Design of a Low-Resource 2D Graphics Engine for FPGAs

University of Cape Town

Prepared by:
Donald Francois Tolmie

Dept. of Electrical Engineering
University of Cape Town

Supervisors:

Professor Andrew Wilkinson
Dept. of Electrical Engineering
University of Cape Town

Mr Samuel Ginsberg
Dept. of Electrical Engineering
University of Cape Town

Submitted to the Department of Electrical Engineering at the University of Cape Town in partial fulfilment of the academic requirements for a Master of Science degree in Electrical Engineering.

October 2018

The copyright of this thesis vests in the author. No quotation from it or information derived from it is to be published without full acknowledgement of the source. The thesis is to be used for private study or non-commercial research purposes only.

Published by the University of Cape Town (UCT) in terms of the non-exclusive license granted to UCT by the author.

University of Cape Town

Declaration

I know the meaning of plagiarism and declare that all the work in the document, save for that which is properly acknowledged, is my own. This thesis/dissertation has been submitted to the Turnitin module (or equivalent similarity and originality checking software) and I confirm that my supervisor has seen my report and any concerns revealed by such have been resolved with my supervisor.

Signed by candidate

Donald Francois Tolmie Date: October 6, 2018

Abstract

This study focused on the design and implementation of a low-resource graphics engine, MicroGE, which can be implemented on an FPGA. MicroGE uses a minimal amount of FPGA resources when compared to other graphics engines. After researching existing graphics engines, it was discovered that most make use of a memory space to store frame buffer data. Because of the restrictions that were imposed on the design of MicroGE, it could not incorporate a large enough memory space to store a frame buffer. It was specified that MicroGE should be able to fit on low-resource FPGAs, without any external memory components. Also, MicroGE should be able to fit on modern, high-resource, FPGAs without using a significant amount of those FPGAs' resources. These goals were achieved by designing MicroGE according to an architecture which differs from the ones of existing graphics engines. MicroGE only renders parts of the video frame, which can be stored in a small memory space, before those parts are transmitted to an HDMI or DVI monitor. After the design was completed, MicroGE, along with other components, was implemented in a VHDL design. Hardware was developed, which contained a Spartan-6 LX25 FPGA, to verify this VHDL. Other verification methods, including the use of VHDL test benches, were also used to verify the VHDL design. A software library, MGAPI, was developed on an Arduino Due microcontroller board. This software library allowed the Arduino Due to display graphics on an HDMI monitor via MicroGE. The Arduino Due was able to update the display of 1000 graphics primitives within 111 ms. The internal FPGA RAM resource usage of MicroGE, 792 kb, was found to be significantly lower than the amount of memory required for a frame buffer. Even though these results were satisfactory, there are still many improvements that can be made to MicroGE. These improvements include increasing the number of rendering capabilities, optimisation of power usage, and increasing the control and video output interfaces.

Acknowledgements

I would like to thank my supervisors, Mr Samuel Ginsberg and Professor Andrew Wilkinson, for their guidance throughout the last two years. Mr Samuel Ginsberg is a very inspiring individual with a passion for electronics. He was very enthusiastic about my Master's degree and was always willing to assist me. My administrative supervisor, Professor Andrew Wilkinson, was always available whenever I required his assistance with any administrative tasks.

I would like to thank my parents, Dr Francois Tolmie and Dr Ansa Tolmie, for all of their support throughout the last two years. They encouraged me to undertake my Master's degree to the best of my abilities. They also reminded me to maintain a balance between my studies, full-time job at Peralex Electronics, and my personal life.

I would like to thank my boss at Peralex, Mr Clifford van Dyk, who taught me how to reason when having to solve difficult problems. The techniques that he taught me allowed me to find creative solutions to many of the problems that I encountered throughout my Master's degree.

I would like to thank one of my colleagues at Peralex, Dr Gavin Teague, for the advice that he gave me on VHDL design. This advice allowed me to solve a very difficult problem that I encountered during the implementation phase of my Master's degree.

I would like to thank Rob Steltman, one of the owners of Barracuda Holdings, for his assistance with my PCB layout. He taught me valuable verification methods and design techniques which allowed me to design a PCB that was fully functional.

Lastly, I would like to thank my friends at UCT for their emotional support throughout the last two years.

