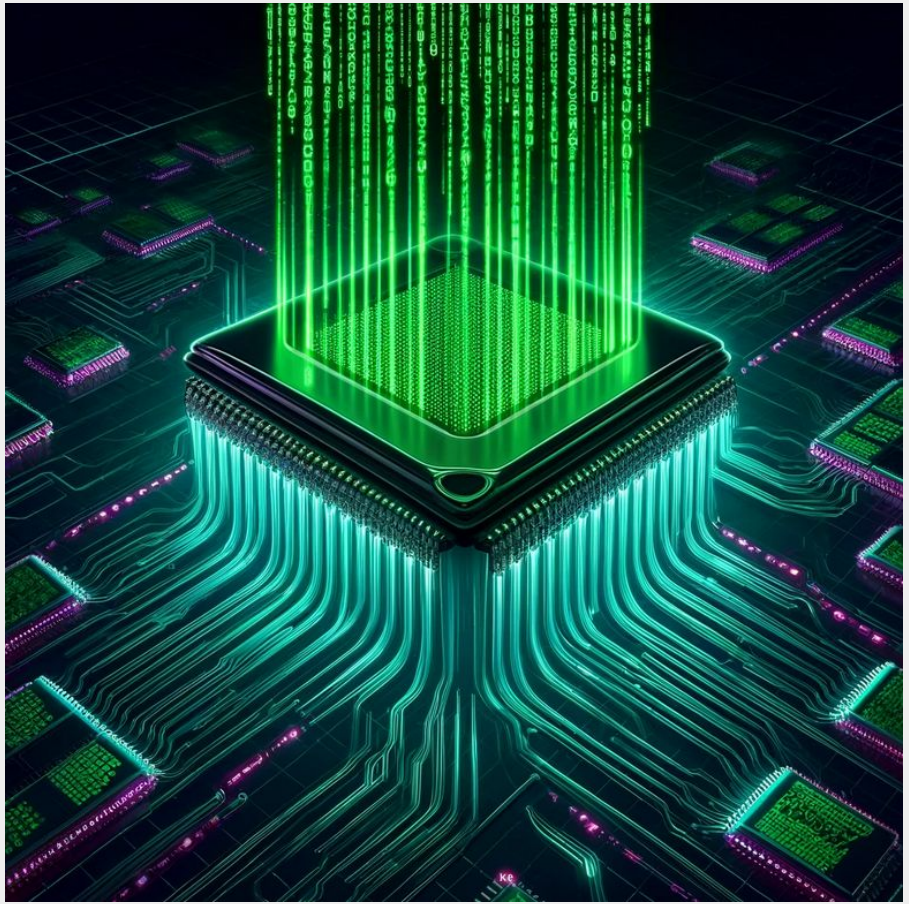
UNSW SYDNEY

COMP1521 24T2 Lec03

MIPS: DATA



2024

Hammond Pearce

Inspired from Abiram's Material

Lecture chat

https://cgi.cse.unsw.edu.au/~cs1521/accord/

Recap of lec02

- We can write more fun assembly now!
- We can `syscall` things in and out of the “operating system”
- We can convert ridiculous C constructs like “loops” and “conditionals” into their one **true** representation - branch + goto

Recap exercise

- Open Mipsy
- Use a syscall to get an integer from the user
- Check if the integer is even:
  - if so, syscall to print the integer
  - if not, syscall to print 0
- Return 0

Put your answer in the lecture chat →

Recap exercise

```
.text
main:
        li      $v0, 5
        syscall
        move    $t0, $v0
        andi    $t1, $t0, 1
        bgtz    $t1, is_odd
is_even:
        move    $a0, $t0
        li      $v0, 1
        syscall
        b       prog_end
is_odd:
        li      $a0, 0
        li      $v0, 1
        syscall
prog_end:
        li $v0, 0
        jr $ra
```

li vs la vs move

- **li** (load immediate) is for immediate, ***fixed values*** that you need to load into a register with an instruction
- **la** (load address) is for loading ***fixed addresses*** into a register
  - remember, labels really just represent addresses!
- **move** is for copying values ***between two registers***

TODAY: Data and Memory

How do we store/use interesting data?

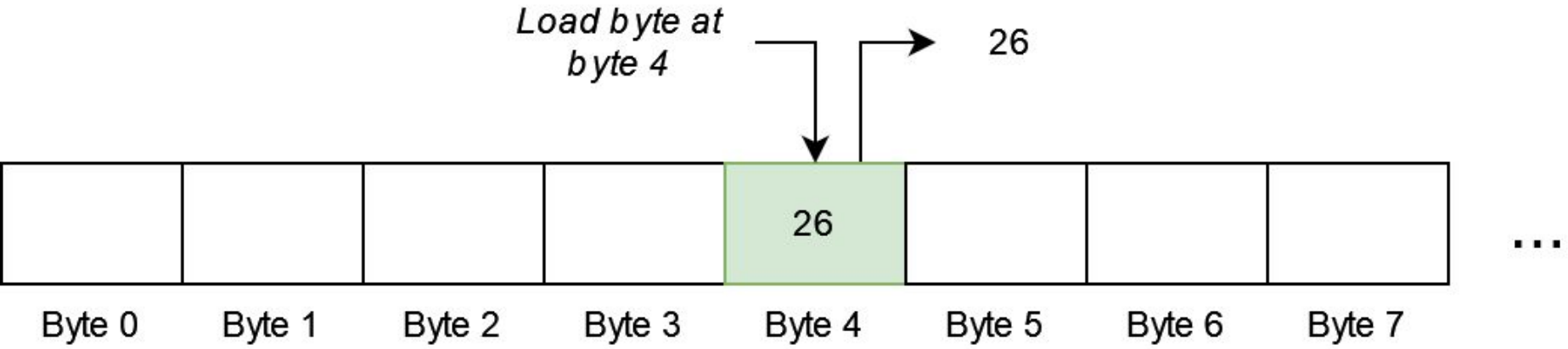
How does the data segment really work?

