Physically Immersive Gaming

By:

Sergio Haro

Ceryen Tan

PJ Steiner

Abstract:

As video game console hardware becomes faster and more powerful, game developers are struggling to keep up with the complexity of such systems. The release of the Nintendo Wii broke away from the norm and put innovation through game play above visual advancements and attractions. Our project continues down the path opened up by Nintendo and puts physical interaction with video games at the forefront. The project involves a suit with a variety of sensors that maps certain predetermined gestures such as walking, jumping, and throwing, to button presses on a standard game controller. This "suit controller" will be connected wirelessly to a computer which has a Nintendo NES emulator installed. A player can then use the suit to play a variety of classic games that the NES has to offer.

6.111

Project Report

TA: David Wentzloff

May 17, 2007

TABLE OF CONTENTS

Introduction....1

System Overview....1

Sensors and Wireless Unit....3

Wireless....3

Transmitting Wireless Card....3

Receiving Wireless Card....5

Testing And Debugging Wireless Unit....5

Signal Processing Unit....6

The Distributor....7

The Motion Modules....7

The Interpreter....10

Testing The Signal Processing Unit....10

Keymapper....11

PS/2 Keyboard Module....12

PS/2 Electrical Protocol....13

PS/2 Keyboard Protocol....14

Implementation....15

Testing....18

Conclusion....19

Acknowledgements....19

INTRODUCTION

For our project, we wanted to implement a new way to play old, classic games. Instead of sitting on the couch or chair to play video games, we wanted to make things more interactive while at the same time bringing new life into classic games. Our project brings players into the video game world by forcing them to act out the movements players would like their digital avatars to make. Furthermore, we did not want to restrict the player to be within six feet of his television/monitor thus, we made our system wireless. To make our system more general we decided against creating our own video game but rather interface with a computer so that one could use the system to play any emulated console plus any computer game. Finally, we made the system programmable, in that the keys that each action corresponded to can be changed. This gives the system flexibility and uses beyond just playing video games.

SYSTEM OVERVIEW

We divided our project divided into four subsystems (Figure 1). The first subsystem is the actual suit. The suit has five sensors that map actions to button presses (Figure 2). We use accelerometers on each leg to determine “button presses” on the directional pad. Jumping motions correspond to a press on the up button. Ducking corresponds with a press on the down button. Walking motion corresponds to either the left or right button is being pressed. We make the left/right decision with a gyroscope placed on the torso to determine direction. Accelerometers on the arms will be used for other button presses. A right arm movement corresponds to a press of the “A” button while a left arm movement is a press of the “B” button. The start and select buttons are actual buttons on the FPGA. This layout lends itself for certain types of games, mainly side scrolling games. Other types of games might be unplayable, however for the scope of this project we are focusing on side scrolling video games such as Super Mario Bros. or Metroid.

The second subsystem is the wireless communication layer. Using a RF system, the suit sends the data gathered by the sensors to one of the FPGA labkits.

The third subsystem is the signal-processing unit. This system takes in the raw data from the suit and attempt to figure out what motion, if any, is being committed.

The fourth and final subsystem is the PS/2 communication layer. This system partially implements the PS/2 keyboard input interface for a computer. The fourth subsystem translates the “button press” outputs out of the third subsystem into a programmable set of keys on a keyboard and transmits these keys to a computer to finally play a game on the NES emulator.

In the following section, we will go into detail on the implementation of the four different subsystems. We first explain the implementation of the Sensors and Wireless unit. Afterwards, we explain how we accomplish motion detection in the Signal Processing Unit. Next, we discuss the KeyMapper, a module that allows us to program the keys we send to the computer. Finally, we discuss the implementation and details of the PS/2 keyboard interface.

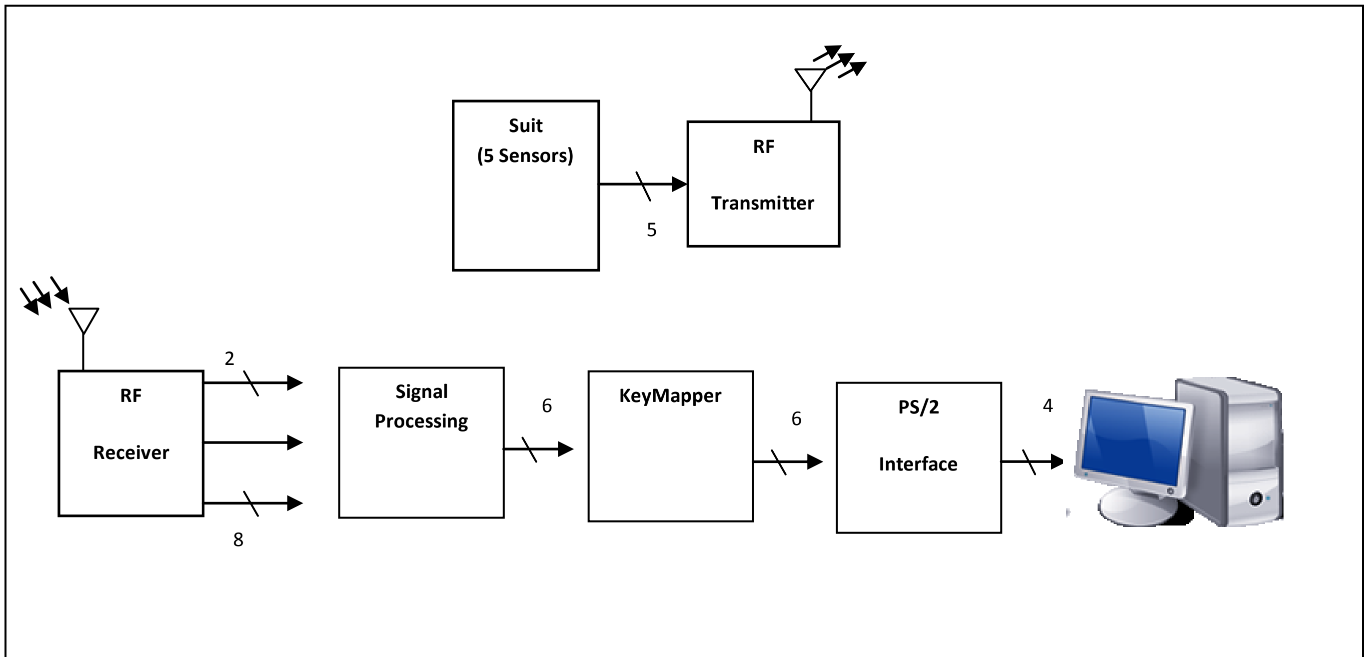

Figure 1 - System Diagram of Overall System

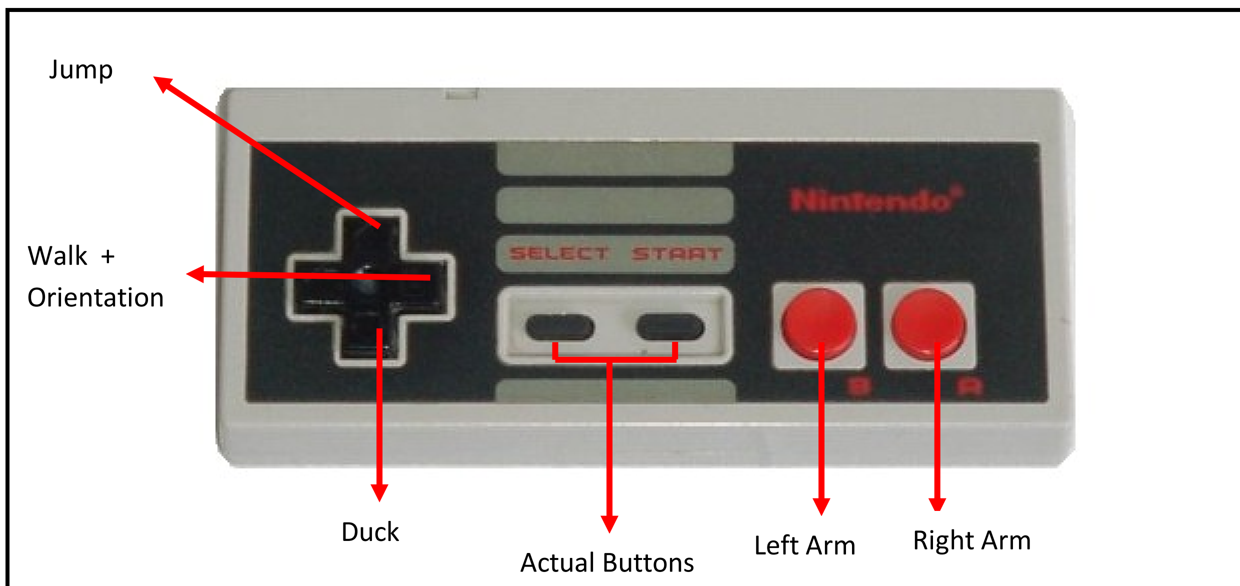

Figure 2 - Mapping of Gestures to Controller Buttons.

