LED7
POWER
D2
LED1
LED2
SW2
SW3
SW1
JP11
R33
JP2
U15
U14
IDT71V016
SA12PH
H0136P
U11
U9
U8
J1
U6
C5
D3
SW8
SW7
SW6
U3
VER 1.1
48/01
ALTERA
EPM7064AE
TC44-7
T 9AF40
0101C
U4
JP3
LM2676
S-3.3
SW10
C1
JP12
JP13
R23
J3
C22
C7
C21
RP8
PMC JTAG
C8
C9
D4
D5
C20
C19
R19
U12
PMC JN3
PMC JN4
U13
R24
143
JP14
C55
ALTERA
APEX
JP8
RT3
RB2
U5
Y1
C17
U10
TP3
144
C65
U2
R20
U7
C4
JP9
JP10
TP4

Abstract

This report describes our work as part of the project oriented course “Advanced Digital Design” taught at EPFL. We designed a VGA controller and three demonstration applications. The first application is very simple and displays a “snow” effect based on random pixels (something resembling what a TV set displays when there is no signal). The second application allows us to display a simple color image on the screen. The third application runs the famous game of life. We also designed a unit for bi-directional communication with a PC using the serial port.

The goal of this course was to provide us with an overview of what the challenges are when developing a large hardware project.

We synthesized our project for an Altera VHDL (FPGA) board and connected a normal computer monitor to show our results.

Introduction

The graphics card is the hardware that allows a computer to display images on a screen. The screen is represented in memory as a large set of pixels. Each pixel has a red, green and blue component. The graphics card is responsible to transmit these pixels through analog signals. VGA is a standard used to transmit this data to the screen. It was marketed by IBM in 1987. The VGA specifications are a screen size of 640 by 480 pixels and the colors are encoded as 0V – 0.7V (actually the specification is much more complex, describes palette modes, etc.). Although the VGA standard is very low resolution, it is the common denominator of all graphics card.

For detailed information on the VGA controller, please refer to the course’s handout (VGA.pdf).

The famous game of life is a cellular automaton (invented in 1970 by John Conway). It is a topic broadly studies in computer sciences and mathematics. We will only give a vague explanation on how the game works, without going into the details. The game of life is “played” on a rectangle board, where each cell can be either alive (set to 1) or dead (set to 0). Every cycle, each cell changes it’s state depending on it’s current value and the state of it’s 8 neighboring cells. The rules of the game state that if a cell is alive and has more than 3 neighbors or less than 2 neighbors then it will die, otherwise it will survive. If the cell is dead and has 3 neighbors, then it will take birth. The fun part in the game of life is to find initial patterns that do interesting things. Once the initial pattern is loaded, the user doesn’t interact with the world (he just looks at the evolution of his pattern).

For more detailed information on the game of life, refer to the course’s handout (GameOfLife.pdf).

In order to create the initial pattern, we wrote a Java user interface that communicates with the FPGA through the serial port (RS232).

See UART.pdf.

Development environment

We used the hardware description language VHDL to program all the hardware components. We used ModelSim to compile and simulate our code. We then used Leonardo to synthesize the code and Quartus to perform the routing and FPGA programming. We suggest using Teraterm as the terminal emulator to communicate with the card, as we had some problems with Hyperterm.

The development board used was Altera's Excalibur series. The board contains the EP20K200EFC484-2X FPGA, which has 200.000 gates and 106.496 bits of memory.

For debugging purpose we used the 7 segment leds on the card and various types of switches and buttons.

The card also has a RAM of 64K x 32 bits. This RAM is used to store the image or game of life state. Most modern graphics controllers have simultaneous read/ write RAMs. Unfortunately, it wasn't the case for us, so we had to make sure to never read and write at the same time. The calculation of the life states therefore needs to be interlaced with the screen rendering.

Development process

The project was divided into 4 assignments. Since we didn't need to hand in each assignment, we weren't forced to follow the course's guidelines. Here is how we worked:
At first we wrote the RS232 controller. We thought that it would be a wise idea to implement bi-directional communication, since it didn't involve a lot of extra work. It turns out, this saved us a lot of debugging time later on. We then wrote the VGA related code. The VGA controller is actually much simpler than the word sounds (people are always impressed when you tell them you made a graphics card from scratch). To test the VGA controller we created an image renderer. This involved creating a memory controller to store and read from the RAM. Each pixel of the image is stored as 4 bits, the first bit is ignored, the next bits represent the red, green and blue components. This means we can only display up to 8 colors. This limitation is due to the amount of RAM we have, and also due to the digital-analog converter that we have (a simple diode) that generates either 0V or 0.7V based on the digital input signal (that is either 0V or 5V).

We then implemented the game of life. Our game of life is 640 by 480 cells (each cells corresponds to one pixel on the screen) and the top-bottom, right-left are wrapped. This involved extending the memory controller. The life pattern is saved as 1 bit per cell value, but the pattern is saved twice in memory, because it's much easier to calculate the next generation of cells without erasing the current generation's values (since the game is wrapped) and then swapping memory regions. Our game of life runs at the screen refresh rate (60 Hz).

When transferring data to the board, the first byte specifies if we want to display an image (first byte = 0x02), in which case we must transfer 153601 bytes, or the game of life (first byte = 0x01) in which case we must transfer must send 38401

bytes. This is handled by a multiplex we added to the rs232 code.

One of our design goals (which wasn't stated in the hand out) was to design the system in such a way that we can retransmit a new life pattern or a new image at any time. We have achieved this at the cost of some extra complexity and time investment.

We finally implemented a hardware pseudo-random number generator just for the fun of doing it. We used a technique known as linear feedback shift register (lfsr). We won't go into the mathematical details (which involve galois fields and Fibonacci numbers) of this random generator. We used this random number generator to display the "snow" effect while transferring data to the board.

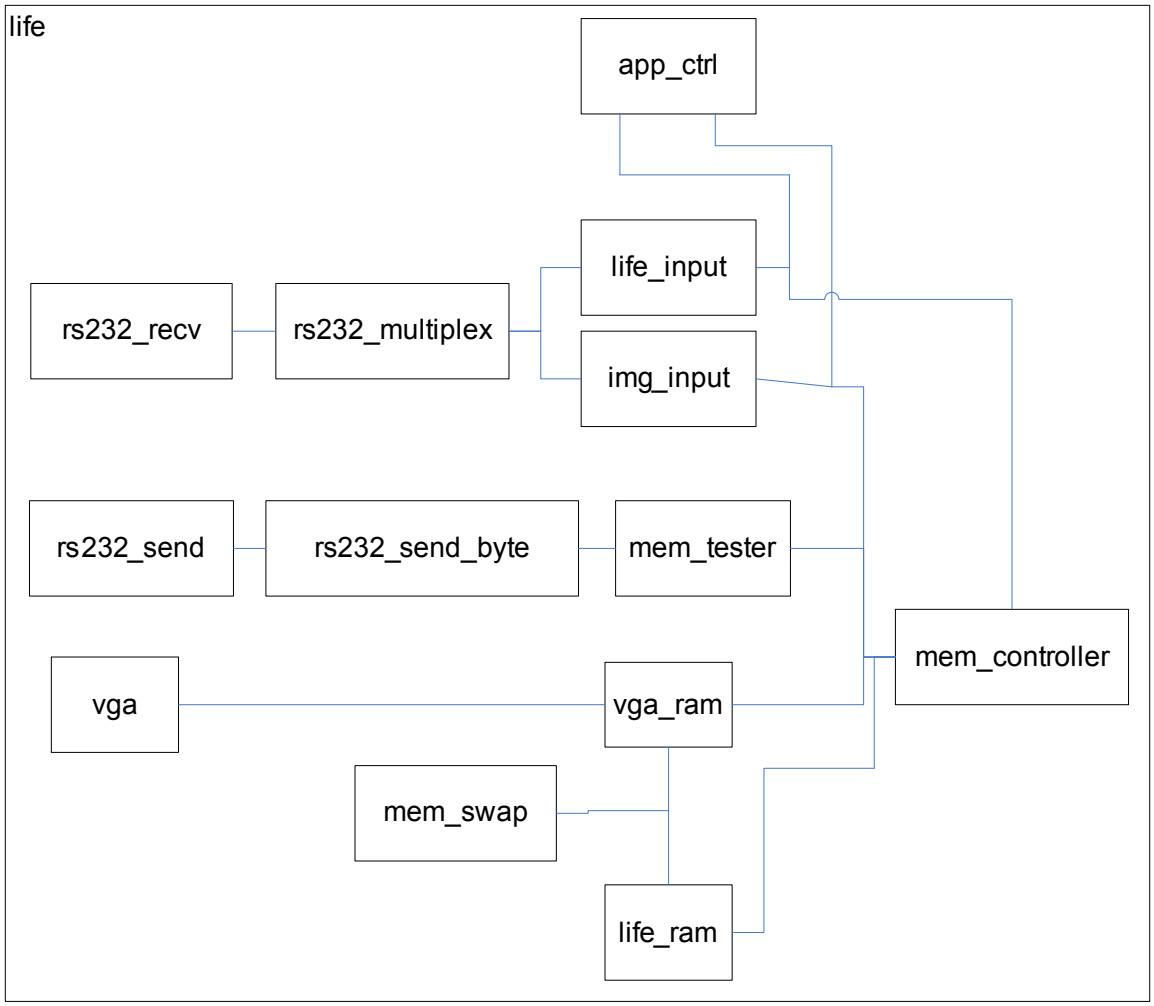
Note: It was said during our lectures that it is impossible to go beyond 640x480 cells or 60Hz. We would like to point out here that this is not impossible. We could have created a larger grid and used buttons to scroll the display. We could have also calculated more generations per screen refresh (which would mean we would be running faster than 60Hz, but not displaying every generation). A third interesting option would have been to link two cards together and have the game run on two screens (with gliders going from one screen to the other). If time had permitted us, we would have probably tried this crazy idea.

Components

Our project is based on several independent components. See figure on following page for an overall view. We will present the components from the simplest to the most complex (hence the more interesting components are presented last). The most important components are the following:

- life.vhd – This component is the top most and ties everything together.
- basic_cell.vhd – Calculates the future state of a cell given the cell and it's 8 neighbors.
- vga.vhd – Generates the VGA signals.
- app_ctrl.vhd – Controls which application is currently running and generates a signal (sync_counter) which is used to synchronize the VGA with the game of life.
- mem_controller.vhd – This part controls the RAM. It allows, in a very flexible manner, for every component to safely perform a read or a write access.

Overall view



Counter

This is a very fundamental component. For simplicity we designed one counter that we reused throughout the project (10 bits counter). We could have designed individual counters (or used the generic type in VHDL) to save bits, but we consider the advantage of ease of development to be worth the extra few bits.
The counter's value is saved in a register and is presented to the outside world (signal value). Every clock cycle, if the inc input signal is set, then the counter value is incremented until it matches data. When the counter reaches data, it raises the timeout signal and reset's itself.

Note: the data (target value) isn't saved in a register.

rs232_recv

rs232_recv_shift_register

Shift registers are another fundamental components in our system. We have different shift registers, since we needed different types each time. The rs232_recv_shift_register takes a single bit as input and generates 8 output bits. The shift happens only when the shift signal is set.

Note: We didn't provide any reset option, since we don't need it (we know that we will always be reading the dataout signal after 8 bits have been pumped in).

The rs232_recv component is used to read data from the serial port. Like many other more complex components, it is based on a state machine. It uses two counters, one to keep track of the 115200 bauds rate and one to remember how many bits of data have been read.
The rs232_recv component raises the rs232_recv_ack signal when it has received 8 bits of data. For an understanding of how the state machine works, refer to UART.pdf.

rs232_multiplex

This component is very basic, it allows for the multiplexing of the game of life with the image renderer. It is connected to rs232_recv. It starts out in the idle state, and looks at the first byte (a byte = 8 bits) that the rs232_recv gets. If the byte is 0x01, then it will set itself into life mode. If the byte is 0x02, then it will set itself into img mode. Otherwise it will stay in idle mode. Remember we stated one of our design goals was to be able to reload images and life patterns. In order for that to happen, the rs232_multiplex must know when the last byte of data has arrived. This happens by receiving a done signal from the life_input or img_input components. Again this component is very generic and could be extended to multiplex with other applications.

img_input

life_input

The objective of these components is to receive the data from rs232_multiplex to which they are connected, and to save the data at the right place in memory. The actual memory access is performed in mem_controller (see below). So all these components do is generate the memory address and raise a signal (img_input_done and life_input_done) when the data has finished arriving. The RAM is accessed by 32 bits at a time (we can read or write 32 bits at a time). So here is how we laid out the memory:

- For the images, the 9 bits of the row are concatenated to the 7 bits of the col. The last 3 bits of the col are used to select one pixel (remember a pixel is stored as 4 bits) among 8 in the 32 bits.
- For the game of life, the top most bit controls which part in ram we want to access (remember we are swapping between two regions). The second bit is always 0. The next 9 bits are the row and the following 5 bits come from the col (higher 5 bits). 5 col bits (lower 5 bits) are used to select a cell among the 32 bits.

app_ctrl

This component tells the other components (through the mode signal) which application is running and generates a signal used to synchronize the VGA with the game of life. The state is defined as two bits, "00" means we are transferring

data on the rs232 (and the VGA must thus display the “snow” effect), “01” means we are in the game of life mode and “11” means we are displaying an image.
The synchronization signal is a counter from 0 to 31 in the case of the game of life and a counter from 0 to 8 in the case of the image display mode.

