COMP1521
TUTORIAL 3

MIPS DIRECTIVES -> MIPS MEMORY -> MIPS ARRAYS

Assignment 1

- Start early!! Help sessions which start this week are always quite empty early on, so take advantage of the extra help
- Don’t ignore style (I’ll go more in-depth next week), but it is work 20% of the assignments mark!
  - You get 12 of the 20 simply just filling out some comments (easiest marks ever)

MIPS DIRECTIVES

MIPS Memory Directives

- So far, we've only really managed data in MIPS through using registers, using our li, la, and move instructions, and the only time we think the .data section is for passing the address of a string.

- So how do we manipulate this data? As you might have seen with strings (.asciiz) is another directive we use to define our memory layout within the data section

```
.data
    string: .asciiz "Hello World!\n"
```

- There are many directives to define different size data and arrays for us to manipulate from the .data section

MIPS Memory Directives

These 3 can take one or more arguments

Directive	What it does
.space [number]	Reserves [number] bytes of uninitialised memory
.byte [value], ...	Stores [value] within ONE byte
.half [value], ...	Stores [value] within TWO bytes
.word [value], ...	Stores [value] within FOUR bytes
.asciiz ["string"]	Stores a null terminated ascii [string] in memory
.align [1 or 2 or 3]	Align the next directive to an address divisible by 2, 4 or 8

```
.data
    empty:  .space 16
    num:    .word 42
```

MIPS Memory Directives With Multiple Arguments

- As alluded to before .byte, .half and .word can be used to store multiple arguments, can you imagine how this could be useful? Arrays!

- Mipsy can do this in two ways:
  - A list of args
    - Label: .directive arg1, arg2, arg3, … (as many args as you need)

```
.data
    array: .word 1, 2, 3, 4, 5, 6
```

  - The value to initialise, and how many times
    - Label: .directive, value:number

```
.data
    array: .word 0:6
```

```
int array[6] = {0};
```

MIPS Memory Directives

- The memory of a MIPS program is mapped sequentially from the .data section, this means the order they appear in the program is the order in memory.
- And directives such as .word and .half will always be aligned to an address divisible by 4 and 2 respectively, this will even have padded memory if necessary

```
    .data
one:    .byte 'a'
two:    .word 3
three:  .byte 'b'
four:   .half 5
```

Data segment

0x10010000	61	?	?	?	a	?	?	?
0x10010004	03	00	00	00	.	\0	\0	\0
0x10010008	62	?	05	00	b	?	.	\0

Hexadecimal (short summary)

- As massive binary numbers are difficult to work with (1's and 0's), we often use hexadecimal values which is a base-16 number system that serves as a human-friendly shorthand for binary.
- If a number is to start with 0x it means we are working with a hexadecimal value, and most notably you'll notice most addresses are written this way

154 = 1 * 10^2 + 5 * 10^1 + 4 * 10^0 = 154
0x154 = 1 * 16^2 + 5 * 16^1 + 4 * 16^0 = 340

Decimal	Hexadecimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

Example: q2

- If the data segment of a particular MIPS program starts at the address 0x10010020, then what addresses are the following labels associated with, and what value is stored in each 4-byte memory cell?

```
    .data
a:  .word   42
b:  .space  4
c:  .asciiz "abcde"
    .align  2
d:  .byte   1, 2, 3, 4
e:  .word   1, 2, 3, 4
f:  .space  1
```

Label	Address	Contents
	0x10010020	

Example: q2

- If the data segment of a particular MIPS program starts at the address 0x10010020, then what addresses are the following labels associated with, and what value is stored in each 4-byte memory cell?

```
.data
a:   .word    42
b:   .space   4
c:   .asciiz  "abcde"
     .align   2
d:   .byte    1, 2, 3, 4
e:   .word    1, 2, 3, 4
f:   .space   1
```

Label	Address	Contents	Layout
a	0x10010020	42	0x0000002A
b	0x10010024	????	0x????????
c	0x10010028	'a', 'b', 'c', 'd'	0x61626364
	0x1001002C	'e', '\0', ?, ?	0x6500????
d	0x10010030	1, 2, 3, 4	0x01020304
e	0x10010034	1	0x00000001
	0x10010038	2	0x00000002
	0x1001003C	3	0x00000003
	0x10010040	4	0x00000004
f	0x10010044	?	0x????????

