Bonus early-to-class Golf challenge

- Get two integers from the user A, B
- Add them to a constant 66 (to get A + B + 66)
- Print the sum
- **Use only “real” MIPS instructions (no pseudo-instructions)**
- Fewest total instructions “wins”

UNSW
SYDNEY

COMP1521 24T2 Lec04

MIPS: DATA (continued)

2024

Hammond Pearce

Inspired from Abiram’s Material

Lecture chat

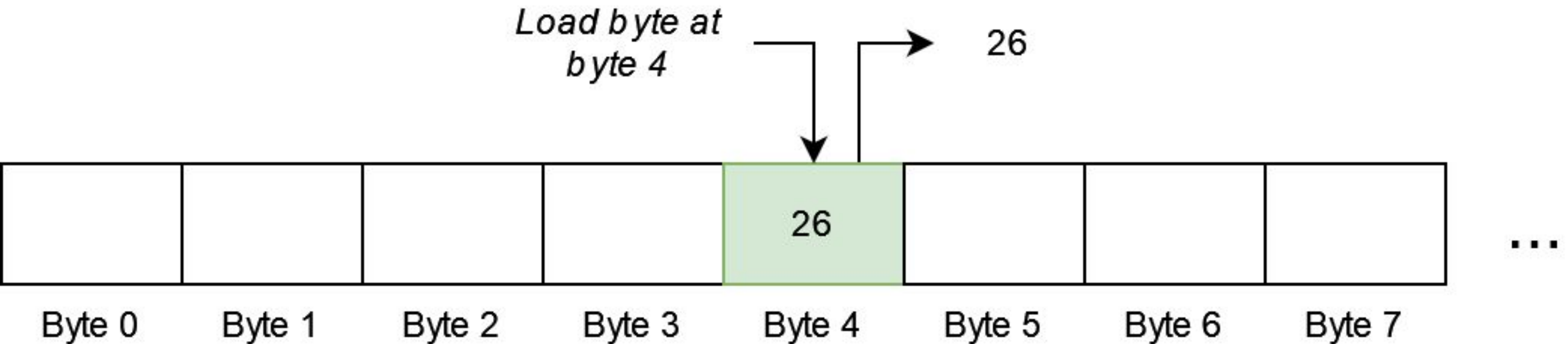
https://cgi.cse.unsw.edu.au/~cs1521/accord/

Recap of lec03

- Arrays and memory
- We'll pick up where we left off

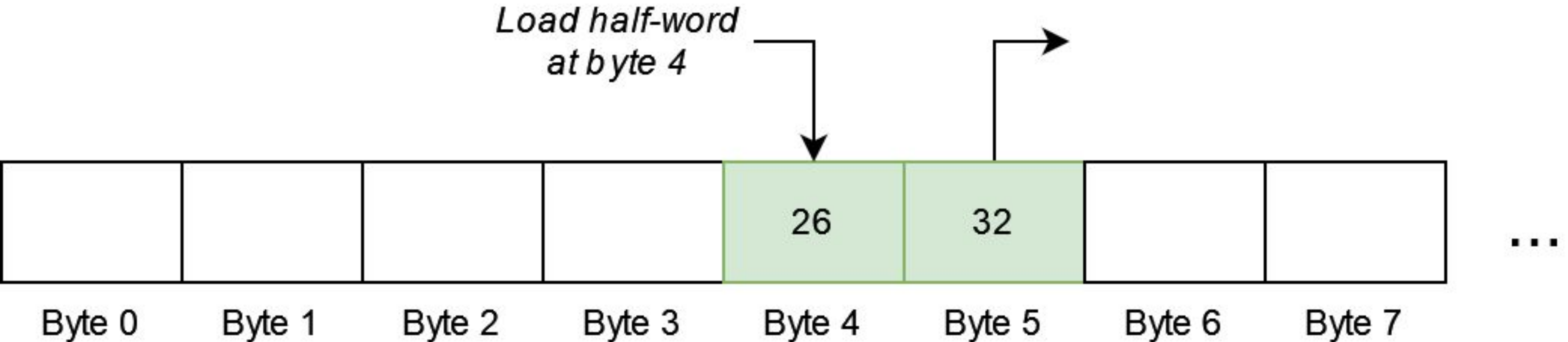
What be memory

- 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.:



