The Local

6.111 Final Project

Chris Perez, Justin Pun, and Jonathan Varsanik
TA: Chris Forker
5/12/05

Abstract

"The Local" is a device to make people more familiar with their surroundings. The operation consists of two modes. In the first mode, the position of the device will be indicated on a map showing the surrounding area. When the device is pointed at a building and the information button is pressed, information about the indicated building will be displayed. The second mode is a tour guide mode. In this mode, the position will be displayed as before, but whenever the device is within a specified radius from a building, the information about the building will be displayed. The position of the device will be determined with a GPS module and the orientation with a compass module. This data is decoded and compared to a shared memory network, determining what is at that position. All the information will be displayed on a VGA monitor.

Introduction

The human race has always had an innate desire to know about their surroundings. The cavemen wanted to know about the occupants of the neighboring caves. Galileo suffered persecution for probing our position in the universe. And now, “the Local” allows modern day explorers to follow the same path as these brave pioneers.

“The Local” was designed to be a handheld device that will display its current location on a map of the surroundings. “The Local” has two modes of operation. First is the “Tour Guide” mode. In this mode, when the user nears a building, information about that building displays on the screen of the local, below the map, enlightening the intrepid adventurer. Also, the nearby building is highlighted, ensuring that the user is properly oriented.

The second mode of operation is the “Info” mode. In this mode, the user is free to wander about the area without any information being displayed. However, should the curious surveyor want to know more about a mysterious building seen in the distance ahead, at the press of a button, the knowledge shall be theirs.

To generate this awesome power, “The Local” was built form three main components. The first component, the Position Module, takes in signals from a GPS chip and 802.11 wireless signals to determine the position. The second part, the Identifier module, performs a search through a memory network of ROM modules containing physical coordinates with the goal of identifying any buildings of interest to the user. The final, Video module, takes stored map and building information, as well as the position, and displays them for the user to see.

Now, as Columbus ventured into the unknown, we will explore each of these modules in detail.

Position Module

The purpose of the position module is to provide the other modules with the current position. This is accomplished by decoding the signals from a GPS chip, using them to determine the location, transforming the coordinates to the coordinate system used for the rest of the system, and outputting the calculated coordinates. If the GPS is unable to determine a position, the position module also uses the signals from local 802.11 wireless internet access points to determine its location. The module is instantiated in the top-level file (labkit.v) as the GPStop.v module. The inputs to the system are: data from the GPS, data from the wireless receiver (laptop), the system clock, and the reset button. The outputs are the current coordinates.

Design Description

The position module contains four main components: RS232 Decoder, Serial Clock, GPS FSM, and Wireless FSM. These components are instantiated within the GPStop module, where the inputs are synchronized, the components are connected, and the outputs are selected with a multiplexer. The RS232 Decoder decodes the incoming serial signal and outputs the corresponding ascii data to the connected FSM. The serial clock is the clock that the RS232 Decoder uses to time the sampling of the incoming serial data. The FSMs interpret the incoming data and output the current position indicated by the data. The GPS FSM also outputs the “fix” signal, which is a part of the message sent by the GPS chip indicating if it is able to determine its current position. The coordinates are fed to a multiplexer that selects the position to output. Each module will now be explored in further detail. The code for each module is included in the Appendix.

GPS Top

The GPS Top module is the container for all the other modules. The GPS Top module provides a simple connection to the rest of the labkit and the other modules of the system. It takes the two data inputs: one from the GPS and the other from the Wireless receiver, as well as the reset, and outputs the position.

The position that it output is chosen from the positions produced by the GPS and the Wireless receiver based on the fix signal from the GPS. If the GPS is able to get a fix, than that position is used. Otherwise, we are probably inside, and the system switches seamlessly to the position indicated by the Wireless receiver. This operation is transparent to anyone outside of the GPS Top module. The components of the GPS module are connected in the manner indicated in the Figure below. There are multiple instantiations of the serial clock and RS232 decoder, one for each of the position signals that are possible.

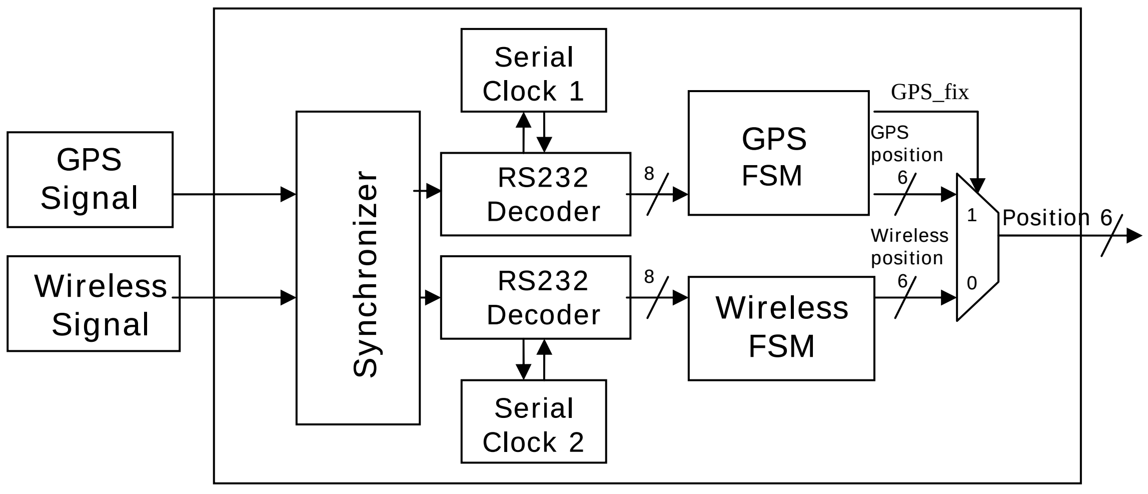


Figure 1: Block diagram of the Position module.

Synchronizer

The synchronizer is the same synchronizer that has been used in the past two 6.111 labs. It is important to synchronize all inputs to avoid the dreaded metastability. The synchronizer module was one bit wide, but I created multiple instantiations: one for each bit of input. The GPS, wireless, compass, and reset signals all went through a synchronizer. The data from the GPS and Wireless receiver then went to their respective RS232 decoders.

Serial Clock

The serial clock was the module used to time the sampling of the incoming serial data from the GPS and the Wireless receiver. The important parameters are the clock frequency, the baud rate of the serial data, and the number of clock cycles per data bit, which is the quotient of the previous two parameters. The serial clock takes as an input the "counting" signal from the RS232 decoder. When this signal goes high, the serial clock begins counting. The module will count until the "counting" signal is set low, at which point it will reset the count to zero and prepare to begin counting again.

