School of Electronic and Electrical Engineering

UNIVERSITY OF LEEDS

ELEC5563 Individual Project

Implementation of Counter, State Machine, function generator and oscilloscope on the FPGA using Verilog HDL

Yingjie Luan

September 23, 2015

Contents

1 Introduction

1.1 Objective

The main objective of this project is to build a digital Oscilloscope based on FPGA using *verilog* language and *Schematic Design*. In general, I am required to read the data from the AD converter and then mapping the data onto monitor via VGA according to a specifically designed timing sequence.

1.2 Main Aspect

After one month of works, I have successfully implemented the primary goal, which is to mapping the AD data onto VGA. Not only so, an edge triggering mechanism is implemented for providing a static image, an arbitrary function generator is also implemented as a byproduct of this project. Oscilloscope itself is fully configurable, we can use the switches to control the sample frequency, wave time division, wave center location and wave amplitude.

Below is the project result:

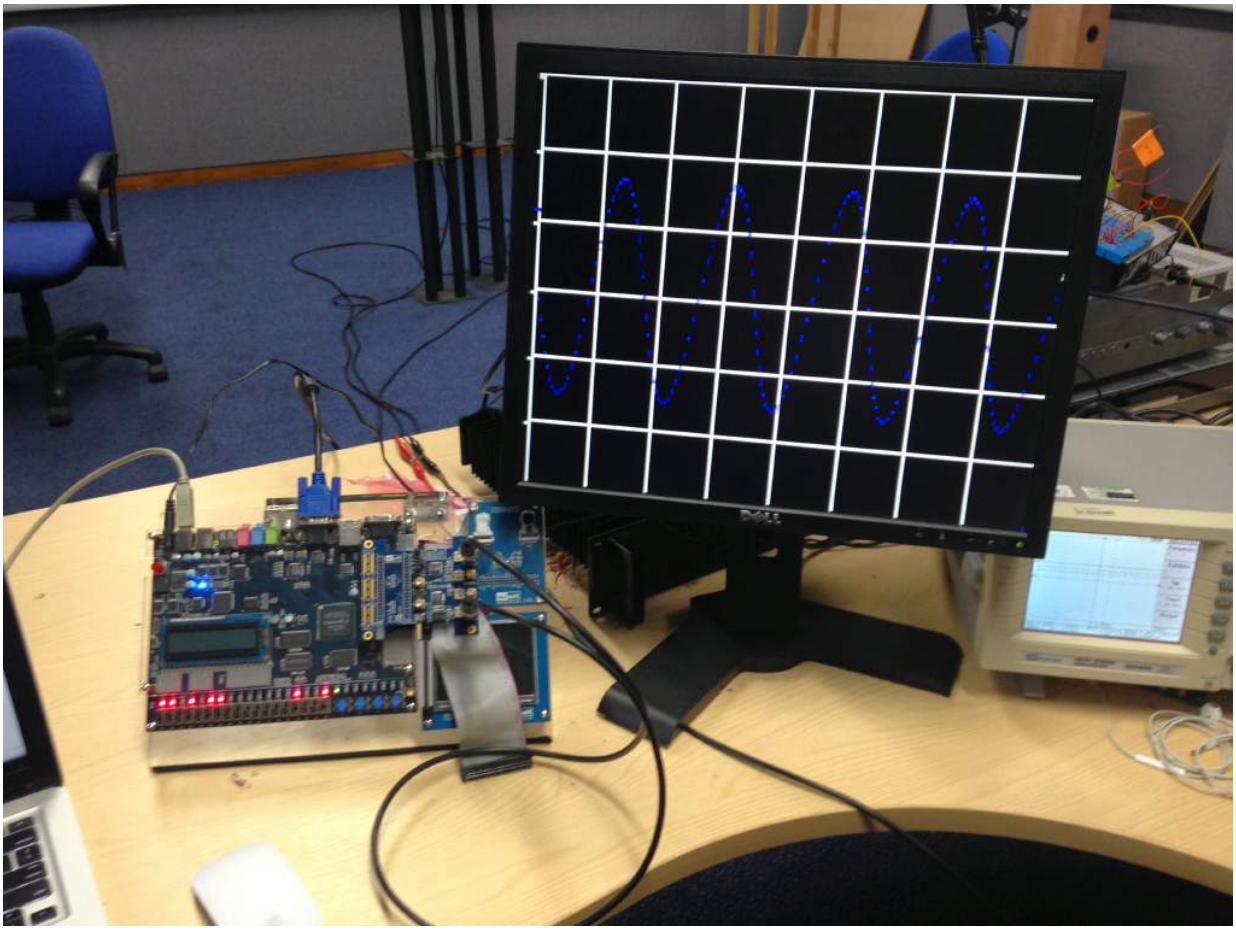

Figure 1: This is the working result. The oscilloscope is reading the value from the generator

The board I was working on is called *DE2-35*, which is an implementation of the Altera Cyclone®II FPGA chip 2C35 and the chip code is EP2C35F672C6. The board itself is designed for the purpose of evaluation and education and itself is built in with all sorts of basic components, *i.e.* switches, push button, LCD screen, VGA modulator, GPIO pins etc. Beside from the FPGA board, I also used THDB_ADA(ADA) daughter board for doing AD and DA conversion. Below is the specification of the chip and the AD-DA daughter board according to [1] and [5].

The technical specification of the AD-DA daughter

- The AD channel is of 14-bit resolution and data rate up to 65 MSPS. Input volage range 2V p-p.
- The DA channel is of 14-bit resolution and data rate up to 125 MSPS. Output range 2V p-p.

The technical specification of the chip EP2C35F672C6

- 33,216 LEs

- 105 M4K RAM blocks
- 483,840 total RAM bits
- 35 lembedded multipliers
- 4PLLs
- 475 user I/O pins
- FineLine BGA 672-pin package

For the software part of the project, a variety of software was used, I used *quartus* for compiling the source code and downloading the program onto the board using JTAG. *Git* for managing the development of the code. *iverilog* for debugging. *Matlab* for doing calculation and generating .mif file. *verilog HDL* is the programming language.

The project itself is hosted at *https://github.com/y1275963/vga_basics*. With around 24 branches and 80 commits.

2 Board Specification

2.1 Board Details

For the board itself, I implemented a single route oscilloscope at the sampling frequency of 13.5Mhz or 27Mhz or 54Mhz and an arbitrary single route capable of generating wave at the base frequency of 97.66Khz and of the data point of 1024 points.

2.2 The controlling method

Below is the supported control method:

Figure 2: This is the overall view of the controlling method.

- Switch 17: Switch for turning the monitor on and off.
- Switch 16: Switch for stoping refreshing the screen.
- Switch 15: Switch for letting the screen only show the background.
- Switch 13-14: Switches for changing time division by changing sampling frequency.

  Specifically:
  - Switch 14 on, Switch 13 off: Sampling at 13.5Mhz.
  - Switch 14 off, Switch 13 on: Sampling at 54Mhz.
  - Switch 14 and 13 both on or off: Sampling at 27Mhz.
- Switch 11-12: Supporting for changing time division by changing sample rate.

  Specifically:
  - Switch 12 and 11 both off: Sampling at normal speed.
  - Switch 12 off and 11 on: Sampling at 1 time faster normal speed.
  - Switch 12 on and 11 off: Sampling at 2 time faster normal speed.
  - Switch 12 and 11 both on: Sampling at 3 time faster normal speed.
- Switch 8-10: Supporting for shift the wave up.

  When switch is both off, no shift is made. And by turning it on or off, we can achieve shifting by 1,2,3,4,5,6,7 pixel above the central line.
- Switch 5-7: Supporting for shift the wave down.

  When switch is both off, no shift is made. And by turning it on or off, we can achieve shifting by 1,2,3,4,5,6,7 pixel below the central line.
- Switch 1-4: Supporting for changing the amplitude of the wave.

  The changing factor is done by a factor of 2. And switch 4-3 allowing for scale up, switch 2-1 allowing for scale down.
- Switch 0: Supporting for turning the function generator on and off.
- Push button 0: Supporting for changing the signal generator generated wave. The design itself allowing the program to record 4 sets of arbitrary waves before compiling the program. (Matlab program 3.7.1 is provided for generating such wave)

2.3 The measuring method

Below is the supported measuring method to get the data from the board.

- The VGA monitor

  The monitor is the main source of information. We can use it to see the shape of the wave. Beside from this, because the regular grid is implemented as the background, measurement is made possible. Two measurement tables 6.1 6.2 are provided for measuring the frequency and the amplitude of the wave.
- The state machine debugger

Figure 3: This is the state indicator.

LED Green 7 and 6 are implemented for indicating state machine's current state.(Image blow)

Below is the meaning of the light:

- LED 7 is on, 6 is on, state 11: Meaning that the system is sampling.
- LDE 7 is on, 6 is off, state 10: Meaning that the system is delayed, to moderate the refresh rate down to around 50hz.
- LED 7 is off, 6 is on, state 01: Meaning that the system is drawing the wave from the RAM according to the sampled signal.
- LED 7 is off, 6 is off, state 00: Meaning that the system is clean the screen by redrawing a regular grid background.

