بسم الله الرحمـ ـن الرحيـ ـم

In the name of God most gracious most merciful

Alexandria University
Faculty of Engineering
Communications and Electronics department

VHDL IMPLEMENTATION OF OSCILLOSCOPE

USING

FPGA

Team Work

Amr Mohamed Reda Mallah
Fady Mahmoud Samy Yacout
Kareem Ahmad Abd el Fattah
Kareem Aly Morsy
Mohammed Gamal el Shahawy
Mazen Abd el Aziz Aswa
Sherief Ezz el Din Zakzouk

http://groups.yahoo.com/group/Radjab/
Radjab@yahoogroups.com

Abstract

The objective of the system is to make a digital oscilloscope. The oscilloscope was not to be interfaced to a computer, as many of the scope cards available today in the market. Instead it would be directly interfaced to display the signal on a VGA monitor, without the need for a computer. First, the development of the system was divided in to two stages. The first was the analog to digital conversion which is common to any digital oscilloscope, while the second was the interface with the monitor to display the converted analog signal. The whole system was to implemented using VHDL, the system will be broken down to a number of modules, each to carry out a certain task. In the end; the modules will be integrated together using a schematic.

Acknowledgements

We would like to thank Dr Hossam El-Din for his efforts in assisting us

with this project, and for the use of his FPGA.

And special thanks to Khaled Ramadan for his monitor.

Many thanks must also go to

our parents thank you for supporting us throughout this endeavor.

Introduction

Starting with first stage of design (A/D) we developed the oscilloscope module. This module will take the place of most of the physical IC's used in a digital oscilloscope PCI slot card. Looking at a schematic of a digital oscilloscope, we wrote the VHDL code to replace the physical chips. They were connected together in the same manner as in real life but by using a schematic representation of each code. By connecting the FPGA to the remaining IC's the oscilloscope module was formed. The basic idea of this module is to allow control of the A/D and to store the converted signal on a memory (later referred to as "oscilloscope memory"). The oscilloscope will work in two modes, first is the write mode during which conversion takes place and the storage of the data to the memory is done. The second mode is the read mode, where stored values on the memory are fed to the following stage of the system. Before a cycle would start; control signals to the clock divider must be set to determine the sampling frequency used to drive the analog to digital converter (ADC). During a write cycle, control signal are sent to the memory to allow writing to it. The clock is fed to a counter that will generate the memory's sequential addresses. Therefore the whole memory will be filled with the converted values of the signal. The roll over of the counter most significant bit will indicate the end if the write cycle and the start of the read cycle. Now the ADC is deactivated and memory control signals are set for read from the memory and the clock driving the address generating counter will be fed from an external source coming from the following stage that will receive the data. This is to allow synchronization between the two stages.

Moving on to the second stage of development, we want to develop an interface between a memory and the VGA monitor, where data stored on the memory will be displayed on the screen studying the concept of how a VGA monitor works , a program developed that will generate all the necessary signals that will control the display. A VGA monitor displays frames of information with precise timing. Each frame is made up of lines and each line is made up of pixels. Synchronization signals are needed to move form pixel to pixel and from line to line. Horizontal and vertical synchronization signals were generated by calculating there exact timing. Other signals are RGB that controls the color of each pixel. The data of each pixel is stored in two bits in the memory (later referred to as "VGA memory"). These data a sent to a D/A converter that changes the data to analog signals that are accepted by the VGA monitor. A VGA generator module was developed to carry out the receding task.

This meant that the data obtained from the first stage (oscilloscope module) could not be sent directly to the VGA generator. This gave rise to the converter module. This module will be connected to the oscilloscope module while operating in the

read mode, then perform some processing to the data to change it to the pixel form accepted by the VGA generator module.
On the VGA monitor screen, the line in which a pixel is on; will determine its value (amplitude) i.e. a pixel on the top line will have a higher value than a pixel on the bottom line.   While its position in the line (left or right of screen) will determine its time, i.e. a pixel on the left of the screen is before (in time) a pixel on the right. Thus the address of a certain pixel in the VGA memory can be determined depending on the value stored in the oscilloscope memory location and the address of this location. A location at the bottom of the oscilloscope memory will be displayed on the left of the screen while a location on top of the oscilloscope memory will be displayed on the right of the screen. Locations with high values (e.g. all 1's) stored in; will be displayed near the top of the screen, while locations with small values (e.g. all 0's) will be displayed near the bottom of the screen. Therefore reading sequentially through the oscilloscope memory from bottom to top will fill locations of pixels from the left to right of the screen. Every memory location in the oscilloscope memory will be represented by a pixel on the screen, which is stored in two bits in the VGA memory. By knowing the number of lines on the screen and the number of levels (resolution) of the ADC a mathematical relation can be deduced to determine the location of the two bits specified for the pixel to be lit to draw the signal on the screen. Another function performed by the converter module is the reset of the locations in the VGA memory after the display of the frame, to allow the storage of the next frame on a blank screen.

Now the system is ready to be integrated to form the digital oscilloscope. But still all the modules must be synchronized to allow the flow of data correctly in the right direction. First conversion of the signal to digital form, then conversion of the digital data in to pixel bits, then display of the data on the screen. Therefore some synchronization was needed to allow control over the whole system. This is the task of the controller module. This module acts like a traffic light for the flow of data. First it signals the start of the A/D conversion by operating the oscilloscope in the write mode. When the write cycle ends, a signal is sent to the controller module, which in turn sends a signal to the converter module to start blanking (reset) of the VGA memory. Then the conversion of the data to pixel bits by operating the oscilloscope module in the read mode. This only start after the converter modules signals the end of its blanking cycle. After conversion end, the data stored in the VGA memory is ready to be displayed. Therefore the controller module sends a signal to the VGA generator module to start display, therefore accessing the stored pixel bits in the VGA memory. When the display is over a signal is sent to the controller module thus starting the whole cycle again, producing a dynamic picture on the screen of the analog signal.

Software Blocks

Oscilloscope

Role of the Oscilloscope

The oscilloscope module is the part of the system which deals with the analog signal and converts it in to a digital form that is then stored in to a memory. This module is made up of a multiplexer, a counter and a clock divider. The multiplexer switches between two states according to the oscilloscope's mode of operation. Thus feeding the appropriate control signals to the memory and the clock to the counter that will maintain synchronization with the rest of the system. The counter generates sequential memory addresses during both the read and writes cycles with speeds depending on its inputted clock. The counter has an enable signal that will activate and deactivate it accordingly to allow synchronization with the rest of the system.

Oscilloscope modes of operation

- **Write mode:** During this mode the oscilloscope memory is being filled with the signals amplitudes generated by the analog to digital converter. The data are filled sequentially with every clock cycle. Therefore the oscilloscope memory contains the time varying amplitude of the input signal.
- **Read mode:** Now the stored data in the oscilloscope memory will be passed forward to the rest of the system to be processed and made ready for display on the VGA screen.

Write Cycle

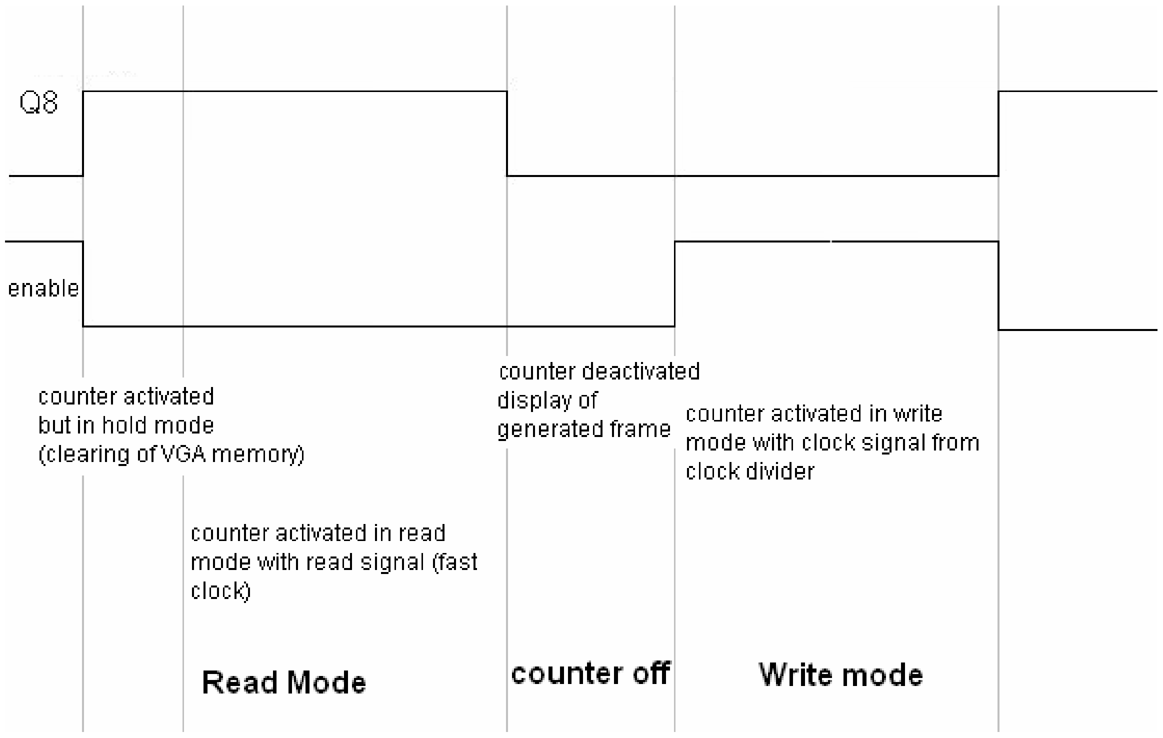
During a write cycle, first of all the output of the counter is reset ("000000000") therefore Q8=0 which is connected to the multiplexers select input that determines which set of control signals are passes to its output port Y. Y0 will be the clock, which is inputted from the clock divider. The clock divider also sends this clock to the analog to digital converter (ADC) which keeps the counter and ADC synchronized. Y1='1' which deactivates the output enable of the memory. Y2=clock which will enable writing to the memory in synchronous with the output data from the ADC. This cycle will repeat until Q8 goes high which means all the 256 memory location in the oscilloscope memory have been filled. This changes the select input of the multiplexer to '1'. This signals the end of the write cycle and the beginning of the read cycle.

Read Cycle

During the read cycle, Q8='1' which brings read control signals to the multiplexers output port. Y0 will equal the read signal, which is generated from the following stage that will read the stored data (Converter/frame generator). Y1='0' which activates the output enable of the oscilloscope. Y2='1', thus deactivating writing to the memory. The generated oscilloscope address from the counter is now controlled by the read signal. Therefore the converter module can keep the counter in a hold mode, by keeping the read signal constant, while performing other tasks(clearing of the VGA memory), then sending a fast clock to read through the whole memory.

