CS151C Winter Quarter 2004
Design of Digital Systems Final Project
Prof. Leon Alkalai

Computer Graphics:
Where Straight Lines, Aren't

Jong, Lorraine
Shirachi, Lisa
Wang, Sherman

3/23/2004

ABSTRACT

In selecting line drawing algorithms as the basis of our project, we were interested in the details and difficulties of representing such a simple element in the digital world. Representing a line on screen seems trivial, but it is still a major area of study within the field of Computer Graphics. The complications involved with representing this fundamental element in VHDL and using the available FPGAs, were nearly overwhelming, and a great number of workarounds were conceived and implemented to compensate for the Xilinx software's deficiencies.

During the course of this quarter, we learned many things with regards to designing and implementing a system. Specifically, given an original high-level description of a system, we became familiar with the process of breaking it into lower-level modules, each of which performed a given function. However, once the overall system had been divided into smaller systems, the implementation had to deal with constraints such as the size of the FPGA, the subset of VHDL supported by the available software, etc. Thus, at each step, we had to refine the original idea, and in some cases, re-implement the system until it matched those limitations, while still providing the desired functionality. The tradeoff for basic functionality included the elimination of some features of the system.

Our first algorithm is the Digital Differential Analyzer which requires floating-point intermediate values. Our second is the Midpoint Line Algorithm, a special case of Bresenham's Line Algorithm, which is famous for its speed and accuracy.

---

1. Introduction

An ideal line is one that is smooth and infinite, however this is not possible to represent on monitors due to the finite number of pixels used within them. In order to translate what comes easily with a pen and paper, a line drawing algorithm must consider where the line would naturally fall and determine how best to represent each portion of the line by determining which pixel to turn "on."

One intuitive way of doing this is using the equation of a line: $y_2 - y_1 = m(x_2 - x_1) + b$. We can step along one axis while using the slope, m, to calculate the coordinate of the other. The Digital Differential Analyzer method applies this by stepping along the axis of greatest change (end point – start point) and incrementing the other by m. The general algorithm (psuedocode) follows:

DDA Algorithm [1]

```
procedure DDA( x1, y1, x2, y2: integer);
var
  dx, dy, steps: integer;
  x_inc, y_inc, x, y: real;
begin
  dx := x2 - x1; dy := y2 - y1;
  if abs(dx) > abs(dy) then
     steps := abs(dx); {steps is larger of dx, dy}
  else
     steps := abs(dy);
  x_inc := dx/steps; y_inc := dy/steps;
  x:=x1; y:=y1;
  set_pixel(round(x), round(y));
  for i := 1 to steps do
    begin
      x := x + x_inc;
      y := y + y_inc;
      set_pixel(round(x), round(y));
    end;
end; {DDA}
```

The key elements in implementing this algorithm are division, floating-point representation and rounding. DDA requires a floating point addition and round at each step. In hardware these are rather difficult and expensive tasks to implement.

Jack Bresenham's line algorithm eliminates the need for floating point by acknowledging that there are only two candidate pixels to choose from at each point to best represent the line. His algorithm considers the centers of these two pixels and uses a decision variable based on the difference between start and end points, to determine whether or not to go horizontal/vertical or diagonal. A variation of Bresenham's Line Algorithm is the Midpoint Line Algorithm. It calculates the middle point between the two possible next pixels and uses it as a basis for deciding which direction to go.

*NOTE: Although we originally had 3 separate modules with each being assigned to each member, we ended up eliminating a module and collaboratively worked on all parts of the system together.

Midpoint Algorithm [2]

```
dx = x2-x1; dy = y2-y1;
d = 2*dy-dx;
inc_E = 2*dy; inc_NE = 2*(dy-dx);
y = y1;
for (x=x1; x <= x2; x++){
  setpixel(x,y);
  if(d > 0){
    y = y+1;
    d = d+inc_NE;
  } else {
    d = d+inc_E;
  }
}
```

2. System Level Description

An overall view to our system is that it takes an input for which algorithm to execute, and outputs a fan-out of lines to the screen.

Our system is composed of several module groups divided into a front-end, a middle (computational) region, and a back-end. The front-end is mainly concerned with taking in the user inputs. The line computations occur within the middle region of the system. The back-end takes the results from the middle region and displays them to the screen.

A walkthrough of how the system functions begins with the user specifying which module to execute and eventually display to the screen. This input is taken and passed onto the middle region, where the specified line algorithm is used to calculate the points which make up the line, given two endpoints, each of which is stored into a buffer. The middle region operates autonomously, with hard-coded endpoints, to compute a fan-out of lines to display on the screen. As the individual points of the lines are computed, they are stored to an internal buffer contained in this region. When the last of these calculations are completed, the buffer is read out to the back-end of the system and subsequently displayed to the screen.

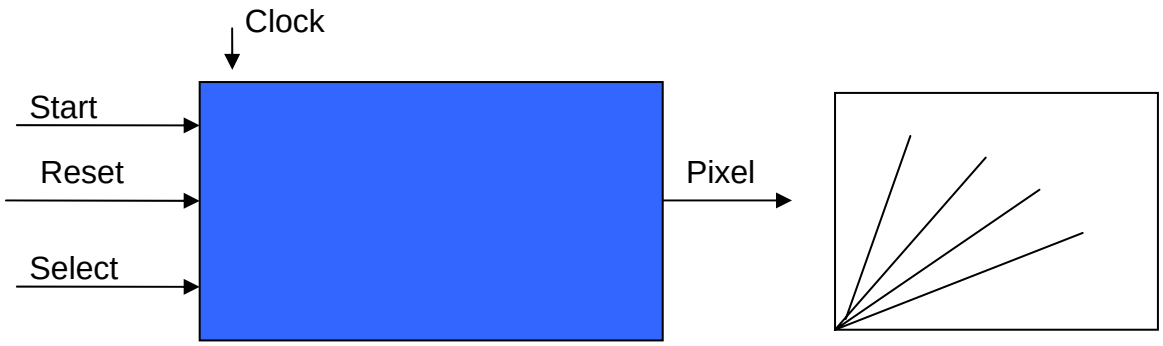


Figure 1. System View

2.1 Subsystem Level Descriptions

The front-end contains the user-input and selector modules. The middle region holds the DDA and midpoint algorithm modules, a buffer module, and a helper module which aids the line computations. The back-end of the system consists of the VGA-output module [3].

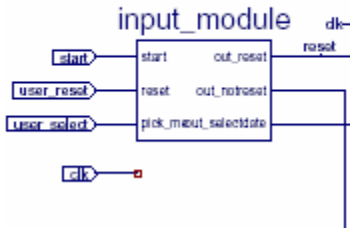


Figure 1: Input Module

The input module, as shown above, receives 3 external inputs. These are the `start`, `reset` and `select` bits. Specifically, the `reset` bit resets the entire system by emitting a high signal to all of the modules except for the VGA module, which resets on a low signal. The fan-out displayed by the system is decided by the user with the `select` bit. When the system is enabled, the value on the select line is piped to the feeder module.

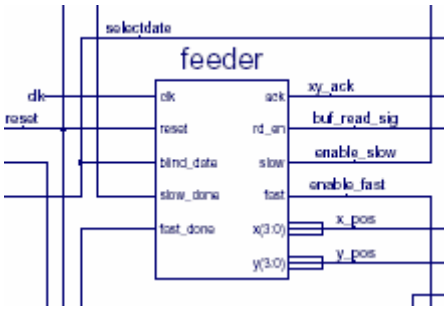


Figure 2: Feeder Module

The feeder module, as shown above, takes 5 inputs. This module runs on the system's clock and receives the `reset` signal from the input module. It also receives the `select` signal and determines which of the 2 algorithm modules to enable and sends the `enable` signals to each module. Two of the inputs take in `done` signals from the modules. This is to indicate that the specified module is finished calculating a given line and should be fed the next set of endpoints. In response to a `done` signal, the feeder module outputs the `xy_ack` signal to indicate it has received the signal. Next, the feeder module transmits the next set of endpoints for the next line to be drawn. These `done` and `ack` signals keep the two modules synchronized throughout the entire line computation process. When the fan-out is complete, the feeder then sends a `read` signal to the picture buffer module, indicating that no more data will be written and it can now be read.

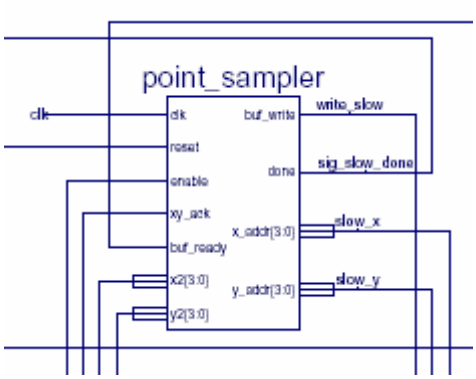


Figure 3: DDA Algorithm Module

The DDA Algorithm module receives a total of 7 signals. This includes `clock`, `reset`, and `enable`, along with `x2` and `y2` from the feeder module. While the module is enabled, it computes a set of points, which are subsequently stored to the picture buffer. This is repeated until the endpoints are reached, in which it then transmits a `done` signal to the feeder module so the next set of end points may be sent. Execution is then halted until an `ACK` signal is received.

In terms of the implementation of the DDA algorithm, it is very expensive and the available software and hardware is unable to handle division or floating point values. To get around this, we implemented a state machine to provide integer division; however, the algorithm also requires floating point values to determine which pixel on the screen it is closer to so it may adjust the rate at which the line is incrementing in the x or y direction. As a result of this deficiency, the coordinates generated by the algorithm module for each pixel are such that the line traverses diagonally until the limiting x or y position is hit, at which point it travels vertically or horizontally.

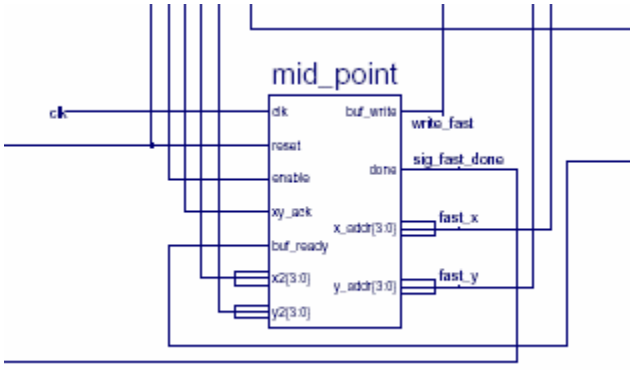


Figure 4: Midpoint Algorithm Module

The midpoint algorithm module's operation is similar to DDA algorithm module described above. Since the midpoint algorithm is designed around integer arithmetic, the workarounds needed for DDA are not needed. Thus, the midpoint algorithm is more accurately implemented than DDA.

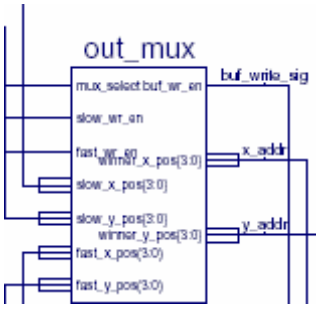


Figure 5: Mux Module

The mux module's behavior follows that of a typical multiplexer. Specifically, the midpoint and DDA modules send pixel coordinates and write-enable signals as inputs, with the `select` bit choosing which set of signals that ultimately pass through to the picture buffer.

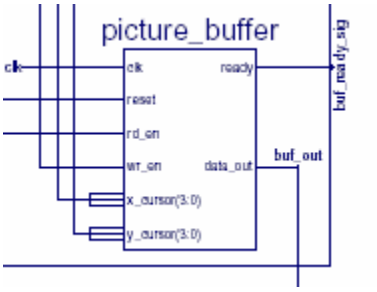


Figure 6: Picture buffer Module

As with other modules, the picture buffer takes in `clock` and `reset`, as well as a read and write enable signal. At the beginning of execution, when `reset` is set high, the buffer is initially cleared out with all elements set to '1' to denote a completely white screen. When the write enable line is high, the x and y coordinates are used as indexes into the 2-dimensional array of 1-bit registers, where a '0' is stored to denote a point on a line. A ready signal is then transmitted to the algorithm modules as acknowledgement that the write has been completed, and another set of points may be sent. Reading of the buffer to the VGA module, which occurs when the fan-out of lines is complete, is done serially. Each clock cycle, one bit is sent to the VGA module in sequential order, so that the lines may be displayed on the screen.

Using the system clock, the picture buffer should be able to continuously feed data to the VGA module. However, this was found to not be the case. Synchronizing the VGA module to accept external signals was not possible and thus a dynamically-determined display was not feasible given time constraints.

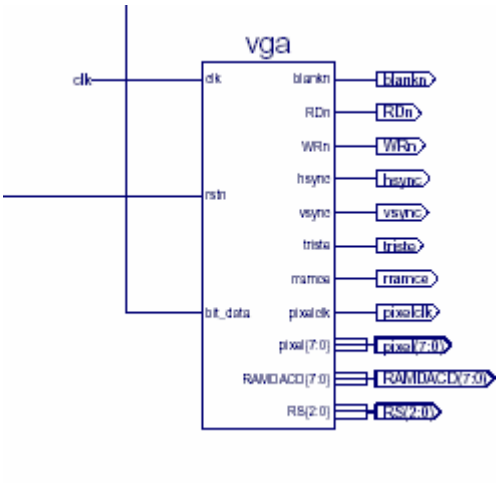


Figure 7: Modified VGA Module

The VGA module is based on one originally implemented at the University of Queensland, Australia [3]. It was modified such that it would receive a data bit from the picture buffer module, and after testing the bit value, would either map a black or white signal to the screen. During a single clock cycle, the picture buffer module feeds the vga module with an element of its buffer, at which point the vga module evaluates it and maps the corresponding color to the screen at its present location. Similar to the other modules of the system, the vga module receives the `clock` and `reset` signals. However, in this case, its reset signal, `rstn`, is "active-low". In terms of inter-module synchronization, the system clock is used for this purpose.

