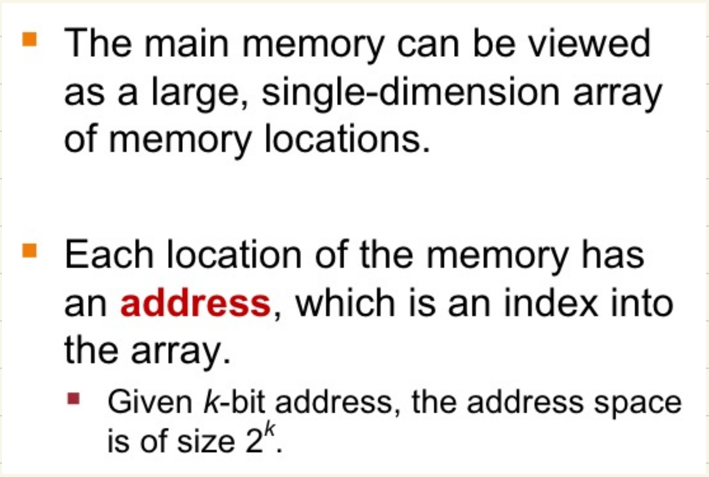
CS2100 - L08 - MIPS: Memory & Branches

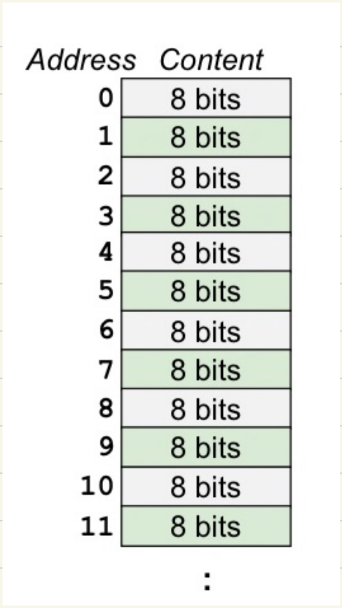
Week 4 + 5

- Memory
- MIPS Instructions Part II
  - Memory Instructions
  - Control Flow Instructions
- Handling Arrays

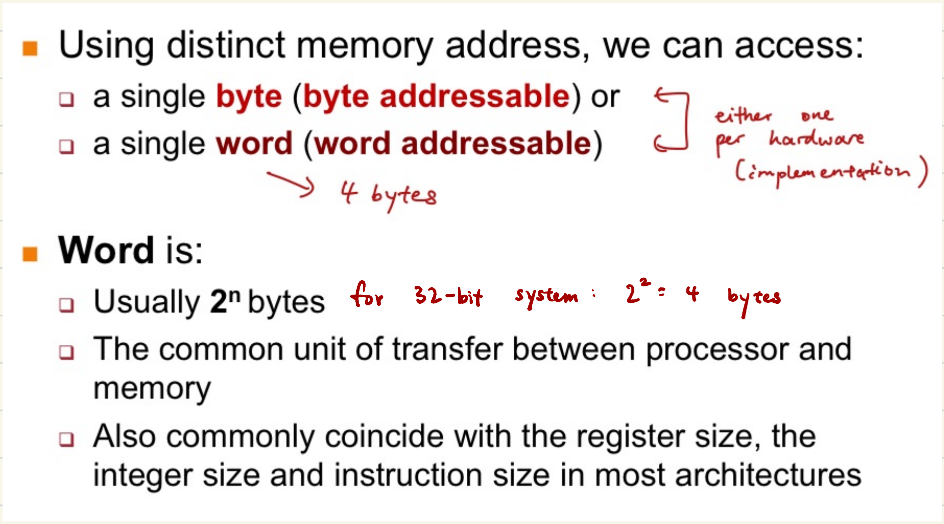
Memory



- The main memory can be viewed as a large, single-dimension array of memory locations.
- Each location of the memory has an **address**, which is an index into the array.
  - Given *k*-bit address, the address space is of size 2^k.



