Computer Architecture & Assembly Language 14:332:331

Lecture 2

MIPS Assembler

Naghmeh Karimi

Fall 16

Adapted from *Computer Organization and Design*, *5th Edition*, Patterson & Hennessy, © 2013, Elsevier, *and Computer Organization and Design*, *4th Edition*, Patterson & Hennessy, © 2008, Elsevier and Mary Jane Irwin's slides from Penn State University.

Assembly Language

- Language of the machine
- More primitive than higher level language

  e.g., no sophisticated control flow
- Very restrictive

  e.g., MIPS arithmetic instructions
- We work on the MIPS instruction set architecture
  - similar to other architectures developed since the 1980's
  - used by NEC, Nintendo, Silicon Graphics, Sony, …
  - 32-bit architecture
    - 32 bit data line and address line
    - data and addresses are 32-bit

Instruction Set

- The repertoire of instructions of a computer
- Different computers have different instruction sets
  - But with many aspects in common
- Early computers had very simple instruction sets
  - Simplified implementation
- Many modern computers also have simple instruction sets

RISC - Reduced Instruction Set Computer

- **RISC** philosophy
  - fixed instruction lengths
  - load-store instruction sets
  - limited number of addressing modes
  - limited number of operations

Example:

- MIPS, Sun SPARC, HP PA-RISC, IBM PowerPC …

- **CISC** (C for complex), e.g., Intel x86

MIPS (RISC) Design Principles

- Simplicity favors regularity

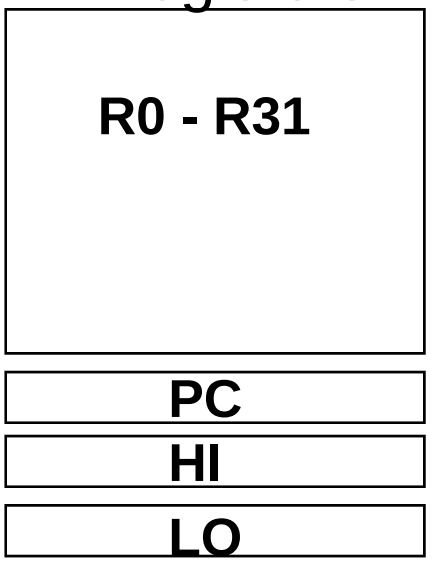
  (regularity makes implementation simpler and simplicity enables higher performance at lower cost)

  - fixed size instructions
  - small number of instruction formats
  - opcode always the first 6 bits
- 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
- Good design demands good compromises
  - three instruction formats

MIPS-32 ISA

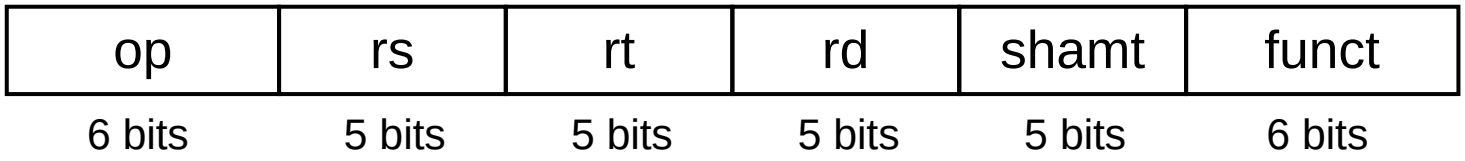
- Instruction Categories
  - Computational
  - Load/Store
  - Jump and Branch
  - Floating Point
    - coprocessor
  - Memory Management
  - Special

Registers



3 Instruction Formats: all 32 bits wide

MIPS R-format Instructions



- Instruction fields
  - op: operation code (opcode)
  - rs: first source register number
  - rt: second source register number
  - rd: destination register number
  - shamt: shift amount (00000 for now)
  - funct: function code (extends opcode)

Register Operand Example

- C code:

```
f = (g + h) - (i + j);
```

  - f, …, j in $s0, …, $s4

- Compiled MIPS code:

```
add $t0, $s1, $s2
add $t1, $s3, $s4
sub $s0, $t0, $t1
```

R-format Example

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

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

special	$s1	$s2	$t0	0	add
0	17	18	8	0	32
000000	10001	10010	01000	00000	100000

$$00000010001100100100000000100000_2 = 02324020_{16}$$

MIPS Arithmetic Instructions

- MIPS assembly language arithmetic statement

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

- Each arithmetic instruction performs one operation
- Each specifies exactly three operands that are all contained in the datapath's register file (`$t0,$s1,$s2`)

destination ← source1 op source2

- Instruction Format (**R format**)

Hexadecimal (Review)

