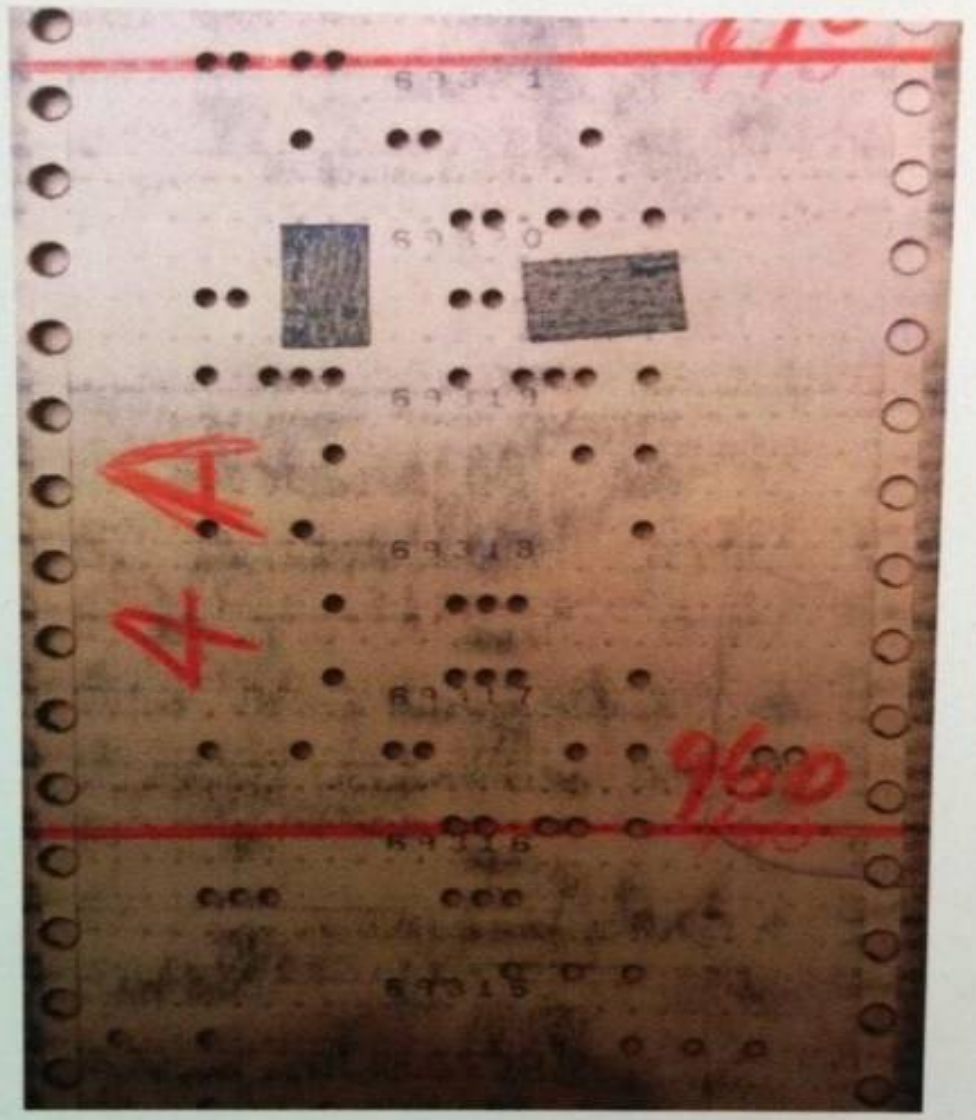
MIPS and Basic Assembly Language

CS 64: Computer Organization and Design Logic
Lecture #4

Ziad Matni
Dept. of Computer Science, UCSB

This Week on "Didja Know Dat?!"

The "Patch"



Small corrections to the programmed sequence could be done by patching over portions of the paper tape and re-punching the holes in that section.

Image courtesy of the Smithsonian Archives Center.

Why do CPU programmers celebrate Christmas and Halloween on the same day?