Deactivation of counter

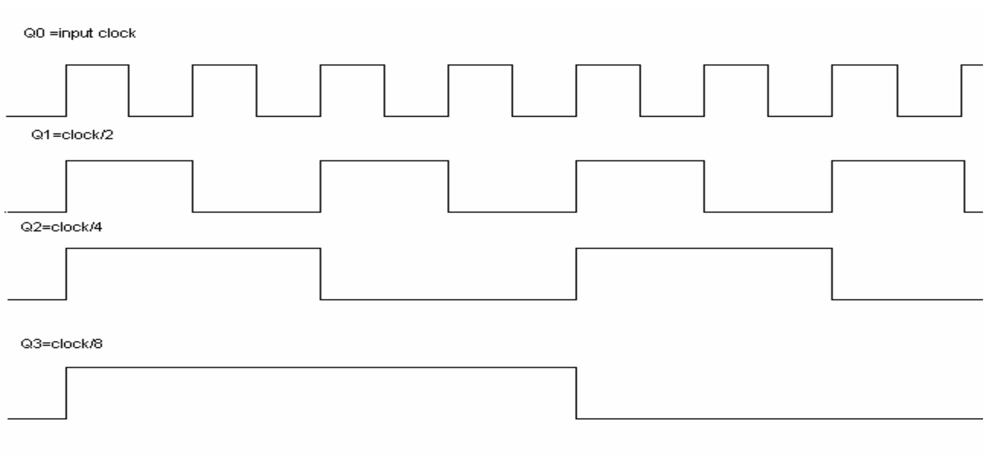
In the end of the read cycle Q8 will equal to '0', this should start the write cycle, but not until the rest of the system is ready to proceed. Now comes the role of the enable input to the counter. This signal determines when the counter is active and when its not. The counter is active in both the read and write cycles i.e. when Q8='1' (read cycle) and during the write cycle only if the rest of the system is ready to proceed. Therefore the active condition is when Q8='1' OR enable='1' (i.e. the system is ready to proceed with another write cycle). This therefore actives the counter during both the read and the write cycles, each with its appropriate clock, and deactivates it when the memory is not in use. This means the counter is deactivated only when the system is working on the display of a generated frame.


Q8
enable
counter activated but in hold mode (clearing of VGA memory)
counter activated in read mode with read signal (fast clock)
counter deactivated display of generated frame
counter activated in write mode with clock signal from clock divider
Read Mode
counter off
Write mode

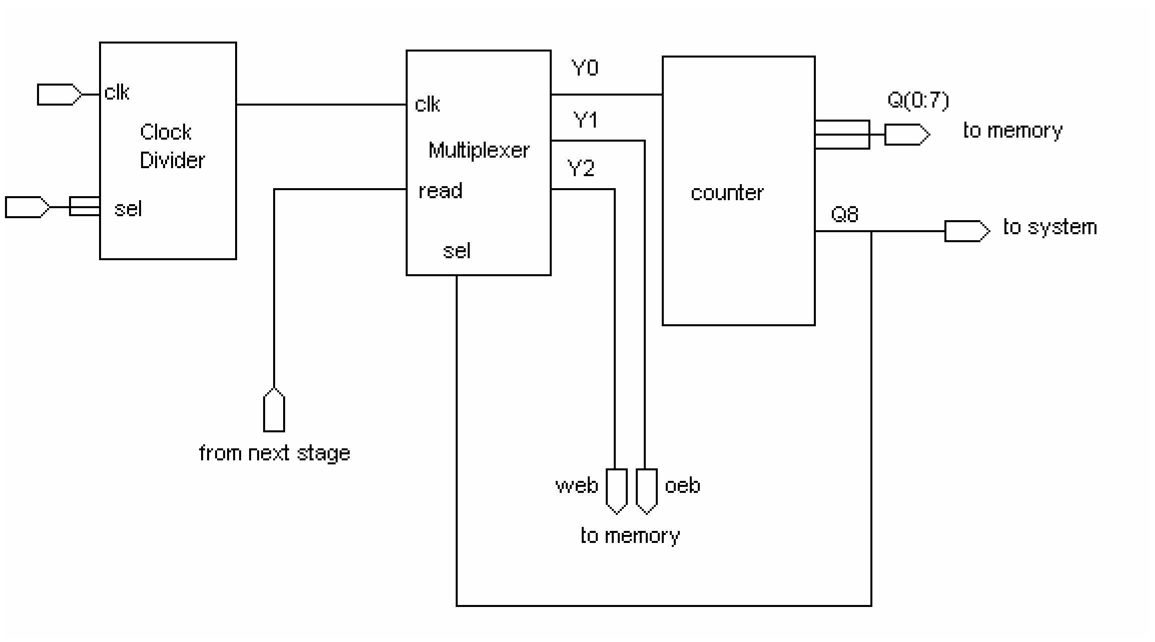
Clock Divider

The clock used during the write cycle is inputted from the clock divider. As it is obvious from its name this component divides a fast clock to obtain slower signals. This is done by using the fast signal as a driving clock to a counter that increment every clock cycle. Depending on the select a certain bit of the counter is routed to the output. The least significant bit of the counter will have the same frequency as the input clock. Higher order bits will have slower frequencies. Therefore the fast input clock can be divided by a factor of (2^N) where N is the number of the counter's bits. This slow signal also drives the ADC, there fore its known as the sampling frequency. Therefore it determines the time base of the oscilloscope.

The ADC is only used during the write signal (Q8='0') therefore Q8 is used as the chip enable of the analog to digital. Q8 is also used by rest of the system for control; therefore it will be used as an output from the oscilloscope module.



Output from clock divider



Block Diagram of oscilloscope

select	‘0’	‘1’
Y0 counter clock	Clock	Read
Y1 output enable	‘1’	‘0’
Y2 write enable	Clock	‘1’

Table of control signals during different modes of operation

VGA Interface

VGA Color Signals

There are three signals -- red, green, and blue -- that send color information to a VGA monitor. These three signals each drive an electron gun that emits electrons which paint one primary color at a point on the monitor screen.
Analog levels between 0 (completely dark) and 0.7 V (maximum brightness) on these control lines tell the monitor what intensities of these three primary colors to combine to make the color of a dot (or *pixel*) on the monitor's screen.
Each analog color input can be set to one of four levels by two digital outputs using a simple two-bit digital-to analog converter (see Figure 1). The four possible levels on each analog input are combined by the monitor to create a pixel with one of 4×4×4 = 64 different colors. So the six digital control lines let us select from a palette of 64 colors.

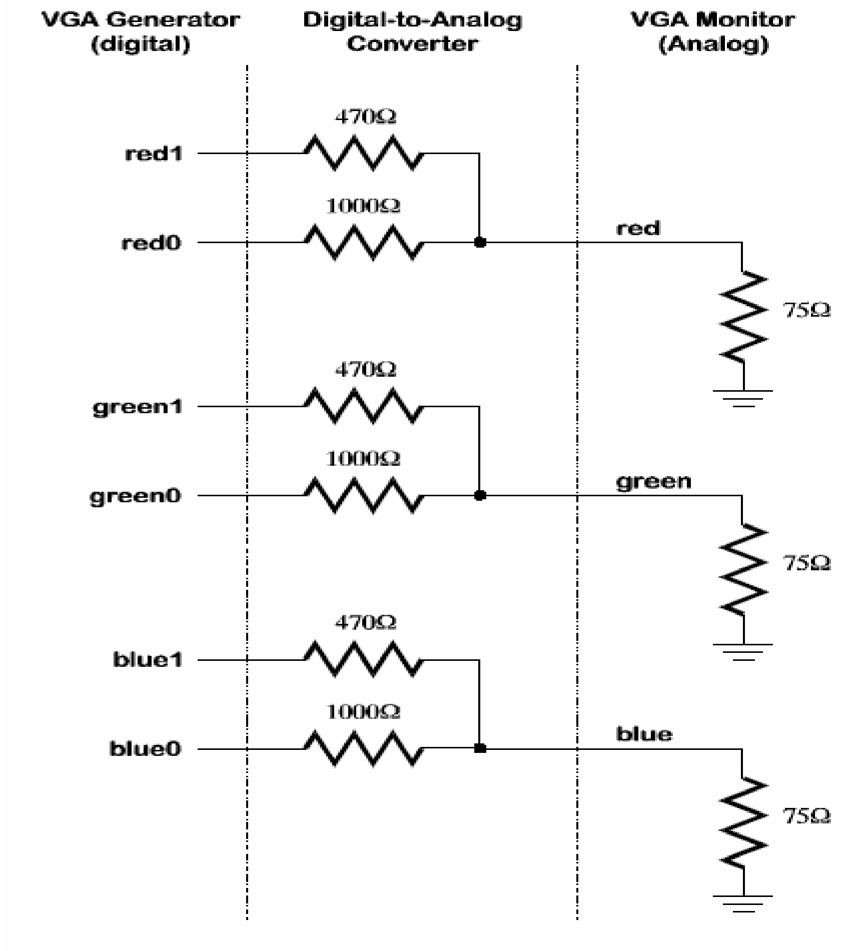


Figure 1: Digital-to-analog VGA monitor interface.

VGA Signal Timing

A single dot of color on a video monitor doesn't impart much information. A horizontal *line* of pixels carries a bit more information. But a *frame* composed of multiple lines can present an image on the monitor screen. A frame of VGA video typically has 480 lines and each line usually contains 640 pixels. In order to paint a frame, there are deflection circuits in the monitor that move the electrons emitted from the guns both left-to-right and top-to-bottom across the screen. These deflection circuits require two synchronization signals in order to start and stop the deflection circuits at the right times so that a line of pixels is painted across the monitor and the lines stack up from the top to the bottom to form an image. The timing for the VGA synchronization signals is shown in Figure 2.
Negative pulses on the *horizontal sync* signal mark the start and end of a line and ensure that the monitor displays the pixels between the left and right edges of the visible screen area. The actual pixels are sent to the monitor within a 25.17 μs window. The horizontal sync signal drops low a minimum of 0.94 μs after the last pixel and stays low for 3.77 μs. A new line of pixels can begin a minimum of 1.89 μs after the horizontal sync pulse ends. So a single line occupies 25.17 μs of a 31.77 μs interval. The other 6.6 μs of each line is the *horizontal blanking interval* during which the screen is dark.
In an analogous fashion, negative pulses on a *vertical sync* signal mark the start and end of a frame made up of video lines and ensure that the monitor displays the lines between the top and bottom edges of the visible monitor screen.
The lines are sent to the monitor within a 15.25 ms window. The vertical sync signal drops low a minimum of 0.45 ms after the last line and stays low for 64 μs. The first line of the next frame can begin a minimum of 1.02 ms after the vertical sync pulse ends. So a single frame occupies 15.25 ms of a 16.784 ms interval. The other 1.534 ms of the frame interval is the *vertical blanking interval* during which the screen is dark.

