COMP1521

Week 3 - MIPS Data

Overview

Admin

MIPS memory directives (Q2-3)

MIPS memory instructions (Q4)

MIPS arrays (Q5-7)

Admin

This lab (W3) is due next monday (W4 monday).

Assignment 1 was released on tuesday

The first weekly test is releasing this week.

Help sessions are now running in person and online, see the link on the course website for time and location.

Admin: Assignment 1

If you start early, you can go to earlier help sessions (if you need) and tutors will be able to spend more time helping you. If you attend the help sessions in the couple days before the assignment is due, the amount of time tutors can spend on each student is limited.

The same applies to asking for help on the forum. Earlier posts will receive faster replies, and as we get closer to the deadline it will take longer for tutors to get through the posts.

The style mark is worth 20%. 12/20 of those marks are allocated to FILLING OUT THE FUNCTION COMMENTS.

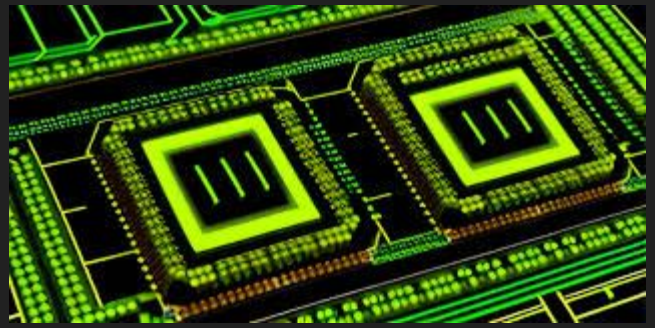
Please fill out the function comments, we hate deducting 12 marks from a student who has beautiful MIPS code because they didn’t fill out the function comments.

MIPS memory directives

Memory (.data) VS Registers

When we use instructions like “li”, “la” and “move”, we are manipulating the values inside registers.

So far we have not manipulated any of the values inside memory (specifically inside the .data section, where global variables are stored).



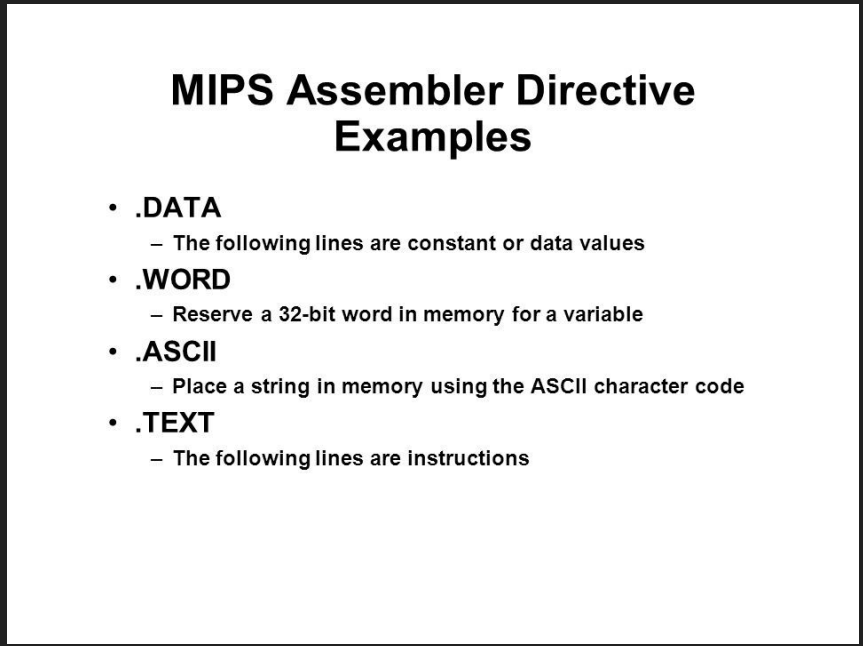
MIPS memory directives

So, how can we manipulate values inside .data?

One easy way is to define those values inside our MIPS program using memory directives.

Memory directives essentially “set up” memory right before the process starts running.

Let’s look at some of these directives.



MIPS memory directives

The general format of memory directives is as follows:

(inside the .data section, specified by first writing the “.data” directive)

label:

    .directive [arg1] [arg2] …

Where in most cases, there is only one argument for the directive.

The most familiar example would be:

my_string:

    .asciiz “Hello World!\n”

Where “Hello World!\n” is the argument for our directive “.asciiz”.

We can use the label “my_string” to refer to the address of that string in our MIPS program.

