RTL SPI Modules User Guide

Kyle Infantino and Dilan Lakhani
https://github.com/pymtl/pymtl3-spi

This guide provides information on how to use the modules found in the pymtl/pymtl3-spi GitHub repository. These pre-assembled RTL modules allow for communication to a chip over SPI while abstracting away the intricacies of the SPI protocol. Instead, the SPI communication can be controlled through the use of the familiar val/rdy interface.

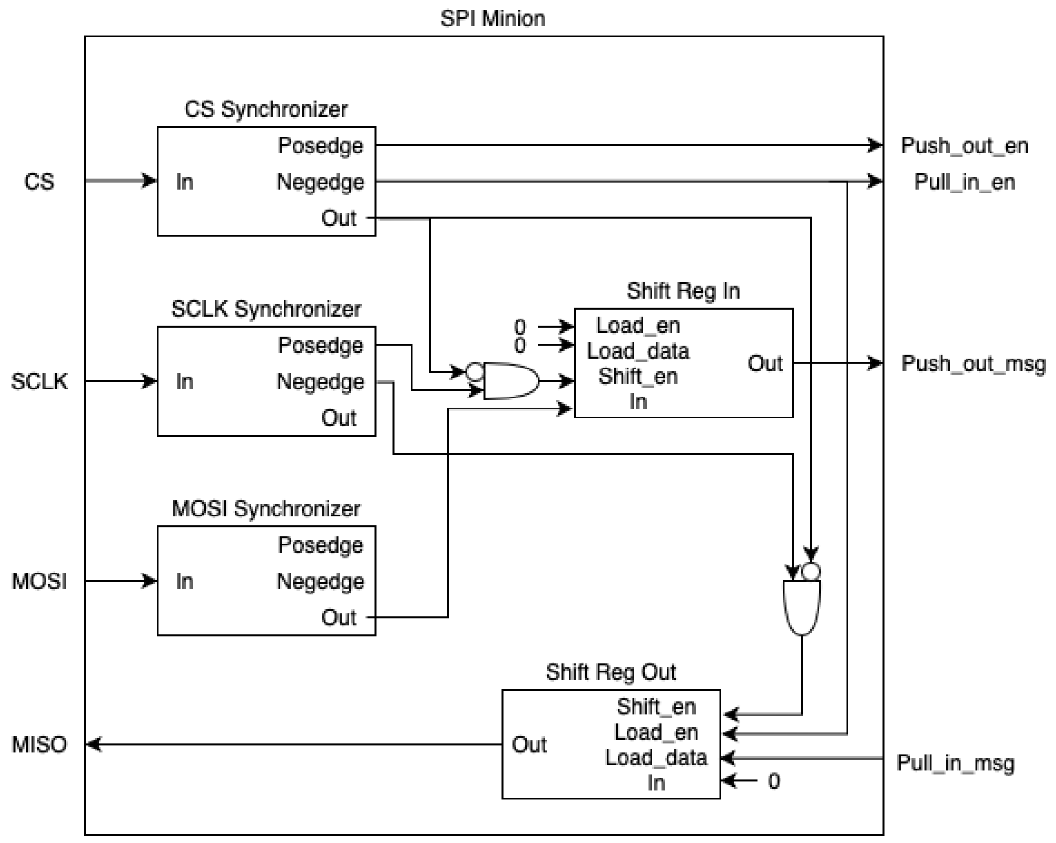
The main motivation for these designs is to communicate with a chip when limited pins are available. SPI allows for full duplex communication between a master and minion while only requiring 4 pins, making it a logical protocol to use when taping out a chip.

A few restrictions to note: To ensure synchronization between the SCLK and the on-chip clock period, SCLK must have a period at least 6 times longer than that of the chip. It should also be noted that SPI is inherently a push/pull protocol. This means that the SPI master completely controls when data is sent and received from the minion. Although this detail is largely abstracted away through the use of the SPI Minion Adapter and SPI Test Harness, it is still a fact to keep in mind.

All modules are provided with both PyMTL and SystemVerilog implementations.