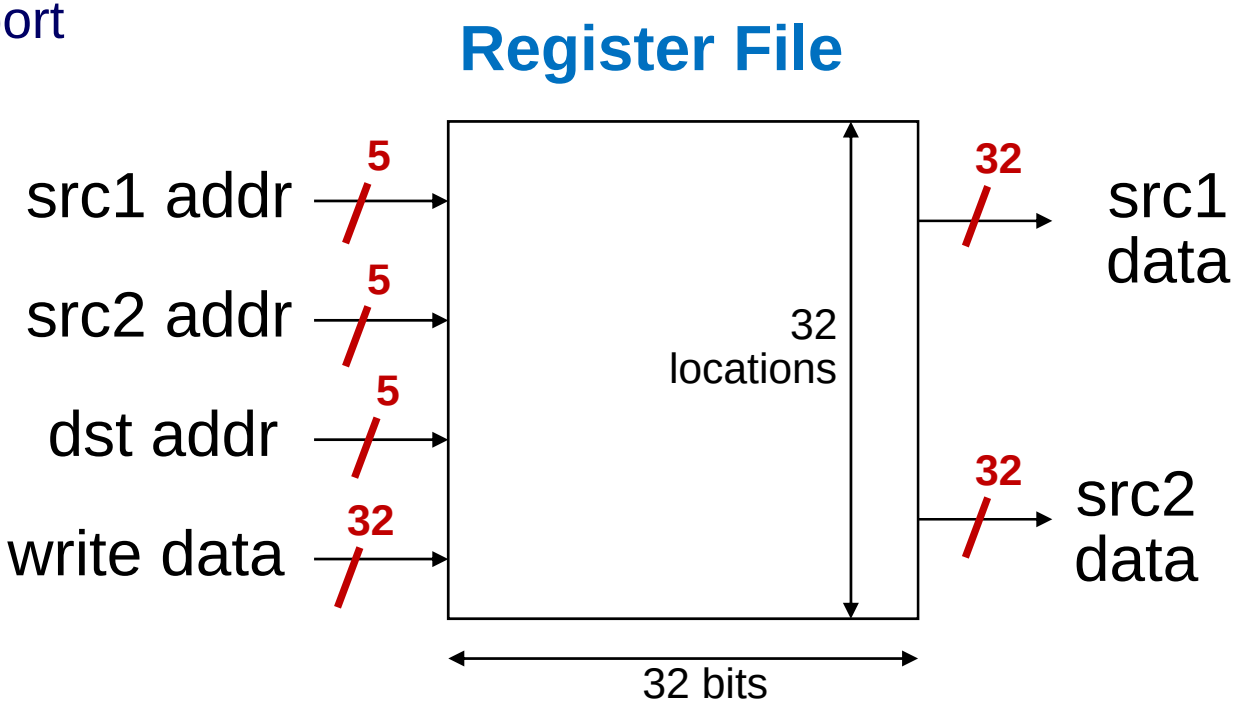
- Base 16
  - Compact representation of bit strings
  - 4 bits per hex digit

0	0000	4	0100	8	1000	c	1100
1	0001	5	0101	9	1001	d	1101
2	0010	6	0110	a	1010	e	1110
3	0011	7	0111	b	1011	f	1111

- Example: eca8 6420
  - 1110 1100 1010 1000 0110 0100 0010 0000

MIPS Register File

- Operands of arithmetic instructions must be from a limited number of special locations contained in the datapath's register file
  - Holds thirty-two 32-bit registers
    - With two read ports &
    - One write port



MIPS Register File (Cont'd)

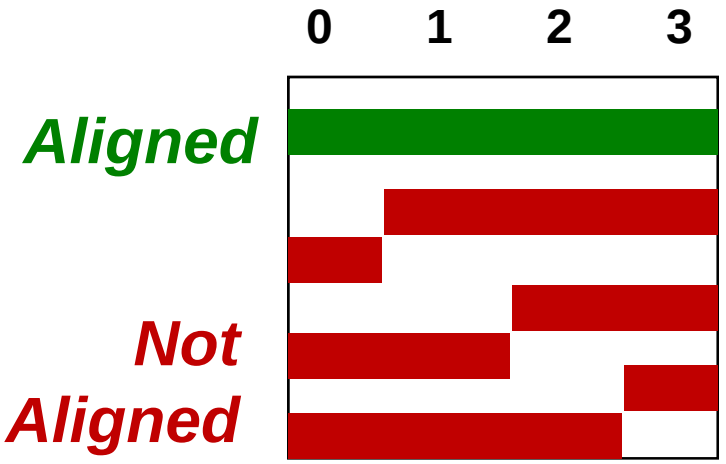
- Registers are
  - Faster than main memory
  - Can hold variables so that
    - code density improves (since register are named with fewer bits than a memory location)
  - Register addresses are indicated by using $

Naming Conventions for Registers

0	$zero constant 0 (Hdware)
1	$at reserved for assembler
2	$v0 expression evaluation &
3	$v1 function results
4	$a0 arguments
5	$a1
6	$a2
7	$a3
8	$t0 temporary: caller saves
. . .	(callee can clobber)
15	$t7
16	$s0 callee saves
. . .	(caller can clobber)
23	$s7
24	$t8 temporary (cont'd)
25	$t9
26	$k0 reserved for OS kernel
27	$k1
28	$gp pointer to global area
29	$sp stack pointer
30	$fp frame pointer
31	$ra return address (Hdware)

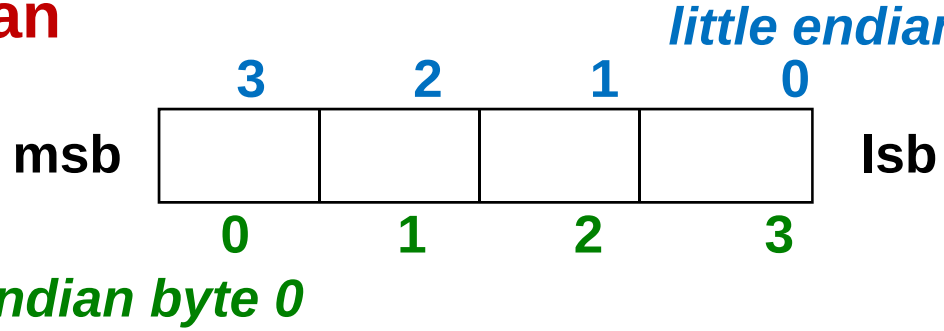
Memory Operands

- In MIPS, arithmetic instructions' operands must be registers.
- What if a program includes several variables? (only 32 registers are available)
  - Store variables in memory
  - Load values from memory into registers before use
  - Store result from register to memory
- Memory is byte addressed
  - Each address identifies an 8-bit byte
- Words are **aligned** in memory
  - Address must be a multiple of 4



