Newtonian N-Body Simulator

Michelle Teh, Michael Witten
{mpteh,mfwitten}@mit.edu

May 17, 2007

Massachusetts Institute of Technology
6.111 Introductory Digital Systems Laboratory

Professor Anantha Chandrakasan
TA: David Wentzloff

Abstract

The objective of the project is to create a simulation of objects under the influence of gravitational force. These processes are performed part in parallel and part in serial. The objects are represented as circles on the double buffered VGA display. A user interacts with this system using a PS/2 Mouse; the user can add and track objects or scale the display's coordinates. Many of the components of this project are completed, but the system has yet to be integrated.

Contents

List of Tables

List of Figures

1 Overview

The *N-body problem* involves calculating the results of the physical interactions between n entities. The goal of the N-body problem is to calculate the state of the entities at any given time after the initial conditions have been set.

In our case, these entities are spheres of uniformly distributed matter, and we are trying to calculate the results of the gravitational forces between them. This specialization of the problem is known more familiarly as the *Newtonian* N-body problem, as it employs Newton's law of gravitation.

Ideally there would be some function that calculates the state of the matter at any input time t; however, no such *closed-form* solution is known for the general case, so that the problem can only be solved via iterative approaches, such as the *Euler Step Method*s and their ilk. Unfortunately the iterative method scales very poorly: Each of the n entities interacts with the other $n-1$ entities, for a total of $n(n-1) = n^2 - n$ interactions. Thus the problem grows according to $O(n^2)$.

In practice, the number of interaction calculations can be approximately halved by exploiting Newton's law of equal but opposite forces. Besides fancy specializations and other tricks, the only method to speed up the calculations is to exploit the property of superposition: With the assumption that the interaction between any 2 bodies is independent of any other 2-body interaction, all of the interactions can be calculated in parallel; such a propertly lends the N-Body Problem nicely to a solution in hardware, such as on an FPGA. This project has attempted to do just that.

However, parallelism of calculation involves a balance beween space and time. The more that is done in parallel, the more hardware that is required. For instance, using a Virtex II (family XC2V6000, speed grade 4, in a BF957 package) to provide a pipeline for each of the $n(n-1)$ interactions would only allow for calculating just the accelerations between roughly 3 spheres.

We therefore found it necesary to compromise pure parallelism for an increase in usable resources by utilizing one pipeline for all interactions; this necessarily introduces a dependency on shared hardware, such as when accelerations must be accumulated, as described in Section 2.2.1.

With the ability to calculate successive iterations of the matter's properties, we can then display the results in any number of ways. We have decided to represent each *frame of data* as a visual representation of the physical scene by drawing circles to represent the objects. Given the ability to visualize the data, we have decided to include user interaction via the ability to manipulate the positions of objects through a PS/2 mouse input.

While we were not successful in creating a final system, many of the necessary and more troublesome parts have been designed. This report details those results. A simplified block diagram of the entire system is given in Figure 1.

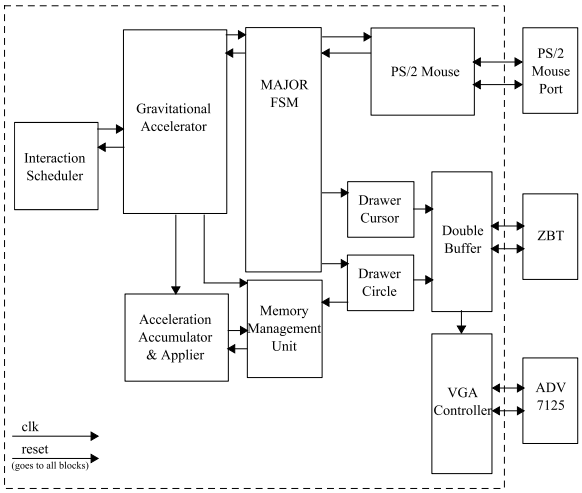


Figure 1: The High-Level Block Diagram.

2 Modules and Implementation

2.1 Mathematics

The modules that implement arithmetic are designed in a way that promotes flexibility of number representation and maintainability of Verilog code. In particular, all real number operations are either binary or unary, taking either 2 arguments or 1 argument, respectively. Consequently, any module that adheres to this interface can be used as a drop-in replacement, provided it is used with modules that employ the same underlying number representation.

Indeed, the infrastructure for introducing such a variety of formats has already been implemented and allows a designer to switch the entire system between formats with a few macro definitions, allowing for a much more flexible design process. At any time, arithmetic for the entire system can be converted between floating point, fixed point, or something as exotic as binary-coded decimal numbers, all transparently to any higher-level modules; this is important when the usage of FPGA resources is not fully determinable during the design phase, thus reducing the magnitude of an entire design iteration.

