3D Wireless Mouse

Our project is a wireless, 3-dimensional, handheld computer mouse. The user will be able to control the computer cursor by rotating his/her hand in a joystick fashion. It will be implemented using a 3-axis accelerometer for sensor input, a RF radio kit for wireless capability, and the Labkit. The Labkit will be connected to the computer using a PS/2 interface.

Shirley Li, Matthew Tanwanteng, and Joseph Cheng
TA: Charlie Kehoe
6.111 Introductory Digitial Systems Laboratory
12 May 2005

1. Introduction

The 3D Wireless Mouse takes in the user's hand acceleration as input and moves the mouse on a normal personal computer appropriately. The initial goal of the project was to calculate the exact position of the hand to determine the movement of the mouse on the computer screen. However, that goal required more hardware parts, more money, and more time than we can afford. The final project analyzes the tilt of the user's hand and moves the computer mouse accordingly.

The user puts on the 3D Wireless Mouse glove on the right hand, and plugs the device into the PS/2 port of any personal computer. The reset button is then pushed to initialize the entire system. On tilting the glove left and right, the mouse on the computer screen moves left and right. On tilting the glove forward and backwards, the mouse on the computer screen moves up and down. The user can use additional push buttons to left and right click.

The system can be broken up into three main sections: sending data wirelessly from the accelerometer to the FPGA, filtering and calculations in the FPGA, and then the PS/2 interface to communicate with a computer. Figure 1 shows a rough block diagram of the system. The user interacts the system by moving the accelerometer around in the air. At any moment in time, the user can also re-initialize the system.

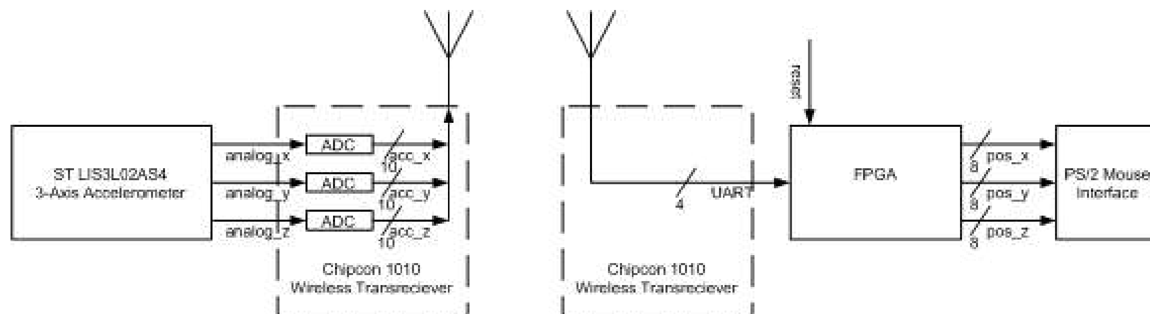


Figure 1: General Block Diagram.

2. Description and Implementation

2.1 Wireless Segment

The wireless segment includes the accelerometer chip and the CC1010 RF radio. The accelerometer is the only sensor input to the system. It is used to determine the position of the hand. An RF radio transceiver wirelessly transmits the accelerometer data to a base station that is connected to the Labkit via an UART (Universal Asynchronous Receiver/Transmitter) cable. Thus, the output of the wireless segment to the Labkit is serial asynchronous acceleration data. The accelerometer and transceiver are powered by 3V lithium batteries.

2.1.1. Accelerometer

The accelerometer being used is the LIS30L02 3-axis model from ST Microelectronics. It has a 2g range and analog voltage outputs for each axis that are radiometric to `Vdd`. The x and y-axes have maximum bandwidths of 4 KHz. The z-axis has a maximum bandwidth of 2.5 KHz. The

accelerometer is mounted on the inside of a glove such that when worn, the chip lies flat on top of the hand, with the z-axis parallel to the gravity vector.

2.1.2. CC1010 RF Radio Hardware

The Chipcon CC1010 RF radio is used to implement the wireless capability. The radio kit has two identical transceivers and an evaluation board. Each transceiver is capable of transmitting and receiving data, but for the purpose of this project, one transceiver functions solely as a transmitter and one solely as a receiver. The evaluation board is essentially a docking station for the receiver. It provides power to the receiver and contains several LED's that are useful for debugging, as well as an array of I/O pins, potentiometers, and the UART port that will be used to interface with the Labkit.

There is a 10-bit ADC and a microcontroller on board each transceiver. The microcontrollers are programmed in C and control the operations of the transceivers. Each ADC has three input ports from which data can be taken and digitalized. The transmitter ADC input ports are connected directly to the accelerometer outputs. The receiver ADC is not used.

2.1.3. Transmitter Algorithm

The functionality of the transmitter is to continuously sample the accelerometer and send the digital data to the receiver base station. During each cycle of operation, each ADC input port is activated and a conversion occurs. Only the top eight bits are taken. Thus, each accelerometer axis gives one byte of data. After all three inputs are sampled, the three bytes of data are transmitted together as one packet. The transmitter executes this procedure in an infinite loop.

The ADC is operated in single-conversion mode, which means conversions are manually initiated, rather than automatically initiated by a timer. The three inputs are sampled right after each other. The output of the ADC is unipolar, ranging from 0 to 1023, with 1023 corresponding to `Vdd`. However, since it is easier to transmit in bytes, only the 8 MSB are saved.

In each cycle of operation, after all three inputs are digitalized, a 3-byte long packet is transmitted at 2.4 kBaud using the Manchester encoding scheme. Manchester encoding works in the following way: a '1' is represented by a high frequency f_1 followed by a low frequency f_0 and a '0' is represented by a low frequency f_0 followed by a high frequency f_1. The DC component of the transmitted signal is therefore 0, which is desirable with regard to energy conservation. For these transceivers, f_0 and f_1 are centered around 868 MHz and separated by 64 KHz. Also, since there is one transition per encoded bit, a clock is effectively sent along with the data. Thus, Manchester encoding is a synchronous protocol and allows for easy resynchronization of the data at the receiver.

Every time a packet is sent, a preamble and a sync byte are appended to the front of the data. The preamble alerts the receiver that there is incoming data, and consists of 7 bytes of alternating 0's and 1's. The sync byte is used to signal the beginning of the data sequence and is set to be 10100101. Figure 2.1.3 explains Manchester encoding and also the structure of the preamble.

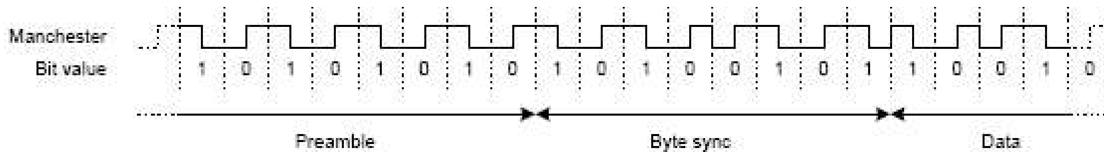


Figure 2.1.3: Manchester encoding.