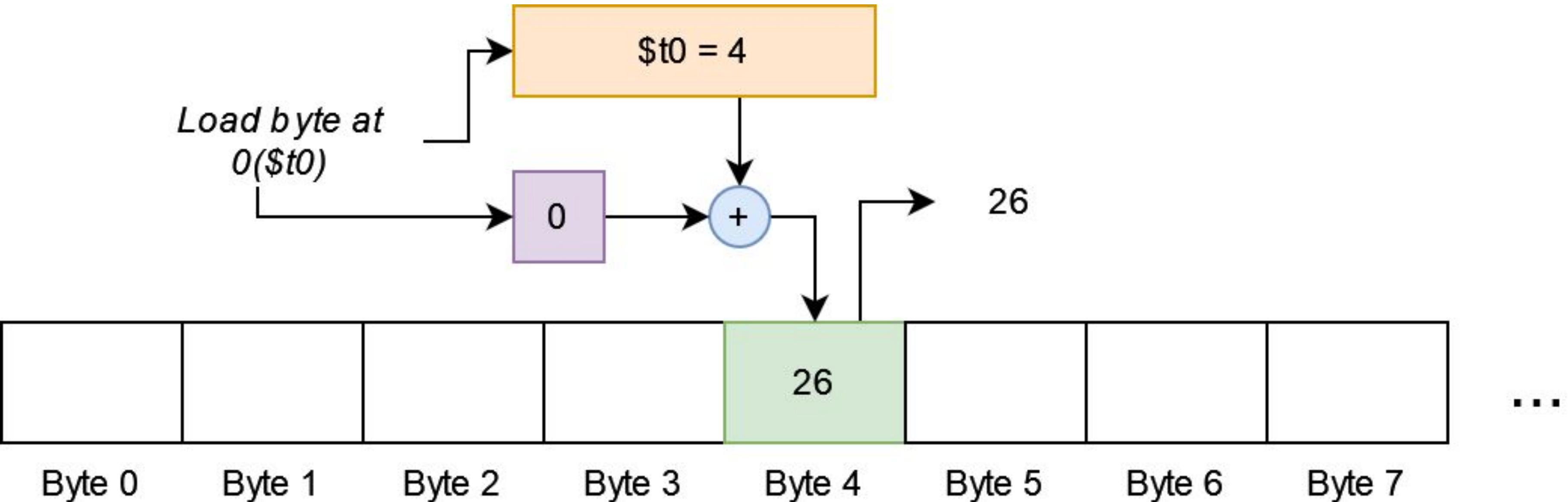
Bytes, half-words, words

- 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

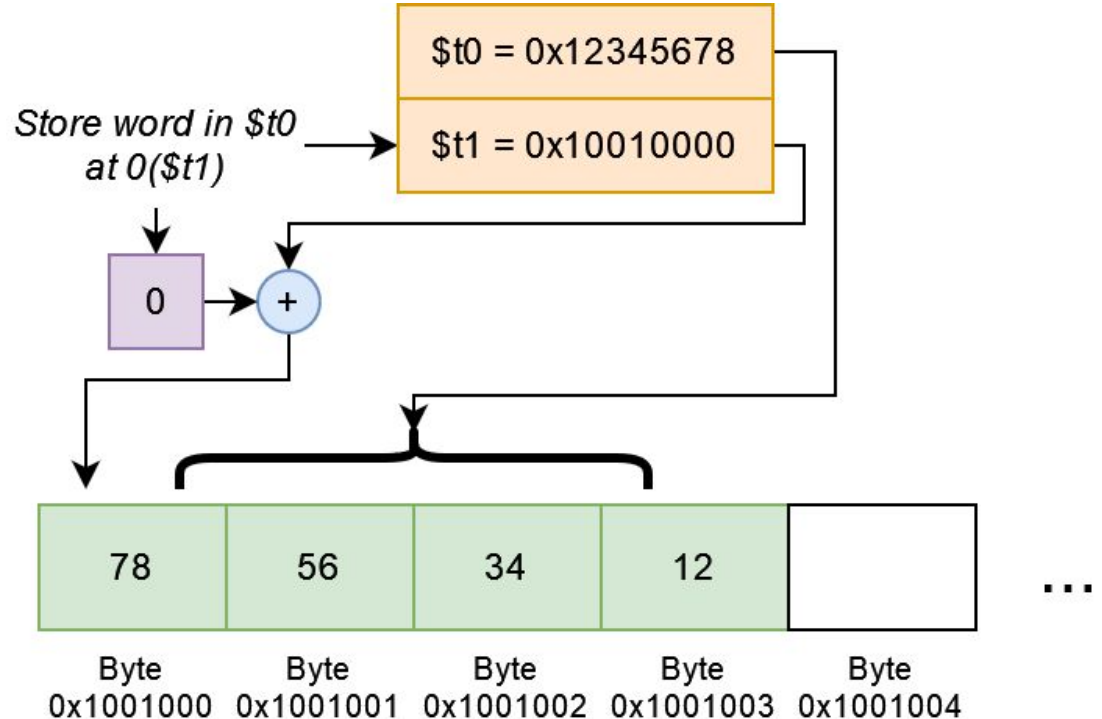
- Memory addresses in load/store instructions are the sum of:
  - Value in a specific register
  - And a 16-bit constant (often 0)



