Computer Organization & Architecture

Lecture 02: MIPS ISA Review

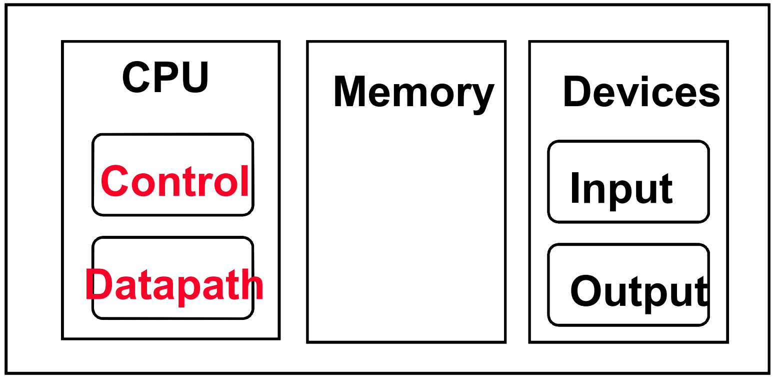
Slides are by Mary Jane Irwin

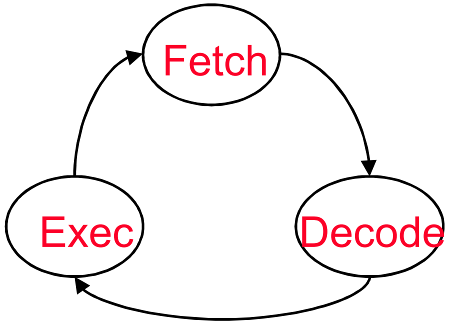
[Adapted from *Computer Organization and Design*,

Patterson & Hennessy, © 2005, UCB]

(vonNeumann) Processor Organization

- Control needs to
  1. input instructions from Memory
  2. issue signals to control the information flow between the Datapath components and to control what operations they perform
  3. control instruction sequencing





- Datapath needs to have the
  - components – the functional units and storage (e.g., register file) needed to execute instructions
  - interconnects - components connected so that the instructions can be accomplished and so that data can be loaded from and stored to Memory

For a given level of function, however, that system is best in which one can specify things with the most simplicity and straightforwardness. … Simplicity and straightforwardness proceed from conceptual integrity. … Ease of use, then, dictates unity of design, conceptual integrity.

The Mythical Man-Month, Brooks, pg 44

RISC - Reduced Instruction Set Computer

- RISC philosophy
  - fixed instruction lengths
  - load-store instruction sets
  - limited addressing modes
  - limited operations
- MIPS, Sun SPARC, HP PA-RISC, IBM PowerPC, Intel (Compaq) Alpha, …
- Instruction sets are measured by how well compilers use them as opposed to how well assembly language programmers use them

Design goals: speed, cost (design, fabrication, test, packaging), size, power consumption, reliability, memory space (embedded systems)

MIPS R3000 Instruction Set Architecture (ISA)

- Instruction Categories
  - Computational
  - Load/Store
  - Jump and Branch
  - Floating Point
    - coprocessor
  - Memory Management
  - Special

Registers

Registers
R0 - R31
PC
HI
LO

3 Instruction Formats: all 32 bits wide

OP	**rs**	**rt**	**rd**	**sa**	**funct**	R format
OP	**rs**	**rt**	**immediate**			I format
OP	**jump target**					J format

Review: Unsigned Binary Representation

Hex	Binary	Decimal
0x00000000	0…0000	0
0x00000001	0…0001	1
0x00000002	0…0010	2
0x00000003	0…0011	3
0x00000004	0…0100	4
0x00000005	0…0101	5
0x00000006	0…0110	6
0x00000007	0…0111	7
0x00000008	0…1000	8
0x00000009	0…1001	9
	…	
0xFFFFFFFC	1…1100	2^{32} - 4
0xFFFFFFFD	1…1101	2^{32} - 3
0xFFFFFFFE	1…1110	2^{32} - 2
0xFFFFFFFF	1…1111	2^{32} - 1

2^{31}	2^{30}	2^{29}	. . .	2^{3}	2^{2}	2^{1}	2^{0}	bit weight
31	30	29	. . .	3	2	1	0	bit position
1	**1**	**1**	**. . .**	**1**	**1**	**1**	**1**	**bit**

1 0 0 0 . . . 0 0 0 0 - 1

$\mathbf{2^{32} - 1}$

Aside: Beyond Numbers

- American Std Code for Info Interchange (ASCII): 8-bit bytes representing characters

