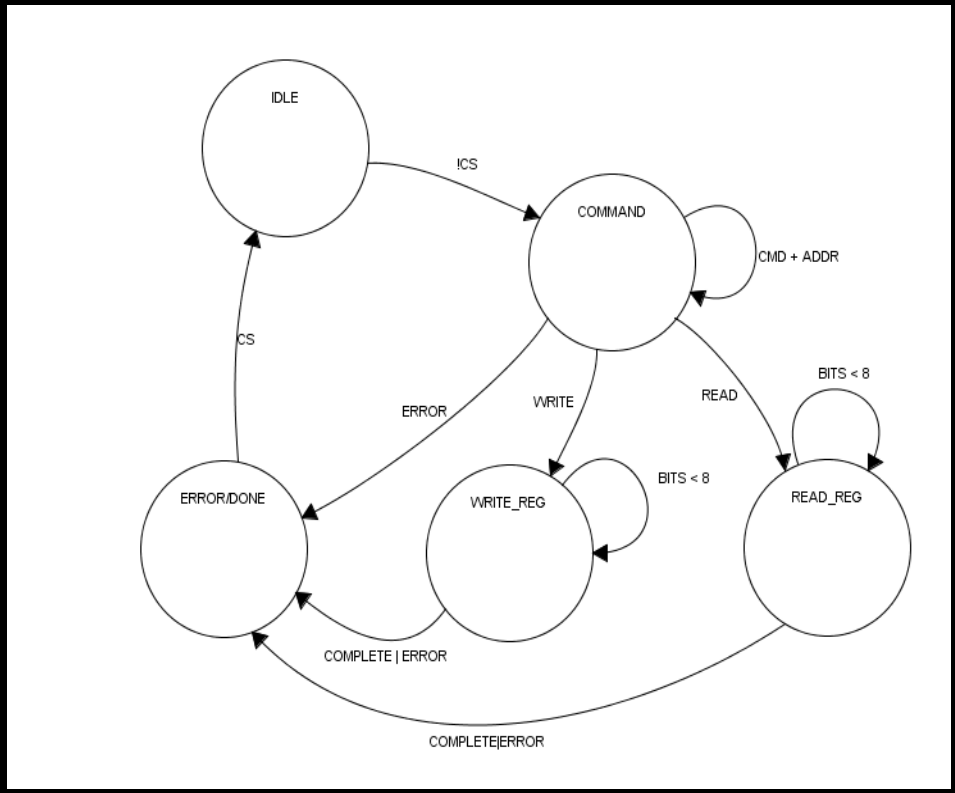
L5 SPI Slave Lab

- Dedicate one Tiva to master and one to slave Behavior
- Most of the master and much of the slave functionality can be leveraged from prior labs.
- We will take two lab periods to complete this lab and extend only if necessary.

How does the Slave Operate?

