# Deep3D

Alice Reyzin and Elliott Prechter
6.111 Final Project
12/9/2004

# Abstract

The Deep3D project was an attempt to create a hardware transformation and rasterization 3d rendering system. It would read a scene of triangle from a ROM and allow the user to navigate through the scene by modifying their position and orientation through a Nintendo controller. The triangles would be transformed and drawn in wire-frame to a VGA monitor in real-time. The entire pipeline would be run through twice with different eye coordinates and rendered once in red and once in blue for each eye, thus giving a stereoscopic view of the scene.

The final project ran into many issues during testing and implementation, especially regarding timing. Both the rasterization system and the transformation system would not fit on a single 10K70 FPGA, and thus only ran separately. Also, the final project did not have two rendering passes for each eye, although this was mostly due to time constraints on the project deadline.

# Introduction

## *System Brief*

Deep3D is a digital circuit designed to transform and render a 3D world in real-time. The user will control a floating camera with three degrees of freedom: two angles and forward/backward movement. The environment is pre-specified and loaded into ROM memory. Scene data is stored as a list of triangles that all exist in world-space. Each triangle consists of 3 vertices with x,y,z coordinates stored in 17-bit sign magnitude fixed-point format, with 8-bits decimal.

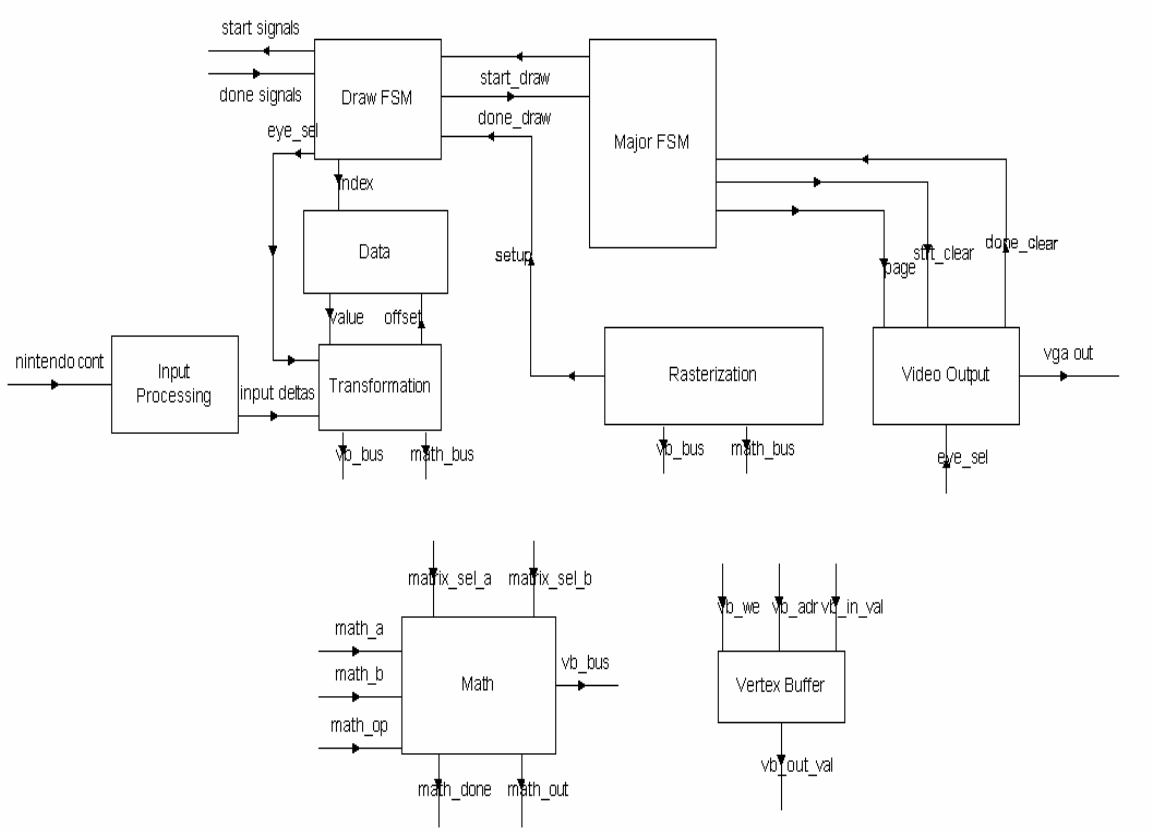
General control flow consists of first clearing a back-buffer. Upon completion of this step, the system takes input from the Nintendo controller, and computes an inverse transform for the current position and orientation of the user. The system then sequentially looks up each triangle, and loads the associated vertices into a vertex buffer. A camera to world transform is performed on the vertices in the vertex buffer, and then they are projected into screen coordinates. At this point the clipping unit clips the three lines against the screen, and finally the rasterizer draws to the back-buffer. Once all the triangles are drawn, the video buffers are swapped, and the process is repeated. The front-buffer is continuously output to a VGA monitor.

## *High-Level Design*

## Module Description and Implementation

### Block Diagram

Figure 2: High Level Block Diagram




### Control FSMs

**High Level Control Flow**

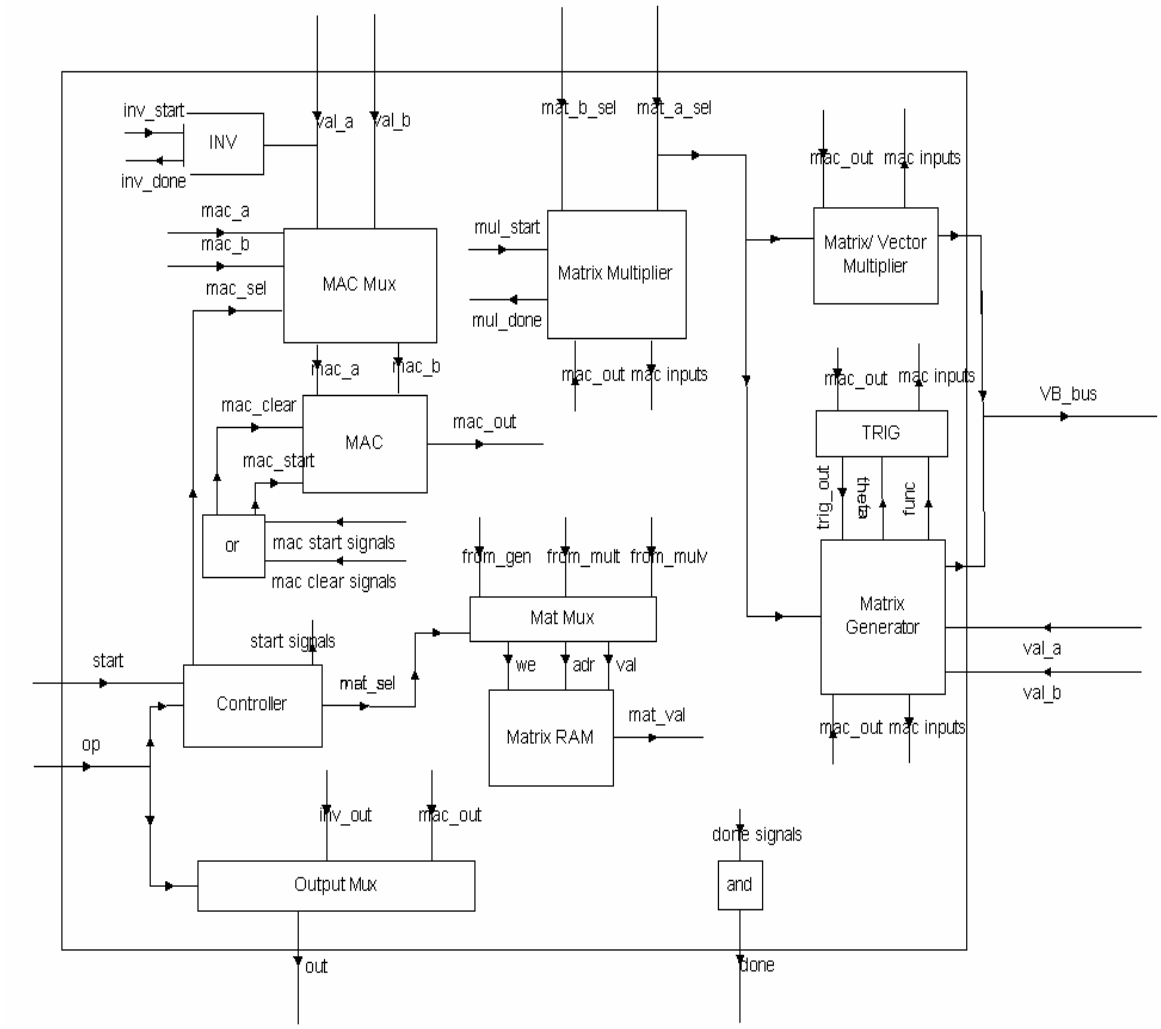
Figure 1 is a view of the system at the highest level of abstraction. The system is broken down into nine main units. The Major FSM is responsible for sending a start signal to the video output module so that it clears the back buffer, and then sending a start signal to the Draw FSM to begin drawing the scene. The Draw FSM first tells the transformation system to update the camera's position/orientation by reading input from the Input Processing module. Then, it sequentially reads in the triangles from Data. Each triangle is sent to the transformation system to convert it into three final x and y coordinates for rasterization, and the triangle is drawn in wireframe by the rasterization

module. The DrawFSM cycles through all triangles and then sends a done signal, signaling the MajorFSM to flip the *page_sel* signal so that the front and back video buffers are swapped. The MajorFSM then repeats this process continuously.

Breakdown by Partner Responsibility

**Section 1: Modules Designed/Implemented by Elliott**

## Math Module




In order to perform 3D transformation, Deep3D needed the following capabilities: a fixed-point number representation, transform generation, transform application, transform concatenation, sin and cosine calculation, multiply and accumulate, and a way to compute multiplicative inverses. The math unit performs all of these functions.

The most notable input to the math unit is *op*, which can be one of the following values:

| | |
|---|---|
| OP_INV | 0 |
| OP_MUL | 1 |
| OP_MUL_AC | 2 |
| OP_GEN_TRANS | 3 |
| OP_GEN_ROT_NOD | 4 |
| OP_GEN_ROT_SHAKE | 5 |
| OP_MAT_MUL | 6 |
| OP_MAT_VEC_MUL | 7 |

These values are mostly self-explanatory. OP_MUL_AC stands for multiply and accumulate, which tells the math unit to multiply the inputs *val_a* and *val_b*, and add that value to the last value that the MAC computed. This is useful for doing operations such as dot products. OP_GEN_ROT_NOD tells the math unit to generate a rotation matrix about the angle specified in *val_a* (see **Trig** section for details on angle format), and around the x axis, and to store it in the matrix *mat_sel_a*. OP_MAT_VEC_MUL will be explained in the section describing the Matrix/Vector multiplier.

The main job of the Math unit is to provide control signals to its sub-modules, and to handle muxing of various signals between each sub-module depending on the operation being performed. For example, the MAC's control signals are driven externally when performing OP_MUL or OP_MUL_AC, but when performing OP_MAT_MUL or OP_MAT_VEC_MUL, the math unit needs to drive the MAC's control signal.

**MAC**

The heart of the math unit is the Multiply and Accumulate (MAC). The number format is 17-bits: 1 bit for positive/negative and 8.8 fixed-point for the rest. Inputs consist of 17-bit *a* and *b* inputs, *start* and *clear* signals, and a 17-bit *z* output and a 1-bit *overflow* and *done* output. The MAC unit can be further broken down into a 8-bit/8-bit multiplier, a 17-bit converter for both to and from 2's complement, a mux to select appropriate bits from each 8-bit multiply to add to the sum, and an fsm for control flow.

When *start* pulses high, *clear* is checked: if it is one then the operation is a multiply, and if it is zero then a multiply is performed and added to the previous total. At the next stage, a series of four multiplies are performed: one for the upper/upper 8-bits, upper/lower 8-bits, lower/upper 8-bits, and lower/lower 8-bits. An addition or subtraction to an internal *total* register is performed after each stage, as well as necessary shifting of the product. For upper/upper multiplication, the lower 8 bits of the product are selected and added as the upper 8-bits of a 17-bit fixed point. For upper/lower or lower/upper multiplication, the product bits 12 through 0 are selected and shifted down by four to be added to total. Finally, for lower/lower multiplication, the high 8 bits of the product are selected and shifted down by 8 before adding to total.

The addition and multiplication stages are done in a pipelined fashion, so that the addition of the previous multiply is performed at the same time as the next 8-bit parts of *a* and *b* are being multiplied. This modules's *done* signal goes high when in idle stage, which occurs five clock cycles after *start* pulses high.

**Trig**

The trig unit is required for generation of rotation matrices (covered in a later section). It's responsible for calculation of sine and cosine. It was designed so that rotations would be smooth and accurate: rather than using a lookup table technique, it uses the MAC and series expansions of sine and cosine.

Input to the trig unit is an 11-bit *angle*, a 1-bit signal *cosine* specifying calculation of cosine instead of sine, a 17-bit output *z*, and the following control signals for the mac unit: *mac_a*, *mac_b*, *mac_clear*, *mac_start*, *mac_done*. The format for the angle is neigther radians nor degrees: angle ranges from 0 to 0x7FF, which correspond to zero and two pi. This format was chosen for a very specific reason. First of all, by using a format whose 2PI value is 0x7FF, this value naturally wraps around when added or subtracted to, so modulations throughout the rest of the system are not needed when dealing with angles. Secondly, it can be multiplied by the scaling factor 0x000C9 to convert it to radians, which allows for the full range of 0 to 2PI that our 8.8 fixed point number system can represent. Another property of it that is useful is that the scaling factor to convert to radians is represented well in our system (i.e. small value of trailing decimal bits that must be cut to fit into an 8-bit decimal part).

The trig units uses the following expansions for sine and cosine calculation:

sinx = x - x^2/6
cosx = 1 - x^2/2 + x^4/24

Both of these are fairly accurate but only from 0 to PI/4. Thus, although it seems possible to calculate sine/cosine with only sine (since it is the simpler expansion), it actually is far too inaccurate since ranges from PI/4 to PI/2 aren't accurate enough.

The trig module first checks to see if *cosine* is high. If so, it negates the angle and subtracts it from PI/2, so that it may be treated as a sine calculation. Then, trig identities are employed from state to state, using to MAC, to convert to the range of 0 to PI/2. If the angle is greater than PI/4, it subtracts it from PI/2 and uses the cosine expansion, otherwise it uses the sine expansion.

**Inverter**

The inverter was designed by Alice, so refer to **Section 2**, later in the paper.

