MIPS MEMORY ACCESS AND FUNCTION CALLS

LECTURE 09-2

JIM FIX, REED COLLEGE CS2-S20

TODAY'S PLAN

WE'LL LOOK MORE AT MEMORY ACCESS IN MIPS...

- REVIEW LOAD/STORE INSTRUCTIONS, SOLUTIONS TO LAB 09 EXERCISES
- EXAMINE ARRAY AND STRUCT REPRESENTATION AND ACCESS
- WE'LL USE LOAD/STORE AT AN **OFFSET** FROM AN ADDRESS
- LINKED LIST CONSTRUCTION AND TRAVERSAL
- WE'LL LOOK AT CALL STACK LAYOUT AND CALLING CONVENTIONS

SOLUTION TO LAB 09 EXERCISE 1

- This performs the division of $t1 by $t2. The quotient is put in $t0.

```
1.  divide:
2.    li      $t0,0
3.  divide_loop:
4.    ble     $t1, $t2, done
5.    sub     $t1, $t1, $t2
6.    addi    $t0, $t0,1
7.    b       divide_loop
8.  done:
9.    li      $v0, 1
10.   move    $a0, $t0
11.   syscall
12.   li      $v0, 1
13.   move    $a0, $t1
14.   syscall
```

SOLUTION TO LAB 09 EXERCISE 1

- This performs the division of $t1 by $t2. The quotient is put in $t0.

```
1.  divide:
2.    li      $t0,0
3.  divide_loop:
4.    ble     $t1, $t2, done
5.    sub     $t1, $t1, $t2    # Subtract divisor from the dividend.
6.    addi    $t0, $t0,1
7.    b       divide_loop
8.  done:
9.    li      $v0, 1
10.   move    $a0, $t0
11.   syscall
12.   li      $v0, 1
13.   move    $a0, $t1
14.   syscall
```

SOLUTION TO LAB 09 EXERCISE 1

- This performs the division of $t1 by $t2. The quotient is put in $t0.

```
1.  divide:
2.    li      $t0,0
3.  divide_loop:
4.    ble     $t1, $t2, done
5.    sub     $t1, $t1, $t2   # Subtract divisor from the dividend.
6.    addi    $t0, $t0,1      # Count the number of subtractions.
7.    b       divide_loop
8.  done:
9.    li      $v0, 1
10.   move    $a0, $t0
11.   syscall
12.   li      $v0, 1
13.   move    $a0, $t1
14.   syscall
```

SOLUTION TO LAB 09 EXERCISE 1

- This performs the division of $t1 by $t2. The quotient is put in $t0.

```
1.  divide:
2.    li      $t0,0                 # Initialize the count to 0.
3.  divide_loop:
4.    ble     $t1, $t2, done
5.    sub     $t1, $t1, $t2   # Subtract divisor from the dividend.
6.    addi    $t0, $t0,1      # Count the number of subtractions.
7.    b       divide_loop
8.  done:
9.    li      $v0, 1
10.   move    $a0, $t0
11.   syscall
12.   li      $v0, 1
13.   move    $a0, $t1
14.   syscall
```

SOLUTION TO LAB 09 EXERCISE 1

- This performs the division of $t1 by $t2. The quotient is put in $t0.

```
1.  divide:
2.    li      $t0,0                 # Initialize the count to 0.
3.  divide_loop:
4.    ble     $t1, $t2, done        # Stop when the dividend is smaller.
5.    sub     $t1, $t1, $t2         # Subtract divisor from the dividend.
6.    addi    $t0, $t0,1            # Count the number of subtractions.
7.    b       divide_loop
8.  done:
9.    li      $v0, 1
10.   move    $a0, $t0
11.   syscall
12.   li      $v0, 1
13.   move    $a0, $t1
14.   syscall
```

SOLUTION TO LAB 09 EXERCISE 1

- This performs the division of $t1 by $t2. The quotient is put in $t0.

```
1.  divide:
2.    li      $t0,0             # Initialize the count to 0.
3.  divide_loop:
4.    ble     $t1, $t2, done    # Stop when the dividend is smaller.
5.    sub     $t1, $t1, $t2     # Subtract divisor from the dividend.
6.    addi    $t0, $t0,1        # Count the number of subtractions.
7.    b       divide_loop
8.  done:
9.    li      $v0, 1            # Output the quotient.
10.   move    $a0, $t0
11.   syscall
12.   li      $v0, 1
13.   move    $a0, $t1
14.   syscall
```

SOLUTION TO LAB 09 EXERCISE 1

- This performs the division of $t1 by $t2. The quotient is put in $t0.

```
1.  divide:
2.    li      $t0,0             # Initialize the count to 0.
3.  divide_loop:
4.    ble     $t1, $t2, done    # Stop when the dividend is smaller.
5.    sub     $t1, $t1, $t2     # Subtract divisor from the dividend.
6.    addi    $t0, $t0,1        # Count the number of subtractions.
7.    b       divide_loop
8.  done:
9.    li      $v0, 1            # Output the quotient.
10.   move    $a0, $t0
11.   syscall
12.   li      $v0, 1            # Output the remainder.
13.   move    $a0, $t1
14.   syscall
```

MAKE A LOWERCASE C STRING ALL CAPS

- The code below modifies the bytes of a character string at **`string_ptr`**.

```
1.      la      $s0, string_ptr
2.  loop:
3.      lb      $t0,($s0)    # Fetch the next character.
4.      beqz    $t0,done     # See if it's the null character.
5.                           # If it's not,
6.      addi    $t0,$t0,-32 # change the character's code.
7.      sb      $t0,($s0)    # Store that change.
8.
9.      addi    $s0,$s0,1
10.     b       loop
11. done:
```

MAKE A LOWERCASE C STRING ALL CAPS

- The code below modifies the bytes of a character string.

```
1.      la      $s0, string_ptr
2.  loop:
3.      lb      $t0,($s0)     # Fetch the next character.
4.      beqz    $t0,done      # See if it's the null character.
5.                            # If it's not,
6.      addi    $t0,$t0,-32   # change the character's code.
7.      sb      $t0,($s0)     # Store that change.
8.
9.      addi    $s0,$s0,1
10.     b       loop
11. done:
```

- These lines get a character from the string, modify it, then store the change.

MAKE A LOWERCASE C STRING ALL CAPS

- The code below modifies the bytes of a character string.

```
1.      la      $s0, string_ptr
2.  loop:
3.      lb      $t0,($s0)    # Fetch the next character.
4.      beqz    $t0,done     # See if it's the null character.
5.                           # If it's not,
6.      addi    $t0,$t0,-32 # change the character's code.
7.      sb      $t0,($s0)    # Store that change.
8.
9.      addi    $s0,$s0,1
10.     b       loop
11. done:
```

- This line advances the pointer to the next character in the string.

MAKE A LOWERCASE C STRING ALL CAPS

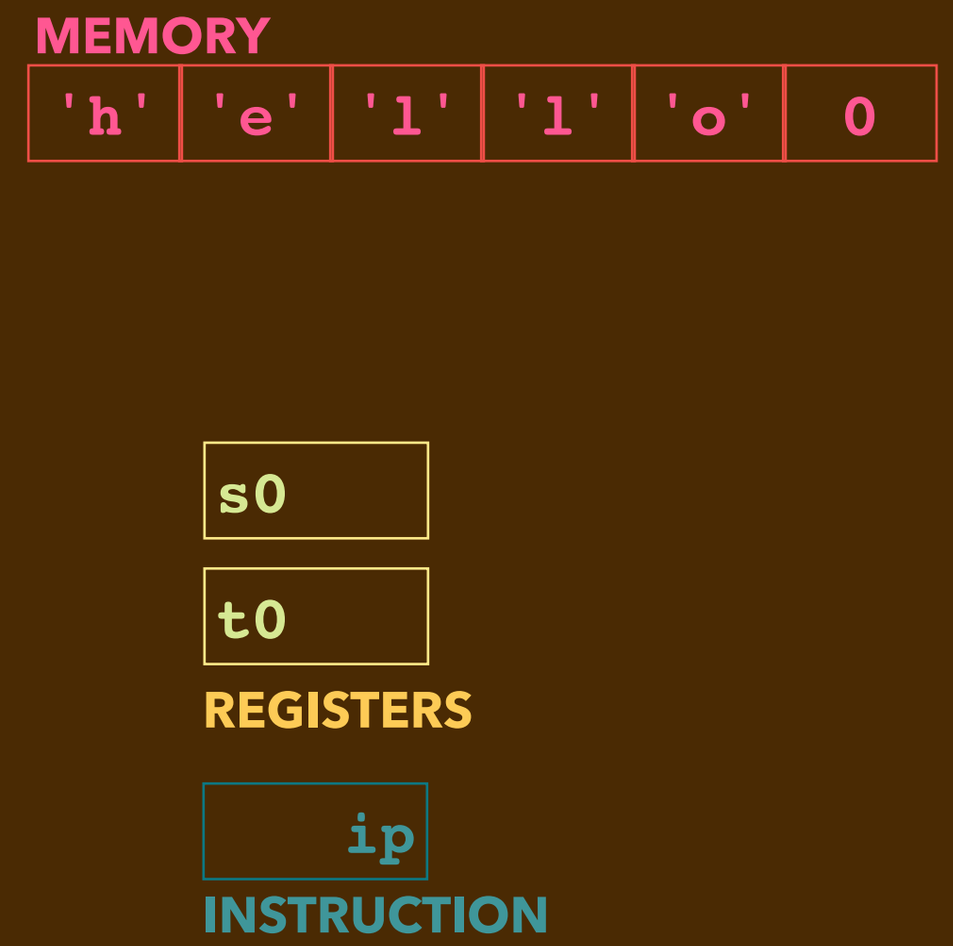
- The code below modifies the bytes of a character string.

```
1.      la      $s0, string_ptr
2.  loop:
3.      lb      $t0,($s0)    # Fetch the next character.
4.      beqz    $t0,done     # See if it's the null character.
5.                           # If it's not,
6.      addi    $t0,$t0,-32 # change the character's code.
7.      sb      $t0,($s0)    # Store that change.
8.
9.      addi    $s0,$s0,1
10.     b       loop
11. done:
```

- This line checks whether we've hit the end of the string, the null character.

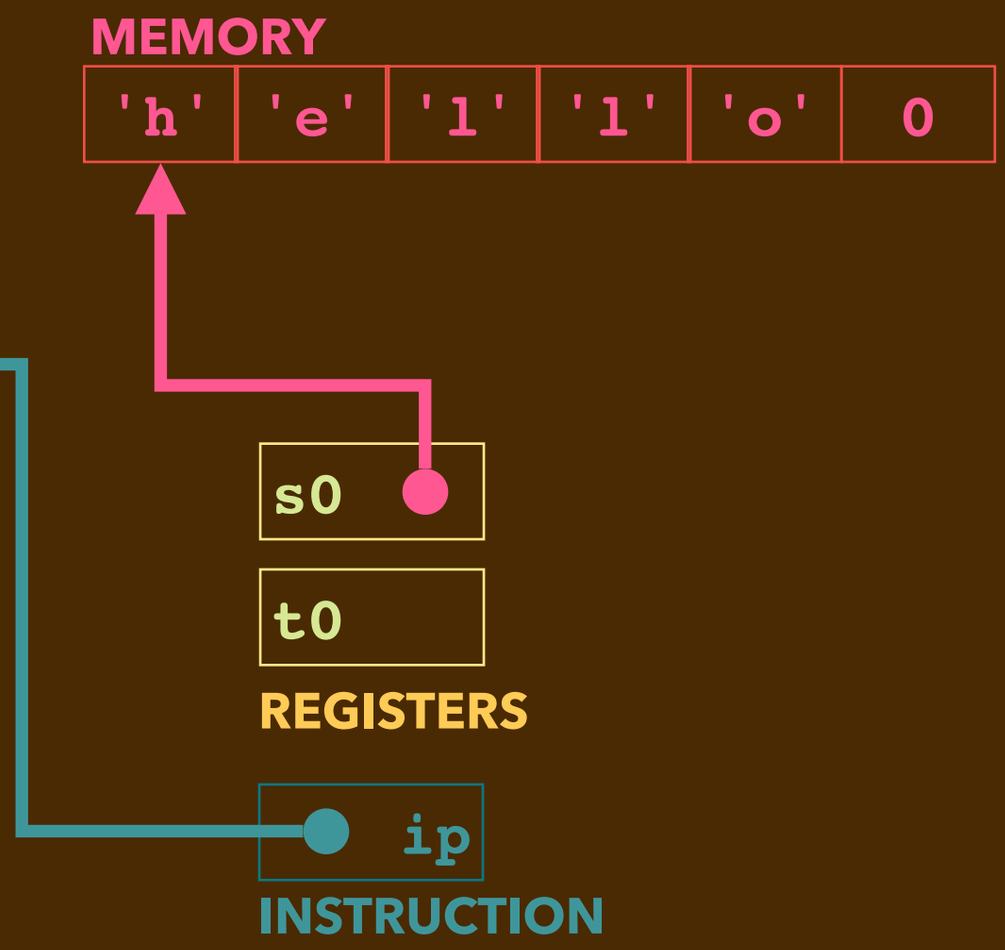
ALLCAPS CODE IN ACTION

```
1.          .data
2.  string_ptr: .ascii "hello\000"
3.          .text
4.  main:
5.      la      $s0,string_ptr
6.  loop:
7.      lb      $t0,($s0)
8.      beqz    $t0,done
9.
10.     addi    $t0,$t0,-32
11.     sb      $t0,($s0)
12.
13.     addi    $s0,$s0,1
14.     b       loop
15. done:
```



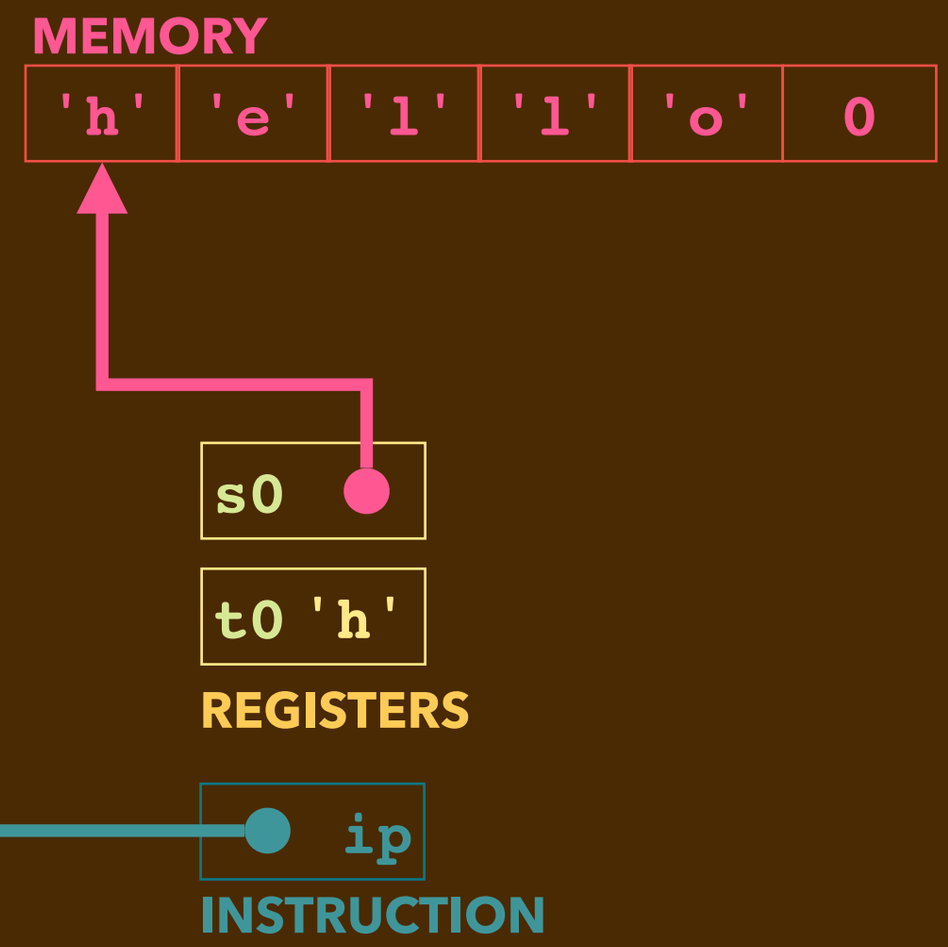
ALLCAPS CODE IN ACTION

```
1.          .data
2.  string_ptr: .ascii "hello\000"
3.          .text
4.  main:
5.      la      $s0,string_ptr
6.  loop:
7.      lb      $t0,($s0)
8.      beqz    $t0,done
9.
10.     addi    $t0,$t0,-32
11.     sb      $t0,($s0)
12.
13.     addi    $s0,$s0,1
14.     b       loop
15. done:
```