Sensors and Wireless Unit (By PJ Steiner)

The first of the three main components of the physically immersive video game system is the Sensor and Wireless module. This module includes the five sensors mounted on the player's body (four ADXL322EB accelerometers and one ADXRS300EB gyroscope), the circuitry associated with them, the transmitting wireless card (Chipcon CC1010EB), and the receiving wireless card (a second Chipcon CC1010EB). An overall block diagram of the Sensor and Wireless module is shown below.

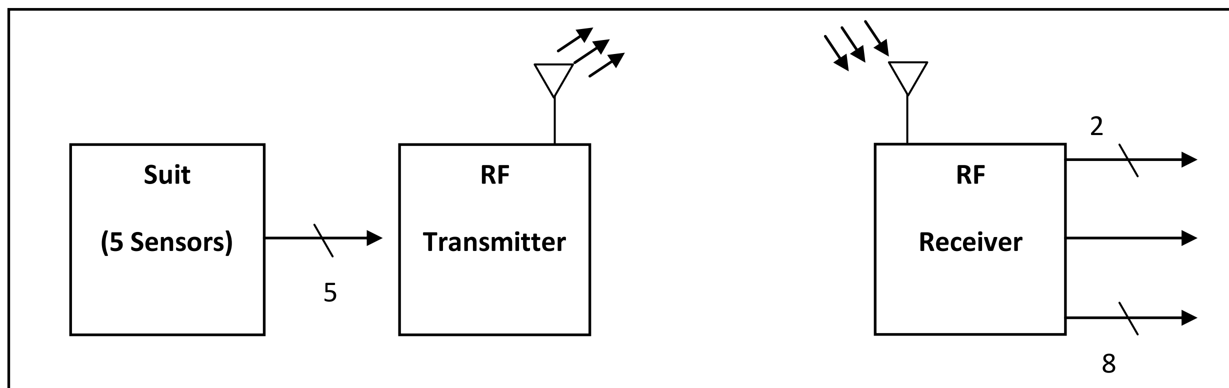

Figure 3 - Block Diagram of Sensor and Wireless Unit.

Because the ADC provided on the CC1010 wireless card has only three channels, the five sensors must somehow time-share one or more of the ADC channels if samples from each are to be taken. This is accomplished by time-multiplexing the outputs of the two accelerometers attached to the player's arms and the outputs of the two accelerometers attached to the player's legs using a single AD8182, which contains two analog switches. The time-multiplexed output of the first switch is tied to AD0, the first channel of the ADC; the time-multiplexed output of the second switch is tired to AD1, the second channel of the ADC; and the output of the gyroscope is tied to AD2, the third channel of the ADC. The task of generating select signals for each analog switch and then sampling its output at the appropriate time is delegated to the transmitting wireless card.

Wireless

The CC1010 wireless card, two of which are used in the Sensor and Wireless module, is controlled by an onboard 8051 microcontroller. For this project, the microcontroller was programmed in the C language using the HAL (hardware abstraction layer) provided by Chipcon, the makers of the CC1010. The HAL provides a number of macros and C functions which greatly simplify access to and use of the capabilities of the wireless card. The transmitting (TX) wireless card and the receiving (RX) wireless card run different microcontroller programs. The details of the operation of each program are described below.

Transmitting Wireless Card

The transmitting wireless card is responsible for generating select signals for the analog switches used to time-multiplex sensor data, sampling data from each sensor, and transmitting the sensor data to the receiving wireless card. When the card is first powered on, the transmission program executes the

following one-time initialization sequence:

- Disables the watchdog timer by calling WDT_ENABLE(...)
- Configures the onboard ADC to take single samples using VCC as the reference voltage by calling halCofigADC(...)
- Powers on the ADC by calling ADC_POWER(...)
- Sets P1.0 and P1.1, 8051 registers tied to pins on the CC1010, to be outputs
- Calibrates the radio with a call to halRFCalib(...)
- Turns the radio on in transmit mode with a call to halSetRxTxOff(...)

After executing the initialization procedure, the program enters an infinite loop. On every iteration of the loop, the program constructs a two byte packet to send. The first byte of the packet, packet[0], contains a sensor reading, while the second byte, packet[1], contains a type. The type indicates which sensor the data is from: 0 indicates a reading from the accelerometer on the player's left leg, 1 a reading from the accelerometer on the player's right leg, 2 a reading from the accelerometer on the player's left arm, 3 a reading from the accelerometer on the player's right arm, and 4 a reading from the gyroscope on the player's torso. Which type of reading the transmission program takes and sends on each iteration of the loop is determined by a counter, selectb, which is initialized to 0. Each iteration of the loop, the program does the following:

- If selectb is equal to 0, sets P1.0 (select input to the first analog switch) to 0 and selects AD0 as the ADC input
- If selectb is equal to 1, sets P1.0 to 1 and selects AD0 as the ADC input
- If selectb is equal to 2, sets P1.1 (select input to the second analog switch) to 0 and selects AD1 as the ADC input
- If selectb is equal to 3, sets P1.1 to 1 and selects AD1 as the ADC input
- If selectb is equal to 4, selects AD2 as the ADC input
- Gets a sample from the ADC and stores it in packet[0]
- Sets packet[1] to the value of selectb
- Transmits the two byte packet with a call to halRFSendPacket(...)
- Increments selectb if it is less than 4; otherwise, sets it to 0
- Spins for a number of cycles in order to give the wireless card enough time to send the packet

When running at the maximum rate the CC1010 is capable of transmitting data, this transmission program is capable of transmitting a packet slightly faster than once every 4 ms.

Receiving Wireless Card

The receiving wireless card is responsible for receiving packets sent by the transmitting wireless card and writing them to its output pins, which are tied to the inputs of the 6.111 labkit responsible for signal processing. Immediately after outputting a new packet, the receiving wireless card also briefly pulses a 'have' signal on another output pin to indicate to the signal processing module that a new packet has arrived.

When the receiving card is first powered on, it executes the following initialization sequence:

- Disables the watchdog timer by calling WDT_ENABLE(...)
- Sets P1.{0-6} and P2.{0-4} as outputs
- Calibrates the radio with a call to halRFCalib(...)
- Turns on the radio in receive mode with a call to halRFSetRxTxOff(...)

The receiving program, like the transmitting program, enters an infinite loop after initializing the radio and hardware. The program maintains a 2-byte array ('packet') in which it stores the latest packet it has received. On each iteration of the loop, the receiving program does the following:

- Receives a packet by calling halRFReceivePacket(...)
- Writes packet[0] (sensor reading) to eight pins {P2.0 P1.6 P1.5 P1.4 P1.3 P1.2 P1.1 P1.0}
- Writes the three LSBs of packet[1] (reading type) to the three pins {P2.3 P2.2 P2.1}
- Sets P2.4 to 1, spins briefly, and sets P2.4 to 0 (producing the 'have' pulse)

Debugging and Testing the Wireless Unit

Working with the CC1010 wireless cards provided some challenges not present when working in Verilog. There was no readily available way to simulate the performance of the two wireless cards together, so testing and debugging was performed primarily with the cards themselves. The logic analyzer was an invaluable tool for viewing output and attempting to diagnose problems with the code.

