Flow Control in Assembly Language

CS 64: Computer Organization and Design Logic
Lecture #5
Fall 2018

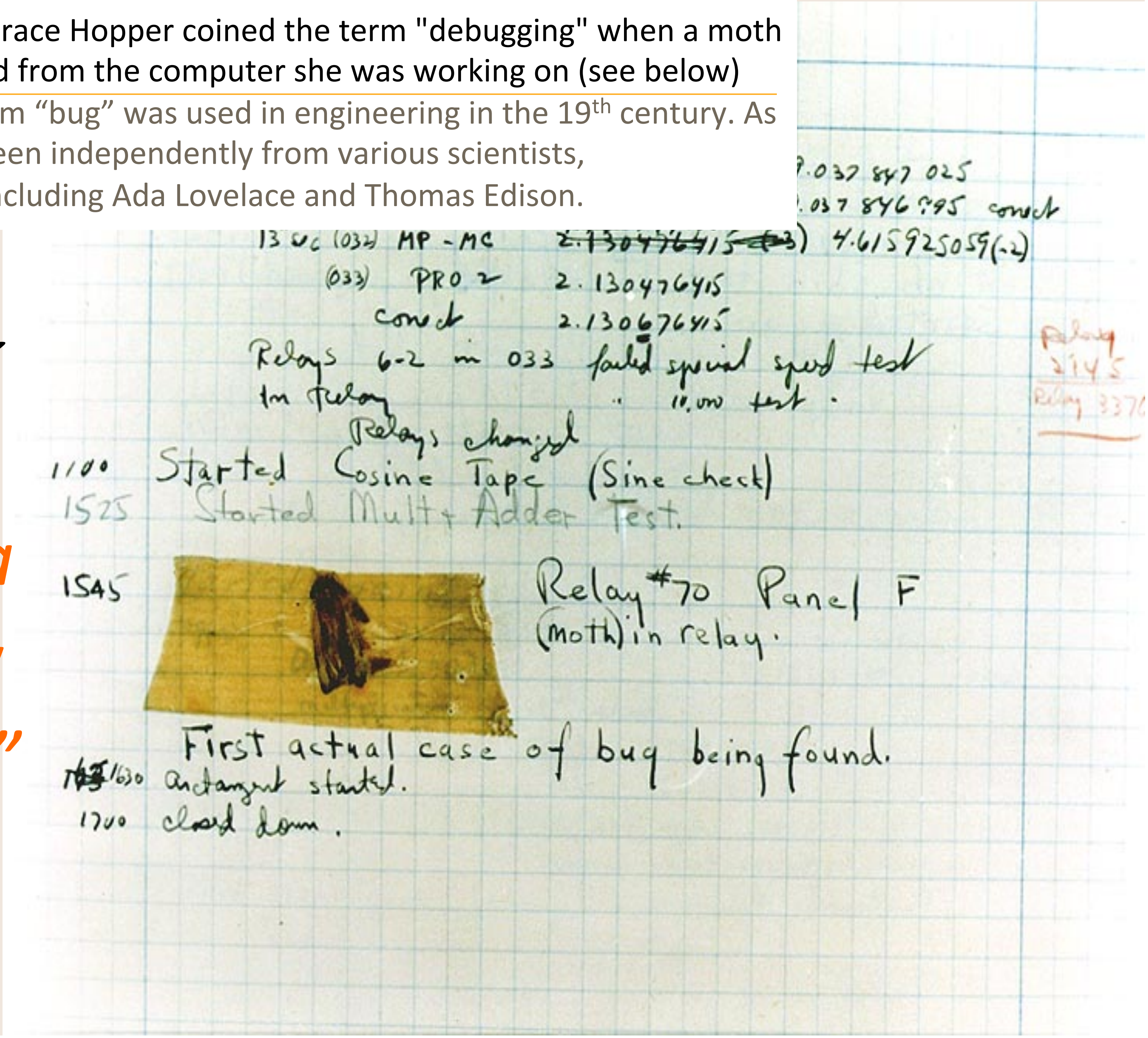
Ziad Matni, Ph.D.
Dept. of Computer Science, UCSB

Legend: Adm. Grace Hopper coined the term "debugging" when a moth was removed from the computer she was working on (see below)

Reality: The term “bug” was used in engineering in the 19th century. As seen independently from various scientists, including Ada Lovelace and Thomas Edison.

This Week on “Didja Know Dat?!”

10/17/18



.037 847 025
.037 846 995 correct
13 uc (033) MP - MC ~~2.130476415 (-3)~~ 4.615925059(-2)
(033) PRO 2 2.130476415
correct 2.130676415
Relays 6-2 in 033 failed special speed test
In relay 11.000 test.
Relays changed
1100 Started Cosine Tape (Sine check)
1525 Started Mult + Adder Test.
1545 Relay #70 Panel F
(moth) in relay.
First actual case of bug being found.
1630 Antangent started.
1700 closed down.