**Matrix RAM**

In order to perform matrix operations, at least three 4x4 matrices needed to be stored. This is due to the fact that a matrix multiplication needs *a* and *b* terms as well as a place to store the result. Matrix RAM is indexed by *mat*, *row¸col*, which correspond to the matrix (0, 1, or 2), the column and row. The last row is always 0,0,0,1, so these values are returned via. logic and not stored.

**Matrix Generator**

The matrix generator is responsible for generating primitive transformations: rotation parallel to the x axis, rotation parallel to the y axis, and translation. Inputs are 17-

bit *a,b,c* values, and a 2-bit *opcode* that correspond to translation, x rotation, and y rotation sequentially. It directly controls the trig unit since this is the only module that depends on it, and also utilizes Matrix RAM. If the *opcode* is for translation, then the translation values are *a,b,c*. If the *opcode* is rotation then the angle is the first 11-bits of *a*.

The Matrix Generator cycles through all the rows and columns of the result matrix, and utilizes the trig unit to calculate sine or cosine when needed, and writes to the Matrix RAM. All angles are assume to correspond to counter-clockwise rotations. The specified matrix that is being written is controlled by the Math module's control logic.

### Matrix Multiplier

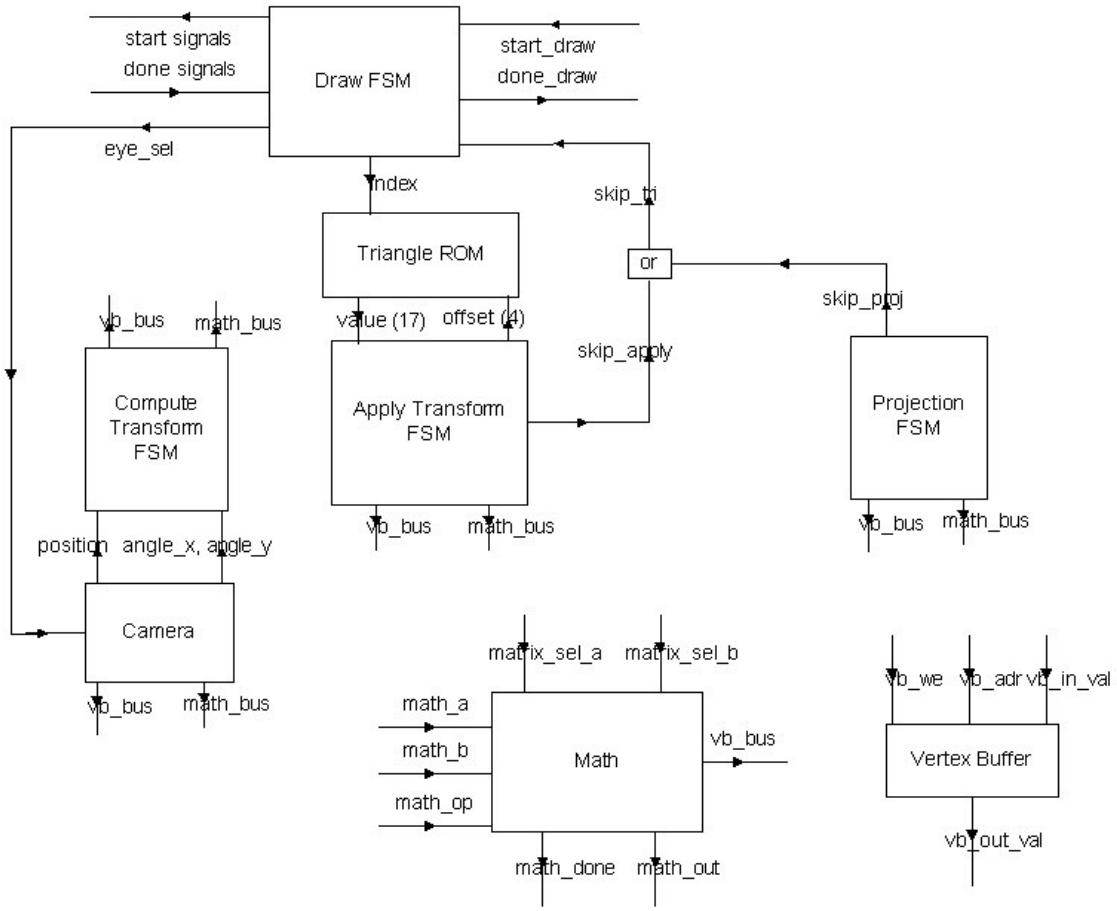
The matrix multiplier is needed to concatenate transforms. It utilizes the MAC unit and the Matrix RAM. Important inputs are *mat_a_sel* and *mat_b_sel*, which select the two matrices that are being multiplied. In the current design, these values are specified are being unequal and of the range zero to three. These select the two matrices multiplied (A*B), and the result is written to the third matrix in matrix RAM. Since no output matrix is directly specified, this means that a matrix cannot be squared; however this is an operation that Deep3D does not require.

The Matrix Multiplier cycles through each row and column of the result matrix. For each, it performs a dot-product via. the MAC module by fetching numbers from the current row from matrix A, and numbers from the current column from matrix B.

### Matrix/Vector Multiplier

The matrix/vector multiplier performs operations on the vertices contained in the Vertex Buffer (which will be described in the **Transformation System**). It utilizes the Vertex Buffer, the MAC unit, and Matrix RAM. Vertices are stored as a fourth-dimensional homogenous coordinates, of the form (x,y,z,1). This allows 4x4 matrices to perform translation. The Matrix/Vector multiplier works by performing a dot product on each row of the matrix specified in *mat_a_sel* with the each of the vertices in the Vertex Buffer, and then storing the results back into the Vertex Buffer. Thus all three vertices in the Vertex Buffer are affected (see **Transformation System** for description of the Vertex Buffer).

## Transformation System




The transformation system is responsible for converting the triangle in the vertex buffer from world-space to screen-space depending on the camera's orientation and position. It is also responsible for making sure those triangles too close to the camera or those with coordinates calculated with overflows by the math unit are completely skipped.

Almost every module in the Transformation System utilize both the Math Module and the Vertex Buffer, and thus another job of the Transformation System is handle muxing of these signals depending on which of the Minor FSMs are busy. Each system (Compute Transform, Apply Transform, and Projection) are run sequentially, and thus using the *done* signals for the minor FSMs is enough for muxing I/O signals to both the Math and Vertex Buffer.

**Vertex Buffer**

Although the Vertex Buffer is actually placed within the Math Module, it is treated as part of the transformation system. The vertex buffer's input signals may be driven by many parts of the system, but the Math Module always takes precedence is the current operation is OP_MAT_VEC_MUL (apply a matrix to the vertices in the vertex buffer.) The placement inside the math unit is only at the syntax level, since it allowed for easier testing: the control signals operate independently of the math unit unless the operation is, naturally, OP_MAT_VEC_MUL.

The purpose of the Vertex Buffer is to store the current vertices being transformed. This allows transformation of large amounts of data without actually passing vertices through the system. The vertex buffer is essentially a RAM that holds three vertices indexed by 2 the bit signal *vert*. The dimension is chosen by the 2-bit signal *xyz*.

**Compute Transform**

The compute transform instructs the math unit to create an inverse-camera transform and place it into the matrix 0 position in Matrix RAM. Inputs to this module are *neg_pos_x*, *neg_pos_y*, and *neg_pos_z* for the negative position, and *neg_angle_nod*, and *neg_angle_shake*, for the negative angles parallel to the y_axis and x_axis, respectively.

The compute transform module performs several steps to create an inverse-camera transform. First, it signals the math module to generate a rotation matrix parallel to the y_axis with the negative of the camera's y angle and store it is matrix 0. Then the x rotation matrix is generated, with the negative of the camera's x angle, and stored in matrix 1, and a multiplication is performed and output to matrix 2. A translation matrix is generated and output to matrix 1, and finally matrices 1 and 2 are multiplied, leaving the final transform in matrix 0: $T^{-1}R_y^{-1}R_x^{-1}$.

**Apply Transform**

The apply transform module is responsible for fetching the current triangle's vertices and placing them in the Vertex Buffer, and applying the inverse camera transform. The apply transform utilizes the math unit and the vertex buffer. It controls the coordinates by using the *tri_offset* parameter to the triangle ROM, which selects the component of the triangle (x1, y1, z1, x2, y2, etc.). The current triangle is specified by the DrawFSM.

**Projection**

The projection system is responsible for taking camera-space vertices and converting them to screen-space. For each vertex in the vertex buffer, the z value is checked. The camera's near place (the plane that all triangle are ultimately projected to) is 160 units in front of the camera. Thus, the value of ½ was selected as the smallest z value allowed to be projected (since this results in a triangle that is twice its size in ROM). If the z value is smaller, then the entire triangle must be rejected, and the output *z_skip* is set high, telling the DrawFSM to continue to process the next triangle.

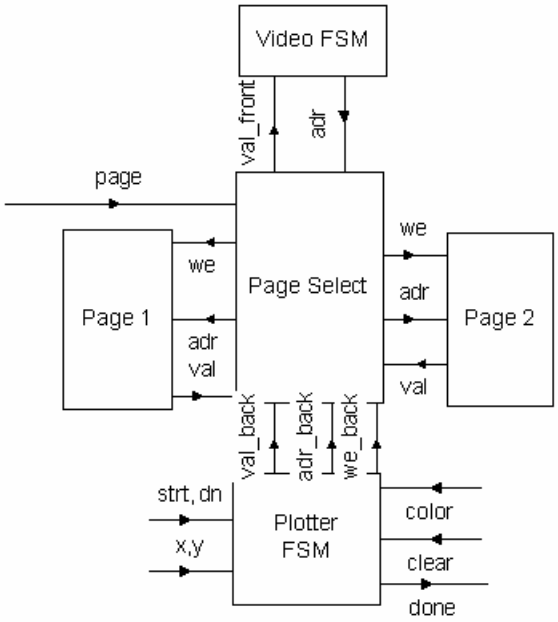
Next, an inverse z coordinate is computed. This value is multiplied by one-eighth of the screen width to give a scaling factor for the x coordinate. Although the mathematically correct value to use is one-half (if we want to have a 90 degree field of vision horizontally), we needed about 2 more bits of whole part for our numbers (i.e. it uses a 10.6 fixed point system). This is due to the fact that screen coordinates ranges from

0 to 320 or 0 to 240, and we need to represent off-screen points, so -1024 to 1024 seemed a worthy representation.

After this multiplication by screen_width/8/z, then one eight of the screen width is added to the x coordinate, which is the final conversion to screen space. The same operation is performed on the y coordinate, except that the scaling factor is almost pre-multiplied by an aspect ratio (giving around 60 degree field of vision vertically), and the y value is actually subtracted from one-either of the screen height rather than added (since the y-axis is flipped from the perspective of a VGA monitor).

Utilizing a matrix for this operation was considered, but was discarded for several reasons. First of all, division by the z coordinate is not part of the matrix/vector apply. Although this could be done by modifying the vertex buffer to include a w coordinates (and using standard homogenous coordinate math), the 8.8 fixed-point system is not accurate enough for such an operation. The exact problem is that multiplication by one-eigth the screen width or height followed by division by z will most likely result in an overflow, and thus this value must be computed at a different step.

## Video System




### Page Select

Page select choose which page of memory is used as the back buffer, and which is the front buffer, via the 1-bit signal *page*_sel. This is used to implement a standard page-flipping technique. The back-buffer is driven by the rasterization system while the front-buffer is read by the Video module.

### Video

The video system outputs to a VGA monitor at 320x240 resolution. The Verilog source is almost identical to that presented in 6.111 lecture, except with horizontal sync and blanking times modified for a 12MHz clock and a 320 pixel horizontal resolution. The video output is done at a resolution of 320x480 by doubling the memories 240 resolution.

### Plotter

The plotter is detailed in **Section 2**.

## Section 2: Modules Designed/Implemented by Alice

# Math Unit

### Inverter

This unit calculates the multiplicative inverse of a number. Because our number format is symmetrical, the inverter has almost no inputs for which it overflows or contains an invalid output. The only two such values are 0 and 1/256, since 1/256 has inverse 256, which is just barely too large for our number format to deal with. The inverter tests directly for these inputs and returns "FFFF" if it encounters one of them.

Other numbers are processed almost exactly the way that one would think of handling long division by hand. The sign bit is set to the sign bit of the input. A variable *numerator* is then created and set to 0002. A variable *current_bit* is set to 0. The inverter then starts to loop. If *numerator* is greater than the input, we subtract the input from *numerator* and set the bit of the output corresponding to *current_bit* to 1 (ie. if *current_bit* is 0, we would modify the most significant bit). Otherwise, *numerator* is doubled, the bit of the output corresponding to *current_bit* remains 0, and *current_bit* is incremented. When every bit of the output has been determined, that is, we have completed the loop for *current_bit* equal to15, we are done.

This process of determining the output bit by bit is straightforward and fast. The only problem results from the fact that the inverse of a very large number will not be very accurate, since it will only contain two significant bits. This is more of a limitation of our number format than of the inverter itself. Since even greater limitations pertaining to large numbers are present in the MAC unit, this inaccuracy is not highly significant.

# Rasterization System

The rasterization system is responsible for implementing 2D line clipping and for determining the pixels that best correspond to a given triangle's wireframe. This system repeatedly calls the plotter in order to plot pixels onto the screen. Because plotting every pixel is a relatively long process, the rasterizer is designed to run in parallel with the transformation pipeline. So, the rasterizer FSM's states are separated into two distinct stages, of which one uses common system resources, and the other one can run fully independently.

The first stage uses the math module and the vertex buffer, and therefore cannot run at the same time as the next triangle is being transformed. In this stage the rasterizer accesses the vertex buffer in order to copy from it the necessary x and y coordinates. It then runs the clipping on all three lines in the triangle. Because it is necessary to calculate slope when doing 2D clipping, the clipper needs to use the MAC module and the inverter, which are parts of the math unit.

After the first stage is complete, the rasterizer sends out a *done_setup* signal, which lets the drawFSM know that it can give control of the vertex buffer and the math unit to the transform computation. In the second stage, the rasterizer calls the line_draw sub module once for every line and calls the plotter for every pixel that needs to be drawn on the screen.

**Clipping**

The clipping sub unit uses the math module in order to compute the slope of the given line and to calculate its intersection with the screen. This module will only clip one endpoint of a line at a time, so if both endpoints are off of the screen, the module will need to be called twice in order to find both of the new end points. It makes the following assumptions about its input:

i) The endpoint to be clipped is off the screen – that is, clipping is necessary.
ii) The end points of the line are not both off the screen in the same direction. That is, the two endpoints cannot both be located above the screen, nor can they both be located to the left of the screen, etc.

The clipping unit has three main outputs: the x coordinate of the new endpoint, the y coordinate of the new end point and a *no_line* output indicating that the line does not cross the screen at all. If the *no_line* output is high, the line should be completely thrown out by the rasterizer.

