6.111: Introductory Digital Systems Laboratory

Group #11: Final Project Report
User-Friendly Stylus and Video Projection CAD System

Jeremy Schwartz, Paul Peeling, Faraz Ahmad

Abstract

We implemented a two-dimensional CAD system in Verilog. The pointer input was from a digital camera detecting the position of a user held stylus on a surface. The video output was to a screen which will display the CAD GUI onto the same surface, so that the stylus position corresponds with the output pointer position. We included a standard CAD toolbox in this package, incorporating features such as line and shape drawing, grouping and snapping objects to grids. All parts of the project were successfully implemented, though timing issues with the external SRAM caused the output to appear distorted at times.

Description

The motivation behind this project was to combine the power of computer drawing with the simplicity of hand drawing. The project is a two-dimensional Computer Aided Design (CAD) package modelled in Verilog and implemented on an FPGA. This package incorporates a range of standard CAD tools, such as drawing lines, polygons, regular shapes, and snapping to grids and objects. The output is standard VGA, and shows the graphical user interface (GUI) for the system. This interface will consist of a drawing area, and a fixed toolbox command set.

The innovation in this project is the user-friendly input scheme. A digital camera monitors the position of a passive stylus. This stylus is designed to be used on the video display surface. For this project, a flat-panel LCD display screen is used, but conceivably an LCD video projector projecting onto a white surface could be suitable. The reasoning behind this design is that a stylus is a more suitable input device to a drawing program than a mouse or trackball, and it is visually intuitive to be able to use a pointer on the surface on which the user is drawing. In order to make the parallel with hand drawing as clear as possible, one of our early constraints was that the stylus be `dumb'---in other words, the stylus could have no circuitry beyond a small LED.

The stylus consists of a red LED mounted onto an adapted writing implement, with a switch providing similar functionality to a mouse button. The LED will allow the stylus position to be detected by filtering the camera output, looking for a combination of hue and saturation characteristics. This position will then be mapped into a pixel position and outputted only when the switch is pressed.

The design of the project has three main components: video input, a processing unit for the drawing program, and video output, each of which will be described in separate sections:

Video Input subsystem: Jeremy Schwartz

One of the early constraints of this project was that the stylus could contain nothing more complicated than an LED. As a result, we decided to take input by positioning a camera directly above the drawing surface. This module could then ``watch'' the LED at the tip of the stylus, and pass an X and Y location to the drawing core for each frame of the camera. This part of the project was designed and implemented by Jeremy Schwartz.

From the start of the project, it became clear that any type of frame buffer memory could and should be avoided at all costs. Since the entire task was to find and report a centroid location of a bunch of points, I was determined to run this module `on-the-fly.' As a result, the system ended up being heavily pipelined. Refer to the block diagram for this module to get a better understanding of the concept. Below is a quick step-by-step description of the functionality of this module.

The video input module analyzes the information of each pixel as it comes from the video decoder. Any point that falls within the desired range has its position added to a running sum in both the X and Y directions. At the end of each frame, the module takes the sum in each direction and divids it by the total number of `passed' points, resulting in an average value for both X and Y. This average value represents the center of mass of the passed points, and it is the entire output from this module.

