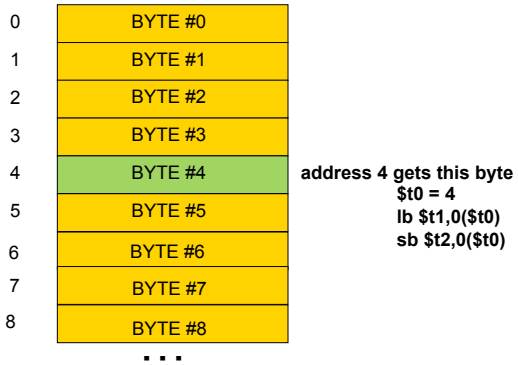
Memory transfer instructions

- How to get values to/from memory?
  - Also called *memory access* instructions
- Only two types of instructions
  - Load: move data from memory to register (“load the register”)
    - e.g., lw $s5, 4($t6) # $s5 ⇐ memory[$t6 + 4]
  - Store: move data from register to memory (“save the register”)
    - e.g., sw $s7, 16($t3) # memory[$t3+16] ⇐ $s7
- In MIPS (32-bit architecture) there are memory transfer instructions for
  - 32-bit word: “int” type in C (lw, sw)
  - 16-bit half-word: “short” type in C (lh, sh; also unsigned lhu)
  - 8-bit byte: “char” type in C (lb, sb; also unsigned lbu)

Memory view

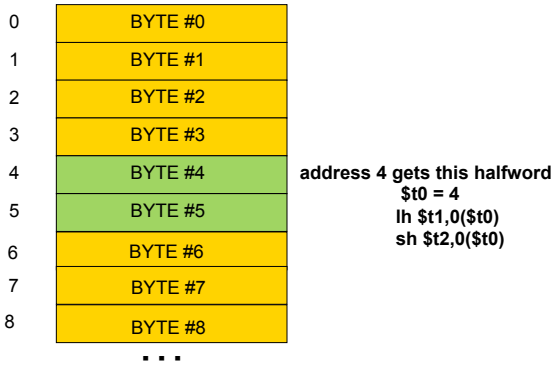
- Memory is a large, single-dimension 8-bit (byte) array with an address to each 8-bit item (“byte address”)
- A ***memory address is just an index into the array***



- loads and stores give the index (address) to access

Memory view

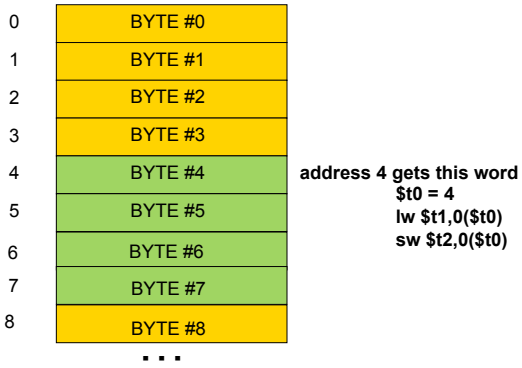
- Memory is a large, single-dimension 8-bit (byte) array with an address to each 8-bit item (“byte address”)
- A ***memory address is just an index into the array***



- loads and stores give the index (address) to access

Memory view

- Memory is a large, single-dimension 8-bit (byte) array with an address to each 8-bit item (“byte address”)
- A ***memory address is just an index into the array***



- loads and stores give the index (address) to access

Effective Address calculation

- ***Effective memory address*** specified as *immediate($register)*
  - Register to keep the ***base address***
  - Immediate to determine an ***offset*** from the base address
  - Thus, address is *contents of register + immediate*
  - The offset can be positive or negative, 16-bit value (uses I-format)
- Suppose base register $t0=64, then:

```
lw    $t0, 12($t1)      address = 64 + 12 = 76
lw    $t0, -12($t1)     address = 64 - 12 = 52
```

- MIPS uses this simple address calculation; other architectures such as PowerPC and x86 support different methods

Hint on addresses (`la` - load address)

- Often, you need to reference a particular variable.

```
.data
var:   .word        1000
```

assembler directive to declare data (word)

- How to reference **`var`**?

```
la   $t0,var
lw   $t1,0($t0)
```

puts the address of variable "var" into $t0

value at the address in $t0 is loaded ino $t1

- **`la`** is a "pseudo-instruction". It is turned into a sequence to put a large address constant into $t0.

```
lui   $at,upperbitsofaddres
ori   $t0,$1,lowerbitsofaddress
```

Let's try an in-class exercise together!

