Digital Whiteboard
6.111 Final Project

Nathan Davis, Shaun Foley
Javier Castro
May 18, 2006

Abstract

We implement a Digital Whiteboard system controlled by two accelerometers. Whiteboards present interesting venues for novel computer interfaces, as the traditional mouse+keyboard approach destroys the spontaneity afforded by whiteboards. We use two accelerometers, one to control drawing and one to give commands. The implementation is roughly split along the same lines. Although the project was not fully completed or integrated, it does show the feasibility of such an interface.

Contents

List of Figures

List of Tables

1 Introduction - ND, SF

Our project, The Digital Whiteboard, began with the idea of using accelerometers to control a system. Whiteboards are a good candidate for interfaces where the typical mice plus keyboard is insufficient. Instead, users use one hand as a cursor and control drawing commands via gesture recognition with the other hand. To accomplish this we make use of accelerometers to be mounted on hands, gloves, or a wand. The accelerometers provide high resolution acceleration data which is in turn used to produce velocites and positions that update in a realistic fashion as the accelerometers are moved. With these data we control an onscreen VGA interface. We wanted to create a highly modular design to allow easy interoperability and reconfiguration of the system. Our design methodology was tailored around a flexible submodule design and an intuitive user interface.

The original system contained three sections, an accelerometer interface, a movement interpreter, and a whiteboard controller. The first converts accelerometer data to digital form via an ADC then massages it through filters and integrators to provide velocity and position data. The second interprets movement sequences as concrete commands, which are then passed on. Finally, the whiteboard controller takes these command and position inputs and performs the desired action.

2 Module Descriptions

2.1 Accelerometer Interface - ND

2.1.1 AD670 Controller

Given that the project required the use of analog accelerometers, we had to implement an analog to digital conversion system to interface the accelerometers to the labkit. We selected the AD670 analog to digital converter. The AD670 is an 8-bit, single channel ADC with a sampling rate of 100 kilosamples per second. In order to properly control the ADC a controller interface was written.

The Controller Module takes as inputs the 31.5MHz system clock, a global reset, and a status signal. The outputs of the module are dataavail and r_w_bar. Within the Controller Module a 9 state FSM is implemented to properly time the interaction with the ADC. A counter is implemented providing an enable 10,000 times per second to the FSM. The default state in the FSM is IDLE, in which dataavail is logical low and r_w_bar is logical high. When the IDLE state of the FSM receives an enable it transitions to the CONV0 state. In CONV0 r_w_bar is brought low and a separate counter is started. From state CONV0 the FSM transitions to CONV1 in which r_w_bar is maintained logical low. From state CONV1 the FSM transition to CONV2 and maintains r_w_bar logical low. The FSM will remain in CONV2 until the counter started in CONV0 reaches the fixed value of 10. This ensures that r_w_bar has been kept logical low for 10 clock periods. This is necessary since the ADC will not begin a convert sequence unless r_w_bar is maintained logical low for at least 300 nanoseconds. Once the count reaches 10 the FSM will assume a convert has successfully started.

The FSM will then transition to WAITSTATUS0 and r_w_bar is reset to its default of logical high. The purpose of WAITSTATUS0 is to wait for a response from the ADC signaling that the A/D conversion sequence successfully began. The input status will go logical high when this occurs and the FSM will then transition to WAITSTATUS1. The FSM will remain in WAITSTATUS1 until the status signal goes logical low indicating the

