MIPS Instructions
Overview of Functions in MIPS

CS 64: Computer Organization and Design Logic
Lecture #8
Winter 2019

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

Administrative

- Lab 4 due Monday, Feb. 4th
- Midterm Exam on Tuesday, Feb. 5th

What's on the Midterm?

What's on It?

- Everything we've done so far from start to end of this week

What Should I Bring?

- Your pencil(s), eraser, MIPS Ref. Card
- THAT'S ALL!

What Else Should I Do?

- Come to the classroom 5-10 minutes EARLY
- I will have some of you re-seated
- Bring your UCSB ID

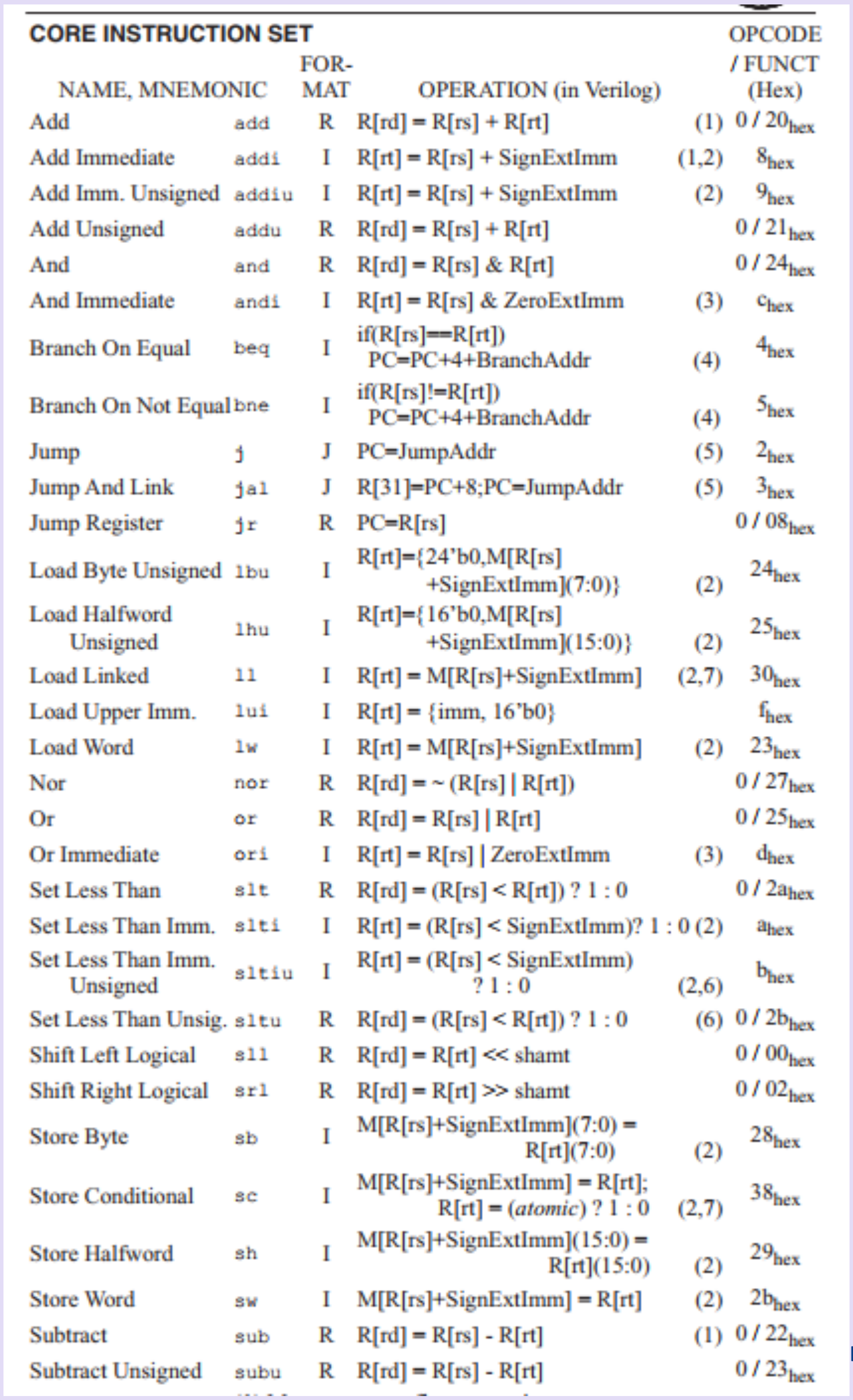
Lecture Outline

- MIPS Instructions
  - How they are represented
- Overview of Functions in MIPS

Any Questions From Last Lecture?

