What's "vSPI"?

vSPI is a Verilog implementation of an SPI slave. Think of it as a very fast serial port. It can reliably transfer data at 27.9 mbps on an Atlys FPGA devkit (a Spartan-6 with a 100 MHz system clock).

You can use it to send data between your FPGA/ASIC project and other devices, such as a desktop computer I'm using it to send data between a self-flying RC-helicopter and my PC. If all goes according to plan, I'll be able to see live video from the helicopter's camera on my PC. I'll also be able to inject test data and make sure my logic works with known test vectors.

What's included?

So far, vSPI consists of three parts:

- **spiifc**: The minimal logic that implements the SPI slave. spiifc takes the usual four SPI lines (MOSI, MISO, SS, SCLK) and has interfaces for input and output buffer memories as well as a register bank (more on what they do is below). If you work directly with spiifc, you'll need to figure out how to get this stuff to interface with the rest of your project. This is a good place to start if you have a project without a system bus or a non-PLB bus.

- **PLB interface**: vSPI includes also includes a PLB interface if desired. PLB is one of the system bus protocols supported by the Microblaze processor provided in Xilinx's EDK (sometimes known as XPS). I may also add support for one of ARM's AMBA bus protocols later (AXI, etc.), but there is zero support for thst right now.

- **spilib python library**: spilib is a python library that is used on your PC to make talking with spiifc easier. It is currently built on TotalPhase's

Cheetah SPI USB/SPI adapter API. It makes interactions between a PC (master) and spiifc (slave) simple.

Components

spiifc

spiifc uses two 4KB buffers to send data to the master (master in, slave out - MISO) and to receive data from the master (MOSI). Spiifc provides interfaces to these buffer SRAMs but you must provide the SRAMs. When spiifc receives data from the master, it writes each byte in succession to the MOSI buffer. When the transfer is complete, you'll be able to access the transfer by reading the SRAM. Sending data is similar: write your transmission to the MISO buffer. The master will then be able to access the data on it's next read.

You'll want to use a dual port memory so that spiifc has one port and you can access the other. Note that spiifc only needs a read port for the MOSI memory and a write port for the MISO memory, so you can get by with a read port and a write port rather than true dual ported memories.

spiifc also provides access to a register bank of 16 registers. This is useful for synchronizing control between the slave and master. For example, if the slave wants to indicate that new data is available in the MISO buffer, it could change a register value. The master could poll this register periodically to see if it should perform another read. Again, spiifc provides an interface for the regfile, but you must provide it yourself.

The spiifc.v verilog in /src/spi_base

The spiifc testbenches are in /test/spi_base

Protocol

spiifc uses a really simple protocol. All data is transmitted such that the first bit is the most significant, and the last bit is the least significant

The first byte is the command. The remaining bytes of the packet depend on the command. Here are descriptions of the commands from the master's point of view:

- **Read Start (0x01)**: Sends the data stored in the MISO buffer to the master, starting at the byte (address 0) in the MISO buffer. SPI sends a bit back to the master for every bit it sends to the slave, so just send as many garbage bytes as you want to read out from the MISO buffer.

- **Write Start (0x03)**: Writes data into the MOSI buffer, starting at address zero. Just write as many bytes as you want after the command, and they will be recorded in the buffer.

- **Read Register (0x80 to 0x8F)**: Reads the value stored in one of the sixteen registers. Write four garbage bytes after the command to get the 32-bit register value. The first byte is most significant, the fourth is the least significant. The least significant four bits indicate which register to read from.

- **Write register (0xC0 to 0xCF)**: Writes a value to one of the sixteen registers. The four bytes after the command is the value to write. The most significant byte is first, the least significant is last. The least significant four bits indicate the register to write to.

Of course, if you use spilib, all of this protocol stuff is taken for you. Just call the provided functions for each of the commands.

EDK peripheral

The EDK peripheral takes care of a lot of the setup if you're using a Xilinx Microblaze system with a PLB system bus. RAMBs are included for the MISO and MOSI buffers and a register file is provided for the spiifc registers.

