UNIVERSITY OF SOUTHAMPTON

Vector Parallel architecture for graphics acceleration in FPGA

by

Karthik Sathyanarayanan

A thesis submitted in partial fulfillment for the degree of MSc

in the
Faculty of Engineering, Science and Mathematics
School of Electronics and Computer Science

December 2020

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF ENGINEERING, SCIENCE AND MATHEMATICS
SCHOOL OF ELECTRONICS AND COMPUTER SCIENCE

Master of Science

by Karthik Sathyanarayanan

This project designs a ARM System on Chip which displays a jagged Christmas tree using both, just the included processor of the system on chip as well as a dedicated hardware accelerator module which accelerates the rasterisation of triangles. This is displayed on a 640x480 VGA screen connected to an Intel DE-1 FPGA

Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank Iain McNally for his guidance and advice throughout the course of the project.

Chapter 1

Introduction

According to Jim Smith “The most efficient way to execute a vectorizable application is a vector processor” as mentioned in the book Computer Architecture A Quantitative Approach Fourth Edition [3]. Since the early 1970s, supercomputers can take advantage of certain common characteristics of computational algorithms in application specific programs. In many large scale programs, such as graphics processing, significant computation time is spent on arithmetic operations that are performed on many operands which are usually in the form of multi dimensional array. These programs require identical operations to be performed over many operands as mentioned by K Miura’s lectures [9]. This observation has led to many innovations such as pipelining techniques and parallelism in computer architecture. Vector parallel architecture has considerable potential and that computer architects can use to increase performance of processors without significantly increasing the energy demands and design complexity. Vector parallel architecture has not been tried on in an application for rasterisation of triangles on an FPGA in previous student projects. Since the operation of rasterising triangle is inherently a process involving lots of arithmetic operations performed on multiple operands, it is a good candidate to test a vector parallel architecture. This project is done by using a soft core ARM SoC based on a Cortex M0 processor with AHB-lite interface which would be modified to incorporate application specific AHB-lite slaves in an Intel DE-1 FPGA. Systemverilog was used to make the hardware designs and all the designs are simulated in Cadence NCSim and synthesised using Intel Quartus. This project aimed at making a graphics accelerator using ARM SoC on a FPGA. This project also studies a vector parallel approach in improving the performance of a graphics accelerator .This system uses an AHB-LITE interface as described in Figure 1.1:

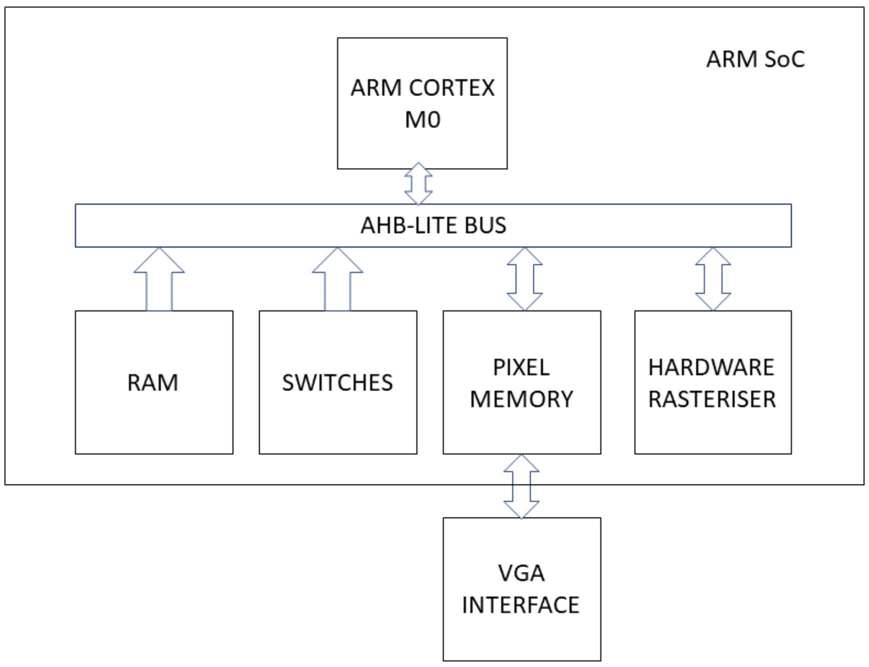


FIGURE 1.1: Overview of System

An ARM Cortex M0 processor is used as the central processing unit for the system. It uses a simple AHB-LITE bus that acts as an interface between the Cortex M0 a third party processor which is part of ARM Cortex M0 Design start which acts as an AHB master and all the other units in the ARM-SoC which act as AHB slaves. The blocks mentioned in the Figure 1.1 as RAM and SWITCHES are part of custom third party modules taken and adapted for the project from Iain Mcnally's course [6]. The PIXEL MEMORY and HARDWARE RASTERISER are application specific AHB-LITE slave designed for this project. A VGA INTERFACE was also included whose design is an adapted from James Hamblen's work on Razzle [7] which will be discussed in further sections.

Chapter 2

Background research

2.1 Arm System on chip

A system on chip usually comprises of ARM M0 design start processor core, AHB-LITE BUS interconnect, Data memory (RAM), fixed program memory (ROM) and application specific interfaces. It also includes an application specific software written in C. The AHB-LITE interconnect is illustrated in Figure 2.1. The individual slaves(ROM, RAM, Keyboard interface, Display interface) are selected via higher order bits of HADDR, while the lower order bits are used to select the individual registers inside the slaves. The signal HWDATA, broadcasts Write data to all the slaves. while the multiplexer selects and reads data from only one slave at a time via HRDATA. The address decoder controls the multiplexer for HRDATA via a register adding a single cycle delay.

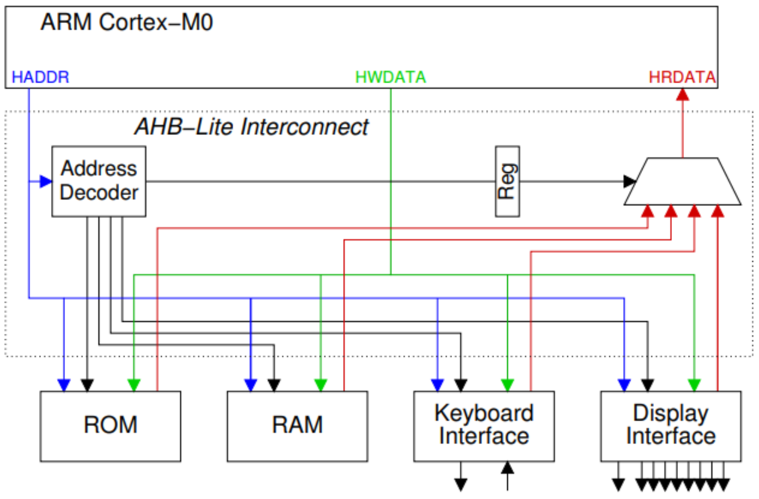


FIGURE 2.1: ARM System on a chip, Source(Iain Mcnally) [5]

. There are basically 4 components of the AHB-LITE interconnect system :

- Master
- Slaves/peripherals
- Address Decoder
- Multiplexor

The Address Decoder and the Multiplexor form part of the AHB-LITE interconnect bus. The AHB-Master illustrated in Figure 2.4 (in our case the Cortex M0) executes 16 bit instructions from ARM Thumb2 instruction set. This processor is capable of handling 32 bit wide data. The processor core is capable of being programmed using simple C code. The core is ARM v6 M architecture which is based on Von Neumann architecture.

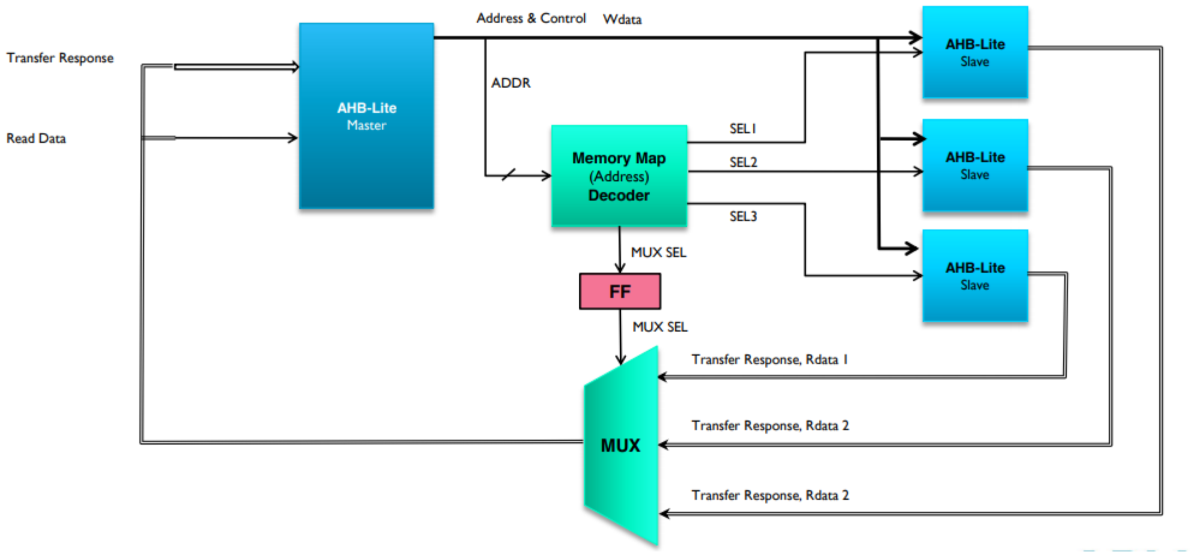


FIGURE 2.2: AHB-LITE mux and decoder, Source(Iain Mcnally) [5]

The AHB-Slave as illustrated in Figure 2.3 are addressed via the AHB-interconnect bus and the flow of data takes place via Write Data from the master at various addresses as dictated by Address and control ports.