This component uses the rs232_recv_ack signal to know when the application goes back to waiting for rs232 data to finish transferring.

rs232_send

rs232_send_shift_register

This component performs the reverse operation of the rs232_recv_shift_register. It takes as input 8 bits and shifts 1 bit out whenever the shift signal is set.

The rs232_send component is very similar to the rs232_recv component. It uses counters and rs232_send_shift_register to output data to the serial port.

rs232_send_byte

This component is a helper component that will take a byte and send it using the rs232_send component. It will then notify (by raising byte_send_done) that the data has gone.

mem_tester

This is another helper component, it will use rs232_send_byte to dump part of the memory on the serial port.

led_simple_display

A very simple component to display data on the leds.

led_ascii_display

A similar component, but takes as input the ascii value of a digit and displays the corresponding digit on the 7 segment leds (it maps a digit to the right bit pattern).

vga

vga_controlv

vga_controlh

These components generate the hsync and vsync signals, as described in the specification (refer to VGA.pdf). The vga_controlh also generates signals to tell when to increment the vga_controlv.

This component puts vga_controlh and vga_controlv together. It also generates the col and row values that are used by vga_ram to load and display the right pixels at the right time.

lfsr

This component generates a pseudo-random sequence of bits. It's used by vga_ram to display random pixels on the screen.

vga_ram

vga_shift_register

This component is used to postpone the hsync and vsync signals by 32 or 8 clock cycles. We must do this in order to have enough time to load data from the RAM before sending both, the pixels and sync signals to the monitor.

This is among the most complex components in our system. Basically it loads data from the RAM (either 8 pixels or 32 cells) based on the current row and col values into a register. The register is then used to display these pixels on the screen, but in such a way that we send the pixel either exactly 8 or 32 clock cycles. By doing this we make sure that the pixels correspond to the sync signals, but also we make sure that the register get overridden exactly when the last pixel has been displayed.
This component is responsible for sending black pixels when the VGA is in either horizontal sync or vertical sync.
When in life mode, the vga_ram will load the data from the ram at the cycle 30 and 31. This means all the other cycles can be freely used by life_ram to read and store data.

life_ram

basic_cell

Performs the basic game of life computation.

Note: the future state isn't saved in a register, nor is the current state. The registers are actually in the life_ram component. This component is therefore purely combinatory.

large_cell

This component allows us to compute 32 life cells at the same time. Since the RAM is read/written 32 bits at a time, we found it to be a good design decision to calculate the life cells in chunks of 32 bits.

This is the second complex component in our system. It basically performs 9 load requests (each load takes 1 cycle) and then waits one cycle (so that basic_cell is done it's calculation) and then writes back the data (writing data takes 3 cycles). Since each load and write also takes an extra cycle due to the structure of mem_controller, that means we calculate 32 cells in 23 clock ticks. Since VGA uses clock cycles 30 and 31, we don't do anything with the remaining 7 clock cycles. If you calculate the amount of time needed to calculate an entire generation of life, you will find a value much smaller than the amount of time

needed to update the entire screen; so this implies we have a lot of extra cycles that we can “waste” wherever it is convenient.
The computation starts over again after a memory swap has happened.

ram_swap

This component controls when the ram can be swapped. The rules are the following: the mode must be life, the screen must be in the vertical sync state and the computation of the life is idle.

mem_controller

This is the third complex component in our system. The RAM access is controller here. The mem_controller is connected to the following 5 components: life_input (for writing), img_input (for writing), life_ram (for reading and writing), vga_ram (for reading) and mem_tester (for reading).
The mem_controller checks who wants to access the RAM and grants it based on the following priority rule (from greatest priority to lowest):
img_input > life_input > life_ram_load > mem_tester > life_ram_save > vga_ram

This priority is only for debugging purpose. There is actually never any conflict since the ram and life are synchronized by app_ctrl. However the mem_controller generates an ack signal to the component it has granted access. This means we can change our memory access policy and we could tell vga_ram and life_ram to try accessing the ram until they are granted to do so. This implies the vga_ram would become even more complicated because we would have to manage the case when the component isn’t granted access and must therefore maintain a cache.

life

This is the top most component. All it does is tie all the other components together.

Problems encountered during development

The first problem which we had was related to the rs232 rx signal. We had learned in other courses that signals coming from the outside should go through a series of cascading D flip-flops. This should be done to avoid setting the system in a meta-stable state. At first we thought that we wouldn’t need these flip-flops because we know the rate at which the rx signal varies. This is indeed true, but we don’t know when the first rx change will occur, and this can set our entire rs232 circuit into a meta-stable state. We therefore added two levels of flip-flops. There is no rule as to how many levels to have, probabilistic laws say that the more levels you add, the less likely you are to have a meta-stable state (the risk of having a meta stable state is never 0 !).

The second problem we had was related to the buttons on the board generating multiple transitions (due to the mechanical design of the buttons). We solved this problem by required alternating between two buttons (this is relevant only to our debugging process).

The third problem we had was related to the VHDL language. It turns out the tools we had wouldn't raise a warning if we wrote something like:
if (signal <= some_value) then

This simple mistake took us some time to find, and once we realized our tools were too liberal regarding the language, we wrote a perl script to find such mistakes. It would be great if the VHDL related tools could detect common programming mistakes.

Testing the system

In order to test the system, you must load the life.sof file using Quartus and then launch our Java GUI:
java -Xms256m -Xmx256m -classpath "comm.jar;jiu.jar;." GUI

As you can see we used the comm API (Sun) for transferring data over the serial port and JIU (Java Imaging Utility) to manipulate images (different methods for reducing number of colors, image resizing).

Note: As of the writing of this document, the java code wasn't fully functional.

Possible improvements

There are many things we could have improved, had we had enough time:

- Better documentation of the code we wrote.
- More technical details in this document.
- Written a generic counter to save bits.
- Written a generic shift register to save in number of files.
- Implement the full rs232 protocol (with handshaking, ack and parity checking).

Conclusion

To end this report, all we can say is we had a lot of fun doing hardware development. We were able to meet the requirements very quickly, and we were also able to extend our project without too much trouble. What is very amazing is that we were able to work as a team of two students without any problems. Usually hardware development is more difficult in teams, probably because one small mistake can break everything, but this wasn't at all the case in our team.

References

We used the lecture slides and notes, mainly the following files :

- UART.pdf
- VGA.pdf
- DescriptionRam.pdf
- SimulationAfter.pdf
- ManuelOutilsFPGA.pdf
- CarteAltera.pdf
- GameOfLife.pdf

VHDL Source Code

app_ctrl.vhd

```
-- app_ctrl.vhd

-- Coded by Alok Menghrajani & Peter Amrhyn

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;


entity app_ctrl is
  port (
    clk, reset: in std_logic;
     rs232_recv_ack: in std_logic;         -- 00 this signal will set the system in rs232
recv mode.
     life_input_done, img_input_done: in std_logic;        -- 01 this signal will set the
system in life mode.
                                                            -- 11 this signal will set the
system in img mode.
    mode: out std_logic_vector(1 downto 0);
     sync_counter: out std_logic_vector(9 downto 0));  -- this counter is used to sync the
vga with life.
end app_ctrl;

architecture synth_app_ctrl of app_ctrl is
  component counter is
    port(clk: in std_logic;
         reset: in std_logic;
         inc: in std_logic;
         data: in std_logic_vector(9 downto 0);
         timeout: out std_logic;
         value: out std_logic_vector(9 downto 0));
  end component;
  signal current_state, next_state: std_logic_vector(1 downto 0);
  signal counter_data: std_logic_vector(9 downto 0);
  signal ctrl_reset, counter_timeout: std_logic;
begin
  process(current_state)
  begin
    if (current_state = "11") then
      counter_data <= conv_std_logic_vector (7, 10);
    else
      counter_data <= conv_std_logic_vector(31, 10);
    end if;
  end process;
   counter_unit: counter port map (clk => clk, reset => ctrl_reset, inc => '1', data =>
counter_data,
                                   timeout => counter_timeout, value => sync_counter);

  mode <= current_state;

  process (clk, reset)
  begin
    if (reset = '1') then
      current_state <= "00";
    elsif (clk'event and clk='1') then
      current_state <= next_state;
    end if;

  end process;

  process(current_state, reset, rs232_recv_ack, life_input_done, img_input_done)
  begin
    ctrl_reset <= reset;
    if (rs232_recv_ack = '1') then
      next_state <= "00";
    elsif (life_input_done = '1') then
        ctrl_reset <= '1';
        next_state <= "01";
    elsif (img_input_done = '1') then
      ctrl_reset <= '1';
```

```
      next_state <= "11";
    else
      next_state <= current_state;
    end if;
  end process;

end synth_app_ctrl;
```

basic_cell.vhd

```
-- basic_cell.vhd

-- Coded by Alok Menghrajani & Peter Amrhyn

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity basic_cell is
  port(nw, n, ne, e, se, s, sw, w, m: in std_logic;
       next_state: out std_logic);
end basic_cell;

architecture synth_basic_cell of basic_cell is
  signal temp1 : std_logic_vector(1 downto 0);
  signal temp2 : std_logic_vector(1 downto 0);
  signal temp3 : std_logic_vector(1 downto 0);
  signal temp4 : std_logic_vector(2 downto 0);
  signal sum : std_logic_vector(3 downto 0);
begin
  temp1 <= ('0' & nw) + ('0' & n) + ('0' & ne);
  temp2 <= ('0' & e) + ('0' & se) + ('0' & s);
  temp3 <= ('0' & sw) + ('0' & w);
  temp4 <= ('0' & temp1) + ('0' & temp2);
  sum <= ("00" & temp3) + ('0' & temp4);

  process(m, sum)
  begin
    next_state <= '0';
    if (sum="0011") then
       next_state <= '1';                  -- 3 neighbours means birth or survival (m=0 or
m=1)
    elsif(m='1' and sum="0010") then
      next_state <= '1';                -- 2 neighbours means birth (iff m=1)
    end if;
  end process;
end synth_basic_cell;
```

counter.vhd

```
-- counter.vhd

-- Coded by Alok Menghrajani & Peter Amrhyn
--
-- This unit is a "generic" (it's not that generic,
-- because it always uses 10 bits) counter. It
-- counts from 0 to data. When it reaches data, it
-- raises the timeout signal and resets itself on
-- the next clk rise.

-- The counter only counts when the inc input signal
-- is set.

-- Note: the data is not kept in a register (there is
--       no load signal), so it must always be set by
--       the unit using the counter.

-- Value is the counter's current value.

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity counter is
```

```
  port(clk: in std_logic;
       reset: in std_logic;
       inc: in std_logic;
       data: in std_logic_vector(9 downto 0);
       timeout: out std_logic;
       value: out std_logic_vector(9 downto 0));
end counter;

architecture synth_counter of counter is
  signal reg: std_logic_vector(9 downto 0);
begin
  process(reset, clk)
  begin
    if (reset='1') then
      reg<=(others => '0');
      timeout <= '0';
    elsif (clk'event and clk='1') then
      if (inc='1') then
        if (reg = data) then
          timeout <= '1';
        else
          timeout <= '0';

        end if;
        if (reg = data) then
          reg<=(others => '0');
        else
          reg <= reg + conv_std_logic_vector(1, 9);
        end if;
      else
        timeout <= '0';
      end if;
    end if;
  end process;

  value <= reg;
end synth_counter;
```

img_input.vhd

```
-- life_input.vhd

-- Coded by Alok Menghrajani & Peter Amrhyn

-- Three counters are used, to keep track of the col/row and
-- when to write.

-- row (0-479) => 9 bits                       ^^^^^^^^^
-- col (0-1024) => 7 most significant bits             ^^^^^

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity img_input is
  port(clk: in std_logic;
       reset: in std_logic;
       -- rs232
       img_recv_ack: in std_logic;
       img_recv_data: in std_logic_vector(7 downto 0);
       -- mem
       img_input_write_req: out std_logic;
       img_input_write_ack: in std_logic;
       img_input_addr: out std_logic_vector(15 downto 0);
       img_input_data: out std_logic_vector(31 downto 0);
       img_input_done: out std_logic);
end img_input;

architecture synth_img_input of img_input is
  component counter is
    port (clk: in std_logic;
          reset: in std_logic;
          inc: in std_logic;
```

```
          data: in std_logic_vector(9 downto 0);
          timeout: out std_logic;
          value: out std_logic_vector(9 downto 0));
  end component;

  signal byte_timeout, col_timeout, row_timeout: std_logic;
    signal byte_value, col_value, row_value, byte_data, col_data, row_data :
std_logic_vector(9 downto 0);

  component rs232_input_shift_register is
    port (clk: in std_logic;
          datain: in std_logic_vector(7 downto 0);
          shift: in std_logic;
          dataout: out std_logic_vector(31 downto 0));
  end component;

begin
  byte_data <= conv_std_logic_vector(3, 10);
  col_data <= conv_std_logic_vector(79, 10);
  row_data <= conv_std_logic_vector(479, 10);

byte_counter: counter port map (clk => clk, reset => reset, inc => img_recv_ack, data =>
byte_data,
                                timeout => byte_timeout, value => byte_value);
col_counter: counter port map (clk => clk, reset => reset, inc => byte_timeout, data =>
col_data,
                               timeout => col_timeout, value => col_value);
row_counter: counter port map (clk => clk, reset => reset, inc => col_timeout, data =>
row_data,
                               timeout => row_timeout, value => row_value);
shift_reg: rs232_input_shift_register port map (clk => clk, datain => img_recv_data,
shift => img_recv_ack, dataout => img_input_data);

  img_input_addr <= row_value(8 downto 0) & col_value(6 downto 0);
  img_input_done <= row_timeout;
  img_input_write_req <= byte_timeout;

end synth_img_input;
```