Memory Operands (Cont'd)

- MIPS is **Big Endian**



  - **Big Endian :**
    - Most-significant byte at least address of a word
    - Leftmost byte is word address
    - Example: IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA

  - **Little Endian:**
    - Least-significant byte at least address
    - Rightmost byte is word address
    - Example: Intel 80x86, DEC Vax, DEC Alpha (Windows NT)

Registers vs. Memory

- Registers are faster to access than memory
- Operating on memory data requires loads and stores
- Compiler must use registers for variables as much as possible
  - Only spill to memory for less frequently used variables

Accessing Memory

- 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
```

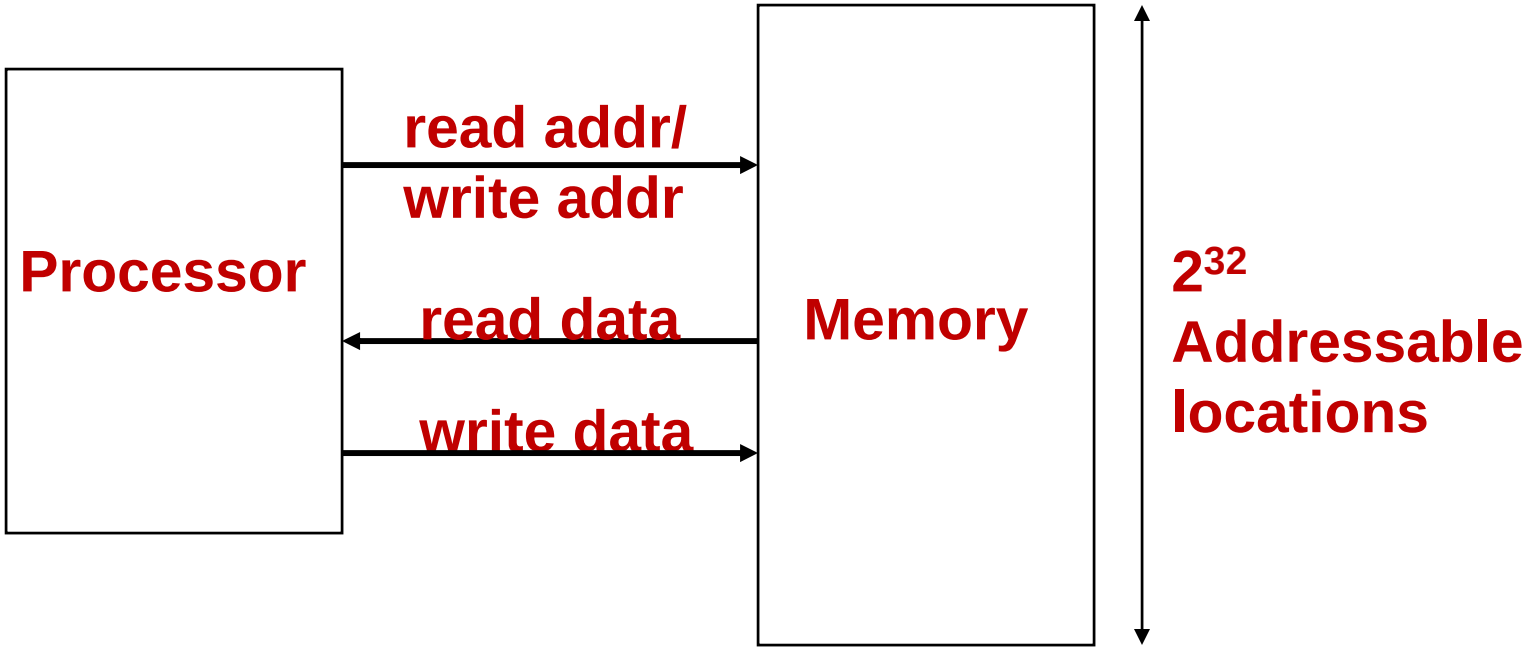
- The data transfer instruction must specify
  - where in memory to read from (load) or write to (store) – **memory address**
  - where in the register file to write to (load) or read from (store) – **register destination (source)**

- The memory address (a 32 bit address) is formed by adding the offset (- or +) to the contents of the base address register

- A 16-bit field (offset) meaning access is limited to memory locations within a region of $\pm 2^{13}$ words ($\pm 2^{15}$ bytes) of the address in the base register

Processor – Memory Interconnections

- Memory is viewed as a large, single-dimension array, with an address
- A memory address is an index into the array



Memory Operand Example 1

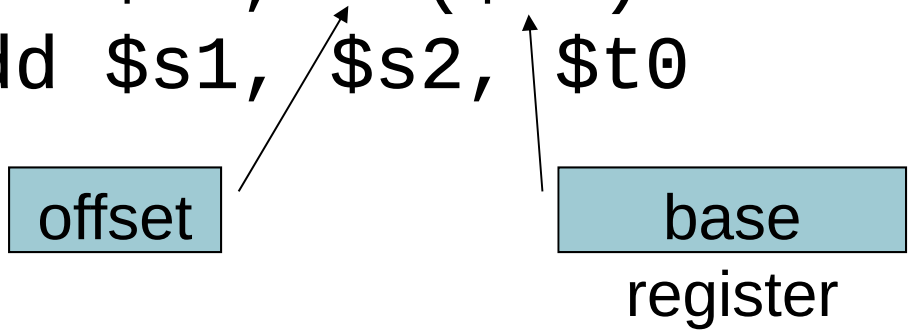
- C code:

```
g = h + A[8];
```

  - g in $s1, h in $s2, base address of A in $s3

- Compiled MIPS code:
  - Index 8 requires offset of 32
    - 4 bytes per word