- Create a word (integer) variable "myVar"
- Give the variable the value 20
- Print the value to the console (Run I/O window)
- Terminate the program
- Extension: Add 10 to the value, store it to myVar, print it

- To do this, we'll need to use:
  - Data segment declaration with a word variable type
  - Instruction segment declaration
  - Load word instruction
  - Syscall instruction
  - Assorted la and li instructions

Let's try an in-class example together

- Consider the C program and rewrite as MIPS

```
void fun(void) {
        int a=10,b=20,c=30;
        a=a+10;
        b=0;
        c=a+b;
}
```

Machine code example

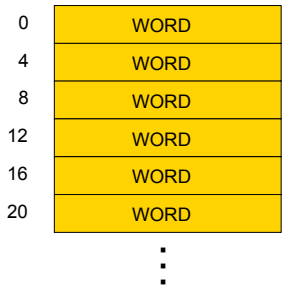
```
void swap(int v[], int k)
{
        int temp;
        temp = v[k];
        v[k] = v[k+1];
        v[k+1] = temp;
}
```

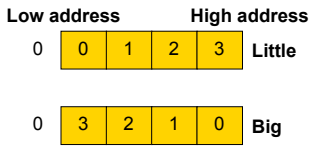
```
swap:
        sll     $t0, $a1, 2
        add     $t1, $a0, $t0
        lw      $t3, 0($t1)
        lw      $t4, 4($t1)
        sw      $t4, 0($t1)
        sw      $t3, 4($t1)
        jr      $ra
```

Let's try it in MARS!!!! (mips4.asm)

Memory organization

- 32-bit byte address
  - 2^{32} bytes with byte addresses from 0 to $2^{32} - 1$
  - 2^{30} words with byte addresses 0, 4, 8, …, $2^{32} - 4$
- Words are aligned
  - 2 least significant bits (LSBs) of an address are 0s
- Half words are aligned
  - LSB of an address is 0
- Addressing within a word
  - Which byte appears first and which byte the last?
  - Big-endian vs. little-endian
    - "Little end (LSB) comes first (at low address)"
    - "Big end (MSB) comes first (at low address)"

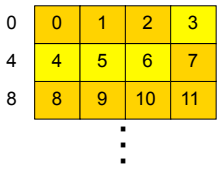




Let's try it in MARS!!!! (mips5.asm)

More on alignment

- A misaligned access
  - Assume $t0=0, then lw $s4, 3($t0)
- How do we define a word at address?
  - Data in byte 0, 1, 2, 3
    - If you meant this, use the address 0, not 3
  - Data in byte 3, 4, 5, 6
    - If you meant this, it is indeed misaligned!
    - Certain hardware implementation may support this; usually not
    - If you still want to obtain a word starting from the address 3 – get a byte from address 3, a word from address 4 and manipulate the two data to get what you want



- Alignment issue does not exist for byte access

Shift instructions

Name	Fields						Comments
R-format	op	NOT USED	rt	rd	shamt	funct	shamt is “shift amount”

- Bits change their positions inside a word
- <op> <r_{target}> <r_{source}> <shift_amount>
- Examples
  - sll $s3, $s4, 4 # $s3 ⇐ $s4 << 4
  - srl $s6, $s5, 6 # $s6 ⇐ $s5 >> 6
- Shift amount can be in a register (“shamt” is not used)
- Shirt right arithmetic (sra) keeps the sign of a number
  - sra $s7, $s5, 4

Let’s try it in MARS!!!! (mips6.asm)