Relay 2145
Relay 337

Administrative

- Reminder that your midterm exam is on **October 31st**
  - Same time/place as regular lecture
  - DSP students: make arrangements ASAP

- Lab 1 is graded

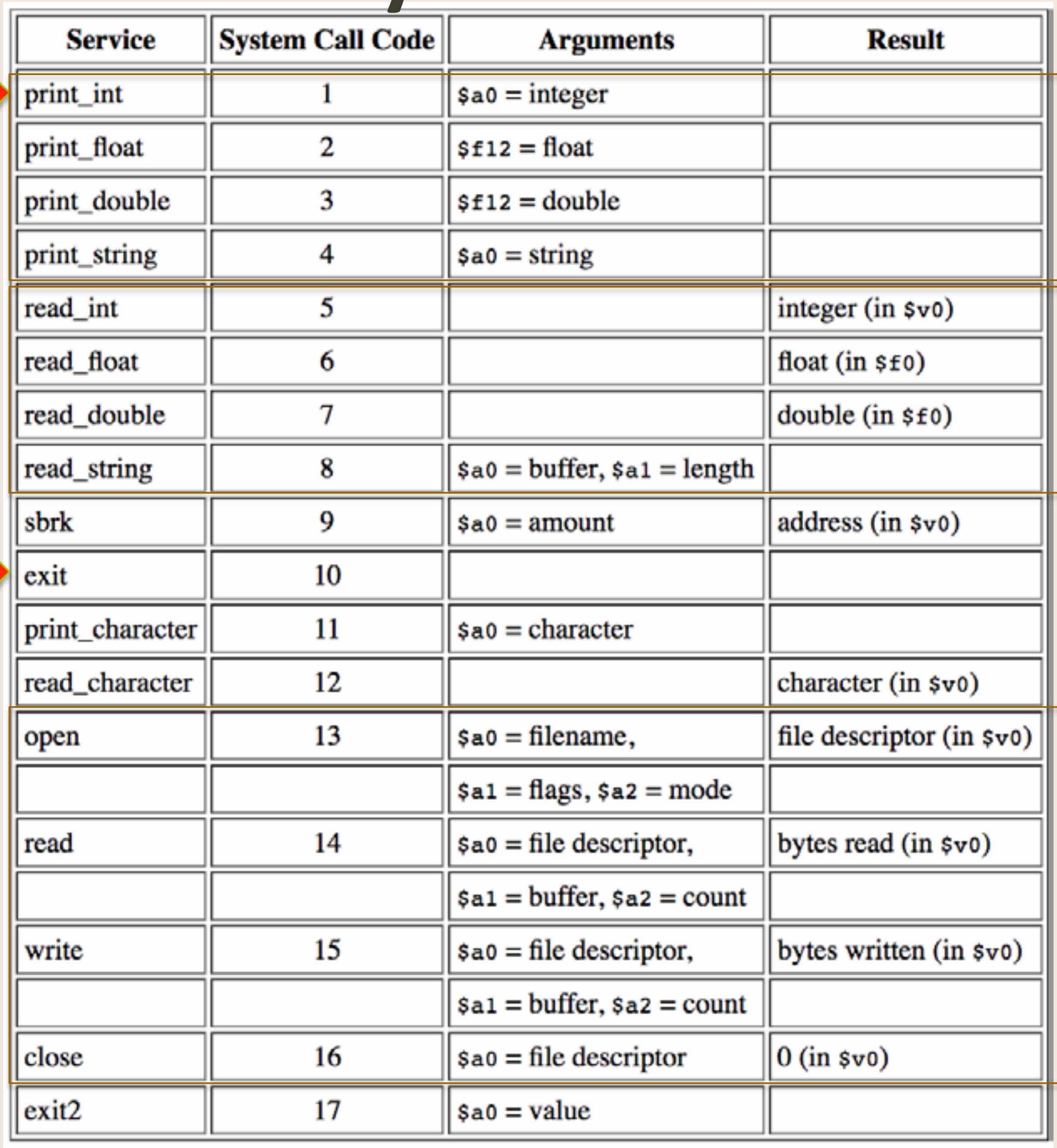
Lecture Outline

- More on instructions in MIPS
- Operand Use
- **.data** Directives and Basic Memory Use
- Flow Control in Assembly
  - With Demos!

Any Questions From Last Lecture?

MIPS System Services

Examples We've Seen So Far…



Service	System Call Code	Arguments	Result
print_int	1	`$a0` = integer	
print_float	2	`$f12` = float	
print_double	3	`$f12` = double	
print_string	4	`$a0` = string	
read_int	5		integer (in `$v0`)
read_float	6		float (in `$f0`)
read_double	7		double (in `$f0`)
read_string	8	`$a0` = buffer, `$a1` = length	
sbrk	9	`$a0` = amount	address (in `$v0`)
exit	10		
print_character	11	`$a0` = character	
read_character	12		character (in `$v0`)
open	13	`$a0` = filename,	file descriptor (in `$v0`)
		`$a1` = flags, `$a2` = mode	
read	14	`$a0` = file descriptor,	bytes read (in `$v0`)
		`$a1` = buffer, `$a2` = count	
write	15	`$a0` = file descriptor,	bytes written (in `$v0`)
		`$a1` = buffer, `$a2` = count	
close	16	`$a0` = file descriptor	0 (in `$v0`)
exit2	17	`$a0` = value	