How do we:

- Store and increment a global variable?
- Work with 1D arrays?
- Work with 2D arrays??
- C Structs !?

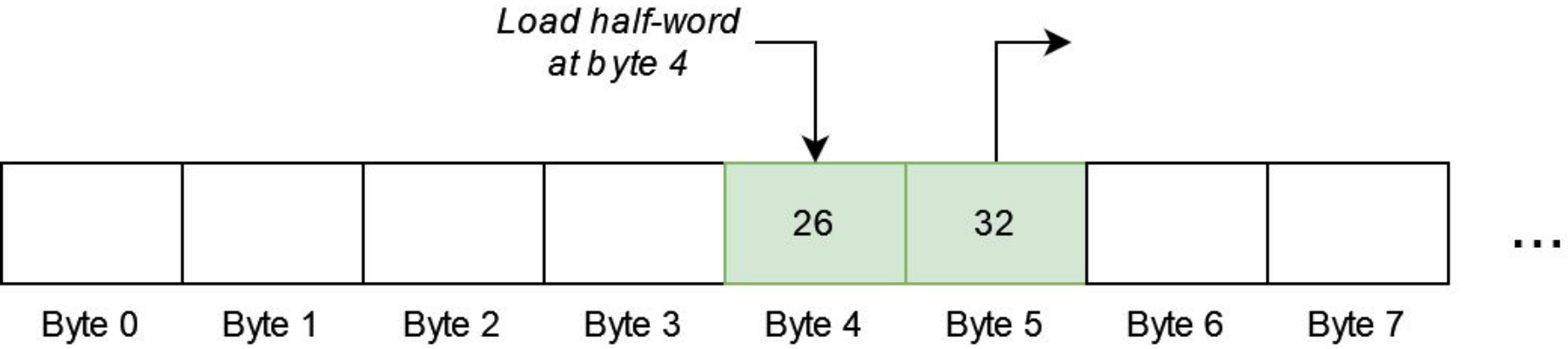
What be memory

- We mentioned you can think of it like a large 1D array
- Typically memory systems let us load and store bytes (not bits)
- Each byte (usually 8 bits) has a unique **address**
  - So memory can be thought of as one large array of bytes
  - Address = index into the array, e.g.:



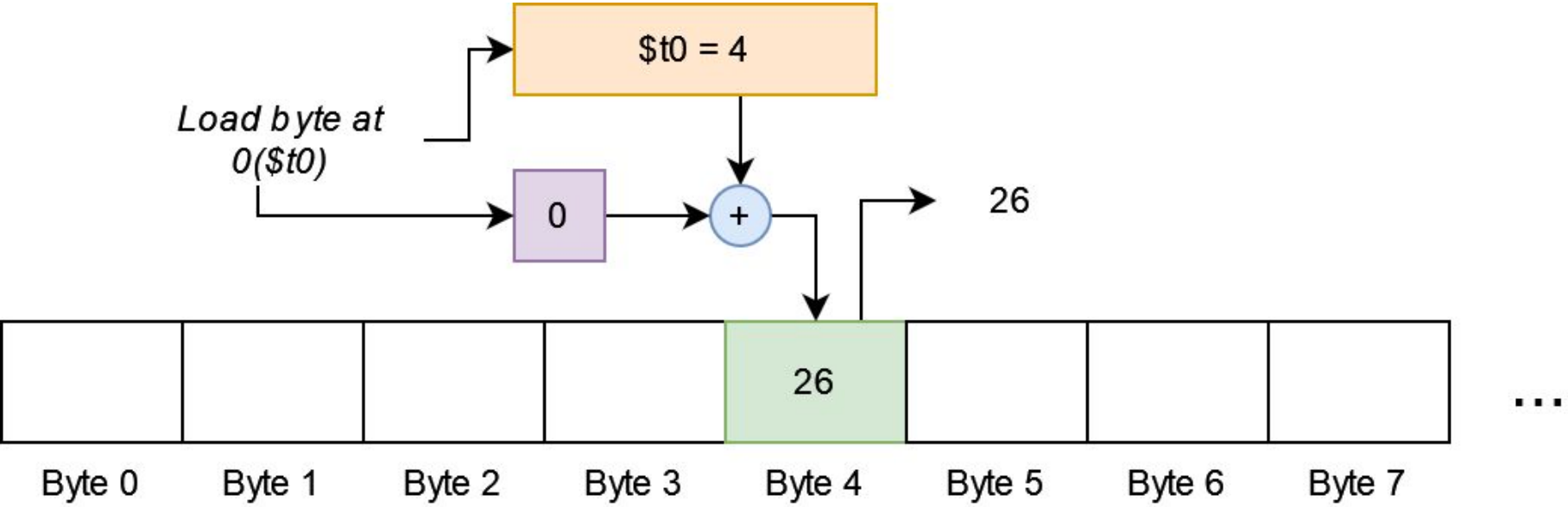
Bytes, half-words, words

- Typically, small groups of bytes can be loaded/stored at once
- E.g. in MIPS:
  - 1-byte (a byte) loaded/stored with ..... `lb/sb`
  - 2-bytes (a half-word) loaded/stored with..... `lh/sh`
  - 4-bytes (a word) loaded/stored with..... `lw/sw`



