GPS Tracking System

Design Report
Final Project
6.111 / Massachusetts Institute of Technology
December 12, 2006

Alexander Valys

Abstract

A mobile GPS data logger and visualization system was implemented on a Digilent Nexys board, and the 6.111 labkit. Position data is read from a third-party GPS receiver, stored on a third-party Flash ROM, and then rendered in 2D and 3D on a VGA display through a generalized coordinate transformation and rendering system.

Most of the implementation goals were achieved. However, due to unexpected difficulties in a variety of areas, notably debugging the 3D transformation module, the final user interface is not as polished as was originally intended.

Contents

List of Figures

List of Tables

1 Overview

The goal of this project was to create a device that is capable of keeping track of where it's been, and displaying that information to the user. Because of the difficulties involved in implementing a GPS chip from scratch, it was decided to use one from a third party, and concentrate on the logging and visualization components for this project.

While the initial plan was to implement the entire system on a Digilent Nexys board, it quickly became apparent that the Nexys did not posess enough resources or I/O capability to permit this. So, the project was implemented in two parts. A Nexys board was used to implement the data logging component, reading data from the GPS chip and storing it on a Flash ROM, and the 6.111 labkit was used to generate the data visualizations.

1.0.1 Defaults

Unless specified otherwise, all sequential transitions occur on the rising edge of the clock, and all FSMs are Moore-type.

1.1 Mobile Data Logger

The mobile data logger has very little user interface: the user simply powers it on and lets it run. However, the LEDs and four-digit display provide some status information.

Table 1: Nexys LED assignments

Thing	Purpose
LED 0	RS232 Data In
LED 1	Navigation Valid
LED 2	Flash Write-In-Progress
LED 3	Flash Ready-For-Command
LED 7	Reset
Numeric Display	Current Speed

LED 0 illuminates when data is being received from the GPS chip over the RS232 chip. LED 1 illuminates when the GPS chip has obtained a lock on at least four satellites, and is now providing valid position information. LED 2 illuminates when data is being written to the flash, and when it is being erased. LED 3, somewhat redundantly, illuminates when the Flash ROM is idle. Finally, the four-digit numeric display displays the current speed of the data logger, as meters per 100 seconds, in hex.

This means that, after power-up, the user observes the following behavior from the Nexys:

1. The RS232 data in light flashes continuously, one per second, as long as the connection with the GPS chip is intact.

2. For the first 17 seconds after initial power-up, the write-in-progress LED is illuminated as the ROM is erased, and the ready-for-command LED is dark.

3. At some point, approximately 45-60 seconds after power-up, the navigation valid LED illuminates as the GPS chip obtains a position fix. Subsequently to this, the write-in-progress begins to illuminate once per second, immediately following the pulses of the RS232 data-in light, indicating that the position history is being written to the ROM.

Pushbutton seven on the Nexys board is the user reset: pressing it resets all logic on the board to its default state. As a consequence, the Flash ROM is erased as well. However, the GPS chip is not affected, and thus will not lose its position fix.

1.1.1 Connections and Power

The Flash ROM connects to the Nexys through one of the board's 6-pin accessory headers, and is powered from the board's internal 3.3v power regulator. The GPS chip is connected via another of the headers, but receives power directly from the source the board is connected to.

While the GPS chip uses the RS232 protocol, the TX and RX lines are TTL-level, which means that they can be connected directly to pins on the FPGA.

Warning: the Nexys can be powered via wall adapter, USB, or battery, and can handle any voltage in the range 4V-10V. The GPS chip is driven directly from the power input to the Nexys, but its input range is much more limited: 4.5V to 6.5V.

1.2 Visualization System

The visualization system was originally intended to have a fairly elaborate interface, displaying labeled axes, summary information for the data set (max speed, max altitude, etc.), and allowing the user to use the mouse to switch between visualizations, pan, scroll, zoom and rotate within them, and so forth. Though its implementation would be straightforward, this functionality could not be completed in time, and thus the systems's displays are rather bare, and interaction with the user is accomplished using the labkit pushbuttons. However, the primary functionality has been completed.

The visualization system allows the user to choose between several different views of the data. Three are rendered in 2D: altitude vs. time, velocity vs. time, and latitude vs. longitude. One is rendered in 3D: latitude, longitude and altitude.

The user can zoom, pan, and (for 3D) rotate these visualizations using the buttons the labkit. For the 2D position display, the directional buttons (up,

down, left and right) move the view to the north, south, east and west, respectively, and by holding the third pushbutton and either the up or down button, the view can be zoomed in and out.

For the 2D altitude and velocity displays, the left and right buttons scroll through the time axis, and the up and down buttons adjust the scale of the y-axis. Holding the third pushbutton and the up/down buttons adjusts the scale of the x-axis.

For the 3D displays, the pushbuttons adjust the latitude and longitude coordinates of the camera. Holding the third pushbutton and the up/down buttons adjusts the altitude of the camera. Holding the second and first pushbuttons, in combination with the directional buttons, adjusts the rotation of the camera around the appropriate axes.

2 Mobile Data Logger Implementation

The mobile data logger's implementation is fairly simple: it reads data from the SiRF StarIII GPS receiver over RS232, decodes the appropriate messages, formats them for storage, and then writes them to the Flash ROM.

The entire system is driven by the built-in 25 MHz clock of the Nexys.

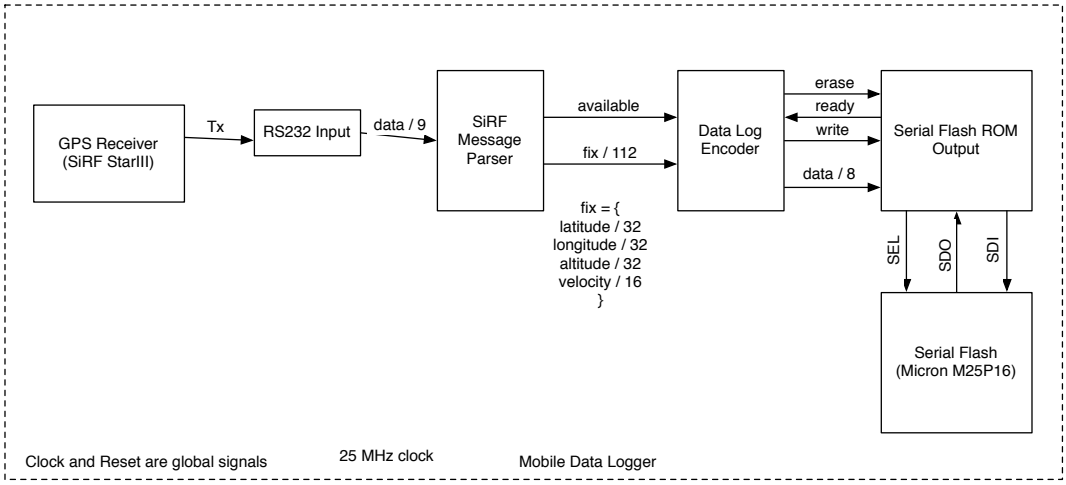

Figure 1: Mobile data logger block diagram.

2.1 RS232 Input Module

The RS232 module is responsible for decoding RS232 data coming from the GPS receiver. Before we go into its implementation details, it will help to discuss how the RS232 protocol itself works.

Table 2: RS232 Input Module Ports

Signal	Width	I/O	Description
clk	1	I	Clock signal
reset	1	I	Reset
baud16	1	I	Enable signal at 16x baud rate
rx	1	I	RS232 RX signal
data	8	O	Byte of serial data
available	1	O	High when *data* is valid

2.1.1 RS232 Protocol

RS232 is a serial protocol for data transmission: data is transmitted one bit at a time over a single line. The "baud rate" of an RS232 connection defines how many bits may be transmitted per second. Each line is unidirectional, but most RS232 connections are bidirectional, using two lines, one in each direction.

When the line is idle, it is held at a logical 1. When a byte is to be transmitted, a 0 (the "start bit") is transmitted by pulling the line low for a single baud unit. Following the start bit, a byte of data is transmitted, MSB first. After the LSB of the byte is transmitted, a 1 (the "stop bit") is transmitted, after which the line is idle again, a single byte having been sent.

Beyond baud rate, there are a number of different parameters that result in variations to this description: there may be a parity bits, more than one stop bit, and so forth. Both devices communicating over RS232 must agree in advance on the protocol they will be using. The SiRF StarIII chip used here transmits at 57600 baud, with one start bit, one stop bit, and no parity bits.

2.1.2 Implementation Overview

The RS232 input module uses a technique called oversampling to ensure that it receives data accurately. It samples the RX line at a frequency sixteen times the baud rate (thus, $57600 * 100 = 921600$ times per second). When it detects a start bit, it waits until 24 16x-baud ticks have gone by (indicating that it is now 8 16x-baud-ticks into the transmission of the MSB), and then starts a counter that samples the eight data bits at 1-baud intervals. In this way, it ensures that it is sampling each bit approximately in the middle of the baud period, thus (hopefully) minimizing the effects of interference and capacitance on the value received. It also reduces the chance of a slightly out-of-frequency baud generator (as ours is) causing problems, by ensuring maximum margins on both "sides" of the bit being transmitted.