Let *x_to_clip* denote the x coordinate of the end point to be clipped, and similarly for the y coordinate. The logic of the clipping unit is as follows:

a) Calculate dx and dy for use in later calculations. This does not require the math unit.
b) *x_to_clip* is outside of the horizontal range of the screen , calculate 1/dx using the inverter. If not, go to step *g*
c) Multiply the 1/dx output by dy using the math unit to calculate the slope
d) Multiply the slope by the difference between *x_to_clip* and the nearest on-screen x coordinate using the math unit. This will give the change in y along the line as we move into the valid x coordinate range for the screen.
e) Add the result of step *d* to *y_to_clip*.
f) If the result is within the y range of the screen, we have found a point at which our line intersect the screen and we are done. Otherwise, we continue to step *g*
g) Calculate dy. Then go through the calculation detailed in steps *c* through *e* except with x and y values interchanged. If the result is within the valid x range of the screen, we have found valid x and y coordinates and we are done. If the result is outside the valid x range, *no_line* goes high and we are also done.

One of the most difficult things about the clipping unit was figuring out an appropriate number format to use. The format of the output is 9 bits for the x coordinate and 8 bits for the y coordinate, where the point (0, 0) corresponds to the upper leftmost corner of the screen. The format for the inputs to the clipper is 13 bit sign magnitude. That is, any negative x coordinate is located to the left of the screen, whereas a positive x coordinate is either on the screen (if it is smaller than 320, corresponding to our screen resolution) or to the right of the screen. Similarly, a negative y coordinate is above the screen and a positive one is on the screen if it is smaller than 240 or below the screen otherwise. This format was chosen because it was sufficiently versatile – any line that has coordinates within 8 times the screen size will be correctly clipped – and because it allowed the calculations to use our math unit and still generate sufficiently accurate results.

**Line_Draw**

The line draw module takes in two points that are on the screen and calculates the screen pixels that most accurately represent the line between them. In order to do this, it uses Bresenham's algorithm, which has the benefit of relying fully on integer arithmetic, and therefore being both conceptually and logically simpler to implement.

Bresenham's algorithm does not guarantee that a line from point *a* to point *b* will be drawn the same way as one from point *b* to point *a*. Therefore, it is necessary to be sure that line draw always draws the line starting from the point with the smaller y coordinate. Otherwise, if two triangles shared a side, the resulting line could look fragmented and fractured.

After line draw chooses which point is the starting point and which is the ending point, it decides whether the dx is greater than the dy or vice versa. If the slope is less than one, then there is one pixel drawn per x coordinate on the screen, and there could be many pixels for the same y coordinate. Otherwise, there is one pixel drawn per y coordinate. After this decision is reached, the line draw module cycles through its states, updating the x and y coordinates accordingly until it reaches the endpoint.

# Input

The input into the system comes in from a simple Nintendo Emulation System controller. The controller contains eight buttons, four of which are arrow buttons. The arrow buttons are used to control the angle where the user is currently facing. For example, pressing the "up" button causes the user to "look up" and therefore causes all of the triangles displayed on the screen to shift downwards along a spherical trajectory. Pressing the "A" button causes the user to move backwards (out of the screen), while pressing the "B" button causes the user to move forwards (out of the screen).

There are five wires that connect the Nintendo controller to the rest of the system. The four inputs to the controller are *power*, *ground*, *latch*, and *pulse* and its one output is *data*. *Latch* and *pulse* are both provided by the input FSM. The serial input protocol for the Nintendo controller is a simple one – whenever the data needs to be read, *latch* is held high for 12um. At this point, the output of button "A" is stable on the *data* output from

the controller. *Data* is low if A is being pressed and high otherwise. At this point, the input FSM waits for 6us and then holds *pulse* high for another 6us, after which the output of button "B" becomes available. This process is repeated to read the outputs of buttons "select", "start", "up", "down", "left" and "right" respectively.

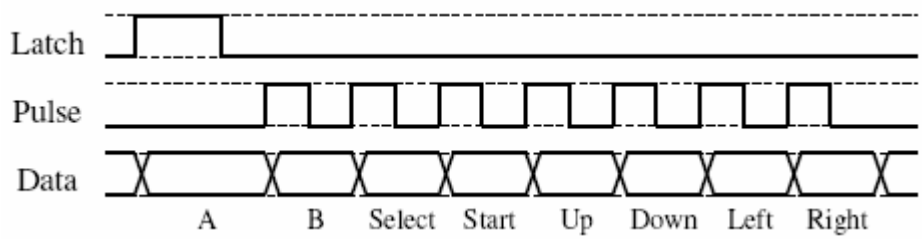


Figure 1: Nintendo Emulation Controller Input Protocol

The input FSM is called once per frame in order to determine the change in the user's viewing angles and position. The change in the angles and position is then added to registers that store the angles and position for the current transform. Because originally we were planning to call the input FSM many times per frame, the format for the "change in angle" outputs remained in the code as 11 bit numbers. However, the way that the input FSM is used currently, these could easily be changed to 2 bit numbers representing either one, negative one, or zero.

## Video System

The video system consists of two rams that store the data to display, a page flipping system that displays the contents of one RAM while writing to another, a video display system that handles syncing and blacking signals, and a plotter that writes to RAM and also contains the functionality of clearing the RAM.

### Plotter

The plotter consists, conceptually, of two separate FSMs. One of the FSMs clears the entire page of RAM. That is, it sequentially outputs the addresses and appropriate write enable signals and color in order for the RAM page to be blanked. This FSM starts executing on a *clear* signal.

The other FSM is one that writes a given color to a given point in RAM. This FSM begins running on the *start* signal. It calculates the appropriate address given the x and y screen coordinates of the point and also sets write enable appropriately. If the address calculated is out of range, it will simply write to address 0 instead. Although the plotter should never actually be passed invalid x or y coordinates, this check guarantees that if another module makes a mistake and sends the plotter incorrect coordinates, we will still be able to see something on the screen, rather than having all of the values in RAM unpredictably corrupted. This fix was very useful for debugging our video and rasterization systems.

When we originally calculated the timing for writing to our RAMs, we thought that a single clock cycle write enable high would be sufficient, since running at our 12MHz clock we had just enough time for the 70ns write to complete. However, due to imperfect and noisy wiring and an overheating lab kit, the number of clock cycles used

for a write gradually grew until it became large enough to be a very significant slow down.

## Testing and Debugging

The testing and debugging of this project took up a great deal of time. One of the most difficult systems to debug was the video system, due to the combination of hardware and software problems. One of the most frustrating things about testing the video system was the fact that as the lab kit heated up, more write cycles were needed for the RAM to work properly. This meant that the video would suddenly display completely incorrect values, just after we thought we had gotten it fixed.

However, once the video system was fully functional, we were able to do a much better job of debugging our other systems. For example, testing the rasterizer was extremely difficult in simulation since it was hard to calculate whether the points output would actually look right on the screen except for simple lines such as horizontal and vertical ones. Also, some of the simulations needed to be extremely long in order to show the entire operation of a module, and having a video output to test these modules was extremely important.

Unfortunately, some of the testing of the combined system remained difficult even with the video output working, due to the fact that compilation often took up to 10 minutes, and it was hard to be able to test every case thoroughly considering the amount of wait time incurred.

One of the bugs that was hardest to catch was the fact that the built in Verilog “>” and “<” operators did not seem to be working as expected in certain places. This caused clipping to generate completely invalid results and also caused problems in the rasterizer and plotter. We were never able to conclusively figure out what the operations were doing incorrectly, but once we built our own greater_than and less_than modules the bugs disappeared.

## Conclusion

Project results were mixed. The positive side was that all the major systems were completed and worked as designed. The negative side was that the clipping and rasterization system did not fit with the rest of the system on a single 10K70. We also did not have time to implement a separate camera for each of the user’s eyes, so that we could render each in a different color and use 3D glasses.

Both team members come primarily from the computer science background, and thus most testing that worked in simulation was assume to work in circuitry. One of the important lessons taken away from the project was to always over estimate the amount of time it takes to verify operation in circuitry even if simulation is working.

# Appendix: Verilog Source Code

## Code Implemented by Alice

### Inverter Code

```
module Inv(clk, a17, reset, start, done, z17);
      /*
            Contract: Do not send start signal when busy
      */

      input clk, start,reset; input [16:0] a17; output done;
      output [16:0] z17;

      reg[16:0] numerator, inv;     reg[3:0] current_bit;

      assign z17 = inv;

      reg state;
      parameter S_IDLE = 0;
      parameter S_BUSY = 1;

      always @ (posedge clk) begin
            if( reset) begin state <= S_IDLE; current_bit <= 0;
numerator <= 2'b10; inv <=0; end
            else if( start ) begin
                  //this is the one case of failure, treat h.01 as
positive or negative zero
                  if( a17[15:0] == 1 ) inv <= {a17[16],16'hFFFF}; else
begin
                        //overflow <= 0;
                        state <= S_BUSY;
                        inv[15:0] <= 0; //clear current inverse output
                        inv[16] <= a17[16];     //copy sign bit
                  end
            end

            if( (state == S_BUSY) & ~reset ) begin
                  if (numerator >= a17[15:0]) begin
                        numerator <= numerator - a17[15:0];
                        case(current_bit)
                              5'b00000: inv[15] <= 1;
                              5'b00001: inv[14] <= 1;
                              5'b00010: inv[13] <= 1;
                              5'b00011: inv[12] <= 1;
                              5'b00100: inv[11] <= 1;
                              5'b00101: inv[10] <= 1;
                              5'b00110: inv[9] <= 1;
                              5'b00111: inv[8] <= 1;
                              5'b01000: inv[7] <= 1;
                              5'b01001: inv[6] <= 1;
                              5'b01010: inv[5] <= 1;
```

```
                        5'b01011: inv[4] <= 1;
                        5'b01100: inv[3] <= 1;
                        5'b01101: inv[2] <= 1;
                        5'b01110: inv[1] <= 1;
                        5'b01111: inv[0] <= 1;
                        default:;
                  endcase
            end else begin
                  numerator <= numerator << 1;
            end

            if( current_bit == 15) begin  state <= S_IDLE;
            current_bit <= 0; numerator <= 2'b10; end
            else current_bit <= current_bit + 1;
      end
  end

  assign done = (state == S_IDLE) & ~start & ~reset;

endmodule
```

## Line Clipping Module Code

```
module trim_line(clk, start, done, x_to_trim, y_to_trim, x_end, y_end,
x_intersect,
                   y_intersect, math_a, math_b, op, math_start,
math_done, math_invalid, math_out, no_line);
input clk, math_done, math_invalid, start;
input[12:0] x_to_trim, y_to_trim, x_end, y_end;
input[16:0] math_out;

output done, math_start, no_line;
output[8:0] x_intersect;
output[7:0] y_intersect;
output[16:0] math_a, math_b;
output[2:0] op;

reg no_line;
reg[2:0] op;
reg[8:0] x_intersect;
reg[7:0] y_intersect;

reg [16:0] math_a;
reg[16:0] math_b;
reg math_start;

reg[13:0] dy;
reg[13:0] dx;

parameter X_MAX = 319;
parameter Y_MAX = 239;

reg[2:0] state;

parameter IDLE = 0;
parameter CALC_VARS = 1;
parameter PRESET = 2;
parameter CALC_INV_X = 3;
parameter FIND_SLOPE_X = 4;
parameter INTERSECT_WITH_LINE_X = 5;
parameter FIND_Y_COORD = 6;
parameter CALC_INV_Y = 7;
parameter FIND_SLOPE_Y = 8;
parameter INTERSECT_WITH_LINE_Y = 9;
parameter FIND_X_COORD = 10;

wire g1, g2, g3, g4, g5, g6;

greater_than gr1(y_to_trim, y_end, g1);
greater_than gr2(x_to_trim, x_end, g2);
```

```
greater_than gr3(x_to_trim, X_MAX, g3);
greater_than gr4(math_out, Y_MAX, g4);
greater_than gr5(math_out, X_MAX, g5);
greater_than gr6(state, FIND_X_COORD, g6);

always@(posedge clk)
begin
      if(start) state<=CALC_VARS;
else
      begin

      if(state == CALC_VARS)
      begin
            no_line <= 0;
            if(y_to_trim[12] != y_end[12]) dy[13:0] <= {y_to_trim[12],
y_to_trim[11:0] + y_end[11:0]};
            else if(g1) dy[13:0] <= {y_to_trim[12], y_to_trim[12:0] -
y_end[12:0]};
            else dy[13:0] <= {~y_to_trim[12], y_end[12:0] -
y_to_trim[12:0]};
            if(x_to_trim[12] != x_end[12]) dx[13:0] <= {x_to_trim[12],
x_to_trim[11:0] + x_end[11:0]};
            else if(g2) dx[13:0] <= {x_to_trim[12], x_to_trim[12:0] -
x_end[12:0]};
            else dx[13:0] <= {~x_to_trim[12], x_end[12:0] -
x_to_trim[12:0]};
            state <= PRESET;
      end

      if(state == PRESET)
            if(g3) //x needs trimming
            begin
            op <= 0;
            math_a <= {dx[13], 3'b000, dx[12:0]};
            state <= CALC_INV_X;
            math_start <=1;
      end
      else
            begin
            op <= 0;
            math_a <= {dy[13], 3'b000, dy[12:0]};
            state <= CALC_INV_Y;
            math_start <= 1;
      end //ends state == CALC_INV_START_X;

      if(state == CALC_INV_X & math_done)
            begin
            op <= 1;
            math_a <= math_out;
            math_b <= {dy[13], 3'b000, dy[12:0]};
            state <=  FIND_SLOPE_X;
            math_start <= 1;
      end //ends state == CALC_INV_X

      if(state == FIND_SLOPE_X & math_done)
            begin
            op<= 1;
```

```
            math_a <= math_out;
            math_b <= (x_to_trim[12] ==1)? {5'b00000, x_to_trim[11:0]}:
{5'b00000, x_to_trim[11:0] - X_MAX};
            state <= INTERSECT_WITH_LINE_X;
            math_start <= 1;
      end

      if(state == INTERSECT_WITH_LINE_X & math_done)
            begin
            op <= 2;
            math_a <= 9'b100000000;
            math_b <= y_to_trim;
            math_start <= 1;
            state <= FIND_Y_COORD;
      end

      if(state == FIND_Y_COORD & math_done)
            begin
            if(g4)
                  begin
                  op <= 0;
                  math_a <= {dy[13], 3'b000, dy[12:0]};
                  state <= CALC_INV_Y;
                  math_start <= 1;
            end
            else
                  begin
                  state <= IDLE;
                  x_intersect <= (x_to_trim[12] == 1)? 0: X_MAX;
                  y_intersect <= math_out[7:0];
            end
      end

      if(state == CALC_INV_Y & math_done)
            begin
            op <= 1;
            math_a <= math_out;
            math_b <= {dx[13], 3'b000, dx[12:0]};
            state <=  FIND_SLOPE_Y;
            math_start <= 1;
      end //ends state == CALC_INV_X

      if(state == FIND_SLOPE_Y & math_done)
            begin
            op<= 1;
            math_a <= math_out;
            math_b <= (y_to_trim[12] ==1)? {5'b00000, y_to_trim[11:0]}:
{5'b00000, y_to_trim[11:0] - Y_MAX};
            state <= INTERSECT_WITH_LINE_Y;
            math_start <= 1;
      end

      if(state == INTERSECT_WITH_LINE_Y & math_done)
            begin
            op <= 2;
            math_a <= 9'b100000000;
            math_b <= y_to_trim;
```

```
            state <= FIND_X_COORD;
            math_start <= 1;
      end

      if(state == FIND_X_COORD & math_done)
            begin
            if(g5)
                  begin
                  no_line <= 1;
                  state <=IDLE;
            end
            else
                  begin
                  state <= IDLE;
                  y_intersect <= (y_to_trim[12] == 1)? 0: Y_MAX;
                  x_intersect <= math_out[8:0];
            end
      end

      if(g6)
            begin
            state <= IDLE;
      end

      if(math_start == 1) math_start <= 0;

end // ends not start
end //ends always statement

assign done = (state == IDLE & ~start);

endmodule
```