When the counting is running, count increments with every cycle of the 27-Megahertz clock until the count value is equal to half of the clock cycles per data bit minus one. At this time the output, enable, is set high for one clock cycle. The counter continues incrementing until it is equal to the number of clock cycles per data bit. When the counter reaches this value, it is reset to zero and the module continues counting from zero again.

In this manner, the serial clock pulses the enable signal once every period of the incoming serial data. However, it also has an offset, so the enable signal is pulsed in the middle of an incoming data bit. This method of operation was chosen to maximize the probability of sampling the correct bits. If the module were to sample near the edge of an incoming bit signal, it is possible to have some delay cause the sampling to occur on a different bit.

The serial clock also takes the global reset as an input, and when this is asserted, the counter and the enable signal are set to zero. Now that we understand the sample clock, we will look at how this clock signal is used to decode a serial signal.

RS232 Decoder

The RS232 Decoder module samples the serial input data to determine the eight-bit ascii characters that are being sent. It basically waits to detect the start bit on the signal, upon which it starts the serial clock. On every pulse of the enable signal from the serial clock, it samples the next bit until it reaches the stop bit. This process is pretty straightforward; the only difficulty is knowing when the start bit occurs. This is done in the first state, WAIT. The state transition diagram for the RS232 Decoder is shown on the next page.

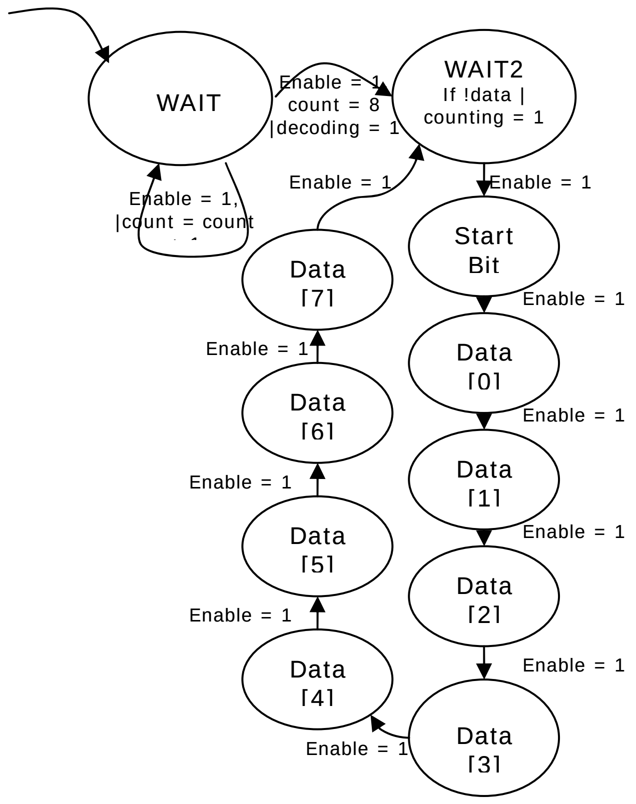


Figure 2: State transition diagram for the RS232 Decoder

The wait state is the initial state of the module. This state is where the module ensures that it will detect the start bit. The data line is high when it is idle, and the start bit is one bit low. Therefore, the start bit is a low proceeded by many highs. So, in the WAIT state, if the signal is high, the counting signal is set high, starting the serial clock, and a counter is incremented on every pulse of the enable signal when the signal is still high. If the count ever reaches ten, we know that the data is in its idle state, because all signals sent are eight bytes preceded by the low start bit, and therefore no signal could ever be high for 10 bits. So, when the count is ten, the WAIT state continues, but the counter no longer increments. Finally, when the data goes low, if the count is less than ten, we know that we began sampling in a byte of data, so we should not begin decoding. In this case, the counter is set back to zero, the module remains in the WAIT state, and waits for the data to go high again to begin the counting again. If count is equal to 10 when the data goes low, we know that this is the start bit for the first byte of a message. In this case the counting signal is pulsed low to reset the serial clock so we are synchronized with the data. Also, the internal register, decoding, is set high, indicating that we have found a start bit, and are now locked onto the signal. The module will not have to wait to assure detection of the start bit again. We now know when the start bit will be coming.

The next state is the WAIT2 state. This is the state where the module waits for the start bit once the decoding bit is high. Because there is a specified number of bits in a message and they are contained within a start and stop bit, the counting no longer has to be performed to assure that the module does not mistake a signal with a high-low transition for the high-low transition of the start bit. So, in the WAIT2 state, when the signal goes low, the counting signal is set high, starting the serial clock. The state is moved to the start bit state.

The next ten states detect the ten bits in the transmitted signal. On the pulses of the enable signal, the data is sampled, and stored in the appropriate registers of the data signal. The first and tenth bits are not stored, as they are the start and stop bits. The start and stop bits are checked to see that they are low and high, respectively. If these bits are not correct, something is wrong, and the module returns to the WAIT state to try and get back in sync with the signal. If all then states are processed, the eight-bit value of data is put into the data output register, and the data ready output is pulsed high. When this signal is pulsed high, the FSM that is collecting this data will know that it is ready to sample the data byte.

The module then returns to the WAIT2 state, because it knows that the next time the signal is low, it will be on the start bit. On reset, the module is put into the WAIT state, and the counting, decoding, data ready, and data registers are set to zero. Next the FSMs that interpret the decoded ascii data will be explored.

GPS FSM

The GPS FSM parses the ascii data that is coming from the GPS through the RS232 decoder. The message from the GPS chip is a comma separated list of ascii characters. The GPS samples the ascii data from the rs232 decoder module when the data ready signal from that module is pulsed high. Each of these bytes are put together to form a message from the GPS chip. The message starts with a standard header, and terminates with a standard sequence. By listening for the header and then stepping through the states of the FSM whenever a comma is detected, the GPS FSM is able to extract the position of the module. A state transition diagram is not included because the transitions are entirely linear, and it is possible to therefore describe accurately without the need for a diagram.

The first state of the FSM is the IDLE state, where the module waits for the initial character of the header, “$”, which corresponds to a decimal value of 36. When this value is detected, the FSM moves to the HEADER state, where it looks for the next character in the header, “G”, which corresponds to 71. Because the GPS sends several message formats, it is necessary to ensure that we are reading the correct message. If this value is not detected, the FSM returns to the idle state, as this is therefore not the message that we are looking for.