Table of Contents

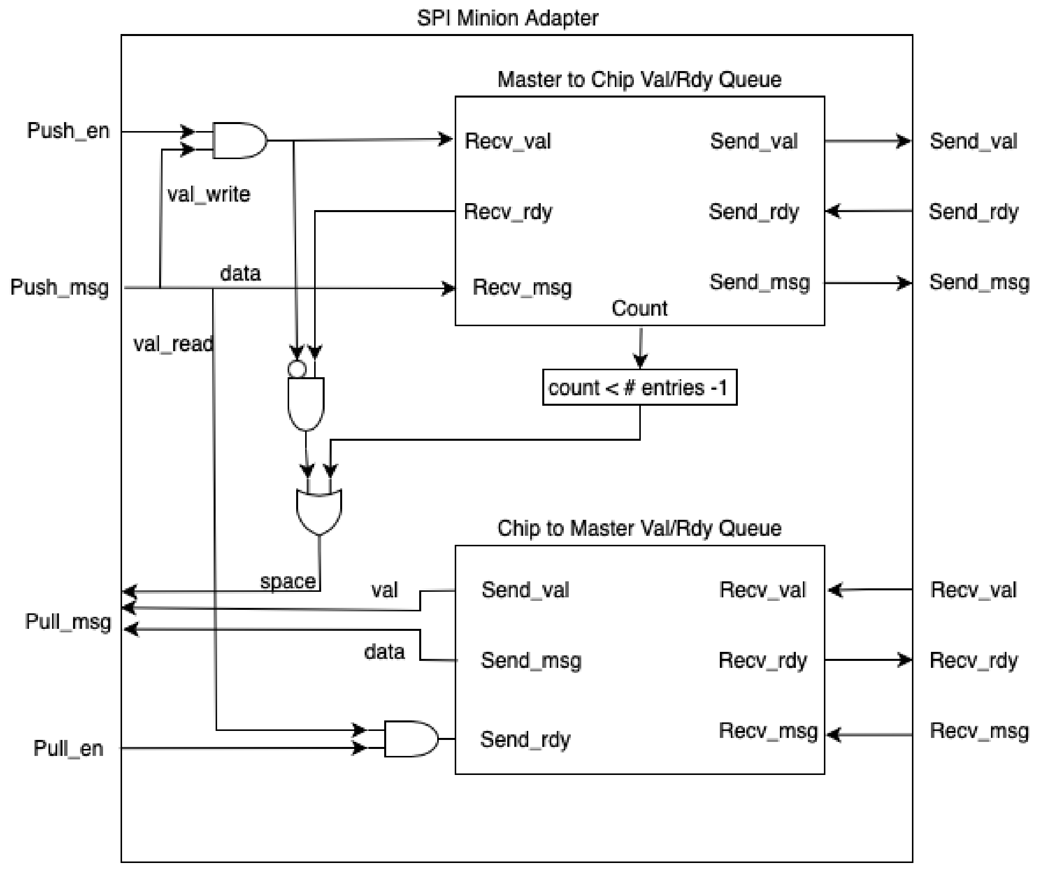
SPI Minion



The SPI Minion sends and receives messages over SPI and transfers them using a Push/Pull interface. The module takes in an “nbits” parameter which represents the size of the shift registers and SPI packets. This parameter can take any value, but the physical SPI Driver can only send messages that are a whole number of bytes. The push and pull messages will also be “nbits” long.

In addition to the two interfaces, the minion also contains a parity bit. This parity bit is the result of an XOR operation on all but the two most significant bits of the push message, followed by an AND with the push enable signal. The reason for not including the two most significant bits is that the minion is meant to be used with the SPI Minion Adapter, in which case the two most significant bits of the message would be control bits and not part of the actual data being communicated. The AND operation ensures that the parity will only output a non-zero value when the message is “valid.”