## Rasterizer High Level Module Code

```
module rasterizer(clk, start, x_out, y_out, plotter_start,
plotter_done,
                  vb_point, vb_coord, v_buffer_out, math_a, math_b,
math_op, math_start, math_done, math_invalid, math_out, done_setup,
done);

input clk, start, math_done, math_invalid, plotter_done;
input[12:0] v_buffer_out;
input[16:0] math_out;
output[16:0] math_a, math_b;
output[2:0] math_op;

output math_start, done_setup, done, plotter_start;
output[1:0] vb_point, vb_coord; //x1, y1, z1, x2, y2, z2, x3, y3, z3
output[8:0] x_out;
output[7:0] y_out;

reg[8:0] l1_x1, l1_x2, l2_x1, l2_x2, l3_x1, l3_x2;
reg[7:0] l1_y1, l1_y2, l2_y1, l2_y2, l3_y1, l3_y2;

reg[12:0] x1, x2, x3, y1, y2, y3; //12 bits + 1 bit sign magnitude
reg trim_start, raster_start;

wire  trim_done, raster_done;
wire[13:0] x_to_trim, x_end, y_to_trim, y_end;

wire[8:0] x_intersect;
wire[7:0] y_intersect;

wire[8:0] x_coord0, x_coord1;
wire[7:0] y_coord0, y_coord1;

wire[10:0] dec;

line_draw draw(clk, x_coord0, y_coord0, x_coord1, y_coord1, x_out,
y_out,
                       plotter_start, plotter_done, raster_start,
raster_done, dec);

trim_line trim(clk, trim_start, trim_done, x_to_trim, y_to_trim, x_end,
y_end,
                  x_intersect, y_intersect, math_a, math_b, math_op,
math_start, math_done,
                  math_invalid, math_out, no_line);

reg[4:0] state;
```

```
reg skip_1, skip_2, skip_3;

parameter X_MAX = 320;
parameter Y_MAX = 240;

parameter IDLE = 0;
parameter SETUP_START = 1;
parameter GET_Y1 = 2;
parameter GET_X2 = 3;
parameter GET_Y2 = 4;
parameter GET_X3 = 5;
parameter GET_Y3 = 6;

parameter FIRST_LINE = 7;
parameter TRIM_FIRST1 = 8;
parameter TRIM_FIRST2 = 9;
parameter SECOND_LINE = 10;
parameter TRIM_SECOND1 = 11;
parameter TRIM_SECOND2 =12;

parameter THIRD_LINE = 13;
parameter TRIM_THIRD1 = 14;
parameter TRIM_THIRD2 = 15;

parameter DRAW_PREP = 16;
parameter DRAW_FIRST =17;
parameter DRAW_SECOND =18;
parameter DRAW_THIRD =19;

wire g1, g2, l1, l2, l3, l4, l5, l6;

greater_than gr1(start, DRAW_THIRD, g1);
greater_than gr2(state, TRIM_THIRD2, g2);

less_than less1(x1, X_MAX, l1);
less_than less2(y1, Y_MAX, l2);
less_than less3(x2, X_MAX, l3);
less_than less4(y2, Y_MAX, l4);
less_than less5(x3, X_MAX, l5);
less_than less6(y3, Y_MAX, l6);



always@(posedge clk)
begin
if(start) state <= SETUP_START;
else
      begin
      case(state)
            SETUP_START: begin x1 <= v_buffer_out[12:0]; state <=
GET_Y1; end
            GET_Y1: begin y1<= v_buffer_out[12:0]; state <= GET_X2; end
            GET_X2: begin x2<= v_buffer_out[12:0]; state <= GET_Y2; end
            GET_Y2: begin y2<= v_buffer_out[12:0]; state <= GET_X3; end
            GET_X3: begin x3<= v_buffer_out[12:0]; state <= GET_Y3; end
            GET_Y3: begin y3<= v_buffer_out[12:0]; state <= FIRST_LINE;
end
```

```
      endcase

      if (g1) state <= IDLE;

      if(state == FIRST_LINE)
            begin
            // ix x and y are negative they will automatically be
discounted
            if((x1[12] ==1 & x2[12] == 1) |  //to the left
                  (y1[12] ==1 & y2[12] == 1) |  //on the top
                   (~l1 & ~l3 & x1[12] & x1[12] == 0 & x2[12] == 0) |
//to the right
                  (~l2 & ~l4 & y1[12] == 0 & y2[12] ==0))  // on the
bottom
                  begin
                  skip_1 <= 1;
                  state <= SECOND_LINE;
            end

            if(l1 & l2 & l3 & l4)
            begin
                  l1_x1 <= x1[8:0];
                  l1_y1 <= y1[7:0];
                  l1_x2 <= x2[8:0];
                  l1_y2 <= y2[7:0];
                  skip_1 <= 0;
                  state <=SECOND_LINE;
            end

            else if(l1 & l2)
                  begin
                  l1_x1 <= x1;
                  l1_y1 <= y1;
                  state <= TRIM_FIRST2;
                  trim_start <= 1;
            end

            else begin
                  state <= TRIM_FIRST1;
                  trim_start <= 1;
            end
      end   //state == FIRST_LINE


      if(state == TRIM_FIRST1 & trim_done)
            begin
            if(no_line)
                  begin
                  skip_1 <=1;
                  state <= SECOND_LINE;
            end
            else
                  begin
                  l1_x1 <= x_intersect;
                  l1_y1 <= y_intersect;
                  if(l3 & l4)
                        begin
```

```
                        state<=SECOND_LINE;
                        l1_x2 <= x2;
                        l1_y2 <= y2;
                  end
                  else begin
                        state <= TRIM_FIRST2;
                        trim_start <= 1;
                  end
            end
      end //state == TRIM_FIRST1 & trim_done

      if(state == TRIM_FIRST2 & trim_done)
            begin
            l1_x2 <= x_intersect;
            l1_y2 <= y_intersect;
            skip_1 <= 0;
            state <= SECOND_LINE;
      end //state == TRIM_FIRST2 & trim_done


      if(state == SECOND_LINE)
            begin
            // ix x and y are negative they will automatically be
discounted
            if((x2[12] ==1 & x3[12] == 1) |  //to the left
                  (y2[12] ==1 & y3[12] == 1) |  //on the top
                  (~l3 & ~l5 & x2[12] == 0 & x3[12] == 0) |     // to
the right
                  (~l4 & ~l6 & y2[12] == 0 & y3[12] == 0)) //on the
bottom
                  begin
                  skip_2 <= 1;
                  state <= THIRD_LINE;
            end
            if(l3 & l4 & l5 & l6)
                  begin
                  l2_x1 <= x2[8:0];
                  l2_y1 <= y2[7:0];
                  l2_x2 <= x3[8:0];
                  l2_y2 <= y3[7:0];
                  skip_2 <= 0;
                  state <=THIRD_LINE;
            end

            else if(l3 & l4)
                  begin
                  l2_x1 <= x2;
                  l2_y1 <= y2;
                  state <= TRIM_SECOND2;
                  trim_start <= 1;
            end

            else begin
                  state <= TRIM_SECOND1;
                  trim_start <= 1;
            end
      end   //state==FIRST_LINE
```

```
      if(state == TRIM_SECOND1 & trim_done)
            begin
            if(no_line)
                  begin
                  skip_2 <=1;
                  state <= THIRD_LINE;
            end
            else
                  begin
                  l2_x1 <= x_intersect;
                  l2_y1 <= y_intersect;
                  if(l5 & l6)
                        begin
                        state<=THIRD_LINE;
                        l2_x2 <= x3;
                        l2_y2 <= y3;
                  end
                  else begin
                        state <=TRIM_SECOND2;
                        trim_start <= 1;
                  end
            end
      end //state == TRIM_SECOND1 & trim_done

      if(state == TRIM_SECOND2 & trim_done)
            begin
            l2_x2 <= x_intersect;
            l2_y2 <= y_intersect;
            skip_2 <= 0;
            state <= THIRD_LINE;
      end //state == TRIM_SECOND2 & trim_done

      if(state == THIRD_LINE)
            begin
            // ix x and y are negative they will automatically be
discounted
            if((x3[12] ==1 & x1[12] == 1) |  //to the left
               (y3[12] ==1 & y1[12] == 1) |  //on the top
                  (~l5 & ~l1 & x3[12] == 0 & x1[12] == 0) |     // to
the right
                  (~l6 & ~l2 & y3[12] == 0 & y1[12] == 0)) //on the
bottom
                  begin
                  skip_1 <= 1;
                  state <= DRAW_PREP;
            end
            if(l5 & l6 & l1 & l2)
            begin
                  l3_x1 <= x3[8:0];
                  l3_y1 <= y3[7:0];
                  l3_x2 <= x1[8:0];
                  l3_y2 <= y1[7:0];
                  skip_3 <= 0;
                  state <=DRAW_PREP;
            end
```

```
            else if(l5 & l6)
                  begin
                  l3_x1 <= x3;
                  l3_y1 <= y3;
                  state <= TRIM_THIRD2;
                  trim_start <= 1;
            end

            else begin
                  state <= TRIM_THIRD1;
                  trim_start <= 1;
            end
      end   //state==THIRD_LINE

      if(state == TRIM_THIRD1 & trim_done)
            begin
            if(no_line)
                  begin
                  skip_3 <=1;
                  state <= DRAW_PREP;
            end
            else
                  begin
                  l3_x1 <= x_intersect;
                  l3_y1 <= y_intersect;
                  if(l1 & l2)
                        begin
                        state<=DRAW_PREP;
                        l3_x2 <= x1;
                        l3_y2 <= y1;
                  end
                  else begin
                        state <= TRIM_THIRD2;
                        trim_start <= 1;
                  end
            end
      end //state == TRIM_THIRD1 & trim_done

      if(state == TRIM_THIRD2 & trim_done)
            begin
            l3_x2 <= x_intersect;
            l3_y2 <= y_intersect;
            skip_3 <= 0;
            state <= DRAW_PREP;
      end //state == TRIM_THIRD2 & trim_done

      if(state == DRAW_PREP)
            begin
            if(skip_1 & skip_2 & skip_3) state <= IDLE;
            else if(skip_1 & skip_2) begin state <= DRAW_THIRD;
raster_start <= 1; end
            else if(skip_1) begin state <= DRAW_SECOND; raster_start <=
1; end
            else begin state <= DRAW_FIRST; raster_start <= 1; end
      end

      if(state == DRAW_FIRST & raster_done)
```

```
            begin
            if(skip_2 & skip_3) begin state <= IDLE; raster_start <= 1;
end
            else if(skip_2) begin state <= DRAW_THIRD; raster_start <=
1; end
            else begin  state<= DRAW_SECOND; raster_start<= 1; end
      end

      if(state == DRAW_SECOND & raster_done)
            begin
            if(skip_3) state<= IDLE;
            else begin state <= DRAW_THIRD; raster_start<= 1; end
      end

      if(state == DRAW_THIRD & raster_done)
      state <= IDLE;


      end // ends !start

if(raster_start) raster_start <=0;
if(trim_start) trim_start <= 0;

end // ends always block

assign done = (state == IDLE & ~start);
assign done_setup = (done | g2);
assign vb_point = (state == GET_Y1 | state == SETUP_START)? 0: (state
== GET_X2 | state == GET_Y2)? 1: 2;
assign vb_coord = (state == SETUP_START | state == GET_X2 | state ==
GET_X3)? 0:1;


assign x_to_trim =      (state == TRIM_FIRST2 | state == TRIM_SECOND1)?
x2:
                              (state == TRIM_SECOND2 | state ==
TRIM_THIRD1)? x3: x1;

assign x_end =    (state == TRIM_FIRST1 | state == TRIM_SECOND2)? x2:
                        (state == TRIM_FIRST2 | state == TRIM_THIRD1)?
x1: x3;


assign y_to_trim = (state == TRIM_FIRST2 | state == TRIM_SECOND1)? y2:
                              (state == TRIM_SECOND2 | state ==
TRIM_THIRD1)? y3: y1;


assign y_end =    (state == TRIM_FIRST1 | state == TRIM_SECOND2)? y2:
                        (state == TRIM_FIRST2 | state == TRIM_THIRD1)?
y1: y3;

assign x_coord0 = (state == DRAW_FIRST)? l1_x1 :
                              (state == DRAW_SECOND)? l2_x1: l3_x1;
assign x_coord1 = (state == DRAW_FIRST)? l1_x2 :
                              (state == DRAW_SECOND)? l2_x2: l3_x2;
assign y_coord0 = (state == DRAW_FIRST)? l1_y1 :
```

```
                              (state == DRAW_SECOND)? l2_y1: l3_y1;
assign y_coord1 = (state == DRAW_FIRST)? l1_y2 :
                              (state == DRAW_SECOND)? l2_y2: l3_y2;

endmodule
```