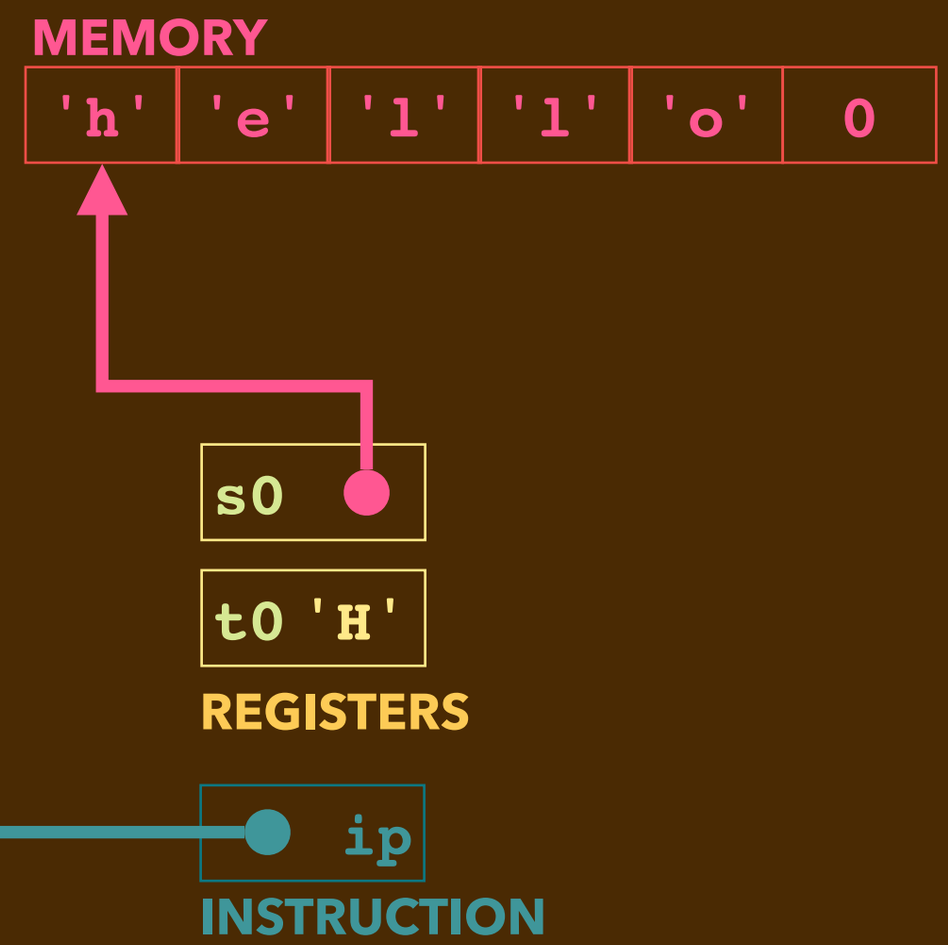
ALLCAPS CODE IN ACTION

```
1.          .data
2.  string_ptr: .ascii "hello\000"
3.          .text
4.  main:
5.      la      $s0,string_ptr
6.  loop:
7.      lb      $t0,($s0)
8.      beqz    $t0,done
9.
10.     addi    $t0,$t0,-32
11.     sb      $t0,($s0)
12.
13.     addi    $s0,$s0,1
14.     b       loop
15. done:
```



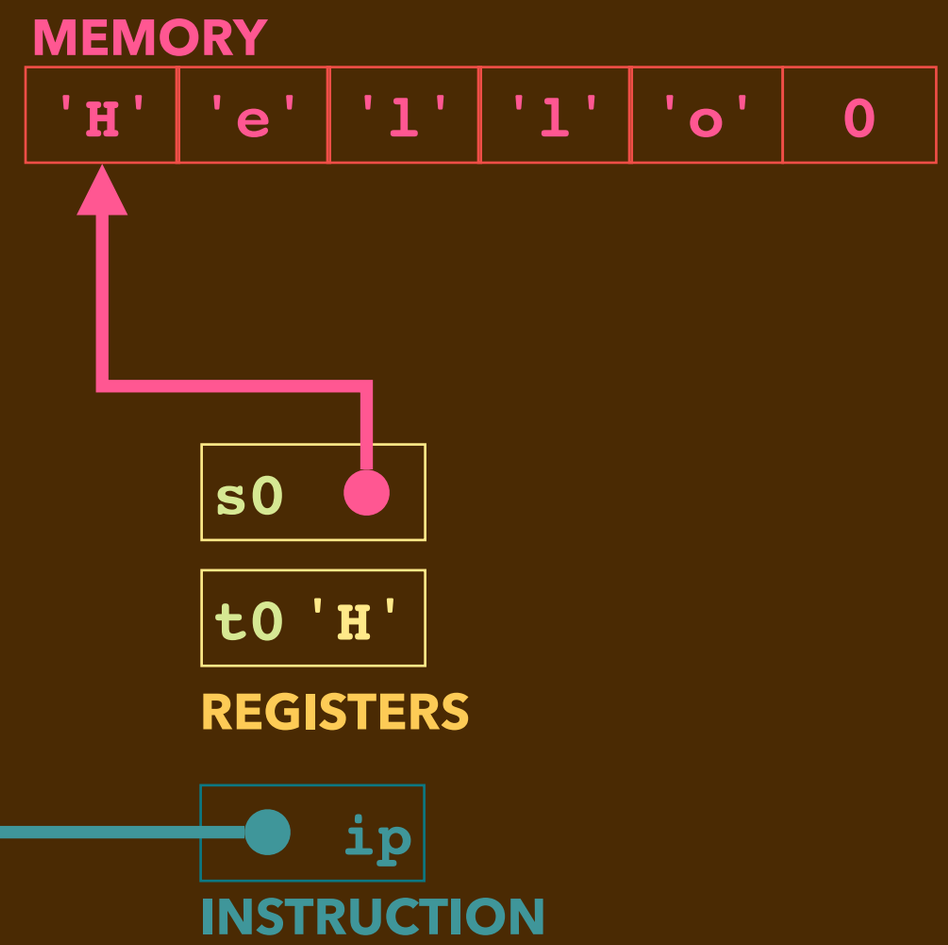
ALLCAPS CODE IN ACTION

```
1.          .data
2.  string_ptr: .ascii "hello\000"
3.          .text
4.  main:
5.      la      $s0,string_ptr
6.  loop:
7.      lb      $t0,($s0)
8.      beqz    $t0,done
9.
10.     addi    $t0,$t0,-32
11.     sb      $t0,($s0)
12.
13.     addi    $s0,$s0,1
14.     b       loop
15. done:
```



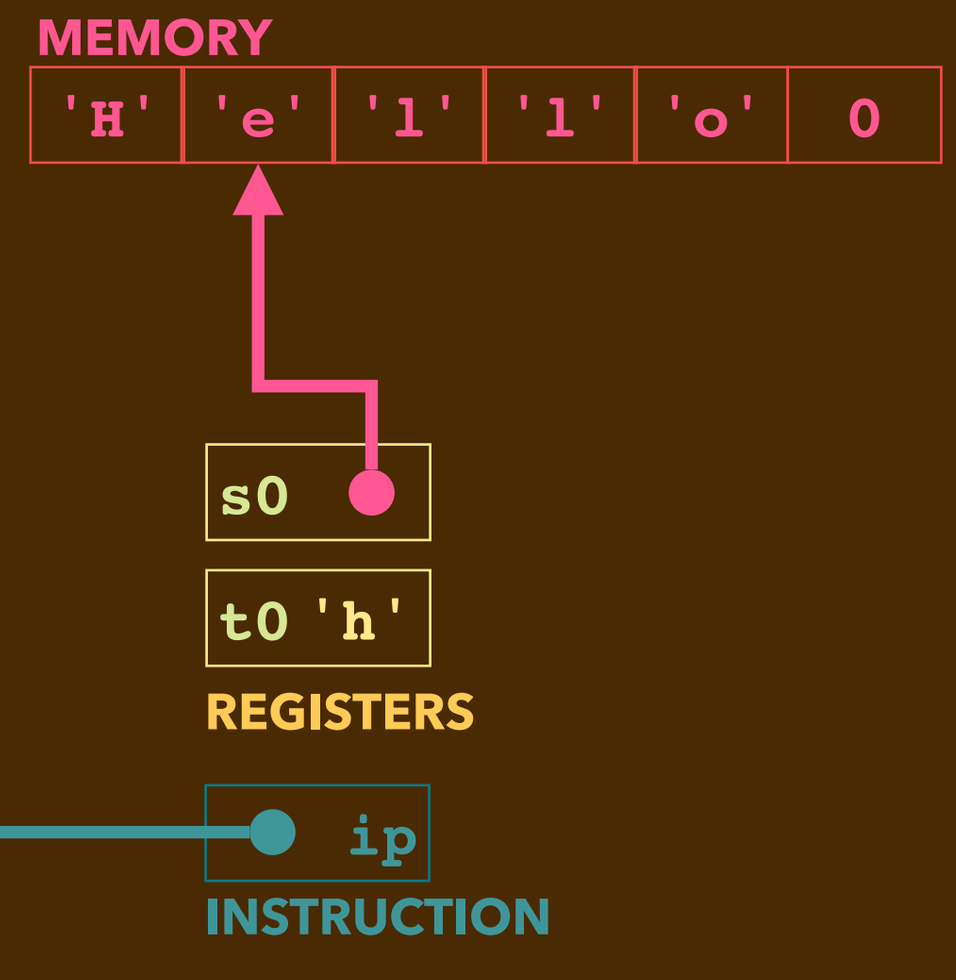
ALLCAPS CODE IN ACTION

```
1.          .data
2.  string_ptr: .ascii "hello\000"
3.          .text
4.  main:
5.      la      $s0,string_ptr
6.  loop:
7.      lb      $t0,($s0)
8.      beqz    $t0,done
9.
10.     addi    $t0,$t0,-32
11.     sb      $t0,($s0)
12.
13.     addi    $s0,$s0,1
14.     b       loop
15. done:
```



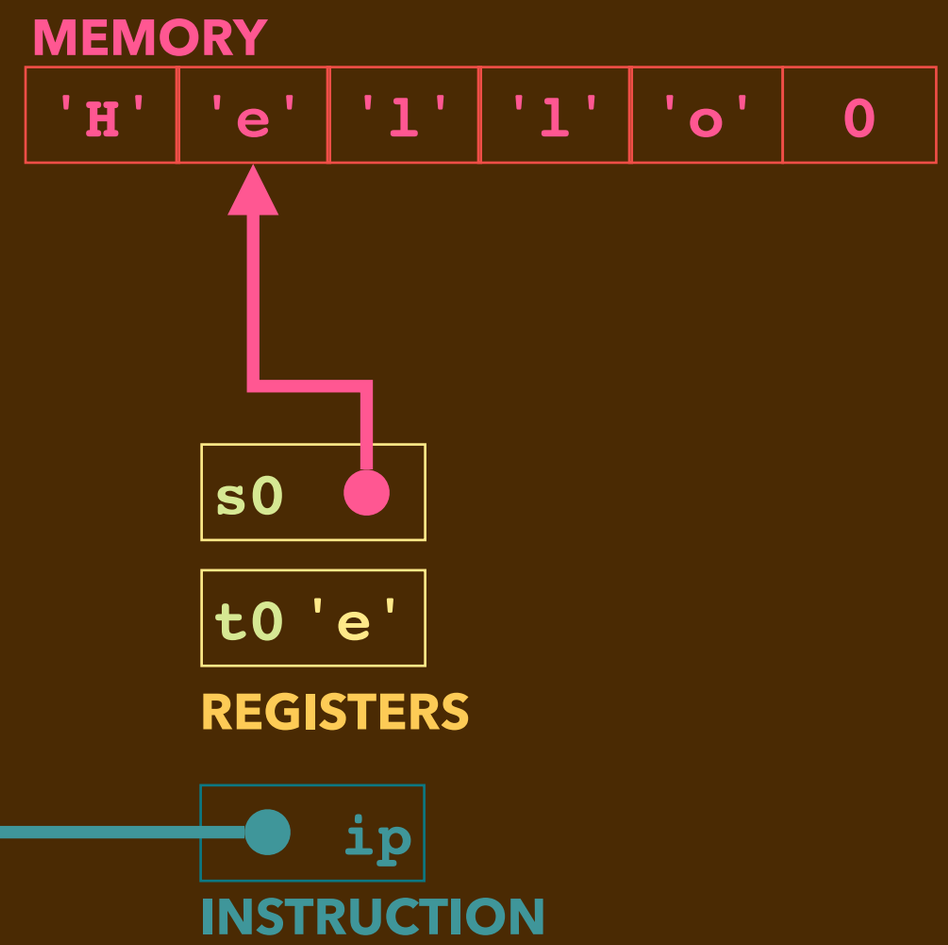
ALLCAPS CODE IN ACTION

```
1.          .data
2.  string_ptr: .ascii "hello\000"
3.          .text
4.  main:
5.      la      $s0,string_ptr
6.  loop:
7.      lb      $t0,($s0)
8.      beqz    $t0,done
9.
10.     addi    $t0,$t0,-32
11.     sb      $t0,($s0)
12.
13.     addi    $s0,$s0,1
14.     b       loop
15. done:
```



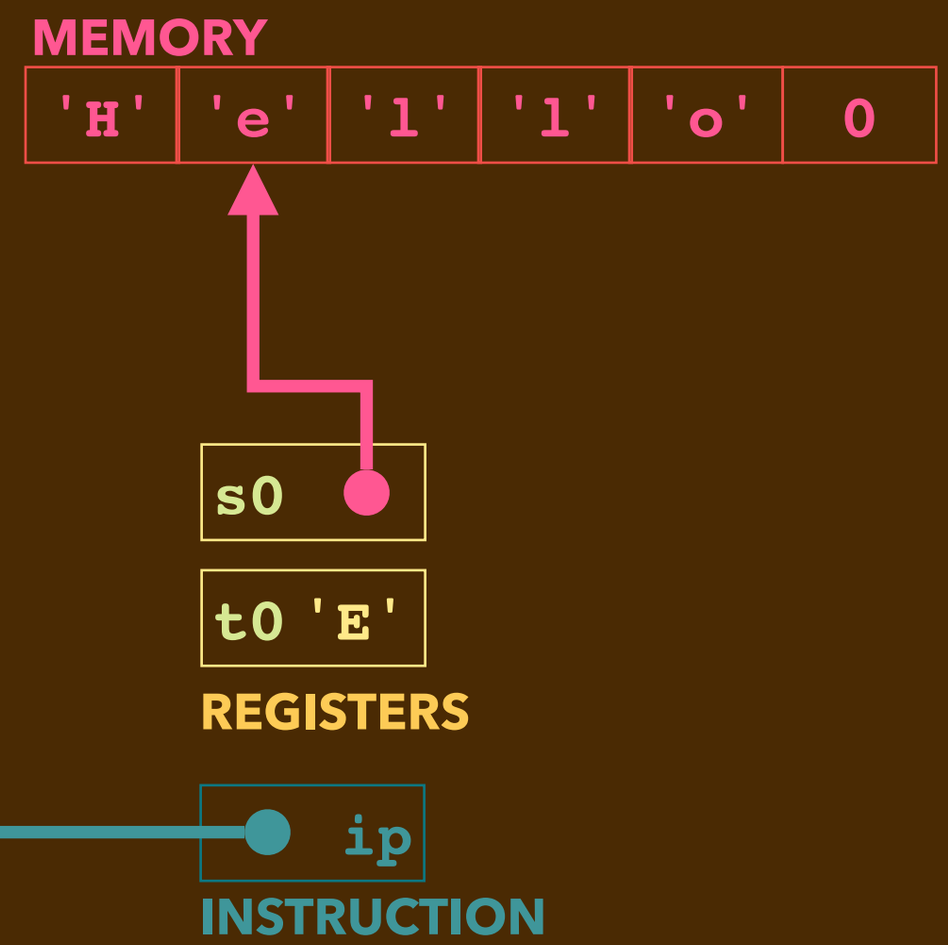
ALLCAPS CODE IN ACTION

```
1.          .data
2.  string_ptr: .ascii "hello\000"
3.          .text
4.  main:
5.      la      $s0,string_ptr
6.  loop:
7.      lb      $t0,($s0)
8.      beqz    $t0,done
9.
10.     addi    $t0,$t0,-32
11.     sb      $t0,($s0)
12.
13.     addi    $s0,$s0,1
14.     b       loop
15. done:
```



