EE 552: Spatial Mouse
Final Report

April 2, 2001

Gordon Young
Jeff Chan
Jim Wong
Vincci Liu

Declaration of Original Content

The design elements of this project and report are entirely the original work of the authors and have not been submitted for credit in any other course except as follows:

- Theremin oscillator circuit schematic taken from [2]
- PS/2 mouse protocol and operation taken from [3]

_____		_____	
Gordon Young	Date	Jim Wong	Date
_____		_____	
Jeffrey Chan	Date	Vincci Liu	Date

Abstract

The spatial mouse system is to replace a traditional PS/2 mouse by translating hand movement by a user into mouse movement communicated through a PS/2 port to a computer. A “planar mouse pad” consisting of two antennae, one for each dimension, measures an absolute position of the hand. The varying capacitative effects due to the hands’ proximity to the antenna can be exploited by a Schmidt trigger based RC circuit to produce a varying oscillating digital signal. This oscillating signal, along with button click signals, can be processed by an FPGA into mouse movement commands. The FPGA massages the commands into a format that abides by PS/2 timing restrictions and mimics a mouse through the PS/2 port to the PC.

Table of Contents

Achievements

Oscillator Circuit

The operation of the oscillator circuit is fully functional with adequate accuracy to generate a signal frequency that is capable of differentiating between discrete proximities of the hand. Modifications were made to the circuit to adjust for initial instability of the oscillator. In particular, the resistance and capacitor values were adjusted to provide an idle frequency of $\cong$ 8.3 MHz.

Design changes have also been made on the operation of the oscillator. In a previous design decision, a matching oscillator was to be constructed for each user manipulated oscillator. By implementing this configuration, the second oscillator would provide a reference frequency that would be used in a beat generator with the user-controlled signal. The resulting beat frequency would be of lower frequency and thus components such as counters would require fewer resources to implement. This arrangement was simulated and verified using CAD tools but upon implementation in the laboratory, it was found that the user's manipulation of the oscillator would also produce the same effect of the reference oscillator. A decision was then made to revert back to the use of a single oscillator for each axis.

In addition, during the design process various antenna configurations were tested in the lab to allow for a both stability and sensitivity to the user. Experimental observation showed that a steel plate provided superior performance over wire antennas. With the use of a steel plate antenna, a user is able to accurately control the frequency of the oscillator.

Signal Processor

The design and implementation of the signal processor proceed without major difficulty. Upon initialization by the user, the signal processor will sample the signal for both the x and y axis. This frequency will be used as the reference frequency by which all subsequent hand movements will be compared against. The signal processor would then sample ensuing data from the x and y oscillators to produce an op-code that will be used to encode mouse movements to the host PC. This required that the signal processor be able to distinguish distinct proximity of the users hand in reference to the original set location.

The major challenge of designing the signal processor component was to reduce the required number of logic blocks to implement the component on the FPGA. The signal processor contains large counters and registers that are required to process the high frequency signals. As mentioned above, the use of a beat frequency generator was unsuccessfully implemented. To help alleviate the larger registers and counters that will be required, the oscillator frequency of the user interface was reduced from over 20 MHz to approximately 8.3 MHz. The reduced frequency continued to provide adequate accuracy as well as stability.

In the original design, the signal process was to produce a 4-bit op-code to convey mouse movement information to be transmitted to the host PC. The 4-bit op-code did not provide a sufficient number of distinct magnitudes of mouse movements. The op-code on the final design has been expanded to 9-bits to provide a greater range of movement. A 9-bit op-code is chosen because in the PS/2 format of mouse movement packets, a direction bit and 8 magnitude bits specify movement in each axis.

Currently not all 9-bits are used but only utilize 5 distinct magnitudes of both positive and negative mouse movement. The implementation of the full 9 bits is to facilitate future developments.

Mouse Commander

The mouse commander is responsible for conveying mouse movement data to the host PC and mouse initialization upon host PC startup. The op-code provided by the signal processor is used to transmit data packets to the host conveying mouse movements.

The transmission procedure required the Spatial Mouse to meet strenuous timing requirements. The host PC would signal to the Spatial Mouse when it is available to receive data packets. The mouse commander component is responsible for asserting both the data and corresponding clock signals on to the PS/2 interface. The mouse commander must be aware during transmission that the host PC has stalled the process and resume transmission upon being signaled by the PC.

In particular, during transmission data being sent to the host PC is to be changed in between the rising and falling of the PS/2 clock signal. To achieve this, a second clock was generated with a phase shift from the PS/2 clock in order to allow for this. Using both simulations and experimentation in the lab, timing requirements for transmission was successfully met. The mouse commander is able to accurately control mouse movement to the host PC.

Problems were encountered during the interface demonstration where the mouse commander on isolated incidents was observed to send extra data mouse packets to the host PC. This resulted in the lost of control of the mouse pointer. The cause of this phenomenon was suspected to be due to insufficient debouncing of the push button. After investigation, a decision was made to implement the mouse commander in the overall Spatial Mouse design without alteration to the design of the mouse commander. Using simulations and extensive lab investigation, the phenomenon as described above was no longer observed.

The design of the mouse command required experimentation in the lab with a traditional PS/2 mouse to investigate actual mouse behavior. Upon investigation, the mouse upon power produces an initialization sequence to the host computer. This sequence is required to enable the mouse. This initialization routine was designed and verified using simulations. Upon implementation on the FPGA, the Spatial Mouse is unable to reproduce the same initialization routine. Work is continuing to complete this aspect of the design and currently requires the use of a tradition mouse for initialization.

Spatial Mouse

The design of the overall Spatial Mouse has been verified with both simulation and interface to a host PC with exception to the initialization routine. A two dimensional mouse has been successfully implemented on the FPGA. The user is able to accurately control mouse movement in both axis.

Description of Operation

This document outlines the development of the Spatial Mouse with a PS/2 communication port. Our design is based upon a musical instrument called a Theremin. A physicist named Leon Theremin invented the Theremin in 1919. This musical instrument allows a user to interact with the instrument without actually coming in contact with it. Instead, the user manipulates the instrument through external antennas. The Theremin had two external antennas through which the user was able to control both the pitch and volume of the output by positioning his/her hand near the two antennas.

The underlying mechanism by which a Theremin device works is based on the manipulation of a frequency oscillator. An oscillator circuit is implemented but with an antenna. The oscillatory frequency is a function of the resistance and capacitance of the circuit. When the user interacts with the external antenna, the capacitance between the users hand and the antenna is varied as the proximity of the hand to the antenna changes. As a result the circuit's oscillatory frequency changes correspondingly.

The Spatial Mouse system will consist of two protruding antennae through which the user will manipulate the mouse. The proximity of the users hand to each antenna will produce a corresponding signal in a range of analog frequencies. The analog signals will be representative of mouse movements in the x and y-axis.

The signals will be used as the input to the FPGA. Thus there will be two signals to be processed by the FPGA to represent movement in the x and y-axis respectively. The FPGA will be responsible for determining the frequency of the input signal from the external oscillator circuit. Once the frequency is extracted from the input signals, they will be translated to represent mouse movements. The movements of the mouse will be communicated to a PC through the PS/2 port that is available on the FPGA board. It is our goal to demonstrate the functionality of the Spatial Mouse through the aid of a visual application such as a video game.

Mouse Operation:

The spatial mouse is operated through the use of a controller that the user will have in his/her hand while moving in the predefined area. On the controller there are 3 buttons, mouse enable, left-mouse button and right mouse button. Through this controller, the user initializes and enables the Spatial Mouse. The Spatial Mouse is first initiated by having the user place their hand in the neutral position of the "mouse pad" and then depress the mouse enable button. The mouse-enable button initializes the mouse and sets the reference point which subsequent hand positions are measured. The user will have to depress the enable button while using the mouse. Once the enable button is released, the users hand movements will no longer control the Spatial Mouse, thus this allows the user to free his hand for other tasks without manipulating the Spatial Mouse. This is analogous to allowing the user to take their hand off a real mouse. If the user wants to reengage control of the mouse, he/she will again repeat the initialization routine by placing his/her hand in the neutral position and depressing the enable button.

Once the Spatial Mouse is initialized and the enable button is still depressed, the user is able to control the movement of the mouse by positioning his/her hand in the defined two-dimensional "mouse pad". The mouse is operated by absolute position of the users hand within the "mouse pad" which is communicated to the PC through the PS/2 port.

Currently, our design allow for the mouse movement in 8 directions and the use of the left and right mouse buttons.

Additional features that we plan to implement when the basic goals are met and time permitting include:

- Refine mouse movement to more closed imitate the full movement of the mouse seen in commercial available products.
- Addition of axis to design to allow greater control and dimensions of mouse movement.
- Produce an audible tone that will give feed back to user.

Analog Oscillator Circuit:

See attached schematics in the Appendix. The external analog circuitry includes a Schmitt trigger along with a resistor and capacitor. The antenna will provide a variable capacitance, which will alter the frequency of the output on the Schmitt trigger.

PS/2 Protocol:

The mouse and computer communicate via the PS/2 communication. The PS/2 serial communication is asynchronous and includes 2 signals, clock and data. The two signals are bi-directional and are open drain. Normally the signals are held high by a pull up resistor and the signal can be pulled low by either the computer host or Spatial Mouse.

The computer host always has priority and controls the state of the data and clock buses. The bus can be in 3 possible states; idle, inhibit or request to send.

Bus State	Description
Idle	Both the clock and data lines are allowed to float high. During this state, the mouse is free to transmit data packets to the computer when set.
Inhibit	The clock is held low by the host. During this state, the Spatial Mouse cannot transmit data packets to the host.
Request to send	The data line is held low and the clock is allowed to float high. During this state, the Spatial Mouse must get ready to receive commands from the host.

Table 1. Spatial Mouse Bus States.

Format of Data Packets:

The communication between the host computer and Spatial Mouse is handled through data packets of 11 bits long. Each communication packet includes a start bit, 8-bit data payload, an odd parity bit and a stop bit.

Bit 1	Bit 2	Bit 3	Bit 4	Bit 5	Bit 6	Bit 7	Bit 8	Bit 9	Bit 10	Bit11
Start bit 0	Data bit 1 LSB	Data bit 2	Data bit 3	Data bit 4	Data bit 5	Data bit 6	Data bit 7	Data bit 8 MSB	Odd Parity Bit	Stop Bit 1

Figure 1. PS/2 Data Packet Format

Data Transmission to Host

While the clock and data bus is in the idle state, the Spatial Mouse is allowed to send data packets to the host PC communicating mouse movement and button status.

The Spatial Mouse signals the start of a transmission packet by pulling the clock line low and then allowing the clock line to float back high 11 times. The host computer will then sample the data line 11 times on the falling edge of the clock. The Spatial Mouse will shift out the transmission bit when the clock signal is high.

The data packets to encode mouse movement and button status is transmitted in 3 consecutive packages of 11-bits that follow

	D7	D6	D5	D4	D3	D2	D1	D0
1st	YV	XV	YS	XS	1	0	R	L
2nd	X7	X6	X5	X4	X3	X2	X1	X0
3rd	Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0

L	Left Button State (1 = pressed down)
R	Right Button State (1 = pressed down)
X0-X7	Movement in the X direction
Y0-Y7	Movement in the Y direction
XS	Direction of the movement in the X axis (1 = UP)
YS	Direction of the movement in the Y axis (1 = LEFT)
XV,YV	Overflow of the movement data bits (1 = overflow has occurred)

Figure 2. Mouse movement/button package format.

Design Details

The implementation is done through design, implementation of individual components. Throughout the design process, communication specifications linking each component block are determined to ensure correct functionality in the final design. The main components of the Spatial Mouse design are outlined in Figure 3.

Hand-Held Clicker

- Active low trigger and buttons.
- The *activate trigger* will send a continuous logic low signal while depressed. This signal indicates to the digital circuit that the mouse enabled by the user (i.e. when trigger is depressed, the mouse is being held and when trigger is released, the mouse is being released).
- The *right thumb button* will send a logic low signal while depressed. This button is equivalent to a right mouse button.
- The *left thumb button* will send a logic low signal while depressed. This button is equivalent to a left mouse button.

Planar Spatial Mouse Pad

- *X-antenna* and *Y-antenna* set up in a planar space perpendicular to the horizon.
- Each antenna senses the proximity of the user's hand by capturing the capacitance resulting between the hand and antenna. The varying capacitance due to movement of the hand varies the time constant of the oscillating circuit to in turn output oscillations with frequencies proportional to hand proximity (in each dimension).
- Due to possible interference of the oscillator circuits, each oscillator representing the X and Y-axis will be alternately switched on and sampled. The control signals to enable each oscillatory is produced by the processor.