stdout

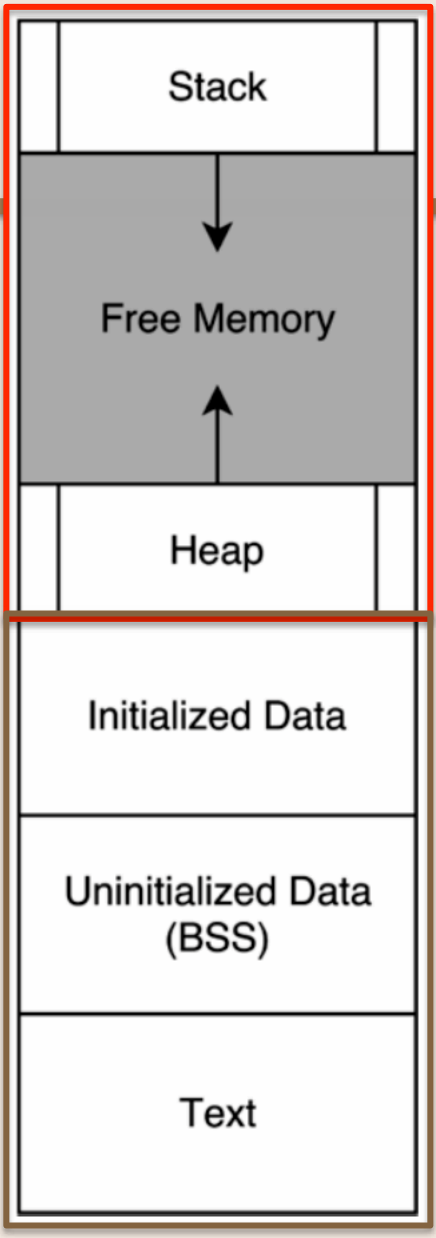
stdin

File I/O

Now Let’s Make it a Full Program (almost)

- We need to tell the assembler (and its simulator) **which bits** should be placed **where** in memory
  - Bits?

(remember: everything ends up being a bunch of 1’s and 0’s !)



Allocated as program RUN

Constants to be used in the program (like strings)

Allocated at program LOAD

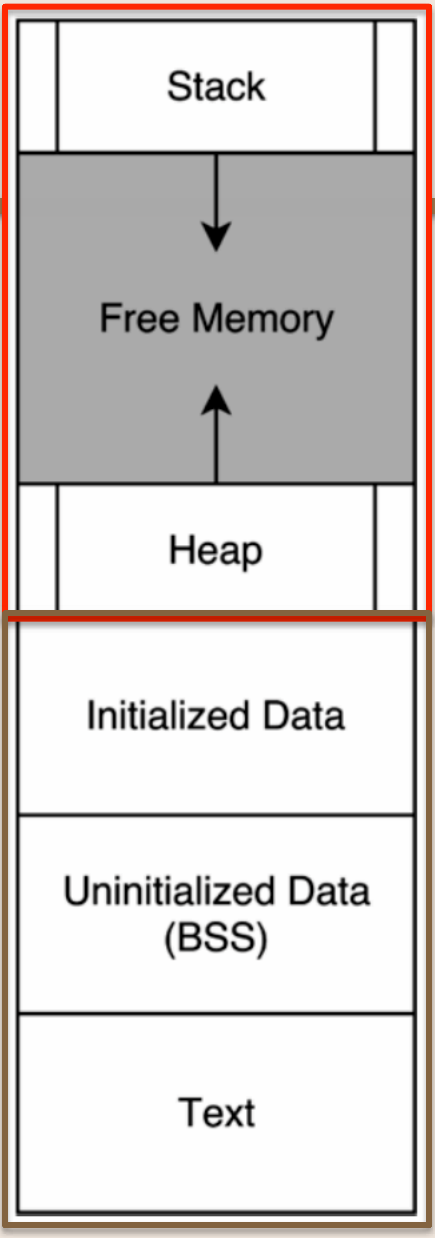
mutable global variables

the text of the program

Marking the Code

- For the simulator, you'll need a **.text** directive to specify code

```
.text

# Main program
li $t0, 5
li $t1, 7
add $t3, $t0, $t1

# Print to standard output
li $v0, 1
move $a0, $t3
syscall

# End program
li $v0, 10
syscall
```



Allocated as program RUN

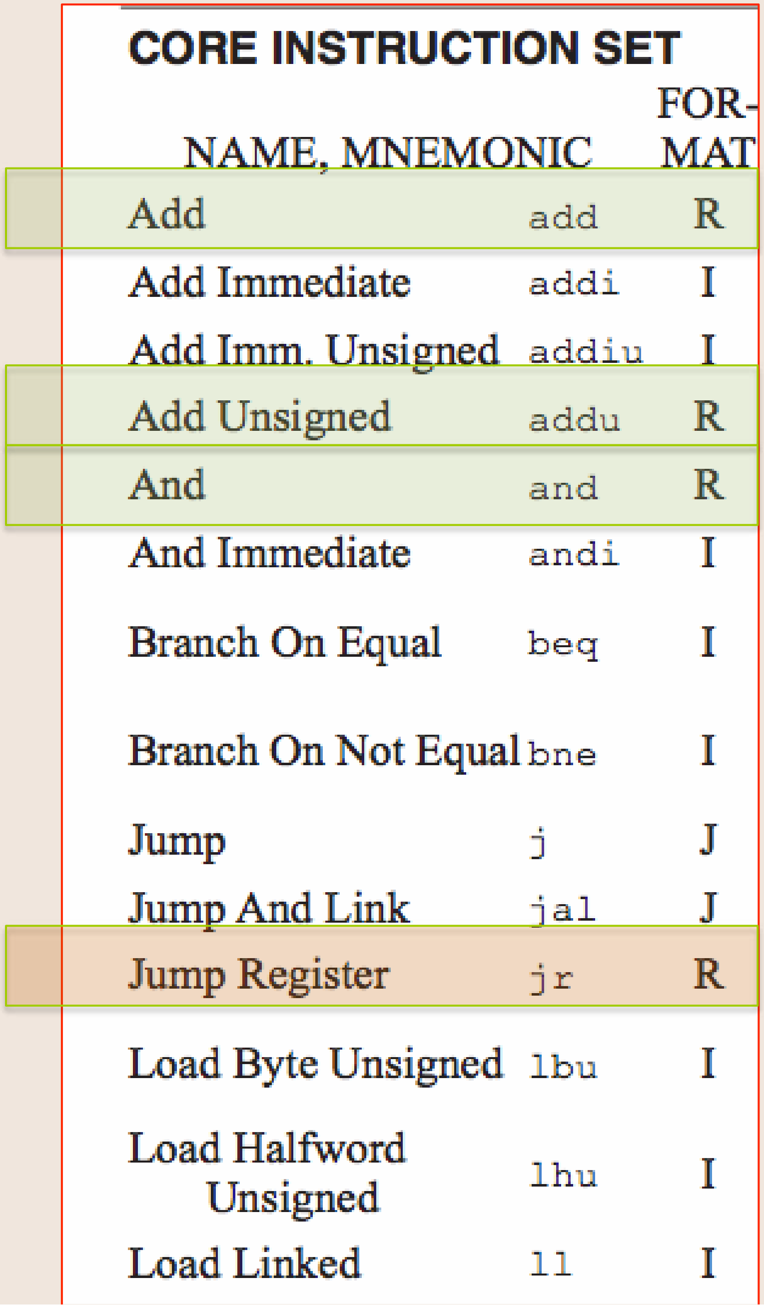
Constants to be used in the program (like strings)