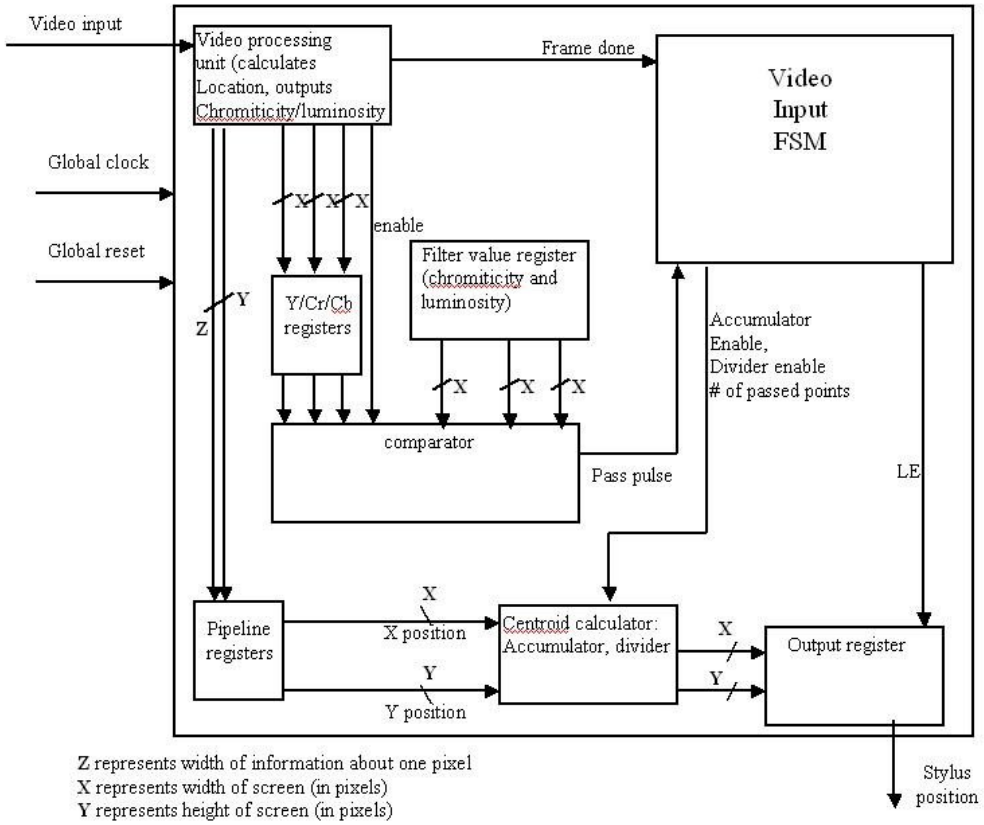
There are a few issues that are worth further explanation. The first and most obvious issue is how exactly the comparison gets done. The first key component here is the video handler FSM. This is a simple FSM that splits up the incoming decoded video signal into three separate streams: one for Y, one for Cr, and one for Cb. These streams are registered and fed (in parallel) into a comparator. Due to the implementation, the latest values of Y, Cr, and Cb are always in the registers.

The video handler FSM is also responsible for keeping track of the pixel location of each point that it processes. These values are fed into an accumulator (delayed by 4 cycles) which keeps a separate running sum of the X and Y positions of all passed points. Due to the style of the NTSC stream, the video handler only increments the pixel location when it registers a Y value. Additionally, attention must be paid to the fact that the signal is interleaved. The current system analyzes both even and odd fields to get its information.

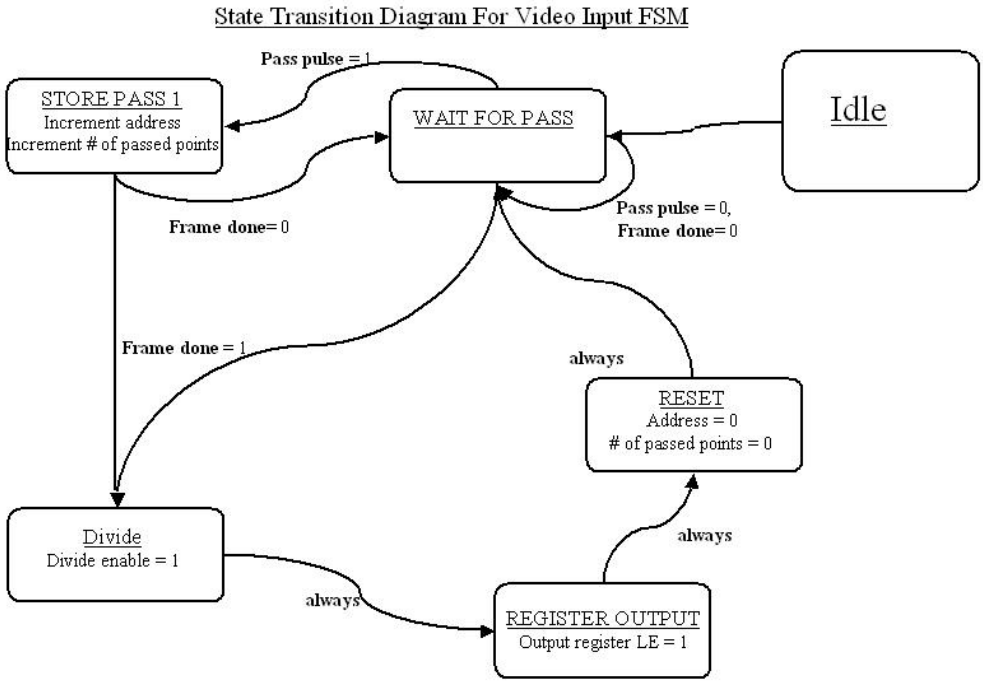
The actual comparison process is handled by the comparator component. This task is fairly straightforward: if the values of Y, Cr, and Cb fall within some set range of the desired color values, then the comparator sends out a `pass' signal. The comparator also has an enable signal, to prevent it from accidentally registering passes during any blanking period. This enable signal is controlled by the video handler.

