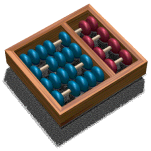

UNICAMP - Instituto de Computação

MC613 - Laboratório de Circuitos Lógicos

Tutorial - Escrita em vídeo VGA

Monitores: Caio Hoffman & George Gondim
Professores: Mario Lúcio Côrtes & Guido Costa Souza de Araújo

Maio de 2013

Sumário

Lista de Tabelas

Conteúdo do Tutorial

1. Utilização o VGA
   (a) Componente VGACON
   (b) Sinais de entrada e saida.
2. Exemplo de utilização
3. Exercícios propostos.

Materiais

- Quartus 9.1 sp2 da Altera.

1 Introdução

O objetivo desse tutorial é fazer uma descrição muito breve de como escrever em um monitor de vídeo usando a saída VGA da placa DE1 da Altera, por meio do componente ***vgacon***. Primeiramente, será apresentado o componente, seus sinais de entrada e saída. Por conseguinte, será exibido um pequeno exemplo.

1.1 Componente VGACON

Tem-se dizer que a utilização do componente não é compulsória, isto é, é apenas uma facilidade e com fim específico de escrever na tela de um monitor com entrada analógica. Em geral, o componente é satisfatório para os propósitos da maioria dos projetos desenvolvidos nesta disciplina.

1.1.1 Por dentro do componente

A maior vantagem de sua utilização é não requer do usuário cuidados com temporização e sincronização. Em poucas palavras, pode-se resumir a utilidade do componente como um "escritor" de *pixels*. No código 1 está a implementação do componente.

Há um detalhe importante. O ***vgacon*** utiliza uma memória cuja instanciação requer um arquivo **.mif** que é um arquivo de descrição de memória. Por causa do tamanho do arquivo ele não é mostrado neste documento, mas está em disponível para download no site da disciplina.

1.1.2 Sinais de Entrada

Os sinais de entrada do componente estão descritos entre as linhas 84 e 89 do código 1. Eles são os *clocks clk27M* e *write_clk*, *reset* ativo baixo *rtsn*, permissão de escrita na memória de vídeo *write_enable*, endereço de escrita *write_addr* e, finalmente, a cor do pixel *data_in* que tem apenas 8 opções. Vale dizer que é possível usar o mesmo valor do *clk27M* para o *write_clk*.

Embora a entidade conte com sinais de entrada *generic* que permitem alterar a

resolução horizontal e vertical da tela, uma simples alteração desses valores não irá funcionar como esperado. Isso, por que o valor da resolução corresponde proporcionalmente ao tamanho da memória de vídeo e, esta, por sua vez, está descrita no arquivo **.mif**. Além disso, a memória usada no componente é bem pequena, menor do 20 KiB. É possível usar outras memórias disponíveis na placa DE1, o que permitiria aumentar a resolução e o número de cores, mas para tanto, é necessário alterar o código 1.

Para escrever o pixel desejado, além da cor escolhida no sinal *data_in*, deve-se calcular a posição desejada (enviada através do sinal *write_addr*), que nada mais é do que um número da natural resultante do produto das posições vertical e horizontal do pixel na tela. A Tabela 1 descreve a associação dos sinais de entrada e os pinos.

Tabela 1: Associação de Pinos de Entrada do ***vgacon***.

Sinal de Entrada	Pino
clk27M	D12
rstn	R22

1.1.3 Sinais de Saída

O sinal *vga_clk* é um sinal que não precisa ser associado a pino de saída algum da placa DE1, portanto, sua não utilização não compromete o funcionamento do componente. Seu usa fica restrito a leitura de sinal, o que normalmente não apresenta necessidade.

Os sinais de saída conforme as linhas 90 e 91 do código 1 serão associados ao pinos de saída da placa DE1 conforme a Tabela 2. Os sinais *hsync* e *vsync* são sinais de sincronização e calculados pelo componente a partir do *clock* de entrada. Como pode-se ver, existem 12 bits para cores (4 bits para cada: *Red*, *Green* e *Blue*), ou seja, 4Ki cores. A despeito disso, o ***vgacon*** só trabalha com 8 cores por conta do uso de uma memória limitada.

Tabela 2: Associação de Pinos de Saídas do ***vgacon***.