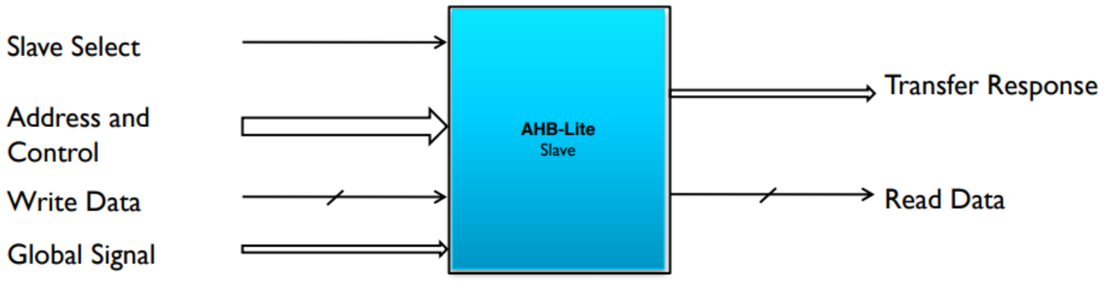


FIGURE 2.3: AHB-LITE slave, Source(Iain Mcnally) [5]

The address and control ports are dictated by the decoder and multiplexor as shown in Figure 2.2. Individual slave select signals are generated from the high order bits of `HADDR`. The low order bits of `HADDR` are used to select registers within the slaves.

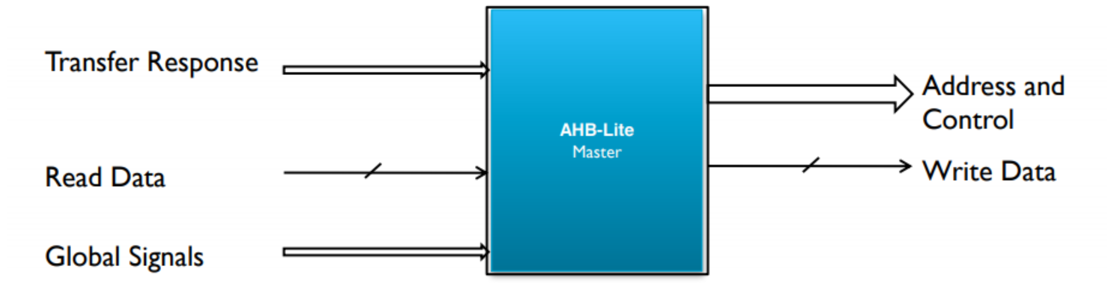


FIGURE 2.4: AHB-LITE master, Source(Iain Mcnally) [5]

The slaves can be categorised into 2 parts, custom third party modules and application specific module. The custom third party ones are as follows:

- **RAM** : This acts as a program memory in case of FPGA which is synthesised as a synchronous RAM and reads a hex file which dictates what instructions are to be run by the Cortex M0. These have byte select functions which assigns `HWDATA` and `HRDATA` based on `HADDR` and `HSIZE`. The RAM memory takes 2 clock cycles to write and a single clock cycle to read.
- **SWITCHES** : This is an AHB Slave that acts as an interface for the Switches and buttons of the FPGA board.

The following sections will cover the James Hamblen's work on VGA interface which connects to the VGA screen.

2.2 VGA Interface

This section shows the working of a VGA interface by describing James Hamblen's work of Razzle which shows a fractal type image [7] as shown in Figure 2.5

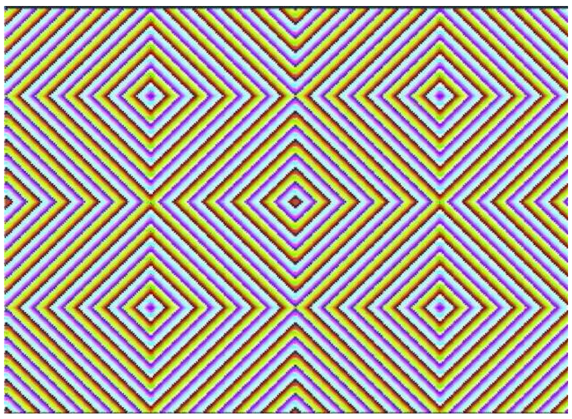


FIGURE 2.5: Razzle ,Source (James Hamblen) [8]

A VGA screen is divided into 640 x 480 pixels. The VGA interface has 2 active signals `HSync` and `VSync` which are used for video synchronisation. There are also 3 analogue signals used to control the red, blue and green values of a pixel. The interface draws screen at least 60 times a second. When the `HSync` and the `VSync` signals are high, the screen starts painting the pixels horizontally as shown in Figure 2.6 . Once the 640th pixel has been reached, it starts painting the pixel value for the 2nd row of the screen and so on. The counting of the pixels in horizontal and vertical are dictated by `Hcount` and `Vcount` .The controller draws each pixel one by one from left to right, from top to bottom. When the `Hcount` counts until 659, the `HSync` signal will be triggered to inform the monitor to start another horizontal line. There will be a horizontal blanking interval where a front porch and a back porch inserted before and after the `HSync` signal respectively. This is called blanking interval. This is done similarly when `Vcount` counts till 480 after a row of pixels have been counted and then triggers `VSync` signal . There is the same blanking interval similar to the one in `HSync` during `VSync` .

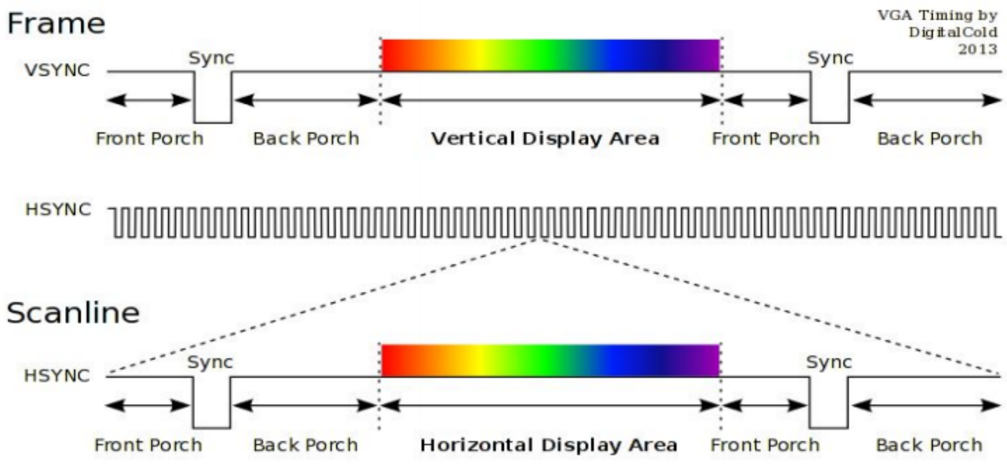


FIGURE 2.6: VGA timing diagram, Source(Knebel C et al.) [10]

2.3 Barycentric Algorithm

A barycentric coordinate system is a coordinate system where location of every point can be denoted as an affine transformation of triangle vertices. This property is being exploited to denote the pixels which are inside and outside the triangle. This coordinate system helps us identify position of P in terms of a, b, c as illustrated in Figure 2.7 . Point P can be expressed as:

$$P = \lambda_1 a + \lambda_2(b - a) + \lambda_2(c - a)$$

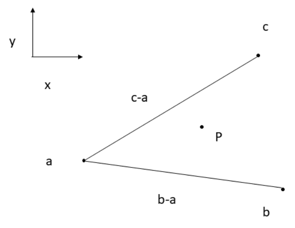
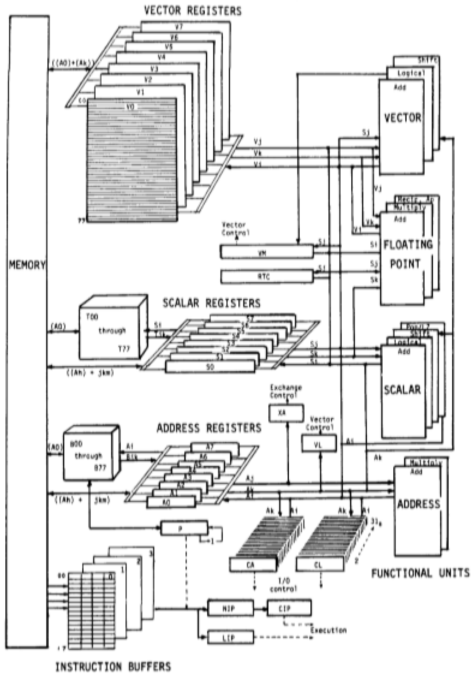


FIGURE 2.7: Barycentric algorithm Source [2]

Point P lies inside the triangle if for all the affine transformations of the combinations of λ_1, λ_2 and λ_3 where $0 < \lambda_1 < 1$, $0 < \lambda_2 < 1$,$0 < \lambda_3 < 1$.

2.4 CRAY-1

CRAY-1 is one of the earliest supercomputers utilising vector parallel architecture[4]. It comprises of a main memory feeding data to and from a set of scalar and vector registers. The ALU operations are performed by 12 independent functional units. The maximum size of main memory on CRAY-1 is 1 million 64-bit words which are divided into 16 memory banks that operate in a concurrent manner. [12]



Source(Russel et al.)[12]

FIGURE 2.8: Pipelined computation in CRAY-1.

We shall focus on the vector parallel operation that CRAY-1 achieves in its processor. The vector registers ($V_0, V_1 ... V_7$) take in the data from the main memory. Each vector register holds a single vector 64 bits wide. These are connected to the functional units via ports(V_i, V_j, V_k) that do the arithmetic and vector operations such as ADD, Logical shift, Multiply etc. These functional units are completely pipelined and can start a new operation every new clock cycle. The vector control registers control the flow of data between ALUs and vector registers. There also few scalar registers that can likewise provide data as input to the vector functional units as well as compute address to pass to the vector control registers. These are normal 32 bit general purpose registers and 32 floating point registers. [12].

2.5 Previous Work

The previous student projects were also studied like Hannah Lee's Further Development of Graphical interface for Computer Games which also used an ARM SoC using an M0 soft core that used for standard and barycentric algorithm to display triangles on a 640x480 VGA screen. It was synthesised on a Nexys 4 FPGA board [11]. The other student project that was also extensively studied was Bing Xue's work on graphical interface for computer games [1] that was used to display a 3D game on a Nexys 4 FPGA board. Advanced functionalities of the project included instant rendering, 3D transforms, parallel rasterization and double frame buffers.

