COMP1521 25T2

Week 2 Lecture 1

MIPS: Control and Data

Adapted from Angela Finlayson, Abiram Nadarajah, Hammond Pearce, Andrew Taylor and John Shepherd's slides

C revision sessions

- Anna Brew has kindly arranged for C revision sessions to take place Thu 12th June 10am-12pm via blackboard collaborate.

- More info on the forum under announcements
    - Week 2 Revision Lab + some resources for learning/revising C - Announcements - COMP1521

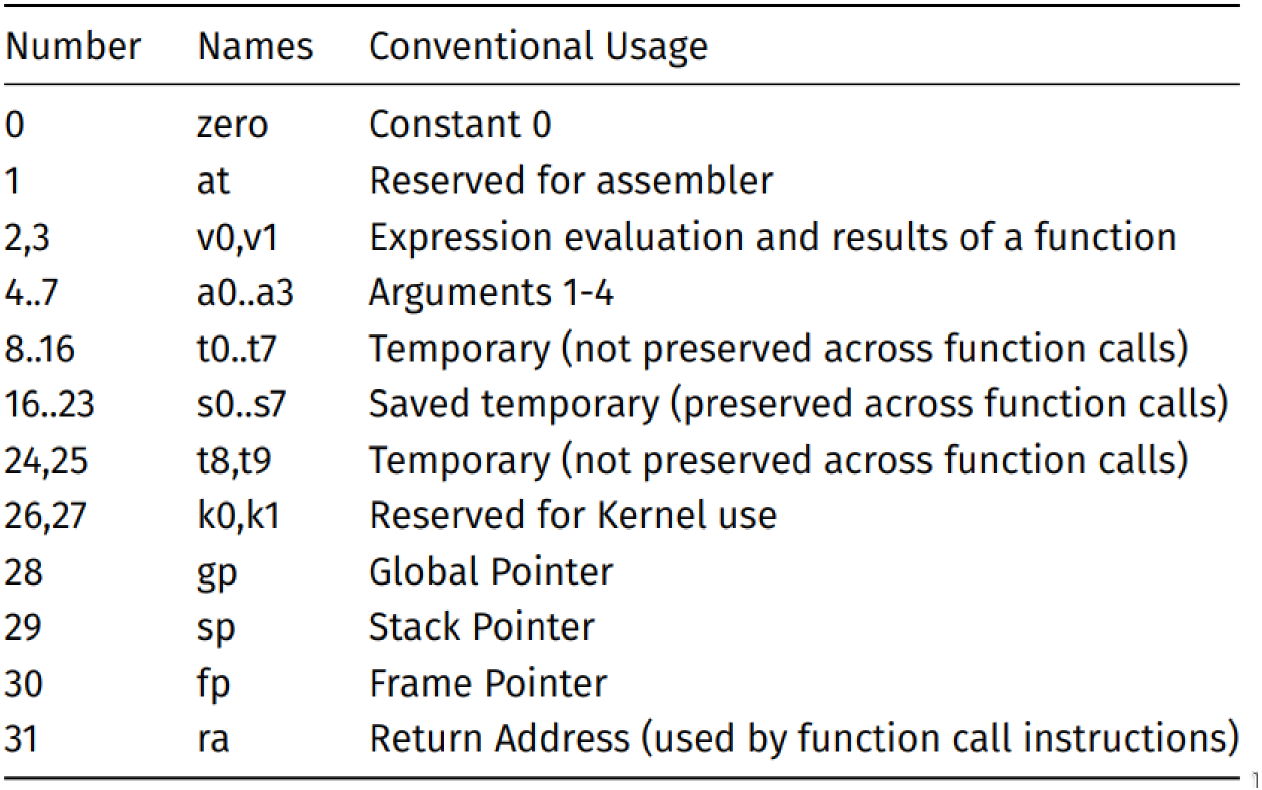
Today’s Lecture

- Recap Lecture 2
- More on loops
  - break and continue
- Data and Memory
  - Global variables
  - Pointers

Recap of the Last Lecture

- We can write instructions that act on **registers**
- We can write instructions that perform simple **arithmetic**
- We can `syscall` to pass control to the Operating System (OS)
- We can convert constructs like “loops” and “conditionals” into **goto** and **branch**