Moreover, the reuse of abstracted interfaces via heavy use of Verilog preprocessor directives provides central points from which entire subtrees of modules can be appropriately modified, thereby increasing maintainability and consistency. For instance, it was discovered that the Xilinx IP for floating point numbers use a different interface than the number modules of this project. However, since all arithmetic operators are created on the fly via macro expansion upon Verilog compilation, only 2 lines of one file were necessary to change in order to properly connect ports for all uses of our arithmetic modules.

Thus far floating point arithmetic has been implemented by the modules **FloatAdd**, **FloatSub**, **FloatMul**, **FloatDiv**, **FloatNeg**, and **FloatSqrt**, which are further built on either IP core or simulatable Verilog code, depending on the testing environment.

Those modules are transparently used by a higher level of modules: **RealAdd**, **RealSub**, **RealMul**, **RealDiv**, **RealNeg**, and **RealSqrt**, which can of course transparently use other number representations.

These modules are then used to build representation independent vector operations: **VectorAdd**, **VectorDivReal**, **VectorDot**, **VectorLength**, **VectorMulReal**, **VectorNeg**, **VectorNormalize**, and **VectorSub**.

It is assumed that these modules can be presented with new data on every cycle. The latencies are automatically accumulated and bubbled up through the abstractions, so that the entire system up to the highest levels can automatically reconfigure itself upon changes in the lowest levels, reducing and eliminating tedious and error-prone refactoring of Verilog code by hand[1].

With the modules for mathematics at hand, it is possible to create more elaborate calculations for the purposes of our physics simulation.

2.2 Gravitational Accelerator

The **GravitationalAccelerator** module produces the accelerations due to the gravitational interaction between 2-bodies. These accelerations are calculated according to Newton's inverse square law:

$$F = G\frac{m_1 m_2}{r^2} \tag{1}$$

The module takes as inputs the 2 objects' masses (as real numbers) pre-multiplied by the Gravitational Constant and their centers of gravity (as vectors in an absolute Cartesian coordinate system). The appropriate 2-body system accelerations are produced after the required latency. The block diagram is given in Figure 2 and the internal pipeline of calculations is given

[1]The vector operations are not as elaborately macro expanded, as they already constitute modules at a high enough level to enjoy separation from the gritty details of the lower levels.

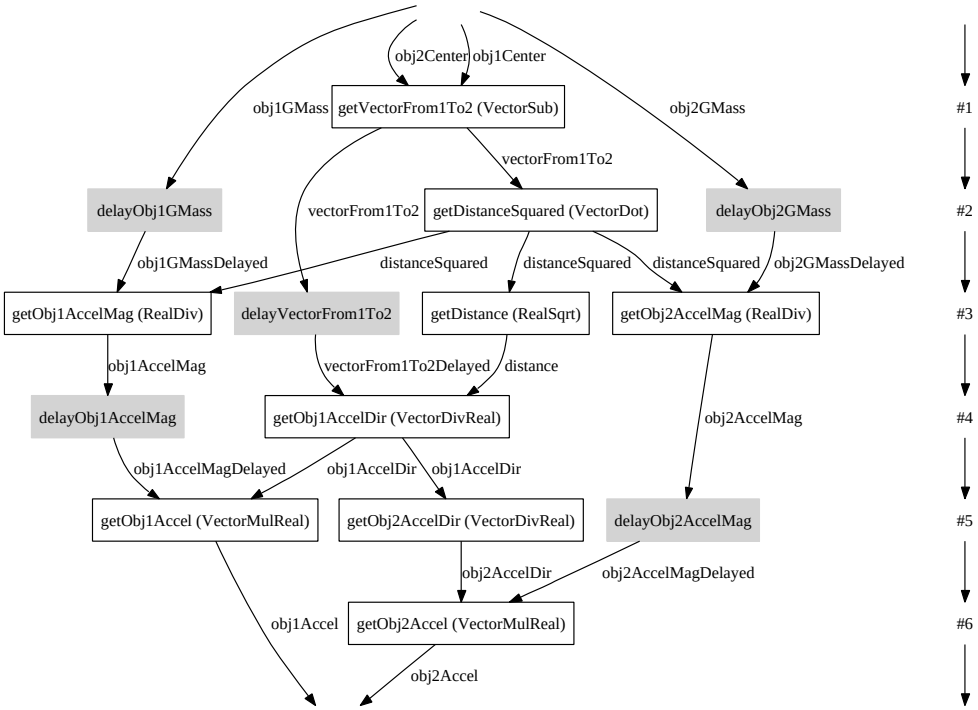


Figure 2: The Interaction Scheduler Module.