Chapter 3

Hardware Design

3.1 Overview of System

This system uses ARM AHB-LITE system on a chip to implement the design.The design uses Altera De1-SoC(CSEMA5F31C6N) FPGA to implement the system. A 7 inch VGA monitor is used as a VGA screen. Intel Quartus Lite is used for the synthesis of the hardware and its built in Programmer load the .sof file into the DE-1 SoC board via the USB interface. The ARM Cortex M0 is the main processor that is being used for this project. This processor source code is an obfuscated code designed not to be reverse engineered. Figure 3.1 showcases the high level overview of the system. The modules coloured in blue in Figure 3.1 are custom third party modules. These include the AHB-LITE slaves RAM and SWITCHES. The SWITCHES are functionally irrelevant to the design and are a relic carried forward from custom third party modules and this has been used to test the functionality of other AHB-LITE slave modules. The button 2 of the FPGA is used as a negative edge triggered reset for the entire system. All the AHB-LITE slaves are connected to the the M0 processor using AHB-LITE interconnect. The custom hardware that was built for this project are PIXEL MEMORY, HARDWARE RASTERISER and VGA INTERFACE. These modules are explained in detail in further sections of this chapter. The RAM reads the code.hex file which gives the sequence of instructions to be followed for the M0 processor to execute. Using software, the pixel position coordinates are written into the HARDWARE RASTERISER. The HARDWARE RASTERISER spits out the pixel data for the position which is read by the software and these are then written into the PIXEL MEMORY through the software. The PIXEL MEMORY stores the pixel values for all the pixel positions. This is then read by the VGA INTERFACE which converts the digital pixel data into real pixels on the VGA screen.

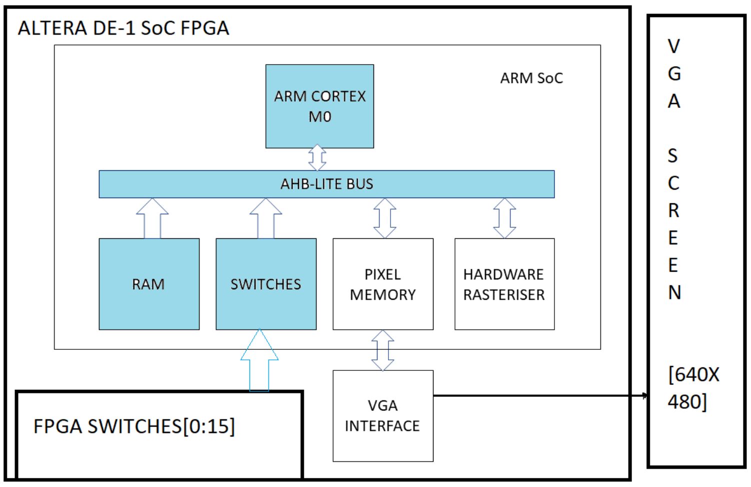


Figure 3.1: Overview system

3.2 VGA interface

The Figure 3.2 illustrates the VGA interface's signals and how it interact with other modules. Other than the usual VGA signals as discussed earlier in Chapter 2 VGA interface section, there are additional 3 signals, mainly `pixel` , `pixel_x` and `pixel_y` . The signal `pixel` is an input while the other two are output. `pixel_x` and `pixel_y` are the pixel positions in x and y direction. These range from 0 to 640 and 0 to 480 respectively as discussed in Chapter 2 VGA interface section . The internal counters of `H_count` and V_count are assigned to `pixel_x` and `pixel_y` if `video_on_H` and `video_on_V` are high. These are control signals that are set high during the duration as long as pixels lie inside the range of the VGA screen. The input `pixel` is the pixel data incoming from the ARM SoC's `ahb_pix_mem` module. This is assigned to the `Green_data` of VGA screen for this project.

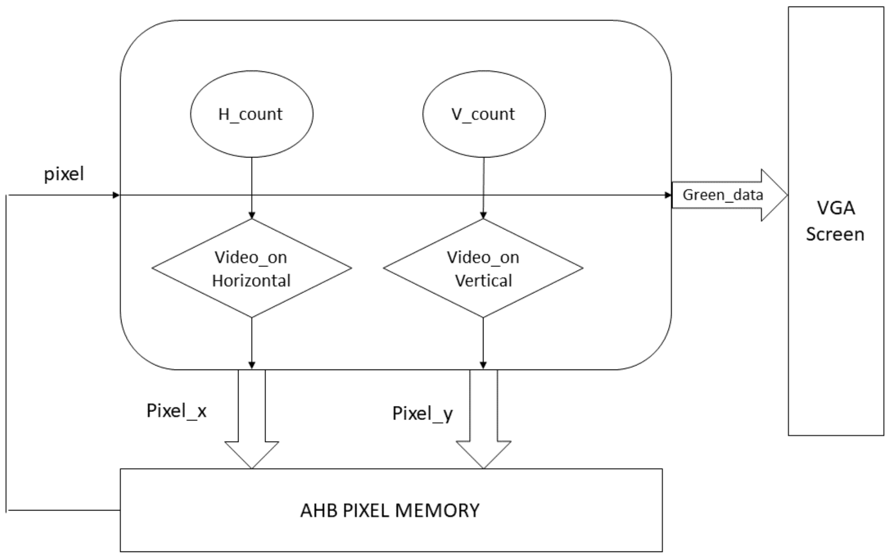


FIGURE 3.2: VGA interface block diagram

3.3 Rasterisation

Rasterisation is the process used to describe the way in which images are converted from electronic data into pixels on screen. The task of rasterisation involves lots of computations done over the entire range of pixels in the screen to decide which part of the screen is to be switched ON and which part to be switched OFF. Barycentric rasterisation which is inherently parallel in nature as discussed in Chapter 2 barycentric algorithm section, is a perfect candidate for vector parallel architecture which will be explored in further sections.

3.4 Hardware acceleration module

The hardware acceleration module implements the barycentric algorithm discussed in Chapter 2 Barycentric algorithm section, to rasterise the triangle. It realises the following equations which are a rearrangement of the barycentric coordinate equations to reduce division operation :

$$\lambda_1 DetT = ((y2 - y3) * (x - x3)) + ((x3 - x2) * (y - y3)) \tag{3.1}$$

$$\lambda_2 DetT = ((y3 - y1) * (x - x3)) + ((x1 - x3) * (y - y3)) \tag{3.2}$$

$$DetT = ((y2 - y3) * (x1 - x3)) + ((x3 - x2) * (y1 - y3)) \tag{3.3}$$

The module then compares the value of $\lambda_1 DetT$ with DetT to get a 1 bit value called as `L1_positive` and similarly for $\lambda_2 DetT$ as `L2_positive.`Instead of explicitly calculating λ_3 as $\lambda_1 + \lambda_3$,`L1_positive` and `L2_positive` are added and compared with DetT to give `L3_positive`. If all the values in `L1_positive` ,`L2_positive` ,`L3_positive` are positive the pixel in the particular coordinate value of `x` and `y` is `1` as shown in the systemverilog code below.

```
if ((L1_detT >= 0) == (detT >= 0))
L1_positive <= '1 ;
else
L1_positive <= '0 ;



if ((L2_detT >= 0) == (detT >= 0))
    L2_positive <= '1 ;
else
L2_positive <= '0 ;

if (((L1_detT + L2_detT) <= detT) == (detT >= 0))
    L3_positive <= '1 ;
else
L3_positive <= '0 ;

if((L1_positive == '1) && (L2_positive == '1) &&  (L3_positive == '1 ))
    inside_triangle <= '1 ;
else
    inside_triangle <= '0 ;
```

The following subsections will cover the arrangement of ALMS, adders and comparators to get a pixel data value for every pixel position of `x` and `y` .

3.4.1 Hardware realisation of L1_DetT , L1_DetT and DetT

Figure 3.3, 3.2 and 3.3 illustrates the arithmetic logic to compute a value of `L1_DetT` , `L2_DetT` and `DetT` as discussed in equations 3.1, 3.2 and 3.3. x, y, x1, y1, x2, y2, x3 and y3 arrive in as signed 2s compliment 11 bit numbers. The computation results in a maximum range of 32 bit signed number.

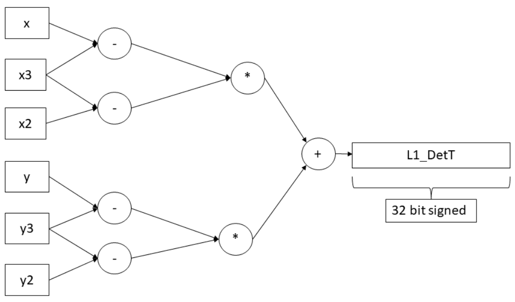


FIGURE 3.3: $\lambda_1 DetT$ arithmetic logic

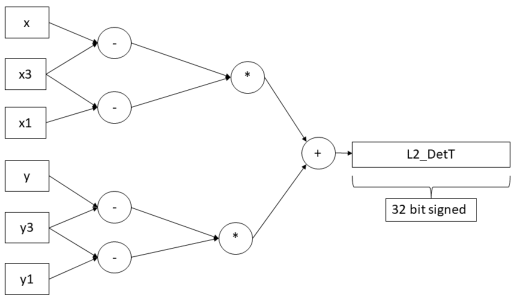


FIGURE 3.4: $\lambda_2 DetT$ arithmetic logic

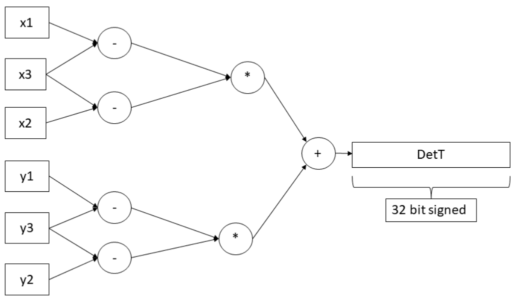


FIGURE 3.5: $DetT$ arithmetic logic

3.4.2 Hardware realisation of L1_positive, L2_positive and L3_positive

When all these value have been computed, these values λ_1 and λ_2 and λ_3 are checked to see if they are greater than 0. If the XNOR of `L1_DetT` and `DetT` is `true` , `L1_positive` is set to 1. Similarly if XNOR of `L2_DetT` and `DetT` is `true` , `L2_positive` is set to 1. This is illustrated in Figure 3.6