Then, on every high pulse of the data ready signal, the GPS FSM continues to step through each character of the header, ensuring that the message is the proper one. When it receives a comma, and if the header was correct, the FSM moves to the TIME state.

The information that the GPS sends immediately following the header is the time. This is a nine-byte value of the UTC time given by the satellites to 1ms accuracy. As we do not need this information, the data in this state is simply not used. The FSM waits for the ascii data to be equal to 44, which designates a comma. Only then does the FSM move to the LATDEGREE state.

After the time, the GPS sends the Latitude coordinates. These coordinates are in the format of Degrees * 100 + minutes. So, the first three digits are the degrees of latitude. For the scope of our project, we were always in the same state, so the degrees were not going to change. As a result, this state incremented a counter with every successive pulse of the data ready signal until the count reached three, when the FSM moved to the LATMIN state.

In the LATMIN state, the GPS FSM receives the minutes component of the latitude coordinate, which do matter for this project. The degrees are sent in the format of two characters, a decimal point, and then three characters. So, with each pulse of the data ready signal form the rs232 decoder, the GPS FSM first checks to see if the value is 46, which would represent the decimal point. If the decimal point is encountered, nothing is changed and the FSM remains in the LATMIN state to sample the values after the decimal point on the next pulse of data ready. When the value is not 36, but represents a number, the GPS FSM multiplies the current value in the latitude register by ten, shifting it to make room for the next digit. Then, the module takes the value in ascii data, subtracts 48 from it (which converts it to the equivalent integer) and adds it to the value in the latitude register. With this process, the decimal values are appended to produce the minutes value of the latitude coordinate (multiplied by 10000). The message protocol is very specific about the number of digits that will be sent, so we can count on there always being this number of digits, even if they are trailing zeros.

At any time in this process, if a comma is encountered, the data in the register is then transformed from the GPS coordinates to the locally flat coordinates used in the rest of the system using a hard coded coordinate transform. If the system were to be used outside of Massachusetts, another coordinate transform would have to be programmed in. After performing the coordinate transform, the FSM moves to the next state, NS.

The NS state is for the part of the message where the GPS indicates if the Latitude coordinate is in the Northern or Southern hemispheres. Since we are not going to be taking this project down south right now, this value was ignored. When the comma is detected, the next state is LONGDEG.

Now, the GPS sends the Longitude data in the same format that it sent the Latitude data. The GPS FSM treats them the same, too. The first three digits here are ignored, as they will not be a factor in the scope of this project. The next state is LATMIN.

In LATMIN, the FSM evaluates the Latitude coordinates the same way as the Longitude coordinates were evaluated before. However, this time the values are stored in the latitude register. Also, the coordinate transform is slightly different, to account for the spherical asymmetries in the globe and the difference between Latitude and Longitude. After the comma, the next state is EW, where the data indicating if we are in the Eastern or Western hemispheres is ignored.

The next state is the FIX state. This information from the GPS is the signal that indicates if chip was able to get a fix. This is 49, which corresponds to a “1” if a fix is available, and 48, which corresponds to “0”, if a fix is not possible. This result is put into the fix register and output. This output is used by a multiplexer in the top module to determine which position information to use.

There are seven more states corresponding to the seven more pieces of the message. This data is ignored. It is possible to get this information if a user were interested, but the project right now wants the positioning to be transparent to the user, so information about number of satellites and other GPS specific datum are not important. These states, therefore just listen for the comma, and then transition when it is detected. When the last state is reached, and it detects a comma, the GPS FSM pulses the position ready signal, indicating to the top module that the position has been updated and it is safe to sample. The GPS then returns to the idle state to listen for the header again.

Wireless Input

When a user ventures indoors, we would like the system to still function. However, due to its high frequency (which is necessary for the accuracy of the GPS system), the signals cannot propagate into buildings, or even through a dense cloud cover. If this is ever the case, we need a way to determine the current position. The method that we used to implement this extension of our project is to decode 802.11 wireless signals from network accespoints. These accesspoints can be queried for their unique identifier. With two accesspoints, their identifiers, a knowledge of the accesspoint locations, and the signal strengths coming from the accesspoint, it is possible to triangulate the position of the user.

This portion of the system was implemented outside of the labkit, on a laptop. This was chosen because the wireless information is easily decoded with the existing architecture on a mobile computing platform. Also, the unique identifiers of the accesspoints require 95 bits to store, so the position calculation was also performed on the laptop to avoid the hassle of these ridiculously wide busses, or the need to encode 95 states to parse and compare identifiers in order to determine locations.

The information was gathered using the NetStumbler program, which queries local accesspoints, and records relevant information, including their identifier, and signal strength. Then, a script was written in VBScript to parse this information while the program is running, and calculate the position. To calculate the position, the script made a list of all known accesspoints that responded to the query that was sent out. Then, stepping through this list, their positions are averaged, weighted by their signal strength, to triangulate the user's position relative to all available accesspoints. This data is then sent through the computer's serial port to the labkit, where the position information is decoded and used.

Wireless FSM

The Wireless FSM is very similar in operation to the GPS FSM. It listens for a start sequence from the Wireless input and then extracts the coordinate information from the following comma separated list of values. The difference, however is that the coordinate processing is done before this system sees it. So, the Wireless FSM, extracts the coordinate information, and does not have to do any calculations. When the data is ready, the Wireless FSM pulses its position ready signal.

The first state in the Wireless FSD is the IDLE state, where it waits for the header sequence, which in the protocol that I designed is "$". When this value is detected, the FSM moves to the next state, XPOS.

In the XPOS state, similar to the position states of the GPS FSM. On the data ready signal, the data is checked to see if it is a comma. If it is, then the FSM moves to the YPOS state. Otherwise, the current value in the x register is multiplied by ten. Then the incoming data is sampled, converted to the corresponding integer, and added to the value in the x register. This parses the integers to form the transmitted decimal position.

In the YPOS state, the operation is identical, except the y register is updated. Also, when the comma is encountered, indicating the end of the message, the position ready signal is pulsed, indicating that the position is safe to be ready by other modules.