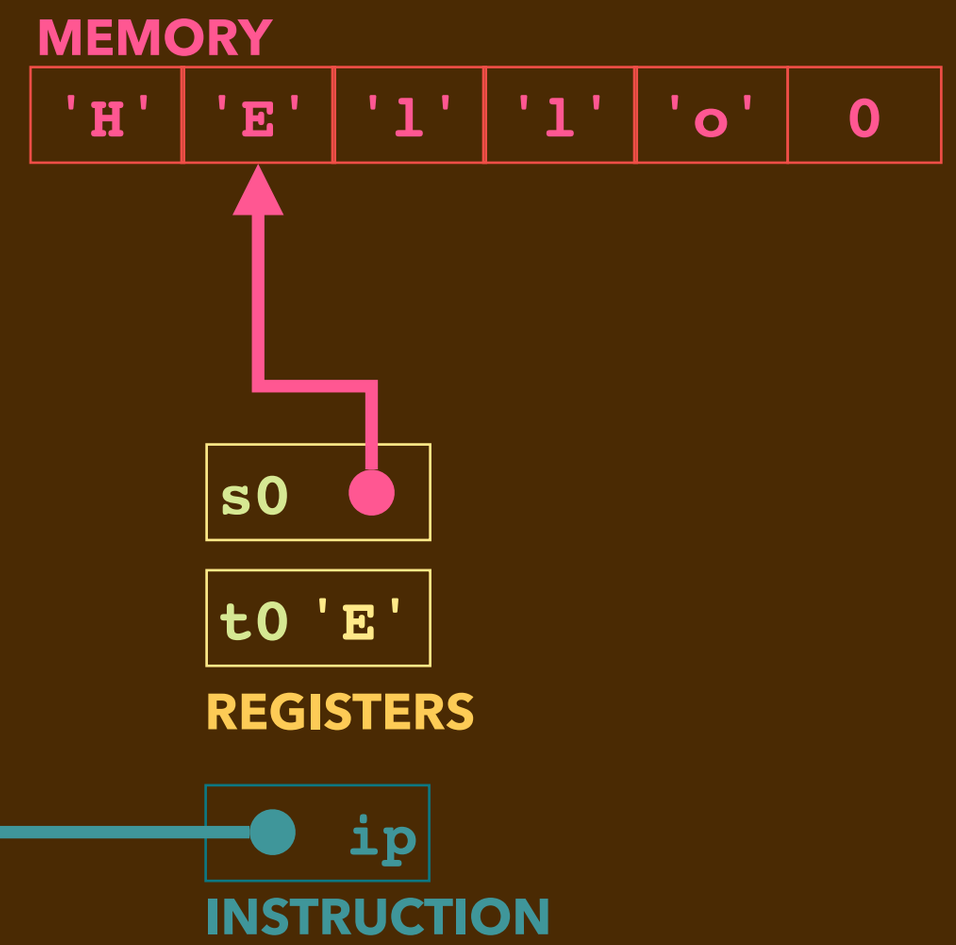
ALLCAPS CODE IN ACTION

```
1.          .data
2.  string_ptr: .ascii "hello\000"
3.          .text
4.  main:
5.      la      $s0,string_ptr
6.  loop:
7.      lb      $t0,($s0)
8.      beqz    $t0,done
9.
10.     addi    $t0,$t0,-32
11.     sb      $t0,($s0)
12.
13.     addi    $s0,$s0,1
14.     b       loop
15. done:
```



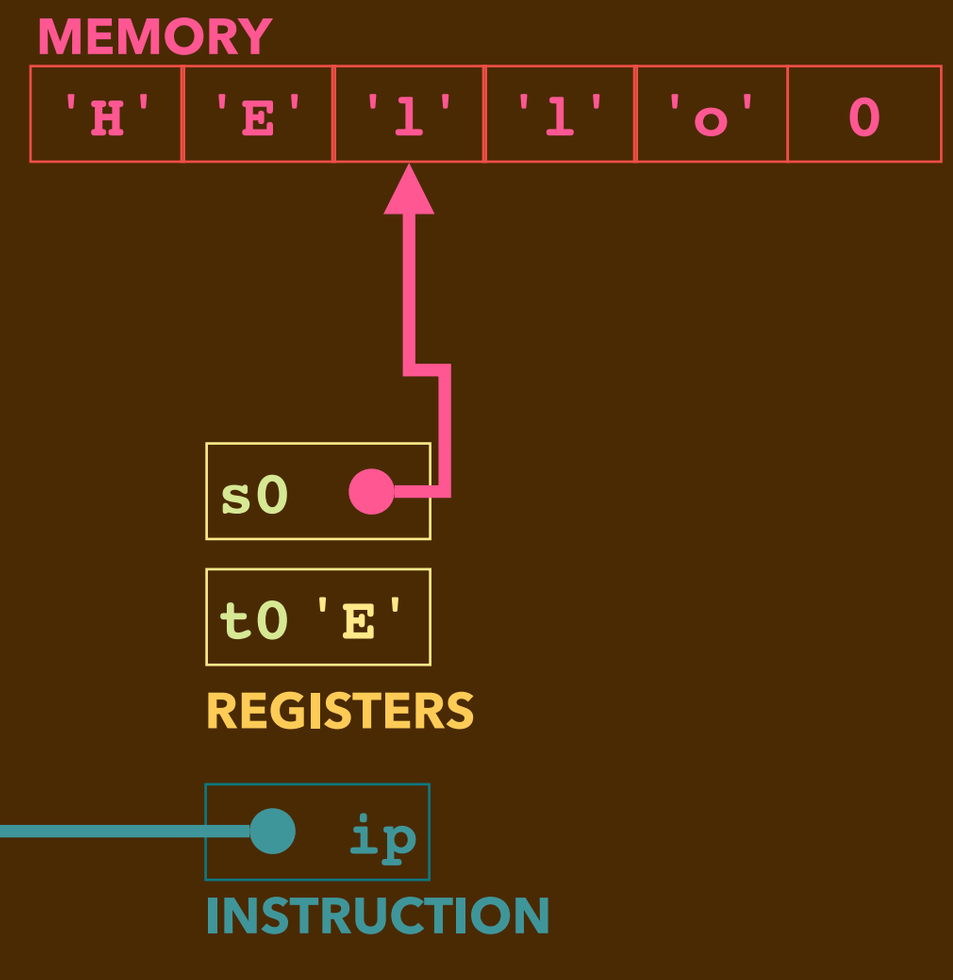
ALLCAPS CODE IN ACTION

```
1.          .data
2.  string_ptr: .ascii "hello\000"
3.          .text
4.  main:
5.      la      $s0,string_ptr
6.  loop:
7.      lb      $t0,($s0)
8.      beqz    $t0,done
9.
10.     addi    $t0,$t0,-32
11.     sb      $t0,($s0)
12.
13.     addi    $s0,$s0,1
14.     b       loop
15. done:
```



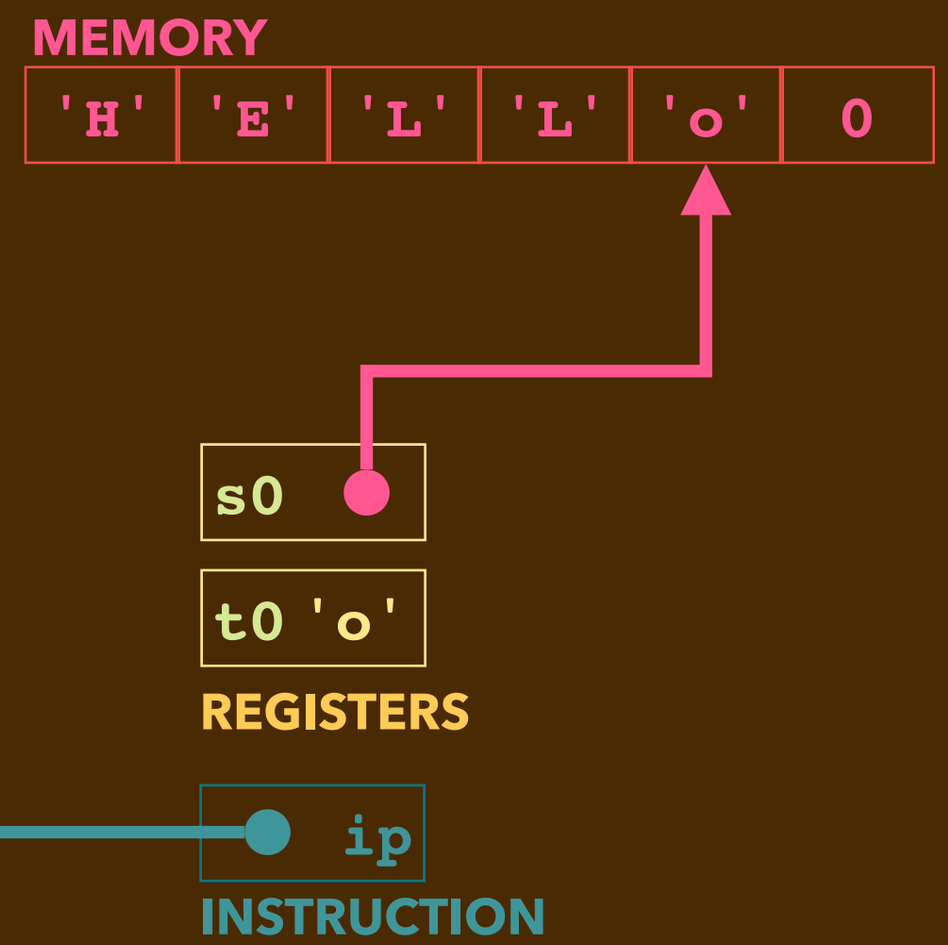
ALLCAPS CODE IN ACTION

```
1.          .data
2.  string_ptr: .ascii "hello\000"
3.          .text
4.  main:
5.      la      $s0,string_ptr
6.  loop:
7.      lb      $t0,($s0)
8.      beqz    $t0,done
9.
10.     addi    $t0,$t0,-32
11.     sb      $t0,($s0)
12.
13.     addi    $s0,$s0,1
14.     b       loop
15. done:
```



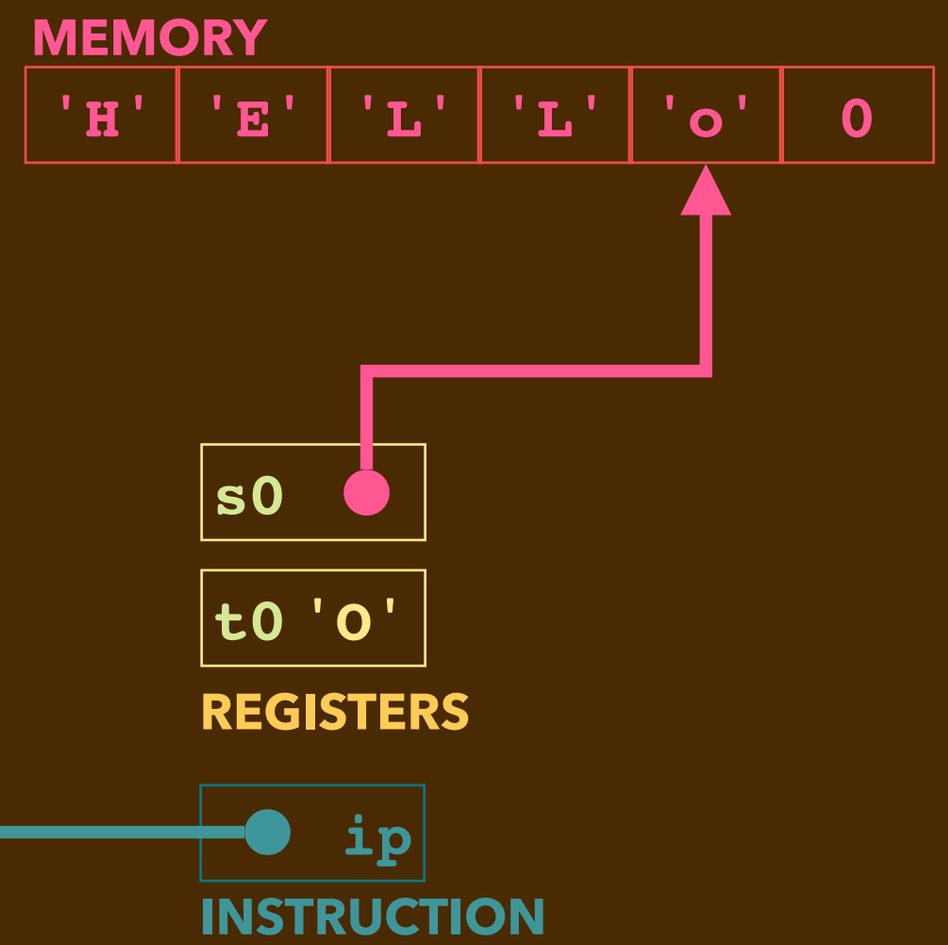
ALLCAPS CODE IN ACTION

```
1.          .data
2.  string_ptr: .ascii "hello\000"
3.          .text
4.  main:
5.      la      $s0,string_ptr
6.  loop:
7.      lb      $t0,($s0)
8.      beqz    $t0,done
9.
10.     addi    $t0,$t0,-32
11.     sb      $t0,($s0)
12.
13.     addi    $s0,$s0,1
14.     b       loop
15. done:
```



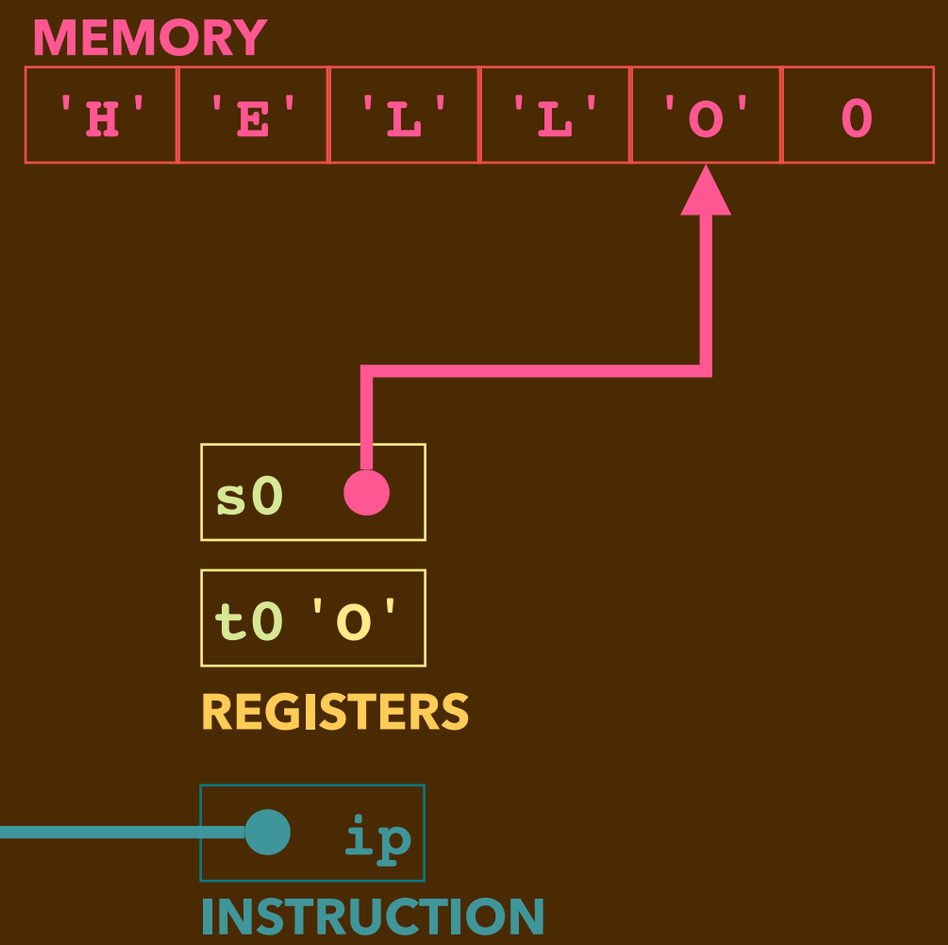
ALLCAPS CODE IN ACTION

```
1.          .data
2.  string_ptr: .ascii "hello\000"
3.          .text
4.  main:
5.      la      $s0,string_ptr
6.  loop:
7.      lb      $t0,($s0)
8.      beqz    $t0,done
9.
10.     addi    $t0,$t0,-32
11.     sb      $t0,($s0)
12.
13.     addi    $s0,$s0,1
14.     b       loop
15. done:
```