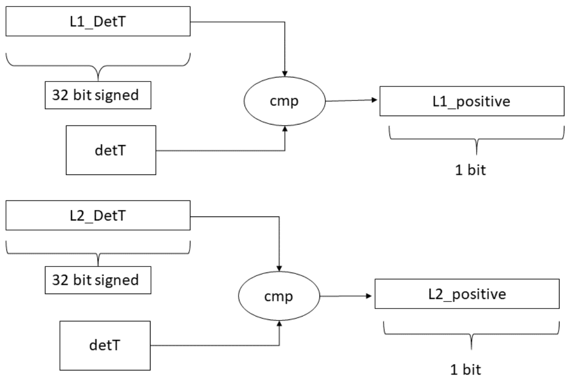


FIGURE 3.6: λ_1 and λ_2 positivity check arithmetic logic

For `L3_positive` , the summation of `L1_DetT` and `L2_DetT` are XNORed with `DetT` after they have been checked to be greater than 0. This is illustrated in Figure 3.7

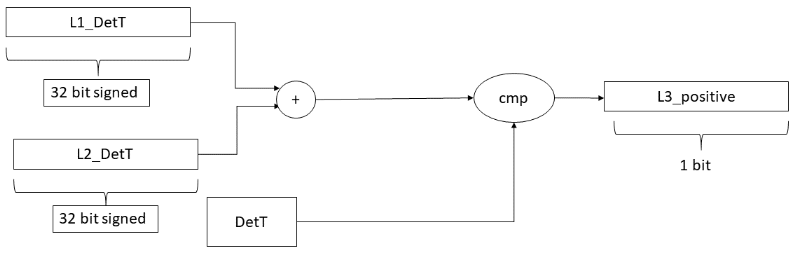


FIGURE 3.7: λ_3 positivity check arithmetic logic

3.4.3 Hardware realisation of inside_triangle

To calculate if the pixel is the said x and y coordinate is 0 or 1, the values of `L1_positive` and `L2_positive` and `L3_positive` are ANDed together and if the resulting operation is 1, the port inside_triangle is set to 1 else it is 0. This is illustrated in Figure 3.8 .

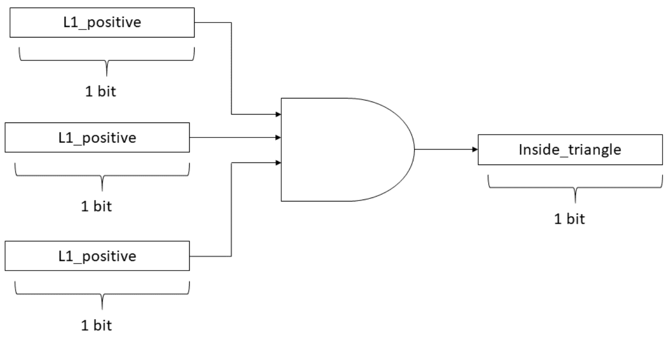


FIGURE 3.8: Logic to check if pixel value is 0 or 1

3.5 Pixel memory module

This section covers the pixel memory module where the pixel value of all pixels in the VGA screen are stored in a dual port synchronous RAM.

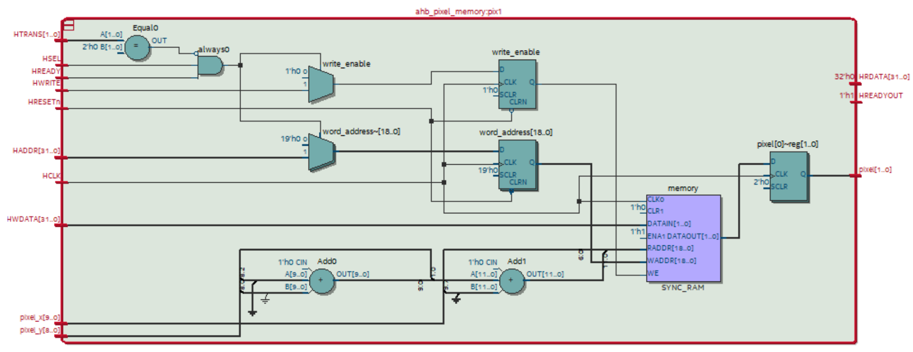


FIGURE 3.9: pixel memory module as synthesised

Figure 3.9 illustrates a high level SoC view of the AHB-LITE SLAVE module for the pixel memory. Compared to the earlier described AHB-LITE slave module for the hardware acceleration, this one has particularly 3 extra non AHB-LITE signals. These 3 signals are `pixel_x` , `pixel_y` and `pixel` . The input of `pixel_x` and `pixel_y` are brought in from the VGA interface. These are used to calculate the address of the pixel position defined by

$$pixel\_address = pixel\_x + (screenlength) * pixel\_y$$

$$screenlength = 640$$

This makes the synchronous memory's size to be 1 bit by 307200 bits with a dual addressable, one from `pixel_address` and the other from `HADDR` .The write to this synchronous memory is done by `HWDATA` .The data is read out of the memory via `pixel` port which acts as an input to the VGA interface.

3.6 Vector parallelisation study

Vector parallelisation of this graphics accelerator would involve pipelining the entire graphics accelerator in such a manner that every clock cycle, a pixel value is spit out by the combination `ahb_hwa` and `ahb_pix_mem` modules and there is no delay in the throughput of the barycentric rasterisation algorithm and updating of pixel value in the memory. This is further illustrated in the sections below.

3.6.1 Pipelining of L1_DetT, L2_DetT and DetT

The Figure 3.10 shows the pipelining involved in the calculation of `L1_DetT` , `L2_DetT` and `DetT` are done in parallel .In this pipelining approach, the values of row vector x[0:640] and y[0:480] are made using counters on the fly and hence there are no software associated instruction delay of `HWRITE` as implemented and discussed in the design of hardware rasterisation module in Chapter 3 hardware acceleration module section, to get the values of x and y. The outputs of the subtraction operations are stored in pipeline registers 1 to be accessible to the multipliers to start multiplying. Similarly, the output of multipliers is stored in pipeline registers 2 to be available to the adders immediately to start adding. In this arrangement of the multipliers, adders and subtractors throughput is 1 output every clock cycle in `L1_DetT` ,`L2_DetT` and `DetT` .

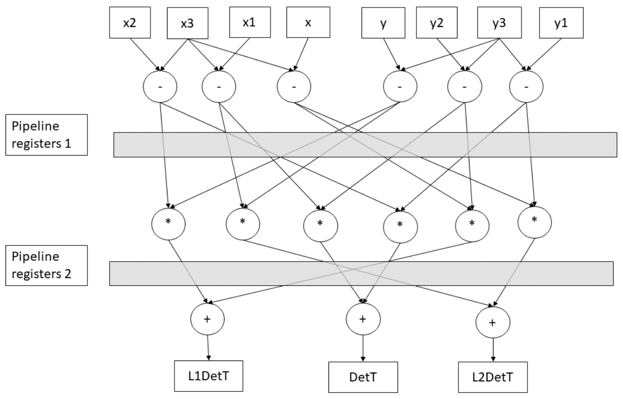


FIGURE 3.10: Pipelining of $\lambda_1 DetT$, $\lambda_2 DetT$ and $DetT$

3.6.2 Pipelining of L1_positive, L2_positive and L3_positive

In the pipelining of `L1_positive` ,`L2_positive` and `L3_positive` , we can see from Figure 3.11 that comparators of `L1_positive` and `L2_positive` finish earlier than `L3_positive` when done in parallel. Hence during pipelining, we are including extra pipeline registers for them during the `L3_positive` comparator operation so that the throughput of `L1_positive` , `L2_positive` and `L3_positive` is 1 output for every clock cycle.

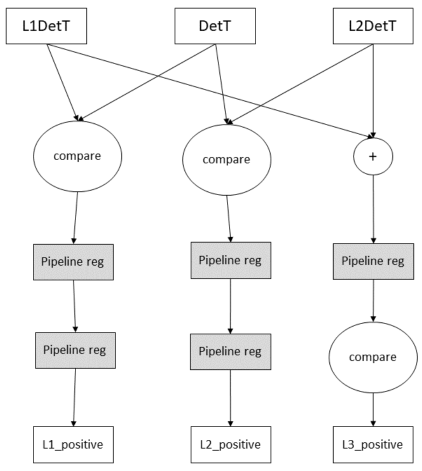


FIGURE 3.11: Pipelining of λ_1, λ_2 and λ_3 positivity check

3.6.3 Pipelining of pixel value

The final pipelining would be in the AND operation of `L1_positive` , `L2_positive` and `L3_positive` to get the value of `inside_triangle` . However, in a pipelined design, the pixel memory can also be incorporated such that the synchronous ram is written in every clock cycle. For this, as shown in Figure 3.12 the pixel address in calculated and every intermediate result is stored in pipeline registers. The number of pipeline registers after the addition operation where the pixel address has already been calculated must match the delay taken for the calculation of `inside_triangle` . In this example of pipelining, the maximum number of pipeline registers used are through the critical path of `L3_positive` ie 4 pipeline registers as well as the register delay of `L1_DetT` and `L3_positive` . The delay taken for the arithmetic and comparison operations should also be accounted. If all the timing considerations are met, then at every new clock cycle a new address of the synchronous pixel memory would be updated with the pixel value of the triangle. A pipelined approach like this would take fewer clock cycles to compute the pixel value for the every pixel position and update it on the VGA screen instantaneously.