Code example

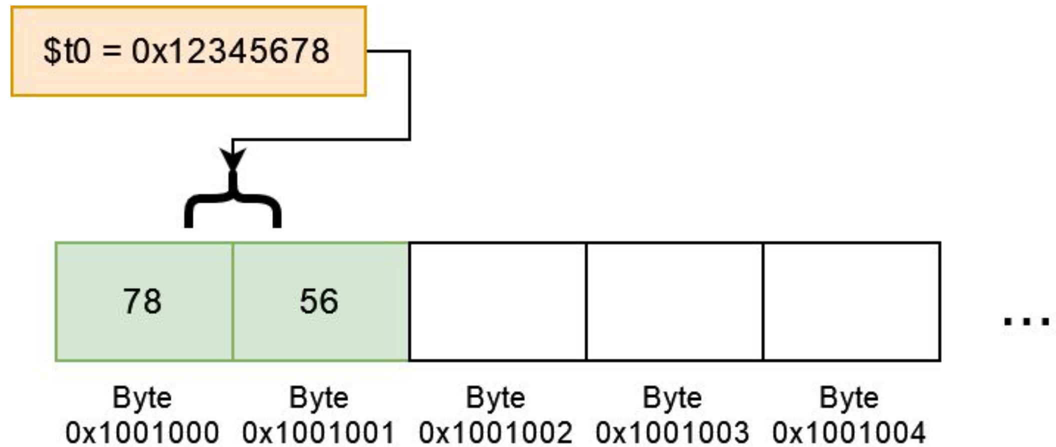
- Mipsy-web is **little-endian**

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



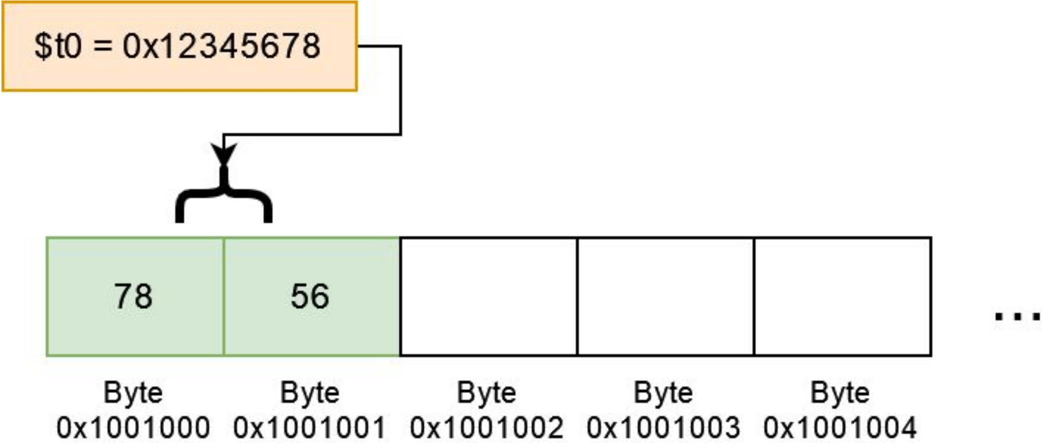
Examples

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



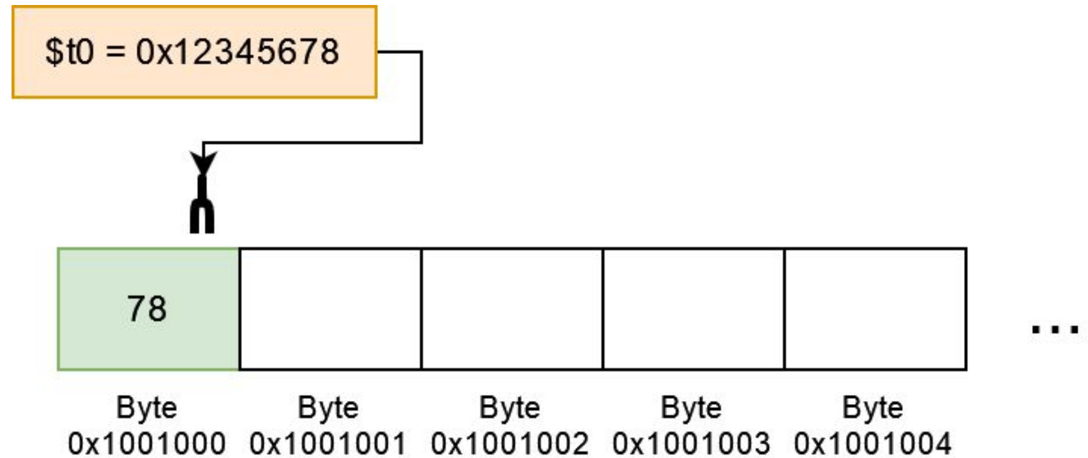
Examples

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



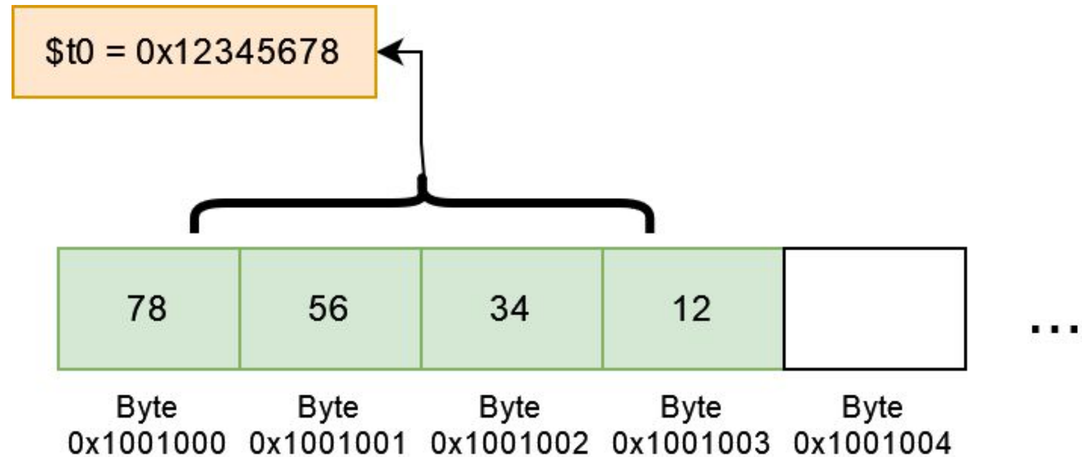
Examples

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



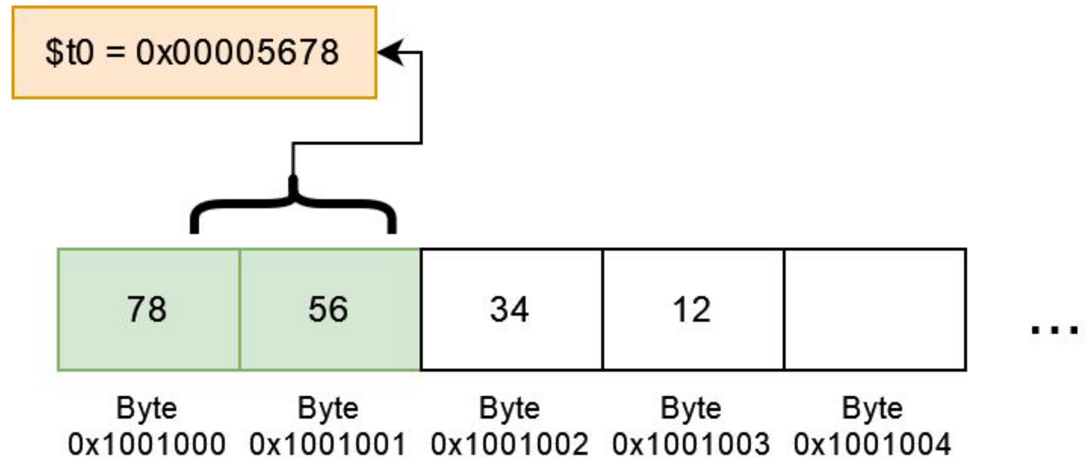
Loading Examples

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



