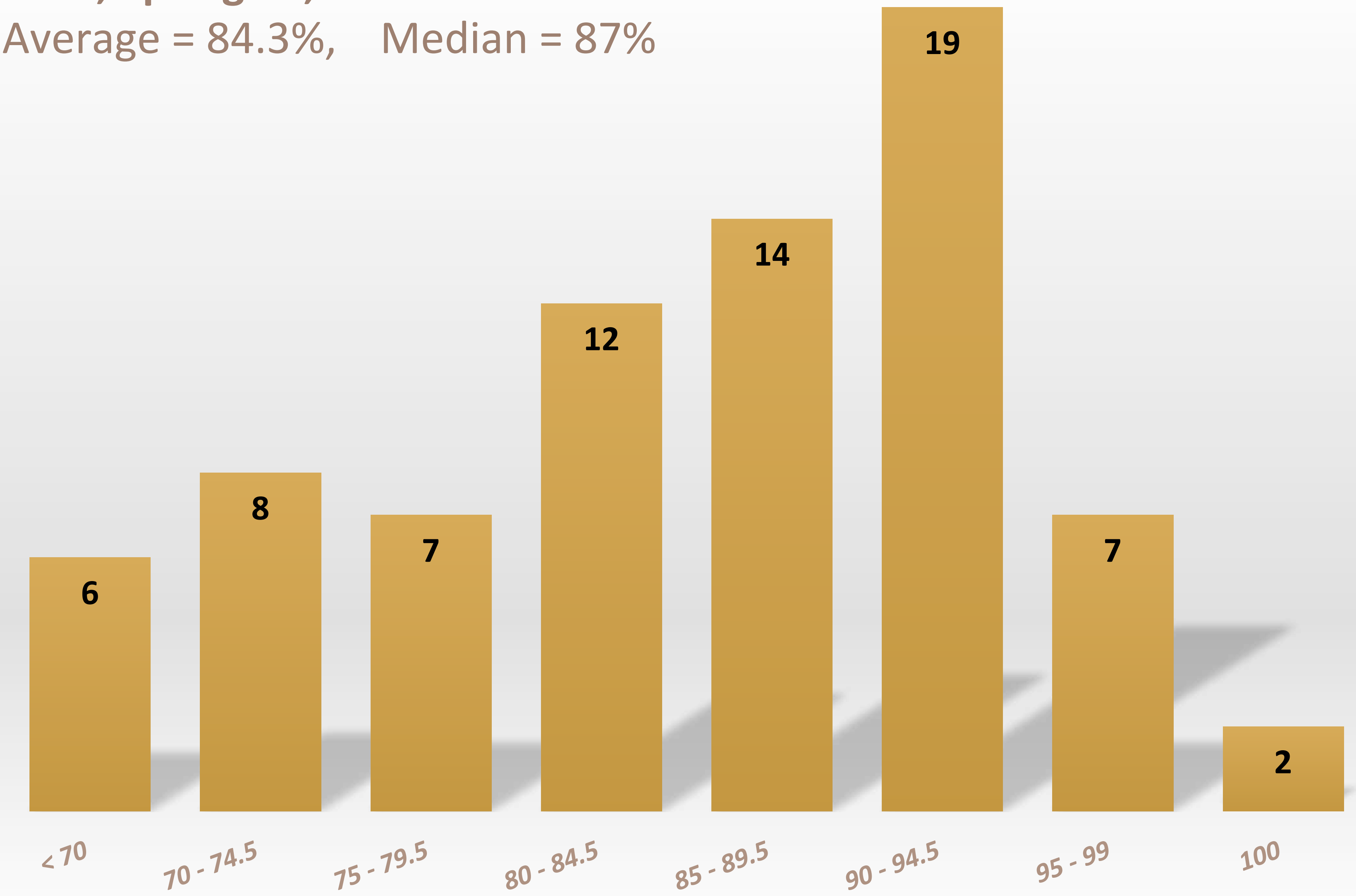
MIPS Assembly:
More about MIPS Instructions
Using Functions in MIPS