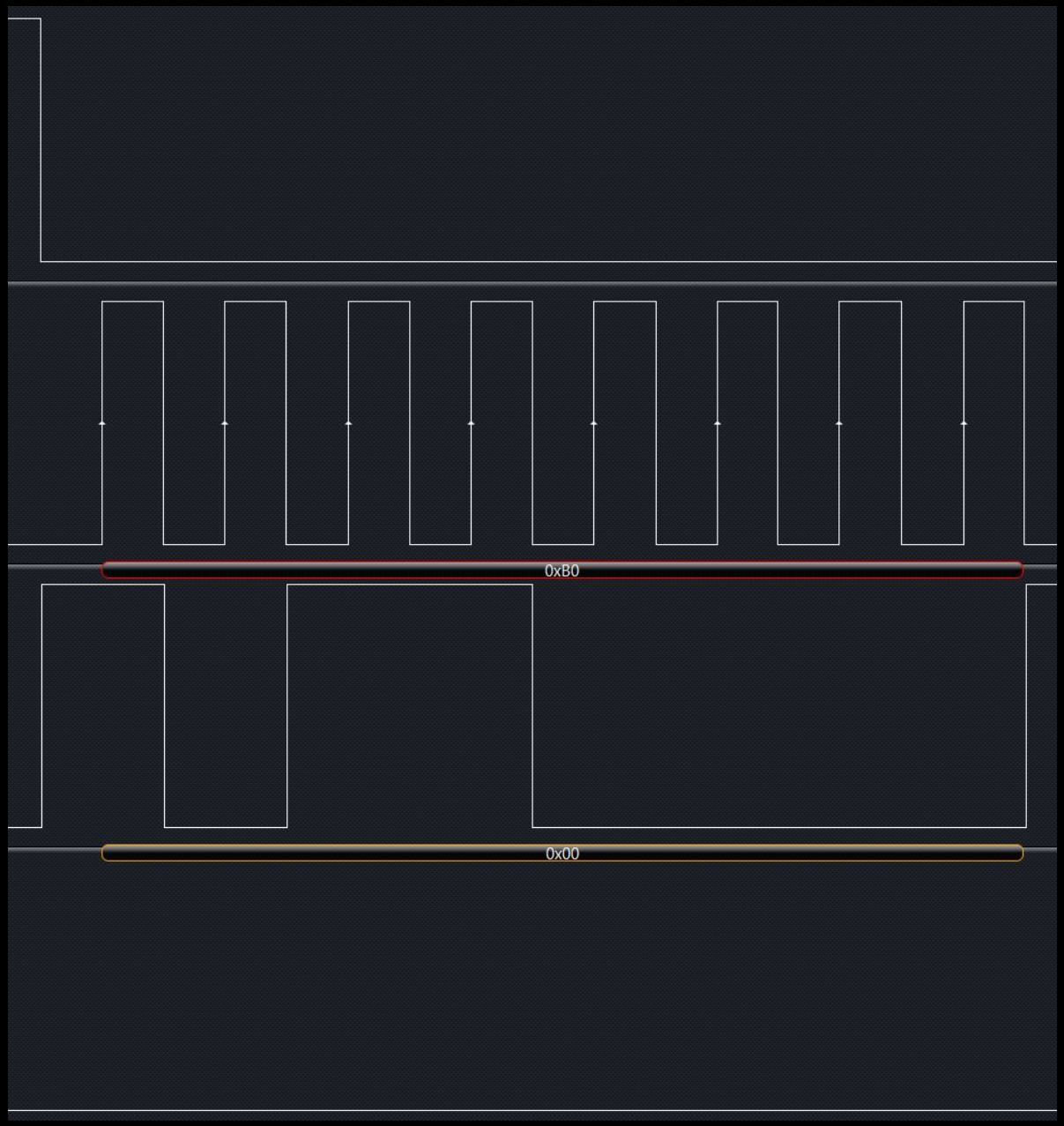
- Is completely asynchronous (responding only to inbound SPI Clock)
- Need a behavioral description before we start to code.
- Slave will carry a rash of state…
- I strongly suggest that you code the state into a state machine like below.

- Based on state where are we?
- Example for simplified state machine.
- Other suggestion would be to use a hierarchal machine but it's not really that complex.
- Working through a set of commands.



