LOGISTICS

- I will try a "hands up" protocol today.
  - (Zoom, I think, allows you to raise your hand.)
  - Will try to stop every 12-15 minutes for questions.
- Let's test this now...
  - Before I begin, **do you have any questions?**

LOGISTICS (CONT'D)

- I will be assigning Homework 08 today. It is due next Wednesday 3pm.
  - Please look it over tonight and/or early tomorrow.
- The TAs and I will be holding Zoom "lab office hours" tomorrow (Thursday) from 1:30-4:30pm PST, probably as 45 minute help sessions.
  - I will send Zoom meeting links so you can connect with some session.
- *Please start working on Homework 08 right away and ask questions!*

LOGISTICS (CONT'D)

- I'd like to set up office hours for the remainder of the semester...
  - I will send an office hour poll over email.
  - I'm also working to create a Slack channel for CS2 questions/discussion.
    - The TAs and I would then hover over that channel during the day.

MEMORY ACCESS IN MIPS

LECTURE 08-2

JIM FIX, REED COLLEGE CS2-S20

TODAY'S PLAN

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

- REVIEW LOAD/STORE INSTRUCTIONS, SOLUTIONS TO LEC08-1EXERCISES
- EXAMINE ARRAY REPRESENTATION AND ACCESS
- EXAMINE STRUCT REPRESENTATION AND ACCESS
- WE'LL USE LOAD/STORE AT AN OFFSET FROM AN ADDRESS
- LINKED LIST CONSTRUCTION AND TRAVERSAL

RECALL: MIPS MEMORY STUFF

- A MIPS program can reserve space for data within its 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.

```
    .data
string_ptr: .asciiz "Here is a null-terminated string."
int_ptr:    .word 101, 42, 18
area_ptr:   .space 20
    .text
main:
     la $s0, int_ptr
     la $s1, area_ptr
     lw $t0, ($s0)
     addiu $t0, $t0, 50
     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.         addiu $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.      addiu $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.      addiu $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.     addiu $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.

```
     .data
string_ptr: .asciiz "Here is a null-terminated string."
int_ptr:    .word 101, 42, 18
area_ptr:   .space 20
     .text
main:
      la $s0, int_ptr
      la $s1, area_ptr
      lw $t0, ($s0)
      addiu $t0, $t0, 50
      sw $t0, ($s1)
```

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

ARRAY-LIKE CODING

- 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.

```
        la $s0, area_ptr
        li $t1, 5           # Count out 5 inputs.
input_loop:
        beqz $t1, input_done

        li $v0, 5           #
        syscall             # Get an integer input.

        sw $v0,($s0)        # Store it in the array.
        addiu $s0,$s0,4     # Advance the pointer by 4 bytes.
        addiu $t1,$t1,-1    # Decrement the count.
        b input_loop
```

ARRAY-LIKE CODING

- 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.

```
      la $s0, area_ptr
      li $t1, 5             # Count out 5 inputs.
input_loop:
      beqz $t1, input_done

      li $v0, 5             #
      syscall               # Get an integer input.

      sw $v0,($s0)          # Store it in the array.
      addiu $s0,$s0,4       # Advance the pointer by 4 bytes.
      addiu $t1,$t1,-1      # Decrement the count.
      b input_loop
```

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

ARRAY-LIKE CODING

- 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.     addiu $s0,$s0,4     # Advance the pointer by 4 bytes.
11.     addiu $t1,$t1,-1    # Decrement the count.
12.     b input_loop
```

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

ARRAY-LIKE CODING

- 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.

```
        la $s0, area_ptr
        li $t1, 5           # Count out 5 inputs.
input_loop:
        beqz $t1, input_done

        li $v0, 5           #
        syscall             # Get an integer input.

        sw $v0,($s0)        # Store it in the array.
        addiu $s0,$s0,4     # Advance the pointer by 4 bytes.
        addiu $t1,$t1,-1    # Decrement the count.
        b input_loop
```

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

ARRAY-LIKE CODING

- 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.

```
    la $s0, area_ptr
    li $t1, 5            # Count out 5 inputs.
input_loop:
    beqz $t1, input_done

    li $v0, 5            #
    syscall              # Get an integer input.

    sw $v0,($s0)         # Store it in the array.
    addiu $s0,$s0,4      # Advance the pointer by 4 bytes.
    addiu $t1,$t1,-1     # Decrement the count.
    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).

```
        li    $t0, 0
sum_loop:
        beqz  $t1, sum_done
        lw    $t2,($s0)
        addu  $t0,$t0,$t2
        addiu $s0,$s0,4
        addiu $t1,$t1,-1
        b     sum_loop
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.      addiu $s0,$s0,4
7.      addiu $t1,$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.

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.          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.            move $a0,$t0     # output the character's code.
7.            li $v0,1
8.            syscall

10.           addiu $s0,$s0,1
11.           b loop
12.       done:
```

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

OUTPUTTING THE ASCII CODES OF A CHARACTER STRING

- This code is similar, though instead we access the bytes of a character string.