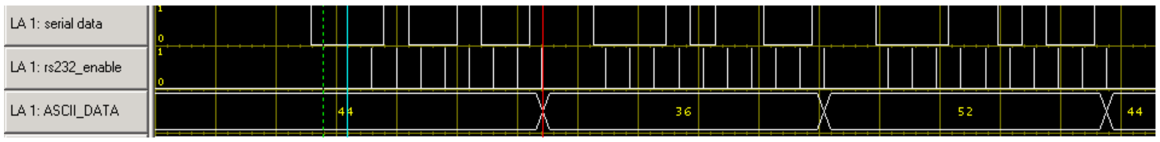
Methodology

The position module was designed to reliably decode the information sent over a standard protocol. The known design of implementing a divider and having a separate module to decode the bytes seemed logical. Finally, an FSM that stepped through the message syntax, extracting the relevant data made the system simple to debug and offered the opportunity for expansion.

The implementation of the Wireless portion was a bonus, brought about by the necessity to demonstrate this project inside. It offers an exciting addition to the project. The position processing was done in the laptop instead of the labkit to avoid the huge busses that would be necessary for the identifiers of the accesspoints.

Testing/Simulations

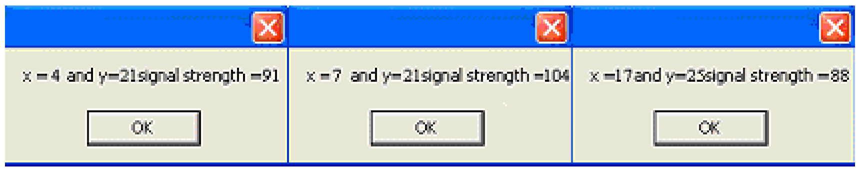
Each module was independently written and simulated. When this was completed, it was easy to test the operation of the pair of the Serial Clock and RS232 Decoder. A serial cable was connected from the computer to the labkit. Using the Hyperterminal program, we were able to send simple test signals to evaluate the functionality of the decoding system. Below is a picture of these components correctly decoding some transmitted characters. Note how the enable signal is in the center of each data bit, and does not occur between bytes. The one problem that I did have with the interface was the Serial Clock. I was using the given parameters of the clock rate and the data rate, and dividing them in the program, but this resulted in a non-integer, so the count was never equal to that value, and the module never would pulse the enable signal. This problem was discovered through use of simulating with different values for the parameters.



Next, we were able to hook up the GPS and the Wireless receiver to this simple system (through synchronizers) to be sure that we can get the ascii characters from the signals. It is important to note that this was the only sunny day that we were testing the GPS, and the only time that a position was read from it. All the other days it was cloudy, and a fix could not be obtained, even when the chip was placed near a window. That is why the wireless expansion to the project was implemented. While the system was origionally intended to be solely an outdoor system, the possibility of position resolution while inside was an exciting bonus.

Finally, the whole GPS Top system was put together and simulated. The Appendix contains a screenshot of a simulation of it decoding and transforming some sample coordinates.

Finally, to test the accuracy of the Wireless receiver, I changed to program to output the data to the screen of the laptop, instead of to the serial port. Then the laptop was taken through the lab to ensure that the indicated location moved. Below are three examples of different positions being displayed.



```
x =4 and y=21signal strength =91
```

```
x =7 and y=21signal strength =104
```

```
x =17and y=25signal strength =88
```

Building Identifier Module

Summary of Functionality

The building identifier module acts as the main interface between the device and the FPGA ROM memory elements containing the physical locations of each of the buildings of interest. The building identifier module takes as input physical coordinates of the device with respect to some predetermined origin point. The identifier also takes in as input directional data that is continuously produced by a digital compass wired onto the 6.111 lab kit. When the device is operating under its normal mode of operation, the identifier module simply passes the physical location of the device to the video UI module, where the device's current position is indicated on the video monitor. Also handled by the identifier module is the mode-switching feature of the device. Anytime the user switches the 'tour mode' switch on, the identifier module initiates a search mechanism which performs a parallel linear search through each of the ROM memories, with each ROM memory element containing coordinates of only one building. If the location of the device is found in any of the building memory elements, than the search mechanism quits while the identifier module passes the building number of the building the device is currently located in to the video module. Another feature of operating in tour mode is that if the device is not located in a building but is instead outside and within approximately 10 meters of a particular building, then that building's number is sent to the video module. Thus the device, when operating in tour mode, simulates a tourist device, giving information to the user holding the device about a specific building anytime the user approaches that building.

The identifier module also handles the 'point-and-click' building identifier feature. This feature allows for the user to point the device at a specific building when outside and get that building's information at the press of a button. In addition to this, the identifier is able to locate buildings in the line-of-sight of the device even when the device is located inside a building. Thus anytime the user looks out a window and wishes to know what building he or she is looking at, or just wishes to know what building is north of the building the user is currently in, the info button can be pressed and information about the particular building being pointed at will pop up on screen.

Identifier Sub-Modules

What follows here are detailed descriptions of the sub-modules of the building identifier system essential to the functionality of the identifier as well as to the functionality of the device as a whole. The detailed functional block diagram shown in Figure 1. illustrates how all the sub-modules are tied together and how they interface with the ROM memory elements within the FPGA.

Synchronizer and Interface to Other System Modules

Because the user controls the mode of the device as well as the system reset through switches wired into the lab kit, a synchronizer is used to keep the signals from the external environment synchronous with the system clock of 27MHz. Furthermore, in order to prevent any glitches in the output coordinates (a 12-bit signal) from the device locator module as well as in the directional data from the compass (a 4-bit signal), these signals are registered and thus delayed by one clock cycle before the identifier module reads them in. The 15-bit output point representing the device location as well as the indicated building number (a 6-bit value) is registered before it is sent to the video/user interface module.

