Introduction

In this lab, you will implement a protocol decoder which can decode a parallel signal into bytes. This lab is intended to familiarize you with the process of writing and submitting code for this course.

Your Code

You will write your code in the `code/` directory. It will be graded by running `cd ./code ; make`, and then running test cases on `./a.out`. In between each test case, your code will be cleaned using `cd ./code ; make clean`. All compilation and grading will use the CSCE Linux labs as a reference system (e.g. if you can show your code works on one of the Linux lab machines, it will be considered to work for the purposes of this course). **Be sure to test your code in the Linux labs even if you write it on your own personal system**.

You may use any programming language for which a compiler or interpreter is provided in the Linux labs, with the following caveats:

- We will provide you with templates for C and Python. If you choose to use a different language, you accept full responsibility for ensuring your `Makefile` works with the provided grading script.
  - The TAs will help you with solving the projects only on a best-effort basis if you use a language other than Python or C.
- You may use third-party libraries or tools for aspects of your program which are not related to the lab assignment; for example, a student completing the project in C might choose to integrate a third-party hash-table library. You **must** clearly document the origin of any such libraries and clearly indicate they are not your own work.
  - For students using Python, please create an appropriate `requirements.txt` file in the same folder as your makefile.
- You **may not** use libraries or tools which solve the assignment for you. For example, using an off-the-shelf protocol decoder as a library constitutes cheating and is not permitted.
  - You do not have to explicitly cite libraries, sample code, or templates we give you as part of the course materials, we assume you are using them.
- If you write code that attempts to interfere with the grading scripts, this is considered cheating and you will receive a 0.

Running Your Code

We will run your code via the provided `grade.sh` script. You can run your code through all test cases provided using the shell command `sh grade.sh`. The grading script can also run only specific tests, and supports other more advanced options for debugging. Please run `sh grade.sh --help` for more information.

Keep in mind, the `grade.sh` script **is the only supported way to run your code**. Especially if you use Python, it is important you run your code via this script or it will not be setup with the proper virtual environment.

After running normally, the `sh grade.sh` script will display your anticipated score on the assignment. This score is an **estimate** based on the performance of your program with the provided test cases. Your code will be graded using different test cases, however the test cases we provide will be representative to the greatest extent possible (e.g. edge cases in the test cases uses for grading will also appear in the ones we provide as examples).

Before you run your code, you should create the file `userid.txt` in the same folder as your `Makefile`. In this file, you should write your UofSC network ID. For example, if your school email is `jsmith@email.sc.edu`, you should write `jsmith` into the `userid.txt` file.

Requirements

You will write a program which reads in pre-recorded signal data, which will allow it to determine the value of various digital signals at specific points in time. In this case, the data represents a parallel bus, which uses 8 separate 1-bit digital data signals, and one clock signal to transmit one byte at a time. The C and Python project templates include libraries (see *Appendix B*, and *Appendix C* respectively) for parsing the input signal, but you can write your own if you wish, see *Appendix A* for a description of the input format.

The eight digital signals are labeled `s0` to `s7`, and the clock signal is labeled `clk`. Data is arranged such that `s7` stores the most significant bit (MSB) of each data byte, and `s0` stores the least significant big (LSB) of each data byte. Data should be sampled on the rising clock edge.

For each clock edge sampled, your program should output one line containing the hexadecimal value of the byte read. Note that you should use lower case letters (e.g. `ab`, not `AB`), you should NOT include a leading `0x` (e.g. `ab`, not `0xab`), and you SHOULD include a leading zero (e.g. `01` not `1`).

HINT: in Python, you can format your output with `"print({:02x}".format(some_number))`, and in C you can use `printf("%02x\n", some_number)`.

For example, consider the following signal:


```
        +------------------...
        |
s0    --+

        +----+  +-----------...
        |    |  |
s1    --+    +--+


s2    ----------------------...


s3    ----------------------...

            +---------------...
            |
s4    -------+

        +---+
        |   |
s5    ---+   +---------------...


s6    ------------------------...

                +-------+
                |       |
s7    ----------+       +------...

          +----+    +----+
          |    |    |    |
clk   ----+    +----+    +----...
```

We observe that at the first rising clock edge, the signal values `s7`, `s6`, ...,`s1`, `s0` are: 0, 0, 1, 0, 0, 0, 1, 1, and for the second clock edge, the signal values are 1, 0, 0, 1, 0, 0, 1, 1. This means the output from our program should be:

```
23
93
```

Notice that `0b00100011` in binary is `0x23` in hexadecimal, and `0b10010011` in binary is `0x93` in hexadecimal.

Submitting Your Code

When you are ready to turn in your code for grading, you should run the command `sh grade.sh --pack`. A file will be created called `lab_2021sp_parallel_youruserid.tar.gz`, with `youruserid` replaced with whatever user ID you specified previously.

Rubric

Except for the example test case shown earlier in this document, all of the other test cases are generated randomly. We will run your code on a similar (but different) set of randomly generated test cases created by the same program. Your score on the assignment will be the fraction of those test cases for which your program produces the correct output.

Appendices

Appendix A - Input Data Format

Note that parsing libraries are provided in the project templates for C and Python. You do not need to write your own parser for the input data format unless you are using a language other than C or Python. If you are using C or Python and wish to write your own parser, this is allowed but not recommended. You are solely responsible for any errors resulting from improperly written parsers.