2.1.4. Receiver Algorithm

The receiver continuously listens for the repeating 0-1 preamble pattern that indicates incoming data. When it successfully receives at least 7 alternating 0's and 1's, it will start to look for the sync byte. If the sync byte is received, then the data is read in, stored, and sent to the UART port. If the sync byte is not received after a deviation from the 0-1 pattern, then the receiver goes back to preamble detection mode.

The received data is sent to the Labkit via a UART cable. UART, or Universal Asynchronous Receiver/Transmitter, is an asynchronous serial protocol. This means that data is transmitted serially through it without a clock signal. Thus, the data rate must be predetermined and the receiving end of the UART must know what it is. The data rate for this project was set to 115 kBaud. When there is no data being transmitted, the data line remains in a high idle state. Data is sent in bytes, and each byte is preceded by a start bit and followed by a stop bit. An optional parity bit is sometimes sent for error detection, but was not used in this project. The start and stop bits are active low. Figure 2.1.4 shows an example waveform of the output of the UART when one byte of data is sent.

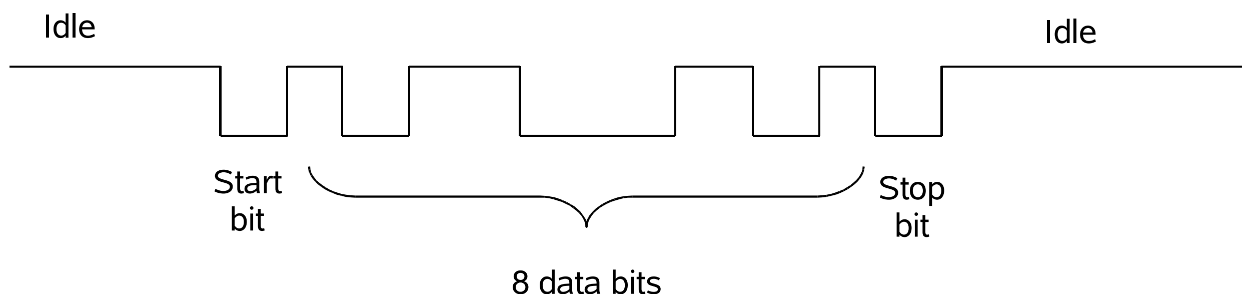


Figure 2.1.4: UART output

To read in the serial data, the data line needs to be continuously sampled so that the start bit is detected. Once the start bit is detected, the data line is sampled at the baud rate (115,200 bits/sec) to extract the data bits. After 8 samples are taken, the interface then checks for the stop bit. If the stop bit is detected, then the data is saved. If the stop bit is not detected, then the byte of data is discarded.

2.2. FPGA Serial Interface (by Joseph Cheng)

The FPGA interfacing allows the Labkit to communicate with the CC1010. The CC1010 connects to the Labkit through a UART port. The FPGA uses a bunch of registers controlled by a finite state

machine to take the serial data and convert it into parallel data. The finite state machine reads in the data from the UART, and it loads the data appropriately into a bunch of registers. Then, it extracts the 8-bit binary offset data point. The data read is then converted to twos complement.

Part of the UART interface is a set of registers that latch the acceleration data from each axis. The data of the three-axis comes in through the same UART port serially, so the registers store the different acceleration data appropriately. When the user first resets the system, the first set of three acceleration data points is registered. This registered value is used as a reference point for other acceleration data. Whenever a new value is read in, it is added to the negative value of the reference point.

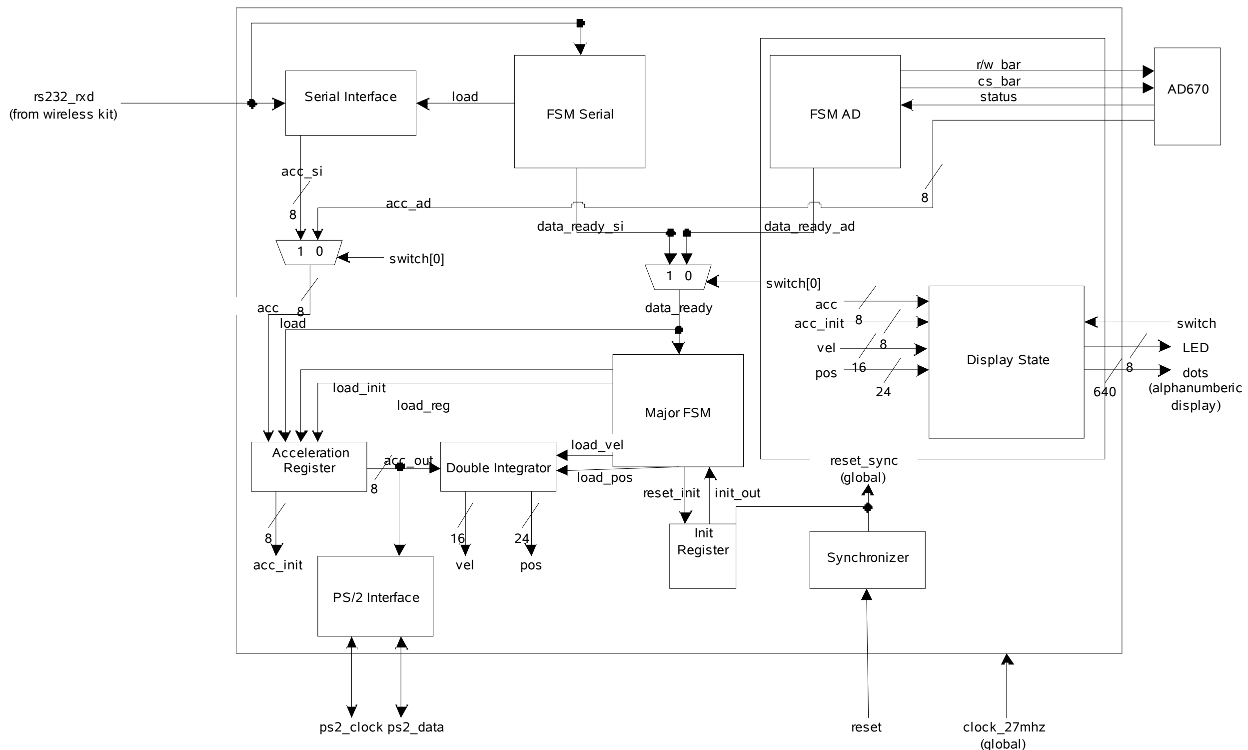


Figure 2.2: Detailed Block Diagram of Serial Interface and Calculations.

The diagram in Figure 2.2. shows the different portions of the Serial Interface and the calculations. The debugging modules were also included to fully show how the user can interact with the system. Note that in this diagram only one axis is being considered for the acceleration data, velocity data calculations, and etc. For the other axis, the same modules are integrated with the same FSMs.