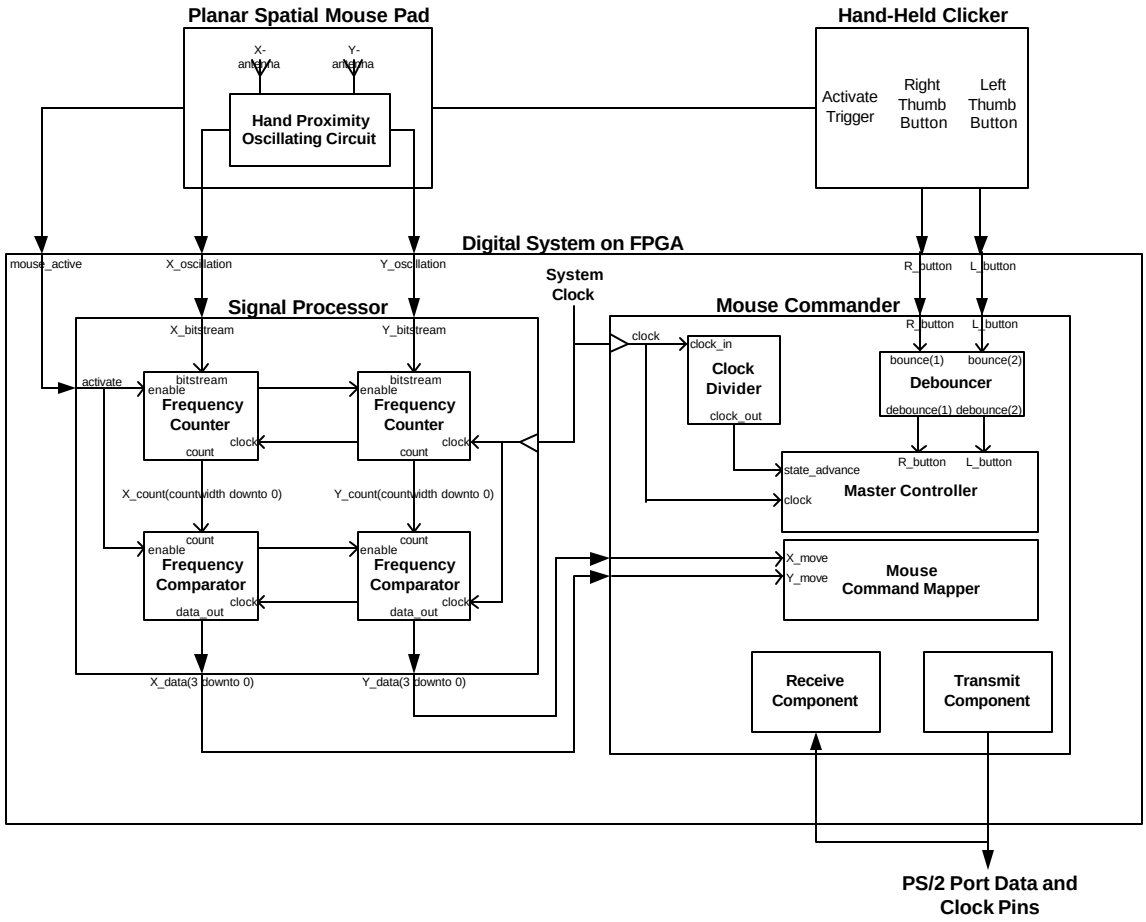


Figure 3. Spatial Mouse Components.

Signal Processor

A detailed block diagram of the signal process for one axis is illustrated in Figure 4.

- Frequency Counter
  - A synchronous counter generates a sampling period T, resetting every T.
  - An asynchronous counter counts the number of oscillations in incoming bitstream within period T.
  - A register holds and outputs *count*, the number of oscillation counts in last period T. The register updates *count* every period T as long as the *enable* is on.

- Frequency Comparator
    - When *enable* initially goes “on”, *count* is stored in a register (called calibration register) where successive counts will be referenced to this stored value. If *enable* goes “off”, the calibration register will reset and a new value of *count* will be stored the next time *enable* goes “on”.
    - A comparator circuit subtracts each successive *count* from the value stored in the calibration register every period T. The determined frequency of subsequent samples is compared against set ranges, which determine direction and degree of movement. These ranges are set as generic parameters and can be altered in the VHDL code to adjust for varying antenna behavior. The comparator produces a corresponding op-code.
    - *Op-code* contains 9 bits of information: a direction bit, and 8 bits to express levels of movement intensity. The direction bit maps to the sign of the difference from the comparator circuit. The intensity bits map to a range table from the comparator circuit.

Figure 4. Frequency Sampler Block Representation.

Mouse Commander

- Dual Phase Clock
    - The dual_phase_clock generates two clock signals that are ¼period out of phase from each other.
    - A 25.0KHz clock is fed into the dual_phase_clock to yield two 12.5KHz clock signals: *clock_lead* and *clock_lag*. To generate the *clock_lead* signal, a 25.0KHz input signal is used to clock a rising edge triggered d-flipflop with inverted q feedback to d. To generate the *clock_lag* signal, a 25.0KHz input signal is used to clock a falling edge triggered d-flipflop with inverted q feedback to d. Upon clock reset, the lead-lag relationship is consistently maintained by allowing the flipflops to synchronously reset only on the falling edge of the external clock signal (i.e. if external reset asserted high, *phase_reset* goes high only falling edge of clock). The frequency and phase relationships between the three clocks are illustrated below.

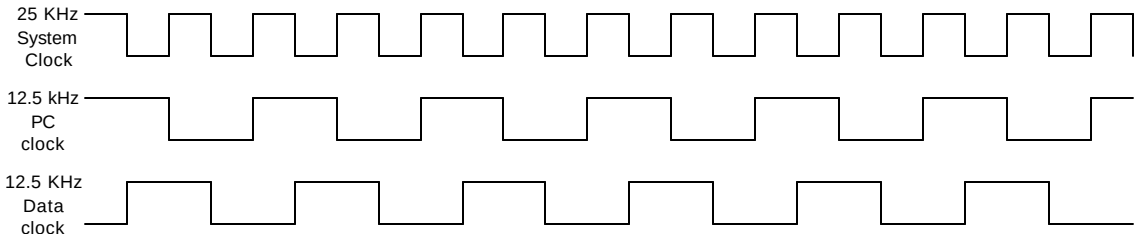


Figure 5. Dual phase clock timing.

Figure 6. Dual phase clock.

- Master Controller
    - The timing controller massages the mouse movement and button signals into serial transmission that adheres to the PS/2 communication protocol. The controller is responsible for the timing and coordination of transmission and reception of data.

- Mouse Command Mapper
    - The mapper is responsible for receiving the 9-bit op-code from the signal processor unit and translating it into PS/2 mouse data packets.

- Transmitter
    - The mapper will produce the required data packets to be transmitted to the host. The transmitter is then responsible for putting the data bits onto the data bus for the host computer to read. The transmitter also initiates timing and assertion of the clock bus. A shift register is used to serially transmit the required data packets.

- Tri-state bus
    - The bus and data lines are bi-directional and will have to assert a high impedance for a logic high and a '0' for logic low. The bi-directional clock and data bus will require a tristate output. A pull up in the host is will maintain a logic high if not driven.

- Data Transmission Component
    - Built structurally from the transmit_contoller and Tx_shift_register, the transmitter works hand-in-hand with the dual_phase_clock to transmit serial data to a PC while adhering to the PS/2 timing standards. The transmit_controller utilizes the *data_clock* signal generated by the dual_phase_clock to properly synchronize the serial data with the 12.5 KHz *PC_clock* also generated by the dual_phase_clock. The Tx_shift_register shifts the serial data out on every pulse of the 12.5KHz *data_clock* but is only allowed to do so when enabled by the transmit_contoller.

The following sections outline the mouse to PC data transmission in detail.

Figure 7. PS/2 transmit component.

Tx_shift_register

The Tx_shift_register is an 11-bit shift register with parallel loading. When *load* is asserted high, the register is parallel loaded at *port d* with 11 bits (1 PS/2 packet). When *shift* is pulsed, one bit is outputted at port *q* on the pulse's rising edge while *enable_shift* must be maintained high for any shifting to take place.

The shift_out_register component is described behaviorally. Initially the *load* signal is asserted high to load the register with input at *port d* so that the index value of the 11 bit wide input is set to zero pointing to the least significant bit. An internal 11 bit wide variable is used to hold the incoming data so that the data is not lost. Then once the register detects a rising edge of the *shift* signal it checks if the register has been enabled. If the "shift enable" signal is high, then the register proceeds to shift out one bit at a time starting at bit(0). After each shift, the index counter for the input data is incremented by one so that successive bits can be outputted.

Transmit_Controller

The transmit_controller is required for ensuring proper PS/2 timing of each data frame sent from the mouse to the PC. The transmission timing restrictions for the PS/2 protocol are shown below.

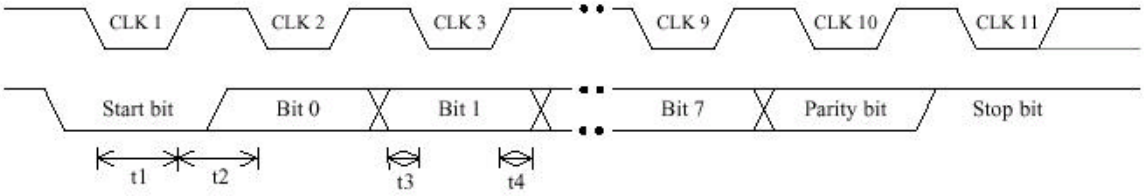


Figure 8. Transmission timing.

TIMING PARAMETER DESCRIPTION	MIN. TIME	MAX. TIME
t1 DURATION OF CLK LOW	30 µSEC	50 µSEC
t2 DURATION OF CLK HIGH	30 µSEC	50 µSEC
t3 VALID DATA BEFORE FALLING EDGE OF CLOCK	5 µSEC	
t3 VALID DATA AFTER RISING EDGE OF CLOCK		5 µSEC

The T1 and T2 timing criteria can be met by using a 12.5 KHz (80us) *PC_clock*. The other timing parameters, T3 and T4, can be satisfied if a new data bit is sent a ¼period after each rising edge of the *PC_clock*. Thus, a *data_clock* that lags the *PC_clock* by a ¼period could be used to trigger each shift of the serial data. The diagram below illustrates the phase relationships between the *PC_clock*, the *data_clock* and the serially transmitted data to be shift-triggered by the *data_clock*.

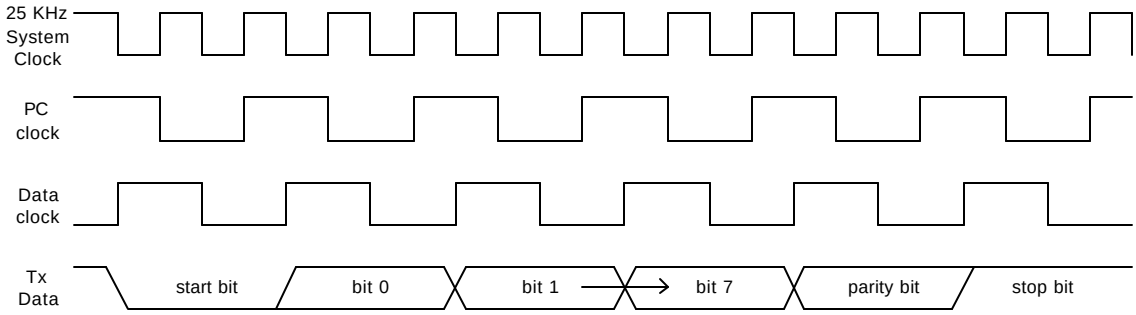


Figure 9. Data transmission process timing.

As shown in the previous timing diagram, the mouse initiates the start bit of a byte transmission by firstly dropping the floating data line and then dropping the floating clock line. Similarly, termination requires that the mouse firstly allows the data line to float high (as a stop bit) and then allow the clock line to float high after the 11th clock pulse. Governing proper initiation and termination, as well as the correct number of bits transmitted is represented in the following state diagram for the transmit_controller.

Figure 10. Transmit component state machine.

For the following description, please refer to the mouse to PC transmission block diagram as well as the state diagram above. The transmit_controller is advanced on the rising edges of the 25KHz system clock but also needs to look at the 12.5 KHz *data_clock* to ensure proper initiation of data transmission. Transmission of an entire byte requires that the *Tx_enable* is asserted high for the entire duration byte transmission. In the start state, the controller disables shifting in the Tx_shift_register and does not allow a clock to be driven onto the PC-mouse clock bus. The controller remains in the start state until *Tx_enable* is asserted high. When transmission is

enabled, the controller moves into the load_data state where it sends a pulse to the Tx_shift_register to load next byte packet. While in this state, the controller checks the *data_clock*. If data_clock is low, the controller advances to the enable_shift state on the next system clock pulse, but if the data_clock is high, the controller will stall in the load_data state for one more system clock period. This timing maneuver ensures that the start bit will be asserted before the PC-mouse clock bus is dropped low from its floating high. In the enable_shift state, the controller starts asserting a high on *enable_shift* of the Tx_shift_register making it sensitive to the *data_clock* shift signal. Before the next state is reached, the start bit will have been asserted on the PC-mouse data bus. On the next system clock pulse, the controller advances to the drive_PC_clock state where starts enabling *PC_clock* to write onto the PC-mouse clock bus. Before the next state is reached, the *PC_clock* will have driven the PC-mouse clock bus low for the first time. The controller next enters the shift state where it remains in this state until all 11 bits have be transmitted. A counter (*shift_count*) that increments on each state advance indicates to the controller when to enter the stop state. In the stop state, the *shift_enable* and *data_clock* are deactivated and a done signal goes high to signify to a higher level entity that the byte transmission is complete.

- Receive Command Component
    - The Receive component is used during the initialization routine. Actual data is not captured but used to provide timing of the clock so the host can send an initialization command. Since the initialization routine requires a constant response of an acknowledge, the host command is not stored.
    - In the current design of the Spatial Mouse, the ability to receive host commands has been disabled. It was observed that the host does not send commands to the mouse during normal operation. The receiver has been included in the design to facilitate future receiving capabilities if required.

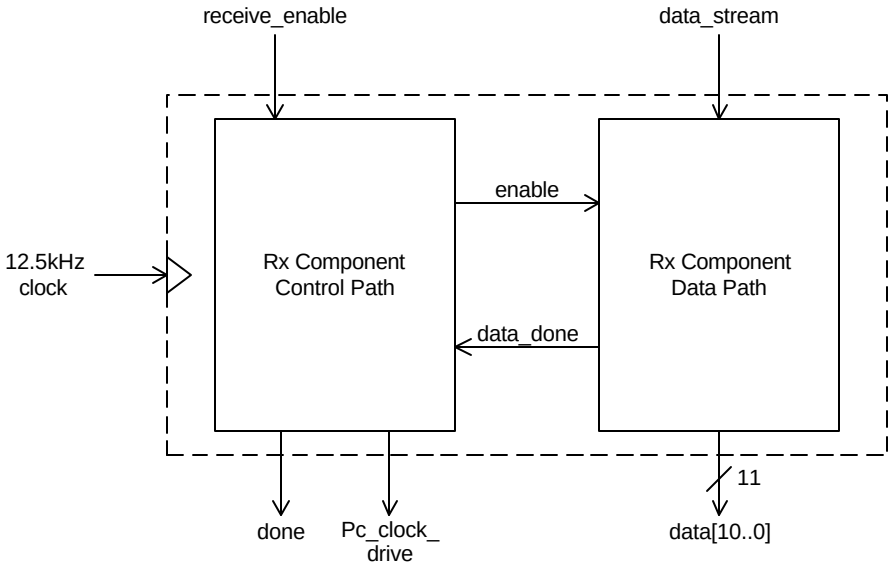


Figure 11. PS/2 receive component.

