

---

# EECE 321: Computer Organization

Mohammad M. Mansour  
*Dept. of Electrical and Compute Engineering*  
*American University of Beirut*

Lecture 6: MIPS ISA

---

# Announcements

---

- Office Hours:
  - Mondays: 1:00-2:00pm
  - Tuesdays: 11:00-12:00 noon
  - Fridays: 9:00-10:00am
  
- Reading assignment
  - Sections 2.4, 2.5, 2.6

# Data Transfers: Memory to Registers

---

- To transfer a word of data, we need to specify two things:
  - Register to receive data: specify this by number (\$0-\$31) or symbolically (\$s0,...,\$t0,...)
  - Memory address: more difficult
- Think of memory as a single one-dimensional array, so we can address it simply by supplying a pointer to a memory address.
- Other times, we want to be able to offset from this pointer.
- Remember: “Load FROM memory”
- To specify a memory address to copy from, need to specify two things:
  - A register containing a pointer to memory
  - A numerical offset (in bytes)
- The desired memory address is the sum of these two values.
- Syntax: `offset($reg)`
  - Example: `8($t0)`
  - specifies the memory address pointed to by the value in `$t0`, plus an offset of 8 bytes

# Data Transfers: Memory to Registers

---

- Load Instruction Syntax:

1 2, 3(4)

where:

1) operation name: “lw” ... Load Word

2) Register that will receive value

3) Numerical offset in bytes

4) Register containing pointer (address) to memory: Called based register

- MIPS Instruction Name:

- lw (meaning Load Word, so 32 bits or one word are loaded at a time)



- Example: lw \$t0,12(\$s0)

- This instruction will take the pointer in \$s0, adds 12 bytes to it, and then loads the value from the memory pointed to by this calculated sum into register \$t0.

- Notes:

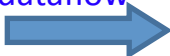
- \$s0 is called the base register

- 12 is called the offset

- offset is generally used in accessing elements of array or structure: base register points to beginning of array or structure

# Data Transfers: Registers to Memory

---

- Also want to store from register into memory
    - Store instruction syntax is identical to Load's
  - MIPS Instruction Name: “sw” or Store Word
    - 32 bits or one word are stored at a time
- dataflow  

- Example: sw \$t0, 12(\$s0)
    - This instruction will take the pointer in \$s0, add 12 bytes to it to form an address to memory, and then stores the contents of register \$t0 into memory at that address
  - Remember: “Store INTO memory”
  - Pointers versus Values:
    - Key Concept: A register can hold any 32-bit value. That value can be a
      - (signed) int, an unsigned int, a pointer (memory address), and so on.
    - If you write add \$t2, \$t1, \$t0, then \$t0 and \$t1 better contain values
    - If you write lw \$t2, 0(\$t0), then \$t0 better contain a pointer
  - These shouldn't be mixed up!

# Memory Addresses and Compilation

---

- Every word in memory has an **address**, similar to an **index** in an array
- Early computers numbered words like C numbers elements of an array:

– Memory[0], Memory[1], Memory[2], ...

