



iFYP

Lab Manual # 02 – Data Transfer, Decision Making, Logical Ops

(Arrays in RISC-V Assembly, Bit Manipulation and Implementation of factorial function)

Release: 1.0

Date: 04-March-2024

NUST Chip Design Centre (NCDC), Islamabad, Pakistan



Copyrights ©, NUST Chip Design Centre (NCDC). All Rights Reserved. This document is prepared by NCDC and is for intended recipients only. It is not allowed to copy, modify, distribute or share, in part or full, without the consent of NCDC officials.

Revision History

Revision Number	Revision Date	Revision By	Nature of Revision	Approved By
1.0	04/03/2024	Hira Sohail, Muhammad Bilal	Complete manual	Dr. Waqar



Contents

Objective	3
Tools	3
Assembly Language	3
Introduction to Venus	3
Exercise 1: Venus Basics	3
Translating from C to RISC-V	5
Exercise 2: Using the Venus Debugger	8
Lab Task 1- Using Memcheck.....	10
Lab Task 2- Arrays In Assembly	11
Lab Task 3- Arrays partitioning	12
Lab Task 4- Factorial.....	12
Lab Task 5 – Bit Manipulation.....	13
Reference	15



Objective

The objective of this lab is to:

- Install and get familiar to Venus simulator.
- Write an assembly code translating C to RISC-V
- Perform simple tasks using assembly language such as factorial function, bitwise operation.

Tools

- Venus

Assembly Language

Assembly language is a low-level language that helps to communicate directly with computer hardware. It uses mnemonics to represent the operations that a processor must do.

It is an intermediate language between high-level languages like C++ and the machine language.

In an assembly language program, each instruction represents a single operation that the computer's CPU can perform. These can include simple arithmetic and logical operations, such as adding and subtracting values, as well as more complex operations that involve manipulating data stored in the computer's memory. Assembly language programs are typically written in a text editor and then assembled using a specialized software tool called an assembler.

Introduction to Venus

For the next few labs, we will work with several RISC-V assembly files, each of which has a ".s" file extension. To run these, we will be using [Venus](#), an academic RISC-V assembler and simulator. For these labs we will use Venus website for simplicity but we recommend you to go through these instruction for Venus simulator, [Venus reference](#).

Exercise 1: Venus Basics

You can "mount" a folder from your local device onto Venus's web frontend, so that edits you make within the browser Venus editor are reflected in your local file system, and vice versa. If you don't do this step, files created and edited in Venus will be lost each time you close the tab, unless you copy/paste them to a local file.

This exercise will walk you through the process of connecting your file system to Venus, which should save you a lot of trouble copy/pasting files between your local drive and the Venus editor.



1. Firstly, you are required to clone the following repository <https://github.com/61c-teach/sp24-lab-starter.git>
2. In the cloned repository, switch the directory to `tools` and run the script `download.sh` with the command `./download.sh`. This will download all the necessary tools to perform the labs.
3. In your labs folder, run `java -jar tools/venus.jar -dm .`. This will expose your lab directory to Venus on a network port.
 - You should see a big "Javalin" logo.
 - If you see a message along the lines of "port unable to be bound", then you can specify another port number explicitly by appending `--port PORT_NUM` to the command (for example, `java -jar tools/venus.jar . -dm --port 6162` will expose the file system on port 6162).
4. **Open** <https://venus.cs61c.org> in your web browser (Chrome or Firefox are recommended).
5. In the Venus web terminal, run `mount local labs` (if you chose a different port, replace "local" with the full URL, such as `http://localhost:6162`). This connects Venus to your file system.
 - In your browser, you may see a prompt saying `Key has been shown in the Venus mount server! Please copy and paste it into here..`. You should be able to see a key in the most recent line of your local terminal output; just copy and paste it into the dialog.
6. **Go to** the "Files" tab. You should now be able to see your `labs` directory under the `labs` folder.
7. **Navigate** to `lab03`, and make sure it works by hitting the `Edit` button next to `ex1_hello.s`. This should open in the `Editor` tab.
 - You should see the contents of `ex1_hello.s` in the editor. This editor behaves like most other text editors, albeit without many of the fancier features.
8. To assemble the program open in the editor, **click** the "Simulator" tab, and **click** "Assemble & Simulate from Editor".
 - In the future, if you already have a program open in the simulator, click "Re-assemble from Editor" instead. Note that this will overwrite everything you have in the simulator tab, such as an existing debugging session.
9. To run the program, **click** "Run".
 - You can see other buttons like "Step", "Prev", and "Reset". You will use these buttons later on in the lab and during your assignments.
10. Go back to the editor tab, and **edit** `ex1_hello.s` so that the output prints `2023`.
 - Hint: The value in `a1` is printed when `ecall` executes. What `ecall` does is out of scope for this class though.