Example: q3

- Give the MIPS directives to represent the following variables
  - int u;
    - u: .space 4
  - int v = 42;
    - v : .word 42
  - char w;
    - w: .space 1
  - char x = ‘a’;
    - x: .byte ‘a’
  - double y;
    - y: .space 8
  - int z[20];
    - z: .space 80 (20*4-byte ints)

MIPS MEMORY

MIPS Memory Instructions

- Now we've seen how to prepare our memory with directives before the process begins, but what about during?
- Something to keep in mind before we start, labels are memory addresses (you may have realised this already!), so every time we work with a label it is just an address in memory.
  - This allows us to do addition with labels and is how we can traverse our data section!

MIPS Memory Instructions (load)

- To read from a memory address we can use load instructions, there are 3 types we use:
  - load byte
    - lb $t0, label
    - Loads a single byte value from the address of the label into $t0
  - load half
    - lh $t0, label
    - Loads a 2-byte value from the address of the label into $t0
  - load word
    - lw $t0, label
    - Loads a 4-byte value from the address of the label into $t0

MIPS Memory Instructions (save)

- To read from a memory address we can use save instructions, there are 3 types we use:
  - save byte
    - sb $t0, label
    - Saves a single byte value from $t0 into the address of the label
  - save half
    - sh $t0, label
    - Saves a byte value from $t0 into the address of the label
  - save word
    - sw $t0, label
    - Saves a byte value from $t0 into the address of the label

MIPS Memory Instructions

- But we don't always want to save directly the address referenced by a label, what if we wanted the next 4-byte value? There is typically 3 ways we can approach this:

- Label outside of the brackets (address + value inside register in brackets)

```
li      $t1, 4                  # int offset = 4
lw      $t0, label($t1)         # load value at label + offset
```

- Small constant offset (like 4) with address in register within brackets

```
la      $t0, label              # address in $t0
lw      $t1, 4($t0)             # load value at $t0 + 4
```

- Explicit address calculation
  - You calculate the offset and add it to the address and have the new address within the brackets

```
la      $t0, label              # address in $t0
li      $t1, 4                  # int offset = 4
add     $t1, $t1, $t0           # new_address = $t0 + 4
lw      $t2, ($t1)              # load value at new address
```

MIPS Memory Instructions

- What address will be calculated and what value will be loaded into $t0, after each statement?

Address	Data	Definition
0x10010000	aa:	.word 42
0x10010004	bb:	.word 666
0x10010008	cc:	.word 1
0x1001000C		.word 3
0x10010010		.word 5
0x10010014		.word 7

Statement	Address	Data ($t0)
la $t0, aa		
lw $t0, bb		
lb $t0, bb		
lw $t0, aa+4		
li $t1, 8 lw $t0, cc($t1)		
la $t1, cc lw $t0, 2($t1)		

MIPS Memory Instructions (q4.s)

- What address will be calculated and what value will be loaded into $t0, after each statement?

Address	Data	Definition
0x10010000	aa:	.word 42
0x10010004	bb:	.word 666
0x10010008	cc:	.word 1
0x1001000C		.word 3
0x10010010		.word 5
0x10010014		.word 7

Statement	Address	Data ($t0)
la $t0, aa	0x10010000	N/A
lw $t0, bb	0x10010004	666
lb $t0, bb	0x10010004	0xffffff9a
lw $t0, aa+4	0x10010004	666
li $t1, 8 lw $t0, cc($t1)	0x10010010	5
la $t1, cc lw $t0, 2($t1)	0x1001000A	Error (not 4-byte aligned)

MIPS ARRAYS

MIPS Arrays

- Well, unfortunately there is no array instructions, we only have our save and load instructions to work with.
- Luckily, we can do everything we want with just these if we just use directives to reserve the amount of memory we need for the array
- The reality is, we just reserve contiguous sections of memory for our array, which we will index into with the labels address and an offset
- Which can get very confusing when you realise, we treat a 2D array as a big 1D array and these two arrays below would look identical in memory in MIPS
  - int array[9] = {0, 1, 2, 3, 4, 5, 6, 7, 8}