3. RTL Description

The majority of the modules thus presented are fairly basic in nature. For example, the input module is essentially a selector. The output mux is exactly as described; a multiplexer. The picture buffer is a simple array of 1-bit registers, with a read mode, write mode, and a reset mode which writes a '1' to all registers. Thus, the focus of this discussion will lie with the larger modules in our system. Namely, the Feeder, the DDA, and the Mid-point algorithm modules.

Structurally, there is not much to the Feeder. There are selectors and multiplexers, a comparator, and a register. The inputs and outputs of these primitive modules are mentioned below.

The Feeder controls the majority of the system by passing signals to particular modules as appropriate. The enable lines to enable the algorithm modules basically form an internal selector based on the input of the blind_date line. The x and y coordinate groups to send to the algorithm modules are fed by two five-element ROM files which are used as the inputs to a mux. The input to this multiplexer is a register which is incremented every time the algorithm module sends back a done signal to indicate it is done processing a line. Thus, the write-enable bit on this register is tied to the done signal inputs ORed together. A comparator checks the value of the register against the internal value of 5, and enables the rd_en line when the register contains a value of 5 or above.

From an RTL perspective, the DDA module is much more interesting than the Feeder. There are eight states in the control subsystem of the DDA module, and seven registers for internal storage. All of the registers are behind multiplexers, as they can all take in inputs from multiple sources to be used later. There are comparators set up between the register called "temp" and inputs x2 and y2. Registers "x" and "y" are also compared against x2 and y2, and register "i" is compared with register "steps".

The first state, s1wait, is a system startup synchronization state, and nothing meaningful occurs here except for a transition to s1.

In state s1, the mux selecting temp is set to 0, storing a zero value in temp. The result of the comparator between x2 and y2 determines the value stored in register steps; if y2 < x2, then steps takes on the value in x2. Otherwise, steps will hold the value in y2. Register i's mux is also set to 0, and the write enable on these three registers is set to high.

State s2 is one of the pseudo-division steps required in the DDA algorithm. Register temp has write enable set to high, mux selected to 1, and stores the value of "temp - steps". Register i also has write enable set high and its mux set to 1, and the value is incremented by 1. The result of a compare between register temp and x2 determines the write enable on register x_inc and the next state of the machine. If temp is less than x2, registers temp and i have their muxes selected to 0 as they store zero values in preparation for the next state.

State s3 operates in much the same way as s2, except the comparison to determine the next state and value of y_inc is determined by a comparison between temp and y2. If the comparison "temp < y2" is not true, then the write enable on y_inc is not set. Otherwise, it is set, as is the write enable on temp and i.

The next state, s4, is a preparation state for calculating all the points in the line. Registers x, y, and i are all write-enabled, and their muxes are set to 0 to store zero values within them.

State s5 contains the meat of the algorithm. There is a comparison between register i and steps, and the result of "i < steps" is ANDed with the results of comparisons between register x and x2, register y and y2. If these conditions pass, the write enable on registers x and/or y are set to 1, and they take on the value of "x + x_inc" or "y + y_inc" respectively. Register i also has its write enable set and is incremented by 1.

The final two states, s6 and done, do not really concern the data subsystem of the module. They are for synchronization in writing data to the picture buffer and requesting a new set of points from the Feeder module.

The Midpoint module is similar to the DDA module in its use of states. While there are fewer states overall, there are corresponding synchronization states for startup, with the Feeder module, and with the picture buffer module. Thus, out of a possible seven states -- s1wait, s1, s2, s3a, s3b, s4, and stop -- the focus will be laid upon states s1, s2, and s3b.

Structurally, each of the nine registers used within the Midpoint module are behind multiplexers to select their inputs to store. There are comparators between inputs x2 and y2, registers "a" and "temp2", and tests to see if register "d" is negative or positive. Incrementors exist for registers "a" and "b", and there is a subtractor which takes in inputs from registers "db" and "da".

State s1 has registers "x", "y", "da", "a", "db", "b", and "d" all with their write-enable bits set high. A value of zero is stored in registers "x" and "y", with their respective muxes also set to zero. A comparison between inputs x2 and y2 determine the inputs to the muxes for "da", "a", "db", and "b". If x2 is greater than or equal to y2, then "da" receives x2's value, "a" gets the value of "x", "db" gets the value of y2, and "b" gets the value of "y". Otherwise, the x's and y's are reversed, with "da" receiving y2, etc. Lastly, after all of the registers are assigned, register "d" obtains the value of "da" subtracted from "db".

State s2 acts as the condition check for a while loop. Again, the comparison of "x2 >= y2" is checked again. This determines if register "x", "y", and "temp2" get the values of "a", "b", and x2 respectively, or vice versa for the x and y statements. Thus, the result of the comparator sets the muxes for registers "x", "y", and "temp2", and the write-enable bits for those registers are also set.

State s3b begins with a check to see if register "d" is negative. The result of this compare feeds into the mux for register temp and determines whether the value of register "db" or the subtraction of "da" from "db" is stored to that register. If the second branch is taken, register "b" is incremented. Register "a" is also incremented, and register "d" takes in the result of "d" and "temp". The muxes for these registers are set appropriately, and write-enable is set for all the registers involved in a store operation.

.

4. VHDL Simulations

In the case of most modules, state machines were implemented in VHDL to control the outputs. The conditions for the state transitions were normally based on inputs and/or the clock cycle.

The provided Xilinx ModelSim simulator was used as a foundation for testing the functionality of the system as well as individual modules. By using the simulation software, signals were able to be manually set and the behavior of signals could be viewed, as they propagated through the system and consequently enabled and started the other modules. Specifically, choosing one of the two algorithms to execute, the calculated x and y-coordinates could be seen and the corresponding pixel color that would be output, could be viewed as well.

One issue encountered was that while the functionality of the system could be verified using the simulator, implementing it on the provided FPGA was a hurdle. After working around the spacing issues by redesigning modules, timing was a major issue with regards to the VGA interface. Among the modules that were designed specifically for our system, synchronization was not an issue while effective communication with the VGA module required essentially re-designing it which was not the purpose of the project.

The system was completely simulated using ModelSim, however, while the system file was implemented and the programming file was created, the actual behavior of the system on the board is not known, as synchronization difficulties between the designed system and the VGA module seemed to prevent any signals from being output to the monitor. (see Appendix A)

5. Conclusions

The designed system effectively implemented two different algorithms for calculating the intermediate points in a line given the two endpoints. In designing this system, it was illustrated how the drawing of a simple line is more complex than initially thought. While the calculated points were not able to be visually shown on a monitor display, simulation illustrated the issues that arise upon calculating a line and reinforced the fact that different algorithms for calculating points in a line, produce different results that significantly impact the appearance of a line.

6. References

Conversions with a number of our classmates has helped us steer clear of a number of obstacles namely, reading from SRAM to output to the VGA. The general consensus when it came to the VGA was that it's problematic, knowing that really wasn't much help to us and a lot of problems we ran into weren't logical, but it didn't make as us any worse.

[1] ACM SIGGRAPH Educational Committee, Hypermedia and Visualization Laboratory, Georgia State University, National Science Foundation, "HyperGraph," May 1998, http://www.siggraph.org/education/materials/HyperGraph/hypergraph.htm.

[2] Hill, Francis S. Jr., *Computer Graphics Using OpenGL*, New York: Prentice Hall, 2000.

[3] School of Computer Science and Electrical Enginneering, University of Queensland, "VHDL Interfaces and Example Designs: VGA Interface," February 2001, http://www.csee.uq.edu.au/.

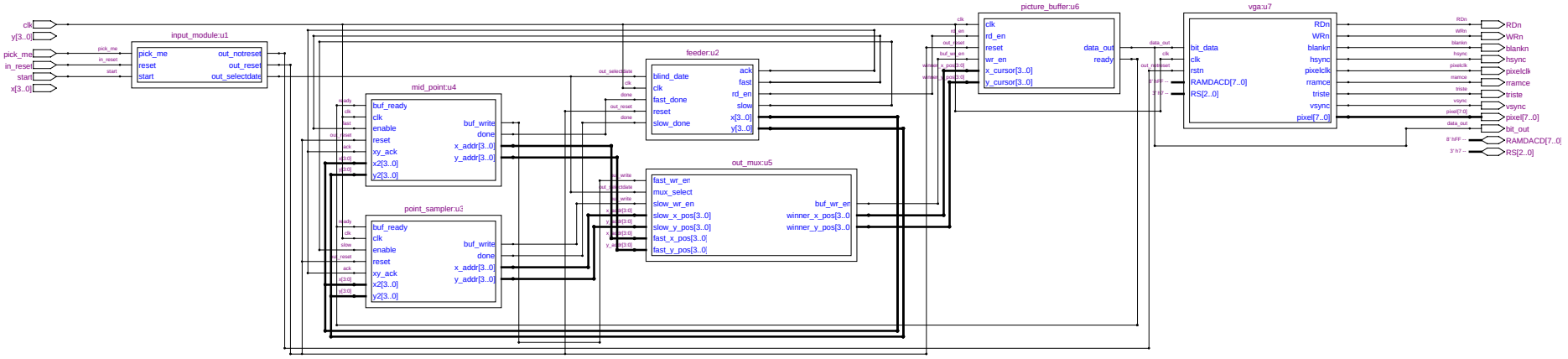
```
1: library IEEE;
2: use IEEE.std_logic_1164.all;
3: package linepack is
4:   constant zero : std_logic_vector := "0000";
5:   constant width : integer := 3;
6:   constant screen_dim : integer := 15;
7:
8: end linepack;
```

```vhdl
library IEEE;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use work.LINEPACK.all;

entity lines is
    port (clk : in std_logic;
        start : in std_logic;
        in_reset : in std_logic;
        pick_me : in std_logic;
        x: in std_logic_vector(width downto 0); -- the end point
        y: in std_logic_vector(width downto 0);

        pixel: out STD_LOGIC_VECTOR (7 downto 0);               -- RAMDAC pixel lines
        blankn: out STD_LOGIC;                                  -- RAMDAC blank signal
        RDn: out STD_LOGIC;
 -- RAMDAC RDn connection
        WRn: out STD_LOGIC;                                     -- RAMDAC WRn connection
        RAMDACD: inout STD_LOGIC_VECTOR (7 downto 0);   -- RAMDAC data lines
        RS: inout STD_LOGIC_VECTOR (2 downto 0);                -- RAMDAC RS lines
        hsync: out STD_LOGIC;                                   -- horizontal sync for monitor
        vsync: out STD_LOGIC;                                   -- vertical sync for monitor
        triste: out STD_LOGIC;                                  -- signal to tristate ethernet PHY
        rramce: out STD_LOGIC;                                  -- right ram chip enable
        pixelclk: out STD_LOGIC;                                -- RAMDAC pixel clock
        bit_out : out std_logic  -- output from buffer, feed to vgaout later?

    );
end lines;

architecture structural of lines is
    component input_module is
        port (
            start : in std_logic;
            reset : in std_logic;
            pick_me : in std_logic;
            out_reset : out std_logic;
            out_notreset : out std_logic;
            out_selectdate : out std_logic
                );
    end component;

    component feeder is
        port (
            clk : in std_logic;
            reset : in std_logic;
            blind_date : in std_logic;
            slow_done: in std_logic;
            fast_done: in std_logic;
            ack : out std_logic;
            rd_en : out std_logic;
            x, y : out std_logic_vector(width downto 0);
            slow : out std_logic;
            fast : out std_logic
            );
    end component;

    component point_sampler is
        port (
            clk : in std_logic;
```

