Lab 2: Serial Peripheral Interface (SPI)

1 Introduction

1.1 Purpose

The purpose of this assignment is to gain experience in designing RTL for tapeout in a real chip. This lab will feature designing a module with both positive-edge-sensitive and negative-edge-sensitive behavior. Further, the lab will begin to introduce the concept of off-chip interfaces, and ultimately provide RTL which will be used in your final chip implementation.

1.2 Background

The Serial Peripheral Interface (SPI) is a common interface used for communicating with electronics. It implements synchronous serial communication, meaning that the signals are clocked and that the signal is a single bit (as opposed to a multi-bit bus). The general idea of SPI is that an n-bit data word is transmitted one bit per cycle from one device to another. Figure 1 shows an example where $n = 8$.

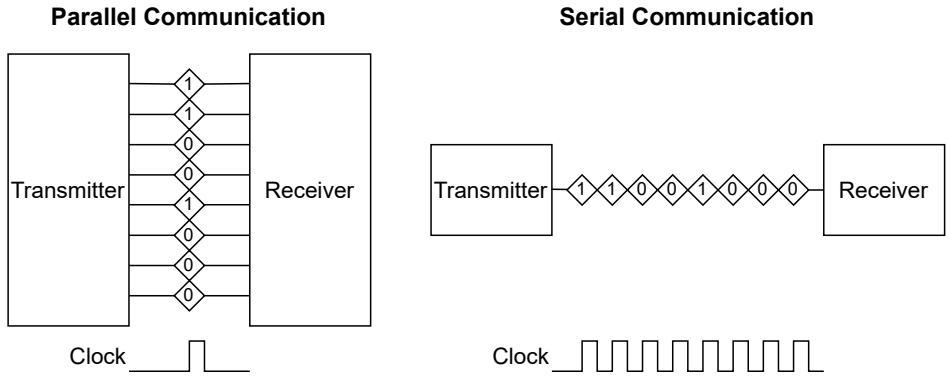

Figure 1: Example of parallel vs. serial communication for 8 bits of data

SPI modules are common in low-end electronics and other embedded systems because of their simplicity. SPI is implemented with only 4 I/Os and can communicate arbitrary data, albeit slowly (usually under 100 Mb/s). There is a good chance you have already used SPI if you have taken an embedded systems course.

Many high-performance electronics also use SPI (or similar) for *out-of-band communication*, meaning communicating data outside of the primary high-speed interfaces (PCIe, DDR, etc.). High-speed interfaces often require delicate calibration, configuration, or other setup to initialize the communication channel when the chip is powered up. To avoid these issues, SPI can be used because it doesn't require any calibration or initialization (aside from asserting a reset signal) and is extremely useful for sending configuration data and other data where bandwidth is not a concern.

To summarize, we use SPI because it is a very simple interface, and *the simpler something is, the easier it is for it to be bug-free*. When designing a real chip, you do not want to have a bug in your interface which prevents you from testing the rest of the chip.

2 Task Overview

You are provided the system diagram as shown which includes an external device and your chip. Your chip consists of a SPI module, which you will need to implement in SystemVerilog, and an on-chip RAM, which is provided to you.

SPI interfaces need to communicate with *something*, so your SPI module will be translating messages sent from off-chip in SPI mode 0 (shift data out on negedge `sclk`, sample data on posedge `sclk`) and interfaces it to an on-chip memory. You will also need to implement SystemVerilog test benches to verify that your module complies with the specifications outlined in this assignment.

3 SPI Protocol Specification

SPI forms a communication channel between two devices: one which is the **main** device and one which is the **sub** device[1]. In real chips, the main device is often implemented by a test harness and the sub device is implemented on-chip. This is illustrated in Figure 2. You will be designing the Main module (as a test bench) and the Sub Module (as a SystemVerilog design).

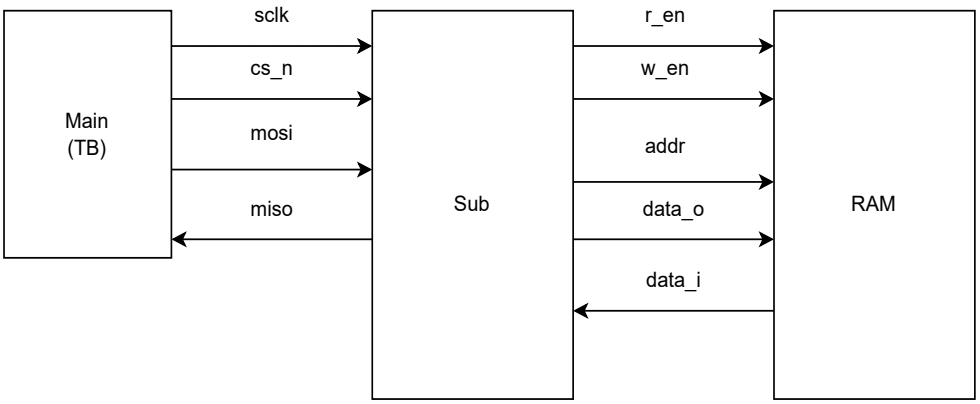

Figure 2: SPI system showing Main-to-Sub communication and Sub-to-RAM interface.