<u>SPI Minion Adapter</u>



Push_msg

Val_write	Val_read	Data

Pull_msg

Val	Space	Data

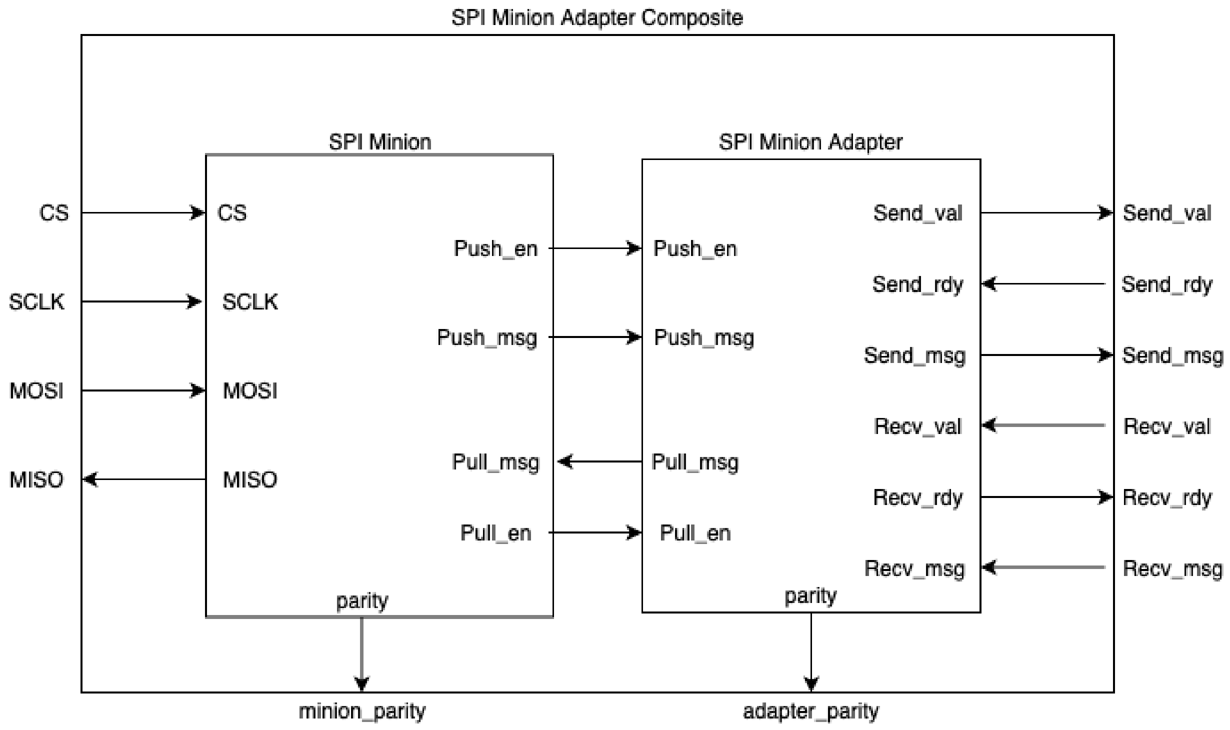
The SPI Minion Adapter is designed to be used along with the SPI Minion to convert the inherent Push/Pull behavior of the SPI protocol to a more flow-controlled val/rdy interface. The module takes as parameters “nbits” and “num_entries”. The “nbits” parameter signifies the number of bits in the pull and push messages. These messages contain the two control bits which are then stripped off, meaning the send and receive messages will have length nbits-2. The “num_entries” parameter indicates the size of each of the queues in the adapter. This value determines how many messages to be sent or received can be held in the adapter at a time.

To implement flow control over SPI, two control bits are added to the beginning of each push and pull message as seen above. Since communication is completely controlled by the Master in the SPI protocol, these flow control bits are used to indicate to the Master if the Minion is ready to receive more data or if it has data to send. Since SPI is full duplex, meaning on every transmission data is sent both ways, the data flowing in either direction may not be valid all the time, motivating the need for valid bits. The most significant bit of the push message is the val_write bit. This bit indicates that the SPI Master has a valid message to send to the SPI Minion. The second most significant bit is the val_read bit. This bit indicates that the SPI Master is ready to receive the next message from the SPI Minion. The most significant bit of the pull message is a val bit that is used to indicate if the message being sent from the SPI Minion to the

Master is valid. The second most significant bit "space" is used to indicate if the SPI Minion Adapter has space in its queue to receive another message from the Master.

