
Lecture 3

Review: Signed Binary Representation

	2'sc binary	decimal
$-2^3 =$	1000	-8
$-(2^3 - 1) =$	1001	-7
	1010	-6
	1011	-5
	1100	-4
	1101	-3
	1110	-2
	1111	-1
	0000	0
	0001	1
	0010	2
	0011	3
	0100	4
$2^3 - 1 =$	0101	5
	0110	6
	0111	7

Review: MIPS Number Representations

- ❑ 32-bit signed numbers (2's complement):

0000 0000 0000 0000 0000 0000 0000 0000	= 0 _{ten}	
0000 0000 0000 0000 0000 0000 0000 0001	= + 1 _{ten}	<i>maxint</i>
...		
0111 1111 1111 1111 1111 1111 1111 1110	= + 2,147,483,646 _{ten}	
0111 1111 1111 1111 1111 1111 1111 1111	= + 2,147,483,647 _{ten}	<i>maxint</i>
1000 0000 0000 0000 0000 0000 0000 0000	= - 2,147,483,648 _{ten}	<i>minint</i>
1000 0000 0000 0000 0000 0000 0000 0001	= - 2,147,483,647 _{ten}	
...		
1111 1111 1111 1111 1111 1111 1111 1110	= - 2 _{ten}	
1111 1111 1111 1111 1111 1111 1111 1111	= - 1 _{ten}	

MSB (circled) (circled) LSB

- ❑ Converting <32 bit values into 32 bit values

- copy the most significant bit (the sign bit) into the "empty" bits

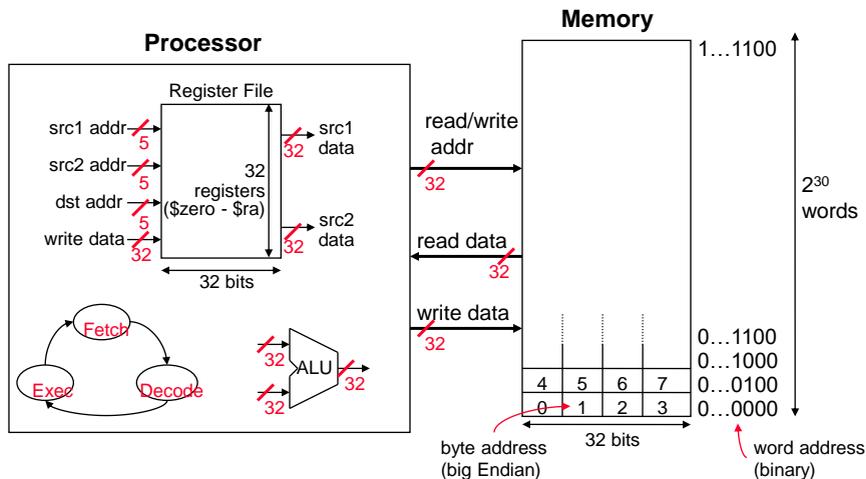
0010 -> 0000 0010

1010 -> 1111 1010

- sign extend versus zero extend

Review: MIPS Organization

- ❑ Arithmetic instructions – to/from the register file
- ❑ Load/store instructions – from/to memory



Review: MIPS Instructions, so far

Category	Instr	OpCode	Example	Meaning
Arithmetic (R format)	add	0 & 20	add \$s1, \$s2, \$s3	$\$s1 = \$s2 + \$s3$
	subtract	0 & 22	sub \$s1, \$s2, \$s3	$\$s1 = \$s2 - \$s3$
Arithmetic (I format)	add immediate	8	addi \$s1, \$s2, 4	$\$s1 = \$s2 + 4$
Data transfer (I format)	load word	23	lw \$s1, 100(\$s2)	$\$s1 = \text{Memory}(\$s2+100)$
	store word	2b	sw \$s1, 100(\$s2)	$\text{Memory}(\$s2+100) = \$s1$