Address	Content
0	8 bits
1	8 bits
2	8 bits
3	8 bits
4	8 bits
5	8 bits
6	8 bits
7	8 bits
8	8 bits
9	8 bits
10	8 bits
11	8 bits
	:

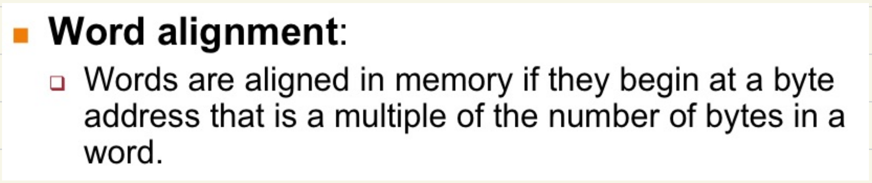


- Using distinct memory address, we can access:
  - a single **byte** (**byte addressable**) or
  - a single **word** (**word addressable**)

$\rbrace$ either one per hardware (implementation)

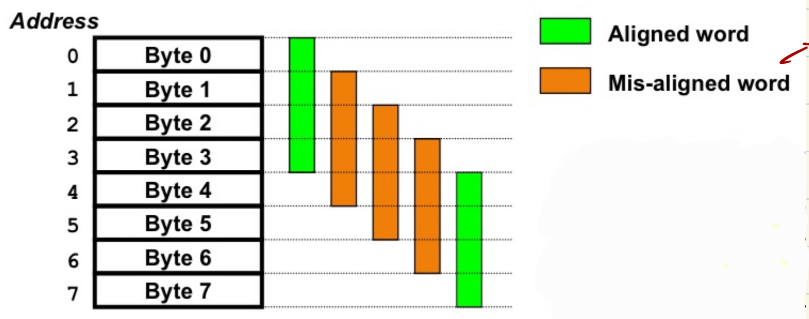
$\searrow$ 4 bytes

- **Word** is:
  - Usually **2^n** bytes for 32-bit system : $2^2 = 4$ bytes
  - The common unit of transfer between processor and memory
  - Also commonly coincide with the register size, the integer size and instruction size in most architectures



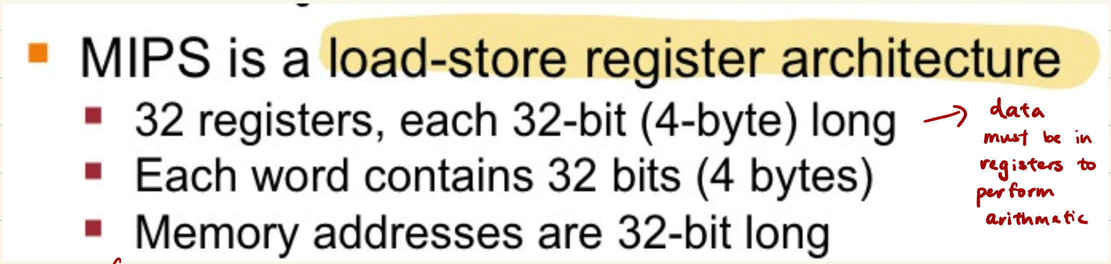
- **Word alignment**:
  - Words are aligned in memory if they begin at a byte address that is a multiple of the number of bytes in a word.

Example: If a word consists of 4 bytes, then:



→ much slower, takes more transfer units

MIPS - Memory Instructions



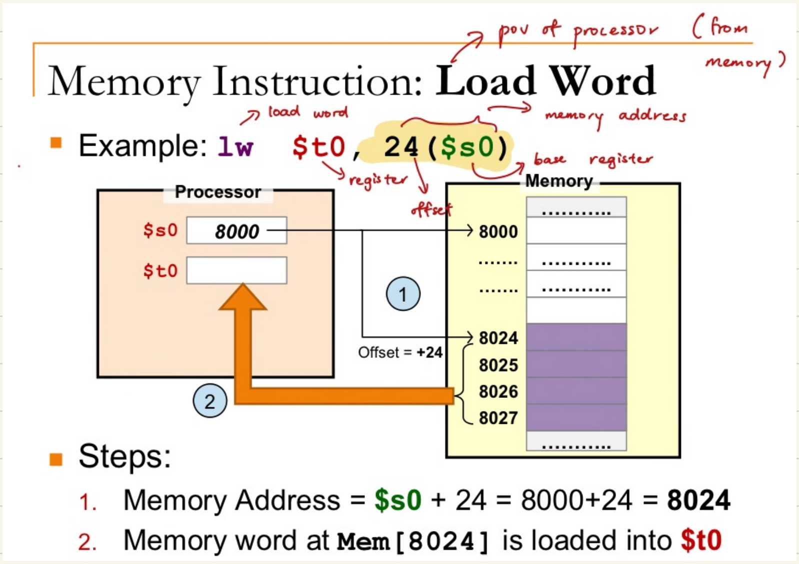
- MIPS is a load-store register architecture
  - 32 registers, each 32-bit (4-byte) long
  - Each word contains 32 bits (4 bytes)
  - Memory addresses are 32-bit long

→ data must be in registers to perform arithmatic

↳ MIPS uses byte addresses, so consecutive words differ by 4 bytes

↳ each byte has a memory address

- Load word :



Memory Instruction: **Load Word**

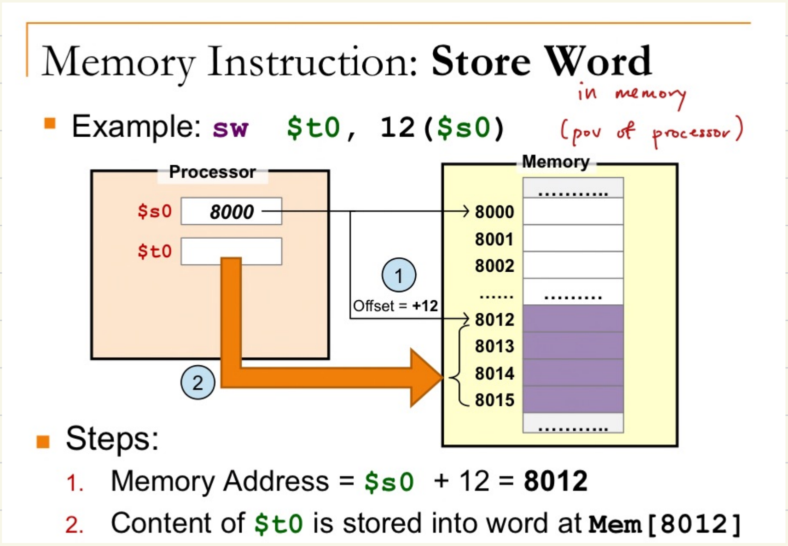
pov of processor (from memory)

- Example: `lw $t0, 24($s0)`

load word · register · offset · memory address · base register

- Steps:
  1. Memory Address = **$s0** + 24 = 8000+24 = **8024**
  2. Memory word at `Mem[8024]` is loaded into **$t0**

- Store word :



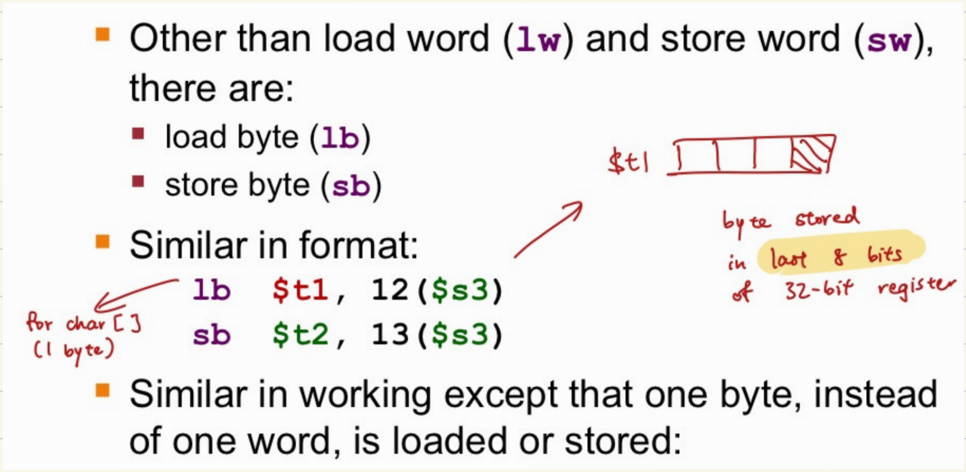
Memory Instruction: **Store Word**

in memory (pov of processor)