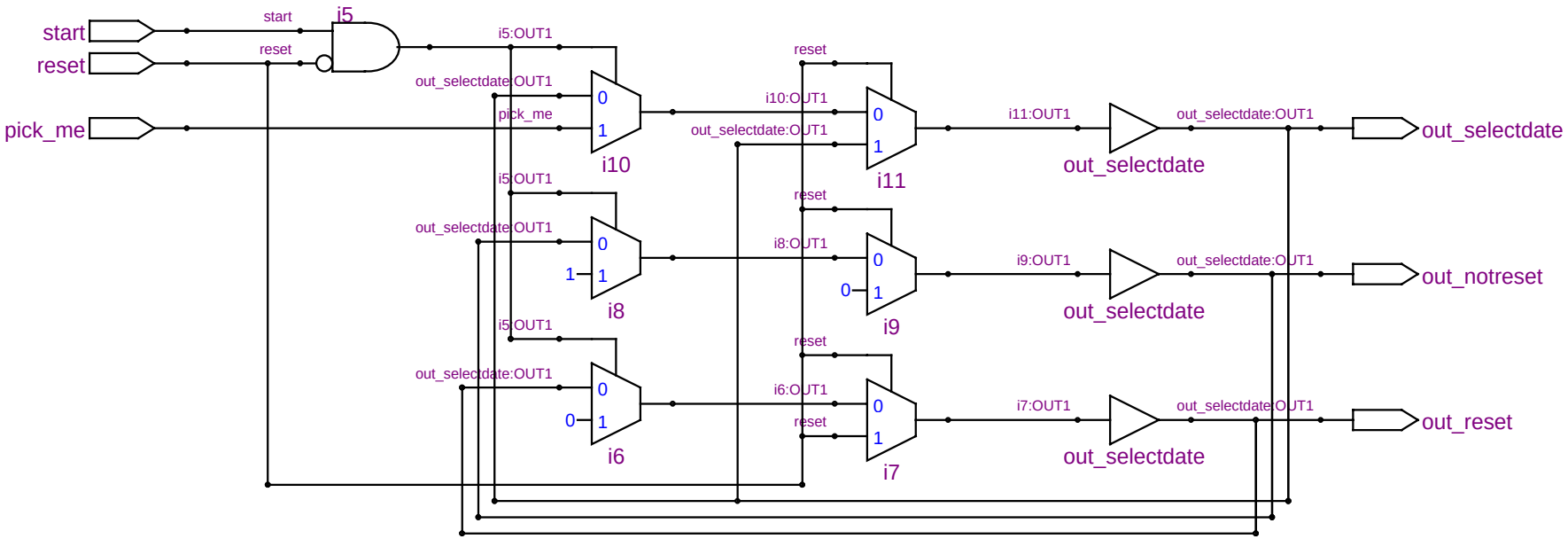
```
61:             reset : in std_logic;
62:             enable : in std_logic;
63:             x2 : in std_logic_vector(width downto 0);
64:             y2 : in std_logic_vector(width downto 0);
65:             xy_ack : in std_logic;
66:             buf_ready : in std_logic;
67:             buf_write : out std_logic;
68:             done : out std_logic;
69:             x_addr : out std_logic_vector(width downto 0);
70:             y_addr : out std_logic_vector(width downto 0)
71:         );
72:     end component;
73:
74:     component mid_point is
75:         port (
76:             clk : in std_logic;
77:             reset : in std_logic;
78:             enable : in std_logic;
79:             x2 : in std_logic_vector(width downto 0);
80:             y2 : in std_logic_vector(width downto 0);
81:             xy_ack : in std_logic;
82:             buf_ready : in std_logic;
83:             buf_write : out std_logic;
84:             done : out std_logic;
85:             x_addr : out std_logic_vector(width downto 0);
86:             y_addr : out std_logic_vector(width downto 0)
87:         );
88:     end component;
89:
90:     component out_mux is
91:         port (
92:             mux_select : in std_logic;
93:             slow_x_pos : in std_logic_vector(width downto 0);
94:             slow_y_pos : in std_logic_vector(width downto 0);
95:             slow_wr_en : in std_logic;
96:             fast_x_pos : in std_logic_vector(width downto 0);
97:             fast_y_pos : in std_logic_vector(width downto 0);
98:             fast_wr_en : in std_logic;
99:             winner_x_pos : out std_logic_vector(width downto 0);
100:            winner_y_pos : out std_logic_vector(width downto 0);
101:            buf_wr_en : out std_logic
102:        );
103:    end component;
104:
105:    component picture_buffer is
106:        port (
107:            clk : in std_logic;
108:            reset : in std_logic;
109:            rd_en : in std_logic;               -- enable read
110:            wr_en : in std_logic;               -- enable write
111:
112:            -- x, y position to be input by the algorithm modules
113:            x_cursor : in std_logic_vector(width downto 0);
114:            y_cursor : in std_logic_vector(width downto 0);
115:            ready : out std_logic;              -- signal to other modules; ready to
    write
116:            data_out : out std_logic
117:        );
118:    end component;
119:
120:    component vga is
121:        port (
122:            clk: in STD_LOGIC;
     -- clock
123:            rstn: in STD_LOGIC;
     -- asynchronous active low reset
124:            bit_data : in std_logic;                -- data from picture buffer
125:            pixel: out STD_LOGIC_VECTOR (7 downto 0);           -- RAMDAC pixel lines
```

```
126:             blankn: out STD_LOGIC;
      -- RAMDAC blank signal
127:             RDn: out STD_LOGIC;
      -- RAMDAC RDn connection
128:                 WRn: out STD_LOGIC;
      -- RAMDAC WRn connection
129:                 RAMDACD: inout STD_LOGIC_VECTOR (7 downto 0);   -- RAMDAC data lines
130:                 RS: inout STD_LOGIC_VECTOR (2 downto 0);                -- RAMDAC RS
    lines
131:             hsync: out STD_LOGIC;
      -- horizontal sync for monitor
132:             vsync: out STD_LOGIC;
      -- vertical sync for monitor
133:             triste: out STD_LOGIC;
      -- signal to tristate ethernet PHY
134:             rramce: out STD_LOGIC;
      -- right ram chip enable
135:             pixelclk: out STD_LOGIC
      -- RAMDAC pixel clock
136:         );
137:     end component;
138:
139:       signal selectdate : std_logic;
140:       signal enable_slow : std_logic;
141:       signal enable_fast : std_logic;
142:       signal slow_x : std_logic_vector(width downto 0);
143:       signal slow_y : std_logic_vector(width downto 0);
144:       signal fast_x : std_logic_vector(width downto 0);
145:       signal fast_y : std_logic_vector(width downto 0);
146:       signal x_pos : std_logic_vector(width downto 0); -- xy_input_mod -> algorithm
147:       signal y_pos : std_logic_vector(width downto 0);
148:       signal sig_slow_done : std_logic;
149:       signal sig_fast_done : std_logic;
150:       signal reset : std_logic;
151:       signal notreset : std_logic;
152:       signal xy_ack : std_logic;
153:       signal buf_ready_sig : std_logic;                    -- buffer ready to write
154:       signal write_slow : std_logic;                       -- "slow" assert write on
    buffer
155:       signal write_fast : std_logic;                       -- "fast" assert write on
    buffer
156:       signal buf_read_sig : std_logic;
157:       signal buf_write_sig : std_logic;                    -- buffer write signal
    (from mux)
158:       signal x_addr : std_logic_vector(width downto 0);    -- x address for buffer
159:       signal y_addr : std_logic_vector(width downto 0);    -- y address for buffer
160:       signal buf_out : std_logic;                          -- buffer data out line
161:
162:       begin
163:         bit_out <= buf_out;
164:
165:       u1: input_module
166:             port map(start, in_reset, pick_me, reset, notreset, selectdate);
167:
168:       u2: feeder
169:             port map(clk, reset, selectdate, sig_slow_done, sig_fast_done, xy_ack,
    buf_read_sig, x_pos, y_pos, enable_slow, enable_fast);
170:
171:       u3: point_sampler
172:             port map(clk, reset, enable_slow, x_pos, y_pos, xy_ack, buf_ready_sig,
    write_slow, sig_slow_done, slow_x, slow_y);  -- change x/y -> x_pos/y_pos
173:
174:       u4: mid_point
175:             port map(clk, reset, enable_fast, x_pos, y_pos, xy_ack, buf_ready_sig,
    write_fast, sig_fast_done, fast_x, fast_y);  -- change x/y -> x_pos/y_pos
176:
177:       u5: out_mux
178:             port map(selectdate, slow_x, slow_y, write_slow, fast_x, fast_y,
    write_fast, x_addr, y_addr, buf_write_sig);
```

```
179:
180:       u6: picture_buffer
181:             port map(clk, reset, buf_read_sig, buf_write_sig, x_addr, y_addr,
    buf_ready_sig, buf_out);
182: --             port map(clk, reset, buf_read_sig, buf_write_sig, x_addr, y_addr,
    buf_ready_sig, bit_out);
183:
184:       u7: vga
185:             port map(clk, notreset, buf_out, pixel, blankn, RDn, WRn, RAMDACD, RS,
    hsync, vsync, triste, rramce, pixelclk);
186:
187:
188: end structural;
```

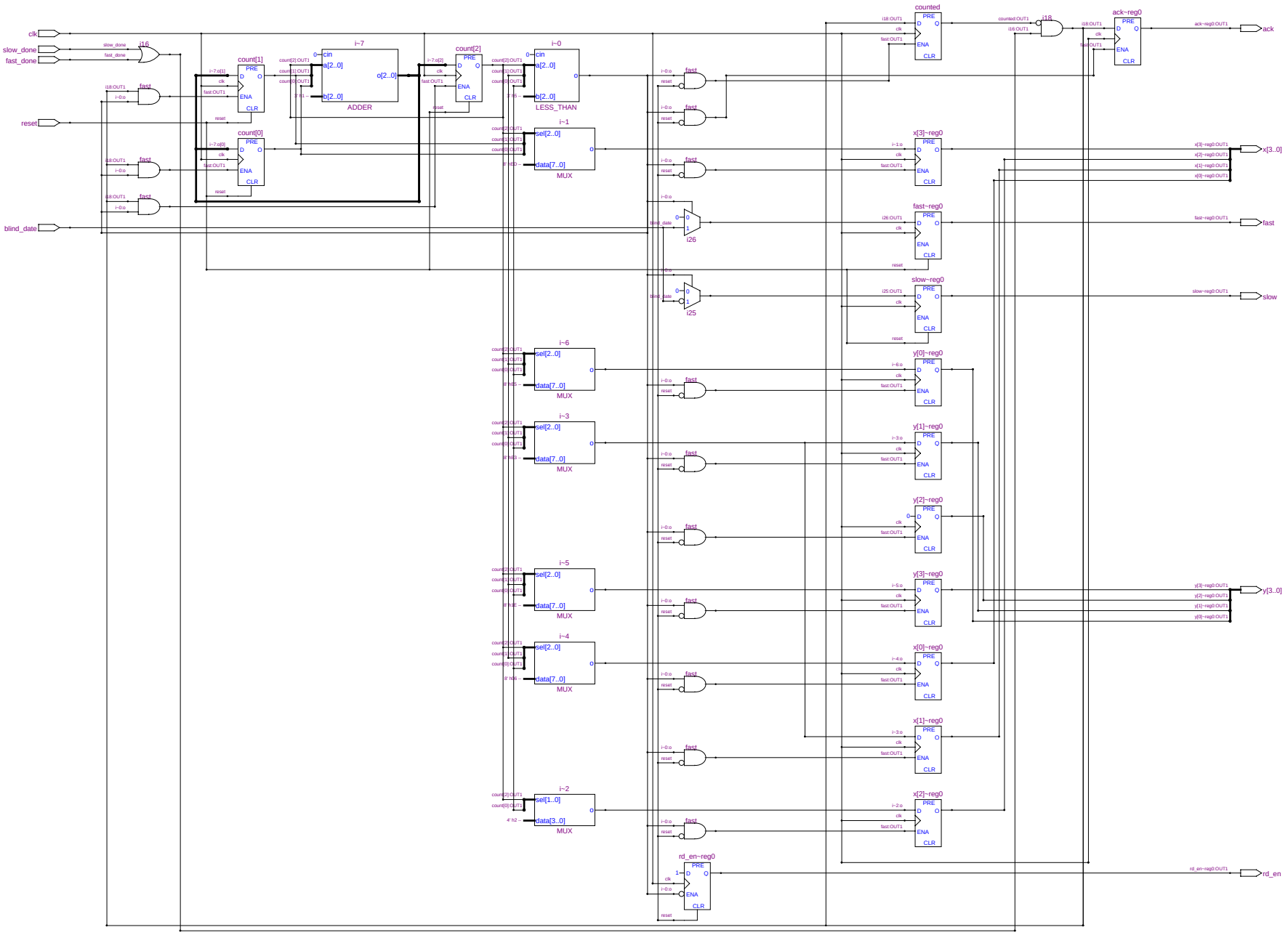

clk
y[3..0]
pick_me
in_reset
start
x[3..0]
input_module:u1
pick_me
reset
start
out_notreset
out_reset
out_selectdate
mid_point:u4
point_sampler:u3
buf_ready
clk
enable
reset
xy_ack
x2[3..0]
y2[3..0]
buf_write
done
x_addr[3..0]
y_addr[3..0]
feeder:u2
blind_date
clk
fast_done
reset
slow_done
ack
fast
rd_en
slow
x[3..0]
y[3..0]
out_mux:u5
fast_wr_en
mux_select
slow_wr_en
slow_x_pos[3..0]
slow_y_pos[3..0]
fast_x_pos[3..0]
fast_y_pos[3..0]
buf_wr_en
winner_x_pos[3..0]
winner_y_pos[3..0]
picture_buffer:u6
clk
rd_en
reset
wr_en
x_cursor[3..0]
y_cursor[3..0]
data_out
ready
vga:u7
bit_data
clk
rstn
RAMDACD[7..0]
RS[2..0]
RDn
WRn
blankn
hsync
pixelclk
rramce
triste
vsync
pixel[7..0]
bit_out
RAMDACD[7..0]
RS[2..0]

```
 1: library IEEE;
 2: use IEEE.std_logic_arith.all;
 3: use IEEE.std_logic_1164.all;
 4: use IEEE.std_logic_unsigned.all;
 5: use work.LINEPACK.all;
 6:
 7: entity input_module is
 8:     port (
 9:         start : in std_logic;
10:         reset : in std_logic;
11:         pick_me : in std_logic;
12:         out_reset : out std_logic;
13:         out_notreset : out std_logic;
14:         out_selectdate : out std_logic
15:         );
16: end input_module;
17:
18: architecture behavioral of input_module is
19: begin
20:     process (reset, start, pick_me)
21:     begin
22:         if (reset = '1') then
23:             out_reset <= reset;
24:             out_notreset <= '0';
25:         elsif (start = '1' and reset = '0') then
26:                 out_reset <= '0';
27:             out_notreset <= '1';
28:             out_selectdate <= pick_me;
29:         end if;
30:     end process;
31:
32: end behavioral;
33:
```


start
reset
pick_me
start
reset
i5
i5:OUT1
out_selectdate:OUT1
pick_me
0
1
i10
i10:OUT1
out_selectdate:OUT1
reset
0
1
i11
i11:OUT1
out_selectdate
out_selectdate:OUT1
out_selectdate
i5:OUT1
out_selectdate:OUT1
1
0
1
i8
i8:OUT1
reset
0
1
i9
i9:OUT1
out_selectdate
out_selectdate:OUT1
out_notreset
i5:OUT1
out_selectdate:OUT1
0
0
1
i6
i6:OUT1
reset
0
1
i7
i7:OUT1
out_selectdate
out_selectdate:OUT1
out_reset

```vhdl
library IEEE;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use work.LINEPACK.all;

entity feeder is
    port (
        clk : in std_logic;
        reset : in std_logic;
        blind_date : in std_logic;
        slow_done: in std_logic;
        fast_done: in std_logic;
        ack : out std_logic;
        rd_en : out std_logic;
        x, y : out std_logic_vector(width downto 0);
        slow : out std_logic;
        fast : out std_logic
        );
end feeder;

architecture behavioral of feeder is
begin

    process(clk, reset, blind_date)
        subtype dimension is std_logic_vector(width downto 0);
        type d_array is array(0 to 4) of dimension;
        constant x_values : d_array := ("1010", "0111", "1001", "1100", "0000");
        constant y_values : d_array := ("0011", "1010", "1001", "1000", "1000");
        variable count : std_logic_vector(2 downto 0) := "000";
        variable counted : std_logic := '0';

        begin

            if reset='1' then
            -- reset the entire system
                count := "000";
                rd_en <= '0';
                slow <= '0';
                fast <= '0';
            elsif (rising_edge(clk) and clk = '1') then
                if count < "101" then
                    case blind_date is
                       when '0' =>
                           slow <= '1';
                           fast <= '0';
                       when '1' =>
                           slow <= '0';
                           fast <= '1';
                       when others =>
                           slow <= '0';
                           fast <= '0';
                    end case;

                    case count is
                        when "000" =>
                            x <= x_values(0);
                            y <= y_values(0);
                        when "001" =>
                            x <= x_values(1);
                            y <= y_values(1);
                        when "010" =>
                            x <= x_values(2);
                            y <= y_values(2);
                        when "011" =>
                            x <= x_values(3);
                            y <= y_values(3);
                        when "100" =>
```