CS 64: Computer Organization and Design Logic
Lecture #8

Ziad Matni
Dept. of Computer Science, UCSB

CS 64, Spring 18, Midterm#1 Exam

Average = 84.3%, Median = 87%



Reviewing Your Midterm#1 Exam

If your FAMILY name is:	**Then go see T.A.:**
BAAS to LIANG (inclusive)	Fatih Bakir Wednesdays 5 - 7 PM
LIN to ZHOU (inclusive)	Bay-Yuan Hsu Fridays 11 AM - 1 PM

- T.A. office hours are in PHELP 3525
- Exams have to stay with T.As at ALL times
- No copying / no photos of exams allowed

Lecture Outline

- Constructing Instructions in MIPS
- Functions in MIPS

Last Week’s Exercise

- 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					

5-Minute Pop Quiz!!!

Write the machine code, in Hex,

for this MIPS instruction:

```
sub $t1, $t5, $t2
```

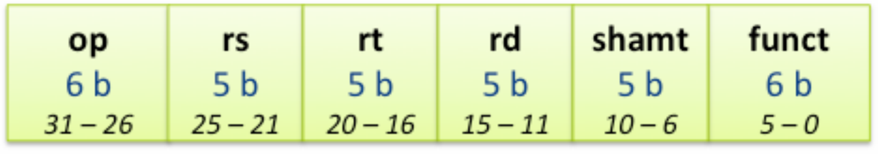
5-Minute Pop Quiz!!!

Write the machine code, in Hex,

for this MIPS instruction:

`sub $t1, $t5, $t2` **0x01AA4822**

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*

- What if you wanted an instruction to

  add using an immediate value?
  load/save from/to memory?

  - Why are these instructions unusable with an 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 ±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*

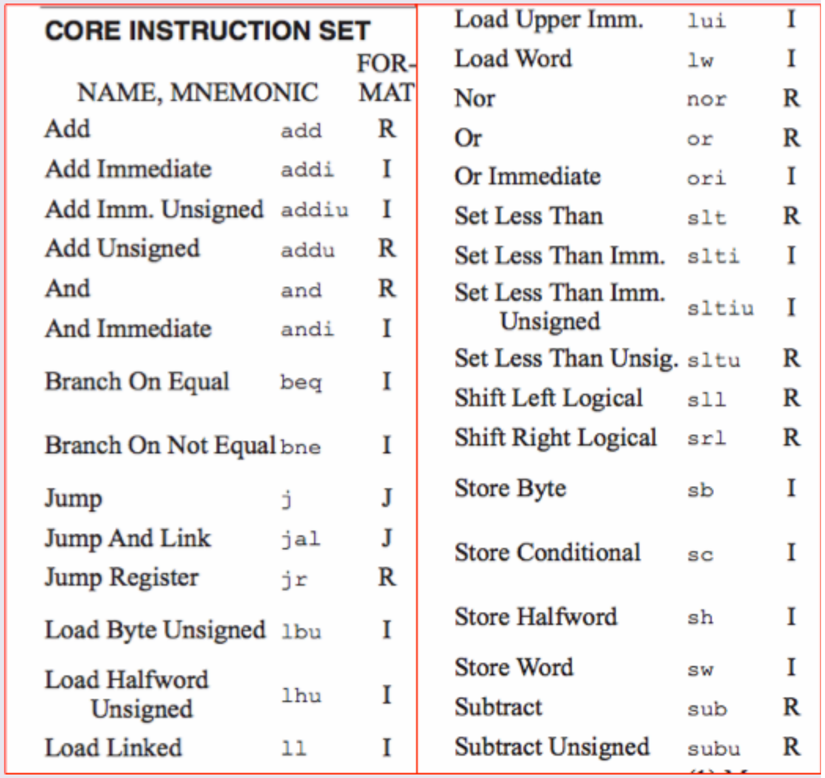
- The I-Type ***address*** field is a signed number
  - It can be positive or negative

- 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?

