Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science
6.111 – Introductory Systems Laboratory (Spring 2004)

SmartKit
A Digital Near Miss Learner

Abstract

This report describes the design and implementation of a digital near miss learner. Divided into three stages our system is comprised of an input stage, processing stage, and output stage. The user inputs a shape with a laser pointer. This image is captured with an NTSC camera and saved to RAM. The coordinates of the shapes endpoints are extracted and used in order to determine the input shape's resemblance to 4 "learned" shapes. A score is calculated and used to pick which definition shape the input shape has been determined to be. This definition shape is then displayed on a VGA monitor along with the user input shape.

Orhan Dagli
Jo-Al Lopez
Sedrick Tydus
May 13, 2004
TA: Cemal Akcaba

Table of Contents

LIST OF FIGURES **2**

SYSTEM OVERVIEW **3**

INPUT STAGE **3**

- **DECODING NTSC CAMERA DATA** 4
- **WRITING TO RAM** 5
- **FINDING LASER POINTER DRAWING COORDINATES** 7
- **TESTING** 7

PROCESSING STAGE (AI MODULE) **8**

- **FLOATING POINT ARITHMETIC LOGIC UNIT (ALU)** 8
- **ALGORITHM** 9
  - ***Training*** *9*
  - ***Recognition*** *10*
- **IMPLEMENTATION** 11
- **TESTING** 12

DISPLAY STAGE **12**

- **SYNC_GEN** 13
- **DISP_VGA** 13
- **SHAPE_DISP** 14
- **TESTING** 14

CONCLUSION **15**

APPENDIX **16**

List of Figures & Tables

Figure 1: *Block Diagram of Near Miss Learner----------3*

Figure I1: *State Transition Diagram for DECODER_WRITE_TO_RAM Module----------5*

Figure I2*: State Transition Diagram for VIDEO_DECODER Module----------6*

Figure AI1*: Floating Point Arithmetic Unit Block Diagram----------9*

Figure AI2*: The "four values"----------10*

Figure AI3*: AI Module Block Diagram----------11*

FigureD1*: Block Diagram of Display Module----------13*

Figure D2*: State Transition Diagram of shape_disp FSM----------14*

System Overview

Near miss learning is an Artificial Intelligence concept first introduced by Professor Patrick H. Winston in his PhD Thesis at MIT (See FigureNM1). Whereas most other machine learning methods need a batch of training data, a near miss learner can learn from every new example. It can take both positive and negative examples and improve the definition by extending or restricting it, respectively.

Sketch training/recognition is a field where the near miss learning idea is implemented in real life. Several examples of a shape are used to train a system so that it can recognize the shape later on. Character recognition on PDAs is a common example of sketch recognition. Sketch training/recognition is different from a complete near miss learner in that it makes use of positive examples, but not negative examples. Since the aim of sketch training is to extract the average characteristics of a shape, negative examples are unnecessary.

The aim of our final project is to build a digital system capable of sketch training and recognition. We have done this by using a modular design approach and dividing up the system into three stages. The user inputs a drawing with a laser pointer against a dark background. As the user is drawing the shape a camera captures the image. The image from the camera is analyzed and processed in a way that results in the complete shape being stored in RAM. With an image stored, the processing stage is able to use an AI algorithm to compare the shape to shapes it has already learned. Concurrently, the third stage, displays the drawn image on a VGA screen for the user to see. When the AI module has completed and identified the input shape the display will output the correct shape that it had previously learned.

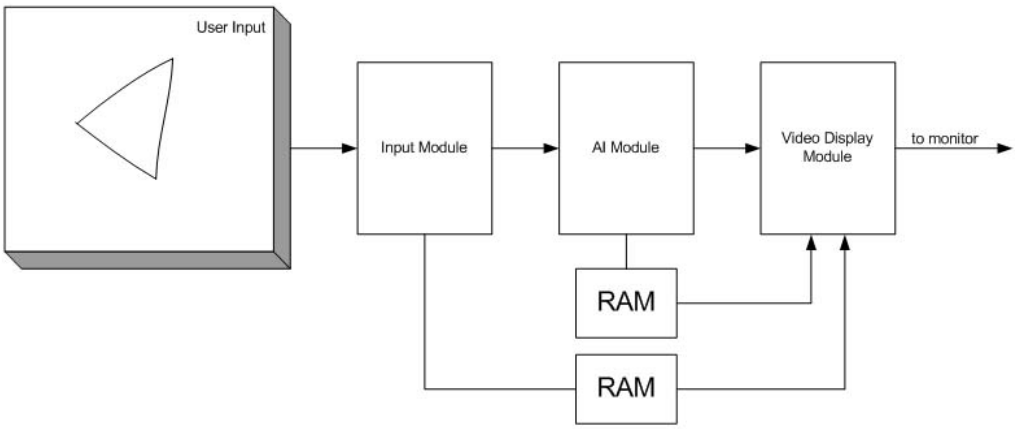

Figure 1: Block Diagram of Near-Miss Learner

Input Stage – Jo-Al Lopez

The overall input to the digital system is an analog video signal that is produced by a user using a bright red laser on a black background. This analog signal is captured by an NTSC (National Television System Committee) camera and is then passed on to our actual digital system as the input signal. The input stage must digitize this analog video signal, recognize when the laser pointer is on, and store the image that the user draws with the laser pointer. This data is stored in a RAM (random access memory) for the output stage to read and display using a VGA monitor. In addition to capturing the user-made drawing, the input system must also figure out each of the coordinates of the endpoints of each stroke that the user makes with the laser pointer. These coordinates are then passed on to the processing/AI stage of the digital system.

-Decoding NTSC Camera Data-

An input to the input stage of our system is an image created by a user using a laser pointer. Since the data from the NTSC camera is analog data, it must be digitized in order for it to be useful. This is done by the ADV7185 video decoder chip. The ADV7185 decodes the analog signal that makes up an image and represents it with digital data of luminance and chrominance values. A digital image is essentially composed of pixels, lines, fields, and frames. All this data is embedded and can be decoded from the digital video signal that is produced by the ADV7185 chip. The first task when designing the input stage was to decode the digital data from the ADV7185 chip and determine when the beginning of a line, field, active data, and blanking data occurred. This was done using the VIDEO_DECODER module.

In order to determine when a line, field, active data, and blanking occur, the VIDEO_DECODER module takes as input the digital data from the ADV7185 chip. There are four "regions" of a stream of digital data that are very important. These regions include: the odd field, the even field, active data, and blanking data. The only useful part of digital video is the active parts of the odd and even fields, all the rest is blanking data and does not contain any useful information regarding the image. In order to detect when active data occurs, the VIDEO_DECODER module looks for certain flags, within the stream of digital video coming from the ADV7185. These "flags" are strategically placed within the stream of digital data and are called TRS (timing recognition symbols). They are strategically placed because they occur at times that active digital video is not being decoded.

The TRS for digital data from the ADV7185 is the specific sequence of 8'hFF, 8'h00, 8'h00, 8'hXY. The VIDEO_DECODER module's job is to examine the stream of digital data and extract when active video occurs. Each data byte from the ADV7185 represents a luminance and chrominance value for a pixel, therefore two bytes represents a full pixel of data. When examining the stream of digital data (the stream of bytes), the VIDEO_DECODER module looks for the specific sequence of 8'hFF, 8'h00, 8'h00. Once it has found this sequence, it knows that the either the end of active video or the start of active video has been discovered. A normal stream of digital data includes active data and a period of blanking (non-active) data. Since the TRS sequence within the digital data represents either the start or end of active data, the VIDEO_DECODER module must also discern between the two. For all intensive purposes, the beginning of a line of video may be denoted by the TRS sequence that corresponds to an end of active video, and the start of actual active video may be denoted by the TRS sequence that corresponds to the start of active video.

The VIDEO_DECODER module checks first for the TRS sequence 8'hFF, 8'h00, 8'h00. If the XY word (2 byte word following the valid TRS sequence) corresponds to a valid word that should follow an end of active video, it asserts the signal *EAV*. Once the VIDEO_DECODER module finds the end of active video, it begins to look for the same TRS 8'hFF, 8'h00, 8'h00. If it has found this sequence it asserts the signal *SAV* since once that specific sequence has been found once, we can infer that a line of video has begun, and once it has found it a second time, we can infer that the active part of that same line has begun.

In addition to checking for the end and beginning of active video, the VIDEO_DECODER module also outputs the signals *SOF* and *SEF*. These signals denote only when the active part of the odd and even fields begin. As stated before, an image is represented by two fields, which compose a frame of video. These two fields are called the odd and even fields and are interlaced when displayed on a monitor. The VIDEO_DECODER module knows when the beginning of the two different fields begins based on if the signal in the active region of data, as well as based upon the XY word after a TRS sequence

A final task that VIDEO_DECODER performs is conversion of the digital data from the ADV7185 into binary digital data. Since each pixel is represented by two bytes, only the

second byte is useful since it represents the luminance value of the signal. We only care about luminance because our system is based on black and white. If the data byte that corresponds to luminance is above a certain threshold, the *data_output* of the module will be one (white), otherwise it will be zero (black). The module knows when the byte is luminance or chrominance based on a counter that counts up to 3 (4 total values including zero). When the count is one or three, then a luminance value is being represented and we can convert it to either a one or zero.