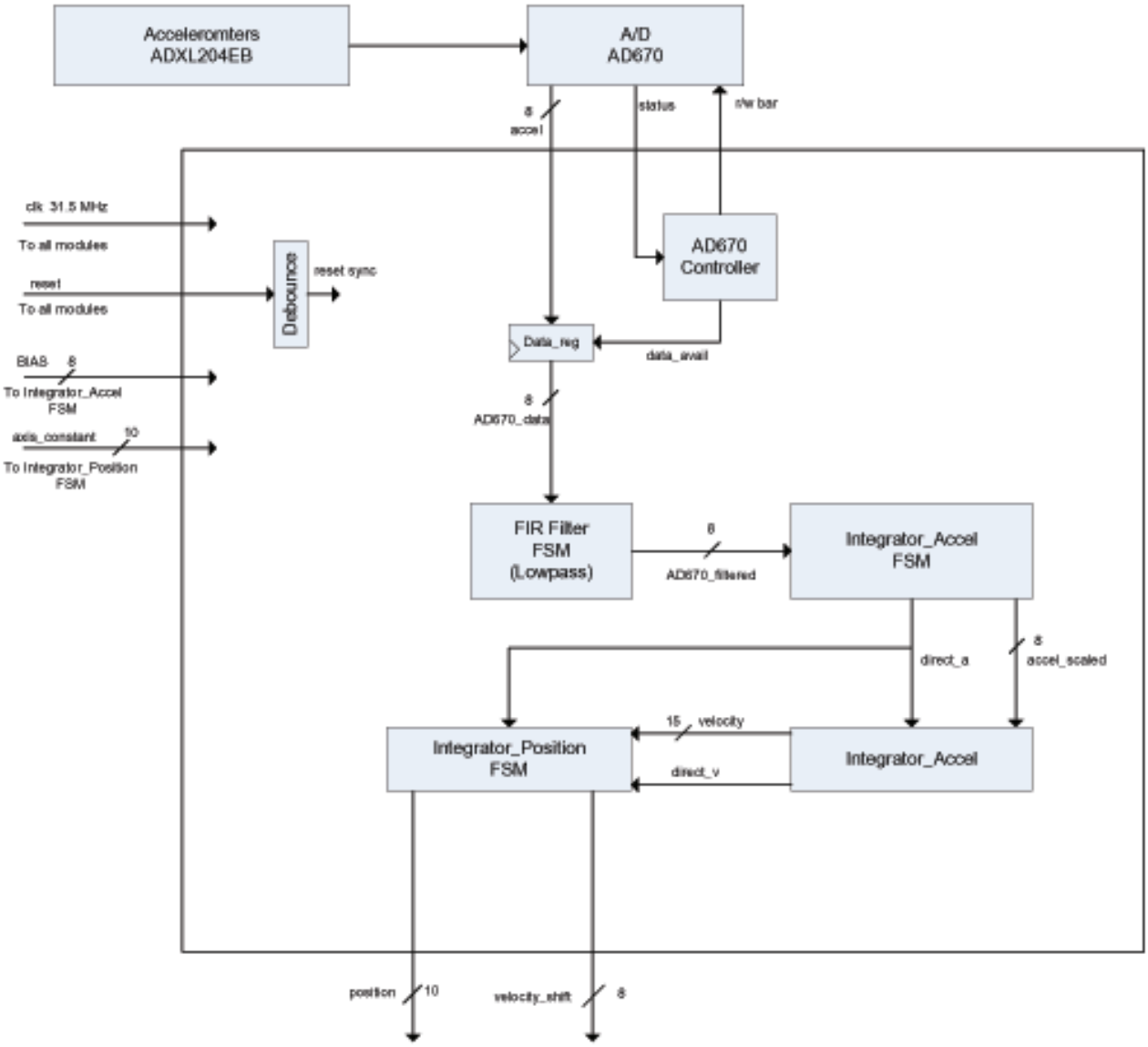

Figure 1: Accelerometer Interface

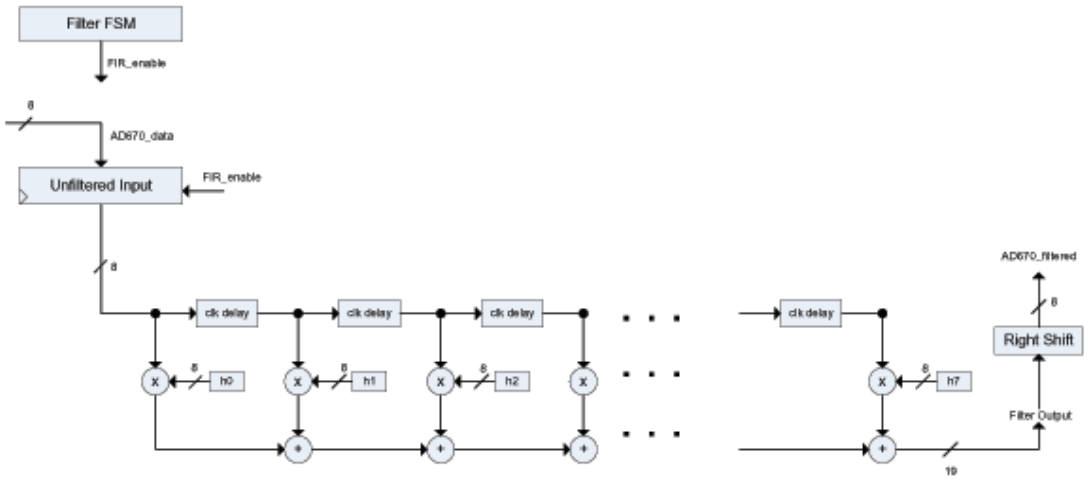

Figure 2: FIR Filter

completion of the A/D convert sequence. The FSM then transitions to READDELAY0 and starts another counter. The FSM will then transition to READDELAY1 and will remain there until the counter started in READDELAY0 reaches 10, ensuring the proper data setup time of approximately 250 nanoseconds. At this point the FSM transitions to its last state, READCYCLE, and sets dataavail to logical high. The FSM then transitions back to IDLE and waits for the FSM enable signal to begin another convert. When dataavail goes high we know the data is valid on the FSM and will remain valid until the start of another convert sequence.