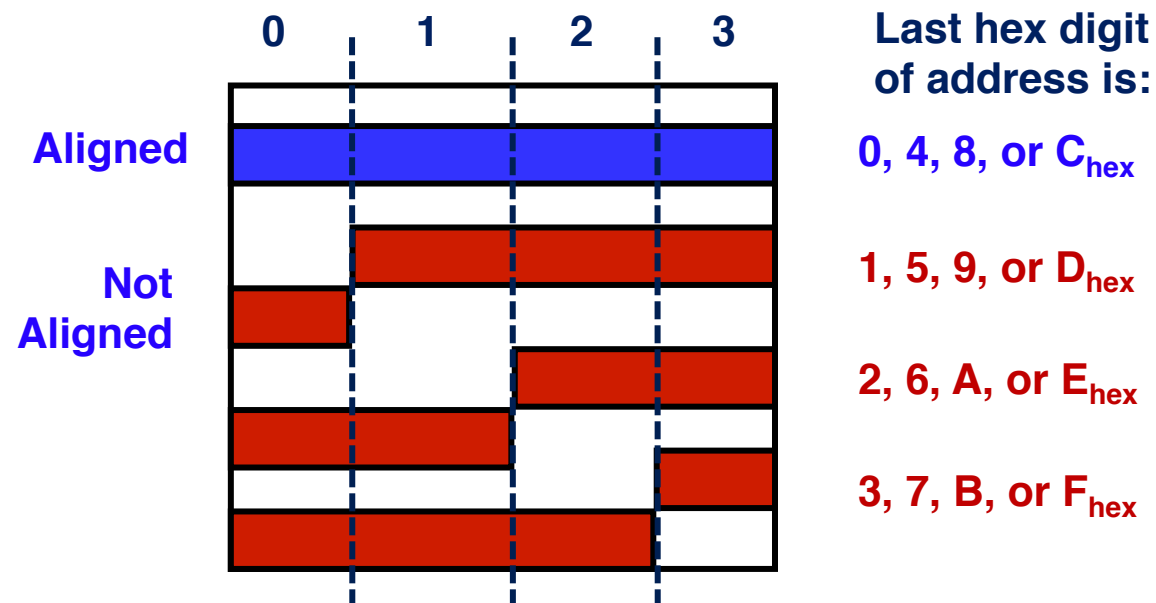


Called the **address** of a word

- Computers needed to access 8-bit bytes as well as words (4 bytes/word)
- Today machines address memory as bytes, (i.e., “Byte Addressable”) hence 32-bit (4 byte) word addresses differ by 4
  - Memory[0], Memory[4], Memory[8], ...
- What offset in **lw** to select **A[8]** in C?
  - $4 \times 8 = 32$  to select **A[8]**; byte v. word
- Compile by hand using registers the following C statement: **g = h + A[8];**
  - Assume the following mappings: **g:\$s1**, **h:\$s2**, **base address of A:\$s3**
  - First, transfer from memory to register.
  - This is done by adding 32 to **\$s3** to select **A[8]**, then put value into **\$t0**
    - **lw \$t0, 32(\$s3)**      # \$t0 gets A[8]
  - Next add loaded value to **h** and place in **g**
    - **add \$s1, \$s2, \$t0**      # \$s1 = h + A[8]

## A Note About Memory

- Pitfall: Forgetting that sequential word addresses in machines with byte addressing do not differ by 1.
  - Many assembly language programmers have made errors assuming that the address of the next word can be found by incrementing the address in a register by 1 instead of by the **word size in bytes**.
  - So remember that for both **lw** and **sw**, the sum of the **base address** and the **offset** must be a multiple of 4 (to be word aligned)
- MIPS requires that all words start at byte addresses that are multiples of 4 bytes
- Called Alignment: objects must fall on address that is multiple of their size.