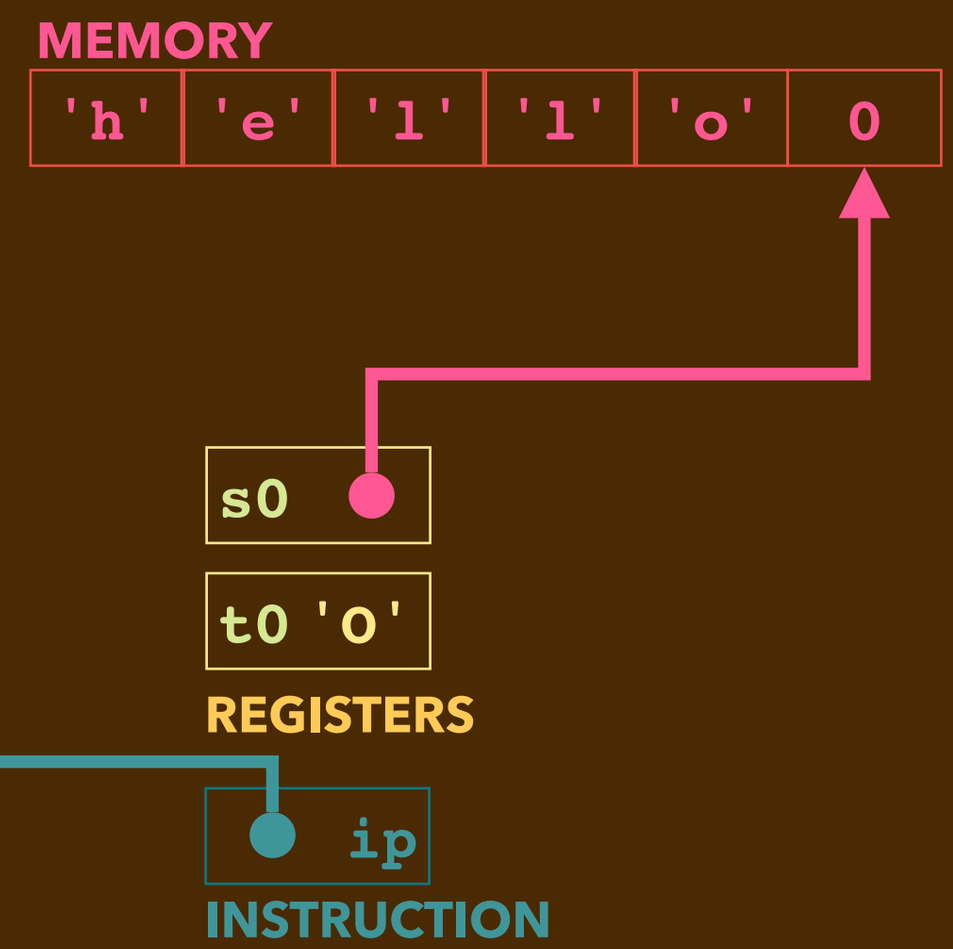
ALLCAPS CODE IN ACTION

```
1.          .data
2.  string_ptr: .ascii "hello\000"
3.          .text
4.  main:
5.      la      $s0,string_ptr
6.  loop:
7.      lb      $t0,($s0)
8.      beqz    $t0,done
9.
10.     addi    $t0,$t0,-32
11.     sb      $t0,($s0)
12.
13.     addi    $s0,$s0,1
14.     b       loop
15. done:
```



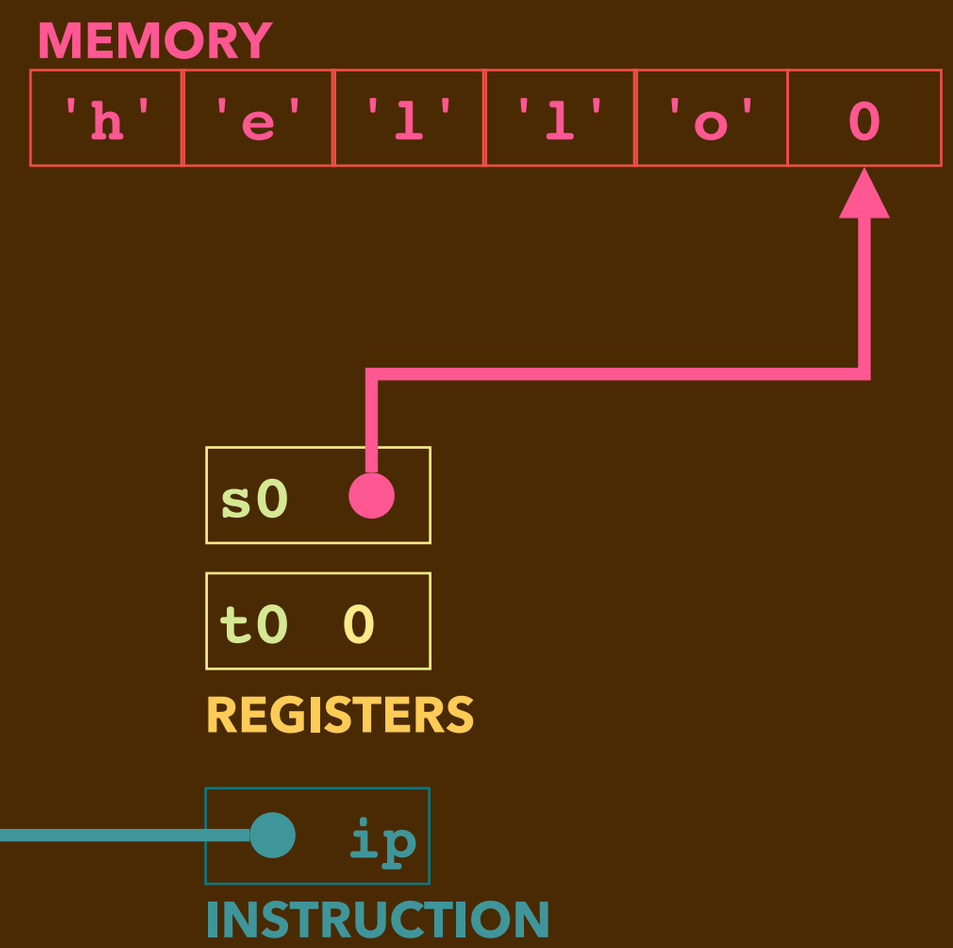
ALLCAPS CODE IN ACTION

```
1.          .data
2.  string_ptr: .ascii "hello\000"
3.          .text
4.  main:
5.      la      $s0,string_ptr
6.  loop:
7.      lb      $t0,($s0)
8.      beqz    $t0,done
9.
10.     addi    $t0,$t0,-32
11.     sb      $t0,($s0)
12.
13.     addi    $s0,$s0,1
14.     b       loop
15. done:
```



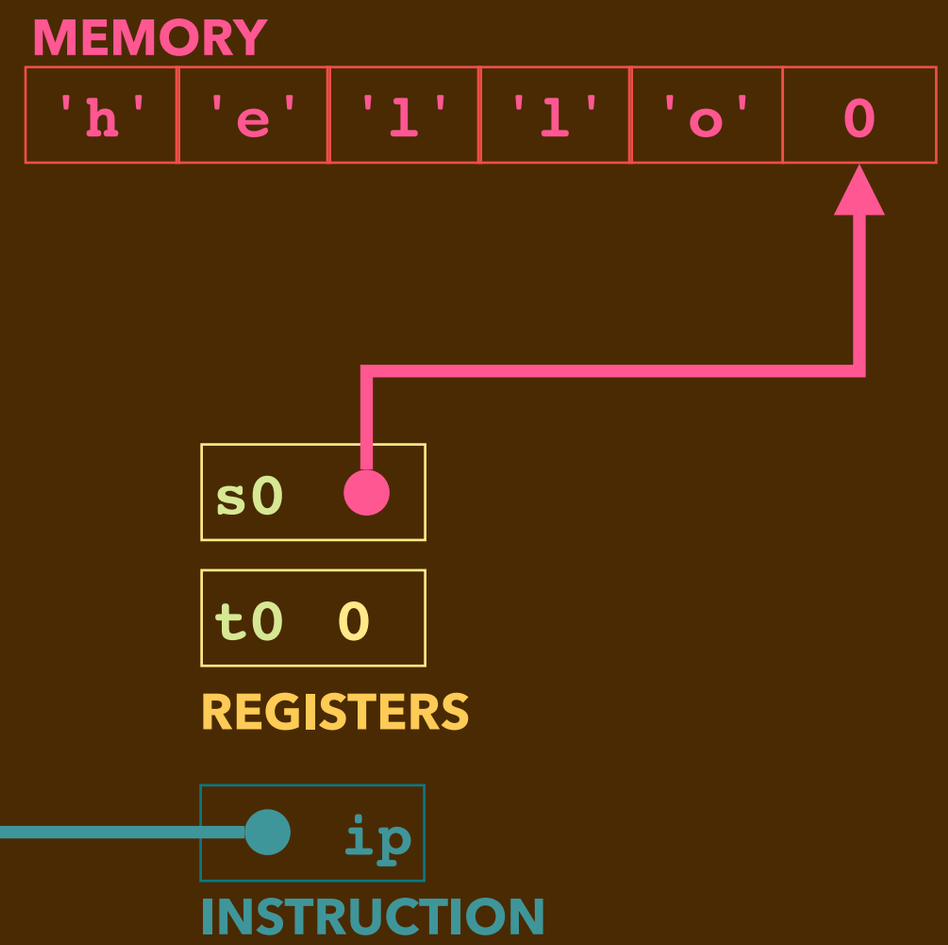
ALLCAPS CODE IN ACTION

```
1.          .data
2.  string_ptr: .ascii "hello\000"
3.          .text
4.  main:
5.      la      $s0,string_ptr
6.  loop:
7.      lb      $t0,($s0)
8.      beqz    $t0,done
9.
10.     addi    $t0,$t0,-32
11.     sb      $t0,($s0)
12.
13.     addi    $s0,$s0,1
14.     b       loop
15. done:
```



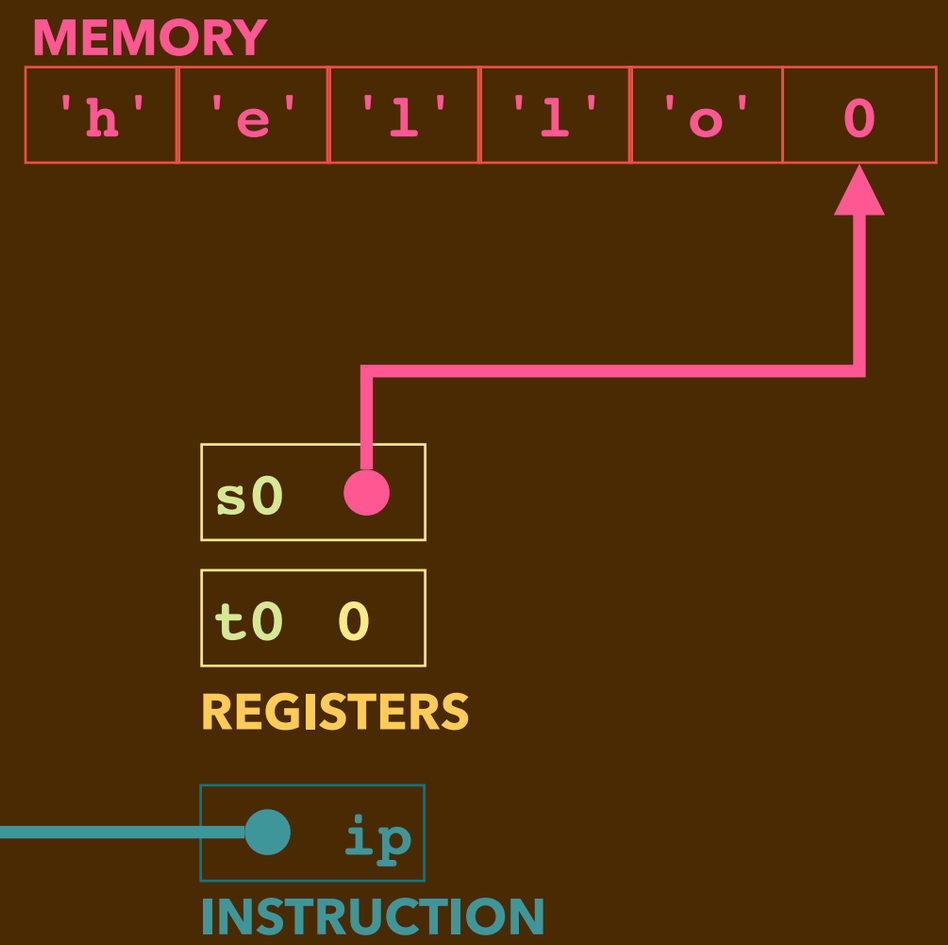
ALLCAPS CODE IN ACTION

```
1.          .data
2.  string_ptr: .ascii "hello\000"
3.          .text
4.  main:
5.      la      $s0,string_ptr
6.  loop:
7.      lb      $t0,($s0)
8.      beqz    $t0,done
9.
10.     addi    $t0,$t0,-32
11.     sb      $t0,($s0)
12.
13.     addi    $s0,$s0,1
14.     b       loop
15. done:
```



ALLCAPS CODE IN ACTION

```
1.          .data
2.  string_ptr: .ascii "hello\000"
3.          .text
4.  main:
5.      la      $s0,string_ptr
6.  loop:
7.      lb      $t0,($s0)
8.      beqz    $t0,done
9.
10.     addi    $t0,$t0,-32
11.     sb      $t0,($s0)
12.
13.     addi    $s0,$s0,1
14.     b       loop
15. done:
```



MIPS MEMORY .DATA SEGMENT

- You can reserve space for data other than strings in the memory image:

```
1.      .data
2.  string_ptr: .asciiz "Here is a null-terminated string."
3.  int_ptr:    .word 101, 42, 18
4.  area_ptr:   .space 20
```

  - **`.asciiz`** sets aside space for a null-terminated character sequence.
  - **`.word`** sets aside space for 4-byte (integer) values.
  - **`.space`** sets aside a contiguous region of uninitialized bytes.
- Labels give addresses we can load into a register (**`LA`**); treat as a pointer.
- We can read and write memory using an address in a register (**`LW`**; **`SW`**)

EXAMPLE

- Here is code that reads 101 from memory and writes 101+50 to it.

```
1.      .data
2.  string_ptr: .asciiz "Here is a null-terminated string."
3.  int_ptr:    .word 101, 42, 18
4.  area_ptr:   .space 20
5.      .text
6.  main:
7.      la $s0, int_ptr
8.      la $s1, area_ptr
9.      lw $t0, ($s0)
10.     addi $t0, $t0, 50
11.     sw $t0, ($s1)
```

EXAMPLE

- Here is code that reads 101 from memory and writes 101+50 to it.

```
1.      .data
2.  string_ptr: .asciiz "Here is a null-terminated string."
3.  int_ptr:    .word 101, 42, 18
4.  area_ptr:   .space 20
5.      .text
6.  main:
7.      la $s0, int_ptr
8.      la $s1, area_ptr
9.      lw $t0, ($s0)
10.     addi $t0, $t0, 50
11.     sw $t0, ($s1)
```

- The first instruction (line 07) sets s0 to point to the first word at int_ptr.

EXAMPLE

- Here is code that reads 101 from memory and writes 101+50 to it.

```
1.      .data
2.  string_ptr: .asciiz "Here is a null-terminated string."
3.  int_ptr:    .word 101, 42, 18
4.  area_ptr:   .space 20
5.      .text
6.  main:
7.      la $s0, int_ptr
8.      la $s1, area_ptr
9.      lw $t0, ($s0)
10.     addi $t0, $t0, 50
11.     sw $t0, ($s1)
```

- The second instruction (line 08) sets s1 to point to the first word at area_ptr.

EXAMPLE

- Here is code that reads 101 from memory and writes 101+50 to it.

```
1.      .data
2.  string_ptr: .asciiz "Here is a null-terminated string."
3.  int_ptr:    .word 101, 42, 18
4.  area_ptr:   .space 20
5.      .text
6.  main:
7.      la $s0, int_ptr
8.      la $s1, area_ptr
9.      lw $t0, ($s0)
10.     addi $t0, $t0, 50
11.     sw $t0, ($s1)
```

- Then it loads the value of 101 into register t0 with a **LW** in line 09.

EXAMPLE

- Here is code that reads 101 from memory and writes 101+50 to it.

```
1.      .data
2.  string_ptr: .asciiz "Here is a null-terminated string."
3.  int_ptr:    .word 101, 42, 18
4.  area_ptr:   .space 20
5.      .text
6.  main:
7.      la $s0, int_ptr
8.      la $s1, area_ptr
9.      lw $t0, ($s0)
10.     addi $t0, $t0, 50
11.     sw $t0, ($s1)
```

- It adds 50 to that, making t0 contain 151 in line 10.