During the course of the Sensor and Wireless module's development, one major problem arose that took some time to resolve. It seemed, based on output from the receiving wireless card viewed on the logic analyzer that packets were arriving at the receiving card out of order and with their sensor data and type fields jumbled. After investigating the possibility of bits flipping, causing erroneous type values to be displayed, it was determined that the problem was actually with the halRFSendPacket(...) function, which is called by the transmitting wireless card. This function begins the process of sending a packet when called, but returns before sending is complete. The original transmission code did not wait at the end of each iteration of the infinite loop -- it simply looped again and attempted to send another packet. When halRFSendPacket(...) was called again before the first packet had finished transmitting, the new packet was simply not sent. Packets were transmitted with types 0, 1, 2, 3, 4, but arrived in with types 0, 2, 4, 1, 3 -- obviously, exactly every other packet was not being sent! The addition of the wait after transmitting packets allowed each transmission to complete, and no more trouble with dropped packets was experienced.

Signal Processing (By Sergio Haro)

The Signal Processing unit is the core logic module for the project. Its function is to translate the three inputs (*enable, type,* and *data*) from the wireless unit into particular motions (walking, jumping, ducking, and arm punches). The *enable* input is a signal signifying when type and data have been changed and are valid again. The *type* input is a three-bit signal identifying the source of *data*. The d*ata* input is an eight-bit reading from a sensor. The Signal Processing unit outputs six enable signals corresponding to the six actions: left, right, up, down, a, and b. The six actions are meant to represent the keys on an original Nintendo NES as shown on Figure 1.

The Signal Processing unit can be broken up into three different layers, as shown in Figure 2. The top layer, the Distributor, takes in the three inputs from the wireless unit and shuffles off data to the appropriate modules in the middle layer. For example, if the data coming in is from the left leg sensor, the Distributor sends the data to the Jump, Duck, and Walking modules. Or if the data is from the right arm sensor, the Distributor sends the data to the Right Arm module. The middle modules are called motion modules as the function of each module is to detect a particular motion. The motion modules each require data from a subset of the sensors. The final layer is the Interpreter. The function of the Interpreter is to take in the enables from the motion modules and produce a set of action signals. The Interpreter exists so that one could combine sets of motions into an action. For example, the right action is only triggered when a walking motion is occurring and the player is facing the right direction.

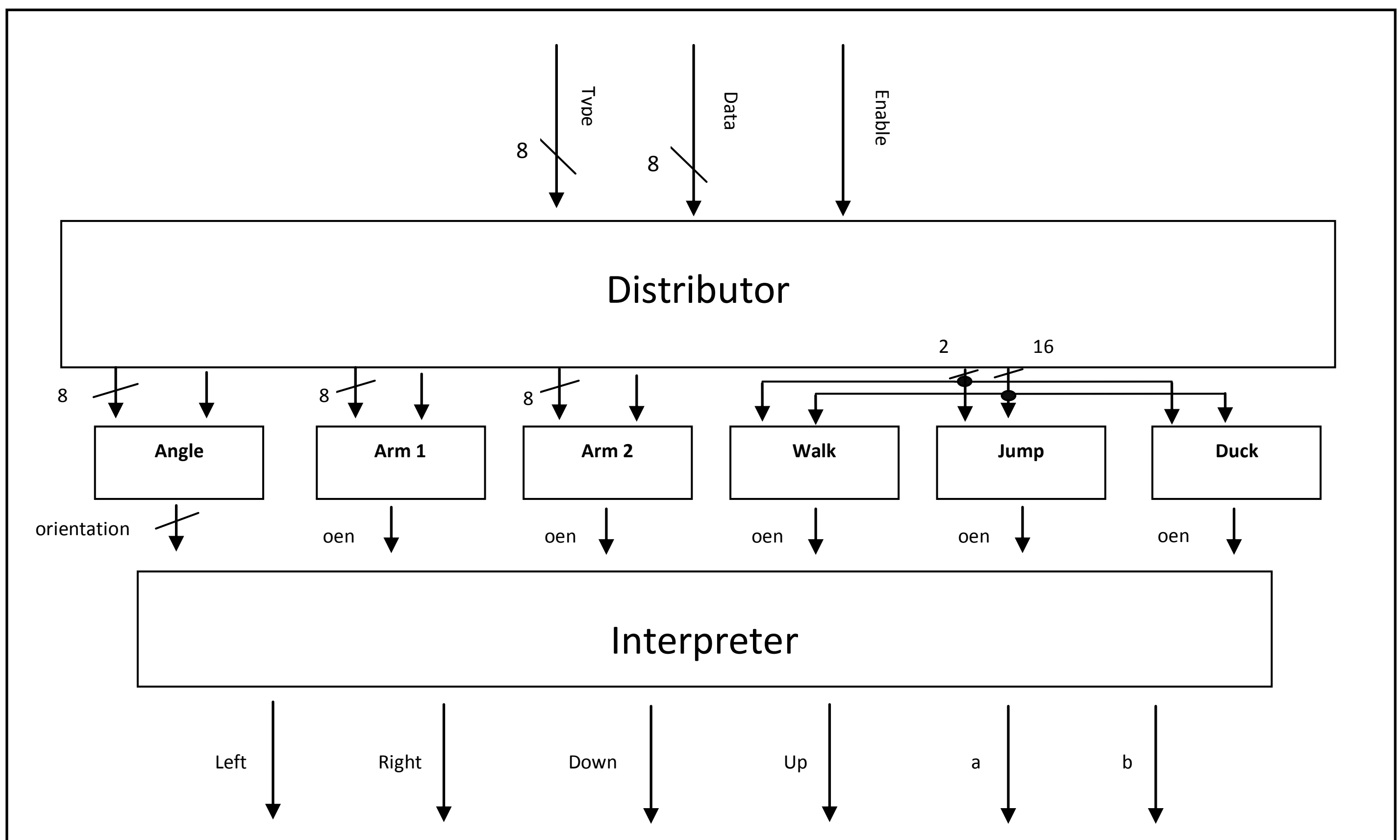

Figure 4 – Block Diagram of Overall Signal Processing Unit

In the following sections, we will go into further detail on each layer, beginning at the top layer and working down from there.

The Distributor

In this section, we will go to the details of the implementation of the Distributor module. As mentioned earlier, the Distributor has 3 inputs. *Enable* is a one-bit signal signifying new, valid data. *Type* is a three-bit signal representing the source of *data. Data* is an eight-bit signal representing the reading from the given sensor. These inputs are processed to produce five eight-bit signals and five corresponding one-bit enable signals. The outputs correspond to *left_leg, right_leg, left_arm, right_arm,* and *angle*. Each output has a corresponding enable signal that is active high when the data in the output has changed. The function of the Distributor, as mentioned prior, is to take the data from the five sensors that is being time multiplexed onto the wireless link and de-multiplex it so that the motion modules can all run in parallel. The Distributor also has the task of holding outputs to their last value until data from the sensor is sent again.

Each output is calculated separately in an identical manner. Thus the Distributor is actually five virtually identical units running in parallel. We say virtually because the parameters for each unit must change. The logic for these parallel units is best shown in a logic diagram which can be seen in Figure 3. As a minor note, all the inputs to the Distributor are synchronized to the system clock prior to being passed to the Distributor.