The receive process is initiated by holding the *receive_enable* high. The *receive_enable* must be held high for the duration of the receive process. Upon detection of a logic '0' on the *receive_enable* signal prior to receiving the full 11-bit packet, the process will abort and no data will result.

Upon asserting the *receive_enable* signal high for the duration of 11 clock cycles, the done signal will assert '1' and the 11-bit data package will be available on the data output. Also, during the 11

clock cycles of the receive state, the *Pc_clock_drive* will be asserted high. The *Pc_clock_drive* signal when asserted will enable an output tri-state buffer to drive the PS/2 clock bus.

The following state machine summarizes the control of the receive component.

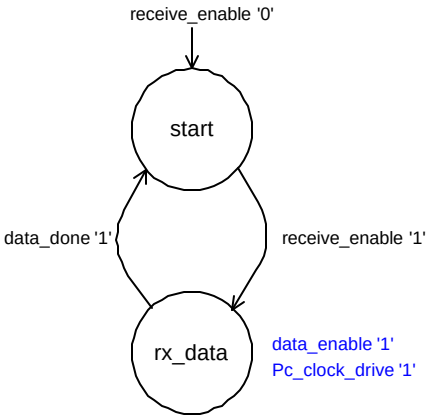


Figure 12. Receive component controller.

The data path of the receive component is comprised of a shift register with an *enable* input. Also a counter is used to signal the reception of 11-bits of data. The purpose of the *done* signal is to provide timing for a higher level controller to regulate the reception of commands from the host computer and then the transmit the corresponding response.

Data Sheet

The Spatial Mouse is designed for a 25.175 MHz crystal oscillator on the UP-1 board.

The external oscillator circuitry is designed for oscillation frequency 8.3 MHz.

The processor component is designed for optimal performance when oscillator frequency is in the range of 5-20 MHz.

The following is the pin assignment for the Spatial Mouse implementation on UP1 board.

Name	Pin Number	Pin Type	Use of Pin
Reset	FLEX_PB1 – pin 28	Input	Use of one push button on the board as it resets the movement tracker so user can calibrate system to their hand movement.
X_Data from oscillator	70	Input	Input data from oscillator regarding movement along x-axis (position & magnitude).
X oscillator enable	15	Output	Signal to enable x-axis oscillator.
Y_Data from oscillator	75	Input	Input data from oscillator regarding movement along y-axis (position & magnitude)
Y oscillator enable	17	Output	Signal to enable y-axis oscillator.
Mouse Active Button	73	Input	Enables system to begin processing frequency changes and translates changes into on-screen movement.
Mouse Clock	100	In/Output Bi-dir	Sends the clock pulse of the mouse along the PS/2 line to communicate with the PS/2 port of the computer.
Mouse Data	98	In/Output Bi-dir	Sends the bitstream containing the data regarding on-screen movement of the cursor to the computer.
Mouse Left Button	75	Input	Input data if user clicks the “left button” to indicate an action.
Mouse Right Button	78	Input	Input data if user clicks the “right” to indicate an action.

Table 2. Pin Assignment for Flex10k

FPGA Resource Requirements

In order to estimate the number of resources required to implement the entire design, a complete compilation of the Spatial Mouse design was completed. The results are shown in the table below. Logic blocks used for internal signals and intermediate registers used in the overall Spatial Mouse design.

Component	# Logic Blocks
Signal Processor	612
Mouse Commander	236
Mouse Mapper	27
Clock ticked	17
Spatial Mouse internal signals	72
Spatial Mouse Total	**964**

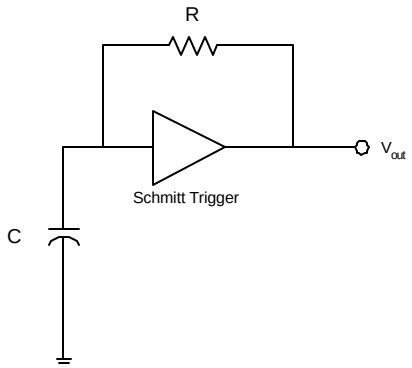
Figure 13. Resource usage for final design compilation

Experimentation Results

Analog Oscillator:

Two different oscillator configurations were implemented in the laboratory. The two oscillator configurations are shown below.

RC Schmitt Trigger Oscillator



This simple RC circuit produces a periodic square waveform. The frequency of the output waveform is controlled by the charging and discharging time of the capacitor.

The period of the oscillation can be approximated by:

$$T = RC \ln\left(\frac{V_{ss} - V_i}{V_{ss} - V_f}\right) = 1..0586RC \approx RC$$

Experimental results recorded in the lab confirm that the oscillation period is described by the above equation.

Investigation of user interaction with the circuit was also done through an antenna. Initial results showed little change to oscillator frequency as a user approached the antenna with their hand. Subsequent investigation showed that the initial oscillator frequency greatly affects the amount of change a user is able to induce on the output frequency.

As the initial idle output frequency of the oscillator is increased, the user is able to exert a great degree of change in the oscillator frequency. Initial oscillator frequencies in the range of 1-50kHz resulted in negligible changes in frequency. As the frequency was further increased by changing capacitor and resistor values, it was found that the user is able to change the output frequency by approximately 10% when oscillator frequency is in the range of 10-25MHz. During the investigation, it was also found that the oscillation frequency would drift in a small range without user interaction due to environmental noise and factors such as temperature. In the design of the frequency counter, we will provide a buffer range in which the frequency may drift without translating to mouse movement. This actual is to be determined with further investigation.

After thorough investigation into suitable antennas, flat metallic plates have been found to give the most stable and responsive oscillations from user interface.

An oscillator using 555-timer was also implemented and found to have the same characteristics as the RC Schmitt trigger circuit. Due to the larger number of discrete components required implementing the configuration in comparison to the Schmitt trigger, the Schmitt trigger has been chosen as the oscillatory configuration.

Simultaneous Use of X and Y-axis oscillator

Initial implementation utilizing both x and y axis simultaneously resulted in cross interference between the oscillators. Manipulation of one oscillator would al cause parallel affects on the other oscillator. Thus to circumvent this problem, each oscillator would be alternately switched on and off.

The NAND gate Schmitt triggers provide two inputs. One would be used for the antenna and the other input would be used for an enable signal. The processor unit would produce complementary enable signals so each oscillator would be operated and sampled in alternate cycles.

PS/2 Computer Interface:

Upon completion of the transmitter and timing controller component, the code was loaded onto the FPGA and a subsequently connected to the PS/2 port on a host PC. A scope was attached to both the clock and data lines to observe waveforms.

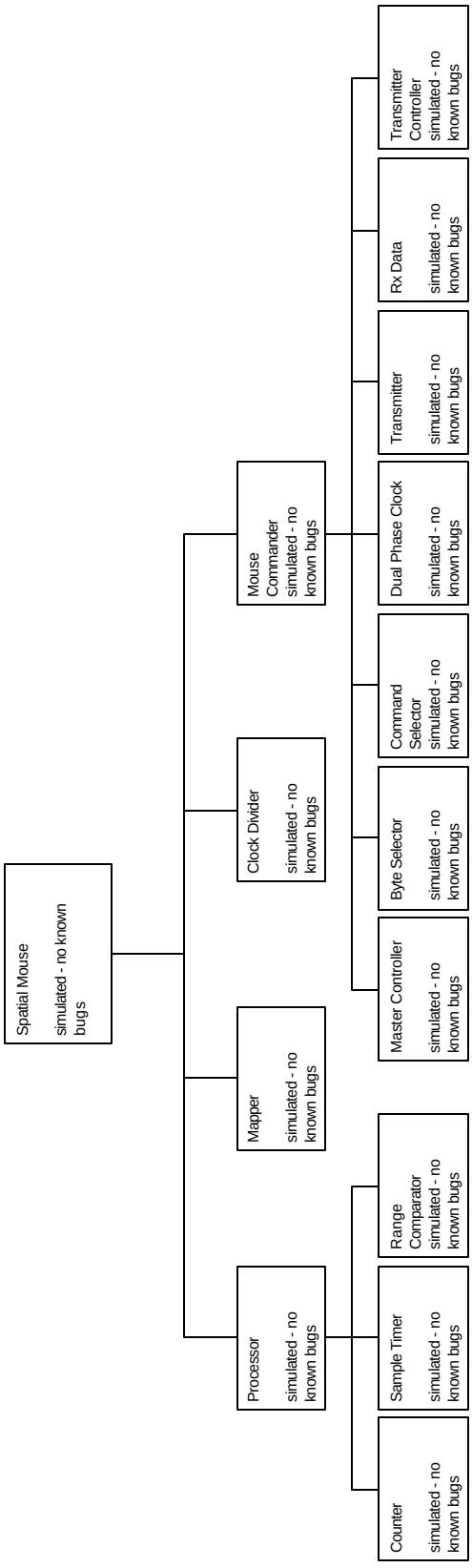
Up to this point, the entire design process was based upon documentation of the PS/2 protocol and lacked solid data. This allowed us to observe and confirm the data that was presented in the documents.

The transmitter was loaded with static data package that would be sent to the computer upon the push of a button on the UP-1 board. Initial trials failed to communicate to the host computer. Investigation with the scope on the data and clock lines indicated it was in the idle state. This meant that the computer was ready to accept data packets but our attempts have failed. Further investigation showed that the computer would inhibit the bus for a short period following the transmission of each of the 3 data packets.

The timing controller was altered to account for this behavior and on subsequent tests, we were able to successfully manipulate the mouse cursor. Although the data sent to the host was static and resulted in the repetition of the set movement, communication to the host was successful.

In the PS/2 protocol documentation, very complex communication between the host to the device was outlined, requiring the device to receive and process commands. Responses would have to be sent back to the host based on the commands. This process was very complex and convoluted, requiring much of our design time. But the interface investigation that was completed above, our observations did not show that the computer communicated to the device except during initial startup. Further investigation will be completed and if warranted, the decoder component and response generation of the mouse would be greatly reduced.

Design Hierarchy



References

[1] Dr. Duncan Elliott. **EE552 Course Home Page. 2001**.
http://www.ee.ualberta.ca/~elliott/ee552/

[2] Lindsay Reid & Brendan Dougan **Digital Theremin Circuit**. Internet. 2000.
http://www.physics.gla.ac.uk/~kskeldon/PubSci/exhibits/E9/

[3] Nicola Asuni **Hardware Resources** Internet. 2000. http://www.technick.net

[4] Max **Maxies Pages**. Internet. 2000. http://www.maxiespages.com/

[5] Altera Corp. **UP-1 & Flex10K Documentation/Datasheets** Internet. 2000.
http://www.ee.ualberta.ca/~elliott/ee552/AlteraDoc/

[6] Synaptics, Inc. **Synaptics TouchPad Interfacing Guild**. Internet. 1998.
http://www.synaptics.com/decaf/utilities/tp-intf2-4.PDF

Appendix

Design Verification Index

Component	Test Description	Result
Mouse Commander	Serial transmission of mouse movement bytes to PC.	Verified with simulation. Verified with interface with host computer.
Transmitter Component	Timing and transmission of static mouse movement data to the host PC. Utilization of transmit component and bi-directional port.	Verified with simulation. Verified with interface with host computer.
Receiver component	The functionality to receive and signal the completion of single 11-bit package of data.	Verified with simulation.

Mouse Commander

The mouse commander is responsible for the transmission of data packets to the host PC. The signal processor generates a 9-bit op-code, which is used by the mouse commander to transmit the corresponding 3 mouse movement bytes to the PC.

The mouse commander is responsible for the timing and continually checks the status of the data and clock lines. After the transmission of a byte, the host computer will inhibit the mouse from transmitting the next byte to allow for processing. The mouse commander initiates the transmission of the next byte once the host PC returns to the idle state.

In the test cases, the mouse commander is provided an op-code and transmission is initiated. The bus is inhibited after the transmission of each data packet. Upon returning the bus to the idle state, the mouse commander proceeds to transmit the remaining data packets.

The test cases show that the mouse commander properly initiates the transmission of data packets and correctly transmits the 3 data packets to the host PC.

Transmitter Component:

The transmitter component is responsible for asserting both the data and clock lines during the transmission of data to the computer. As outlined in the PS/2 protocol, the communication of data is through data packages of 11 bytes.

The first test case shows the correct functionality of the transmitter component during operation. The *start_button* signal is held low to initiate transmission of three data packets. The data is asserted on the *PC_data_bus* starting with the least significant bit. The sequence of data can be verified to be that of the *Tx_data_in*. Also, *tx_done* is asserted high by the transmitter to signal the successful transmission of the data packet.

Following the transmission of the first data packet, the PC_clock_bus is held low to simulate the PC. The PC will inhibit the PC_clock_bus to process the information. When the PC is ready to receive the next packet, the PC_clock_bus is allowed to float high. The waveform shows the subsequent transmission of the remaining two data packets. Prior to the transmission of a packet, the PC_clock_bus is verified to be high.

In the simulation, the data and clock buses are either asserted ‘0’ or ‘Z’, as both buses will float to a high logic when high impedance through a pull up in the computer.

Receiver Component:

The receiver component is to receive a single 11-bit data packet from a serial data-stream. To initiate the receiving process, the *receive_enable* signal is held high. This signal must be held high for the duration of the process. In the event that the *receive_enable* signal is pulled low during the receive state, the reception process will restart, erasing all data received so far. Upon the completion of 11 clock cycles, the 11 byte data packet will be available on the output with the done signal to logic '1'.

The testing of the receiver was done by inputting a pattern of data on the input data stream and then asserting the *receive_enable* signal for various lengths of time to simulate all working conditions of the receive component.

The first test case is holding receive_enable high for 11 clock cycles and a 11-bit data package should be available on the output port. Also the done signal should go high.

