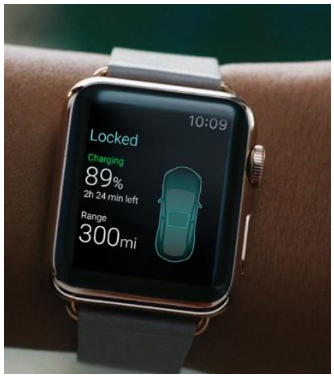
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING
UNIVERSITY OF BRITISH COLUMBIA
CPEN 391 – Computer Systems Design Studio
Fall 2015/2016 Term 2

Exercise 1.7
Adding a Graphics Controller

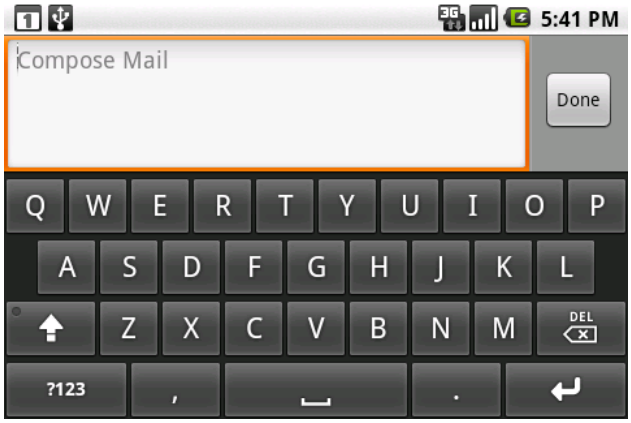
Introduction

Graphics and touchscreen input have become the new de-facto interface for small embedded computers, pretty much replacing keyboards and mice (except for precision work like drawing).

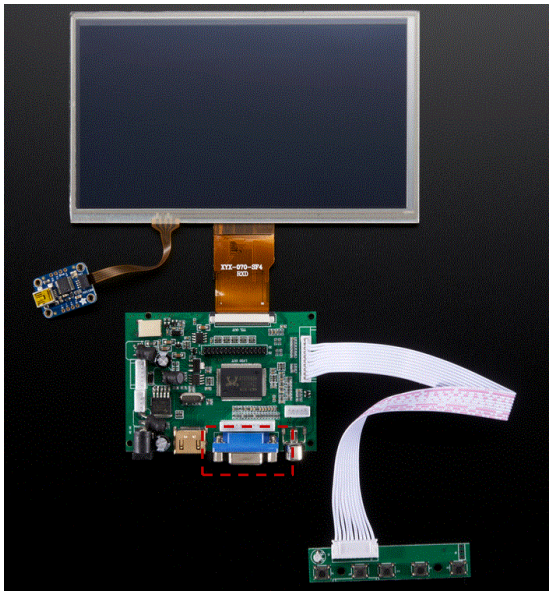
If you look around they are everywhere. Infotainment systems in cars (e.g. GPS and music control, preference settings and car status). Supermarket checkouts, bank ATMs, that's before we get onto phones, laptop, tablets and touch screen watches.



Applications that require keyboard input for data entry are emulated with virtual "pop-up" touch screen keyboards, like the one below for Android



In this exercise we are going to add a high resolution (800 x 480) graphics controller to the DE2 board. This will involve writing some VHDL code to accelerate the drawing of simple shapes such as Pixels and Lines. Using the design we can then draw shapes and text characters. The DE2 has a VGA port connector which you can use to connect to the graphics touchscreen display below (your instructor will supply you with a VGA cable for this).



Altera produce their own Graphics interface via the University Program Files for use with the DE2. However, they have made decisions about its design and chosen various tradeoffs that I don't feel are very good in this day and age. The primary concern is the resolution which is limited to 320 x 240 pixels. Similarly, the text drawing, i.e. simple characters, is limited to 1 font and a small number of pixels. This might have been acceptable 10 years ago but these days it seems a bit limited.

Perhaps more importantly for CPEN 391 and for this exercise in particular, I would like you to experience modifying a simple graphics chip to provide some hardware acceleration of drawing. That's something that's just not possible with the files produced by Altera, as they keep their VHDL/Verilog design files "under wraps" and well away from prying eyes, so we are going to use a different one that I've designed for use in a different class (CPEN 412 - Microcomputer systems design if you are interested in that as an elective next year).

Introduction to VGA and Computer Displays:

The VGA graphics standard is quite old now but was designed to display an image on a Cathode Ray Tube (CRT) monitor or (more likely these days) an LCD display as a 2 dimensional array of 640 pixels across by 480 pixels down (i.e. 640 columns of 480 rows) with each pixel having a Red, Green and Blue (RGB) colour value associated with it. For example using a 6 bit value for each pixel colour (i.e. 2 bits each for R, G and B) will give you 2^6 or 64 individual colour values to choose from for each pixel. This scheme is attractive since the colour value for each pixel could be stored within a single byte, i.e. one byte per pixel of storage required.

Alternatively a colour byte (8 bits) could be stored for each pixel. This would hold *not* the RGB value itself, but rather the 8 bit number of a colour palette - i.e. a lookup table. This 8 bit colour number would act as the index into the palette, which then produces (for example) a 24 bit colour value (8 bits of R+G+B), thus allowing 256 simultaneous displayable colours out of a possible 2^{24} colours. If the palette itself is programmable, then any of those 256 colours could be reprogrammed "on-the-fly" to change the colour of all pixels that have that stored that palette number.

Programmable palettes are useful in creating simple animation effects or making things appear/disappear instantly. For example, we could draw a filled rectangle using a palette number defined to map to the colour black (so that the rectangle doesn't appear while it is being drawn). By reprogramming that palette with a new 24 bit RGB colour the rectangle could be made to suddenly appear.

Historically, graphics controllers were designed to control a Cathode Ray Tube (CRT) screen, like the old-style Television screens, and were based on the idea that an electron gun (actually 3 guns, one red, one green and one blue, but controlled simultaneously and focused on the same spot in the

screen) could render an image on a screen by traversing the 3 gun(s) from left to right to trace out a single horizontal line of 640 pixels and then repeating this to trace out 480 separate lines as the guns move slowly from top to bottom of the screen.