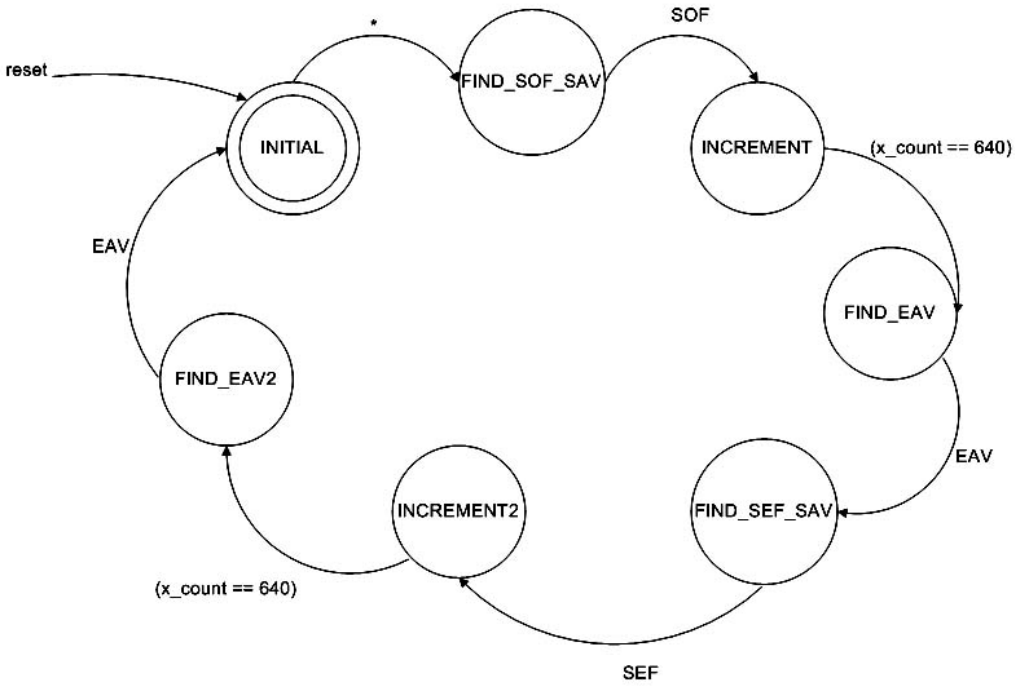

Figure I1: State Transition Diagram for DECODER_WRITE_TO_RAM Module

-Writing to RAM-

A second task of the input stage is to store the image that the user draws with the laser pointer into a RAM. This is done by the module DECODER_WRITE_TO_RAM. This module implants two counters, one for x and one for y. These counters are incremented and dependent upon the outputs from the VIDEO_DECODER module. Since it takes the inputs *SOF, SEF, SAV, EAV,* and *data_out*, it checks where exactly in the stream of video the signal is at. When it is in the active portion of an odd field, which means the laser pointer signal is being represented by the active portion of the digitized stream of video, the x counter is incremented every two clock cycles, (two clock cycles of 27MHz since a word occurs ever clock cycle, there for a full pixel occurs every two clock cycles) representing where on a line the pixel is at. When this module detects the input *EAV*, then the y_counter is incremented and the x_counter is set to zero. This represents the beginning of a new line of active video, and therefore denotes what line new data is on. In both situations, and on every new pixel, only if the data is digital 1 (white), are the signals to write, as well as the data_out and address are set. The address is set to be whatever the x and y counts were, therefore from the address; the output module can know where the white pixel should be on the screen. These three parameters are sent on to a Xilinx COREGEN module that is a RAM. When the RAM gets the write_enable signal, it writes the data into the memory at the specified address, which is a concatenation of the y_counter and x_counter for a total of 18 bits of address.

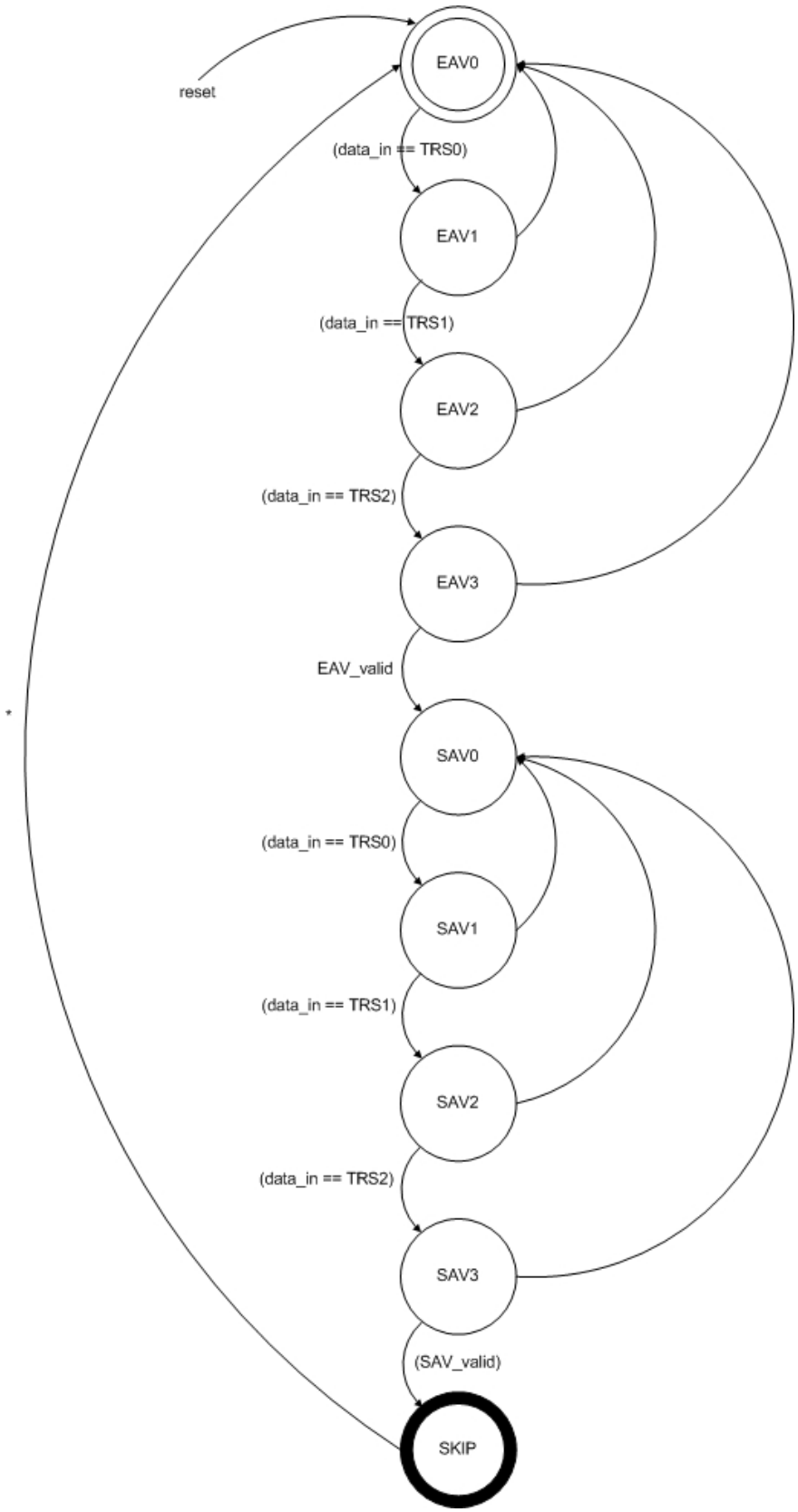

Figure I2: State Transition Diagram for VIDEO_DECODER Module

-Finding Laser Pointer Drawing Coordinates-

Very similar to the module which wrote to RAM, the module EXTRACT_COORDINATES extracts the coordinates of the endpoints of each stroke the user makes with the laser pointer. In order to do this, an input signal that is asserted by the user by pushing a button is taken. When this button is asserted, the NTSC video camera's data is captured for only a single frame. The purpose of this is to only take a snapshot of where the laser pointer is currently. Since it is basically a snapshot of where the laser pointer is at the beginning of a user's stroke, the coordinates of the brightest pixel in the captured frame is computed. The coordinates of the brightest pixel in the frame is computed by concatenating the x and y counters, which have been incremented in the same way as in the DECODER_WRITE_TO_RAM module and only using the specific concatenation corresponding to the brightest pixel. The endpoint of a stroke is computed in the exact same way as the starting point since all this module does is take a "picture" of where the laser pointer currently is located and takes the location of the brightest pixel. This module is very broad and could be used for any number of strokes, only computing the end points of a stroke when the user pushes a button.

Unfortunately, this module works only in principle and its problems could not fully be worked out for full implementation of the system. The counting scheme of the x and y counters was faulty, and hence, the correct coordinates of the laser pointer could not be reliably computed and sent to the AI/processing stage of the system (similarly, since the DECODER_WRITE_TO_RAM module's counting scheme was similar to the EXTRACT_COORDINATES module, the DECODER_WRITE_TO_RAM did not write to the correct address and could not be read by the output stage for displaying).

-Testing-

As stated previously, the input stage did not fully work. The major foundation of the input stage was fully implemented. This foundation was the VIDEO_DECODER module, which outputted all the major signals that denoted exactly the beginning of each field as well as the start of a line of video, and the active part of that line. Past this foundation, the modules used to keep track of the address of an active pixel, did not work. Though it appeared that these modules simulated fine in the Xilinx tool, when actually tested on the kit, they did not behave as desired. When testing the VIDEO_DECODER module, it was very easy to see that the output signals occurred at the proper locations when compared to the TRS occurrences. Once I knew the outputs of the module were functional, I added the section where the data_in from the ADV7185 was converted to either a one or a zero. To test this section, I compared the output of the ADV7185 chip against what the output of my module was. To ensure functionality, the output of my module had to be '1' if the data from the ADV7185 was above 8'd80, corresponding to an IRE level of above 80 (IRE of white is 100 and IRE of black is around 40, much lower than the 80 threshold I chose). If the data from the ADV7185 was lower than this threshold, then the data out of my module has to be logic '0.' At the end of the above tests, this module proved to be functional.