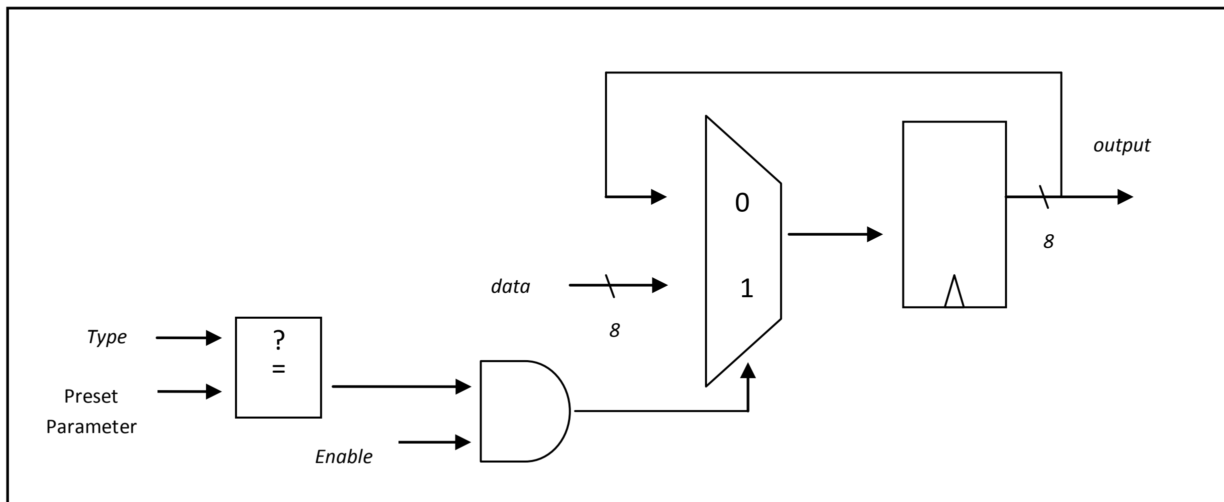

Figure 5 - Logic Diagram of Parallel Block in Distributor

The Motion Modules

In this section we will go into more detail on the motion modules. Since most of the motion modules work from the same principles, we will not go into exact details on every motion module. Instead we will provide an overview on how motion detection was handled. To accomplish these goals

we will use both the **Angle** and **Arm** motion modules as examples since the **Angle** modules uses a gyroscope while the **Arm** modules uses an accelerometer.

After investigating several possible ways to detect motion we settled on pattern matching. We chose this route for multiple reasons. First, pattern matching allows for a simple design and implementation. For our purposes, preciseness was not terribly important since we would be playing video games that originally were played with digital buttons. Secondly, since we were transmitting data through a wireless link we had to design a system that could tolerate lost packets. If we were to use more complicated methods that used integration, a lost packet would mean a deviation in the calculation.

The idea for pattern matching was hatched after analyzing the voltage waveforms produced by accelerometers and gyroscopes under pre-determined motions. Sample waveforms of accelerometer and gyroscope data can be seen if Figures 4 and 5. Figure 4 shows a sample waveform from an accelerometer when extending an arm outwards and then retracting it. The waveform begins with a positive spike (positive acceleration) as the arm is being extended. When an arm reaches its maximum reach, there is a negative spike (deceleration). When the arm begins retracting it causes another negative spike (negative acceleration). The final positive spike is caused by the deceleration as the arm stops at the torso. A gyroscope measures angular velocity and thusly when turning, the waveform increases to the maximum velocity and stays steady until a person stops turning.

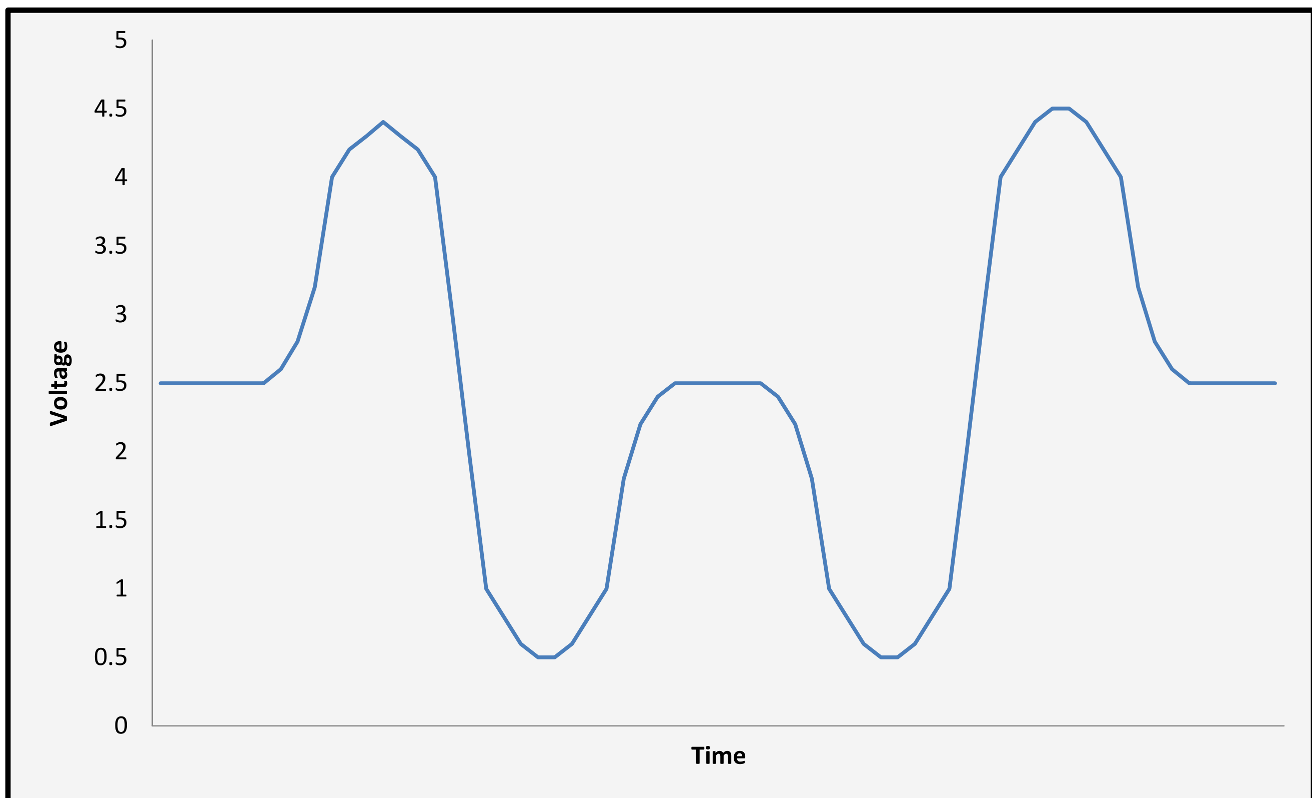

Figure 6- Sample Waveform of Accelerometer during Arm extension and retraction.

