Queen Mary
University of London

ECS404U: COMPUTER SYSTEMS & NETWORKS

2020/21 – Semester 1

Prof. Edmund Robinson, Dr. Arman Khouzani

Lab Week 09: *Getting Further with (MIPS) Assembly*
memory layout, memory access, branching and loops, I/O

November 16/18, 2020

Deadline for submitting your proof of work: ***Next week's Thursday, at 10:00 AM UK time***

Student Name:

Yujeong JUNG

Student ID:

200699145

Brief Feedback to student (commendations, areas for improvement):

Summary and Learning Objectives

This is the second of our three labs on MIPS architecture and MIPS assembly programming. The learning objectives of this week's lab are the following:

- Analysing an example memory layout of a programme, special emphasis on appreciating the idea that in von-Neumann architecture, the instructions and data are both held in memory as bit-sequences;
- Learning how to declare (static) data in assembly, and knowing how they are held in the memory;
- Learning how to load data from the main memory to registers, processing them, and then storing them back to the main memory, one "word" at a time;
- Understanding that an array is essentially consecutive memory units of the same type in the memory; and that a string is essentially an array of characters;
- Having an idea of what a "system call" is; and specifically how to use some of them for I/O (taking a value from console, displaying a value to the console);
- Practising how basic branching instructions can be used to achieve programme flow control by manipulating what instruction get executed next depending on a conditional;
- Learning how to use simple branching instructions to implement a "for loop" in MIPS assembly.

1 Memory Layout

One of the major implications of the von-Neumann architecture is that both data and instructions reside in the same memory (there is no separate storage and communication paths for data versus instructions).[1] Indeed, whenever you run a program on your computer, both the instructions (machine codes) and its associated data (numbers, strings, etc.) are loaded into the main memory from the long term storage. Note that both instructions and data are represented in bits (instructions as 32-bit or 64-bit machine codes). So, in order to prevent mayhem(!), we have to keep the data and code in an orderly way in separate address blocks in the memory. Hence, we need a memory layout (also known as a "memory map").
The memory layout of the (32-bit) MIPS processor is depicted in Figure 1. Keep the following in mind:

- Here, whenever we say memory, we mean the main memory, a.k.a., the primary memory, volatile memory, or RAM;
- Each "Byte" of memory has a unique address. This is in fact not specific to MIPS, or a 32-bit vs 64-bit architecture, and **almost all architectures make memory Byte-addressable**; Note: do not confuse the address of a memory with its content: think of the address as the unique "index" referring to a specific Byte in the memory. Here, the address is 32-bits long, but the memory content identified by that address is just 8 bits long, because it is a single Byte!
- In a 32-bit architecture, everything is 32-bits long, including each address: starting from the address of the first Byte of the memory at **`0x00000000`** (which is of course 32 bits of zeros in binary), to the last Byte of the memory addressed at **`0xFFFFFFFF`** (which is of course 32 bits of ones in binary);

[1] This is as opposed to the Harvard architecture which separates the memory and pathways for data from instructions. Harvard architecture, although much less common, is still used.

- For MIPS, the section of the memory that is given to the program (called the "user" space) starts at address **0x0400000** and ends at **0x7FFFFFF**; The section bellow the user memory (i.e., Bytes **0x00000000** through **0x03FFFFF**) and the section after (i.e., Bytes **0x80000000** through **0xFFFFFFFF**) are reserved for the the Operating System's programmes, and is hence called the "Kernel" space;[2] this is where the kernel level instructions and data are loaded to (e.g. system calls, exception and interrupt handlers, etc).
- The user space is divided into two main parts: **text** and **data**:
  - **text**: where the instructions (machine codes) are stored;
  - **data**: where the program data (like integers, strings, arrays, structures) is stored.