2.2.1 Interaction Scheduler

Although the FPGA has 33,000 slices, it is still insufficient to perform all the necessary calculations in parallel, since it takes approximately:

$$\frac{1}{2}(n^2 - n) \tag{2}$$

In order to deal with this constraint, the hardware is reused with careful pipelining procedures so that the results are produced every cycle after a sufficient latency. An InteractionScheduler module is designed to accurately send objects into the pipeline such that there are no two interactions that are calculated with the same object at the same time. The patterns were observed and several formulae were produced.

To illustrate the formulae produced, lets consider the interaction patterns for 10 objects shown in Table 1. Using Equation 2, 10 objects would involve 90 calculations per unit time. Since each stage of a pipeline cannot process the same object twice, only 5 pairs of interactions can be calculated per stage, thus, 9 stages are required to calculate all the 2-body interactions of 10 objects in the gravitational field. Each object has to be carefully paired in each stage to ensure no calculation duplication in later stages and no clashes in the pipelined accumulator stages. If we are given that there is 3 pipelined accumulator stages for the 10 objects, each number can only be sent in to the pipeline again after at least 3 clock cycles.

Table 1: Stage Pipelining for 10 objects.

Primary	Secondary
0:	1 2 3 4 5 6 7 8 9
1:	2 3 4 5 6 7 8 9
2:	3 4 5 6 7 8 9
3:	4 5 6 7 8 9
4:	5 6 7 8 9
5:	6 7 8 9
6:	7 8 9
7:	8 9
8:	9

Table 2 shows explicitly the pairs of interactions that are sent to the pipeline every cycle such that there are no conflicts in the acceleration accumulation stages or duplication of interaction pairs in later stages. A consistent pattern can be observed:

Table 2: 10-object Interactions.

Stage 9	Stage 8	Stage 7	Stage 6	Stage 5	Stage 4	Stage 3	Stage 2	Stage 1
(0, 9)	(0, 8)	(0, 7)	(0, 6)	(0, 5)	(0, 4)	(0, 3)	(0, 2)	(0, 1)
(1, 8)	(1, 7)	(1, 6)	(1, 5)	(1, 4)	(1, 3)	(1, 2)	(3, 8)	(2, 8)
(2, 7)	**(4, 9)**	(2, 5)	**(3, 9)**	(2, 3)	**(2, 9)**	(4, 8)	**(1, 9)**	(3, 7)
(3, 6)	(2, 6)	(3, 4)	(2, 4)	(6, 8)	(5, 8)	(5, 7)	(4, 7)	(4, 6)
(4, 5)	(3, 5)	**(8, 9)**	(7, 8)	**(7, 9)**	(6, 7)	**(6, 9)**	(5, 6)	**(5, 9)**

For s = (current stage number) and k = (max number of stages),

$$Interaction\ pair = (x, y)\ s.t.\ x + y = s,\ for\ x \leq= s,\ x + y = s + k\ for\ x \gg= s \tag{3}$$

$$Interaction\ pair = (x, k)\ s.t.\ x = (s + k)/2\ for\ s\ odd,\ x = s/2\ for\ s\ even \tag{4}$$

The placement of the interaction pair (x,k) in the pipelined stage has to be carefully thought out to prevent conflicts in the acceleration accumulation stages. In this example, the (x,k) in the odd stages are placed last whereas in the even stages, the (x,k) are placed third from the top.

These formulae are easily implemented as counters in the Interaction Scheduler module since the numbers in the pattern is seen to consistently decrement or increment except for the anomaly, (x,k). The ports of the module are shown in Figure 3.

Figure 3: The Interaction Scheduler Module.

2.3 Data Representation

The n-bodies each have its own data (Table 3) that is placed in nine separate dual port Block RAMs to achieve parallel access of the data such that the gravitational force calculations among objects can be done in parallel. These variables are represented as single-precision floating numbers to support a wide range of magnitude, i.e. the size of a small pebble to the size of the earth. The data BRAMs are instantiated and

handled by the Memory Management Unit module that allows the acceleration accumulator, gravitational accelerator, and the drawer circle modules access to the data.

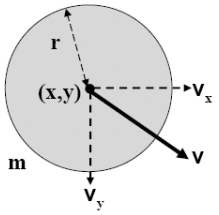


Figure 4: Data Represented as a Circle.