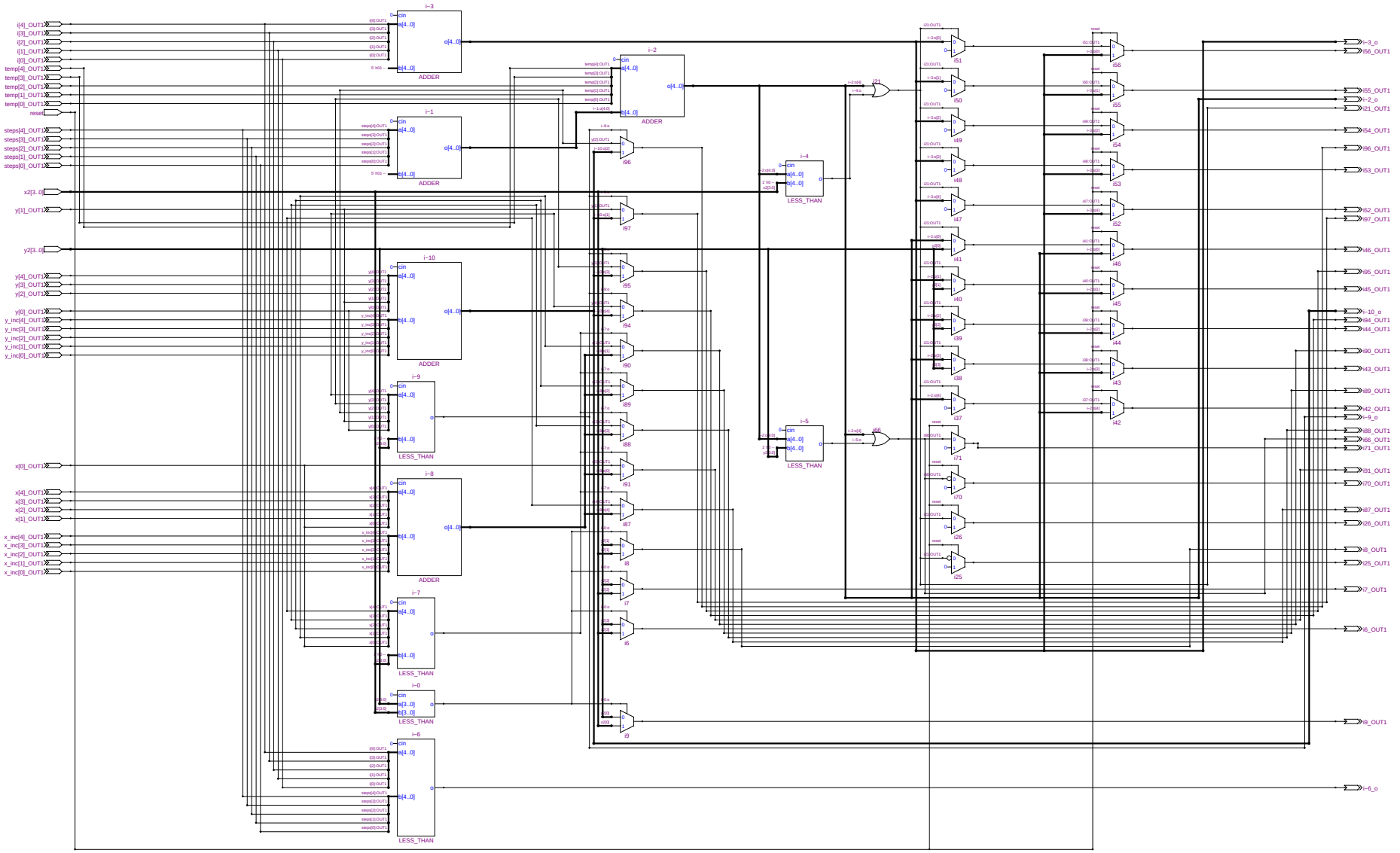
```
69:                             x <= x_values(4);
70:                             y <= y_values(4);
71:                         when others =>
72:                             x <= x_values(0);
73:                             y <= y_values(0);
74:                     end case;
75:
76:                     if (((slow_done = '1') or (fast_done = '1')) and (counted = '0'))
    then
77:                         count := count + '1';
78:                         counted := '1';
79:                         ack <= '1';
80:                     else
81:                         ack <= '0';
82:                         counted := '0';
83:                     end if;
84:                 else
85:                 -- all lines done, enable read for picture buffer
86:                     rd_en <= '1';
87:                     -- disable algoritms
88:                     slow <= '0';
89:                     fast <= '0';
90:                 end if;
91:             end if;
92:     end process;
93: end behavioral;
```


clk
slow_done
fast_done
reset
blind_date
i16
fast
count[1]
count[0]
i~7
ADDER
count[2]
i~0
LESS_THAN
i~1
MUX
i~6
i~3
i~5
i~4
i~2
counted
i18
ack~reg0
ack
x[3]~reg0
x[3..0]
fast~reg0
fast
i26
slow~reg0
slow
i25
y[0]~reg0
y[1]~reg0
y[2]~reg0
y[3]~reg0
y[3..0]
x[0]~reg0
x[1]~reg0
x[2]~reg0
rd_en~reg0
rd_en

```vhdl
library IEEE;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use work.LINEPACK.all;

entity point_sampler is
    port (
        clk : in std_logic;
        reset : in std_logic;
        enable : in std_logic;
        x2 : in std_logic_vector(width downto 0);
        y2 : in std_logic_vector(width downto 0);
        xy_ack : in std_logic;
        buf_ready : in std_logic;
        buf_write : out std_logic;
        done : out std_logic;
        x_addr : out std_logic_vector(width downto 0);
        y_addr : out std_logic_vector(width downto 0)
    );
end point_sampler;

architecture behavioral of point_sampler is
    TYPE stateT is (s1wait, s1, s2, s3, s4, s5, s6, stop);
    SIGNAL state : stateT := s1wait;
    begin
        process(clk, state, enable)
        variable steps : std_logic_vector((width + 1) downto 0);
        variable x_inc : std_logic_vector((width + 1) downto 0);
        variable y_inc : std_logic_vector((width + 1) downto 0);
        variable x : std_logic_vector((width + 1) downto 0);
        variable y : std_logic_vector((width + 1) downto 0);
        variable i : std_logic_vector((width + 1) downto 0);
        variable temp : std_logic_vector((width + 1) downto 0);
        variable temp_high : std_logic;

    begin
        if(enable = '1') then
            if(rising_edge(clk) and clk = '1') then
                case state is

                    -- new "wait" state to solve timing issues
                    -- before, couldn't store "fast" enough into temp
                    -- which led to some funky undefined temp
                    -- and nothing being outputted as a result
                    -- when mated with xy_input module
                    when s1wait =>
                        temp := '0' & x2;
                        state <= s1;

                    when s1=>
                        done <= '0';
                        temp := '0' & x2;
                        if y2 < x2 then
                            steps := '0' & x2;
                        else
                            steps := '0' & y2;
                        end if;

                        -- setup for divide
                        i := '0' & zero;
                        if reset = '1' then
                            state <= s1;
                        else
                            state <= s2;
                        end if;

                    -- dx stuff
```

```
69:                     when s2 =>
70:                         temp := temp + (not(steps) + '1');
71:                         temp_high := temp(width + 1);
72:
73:                         i := i + '1';
74:
75:                         if reset = '1' then
76:                             state <= s1;
77:                         elsif (temp < x2) or (temp_high = '1') then
78:                             x_inc := i;
79:                             temp := '0' & y2;
80:                             i := '0' & zero;
81:                             state <= s3;
82:                         else
83:                             state <= s2;
84:                         end if;
85:
86:                     -- dy stuff
87:                     when s3 =>
88:                         temp := temp + (not(steps) + '1');
89:                         temp_high := temp(width + 1);
90:                         i := i + '1';
91:
92:                         if reset = '1' then
93:                             state <= s1;
94:                         elsif (temp < y2) or (temp_high = '1') then
95:                             y_inc := i;
96:                             state <= s4;
97:                         else
98:                             state <= s3;
99:                         end if;
100:
101:                    when s4 =>
102:                        -- for loop (we wish)
103:                        x := zero & '0';
104:                        y := zero & '0';
105:                        x_addr <= zero;
106:                        y_addr <= zero;
107:                        i := '0' & zero;
108:
109:                        if reset = '1' then
110:                            state <= s1;
111:                        else
112:                            state <= s5;
113:                        end if;
114:
115:                    when s5 =>
116:                        if reset = '1' then
117:                            state <= s1;
118:                        elsif i < steps then
119:                           if x < ('0' & x2) then
120:                               x := x + x_inc;
121:                           end if;
122:                           if y < ('0' & y2) then
123:                               y := y + y_inc;
124:                           end if;
125:                           i := i + '1';
126:
127:                           state <= s6;
128:                        else
129:                          done <= '1';
130:                          state <= stop;
131:                        end if;
132:
133:                    when s6 =>
134:                        buf_write <= '1';
135:                        x_addr <= x(width downto 0);
136:                        y_addr <= y(width downto 0);
```

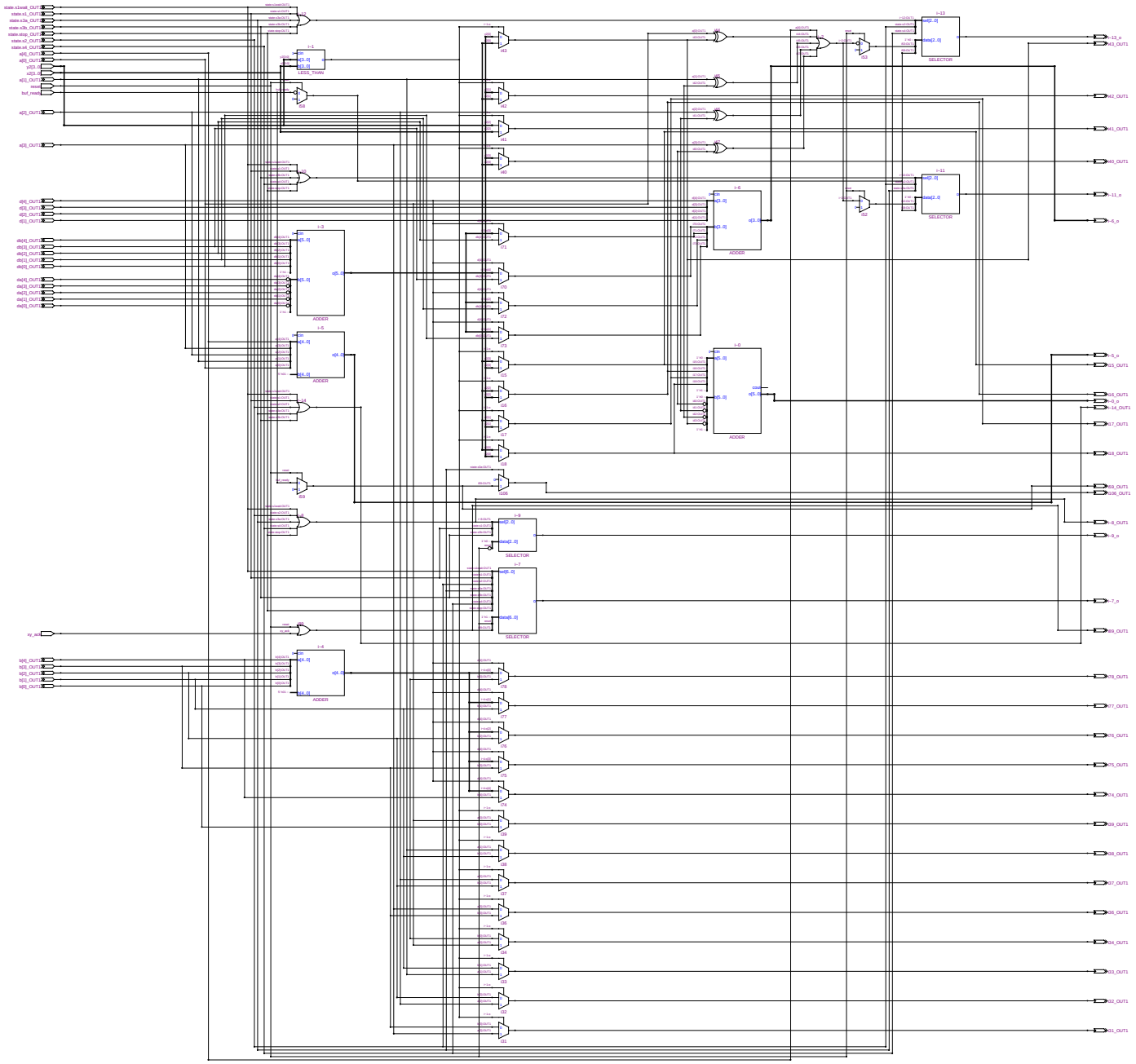
```
137:
138:                         if reset = '1' then
139:                             state <= s1;
140:                         elsif buf_ready = '0' then
141:                             state <= s6;
142:                         else
143:                             state <= s5;
144:                         end if;
145:
146:                     when stop =>
147:                         done <= '1';
148:                         if ((reset = '1') or (xy_ack = '1')) then
149:                             state <= s1;
150:                             done <= '0';
151:                         else
152:                             state <= stop;
153:                             done <= '1';
154:                         end if;
155:                 end case;
156:             end if;
157:         else
158:             done <= '0';
159:             buf_write <= '0';
160:         end if;
161:
162:     end process;
163: end behavioral;
164:
```