Sinal de saída	Pino
red[3]	B7
red[2]	A7
red[1]	C9
red[0]	D9
green[3]	A8
green[2]	B9
green[1]	C10
green[0]	B8
blue[3]	B10
blue[2]	A10
blue[1]	D11
blue[0]	A9
hsync	A11
vsync	B11

2 Exemplo

O exemplo é mostrado no código 2. Seu funcionamento é bem simples: uma linha azul percorre a tela preta enquanto a chave (*switch*) 0 da placa DE1 não está ativa. Quando ativada, a linha para de percorrer a tela e um pixel verde é pintado na posição de parada.

Observando as linhas 31, 32, 41 e 42 nota-se que a resolução da tela não é a padrão do componente ***vgacon***, isto é, o arquivo **.mif** original foi alterado. As linhas 57 à 75 descrevem um processo para divisão da frequência do sinal *clk27M* da entidade ***test***. Esse novo *clock* (*slow_clock*) é usado para escrever os *pixels* azuis no vídeo.

O processo descrito entre as linhas 78 e 93 controla o funcionamento do exemplo. A cada borda de subida do *slow_clock* o endereço é acrescido de uma posição e um pixel azul é escrito. Para executar o exemplo é necessário a atribuição dos pinos mostrada na Tabela 3, ou importar o arquivo **vga_tutorial_pin_assignments.csv** no Quartus.

Tabela 3: Associação de Pinos do Exemplo.

Sinal	Pino	Tipo
clk27M	E12	Entrada
reset	T21	Entrada
switch	L2	Entrada
VGA_R[3]	B7	Saída
VGA_R[2]	A7	Saída
VGA_R[1]	C9	Saída
VGA_R[0]	D9	Saída
VGA_G[3]	A8	Saída
VGA_G[2]	B9	Saída
VGA_G[1]	C10	Saída
VGA_G[0]	B8	Saída
VGA_B[3]	B10	Saída
VGA_B[2]	A10	Saída
VGA_B[1]	D11	Saída
VGA_B[0]	A9	Saída
VGA_HS	A11	Saída
VGA_VS	B11	Saída

3 Exercício Propostos

1. Altere o exemplo para que a linha azul percorra a tela na vertical, ao invés de percorrer a tela na horizontal.

2. Altere o exemplo para que a linha percorra a tela de forma oblíqua e, além disso, altere as cores das linhas em algum momento.