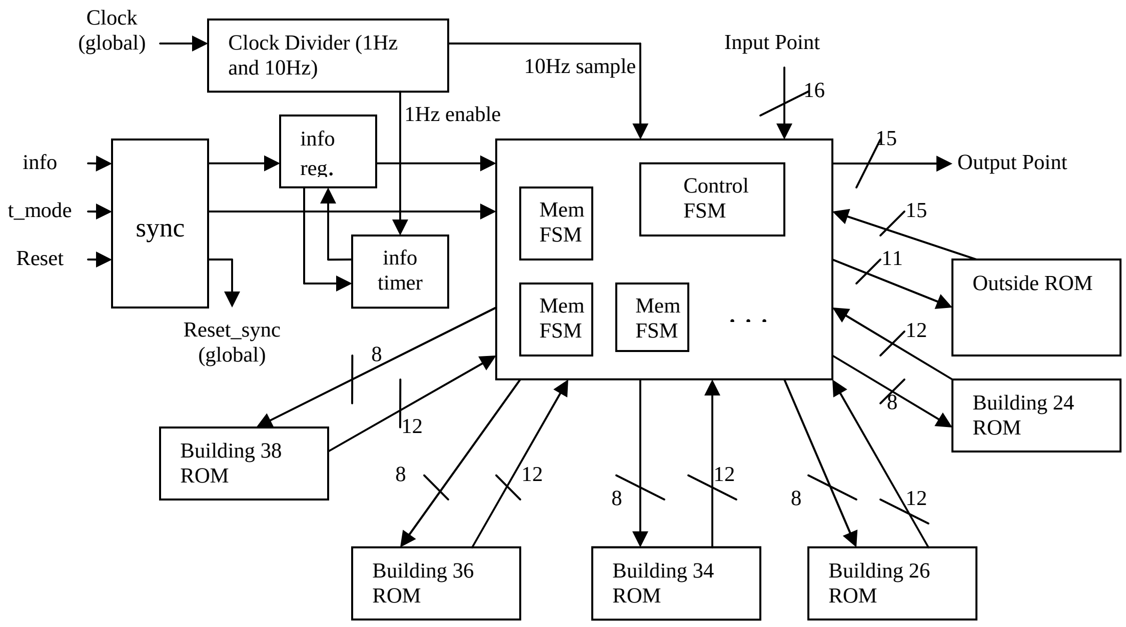


Figure 3. Functional Block Diagram for Identifier Module

Info Button Register and Timer

When the user presses the info button, the info signal is held low (due to the negative logic of the lab kit buttons) for multiple samples of the input point in order to keep the information about the building being pointed at on screen for a useful length of time. To carry this out, a special register and timer module was implemented to hold the info pulse low for a 10 second interval. While the info pulse is being held low, any subsequent info button presses are ignored.

On the button press, a trigger is sent to the timer module which counts off ten periods of the 1Hz enable signal produced by the clock divider module. Once the count reaches its final value, an expire signal is sent to the info register, where the info signal is brought back high until the next time the button is pressed. The Verilog code describing this modules functionality can be found in the appendix.

Memory Storage of Coordinates

To assign coordinates to physical locations on MIT's main campus, points were read off of a scaled map of MIT. For the purposes of simulation and testing, the map was restricted to a specific area around MIT's Building 34 and its four closest surrounding buildings. This area, approximately 18,225 m^2 of MIT's main campus, makes up the operating region of the device, outside of which the device cannot operate. An image of the map we were concerned with in designing this device can be found in the appendix.

Because of the limited accuracy by which the locator module can pinpoint the device's physical location on the map, a resolution of about 3m was chosen to work with. A 45x45 point grid was juxtaposed onto the scaled map. This grid was designed so that there would be approximately 3m spacing between each physical location, both in the N-S and E-W directions, with each point corresponding to one of the points on the map. Because the design process would go by smoother if negative values were disregarded, the top-left most corner of the map was chosen to be the origin of the map. Thus any physical location within the 135mx135m operating region could be addressed by a 6-bit positive x-coordinate ranging from 0 to 45 and a 6-bit positive y-coordinate also ranging from 0 to 45 (the x-axis of the grid on the map was aligned in the E-W direction while the y-axis was aligned in the N-S direction).

The procedure used to store physical coordinates onto ROM memory blocks within the FPGA is simple. Each ROM block contains all of the 12-bit coordinates located within the walls of one specific building. The ordering of the points within the ROM is of little importance for this application; because the sample period of 100 ms is so large compared to the system clock period of approximately 37 ns, a simple linear search through an arbitrarily ordered set of coordinate points proves to be sufficient enough.

The address space depth of each building ROM is 256 words, with each word stored in memory being 12 bits long. The six MSBs of each word are used to store the x-coordinate while the six remaining bits represent the y-coordinate. Such specifications of these building ROMs show that only one 18kb block of the 144 SelectRam™ blocks within the Xilinx FPGA is required to implement each ROM.

For points not located within any of the five buildings the device can operate within, coordinates are stored within a separate ROM with an address space depth of 2048 words, with each word being 15 bits long. The three MSBs of each word stored in this ROM are used to represent the building closest to that point whose coordinates are stored in the remaining 12 bits of the word. In order for a building to be ‘close’ to a point outside, that building has to be within 10 meters of the point. If there is no building within that proximity of the point, then the three MSBs for the word are set to 0. The numbering scheme used to refer to the five MIT buildings is as shown in Table 1.

Building	Number Assigned to building
38	1
34	2
36	3
24	4
26	5

Table 1. Internal Numbering Scheme for Buildings

Once each memory initialization file is written for each of the six ROM elements, the Xilinx Core Generator was used to create the core for each of the memories. These memories interface with the identifier module through minor FSMs, with each minor FSM handling the reading from one specific ROM.

Reading Building/Outside Minor FSM Operation

The state transition diagram for the minor FSMs used to read from each of the building ROMs and the ROM storing points outside is as shown in Figure 2.