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

MIPS Arithmetic Instructions

- MIPS assembly language arithmetic statement

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

- Each arithmetic instruction performs only one operation
- Each arithmetic instruction fits in 32 bits and specifies exactly three operands

  destination $\leftarrow$ source1 op source2

- Those operands are all contained in the datapath's register file (`$t0,$s1,$s2`) – indicated by $
- Operand order is fixed (destination first)

MIPS Arithmetic Instructions

- MIPS assembly language arithmetic statement

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

- Each arithmetic instruction performs only one operation

- Each arithmetic instruction fits in 32 bits and specifies exactly three operands

destination ← source1 op source2

- Operand order is fixed (destination first)

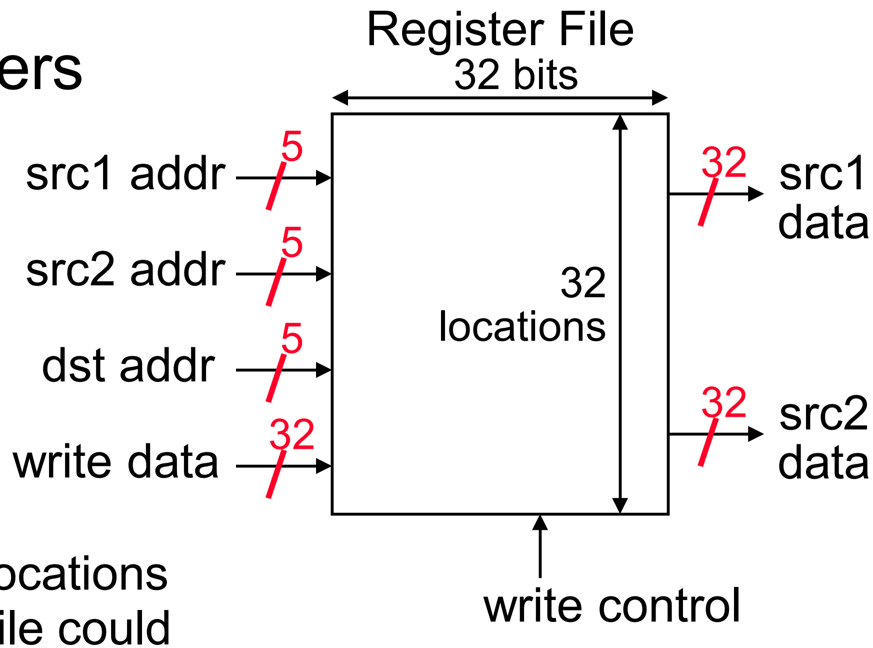
- Those operands are all contained in the datapath's register file (`$t0,$s1,$s2`) – indicated by $

Aside: MIPS Register Convention

Name	Register Number	Usage	Preserve on call?
$zero	0	constant 0 (hardware)	n.a.
$at	1	reserved for assembler	n.a.
$v0 - $v1	2-3	returned values	no
$a0 - $a3	4-7	arguments	yes
$t0 - $t7	8-15	temporaries	no
$s0 - $s7	16-23	saved values	yes
$t8 - $t9	24-25	temporaries	no
$gp	28	global pointer	yes
$sp	29	stack pointer	yes
$fp	30	frame pointer	yes
$ra	31	return addr (hardware)	yes

MIPS Register File

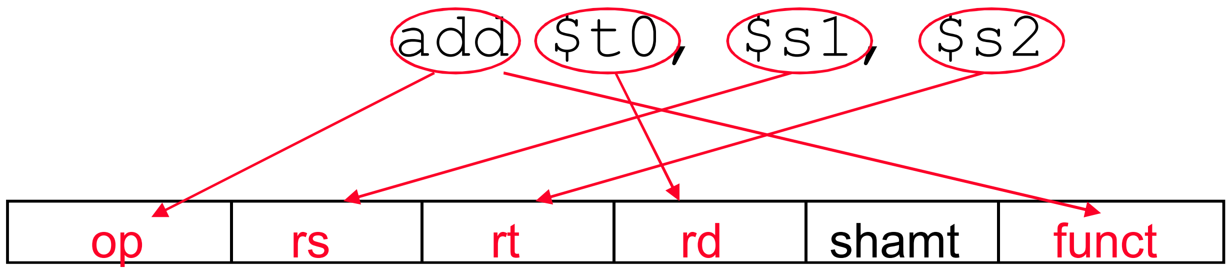
- Holds thirty-two 32-bit registers
  - Two read ports and
  - One write port