large_cell.vhd

```
-- large_cell.vhd

-- Coded by Alok Menghrajani & Peter Amrhyn

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity large_cell is
  port(n, m, s: in std_logic_vector(33 downto 0);
       newgen: out std_logic_vector(31 downto 0));
end large_cell;

architecture synth_large_cell of large_cell is
  component basic_cell is
    port(nw, n, ne, e, se, s, sw, w, m : in std_logic;
         next_state : out std_logic);
  end component;
begin
Life32: for i in 1 to 32 generate
        life32_map: basic_cell port map(m => m(i), nw => n(i-1),
                                        n => n(i), ne => n(i+1), e => m(i+1), se => s(i
+1), s => s(i),
                                               sw => s(i-1), w => m(i-1), next_state =>
newgen(i-1));
        end generate Life32;
end synth_large_cell;
```

led_ascii_display.vhd

```
-- led_ascii_display.vhd

-- Coded by Alok Menghrajani & Peter Amrhyn
```

```
--
-- This unit is for debugging purpose. It
-- allows us to display ascii decoded digits.

-- Valid values are currently 0 (ascii 48=0x30) to
-- 9 (ascii 57=0x39).

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

--   s0
--  s  s
--  3  5
--   s1
--  s  s
--  4  6
--   s2   .s7

entity led_ascii_display is
  port(data: in std_logic_vector(6 downto 0);
       leds: out std_logic_vector(6 downto 0));
end led_ascii_display;

architecture synth_led_ascii_display of led_ascii_display is
begin
  process(data)
  begin
    case data is
      when "0000000" => -- clr
      leds(0) <= '1'; leds(1) <= '1'; leds(2) <= '1'; leds(3) <= '1';
      leds(4) <= '1'; leds(5) <= '1'; leds(6) <= '1';
      when "0110000" => -- 0
                    leds(0) <= '0';
        leds(3) <= '0';             leds(5) <= '0';
                    leds(1) <= '1';
        leds(4) <= '0';             leds(6) <= '0';
                    leds(2) <= '0';
      when "0110001" => -- 1
                    leds(0) <= '1';
        leds(3) <= '1';             leds(5) <= '0';
                    leds(1) <= '1';
        leds(4) <= '1';             leds(6) <= '0';
                    leds(2) <= '1';
      when "0110010" => -- 2
                    leds(0) <= '0';
        leds(3) <= '1';             leds(5) <= '0';
                    leds(1) <= '0';
        leds(4) <= '0';             leds(6) <= '1';
                    leds(2) <= '0';
      when "0110011" => -- 3
                    leds(0) <= '0';
        leds(3) <= '1';             leds(5) <= '0';
                    leds(1) <= '0';
        leds(4) <= '1';             leds(6) <= '0';
                    leds(2) <= '0';
      when "0110100" => -- 4
                    leds(0) <= '1';
        leds(3) <= '0';             leds(5) <= '0';
                    leds(1) <= '0';
        leds(4) <= '1';             leds(6) <= '0';
                    leds(2) <= '1';
      when "0110101" => -- 5
                    leds(0) <= '0';
        leds(3) <= '0';             leds(5) <= '1';
                    leds(1) <= '0';
        leds(4) <= '1';             leds(6) <= '0';
                    leds(2) <= '0';
      when "0110110" => -- 6
                    leds(0) <= '0';
        leds(3) <= '0';             leds(5) <= '1';
                    leds(1) <= '0';
        leds(4) <= '0';             leds(6) <= '0';
                    leds(2) <= '0';
      when "0110111" => -- 7
```

```
                    leds(0) <= '0';
        leds(3) <= '1';             leds(5) <= '0';
                    leds(1) <= '1';
        leds(4) <= '1';             leds(6) <= '0';
                    leds(2) <= '1';
      when "0111000" => -- 8
                    leds(0) <= '0';
        leds(3) <= '0';             leds(5) <= '0';
                    leds(1) <= '0';
        leds(4) <= '0';             leds(6) <= '0';
                    leds(2) <= '0';
      when "0111001" => -- 9
                    leds(0) <= '0';
        leds(3) <= '0';             leds(5) <= '0';
                    leds(1) <= '0';
        leds(4) <= '1';             leds(6) <= '0';
                    leds(2) <= '0';
      when others => -- err
                    leds(0) <= '0';
        leds(3) <= '0';             leds(5) <= '1';
                    leds(1) <= '0';
        leds(4) <= '0';             leds(6) <= '1';
                    leds(2) <= '0';

    end case;
  end process;
end synth_led_ascii_display;
```

led_simple_display.vhd

```
-- led_simple_display.vhd

-- Coded by Alok Menghrajani & Peter Amrhyn
--
-- This unit is for debugging purpose. It
-- allows us to display 7 bits on the
-- 7 segment leds

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

--   s0
--  s  s
--  3  5
--   s1
--  s  s
--  4  6
--   s2   .s7

entity led_simple_display is
  port(data: in std_logic_vector(6 downto 0);
       leds: out std_logic_vector(6 downto 0));
end led_simple_display;

architecture synth_led_simple_display of led_simple_display is
begin
  leds <= data;
end synth_led_simple_display;
```

lfsr.vhd

```
-- lfsr.vhd

-- Coded by Alok Menghrajani & Peter Amrhyn

-- Hardware pseudo-random number generator
-- based on the linear feedback shift register technique.

-- This implementation is based on 8 bits and provides
-- a random bit pattern of length 255. The tab used is
-- 0,1,2,7

library ieee;
```

```
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity lfsr is
  port (
    clk, reset: in std_logic;
    rand: out std_logic);
end lfsr;

architecture synth_lfsr of lfsr is
  signal reg: std_logic_vector(31 downto 0);
  signal f: std_logic;
begin
  process(clk, reset)
  begin
    if (reset='1') then
      reg <= (0 => '1', others => '0');
    elsif (clk'event and clk='1') then
      reg <= f & reg(31 downto 1);
    end if;
  end process;
  rand <= reg(0);
  f <= reg(0) xor reg(3);
end synth_lfsr;
```