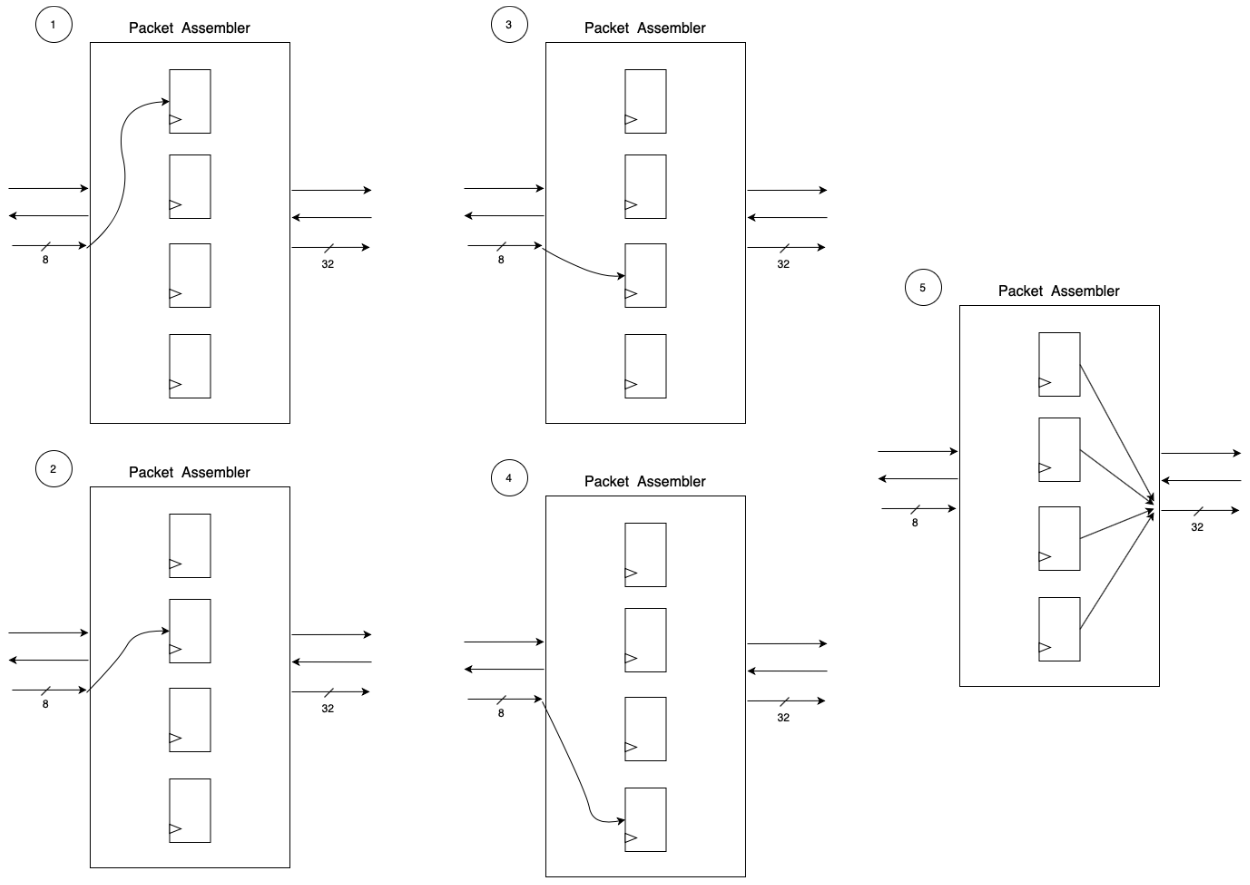
The SPI Minion Adapter also includes a parity bit which consists of the bits of the send message XORed together and ANDed with the send_val bit. This ensures the parity bit can only display a nonzero value when the message is valid.

SPI Minion Adapter Composite



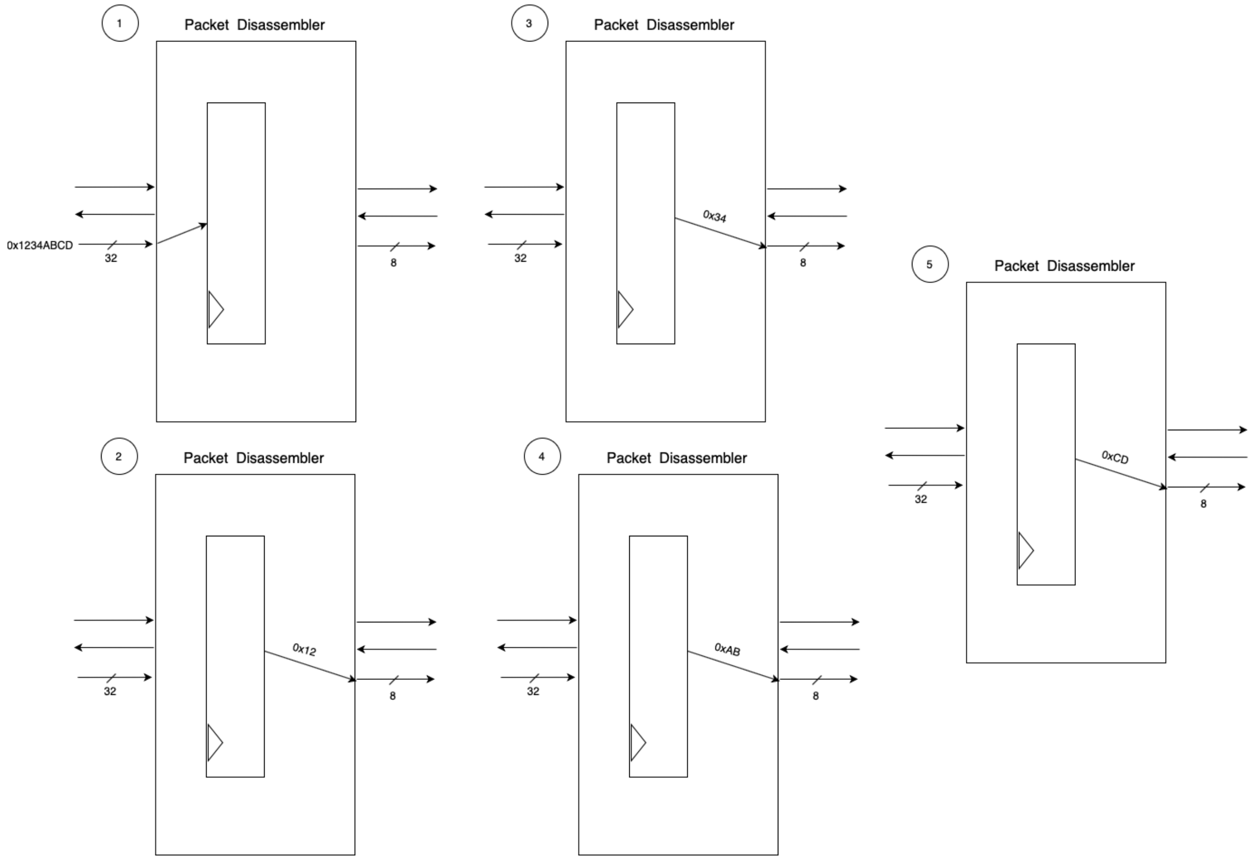
The SPI Minion Adapter Composite simply instantiates the SPI Minion and SPI Minion Adapter and connects them. The resulting module contains an SPI interface, Val/rdy interface, and the two parity bits from the minion and minion adapter. This should be the primary module used for interfacing with SPI through val/rdy.