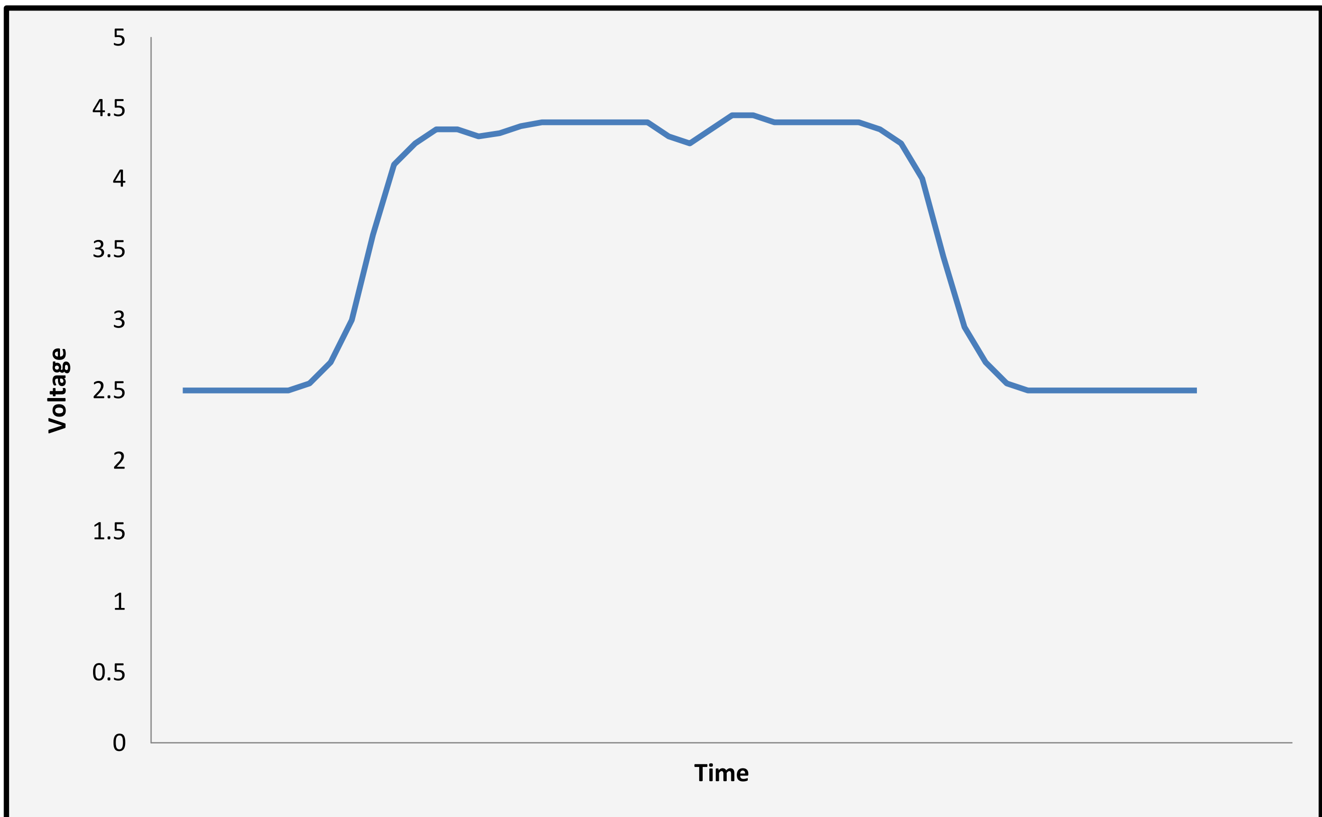

Figure 7- Sample Wave of Gyroscope during a quarter turn

Pattern matching for motion modules using accelerometer data was accomplished by first placing thresholds on the maximum and minimum median values. The thresholds are pre-calculated manually by analyzing the data. In the future this can be accomplished in real-time by calculating a running average. After setting the thresholds, finite state machines (FSM) were designed to walk through the expected transitions through the thresholds. If an unexpected transition occurred or the FSM timed out, the FSM would return to an initial state. For example, take the sample data in Figure 4, the finite state machine for the arm module will be looking for a positive spike above the maximum threshold, followed by a negative spike under the minimum threshold. The FSM will then make sure the incoming data stays below the maximum threshold before spiking positive once more.

In the case of the gyroscope, more parameters were necessary. The job of the Angle module was to detect at least a quarter turn and maintain state regarding the currently faced direction. The currently facing direction is held in a two-bit variable representing one of four values (Forward, Left, Right, and Back). Calculating quarter turns was accomplished by calculating the length of time the data spent above or below the thresholds. This time is then divided by a pre-determined time for a quarter turn. Finally, if the data was above maximum threshold the number of quarter turns was added to the stored variable. Else, the number of quarter turns was subtracted from the stored variable. This approach took into account the fact that adders would overflow values and thus simulate the "overflow" that occurs when someone makes a 360 degree turn.

The Interpreter

The function of the interpreter is quite simple. All it has to do is combine inputs to produce action enables. It takes in as inputs the enable signals from all the motion modules and outputs five action enables. For most of the outputs this is a one to one mapping. For example, the enable from the left arm module maps directly to the action enable for the "*a*" action. The only semi-complicated cases are the left and right actions. These actions combine the orientation given by the angle module with a walking enable. Thus, if the player is facing to the right and is walking then the *Right* action is enabled, similarly for the *Left* action. Figure 6 shows the logic diagram for this calculation.

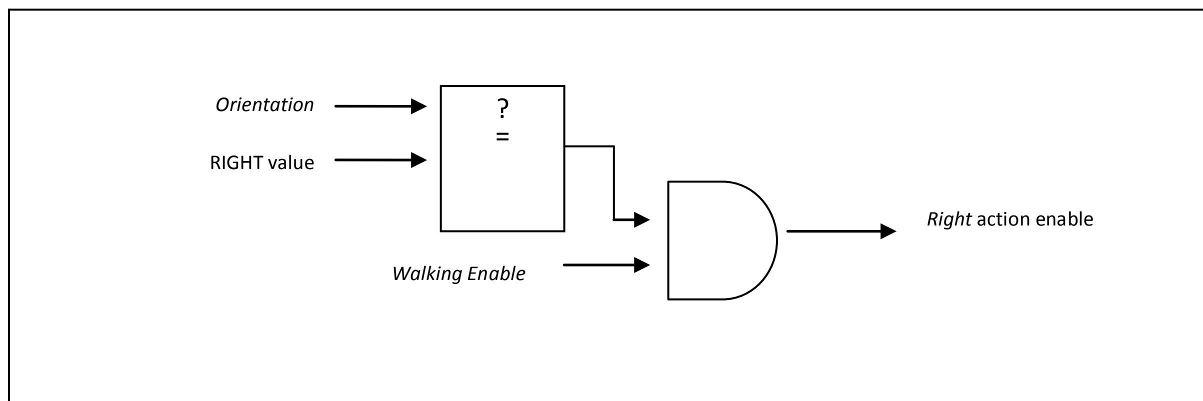

Figure 8 - Logic Diagram for Combining Orientation with Walking motion to produce *Right* action enable

Testing the Signal Processing Unit

Testing the Signal Processing unit was done in three parts. While being developed, testing was executed through simulations on ModelSim. The devised test benches were minimal tests designed to test correct behavior under ideal conditions. These simulations helped catch many simple errors in the interconnections. As the number of individual modules grew, the tests became integration tests designed to test the communications between modules.

Once most of the modules had been written testing moved onto the FPGA. At this time, the wireless unit was still not operation thus fake data was inputted through a set of eight switches on the FPGA. These eight switches were combined to form an eight-bit *data* value. Selection of *Type* was accomplished by scrolling through a selection menu with buttons on the FPGA. A final button press submitted the data. This method of inputting data was used to test the basic behavior of the unit. Enables from the motion modules were tied to LED's so one could detect when a motion module had recognized a module.

The final testing was done when the wireless unit was working. The final test was accomplished in the real world with sensors submitting data to the Signal Processing unit. Again, motion modules were tied to LED's for a visual indicator when motions were performed and recognized. This test involved much debugging through the use of the Logic Analyzer. Almost all signals that flowed through the signal processing unit were outputted to the Logic Analyzer were the team could look at the data and see where it was incorrect.

KeyMapper (By PJ Steiner)

The keyMapper module is responsible for providing an interface allowing the player to choose what key is mapped to each action produced by the signal processing module. The module continually provides six 16-bit scancodes to the PS/2 module which serve to parameterize its operation. They keyMapper is realized as a menu-driven interface which draws text on the dot display of the 6.111 labkit. The user can select an action he wants to edit, and then assign any of the lowercase letters a-z or the numbers 0-9 to that action.

The keyMapper is implemented using four subodules: a 'pulse' module, which converts positive edges to single clock cycle pulses (used to ensure that a single button press corresponds to just one action in the menu interface); a 'scancodes' module, which converts single bytes to PS/2 make scancodes; a finite state machine, which controls the module; and a 'dots' module which converts bytes to 40-bit numbers used to display text to the 6.111 labkit dot display.

The finite state machine governing operation of the keyMapper is a twelve-state FSM. There are two states corresponding to each of the six possible actions (up, down, left, right, a, and b). One is a state given the name of the action, in which the name of the action is displayed on the labkit dot display. The other is a 'set' state, in which the user is changing the mapping for the action currently displayed. The details of the FSM’s implementation are given below:

inputs: *reset, clock, up, down, left, right, enter*
outputs: *chars, uv, dv, lv, rv, av, bv*

Up: Display "UP" on the dot display. If enter is high, set the state to Up Select and the register *dispv* to *uv*.
Down: Display "DOWN" on the dot display. If enter is high, set the state to Down Select and the register *dispv* to the value of *dv*.
Left: Display "LEFT" on the dot display. If enter is high, set the state to Left Select and the register *dispv* to the value of *lv*.
Right: Display "RIGHT" on the dot display. If enter is high, set the state to Right Select and the register *dispv* to the value of *rv*.
A: Display "A" on the dot display. If enter is high, set the state to A Select and the register *dispv* to the value of av.
B: Display "B" on the dot display. If enter is high, set the state to B Select and the register *dispv* to the value of *bv*.