MIPS Reference Card

- Let’s take a closer look at that card…
- Found as PDF on class website



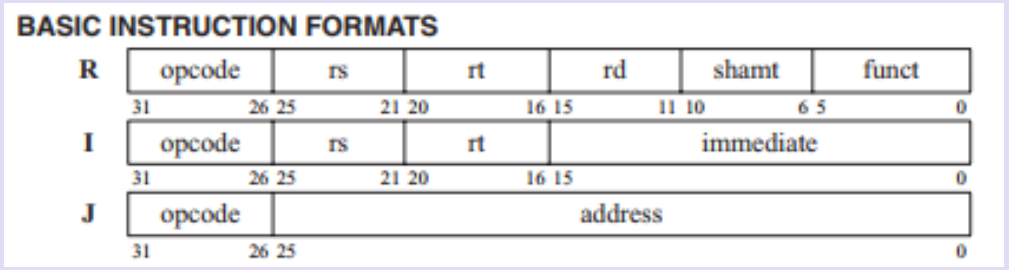
CORE INSTRUCTION SET

NAME, MNEMONIC		FOR-MAT	OPERATION (in Verilog)		OPCODE / FUNCT (Hex)
Add	add	R	R[rd] = R[rs] + R[rt]	(1)	0 / 20_{hex}
Add Immediate	addi	I	R[rt] = R[rs] + SignExtImm	(1,2)	8_{hex}
Add Imm. Unsigned	addiu	I	R[rt] = R[rs] + SignExtImm	(2)	9_{hex}
Add Unsigned	addu	R	R[rd] = R[rs] + R[rt]		0 / 21_{hex}
And	and	R	R[rd] = R[rs] & R[rt]		0 / 24_{hex}
And Immediate	andi	I	R[rt] = R[rs] & ZeroExtImm	(3)	c_{hex}
Branch On Equal	beq	I	if(R[rs]==R[rt]) PC=PC+4+BranchAddr	(4)	4_{hex}
Branch On Not Equal	bne	I	if(R[rs]!=R[rt]) PC=PC+4+BranchAddr	(4)	5_{hex}
Jump	j	J	PC=JumpAddr	(5)	2_{hex}
Jump And Link	jal	J	R[31]=PC+8;PC=JumpAddr	(5)	3_{hex}
Jump Register	jr	R	PC=R[rs]		0 / 08_{hex}
Load Byte Unsigned	lbu	I	R[rt]={24'b0,M[R[rs] +SignExtImm](7:0)}	(2)	24_{hex}
Load Halfword Unsigned	lhu	I	R[rt]={16'b0,M[R[rs] +SignExtImm](15:0)}	(2)	25_{hex}
Load Linked	ll	I	R[rt] = M[R[rs]+SignExtImm]	(2,7)	30_{hex}
Load Upper Imm.	lui	I	R[rt] = {imm, 16'b0}		f_{hex}
Load Word	lw	I	R[rt] = M[R[rs]+SignExtImm]	(2)	23_{hex}
Nor	nor	R	R[rd] = ~ (R[rs] | R[rt])		0 / 27_{hex}
Or	or	R	R[rd] = R[rs] | R[rt]		0 / 25_{hex}
Or Immediate	ori	I	R[rt] = R[rs] | ZeroExtImm	(3)	d_{hex}
Set Less Than	slt	R	R[rd] = (R[rs] < R[rt]) ? 1 : 0		0 / $2a_{hex}$
Set Less Than Imm.	slti	I	R[rt] = (R[rs] < SignExtImm)? 1 : 0	(2)	a_{hex}
Set Less Than Imm. Unsigned	sltiu	I	R[rt] = (R[rs] < SignExtImm) ? 1 : 0	(2,6)	b_{hex}
Set Less Than Unsig.	sltu	R	R[rd] = (R[rs] < R[rt]) ? 1 : 0	(6)	0 / $2b_{hex}$
Shift Left Logical	sll	R	R[rd] = R[rt] << shamt		0 / 00_{hex}
Shift Right Logical	srl	R	R[rd] = R[rt] >> shamt		0 / 02_{hex}
Store Byte	sb	I	M[R[rs]+SignExtImm](7:0) = R[rt](7:0)	(2)	28_{hex}
Store Conditional	sc	I	M[R[rs]+SignExtImm] = R[rt]; R[rt] = (*atomic*) ? 1 : 0	(2,7)	38_{hex}
Store Halfword	sh	I	M[R[rs]+SignExtImm](15:0) = R[rt](15:0)	(2)	29_{hex}
Store Word	sw	I	M[R[rs]+SignExtImm] = R[rt]	(2)	$2b_{hex}$
Subtract	sub	R	R[rd] = R[rs] - R[rt]	(1)	0 / 22_{hex}
Subtract Unsigned	subu	R	R[rd] = R[rs] - R[rt]		0 / 23_{hex}