```
lw  $t0, 32($s3)    # load word
add $s1, $s2, $t0
```



Memory Operand Example 2

- C code:

```
A[12] = h + A[8];
```

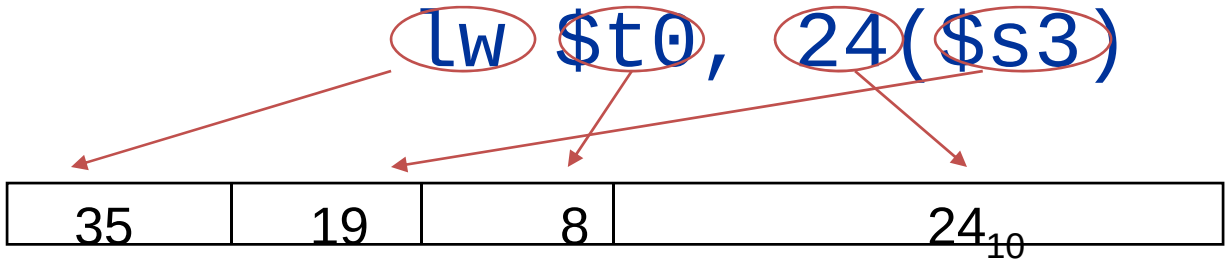
  - h in $s2, base address of A in $s3

- Compiled MIPS code:
  - Index 8 requires offset of 32