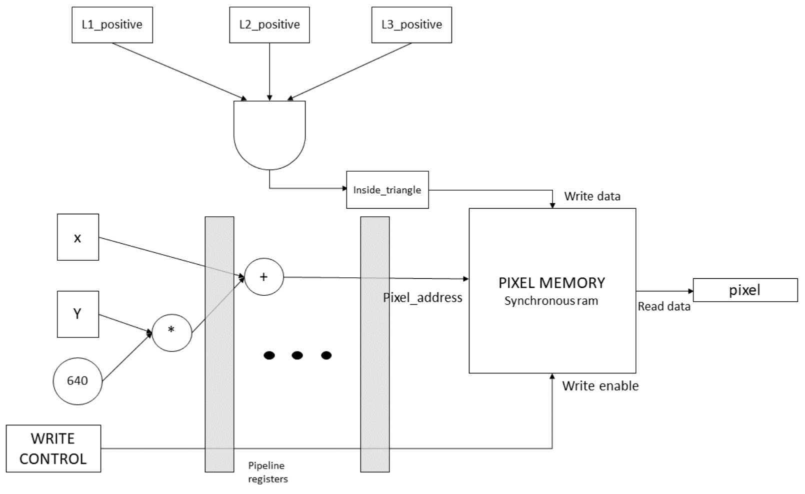


FIGURE 3.12: Pipelining of pixel value

Chapter 4

Software Design

The ARM SoC was programmed in C code and compiled in university UNIX machine and then converted to a hex file. This hex file is read by the program memory(`RAM`) which gives the sequence of instructions to be executed by the ARM M0 processor. This chapter will discuss the designs in software.

4.1 Memory Map

The software helps the M0 processor communicate with its various slaves. Different slaves have different memory locations. The address decoder decides the address of different locations and with the help of partial memory addressing, the decoder logic used to identify the slaves is reduced. In this system, word aligned addressing is used to select the data from various slaves as shown in table 4.1. AHB_SW_BASE stores the data coming in from the switches. AHB_PIX_BASE stores the 307200 pixel values for all the pixel positions in the VGA screen. This data is stored by calling a subroutine write_pix in the `main` function which writes takes in the position of x and y coordinate of the pixel as well as the colour of the pixel. AHB_HWA_BASE stores the data for triangle coordinates x1,y1,x2,y2,x3 and y3. It also takes in the pixel position coordinate x and y via HWA_REGS[6] and HWA_REGS[7] respectively.These are accessed by the subroutine write_hwa .The pixel value is read by the ARM S0C by the subroutine read_hwa and read the register HWA_REGS[8]. All these addresses are word aligned. This means that if HWA_REGS[0] addresses 60000000 , HWA_REGS[1] will address 60000016 ,2 bytes incremented.

Name	Address
AHB_SW_BASE	0x40000000
AHB_PIX_BASE	0X50000000
AHB_HWA_BASE	0X60000000

Table 4.1: Memory Map

4.2 Software Rasterisation

The difference between software and hardware rasterisation is that the data flow and logic handled by `AHB_HWA` is written as a number of for loops in C code executed by ARM M0 processor.

```
int main()
 int x1 = 200; int y1 = 199;
 int x2 = 450; int y2 = 240;
 int x3 = 320; int y3 = 30;
 int L1_detT ; int L2_detT ; int detT ;
 int L1_positive, L2_positive, L3_positive ;

for (int x = 0 ; x < 640 ; x++){
    for (int y = 0 ; y < 480 ; y++) {
     L1_detT   =   ((y2-y3) *  (x-x3)) + ((x3-x2) *  (y-y3)) ;
     L2_detT   =   ((y3-y1) *  (x-x3)) + ((x1-x3) *  (y-y3)) ;
     detT      =   ((y2-y3) * (x1-x3)) + ((x3-x2) * (y1-y3)) ;

     L1_positive = ((L1_detT >= 0) == (detT >= 0)) ;
     L2_positive = ((L2_detT >= 0) == (detT >= 0)) ;
     L3_positive = (((L1_detT + L2_detT) <= detT) == (detT >= 0)) ;
        if(L1_positive && L2_positive && L3_positive)
            write_pix(x,y,1)
    }
}
```

The code above does the same hardware rasterisation calculation for a triangle as discussed in Chapter 3 Hardware acceleration module section for all the pixel position values. Detailed code is available in Appendix A.

Chapter 5

Testing

This chapter will explain the testing and simulation done before the final implementation of the design in the Intel DE-1 FPGA. The following sections will explain the simulation results done for the entire system in Cadence NCSim.

5.1 VGA interface simulation

Figure 5.1 illustrates the red, green and blue signals along with `VSync` ,`HSync` ,`clock` and `VGA_Blank_N` signal. These simulations show that the VGA interface is working just fine and the signal changes in the ports shown are similar to the functions of these signals as discussed in Chapter 2 in the VGA interface section. This design uses just green data, hence we can see that green data goes to 255 changing from the default 0 after the `VGA_BLANK_N` (Blanking signal) signal has run its course. To check if the VGA interface is writing the data, we look and see if the `VGA_G` is writing any data after the `VGA_BLANK_N` has run as discussed in Chapter 2 VGA interface section.

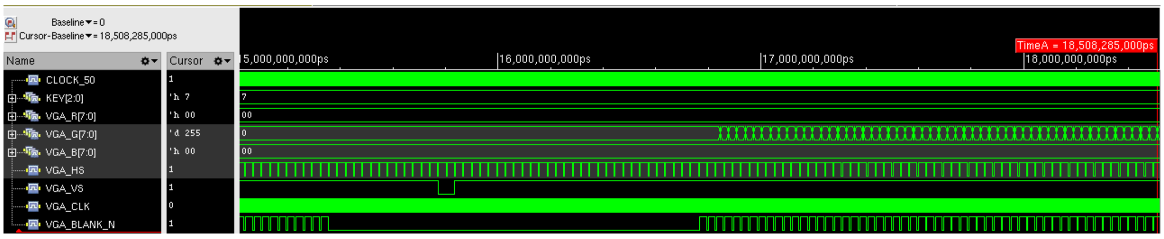


FIGURE 5.1: VGA interface simulation

5.2 Hardware accelerator simulation

The hardware accelerator was also simulated in NCSim. The Figure 5.2 illustrates the writing of data into the registers x1, y1, x2, y2, x3, and y3 at word address 0,1,2,3,4 and 5. We can see from the simulation that the address is word aligned as discussed in Chapter 4 Memory map section and the data arrive at expected timing.

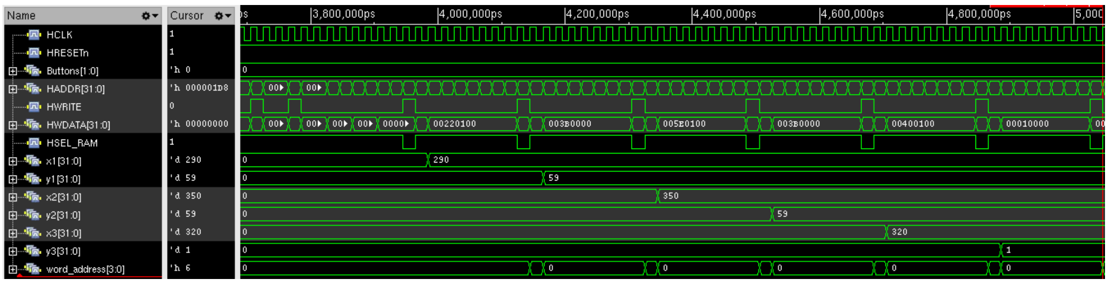


FIGURE 5.2: Hardware accelerator input simulation

The Figure 5.3 shows the hardware rasterisation for a where the triangle coordinates are 320 and 10. The values of `L1_DetT` ,and `L2_DetT` are exactly as expected. The values of `L1_positive` ,`L2_positive` and `L3_positive` are 1 and hence the value of `inside_triangle` is 1 and this is in line with our expectation.

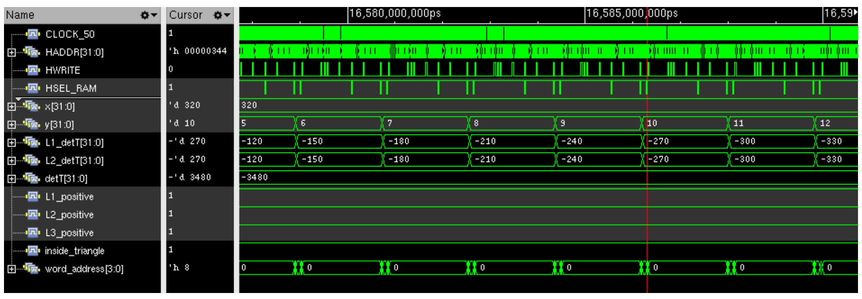


FIGURE 5.3: Hardware accelerator output simulation

5.3 Pixel memory simulation

Figure 5.4 illustrates the calculation of pixel address using the values of `pixel_x` and `pixel_y`. At the pixel coordinate (304,4) , the value of pixel address is in line with the expected value. The storage of data was further tested through memory viewer. However due to the sheer size of the memory array, all the memory viewer could tell was if any data was going inside the synchronous ram. This is better tested through "`$display` " in the hardware code which gives out a display in the console window. This was also extensively used to see the flow of data.

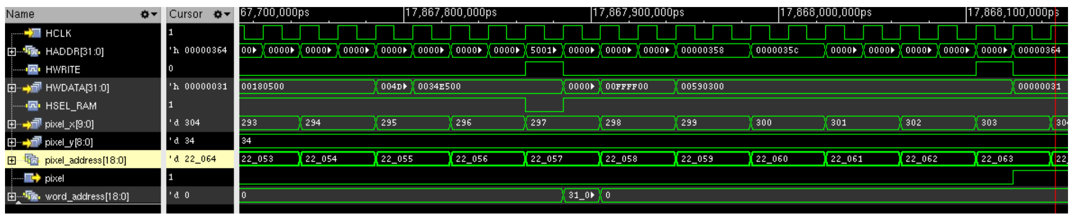


FIGURE 5.4: Pixel memory simulation

5.4 Pattern

The pattern that was chosen to be made on the VGA screen was a jagged Christmas tree. As shown in the Figure 5.5, the Christmas tree consists of 3 triangles and a vertical bar. The three triangles' coordinates are namely :

- Triangle A (290,59), (350,59), (320,1)
- Triangle B (200,199), (450,240), (320,30)
- Triangle C (140,359), (500,270), (320,90)

The tree trunk is formed by firing up the vertical column pixels between 240 and 480. and similarly the horizontal pixels between 310 and 330. Since only green pixels are fired up by the VGA interface, this gives us a silhouette of a jagged Christmas tree in green. The coordinates for the triangle were purposefully chosen to give a jagged tree as this checks the capability of rasterising obtuse triangles and overlapping triangles.