The other FSM in the system handles the rest of the process. When it gets a pass pulse from the comparator, it enables the accumulator and increments an internal counter that keeps track of the number of passed points in that frame. When the frame-done signal is asserted by the video handler, this FSM activates the divider and registers the final output.

Video Input Block Diagram



Video Input FSM Transition Diagram



Video Output – Faraz Ahmad

The Video Output is intended to be using a standard VGA output, used mostly for computer monitors, where the ability to transmit a sharp, detailed image is essential. VGA uses separate wires to transmit the three color component signals and vertical and horizontal synchronization signals. The overall resolution produced was 640*480 @ 60Hz and this was very deliberate as a resolution any larger than this would mean an excessive amount of SRAM would be needed to store the screen data. This resolution required a pixel clock of 25MHz

The overall block diagram is shown overleaf.

Module Description & Implementation

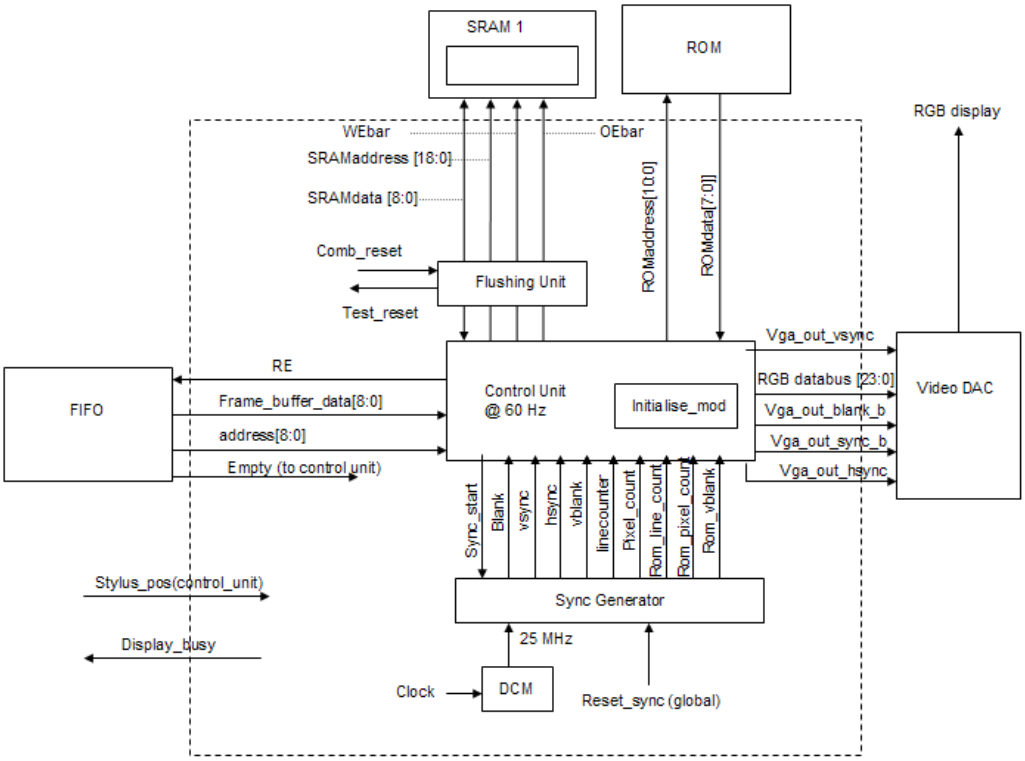
SRAM

The SRAM stores the frame data for the video. We use 19 address bits to address the 640*480 locations. The addressing is deliberately the same as the pixel co-ordinates on the screen, with the 9 representing the y position and the bottom 10bits the x position. For example, the address for the pixel in the upper hand corner would be (9'b0,10'b0) and for the bottom right hand corner (640, 480) (this must be converted into binary representation). This means we can now conveniently use the Sync Gen counters for addressing. Each address stores a 9 bit rgb value, the control unit separates it into 3 groups of 3 bits, which the control unit uses as the most significant bits for the rgb databus to the DAC.