## Plotter Code

```
module plotter(clk, start, clear, done, x_coord, y_coord, ram_we,
ram_addr, ram_value, color);
input clk, start, clear;
input[1:0] color;
output done, ram_we;
input[8:0] x_coord;
input[7:0] y_coord;
output[16:0] ram_addr;
output[1:0] ram_value;

reg[3:0] state;
parameter IDLE =0;
parameter SETUP = 1;
parameter WRITE_RAM1 = 2;
parameter WRITE_RAM2 = 3;
parameter WRITE_RAM3 = 4;
parameter WRITE_RAM4 = 5;
parameter WRITE_RAM5 = 6;
parameter CLEAR1 = 7;
parameter CLEAR_WRITE = 8;
parameter CLEAR_POST = 9;

reg[16:0] ram_addr;
reg[1:0] ram_value;

parameter X_MAX = 320;
parameter Y_MAX = 240;
//reg[16:0] clear_addr;

wire[16:0] temp_adr;
assign temp_adr = x_coord + (y_coord<<8) + (y_coord<<6);

reg ram_we;


always@(posedge clk) begin
      if(start) begin
            state <= SETUP;
            //ram_addr<= (temp_adr < X_MAX * Y_MAX )? temp_adr : 0;
            ram_addr <= temp_adr;
            ram_value <= color;
      end else if(clear) begin
            state<= CLEAR1;
            ram_addr <= 0;
            ram_value <= 2'b00;
      end else begin
```

```
            case(state)
                  SETUP: begin      state <= WRITE_RAM1;    ram_we <=
1;    end
                  WRITE_RAM1: state<= WRITE_RAM2;
                  WRITE_RAM2: state<= WRITE_RAM3;
                  WRITE_RAM3: state<= WRITE_RAM4;
                  WRITE_RAM4: state<= WRITE_RAM5;
                  WRITE_RAM5: begin state <= IDLE;    ram_we <= 0;
      end

                  CLEAR1: begin state <= CLEAR_POST; ram_we<= 1; end
                  //WAIT:     begin state <= CLEAR_WRITE;   end
                  //CLEAR_WRITE: begin state <= CLEAR_POST; end
                  CLEAR_POST: if( ram_addr == 76799) begin
                                          state <= IDLE;    ram_we <=
0;
                                    end else begin
                                          ram_addr <= ram_addr + 1;
      state <= CLEAR1;
                                    end
                  default: state <= IDLE;
            endcase
      end
end

assign done = (state == IDLE) & ~start & ~clear;


endmodule
```

## Line Draw Module Code

```
module line_draw(clk, x_coord0, y_coord0, x_coord1, y_coord1, x_out,
y_out, plotter_start, plotter_done, start, done, dec, g1);
input clk, plotter_done, start;
output done, plotter_start;
input[8:0] x_coord0, x_coord1; //these range from 0 to 320
input[7:0] y_coord0, y_coord1; //these range from 0 to 240
output[8:0] x_out; //0 to 320
output[7:0] y_out; //0 to 240
output[10:0] dec;
output g1;
reg [8:0] x0, x1;
reg [7:0] y0, y1;
reg [8:0] dx;
reg [7:0] dy;
reg [10:0] dec;

reg[1:0] sx;
reg sy;

reg[2:0] state;
parameter IDLE = 0;
parameter SET_XY =  1;
parameter ASSIGN_VARS1 = 2;
parameter ASSIGN_VARS2 = 3;
parameter LOOP_DRAW_START = 4;
parameter LOOP_DRAW = 5;
parameter LOOP_INC_VARS = 6;


wire g2, g3, g4, l1;
greater_than gr1(y_coord0, y_coord1, g1);
greater_than gr2(x1, x0, g2);
greater_than gr3(y1, y0, g3);
greater_than gr4(dy, dx, g4);
less_than less1(dy, dx, l1);

always@(posedge clk)
begin
if(start) state<= SET_XY;
else
begin

case (state)
 SET_XY:          state <= ASSIGN_VARS1;
 ASSIGN_VARS1:    state <= ASSIGN_VARS2;
 ASSIGN_VARS2:    state <= LOOP_DRAW_START;
 LOOP_DRAW_START:       state <= LOOP_DRAW;
```

```
 LOOP_DRAW:             state<= (plotter_done)? LOOP_INC_VARS:
LOOP_DRAW;
 LOOP_INC_VARS:   state<= LOOP_DRAW_START;
 default: state <= IDLE;
endcase

if(state == SET_XY)
begin
 if (g1) //flips around vertices to guaranty that

                  //lines are drawn in same direction each time
      begin
      x0 <= x_coord1;
      x1 <= x_coord0;
      y0 <= y_coord1;
      y1 <= y_coord0;

      end
  else
        begin
       x0 <= x_coord0;
      x1 <= x_coord1;
      y0 <= y_coord0;
      y1 <= y_coord1;
  end
end  //ends if state == SET_XY

if(state == ASSIGN_VARS1) //assigns the dx, dy, sy, sx
begin
dx[8:0] <= (g2)? x1-x0: x0-x1; //positive
dy[7:0] <= y1[7:0] - y0[7:0]; //positive

sx <= (g2)? 2'b01 : (x1 == x0)? 2'b00 : 2'b11;  //1, 0, or -1
sy <= (y1==y0)? 0: 1; //1 or 0
end //ends if state==ASSIGN_VARS1

if(state == ASSIGN_VARS2)
begin
if(g4) dec[10:0] <= {2'b00, dy[7:0], 1'b0} + ~{2'b00, dx[8:0]} +1;
//dec is in 2's complement format
else dec[10:0] <= {1'b0, dx[8:0], 1'b0} + ~{3'b000, dy[7:0]} +1;
end //ends state == ASSIGN_VARS2

if (state==LOOP_INC_VARS) //increments the variables in the loop
begin
if(l1)
begin
  if(x0 == x1) state<= IDLE; //break out of the loop
  else
      begin
            if(dec[10] == 0) //dec is positive
            begin
                  dec <= dec + ~{1'b0, dx[8:0], 1'b0} +1 + {2'b00,
dy[7:0], 1'b0};
                  y0 <= y0 + sy;
            end //ends if decy>0
            else  dec[10:0] <= dec[10:0] + {2'b00, dy[7:0], 1'b0};
```

```
            x0 <= (sx == 2'b01)? x0 + 1: (sx == 2'b11)? x0 - 1: x0;
      end //ends x0 != x1
end //ends if dy<dx

else
begin
  if(y1 == y0) state<= IDLE; // break out of the loop
  else
      begin
      if(dec[10] == 0) //dec is positive
      begin
            dec <= dec + ~{2'b00, dy[7:0], 1'b0} +1 + {1'b0, dx[8:0],
1'b0};
            x0 <= (sx == 2'b01)? x0 + 1: (sx == 2'b11)? x0 -1: x0;
      end //ends if dec>0
      else dec[10:0] <= dec[10:0] + {1'b0, dx[8:0], 1'b0};
      y0 <= y0 + sy;
      end // ends y1!= y0
end //ends dy >= dx

end // state is LOOP_INC_VARS

end //ends !start
end //ends always statement

assign done = (state == IDLE & ~start);
assign plotter_start = (state == LOOP_DRAW_START);
assign x_out = x0;
assign y_out = y0;

endmodule
```

## Input Module Code

```
module input_device(clk, reset, out_shake, out_nod, out_forward,
out_back, latch, pulse, data,done);

input clk, reset, data;
output latch, pulse, done;

output[10:0] out_nod, out_shake;
output out_forward, out_back;

parameter SIX_us = 72;
parameter TWELVE_us = 144;

parameter LATCH_STATE = 1;
parameter GET_A = 2;
parameter PULSE_B = 3;
parameter GET_B = 4;
parameter PULSE_SEL = 5;
parameter GET_SEL = 6;
parameter PULSE_START = 7;
parameter GET_START = 8;
parameter PULSE_UP = 9;
parameter GET_UP = 10;
parameter PULSE_DOWN = 11;
parameter GET_DOWN = 12;
parameter PULSE_LEFT = 13;
parameter GET_LEFT =14;
parameter PULSE_RIGHT =15;
parameter GET_RIGHT = 16;
parameter IDLE = 0;

reg[4:0] state;
reg[10:0] out_nod, out_shake;
reg out_forward, out_back;
reg[7:0] counter;

reg count_6;
reg count_12;
reg latch, pulse;

always@(posedge clk)
begin
      if(counter == TWELVE_us) begin
            count_12 <= 1;
            count_6 <= 1;
            counter <= 0;
```

```
end
else if(counter == SIX_us) begin
      count_6 <= 1;
      counter <= counter + 1;
end
else counter <= counter +1;


if(count_6) count_6 <= 0;
if(count_12) count_12 <= 0;

if(reset)
      begin
      out_nod <= 0;
      out_shake <= 0;
      out_forward <=0;
      out_back <= 0;
      counter <= 0;
      latch <= 0;
      pulse <= 0;
      state <= LATCH_STATE;
end

else if(state == LATCH_STATE)
      begin
      latch <= 1;
      if(count_12) state <= GET_A;
end

else if(state == GET_A)
      begin
      latch <= 0;
      if(count_6)
            begin
            if(~data) out_forward <= 1;
            state <= PULSE_B;
      end
end

else if(state == PULSE_B)
      begin
      pulse <= 1;
      if(count_6) state <= GET_B;
end

else if(state == GET_B)
      begin
      pulse <= 0;
      if(count_6)
            begin
            if(~data) out_back <= 1;
            state <= PULSE_SEL;
      end
end

else if(state == PULSE_SEL)
      begin
```

```
      pulse <= 1;
      if(count_6) state <= GET_SEL;
end

else if(state == GET_SEL)
      begin
      pulse <= 0;
      if(count_6)
            begin
            state <= PULSE_START;
      end
end

else if(state == PULSE_START)
      begin
      pulse <= 1;
      if(count_6) state <= GET_START;
 end

else if(state == GET_START)
      begin
      pulse <= 0;
      if(count_6)
            begin
            state <= PULSE_UP;
      end
end

else if(state == PULSE_UP)
      begin
      pulse <= 1;
      if(count_6) state <= GET_UP;
end

else if(state == GET_UP)
      begin
      pulse <= 0;
      if(count_6)
            begin
            if(~data) out_nod <= out_nod + 1;
            state <= PULSE_DOWN;
      end
end

else if(state == PULSE_DOWN)
      begin
      pulse <= 1;
      if(count_6) state <= GET_DOWN;
end

else if(state == GET_DOWN)
      begin
      pulse <= 0;
      if(count_6)
            begin
            if(~data) out_nod <= out_nod - 1;
            state <= PULSE_LEFT;
```

```
            end
      end

      else if(state == PULSE_LEFT)
            begin
            pulse <= 1;
            if(count_6) state <= GET_LEFT;
      end

      else if(state == GET_LEFT)
            begin
            pulse <= 0;
            if(count_6)
                  begin
                  if(~data) out_shake <= out_shake - 1;
                  state <= PULSE_RIGHT;
            end
      end

      else if(state == PULSE_RIGHT)
            begin
            pulse <= 1;
            if(count_6) state <= GET_RIGHT;
      end

      else if(state == GET_RIGHT)
            begin
            pulse <= 0;
            if(count_6)
                  begin
                  if(~data) out_shake <= out_shake + 1;
                  state <= IDLE;
            end
      end

      else state <= IDLE;

end

      assign done = (state==IDLE) & ~reset;



endmodule
```

# Code Implemented by Elliott

## MAC

```
module MAC(clk,a17,b17,clear,start,done,z17,overflow,clear_overflow);
	/*
		Contract: Do not send start signal when busy
	*/

	input clk,clear,start;		output done,overflow;		input clear_overflow;
	input [16:0] a17, b17;		output [16:0] z17;

	/*
		Note: a17 and b17 are inputs only, but are marked inout b/c they
			are tri-stated

		This multiplies a*b and adds it to a sum. Done goes high
		when z17 is stable. Clear sets the sum back to zero and sets the
		state back to idle. If you send high on clear and start at the
		same time, it will begin normal operation from a zero total.

		All commands must be single clock pulses. a17 and b17 must be
		held from start until done goes high.

		SPEED: This is pipelined (multiply and then add in two stages)
				which eliminates 15ns from the addition stage, and
				places the burden entirely on the multiply

			Each cycle requires the PD of the multiplier_mux.
			(PD of the adder is 15 ns, or 66Mhz speed)
	*/

	//registers
	reg [2:0] state_reg;		reg [16:0] total_reg;		reg [1:0] sel_reg; reg overflow_reg;
	reg [16:0] last_product17_reg;

	//outputs
	assign z17 = total_reg;		assign overflow = overflow_reg;

	//multiplier
	wire [17:0] last_product17;		wire mul_overflow;
	mul17_mux theMul17_mux( sel_reg, a17, b17, last_product17, mul_overflow );

	//adder (adds last product from reg to this total)
	wire [16:0] sum17;		wire add_overflow;
	adder17 theAdder17( total_reg, last_product17_reg, sum17, add_overflow );

	parameter S_IDLE = 0;		parameter S_ACCUM_1 = 1;		parameter S_ACCUM_2
= 2;
	parameter S_ACCUM_3 = 3;		parameter S_ACCUM_4 = 4;

	//note: on 'start', we can always assume that sel_reg is zero, so we can use the first mult.
```

```
//also, note last_product17_reg always gets overwritten on start
always @ (posedge clk) begin

	if( clear_overflow ) overflow_reg <= 0;

	if( clear & ~start) begin
		total_reg <= 0;	sel_reg <= 0;
		state_reg <= S_IDLE;
	end

	if( clear & start) begin
		total_reg <= 0;
		overflow_reg <= mul_overflow;
	end

	if( start & ~clear) begin
		overflow_reg <= overflow_reg | mul_overflow;
	end

	if( start ) begin
		sel_reg <= 1;
		last_product17_reg <= last_product17;
		state_reg <= S_ACCUM_1;
	end

	if( ~clear & ~start ) begin
		case(state_reg)
			S_ACCUM_1: begin
				total_reg <= sum17;
				last_product17_reg <= last_product17;
				overflow_reg <= overflow_reg | mul_overflow |
add_overflow;
				sel_reg <= 2;
				state_reg <= S_ACCUM_2;
			end
			S_ACCUM_2: begin
				total_reg <= sum17;
				last_product17_reg <= last_product17;
				overflow_reg <= overflow_reg | mul_overflow |
add_overflow;
				sel_reg <= 3;
				state_reg <= S_ACCUM_3;
			end
			S_ACCUM_3: begin
				total_reg <= sum17;
				last_product17_reg <= last_product17;
				overflow_reg <= overflow_reg| mul_overflow | add_overflow;
				sel_reg <= 0;
				state_reg <= S_ACCUM_4;
			end
			S_ACCUM_4: begin
				total_reg <= sum17;
				overflow_reg <= overflow_reg | add_overflow;
				sel_reg <= 0;
				state_reg <= S_IDLE;
			end
```

```
                                        default: state_reg <= S_IDLE;
                                endcase
                        end

        end //posedge clk

        assign done = ( state_reg == S_IDLE ) & ~start & ~clear;

endmodule

module mul17_mux(sel2, a17, b17, z17, overflow);

        input [1:0] sel2;   input [16:0] a17,b17;
        output [16:0] z17;              output overflow;

        //chooses appropriate bits from a17,b17 to produce a8,b8, which are multiplied
                //to produce a partial z17 product
        /*
                sel: 00 -> a[15:8]*b[15:8] = z[7:0] (H*H)
                                01 -> a[15:8]* b[7:0] = z[12:0] (H*L)
                                10 -> a[7:0] *b[15:8] = z[12:0] (L*H)
                                11 -> a[7:0] * b[7:0] = z[15:8] (L*L)
        */
        wire [7:0] a8, b8; wire [15:0] z16;
        mul8b theMul8b( a8, b8, z16 );

        assign a8 = (sel2[0] == 0) ? a17[15:8] : a17[7:0];
        assign b8 = (sel2[1] == 0) ? b17[15:8] : b17[7:0];

        assign z17[16] = a17[16] ^ b17[16];             //sign bit

        assign z17[15:0] = (sel2 == 0) ? z16[7:0]<<8 : //low 8 bits to high 8 bits
                                        (sel2 == 3) ? z16[15:8] : //high 8 bits to low 8 bits
                                        z16[12:0]; //middle bits to middle bits

        assign overflow = (sel2 == 0) ? (z16[15:8] > 0) : 0; //only H*H can overflow

endmodule


module adder17(a17,b17,z17,overflow);

        input [16:0] a17,b17;       output [16:0] z17;              output overflow;

        /*
                adds two 17-bit sign magnitude #s
                pure combinational logic. delay for stability is 15 nanoseconds (~66mhz clock)

                If there's an overflow, the number will wrap-around and overflow will be high
        */

        //first, create 2's complement versions of each
        wire [16:0] flipa, flipb, comp_neg_a, comp_neg_b, compa, compb;

        //flip bits
        assign flipa = {1'b1,~a17[15:0]};    assign flipb = {1'b1,~b17[15:0]};
```

```
	//add one
	add1 add1FlipA( flipa, comp_neg_a );
	add1 add1FlipB( flipb, comp_neg_b );

	//choose between positive and negative representation
	assign compa = a17[16] ? comp_neg_a : a17;
	assign compb = b17[16] ? comp_neg_b : b17;

	//now, perform the addition
	wire [16:0] out_comp;
	add17 theAdder( compa, compb, out_comp );

	//assign x = out_comp;

	//convert back to sign magnitude form
	wire [16:0] out_minus_one, out_negative;
	sub1 theSub1( out_comp, out_minus_one );

	assign out_negative = {1'b1, ~out_minus_one[15:0]};

	//choose positive or negative for actual output
	assign z17 = out_comp[16] ? out_negative : out_comp;

	assign overflow = (a17[16] & b17[16] & ~z17[16]) | //neg+neg=pos
	(~a17[16] & ~b17[16] & z17[16]); //pos+pos=neg

endmodule
```