life.vhd

```
-- life.vhd

-- Coded by Alok Menghrajani & Peter Amrhyn
--
-- This is the top most unit of our system.
-- Right now it runs a game of life with the
-- input (game setup) uploaded through the serial port
-- and the display on a 640x480 vga screen.
--
-- We might support image uploading too.

-- Refer to signals.txt for the mapping of signals
-- from the altera card to the vhdl code.

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity life is
  port(clk: in std_logic;
       nreset: in std_logic;
       rx: in std_logic;
       tx: out std_logic;
       buttons: in std_logic_vector(3 downto 0);
       switches: in std_logic_vector(7 downto 0);
       led1: out std_logic_vector(7 downto 0);
       led2: out std_logic_vector(7 downto 0);
       ram_data: inout std_logic_vector(31 downto 0);
       ram_addr: out std_logic_vector(15 downto 0);
       ram_we: out std_logic;
       ram_oe: out std_logic;
       ram0_cs: out std_logic;
       ram1_cs: out std_logic;
       ram0_ble: out std_logic;
       ram1_ble: out std_logic;
       ram0_bhe: out std_logic;
       ram1_bhe: out std_logic;
       r, g, b: out std_logic;
       hsync_out, vsync_out: out std_logic);
end life;

architecture synth_life of life is
  component led_ascii_display is
    port(data: in std_logic_vector(6 downto 0);
         leds: out std_logic_vector(6 downto 0));
  end component;
```

```
component led_simple_display is
  port(data: in std_logic_vector(6 downto 0);
       leds: out std_logic_vector(6 downto 0));
end component;
component rs232_recv is
  port(clk: in std_logic;
       reset: in std_logic;
       rx: in std_logic;
       rs232_recv_ack: out std_logic;
       rs232_recv_data: out std_logic_vector(7 downto 0));
end component;
component rs232_send is
  port(clk: in std_logic;
       reset: in std_logic;
       tx: out std_logic;
       rs232_send_done: out std_logic;
       rs232_send_data: in std_logic_vector(7 downto 0);
       rs232_sending: in std_logic);
end component;
component img_input is
  port(clk: in std_logic;
       reset: in std_logic;
       img_recv_ack: in std_logic;
       img_recv_data: in std_logic_vector(7 downto 0);
       img_input_write_req: out std_logic;
       img_input_write_ack: in std_logic;
       img_input_addr: out std_logic_vector(15 downto 0);
       img_input_data: out std_logic_vector(31 downto 0);
       img_input_done: out std_logic);
end component;
component life_input is
  port(clk: in std_logic;
       reset: in std_logic;
       life_recv_ack: in std_logic;
       life_recv_data: in std_logic_vector(7 downto 0);
       life_input_write_req: out std_logic;
       life_input_write_ack: in std_logic;
       life_input_addr: out std_logic_vector(15 downto 0);
       life_input_data: out std_logic_vector(31 downto 0);
       life_input_done: out std_logic);
end component;
component mem_controller is
port(clk: in std_logic;
     reset: in std_logic;
     ram_data: inout std_logic_vector(31 downto 0);
     ram_addr: out std_logic_vector(15 downto 0);
     ram_cs: out std_logic;
     ram_we: out std_logic;
     ram_swap: in std_logic;
     -- life_input (WRITE)
     life_input_write_req: in std_logic;
     life_input_write_ack: out std_logic;
     life_input_addr: in std_logic_vector(15 downto 0);
     life_input_data: in std_logic_vector(31 downto 0);
     -- img_input (WRITE)
     img_input_write_req: in std_logic;
     img_input_write_ack: out std_logic;
     img_input_addr: in std_logic_vector(15 downto 0);
     img_input_data: in std_logic_vector(31 downto 0);
    -- mem_tester (READ)
     mem_tester_read_req: in std_logic;
     mem_tester_read_ack: out std_logic;
     mem_tester_addr: in std_logic_vector(15 downto 0);
     mem_tester_data: out std_logic_vector(31 downto 0);
     -- life_ram_load (READ)
     life_ram_load_read_req: in std_logic;
     life_ram_load_read_ack: out std_logic;
     life_ram_load_addr: in std_logic_vector(15 downto 0);
     life_ram_load_data: out std_logic_vector(31 downto 0);
     -- life_ram_save (WRITE)
     life_ram_save_write_req: in std_logic;
     life_ram_save_write_ack: out std_logic;
     life_ram_save_addr: in std_logic_vector(15 downto 0);
     life_ram_save_data: in std_logic_vector(31 downto 0);
     -- vga_ram (READ)
     vga_ram_read_without_mem_swap: in std_logic;
```

```
        vga_ram_read_req: in std_logic;
        vga_ram_read_ack: out std_logic;
        vga_ram_addr: in std_logic_vector(15 downto 0);
        vga_ram_data: out std_logic_vector(31 downto 0);
        but2, but3: in std_logic);
  end component;
  component mem_tester is
    port(clk: in std_logic;
         reset: in std_logic;
         debug_start: in std_logic;       -- when this signal is set to 1, the mem_tester
will start dumping the memory.
         mem_tester_read_req: out std_logic;
         mem_tester_read_ack: in std_logic;
         mem_tester_addr: out std_logic_vector(15 downto 0);
         mem_tester_data: in std_logic_vector(31 downto 0);
         byte_send_done: in std_logic;
         byte_send_data: out std_logic_vector(7 downto 0);
         byte_sending: out std_logic);
  end component;
  component rs232_send_byte is
    port(clk: in std_logic;
         reset: in std_logic;
         byte_send_data: in std_logic_vector(7 downto 0);
         byte_sending: in std_logic;
         byte_send_done: out std_logic;
         rs232_send_done: in std_logic;
         rs232_send_data: out std_logic_vector(7 downto 0);
         rs232_sending: out std_logic);
  end component;
  component life_ram is
    port(clk: in std_logic;
         reset: in std_logic;
             start: in std_logic;      -- enable/disable the life calculation -- restart
calculation (send from vga_ram
                                       --after the swap has happened)
         mode: in std_logic_vector(1 downto 0);
         sync_counter: in std_logic_vector(9 downto 0);
         life_ram_done: out std_logic; -- tell the system that the memory can be swapped.
         life_ram_load_read_req: out std_logic;
         life_ram_load_read_ack: in std_logic;
         life_ram_load_data: in std_logic_vector(31 downto 0);
         life_ram_load_addr: out std_logic_vector(15 downto 0);
         life_ram_save_write_req: out std_logic;
         life_ram_save_write_ack: in std_logic;
         life_ram_save_data: out std_logic_vector(31 downto 0);
         life_ram_save_addr: out std_logic_vector(15 downto 0));
  end component;
  component vga is
    port (
      clk, reset: in std_logic;
      hsync, vsync: out std_logic;

      is_hsync, is_vsync, is_sync: out std_logic;
      value1, value2: out std_logic_vector (9 downto 0));
  end component;
  component vga_ram is
    port (
      clk, reset: in std_logic;
      hsync_in, vsync_in: in std_logic;
      issync_in: in std_logic;
      col, row: in std_logic_vector (9 downto 0);
      hsync_out, vsync_out: out std_logic;
      r, g, b: out std_logic;
      vga_ram_read_req: out std_logic;
      vga_ram_read_ack: in std_logic;
      vga_ram_addr: out std_logic_vector(15 downto 0);
      vga_ram_data: in std_logic_vector(31 downto 0);
      vga_ram_read_without_mem_swap: out std_logic;
      mode: in std_logic_vector(1 downto 0);
      sync_counter: in std_logic_vector(9 downto 0));
  end component;
  component ram_swap is
    port (
      clk, reset: in std_logic;
      is_vsync: in std_logic;
      life_ram_done: in std_logic;
```

```
      mode: in std_logic_vector(1 downto 0);
      but4: in std_logic;
      ram_swap: out std_logic);
  end component;
  component app_ctrl is
    port (
      clk, reset: in std_logic;
       rs232_recv_ack: in std_logic;        -- 00 this signal will set the system in rs232
recv mode.
       life_input_done, img_input_done: in std_logic;       -- 01 this signal will set the
system in life mode.
      mode: out std_logic_vector(1 downto 0);
       sync_counter: out std_logic_vector(9 downto 0));  -- this counter is used to sync
the vga with life.
  end component;
  component rs232_multiplex
    port (
      clk, reset: in std_logic;
      rs232_recv_ack: in std_logic;
      rs232_recv_data: in std_logic_vector(7 downto 0);
      life_recv_ack: out std_logic;
      life_recv_data: out std_logic_vector(7 downto 0);
      life_done: in std_logic;
      img_recv_ack: out std_logic;
      img_recv_data: out std_logic_vector(7 downto 0);
      img_done: in std_logic);
  end component;

  signal reset: std_logic;
  signal life_input_done, img_input_done: std_logic;
  signal life_ram_done: std_logic;

  signal but1, but2, but3, but4: std_logic;

  -- rs232_multiplex
  signal life_recv_ack, img_recv_ack: std_logic;
  signal life_recv_data, img_recv_data: std_logic_vector(7 downto 0);

  -- vga
  signal hsync, vsync, is_hsync, is_vsync, is_sync, swap: std_logic;
  signal value1, value2: std_logic_vector(9 downto 0);

  -- rs232
  signal rx1, rx2 : std_logic;
  signal rs232_recv_data, rs232_send_data, byte_send_data: std_logic_vector(7 downto 0);
  signal rs232_recv_ack: std_logic;
  signal byte_send_done, byte_sending, rs232_send_done, rs232_sending: std_logic;

  -- mem_controller
  signal ram_cs, ram_writeenable: std_logic;
    signal mem_tester_read_req, mem_tester_read_ack, life_input_write_req,
life_input_write_ack: std_logic;
  signal img_input_write_req, img_input_write_ack: std_logic;
    signal life_ram_load_read_req, life_ram_load_read_ack, life_ram_save_write_req,
life_ram_save_write_ack: std_logic;
  signal vga_ram_read_req, vga_ram_read_ack: std_logic;
    signal mem_tester_addr, life_input_addr, life_ram_load_addr, life_ram_save_addr:
std_logic_vector(15 downto 0);
  signal img_input_addr: std_logic_vector(15 downto 0);
    signal mem_tester_data, life_input_data, life_ram_load_data, life_ram_save_data:
std_logic_vector(31 downto 0);

  signal img_input_data: std_logic_vector(31 downto 0);
  signal vga_ram_addr: std_logic_vector(15 downto 0);
  signal vga_ram_data: std_logic_vector(31 downto 0);
  signal vga_ram_read_without_mem_swap: std_logic;

  signal sync_counter: std_logic_vector(9 downto 0);
  signal mode: std_logic_vector(1 downto 0);
begin
  -- Solve the problem with the button being active at 1
  reset <= not(nreset);

  but1 <= not(buttons(0));
  but2 <= not(buttons(1));
  but3 <= not(buttons(2));
```

```
  but4 <= not(buttons(3));

  -- Solve the problem of rx being an external signal
  process(clk, reset)
  begin
    if (reset = '1') then
      rx1 <= '1';
      rx2 <= '1';
    elsif (clk'event and clk='1') then
      rx1 <= rx;
      rx2 <= rx1;
    end if;
  end process;

  -- rs232_multiplex
  rs232_mutliplex_unit: rs232_multiplex port map (clk => clk, reset => reset,
                                                  rs232_recv_ack => rs232_recv_ack,
rs232_recv_data => rs232_recv_data,
                                                    life_recv_ack => life_recv_ack,
life_recv_data => life_recv_data, life_done => life_input_done,
                                                     img_recv_ack => img_recv_ack,
img_recv_data => img_recv_data, img_done => img_input_done);

  -- app_ctrl
    app_ctrl_unit: app_ctrl port map (clk => clk, reset => reset, rs232_recv_ack =>
rs232_recv_ack,
                                    life_input_done => life_input_done, img_input_done =>
img_input_done,
                                    mode => mode, sync_counter => sync_counter);

  -- ram_swap
   ram_swap_unit: ram_swap port map (clk => clk, reset => reset, is_vsync => is_vsync,
life_ram_done => life_ram_done,
                                    mode => mode, ram_swap => swap, but4 => but4);

  -- rs232_recv
  rs232_recv_unit: rs232_recv port map (clk => clk, reset => reset,
                                        rx => rx2, rs232_recv_ack => rs232_recv_ack,
                                        rs232_recv_data => rs232_recv_data);
  -- rs232_send
  rs232_send_unit: rs232_send port map (clk => clk, reset => reset,
                                        tx => tx, rs232_send_done => rs232_send_done,
                                         rs232_send_data => rs232_send_data, rs232_sending
=> rs232_sending);
  -- img_input
    img_input_unit : img_input port map (clk => clk, reset => reset, img_recv_ack =>
img_recv_ack, img_recv_data => img_recv_data,
                                            img_input_write_req => img_input_write_req,
img_input_write_ack => img_input_write_ack,
                                          img_input_addr => img_input_addr, img_input_data
=> img_input_data,
                                        img_input_done => img_input_done);
  -- life_input
   life_input_unit : life_input port map (clk => clk, reset => reset, life_recv_ack =>
life_recv_ack, life_recv_data => life_recv_data,
                                         life_input_write_req => life_input_write_req,
life_input_write_ack => life_input_write_ack,
                                                    life_input_addr => life_input_addr,
life_input_data => life_input_data,
                                        life_input_done => life_input_done);
  -- mem_controller
   mem_controller_unit: mem_controller port map (clk => clk, reset => reset, ram_data =>
ram_data, ram_addr => ram_addr,
                                                        ram_cs => ram_cs, ram_we =>
ram_writeenable, ram_swap => swap,
                                                            img_input_write_req =>
img_input_write_req, img_input_write_ack => img_input_write_ack,
                                                    img_input_addr => img_input_addr,
img_input_data => img_input_data,
                                                        life_input_write_req =>
life_input_write_req, life_input_write_ack => life_input_write_ack,
                                                    life_input_addr => life_input_addr,
life_input_data => life_input_data,
                                                            mem_tester_read_req =>
mem_tester_read_req, mem_tester_read_ack => mem_tester_read_ack,
```

```
                                                        mem_tester_addr => mem_tester_addr,
mem_tester_data => mem_tester_data,
                                                                life_ram_load_read_req =>
life_ram_load_read_req, life_ram_load_read_ack => life_ram_load_read_ack,
                                                    life_ram_load_addr => life_ram_load_addr,
life_ram_load_data => life_ram_load_data,
                                                                life_ram_save_write_req =>
life_ram_save_write_req, life_ram_save_write_ack => life_ram_save_write_ack,
                                                    life_ram_save_addr => life_ram_save_addr,
life_ram_save_data => life_ram_save_data,
                                                        vga_ram_read_req => vga_ram_read_req,
vga_ram_read_ack => vga_ram_read_ack,
                                                                vga_ram_addr => vga_ram_addr,
vga_ram_data => vga_ram_data,
                                                            vga_ram_read_without_mem_swap =>
vga_ram_read_without_mem_swap,
                                                    but2 => but2, but3 => but3);

  ram0_cs <= not(ram_cs);               -- active at 0 !
  ram1_cs <= not(ram_cs);
  ram_we <= not(ram_writeenable);
   ram_oe <= ram_writeenable when ram_cs='1' else '1';  -- oe is selected when cs='1' and
we are not writing.
  ram0_ble <= not(ram_cs);
  ram0_bhe <= not(ram_cs);
  ram1_ble <= not(ram_cs);
  ram1_bhe <= not(ram_cs);

  -- game of life
     life_ram_unit:  life_ram  port  map  (clk  =>  clk,  reset  =>  reset,  mode  =>  mode,
sync_counter => sync_counter,
                                    start => swap,
                                                            life_ram_done => life_ram_done,
life_ram_load_read_req => life_ram_load_read_req,
                                              life_ram_load_read_ack => life_ram_load_read_ack,
life_ram_load_data => life_ram_load_data,
                                                    life_ram_load_addr => life_ram_load_addr,
life_ram_save_write_req => life_ram_save_write_req,
                                           life_ram_save_write_ack => life_ram_save_write_ack,
life_ram_save_data => life_ram_save_data,
                                    life_ram_save_addr => life_ram_save_addr);

  -- vga
  vga_unit: vga port map (clk => clk, reset => reset, hsync => hsync, vsync => vsync,
                          is_hsync => is_hsync, is_vsync => is_vsync, is_sync => is_sync,
                          value1 => value1, value2 => value2);

  vga_ram_unit: vga_ram port map (clk => clk, reset => reset, hsync_in => hsync, vsync_in
=> vsync, issync_in => is_sync,
                                   col => value1, row => value2, hsync_out => hsync_out,
vsync_out => vsync_out, r => r,
                                   g => g, b => b, vga_ram_read_req => vga_ram_read_req,
vga_ram_read_ack => vga_ram_read_ack,
                                             vga_ram_addr => vga_ram_addr, vga_ram_data =>
vga_ram_data, vga_ram_read_without_mem_swap =>
                                              vga_ram_read_without_mem_swap, mode => mode,
sync_counter => sync_counter);

  -- mem_tester
  mem_tester_unit: mem_tester port map (clk => clk, reset => reset, debug_start => but1,
                                          mem_tester_read_req => mem_tester_read_req,
mem_tester_read_ack => mem_tester_read_ack,
                                                        mem_tester_addr => mem_tester_addr,
mem_tester_data => mem_tester_data,
                                       byte_sending => byte_sending, byte_send_data =>
byte_send_data, byte_send_done => byte_send_done);

     rs232_send_byte_unit:  rs232_send_byte  port  map  (clk  =>  clk,  reset  =>  reset,
byte_send_data => byte_send_data, byte_sending => byte_sending,
                                                          byte_send_done => byte_send_done,
rs232_send_done => rs232_send_done,
                                                        rs232_send_data => rs232_send_data,
rs232_sending => rs232_sending);

  -- led one
  led_one: led_ascii_display port map(data => "0001001", leds => led1(6 downto 0));
```

```
  led1(7) <= but1;

  -- led two
  led_two: led_ascii_display port map(data => "0000110", leds => led2(6 downto 0));
  led2(7) <= but2;

end synth_life;
```

life_input.vhd

```
-- life_input.vhd

-- Coded by Alok Menghrajani & Peter Amrhyn
--
-- This unit is used to write 32 bits representing
-- 32 pixels from the rs232 into the RAM. life_done
-- is raised once 640*480 (after 9600 writes) pixels
-- have been written.
--
-- Three counters are used, to keep track of the col/row and
-- when to write.
--
-- The memory is split in two parts, allowing us to read from one
-- part while (we don't read & write at the same time, we just
-- write in one part in order not to overwrite the data needed for the
-- current state) writing in the other, and vice-versa.
--
--                               ADDR =      0srrrrrrrrrccccc
-- which memory to use => 1 bit               ^
-- row (0-479) => 9 bits                       ^^^^^^^^^
-- col (0-1024) => 5 most significant bits             ^^^^^
-- Note: the MSB is always 0, so half the memory address space isn't used.
--
-- This unit depends on
--   counter
--   rs232_input_shift_register
--
-- Note: life_input_done is not raised at the same time as the
--       last life_ram_write (it takes 4 (???) clk ticks to propagate
--       through the counters)

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity life_input is
  port(clk: in std_logic;
       reset: in std_logic;
       -- rs232
       life_recv_ack: in std_logic;
       life_recv_data: in std_logic_vector(7 downto 0);
       -- mem
       life_input_write_req: out std_logic;
       life_input_write_ack: in std_logic;
       life_input_addr: out std_logic_vector(15 downto 0);
       life_input_data: out std_logic_vector(31 downto 0);
       life_input_done: out std_logic);
end life_input;

architecture synth_life_input of life_input is
  component counter is
    port (clk: in std_logic;
          reset: in std_logic;
          inc: in std_logic;
          data: in std_logic_vector(9 downto 0);
          timeout: out std_logic;
          value: out std_logic_vector(9 downto 0));
  end component;
  signal byte_timeout, col_timeout, row_timeout: std_logic;
    signal byte_value, col_value, row_value, byte_data, col_data, row_data :
std_logic_vector(9 downto 0);

  component rs232_input_shift_register is
    port (clk: in std_logic;
```

```
          datain: in std_logic_vector(7 downto 0);
          shift: in std_logic;
          dataout: out std_logic_vector(31 downto 0));
  end component;

begin
  byte_data <= conv_std_logic_vector(3, 10);
  col_data <= conv_std_logic_vector(19, 10);
  row_data <= conv_std_logic_vector(479, 10);

byte_counter: counter port map (clk => clk, reset => reset, inc => life_recv_ack, data =>
byte_data,
                                timeout => byte_timeout, value => byte_value);
col_counter: counter port map (clk => clk, reset => reset, inc => byte_timeout, data =>
col_data,
                               timeout => col_timeout, value => col_value);
row_counter: counter port map (clk => clk, reset => reset, inc => col_timeout, data =>
row_data,
                               timeout => row_timeout, value => row_value);
shift_reg: rs232_input_shift_register port map (clk => clk, datain => life_recv_data,
shift => life_recv_ack, dataout => life_input_data);

  life_input_addr <= "00" & row_value(8 downto 0) & col_value(4 downto 0);
  life_input_done <= row_timeout;
  life_input_write_req <= byte_timeout;

end synth_life_input;
```