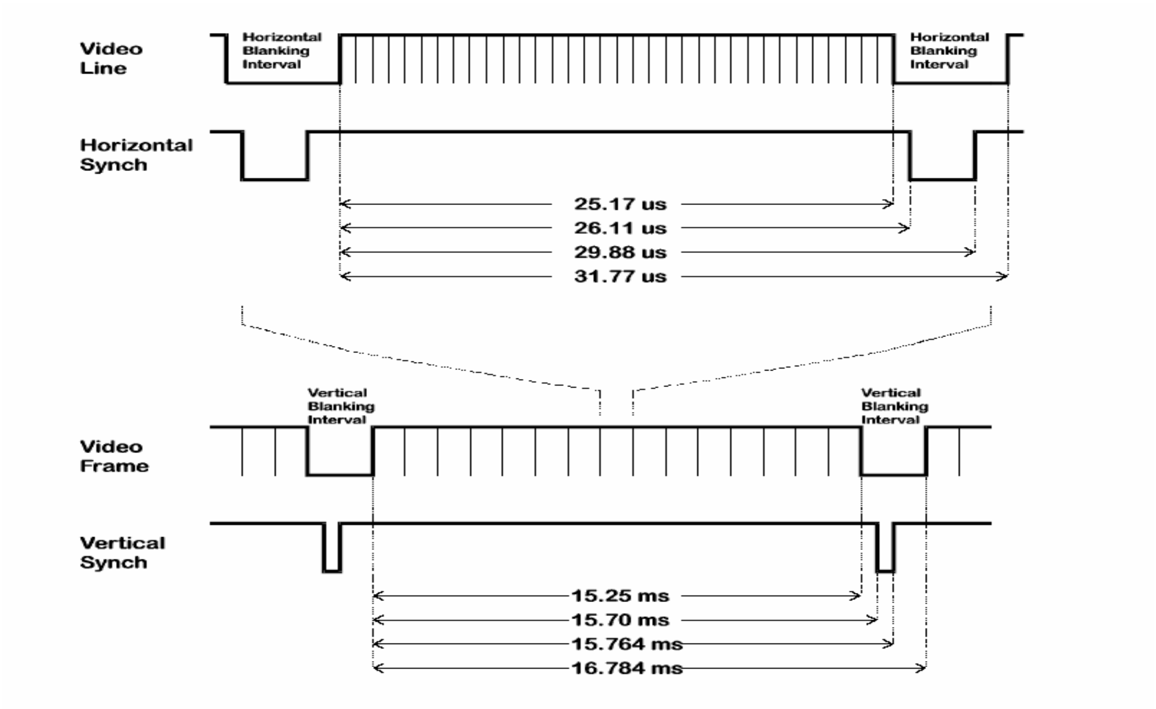


Figure 2: VGA signal timing.

VGA Signal Generator Algorithm

Now we have to figure out a process that will send pixels to the monitor with the correct timing and framing. We can store a picture in RAM. Then we can retrieve the data from the RAM, format it into lines of pixels, and send the lines to the monitor with the appropriate pulses on the horizontal and vertical sync pulses.
The pseudo code for a single frame of this process is shown in Listing 1. The pseudo code has two outer loops: one which displays the *L* lines of visible pixels, and another which inserts the *V* blank lines and the vertical sync pulse.
Within the first loop, there are two more loops: one which sends the *P* pixels of each video line to the monitor, and another which inserts the *H* blank pixels and the horizontal sync pulse.
Within the pixel display loop, there are statements to get the next byte from the RAM. Each byte contains four two-bit pixels. A small loop iteratively extracts each pixel to be displayed from the lower two bits of the byte. Then the byte is shifted by two bits so the next pixel will be in the right position during the next iteration of the loop. Since it has only two bits, each pixel can store one of four colors. The mapping from the two-bit pixel value to the actual values required by the monitor electronics is done by the
COLOR_MAP () routine.

```
/* send L lines of video to the monitor */
for line_cnt=1 to L
/* send P pixels for each line */
for pixel_cnt=1 to P
/* get pixel data from the RAM */
data = RAM (address)
address = address + 1
/* RAM data byte contains 4 pixels */
for d=1 to 4
/* mask off pixel in the lower two bits */
pixel = data & 00000011
/* shift next pixel into lower two bits */
data = data>>2
/* get the color for the two-bit pixel */
color = COLOR_MAP (pixel)
send color to monitor
d = d + 1
/* increment by four pixels */
pixel_cnt = pixel_cnt + 4
/* blank the monitor for H pixels */
for horiz_blank_cnt=1 to H
color = BLANK
send color to monitor
/* pulse the horizontal sync at the right time */
if horiz_blank_cnt>HB0 and horiz_blank_cnt<HB1
hsync = 0
else
hsync = 1
horiz_blank_cnt = horiz_blank_cnt + 1
line_cnt = line_cnt + 1
/* blank the monitor for V lines and insert vertical sync */
for vert_blank_cnt=1 to V
color = BLANK
send color to monitor
/* pulse the vertical sync at the right time */
if vert_blank_cnt>VB0 and vert_blank_cnt<VB1
vsync = 0
else
vsync = 1
vert_blank_cnt = vert_blank_cnt + 1
/* go back to start of picture in RAM */
address = 0
```

VGA signal generation pseudo code.

Figure 3 shows how to pipeline certain operations to account for delays in accessing data from the RAM. The pipeline has three stages:
Stage 1: The circuit uses the horizontal and vertical counters to compute the address where the next pixel is found in RAM. The counters are also used to

determine the firing of the sync pulses and whether the video should be blanked. The pixel data from the RAM, blanking signal, and sync pulses are latched at the end of this stage so they can be used in the next stage.

Stage 2: The circuit uses the pixel data and the blanking signal to determine the binary color outputs. These outputs are latched at the end of this stage.

Stage 3: The binary color outputs are applied to the DAC which sets the intensity levels for the monitor's color guns. The actual pixel is painted on the screen during this stage

.

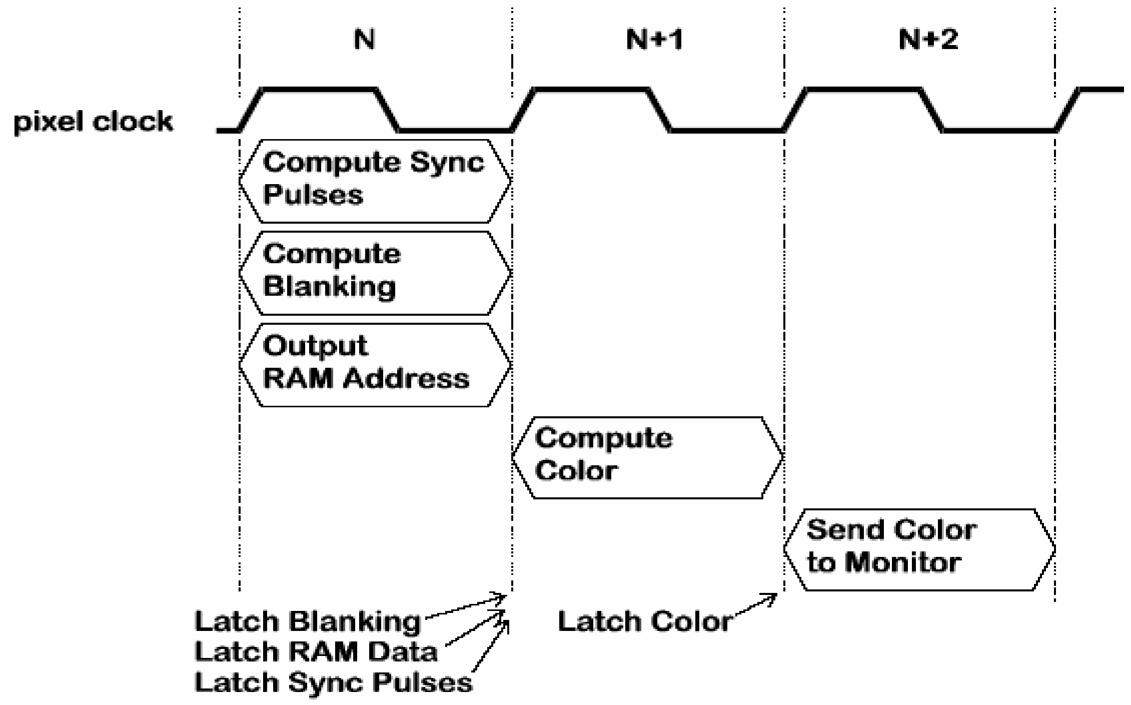


Figure 3: Pipelining of VGA signal generation tasks.

Converter/Pixel generator

The need of the Converter

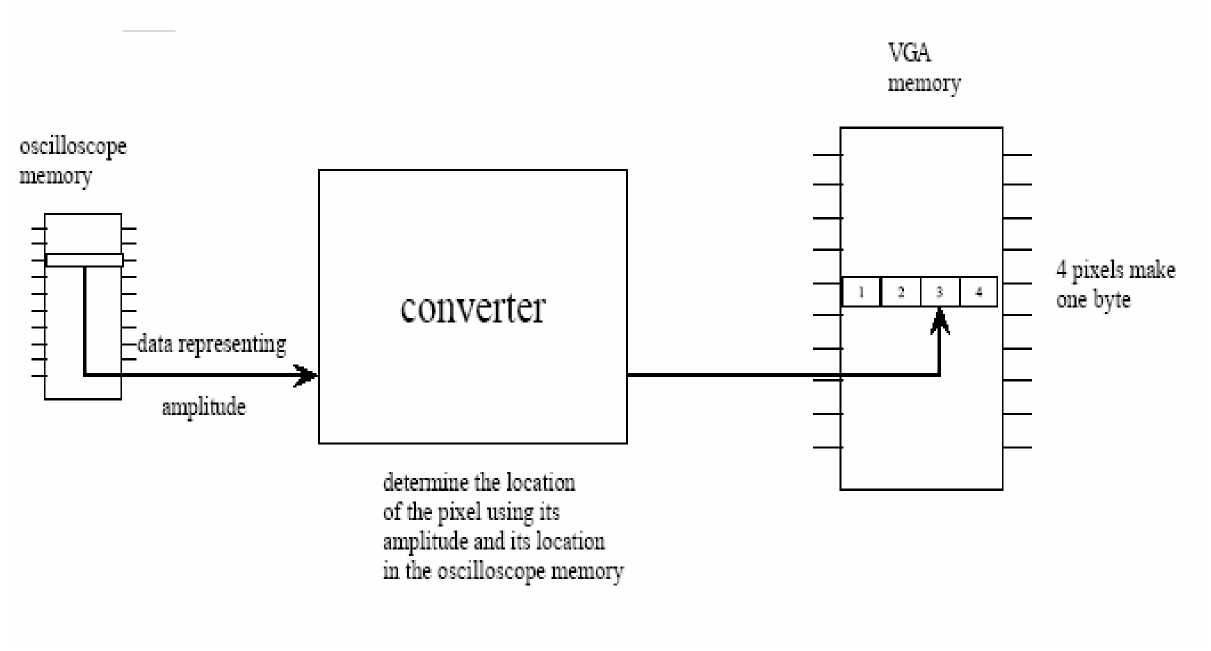
The Converter is considered the backbone of the system; it represents the link between the Oscilloscope and the VGA driver parts.
We need to display the data in the oscilloscope memory on the screen but the data stored in it represent only the amplitude of the signal at a specific time, while the VGA memory should contain a static frame or an image to be displayed on screen; and hence we started to think of the converter or the "Frame Generator".

Basic Idea

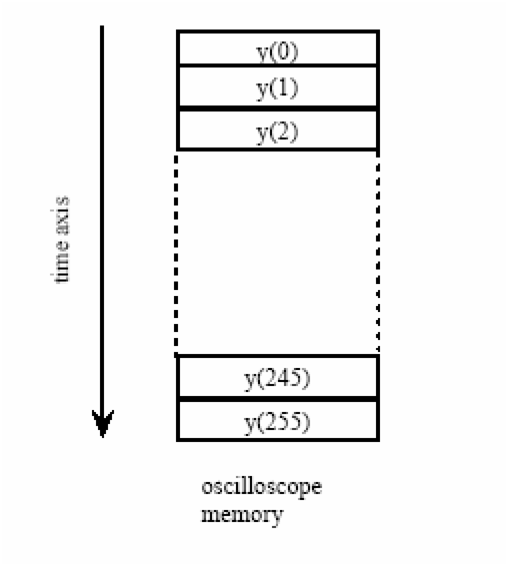
The idea of the converter depends on that the data stored in one byte of the oscilloscope memory can represent two coordinate on the horizontal and the vertical axes:

- Vertical axis; or the amplitude axis which can be extracted from the value or the data stored in the byte.

- Horizontal axis; we can call it the time axis, and we can get it knowing the location of the byte in the oscilloscope memory.