The second two modules in the input stage were not functional at the time of this report. When tested using the logic analyzer, it was apparent that the counter that corresponded to the x-coordinate was not counting correctly and hence was not portraying the true location of the laser-pointer. This x-value kept jumping and did not stay stable enough to get a true reading. Some problems were also noted concerning the TRS of the incoming digitized data. At instances that the TRS sequence should have been valid, and in fact was valid, the logic analyzer would display a value that was a bit or two (out of the eight bits) that was incorrect. This could have thrown off the FSM's in the modules since they were looking for a specific sequence in order to work. One explanation of this "bit flipping" could have been some noise within the kit that was corrupting the digital output of the ADV7185, and hence feeding the incorrect input to the input stage's modules. At the time of submission, this bit

flipping problem could not be explained nor corrected. With an incorrect input to the input modules, it can easily be seen that the count of the x and y would be thrown off and be incorrect, making it non-functional. Future considerations of this problem could be investigating what could cause significant enough noise to disrupt the output of the ADV7185 and correct this problem. With this problem corrected, the counters could work, as they did in simulation, and the input stage could be functional, as well as the entire system.

Processing Stage (AI Module) – Orhan Dagli

The AI module is responsible for the processing of input shapes, and accomplishing training and recognition of these shapes. The user communicates directly with the AI module (i.e. through switches) to initiate the system. The AI module then asks for shape coordinates from the input stage, and starts its computations. When it is done, it conveys its findings to the output stage for display.

-Floating Point Arithmetic Logic Unit (ALU)-

The training and recognition tasks require a lot of arithmetic computations that need to be done at high precision. These computations include division and square rooting. Therefore a Floating Point Arithmetic Logic Unit needs to be implemented as the core of the AI module.

The FPALU built for the project is capable of performing five operations on operands represented in the IEEE single precision floating point standard: addition, subtraction, multiplication and square rooting. In this standard, numbers are represented in 32 bits - the first bit is reserved for the sign, the next eight bits represent the exponent and the last twenty three bits are the mantissa:

```
S    EEEEEEEE    MMMMMMMMMMMMMMMMMMMMMMM
31  30          23  22                    0
```

The value (V) of the number represented is determined as follows:

* If E=255 and M is nonzero, then V=NaN ("Not a number")
* If E=255 and M is zero and S is 1, then V=-Infinity
* If E=255 and M is zero and S is 0, then V=Infinity
* If 0<E<255 then V=(-1)**S * 2 ** (E-127) * (1.M) where "1.M" is intended to represent the binary number created by prefixing M with an implicit leading 1 and a binary point.
* If E=0 and M is nonzero, then V=(-1)**S * 2 ** (-126) * (0.M) These are "unnormalized" values.
* If E=0 and M is zero and S is 1, then V=-0
* If E=0 and M is zero and S is 0, then V=0

Therefore, the number 10 is represented in this format as follows: 0 10000010 01000000000000000000000 = 4120 0000 (in hex)

Verilog supports integer add (+), subtract (-) and multiply (*) functions. For floating point addition, subtraction and division, these integer operations are used to calculate the new exponent and mantissa. For example, addition is carried out in the following steps:

1- Take the difference between the exponents of the operands.
2- Shift the mantissa of the smaller number to the right as many bits as this difference, and change its exponent with the larger number's exponent.
3- Add (+) the mantissas and normalize the result.

Verilog also has the integer division (/) function, but neither the Altera FPGA nor the Xilinx FPGA supports this function. Therefore long division is used for floating point division. The divisor's mantissa is shifted right by one bit at every clock period and compared to the dividend's mantissa. It takes a fixed twenty three clock cycles (the mantissa's length) to perform floating point division.

Floating point square rooting, on the contrary, cannot be implemented by itself; it needs to use the other four operations. Newton's method can be used to estimate that the equation (x^2 - A = 0) has the root:

x_1 = (x_0^2 + A) / (2 * x_0)

If A is used as the initial estimate and the computed x_1 is recursively substituted for x_0 in the equation, the result converges to the square root of A. 32 recursions are used to get a highly precise solution.

The five floating point operations are each implemented using one Finite State Machine. These FSMs are instantiated in a top level FSM which takes in the operands together with a selector signal 'alufn'. This signal selects the operation to be performed on the operands. See FigureAI1.

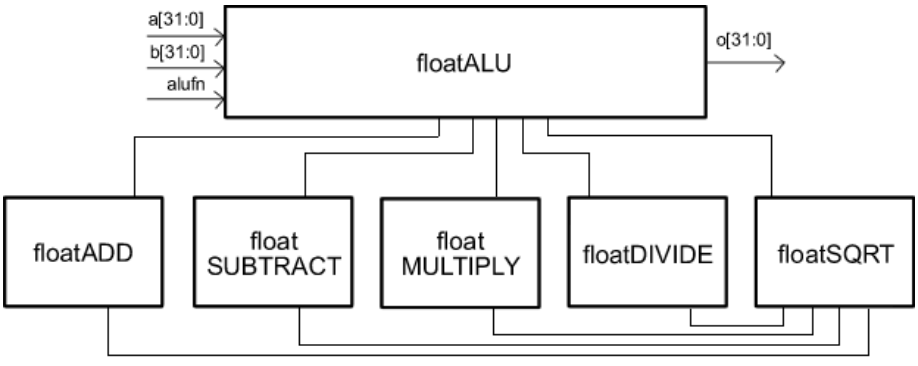

Figure AI1. Floating Point Arithmetic Logic Unit Block Diagram

-Algorithm-

Training

Concisely, training involves using several examples of a shape to create a definition of the shape. The challenge is to find an algorithm that extracts from the examples the properties that are particular to only that shape. The stroke coordinates coming from the input stage cannot simply be averaged and stored as a definition. The system should be able to define all versions of a shape (i.e. scaled, rotated, translated, etc) in a single definition. Therefore, the algorithm should make use of ratio relations between the strokes.

The algorithm used in the project is extremely reliable. The definitions it creates are not mathematical proofs of the shapes, but the algorithm can produce definitions that will:
- include all examples of the shape defined.
- exclude all kinds of other shapes.

As suggested before, the algorithm compares every stroke with each other and

computes ratio relations. The high-level steps taken are as follows:
- Take the stroke coordinates of a single example.
- Compute the length of each stroke and the angle it makes with the horizontal.
- Compute the "four values" (discussed later) between each stroke pair.
- Carry out the first three steps for all the examples in the training set.
- Compute the average and standard deviation of the four values between each stroke pair of each training example.
- Store these average and standard deviation values as the definition of the shape.

The "four values" are the heart of the algorithm. They perfectly define the relative position of two lines in the x-y axis. Therefore, all the "four values" between each pair of stroke in the shape will perfectly define the relative positions of all the lines in the shape, thereby defining the shape. These "four values" are the following:

1- The length ratio between the two lines.
2- The angle between the two lines.
The other two values depend on a third imaginary line that is drawn between the starting points of the two lines.
3- The length ratio between the first line and the third imaginary line.
4- The angle between the first line and the third imaginary line.

The following figure (Figure AI2) shows how these four values define the relative position of the two lines in a cross. If you start off with one of the lines and no information, you don't know where the other line lies. If you are given the values # 2, 3, 4 and 1 respectively, you can gradually limit the possible locations of the second line down to its exact place.

Figure AI2. The "four values"

Recognition

Recognition is done by comparing a new drawing with each of the existing definitions, and calculating an error value for each comparison. The smallest error gives the definition that the new drawing is most similar to.

Similar to training, the "four values" of the new drawing are computed during recognition. The difference of each of these values with their corresponding averages in a definition are taken. If this difference is less than the corresponding standard deviation in the definition, the error is not incremented. If bigger, the difference is squared and added to the error value, so that the error grows exponentially. This score calculation is done for every definition stored.

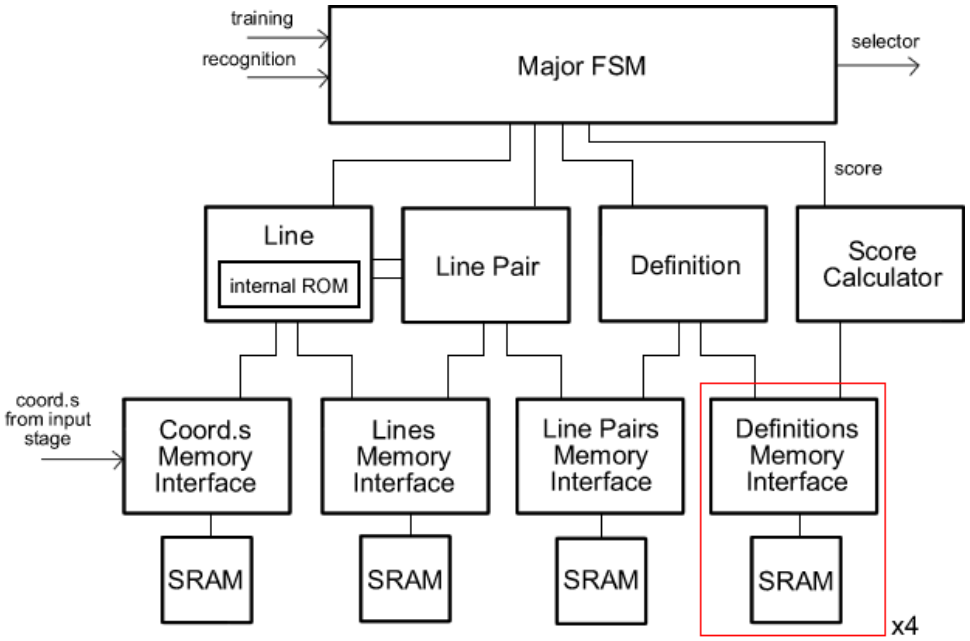

Figure AI3. AI module Block Diagram

-Implementation-

The algorithm described is implemented in hardware using nine Finite State Machines, four SRAMs, one ROM and the Floating Point Arithmetic Logic Unit (See FigureAI3). There are three levels of hierarchy throughout the module. The Major FSM is at the top level. The "Line", "LinePair", "Definition" and "ScoreCalculator" FSMs are at the intermediary level. The SRAM interfaces, each of which interface with one of the SRAMs, are at the bottom of the hierarchy.

