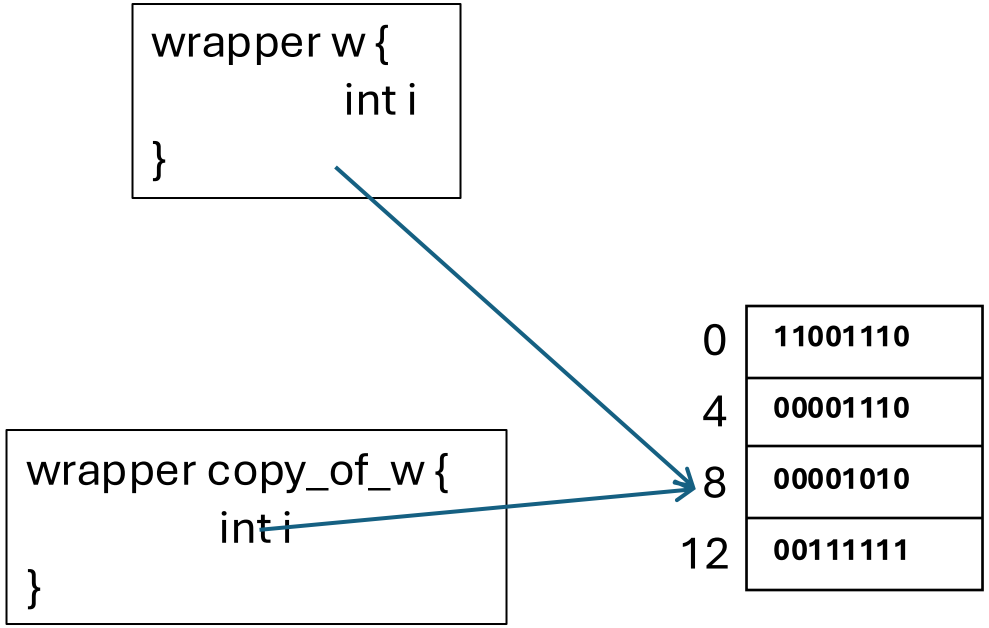
Java Argument Passing

Copying a Primitive Data Type

int x = 10

int copy_of_x = 10

Copying an Object



Pointers in C

```
int x = 7;

int *y;  //y is a pointer
y = &x;  //y = address of x

int z = *y; // dereferences y
```

- We can do “call by value” using x
  - Pass in a copy of the value of x
- We can emulate “call by reference” using y
  - Pass in a pointer to memory location of x

C

In main:

```
int var = 10;
int *pvar = &var;

double_it(pvar);
printf("%d\n", *pvar);
```

What gets printed?

A. 10
B. 20
C. Runtime error
D. None of the above

```
void double_it(int *p){
    *p = *p * 2;
}
```

C

In main:

```
int var = 10;
int *pvar = &var;

double_it(var);
printf("%d\n", var);
```

What gets printed?

A. 10
B. 20
C. Runtime error
D. None of the above

```
void double_it(int p){
    p = p * 2;
}
```

Rust

- Rust is call-by-value, and behaves similarly to Java by default

- Rust also lets us create explicit pointers (references) to both primitives and objects, like C

In Assembly

- If $t0 is an int, it holds the actual data

- if $t0 is a pointer, it holds the address of where the data is in memory

```
while (curr != tail){
    display(curr);
    curr = curr.next;
}
```

```
class Node {
    int val;    // offset = 0
    Node next;  // offset = 4
}
```

The high level equivalent of `lw $s0, 4($s0)` is

A. display(curr);

B. curr = curr.next;

C. curr != tail

D. There is no high level equivalent.

```
        la      $s0, head
        la      $s1, tail
top:    beq     $s0, $s1, out
        move    $a0, $s0
        jal     display
        lw      $s0, 4($s0)
        j       top
out:
```

Iterate Through A Linked List

```
      la   $s0, head
      la   $s1, tail
top: beq  $s0, $zero, out
      move $a0, $s0
      jal  display
      lw   $s0, 4($s0)
      j    top
out:
```

head is at address 0x1000
tail is at address 0x102C

$s0
$s1
$a0

Address	Value
0x1000	5
0x1004	0x1014
0x1008	
0x100C	7
0x1010	0x1024
0x1014	8
0x1018	0x100C
0x101C	
0x1020	
0x1024	13
0x1028	0x102C
0x102C	

The Heap

- To allocate memory on the heap, use the sbrk syscall – this takes a number of bytes, and returns the address of the allocated memory
- Now use sw, lw, etc to use that allocated memory

Create a New Node

```
new_node:
    li   $a0, 8
    li   $v0, 9
    syscall        # sbrk(8)
    # $v0 now holds the new node
    # Fill in the data fields in the new node
    sw   $zero, 0($v0) # node.val = 0
    sw   $zero, 4($v0) # node.next = null
    jr   $ra
```

Inserting a new node after the current node in the list

```
class Node {
      int val;      // offset = 0
      Node next;    // offset = 4
}
```

```
lw     $t1, 4($s0)
sw     $t1, 4($s1)
sw     $s1, 4($s0)
```

Assume $s0 holds current's base address and $s1 holds newnode's base address

	lw $t1, 4($s0)	**sw $t1, 4($s1)**	**sw $s1, 4($s0)**
A	$t1 = current.next	current.next = newnode.next	newnode.next = current
B	$t1 = current.previous	newnode.previous = current.previous	current.previous = newnode
C	$t1 = current.next	newnode.next = current.next	current.next = newnode
D	$t1 = newnode.next	newnode.next = current.next	current.next = newnode

Reading

- Next lecture: Digital Logic
  - 3.2