Lab2: Introduction to Memory, Addressing, and Branching

Learning Outcomes:

After completing this lab, you will be able to:
- Understand different segments in memory.
- Understand the difference between direct and indirect addressing.
- Use branching instructions.

➤ Download the `Lab02_Example.s` file from the Moodle course page and open it in the Ripes tool.

➤ There is a Moodle quiz which will be opened at 8.15am and closed at 10.15am. There are four pages in the quiz. The 1st page contains questions on theory parts and the subsequent pages contain questions from Exercise 1, Exercise 2 and Exercise 3 respectively. After completing each of the exercises, answer the questions in Moodle. Before accessing the exercise questions on Moodle make sure to do the corresponding exercise first.

Introduction to Memory

Go to the memory tab of Ripes by clicking the icon from the left sidebar. Use the options given at the bottom of the UI to change the display type and go to each specific memory area.
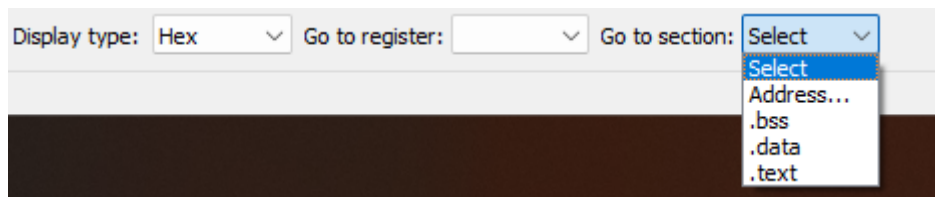

Figure 1: Memory Options

There are three main segments of memory supported by RIPES.

1. **Data** - The data segment stores the initialised data. It generally supports both reading and writing. Observe that in the assembly code the data segment starts with the `'.data'` directive.

```
1  .data
2      x:          .word       1541                # Declare a word with value 1541
3      array:      .word       10, 20, 30, 40, 50  # Declare an array of five elements
4
5      # A string can be declared with string directive
6      newline:    .string     "\n"
7      str:        .string     "abcdefghijklmn"
```

Figure 2: Definition of the data segment in the Assembly file

Address	Word	Byte 0	Byte 1	Byte 2	Byte 3
0x10000028	0	0	X	X	X
0x10000024	1852664939	107	108	109	110
0x10000020	1785292903	103	104	105	106
0x1000001c	1717920867	99	100	101	102
0x10000018	1650524170	10	0	97	98
0x10000014	50	50	0	0	0
0x10000010	40	40	0	0	0
0x1000000c	30	30	0	0	0
0x10000008	20	20	0	0	0
0x10000004	10	10	0	0	0
0x10000000	1541	5	6	0	0

Figure 3: Values in the data segment of memory (display type is set to signed)

2. **BSS** - The BSS (Block Started by Symbol) segment contains uninitialized data. This segment also supports both reading and writing. Observe that in the assembly code provided, the BSS segment starts with the '.bss' directive. See the values stored in this area of the memory using the memory tab.

3. **Text** - The text segment contains instruction memory. Hence, it is generally read-only. Observe that in the assembly code provided, the text segment starts with the '.text' directive. See the values stored in this area of the memory using the memory tab.

Introduction to Addressing Modes

Following are the major classes of addressing supported by RISC-V architecture:

1. **Direct addressing**
   The immediate value of the instruction specifies the memory address that will be referenced. Only one memory access is required to access the memory location.

```
15      # Direct Addressing
16      lw a0, x    # Load the value in memory location x directly to the register a0
```

Figure 4: Sample code for direct addressing

Note that the value stored in memory location `x` is directly loaded into the register `a0`.

★ We will not be able to reference the complete memory space with direct addressing in RISC-V architecture. Can you identify the reason for that?