In reality, as we can see from the image above, because the system is the state of delay for most of its functioning time, we can see LED 7 is always on, and LED 6 is glimmering. Which means the system spent most of its time in delaying(state 10).

- Switch indicator

  The switch indicator is used to indicator which switch is on or off, as we can see from the image below:

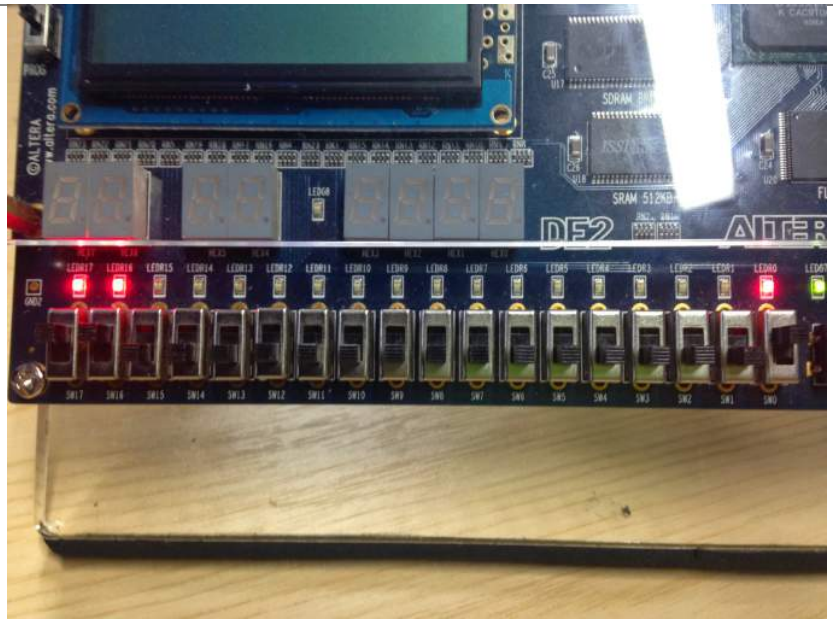

Figure 4: This is switch indicator.

2.4 The work flow

This is the overall schematic chart:

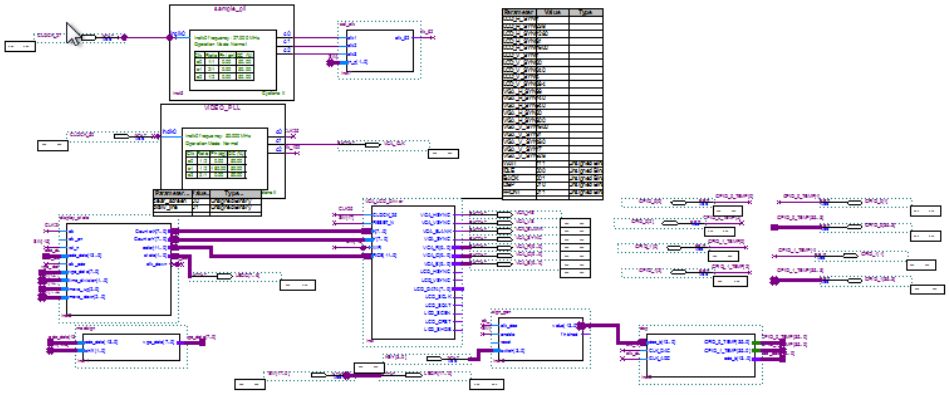

Figure 5: This is the work flow of the oscilloscope.

The board will execute two jobs simultaneously, below is the flowchart of the Oscilloscope:

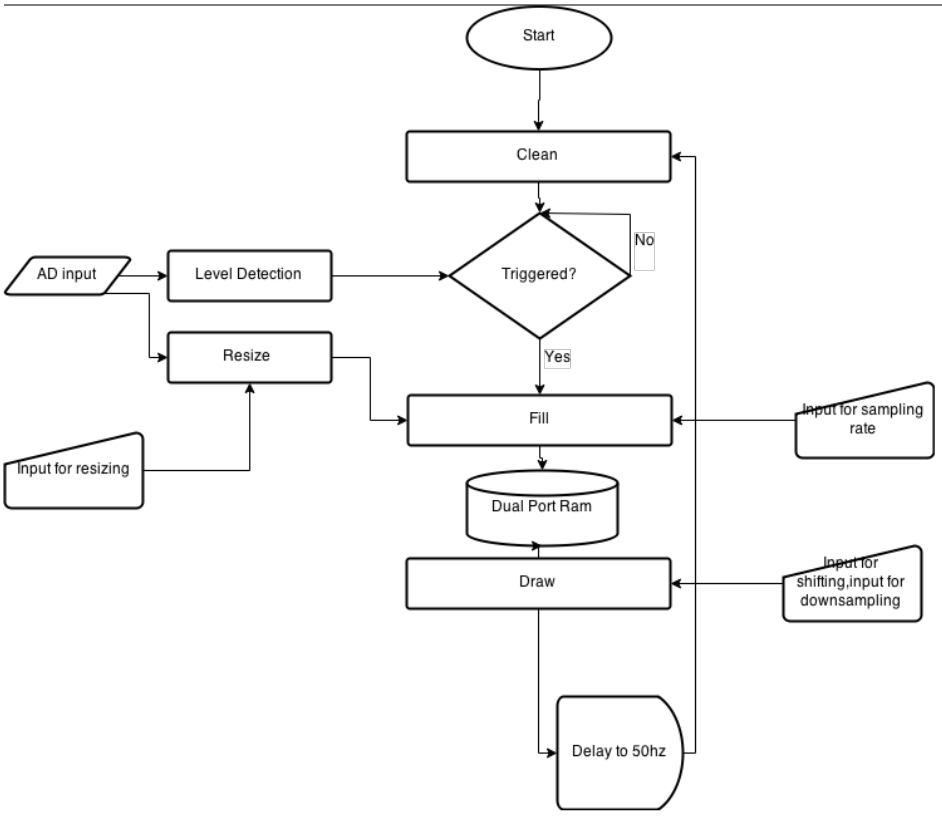

Figure 6: This is the work flow of the oscilloscope.

Below is the flowchart of the arbitrary function generator:

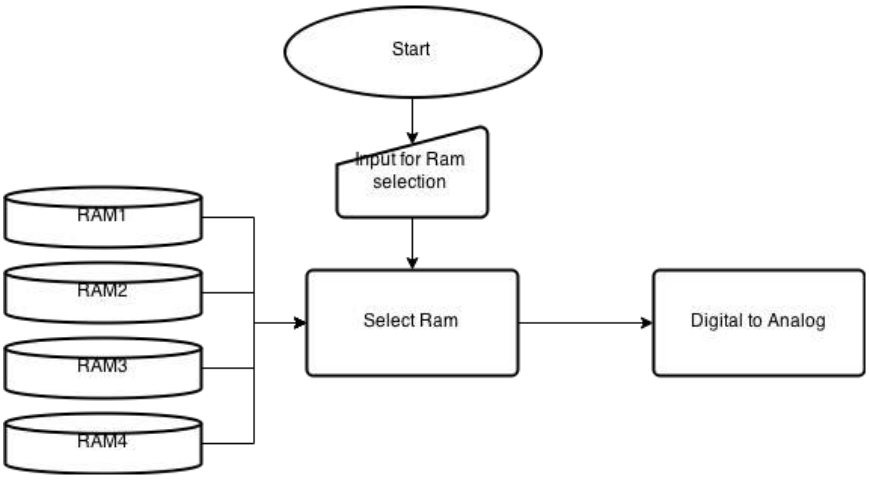

Figure 7: This is work flow of the function generator.

3 Implementation Details

3.1 The Counter

The counter is the building block of the entire project and it is used throughout the entire project. Below is the detailed list of the application of the counters:

- In drawing process: Counters are used to provide VGA the pixel coordinates.
- In delay process: Counter is used to crate a delay to lower down the refresh rate.
- In sampling process and AD process: Counter is used to provide the RAM with memory index. A counter in sampling process also support downsampling the RAM.

To connect counters with state machine, all the counters are implemented with 2 input signals and 1 output signal:

- enable: To start the counter
- reset: To reset the counter
- finished: indicate if the counter has finished one round or not.

3.1.1 The basic Counter

All the counters are built on top the basic structure from the one blow:

```
input clk;
input enable,reset;

output finished;
output reg [7:0] CounterX;
wire CounterXmaxed = (CounterX==8'd159);
assign finished = (CounterXmaxed==1);

always @(posedge clk)
begin
    if(reset == 1)
        CounterX <= 0;
    else
    begin
    if(CounterXmaxed)
        CounterX <= 0;
    else if(enable == 1)
        CounterX <= CounterX + 1;
     end
end
```

By setting the value of the maximum value of the counter, we can manage to let the counter counts from 0 to its max value according to the clock. And the finished signal is emitted right at the moment the counter reach its maximum value.

3.1.2 Jumping Counter

The down sampling is implemented by down sampling the output of the RAM instead of down sampling the output from the AD converter.