2.1.2 AD670 Datareg

The purpose of this module is to prevent data glitches from occuring. A convert sequence on the ADC lasts for 10 microseconds, during this time the data is not valid. The datareg module takes as inputs the system clock, a global reset, and the dataavail signal from the AD670 Controller Module. With every clock cycle this module checks whether it receives a dataavail signal. When dataavail pulses high for one clock cycle this module registers the input from the ADC and stores it until the next dataavail pulse. In this way we prevent data glitches from occuring as a result of invalid data during A/D conversions.

2.1.3 FIR Filter FSM

The purpose of the FSM FIR filter module shown in Figure 2 is to implement a low-pass filter on the raw digitized accelerometer date. The filter algorithm operates by successive multiplication of new accelerometer data per ADC sample by the appropriate filter coefficient. When the filter receives new data, 10,000 times per second, it will begin a low-pass filter computation. In this algorithm the low-pass filter is implemented with an 8-tap 8-bit depth set of filter coefficients generated by the Filter Design Toolbox in MATLAB. The FSM contains an IDLE state, 8 computation states for the 8 taps of the filter: SHIFT0 - SHIFT7 and a last state, FINAL0 for final sum assignment. A counter is implemented in the filter to enable both the start of filter computation and timer for the FSM state

changes. The counter produces an enable signal for both purposes 10,000 times per second.

When the filter gets an enable signal it will start the computation subroutine by transitioning from an IDLE state to SHIFT0 state. In SHIFT0 the first acceleration sample is multiplied by the first filter coefficient, h0. Then the FSM transitions to SHIFT1 in which the same acceleration sample is multiplied by the next filter coefficient h1, while an updated sample from the ADC is loaded into the previous state, SHIFT0, and multiplied by its coefficient h0. This proceeds sequentially until the end of the FIR filter algorithm in state SHIFT7 and coefficient h7. The last state transition is to FINAL0 in which the total sum is computed, a 17-bit number, and then right shifted appropriately to obtain the desired 8-bit number representing the low-pass filtered acceleration.

2.1.4 Integrator Accel FSM

The Integrator Accel FSM module takes as inputs the system clock, global reset, filtered acceleration and BIAS. The module outputs an acceleration flag direct_a and a scaled acceleration. The purpose of the module is to compare the acceleration received from the accelerometer to the BIAS value. BIAS represents the zero bias acceleration when both axes of the accelerometer are perpendicular to the direction of gravity. If this module receives a larger value than the BIAS it knows that the accelerometer is being accelerated upward in the force of gravity, resulting in an acceleration larger than 1g. Conversely, if the module receives a smaller value than the BIAS it knows that the accelerometer is being accelerated downward in the force of gravity, resulting in an acceleration less than 1g. If acceleration is greater than 1g then acceleration flag direct_a is set to 1 and the scaled acceleration output is equal to the input acceleration value minus the BIAS. If acceleration is less than 1g then direct_a is set to 0 and the scaled acceleration output is equal to the BIAS minus the input acceleration.

2.1.5 Integrator Accel

The Integrator Accel module shown in Figure 3 is used to compute velocities from acceleration. This module takes as inputs the system clock, global reset, and the direct_a acceleration direct flag from the Integrator Accel FSM. A counter sets the rate of velocity computation to 100 times per second. This module sets a direct flag for direction of velocity, direct_v. The design choice for this module was to use unsigned numbers, thereby using a magnitude for velocity and the direct_v flag to determine if the velocity is up or down. If direct_a is 1 and direct_v is 1 then both acceleration and velocity are up and the subroutine will successively increment the velocity by the incoming acceleration at a rate of 100 times per second. The velocity register is limited to a value of 16383. Similarly if both direct_a and direct_v are zero then acceleration and velocity are down and the velocity register will increment until it reaches the value 16383. If direct_a is 1 and direct_v is 0 then acceleration is up and velocity is down, so the velocity register is decremented until it reaches 0 at which time direct_v is set to 1 and the velocity register begins incrementing. Similarly if direct_a is 0 and direct_v is 1 then acceleration is down while velocity is up and the velocity register will decrement until it reaches 0 at which time direct_v will be set to 0 and the velocity register will increment until 16383.