2.1.3 FSM

The RS232 module is a simple Moore FSM, with four states. All state transitions occur on the baud16 clock enable.

WAITING In the waiting state, the RX line is idle (logical 1), and the module is simply waiting for a start bit. Once a start bit is found (the line goes low), the module transitions to the SYNCHRONIZE state.

SYNCHRONIZE In the synchronize state, the module waits 24 baud16-ticks, until the start bit has been transmitted, and the MSB is halfway through being transmitted. At this point, the modules enters the SAMPLE state.

SAMPLE In the sample state, the module reads eight samples from the RX line, sampling the first the immediately and waiting one baud-tick for the next seven, reading them into a shift register that will contain the received byte at the end of the process.

Once the full byte has been read, the module transitions to the STOP_BIT state.

STOP_BIT In the stop bit state, the module simply waits another 24 baud16-ticks until the stop bit has passed, and then returns to the WAITING state.

2.2 SiRF Message Parser Module

Table 3: SiRF Message Parser Ports

Signal	Width	I/O	Description
clk	1	I	Clock signal
reset	1	I	Reset
data	8	I	RS232 byte in
dataAvail	1	I	High when *data* valid
fix	112	O	Current position
fixAvail	1	O	High when *fix* valid
messageID	16	O	ID of the last SiRF message received
navValid	1	O	High when the GPS chip has obtained a position lock.

The SiRF message parser is responsible for reading data coming from the GPS chip through the RS232 receiver module, one byte at a time, parsing the SiRF messages that the data contains, and extracting the relevant position information from them.

As outputs, it supplies position information via the *fix* output as it is received, as well as the ID of the last message received, and a signal indicating whether the GPS chip has obtained a satellite lock (*navValid*).

2.2.1 SiRF Message Format

The GPS chip transmits messages in the SiRF binary message format. This defines a number of different message types, identified by a 2-byte ID, all sharing a commond header structure.

Table 4: SiRF Header Format

Start Sequence	Payload Length	Payload	Checksum	End Sequence
0xA0, 0xA2	2 bytes	1023 bytes	2 bytes	0xB0, 0xB3

We are interested in messages with ID $0x41$ ("Geodetic Navigation Data") in the SiRF specification), and don't care about the rest. These messages are 91 bytes long, and contain a variety of data fields, including the ones we are interested in: latitude, longitude, altitude, and velocity.

2.2.2 Implementation Overview

The message parser module was implemented with two FSMs, one "major', one "minor'.

2.2.3 Major FSM

The major FSM is responsible for parsing all incoming message headers, identifying when a message with ID 0x41 is being received, and alerting the minor FSM when this is the case.

It consists of seven states.

RESET The reset state simply clears all outputs and prepares to enter the IDLE state once *reset* goes low.

IDLE In the idle state, no message is currently being received. When we read the first start bit (0xA0) of the SirF message header, we transition to READ_S2.

READ_S2 In the READ_S2 state, we verify that the next byte in the stream is the second start bit (0xA2). If so, we continue to the READ_LENGTH_1 state. If not, we return to IDLE, confused.

READ_LENGTH In the two READ_LENGTH states, we read the next two bytes in the stream, representing the message length, and store their value for later use. The READ_ID state always follows.

READ_ID In the READ_ID state, we read the next byte in the stream (the message ID field of the binary protocol header). If the message ID equals 0x41, we alert the minor FSM by raising the internal *parserStart* signal high.

Regardless of the value of the message ID, the next state is WAIT_MESSAGE.

The last message ID received is stored in a register, and is one of the outputs of the module as a whole.

WAIT_MESSAGE In this state, we simply read bytes and increment a counter, until the number of bytes we've read equals the message length (cached earlier). At which point, we know that this message is over, so we return to the IDLE state, and await another.

Explicitly waiting until the exact number of bytes have been read ensures that we will not get confused if the start and/or stop sequences ever occur in the data stream.

2.2.4 Minor FSM

The minor FSM is responsible for parsing the payload of a "Geodetic Navigation Data" SiRF message. The *parserStart* signal, controlled by the major FSM, indicates when one of these messages is being received.

The minor FSM generates values for the *fix*, *fixAvail*, and *navValid* outputs of this entire module.

It consists of nine states.

RESET Reset simply initializes the FSM's internal values, and prepares it to enter the IDLE state once *reset* falls.

IDLE In the IDLE state, no message is being received. If the *parserStart* signal goes high, it enters the READ_VALID state.

READ_VALID This state is responsible for reading a two-byte indicator at the beginning of the message, indicating the type of position information the GPS is supplying. It uses a two-byte shift register and a counter to read this field, and then transitions to the READ_LATITUDE state, after discarding the next 20 bytes in WAIT.

WAIT The WAIT state is used by several other FSM states to wait until a certain number of bytes have arrived (discarding them), and then transition to another state.

This is useful, because the message we're reading contains a large amount of information, and we need only a small subset of it.

The *afterWaitState* register is used to store the state to transition to after this one.

READ_LATITUDE, READ_LONGITUDE, READ_ALTITUDE, READ_SPEED These states read their respective values from the message, and store them in a register. After the final value in the message is read (in READ_SPEED), we return to IDLE.

Table 5: Log Encoder Module Ports

Signal	Width	I/O	Description
clk	1	I	Clock signal
reset	1	I	Reset
fix	112	I	Current position
fixAvail	1	I	High when *fix* valid
data	8	O	Data to write.
write	1	O	High when *data* is valid for writing.
ready	1	I	High when the flash is accepting data.

2.3 Log Encoder Module

The log encoder module is responsible for taking the 112-bit position fixes produced by the SiRF message parser, and serializing them into a sequence of 8-bit values to be stored on the flash.

It uses an internal counter and a shift register to write out the position values - a simple implementation.

2.4 Flash ROM Writer

Table 6: Flash ROM Writer Module Ports

Signal	Width	I/O	Description
clk	1	I	Clock signal
reset	1	I	Reset
memCE	1	O	Flash chip enable
memIn	1	I	Flash serial data in
memOut	1	O	Flash serial data out
memClk	1	O	Flash serial clock
data	8	I	Data to write
write	1	I	High when *data* should be written
ready	1	O	High when the flash is idle
wip	1	O	High when the flash is being written to

The Flash ROM writer is responsible for interfacing with the ST Microelectronics serial flash: erasing it, and writing bytes to it. It raises *ready* high when it is able to accept data for writing. When *write* is high, it writes the byte found on *data* to the next available address on the Flash, starting at zero and automatically incrementing with each write.

The amount of time the erase and write operations take is not fixed, so after performing one of these operations we must constantly poll the chip until it indicates that the operation has completed.

2.4.1 Serial Flash Protocol

Communication with the flash chip occurs using a four-line bidirectional serial protocol. The serial clock, chip enable, and data input are driven by the FPGA, and the data output line is driven by the flash chip.

The protocol is based on commands. Each command is initiated by bringing the chip enable low, followed by the transmission of an 8-bit command ID on the input line, followed by (depending on the command) a sequence of additional data such as an address, or data to write. Some commands produce a response from the chip on the output line. Raising the chip enable high ends the command. Some commands will not be recognized if the chip enable is not driven high at precisely the right moment: e.g. after the command ID, after the address, etc.

Once a byte has been written to, it cannot be changed again until the entire sector its in (or the entire chip itself) is erased. Erasing sets every bit on the chip to 1.

Before issuing any command that writes or erases part of the ROM, the write enable command must be issued. The write-enable latch inside the ROM is reset after every write, and thus the command must be re-issued before every write.

The commands we need to worry about, then, are: write enable, page program (to write data), bulk erase (to erase the entire chip), and check write-in-progress (to let us know when the page program and erase commands have finished executing).

2.4.2 FSM

The module uses one major and two minor FSMs to interact with the serial flash. The minor FSMs are responsible for shifting data in and out on the I/O lines, and quite simple. The major FSM is more complicated: it consists of six states.

RESET The RESET state initializes all values to their defaults. The next state is always ISSUE_WEN, followed by ISSUE_BULK_ERASE;

ISSUE_WEN The ISSUE_WEN state issues a write-enable command, allowing subsequent commands to modify the data stored on the ROM, and then transitions to the state stored in the *afterWENState* register.

ISSUE_BULK_ERASE This state issues a bulk erase command, initializing the contents of the entire ROM to 0xFF and allowing it to be rewritten, then transitions to ISSUE_WIP, in order to wait out the erase cycle.

ISSUE_WIP This state issues a command that asks the ROM to return its status register. This state merely issues the command - the returned value is read in the CHECK_WIP state, which always immediately follows this one.