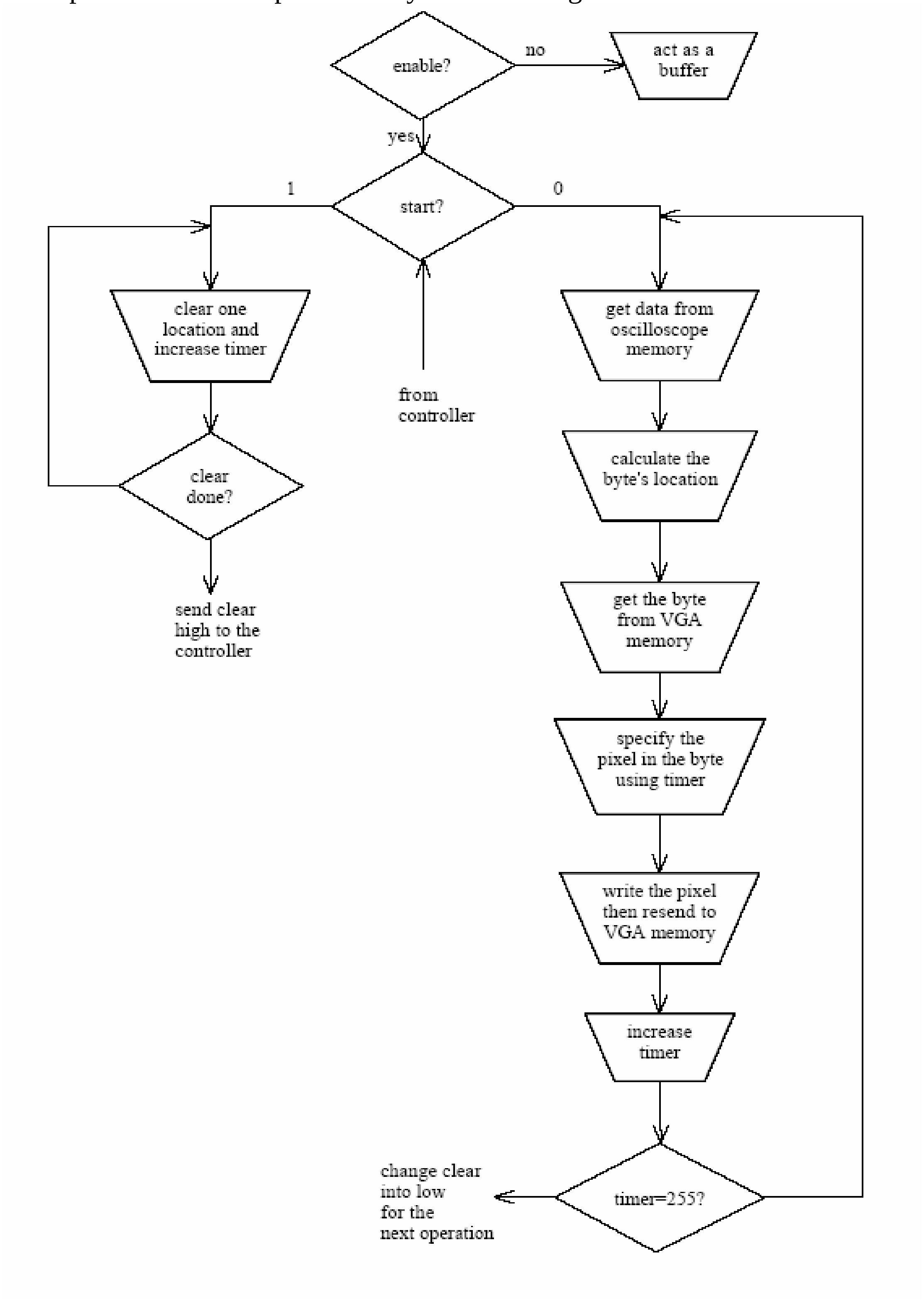

VGA memory
oscilloscope memory
converter
data representing amplitude
1
2
3
4
4 pixels make one byte
determine the location of the pixel using its amplitude and its location in the oscilloscope memory

The design was adjusted to fill only 256 location of the Oscilloscope memory in order to represent each location by one pixel on the screen, and since we have got only 256 pixels on the screen we made the previous adjustments.



The operation of the converter:

The operation can be represented by the following flowchart:


enable?
no
act as a buffer
yes
start?
1
0
clear one location and increase timer
clear done?
send clear high to the controller
from controller
get data from oscilloscope memory
calculate the byte's location
get the byte from VGA memory
specify the pixel in the byte using timer
write the pixel then resend to VGA memory
increase timer
timer=255?
change clear into low for the next operation

As seen in the previous flowchart, the converter has got two main operations which are controlled by a signal (start) from the controller:

- Clear the VGA memory, which is very important to get rid of the old frame and prepare the memory for the new one.

- The conversion process, which converts the amplitudes stored in the Oscilloscope memory into pixels in the VGA memory.

Clearing process:

It is simply a normal counter connected to the address bus of the VGA memory sending "11111111" to each byte in the memory sequentially. The data sent in this process represent the background color of the screen which will be white in this case.

Drawing both the vertical and horizontal lines is another function of this process; red pixels are placed in the memory locations corresponding to the vertical and the horizontal axis on the screen, the location of these pixels can be determined using the counter.

When this process is done, a signal (clr) is sent to the controller in order to switch operations.

Conversion process:

Which is the main aim of Converter; this process takes place in 256 clock cycle since we deal with only 256 locations in the oscilloscope memory.

Each clock cycle is divided into two parts; the first part is triggered by the rising edge of the clock while the other one is triggered by the falling edge.

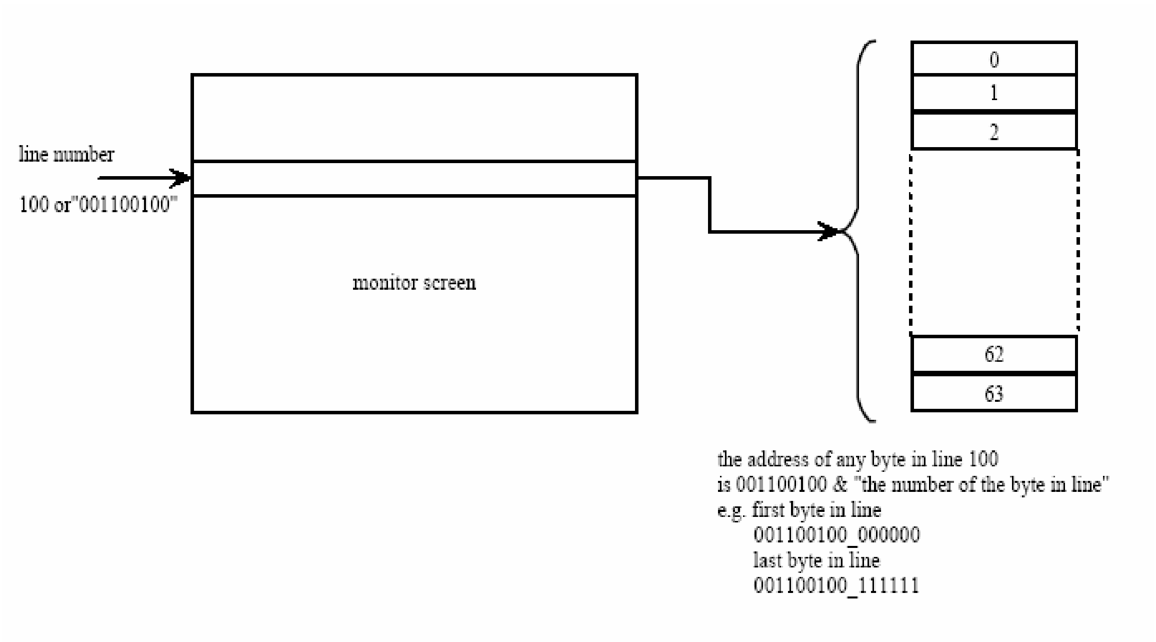
In the first half clock cycle; data is collected from the oscilloscope memory, address of the byte in VGA memory is calculated and the byte at this address is read.

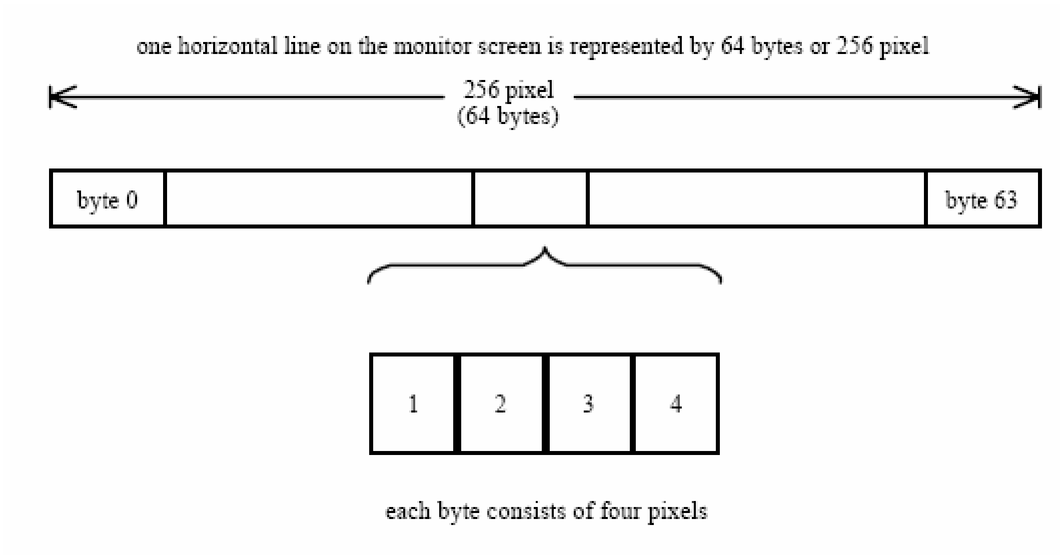
In the second half cycle; the location of the pixel in the byte collected from the VGA memory is determined then it is masked on the byte according to its location. Afterwards the byte is sent to the VGA memory and the timer increases.

We have divided each cycle into two halves in order to give the VGA memory enough time between both read and write cycles.

Detailed conversion process:

The most difficult part in this process is to find out where the pixel exists. The following diagram will show you how we figured out a solution for this problem.




one horizontal line on the monitor screen is represented by 64 bytes or 256 pixel
256 pixel
(64 bytes)
byte 0
byte 63
1
2
3
4
each byte consists of four pixels

As shown in the previous diagram, each frame on the screen consists of a number of lines, and the line consists of 64 bytes, each byte consists of 4 pixels. All that we need is to find out the address of the byte in which the pixel exists and the location of this pixel in the byte.

The data collected from the oscilloscope memory representing the amplitude as mentioned before can be used to determine the pixel's vertical coordinate, which is very logical since the higher the value gets the higher the line gets.

After determining the line, we have to determine which byte in the line containing the pixel; a counter triggered with each clock cycle does the trick.
That counter represents the time axis (i.e. each count represents one increment in the horizontal axis) or it represents the pixel number.

Since we need to get the byte location not the pixel location, and the byte consists of four pixels; then by dividing the counter by 4 we can have the byte location(the division process is done by removing the least two significant bits from the counter).

To determine the location of the pixel in the byte we use the least two significant bits from the counter.

Converter disabled

When the converter is disabled it plays a very important role in the system.
Since both the Converter and the VGA Driver are connected to the VGA memory using the same data and address bus with the same write and output enable, many problem arise.