To avoid flickering, this rendering of 480 lines of 640 columns per screen was typically performed 50 - 60 times (or frames) per second – referred to as the refresh rate. By controlling the amplitude of the 3 colour guns and letting their colours *mix* at the point of focus as they traverse the screen, the colour at each row/column point, i.e. each pixel, could be defined.

To maintain synchronisation between display (*which moves the guns left to right and top to bottom*) a VGA *controller* would have to generate signals for the display to direct it to return its guns to the LHS of the screen at the end of a "horizontal trace or row", and to return to the top when it had reached the bottom right hand corner or last pixel in the frame. These signals are referred to a **HSync** and **VSync** for horizontal and vertical sync, i.e. return to the left hand side of the screen and start a new horizontal line, or, return to the top of the screen and start a new vertical frame.

Today's LCD and TFT displays do not have electron guns, but nevertheless, their VGA inputs present an external electrical interface that mimics the conventional CRT monitor signals of old (*they also have more modern alternative inputs like HDMI, but that's another matter.*) That is, they take analogue R,G,B colour values (*i.e. a voltage representing the brightness of each* component of the red/green/blue colour) plus **HSync** and **VSync** signals to control the display.

To display an image, we must first build up that image in memory as a 2 dimensional array of columns and rows or [X,Y] pixel data with a colour value or palette number specified for each of those pixels. For example a 640 by 480 resolution display with 1 byte of colour information per pixel would require 307,200 bytes of memory to hold the image to be displayed – this memory is referred to as the *frame buffer*.

The VGA controller *displays* the image by accessing memory in the "frame buffer" with a row and column number (usually implemented by a pair of counters) to extract from memory the colour information for each [X,Y] coordinate or pixel on the screen, based on where the electron gun *should* be at each moment in time. It then presents it to the display (*via a digital to analogue converter or DAC*) as an analogue RGB value where it defines the colour of the pixel under the guns at that instant. That is, there is a memory location in the frame buffer that holds the colour information for every [X,Y] coordinate/pixel on the screen.

NOTE: The VGA controller does not control the position of the electron guns in the CRT, rather, the CRT display itself traverses the guns *left to right* and *top to bottom* at its own *set speed* after the HSync and VSync signals are received from the VGA controller, thus the VGA controller can, using a crystal oscillator as a timing reference figure out where the guns will *be* any time after that and provide the correct R,G,B value to display the image. The only control/synchronisation information given to the CRT is "start of a new line" (HSync) or "start a new frame" (VSync).

There is a good introduction to all this stuff at the following web address, please have a read of it before going any further. You don't need to understand the circuit presented (*but it won't hurt*), since I'm giving you the VHDL for that, you just need to understand the terminology and timing

http://faculty.lasierra.edu/~ehwang/public/mypublications/VGA%20Monitor%20Controller.pdf

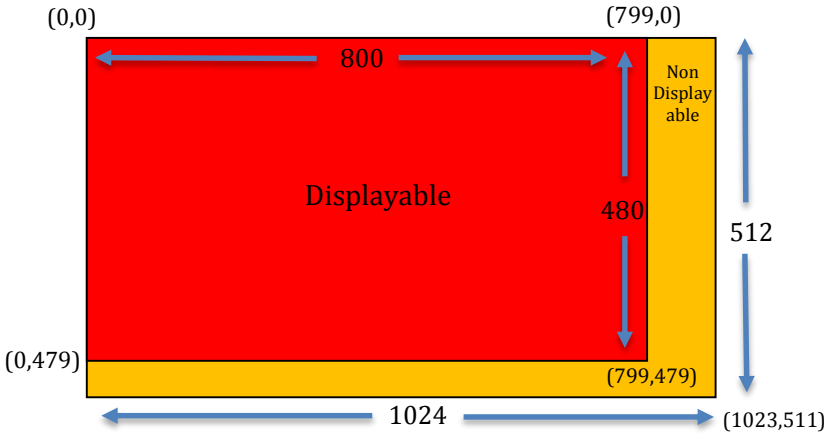
Specification:

For this assignment we are going to design a simple 800x480 pixel display with a refresh or frame rate of 50 frames per second with 1 byte of colour information per pixel. This will be presented to a programmable colour palette lookup table with 256 predefined colours (you can edit these if you like in Quartus or program them on the fly in 'C' code).

For this resolution we need at least a *10 bit unsigned counter* to keep track of the horizontal, or column pixel count (i.e. something that can count up to at least 800) and a *9 bit unsigned counter* for the vertical or row pixel count (i.e. something that can count up to at least 480).

Combining these gives us a *19 bit counter* that we can present to a memory chip as a 19 bit [X,Y] address. For this we need a memory chip of at least 2^{19} locations or 512k bytes. This works perfectly with the 512 k static ram chip on the DE2 which is organised as 256k x 16 bits thus each location in the chip will actually hold the colour for 2 pixels (*you can use the least significant bit of the column address or X coord as a means of accessing 1 or other byte held at each location*, the remaining 18 bits go to the memory chip to provide 2^{18} or 256k locations.

In theory our 512k memory holds enough information to render an image which is 1024 pixels (columns) across (2^{10}) by 512 pixels (rows) deep (2^{9}). However, because the row and column counters will not be allowed to count to their max values (they are only going to count from 0-799 and from 0-479), the memory is actually divided into displayable and non-displayable areas as shown below. Any location with a column (X coord) address > 799 or any row (Y coord) address > 479 will not have its data displayed.