Contents

1 Introduction **1**
1.1 Background to the Study . . . 1
1.2 Problem to Investigate . . . 2
1.3 Plan of Development . . . 2
1.4 Requirements of MicroGE . . . 3
1.5 Limitations . . . 4
1.6 Dissertation Overview . . . 4

2 Literature Review **5**
2.1 2D Graphics . . . 5
2.1.1 The Frame Buffer . . . 5
2.1.2 Raster and Vector Graphics . . . 5
2.1.3 Rasterisation of 2D Vector Graphics . . . 6
2.2 Graphics Rendering on FPGAs . . . 8
2.2.1 Software-Intensive Graphics Rendering . . . 8
2.2.2 Hardware Accelerated Graphics Rendering . . . 9
2.2.3 Parallel Processing in Hardware Accelerated Graphics Rendering 11
2.2.4 Control of Graphics Rendering Systems . . . 12
2.2.5 Graphics Rendering Without a Frame Buffer . . . 14
2.3 Video Transmission . . . 14
2.3.1 TMDS . . . 15
2.3.2 HDMI and DVI Video Transmission . . . 16
2.3.3 Video Modes . . . 17
2.3.4 EDID and DDC . . . 17
2.4 Hardware Design Theory . . . 19
2.4.1 PCB Trace Length Matching . . . 19
2.4.2 PCB Trace Impedance Matching . . . 20

3 Theory of Operation **21**
3.1 Overview of MicroGE . . . 21
3.1.1 Rendering Approach of MicroGE . . . 21
3.1.2 Implementation of MicroGE's Rendering Approach . . . 22
3.2 Primitives . . . 22
3.2.1 Primitive Attributes . . . 22
3.2.2 Vector and Raster Primitives . . . 24
3.2.3 Vector Primitive Types . . . 24
3.2.4 Raster Primitive Types . . . 24
3.2.5 Depth Layers . . . 25

3.3 Video Frame Organisation . . . 25
3.3.1 Canvas . . . 25
3.3.2 Grid System . . . 25
3.3.3 Scaling . . . 27
3.3.4 Local Pixel Organisation . . . 28
3.4 Rendering Procedure Overview . . . 29
3.4.1 Line Buffers . . . 29
3.4.2 Rendering Canvas Rows . . . 29
3.5 Vector Rendering . . . 30
3.5.1 Vector Primitives Rendering Procedures Overview . . . 30
3.5.2 Line Primitives Rendering Procedure . . . 31
3.5.3 Triangle Primitives Rendering Procedure . . . 34
3.5.4 Ellipse Primitive Rendering Procedure . . . 35
3.5.5 Rectangle Primitive Rendering Procedure . . . 38
3.5.6 Render Range Equation Optimisation . . . 38
3.6 Raster Rendering . . . 38
3.6.1 Raster Memory . . . 38
3.6.2 Pixel Addressing . . . 39
3.6.3 Image Primitive Rendering Procedure . . . 40
3.6.4 String Memory . . . 41
3.6.5 String Primitive Rendering Procedure . . . 41
3.7 Primitive Processing . . . 42
3.7.1 Primitive Memory and Pointer Memories . . . 42
3.7.2 Primitive Memory Organisation . . . 42
3.7.3 Primitive Selection . . . 43
3.7.4 Primitive Sequencing . . . 43
3.7.5 Primitive Buffer . . . 45
3.7.6 Insert Procedure . . . 45
3.7.7 Delete Procedure . . . 45
3.8 Pipeline Mechanisms . . . 46
3.8.1 Video Transmitter Synchronisation . . . 46
3.8.2 Component Operations . . . 46
3.8.3 Synchronous Component Operation Scheduling . . . 48
3.8.4 Parallel Execution of Component Operations . . . 48
3.8.5 Canvas Segment . . . 49
3.8.6 Render Stages . . . 50

4 Firmware Implementation **52**
4.1 Implementation Overview . . . 52
4.2 MicroGE VHDL Implementation . . . 53
4.2.1 Architecture . . . 53
4.2.2 VHDL Wrapper Components . . . 56
4.2.3 Timing Controller . . . 58
4.2.4 Control Interface Unit . . . 60
4.2.5 Primitive Processor . . . 63
4.2.6 Vector Render Unit . . . 69
4.2.7 Raster Render Unit . . . 72
4.2.8 Line Buffer Controller . . . 75

4.3 Additional VHDL Implementation . . . 78
4.3.1 Clock Generator . . . 78
4.3.2 Video Transmitter . . . 79
4.3.3 EDID Reader . . . 81
4.4 MGAPI C Implementation . . . 84
4.4.1 MGAPI Overview . . . 84
4.4.2 MGAPI High-Level Functions . . . 84
4.4.3 MGAPI Low-level Functions . . . 86