Memory addresses

- Memory addresses in load/store instructions are the sum of:
  - Value in a specific register
  - And a 16-bit constant (often 0)



Code example

- Storing and loading a value (no labels)

Code example

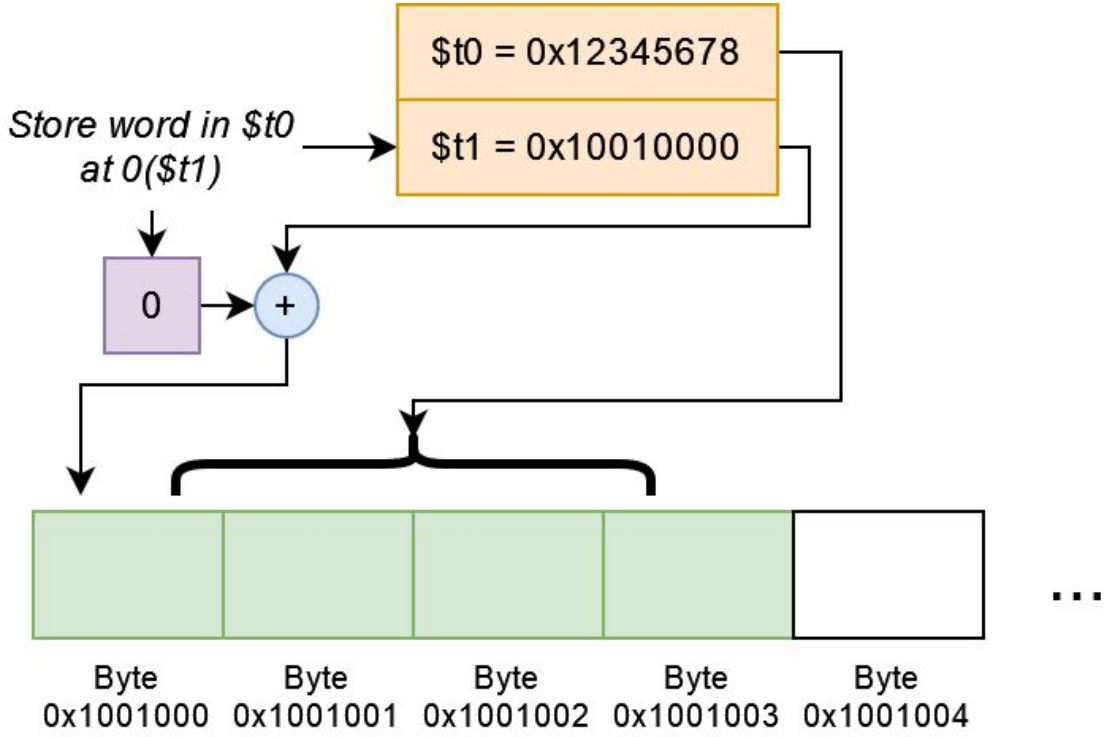
- Storing and loading a value (no labels)

```
.text
main:
        li $t0, 0x12345678
        la $t1, 0x10010000
        sw $t0, 0($t1)
.data
        .word 0
```

Code example

- Storing and loading a value (no labels)

```
.text
main:
        li $t0, 0x12345678
        la $t1, 0x10010000
        sw $t0, 0($t1)
.data
        .word 0
```