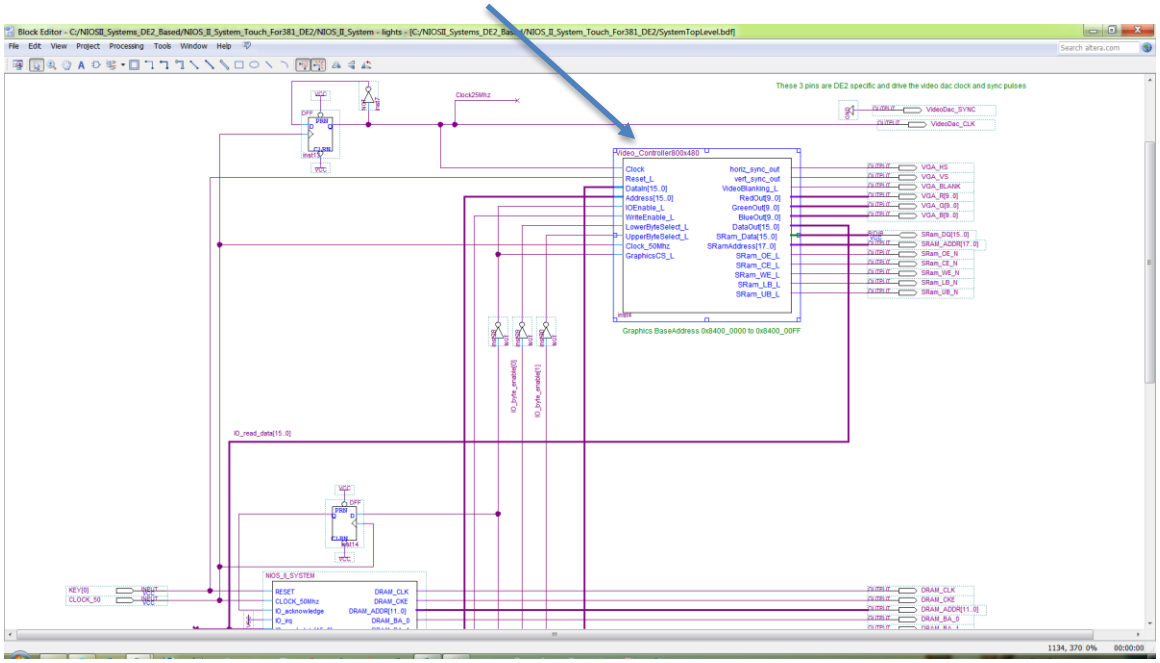


This idea of displayable and non displayable areas is important when we come to writing to the memory to build up the image we want to display. For example, to draw a horizontal line across the displayable screen at the top of our display, starting from top left i.e. coords (0,0) to (799,0) , we would write an 8 bit colour value to all memory locations in the address range 0 – 799.

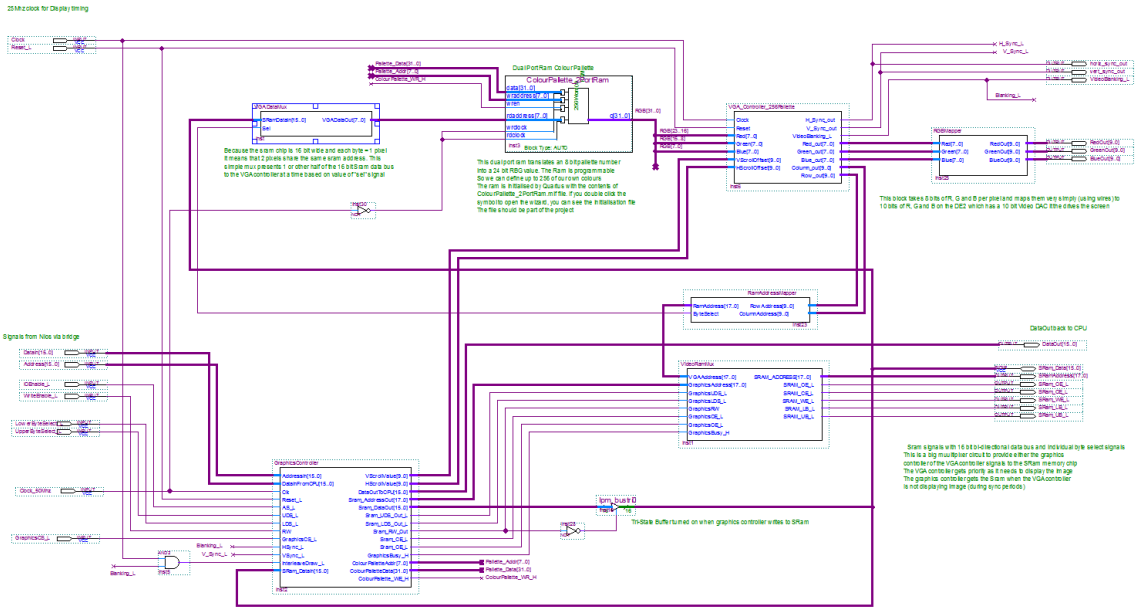
To draw another horizontal line immediately below it, we would write a colour value to all memory locations in the address range 1024 – 1823. That is *adjacent* horizontal lines are separated by 1024 pixels or memory locations (*not 800*).

Adding the graphics interface

1. On the top level of your NIOS computer you should see the following connections made to a sub-system/module called Video_Controller800x480.