Packet Assembler



The Packet Assembler is useful when moving from a narrow bit domain to a wider bit domain. For example, moving from a domain where all packets are 8 bits wide to a domain where packets are 32 bits wide. In this case, we need to assemble four packets from the first domain, then send the concatenated 32-bit packet to the second domain. The Packet Assembler is instantiated with a certain number of internal registers that it uses to accept smaller input packets. If we follow the example from above, we would need 4 internal registers: one for each of the 8-bit packets. Once the Packet Assembler has received 4 valid packets, it will output a valid 32-bit packet. The Packet Assembler is parameterized by input bit width and output bit width, so it can be applied to any domain-crossing situation as long as the input bit width is smaller than the output bit width.

Packet Disassembler



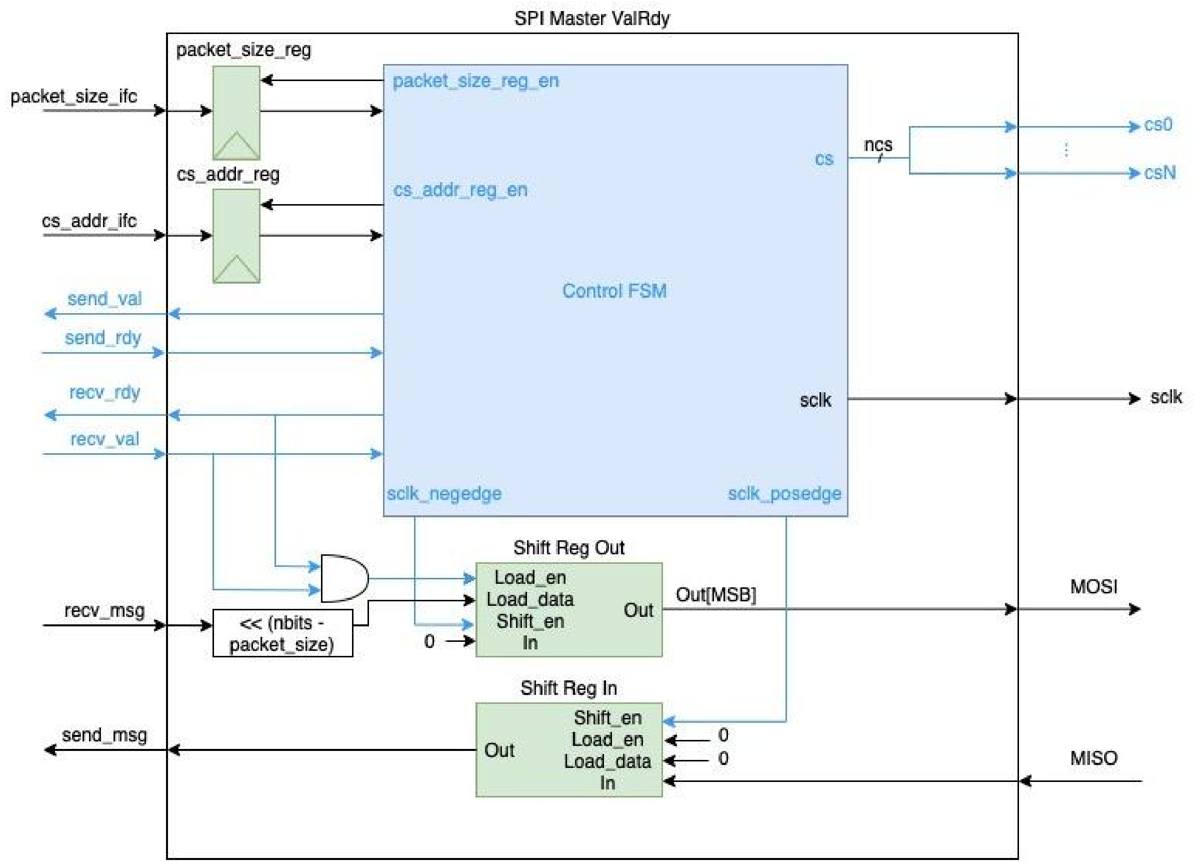
The Packet Disassembler is the opposite of the Assembler. We use it to move from a wide bit domain to a narrow bit domain. For example, moving from a domain where all packets are 32 bits wide to a domain where packets are 8 bits wide. In this case, we need to disassemble the 32 bits into 4 separate 8-bit packets. A valid transaction is started when the Disassembler receives a valid input packet. It stores the input packet in a single internal register. In the next cycle, the Disassembler outputs a valid packet that consists of the most-significant bits of the input packet. If we use the 32-bit input and 8-bit output example and the input packet was 0x1234ABCD, then the first output packet would be 0x12. In the next cycle, the Disassembler would output 0x34, followed by 0xAB, and finally 0xCD. Then, the transaction is over and in the next cycle the Disassembler is ready to accept a new input packet. Like the Assembler, the Disassembler is parameterized by input bit width and output bit width, provided that the input is wider than the output.

