Lab 5 Hints: Overview of SPI Interface Component (accel_spi_rw)

ECE 525.442 FPGA Design using VHDL

This document provides some hints to design the interface to the ADXL362 accelerometer using the Artix7 FPGA on the Nexys-4 DDR. The hints provided discuss the interface to the accelerometer in order to power up the device and read the device IDs and XYZ accelerometer data. The VGA driver and movement of the red square using the ADXL362 are not explained in this document.

The *accel_spi_rw* VHDL component, discussed in the Lab 5 description, consists of several parts which can be implemented as lower-level components or as processes, depending on your design preference. A block level diagram for the *accel_spi_rw* VHDL component is shown in Figure 1. Please note that each block represents either a process or a lower-level component. It is left to you how you want to design the *accel_spi_rw* component.

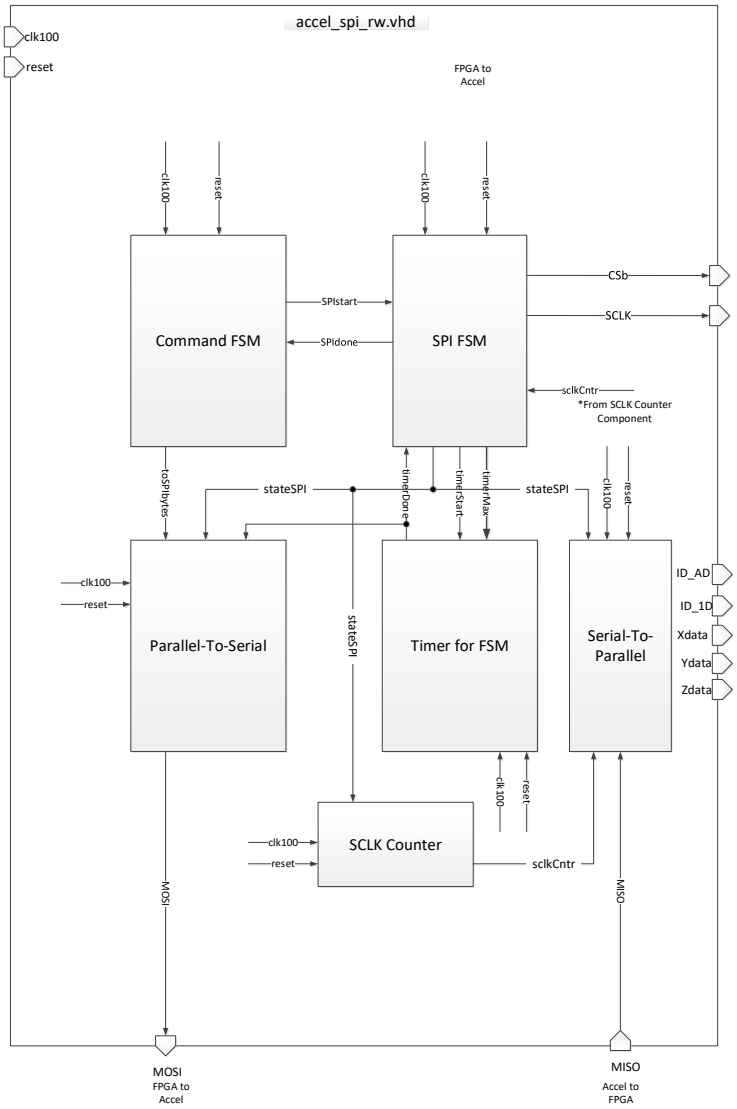

Figure 1: Block diagram for the *accel_spi_rw* VHDL component

The main part of the *accel_spi_rw* component is the "Command FSM" which is a state machine that sets the address and data registers to the SPI device in a specific sequence. Figure 2 shows the finite state machine for the "Command FSM". Note that this may not be the most efficient way to implement the process as you will notice that many of the states are similar and the entire FSM can be implemented with fewer states. This state machine is a straight forward implementation that works and adheres to good coding styles.