- Registers are
  - Faster than main memory
    - But register files with more locations are slower (e.g., a 64 word file could be as much as 50% slower than a 32 word file)
    - Read/write port increase impacts speed quadratically
  - Easier for a compiler to use
    - e.g., (A*B) – (C*D) – (E*F) can do multiplies in any order vs. stack
  - Can hold variables so that
    - code density improves (since register are named with fewer bits than a memory location)

Machine Language - Add Instruction

- Instructions, like registers and words of data, are 32 bits long
- Arithmetic Instruction Format (R format):



op	6-bits	opcode that specifies the operation
rs	5-bits	register file address of the first source operand
rt	5-bits	register file address of the second source operand
rd	5-bits	register file address of the result's destination
shamt	5-bits	shift amount (for shift instructions)
funct	6-bits	function code augmenting the opcode

MIPS Memory Access Instructions

- MIPS has two basic data transfer instructions for accessing memory

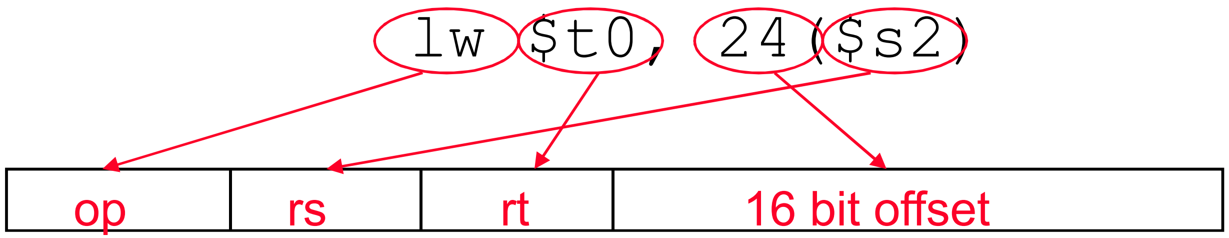
```
lw    $t0, 4($s3)   #load word from memory
sw    $t0, 8($s3)   #store word to memory
```

- The data is loaded into (lw) or stored from (sw) a register in the register file – a 5 bit address

- The memory address – a 32 bit address – is formed by adding the contents of the base address register to the offset value
  - A 16-bit field meaning access is limited to memory locations within a region of $\pm 2^{13}$ or 8,192 words ($\pm 2^{15}$ or 32,768 bytes) of the address in the base register
  - Note that the offset can be positive or negative

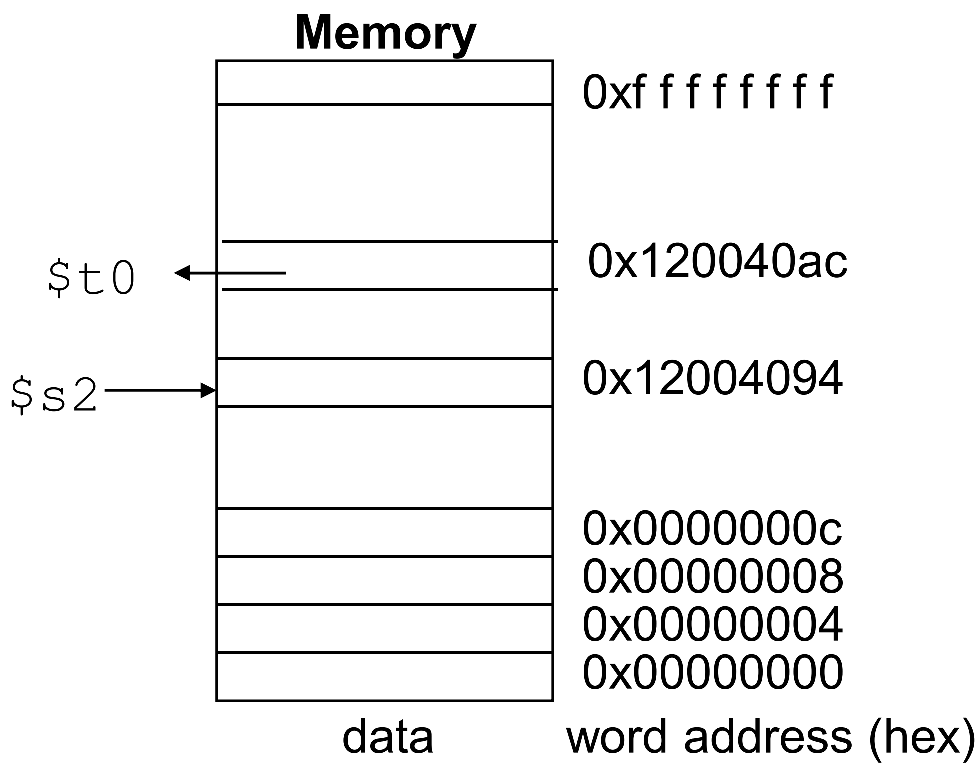
Machine Language - Load Instruction

- Load/Store Instruction Format (I format):