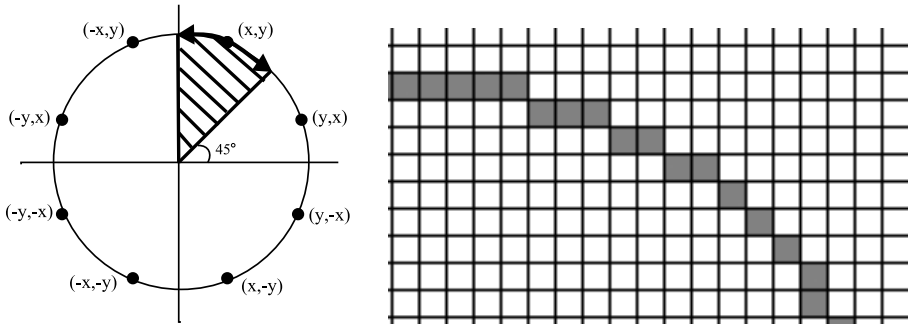
Table 3: Data stored in Block RAMs.

m	mass
r	radius
(x, y)	position
(v_x, v_y)	velocity
(a_x, a_y)	acceleration

2.3.1 Circle Drawing Algorithm

In this project, the n-bodies are represented as circles, drawn using the well established *Bresenham Circle Drawing Algorithm* that involves only simple integer addition and bit shifting, avoiding other costly operations such as general multiplication, division, square roots, or trigonometric functions. The algorithm produces circles of any (non-negative) integer radius composed of points with integer coordinates. Moreover, the generated circle is guaranteed to contain no gaps due to the use of symmetry; its outline is always continuous.

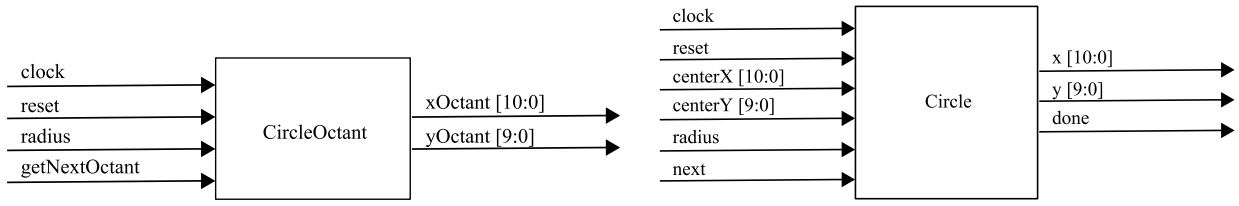
To produce a circle, the locations of points are calculated along only one octant, and a circle's perfect symmetry is used to extrapolate the points of every other octant, as illustrated in Figure 5. In this case, points in one octant are obtained by tracing a path beginning with the point $(0, r)$. If the next point is to the right of the circle, then the trace is moved down in order to bring the trace back onto the circle. This produces a series of steps that are reflected across the abscissa, ordinate, and the line $y = x$ (Refer to McMillan's for a derivation formulas) [3].



(a) The Circle Octant. (b) State Transition Diagram.

Figure 5: Bresenham's Circle Drawing Algorithm.

The **CircleOctant** module designed in this project implements Bresenham's octant tracer. The **Circle** module produces the rest of the points based on symmetry. The ports of these module are shown in Figure 6. A **CircleOctant** is instantiated within the A **Circle**. The **CircleOctant** produces *xOctant* and *yOctant* for a circle of size *radius* when its *next* is asserted. The **Circle** module produces a new point on the circle when its own *next* signal is asserted. New points are calculated until it determines that the circle is complete (when $xOctant > yOctant$), in which it pulses its *done* signal with the last new point.



(a) The Circle Octant Module. (b) The Circle Module.

Figure 6: The Circle Drawing Modules.

2.3.2 Drawer Circle Module

The **DrawerCircle** module is responsible for converting a **Circle**'s points from the Cartesian coordinates (x, y) to a linear memory address in a pixel buffer (see Section 2.4). Given a screen width of w pixels, the address is calculated as follows:

$$\text{address} = x + wy; \tag{5}$$

Any points (x, y) that do not fall within the visible screen are simply not written to the buffer, as determined by controlling a $writeDisable$.

The ports of the Drawer Circle module are shown in Figure 7.

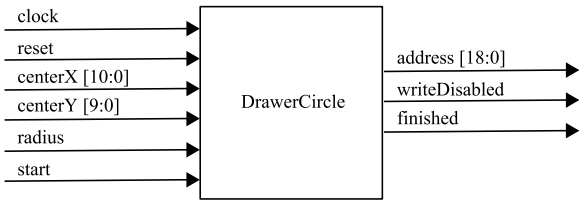


Figure 7: The Drawer Circle Module.