2. **Register Indirect addressing**
The address of the memory location can be stored in a register and the instruction references that register. When the instruction is executed, firstly, it will take the value in the register and then access the memory location with that value as the address. This is generally known as register indirect addressing.

```
19      # Register Indirect Addressing
20      la a0, x        # Load the address of memory location x to the register a0
21      lw a1, 0(a0)    # Load the value of address stored in a0 to register a1
22      addi a1, a1, 400
23      la a2, z        # Load the address of memory location z to the register a2
24      sw a1, 0(a2)    # Store the new value to the memory address stored in a2
```
Figure 5: Sample code for register indirect addressing

In the above example, both `lw` and `sw` commands, which are defined at lines 21 and 24 respectively, use the address stored in another register to access the corresponding memory location.

3. **Displacement addressing**
In displacement addressing, the memory location will be referenced by a displacement to an address stored in a register.

```
27      # Displacement Addressing WRT to address stored in a register
28      # Change the value at index 2 of the array to 100
29      li a0, 100
30      la a1, array    # Load the address of the first element of the array to register a1
31      sw a0, 8(a1)    # Size of word * index = 4 * 2 = 8
```
Figure 6: Sample code for displacement addressing

In the above example, the memory location is accessed by a displacement of 8 to the address stored in register `a1`. The name of an array points to the first element of the array. Therefore, when we load the address of the array, we load the address of the first element of the array to register `a1`. Since the size of a word is 4 bytes and RISC-V uses byte-addressing, we have to increment the address by a multiple of 4 to reach the next memory location. Therefore, the displacement is taken as 8 to access the element at index 2.

String Literals

A string is a null-terminated character (byte) array. A string is declared with the '`.string`' directive.

```
5    # A string can be declared with string directive
6    newline:    .string      "\n"
7    str:        .string      "abcdefghijklmn"
```

Go to the data section of the memory tab and set the display type to ASCII.

Address	Word	Byte 0	Byte 1	Byte 2	Byte 3
0x10000028			X	X	X
0x10000024	nmlk	k	l	m	n
0x10000020	jihg	g	h	i	j
0x1000001c	fedc	c	d	e	f
0x10000018	ba...	...		a	b

You can see that the letters in the string `str` are stored at each byte and followed by an empty byte (the byte at address `0x10000028`). By setting the display type back to signed or unsigned, you can see the value in this byte is zero. It is the null terminator of the string. The null terminator determines the ending point of the character array (string).

The Byte 0 of the word that starts at address `0x10000018` is the newline character. Using the signed/unsigned display type, you can observe that it has an ASCII value of 10. Byte 1 is the null terminator for this string.

```
34   # String Literals
35   la a0, str      # Load the address of the string to the register a0
```

Execute the above instruction and see what is the value in the register `a0`.

x10	a0	0x1000001a

Notice from the memory section that the string `str` starts from the Byte 2 of the word which starts at `0x10000018`. The address of this byte is `0x10000018 + 0x2 = 0x1000001a`. Therefore, the address we load by referencing the `str` memory location is the address of the character array's first byte.

★ Try adding a word immediately after the declaration of the variable `str` in the data section. Change the length of the string by adding or removing characters. What is your observation?

Introduction to Branching

Branching instructions change the value of the program counter if the given condition is satisfied. These instructions will change the instruction which gets executed next. Branching can be used to implement conditional statements as well as loops.

```
38    # Branching
39    li a1, 0        # Loop control variable
40    li a2, 20       # The upper bound to the loop control variable
41    la a3, array    # Reference to the first element of the array
42    add a4, a3, a1  # Calculate the address of the element at the current index
43    lw a0, 0(a4)    # Load the value at current index to register a0
```

In a typical loop, a loop control variable controls the loop and determines when the loop should break. The loop control variable of the above example is stored in the register `a1` and initialised to `0` for the zeroth index. The next line defines the upper bound to the loop control variable. Since the array's size is 5 and the size of a word is 4, the upper bound will be 20. This value will be used to determine when to terminate the loop.

```
45    # Print the value at register a0 to console
46    li a7, 1
47    ecall
48
49    # Print newline
50    la a0, newline
51    li a7, 4
52    ecall
```