Allocated at program LOAD

mutable global variables

the text of the program

List of all Core Instructions in MIPS



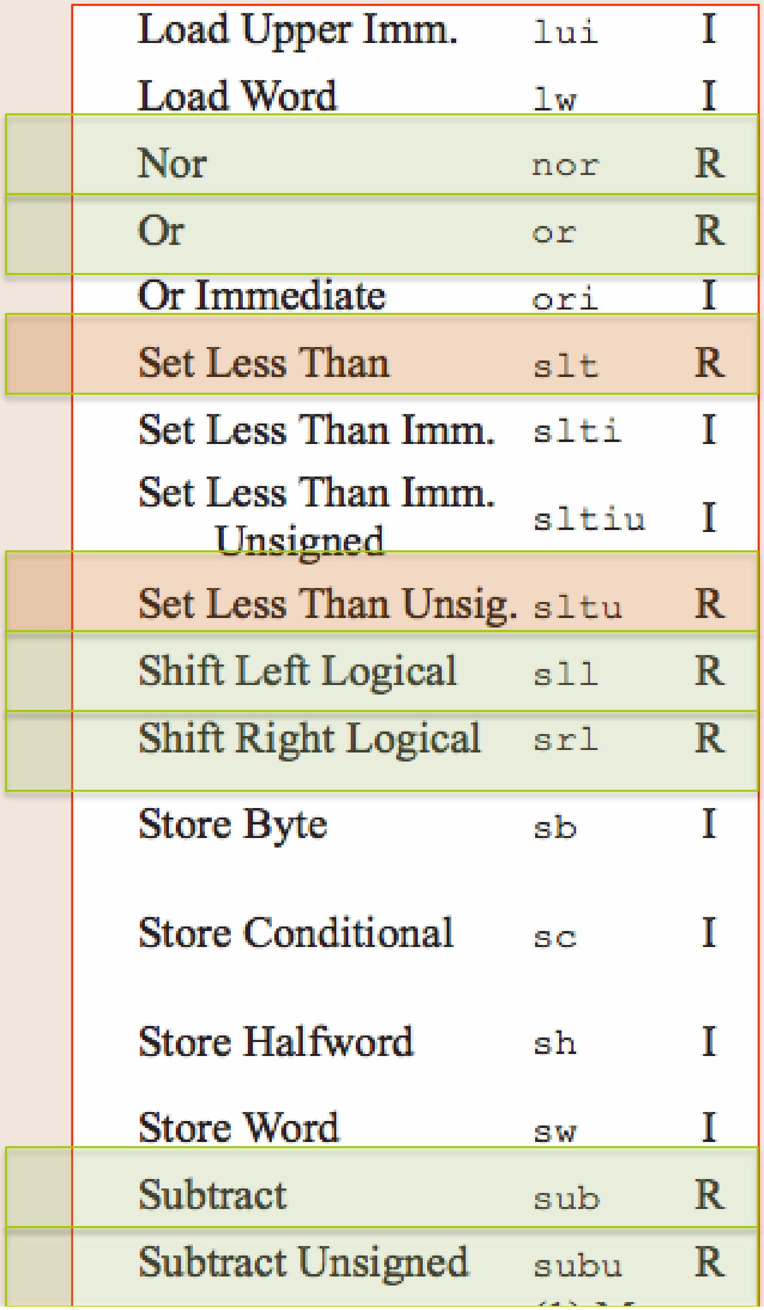
CORE INSTRUCTION SET

NAME, MNEMONIC		FOR-MAT
Add	`add`	R
Add Immediate	`addi`	I
Add Imm. Unsigned	`addiu`	I
Add Unsigned	`addu`	R
And	`and`	R
And Immediate	`andi`	I
Branch On Equal	`beq`	I
Branch On Not Equal	`bne`	I
Jump	`j`	J
Jump And Link	`jal`	J
Jump Register	`jr`	R
Load Byte Unsigned	`lbu`	I
Load Halfword Unsigned	`lhu`	I
Load Linked	`ll`	I

“R”