2.4 Zero Bus Turnaround (ZBT) SRAM Interface

The **ZBT** module (see Figure 8) defines an interface for interacting with a subset of the actual ZBT hardware's features, abstracting away unnecessary details: The ZBT is used simply to read or write into a given location; if *read* is asserted, the ZBT read from the supplied address, otherwise, the supplied data is written to that address.

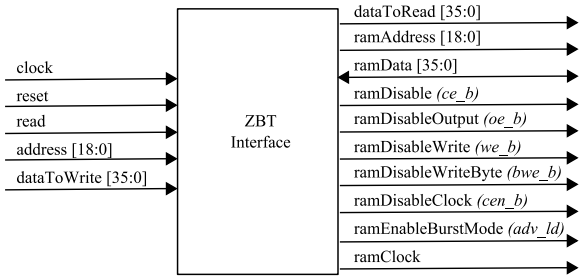


Figure 8: **ZBT** Module.

2.4.1 Double Buffering

The 2 ZBTs of the Labkit are used to implement a double-buffered video RAM. While one of the ZBTs is being read to produce VGA output, the other can be used to store data for the next frame to be displayed.

This approach not only allows for drawing operations that involve random access of pixel data, but also operations that take arbitrary numbers of cycles.

In particular, the use of each ZBT is arbitrated by a **BufferSwitcher** module (see Figure 9), which provides a general interface for sharing any buffer that implements the **ZBT** module user interface. Moreover, a **BufferSwitcher** can transparently moderate the use of buffers between two separate clock domains triggered by $clkA$ and $clkB$, provided that the **BufferSwitcher**'s $clk > clkA, clkB$.

Essentially, when systems A and B agree that a switch can be performed, the moderating **BufferSwitcher** swaps the connections and then informs each system with a pulse synchronized to its respective clock. This way, each system can operate without any knowledge of the other system; each system need only communicate with the **BufferSwitcher**. While the handshaking protocol may waste cycles, especially if the 2 systems are asynchronous, this extra generality provides a means running drawing operations must faster than that the VGA output operations. Furthermore, the handshaking necessarily eliminates the possibility of erroneously accessing the buffer.

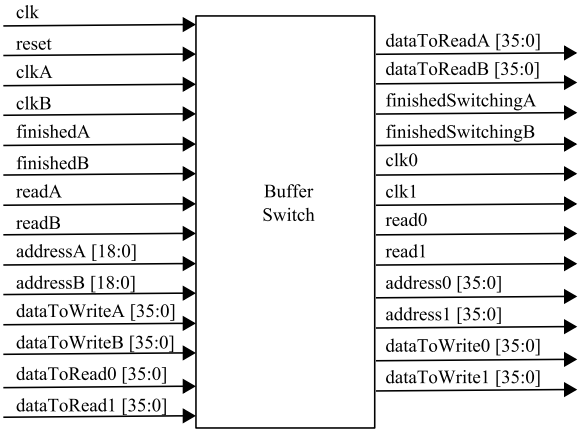


Figure 9: **ZBT** Module.

2.5 User Interface: PS/2 Mouse

The PS/2 Mouse provides an interface for the user to interact with the system. The user can add or track objects in the gravitational field by holding down the left and middle buttons. As soon as the middle button is released the radius is calculated and the circle is drawn on screen. This circle can be tracked until the left button is released. The user can also choose to scale the display coordinates of the screen with the right button.

Unfortunately, the user interface has yet to be implemented. Nonetheless, the PS/2 interface between the mouse and the FPGA has been completed. A user can move the mouse and observe a moving 8 by 8 pixel cursor on screen. The subsequent sections discuss the protocols and implementation of the PS/2 Mouse.

2.5.1 PS/2 Mouse Protocol

The PS/2 mouse utilizes a bidirectional synchronous serial protocol. The clock and data lines, shown in Figure 10, are active high if neither the mouse nor the host (the FPGA) pulls them low.

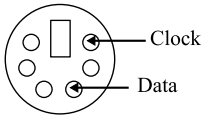


Figure 10: PS/2 Socket.

The protocol for communication between the host and the mouse is shown in Table 4. The mouse transmits its data whenever the bus state is *Idle* whereas the host has to first enter the *Communication Inhibited* state, and then the *Host Request-to-Send* state before sending its data to the mouse. The host enters these states to signal the mouse to begin generating a clock pulse of 10 - 16.7 kHz frequency [1].

Table 4: PS/2 Bidirectional Synchronous Serial Protocol.

Data Line	Clock Line	Bus State
High	High	*Idle*: The mouse is allowed to transmit data.
High	Pulled Low	*Communication Inhibited*: The host halts the mouse from transmitting data.
Pulled Low	High	*Host Request-to-Send*: The host signals the mouse that it wants to send data.