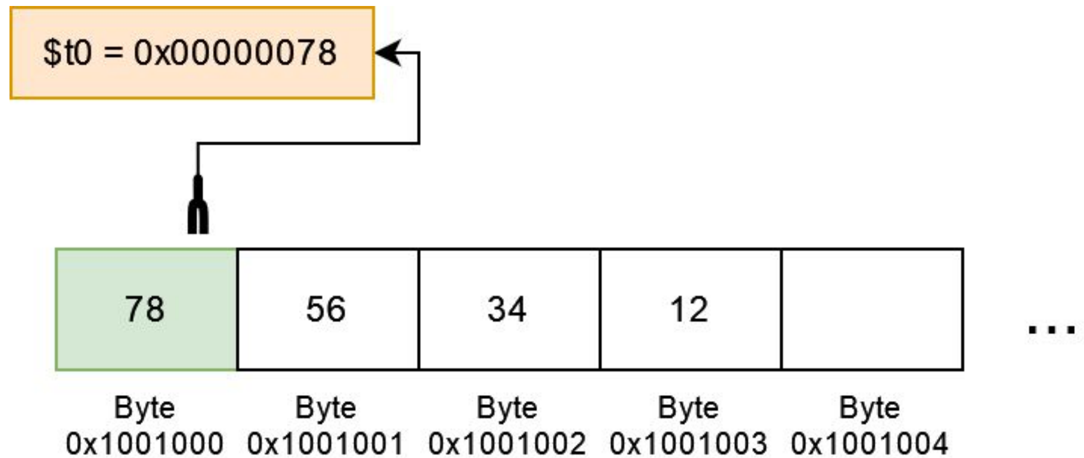
Loading Examples

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



Loading Examples

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



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
```

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

How do we find each element in memory?

We have:

```
char some_array[5] = {‘h’, ‘e’, ‘l’, ‘l’, ‘o’}
```

How do we compute `some_array[3]` in assembly?

How do we get the address of `some_array[3]`?

Demo program 2 - array_ints.c

