

# **MIPS Instructions**

**CS 64: Computer Organization and Design Logic**  
**Lecture #8**

Ziad Matni  
Dept. of Computer Science, UCSB

# Administrative

---

- Your midterm exam is next week on **Thurs. 2/15**

# Lecture Outline

---

- Midterm Exam: What to Expect, How to Prep
- MIPS Instruction Representations

# MIDTERM IS COMING!

- **Thursday, 2/15** in this classroom
- **Starts at 3:30pm **\*\*SHARP\*\*****
  - Please start arriving 5-10 minutes before class
- **I may ask you to change seats**
- Please bring your UCSB IDs with you
- **Closed book: no calculators, no phones, no computers**
- **Only the MIPS Reference Card is allowed**
- **You will write your answers on the exam sheet itself.**



# What's on the Midterm?? 1/2

- Data Representation
  - Convert bin  $\leftrightarrow$  hex  $\leftrightarrow$  decimal  $\leftrightarrow$  bin
  - Signed and unsigned binaries
- Logic and Arithmetic
  - Binary addition, subtraction
    - Carry and Overflow
  - Bitwise AND, OR, NOT, XOR
  - General rules of AND, OR, XOR, using NOR as NOT
- All demos done in class
- Lab assignments 1, 2 , 3 and 4

# What's on the Midterm?? 2/2

## Assembly

- Core components of a CPU
  - How instructions work
- Registers (\$t, \$s, \$a, \$v)
- Arithmetic in assembly (add, subtract, multiply, divide)
  - What's the difference between add, addi, addu, addui, etc...
- Conditionals and loops in assembly
- Conversion to and from Assembly and C/C++
- syscall and its various uses (printing output, taking input, ending program)
- **.data** and **.text** declarations
- Memory in MIPS
- Big Endian vs Little Endian
- R-type and I-type instructions
- Pseudo instructions

# Midterm Study Guide

---

- See the Class Website

# MIPS Reference Card

---

- Let's take a closer look at that card...
- Found inside front cover of your textbook
- Also found as PDF on class website



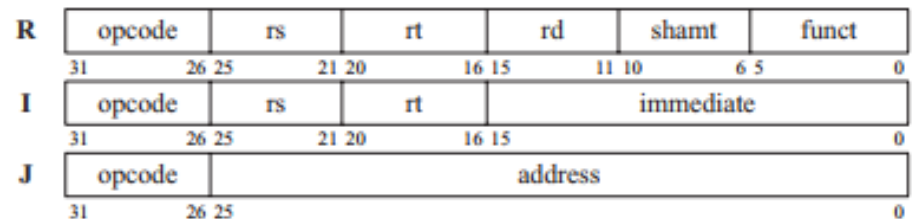
CORE INSTRUCTION SET			OPCODE / FUNCT (Hex)
NAME, MNEMONIC	FOR-MAT	OPERATION (in Verilog)	
Add	add	R R[rd] = R[rs] + R[rt]	(1) 0 / 20 <sub>hex</sub>
Add Immediate	addi	I R[rt] = R[rs] + SignExtImm	(1,2) 8 <sub>hex</sub>
Add Imm. Unsigned	addiu	I R[rt] = R[rs] + SignExtImm	(2) 9 <sub>hex</sub>
Add Unsigned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

## NOTE THE FOLLOWING:

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

2. OPCODE/FUNCT (Hex)

### BASIC INSTRUCTION FORMATS



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	btle	if( $R[rs] \leq R[rt]$ ) PC = Label
Branch Greater Than or Equal	bge	if( $R[rs] \geq R[rt]$ ) PC = Label
Load Immediate	li	$R[rd] = \text{immediate}$
Move	move	$R[rd] = R[rs]$

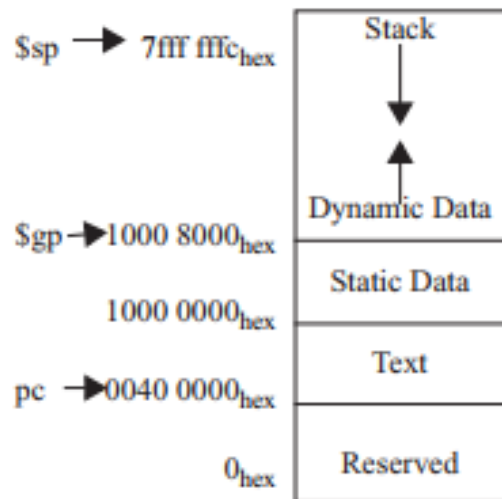
## NOTE THE FOLLOWING:

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...