↑
hex

Instructions for Making Decisions

- ❑ Decision making instructions
 - alter the control flow
 - i.e., change the "next" instruction to be executed
- ❑ MIPS **conditional branch** instructions:

```
bne $s0, $s1, Lbl    #go to Lbl if $s0≠$s1
beq $s0, $s1, Lbl    #go to Lbl if $s0=$s1
```

- ❑ **Example:**if (i==j) h = i + j;

```
        bne $s0, $s1, Lbl1
        add $s3, $s0, $s1
Lbl1:   ...
```

Assembling Branches

Instructions:

```
bne $s0, $s1, Lbl1 #go to Lbl1 if $s0≠$s1
beq $s0, $s1, Lbl1 #go to Lbl1 if $s0=$s1
```

Machine Formats:

op	rs	rt	16 bit number	I format
----	----	----	---------------	----------



5	16	17	????
---	----	----	------

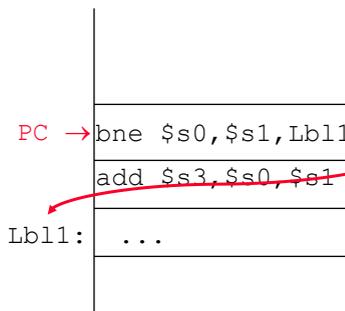
4	16	17	????
---	----	----	------

How is the branch destination address specified?

Specifying Branch Destinations

Could specify the memory address - but that would require a 32 bit field

Could use a “base” register and add to it the 16-bit offset



- which register?

- Instruction Address Register (PC = program counter) - its use is automatically implied by branch

- PC gets updated (PC+4) during the Fetch cycle so that it holds the address of the next instruction

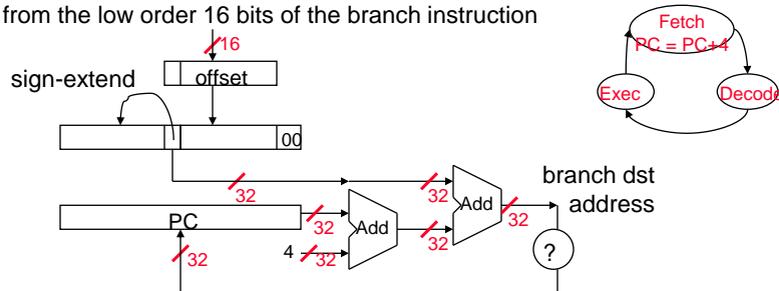
- limits the branch distance to -2^{15} to $+2^{15}-1$ instr's from the (instruction after the) branch

- but most branches are local anyway

Disassembling Branch Destinations

- ❑ The contents of the updated PC (PC+4) is added to the 16 bit branch offset which is converted into a 32 bit value by
 - concatenating two low-order zeros to make it a word address and then sign-extending those 18 bits
- ❑ The result is written into the PC if the branch condition is true - before the next Fetch cycle

from the low order 16 bits of the branch instruction



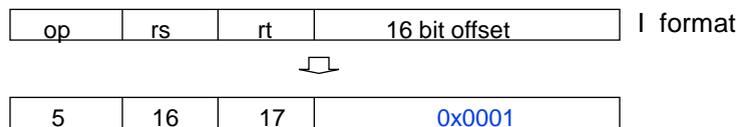
Assembling Branches Example

- ❑ Assembly code

```
bne $s0, $s1, Lb11
add $s3, $s0, $s1
```

```
Lb11: ...
```

- ❑ Machine Format of `bne`:



- ❑ Remember

- After the `bne` instruction is fetched, the PC is updated so that it is addressing the `add` instruction (PC = PC + 4).
- The offset (plus 2 low-order zeros) is sign-extended and added to the (updated) PC

Another Instruction for Changing Flow

- ❑ MIPS also has an **unconditional branch** instruction or **jump** instruction:

```
j Lbl          #go to Lbl
```

