MIPS Assembly: More about Memory and Instructions

CS 64: Computer Organization and Design Logic
Lecture #7

Ziad Matni
Dept. of Computer Science, UCSB

Lecture Outline

- Global variables and memory
- Arrays in MIPS
- Pseudo-Instructions
- Constructing Instructions in MIPS

Global Variables

Recall:

- Typically, global variables are placed directly in memory, not registers
- **lw** and **sw** for **load word** and **save word**
  - **lw** is NOT the same as **la** is NOT the same as **move**!!!
  - Syntax:

    lw ***register_destination***, **N**(***register_with_address***)

    Where **N = offset of address in bytes**
- Let's take a look at: ***access_global.asm***

access_global.asm

Load Address (la) and Load Word (lw)

```
.data
myVariable: .word 42

main:                                         $t0 = &myVariable
la $t0, myVariable             ← WHAT'S IN $t0??
lw $t1, 0($t0)                 ← WHAT DID WE DO HERE??

li $v0, 1
move $a0, $t1
syscall                        ← WHAT SHOULD WE SEE HERE??
```

access_global.asm

Store Word (sw)

```
li $t1, 5
sw $t1, 0($t0)          ← WHAT'S IN $t0 AGAIN??

li $t1, 0
lw $t1, 0($t0)          ← WHAT DID WE DO HERE??

li $v0, 1
move $a0, $t1
syscall              ← WHAT SHOULD WE SEE HERE??
```

Arrays

- Question:

As far as memory is concerned, what is the *major difference* between an **array** and a **global variable**?

  – Arrays contain multiple elements

- Let's take a look at:
  – print_array1.asm
  – print_array2.asm
  – print_array3.asm

print_array1.asm

```
int myArray[]
   = {5, 32, 87, 95, 286, 386};
int myArrayLength = 6;
int x;

for (x = 0; x < myArrayLength; x++)
{
   print(myArray[x]);
   print("\n");
}
```

```
# C code:
# int myArray[] =
#     {5, 32, 87, 95, 286, 386}
# int myArrayLength = 6
# for (x = 0; x < myArrayLength; x++) {
#   print(myArray[x])
#   print("\n") }
.data
newline: .asciiz "\n"
myArray: .word 5 32 87 95 286 386
myArrayLength: .word 6

.text
main:
     # t0: x
     # initialize x
     li $t0, 0
loop:
     # get myArrayLength, put result in $t2
     # $t1 = &myArrayLength
     la $t1, myArrayLength
     lw $t2, 0($t1)

     # see if x < myArrayLength
     # put result in $t3
     slt $t3, $t0, $t2
     # jump out if not true
     beq $t3, $zero, end_main

     # get the base of myArray
     la $t4, myArray

     # figure out where in the array we need
     # to read from. This is going to be the array
     # address + (index << 2). The shift is a
     # multiplication by four to index bytes
     # as opposed to words.
     # Ultimately, the result is put in $t7
     sll $t5, $t0, 2
     add $t6, $t5, $t4
     lw $t7, 0($t6)

     # print it out, with a newline
     li $v0, 1
     move $a0, $t7
     syscall
     li $v0, 4
     la $a0, newline
     syscall

     # increment index
     addi $t0, $t0, 1

     # restart loop
     j loop

end_main:
     # exit the program
     li $v0, 10
     syscall
```

print_array2.asm

- Same as print_array1.asm, ***except that*** in the assembly code, we lift redundant computation out of the loop.

- This is the sort of thing a decent compiler (**clang** or **gcc** or **g++**, for example) will do with a HLL program