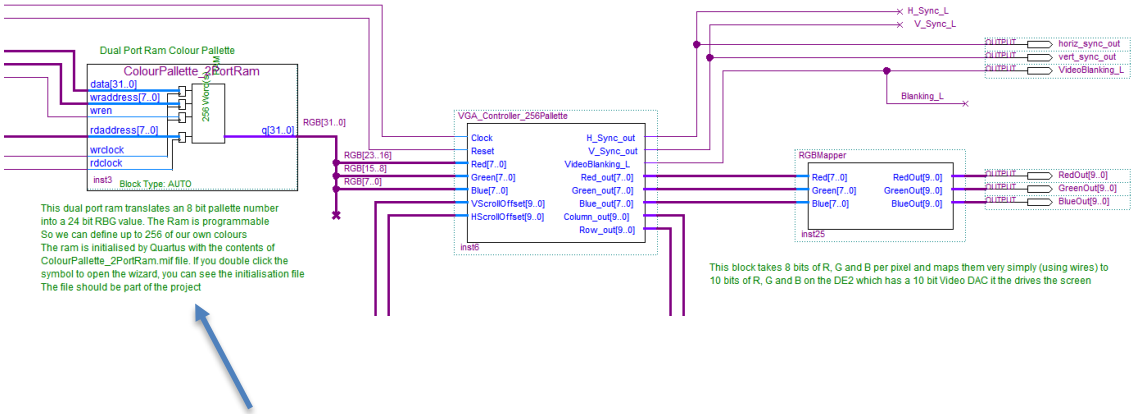


2. Double click on the above symbol to open the design file - you should see this.



The Circuit - VGA controller and Colour Palette

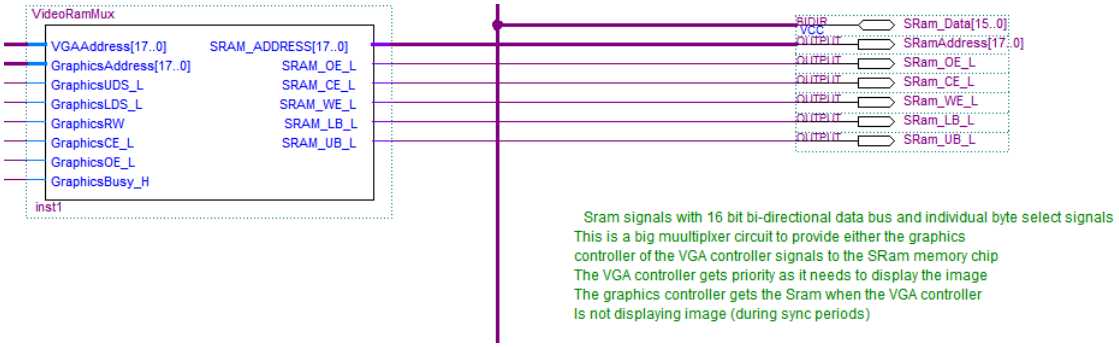
You should see something like this at the top of the circuit.



This contains a programmable colour palette (*lookup table*) which takes a byte representing a colour for each pixel from memory and turns it into 8 bits of R, G and B which is then given to the VGA controller to send to an RGB mapper (*which converts 8 bit into 10 bit values required for the video DAC on the DE2 – nothing complicated*) and eventually to a DAC chip located on the DE2 board (*not shown but connected via the red green and blue out signals*).

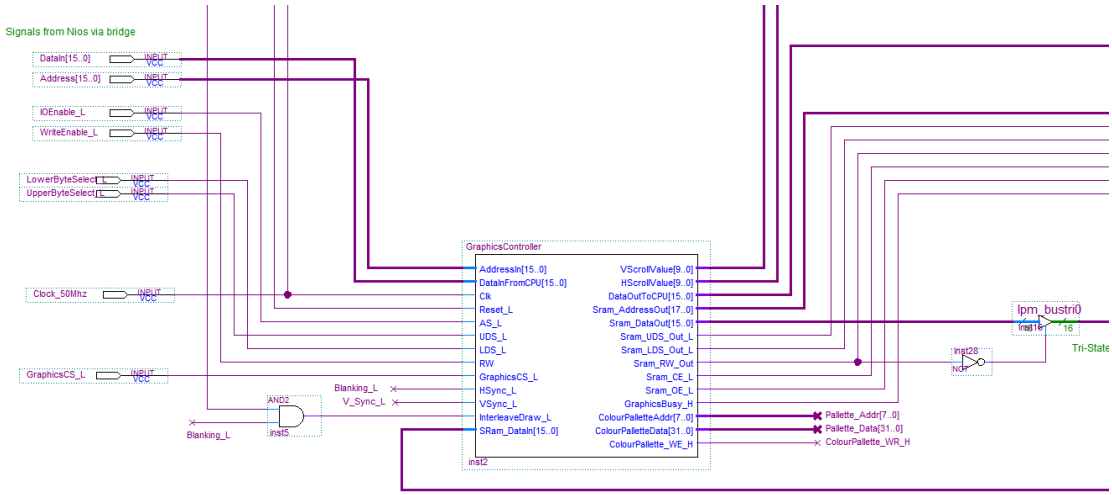
The Memory Interface

Beneath the above circuit is the interface to the memory via a 2 way multiplexer. This circuit allows either the above VGA controller or the Graphics controller (*not yet discussed*) to access the external memory chip on the DE2 (*again not shown but connected by the various Sram signals on the RHS*).



The Graphics Controller

At the bottom LHS of the circuit is the graphics controller that's of interest to us. The LHS of this circuit is connected to the NIOS processor via the Avalon to external bridge circuit (*the same bridge that we connect the serial ports to in exercise 1.3 onwards*)



To the right are signals that attempt to drive the external memory chip on the DE2 (*via the multiplexer discussed above*) and also the programmable colour palette.

Writing C code to drive the Graphics Chip

At the moment the graphics chip only implements three things in hardware

1. Write a colour value (palette number) to a pixel.
2. Read a colour value (palette number) from a pixel.
3. Program one of the 256 colour palettes with a 24 bit RGB value.

In theory any image can be built up using these 3 functions although they might not be very fast. For instance lines and text characters can be built up as a series of pixels. Rectangles and triangles as a series of lines etc. Complex shapes can be filled by searching along the edges of shape boundaries (i.e. reading pixels).

Graphics Chip Programming

Inside the graphics controller chip are a number of registers that NIOS can write to, to drive the graphics controller. These registers are listed below and expressed in 'C' as a series of **#define's** along with the addresses where they are currently mapped to in the design (based on the Avalon to external bridge).

```
// graphics registers all address begin with '8' so as to by pass data cache on NIOS

#define GraphicsCommandReg              (*(volatile unsigned short int *)(0x84000000))
#define GraphicsStatusReg               (*(volatile unsigned short int *)(0x84000000))
#define GraphicsX1Reg                   (*(volatile unsigned short int *)(0x84000002))
#define GraphicsY1Reg                   (*(volatile unsigned short int *)(0x84000004))
#define GraphicsX2Reg                   (*(volatile unsigned short int *)(0x84000006))
#define GraphicsY2Reg                   (*(volatile unsigned short int *)(0x84000008))
#define GraphicsColourReg               (*(volatile unsigned short int *)(0x8400000E))
#define GraphicsBackGroundColourReg     (*(volatile unsigned short int *)(0x84000010))

/*********************************************************************************************
** This macro pauses until the graphics chip status register indicates that it is idle
*********************************************************************************************/
```

```
#define WAIT_FOR_GRAPHICS               while((GraphicsStatusReg & 0x0001) != 0x0001);
```