After the converter finishes its operation and gets disabled the data keeps latched on its pins and when the VGA driver starts to read from its memory it would read the latched data instead.

Another problem which is that the address bus, output enable and the write enable of the memory have got multiple inputs (i.e. both Converter and VGA Driver can access those pins at the same time).

The data latching problem was solved by changing the data port of the converter into tri state (high impedance) and turns it to an input port while the Converter is disabled to avoid latching completely.

The address problem was solved using a simple multiplexer.

While the enable signals problem was solved by sending both signals from the Converter since both the converter and the VGA Driver never work in the same time.

Controller

What is the controller?

It’s the main module that organizes the job of all the other components of the circuit. It synchronizes the signals coming into and from the other modules to prevent overlapping and to make sure that all the data is stored in memory, converted into pixel forum and then read to be displayed on the screen for a fixed number of frames for human eye comfort (steady picture). So the controller adjusts the timing needed for all these operations by controlling signals that links all the modules.

A closer look at the circuit synchronization:

- During data conversion by the ADC and storage into the oscilloscope’s memory (Q8=’0’), the VGACORE module is reading the pre-stored data (pixels) in the VGA memory and displaying it on the screen.

Counter (ON)	ADC (ON)
CONVERTER (OFF)	DISPLAY (ON)

- When we reach the last location in the oscilloscope’s memory (full, Q8=’1’), we need to stop the ADC till we convert all the data into pixels to be displayed on the monitor. During the conversion operation the display must be off or random unwanted output will be displayed.

COUNTER (OFF)	ADC (OFF)
CONVERTER (ON)	DISPLAY (OFF)

Controlling signals description:

- **Q8:** Most significant bit of the counter, input to the controller.
- **Clr:** clear signal sent from the converter to the controller.
- **Start:** Signal output from the controller to the converter.
- **V_done:** Signal output from VGACORE module to the controller.
- **Enable:** Signal output from the controller to the converter.

- **En:** enable signal sent from controller to counter.
- **V_reset:** signal sent to VGA core to reset it.

.

Controlling signals flow:

First Q8 is initialized with a high signal, the converter is enabled and the VGA disabled (reset). This is done by sending enable=’1’ to the converter form the controller. The controller then receives a clr signal=’0’ from the converter, thus sending start=’1’ to the converter and read=’0’ to the multiplexer that routes it to the counter. This keeps the counter in a hold state. When the converter sends clr=’1’, the controller sends start=’0’, and read= clk (12MHz clock) to the counter via the multiplexer. This gets the counter out of the hold state.

When the converter finishes its job, Q8 goes low. This enables the VGA, which starts the display of the data stores in the VGA memory and the converter is disabled. Meanwhile the counter is disabled as V_done=’0’ because the VGA was reset earlier when Q8 was high, this signal is sent from the VGA core to the controller and routed to the counter via its ‘en’ signal. When 200 frames are displayed by the VGA, V_done goes high which enables the counter again, therefore starting a new write cycle. Also the VGA is still enabled continuing to display pre-stored data in VGA memory until Q8 goes high again and the cycle repeats.

System overview

In the previous sections each software block was described separately, the goal of this section is to provide a whole view of the system to the reader explaining the connections and the handshaking between the several blocks; it also provides a complete signal flow explanation from the input analog signal till the output display on the monitor

Signal flow

The analog signal is first fed to the analog to digital converter which converts its amplitudes to a number of bits which are stored in the first ram referred to as oscilloscope memory. The addresses of the oscilloscope ram are generated by the counter which is driven by the write clock which is the same clock of the ADC. After 256 counts the oscilloscope memory is now full and is ready to be read, so the multiplexer switches the input clock of the counter to the fast clock (12 MHz) provided from the controller.
After that the conversion of the amplitudes to one frame takes place through pixels generator which completes its job through two stages first clearing the second memory referred to as VGA ram and plotting the axis .the second stage is reading the oscilloscope memory data and filling the VGA ram as explained in the conversion part .after reading the 256 location of the oscilloscope memory the VGA ram is now filled with the appropriate frame, and it is now the turn of VGA driver to display the generated frame, and then the whole cycle repeats again.

Interconnection and handshaking

In the beginning the oscilloscope counter will be in the write cycle, until Q8 equals to ‘1’. Q8 is considered as the master control signal in the system. It will be passed to the controller and then from the controller to reset the VGA, routs the converter address to the output of the address switch and enable the converter. The clock fed to the counter will be generated from the controller according to the status of the clear signal sent from the converter to the controller.

At the start of the converter’s operation it will send a ‘0’clear signal to the controller indicating that the VGA RAM is still not cleared. So the controller will send a ‘1’ start signal to the converter to initiate the clearing operation and plotting the axis. During this stage the oscilloscope’s counter is in a hold state controlled by the ‘0’ read signal fed from the controller. When the clearing process ends a ‘1’ clear signal is sent to controller indicating end of clearing. At this moment a

'0' start signal is sent from the controller to the converter to start interfacing with the oscilloscope memory and placing the pixels in the VGA RAM. The oscilloscope counter is now fed with the read clock signal (12 MHz). After the 256 locations of the oscilloscope memory are accessed, Q8 falls to '0' indicating the end of conversion, the start of display and routing the VGA address to the output of the switch.

The VGA core will start displaying the generated frame 200 times to be acceptable by the viewer's eye. During this time the counter is disabled as well as the converter. After the display of 200 frame VGA done; which is fed to the controller; signal goes high, which passes it to the counter. This enables the counter and the beginning of a new write cycle. During the write cycle the VGA core is still activated, and VGA done is still high. When the write cycle ends, the reset signal is fed to the VGA core, therefore the VGA done signal goes low, and conversion starts again causing the display to blank till Q8 reaches '0'. And the cycle repeats.

Timing

The time taken by the whole cycle can be broken down into time taken by the following procedures:

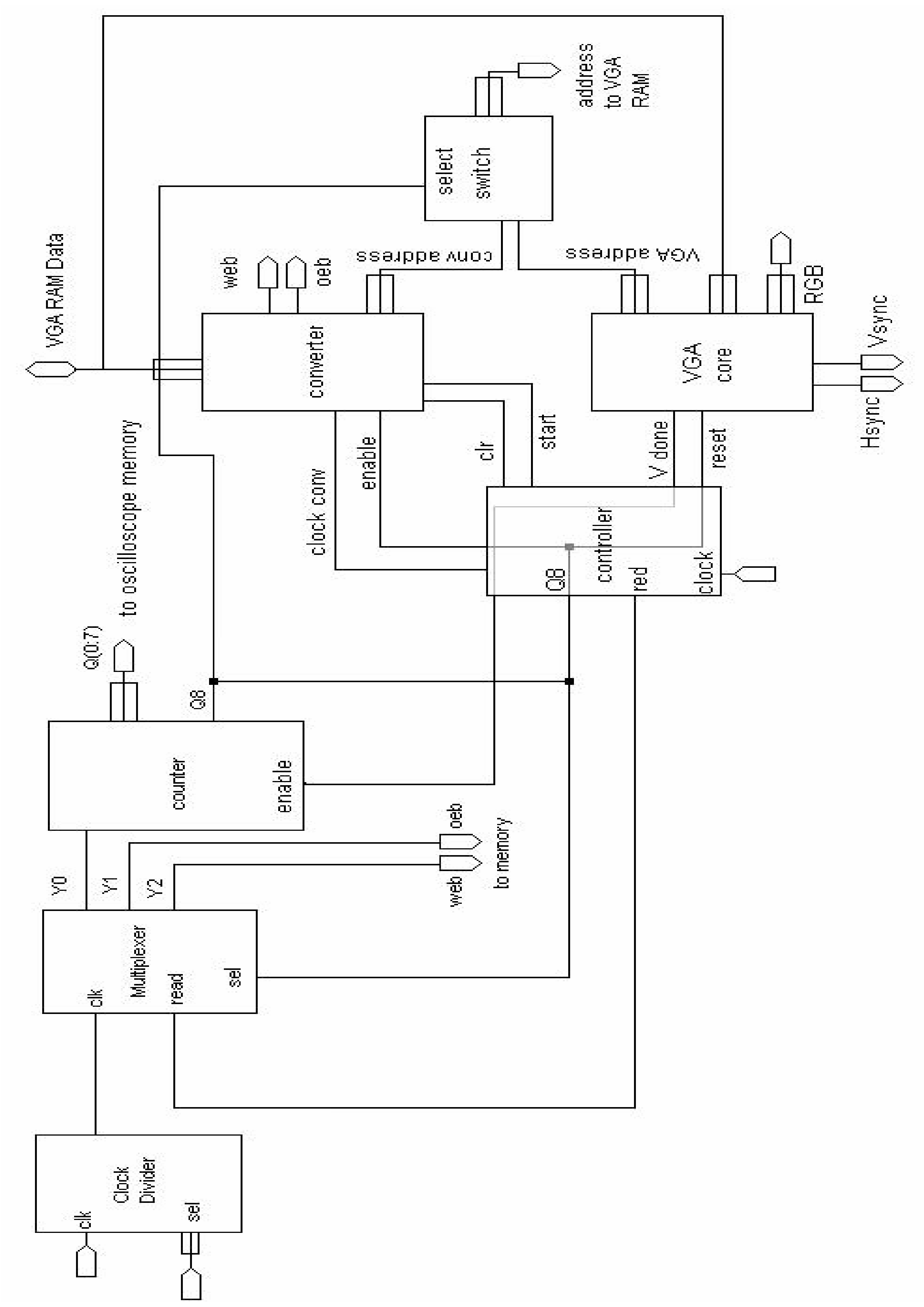
- Writing cycle in the oscilloscope memory
- Conversion
    - Clearing of VGA RAM
    - Writing in VGA RAM
- Display of 200 frames

First the write cycle takes 256 clock cycle of the analog to digital converter clock and since this clock is variable depending on the sampling frequency the length of this cycle varies.

Second: clearing the memory of the VGA memory takes 32768 clock cycle of the converter clock which is 12MHz, the time of this process is 2.73 msec.
The writing in VGA memory process takes 256 clock cycle of the 12 MHz clock and since this operation takes 21.3 µsec.

Third: displaying of 200 frames takes the time of displaying one frame multiplied by 200 which is 3.344 sec, knowing that one frame takes 16.784 msec.

The total display time of VGA is equal to the time of displaying 200 frames plus the time of writing 256 locations in the Oscilloscope memory.