Code example

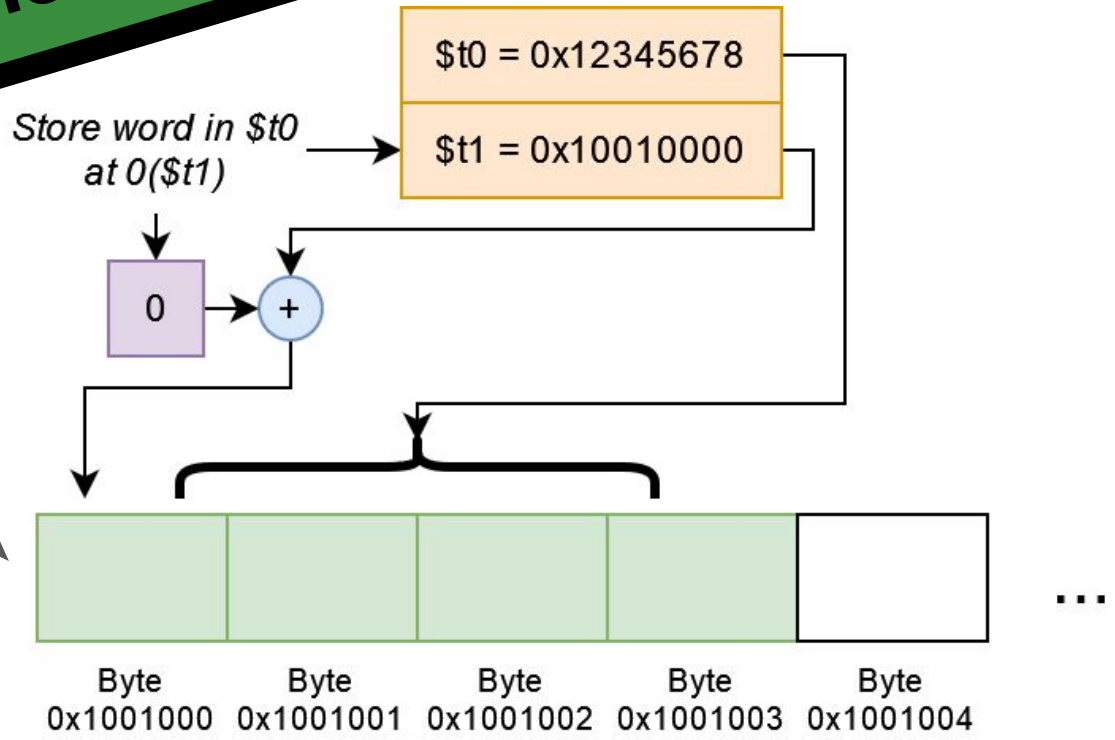
- Storing and l

What order will these bytes be?

```
        $t0, 0x12345678
        la $t1, 0x10010000
        sw $t0, 0($t1)

.data

        .word 0
```



New concept: Endian-ness

- “What order to put things in” is a hard question to answer

New concept: Endian-ness

- “What order to put things in” is a hard question to answer
- The answer is based on an egg

Which “end” of a boiled egg to break?

- “Endian” comes from the 1726 novel “Gulliver's Travels” by Jonathan Swift
- In the story, there is conflict between sects of Lilliputians divided into those breaking the shell of a boiled egg from the **big end** or from the **little end**.

Which “end” of a boiled egg to break?

- The difference between Big-Endians (break big end) and Little-Endians led to:
  - Six rebellions
  - One Emperor losing his life
  - Another his crown

- This was perhaps a commentary on something other than “byte” order

New concept: Endian-ness

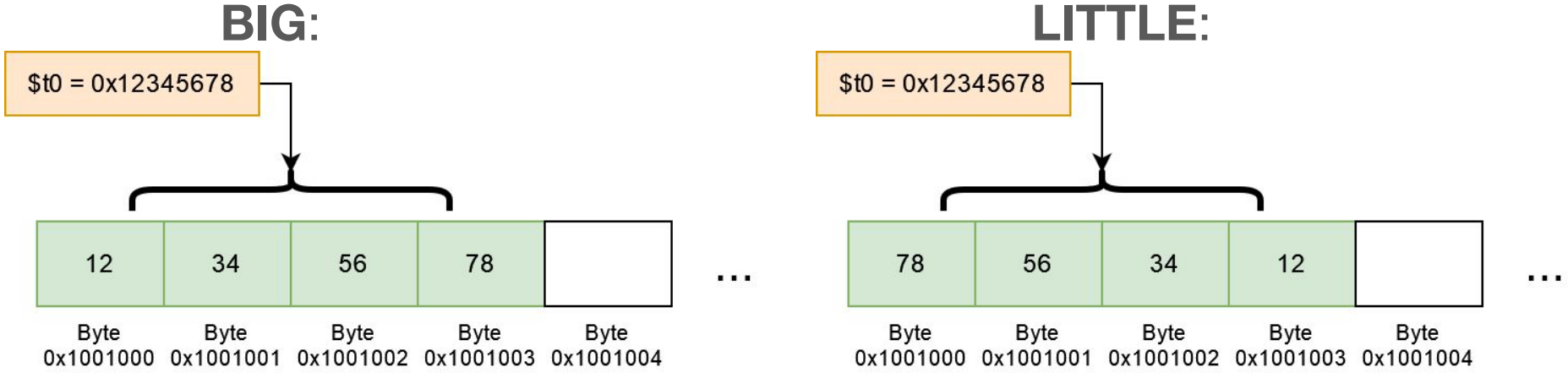
- “What order to put things in” is a hard question to answer

New concept: Endian-ness

- “What order to put things in” is a hard question to answer
- Two schools of thought:
  - **Big**-endian: MSB at the “low address” - big bits “first!”
  - **Little**-endian: MSB at the “high address” - big bits “last!”

New concept: Endian-ness

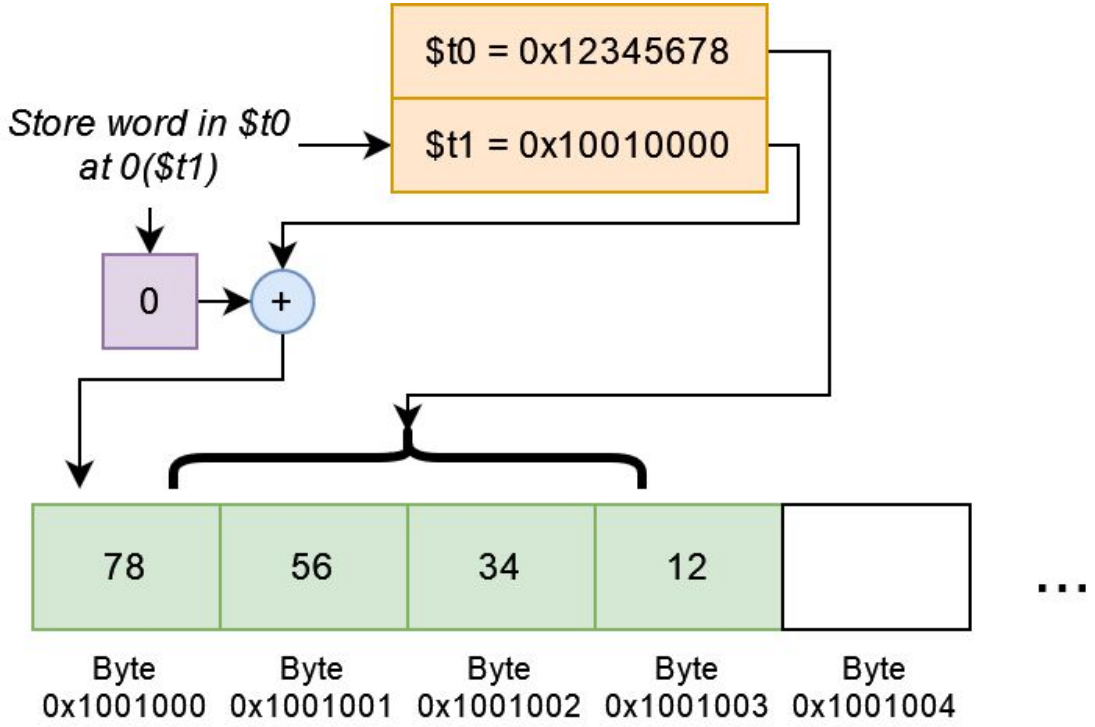
- "What order to put things in" is a hard question to answer
- Two schools of thought:
  - **Big**-endian: MSB at the "low address" - big bits "first!"
  - **Little**-endian: MSB at the "high address" - big bits "last!"