A Códigos VHDL

A.1 Componente VGACON

Código VHDL 1: vgacon.vhd

```
1  ----------------------------------------------------------------------------
2  -- Title      : VGA Controller for DE1 boards
3  -- Project    :
4  ----------------------------------------------------------------------------
5  -- File       : vgacontop.vhd
6  -- Author     : Rafael Auler
7  -- Company    :
8  -- Created    : 2010-03-21
9  -- Last update: 2010-03-26
10 -- Platform   :
11 -- Standard   : VHDL'2008
12 ----------------------------------------------------------------------------
13 -- Description:
14 ----------------------------------------------------------------------------
15 -- Copyright (c) 2010
16 ----------------------------------------------------------------------------
17 -- Revisions  :
18 -- Date        Version  Author          Description
19 -- 2010-03-21  1.0      Rafael Auler  Created
20 -- 2010-03-26  1.1      Rafael Auler    Working 64x60 display w/ internal mem.
21 -- 2010-03-26  1.2      Rafael Auler    Working with arbitrary res. (up to
22 --                                      640x480, tied to on-chip memory
23 --                                      availability). Defaults to 128x96.
24 ----------------------------------------------------------------------------
25
26
27 -- How sync signals are generated for 640x480
28 -- Note: sync signals are active low
29 ----------------------------------------------------------------------------
30 -- Horizontal sync:
31 -- ____________________--________
32 --
33 --              |    |  |       |
34 -- <------------>
35 --    640
36 -- <------------------>
37 --    660
38 -- <---------------------->
39 --    756
40 -- <------------------------------>
41 --    800
42 ----------------------------------------------------------------------------
43 -- Vertical sync:
44 -- ____________________--________
45 --
46 --             |    |  |      |
47 -- <---------->
48 --    480
49 -- <---------------->
50 --    494
51 -- <------------------->
52 --    495
53 -- <------------------------->
54 --    525
55 ----------------------------------------------------------------------------
56
57 ----------------------------------------------------------------------------
58 -- Notes:
59 -- write_clk, write_addr, write_enable and data_in are input signals used to
60 -- write to this controller memory and thus altering the displayed image on VGA.
61 --
```

```
62  -- "data_in" has 3 bits and represents a single image pixel.
63  -- (high bit for RED, middle for GREEN and lower for BLUE - total of 8 colors).
64  --
65  -- These signals follow simple memory write protocol (we=1 writes
66  -- data_in to address (pixel number) write_addr. This last signal may assume
67  -- NUM_HORZ_PIXELS * NUM_VERT_PIXELS different values, corresponding to each
68  -- one of the displayable pixels.
69  ------------------------------------------------------------------------------
70
71  library ieee;
72  use ieee.std_logic_1164.all;
73
74  entity vgacon is
75    generic (
76      --  When changing this, remember to keep 4:3 aspect ratio
77      --  Must also keep in mind that our native resolution is 640x480, and
78      --  you can't cross these bounds (although you will seldom have enough
79      --  on-chip memory to instantiate this module with higher res).
80      NUM_HORZ_PIXELS : natural := 128;  -- Number of horizontal pixels
81      NUM_VERT_PIXELS : natural := 96);  -- Number of vertical pixels
82
83    port (
84      clk27M, rstn              : in  std_logic;
85      write_clk, write_enable   : in  std_logic;
86      write_addr                : in  integer range 0 to
87                                      NUM_HORZ_PIXELS * NUM_VERT_PIXELS - 1;
88      data_in                   : in  std_logic_vector(2 downto 0);
89      vga_clk                   : buffer std_logic;       -- Ideally 25.175 MHz
90      red, green, blue          : out std_logic_vector(3 downto 0);
91      hsync, vsync              : out std_logic);
92  end vgacon;
93
94  architecture behav of vgacon is
95    -- Two signals: one is delayed by one clock cycle. The monitor control uses
96    -- the delayed one. We need a counter 1 clock cycle earlier, relative
97    -- to the monitor signal, in order to index the memory contents
98    -- for the next cycle, when the pixel is in fact sent to the monitor.