Here are some software routines to drive some simple graphics functions

```
// #defined constants representing values we write to the graphics 'command' register to get
// it to draw something. You will add more values as you add hardware to the graphics chip
// Note DrawHLine, DrawVLine and DrawLine at the moment do nothing - you will modify these

#define DrawHLine               1
#define DrawVLine               2
#define DrawLine                3
#define PutAPixel               0xA
#define GetAPixel               0xB
#define ProgramPaletteColour    0x10

// defined constants representing colours pre-programmed into colour palette
// there are 256 colours but only 8 are shown below, we write these to the colour registers
//
// the header files "Colours.h" contains constants for all 256 colours
// while the course file ColourPaletteData.c contains the 24 bit RGB data
// that is pre-programmed into the palette

#define BLACK                   0
#define WHITE                   1
#define RED                     2
#define LIME                    3
#define BLUE                    4
#define YELLOW                  5
#define CYAN                    6
#define MAGENTA                 7

/*********************************************************************************************
* This function writes a single pixel to the x,y coords specified using the specified colour
* Note colour is a byte and represents a palette number (0-255) not a 24 bit RGB value
*********************************************************************************************/
void WriteAPixel(int x, int y, int Colour)
{
        WAIT_FOR_GRAPHICS;                              // is graphics ready for new command

        GraphicsX1Reg = x;                              // write coords to x1, y1
        GraphicsY1Reg = y;
        GraphicsColourReg = Colour;                     // set pixel colour
        GraphicsCommandReg = PutAPixel;                 // give graphics "write pixel" command
}

/*********************************************************************************************
* This function read a single pixel from the x,y coords specified and returns its colour
* Note returned colour is a byte and represents a palette number (0-255) not a 24 bit RGB value
*********************************************************************************************/

int ReadAPixel(int x, int y)
{
        WAIT_FOR_GRAPHICS;                      // is graphics ready for new command

        GraphicsX1Reg = x                       // write coords to x1, y1
        GraphicsY1Reg = y;
        GraphicsCommandReg = GetAPixel;         // give graphics a "get pixel" command

        WAIT_FOR_GRAPHICS;                      // is graphics done reading pixel
        return (int)(GraphicsColourReg) ;       // return the palette number (colour)
}
```

```
/**************************************************************************
** subroutine to program a hardware (graphics chip) palette number with an RGB value
** e.g. ProgramPalette(RED, 0x00FF0000) ;
**
**************************************************************************/

void ProgramPalette(int PaletteNumber, int RGB)
{
    WAIT_FOR_GRAPHICS;
    GraphicsColourReg = PaletteNumber;
    GraphicsX1Reg = RGB >> 16   ;        // program red value in ls.8 bit of X1 reg
    GraphicsY1Reg = RGB ;                // program green and blue into ls 16 bit of Y1 reg
    GraphicsCommandReg = ProgramPaletteColour; // issue command
}

/**************************************************************************************************
This function draw a horizontal line, 1 pixel at a time starting at the x,y coords specified
**************************************************************************************************/

void HLine(int x1, int y1, int length, int Colour)
{
        int i;

        for(i = x1; i < x1+length; i++ )
                WriteAPixel(i, y1, Colour);
}

/**************************************************************************************************
This function draw a vertical line, 1 pixel at a time starting at the x,y coords specified
**************************************************************************************************/

void VLine(int x1, int y1, int length, int Colour)
{
        int i;

        for(i = y1; i < y1+length; i++ )
                WriteAPixel(x1, i, Colour);
}

/**************************************************************************
** Implementation of Bresenhams line drawing algorithm
**************************************************************************/

void Line(int x1, int y1, int x2, int y2, int Colour)
{
    int x = x1;
    int y = y1;
    int dx = abs(x2 - x1);
    int dy = abs(y2 - y1);

    int s1 = sign(x2 - x1);
    int s2 = sign(y2 - y1);
    int i, temp, interchange = 0, error ;

// if x1=x2 and y1=y2 then it is a line of zero length

    if(dx == 0 && dy == 0)
        return ;

 // must be a complex line so use bresenhams algorithm
    else    {

// swap delta x and delta y depending upon slop of line

        if(dy > dx) {
            temp = dx ;
            dx = dy ;
            dy = temp ;
            interchange = 1 ;
        }

// initialise the error term to compensate for non-zero intercept

        error = (dy << 1) - dx ;    // (2 * dy) - dx
```

```
// main loop
        for(i = 1; i <= dx; i++)    {
            WriteAPixel(x, y, Colour);

            while(error >= 0)   {
                if(interchange == 1)
                    x += s1 ;
                else
                    y += s2 ;

                error -= (dx << 1) ;    // times 2
            }

            if(interchange == 1)
                y += s2 ;
            else
                x += s1 ;

            error += (dy << 1) ;    // times 2
        }
    }
}
```

Inside the Graphics Chip

Although we can draw lines in software (*using the functions above and the write pixel hardware of the graphics chip*), we want to accelerate the drawing of these things replacing the software routines with hardware.

Here for instance is the VHDL code to write a pixel to the Frame buffer memory. You can locate this near the bottom of the Graphics Controller VHDL file. The graphics controller is based around a state machine and one of those states is DrawPixel which is entered when NIOS writes a PutAPixel command (*hex 0A*) into the graphics chip's command register (*see above*). It takes the value of the colour, x1 and y1 registers and writes the colour to the memory chip at the x1,y1 coords

```
-------------------------------------------------------------------------------------------------------------------------------------------
            elsif(CurrentState = DrawPixel) then
-------------------------------------------------------------------------------------------------------------------------------------------
-- This state is responsible for drawing a pixel to an X,Y coordinate on the screen
-- Your program/NIOS should have written the colour palette number to the "Colour" register
-- and the coords to the x1 and y1 register before writing to the command register with a "Draw pixel" command)

-- first we have to wait until it safe to draw to the memory
-- (i.e. wait for a horizontal sync from the display controller that indicates that it is not displaying anything)

                    if(OKToDraw_L = '0') then

-- if VGA is not displaying at this time, then we can draw, otherwise wait for video blanking during Hsync
-- the address of the pixel is formed from the 9 bit y coord that indicates a "row" (1 out of a maximum of 512 rows)
-- coupled with a 9 bit 'X' or column address within that row. Note a 9 bit X address is used for a maximum of 1024
-- columns or horizontal pixels. You might thing that 10 bits would be required for 1024 columns and you would be
-- correct, except that the address we are issuing holds two pixels (the memory chip is 16 bit wide remember so
-- each location/address we issue corresponds to that of 2 pixels)

-- issue 9 bit x address even though it goes up to 1024 which would mean 10 bits, because each address = 2 pixels/bytes
                        Sig_AddressOut      <= Y1(8 downto 0) & X1(9 downto 1);

-- we are intending to draw a pixel so set memory chip RW to '0' for a write to memory
                        Sig_RW_Out          <= '0';
                        if(X1(0) = '0')     then    -- if the address/pixel is an even numbered one

-- enable write to upper half of Sram memory chip's data bus to access 1 pixel at that location
                                            Sig_UDS_Out_L       <= '0';
                        else

-- else write to lower half of Sram memory chip data bus to get the other pixel at that address
                                            Sig_LDS_Out_L       <= '0';

                        end if;
```

```
-- the data (colour) that we write comes from the default value assigned to Sig_DataOut previously
-- you will recall that this is the value of the Colour register

                                NextState <= IDLE;              -- return to idle state after writing pixel

                        else
-- otherwise stay here until we manage to write the pixel (during an HSync period)
                                NextState <= DrawPixel;
                        end if ;
```