Recap: MIPS registers



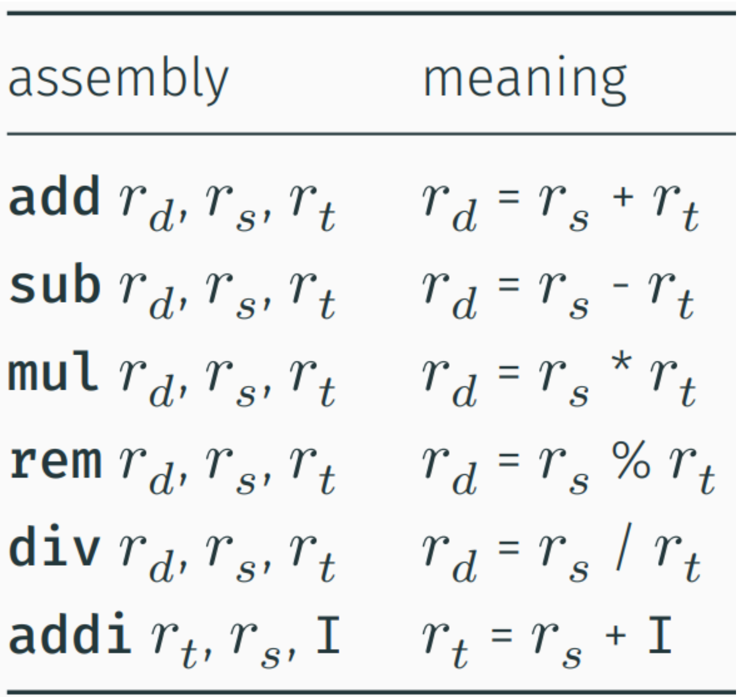
Number	Names	Conventional Usage
0	zero	Constant 0
1	at	Reserved for assembler
2,3	v0,v1	Expression evaluation and results of a function
4..7	a0..a3	Arguments 1-4
8..16	t0..t7	Temporary (not preserved across function calls)
16..23	s0..s7	Saved temporary (preserved across function calls)
24,25	t8,t9	Temporary (not preserved across function calls)
26,27	k0,k1	Reserved for Kernel use
28	gp	Global Pointer
29	sp	Stack Pointer
30	fp	Frame Pointer
31	ra	Return Address (used by function call instructions)

Recap: Putting data in registers

- **li** (load immediate) is loading a **fixed value** into a register
  - `li $t0, 7`
- **la** (load address) is for loading a **fixed address** into a register
  - remember, labels really just represent addresses!
  - `la $t0, my_label`
- **move** is for copying value from a **register** into another register
  - `move $t0, $t1`

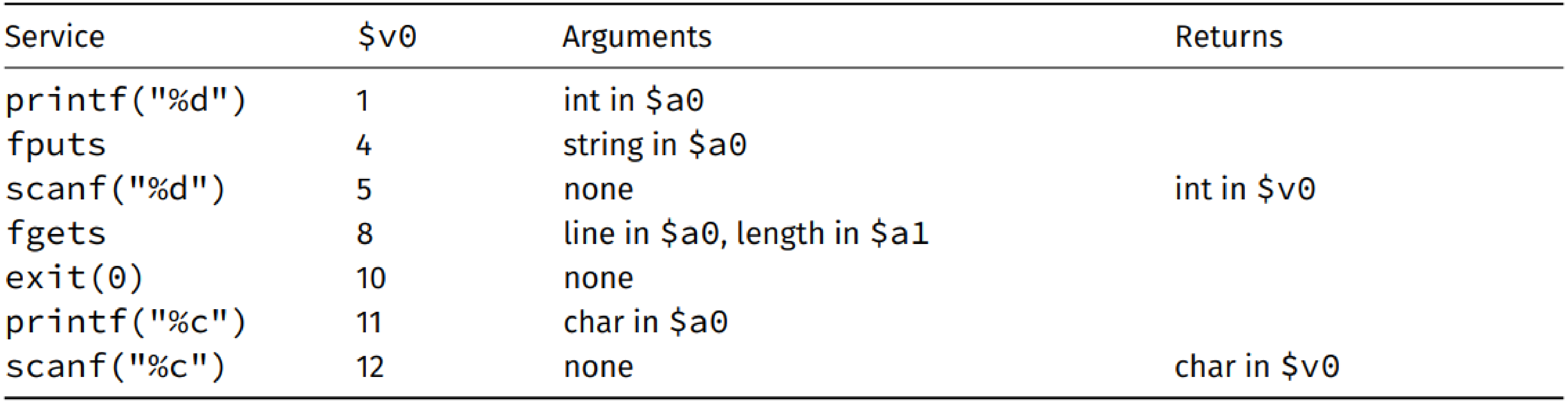
Recap: simple arithmetic

- I-type accepts immediate (constants in the instruction)
- R-type accepts registers