11. **Save** the changes you just made by hitting Cmd + s on macOS and Ctrl + s on Windows/Linux. This will update your local copy of the file.
12. **Open** the file on your local machine using the text editor of your choice to check and make sure it matches what you have in the web editor.
 - **Note:** If you make any changes to a file in your local machine using a text editor, if you had the same file open in the Venus editor, you'll need to reopen it from the "Files" menu to get the new changes.
13. **Run** the program again, you should now see 2023 if you modified the source file correctly.

Translating from C to RISC-V

In this example, we are going to walk through translating a C program to a RISC-V program. The following program will print out the nth Fibonacci number. Even though this section is a bit long, please read through it!

```
#include <stdio.h>

int n = 12;

// Function to find the nth Fibonacci number
int main(void) {
    int curr_fib = 0, next_fib = 1;
    int new_fib;
    for (int i = n; i > 0; i--) {
        new_fib = curr_fib + next_fib;
        curr_fib = next_fib;
        next_fib = new_fib;
    }
    printf("%d\n", curr_fib);
    return 0;
}
```

Let's break down how we'll translate this step-by-step. Open `fib.s` in the Venus editor. First, we need to define the global variable `n`. In RISC-V global variables are declared under the `.data` directive. This represents the data segment. It will look like this:

```
.data
n: .word 12
```

- `n` is the name of the variable
- `.word` means that the size of the data is one word
- `12` is the value that is assigned to `n`

It is noteworthy that the `.data` section must initialize all the global variable declared under the scope of the section must be initialized with a value.



Quiz Question: How can we declare variables without initialization in RISC-V Assembly?

Ans.

Let's move on to initializing `curr_fib` and `next_fib`

```
.text
main:
    add t0, x0, x0 # curr_fib = 0
    addi t1, x0, 1 # next_fib = 1
```

Here we have added the `.text` directive. Everything under this directive is our executable code.

Remember that `x0` cannot be modified and always holds the value 0.

We don't need to do anything to declare `new_fib` (we don't declare variables in RISC-V).

Next, let's get to the loop. We'll start with setting up the loop variables. The following code will set `i` to `n`

```
la t3, n # load the address of the label n
lw t3, 0(t3) # get the value that is stored at the address denoted by the label n
```

You can think of the code above as doing something along the lines of

```
t3 = &n;
t3 = *t3;
```

We have a new instruction here `la`. This instruction loads the address of a label. The first line essentially sets `t3` to be a pointer to `n`. Next, we use `lw` to dereference `t3` which will set `t3` to the value stored at `n`.

Now, you're probably thinking, "Why can't we directly set `t3` to `n`?" In the `.text` section, there is no way that we can directly access `n`. (Think about it. We can't say `add t3, n, x0`. The arguments to `add` must be registers and `n` is not a register.) The only way that we can access it is by obtaining the address of `n`. Once we obtain the address of `n`, we need to dereference it which can be done with `lw`. `lw` will reach into memory at the address that you specify and load in the value stored at that address. In this case, we specified the address of `n` and added an offset of 0.