life_ram.vhd

```
-- life_ram.vhd

-- Coded by Alok Menghrajani & Peter Amrhyn
--
-- This unit is used to load 9 times 32 "pixels" into the large_cell unit
-- calculate the next generation for the 32 pixels, and write
-- the result back into the RAM.
--
-- In order to keep this unit in sync with the vga code, we must
-- respect a 32 clk cycle.

-- Rules to swap memory:
-- life_ram must be done (must be in waiting state)
-- vga must reach bottom of screen
-- If the swap happens then a flag must be set (because the vsync signal
-- lasts for a long time) and the life_ram must be aware of it (restart must
-- be set).

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity life_ram is
  port(clk: in std_logic;
       reset: in std_logic;
       mode: in std_logic_vector(1 downto 0);
       sync_counter: in std_logic_vector(9 downto 0);

       start: in std_logic;
    -- enable/disable the life calculation -- restart calculation (send from vga_ram
    --after the swap has happened)
       -- we need to start signals, one that comes from life (that is used to
       -- start when the rs232 finishes but before the first swap, the second
       -- to restart once the swap has happened. this is due to the fact that we
       -- wanted to be able to debug easily. An alternative would have been
       -- to always first swap then calculate, and the first screen would be
       -- wrong but nobody would notice.
       life_ram_done: out std_logic; -- tell the system that the memory can be swapped.
       life_ram_load_read_req: out std_logic;
       life_ram_load_read_ack: in std_logic;
       life_ram_load_data: in std_logic_vector(31 downto 0);
       life_ram_load_addr: out std_logic_vector(15 downto 0);
       life_ram_save_write_req: out std_logic;
       life_ram_save_write_ack: in std_logic;
```

```
       life_ram_save_data: out std_logic_vector(31 downto 0);
       life_ram_save_addr: out std_logic_vector(15 downto 0));
end life_ram;

architecture synth_life_ram of life_ram is
  component counter is
    port(clk: in std_logic;
         reset: in std_logic;
         inc: in std_logic;
         data: in std_logic_vector(9 downto 0);
         timeout: out std_logic;
         value: out std_logic_vector(9 downto 0));
  end component;
  component large_cell is
    port(n, m, s: in std_logic_vector(33 downto 0);
         newgen: out std_logic_vector(31 downto 0));
  end component;
    -- we need to convert 9x32 bits into the 3x34 bits that interest us (input for
large_cell)
  signal tnw, tn, tne, tw, tm, te, tsw, ts, tse: std_logic_vector(31 downto 0);
  signal n, m, s: std_logic_vector(33 downto 0);
  signal newgen: std_logic_vector(31 downto 0);
  -- counters
  signal col_value, row_value, col_data, row_data: std_logic_vector(9 downto 0);
  signal col_timeout, row_timeout: std_logic;
  -- we need to calculate or keep track of the previous and next values of the counters
  signal rowm1, rowp1, colm1, colp1: std_logic_vector(9 downto 0);
  type state is (stopped, paused, waiting, l1, l2, l3, l4, l5, l6, l7, l8, l9, w, r);
  signal current_state, next_state : state;
  -- CONTROL SIGNALS
  signal ctrl_col_inc: std_logic;
  signal save_data: state;
  signal ctrl_reset: std_logic;
begin
  col_data <= conv_std_logic_vector(19, 10);
     col_counter_unit: counter port map (clk => clk, reset => ctrl_reset, inc =>
ctrl_col_inc, data => col_data, timeout => col_timeout, value => col_value);
  row_data <= conv_std_logic_vector(479, 10);
     row_counter_unit: counter port map (clk => clk, reset => ctrl_reset, inc =>
col_timeout, data => row_data, timeout => row_timeout, value => row_value);
  large_cell_unit: large_cell port map (n => n, m => m, s => s, newgen => newgen);


  process(clk, reset)
  begin
    if (reset = '1') then
      current_state <= stopped;
      tnw <= (others => '0'); tn <= (others => '0'); tne <= (others => '0');
      tw <= (others => '0'); tm <= (others => '0'); te <= (others => '0');
      tsw <= (others => '0'); ts <= (others => '0'); tse <= (others => '0');
    elsif (clk'event and clk='1') then
      current_state <= next_state;

      case save_data is
        when l1 =>
          tnw <= life_ram_load_data;
        when l2 =>
          tn <= life_ram_load_data;
        when l3 =>
          tne <= life_ram_load_data;
        when l4 =>
          tw <= life_ram_load_data;
        when l5 =>
          tm <= life_ram_load_data;
        when l6 =>
          te <= life_ram_load_data;
        when l7 =>
          tsw <= life_ram_load_data;
        when l8 =>
          ts <= life_ram_load_data;
        when l9 =>
          tse <= life_ram_load_data;
        when others => null;
      end case;
    end if;
  end process;
```

```
  life_ram_save_data <= newgen;
  life_ram_save_addr <= "00" & row_value(8 downto 0) & col_value(4 downto 0);

  process(reset, current_state, start, mode, sync_counter, row_timeout,
          life_ram_load_read_ack, rowm1, colm1, rowp1, colp1, row_value, col_value)
  begin
    life_ram_save_write_req <= '0';
    life_ram_load_read_req <= '0';
    life_ram_load_addr <= (others => '0');
    ctrl_col_inc <= '0';
    save_data <= paused;
    life_ram_done <= '0';

    if (mode = "00") then
      next_state <= stopped;
      ctrl_reset <= '1';
    else
      ctrl_reset <= reset;
      case current_state is

        when stopped =>
            next_state <= waiting;

         when paused =>                       -- the default behaviour is to wait for start
to be activated.
          life_ram_done <= '1';

          if (start = '1') then
            next_state <= waiting;
          else
            next_state <= paused;
          end if;

        when waiting =>                    -- wait for sync to happen
          if (row_timeout = '1') then
            next_state <= paused;          -- end of calculation of entire map
          elsif (sync_counter = conv_std_logic_vector(31, 10)) then
            next_state <= l1;
          else
            next_state <= waiting;
          end if;

        when l1 =>                         -- load 9 times 32 pixels.
          life_ram_load_read_req <= '1';
          life_ram_load_addr <= "00" & rowm1(8 downto 0) & colm1(4 downto 0);
          if (life_ram_load_read_ack = '1') then
            save_data <= l1;
            next_state <= l2;
          else
            next_state <= l1;
          end if;

        when l2 =>
          life_ram_load_read_req <= '1';
          life_ram_load_addr <= "00" & rowm1(8 downto 0) & col_value(4 downto 0);
          if (life_ram_load_read_ack = '1') then
            save_data <= l2;
            next_state <= l3;
          else
            next_state <= l2;
          end if;

        when l3 =>
          life_ram_load_read_req <= '1';
          life_ram_load_addr <= "00" & rowm1(8 downto 0) & colp1(4 downto 0);
          if (life_ram_load_read_ack = '1') then
            save_data <= l3;
            next_state <= l4;
          else
            next_state <= l3;
          end if;

        when l4 =>
          life_ram_load_read_req <= '1';
          life_ram_load_addr <= "00" & row_value(8 downto 0) & colm1(4 downto 0);
```

```
        if (life_ram_load_read_ack = '1') then
          save_data <= l4;
          next_state <= l5;
        else
          next_state <= l4;
        end if;

      when l5 =>
        life_ram_load_read_req <= '1';
        life_ram_load_addr <= "00" & row_value(8 downto 0) & col_value(4 downto 0);
        if (life_ram_load_read_ack = '1') then
          save_data <= l5;
          next_state <= l6;
        else
          next_state <= l5;
        end if;

      when l6 =>
        life_ram_load_read_req <= '1';
        life_ram_load_addr <= "00" & row_value(8 downto 0) & colp1(4 downto 0);
        if (life_ram_load_read_ack = '1') then
          save_data <= l6;
          next_state <= l7;
        else
          next_state <= l6;
        end if;

      when l7 =>
        life_ram_load_read_req <= '1';
        life_ram_load_addr <= "00" & rowp1(8 downto 0) & colm1(4 downto 0);
        if (life_ram_load_read_ack = '1') then
          save_data <= l7;
          next_state <= l8;
        else
          next_state <= l7;
        end if;

      when l8 =>
        life_ram_load_read_req <= '1';
        life_ram_load_addr <= "00" & rowp1(8 downto 0) & col_value(4 downto 0);
        if (life_ram_load_read_ack = '1') then
          save_data <= l8;
          next_state <= l9;
        else
          next_state <= l8;
        end if;

      when l9 =>
        life_ram_load_read_req <= '1';
        life_ram_load_addr <= "00" & rowp1(8 downto 0) & colp1(4 downto 0);
        if (life_ram_load_read_ack = '1') then
          save_data <= l9;
          next_state <= w;
        else
          next_state <= l9;
        end if;

      when w =>
        next_state <= r;

      when r =>                         -- and simply write result back in ram
        life_ram_save_write_req <= '1';
        ctrl_col_inc <= '1';
        next_state <= waiting;

      when others => null;
    end case;
  end if;
end process;

process(tnw, tn, tne, tw, tm, te, tsw, ts, tse)
begin
  n(33) <= tnw(0); m(33) <= tw(0); s(33) <= tsw(0);
  n(32 downto 1) <= tn(31 downto 0);
  m(32 downto 1) <= tm(31 downto 0);
  s(32 downto 1) <= ts(31 downto 0);
```

```
    n(0) <= tne(31); m(0) <= te(31); s(0) <= tse(31);
  end process;

  process(row_value)
  begin
    if (row_value = conv_std_logic_vector(0, 10)) then
      rowm1 <= conv_std_logic_vector(479, 10);
      rowp1 <= conv_std_logic_vector(1, 10);
    elsif (row_value = conv_std_logic_vector(479, 10)) then
      rowm1 <= conv_std_logic_vector(478, 10);
      rowp1 <= conv_std_logic_vector(0, 10);
    else
      rowm1 <= row_value - conv_std_logic_vector(1, 10);
      rowp1 <= row_value + conv_std_logic_vector(1, 10);
    end if;
  end process;

  process(col_value)
  begin
    if (col_value = conv_std_logic_vector(0, 10)) then
      colm1 <= conv_std_logic_vector(19, 10);
      colp1 <= conv_std_logic_vector(1, 10);
    elsif (col_value = conv_std_logic_vector(19, 10)) then
      colm1 <= conv_std_logic_vector(18, 10);
      colp1 <= conv_std_logic_vector(0, 10);
    else
      colm1 <= col_value - conv_std_logic_vector(1, 10);
      colp1 <= col_value + conv_std_logic_vector(1, 10);
    end if;
  end process;
end synth_life_ram;
```