24_{10} + $s2 =

$$\begin{array}{r} \ldots\ 0001\ 1000 \\ +\ \underline{\ldots\ 1001\ 0100} \\ \ldots\ 1010\ 1100 = \\ \texttt{0x120040ac} \end{array}$$



Byte Addresses

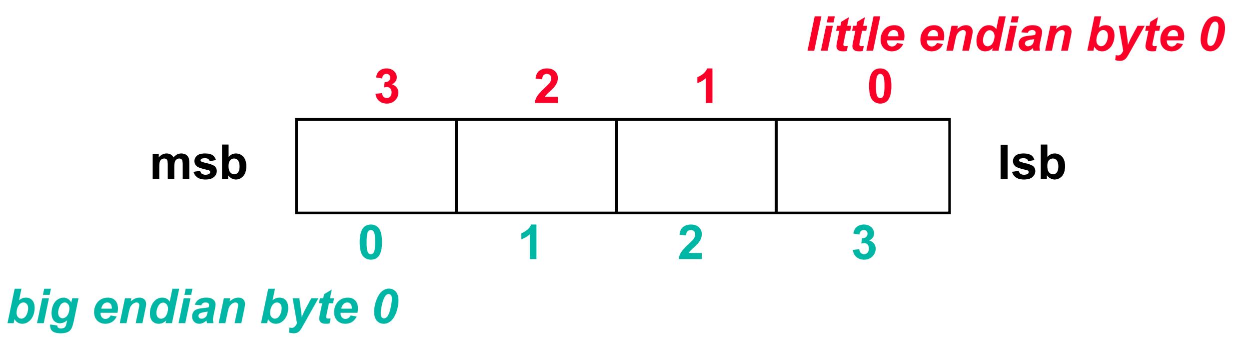
- Since 8-bit bytes are so useful, most architectures address individual bytes in memory
  - The memory address of a word must be a multiple of 4 (alignment restriction)

- Big Endian: leftmost byte is word address

  IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA

- Little Endian: rightmost byte is word address

  Intel 80x86, DEC Vax, DEC Alpha (Windows NT)



Aside: 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 offset

- 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 a byte in memory
    - what happens to the other bits in the memory word?

MIPS Control Flow Instructions

- MIPS conditional branch instructions:

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

- Ex: `if (i==j) h = i + j;`

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

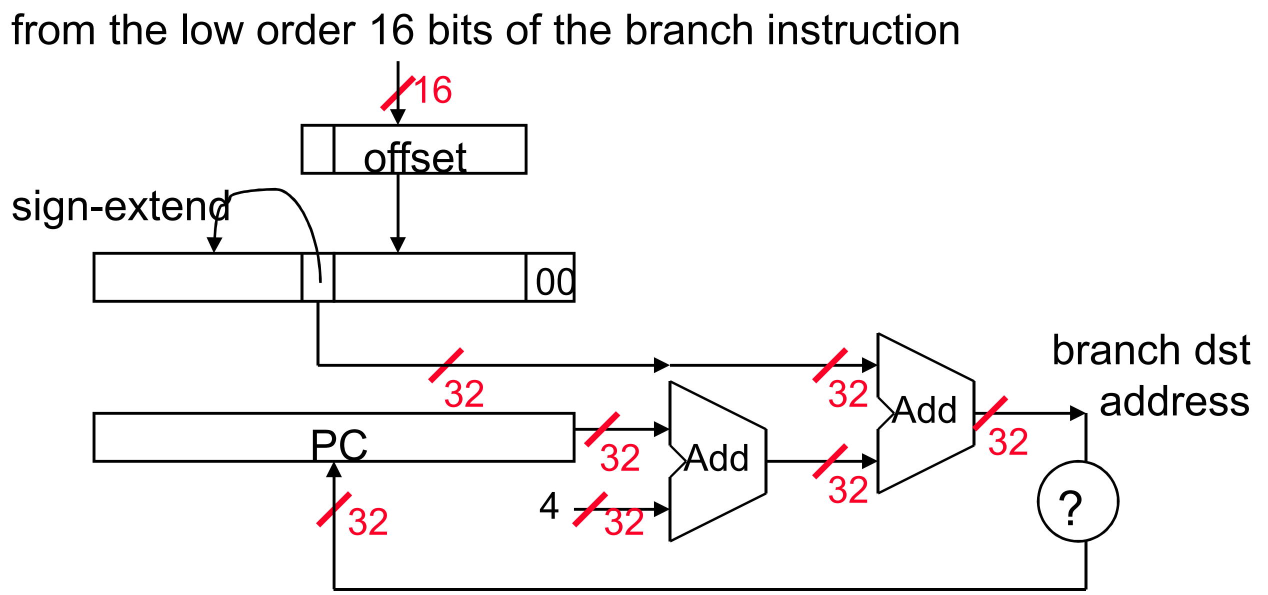
- Instruction Format (I format):