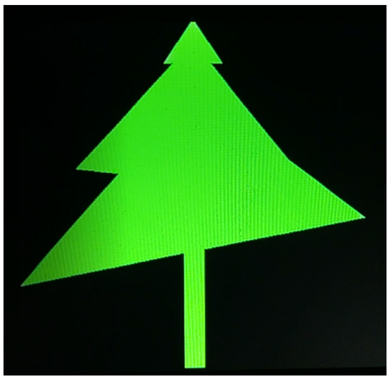


FIGURE 5.5: Triangle image

Chapter 6

Comparison and evaluation

6.1 Performance comparison

This chapter will compare the software triangle rasterisation and the hardware accelerator to objectively evaluate to see if the hardware accelerator gives any advantages over the software algorithm. This chapter will also discuss the hardware overhead that arises from the inclusion of hardware accelerator. Figures 6.1 and 6.2 are frames of the triangle rasterisation process as captured by a camera recording at 240 frames per second. The entire rasterisation of the Christmas tree in rasterisation took approximately 6 seconds while the hardware accelerated design did the same in just 2 seconds. A case by case study for the rasterisation of individual triangles could also be done in the software rasterisation design. As shown in Figure 6.1, the rasterisation of triangle C took around 198ms. Compared to the software design, the hardware accelerated one took just 90ms to rasterise the entire Christmas tree.

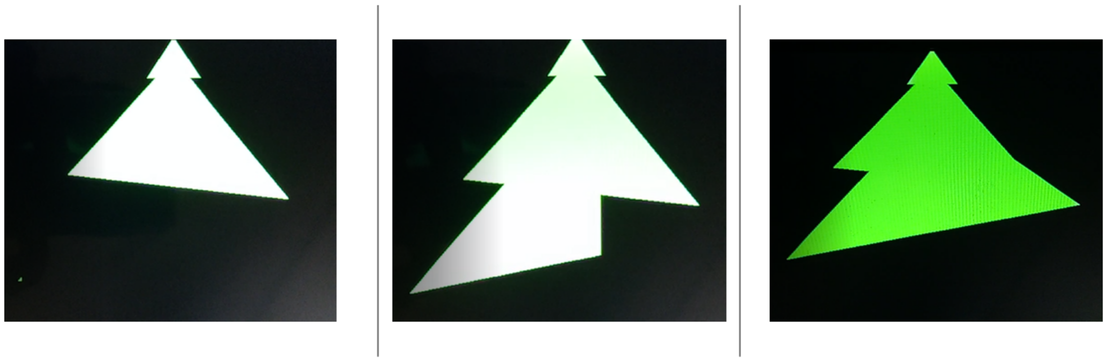


FIGURE 6.1: Frames of Christmas tree rasterisation done in software

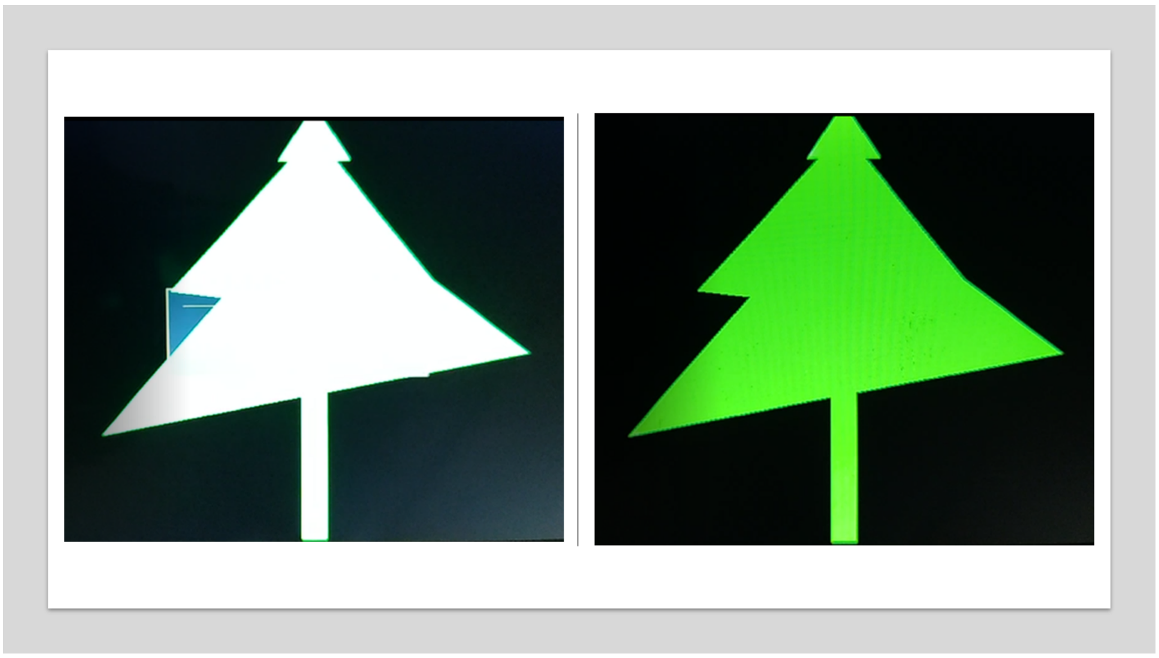
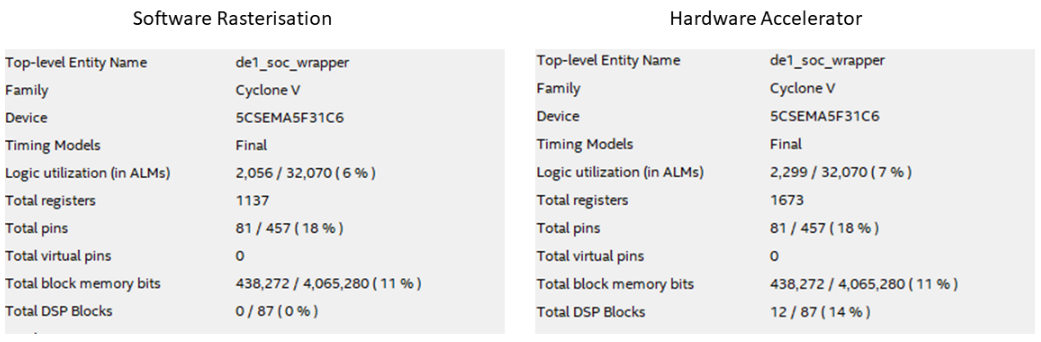


FIGURE 6.2: Frames of Christmas tree rasterisation done in hardware

This can be attributed to the nature of the triangle rasterisation in software, happening in a serial fashion by performing the rasterisation calculation for each and every pixel as well as sequential running of the for loops for rasterisation of different triangles.

6.2 Synthesis Comparison

This section will discuss the synthesis results of both hardware accelerator and software rasterisation. Figure 6.3 illustrates the synthesis summary of both the hardware accelerator as well as the software rasterisation design. The numbers for both these designs are similar for the number of memory block units. The main difference arise in the number of ALMs used, total registers and DSP blocks used. The DSP blocks are used to access the embedded multipliers in the FPGA for multiplication operation. We can see that there is only a 1% difference between the usage of ALMs of hardware accelerator and software rasteriser design.


Software Rasterisation

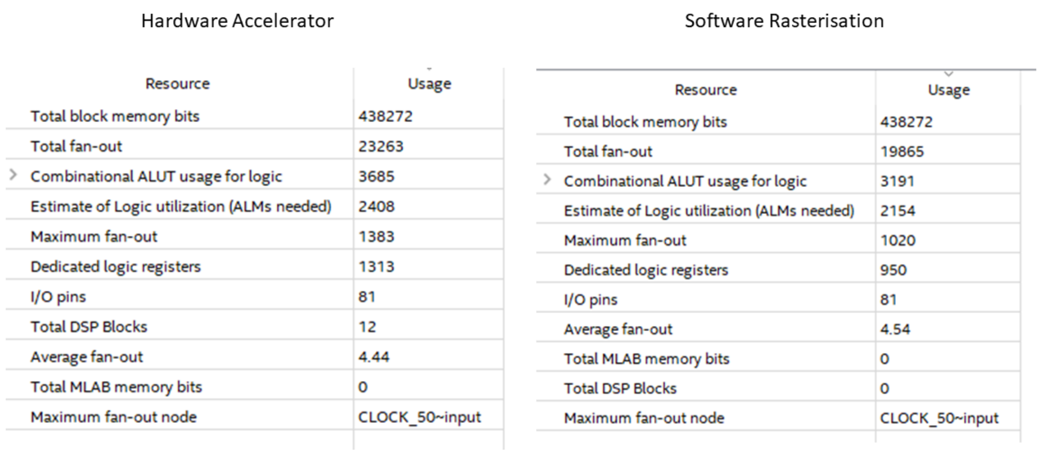
Top-level Entity Name	de1_soc_wrapper
Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	Final
Logic utilization (in ALMs)	2,056 / 32,070 (6 %)
Total registers	1137
Total pins	81 / 457 (18 %)
Total virtual pins	0
Total block memory bits	438,272 / 4,065,280 (11 %)
Total DSP Blocks	0 / 87 (0 %)

Hardware Accelerator

Top-level Entity Name	de1_soc_wrapper
Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	Final
Logic utilization (in ALMs)	2,299 / 32,070 (7 %)
Total registers	1673
Total pins	81 / 457 (18 %)
Total virtual pins	0
Total block memory bits	438,272 / 4,065,280 (11 %)
Total DSP Blocks	12 / 87 (14 %)

FIGURE 6.3: Synthesis summary comparison

Figure 6.4 goes into the detail for the resource usage summary differences between both the designs. We can see how the increase of logic utilisation as illustrated in Figure 6.3 is allocated in the final synthesis. Figure 6.4 also illustrates the fan out of registers. The detailed report of DSP block utilisation is illustrated in Figure6.5 showing the number of signed and unsigned multipliers, adders etc.