The second test case is illustrating the receive_enable signal held high for less than 11 clock cycles.

The third test case is a repeat of the first test case but also used to show that the system restarts and must receive 11 clock cycles in order to produce the next data packet.

The waveform from the testing can be found on the following page. From the waveform simulation, it can be verified that the serial receiving component is fully function.

Optimization for speed is not a concern with this design as the rate at which data transmission occurs is set at a predefined value. The clock in the simulation is 25kHz and is representative of actual clock rate used in the PS/2 protocol.

Spatial Mouse:

The test bench tests the functionality of the overall Spatial Mouse design. The operation of a Spatial Mouse is simulated and tested in the following setup.

The test bench comprises of 7 possible input frequencies ranging from 5 MHz to 7 MHz. The frequencies to the x and y input data streams are controlled with an 8-to-1 Multiplexer. Initially, both the x and y input frequencies are set to 6 MHz. The Spatial Mouse is then activated and allowed to initialize.

Following the initialization, the x and y input signals are independently varied and the resulting transmission of data packets is verified to expected results. The test bench includes testing of a range of frequencies to produce both positive and negative mouse movements in both x and y axis. In addition, the Spatial Mouse is tested with frequencies in both extremes where the Spatial Mouse is expected to remain motionless.

Signal Processor

The processor receives an oscillating signal form the external oscillator (x_bitstream and y_bitstream) and translates this frequency into and opcode used by the mouse commander. An activate signal sets the most recently sampled frequency as the reference frequency when first asserted and for the remaining duration of its assertion, opcodes are generated by comparison to that reference frequency. A nine bit opcode is generated with the most significant bit indicating whether the current frequency is negative or positive relative to the reference frequency and the remaining eight bits reflecting the magnitude of difference between the current and reference frequencies.

The processor testbench simulates various bitstream frequencies that would otherwise come form the external oscillator and verifies that the correct opcodes are generated. The activate signal was asserted two times in the testbench to show two separate calibrations of reference frequency. Upon assertion of the activate signal, the frequencies were varied to show that different opcodes are generated appropriately according to the difference range between current and reference frequencies. The clock and various bitstream frequency signals were not included on the waveform since they are merely black bars of high frequency that absorb much processing resourse to display. Rather, the frequency select signals (x_select and y_select) are shown to reflect the current frequency on the bitstream inputs.