The system works like a factory. It starts with the raw data (stroke coordinates) and works on it to produce the final product: the definition (in training) or the score (in recognition).

The four SRAMs are the processing line. They store information at different levels. The first stores the coordinates of the stroke end-points of one drawing. It basically accumulates the data coming from the input stage. The second SRAM stores the lengths and angles of the strokes in one drawing. The third stores the "four values" of all stroke pairs in all the drawings in the training set. There are multiple copies of the fourth SRAM; each stores one definition (i.e. the average and standard deviation of each of the "four values" between each stroke pair).

The intermediary FSMs are the workers. They each read from an SRAM and produce the next level of data. For example, the "Line" FSM uses the stroke coordinates (SRAM1) to calculate the stroke lengths and angles (SRAM2). The "LinePair" FSM takes that and produces the length/angle ratios between line pairs (SRAM3). Finally, the "Definition" FSM averages these values and produces the definitions (SRAM4). "ScoreCalculator" FSM, on the other hand, compares data stored in the SRAMs 3 and 4 to calculate score. Each of these worker FSMs have direct access to the ALU in order to do the computations they need.

The SRAM interfaces are able to write and read from the SRAMs at different sequences. For example, the third SRAM interface writes in the line pair properties of one training example at a time. However, when the "Definition" FSM asks for data, the interface has to be able to pick the same property of the same line pair from every training example.

The Major FSM controls the order of operation. The user can initiate the Major FSM with the 'train' or the 'recognize' switches. When the user tells the system to train, the Major FSM starts the "Line" FSM, waits until it is done, and starts the "LinePair" FSM. It does this for all the examples in the training set, until all the line properties accumulate in the third SRAM. Then, the Major FSM starts the "Definition" FSM which reads the data in the third SRAM, and creates a new definition.

Similarly, when the 'recognize' switch is turned on, the "Line" and "LinePair" FSMs run in order, but only once. Then the ScoreCalculator compares data in the third SRAM, with the contents of all the copies of the fourth SRAM. It sends the score from each comparison to the Major FSM. The Major FSM keeps track of the lowest score (i.e. lowest error), and when all comparisons are made, it communicates to the output stage what type of shape the system recognizes the new drawing to be (i.e. which definition produced the lowest error).

-Testing-

For individual testing purposes of the AI module, an internal module was built to supply the system with pre-calculated inputs at expected times. The inputs first trained the system with a training set of right-angled isosceles triangles and then asked to recognize a new triangle that was not right-angled. ModelSim was used primarily to simulate how each part of the system behaved with these inputs. The code was debugged until the system could compute the expected error value (Java code was used to compute what error value to expect). When it worked in simulation, it worked on the labkit as well.

Display Stage – Sedrick Tydus

Our system displays its outputs on a VGA monitor. VGA differs from a video in a few important ways. Most importantly, instead of using luminance and chrominance values, each pixel is represented as a combination of the combination of the colors red, blue and green. Although pixels are represented differently than in video, VGA is still a stream of frames each made up of lines of pixels. VGA uses no interlacing and pixels are transmitted from left to right on the screen with the lines going from top to bottom. The end of each line is defined by a horizontal synchronization signal and likewise the end of each frame is defined by a vertical synchronization signal.

We have used the ADV7125 triple 8-bit video DAC to create the analog signals to drive the VGA display. On each cycle of the pixel clock the ADV7125 reads an 8-bit digital value for each color and outputs the analog values with a two-cycle pipeline delay. The ADV7125 also takes a composite blank and synch signal which override the RGB inputs. Both signals are active low and a blank signal forces all three DAC outputs to 0 (black) while a sync signal forces the green output of the DAC to its sync level which is below black. Additionally, the VGA must be directly supplied with a vertical and horizontal synch. These sync signals correspond to the composite sync, except they must be delayed two cycles to account for the pipeline delay of the ADV7125.

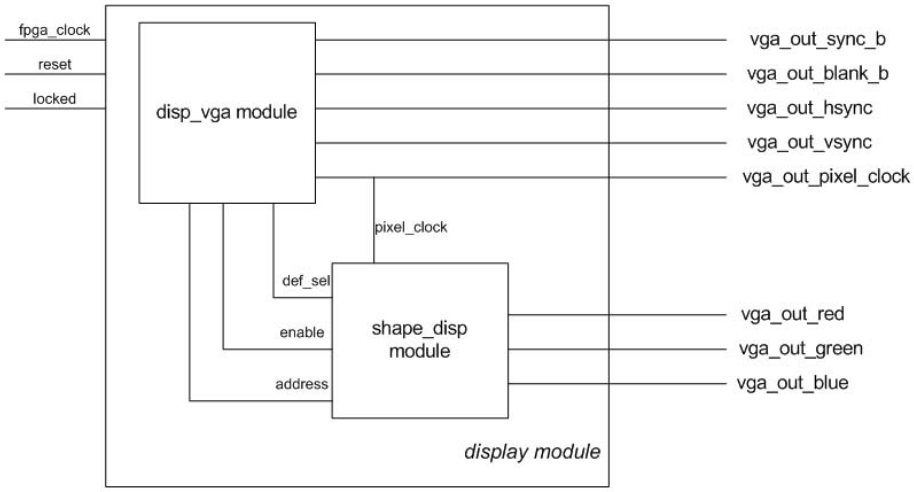

Figure D1: Block Diagram of Display Module

There are two main sub-modules in our system which control the data sent to the ADV7125 and the hsync and vsync signals which are sent directly to the display. The disp_vga sub-module creates the VGA control signals and addresses the ROMs and RAM on which images are stored. Internally it has its own sub-module which is dedicated to generating all sync and blanking signals. The shape_disp connects the address from the disp_vga module to all memories and selects which output to use. All code for these modules is included in the appendix. Figure # is a top level block diagram of the display system.

-sync_gen-

The sync_gen module and the disp_vga module work hand in hand to create the necessary outputs on each cycle of the pixel clock. The pixel clock is created using a DCM. We take the 27 MHz clock supplied by the lab kit and create a 25 MHz pixel clock. Our pixel clock is 25 MHz because we have chosen our resolution to be 640 x 480. This means that each line of video must have 640 active pixels, 16 front porch pixels, 96 sync signals and 48 back porch pixels. Therefore each horizontal line has a total of 800 pixels. Each frame thus will have 480 active lines, 11 front porch lines, 2 vertical sync lines and 31 back porch lines. Each frame has a total of 524 lines.

The sync_gen module has two counters; one keeps track of the pixel count the other keeps track of the line count. When either of these reaches their maximum value they are reset. Based on the values of these counters the appropriate sync signals are generated. The vsync and hsync signals are asserted two cycles earlier than all other VGA control signals to account for the pipeline delay of the DAC.

-disp_vga-

The disp_vga module is responsible for drawing shapes to the screen. It does this by addressing memories which have been loaded with the display images for our system. The display of our system uses the upper left hand corner of the screen to show the definition shape and the rest of the screen is used for the user input shape. This module uses the pixel count and line count generated by the sync_gen module to determine the region of the screen and create the appropriate address for the various memory units which contain all images to be displayed. If the location is within the upper left region then the address runs from 0 to 16383 to display a 128 x 128 image from ROMs. Otherwise (when in the remaining region of

the screen) the address will increment to display the 320 x 240 user input image that is stored in RAM. The address from this module is output and sent to all memories in our system.

-shape_disp-

The shape_disp module is an fsm which selects which shape in memory to display. The user input is always displayed, but the definition shape is only shown after processing takes place and a definition shape is selected. This fsm takes two inputs from the processing stage, an enable signal and a selector which chooses the shape to display.
The fsm will stay in the IDLE state until it receives the enable state. In the ENABLE state the fsm looks at the selector inputs and switches state accordingly. Our system stores four definition states, thus two bits are required for the def_sel input. The inputs can therefore be 0 through 3 and result in the display of an isosceles triangle, square, equilateral triangle, and a square respectively. The display of each shape has a corresponding state in which the RGB VGA outputs (vga_out_red, vga_out_green, vga_out_blue) are connected to the data pins of the appropriate ROM. Figure # is a state transition diagram of the shape_disp module.

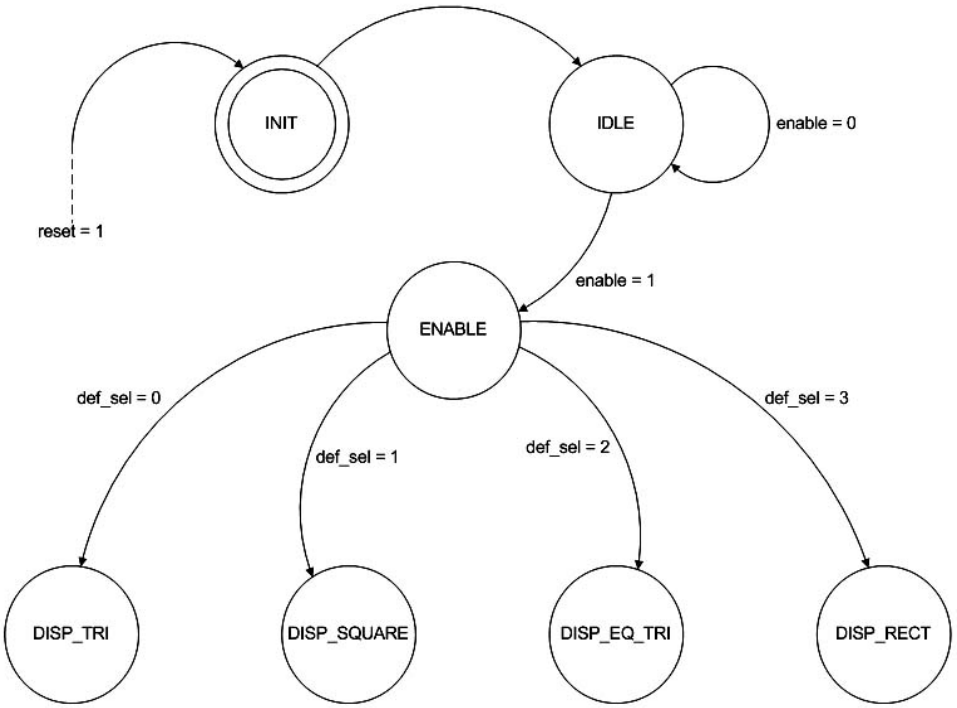

Figure D2: State Transition Diagram for shape_disp FSM

-Testing-

The display system was easily isolated and therefore easy to test. The display module was simulated in pieces first ensuring that all sync signals were generated correctly. From here, we began to test the writing of various images to ROM. The ability to display a single image in a single location enabled us to move forward and test the display of multiple images. The switches and buttons on the labkit were used to simulate the inputs from the AI module. Given the long compile and build times it was very important to ensure proper simulation before attempting to test and debut on the labkit.

