CSC 3210 Computer Organization and programming

LAB 9

Input and Output in RARS

INSTRUCTIONS FOR LAB 9

Tasks to be done in this lab:

- Learn how to read from STDIN.
- Learn how to write to STDOUT.
- Learn how to test a program with piped input, input redirection, and use output redirection.

ASSIGNMENT FOR LAB 9

- There are 3 parts in this lab.
- Answer the questions that are mentioned in bold.

INTRODUCTION

- A common operation in computing is to read some data, process it, and write it out. Sometimes this requires minimal processing. For example, a bank may allow you to download your transactions in .csv form, such as:

  "02/28/2025","-100.00","*","","ATM Withdrawal"

  "03/04/2025","0.22","*","","Interest Payment"

- The asterisk indicates that it was processed, while the empty string holds the check number, and it is empty since these lines do not involve a check.
- Meanwhile, another program tracks the account, and needs input like this:

-	100.00	ATM Withdrawal	February 28, 2025	x
+	0.22	Interest Payment	March 4, 2025	x

INTRODUCTION

- The data is the same, but the order and meaning is slightly different (requiring a symbol in column 1 to indicate addition/subtraction, using "x" instead of "*", spelling out the date, using columns of fixed width for each field).
- To make the data from the .csv file work as the input to the account tracker, some program needs to read in the .csv file and output the format that the account tracker uses. This may sound easy, however parsing input correctly can be challenging.
- In this lab, we will do something similar but easier. We will read in a character, read in an integer, and read in a string.
- After that, we will write these values out in a different order. **We will use RARS for this assignment**, because the VSCode Venus simulator does not support many of the things described here.

PART-1

- We are going to examine how to get input. For the char and int values, we can simply use a few labels in the initialized .data section. However, strings are a bit more challenging since these are effectively arrays of chars.
- We will need a place to store the input string, and here create a buffer of 10 bytes. The 10 is just an example, and in fact the code does not even use that much. A more realistic number is 1000, or even 10000, but whatever number we choose, it might not be enough.
- The .bss section is for uninitialized data. Thus, the following lines should reserve 10 bytes for use with the program. But neither simulator appears to support this (yet).

  .bss
  myarray: 10, 4

- There is another way to do this. It is possible to use #define to specify the amount of space to dedicate to our buffer, where the pre-processor replaces the defined name with the value.

PART-1

- In the code below, there is a space between the pound-sign and the "define", which causes it to be interpreted as a comment. This is intentional. The first two lines simply indicate alternative ways to do this, and are not really needed. Some RISC-V assemblers use .equ, while RARS uses .eqv.

```
# define  BUFFER_SIZE  10
#.equ  BUFFER_SIZE  10
.eqv  BUFFER_SIZE  10
```

- A buffer is simply an array. In the data section, we reserve space with the .space directive. The following says to reserve "BUFFER_SIZE" bytes of space at the label "mybuffer".

```
.align 2
mybuffer: .space   BUFFER_SIZE
```

- The .align 2 is there to avoid a problem. You may get an error like "Store address not aligned to word boundary" meaning that the simulator expects the address to be located on a word offset, not a byte. When the processor reads in a word, it reads in 4 bytes. Having some data at an offset that is not evenly divisible by 4 can cause this error, so we indicate that it should be aligned. This means that some memory might be reserved but not used as a result, since the assembler skips over to the next word boundary.

PART-1

- You may want to add the following definitions, to make your code more readable.

```
.eqv  STDIN       0
.eqv  STDOUT      1
.eqv  STDERR      2
.eqv  READ        63
.eqv  WRITE       64
.eqv  SIMPLE_READ  8
.eqv  SIMPLE_WRITE 4
.eqv  NL          10
```

- STDIN and STDOUT stand for standard input and standard output, respectively. These are the defaults for input and output, but can be "piped" or re-directed when invoking the program from a shell. You are familiar with using a shell, even if you have not heard of this term: when you connect with the SNOWBALL server and type commands at the prompt, you are interacting with a shell.
- As you may have guessed, STDERR means the standard error channel, which could be different than output. We will not use it in this lab, but it is good to know about. You should recognize the NL is the code for a newline character.