## TRIG

```
module Trig(clk, angle12, cosine, reset, start, done, z17,
	mac_a_bus, mac_b_bus, mac_clear_or, mac_start_or, mac_done, mac_out);

	/*
		Contract: Do not send start signal when busy
		Do not drive MAC bus when using
		MAC can be driven again externally exactly when done goes high

		Input angle is from 0 to 7FF (11 bits)
		0 to 11'h800 --> 6.48/800 == 17'000C9 is scale factor

		cosine==1 for cosine, 0 for sine

		use expansions (these are very accurate for 0 to pi/4)
			sinx = x - x^2/6
			cosx = 1 - x^2/2 + x^4/24

		Note: we actually could use sinx expansion for both sin and cos, but we need the
		cos expansion in certain cases
		to increase accuracy for certain ranges of sine

		Time/Accuracy:
			Approx 80 cycles to calculate
			Error rate about +-0.004
	*/
```

```
    input clk, reset, start;        input [10:0] angle12;       output [16:0] z17;
    output done;        input cosine;

    //standard use of the mac-bus
    input mac_done;     input [16:0] mac_out;       output [16:0] mac_a_bus, mac_b_bus;
                                                                                output
mac_clear_or, mac_start_or;


    parameter SCALE = 17'h000C9;

    parameter PI = 17'h00324;                       parameter TWO_PI = 17'h00648;
    parameter HALF_PI = 17'h00192;                  parameter FOURTH_PI = 17'h000C9;

    parameter NEG_ONE_SIXTH = 17'h1002B;        parameter NEG_ONE_HALF =  17'h10080;
    parameter POS_ONE_24TH = 17'h0000B;             parameter NEG_ONE = 17'h10100;
    parameter ONE = 17'h00100;                      parameter ONE_HALF = 17'h00080;


    parameter S_IDLE = 0;                               parameter S_CHECK_COS = 1;
    parameter S_CONV_COS = 2;                           parameter S_LIMIT_POS = 3;
    parameter S_LIMIT_PI = 4;                       parameter S_LIMIT_HALF_PI = 5;
    parameter S_LIMIT_FOURTH_PI = 6;            parameter S_CALC_SIN_1 = 7;
    parameter S_CALC_SIN_2 = 8;                         parameter S_CALC_SIN_3 = 9;
    parameter S_CALC_SIN_4 = 10;                    parameter S_CALC_COS_1 = 11;
    parameter S_CALC_COS_2 = 12;                    parameter S_CALC_COS_3 = 13;
    parameter S_CALC_COS_4 = 14;                    parameter S_CALC_COS_5 = 15;
    parameter S_CALC_FINISH = 16;


    reg [4:0] state;        reg [16:0] mac_a, mac_b;    reg mac_start;      reg mac_clear;
    reg flip_sign_result;       reg [16:0] z17;     reg [10:0] N;


    always @ (posedge clk) begin
        if( reset ) begin
            state <= S_IDLE;
        end else if (start) begin
            mac_a <= angle12; mac_b <= SCALE; mac_start <= 1;     mac_clear <= 1;
            state <= S_CHECK_COS;
        end else begin
            case(state)
                S_CHECK_COS: if( mac_done ) begin
                    if( cosine ) begin
                        mac_a <= mac_out;mac_b <= NEG_ONE;          mac_start
<= 1;   mac_clear <= 1;
                        state <= S_CONV_COS;
                    end else state <= S_LIMIT_POS;
                end
                S_CONV_COS: if( mac_done) begin
                    mac_a <= HALF_PI;           mac_b <= ONE;     mac_start <= 1;
                    state <= S_LIMIT_POS;
                end
                S_LIMIT_POS: if(mac_done) begin
                    if( mac_out[16] ) N <= TWO_PI - mac_out[10:0];
                    else N <= mac_out[10:0];
                    state <= S_LIMIT_PI;
                end
                S_LIMIT_PI: begin
                    if( N >= PI[10:0] ) begin       N <= N - PI;        flip_sign_result <=
1;      end
```

```
                                        else        flip_sign_result <= 0;
                                        state <= S_LIMIT_HALF_PI;
                                end
                                S_LIMIT_HALF_PI: begin
                                        if( N >= HALF_PI[10:0] )      N <= PI - N;
                                        state <= S_LIMIT_FOURTH_PI;
                                end
                                S_LIMIT_FOURTH_PI: begin
                                        if( N >= FOURTH_PI[10:0]) begin N <= HALF_PI[10:0] - N; state
<= S_CALC_COS_1; end
                                        else state <= S_CALC_SIN_1;
                                end

                                S_CALC_SIN_1: begin
                                        mac_a <= N;          mac_b <= N;          mac_start <= 1;
        mac_clear <= 1;     //x2 is macout
                                        state <= S_CALC_SIN_2;
                                end
                                S_CALC_SIN_2: if( mac_done) begin
                                        mac_a <= mac_out;mac_b <= N;          mac_start <= 1;
        mac_clear <= 1;     //x3 is macout
                                        state <= S_CALC_SIN_3;
                                end
                                S_CALC_SIN_3: if( mac_done) begin
                                        mac_a <= mac_out;mac_b <= NEG_ONE_SIXTH;              mac_start
<= 1;       mac_clear <= 1; //-x3/6 is macout
                                        state <= S_CALC_SIN_4;
                                end
                                S_CALC_SIN_4: if( mac_done ) begin
                                        mac_a <= N;          mac_b <= ONE;     mac_start <= 1;       //-x3/6 +
x is macout
                                        state <= S_CALC_FINISH;
                                end

                                S_CALC_COS_1: begin
                                        mac_a <= N;          mac_b <= N;          mac_start <= 1;
        mac_clear <= 1;     //x2 is macout
                                        state <= S_CALC_COS_2;
                                end
                                S_CALC_COS_2: if( mac_done) begin
                                        mac_a <= mac_out;mac_b <= mac_out;              mac_start <= 1;
        mac_clear <= 1; //x4 is macout
                                        z17 <= mac_out;    //z17 is storing x2 temporarily
                                        state <= S_CALC_COS_3;
                                end
                                S_CALC_COS_3: if( mac_done) begin
                                        mac_a <= mac_out;mac_b <= POS_ONE_24TH; mac_start <= 1;
        mac_clear <= 1 ; //x4/5! is macout
                                        state <= S_CALC_COS_4;
                                end
                                S_CALC_COS_4: if(mac_done) begin
                                        mac_a <= z17;       mac_b <= NEG_ONE_HALF;              mac_start
<= 1;       //x4/5! - x2/2 is macout
                                        state <= S_CALC_COS_5;
                                end
                                S_CALC_COS_5: if( mac_done) begin
                                        mac_a <= ONE;     mac_b <= ONE;     mac_start <= 1;       //x4/5! -
x2/2 + 1 is macout
                                        state <= S_CALC_FINISH;
                                end

                                S_CALC_FINISH: if( mac_done) begin
```

```
                                        z17 <= {mac_out[16] ^ flip_sign_result,mac_out[15:0]};
                                        state <= S_IDLE;
                                end
                        endcase
                end

                if( mac_clear ) mac_clear <= 0;                 if( mac_start ) mac_start <= 0;//level to pulse
conversion
        end

        assign done = (state == S_IDLE) & ~start & ~reset;

        assign mac_clear_or = mac_clear;        assign mac_start_or = mac_start;

        assign mac_a_bus =  mac_a;
        assign mac_b_bus =  mac_b;

endmodule
```

## Matrix Generation

```
module MatrixGen(clk, start, done, reset, a, b, c, op,
                                        mb_row, mb_col, mb_we, mb_in,
                                        trig_done, trig_start, trig_cosine_enable, trig_out);

        input trig_done;        output trig_start, trig_cosine_enable;  input [16:0] trig_out;

        input clk, start, reset;        output done; input [1:0] op;
        input [16:0] a,b,c;     //angle is in a, [tx,ty,tz] is in a,b,c

        //standard use of matrix-buffer bus
        output [1:0] mb_row, mb_col;    output mb_we;           output [16:0]           mb_in;

        reg trig_start;
        reg trig_cosine_enable;
        reg sign_flip_trig;

        reg [2:0] state;        reg [1:0] mb_row, mb_col;       reg [16:0] mb_in;       reg mb_we;

        //reg [16:0] cos,sin;

        parameter OP_TRANS = 0;         parameter OP_ROT_NOD = 1;       parameter OP_ROT_SHAKE = 2;

        parameter S_IDLE = 0;   parameter S_VAL = 1;    parameter S_WAIT_TRIG = 2;
        parameter S_WE_HIGH = 3;        parameter S_WE_2 = 4;   parameter S_WE_3 = 5;
        parameter S_WE_LOW = 6;         parameter S_ADR = 7;


        /*
                All angles are in counter-clockwise format.

                Trans:                                                          Rot-Nod:
        Rot-Shake:
                                [1      0       0       tx]                                     [1
        0       0       0]                      [c      0       s       0]
                                [0      1       0       ty]                                     [0
        c       -s      0]                      [0      1       0       0]
                                [0      0       1       tz]                                     [0
        s       c       0]                      [-s     0       c       0]
        */

        always @ (posedge clk) begin
                if( reset ) begin       state <= S_IDLE;        mb_we <= 0;             end
                else if( start ) begin
                        mb_row <= 0;            mb_col <= 0;            state <= S_VAL;         mb_we <=0;
                end else begin
                        case( state )
```

```
S_VAL: begin
        case( op)
            OP_TRANS: begin
                if( mb_col == mb_row ) mb_in <= 17'h00100;
                else if( mb_col == 3) begin
                    if( mb_row == 0)        mb_in <= a;
                    else if( mb_row == 1) mb_in <= b;
                    else mb_in <= c;
                end else mb_in <= 0;
                state <= S_WE_HIGH;
            end

            OP_ROT_NOD:     begin
                if( (mb_row == 0)&(mb_col==0)) begin            mb_in <=
17'h00100;   state <= S_WE_HIGH;    end
                else if( mb_row == mb_col) begin
                    sign_flip_trig <= 0; trig_cosine_enable <= 1;
trig_start <= 1; state <= S_WAIT_TRIG;
                end else if( ((mb_row==2)&(mb_col==1)) ) begin
                    sign_flip_trig <= 0; trig_cosine_enable <= 0;
trig_start <= 1; state <= S_WAIT_TRIG;
                end else if( ((mb_row==1)&(mb_col==2)) ) begin
                    sign_flip_trig <= 1; trig_cosine_enable <= 0;
trig_start <= 1; state <= S_WAIT_TRIG;
                end else begin
                    mb_in <= 0; state <= S_WE_HIGH;
                end
            end

            OP_ROT_SHAKE:   begin
                if( (mb_row==1)&(mb_col==1) ) begin             mb_in <=
17'h00100;   state <= S_WE_HIGH;    end
                else if( mb_row == mb_col) begin
                    sign_flip_trig <= 0; trig_cosine_enable <= 1;
trig_start <= 1;            state <= S_WAIT_TRIG;
                end else if( ((mb_row==2)&(mb_col==0)) ) begin
                    sign_flip_trig <= 0; trig_cosine_enable <= 0;
trig_start <= 1; state <= S_WAIT_TRIG;
                end else if( ((mb_row==0)&(mb_col==2)) ) begin
                    sign_flip_trig <= 1; trig_cosine_enable <= 0;
trig_start <= 1; state <= S_WAIT_TRIG;
                end else begin
                    mb_in <= 0; state <= S_WE_HIGH;
                end
            end
        endcase
end

S_WAIT_TRIG: if( trig_done ) begin
        mb_in <= {trig_out[16]^sign_flip_trig, trig_out[15:0] };
        state <= S_WE_HIGH;
end

S_WE_HIGH: begin    mb_we <= 1;             state <= S_WE_2;        end

S_WE_2:     state <= S_WE_3;

S_WE_3: state <= S_WE_LOW;

S_WE_LOW: begin             mb_we <= 0;             state <= S_ADR;         end

S_ADR: begin
        if( mb_col == 3 ) begin
            if( mb_row == 2 ) begin
                state <= S_IDLE;
            end else begin
                mb_row <= mb_row + 1; mb_col <= 0;
                state <= S_VAL;
            end
        end else begin
            mb_col <= mb_col + 1;
            state <= S_VAL;
        end
end
default: state <= S_IDLE;
    endcase
end
if( trig_start ) trig_start <= 0;           //level to pulse
end
```

```
	assign done = (state == S_IDLE) & ~reset & ~start;

endmodule
```