- ❑ Example:

```
if (i!=j)
    h=i+j;
else
    h=i-j;
```

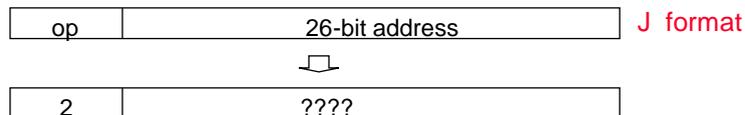
```
        beq $s0, $s1, Else
        add $s3, $s0, $s1
        j   Exit
Else:   sub $s3, $s0, $s1
Exit:   ...
```

Assembling Jumps

- ❑ Instruction:

```
j Lbl          #go to Lbl
```

- ❑ Machine Format:



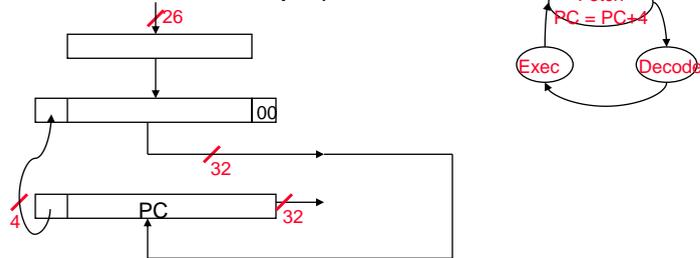
- ❑ How is the jump destination address specified?

- As an absolute address formed by
 - concatenating 00 as the 2 low-order bits to make it a word address
 - concatenating the upper 4 bits of the current PC (now PC+4)

Disassembling Jump Destinations

- The low order 26 bits of the jump instr converted into a 32 bit jump destination address by
 - concatenating two low-order zeros to create an 28 bit (word) address and then concatenating the upper 4 bits of the current PC (now PC+4) to create a 32 bit (word) address
- that is put into the PC prior to the next Fetch cycle

from the low order 26 bits of the jump instruction



Assembling Branches and Jumps

- Assemble the MIPS machine code (in decimal is fine) for the following code sequence. Assume that the addr of the `beq` instr is `0x00400020hex`

```

        beq  $s0, $s1, Else
        add  $s3, $s0, $s1
        j    Exit
Else:   sub  $s3, $s0, $s1
Exit:   ...

```

```

0x00400020          4      16      17          2
0x00400024          0      16      17      19      0  0x20
0x00400028          2      0000  0100  0 ... 0  0011  002
                    jmp dst = (0x0) 0x040003 002(002)
                    = 0x00400030
0x0040002c          0      16      17      19      0  0x22
0x00400030          ...

```

Compiling While Loops

- Compile the assembly code for the C `while` loop where `i` is in `$s0`, `j` is in `$s1`, and `k` is in `$s2`

```
while (i!=k)
    i=i+j;

Loop:   beq  $s0, $s2, Exit
        add $s0, $s0, $s1
        j   Loop
Exit:   . . .
```

- **Basic block** – A sequence of instructions without branches (except at the end) and without branch targets (except at the beginning)

Branching Far Away

- What if the branch destination is further away than can be captured in 16 bits?
- The assembler comes to the rescue – it inserts an unconditional jump to the branch target and inverts the condition

```
beq  $s0, $s1, L1
```

becomes

```
    bne  $s0, $s1, L2
    j    L1
L2:
```

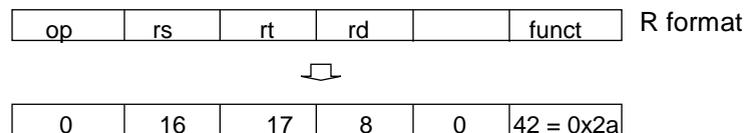
More Instructions for Making Decisions

- ❑ We have `beq`, `bne`, but what about branch-if-less-than?