To support the down sampling, we simply change the increment factor of the counter and hence we change the RAM address we are going to access, below is an example of the jumping counter:

```
input [1:0] time_division;
wire [2:0] read_time_division = time_division + 1; // to avoid 0, start from 1

output reg [7:0] read_CounterX;

wire read_CounterX_Maxed = (read_CounterX >= 8'd1023);

always @(posedge clk)
begin
    if(reset == 1)
        read_CounterX <= 0;
    else
    begin
    if(read_CounterX_Maxed)
        read_CounterX <= 0;
    else if(enable == 1)
        read_CounterX <= read_CounterX + read_time_division + 1;
     end
end
```

3.1.3 Counter on top of the Counter

To support providing (x, y) coordinators for the VGA module while cleaning the whole screen, a Y counter is built on top of a X counter, blow is the example:

```
input clk;
input reset,enable;

output reg [7:0] CounterX,CounterY;
output finished;

wire [14:0] address;

wire CounterXmaxed = (CounterX==8'd159); // 159
wire CounterYmaxed = (CounterY==8'd119); // 119
assign finished = ((CounterXmaxed == 1) && (CounterYmaxed == 1));

assign address = CounterX + CounterY * 160;

always @(posedge clk)
begin
    if(reset == 1)
        CounterX <= 0;
    else
    begin
    if(CounterXmaxed)
      CounterX <= 0;
    else if(enable == 1)
      CounterX <= CounterX + 1;
    end
end

always @ (posedge clk)
begin
    if(reset == 1)
        CounterY <= 0;
    else
    begin
        if(CounterXmaxed == 1)
```

3.1.4 iverilog output for testing

Because we need a reliable counter to make sure it goes exactly as we wish, *iverilog* is used for testing, below is its output:

This is snapshot of the testing from the basic counter:

```
ticket:23370    ,enable:1       ,reset:0        ,x:155  ,finished :0
ticket:23380    ,enable:1       ,reset:0        ,x:156  ,finished :0
ticket:23390    ,enable:1       ,reset:0        ,x:157  ,finished :0
ticket:23400    ,enable:1       ,reset:0        ,x:158  ,finished :0
ticket:23410    ,enable:1       ,reset:0        ,x:159  ,finished :1
ticket:23420    ,enable:1       ,reset:0        ,x:  0  ,finished :0
```

Below is the snapshot of the testing from the jumping counter:

```
ticket:23950    ,enable:1       ,reset:0        ,x:147  ,finished :0
ticket:23960    ,enable:1       ,reset:0        ,x:150  ,finished :0
ticket:23970    ,enable:1       ,reset:0        ,x:153  ,finished :0
ticket:23980    ,enable:1       ,reset:0        ,x:156  ,finished :0
ticket:23990    ,enable:1       ,reset:0        ,x:159  ,finished :1
ticket:24000    ,enable:1       ,reset:0        ,x:  0  ,finished :0
ticket:24010    ,enable:1       ,reset:0        ,x:  3  ,finished :0
ticket:24020    ,enable:1       ,reset:0        ,x:  6  ,finished :0
```

Below is the snapshot of the testing from the xy counter:

```
ticket:575970   ,enable:1       ,reset:0        ,x:155  ,y:119  ,finished :0
ticket:575980   ,enable:1       ,reset:0        ,x:156  ,y:119  ,finished :0
ticket:575990   ,enable:1       ,reset:0        ,x:157  ,y:119  ,finished :0
ticket:576000   ,enable:1       ,reset:0        ,x:158  ,y:119  ,finished :0
ticket:576010   ,enable:1       ,reset:0        ,x:159  ,y:119  ,finished :1
ticket:576020   ,enable:1       ,reset:0        ,x:  0  ,y:  0  ,finished :0
ticket:576030   ,enable:1       ,reset:0        ,x:  1  ,y:  0  ,finished :0
ticket:576040   ,enable:1       ,reset:0        ,x:  2  ,y:  0  ,finished :0
ticket:576050   ,enable:1       ,reset:0        ,x:  3  ,y:  0  ,finished :0
```

Below is the example of the test bench:

```
module ramfill_tb;

// module ramfill(clk_adc,enable,reset,finfished,adc_data,vga_data,CounterX);
reg clk;
wire [7:0] CounterX;
reg enable,reset;
wire finished;

vga_sin u0(
    .clk(clk),
    .reset  (reset),
    .enable (enable),
    .finished (finished),
    .CounterX   (CounterX));

always
   #5 clk = ~clk;
```

```
initial begin
    reset = 0;
    enable = 0;
    clk = 1;

    #10 reset = 1;//after 10 tickets
    #10 reset = 0;//then after 10 tickets,reset is set 1 after reset == 1 is kept for ↩
        10 tickets
    #10 enable = 1;
    #20000 reset = 1;//try again
    #200 reset = 0;
    #2000 enable = 1;
    #2000 enable = 0;
    #200 reset = 0;
    #25 $finish;
end


initial  begin
    $monitor("ticket:%g\t,enable:%b\t,reset:%b\t,x:%d\t,finished :%b",$time,enable,↩
        reset,CounterX,finished);
end

endmodule
```

3.2 The RAM

It would be good if we can write some test bench with RAM, but because it is a Mega function and I cannot find a way to compile it.

The RAM is used throughout the project as well, below is a list of its usage:

- In clean process: A RAM is used to provide the grid image which is going to be drawn.
- In sampling process: A RAM is used to store the sampling information
- In drawing process: The same RAM is used to retrieve stored sampled data.
- In DA process: 4 RAMs are used to store the pro recored wave information.

3.2.1 One Port reading only RAM

This is the basic usage of the RAM, and the purpose of this is simply to get the initialing memory information from the RAM, below is an example:

```
wire rambuffer;
ram_background ram_entity(
    .address(address),
    .clock(clk),
    .wren(1'b0),
    .q(rambuffer));
endmodule
```

The *address* nets are defined by other counters.

3.2.2 Dual Port RAM

That is hardest part of the entire project, and the natural way of doing so is by using a FIFO mega function [2]. But because *quartus* only provide fixed length FIFO and due to various

other reasons, I cannot figure out a right way of stopping the FIFO from overflow, an image is provided to demonstrate the problem:

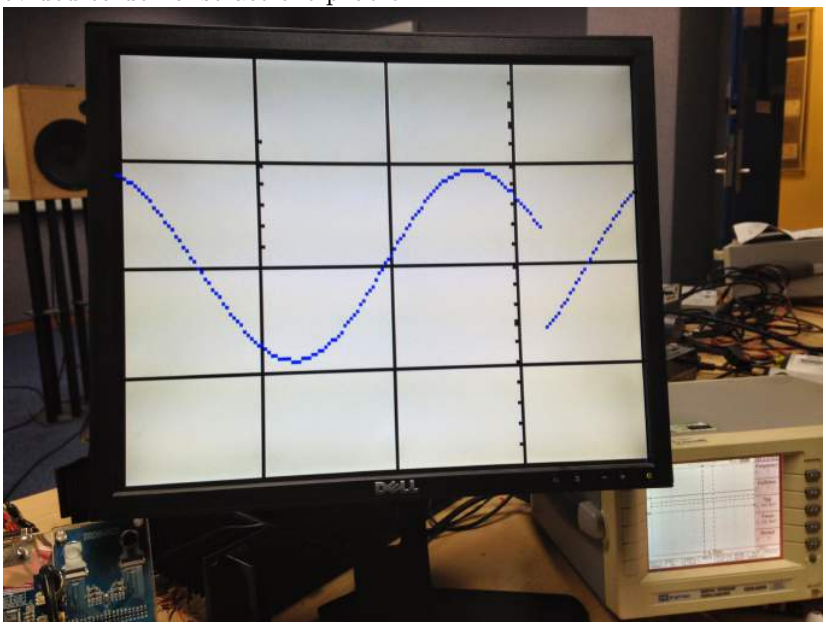

Figure 8: There is a periodic unsolvable overflow problem at each image when using FIFO.

And a dual port asynchronous memory is a core component of the entire design, it is used to link the AD sampled data to the image on the screen. Its difficulties are listed as follows:

- It must support one reading port and one writing port running at different speed.
- It is written at writing speed, but at the reading cycle the finishing of the writing must be able to be detected. And the clock shift is unknown.
- There is a trigging mechanism in between, so the port is not being written periodically.

And the solution is the example below:

```
// sync clocks
reg prolong_finished = 0;
always @(posedge w_finished) // flip flop
begin
    prolong_finished = prolong_finished + 1;
end

reg previous_signal;
reg r_finished;
always @(posedge clk)
begin
    previous_signal <= prolong_finished;
    r_finished <= previous_signal ^ prolong_finished;
    // or with its previous value.
end
```

The writing finished signal will change the state of a flip-flop register, and the reading clock will be constantly detecting the level of the register and when it detects the difference it will output a reading finished signal.

By doing so, I successfully made the writing finished clock readable from the reading cycle. Therefor, the state machine will be able to detect if the sampling is finished at a different clock.