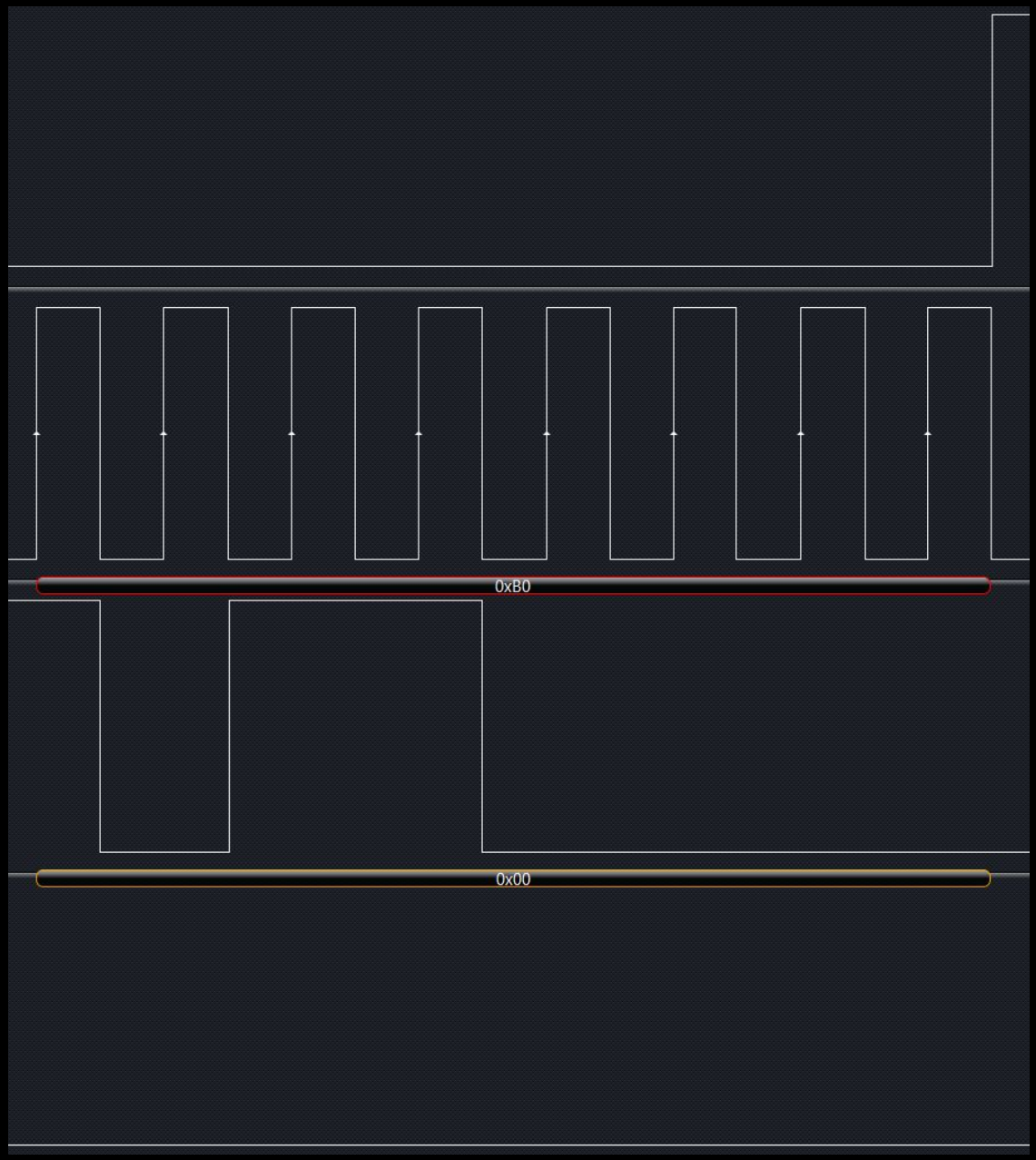
SPI Write Part 1

- This is the digital send of a write operation that we have used so far.
- The first byte has 4 bits of preamble 0xb for a `write` followed by 4 bits of address.
- This is the first byte of a two byte command that will store the second byte at address 0