Packet SerDes

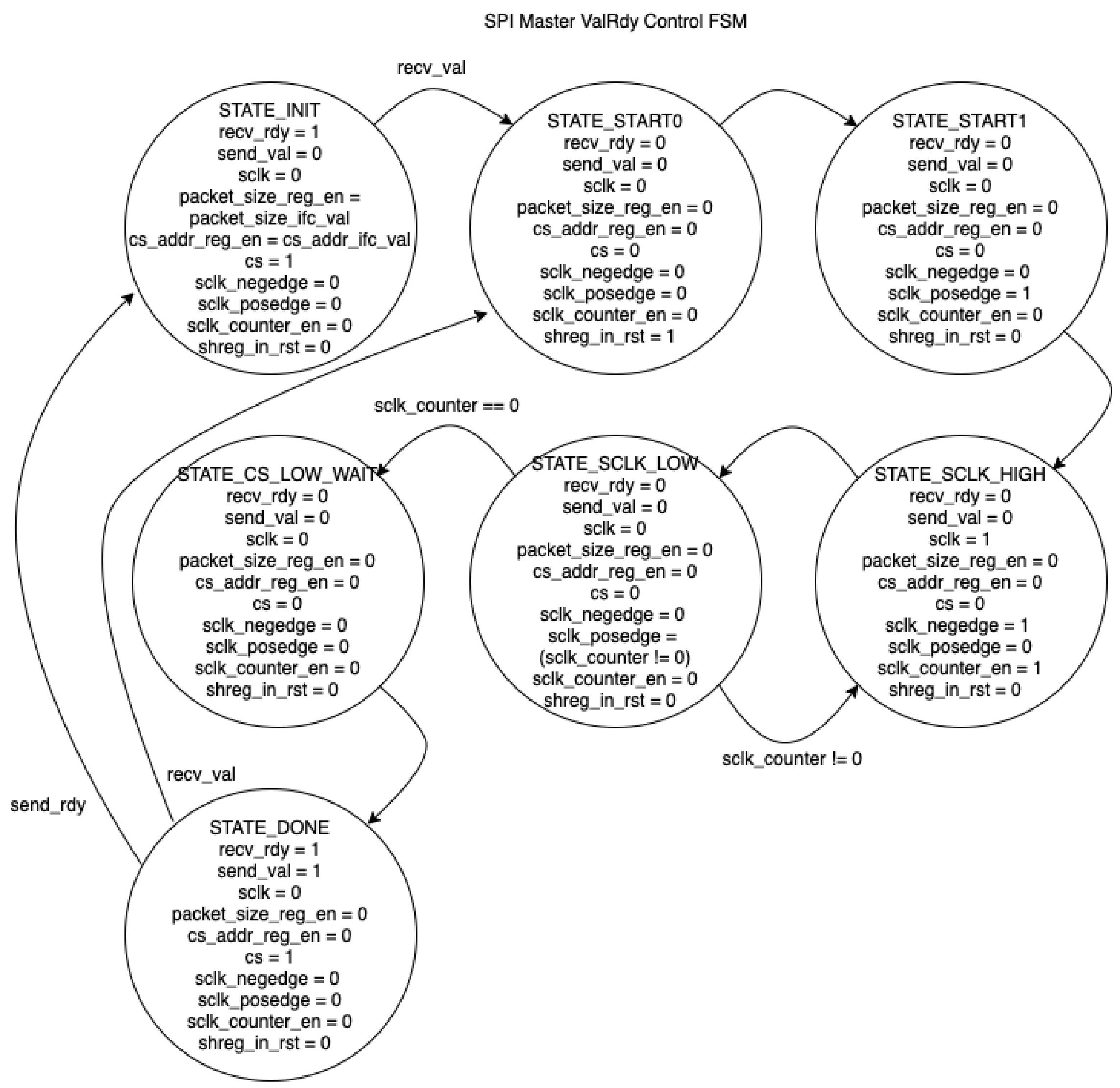
The Packet SerDes module is an easier-to-use alternative to the Packet Assembler and Packet Disassembler. This module takes “nbits_in” and “nbits_out” parameters to indicate the number of input bits and number of output bits. Based on these two parameters, the SerDes component will instantiate the correct component of either Assembler or Disassembler.