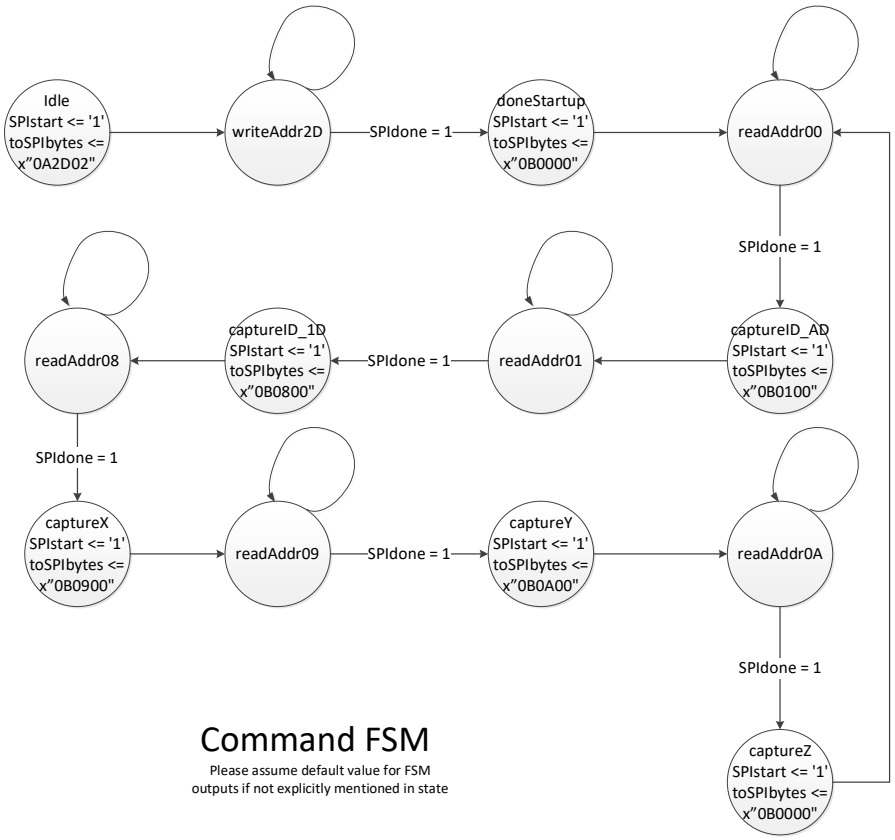

Figure 2: State transition diagram for "Command FSM"

The first command that is issued to the ADXL362 is the hex command 0x0A2D02 which prepares the ADXL362 device for a write command (0x0A), followed by the register address (0x2D), followed by the data value (0x02) which, according to the data sheet, powers up the ADXL362. This command is set in the "idle" state using a 24-bit register called "toSPIbytes". Once these three bytes are set in the "idle" state, a pulse called "SPIstart" is also set high in the "idle" state which is used in a secondary state machine called "SPI FSM". This FSM will be discussed later. The "Command FSM" progress into a *wait* state called "writeAddr2D" and remains in that state till the secondary FSM issues an "SPIdone" pulse. You can think of the two pulses "SPIstart" and "SPIdone" as handshaking signals between the two FSMs. Once the secondary "SPI FSM" is done writing to the ADXL362, the "Command FSM" transitions from "writeAddr2D" to the next state which is "doneStartup". During this state, the three-byte (or 24-bit) command x"0B0000" is set and stored in register "toSPIbytes" where the byte 0x0B prepares the ADXL362 for a read command, followed by the register address (0x00), followed by all zeros since this is a read command. According to the ADXL362 datasheet, this is a read command of the device ID at register 0x00. That should return a constant ID value 0xAD from the ADXL362 device. Additionally, the "SPIstart" pulse is set in the "doneStartup" state and the "Command FSM" transitions to the "readAddr00" state. The "SPIstart" pulse is used once again in the secondary state machine "SPI FSM" to toggle the SPI signals with specific timing in order to read the ID from the ADXL362 device. The "Command FSM" remains in the readAddr00" state until the secondary "SPI FSM" completes reading the ID from the ADXL362,

at which point the "SPI FSM" sets the SPIdone signal and the "Command FSM" transitions to the next state called "captureID_AD" which captures the 8-bit data that is read back from the ADXL362. The task of reading the IDs and XYZ values from the ADXL362 continues in a similar manner until the "Command FSM" reaches the final state "captureZ" which captures the 8-bit Z value of the ADXL362, after which the whole process of reading the IDs and XYZ values is repeated indefinitely.