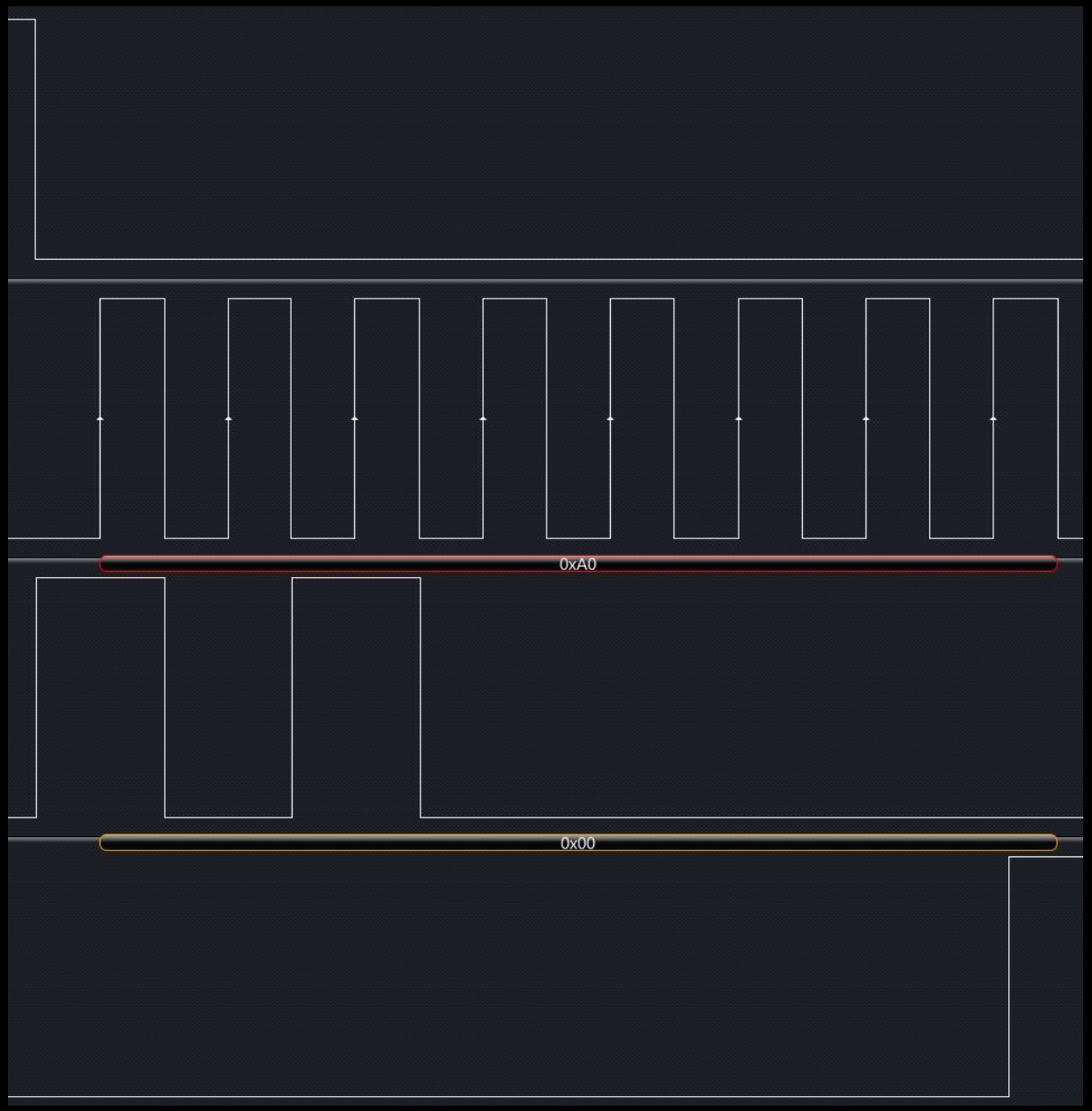
SPI Write Part 2

- The second byte (sorry that it`s also 0xb0) is sent and sampled on the clock rising edge.
- Note that after the second byte write CS goes high and the operation is complete.