How does this work

The signal Sig_AddressOut will be presented to the memory chip to specify what location in the memory chip we are accessing. It is formed as an 18 bit address constructed from the 9 bit row (y coord) and 9 bit column (x coord) values i.e. y1 + x1 values using VHDL's concatentation operator '&' i.e. Y1(8 downto 0) & X1(9 downto 1);

That is, the 18 bit address presented to memory is YYYYYYYYYYXXXXXXXXXX which identifies a row and a column on the screen. Note that the 18 bit address specifies the contents of a 16 bit word (the memory chip on the DE2 is a '**x16**' chip, so each address is actually the address of 2 pixels. We need to resolve that to 1 pixel - this is explained next.

The Signals 'Sig_UDS_L' and 'Sig_LDS_L' are data strobes (*think of them as chip enables*) for the upper and lower bytes of data stored within each memory location. To see how this works, imagine a 16 bit wide location in memory containing two 8 bit pixel values.

The way the circuit has been designed is that the left most pixel has been designed to be stored in the upper byte of the 16 bit memory. The right most pixel is stored in the lower byte of the 16 bit memory. Given that a row always begins with an 'X' coord of 0, an even numbered pixel address (such as 0, 2, 4 etc.) can be accessed via the upper byte of data within each memory location while an odd numbered pixel address (such 1,3,5 etc.) is accessed via the lower byte of memory - this is important.

This following code effectively decides which half of the memory chip to access (i.e. which pixel) based on the least significant bit of the 'X' coordinate. If the least significant bit of register X1 is '0' i.e. an even pixel address, then activate 'Sig_UDS_L' so that the value of the colour register is written to the upper byte of the memory location, otherwise (if the pixel address is odd), activate 'Sig_LDS_L' to write to the lower byte of the memory location

```
if(X1(0) = '0')     then
        -- if the address/pixel is an even numbered one
        Sig_UDS_Out_L   <= '0';   -- enable write to upper half of Sram
else
        Sig_LDS_Out_L   <= '0';   -- else write to lower half of Sram
end if;
```

The following code states that we should drive 'Sig_RW_Out' to '0' meaning that the memory chip will perform a store of the data that we are presenting to it. The *default* value for this signal is '1' meaning that unless we state otherwise, we will present a "read" signal to the memory (*which is the safe thing to do*).

```
Sig_RW_Out                      <= '0';
```

What data gets written?

If you look further up in the VHDL file for the graphics controller you will see this "*default*" value that is always presented to the memory chip (unless we override it in our VHDL). This effectively states that the data we "present" to both halves (bytes/pixels) of the memory chip is the value of the colour register (*which we will have programmed in C before issuing the write pixel command).*

```
Sig_DataOut       <= Colour(7 downto 0) & Colour(7 downto 0);
```

Even though we present the contents of the Graphics chip colour register to both bytes of the memory chip, it does *not* mean that both pixels at the same location will be updated - that depends upon 'Sig_UDS_Out_L 'and 'Sig_LDS_Out_L' which in turn depends upon the 'x' coordinate of the pixel we are trying to write to.

In theory, if we modify our graphics chip to draw a *horizontal line*, we might be able to design it so that it updates both pixels at a time (thus doubling the drawing speed), but we'll have to take care to consider start and end coords for the line - always writing both pixels could mean we start the line 1 pixel before we should and perhaps end it 1 pixel beyond where it should stop. To keep things simple, you could just write 1 pixel at a time.

How does the Graphics chip decide when to write to memory?

When the graphics chip wants to access the memory, it should always be aware of the fact that the memory may be in use by the VGA display controller. After all, the image has to be drawn 50-60 times per second with only small periods of time - during the horizontal sync periods when it is safe to take away the memory chip from the VGA display controller and give it to the graphics controller.

The following VHDL test checks the status of the 'OKToDraw_L' signal. If it is '0' it means the graphics chip has the memory and can read/write from/to it. This is something the graphics controller must *ALWAYS* check with every pixel it reads or write. If this signal is '1' then the graphics controller does **NOT** have the memory and must wait (meaning it must stay in the same 'state'). Only when it is '0' can we present signals (in particular the RW signal) to the memory.

```
-- if VGA display controller is not displaying at this time, then we can draw, otherwise wait for HSync

        if(OKToDraw_L = '0') then
```

Check out the full code in the VHDL for the graphics chips

OK I understand - what do I have to do?

For this exercise you should implement VHDL code in the graphics controller to draw horizontal and vertical lines and draw a line from anywhere to anywhere. A place holder already exists for you to write your code - see below. Remember the line drawing algorithm you did in 353 using Bresenhams algorithm. The software (C code) version of this algorithm is given above.

```
-------------------------------------------------------------------------------------------------------------
                elsif(CurrentState = DrawHline) then
-------------------------------------------------------------------------------------------------------------
                        NextState <= IDLE;

-------------------------------------------------------------------------------------------------------------
                elsif(CurrentState = DrawVline) then
-------------------------------------------------------------------------------------------------------------
                        NextState <= IDLE;

-------------------------------------------------------------------------------------------------------------
                elsif(CurrentState = DrawLine) then
-------------------------------------------------------------------------------------------------------------
                        NextState <= IDLE;

                end if ;
```

At the moment the default behavior of this code is to do nothing. So if you write commands such as DrawHLine (*see #define above*) to the Graphics controller command register, it will move to the above 'DrawHline' state - do nothing and immediately return to the IDLE state. Your job is implement the above states. Obviously when drawing a line we now use X1,Y1, X2 and Y2 registers.