i[4]_OUT1
i[3]_OUT1
i[2]_OUT1
i[1]_OUT1
i[0]_OUT1
temp[4]_OUT1
temp[3]_OUT1
temp[2]_OUT1
temp[1]_OUT1
temp[0]_OUT1
reset
steps[4]_OUT1
steps[3]_OUT1
steps[2]_OUT1
steps[1]_OUT1
steps[0]_OUT1
x2[3..0]
y[1]_OUT1
y2[3..0]
y[4]_OUT1
y[3]_OUT1
y[2]_OUT1
y[0]_OUT1
y_inc[4]_OUT1
y_inc[3]_OUT1
y_inc[2]_OUT1
y_inc[1]_OUT1
y_inc[3]_OUT1
x[0]_OUT1
x[4]_OUT1
x[3]_OUT1
x[2]_OUT1
x[1]_OUT1
x_inc[4]_OUT1
x_inc[3]_OUT1
x_inc[2]_OUT1
x_inc[1]_OUT1
x_inc[0]_OUT1
i~3
i~2
i~1
i~10
i~9
i~8
i~7
i~0
i~6
i~4
i~5
ADDER
LESS_THAN
i96
i97
i95
i94
i90
i89
i88
i91
i87
i8
i7
i6
i9
i21
i66
i51
i50
i49
i48
i47
i41
i40
i39
i38
i37
i71
i70
i26
i25
i56
i55
i54
i53
i52
i46
i45
i44
i43
i42
i3_o
i56_OUT1
i55_OUT1
i2_o
i21_OUT1
i54_OUT1
i96_OUT1
i53_OUT1
i52_OUT1
i97_OUT1
i46_OUT1
i95_OUT1
i45_OUT1
i10_o
i94_OUT1
i44_OUT1
i90_OUT1
i43_OUT1
i89_OUT1
i42_OUT1
i9_o
i88_OUT1
i66_OUT1
i71_OUT1
i91_OUT1
i70_OUT1
i87_OUT1
i26_OUT1
i8_OUT1
i25_OUT1
i7_OUT1
i6_OUT1
i9_OUT1
i6_o

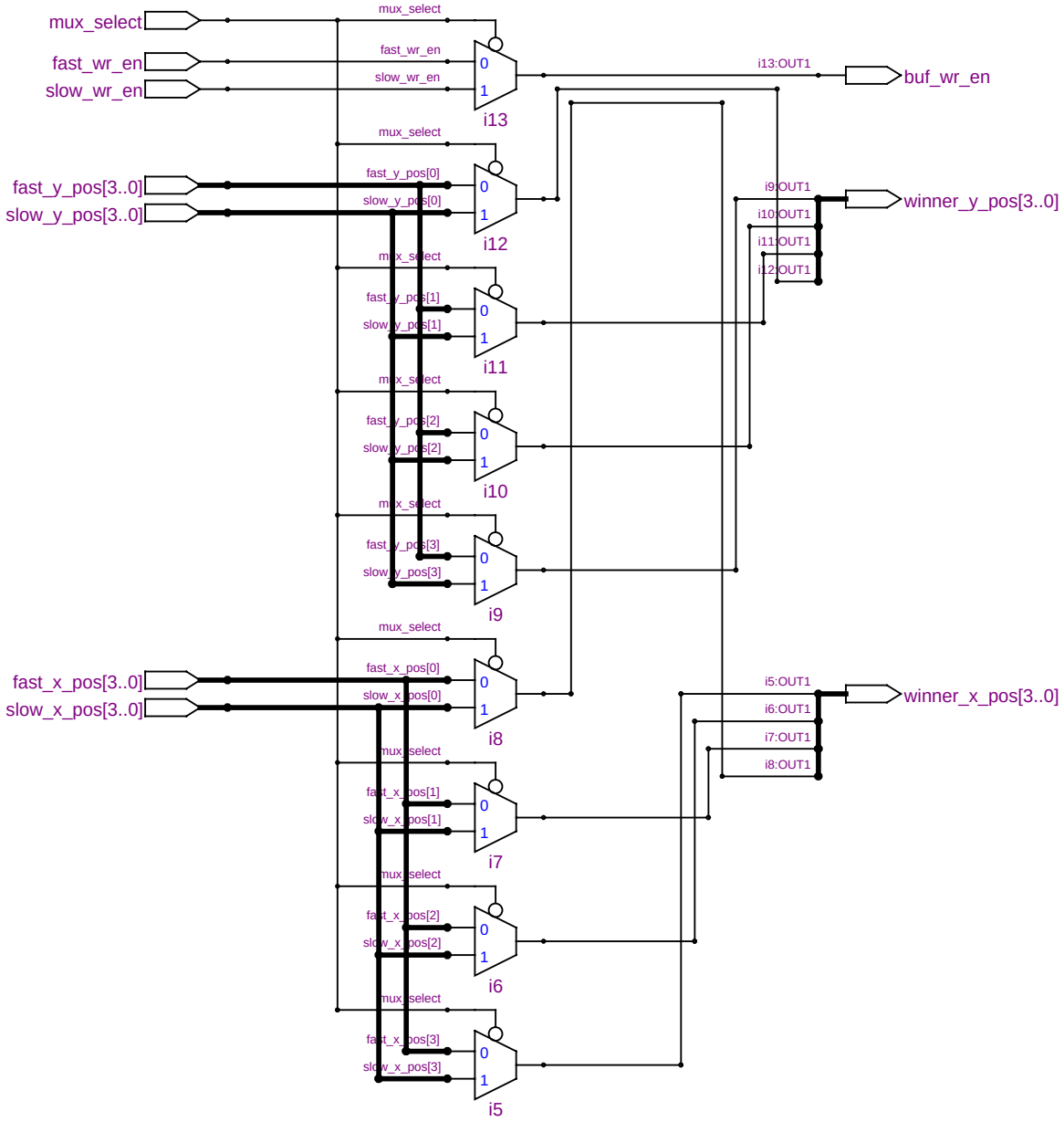
```vhdl
library IEEE;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use work.LINEPACK.all;

entity mid_point is
    port (
        clk : in std_logic;
        reset : in std_logic;
        enable : in std_logic;
        x2 : in std_logic_vector(width downto 0);
        y2 : in std_logic_vector(width downto 0);
        xy_ack : in std_logic;
        buf_ready : in std_logic;
        buf_write : out std_logic;
        done : out std_logic;
        x_addr : out std_logic_vector(width downto 0);
        y_addr : out std_logic_vector(width downto 0)
    );
end mid_point;

architecture behavioral of mid_point is
    TYPE stateT is (s1wait,s1,s2,s3a,s3b,s4,stop);
    SIGNAL state : stateT := s1wait;

    begin
    process(clk, state, reset)
        variable d : std_logic_vector((width + 1) downto 0);  -- decision variable
        variable x : std_logic_vector((width + 1) downto 0);
        variable y : std_logic_vector((width + 1) downto 0);
        variable temp : std_logic_vector((width + 1) downto 0);
        variable temp2 : std_logic_vector((width + 1) downto 0);
        variable temp3 : std_logic_vector((width + 1) downto 0);

        variable da : std_logic_vector((width + 1) downto 0);
        variable a : std_logic_vector((width + 1) downto 0);
        variable db : std_logic_vector((width + 1) downto 0);
        variable b : std_logic_vector((width + 1) downto 0);

    begin
        if(enable = '1') then
          if(rising_edge(clk) and clk = '1') then
              case state is
                  when s1wait =>
                      state <= s1;

                  when s1 =>
                  -- setting vaules
                      x := '0' & zero;
                      y := '0' & zero;

                               -- always assume starting point is
                               --   to the left and below the end point
                               --     x1 < x2 => x_inc = 1
                               --     y1 < y2 => y_inc = 1

                      if x2 >= y2 then
                      -- small slope (m <= 1)
                      -- step along x-coordinate
                          da := '0' & x2;
                          a := x;
                          db := '0' & y2;
                          b := y;
                      else
                      -- steep slope (m >1)
                      -- step along y-coordinate
                          da := '0' & y2;
```

```
69:                         a := y;
70:                         db := '0' & x2;
71:                         b := x;
72:                     end if;
73:
74:                     d := db - da;
75:
76:                     if reset = '1' then
77:                         state <= s1;
78:                     else
79:                         state <= s2;
80:                     end if;
81:
82:                 when s2 =>
83:                 -- condition for while loop
84:                     if x2 >= y2 then
85:                         -- refresh x & y
86:                         x := a;
87:                         y := b;
88:                         temp2 := '0' & x2;
89:                     else
90:                         -- refresh x & y
91:                         y := a;
92:                         x := b;
93:                         temp2 := '0' & y2;
94:                     end if;
95:
96:                     if reset = '1' then
97:                         state <= s1;
98:                     elsif (a /= temp2) then
99:                         state <= s3a;
100:                    else
101:                        state <= s4;
102:                    end if;
103:
104:                when s3a =>
105:                    -- enable write on buffer, ready x and y values
106:                    buf_write <= '1';
107:                    x_addr <= x(width downto 0);
108:                    y_addr <= y(width downto 0);
109:
110:                    if reset = '1' then
111:                        state <= s1;
112:                    -- hold values on x,y until buffer is ready
113:                    elsif buf_ready = '0' then
114:                        state <= s3a;
115:                    else
116:                        state <= s3b;
117:                    end if;
118:
119:                when s3b =>
120:                -- update x,y,d
121:                    if (d(width + 1) = '1') then
122:                    -- go horizontal(temp=x)/vertical(temp=y)
123:                       temp := db;
124:                    else
125:                    -- go diagonal
126:                       temp := db - da;
127:                       b := b + '1';
128:                    end if;
129:
130:                    a := a + '1';
131:                    d := d + (temp(width downto 0) & '0');
132:
133:                    if reset = '1' then
134:                        state <= s1;
135:                    else
136:                        state <= s2;
```

```
                    end if;

                when s4 =>
                    -- write last endpoint
                    buf_write <= '1';
                    x_addr <= x(width downto 0);
                    y_addr <= y(width downto 0);

                    if reset = '1' then
                        state <= s1;
                    -- hold last values on x, y until buffer is ready
                    elsif buf_ready = '0' then
                        state <= s4;
                    else
                        state <= stop;
                    end if;

                when stop =>
                  if ((reset = '1') or (xy_ack = '1')) then
                      state <= s1;
                      done <= '0';
                              else
                      state <= stop;
                                              done <= '1';
                                  end if;
            end case;
          end if; --end clock
        else
            done <= '0';
            buf_write <= '0';
        end if; --end enable
    end process;
end behavioral;