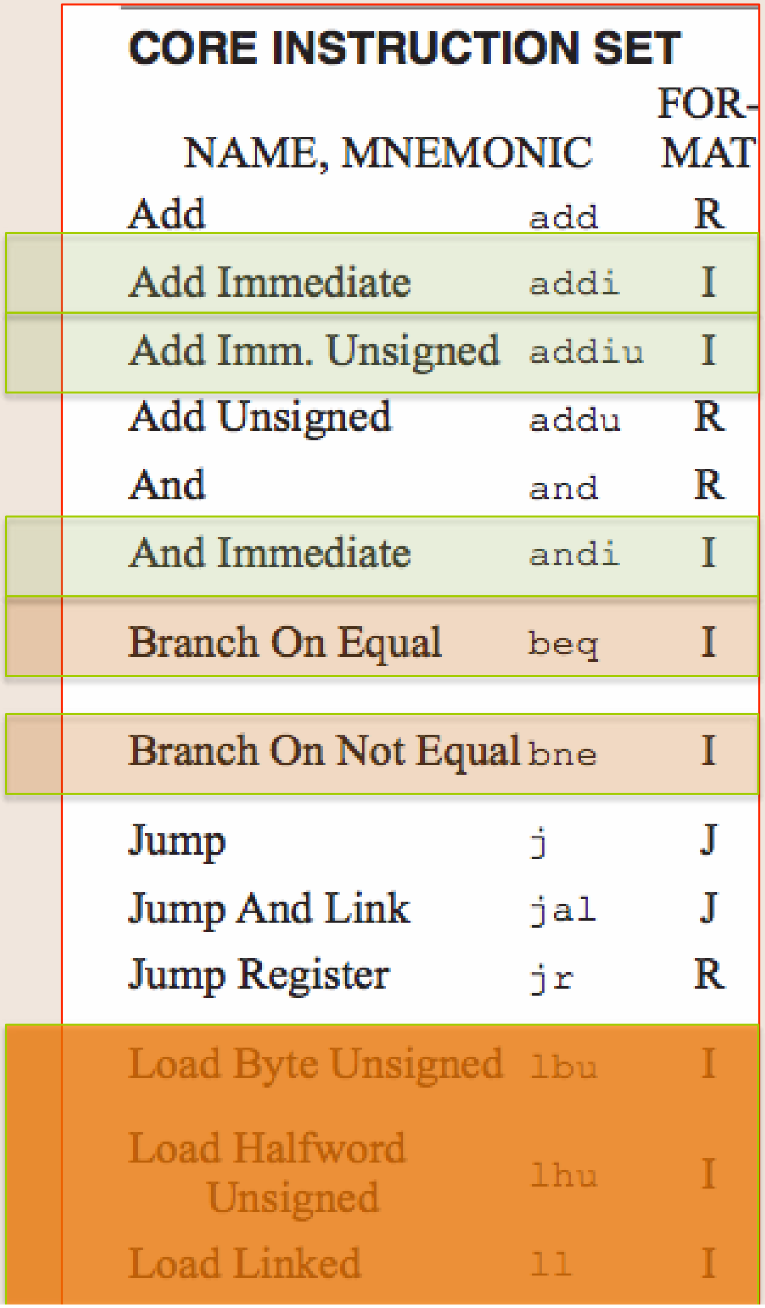
Arithmetic

Branching



Load Upper Imm.	`lui`	I
Load Word	`lw`	I
Nor	`nor`	R
Or	`or`	R
Or Immediate	`ori`	I
Set Less Than	`slt`	R
Set Less Than Imm.	`slti`	I
Set Less Than Imm. Unsigned	`sltiu`	I
Set Less Than Unsig.	`sltu`	R
Shift Left Logical	`sll`	R
Shift Right Logical	`srl`	R
Store Byte	`sb`	I
Store Conditional	`sc`	I
Store Halfword	`sh`	I
Store Word	`sw`	I
Subtract	`sub`	R
Subtract Unsigned	`subu`	R

List of all Core Instructions in MIPS



CORE INSTRUCTION SET

NAME, MNEMONIC		FORMAT
Add	add	R
Add Immediate	addi	I
Add Imm. Unsigned	addiu	I
Add Unsigned	addu	R
And	and	R
And Immediate	andi	I
Branch On Equal	beq	I
Branch On Not Equal	bne	I
Jump	j	J
Jump And Link	jal	J
Jump Register	jr	R
Load Byte Unsigned	lbu	I
Load Halfword Unsigned	lhu	I
Load Linked	ll	I