2.2.1. Synchronizer

The Synchronizer module takes in the 27 MHz clock signal and a user inputted `reset` signal. The module then outputs a synchronized, glitch-free `reset_sync` signal. The purpose of this module is to make sure that the user inputted `reset` signal is glitch-free and synchronized to the clock edge to avoid problems. Without this module, any sort of glitch will make the system behave unpredictably. The Synchronizer module is implemented with two registers in series. Therefore, it

takes two clock signals before the actual reset signal from the user is outputted as the reset_sync. This reset_sync will be all other modules in the system.

2.2.2. Serial Interface

The Serial Interface takes in a serial line and converts the data into parallel data. This module takes in the 27 MHz clock signal, a load signal, and a data rxd signal. The serial rxd signal comes from the Universal Asynchronous Receiver/Transmitter (UART). Since the wireless kit is sending three packets, the Serial Interface module converts the serial signal into a set of three 8-bits data points: acc_x, acc_y, and acc_z.

The Serial Interface module is implemented with a set of 24 registers. Every time a valid data point comes in, a high load signal registers the new data and keeps the old data in the next register. The 24 registers allow the module to register 24 bits. In this way, 3 bytes of 8 bits can be extracted from the serial line. Note that the loading of the signal is synchronized to a positive clock edge.

2.2.3. FSM Serial

A finite state machine (FSM) controls the Serial Interface module, so that the appropriate data points are registered into the system. This module takes in the 27 MHz clock, the reset_sync from the Synchronizer, and the rxd signal from the UART. The module then outputs a load_data signal to load the appropriate data into the Serial Interface and a data_ready signal to let the system know that the Serial Interface has valid data. The rxd signal is registered to prevent glitching. See Figure 2.2.3 for a FSM diagram.

The data comes in through the rxd signal, and 3 bytes are read. Each bit begins with a low start bit and stops with a high end bit. The start and stop bits let the reader know when to start reading the data and when to stop reading the data. The reader also knows if it is reading a valid byte by looking for a low start bit, 8 bits of data, and then a high stop bit. If these specifications are not met, the data is trashed. Note that 3 bytes are sent in one packet. The data is coming in at 115200 bits per second (bps), so it takes about 234 clock cycles for one bit to come in.

Upon reset, the FSM starts at the WAIT_START state. At this state, the FSM waits for a low stop bit coming in through the rxd signal. The FSM resets the bytes counter to 0, and during the next states, the FSM will increment this counter for every byte read.

On a low start bit, the FSM transitions to the START state. A baud rate counter is started. When the counter counts 117 clock cycles, the FSM checks to make sure that the rxd signal is still a valid low start bit. If it's invalid, the FSM transitions back to the WAIT_START state. Otherwise, the FSM keeps incrementing the baud rate counter. When the baud rate counter counts up to 234 clock cycles, the FSM transitions to the next state, READ_DATA. The FSM also starts a bits counter to count the number of bits being read.

At the READ_DATA state, the FSM restarts the baud rate counter. When the baud rate counter is at 117, the FSM sends out a high load_data signal to the Serial Interface to load in the valid data bit. When the baud rate counter is at 234, the FSM transitions to the next state. If the bit

counter is at 8, the next state is the STOP state. If the bit counter is smaller than 8, then the FSM will restart the READ_DATA state and read the next bit.

At the STOP state, the baud rate counter is restarted. When this counter is at 117 clock cycles, then the FSM checks to make sure that the `rxd` signal is a valid high stop bit. If it is not, the FSM transitions back to the WAIT_START state. When the baud rate counter is at 234 clock cycles, the FSM checks the bytes counter. If the bytes counter counts less than 3 bytes, then the FSM goes back to the START state to read the next byte. If the counter counts the current word as the third byte, then the FSM transitions to the WAIT_START state, and waits for the next packet of data.

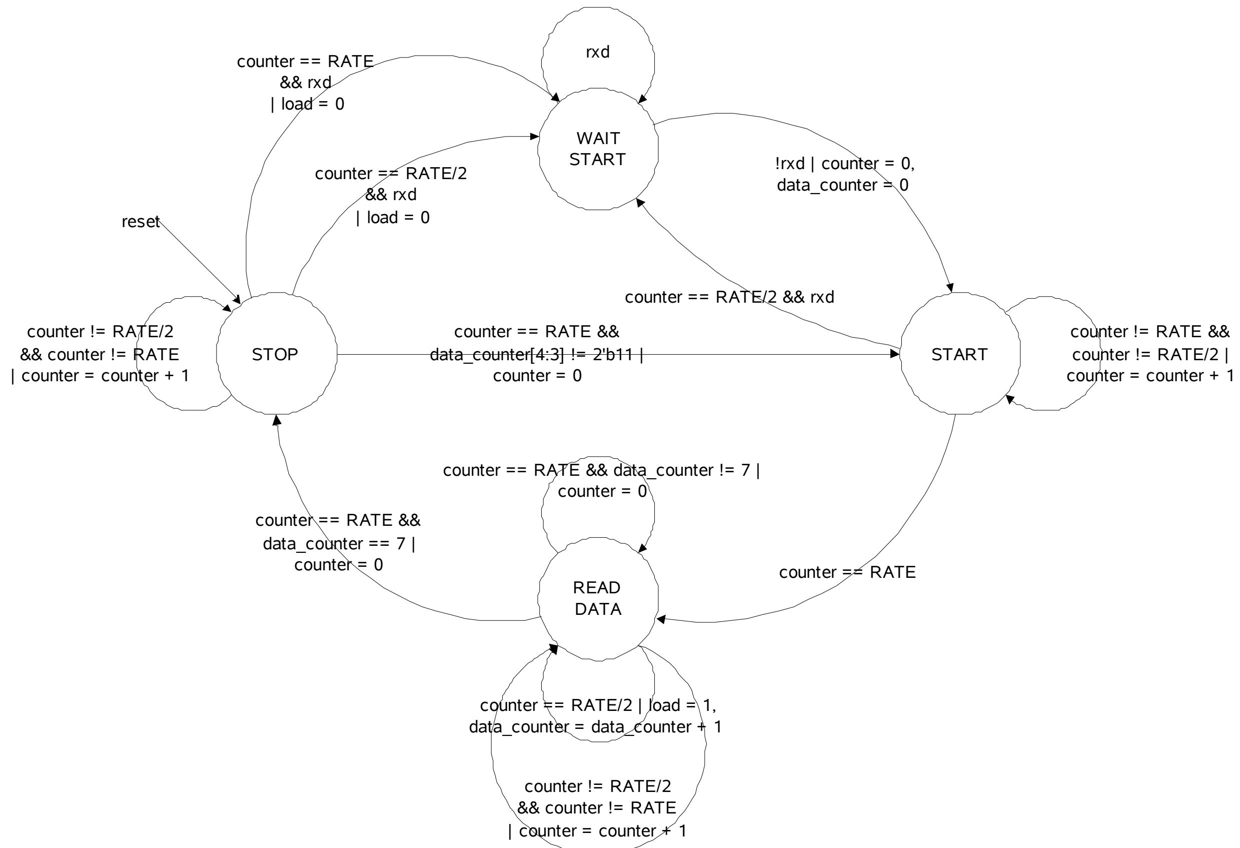


Figure 2.2.3: FSM Serial Module State Diagram.