```

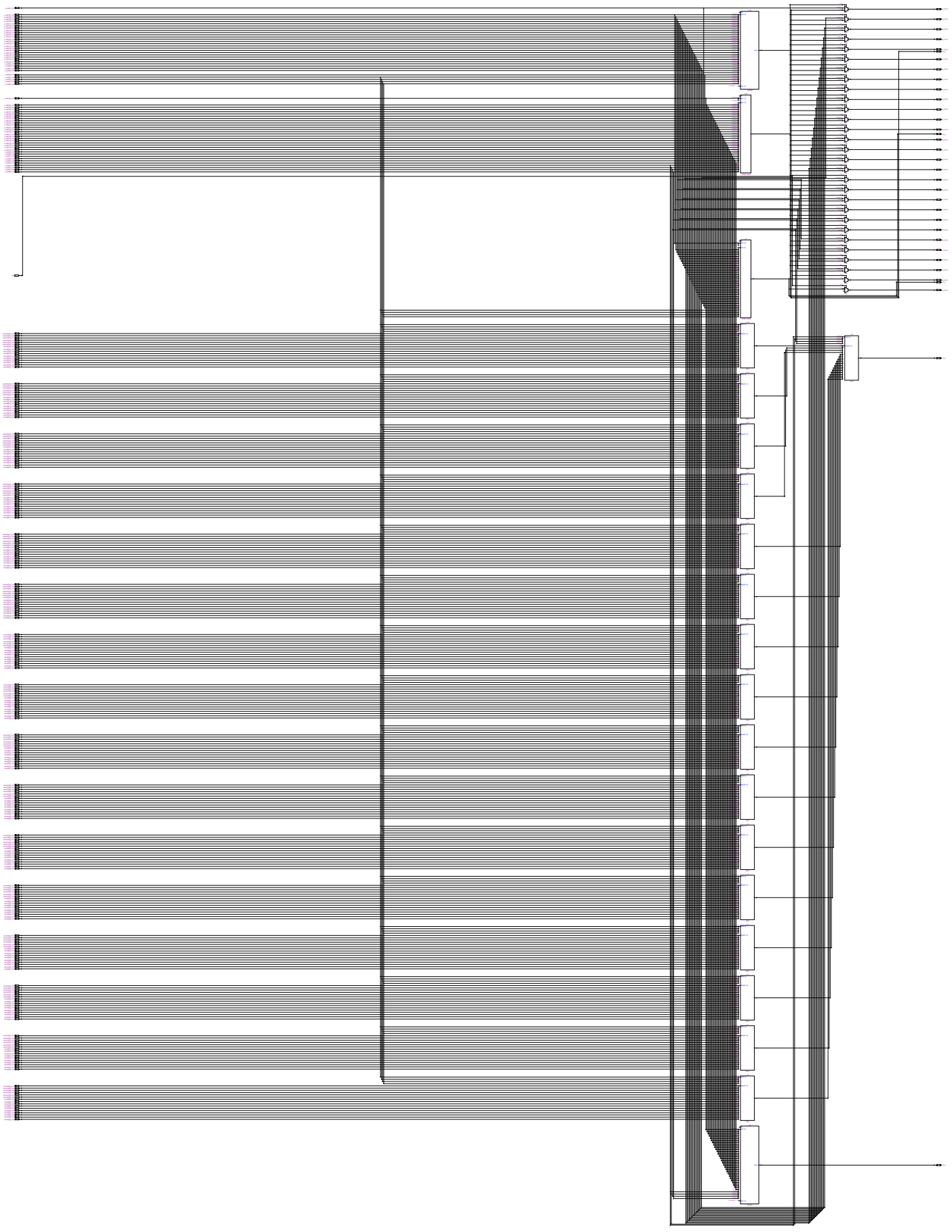


```vhdl
library IEEE;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use work.LINEPACK.all;

entity out_mux is
    port (
        mux_select : in std_logic;
        slow_x_pos : in std_logic_vector(width downto 0);
        slow_y_pos : in std_logic_vector(width downto 0);
        slow_wr_en : in std_logic;
        fast_x_pos : in std_logic_vector(width downto 0);
        fast_y_pos : in std_logic_vector(width downto 0);
        fast_wr_en : in std_logic;
        winner_x_pos : out std_logic_vector(width downto 0);
        winner_y_pos : out std_logic_vector(width downto 0);
        buf_wr_en : out std_logic
        );
end out_mux;

architecture behavioral of out_mux is
begin
    process (mux_select, slow_x_pos, slow_y_pos, slow_wr_en, fast_x_pos, fast_y_pos, fast_wr_en)
    begin
        case mux_select is
            when '0' =>
                winner_x_pos <= slow_x_pos;
                winner_y_pos <= slow_y_pos;
                buf_wr_en <= slow_wr_en;
            when '1' =>
                winner_x_pos <= fast_x_pos;
                winner_y_pos <= fast_y_pos;
                buf_wr_en <= fast_wr_en;
            when others =>
                winner_x_pos <= "1111";
                winner_y_pos <= "1111";
                buf_wr_en <= '0';
        end case;
    end process;
end behavioral;

```


mux_select
fast_wr_en
slow_wr_en
fast_y_pos[3..0]
slow_y_pos[3..0]
fast_x_pos[3..0]
slow_x_pos[3..0]
i13
i12
i11
i10
i9
i8
i7
i6
i5
i13:OUT1
buf_wr_en
i9:OUT1
i10:OUT1
i11:OUT1
i12:OUT1
winner_y_pos[3..0]
i5:OUT1
i6:OUT1
i7:OUT1
i8:OUT1
winner_x_pos[3..0]

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use work.LINEPACK.all;

entity picture_buffer is
    port (
        clk : in std_logic;
        reset : in std_logic;
        rd_en : in std_logic;               -- enable read
        wr_en : in std_logic;               -- enable write

        -- x, y position to be input by the algorithm modules
        x_cursor : in std_logic_vector(width downto 0);
        y_cursor : in std_logic_vector(width downto 0);
        ready : out std_logic;              -- signal to other modules; ready to write
        data_out : out std_logic
        );
end picture_buffer;


architecture behavioral of picture_buffer is
    type stateT is (s1wait, s1, s2, done);
    signal state : stateT := s1wait;

    type bar is array (0 to screen_dim) of std_logic;
    type foo is array (0 to screen_dim) of bar;
    signal mem : foo;

    begin
    process (clk, rd_en)
        variable x_addr : integer := 0;
        variable y_addr : integer := 0;
    begin
    if (rd_en = '1') then
        if (rising_edge(clk) and clk = '1') then
            case state is
                when s1wait =>
                    state <= s1;

                when s1 =>
                    -- set x and y address to 0
                    x_addr := 0;
                    y_addr := 0;

                    if reset = '1' then
                        state <= s1;
                    else
                        state <= s2;
                    end if;

                when s2 =>
                    data_out <= mem (y_addr) (x_addr);



                    if reset = '1' then
                        state <= s1;
                    -- if try to access row outside of screen, finished
                    elsif (screen_dim - 1) < y_addr then
                        state <= done;
                    -- if y is within screen, x is going off the screen
                    -- move down a row
                    elsif (screen_dim - 1) < x_addr then
                        y_addr := y_addr + 1;
                        x_addr := 0;
                        state <= s2;
                    -- go right one cell otherwise
```

```
                    else
                        x_addr := x_addr + 1;
                        state <= s2;
                    end if;

                when done =>
                    if reset = '1' then
                        state <= s1;
                    else
                        state <= done;
                    end if;
            end case;
        end if;
    end if;
    end process;

    -- write buffer process
    -- requires x, y coordinates from algorithm module
    process (clk, wr_en)
        variable delay : std_logic := '0';
    begin
        -- if reset not selected
        -- every clock, write a '0' (black) to memory location
        -- selected byte to be determined by the algorithm
        if (reset = '0') then
            if (rising_edge(clk) and clk = '1') then
                if (wr_en = '1') and (delay = '0') then
                    mem (CONV_INTEGER(y_cursor)) (CONV_INTEGER(x_cursor)) <= '0';
                    delay := '1';
                    ready <= '0';
                else
                    delay := '0';
                    ready <= '1';
                end if;
            end if;
        -- if reset is high "blank" out the memory
        else
            mem (0) <= X"FFFF";
            mem (1) <= X"FFFF";
            mem (2) <= X"FFFF";
            mem (3) <= X"FFFF";
            mem (4) <= X"FFFF";
            mem (5) <= X"FFFF";
            mem (6) <= X"FFFF";
            mem (7) <= X"FFFF";
            mem (8) <= X"FFFF";
            mem (9) <= X"FFFF";
            mem (10) <= X"FFFF";
            mem (11) <= X"FFFF";
            mem (12) <= X"FFFF";
            mem (13) <= X"FFFF";
            mem (14) <= X"FFFF";
            mem (15) <= X"FFFF";
        end if;
    end process;

end behavioral;
```



```
 1: ----------------------------------------------------------------------------
 2: -- hicolvga.vhd
 3: --
 4: -- Author(s):     Jorgen Peddersen and Ashley Partis
 5: -- Created:       Jan 2001
 6: -- Last Modified: Jan 2001
 7: --
 8: -- This code acts as a top level for the VGA output project.  The RAMDAC should
 9: -- be set up to program high colour dual-edged mode.  This displays a pattern
10: -- of horizontal and vertical bands of colour which can have their colour
11: -- changed and rotated around the screen.  Experimentation is the best way to
12: -- understand what each switch does.
13: ----------------------------------------------------------------------------
14: library IEEE;
15: use IEEE.std_logic_1164.all;
16: use IEEE.std_logic_unsigned.all;
17:
18: entity vga is
19:     port (
20:         clk: in STD_LOGIC;
     -- clock
21:         rstn: in STD_LOGIC;
     -- asynchronous active low reset
22:
23:         bit_data : in std_logic;                -- data from picture buffer
24:
25:         -- DIP switches 1-8
26: --          red: in STD_LOGIC_VECTOR(1 downto 0);                          -- switches to
    control the red component of the test pattern
27: --          green: in STD_LOGIC_VECTOR(1 downto 0);                        -- switches to
    control the green component of the test pattern
28: --          blue: in STD_LOGIC_VECTOR(1 downto 0);                         -- switches to
    control the blue component of the test pattern
29: --          vertRotate: in STD_LOGIC;                                              -- if
    high, the pattern will rotate vertically
30: --          horiRotate: in STD_LOGIC;                                              -- if
    high, the pattern will rotate horizontally
31:
32:         -- PB 1-2
33: --          vertDirection: in STD_LOGIC;                                   -- direction
    of vertical rotation
34: --          horiDirection: in STD_LOGIC;                                   -- direction
    of horizontal rotation
35:
36:         pixel: out STD_LOGIC_VECTOR (7 downto 0);              -- RAMDAC pixel lines
37:         blankn: out STD_LOGIC;                                             --
    RAMDAC blank signal
38:         RDn: out STD_LOGIC;
     -- RAMDAC RDn connection
39:                 WRn: out STD_LOGIC;
     -- RAMDAC WRn connection
40:                 RAMDACD: inout STD_LOGIC_VECTOR (7 downto 0);    -- RAMDAC data lines
41:                 RS: inout STD_LOGIC_VECTOR (2 downto 0);                   -- RAMDAC RS
    lines
42:         hsync: out STD_LOGIC;                                              --
    horizontal sync for monitor
43:         vsync: out STD_LOGIC;                                              --
    vertical sync for monitor
44:         triste: out STD_LOGIC;                                             --
    signal to tristate ethernet PHY
45:         rramce: out STD_LOGIC;                                             --
    right ram chip enable
46:         pixelclk: out STD_LOGIC                                            --
    RAMDAC pixel clock
47:     );
48: end vga;
49:
50: architecture vga_arch of vga is
```

```
51:
52: -- control VGA signals
53: component vgacore
54:         generic (
55:                 H_SIZE : integer;
     -- horizontal size of input image, MAX 800
56:                 V_SIZE : integer
     -- vertical size of input image, MAX 600
57:         );
58:         port
59:         (
60:                 reset: in std_logic;                                    --
    asynchronous active low reset
61:                 clock: in std_logic;                                    --
    clock
62:                 hsyncb: buffer std_logic;                               --
    horizontal (line) sync
63:                 vsyncb: out std_logic;                                  --
    vertical (frame) sync
64:                 latch: out STD_LOGIC;                                   --
    latches new rgb value
65:                 enable: out STD_LOGIC;                                  --
    enable/ground RGB output lines
66:                 hloc: out std_logic_vector(9 downto 0);         -- horizontal address
    to be decoded for video RAM
67:                 vloc: out std_logic_vector(9 downto 0)          -- vertical address to
    be decoded for video RAM
68:         );
69: end component;
70:
71: -- Program the RAMDAC
72: component prgramdacver2
73:     port (
74:         clk: in STD_LOGIC;                                              --
    Clock
75:         rstn: in STD_LOGIC;                                             --
    Asynchronous active low reset
76:         start: in STD_LOGIC;                                    -- Start signal
77:         done: out STD_LOGIC;                                    -- Asserted
    when programming is finished
78:         WRn: out STD_LOGIC;                                             --
    Write line to RAMDAC (active low)
79:         RDn: out STD_LOGIC;                                             --
    Read line to RAMDAC (active low)
80:         RS: inout STD_LOGIC_VECTOR (2 downto 0);        -- Register select lines to
    the RAMDAC
81:         data: inout STD_LOGIC_VECTOR (7 downto 0)       -- Bidirectional data line to
    RAMDAC
82:     );
83: end component;
84:
85: -- State signals
86: type STATETYPE is (stReset, stWaste1, stWaste2, stWaste3, stWait, stForever);
87: signal presState: STATETYPE;
88:
89: -- signals so that hsync and vsync can be read
90: signal hsyncInt : STD_LOGIC;
91: signal vsyncInt : STD_LOGIC;
92:
93: -- signals to cue different processes
94: signal startProg : STD_LOGIC;
95: signal startVGA : STD_LOGIC;
96: signal resetVGA : STD_LOGIC;
97: signal done : STD_LOGIC;
98:
99: -- signals to generate test pattern
100: signal hloc : std_logic_vector(9 downto 0);                           -- horizontal
    location of each pixel
```

```
101: signal vloc : std_logic_vector(9 downto 0);                              -- vertical
     location of each pixel
102: -- signal vertoffset : std_logic_vector(8 downto 0);           -- offset for vertical
     rotation
103: -- signal horioffset : std_logic_vector(8 downto 0);           -- offset for
     horizontal rotation
104: signal vertVal : std_logic_vector(8 downto 0);                 -- adds offset to
     pixel location
105: signal horiVal : std_logic_vector(8 downto 0);                 -- adds offset to
     pixel location
106: signal pixelData : STD_LOGIC_VECTOR(15 downto 0);              -- colour to write for
     each pixel
107:
108: begin
109:         -- VGA controller
110:         cycler : vgacore
111:         generic map (
112:                 H_SIZE => 16,
113:                 V_SIZE => 16
114:         )
115:         port map(
116:                 reset => resetVGA,
117:                 clock => clk,
118:                 hsyncb => hsyncInt,
119:                 vsyncb => vsyncInt,
120:                 latch => open,
121:                 enable => blankn,
122:                 hloc => hloc,
123:                 vloc => vloc
124:         );
125:
126:     -- RAMDAC programmer
127:     RAMDACprog : prgramdacver2 port map (
128:         clk => clk,
129:         rstn => rstn,
130:         start => startProg,
131:         done => done,
132:         WRn => WRn,
133:         RDn => RDn,
134:         RS => RS,
135:         data => RAMDACD
136:     );
137:
138:         -- This is a simple mealy state machine that cues the VGA controller
139:         -- when the RAMDAC is finished programming
140:         process(clk, rstn)
141:         begin
142:                 if rstn = '0' then
143:                         presState <= stReset;
144:                 elsif clk'event AND clk = '1' then
145:                         case presState is
146:                                 when stReset =>
147:                                         presState <= stWaste1;
148:                                 when stWaste1 =>
149:                                         presState <= stWaste2;
150:                                 when stWaste2 =>
151:                                         presState <= stWaste3;
152:                                 when stWaste3 =>
153:                                         presState <= stWait;
154:                                 when stWait =>
155:                                         if done = '0' then
156:                                                 presState <= stWait;
157:                                         else
158:                                                 presState <= stForever;
159:                                         end if;
160:                                 when stForever =>
161:                                         presState <= stForever;
162:                         end case;
```