op	rs	rt	16 bit offset

- How is the branch destination address specified?

Specifying Branch Destinations

- Use a register (like in lw and sw) added to the 16-bit offset
  - which register? Instruction Address Register (the PC)
    - its use is automatically implied by instruction
    - 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 $+2^{15}-1$ instructions from the (instruction after the) branch instruction, but most branches are local anyway



More Branch Instructions

- We have `beq`, `bne`, but what about other kinds of brances (e.g., branch-if-less-than)? For this, we need yet another instruction, `slt`

- Set on less than instruction:

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

- Instruction format (R format):

op	rs	rt	rd		funct

More Branch Instructions, Con't

- Can use `slt`, `beq`, `bne`, and the fixed value of 0 in register `$zero` to create other conditions
  - less than `blt $s1, $s2, Label`

```
slt  $at, $s1, $s2       #$at set to 1 if
bne  $at, $zero, Label   #  $s1 < $s2
```

  - less than or equal to `ble $s1, $s2, Label`
  - greater than `bgt $s1, $s2, Label`
  - great than or equal to `bge $s1, $s2, Label`

- Such branches are included in the instruction set as pseudo instructions - recognized (and expanded) by the assembler
  - Its why the assembler needs a reserved register (`$at`)

Other Control Flow Instructions

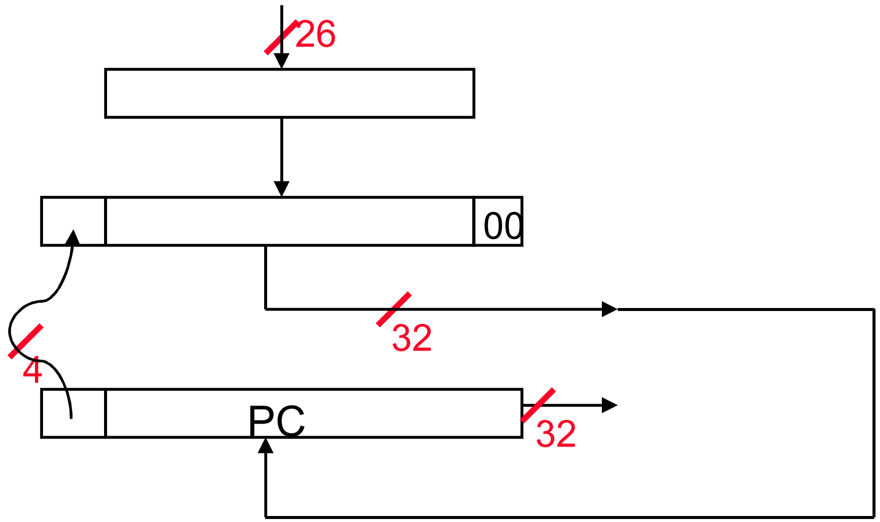
- MIPS also has an unconditional branch instruction or jump instruction:

```
j   label         #go to label
```

- Instruction Format (J Format):

op	26-bit address

from the low order 26 bits of the jump instruction



Aside: 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:
```

Instructions for Accessing Procedures

- MIPS procedure call instruction:

```
jal   ProcedureAddress      #jump and link
```

- Saves PC+4 in register $ra to have a link to the next instruction for the procedure return

- Machine format (J format):

op	26 bit address

- Then can do procedure return with a

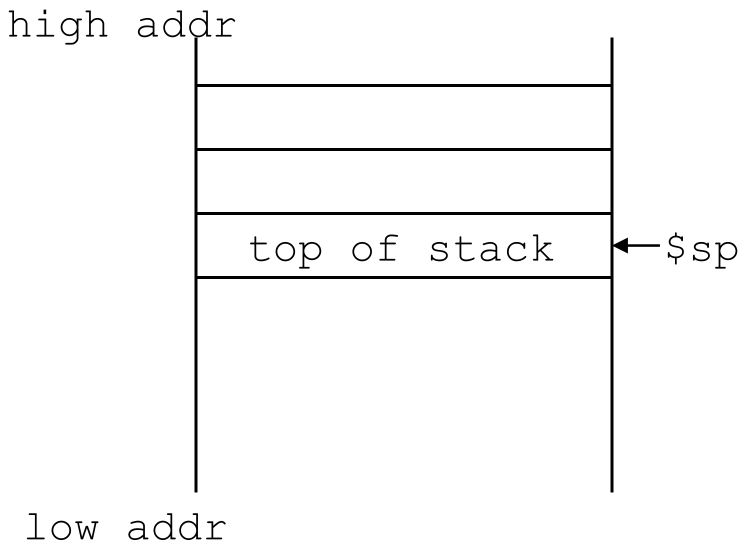
```
jr    $ra                   #return
```

- Instruction format (R format):

op	rs				funct

Aside: Spilling Registers

- What if the callee needs more registers? What if the procedure is recursive?
  - uses a stack – a last-in-first-out queue – in memory for passing additional values or saving (recursive) return address(es)



- One of the general registers, $sp, is used to address the stack (which “grows” from high address to low address)
  - add data onto the stack – push

    $sp = $sp – 4
    data on stack at new $sp

  - remove data from the stack – pop

    data from stack at $sp
    $sp = $sp + 4

MIPS Immediate Instructions

- Small constants are used often in typical code
- Possible approaches?
  - put “typical constants” in memory and load them
  - create hard-wired registers (like $zero) for constants like 1
  - have special instructions that contain constants !

```
addi  $sp, $sp, 4      #$sp = $sp + 4
slti  $t0, $s2, 15     #$t0 = 1 if $s2<15
```

- Machine format (I format):

op	rs	rt	16 bit immediate

I format

- The constant is kept inside the instruction itself!
  - Immediate format limits values to the range $+2^{15}-1$ to -2^{15}

Aside: How About Larger Constants?

- We'd also like to be able to load a 32 bit constant into a register, for this we must use two instructions

- a new "load upper immediate" instruction

```
lui $t0, 1010101010101010
```

16	0	8	1010101010101010

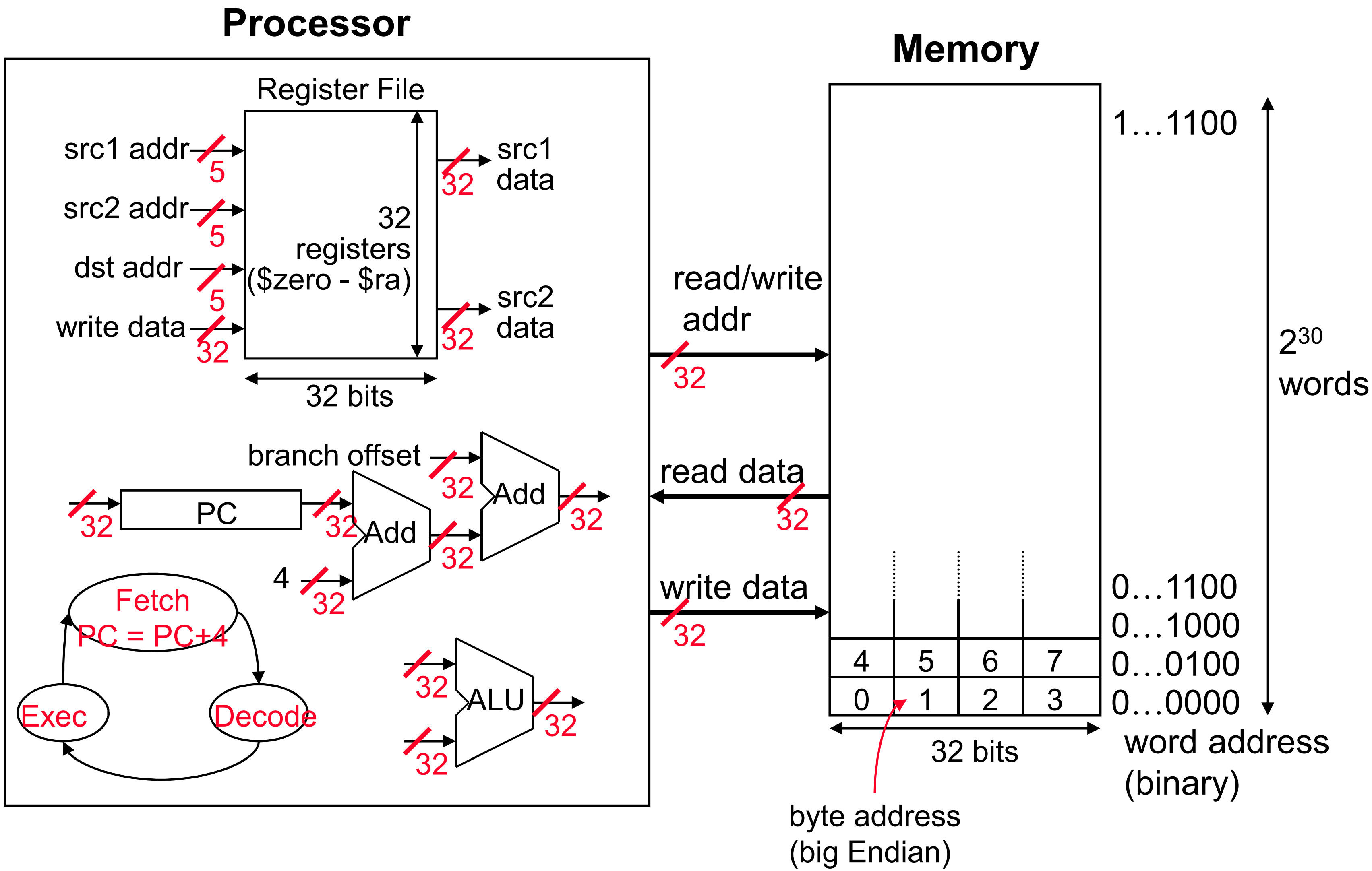
- Then must get the lower order bits right, use