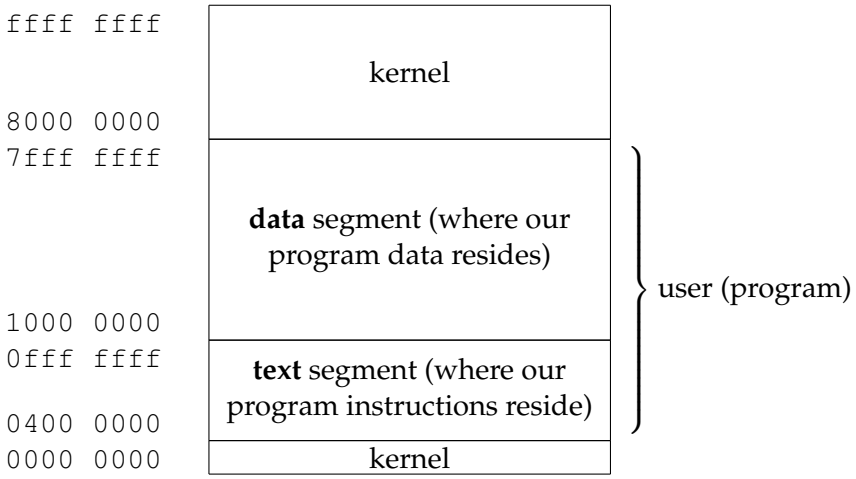

Figure 1: Memory layout in a (32-bit) MIPS computer.

After studying the above information, answer the following questions:

Q1. Knowing that each Byte has a unique address, and each address in a 32-bit architecture is 32-bits long, what is the biggest size of memory that a 32-bit CPU can have?

0x00000000 to 0xFFFFFFFF
0 to 4,294,967,295

1GB=1024^3

(the biggest size of memory that a 32-bit CPU)= 4,294,967,295 bytes / 1024^3
=3.999999 GB

Q2. Given the allocated address regions, what is the size of the user's text segment? How many (actual) MIPS instructions is that?

[2]Kernel is the name describing the programme that constitutes the main part of an operating system. The kernel is loaded into the memory at start-up, i.e., when the computer is turned on, and always resides in the memory. It is responsible for mediating access of the programmes to different parts of the computer like the cpu, memory, I/O, etc.

0400 0000 to 1000 0000
67,108,864 to 268,435,456
(difference)= 201,326,592 bytes

8bits = 1 byte
so, each instruction is 32 bits (= 4bytes)
201,326,592 / 4 =50,331,648

Q3. What is the size of the user's data segment? If we want to fill it up with integers, how many numbers can we fit?

1000 0000 to 7fff ffff
268,435,456 to 2,147,483,647
(difference)=1,879,048,191 bytes

2 text vs. data

In the previous lab, we saw simple examples of mips assembly codes, which goes in the **`text`** segment of the memory. What about program data?

A typical assembly code is divided into a **`text`** and a **`data`** section, denoted by "directives" **`.text`** and **`.code`**. Assembler directives are not themselves cpu instructions, they just direct the assembler to do something. The instructions (the assembly code) go after the **`.text`** directive and the definitions of data go after **`.data`**. The syntax for declaring a data variable is as follows:

label of the variable : .data type initial value

For instance, the following code just defines three data values. To be exact, the program itself does nothing: just exits. But the content of the memory is affected.

```
1     # All program code is placed after the
2     # .text assembler directive
3
4 .text
5   addi $v0, $zero, 10     # Sets $v0 to "10" to select exit syscall
6   syscall                 # Exit
7
8
9
10    # All memory structures are placed after the
11    # .data assembler directive
12
13 .data
14     MY_MSG: .asciiz "QMUL!"
15     MY_INT: .word 15
16     MY_INT_ARRAY:
17         .word    0
18         .word    10
19         .word    5
20     MY_INT_ARRAY_LEN:    .word    3
```

Codes/week9_data_example.asm

In particular:

- **`MY_MSG`** is a null terminated string of ascii characters. A null terminated string is just an array of characters whose end is demarcated by a null character (ascii code of **`0x00`**). The string is initialised to '**`QMUL!`**' .
- **`MY_INT`** is just a 32-bit word (remember that in a 32-bit architecture, a "word" just refers to 32-bits or 4 Bytes), initialised to 15 (which is going to be **`0x0000 000f`**);
- **`MY_INT_ARRAY`** is an array of 3 "words", initialised to numbers 0, 10 and 5 respectively. Note that an array is just a sequence of data of the same type stored sequentially in the memory;
- **`MY_INT_ARRAY_LEN`** Since there is nothing inherently in the MIPS assembly that tells us the length of an array, we should somehow keep track of it; Either by explicitly storing a value for its length, or by demarcating the end by using a special element (like we did for a null-terminated string – keep in mind that a string is just an array of characters).