```
lw  $t0, 32($s3)    # load word
add $t0, $s2, $t0
sw  $t0, 48($s3)    # store word
```

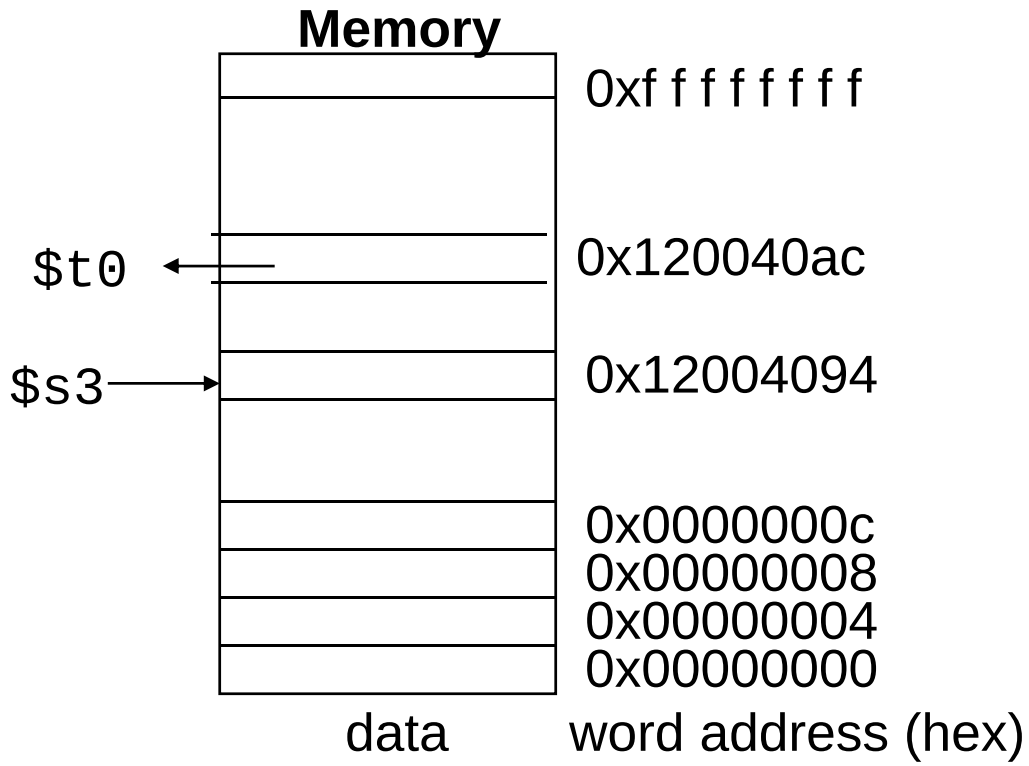
Load Instruction (Example)

- Load/Store Instruction Format (I format):



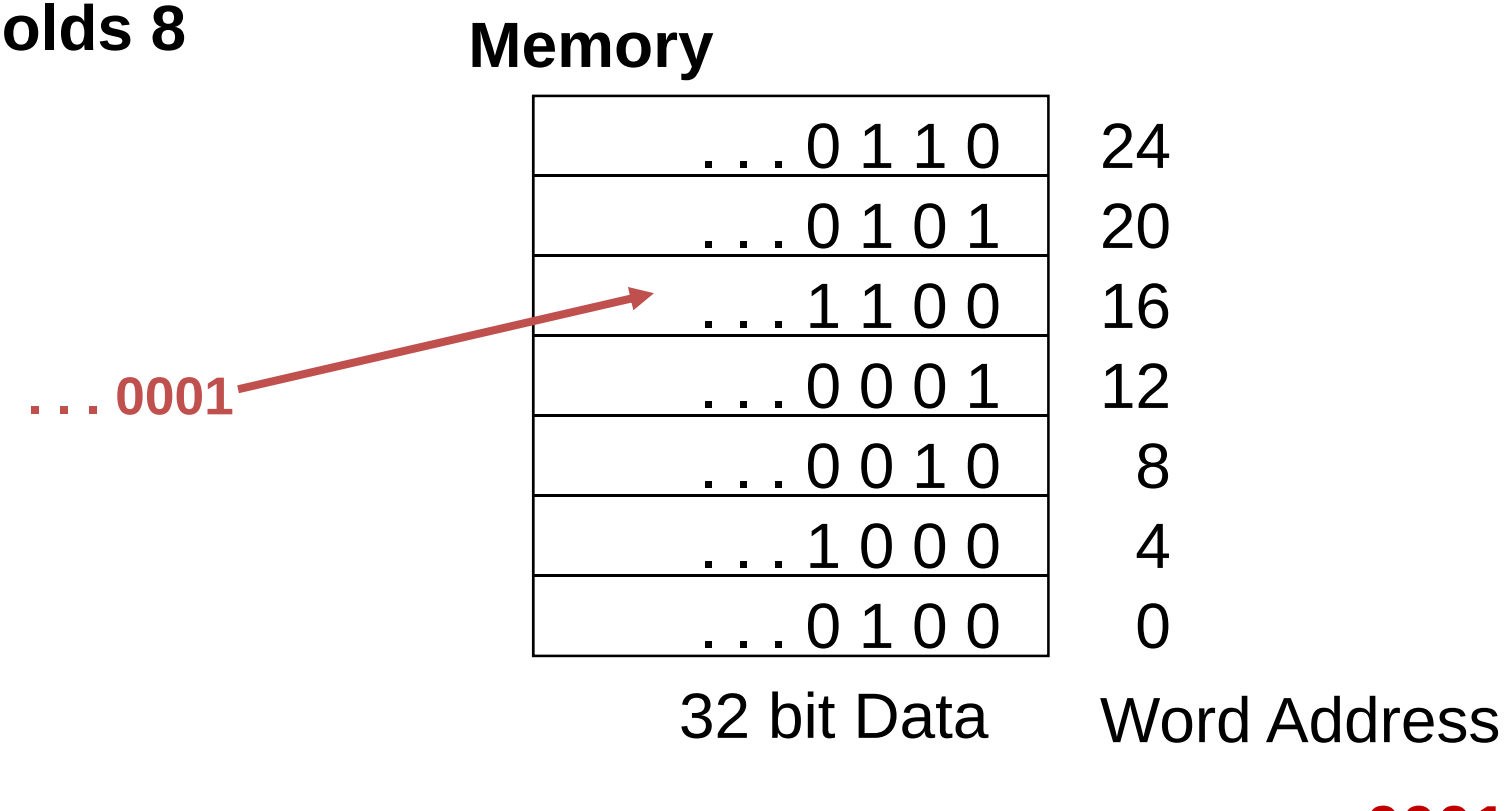
24_{10} + $s3 =

. . . 0001 1000
+ . . . 1001 0100
. . . 1010 1100 =
0x120040ac



MIPS Memory Instruction (Example)

$s3 holds 8



. . . 0001

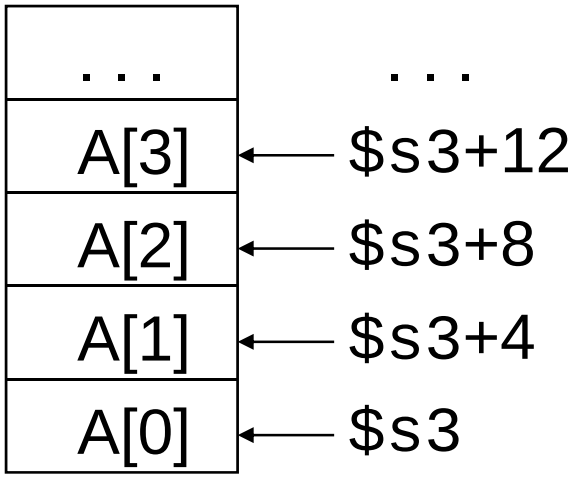
```
lw   $t0, 4($s3)  #what? is loaded into
  $t0 ?
sw   $t0, 8($s3)  #$t0 is stored where?
```

in location 16

Compiling with Loads and Stores

- Assuming variable b is stored in $s2 and that the base address of array A is in $s3, what is the MIPS assembly code for the C statement

A[8] = A[2] - b

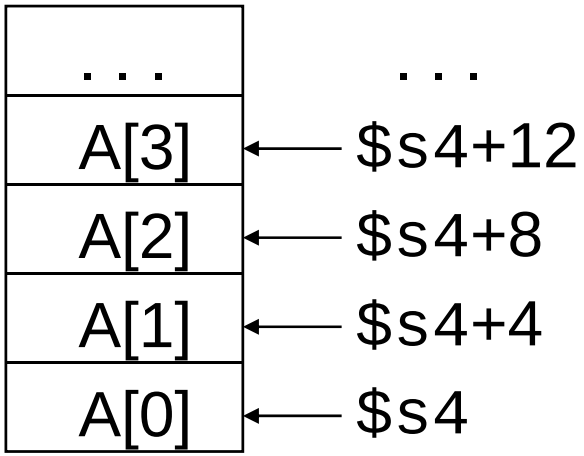



```
lw   $t0, 8($s3)
sub  $t0, $t0, $s2
sw   $t0, 32($s3)
```

Compiling with a Variable Array Index

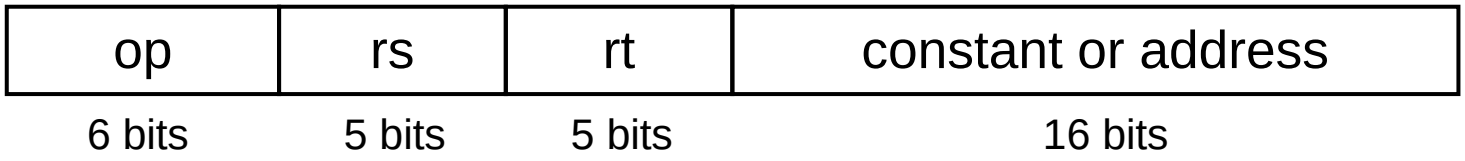
- Assuming that the base address of array A is in register $s4, and variables b, c, and i are in $s1, $s2, and $s3, respectively, what is the MIPS assembly code for the C statement

c = A[i] - b