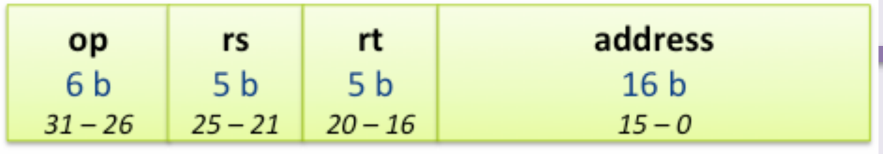
Ans: 2^{15} - 1



CORE INSTRUCTION SET

NAME, MNEMONIC		FOR-MAT
Add	add	R
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

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 = 16	means "$s0"
rt = 8	means "$t0"
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, 0x1   0x30E40001
```

`srl` vs `sra`
Shift-Right Logic vs Arithmetic

- `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)

IMPLICATIONS:

- `srl` should NOT be used with negative numbers
  - That is, **unsigned** use only
- `sra` should be used with **signed** numbers
  - Can also be used with unsigned, but there's srl for that…

sra vs **srl** Exercise

DEMO!
srlsra.asm

- Is **sra (-19) >> 2** the same as **-(srl (19)) >> 2**?

```
19 =     0000000000000000 0000000000010011
-19 =    1111111111111111 1111111111101101

                                                  goes away
sra(-19) = 1111111111111111 1111111111111011 (01)
         = -5
                                                      goes away
-srl(19) = -(0000000000000000 0000000000000100) (11)
         = -4
```

sra vs srl Exercise

DEMO!
shiftDemo.asm

- `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
```

Functions

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

- Why not?
  - Memory management is a must 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?

Implementing Functions

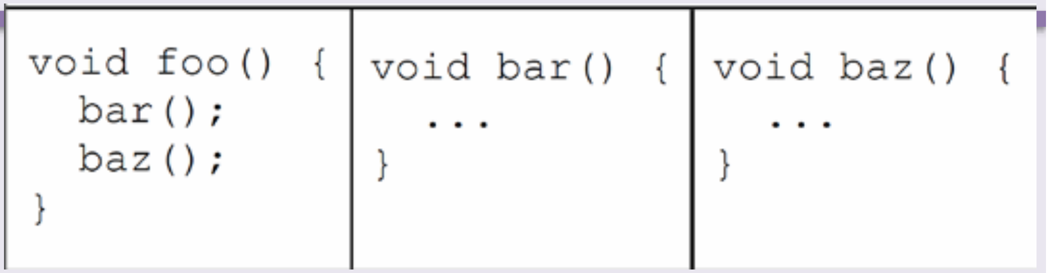
What capabilities do we need for functions?

1. Ability to execute code elsewhere
   - Branches and jumps

2. Way to pass arguments
   - There a way (convention) to do that that we'll learn…

3. Way to return values
   - Registers

Jumping to Code



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

- We have ways to jump to code (`j` instruction)
- But what about ***jumping back***?
  - 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)
    (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…!

Passing and Returning Values in MIPS

Demo: *print_ints.asm*

- Illustrates the use of a printing sub-routine (i.e. like a simple function)

Passing and Returning Values in MIPS

Demo: *print_ints.asm*

- Illustrates the use of a printing sub-routine (i.e. like a simple function)

- How would you write this function in C++?

```
void print_ints(int a0, int a1)
{
 cout << a0 << endl << a1 << endl;
}
```

Passing and Returning Values in MIPS

Demo: *add_ints.asm*

- Illustrates the use of an adding sub-routine
  (i.e. like a simple function that returns a value)

Passing and Returning Values in MIPS

Demo: *add_ints.asm*

– Illustrates the use of an adding sub-routine (i.e. like a simple function that returns a value)

- How would you write this function in C++?

```
int add_ints(int a0, int a1)
{
 v0 = a0 + a1;
 return (v0);
}
```

YOUR TO-DOs

- Finish assignment/Lab #4
  - Assignment due on FRIDAY

</LECTURE>