99    signal h_count, h_count_d : integer range 0 to 799;  -- horizontal counter
100   signal v_count, v_count_d : integer range 0 to 524;  -- vertical counter
101   -- We only want to address HORZ*VERT pixels in memory
102   signal read_addr : integer range 0 to NUM_HORZ_PIXELS * NUM_VERT_PIXELS - 1;
103   signal h_drawarea, v_drawarea, drawarea : std_logic;
104   signal data_out : std_logic_vector(2 downto 0);
105 begin  -- behav
106
107   -- This is our PLL (Phase Locked Loop) to divide the DE1 27 MHz
108   -- clock and produce a 25.2MHz clock adequate to our VGA controller
109   divider: work.vga_pll port map (clk27M, vga_clk);
110
111   -- This is our dual clock RAM. We use our VGA clock to read contents from
112   -- memory (pixel color value). The user of this module may use any clock
113   -- to write contents to this memory, modifying pixels individually.
114   vgamem : work.dual_clock_ram
115   generic map (
116     MEMSIZE => NUM_HORZ_PIXELS * NUM_VERT_PIXELS)
117   port map (
118     read_clk       => vga_clk,
119     write_clk      => write_clk,
120     read_address   => read_addr,
121     write_address  => write_addr,
122     data_in        => data_in,
123     data_out       => data_out,
124     we             => write_enable);
125
126   -- purpose: Increments the current horizontal position counter
127   -- type   : sequential
128   -- inputs : vga_clk, rstn
129   -- outputs: h_count, h_count_d
130   horz_counter: process (vga_clk, rstn)
131   begin  -- process horz_counter
```

```
132     if rstn = '0' then                  -- asynchronous reset (active low)
133       h_count <= 0;
134       h_count_d <= 0;
135     elsif vga_clk'event and vga_clk = '1' then  -- rising clock edge
136       h_count_d <= h_count;             -- 1 clock cycle delayed counter
137       if h_count = 799 then
138         h_count <= 0;
139       else
140         h_count <= h_count + 1;
141       end if;
142     end if;
143   end process horz_counter;
144
145   -- purpose: Determines if we are in the horizontal "drawable" area
146   -- type    : combinational
147   -- inputs : h_count_d
148   -- outputs: h_drawarea
149   horz_sync: process (h_count_d)
150   begin  -- process horz_sync
151     if h_count_d < 640 then
152       h_drawarea <= '1';
153     else
154       h_drawarea <= '0';
155     end if;
156   end process horz_sync;
157
158   -- purpose: Increments the current vertical counter position
159   -- type    : sequential
160   -- inputs : vga_clk, rstn
161   -- outputs: v_count, v_count_d
162   vert_counter: process (vga_clk, rstn)
163   begin  -- process vert_counter
164     if rstn = '0' then                  -- asynchronous reset (active low)
165       v_count <= 0;
166       v_count_d <= 0;
167     elsif vga_clk'event and vga_clk = '1' then  -- rising clock edge
168       v_count_d <= v_count;             -- 1 clock cycle delayed counter
169       if h_count = 699 then
170         if v_count = 524 then
171           v_count <= 0;
172         else
173           v_count <= v_count + 1;
174         end if;
175       end if;
176     end if;
177   end process vert_counter;
178
179   -- purpose: Updates information based on vertical position
180   -- type    : combinational
181   -- inputs : v_count_d
182   -- outputs: v_drawarea
183   vert_sync: process (v_count_d)
184   begin  -- process vert_sync
185     if v_count_d < 480 then
186       v_drawarea <= '1';
187     else
188       v_drawarea <= '0';
189     end if;
190   end process vert_sync;
191
192   -- purpose: Generates synchronization signals
193   -- type    : combinational
194   -- inputs : v_count_d, h_count_d
195   -- outputs: hsync, vsync
196   sync: process (v_count_d, h_count_d)
197   begin  -- process sync
198     if (h_count_d >= 659) and (h_count_d <= 755) then
199       hsync <= '0';
200     else
201       hsync <= '1';
```

```
202     end if;
203     if (v_count_d >= 493) and (v_count_d <= 494) then
204       vsync <= '0';
205     else
206       vsync <= '1';
207     end if;
208   end process sync;
209
210   -- determines whether we are in drawable area on screen a.t.m.
211   drawarea <= v_drawarea and h_drawarea;
212
213   -- purpose: calculates the controller memory address to read pixel data
214   -- type   : combinational
215   -- inputs : h_count, v_count
216   -- outputs: read_addr