Code example

- Mipsy-web is **little-endian**

```
.text
main:
        li $t0, 0x12345678
        la $t1, 0x10010000
        sw $t0, 0($t1)
.data
        .word 0
```



Code example

- Storing and loading a value (labels)

Code example

- Storing and loading a value (labels)

```
.text
main:
        li $t0, 0x12345678
        la $t1, my_label
        sw $t0, 0($t1)
.data
my_label:
        .word 0
```

Bytes, half-words, words (part 2)

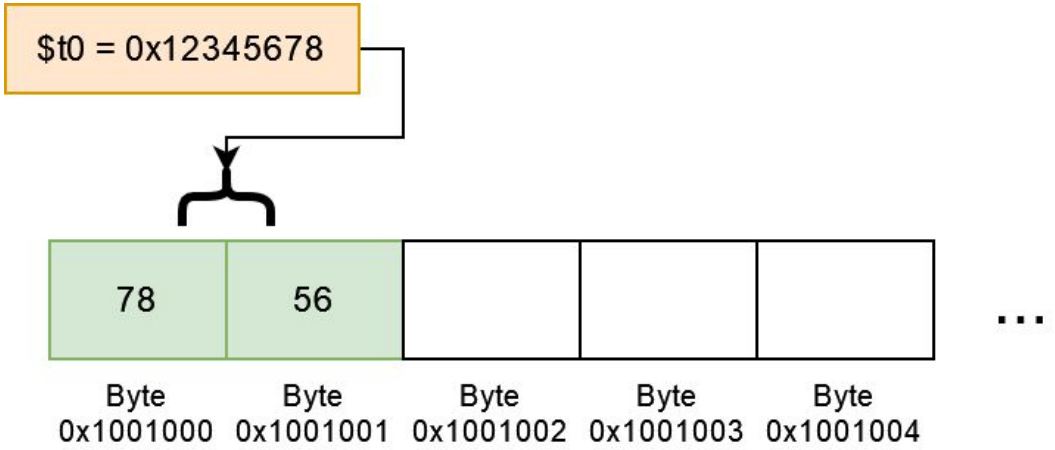
- **`sh/sb`** use the low (least-significant) bits of the source register
- **`lh/lb`** assume the loaded byte/halfword is signed
  - The destination register top bits are set to the sign bit
- **`lhu/lbu`** for doing the same thing, but unsigned

Examples

```
.text
main:
        li $t0, 0x12345678
        la $t1, my_label
        sh $t0, 0($t1)
.data
my_label:
        .word 0
```

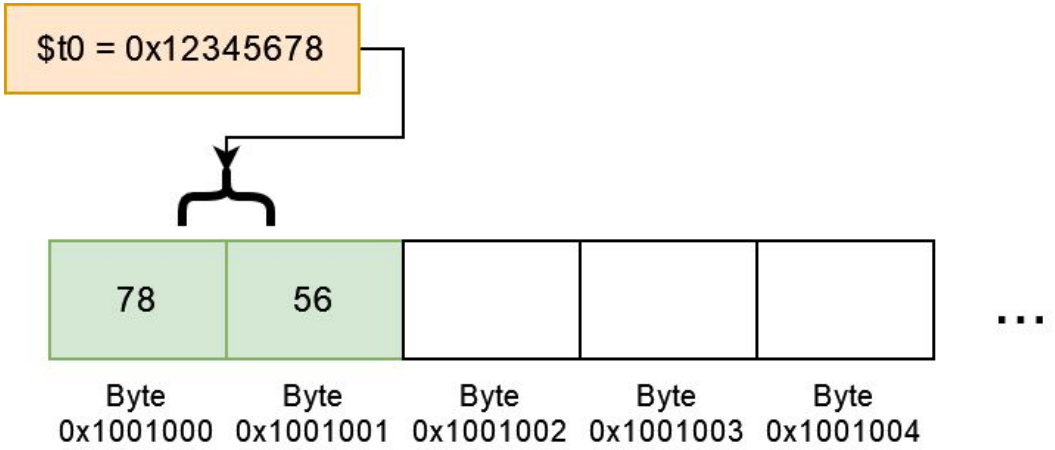
Examples

```
.text
main:
        li $t0, 0x12345678
        la $t1, my_label
        sh $t0, 0($t1)
.data
my_label:
        .word 0
```