mem_controller.vhd

```
-- mem_controller.vhd

-- Coded by Alok Menghrajani & Peter Amrhyn
--
-- This is the memory controller. It decides who between the
-- different components (life input, life calc, vga, image writer,
-- image reader, ram tester) gets to access the ram.
--
-- Every read/write is acknowledged (Note: for now the write
-- is acknowledged before (2 clock cycles) the write actually finishes).
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity mem_controller is
  port(clk: in std_logic;
       reset: in std_logic;
       ram_data: inout std_logic_vector(31 downto 0);
       ram_addr: out std_logic_vector(15 downto 0);
       ram_cs: out std_logic;
       ram_we: out std_logic;
        ram_swap: in std_logic;                                    -- this allows us to
select which part in memory to use.
       -- life_input (WRITE)
       life_input_write_req: in std_logic;
       life_input_write_ack: out std_logic;
       life_input_addr: in std_logic_vector(15 downto 0);
       life_input_data: in std_logic_vector(31 downto 0);
       -- img_input (WRITE)
       img_input_write_req: in std_logic;
       img_input_write_ack: out std_logic;
       img_input_addr: in std_logic_vector(15 downto 0);
       img_input_data: in std_logic_vector(31 downto 0);
       -- mem_tester (READ)
       mem_tester_read_req: in std_logic;
       mem_tester_read_ack: out std_logic;
       mem_tester_addr: in std_logic_vector(15 downto 0);
       mem_tester_data: out std_logic_vector(31 downto 0);
```

```
        -- life_ram_load (READ)
        life_ram_load_read_req: in std_logic;
        life_ram_load_read_ack: out std_logic;
        life_ram_load_addr: in std_logic_vector(15 downto 0);
        life_ram_load_data: out std_logic_vector(31 downto 0);
        -- life_ram_save (WRITE)
        life_ram_save_write_req: in std_logic;
        life_ram_save_write_ack: out std_logic;
        life_ram_save_addr: in std_logic_vector(15 downto 0);
        life_ram_save_data: in std_logic_vector(31 downto 0);
        -- vga_ram (READ)
        vga_ram_read_req: in std_logic;
        vga_ram_read_without_mem_swap: in std_logic;
        vga_ram_read_ack: out std_logic;
        vga_ram_addr: in std_logic_vector(15 downto 0);
        vga_ram_data: out std_logic_vector(31 downto 0);
        but2, but3: in std_logic);
end mem_controller;

architecture synth_mem_controller of mem_controller is
  type state is (init, w0, w1, w2, r0);
  signal current_state, next_state: state;
  signal buf_data: std_logic_vector(31 downto 0);
  signal buf_addr: std_logic_vector(15 downto 0);
     type load_from is (none, life_input, img_input, life_ram_load, life_ram_save,
mem_tester, vga_ram);
  signal current_ram : std_logic;
  -- CONTROL SIGNALS
  signal ctrl_buf: load_from;
begin
  ram_addr <= buf_addr;

  process(clk, reset)
  begin
    if (reset='1') then
      current_state <= init;
      buf_data <= (others => '0');
      buf_addr <= (others => '0');
      life_input_write_ack <= '0';
      img_input_write_ack <= '0';
      life_ram_save_write_ack <= '0';
      mem_tester_read_ack <= '0';
      life_ram_load_read_ack <= '0';
      vga_ram_read_ack <= '0';
      current_ram <= '0';
    elsif (clk'event and clk='1') then
      current_state <= next_state;

      life_input_write_ack <= '0';
      img_input_write_ack <= '0';
      life_ram_save_write_ack <= '0';
      mem_tester_read_ack <= '0';
      life_ram_load_read_ack <= '0';
      vga_ram_read_ack <= '0';

      if (ram_swap = '1') then
        current_ram <= not(current_ram);
      end if;

      case ctrl_buf is
        when img_input =>
          buf_addr <= img_input_addr(15 downto 0);
          buf_data <= img_input_data;
          img_input_write_ack <= '1';
        when life_input =>
          buf_addr <= current_ram & life_input_addr(14 downto 0);
          buf_data <= life_input_data;
          life_input_write_ack <= '1';
        when life_ram_load =>
          buf_addr <= current_ram & life_ram_load_addr(14 downto 0);
          life_ram_load_read_ack <= '1';
        when mem_tester =>
          if (but2 = '1') then
            buf_addr <= not(current_ram) & mem_tester_addr(14 downto 0);
          else
            buf_addr <= current_ram & mem_tester_addr(14 downto 0);
```

```
            end if;
            mem_tester_read_ack <= '1';
          when life_ram_save =>
            buf_addr <= not(current_ram) & life_ram_save_addr(14 downto 0);
            buf_data <= life_ram_save_data;
            life_ram_save_write_ack <= '1';
          when vga_ram =>
            if (vga_ram_read_without_mem_swap = '1') then
              buf_addr <= vga_ram_addr(15 downto 0);
            else
              if (but3 = '1') then
                buf_addr <= not(current_ram) & vga_ram_addr(14 downto 0);
              else
                buf_addr <= current_ram & vga_ram_addr(14 downto 0);
              end if;
            end if;
            vga_ram_read_ack <= '1';
          when others => null;
        end case;
      end if;
    end process;

     process(current_state, img_input_write_req, life_input_write_req, mem_tester_read_req,
  life_ram_load_read_req, life_ram_save_write_req,
            buf_addr, buf_data, ram_data, vga_ram_read_req)
    begin
      mem_tester_data <= "00100010001000100010001000100010";    -- this data should
      life_ram_load_data <= "00100010001000100010001000100010"; -- never be used.
      vga_ram_data <= "00100010001000100010001000100010"; -- never be used.


      ram_cs <= '0';
      ram_we <= '0';
      ctrl_buf <= none;

      case current_state is

        when init =>
          ram_data <= (others => 'Z');
          if (life_input_write_req = '1') then
            ctrl_buf <= life_input;
            next_state <= w0;
          elsif (img_input_write_req = '1') then
            ctrl_buf <= img_input;
            next_state <= w0;
          elsif (mem_tester_read_req = '1') then
            ctrl_buf <= mem_tester;
            next_state <= r0;
          elsif (life_ram_load_read_req = '1') then
            ctrl_buf <= life_ram_load;
            next_state <= r0;
          elsif (life_ram_save_write_req = '1') then
            ctrl_buf <= life_ram_save;
            next_state <= w0;
          elsif (vga_ram_read_req = '1') then
            ctrl_buf <= vga_ram;
            next_state <= r0;
          else
            next_state <= init;
          end if;

        when w0 =>
          ram_data <= buf_data;
          ram_cs <= '1';
          ram_we <= '1';
          next_state <= w1;

        when w1 =>
          ram_data <= buf_data;
          ram_cs <= '1';
          ram_we <= '1';
          next_state <= w2;

        when w2 =>
          ram_data <= buf_data;
          ram_cs <= '1';
```

```
        ram_we <= '1';
        next_state <= init;

      when r0 =>
        mem_tester_data <= ram_data;
        life_ram_load_data <= ram_data;
        vga_ram_data <= ram_data;
        ram_cs <= '1';
        next_state <= init;

    end case;
  end process;
end synth_mem_controller;
```

mem_tester.vhd

```
-- mem_tester.vhd

-- Coded by Alok Menghrajani & Peter Amrhyn
--
-- This is a component to test the memory.

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity mem_tester is
  port(clk: in std_logic;
       reset: in std_logic;
        debug_start: in std_logic;        -- when this signal is set to 1, the mem_tester
will start dumping the memory.
       mem_tester_read_req: out std_logic;
       mem_tester_read_ack: in std_logic;
       mem_tester_addr: out std_logic_vector(15 downto 0);
       mem_tester_data: in std_logic_vector(31 downto 0);
       byte_send_done: in std_logic;
       byte_send_data: out std_logic_vector(7 downto 0);
       byte_sending: out std_logic);
end mem_tester;

architecture synth_mem_tester of mem_tester is
  component counter is
    port(clk: in std_logic;
         reset: in std_logic;
         inc: in std_logic;
         data: in std_logic_vector(9 downto 0);
         timeout: out std_logic;
         value: out std_logic_vector(9 downto 0));
  end component;

  type state is (init, waiting, fetching, newline);
  signal current_state, next_state: state;
  signal counter1_timeout, counter2_timeout: std_logic;
   signal counter1_value, counter2_value, counter1_data, counter2_data: std_logic_vector(9
downto 0);
  -- CONTROL SIGNALS
  signal ctrl_inc1, ctrl_inc2: std_logic;
  -- mem_tester_ram_req
  -- rs232_send_byte_req
begin
  -- counter1 is the number of memory cols to display
  -- counter2 is the number of memory lines (rows) to display
  counter1_data <= conv_std_logic_vector(5, 10);
  counter2_data <= conv_std_logic_vector(5, 10);
  mem_tester_addr <= "00" & counter2_value(8 downto 0) & counter1_value(6 downto 2);

  process(clk, reset)
  begin
    if (reset = '1') then
      current_state <= init;
    elsif (clk'event and clk='1') then
      current_state <= next_state;
    end if;
  end process;
```

```
   counter1_unit: counter port map (clk => clk, reset => reset, inc => ctrl_inc1, data =>
counter1_data, timeout => counter1_timeout,
                                    value => counter1_value);
   counter2_unit: counter port map (clk => clk, reset => reset, inc => ctrl_inc2, data =>
counter2_data, timeout => counter2_timeout,
                                    value => counter2_value);

  -- the byte to send will depend on the 2 lower bits of counter1
  process(counter1_value, mem_tester_data)
  begin
    if (counter1_value(1 downto 0) = "11") then
      byte_send_data <= mem_tester_data(7 downto 0);
    elsif (counter1_value(1 downto 0) = "10") then
      byte_send_data <= mem_tester_data(15 downto 8);
    elsif (counter1_value(1 downto 0) = "01") then
      byte_send_data <= mem_tester_data(23 downto 16);
    else  -- counter1_value(1 downto 0) = "00"
      byte_send_data <= mem_tester_data(31 downto 24);
    end if;
  end process;

    process(current_state,  debug_start,  mem_tester_read_ack,  counter1_timeout,
byte_send_done, counter2_timeout)

  begin
    ctrl_inc1 <= '0';
    ctrl_inc2 <= '0';
    mem_tester_read_req <= '0';
    byte_sending <= '0';

    case current_state is
      when init =>
        if (debug_start = '1') then
          next_state <= fetching;
        else
          next_state <= init;
        end if;

      when waiting =>
        -- send data to rs232 and wait
        if (counter1_timeout = '1') then
          next_state <= newline;
        elsif (byte_send_done = '1') then
          next_state <= fetching;
        else
          next_state <= waiting;
        end if;

      when fetching =>
        -- get data from RAM
        if (counter2_timeout = '1') then
          next_state <= init;
        elsif (mem_tester_read_ack = '1') then
          ctrl_inc1 <= '1';
          byte_sending <= '1';
          next_state <= waiting;
        else
          mem_tester_read_req <= '1';
          next_state <= fetching;
        end if;

      when newline =>
        -- wait for previous rs232 to finish (the timeout occured before the byte_
        -- send was done).
        if (byte_send_done = '1') then
          ctrl_inc2 <= '1';
          next_state <= fetching;
        else
          next_state <= newline;
        end if;
      when others => null;
    end case;
  end process;
end synth_mem_tester;
```

ram_swap.vhd

```
-- ram_swap.vhd

-- Coded by Alok Menghrajani & Peter Amrhyn

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity ram_swap is
  port (
    clk, reset: in std_logic;
    is_vsync: in std_logic;
    life_ram_done: in std_logic;
    mode: in std_logic_vector(1 downto 0);
    but4: in std_logic;
    ram_swap: out std_logic);
end ram_swap;

architecture synth_ram_swap of ram_swap is
  type state is (can_swap, swapped, stopped);
  signal current_state, next_state: state;
begin
  process(clk, reset)
  begin
    if (reset = '1') then
      current_state <= stopped;
    elsif (clk'event and clk='1') then
      current_state <= next_state;
    end if;
  end process;

  process(current_state, is_vsync, life_ram_done, mode, but4)
  begin
    ram_swap <= '0';
    if (mode = "00") then
      next_state <= stopped;
    elsif (mode = "01") then
      case current_state is
        when stopped =>
          next_state <= can_swap;

        when can_swap =>
          if ((is_vsync = '1') and (life_ram_done = '1') and (but4 = '0')) then
            ram_swap <= '1';
            next_state <= swapped;
          else
            next_state <= can_swap;
          end if;

        when swapped =>
          if (is_vsync = '0') then
            next_state <= can_swap;
          else
            next_state <= swapped;
          end if;
      end case;
    else
      next_state <= stopped;
    end if;
  end process;
end synth_ram_swap;
```

rs232_multiplex.vhd

```
-- rs232_multiplex.vhd

-- Coded by Alok Menghrajani & Peter Amrhyn

-- This code allows us to simultaneously have
-- an image renderer and game of life running.

-- It basically looks at the first 8 bits,
-- if it's a 0x01 then the data is interpreted for
```

```
-- the game of life.
-- if it's a 0x02 then the data is interpreted for
-- the image renderer.

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity rs232_multiplex is
  port (
    clk, reset: in std_logic;
    rs232_recv_ack: in std_logic;
    rs232_recv_data: in std_logic_vector(7 downto 0);
-- life
    life_recv_ack: out std_logic;
    life_recv_data: out std_logic_vector(7 downto 0);
    life_done: in std_logic;
-- img
    img_recv_ack: out std_logic;
    img_recv_data: out std_logic_vector(7 downto 0);
    img_done: in std_logic);
end rs232_multiplex;

architecture synth_rs232_multiplex of rs232_multiplex is
  type state is (idle, life, img);
  signal current_state, next_state: state;
begin
  process(clk, reset)
  begin
    if (reset='1') then
      current_state <= idle;
    elsif (clk'event and clk='1') then
      current_state <= next_state;
    end if;
  end process;

  life_recv_data <= rs232_recv_data;
  img_recv_data <= rs232_recv_data;

  process(current_state, rs232_recv_ack, rs232_recv_data, life_done, img_done)
  begin
    life_recv_ack <= '0';
    img_recv_ack <= '0';

    case current_state is

      when idle =>
        if (rs232_recv_ack = '1') then
          if (rs232_recv_data = conv_std_logic_vector(1, 8)) then
            next_state <= life;
          elsif (rs232_recv_data = conv_std_logic_vector(2, 8)) then
            next_state <= img;
          else
            next_state <= idle;
          end if;
        else
          next_state <= idle;
        end if;

      when life =>
        if (life_done = '1') then
          next_state <= idle;
        elsif (rs232_recv_ack = '1') then
          life_recv_ack <= '1';
          next_state <= life;
        else
          next_state <= life;
        end if;

      when img =>
        if (img_done = '1') then
          next_state <= idle;
        elsif (rs232_recv_ack = '1') then
          img_recv_ack <= '1';
          next_state <= img;
        else
```

```
          next_state <= img;
        end if;
    end case;
  end process;

end synth_rs232_multiplex;
```