Let's get down to the loop now. First, we'll create the outer structure below:

```
fib:
    beq t3, x0, finish # exit loop once we have completed n iterations
    ...
    ...
    addi t3, t3, -1 # decrement counter
    j fib # loop
finish:
```

The first line (`fib:`) is a label that we will use to jump back to the beginning of the loop.



The next line (`beq t3, x0, finish`) specifies our terminating condition. Here, we will jump to another label, `finish`, once `t3` (which is representing `i`) reaches `0`.

The next line (`addi t3, t3, -1`) decrements `i` at the end of the loop body. It's important to do this at the end because `i` is used in the loop body. If we updated it right after `beq`, then it would not have the correct value in the loop body.

The next instruction jumps back to the start of the loop.

Now, let's add in the loop body.

```
fib:
    beq t3, x0, finish # exit loop once we have completed n iterations
    add t2, t1, t0 # new_fib = curr_fib + next_fib;
    mv t0, t1 # curr_fib = next_fib;
    mv t1, t2 # next_fib = new_fib;
    addi t3, t3, -1 # decrement counter
    j fib # loop
finish:
```

Nothing special here. The corresponding C lines are written in the comments.

Let's print out the nth Fibonacci number!

```
finish:
    addi a0, x0, 1 # argument to ecall to execute print integer
    addi a1, t0, 0 # argument to ecall, the value to be printed
    ecall # print integer ecall
```

Printing is a system call. You'll learn more about these later in the semester, but a system call is essentially a way for your program to interact with the Operating System. To make a system call in RISC-V, we use a special instruction called `ecall`. To print out an integer, we need to pass two arguments to `ecall`. The first argument specifies what we want `ecall` to do (in this case, print an integer). To specify that we want to print an integer, we pass a `1`. The second argument is the integer that we want to print out.

In C, we are used to functions looking like `ecall(1, t0)`. In RISC-V, we cannot pass arguments in this way. To pass an argument, we need to place it in an argument register (`a0-a7`). When the function executes, it will look in these registers for the arguments. (If you haven't seen this in lecture yet, you will soon). The first argument should be placed in `a0`, the second in `a1`, etc.

To set up the arguments, we placed a `1` in `a0` and we placed the integer that we wanted to print in `a1`.

Next, let's terminate our program! This also requires `ecall`

```
addi a0, x0, 10 # argument to ecall to terminate
ecall # terminate ecall
```

In this case, `ecall` only needs one argument. Setting `a0` to `10` specifies that we want to terminate the program.

And there you have it! Here's our full program!

```
.data
n: .word 12
```




```
.text
main:
    add t0, x0, x0 # curr_fib = 0
    addi t1, x0, 1 # next_fib = 1
    la t3, n # load the address of the label n
    lw t3, 0(t3) # get the value that is stored at the address denoted by the label n
fib:
    beq t3, x0, finish # exit loop once we have completed n iterations
    add t2, t1, t0 # new_fib = curr_fib + next_fib;
    mv t0, t1 # curr_fib = next_fib;
    mv t1, t2 # next_fib = new_fib;
    addi t3, t3, -1 # decrement counter
    j fib # loop
finish:
    addi a0, x0, 1 # argument to ecall to execute print integer
    addi a1, t0, 0 # argument to ecall, the value to be printed
    ecall # print integer ecall
    addi a0, x0, 10 # argument to ecall to terminate
    ecall # terminate ecall
```

Exercise 2: Using the Venus Debugger

There are two ways of opening a file in the Venus debugger:

1. Through the editor
 1. **Open** `fib.s` into the Venus editor.
 2. **Click** the "Simulator" tab and **click** the "Assemble & Simulate from Editor" (or the "Re-assemble from Editor") button. The current instruction is highlighted in a light blue color. Similar to `cgdb`, the current instruction is the instruction has not been executed, but is about to be executed.
2. Through the "Files" tab
 1. **Click** the "Venus" tab, then **click** the "Files" tab.
 2. **Navigate** to the `fib.s` file, which should be located in the `labs` folder under `lab03`.
 3. **Click** the "VDB" button next to the name of the file.