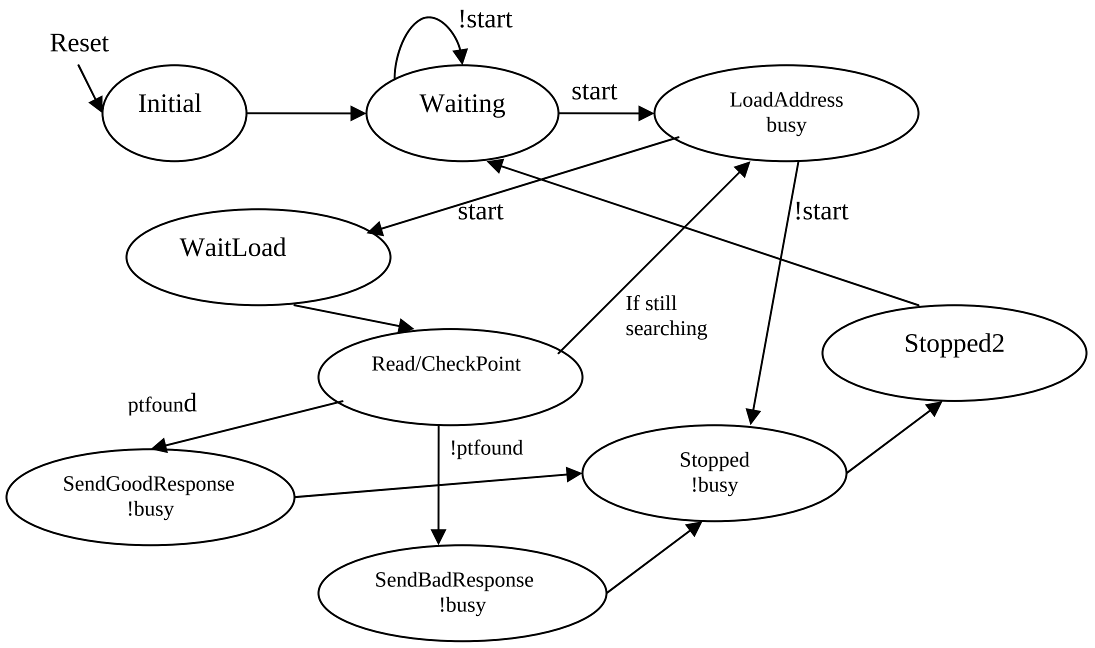


Figure 4: State Transition Diagram for Minor FSM Controlling Memory Reading

As can be seen in Figure 4, the minor FSMs all are in a wait state waiting for the trigger signal to be set high by the major control FSM within the identifier module. Once the trigger goes high, the FSM cycles through a main loop consisting of the LoadAddress, WaitLoad, and Read/CheckPoint states. Each time the FSM enters the LoadAddress state, the 8-bit address value (or 11-bit in the case of the minor FSM controlling the ROM storing outside information) is incremented by one. In the Read/CheckPoint state, the point output by the ROM is compared to the 12-bit search point sent out to all minor FSMs synchronously. If the point is found in the ROM, the FSM enters the SendGoodResponse state, where a response signal is set high and sent to the control FSM. If the point is not found in the ROM and the address has cycled through all available addresses within the ROM, the FSM enters the SendBadResponse state, where the response signal is set low. Upon completing the search, the FSM enters the Stopped states and returns to the Waiting state to await the next search. One nice feature of this FSM is its ability to bypass the search and enter the Stopped state once any of the other minor FSMs sends a high response indicating that the point has already been found elsewhere.

The minor FSM reading from the ROM storing locations outside of the buildings operates very similarly, except with one difference. Because the incoming point from memory is 15 bits long, the search point is only compared to the 12 LSBs. If the point is found, then a register stores the three MSBs, corresponding to the closest building from that point outside. If there is no such building within 10 meters of that point, then the register will just store the value 0. Thus, this FSM outputs also the number of the closest building from any point outside. The Verilog code describing the minor FSMs is located in the appendix.

Major FSM Operation

The state transition diagram for the major FSM used to control the operation of the minor FSMs is as shown in Figure 3. The control FSM cycles through a set of 15 states and takes in as input the info and t_mode signals, as well as the 10Hz sample signal produced by the clock divider module. The 16-bit device location and orientation is also input and used to determine the 12-bit search point sent out to the minor FSMs. The responses from the memory modules (as well as the closest building 3-bit signal) are sent in from the minor FSMs and used to determine state transitions.

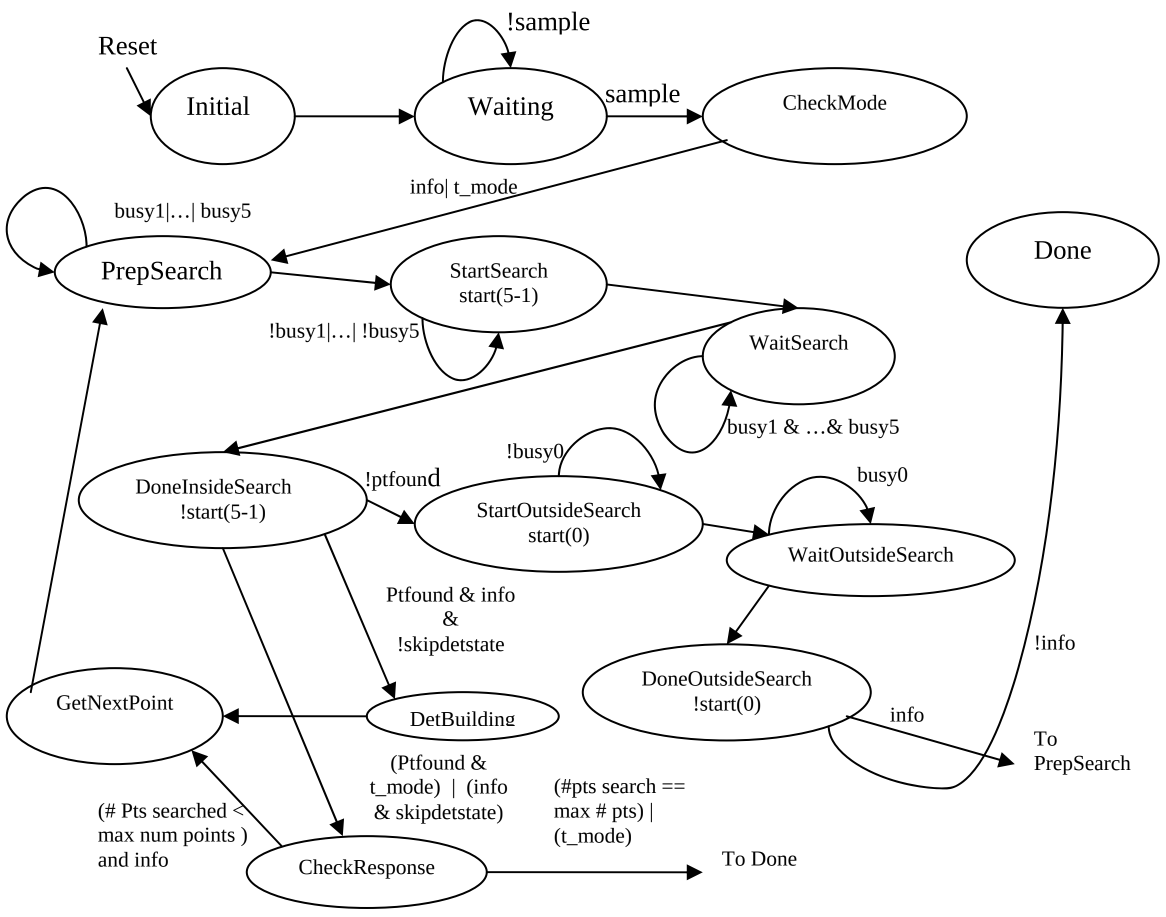


Figure 5. Major FSM State Transition Diagram

Upon reset, the Initial state is entered. Once the sample pulse is set high, the major FSM begins its operation by entering the CheckMode state where the FSM determines what mode (info, t_mode) the device is operating in. If the device is in tour mode or info mode, the search point and trigger signals are sent to the minor FSMs where the search through all the buildings for the current search point is initiated. If the point can’t be found inside a building, the search through the points outside begins. If the point is found inside and the device is operating in tour mode, the building number of the building the device is in is output to the video/UI module.