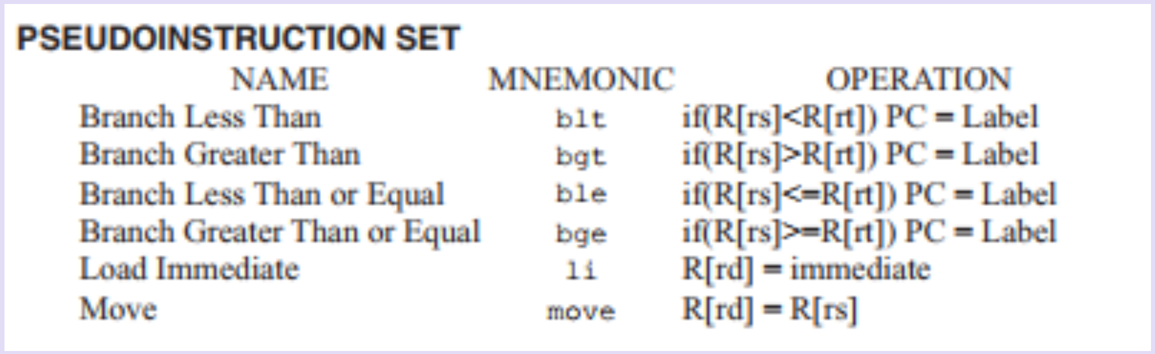
NOTE THE FOLLOWING:

1. Instruction Format Types:
   R vs **I** vs **J**

2. OPCODE/FUNCT (Hex)

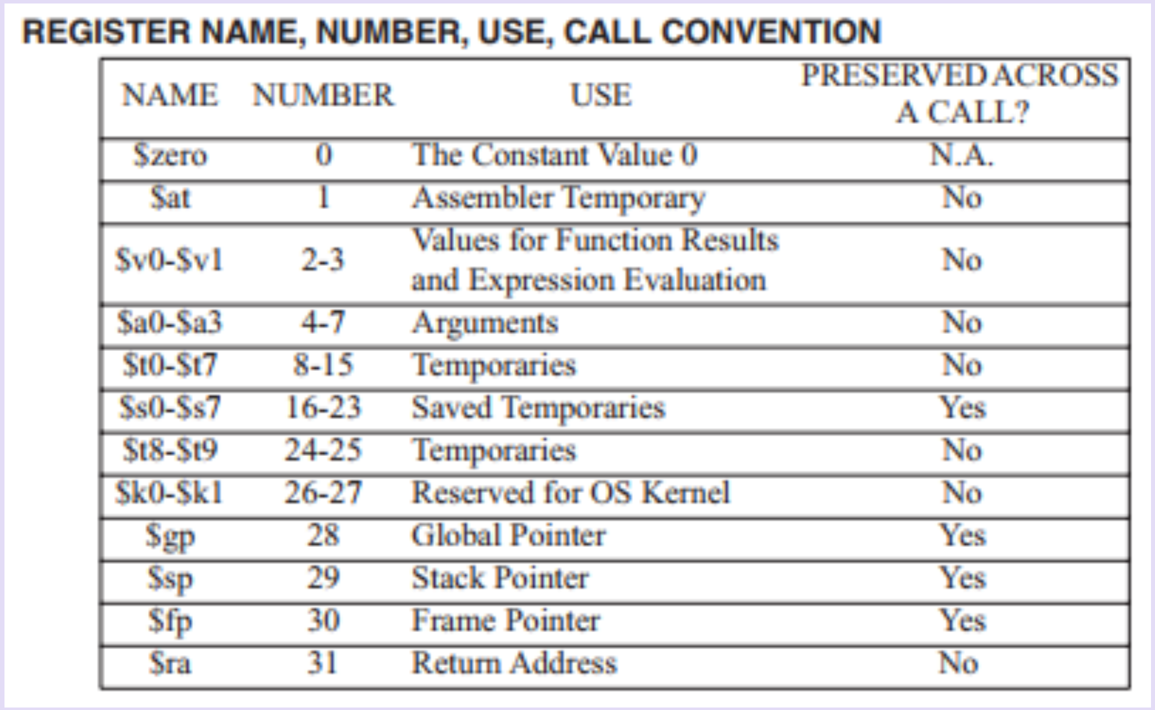


3. Instruction formats:
   Where the actual bits go



PSEUDOINSTRUCTION SET

NAME	MNEMONIC	OPERATION
Branch Less Than	`blt`	if(R[rs]<R[rt]) PC = Label
Branch Greater Than	`bgt`	if(R[rs]>R[rt]) PC = Label
Branch Less Than or Equal	`ble`	if(R[rs]<=R[rt]) PC = Label
Branch Greater Than or Equal	`bge`	if(R[rs]>=R[rt]) PC = Label
Load Immediate	`li`	R[rd] = immediate
Move	`move`	R[rd] = R[rs]



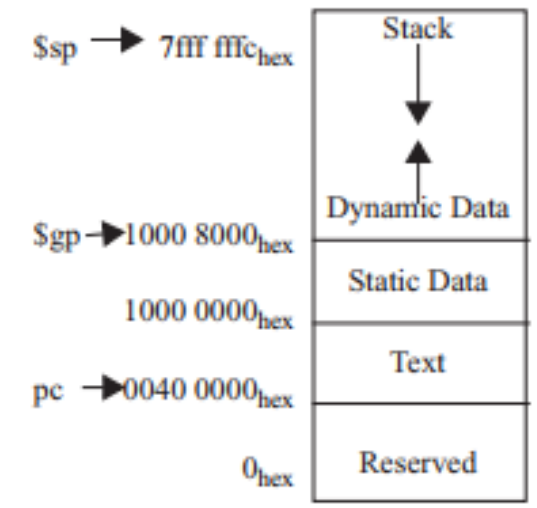
REGISTER NAME, NUMBER, USE, CALL CONVENTION

NAME	NUMBER	USE	PRESERVED ACROSS A CALL?
$zero	0	The Constant Value 0	N.A.
$at	1	Assembler Temporary	No
$v0-$v1	2-3	Values for Function Results and Expression Evaluation	No
$a0-$a3	4-7	Arguments	No
$t0-$t7	8-15	Temporaries	No
$s0-$s7	16-23	Saved Temporaries	Yes
$t8-$t9	24-25	Temporaries	No
$k0-$k1	26-27	Reserved for OS Kernel	No
$gp	28	Global Pointer	Yes
$sp	29	Stack Pointer	Yes
$fp	30	Frame Pointer	Yes
$ra	31	Return Address	No

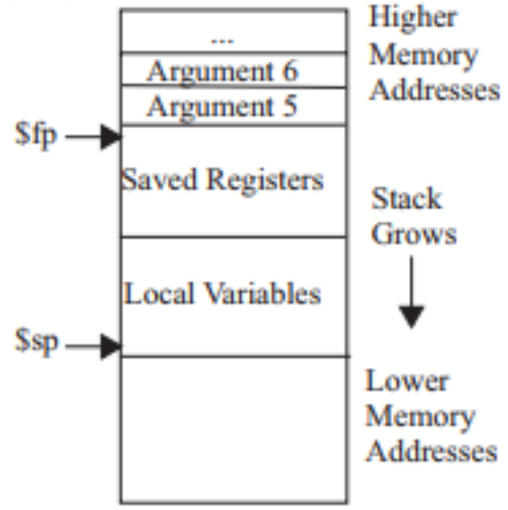
NOTE THE FOLLOWING:

1. Pseudo-Instructions
   - There are more of these, but in CS64, you are ONLY allowed to use these + **la**
2. Registers and their numbers
3. Registers and their uses
4. Registers and their calling convention
   - A LOT more on that later…