This exercise will ask you to write down your answers in `ex2_answers.txt`. The question numbers may be different from the step numbers, please be careful!

1. **Open** `fib.s` in the Venus debugger using one of the two ways listed above.
 - Question 1: What is the machine code of the highlighted instruction? The answer should be a 32 bit hexadecimal number, with the `0x` prefix.
 - Question 2: What is the machine code of the instruction at address `0x34`? The answer should be a 32 bit hexadecimal number, with the `0x` prefix.
2. **Click** the "step" button to advance to the next instruction. The second instruction should now be highlighted.



3. **Click** the "prev" button to undo the last executed instruction. Note that undo may or may not undo operations performed by **ecall**, such as exiting the program or printing to console.
4. On the right side of the screen, **click** the "Registers" tab to view the values of all 32 registers. This tab may be already selected if it the title is highlighted in yellow. Make sure you look at the integer registers, not the floating-point registers.
 - Question 3: What is the value of the **sp** register? The answer should be a 32 bit hexadecimal number, with the **0x** prefix.
5. **Continue stepping** until the value in **t1** changes.
 - Question 4: What is the new value of the **t1** register? The answer should be a 32 bit hexadecimal number, with the **0x** prefix.
 - Question 5: What is the machine code of the current instruction? The answer should be a 32-bit hexadecimal number, with the **0x** prefix.
6. **Step** until you are at address **0x10**. At this point, **t3**'s value has been updated.
 - Question 6: What is the value of the **t3** register? The answer should be a 32 bit hexadecimal number, with the **0x** prefix.
7. If we look at the current instruction, we're loading from the **t3** register. **Use** the "Memory" tab (next to the "Registers" tab), and **input** the answer of question 6 (the value of **t3**) into the "Address" box. You may need to scroll down on the memory tab before it is visible. **Press** "Go" to go to that memory address
 - Question 7: What is the byte that **t3** points to? The answer should be an 8 bit (1 byte) hexadecimal number, with the **0x** prefix.
8. **Set a breakpoint** at address **0x28** by clicking the row at that address. The row should turn light red and a breakpoint symbol should appear.
9. **Continue** until the breakpoint by pressing "Run".
 - Question 8: What is the value of the **t0** register? The answer should be a 32 bit hexadecimal number, with the **0x** prefix.
10. **Continue** 6 more times.
 - Question 9: What is the new value of the **t0** register? The answer should be a 32 bit hexadecimal number, with the **0x** prefix.
11. Sometimes, reading hexadecimal values aren't very helpful. **Set** the display settings to "Decimal" by using the dropdown at the bottom of the register tab. This can also be done for the memory tab.
 - Question 10: What is the value of the **t0** register in decimal? The answer should be a decimal number without a prefix.
12. **Click** the instruction at address **0x28** again to unset the breakpoint.
13. **Click** run to finish running the program, since there are no more breakpoints.
 - Question 11: What is the output of the program? The answer should be a decimal number without a prefix.

Venus: Memcheck

In C programming **valgrind** was the go-to tool for debugging memory access errors (such as **Segmentation fault (core dumped)**). For Venus, we have a feature called "memcheck" that accomplishes something similar. The memcheck error messages are designed to mimic **valgrind** error messages.



Memcheck comes in two modes:

- Normal mode (or just "memcheck"): This mode will show any invalid reads or writes to memory. If there is unfreed memory when the program exits, it will also print out the number of bytes of unfreed memory.
- Verbose mode (or "memcheck verbose"): In addition to normal mode, this mode also prints out every memory read/write, along with a list of blocks that were not freed when the program exits.

You can enable these modes under the Venus tab. If both "Enable Memcheck?" and "Enable Memcheck Verbose?" are selected, memcheck will run in verbose mode.