2.3. FPGA Calculations

Originally, we were planning on calculating the absolute position of the mouse. However, upon many test on the accelerometer there are major problems with calculating the absolute position. These problems will be discussed more in depth late in section 3. The theory and implementation of calculating absolute position will be discussed here.

The data coming in from the wireless kit is acceleration. What is needed to move the mouse around is the relative position. This is done through integrating the acceleration twice. The first integration outputs velocity; the second integration outputs the position. To implement the integration function, an accumulator was used to add up all the different samples. To just get the change in position from

the last sample, the accumulator can be reset to 0, so that only the change in position can be calculated.

After each integration step, the number of bits increases. Since acceleration is not that high, I assumed the velocity to be 15 bits and the position to be 20 bits. The number of bits can be modified later if the number of bits is creating issues with calculations.

2.3.1. Initial Register

The Initial Register module takes in the `reset_sync` signal, the 27 MHz `clock` signal, and an `init_reset` signal. The module outputs an `init_out` signal. The Initial Register module registers the `reset_sync` signal. When a high `reset_sync` signal is registered, the system can perform its many steps of initializing before resetting the `reset_sync` signal back to a low with an active low `init_reset` signal. The Initial Register essentially behaves like a walk request register, where the walk request is the same as a high `reset_sync` signal.

2.3.2. Acceleration Register

The Acceleration Register module takes in a 27 MHz `clock` signal, a `load` signal, a `load_reg` signal, `load_init` signal, and an 8-bit `acc_in` signal. The module then outputs an 8-bit `acc_init` and an 8-bit `acc_out`. See Figure 2.3.2 for more detail on the module.

On each high `load` signal, the module latches the current 8-bit `acc_in` signal. The `load` signal comes from the `data_ready` signal of the Serial FSM. This allows the valid data to be latched into the system.

The Acceleration Register module registers up to the past four data points, so that it can calculate the current average as an 8-bit `acc_reg` data. This calculation smooth out the data points to filter out some of the noise.

On a high `load_init` signal, the Acceleration Register latches the current `acc_reg`. The module then converts the data to be in twos complement. Afterwards, the twos complement data is converted the negative version of the data and outputted as an 8-bit `acc_init`. This data point is used as a reference point to do all other calculations. This data point is considered to be the equilibrium data point. Therefore, when this negative data point is added to the newer data points, the output is adjusted according to this reference. The adjusted output is the 8-bit `acc_out` signal.

Note that for the final tilt mouse, the `acc_out` signal is directly sent to the PS/2 interface.

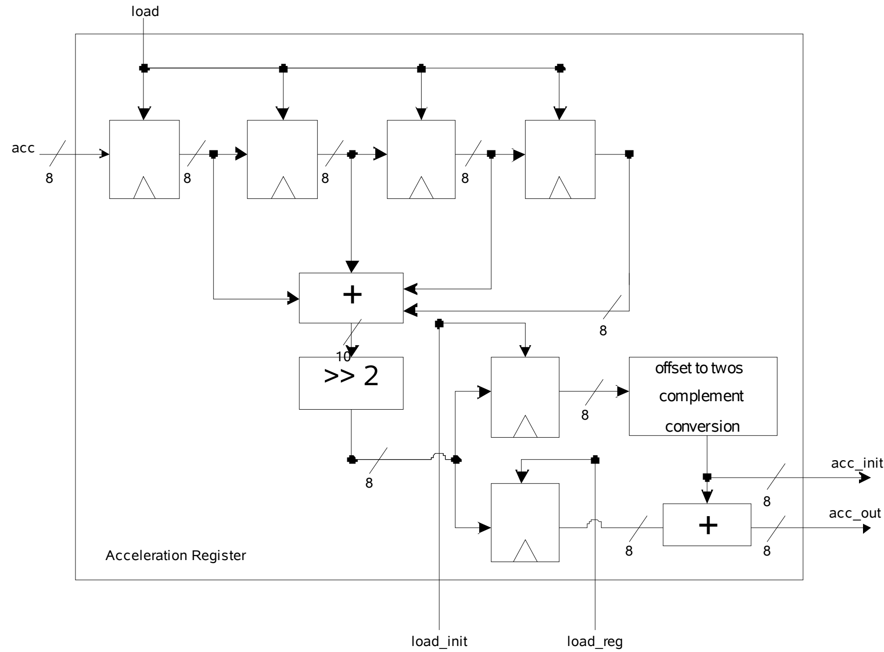


Figure 2.3.2: Acceleration Register Block Diagram.

2.3.3. Double Integrator

The Double Integrator module takes in the 27 MHz `clock` signal, an 8-bit `acc` data point, an active low `reset_vel` signal, a `load_vel` signal, an active low `reset_pos` signal, and a `load_pos` signal. An 8-bit `vel` and 8-bit `pos` is outputted from the module.

The purpose of this module is to take in the 8-bit `acc` data point representing acceleration, integrate to get the 8-bit `vel` data point representing velocity, and integrate again to get the 8-bit `pos` data point representing position.

To implement the integration addition accumulators are used. A low `reset_vel` signal resets the velocity accumulator to 0. A high `load_vel` signal registers the 8-bit `acc` data point and adds it to the current `vel`. The same implementation is used to integrate from velocity to position, except a low `reset_pos` signal resets the position accumulator to 0 and a high `load_pos` signal registers the current `vel` data.

Figure 2.3.2 shows how the registers and accumulators are placed together to create the Double Integrator module.

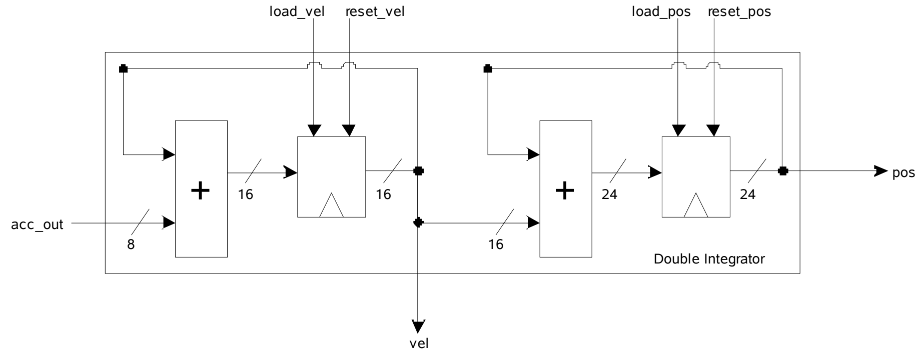


Figure 2.3.2: Double Integrator Block Diagram.