Examples

```
.text
main:
        li $t0, 0x12345678
        la $t1, my_label
        sh $t0, 0($t1)
.data
my_label:
        .word 0
```

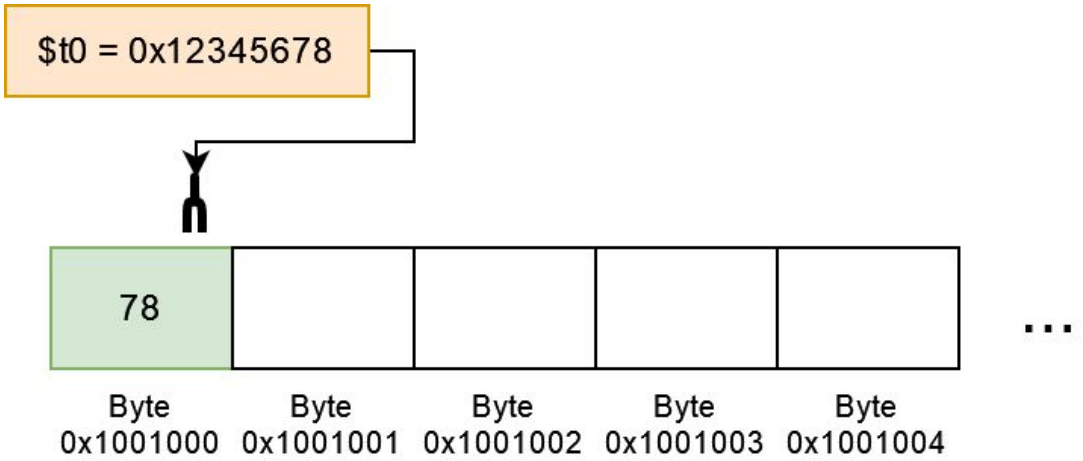


Examples

```
.text
main:
        li $t0, 0x12345678
        la $t1, my_label
        sb $t0, 0($t1)
.data
my_label:
        .word 0
```

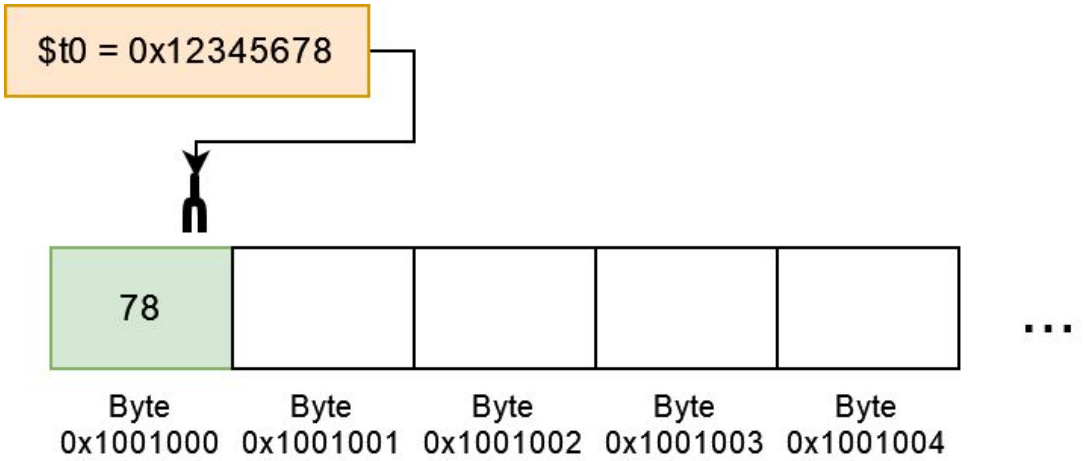
Examples

```
.text
main:
        li $t0, 0x12345678
        la $t1, my_label
        sb $t0, 0($t1)
.data
my_label:
        .word 0
```



Examples

```
.text
main:
        li $t0, 0x12345678
        la $t1, my_label
        sb $t0, 0($t1)
.data
my_label:
        .word 0
```

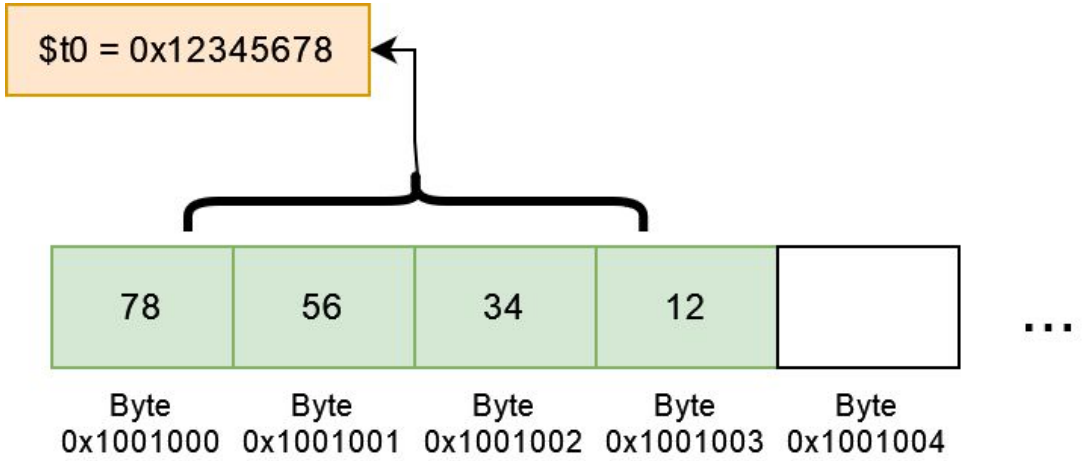


Loading Examples

```
.text
main:
        la $t1, my_label
        lw $t0, 0($t1)
.data
my_label:
        .word 0x12345678
```