EXAMPLE

- Here is code that reads 101 from memory and writes 101+50 to it.

```
1.      .data
2.  string_ptr: .asciiz "Here is a null-terminated string."
3.  int_ptr:    .word 101, 42, 18
4.  area_ptr:   .space 20
5.      .text
6.  main:
7.      la $s0, int_ptr
8.      la $s1, area_ptr
9.      lw $t0, ($s0)
10.     addi $t0, $t0, 50
11.     sw $t0, ($s1)
```

- Finally it writes the 4-byte value of 151 into memory referenced by area_ptr.

ARRAY CODE

- We can treat areas in the .data segment as integer arrays.
- The code below reads in a sequence of integers, placing them at area_ptr.

```
1.      la $s0, area_ptr
2.      li $t1, 5           # Count out 5 inputs.
3.  input_loop:
4.      beqz $t1, input_done
5.
6.      li $v0, 5           #
7.      syscall             # Get an integer input.
8.
9.      sw $v0,($s0)        # Store it in the array.
10.     addi $s0,$s0,4      # Advance the pointer by 4 bytes.
11.     addi $t1,$t1,-1     # Decrement the count.
12.     b input_loop
```

ARRAY CODE

- At the top we have: `area_ptr:    .space 20`
- The code below reads in a sequence of integers, placing them at area_ptr.

```
1.      la $s0, area_ptr
2.      li $t1, 5           # Count out 5 inputs.
3.  input_loop:
4.      beqz $t1, input_done
5.
6.      li $v0, 5           #
7.      syscall             # Get an integer input.
8.
9.      sw $v0,($s0)        # Store it in the array.
10.     addi $s0,$s0,4      # Advance the pointer by 4 bytes.
11.     addi $t1,$t1,-1     # Decrement the count.
12.     b input_loop
```

- The first line loads a pointer value into s0. The start of the array.

ARRAY CODE

- At the top we have: `area_ptr:    .space 20`
- The code below reads in a sequence of integers, placing them at area_ptr.

```
1.      la $s0, area_ptr
2.      li $t1, 5           # Count out 5 inputs.
3.  input_loop:
4.      beqz $t1, input_done
5.
6.      li $v0, 5           #
7.      syscall             # Get an integer input.
8.
9.      sw $v0,($s0)        # Store it in the array.
10.     addi $s0,$s0,4      # Advance the pointer by 4 bytes.
11.     addi $t1,$t1,-1     # Decrement the count.
12.     b input_loop
```

- Lines 06 and 07 get an integer from the console, put it in v0.

ARRAY CODE

- At the top we have: `area_ptr:    .space 20`
- The code below reads in a sequence of integers, placing them at area_ptr.

```
1.      la $s0, area_ptr
2.      li $t1, 5           # Count out 5 inputs.
3.  input_loop:
4.      beqz $t1, input_done
5.
6.      li $v0, 5           #
7.      syscall             # Get an integer input.
8.
9.      sw $v0,($s0)        # Store it in the array.
10.     addi $s0,$s0,4      # Advance the pointer by 4 bytes.
11.     addi $t1,$t1,-1     # Decrement the count.
12.     b input_loop
```

- These lines are key: We store the *int* in memory then advance that pointer.

ARRAY CODE

- At the top we have: `area_ptr:    .space 20`
- The code below reads in a sequence of integers, placing them at area_ptr.

```
1.      la $s0, area_ptr
2.      li $t1, 5           # Count out 5 inputs.
3.  input_loop:
4.      beqz $t1, input_done
5.
6.      li $v0, 5           #
7.      syscall             # Get an integer input.
8.
9.      sw $v0,($s0)        # Store it in the array.
10.     addi $s0,$s0,4      # Advance the pointer by 4 bytes.
11.     addi $t1,$t1,-1     # Decrement the count.
12.     b input_loop
```

- Since integers are four bytes wide, we advance the pointer by four.

SUMMING AN ARRAY

- The code sums an array of integers in memory.
  - t1 initially holds the array's length. The loop counts down.
  - t0 holds the sum.
  - s0 points to the start of the array, and is advanced (by four).

```
1.      li    $t0, 0
2.  sum_loop:
3.      beqz  $t1, sum_done
4.      lw    $t2,($s0)
5.      add   $t0,$t0,$t2
6.      addi  $s0,$s0,4
7.      addi  $t1,$t1,-1
8.      b     sum_loop
9.  sum_done:
```

SUMMING AN ARRAY

- The code sums an array of integers in memory.
  - t1 initially holds the array's length. The loop counts down.
  - t0 holds the sum.
  - s0 points to the start of the array, and is advanced (by four).

```
1.      li    $t0, 0
2.  sum_loop:
3.      beqz  $t1, sum_done
4.      lw    $t2,($s0)
5.      addu  $t0,$t0,$t2
6.      addi  $s0,$s0,4
7.      addi  $t1,$t1,-1
8.      b     sum_loop
9.  sum_done:
```

- Lines 04-06 are key: fetch the next array value, add it to the sum; advance.

SWAPPING CONSECUTIVE ITEMS IN AN ARRAY

- Many sort algorithms involve a "neighbor swap" operation. In C++

```
1.  int tmp1 = a[i];
2.  int tmp2 = a[i+1];
3.  a[i] = tmp2;
4.  a[i+1] = tmp1;
```

- Here is the MIPS code equivalent (assuming s1 is &a[i]):

```
1.      addi   $s2,$s1,4
2.      lw     $t1,($s1)
3.      lw     $t2,($s2)
4.      sw     $t2,($s1)
5.      sw     $t1,($s2)
```

SWAP, USING OFFSETS

- Many sort algorithms involve a "neighbor swap" operation. In C++

```
1.  int tmp1 = a[i];
2.  int tmp2 = a[i+1];
3.  a[i] = tmp2;
4.  a[i+1] = tmp1;
```

- Here is the MIPS code equivalent (assuming s1 is &a[i]):

```
1.      lw    $t1,0($s1)
2.      lw    $t2,4($s1)
3.      sw    $t2,0($s1)
4.      sw    $t1,4($s1)
```

- The code above is using "offset addressing".
  - ➡ The notation **`k($r)`** means "memory at address r+k"

LOADING AND STORING AT AN OFFSET FROM AN ADDRESS

LOAD A (FOUR BYTE) VALUE FROM AN ADDRESS IN MEMORY AT AN OFFSET

`LW` *destination* `,` *offset* `(` *source* `)`

- Load four bytes starting at ***offset*** bytes from the address stored in ***source***

STORE A (FOUR BYTE) VALUE TO AN ADDRESS IN MEMORY AT AN OFFSET

`SW` *source* `,` *offset* `(` *destination* `)`

- Store four bytes starting at ***offset*** bytes from the address stored in ***destination***

NOTE: offset must be a constant value!!!

➡ Some of you will be tempted to write `$t1($s1)` to mean `a[i]`.

COMPILER USE OF OFFSETS

- Consider this C++ code:

```
1.  void fcn(int a, int b) {
2.       ...
3.       int x = a - b;
4.       int y = b + 10;
5.       ...
6.  }
```

- Here is MIPS code that mimics what a C++ compiler might generate

```
1.       fcn:
2.            ...
3.            lw    $t0, 0($fp)
4.            lw    $t1, -4($fp)
5.            sub   $t2,$t0,$t1
6.            sw    $t2, -8($fp)
7.            addi  $t3,$t1,10
8.            sw    $t3, -12($fp)
9.            ...
```

COMPILER USE OF OFFSETS

- Consider this C++ code:

```
1.  void fcn(int a, int b) {
2.       ...
3.       int x = a - b;
4.       int y = b + 10;
5.       ...
6.  }
```

- Here is MIPS code that mimics what a C++ compiler might generate

```
1.       fcn:
2.            ...
3.            lw    $t0, 0($fp)
4.            lw    $t1, -4($fp)
5.            sub   $t2,$t0,$t1
6.            sw    $t2, -8($fp)
7.            addi  $t3,$t1,10
8.            sw    $t3, -12($fp)
9.            ...
```

fp is the register used as a "stack frame pointer"

COMPILER USE OF OFFSETS

- Consider this C++ code:

```
1.  void fcn(int a, int b) {
2.       ...
3.       int x = a - b;
4.       int y = b + 10;
5.       ...
6.  }
```

- Here is MIPS code that mimics what a C++ compiler might generate

```
1.       fcn:
2.            ...
3.            lw     $t0, 0($fp)
4.            lw     $t1, -4($fp)
5.            sub    $t2,$t0,$t1
6.            sw     $t2, -8($fp)
7.            addi   $t3,$t1,10
8.            sw     $t3, -12($fp)
9.            ...
```

a is being held at of offset of 0 bytes in the frame

COMPILER USE OF OFFSETS

- Consider this C++ code:

```
1.  void fcn(int a, int b) {
2.       ...
3.       int x = a - b;
4.       int y = b + 10;
5.       ...
6.  }
```

- Here is MIPS code that mimics what a C++ compiler might generate

```
1.       fcn:
2.            ...
3.            lw     $t0, 0($fp)
4.            lw     $t1, -4($fp)
5.            sub    $t2,$t0,$t1
6.            sw     $t2, -8($fp)
7.            addi   $t3,$t1,10
8.            sw     $t3, -12($fp)
9.            ...
```

b is being held at of offset of -4 bytes in the frame

COMPILER USE OF OFFSETS

- Consider this C++ code:

```
1.  void fcn(int a, int b) {
2.       ...
3.       int x = a - b;
4.       int y = b + 10;
5.       ...
6.  }
```

- Here is MIPS code that mimics what a C++ compiler might generate

```
1.       fcn:
2.            ...
3.            lw     $t0, 0($fp)
4.            lw     $t1, -4($fp)
5.            sub    $t2,$t0,$t1
6.            sw     $t2, -8($fp)
7.            addi   $t3,$t1,10
8.            sw     $t3, -12($fp)
9.            ...
```

x is being held at of offset of -8 bytes in the frame

OFFSETS FOR ACCESSING STRUCT COMPONENTS

- Consider this C++ struct definition for a 3-D coordinate:

```
1.      struct coord {
2.          int x;
3.          int y;
4.          int z;
5.      };
```

- Here might be the use of this coord struct in other code:

```
6.      coord* p1;
7.      coord* p2;
8.      ...
9.      p2->x = 17;
10.     p2->y = p1->y;
11.     p2->z++;
```

- The compiler will lay out x,y,z contiguously in memory, as 24 bytes.

OFFSETS FOR ACCESSING STRUCT COMPONENTS

- Each access to a struct's component will be at an offset from its pointer.

```
1.      coord* p1;
2.      coord* p2;
3.      ...
4.      p2->x = 17;
5.      p2->y = p1->y;
6.      p2->z++;
```

- Here might be the MIPS code that a compiler would generate:

```
1.      li    $t1,17
2.      sw    $t1,0($s2)
3.
4.      lw    $t2,4($s1)
5.      sw    $t2,4($s2)
6.
7.      lw    $t3,8($s2)
8.      addi  $t3,$t3,1
9.      sw    $t3,8($s2)
```

OFFSETS FOR ACCESSING STRUCT COMPONENTS

- Each access to a struct's component will be at an offset from its pointer.

```
1.      coord* p1;
2.      coord* p2;
3.      ...
4.      p2->x = 17;
5.      p2->y = p1->y;
6.      p2->z++;
```

- Here might be the MIPS code that a compiler would generate:

```
1.      li    $t1,17
2.      sw    $t1,0($s2)
3.
4.      lw    $t2,4($s1)
5.      sw    $t2,4($s2)
6.
7.      lw    $t3,8($s2)
8.      addi  $t3,$t3,1
9.      sw    $t3,8($s2)
```

x is being held at offset 0

OFFSETS FOR ACCESSING STRUCT COMPONENTS

- Each access to a struct's component will be at an offset from its pointer.

```
1.      coord* p1;
2.      coord* p2;
3.      ...
4.      p2->x = 17;
5.      p2->y = p1->y;
6.      p2->z++;
```

- Here might be the MIPS code that a compiler would generate:

```
1.      li    $t1,17
2.      sw    $t1,0($s2)
3.
4.      lw    $t2,4($s1)
5.      sw    $t2,4($s2)
6.
7.      lw    $t3,8($s2)
8.      addi  $t3,$t3,1
9.      sw    $t3,8($s2)
```

x is being held at offset 0

y is being held at offset 4

OFFSETS FOR ACCESSING STRUCT COMPONENTS

- Each access to a struct's component will be at an offset from its pointer.

```
1.      coord* p1;
2.      coord* p2;
3.      ...
4.      p2->x = 17;
5.      p2->y = p1->y;
6.      p2->z++;
```

- Here might be the MIPS code that a compiler would generate:

```
1.      li    $t1,17
2.      sw    $t1,0($s2)
3.
4.      lw    $t2,4($s1)
5.      sw    $t2,4($s2)
6.
7.      lw    $t3,8($s2)
8.      addi  $t3,$t3,1
9.      sw    $t3,8($s2)
```

x is being held at offset 0

y is being held at offset 4

z is being held at offset 8

LINKED LIST CODE

- Consider this C++ struct definition for a linked list node:

```
1.      struct node {
2.          int data;
3.          struct node* next;
4.      };
```

- The code below builds a linked list storing the sequence 32, 57, 11

```
5.      node nodes[3];
6.      node* n1 = &nodes[0];
7.      node* n2 = &nodes[1];
8.      node* n3 = &nodes[2];
9.
10.     n1->data = 32;
11.     n2->data = 57;
12.     n3->data = 11;
13.
14.     n1->next = n2;
15.     n2->next = n3;
16.     n3->next = nullptr
```

LINKED LIST CODE CONVERTED TO MIPS

```
1.  node nodes[3];
2.
3.
4.
5.  node* n1 = &nodes[0];
6.  node* n2 = &nodes[1];
7.  node* n3 = &nodes[2];
8.
9.  n1->data = 32;
10.
11. n2->data = 57;
12.
13. n3->data = 11;
14.
15.
16. n1->next = n2;
17. n2->next = n3;
18.
19. n3->next = nullptr
```

```
1.      .data
2.  nodes: space 24
3.      .text
4.         ...
5.      la    $s1,nodes
6.      addi  $s2,$s1,8
7.      addi  $s3,$s1,16
8.
9.      li    $t0,32
10.     sw    $t0,($s1)
11.     li    $t0,57
12.     sw    $t0,($s2)
13.     li    $t0,11
14.     sw    $t0,($s3)

16.     sw    $s2,4($s1)
17.     sw    $s3,4($s2)
18.     li    $t0,0
19.     sw    $t0,4($s3)
```

LINKED LIST CODE CONVERTED TO MIPS

```
1.  n1->data = 32;
2.
3.  n2->data = 57;
4.
5.  n3->data = 11;
6.
7.
8.  n1->next = n2;
9.  n2->next = n3;
10.
11. n3->next = nullptr
```

```
1.      li    $t0,32
2.      sw    $t0,0($s1)
3.      li    $t0,57
4.      sw    $t0,0($s2)
5.      li    $t0,11
6.      sw    $t0,0($s3)
8.      sw    $s2,4($s1)
9.      sw    $s3,4($s2)
10.     li    $t0,0
11.     sw    $t0,4($s3)
```

- The **`data`** field is at offset 0 from the node pointer.
- The **`next`** field is at offset 4 from the node pointer.

TRAVERSING A LINKED LIST

- MIPS code that outputs a linked list

```
1.  print:
2.      move    $s1, $s0            # current = first;
3.  print_loop:
4.      beqz    $s1, done           # if current==nullptr goto done;
5.  print_data:
6.      lw      $a0, ($s1)          # print(current->data);
7.      li      $v0, 1
8.      syscall
9.      lw      $s1, 4($s1)         # current = current->next;
10.     b       print_loop
11. done:
```

- Check out my sample "inorder.s" that builds a linked list in sorted order.

FUNCTION CALLS IN MIPS

The MIPS ystem calls hint at a more general mechanism we need, namely...

Q: How do we mimic C++'s function calling mechanism in MIPS?

A: By following the MIPS function calling conventions and stack discipline.