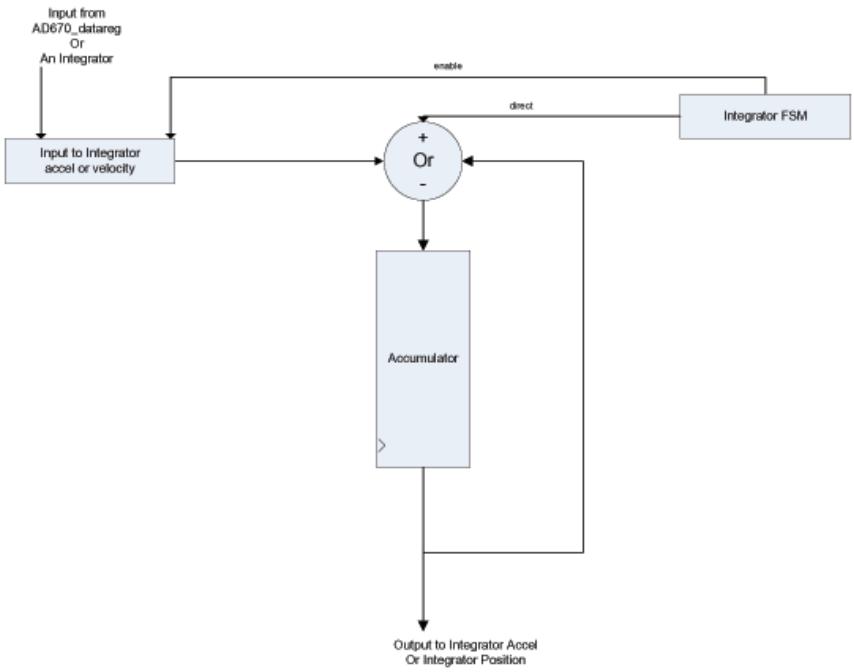

Figure 3: Generic Integrator Block

2.1.6 Integrator Position

The purpose of the Integrator Position module is to produce reasonable position data for a reticule display on a VGA display. With accelerometers mounted as per the user's preference (hands, on a marker or wand...) the Integrator Position module will calculate position data calibrated to the appropriate movement rate to allow easy interaction with the onscreen display. To this end the module takes as inputs the system clock, global and local resets, direct_a, an axis_constant, acceleration and velocity. Additionally the module makes use of a counter to limit next position computation rate to 100 times per second. The module computes a scaled velocity by right shifting the velocity register from Integrator Accel and outputs it for gesture recognition.

To compute position the module uses an axis_constant, which can be input during instantiation, that specifies the maximum position. Given our implementation of the VGA display we use axis constants of 639 for x and 469 for y. The module uses the acceleration direction direct_a and the filtered acceleration to compute position. If direct_a is 1 then position is incremented until it reaches the axis_constant at which time the position register maintains its value. If direct_a is 0 the position will decrement until the position is no longer larger than the incoming acceleration at which the position register will be set to 0. This allows for smooth motion along an axis from 0 to the axis_constant.

To lower next position computation sizes, and to produce a more realistic feel to the onscreen motion the module concatenates 0's with the first 4 MSB's of the incoming filtered acceleration.

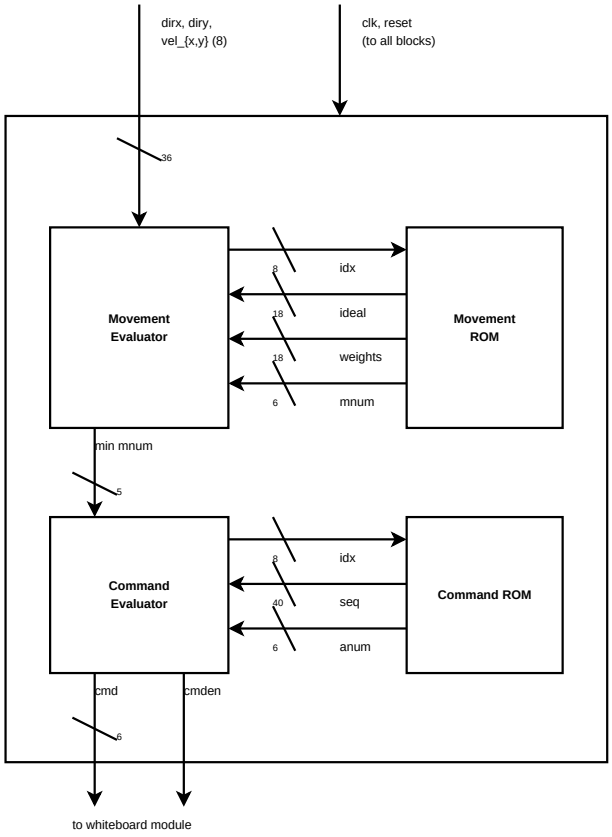

Figure 4: Planned Movement Interpreter block diagram.

2.1.7 Analog Front End

For this project we selected the Analog Devices ADXL204EB high-precision two-axis accelerometer. The acceleration resolution of this accelerometer was quite high and more than sufficient for the purposes of the project.

The accelerometer operated off a single voltage source Vss in the range of 3v to 6v; it was powered by the 3.3v onboard voltage source from the labkit. The accelerometers were adequately decoupled from the power supply with the use of a 22 uF capacitor connected between source and ground. Furthermore, we implemented analog low-pass filtering by connecting the output acceleration terminals for X and Y to ground with 0.47 uF capacitors. This effectively attenuated all signals above approximately 10Hz.

The impedance into the Vin terminal of the ADC was high enough that it required the use of LM741 op-amps wired as voltage followers for the output of the accelerometers to the input of the ADC's. Since our project made use of three accelerometers each with two channels of data, we used six AD670 ADCs and six LM741 op-amps.