ROM

The ROM contains the data for the toolbar. It simply contains a file of binary 1s and 0s which represents the pixels on the screen, from which the image is produced. A 1 signifies that it is the foreground and a 0 that the pixel is a background. The tool bar is 32 pixels wide and runs along the left hand edge of the screen. The ROM file is implemented as four 8 bit 'words', making up 32 pixels. We use the ROM line counter for the 9 MSB of the ROM address as these select the line on the screen file. We then use the two MSB from the rom_pixel_count to select which 'word' we are reading. The initialization module then reads each word and does the appropriate processing.

Video Output Block Diagram



FIFO

The FIFO demarcates a clear boundary between the video output and the command processing portion of the project. The pixel rgb data and address which are needed to be changed are put into the FIFO from the command side and when the video module is ready to transfer data, the pixel address and data is taken out from the video module. This structure gives a clean interface and is also very efficient. For the vast majority of the time, most of the pixels on the screen will remain the same and so it is far more efficient changing only the pixels that need to change rather than refreshing the entire screen. It also gives much more time for the video module to transfer the data during the blanking period.

Sync Generator

This module produces all the necessary counters, blanking and sync signals required for VGA output. The counters are required to count the number of pixels and the number of lines produced on the screen. There are two sets of counters; one for normal VGA output and one for ROM initialisation (rom_pixel_count, rom_line_count, these count the number of pixels in a 32 pixel wide toolbar). The sync signals and blank signals are produced when the pixel count and line count are in the appropriate range (eg the sync signals are produced after the video has passed through the active video and front porch regions). The module also produces a vblank

and rom_vblank signals; these are important as they indicate when the screen has finished with active video operation and is refreshing the screen.

Control Unit

The control unit is the key module for the VGA output, as it co-ordinates the operations of all the modules.

The control unit has two key functions; to perform the appropriate the processing of its incoming signals and to control the video.

Signal Processing: The control unit pipelines all the signals to the DAC (eg sync & blank signals) by two cycles to compensate for the DAC's pipeline delay. It also registers the SRAMaddress and SRAMdata outputs.

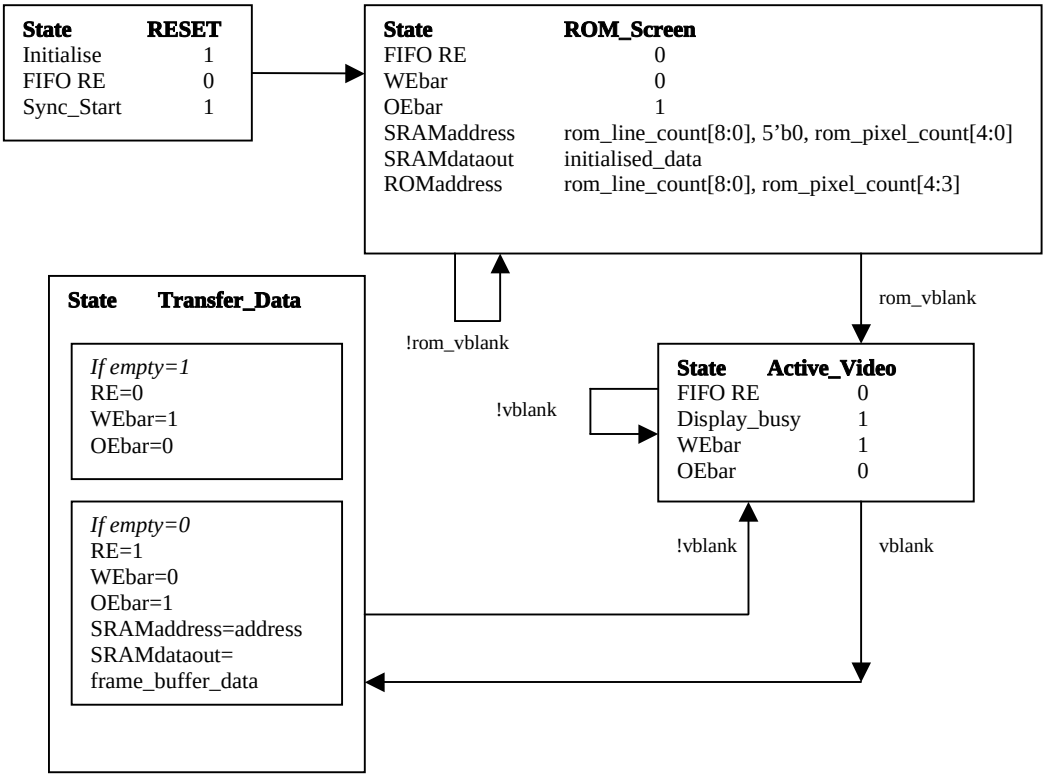
Video Control: The control unit contains an FSM that controls the video. The state transition diagram for this is shown in Fig 1.