- Example: `sw $t0, 12($s0)`

- Steps:
  1. Memory Address = `$s0` + 12 = **8012**
  2. Content of `$t0` is stored into word at `Mem[8012]`

- Load word, store word:



- Other than load word (`lw`) and store word (`sw`), there are:
  - load byte (`lb`)
  - store byte (`sb`)
- Similar in format:

```
lb   $t1, 12($s3)
sb   $t2, 13($s3)
```

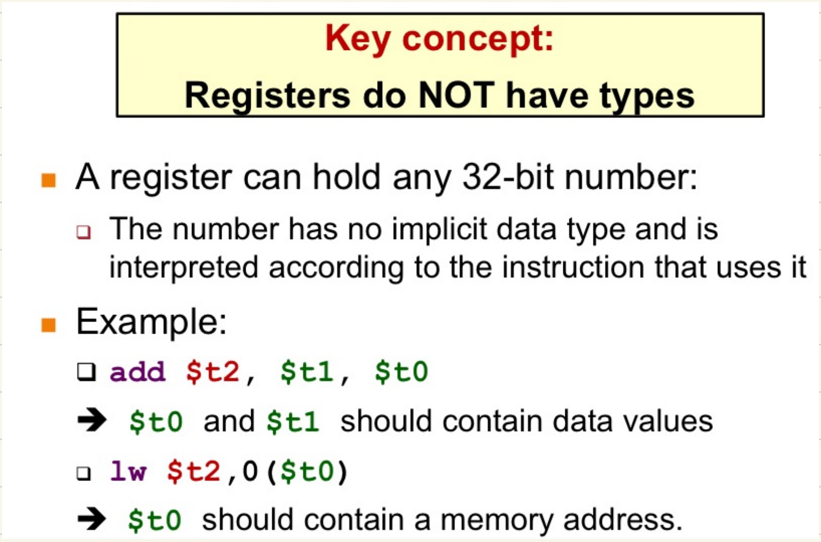
- Similar in working except that one byte, instead of one word, is loaded or stored:

for char[ ]
(1 byte)

$t1

byte stored in last 8 bits of 32-bit register

Address vs. Value



Key concept:

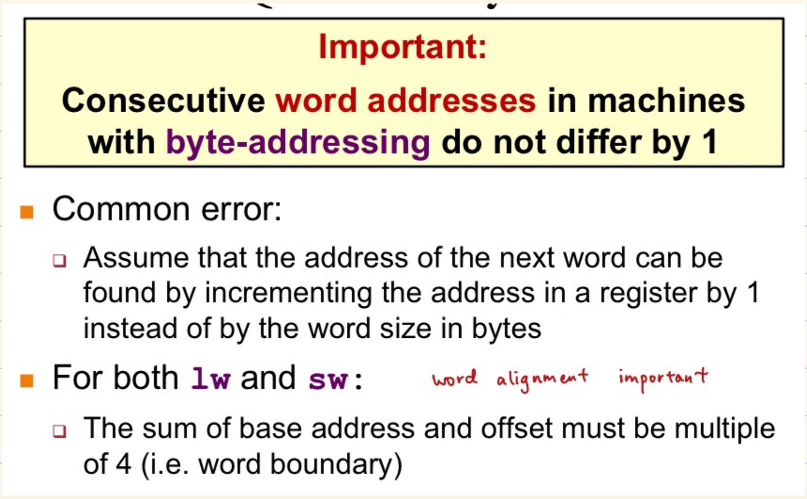
Registers do NOT have types

- A register can hold any 32-bit number:
  - The number has no implicit data type and is interpreted according to the instruction that uses it
- Example:
  - `add $t2, $t1, $t0`

  ➔ `$t0` and `$t1` should contain data values
  - `lw $t2,0($t0)`

  ➔ `$t0` should contain a memory address.