MIPS memory directives

Here are the string directives:

.asciiz “string”

> Store the ASCII string specified by “string” in memory followed by a null terminator.

.ascii “string”

> Store the ASCII string specified by “string” in memory.

MIPS memory directives

General directives:

.space [number] = reserve [number] bytes of uninitialised memory.

.byte [value] = store [value] inside one byte of memory

.half [value] = store [value] inside one half (two bytes) of memory

.word [value] = store [value] inside one word (four bytes) of memory

.align [1||2||3] = align the next directive to an address divisible by (2||4||8)

MIPS memory directives

We can also use directives to store multiple values in a row (sounds familiar?)

.word arg1, arg2, arg3, … =

allocate a series of 4 byte values in memory, storing arg1, arg2, etc

We can also use a shorthand to specify the repetition of a specific value

.byte value:number =

allocate a series of [number] 1 byte values in memory, each initialised to [value]

This syntax works with .word, .half and .byte (but not .space!).

MIPS memory directives

In a MIPS program, all directives in the .data section are mapped sequentially into memory.

So, the order that they appear in the program is the order they will appear in memory.

Words will always be stored at an address divisible by 4 (skipping some memory if necessary).

The same is true for “half”s, they are always stored at an address divisible by 2.

They appear starting the lowest address inside .data (0x10010000 or 268500992) and go up in addresses.

Sidenote: Hexadecimal??

0x10010000??? What does that mean?

If a number starts with “0x”, it means we are using Hexadecimal notation.

Customarily, memory addresses will be written this way.

Each digit of hexadecimal counts for “16”, whereas each digit of decimal (normal numbers) counts for “10”.

Hexadecimal

123 =

1 * 10^2 + 2 * 10^1 + 3 * 10^0 = 123

0x123 =

1 * 16^2 + 2 * 16^1 + 3 * 16^0 = 291

Just a different way of writing numbers.

We will spend more time on it in later weeks.

Tutorial Q2

2. 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
```

Tutorial Q3

3. Give MIPS directives to represent the following variables:

   a. `int u;`
   b. `int v = 42;`
   c. `char w;`
   d. `char x = 'a';`
   e. `double y;`
   f. `int z[20];`

   Assume that we are placing the variables in memory, at an appropriately-aligned address, and with a label which is the same as the C variable name.

MIPS memory instructions

Memory instructions

We saw how we can use directives to “set up” memory before the process begins.

How can we interact with memory while the process is running?

assembly	meaning	bit pattern
`lb` r_t, I(r_s)	r_t = mem[r_s+I]	`100000ssssstttttIIIIIIIIIIIIIIII`
`lh` r_t, I(r_s)	r_t = mem[r_s+I] |	`100001ssssstttttIIIIIIIIIIIIIIII`
	mem[r_s+I+1] << 8	
`lw` r_t, I(r_s)	r_t = mem[r_s+I] |	`100011ssssstttttIIIIIIIIIIIIIIII`
	mem[r_s+I+1] << 8 |	
	mem[r_s+I+2] << 16 |	
	mem[r_s+I+3] << 24	
`sb` r_t, I(r_s)	mem[r_s+I] = r_t & 0xff	`101000ssssstttttIIIIIIIIIIIIIIII`
`sh` r_t, I(r_s)	mem[r_s+I] = r_t & 0xff	`101001ssssstttttIIIIIIIIIIIIIIII`
	mem[r_s+I+1] = r_t >> 8 & 0xff	
`sw` r_t, I(r_s)	mem[r_s+I] = r_t & 0xff	`101011ssssstttttIIIIIIIIIIIIIIII`
	mem[r_s+I+1] = r_t >> 8 & 0xff	
	mem[r_s+I+2] = r_t >> 16 & 0xff	
	mem[r_s+I+3] = r_t >> 24 & 0xff	

Sidenote: what is a label?

A label really refers to a memory address.

A memory address is just some value (like 0x10010000 or 268500992).

When working with labels, keep in mind that when the program is running, labels are really just numbers like 0x10010000 or 268500992.

This means that, for example, we can do addition with labels (like 268500992 + 4).

Memory instructions

In order to write to a certain memory address, we can use the “save” instruction.

```
sb      $t0,    label
```

Will write a 1-byte value in $t0 to the byte at the address referred to by label.

```
sh      $t0,    label
```

Will write a 2-byte value in $t0 to the two bytes following the address referred to by label.

```
sw      $t0,    label
```

Will write a 4-byte value in $t0 to the four bytes following the address referred to by label.

Memory instructions

In order to read from a memory address, we can use the “load” instruction.