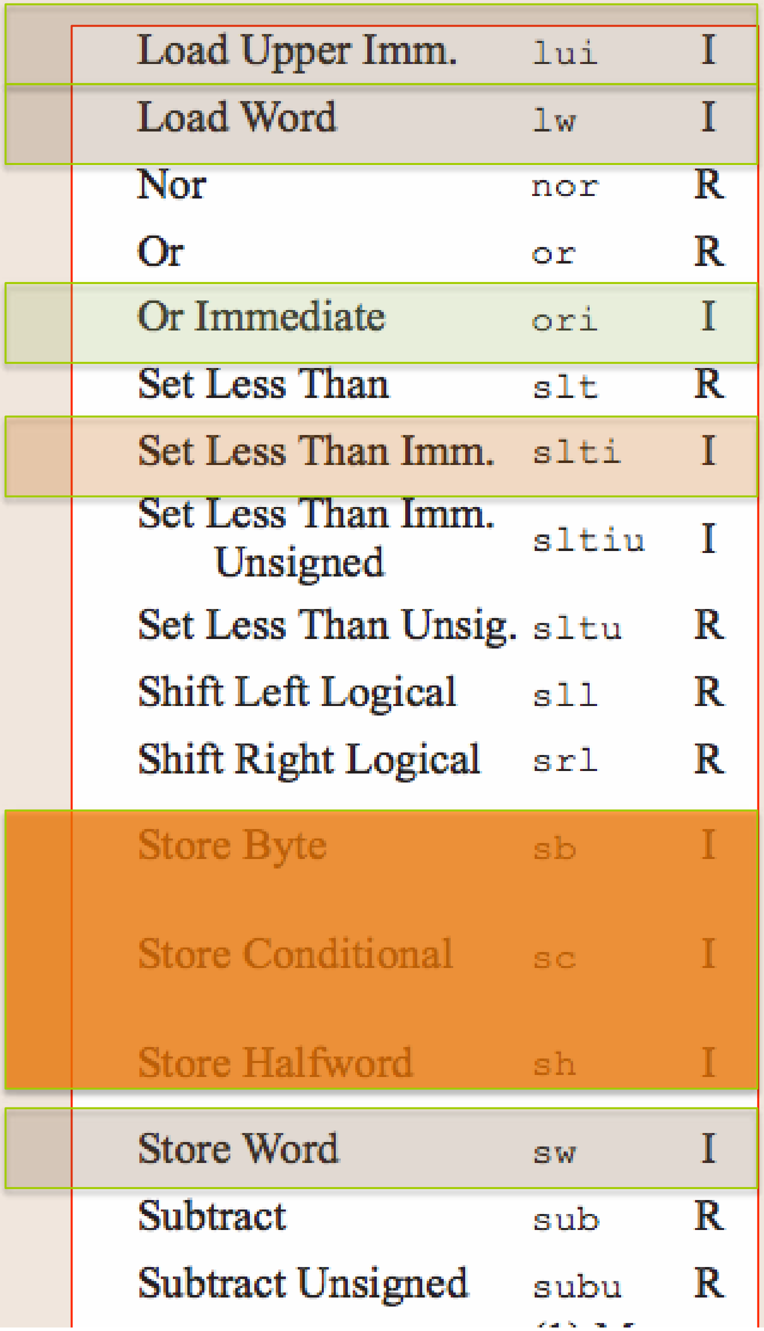
"I"

Arithmetic

Branching

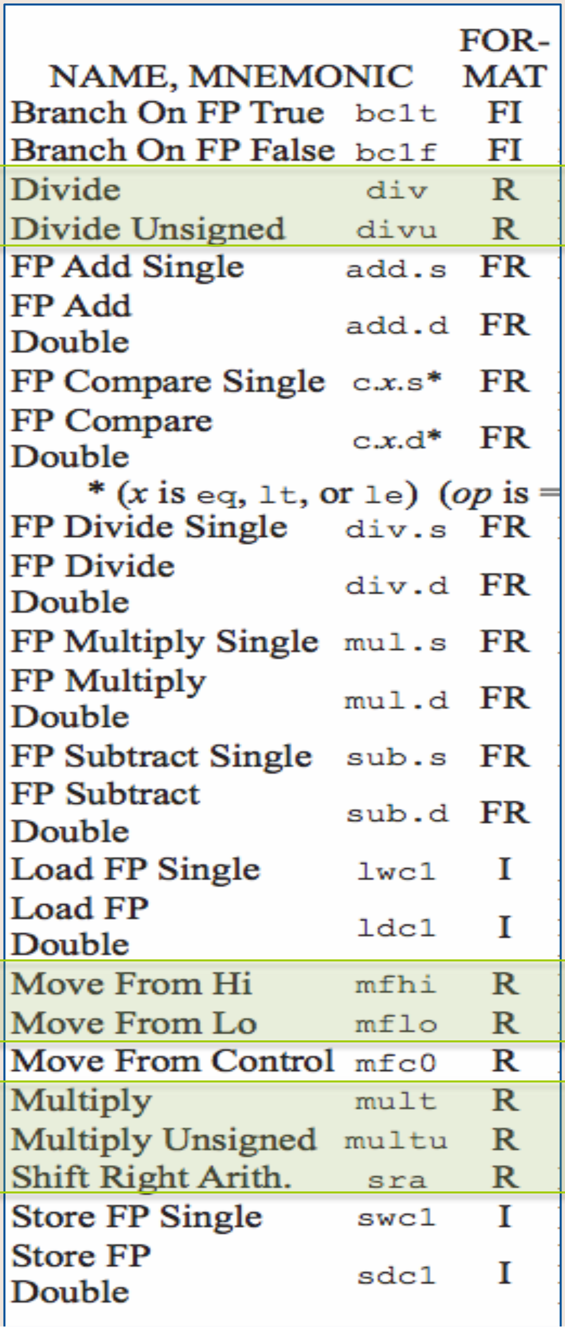
Memory

Not for CS64



Load Upper Imm.	lui	I
Load Word	lw	I
Nor	nor	R
Or	or	R
Or Immediate	ori	I
Set Less Than	slt	R
Set Less Than Imm.	slti	I
Set Less Than Imm. Unsigned	sltiu	I
Set Less Than Unsig.	sltu	R
Shift Left Logical	sll	R
Shift Right Logical	srl	R
Store Byte	sb	I
Store Conditional	sc	I
Store Halfword	sh	I
Store Word	sw	I
Subtract	sub	R
Subtract Unsigned	subu	R

List of the Arithmetic Core Instructions in MIPS



NAME, MNEMONIC		FOR-MAT
Branch On FP True	`bc1t`	FI
Branch On FP False	`bc1f`	FI
Divide	`div`	R
Divide Unsigned	`divu`	R
FP Add Single	`add.s`	FR
FP Add Double	`add.d`	FR
FP Compare Single	`c.x.s*`	FR
FP Compare Double	`c.x.d*`	FR
* (*x* is `eq`, `lt`, or `le`) (*op* is =		
FP Divide Single	`div.s`	FR
FP Divide Double	`div.d`	FR
FP Multiply Single	`mul.s`	FR
FP Multiply Double	`mul.d`	FR
FP Subtract Single	`sub.s`	FR
FP Subtract Double	`sub.d`	FR
Load FP Single	`lwc1`	I
Load FP Double	`ldc1`	I
Move From Hi	`mfhi`	R
Move From Lo	`mflo`	R
Move From Control	`mfc0`	R
Multiply	`mult`	R
Multiply Unsigned	`multu`	R
Shift Right Arith.	`sra`	R
Store FP Single	`swc1`	I
Store FP Double	`sdc1`	I