Byte vs. Word

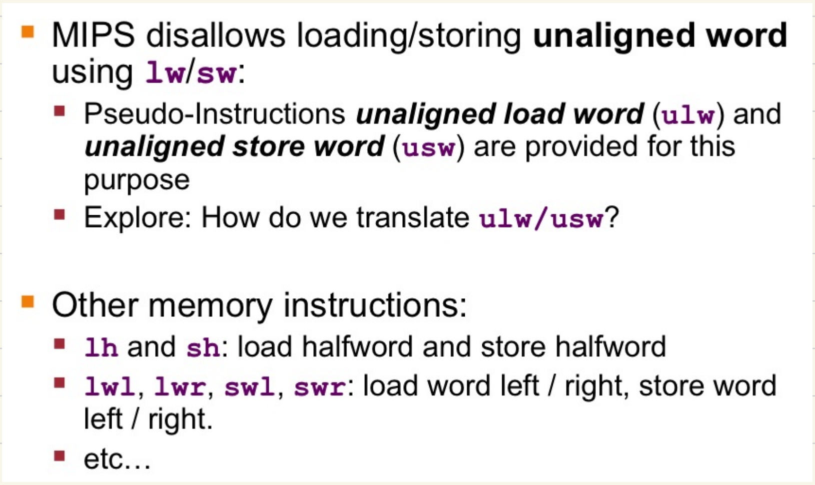


Important:

Consecutive word addresses in machines with byte-addressing do not differ by 1

- Common error:
  - Assume 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
- For both `lw` and `sw`: word alignment important
  - The sum of base address and offset must be multiple of 4 (i.e. word boundary)

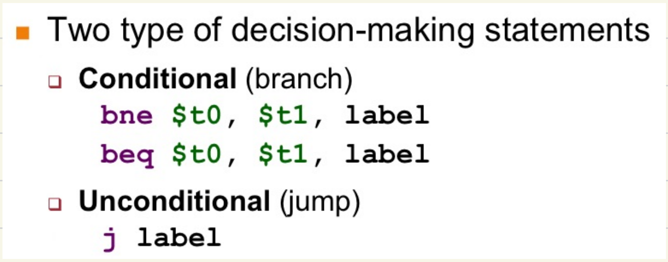
MIPS – Other Memory Instructions



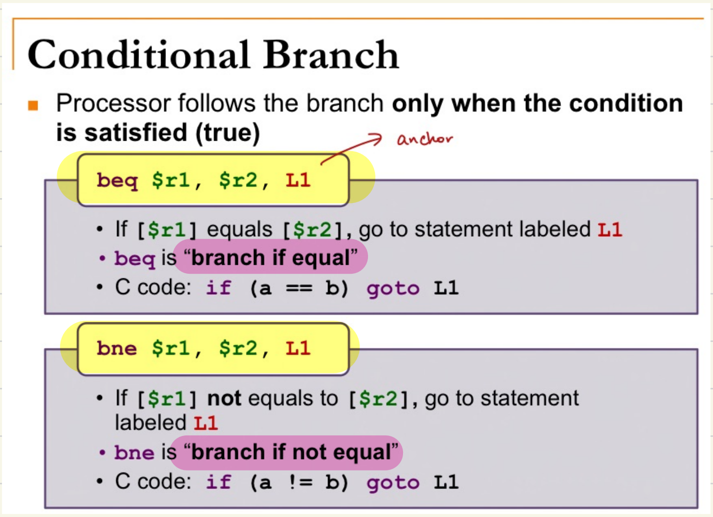
- MIPS disallows loading/storing **unaligned word** using `lw`/`sw`:
  - Pseudo-Instructions ***unaligned load word*** (`ulw`) and ***unaligned store word*** (`usw`) are provided for this purpose
  - Explore: How do we translate `ulw/usw`?
- Other memory instructions:
  - `lh` and `sh`: load halfword and store halfword
  - `lwl`, `lwr`, `swl`, `swr`: load word left / right, store word left / right.
  - etc…