```
add   $t1, $s3, $s3     #array index i is in $s3
add   $t1, $t1, $t1     #temp reg $t1 holds 4*i
add   $t1, $t1, $s4     #addr of A[i] now in $t1
lw    $t0, 0($t1)
sub   $s2, $t0, $s1
```

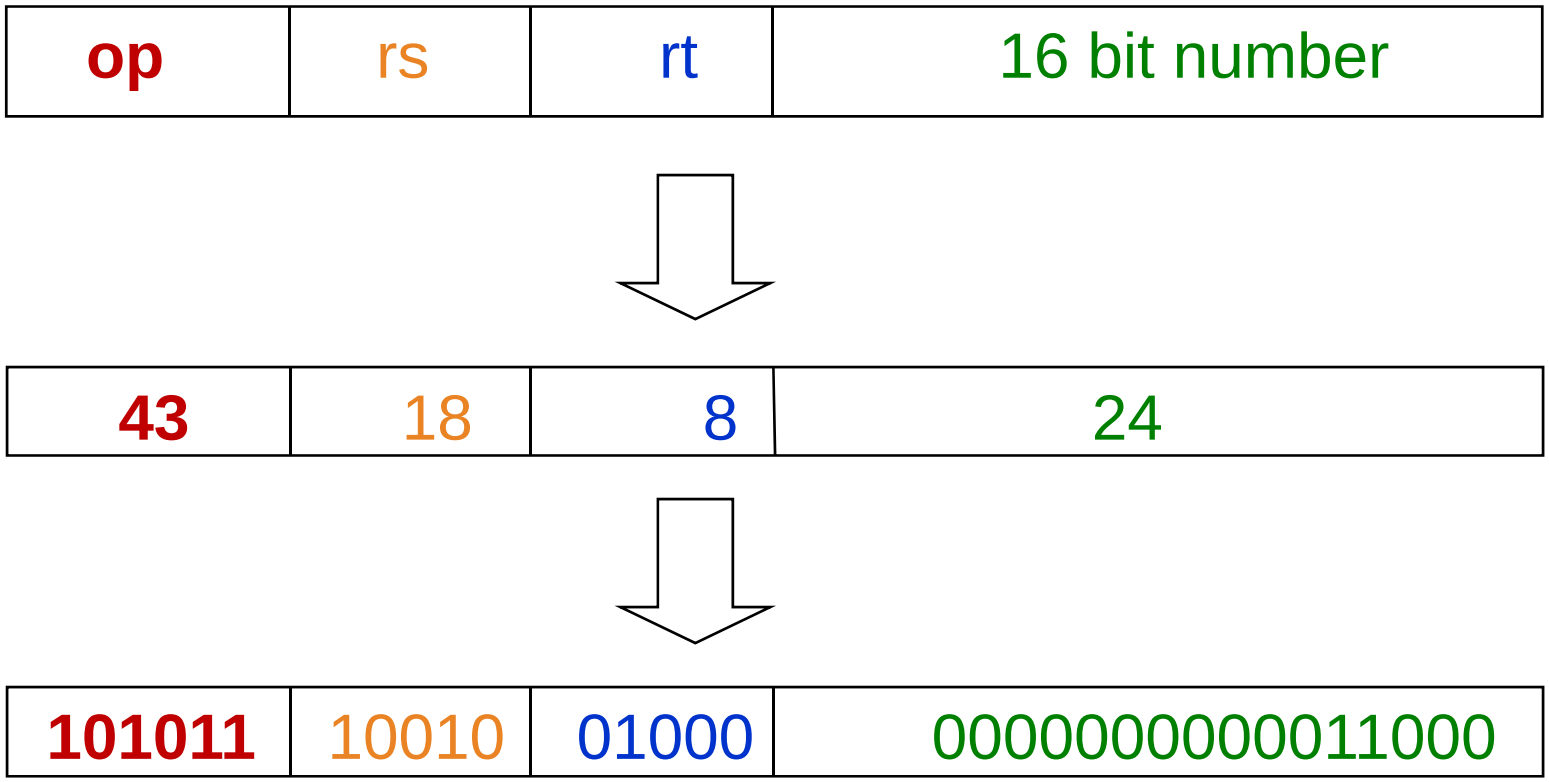
MIPS I-format Instructions



- Immediate arithmetic and load/store instructions
  - rt: destination register number
  - Constant: -2^{15} to $+2^{15} - 1$
  - Address: offset added to base address in rs

Machine Language - Store Instruction

- Example: sw **$t0**, **24**(**$s2**)



Immediate Operands

- Constant data specified in an instruction

```
addi $s3, $s3, 4
```

- No subtract immediate instruction
  - Just use a negative constant