Note that the indentation and comments are of course optional.

Q4. Based on what we know about ascii representation of characters, binary representation of integers, and the information you just learned, predict what is going to be stored in the user data segment of the memory when this program runs.

In particular, fill in the following table of addresses and values (for at least 10 or so rows – you don't have to fill all). For compactness, show values in HEX (as opposed to binary!). Recall that each word is 32 bits (4 Bytes), and each (8-bit) ascii character is one Byte, and each Byte has its own address. Assume these three data entries are stored in the memory in the same order that we have declared them, starting from address **`0x1001 0000`**. An example is done for you.

Address of the *Byte* (Hex)	**Content of the Byte (Hex)**	Explanation
eg: **`0x1001 0000`**	**`0x51`**	8-bit ascii for `Q`
0x1001 0001	0x4d	8-bit ascii for M
0x1001 0002	0x55	8-bit ascii for U
0x1001 0003	0x4c	8-bit ascii for L
0x1001 0004	0x21	8-bit ascii for !
0x1001 0005	0x00	String termination
0x1001 0006	0x00	- (byte aligment)
0x1001 0007	0x0F	- (byte aligment)
0x1001 0008	0x00	No data
0x1001 0009	0x00	No data
0x1001 000a	0x00	No data
0x1001 000b	0x00	No data
0x1001 000c	0x00	No data
0x1001 000d	0x00	No data
0x1001 000e	0x00	No data
0x1001 000f	0x00	No data
⋮	⋮	⋮

Q5. Now from QM+, download the assembly file named **`week9_data_example.asm`**; save it somewhere you can locate it, and from your **`QtSpim`** load it (recall: always use "`File`" -> "`Reinitialize and Load File`"). Now from the "tabs" switch to **Data** (as opposed to **Text**). If you don't see a "Data" tab, you might need to switch it on under "`Window`".

Check the content of the data against what you came up with in the previous question. Identify any differences. Note that of course you should get the same values. But in case you didn't, in the following box write down the source of your mistake.

Same value.
There is no difference.

Note: If you are struggling with this question and are stuck in it during the lab, move on to the rest of the tasks and come back to it at the end if you have time.

3 Basics of memory access

Recall that MIPS is an example of load/store architecture: any operation can only be done on the contents of registers, not directly on the content of the main memory. This means that in order to process data we have to first read them from memory into registers.

We can read the data we have stored in memory using the following commands:

- Load address:
  `la rt, <label of the data>`
  puts the address of the variable denoted by its label (which we have declared in the **`.data`** section of our code) in register **`rt`**. For instance, **`la $t0, MY_MSG`** loads the address of the starting Byte of our string which we labelled **`MY_MSG`** into register **`t0`**.
- Load word:
  `lw rt, offset(rs)`
  puts 4 bytes (a word) from the memory into register **`rt`**, whose starting address (address of the its first Byte) is **`$rs + offset`**. For instance, **`lw $t1, 0($t0)`** loads a word (4 bytes) into register **`t1`** from memory where the starting address is just the content of register **`t0`**.

To store values back in memory, we can use the following instruction:

- Store word:
  `sw rt, offset(rs)`
  the content of register **`rt`** is written in memory whose address (of its first Byte is) **`$rs + offset`**. Note that as the only exception to the general rule, here, the destination is written second!

Q6. Write a MIPS Assembly code that increment the value of **`MY_INT`** that we declared in our previous code in the memory.

```
la $v0 , MY INT # v0=address(MY  INT)
lw $tzero, 10($t10) # tzero=value(MYINT)
addi  $v0, $tzero, 10 # v0++
sw $v0, 10($t10) # value(MYINT)++
```

4 System calls and I/O

System calls are just special purpose kernel programs (which are in turn themselves just some MIPS instructions) that the processor executes for us. We have been using one of the system calls already: system call "10": for exiting our program. Other examples of useful system calls include getting inputs from console and printing some output to the console. We invoke system calls by putting their number into register **`v0`**, and if they need any parameters, putting them into register **`a0`** and then calling the instruction **`syscall`**. The processor executes the system call function identified by the number we have put in register **`$v0`**.