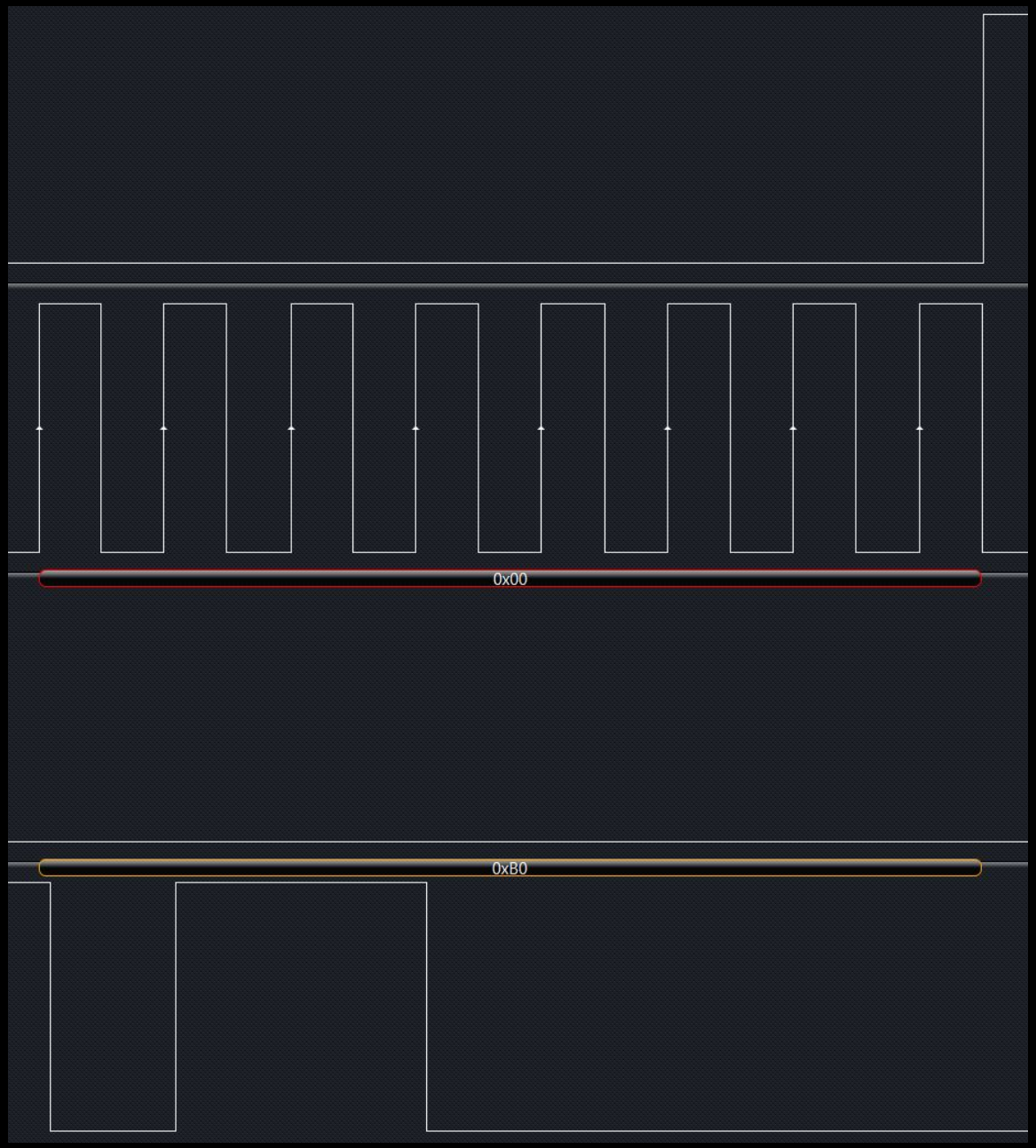
SPI Read Part 1

- The read operation is indicated by the upper byte of the first transfer = 0xA the remaining 4 bits are the register address (0-15)
- Note that the slave (setting MISO) needs to prepare for the next clock rising edge by getting his first bit ready.