Up Select: If *enter* is high, set the state to Up and set *uv* to the value of *dispv*. Otherwise, if left is high, increase the value of *dispv*.
Down Select: If enter is high, set the state to Down and set *dv* to the value of *dispv*. Otherwise, if up is high, increment *dispv*; if, instead, down is high, decrement *dispv*. Ensure that *dispv* does not go below 0 or above its maximum value.
Left Select: If enter is high, set the state to Left and set *lv* to the value of *dispv*. Otherwise, if up is high, increment *dispv*; if, instead, down is high, decrement *dispv*. Ensure

that *dispv* does not go below 0 or above its maximum value.

Right Select: If *enter* is high, set the state to Right and set *rv* to the value of *dispv*. Otherwise, if up is high, increment *dispv*; if, instead, down is high, decrement *dispv*. Ensure that *dispv* does not go below 0 or above its maximum value.

A Select: If *enter* is high, set the state to A and set *av* to the value of dispv. Otherwise, if up is high, increment *dispv*; if, instead, down is high, decrement dispv. Ensure that *dispv* does not go below 0 or above its maximum value.

B Select: If *enter* is high, set the state to B and set *bv* to the value of *dispv*. Otherwise, if up is high, increment *dispv*; if, instead, down is high, decrement *dispv*. Ensure that *dispv* does not go below 0 or above its maximum value.

Essentially, when in one of the first states, the labkit is simply displaying the name of the state, allowing the user to edit the mapping for that action if he presses enter. When the use presses enter and the FSM enters a 'select' state, it begins displaying the value of *dispv*, which is added to the labkit display. When he presses enter again, this value is saved, and the user can select another action to edit the mapping for.

PS/2 Keyboard Module (By Ceryen Tan)

The PS/2 keyboard module is responsible for taking actions performed by the user and converting them into keyboard presses that are sent to a computer. The basic idea is to use these keyboard presses to control computer games. Because most computer games can be controlled by the keyboard, converting user actions into keyboard presses allows the user to play most computer games using just their body.

Actions are mapped to keys through the use of the KeyMapper module. There are six possible actions that the user can perform. For each of these six actions, the KeyMapper module outputs a 16-bit scan code that refers to a particular key on the keyboard. An action might, for example, be mapped to a 16-bit scan code of 0x21, which refers to the 'C' key on the keyboard. The PS/2 keyboard module makes use of these key mappings when sending keyboard presses to the computer. Because the KeyMapper module is programmable, it is possible to change the particular key that an action is mapped to.

The Signal Processing module outputs six wires corresponding to the six actions that the user can perform. Each wire signals high whenever a particular action is active. Active actions are treated as pressed keyboard keys, which causes keyboard presses to be sent to the computer. The output of the Signal Processing module therefore controls what keys are pressed.

Keyboard presses are sent to the computer using the PS/2 interface. The PS/2 interface was chosen as it is an extremely simple interface dedicated to just keyboards and mice. One alternative considered was to send keyboard data over the USB interface. However, the USB interface is considerably more complex without any real gains in functionality.

In order to send keyboard presses to the computer over the PS/2 interface, the PS/2 keyboard module implements significant portions of the PS/2 electrical and keyboard protocols. The PS/2

electrical protocol is a low-level protocol that specifies how data is electrically sent over the PS/2 interface. The PS/2 keyboard protocol is a high-level protocol that specifies what keyboard data to send. The distinction is that while the PS/2 keyboard protocol focuses on *what* data to send, the PS/2 electrical protocol focuses on *how* to send that data. The implementation of these two protocols allows the FPGA to pretend to be a keyboard.

The following two sections detail the PS/2 electrical and keyboard protocols. The section afterwards discusses how these protocols are actually implemented in on the FPGA.

PS/2 Electrical Protocol

The PS/2 interface is an electrical interface between a device and a host that allows for bidirectional transmission of data. The PS/2 interface feeds four lines between the device and the host: a +5 Volt power line, a ground, a serial *data* line, and a *clock* line. In the idle state, the *data* and *clock* lines are high. Both the device and host are capable of pulling these lines low. By properly manipulating the *data* and *clock* lines, data transmission can be made possible in both directions over the serial *data* line.

Data is transmitted through the PS/2 interface in 11-bit packets. Packets begin with a start bit that is always 0. This is followed by 8 data bits arranged with least significant bit first. The packet ends with a single parity bit (odd parity) and a stop bit of 1.

When the device wants to send a packet of data to the host, the device first generates a clock signal of 10-16.7 kHz on the *clock* line. The device then pushes packet bits onto the *data* line one bit at a time using the *clock* signal. This serializes the packet of data over the *data* line. New bits are pushed onto the *data* line when the *clock* signal is high. After a packet is successfully transmitted, *clock* signal generation is stopped. The timing involved in sending a packet of data from the device is illustrated in Figure **INSERT.**

The device is only allowed to send packets when *clock* has been high for at least 50 microseconds. The device must wait until *clock* is continuously high before sending packets.

The host can signal that it wants to send data to the device by pulling *clock* low for at least 100 microseconds, which inhibits PS/2 communication, then pulling *data* low and releasing *clock.* When this happens, the device is required to generate a *clock* signal. The host uses this *clock* signal to push bits onto the *data* line. The device can grab data bits from the *data* line on positive transitions of the *clock* signal. After the entire 11-bit packet is received, the device is required to acknowledge the data packet by pulling the *data* line low for a clock cycle. Afterwards, generation of the *clock* signal stops. The timing involved in receiving data from the host is illustrated in Figure **INSERT.** Note that the acknowledge pulse transitions in the middle of the last high *clock* pulse. This timing differs from the rest of the receive timing.

The host can choose to inhibit PS/2 communication at any given time by pulling *clock* low for at least 100 microseconds. When PS/2 communication is inhibited, the device is required to abort any data

transmissions that it is sending or receiving. If communication is inhibited while the device is in the middle of a transmitting a packet, the device must abort the transmission and prepare to resend the packet after inhibited communication ends. If the packet contains part of a multi-byte message, the device must prepare to resend the entire multi-byte message.

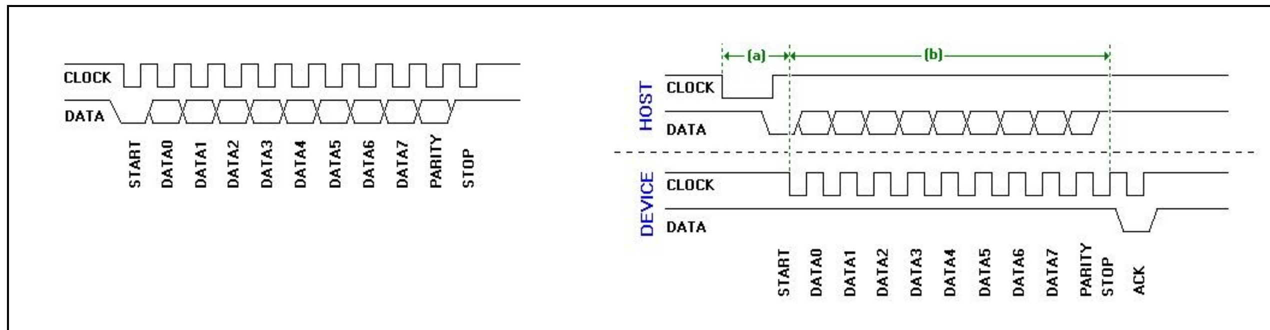

Figure 9 - a) Device-to-Host communication, b) Host-to-Device communication

PS/2 Keyboard Protocol

In the PS/2 keyboard protocol, key presses are sent using pairs of make and break scan codes. When a key is initially pressed, the make scan code for the key is sent to the computer using the above PS/2 electrical interface. When a key is released, the break scan code is sent. This make/break system allows the computer to keep track of which keys are pressed, even when multiple keys are pressed.