Because Oct-31 = Dec-25

Lecture Outline

- MIPS core processing blocks
- Basic programming in assembly
- Arithmetic programs
- Control logic programs

Any Questions From Last Lecture?

MIPS

Why MIPS?

- MIPS:
  - a **r**educed **i**nstruction **s**et **c**omputer (RISC) architecture developed by MIPS Technologies (1981)
- Relevant in the *embedded systems* domain
  - Computer systems with a dedicated function within a larger mechanical or electrical system
  - Example: low power and low cost processors in MP3 players or "Smart" Appliances
- All modern commercial processors share the same core concepts as MIPS, just with extra stuff... but more importantly...

MIPS is Simpler…

… than other instruction sets for CPUs

So it's a great learning tool

- Dozens of instructions (as opposed to hundreds)
- Lack of redundant instructions or special cases
- 5 stage pipeline versus 24 stages

Note: Pipelining in CPUs

- Pipelining is a fundamental design in CPUs
- Allows multiple instructions to go on at once
  - a.k.a instruction-level parallelism

Basic five-stage pipeline

Instr. No. \ Clock cycle	1	2	3	4	5	6	7
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX

(IF = Instruction Fetch, ID = Instruction Decode, EX = Execute, MEM = Memory access, WB = Register write back).

Code on MIPS

Original

```
x = 5;
y = 7;
z = x + y;
```

MIPS

```
li $t0, 5
li $t1, 7
add $t3, $t0, $t1
```

Code on MIPS

Original

```
x = 5;
y = 7;
z = x + y;
```

MIPS

```
li $t0, 5
li $t1, 7
add $t3, $t0, $t1
```

load immediate: put the given value into a register

$t0: temporary register **0**

Code on MIPS

Original

```
x = 5;
y = 7;
z = x + y;
```

MIPS

```
li $t0, 5
li $t1, 7
add $t3, $t0, $t1
```

load immediate: put the given value into a register

$t1: temporary register **1**

Code on MIPS

Original

```
x = 5;
y = 7;
z = x + y;
```

MIPS

```
li $t0, 5
li $t1, 7
add $t3, $t0, $t1
```

add: add the rightmost registers, putting the result in the first register

$t3: temporary register **3**

Available Registers in MIPS

- 32 registers in all
  - Refer to your MIPS Reference Card
- For the moment, let's only consider registers **$t0 thru $t9**

NAME	NUMBER	USE
$zero	0	The Constant Value 0
$at	1	Assembler Temporary
$v0-$v1	2-3	Values for Function Results and Expression Evaluation
$a0-$a3	4-7	Arguments
$t0-$t7	8-15	Temporaries
$s0-$s7	16-23	Saved Temporaries
$t8-$t9	24-25	Temporaries
$k0-$k1	26-27	Reserved for OS Kernel
$gp	28	Global Pointer
$sp	29	Stack Pointer
$fp	30	Frame Pointer
$ra	31	Return Address

Assembly

- The code that you see is MIPS assembly

```
li $t0, 5
li $t1, 7
add $t3, $t0, $t1
```

- Assembly is *almost* what the machine sees. For the most part, it is a **direct** translation to binary from here (known as machine code)

- An **assembler** takes assembly code and changes it into the actual 1's and 0's for machine code
  - Analogous to a compiler for HL code

Machine Code/Language

- What a CPU actually accepts as input
- What actually gets executed

- Each instruction is represented with **32 bits**

- There are **three** different instruction ***formats***: **R**, **I**, and **J**
  - These allow for instructions to take on different roles
  - R-Format is used when it's all about registers
  - I-Format is used when you involve numbers
  - J-Format is used when you do code "jumping" (i.e. branching)

Instruction Register

?

Registers

$t0: ?
$t1: ?
$t2: ?

Since all instructions are 32-bits, then they each occupy 4 Bytes of memory.

Memory is addressed in Bytes (more on this later).

Program Counter

?

Memory

?

Arithmetic Logic Unit

?

Instruction Register

```
?
```

Registers

```
$t0: ?
$t1: ?
$t2: ?
```

Since all instructions are 32-bits, then they each occupy 4 Bytes of memory.