The next FSM to discuss is the "SPI FSM" shown in Figure 3. This FSM is primarily responsible for generating the SPI signals that go from the FPGA to the ADXL362, namely the SCLK and CSb.

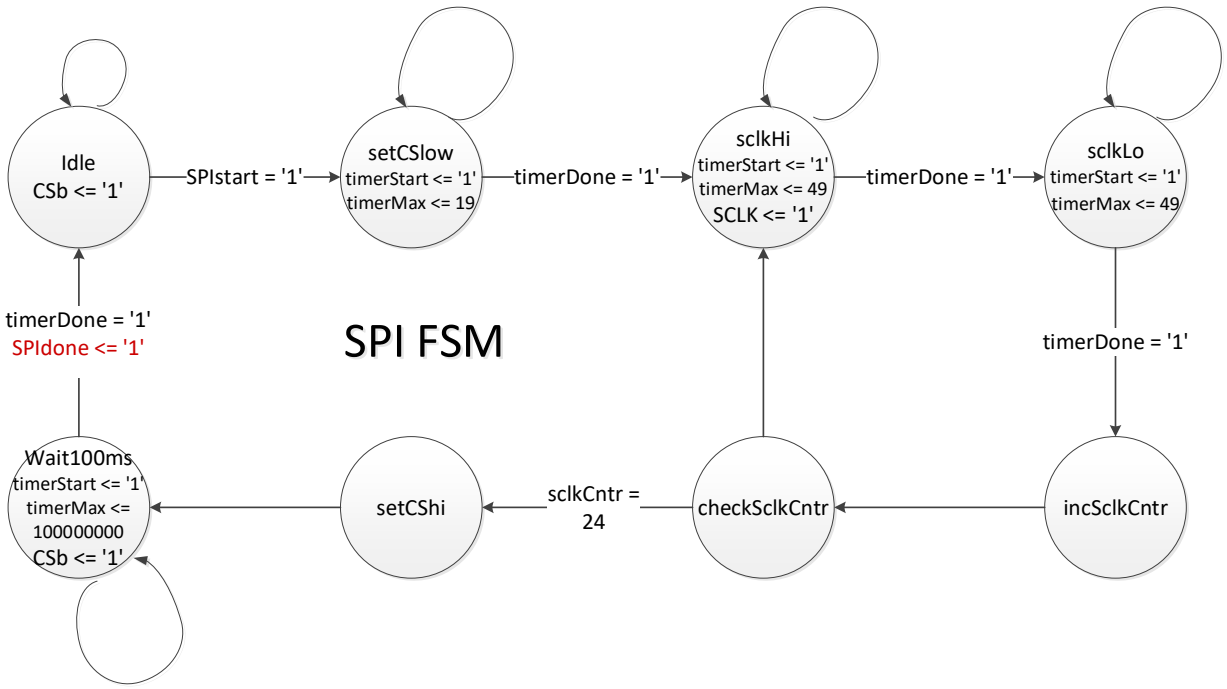

Figure 3: "SPI FSM" for generating the SPI signals to the ADXL363

This FSM starts in the "idle" state and remains there until the "Command FSM" issues a start pulse by setting "SPIstart" to high. The "SPI FSM" transitions to the next state called "setCSlow" that sets the ADXL362 chip select (CS) low for 20 clock cycles. Note that a timer process is used in conjunction with the "SPI FSM" to remain in the "setCSlow" for 20 clock cycles of the 100MHz clock. If you do the calculation, "SPI FSM" should remain in the "setCSlow" for 200 ns before SCLK starts toggling, and according to the datasheet, that corresponds to Css in table 10. Notice that Css is required to be a minimum of 100 ns, so I chose Css in my design to be 200 ns to allow for plenty of margin. While in the "setCSlow" state, a timer signal called "timerStart" is set high and remains high will in that state.