2.3.5. Major FSM

The Major FSM controls the entire system of calculations. The Major FSM first makes sure that upon initialization the system takes in enough data to find a good average. Then, the module does the necessary data loading to perform the rest of the calculations. Figure 2.3.5 shows the FSM state diagram.

This module takes in the 27 MHz `clock`, the global `reset_sync` signal, the `data_ready` signal from the Serial Interface, and the `init_out` signal from the Initial Register. The Major FSM outputs an `init_reset` signal to the Initial Register, `load_init` and `load_reg` signals to the Acceration Register, `load_filter` to a possible Filter module, and `load_int_1` and `load_int_2` signals to the Double Integrator module.

The Major FSM starts in the WAIT state. On a high `init_out` signal from the Initial Register (meaning a reset from the user), the state transitions to the INITIAL WAIT state. Here, the FSM waits until it receives a high `data_ready` signal to know that there is valid data ready. Also, at this state, an `init_counter` is reset to 0. This counter counts to make sure there is the right number of data points to take the first average. It is important that the system gets the right number of data points for the first average, because this first average is `acc_init`, the initial acceleration used as a reference point. All other accelerations will be subtracted from the initial to get relative accelerations. In the final project, 4 data points are averaged each time.

On a high `data_ready` signal, the FSM transitions to the INITIAL state. At this state, a high `load_init` signal is sent to the Acceleration Register to load the initial data point. If `init_counter` is equal to 4, the next state is the WAIT state again. If the counter is not at 4 yet, the system must take in more data points, so the FSM transitions back to INITIAL WAIT and increments the `init_counter`.

At the WAIT state, if a high `data_ready` signal is received, the FSM transitions to the REGISTER DATA state. Here, the module sends out a high `load_reg` signal to load the data point to a register.

The next state two states are the INTEGRATE 1 and the INTEGRATE 2states. Here, a high `load_int_1` signal for INTEGRATE 1 or `load_int_1` signal for INTEGRATE 2 is sent to

the Double Integrator module to perform the necessary computations for integration. The next state is then the WAIT state to wait for the next valid data point.

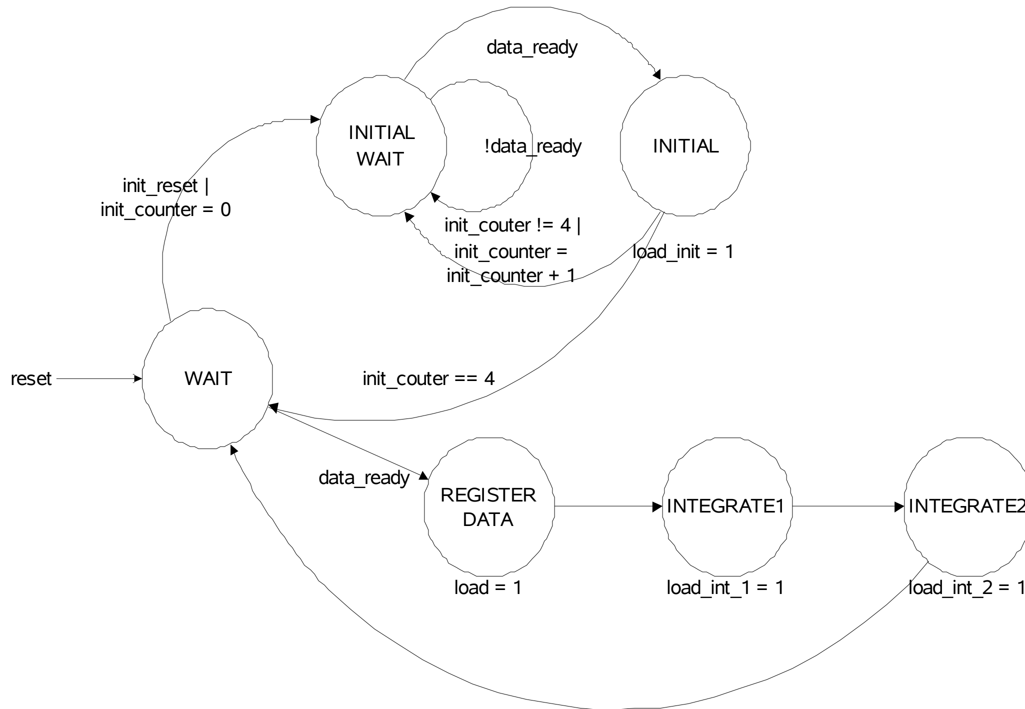


Figure 2.3.5: Major FSM Module State Diagram.

2.4. FPGA PS/2 Interface

The PS/2 Mouse Interface takes the position data collected from the accelerometer and translates the data into mouse movement and sends it to the host in PS/2 format. The PS/2 Protocol also requires host-to-device communications that set different device options such as device resolution and requires the proper responses to requests in order for the device to be recognized, and so the interface controls the input and output status of the clock and data bus lines to account for the bidirectional communication.

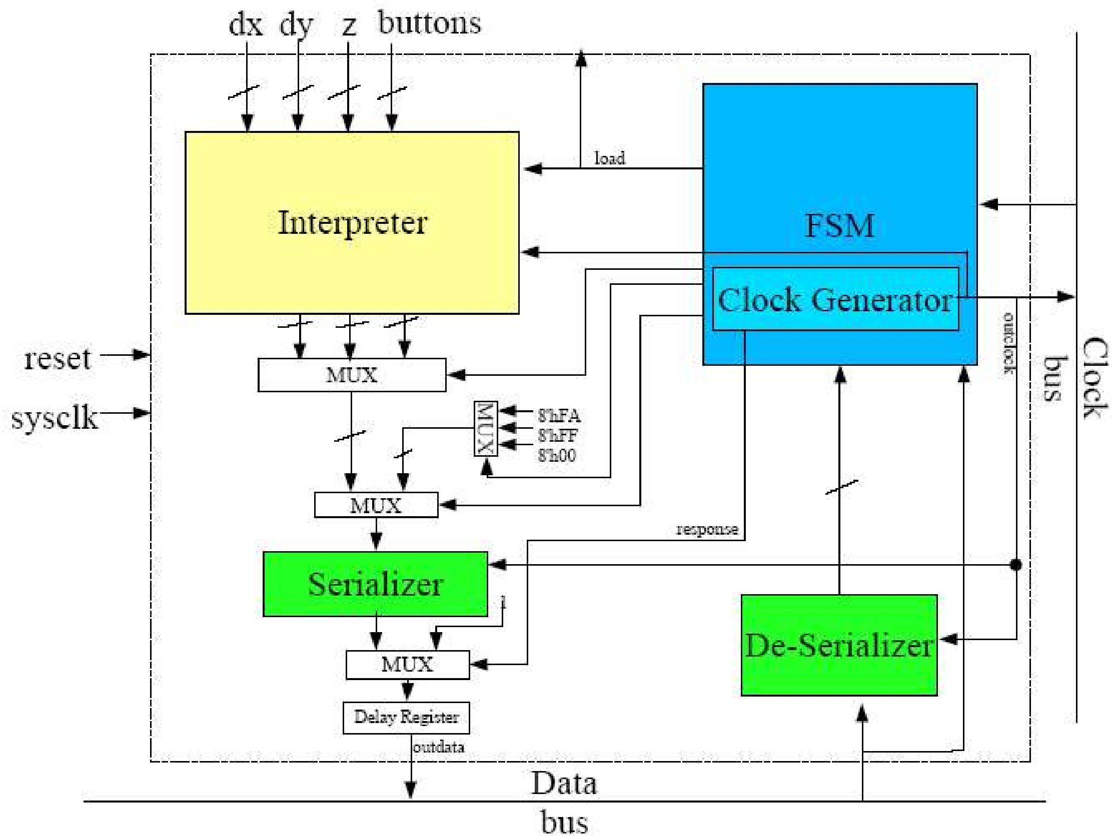
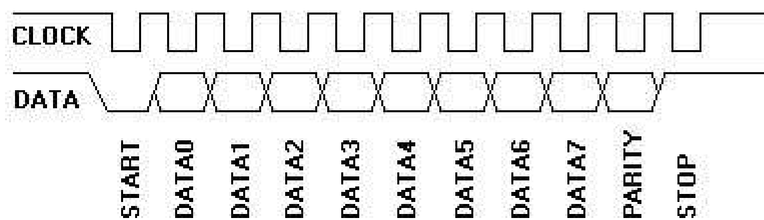