<u>SPI Master Val/Rdy</u>

Like the SPI Minion, the SPI Master abstracts away the intricacies of the SPI protocol and messages to be sent and received over SPI through a val/rdy interface. The SPI protocol supports one master and multiple minions where all minions share the same SCLK, MOSI, and MISO lines. As a result, additional minions can be added at the cost of one additional CS port per minion. Since different SPI minions may have different SPI packet sizes, this master also supports sending and receiving variable SPI packet sizes up to a parameterized number of bits. To configure which minion should be addressed, as well as the SPI packet size, the packet_size and cs_addr registers can be set. These registers can be written to any time an SPI transaction is not currently in progress.



The datapath for the master largely consists of shift registers used to serialize/deserialize the data. Since the SPI SCLK is controlled by this component, there is no need for a synchronizer on the input MISO signal. When data is sent to the SPI Master to be sent to a minion, the data is shifted by 32 minus the SPI packet size. This means that if the SPI packet size is 1, the data will be shifted such that the least significant bit in the 32 bit register becomes the most significant bit. The reason for this is that when the data is serialized the most significant bit is sent first. Since the registers will shift on every SCLK edge while any CS is low, the number of bits sent and received can be configured by simply changing the number of times SCLK is toggled while CS is low. To read from a minion device, a valid request must first be sent to the minion to initiate a transaction. The returned message can then be read from the master once the transaction is complete.



The FSM for the SPI Master consists of seven states. STATE_INIT serves as the default state for the SPI Master when there is no transaction in progress. In this state, both CS are low and the packet size and CS address can be set by the processor. Once the master receives a valid request from the processor, a transaction begins. The START0 state lowers the chip select line to start the transaction and resets the MISO shift register. The START1 state raises the SCLK for the first time before moving into data transmission. The SCLK HIGH and LOW states are used to toggle the SCLK while reading and writing data to/from the MOSI and MISO lines. The CS_LOW_WAIT state keeps the CS line low for one cycle after the last data is read. The DONE state raises CS to complete the transaction. If the master receives a new message to send, the FSM moves back to the START0 state. If the data is read while in the DONE state, the FSM moves back to the INIT state.

LoopBack

The Loopback module simply reads in a message of “nbits” and outputs it the next cycle. This is done using val/rdy for both interfaces. This component is mostly used for building simple test components or as a design-for-test component.

LoopThrough

The LoopThrough module is another simple component mainly used in test components or as a design-for-test component. The LoopThrough component is similar to the LoopBack component with the addition of a pass through capability. If the 1 bit select input is set to 1, the LoopThrough behaves in the same manner as the LoopBack. If the select input is set to 0, this component is effectively transparent, reading a message in one val/rdy interface and outputting it on another.