217   gen_r_addr: process (h_count, v_count)
218   begin  -- process gen_r_addr
219     read_addr <= h_count / (640 / NUM_HORZ_PIXELS)
220                  + ((v_count/(480 / NUM_VERT_PIXELS))
221                     * NUM_HORZ_PIXELS);
222   end process gen_r_addr;
223
224   -- Build color signals based on memory output and "drawarea" signal
225   -- (if we are not in the drawable area of 640x480, must deassert all
226   --  color signals).
227   red   <= (others => data_out(2) and drawarea);
228   green <= (others => data_out(1) and drawarea);
229   blue  <= (others => data_out(0) and drawarea);
230
231 end behav;
232
233
234 ---------------------------------------------------------------------------
235 -- The following entity is a dual clock RAM (read operates at different
236 -- clock from write). This is used to isolate two clock domains. The first
237 -- is the 25.2 MHz clock domain in which our VGA controller needs to operate.
238 -- This is the read clock, because we read from this memory to determine
239 -- the color of a pixel. The second is the clock domain of the user of this
240 -- module, writing in the memory the contents it wants to display in the VGA.
241 ---------------------------------------------------------------------------
242
243 library ieee;
244 use ieee.std_logic_1164.all;
245
246 entity dual_clock_ram is
247
248   generic (
249     MEMSIZE : natural);
250   port (
251     read_clk, write_clk         : in  std_logic;  -- support different clocks
252     data_in                     : in  std_logic_vector(2 downto 0);
253     write_address, read_address : in  integer range 0 to MEMSIZE - 1;
254     we                          : in  std_logic;  -- write enable
255     data_out                    : out std_logic_vector(2 downto 0));
256
257 end dual_clock_ram;
258
259 architecture behav of dual_clock_ram is
260   -- we only want to address (store) MEMSIZE elements
261   subtype addr is integer range 0 to MEMSIZE - 1;
262   type mem is array (addr) of std_logic_vector(2 downto 0);
263   signal ram_block : mem;
264   -- we don't care with read after write behavior (whether ram reads
265   -- old or new data in the same cycle).
266   attribute ramstyle : string;
267   attribute ramstyle of dual_clock_ram : entity is "no_rw_check";
268   attribute ram_init_file : string;
269   attribute ram_init_file of ram_block : signal is "vga_mem.mif";
270
271 begin  -- behav
```

```
272
273   -- purpose: Reads data from RAM
274   -- type    : sequential
275   -- inputs : read_clk, read_address
276   -- outputs: data_out
277   read: process (read_clk)
278   begin  -- process read
279     if read_clk'event and read_clk = '1' then  -- rising clock edge
280       data_out <= ram_block(read_address);
281     end if;
282   end process read;
283
284   -- purpose: Writes data to RAM
285   -- type    : sequential
286   -- inputs : write_clk, write_address
287   -- outputs: ram_block
288   write: process (write_clk)
289   begin  -- process write
290     if write_clk'event and write_clk = '1' then  -- rising clock edge
291       if we = '1' then
292         ram_block(write_address) <= data_in;
293       end if;
294     end if;
295   end process write;
296
297 end behav;
298
299 ---------------------------------------------------------------------------
300 -- The following entity is automatically generated by Quartus (a megafunction).
301 -- As Altera DE1 board does not have a 25.175 MHz, but a 27 Mhz, we
302 -- instantiate a PLL (Phase Locked Loop) to divide out 27 MHz clock
303 -- and reach a satisfiable 25.2MHz clock for our VGA controller (14/15 ratio)
304 ---------------------------------------------------------------------------
305
306 LIBRARY ieee;
307 USE ieee.std_logic_1164.all;
308
309 LIBRARY altera_mf;
310 USE altera_mf.all;
311
312 ENTITY vga_pll IS
313   PORT
314   (
315     inclk0    : IN STD_LOGIC  := '0';
316     c0    : OUT STD_LOGIC
317   );
318 END vga_pll;
319
320
321 ARCHITECTURE SYN OF vga_pll IS
322
323   SIGNAL sub_wire0  : STD_LOGIC_VECTOR (5 DOWNTO 0);
324   SIGNAL sub_wire1  : STD_LOGIC ;
325   SIGNAL sub_wire2  : STD_LOGIC ;
326   SIGNAL sub_wire3  : STD_LOGIC_VECTOR (1 DOWNTO 0);
327   SIGNAL sub_wire4_bv : BIT_VECTOR (0 DOWNTO 0);
328   SIGNAL sub_wire4  : STD_LOGIC_VECTOR (0 DOWNTO 0);
329
330
331
332   COMPONENT altpll
333   GENERIC (