To access the registers, simply access the peripheral's main memory mapped region. For example, if the region is mapped to 0x85000000, you could do the following:

```
 u32 * pSpiifcBase = (u32 *)0x85000000;
 pSpiifcBase[0] = 0xFEEDFACE;             // write 0xFEEDFACE to spi
register 0
 xil_printf("0x%08x\n", pSpiifcBase[15]); // write the value in spi
register 15
```

The MOSI buffer (Memory 0) and MISO buffer (Memory 1) have their own memory mapped regions and are directly accessible as well.

Note that if the spi master and Microblaze both try to write to the same register at the same cycle, the spi master will silently win and the Microblaze's write will be thrown out.

There is no interrupt support right now, but it might be added later.

- The PLB peripheral is stored in /projnav/xps/pcores/spiifc*v1*00_a
- The peripheral ISE project is in /projnav/xps/pcores/spiifc*v1*00_a/projnav (includes ISE configurations for all deployable tests and behavioral testbenches)
- An example EDK project is in /projnav/xps
- An example XSDK workspace is /projnav/xps/SDK/SDK_Workspace

spilib

spilib is a python library for controlling a Cheetah USB/SPI adapter to communicate with vSPI. Functions for all spiifc commands are supported.

spilib is located in /scripts/master/spilib.py. Check out spitest.py for examples of how to use it.

Examples

Several examples exist to show you how to use vSPI:

Behavioral testbenches:

- spiifc*tb*rc: writes a test pattern to the MOSI buffer (receive from the slave's perspective)
- spiifc*tb*tx: reads from the MISO buffer (transmit from the dlave's perspective)
- spiifc*writereg*tb: writes to a register, then reads it back

FPGA-ready spiifc demo (for Atlys Spartan-6 FPGA devkit)

- spiwrap: displays the least significant byte of the latest 4 bytes written to the MOSI buffer on the Atlys devkit's 8 LEDs.

- spiloop: uses one buffer for both MISO and MOSI buffer interfaces. Anything written to MOSI is immediately readable in the MISO buffer. Use with the MemLoopback test in spitest.py to continuously send 4KB packets and verify you get the same ones back.

EDK (XPS/XSDK) demo (for Atlys Spartan-6 FPGA devkit):

Use in combination with the XpsLoopback spitest.py test.

The master PC and and slave FPGA Microblaze PLB system work together in the following steps:

- The PC writes 4 KB of random data to the slave's vSPI MOSI buffer.
- The PC writes 1 to the least significant bit (bit 0) of vSPI's register 0
- The FPGA waits until Reg 0/bit 0 is 1 before proceeding (so it knows when the master is done writing to the MOSI buffer).
- The FPGA changes Reg 0 / bit 0 back to 0
- The FPGA uses Xilinx's DMA controller to copy the 4 KB data from the

MOSI buffer to the MISO buffer.
- The FPGA changes Reg 0 / bit 1 to 1 to indicate new data is ready in the MISO buffer.
- The PC waits for Reg 0 / bit 1 to be 1
- The PC reads the value of the MISO buffer
- The PC verifies the random data matches the data just received
- Repeat!

Note that the EDK demo can only operate vSPI at 22 mbps. This is a result of the 66.67 MHz bus clock.

Verified hardware

vSPI has been tested on an Atlys Spartan-6 LX45-2 FPGA interacting with a Cheetah USB/SPI adapter. Max speed is 27.9 mbps with a 100 MHz system clock. Please let me know if you can or cannot get it to work on other hardware.

vSPI has also been tested on a custom-designed platform for remote-controlled helicopters based on low power Spartan-6L LX150. The MISO and MOSI buffers in the PLB peripheral needed to be regenerated due to the slight technology change switching to low power, but nothing else. vSPI has been used to send and receive test vectors, and remotely display the helicopter's video camera live. Video demo: http://flic.kr/p/bEZpAK.

License

If you use vSPI, whether for free or commercial purposes, I only ask that you let me know so that I can publicly keep track of who is using it. I don't care if you use it as is or modify it, so long as it isn't used in technologies to physically hurt or kill anyone (missile guidance systems, etc.).

If you want to use vSPI for any reason but wish to do so without publicly stating so, we can work out an alternative licensing agreement.

The vSPI project retains all ownership of code published here. Meaning, don't take the code, claim ownership, and then somehow sue the vSPI project.

You can reach me at buzz.vspi@clearhive.com.