print_array3.asm

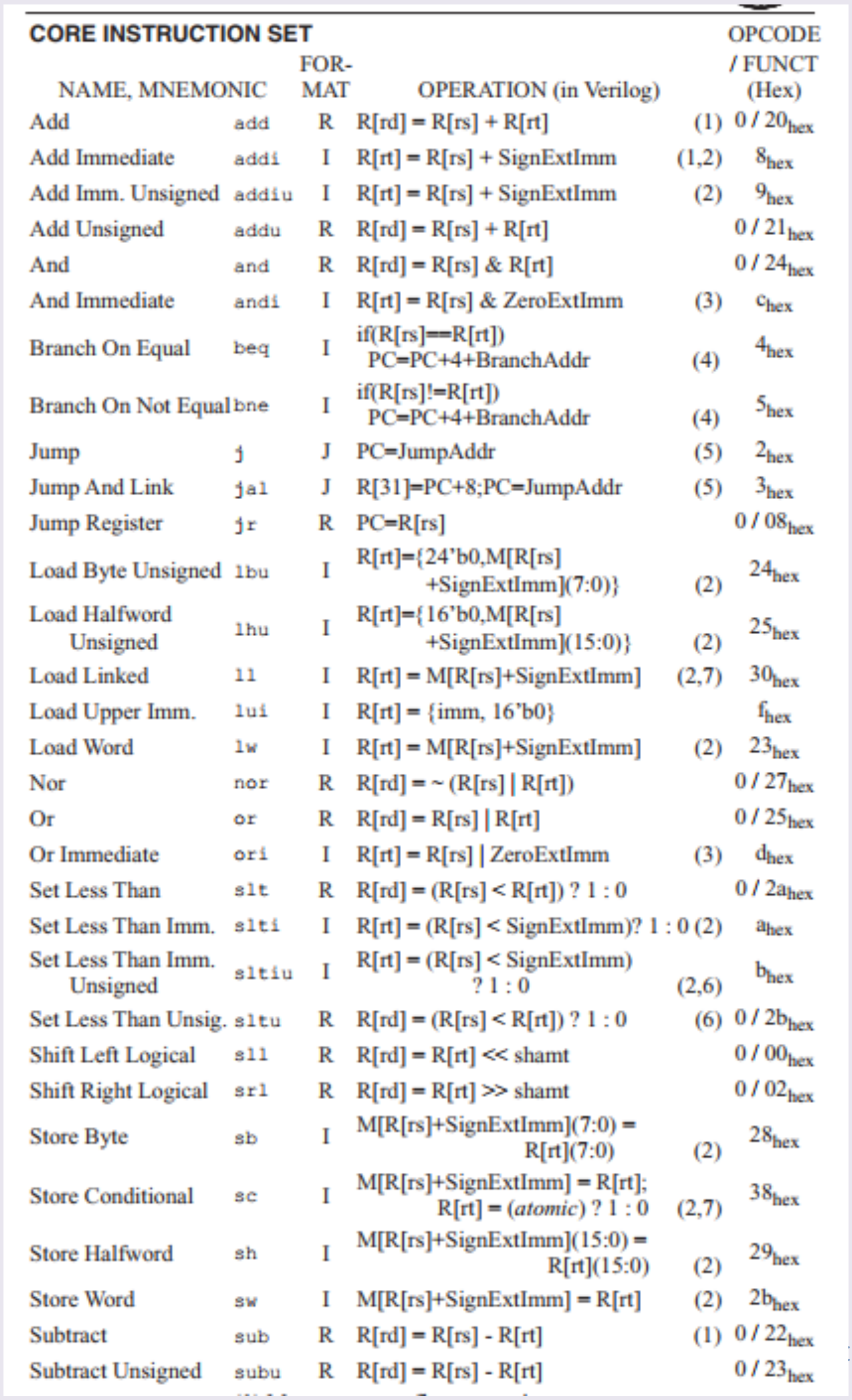
```
int myArray[]
   = {5, 32, 87, 95, 286, 386};
int myArrayLength = 6;
int* p;

for ( p = myArray; p < myArray + myArrayLength; p++)
{
   print(*p);
   print("\n");
}
```