Lab Task 1- Using Memcheck

This exercise will ask you to write down your answers in [ex3_answers.txt](#). The question numbers may be different from the step numbers, please be careful!

1. **Open** [ex3_memcheck.s](#) in the Venus editor and **read** through the entire program to get an idea of what it does.
2. **Run** the program. Oh no, the program errors! Let's look at the error message.
 - Question 1: What address did the program try to access, but caused the error? The answer should be a 32-bit hexadecimal number, with the **0x** prefix.
 - Question 2: How many bytes was the program trying to access? The answer should be a decimal number without a prefix.
3. This seems like a memory error, so let's give memcheck a shot. **Enable** memcheck (normal mode) and **reopen** [ex3_memcheck.s](#) in VDB.
4. **Run** the program. Look, a memcheck error with more details! **Read** the error carefully.
 - Question 3: What address did the program try to access, but caused the error? The answer should be a 32 bit hexadecimal number, with the **0x** prefix.
 - Question 4: How many bytes were allocated in the block related to the error? The answer should be a number without units.
 - Question 5: Which line of the source file caused this error? The answer should be a number.
5. **Compare** your answer to Question 2 and Question 4. Note that memcheck may change the memory address that **malloc** returns.
6. Let's try to debug this error. **Recall** that **t1** contains the loop counter.
 - Question 6: What is the value of **t1** based on the memcheck error message? The answer should be a decimal number.
7. **Fix** this error in the source code and **save** the file.
8. **Run** the program again. The process complete without any invalid access errors. However, it complains that there's some unfreed memory.
 - Question 7: How many bytes were not freed when the program exited? The answer should be a decimal number without units.
9. **Rerun** the program with memcheck in verbose mode. Remember to reopen the file in VDB.
 - Question 8: What is the address of the block that was not freed? The answer should be a 32 bit hexadecimal number, with the **0x** prefix.
10. **Fix** this error by calling **free**.
11. **Disable** memcheck for the next two tasks



Lab Task 2- Arrays In Assembly

Consider the discrete-valued function f defined on integers in the set $\{-3, -2, -1, 0, 1, 2, 3\}$. Here's the function definition:

```
f(-3) = 6  
f(-2) = 61  
f(-1) = 17  
f(0) = -38  
f(1) = 19  
f(2) = 42  
f(3) = 5
```

Implement the function in `ex4_discrete_fn.s` in RISC-V, with the condition that your code may **NOT** use any branch and/or jump instructions! Make sure that your code is saved locally. We have provided some hints in case you get stuck.

All output values are stored in the output array which is passed to f through register `a1`. You can index into that array to get the output corresponding to the input.

Make sure that you only write to the `t` and `a` register. If you use other registers, strange things may happen (you'll learn about why soon).

HINTS:

1. You can access the values of the array using `lw`.
2. `lw` requires that the offset is an immediate value. When we compute the offset for this problem, it will be stored in a register. Since we cannot use a register as the offset, we can add the value stored in the register to the base address to compute the address of the index that we are interested in. Then we can perform a `lw` with an offset of 0.

In the following example, the index is stored in `t0` and the pointer to the array is stored in `t1`. The size of each element is 4 bytes. In RISC-V, we have to do our own pointer arithmetic, so:

- We need to multiply the index by the size of the elements of the array.
- Then we add this offset to the address of the array to get the address of the element that we wish to read.
- Read the element.

```
slli t2, t0, 2 # step 1 (see above)  
add t2, t2, t1 # step 2 (see above)  
lw t3, 0(t2) # step 3 (see above)
```

3. $f(-3)$ should be stored at offset 0, $f(-2)$ should be stored at offset 1, and so on



Testing

To test your function, open `ex4_discrete_fn_tester.s` and run it through the simulator. make sure that the test passes locally before you submit.

You can also test your code using the command line with the following command.

```
$ java -jar tools/venus.jar lab03/ex4_discrete_fn_tester.s
```

Lab Task 3- Arrays partitioning

An array `array1` contains the sequence:

-1 22 8 35 5 4 11 2 1 78

each element of which is word.