SPI has only 4 signals, which are defined in Table 1.

Signal	I/O	Meaning	Definition
`sclk`	In	Sub clock	Primary clock which the SPI Sub runs on
`cs_n`	In	Chip select	Indicates whether the SPI is active. `n` indicates this is an **active-low** signal. In other words, this is an active-high, synchronous reset
`mosi`	In	Main out, Sub in	Data signal from Main interface to Sub interface
`miso`	Out	Main in, Sub out	Data signal from Sub interface to Main interface

Table 1: Definition of SPI signals. I/O direction is defined from the perspective of the sub device.

SPI has 4 operating modes (which you can view here if you are curious), however in this lab, you will **only** implement SPI mode 0. SPI mode 0 means that data is transmitted from the main (i.e the test bench that you write) on the negative edge of `sclk` *or* negative edge of `cs_n`, and sampled (captured) by the sub on the positive edge of `sclk`.

For both transmitting and receiving, a data word is transmitted by sending one bit at a time, in the order of most significant bit (MSB) first to least significant bit (LSB) last. Figure 3 shows a sample timing diagram demonstrating sending the bits `0101` from main to sub.

The transaction is broken down as follows:

- Edge 3: sub device exits from reset *and* main transmits a 0
- Edge 4: sub captures the 0
- Edge 5: main transmits a 1

[1]You may find terminology other than main/sub used elsewhere, however these are the terms we will use in this class.

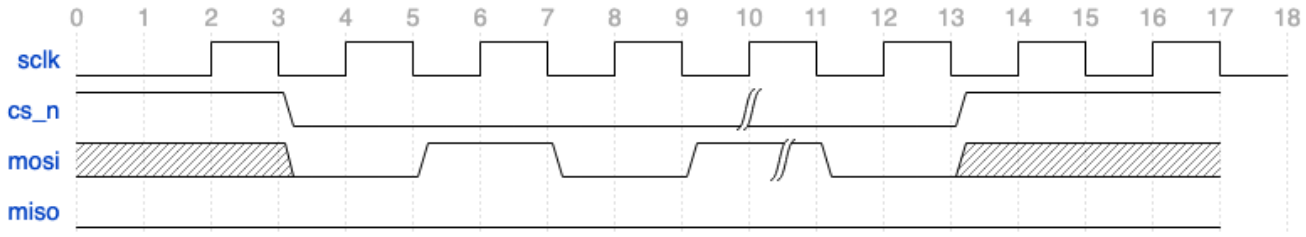

Figure 3: SPI example of transmitting the bits `0101` from main to sub.

- Edge 6: sub captures the 1
- Edge 7: main transmits a 0
- Edge 8: sub captures the 0
- Edge 9: main transmits a 1
- Edge 10: sub captures the 1
- Edge 13: sub resets when `cs_n` is asserted

Note that data is captured on the posedge of sclk when cs_n is low, and the SPI sub resets on negedge sclk when cs_n is high. For simplicity, your SPI sub implementation will differ from the common SPI standard in that when `cs_n` is `1`, `miso` should be `0`, not `Z` (high impedance).

4 SPI Message Specification

While SPI defines the physical protocol, it supports arbitrary messaging (e.g., how many bits are sent at a time, what the bits mean, and how the sub responds). For this lab, we will define our own message protocol. This protocol is designed specifically to read from and write to a 1 KB on-chip memory (external to your module) while also reserving space for future use.

4.1 Message Format

A message will be defined as 44 bits, composed of a 2-bit operation type, a 10-bit address, and 32-bit data. Op 0 will indicate a read operation and Op 1 will indicate a write operation. The remaining bit is reserved for future use. The address field will use 8 bits for fully addressing the 1 KB memory using 32-bit words. The upper 2 bits are reserved for future use. The data field is for transmitting 32-bit data words.