Memory is addressed in Bytes (more on this later).

Program Counter

```
0
```

Memory

```
0: li $t0, 5
4: li $t1, 7
8: add $t3, $t0, $t1
```

Arithmetic Logic Unit

```
?
```

Instruction Register

```
li $t0, 5
```

Registers

```
$t0: ?
$t1: ?
$t2: ?
```

Since all instructions are 32-bits, then they each occupy 4 Bytes of memory.

Memory is addressed in Bytes (more on this later).

Program Counter

```
0
```

Memory

```
0: li $t0, 5
4: li $t1, 7
8: add $t3, $t0, $t1
```

Arithmetic Logic Unit

```
?
```

Instruction Register

```
li $t0, 5
```

Registers

```
$t0: 5
$t1: ?
$t2: ?
```

Since all instructions are 32-bits, then they each occupy 4 Bytes of memory.

Memory is addressed in Bytes (more on this later).

Program Counter

```
0
```

Memory

```
0: li $t0, 5
4: li $t1, 7
8: add $t3, $t0, $t1
```

Arithmetic Logic Unit

```
?
```

Instruction Register

```
li $t0, 5
```

Registers

```
$t0: 5
$t1: ?
$t2: ?
```

Since all instructions are 32-bits, then they each occupy 4 Bytes of memory.

Memory is addressed in Bytes (more on this later).

Program Counter

```
4
```

Memory

```
0: li $t0, 5
4: li $t1, 7
8: add $t3, $t0, $t1
```

Arithmetic Logic Unit

```
0 + 4 = 4
```

Instruction Register

```
li $t1, 7
```

Registers

```
$t0: 5
$t1: ?
$t2: ?
```

Since all instructions are 32-bits, then they each occupy 4 Bytes of memory.

Memory is addressed in Bytes (more on this later).

Program Counter

```
4
```

Memory

```
0: li $t0, 5
4: li $t1, 7
8: add $t3, $t0, $t1
```

Arithmetic Logic Unit

```
?
```

Instruction Register

```
li $t1, 7
```

Registers

```
$t0: 5
$t1: 7
$t2: ?
```

Since all instructions are 32-bits, then they each occupy 4 Bytes of memory.

Memory is addressed in Bytes (more on this later).

Program Counter

```
4
```

Memory

```
0: li $t0, 5
4: li $t1, 7
8: add $t3, $t0, $t1
```

Arithmetic Logic Unit

```
?
```

Instruction Register

```
li $t1, 7
```

Registers

```
$t0: 5
$t1: 7
$t2: ?
```

Since all instructions are 32-bits, then they each occupy 4 Bytes of memory.

Memory is addressed in Bytes (more on this later).

Program Counter

```
8
```

Memory

```
0: li $t0, 5
4: li $t1, 7
8: add $t3, $t0, $t1
```

Arithmetic Logic Unit

```
4 + 4 = 8
```

Instruction Register

```
add $t3, $t0, $t1
```

Registers

```
$t0: 5
$t1: 7
$t2: ?
```

Since all instructions are 32-bits, then they each occupy 4 Bytes of memory.

Memory is addressed in Bytes (more on this later).

Program Counter

```
8
```

Memory

```
0: li $t0, 5
4: li $t1, 7
8: add $t3, $t0, $t1
```

Arithmetic Logic Unit

```
?
```

Instruction Register

```
add $t3, $t0, $t1
```

Registers

```
$t0: 5
$t1: 7
$t2: ?
```

Since all instructions are 32-bits, then they each occupy 4 Bytes of memory.

Memory is addressed in Bytes (more on this later).