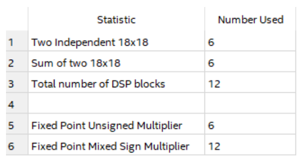
Hardware Accelerator

Resource	Usage
Total block memory bits	438272
Total fan-out	23263
Combinational ALUT usage for logic	3685
Estimate of Logic utilization (ALMs needed)	2408
Maximum fan-out	1383
Dedicated logic registers	1313
I/O pins	81
Total DSP Blocks	12
Average fan-out	4.44
Total MLAB memory bits	0
Maximum fan-out node	CLOCK_50~input

Software Rasterisation

Resource	Usage
Total block memory bits	438272
Total fan-out	19865
Combinational ALUT usage for logic	3191
Estimate of Logic utilization (ALMs needed)	2154
Maximum fan-out	1020
Dedicated logic registers	950
I/O pins	81
Average fan-out	4.54
Total MLAB memory bits	0
Total DSP Blocks	0
Maximum fan-out node	CLOCK_50~input

FIGURE 6.4: Resource usage summary comparison



	Statistic	Number Used
1	Two Independent 18x18	6
2	Sum of two 18x18	6
3	Total number of DSP blocks	12
4		
5	Fixed Point Unsigned Multiplier	6
6	Fixed Point Mixed Sign Multiplier	12

FIGURE 6.5: DSP Block usage in hardware accelerator design

The main numbers that we focus on are the combinational ALUT used for logic, ALMs needed and dedicated logic registers. The following table 6.1 calculates and shows the overhead increase in the following parameters for the hardware accelerator design.

Parameter	Percentage increase
ALUTs	15.48
ALMs	11.79
Dedicated Logic registers	38.2

TABLE 6.1: Percentage increase in resource usage in hardware accelerator design

Chapter 7

Conclusions

The main objective of this project was to develop a vector parallel graphics accelerator on an FPGA. The hardware accelerator combined with the pixel memory and graphical interface was integrated with an ARM SoC and was successfully implemented. However the Vector parallel part of the project was done solely as a study and was not implemented. This suggests that the project was a partial success and achieved most of the aims of the project. The hardware accelerator design was practically shown to be faster than the software rasteriser version.

7.1 Limitation of design

The hardware accelerator though shown to be faster than the software rasteriser, is not optimised well enough for the ARM SoC. The hardware accelerator is not used well enough to point where bottlenecks arise in the system. The system design is also not optimised to rasterise any image by creating a triangle mesh and filling it with colours, textures etc. The design does not include conditions for clipping of triangles at the edge of the screen, perspective projection for 3d images etc. Last but not the least, the registers for `L1_detT` , `L2_DetT` and `DetT` are full 32 bit registers. This can be further optimised to use less of the adders to complete the addition, as currently there are more adders used in the DSP block than required.

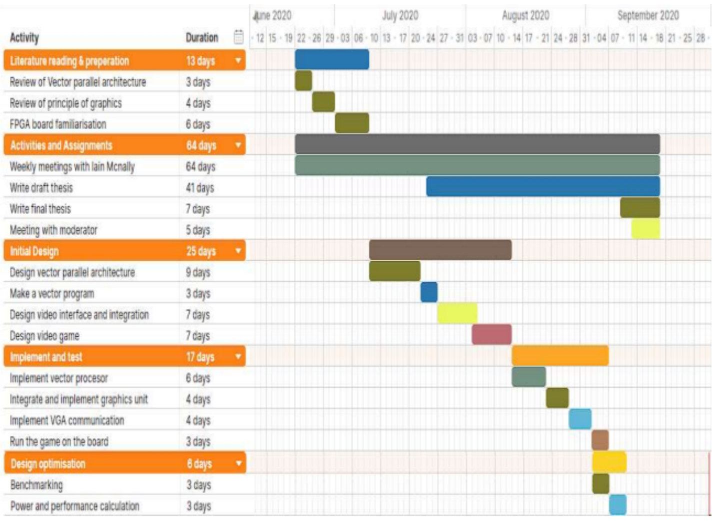
7.2 Future work and applications

There is lot of work remaining to be done and a lot that could potentially be improved upon. The vector parallel pipelining is left to be implemented and practically verify the proposed benefits as discussed in Chapter 1. The full pipelined vector approach has the capability to rasterise images that would not have been possible to be rasterised before on an FPGA based graphics processor designs. Since the hardware accelerator and vector parallel architectures are just techniques to improve performance of multiple arithmetic calculations, this design can also potentially be used for supercomputer applications. FPGA based graphical processing units are inherently scalable and have seen applications for server based computations offering various advantages such as design upgrades, reliability etc. A graphics hardware accelerator too has the potential to be scaled up into a server. This makes the applications of the design range from complex data visualisation ranging from weather data, scientific data, medical data and so on.

Chapter 8

Project Management

The original project plan as shown in Figure 8.1 was to design a processor based around vector parallel architecture and a graphics interface unit. The processor was imagined running a game and a vector program accelerating the graphics unit.



Activity	Duration
Literature reading & preperation	13 days
Review of Vector parallel architecture	3 days
Review of principle of graphics	4 days
FPGA board familiarisation	6 days
Activities and Assignments	64 days
Weekly meetings with Iain Mcnally	64 days
Write draft thesis	41 days
Write final thesis	7 days
Meeting with moderator	5 days
Initial Design	25 days
Design vector parallel architecture	9 days
Make a vector program	3 days
Design video interface and integration	7 days
Design video game	7 days
Implement and test	17 days
Implement vector procesor	6 days
Integrate and implement graphics unit	4 days
Implement VGA communication	4 days
Run the game on the board	3 days
Design optimisation	6 days
Benchmarking	3 days
Power and performance calculation	3 days

Figure 8.1: Original project plan

However on advice from my supervisor, an ARM SoC was used with a VGA interface and the game was initially decided to be designed in the software and the the various AHB Slaves would be used to design the VGA interface, rasterise triangles, store the pixel data etc. However the plan was deviated due to various factors to make a VGA interface and a hardware design to just rasterise triangles. The vector parallelisation was studied in the last few weeks of the deadline and it was not possible to implement the vector pipelining and submit on time. Hence the vector parallel part of the graphics accelerator was done purely as a feasibility study. The project was delayed and the deadline was extended due to COVID-19 and the resulting problems accompanying isolation and remote work during lockdown. The actual project progress looked like as described in table 8.1 .

Milestones	**Completion date**
ARM SoC initial design	14/07/20
VGA interface design	27/07/20
ARM SoC integration with VGA interface	06/08/20
Pixel memory as AHB-LITE slave	26/09/20
Software rasterisation	07/10/20
Hardware rasterisation	15/11/20
Vector parallel pipelining study	24/11/20
Draft report and demonstration	2/12/20

Table 8.1: Actual project progress

Due to limitation of time at the end of the project, a decision was taken not to implement the vector parallel pipelining study that was done for the project. During the month of October, feasibility of using buttons to control the coordinates of triangles was also studied which could have been potentially used to move objects in a game. However, it was taking more time than expected and the task to run on the graphics accelerator would be just a static pattern was decided. The project management was certainly not the best, however the project was adapted and changed due to the arising circumstances in an acceptable manner. The project was developed and version controlled in University Gitlab which provided with excellent redundancy in case of rolling back the code and also provided option of implementing contingency plans due to the availability of previous versions of code.

Appendix A

Appendix

A.1 Software Rasteriser

```
    #define __MAIN_C__

#include <stdint.h>
#include <stdio.h>
#include <stdbool.h>
#include <math.h>

// Define the raw base address values for the i/o devices

#define AHB_SW_BASE                                 0x40000000
#define AHB_PIX_BASE                                0x50000000

// Define pointers with correct type for access to 32-bit i/o devices
volatile uint32_t* SW_REGS = (volatile uint32_t*) AHB_SW_BASE;
volatile uint32_t* PIX_REGS = (volatile uint32_t*) AHB_PIX_BASE;

#include <stdint.h>

////////////////////////////////////////////////////////////////////
// Functions provided to access i/o devices
///////////////////////////////////////////////////////////////////

void write_pix( int p_x, int p_y, int colour) {
  int pix_address ;
  pix_address = p_x + 640*p_y ;
  PIX_REGS[pix_address] = colour;
}


uint32_t read_switches(int addr) {

  return SW_REGS[addr];

}

bool check_switches(int addr) {

  int status, switches_ready;

  status = SW_REGS[2];

  // use the addr value to select one bit of the status register
  switches_ready = (status >> addr) & 1;

  return (switches_ready == 1);

}

void wait_for_any_switch_data(void) {

  // this is a 'busy wait'

  //  ( it should only be used if there is nothing
  //   else for the embedded system to do )

  while ( SW_REGS[2] == 0 );

  return;

}
struct triangle {
  int x1 , y1, x2, y2, x3, y3, L1_detT, L2_detT, detT, L1_positive, L2_positive, L3_positiv
} ;

/////////////////////////////////////////////////////////////////
// Main Function
/////////////////////////////////////////////////////////////////
```

```
int main(void) {
while(1) {
  struct triangle A ;
  struct triangle B ;
  struct triangle C ;
// LOOP FOR A
   A.x1 = 290;
   A.y1 = 59;

   A.x2 = 350;
   A.y2 = 59;

   A.x3 = 320;
   A.y3 = 1;


for ( A.x = 0 ; A.x < 640 ; A.x++){
for ( A.y = 0 ; A.y < 59 ; A.y++) {
A.L1_detT   =   ((A.y2-A.y3) *  (A.x-A.x3)) + ((A.x3-A.x2) *  (A.y-A.y3)) ;
A.L2_detT   =   ((A.y3-A.y1) *  (A.x-A.x3)) + ((A.x1-A.x3) *  (A.y-A.y3)) ;
A.detT  =     ((A.y2-A.y3) * (A.x1-A.x3)) + ((A.x3-A.x2) * (A.y1-A.y3)) ;

A.L1_positive = ((A.L1_detT >= 0) == (A.detT >= 0)) ;
A.L2_positive = ((A.L2_detT >= 0) == (A.detT >= 0)) ;
A.L3_positive = (((A.L1_detT + A.L2_detT) <= A.detT) == (A.detT >= 0)) ;

if(A.L1_positive && A.L2_positive && A.L3_positive)
write_pix(A.x,A.y,1);
}
}
// LOOP FOR B
   B.x1 = 200;
   B.y1 = 199;

   B.x2 = 450;
   B.y2 = 240;

   B.x3 = 320;
   B.y3 = 30;
```

```
for ( B.x = 0 ; B.x < 640 ; B.x++){
for ( B.y = 0 ; B.y < 480 ; B.y++) {
B.L1_detT   =   ((B.y2-B.y3) *  (B.x-B.x3)) + ((B.x3-B.x2) *  (B.y-B.y3)) ;
B.L2_detT   =   ((B.y3-B.y1) *  (B.x-B.x3)) + ((B.x1-B.x3) *  (B.y-B.y3)) ;
B.detT  =     ((B.y2-B.y3) * (B.x1-B.x3)) + ((B.x3-B.x2) * (B.y1-B.y3)) ;

B.L1_positive = ((B.L1_detT >= 0) == (B.detT >= 0)) ;
B.L2_positive = ((B.L2_detT >= 0) == (B.detT >= 0)) ;
B.L3_positive = (((B.L1_detT + B.L2_detT) <= B.detT) == (B.detT >= 0)) ;

if(B.L1_positive && B.L2_positive && B.L3_positive)
write_pix(B.x,B.y,1);
}
}
// LOOP FOR C
   C.x1 = 140;
   C.y1 = 359;

   C.x2 = 500;
   C.y2 = 270;

   C.x3 = 320;
   C.y3 = 90;



for ( C.x = 0 ; C.x < 640 ; C.x++){
for ( C.y = 0 ; C.y < 480 ; C.y++) {
C.L1_detT   =   ((C.y2-C.y3) *  (C.x-C.x3)) + ((C.x3-C.x2) *  (C.y-C.y3)) ;
C.L2_detT   =   ((C.y3-C.y1) *  (C.x-C.x3)) + ((C.x1-C.x3) *  (C.y-C.y3)) ;
C.detT  =     ((C.y2-C.y3) * (C.x1-C.x3)) + ((C.x3-C.x2) * (C.y1-C.y3)) ;

C.L1_positive = ((C.L1_detT >= 0) == (C.detT >= 0)) ;
C.L2_positive = ((C.L2_detT >= 0) == (C.detT >= 0)) ;
C.L3_positive = (((C.L1_detT + C.L2_detT) <= C.detT) == (C.detT >= 0)) ;

if(C.L1_positive && C.L2_positive && C.L3_positive)
write_pix(C.x,C.y,1);
}
}
```

```
// LOOP FOR TRUNK
for ( int i = 0 ; i < 640 ; i++){
for ( int j = 240 ; j < 480 ; j++) {
if( (i >= 310) && (i <= 330) )
write_pix(i,j,1);
}
}
}
}
```