```
addi $s2, $s1, -1
```

The Constant Zero

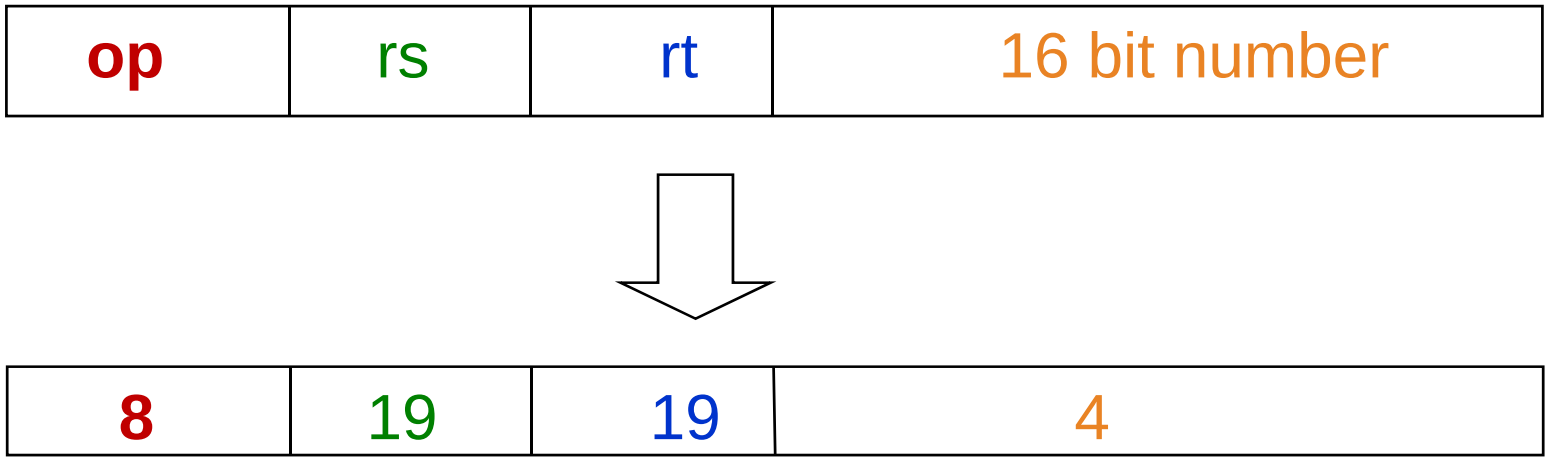
- MIPS register 0 ($zero) is the constant 0
  - Cannot be overwritten
- Useful for common operations
  - E.g., move between registers

```
add $t2, $s1, $zero
```

Machine Language – Immediate Instructions

- What instruction format is used for the `addi` ?

```
addi  $s3, $s3, 4 #$s3 = $s3 + 4
```

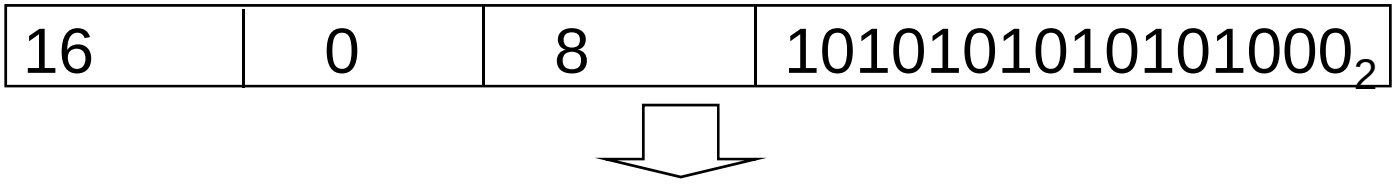


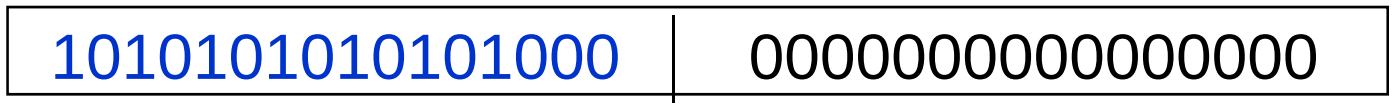
- Machine format:
  - The constant is kept inside the instruction itself!
    - So must use the I format – Immediate format
    - Limits immediate values to the range $+2^{15}-1$ to -2^{15}

Aside: How About Larger Constants?

- To load a 32 bit constant into a register, we must use two instructions
- a new "load upper immediate" instruction

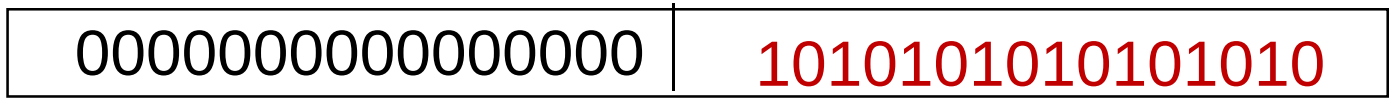
```
lui $t0, 1010101010101000
```

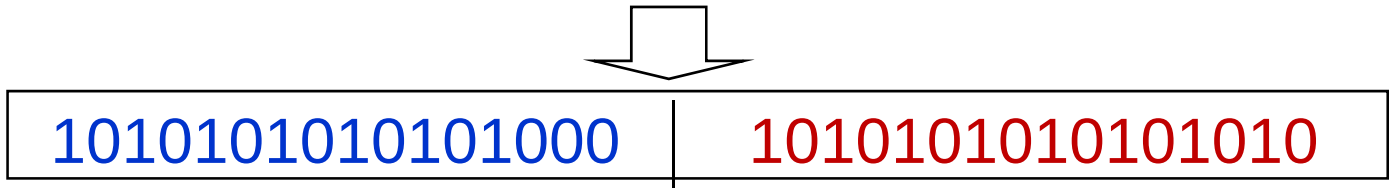




- Then must get the lower order bits right, use

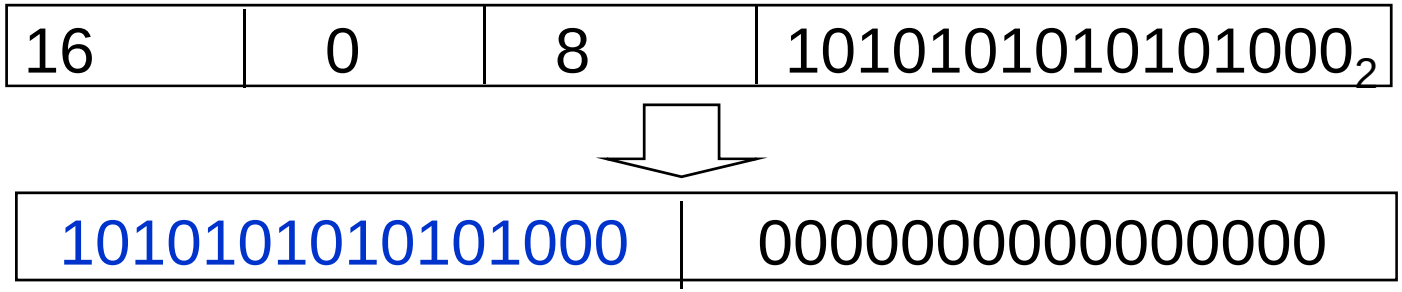
```
ori $t0, $t0, 1010101010101010
```



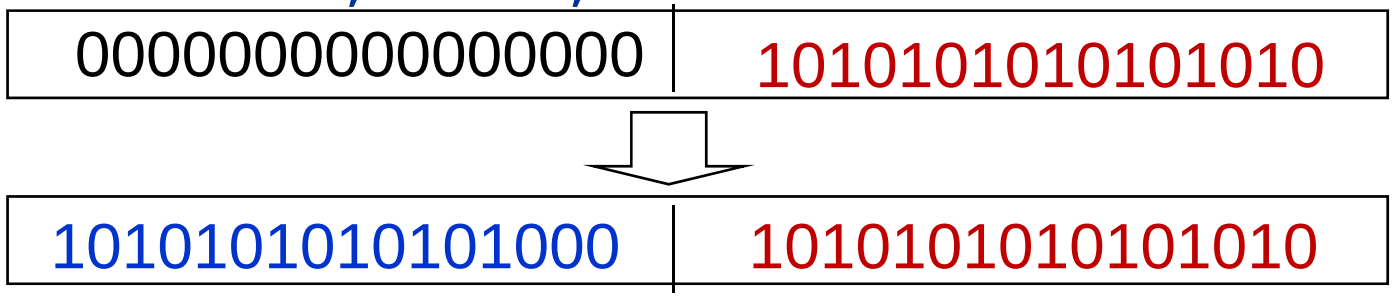


Aside: How About Larger Constants?