Rearrange the element order in this array such that,

- All the elements smaller than the 3rd element (i.e. 8) are on the left of it,
- All the elements bigger than the 3rd element (i.e. 8) are on the right of it.

Lab Task 4- Factorial

In this task, you will be implementing the factorial function in RISC-V using assembly. This function takes in a single integer parameter `n` and returns `n!`. A stub of this function can be found in the file `factorial.s`.

The argument that is passed into the function is located at the label `n`. You can modify `n` to test different factorials. To implement, you will need to add instructions under the `factorial` label. Note that you may find it helpful to add additional labels to simplify control flow. We recommend that you implement the iterative solution, but you are welcome to implement the recursive solution. You can assume that the factorial function will only be called on positive values with results that won't overflow a 32-bit two's complement integer.

At the start of the factorial call, the register `a0` contains the number which we want to compute the factorial of. Then, place your return value in register `a0` before returning from the function.

Make sure that you only write to the `t` and `a` registers. If you use other registers, strange things may happen (you'll learn about why soon).

Also, make sure you initialize the registers you are using! Venus might show that the registers are initially 0, but in real life they can contain garbage data. Make sure you set the register values that you will be using to some defined number before using them.

Testing



To test your code, you can make sure your function properly returns the correct output. Some examples are $0! = 1$, $3! = 6$, $7! = 5040$ and $8! = 40320$.

To test your function, open `ex5_factorial.s` and run it through the simulator. This will be how we test your function on the autograder, so make sure that the test passes locally before you submit.

You can also test your code using the command line with the following command.

```
java -jar tools/venus.jar lab03/ex5_factorial.s
```

- The report should include your thinking, illustration of implementations, and results of the source code.

Lab Task 5 – Bit Manipulation

Write a RISC-V assembly program that performs various bit manipulation operations on integers. The program should implement a set of operations based on the user's choice:

Toggle Bit: Toggle a specific bit in a given integer.

Check Bit: Check if a specific bit is set in a given integer.

Set Bit: Set a specific bit in a given integer.

Clear Bit: Clear a specific bit in a given integer.

The program should prompt the user for the operation they want to perform and then execute the corresponding function. Each function should take two arguments: the integer to manipulate and the position of the bit to manipulate.

Steps:

Write a RISC-V assembly program that starts by displaying a menu with the options to toggle a bit, check a bit, set a bit, or clear a bit.

Prompt the user for the desired operation.

Prompt the user for the integer to manipulate and the bit position (0-based index).

Based on the user's choice, perform the requested operation using arithmetic operators (addition, subtraction) and bitwise operations (AND, OR, XOR).

Display the result of the operation to the user.

Loop back to the menu to allow the user to perform additional operations.

- XOR (xor) can be used to toggle a bit.
- AND (and) can be used to check if a bit is set.
- OR (or) can be used to set a bit.
- AND with a mask can be used to clear a bit.



A pseudo code is provided:

```
.data

menu: .asciiz "Choose an option:\n1. Toggle Bit\n2. Check Bit\n3. Set Bit\n4. Clear Bit\n"

.text

.global _start

_start:

    # Display menu and get user input

    la a0, menu

    li a7, 64 # syscall for print string

    ecall

    # Get user choice and integer and bit position

    # ...

    # Perform operations based on user choice

    # Call the appropriate function for each choice

toggle_bit:

    # Code to toggle the specified bit

    # ...

check_bit:

    # Code to check if the specified bit is set

    # ...

set_bit:

    # Code to set the specified bit

    # ...

clear_bit:

    # Code to clear the specified bit

    # ...

# Add more functions and loops as needed
```



Reference

<https://inst.eecs.berkeley.edu/~cs61c/fa23/>

<https://www.cse.cuhk.edu.hk/~byu/CENG3420/2022Spring/slides/lab1-2.pdf>