334     clk0_divide_by    : NATURAL;
335     clk0_duty_cycle   : NATURAL;
336     clk0_multiply_by    : NATURAL;
337     clk0_phase_shift    : STRING;
338     compensate_clock    : STRING;
339     inclk0_input_frequency    : NATURAL;
340     intended_device_family    : STRING;
341     lpm_hint    : STRING;
```

```
342     lpm_type    : STRING;
343     operation_mode    : STRING;
344     port_activeclock    : STRING;
345     port_areset   : STRING;
346     port_clkbad0    : STRING;
347     port_clkbad1    : STRING;
348     port_clkloss    : STRING;
349     port_clkswitch    : STRING;
350     port_configupdate   : STRING;
351     port_fbin   : STRING;
352     port_inclk0   : STRING;
353     port_inclk1   : STRING;
354     port_locked   : STRING;
355     port_pfdena   : STRING;
356     port_phasecounterselect   : STRING;
357     port_phasedone    : STRING;
358     port_phasestep    : STRING;
359     port_phaseupdown    : STRING;
360     port_pllena   : STRING;
361     port_scanaclr   : STRING;
362     port_scanclk    : STRING;
363     port_scanclkena   : STRING;
364     port_scandata   : STRING;
365     port_scandataout    : STRING;
366     port_scandone   : STRING;
367     port_scanread   : STRING;
368     port_scanwrite    : STRING;
369     port_clk0   : STRING;
370     port_clk1   : STRING;
371     port_clk2   : STRING;
372     port_clk3   : STRING;
373     port_clk4   : STRING;
374     port_clk5   : STRING;
375     port_clkena0    : STRING;
376     port_clkena1    : STRING;
377     port_clkena2    : STRING;
378     port_clkena3    : STRING;
379     port_clkena4    : STRING;
380     port_clkena5    : STRING;
381     port_extclk0    : STRING;
382     port_extclk1    : STRING;
383     port_extclk2    : STRING;
384     port_extclk3    : STRING
385   );
386   PORT (
387       inclk : IN STD_LOGIC_VECTOR (1 DOWNTO 0);
388       clk : OUT STD_LOGIC_VECTOR (5 DOWNTO 0)
389   );
390   END COMPONENT;
391
392 BEGIN
393   sub_wire4_bv(0 DOWNTO 0) <= "0";
394   sub_wire4    <= To_stdlogicvector(sub_wire4_bv);
395   sub_wire1    <= sub_wire0(0);
396   c0    <= sub_wire1;
397   sub_wire2    <= inclk0;
398   sub_wire3    <= sub_wire4(0 DOWNTO 0) & sub_wire2;
399
400   altpll_component : altpll
401   GENERIC MAP (
402     clk0_divide_by => 15,
403     clk0_duty_cycle => 50,
404     clk0_multiply_by => 14,
405     clk0_phase_shift => "0",
406     compensate_clock => "CLK0",
407     inclk0_input_frequency => 37037,
408     intended_device_family => "Cyclone␣II",
409     lpm_hint => "CBX_MODULE_PREFIX=vga_pll",
410     lpm_type => "altpll",
411     operation_mode => "NORMAL",
```

```
412     port_activeclock => "PORT_UNUSED",
413     port_areset => "PORT_UNUSED",
414     port_clkbad0 => "PORT_UNUSED",
415     port_clkbad1 => "PORT_UNUSED",
416     port_clkloss => "PORT_UNUSED",
417     port_clkswitch => "PORT_UNUSED",
418     port_configupdate => "PORT_UNUSED",
419     port_fbin => "PORT_UNUSED",
420     port_inclk0 => "PORT_USED",
421     port_inclk1 => "PORT_UNUSED",
422     port_locked => "PORT_UNUSED",
423     port_pfdena => "PORT_UNUSED",
424     port_phasecounterselect => "PORT_UNUSED",
425     port_phasedone => "PORT_UNUSED",
426     port_phasestep => "PORT_UNUSED",
427     port_phaseupdown => "PORT_UNUSED",
428     port_pllena => "PORT_UNUSED",
429     port_scanaclr => "PORT_UNUSED",
430     port_scanclk => "PORT_UNUSED",
431     port_scanclkena => "PORT_UNUSED",
432     port_scandata => "PORT_UNUSED",
433     port_scandataout => "PORT_UNUSED",
434     port_scandone => "PORT_UNUSED",
435     port_scanread => "PORT_UNUSED",
436     port_scanwrite => "PORT_UNUSED",
437     port_clk0 => "PORT_USED",
438     port_clk1 => "PORT_UNUSED",
439     port_clk2 => "PORT_UNUSED",
440     port_clk3 => "PORT_UNUSED",
441     port_clk4 => "PORT_UNUSED",
442     port_clk5 => "PORT_UNUSED",
443     port_clkena0 => "PORT_UNUSED",
444     port_clkena1 => "PORT_UNUSED",
445     port_clkena2 => "PORT_UNUSED",
446     port_clkena3 => "PORT_UNUSED",
447     port_clkena4 => "PORT_UNUSED",
448     port_clkena5 => "PORT_UNUSED",
449     port_extclk0 => "PORT_UNUSED",
450     port_extclk1 => "PORT_UNUSED",
451     port_extclk2 => "PORT_UNUSED",
452     port_extclk3 => "PORT_UNUSED"
453   )
454   PORT MAP (
455     inclk => sub_wire3,
456     clk => sub_wire0
457   );
458
459
460
461 END SYN;
```