- ❑ New instruction:

```
slt $t0, $s0, $s1    # if $s0 < $s1
                    #   then
                    # $t0 = 1
                    #   else
                    # $t0 = 0
```

- ❑ Machine format:



2

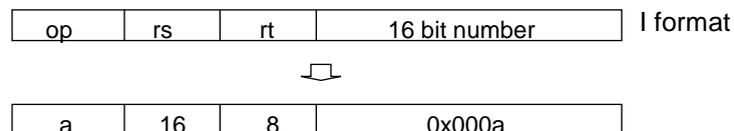
Yet More Instructions for Making Decisions

- ❑ Since constant operands are popular in comparisons, also have `slti`

- ❑ New instruction:

```
slti $t0, $s0, 10   # if $s0 < 10
                    #   then
                    # $t0 = 1
                    #   else
                    # $t0 = 0
```

- ❑ Machine format:



2

Other Branch Instructions

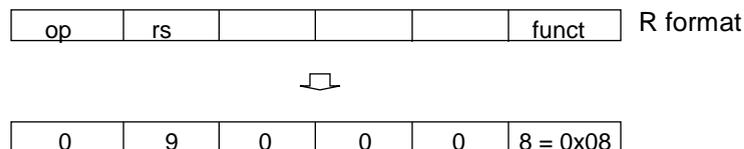
- ❑ Can use `slt`, `beq`, `bne`, and the fixed value of 0 in `$zero` to **create** all relative conditions
 - less than `blt $s1, $s2, Lbl`
`slt $at, $s1, $s2` #`$at` set to 1 if
`bne $at, $zero, Lbl` # `$s1 < $s2`
 - less than or equal to `ble $s1, $s2, Lbl`
 - greater than `bgt $s1, $s2, Lbl`
 - great than or equal to `bge $s1, $s2, Lbl`
- ❑ As pseudo instructions they are recognized (and expanded) by the assembler
- ❑ The assembler needs a reserved register (`$at`)
 - so there are policy of use conventions for registers

Another Instruction for Changing Flow

- ❑ Most higher level languages have `case` or `switch` statements allowing the code to select one of many alternatives depending on a single value
- ❑ Instruction:

```
jr $t1 #go to address in $t1
```

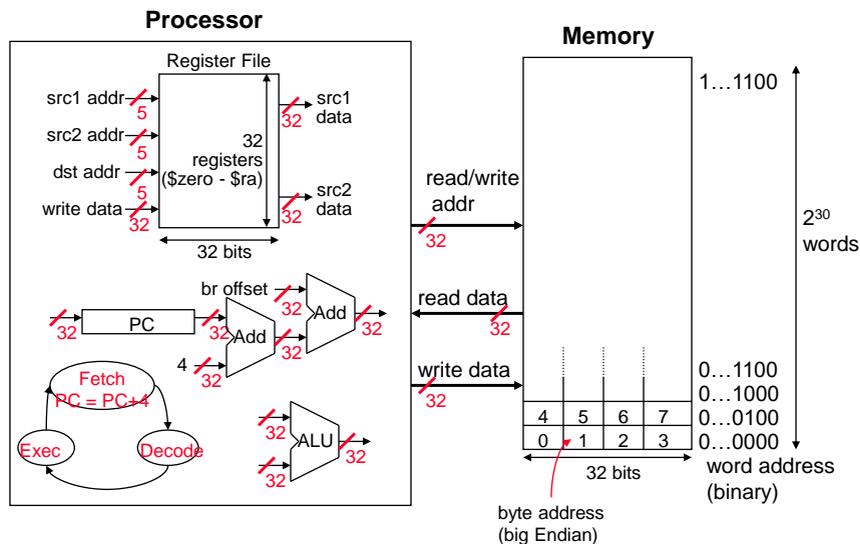
- ❑ Machine format:



Review: MIPS Instructions, so far

Category	Instr	OpCd	Example	Meaning
Arithmetic (R & I format)	add	0 & 20	add \$s1, \$s2, \$s3	\$s1 = \$s2 + \$s3
	subtract	0 & 22	sub \$s1, \$s2, \$s3	\$s1 = \$s2 - \$s3
	add immediate	8	addi \$s1, \$s2, 4	\$s1 = \$s2 + 4
Data transfer (I format)	load word	23	lw \$s1, 100(\$s2)	\$s1 = Memory(\$s2+100)
	store word	2b	sw \$s1, 100(\$s2)	Memory(\$s2+100) = \$s1
Cond. branch (I format)	br on equal	4	beq \$s1, \$s2, L	if (\$s1==\$s2) go to L
	br on not equal	5	bne \$s1, \$s2, L	if (\$s1!=\$s2) go to L
	set on less than immediate	a	slt \$s1, \$s2, 100	if (\$s2<100) \$s1=1; else \$s1=0
(R format)	set on less than	0 & 2a	slti \$s1, \$s2, \$s3	if (\$s2<\$s3) \$s1=1; else \$s1=0
Uncond. jump	jump	2	j 2500	go to 10000
	jump register	0 & 08	jr \$t1	go to \$t1

MIPS Organization



MIPS Data Types

Bit: 0, 1

Bit String: sequence of bits of a particular length

4 bits is a nibble

8 bits is a byte

16 bits is a half-word

32 bits is a word

64 bits is a double-word

Character:

ASCII 7 bit code

Decimal:

digits 0-9 encoded as 0000_2 thru 1001_2

two decimal digits packed per 8 bit byte

Integers: 2's complement

Floating Point

Beyond Numbers

- Most computers use 8-bit bytes to represent characters with the American Std Code for Info Interchange (ASCII)

ASCII	Char	ASCII	Char	ASCII	Char	ASCII	Char	ASCII	Char	ASCII	Char
0	Null	32	space	48	0	64	@	96	`	112	p
1		33	!	49	1	65	A	97	a	113	q
2		34	"	50	2	66	B	98	b	114	r
3		35	#	51	3	67	C	99	c	115	s
4	EOT	36	\$	52	4	68	D	100	d	116	t
5		37	%	53	5	69	E	101	e	117	u
6	ACK	38	&	54	6	70	F	102	f	118	v
7		39	'	55	7	71	G	103	g	119	w
8	bksp	40	(56	8	72	H	104	h	120	x
9	tab	41)	57	9	73	I	105	i	121	y
10	LF	42	*	58	:	74	J	106	j	122	z
11		43	+	59	;	75	K	107	k	123	{
12	FF	44	,	60	<	76	L	108	l	124	
15		47	/	63	?	79	O	111	o	127	DEL

- So, we need instructions to move bytes around

Byte Addresses

- Since **bytes** (8 bits) are so useful, most ISAs support addressing individual bytes in memory
- Therefore, the memory address of a **word** must be a multiple of 4 (**alignment restriction**)

□ **Big Endian:** leftmost byte is word address

MIPS

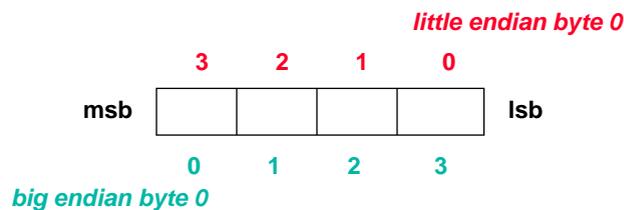
□ **Little Endian:** rightmost byte is word address

Intel 80x86

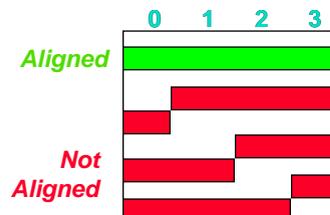
Addressing Objects: Endianness and Alignment

□ **Big Endian:** leftmost byte is word address

□ **Little Endian:** rightmost byte is word address



Alignment restriction: requires that objects fall on address that is multiple of their size



Loading and Storing Bytes

- ❑ MIPS provides special instructions to move bytes

```
lb    $t0, 1($s3)  #load byte from memory
sb    $t0, 6($s3)  #store byte to memory
```

op	rs	rt	16 bit number
----	----	----	---------------

- ❑ What 8 bits get loaded and stored?