Loading Examples

```
.text
main:
        la $t1, my_label
        lw $t0, 0($t1)
.data
my_label:
        .word 0x12345678
```

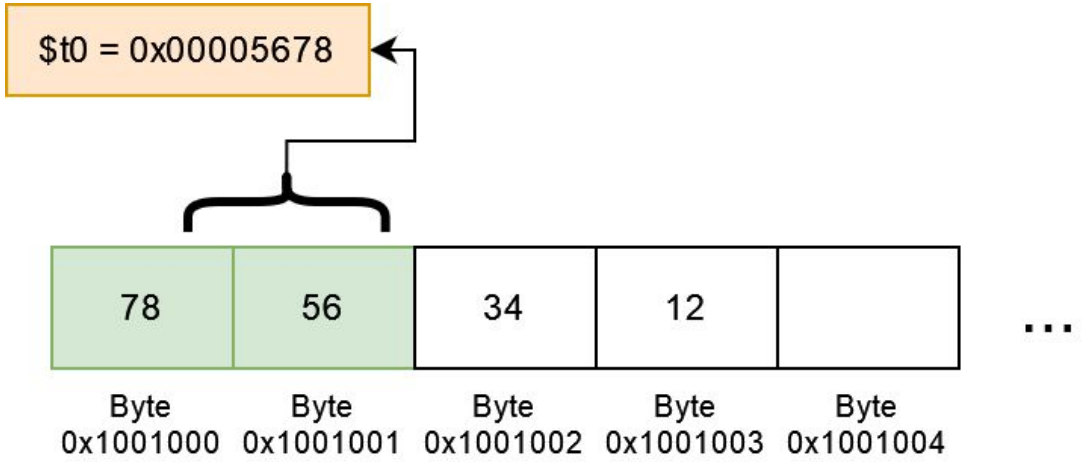


Loading Examples

```
.text
main:
        la $t1, my_label
        lh $t0, 0($t1)
.data
my_label:
        .word 0x12345678
```

Loading Examples

```
.text
main:
        la $t1, my_label
        lh $t0, 0($t1)
.data
my_label:
        .word 0x12345678
```

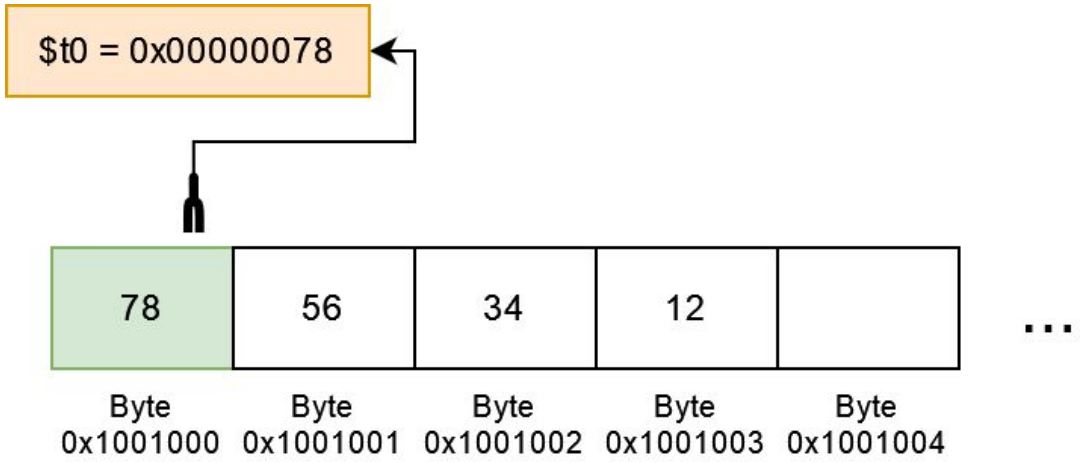


Loading Examples

```
.text
main:
        la $t1, my_label
        lb $t0, 0($t1)
.data
my_label:
        .word 0x12345678
```

Loading Examples

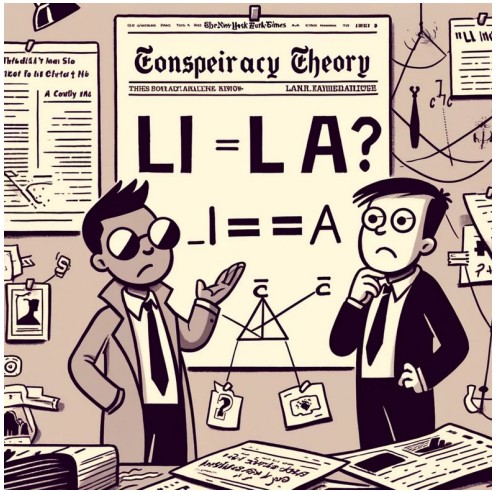
```
.text
main:
        la $t1, my_label
        lb $t0, 0($t1)
.data
my_label:
        .word 0x12345678
```



Setting registers to addresses

- Normally **`la`** is used to load addresses, **`li`** for data
- But this is just convention, and instructions don't actually differ
  - Both are also pseudo-instructions!
- These are all the same instruction! (assume my_label = 0x1001000)

```
li $t1, 0x10010000
li $t1, my_label
la $t1, 0x10010000
la $t1, my_label
```



- But, convention is still useful!