There are timing constraints, occurring in the *Communication Inhibited* and *Host Request-to-Send* states, that have to be satisfied before the host sends commands to the mouse. In the *Communication Inhibited* state, the Clock line has to be pulled low for at least 100 μs before the Data line is pulled low. After the Data line is pulled low in the *Host Request-to-Send* state, the Clock line is held low for an additional 5 μs before the Clock line is released. These timing constraints are essential because of the mouse's slow clock frequency with respect to the FPGA's system clock, otherwise these signals would be missed by the mouse.

Data is transmitted one byte at a time with each byte contained in a frame of 11 bits (Table 5). The host reads the data sent by the mouse on the *falling edge* of the clock signal whereas the mouse reads the data sent by the host on the *rising edge*.

Table 5: The 11-Bit-Frame.

Bit(s)	Significance
0	*Start bit*: Always represented by 0.
1 to 8	*Data bits*: The least significant bit is transferred first.
9	*Odd Parity bit*: The whole frame must have an odd number of '1'.
10	*Stop bit*: Always represented by 1.

A standard PS/2 mouse keeps track of mouse movements from its current location and transmits the information along with the button clicks in a 3-byte-packet [2]. These movements consist of the X-movement and the Y-movement that are each represented as a 9-bit two's complement value. The signed bit is sent in the first byte, as shown in Table 6. Since the 9-bit two's complement value can only represent numbers ranging from -255 to 255, overflow flags for X and Y are transmitted in the first byte.

Table 6: PS/2 Mouse Three-Byte Information.

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Byte 1	Y overflow	X overflow	Y sign bit	X sign bit	Always 1	Middle button	Right button	Left button
Byte 2	X movement							
Byte 3	Y movement							

The PS/2 Mouse needs to be initialized and set to enable Data Reporting before it is ready to stream the 3-byte movement data packets. The initialization process begins with the host sending a *Reset* command. After the mouse acknowledges the receipt of the *Reset* command, it performs a self-diagnostic test known as the *Basic Assurance Test* (BAT) and sets the default values for the following:

- Sample Rate: 100 samples/second
- Resolution: 4 counts/millimeter
- Scaling: 1:1
- Data Reporting Disabled

Upon a successful BAT, the mouse sends the BAT completion code and its Device ID. After the mouse sends the Device ID, the mouse is by default in *Stream* mode. The mouse continuously tracks the cursor movement and button clicks in *Stream* mode. However, the mouse does not issue these tracked movements until it acknowledges that the host has sent a *Data Reporting Enable* command to the mouse.

2.5.2 PSMouse Module

The **PS2Mouse** module is an abstraction of the underlying communication between the mouse and the FPGA. The ports of the module are shown in Figure 11. At every *next* signal, given that the *dataAvailable* signal is high, the **PS2Mouse** module outputs the cursor's x and y locations as well as the *buttons* that have been clicked by the user. A First-In-First-Out (FIFO) buffer, with the depth of 8, is implemented within this module to store decoded mouse movement data packets that have not yet been processed. The $bufferFull$ is set to high as soon as the FIFO is completely filled.

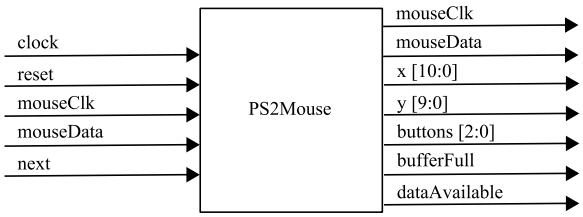


Figure 11: **PS2Mouse** Module.

This underlying communication between the mouse and the FPGA is divided into several modules that the **PS2Module** instantiates:

PS2Sender	sends commands to the mouse.
PS2Receiver	receives packets from the mouse.
PS2Decoder	decodes the packets.
PS2MouseController	arbitrates the flow of data between a **PS2Sender**, a **PS2Receiver**, and a **PS2Decoder**

The connections between these module are shown in Figure 12.

