Introduction

In this lab, you will implement a protocol decoder which can decode an SPI signal. This lab has several components. In the first component, you will decode simple two-exchange transactions, next you will extend your code to decode both two-exchange and streaming transactions. Finally, you will add support for handling arbitrary values of CPOL and CPHA.

Note that the rubric for this lab is structured so that students who correctly complete the first part will get a C, students who complete the first and second parts a B, and students who complete all three an A.

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.

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. The following signals are present in the input waveform:

- `cpha` - CPHA value for the SPI bus (guaranteed to be 0 except in part 3).
- `cpol` - CPOL value for the SPI bus (guaranteed to be 0 except in part 3).
- `miso` - MISO (master in slave out) signal for the SPI bus.
- `mosi` - MOSI (master out slave in) signal for the SPI bus.
- `sclk` - SCLK (serial clock) signal for the SPI bus.
- `ss` - SS (slave select) signal for the SPI bus - **active low**.

Note: note that a correct solution to part 2 is also a correct solution to part 1, and a correct solution to part 3 is also a correct solution to parts 1 and 2. Thus, you do not need to write multiple separate programs.

Note: All hexadecimal numbers should be formatted **without** a leading `0x` and in all-lowercase, and left-zero-padded to be a full byte long.

Note: for all parts of this assignment, you should assume that SPI data is transmitted in MSB-first bit order.

Note: for all parts of this assignment, you may assume that the bus does not have any signal transmission errors, and that all transactions are fully complete (e.g. we don't expect you to handle invalid, corrupted, or partial transactions).

Part 1 - Two-Exchange Transactions

In Part 1, the SPI bus will always have the settings CPHA=0, CPOL=0, and will only contain transactions that contain exactly two exchanges. Each transaction can represent either a read or a write. Both types of transactions begin with a transaction from the master to the slave, where the slave transmits only 0 values to the master, and the master transmits a byte with the following format:

bits 7 (MSB)…2	bit 1	bit 0 (LSB)
address	read/write	stream

Thus, the master transmits a 6 bit address, followed by a `1` if a write is requested, and a `0` if a read is requested. Finally, the `stream` field will always be 0 for this part of the lab.

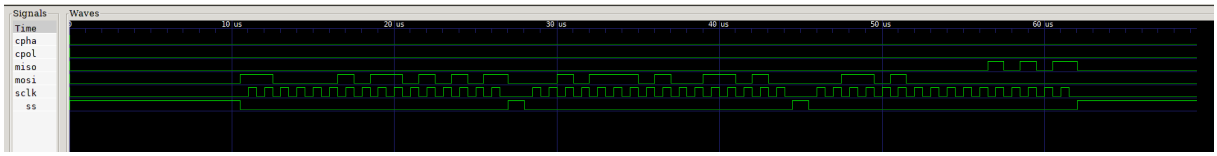
If the first exchange was a read request, the second exchange will send 0 from the master to the slave, and an arbitrary data byte from the slave to the master.

If the first exchange was a write request, the second exchange will send an arbitrary data byte from the master to the slave, and 0 from the slave to the master.

Your program must process the data from the SPI bus, and output a textual log of what events occurred on the bus. Your program will output one line of output for each transaction, in the following format:

- For a read request, the output should be `RD <address> <value>`, where `<address>` is replaced by the address in hexadecimal, and `<value>` is replaced with the value the slave responded with, also in hexadecimal.
- For a write request, the output should be `WR <address> <value>` where `<address>` is replaced by the address in hexadecimal, and `<value>` is replaced with the value the master wrote to the slave.

As an example, consider the following signal trace:



This trace contains three two-exchange transactions:

- The first writes to address `0x30` the value `0xd5`.
- The second writes to address `0x0b` the value `0x9a`.
- The third reads from the address `0x0d` and gets the value `0x15` as a result.

This should correspond to the following output:

```
WR 30 d5
WR 0b 9a
RD 0d 15
```

You can test your code against just this simple example using the command `sh grade.sh --only case000`.

You can test your code against just the test cases for part 1 using this command: `sh grade.sh --only $(cd test_cases ; echo part1_*)`.

Part 2 - Streaming Transactions

In part 2, the SPI bus will work exactly as in part 1, except that for some transactions, the `stream` bit may be set to `1`. In this case, the second exchange will always be from the master to the slave, and will encode a value *N* (in this exchange, the slave always transmits 0 to the master).

The subsequent *N* many exchanges will be sent either from the master to the slave or the slave to the master (depending on whether the read/write flag is 1 or 0 respectively).