Mipsy-web helper pseudo-instruction

- We can just write constant memory address locations
- (We) don't need to load to another register

```
.text
main:
        li $t0, 0x12345678
        la $t1, my_label
        sw $t0, 0($t1)
.data
my_label:
        .word 0
```

=

```
.text
main:
        li $t0, 0x12345678
        sw $t0, my_label
.data
my_label:
        .word 0
```

Other assembler shortcuts

```
sb $t0, 0($t1)  # store $t0 in byte at address in $t1
sb $t0, ($t1)   # same


sb $t0, x       # store $t0 in byte at address labelled x
sb $t1, x+15    # store $t1 15 bytes past address labelled x
sb $t2, x($t3)  # store $t2 $t3 bytes past address labelled x
```

Demo program time - global_increment.c

- Let's write a program which has a global variable
- We will increment it

```c
#include <stdio.h>

int global_counter = 0;

int main(void) {
    // Increment the global counter.
    global_counter++;
    printf("%d", global_counter);
    putchar('\n');
}
```

Demo program time

```
.text
main:
        lw      $t1, global_counter
        addi    $t1, $t1, 1
        sw      $t1, global_counter     # global_counter = global_counter + 1;

        li      $v0, 1                  # syscall 1: print_int
        la      $t0, global_counter     #
        lw      $a0, ($t0)
        syscall                         # printf("%d", global_counter);

        li      $v0, 11                 # syscall 11: print_char
        li      $a0, '\n'
        syscall                         # putchar('\n');

        li      $v0, 0
        jr      $ra # return 0;
.data
global_counter:
        .word 0                         # int global_counter = 0;
```

C has lots of different types

- char ... as byte in memory, or register
- int ... as 4 bytes in memory, or register
- double ... as 8 bytes in memory, or $f? register
- arrays ... sequence of bytes, elements accessed by calculated index
- structs ... sequence of bytes in memory, accessed by constant offset fields

Demo - sizeof.c

Alignment

C standard requires simple types of size N bytes to be stored only at addresses which are divisible by N

- if int is 4 bytes, must be stored at address divisible by 4
- if ‘double is 8 bytes, must be stored at address divisible by 8
- compound types (arrays, structs) must be aligned so their components are aligned
- MIPS requires this alignment

Alignment problems demo - sample_data.s

```
.text

.data

a: .word 16             # int a = 16

b: .space 4             # int b;

c: .space 4             # char c[4];

d: .byte 1,2,3,4        # char d[4] = {1, 2, 3, 4};

e: .byte 0:4            # int8_t e[4] = {0};

f: .asciiz "hello"      # char *f = "hello";

g: .space 4             # int g;
```

Solutions?

Padding with .space

Alignment fix with .align

Demo program - array.c, array_bytes.c

Loop through an array

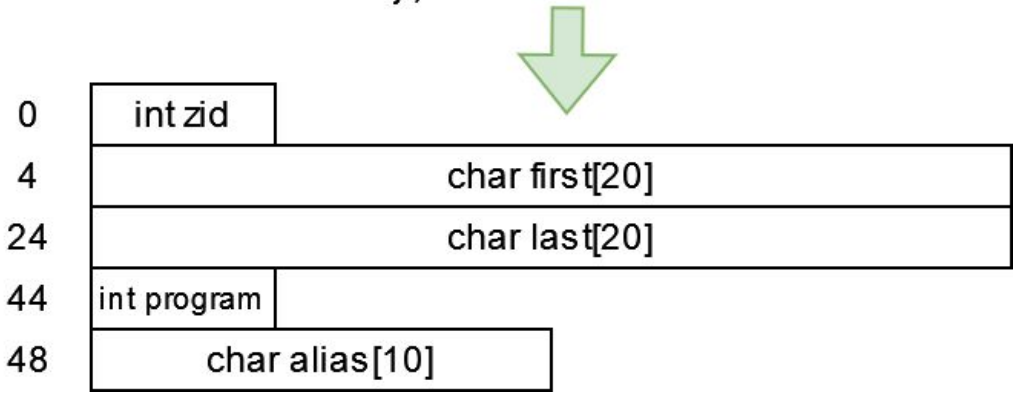
Demo program - flag.c

Loop through a 2D array

Structs!

- Struct values are really just sets of variables at known offsets
- E.g.

```
struct student {
 int zid;
 char first[20];
 char last[20];
  int program;
 char alias[10];
};
```



Demo program - struct.c

Stack variables vs globals?

A char, int or double:

- can be stored in register if local variable and no pointer to it
- otherwise stored on stack if local variable - we'll revisit this
- stored in data segment if global variable

This includes pointer addresses!

Mipsy assembler directives

```
.text                   # following instructions placed in text segment
.data                   # following objects placed in data segment

a: .space 18            # int8_t a[18];
.align 2                # align next object on 4-byte addr
i: .word 42             # int32_t i = 42;
v: .word 1,3,5          # int32_t v[3] = {1,3,5};
h: .half 2,4,6          # int16_t h[3] = {2,4,6};
b: .byte 7:5            # int8_t b[5] = {7,7,7,7,7};
f: .float 3.14          # float f = 3.14;
s: .asciiz "abc"        # char s[4] {'a','b','c','\0'};
t: .ascii "abc"         # char t[3] {'a','b','c'};
```