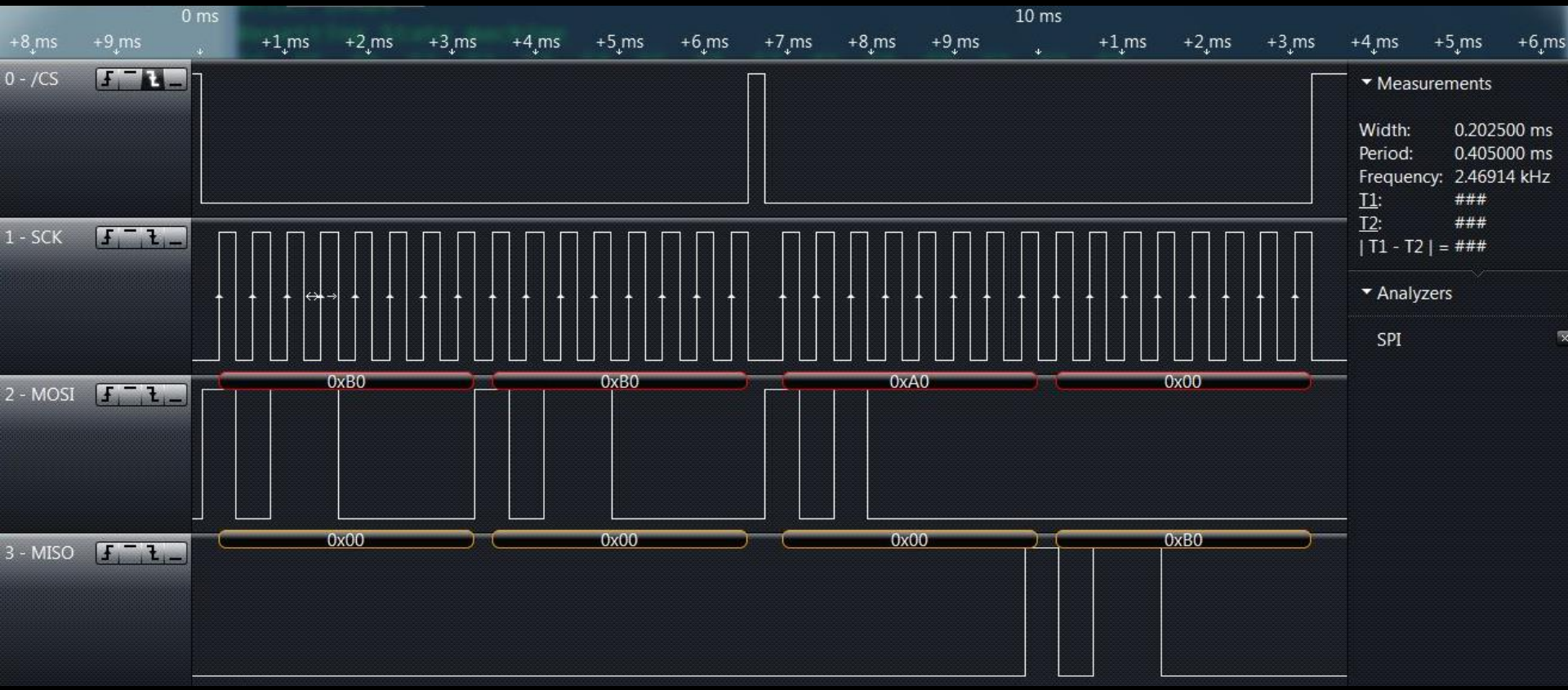
SPI Read Part 2

- For the second part of the read the master sends only 0`s
- The slave continues to have data ready for the master to sample on each rising edge of the clock.