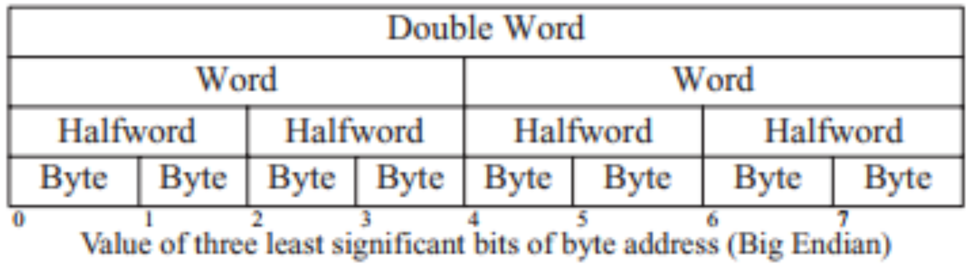
MEMORY ALLOCATION



STACK FRAME



DATA ALIGNMENT



SIZE PREFIXES (10^x for Disk, Communication; 2^x for Memory)

SIZE	PRE-FIX	SIZE	PRE-FIX	SIZE	PRE-FIX	SIZE	PRE-FIX
$10^3, 2^{10}$	Kilo-	$10^{15}, 2^{50}$	Peta-	10^{-3}	milli-	10^{-15}	femto-
$10^6, 2^{20}$	Mega-	$10^{18}, 2^{60}$	Exa-	10^{-6}	micro-	10^{-18}	atto-
$10^9, 2^{30}$	Giga-	$10^{21}, 2^{70}$	Zetta-	10^{-9}	nano-	10^{-21}	zepto-
$10^{12}, 2^{40}$	Tera-	$10^{24}, 2^{80}$	Yotta-	10^{-12}	pico-	10^{-24}	yocto-

The symbol for each prefix is just its first letter, except μ is used for micro.

NOTE THE FOLLOWING:

1. This is only part of the 2nd page that you need to know

Instruction Representation

Recall: A MIPS instruction has 32 bits

32 bits are divided up into 6 fields *(aka the* ***R-Type*** *format)*

• **op** code	6 bits	basic operation
• **rs** code	5 bits	first register source operand
• **rt** code	5 bits	second register source operand
• **rd** code	5 bits	register destination operand
• **shamt** code	5 bits	shift amount
• **funct** code	6 bits	function code

Why did the designers allocate 5 bits for registers?

op	**rs**	**rt**	**rd**	**shamt**	**funct**
6 b	5 b	5 b	5 b	5 b	6 b
31 – 26	*25 – 21*	*20 – 16*	*15 – 11*	*10 – 6*	*5 – 0*

Instruction Representation in R-Type

op	rs	rt	rd	shamt	funct
6 b	5 b	5 b	5 b	5 b	6 b
31 – 26	*25 – 21*	*20 – 16*	*15 – 11*	*10 – 6*	*5 – 0*

- The combination of the **opcode** and the **funct** code tell the processor what it is supposed to be doing
- Example:

```
add $t0, $s1, $s2
```

op	rs	rt	rd	shamt	funct
0	17	18	8	0	32

op = 0, funct = 32	mean "add"
rs = 17	means "$s1"
rt = 18	means "$s2"
rd = 8	means "$t0"
shamt = 0	means this field is unused in this instruction

A full list of codes can be found in your ***MIPS Reference Card***

Exercises

- Using your MIPS Reference Card, write the 32 bit instruction (using the R-Type format and decimal numbers for all the fields) for the following:

```
add $t3, $t2, $s0        0x01505820
addu $a0, $a3, $t0       0x00E82021
sub $t1, $t1, $t2        0x012A4822
```

Exercise: Example Run-Through

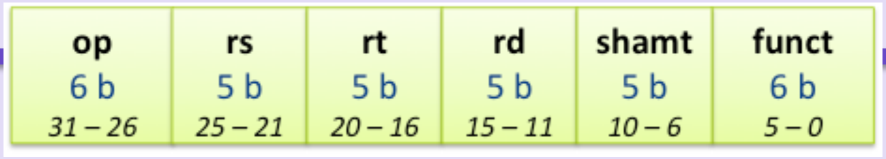
- Using your MIPS Reference Card, write the 32 bit instruction (using the R-Type format) for the following. Express your final answer in hexadecimal.

```
add $t3, $t2, $s0
```

0x01505820

op (6b)	**rs** (5b)	**rt** (5b)	**rd** (5b)	**shamt** (5b)	**funct** (6b)
0	10	16	11	0	32
000000	0 1010	1 0000	0 1011	0 0000	10 0000
00000001010100000101100000100000					
0x01505820					

Instruction Representation



op	rs	rt	rd	shamt	funct
6 b	5 b	5 b	5 b	5 b	6 b
31 – 26	*25 – 21*	*20 – 16*	*15 – 11*	*10 – 6*	*5 – 0*

- The R-Type format is used for many, but not all instructions
  - Why?