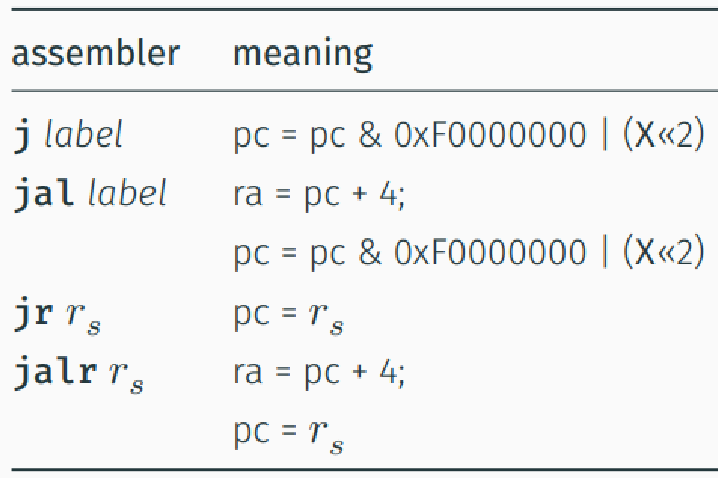
assembly	meaning
add r_d, r_s, r_t	$r_d = r_s + r_t$
sub r_d, r_s, r_t	$r_d = r_s - r_t$
mul r_d, r_s, r_t	$r_d = r_s * r_t$
rem r_d, r_s, r_t	$r_d = r_s \% r_t$
div r_d, r_s, r_t	$r_d = r_s / r_t$
addi r_t, r_s, I	$r_t = r_s + \mathrm{I}$

Recap: syscalls



Service	$v0	Arguments	Returns
`printf("%d")`	1	int in `$a0`	
`fputs`	4	string in `$a0`	
`scanf("%d")`	5	none	int in `$v0`
`fgets`	8	line in `$a0`, length in `$a1`	
`exit(0)`	10	none	
`printf("%c")`	11	char in `$a0`	
`scanf("%c")`	12	none	char in `$v0`

Recap: jump and branch

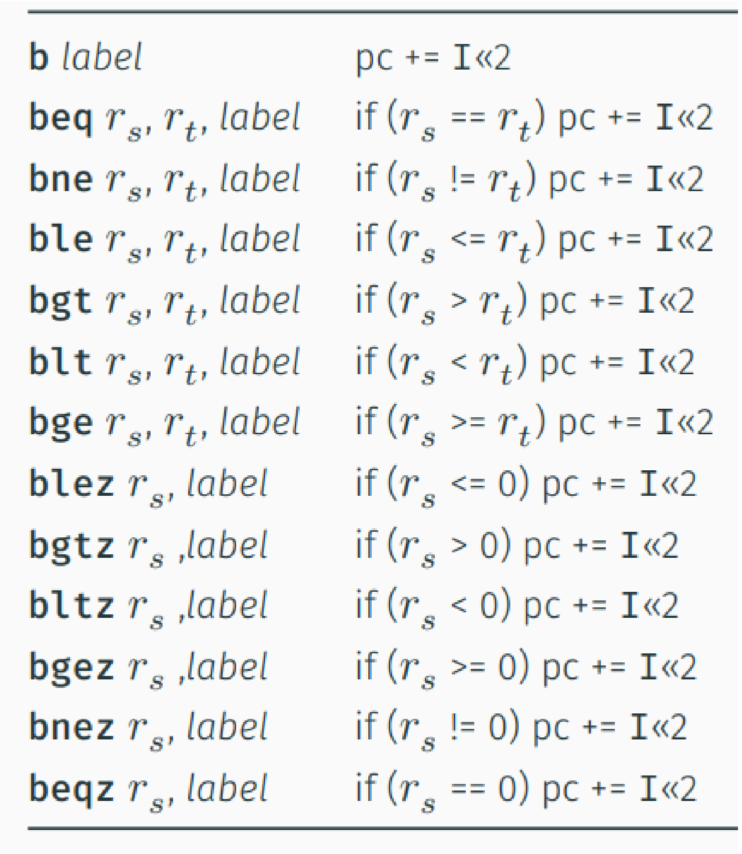


assembler	meaning
`j` *label*	pc = pc & 0xF0000000 | (X«2)

`jal` *label*	ra = pc + 4;
	pc = pc & 0xF0000000 | (X«2)