Below is a simulation result:

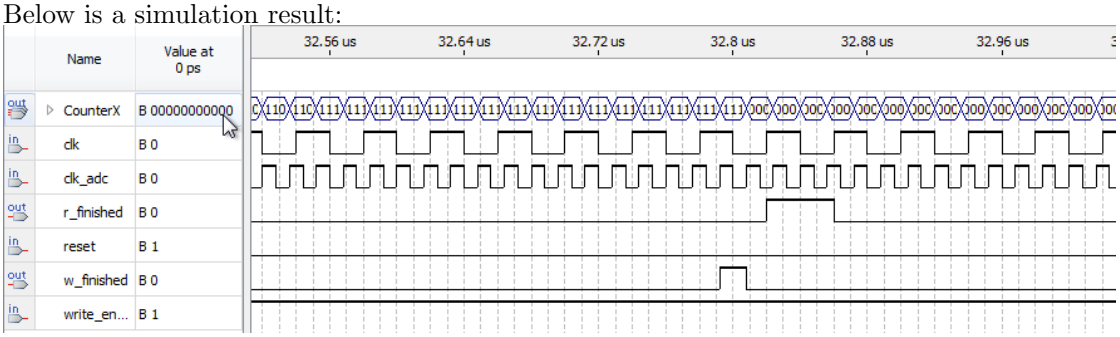

Figure 9: This is simulation of the synchronization of the finished clock.

3.3 The State Machine

The state machine runs at the pixel refresh rate, which is 25Mhz, and it is the main block of the whole project.

The implementation of state machine goes as the following, Once a state is finished a new state will be defined. And the state machine changes its state accordingly.

For all the states, it always goes from enabling its module and reseting all the others modules by using *enable*, *reset* and *finished*.

It is separated in 4 stages:

- clean: Call the clean module to clean the screen, and it goes to *fill* module when it is finished.
- fill: Call the Sampling module to write memory to the RAM, and it goes to *draw* module when it is finished. Notice that the state machine detects the state of the filling module at 25Mhz but the filling module itself runs at sampling frequency.
- draw: Call the drawing module to draw the memory according to the RAM, and it goes to *delay* module when it is finished.
- delay: Call the delay module to delay the whole process, and it goes to *clean* module when it is finished.

Below is the snapshot code of the state machine:

```
always @ (posedge clk)
begin
    if(rst_n == 1)
        state <= clear_screen;
    else
        state <= next_state;
end
```

```
always @ * // combinational circuit
    case(state)
        clear_screen:
        begin
            enable_clear = 1;
            reset_clear = 0;

            enable_sin = 0;
            reset_sin = 1;

            enable_delay = 0;
            reset_delay = 1;

            enable_fill = 0;
            reset_fill = 1;

            CounterX = CounterX_clear;
            CounterY = CounterY_clear;
            color = color_clear;
            if(finished_clear == 0)
                next_state = clear_screen;
            else
                next_state = fill;
        end
        fill:
        begin
            enable_fill = 1;
            reset_fill = 0;

            enable_clear = 0;
            reset_clear = 1;

            enable_sin = 0;
            reset_sin = 1;

            enable_delay = 0;
            reset_delay = 1;

            CounterX = 8'bzzzz_zzz;
            CounterY = 8'bzzzz_zzz;
            color = 12'bzzz_zzz_zzz_zzz;

            if(finished_fill == 0)
                next_state = fill;
            else
                next_state = draw_line;

        end
        draw_line:
        begin
            enable_sin = 1;
            reset_sin = 0;

            enable_delay = 0;
            reset_delay = 1;

            enable_clear = 0;
            reset_clear = 1;

            enable_fill = 0;
            reset_fill = 1;

            CounterX = CounterX_sin;
            CounterY = ram_output + move_down - move_up;
            color = color_sin;
            if(finished_sin == 0)
                next_state = draw_line;
```

```
            else
                next_state = do_nothing;
        end
        do_nothing:
        begin
            enable_delay = 1;
            reset_delay = 0;

            enable_clear = 0;
            reset_clear = 1;

            enable_sin = 0;
            reset_sin = 1;

            enable_fill = 0;
            reset_fill = 1;

            CounterX = 8'bzzzz_zzz;
            CounterY = 8'bzzzz_zzz;
            color = 12'bzzz_zzz_zzz_zzz;
            if(finished_delay == 0)
                next_state = do_nothing;
            else
                next_state = clear_screen;
        end
    endcase
```

3.4 The Trigger

The trigger detects the input signal and emit an enabling signal to start the sampling, two mechanism is implemented and tested.

The first one is simply to use the most significant bit of the input signal:

```
wire last_bit_data = adc_data[13];
reg previous_adc_data;
// wait until trigger happen
always @ (posedge clk_adc)
begin
    previous_adc_data <= last_bit_data;
    if(enable == 1)
    begin
        if((last_bit_data ^ previous_adc_data) == 1) // unchanged until enable returns↩
             to 0.
            write_enable <= 1;
    end
    else
        write_enable <= 0;
end
```

and according to [3], I implemented a different design:

```
always @ (posedge clk_adc)
begin
    if(enable == 1)
    begin
        if(Trigger == 1) // unchanged until enable returns to 0.
            write_enable <= 1;
    end
    else
        write_enable <= 0;
end
```

```
reg Threshold1, Threshold2;
always @(posedge clk_adc) Threshold1 <= (adc_data>=14'b10_000_000_000_000);
always @(posedge clk_adc) Threshold2 <= Threshold1;

wire Trigger = Threshold1 & ~Threshold2;  // if positive edge, trigger!
```

Both of the design work, but according to the performance, the last one works better than the former.

Clearly, we can pin out the *trigger* signal and therefor enable the external trigger signal.

3.5 PLL

PLLs are used to change the clock,

The first one receive signal from 50Mhz and generate 3 signals. Two for VGA module, which are 25Mhz and 25Mhz with 180 degree shift. One for DA module which is 100Mhz.

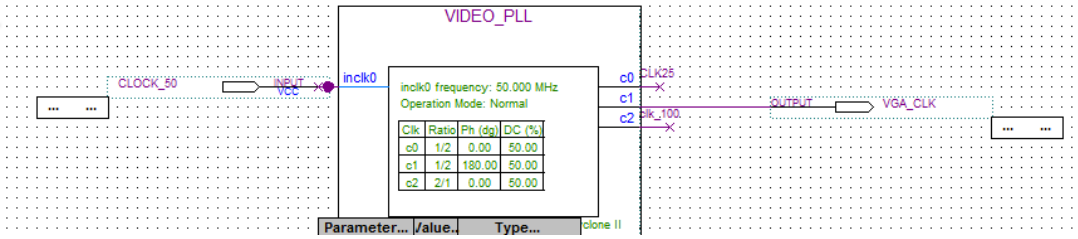

Figure 10: This is the first PLL.

The second one receive signal from 27Mhz and generate 3 signals, all of them are used for providing a adjustable sampling rate, which are 13.5Mhz, 27Mhz and 54Mhz.

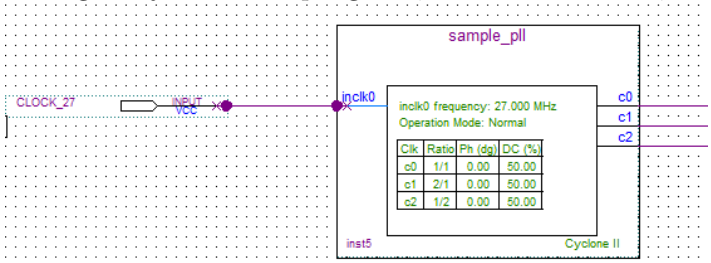

Figure 11: This is the second PLL.

3.6 University Provided modules

3.6.1 The VGA Adaptor Module

VGA module is used to connect the board with the monitor, below is the schematic snapshot:

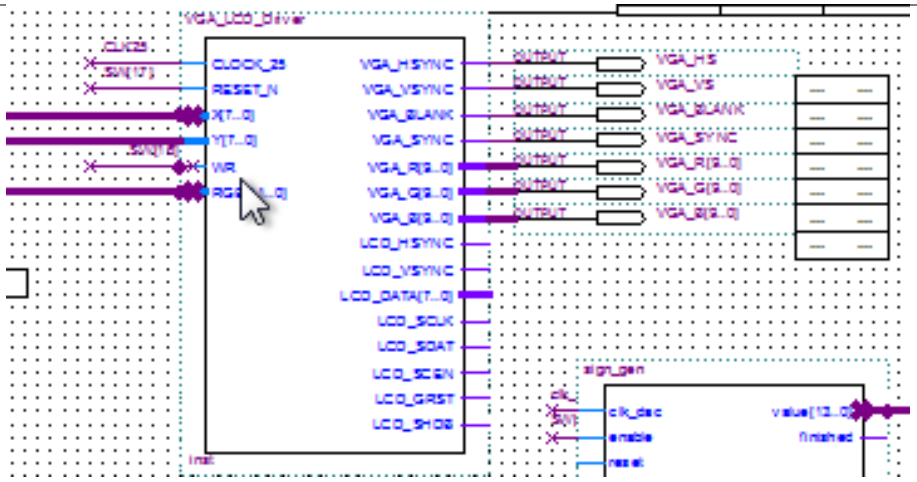

Figure 12: This is the VGA Adaptor.

Although it is a different module, but I use it according to this table [4].