2.2 Movement Interpreter - SF

We unfortunately were unable to complete this module. As a replacement, we provided command inputs via the labkit switches and buttons. A diagram of the intended structure is shown in Figure 4. The idea is that movements are compared to a list of ideal movements stored in a ROM. Each ideal movement has a weight vector associated with each component. The difference between the real and ideal movements is called the delta vector. By taking

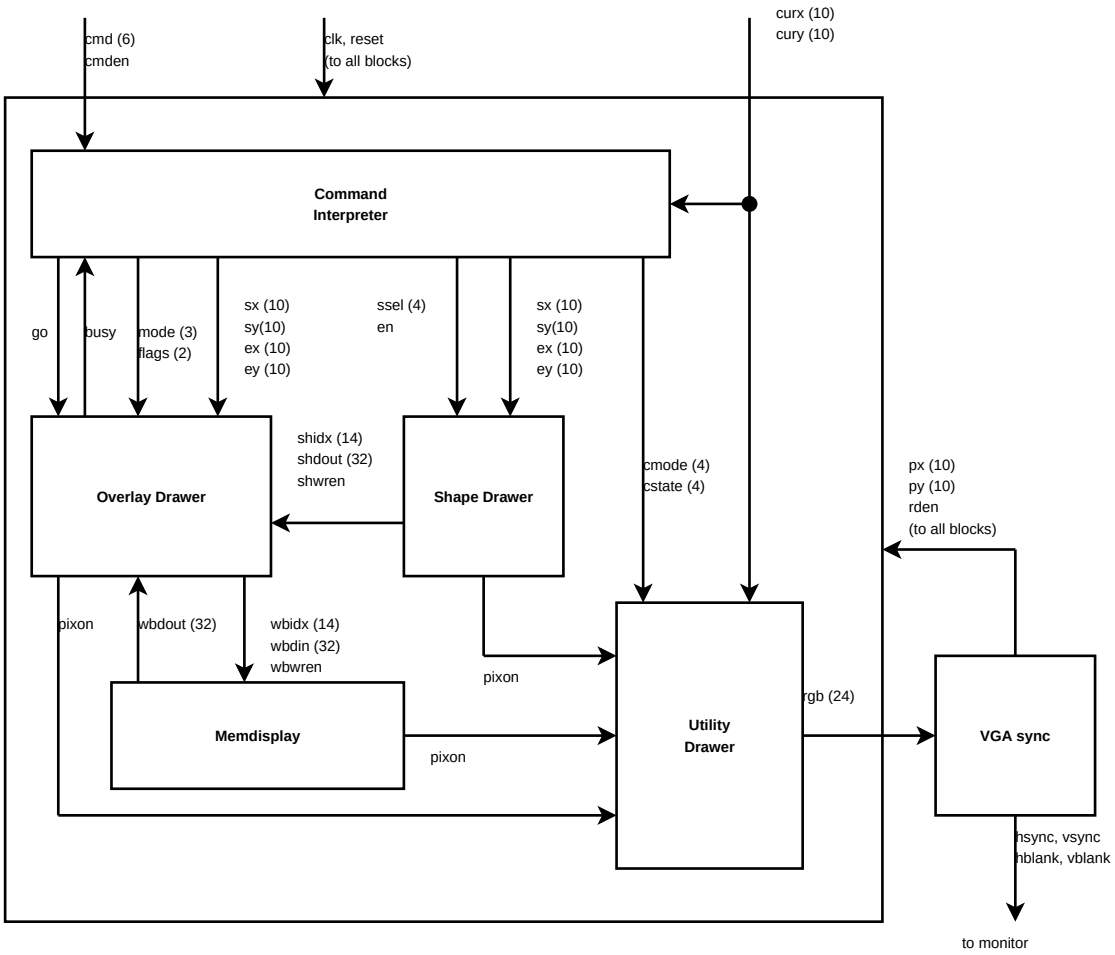

Figure 5: This shows the logical structure of the whiteboard controller. When this differs from the implementation structure, it is noted in the submodule descriptions below.

the dot product between the delta vector and the weight vector we get a scalar that increases as the movement gets further from ideal. The minimum score across all entries in the ROM is passed to the command interpreter. This keeps a queue of movements and compares the list to the list in ROM. If it matches, the queue is cleared and the command passed to the whiteboard.

2.3 Whiteboard Controller - SF

Figure 5 shows an overview of the Whiteboard Controller. There are three interfaces to the rest of the system. From the movement interpreter (though actually from the labkit buttons), *cmden* pulses when *cmd* has a user command; from the labkit, *clk* is a DCM-generated 31.5MHz clock while *reset* reinitializes the whiteboard; from the VGA controller, *px, py* and *rden* are for used for drawing a synchronization. The sole output is a *rgb*, the color for the currently drawn pixel.