```
163:                 end if;
164:         end process;
165:
166:         process(presState)
167:         begin
168:                 case presState is
169:                         when stReset =>
170:                                 startProg <= '1';
171:                                 startVGA <= '0';
172:                         when stWaste1 =>
173:                                 startProg <= '1';
174:                                 startVGA <= '0';
175:                         when stWaste2 =>
176:                                 startProg <= '1';
177:                                 startVGA <= '0';
178:                         when stWaste3 =>
179:                                 startProg <= '1';
180:                                 startVGA <= '0';
181:                         when stWait =>
182:                                 startProg <= '0';
183:                                 startVGA <= '0';
184:                         when stForever =>
185:                                 startProg <= '0';
186:                                 startVGA <= '1';
187:                 end case;
188:         end process;
189:
190:         -- calculate offsets for data that is rotating.  Value will update whenever
191:         -- a new page is called for as it is clocked by vsync.
192: --      process(rstn, vsyncInt)
193: --      begin
194: --              if rstn = '0' then
195: --                      vertoffset <= "000000000";
196: --                      horioffset <= "000000000";
197: --              elsif vsyncInt'event and vsyncInt = '1' then
198: --                      -- update vertical counter if required
199: --                      if vertRotate = '1' then
200: --                              if vertDirection = '0' then
201: --                                      vertoffset <= vertoffset + 1;
202: --                              else
203: --                                      vertoffset <= vertoffset - 1;
204: --                              end if;
205: --                      end if;
206: --                      -- update horizontal counter if required
207: --                      if horiRotate = '1' then
208: --                              if horiDirection = '0' then
209: --                                      horioffset <= horioffset + 1;
210: --                              else
211: --                                      horioffset <= horioffset - 1;
212: --                              end if;
213: --                      end if;
214: --              end if;
215: --      end process;
216:
217: -- Synchronize the horizontal and vertical values with the clock.
218: process(clk,rstn)
219: begin
220:         if rstn = '0' then
221:                 horival <= (others => '0');
222:                 vertval <= (others => '0');
223:         elsif clk'event and clk = '0' then
224: --              if (CONV_INTEGER(vloc) mod 2) = 0 then
225: --                      horiVal <= "000000000";
226: --                      vertVal <= "000000000";
227: --              else
228: --                      horiVal <= vloc(8 downto 0) + horioffset;
229: --                      vertVal <= hloc(8 downto 0) + vertoffset;
230:                         horiVal <= vloc(8 downto 0);
```

```
231:                         vertVal <= hloc(8 downto 0);
232: --              end if;
233:         end if;
234: end process;
235:
236:
237: process(clk, bit_data)
238: begin
239:     if bit_data = '0' then
240:         pixelData <= X"0000";
241:     else
242:         pixelData <= X"FFFF";
243:     end if;
244: end process;
245:
246:
247: -- Calculate pixelData.  This is the colour programmed into each pixel.
248: -- By choosing different combinations of the three component colours (red, green and
     blue)
249: -- different patterns can be formed.  For each colour, the following
250: -- patterns can be chosen from:
251: --         Bit 1   Bit 0         Description
252: --              0              0              Colour does not contribute to any of
     the pixels on the screen
253: --              0              1              Colour forms a test pattern of 32
     scaled vertical bands
254: --              1              0              Colour forms a test pattern of 32
     scaled horizontal bands
255: --              1              1              Every pixel on the screen is given the
     full value of the colour
256: -- The three colours can all be used to control the colour components of each pixel
257: -- so it is possible to combine the values and have something like:
258: -- Full blue component, horizontal green bands and vertical red bands, or
259: -- No red component, horizontal blue bands and horizontal green bands
260: -- (this would appear as horizontal cyan bands) etc.
261:
262: -- pixelData(15) <= '0';
263: -- process(clk, resetVGA)
264: -- begin
265: --      if resetVGA = '0' then
266: --      pixelData(14 downto 10) <= (others => '0');
267: --      elsif clk'event and clk = '1' then
268: --              case red is                                                    --
     choose contribution of red component
269: --                      when "00" =>
270: --                              pixelData(14 downto 10) <=      "00000";
271: --                      when "01" =>
272: --                              pixelData(14 downto 10) <=      horiVal(8 downto 4);
273: --                      when "10" =>
274: --                              pixelData(14 downto 10) <=      vertVal(8 downto 4);
275: --                      when others =>
276: --                              pixelData(14 downto 10) <=      "11111";
277: --              end case;
278: --              case green is                                                  --
     choose contribution of green component
279: --                      when "00" =>
280: --                              pixelData(9 downto 5) <=        "00000";
281: --                      when "01" =>
282: --                              pixelData(9 downto 5) <=        horiVal(8 downto 4);
283: --                      when "10" =>
284: --                              pixelData(9 downto 5) <=        vertVal(8 downto 4);
285: --                      when others =>
286: --                              pixelData(9 downto 5) <=        "11111";
287: --              end case;
288: --              case blue is                                                   --
     choose contribution of blue component
289: --                      when "00" =>
290: --                              pixelData(4 downto 0) <=        "00000";
```

```
291: --                      when "01" =>
292: --                              pixelData(4 downto 0) <=        horiVal(8 downto 4);
293: --                      when "10" =>
294: --                              pixelData(4 downto 0) <=        vertVal(8 downto 4);
295: --                      when others =>
296: --                              pixelData(4 downto 0) <=        "11111";
297: --              end case;
298: --      end if;
299: -- end process;
300:
301: -- handle dual-edged clock to give correct data to the RAMDAC
302: pixel <= pixelData(7 downto 0) when clk = '1' else pixelData(15 downto 8);
303:
304: -- only start VGA after RAMDAC has been programmed
305: resetVGA <= rstn AND startVGA;
306:
307: -- pass the outputs out
308: hsync <= hsyncINT;
309: vsync <= vsyncINT;
310:
311: -- Provide 50MHz pixel clock
312: pixelclk <= clk;
313:
314: -- turn off the ethernet outputs and the right SRAM bank to avoid contention on the
     lines
315: triste <= '1';
316: rramce <= '1';
317:
318: end vga_arch;
```

```
 1: --------------------------------------------------------------------------------
 2: -- prgramdac.vhd
 3: --
 4: -- Author(s):     Ashley Partis and Jorgen Peddersen
 5: -- Created:       Dec 2000
 6: -- Last Modified: Jan 2001
 7: --
 8: -- This code programmes the RAMDAC on the XSV-300 board with data for either
 9: -- high-colour mode or a simple colour map.
10: --
11: --------------------------------------------------------------------------------
12:
13: library IEEE;
14: use IEEE.std_logic_1164.all;
15: use IEEE.std_logic_unsigned.all;
16:
17: entity prgramdacver2 is
18:     port (
19:         clk: in STD_LOGIC;
             -- Clock
20:         rstn: in STD_LOGIC;
             -- Asynchronous active low reset
21:         start: in STD_LOGIC;
     -- Start signal
22:         done: out STD_LOGIC;
     -- Asserted when programming is finished
23:         WRn: out STD_LOGIC;
             -- Write line to RAMDAC (active low)
24:         RDn: out STD_LOGIC;
             -- Read line to RAMDAC (active low)
25:         RS: inout STD_LOGIC_VECTOR (2 downto 0);                    -- Register
    select lines to the RAMDAC
26:         data: inout STD_LOGIC_VECTOR (7 downto 0)                   --
    Bidirectional data line to RAMDAC
27:     );
28: end prgramdacver2;
29:
30: architecture prgramdacver2_arch of prgramdacver2 is
31: -- signal declarations
32: -- FSM states for the main mealy FSM
33: type STATETYPE is (stIdle, stWrite, stWrCycle, stNextWrite
34:                 --, stSetupRGB, stWriteRGB, stNextWriteRGB               -- uncomment
    these states for colour map
35:                 );
36: signal presState: STATETYPE;
37: signal nextState: STATETYPE;
38:
39: -- hardcoded values for initialising the RAMDAC to the values we desire
40: -- location 10 down to 8 is the RS values
41: type TWODIMARRAYDAC is array (0 to 5) of STD_LOGIC_VECTOR (7 downto 0);
42: type TWODIMARRAYRS is array (0 to 5) of STD_LOGIC_VECTOR (2 downto 0);
43:
44: constant initDAC: TWODIMARRAYDAC:=
45:   -- hard code initial control register programming values
46: ( -- DAC(76543210)
47:                 "10000001",                     -- Command reg A gets $81 for high
    colour dual edged mode
48: --              "10100001",                     -- Command reg A gets $A1 for high
    colour single edged mode
49: --              "00000001",                     -- Command reg A gets $01 for colour
    map
50:                 "00000000",                     -- Pallette address reg gets $00
51:                 "11111111",                     -- Read mask reg gets $FF
52:                 "00000010",                     -- Pallette address reg gets $02
53:                 "00000010",                     -- Command reg B gets $02
54:                 "00000000"                      -- Pallette address reg gets $00
55: );
56:
```

```
57: constant initRS: TWODIMARRAYRS:=
58: (  -- RS(210)
59:                 "110",                          -- RS gets Command reg A
60:                 "000",                          -- RS gets Pallette address reg
61:                 "010",                          -- RS gets Read mask reg
62:                 "000",                          -- RS gets Pallette address reg
63:                 "010",                          -- RS gets Command reg B
64:                 "000"                           -- RS gets Pallette address reg
65: );
66:
67: -- initCnt is an integer to index the constant two dimensional arrays initDAC and
    initRS,
68: -- and a signal to increment initCnt
69: signal initCnt: INTEGER range 0 to 5;
70: signal increment: STD_LOGIC;
71:
72: -- signals to create a 12.5MHz clock from the 50MHz input clock to the entity
73: signal divclk: STD_LOGIC;
74: signal gray_cnt: STD_LOGIC_VECTOR (1 downto 0);
75:
76:
77: -- create signals so the data and RS lines can be used as tristate buffers
78: -- this is important as they share lines with the ethernet PHY
79: signal prgData: STD_LOGIC_VECTOR (7 downto 0);
80: signal prgRS: STD_LOGIC_VECTOR (2 downto 0);
81: signal latchData: STD_LOGIC;
82: signal latchRS: STD_LOGIC;
83:
84: -- these are for programming the colourmap of the RAMDAC - to program all 256 lots
85: -- of 3 byte sets of RGB data - uncomment for colour map
86: -- also a signal to increment colourCnt to avoid a race condition
87: --signal prgRGB: STD_LOGIC_VECTOR (7 downto 0);
88: --signal colourCnt: STD_LOGIC_VECTOR (1 downto 0);
89: --signal incColourCnt: STD_LOGIC;
90:
91: begin
92:
93: -- clock divider by 4 to for a slower clock to avoid timing violations
94: -- uses grey code for minimized logic
95: A:      process (rstn, clk)
96:         begin
97:                 if rstn = '0' then
98:                         gray_cnt <= "00";
99:                 elsif clk'EVENT and clk = '1' then
100:                        case(gray_cnt) is
101:                                when "00" => gray_cnt <= "01";
102:                                when "01" => gray_cnt <= "11";
103:                                when "11" => gray_cnt <= "10";
104:                                when "10" => gray_cnt <= "00";
105:                                when others => gray_cnt <= "00";
106:                        end case;
107:                end if;
108:        end process;
109:
110: -- assign the clock that this entity runs off
111:         divclk <= gray_cnt(1);
112:
113: -- read isn't needed, tie high
114:         RDn <= '1';
115:
116: -- main clocked process
117: B:      process (rstn, divclk)
118:         begin
119:                 if rstn = '0' then
120:                         presState <= stIdle;
121:                         initCnt <= 0;
122: -- add these signals for colour map
123: --                      colourCnt <= (others => '0');
```