VGA RAM Data
to oscilloscope memory
web
oeb
converter
conv address
select
switch
address
to VGA
RAM
VGA address
RGB
VGA
core
Vsync
Hsync
clock conv
enable
clr
start
V done
reset
Q8
controller
red
clock
Q(0:7)
Q8
counter
enable
oeb
web
to memory
Y0
Y1
Y2
clk
Multiplexer
read
sel
clk
Clock
Divider
sel

Appendix I

VHDL Codes

Oscilloscope

Multiplexer

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity mux8_4 is
   Port ( clock : in std_logic;
          read : in std_logic;
          y0,y1,y3 : out std_logic;
          sel : in std_logic
         );
end mux8_4;

architecture Behavioral of mux8_4 is

begin
          process(sel,clock,read)
          begin
if (sel='1')then -- read cycle

y0<=read;     -- fast clock for read
y1<='0';     --output enable activated
y3<='1';    -- write enable deactivated

else           -- write cycle

y0<=clock;    -- clock for write  controlled by divider
y1<='1';         -- output enable deactivated
y3<=clock;    --   write enable activated
end if;
          end process;
end Behavioral;
```

Counter

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity conter is
   Port ( clk : in std_logic;
            en: in std_logic;
          cnt : out std_logic;
          address1:out std_logic_vector (7 downto 0)
         );
end conter;

architecture Behavioral of conter is
signal cont : std_logic_vector(8 downto 0):=(others=>'1');
begin
process(clk)
begin
if(en='1' or cont(8)='1')then  --start counting after required number of frames is done
                               -- or write cycle ended
if(clk'event and clk='1')then

cont<=cont+1;
end if;
else
cont<=(others=>'0');
end if;
end process;
cnt<=cont(8);                   --control for read and write cycles
address1<=cont(7 downto 0); --address for Oscilloscope   RAM



end Behavioral;
```

Divider

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity last is
   Port ( sel : in std_logic_vector(3 downto 0);
          clk : in std_logic;
          clok : out std_logic
         );
end last;

architecture Behavioral of last is
signal temp : std_logic_vector(13 downto 0) ;
signal clock : std_logic;
begin

A:process(clk)

begin
if(clk'event and clk='1') then
        temp <= temp + 1;
end if;
end process;

B: process(temp)
        begin

case sel is
when "0000" => clock <= temp(2);--divide clock by 2
when "0001" => clock <= temp(3); --divide clock by 4
when "0010" => clock <= temp(4); --divide clock by 8
when "0011" => clock <= temp(5); --divide clock by 16
when "0100" => clock <= temp(6); --divide clock by 32
when "0101" => clock <= temp(7); --divide clock by 64
when "0110" => clock <= temp(8); --divide clock by 128
when "0111" => clock <= temp(9); --divide clock by 256
when "1000" => clock <= temp(10); --divide clock by 512
when "1001" => clock <= temp(11); --divide clock by 1024
when "1010" => clock <= temp(12); --divide clock by 2048
when "1011" => clock <= temp(13); --divide clock by 4096
when others  => clock <= clk;
```

```
end case;
clok <=clock;
end process;

end Behavioral;
```

VGA Driver

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
entity vgacore is
port
(
reset: in std_logic; -- reset
clock: in std_logic; -- VGA dot clock
hsyncb: buffer std_logic; -- horizontal (line) sync
vsyncb: out std_logic; -- vertical (frame) sync
rgb: out std_logic_vector(5 downto 0); -- red,green,blue colors
addr: out std_logic_vector(14 downto 0); -- address to video RAM
data: in std_logic_vector(7 downto 0); -- data from video RAM
csb: out std_logic; -- video RAM chip enable
oeb: out std_logic; -- video RAM output enable
web: out std_logic; -- video RAM write enable
vdone : out std_logic
);
end vgacore;
architecture vgacore_arch of vgacore is
signal hcnt: std_logic_vector(8 downto 0); -- horizontal pixel counter
signal vcnt: std_logic_vector(9 downto 0); -- vertical line counter
signal pixrg: std_logic_vector(7 downto 0); -- byte register for 4 pixels
signal blank: std_logic; -- video blanking signal
signal pblank: std_logic; -- pipelined video blanking signal
----------------------------
signal frm: std_logic_vector(11 downto 0);
----------------------------------
begin
A: process(clock,reset)
begin
-- reset asynchronously clears pixel counter
if reset='1' then
hcnt <= "000000000";
-- horiz. pixel counter increments on rising edge of dot clock
elsif (clock'event and clock='1') then
-- horiz. pixel counter rolls-over after 381 pixels
if hcnt<380 then
hcnt <= hcnt + 1;
else
hcnt <= "000000000";
end if;
end if;
end process;
B: process(hsyncb,reset)
begin
-- reset asynchronously clears line counter
if reset='1' then
vcnt <= "0000000000";
---------------
frm<="000000000000";
vdone<='0';
----------------
-- vert. line counter increments after every horiz. line
elsif (hsyncb'event and hsyncb='1') then
-- vert. line counter rolls-over after 528 lines
if vcnt<527 then
vcnt <= vcnt + 1;
else
vcnt <= "0000000000";
end if;
--------------
if (vcnt=526)then
frm<=frm+1;
if(frm>200)then
vdone<='1';  -- displaying 200 frames for the user acceptance
end if;
----------------
end if;
end if;
end process;
C: process(clock,reset)
begin
-- reset asynchronously sets horizontal sync to inactive
if reset='1' then
hsyncb <= '1';
-- horizontal sync is recomputed on the rising edge of every dot clock
elsif (clock'event and clock='1') then
-- horiz. sync low in this interval to signal start of new line
if (hcnt>=291 and hcnt<337) then
hsyncb <= '0';
else
hsyncb <= '1';
end if;
end if;
end process;
D: process(hsyncb,reset)
begin
-- reset asynchronously sets vertical sync to inactive
```

```
if reset='1' then
vsyncb <= '1';
-- vertical sync is recomputed at the end of every line of pixels
elsif (hsyncb'event and hsyncb='1') then
-- vert. sync low in this interval to signal start of a new frame
if (vcnt>=490 and vcnt<492) then
vsyncb <= '0';
else
vsyncb <= '1';
end if;
end if;
end process;
-- blank video outside of visible region: (0,0) -> (255,479)
E: blank <= '1' when (hcnt>=256 or vcnt>=480) else '0';
-- store the blanking signal for use in the next pipeline stage
F: process(clock,reset)
begin
if reset='1' then
pblank <= '0';
elsif (clock'event and clock='1') then
pblank <= blank;
end if;
end process;
-- video RAM control signals
G:
csb <= '0'; -- enable the RAM
web <= '1'; -- disable writing to the RAM
oeb <= blank; -- enable the RAM outputs when video is not blanked
-- The video RAM address is built from the lower 9 bits of the vertical
-- line counter and bits 7-2 of the horizontal pixel counter.
-- Each byte of the RAM contains four 2-bit pixels. As an example,
-- the byte at address ^h1234=^b0001,0010,0011,0100 contains the pixels
-- at (row,col) of (^h048,^hD0),(^h048,^hD1),(^h048,^hD2),(^h048,^hD3).
H: addr <= vcnt(8 downto 0) & hcnt(7 downto 2);
I: process(clock,reset)
begin
-- clear the pixel register on reset
if reset='1' then
pixrg <= "00000000";
-- pixel clock controls changes in pixel register
elsif (clock'event and clock='1') then
-- the pixel register is loaded with the contents of the video
-- RAM location when the lower two bits of the horiz. counter
-- are both zero. The active pixel is in the lower two bits
-- of the pixel register. For the next 3 clocks, the pixel
-- register is right-shifted by two bits to bring the other
-- pixels in the register into the active position.
if hcnt(1 downto 0)="00" then
pixrg <= data; -- load 4 pixels from RAM
else
pixrg <= "00" & pixrg(7 downto 2); -- right-shift pixel register
end if;
end if;
end process;
-- the color mapper translates each 2-bit pixel into a 6-bit
-- color value. When the video signal is blanked, the color
-- is forced to zero (black).
J: process(clock,reset)
begin
-- blank the video on reset
if reset='1' then
rgb <= "000000";
-- update the color outputs on every dot clock
elsif (clock'event and clock='1') then
-- map the pixel to a color if the video is not blanked
if pblank='0' then
case pixrg(1 downto 0) is
when "00" => rgb <= "110000"; -- red
when "01" => rgb <= "001100"; -- green
when "10" => rgb <= "000011"; -- blue
when others => rgb <= "111111"; -- white
end case;
--otherwise, output black if the video is blanked
else
rgb <= "000000"; -- black
end if;
end if;
end process;
end vgacore_arch;
```

Pixel Generator

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity convert is
    Port (
            enable : in std_logic;
            clk : in std_logic;
            data1 : in std_logic_vector(7 downto 0);
            data2 : inout std_logic_vector(7 downto 0);
            address : out std_logic_vector(14 downto 0);
             oeb: out std_logic;
             web: out std_logic;
             clr: out std_logic;
             start : in std_logic
            );
end convert;

architecture behavioral of convert is

signal temp: std_logic_vector(15 downto 0);
signal dada,dada2 : std_logic_vector(7 downto 0);
signal timer : std_logic_vector(8 downto 0);

begin

process(start,clk)

begin

if(enable='1')then   -- enable converter

--clear memory--
  if(start='1' )then

if(clk'event and clk='1')then
web <= '0';
clr <='0';
oeb <= '1';
timer <= "000000000";
address <=temp(15 downto 1);
temp <= temp + 1;
```

```
-----------------------------------------------------------------------------------------------------
-- plot of horizontal axis
if (temp<32768 and temp>32640 ) then
data2<="00000000";
else
data2 <= "11111111";
end if;
--plot the vertical axis
if (temp(5 downto 0)=32)then
data2<="00111111";
if(temp<32768 and temp>32640)then
data2<="00000000";
end if;

end if;

-------------------------------------------------------------------------------------------------------
if(temp(15 downto 1)=32767)then   -- clear done
clr<='1';
temp<="0000000000000000";
end if;
end if;
------------------------------------------------------------------------------------------------------

elsif(clk'event and clk='1')then

dada <= data1;--get data from data memory

address <= ("011111101"-dada(5 downto 0)) & timer(7 downto 2);   --calculate pixel -----
---address --
web <= '1';
oeb <= '0';
dada2 <= data2;

web <= '0';
oeb <= '1';


--locating the pixel in the specified Byte
elsif(clk'event and clk='0')then

if(timer(1 downto 0) = "00")then
data2<=dada2(7 downto 2) & "10";

elsif(timer(1 downto 0) = "01")then
```

```
data2<=dada2( 7 downto 4) & "10" & dada2(1 downto 0);
elsif(timer(1 downto 0) = "10")then
data2<=dada2(7 downto 6) & "10"& dada2(3 downto 0);

elsif(timer(1 downto 0) = "11")then
data2<= "10" & dada2(5 downto 0) ;

end if;
timer <= timer + 1;
if(timer = 255)then       ---check timer to end the program
clr  <= '0';
end if;
end if;
end if;

if(enable='0')then  -- converter disabled
web<='1';  -- write enable deactivated for  VGA memory
oeb<='0';   -- read enable activated for  VGA memory
data2<="ZZZZZZZZ"; -- data Buffering (high impedance )
dada2<=data2; --turn the data2 port into an input port to avoid latching completely
end if;
end process;

end behavioral;
```

Controller

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity control is
   Port ( Q_9 : in std_logic;
          V_done : in std_logic;
          clr : in std_logic;
          clk : in std_logic;
          clk_conv : out std_logic;
          start : out std_logic;
          V_reset : out std_logic;
           red : out std_logic;
           en: out std_logic;
           enable: out std_logic
           );
end control;

architecture Behavioral of control is

begin

VGA: V_reset <= Q_9; -- VGA should be off while writing in the VGA RAM

osc:  en<=v_done; --control the oscilloscope counter to allow the viewer to  observe the  -
--frames


conv:

clk_conv<=clk;  --sending the clock to the conveter
enable<=Q_9;  -- enable converter

process(clr,Q_9)

begin

if(Q_9='1') then

if(clr='0')then
start <= '1'; --clear the VGA memory
red<='0';--stop the oscilloscope counter
```

```
else
start <= '0';--begin the conversion process
red<=clk;--start the oscilloscope counter(read signal)to work with the same clock of the -
--converter
end if;
end if;

end process;

end Behavioral;
```