OUTLINE:

- SOME SIMPLE C++ EXAMPLES
- CALL/RETURN WITH JAL/JR ; PARAMETER PASSING
- CREATE/PUSH AND TAKE-DOWN/POP OF STACK FRAME
- EXAMINE CONVENTIONS FOR SAVING REGISTERS' VALUES ON THE FRAME

RECALL: FUNCTIONS IN C++

- Consider this C++ program:

```
1.  int two_digits(int tens, int ones) {
2.      return 10 * tens + ones;
3.  }
4.  int times100(int number) {
5.      return 100 * number;
6.  }
7.  int main(void) { int A, B, C, D;
8.      cin >> A;
9.      cin >> B;
10.     cin >> C;
11.     cin >> D;
12.     int hi = two_digits(A,B);
13.     int lo = two_digits(C,D);
14.     int n = times100(hi) + lo;
15.     cout << n << endl;
16. }
```

RECALL: FUNCTIONS IN C++

- Consider this C++ program:

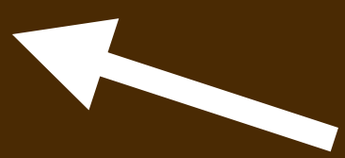
```
1.  int two_digits(int tens, int ones) {
2.      return 10 * tens + ones;
3.  }
4.  int times100(int number) {
5.      return 100 * number;
6.  }
7.  int main(void) { int A, B, C, D;
8.      cin >> A;
9.      cin >> B;
10.     cin >> C;
11.     cin >> D;
12.     int hi = two_digits(A,B);
13.     int lo = two_digits(C,D);
14.     int n = times100(hi) + lo;
15.     cout << n << endl;
16. }
```

call sites

RECALL: FUNCTIONS IN C++

- Consider this C++ program:

```
1.  int two_digits(int tens, int ones) {
2.      return 10 * tens + ones;
3.  }
4.  int times100(int number) {
5.      return 100 * number;
6.  }
7.  int main(void) { int A, B, C, D;
8.      cin >> A;
9.      cin >> B;
10.     cin >> C;
11.     cin >> D;
12.     int hi = two_digits(A,B);
13.     int lo = two_digits(C,D);
14.     int n = times100(hi) + lo;
15.     cout << n << endl;
16. }
```

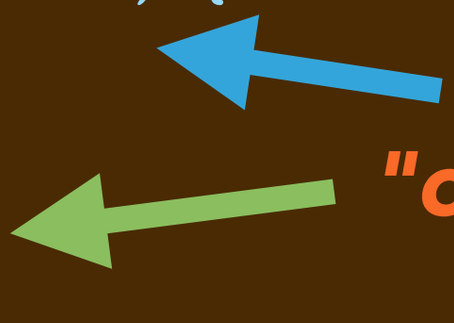


"caller"

RECALL: FUNCTIONS IN C++

- Consider this C++ program:

```
1.  int two_digits(int tens, int ones) {
2.      return 10 * tens + ones;
3.  }
4.  int times100(int number) {
5.      return 100 * number;
6.  }
7.  int main(void) { int A, B, C, D;
8.      cin >> A;
9.      cin >> B;
10.     cin >> C;
11.     cin >> D;
12.     int hi = two_digits(A,B);
13.     int lo = two_digits(C,D);
14.     int n = times100(hi) + lo;
15.     cout << n << endl;
16. }
```



"callees"

"caller"

RECALL: FUNCTIONS IN C++

- Consider this C++ program:

```
1.  int two_digits(int tens, int ones) {
2.      return 10 * tens + ones;
3.  }
4.  int times100(int number) {
5.      return 100 * number;
6.  }
7.  int main(void) { int A, B, C, D;
8.      cin >> A;
9.      cin >> B;
10.     cin >> C;
11.     cin >> D;
12.     int hi = two_digits(A,B);
13.     int lo = two_digits(C,D);
14.     int n = times100(hi) + lo;
15.     cout << n << endl;
16. }
```

This program takes in four digits as input, and then computes and outputs a four digit integer with those digits.

RECALL: FUNCTIONS IN C++

- Consider this C++ program:

```
1.  int two_digits(int tens, int ones) {
2.      return 10 * tens + ones;
3.  }
4.  int times100(int number) {
5.      return 100 * number;
6.  }
7.  int main(void) { int A, B, C, D;
8.      cin >> A;
9.      cin >> B;
10.     cin >> C;
11.     cin >> D;
12.     int hi = two_digits(A,B);
13.     int lo = two_digits(C,D);
14.     int n = times100(hi) + lo;
15.     cout << n << endl;
16. }
```

CONSOLE

```
1 5
2 1
3 3
4 7
5
```

This program takes in four digits as input, and then computes and outputs a four digit integer with those digits.

RECALL: FUNCTIONS IN C++

- Consider this C++ program:

```
1.  int two_digits(int tens, int ones) {
2.      return 10 * tens + ones;
3.  }
4.  int times100(int number) {
5.      return 100 * number;
6.  }
7.  int main(void) { int A, B, C, D;
8.      cin >> A;
9.      cin >> B;
10.     cin >> C;
11.     cin >> D;
12.     int hi = two_digits(A,B);
13.     int lo = two_digits(C,D);
14.     int n = times100(hi) + lo;
15.     cout << n << endl;
16. }
```

CONSOLE

```
1 5
2 1
3 3
4 7
5 5137
```

This program takes in four digits as input, and then computes and outputs a four digit integer with those digits.

RECALL: FUNCTIONS IN C++

- Here is a different version:

```
1.  int two_digits(int tens, int ones) {
2.      return 10 * tens + ones;
3.  }
4.  int times100(int number) {
5.      return 100 * number;
6.  }
7.  int four_digits(int w,int x,int y,int z) {
8.      return times100(two_digits(w,x)) + two_digits(y,z);
9.  }
10. int main(void) { int A, B, C, D;
11.     cin >> A;
12.     cin >> B;
13.     cin >> C;
14.     cin >> D;
15.     cout << four_digits(A,B,C,D) << endl;
16. }
```

RECALL: FUNCTIONS IN C++

- Here is a different version:

```
1.  int two_digits(int tens, int ones) {
2.      return 10 * tens + ones;
3.  }
4.  int times100(int number) {
5.      return 100 * number;
6.  }
7.  int four_digits(int w,int x,int y,int z) {
8.      return times100(two_digits(w,x)) + two_digits(y,z);
9.  }
10. int main(void) { int A, B, C, D;
11.     cin >> A;
12.     cin >> B;
13.     cin >> C;
14.     cin >> D;
15.     cout << four_digits(A,B,C,D) << endl;
16. }
```

- We're going to work to convert this and the earlier example into MIPS code.

FIRST, A DIGRESSION...

- My plan is to talk about call/return and the call stack.
  - ➡ But first, let's talk about multiplication...

MULTIPLICATION

- Consider these two expressions

```
return 10 * tens + ones;

return 100 * number;
```

- Q: How do we perform those multiplications in MIPS?

MULTIPLICATION

- Consider these two expressions

```
return 10 * tens + ones;

return 100 * number;
```

- Q: How do we perform those multiplications in MIPS?
- A1: Repeated addition.

MULTIPLICATION

- Consider these two expressions

```
return 10 * tens + ones;

return 100 * number;
```

- Q: How do we perform those multiplications in MIPS?
- A1: Repeated addition. *Not how multiplication is perfomed. Too slow.*

MULTIPLICATION

- Consider these two expressions

```
return 10 * tens + ones;

return 100 * number;
```

- Q: How do we perform those multiplications in MIPS?
- A1: Repeated addition. *Not how multiplication is perfomed. Too slow.*
- A2: Use the MIPS **`MULT`** instruction, along with **`MFLO`** and **`MFHI`**

MULTIPLICATION

- Consider these two expressions

```
return 10 * tens + ones;

return 100 * number;
```

- Q: How do we perform those multiplications in MIPS?
- A1: Repeated addition. *Not how multiplication is perfomed. Too slow.*
- A2: Use the MIPS **`MULT`** instruction, along with **`MFLO`** and **`MFHI`**
- **A3: That's probably the best way. But let's consider a third way...**

ANSWER 3: USE BIT SHIFTING OPERATIONS

- Using built-in multiplication is fine, but there is another way, too.
- **RECALL:** multiplying by two will shift the bits of a number left:

```
   111     <= binary for the value 7
  1110     <= binary for the value 2*7=14
111000     <= binary for the value 8*7=56
```

ANSWER 3: USE BIT SHIFTING OPERATIONS

- Using built-in multiplication is fine, but there is another way, too.
- **RECALL:** multiplying by two will shift the bits of a number left:

```
   111     <= binary for the value 7
  1110     <= binary for the value 2*7=14
111000     <= binary for the value 8*7=56
```

- **Q:** So how might we multiply by 10?

ANSWER 3: USE BIT SHIFTING OPERATIONS

- Using built-in multiplication is fine, but there is another way, too.
- **RECALL:** multiplying by two will shift the bits of a number left:

```
111      <= binary for the value 7
1110     <= binary for the value 2*7=14
111000     <= binary for the value 8*7=56
```

- **Q:** So how might we multiply by 10?
- NOTE: $10\,x = (2+8)\,x = 2\,x + 8\,x$

ANSWER 3: USE BIT SHIFTING OPERATIONS

- Using built-in multiplication is fine, but there is another way, too.
- **RECALL:** multiplying by two will shift the bits of a number left:

```
111      <= binary for the value 7
1110     <= binary for the value 2*7=14
111000   <= binary for the value 8*7=56
```

- **Q:** So how might we multiply by 10?
- NOTE: $10\,x = (2+8)\,x = 2\,x + 8\,x$
- **A:** We can multiply by 2, then by 8, and sum the two results.
- **I.E...**

ANSWER 3: USE BIT SHIFTING OPERATIONS

- Using built-in multiplication is fine, but there is another way, too.
- **RECALL:** multiplying by two will shift the bits of a number left:

```
111      <= binary for the value 7
1110     <= binary for the value 2*7=14
111000   <= binary for the value 8*7=56
```

- **Q:** So how might we multiply by 10?
- NOTE: $10\,x = (2+8)\,x = 2\,x + 8\,x$
- **A:** We can multiply by 2, then by 8, and sum the two results.
- **I.E.** We can shift left one bit and also shift left three bits. Then add.

ANSWER 3: USE BIT SHIFTING OPERATIONS

- The code below uses the **SLL** instruction to do exactly that with t0:

```
sll  $t1,$t0,1
sll  $t2,$t0,3
addu $t0,$t1,$t2
```

ANSWER 3: USE BIT SHIFTING OPERATIONS

- The code below uses the **SLL** instruction to do exactly that with t0:

```
sll  $t1,$t0,1
sll  $t2,$t0,3                  tmp = tmp * 10
addu $t0,$t1,$t2
```

- It has the effect of multiplying t0 by 10.

ANSWER 3: USE BIT SHIFTING OPERATIONS

- The code below uses the **SLL** instruction to do exactly that with t0:

```
sll  $t1,$t0,1
sll  $t2,$t0,3
addu $t0,$t1,$t2
```

```
tmp = tmp * 10
```

- It has the effect of multiplying t0 by 10.
- **Q:** So how might we multiply by 100?

ANSWER 3: USE BIT SHIFTING OPERATIONS

- The code below uses the **SLL** instruction to do exactly that with t0:

```
sll  $t1,$t0,1
sll  $t2,$t0,3
addu $t0,$t1,$t2
```

```
tmp = tmp * 10
```

- It has the effect of multiplying t0 by 10.
- **Q:** So how might we multiply by 100?
- SAME IDEA: 100 = 64 + 32 + 4
- **A:** So we shift 2, 5, and 6 places left. Add.

MULTIPLICATION BY 100

- The code below multiplies t0 by 100:

```
sll  $t1,$t0,2
sll  $t2,$t0,5
sll  $t3,$t0,6
addu $t0,$t1,$t2
addu $t0,$t0,$t3
```

BACK TO OUR EXAMPLE C++

```
1.  int two_digits(int tens, int ones) {
2.    return 10 * tens + ones;
3.  }
4.  int times100(int number) {
5.    return 100 * number;
6.  }
7.  int main(void) {
8.    int A, B, C, D;
9.    cin >> A;
10.   cin >> B;
11.   cin >> C;
12.   cin >> D;
13.   int hi = two_digits(A,B);
14.   int lo = two_digits(C,D);
15.   int n = times100(hi) + lo;
16.   cout << n << endl;
17. }
```

RENAME VARIABLES WITH REGISTERS

```
1.  int two_digits(int a0,a1) {
2.    return 10 * a0 + a1;
3.  }
4.  int times100(int a0) {
5.    return 100 * a0;
6.  }
7.  int main(void) {
8.    int s0,s1,s2,s3;
9.    cin >> s0;
10.   cin >> s1;
11.   cin >> s2;
12.   cin >> s3;
13.   int s0 = two_digits(s0,s1);
14.   int s1 = two_digits(s2,s3);
15.   int v0 = times100(s0) + s1;
16.   cout << v0 << endl;
17. }
```

LEFT: C++

```
1.  int two_digits(int a0,a1) {
2.    return 10 * a0 + a1;
3.  }
4.  int times100(int a0) {
5.    return 100 * a0;
6.  }
7.  int main(void) {
8.    int s0,s1,s2,s3;
9.    cin >> s0;
10.   cin >> s1;
11.   cin >> s2;
12.   cin >> s3;
13.   int s0 = two_digits(s0,s1);
14.   int s1 = two_digits(s2,s3);
15.   int v0 = times100(s0) + s1;
16.   cout << v0 << endl;
17. }
```

RIGHT: MIPS FOR MAIN

```
main:
... # syscalls to get s0-s3
    ...
    move $a0,$s0
    move $a1,$s1
    jal  two_digits
    move $s0,$v0

    move $a0,$s2
    move $a1,$s3
    jal  two_digits
    move $s1,$v0

    move $a0,$s0
    jal  times100
    add  $a0,$v0,$s1
    ...
... # syscall to output $a0
```

LEFT: C++

```
1.  int two_digits(int a0,a1) {
2.    return 10 * a0 + a1;
3.  }
4.  int times100(int a0) {
5.    return 100 * a0;
6.  }
7.  int main(void) {
8.    int s0,s1,s2,s3;
9.    cin >> s0;
10.   cin >> s1;
11.   cin >> s2;
12.   cin >> s3;
13.   int s0 = two_digits(s0,s1);
14.   int s1 = two_digits(s2,s3);
15.   int v0 = times100(s0) + s1;
16.   cout << v0 << endl;
17. }
```

RIGHT: MIPS FOR MAIN

```
main:
... # syscalls to get s0-s3
    ...
    move $a0,$s0
    move $a1,$s1
    jal  two_digits
    move $s0,$v0

    move $a0,$s2
    move $a1,$s3
    jal  two_digits
    move $s1,$v0

    move $a0,$s0
    jal  times100
    add  $a0,$v0,$s1
    ...
... # syscall to output $a0
```

LEFT: C++

```
1.  int two_digits(int a0,a1) {
2.    return 10 * a0 + a1;
3.  }
4.  int times100(int a0) {
5.    return 100 * a0;
6.  }
7.  int main(void) {
8.    int s0,s1,s2,s3;
9.    cin >> s0;
10.   cin >> s1;
11.   cin >> s2;
12.   cin >> s3;
13.   int s0 = two_digits(s0,s1);
14.   int s1 = two_digits(s2,s3);
15.   int v0 = times100(s0) + s1;
16.   cout << v0 << endl;
17. }
```

RIGHT: MIPS FOR MAIN

```
main:
... # syscalls to get s0-s3
    ...
    move $a0,$s0
    move $a1,$s1
    jal  two_digits
    move $s0,$v0

    move $a0,$s2
    move $a1,$s3
    jal  two_digits
    move $s1,$v0

    move $a0,$s0
    jal  times100
    add  $a0,$v0,$s1
    ...
... # syscall to output $a0
```

LEFT: C++

```
1.  int two_digits(int a0,a1) {
2.    return 10 * a0 + a1;
3.  }
4.  int times100(int a0) {
5.    return 100 * a0;
6.  }
7.  int main(void) {
8.    int s0,s1,s2,s3;
9.    cin >> s0;
10.   cin >> s1;
11.   cin >> s2;
12.   cin >> s3;
13.   int s0 = two_digits(s0,s1);
14.   int s1 = two_digits(s2,s3);
15.   int v0 = times100(s0) + s1;
16.   cout << v0 << endl;
17. }
```

RIGHT: MIPS FOR MAIN

```
main:
... # syscalls to get s0-s3
    ...
    move $a0,$s0
    move $a1,$s1
    jal  two_digits
    move $s0,$v0

    move $a0,$s2
    move $a1,$s3
    jal  two_digits
    move $s1,$v0

    move $a0,$s0
    jal  times100
    add  $a0,$v0,$s1
    ...
... # syscall to output $a0
```

