The MIPS stack: recap

- The stack grows downward in terms of memory addresses.
- The address of the top element of the stack is stored (by convention) in the "stack pointer" register, $sp.

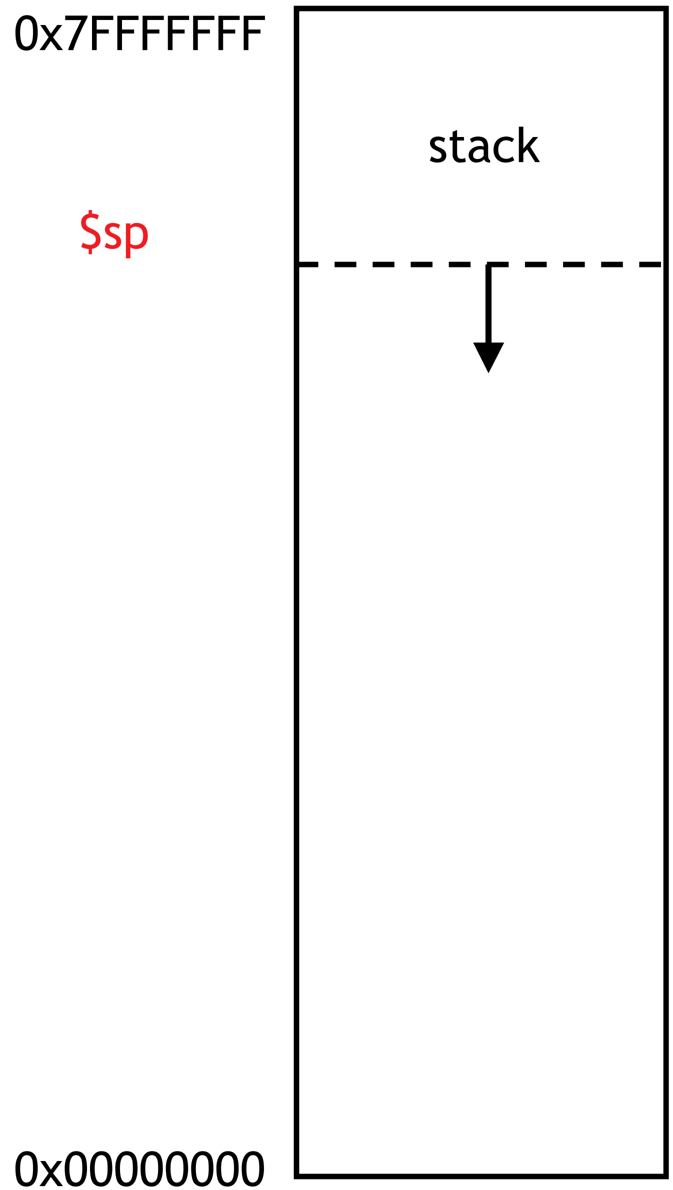
  `0($sp), 1($sp),` … are "used" locations

  `-1($sp), -2($sp),` … are "free"
- MIPS does not provide "push" and "pop" instructions. Instead, they must be done explicitly by the programmer.
- "push" `$t0` simulated with:

```
sub $sp, $sp, 4   # $t0 needs 4 bytes
sw  $t0, 0($sp)   # write to stack
```

- "pop" `$t0` simulated with:

```
lw   $t0, 0($sp)   # read stack top
addi $sp, $sp, 4   # free 4 bytes
```



Using the stack: MP 2's `main`

- Performs a `jal`, so must save `$ra` before and restore it afterwards

```
main:
  addi   $sp, $sp, -4          # grow stack
  sw     $ra, 0($sp)           # save callee-saved register $ra

  jal    iterTraverse          # call your function

  lw     $ra, 0($sp)           # restore $ra
  addi   $sp, $sp, 4           # shrink stack
  jr     $ra
```

A `jal` to a function
Will cause a malfunction
If `ra` is not on the stack

And caller-saved regs
Should be handled like eggs
They're junk when the function comes back

And lastly, no messes
In each of the esses
Make sure you restore 'em - don't slack!