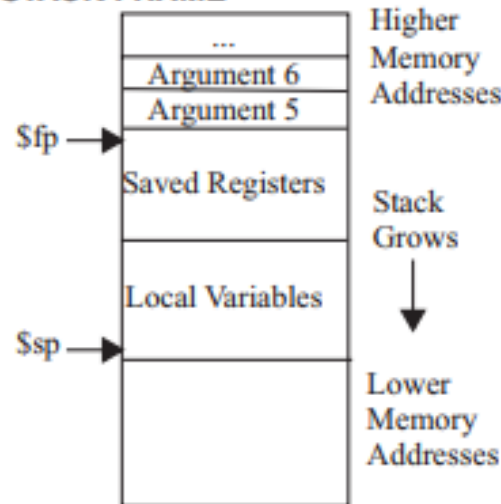
## REGISTER NAME, NUMBER, USE, CALL CONVENTION

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

## MEMORY ALLOCATION



## STACK FRAME



## NOTE THE FOLLOWING:

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

## DATA ALIGNMENT

Double Word							
Word				Word			
Halfword		Halfword		Halfword		Halfword	
Byte	Byte	Byte	Byte	Byte	Byte	Byte	Byte
0	1	2	3	4	5	6	7

Value of three least significant bits of byte address (Big Endian)

## SIZE PREFIXES (10<sup>x</sup> for Disk, Communication; 2<sup>x</sup> for Memory)

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

The symbol for each prefix is just its first letter, except  $\mu$  is used for micro.

# Pseudoinstructions

- These are “macros” for more specific instructions in MIPS
  - Often easier to use than not

- *Example:*     **la \$register, 32-bit memory address**

This is actually 2 MIPS instructions:

*lui \$register, upper-part-of memory address*

*ori \$register, \$register, lower-part-of memory address*

This is all due to the fact that **I-types** of instructions **can only deal with SIXTEEN (16) bits** of data at once and MIPS memory words are actually 32 bits.

What is **lui**? Is there an **lli**? What is **ori**?

# Instruction Representation

Recall: A MIPS instruction has 32 bits

32 bits are divided up into 5 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

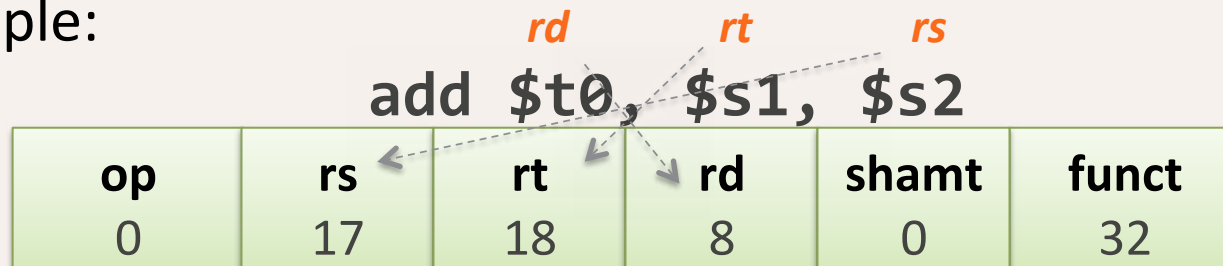
*Why did the designers allocate 5 bits for registers?*

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:



op = 0, funct = 32

rs = 17

rt = 18

rd = 8

shamt = 0

mean “add”

means “\$s1”

means “\$s2”

means “\$t0”

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) for the following. Express your final answer in hexadecimal.

*rd*      *rs*      *rt*  
add \$t3, \$t2, \$s0      0x01505820  
addu \$a0, \$a3, \$t0      0x00E82021  
sub \$t1, \$t1, \$t2      0x012A4822

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?
- What if you wanted an instruction 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/immediate** code  
                                 16 bits                  constant or memory address

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

<b>op</b>	<b>rs</b>	<b>rt</b>	<b>address</b>
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 signed number
  - It can be positive or negative

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

**addi** <sup>rt</sup>\$t0, <sup>rs</sup>\$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/immediate
8	16	8	124

op = 8

mean “addi”

rs = 8

means “\$t0”

rt = 16

means “\$s0”

address/const = 124 is the immediate value  
(note 124 is in decimal)

*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`  
`lw $t1, 0x10008001`      `0x8C098001`

What's -42 in 16-bit binary?  
 +42 = 0000 0000 0010 0110  
 So, -42 = 1111 1111 1101 1010

op	rs	rt	address/immediate
8	10	11	-42
00 1000	0 1010	0 1011	1111 1111 1101 1010
00100001010010111111111111011010			
0x214BFFD6			



# A Review of Bitwise Shifting

- Recall: you can bitwise shift a number to the LEFT or to the RIGHT
  - Shifting left: MIPS instruction `sll`
  - Shifting right: MIPS instruction `srl` ***and*** `sra`
- Why 2 different ones for shifting right??
  - One is called shift right *logical* and the other shift right *arithmetic*

# srl vs sra

- srl replaces the “lost” MSBs with 0s
- sra replaces the “lost” MSBs with *either* 0s (if number is +ve) *or* 1s (if number is –ve)

## EXAMPLE:

```
addi $t0, $zero, 12  
addi $t1, $zero, -12
```

```
srl $s0, $t0, 1  
sra $s1, $t0, 1  
srl $s0, $t1, 1  
sra $s1, $t1, 1
```

# sr1 vs sra

- sr1 replaces the “lost” MSBs with 0s
- sra replaces the “lost” MSBs with *either* 0s (if number is +ve) *or* 1s (if number is –ve)

## IMPLICATIONS:

- sr1 should NOT be used for negative numbers
- sra use for positive numbers is redundant
- When using negative numbers, use sra



# YOUR TO-DOs

---

- Study for the midterm
- NO LAB NEXT WEEK!
- Turn in Assignment/Lab #4

**</LECTURE>**