Conclusion

There were a number of choices to make while designing the AI module in particular: maximum number of lines in a single drawing, number of examples in a training set and maximum number of definitions. The choices made would only affect the size of the memory units to be used. Since the labkit's resources were limited, we chose to work with a maximum of 8-line shapes, have four examples in a training set and a maximum of four definitions. For future/real-life use the capacity can be easily increased by introducing more memory space.

Another design choice made was to have the user draw a shape always in the same way. Otherwise we would need to design a line-matcher that would perform an intensive search to match the lines of two shapes. However, this would be beyond the aims of 6.111 and computationally too expensive. Considering each shape has eight lines with two end-points, line-matching would increase the amount of computations by (16*14*12*10*8*6*4*2). Instead a heuristic search could be used in the future to bring down the number of computations, but the complexity would too high for the purposes of a 6.111 project and might need a processor to implement. Similarly, curves are not allowed in the drawings. The algorithm could be improved to allow curves, but it would be difficult to reliably input curvatures into the system.

To conclude, the SmartKit successfully shows how an Artificial Intelligence concept can be implemented in a digital design. Given pre-calculated input (of shape coordinates), it can be trained to learn shapes, and it reliably recognizes new examples by computing extremely precise error values, no matter how complicated the shapes are and no matter whether they are scaled, translated or rotated.

Appendix

This appendix includes a sample of the relevant Verilog code for the modules described in this report:

-Input Stage-

```
module video_decoder(clk,reset,decode_en,data_in, data_out,data_valid, SOF, SEF, SAV, EAV,state,
                     count);

//Control Inputs
//input locked;
input clk;                                  //global 27 Mhz clock
input reset;                            //global reset
input decode_en;                            //enables the decoder

//Video Input - output video data from ADV7185 video decoder chip: YCrCb values
input [7:0] data_in; //[9:0] data_in;//8 or 10 bits. for 10 bits:bottom 2 bits rep.
fraction(2'b01=.25)

//Debugging Outputs
output [3:0] state;
output [1:0] count;

//Control Outputs

//Decoded Video Outputs
output [7:0] data_out;
output data_valid;                      //indicates decoded data is valid
output SOF;                                 //start of odd field
output SEF;                                 //start of even field
output SAV;                                 //start of active video
output EAV;                                 //end of active video
//output registers
reg EAV;
reg SAV;
reg [7:0] data_out;
reg data_valid;
//internal registers
reg active;
reg F_current;
reg V_current;
reg [1:0] count;
reg count_reset;
reg [3:0] state;                                    //current state of FSM
reg [3:0] nextstate;                            //next state of FSM
//wires
wire One, F, V, H, P3, P2, P1, P0;
wire valid_check;
wire EAV_valid;
wire SAV_valid;
//Assigns
assign One = data_in[7];
assign F   = data_in[6];
assign V   = data_in[5];
assign H   = data_in[4];
assign P3  = data_in[3];
assign P2  = data_in[2];
assign P1  = data_in[1];
assign P0  = data_in[0];
assign valid_check = ((P3 == (V ^ H)) &&
                              (P2 == (F ^ H)) &&
                              (P1 == (F ^ V)) &&
                              (P0 == (F ^ V ^ H)));

//check to see if end of active video(EAV) is valid: H=1, One=1, valid_check=1;
assign EAV_valid = (H && One && valid_check);
//check to see if start of active video(SAV) is valid: H=0, One=1, valid_check=1;
assign SAV_valid = (~H && One && valid_check);

always @ (posedge clk)
```

```
begin
  if (reset == 1)    active <= 1'b0;
  else if (decode_en & SOF) active <= 1'b1;
end

always @ (posedge clk)
begin
  if (reset == 1)
    begin
      F_current <= 1'b1;//resets the current F and V to line 1 values: F=1, V=1
      V_current <= 1'b1;
    end
  else if (SAV == 1)
    begin
      F_current <= F; //F_current is reg'd version of F, so it's the most recent F, while F is
the previous F
      V_current <= V; //since conditioned on SAV==1, F&V current denote field&vertical of only
active video
    end
end

//basic counter which counts on every clock edge of 27MHz clk
always @ (posedge clk)
begin
  if (reset == 1)
    begin                           //counts from 0 to 3 and repeats
      count <= 2'b00;         //count = 0:on first  half of first  sample(Cb0)
    end                       //count = 1:on second half of first  sample(Y0)
  else                        //count = 2:on first  half of second sample(Cr0)
    begin                           //count = 3:on second half of second sample(Y1)
      if (count_reset == 1) count <= 2'b00;
        else count <= (count + 2'b01);
    end
end


//reference white = 100IRE = 8'h64
//black level = 7.5IRE = 8'h07
//blank level = 0IRE = 8'h00
//sync level = 40IRE = 8'h28
always @ (posedge clk)
begin
  if (count == 1 || count == 3)//when count==1 or count==3, the given data_in should be a Y value
    begin
      if (data_in > 8'h50) data_out <= 8'hff;//threshold for a white pixel (data_in >
8'h50=80IRE)
      else data_out <= 8'h00; //else data_out will be black(0)
    end
  else data_out <= 8'h80;
end

//explanation of (SOF) and (SEF):
//if we look at all 525 lines of video, we realize:
//an odd field begins (SOF) when the current field is odd (F_curr=0) and previous field was even
(F=1)
//similar results hold for (SEF): the current field is even (F_current=1) and the previous
//field was odd (F=0).  We AND this with SAV to ensure that (SOF) and (SEF) are really the start
//of only the active portions of the odd and even fields.
assign SOF = (SAV & ~F_current & F);
assign SEF = (SAV & F_current & ~F);


//parameters to denote the TRS. TRS denotes either a SAV or EAV
parameter TRS0 = 8'hff;
parameter TRS1 = 8'h00;
parameter TRS2 = 8'h00;

//parameters (states) for FSM
parameter EAV0 = 0;
parameter EAV1 = 1;
parameter EAV2 = 2;
```

```
parameter EAV3 = 3;
parameter SAV0 = 4;
parameter SAV1 = 5;
parameter SAV2 = 6;
parameter SAV3 = 7;
parameter SKIP = 8;

always @ (posedge clk)
begin
  if (reset == 1) state <= EAV0;
  else state <= nextstate;
end

always @ (state or active or V_current or count or data_in or EAV_valid or SAV_valid)
begin
  count_reset = 1'b0;
  EAV = 1'b0;
  SAV = 1'b0;
  data_valid = 1'b0;

  case(state)
        EAV0:   begin
                          data_valid = (active && ~V_current && (count == 2'b00)); //EAV is start
of line, so count==0
                          if (data_in == TRS0) nextstate = EAV1; //data_in = 8'hff(found TRS0)
                          else nextstate = EAV0;
                        end
        EAV1:   begin
                           if (data_in == TRS1) nextstate = EAV2;
                           else nextstate = EAV0;
                        end
        EAV2:   begin
                          if (data_in == TRS2) nextstate = EAV3;
                          else nextstate = EAV0;
                        end
        EAV3:   begin
                          if (EAV_valid)
                            begin
                              EAV = 1'b1;
                                 nextstate = SAV0;
                            end
                          else nextstate = EAV0;
                        end
        SAV0:   begin
                          if (data_in == TRS0) nextstate = SAV1;
                          else nextstate = SAV0;
                        end
        SAV1:   begin
                          if (data_in == TRS1) nextstate = SAV2;
                          else nextstate = SAV0;
                        end
        SAV2:   begin
                          if (data_in == TRS2) nextstate = SAV3;
                          else nextstate = SAV0;
                        end
        SAV3:   begin
                          if (SAV_valid)
                            begin
                              count_reset = 1'b1;
                                SAV = 1'b1;
                                nextstate = SKIP;
                            end
                          else nextstate = SAV0;
                        end
        SKIP:   begin                           // SKIP state is needed to "skip" the next 8-bit
word after TRS0,
                          nextstate = EAV0;     // TRS1, and TRS2 (i.e. skips the XY word after hFF
h00 h00)
                        end

        default: nextstate = EAV0;
```

```
  endcase
end//always (state or active or V_current or count or data_in or EAV_valid or SAV_valid) begin
endmodule // video_decoder
```