After the initial reset mode which sets the initialise signal high, the control unit enters a ROM initialisation state. The purpose of this state is to load a static screen (in this project, the application was a toolbar). The control unit loads data from all the addresses in the ROM and output the correct rgb data into the correct address into the SRAM. Addressing is critical for this operation to be completed successfully.

A separate initialisation module is implemented in the control unit specifically for this state. It selects each bit within the 8 bit word from the ROM data, by using the two LSB from the rom_pixel_count. It then detects whether the bit is a 1 or a 0, and using the line counter, the logic decides which button the pixel corresponds to and outputs the appropriate colour which is then written into the main SRAM. The main drawback of this scheme is that the pixels on each line of the toolbar can only ever be one of two colours, as we are just setting a colour for the foreground and background respectively. However, it does avoid the use of a more complicated colour encoding scheme from the ROM data. The main advantage with this scheme, the ROM data file which is loaded is extremely simple. The addressing for this data transfer from ROM to RAM via control unit, must be precise and was the main difficulty encountered during implementation.

When the Sync Generator produced the ROM vblank signal, this indicates that an entire screen has been produced and the control unit FSM enters the active video state.

Control Unit FSM: State Transition Diagram



During the active video state, the control unit is simply reading from the SRAM locations. The stylus position and a grid is outputted to the screen to aid drawing. The appropriate counters are used for addressing the SRAM and the 9 bit rgb data values is decoded into a 24 bit rgb value to the DAC.

The final state is the transfer data state, which is entered when vblank goes high indicating that the entire screen has been done. During this 'dead time', the control unit has to now update the SRAM frame data for the next frame. The FIFO read enable is put high and the pixel address and pixel data is taken from the FIFO and copied onto the SRAM address and SRAMdatabus, where the data is then written into the SRAM.

If the FIFO's empty signal is high, this indicates that there is no data to update and so we simply sit in this state, until the vblank goes low, which indicates to the control unit that it is time to enter the active video state. In this way, the FSM now perpetually switches between active video and transfer_data states.

Flushing Unit

It is desirable to have the screen remove whatever is drawn onto the screen when the reset button is pressed. In order for this to happen, the memory needs to be 'flushed' or cleared of all its data. The way this is accomplished is to have a 'staggered' reset.

So, a button reset activates ‘comb_reset’, which initialises the memory address, data and control signals. The address is then incremented from start to finish and the data into the memory is then continuously cleared. During this time, ‘reset_sync’ is high which feeds into all the rest of the modules. This ensures that there is conflict with the SRAMs and only the flushing unit controls the SRAMs. Once the memory is cleared, we then set ‘flush_done’ high which puts reset_sync low and control over the SRAMs reverts back to the control unit. The toolbar and grid is then overlayed as normal.

Conclusions

The video portion of the project did encounter some problems. The main problem which affected the entire project was the storage of the data on the SRAMs. The SRAM is the frame map and so the data on each address represents the pixel colour on each location on the screen. Due to unforeseen timing issues on the ZBT SRAMs, the data was not being written in to the correct address. This was caused by bit errors, which meant that the video output showed pixels in the wrong location, making a shape appear ‘flaky’ (eg a circle would not appear as a clean line but rather there would be a distribution of pixels around a circular line, due to address errors). Attempts were made to rectify the problem, by changing the number of cycles to write to memory, as this would increase setup and hold times, this improved the situation somewhat but did not eliminate the problem.