Input data will be provided in the following format:

- The first line of input will contain a single unsigned decimal integer number, indicating the number of data samples in the file. Call this value n.
- The second line of input will contain a list of signal names, delineated using tab characters.
- The third line of the file will contain the size of each signal in bits as an unsigned decimal number, delineated by tab characters, such that the k-th size is associated with the k-th signal name from the second line.
- The remaining n lines will each contain data samples, consisting of a floating-point timestamp, followed by signal values as unsigned decimal integers, such that the k-th integer corresponds to the k-th signal name from the second line.
- The value of a signal is assumed not to change between data samples.

- Data samples need not occur at any particular interval in terms of timestamp.
- Timestamps are unitless; they represent some arbitrary amount of time, unless the assignment description prescribes a particular unit of time.

Any lines which begin with any amount of whitespace followed by a `#` character should be ignored as comments, and parsing should resume as normal after a `\n` character is encountered.

For example, consider the file:

```
4
s0  s1  clk
1   3   1
0.001   1   3   0
0.002   0   2   0
0.003   1   3   1
0.004   0   0   0
```

This file indicates that:

- There are three signals, named `s0`, `s1`, and `clk`.
- The signal `s1` is three bits in size, and the signals `s0` and `clk` are each one bit in size.
- At time `0.001`, the signal `s0` has a value of `0b1`, the signal `s1` has a value of `0b011`, and the signal `clk` has a value of `0b0`
- At time `0.002`, the signal `s0` has a value of `0b0`, the signal `s1` has a value of `0b010`, and the signal `clk` has a value of `0b0`
- At time `0.003`, the signal `s0` has a value of `0b1`, the signal `s1` has a value of `0b011`, and the signal `clk` has a value of `0b1`
- At time `0.004`, the signal `s0` has a value of `0b0`, the signal `s1` has a value of `0b000`, and the signal `clk` has a value of `0b0`

Appendix B - Python Library Documentation

Full documentation can be seen in the docstring comments in `utils/python_utils/waves.py`. A summary is provided below.

You will need to instance one `Waves` object by using `w = Waves()` (you can use a variable name other than `w` if you wish). You can then load data into it using `w.loadText(string)`. For example, to load all data on standard input, `w.loadText(sys.stdin.read())`. An example is provided in the project template.

Sample data for the Waves object `w` is stored in `w.data`, which is an array. The *i*-th array element stores the sample data for sample index *i*. The array elements are tuples of two elements; the first element

is the floating-point timestamp at which the sample was collected, and the second is a dictionary associating signal names with the value that signal had at the given timestamp.

In short:

- The value of the signal `"x"` at sample index `i` is given by `w.data[i][1]["x"]`.
- The timestamp of sample index `i` is given by `w.data[i][0]`.

Summary of useful functions for a given Waves object `w`:

- `w.signals()-> list[str]` - return a list of signals in the object.
- `w.samples()-> int` - return the total number of samples stored in the object.
- `w.indexOfTime(time: float)-> int` - return the sample index for the given time value.
- `w.signalAt(signal: str, time: float)->int` - return the value of the specified signal at the specified time.
- `w.nextEdge(signal: str, time: float, posedge: bool=True, negedge: bool=True)-> tuple[float, bool]`
  - return a tuple `(t, ok)` where `t >= time` is the time at which the next signal edge occurs for the given signal, and `ok` is `True` if an edge was found, and `False` otherwise.
  - `w.nextEdge("x", 0.3, posedge=True, negedge=False)` finds the next rising edge of the signal `"x"` which occurs at or after 0.3 time units.
- `w.loadText(text: str)` - overwrite the data stored in `w` with the contents of a file formatted as described in *Appendix A*.

Appendix C - C Library Documentation

Please view `utils/c_utils/waves.h` for full C library documentation. An example of usage is provided in the C project template.

Appendix D - Log Format

For each test case that is run, a folder will be created, `./log/testcasename/`. This folder may contain the following files:

- `explain.err` - standard error from the `explain.sh` script (usually only useful to TAs).
- `explain.out` - standard out from the `explain.sh` script (usually only useful to TAs).

- `make_clean.err` - standard error from running `make clean` before your code is run on the test case.
- `make_clean.out` - standard out from running `make clean` before your code is run on the test case.
- `make.err` - standard error from running `make` before your code is run on the test case.
- `make.out` - standard output from running `make` before your code is run on the test case.
- `run.err` - standard error from running your program on the test case input.
- `run.out` - standard output from running your program on the test case input.

Appendix E - Environment Variables

For students who wish to write their own files into the log directory for a test case, certain environment variables are exposed to the process running under the grading script.

- `CSCE317_PROJECT_DIR` - the directory in which `grade.sh` resides.
- `CSCE317_CODE_DIR` - the directory where your code lives, where `make` is run.
- `CSCE317_CASE_ID` - the test case identifier of the case that is being run.
- `CSCE317_LOG_DIR` - the log directory for the running test case.

Utilizing these environment variables is not necessary to complete the assignment, they are merely presented here for those that wish to use them.

Appendix F - Viewing Input Files With `gtkwave`

To help you understand a particular test case better, you may wish to use the GTKWave VCD file viewer to display the contents of a file formatted according to *Appendix A*. To do so, follow these steps:

1. Ensure GTKWave is installed.
2. `sh grade.sh --vcd2txt test_cases/case002/input.txt temp.vcd`
   - Replace `case002` with the test case you are interested in.
3. `gtkwave temp.vcd`
4. `rm temp.vcd`

