Extra Tips for Lab4

# Tip1

Please ***THOROUGHLY*** read the Lab4 documentation in order to understand the working of the Lab4 testbench assembly file and how it uses all the defined functions! In the Lab4 document, you absolutely MUST read the section *“Memory arrangement as defined in Lab4”.* This informs you how the emulated RISC V machine we are working on has its memory compartmentalized as per the requirements of Lab4.

You also MUST read the section *“Student Coded Functions”* to see examples of the arguments a function takes in and what it returns.

# Tip 2

Allocate\_file\_record\_pointers.asm already does 70% of the workload for the Lab4 testbench to work correctly and it has been already provided to you. You can see how it works by assuming that data.csv has the following string:

“Kruger Industrial Smoothing,365\r\nKramerica,0\r\n”

We are working with a small data.csv to get the idea more easily across.

The following discussion rests on the assumption that you have already followed Tip 1 and have read the sections *“Memory arrangement as defined in Lab4”* and *“Student Coded Functions”.*

The string starts at address 0xffff 0000 (where the first character ‘K’ of the first record’s name is stored as a byte), the record’s numeric data starts at 0xffff 001c (location of ‘3’character byte), the second record’s name starts at 0xffff 0021 (location of ‘K’ character byte from Kramerica) and this record’s numeric data starts at 0xffff 002b (location of ‘0’character byte),

If we are to parse this csv file in order to extract a record’s name or numeric data, just knowing the above locations is enough. After all, we know a name ends with a “,” followed by the most significant digit of the numeric data, and the numeric data itself concludes when a ‘\r’ character byte appears.

In the RISC V main memory, we allocate the RISC V heap memory (starting at 0x1004 0000) to create a table of pointers that point to the addresses mentioned above.

These pointer values are 1 word long each so occupy consecutive 4 bytes in memory.

Visualize the pointer table as having 2 columns. Column 1 stores a pointer to a record’s name while column 2 points to the record’s numeric data. If there are n records, the table will have n rows. For our example data.csv, the table would look like this:

|  | Name pointer(Column 1) | Numeric data pointer (Column 2) |
| --- | --- | --- |
| Record 1 | 0xffff 0000 | 0xffff 001c |
| Record 2 | 0xffff 0021 | 0xffff 002b |

Obviously, we can’t fit a 2D table like this physically into RISC V memory. So, the idea is to break down this table ***row-wise*** into its constituent cells and place them one after another. In Lab4, these cells each occupy 1 word (4 bytes) and therefore they are all placed in a word array in the following format in main memory:

| 0xffff 0000 | 0xffff 001c | 0xffff 0021 | 0xffff 002b |
| --- | --- | --- | --- |

Indeed, this is how we store any general 2D data (even 3D,4D,5D, etc.) in a physical memory.

For the above word array, the base address is 0x1004 0000.

Hopefully, this tip gave you a better understanding of how we allocate file record pointers.

# Tip 3

Let’s talk about income\_from\_record.asm As stated in the section *“Student coded functions”*, income\_from\_record function accepts as input the Column 2 data of the file records pointers table. So, if we want to find out the numeric data of record 1 from the above example, we pass the argument 0x1004 0004 to the function. Why? Because in the word array we described just a while ago, note that this is the address where we have the pointer to record 1’s numeric data, 0xffff 001c.

So simply load the data from the argument (0x1004 0004 in our current example) and put it in a register. This register now points to the location of the original numeric data in the file buffer, i.e. 0xffff 001c.

So, start iteratively loading bytes starting at the address. 0xffff 001c until you hit a ‘\r’.

So, we have now successfully found a way to inspect each character byte in a loop from the numeric data!

But the problem now appears to be this: whereas for a ‘1’ digit, you expected a 0x01 byte, you are getting 0x31 instead, or maybe instead of 0x03 for ‘3’, you are getting a 0x33 instead. What happened? Did RARS give up? At this point, we have to be aware that all the characters in RARS are encoded in ASCII. So, if you were to consult the ASCII chart, you would find that the ASCII equivalent of ‘1’ is 0x31 or 49, and for ‘3’, it is 0x33 or 51. So that means in your loop where you iterate over the numeric data characters, you need to convert them to the intended integer value digits di. Then, you can gather these digits through the formula I taught in class lectures during base conversion and get the original numeric data, i.e.:

∑di\*10i , where i goes from 0 to n-1 for an n digit number.

So, for ‘365’, the final outcome would be 3\*100 + 6\*10 + 5\*1

Now the question still remains: how do you map from a character’s ASCII value to its original digit value? You could create space for a new mapping table somewhere in main memory that does this but you are being a TERRIBLE programmer if you frequently access memory in your source code when much simpler solutions exist, whether it’s for an undergrad project assignment or working in Tech. Because memory access requires MANY MANY extra clock cycles and your code runs extremely slow as a result.

So, apply that very simple ***one line of code*** that does the ASCII mapping correctly instead of pointlessly disturbing the memory for a mapping table. In fact, it is in your best interests that you leave the memory undisturbed by not putting any extra data in it because we will be using some more extra space for the auto grader to put extra stuff for verification during the grading process.

# Tip 4

For the functions max\_income and total\_income, use the income\_from\_record function that you ***correctly*** coded to extract the numeric data from each record.

The pseudo code for total\_income should be very self-evident.

For max\_income, start with the first record in the file and assume initially that is the one with max income.

Then through iterating over each remaining record in the file, if ith record has a larger value than what was initially in max\_income, then simply update the value of max\_income as well as the name of the record which has it.

For simplification, we will assume that the maximum income in data.csv does not repeat in multiple records.

A similar rationale applies to the min\_income function as well.

# Tip 5

As stated in the Lab 4 documentation, you are getting 10 points even if you do absolutely nothing. Your only contribution would be to push the provided lab4 material to your git repo and that would automatically earn you 10 points.

However, it is strongly encouraged that you get all full points.

A good coding tip would be to create copies of an asm file which you need to code in. Example, for income\_from\_record.asm, create a copy income\_from\_record\_v2.asm. Now update the .include statement in the lab4 testbench file so that now you can freely modify the v2 asm without worrying if you accidentally removed code that was already working in it.

*Just make sure to change the* ***“ .include "income\_from\_record\_v2.asm"*** *to the original* ***“ .include "income\_from\_record.asm"*** *before pushing the lab 4 testbench file or else it might not compile for the auto grader.*