VHDL Code Index

Component	Status
Signal Processor	
Sample_timer.vhd	Compiled
Range_comparator.vhd	Compiled
Counter.vhd	Compiled
Processor.vhd	Compiled
Processor_pkg.vhd	Compiled
Mouse Commander	
Command_selector.vhd	Compiled
Byte_selector.vhd	Compiled
Master_controller.vhd	Compiled
Mouse_commander.vhd	Compiled
Rx_data.vhd	Compiled
Transmitter.vhd	Compiled
Inverter.vhd	Compiled
Inverter_with_enable.vhd	Compiled
Bidir.vhd	Compiled
Shift_in_reg.vhd	Compiled
Shift_out_reg.vhd	Compiled
Transmitter_controller.vhd	Compiled
Mouse_commander_pkg.vhd	Compiled
Transmitter_pkg.vhd	Compiled
Dual_phase_clock.vhd	Compiled
Dual_phase_clock_pkg.vhd	Compiled
Spatial Mouse	
Spatial_Mouse.vhd	Compiled
Spatial_Mouse_pkg.vhd	Compiled
Mouse_command_mapper.vhd	Compiled
Clock_ticked.vhd	Compiled
Test Benches	
Spatial_mouse_test.vhd	Compiled
Processor_test.vhd	Compiled

```
-----------------------------------------------------------------
-- sample_timer.vhd :
--
-- This component pulses the output signal, “enable”, high
-- for 1 clock cycle out of every “timer_count” (as specified
-- by a generic) clock cycles.
--
-- Role in Spatial Mouse Processor:
--  - clears sample timer by pulsing the reset of sample timer
--  - triggers range comparator computation by pulsing enable
-----------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;

entity sample_timer is
generic (timer_count : positive := 250);
port(
   clock    : in std_logic;
   reset    : in std_logic;
   enable   : out std_logic);
end sample_timer;

architecture behavioural of sample_timer is
begin

process
variable count : natural range 0 to timer_count-1;
begin
   wait until rising_edge(clock);
      if reset = '1' then
         count := 0;
         enable <= '0';
      else
         count := count + 1;
         if count = 0 then
            enable <= '1';
         else
            enable <= '0';
         end if;
      end if;
end process;

end behavioural;
```

```
-----------------------------------------------------------------
-- range_comparator.vhd :
--
-- This component accepts a number (i.e. count) as a calibration
-- count that automatically falls into range_zero when
-- initially enabled.  For the remaining duration of enable
-- assertion, each successive count clocked in is compared to
-- the calibration count and is placed into a range relative to
-- the calibration count.  The range widths are specified as
-- generics.
--
-- Role in Spatial Mouse Processor:  places sampled frequencies
-- (counts) into ranges as specified by easily modifiable generics.
-- Allows for tweaking of digital design to external oscillator
-- performance.
-----------------------------------------------------------------


library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity range_comparator is
generic (
   datawidth   : positive := 10;
   range_zero  : positive := 100;
   range_minus2: positive := 100;
   range_minus1: positive := 100;
   range_plus1 : positive := 100;
   range_plus2 : positive := 100);
port(
   clock       : in std_logic;
   enable      : in std_logic;
   reset       : in std_logic;
   frequency   : in std_logic_vector(datawidth-1 downto 0);
   data_out    : out std_logic_vector(8 downto 0));
end range_comparator;

architecture behaviour of range_comparator is
begin

process

variable calibrated     : std_logic;
variable zero_frequency : std_logic_vector(datawidth-1 downto 0);
variable minus2_min     : std_logic_vector(datawidth-1 downto 0);
variable minus2_max     : std_logic_vector(datawidth-1 downto 0);
variable minus1_max     : std_logic_vector(datawidth-1 downto 0);
variable plus1_min      : std_logic_vector(datawidth-1 downto 0);
variable plus2_min      : std_logic_vector(datawidth-1 downto 0);
variable plus2_max      : std_logic_vector(datawidth-1 downto 0);

begin
   if reset = '1' then
      data_out <= "000000000";
```

```
      calibrated := '0';
   elsif rising_edge(clock) then
      if enable = '1' then
         if calibrated = '0' then
            zero_frequency := frequency;
            calibrated := '1';
         else
            plus2_max := zero_frequency + (range_zero/2) + range_plus1
                         + range_plus2;
            plus2_min := zero_frequency + (range_zero/2) + range_plus1;
            plus1_min := zero_frequency + (range_zero/2);
            minus1_max := zero_frequency - (range_zero/2);
            minus2_max := zero_frequency - (range_zero/2)
                         - range_minus1;
            minus2_min := zero_frequency - (range_zero/2)
                         - range_minus1 - range_minus2;

            if frequency > plus2_max then       -- out of range
               data_out <= "000000000";
            elsif frequency > plus2_min then    -- in range plus2
               data_out <= "000000010";
            elsif frequency > plus1_min then    -- in range plus1
               data_out <= "000000001";
            elsif frequency > minus1_max then   -- in range zero
               data_out <= "000000000";
            elsif frequency > minus2_max then   -- in range minus1
               data_out <= "100000001";
            elsif frequency > minus2_min then   -- in range minus2
               data_out <= "100000010";
            elsif frequency <= minus2_min then  -- below range set
               data_out <= "000000000";
            end if;
         end if;
      else
         calibrated := '0';
         data_out <= "000000000";
      end if;
   end if;

end process;

end behaviour;
```

```
------------------------------------------------------------------
-- counter.vhd :
--
-- A synchronous counter with asynchrounous clear
--
-- Role in Spatial Mouse Processor:
-- - Counts bitstream frequency incoming from external oscillator
--   circuit.
------------------------------------------------------------------


library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;


entity counter is
generic ( counterwidth : positive := 4 );
port(
   clear : in std_logic;
   clock : in std_logic;
   q     : out std_logic_vector(counterwidth-1 downto 0));
end counter;

architecture behaviour of counter is
begin

process
variable count : std_logic_vector(counterwidth-1 downto 0);
begin
   if clear = '1' then
      count := (others => '0');
   elsif rising_edge(clock) then
      count := count + '1';
   end if;

   -- assign internal variable to output q
   q <= count;
end process;

end behaviour;
```

```
--------------------------------------------------------------------
-- processor.vhd :
--
-- The processor counts the frequencies of two incoming bitstreams (two
-- axis) from an external oscillating signal when activated.  The
-- successive frequencies relative to the initial frequency counted
-- upon activation are translated into a opcode that reflects mouse
-- movement in two dimensions.
--
-- Outputs complimentary signals to external oscillators to alternate
-- turning each one on and off.
--
-- This file ties the components of the processor together:
-- - 1 sample timer
-- - 2 counters (one for each oscillating axis)
-- - 2 range comparators (one for each oscillating axis)
```

```
library ieee;
use ieee.std_logic_1164.all;
library work;
use work.processor_pkg.all;


entity processor is
   generic (
      datawidth   : positive := 22;    -- counter and comparator widths
      timer_count : positive := 250;   -- to generate 10ms enables
      range_minus2: positive := 2000;  -- 2000 counts wide ranges
      range_minus1: positive := 2000;
      range_zero  : positive := 2000;
      range_plus1 : positive := 2000;
      range_plus2 : positive := 2000);
   port(
      clock       : in std_logic;
      reset       : in std_logic;
      activate    : in std-logic;
      x_bitstream : in std-logic;
      y_bitstream : in std_logic;
      x_opcode    : out std_logic_vector(8 downto 0);
      y_opcode    : out std_logic_vector(8 downto 0);
      oscillator_enable : buffer std_logic;
      oscillator_enable_inv : buffer std_logic);
end processor;


architecture structural of processor is

-- pulse signal from timer to reset counter and advance sample
signal sample_push   : std_logic;
-- sample_push AND reset for clearing counter
signal counter_clear : std_logic;
-- patch signal to connect x_counter to x_comparator
signal x_counter_comparator_signal : std_logic_vector(datawidth-1
downto 0);
-- patch signal to connect y_counter to y_comparator
```

```
signal y_counter_comparator_signal : std_logic_vector(datawidth-1
downto 0);


begin

complimentary_oscillator_enabler : process(sample_push)
begin
   wait until rising_edge(sample_push);
        oscillator_enable <= not oscillator_enable;
end process complimentary_oscillator_enabler;
oscillator_enable_inv <= not oscillator_enable;

counter_clear <= sample_push or reset;

timer : sample_timer
   generic map(timer_count => timer_count)
   port map(
      clock    => clock,
      reset    => reset,
      enable   => sample_push);

x_counter : counter
   generic map(counterwidth => datawidth)
   port map(
      clear => counter_clear,
      clock => x_bitstream,
      q     => x_counter_comparator_signal);

x_range_comparator : range_comparator
   generic map(
      datawidth      => datawidth,
      range_zero     => range_zero,
      range_minus2   => range_minus2,
      range_minus1   => range_minus1,
      range_plus1    => range_plus1,
      range_plus2    => range_plus2)
   port map(
      clock       => sample_push,
      enable      => activate,
      reset       => reset,
      frequency   => x_counter_comparator_signal,
      data_out    => x_opcode);

y_counter : counter
   generic map(counterwidth => datawidth)
   port map(
      clear => counter_clear,
      clock => y_bitstream,
      q     => y_counter_comparator_signal);

y_range_comparator : range_comparator
   generic map(
      datawidth      => datawidth,
      range_zero     => range_zero,
      range_minus2   => range_minus2,
```

```
      range_minus1   => range_minus1,
      range_plus1    => range_plus1,
      range_plus2    => range_plus2)
   port map(
      clock       => sample_push,
      enable      => activate,
      reset       => reset,
      frequency   => y_counter_comparator_signal,
      data_out    => y_opcode);

end structural;
```

```
--------------------------------------------------------------------------
-- processor_pkg.vhd
--
-- This file contains the declarations for the components required by
-- the processor:
-- - counter
-- - sample_timer
-- - range_comparator
--------------------------------------------------------------------------


library ieee;
use ieee.std_logic_1164.all;

package processor_pkg is

   component counter is
      generic (counterwidth : positive := 4);
      port(
         clear, clock : in std_logic;
         q : out std_logic_vector(counterwidth-1 downto 0));
   end component counter;

   component sample_timer
      generic (timer_count : positive := 250);
      port(
         clock, reset : in std_logic;
         enable : out std_logic);
   end component sample_timer;

   component range_comparator is
      generic (
         datawidth : positive := 10;
         range_zero : positive := 100;
         range_minus2 : positive := 100;
         range_minus1 : positive := 100;
         range_plus1 : positive := 100;
         range_plus2 : positive := 100);
      port(
         clock, enable, reset : in std_logic;
         frequency : in std_logic_vector(datawidth-1 downto 0);
          data_out : out std_logic_vector(8 downto 0));
   end component range_comparator;

end processor_pkg;
```

```
--------------------------------------------------------------------------
-- command_selector.vhd
--
-- A 2-to-1 vector-mux.  The input vector widths are specified by the
-- “datawidth” generic.
--
-- Role in spatial mouse mouse_commander: receives a control signal
-- from master_controller to select between the initialization command
-- or mouse movement command for input into the byte_selector
-- preceding the transmitter.
--------------------------------------------------------------------------


library ieee;
use ieee.std_logic_1164.all;

entity command_selector is
   generic (
      datawidth : positive := 33);
   port(
      bytes_in : in std_logic_vector(datawidth*2-1 downto 0);
      sel      : in std_logic;
      byte_out : out std_logic_vector(datawidth-1 downto 0));
end command_selector;

architecture behavioral of command_selector is

begin

process is

begin
   if sel = '0' then
      byte_out <= bytes_in(datawidth-1 downto 0);
   else
      byte_out <= bytes_in(datawidth*2-1 downto datawidth);
   end if;
end process;

end behavioral;
```

```
--------------------------------------------------------------------------
-- byte_selector.vhd :
--
-- A 4-to-1 vector-mux.  The input vector widths are specified by the
-- “bytewidth” generic.
--
-- Role in spatial mouse mouse_commander: receives a control signal
-- from master_controller to select current byte to be transmitted by
-- the transmitter.
--------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;

entity byte_selector is
   generic (bytewidth : positive := 11);
   port(
      bytes_in : in std_logic_vector(bytewidth*3-1 downto 0);
      sel: in std_logic_vector(1 downto 0);
      byte_out : out std_logic_vector(bytewidth-1 downto 0));
end byte_selector;

architecture behavioral of byte_selector is

begin

process(sel) is

begin
   if sel = "00" then
      byte_out <= bytes_in(bytewidth-1 downto 0);
   elsif sel = "01" then
      byte_out <= bytes_in(bytewidth*2-1 downto bytewidth);
   elsif sel = "10" then
      byte_out <= bytes_in(bytewidth*3-1 downto bytewidth*2);
   else
      byte_out <= (others => '0');
   end if;
end process;

end behavioral;
```

```
--------------------------------------------------------------------------
-- master_controler.vhd :
--
-- Overall controller for mouse commander.  Looks at state of tristate
-- data and clock buses to determine when data transmission can take
-- place.  Controls when consecutive bytes are be transmitted based on
-- host inhibit states.
--------------------------------------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY master_controller IS
   Generic( delay_tx : positive := 5000;
               delay_rx : positive := 300);
   PORT(
      clk   : IN STD_LOGIC;   -- 25MHz system clock
      reset : IN STD_LOGIC; -- reset system ACTIVE HIGH
      enable: IN STD_LOGIC;   -- start statemachine ACTIVE HIGH
      PC_drive_clock: IN STD_LOGIC; -- PS/2 clock bus
      PC_drive_data: IN STD_LOGIC;  -- PS/2 data bus
      rx_done: IN STD_LOGIC; -- Rx done receiving
      tx_done: IN STD_LOGIC; -- Tx done sending
         -- Drive PC clock onto bus
      Pc_clock_drive: OUT STD_LOGIC;
         -- Assert dataline low for line control
      rx_line_control: OUT STD_LOGIC;
            -- Enable rx data component
      rx_data_comp_enable: OUT STD_LOGIC;
            -- Enable tx data component
      tx_data_comp_enable: OUT STD_LOGIC;
            -- Select initialization / Mouse data
      data_command_sel: OUT STD_LOGIC;
            -- Select data byte to transmit
      data_byte_sel: OUT STD_LOGIC_VECTOR(1 downto 0)             );
END master_controller;

ARCHITECTURE behavioural OF master_controller IS

TYPE STATE_TYPE IS (ready_rx, rx_wait, rx_data, line_control_data_1,
line_control_data_10, line_control_data_1010, line_control_data_101,
tx_ACK, start_tx,Tx_1, wait_1, Tx_2, wait_2, Tx_3, wait_3,
delay_next_packet);

SIGNAL state: STATE_TYPE;

BEGIN
   PROCESS (clk)
   variable count : integer;
   variable rx_count : integer;
   variable tx_count : integer;
   BEGIN

      if rising_edge(clk) then
         if reset = '1' then
               state <= start_tx;
```

```
      else

      CASE state IS

         -- Main transmitt loop
         WHEN start_tx =>
            tx_count:=0;
            IF PC_drive_clock = '1' and PC_drive_data = '1' and
                     enable = '1' THEN
               state <= Tx_1;
            END IF;

         WHEN Tx_1 =>
            IF tx_done='1' THEN
               state <= wait_1;
            END IF;

         WHEN wait_1 =>
            IF PC_drive_clock = '1' THEN
               state <= Tx_2;
            END IF;

         WHEN Tx_2 =>
            IF tx_done='1' THEN
