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.

```
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)
```

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.

```
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)
```

▸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.

```
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
```

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
```

▸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.

```
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
```

▸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.

```
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
```

▸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.      addu  $t0,$t0,$t2
6.      addiu $s0,$s0,4
7.      addiu $t1,$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.        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.

```
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:
```

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:
```

▸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.

```
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:
```

▸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:

```
1.        la $s0,string_ptr
2.  loop:
3.        lb $t0,($s0)
4.        beqz $t0,done
5.
6.        move $a0,$s0          # Output the string at the pointer.
7.        li $v0,4
8.        syscall
9.
10.       addiu $s0,$s0,1       # Advance the pointer by 1 byte.
11.       b loop
12. done:
```

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.      addiu $s2,$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]`.

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

```
1.  la $t4,hello_ptr
2.  lb $t3,0($t4) # save the 'h'
3.  lb $t6,1($t4)
4.  sb $t6,0($t4) # move the 'e' left
5.  lb $t6,2($t4)
6.  sb $t6,1($t4) # move the 'l' left
7.  lb $t6,3($t4)
8.  sb $t6,2($t4) # move the 'l' left
9.  lb $t6,4($t4)
10. sb $t6,3($t4) # move the 'o' left
11. 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:

```
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.          subu  $t2,$t0,$t1
6.          sw    $t2, -8($fp)
7.          addiu $t3,$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.           subu  $t2,$t0,$t1
6.           sw    $t2, -8($fp)
7.           addiu $t3,$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.            subu  $t2,$t0,$t1
6.            sw    $t2, -8($fp)
7.            addiu $t3,$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.          subu  $t2,$t0,$t1
6.          sw    $t2, -8($fp)
7.          addiu $t3,$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.          subu  $t2,$t0,$t1
6.          sw    $t2, -8($fp)
7.          addiu $t3,$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.      addiu $t3,$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.      addiu $t3,$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.      addiu $t3,$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.      addiu $t3,$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.        addiu $s2,$s1,8
7.        addiu $s3,$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;                1.      li    $t0,32
2.                                2.      sw    $t0,0($s1)
3.  n2->data = 57;               3.      li    $t0,57
4.                                4.      sw    $t0,0($s2)
5.  n3->data = 11;               5.      li    $t0,11
6.                                6.      sw    $t0,0($s3)
7.
8.  n1->next = n2;               8.      sw    $s2,4($s1)
9.  n2->next = n3;               9.      sw    $s3,4($s2)
10.                              10.      li    $t0,0
11. n3->next = nullptr           11.      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

```
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.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 access and manipulation

- array access

- linked list traversal with access

▸We're holding "lab help sessions" 1:30-4:30pm PST tomorrow.

- ➡ I'll send you some Zoom meeting links with instructions.