Service	Code in **`$v0`**	Arguments	Result
print integer	1	**`$a0`** = integer to print	-
print string	4	**`$a0`** = address of null-terminated string	-
read integer	5	-	$v0 holds integer read

Table 1: Some of the Syscall services available in **`QtSpim`**, along with the detail of how to invoke them.

So, for instance, if we wanted to print the string "QMUL!" in the console, the assembly code is:

```
#   code comes after the .text directive
.text

    addi  $v0, $zero, 4     # to be requesting syscall 4, for printing strings
    la  $a0, MSG       # load the address of the starting byte of our string into $a0
    syscall                 # print the string by issuing the syscall

    addi  $v0, $zero, 10    # Load the syscall to exit.
    syscall                 # Invoke the syscall to exit.

#   the memory structure comes after the .data directive
.data
    MSG:    .asciiz "Hello, World!"
```

Codes/week9_system_calls_example.asm

Q7. For a simple example, write down a code that prompts the user with a message:
`please enter an integer number:`
Then prints a message like
`The number you entered was:` along with the number they entered. Recall: you can use any other text editor to write your code. Then run it on **`QtSpim`**. If all is good, then finally copy/-paste your programme in the provided box.

Call on a demonstrator if you are facing difficulty **after some effort!**

Note: *To see the output in* ***QtSpim****, make sure* `console` *under* `Window` *is ticked!*

```
 # code comes after the .text directive
 # print "The number you entered was:"
.text

 addi   $v0, $zero, 4    # to be requesting syscall 4, for printing strings
 la $a0, MSG1    # load the address of the starting byte of our string into $a0
 syscall

 addi $v0, $zero 1
 add $a0, $zero, $t0
 syscall

 addi $v0, $zero, 5    # Load word contetn of register 5 into register $v0
 la $a0, MSG2        # a0 = print integer
 syscall

 addi $v0, $zero, 10   # Load the syscall to exit.
 syscall             # Invoke the syscall to exit.

 # the memory structure comes after the .data directive
.data
 MSG1:     .asciiz "The number you entered was:"
 MSG 2:    .asciiz " please enter an integer number:"
```

5 Branching and loops

branching instruction can be used to achieve program flow control by manipulating what instruction get executed next depending on a conditional, e.g. the result of a comparison of the values of two registers. The most basic branching instructions in MIPS Assembly are as follows:

- (unconditional) jump:
  `j    <label of a line of code>`
  unconditionally jumps to the instruction with a label defined in the code (load the content of the **`PC`** register with the address of the instruction designated by our label, instead of the next instruction, as is the default progression of the execution).
- Branch on equal:
  `beq    rs, rt, <label of a line of code>`
  jump to the instruction designated with the given label if the contents of the registers **`rs`** and **`rt`** are equal, otherwise, proceed to the next instruction as usual.

Note that we can put labels on anywhere in our code by simply putting a colon after it, e.g.:

```
1 do_something_fancy:
2 addi $t0, $t0, 1
```

Q8. Write a MIPS assembly program (in your favourite text editor, then test it in **QtSpim**, and if it works, copy/paste it here) to do the following: looks at the content of register **s0**, and prints an appropriate message:

- if it is zero, print `it was zero!`
- if it is 1, print `it was one!`
- if it is not zero nor one, print `it was neither!`

6 Main task: putting things together

Q9. Given a natural number N as a variable (input), we would like to sum of natural numbers from 1 to N (inclusive). That is, we are interested in computing:

$$\text{sum} = 1 + 2 + \ldots + N = \sum_{i=1}^{N} i$$

We know how to implement it in a high level language like C, or Java. For instance, the following Java snippet computes what we want:

```
int n = 10;
int sum = 0;
for(int i = 1 ; i <= n ; i++){
    sum = sum + i;
}
System.out.format("The total sum is: %d", sum);
```

However, how is this “actually” done on the “machine”? Given what we have learned so far, write the MIPS Assembly code that implements this task.