2.3.1 VGA Sync

The VGA Sync module is that used and tested in Lab 4, with one extension. Many modules found it useful to coordinate actions on a frame-by-frame basis. Testing for pixel = 0 and line = 0 works, but is too late for many purposes. Rather than sprinkling tests for obscure line counts, I added a redraw-enable (rden) signal to the VGA controller output. This is high one cycle per frame, near the end of the vertical sync period, allowing modules plenty of time to prepare to operate on the (0,0) pixel.

2.3.2 Command Interpreter

The *Command Interpreter* is a major finite state machine that controls the rest of the whiteboard system. It contains all state variables that are used as inputs to other modules. Upon receiving a *cmden* signal, the interpreter checks that it is not currently busy and starts executing the command sent on *cmd*, described in Table 1. While logically a separate entity as displayed in Figure 5, it is implemented directly in the main whiteboard module.

There are two broad classes of commands: Active commands (such as drawing a shape) require user input to complete, while Immediate command (such as signalling done) do not. The former explicitly require input, so it is correct to ignore further commands until the user has finished. The latter commands occur instantaneously from a user perspective, so it is acceptable to temporarily ignore commands for those few cycles. This dichotomy is present in the implementation as well. Immediate commands merely set flags that can be tested in other states. Done, Cancel and Select signals mean different things to different commands, so implementing them as flags allows interpretation based on the context of each individual command. All other commands are Active.

Active commands coordinate other modules to begin processing. These commands could be implemented as formal minor FSMs; however, because they all interface with the overlay and shape modules, incorporating commands as a large case statement directly in the interpreter avoids the need to multiplex external inputs and outputs. Because Active commands share similar "set state–start minor FSMs–wait for input" patterns, there are very few special cases that must be separately handled.

2.3.3 Memdisplay

Whiteboard content is stored in onboard dual-port block RAM implemented via CoreGen. The Memdisplay module shown in Figure 6 encapsulates this RAM, externally exporting one port interface dedicated to normal read and write operations, and one pixel interface dedicated to outputting the currently drawn pixel. The whiteboard is black and white only, so storing a 640x470 (the bottom 10 rows are used to display a status bar) frame requires 300,800 bits. To avoid needing to read from RAM at the pixel clock rate, and to stay within CoreGen's memory depth limits, the RAM operates on a 32-bit wide bus, indexed from 0 to 9399. This results in 32 pixels per address, 20 per line.

To do this, it keeps two 32-bit buffers. When it receives rden, Memdisplay begins a read from the internal RAM port and stores it in the first buffer. When the 16th pixel is being displayed, the pixel buffer starts a read for the next index. When the 30th pixel is being drawn, the module reads the next index into the unused pixel buffer. After the 31st (last) pixel, it switches the buffer and increments the index.

No.	Name	Description
0	Null	No-Op
1	Done	Signals to complete the current command
2	Cancel	Cancels current command
3	Select	Sets Mark to begin a region selection
4	Clear	Erases the currently selected region
5	Cut	Moves the currently selected region to overlay
6	Copy	Copies the currently selected region to overlay
7	Paste	Copies content copied or cut from overlay to whiteboard
8	Erase	Erases area under cursor
9	DrawFree	Draws freehand content to whiteboard
10	DrawLine	Draws line
11	DrawRect	Draws rectangle
12	DrawFRect	Draws filled rectangle
13	DrawCirc	Draws circle
14	DrawFCirc	Draws filled circle

Table 1: Valid commands, associated numbers and brief descriptions.

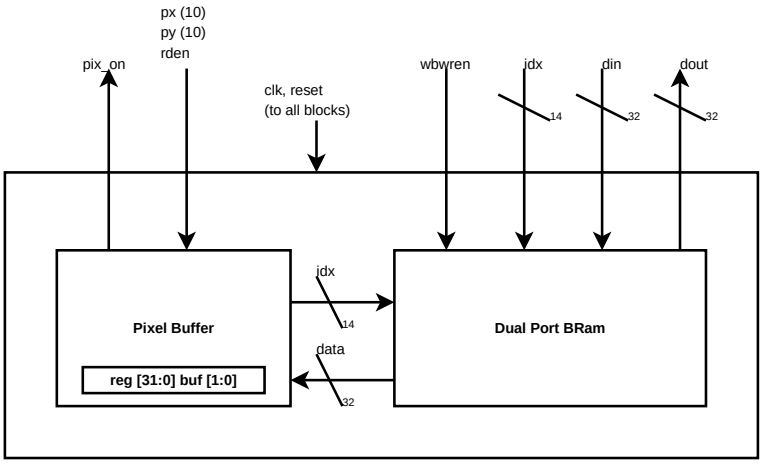

Figure 6: The Memory display buffer encapsulates a BRAM, allowing clients to access memory normally, but also outputting the contents to the VGA display.