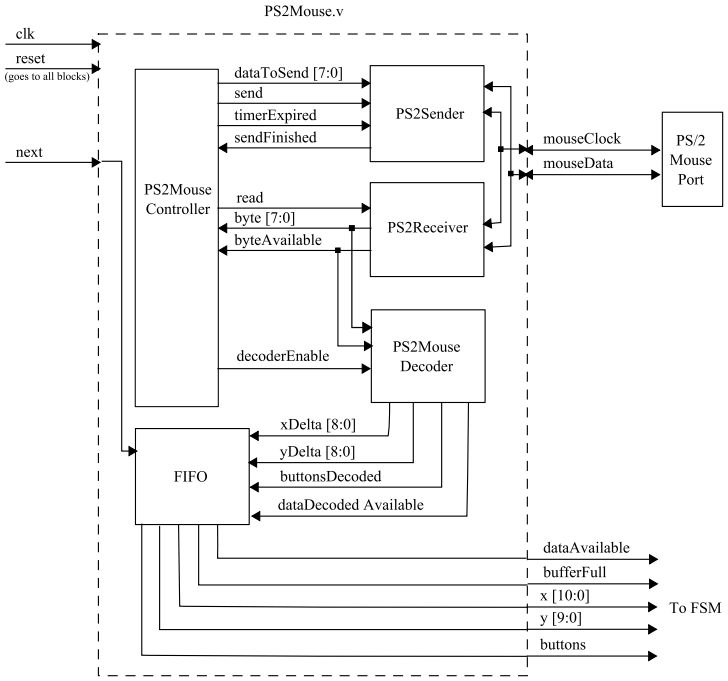


Figure 12: PS/2 Mouse Block Diagram.

These modules are described in the following sections.

2.5.3 PS2MouseController Module

The **PS2MouseController** module acts as a major Finite State Machine (FSM), mediating the PS/2 bus between sending commands to the mouse and receiving packets from the mouse. It is responsible for initializing the mouse and establishing the *Data Reporting Enabled Stream* mode. The controller also instantiates a Timer that maintains the timing constraints required when the FPGA sends commands to the mouse (refer to Section 2.5.1) and ensures the mouse responds to the issued commands in a timely fashion.

The ports to the **PS2MouseController** module are shown in Figure 13. The *send* and *read* signals are issued to notify either the **PS2Sender** or the **PS2Receiver** to begin sending or receiving data from the Data line. It sends data through the *dataToSend* port and receives data through the *dataToReceive* port when *byteAvailable* is high. The *sendFinished* is high when the mouse recognizes that it has received all 11 bits of a frame. When the initialization process is completed, the *decoderEnable* is set to high to notify the **PS2MouseDecoder** to begin extracting the information from the packets.

Figure 13: **PS2MouseController** Module.

The state transition diagram of the **PS2MouseController** is shown in Figure 14. There are a total of nine states to initialize the necessary timers, send the *Reset* and *Data Reporting Enable* commands, and wait for a mouse response (refer to Section 2.5.1). The *decoderEnable* is set and kept high in the STREAM state.

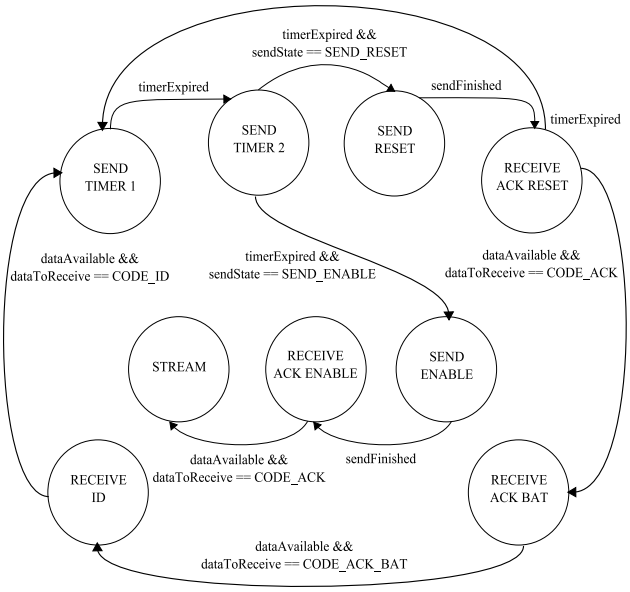
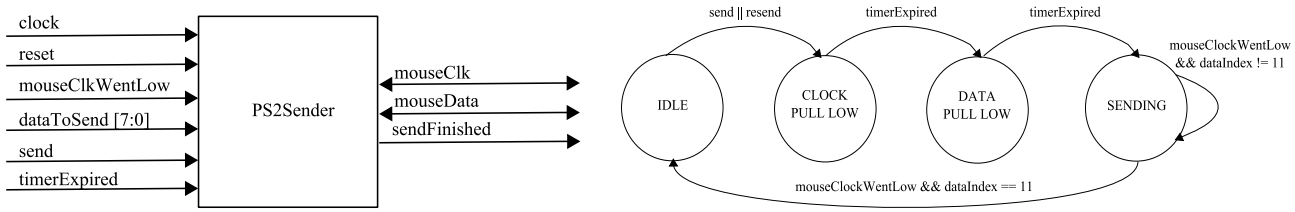


Figure 14: **PS2MouseController** State Transition Diagram.