```
        la      $s0,string_ptr
loop:
        lb      $t0,($s0)   # Fetch the next character.
        beqz    $t0,done    # See if it's the null character.
                            # If it's not,
        move    $a0,$t0     # output the character's code.
        li      $v0,1
        syscall

        addiu   $s0,$s0,1
        b       loop
done:
```

OUTPUTTING THE ASCII CODES OF A CHARACTER STRING

- This code is similar, though instead we access the bytes of a character string.

```
        la      $s0,string_ptr
loop:
        lb      $t0,($s0)       # Fetch the next character.
        beqz    $t0,done        # See if it's the null character.
                                # If it's not,
        move    $a0,$t0         # output the character's code.
        li      $v0,1
        syscall

        addiu   $s0,$s0,1
        b       loop
done:
```

- These four lines load a character from the string and output its ASCII code.

OUTPUTTING THE ASCII CODES OF A CHARACTER STRING

- This code is similar, though instead we access the bytes of a character string.

```
        la      $s0,string_ptr
loop:
        lb      $t0,($s0)   # Fetch the next character.
        beqz    $t0,done    # See if it's the null character.
                            # If it's not,
        move    $a0,$t0     # output the character's code.
        li      $v0,1
        syscall

        addiu   $s0,$s0,1
        b       loop
done:
```

- Notice that we advance the pointer by only one byte (not 4).

OUTPUTTING THE ASCII CODES OF A CHARACTER STRING

- This code is similar, though instead we access 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.      move    $a0,$t0     # output the character's code.
7.      li      $v0,1
8.      syscall
9.
10.     addiu   $s0,$s0,1
11.     b       loop
12. done:
```

- Recall that character strings end with character 0.
    - Line 04 checks to see if we've hit the null-terminating character.

LECTURE 08-1 EXERCISE 1

- Change "hello.asm" so that it instead outputs this to the console

```
hello
ello
llo
lo
o
```

  - ➡Each line starts from a different place in the character string.

SOLUTION TO LECTURE 08–1 EXERCISE 1

- Changing just a few lines, we have our solution:

```
    la $s0,string_ptr
loop:
    lb $t0,($s0)
    beqz $t0,done

    move $a0,$s0          # Output the string at the pointer.
    li $v0,4
    syscall

    addiu $s0,$s0,1       # Advance the pointer by 1 byte.
    b loop
done:
```

SWAPPING CONSECUTIVE ITEMS IN AN ARRAY

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

```
int tmp1 = a[i];
int tmp2 = a[i+1];
a[i] = tmp2;
a[i+1] = tmp1;
```

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

```
    addiu $s2,$s1,4
    lw    $t1,($s1)
    lw    $t2,($s2)
    sw    $t2,($s1)
    sw    $t1,($s2)
```

SWAP, USING OFFSETS

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

```
int tmp1 = a[i];
int tmp2 = a[i+1];
a[i] = tmp2;
a[i+1] = tmp1;
```

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

```
      lw    $t1,0($s1)
      lw    $t2,4($s1)
      sw    $t2,0($s1)
      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]`**.

LECTURE 08-1 EXERCISE 2

- Change "hello.asm" so that it instead outputs this to the console

```
hello
elloh
llohe
lohel
ohell
```

  - ➡Each line is a result of rotating the string's contents by one character.

SOLUTION TO LECTURE 08-1 EXERCISE 2

- The code below "rotates" a length five string within memory

```
la $t4,hello_ptr
lb $t3,0($t4) # save the 'h'
lb $t6,1($t4)
sb $t6,0($t4) # move the 'e' left
lb $t6,2($t4)
sb $t6,1($t4) # move the 'l' left
lb $t6,3($t4)
sb $t6,2($t4) # move the 'l' left
lb $t6,4($t4)
sb $t6,3($t4) # move the 'o' left
sb $t3,4($t4) # place the 'h'
```

- I have a more general solution "rotate.asm" that relies on these two lines

```
lb $t6,1($t4)       # shift a character one spot left
sb $t6,0($t4)       #
```

COMPILER USE OF OFFSETS

- Consider this C++ code:

```
void fcn(int a, int b) {
    ...
    int x = a - b;
    int y = b + 10;
    ...
}
```

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

```
    fcn:
        ...
        lw    $t0, 0($fp)
        lw    $t1, -4($fp)
        subu  $t2,$t0,$t1
        sw    $t2, -8($fp)
        addiu $t3,$t1,10
        sw    $t3, -12($fp)
        ...
```

COMPILER USE OF OFFSETS

- Consider this C++ code:

```
void fcn(int a, int b) {
    ...
    int x = a - b;
    int y = b + 10;
    ...
}
```

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

```
fcn:
    ...
    lw    $t0, 0($fp)
    lw    $t1, -4($fp)
    subu  $t2,$t0,$t1
    sw    $t2, -8($fp)
    addiu $t3,$t1,10
    sw    $t3, -12($fp)
    ...
```

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

COMPILER USE OF OFFSETS

- Consider this C++ code:

```
void fcn(int a, int b) {
    ...
    int x = a - b;
    int y = b + 10;
    ...
}
```

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