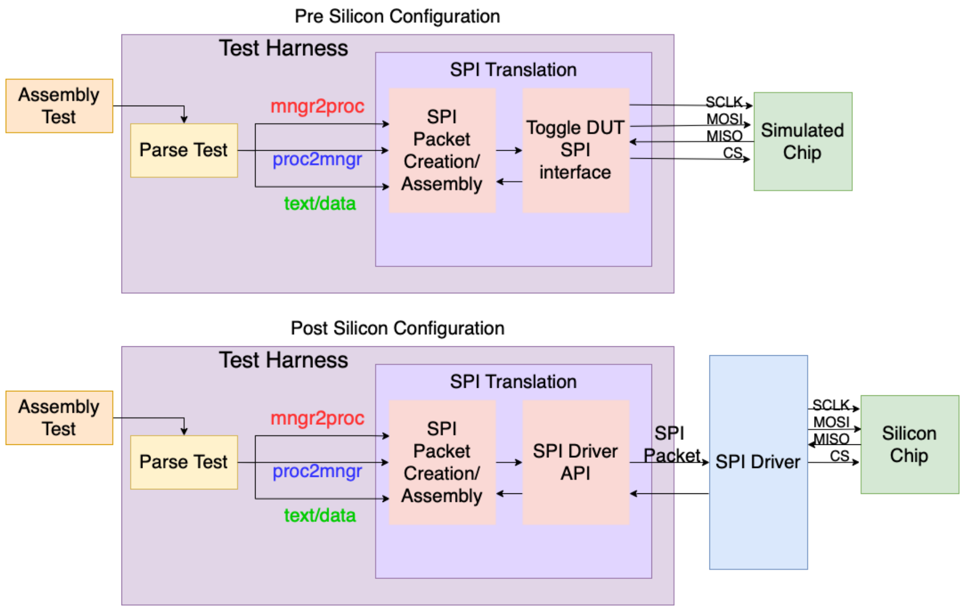
SPI Test Harness

The purpose of the SPI Test Harness is to simplify the process of testing over an SPI interface by completely abstracting away the SPI protocol and flow control used by the SPI Minion Adapter. This allows existing tests to be sent to a chip via SPI with minimal modifications to the tests. As parameters, the harness takes an instantiated design to be tested, the number of different components on the chip able to receive SPI messages, and the number of bits in an SPI packet (this number includes the two flow control bits, any optional component addressing, and the data). The number of components is used to determine how many bits of the SPI packet should be reserved for addressing the different components on the chip. Once on the chip, this address can be used by the router to determine which component to send the message to. For many designs, there will only be one component.

To run a test, the “t_mult_msg” function is called. It takes as parameters “req_len” and “resp_len”, which represent the number of data bits in each request or response message (this value does not include any control or address bits). It also takes a “request_list” and “expected_resp_list”. The request list is a list of Bit objects to be sent to the chip, and the expected response list is a list that will be checked against the actual output of the chip. In the case where a test only wants to write or read a message to/from the chip, the response/request list can be empty and the resp/req_len is a don’t care value. The optional component address is the address of the component this packet should be sent to. If the number of components is 1, the component address is a don’t care value. The “return_msgs” option indicates whether or not the chip responses should be checked or returned. False indicates that the chip output should be checked against the expected response list. This is the default option that allows for automated testing. True indicates that the output will not be checked and will simply be returned. This option should be used when the exact output is uncertain and should be checked manually.

Once the “t_mult_msg” function is called, it will run until it has received all of its expected responses. The function handles all the toggling of all the SPI lines as well as the flow control used with the SPI Minion Adapter. Once all the responses are received, the function will either compare the responses to the expected response list or return them according to the “return_msgs” flag. If the harness never receives the expected number of results, the program will hang until it times out after 1000 consecutive invalid messages from the Minion. If this happens, double check the arguments in the harness instantiation and t_mult_msg function call. Otherwise, it is likely a problem with the RTL. When trying to diagnose issues, remember that the SPI protocol takes 6 chip clock cycles to send a single bit.

To facilitate post-silicon testing, the SPI Test Harness contains a flag “is_phy_test”. By default, this flag is set to False, meaning the tests are run on the simulated model. By flipping this flag to True, the harness will make calls to the SPI Driver library (explained below) instead of toggling the bits in simulation. This allows complete test reusability between pre- and post-silicon testing with just the change of a flag. A visualization of the different testing configurations can be seen below. Note that in this figure the SPI Test Harness is labeled as “SPI Translation”. Examples of how to use the SPI Test Harness module can be found in the “SPI_v3/test/SPITestHarness_test.py” file in this repository.



SPI Driver

As referenced above, the SPI Driver is an independent hardware component that can be used in the post-silicon testing of any design containing an SPI minion. On the host computer side, this device is interfaced with through the use of the included SPI Driver API. The driver translates the received data to SPI and sends it out over the 4 SPI lines. This driver also includes two additional lines A and B that can be controlled through the API. These signals can be used for any signal, although it can be helpful to designate one of these to be the “reset” signal to the design. It is important to note that while none of the 3.3 V or 5 V lines on the driver need be connected to the design, both GND pins must be connected to the design ground for correct functionality. The SPI Driver can be found at https://spidriver.com.