CAD command subsystem: Paul Peeling

The command subsystem implements a basic 2D CAD package, taking in a stylus pointer position and 'click' button as inputs and outputting to a video display through a FIFO data structure, passing information about the address and color of a recently-changed pixel. The scope of this subsystem was intended to demonstrate how the system could be extended easily on a modular basis to implement a fully-fledged CAD package, and that Verilog modelling on a FPGA platform was a suitable method for doing this. The scope was therefore drawing objects such as lines, circles and rectangles, the ability to snap to a grid or to points on these objects, and changing the color of the object as an example property for these objects to have. Four composite commands: moving, copying, resizing and deleting objects were also implemented to illustrate the ease of extending the CAD system.

Architecture

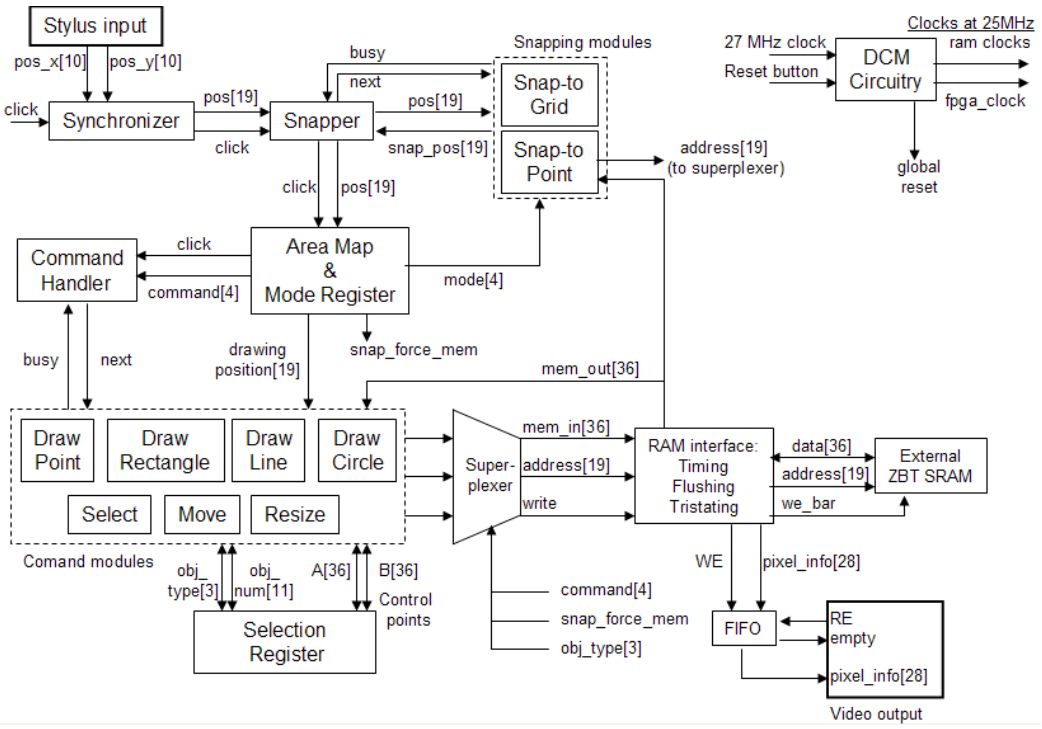
The overall architecture of the subsystem was based on a major-minor FSM approach, suggested as a method in class to provide the necessary modular abstractions. An alternative approach would be to implement a command coding scheme and then carry them out in a microprocessor manner. This would have been more efficient in terms of hardware usage, and also the loss in parallelism would not have affected this system. However within the timescale of the project it would not have been feasible to devise a system without previous laboratory experience, especially as debugging time was severely limited by the completion of the video output stage.

Design Description

The two major FSMs in the subsystem are the Command Handler and Snapper modules. These modules provide next signals and receive busy signals to the command and snapping modules, multiplexing between them based on the current command being processed. The state of each command (e.g. waiting for the end point of a line to be drawn) is stored in the minor FSM that carries out the command operations.

The area map takes in the snapped stylus position and outputs the necessary command to be handled, and the current position of the stylus on the drawing area. The snapper could have been placed before or after the area map. The selection register stores information about the currently selected object in terms of the control points and the object's type and color. The register was not implemented as a separate module, but was incorporated into the command_top system module. A large multiplexer, termed the 'superplexer', selected the required memory address, data and write signals based on the current command. Snap_to_point also required memory access, and the Select module made use of the drawing modules, hence snap_force_mem and obj_type also multiplexed the memory signals.