## Matrix Multiplication

```
module MatrixMul(clk, start, done, reset, mat_a, mat_b,
                                        mb_mat, mb_row, mb_col, mb_out, mb_in, mb_we,
                                        mac_a_bus, mac_b_bus, mac_clear, mac_start, mac_done, mac_out);

	input clk, start, reset;	output done;	input [1:0] mat_a, mat_b;

	//mac-bus (tris)
	input [16:0] mac_out;
	output mac_clear, mac_start;		input mac_done;
	output [16:0] mac_a_bus, mac_b_bus;

	//matrix-buf (muxed)
	output [1:0] mb_mat, mb_row, mb_col;	input [16:0] mb_out; output [16:0] mb_in;
	output mb_we;

	wire [1:0] result_mat;

	reg [3:0] state;	reg [16:0] mac_a;reg mac_clear, mac_start;
                                                reg [1:0] mb_mat;		reg [1:0] r_col, r_row,
mb_col, mb_row;
                                                reg [1:0] iter;		reg mb_we;

	parameter S_IDLE = 0;	parameter S_FETCH_A = 1;	parameter S_FETCH_B = 2;
	parameter S_MAC = 3;	parameter S_WRITE = 4;	parameter S_WE_HIGH = 5;
	parameter S_WE_2 = 6;	parameter S_WE_3 = 7;
	parameter S_WE_LOW = 8;		parameter S_ITER_DOT = 9;	parameter S_ADR = 10;

	always @(posedge clk) begin
		if( reset) begin
			state <= S_IDLE;r_col <= 0;		r_row <= 0;		mb_col <= 0;
	mb_row <= 0;
			mb_mat <= mat_a;		iter <= 0;		mb_we <= 0;	mac_clear <=0;
	mac_start <=0;	mac_a <= 0;
		end else if( start) begin
			state <= S_FETCH_A;	r_col <= 0;		r_row <= 0;		mb_col
<= 0;	mb_row <= 0;
			mb_mat <= mat_a;		iter <= 0;		mb_we <= 0;	mac_clear <=0;
	mac_start <=0;	mac_a <= 0;
		end else begin
			case(state)
				S_FETCH_A: if( mac_done) begin
					mb_mat <= mat_a;
					mb_col <= iter;
					mb_row <= r_row;
					state <= S_FETCH_B;
				end

				S_FETCH_B: begin
```

```
                        mb_mat <= mat_b;
                        mb_col <= r_col;
                        mb_row <= iter;

                        mac_a <= mb_out;
                        state <= S_MAC;
                    end

                    S_MAC: begin
                        iter <= iter + 1;
                        if( iter == 0)      mac_clear <= 1;
                        mac_start <= 1;
                        if( iter == 3 ) state <= S_WRITE;
                        else state <= S_FETCH_A;
                    end

                    S_WRITE: if( mac_done) begin
                        mb_mat <= result_mat;
                        mb_col <= r_col; mb_row <= r_row;
                        state <= S_WE_HIGH;
                    end

                    S_WE_HIGH: begin
                        mb_we <= 1;
                        state <= S_WE_2;
                    end

                    S_WE_2:         state <= S_WE_3;
                    S_WE_3: state <= S_WE_LOW;

                    S_WE_LOW: begin
                        mb_we <= 0;
                        state <= S_ADR;
                    end

                    S_ADR: begin //increment address of result value
                        if( r_col == 3 ) begin
                            if( r_row == 2 ) begin
                                state <= S_IDLE;
                            end else begin
                                r_row <= r_row + 1;         r_col <= 0;
                                iter <= 0;
                                state <= S_FETCH_A;
                            end
                        end else begin
                            r_col <= r_col + 1;
                            iter <= 0;
                            state <= S_FETCH_A;
                        end
                    end
                    default: state <= S_IDLE;
                endcase
            end
            if( mac_clear ) mac_clear <= 0; //level to pulse
            if( mac_start ) mac_start <= 0;
end
```

```
	assign result_mat = ((mat_a==0)&(mat_b==1)) ? 2 :
	                    ((mat_a==0)&(mat_b==2)) ? 1 :
	                    ((mat_a==1)&(mat_b==0)) ? 2 :
	                    ((mat_a==1)&(mat_b==2)) ? 0 :
	                    ((mat_a==2)&(mat_b==0)) ? 1 :
	                     0 ; //2,1

	assign done = (state == S_IDLE) & ~reset & ~start;


	assign mac_a_bus =  mac_a;
	assign mac_b_bus =  mb_out;

	assign mb_in = mac_out;

endmodule
```

## Matrix/Vector Multiplication

```
module MatVec(clk, start, done, reset,
	vb_vert, vb_xyz, vb_we, vb_in, vb_out,
	mb_row, mb_col, mb_out,
	mac_a_bus, mac_b_bus, mac_clear, mac_start, mac_done, mac_out );

	input clk, start, reset;	output done;

	//vb bus (tri-state & or)
	output [1:0] vb_vert, vb_xyz; output vb_we;	output [16:0] vb_in;
	input [16:0] vb_out;

	//mac-bus (tri-state & or)
	output [16:0] mac_a_bus, mac_b_bus;	output mac_clear, mac_start; input mac_done;
	input [16:0] mac_out;

	//matrix-buf (muxed & or)
	output [1:0] mb_row, mb_col;	input [16:0] mb_out;

	parameter S_IDLE = 0;	parameter S_DOT = 2; parameter S_ADR = 3;
	parameter S_WRITE_VB = 4;	parameter S_VB_WE = 5;	parameter S_VB_WE_2 = 6;	parameter
S_VB_WE_3 = 7;
	parameter S_VB_WE_LOW = 8; parameter S_N_VERT = 9;

	reg [16:0] x,y,z;
	reg [3:0] state;	reg [1:0] mb_row, mb_col;
	reg mac_clear, mac_start;	reg [1:0] vert;	reg vb_we;

	always @(posedge clk) begin
		if( reset ) begin
			state <= S_IDLE;	mb_row <= 0;	mb_col <= 0;	mac_clear <= 0;	mac_start
<= 0; vb_we <= 0;
								vert <= 0;
		end else if( start ) begin
			state <= S_DOT;	mac_start <= 1;	mac_clear <= 1;	mb_row <= 0;	vert <= 0;
vb_we <= 0;
			mb_col <= 0;
		end else begin
```

```
                case( state )

                        S_DOT: if( mac_done ) begin
                                mb_col <= mb_col + 1;
                                if(mb_col == 3) state <= S_ADR;
                                else mac_start <= 1;
                        end

                        S_ADR: begin
                                mb_col <= 0;        mb_row <= mb_row + 1;
                                if( mb_row == 0) begin
                                        x <= mac_out;       state <= S_DOT;    mac_start <= 1;
mac_clear <= 1;
                                end else if( mb_row == 1) begin
                                        y <= mac_out;       state <= S_DOT;    mac_start <= 1;
mac_clear <= 1;
                                end else begin
                                        z <= mac_out;       state <= S_VB_WE;
                                end
                        end

                        S_VB_WE: begin   vb_we <= 1;          state <= S_VB_WE_2;          end

                        S_VB_WE_2:       begin      state <= S_VB_WE_3;          end

                        S_VB_WE_3: begin state <= S_VB_WE_LOW; end

                        S_VB_WE_LOW: begin       vb_we <= 0;          state <= S_WRITE_VB;
end



                        S_WRITE_VB: begin
                                mb_col <= mb_col + 1;
                                if( mb_col == 2) state <= S_N_VERT;
                                else state <= S_VB_WE;
                        end

                        S_N_VERT: begin
                                vert <= vert + 1;
                                if( vert == 2 ) beginstate <= S_IDLE;    end
                                else begin
                                        mb_col <= 0;        mb_row <= 0;
                                        mac_start <= 1;     mac_clear <= 1;
                                        state <= S_DOT;
                                end
                        end
                        default:    state <= S_IDLE;
                endcase
        end

        if( mac_clear ) mac_clear <= 0;          //level to pulse
        if( mac_start ) mac_start <= 0;
end

assign done = (state==S_IDLE) & ~reset & ~start;

assign mac_a_bus = mb_out;
assign mac_b_bus = vb_out;

assign vb_vert = vert;
assign vb_xyz =  mb_col;
```

```
	wire [16:0] vb_in;
	assign vb_in = (mb_col==0) ? x : (mb_col==1) ? y : z;

endmodule
```

## Math Module

```
module Math(clk, op, start, done, reset, val_a, val_b, val_c, mat_a_sel, mat_b_sel, out, overflow,
				vb_vert_ex, vb_xyz_ex, vb_we_ex, vb_in_ex, vb_out,
				mb_mat, mb_row, mb_col, mb_we, mb_out);
	//math i/o
	input clk,start,reset;input [2:0] op;	input [16:0] val_a, val_b, val_c;
	input [1:0] mat_a_sel, mat_b_sel;		output done, overflow;		output [16:0] out;

	//vb external
	input [1:0] vb_vert_ex, vb_xyz_ex;	input vb_we_ex;	input [16:0] vb_in_ex;		output [16:0]
vb_out;

	//vb internal
	wire [16:0] vb_in;	wire [1:0] vb_vert, vb_xyz;
	wire vb_we;
	//output [16:0] vb_in;		output [1:0] vb_vert, vb_xyz;
	//output vb_we;
	VertexBuffer theVertexBuffer( vb_vert, vb_xyz, vb_we, vb_in, vb_out );

	//matrix buffer
	output [1:0] mb_row, mb_col, mb_mat;
	output mb_we;	output [16:0] mb_out;		wire [16:0] mb_in;
	//output [1:0] mb_row, mb_col, mb_mat;
	//output mb_we;	output [16:0] mb_out;		wire [16:0] mb_in;
	MatrixBuffer theMatBuf( mb_mat, mb_row, mb_col, mb_we, mb_in, mb_out );

	//inverter
	wire start_inv, done_inv;	wire [16:0] inv_out;
	Inv theInv(clk, val_a, reset, start_inv, done_inv, inv_out);

	//mac
	wire [16:0] mac_a_bus, mac_b_bus;
	wire start_mac,clear_mac;
	wire done_mac;	wire [16:0] mac_out;
	MAC theMac(clk,mac_a_bus,mac_b_bus,clear_mac | reset,start_mac,done_mac,mac_out,overflow,start);

	//trig
	wire trig_cosine, trig_start, trig_done;	wire [16:0] trig_out;
	wire mac_clear_trig, mac_start_trig;
	wire [16:0] mac_a_bus_trig, mac_b_bus_trig;
	Trig theTrig(clk, val_a[10:0], trig_cosine, reset, trig_start, trig_done, trig_out,
				mac_a_bus_trig, mac_b_bus_trig, mac_clear_trig, mac_start_trig, done_mac, mac_out);

	//matrix generator
	wire start_gen, done_gen;	wire [1:0] op_gen;
	wire [1:0] mb_row_gen, mb_col_gen;	wire mb_we_gen;	wire [16:0] mb_in_gen;
	MatrixGen theMatGen(clk, start_gen, done_gen, reset, val_a, val_b, val_c, op_gen,
						mb_row_gen, mb_col_gen, mb_we_gen, mb_in_gen,
						trig_done, trig_start, trig_cosine, trig_out);

	//matrix/vector multiplier
```

```
wire start_matvec, done_macvec;
wire [1:0] vb_vert_matvec, vb_xyz_matvec;	wire vb_we_matvec;		wire [16:0] vb_in_matvec;
wire [1:0] mb_row_matvec, mb_col_matvec;	wire mac_start_matvec, mac_clear_matvec;
wire [16:0] mac_a_bus_matvec, mac_b_bus_matvec;
MatVec theMatVec(clk, start_matvec, done_macvec, reset,
		vb_vert_matvec, vb_xyz_matvec, vb_we_matvec, vb_in_matvec, vb_out,
		mb_row_matvec, mb_col_matvec, mb_out,
		mac_a_bus_matvec, mac_b_bus_matvec, mac_clear_matvec,
mac_start_matvec, done_mac, mac_out );

//matrix/matrix multiplier
wire start_matmul, done_matmul;
wire [1:0] mb_mat_matmul, mb_row_matmul, mb_col_matmul;
wire [16:0] mb_in_matmul;	wire mb_in_we;
wire mac_clear_matmul, mac_start_matmul;
wire [16:0] mac_a_bus_matmul, mac_b_bus_matmul;
MatrixMul theMatrixMul(clk, start_matmul, done_matmul, reset, mat_a_sel, mat_b_sel,
		mb_mat_matmul, mb_row_matmul, mb_col_matmul, mb_out, mb_in_matmul,
mb_we_matmul,
		mac_a_bus_matmul, mac_b_bus_matmul, mac_clear_matmul,
mac_start_matmul, done_mac, mac_out);



//control logic
parameter OP_INV = 0;	parameter OP_MUL = 1;	parameter OP_MUL_AC = 2;
parameter OP_GEN_TRANS = 3;	parameter OP_GEN_ROT_NOD = 4;	parameter
OP_GEN_ROT_SHAKE = 5;
parameter OP_MAT_MUL = 6;	parameter OP_MAT_VEC_MUL = 7;


parameter OP_TRANS = 0;	parameter OP_ROT_NOD = 1;	parameter OP_ROT_SHAKE = 2;
//matrix generator opcode
assign op_gen = (op==OP_GEN_TRANS) ? OP_TRANS :
		(op==OP_GEN_ROT_NOD) ? OP_ROT_NOD : OP_ROT_SHAKE;

//choose start
assign start_inv = start & (op == OP_INV);

assign start_mac = (start & ((op == OP_MUL) | (op == OP_MUL_AC)) ) | mac_start_matmul |
		mac_start_matvec | mac_start_trig;
assign clear_mac = (start & (op == OP_MUL)) | mac_clear_matmul | mac_clear_matvec | mac_clear_trig;

assign start_gen = (start & ((op == OP_GEN_TRANS) | (op == OP_GEN_ROT_NOD) |
		(op == OP_GEN_ROT_SHAKE)) );

assign start_matmul = (start & (op == OP_MAT_MUL) );

assign start_matvec = (start & (op == OP_MAT_VEC_MUL) );

//done
assign done = done_inv & done_mac & done_matmul & done_macvec & done_gen & ~start;

//choose output (either inverter or multiplier)
assign out = (op == OP_INV) ? inv_out : mac_out;

//finally, internal control signals for mb, and vb
assign vb_vert = (op==OP_MAT_VEC_MUL) ? vb_vert_matvec : vb_vert_ex;
assign vb_we = (op==OP_MAT_VEC_MUL) ? vb_we_matvec : vb_we_ex;
assign vb_xyz = (op==OP_MAT_VEC_MUL) ? vb_xyz_matvec : vb_xyz_ex;
assign vb_in = (op==OP_MAT_VEC_MUL) ? vb_in_matvec : vb_in_ex;
```

```
        assign mb_in = (op==OP_MAT_MUL) ? mb_in_matmul : mb_in_gen;
        assign mb_we = (op==OP_MAT_MUL) ? mb_we_matmul : mb_we_gen;

        assign mb_row = (op==OP_MAT_MUL) ? mb_row_matmul :
                                        (op==OP_MAT_VEC_MUL) ? mb_row_matvec :
                                        mb_row_gen;

        assign mb_col = (op==OP_MAT_MUL) ? mb_col_matmul :
                                        (op==OP_MAT_VEC_MUL) ? mb_col_matvec :
                                        mb_col_gen;

        assign mb_mat = (op==OP_MAT_MUL) ? mb_mat_matmul : mat_a_sel;

        wire choose_mac_bus;
        assign choose_mac_bus = ((op==OP_MUL) | (op==OP_MUL_AC));

        assign mac_a_bus = choose_mac_bus ? val_a :
                                        (op==OP_MAT_MUL) ? mac_a_bus_matmul :
                                                (op==OP_MAT_VEC_MUL) ? mac_a_bus_matvec :
                                                mac_a_bus_trig;


        assign mac_b_bus = choose_mac_bus ? val_b :
                                        (op==OP_MAT_MUL) ? mac_b_bus_matmul :
                                                (op==OP_MAT_VEC_MUL) ? mac_b_bus_matvec :
                                                mac_b_bus_trig;

endmodule
```