`jr` r_s	pc = r_s

`jalr` r_s	ra = pc + 4;
	pc = r_s



b *label*	pc += I«2
beq r_s, r_t, *label*	if (r_s == r_t) pc += I«2
bne r_s, r_t, *label*	if (r_s != r_t) pc += I«2
ble r_s, r_t, *label*	if (r_s <= r_t) pc += I«2
bgt r_s, r_t, *label*	if (r_s > r_t) pc += I«2
blt r_s, r_t, *label*	if (r_s < r_t) pc += I«2
bge r_s, r_t, *label*	if (r_s >= r_t) pc += I«2
blez r_s, *label*	if (r_s <= 0) pc += I«2
bgtz r_s ,*label*	if (r_s > 0) pc += I«2
bltz r_s ,*label*	if (r_s < 0) pc += I«2
bgez r_s ,*label*	if (r_s >= 0) pc += I«2
bnez r_s, *label*	if (r_s != 0) pc += I«2
beqz r_s, *label*	if (r_s == 0) pc += I«2

Recap: Simplified C

```c
int i = 0;
while (i < 10) {
    printf("%d\n", i);
    i++;
}
```

```c
int i;
loop_i_to_10__init:
    i = 0;
loop_i_to_10__cond:
    if (i >= 10) goto loop_i_to_10__end;

loop_i_to_10__body:
    printf("%d", i);
    putchar('\n');
loop_i_to_10__step:
    i = i + 1;
    goto loop_i_to_10__cond;
loop_i_to_10__end:

    // ...
```

Sidenote: C break

`break` can be used in a loop to exit the loop unconditionally.

The loop condition here makes this look like an infinite loop, but `break` means it’s possible to leave the loop

```c
while (1) {
    int c = getchar();
    if (c == EOF) break;
}
```

In simplified C, `break` is equivalent to going to the loop’s *end* label.

Sidenote: C break/continue

```
while (1) {
    int c = getchar();
    if (c == ‘n’) break;
}
```

```
    int c;
get_char_loop:
    c = getchar();
if_n:
    if (c == ‘n’)
        goto get_char_loop_end;
end_if_n:
        goto get_char_loop;

get_char_loop_end:
```

Sidenote: C continue

`continue` proceeds to the next iteration of a for loop.

This [terrible] code prints even numbers:

```c
for (int i = 0; i < 10; i++) {
    if (i % 2 != 0) continue;
    printf("%d\n", i);
}
```

In simplified C, `continue` is the equivalent to going to the loop’s *step* label.

MIPS: Data and Memory

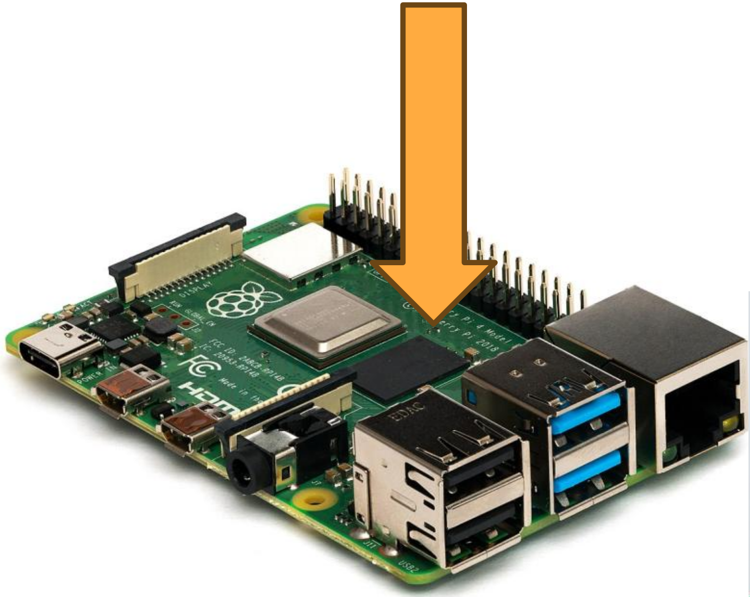
How do we store/use interesting data?

How does the data segment really work?

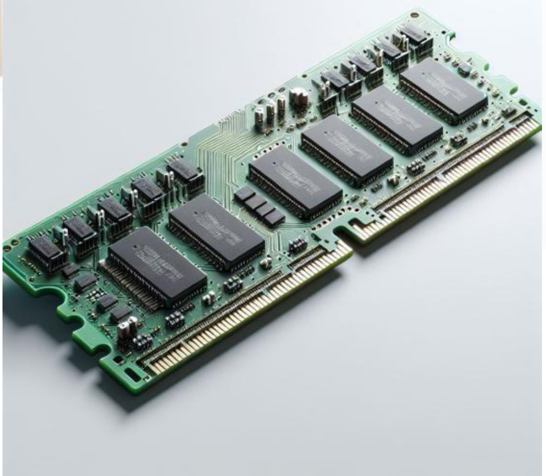
How do we:

- Store simple types like chars and ints?
- Store and increment a global variable?
- Work with pointers?
- Work with 1D arrays?
- Work with 2D arrays??
- C Structs !?