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



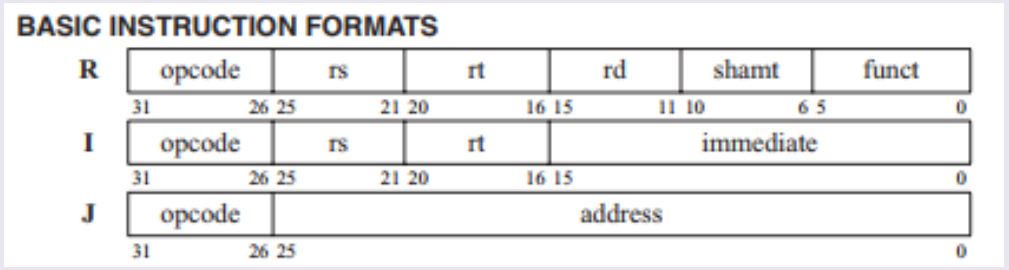
CORE INSTRUCTION SET

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**

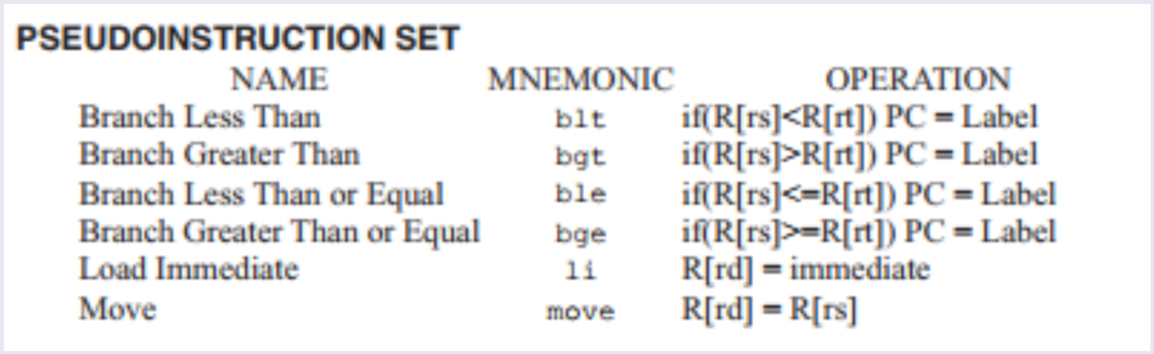
2. OPCODE/FUNCT (Hex)



BASIC INSTRUCTION FORMATS

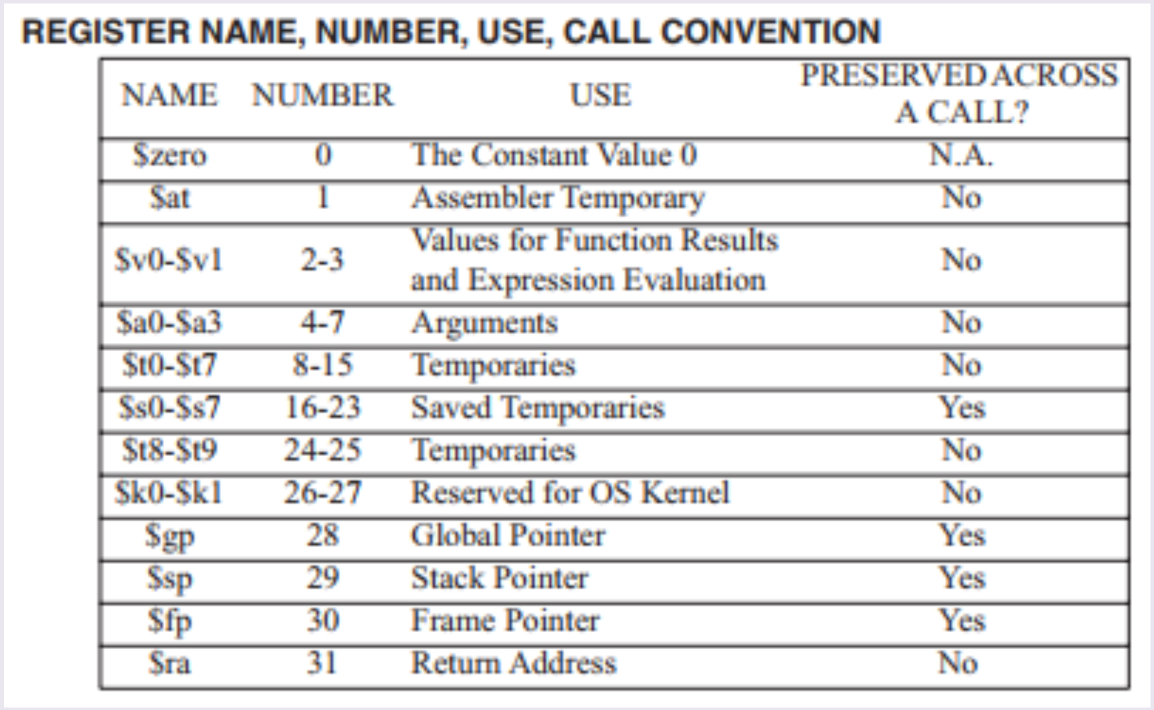
R	opcode	rs	rt	rd	shamt	funct
	31 26	25 21	20 16	15 11	10 6	5 0
I	opcode	rs	rt	immediate		
	31 26	25 21	20 16	15 0		
J	opcode	address				
	31 26	25 0				