MIPS - Control Flow Instructions

Translating C to Assembly : negate the condition for shorter code



- Two type of decision-making statements
  - **Conditional** (branch)
    ```
    bne $t0, $t1, label
    beq $t0, $t1, label
    ```
  - **Unconditional** (jump)
    ```
    j label
    ```



Conditional Branch

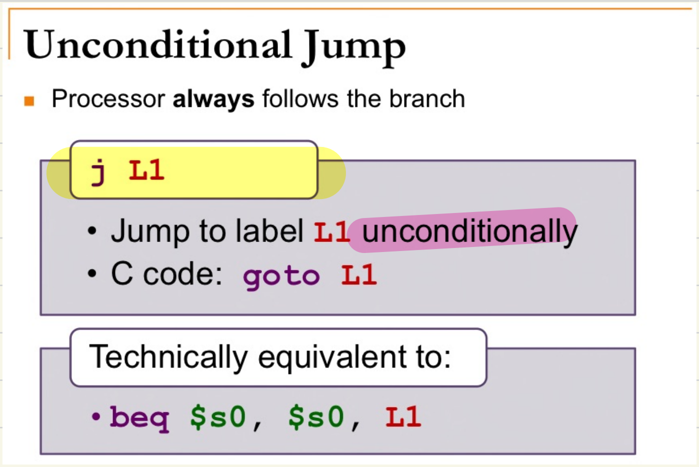
- Processor follows the branch **only when the condition is satisfied (true)**

```
beq $r1, $r2, L1
```
anchor

- If `[$r1]` equals `[$r2]`, go to statement labeled `L1`
- `beq` is "**branch if equal**"
- C code: `if (a == b) goto L1`

```
bne $r1, $r2, L1
```

- If `[$r1]` **not** equals to `[$r2]`, go to statement labeled `L1`
- `bne` is "**branch if not equal**"
- C code: `if (a != b) goto L1`



Unconditional Jump

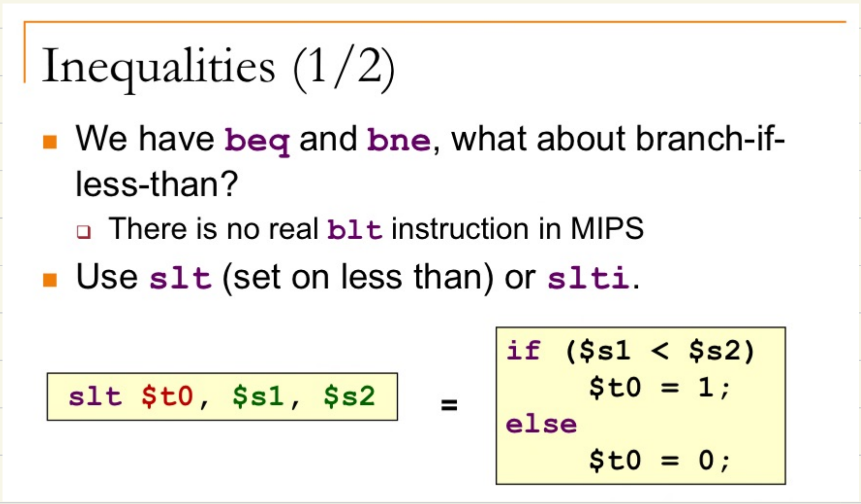
- Processor **always** follows the branch

```
j L1
```

- Jump to label `L1` unconditionally
- C code: `goto L1`

Technically equivalent to:

- `beq $s0, $s0, L1`



Inequalities (1/2)

- We have **beq** and **bne**, what about branch-if-less-than?
  - There is no real **blt** instruction in MIPS
- Use **slt** (set on less than) or **slti**.

```
slt $t0, $s1, $s2
```

=

```
if ($s1 < $s2)
     $t0 = 1;
else
     $t0 = 0;
```