- load byte places the byte from memory in the rightmost 8 bits of the destination register
 - what happens to the other bits in the register?
- store byte takes the byte from the rightmost 8 bits of a register and writes it to the byte in memory
 - leaving the other bytes in the memory word unchanged

Example of Loading and Storing Bytes

- ❑ Given following code sequence and memory state what is the state of the memory after executing the code?

```
add  $s3, $zero, $zero
lb   $t0, 1($s3)
sb   $t0, 6($s3)  ❑ What value is left in $t0?
```

Memory	
0x00000000	24
0x00000000	20
0x00000000	16
0x10000010	12
0x01000402	8
0xFFFFFFFF	4
0x009012A0	0

Data

Word
Address (Decimal)

$\$t0 = 0x00000090$

- ❑ What word is changed in Memory and to what?

$mem(4) = 0xFFFF90FF$

- ❑ What if the machine was **little Endian**?

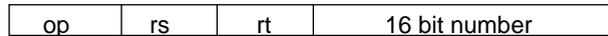
$\$t0 = 0x00000012$

$mem(4) = 0xFF12FFFF$

Loading and Storing Half Words

- ❑ MIPS also provides special instructions to move half words

```
lh    $t0, 1($s3)  #load half word from memory
sh    $t0, 6($s3)  #store half word to memory
```

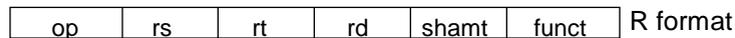


- ❑ What 16 bits get loaded and stored?
 - load half word places the half word from memory in the rightmost 16 bits of the destination register
 - what happens to the other bits in the register?
 - store half word takes the half word from the rightmost 16 bits of the register and writes it to the half word in memory
 - leaving the other half word in the memory word unchanged

Shift Operations

- ❑ Need operations to **pack** and **unpack** 8-bit characters into 32-bit words
- ❑ Shifts move all the bits in a word left or right

```
sll  $t2, $s0, 8    # $t2 = $s0 << 8 bits
srl  $t2, $s0, 8    # $t2 = $s0 >> 8 bits
```

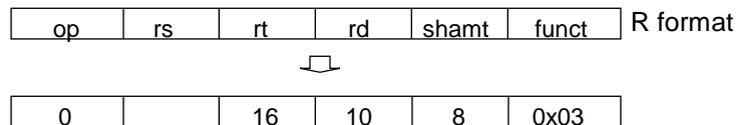


- ❑ Such shifts are called **logical** because they fill with **zeros**
 - Notice that a 5-bit shamt field is enough to shift a 32-bit value $2^5 - 1$ or **31 bit positions**

More Shift Operations

- An arithmetic shift (`sra`) maintain the arithmetic correctness of the shifted value (i.e., a number shifted right one bit should be $\frac{1}{2}$ of its original value; a number shifted left should be 2 times its original value)
 - `sra` uses the most significant bit (sign bit) as the bit shifted in
 - `sll` works for arithmetic left shifts for 2's compl. (so there is no need for a `sla`)

```
sra $t2, $s0, 8    # $t2 = $s0 >> 8 bits
```



-
- Give a specific numerical example (e.g., 6 and -6) illustrating the difference between `sll`, `srl`, and `sra` (and how 6 becomes 3, etc.)

Compiling Another While Loop

- Compile the assembly code for the C `while` loop where `i` is in `$s3`, `k` is in `$s5`, and the base address of the array `save` is in `$s6`

```
while (save[i] == k)
    i += 1;
```

```
Loop:    sll  $t1, $s3, 2
         add  $t1, $t1, $s6
         lw   $t0, 0($t1)
         bne  $t0, $s5, Exit
         addi $s3, $s3, 1
         j    Loop
Exit:    . . .
```

Logical Operations

- There are a number of **bit-wise** logical operations in the MIPS ISA

```
and $t0, $t1, $t2  # $t0 = $t1 & $t2
```

```
or  $t0, $t1, $t2  # $t0 = $t1 | $t2
```

```
nor $t0, $t1, $t2  # $t0 = not($t1 | $t2)
```

op	rs	rt	rd	shamt	funct	R format
----	----	----	----	-------	-------	----------



0	9	10	8		0x24
---	---	----	---	--	------

```
andi $t0, $t1, 0xff00  # $t0 = $t1 & ff00
```

```
ori  $t0, $t1, 0xff00  # $t0 = $t1 | ff00
```