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]



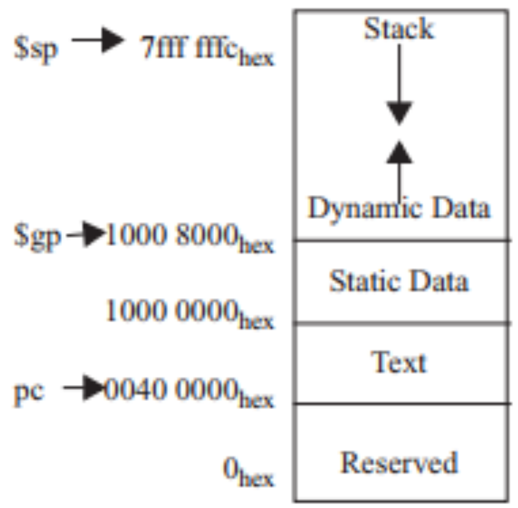
REGISTER NAME, NUMBER, USE, CALL CONVENTION

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

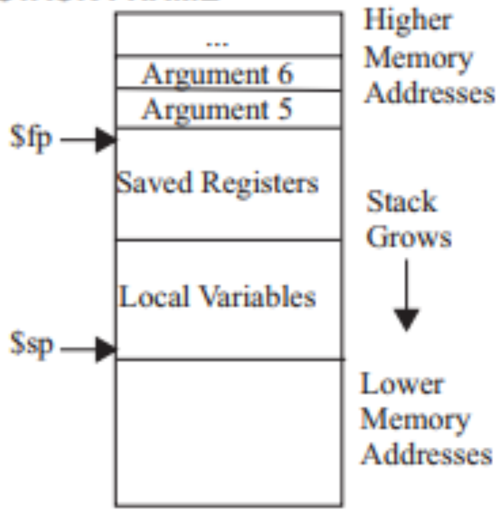
NOTE THE FOLLOWING:

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…

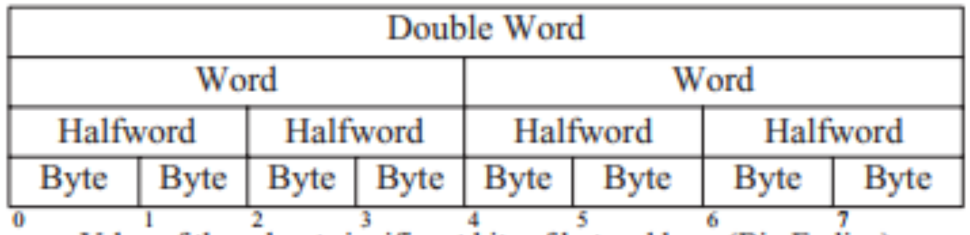
MEMORY ALLOCATION



STACK FRAME



DATA ALIGNMENT

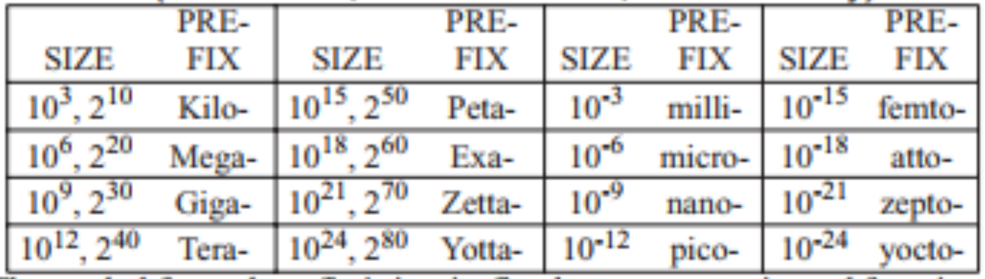


Value of three least significant bits of byte address (Big Endian)

NOTE THE FOLLOWING:

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

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.

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: *rd* *rs* *rt*

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

```
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					

YOUR TO-DOs

- Finish assignment/Lab #3
  - Assignment due on FRIDAY

- Prep for Lab #4 on Monday

</LECTURE>