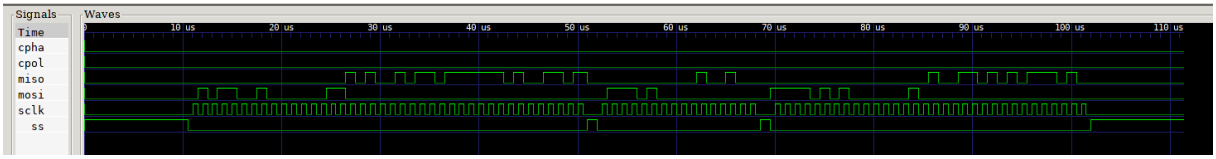
Your program must still output data in the same format as in part 1, but in the event that a transaction is encountered where the `stream` field is 1, it should instead output data in the following format:

- For a read request, the output should be `RD STREAM <address> <value 1> <value 2> ... <value n>`, where the `<address>` is replaced with the address read, and the `<value >`s replaced with the values streamed in order.
- For a read request, the output should be `WR STREAM <address> <value 1> <value 2> ... <value n>`, where the `<address>` is replaced with the address read, and the `<value >`s replaced with the values streamed in order.

NOTE: *N* will never be greater than 32.

NOTE: the output from your program is whitespace-sensitive, take care not to accidentally leave a trailing whitespace while concatenating the list of values.

As an example, consider the following example:



This signal trace contains three transactions and 11 exchanges, which are shown in the following table:

timestamp	slave to master	master to slave
1100.000000	00	59
1900.000000	00	03
2700.000000	a5	00
3500.000000	bf	00
4300.000000	4d	00
5250.000000	00	74
6050.000000	24	00
7000.000000	00	f5
7800.000000	00	02
8600.000000	9a	00
9400.000000	ba	00

Transaction 1:

- The first exchange sends the byte `0x59` to the slave, which corresponds to an address of `0x16`, with a read/write flag of 0 (signifying a read), and a stream flag of 1, indicating this is a streaming transaction.
- The next exchange sends the byte `0x03` to the slave, indicating a streaming read of 3 bytes.
- The subsequent 3 transactions send the values `0x15`, `0xbf`, and `0x4d` from the slave to the master.

Transaction 2:

- The first exchange sends the byte `0x74` to the slave, indicating an address of `0x1d`, a read/write flag of 0, and a stream flag of 0, indicating this is a single read operation.
- The second exchange sends the byte `0x24` from the slave to the master.

Transaction 3:

- The first exchange sends the byte `0xf5` from the master to the slave, indicating an address of 0x3d, a read/write flag of 0, and a stream flag of 1, indicating a streaming read.
- The second exchange sends the byte `0x02` from the master to the slave, indicating that the streaming read is of length 2.
- The final two exchanges send the bytes `0x9a` and `0xba` from the slave to the master.

This should correspond to the following output:

```
RD STREAM 16 a5 bf 4d
RD 1d 24
RD STREAM 3d 9a ba
```

You can test your code against just this simple example using the command `sh grade.sh --only case001`.

You can test your code against just the test cases for part 2 using this command: `sh grade.sh --only $(cd test_cases ; echo part2_*)`.

Part 3

In part 3, the SPI bus will work exactly as in part 2, and the output format of your program will not change. However, your program must read the value of the `cpol` and `cpha` signals when it first starts up and configure itself accordingly. The observed behavior of the other bus signals is guaranteed to be consistent with the settings for CPOL and CPHA.

Note: the value of CPOL and CPHA is guaranteed not to change during the course of a given test case, but may vary between test cases.

Hint: you can reasonably expect that few if any test cases in part 3 will have CPOL=0 and CPHA=0, so as to differentiate them from the test cases of part 2.

You can test your code against just the test cases for part 3 using this command: `sh grade.sh --only $(cd test_cases ; echo part3_*)`.

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_2023sp_spi_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. The test cases are split into three sets corresponding to parts 1, 2, and 3. Your score on each part will be equal to the fraction of test cases you got right in each part.

- part 1 - 72 points
- part 2 - 16 points
- part 3 - 12 points

Total points possible: 100.

By design, students completing part 1 only will earn a C, students completing parts 1 and 2 a B, and students completing all three parts an A.

Note: you can view your overall score even if some test cases throw errors using the command `sh grade.sh --continue`.

Note: due to implementation details of the grading script, you will see an "estimated score" on a scale of 30 points - note that this will be re-scaled to be out of 100 points (e.g. look at the percentage score, not the point counts).

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.

- `CSCE491_PROJECT_DIR` - the directory in which `grade.sh` resides.
- `CSCE491_CODE_DIR` - the directory where your code lives, where `make` is run.
- `CSCE491_CASE_ID` - the test case identifier of the case that is being run.
- `CSCE491_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 --text2vcd 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`