LEFT: C++

```
1.  int two_digits(int a0,a1) {
2.    return 10 * a0 + a1;
3.  }
4.  int times100(int a0) {
5.    return 100 * a0;
6.  }
7.  int main(void) {
8.    int s0,s1,s2,s3;
9.    cin >> s0;
10.   cin >> s1;
11.   cin >> s2;
12.   cin >> s3;
13.   int s0 = two_digits(s0,s1);
14.   int s1 = two_digits(s2,s3);
15.   int v0 = times100(s0) + s1;
16.   cout << v0 << endl;
17. }
```

RIGHT: MIPS FOR MAIN

```
main:
... # syscalls to get s0-s3
    ...
    move $a0,$s0
    move $a1,$s1
    jal  two_digits
    move $s0,$v0

    move $a0,$s2
    move $a1,$s3
    jal  two_digits
    move $s1,$v0

    move $a0,$s0
    jal  times100
    add  $a0,$v0,$s1
    ...
... # syscall to output $a0
```

LEFT: C++

```
1.  int two_digits(int a0,a1) {
2.    return 10 * a0 + a1;
3.  }
4.  int times100(int a0) {
5.    return 100 * a0;
6.  }
7.  int main(void) {
8.    int s0,s1,s2,s3;
9.    cin >> s0;
10.   cin >> s1;
11.   cin >> s2;
12.   cin >> s3;
13.   int s0 = two_digits(s0,s1);
14.   int s1 = two_digits(s2,s3);
15.   int v0 = times100(s0) + s1;
16.   cout << v0 << endl;
17. }
```

RIGHT: MIPS FOR MAIN

call sites

```
main:
... # syscalls to get s0-s3
    ...
    move $a0,$s0
    move $a1,$s1
    jal  two_digits
    move $s0,$v0

    move $a0,$s2
    move $a1,$s3
    jal  two_digits
    move $s1,$v0

    move $a0,$s0
    jal  times100
    add  $a0,$v0,$s1
    ...
... # syscall to output $a0
```

CALLING A FUNCTION IN MIPS

- To mimic the C++ call `int s0 = two_digits(s0,s1);`
- ...we write this MIPS code:

```
move $a0,$s0
move $a1,$s1
jal  two_digits
move $s0,$v0
```

- NOTES:

CALLING A FUNCTION IN MIPS

- To mimic the C++ call `int s0 = two_digits(s0,s1);`
- ...we write this MIPS code:

```
move $a0,$s0
move $a1,$s1
jal  two_digits
move $s0,$v0
```

- NOTES:
  - We pass parameters using the argument registers **a0-a3**

CALLING A FUNCTION IN MIPS

- To mimic the C++ call **`int s0 = two_digits(s0,s1);`**
- ...we write this MIPS code:

```
move $a0,$s0
move $a1,$s1
jal  two_digits
move $s0,$v0
```

- NOTES:
  - We pass parameters using the argument registers **a0-a3**
  - We extract the return value from register **v0**.

CALLING A FUNCTION IN MIPS

- To mimic the C++ call `int s0 = two_digits(s0,s1);`
- ...we write this MIPS code:

```
move $a0,$s0
move $a1,$s1
jal  two_digits
move $s0,$v0
```

- NOTES:
  - We pass parameters using the argument registers **a0-a3**
  - We extract the return value from register **v0**.
  - We use the **JAL** instruction to "*jump and link*" to a **labelled code line**.

CALLING A FUNCTION IN MIPS

- To mimic the C++ call `int s0 = two_digits(s0,s1);`
- ...we write this MIPS code:

```
move $a0,$s0
move $a1,$s1
jal  two_digits
move $s0,$v0
```

- NOTES:
  - We pass parameters using the argument registers **a0-a3**
  - We extract the return value from register **v0**.
  - We use the **JAL** instruction to "*jump and link*" to a **labelled code line**.
  - This saves the **line after the jump** into a register named **ra**.

CALLING A FUNCTION IN MIPS

- To mimic the C++ call `int s0 = two_digits(s0,s1);`
- ...we write this MIPS code:

```
move $a0,$s0
move $a1,$s1
jal  two_digits
move $s0,$v0
```

the "return address" for this call site is here

- NOTES:
  - We pass parameters using the argument registers **a0-a3**
  - We extract the return value from register **v0**.
  - We use the **JAL** instruction to "*jump and link*" to a **labelled code line**.
  - This saves the **line after the jump** into a register named **ra**.

CALLING A FUNCTION IN MIPS

- The ***callee*** two_digits can assume the ***caller*** followed those conventions.

```
move $a0,$s0
move $a1,$s1
jal  two_digits
move $s0,$v0
```

```
two_digits:
  sll $t0,$a0,1
  sll $t1,$a0,3
  add $v0,$t1,$t0
  add $v0,$v0,$a1
  jr  $ra
```

- NOTES:

CALLING A FUNCTION IN MIPS

- The ***callee*** two_digits can assume the ***caller*** followed those conventions.

```
    move $a0,$s0
    move $a1,$s1
    jal  two_digits
    move $s0,$v0
```

```
two_digits:
  sll $t0,$a0,1
  sll $t1,$a0,3
  add $v0,$t1,$t0
  add $v0,$v0,$a1
  jr  $ra
```

- NOTES:

CALLING A FUNCTION IN MIPS

- The ***callee*** two_digits can assume the ***caller*** followed those conventions.

```
    move $a0,$s0
    move $a1,$s1
    jal  two_digits
    move $s0,$v0
```

```
two_digits:
    sll $t0,$a0,1
    sll $t1,$a0,3
    add $v0,$t1,$t0
    add $v0,$v0,$a1
    jr  $ra
```

- NOTES:

CALLING A FUNCTION IN MIPS

- The ***callee*** two_digits can assume the ***caller*** followed those conventions.

```
    move $a0,$s0
    move $a1,$s1
    jal  two_digits
    move $s0,$v0
```

```
two_digits:
    sll $t0,$a0,1
    sll $t1,$a0,3
    add $v0,$t1,$t0
    add $v0,$v0,$a1
    jr  $ra
```

- NOTES:

CALLING A FUNCTION IN MIPS

- The ***callee*** two_digits can assume the ***caller*** followed those conventions.

```
    move $a0,$s0
    move $a1,$s1
    jal  two_digits
    move $s0,$v0
```

```
two_digits:
    sll $t0,$a0,1
    sll $t1,$a0,3
    add $v0,$t1,$t0
    add $v0,$v0,$a1
    jr  $ra
```

- NOTES:

CALLING A FUNCTION IN MIPS

- The ***callee*** two_digits can assume the ***caller*** followed those conventions.

```
    move $a0,$s0
    move $a1,$s1
    jal  two_digits
    move $s0,$v0
```

```
two_digits:
  sll $t0,$a0,1
  sll $t1,$a0,3
  add $v0,$t1,$t0
  add $v0,$v0,$a1
  jr  $ra
```

- NOTES:
  - These steps are the "jump and link" followed by the "jump back" (return).

~~CALLING A FUNCTION IN MIPS~~ WRITING A FUNCTION IN MIPS

- The ***callee*** two_digits can assume the ***caller*** followed those conventions.

```
move $a0,$s0
move $a1,$s1
jal  two_digits
move $s0,$v0
```

```
two_digits:
  sll $t0,$a0,1
  sll $t1,$a0,3
  add $v0,$t1,$t0
  add $v0,$v0,$a1
  jr  $ra
```

- NOTES:

~~CALLING A FUNCTION IN MIPS~~ WRITING A FUNCTION IN MIPS

- The ***callee*** two_digits can assume the ***caller*** followed those conventions.

```
move $a0,$s0
move $a1,$s1
jal  two_digits
move $s0,$v0
```

```
two_digits:
  sll $t0,$a0,1
  sll $t1,$a0,3
  add $v0,$t1,$t0
  add $v0,$v0,$a1
  jr  $ra
```

- NOTES:
  - It grabs its two parameters from **a0** and **a1**.

~~CALLING A FUNCTION IN MIPS~~ WRITING A FUNCTION IN MIPS

- The ***callee*** two_digits can assume the ***caller*** followed those conventions.

```
move $a0,$s0
move $a1,$s1
jal  two_digits
move $s0,$v0
```

```
two_digits:
  sll $t0,$a0,1
  sll $t1,$a0,3
  add $v0,$t1,$t0
  add $v0,$v0,$a1
  jr  $ra
```

- NOTES:
  - It grabs its two parameters from **a0** and **a1**.
  - It computes its result and puts it into **v0**.

~~CALLING A FUNCTION IN MIPS~~ WRITING A FUNCTION IN MIPS

- The ***callee*** two_digits can assume the ***caller*** followed those conventions.

```
move $a0,$s0
move $a1,$s1
jal  two_digits
move $s0,$v0
```

```
two_digits:
  sll $t0,$a0,1
  sll $t1,$a0,3
  add $v0,$t1,$t0
  add $v0,$v0,$a1
  jr  $ra
```

- NOTES:
  - It grabs its two parameters from **a0** and **a1**.
  - It computes its result and puts it into **v0**.
  - It jumps back to the caller using the address value stored in **ra**.

THE PROBLEMS WITH REGISTER CHOICE

- Suppose we had instead used **t0**-**t3** in main, rather than **s0**-**s3**...:

```
    move $a0,$s0
    move $a1,$s1
    jal  two_digits
    move $s0,$v0

    move $a0,$s2
    move $a1,$s3
    jal  two_digits
    move $s1,$v0
...
    move $a0,$s0
    jal  times100
```

```
two_digits:
    sll $t0,$a0,1
    sll $t1,$a0,3
    add $v0,$t1,$t0
    add $v0,$v0,$a1
    jr  $ra
```

- ...

THE PROBLEMS WITH REGISTER CHOICE

- Suppose we had instead used **t0**-**t3** in main, rather than **s0**-**s3**...:

```
    move $a0,$t0
    move $a1,$t1
    jal  two_digits
    move $t0,$v0

    move $a0,$t2
    move $a1,$t3
    jal  two_digits
    move $t1,$v0
...
    move $a0,$t0
    jal  times100
```

- ...

```
two_digits:
    sll $t0,$a0,1
    sll $t1,$a0,3
    add $v0,$t1,$t0
    add $v0,$v0,$a1
    jr  $ra
```

THE PROBLEMS WITH REGISTER CHOICE

- Suppose we had instead used t0-t3 in main, rather than s0-s3...:

```
    move $a0,$t0
    move $a1,$t1
    jal  two_digits
    move $t0,$v0

    move $a0,$t2
    move $a1,$t3
    jal  two_digits
    move $t1,$v0
...
    move $a0,$t0
    jal  times100
```

```
two_digits:
    sll $t0,$a0,1
    sll $t1,$a0,3
    add $v0,$t1,$t0
    add $v0,$v0,$a1
    jr  $ra
```

THE PROBLEMS WITH REGISTER CHOICE

- Suppose we had instead used t0-t3 in main, rather than s0-s3...:

```
    move $a0,$t0
    move $a1,$t1
    jal  two_digits
    move $t0,$v0

    move $a0,$t2
    move $a1,$t3
    jal  two_digits
    move $t1,$v0
...
    move $a0,$t0
    jal  times100
```

@!$%$

```
two_digits:
    sll $t0,$a0,1
    sll $t1,$a0,3
    add $v0,$t1,$t0
    add $v0,$v0,$a1
    jr  $ra
```

!!!!!

- ...then the second call to two_digits would "**clobber**" the value held in **t0**.

THE PROBLEMS WITH REGISTER CHOICE

- We only have a fixed number of registers to work with.
  - ➡ In large enough program, you'll "run out " of variables.
  - ➡ Store some of a function's variables in memory. Use ***function frames***.
- As we know, C++ allows us to have an arbitrarily deep number of calls.
  - Because of recursion, a function could have several outstanding calls.
  - Because of recursion, we cannot predict the depth of the calls.
    - ➡ We manage a ***call stack*** of function frames.

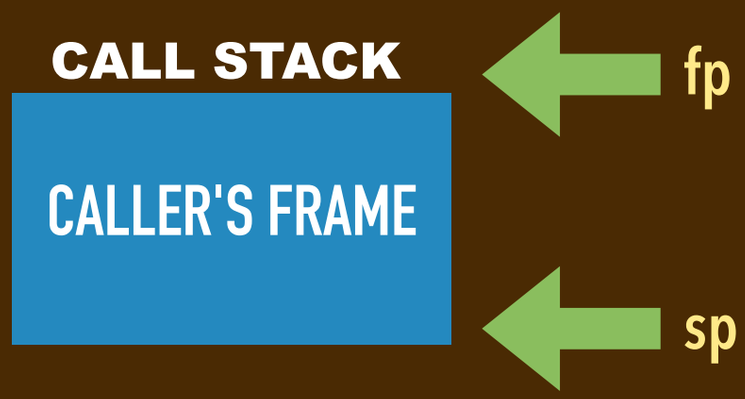
THE PROBLEMS WITH REGISTER CHOICE (CONT'D)

- We still have the problem of avoiding register conflicts.
  - ➡ Adopt conventions to guide us in ***saving/restoring*** registers with calls.
  - There are registers that the caller must save. ***caller-saved registers***
  - There are registers that the caller can assume the callee preserves. ***callee-saved registers***

STACK FRAME DISCIPLINE

- The MIPS calling conventions designate that...
  - register **fp** points to the byte just above the top of a function's frame.
  - register sp points to the byte just at the bottom of a function's frame
- ...and that the callee ***preserve the caller's frame***.

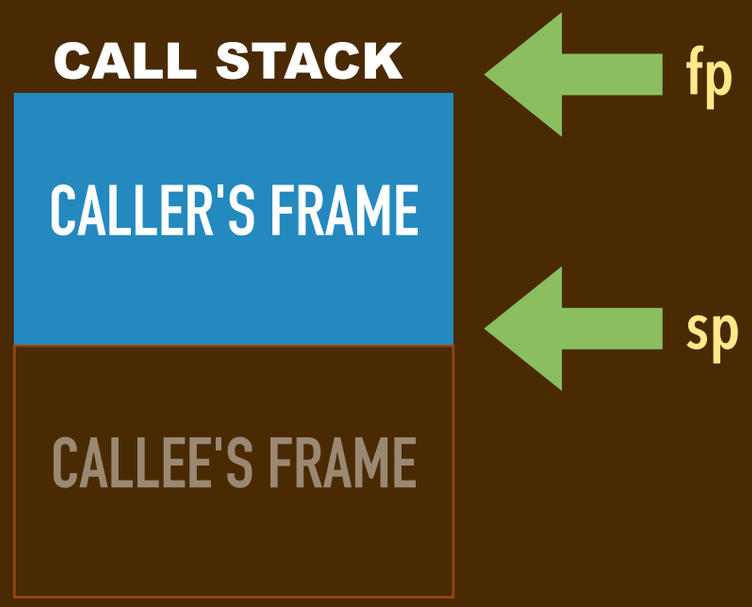
BEFORE THE CALL



STACK FRAME DISCIPLINE

- The MIPS calling conventions designate that...
  - register **fp** points to the byte just above the top of a function's frame.
  - register sp points to the byte just at the bottom of a function's frame
- ...and that the callee ***preserve the caller's frame***.

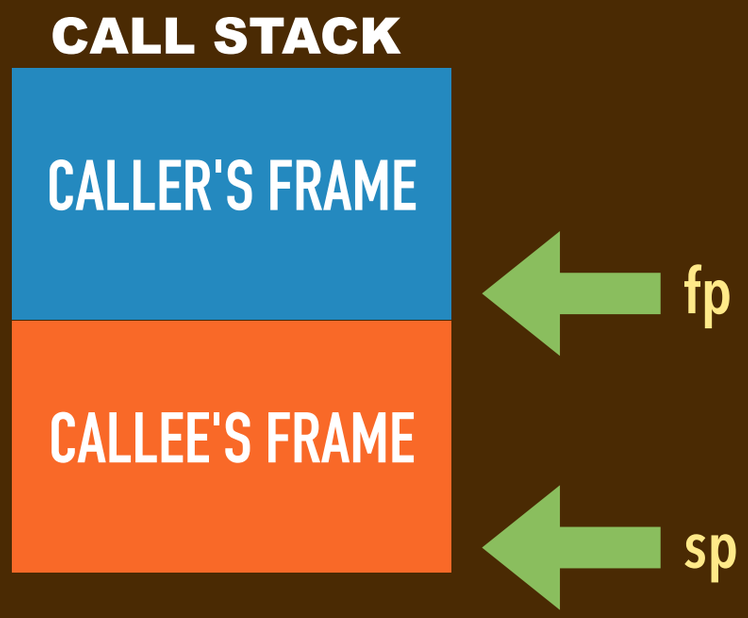
BEFORE THE CALL



STACK FRAME DISCIPLINE

- The MIPS calling conventions designate that...
  - register **fp** points to the byte just above the top of a function's frame.
  - register sp points to the byte just at the bottom of a function's frame
- ...and that the callee ***preserve the caller's frame***.