Figure 2.4: Block Diagram of PS/2 Interface

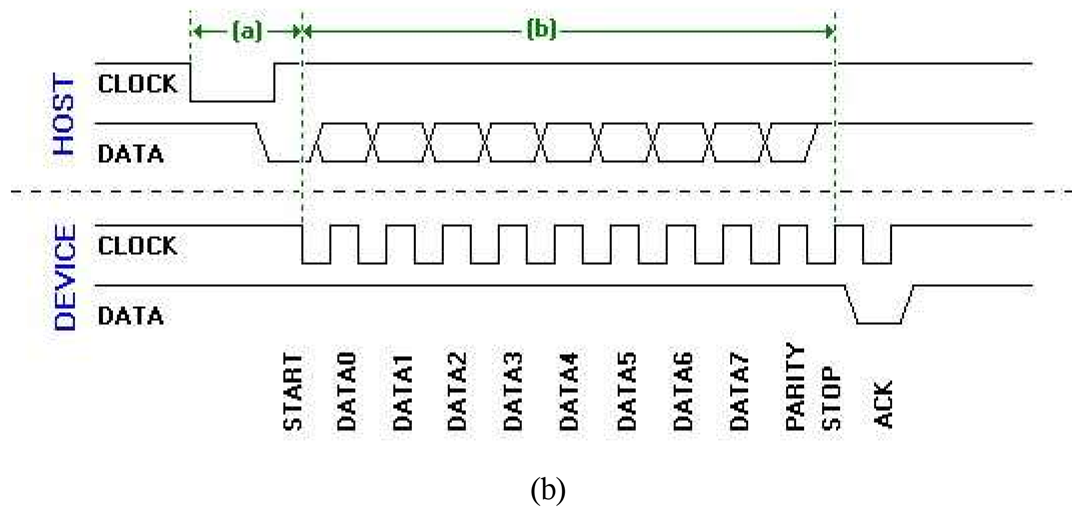
The PS/2 Mouse Interface is composed of five high-level modules, as seen in the block diagram. The interface takes the change in x and y positions, mouse clicks, and a reset signal. The clock and data are open-collector interfaces that are pulled high when no signal is being driven, and the host can inhibit communication by pulling the clock low.

2.4.1. Clock Generator

The clock generator simply creates a clock signal by which the host and device read and write data on the data bus. However the clock cannot always be active; the clock bus is only driven while a byte is being sent.



(a)



(b)

Figure 2.4.1.1: Device-to-host transmission timing (a)
Host-to-device transmission timing (b)
(From http://www.computer-engineering.org/ps2protocol/)

The generator is implemented as a simple FSM with a cycle that handles reading host-to-device requests and another cycle that handles device-to-host transmissions. The FSM is linked to a counter and an always active clock signal and outputs that active clock to the bus depending on the FSM state and the counter value. Note the acknowledgment bit the device sends in the host-to-device transmission is handled by the RESPOND and RESPOND2 states in the following clock generator diagram.

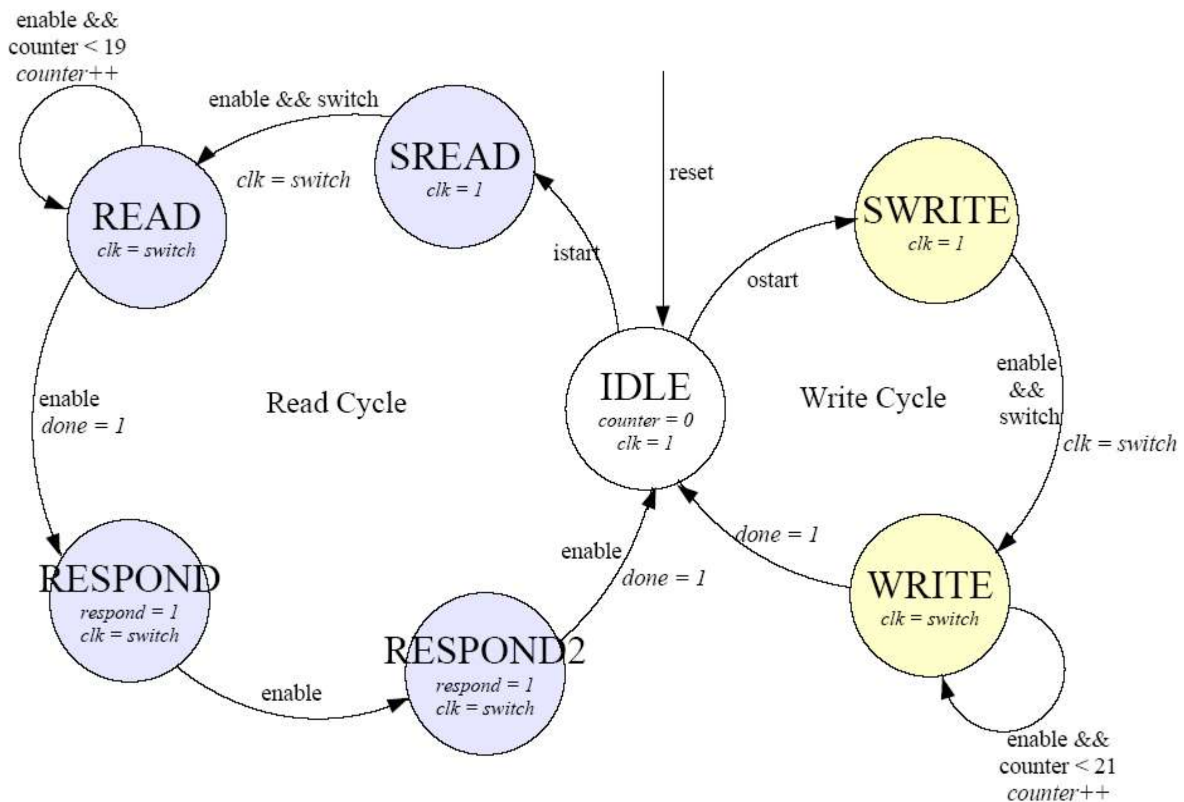


Figure 2.4.1.2: Clock Generator FSM
`enable` causes switch to flip, and switch is the always active `clock` signal.
The output `enable` for the clock signal is the inverted output `clk` signal.
(variables in italics are settings, unformatted variables are conditions)