Block Diagram for CAD command subsystem



Interfacing

The interface with the stylus input was through a synchronizer. The click button was synchronized with the FPGA clock. As the position inputs were not always defined, the synchronizer also registered these inputs and output the previous values received if the current values were out of range.

A FIFO served as the interface between the CAD command and video output subsystems. From the command side, every time a pixel changed color, a composite value of the pixel address and a 9-bit color representation was written to the FIFO. The FIFO was chosen to have a depth of 1024, which is a reasonable amount of pixels for a single drawing command.

The most challenging interface was with the ZBT SRAM onboard the labkit. A separate skewed clock was provided to account for the clock distribution tree in the SRAMs. Correcting for pipelining delays and tristating the FPGA side of the bi-directional data bus was implemented in a custom built interface module. Much of this was implemented at the top level module as the SRAMs had to be driven directly from registered outputs, with very little scope for propagation delay. This code was provided in a modified format for the video output subsystem too, as this used of the other SRAM. Finally, both SRAMs were flushed (all address locations set to 0) before the subsystems were activated by delaying the global reset.

Algorithms & Methodology

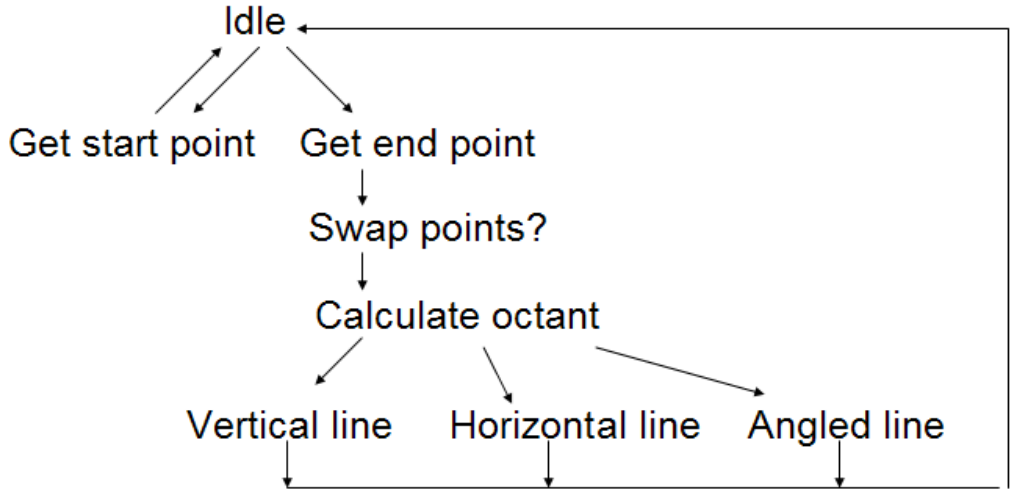
This section describes the methods used to implement the CAD package. FSM transition diagrams are not informative for many of these modules, as they are only controlling the operation of the drawing/snapping modules. There are a total of 11 FSMs in this subsystem; the line-drawing FSM has been included as an example.

- Line drawing. Sequentially takes in two points, and draws a line between them. The module first calculates the gradient of the line, and then applies the Bresenham[1] algorithm, outputting incremented pixel positions on each clock cycle.
- Circle drawing. Takes a center point and a point on the radius of the circle to be drawn. A custom square-root module based on the Megitt[2] algorithm finds the radius of the circle, then an application of the Bresenham[1] algorithm outputs 8 pixel positions in each octant of the 2D plane for each step.
- Rectangle drawing. A basic algorithm of my own devising, swapping the input points if necessary to minimize the number of states used. Point drawing was also implemented.
- Snap-to-grid. Takes in the input position and calculates the nearest grid position to it in terms of the x and y coordinates.
- Snap-to-point. Takes in the input position. Uses a 177-point search pattern (circular, with a depth of 8 pixels from the input position) from an internal ROM to query each address of the SRAM for object information. Pipelining is taken into account, therefore the SRAM can be searched every clock cycle. The search pattern is easily extendable to larger search areas and potentially different search types should this feature ever be required.
- Selection. Takes an input position and queries the SRAM for the object information at that point. If there is nothing there, nothing happens. Otherwise, the module finds the control points (such as the start and end points for a line) by searching the object map in the SRAM, which is ordered by object number (an unique identifier assigned to each object upon drawing it). It then redraws the object in the selection color.
  Each drawing module changes its behavior if the select mode is on, loading in the control points immediately rather than waiting for the user to input them. Prior to selecting an object, select deselects the currently selected object by redrawing it in its original color.