Logic Operations

- Logic operations operate on individual bits of the operand.

```
          $t2 = 0...0 0000 1101 0000
          $t1 = 0...0 0011 1100 0000
and  $t0, $t1, $t2 $t0 = 0...0 0000 1100 0000

or   $t0, $t1 $t2 $t0 = 0...0 0011 1101 0000

nor  $t0, $t1, $t2 $t0 = 1...1 1100 0010 1111
```

How About Larger Constants?

- We'd also like to be able to load a 32-bit constant into a register
- Must use two instructions, new "load upper immediate" instruction

```
lui $t0, 0xaaaa
```

f	0	8	1010101010101010
---	---	---	------------------

- Then must get the lower order bits right, i.e.,

```
ori $t0, $t0, 0xaaaa
```

1010101010101010	0000000000000000
------------------	------------------

0000000000000000	1010101010101010
------------------	------------------

1010101010101010	1010101010101010
------------------	------------------

Review: MIPS Instructions, so far

Category	Instr	OpC	Example	Meaning
Arithmetic (R & I format)	add	0 & 20	add \$s1, \$s2, \$s3	\$s1 = \$s2 + \$s3
	subtract	0 & 22	sub \$s1, \$s2, \$s3	\$s1 = \$s2 - \$s3
	add immediate	8	addi \$s1, \$s2, 4	\$s1 = \$s2 + 4
	shift left logical	0 & 00	sll \$s1, \$s2, 4	\$s1 = \$s2 << 4
	shift right logical	0 & 02	srl \$s1, \$s2, 4	\$s1 = \$s2 >> 4 (fill with zeros)
	shift right arithmetic	0 & 03	sra \$s1, \$s2, 4	\$s1 = \$s2 >> 4 (fill with sign bit)
	and	0 & 24	and \$s1, \$s2, \$s3	\$s1 = \$s2 & \$s3
	or	0 & 25	or \$s1, \$s2, \$s3	\$s1 = \$s2 \$s3
	nor	0 & 27	nor \$s1, \$s2, \$s3	\$s1 = not (\$s2 \$s3)
	and immediate	c	and \$s1, \$s2, ff00	\$s1 = \$s2 & 0xff00
	or immediate	d	or \$s1, \$s2, ff00	\$s1 = \$s2 0xff00
	load upper immediate	f	lui \$s1, 0xffff	\$s1 = 0xffff0000

Review: MIPS Instructions, so far

Category	Instr	OpC	Example	Meaning
Data transfer (I format)	load word	23	lw \$s1, 100(\$s2)	\$s1 = Memory(\$s2+100)
	store word	2b	sw \$s1, 100(\$s2)	Memory(\$s2+100) = \$s1
	load byte	20	lb \$s1, 101(\$s2)	\$s1 = Memory(\$s2+101)
	store byte	28	sb \$s1, 101(\$s2)	Memory(\$s2+101) = \$s1
	load half	21	lh \$s1, 101(\$s2)	\$s1 = Memory(\$s2+102)
	store half	29	sh \$s1, 101(\$s2)	Memory(\$s2+102) = \$s1
Cond. branch (I & R format)	br on equal	4	beq \$s1, \$s2, L	if (\$s1==\$s2) go to L
	br on not equal	5	bne \$s1, \$s2, L	if (\$s1!=\$s2) go to L
	set on less than immediate	a	sli \$s1, \$s2, 100	if (\$s2<100) \$s1=1; else \$s1=0
	set on less than	0 & 2a	slt \$s1, \$s2, \$s3	if (\$s2<\$s3) \$s1=1; else \$s1=0
Uncond. jump	jump	2	j 2500	go to 10000
	jump register	0 & 08	jr \$t1	go to \$t1