2.4.2. Interpreter

The interpreter takes the mouse movement data and converts it into three 8-bit packets.

	Bit 7	*Bit 6*	*Bit 5*	*Bit 4*	*Bit 3*	*Bit 2*	*Bit 1*	*Bit 0*
Byte 1	Y overflow	X overflow	Y sign bit	X sign bit	Always 1	Middle Btn	Right Btn	Left Btn
Byte 2	X Movement							
Byte 3	Y Movement							

Table 2.4.2: PS/2 Mouse Packet

Originally when using positioning to control the mouse, the packet's middle and right button values are set high if the corresponding button input signals are high, and the left button is set as a function of the Z position. If the Z position is above the clicking boundary, the left button is asserted. If the Z position is below the tracking boundary, the X and Y bits are set to zero. For any Z position above the tracking boundary, the input X and Y data can be copied directly into the respective packet other than the sign bit, except when the X or Y has moved beyond the 9-bit limit, in which case the corresponding overflow bit is set high.

Because the end design uses tilt to track movement, there is no absolute Z data. Thus the interpreter's function is to simply package the input X and Y movement data and the clicks into data bytes for serialization.

The interpreter latches onto incoming X and Y data when a load signal from the FSM goes high. It checks to see if there is any X and Y movement or a change in a mouse button state first, and if no changes are detected no updates are required. Otherwise the interpreter sends a signal to the FSM to begin transmission and holds the signal high until the FSM responds to signal the update has been fully transmitted.

2.4.3. Serializer

The serialization module takes data bytes and converts them into 11-bit reversed packets of data in the PS/2 format. The 0-bit is output to the data bus and then the 11-bit packet is bit-shifted, so on the next clock cycle the next bit is output on the 0-bit, until the entire 11-bit packet has been transmitted across the data bus. Serializer is triggered when the clock generator produces a clock and the interpreter update signal is high.

2.4.4. De-Serializer

This module takes the data bus input stream and converts it into an 11-bit packet, which can then be read by the FSM. It stores the bits in a9-bit value as they stream in, bit-shifting left on every cycle, and when the module detects that the data has been loaded in it latches onto the 8-bit request stored in the message and outputs it until it is flushed before the next incoming message. The module loads a default value of all ones into the 9-bit storage value until the host activates the device read state, at which point the module loads the 0 start bit into the LSB. When that 0 reaches the MSB due to bit-shifting the other eight bits contain the relevant data byte. The system does not error check at this time; a simple restart re-synchronizes the device with the host.

2.4.5. Finite State Machine

The finite state machine (FSM) module controls the operation and timing of the separate modules. Because the clock and data buses are bidirectional, the FSM must halt operation of the output when it detects the host sending a request, and must also control the response to the host's request.

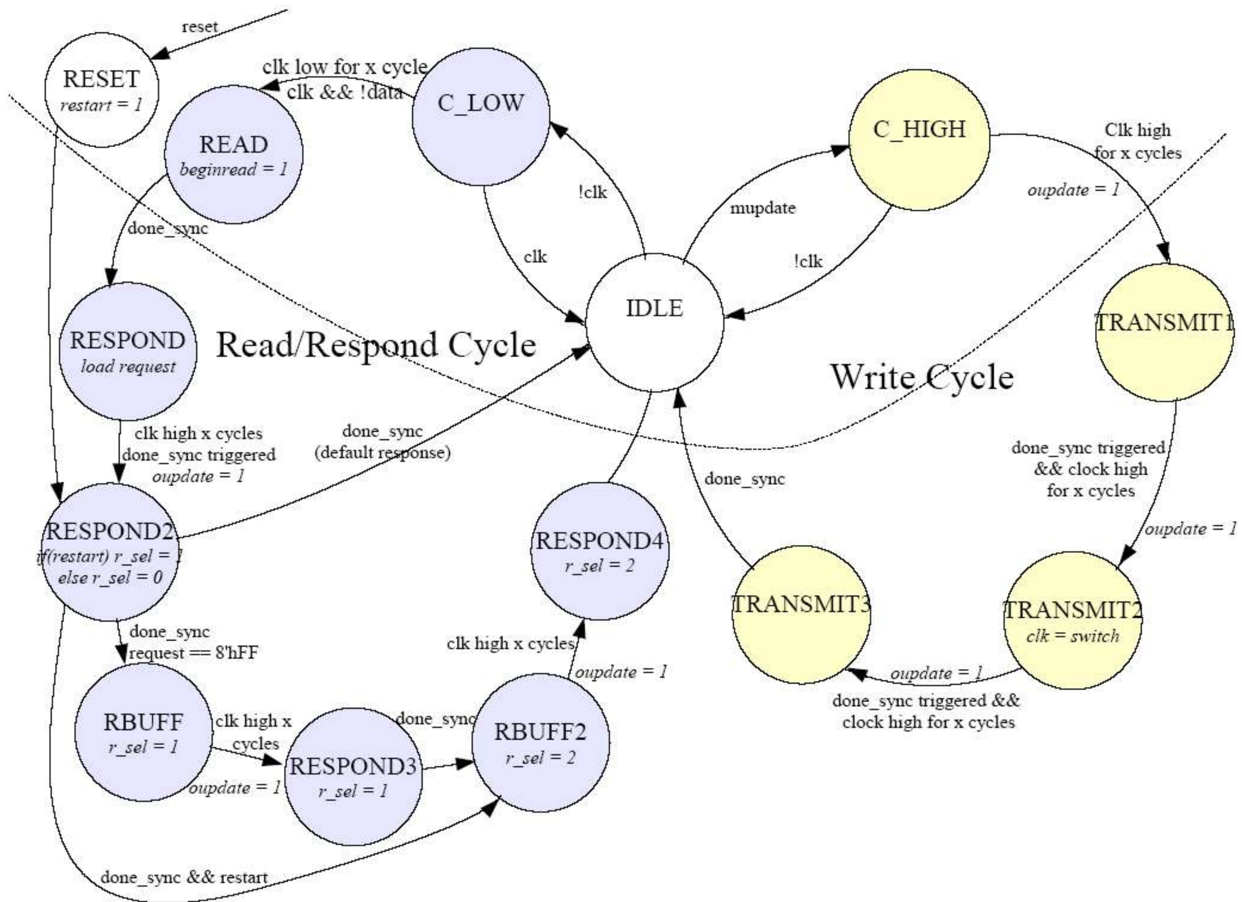


Figure 2.4.5: PS/2 Interface FSM

Above the dotted line, the data `enable` signal is low.
`mupdate` is a signal from the Interpreter initiating device-to-host transmission.
`r_Sel` is a selector that specifies responses to host requests.