rs232_recv.vhd

```
-- rs232_recv.vhd

-- Coded by Alok Menghrajani & Peter Amrhyn
--
-- This unit allows us to write data into the ram through
-- the serial port. (For now, this unit only reads from the
-- serial port, it doesn't write to it).
-- The input for this unit are the reset, clk and rx.

-- The settings for the terminal should be:
-- 115'000 bauds, 1 start bit, 8 data bits, 1 stop bit,
-- no party bits, no flow control.

-- This unit depends on:
--   counter
--   rs232_recv_shift_register

-- This unit uses two counters (one to keep track of the
-- 115000 bauds tming and one to keep track of the 8 bits
-- to be read).

-- Note: It is very IMPORTANT to have the rx (because it is
-- an external signal) go through a set of registers (2 or 3
-- levels) to avoid metastable states.

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity rs232_recv is
  port(clk: in std_logic;
       reset: in std_logic;
       rx: in std_logic;
       rs232_recv_ack: out std_logic;
       rs232_recv_data: out std_logic_vector(7 downto 0));
end rs232_recv;

architecture synth_rs232_recv of rs232_recv is
  component counter is
  port(clk:            in std_logic;
       reset:          in std_logic;
       inc:            in std_logic;
       data:           in std_logic_vector(9 downto 0);
       timeout:          out std_logic;
       value:          out std_logic_vector(9 downto 0));
  end component;
  component rs232_recv_shift_register is
  port(clk: in std_logic;
       datain: in std_logic;
       shift: in std_logic;
       dataout: out std_logic_vector(7 downto 0));
  end component;

  type state is (init, startbit, datawait, dataread, stopbit, stopbit2);
  signal current_state: state;
  signal next_state: state;

  -- sreg enables the shift register
  signal sreg: std_logic;

  -- unit1 is a counter to respect the 115200 bauds rate.
  -- unit2 is a counter to read 8 bits (and save them in reg).
  signal unit1_counter: std_logic_vector(9 downto 0);
  signal unit2_counter: std_logic_vector(9 downto 0);
  signal unit1_timeout: std_logic;
```

```
  signal unit2_timeout: std_logic;
  signal unit1_enable: std_logic;
  signal unit2_enable: std_logic;
  signal unit1_value: std_logic_vector(9 downto 0);
  signal unit2_value: std_logic_vector(9 downto 0);
begin
  unit1: counter port map(clk => clk,
                          reset => reset,
                          inc => unit1_enable,
                          data => unit1_counter,
                          timeout => unit1_timeout,
                          value => unit1_value);
  unit2: counter port map(clk => clk,
                          reset => reset,
                          inc => unit2_enable,
                          data => unit2_counter,
                          timeout => unit2_timeout,
                          value => unit2_value);
  shiftreg: rs232_recv_shift_register port map(clk => clk,
                                               datain => rx,
                                               shift => sreg,
                                               dataout => rs232_recv_data);
  process(reset, clk)
  begin
    if (reset='1') then
      current_state <= init;
    elsif (clk'event and clk='1') then
      current_state <= next_state;
    end if;
  end process;

  process(current_state, unit1_timeout, unit2_timeout, rx)
  begin
    rs232_recv_ack <= '0'; sreg <= '0';
    unit1_counter <= conv_std_logic_vector(286, 10);
    unit2_counter <= conv_std_logic_vector(7, 10);
    unit1_enable <= '0'; unit2_enable <= '0';
    case current_state is
      when init =>
        if (rx='0') then
          next_state <= startbit;
        else
          next_state <= init;
        end if;
      when startbit =>
        unit1_counter <= conv_std_logic_vector(429, 10);
        unit1_enable <= '1';
        if (unit1_timeout='1') then
          next_state <= dataread;
        else
          next_state <= startbit;
        end if;
      when dataread =>
        sreg <= '1';
        unit2_enable <= '1';
        next_state <= datawait;
      when datawait =>
        unit1_enable <= '1';
        if (unit2_timeout='1') then
          next_state <= stopbit;
        elsif (unit1_timeout='1') then
          next_state <= dataread;
        else
          next_state <= datawait;
        end if;
      when stopbit =>
        rs232_recv_ack <= '1';
        unit1_enable <= '1';
        next_state <= stopbit2;
      when stopbit2 =>
        unit1_enable <= '1';
        if (unit1_timeout='1') then
          next_state <= init;
        else
          next_state <= stopbit2;
        end if;
```

```
      when others => null;
    end case;
  end process;
end synth_rs232_recv;
```

rs232_recv_shift_register.vhd

```
-- rs232_recv_shift_register.vhd

-- Coded by Alok Menghrajani & Peter Amrhyn
--
-- This unit is a simple shift register. It is
-- used by the rs232_recv unit. It does a right shift
-- on 1 bit.

-- Note: There is no reset for this unit.

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity rs232_recv_shift_register is
  port(clk: in std_logic;
       datain: in std_logic;
       shift: in std_logic;
       dataout: out std_logic_vector(7 downto 0));
end rs232_recv_shift_register;

architecture synth_rs232_recv_shift_register of rs232_recv_shift_register is
  signal value: std_logic_vector(7 downto 0);
begin
  dataout <= value;
  process(clk)
  begin
    if (clk'event and clk='1') then
      if (shift='1') then
        value <= datain & value(7 downto 1);
      end if;
    end if;
  end process;
end synth_rs232_recv_shift_register;
```

rs232_send.vhd

```
-- rs232_send.vhd

-- Coded by Alok Menghrajani & Peter Amrhyn
--
-- This unit allows us to dump data onto the terminal
-- for debugging purpose.
--
-- The settings for the terminal should be:
-- 115'000 bauds, 1 start bit, 8 data bits, 1 stop bit,
-- no party bits, no flow control.

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity rs232_send is
  port(clk: in std_logic;
       reset: in std_logic;
       tx: out std_logic;
       rs232_send_done: out std_logic;
       rs232_send_data: in std_logic_vector(7 downto 0);
       rs232_sending: in std_logic);
end rs232_send;

architecture synth_rs232_send of rs232_send is
  component counter is
  port(clk:           in std_logic;
```

```
          reset:          in std_logic;
          inc:            in std_logic;
          data:           in std_logic_vector(9 downto 0);
          timeout:          out std_logic;
          value:          out std_logic_vector(9 downto 0));
    end component;
    component rs232_send_shift_register is

    port(clk: in std_logic;
          reset: in std_logic;
          datain: in std_logic_vector(7 downto 0);
          load: in std_logic;
          shift: in std_logic;
          dataout: out std_logic);
    end component;

    type state is (init, datawait, datawrite, stopbit);
    signal current_state: state;
    signal next_state: state;

    -- sreg enables the shift register
    signal shift_do, shift_load: std_logic;

    -- unit1 is a counter to respect the 115200 bauds rate.
    -- unit2 is a counter to read 8 bits (and save them in reg).
    signal unit1_counter: std_logic_vector(9 downto 0);
    signal unit2_counter: std_logic_vector(9 downto 0);
    signal unit1_timeout: std_logic;
    signal unit2_timeout: std_logic;
    signal unit1_enable: std_logic;
    signal unit2_enable: std_logic;
    signal unit1_value: std_logic_vector(9 downto 0);
    signal unit2_value: std_logic_vector(9 downto 0);
  begin
    unit1: counter port map(clk => clk,
                            reset => reset,
                            inc => unit1_enable,
                            data => unit1_counter,
                            timeout => unit1_timeout,
                            value => unit1_value);
    unit2: counter port map(clk => clk,
                            reset => reset,
                            inc => unit2_enable,
                            data => unit2_counter,
                            timeout => unit2_timeout,
                            value => unit2_value);
    shiftreg: rs232_send_shift_register port map(clk => clk, reset => reset,
                                                 datain => rs232_send_data, load =>
  shift_load,
                                              shift => shift_do, dataout => tx);

    process(reset, clk)
    begin
      if (reset='1') then
        current_state <= init;
      elsif (clk'event and clk='1') then
        current_state <= next_state;
      end if;
    end process;

    process(current_state, unit1_timeout, unit2_timeout, rs232_sending)
    begin
      rs232_send_done <= '0'; shift_do <= '0';
      unit1_counter <= conv_std_logic_vector(286, 10);
      unit2_counter <= conv_std_logic_vector(9, 10);
      unit1_enable <= '0'; unit2_enable <= '0';
      shift_load <= '0';
      case current_state is
        when init =>
          if (rs232_sending = '1') then
            shift_load <= '1';
            next_state <= datawait;
          else
            next_state <= init;
          end if;
        when datawrite =>
```

```
        shift_do <= '1';
        unit2_enable <= '1';
        next_state <= datawait;
      when datawait =>
        unit1_enable <= '1';
        if (unit2_timeout='1') then
          next_state <= stopbit;
        elsif (unit1_timeout='1') then
          next_state <= datawrite;
        else
          next_state <= datawait;
        end if;
      when stopbit =>
        rs232_send_done <= '1';
        next_state <= init;
      when others => null;
    end case;
  end process;
end synth_rs232_send;
```

rs232_send_byte.vhd

```
-- rs232_send_byte.vhd

-- Coded by Alok Menghrajani & Peter Amrhyn

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity rs232_send_byte is
  port(clk: in std_logic;
       reset: in std_logic;
       byte_send_data: in std_logic_vector(7 downto 0);
       byte_sending: in std_logic;
       byte_send_done: out std_logic;
       rs232_send_done: in std_logic;
       rs232_send_data: out std_logic_vector(7 downto 0);
       rs232_sending: out std_logic);
end rs232_send_byte;

architecture synth_rs232_send_byte of rs232_send_byte is
  component counter is
  port(clk:           in std_logic;
       reset:         in std_logic;
       inc:           in std_logic;
       data:          in std_logic_vector(9 downto 0);
       timeout:         out std_logic;
       value:         out std_logic_vector(9 downto 0));
  end component;
  signal counter_inc, counter_timeout: std_logic;
  signal counter_data, counter_value: std_logic_vector(9 downto 0);

  signal shift_do, shift_load: std_logic;
  signal data : std_logic_vector(7 downto 0);
  type state is (init, datawait, datashift, space);

  signal current_state: state;
  signal next_state: state;
begin
  counter_data <= conv_std_logic_vector(7, 10);
counter_unit: counter port map(clk => clk,
                               reset => reset,
                               inc => counter_inc,
                               data => counter_data,
                               timeout => counter_timeout,
                               value => counter_value);
  process(reset, clk)
  begin
    if (reset='1') then
      current_state <= init;
      data <= (others => '0');
    elsif (clk'event and clk='1') then
      current_state <= next_state;
```

```
      if (shift_load = '1') then
        data <= byte_send_data;
      elsif (shift_do = '1') then
        data <= data(6 downto 0) & '0';
      end if;
    end if;
  end process;

  process(data, counter_timeout)
  begin
    if (counter_timeout = '1') then
      rs232_send_data <= conv_std_logic_vector(32, 8);   -- ascii spc
    else
      if (data(7) = '0') then
        rs232_send_data <= "00110000";    -- ascii 0
      else
        rs232_send_data <= "00110001";    -- ascii 1
      end if;
    end if;
  end process;

  process(current_state, byte_sending, counter_timeout, rs232_send_done)
  begin
    shift_do <= '0'; shift_load <= '0';
    rs232_sending <= '0'; counter_inc <= '0';
    byte_send_done <= '0';
    case current_state is
      when init =>
        if (byte_sending = '1') then
          shift_load <= '1';
          next_state <= datawait;
        else
          next_state <= init;
        end if;
      when datashift =>
        shift_do <= '1';
        counter_inc <= '1';
        next_state <= datawait;
      when datawait =>
        rs232_sending <= '1';
        if (rs232_send_done='1') then
          next_state <= datashift;
        elsif (counter_timeout='1') then
          next_state <= space;
        else
          next_state <= datawait;
        end if;
      when space =>
        rs232_sending <= '1';
        if (rs232_send_done = '1') then
          byte_send_done <= '1';
          next_state <= init;
        else
          next_state <= space;
        end if;
      when others => null;
    end case;
  end process;
end synth_rs232_send_byte;
```

rs232_send_shift_register.vhd

```
-- rs232_send_shift_register.vhd

-- Coded by Alok Menghrajani & Peter Amrhyn
--
-- This unit is a simple shift register. It is
-- used by the rs232_send unit. It does a right shift
-- on 1 bit.

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
```

```
entity rs232_send_shift_register is
  port(clk: in std_logic;
       reset: in std_logic;
       datain: in std_logic_vector(7 downto 0);
       load: in std_logic;
       shift: in std_logic;

       dataout: out std_logic);
end rs232_send_shift_register;

architecture synth_rs232_send_shift_register of rs232_send_shift_register is
  signal value: std_logic_vector(9 downto 0);
begin
  dataout <= value(0);
  process(clk, reset)
  begin

    if (reset = '1') then
      value <= (others => '1');
    elsif (clk'event and clk='1') then
      if (load = '1') then
        value(9) <= '1';                -- stop bit
        value(8 downto 1) <= datain;    -- data
        value(0) <= '0';                -- start bit
      elsif (shift = '1') then
        value <= '1' & value(9 downto 1);
      end if;
    end if;
  end process;
end synth_rs232_send_shift_register;
```