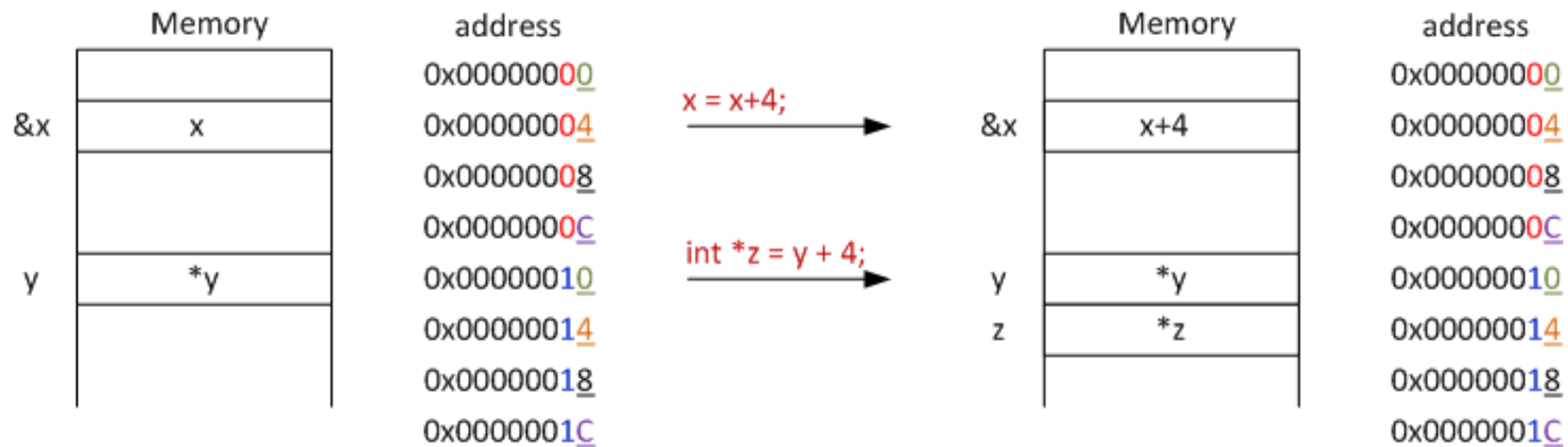
# Role of Registers Versus Memory

---

- What if we have more variables than registers?
  - Compiler tries to keep most frequently used variable in registers
  - Less common in memory: register spilling
- Why not keep all variables in memory?
  - Smaller is faster: registers are faster than memory
  - Registers are more versatile:
    - MIPS arithmetic instructions can read 2, operate on them, and write 1 per instruction
    - MIPS data transfer only read or write 1 operand per instruction, and no operation
- Example: We want to compile the C statement into MIPS
  - `int *x, *y;`
  - `*x = *y`      `/* where x and y are pointers stored in $s0, $s1 */`
  - Remember: `int *x` in C/C++ means x is defined to be a pointer to an integer object
  - Answer:
    - `lw $t0, 0($s1)`      `# Contents of memory at address pointed to by $s1 loaded into $t0`
    - `sw $t0, 0($s0)`      `# $t0 is stored in memory at address pointed to by $s0`