If instead the device is operating in the info mode, then the current building the device is located in is first determined. Once this building is determined, the FSM uses the directional data from the digital compass to determine the coordinates in the line of sight of the device. A search for each one of these points is performed, and once a point in the line of sight is determined to be from a building different from the building the device is currently in, then that building number is sent to the video/UI module where information about that building pops up on screen. When the device is located outside, operation in the info mode is a bit simpler. The line of sight coordinates are searched until one is found within a building. The building number for that building is then output to the video/UI module. Verilog code used to implement this control FSM can be found in the appendix.

Testing and Debugging the Identifier Module

Testing the identifier module turned out to be a simple task because of its simple input/output structure. Various points on the map were sent to the module as input, and the resulting outputs checked. Anytime an unexpected output turned up, signals were sent from the top module of the device to the logic analyzer for analysis. It turned out that most of the mistakes made during the design process could be fixed by paying more attention to timing issues and making sure all signals sent from one module to another independent module were registered and kept glitch-free. Many hours were spent trying to discover hidden glitches that could only be uncovered from an analysis of the logic analyzer outputs.

Video Portion: Overview

The video portion displays a map stored in memory, the current position of the device, highlights any buildings that the device is near when in “tour mode” or any buildings that the device is pointed at when the information button is pressed, and displays building-specific information for a highlighted building. Maps are represented as two-dimensional images, oriented with respect to NSEW cardinal directions.Each building information contains the highlighted building’s number, its name (if applicable), its street address, and any interesting features of the building. The general approach that is used to generate the video display is to first generate bitmap images using Adobe Photoshop, Microsoft Paint, or a combination of the two, use MATLAB to convert these images into data (RGB decimal indices) in the form of comma-separated value (.csv) files, modify the values of the indices to the desired values with Microsoft Excel, replace all paragraph breaks and commas with manual line breaks, verify that the files are in the proper format with Microsoft Command Prompt, generate .coe files from the .csv files with the memory editor tool in the Xilinx CORE Generator, create virtual ROMs initialized with the .coe files using the CORE Generator and use the information stored in the ROMs to determine the appropriate RGB values to be outputted to the ADV7125 chip in the lab kit, from which point the rest of the process of generating the proper signals to the VGA connector is internal to the labkit.

Video Portion: Module Description/Implementation

The entire video portion is implemented in the FPGA. Two modules are sufficient for handling the logic of the video portion: the Video Module and the Display Module, which will be discussed in more detail below. One ROM is created for the map being used and a separate ROM is created for each building information. The organization of the circuitry is shown below in Figure 1.

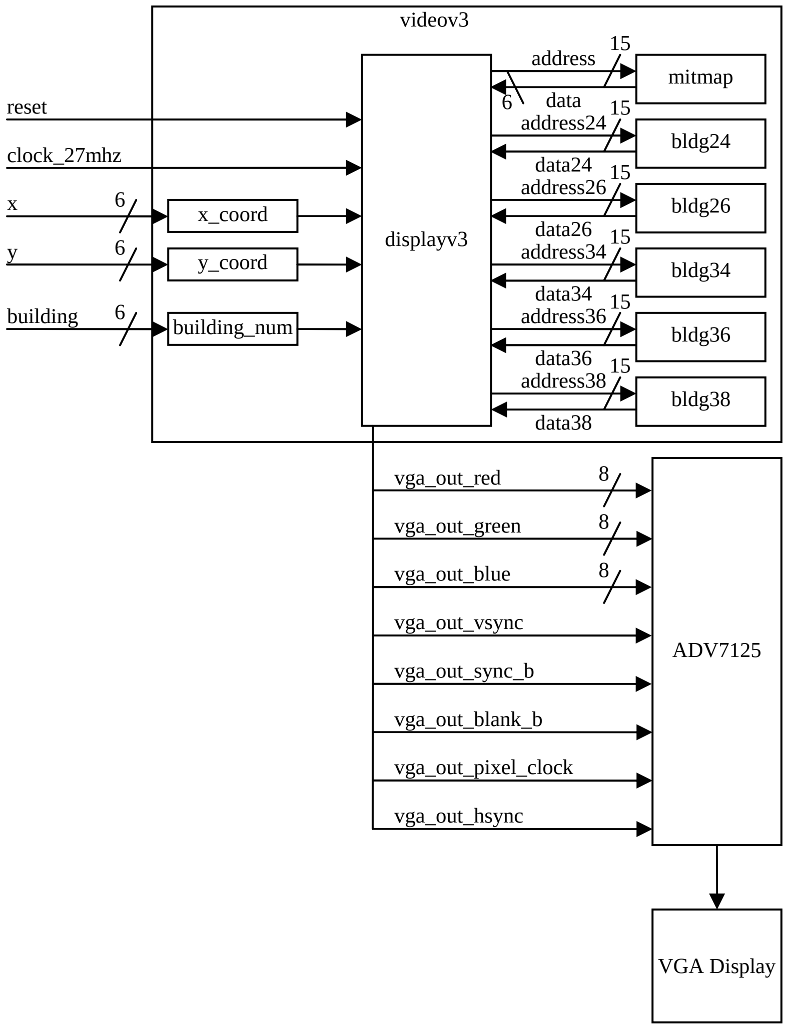


Figure 6: Detailed Block Diagram of Video Portion

Video Module

The Video Module is the top-level module of the video portion. It handles the instantiations of all of the ROMs and the Display Module as well as the wiring between them. The ROMs are timed on the pixel clock outputted by the Display Module to ensure that the Display Module will be able to read from the ROMs at the appropriate frequency. It has five inputs: reset, clock_27mhz, x, y, and building. The reset signal is not accounted for in the video module, but it is passed along to the display module. The clock_27mhz is a 27Mhz clock signal internal to the labkit. This clock signal is also not accounted for in the video module, but is passed along to the display module. The x and y inputs are the current x and y coordinates of the device represented by 6-bit numbers. These numbers are relative to the upper left-hand corner of the map (where x and y are both zero). The x and y inputs are stored into their corresponding registers: x_coord and y_coord, which are independently triggered to update as soon as there is a change in x or y. The building input is the building number of the selected building (or zero if no building is selected) represented as a 6-bit number. The building input is stored into a building_num register, which is also triggered to update as soon as there is a change in the building input.