3.6.2 The AD-DA Adaptor Module

AD-DA adapter module provide a interface for me to control the AD-DA converter. Below is the snapshot of its schematic design:

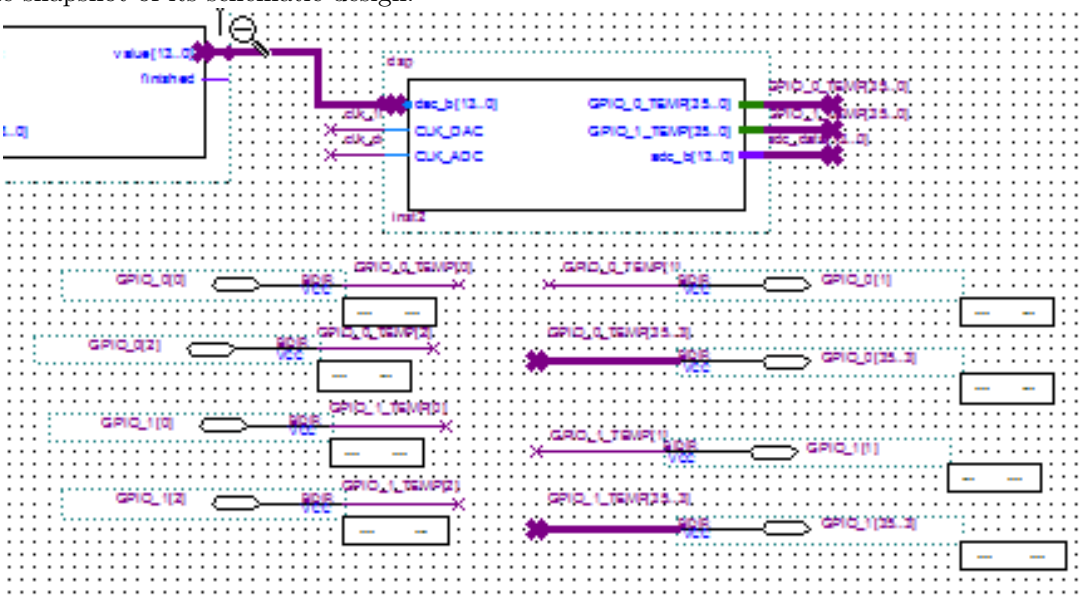

Figure 13: This is the ADDA Adaptor.

3.7 Matlab code

Aside from the *verilog HDL* code, there is roughly 1.5% Matlab code.

3.7.1 Matlab code for RAM initialization file

Because RAM need to read its initialing file, two Matlab files are written, one is to generate the background and another one is to generate wave for DA converter.

The one below is for generating the background, it is able to convert an image into RAM initializing file:

```
function imgscaled = miffilegen(infile, outfname, numrows, numcols)
%miffilegen('photo.jpg','test.mif',120,160) generate black and white image.

img = imread(infile);
bw = im2bw(img,0.5);

imgresized = imresize(bw, [numrows numcols]);

[rows, cols, ~] = size(imgresized);

imgscaled = imgresized;
imshow(imgscaled*16);

fid = fopen(outfname,'w');

fprintf(fid,'-- %3ux%3u 1bit image color values\n\n',rows,cols);
fprintf(fid,'WIDTH = 1;\n');
fprintf(fid,'DEPTH = %4u;\n\n',rows*cols);
fprintf(fid,'ADDRESS_RADIX = UNS;\n');
fprintf(fid,'DATA_RADIX = UNS;\n\n');
fprintf(fid,'CONTENT BEGIN\n');
imshow(imgscaled);
count = 0;
for r = 1:rows
    for c = 1:cols
%         red = uint16(imgscaled(r,c,1));
%         green = uint16(imgscaled(r,c,2));
%         blue = uint16(imgscaled(r,c,3));
%         color = red*(256) + green*16 + blue;
        fprintf(fid,'%4u : %4u;\n',count, imgscaled(r,c));
        count = count + 1;
    end
end
fprintf(fid,'END;');
fclose(fid);

return
```

The code is modified to be able to generate an image directly from Matlab's array for accuracy, and the standard background is a 160 * 120 image shown blow:

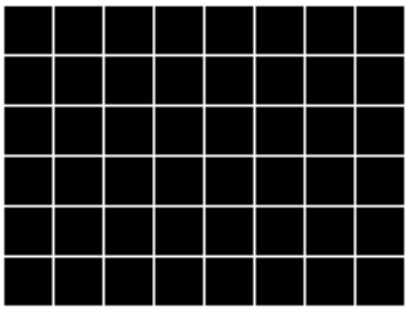

Figure 14: This is the background image.

This one is to write the wave to initialize the RAM for the signal generator:

```
function signal_generator(outfname)
% Depth is 2014, word length is  2^14.
% The DAC frequency is 100Mhz.
dac_freq = 100*10^6;
display(['for length of ',num2str(1024),'the frequency is ',num2str(dac_freq/1024)])
t = linspace(0,2*pi,1024);
y = gen_y(t)  - 0.01;

y = y./max(abs(y)); % in the range of -1 to 1


shift = 2^13-1;
y = int32(y*shift);
y = y + shift;

%0->8191*2

rows = length(y);

fid = fopen(outfname,'w');

fprintf(fid,'-- %3ux%d 1bit sine values\n\n',rows,ceil(log2(double(max(y)))));
fprintf(fid,'WIDTH = %d;\n',ceil(log2(double(max(y)))));
fprintf(fid,'DEPTH = %4u;\n\n',rows);
fprintf(fid,'ADDRESS_RADIX = UNS;\n');
fprintf(fid,'DATA_RADIX = UNS;\n\n');
fprintf(fid,'CONTENT BEGIN\n');
count = 0;
for r = 1:rows
    fprintf(fid,'%4u : %4u;\n',count, y(r));
    count = count + 1;
end
fprintf(fid,'END;');
fclose(fid);
```

And to generate the desired wave we need to define function *gen_y(t)*, below is an example of which:

```
function y = gen_y(x)
```

```
    y = square(x);
```

And at the mean time it can calculate the base frequency of the generated wave:

```
for length of 1024the frequency is 97656.25
```

The real frequency is the base frequency multiples the signal frequency divided by 2π.

3.7.2 Matlab code for doing measurement

Those two codes enable us to tell the frequency and amplitude based on the output image on the screen and the switches:

Measuring Frequency Below is the code for doing so:

```
function fw = cal_fre(sampling_rate,jumping,number_wave)

switch sampling_rate
    case '10'
        fs = 13.5*10^6;
    case '01'
        fs = 54*10^6;
    case {'11','00'}
        fs = 27*10^6;
end

switch jumping
    case '00'
        kj = 1;
    case '01'
        kj = 2;
    case '10'
        kj = 3;
    case '11'
        kj = 4;
end

fw = number_wave * fs /(kj*160);
```

It is simply an implementation of the formula below:

$$f_w = \frac{n * fs}{k_j * 160}$$

Below is a example of usage,

Figure 15: Frequency measurement, reading from switch 14-13 is 01 and reading from switch 12 to 11 is 11. There is around 4.5 waves on the screen.

To measure the frequency, we need to know the shape of the wave, if we look at the image above, we can see that there is roughly 4.6 peak in the whole image, and by putting that number in along with the reading from the switch 14,13 and switch 12,11, we will get the following result,

```
>> cal_fre('10','11',4.6)

ans =

   9.7031e+04
```

Because it is the output from the generator at its base frequency, we can get from section 3.7.1 that it is correct.(The correct frequency is at around 97656.25)

A table from this program is also implemented which can be seen from section 6.1

Measuring Amplitude Below is the code for doing so:

```
function amp = cal_amp(shift,num)

switch shift
    case '0000'
        fb = 7;
    case '0001'
        fb = 8;
    case '0010'
        fb = 9;
```

```
    case '0011'
        fb = 10;
    case '0100'
        fb = 6;
    case '1000'
        fb = 5;
    case '1100'
        fb = 4;
end

bitleft = 12 - fb;
amp = (num*20*2)/(2^(bitleft+1)-1);
```

It is simply an implementation of the formula below:

$$amp = \frac{20 * 2 * n}{2^{bl}}$$

Below is an exmaple:

Figure 16: Amplitude measurement. We used default shift which is 7 bits and the wave roughly occupied 2 grids of the screen.

And if we put in the obtained data into the program, we will get:

```
>> cal_amp('0000', 1.8)

ans =

    1.1429
```

And below is the measurement from the oscilloscope in the lab:

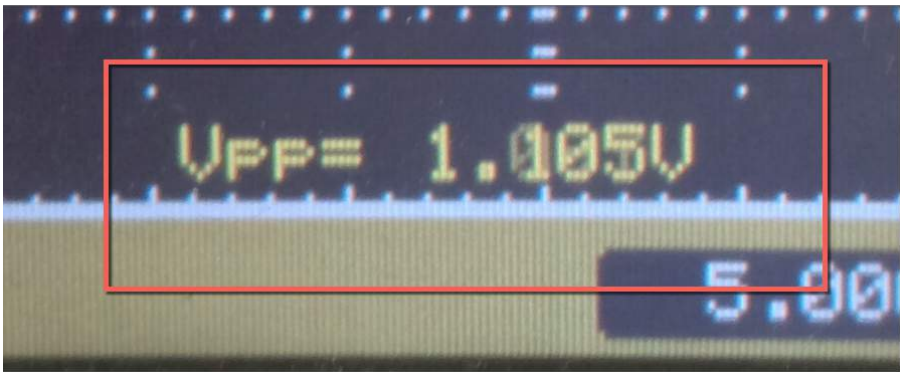

Figure 17: Amplitude real time measurement. Vpp is at around 1v.