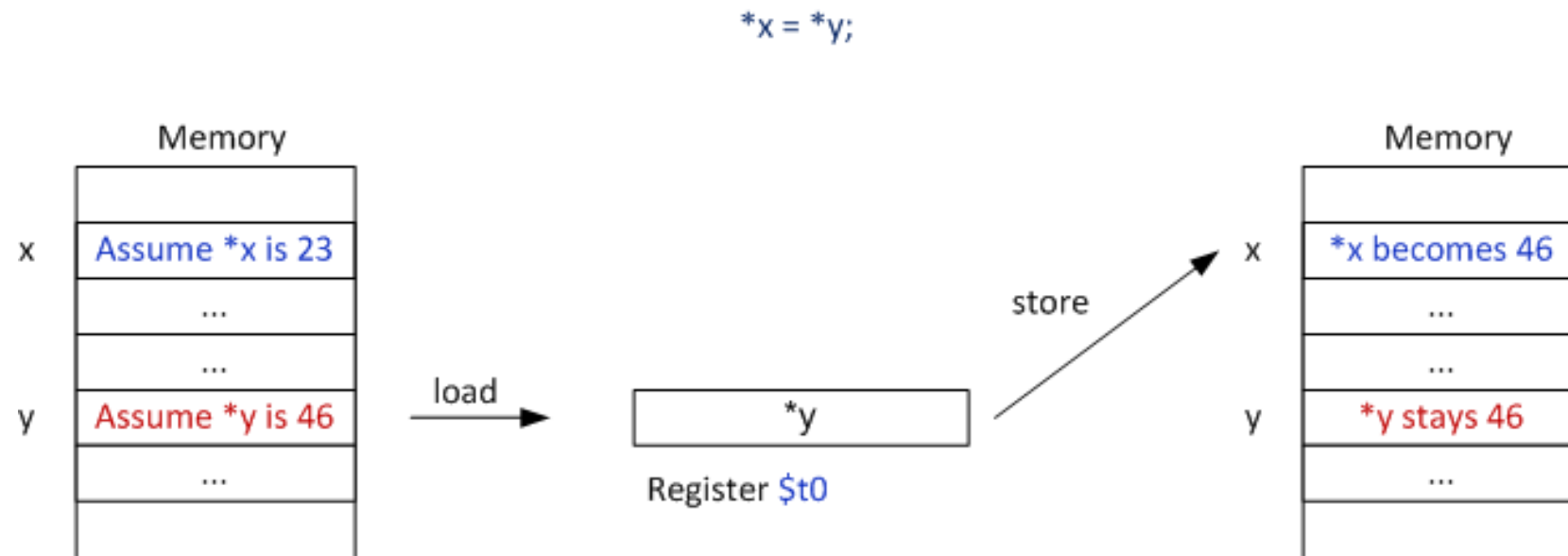
# Pointers in C

- Define variables x, y:
  - `int x;`
  - `int *y;`
- When used in expressions, x is the “value”. To refer to the address in memory where x is stored, use “&x”.
- For the pointer y, y is address, and “\*y” refers to the value pointed to by y.
  - y points to an object that represents an integer
  - Pointer arithmetic: y + 4 is an address that points to the next consecutive word in memory
  - x + 4 increments the value of x by 4



# Pointers in C

- `int *x, *y;`
- `*x = *y`



# Loading and Storing Bytes

---

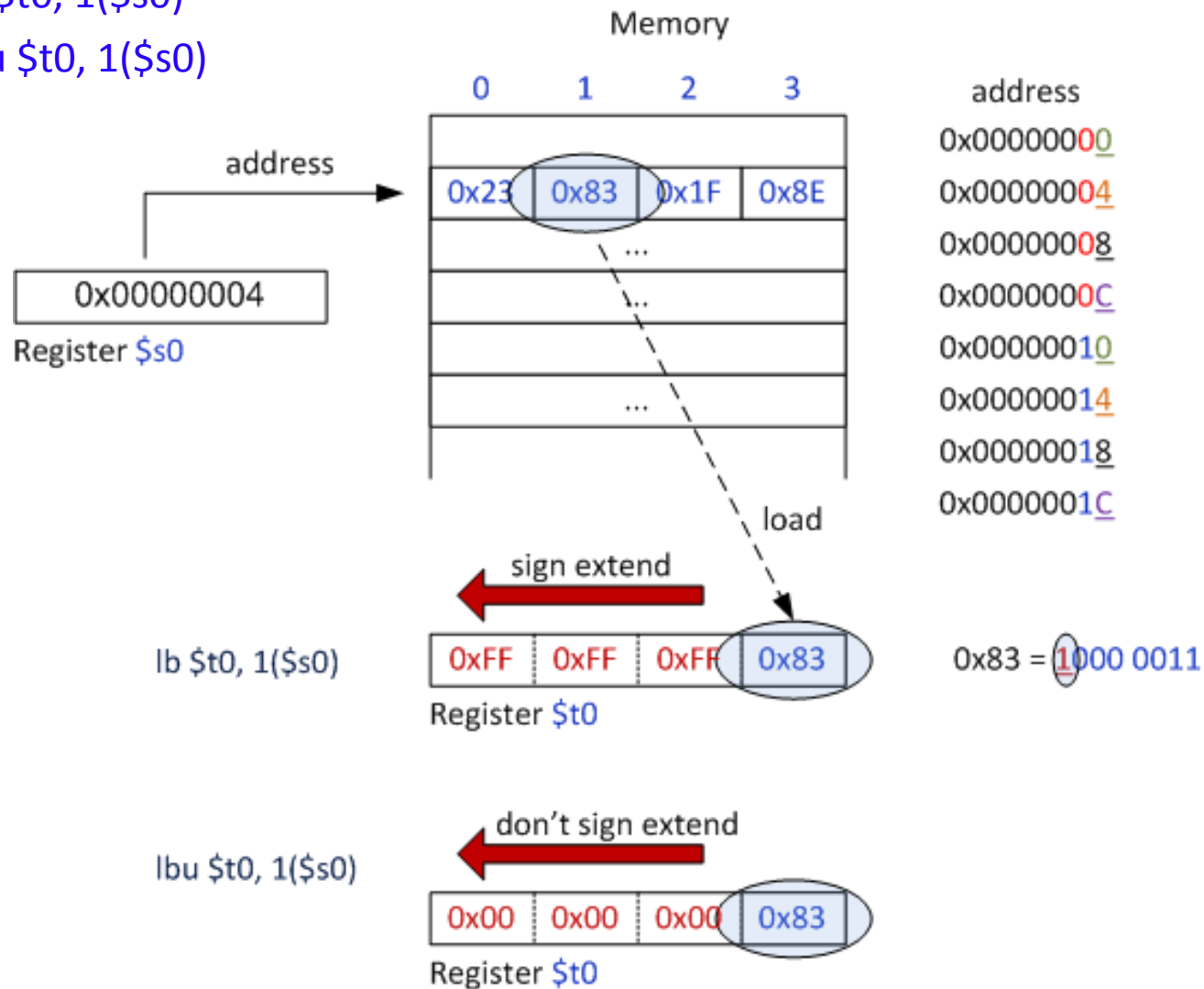
- In addition to word data transfers (*lw*, *sw*), MIPS has byte data transfers:
  - load byte: *lb*
  - store byte: *sb*
- same format as *lw*, *sw*
- What to do with other 24 bits in the 32 bit register?
- *lb*: **sign-extends** to fill upper 24 bits