The read/response cycle on the left of the state diagram is activated when the host signals a request by pulling the clock line low for at least 100 ms, pulling data low, and then releasing clock. The data is read in the READ state, and once it finishes the device processes the request and responds with the appropriate data bytes.

The write cycle on the right is activated when the Interpreter module signals that an input needs to be transmitted to the host. The device can only write if the clock line has been high for 50 microseconds, and so before all write states the diagram indicates this condition must be met.

Upon power up or reset, the FSM enters the RESET state, which then moves into the response states. The purpose of this is to synchronize the device with the host upon device boot-up. Without proper initial synchronization, the device will not be recognized as a mouse input device and inputs will be read by the host as keyboard data (by default).

3. Design

We originally planned to calculate the absolute position with the accelerometer. However, through much testing, we discovered that the change in acceleration due to tilt is much greater then the acceleration caused by hand movements. Therefore, the accelerometer was moved around, the acceleration outputted did not always reflect the acceleration of the hand, but it reflected better the amount of tilt. We did not have the components or time necessary to compensate for this obstacle, so we settled with the tilt mouse.

Different types of filters and decay constants were also considered to minimize noise and clean the signal. The filters intent was to clean out the noise caused by hand vibration. However, the vibration caused by the hand is on the order of 1Hz. A simple averaging technique and disregarding the lower 3 bits were enough to cancel out the noise.

4. Testing and Debugging

For testing the entire system, we tried to modularize the system, so that we can just test each module separately. Each part was tested with simulations on ModelSim (if simulations were possible) and then tested with a set input independent from the other parts.

4.1. Wireless (by Shirley Li)

The wireless portion of the project was relatively self-contained and readily tested. The LED's on the evaluation board were programmed to indicate various activities, such as data reception and packet error. The UART cable could be connected directly to a computer hyperterminal to see the values of different variables or the output of the ADC. The waveforms of the UART output could also be determined by probing the data line with either the oscilloscope or logic analyzer.

4.2. FPGA Serial Interface (by Joseph Cheng)

The Serial Interface was first tested through ModelSim simulations. Through simulation, the right state transitions in the FSM and the right signals were verified. The Serial Interface was ultimately tested with some random signal sent from the wireless kit through the UART cable. The logic analyzer was used to make sure that the signals behave accordingly in the real world. Since each set of data points is inputted at large intervals apart, the logic analyzer was triggered to the start bit of the data points.

4.3. FPGA Calculations (by Joseph Cheng)

The FPGA Calculation modules were also all tested first with ModelSim simulations. The modules were then programmed onto the FPGA.

In order to make this portion independent from the Wireless Kit, and PS/2 Interface, a set of three AD670 analog to digital converters were used to convert the analog output from the accelerometer to an 8-bit digital format. The same FSM from Laboratory 3 was used for this portion to drive the AD670 analog to digital converters. Three AD670 chips were used to convert the three different analog inputs for the three different axes.

The Labkit itself also provided a lot of helpful tools to debug the system. The switches, LEDs, and alphanumeric display were all used to help display the different data points, including the raw accelerometer data, initial acceleration, relative acceleration, velocity, and position. The different displays were especially useful in testing the accelerometer itself, determining the nature of the accelerometer, and fine tuning the calculations for appropriate output.

4.4. FPGA PS/2 Interface (by Matthew Tanwanteng)

The PS/2 Interface was tested and debugged by simulation on ModelSim as done in previous 6.111 labs. Because of the difficulty in emulating a host response, only the write cycle could be thoroughly simulated in ModelSim. Initial testing on the FPGA was limited to making sure transmissions successfully sent the three mouse data packets across a bus with no host device. Testing with an actual host was limited because the computer next to the labkit would not release the data line, and I assumed some other part of the FSM or output was broken or synchronization could only occur on computer restart, which was difficult to test. When it turned out that other computers did not hold the data line like the one connected to the labkit, I was then able to begin testing synchronization with an actual host.

The transmission of mouse updates to a live host introduced some issues that ModelSim testing could not reveal. The host upon receiving a transmission holds the clock low to inhibit further data until it can process the current data, but it sometimes releases the clock for a cycle or brings data low before releasing it. These glitches made it necessary to implement stricter rules on the FSM to make sure it resumed the transmission when the host finished processing and released the clock once again.

The longest issue was synchronization. It took a while before I realized the device has to send the synchronization responses on boot-up without the host sending a request first. Then the exact sequence of responses must be made to following host requests, and leaving a single response out caused a major hold up. That one response caused synchronization to fail, and so despite all other signals being perfectly timed, the host computer still failed to recognize the input as a mouse and nothing worked.

After synchronization succeeded, everything worked perfectly, with the exception of a timing issue which caused duplicate sets of mouse packets to be transmitted and glitches in on-screen mouse movement. That was solved by adding more Interpreter<->FSM communication.

4.5. Integration

Since each component was well separated, integration was not too difficult. The main obstacle was the connection on the wireless kit evaluation board. The power supply was not properly connected to the evaluation board, so the evaluation board kept turning off and on. When one component broke down, the entire system became extremely difficult to debug and to test.

4. Conclusion

The project was completed successfully. The mouse on any computer with a PS/2 port could be controlled wirelessly by tilting the hand. The sensitivity of the cursor in response to hand movement was moderate, giving the user fairly good and smooth control.

The project could be improved and further developed in many ways. The first step would be to implement the project based on absolute positioning of the hand, instead of using acceleration measurements of tilt. This would involve developing an algorithm that effectively compensates for integration drift. This could possibly be done by incorporating proportional or exponential damping factors and bandpass filters in the position calculations. If absolute position capabilities are developed, incorporating mouse button clicks into hand motion would not be difficult. The wireless system could also be improved by increasing the speed of transmission. The rate at which data is being transmitted through the air is currently the limiting factor of the speed of the system. By increasing the transmission rate, more accelerometer data points would be accessible and the movement of the cursor could be more exact.

The project has many fun and practical uses. In addition to using it to perform the regular mouse duties, the project can be used to play mouse-controlled computer games and many other interactive games. Also, since the RF radio has very good range of operation, the mouse would be handy in large lecture hall settings or during presentations.