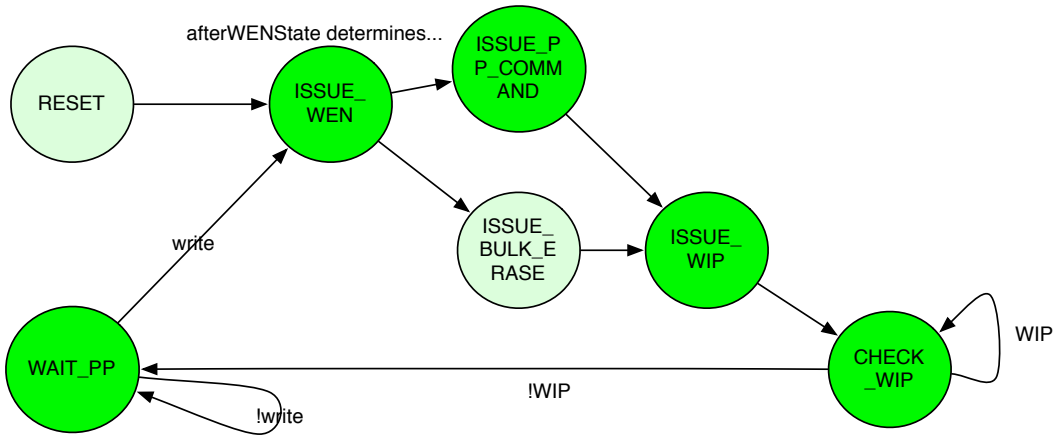

Figure 2: Flash writing FSM state transition diagram.

Table 7: Page program command format.

Command ID	Address	Dummy Byte	Bytes to write
1 byte	3 bytes	1 byte	up to 256 bytes

CHECK_WIP This state reads the value returned by the ROM in response to the status register check issued by the previous state. One of the bits in the status register indicates whether the ROM is currently being written to (in the middle of an erase or write cycle).

The status register is a single byte, but it is continuously repeated on the output line as long as the chip enable is held low. So, this state continually reads the value coming in from the ROM until the WIP bit goes low, at which point it raises the chip enable high, and transitions to the WAIT_PP state.

WAIT_PP In the WAIT_PP state, the module is idle - waiting for *write* to go high so that it can issue a page program (PP) command.

When *write* goes high, we transition to ISSUE_PP_COMMAND.

ISSUE_PP_COMMAND In this state, we issue a page program command that writes a single byte to the currently active address. We then raise the chip enable high, increment the active address by one, and transition to the ISSUE_WIP state to wait for the write to finish.

Once the write has finished, we will be back at the WAIT_PP state, awaiting another byte to write.

The page program command takes the following form:

The dummy byte can take on any value. Up to 256 bytes (a single sector of the ROM) can be written any time, but given the low throughput requirements, we write one byte at a time for simplicity.

3 Visualization System Implementation Overview

The visualization system takes the position/velocity values stored on the ROM. It can be broken into four major subsystems: the ROM reader, the rendering pipeline, the visualization modules, and the video RAM / VGA display.

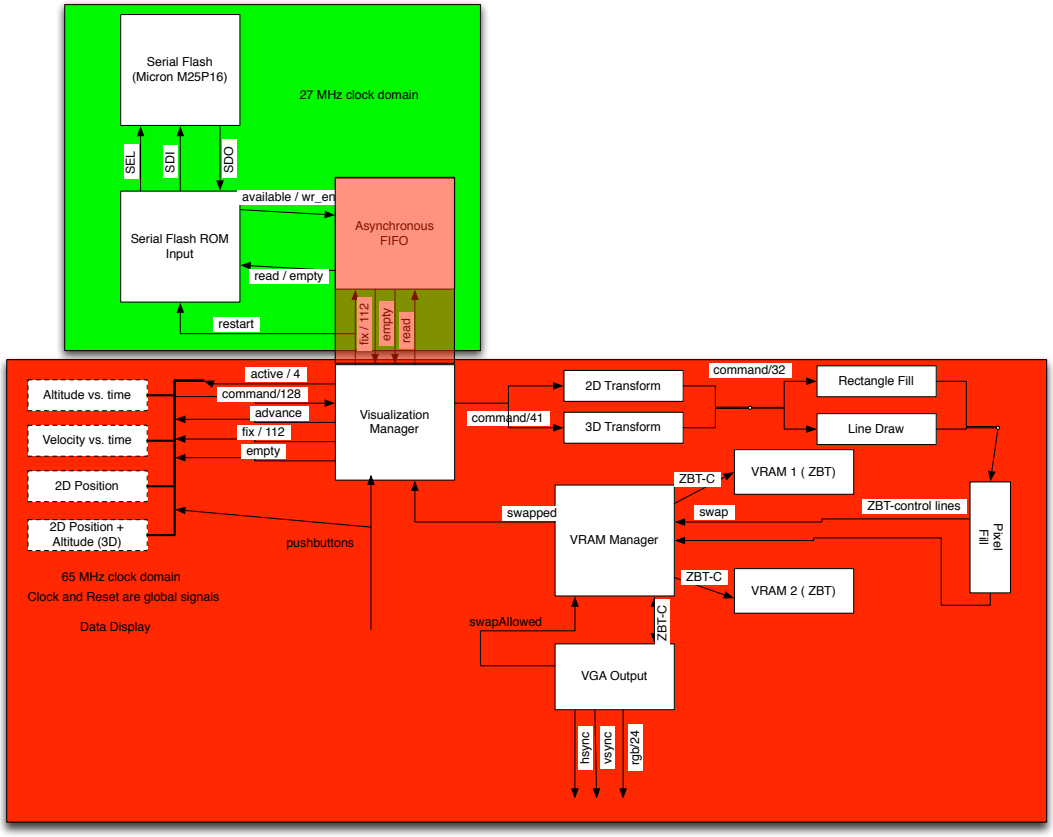

Figure 3: Visualization system block diagram.

4 ROM Reader / Log Decoder

The ROM reader subsystem is responsible for reading logged position/velocity information from the Flash ROM, and making it available to the rest of the system.

This is essentially the inverse of the mobile data logger, described above, and so in the interest of spending more time on the interesting topics, we will not describe its component modules in detail.

As a whole, the ROM reader component can be considered a module with the following ports:

Table 8: Flash ROM Reader Module Ports

Signal	Width	I/O	Description
clk_flash	1	I	27 MHz clock signal
clk_rest	1	I	65 MHz clock signal
reset	1	I	Reset
memCE	1	O	Flash chip enable
memIn	1	I	Flash serial data in
memOut	1	O	Flash serial data out
memClk	1	O	Flash serial clock
position	112	O	Position fix
read_en	1	I	Read the next position, if !*empty*
empty	1	O	Low if no positions are available to be read.

Because reading each logged position takes approximately 152 clock cycles, the positions are stored in a 512-location FIFO queue, to ensure that there is always a constant supply of them ready for the rest of the system ton consume. The *empty* signal is low when this queue has values in it. Raising *read_en* high for one clock when *empty* is low places the next stored position on the *position* output.

The *position* signal contains the following values:

Table 9: Position record format

Latitude	Longitude	Altitude	Velocity
32-bit	32-bit	32-bit	16-bit

All values are integers. The latitude and longitude values are in degrees$*10^7$, the altitude value is in meters$*100$, and the velocity value is in meters/second$*$ 100.

4.1 Clock Speeds

One problem that arose while implementing this component was that, while the rest of the systems run off the same 65 MHz clock that the VGA module requires, the Flash ROM's maximum rated clock speed is 50 MHz. The solution was to make the FIFO that connects this component to the rest of the system asynchronous: the write ports are driven at 27 MHz along with the rest of the ROM reader subsystem, and the read ports at 65 MHz, along with everything else.

4.2 Determining the end of the log

Another issue was how to determine where the last logged position is. Because of the imprecision of the Flash chip's erase functionality, it is not practical to write an integer somewhere with every record indicating how many positions have been stored.

An easy solution was to make note of the fact that erasing the chip sets every bit to 1. Considering that a speed or altitude of 0xFFFF will never be recorded, we know we have reached the end of the log when we see a speed with that value.

However, because doing so makes the visualization modules' implementations simpler, the ROM reader does not stop reading after it finds a speed of 0xFFFF - it will continue on, filling up the FIFO with 1s. It is up to the visualization modules to recognize that this means the end of the data, and stop reading.

4.3 Resetting

The *reset* signal is used both at power-on to initialize the module, and by the visualization modules to tell the ROM reader to start reading from the beginning of the logged data: raising it high clears the FIFO and restarts the flash reader from address zero.

This is perhaps somewhat sloppy - it would be cleaner to have a distinct *restart* signal, and use *reset* solely for power-on initialization, or a user reset of the entire system. On the other hand, one could also call it more efficient to use *reset* for both purposes.