PART-1

- A file descriptor is a common input argument, so you should know about it. Normally, we use "fd" for "file descriptor". If we were to read from a file (or from a pipe), we would refer to the input source by the file descriptor. We also use a file descriptor to refer to the output destination. STDIN has a file descriptor value of 0, and is the default input. STDOUT has a file descriptor value of 1, and is the default output.
- The file descriptor is not needed with ecall with a7 values of 4 and 8 in RARS (print string and read string, respectively), since it uses the defaults. However, using the read string ecall does not appear to return the number of characters read. In fact, there is a RISC-V specification for ecall with a7 values of 63 and 64 for read from a file and write to a file, respectively. In these cases, set a0 to the file descriptor.
- Code to read a character appears a little further down the page. We can use it to read a single character from STDIN, i.e. without specifying the file descriptor. Thus, we could read a string as a character at a time, though this is not as efficient as reading multiple characters. But before we read input, we really should let the user know what we want, so print a string first.

```
# print str1
li    a7, SIMPLE_WRITE
la    a0, str1
ecall
```

PART-1

- In the data section, include the string.

```
str1:     .string  "Enter a char "
```

- Now here is the code to read a character.

```
# Read a character
li     a7, 12    # ecall code for read character
ecall
```

- After the ecall returns, it stores the character read, and since it makes more sense to store it as a byte rather than a word, we do that.

```
# Store the character that we read
la     t0, mychar
sb     a0, 0(t0)      # assuming  mychar: .byte  is used
```

- Next, we read an integer, and store it as a word. Have your code print a message indicating to the user what to enter, like we did above.

```
# Read an integer
li a7, 5
ecall
```

PART-1

```
# Store the word that we read
    la  t0, int1
    sw  a0, 0(t0)
```

- Next, we read a string. Remember that we defined "mybuffer" as an array of bytes. The number of characters to read goes into a1. The program will try to read that many, but it is possible that the number actually read is fewer, i.e. if we are at the end of file/input. Have your code print a message indicating to the user what to enter, like we did above.

```
    # Read a string
    li  a7, READ        # read a string from file given by fd
    li  a0, 0           # fd is 0
    la  a1, mybuffer
    li  a2, BUFFER_SIZE
    ecall
```

- After the ecall, the a0 register will contain the number of characters read. We can store this at the label "temp", defined in the data section (or the bss section).

PART-1

```
# remember this number for later
    mv      s0, a0
    # store the number chars read in temp
    la      t1, temp
    sw      a0, 0(t1)           # store number of chars read
```

- Next, compare the number read to 0 with the beq command. If the result is equal, i.e. the number read is zero, branch to the label "eof_reached", which we define later in the program. "EOF" is short for "end of file", and simply means the end of the input, whether the input is a traditional file or not. Otherwise, jump to the label "read_again".

```
    # compare  a0 to 0
    # if we read 0 chars, we reached EOF
    beq     a0, x0, eof_reached  # Did we read 0 chars?
    j       read_again          # Read another string
```

- There is a logic error with the code as presented. The idea is that we read a string, store the number of chars read, check if the number of chars read is zero, and loop to repeat this if not. But we will later want the number of chars read for the last string that we stored, and doing it this way means that the number of chars read will be zero. Thus, what we should do is read a string, check if the number of chars read is zero, if not then store the number of chars read, and loop to repeat this.

PART-1

- The label "read_again" should be defined before the code that reads a string. After reading the string, have the program print the number of characters read. It should do this every time that the user enters a string.
- You will need to define "eof_reached", and put some code there. Note that "2", "13", and "abc" shown in the first three lines were entered by the user.