The four composite commands use a major-minor FSM structure controlling selection and redrawing:

- Delete: Selects the object and zeros all the address locations at those points.
- Move: Puts one of the control points at the new position, offsets the other control point, deletes the previous object and draws the new moved object.
- Resize: As for move, but keeps one control point stationary and moves the other.
- Copy: As for move, but does not delete the previous object.

Line Drawing FSM transition diagram



Testing, Debugging & Simulation

Testing and debugging was problematic as there was no video output available until the final stages of the project. Simulation was carried out on ModelSim, and the results compared with identical code on the FPGA. For this to be possible, I programmed a number of 'test bench' modules simulating the camera input in terms of position and clicks, and used this when running on the labkit.

Algorithms were tested on ModelSim to observe the outputted pixel positions and judging their correctness. The functionality of the subsystem was increased from milestone to milestone, and a rigid command passing structure implemented to ensure that different combinations of user inputs would not cause problems.

The majority of debugging time was spent on SRAM timing. The major conceptual problem was the skewed clock correction, and therefore calculating pipe line delays from the FPGA to the SRAM. However even when this was accounted for, and outputs correctly tristated, the SRAM behaved unreliably. This was very visible on the video output as clearly bit errors were causing strange circle and line drawing, but also made selection imprecise and sometimes a matter of luck. This was manifested in the selector wrongly selecting an object, or misreading the control point information.

Unfortunately there was very little information to help correct these issues. The SRAMs were so sensitive to timing, that despite timing constraints being put in the design (and also passing), the performance of the system was affected by whichever outputs were being used by the logic analyzer at that particular time, and also completely unrelated blocks of code.

Interfacing issues were largely code-independent, as a significant amount of design work had gone into ensuring the same addressing schemes and predictable, hardware modules (the FIFO and synchronizer).

Extensions

Clearly the system implemented here is a long call from commercial CAD packages, but the idea of the entire project was to serve as proof-of-principle for the user input technique and the hardware CAD implementation. Generally this has been achieved by the command subsystem, as selection and alteration of object properties were functional. There are a number of deficiencies in the current system at this point however, which should be solved before further extension:

- As previously mentioned, the major-minor FSM structure is not efficient in terms of hardware usage for this application, and a microprocessor could be more suitable.
- The FIFO interface on the command size did not take into account whether the FIFO was full or not. For the current system, we were not drawing large numbers of pixels at a time, but if rendering techniques were required, the FIFO would quickly overflow. A solution to this would be a refresh command which periodically cycles through the object map in the memoryand updates the FIFO.
- The system does not cope with intersections between objects. A viable system for doing this was described in the project presentation, but to have implemented it would have introduced a large amount of code which would not be easily testable until a video output was available. One consequence of this is that a new object overwrites an older object at the intersection point, and therefore if this new object was deleted, there would be a gap in the older object.

Conclusion

Due to the limited useful debugging time, this project taught me how to write a large quantity of easily verifiable, robust and transparent code. A significant amount of design work was spent in preparing the block diagram, and the final block diagram differs considerably from the first concept. The command structure was kept as general as possible, and so major code changes were not necessary even as additional requirements such as the drawing modules had to either load user input points or selected control points.

A useful skill learned was to be able to quickly and easily transfer algorithms from computer languages (C++ in the reference cases) to Verilog. FSMs are very amenable to this task, as the structure naturally allows looping and counting. The main requirement is to keep track of variables and when they change values, as assignment in Verilog is a very different concept to that in C++ for instance.

As a team, we learned the importance of designing interfaces between subsystems, using each other for checking code and source of ideas and information, and negotiating scarce resources such as a single labkit between the three of us.

References

1: Breshenham's Algorithm
http://www.funducode.com/freec/graphics/graphics2.htm
2: Self-generating clock using an augmented distribution network
Gerald M Blair, University of Edinburgh
3: Analog Devices – ADV7125 DAC data sheet

Acknowledgements

Charlie Kehoe for support, advice and encouragement
Keith Kowal for advice on video, system and interface design
Chris Forker for helping to identify the SRAM problems
Nathan Ickes for his labkit expertise
Jenny Hyunjoo for much-needed doughnuts and coffee