Mostly used in CS64

The **move** instruction

- The move instruction does not actually show up in SPIM
- It is a ***pseudo-instruction***
- It's easy for us to use, but it's actually a "macro" of another actual instruction

ORIGINAL: `move $a0, $t3`

ACTUAL: `addu $a0, $zero, $t3`

```
# what's addu? what's $zero?
```

Why Pseudocodes?

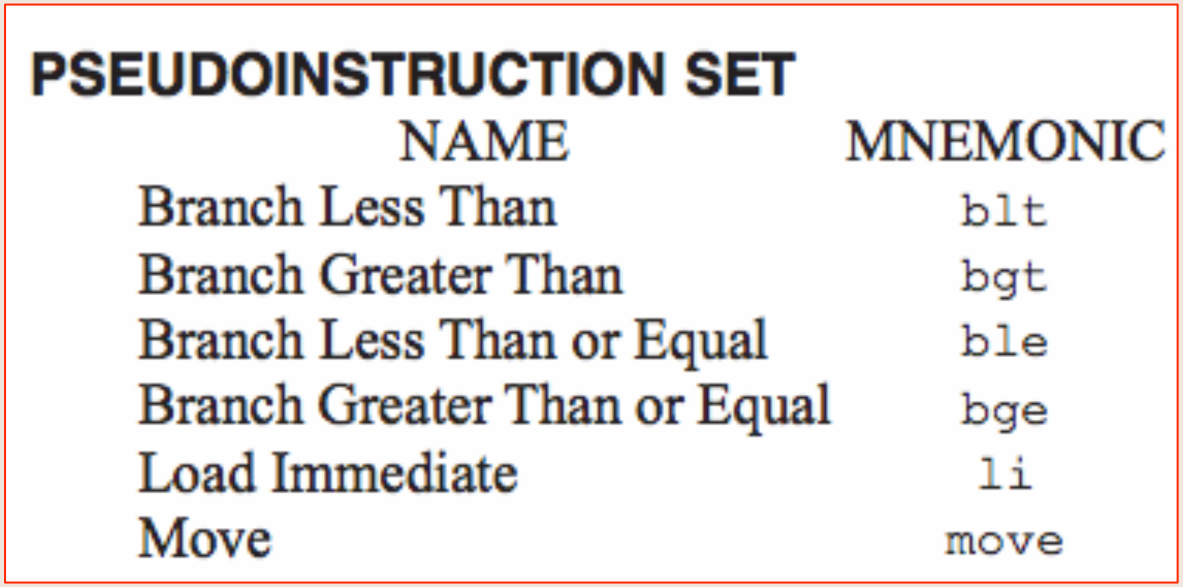
And what's this `$zero`??

- **$zero**
  - Specified like a normal register, but does not behave like a normal register
  - Writes to $zero are not saved
  - Reads from $zero always return 0 value

- Why have **`move`** as a **pseudo-instruction** instead of as an actual instruction?
  - It's one less instruction to worry about
  - One design goal of RISC is to cut out redundancy
  - **`move`** isn't the only one! **`li`** is another one too!

List of all PsuedoInstructions in MIPS

That You Are Allowed to Use in CS64!!!



PSEUDOINSTRUCTION SET

NAME	MNEMONIC
Branch Less Than	`blt`
Branch Greater Than	`bgt`
Branch Less Than or Equal	`ble`
Branch Greater Than or Equal	`bge`
Load Immediate	`li`
Move	`move`

plus this one → Load Address la

ALL OF THIS AND MORE IS ON YOUR HANDY "MIPS REFERENCE CARD" FOUND ON THE CLASS WEBSITE

A Note About Operands

- Operands in arithmetic instructions are limited and are done in a certain order
  - Arithmetic operations always happen in the registers
- Example: *f = (g + h) – (i + j)*
  - The order is prescribed by the parentheses
  - Let's say, **f**, **g**, **h**, **i**, **j** are assigned to registers **$s0**, **$s1**, **$s2**, **$s3**, **$s4** respectively
  - What would the MIPS assembly code look like?

Example 1

Syntax for "add"

`add rd, rs, rt`

destination, source1, source2

$f = (g + h) - (i + j)$

i.e. $\$s0 = \underbrace{\underbrace{(\$s1 + \$s2)}_{} - \underbrace{(\$s3 + \$s4)}_{}}_{}$

```
add $t0, $s1, $s2
add $t1, $s3, $s4
sub $s0, $t0, $t1
```

Example 2

$f = g * h - i$

$$\text{i.e. } \$s0 = \underbrace{\underbrace{(\$s1 * \$s2)}_{} - \$s3}_{}$$

```
mult $s1, $s2
mflo $t0
# mflo directs where the answer of the mult should go
sub $s0, $t0, $s3
```

What’s the Difference Between…

- **add** and **addu** and **addi** and **addiu**
  - **add** : add what’s in 2 registers & put result in another
  - **addu** : same as **add**, but only w/ *unsigned* numbers
  - **addi** : add an integer to what’s in a register & put result in another register
  - **addiu** : same as **addi**, but only w/ *unsigned* numbers

- Syntax:

```
add $rd, $rs, $rt                 (R-Type)
addu $rd, $rs, $rt                (R-Type)
addi $rd, $rs, immediate          (I-Type)
addiu $rd, $rs, immediate         (I-Type)
```

This is a 16-bit number (why not 32b????)