```
ori $t0, $t0, 1010101010101010
```

1010101010101010	0000000000000000
0000000000000000	1010101010101010

1010101010101010	1010101010101010

MIPS Organization So Far

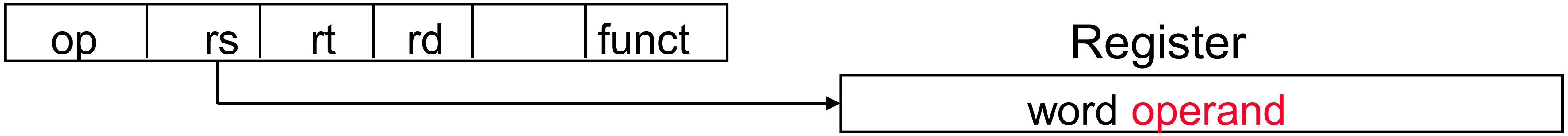


MIPS ISA So Far

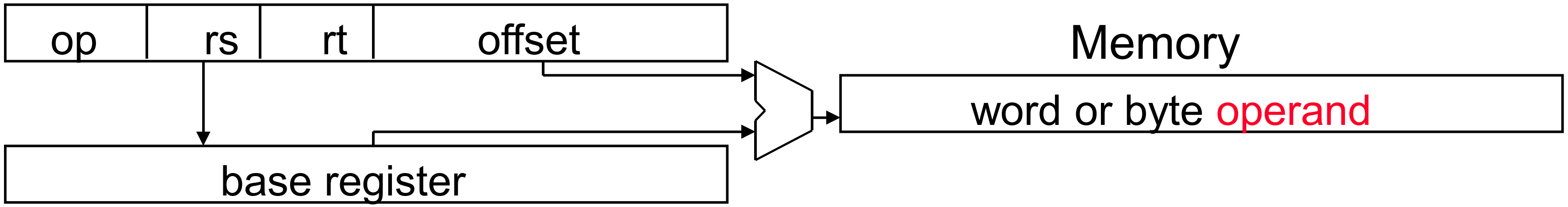
Category	Instr	Op Code	Example	Meaning
Arithmetic (R & I format)	add	0 and 32	add $s1, $s2, $s3	$s1 = $s2 + $s3
	subtract	0 and 34	sub $s1, $s2, $s3	$s1 = $s2 - $s3
	add immediate	8	addi $s1, $s2, 6	$s1 = $s2 + 6
	or immediate	13	ori $s1, $s2, 6	$s1 = $s2 v 6
Data Transfer (I format)	load word	35	lw $s1, 24($s2)	$s1 = Memory($s2+24)
	store word	43	sw $s1, 24($s2)	Memory($s2+24) = $s1
	load byte	32	lb $s1, 25($s2)	$s1 = Memory($s2+25)
	store byte	40	sb $s1, 25($s2)	Memory($s2+25) = $s1
	load upper imm	15	lui $s1, 6	$s1 = 6 * 2^{16}
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	0 and 42	slt $s1, $s2, $s3	if ($s2<$s3) $s1=1 else $s1=0
	set on less than immediate	10	slti $s1, $s2, 6	if ($s2<6) $s1=1 else $s1=0
Uncond. Jump (J & R format)	jump	2	j 2500	go to 10000
	jump register	0 and 8	jr $t1	go to $t1
	jump and link	3	jal 2500	go to 10000; $ra=PC+4

Review of MIPS Operand Addressing Modes

- Register addressing – operand is in a register

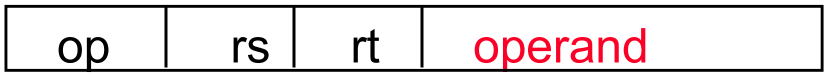


- Base (displacement) addressing – operand is at the memory location whose address is the sum of a register and a 16-bit constant contained within the instruction