-Processing Stage-

```
module floatDivide(clk, reset, go, a, b, o, done);

   input clk, reset;
   input go;
   input[31:0] a, b;
   output[31:0] o;
   output done;
   reg done;
   reg [31:0] o;
   reg [4:0] state, nextstate;
   reg [7:0] exponentOutput, exponentOutput2;
   reg [22:0] mantissaOutput;
   reg [47:0] dividend;
   reg [23:0] divisor;
   reg [24:0] mantissaQuotient;
   reg [47:0] x, y, z;
   reg [24:0] w;
   reg [4:0] counter;
   reg sign;
   reg enableIDLE;
   reg enableSTART_DIVISION;
   reg enableDO_DIVISION;
   reg enableDO_DIVISION1;
   reg enableDO_DIVISION2;
   reg enableDO_DIVISION3;
   reg enableDO_DIVISION4;
   reg enableDO_DIVISION5;
   reg enableDO_DIVISION6;
   reg enableDO_DIVISION7;
   reg enableDO_DIVISION8;
   reg enableDO_DIVISION9;
   reg enableDO_DIVISION10;
   reg enableDO_DIVISION11;
   reg enableDO_DIVISION12;
   reg enableDO_DIVISION13;
   reg enableDO_DIVISION14;
   reg enableDO_DIVISION15;
   reg enableDO_DIVISION16;
   reg enableDO_DIVISION17;
   reg enableDO_DIVISION18;
   reg enableDO_DIVISION19;
   reg enableDO_DIVISION20;
   reg enableDO_DIVISION21;
   reg enableDO_DIVISION22;
   reg enableDO_DIVISION23;
   reg enableDO_DIVISION24;
   reg enableDO_DIVISION25;
   reg enableDO_DIVISION26;
   reg enableOUTPUTREADY;

   parameter IDLE = 5'h00;
   parameter START_DIVISION = 5'h01;
   parameter DO_DIVISION1 = 5'h02;
   parameter DO_DIVISION = 5'h03;
   parameter DO_DIVISION2 = 5'h04;
   parameter DO_DIVISION3 = 5'h05;
   parameter DO_DIVISION4 = 5'h06;
   parameter DO_DIVISION5 = 5'h07;
   parameter DO_DIVISION6 = 5'h08;
   parameter DO_DIVISION7 = 5'h09;
```

```
parameter DO_DIVISION8 = 5'h0a;
parameter DO_DIVISION9 = 5'h0b;
parameter DO_DIVISION10 = 5'h0c;
parameter DO_DIVISION11 = 5'h0d;
parameter DO_DIVISION12 = 5'h0e;
parameter DO_DIVISION13 = 5'h0f;
parameter DO_DIVISION14 = 5'h10;
parameter DO_DIVISION15 = 5'h11;
parameter DO_DIVISION16 = 5'h12;
parameter DO_DIVISION17 = 5'h13;
parameter DO_DIVISION18 = 5'h14;
parameter DO_DIVISION19 = 5'h15;
parameter DO_DIVISION20 = 5'h16;
parameter DO_DIVISION21 = 5'h17;
parameter DO_DIVISION22 = 5'h18;
parameter DO_DIVISION23 = 5'h19;
parameter DO_DIVISION24 = 5'h1a;
parameter DO_DIVISION25 = 5'h1b;
parameter DO_DIVISION26 = 5'h1c;
parameter OUTPUTREADY = 5'h1d;

always @(posedge clk or negedge reset)  begin
   if (!reset) begin
     state <= IDLE;
     done <= 0;
     o <= 0;
            exponentOutput2 <= 0;
          dividend <= 0;
            divisor <= 0;
            sign <= 0;
          mantissaQuotient <= 0;
            exponentOutput <= 0;
          mantissaOutput <= 0;
            counter <= 0;
            x <= 0;
            y <= 0;
            w <= 0;
            z <= 0;
     end
   else begin
     state <= nextstate;

            if (enableIDLE) begin
                   done <= 0;
                   end

            if (enableSTART_DIVISION) begin
                   exponentOutput2 <= a[30:23] - b[30:23] + 127;
            dividend <= {1'b1, a[22:0], 24'h000000};
                   divisor <= {1'b1, b[22:0]};
                   sign <= a[31] ^ b[31];
            mantissaQuotient <= 0;
                   end

            if (enableDO_DIVISION) begin
                   counter <= counter + 1;
                   if (x > y) begin
                   mantissaQuotient <= mantissaQuotient + w;
                   dividend <= dividend - z;
                   end
                   end

            if (enableDO_DIVISION1) begin
                   x <= {24'h000000, dividend[47:24]};
                   y <= {24'h000000, divisor};
                   w <= {1'b1, 24'h000000};
                   z <= {divisor, 24'h000000};
                   end

            if (enableDO_DIVISION2) begin
                   x <= {23'h000000, dividend[47:23]};
```

```
        y <= {24'h000000, divisor};
        w <= {1'h0, 1'b1, 23'h000000};
        z <= {1'h0, divisor, 23'h000000};
        end

if (enableDO_DIVISION3) begin
        x <= {22'h000000, dividend[47:22]};
        y <= {24'h000000, divisor};
        w <= {2'h0, 1'b1, 22'h000000};
        z <= {2'h0, divisor, 22'h000000};
        end

if (enableDO_DIVISION4) begin
        x <= {21'h000000, dividend[47:21]};
        y <= {24'h000000, divisor};
        w <= {3'h0, 1'b1, 21'h000000};
        z <= {3'h0, divisor, 21'h000000};
        end

if (enableDO_DIVISION5) begin
        x <= {20'h00000, dividend[47:20]};
        y <= {24'h000000, divisor};
        w <= {4'h0, 1'b1, 20'h00000};
        z <= {4'h0, divisor, 20'h00000};
        end

if (enableDO_DIVISION6) begin
        x <= {19'h00000, dividend[47:19]};
        y <= {24'h000000, divisor};
        w <= {5'h00, 1'b1, 19'h00000};
        z <= {5'h00, divisor, 19'h00000};
        end

if (enableDO_DIVISION7) begin
        x <= {18'h00000, dividend[47:18]};
        y <= {24'h000000, divisor};
        w <= {6'h00, 1'b1, 18'h00000};
        z <= {6'h00, divisor, 18'h00000};
        end

if (enableDO_DIVISION8) begin
        x <= {17'h00000, dividend[47:17]};
        y <= {24'h000000, divisor};
        w <= {7'h00, 1'b1, 17'h00000};
        z <= {7'h00, divisor, 17'h00000};
        end

if (enableDO_DIVISION9) begin
        x <= {16'h0000, dividend[47:16]};
        y <= {24'h000000, divisor};
        w <= {8'h00, 1'b1, 16'h0000};
        z <= {8'h00, divisor, 16'h0000};
        end

if (enableDO_DIVISION10) begin
        x <= {15'h0000, dividend[47:15]};
        y <= {24'h000000, divisor};
        w <= {9'h000, 1'b1, 15'h0000};
        z <= {9'h000, divisor, 15'h0000};
        end

if (enableDO_DIVISION11) begin
        x <= {14'h0000, dividend[47:14]};
        y <= {24'h000000, divisor};
        w <= {10'h000, 1'b1, 14'h0000};
        z <= {10'h000, divisor, 14'h0000};
        end

if (enableDO_DIVISION12) begin
        x <= {13'h0000, dividend[47:13]};
        y <= {24'h000000, divisor};
```

```
        w <= {11'h000, 1'b1, 13'h0000};
        z <= {11'h000, divisor, 13'h0000};
        end

if (enableDO_DIVISION13) begin
        x <= {12'h000, dividend[47:12]};
        y <= {24'h000000, divisor};
        w <= {12'h000, 1'b1, 12'h000};
        z <= {12'h000, divisor, 12'h000};
        end

if (enableDO_DIVISION14) begin
        x <= {11'h000, dividend[47:11]};
        y <= {24'h000000, divisor};
        w <= {13'h0000, 1'b1, 11'h000};
        z <= {13'h0000, divisor, 11'h000};
        end

if (enableDO_DIVISION15) begin
        x <= {10'h000, dividend[47:10]};
        y <= {24'h000000, divisor};
        w <= {14'h0000, 1'b1, 10'h000};
        z <= {14'h0000, divisor, 10'h000};
        end

if (enableDO_DIVISION16) begin
        x <= {9'h000, dividend[47:9]};
        y <= {24'h000000, divisor};
        w <= {15'h0000, 1'b1, 9'h000};
        z <= {15'h0000, divisor, 9'h000};
        end

if (enableDO_DIVISION17) begin
        x <= {8'h00, dividend[47:8]};
        y <= {24'h000000, divisor};
        w <= {16'h0000, 1'b1, 8'h00};
        z <= {16'h0000, divisor, 8'h00};
        end

if (enableDO_DIVISION18) begin
        x <= {7'h00, dividend[47:7]};
        y <= {24'h000000, divisor};
        w <= {17'h00000, 1'b1, 7'h00};
        z <= {17'h00000, divisor, 7'h00};
        end

if (enableDO_DIVISION19) begin
        x <= {6'h00, dividend[47:6]};
        y <= {24'h000000, divisor};
        w <= {18'h00000, 1'b1, 6'h00};
        z <= {18'h00000, divisor, 6'h00};
        end

if (enableDO_DIVISION20) begin