Practice with pointers: Linked Lists

- Linked lists are implemented in C using structures as follows:

```
struct node {
  int   data;    // data field
  node* next;    // pointer to next node in list
}
```

- If p is a *pointer* to a node, then

  *p is the node itself

  (*p).data is located at address p in memory

  (*p).next is located at address p + sizeof(data) in memory

- If p points to the last node in the list, (*p).next == NULL.
- If t0 points to a node, then the statement t0 = (*t0).next makes t0 point to the next node in the list.
- The above statement can be translated into MIPS as:

```
lw   $t0, 4($t0)
```

- Translate the following C statements into MIPS:

```
(*t0).data = (*t1).data;
(*t0).next = (*t1).next;
```

MIPS functions to traverse lists

- Translate this into MIPS:

```
void printList(node* p) {
  while(p != NULL) {   // NULL == 0
    print((*p).data);
    p = (*p).next;
  }
}
```

```
# BAD code!
printList:
  beq  $a0, $0, PL_done
  # need to save $ra, $a0
  lw   $a0, 0($a0)
  jal  print
  # need to restore $a0
  lw   $a0, 4($a0)
  j    printList
  # need to restore $ra
PL_done:

  jr   $ra
```

```
# GOOD code
printList:
  beq  $a0, $0, PL_done
  addi $sp, $sp, -8
  sw   $ra, 0($sp)
  sw   $a0, 4($sp)
  lw   $a0, 0($a0)
  jal  print
  lw   $ra, 0($sp)
  lw   $a0, 4($sp)
  addi $sp, $sp, 8
  lw   $a0, 4($a0)
  j    printList

PL_done:
  jr   $ra
```

A more efficient solution

```
printList:
  addi  $sp, $sp, -8
  sw    $ra, 0($sp)
  sw    $s0, 4($sp)
  move  $s0, $a0
PL_loop:
  beq   $s0, $0, PL_done         # loop body has
  lw    $a0, 0($s0)              # only the 5
  jal   print                    # “necessary”
  lw    $s0, 4($s0)              # instructions
  j     PL_loop
PL_done:
  lw    $ra, 0($sp)
  lw    $s0, 4($sp)
  addi  $sp, $sp, 8
  jr    $ra
```

Recursive list traversal

- Translate this into MIPS:

```
void printList(node* p) {
  if(p != NULL) {
    print((*p).data);
    printList((*p).next);
  }
}
```

```
# BAD code
printList:
  beq  $a0, $0, PL_done

  lw   $a0, 0($a0)
  jal  print

  lw   $a0, 4($a0)
  jal  printList

PL_done:

  jr   $ra
```

```
# Correct, but messy
printList:
  beq  $a0, $0, PL_done
  addi $sp, $sp, -8
  sw   $ra, 0($sp)
  sw   $a0, 4($sp)
  lw   $a0, 0($a0)
  jal  print
  lw   $a0, 4($sp)
  lw   $a0, 4($a0)
  jal  printList
  lw   $ra, 0($sp)
  addi $sp, $sp, 8

PL_done:
  jr   $ra
```

A simple way to do recursion

```
recursive(args) {
  if(base_condition) {
    // base case stuff
    return;
  }
  else {  // !base_condition
    // recursive case
  }
}
```

```
recursive:
  b_not_base_condition recursive_case

  # base case stuff
  jr   $ra

recursive_case:
  # grow stack to save $ra (at least)
    ...
  jal  recursive
    ...
  # shrink stack to restore $ra + other stuff
  jr   $ra
```

Recursive list traversal

Translate this into MIPS:

```
int printList(node* p) {
  if(p == NULL)
    return 0;
  else {
    print((*p).data);
    return 1 + printList((*p).next);
  }
}
```

```
# Recursive solution
printList:
  bne  $a0, $0, recursive_case
  move $v0, $0
  jr   $ra

recursive_case:
  addi $sp, $sp, -8
  sw   $ra, 0($sp)
  sw   $a0, 4($sp)

  lw   $a0, 0($a0)
  jal  print

  lw   $a0, 4($sp)
  lw   $a0, 4($a0)
  jal  printList
  addi $v0, $v0, 1

  lw   $ra, 0($sp)
  addi $sp, $sp, 8

  jr   $ra
```

Recursive list reverse-traversal

- Translate this into MIPS:

```
void printListReverse(node* p) {
     if(p == NULL)
       printNewline();
     else {
       printListReverse((*p).next);
       print((*p).data);
     }
   }
```

```
printListReverse:
  bne  $a0, $0, PL_recursive
  addi $sp, $sp, -4
  sw   $ra, 0($sp)
  jal  printNewline
  lw   $ra, 0($sp)
  addi $sp, $sp, 4
  jr   $ra

PL_recursive:
  addi $sp, $sp, -8
  sw   $ra, 0($sp)
  sw   $a0, 4($sp)
  lw   $a0, 4($a0)
  jal  printListRecursive
  lw   $a0, 4($sp)
  lw   $a0, 0($a0)
  jal  print
  lw   $ra, 0($sp)
  addi $sp, $sp, 8
  jr   $ra
```