    Hint: how many registers are there? How bits represent a register in R-Type format?

- What if you wanted to load/save from/to memory?
  - Why is this problematic with R-Type format?

A Second Type of Format…

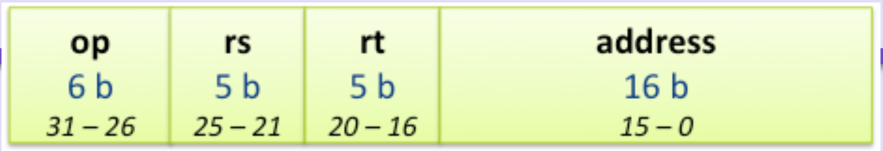
32 bits are divided up into 4 fields *(the **I-Type** format)*

• **op** code	6 bits	basic operation
• **rs** code	5 bits	first register source operand
• **rt** code	5 bits	second register source operand
• **address** code	16 bits	constant or memory address

Note: The I-Type format uses the ***address*** field to access ±2^{15} addresses from whatever value is in the ***rs*** field

op	**rs**	**rt**	**address**
6 b	5 b	5 b	16 b
31 – 26	*25 – 21*	*20 – 16*	*15 – 0*

I-Type Format



op	rs	rt	address
6 b	5 b	5 b	16 b
31 – 26	*25 – 21*	*20 – 16*	*15 – 0*

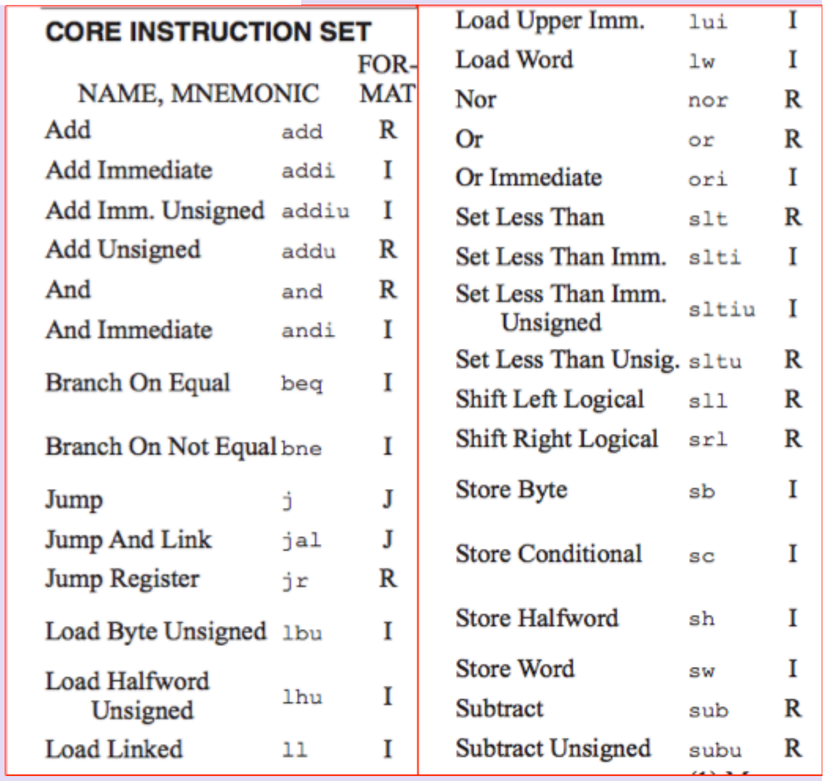
- The I-Type ***address*** field is a <u>signed</u> number

- The **`addi`** instruction is an I-Type, example:

```
addi $t0, $t1, 42
```

  – What is the largest, most positive, number you can put as an immediate?