               state <= wait_2;
            END IF;

         WHEN wait_2 =>
            IF PC_drive_clock = '1' THEN
               state <= Tx_3;
            END IF;

         WHEN Tx_3 =>
            IF tx_done='1' THEN
               state <= wait_3;
            END IF;

         WHEN wait_3 =>
            IF PC_drive_clock = '1' THEN
               state <= delay_next_packet;
            END IF;

         WHEN delay_next_packet =>
            IF tx_count = delay_tx THEN
               state <= start_tx;
            ELSE
               tx_count := tx_count + 1;
            END IF;
      END CASE;
   end if;
   end if;
END PROCESS;

WITH state SELECT
   tx_data_comp_enable  <= --'1' WHEN  tx_ACK,
            '1' WHEN Tx_1 | Tx_2 | Tx_3,
```

```
                    '0'   WHEN  others;

   WITH state SELECT
      data_byte_sel<=   --"00"   WHEN  tx_ACK,
   "00"  WHEN  Tx_1,
   "01"  WHEN  Tx_2,
   "10"  WHEN  Tx_3,
   "00"  WHEN  others;

   WITH state SELECT
      data_command_sel <=  '1'   WHEN  start_tx|Tx_1| wait_1 |
            Tx_2 | wait_2 | Tx_3 | wait_3
      |delay_next_packet,
                  '0'   WHEN  others;

   WITH state SELECT
      rx_data_comp_enable <=  --'1' WHEN rx_data,
                  '0'   WHEN others;

   WITH state SELECT
      Pc_clock_drive <= --'1' WHEN rx_data |
            line_control_data_10 |
   line_control_data_1 |
   line_control_data_101,                              '0' WHEN others;


   WITH state SELECT
      rx_line_control   <= '1' WHEN line_control_data
               '0' WHEN others;

END behavioural;
```

```
--------------------------------------------------------------------------
-- mouse_commander.vhd :
--
-- Sends standard PS/2 mouse action command or initialization commands
-- to the host (PC).
--
-- This file ties the mouse_commander components together:
-- - dual_phase_clock
-- - command_selector
-- - byte_selector
-- - master_controller
-- - transmitter
-- - inverter
-- - inverter_with_enable
-- - 2x bidir
--------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
library work;
use work.mouse_commander_pkg.all;

entity mouse_commander is
   generic (
      Tx_datawidth : positive := 11;
      frames : positive := 3;
      commandwidth : positive := 33);  -- always frames*Tx_datawidth
   port(
      clock : in std_logic;            -- 25.175 Mhz clock
      clock_25_kHz : in std_logic;     -- 25 KHz system clock
      reset : in std_logic;
      enable_send_command : in std_logic;
      command_bytes : in std_logic_vector(commandwidth*2-1 downto 0);
      PC_clock_bus : inout std_logic;  -- tristate buffered clock bus
      PC_data_bus : inout std_logic);  -- tristate buffered data bus
end mouse_commander;

architecture structural of mouse_commander is

signal Tx_enable : std_logic;
signal Rx_enable : std_logic;
signal data_clock: std_logic;
signal PC_clock : std_logic;
signal tristate_clock_enable: std_logic;  -- pulls clock bus low
according to transmitter controller
signal tristate_data_enable: std_logic;   -- pulls data bus low
according to transmitted command bytes

signal Tx_PC_clock_drive : std_logic;
signal Tx_PC_data_drive : std_logic;
signal Tx_done : std_logic;

signal Rx_PC_clock_drive : std_logic;
signal Rx_PC_data_drive : std_logic;
signal Rx_data_in : std_logic_vector(10 downto 0);
signal Rx_done : std_logic;
```

```
signal PC_data_drive : std_logic;
signal PC_clock_drive : std_logic;

signal byte_select : std_logic_vector(1 downto 0);
signal byte_signal : std_logic_vector(Tx_datawidth-1 downto 0);
signal command_select : std_logic;
signal command_signal : std_logic_vector(commandwidth-1 downto 0);

begin

PC_data_clock_generator : dual_phase_clock
   port map(
      reset    => reset,
      clock_in    => clock_25_kHz,
      clock_lead  => PC_clock,
      clock_lag   => data_clock);

command_router : command_selector
   generic map(
      datawidth => commandwidth)
   port map(
      bytes_in => command_bytes,
      sel      => command_select,
      byte_out => command_signal);

byte_router : byte_selector
   generic map(
      bytewidth => Tx_datawidth,
      frames   => frames)
   port map(
      bytes_in => command_signal,
      sel      => byte_select,
      byte_out => byte_signal);

command_frame_controller : master_controller
   port map(
      clk         => clock,
      reset       => reset,
      enable      => enable_send_command,
      PC_drive_clock => PC_clock_bus,
      PC_drive_data  => PC_data_bus,
      rx_done     => Rx_done,
      tx_done     => Tx_done,
      Pc_clock_drive => Rx_PC_clock_drive,
      rx_line_control   => Rx_PC_data_drive,
      rx_data_comp_enable => Rx_enable,
      tx_data_comp_enable => Tx_enable,
      data_command_sel  => command_select,
      data_byte_sel  => byte_select
);

byte_receiver : rx_data
port map(
data_stream => PC_data_bus,
enable      => Rx_enable,
```

```
clk         => PC_clock,
done        => Rx_done,
q           => Rx_data_in
);

byte_transmitter : transmitter
   generic map(
      Tx_datawidth => Tx_datawidth)
   port map(
      clock          => clock_25_kHz,
      enable         => Tx_enable,
      data_clock     => data_clock,
      data_in        => byte_signal,
      data_out       => Tx_PC_data_drive,
      PC_clock_drive => Tx_PC_clock_drive,
      done           => Tx_done);

PC_data_drive <= Tx_PC_data_drive or Rx_PC_data_drive;
data_driver_inverter : inverter
   port map(
      input       => PC_data_drive,
      inv_output  => tristate_data_enable);

PC_clock_drive <= Tx_PC_clock_drive or Rx_PC_clock_drive;
PC_clock_driver_inverter : inverter_with_enable
   port map(
      enable      => PC_clock_drive,
      input       => PC_clock,
      inv_output  => tristate_clock_enable);

PC_Mouse_data_bus : bidir
   port map(
      bidir    => PC_data_bus,
      enable   => tristate_data_enable);

PC_mouse_clock_bus : bidir
   port map(
      bidir    => PC_clock_bus,
      enable   => tristate_clock_enable);

end structural;
```

```
------------------------------------------------------------------
--  Rx data component
--

LIBRARY ieee;
USE ieee.std_logic_1164.all;

entity rx_data is
port(
   data_stream :  in std_logic;
   enable   :  in std_logic;
   clk      :  in std_logic;        -- 25 kHz Clock
   done     :  out std_logic;
   q     :  out std_logic_vector(10 downto 0)
   );
end entity rx_data;

architecture mixed of rx_data is
signal shift_reg_out : std_logic_vector(10 downto 0);

COMPONENT shift_in_reg is
port(
      clock    : IN STD_LOGIC ;
      enable   : IN STD_LOGIC ;
      shiftin  : IN STD_LOGIC ;
      q     : OUT STD_LOGIC_VECTOR (10 DOWNTO 0)
);
end component shift_in_reg;


begin

shift_update: process
variable count : integer;

begin

   wait until rising_edge(clk);
      if count=15 then
         q <= shift_reg_out;
         done <= '1';
         count := 0;
      else
         done <= '0';
      end if;

      if enable = '1' then
         count := count +1;
      end if;

end process shift_update;

rx_buffer: shift_in_reg
   port map(
         clock => clk,
         enable => enable,
```

```
          shiftin  => data_stream,
          q  => shift_reg_out
          );

end mixed;
```

```
--------------------------------------------------------------------------
-- transmitter.vhd
--
-- Transmits a mouse command byte to host.  PS/2 timing requirements
-- are enforced.
--------------------------------------------------------------------------


library ieee;
use ieee.std_logic_1164.all;
library work;
use work.transmitter_pkg.all;

entity transmitter is
   generic (
      Tx_datawidth : positive := 11);
   port(
      clock : in std_logic;
      enable : in std_logic;
      data_clock : in std_logic;
      data_in : in std_logic_vector(Tx_datawidth - 1 downto 0);
      data_out : out std_logic;
      PC_clock_drive : out std_logic;
      done : out std_logic);
end transmitter;

architecture structural of transmitter is

signal load_signal : std_logic;
signal shift_enable_signal : std_logic;

begin

Tx_timer : transmit_controller
   generic map(datawidth => Tx_datawidth)
   port map(
      Tx_enable => enable,
      clock => clock,
      data_clock => data_clock,
      load_register => load_signal,
      shift_enable => shift_enable_signal,
      PC_clock_drive => PC_clock_drive,
      Tx_done => done);

Tx_shift_register : shift_out_register
   generic map(datawidth => Tx_datawidth)
   port map(
      load => load_signal,
      shift => data_clock,
      enable_shift => shift_enable_signal,
      d =>  data_in,
      q => data_out);

end structural;
```

```
--------------------------------------------------------------------------
-- Inverter
--
--------------------------------------------------------------------------


LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY inverter IS

   PORT
   (
      input    : IN  STD_LOGIC;
      inv_output  : OUT    STD_LOGIC
   );

END inverter;

ARCHITECTURE behavioural OF inverter IS
BEGIN

   PROCESS (input)
   BEGIN
      inv_output <= not input;
   END PROCESS;

END behavioural;
```

```
--------------------------------------------------------------------------
-- Inverter with enable
--
--------------------------------------------------------------------------


LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY inverter_with_enable IS

   PORT
   (
      enable   : IN  STD_LOGIC;
      input    : IN  STD_LOGIC;
      inv_output  : OUT STD_LOGIC
   );

END inverter_with_enable;

ARCHITECTURE behavioural OF inverter_with_enable IS
BEGIN

   PROCESS (enable, input)
   BEGIN

      IF enable = '0' THEN

         inv_output <= '0';

      ELSE

         inv_output <= not input;

      END IF;

   END PROCESS;
END behavioural;
```

```
--------------------------------------------------------------------------
-- bi-directional bus
--
--------------------------------------------------------------------------


LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

     ENTITY bidir IS
         PORT(
             bidir   : INOUT STD_LOGIC;
             enable  : IN STD_LOGIC
             );
     END bidir;

     ARCHITECTURE behav OF bidir IS

    Begin
         PROCESS (enable, bidir)
          BEGIN
             IF( enable = '0') THEN
                 bidir <=  'Z';
             ELSE
                 bidir <= '0';
             END IF;
         END PROCESS;
     END behav;
```

```
--------------------------------------------------------------------------
-- Shift in Register
--
--------------------------------------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.all;
LIBRARY lpm;
USE lpm.lpm_components.all;

ENTITY shift_in_reg IS
generic( datawidth : positive := 10);
   PORT
   (
      clock    : IN STD_LOGIC ;
      enable   : IN STD_LOGIC ;
      shiftin  : IN STD_LOGIC ;
      q     : OUT STD_LOGIC_VECTOR (datawidth DOWNTO 0)
   );
END shift_in_reg;


ARCHITECTURE SYN OF shift_in_reg IS

   SIGNAL sub_wire0  : STD_LOGIC_VECTOR (datawidth DOWNTO 0);


   COMPONENT lpm_shiftreg
   GENERIC (
   LPM_WIDTH      : NATURAL;
   LPM_DIRECTION  : STRING
   );
   PORT (
   enable   : IN STD_LOGIC ;
   clock : IN STD_LOGIC ;
   q  : OUT STD_LOGIC_VECTOR (datawidth DOWNTO 0);
   shiftin  : IN STD_LOGIC
   );
   END COMPONENT;

BEGIN
   q    <= sub_wire0(10 DOWNTO 0);

   lpm_shiftreg_component : lpm_shiftreg
   GENERIC MAP (
      LPM_WIDTH => 11,
      LPM_DIRECTION => "LEFT"
   )
   PORT MAP (
      enable => enable,
      clock => clock,
      shiftin => shiftin,
      q => sub_wire0
   );

END SYN;
```

```
--------------------------------------------------------------------------
-- shift_out_register.vhd
--
-- Output shift register with parallel loading input
-- Assertion of load signal asynchronously loads register
-- Rising edge triggered shift only if shift is enabled
-- Outputs a high by default.
--------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;


entity shift_out_register is
   generic (datawidth : positive := 11);
   port(
      load : in std_logic;
      shift : in std_logic;
      enable_shift : in std_logic;
      d : in std_logic_vector(datawidth-1 downto 0);
      q : out std_logic);
end shift_out_register;

architecture behavioral of shift_out_register is
begin

process(load, shift) is

variable bitcount : integer range 0 to datawidth-1;
variable d_hold : std_logic_vector(datawidth-1 downto 0);

begin
   if load = '1' then
      d_hold := d;               -- load input vector into register
      bitcount := 0;             -- reset shift count
   elsif rising_edge(shift) then
      if enable_shift = '1' then
         q <= d_hold(bitcount);     -- output next bit
         bitcount := bitcount + 1;  -- increment bit_count
      else
         q <= '1';
      end if;
   end if;
end process;

end behavioral;
```

```
--------------------------------------------------------------------------
-- transmit_controller.vhd
--
-- synchronizes data transmission with system clock
-- and assertion of data at appropriate clock pulses
--------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;

entity transmit_controller is
   generic (datawidth : positive := 11);
    port (
      Tx_enable : in std_logic;  -- enable transmission
   clock : in std_logic;      -- System clock (25KHz)
   -- data clock required to initialize proper offset
      data_clock : in std_logic;
      -- load register with parallel data sitting at register input
      load_register: out std_logic;
      -- enable data to be shifted out of shift register
      shift_enable: out std_logic;
      -- enable PC_clock to be asserted on clock bus
      PC_clock_drive: out std_logic;
      Tx_done: out std_logic);
end transmit_controller;

architecture state_machine of transmit_controller is

-- name states
type state_type is (start, load_data, enable_shift, drive_PC_clock,
shift, stop);
signal state : state_type;

begin

process(clock, Tx_enable)

variable shift_count : integer range 0 to datawidth*2;
variable shift_done : integer range 0 to datawidth*2;

begin

   shift_done := datawidth*2 -1;
   -- always in start state until Tx_enable = '1'
   if Tx_enable = '0' then       -- reset state
            state <= start;
      shift_count := 0;

    elsif rising_edge(clock) then

      case state is

      -- nothing happens in this state
      when start =>
         state <= load_data;
         shift_count := 0;
```

```
   -- load_register goes high to signal to register to load data.
   -- this state checks that the data and PC clocks are in the
appropriate
   -- phase of 2 possible cycles to advance to the enable shift state,
if not,
   -- the load_register stalls in this state for one more clock pulse.

      when load_data =>
         if data_clock = '0' then
            state <= enable_shift;
         else
            state <= load_data;
         end if;

      -- this state enables the register to shift data out based on the
data_clock
      when enable_shift =>
         state <= drive_PC_clock;
         shift_count := shift_count + 1;

      -- this state drives the PC_clock onto the clock bus
      when drive_PC_clock =>
         state <= shift;
         shift_count := shift_count + 1;

      -- this state allows for continued shifting of the data
      when shift =>
         if shift_count = shift_done then
            state <= stop;
         else
            state <= shift;
         end if;
         shift_count := shift_count + 1;

      when stop =>
         state <= stop;

      end case;

   end if;
end process;

with state select
   load_register  <= '1' when load_data,
                     '0' when others;
with state select
   shift_enable   <= '1' when enable_shift,
                     '1' when drive_PC_clock,
                     '1' when shift,
                     '0' when others;
with state select
   PC_clock_drive    <= '1' when drive_PC_clock,
                     '1' when shift,
                     '0' when others;
```

```
with state select
   Tx_done     <= '1' when stop,
                  '0' when others;

end state_machine;
```

```
--------------------------------------------------------------------------
-- transmitter_pkg.vhd
--
-- This file contains the declarations for the components required by
-- the transmitter:
-- - transmit_controller
-- - shift_out_register
-- - inverter
-------------------------------------------

library ieee;
use ieee.std_logic_1164.all;

package transmitter_pkg is

component transmit_controller is
generic (datawidth : positive := 11);
port (
Tx_enable : in std_logic;
clock : in std_logic;
data_clock : in std_logic;
load_register: out std_logic;
shift_enable: out std_logic;
PC_clock_drive: out std_logic;
Tx_done : out std_logic);
end component transmit_controller;

component shift_out_register is
generic (datawidth : positive := 11);
port(
load : in std_logic;
shift : in std_logic;
enable_shift : in std_logic;
d : in std_logic_vector(datawidth-1 downto 0);
q : out std_logic);
end component shift_out_register;

end transmitter_pkg;
```

```
-- mouse_commander_pkg
--
-- This file contains the declarations for the components required by
-- the mouse_commander:
-- - dual_phase_clock
-- - command_selector
-- - byte_selector
-- - master_controller
-- - transmitter
-- - inverter
-- - inverter_with_enable
-- - 2x bidir
-----------------------------------------------

library ieee;
use ieee.std_logic_1164.all;

package mouse_commander_pkg is

component dual_phase_clock is
   port(
      reset: in std_logic;
      clock_in : in std_logic;
      clock_lead : buffer std_logic;
      clock_lag : buffer std_logic);
end component dual_phase_clock;

component transmitter is
   generic (
      Tx_datawidth : positive := 11);
   port(
      clock : in std_logic;
      enable : in std_logic;
      data_clock : in std_logic;
      data_in : in std_logic_vector(Tx_datawidth - 1 downto 0);
      data_out : out std_logic;
      PC_clock_drive : out std_logic;
      done : out std_logic);
end component transmitter;

component inverter_with_enable is
port(
enable   : IN  STD_LOGIC;
input    : IN  STD_LOGIC;
inv_output  : OUT STD_LOGIC);
end component inverter_with_enable;

component inverter is
port(
input    : IN  STD_LOGIC;
inv_output  : OUT STD_LOGIC);
end component inverter;

component bidir is
port(
bidir   : inout STD_LOGIC;
```

```
enable  : in STD_LOGIC
);
end component bidir;

component command_controller is
port(
clk            : IN  STD_LOGIC;
PC_drive_clock    : IN  STD_LOGIC;
send_command      : IN  STD_LOGIC;
tx_done        : IN  STD_LOGIC;
enable_tx         : OUT    STD_LOGIC;
sel                  : OUT STD_LOGIC_VECTOR(1 downto 0)
);
end component command_controller;

component command_selector is
generic (
datawidth : positive := 33);
port(
bytes_in : in std_logic_vector(datawidth*2-1 downto 0);
sel: in std_logic;
byte_out : out std_logic_vector(datawidth-1 downto 0));
end component command_selector;

component byte_selector is
generic (
frames : positive :=3;
bytewidth : positive := 11);
port(
bytes_in: in std_logic_vector(bytewidth*frames-1 downto 0);
sel: in std_logic_vector(1 downto 0);
byte_out : out std_logic_vector(bytewidth-1 downto 0));
end component byte_selector;

component rx_data is
port(
data_stream : in std_logic;
enable      : in std_logic;
clk         : in std_logic;
done        : out std_logic;
q           : out std_logic_vector(10 downto 0));
end component rx_data;

component shift_in_reg is
port(
clk      : in std_logic;
enable   : in std_logic;
shiftin  : in std_logic;
q        : OUT STD_LOGIC_VECTOR (10 DOWNTO 0));
end component shift_in_reg;

component master_controller is
port(
clk                        : IN STD_LOGIC;      -- 25MHz system clock
reset       : IN STD_LOGIC; -- reset system ACTIVE HIGH
enable      : IN STD_LOGIC;   -- start statemachine ACTIVE HIGH
```

```
PC_drive_clock : IN STD_LOGIC; -- PS/2 clock bus
PC_drive_data  : IN STD_LOGIC;   -- PS/2 data bus
rx_done     : IN STD_LOGIC; -- Rx done receiving
tx_done     : IN STD_LOGIC; -- Tx done sending
Pc_clock_drive : OUT STD_LOGIC;  -- Drive PC clock onto bus
rx_line_control   : OUT STD_LOGIC;  -- Assert dataline low for line
control
rx_data_comp_enable : OUT STD_LOGIC; -- Enable rx data component
tx_data_comp_enable : OUT STD_LOGIC; -- Enable tx data component
data_command_sel  : OUT STD_LOGIC; -- Select initialization / Mouse
data
data_byte_sel  : OUT STD_LOGIC_VECTOR(1 downto 0)     -- Select data
byte to transmit
);
end component master_controller;

end mouse_commander_pkg;
```

```
--------------------------------------------------------------------------
-- spatial_mouse.vhd
--
-- Overall digital design implemented on FPGA
--------------------------------------------------------------------------


library ieee;
use ieee.std_logic_1164.all;
library work;
use work.spatial_mouse_pkg.all;

entity spatial_mouse is
   port(
      clock : in std_logic;
      reset : in std_logic;
      activate : in std_logic;
      left_button : in std_logic;
      right_button : in std_logic;
      x_oscillation : in std_logic;
      y_oscillation : in std_logic;
      PC_clock_bus : inout std_logic;
      PC_data_bus : inout std_logic;
      oscillator_enable : buffer std_logic;
      oscillator_enable_inv : buffer std_logic);
end spatial_mouse;

architecture structural of spatial_mouse is

-- Active Low Signals
signal not_reset : std_logic;
signal not_activate : std_logic;
signal not_left_button : std_logic;
signal not_right_button : std_logic;

-- Internal signals and patches
signal clock_25KHz : std_logic;
signal x_opcode_signal : std_logic_vector(8 downto 0);
signal y_opcode_signal : std_logic_vector(8 downto 0);
signal mouse_command_signal : std_logic_vector(32 downto 0);
signal init_command_signal : std_logic_vector(32 downto 0);
signal command_signals: std_logic_vector(65 downto 0);      --
concatenation of mouse command & init command

begin

-- Active Low Conversions
not_reset <= not reset;
not_activate <= not activate;
not_left_button <= not left_button;
not_right_button <= not right_button;

clock_25KHz_generator : clock_ticked
   generic map(ticks => 503)
   port map(
      clock_in => clock,
```

```
      reset    => not_reset,
      clock_out => clock_25KHz);

oscillator_translator : processor
   generic map(
      datawidth      => 25,      -- counter and comparator widths
      rangewidth     => 15,
      timer_count    => 250,     -- 250 ticks of 25kHz clock to
generate 10ms enables
      range_minus2   => 1000,
      range_minus1   => 1000,
      range_zero     => 1000,
      range_plus1    => 1000,
      range_plus2    => 1000)
   port map(
      clock       => clock_25KHz,
      reset       => not_reset,
      activate    => not_activate,
      x_bitstream => x_oscillation,
      y_bitstream => y_oscillation,
      x_opcode    => x_opcode_signal,
      y_opcode    => y_opcode_signal,
      oscillator_enable => oscillator_enable,
      oscillator_enable_inv => oscillator_enable_inv);

processor_to_mouse_interface : mouse_command_mapper
   port map(
         x_movement     => x_opcode_signal,
         y_movement     => y_opcode_signal,
         left_button    => not_left_button,
         right_button   => not_right_button,
         mapped_command => mouse_command_signal);

init_command_signal <= "11111111111111111111111111111111";
command_signals <= mouse_command_signal & init_command_signal;

mouse_interface : mouse_commander
   generic map(
      Tx_datawidth => 11,
      frames => 3,
      commandwidth => 33)  -- always frames*Tx_datawidth
   port map(
      clock => clock,
      clock_25_kHz => clock_25KHz,
      reset => not_reset,
      enable_send_command => not_activate,
      command_bytes => command_signals,   -- command bytes inputted
here
      PC_clock_bus => PC_clock_bus,
      PC_data_bus => PC_data_bus);

end structural;
```

```
--------------------------------------------------------------------------
-- spatial_mouse_pkg.vhd
--
-- Contains the major component declarations for the overall digital
-- design
--------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;

package spatial_mouse_pkg is

component clock_ticked is
generic (ticks : positive := 503);
port(
   clock_in, reset : in std_logic;
   clock_out : buffer std_logic);
end component clock_ticked;

component processor is
   generic (
      datawidth   : positive := 22;    -- counter and comparator widths
      rangewidth  : positive := 15;
      timer_count : positive := 250;
      range_minus2 : positive := 2000;
      range_minus1 : positive := 2000;
      range_zero  : positive := 2000;
      range_plus1 : positive := 2000;
      range_plus2 : positive := 2000);
   port(
      clock, reset : in std_logic;
      activate, x_bitstream, y_bitstream : in std_logic;
      x_opcode, y_opcode : out std_logic_vector(8 downto 0);
      oscillator_enable : buffer std_logic;
      oscillator_enable_inv : buffer std_logic);
end component processor;

component mouse_command_mapper is
      port(
         x_movement : in std_logic_vector(8 downto 0);
         y_movement : in std_logic_vector(8 downto 0);
         left_button : in std_logic;
         right_button : in std_logic;
         mapped_command : out std_logic_vector(32 downto 0));
end component mouse_command_mapper;

component mouse_commander is
   generic (
      Tx_datawidth : positive := 11;
      commandwidth : positive := 33);
   port(
      clock : in std_logic;
      clock_25_kHz : in std_logic;
      reset : in std_logic;
enable_send_command : in std_logic;
command_bytes : in std_logic_vector(commandwidth*2-1 downto 0);
```

```
PC_clock_bus : inout std_logic;
PC_data_bus : inout std_logic);
end component mouse_commander;

end spatial_mouse_pkg;
```

```
----------------------------------------------------------------------
-- Glue logic between processor and mouse interface
--
-- Maps processor opcode to
----------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;

entity mouse_command_mapper is
      port(
         x_movement : in std_logic_vector(8 downto 0);
         y_movement : in std_logic_vector(8 downto 0);
         left_button : in std_logic;
         right_button : in std_logic;
         mapped_command : out std_logic_vector(32 downto 0));
end mouse_command_mapper;

architecture logic of mouse_command_mapper is

-- signals for generating correct parity bit
signal parity1 : std_logic;
signal parity2 : std_logic;
signal parity3 : std_logic;

begin

parity1 <= not ('0' xor '0' xor x_movement(8) xor y_movement(8) xor '1'
      xor '0' xor left_button xor right_button);

parity2 <= not (x_movement(7) xor x_movement(6) xor x_movement(5)
        xor x_movement(4)
        xor x_movement(3) xor x_movement(2) xor x_movement(1) xor
        x_movement(0));

parity3 <= not (y_movement(7) xor y_movement(6) xor y_movement(5) xor
        y_movement(4) xor y_movement(3) xor y_movement(2) xor
        y_movement(1) xor y_movement(0));

mapped_command <= '1' & parity3 & y_movement(7 downto 0) & '0'
         & '1' & parity2 & x_movement(7 downto 0) & '0'
         & '1' & parity1 & "00" & y_movement(8) &
      x_movement(8) & "10"
         & right_button & left_button & '0';

end logic;
```

```
--------------------------------------------------------------------------
-- clock_ticked.vhd :
--
-- Generates a slower clock from a much faster clock.
--
-- Role in Spatial Mouse: generates a 25 KHz system clock that is
-- useful for meeting PS/2 timing requirements.
--------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;

entity clock_ticked is
generic (ticks : positive := 503);
port(
   clock_in, reset : in std_logic;
   clock_out : buffer std_logic);
end clock_ticked;

architecture behaviour of clock_ticked is
begin

process

variable count : natural range 1 to ticks;

begin
    wait until rising_edge(clock_in);
    if reset = '1' then
      count := 1;
      clock_out <= '0';
    else
        count := count + 1;
        if count = ticks then
          clock_out <= not clock_out;
        end if;
   end if;
end process;

end behaviour;
```

```
------------------------------------------------------------
-- dual_phase_clock.vhd
--
-- generates two clocks 1/4 out of phase from each other
-- the output clocks are 1/2 the frequency of the input clock
------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
library work;
use work.dual_phase_clock_pkg.all;

entity dual_phase_clock is
   port(
      reset: in std_logic;
      clock_in : in std_logic;
      clock_lead : buffer std_logic;
      clock_lag : buffer std_logic);
end dual_phase_clock;

architecture structural of dual_phase_clock is

signal phase_reset : std_logic;  -- this signal ensures that data clock
lags PC clock by 1/4 period
signal clock_lead_inv, clock_lag_inv : std_logic;

begin

process(clock_in) is          -- this process ensures that data clock
will lag PC clock by 1/4 period
begin
   wait until falling_edge(clock_in);
   if reset = '1' then
      phase_reset <= '1';
   else
      phase_reset <= '0';
   end if;
end process;

clock_lead_inv <= not clock_lead;
clock_lag_inv <= not clock_lag;

lead_clock_generator : d_flipflop_rising
   port map(
      clock => clock_in,
      clear => phase_reset,
      d => clock_lead_inv,
      q => clock_lead);

lag_clock_generator : d_flipflop_falling
   port map(
      clock => clock_in,
      clear => phase_reset,
      d => clock_lag_inv,
      q => clock_lag);
```

```
end structural;
```

```
--------------------------------------------------------------------------
-- dual_phase_clock_pkg.vhd
--
-- Declarations for components required by dual+phase_clock
--------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;

package dual_phase_clock_pkg is

component d_flipflop_rising is
port(
clock, clear : in std_logic;
d : in std_logic;
q : out std_logic);
end component d_flipflop_rising;

component d_flipflop_falling is
port(
clock, clear : in std_logic;
d : in std_logic;
q : out std_logic);
end component d_flipflop_falling;

end dual_phase_clock_pkg;
```

```
--------------------------------------------------------------------------------
-- Spatial Mouse Test Bench
--
-- Tests overall spatial mouse functionality
--------------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;

entity mux_8_to_1 is
port(
   in_0  : in std_logic;
   in_1  : in std_logic;
   in_2  : in std_logic;
   in_3  : in std_logic;
   in_4  : in std_logic;
   in_5  : in std_logic;
   in_6  : in std_logic;
   in_7  : in std_logic;
   sel   : in std_logic_vector(2 downto 0);
   output : out std_logic
);
end mux_8_to_1;

architecture behav of mux_8_to_1 is

begin

process( sel,in_0,in_1,in_2,in_3,in_4,in_5,in_6,in_7 )
   begin
      case sel is
         WHEN "000" =>
            output <= in_0;

         WHEN "001" =>
            output <= in_1;

         WHEN "010" =>
            output <= in_2;

         WHEN "011" =>
            output <= in_3;

         WHEN "100" =>
            output <= in_4;

         WHEN "101" =>
            output <= in_5;

         WHEN "110"=>
            output <= in_6;

         WHEN "111"=>
            output <= in_7;

         WHEN others =>
```

```
            output <= in_0;

      end case;
end process;

end behav;



library ieee;
use ieee.std_logic_1164.all;

package test_pkg is

component mux_8_to_1 is
   port  (
            in_0  : in std_logic;
            in_1  : in std_logic;
            in_2  : in std_logic;
            in_3  : in std_logic;
            in_4  : in std_logic;
            in_5  : in std_logic;
            in_6  : in std_logic;
            in_7  : in std_logic;
            sel   : in std_logic_vector(2 downto 0);
            output : out std_logic
         );
   end component mux_8_to_1;
end package test_pkg;



library ieee;
use ieee.std_logic_1164.all;

library work;
use work.spatial_mouse_pkg.all;
use work.test_pkg.all;


entity spatial_mouse_test is
end spatial_mouse_test;

architecture mixed of spatial_mouse_test is

-- Frequency generator constants
constant T_halfclock_25_MHz   : time := 20 ns;
constant T_halfclock_5_MHz       : time := 100 ns;
constant T_halfclock_5_75_MHz : time := 86.96 ns; --86.96 ns
constant T_halfclock_5_85_MHz : time := 85.47 ns;
constant T_halfclock_6_MHz       : time := 83.3 ns;
constant T_halfclock_6_15_MHz : time := 81.3 ns; -- 81.1 ns
constant T_halfclock_6_25_MHz : time := 80 ns;
constant T_halfclock_7_MHz       : time := 71 ns;
```

```
constant T_prop: time := 10 ns; -- avoid hold time violations

-- Test input frequencies
signal clock_25_MHz : std_logic;