2.3.4 Shape Drawer

The Shape Drawer uses combinational logic to compute for a selected shape whether or not the current pixel should be on or off. For some shapes, such as a filled rectangle, this was just a matter of testing bounds. Others such as lines and unfilled circles required more work. In addition to drawing the shape pixel by pixel, this module provides outputs usable by memory controllers. This is the reverse of the process used by the Memdisplay. As the VGA pixel count progresses, the current buffer is OR'ed with the pixel position mod 32. Before advancing to the next group of 32, the current buffer is written to memory. As it advances to that group, the current buffer is swapped and the index advanced. Note that the shape is not directly connected to memory, so these writes have no effect unless the overlay, described below, is in the correct mode. This prevents accidental memory corruption.

2.3.5 Overlay Drawer

The Overlay began as a temporary buffer, similar to kill-rings in emacs or clipboards in Microsoft Windows. Its function would then be to copy whiteboard data from Memdisplay to its own buffer, and vice-versa. Two minor FSMs were used for this—Ov2Wb to transfer from the overlay to the whiteboard and Wb2Ov to transfer from the whiteboard to the overlay. These are virtually identical, though Ov2Wb has some parameters that are superfluous to Wb2Ov. Both take two points specifying a region to be transferred from one buffer to the other. The extra Ov2Wb options control first, whether the entire overlay buffer should be copied regardless of the region, and second whether the whiteboard content should be merged or replaced. By adding these options, the same code can also blank the entire whiteboard and erase a region.

So, the core function of this module is to start and stop the minor FSMs, and to ensure that only the active FSM has access to the overlay and whiteboard memories. As it turns out, such functionality was also going to be required to draw shapes to the whiteboard. Thus, I added inputs for the index and data from the shape drawer. The big advantage here is that the shape drawer does not need to *read* any memory. It can write to the overlay, and the overlay takes care of merging the newly drawn shape with the existing whiteboard contents.

Because the overlay buffer shares the same memory requirements as the whiteboard buffer, it is implemented as another Memdisplay instantiation inside the overlay module. The full block diagram is in Figure 7.

2.3.6 Utility Drawer

The bottom ten pixels are dedicated to a status bar that gets drawn by the utility module, shown in Figure 8. This displays the current cursor coordinates, current command, and current command state (more useful for implementors than for users). All other modules output a pixel as "on" or "off," with the actual rgb value depending on which module is on. Because the utility module outputs different colors for the position, command and state text, it is convenient to let it select which color actually gets displayed. In reality, the color selection functionality was coded within the top level whiteboard controller. This requires fewer module connections merely for passing information to submodules.

The Charblock array handles the set of 5x7 pixel characters. Once per frame, the updater reads the position and state information, looks up the corresponding character in

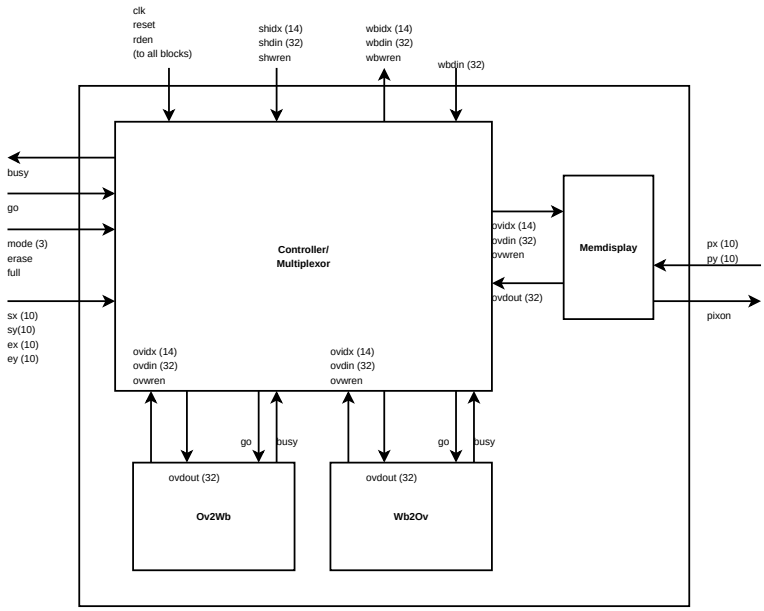

Figure 7: The overlay drawer multiplexes access to whiteboard and overlay buffers, and controls the minor FSM modules that transfer between various buffers.