A table from this program is also implemented which can be seen from section 6.2

3.7.3 Matlab code for setting up refresh rate

This code enable us to calculate how much delay is needed to change the refresh rate down to 50hz:

```
x = 160;
y = 120;
f = 25*10^6;
t = 1/f;

screen_time = t*x*y;
line_time = t*x;


freq = 48;
one_hz_time = 1/freq;
disp(['to draw a whole screen:',num2str(screen_time) ]);
disp(['to draw a whole line:',num2str(line_time) ]);
disp(['to delay at ',num2str(freq),'hz. 25Mhz needed(positive edge): ',num2str(↩
    one_hz_time/t),'. bit needed ',num2str(ceil(log2(one_hz_time/t)))]);
```

And it is the output:

```
to draw a whole screen:0.000768
to draw a whole line:6.4e-06
to delay at 48hz. 25Mhz needed(positive edge): 520833.3333. bit needed 19
```

3.7.4 Matlab code for calculating biased shift

Because the center of the AD data is at 01_111_111_111_111 and our screen's center is at 60. We need to balance the bias by introducing some artificial biases,

```
tk = [1];
```

```
original = repmat(tk,1,13);
display('the original center')
bi2de(original)

for i = 3:11
    now_center = bi2de(original(1:end-i));
    if now_center-60 > 0
        bit_needed=ceil(log2(now_center)+1);
        sign = 'plus';
    else
        bit_needed=ceil(log2(60- now_center));
        sign = 'negative';
    end
    display(['shift by ',num2str(i),', to bias it ',sign,' ',num2str(now_center-59)])
end
```

And it is the output:

```
shift by 3, to bias it plus 964
shift by 4, to bias it plus 452
shift by 5, to bias it plus 196
shift by 6, to bias it plus 68
shift by 7, to bias it plus 4
shift by 8, to bias it negative -28
shift by 9, to bias it negative -44
shift by 10, to bias it negative -52
shift by 11, to bias it negative -56
```

4 Module Specification

Below is the list of modules I implemented, besides from this are some Mega function modules.

4.1 VGA Module

VGA module is simply the implementation of the module in section 3.6.1.

4.2 AD-DA Module

AD module AD module is simply the module from section 3.6.2.

DA module DA module is combined in two parts, first one is the module from section 3.6.2. And then a simple counter 6.4.8 is implemented to keep recording of the value of the push button. The system will change its output wave according to this value.

4.3 State Machine Module

State machine 6.4.3 is combined in 4 parts, which can be seen from section 4.4, section 4.5, section 4.6 and section 4.7. And the implementation of controlling part is in section **??**. Besides from which, it also has a built in dual port RAM for section 4.5 to read and section 4.6 to write.

4.4 Drawing Background Module

This is an implementation 6.4.1 of section 3.1.3. And it also has a built in RAM which stores the background image. The source code is in 6.4.1

4.5 Drawing Wave Module

This is an implementation 6.4.6 of section 3.1.1 for providing X coordinate and 3.1.2 for providing RAM reading address. By adding the input value it also supports shift.

4.6 Storing AD Data Module

This is an implementation 6.4.5 of section 3.1.1 for providing RAM writing address and an implementation of section 3.4 for supporting triggering and an implementation of section 3.2.2 for section 4.3 to detects its behavior in reading cycle.

4.7 Pause Module

This is an implementation 6.4.2 of the section 3.1.1 to simply cause a delay to scale down the refresh rate at around 50hz. The counter's value is calculated by section 3.7.3.

4.8 Sampling rate switch module

This module 6.4.7 is simply to change its output frequency according to the switches value. The input frequencies are generated by PLL.

4.9 Resize module

This module 6.4.4 is simply to resize the input by doing bits shift, and the bias value can be found in section 3.7.4

5 Conclusion and potential works

For this project I learnt the advantage and constraints of the FPGA development. The advantage is the FPGA is not programming and by natural it is in parallel. But due to it is a description of the connectivity of the hardware rather than an explicit description of human login the implementation of an algorithm general is harder than a modern programming language.

Below is a list of potential works:

- It would be great if we can use a *soft core* and use C or Handel C along with Verilog HDL programming language.
- According to *quartus*, my design has used 67% of the memory and 31% of pins, total login elements consummation is at around 2%. Colorful background is not possible due to the restriction of the RAM. But we may can implement the design at a higher resolution.
- By implementation more trigger, we will be able to detect the value of the wave(i.e. frequency, amplitude) by the board itself and we can further exploit the functionality of the LCD to show the value.
- Only one route of the AD is implemented, if time allowing, it is possible to change into two route.

- By pining out the trigger to pin EXT_CLOCK, we can use external to trigger out plotting.

6 Appendix

6.1 Frequency Table

The first column is for switch 14-13 and the second one is for switch 12-11.

		1	2	3	4	5	6	7	8	9	10
10	00	84375	168750	253125	337500	421875	506250	590625	675000	759375	843750
10	01	42187.5	84375	126562.5	168750	210937.5	253125	295312.5	337500	379687.5	421875
10	10	28125	56250	84375	112500	140625	168750	196875	225000	253125	281250
10	11	21093.75	42187.5	63281.25	84375	105468.75	126562.5	147656.25	168750	189843.75	210937.5
01	00	337500	675000	1012500	1350000	1687500	2025000	2362500	2700000	3037500	3375000
01	01	168750	337500	506250	675000	843750	1012500	1181250	1350000	1518750	1687500
01	10	112500	225000	337500	450000	562500	675000	787500	900000	1012500	1125000
01	11	84375	168750	253125	337500	421875	506250	590625	675000	759375	843750
11	00	168750	337500	506250	675000	843750	1012500	1181250	1350000	1518750	1687500
11	01	84375	168750	253125	337500	421875	506250	590625	675000	759375	843750
11	10	56250	112500	168750	225000	281250	337500	393750	450000	506250	562500
11	11	42187.5	84375	126562.5	168750	210937.5	253125	295312.5	337500	379687.5	421875

6.2 Amplitude Table

switch4-1	one grid	two grid	three grid
0000	0.63492	1.2698	1.9048
0001	1.2903	2.5806	3.871
0010	2.6667	5.3333	8
0011	5.7143	11.4286	17.1429
0100	0.31496	0.62992	0.94488
1000	0.15686	0.31373	0.47059
1100	0.078278	0.15656	0.23483

6.3 Development History

Writer	Date	Commit
Yingjie	2015-04-01	Initial commit
Yingjie	2015-04-01	git ignore
Yingjie	2015-04-01	.gitignore is now working
Yingjie	2015-04-01	Error gitignore fixed
Yingjie	2015-04-02	1 bit ram success
Yingjie	2015-04-02	now it is a moving images, but it is correct, may have to do with the address_ma
Yingjie	2015-04-07	Edge from negative to positive
Yingjie	2015-04-07	Adding matlab source code for black and white image
Yingjie	2015-04-07	Now it is static image
Yingjie	2015-04-07	From manual address to mulitiplication
Yingjie	2015-04-07	Clear module removed, working on sine module

Yingjie	2015-04-07	Now this module can draw a sine wave
Yingjie	2015-04-07	add the matlab .mif generator
Yingjie	2015-04-21	Delete vga to lcd
Yingjie	2015-04-21	ADC AND VGA
Yingjie	2015-04-21	working on state machine
Yingjie	2015-04-21	Starting on state machine from vga+adc and two ms
Yingjie	2015-04-21	Removed previous display module, testing sinewave with lock
Yingjie	2015-04-21	lock success
Yingjie	2015-04-21	Suggestted LCD_VGA.bdf file, only ad, leave the display vacant
Yingjie	2015-04-21	testing state machine
Yingjie	2015-04-21	testing on transfer inputs
Yingjie	2015-04-21	A initial value bug
Yingjie	2015-04-21	bug, only last item works
Yingjie	2015-04-21	Now a configurable sine wave, sw1: a bug to be solved
Yingjie	2015-04-21	Bug fixed, sw16 to draw sine or not
Yingjie	2015-04-21	Remove bug in sine display, moving on state machine
Yingjie	2015-04-21	Now at the same time
Yingjie	2015-04-21	New image overlapping
Yingjie	2015-04-21	A static image overlapping
Yingjie	2015-04-21	static image,perfect sequence
Yingjie	2015-04-21	Now a static overlapping image with a good background
Yingjie	2015-04-22	State machine merged
Yingjie	2015-04-22	Now connected to vga, working on trigger
Yingjie	2015-04-22	blank initial screen
Yingjie	2015-04-23	temp_commit
Yingjie	2015-04-24	reverted to state machine
Yingjie	2015-04-24	background,sin_draw now is state_machine, wokring on introduce adc using fifo
Yingjie	2015-04-24	Introduce test bench using iverilog
Yingjie	2015-04-24	state machine for delay,sin_show,clean,woking on fifo
Yingjie	2015-04-24	Add mif file back
Yingjie	2015-04-24	Now switch with led on, changed LCM_VGA.bdf file
Yingjie	2015-04-24	first fifo mapper
Yingjie	2015-04-24	Added a always writing adc to the module under draw_sin, going to implement t
Yingjie	2015-04-24	Now I can see from the vga, woking on trigger and data_mapper
Yingjie	2015-04-24	Now the waveform is in the center
Yingjie	2015-04-25	Temp commit at 25 April
Yingjie Luan	2015-04-26	working on manual ram, fifo is too complicated
Yingjie Luan	2015-04-26	finished writing manually ram
Yingjie Luan	2015-04-26	Successeed
Yingjie Luan	2015-04-26	'reset' bug fixed by change the default status to do_nothing
Yingjie Luan	2015-04-26	synthesis adc clock
Yingjie Luan	2015-04-26	working on synchonzing finished signal
Yingjie Luan	2015-04-26	Sync Proved success
Yingjie Luan	2015-04-26	change background
Yingjie Luan	2015-04-26	Rename trigger module into sample time