5 Video RAM

In order to support the (regrettably unfinished) advanced graphics functionality that was planned for this project (such as transparency, anti-aliasing, and so forth), it was decided to use 18-bit color, and a 1024x768 display resolution. Also, because the visualizations will be drawing figures of arbitrary complexity, it was decided to use video RAM instead of sprites. To avoid the hassle of having to perform the calculations only in the horizontal and vertical blanking periods of the VGA rendering, a double-buffered memory system was deemed necessary as well.

Since the labkit posesses two ZBT rams, the obvious choice was to use each of them as a separate video buffer. The addressing scheme posed a minor challenge, since the system needed to be able to store and address 768k values of 18 bits each, while the ZBTs were organized as 512k values of 36 bits each. While the storage capacity was there, the ZBTs are not laid out in the optimal pattern.

The solution was to use a virtual memory address for reads and writes, and implement circuitry to decode it into a physical address. The virtual address was

20 bits long, with the high-order 10 bits corresponding to the y pixel coordinate, and the low-order 10 corresponding to the x pixel coordinate.

The video RAM circuitry uses the high-order 19 bits of this virtual address as the physical address passed to the RAMs themselves, and uses the LSB to determine whether the upper or lower 18 bits of the memory location are being read from/written to. This is illustrated in the diagram below.

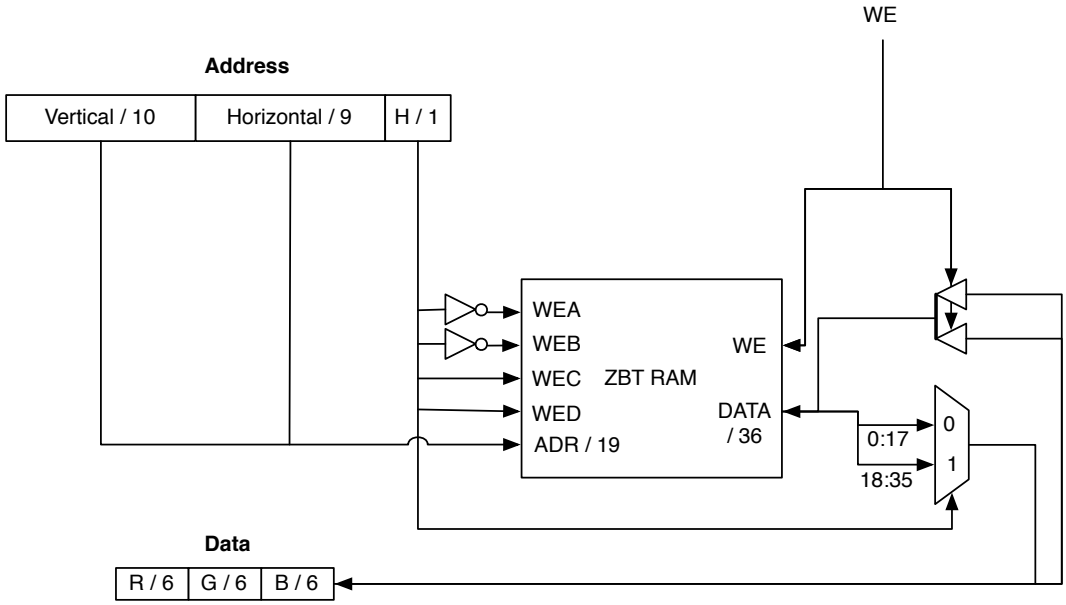

Figure 4: Video RAM virtual address decoding

5.1 VRAM Module

The VRAM module is the gateway between the rest of the system, and the ZBT RAMS. It converts the virtual addresses and read/write signals into physical addresses going out to the ZBTs.

At any given point, one of the RAMs is designated as 'active', and the other as 'inactive'. The 'active' RAM is the one that is driving the display, and the 'inactive' one is either idle, or being written to by the the rendering subsystem.

Internally, the VRAM uses two instances of a submodule ("addrDecoder") to convert between vritual and physical addresses, and a series of muxes to determine which address decoder the incoming 'active' and 'inactive' signals are connected to.

When the *swap* input goes high, the module waits until the *swapAllowed* input goes high, and then switches the roles of the active and inactive VRAMs, raising *swapped* high for one cycle once this occurs. The *swapAllowed* signal

Table 10: VRAM Module Ports

Signal	Width	I/O	Description
clk	1	I	27 MHz clock signal
reset	1	I	Reset
activeAddr	20	I	Virtual active VRAM address
activeRead	18	O	Virtual active VRAM data
activeWE	1	I	Virtual active VRAM write enable
activeWrite	18	I	Virtual active VRAM data write
inactiveAddr	20	I	Virtual inactive VRAM address
inactiveRead	18	O	Virtual inactive VRAM data
inactiveWE	1	I	Virtual inactive VRAM write enable
inactiveWrite	18	I	Virtual inactive VRAM data write
swap	1	I	Request to swap active/inactive VRAMs
swapped	1	O	High for one clock when a swap has occurred
swapAllowed	1	I	High when a VRAM swap is allowed by the VGA module
ZBT 0	N/A	N/A	Control signals for ZBT zero
ZBT 1	N/A	N/A	Control signals for ZBT one

is generated by the VGA module, and is asserted in the middle of the vertical blanking interval.

5.2 VGA Module

The VGA module was taken directly from the lab five assignment, and minimal modifications were made. Primarily, it was modified to read pixel data from the active ZBT RAM, and to generate the *swapAllowed* output during the vertical blanking interval (when vcount equals 790, specifically).

6 Rendering Pipeline

The rendering pipeline is the core of the project. It is responsible for taking real-world coordinates from the visualization modules, transforming them into pixel (1024x768) coordinates, and then drawing the appropriate shapes on the screen.

It is divided into three basic stages: transformation, drawing, and pixel coloring, illustrated in figure 3. Note that the first two stages consist of multiple modules, each implementing a subset of the functionality that the stage performs.

The pipeline is controlled through rendering commands, which are 40-bit values somewhat similar to bytecode in a processor. Each command specifies a single operation for the pipeline to perform. They move through the pipeline from 'left' to 'right', entering the transformation stage first, then progressing through the drawing stage, and ending their journey at the pixel coloring stage.

Example commands include "draw 2D line", "transform 2D point", "set 3D camera parameters", and so forth.

Commands entering the pipeline take the following format:

Table 11: Pipeline command format

Transform Op	Draw Op	Pixel Op	Data Segment
3 bits	3 bits	3 bits	32 bits

The first 9 bits of every command is the command header, which contains opcodes for each stage in the pipeline. 0b000 is a NOOP opcode: its presence in any of the 3-bit stage opcode fields indicates that the corresponding stage should not perform any operation, and merely pass the command along to the next stage.

The specific other commands that each module (and, by extension, the pipeline) supports are listed in the module descriptions below. Those sections also explain the details of the 2D and 3D transforms.

6.1 Commands and Modules

Note that each stage of the pipeline consists of multiple modules, and, to allow for maximum flexibility, the output of every module in a stage is fed to the modules in the next stage. For example, this allows the 2D transform module to be used to transform coordinates and supply them as input to either of the drawing modules with equal flexibility. Similarly, it allows the drawing modules to *receive* their points from either the 2D or 3D transform modules (or, for that matter, from outside the pipeline, with no transformation at all) through the exact same mechanism.

The implementation of these inter-stage connections was a minor challenge: how to determine which module in each stage provided the output for the whole stage? A number of solutions were considered, among them muxes and tristate buses. The final solution consisted of the following rules:

1. The output command of every stage is the logical OR of the output command of each individual module within it.
2. Every stage opcode is recognized by exactly *one* module within the stage.
3. When a module sees an opcode that it does not recognize, it outputs an all-zero command.
4. When a module sees a NOOP opcode, it outputs the input command verbatim on its output.

So, when a command enters a stage, only one module recognizes it, and outputs a non-zero command - the others output zeros. The logical OR means that the output of the stage is the output of the module that recognized the opcode. This avoids the delays and additional hairy logic of muxes, and the potential bus fights of tristate drivers.

6.2 Command Widths and Serialized Commands

The initial plan was for every command to be 128 bits long: this would allow for most common pipeline operations to be fully described in a single command. However, based on advice that this may lead to routing problems, and would be difficult to debug on the logic analyzer, this was reduced to 41 bits for the transform-stage commmands, and 32 bits for the drawing- and pixel-stage commands.[1] This means that most common operations are described by a sequence of commands: for instance, the "set 3D camera limits" operation consists of six commands, one for each 3D camera parameter.

6.3 The Advance Signal

In the execution of a command, a stage may issue multiple subcommands of its own. For instance, a command to draw a line from point $(100, 100)$ to $(100, 200)$ will result in 200 pixel fill commands being executed.

Here, one notices a bit of a timing issue: commands like "draw line" can take a variable number of clock cycles, depending on the length of the line being drawn. The solution is to allow every stage in the pipeline to halt the execution of all stages before it, until it has completed whatever operation it is performing. The signal used to implement this is called *advance*: every stage in the pipeline takes as input the logical AND of all the advance signals of the stages following it. When the *advance* input to a given stage is low, that means that the next stage will ignore any commands it receives, until *advance* goes high again.