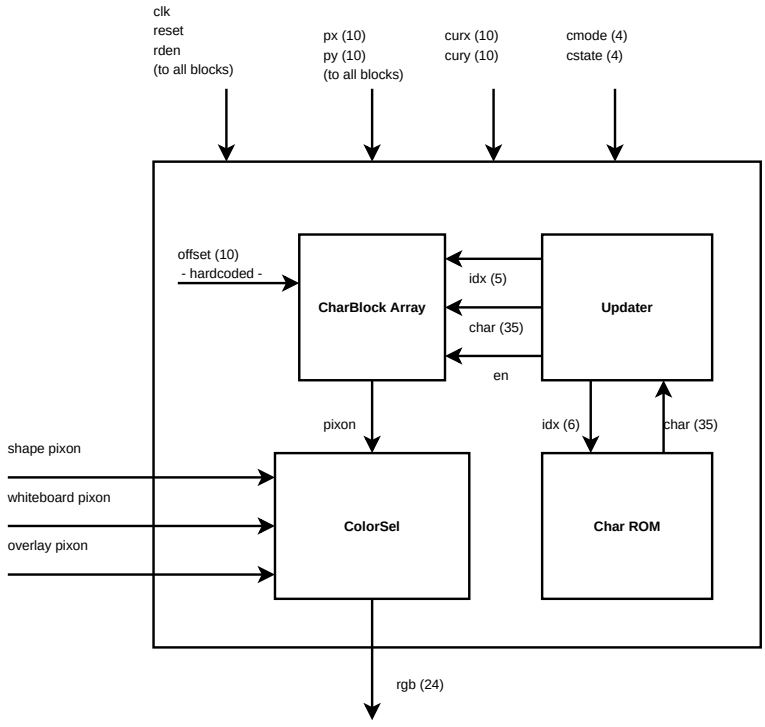

Figure 8: Module responsible for displaying status text and selecting the color associated with “on” pixels from other modules.

the ROM and signals the array to update a block specified by the index. Individual blocks in the array register the last 35-bit value and output whether or not the pixel specified by px and py should be on or not. All blocks have the same y coordinate, but each has a hardcoded x offset. By subtracting this, I could code all blocks as if they started at (0,0) and instantiate as many blocks as desired. The Char ROM coe file was generated from a custom ASCII-art font file in C.

3 Testing and Debugging

3.1 Accelerometer Interface - ND

A significant portion of the project development time was spent on testing and debugging. Initially, it took several design iterations before the AD670 controller timing successfully operated the ADC. Testbenches and ModelSim were most useful during the design phase of the AD670 controller for testing and debugging purposes. When it was apparent which state transitions were causing problems in the controller, I was able to debug it much more quickly.

In the Integrator Accel module I used both the Logic Analyzer and the combination of testbenches and ModelSim. These made it clear that there was a problem in the logic I used to compute velocities. In the initial design stages I had failed to take into account the need for a velocity direct flag in addition to the acceleration direct flag. As a result, the velocity would compute correctly incrementing from 0 to 16383, however, with an acceleration direct of 0 the register would quickly decrement until it overflowed negative and wrapped around to a value of 1023 and froze. After close inspection of the analyzer outputs it was obvious the error I had made. After I added the correct control logic to the Integrator Accel module the velocities computed as desired.

We had initially wanted the onscreen position to be computed as an integration of the velocity data. This proved problematic, and after closer scrutiny it seemed to be an ineffective way to control onscreen movement. After various difficulties in controlling the onscreen movement by position data computed from velocity we decided instead to compute position data strictly from the filtered acceleration data and acceleration direct flag. In this instance, the VGA display and position reticule were most useful for testing and debugging purposes.

3.2 Whiteboard Controller - SF

Debugging naturally began with testbenches for individual modules. I was most worried about the modules within Shape Drawer and Memdisp that aggregated pixels to 32-bit memory blocks, and vice versa. As the entire system was clocked off of the pixel clock, one cycle mismatches would be very noticable. These were somewhat annoying to test, since ModelSim needed to compute testbenches past 1-2ms to see the relevant data. A testbench for the shapedrawing memory interface is included in the Appendix.

When testing modules that needed to read or write memory, I came across chicken-and-egg problems—it's difficult to detect errors reading from zero-initialized memory, and it's difficult to test writes unless you can verify the contents. Here, the logic analyzer proved useful. By examining the memory buses, I could track down which module interfaces were getting the right data and which were not. The aforementioned testbenches showed that data was being sent. Combined with the analyzer results, this convinced me that the right

data was being written to memory.

Testing reads was easier. When memory gets displayed correctly except at columns 31, 63, 95, 127... then it's a fairly safe bet there is a fencepost error with the pixel buffer output. Debugging reads was also easier, as I had finished the status bar output and shape drawing. These completed modules made it easier to give the memory inputs to test debugging hypotheses.

4 Conclusion - ND, SF

A modular design aided implementation, and many subsystems indeed work correctly. The ADC controller and FIR filter systems produce valid valid velocity and position data available for use. The VGA interface works magnificently, and the onscreen status display accurately displays the location of the onscreen position reticule. We unfortunately ran out of time to complete the full whiteboard controller; however, the portions completed show many important components functioning correctly. The shape drawing, for instance, works by having the shape module draw to the overlay, and then merge the overlay with the whiteboard.

Still, too much faith in modularity can be bad. Integrating modules that have been written by a single author is very different than interfacing two mini-projects by different implementors. In hindsight, this integration phase should have occurred much earlier.