2.5.4 PS2Sender Module

The **PS2Sender** module is responsible for sending the commands exerted by the **PS2MouseController** module when the *send* signal is high. Before it begins to send the commands (*dataToSend*) bit by bit, it pulls the Clock line (*mouseClk*) low for 100 μs in the PULL_CLK_LOW state (Figure 15). In the PULL_DATA_LOW state, the Data line (*mouseData*) is pulled low and after approximately 5 μs the Clock line is released. This transitions to the SENDING state when the mouse begins pulsing the Clock line. Since the mouse reads the sent data at the *rising edge* of the clock, the bit being sent is changed at the *falling edge* (*mouseClkWentLow*). The *sendFinished* signal is pulsed high when the mouse pulls the Data line low for a clock cycle upon receiving all 11 bits of a frame. The *timerExpired* signal sent from the **PS2MouseController** is responsible for timing the 100 μs and 5 μs constraints.

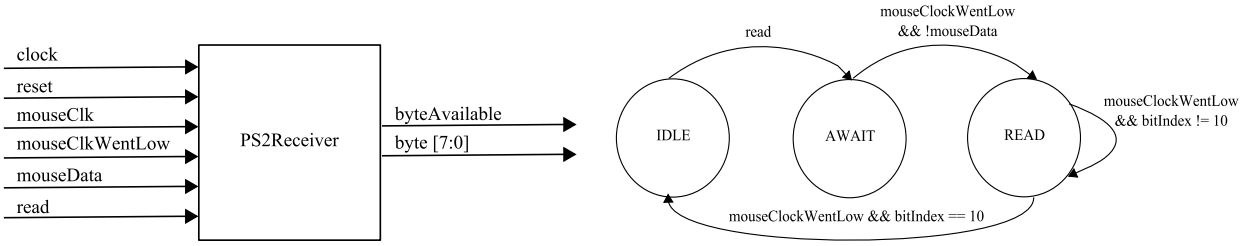


(a) Module. (b) State Transition Diagram.

Figure 15: **PS2Sender** Module.

2.5.5 PS2Receiver Module

The **PS2Receiver** module is responsible for retrieving data from the Data line (*mouseData*) when the **PS2MouseController** sets the *read* signal high (Figure 16). The bit is read from the Data line at every *falling edge* of the clock (*mouseClkWentLow*) and after 11 bits are read, the *byteAvailable* is pulsed high with the received data bits on the *byte* port. The *byte* is sent to the **PS2MouseController** during the initialization process, but sent to the **PS2MouseDecoder** to be decoded during the Stream mode.



(a) Module. (b) State Transition Diagram.

Figure 16: **PS2Receiver** Module.

2.5.6 PS2MouseDecoder Module

The **PS2MouseDecoder** module receives the *byte* from the **PS2Receiver** when the *decoderEnable* and the *byteAvailable* signals are high (Figure 17). The module keeps track of the byte number and outputs *deltaX*, *deltaY*, and *buttons* as soon as it receives the third byte. It sets the *dataAvailable* signal high to notify the PS/2 Mouse module that then converts *deltaX* and *deltaY* into specific *xy* locations.

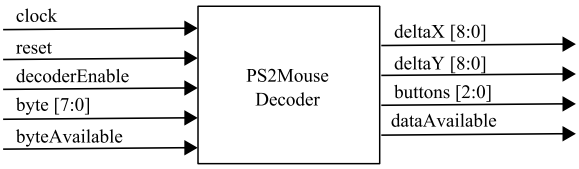


Figure 17: **PS2MouseDecoder** Module.

3 Testing

The modules implemented are rigorously tested in the following simulation environments:

- ModelSim SE (Special Edition)
- Xilinx ISE Simulator Lite
- Icarus Verilog

Refer to the Appendix for the simulation waveforms, testbenches, and verilog implementation of the project modules.

4 Conclusion

This project has further enlightened us about timing constraints in hardware. Dealing with the PS/2 and the ZBT has been a great challenge because of the timing constraints that needed to be met due to the different clock domains. Besides these interfaces, pipelining the gravitational calculations and creating flexibility between the use of floating or fixed point numbers required much detailed thinking and planning. The project could have been more successful had there been an additional person to help out with the workload or simply more time.

References

[1] A. Chapweske, *The PS/2 Mouse Protocol*,
http://www.computer-engineering.org/ps2protocol (May 2002)

[2] A. Chapweske, *The PS/2 Mouse Interface*,
http://www.computer-engineering.org/ps2mouse (Apr 2002)

[3] L. McMillan, *Circle Drawing Algorithm*,
http://www.cs.unc.edu/ mcmillan/comp136/Lecture7/circle.html (Sept 1996)