Global Variables, Arrays, and Strings

- Typically, global variables are placed directly in memory and **not** registers
  - Why might this be?
    - Ans: Not enough registers…
      esp. if there are multiple variables

- What do you think we do with arrays? Why?
- What do you think we do with strings? Why?

- We use the **.data** directive
  - To declare variables, their values, and their names used in the program
  - Storage is allocated in main memory (RAM)

.data Declaration Types

w/ Examples

```
var1:    .byte 9         # declare a single byte with value 9
var2:    .half 63        # declare a 16-bit half-word w/ val. 63
var3:    .word 9433      # declare a 32-bit word w/ val. 9433
num1:    .float 3.14     # declare 32-bit floating point number
num2:    .double 6.28    # declare 64-bit floating pointer number
str1:    .ascii "Text"   # declare a string of chars
str3:    .asciiz "Text"  # declare a null-terminated string
str2:    .space 5        # reserve 5 bytes of space (useful for arrays)
```

These are now reserved in memory and we can call them up by loading their memory address into the appropriate registers.

Highlighted ones are the ones most commonly used in this class.

`li` *vs* `la`

- li Load Immediate
  - Use this when you want to put an integer value into a register
  - Example: `li $t0, 42`
- la Load Address
  - Use this when you want to put an address value into a register
  - Example: `la $t0, myLittlePony`

    where "myLittlePony" is a pre-defined label for something in memory (defined under the .data directive).

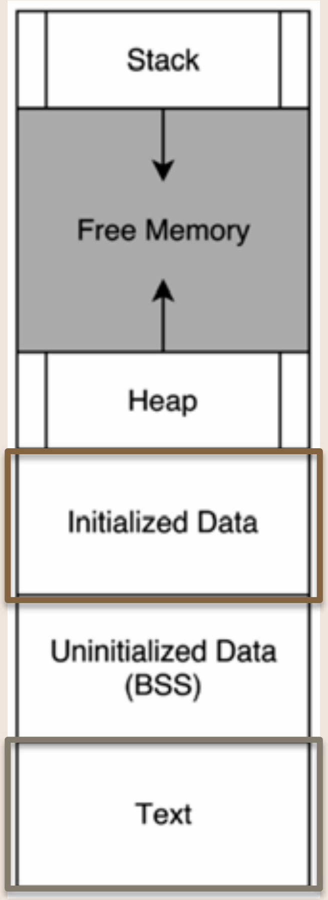
Example 3

What does this do?

```
.data
name: .asciiz “Jimbo Jones is ”
rtn: .asciiz “ years old.\n”

.text
main:
    li $v0, 4
    la $a0, name     # la = load memory address
    syscall

    li $v0, 1
    li $a0, 15
    syscall

    li $v0, 4
    la $a0, rtn
    syscall

    li $v0, 10
    syscall
```

What goes in here? →

What goes in here? →



Conditionals

- What if we wanted to do:

  `if (x == 0) { printf(“x is zero”); }`

  - Can we write this in assembly with what we know?
    - No… we haven’t covered **if-else** (aka *branching*)

- What do we need to implement this?
  - A way to *compare* numbers
  - A way to *conditionally execute* code

Relevant Instructions in MIPS

for use with branching conditionals

- Comparing numbers:

  set-less-than (`slt`)

  - Set some register (i.e. make it "1") if a less-than comparison of some other registers is true

- Conditional execution:

  branch-on-equal (beq)
  branch-on-not-equal (bne)

  - "Go to" some other place in the code (i.e. jump)

if (x == 0) { printf(“x is zero”); }

```
.data
   x_is_zero: .asciiz “x is zero”

.text
   bne $t0, $zero, after_print
   li $v0, 4
   la $a0, x_is_zero
   syscall

after_print:
   li $v0, 10
   syscall
```

Create a constant string called “**x_is_zero**”

If **$t0 != 0** go to the block labeled as “***after_print***”

(otherwise) prepare to print a string…

…and that string is inside of “x_is_zero”

End the program

Note the flow

Loops

- How might we translate the following C++ to assembly?

```
n = 3;
sum = 0;
while (n != 0)
{
    sum += n;
    n--;
}
cout << sum;
```

`n = 3; sum = 0;` `while (n != 0) { sum += n; n--; }`

```
.text
main:
    li $t0, 3    # n
    li $t1, 0    # running sum
loop:
    beq $t0, $zero, loop_exit
    addu $t1, $t1, $t0
    addi $t0, $t0, -1
    j loop

loop_exit:
    li $v0, 1
    move $a0, $t1
    syscall

    li $v0, 10
    syscall
```

Set up the variables in $t0, $t1

If **$t0 == 0** go to “loop_exit”

(otherwise) make $t1 the (unsigned) sum of $t1 and $t0 (i.e. **sum += n**)

decrement $t0 (i.e. **n--**)

jump to the code labeled “loop” (i.e. **repeat loop**)

prepare to print out an integer, which is inside the $t1 reg. (i.e. **print sum**)

end the program

Let’s Run More Programs!!

Using SPIM

- More!!
- This time exploring conditional logic and loops

DEMO !!!

These assembly code programs are made available to you via the class webpage

YOUR TO-DOs

- Review ALL the demo code
  - Available via the class website

- Assignment #3
  - Due Friday

</LECTURE>