```
lui $t0, 1010101010101000
```




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



Can addi be used instead of ori?

No, addi sign-extends the 16 bit constant before doing addition but ori/andi 0-extends the 16 bit constant before doing or/and.

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
```

- 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? (sign-extend)
  - store byte takes the byte from the rightmost 8 bits of a register and writes it to a byte in memory (leaves other bits in the memory intact)

Example of Loading and Storing Bytes

•Given following code sequence and memory state (contents are given in hexidecimal), what is the state of the memory after executing the code?

```
add   $s3, $zero, $zero
lb    $t0, 1($s3)
sb    $t0, 6($s3)
```

Memory		
0 0 0 0 0 0 0 0	24	
0 0 0 0 0 0 0 0	20	
0 0 0 0 0 0 0 0	16	
1 0 0 0 0 0 1 0	12	
0 1 0 0 0 4 0 2	8	
F F F F F F F F	4	
0 0 9 0 1 2 A 0	0	
Data	Word Address (Decimal)	

- What value is left in $t0?
- What if the machine was little Endian?

Review: MIPS Instructions, so far

Category	Instr	Op Code	Example	Meaning
Arithmetic (R format)	**add**	**0 and 32**	**add $s1, $s2, $s3**	**$s1 = $s2 + $s3**
	subtract	**0 and 34**	**sub $s1, $s2, $s3**	**$s1 = $s2 - $s3**
Data transfer (I format)	**load word**	**35**	**lw $s1, 100($s2)**	**$s1 = Memory($s2+100)**
	store word	**43**	**sw $s1, 100($s2)**	**Memory($s2+100) = $s1**
	load byte	**32**	**lb $s1, 101($s2)**	**$s1 = Memory($s2+101)**
	store byte	**40**	**sb $s1, 101($s2)**	**Memory($s2+101) = $s1**

Review: MIPS Data Types

Integer: (signed or unsigned): 32 bits

Character: 8 bits

Floating point numbers: 32 bits

Memory addresses (pointers): 32 bits

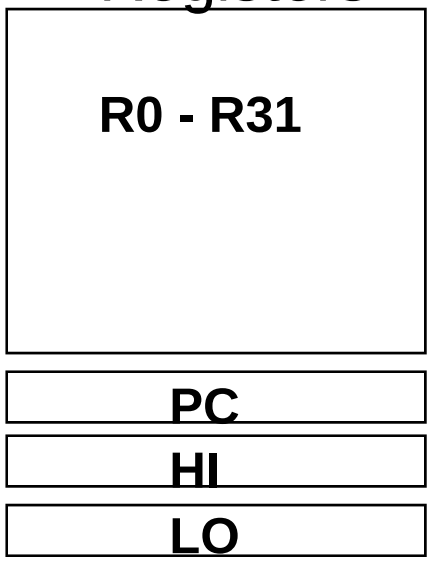
Instructions: 32 bits

Bit String: sequence of bits of a particular length

- **8 bits is a byte**
- **16 bits is a half-word**
- **32 bits (4 bytes) is a word**
- **64 bits is a double-word**

Review: MIPS-32 ISA

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



3 Instruction Formats: all 32 bits wide