Additionally, a "timerMax" signal is set to the number of clock cycles you want to remain in that state. The "timerStart" and "timerMax" are both used in a process called "Timer for FSM". This process has been provided for you in lecture 7G and behaves like a timer that starts

counting once "timerStart" goes high and once the internal counter reaches "timerMax", the timer process generates a "timerDone" pulse which is used by the "SPI FSM" to transition to the next state; "sclkHi" in this case. The "SPI FSM", when in the "sclkHi" state, sets the SCLK high for 50 clock cycles (of the 100 MHz clock) using the timer process and, when in the "sclkLo" state, sets the SCLK low for another 50 clock cycles (of the 100 MHz clock). Using the two states "sclkHi" and "sclkLo", we can generate an SCLK period of 100 clock cycles (of the 100 MHz clock) or an SCLk period of 1 us, and according to the datatsheet in table 10, we meet the minimum timing requirements with plenty of margin. In this design, we are actually running the SCLK frequency at 1MHz and that is fine as long as we don't exceed 8MHz for SCLK. Notice that every time we complete one period of the SCLK, that is every time we transition out of state "sclkLo", we increment the SCLK counter and then check the value of the SCLK counter. This is done in states "incSclkCntr" and "checkSclkCntr" respectively. These two states are needed to ensure that we toggle the SCLK for only 24 periods, which is the communication protocol for the ADXL362 SPI device. You can create an additional external process to the "SPI FSM" that increments a counter called "sclkCntr" and when this counter reaches 24, the "SPI FSM" transitions from "checkSclkCntr" state to "setCShi". Assuming that "sclkCntr" has not reached 24 periods of SCLK yet, then the "SPI FSM" keeps on cycling through states "sclkHi", "sclkLo", "incSclkCntr", and "checkSclkCntr" states. Notice that this causes the SCLK to be low for 20 ns more than SCLK being high and that is perfectly fine and does not violate the SPI timing in any way. You can account for this by setting the "timerMax" in state "sclkLo" to 47 instead of 49. If the "sclkCntr" reaches 24, then the "SPI FSM" transitions to setting the CS high in "setCShi" state. After that, the "SPI FSM" then transitions to a state that waits for 100 ms before completing the "SPI FSM" and setting the "SPIdone" signal high. Remember that the signal "SPIdone" is used as a handshake signal for the main "Command FSM". Don't forget to clear the "sclkCntr" once you are done with the "SPI FSM"; this can be done in state "wait100ms". If you are wondering that the state "wait100ms" is used for, I created that state to slow down the polling of the ADXL362 to just a few Hz so that the movement of the red square is not overly sensitive and jumpy. One final note about the "SPI FSM" is that the "SPIdone" signal goes high during "wait100ms" AND timerDone goes high; this is why "SPIdone" is highlighted in red to bring that to your attention.

The final two processes to discuss are the "Parallel-To-Serial" process that drives the MOSI from the FPGA to ADXL362 and the "Serial-To_Parallel" process that uses the MISO to reconstruct the 8-bit data values from the ADXL362. The "Parallel-To-Serial" process uses the register called "toSPIbytes" to convert the 24-bit command to a serial sequence that is then driven to the ADXL362 using the MOSI output. As a hint for creating this process, you will need to load the register "toSPIbytes" into a 24-bit shift register when "SPIstart" pulses high. Then, when the "SPI FSM" is in state "sclkHI" state AND timerDone goes high, you will need to cyclically shift the 24-bit shift register one bit to the left. The MSB of the 24-bit shift register then becomes the MOSI output.

As for the "Serial-To-Parallel" process which reconstructs the serial data from the ADXL362 into 8-bit values, you should use another (different) 24-bit shift register that shifts in the MISO signal into the LSB of the register when the "SPI FSM" is in the "schkSclkCntr" state AND as long as sclkCntr < 24. The 8 LSBs of that 24-bit shift register will contain the data values you

need and you are required to capture these values during the states "captureID_AD", "captureID_1D", "captureX", "captureY", and "captureZ" respectively. Therefore, you will need five separate 8-bit registers as part of the "Serial-To-Parallel" design to capture the two IDs and XYZ values respectively. The IDs and XYZ values are then used in the top level for displaying to the sever segment display. Only the X and Y values will be used for moving the red square.