Switch

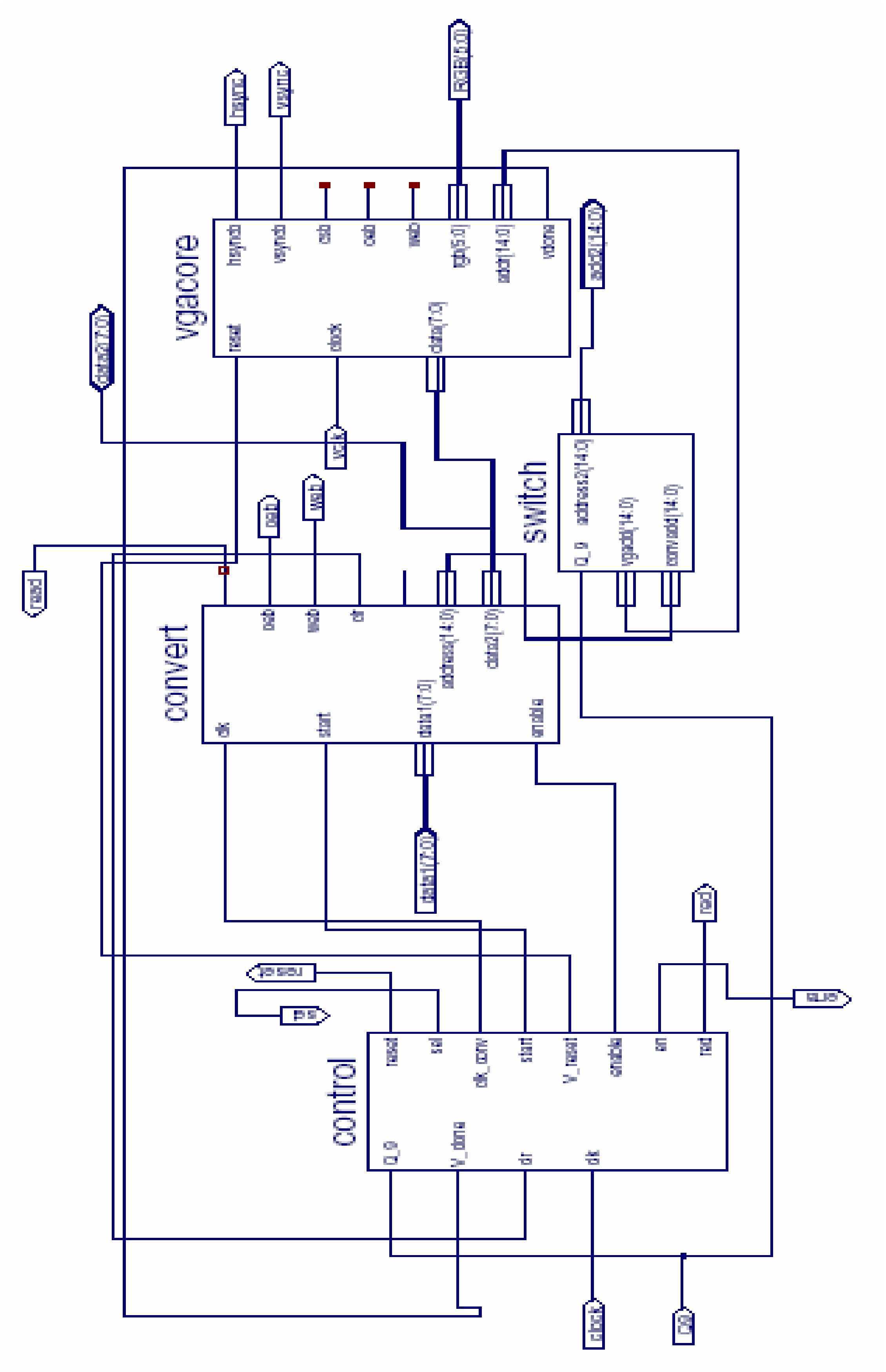
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity switch is
   Port ( vgadd : in std_logic_vector(14 downto 0);
          convadd : in std_logic_vector(14 downto 0);
          address2 : out std_logic_vector(14 downto 0);
          Q_9 : in std_logic);
end switch;

architecture Behavioral of switch is

begin
process(Q_9)
begin
if(Q_9='1')then
          ddress2 <= convadd;
else
          ddress2 <= vgadd;
end if;
end process;


end Behavioral;
```

VGA Control Convert

```
-- Vhdl model created from schematic samkmorsy.sch - Tue Jun 22 22:28:12 2004

LIBRARY ieee;
LIBRARY UNISIM;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;
USE UNISIM.Vcomponents.ALL;

ENTITY samkmorsy IS
   PORT ( Q9          :     IN      STD_LOGIC;
          clock       :     IN      STD_LOGIC;
          data1       :     IN      STD_LOGIC_VECTOR (7 DOWNTO 0);
          vclk        :     IN      STD_LOGIC;
          RGB         :     OUT     STD_LOGIC_VECTOR (5 DOWNTO 0);
          add2        :     OUT     STD_LOGIC_VECTOR (14 DOWNTO 0);
          ena         :     OUT     STD_LOGIC;
          hsync       :     OUT     STD_LOGIC;
          oeb         :     OUT     STD_LOGIC;
          read        :     OUT     STD_LOGIC;
          red         :     OUT     STD_LOGIC;
          reset       :     OUT     STD_LOGIC;
          vsync       :     OUT     STD_LOGIC;
          web         :     OUT     STD_LOGIC;
          data2       :     INOUT           STD_LOGIC_VECTOR (7 DOWNTO 0));

end samkmorsy;

ARCHITECTURE SCHEMATIC OF samkmorsy IS
   SIGNAL VGA_add   :     STD_LOGIC_VECTOR (14 DOWNTO 0);
   SIGNAL XLXN_15   :     STD_LOGIC;
   SIGNAL XLXN_16   :     STD_LOGIC;
   SIGNAL XLXN_17   :     STD_LOGIC;
   SIGNAL XLXN_23   :     STD_LOGIC;
   SIGNAL XLXN_24   :     STD_LOGIC;
   SIGNAL converter_add   :     STD_LOGIC_VECTOR (14 DOWNTO 0);
   SIGNAL vdone     :     STD_LOGIC;

   ATTRIBUTE fpga_dont_touch : STRING ;

   COMPONENT control
      PORT ( Q_9      :     IN      STD_LOGIC;
             V_done   :     IN      STD_LOGIC;
```

```
        clr         :   IN    STD_LOGIC;
        clk         :   IN    STD_LOGIC;
        reset       :   OUT   STD_LOGIC;
        clk_conv    :   OUT   STD_LOGIC;
        start       :   OUT   STD_LOGIC;
        V_reset     :   OUT   STD_LOGIC;
        en          :   OUT   STD_LOGIC;
        enable      :   OUT   STD_LOGIC;
        red         :   OUT   STD_LOGIC);
END COMPONENT;

COMPONENT convert
  PORT ( enable     :   IN    STD_LOGIC;
        clk         :   IN    STD_LOGIC;
        start       :   IN    STD_LOGIC;
        data1       :   IN    STD_LOGIC_VECTOR (7 DOWNTO 0);
        data2       :   INOUT       STD_LOGIC_VECTOR (7 DOWNTO 0);
        read        :   OUT   STD_LOGIC;
        oeb         :   OUT   STD_LOGIC;
        web         :   OUT   STD_LOGIC;
        clr         :   OUT   STD_LOGIC;
        done        :   OUT   STD_LOGIC;
        address     :   OUT   STD_LOGIC_VECTOR (14 DOWNTO 0));
END COMPONENT;

COMPONENT switch
  PORT ( Q_9        :   IN    STD_LOGIC;
        vgadd       :   IN    STD_LOGIC_VECTOR (14 DOWNTO 0);
        convadd     :   IN    STD_LOGIC_VECTOR (14 DOWNTO 0);
        address2    :   OUT   STD_LOGIC_VECTOR (14 DOWNTO 0));
END COMPONENT;

COMPONENT vgacore
  PORT ( reset      :   IN    STD_LOGIC;
        clock       :   IN    STD_LOGIC;
        data        :   IN    STD_LOGIC_VECTOR (7 DOWNTO 0);
        hsyncb      :   OUT   STD_LOGIC;
        vsyncb      :   OUT   STD_LOGIC;
        csb         :   OUT   STD_LOGIC;
        oeb         :   OUT   STD_LOGIC;
        web         :   OUT   STD_LOGIC;
        rgb         :   OUT   STD_LOGIC_VECTOR (5 DOWNTO 0);
        addr        :   OUT   STD_LOGIC_VECTOR (14 DOWNTO 0);
        vdone       :   OUT   STD_LOGIC);
END COMPONENT;
```

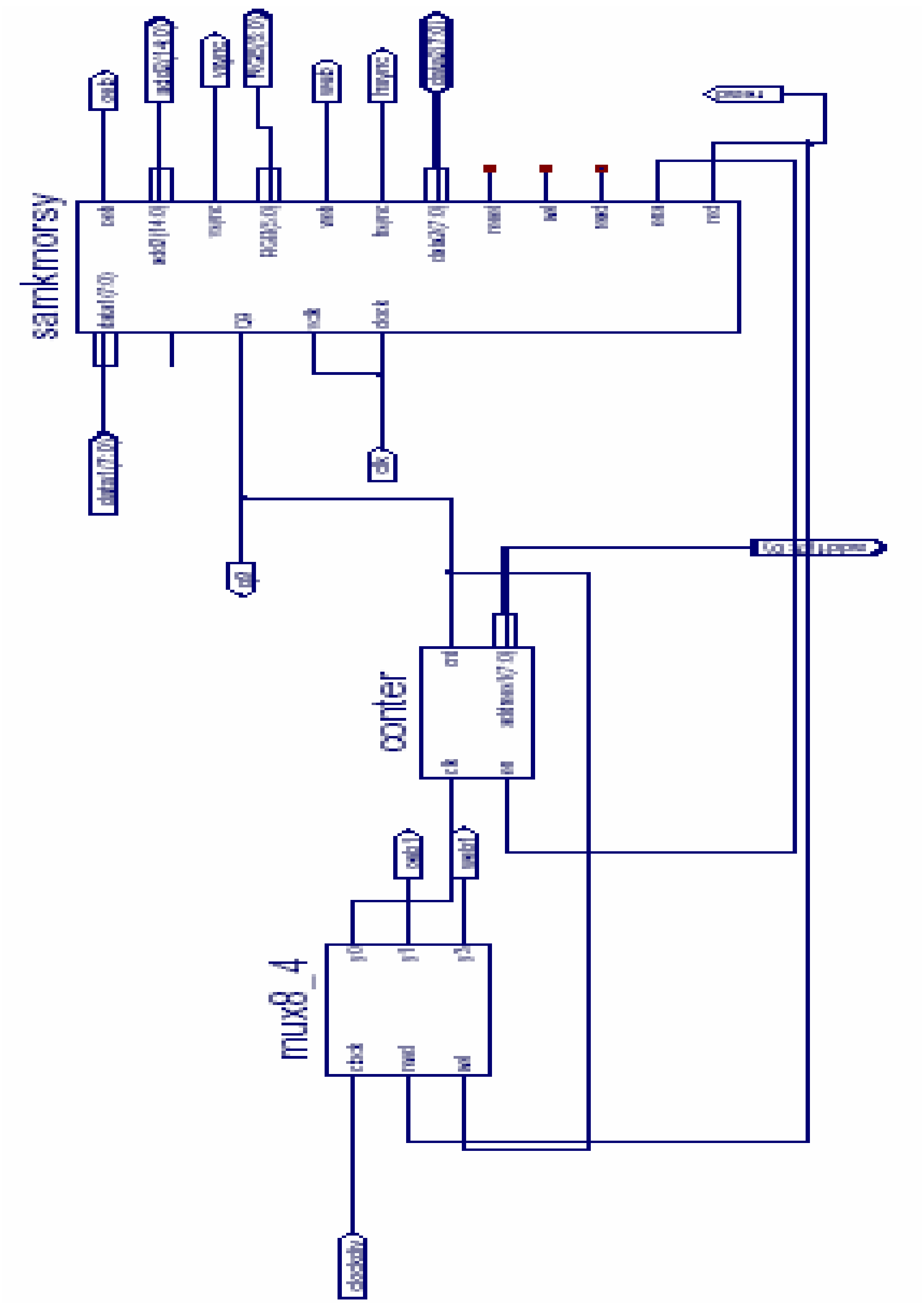
```
BEGIN