On-board RAM

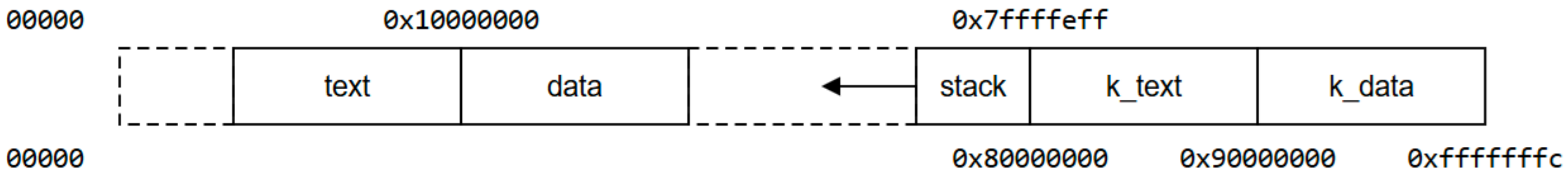


SO-DIMM RAM



What AI thinks RAM looks like…

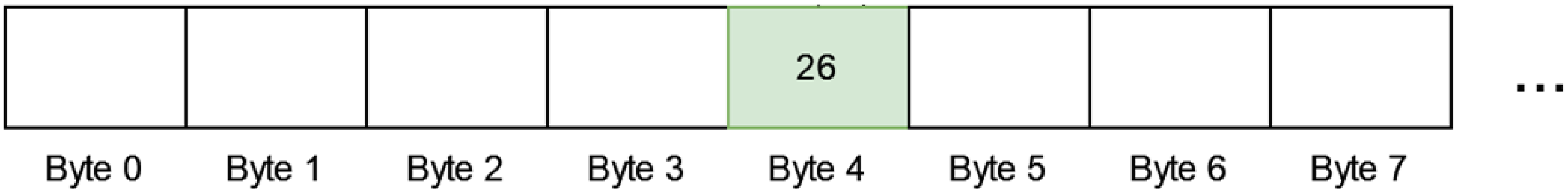
MIPS Memory Layout



- MIPS addresses are 32 bits (4 bytes)
- Notes:
  - There is no heap like in C, but the data segment can expand (not needed in this course except maybe challenge exercises)
  - The text segment is the only segment that is executable
  - The text segment is writable, unlike a real system

Memory Addresses

- Data will live at an address in memory
- We can think of it like a large 1D array
- 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
  - Eg. The byte at address 4 below has the value 26



				26				...
Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	

Common Data types in C

Note: sizeof(dtv) in C will return the *size*, in bytes, of the data type or variable named ‘dtv’.

Data Type	Size in Bytes	Location
`char`	1	Memory, Register
`int`	4	Memory, Register
`pointer`	4 (32 bit architectures)	Memory, Register
`array`	Sequence of basic type; elements accessed by calculated index	Memory
`struct`	Set of data types; elements accessed by calculated offset	Memory

Local vs Global variables in MIPS

Local Variables:

- Stored in **registers** (if possible) for speed:
- Otherwise stored on **stack** - we'll revisit this next week

Global Variables:

- Stored in the in **data segment**

Initialising Global Data

Using directives to initialise memory

```
    .word   42      # initialises a 4 byte value to 42
    .half   7       # initialises a 2 byte value to 7
    .byte   'a'     # initialises a 1 byte value to ‘a’
    .asciiz "hello" # initialises a string
```

We can also just ask for some memory without initialising it (typically we prefer to initialise it)

```
    .space 8        # set aside 8 uninitialised bytes
```

C equivalence

```
int a = 42;               // .word 42
short b = 7;              // .half 7
char c = 'a';             // .byte ‘a’
char d[6] = "hello";   // .asciiz "hello"

char space[8];            // .space 8

int main (void) {
    return 0;
}
```