Think about how to draw a line maybe you always assume that x1 is less than x2 and ditto for y1 and y2. If you assume that, then your C code will have to check and perhaps swaps the coords over before writing to them to the graphics chip if they are the wrong way around (e.g. x1 > x2).

How will you keep track of which pixel to write? When drawing a horizontal line it's easy. Perhaps the easiest way is to increment the x1 register in the graphics chip after each pixel write to memory and stop when it's value equals x2. For a vertical line you could add 1024 to Y1 and keep going until it equals Y2. *Think about what pixel you write - just as in C when accessing arrays, it's easy to access one too many or one two few elements. In this case it's easy to draw one too many or one two few pixels.*

Both of the above approaches will involve adding some extra signals and code to the graphics controller, e.g. a signal to increment 'x1' which can be driven after you write a pixel. The VHDL for the x1 register will have to be modified to take account of this possibly new 'increment' signal and add 1 to the value of 'x1', ditto add 1024 to y1 when drawing a vertical line

Think about a horizontal line of 0 length e.g. when 'x1' starts off being the same value as 'x2'. What do you do? Do you draw a pixel or not? What is correct and do you do it?

Think about negative X/Y coords - deal with those in C code before you even give them to the graphics chip (which should use only positive coords.)

What about the complex Bresenhams line drawing algorithm?

For the line drawing, you will probably need some temporary 'registers' that can store intermediate results between clock edges and states (*you probably did something similar to this in ECE 353*) and some signals to control those registers. For example if you look at the C code implementation of the Bresenhams algorithm above, you'll see it contains some temporary variables e.g. x, y, dx, dy, error etc.

Here is an example of how to create a 16 bit variable called 'x' and how to load it and get it to remember a value. We can do this kind of thing for as many registers/variables as we like.

```
-- declare some signals for the 16 bit variables/register

Signal    X              : std_logic_vector(15 downto 0) ;     -- the variable 'x'
Signal    X_Data         : std_Logic_Vector(15 downto 0);      -- a signal carrying data to be stored in 'x'
Signal    X_Load_H       : std_logic;                          -- a signal to store/update x
```

IMPORTANT

Make sure we have a ***default value*** for the load and data signals somewhere inside the state machine, otherwise memory will be created for it.

```
X_Load_H        <= '0';          -- to have a default value for all signals to avoid inferring
X_Data          <= X"0000";      -- latches/storage.

-------------------------------------------------------------------------------------------------------------------
-- Here is an Process we could write to hold 'x'. The data is stored on a rising clock edge
-------------------------------------------------------------------------------------------------------------------
process(Clk)
Begin
        if(rising_edge(Clk)) then
                if(X_Load_H = '1') then        -- if told to store data by state machine
                        X <= X_Data;           -- copy X_Data to X on next rising edge and remember it
                end if;
        end if;
end process;
```

Inside the state machine we could write something like this where we update the variable error (a register) with the value of the **C** expression **error = error - (x1 << 1) ;** which is part of Bresenhams algorithm). Assume variables **error** and **x1** are **16 bit values** stored in VHDL defined registers.

```
-------------------------------------------------------------------------------------------------------------------------------
        elsif(CurrentState = DrawLine) then
-------------------------------------------------------------------------------------------------------------------------------
                error_Data      <= error - (dx(14 downto 0) & '0');     -- error -= dx * 2
                error_Load_H    <= '1' ;                                -- load the result
```

We just need to break the C code algorithm into a number of states in VHDL. Remember, the values in registers are only updated on the **next edge** of the clock, so if you want to assign a value to a register and then use it later, you have to split that up into at least 2 states, 1 state to set up the data and load signals for the register then wait for the clock to update the register before moving to a new state where by you can use the new result. Don't try to load a value and use the new value in the same state.

Here for example is the start of a VHDL implementation of the Draw line state (Bresenhams algorithm) where we initialise the variables. The C code is given below.

'C' Code Implementation

```
/**************************************************************************
** Implementation of Bresenhams line drawing algorithm
**************************************************************************/

void Line(int x1, int y1, int x2, int y2, int Colour)
{
    int x = x1;
    int y = y1;
    int dx = abs(x2 - x1);
    int dy = abs(y2 - y1);

    int s1 = sign(x2 - x1);
    int s2 = sign(y2 - y1);
    int i, temp, interchange = 0, error ;
```

VHDL Code Implementation

- Assume x, y, dx, dy, s1 and s2 are 16 bit std_logic_vector VHDL signals
- Assume x2Minusx1 and y2Minusy1 are 16 bit std_logic_vector VHDL variables

```
-------------------------------------------------------------------------------------------------------------------------------
                elsif(CurrentState = DrawLine) then
-------------------------------------------------------------------------------------------------------------------------------
                        x_Data <= x1;
                        y_Data <= y1;

                        x2Minusx1 := x2 - x1;
                        y2Minusy1 := y2 - y1;

                        dx_Data <= abs(signed(x2Minusx1));      -- assign immediately, no storage
                        dy_Data <= abs(signed(y2Minusy1));      -- assign immediately, no storage

-- calculate s1= sign(x2 - x1)

                        if(x2Minusx1 < 0) then
                                s1_Data <= X"FFFF";             -- s1 = -1 (in 2's complement)
                        elsif (x2Minusx1 = 0) then
                                s1_Data <= X"0000";             -- s1 = 0
                        else
                                s1_Data <= X"0001";             -- s1 = 1
                        end if;
```

```
-- calculate s2= sign(y2 - y1)

                    if(y2Minusy1 < 0) then
                        s2_Data <= X"FFFF";                 -- s1 = -1
                    elsif (y2Minusy1 = 0) then
                        s2_Data <= X"0000";                 -- s1 = 0
                    else
                        s2_Data <= X"0001";                 -- s1 = 1
                    end if;

                    interchange_Data <= X"0000";

                    x_Load_H <= '1';
                    y_Load_H <= '1';
                    dx_Load_H <= '1';
                    dy_Load_H <= '1';
                    s1_Load_H <= '1';
                    s2_Load_H <= '1';
                    interchange_Load_H <= '1';

-- need a new state after here to allow storage of results to take place on next clock

                    NextState <= DrawLine1;
```

Displaying Text

We are not going to try to accelerate the drawing of characters (we'll do that and other shapes to courses like CPEN 412). For the moment we'll use a purely software ('C' code) approach by drawing characters out of software fonts, 1 pixel at a time.