Yingjie Luan	2015-04-26	trigger enabled, but image not still
Yingjie Luan	2015-04-26	changed a trigger design, still not working
Yingjie Luan	2015-04-26	The clock changer is wrong,now a static image
Yingjie Luan	2015-04-26	trigger with new design
Yingjie Luan	2015-04-26	Pin out state and clock, not functioning
Yingjie Luan	2015-04-26	Functioning with new trigger
Yingjie Luan	2015-04-26	Merge branch 'new_compressor'
Yingjie Luan	2015-04-26	Added a converter module
Yingjie Luan	2015-04-26	Changed default state
Yingjie Luan	2015-04-27	Update Matlab Function
Yingjie Luan	2015-04-29	3 sample clock
Yingjie Luan	2015-04-29	Implemented sample clock
Yingjie Luan	2015-04-29	Merged signal generator
Yingjie Luan	2015-04-29	Sample more data
Yingjie Luan	2015-04-29	Working on time division
Yingjie Luan	2015-04-29	time division
Yingjie Luan	2015-04-29	MOVE up and down
Yingjie Luan	2015-04-29	rescale
Yingjie Luan	2015-04-29	Multiple waves signal generator
Yingjie Luan	2015-04-29	Update wave
Yingjie Luan	2015-04-29	Signal_selector_without_test

6.4 Full Source Code

6.4.1 clear.v

```
module clear(CounterX,CounterY,clk,reset,enable,finished,color);

input clk;
input reset,enable;

output reg [7:0] CounterX,CounterY;
output finished;

wire [14:0] address;

wire CounterXmaxed = (CounterX==8'd159); // 159
wire CounterYmaxed = (CounterY==8'd119); // 119
assign finished = ((CounterXmaxed == 1) && (CounterYmaxed == 1));

assign address = CounterX + CounterY * 160;

always @(posedge clk)
begin
  if(reset == 1)
    CounterX <= 0;
  else
  begin
  if(CounterXmaxed)
    CounterX <= 0;
  else if(enable == 1)
    CounterX <= CounterX + 1;
  end
end
```

```
always @ (posedge clk)
begin
  if(reset == 1)
    CounterY <= 0;
  else
  begin
    if(CounterXmaxed == 1)
    begin
      if(CounterYmaxed)
        CounterY <= 0;
      else if(enable == 1)
        CounterY <= CounterY + 1;
    end
  end
end

// output color;
// endmodule


output [11:0] color = {12{rambuffer}};
wire rambuffer;
ram_background ram_entity(
  .address(address),
  .clock(clk),
  .wren(1'b0),
  .q(rambuffer));

endmodule
```

6.4.2 delay.v

```
module delay(clk,enable,reset,finished);

// module vga_sin(CounterX,CounterY,color,clk,enable,reset,finished);

// to draw a whole screen:0.000768
// to draw a whole line:6.4e-06
// to delay at 48hz rate:0.02083325mhz needed(positive edge)520833.3333

input clk,enable,reset;
output finished;

parameter cycle = 'd520833;

reg [19:0] counter; // log2(520833.333) ~= 19
wire countermaxed = (counter == cycle);
assign finished = countermaxed;

always @(posedge clk)
begin
  if(reset == 1)
    counter <= 0;
  else
  begin
  if(countermaxed)
      counter <= 0;
  else if(enable == 1)
      counter <= counter + 1;
  end
end

endmodule
```

6.4.3 display_state.v

```
// module display_state(CounterX,CounterY,color,clk,clk_en,rst_n);
module display_state(CounterX,CounterY,color,clk,clk_en,rst_n,
  adc_data,clk_adc,state,clk_down,vga_data,time_division,
  move_up,move_down);

input clk;
input clk_en,rst_n;//clk_en is not implemented

output reg [7:0] CounterX,CounterY;
output reg [11:0] color;

// module vga_sin(CounterX,CounterY,color,clk,enable,reset,finished);
// module clear(CounterX,CounterY,clk,reset,enable,finished,color);
wire [7:0] CounterX_clear,CounterY_clear,CounterX_sin;
wire [11:0] color_clear, color_sin;
reg enable_clear,enable_sin,reset_clear,reset_sin,enable_delay,reset_delay;
wire finished_clear,finished_sin,finished_delay;

input [2:0] move_up,move_down;

sample_clock htime_entity(
  .clk_in(clk_adc),
  .clk_out(clk_down));
// ram_flash ram_entity(
//   .data(data_flash_reg), .wraddress(wraddress), .wren(Acquiring), .wrclock(↩
    clk_flash),
//   .q(ram_output), .rdaddress(rdaddress), .rden(rden), .rdclock(clk)
// );
wire [7:0] ram_output;
input [7:0] vga_data;//= (adc_data >> 7)-4;

input [13:0] adc_data;
input clk_adc;

manual_ram ram_entity(
  .data(vga_data), .wraddress(CounterX_fill),.wren(write_enable),.wrclock(clk_down),
  .q(ram_output), .rdaddress(read_CounterX_sin),.rden(enable_sin),.rdclock(clk));

// module ramfill(clk_adc,enable,reset,finished,CounterX);

reg enable_fill,reset_fill;
wire finished_fill;
wire [10:0] CounterX_fill;
output wire clk_down;

wire write_enable;
ramfill fill_module(
  .clk_adc(clk_down),
  .enable(enable_fill),
  .reset(reset_fill),
  .clk(clk),
  .adc_data(adc_data),
  .r_finished(finished_fill),
  .write_enable(write_enable),
  .CounterX(CounterX_fill));

clear clear_module(
  .CounterX(CounterX_clear),
  .CounterY(CounterY_clear),
  .color(color_clear),
  .clk(clk),
  .enable(enable_clear),
  .reset(reset_clear),
  .finished(finished_clear));
```

```
// module vga_sin(CounterX,CounterY,color,clk,enable,reset,finished,clk_adc,adc_data);
input [1:0] time_division;
wire [10:0] read_CounterX_sin;
vga_sin sin_module(
  .CounterX(CounterX_sin),
  .color(color_sin),
  .clk(clk),
  .enable(enable_sin),
  .reset(reset_sin),
  .read_CounterX(read_CounterX_sin),
  .time_division(time_division),
  .finished(finished_sin));

delay delay_module(
  .clk(clk),
  .enable(enable_delay),
  .reset(reset_delay),
  .finished(finished_delay));
//  defparam delay_module.cycle ='d10008;

parameter [1:0] clear_screen = 2'b00, draw_line = 2'b01, do_nothing = 2'b10, fill = 2'↩
    b11;

output reg [1:0] state;
reg [1:0] next_state;

initial begin
  state = clear_screen;
end

always @ (posedge clk)
begin
  if(rst_n == 1)
    state <= clear_screen;
  else
    state <= next_state;
end

always @ * // combinational circuit
  case(state)
    clear_screen:
    begin
      enable_clear = 1;
      reset_clear = 0;

      enable_sin = 0;
      reset_sin = 1;

      enable_delay = 0;
      reset_delay = 1;

      enable_fill = 0;
      reset_fill = 1;

      CounterX = CounterX_clear;
      CounterY = CounterY_clear;
      color = color_clear;
      if(finished_clear == 0)
        next_state = clear_screen;
      else
        next_state = fill;
    end
    fill:
    begin
      enable_fill = 1;
      reset_fill = 0;
```

```
      enable_clear = 0;
      reset_clear = 1;

      enable_sin = 0;
      reset_sin = 1;

      enable_delay = 0;
      reset_delay = 1;

      CounterX = 8'bzzzz_zzz;
      CounterY = 8'bzzzz_zzz;
      color = 12'bzzz_zzz_zzz_zzz;

      if(finished_fill == 0)
        next_state = fill;
      else
        next_state = draw_line;

    end
    draw_line:
    begin
      enable_sin = 1;
      reset_sin = 0;

      enable_delay = 0;
      reset_delay = 1;

      enable_clear = 0;
      reset_clear = 1;

      enable_fill = 0;
      reset_fill = 1;

      CounterX = CounterX_sin;
      CounterY = ram_output + move_down - move_up;
      color = color_sin;
      if(finished_sin == 0)
        next_state = draw_line;
      else
        next_state = do_nothing;
    end
    do_nothing:
    begin
      enable_delay = 1;
      reset_delay = 0;

      enable_clear = 0;
      reset_clear = 1;

      enable_sin = 0;
      reset_sin = 1;

      enable_fill = 0;
      reset_fill = 1;

      CounterX = 8'bzzzz_zzz;
      CounterY = 8'bzzzz_zzz;
      color = 12'bzzz_zzz_zzz_zzz;
      if(finished_delay == 0)
        next_state = do_nothing;
      else
        next_state = clear_screen;
    end
  endcase

endmodule
```