The *advance* signal is also provided to components outside the pipeline (i.e. the visualization modules), to let them know when the pipeline is accepting commands.

As-implemented, the handling of *advance* in the various pipeline stages is sub-optimal. For optimal performance, each stage should only wait for the *advance* signal to go high when it needs to send a command to the next stage. If a stage is performing a lengthy calculation, it need not worry about *advance* until the calculation is over, and it needs to pass on the results.

In practice, however, this is not how I implemented things: I used *advance* as sort of a clock enable: if it goes low, everything in the pipeline preceding the stage that de-asserted it will grind to a halt. This makes the implementation and debugging much simpler, but would have to be changed if optimum performance was required, as it results in wasted clock cycles.

6.4 An Example Command Sequence

My explanations are likely rather poor, so just to illustrate how all this stuff works, let's go through a simple example. We want to blank out the screen, set the 2D view limits to $x_0 = 0, y_0 = 0, x_1 = 100, y_1 = 100$, and then draw

[1] The transform-stage commands must be 41 bits long to allow room for the 9-bit header and 32-bit input value - the input values of the other stages need not be as large

a line from $(20, 20)$ to $(80, 80)$. All these values are in our pre-transform, 2D coordinate system.

First, we issue a command to the rectangle drawing module directly by setting the transformation stage opcode to NOOP, and the draw stage opcode to 0b010 (the opcode for "draw rectangle"). For the pixel stage opcode, we use 0b001 (normal pixel fill, the only pixel operation implemented). In the data segment of this command, we specify the color of the rectangle we wish to draw: black. The rectangle drawing module will store the pixel fill opcode and color value we specify, and use them in the pixel fill subcommands it issues on our behalf.

Following this command, we issue two more with the same opcode pattern, but different data: the data segment of the first command contains the x and y coordinates of our rectangle's upper-left corner [(0,0)], and the second the x and y coordinates of it's bottom-right corner [(1023,767)].

advance will go low for a while as the rectangle drawing module does its work. Once it finishes, we issue a command that sets the 2D transform view limits. In this command, we set the transform stage opcode to 0b100, and the other opcodes to NOOP. The data of the first command contains the 32-bit value we desire for $x0$ (0). We issue three more commands identical to this one, containg the values for $y0, x1$ and $y1$, respectively.

advance will once again go low as the 2D transform performs preliminary calculations on these values. Once it goes high, we want to draw our line. So, we issue a line-draw command, identical in structure to our rectangle-draw command except for the opcode. We specify the color as red.

But now, instead of supplying the points to draw *ourselves* by passing them in directly, we will get the 2D transform module to do it. We send out two sets of two commands each. The transform opcode of these commands is set to 0b101, the appropriate opcode for a 2D transform, and the draw opcode is set to 0b010. The data segments of these four commands are $20, 20, 80$ and 80, corresponding to starting and ending pre-transform coordinates of the lines we wish to draw. The 2D transform module performs the appropriate calulcations, and issues two commands: the first containing the transformed version of our starting point, and the second the transformed version of our ending point. It sets the draw opcode of its output commands to the draw opcode we provided on the input, and so these commands are picked up by the line drawing module, and it draws a line from pixel (204,153) to (818, 613).

Note the flexibility that this allows: let's say we wanted to draw a line from the center of the screen, to a certain point in our pre-transform coordinate system. We would simply issue our same line-drawing command, provide the first point ourself [(511,383)], and then issue a 2D transform command to provide the second point.

6.5 Extensions and Generality

A cool thing about this design is that the rendering system is completely non-specific, and knows nothing whatsoever about geographical coordinates[2]. It can be used to render visualizations of *any* data set. Note that the *only* difference between the the three 2D visualizations are the view limits on the 2D transformation, and the data being sent to it.

In addition, the rendering pipeline as implemented here is really quite general-purpose, and can be extended quite easily to include essentially arbitrarily complex graphical techniques, moreso than you might infer based on the limited functionality implemented within it thus far.

For instance, the initial implementation plan was to allow text to be drawn to the screen: this would be implemented as just another module and set of opcodes in the drawing stage. Additionally, 8-bit alpha transparency was planned as an additional mode for the pixel fill stage: it was to be enabled simply by changing the pixel fill opcode on the commands being issued. [3]

In addition, one can imagine a 3D engine operating on polygons, in which the coordinates of their vertices are transformed and then passed to a generalied polygon-fill module. Clipping, lighting, texturing, and so forth could easily be implemented through additional modules and stages. Implementing an advanced rendering system such as this was one of the early 'if time permits' goals of this project, albeit one that proved ludicrously out of reach.

7 Rendering Pipeline Module Descriptions

This section contains detailed descriptions of the implementations of the various modules that make up the rendering pipeline.

All pipeline modules share the same set of input and output ports, except the pixel fill module, which has no *commandOut* port, replacing it with video RAM control lines.

Table 12: Generalized pipeline module ports

Signal	Width	I/O	Description
clk	1	I	27 MHz clock signal
reset	1	I	Reset
commandIn	41 or 32	I	Pipeline command in
commandOut	32	O	Pipeline command out
advanceIn	1	I	Pipeline advance input
advanceOut	1	O	Pipeline advance output

[2] Actually, there are some hardcoded scaling factors in the 3D module, but these could be removed and made settable quite easily.

[3] The remnants of this plan can still be found throughout the system.

7.1 2D Transform

As you might expect, the 2D transform module is responsible for performing two-dimensional coordinate transformations. Its operation is defined by four parameters: $x0, y0, x1$, and $y1$. These define the area in the source coordinate system that will be mapped onto screen coordinates: $x0$ represents the left edge of the screen, $y0$ the top edge. These values are 32-bit signed integers.

As input for transformation, the module accepts a single pair of 32-bit x and y values, and transforms them into 10-bit pixel coordinates. The module assumes that the input values fall within the view boundaries: if not, the module's behavior is undefined.

Mathematically, this means that given a point to transform (a, b) within the view boundaries, the output point will be given by:

$$x = (a - x_0)\frac{1023}{x_1 - x_0}$$
$$y = (b - y_0)\frac{767}{y_1 - y_0}$$

Note that this involves a division, something that is very expensive to perform on an FPGA. Fortunately however, this division does not need to involve the point being transformed: it is a function only of the view limits. So, what we can do is calculate $\frac{1023}{x_1 - x_0}$ and $\frac{767}{y_1 - y_0}$ when the view limits are set (which is presumably not very often, only once per frame), and then transform each point simply by multiplying the stored results of those calculations by $(a - x_0)$ and $(b - y_0)$.

And in fact, that is exactly what this module does. It uses a Coregen pipelined divider to calculate the appropriate view limit multiplication factors, and a Coregen multiplier to perform the actual point multiplications.

All operations are performed in signed, 11.20 fixed-point values, which was chosen to be on the conservative side: such precision is not likely necessary.

The implementation of this module is as a pair of state machines: the primary state machine performs the standard point transformations, but defers to the secondary state machine to carry out the view-limit factor divisions.

7.1.1 Command Formats

2D Set Limit The 2D set limit operation takes the following form.

Table 13: 2D Set Limit Command Format

Transform Opcode	Draw Opcode	Pixel Opcode	x_0, y_0, x_1, y_1
0b001	0b000	0b000	32-bit signed

Four commands are required to complete the operation: the first with x_0 as the data, the second with x_1, the third with y_0, the fourth with y_1.

2D Transform The 2D transform operation takes the following form.

Table 14: 2D Transform Command Format

Transform Opcode	Draw Opcode	Pixel Opcode	x, y
0b011	0b000	0b000	32-bit signed

Two commands are required to complete the operation: the first with the x-coordinate in the data segment, the second with the y-coordinate.

7.2 3D Transform

Please see section 8 for a discussion of the 3D transformation module.

7.3 Rectangle Draw

The rectangle draw module is extremely simple: it simply takes in two (x, y) pixel pairs, and draws a rectangle with the first pair as its upper-left corners, and the second pair as the bottom-right corner.

The implementation is a simple state machine with some internal counters to keep track of which pixel is currently being colored.

7.3.1 Command Format

The rectangle draw module performs a single operation, requiring three commands to fully specify.

Command Format One - Color Specification This command specifies the color of the rectangle that should be drawn.

Table 15: Rectangle Draw Command Format - Color

Transform Opcode	Draw Opcode	Pixel Opcode	Color	Alpha
0bXXX	0b010	0bXXX	18-bit	8-bit

Command Format Two - Starting/Ending Coordinates This command specifies either the starting or ending (x, y) coordinate pair that defines the rectangle. It should be issued twice, once for each pair.