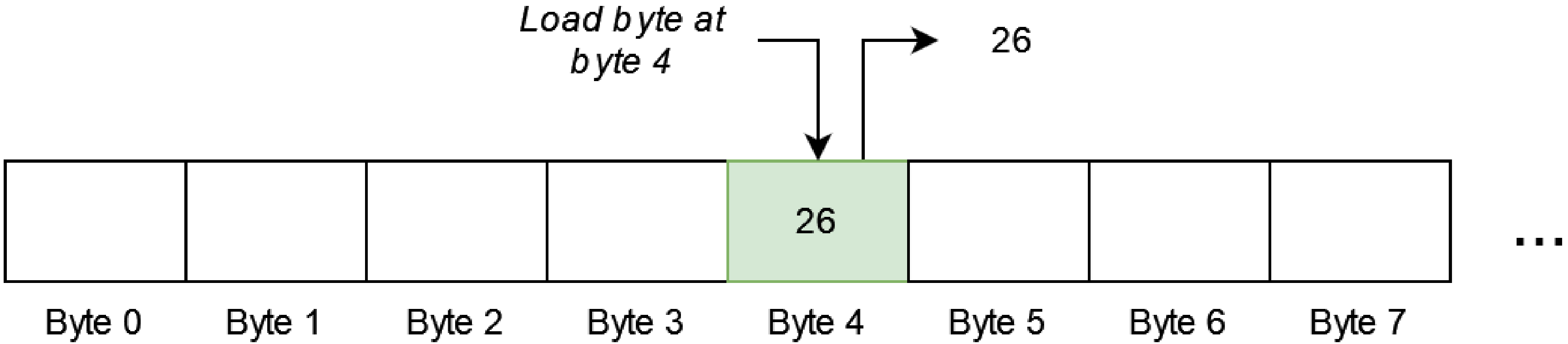
Accessing Memory

- Loading data:
  - To perform computations, data must first be transferred from memory into the CPU registers

- Storing data:
  - Modified data is written back from the CPU registers to memory

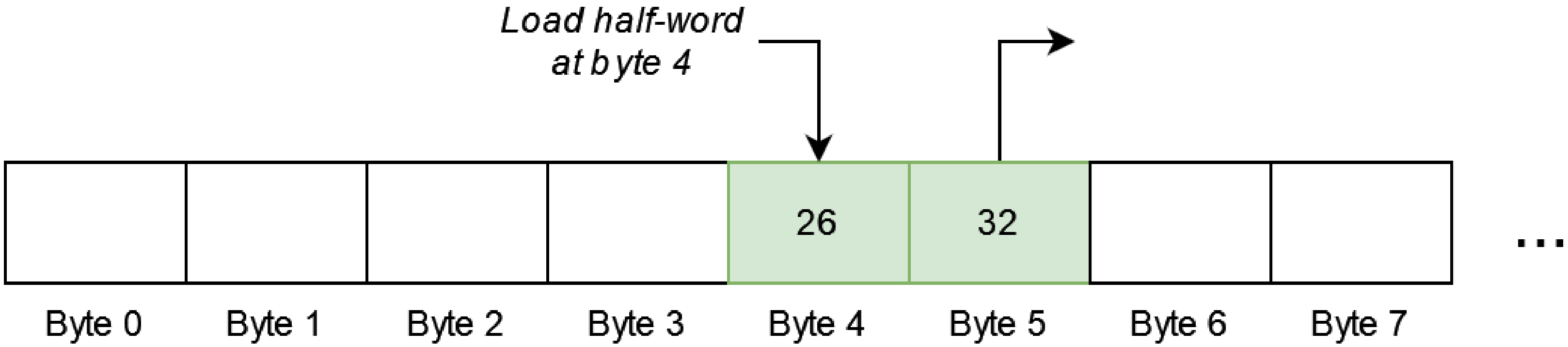
Loading from Memory

- E.g Loading the byte from address 4 would load the byte containing 26 in the specified register