A.2 Exemplo

Código VHDL 2: Exemplo, arquivo: test.vhd.

```
1  LIBRARY ieee;
2  USE ieee.std_logic_1164.ALL;
3
4  ENTITY test IS
5    PORT (
6      switch        : IN STD_LOGIC;
7      clk27M        : IN STD_LOGIC;
8      reset       : IN STD_LOGIC;
9      VGA_R, VGA_G, VGA_B : OUT STD_LOGIC_VECTOR (3 DOWNTO 0);
10     VGA_HS, VGA_VS    : OUT STD_LOGIC
11   );
12 END ENTITY;
13
```

```
14  ARCHITECTURE behavior OF TEST IS
15    COMPONENT vgacon IS
16      GENERIC (
17        NUM_HORZ_PIXELS : NATURAL := 128; -- Number of horizontal pixels
18        NUM_VERT_PIXELS : NATURAL := 96   -- Number of vertical pixels
19      );
20      PORT (
21        clk27M, rstn              : IN STD_LOGIC;
22        write_clk, write_enable   : IN STD_LOGIC;
23        write_addr                : IN INTEGER RANGE 0 TO NUM_HORZ_PIXELS * NUM_VERT_PIXELS -
              1;
24        data_in                   : IN STD_LOGIC_VECTOR (2 DOWNTO 0);
25        red, green, blue          : OUT STD_LOGIC_VECTOR (3 DOWNTO 0);
26        hsync, vsync              : OUT STD_LOGIC
27      );
28    END COMPONENT;
29
30    CONSTANT CONS_CLOCK_DIV : INTEGER := 1000000;
31    CONSTANT HORZ_SIZE : INTEGER := 160;
32    CONSTANT VERT_SIZE : INTEGER := 120;
33
34    SIGNAL slow_clock : STD_LOGIC;
35
36    SIGNAL clear_video_address  ,
37      normal_video_address  ,
38      video_address     : INTEGER RANGE 0 TO HORZ_SIZE * VERT_SIZE - 1;
39
40    SIGNAL clear_video_word   ,
41      normal_video_word   ,
42      video_word        : STD_LOGIC_VECTOR (2 DOWNTO 0);
43
44    TYPE VGA_STATES IS (NORMAL, CLEAR);
45    SIGNAL state : VGA_STATES;
46  BEGIN
47    vga_component: vgacon
48    GENERIC MAP (
49      NUM_HORZ_PIXELS => HORZ_SIZE,
50      NUM_VERT_PIXELS => VERT_SIZE
51    ) PORT MAP (
52      clk27M      => clk27M   ,
53      rstn      => reset    ,
54      write_clk   => clk27M   ,
55      write_enable  => '1'      ,
56      write_addr      => video_address,
57      data_in         => video_word ,
58      red       => VGA_R    ,
59      green     => VGA_G    ,
60      blue      => VGA_B    ,
61      hsync     => VGA_HS   ,
62      vsync     => VGA_VS
63    );
64
65    video_word <= normal_video_word WHEN state = NORMAL ELSE clear_video_word;
66
67    video_address <= normal_video_address WHEN state = NORMAL ELSE clear_video_address;
68
69    clock_divider:
70    PROCESS (clk27M, reset)
71      VARIABLE i : INTEGER := 0;
72    BEGIN
73      IF (reset = '0') THEN
74        i := 0;
75        slow_clock <= '0';
76      ELSIF (rising_edge(clk27M)) THEN
77        IF (i <= CONS_CLOCK_DIV/2) THEN
78          i := i + 1;
79          slow_clock <= '0';
80        ELSIF (i < CONS_CLOCK_DIV-1) THEN
81          i := i + 1;
82          slow_clock <= '1';
```

```
83        ELSE
84          i := 0;
85        END IF;
86      END IF;
87    END PROCESS;
88
89    vga_clear:
90    PROCESS(clk27M, reset, clear_video_address)
91    BEGIN
92      IF (reset = '0') THEN
93        state <= CLEAR;
94        clear_video_address <= 0;
95        clear_video_word <= "000";
96      ELSIF (rising_edge(clk27M)) THEN
97        CASE state IS
98          WHEN CLEAR =>
99            clear_video_address <= clear_video_address + 1;
100           clear_video_word <= "000";
101           IF (clear_video_address < HORZ_SIZE * VERT_SIZE-1) THEN
102             state <= CLEAR;
103           ELSE
104             state <= NORMAL;
105           END IF;
106         WHEN NORMAL =>
107           state <= NORMAL;
108       END CASE;
109     END IF;
110   END PROCESS;
111
112   vga_writer:
113   PROCESS (slow_clock, reset, normal_video_address)
114   BEGIN
115     IF (reset = '0') THEN
116       normal_video_address <= 0;
117       normal_video_word <= "000";
118     ELSIF (rising_edge(slow_clock)) THEN
119       CASE switch IS
120         WHEN '1' =>
121           normal_video_address <= normal_video_address + 1;
122           normal_video_word <= "001";
123         WHEN OTHERS =>
124           normal_video_word <= "010";
125       END CASE;
126     END IF;
127   END PROCESS;
128 END ARCHITECTURE;
```