Program Counter

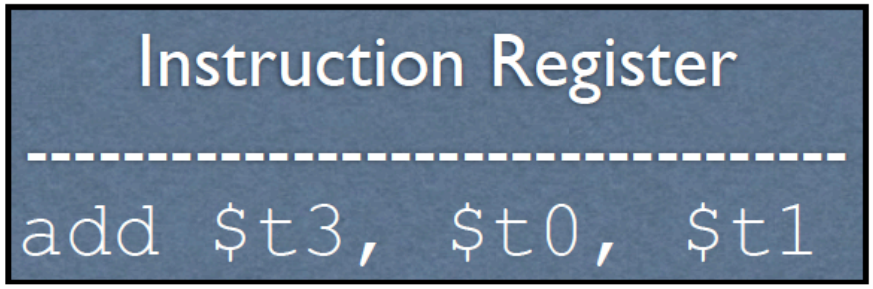
```
8
```

Memory

```
0: li $t0, 5
4: li $t1, 7
8: add $t3, $t0, $t1
```

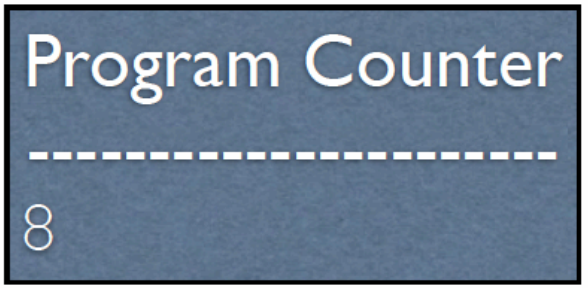
Arithmetic Logic Unit

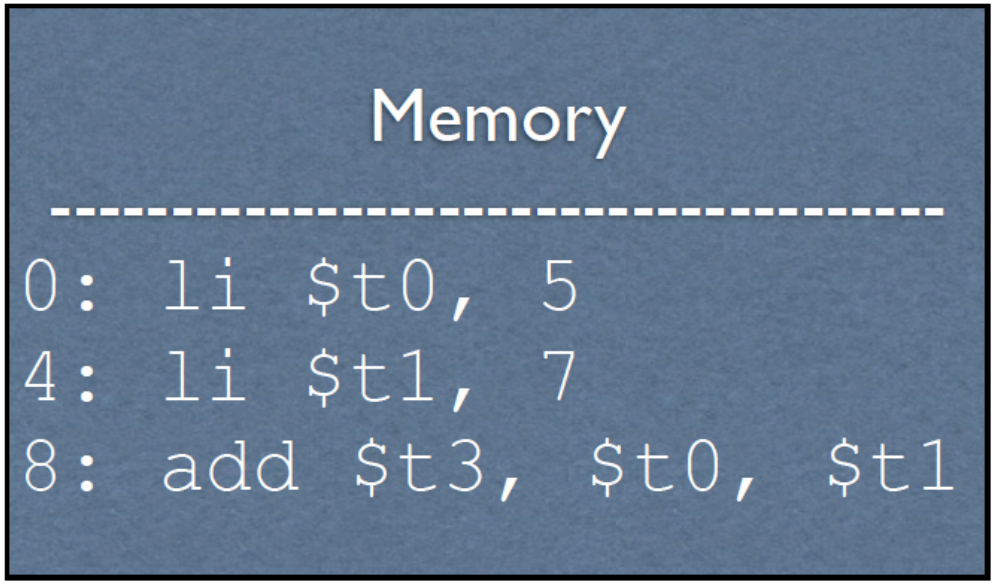
```
5 + 7 = 12
```

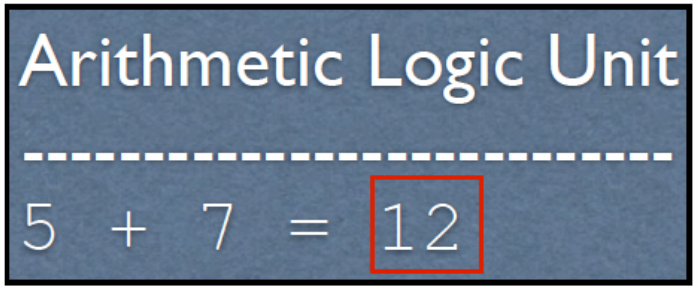


Since all instructions are 32-bits, then they each occupy 4 Bytes of memory.

Memory is addressed in Bytes (more on this later).