Bytes, half-words, words

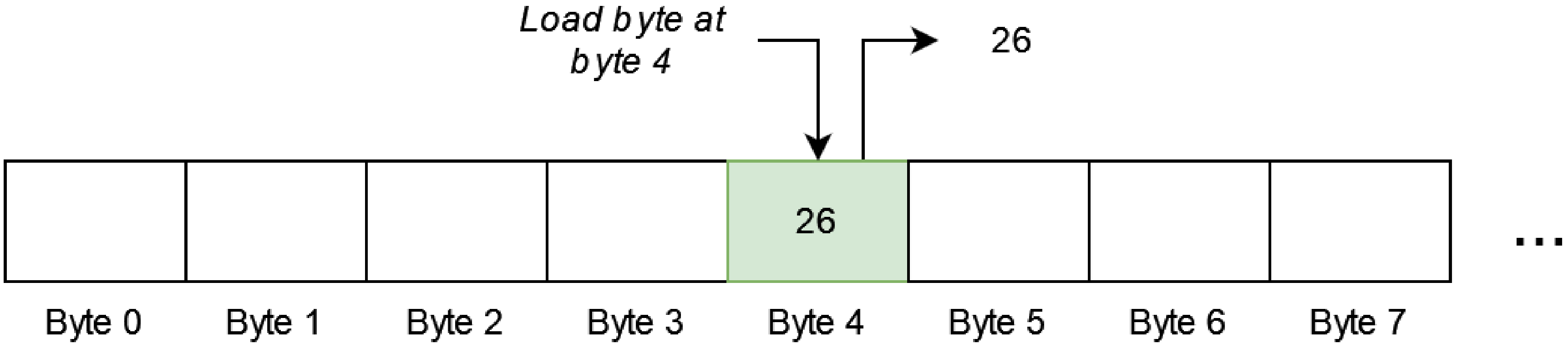
- Typically, small groups of bytes can be loaded/stored at once
- E.g. in MIPS:
    - 1-byte (a byte) loaded/stored with …………………………. `lb/sb`
    - 2-bytes (a half-word) loaded/stored with…………………… `lh/sh`
    - 4-bytes (a word) loaded/stored with……………………….. `lw/sw`



Working with Memory Addresses in MIPS

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

    -
      ```
      la      $t0, 4
      lb     $t1, 0($t0)
      ```



Loading/Storing a byte from/to Memory

Loading a byte (no labels)

```
        .text
main:
        la      $t1, 0x10010000
        lb      $t0, 0($t1)

        .data
        .byte 'Q'
```

Storing a byte (no labels)

```
        .text
main:
        li      $t0, 'y'
        la      $t1, 0x10010000
        sb      $t0, 0($t1)
```

Labels

- We do NOT want to keep track of the memory locations and hard code them ourselves.
- What if we add/remove variables as we develop our code?
- We use labels which are used by the assembler to represent the memory locations.

Loading/Storing a byte from/to Memory

loading a byte with labels

```
        .text
main:
        la      $t1, my_letter
        lb      $t0, 0($t1)

        .data
my_letter:
        .byte 'Q'
```

storing a byte with labels

```
        .text
main:
        li      $t0, 'y'
        la      $t1, my_letter
        sb      $t0, 0($t1)

        .data
my_letter:
        .space 1
```

Loading/Storing a word from/to Memory

loading a word

```
        .text
main:
        la      $t1, my_word
        lw      $t0, 0($t1)

        .data
my_word:
        .word 10
```

storing a word

```
        .text
main:
        li      $t0, 9
        la      $t1, my_word
        sw      $t0, 0($t1)

        .data
my_word:
        .space 4
```

Mipsy short cuts

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

```
        .text
main:
        li $t0, 42
        la $t1, my_label
        sw $t0, 0($t1)
        .data
my_label:
        .word 0
```

=

```
        .text
main:
        li $t0, 42
        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 with a global variable and 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');
}
```

Alignment

- C standard and MIPS requires simple types of size N bytes to be stored only at addresses which are divisible by N
  - a 4 byte int , must be stored at address divisible by 4
  - an 8 byte double, must be stored at address divisible by 8
  - Compound types (arrays, structs) must be aligned so their components are aligned
- Example:
  - If you are using lw, or sw, you must be loading/storing the 4 bytes from/to an address divisible by 4

Alignment problem demo - sample_data.s

```
        .text
main:
        li      $t0,    99
        sw      $t0,    g               # g = 99

        li      $v0, 0                  # return 0
        jr      $ra

        .data
f:              .asciiz "hello"         # char f[] = "hello";
g:              .space 4                # int g;
```

Alignment Solutions

Padding with `.space`

```
    .data
f:  .asciiz "hello"         # char f[] = "hello";
    .space 2                # padding - we have to calculate the space
                            # ourselves. Error prone. May break if we modify
                            # our string “hello”
g:  .space 4                # int g;
```

Alignment fix with **.align**

```
    .data
f:  .asciiz "hello"         # char f[] = "hello";
    .align 2                # align next object on 4 byte address (2 pow 2)
                            # (2 to the power of 2) less error prone
g:  .space 4                # int g;
```

Pointer Example

```c
int answer = 42;

int main(void) {
    int i;
    int *p;
    p = &answer;
    i = *p;
    printf("%d\n", i);
    *p = 27;
    printf("%d\n", answer);
    return 0;
}
```

What would this print?