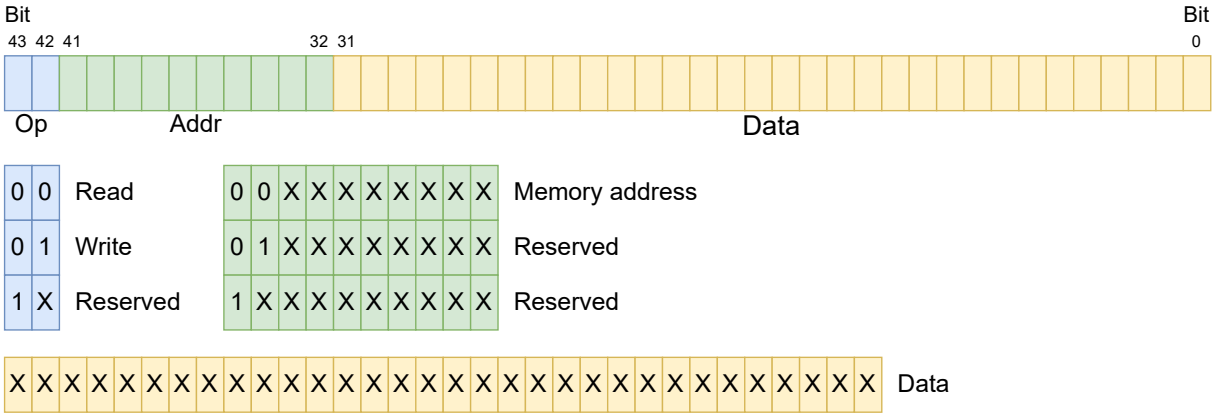

Figure 4: Message format (44 bits).

4.2 Response Format

The SPI Sub will **always respond to every message it receives from the Main**. After the last bit of the Main's message is received, the Sub begins transmitting its response on the `miso` line. The response from the SPI Sub also follows the 44-bit format:

- **Read operation**: Response includes the opcode, requested address, and the data read from memory.
- **Write operation**: Response echoes the original message (opcode, address, and written data).

4.3 Timing and Feedback Behavior

After the last bit of the 44-bit message on `mosi` is captured by the sub:

- On the **next posedge of** `sclk`: The sub asserts `w_en` or `r_en` high for one cycle to access memory.
- On the **following negedge of** `sclk`: The sub begins transmitting its 44-bit feedback on `miso`, starting with the MSB.

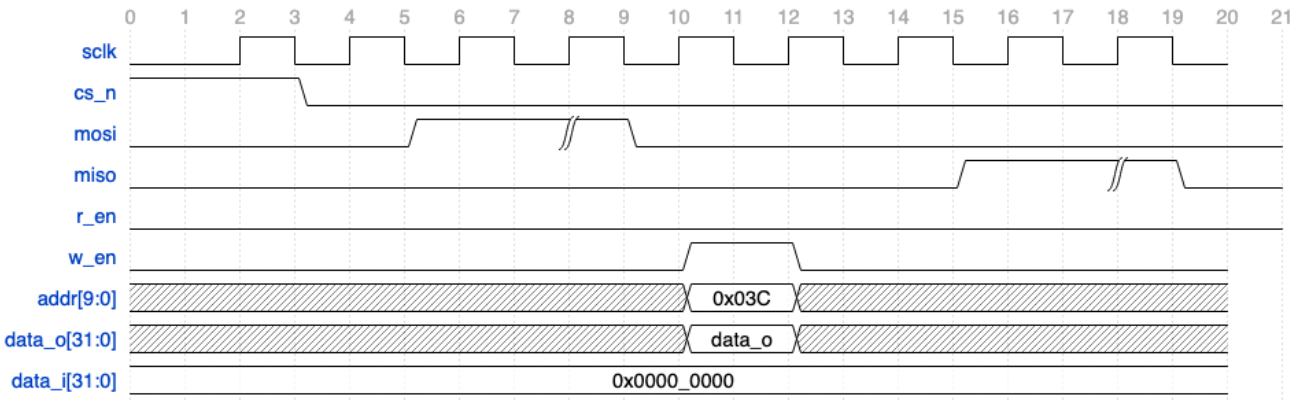

Figure 5: Feedback behavior for a write message.

Example cycle breakdown:

- Edge 7: Last bit of the message sent by main.
- Edge 8: Last bit sampled by sub.
- Edge 10: Sub asserts `w_en` or `r_en`.
- Edge 13: Sub begins transmitting feedback (MSB first).

To help you understand the complete behavior of both write and read transactions, we will also provide a `.vcd` waveform file that you can view to study the protocol timing in detail.

4.4 Reserved Behavior

If the SPI Sub receives a reserved opcode or reserved address, the behavior is **undefined**. Undefined behavior means anything may happen. Your module will not be tested for undefined behavior, and your testbenches should not explicitly test for it.