```
lb      $t0,    label
```

Will load a 1-byte value from the address referred to by label into $t0.

```
lh      $t0,    label
```

Will load a 2-byte value from the two bytes following the address referred to by label into $t0.

```
lw      $t0,    label
```

Will load a 4-byte value from the four bytes following the address referred to by label into $t0.

Memory instructions

In the previous slides we used “label” to refer to a memory address. But what if we wanted to refer to an address like “label + 4”?

An alternative syntax can be used:

```
li      $t1,    4
lw      $t0,    label($t1)
```

When we write “value($register)” in these instructions, it just means value + $register.

Memory instructions

We could also do this instead:

```
la      $t1,    label
lw      $t0,    4($t1)
```

Both ways we a referring to label + 4, we have just changed which value is in the register and which is written.

Memory instruction

We could also do this:

```
la      $t1,    label
li      $t2,    4
add     $t1,    $t1,    $t2
lw      $t0,    ($t1)
```

Again referring to label + 4, we have just added both values into the same register and only used the register.

Memory instructions

What if want to refer to an address at x + y, where x is the value inside $t0 and y is the value inside $t1? Both values are now inside registers.

This won’t work:

```
lw      $t2,    $t1($t0)    # parse fail, check your syntax
```

But we can do this:

```
add     $t1,    $t1,    $t0
lw      $t2,    ($t1)
```

Tutorial Q4

4. Consider the following memory state:

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

What address will be calculated, and what value will be loaded into register `$t0`, after each of the following statements (or pairs of statements)?

a.
```
la   $t0, aa
```

b.
```
lw   $t0, bb
```

c.
```
lb   $t0, bb
```

d.
```
lw   $t0, aa+4
```

e.
```
la   $t1, cc
lw   $t0, ($t1)
```

f.
```
la   $t1, cc
lw   $t0, 8($t1)
```

g.
```
li   $t1, 8
lw   $t0, cc($t1)
```

h.
```
la   $t1, cc
lw   $t0, 2($t1)
```

MIPS Arrays

How can we implement an array?

You may have noticed there aren’t any array instructions.

All we have is save and load.

What if we were to allocate enough memory for say 16 bytes in a row using a directive, and then we just used save and load instructions to read and write to our “array”?

Let’s try that!

An array

Define the array

```
        .data
array:
                .byte 0:16              # 16 sequential bytes of memory initialised to 0
```

Access the array

```
        .text
        lb          $t0,        array           # load one byte from array[0], the first element
```

An array

Access other indexes in the array:

```
li      $t1,    4
lb      $t0,    array($t1)      # access one byte at (array+4) or array[4]
```

Note that this would access the FIFTH element in the array!!!

But since indexes start at 0, it’s really array[4]

array + 0	array + 1	array +2	array + 3	array + 4	array + 5
1	2	3	4	5	6
array[0]	array[1]	array[2]	array[3]	array[4]	array[5]

What about an array of words?

```
        .data
array:          .word 0:16
```

Let’s load the first element again (this time using lw to load four byte value)

```
        lw      $t0,    array
```

For the other elements, let’s think about how this actually looks in memory:

(Since each element is now 4 bytes large)

array + 0	array + 4	array + 8	array + 12	array + 16	array + 20
1	2	3	4	5	6
array[0]	array[1]	array[2]	array[3]	array[4]	array[5]

An array of words

So, to load the second element we actually need to load from the address “array + 4”.

```
li      $t1,    4
lw      $t0,    array($t1)
```

And for the third element we need to load from the address “array + 8”.

```
li      $t1,    8
lw      $t0,    array($t1)
```

A general solution to arrays

So in general, we can define an array using a data directive appropriate to the size of the element (typically .byte for chars and .word for ints)

For example, “.word 0:1024” or “.byte 0:64” or even we could use “.space 256”

We can access an element at array[i] by accessing the address:

array + (i * size)

Where size is the size in bytes of the element.

For word arrays we should use i * 4, since each element is 4 bytes.

For byte arrays we should use i * 1 or just i, since each element is 1 byte.

“A ray”

Tutorial Q5

5. Translate this C program to MIPS assembler

```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++;
    }
}
```

Tutorial Q6

6. Translate this C program to MIPS assembler

```c
// 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++;
    }
}
```

Tutorial Q7

7. Translate this C program to MIPS assembler

```c
// A simple program that adds 42 to each element of an array

#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++;
    }
}
```