What complicates matters is that make and break scan codes can be variable in length. Most make scan codes are one-byte wide. Certain “extended keys” have make scan codes that are two bytes wide. Break codes also vary from two bytes to three bytes. Because these scan codes are variable length, the capability to transmit variable length messages is important in the implementation of the keyboard.

What makes the make/break system extremely complicated is something that is known as typematic behavior. When a key is initially pressed, it first results in a single make code sent to the computer. If they key is held down, then after some initial delay, the key becomes typematic and the make code is repeated with some frequency. This behavior explains why keys repeat when they are held down. The typematic behavior halts when another key is pressed or when the typematic key is released. When multiple keys are held down, the last key pressed becomes typematic. For example, holding down A, then holding down both A and C will cause a stream of A's followed by a stream of C's.

The delay between a key's initial make code and the key becoming typematic is known as the typematic delay. The frequency at which the key is repeated afterwards is known as the typematic rate. For this project, values of 0.25 seconds and 30 cps were chosen.

When the computer starts up, the keyboard is required to go through an initialization process. The keyboard is first required to send a self-test completion code of 0xAA. This is required to happen

within the first few seconds of computer startup. During startup the host may then send various commands to the keyboard. A number of these commands are shown below:

0xFF – Reset – Keyboard responds with “ack” (0xFA) and resets. Resetting causes the keyboard to resend the self-test completion code of 0xAA

0xF3 – Set Typematic Rate/Delay – This command is followed by a one-byte argument. The keyboard sends “ack” (0xFA) in response to both command and argument

0xF2 – Read ID – Keyboard responds with device ID of 0xAB, 0x83

0xEE – Echo – Keyboard responds with 0xEE

0xED – Set/Reset LEDs – This command is also followed by a one-byte argument. The keyboard sends “ack” (0xFA) in response to both command and argument.

A number of other keyboard commands are specified in the PS/2 keyboard protocol. However, these other commands are unlikely to be used in most situations. In addition, most of these commands simply require an “ack” (0xFA) reply. These other commands can be safely ignored.

Implementation

To hook the FPGA up to the PS/2 interface, a PS/2 keyboard cable is taken from a real keyboard. This cable provides the four PS/2 lines of *power*, *ground*, *data*, and *clock*. Note that in order for the FPGA to share the *data* and *clock* lines with the computer, an open collector circuit is required on the FPGA end of the two lines. The open collector circuit is what allows the FPGA to pull down the *data* and *clock* lines. The open collector circuit used is shown in Figure **INSERT**. To pull down the *data* and *clock* lines, the FPGA outputs a low signal to *data_out* and *clock_out*. At the same time, the status of the *data* and *clock* lines can be read through *data_in* and *clock_in*. For simplicity, the rest of this section does not refer to the *data_in, data_out, clock_in,* and *clock_out* lines directly, but instead refers to the *data* and *clock* lines. However, it is important to keep in mind that outputting to these lines requires *data_out* and *clock_out*, while inputs from these lines require *data_in* and *clock_in*.

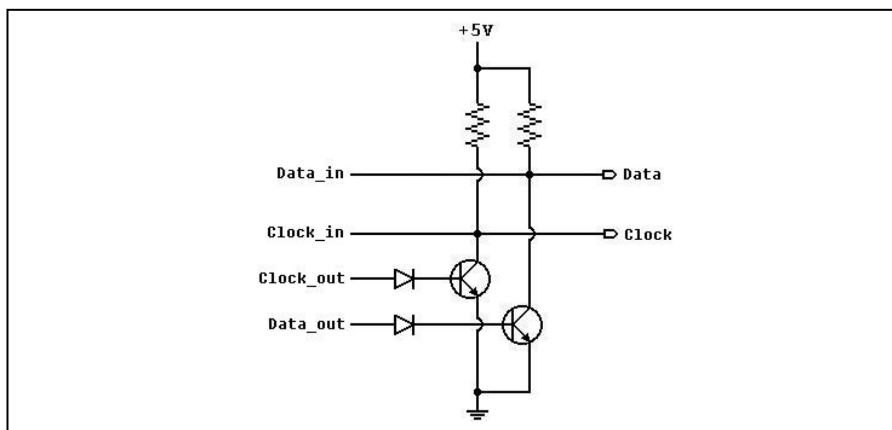

Figure 10 - Open Collector circuit.

The PS/2 electrical and keyboard protocols are implemented in Verilog to allow the FPGA to act as a keyboard. The two protocols are implemented in extremely modular form. A block diagram of the system is shown in Figure **INSERT**. The right half of the block diagram implements the PS/2 electrical protocol, while the left half implements the PS/2 keyboard protocol.

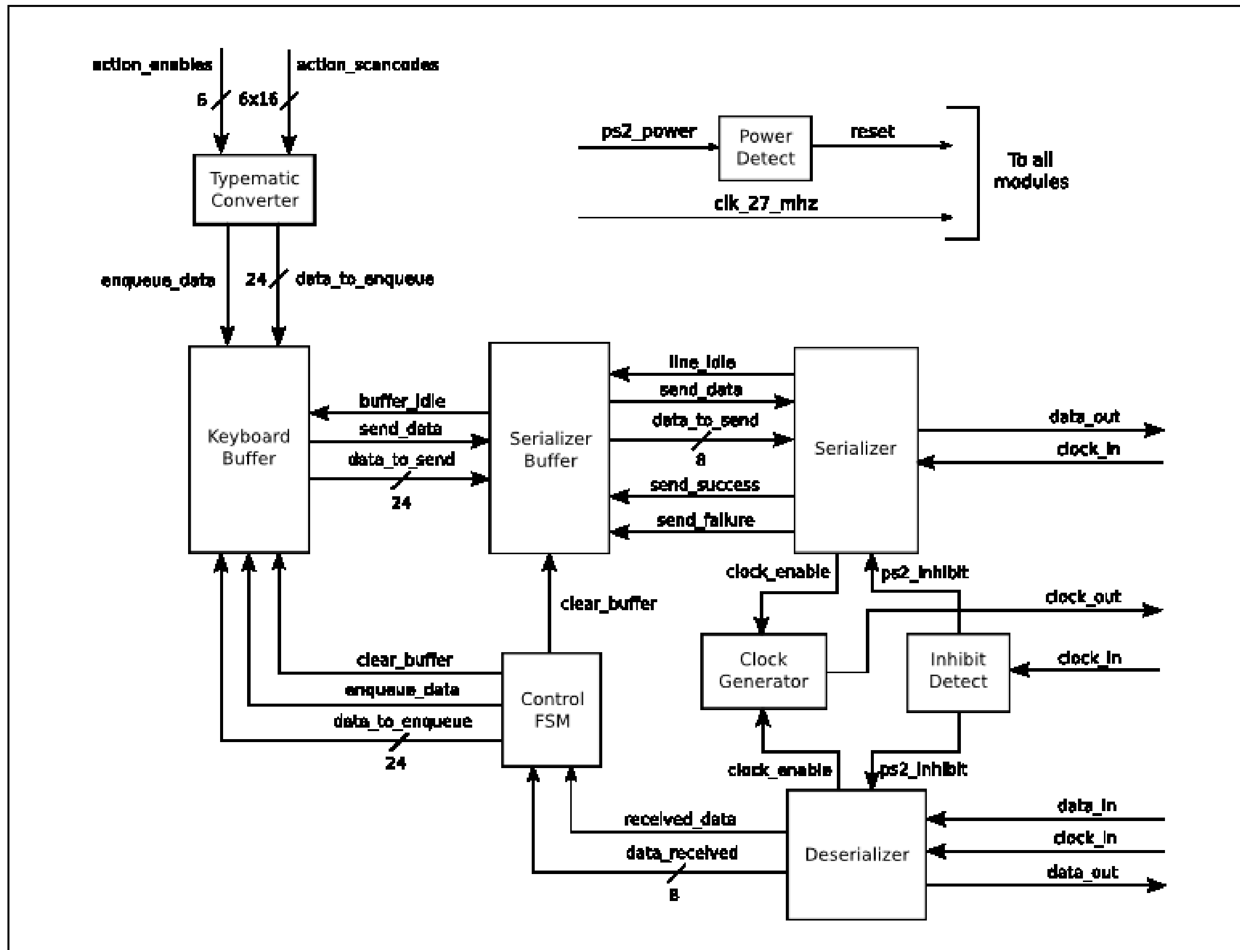

Figure 11 - Block diagram for the PS/2 Keyboard Module.