```
fcn:
    ...
    lw    $t0, 0($fp)
    lw    $t1, -4($fp)
    subu  $t2,$t0,$t1
    sw    $t2, -8($fp)
    addiu $t3,$t1,10
    sw    $t3, -12($fp)
    ...
```

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

COMPILER USE OF OFFSETS

- Consider this C++ code:

```
void fcn(int a, int b) {
    ...
    int x = a - b;
    int y = b + 10;
    ...
}
```

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

```
fcn:
    ...
    lw    $t0, 0($fp)
    lw    $t1, -4($fp)
    subu  $t2,$t0,$t1
    sw    $t2, -8($fp)
    addiu $t3,$t1,10
    sw    $t3, -12($fp)
    ...
```

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

COMPILER USE OF OFFSETS

- Consider this C++ code:

```
void fcn(int a, int b) {
    ...
    int x = a - b;
    int y = b + 10;
    ...
}
```

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

```
fcn:
        ...
        lw     $t0, 0($fp)
        lw     $t1, -4($fp)
        subu   $t2,$t0,$t1
        sw     $t2, -8($fp)
        addiu  $t3,$t1,10
        sw     $t3, -12($fp)
        ...
```

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:

```
struct coord {
    int x;
    int y;
    int z;
};
```

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

```
coord* p1;
coord* p2;
...
p2->x = 17;
p2->y = p1->y;
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.

```
coord* p1;
coord* p2;
...
p2->x = 17;
p2->y = p1->y;
p2->z++;
```

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

```
li    $t1,17
sw    $t1,0($s2)

lw    $t2,4($s1)
sw    $t2,4($s2)

lw    $t3,8($s2)
addiu $t3,$t3,1
sw    $t3,8($s2)
```

OFFSETS FOR ACCESSING STRUCT COMPONENTS

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

```
coord* p1;
coord* p2;
...
p2->x = 17;
p2->y = p1->y;
p2->z++;
```

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

```
li    $t1,17
sw    $t1,0($s2)

lw    $t2,4($s1)
sw    $t2,4($s2)

lw    $t3,8($s2)
addiu $t3,$t3,1
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.

```
coord* p1;
coord* p2;
...
p2->x = 17;
p2->y = p1->y;
p2->z++;
```

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

```
li    $t1,17
sw    $t1,0($s2)

lw    $t2,4($s1)
sw    $t2,4($s2)

lw    $t3,8($s2)
addiu $t3,$t3,1
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.

```
coord* p1;
coord* p2;
...
p2->x = 17;
p2->y = p1->y;
p2->z++;
```

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

```
li    $t1,17
sw    $t1,0($s2)

lw    $t2,4($s1)
sw    $t2,4($s2)

lw    $t3,8($s2)
addiu $t3,$t3,1
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:

```
struct node {
    int data;
    struct node* next;
};
```

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

```
node nodes[3];
node* n1 = &nodes[0];
node* n2 = &nodes[1];
node* n3 = &nodes[2];

n1->data = 32;
n2->data = 57;
n3->data = 11;

n1->next = n2;
n2->next = n3;
n3->next = nullptr
```

LINKED LIST CODE CONVERTED TO MIPS

```
node nodes[3];
node* n1 = &nodes[0];
node* n2 = &nodes[1];
node* n3 = &nodes[2];
n1->data = 32;
n2->data = 57;
n3->data = 11;
n1->next = n2;
n2->next = n3;
n3->next = nullptr
```

```
        .data
nodes: space 24
        .text
          ...
        la     $s1,nodes
        addiu  $s2,$s1,8
        addiu  $s3,$s1,16
        li     $t0,32
        sw     $t0,($s1)
        li     $t0,57
        sw     $t0,($s2)
        li     $t0,11
        sw     $t0,($s3)
        sw     $s2,4($s1)
        sw     $s3,4($s2)
        li     $t0,0
        sw     $t0,4($s3)
```

LINKED LIST CODE CONVERTED TO MIPS

```
n1->data = 32;

n2->data = 57;

n3->data = 11;


n1->next = n2;
n2->next = n3;

n3->next = nullptr
```

```
li    $t0,32
sw    $t0,0($s1)
li    $t0,57
sw    $t0,0($s2)
li    $t0,11
sw    $t0,0($s3)

sw    $s2,4($s1)
sw    $s3,4($s2)
li    $t0,0
sw    $t0,4($s3)
```

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

TRAVERSING A LINKED LIST

- MIPS code that outputs a linked list

```
print:
    move    $s1, $s0          # current = first;
print_loop:
    beqz    $s1, done         # if current==nullptr goto done;
print_data:
    lw      $a0, ($s1)        # print(current->data);
    li      $v0, 1
    syscall
    lw      $s1, 4($s1)       # current = current->next;
    b       print_loop
done:
```

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

HOMEWORK 08

- Go to the syllabus page and accept the Git Classroom assignment.
- It is due next Wednesday, April 8th, at 3pm PST.
- Exercises: MIPS programs
  - string manipulation
  - integer and bit operations
  - array manipulation
  - linked list traversal
- We're holding "lab help sessions" 1:30-4:30pm PST tomorrow.
  - ➡ I'll send you some Zoom meeting links with instructions.