Table 16: Rectangle Draw Command Format - Endpoints

Transform Opcode	Draw Opcode	Pixel Opcode	x	y	unused
0bXXX	0b010	0bXXX	10-bit	10-bit	6-bit

7.4 Line Draw

The line draw module is an implementation of Bresenham's algorithm for drawing lines on a raster display. It accepts input of the same form as the rectangle draw module, but instead of filling the area between the two supplied points with a rectangle, it draws a line between them.

I do not reproduce it here - the implementation I used is based on the pseudocode available at Wikipedia[4]. It it possible to implement the algorithm using only additions and bit shifts, which makes it particularly appealing for this project.

This module is only slightly more complicated than the rectangle fill module: it uses an internal submodule ("paramCalc") to determine the various parameters that the algorithm uses, based on the line endpoints, and with that data in hand a few states are sufficient to loop through the algorithm itself.

7.4.1 Command Format

The line draw module performs a single operation, requiring three commands to fully specify.

Command Format One - Color Specification This command specifies the color that line should be drawn in.

Table 17: Line draw command format - Color

Transform Opcode	Draw Opcode	Pixel Opcode	Color	Alpha
0bXXX	0b001	0bXXX	18-bit	8-bit

Command Format Two - Starting/Ending Coordinates This command specifies the two endpoints of the line. It should be issued twice, once for each endpoint.

Table 18: Line draw command format - endpoints

Transform Opcode	Draw Opcode	Pixel Opcode	x	y	unused
0bXXX	0b001	0bXXX	10-bit	10-bit	6-bit

Note that invoking a rectangle draw operation requires issuing one command of format one, and two commands of format two.

7.5 Pixel Fill

The pixel fill module colors in a single pixel in the inactive video RAM. It is also responsible for requesting that the VRAM module swap the active/inactive

[4] `http://en.wikipedia.org/wiki/Bresenhams_algorithm`

RAMS. The choice to make it responsible for the latter functionality was somewhat arbitrary - I wanted to make the swap command part of the pipeline, and there didn't seem to be anywhere else to put it.

This module's implementation is also a simple state machine (four states), which just takes the 20-bit pixel address and 18-bit right off the commands, and puts them on the RAM control lines. Nothing to it.

7.5.1 Command Format

The pixel file module responds to two commands.

Normal Pixel Fill - Command One This is is the first command of the normal pixel fill operation, specifying which pixel is to be colored.

Table 19: Pixel Fill Command Format - Location

Transform Opcode	Draw Opcode	Pixel Opcode	x	y	unused
0bXXX	0bXXX	0b001	10-bit	10-bit	6-bit

Normal Pixel Fill - Command Two This is the second command of the normal pixel fill operation, specifying which color to color the pixel specified in the previous command.

Table 20: Pixel Fill Command Format - Color

Transform Opcode	Draw Opcode	Pixel Opcode	Color	Alpha
0bXXX	0bXXX	0b001	18-bit	8-bit

VRAM Swap Tells the VRAM manager to swap the active/inactive memories at the next opportunity.

Table 21: VRAM Swap Command Format

Transform Opcode	Draw Opcode	Pixel Opcode	Unused
0bXXX	0bXXX	0b011	26-bit

8 3D Transformation Module

The 3D transformation module is responsible for transforming points in any arbitrary 3-dimensional coordinate system (in this case, latitude/longitude/altitude, although latitude/longitude/velocity was planned as well) into screen coordinates. Like the 2D transformation module, the transformation it performs is

determined by a set of view parameters: six, in this case. The parameters control the location and rotation of a camera in the 3D world: incoming coordinates are transformed into 2D using a perspective projection that emulates the view a person would see if they were at the camera's location, looking out. These parameters are: cam_x, cam_y, cam_z, determining the location of the camera in 3D space, and $\theta_x, \theta_y, \theta_z$, determining the rotation of the camera around the x, y, and z axes. These parameters are sufficient to describe any possible camera location, meaning we can use the 3D transformation module to render views of our data from any perspective we choose.

8.1 Math

The 3D transformation is implemented as a sequence of 4x4 matrix transformations, in homogeneous coordinates.

When the camera parameters are set, we use them to compute a transformation matrix M_T that will be used on all points being rendered. The transformation matrix is itself the product of the following matrices:

Perspective Provides a perspective distortion (parallel lines converge, objects in the distance appear smaller). Without this, the projection would be orthographic - not realistic at all. θ_{fv} is the field-of-view of the camera - 30 degrees to either side, in this case.

$$M_p = \begin{bmatrix} cot(\theta_{fv}) & 0 & 0 & 0 \\ 0 & cot(\theta_{fv}) & 0 & 0 \\ 0 & 0 & 1 & 20 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{1}$$

X-Rotation Rotates around the x-axis.

$$M_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos(\theta_x) & sin(\theta_x) & 0 \\ 0 & -sin(\theta_x) & cos(\theta_x) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2}$$

Y-Rotation Rotates around the y-axis.

$$M_y = \begin{bmatrix} cos(\theta_y) & 0 & -sin(\theta_y) & 0 \\ 0 & 1 & 0 & 0 \\ sin(\theta_y) & 0 & cos(\theta_y) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3}$$

Z-Rotation Rotates around the z-axis.

$$M_z = \begin{bmatrix} cos(\theta_z) & sin(\theta_z) & 0 & 0 \\ -sin(\theta_z) & cos(\theta_z) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4}$$

Translation Moves the camera to the desired location.

$$M_t = \begin{bmatrix} 1 & 0 & 0 & -cam_x \\ 0 & 1 & 0 & -cam_y \\ 0 & 0 & 1 & -cam_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5}$$

Scaling Converts all coordinates to meters (corrects for difference in distance between one degree longitude and one degree latitude).

$$M_s = \begin{bmatrix} 0.0084 & 0 & 0 & 0 \\ 0 & 0.0111 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{6}$$

The final transformation matrix M_T is then given by:

$$M_T = M_p * M_x * M_y * M_z * M_t * M_s \tag{7}$$

Then, given a point (x, y, z), we can construct a column vector

$$P = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{8}$$

To transform this point, we multiply $M_T * P$, giving us another column vector

$$P^* = \begin{bmatrix} x^* \\ y^* \\ z^* \\ w \end{bmatrix} \tag{9}$$

Our final coordinates are then given by $x^*/w, y^*/w, z^*/w$. If these coordinates are in the range $-1 \leq x, y, z \leq 1$, then the point is displayable - otherwise it is not. For the x-coordinate, -1 corresponds to the left side of the screen, +1 to the right. For the y-coordinate, -1 corresponds to the top of the screen, 1 to the bottom. The z-coordinate indicates where the point would be in the depth buffer, were we to implement one in this system.

What all this math boils down to, then, is the following: whenever the camera settings change, we perform a series of matrix multiplications that give us a transformation matrix, which we then save. For every point that comes

in, we perform a 4x4 times 4x1 matrix multiplication, giving us a 4x1 column vector of points in homogenous coordinates. We divide the x^*, y^*, and z^* values in the resulting vector by the w value, and check to see that they fall in the range $-1 \leq x, y, z \leq 1$. If so, we add one to x and y to bring them in the range 0-2, and them multiply by $\frac{1023}{2}$ and $\frac{767}{2}$ to get our screen coordinates.

Unfortunately, this computation involves a division that must occur for every transformed point: we cannot avoid it as we do in the 2D transform.

8.2 Matrices and Matrix Multipliers

We will now discuss how this transformation has been implemented in hardware. First: all calculations are performed in 32.16 fixed-point, meaning all numerical values involved are 48 bits long.

8.2.1 Matrices

Read-only matrices are represented by three signals: the two-bit signals i and j, indicating which cell of the matrix is to be read, and the 48-bit *value* signal, containing the value of that cell.

Writable matrices involve additional signals: a *we* signal (write-enable), and a 48-bit *write* signal, containing the value to write in the cell specified by i and j.

Because of performance problems discovered during the implementation of the calculations, matrices are required to be two-stage pipelined, meaning that the *value* signal is valid on the third rising edge of the clock after i and j are specified. Writes are delayed as well.

A number of different modules "implementing" this interface were written. There are generalized, writable 4x4 and column matrices that use registers for storage, as well as ROM-based ones for constant matrices that never change (like the perspective and scaling matrices).

8.2.2 Matrix Multiplier

The matrix multiplier module performs the matrix multiplication

$$R = A * B \tag{10}$$

where A, B and R are 4x4 matrices. By setting the parameter B_IS_COLUMN, instances of it can also perform the same calculation with B and R as column vectors.