```
124: --                      prgRGB <= (others => '0');
125:                 elsif divclk'event and divclk = '1' then
126:                         presState <= nextState;
127:                         -- increment initCnt
128:                         if increment = '1' then
129:                                 -- overflow initCnt when it hits 5 as integers don't
     overflow
130:                                 if initCnt < 5 then
131:                                         initCnt <= initCnt + 1;
132:                                 else
133:                                         initCnt <= 0;
134:                                 end if;
135:                         end if;
136: -- add these signals for colour map
137: --                      if incColourCnt = '1' then
138: --                              if colourCnt = "10" then
139: --                                      colourCnt <= "00";
140: --                                      prgRGB <= prgRGB + 1;
141: --                              else
142: --                                      colourCnt <= colourCnt + 1;
143: --                              end if;
144: --                      end if;
145:                 end if;
146:         end process;
147:
148: -- Main FSM process
149: C:      process (presState, start, initCnt)
150:         begin
151:                 -- default signals and outputs for each FSM state
152:                 -- note that the latch data and rs signals are defaulted to 1, so are
153:                 -- only 0 in the idle state
154:                 WRn <= '1';
155:                 increment <= '0';
156: --              incColourCnt <= '0';
157:                 prgData <= (others => '0');
158:                 prgRS <= "001";
159:                 latchData <= '1';
160:                 latchRS <= '1';
161:                 done <= '0';
162:
163:                 case presState is
164:                         when stIdle =>
165:                                 -- wait for start signal from another process
166:                                 if start = '1' then
167:                                         nextState <= stWrite;
168:                                         -- setup for the first write to the RAMDAC for
     use by setting the register select
169:                                         -- lines and the data lines
170:                                         prgRS <= initRS(initCnt);
171:                                         prgData <= initDAC(initCnt);
172:                                 else
173:                                         nextState <= stIdle;
174:                                         latchData <= '0';
175:                                         latchRS <='0';
176:                                 end if;
177:
178:                         when stWrite =>
179:                                 -- hold the register select and data lines for the
     write cycle
180:                                 -- and set the active low write signal
181:                                 nextState <= stWrCycle;
182:                                 prgRS <= initRS(initCnt);
183:                                 prgData <= initDAC(initCnt);
184:                                 WRn <= '0';
185:
186:                         when stWrCycle =>
187:                                 -- continue if all 5 registers that needed programming
     have been written to
```

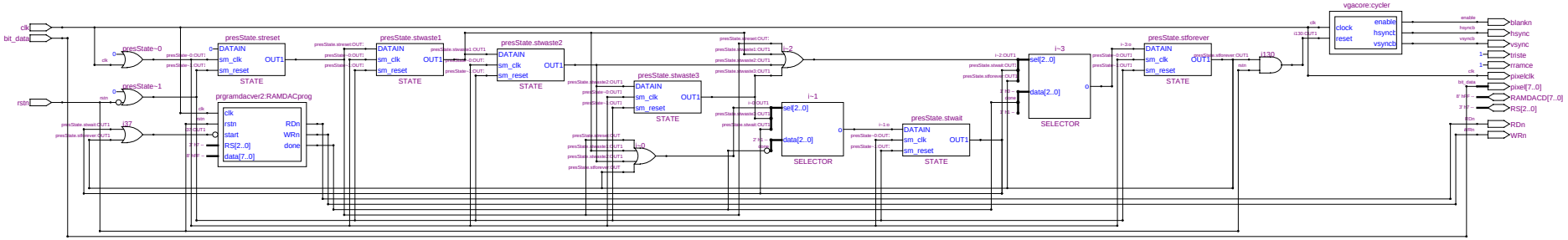
```
188:                                 if initCnt = 5 then
189:                                         nextState <= stIdle;
190:                                         done <= '1';
191: -- comment the two lines above and uncomment the one below for a colour map
192: --                                       nextState <= stSetupRGB;

193:                                 -- continue writing to the registers
194:                                 else
195:                                         nextState <= stNextWrite;
196:                                 end if;
197:                                 -- hold the data to be sure the hold times aren't
     violated
198:                                 prgRS <= initRS(initCnt);
199:                                 prgData <= initDAC(initCnt);
200:                                 -- increment initCnt to program the next register
201:                                 increment <= '1';
202:
203:                         when stNextWrite =>
204:                                 nextState <= stWrite;
205:                                 -- setup for the next write cycle
206:                                 prgRS <= initRS(initCnt);
207:                                 prgData <= initDAC(initCnt);
208:
209:                         -- start programming the RGB values to the colour map
210:                         -- note RS is defaulted to 001, which is what is required
211:                         -- for programming the colour map
212:                         -- These steps program the RAMDACs colour map.  To set the
     colours
213:                         -- see the if statement below the end of the case statement
214: --                      when stSetupRGB =>
215: --                              nextState <= stWriteRGB;
216: --                      when stWriteRGB =>
217: --                              nextState <= stNextWriteRGB;
218: --                              WRn <= '0';
219: --                      when stNextWriteRGB =>
220: --                              -- if all 256 sets of 3 byte RGB values are programmed,
      then go back
221: --                              -- to the idle state and assert done
222: --                              if prgRGB = "11111111" and colourCnt = "10" then
223: --                                      nextState <= stIdle;
224: --                                      done <= '1';
225: --                              else
226: --                                      nextState <= stSetupRGB;
227: --                              end if;
228: --                              incColourCnt <= '1';
229:                 end case;
230:
231:                 -- the following statement will program the RAMDACs colour map.  To
     change the
232:                 -- colours it programs with, comment out different lines
233: --              if presState = stSetupRGB or presState = stWriteRGB or presState =
     stNextWriteRGB then
234: --                      if colourCnt = "00" then                                -- Red
     component
235: --                              prgData <= prgRGB;
          -- Full Red scaling
236: --                              prgData <= prgRGB(7 downto 5) & "11111";-- 3:3:2 Red
     scaling
237: --                              prgData <= (others => '0');
      -- No Red component
238: --                      elsif colourCnt = "01" then                             --
     Green component
239: --                              prgData <= prgRGB;
          -- Full Green scaling
240: --                              prgData <= prgRGB(4 downto 2) & "11111";-- 3:3:2 Green
     scaling
241: --                              prgData <= (others => '0');
      -- No Green component
```

```
242: --                         else
             -- Blue component
243: --                                 prgData <= prgRGB;
             -- Full Blue scaling
244: --                                 prgData <= prgRGB(1 downto 0) & "11111";-- 3:3:2 Blue
     scaling
245: --                                 prgData <= (others => '0');
      -- No Blue component
246: --                         end if;
247: -- Leave the following line commented if using the if statement above
248: --                         prgData <= prgRGB;
             -- Full Grey scaling
249: --                 end if;
250:         end process;
251: --
252: -- assign data and RS prgData and prgRS repsectively when they need to be latched
253: -- otherwise keep them at high impedance to create a tri state buffer
254:         data <= prgData when latchData = '1' else (others => 'Z');
255:         RS <= prgRS when latchRS = '1' else (others => 'Z');
256:
257: end prgramdacver2_arch;
```

```
1: ----------------------------------------------------------------------------
2: -- vgacore.vhd
3: --
4: -- Author(s):     Ashley Partis and Jorgen Peddersen
5: -- Based largely on a version on the XESS (www.xess.com) page.  Thanks to XESS.
6: -- Created:       Jan 2001
7: -- Last Modified: Feb 2001
8: --
9: -- Creates VGA timing signals to a monitor, timings are currently for 72Hz @
10: -- 800 * 600.  To change the resolution or refresh rate, change the value of
11: -- the constants and the generics to whatever is desired.  Changing the
12: -- resolution and / or refresh also means the clock speed may have to change
13: -- (currently based off a 50MHz clock).
14: --
15: ----------------------------------------------------------------------------
16:
17: library IEEE;
18: use IEEE.std_logic_1164.all;
19: use IEEE.std_logic_unsigned.all;
20:
21: entity vgacore is
22:         generic (
23:                 H_SIZE : integer := 800;                                 --
    horizontal size of input image, MAX 800
24:                 V_SIZE : integer := 600                                 --
    vertical size of input image, MAX 600
25:         );
26:         port
27:         (
28:                 reset: in std_logic;                                    --
    asynchronous active low reset
29:                 clock: in std_logic;                                    --
    clock
30:                 hsyncb: buffer std_logic;                               --
    horizontal (line) sync
31:                 vsyncb: out std_logic;                                  --
    vertical (frame) sync
32:                 latch: out STD_LOGIC;                                   --
    latches new rgb value
33:                 enable: out STD_LOGIC;                                  --
    enable/ground RGB output lines
34:                 hloc: out std_logic_vector(9 downto 0);         -- horizontal address
    to be decoded for video RAM
35:                 vloc: out std_logic_vector(9 downto 0)          -- vertical address to
    be decoded for video RAM
36:         );
37: end vgacore;
38:
39: architecture vgacore_arch of vgacore is
40:
41: -- one of the sync signals
42: --
43: --                     |<------- Active Region ------------>|<----------- Blanking
    Region ---------->|
44: --                     |           (Pixels)                 |
                |
45: --                     |                                    |
                |
46: --                     |                                    |
                |
47: --         ------------+-------------- ... -----------------+-------------                -
    -------------+--------
48: --         |           |                                    |            |
    |           |
49: --         |           |                                    |<--Front    |<---Sync
    |<---Back     |
50: --         |           |                                    |    Porch-->|     Time---
    >|    Porch--->|
```

```
51: --  -------           |                                   |           ---------------
              |
52: --                    |                                   |
              |
53: --                    |<--------------------------------- Period ----------------------
    ------------>|
54: --
55: -- horizontal timing signals
56: constant H_PIXELS: INTEGER:= H_SIZE;
57: constant H_FRONTPORCH:  INTEGER:= 56 + (800 - H_SIZE) / 2;
58: constant H_SYNCTIME: INTEGER:= 120;
59: constant H_BACKPORCH: INTEGER:= 63 + (800 - H_PIXELS) / 2;
60: constant H_PERIOD: INTEGER:= H_SYNCTIME + H_PIXELS + H_FRONTPORCH + H_BACKPORCH;
61:
62: -- vertical timing signals
63: constant V_LINES: INTEGER:= V_SIZE;
64: constant V_FRONTPORCH: INTEGER:= 37 + (600 - V_SIZE) / 2;
65: constant V_SYNCTIME: INTEGER:= 6;
66: constant V_BACKPORCH: INTEGER:= 23 + (600 - V_SIZE) / 2;
67: constant V_PERIOD: INTEGER:= V_SYNCTIME + V_LINES + V_FRONTPORCH + V_BACKPORCH;
68:
69: signal hcnt: std_logic_vector(10 downto 0);
             -- horizontal pixel counter
70: signal vcnt: std_logic_vector(9 downto 0);
             -- vertical line counter
71:
72: begin
73:
74: -- control the reset, increment and overflow of the horizontal pixel count
75: A: process(clock, reset)
76: begin
77:         -- reset asynchronously clears horizontal counter
78:         if reset = '0' then
79:                 hcnt <= (others => '0');
80:         -- horiz. counter increments on rising edge of dot clock
81:         elsif (clock'event and clock = '1') then
82:                 -- horiz. counter restarts after the horizontal period (set by the
    constants)
83:                 if hcnt < H_PERIOD then
84:                         hcnt <= hcnt + 1;
85:                 else
86:                         hcnt <= (others => '0');
87:                 end if;
88:         end if;
89: end process;
90:
91: -- control the reset, increment and overflow of the vertical line counter after every
    horizontal line
92: B: process(hsyncb, reset)
93: begin
94:         -- reset asynchronously clears line counter
95:         if reset='0' then
96:                 vcnt <= (others => '0');
97:         -- vert. line counter increments after every horiz. line
98:         elsif (hsyncb'event and hsyncb = '1') then
99:                 -- vert. line counter rolls-over after the set number of lines (set by
    the constants)
100:                if vcnt < V_PERIOD then
101:                        vcnt <= vcnt + 1;
102:                else
103:                        vcnt <= (others => '0');
104:                end if;
105:        end if;
106: end process;
107:
108: -- set the horizontal sync high time and low time according to the constants
109: C: process(clock, reset)
110: begin
```

```
111:         -- reset asynchronously sets horizontal sync to inactive
112:         if reset = '0' then
113:                 hsyncb <= '1';
114:         -- horizontal sync is recomputed on the rising edge of every dot clock
115:         elsif (clock'event and clock = '1') then
116:                 -- horiz. sync is low in this interval to signal start of a new line
117:                 if (hcnt >= (H_FRONTPORCH + H_PIXELS) and hcnt < (H_PIXELS +
    H_FRONTPORCH + H_SYNCTIME)) then
118:                         hsyncb <= '0';
119:                 else
120:                         hsyncb <= '1';
121:                 end if;
122:         end if;
123: end process;
124:
125: -- set the vertical sync high time and low time according to the constants
126: D: process(hsyncb, reset)
127: begin
128:         -- reset asynchronously sets vertical sync to inactive
129:         if reset = '0' then
130:                 vsyncb <= '1';
131:         -- vertical sync is recomputed at the end of every line of pixels
132:         elsif (hsyncb'event and hsyncb = '1') then
133:                 -- vert. sync is low in this interval to signal start of a new frame
134:                 if (vcnt >= (V_LINES + V_FRONTPORCH) and vcnt < (V_LINES +
    V_FRONTPORCH + V_SYNCTIME)) then
135:                         vsyncb <= '0';
136:                 else
137:                         vsyncb <= '1';
138:                 end if;
139:         end if;
140: end process;
141:
142: -- whether it should latch the current data or not
143: -- (always with a 50MHz clock - blanking is handled on the RAMDAC by asserting a
    signal)
144: latch <= NOT reset;
145:
146: -- asserts the blaking signal (active low)
147: E: process (clock)
148: begin
149:         if clock'EVENT and clock = '1' then
150:                 -- if we are outside the visible range on the screen then tell the
    RAMDAC to blank
151:                 -- in this section by putting enable low
152:                 if hcnt >= H_PIXELS or vcnt >= V_LINES then
153:                         enable <= '0';
154:                  else
155:                         enable <= '1';
156:                  end if;
157:         end if;
158: end process;
159:
160: -- The video RAM address is built from the lower 9 bits of the vertical
161: -- line counter and bits 7-2 of the horizontal pixel counter.
162: -- Allows easy access for the current address of the current pixel in RAM
163: H:
164: hloc <= hcnt(9 downto 0);
165: vloc <= vcnt(9 downto 0);
166:
167: end vgacore_arch;
```


clk
bit_data
rstn
presState~0
presState~1
presState.streset
presState.stwaste1
presState.stwaste2
presState.stwaste3
presState.stwait
presState.stforever
STATE
programdacver2:RAMDACprog
clk
rstn
start
RS[2..0]
data[7..0]
RDn
WRn
done
i~1
i~2
i~3
i~0
sel[2..0]
data[2..0]
SELECTOR
i130
vgacore:cycler
clock
reset
enable
hsyncb
vsyncb
blankn
hsync
vsync
inste
framce
pixelclk
pixel[7..0]
RAMDACD[7..0]
RS[2..0]
RDn
WRn