        x <= {5'h00, dividend[47:5]};
        y <= {24'h000000, divisor};
        w <= {19'h00000, 1'b1, 5'h00};
        z <= {19'h00000, divisor, 5'h00};
        end

if (enableDO_DIVISION21) begin
        x <= {4'h0, dividend[47:4]};
        y <= {24'h000000, divisor};
        w <= {20'h00000, 1'b1, 4'h0};
        z <= {20'h00000, divisor, 4'h0};
        end

if (enableDO_DIVISION22) begin
        x <= {3'h0, dividend[47:3]};
        y <= {24'h000000, divisor};
        w <= {21'h000000, 1'b1, 3'h0};
```

```
                z <= {21'h000000, divisor, 3'h0};
                end

            if (enableDO_DIVISION23) begin
                x <= {2'h0, dividend[47:2]};
                y <= {24'h000000, divisor};
                w <= {22'h000000, 1'b1, 2'h0};
                z <= {22'h000000, divisor, 2'h0};
                end

            if (enableDO_DIVISION24) begin
                x <= {1'h0, dividend[47:1]};
                y <= {24'h000000, divisor};
                w <= {23'h000000, 1'b1, 1'h0};
                z <= {23'h000000, divisor, 1'h0};
                end

            if (enableDO_DIVISION25) begin
                x <= dividend[47:0];
                y <= {24'h000000, divisor};
                w <= {24'h000000, 1'b1};
                z <= {24'h000000, divisor};
                end

            if (enableDO_DIVISION26) begin
                if (mantissaQuotient[24]) begin
                exponentOutput <= exponentOutput2;
                mantissaOutput <= mantissaQuotient[23:1];
                        end
                else begin
                exponentOutput <= exponentOutput2 - 1;
                mantissaOutput <= mantissaQuotient[22:0];
                end
                end

            if (enableOUTPUTREADY) begin
                counter <= 0;
                if (a[30:0] == 0) o <= 32'h00000000;
            else if (b[30:0] == 0) o <= 32'h7f800000;
            else o <= {sign, exponentOutput, mantissaOutput};
            done <= 1;
                end

    end
end

always @(state or go)
  begin
    enableIDLE = 0;
    enableSTART_DIVISION = 0;
    enableDO_DIVISION = 0;
    enableDO_DIVISION1 = 0;
    enableDO_DIVISION2 = 0;
    enableDO_DIVISION3 = 0;
    enableDO_DIVISION4 = 0;
    enableDO_DIVISION5 = 0;
    enableDO_DIVISION6 = 0;
    enableDO_DIVISION7 = 0;
    enableDO_DIVISION8 = 0;
    enableDO_DIVISION9 = 0;
    enableDO_DIVISION10 = 0;
    enableDO_DIVISION11   = 0;
    enableDO_DIVISION12   = 0;
    enableDO_DIVISION13   = 0;
    enableDO_DIVISION14   = 0;
    enableDO_DIVISION15   = 0;
    enableDO_DIVISION16   = 0;
    enableDO_DIVISION17   = 0;
    enableDO_DIVISION18   = 0;
    enableDO_DIVISION19   = 0;
    enableDO_DIVISION20   = 0;
```

```
enableDO_DIVISION21    = 0;
enableDO_DIVISION22    = 0;
enableDO_DIVISION23    = 0;
enableDO_DIVISION24    = 0;
enableDO_DIVISION25    = 0;
enableDO_DIVISION26    = 0;
enableOUTPUTREADY =    0;

case (state)

  IDLE:begin
       enableIDLE = 1;
     if (go) nextstate = START_DIVISION;
     else nextstate = IDLE;
  end

  START_DIVISION: begin
     nextstate = DO_DIVISION1;
     enableSTART_DIVISION = 1;
  end

  DO_DIVISION: begin
     enableDO_DIVISION = 1;
        case (counter)
               5'd00:  nextstate = DO_DIVISION2;
               5'd01:  nextstate = DO_DIVISION3;
               5'd02:  nextstate = DO_DIVISION4;
               5'd03:  nextstate = DO_DIVISION5;
               5'd04:  nextstate = DO_DIVISION6;
               5'd05:    nextstate = DO_DIVISION7;
               5'd06:  nextstate = DO_DIVISION8;
               5'd07:  nextstate = DO_DIVISION9;
               5'd08:  nextstate = DO_DIVISION10;
               5'd09:  nextstate = DO_DIVISION11;
               5'd10:  nextstate = DO_DIVISION12;
               5'd11:  nextstate = DO_DIVISION13;
               5'd12:  nextstate = DO_DIVISION14;
               5'd13:  nextstate = DO_DIVISION15;
               5'd14:    nextstate = DO_DIVISION16;
               5'd15:  nextstate = DO_DIVISION17;
               5'd16:  nextstate = DO_DIVISION18;
               5'd17:  nextstate = DO_DIVISION19;
               5'd18:  nextstate = DO_DIVISION20;
               5'd19:  nextstate = DO_DIVISION21;
               5'd20:  nextstate = DO_DIVISION22;
               5'd21:  nextstate = DO_DIVISION23;
               5'd22:  nextstate = DO_DIVISION24;
               5'd23:  nextstate = DO_DIVISION25;
               5'd24:  nextstate = DO_DIVISION26;
               endcase
     end

  DO_DIVISION1:       begin
     nextstate = DO_DIVISION;
     enableDO_DIVISION1 = 1;
     end

  DO_DIVISION2:       begin
     nextstate = DO_DIVISION;
     enableDO_DIVISION2 = 1;
     end

  DO_DIVISION3:       begin
     nextstate = DO_DIVISION;
     enableDO_DIVISION3 = 1;
     end


  DO_DIVISION4:       begin
     nextstate = DO_DIVISION;
     enableDO_DIVISION4 = 1;
```

```
   end

 DO_DIVISION5:       begin
   nextstate = DO_DIVISION;
   enableDO_DIVISION5 = 1;
   end

DO_DIVISION6:        begin
   nextstate = DO_DIVISION;
   enableDO_DIVISION6 = 1;
   end

DO_DIVISION7:        begin
   nextstate = DO_DIVISION;
   enableDO_DIVISION7 = 1;
   end

DO_DIVISION8:        begin
   nextstate = DO_DIVISION;
   enableDO_DIVISION8 = 1;
   end

DO_DIVISION9:        begin
   nextstate = DO_DIVISION;
   enableDO_DIVISION9 = 1;
   end

DO_DIVISION10:       begin
   nextstate = DO_DIVISION;
   enableDO_DIVISION10 = 1;
   end

DO_DIVISION11:       begin
   nextstate = DO_DIVISION;
   enableDO_DIVISION11 = 1;
   end

DO_DIVISION12:       begin
   nextstate = DO_DIVISION;
   enableDO_DIVISION12 = 1;
   end

DO_DIVISION13:       begin
   nextstate = DO_DIVISION;
   enableDO_DIVISION13 = 1;
   end

DO_DIVISION14:       begin
   nextstate = DO_DIVISION;
   enableDO_DIVISION14 = 1;
   end

DO_DIVISION15:       begin
   nextstate = DO_DIVISION;
   enableDO_DIVISION15 = 1;
   end

DO_DIVISION16:       begin
   nextstate = DO_DIVISION;
   enableDO_DIVISION16 = 1;
   end

DO_DIVISION17:       begin
   nextstate = DO_DIVISION;
   enableDO_DIVISION17 = 1;
   end

DO_DIVISION18:       begin
   nextstate = DO_DIVISION;
   enableDO_DIVISION18 = 1;
   end
```

```
          DO_DIVISION19:       begin
             nextstate = DO_DIVISION;
             enableDO_DIVISION19 = 1;
             end

          DO_DIVISION20:       begin
             nextstate = DO_DIVISION;
             enableDO_DIVISION20 = 1;
             end

          DO_DIVISION21:       begin
             nextstate = DO_DIVISION;
             enableDO_DIVISION21 = 1;
             end

          DO_DIVISION22:       begin
             nextstate = DO_DIVISION;
             enableDO_DIVISION22 = 1;
             end

          DO_DIVISION23:       begin
             nextstate = DO_DIVISION;
             enableDO_DIVISION23 = 1;
             end

          DO_DIVISION24:       begin
             nextstate = DO_DIVISION;
             enableDO_DIVISION24 = 1;
             end

          DO_DIVISION25:       begin
             nextstate = DO_DIVISION;
             enableDO_DIVISION25 = 1;
             end

          DO_DIVISION26: begin
             nextstate = OUTPUTREADY;
             enableDO_DIVISION26 = 1;
             end

          OUTPUTREADY: begin
                enableOUTPUTREADY = 1;
             nextstate = IDLE;
          end



        endcase // case(state)

     end

endmodule // floatDivide
```