vga.vhd

```
-- vga.vhd

-- Coded by Alok Menghrajani & Peter Amrhyn

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity vga is
  port (
    clk, reset: in std_logic;
    hsync, vsync: out std_logic;
    is_hsync, is_vsync, is_sync: out std_logic;
    value1, value2: out std_logic_vector (9 downto 0));
end vga;

architecture synth_vga of vga is
  component vga_controlh is
    port (
      clk: in std_logic;
      reset: in std_logic;
      timeout1: in std_logic;
      data1: out std_logic_vector(9 downto 0);
      hsync: out std_logic;
      inc1: out std_logic;
      inc2: out std_logic;
      sync1: out std_logic);
   end component;

  component vga_controlv is
    port (
      clk: in std_logic;
      reset: in std_logic;
      timeout2: in std_logic;
      data2: out std_logic_vector(9 downto 0);
      sync2: out std_logic;
      vsync: out std_logic);
  end component;

  component counter
    port(clk: in std_logic;
```

```
          reset: in std_logic;
          inc: in std_logic;
          data: in std_logic_vector(9 downto 0);
          timeout: out std_logic;
          value: out std_logic_vector(9 downto 0));
  end component;

  signal data, data1: std_logic_vector(9 downto 0);
  signal inc1, inc2, timeout, timeout1: std_logic;
  signal sync1, sync2: std_logic;
begin
   I0: vga_controlh port map (clk => clk, reset => reset, timeout1 => timeout, data1 =>
data, hsync => hsync,
                              inc1 => inc1, inc2 => inc2, sync1 => sync1);
   I1: vga_controlv port map (clk => clk, reset => reset, timeout2 => timeout1, data2 =>
data1, sync2 => sync2, vsync => vsync);
   counter1: counter port map (clk => clk, reset => reset, inc => inc1, data => data,
timeout => timeout, value => value1);
   counter2: counter port map (clk => clk, reset => reset, inc => inc2, data => data1,
timeout => timeout1, value => value2);

  is_hsync <= sync1;
  is_vsync <= sync2;
  is_sync <= sync1 or sync2;
end synth_vga;
```

vga_controlh.vhd

```
-- vga_controlh.vhd
-- Coded by Alok Menghrajani & Peter Amrhyn

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity vga_controlh is
    port (
      clk: in std_logic;
      reset: in std_logic;
      timeout1: in std_logic;
      data1: out std_logic_vector(9 downto 0);
      hsync: out std_logic;
      inc1: out std_logic;
      inc2: out std_logic;
      sync1: out std_logic);
end vga_controlh;

architecture synth of vga_controlh is
  type state is (hw, hs1, hs2, hs3);
  signal current_state: state;
  signal next_state: state;
begin
  inc1 <= '1';
  process(reset, clk)
  begin
    if (reset='1') then
      current_state <= hw;
    elsif (clk'event and clk='1') then
      current_state <= next_state;
    end if;
  end process;

  process(timeout1, current_state)
  begin
    case current_state is
    when hw =>
      data1 <= conv_std_logic_vector(639, 10);
      inc2 <= '0';
      sync1 <= '0';
      hsync <= '1';
      if (timeout1 = '0') then
        next_state <= hw;
      else
        next_state <= hs1;
        sync1 <= '1';
```

```
        hsync <= '1';
        inc2 <= '0';
        data1 <= conv_std_logic_vector(19, 10);
      end if;
    when hs1 =>
      data1 <= conv_std_logic_vector(19, 10);
      inc2 <= '0';
      sync1 <= '1';
      hsync <= '1';
      if (timeout1 = '0') then
        next_state <= hs1;
      else
        next_state <= hs2;
        data1 <= conv_std_logic_vector(95, 10);
        inc2 <= '0';
        sync1 <= '1';
        hsync <= '0';
      end if;
    when hs2 =>
      data1 <= conv_std_logic_vector(95, 10);
      inc2 <= '0';
      sync1 <= '1';
      hsync <= '0';
      if (timeout1 = '0') then
        next_state <= hs2;
      else
        next_state <= hs3;
        data1 <= conv_std_logic_vector(43, 10);
        inc2 <= '0';
        sync1 <= '1';
        hsync <= '1';
      end if;
    when hs3 =>
      data1 <= conv_std_logic_vector(43, 10);
      inc2 <= '0';
      sync1 <= '1';
      hsync <= '1';
      if (timeout1 = '0') then
        next_state <= hs3;
      else
        next_state <= hw;
        data1 <= conv_std_logic_vector(639, 10);
        inc2 <= '1';
        sync1 <= '0';
        hsync <= '1';
      end if;
    end case;
  end process;
end synth;
```

vga_controlv.vhd

```
-- vga_controlv.vhd
-- Coded by Alok Menghrajani & Peter Amrhyn

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity vga_controlv is
  port(clk: in std_logic;
       timeout2: in std_logic;
       reset: in std_logic;
       sync2: out std_logic;
       vsync: out std_logic;
       data2: out std_logic_vector(9 downto 0));
end vga_controlv;

architecture synth of vga_controlv is
  type state is (vw, vs1, vs2, vs3);
  signal current_state: state;
  signal next_state: state;
begin
  process(reset, clk)
  begin
```

```
    if (reset='1') then
      current_state <= vw;
    elsif (clk'event and clk='1') then
      current_state <= next_state;
    end if;
  end process;

  process(current_state, timeout2)
  begin
    case current_state is
    when vw =>
      data2 <= conv_std_logic_vector(479, 10);
      sync2 <= '0';
      vsync <= '1';
      if (timeout2 = '0') then
        next_state <= vw;
      else
        next_state <= vs1;
        data2 <= conv_std_logic_vector(13, 10);
        sync2 <= '1';
        vsync <= '1';
      end if;
    when vs1 =>
      data2 <= conv_std_logic_vector(13, 10);
      sync2 <= '1';
      vsync <= '1';
      if (timeout2 = '0') then
        next_state <= vs1;
      else
        next_state <= vs2;
        data2 <= conv_std_logic_vector(0, 10);
        sync2 <= '1';
        vsync <= '0';
      end if;
    when vs2 =>
      data2 <= conv_std_logic_vector(0, 10);
      sync2 <= '1';
      vsync <= '0';
      if (timeout2 = '0') then
        next_state <= vs2;
      else
        next_state <= vs3;
        data2 <= conv_std_logic_vector(29, 10);
        sync2 <= '1';
        vsync <= '1';
      end if;
    when vs3 =>
      data2 <= conv_std_logic_vector(29, 10);
      sync2 <= '1';
      vsync <= '1';
      if (timeout2 = '0') then
        next_state <= vs3;
      else
        next_state <= vw;
        data2 <= conv_std_logic_vector(479, 10);
        sync2 <= '0';
        vsync <= '1';
      end if;
    end case;
  end process;
end synth;
```

vga_ram.vhd

```
-- vga_ram.vhd
-- Coded by Alok Menghrajani & Peter Amrhyn

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity vga_ram is
  port (
    clk, reset: in std_logic;
```

```
    hsync_in, vsync_in, issync_in: in std_logic;
    col, row: in std_logic_vector (9 downto 0);
    hsync_out, vsync_out: out std_logic;
    r, g, b: out std_logic;
    vga_ram_read_req: out std_logic;
    vga_ram_read_ack: in std_logic;
    vga_ram_addr: out std_logic_vector(15 downto 0);
    vga_ram_data: in std_logic_vector(31 downto 0);
    vga_ram_read_without_mem_swap: out std_logic;

    mode: in std_logic_vector(1 downto 0);
    sync_counter: in std_logic_vector(9 downto 0));
end vga_ram;

architecture synth_vga_ram of vga_ram is
  component vga_shift_register is
    port(clk: in std_logic;
         reset: in std_logic;
         hsync_in: in std_logic;
         vsync_in: in std_logic;
         issync_in: in std_logic;
         hsync_out32: out std_logic;
         vsync_out32: out std_logic;
         issync_out32: out std_logic;
         hsync_out8: out std_logic;
         vsync_out8: out std_logic;
         issync_out8: out std_logic);
  end component;
  component lfsr is
    port (
      clk, reset: in std_logic;
      rand: out std_logic);
  end component;
  signal buf: std_logic_vector(31 downto 0);
  signal pixelr, pixelg, pixelb: std_logic;
  signal pixels: std_logic_vector(3 downto 0);
  type status is (none, ok, failed);
  signal load_buf: status;
  signal issync32, issync8: std_logic;
  signal rand: std_logic;
  signal hsync32, vsync32, hsync8, vsync8: std_logic;
begin
    vga_shift_register_unit: vga_shift_register port map (clk => clk, reset => reset,
hsync_in => hsync_in,

                                                          vsync_in => vsync_in, issync_in
=> issync_in,
                                                                  hsync_out32 => hsync32,
vsync_out32 => vsync32, issync_out32 => issync32,
                                                                hsync_out8 => hsync8, vsync_out8
=> vsync8, issync_out8 => issync8);

  process(mode, row, col)
  begin
    if (mode = "11") then
      vga_ram_addr <= row(8 downto 0) & col(9 downto 3);
    else
      vga_ram_addr <= "00" & row(8 downto 0) & col(9 downto 5);
    end if;
  end process;
  lfsr_unit: lfsr port map (clk => clk, reset => reset, rand => rand);

  -- output depends on buf and counter's value
  process(buf, sync_counter, mode, rand, pixels)
  begin
    pixels <= (others => '0');
    pixelr <= '0'; pixelg <= '1'; pixelb <= '0';
    if (mode = "00") then
      pixelr <= rand;
      pixelg <= rand;
      pixelb <= rand;
    elsif (mode = "01") then
      -- life
      pixelr <= buf(31 - CONV_INTEGER(sync_counter));
      pixelg <= buf(31 - CONV_INTEGER(sync_counter));
      pixelb <= buf(31 - CONV_INTEGER(sync_counter));
```

```
  elsif (mode = "11") then
    -- img
    if (sync_counter(2 downto 0) = "000") then
      pixels <= buf(31 downto 28);
    elsif (sync_counter(2 downto 0) = "001") then
      pixels <= buf(27 downto 24);
    elsif (sync_counter(2 downto 0) = "010") then
      pixels <= buf(23 downto 20);
    elsif (sync_counter(2 downto 0) = "011") then
      pixels <= buf(19 downto 16);
    elsif (sync_counter(2 downto 0) = "100") then
      pixels <= buf(15 downto 12);
    elsif (sync_counter(2 downto 0) = "101") then
      pixels <= buf(11 downto 8);
    elsif (sync_counter(2 downto 0) = "110") then
      pixels <= buf(7 downto 4);
    elsif (sync_counter(2 downto 0) = "111") then
      pixels <= buf(3 downto 0);
    end if;
    pixelr <= pixels(2);
    pixelg <= pixels(1);
    pixelb <= pixels(0);
  end if;
end process;

process(sync_counter, mode, vga_ram_read_ack)
begin
  vga_ram_read_req <= '0';
  vga_ram_read_without_mem_swap <= '0';
  load_buf <= none;

  if (mode = "01") then
    if (sync_counter = conv_std_logic_vector(30, 10)) then
      vga_ram_read_req <= '1';
      load_buf <= none;
    elsif (sync_counter = conv_std_logic_vector(31, 10)) then
      vga_ram_read_req <= '0';
      if (vga_ram_read_ack = '1') then
        load_buf <= ok;
      else
        load_buf <= failed;
      end if;
    end if;
  elsif (mode = "11") then
    if (sync_counter = conv_std_logic_vector(6, 10)) then
      vga_ram_read_req <= '1';
      vga_ram_read_without_mem_swap <= '1';
      load_buf <= none;
    elsif (sync_counter = conv_std_logic_vector(7, 10)) then
      vga_ram_read_req <= '0';
      if (vga_ram_read_ack = '1') then
        load_buf <= ok;
      else
        load_buf <= failed;
      end if;
    end if;
  end if;
end process;

process(reset, clk)
begin
  if (reset = '1') then
    r <= '0';
    g <= '0';
    b <= '0';
    hsync_out <= '1';
    vsync_out <= '1';
    buf <= "00000000000000000111111111111111";
  elsif (clk'event and clk='1') then
    if (load_buf = ok) then
      buf <= vga_ram_data;
    end if;

    if (mode = "11") then
      hsync_out <= hsync8;
      vsync_out <= vsync8;
```

```
        if (issync8 = '1') then
          r <= '0';
          g <= '0';
          b <= '0';
        else
          r <= pixelr;
          g <= pixelg;
          b <= pixelb;
        end if;
      else
        hsync_out <= hsync32;
        vsync_out <= vsync32;
        if (issync32 = '1') then
          r <= '0';
          g <= '0';
          b <= '0';
        else
          r <= pixelr;
          g <= pixelg;
          b <= pixelb;
        end if;
      end if;
    end if;
  end process;
end synth_vga_ram;
```

vga_shift_register.vhd

```
-- vga_shift_register.vhd
-- Coded by Alok Menghrajani & Peter Amrhyn
library ieee;

use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity vga_shift_register is
  port(clk: in std_logic;
       reset: in std_logic;
       hsync_in: in std_logic;
       vsync_in: in std_logic;
       issync_in: in std_logic;
       hsync_out32: out std_logic;
       vsync_out32: out std_logic;
       issync_out32: out std_logic;
       hsync_out8: out std_logic;
       vsync_out8: out std_logic;
       issync_out8: out std_logic);
end vga_shift_register;

architecture synth of vga_shift_register is
  signal hvalue: std_logic_vector(31 downto 0);
  signal vvalue: std_logic_vector(31 downto 0);
  signal issync: std_logic_vector(31 downto 0);
begin
  process(reset, clk)
  begin

    if (reset='1') then
      hvalue<= (others => '1');
      vvalue <= (others => '1');
      issync <= (others => '0');
    elsif (clk'event and clk='1') then
      hvalue <= hsync_in & hvalue(31 downto 1);
      vvalue <= vsync_in & vvalue(31 downto 1);
      issync <= issync_in & issync(31 downto 1);
    end if;
  end process;
  hsync_out32 <= hvalue(0);
  vsync_out32 <= vvalue(0);
  issync_out32 <= issync(0);

  hsync_out8 <= hvalue(24);
  vsync_out8 <= vvalue(24);
```

```
    issync_out8 <= issync(24);
  end synth;
```