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**

  - int array[3][3] = {{0, 1, 2}, {3, 4, 5}, {6, 7, 8}}

0	1	2
3	4	5
6	7	8

MIPS Arrays Example (.byte)

- Consider we create an array to hold 10 numbers that is int array[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
  - Depending on how we initialise this array can completely change how we offset into it for our load and store instructions
  - Using .byte

```
array:   .byte 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
```

  - Now let's calculate the offset for each index of the array

Index	array[0]	array[1]	array[2]	array[3]	array[4]	array[5]	array[6]	array[7]	array[8]	array[9]
Offset										
Value	0	1	2	3	4	5	6	7	8	9

MIPS Arrays Example (.byte)

- Consider we create an array to hold 10 numbers that is int array[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
  - Depending on how we initialise this array can completely change how we offset into it for our load and store instructions
  - Using .byte

```
array:   .byte 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
```

  - Now let's calculate the offset for each index of the array

Index	array[0]	array[1]	array[2]	array[3]	array[4]	array[5]	array[6]	array[7]	array[8]	array[9]
Offset	array + 0	array + 1	array + 2	array + 3	array + 4	array + 5	array + 6	array + 7	array + 8	array + 9
Value	0	1	2	3	4	5	6	7	8	9

MIPS Arrays Example (.word)

- Consider we create an array to hold 10 numbers that is int array[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
  - Depending on how we initialise this array can completely change how we offset into it for our load and store instructions
  - Using .byte

```
array:  .word 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
```

  - Now let's calculate the offset for each index of the array

Index	array[0]	array[1]	array[2]	array[3]	array[4]	array[5]	array[6]	array[7]	array[8]	array[9]
Offset										
Value	0	1	2	3	4	5	6	7	8	9

MIPS Arrays Example (.word)

- Consider we create an array to hold 10 numbers that is int array[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
  - Depending on how we initialise this array can completely change how we offset into it for our load and store instructions
  - Using .byte

```
array:  .word 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
```

  - Now let's calculate the offset for each index of the array

Index	array[0]	array[1]	array[2]	array[3]	array[4]	array[5]	array[6]	array[7]	array[8]	array[9]
Offset	array + 0	array + 4	array + 8	array + 12	array + 16	array + 20	array + 24	array + 28	array + 32	array + 36
Value	0	1	2	3	4	5	6	7	8	9

Example: q5.s

- Translate this C program to MIPS that will read 10 characters in from stdin, then store them in an array

```c
// A simple program that will read 10 numbers into an array

#define N_SIZE 10

#include <stdio.h>

int main(void) {
    int i;
    int numbers[N_SIZE] = {0};

    i = 0;
    while (i < N_SIZE) {
        scanf("%d", &numbers[i]);
        i++;
    }
}
}
```

MIPS code is getting too large for slides, check the q5.s file on the github repo for the solution!

Example: q6.s

- Translate this C program to MIPS that will print each value from an array

```
// A simple program that will print 10 numbers from an array

#define N_SIZE 10

#include <stdio.h>

int main(void) {
    int i;
    int numbers[N_SIZE] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

    i = 0;
    while (i < N_SIZE) {
        printf("%d\n", numbers[i]);
        i++;
    }
}
```

MIPS code is getting too large for slides, check the q6.s file on the github repo for the solution!

Example: q7.s

- Translate this C program to MIPS that will read a value from an array, and if it is less than 0, then you will add 42 to that value and store it back into the array

```
// A simple program that adds 42 to each element of an array if
// the value is less than 0

#define N_SIZE 10

int main(void) {
    int i;
    int numbers[N_SIZE] = {0, 1, 2, -3, 4, -5, 6, -7, 8, 9};

    i = 0;
    while (i < N_SIZE) {
        if (numbers[i] < 0) {
            numbers[i] += 42;
        }
        i++;
    }
}
```

MIPS code is getting too large for slides, check the q7.s file on the github repo for the solution!

FIN