The multiplier works by relying on a sub-module, the partial result calculator, which is essentially a minor FSM. The PRC computes a value for each cell in the result matrix, of which there are sixteen in all. It does this be summing the terms in the A matrix of the same row as the cell in the result matrix, multiplied by the corresponding term in B matrix in the same column as the result cell. More formally, it computes $R_{ij} = A_{i1}B_{j1} + A_{i2}B_{j2} + A_{i3}B_{j3} + A_{i4}B_{j4}$,

Table 22: Matrix Multiplier Module Ports

Signal	Width	I/O	Description
clk	1	I	27 MHz clock signal
reset	1	I	Reset
start	1	I	Start multiplying
busy	1	O	High when multiplying
done	1	O	High for one cycle when done multiplying
ai	2	O	i-index for A matrix
aj	2	O	j-index for A matrix
av	48	I	A matrix cell value
bi	2	O	i-index for B matrix
bj	2	O	j-index for B matrix
bv	48	I	B matrix cell value
ri	2	O	i-index for R matrix
rj	2	O	j-index for R matrix
rv	48	O	R matrix write value
rwe	1	O	R matrix write enable

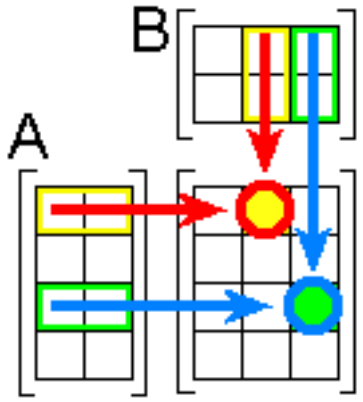

Figure 5: Illustration of matrix multiplication, showing the results of a PRC computation in cells (0,1) and (2,2). Note that these matrices are not 4x4. Source: `http://en.wikipedia.org/wiki/Matrix_multiplication`

doing this (as instructed by the matrix multiplier) once for each of the sixteen cells in the result matrix.

8.3 3D Transformation Module

Like the 2D transformation module, the 3D transformation module uses a minor FSM to perform the appropriate matrix multiplications when the camera settings change. This minor FSM takes $cam_x, cam_y, cam_z, \theta_x, \theta_y, \theta_z$ as inputs, and produces the appropriate transformation matrix, as discussed above.

It does this using two general-purpose matrices R1 and R2, and a number of hard-coded modules, corresponding to each of the matrices that are multiplied together to form the final transformation matrix. The matrices that require sine and cosine values are provided with them by a Coregen lookup table. The minor FSM alternates between the two general-purpose matrices to produce a result: using one matrix as A and the other as R, and then switching positions (using the result of the previous calculation as the input to the new calculation, all the while using each of the other matrices as B in sequence). At the end of the procedure, a mux connects whichever matrix (R1 or R2) received the last result to the minor FSM's output ports, allowing the values to be read from elsewhere in the system.

As for how to handle the actual transformation of points, here we run up against two problems. First, the interaction model I used with the 2D transform, where something external to the pipeline sends the command to the drawing stage, and the transform only provides the points, does not work here, because unlike in the 2D transform, we don't know if a point will be visible until the transform has been completed. The drawing modules could be stuck in an undesirable state if we issue a command telling them to expect some points to be coming down the pipeline, and then they never appear because it turns out they're not visible.

Additionally, there is the problem of the divider. As we mentioned above, unlike in the 2D transform, it is impossible to escape the need to use a divider during the transformation process of every point. The high latency of the Coregen pipelined divider will slow things down considerably, unless we pipeline the 3D transformation module itself. Which is exactly what was done.

8.3.1 "Burst" Mode

For some reason, I decided to call this pipelined transformation "burst" mode. Not the most accurate name, but once you've named the file and the module, it's hard to go back.

Burst mode simply means that the user of the pipeline tells the 3D transform module, via the command that initiates burst mode, what kind of shapes it is going to be drawing, and then supplies the 3D transform with a constant stream of points that specify these shapes in pre-transform space. The 3D transform them transforms the points, determines which ones are visible, and constructs and issues the appropriate drawing commands to display them on its own. This eliminates the need for any kind of complicated interaction between the visualization modules, the drawing modules, and the transformation.

This is implemented using another FSM managed by the 3D transformation module, named "burster". When burst mode is initiated, the 3D transformation module actually hands over control of its pipeline command output to the burst module, and the burst module begins reading its input commands, waiting for points to transform.

When a point to transform arrives, the burst module uses another instance of the matrix multiplier, this time configured to act on column vectors. The result

it gets out of here is the 4-value homogenous-coordinate vector mentioned above. Interfacing with a Coregen pipelined divider, it sets up the three divisions that are required to determine the point's visibility and eventual coordinates.

The results of these divisions are not handled by the burst module, however. The quotient and remainder outputs of the divider are connected to the "final transform" module, which determines whether they are in the appropriate -1 to 1 range, and if so, adds 1 to them, multiplies them by $\frac{1023}{2}$ and $\frac{767}{2}$, and then generates the appropriate command for the drawing stage based on the information passed in at the initiation of burst mode. If the points are not visible, they are simply discarded.

This functionality is separated into three separate modules for organizational purposes only, to keep the clutter down in each of them: the 3D transform, burst and final transform modules are tightly coupled, and not intended for reuse elsewhere in the system.

When whatever is using the pipeline has finished with burst mode, it issues a burst stop command. The burst module then holds advance low until all results moving through the divider have been processed, essentially "flushing" the pipeline to ensure it is clear before allowing new commands to enter it.

9 Visualization Modules

The visualization modules are responsible for taking the position fixes being read from the flash ROM and turning them into commands issued to the rendering pipeline, that draw the appropriate visualization to the screen.

Three visualization modules were implemented for this system: one draws the 2D altitude vs. time and velocity vs. time graphs, one draws the 2D position graph, and one draws the 3D position graph.

Each shares a common set of input and outputs, and essentially a common internal design and state machine. The altitude and velocity graphs were similar enough to be implemented within the same visualization module, but the other two had to be separated.

The *active* input is high when the module is the currently-selected visualization. The other signals allow the module to determine when a video RAM swap has occurred, and to control the log reading subsystem. Raising *qRestart* high resets the log reader, causing it to start over from address zero in the flash ROM. Finally, each module has inputs from the pushbuttons on the labkit allowing the camera limits to be changed.

And as with every other component that touches the rendering pipeline, the *advance* signal is used as a clock enable.

Internally, the modules have a major FSM, responsible for determining what drawing commands should be sent, driving several minor FSMs, which are what actually generate the commands. These FSMs provide an abstraction for the details of the command format, allowing the visualization modules to concern themselves only with what data needs to be drawn, not how to construct and

Table 23: Generalized visualization module ports

Signal	Width	I/O	Description
clk	1	I	27 MHz clock signal
reset	1	I	Reset
active	1	I	High when module is selected
swapped	1	I	High when the video RAMs have swapped
qEmpty	1	I	High when the position FIFO is empty
qRead	1	O	Read enable for the FIFO queue
qRestart	1	O	Restart signal for the log reader subsystem
qPos	112	I	Position value from the FIFO queue
pipeCmd	41	O	Rendering command for the pipeline
pipeAdv	1	I	Pipeline advance signal
¡camera controls¿	¡any¿	I	Camera controls (up, down, right, etc.)

issue the commands to draw it. A mux in the visualization module determines which minor FSM's output is connected to the pipeline.

The minor FSMs implemented were:

1. Set 2D Limit
2. Set 3D Limit
3. Draw 2D
4. Draw 3D

Each was implemented similarly, with a port list along these lines:

Table 24: Visualization module minor FSM module ports

Signal	Width	I/O	Description
clk	1	I	27 MHz clock signal
reset	1	I	Reset
start	1	I	Start signal - issue command
done	1	O	Done - command issued
cmd	41	O	Command output for pipeline
advance	1	I	Pipeline advance
¡parameters¿	¡any¿	I	Values to use in command

Each minor FSM takes as inputs the values to use in the command that it issues: for example, the "Set 2D Limits" minor FSM takes x_0, y_0, x_1, and y_1, and the "Draw 2D" minor FSM takes x_0, y_0, color, alpha, draw mode (line/rectangle/pixel), and so forth.

To illustrate in detail how the visualization modules are structured, we will look at the three-dimensional case, as the others are simplified versions of the same thing.

9.1 3D Visualization Module

The 3D visualization module's basic block structure is illustrated in figure whatever.

Its primary component is an FSM consisting of eleven states. In essence, it runs in a constant loop: setting the camera parameters, blanking the video RAM, looping through every logged position and drawing it on the screen, issuing a VRAM swap command, then going idle and waiting for the user to change the camera settings, thus starting the whole process over again.

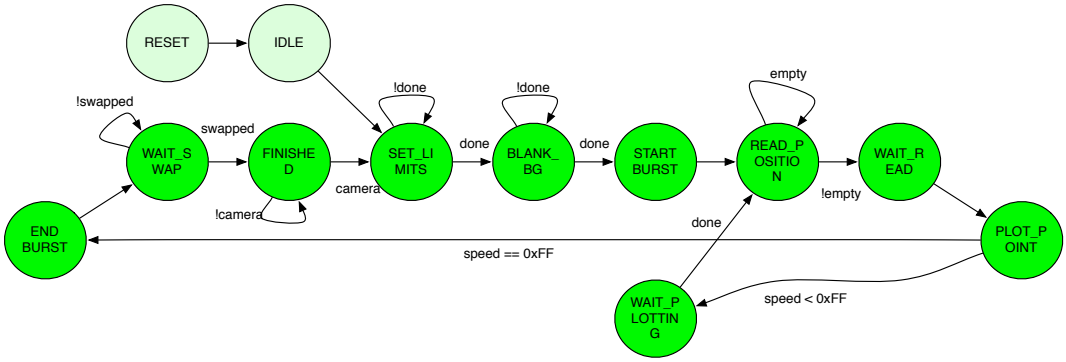

Figure 6: 3D Visualization Module state transition diagram.