The PS/2 electrical protocol is handled in the Serializer and Deserializer modules. These modules handle the sending and receiving of data packets over the PS/2 interface. These two modules make use of two additional modules in order to operate: the Clock Generator and the Inhibit Detector. The Clock Generator generates the clock signals necessary to send and receive bits. The Clock Generator is idle by default and can be enabled by either the Serializer or Deserializer. The Inhibit Detector is responsible for detecting when the host has inhibited communication. When communication is inhibited by the host, the Inhibit Detector outputs a high *inhibit* signal. This signal is used by the Serializer module to abort sending data and by the Deserializer module to begin receiving data.

To send data using the Serializer module, the *send_data* line is held high for one clock cycle. This causes the Serializer to store the byte of data sent in through the *data_to_send* line and to start sending the data over the PS/2 interface. While sending data, the Serializer enables the Clock Generator by holding *clock_enable* high. If the byte of data is successfully sent across the PS/2 interface, then *send_success* is held high for one clock cycle. If communication is inhibited in the middle of sending the byte of data, then the Serializer aborts and sets *send_failure* high for one clock cycle.

Note that the Serializer module is only capable of sending data when the *clock* line is continuously high for more than 50 microseconds. The Serializer module monitors the *clock* line and asserts *line_idle* when *clock* is continuously high. This is used to signal when the Serializer is ready to send data. If *send_data* is asserted while *line_idle* is low, when the Serializer is not ready to send data, then the Serializer simply ignores the data input.

The Deserializer module works on its own in order to receive data packets from the computer. The module waits for an *inhibit* signal from the Inhibit Detector. After receiving this signal, it waits for the *data* line to go low and the *clock* line to go high. When this happens, the Deserializer enables the Clock Generator and begins capturing data bits on the positive edge transitions of the clock. After an entire packet is received, the Deserializer pulls the *data* line low for the acknowledge signal. The data that is received by the Deserializer is passed out through *received_data*, and *data_received* is asserted high to signal the received data.

Both of the Serializer and Deserializer modules are built using shift registers. FSMs are used to control the timings of the register shifts. Note that it not possible for the Serializer and Deserializer modules to be simultaneously active. In order for the Deserializer module to be active, communication would need to be inhibited. This would cause the Serializer to abort any data transmissions. Conversely, when the Serializer is active, then communication is not inhibited, which prevents the Deserializer from being activated.

The remaining modules implement the PS/2 keyboard protocol. The Typematic Converter, Keyboard Buffer, and Serializer Buffer modules work together to send scan codes over the PS/2 interface. The Typematic Converter takes in *action_enables* and *action_scancodes* from the KeyMapper and Signal Processing modules. It converts the enable signals into the make and break codes to send over the PS/2 interface. The make and break codes are fed into a Keyboard Buffer, which temporarily stores them until they can be sent. The Keyboard Buffer is necessary as make and break codes can often be generated faster than they can be sent over the PS/2 interface. Dropping any of these codes can cause major problems, and as such a buffer is necessary to temporarily store the scan codes while the PS/2 interface is busy. The Serializer Buffer allows for the sending of messages that are multiple bytes long. It is capable of sending messages that are variable in length. It also automatically resends entire messages when a single byte in the message is interrupted.

The Typematic Converter works by monitoring actions. When an action is initially enabled, the make code for that action is sent to the Keyboard Buffer. When an action is disabled, the break code for that action is sent to the Keyboard Buffer. Typematic behavior is implemented by monitoring how long

the last action enabled lasts. When an action lasts longer than the typematic delay, then the action becomes typematic and the make code for that action is repeatedly sent. Typematic behavior is interrupted by when the typematic action is disabled or when another action is enabled.

The Keyboard Buffer is capable of storing up to eight scan codes at a time. Scan codes are enqueued onto the Keyboard Buffer when they are generated. The Keyboard Buffer attempts to send these scan codes to the Serializer Buffer in the order that the scan codes are received. When the Serializer Buffer is seen to be idle, the Keyboard Buffer sends it new scan codes to transmit. Scan codes are stored as 24 bits as the largest single message that can be sent is a three byte break code. Scan codes that are shorter than three bytes are stored with zero padding attached to the end of the scan code.

The Serializer Buffer takes in messages of up to three bytes and feeds bytes into the Serializer one byte at a time. When the Serializer fails to send any one byte, then the Serializer Buffer restarts sending from the first byte. The Serializer Buffer outputs a *buffer_idle* signal to signal when the buffer is idle. This signal is used by the Keyboard Buffer so that the Keyboard Buffer knows when to send the Serializer Buffer new data. The Serializer Buffer automatically recognizes when messages are shorter than three bytes by looking for zero padding. The zero padding is not sent over the PS/2 interface.

The Control FSM module is responsible for sending the self-test completion code on computer startup, and for responding to messages received from the computer. When the computer initially turns on, the Control FSM attempts to send the self-test completion code (0xAA) approximately 500 milliseconds after power on. Afterwards, the Control FSM listens to any messages received by the Deserializer. The Control FSM is capable of responding to these messages by directly enqueuing data into the Keyboard Buffer. Note that when a command is received, the Control FSM clears both the Keyboard Buffer and the Serializer Buffer so that a response to the message can be sent immediately. The clearing of buffers is a behavior that is actually specified in the PS/2 keyboard protocol.

The last module is the Power Detect module. This module figures out when the computer has turned on by monitoring the *power* line from the PS/2 interface. When the *power* line transitions from low to high, the Power Detect module generates a *reset* signal that runs to all of the modules in the system. This reset signal clears all the modules, simulating a blank slate for the keyboard.

Testing

Testing for the PS/2 interface was found to be an amazingly difficult and tedious process. The primary cause for this was that the PS/2 interface did not appear to easily recover from errors. For example, sending invalid data to the computer often froze the PS/2 interface, necessitating a full restart of the computer. Early in the project when it was time to test sending of data to the computer, restarts were required every few minutes. The need to restart very often significantly slowed down all efforts to debug the system. While debugging itself was not particularly difficult, the process of debugging was drawn out to take up a significant amount of time.

This problem was exacerbated when it came time to test initialization code. The idea behind initialization was to allow the computer to properly recognize the FPGA as a PS/2 keyboard on startup. However, testing whether the computer properly recognized the FPGA as a keyboard also required full restarts of the computer. Every single change to the code required a full restart of the computer to test. Debugging also required frequent restarts of the computer as in order to collect new data on the logic analyzer, the computer would need to be restarted to capture the initial interaction between the FPGA and the computer. This frequent need to restart the computer made the debugging process extremely long and tedious.

What also contributed to the problem was the fact that the PS/2 interface was underspecified in a few key areas. For example, what exactly happens on computer startup is something that varies from computer to computer. In order to make the FPGA be recognized on the computer, figuring out what the computer did on startup was necessary. This was not possible for the longest time until a real keyboard was opened to determine how a real keyboard interacts with the computer. It was discovered with a real keyboard that the specifications for the protocol is actually conflicted in at least one place. Only after this was discovered was it made possible for the PS/2 keyboard module to work.

Conclusion

Overall, we think our project was successful. Although the controls were a bit touchy, it was very possible to play video games with our system. In fact, in our short time testing the complete system, various players were able to complete the first level in Super Mario World. In terms of integration problems, the simple interfaces we designed between modules allowed us to quickly debug problems that occurred when integrating. Although we had some difficulties in equipment shortcomings, such has non-functioning 6.111 labkits and missing logic analyzers, the distribution of work allowed us to time multiplex the use of such equipment.

Acknowledgments

We would like to thank Gim P. Hom and Dave our T.A. for their assistance is figuring out how to approach and solve several of the design problems we faced. We would also like to thank Wendi Li, the 6.111 L.A., for entertaining our group even though our lab space was at the extreme end of the lab.