Ans: 2^{15} - 1



CORE INSTRUCTION SET

NAME, MNEMONIC		FOR-MAT			
Add	add	R	Load Upper Imm.	lui	I
Add Immediate	addi	I	Load Word	lw	I
Add Imm. Unsigned	addiu	I	Nor	nor	R
Add Unsigned	addu	R	Or	or	R
And	and	R	Or Immediate	ori	I
And Immediate	andi	I	Set Less Than	slt	R
Branch On Equal	beq	I	Set Less Than Imm.	slti	I
Branch On Not Equal	bne	I	Set Less Than Imm. Unsigned	sltiu	I
Jump	j	J	Set Less Than Unsig.	sltu	R
Jump And Link	jal	J	Shift Left Logical	sll	R
Jump Register	jr	R	Shift Right Logical	srl	R
Load Byte Unsigned	lbu	I	Store Byte	sb	I
Load Halfword Unsigned	lhu	I	Store Conditional	sc	I
Load Linked	ll	I	Store Halfword	sh	I
			Store Word	sw	I
			Subtract	sub	R
			Subtract Unsigned	subu	R

Instruction Representation in I-Type

op	rs	rt	address
6 b	5 b	5 b	16 b
31 – 26	*25 – 21*	*20 – 16*	*15 – 0*

- Example:

`addi $t0, $s0, 124`

op	rs	rt	address/const
8	16	8	124

op = 8	mean "addi"
rs = 16	means "$s0"
rt = 8	means "$t0"
address/const = 124	is the immediate value

A full list of codes can be found in your ***MIPS Reference Card***

Exercises

- Using your MIPS Reference Card, write the 32 bit instruction (using the I-Type format and decimal numbers for all the fields) for the following:

```
addi $t3, $t2, -42      0x214BFFD6
andi $a0, $a3, 1        0x30E40001
slti $t8, $t8, 14       0x2B18000E
```

Functions

- Up until this point, we have not discussed **functions**

- Why not?
  - If you want to do functions, you need to use **the stack**
  - Memory management is a <u>must</u> for the call stack ...
    though we can make *some* progress without it

- Think of recursion…
  - How many variables are we going to need ahead of time?
  - What memory do we end up using in recursive functions?
  - We don't always know…

Implementing Functions

What capabilities do we need for functions?

1. Ability to execute code elsewhere
   – Branches and jumps

2. Way to pass arguments in and out of the func.
   – There a way (aka convention) to do that that we'll learn about
   – We'll use the registers to do function I/O

Jumping to Code

```
void foo() {
  bar();
  baz();
}
```

```
void bar() {
   ...
}
```

```
void baz() {
   ...
}
```

- We have ways to jump unconditionally to code (**`j`** instruction)

- But what about ***jumping back***?
  - That is, after you're done with a function?
  - We'll need a way to *save* where we were (so we can "jump" back)

- **Q**: What do need so that we can do this on MIPS?
  - **A**: A way to store the program counter (**$PC**) multiple times

  (to tell us where the *next* instruction is so that we know *where* to return!)

Calling Functions on MIPS

- Two crucial instructions: `jal` and `jr`
- One specialized register: **`$ra`**

- `jal` (**jump-and-link**)
  - Simultaneously **jump to an address**, and **store the location of the next instruction** in register **`$ra`**

- `jr` (**jump-register**)
  - **Jump to the address stored in a register**, often **`$ra`**

Simple Call Example

- See program: **simple_call.asm**

```
# Calls a function (test) which immediately returns
.text
test:  # return to whoever made the call
       jr $ra

main:  # do stuff…
       # then call the test function
       jal test

exit:  # exit
       li $v0, 10
       syscall
```

*Note: SPIM always starts execution at the line labeled “**main**”*

Passing and Returning Values

- We want to be able to call arbitrary functions without knowing the implementation details
- So, we need to know our pre-/post-conditions
- Q: How might we achieve this in MIPS?
  - A: We designate specific registers for **arguments** and **return values**

Passing and Returning Values in MIPS

- Registers **`$a0`** thru **`$a3`**
  - **Argument registers**, for passing function arguments
- Registers **`$v0`** and **`$v1`**
  - **Return registers**, for passing return values
- What if we want to pass >4 args?
  - There are ways around that…
    but we won't discuss them in CS64…!

YOUR TO-DOs

- Review ALL the demo codes
  - Available via the class website
- Study for Midterm Exam
  - Review Practice Exam
- Remember: Midterm next week!!!

</LECTURE>