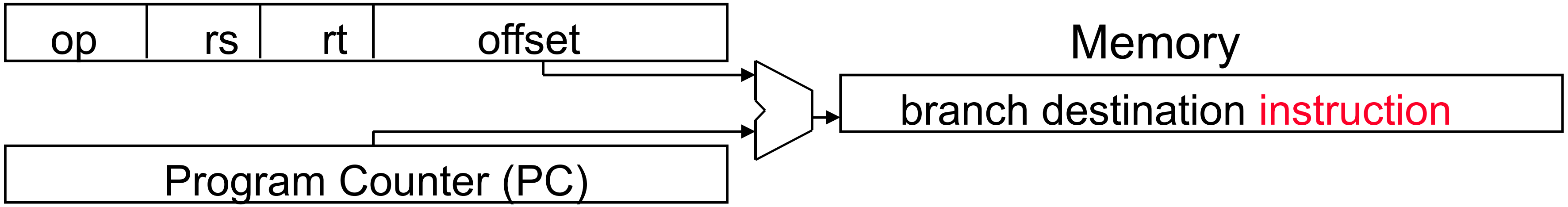
  - Register relative (indirect) with 0($a0)
  - Pseudo-direct with addr($zero)

- Immediate addressing – operand is a 16-bit constant contained within the instruction

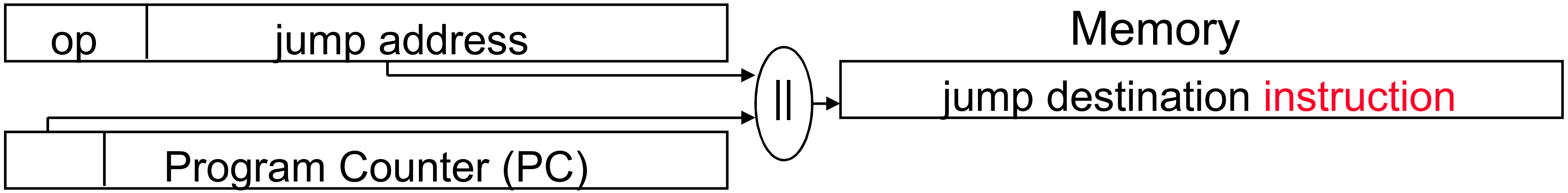


Review of MIPS Instruction Addressing Modes

- PC-relative addressing –instruction address is the sum of the PC and a 16-bit constant contained within the instruction



- Pseudo-direct addressing – instruction address is the 26-bit constant contained within the instruction concatenated with the upper 4 bits of the PC



MIPS (RISC) Design Principles

- Simplicity favors regularity
  - fixed size instructions – 32-bits
  - small number of instruction formats
  - opcode always the first 6 bits
- Good design demands good compromises
  - three instruction formats
- Smaller is faster
  - limited instruction set
  - limited number of registers in register file
  - limited number of addressing modes
- Make the common case fast
  - arithmetic operands from the register file (load-store machine)
  - allow instructions to contain immediate operands

Next Lecture

- Next lecture
  - MIPS ALU Review
    - Reading assignment – PH, Chapter 3
    - Your work starts here. Understanding MIPS instruction set and SPIM http://spimsimulator.sourceforge.net/