- Ex: *lb \$t0, 1(\$s0)* **#load byte from memory**
  - Note that offset need not be a multiple of 4 in *lb* instruction
- Normally don't want to sign extend
  - Example when dealing with characters
- MIPS instruction that doesn't sign extend when loading bytes:
  - load byte unsigned: *lbu*

## Load Byte Example

- `lb $t0, 1($s0)`
- `lbu $t0, 1($s0)`



# Making Decisions

---

- All instructions so far only manipulate data ... we've built a *calculator*.
- In order to build a computer, we need ability to make decisions.
  - C (and MIPS) provide **labels** to support “goto” jumps to places in the code.
  - C: Horrible style; MIPS: Necessary!
- C Decisions: **if Statements**
- 2 kinds of **if statements** in C
  - if (condition) clause
  - if (condition) clause1 else clause2
- Rearrange 2nd if into following:
  - if (condition) goto **L1**;  
    Clause2;  
    goto L2;  
    **L1: clause1**;  
    L2: ...
- Note: Labels are simply locations of statements in your code, and not instructions
- Not as elegant as if-else, but same meaning.

# MIPS Decision Instructions: BEQ, BNE

---

- Decision instruction in MIPS:
  - `beq register1, register2, L1`
  - `beq` is “Branch if (registers are) equal”  
Same meaning as (using C):  
`if (register1==register2) goto L1`
- Complementary MIPS decision instruction
  - `bne register1, register2, L1`
  - `bne` is “Branch if (registers are) not equal”  
Same meaning as (using C):  
`if (register1!=register2) goto L1`
- `beq` and `bne` are called conditional branches

## MIPS Goto Instruction: J

---

- In addition to conditional branches, MIPS has an unconditional branch:
  - `j label`
- Called a Jump Instruction: jump (or branch) directly to the given label without needing to satisfy any condition.
- Same meaning as (using C):
  - `goto label`
- Technically, it's the same as:
  - `beq $0, $0, label`  
since it always satisfies the condition.