signal osc_freq_5_MHz : std_logic;
signal osc_freq_5_75_MHz : std_logic;
signal osc_freq_5_85_MHz : std_logic;
signal osc_freq_6_MHz : std_logic;
signal osc_freq_6_15_MHz : std_logic;
signal osc_freq_6_25_MHz : std_logic;
signal osc_freq_7_MHz : std_logic;

-- Interface buttons
signal reset_internal : std_logic;
signal activate_internal : std_logic;
signal left_button_internal : std_logic;
signal right_button_internal : std_logic;

-- control oscillator frequency
signal x_osc_select : std_logic_vector(2 downto 0);
signal y_osc_select : std_logic_vector(2 downto 0);
signal x_osc_internal : std_logic;
signal y_osc_internal : std_logic;

-- PC bidirectional bus
signal PC_clock_bus_internal : std_logic;
signal PC_data_bus_internal : std_logic;


component spatial_mouse is
      port(
         clock : in std_logic;
         reset : in std_logic;
         activate : in std_logic;
         left_button : in std_logic;
         right_button : in std_logic;
         x_oscillation : in std_logic;
         y_oscillation : in std_logic;
         PC_clock_bus : inout std_logic;
         PC_data_bus : inout std_logic
   );
end component spatial_mouse;


begin

-- 25 Mhz clock generator
   clock_gen_25_MHz : process
   begin
      clock_25_MHz <= '0';
      wait for T_halfclock_25_MHz;
      clock_25_MHz <= '1';
      wait for T_halfclock_25_MHz;
   end process clock_gen_25_MHz;

-- 5 Mhz clock generator
```

```
   signal_gen_5_MHz : process
   begin
      osc_freq_5_MHz <= '0';
      wait for T_halfclock_5_MHz;
      osc_freq_5_MHz <= '1';
      wait for T_halfclock_5_MHz;
   end process signal_gen_5_MHz;

-- 5.75 Mhz clock generator
   signal_gen_5_75_MHz : process
   begin
      osc_freq_5_75_MHz <= '0';
      wait for T_halfclock_5_75_MHz;
      osc_freq_5_75_MHz <= '1';
      wait for T_halfclock_5_75_MHz;
   end process signal_gen_5_75_MHz;

-- 5.85 Mhz clock generator
   signal_gen_5_85_MHz : process
   begin
      osc_freq_5_85_MHz <= '0';
      wait for T_halfclock_5_85_MHz;
      osc_freq_5_85_MHz <= '1';
      wait for T_halfclock_5_85_MHz;
   end process signal_gen_5_85_MHz;

-- 6 Mhz clock generator
   signal_gen_6_MHz : process
   begin
      osc_freq_6_MHz <= '0';
      wait for T_halfclock_6_MHz;
      osc_freq_6_MHz <= '1';
      wait for T_halfclock_6_MHz;
   end process signal_gen_6_MHz;

-- 6.15 Mhz clock generator
   signal_gen_6_15_MHz : process
   begin
      osc_freq_6_15_MHz <= '0';
      wait for T_halfclock_6_15_MHz;
      osc_freq_6_15_MHz <= '1';
      wait for T_halfclock_6_15_MHz;
   end process signal_gen_6_15_MHz;

-- 6.25 Mhz clock generator
   signal_gen_6_25_MHz : process
   begin
      osc_freq_6_25_MHz <= '0';
      wait for T_halfclock_6_25_MHz;
      osc_freq_6_25_MHz <= '1';
      wait for T_halfclock_6_25_MHz;
   end process signal_gen_6_25_MHz;

-- 7 Mhz clock generator
   signal_gen_7_MHz : process
   begin
```

```
      osc_freq_7_MHz <= '0';
      wait for T_halfclock_7_MHz;
      osc_freq_7_MHz <= '1';
      wait for T_halfclock_7_MHz;
   end process signal_gen_7_MHz;


-- Spatial Mouse test subject
Spatial_mouse_component : component spatial_mouse
port map(
   clock => clock_25_MHz,  -- 25 MHz system clock
   reset => reset_internal,
   activate => activate_internal,
   left_button => left_button_internal,
   right_button => right_button_internal,
   x_oscillation => x_osc_internal,
   y_oscillation => y_osc_internal,
   PC_clock_bus => PC_clock_bus_internal,
   PC_data_bus => PC_data_bus_internal
   );

-- Component to change x input oscillator frequency
x_osc_input_sel : component mux_8_to_1
port map(
   in_0 => osc_freq_5_MHz,
   in_1 => osc_freq_5_75_MHz,
   in_2 => osc_freq_5_85_MHz,
   in_3 => osc_freq_6_MHz,
   in_4 => osc_freq_6_15_MHz,
   in_5 => osc_freq_6_25_MHz,
   in_6 => osc_freq_7_MHz,
   in_7 => osc_freq_6_MHz,
   sel => x_osc_select,
   output => x_osc_internal
   );

-- Component to control y input oscillator frequency
y_osc_input_sel : component mux_8_to_1
port map(
   in_0 => osc_freq_5_MHz,
   in_1 => osc_freq_5_75_MHz,
   in_2 => osc_freq_5_85_MHz,
   in_3 => osc_freq_6_MHz,
   in_4 => osc_freq_6_15_MHz,
   in_5 => osc_freq_6_25_MHz,
   in_6 => osc_freq_7_MHz,
   in_7 => osc_freq_6_MHz,
   sel => y_osc_select,
   output => y_osc_internal
   );


-------------------------------------------------
-- Process to activate mouse
activate_mouse : process
begin
```

```
   -- initial startup, reset machine
   activate_internal <= '1';
   reset_internal <= '1';
   wait for 1 ms;
   reset_internal <= '0';
   wait for 1 ms;
   reset_internal <= '1';
   wait for 1 ms;

   -- activate mouse
   activate_internal <= '0';
   wait for 60 ms;

   -- deactivate mouse
-- activate_internal <= '1';
-- wait for 20 ms;

   wait;
end process activate_mouse;
---------------------------------------------------

---------------------------------------------------
-- process to input oscillator signal
emu_oscillator : process
begin

   -- initially 6 MHz osc
   x_osc_select <= "011";
   y_osc_select <= "011";
   wait for 3 ms;

   -- x_osc to 6.15 MHz
   -- y_osc to 5.75 MHz
   x_osc_select <= "100";
   y_osc_select <= "001";
   wait for 5 ms;


   -- x_osc to 6.25 MHz
   -- y_osc to 5.85 MHz
   x_osc_select <= "101";
   y_osc_select <= "010";
   wait for 5 ms;


   -- x_osc to 7 MHz
   -- y_osc to 5 MHz
   x_osc_select <= "110";
   y_osc_select <= "000";
   wait for 5 ms;

   -- x_osc to 5.85 MHz
   -- y_osc to 6.15 MHz
   x_osc_select <= "010";
   y_osc_select <= "100";
```

```
   wait for 5 ms;

   -- x_osc to 5.75 MHz
   -- y_osc to 6.25 MHz
   x_osc_select <= "001";
   y_osc_select <= "101";
   wait for 5 ms;

   -- x_osc to 5 MHz
   -- y_osc to 7 MHz
   x_osc_select <= "000";
   y_osc_select <= "110";
   wait for 5 ms;


   wait;
end process emu_oscillator;
-------------------------------------------------

-------------------------------------------------
-- process to control Data and Clock bus
bus_control : process
begin

   -- Clk and data buses initially '1'
   -- Mouse in idle state
   PC_data_bus_internal <= '1';
   PC_clock_bus_internal <= '1';
-- wait for  1550 us;

-- PC_data_bus_internal <= '1';
-- PC_clock_bus_internal <= '0';
-- wait for 200 us;

-- PC_data_bus_internal <= '1';
-- PC_clock_bus_internal <= '1';
-- wait for 1100 us;


   wait;
end process bus_control;
-------------------------------------------------

-------------------------------------------------
-- process to control Mouse buttons
button_control : process
begin

   left_button_internal <= '1';
   right_button_internal <= '1';
-- wait for 12 ms;

   wait;

end process button_control;
-------------------------------------------------
```

```
end mixed;
```

```
--------------------------------------------------------------------------
-- processor_testbench2.vhd
--
-- Tests processor functionality
--------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
library work;
use work.processor_pkg.all;

entity processor_testbench2 is
end processor_testbench2;

architecture testbench of processor_testbench2 is

-- internal signals used to dirve or receive signals on/from chip ports
signal clock : std_logic;
signal reset : std_logic;
signal activate : std_logic;
signal x_bitstream : std_logic;
signal y_bitstream : std_logic;
signal x_opcode : std_logic_vector(8 downto 0);
signal y_opcode : std_logic_vector(8 downto 0);

-- half periods used to generate system clock and simulate bitstreams
constant T_halfclock : time := 20.0 us;   -- half period for 25KHz
sysem clock
constant T_6p0 : time := 83.3 ns;   -- half period for 6.0 MHz
bitstream
constant T_6p1 : time := 82.0 ns;   -- half period for 6.1 MHz
bitstream
constant T_6p2 : time := 80.6 ns;   -- half period for 6.2 MHz
bitstream
constant T_6p3 : time := 79.4 ns;   -- half period for 6.3 MHz
bitstream
constant T_6p4 : time := 78.1 ns;   -- half period for 6.4 MHz
bitstream
constant T_6p5 : time := 76.9 ns;   -- half period for 6.5 MHz
bitstream
constant T_6p6 : time := 75.8 ns;   -- half period for 6.6 MHz
bitstream
constant T_6p7 : time := 74.6 ns;   -- half period for 6.7 MHz
bitstream
constant T_6p8 : time := 73.5 ns;   -- half period for 6.8 MHz
bitstream
constant T_6p9 : time := 72.5 ns;   -- half period for 6.9 MHz
bitstream
constant T_7p0 : time := 71.4 ns;   -- half period for 7.0 MHz
bitstream
constant T_7p1 : time := 70.4 ns;   -- half period for 7.1 MHz
bitstream

-- various bitream frequencies used for simulation of x_bitstream
signal bitstream_6p0MHz : std_logic;
signal bitstream_6p1MHz : std_logic;
```

```
signal bitstream_6p2MHz : std_logic;
signal bitstream_6p3MHz : std_logic;
signal bitstream_6p4MHz : std_logic;
signal bitstream_6p5MHz : std_logic;
signal bitstream_6p6MHz : std_logic;
signal bitstream_6p7MHz : std_logic;
signal bitstream_6p8MHz : std_logic;
signal bitstream_6p9MHz : std_logic;
signal bitstream_7p0MHz : std_logic;
signal bitstream_7p1MHz : std_logic;

-- control signal used to drive any of various frequencies onto
x_bitstream
signal x_select : integer range 0 to 11;
signal y_select : integer range 0 to 11;


begin

-- insert processor component to be tested
processor_component : processor
port map(
   clock => clock,
   reset => reset,
   activate => activate,
   x_bitstream => x_bitstream,
   y_bitstream => y_bitstream,
   x_opcode => x_opcode,
   y_opcode => y_opcode);

-- mux used to drive any of various frequencies onto x_bitstream
x_bitstream_mux : process(x_select, bitstream_6p0MHz, bitstream_6p1MHz,
                bitstream_6p2MHz, bitstream_6p3MHz, bitstream_6p4MHz,
                bitstream_6p5MHz, bitstream_6p6MHz, bitstream_6p7MHz,
                bitstream_6p8MHz, bitstream_6p9MHz, bitstream_7p0MHz,
                bitstream_7p1MHz)
begin
   case x_select is
      when 0 =>
         x_bitstream <= bitstream_6p0MHz;
      when 1 =>
         x_bitstream <= bitstream_6p1MHz;
      when 2 =>
         x_bitstream <= bitstream_6p2MHz;
      when 3 =>
         x_bitstream <= bitstream_6p3MHz;
      when 4 =>
         x_bitstream <= bitstream_6p4MHz;
      when 5 =>
         x_bitstream <= bitstream_6p5MHz;
      when 6 =>
         x_bitstream <= bitstream_6p6MHz;
      when 7 =>
         x_bitstream <= bitstream_6p7MHz;
      when 8 =>
         x_bitstream <= bitstream_6p8MHz;
```

```
      when 9 =>
         x_bitstream <= bitstream_6p9MHz;
      when 10 =>
         x_bitstream <= bitstream_7p0MHz;
      when 11 =>
         x_bitstream <= bitstream_7p1MHz;
   end case;
end process x_bitstream_mux;

-- mux used to drive any of various frequencies onto y_bitstream
y_bitstream_mux : process(y_select, bitstream_6p0MHz, bitstream_6p1MHz,
                bitstream_6p2MHz, bitstream_6p3MHz, bitstream_6p4MHz,
                bitstream_6p5MHz, bitstream_6p6MHz, bitstream_6p7MHz,
                bitstream_6p8MHz, bitstream_6p9MHz, bitstream_7p0MHz,
                bitstream_7p1MHz)
begin
   case y_select is
      when 0 =>
         y_bitstream <= bitstream_6p0MHz;
      when 1 =>
         y_bitstream <= bitstream_6p1MHz;
      when 2 =>
         y_bitstream <= bitstream_6p2MHz;
      when 3 =>
         y_bitstream <= bitstream_6p3MHz;
      when 4 =>
         y_bitstream <= bitstream_6p4MHz;
      when 5 =>
         y_bitstream <= bitstream_6p5MHz;
      when 6 =>
         y_bitstream <= bitstream_6p6MHz;
      when 7 =>
         y_bitstream <= bitstream_6p7MHz;
      when 8 =>
         y_bitstream <= bitstream_6p8MHz;
      when 9 =>
         y_bitstream <= bitstream_6p9MHz;
      when 10 =>
         y_bitstream <= bitstream_7p0MHz;
      when 11 =>
         y_bitstream <= bitstream_7p1MHz;
   end case;
end process y_bitstream_mux;

-- clock generator
system_clock_generator : process
begin
   clock <= '0';
   wait for T_halfclock;
   clock <= '1';
   wait for T_halfclock;
end process system_clock_generator;

-- 6.0MHz bitstream generator
bitstream_6p0MHz_generator : process
begin
```

```
   bitstream_6p0MHz <= '0';
   wait for T_6p0;
   bitstream_6p0MHz <= '1';
   wait for T_6p0;
end process bitstream_6p0MHz_generator;

-- 6.1MHz bitstream generator
bitstream_6p1MHz_generator : process
begin
   bitstream_6p1MHz <= '0';
   wait for T_6p1;
   bitstream_6p1MHz <= '1';
   wait for T_6p1;
end process bitstream_6p1MHz_generator;

-- 6.2MHz bitstream generator
bitstream_6p2MHz_generator : process
begin
   bitstream_6p2MHz <= '0';
   wait for T_6p2;
   bitstream_6p2MHz <= '1';
   wait for T_6p2;
end process bitstream_6p2MHz_generator;

-- 6.3MHz bitstream generator
bitstream_6p3MHz_generator : process
begin
   bitstream_6p3MHz <= '0';
   wait for T_6p3;
   bitstream_6p3MHz <= '1';
   wait for T_6p3;
end process bitstream_6p3MHz_generator;

-- 6.4MHz bitstream generator
bitstream_6p4MHz_generator : process
begin
   bitstream_6p4MHz <= '0';
   wait for T_6p4;
   bitstream_6p4MHz <= '1';
   wait for T_6p4;
end process bitstream_6p4MHz_generator;

-- 6.5MHz bitstream generator
bitstream_6p5MHz_generator : process
begin
   bitstream_6p5MHz <= '0';
   wait for T_6p5;
   bitstream_6p5MHz <= '1';
   wait for T_6p5;
end process bitstream_6p5MHz_generator;

-- 6.6MHz bitstream generator
bitstream_6p6MHz_generator : process
begin
   bitstream_6p6MHz <= '0';
   wait for T_6p6;
```

```
   bitstream_6p6MHz <= '1';
   wait for T_6p6;
end process bitstream_6p6MHz_generator;

-- 6.7MHz bitstream generator
bitstream_6p7MHz_generator : process
begin
   bitstream_6p7MHz <= '0';
   wait for T_6p7;
   bitstream_6p7MHz <= '1';
   wait for T_6p7;
end process bitstream_6p7MHz_generator;

-- 6.8MHz bitstream generator
bitstream_6p8MHz_generator : process
begin
   bitstream_6p8MHz <= '0';
   wait for T_6p8;
   bitstream_6p8MHz <= '1';
   wait for T_6p8;
end process bitstream_6p8MHz_generator;

-- 6.9MHz bitstream generator
bitstream_6p9MHz_generator : process
begin
   bitstream_6p9MHz <= '0';
   wait for T_6p9;
   bitstream_6p9MHz <= '1';
   wait for T_6p9;
end process bitstream_6p9MHz_generator;

-- 7.0MHz bitstream generator
bitstream_7p0MHz_generator : process
begin
   bitstream_7p0MHz <= '0';
   wait for T_7p0;
   bitstream_7p0MHz <= '1';
   wait for T_7p0;
end process bitstream_7p0MHz_generator;

-- 7.1MHz bitstream generator
bitstream_7p1MHz_generator : process
begin
   bitstream_7p1MHz <= '0';
   wait for T_7p1;
   bitstream_7p1MHz <= '1';
   wait for T_7p1;
end process bitstream_7p1MHz_generator;

test_sequence : process
begin

   -- 0ms: pulse reset at start
   reset <= '0';
   wait for 9 ms;
   reset <= '1';
```

```
   wait for 1 ms;
   reset <= '0';

   -- 10ms: unactivated, x_bitstream = 6.6MHz, y_bitstream = 6.3MHz
   x_select <= 6;
   y_select <= 3;
   wait for 10 ms;

   -- 20ms: activate/calibrate, x_bitstream = 6.6MHz, y_bitstream
6.3MHz
   activate <= '1';
   wait for 10 ms;

   -- 30ms: activated, x_bitstream = 6.6MHz, y_bitstream = 6.3MHz
   wait for 10 ms;

   -- 40ms: activated, x_bitstream = 6.7MHz, y_bitstream = 6.6MHz
   x_select <= 7;
   y_select <= 6;
   wait for 10 ms;

   -- 50ms: activated, x_bitstream = 6.9MHz, y_bitstream = 6.8MHz
   x_select <= 9;
   y_select <= 8;
   wait for 10 ms;

   -- 60ms: activated, x_bitstream = 7.1MHz, y_bitstream = 6.3MHz
   x_select <= 11;
   y_select <= 3;
   wait for 10 ms;

   -- 70ms: activated, x_bitstream = 6.0MHz, y_bitstream = 7.0MHZ
   x_select <= 0;
   y_select <= 10;
   wait for 10 ms;

   -- 80ms: activated, x_bitstream = 6.3MHz, y_bitstream = 6.0MHz
   x_select <= 3;
   y_select <= 0;
   wait for 10 ms;

   -- 90ms: activated, x_bitstream = 6.5MHz, y_bitstream = 6.0MHz
   x_select <= 5;
   y_select <= 0;
   wait for 10 ms;

   -- 100ms: deactivate, x_bitstream = 6.5MHz, y_bitstream = 6.0MHz
   activate <= '0';
   wait for 10 ms;

   -- 110ms: reactivate/recalibrate, x_bitstream = 6.8MHz, y_bitstream
= 6.5MHz
   activate <= '1';
   x_select <= 8;
   y_select <= 5;
   wait for 10 ms;
```

```
   -- 120ms: activated, x_bitstream = 6.3MHz, y_bitstream = 6.6MHz
   x_select <= 3;
   y_select <= 6;
   wait for 10 ms;

   -- 130ms: activated, x_bitstream = 6.0MHz, y_bitstream = 6.8MHz
   x_select <= 0;
   x_select <= 8;
   wait for 10 ms;

   -- 140ms: activated, x_bitstream = 7.1MHz, y_bitstream = 6.2MHz
   x_select <= 11;
   y_select <= 2;
   wait for 10 ms;

   -- finish
   wait for 30 ms;

end process test_sequence;

end testbench;
```