Adding More Functionality

- What about: display results???? *Yes, that's kinda important…*
- What would this entail?
  - Engaging with Input / Output part of the computer
  - i.e. talking to devices
    - Q: What usually handles this? **A: the operating system**

- So we need a way to tell
  the operating system to kick in

Talking to the OS

- We are going to be running on MIPS ***emulator*** called **SPIM**
  - Optionally, through a program called **QtSPIM** (GUI based)
  - *What is an emulator?*

- We're not actually running our commands on a MIPS processor!!!

  …so, in other words… we're "faking it"

- Ok, so how might we print something onto std.out?

SPIM Routines

- MIPS features a **`syscall`** instruction, which triggers a ***software interrupt***, or ***exception***

- Outside of an emulator (i.e. in the real world), these instructions **pause the program** and tell the OS to go do something with I/O

- Inside the emulator, it tells the emulator to go do something with I/O

`syscall`

- So we have the OS/emulator's attention, but how does it know what we want?
- The OS/emulator has access to the registers
- We put special values (codes) in the registers to indicate what we want
  - These are codes that can't be used for anything else, so they're understood to be just for `syscall`

Program Files for MIPS Assembly

- The files have to be text
- Typical file extension type is **.asm**
- To leave comments,
  use **#** at the start of the line

(Finally) Printing an Integer

- For SPIM, if register **$v0** contains **1** and then we issue a **syscall**, then SPIM will *print whatever **integer** is stored in register **$a0***

  ← *this is a specific rule using a specific code*

  – Note: $v0 is used for other stuff as well − more on that later…

  – When $v0=1, syscall is *expecting* an integer!

- Other values in **$v0** indicate other types to **syscall**

  Examples:

  – $v0 = 3 means **double (or the mem address of one) in $a0**

  – $v0 = 4 means **string (or the mem address of one) in $a0**

  – We'll explore some of these later, but check MIPS ref card for all of them

(Finally) Printing an Integer

- Remember, the usual syntax to load immediate a value into a register is:

```
li <register>, <value>
```

  - Example: **li $v0, 1** # PUTS THE NUMBER 1 INTO REG. $v0

- To make sure that the register **$a0** has the value of what you want to print out (let's say it's in another register), use the **move** command:

```
move <to register>, <from register>
```

  - Example: **move $a0, $t0** # PUTS THE VALUE IN REG. $t0 INTO REG. $a0

Ok… So About Those Registers
MIPS has 32 registers, each 32 bits

NAME	NUMBER	USE
$zero	0	The Constant Value 0
$at	1	Assembler Temporary
$v0-$v1	2-3	Values for Function Results and Expression Evaluation
$a0-$a3	4-7	Arguments
$t0-$t7	8-15	Temporaries
$s0-$s7	16-23	Saved Temporaries
$t8-$t9	24-25	Temporaries
$k0-$k1	26-27	Reserved for OS Kernel
$gp	28	Global Pointer
$sp	29	Stack Pointer
$fp	30	Frame Pointer
$ra	31	Return Address

Augmenting with Printing

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

# Print an integer to std.output
li $v0, 1
move $a0, $t3
syscall
```

We're Not Quite Done Yet! Exiting an Assembly Program in SPIM

- If you are using SPIM, then you need to say when you are done as well
  - Most HLL programs do this for you automatically

- How is this done?
  - Issue a `syscall` with a special value in **$v0 = 10** (decimal)

Augmenting with Exiting

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

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

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

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 !)

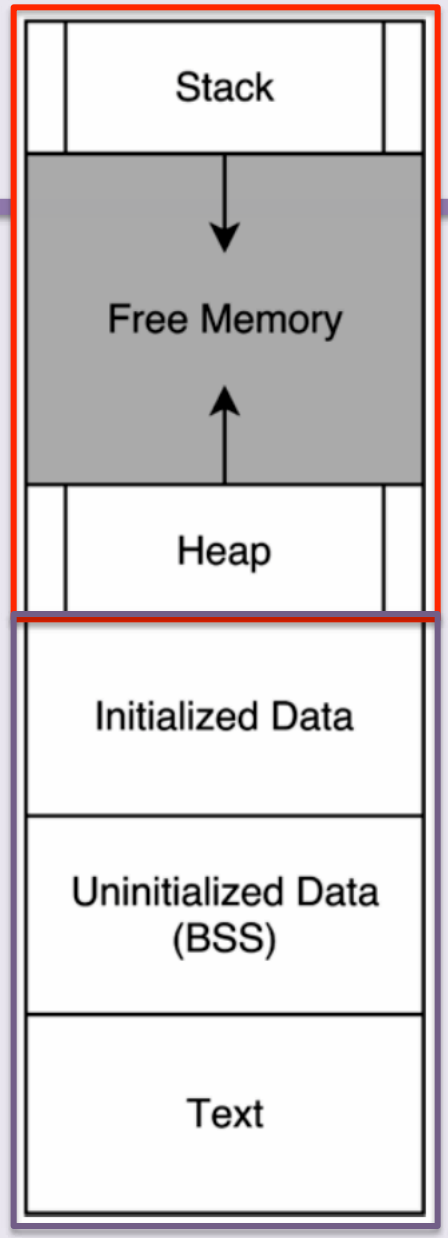
Allocated as program RUN

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
```