How could we write this in MIPS?

Dealing with ISA extensions

- Suppose a CPU is released with an extension to the MIPS ISA.
  - Suppose syscall is a new instruction
  - Suppose an assembler that understands the encoding of syscall has not yet been released!

Dealing with ISA extensions

- Directives are not limited to the .data section

```
        .text
main:
        li $a0, 42       # Prepare 42
        li $v0, 1        # 1 is the syscall for print_int
        .word 0x0000000C # syscall instruction

        li $v0, 0         # Return 0
        jr $ra
```

- You could write your entire program in machine code! (not recommended…)

What did we learn today?

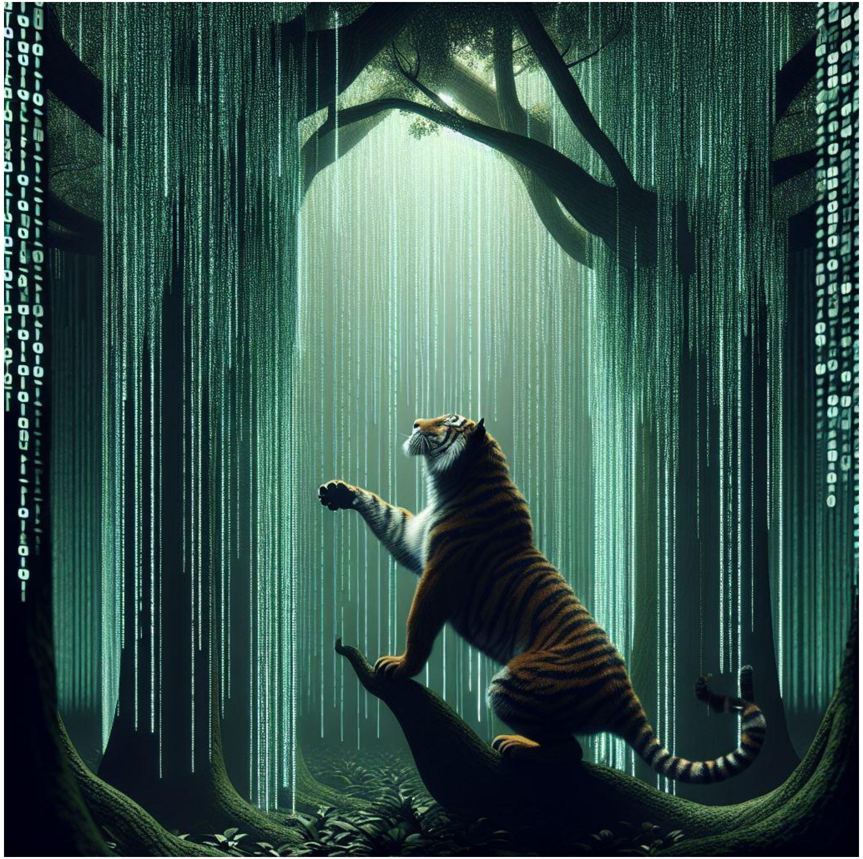
- MIPS
    - Recap of if statements
    - Loops
    - MIPS Data
        - loading and storing data
        - ints, chars, pointers
        - Alignment

- Next lecture:
    - 1D Arrays, 2D arrays (twice the fun), structs

Reach Out

Content Related Questions:
Forum

Admin related Questions email:
cs1521@cse.unsw.edu.au



Student Support | I Need Help With…

My Feelings and Mental Health

Managing Low Mood, Unusual Feelings & Depression

Mental Health Connect — student.unsw.edu.au/**counselling**
Telehealth

Mind HUB — student.unsw.edu.au/**mind-hub**
Online Self-Help Resources

In Australia Call Afterhours UNSW Mental Health Support Line — 1300 787 026
5pm-9am

Outside Australia Afterhours 24-hour Medibank Hotline — +61 (2) 8905 0307

Uni and Life Pressures

Stress, Financial, Visas, Accommodation & More

Student Support
Indigenous Student Support — student.unsw.edu.au/**advisors**

Reporting Sexual Assault/Harassment

Equity Diversity and Inclusion (EDI) — edi.unsw.edu.au/**sexual-misconduct**

Educational Adjustments

To Manage my Studies and Disability / Health Condition

Equitable Learning Service (ELS) — student.unsw.edu.au/**els**

Academic and Study Skills

Academic Language Skills — student.unsw.edu.au/**skills**

Special Consideration

Because Life Impacts our Studies and Exams

Special Consideration — student.unsw.edu.au/**special-consideration**