SPI Write/Read All Together

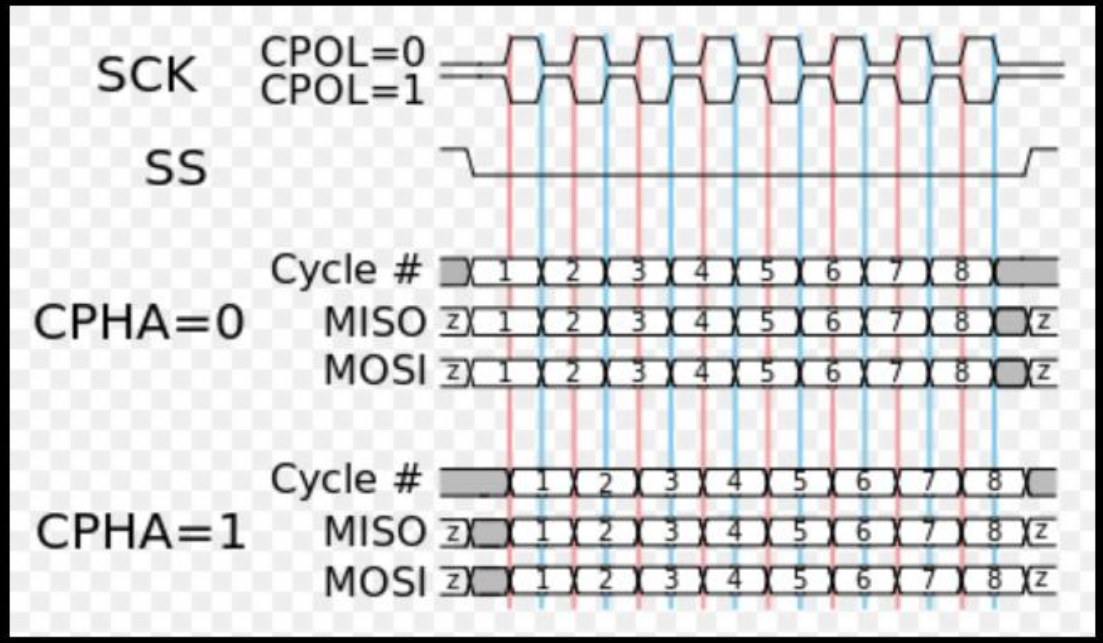
- Four actual transfers, two in the write and two in the read.
- Different Devices will have different protocols and not all are limited to 8 bit.
- A 12-bit ADC may have a 12-bit read cycle available but will probably have a mode to return the data in two subsequent 8-bit reads to support old processors.



SPI Options

- When trying to talk to an SPI device for the first time there are some differences that you have to account for..
- CPOL (Clock Polarity) defines the resting clock state
- CPHA (Clock Phase) defines whether you latch on leading or trailing edge

- Device Manufacturers sometimes state this clearly and sometimes not.
- SS we call !CS



Generic State Machine Code

```
static void
_slaveTask( void *parm )
{
    //
    // Set the possible states my machine can have.
    // initialize to ERROR
    //
    enum {S_IDLE, S_CMD, S_READ, S_WRITE, S_ERROR} state = S_ERROR;

    while(1)
    {
        xQueueReceive(_q, &qData, portMAX_DELAY);

        switch(state)
        {
            case S_IDLE:
                if (/* valid */)
                {
                    // act on interesting data only
                    if (/* data */)
                        state = S_CMD;
                    else
                        state = S_ERROR;
                }
                break;

            case S_CMD:
                '''
                break;
```

- State variable defined locally and bound to known states
- Block on inbound data so that it can free run
- Perfectly legal to stay in a state ‘by not setting a new state’
- it is normal to maintain counters within a state.

Generic State Machine Code (continued)

```
            case S_WRITE:
                '''
                break;

            case S_ERROR:
                //
                // If we find the error state, Just hang here until
                // CS is high and CLK is low.
                // Note that since I set the prior after this switch
                // I need to watch for the edges as well
                //
                if (((lcs == HIGH) || (qData == T_CS_HIGH))
                    && ((lclk = LOW) || (qData == T_CLK_LOW)))
                {
                    state = S_IDLE;
                }
                break;

            default:
                assert(("default case",0));
                state = S_ERROR;
                break;
        }

        // perform state independent data here...

        // setting last_<name> variables.

        // setting output devices, LED's GPIO's whatever
        // if possible it's a little more clean here.. but
        // like MISO, my need imnplementation in the actual
        // state cases.
    }
}
```

- If the state variable encounters an unknown state, assert.
- This code may run in release mode someday, It is important to still do something if the assert() were not there.
- Not sure of the logic here, what would exit S_ERROR state.
- (cs==HIGH && T_CLK_LOW) || (clk==LOW and T_CS_HIGH) may be better?

Implementation Stages

- We generally only have on programming machine per team and later will want a windows machine around to run the lab on.
- Write your generic master ‘before’ you come to lab.
- Work out your slave state machine ‘before’ you come to lab.
- Interrupt debugging is difficult.
- If possible, Design your program to be written and debugged in a slower polled method then switch after all the logic works.
- Fortunately we have such a system.
- Adding the Watchdog is an afterthought but I will supply some code by the second week of class.
  - Review the watchdog section of the book starting at Pg 71
  - Review the Watchdog Timer section of the Tiva Driver Library API