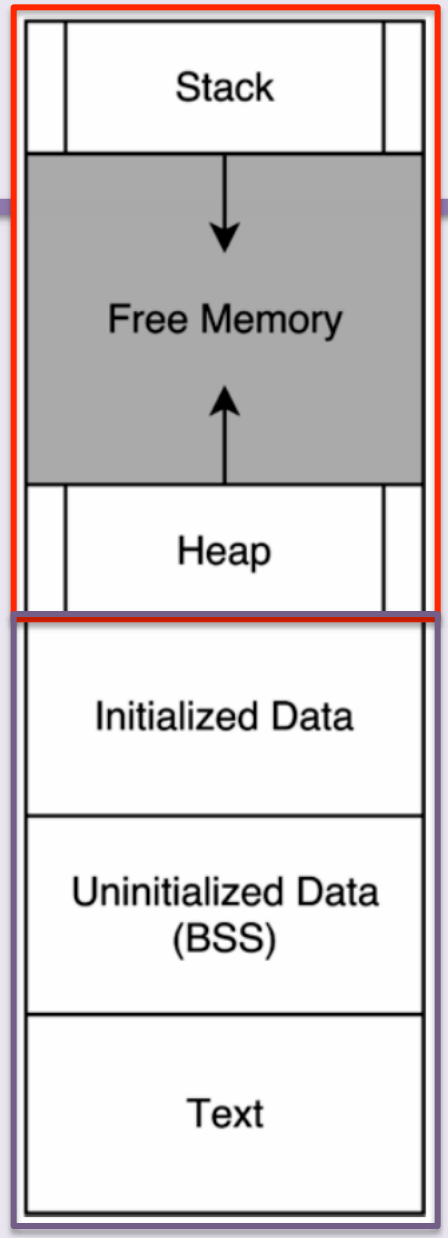
Constants to be used in the program (like strings)

mutable global variables

the text of the program

Allocated as program RUN

Allocated at program LOAD



The `move` instruction

- The move instruction does not actually show up in SPIM

- It is a ***pseudo-instruction*** for us to use, but it's actually translated into an 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 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
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

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`

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

MIPS Peculiarity: **NOR used a NOT**

- How to make a NOT function using ***NOR*** instead
- Recall: NOR = NOT OR
- Truth-Table:

A	B	A NOR B
0	0	**1**
0	1	**0**
1	0	**0**
1	1	**0**

Note that:

0 NOR x = NOT x

- So, in the absence of a NOT function,
  use a NOR with a 0 as one of the inputs!

Let's Run This Program Already!
Using SPIM

- We'll call it **simpleadd.asm**
- Run it on CSIL as: `$ spim -f simpleadd.asm`

- We'll also run other arithmetic programs and explain them as we go along
  - TAKE NOTES!

YOUR TO-DOs

- Review ALL the demo code
  - Available via the class website
- Assignment #1
  - Due Friday
- Assignment/Lab #2
  - Lab on **MONDAY**
  - Submit it via **turnin**

</LECTURE>