  Enter a char 2
  Enter an int 13
  Enter a string abc
  Number of chars read is 4
  Enter a string Number of chars read is 0

  The program should end after that.
- Put all of this together, along with anything else that you need to make a fully functioning program. Show the program, and that it assembles and runs.

  java -jar /home/mweeks/rars1_6.jar lab9_pt1.s
- When you run it, it will expect input from the keyboard, so type something in appropriate values and press return. When entering many characters (i.e. a string), you need to specify when you are done. If you simply press return, the number of characters read is 1: the return key. You could have your program look for this, but then it will stop whenever there is a blank line, instead of a true end of input. Press CTRL-D (that is, hold down the control key and press the D key) to indicate the end of your input.

PART-1

QUESTIONS:

1. **Why are "Enter a string" and "Number of chars read is 0" on the same line?**
2. **Suppose that the user does not follow directions, and enters "1" for "Enter a char", then "a" for "Enter an int". What will happen?**
3. **Did you (or the given code) need to do anything to make sure that the register holding the length is not over-written?**

PART-2

- One problem with the code from part 1 is that it's hard to verify that it works. In this part, we will write the input back to the STDOUT. Code to do this is as follows. Remember to copy your lab9_pt1.s code to a new file (lab9_pt2.s) before getting started with this part.

```
# write a string
li  a7, WRITE      # write to file
li  a0, STDOUT     # fd for STDOUT
la  a1, mybuffer
la  t0, temp
lw  a2, 0(t0)      # number of chars
ecall
```

- The code below can be used to print the integer and the char.

```
# print int1
la    t0, int1
lw    a0, 0(t0)          # Value to print
li    a7, 1
ecall
```

PART-2

- This is how we print a space (ASCII value 32).

```
# print space
li    a7, 11
li    a0, 32
ecall
```

- The count of characters to print goes into the a2 register. The a1 register gets the buffer's address. The a0 register stores the file descriptor, and we use the STDOUT value (1). The a7 register contains the action to perform, here it uses the value 64 (write to a file). Note that the order that we put these values into these registers does not matter, as long as the registers have the correct values when the computer reaches the ecall command.
- However we do it, we will not end up with a program that reads in all input then writes all of the output. This is because we do not know ahead of time exactly how much input there will be. (There can be exceptions.
- For example, if we are dealing with files, we can determine the file size, dynamically allocate just enough memory to hold it all, then read the file in all at once. But this does not guarantee that the program will be efficient, such as if the file size exceeds available RAM.) In general, we will not know the size of the input in advance. Instead, the program will read in some input, write the output, then repeat the process until we reach the end of the input.

PART-2

- The only decision that we will make is how much space to dedicate to the buffer in advance. When you work with multiple characters at a time, be aware that the count used for output should be the number of characters actually read, not the number that you were expecting to read.
- To make this work with the code from part 1, the code performing the write should be done between the check for EOF and the jump to "read_again". Like before, show your program for part 2, assemble and run it, and show that it works.
- It should print the values read, with appropriate text before it, and use another (printable) character to surround the values. In the example output below, it shows exclamation marks around the values, but you can use something else.

    Enter a char 2
    Enter an int 13
    Enter a string abc
    Number of chars read is 4
    Enter a string Number of chars read is 0
    String read is !abc
    !
    Int value read is !13!
    Char value read is !50!

PART-2

QUESTIONS:

1. **Why are the exclamation marks on different lines?**
2. **What happens if the user enters several strings before entering CTRL-D?**
3. **Why is the char value reported as 50?**
4. **What happens if the user types CTRL-D (without entering anyting else) the first time that the program asks for a string?**
5. **What happens if the user types CTRL-D (after entering "abc") the first time that the program asks for a string?**
6. **If you enter a string that is larger than the buffer, how many chars read does the program report?**
7. **If you enter a string that is larger than the buffer, then enter a string that is shorter and press CTRL-D, what does the program report as the string read, and why?**

PART-3

- Remember to copy your lab9_pt2.s code to a new file (lab9_pt3.s) before getting started with this part. For this part, comment out the code that prints the prompts. We will expect that the user "knows" what to enter because the input will be entered all at once.
- The defaults for standard input and standard output are the keyboard and the terminal. On a computer like SNOWBALL, running Linux/Unix, you can easily redirect input/output to/from a program. You can create a file of test input and use it with the program. Here is one to use; call it "testfile.txt".
- Ideally, we should be able to use the code from lab9_pt2.s as is, but there are a few problems (not counting the prompts appearing, which is just an aesthetic issue):
  - Using the read with an fd (READ) does not appear to work when input is redirected. (This could be a bug with RARS. This behavior is not seen in other environments, like C.)
  - Using the read string (SIMPLE_READ), we do not know how long the string is.
  - The read string function includes the newline character, even when the string is empty.