Loop through an array of integers

How do we find each element in memory?

We have:

```
int some_int_array[5] = {3, 1, 4, 1, 5}
```

How do we compute `some_int_array[3]` in assembly?

How do we get the address of `some_int_array[3]`?

Base + (sizeof(int)*index)

Demo program - 2d.c, flag.c

Loop through a 2D array

`struct student students[2][5] = {{....}}`

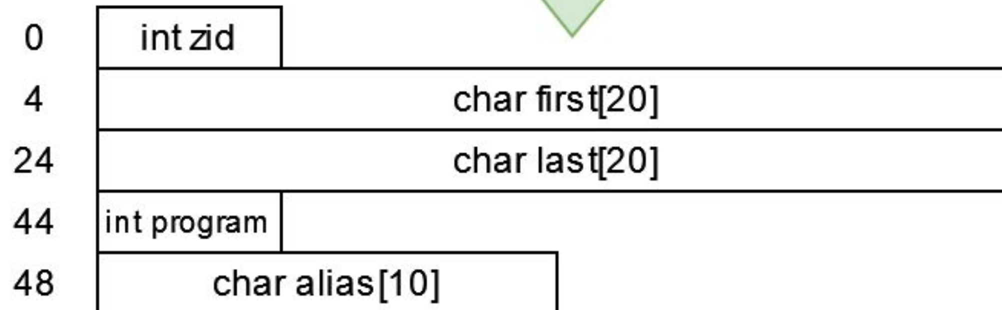
How do we compute **`some_int_array[1][3]`** in assembly?

How do we get the address of **`some_int_array[1][3]`**?

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'};
```