Here is a part example of a simple 5x7 pixel font created out of arrays of characters. This and other fonts can be found in the file "Fonts.c" on connect. The larger ones in that file were created using a free online font creation program called The Dot Factory (see http://www.eran.io/the-dot-factory-an-lcd-font-and-image-generator) which creates a bit mapped font file from any installed Windows font.

```
const unsigned char Font5x7[95][7] = {
            {0x0,0x0,0x0,0x0,0x0,0x0,0x0},          // ' '
            {0x4,0x4,0x4,0x4,0x0,0x0,0x4},          // '!'
            {0xa,0xa,0xa,0x0,0x0,0x0,0x0},          // '"'
            {0xa,0xa,0x1f,0xa,0x1f,0xa,0xa},        // '#'
            {0x4,0xf,0x14,0xe,0x5,0x1e,0x4},        // '$'
            {0x18,0x19,0x2,0x4,0x8,0x13,0x13},      // '%'
            {0xc,0x12,0x14,0x8,0x15,0x12,0xd},      // '&'
            {0xc,0x4,0x8,0x0,0x0,0x0,0x0},          // '''
            {0x1,0x2,0x4,0x4,0x4,0x2,0x1},          // '('

            // etc etc.

 };
```

In the example above the character '#' is implemented as 5 pixels across by 7 rows down. The interpretation of the values : {0xa,0xa,0x1f,0xa,0x1f,0xa,0xa} are clear to see when you write them on in binary and line them up vertically as shown below:-

```
00001010        // hex 0a
00001010        // hex 0a
00011111        // hex 1f
00001010        // hex 0a
00011111        // hex 1f
00001010        // hex 0a
00001010        // hex 0a
```

Of course they are stored as bytes (8 bits) but we only display the least significant 5 bits. Here is a function to do that that uses only the pixel write function

```
/*************************************************************************************************
** This function draws a single ASCII character at the coord and colour specified
** it optionally ERASES the background colour pixels to the background colour
** This means you can use this to erase characters
**
** e.g. writing a space character with Erase set to true will set all pixels in the
** character to the background colour
**
*************************************************************************************************/

void OutGraphicsCharFont1(int x, int y, int fontcolour, int backgroundcolour, int c, int Erase)
{
// using register variables (as opposed to stack based ones) may make execution faster
// depends on compiler and CPU

   register int row, column, theX = x, theY = y ;
   register int pixels ;
   register char theColour = fontcolour  ;
   register int BitMask, theC = c ;

// if x,y coord off edge of screen don't bother
// XRES and YRES are #defined to be 800 and 480 respectively
    if(((short)(x) > (short)(XRES-1)) || ((short)(y) > (short)(YRES-1)))
        return ;


// if printable character subtract hex 20
   if(((short)(theC) >= (short)(' ')) && ((short)(theC) <= (short)('~'))) {
      theC = theC - 0x20 ;

      for(row = 0; (char)(row) < (char)(7); row ++)      {

// get the bit pattern for row 0 of the character from the software font
         pixels = Font5x7[theC][row] ;
         BitMask = 16 ;

         for(column = 0; (char)(column) < (char)(5); column ++)  {

// if a pixel in the character display it
            if((pixels & BitMask))
               WriteAPixel(theX+column, theY+row, theColour) ;

            else {
               if(Erase == TRUE)

// if pixel is part of background (not part of character)
// erase the background to value of variable BackGroundColour

                     WriteAPixel(theX+column, theY+row, backgroundcolour) ;
            }
            BitMask = BitMask >> 1 ;
         }
      }
   }
}
```

The above function can serve as a template for writing other functions to display other fonts.

What do I have to demonstrate?

1. Demonstrate hardware acceleration by writing a function to draw lots of horizontal and vertical and Bresenham lines at random coordinates and in random colours. The more of these you implement, the more marks you will get for this exercise, obviously Bresenham is more difficult, so will gain more marks.

2. Build a small library of functions to draw things line '*rectangles*,' '*filled rectangles*', 'filled rectangles with a border', 'triangles' etc. There is also a *'circle'* and *'arc'* drawing algorithm that Bresenham came up with, see if you can dig that out on the web and implement it. (Check out http://en.wikipedia.org/wiki/Midpoint_circle_algorithm). The circle algorithm is based on drawing a set of 8 x 45 degree arcs in opposite quadrants and is pretty easy to implement so why not add 'arc' and 'circle' drawing functions to your library.

3. Demonstrate the display of text strings on the screen.

4. Of course if you don't get the hardware acceleration to work, then you can always fall back on the software implementation based on plotting lines as a series of pixels but that won't get you so many marks!!

To think about for the project.

Although not necessary for this exercise, Project 1 involves some kind of touch screen user interface, so you might like to think about this while completing this exercise. Things you might like to consider:-

- Have the ability to draw a button in a specified colour with a text string inside starting at any [x,y] coordinate on the screen.
- Have the ability to detect when the user presses the button (*and releases it*) and provide some feedback to the user that the button has been selected e.g. the colour of the button could change or perhaps it's size.
- Develop a generic menu function which can be passed an array of strings. The function draws menu buttons one per string, waits for the user to select/press one and then return the integer number/index of the button.
- Perhaps add a **virtual keyboard** that you call to allow the user to enter some text. When the enter/return button is pressed it returns the string entered and the keyboard disappears.
- Perhaps add some kind of pop up **dialog box** for things like "Do you really want to do 'x'" and have a yes/no button response.
- There's lots of ideas you could come up with at this point to dictate the look and feel (i.e. the design) of a user interface.