The next section prints the value at the current index and a new line. Built-in system calls in Ripes are used for console printing. More details about printing to the console and other system calls can be found at https://github.com/mortbopet/Ripes/blob/master/docs/ecalls.md.

```
54    # Branch to begining of the loop
55    addi a1, a1, 4   # Increment the loop variable by the size of word
56    blt a1, a2, -36  # Go 9 instructions backwards to continue printing if not completed
```

This section increments the loop control variable by word size. Then, it will be compared with the maximum value. If the variable is less than that, the program will decrement the program counter by 9 instructions to continue the loop. Otherwise, it will move to the next instruction, exiting the loop.

```
58    # End the program
59    li a0, 0
60    li a7, 93
61    ecall
```

The final section will exit the program by calling the exit system call with status code 0.

Exercise 1

Given an N-element vector (i.e., array) `A`, generate another vector `B`, such that `B` only contains those elements of `A` that are even numbers greater than 0.

For example: suppose `N = 12` and `A = [0, 1, 2, 7, -8, 4, 5, 12, 11, -2, 6, 3]`. Then B would be `B = [2, 4, 12, 6]`.

Write a RISC-V assembly program using branching instructions.

Exercise 2

Given two N-element vectors (i.e., arrays) `A` and `B`, create another vector `C`, defined as:

```
C[i] = |A[i] + B[N-i-1]|, i = 0,..,N-1.
```

Write a program in RISC-V assembly that computes the new vector. Use 12-element arrays in your program.

Exercise 3

Write a RISC-V assembly program that computes the first 12 numbers in the Fibonacci sequence, and stores the result in a finite vector (i.e. array), *V*, of length 12. This infinite sequence of Fibonacci numbers is defined as:

```
V[0]=0, V[1]=1, V[i]=V[i-1]+V[i-2] (where i=0,1,2...)
```

In other words, the Fibonacci number corresponding to element `i` is the sum of the two previous Fibonacci numbers in the series. Table 1 shows the Fibonacci numbers for `i = 0` to `8`.

Table 1. Fibonacci series

i	0	1	2	3	4	5	6	7	8
V	0	1	1	2	3	5	8	13	21

The dimension of the vector, *N*, must be defined in the program as a constant.

Additional Exercises

1. **Detect an overflow using the available instructions.**

    Write an assembly program to set `a3 = 1` if there is an overflow after executing the `add a0, a1, a2` instruction on **unsigned numbers.** If there is no overflow, a3 should remain 0.
            add a0, a1,a2

    Use the following table to summarise all the conditions that an overflow could occur:

a1	a2	a0 (a1 + a2)	

    Detect an overflow using the available instructions (signed).

    What are the conditions for overflow and underflow for adding instruction with **signed values** in two's complement representation?

    ```
    add a0, a1,a2
    ```

    Use the following table to summarise all the conditions in that an overflow could occur:

a1	a2	a0 (a1 + a2)	

    Write an assembly program to set `a3 = 1` if there is an overflow after executing the `add a0, a1, a2` instruction with **signed numbers**. If there is no overflow/underflow, a3 should remain 0.

2. Write an assembly program to calculate the absolute value of a signed integer in two's complement format.

3. Write a program in RISC-V assembly that computes the factorial of a given non-negative number, n, using iterative multiplications. You should test your program for multiple values of n.

4. Using branching instructions, write a RISC-V assembly program that finds the greatest common divisor of two numbers, a and b, according to the Euclidean algorithm. The values a and b should be statically defined variables in the program. Name the program GCD.S. Here is some additional information about the Euclidean algorithm:https://www.khanacademy.org/computing/computerscience/cryptography/modarithmetic/a/the-euclidean-algorithm. You can also simply google "Euclidean algorithm" or use the following hint.

Hint :

```
int gcd(int a, int b)
{
    int c;
    while (b != 0)
    {
        c = a % b;
        a = b;
        b = c;
    }
    return a;
}
```