  XLXI_4 : control
    PORT MAP (Q_9=>Q9, V_done=>vdone, clr=>XLXN_24, clk=>clock, reset=>reset,
    sel=>sel, clk_conv=>XLXN_17, start=>XLXN_16, V_reset=>XLXN_23, en=>ena,
    enable=>XLXN_15, red=>red);

  XLXI_2 : convert
    PORT MAP (enable=>XLXN_15, clk=>XLXN_17, start=>XLXN_16,
    data1(7)=>data1(7), data1(6)=>data1(6), data1(5)=>data1(5),
    data1(4)=>data1(4), data1(3)=>data1(3), data1(2)=>data1(2),
    data1(1)=>data1(1), data1(0)=>data1(0), data2(7)=>data2(7),
    data2(6)=>data2(6), data2(5)=>data2(5), data2(4)=>data2(4),
    data2(3)=>data2(3), data2(2)=>data2(2), data2(1)=>data2(1),
    data2(0)=>data2(0), read=>read, oeb=>oeb, web=>web, clr=>XLXN_24,
    done=>open, address(14)=>converter_add(14),
    address(13)=>converter_add(13), address(12)=>converter_add(12),
    address(11)=>converter_add(11), address(10)=>converter_add(10),
    address(9)=>converter_add(9), address(8)=>converter_add(8),
    address(7)=>converter_add(7), address(6)=>converter_add(6),
    address(5)=>converter_add(5), address(4)=>converter_add(4),
    address(3)=>converter_add(3), address(2)=>converter_add(2),
    address(1)=>converter_add(1), address(0)=>converter_add(0));

  XLXI_1 : switch
    PORT MAP (Q_9=>Q9, vgadd(14)=>VGA_add(14), vgadd(13)=>VGA_add(13),
    vgadd(12)=>VGA_add(12), vgadd(11)=>VGA_add(11), vgadd(10)=>VGA_add(10),
    vgadd(9)=>VGA_add(9), vgadd(8)=>VGA_add(8), vgadd(7)=>VGA_add(7),
    vgadd(6)=>VGA_add(6), vgadd(5)=>VGA_add(5), vgadd(4)=>VGA_add(4),
    vgadd(3)=>VGA_add(3), vgadd(2)=>VGA_add(2), vgadd(1)=>VGA_add(1),
    vgadd(0)=>VGA_add(0), convadd(14)=>converter_add(14),
    convadd(13)=>converter_add(13), convadd(12)=>converter_add(12),
    convadd(11)=>converter_add(11), convadd(10)=>converter_add(10),
    convadd(9)=>converter_add(9), convadd(8)=>converter_add(8),
    convadd(7)=>converter_add(7), convadd(6)=>converter_add(6),
    convadd(5)=>converter_add(5), convadd(4)=>converter_add(4),
    convadd(3)=>converter_add(3), convadd(2)=>converter_add(2),
    convadd(1)=>converter_add(1), convadd(0)=>converter_add(0),
    address2(14)=>add2(14), address2(13)=>add2(13), address2(12)=>add2(12),
    address2(11)=>add2(11), address2(10)=>add2(10), address2(9)=>add2(9),
    address2(8)=>add2(8), address2(7)=>add2(7), address2(6)=>add2(6),
    address2(5)=>add2(5), address2(4)=>add2(4), address2(3)=>add2(3),
    address2(2)=>add2(2), address2(1)=>add2(1), address2(0)=>add2(0));

  XLXI_3 : vgacore
    PORT MAP (reset=>XLXN_23, clock=>vclk, data(7)=>data2(7),
    data(6)=>data2(6), data(5)=>data2(5), data(4)=>data2(4),
    data(3)=>data2(3), data(2)=>data2(2), data(1)=>data2(1),
    data(0)=>data2(0), hsyncb=>hsync, vsyncb=>vsync, csb=>open, oeb=>open,
    web=>open, rgb(5)=>RGB(5), rgb(4)=>RGB(4), rgb(3)=>RGB(3),
    rgb(2)=>RGB(2), rgb(1)=>RGB(1), rgb(0)=>RGB(0), addr(14)=>VGA_add(14),
    addr(13)=>VGA_add(13), addr(12)=>VGA_add(12), addr(11)=>VGA_add(11),
    addr(10)=>VGA_add(10), addr(9)=>VGA_add(9), addr(8)=>VGA_add(8),
    addr(7)=>VGA_add(7), addr(6)=>VGA_add(6), addr(5)=>VGA_add(5),
    addr(4)=>VGA_add(4), addr(3)=>VGA_add(3), addr(2)=>VGA_add(2),
    addr(1)=>VGA_add(1), addr(0)=>VGA_add(0), vdone=>vdone);

END SCHEMATIC;
```


vgacore
hsyncb
vsyncb
csb
oeb
web
rgb(5:0)
addr(14:0)
vdone
reset
clock
data(7:0)
hsync
vsync
RGB(5:0)
addr3(14:0)
data2(7:0)
vclk
switch
Q_0
address2(14:0)
vgadd(14:0)
convadd(14:0)
oeb
web
reset
convert
clk
start
data1(7:0)
enable
csb
web
clr
address(14:0)
data2(7:0)
data1(7:0)
control
reset
sel
clk_conv
start
V_reset
enable
en
rsd
Q_0
V_done
clr
clk
rsd
ens
sel
clock
Q0

Overall System

```
-- Vhdl model created from schematic batmouse.sch - Tue Jun 22 22:41:32 2004

LIBRARY ieee;
LIBRARY UNISIM;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;
USE UNISIM.Vcomponents.ALL;

ENTITY batmouse IS
   PORT ( clk          :    IN      STD_LOGIC;
          clockdiv     :    IN      STD_LOGIC;
          data1        :    IN      STD_LOGIC_VECTOR (7 DOWNTO 0);
          RGB          :    OUT     STD_LOGIC_VECTOR (5 DOWNTO 0);
          add1         :    OUT     STD_LOGIC_VECTOR (7 DOWNTO 0);
          add2         :    OUT     STD_LOGIC_VECTOR (14 DOWNTO 0);
          hsync        :    OUT     STD_LOGIC;
          oeb          :    OUT     STD_LOGIC;
          oeb1         :    OUT     STD_LOGIC;
          q9           :    OUT     STD_LOGIC;
          vsync        :    OUT     STD_LOGIC;
          web          :    OUT     STD_LOGIC;
          web1         :    OUT     STD_LOGIC;
          data2        :    INOUT           STD_LOGIC_VECTOR (7 DOWNTO 0));

end batmouse;

ARCHITECTURE SCHEMATIC OF batmouse IS
   SIGNAL XLXN_13  :    STD_LOGIC;
   SIGNAL enable   :    STD_LOGIC;
   SIGNAL q9_DUMMY      :    STD_LOGIC;

   ATTRIBUTE fpga_dont_touch : STRING ;

   COMPONENT conter
      PORT ( clk       :    IN      STD_LOGIC;
             en        :    IN      STD_LOGIC;
             cnt       :    OUT     STD_LOGIC;
             address1  :    OUT     STD_LOGIC_VECTOR (7 DOWNTO 0));
   END COMPONENT;

   COMPONENT mux8_4
      PORT ( clock     :    IN      STD_LOGIC;
             sel       :    IN      STD_LOGIC;
```

```
        y0              :       OUT     STD_LOGIC;
        y1              :       OUT     STD_LOGIC;
        y3              :       OUT     STD_LOGIC);
  END COMPONENT;

  COMPONENT samkmorsy
    PORT ( data1        :       IN      STD_LOGIC_VECTOR (7 DOWNTO 0);
        vclk            :       IN      STD_LOGIC;
        clock           :       IN      STD_LOGIC;
        Q9              :       IN      STD_LOGIC;
        data2           :       INOUT           STD_LOGIC_VECTOR (7 DOWNTO 0);
        oeb             :       OUT     STD_LOGIC;
        add2            :       OUT     STD_LOGIC_VECTOR (14 DOWNTO 0);
        vsync           :       OUT     STD_LOGIC;
        RGB             :       OUT     STD_LOGIC_VECTOR (5 DOWNTO 0);
        web             :       OUT     STD_LOGIC;
        hsync           :       OUT     STD_LOGIC;
        reset           :       OUT     STD_LOGIC;
        sel             :       OUT     STD_LOGIC;
        ena             :       OUT     STD_LOGIC;
        red             :       OUT     STD_LOGIC);
  END COMPONENT;

BEGIN
  q9 <= q9_DUMMY;

  XLXI_4 : conter
    PORT MAP (clk=>XLXN_13, en=>enable, cnt=>q9_DUMMY,
address1(7)=>add1(7),
    address1(6)=>add1(6), address1(5)=>add1(5), address1(4)=>add1(4),
    address1(3)=>add1(3), address1(2)=>add1(2), address1(1)=>add1(1),
    address1(0)=>add1(0));

  XLXI_3 : mux8_4
    PORT MAP (clock=>clockdiv,  sel=>q9_DUMMY, y0=>XLXN_13,
    y1=>oeb1, y3=>web1);

  XLXI_2 : samkmorsy
    PORT MAP (data1(7)=>data1(7), data1(6)=>data1(6), data1(5)=>data1(5),
    data1(4)=>data1(4), data1(3)=>data1(3), data1(2)=>data1(2),
    data1(1)=>data1(1), data1(0)=>data1(0), vclk=>clk, clock=>clk,
    Q9=>q9_DUMMY, data2(7)=>data2(7), data2(6)=>data2(6), data2(5)=>data2(5),
    data2(4)=>data2(4), data2(3)=>data2(3), data2(2)=>data2(2),
    data2(1)=>data2(1), data2(0)=>data2(0), oeb=>oeb, add2(14)=>add2(14),
    add2(13)=>add2(13), add2(12)=>add2(12), add2(11)=>add2(11),
    add2(10)=>add2(10), add2(9)=>add2(9), add2(8)=>add2(8), add2(7)=>add2(7),
    add2(6)=>add2(6), add2(5)=>add2(5), add2(4)=>add2(4), add2(3)=>add2(3),
    add2(2)=>add2(2), add2(1)=>add2(1), add2(0)=>add2(0), vsync=>vsync,
    RGB(5)=>RGB(5), RGB(4)=>RGB(4), RGB(3)=>RGB(3), RGB(2)=>RGB(2),
    RGB(1)=>RGB(1), RGB(0)=>RGB(0), web=>web, hsync=>hsync, reset=>open,
    sel=>open, ena=>enable, red=>read_DUMMY);

END SCHEMATIC;
```


samkmorsy
conter
mux8_4