A.2 Hardware accelerator C code

```
#define __MAIN_C__

#include <stdint.h>
#include <stdio.h>
#include <stdbool.h>
#include <math.h>

// Define the raw base address values for the i/o devices

#define AHB_SW_BASE                                0x40000000
#define AHB_PIX_BASE                               0x50000000
#define AHB_HWA_BASE 0x60000000

// Define pointers with correct type for access to 32-bit i/o devices
volatile uint32_t* SW_REGS = (volatile uint32_t*) AHB_SW_BASE;
volatile uint32_t* PIX_REGS = (volatile uint32_t*) AHB_PIX_BASE;
volatile uint32_t* HWA_REGS = (volatile uint32_t*) AHB_HWA_BASE;

#include <stdint.h>

//////////////////////////////////////////////////////////////////
// Functions provided to access i/o devices
/////////////////////////////////////////////////////////////////

void write_pix( int p_x, int p_y, int colour) {
  int pix_address ;
  pix_address = p_x + 640*p_y ;
  PIX_REGS[pix_address] = colour;
}

void write_hwa (int ax1, int ay1, int ax2, int ay2, int ax3, int ay3, int ax, int ay){
  HWA_REGS[0] =ax1;
  HWA_REGS[1] =ay1;
  HWA_REGS[2] =ax2;
  HWA_REGS[3] =ay2;
  HWA_REGS[4] =ax3;
  HWA_REGS[5] =ay3;
  HWA_REGS[6] =ax;
  HWA_REGS[7] =ay;
}

uint32_t read_hwa (void) {

  return HWA_REGS[8];

}
uint32_t read_switches(int addr) {

  return SW_REGS[addr];

}

bool check_switches(int addr) {

  int status, switches_ready;

  status = SW_REGS[2];

  // use the addr value to select one bit of the status register
  switches_ready = (status >> addr) & 1;

  return (switches_ready == 1);

}

void wait_for_any_switch_data(void) {
  // this is a 'busy wait'

  //  ( it should only be used if there is nothing
  //   else for the embedded system to do )

  while ( SW_REGS[2] == 0 );

  return;

}

struct triangle {
  int x1 , y1, x2, y2, x3, y3, x ,y  ;
} ;
//////////////////////////////////////////////////////////////////
// Main Function
//////////////////////////////////////////////////////////////////
int main(void) {

while(1) {
   struct triangle A ;
   struct triangle B ;
   struct triangle C ;

   A.x1 = 290;
   A.y1 = 59;

   A.x2 = 350;
   A.y2 = 59;

   A.x3 = 320;
   A.y3 = 1;

//LOOP FOR B CHECKS ABSTRACT OBTUSE TRIANGLES
   B.x1 = 200;
   B.y1 = 199;

   B.x2 = 450;
   B.y2 = 240;

   B.x3 = 320;
   B.y3 = 30;

   C.x1 = 140;
   C.y1 = 359;

   C.x2 = 500;
   C.y2 = 270;

   C.x3 = 320;
   C.y3 = 90;

// LOOP FOR A
write_hwa(A.x1,A.y1,A.x2,A.y2,A.x3,A.y3,0,0);
for ( A.x = 0 ; A.x < 640 ; A.x++){
HWA_REGS[6] = A.x ;
for ( A.y = 0 ; A.y < 59 ; A.y++) {
HWA_REGS[7] = A.y ;
int hwa_val = HWA_REGS[8] ;
if(hwa_val == 1)
write_pix(A.x,A.y,1);
}
}
// LOOP FOR B
write_hwa(B.x1,B.y1,B.x2,B.y2,B.x3,B.y3,0,0);
for ( B.x = 0 ; B.x < 640 ; B.x++){
HWA_REGS[6] = B.x ;
for ( B.y = 0 ; B.y < 480 ; B.y++) {
HWA_REGS[7] = B.y ;
int hwa_val = HWA_REGS[8] ;
if(hwa_val == 1)
write_pix(B.x,B.y,1);
}
}
// LOOP FOR C
write_hwa(C.x1,C.y1,C.x2,C.y2,C.x3,C.y3,0,0);
for ( C.x = 0 ; C.x < 640 ; C.x++){
HWA_REGS[6] = C.x ;
for ( C.y = 0 ; C.y < 480 ; C.y++) {
HWA_REGS[7] = C.y;
int hwa_val = HWA_REGS[8] ;
if(hwa_val == 1)
write_pix(C.x,C.y,1);
}
}
// LOOP FOR TRUNK
for ( int i = 0 ; i < 640 ; i++){
for ( int j = 240 ; j < 480 ; j++) {
if( (i >= 310) && (i <= 330) )
write_pix(i,j,1);
}
}
}

}
```

Bibliography

[1] Bing Xue. *Graphical Interface for Computer Games.* URL: https://secure.ecs.soton.ac.uk/notes/bim/e_archive/COMP6200/1718/bx1u17/pdfs/Dissertation.pdf.

[2] Duncan Fyfe Gillies. *Graphics course.* URL: https://www.doc.ic.ac.uk/~dfg/graphics/graphics2010/GraphicsSlides08.pdf.

[3] John L Hennessy et al. *Computer Architecture A Quantitative Approach Fourth Edition.* Tech. rep. 2011. URL: https://books.google.com/books?hl=en&lr=&id=gQ-fSqbLfFoC&oi=fnd&pg=PP1&dq=computer+architecture+a+quantitative+approach&ots=mZrsPQZZur&sig=-LqTjAgRzhM4fm08IDx_eT7ILh0.

[4] RW Hockney and CR Jesshope. *Parallel Computers 2: architecture, programming and algorithms.* 2019. URL: https://books.google.com/books?hl=en&lr=&id=7ZKpDwAAQBAJ&oi=fnd&pg=PP9&dq=%40book%7Bhockney2019parallel,+++title%3D%7BParallel+Computers+2:+architecture,+programming+and+algorithms%7D,+++author%3D%7BHockney,+Roger+W+and+Jesshope,+Chris+R%7D,+++year%3D%7B2019%7D,+++publisher%3D%7BCRC+Press%7D+%7D&ots=D2BADsMlBe&sig=ZPl5heFmrnxpPaqrPkokkVqtvWI.

[5] Iain Mcnally. *ARM System on Chip.* Tech. rep.

[6] Iain Mcnally. *ARM System on Chip (FPGA version).* URL: https://www.southampton.ac.uk/~bim/notes/cad/lab/system_on_chip/system_on_programmable_chip.php.

[7] James Hamblen. *Altera UP3 Board Resources.* URL: http://hamblen.ece.gatech.edu/UP3/.

[8] James Hamblen. *razzle.jpg (640462).* URL: http://hamblen.ece.gatech.edu/UP3/razzle.jpg.

[9] K Miura. *Vector-Parallel Approach To High Performance Computing... - Google Scholar.* URL: https://scholar.google.com/scholar?hl=en&as_sdt=0%2C5&q=Vector-Parallel+Approach+To+High+Performance+Computing+And+The+VPP500%2C+a+lecture+by+Kenichi+Miura.+This+video+was+recorded+on+September%2C+1993.&btnG=.

[10] Chris Knebel et al. *Video Graphics Array (VGA)*. Tech. rep.

[11] Hannah Lee and Ying Shih. *Further Development of Graphical Interface for Computer Games Project supervisor: Iain McNally Second examiner: Kirk Martinez A project report submitted for the award of MEng in Electrical and Electronic Engineering*. Tech. rep. 2018.

[12] Richard M. Russell. "The CRAY-1 Computer System". In: *Communications of the ACM* 21.1 (Jan. 1978), pp. 63–72. ISSN: 15577317. DOI: `10.1145/359327.359336`.