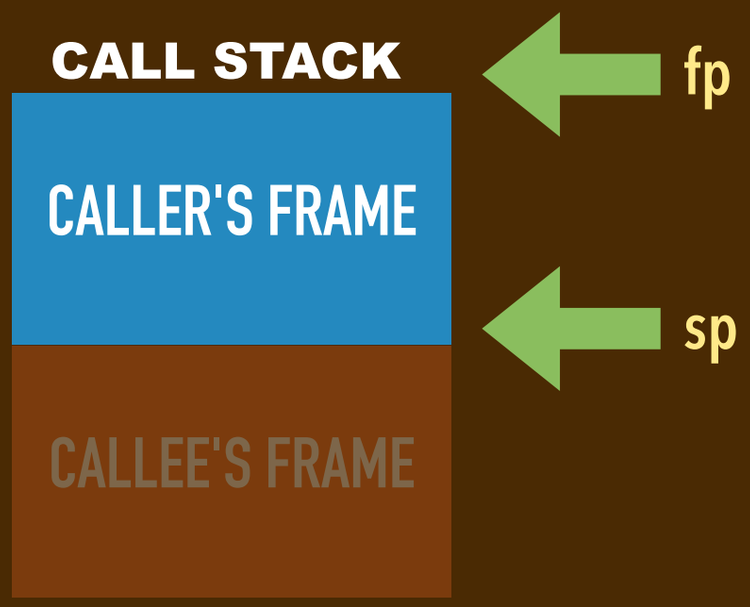
DURING THE CALL



STACK FRAME DISCIPLINE

- The MIPS calling conventions designate that...
  - register **fp** points to the byte just above the top of a function's frame.
  - register sp points to the byte just at the bottom of a function's frame
- ...and that the callee ***preserve the caller's frame***.

AFTER THE CALL



STACK FRAME DISCIPLINE (CONT'D)

- The MIPS calling conventions designate that...
  - the frame size should be at least 32 bytes
  - the addresses in fp and sp should be ***word-aligned*** (multiples of 4).
  - (some say they should be ***double-word aligned*** (multiples of 8)

CALLEE-SAVED REGISTERS

- The MIPS calling conventions designate that...
  - Registers need to be preserved with a function call. No clobbering!
- Some registers are "***callee-saved***"
  - ➡ The function called must save the values of these registers on the stack before using them.
  - ➡ It must restore their values from the stack before it retuns to the caller.
  - These registers' values are guaranteed to be preserved with a function call.

CALLER-SAVED REGISTERS

- The MIPS calling conventions designate that...
  - Registers need to be preserved with a function call. No clobbering!
- Some registers are "***caller-saved***"
  - ➡ The caller saves these on the stack before calling a function.
  - ➡ The caller restores them from the stack after the call.
- These registers' values may not be preserved with a function call.

FOUR_DIGITS IN C++

```
1.  int two_digits(int tens, int ones) {
2.      return 10 * tens + ones;
3.  }
4.  int times100(int number) {
5.      return 100 * number;
6.  }
7.  int four_digits(int w,int x,int y,int z) {
8.      return times100(two_digits(w,x)) + two_digits(y,z);
9.  }
10. int main(void) { int A, B, C, D;
11.     cin >> A;
12.     cin >> B;
13.     cin >> C;
14.     cin >> D;
15.     cout << four_digits(A,B,C,D) << endl;
16. }
```

FOUR_DIGITS IN MIPS (VERSION 1)

```
four_digits:
    move $s0,$a0
    move $s1,$a1
    move $s2,$a2
    move $s3,$a3

    move $a0,$s0
    move $a1,$s1
    jal  two_digits
    move $s0,$v0

    move $a0,$s2
    move $a1,$s3
    jal  two_digits
    move $s1,$v0

    move $a0,$s0
    jal  times100
    add  $v0,$v0,$s1

    jr $ra
```

FOUR_DIGITS IN MIPS (VERSION 1)

```
four_digits:
     move $s0,$a0
     move $s1,$a1
     move $s2,$a2
     move $s3,$a3

     move $a0,$s0
     move $a1,$s1
     jal  two_digits
     move $s0,$v0

     move $a0,$s2
     move $a1,$s3
     jal  two_digits
     move $s1,$v0

     move $a0,$s0
     jal  times100
     add  $v0,$v0,$s1

     jr $ra
```

No real need to copy into s0-s3 here. Can access directly,

FOUR_DIGITS IN MIPS (VERSION 2) USING ARGUMENTS DIRECTLY

```
four_digits:
    # Forward $a0 and $a1 to the call to two_digits.
    jal  two_digits
    move $s0,$v0

    move $a0,$a2
    move $a1,$a3
    jal  two_digits
    move $s1,$v0

    move $a0,$s0
    jal  times100
    add  $v0,$v0,$s1

    jr $ra
```

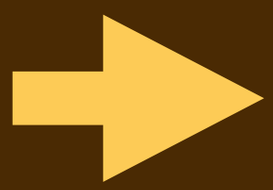
FOUR_DIGITS IN MIPS (VERSION 2) USING ARGUMENTS DIRECTLY

```
four_digits:
    # Forward $a0 and $a1 to the call to two_digits.
    jal  two_digits
    move $s0,$v0

    move $a0,$a2
    move $a1,$a3
    jal  two_digits
    move $s1,$v0

    move $a0,$s0
    jal  times100
    add  $v0,$v0,$s1

    jr $ra
```



Let's follow the stack discipline, lay out a frame.

FOUR_DIGITS IN MIPS (VERSION 3) USING STACK FRAME

```
four_digits:
    sw   $ra,-4($sp) # Save $ra so we can make calls too.
    sw   $fp,-8($sp) # Save caller's frame pointer.
    move $fp,$sp     # Set up our frame pointer (frame top).
    addi $sp,$sp,-32 # Set up the bottom of our frame.
    jal  two_digits
    move $s0,$v0
    move $a0,$a2
    move $a1,$a3
    jal  two_digits
    move $s1,$v0
    move $a0,$s0
    jal  times100
    add  $v0,$v0,$s1 # Set up return value.
    addi $sp,$sp,32  # Restore caller's frame bottom.
    lw   $fp,-8($sp) # Restore caller's frame pointer.
    lw   $ra,-4($sp) # Restore caller's return address.
    jr   $ra         # Return.
```

FOUR_DIGITS IN MIPS (VERSION 4) SAVING S REGISTERS

```
four_digits:
    sw   $ra,-4($sp)
    sw   $fp,-8($sp)
    sw   $s0,-12($sp) # By convention, S registers must be preserved
    sw   $s1,-16($sp) # They are "callee-saved" registers.
    move $fp,$sp
    addi $sp,$sp,-32
    jal  two_digits
    move $s0,$v0
    move $a0,$a2
    move $a1,$a3
    jal  two_digits
    move $s1,$v0
    move $a0,$s0
    jal  times100
    add  $v0,$v0,$s1
    addi $sp,$sp,32
    lw   $s1,-16($sp) # Restore $s0 and $s1 for the caller.
    lw   $s0,-12($sp) #
    lw   $fp,-8($sp)
    lw   $ra,-4($sp)
    jr   $ra
```

FOUR_DIGITS IN MIPS (VERSION 4) SAVING S REGISTERS

```
four_digits:
    sw   $ra,-4($sp)
    sw   $fp,-8($sp)
    sw   $s0,-12($sp) # By convention, S registers must be preserved
    sw   $s1,-16($sp) # They are "callee-saved" registers.
    move $fp,$sp
    addi $sp,$sp,-32
    jal  two_digits
    move $s0,$v0        # We save S registers because we use s0 here...
    move $a0,$a2
    move $a1,$a3
    jal  two_digits
    move $s1,$v0        # and s1 here
    move $a0,$s0        # and s0 here
    jal  times100
    add  $v0,$v0,$s1    # and s1 here.
    addi $sp,$sp,32
    lw   $s1,-16($sp) # Restore $s0 and $s1 for the caller.
    lw   $s0,-12($sp) #
    lw   $fp,-8($sp)
    lw   $ra,-4($sp)
    jr   $ra
```

FOUR_DIGITS IN MIPS (VERSION 4) SAVING S REGISTERS

```
four_digits:
    sw   $ra,-4($sp)
    sw   $fp,-8($sp)
    sw   $s0,-12($sp) # By convention, S registers must be preserved
    sw   $s1,-16($sp) # They are "callee-saved" registers.
    move $fp,$sp
    addi $sp,$sp,-32
    jal  two_digits
    move $s0,$v0
    move $a0,$a2
    move $a1,$a3
    jal  two_digits
    move $s1,$v0
    move $a0,$s0
    jal  times100
    add  $v0,$v0,$s1
    addi $sp,$sp,32
    lw   $s1,-16($sp) # Restore $s0 and $s1 for the caller.
    lw   $s0,-12($sp) #
    lw   $fp,-8($sp)
    lw   $ra,-4($sp)
    jr   $ra
```

- *Even better, we can assume two_digits follows this same convention.*

➡*It saves/restores the S registers too.*

FOUR_DIGITS IN MIPS (VERSION 4) SAVING S REGISTERS

```
four_digits:
    sw   $ra,-4($sp)
    sw   $fp,-8($sp)
    sw   $s0,-12($sp) # By convention, S registers must be preserved
    sw   $s1,-16($sp) # They are "callee-saved" registers.
    move $fp,$sp
    addi $sp,$sp,-32
    jal  two_digits
    move $s0,$v0
    move $a0,$a2
    move $a1,$a3
    jal  two_digits
    move $s1,$v0
    move $a0,$s0
    jal  times100
    add  $v0,$v0,$s1
    addi $sp,$sp,32
    lw   $s1,-16($sp) # Restore $s0 and $s1 for the caller.
    lw   $s0,-12($sp) #
    lw   $fp,-8($sp)
    lw   $ra,-4($sp)
    jr   $ra
```

- *As does times100.*
- *As the callee, it preserves S registers.*

FOUR_DIGITS IN MIPS (VERSION 4) PROBLEM!

```
four_digits:
    sw   $ra,-4($sp)
    sw   $fp,-8($sp)
    sw   $s0,-12($sp)
    sw   $s1,-16($sp)
    move $fp,$sp
    addi $sp,$sp,-32
    jal  two_digits
    move $s0,$v0
    move $a0,$a2
    move $a1,$a3
    jal  two_digits
    move $s1,$v0
    move $a0,$s0
    jal  times100
    add  $v0,$v0,$s1
    addi $sp,$sp,32
    lw   $s1,-16($sp)
    lw   $s0,-12($sp)
    lw   $fp,-8($sp)
    lw   $ra,-4($sp)
    jr   $ra
```

@!$%????

- Unfortunately we cannot assume that a2 and a3 are preserved by two_digits.

FOUR_DIGITS IN MIPS (VERSION 4) PROBLEM!

```
four_digits:
     sw   $ra,-4($sp)
     sw   $fp,-8($sp)
     sw   $s0,-12($sp)
     sw   $s1,-16($sp)
     move $fp,$sp
     addi $sp,$sp,-32
     jal  two_digits
     move $s0,$v0
     move $a0,$a2
     move $a1,$a3
     jal  two_digits
     move $s1,$v0
     move $a0,$s0
     jal  times100
     add  $v0,$v0,$s1
     addi $sp,$sp,32
     lw   $s1,-16($sp)
     lw   $s0,-12($sp)
     lw   $fp,-8($sp)
     lw   $ra,-4($sp)
     jr   $ra
```

@?$%!!!!

- Unfortunately we cannot assume that a2 and a3 are preserved by two_digits.
- We have to assume they're clobbered.

FOUR_DIGITS IN MIPS (VERSION 5) SAVING A REGISTERS

```
four_digits:
    sw   $ra,-4($sp)
    sw   $fp,-8($sp)
    sw   $s0,-12($sp)
    sw   $s1,-16($sp)
    move $fp,$sp
    addi $sp,$sp,-32
    sw   $a2,-20($fp) # Save the arguments. They are caller-saved.
    sw   $a3,-24($fp) #
    jal  two_digits
    move $s0,$v0
    lw   $a0,-20($fp)  # Restore them after the call.
    lw   $a1,-24($fp)  #
    jal  two_digits
    move $s1,$v0
    move $a0,$s0
    jal  times100
    add  $v0,$v0,$s1
    addi $sp,$sp,32
    lw   $s1,-16($sp)
    lw   $s0,-12($sp)
    lw   $fp,-8($sp)
    lw   $ra,-4($sp)
    jr   $ra
```

FOUR_DIGITS IN MIPS (VERSION 6) NOW USING T REGISTERS

```
four_digits:
    sw   $ra,-4($sp)
    sw   $fp,-8($sp)
    move $fp,$sp
    addi $sp,$sp,-32
    sw   $a2,-20($fp)
    sw   $a3,-24($fp)
    jal  two_digits
    move $t0,$v0
    lw   $a0,-20($fp)
    lw   $a1,-24($fp)
    jal  two_digits
    move $t1,$v0
    move $a0,$t0
    jal  times100
    add  $v0,$v0,$t1
    addi $sp,$sp,32
    lw   $fp,-8($sp)
    lw   $ra,-4($sp)
    jr   $ra
```

FOUR_DIGITS IN MIPS (VERSION 6) NOW USING T REGISTERS

```
four_digits:
    sw   $ra,-4($sp)
    sw   $fp,-8($sp)
    move $fp,$sp
    addi $sp,$sp,-32
    sw   $a2,-20($fp)
    sw   $a3,-24($fp)
    jal  two_digits
    move $t0,$v0
    lw   $a0,-20($fp)
    lw   $a1,-24($fp)
    jal  two_digits
    move $t1,$v0
    move $a0,$t0
    jal  times100
    add  $v0,$v0,$t1
    addi $sp,$sp,32
    lw   $fp,-8($sp)
    lw   $ra,-4($sp)
    jr   $ra
```

@?$%!!!!

@?$%!!!!

- Problem: The MIPS calling conventions designate T registers as "caller-saved."

FOUR_DIGITS IN MIPS (VERSION 6) USING T REGISTERS

```
four_digits:
    sw   $ra,-4($sp)
    sw   $fp,-8($sp)
    move $fp,$sp
    addi $sp,$sp,-32
    sw   $a2,-20($fp)
    sw   $a3,-24($fp)
    jal  two_digits
    move $t0,$v0
    sw   $t0,-12($fp)
    lw   $a0,-20($fp)
    lw   $a1,-24($fp)
    jal  two_digits
    move $t1,$v0
    sw   $t1,-16($fp)
    lw   $t0,-12($fp)
    move $a0,$t0
    jal  times100
    lw   $t1,-16($fp)
    add  $v0,$v0,$t1
    addi $sp,$sp,32
    lw   $fp,-8($sp)
    lw   $ra,-4($sp)
    jr   $ra
```

FOUR_DIGITS IN MIPS (VERSION 6) WITH SOME CLEAN-UP

```
four_digits:
    sw   $ra,-4($sp)
    sw   $fp,-8($sp)
    move $fp,$sp
    addi $sp,$sp,-32
    sw   $a2,-16($fp)
    sw   $a3,-20($fp)
    jal  two_digits
    sw   $v0,-12($fp)
    lw   $a0,-16($fp)
    lw   $a1,-20($fp)
    jal  two_digits
    lw   $a0,-12($fp)
    sw   $v0,-12($fp)
    jal  times100
    lw   $t1,-12($fp)
    add  $v0,$v0,$t1
    addi $sp,$sp,32
    lw   $fp,-8($sp)
    lw   $ra,-4($sp)
    jr   $ra
```

MIPS CALLING CONVENTIONS SUMMARY: THE CALLER

- Before the caller calls a function…
  - It saves caller-saved registers (a0-a3, t0-t9) onto its stack frame.
  - It places the parameters into registers a0-a3.
  - It pushes 5th, 6th, etc parameters onto the bottom of its stack frame.
- Using JAL saves a return address to register ra.
- After the function is called…
  - The caller restores registers it has saved, if needed.
  - It extracts the return value from register v0.

MIPS CALLING CONVENTIONS SUMMARY: THE CALLEE

- When a function is called…
  - It saves callee-saved registers (fp, sp, ra, s0-s9) onto its stack frame.
  - It extracts argument registers a0-a3 and from slots just above its frame
- Before a function returns…
  - It puts the return value into register v0.
  - It restores registers for the caller, including fp, sp, and ra.
- It then performs `JR $RA` to return control back to the caller.

MIPS CALLING CONVENTIONS SUMMARY

- ANY QUESTIONS????