Q: Why so much duplication? Isn't that inefficient?
Yes, it is inefficient. However, this protocol is designed for **simplicity and debugging**, not for speed. For example:

- If a write response returns incorrect data, the issue is likely with the SPI interface.
- If the write response is correct but a subsequent read to the same address is wrong, the problem is likely in the memory.

4.5 SPI Transactions

While the SPI physical specification allows simultaneous transfers on `miso` and `mosi`, we will **not** allow this in the lab:

- During the Main's message: The Sub must hold `miso` = 0.
- During the Sub's response: The Main must hold `mosi` = 0.

5 Memory Specification

The memory interface has the signals shown in Table 2. You can assume that `data_i` will be ready the same cycle on which `r_en` is asserted.

Signal	I/O	Meaning	Definition
`r_en`	Out	Read enable	Enables reading from the external memory
`w_en`	Out	Write enable	Enables writing to the external memory
`addr`	Out	Address	Memory address of the external memory
`data_o`	Out	Data out	Data stored to the memory when `w_en` is active
`data_i`	In	Data in	Data read from the memory when `r_en` is active

Table 2: Definition of memory signals. I/O direction is defined from the perspective of the sub module.

6 Objective

- Your task is to design the **SPI submodule** and a **testbench file** that acts as the main module, according to the specifications in Sections 3 and 4.
- Use the following module interfaces:

  For Design (`spi_sub.sv`):

```
module spi_sub (
  // SPI signals
  input  logic sclk,
  input  logic cs_n,
  input  logic mosi,
  output logic miso,

  // Memory signals
  output logic        r_en,
  output logic        w_en,
  output logic [ 9:0] addr,
  output logic [31:0] data_o,
  input  logic [31:0] data_i
);
```

  For Testbench (`spi_tb_*.sv`):

```
module spi_tb (
  output  logic        sclk,
  output  logic        r_en,
  output  logic        w_en,
  output  logic [9:0]  addr,
  output  logic [31:0] data_o,
  input   logic [31:0] data_i
);
```

- We will provide you with the following ready-made components:
  1. **RAM module** (`ram.sv`) – Implements the 1 KB memory.
  2. **SPI top module** (`tb_top.sv`) – Connects your SPI submodule to the RAM module.
- You do **not** need to modify these provided files. Simply include them in your compilation command while testing your design.
- When compiling and running with **VCS**, include all required files:
  1. Your SPI submodule file
  2. Your testbench (main) file
  3. The provided RAM module file
  4. The provided SPI top module file

Example VCS Command

```
vcs -full64 -sverilog spi_sub.sv spi_tb_write.sv ram.sv tb_top.sv -debug_access+all
./simv
```

6.1 Submission Requirements

- Create one or more **black-box testbenches** to test your design.
  - Do not access any internal signals of the design under test (`spi_sub`).
  - You may use both the SPI signals and memory signals, just not internal variables of the `spi_sub` module.
- Submit the following files to the autograder:
  - `spi_sub.sv` (design)
  - `spi_tb_*.sv` (1 to 10 testbenches)
- The `*` indicates a wildcard pattern match, meaning it can be any valid character in a Unix filepath. Example filenames include:
  - `spi_tb_1.sv`
  - `spi_tb_2.sv`
  - `spi_tb_longMsg.sv`
- For instructions on how to submit files to the autograder, please see the submission guide on **BrightSpace**.

6.2 Testbench Output

Testbenches must report their status by printing either:

- `@@@PASS` if the design behavior is correct
- `@@@FAIL` if the design behavior is incorrect

These strings must be printed **verbatim** (case-sensitive, and including the `@` symbol). Each testbench should print **exactly one** status message. If a testbench prints multiple status strings, or both `@@@PASS` and `@@@FAIL`, it will be ignored by the autograder.

It is therefore recommended that you:

- Use `$display("@@@PASS")` only at the end of your testbench, after all checks pass, followed by `$finish()`.
- Call `$display("@@@FAIL")` immediately when a failure is detected, followed by `$finish()`.

You are free to display other strings for debugging (e.g., `"pass"`, `"TEST FAILED"`), but they will be ignored by the autograder.

7 Grading

Both your design **and** your testbench(es) will be graded by the autograder. Your design will be checked for correctness by running it against several hidden instructor test cases. You will receive points for each test that you pass. Your testbench(es) will be checked for coverage and thoroughness by running it/them against several hidden instructor buggy designs. You will receive points for each buggy design you catch, meaning at least 1 testbench correctly reports the design as failing. Test benches will be ignored if they report a correct design as buggy. Your final score is determined by the total score of your **best** submission at the time of the deadline.

This assignment has a firm deadline (check BrightSpace). No late submissions will be accepted.