Display Module

The Display Module handles all of the logic necessary in determining the proper values to output to the ADV7125. Its reset input resets all pixel, line and frame counters to zero, all ROM read addresses to zero, and all sync signals to one (sync signals are active low). The Display Module takes the clock_27mhz signal (discussed above) and uses it to generate the 40.000Mhz pixel clock signal needed to drive an 800x600 resolution video output at 60Hz by utilizing digital clock managers (DCMs). The x_coord and y_coord inputs are used to determine where the device is currently located on the map. The location is indicated by a small green square, which is accomplished by finding the current position and coloring that pixel, along with a few adjacent pixels in each direction, the same green color.

Each address in the map ROM contains data for a single pixel of the video. Each pixel's data is a 6-bit number used to indicate whether the pixel is part of a building or not and, if so, which building that pixel belongs to. Thus, the map ROM consists of zeros and building numbers which can be quickly compared with zero and the value of the building input to determine what color the pixel will be. If the pixel data is equal to zero, the pixel should be black. If the pixel data is not zero, the pixel should be blue – unless the pixel data is equal to the value of the building input, which means that the pixel belongs to the currently selected building and thus the pixel should be red. If the building input value is non-zero, which indicates that the building with the number equal to the building input is selected, the corresponding building information is displayed below the map. Thus, the Display Module also has a 1-bit input from each of the building information ROMs. This informs the Display Module whether the next pixel drawn should be black (0) or white (1) when drawing the portion of the video that corresponds to the building information.

The addresses used to access the ROMs are controlled in a very image-size specific manner. The addresses are incremented along with the pixel counter at every clock cycle. However, they are not accounted for beyond the borders of the images. They are then reset to the proper values to continue drawing their respective images at the appropriate time.

Appropriate timing values for the video output (meaning the values for the active video, front porch, sync pulse and back porch for both the horizontal and vertical signals) were assigned based on the VGA Timings given on the lab kit's web page:

http://www-mtl.mit.edu/Courses/6.111/labkit/vga.shtml.

Video Portion: Design Decisions and Trade Offs

Due to time and resource constraints, many design decisions were made to expedite the completion of a working model of the project. RGB colors were hard-coded to save time, space, and simplify implementation. Initializing the map ROM with building numbers instead of RGB indices allowed for quick comparisons with the value of the building input to determine which building (if any) is selected and for the removal of the need for a RGB index lookup-table ROM to determine what combination of red, green, and blue values to output. Use of a RGB index lookup-table ROM would also require extra timing considerations to be taken into account. A helper FSM would most likely be needed to ensure that proper RGB values would be ready when the Display Module needed them. Also, if RGB indices were used instead of building numbers, the map ROM's width would need to be increased. Thus, not only would an additional lookup-table ROM be needed, the map ROM's size would need to be larger as well – taking up valuable space in the FPGA.

The external Flash ROM of the lab kit would have been large enough to store much larger map images and more stylized building information. However, the initial read time of the external ROM ranges between 110 and 120 ns, which is much too slow for a 60Hz display running a pixel clock of 40.000MHz. The average read time after the initial read time is 25ns, which is just equal to the period needed for the display. Thus, hypothetically, if a busy/wait signal were implemented to account for the initial read delay, the external ROM could be used to store information used to drive the display. However, rather than deal with the timing issues and investing a large amount of time (the process would require loading chunks of data one at a time into the FPGA to be transferred to the external ROM) in a design that may end up not working, it was decided that a smaller scaled implementation with much less risk be employed.

Hard-coded values were also used in logical handlers for pixel and line counters as well as addresses. The current implementation is a very image-size specific one. It would work very well with small handheld devices where the image size is constrained. An alternative implementation would be to assign special values to the data representing the edges of images. These special values would allow the display to scroll images through a more interactive user interface without over-scrolling. Thus, the current implementation is great for size-constrained images, however, if desired features include zooming and scrolling, an alternative method of implementation may be desired.

Video Portion: Testing and Debugging

After completing a few versions of the video and display modules (where each version added a new feature), the long process of generating proper ROMs to test the display modules began. Upon initial connection, the video was a distorted arrangement of the correct colors. Noticing that there was a skew involved, the problem was identified as an incorrectly hard-coded image size. Updating the hard-coded value to represent the actual image size solved the problem. The image, however, was still too far to the left and upward so that part of the map was cut off. The LCD monitor used to display the video inexplicably reported that the VGA signal generated by the lab kit ran at 63Hz instead of 60Hz, which may explain why the image was not centered. Mathematically, all parameters were specified to generate a 60Hz signal. Tweaking parameters only made the video worse. Ultimately, adjusting the monitor's settings corrected the positioning of the video.

When attempting the implementation of a FSM to handle the displaying of building information, the building information appeared as both scrolling and flickering underneath the map. Testing the displaying of building information also revealed an overlay of the map image on the area of the display below the map, which was the result of a missing conditional statement. Adjustments made to the code controlling the displaying of building information only adjusted the apparent direction (the image can appear to be moving left when in reality all lines are drawn left to right) and speed at which the building information scrolled and did not resolve the flickering. Desire to create a working model resulted in the decision to implement the displaying of building information in a method analogous to the method of displaying of the map. Thus, the FSM design was scrapped along with the idea to allow for colored pictures of the buildings to be displayed as part of the building information since it would require a RGB index lookup-table ROM and thus a FSM for timing purposes.

Conclusions

This project built upon our knowledge from 6.111. Specifically it built on the knowledge of the importance of modularity, and to simulate and test each module. Specific knowledge that was gained includes wireless protocols, serial data transfer protocols, GPS protocols, and to not use a GPS if I am going to be inside. Also, data entry, memory structure, and search algorithm skills had to be built and solidified in order to complete the project. Finally, experience was built in the areas of ROMs, video timing, sync signals, color maps, and video displays.

The project was an exciting endeavor, and we are all excited about the infinite possibilities that we can explore in the future.

Acknowledgements

Thank you to Charlie, he told me how to spell "clock."
Also, some code from the new labkit tutorials was referenced to get an idea for system architecture in the serial interface and video portions of the project.