## Transform Apply

```
module ApplyTF(clk, start, reset, done,
                                tri_offset, tri_out,
                                vb_vert, vb_xyz, vb_we, vb_in,
                                math_op, math_start, math_done, mat_a_sel, math_overflow, skip);

        input clk, start, reset;        output done;        input math_overflow;
        input [16:0] tri_out;           output [3:0] tri_offset;        output skip;

        output [1:0] vb_vert;           output [1:0] vb_xyz;            output vb_we;
        output [16:0] vb_in;

        input math_done;    output math_start;   output [2:0] math_op;          output [1:0] mat_a_sel;

        reg [2:0] state;        reg [1:0] vb_vert, vb_xyz;      reg [3:0] tri_offset;
                                                                reg vb_we;          reg math_start;     reg skip;

        parameter OP_MAT_VEC_MUL = 7;

        //assign math_op = OP_MAT_VEC_MUL;
        reg [2:0] math_op;
        assign mat_a_sel = 0; //should contain the final transform (identity)

        parameter S_IDLE = 0;       parameter S_START = 1;      parameter S_WE_HIGH = 2;
        parameter S_WE_LOW = 3;     parameter S_ADR = 4;        parameter S_APPLY = 5;
        parameter S_FINISH = 6;

        always @(posedge clk) begin
                if( reset ) begin
```

```
                        state <= S_IDLE;    vb_we <= 0;         math_start <= 0;     skip <= 0;              math_op
<= 0;
                end else if( start ) begin
                                vb_vert <= 0;         vb_xyz <= 0;         tri_offset <= 0;       math_op <= 0;
                                vb_we <= 0;
                        skip <= 0;
                        state <= S_START; //do NOT begin at S_APPLY
                end else begin
                        case( state )
                                S_START: begin
                                        //let stuff get stable before we high
                                        state <= S_WE_HIGH;
                                end

                                S_WE_HIGH: begin
                                        vb_we <= 1;
                                        state <= S_WE_LOW;
                                end

                                S_WE_LOW: begin
                                        vb_we <= 0;
                                        state <= S_ADR;
                                end

                                S_ADR: begin
                                        tri_offset <= tri_offset + 1;
                                        if( vb_xyz == 2) begin vb_xyz <= 0; vb_vert <= vb_vert + 1; end
                                        else vb_xyz <= vb_xyz + 1;
                                        if( tri_offset == 8) begin state <= S_APPLY; math_op <=
OP_MAT_VEC_MUL; end
                                        else state <= S_WE_HIGH;
                                end

                                S_APPLY: begin
                                        math_start <= 1;
                                        state <= S_FINISH;
                                end

                                S_FINISH: if(math_done) begin //check for overflow
                                        skip <= math_overflow;
                                        state <= S_IDLE;    math_op <= 0;
                                end
                                default: state <= S_IDLE;
                        endcase
                end
                if(math_start) math_start <= 0;             //level to pulse
        end

        assign vb_in = tri_out;

        assign done = (state == S_IDLE) & ~start & ~reset;

endmodule
```

## Transform Compute

```
module ComputeTF(clk, start, done, reset, neg_pos_x, neg_pos_y, neg_pos_z, neg_angle_nod, neg_angle_shake,
                                        math_op, math_start, math_done, math_a, math_b, math_c,
                                        mat_a_sel, mat_b_sel);

        input clk, start, reset;        output done;

        input [16:0] neg_pos_x, neg_pos_y, neg_pos_z;   //optimize later
        input [10:0] neg_angle_nod, neg_angle_shake;

        output [2:0] math_op;           output math_start;   input math_done;
        output [16:0] math_a, math_b, math_c; output [1:0] mat_a_sel, mat_b_sel;

        parameter S_IDLE = 0;       parameter S_ROT_SHAKE = 1;          parameter S_ROT_NOD = 2;
        parameter S_MUL_ROT = 3; parameter S_TRANS = 4;     parameter S_CONCAT = 5;
        parameter S_FINAL = 6;

        parameter OP_GEN_TRANS = 3;       parameter OP_GEN_ROT_NOD = 4;   parameter
OP_GEN_ROT_SHAKE = 5;
        parameter OP_MAT_MUL = 6;

        reg [2:0] state;
        reg [2:0] math_op;
        reg math_start;
        reg sel_shake, sel_nod;
        reg [1:0] mat_a_sel, mat_b_sel;

        always @(posedge clk) begin
                if( reset) begin
                        math_start <= 0;    state <= S_IDLE;   math_op <= 0;       sel_shake <=0;
                        sel_nod <=0;        mat_a_sel <=0;     mat_b_sel <= 0;
                end else if( start) begin
                        state <= S_ROT_SHAKE;    math_op <= 0;       math_start <= 0;     sel_shake <=0;
                        sel_nod <=0;        mat_a_sel <=0;     mat_b_sel <= 0;
                end else begin
                        case( state )
                                S_ROT_SHAKE: begin
                                        math_op <= OP_GEN_ROT_SHAKE;
                                        sel_shake <= 1;     sel_nod <= 0;
                                        math_start <= 1;
                                        mat_a_sel <= 0;
                                        state <= S_ROT_NOD;
                                end

                                S_ROT_NOD: if(math_done) begin
                                        math_op <= OP_GEN_ROT_NOD;
                                        sel_shake <= 0;     sel_nod <= 1;
                                        math_start <= 1;
                                        mat_a_sel <= 1;
                                        state <= S_MUL_ROT;
                                end

                                S_MUL_ROT: if(math_done) begin
                                        math_op <= OP_MAT_MUL;
                                        mat_a_sel <= 0;     mat_b_sel <= 1;
                                        math_start <= 1;
                                        state <= S_TRANS;
                                end
```

```
                    S_TRANS: if(math_done) begin
                              math_op <= OP_GEN_TRANS;
                              mat_a_sel <= 1;     sel_shake <=0;       sel_nod <= 0;
                              math_start <= 1;
                              state <= S_CONCAT;
                    end

                    S_CONCAT: if(math_done) begin
                              math_op <= OP_MAT_MUL;
                              mat_a_sel <= 2;     mat_b_sel <= 1;
                              math_start <= 1;
                              state <= S_FINAL;
                    end

                    S_FINAL: if(math_done) state <= S_IDLE;


                    default:   state <= S_IDLE;
                 endcase
            end
            if( math_start ) math_start <= 0; //level to pulse
        end



        assign math_a = sel_shake ? neg_angle_shake :
                              sel_nod ? neg_angle_nod :
                              neg_pos_x;

        assign math_b = neg_pos_y;  assign math_c = neg_pos_z;

        assign done = (state == S_IDLE) & ~reset & ~start;

endmodule
```

## Projection

```
module Proj(clk, start, done, reset,
                    vb_vert, vb_xyz, vb_we, vb_in, vb_out,
                    math_op, math_start, math_done, math_overflow,
                    math_a, math_b, math_out,
                    skip);

        input clk, start, reset;        output done, skip;

        output [1:0] vb_vert;           output [1:0] vb_xyz;          output vb_we;
        output [16:0] vb_in;            input [16:0] vb_out;

        input math_overflow;
        input math_done;  output math_start;   output [2:0] math_op;
        output [16:0] math_a, math_b;          input [16:0] math_out;

        reg [4:0] state;      reg skip;  reg [16:0] math_a, math_b,temp_z;      reg math_start;
                                                reg [2:0] math_op;  reg [1:0] vb_vert, vb_xyz;     reg
vb_we;

                                                //output must be clamped to decimal value (10-bit + sign)
                                                assign vb_in = {math_out[16], 6'b000000,
math_out[15:6]};
```

```
        parameter OP_INV = 0;       parameter OP_MUL = 1;       parameter OP_MUL_AC = 2; //math ops that we
need

//      parameter Dx = (17'h1A000);//-160  (2 xtra whole bits)
        //parameter Dy = (17'h17800); //-120  (2 xtra whole bits)

        parameter Dx = 17'h02800; //160/4 (2 xtra whole bits)
        parameter NEG_Dy = 17'h11E00; //-120/4 (2 xtra whole bits)
        parameter Dy = 17'h01E00;   //120 / 4
        parameter Z_NEAR = 17'h00080;//0.5
        parameter ONE = 17'h00100;



        parameter S_IDLE = 0;       parameter S_FETCH_Z = 1;  parameter S_MUL_DX = 2;
        parameter S_FETCH_X = 3;  parameter S_ADD_X = 4;     parameter S_STORE_X = 5;
        parameter S_X_WE_2 = 6;   parameter S_X_WE_3 = 7;    parameter X_WE_LOW = 8;
        parameter S_FETCH_Y = 9;  parameter S_FETCH_Y_2 = 10;          parameter S_ADD_Y = 11;
        parameter S_STORE_Y = 12;           parameter S_Y_WE_2 = 13;  parameter S_Y_WE_3 = 14;
        parameter Y_WE_LOW = 15;            parameter S_ADR = 16;

        always @(posedge clk) begin
                if( reset ) begin
                        state <= S_IDLE;   skip <= 0;            vb_we <= 0;         math_start <= 0;
                end else if( start ) begin
                        state <= S_FETCH_Z;                      vb_xyz <= 2;        vb_vert <= 0;
                        skip <= 0;         vb_we <= 0;           math_start <= 0;    math_op <= 0;
                end else begin
                        case( state )

                                S_FETCH_Z: begin
                                        if( vb_out[16] )     begin      skip <= 1;            state <= S_IDLE;
        end     //clip on negative z values
                                        else if( vb_out < Z_NEAR)   begin      skip <= 1;            state <=
S_IDLE; end //clip closer then 1/2
                                        else begin
                                                math_a <= vb_out; //math_a == z
                                                math_op <= OP_INV;
                                                math_start <= 1;
                                                state <= S_MUL_DX;
                                        end
                                end

                                S_MUL_DX: if(math_done) begin
                                        math_a <= math_out;          //math_a == 1/z
                                        temp_z <= math_out;
                                        math_b <= Dx;
                                        vb_xyz <= 0;        //choose x for next state
                                        math_op <= OP_MUL;
                                        state <= S_FETCH_X;
                                        math_start <= 1;
                                end

                                S_FETCH_X: if(math_done) begin
                                        math_a <= math_out;          //math_a == D/z / 4
                                        math_b <= vb_out; //math_b == x
                                        state <= S_ADD_X;
                                        math_start <= 1;
                                end

                                S_ADD_X: if( math_done ) begin
```

```
                                        if( math_overflow )begin        skip <= 1;              state <= S_IDLE;
        end         //clip if x is out of bounds
                                        else begin
                                                math_op <= OP_MUL_AC;   //D*x/z   /4 is already in the
buffer
                                                math_a <= Dx;
                                                math_b <= ONE; //so we need to add D
                                                state <= S_STORE_X;
                                                math_start <= 1;
                                        end
                                end

                                S_STORE_X: if(math_done) begin
                                        if( math_overflow )begin        skip <= 1;              state <= S_IDLE;
        end         //clip if x is out of bounds
                                        //vb_in is already linked to math out
                                        else begin              vb_we <= 1;             state <= S_X_WE_2;
        end
                                end

                                S_X_WE_2:           state <= S_X_WE_3;
                                S_X_WE_3:           state <= X_WE_LOW;

                                X_WE_LOW: begin             vb_we <= 0;             state <= S_FETCH_Y;
        end

                                S_FETCH_Y: begin            //compute -Dy / z
                                        math_a <= temp_z; math_b <= NEG_Dy;             math_start <= 1;
        //op is still OP_MUL
                                        vb_xyz <= 1;
                                        state <= S_FETCH_Y_2;
                                        math_op <= OP_MUL;
                                end

                                S_FETCH_Y_2: if( math_done ) begin
                                        math_a <= math_out;             //-d/z /4
                                        math_b <= vb_out; //y
                                        math_start <= 1;
                                        state <= S_ADD_Y;
                                end

                                S_ADD_Y: if( math_done) begin
                                        if( math_overflow )begin        skip <= 1;              state <= S_IDLE;
        end         //clip if y is out of bounds
                                        else begin
                                                math_op <= OP_MUL_AC;
                                                math_b <= ONE;    math_a <= Dy;
                                                state <= S_STORE_Y;
                                                math_start <= 1;
                                        end
                                end


                                S_STORE_Y: if(math_done) begin
                                        if( math_overflow) begin        skip <= 1;              state <= S_IDLE;
        end //clip if y if out of bounds
                                        else begin              vb_we <= 1;             state <= S_Y_WE_2;
        end
                                end

                                S_Y_WE_2:           state <= S_Y_WE_3;
                                S_Y_WE_3:           state <= Y_WE_LOW;
```

```
                        Y_WE_LOW: begin         vb_we <= 0;         state <= S_ADR;    end

                        S_ADR: begin
                                vb_vert <= vb_vert + 1;
                                if( vb_vert == 2 )    state <= S_IDLE;    //we're done!
                                else begin            state <= S_FETCH_Z;         vb_xyz <= 2;
        end         //do next vertex
                        end
                        default: state <= S_IDLE;
                    endcase
                end
                if( math_start ) math_start <= 0;
        end

        assign done = (state == S_IDLE) & ~start & ~reset;


endmodule
```