-Output Stage-

```
module display(fpga_clock,
               locked,
               reset,
               enable,
               def_sel,
              vga_out_sync_b,
              vga_out_blank_b,
              vga_out_pixel_clock,
              vga_out_hsync,
              vga_out_vsync,
               vga_out_red,
               vga_out_green,
               vga_out_blue);
```

```
  input fpga_clock, reset;
  input locked;
  input enable:
  input [1:0] def_sel;
  output vga_out_sync_b, vga_out_blank_b, vga_out_pixel_clock, vga_out_hsync, vga_out_vsync;
  output [7:0] vga_out_red, vga_out_green, vga_out_blue;

  wire [13:0] address;
  wire vga_out_pixel_clock;
  shape_disp(fpga_clock,
             reset,
             locked,
             vga_out_pixel_clock,
             shape_reset,
             address,
              enable,
              def_sel,
             vga_out_red,
             vga_out_green,
             vga_out_blue);
  disp_vga disp_vga(fpga_clock,
                    vga_out_sync_b,
                   vga_out_blank_b,
                   vga_out_pixel_clock,
                   vga_out_hsync,
                   vga_out_vsync,
                   address);
endmodule

module disp_vga(fpga_clock,
                vga_out_sync_b,
               vga_out_blank_b,
               vga_out_pixel_clock,
               vga_out_hsync,
               vga_out_vsync,
               disp_address);

   input fpga_clock;
   output vga_out_sync_b, vga_out_blank_b;
   output vga_out_pixel_clock;
   output vga_out_hsync, vga_out_vsync;
   output [16:0] disp_address;
   reg [16:0] disp_address;
   reg vga_out_sync_b, vga_out_blank_b;
   reg vga_out_hsync, vga_out_vsync;
   wire clock_fpga, clock_fpgab;
   wire pixel_clock;

   BUFG fpga_clock_buf (.O(clock_fpgab), .I(clock_fpga));
   DCM pixel_clock_dcm (.CLKFB(clock_fpgab),
                        .CLKIN(fpga_clock),
                        .RST(1'b0),
                        .CLK0(clock_fpga),
                        .CLKFX(pixel_clock));
   // synthesis attribute DLL_FREQUENCY_MODE of pixel_clock_dcm is "LOW"
   // synthesis attribute DUTY_CYCLE_CORRECTION of pixel_clock_dcm is "TRUE"
   // synthesis attribute STARTUP_WAIT of pixel_clock_dcm is "TRUE"
   // synthesis attribute DFS_FREQUENCY_MODE of pixel_clock_dcm is "LOW"
   // synthesis attribute CLKFX_DIVIDE of pixel_clock_dcm is 27
   // synthesis attribute CLKFX_MULTIPLY of pixel_clock_dcm is 25
   // synthesis attribute CLK_FEEDBACK of pixel_clock_dcm  is "1X"
   // synthesis attribute CLKOUT_PHASE_SHIFT of pixel_clock_dcm is "NONE"
   // synthesis attribute PHASE_SHIFT of pixel_clock_dcm is 0
   // synthesis attribute clkin_period of pixel_clock_dcm is "37.04ns"

   assign vga_out_pixel_clock = pixel_clock;
   wire [10:0] pixel_count;
   wire hsync, vsync, comp_sync, blank, v_blank;
   wire [10:0] line_count;
   sync_gen sync(.pixel_clock(pixel_clock),
                 .reset(1'b0),
```

```
                .h_synch_delay(hsync),
                .v_synch_delay(vsync),
                .comp_synch(comp_sync),
                .blank(blank),
                .v_blank(v_blank),
                .pixel_count(pixel_count),
                 .line_count(line_count));

  //generate pixel data
   always @ (posedge pixel_clock) begin
      vga_out_hsync = !hsync;
      vga_out_vsync = !vsync;
      vga_out_sync_b = ~comp_sync;
      vga_out_blank_b = ~blank;
   end
   always @ (negedge pixel_clock) begin
       if ((line_count < 128) && (pixel_count < 128))
         begin
           disp_address = ((128 * line_count) + pixel_count);
         end
     //else if ((line_count > 351) && (pixel_count < 128))
       //  begin
       //    disp_address = ((128 * (line_count - 11'd352)) + pixel_count);
     //  end
       else disp_address = 10;
   end

endmodule

module sync_gen(pixel_clock, reset, h_synch, v_synch,
                comp_synch, blank, pixel_count, line_count, v_blank);

   input pixel_clock;
   input reset;
   output h_synch, v_synch;
   output comp_synch;
   output blank, v_blank;
   output [10:0] pixel_count, line_count;
   reg [10:0] pixel_count;
   reg [10:0] line_count;
   reg h_synch, v_synch;
   reg h_synch_delay1, v_synch_delay1;
   reg h_synch_delay2, v_synch_delay2;
   reg h_c_synch, v_c_synch;
   reg comp_synch;
   reg h_blank, v_blank;
   reg blank;

   always @ (posedge pixel_clock) begin
      if (reset)
        begin
           pixel_count = 11'h000;
           line_count = 11'h000;
        end
      else if (pixel_count == 799)
        begin
           pixel_count = 11'h000;
           if (line_count == 523)
             line_count = 11'h000;
           else
             line_count = line_count + 1;
        end
      else
        pixel_count = pixel_count +1;
   end

   always @ (posedge pixel_clock) begin
      if (reset)
        h_synch_delay1 = 1'b0;
      else if (pixel_count == 655)
        h_synch_delay1 = 1'b1;
```

```
   else if (pixel_count == 751)
     h_synch_delay1 = 1'b0;
end

always @ (posedge pixel_clock) begin
   if (reset)
     v_synch_delay1 = 1'b0;
   else if (pixel_count == 799)
     begin
        if (line_count == 490)
          v_synch_delay1 = 1'b1;
        else if (line_count == 492)
          v_synch_delay1 = 1'b0;
     end
end

always @ (posedge pixel_clock) begin
   if (reset)
     begin
        h_synch_delay1 <= 1'b0;
        v_synch_delay1 <= 1'b0;
        h_synch_delay2  <= 1'b0;
        v_synch_delay2  <= 1'b0;
     end
   else
     begin
        h_synch_delay2 <= h_synch_delay1;
        v_synch_delay2 <= v_synch_delay1;
        h_synch  <= h_synch_delay2;
        v_synch  <= v_synch_delay2;
     end
end

always @ (posedge pixel_clock) begin
   if (reset)
     h_blank = 1'b0;
   else if (pixel_count == 638)
     h_blank = 1'b1;
   else if (pixel_count == 798)
     h_blank = 1'b0;
end

always @ (posedge pixel_clock) begin
   if (reset)
     v_blank = 1'b0;
   else if ((line_count == 479) && (pixel_count == 798)))
     v_blank = 1'b1;
   else if ((line_count == 524) && (pixel_count == 798))
     v_blank = 1'b0;
end


always @ (posedge pixel_clock) begin
   if (reset)
     blank = 1'b0;
   else if (h_blank || v_blank)
     blank = 1'b1;
   else
     blank = 1'b0;
end

always @ (posedge pixel_clock) begin
   if (reset)
     h_c_synch = 1'b0;
   else if (pixel_count == 654)
     h_c_synch = 1'b1;
   else if (pixel_count == 750)
     h_c_synch = 1'b0;
end

always @ (posedge pixel_clock) begin
```

```
      if (reset)
        v_c_synch = 1'b0;
      if (pixel_count == 798)
        if (line_count == 490)
          v_c_synch = 1'b1;
        else if (line_count == 492)
          v_c_synch = 1'b0;
   end

   always @ (posedge pixel_clock) begin
      if (reset)
        comp_synch = 1'b0;
      else
        comp_synch = (v_c_synch ^ h_c_synch);
   end

endmodule

module shape_disp(fpga_clock,
                  reset,
                  locked,
                  pixel_clock,
                  shape_reset,
                  in_address,
                   enable,
                   def_sel,
                  vga_out_red,
                  vga_out_green,
                  vga_out_blue);

  input fpga_clock, reset, locked;
  input pixel_clock;
  input shape_reset;
  input [13:0] in_address;
  input enable;
  input [1:0] def_sel;
  output [7:0] vga_out_red, vga_out_green, vga_out_blue;
  reg [7:0] vga_out_red, vga_out_green, vga_out_blue;
  wire [23:0] square_rom_out, triang_rom_out, eq_triang_rom_out, rect_rom_out;
  square_rom square_rom1(in_address, pixel_clock, square_rom_out);
  triang_rom triang_rom1(in_address, pixel_clock, triang_rom_out);
  eq_triang_rom eq_triang_rom1(in_address, pixel_clock, eq_triang_rom_out);
  rect_rom rect_rom1(in_address, pixel_clock, rect_rom_out);

  reg [2:0] state, next;

  parameter INIT = 0;
  parameter IDLE = 1;
  parameter ENABLE = 2;
  parameter DISP_SQUARE = 3;
  parameter DISP_TRI = 4;
  parameter DISP_RECT = 5;
  parameter DISP_EQ_TRI = 6;

  always @ (posedge fpga_clock) begin
    if (reset)
      state <= INIT;
    else begin
      state <= next;
    end
  end

  always @ (state or enable or def_sel or locked or in_address) begin
    case (state)
      INIT:
        begin
            if (locked) next = IDLE;
            else next = INIT;
          end
      IDLE:
        begin
```

```
      vga_out_red <= 8'h00;
    vga_out_green <= 8'h00;
    vga_out_blue <= 8'h00;
         if (enable)
           next = ENABLE;
    else next = IDLE;
     end
ENABLE:
  begin
      vga_out_red <= 8'h00;
    vga_out_green <= 8'h00;
    vga_out_blue <= 8'h00;
      if (def_sel == 2'b01) begin
          vga_out_red <= square_rom_out[23:16];
          vga_out_green <= square_rom_out[15:8];
          vga_out_blue <= square_rom_out[7:0];
          next = DISP_SQUARE;
    end
         if (def_sel == 2'b00) begin
           vga_out_red <= triang_rom_out[23:16];
           vga_out_green <= triang_rom_out[15:8];
           vga_out_blue <= triang_rom_out[7:0];
           //next = DISP_TRI;
         end
         if (def_sel == 2'b11) begin
           vga_out_red <= rect_rom_out[23:16];
           vga_out_green <= rect_rom_out[15:8];
           vga_out_blue <= rect_rom_out[7:0];
           next = DISP_RECT;
         end
         if (def_sel == 2'b10) begin
           vga_out_red <= eq_triang_rom_out[23:16];
           vga_out_green <= eq_triang_rom_out[15:8];
           vga_out_blue <= eq_triang_rom_out[7:0];
           next = DISP_EQ_TRI;
         end
         else next = ENABLE;
     end

DISP_SQUARE:
     begin
       if (shape_reset) next = IDLE;
          else begin
       vga_out_red <= square_rom_out[23:16];
          vga_out_green <= square_rom_out[15:8];
          vga_out_blue <= square_rom_out[7:0];
          next = DISP_SQUARE;
          end
     end
DISP_TRI:
     begin
       if (shape_reset) next = IDLE;
          else begin
       vga_out_red <= triang_rom_out[23:16];
          vga_out_green <= triang_rom_out[15:8];
          vga_out_blue <= triang_rom_out[7:0];
       next = DISP_TRI;
          end
  end
  DISP_RECT:
     begin
       if (shape_reset) next = IDLE;
          else begin
       vga_out_red <= rect_rom_out[23:16];
          vga_out_green <= rect_rom_out[15:8];
          vga_out_blue <= rect_rom_out[7:0];
          next = DISP_RECT;
          end
     end
   DISP_EQ_TRI:
     begin
```

```
            if (shape_reset) next = IDLE;
               else begin
            vga_out_red <= eq_triang_rom_out[23:16];
               vga_out_green <= eq_triang_rom_out[15:8];
               vga_out_blue <= eq_triang_rom_out[7:0];
               next = DISP_EQ_TRI;
               end
          end
       default:
        next = INIT;

    endcase
 end
endmodule
```