RESET The RESET state is active when the system is first powered up, and when the visualization module is deselected. The next state is always IDLE.

IDLE The IDLE state's name, I have just realized, is a relic from when these modules operated differently. Now, it is actually not an idle state at all - instead, it merely sets the initial camera limits, and tells the 3D-limit-setting minor FSM to start issuing the appropriate commands to transmit them to the 3D transform module. Following that, it transitions to the SET_LIMITS state.

SET_LIMITS In this state, we wait for the 3D-limit-setting minor FSM to finish sending its commands (and, indirectly through the *advance* signal, for the 3D transform to finish initializing itself with the new camera settings). Once that's completed, we tell the 2D-drawing minor FSM to blank the entire screen by drawing a black rectangle, and transition to the BLANK_BG state.

BLANK_BG In this state, we wait for the screen to be blanked as a result of the previous state, and then transition to START_BURST.

START_BURST In START_BURST, we direct the 3D transform to enter burst mode (see INSERT-REF-HERE) by issuing the appropriate command

directly (no minor FSM involved), which specifies the drawing parameters that we would like it to cache. Following that, we enter READ_POSITION.

READ_POSITION In this state, we ensure that FIFO queue has data in it for us to read: if it does, we raise *qRead* high and transition to WAIT_READ to read the next stored position, and if not, we continue waiting.

Note that we will never be stuck waiting here indefinitely, because the queue is constantly being supplied with new data. Remember that the ROM reader does not stop reading when it reaches the end of the log - it is up to the visualization module to determine when this is the case. See BLAH above and BLAH2 below.

WAIT_READ This is simply a wait state to allow for the FIFO to see our *qRead* signal from the previous state, and produce the next value. Next is PLOT_POINT.

PLOT_POINT Here we configure the 3D-drawing module to draw a line between the current position and the last position (cached in registers), and raise *d3DrawStart* high to command it to start. We then transition to the WAIT_PLOTTING state.

The exception is if the current 'position' indicates the end of the log (if it contains all 1s), in which case we have finished drawing the current frame, so we issue a stop-burst command directly to the 3D transform module (no minor FSM involved), and then we transition to STOP_BURST. At this point, we also raise *qRestart* high, to reset the ROM reader. It will remain high for several states, in order to ensure that the signal makes it across the two clock domains.

Note that, in the case of the 3D visualizations, the 3D transform module in the first stage of the pipeline determines whether each point is visible, based on the output of the coordinate transformation. The 2D visualization modules and 2D transform behave differently - the 2D transform's behavior is unspecified when it is passed a point outside of the view limits, so it is up to the 2D visualization modules to only pass along points where $x_0 < x < x_1$ and $y_0 < y < y_1$.

WAIT_PLOTTING Here, we simply wait for the 3D-drawing module to indicate that it has finished issuing the commands we directed it to in the last state, and then return to READ_POSITION to obtain the next logged position.

STOP_BURST In STOP_BURST, having reached the end of the logged data, we wait for the 3D transform module to clear its pipeline of points under transform. Once it has indicated that this has occurred (by *advance* going high), we issue a VRAM swap command, and transition to WAIT_SWAP.

WAIT_SWAP In WAIT_SWAP, we wait for the VRAM module to indicate to us that a VRAM swap has occurred, and then transition to FINISHED.

Note that, if we did not wait for the VRAMs to swap, we could end up immediately starting to draw another frame, and accidentally corrupting part of the just-rendered frame in the process, before it is displayed on the screen.

FINISHED In FINISHED, the module is idle. It has finished drawing a frame, and has nothing to do.

We bring *qRestart* low here, having kept it high for several clock cycles to ensure that it gets through, allowing the ROM reader to begin reading new positions.

The module waits in the idle state until the user requests that the view be changed, by pushing one of the labkit pushbuttons and thus raising one of the camera-adjustment input signals high. When this happens, the module makes the appropriate changes to the camera settings, and transitions to SET_LIMIT to apply them and start the rendering process over again, with the new settings.

9.2 Visualization Module Manager

The visualization module manager is responsible for determining which visualization module is active. It drives the *active* signals going out to each of the visualization modules, and takes as input the modules' pipeline commands. A mux determines the commands that are actually sent to the pipeline, depending on which module is active.

A 2-bit counter keeps track of which module is active. The counter is incremented whenever the user requests the active visualization be changed.

Note that, since a single module provides both the time vs. speed and time vs. altitude visualizations, it has two *active* signals going to it, one for each visualization.

10 Implementation Notes: Testing and Debugging

My testing procedures evolved as the project progressed. At first, I relied solely on the labkit and logic analyer for debugging. Even when the build times were still in the range of just a few minutes, that quickly grew tiresome, and I switched to using text-based testbenches in ModelSim, which made things easier, but was still rather clunky. Then, a bit of a paradigm shift occurred when I discovered ModelSim's waveform window, and my productivity jumped enormously. If I had known about the waveform display from the beginning of the project, I'm sure I could have gotten a lot more done.

Additionally, being able to simulate Coregen cores in Modelsim was well worth the day it took to figure out - I doubt I would have finished without that capability, given how heavily my project depended on Coregen components.

Nevertheless, even after discovering the waveform window and Coregen simulation libraries, testing and debugging consumed an *enormous* amount of time on this project. I'd venture to guess that well over the majority of my time was spent hunting down bugs. And a good number of them were the result of some flaky Xilinx behavior, which got on my nerves at times. I'd like to know who thought of the idea of automatically creating new wires in port connections if you make a typo. A fair amount of errors were my fault, but would have been caught at compile time in other languages: assigning a 48-bit value to a 32'bit bus, or vice-versa. It would be nice if the language had more type safety, although I suppose it's hard to define what that means when you're dealing with an HDL.

The 3D transformation in particular proved to be a challenge to test. Getting the logic working in simulation was no problem, but timing problems once it was deployed on the labkit could be quite tricky. I often discovered bizarre behavior that was only solved by changing seemingly innocuous things in my code: in one case, assign three variables using a statement of the form {x, y, z} = {a, b, c} caused the registers in another module to start transitioning before they had even been valid on the rising edge of *any* clk. In another case, in my generalized matrix module, storing the cells of the matrix as a doubly-indexed array (e.g. value = contents[i][j]) resulted in the second-highest-order byte of cell $3, 2$ always being overwritten with 0xFF. Only that cell and only that byte. Changing the matrix to use a singly-indexed array, and simply accessing it with the statement value = contents[{i,j}] solved that problem. Both experiences were rather frustrating.

11 Known Bugs

This is just a list of a few known bugs in the system, and their solutions (if any).

1. 2D time vs. altitude and time vs. speed visualizations go blank if you scroll left of t=0. Solution: prevent users from doing so.

2. Apparent glitches in data coming from GPS chip. Solution: implement checksum-checking and rejection of invalid messages.

3. Graphical glitches in 3D projection. Solution: use 42.16 fixed-point (or floating-point) arithmetic in matrix operations, vs. 32.16.

4. 3D projection often freezes. Solution: The immediate cause is known to be that the *advance* signal goes permanently low - root cause and solution unknown.

5. Leftmost two horizontal pixels in all rows do not display properly. Solution: modify VGA module to account for the two-cycle latency of the ZBT RAMs.

12 Conclusion

So, in the end, I did achieve my goal: build a system that can tell you where it's been. I got less done than I planned, and much less than I hoped, but the major functionality is there.

While there were some frustrating moments, I enjoyed this project. What I did get done was still very satisfying - it was great to flip the switch and see my 3D visualization scrolling around smoothly, even if I was only using 2 colors out of the 262,000 I built in support for. Somehow working on hardware seems more fun than working on software - although after spending three straight weeks in the lab until 12:00 AM, I have to admit the novelty has worn off a bit.

If I could start over again, there are innumerable things I would do differently. Looking at my code, you can definitely see a progression as I learn more about the language and the hardware, and try out different ways of doing things. As a trivial example, some of my FSMs have states named for what calculations they set up, and others have states named for what calculations they wait for (that the previous state set up). But there are more significant differences, too - the first modules I wrote were essentially state machines that did nothing but emulate the execution of ordinary computer programs, but gradually I gained more experience and learned to think about things from a more hardware-oriented perspective. It would be nice to go back and adjust my earlier code to benefit from what I learned in the later stages of the project.

All in all, it was an awesome course, and I learned a lot, while (generally) having fun. Thank you for the opportunity to take it.