PART-3

We can address these problems by:

- Use SIMPLE_READ instead of READ.
- Checking for a newline character in the buffer, instead of length.

```
# Did we get a NL?
la      t0, mybuffer
lb      a0, 0(t0)           # Char to check
li      a1, NL
beq     a0, a1, eof_reached  # Did we read 0 chars?
```

- Using the write string (SIMPLE_WRITE) instead of WRITE, since the latter expects the number of characters.

We could have taken a different approach: scanning the input to find the NL char, counting each character, then reporting the length. This does not get around the problem of redirected STDIN not working as expected.

One way to specify the input, in a non-interactive way, is to pipe the output from one command to the program. The next two examples do this.

```
echo "7\n8\nabc" | java -jar /home/mweeks/rars1_6.jar lab9_pt3.s
```

PART-3

- The "echo" command normally echoes the string to the output, so a command like echo "abc" simply prints "abc". In the example above, the string "7\n8\nabc" is piped to the lab9_pt3 program. After the last character ("c") is reached, the program should attempt another read, find that 0 characters were read, and quit.
- If there is a problem with the program repeating part of the last string when it has already found the EOF, come up with a way to fix it. For example,

  ```
  echo "s\n42\ncat" |  java -jar rars1_6.jar lab9_pt3.s
  ```

  might show the string "cat" followed by a newline, then show another string consisting of a newline, "t", then another newline.
- Show that your program works.
- The second example uses the "cat" command, and it will output the contents of "testfile.txt". Here, however, it sends the output from the cat command to the lab9_pt3 as the input.

  ```
  cat testfile.txt | java -jar /home/mweeks/rars1_6.jar lab9_pt3.s
  ```
- Show that this works.

PART-3

- Next, we have examples of redirection using files. First, the following command says to use "testfile.txt" as the input to "lab9_pt3".

  java -jar /home/mweeks/rars1_6.jar lab9_pt3.s < testfile.txt

- Verify that this works. It should appear to be the same as the "cat" example from earlier.
- A second file redirection is as follows.

  java -jar /home/mweeks/rars1_6.jar lab9_pt3.s > testout

- When you run that, it will send the output from "lab9_pt3" to "testout", overwriting the "testout" file if it already exists. However, the program expects input, and this command does not specify an alternate input source. So it will still expect you to type using the keyboard until it gets CTRL-D.
- Therefore, anything that you type, before the CTRL-D, will be stored in the "testout" file. Try this, and show that it works. Use "cat" on "testout" to verify it.

PART-3

- Now we can put the input redirection and output redirection together:

  java -jar /home/mweeks/rars1_6.jar lab9_pt3.s < testfile.txt > testout

- This causes "testfile.txt" to be the input, and "testout" receives the output. When done, the two files should be the same. Try it, and verify this with the "diff" command.
- You might wonder why we have gone to so much trouble making a program that simply echos the input to the output. The idea here is to give you some experience with a program that can read input and write output.
- You could easily add to this program to do something more interesting, such as filtering out non-ASCII characters from a file, or automatically capitalizing a file's contents, or automatically making a file's contents lower case.

PART-3

Questions:

1. **Do we get the same results whether the input comes from "echo", "cat", or input redirection ("<")?**
2. **Do we get the same results whether the output goes to the terminal or a file using output redirection (">")?**
3. **What is the advantage of using input redirection and output redirection versus entering it interactively?**

IMPORTANT NOTE:

Remember that we will grade your lab report so it is vital to turn that in. The other files (your code, a text version of any log file, etc.) are to document your work in case we need more information.