6.4.4 modisign.v

```
module modisign(adc_data,vga_data,shift);

// horizonal resize
input [13:0] adc_data;
output reg [7:0] vga_data;

input [3:0] shift; // 0, 1, 2, 3
// matlab function provided for calculate bits real shift


// original center is 8191 {13[1]}, the center shift as well.
always @ *
begin
  case(shift)
  0:
  begin
    vga_data = (adc_data >> 7) - 4 ;
  end
  1:
  begin
    vga_data = (adc_data >> 8) + 28 ;
  end
  2:
  begin
    vga_data = (adc_data >> 9) + 44 ;
  end
  3:
  begin
    vga_data = (adc_data >> 10) + 52 ;
  end
  4:
  begin
    vga_data = (adc_data >> 6) - 68 ;
  end
  8:
  begin
    vga_data = (adc_data >> 5) - 196 ;
  end
  12:
  begin
    vga_data = (adc_data >> 4) - 452 ;
  end
  default:
  begin
    vga_data = (adc_data >> 7) - 4 ;
  end

  endcase
end
// values of data_shift :( as resolution goes lower
// shift by 7: 4
// shift by 8:

endmodule
```

6.4.5 ramfill.v

```
// module ramfill(clk_adc,enable,reset,finfished,adc_data,vga_data);
module ramfill(clk_adc,enable,reset,r_finished,adc_data,CounterX,clk,write_enable);
```

```
input clk_adc,clk;
input enable,reset;
input [13:0] adc_data;
output reg write_enable;
output reg r_finished;


wire last_bit_data = adc_data[13];
reg previous_adc_data;

// wait until trigger happen
always @ (posedge clk_adc)
begin
  previous_adc_data <= last_bit_data;
  if(enable == 1)
  begin
    if(Trigger == 1) // unchanged until enable returns to 0.
      write_enable <= 1;
  end
  else
    write_enable <= 0;
end

reg Threshold1, Threshold2;
always @(posedge clk_adc) Threshold1 <= (adc_data>=14'b10_000_000_000_000);
always @(posedge clk_adc) Threshold2 <= Threshold1;

wire Trigger = Threshold1 & ~Threshold2;  // if positive edge, trigger!


// sync clocks
reg prolong_finished = 0;
always @(posedge w_finished)
begin
  prolong_finished = prolong_finished + 1;
end

reg previous_signal;
always @(posedge clk) // flip flop
begin
  previous_signal <= prolong_finished;
  r_finished <= previous_signal ^ prolong_finished;
  // or with its previous value.
end


// build a 160 counter
output reg [10:0] CounterX;

wire CounterXmaxed = (CounterX=='d2047); // 2048 words for late down sampling
wire w_finished;
assign w_finished = (CounterXmaxed==1);


always @(posedge clk_adc)
begin
  if(reset == 1)
    CounterX <= 0;
  else
  begin
  if(CounterXmaxed)
      CounterX <= 0;
  else if(write_enable == 1)
      CounterX <= CounterX + 1;
  end
end

endmodule
```

6.4.6 vga_sin.v

```
module vga_sin(CounterX,color,clk,enable,reset,finished,read_CounterX,time_division);
// module vga_sin(CounterX,CounterY,color,clk,enable,reset,finished);
// module vga_sin(CounterX,CounterY,color,clk,enable,reset,finished,clk_adc,adc_data);


input clk;
input enable,reset;
input [1:0] time_division;
wire [2:0] read_time_division = time_division + 1; // to avoid 0, start from 1

output finished;
output reg [7:0] CounterX;
output [11:0] color;


assign color = 12'hF00; // color to be drawn

wire CounterXmaxed = (CounterX==8'd159); // 159
assign finished = (CounterXmaxed==1);

always @(posedge clk)
begin
  if(reset == 1)
    CounterX <= 0;
  else
  begin
  if(CounterXmaxed)
      CounterX <= 0;
  else if(enable == 1)
      CounterX <= CounterX + 1;
   end
end

output reg [10:0] read_CounterX;
wire read_CounterX_Maxed = (read_CounterX >= 'd2047);
always @(posedge clk)
begin
  if(reset == 1)
    read_CounterX <= 0;
  else
  begin
  if(read_CounterX_Maxed)
      read_CounterX <= 0;
  else if(enable == 1)
      read_CounterX <= read_CounterX + read_time_division;
   end
end

// endmodule

// module collect_data(clk_adc,enable,reset,finished,data_in,data_out,read_busy);
// input clk_adc;
// input [13:0] adc_data;

// output [7:0] CounterY;

// // for the moment, the fifo will read the data when the drawer is off continously ↩
    until full. The word length is not
// // correct as well, it 128 instead of 120.. Going to implmented the trigger after ↩
    testing the adc
// //module collect_data(clk_adc,clk,enable,reset,finished,data_in,data_out,read_busy)↩
    ;

// collect_data fifo_entity(
```

```
//   .clk_adc(clk_adc),
//   .clk(clk),
//   .data_in(adc_data),
//   .data_out(CounterY),
//   .read_busy(~enable)); //"when enabled it is busy to draw on the screen, usually, ↩
    enable is the opposite of reset and finifhsed

endmodule
```

6.4.7 sel_clk.v

```
module sel_clk(clk1,clk2,clk3,in_p,clk_62);

input clk1,clk2,clk3;
input [1:0] in_p;

output reg clk_62;

always @(in_p)
begin
  case(in_p)
    'd0:clk_62 = clk1;
    'd1:clk_62 = clk2;
    'd2:clk_62 = clk3;
    default: clk_62 = clk1;
  endcase
end

endmodule
```

6.4.8 signal_gen.v

```
module sign_gen(clk_dac,value,enable,reset,finished,switch);

input clk_dac;
input enable,reset;
output finished;


// 4 ram block with different initial momeory.
sin_test sin_entity(
  .address(CounterX),
  .clock(clk_dac),
  .wren(1'b0),
  .q(val1));
sin_test1 sin_entity1(
  .address(CounterX),
  .clock(clk_dac),
  .wren(1'b0),
  .q(val2));
sin_test2 sin_entity2(
  .address(CounterX),
  .clock(clk_dac),
  .wren(1'b0),
  .q(val3));
sin_test3 sin_entity3(
  .address(CounterX),
  .clock(clk_dac),
  .wren(1'b0),
  .q(val4));
```

```
sin_test1 sin_entity1(
  .address(CounterX),
  .clock(clk_dac),
  .wren(1'b0),
  .q(val2));

sin_test2 sin_entity2(
  .address(CounterX),
  .clock(clk_dac),
  .wren(1'b0),
  .q(val3));

sin_test3 sin_entity3(
  .address(CounterX),
  .clock(clk_dac),
  .wren(1'b0),
  .q(val4));



input switch;
reg [1:0] re_switch;

wire [13:0] val1,val2,val3,val4;
output reg [13:0] value;

initial
begin
  re_switch <= 'b00;
end

always @ (posedge switch)
  re_switch <= re_switch + 1'b1;

always @ *
  case(re_switch)
  'b00:
  value = val1;
  'b01:
  value = val2;
  'b10:
  value = val3;
  'b11:
  value = val4;
  endcase




reg [9:0] CounterX;
// a 1024 counter
// module vga_sin(CounterX,color,clk,enable,reset,finished,read_CounterX,time_division↩
    );
wire CounterXmaxed = (CounterX=='d1023); // samples 1024
wire finished;
assign finished = (CounterXmaxed==1);


always @(posedge clk_dac)
begin
  if(reset == 1)
    CounterX <= 0;
  else
  begin
```

```
  if(CounterXmaxed)
      CounterX <= 0;
  else if(enable == 1)
      CounterX <= CounterX + 1;
  end
end

endmodule
```

References

[1] Altera. *DE2UserManual*.

[2] ALTRA. *SCFIFO and DCFIFO IP Cores User Guide*. ALTRA, 101 Innovation Drive, San Jose, CA 95134, 12 2014.

[3] fpga4fun.com. http://www.fpga4fun.com/digitalscope_hdl3.html.

[4] University of Toronto. http://www.eecg.utoronto.ca/ jayar/ece241_07f/vga/vga-interface.html.

[5] www.terasic.com. *THDB*. terasic.