5 Hardware Development 87
5.1 Hardware Development Overview . . . 87
5.2 Circuit Design . . . 88
5.2.1 Power Distribution . . . 88
5.2.2 FPGA . . . 92
5.2.3 FPGA Configuration . . . 94
5.2.4 HDMI Interface . . . 96
5.2.5 Reference Clock . . . 99
5.2.6 Debugging Circuitry and SPI Interface . . . 99
5.2.7 External Memory . . . 100
5.3 PCB Layout . . . 101
5.3.1 PCB Layout Process Overview . . . 101
5.3.2 Component Placement Planning . . . 101
5.3.3 PCB Stackup . . . 101
5.3.4 Footprint Creation . . . 102
5.3.5 Component Placement . . . 103
5.3.6 PCB Layer Allocation . . . 103
5.3.7 Routing . . . 104
5.3.8 Final PCB Design . . . 106
5.4 Fabrication and Assembly . . . 109
5.4.1 PCB Fabrication . . . 109
5.4.2 PCB Assembly . . . 109
5.4.3 Assembled PCB . . . 110
5.5 Hardware Verification . . . 111

6 Verification and Results 113
6.1 Verification and Results Overview . . . 113
6.2 Test Bench Firmware Verification . . . 113
6.2.1 VHDL Test Bench . . . 113
6.2.2 Test Bench Simulator . . . 114
6.2.3 Verification Methods Overview . . . 114
6.2.4 Signal Waveform Verification . . . 114
6.2.5 Memory Contents Verification . . . 115
6.2.6 Automated Verification . . . 115
6.3 Test System Firmware Verification . . . 117
6.3.1 Tests System . . . 117
6.3.2 SPI Interface Verification . . . 119
6.3.3 Display Monitor Output Verification . . . 119
6.3.4 Debugging of VHDL Design . . . 119
6.4 Test Patterns . . . 120

6.5 Resource Utilisation Results 124
6.6 Performance Results 126
6.6.1 Raster Memory Update Speed Test 126
6.6.2 String Memory Update Speed Test 127
6.6.3 Primitive Memory Update Speed Test 128

7 Conclusions and Recommendations **129**
7.1 Conclusions 129
7.2 Recommendations 131

Bibliography **136**

Appendix A Hardware Design Schematics **137**

Appendix B PCB Layout, Assembly and Testing **154**

Appendix C FPGA Resource Utilisation Summary **169**

Appendix D MicroGE Register Map **173**

Appendix E MGAPI Functions **175**

Appendix F Ethics Approval Form **177**

List of Figures

2.1 Explanation of 2D line rasterisation 6
2.2 Explanation of an edge function, a), a triangle formed by three edge functions, b), and a right-angled triangle formed by one edge function 7
2.3 Explanation of 2D solid triangle rasterisation 8
2.4 Explanation of 2D solid ellipse rasterisation 8
2.5 Example of how a graphics rendering system was implemented on an Intel (Altera) FPGA [6] 9
2.6 Example of how a graphics rendering system was implemented on a Xilinx FPGA [7] 10
2.7 Comparison of hardware accelerated graphics rendering to pure software rendering in the study performed by [9] 10
2.8 Block diagram of BADGE [10] 12
2.9 Block diagram of D/AVE 2D [11] 13
2.10 Block diagram of a graphics rendering system which does not incorporate a frame buffer [15] 15
2.11 Block diagram describing the encoding, transmission and decoding of an HDMI video frame [17] 16
2.12 Example of the TMDS periods in the video frame of one of the HDMI video modes [17] 18
2.13 Cross-sectional view of differential PCB traces [25] 20

3.1 Block diagrams summarising the rendering approaches of the researched graphics engines 22
3.2 Block diagram summarising the rendering approach of MicroGE . . . 22
3.3 Block diagram describing how the different roles of MicroGE's rendering approach are fulfilled 23
3.4 Description of how a primitive is rendered from its attributes 23
3.5 Illustration of the vector primitives 24
3.6 Illustration of the raster primitives 25
3.7 Description of the depth layers 25
3.8 Explanation of how the canvas is transmitted along with the HDMI video frame 26
3.9 Description of the global and local grid systems 27
3.10 Explanation of the PixelsPerGrid property 28
3.11 Description of the local pixel organisation 29
3.12 Simplified explanation of how the two line buffers are utilised 30
3.13 Simplified explanation of how a canvas row is rendered 30
3.14 Description of the render ranges and render coordinate systems of the vector primitives 31

3.15 Example of one of the line primitives' coordinate transformation procedures . . . 32
3.16 Description of the coordinate transformation procedures for the line primitives . . . 33
3.17 Description of the render range and render coordinate system of the line primitives, where $m = \frac{dy}{dx}$ . . . 34
3.18 Description of the coordinate transformation procedures for the triangle primitives . . . 36
3.19 Description of the render range and render coordinate system of the triangle primitives, where $m = \frac{dy}{dx}$ . . . 37
3.20 Description of the coordinate transformation procedure, render coordinate system and render range of the ellipse primitive . . . 37
3.21 Description of MicroGE's raster memory . . . 39
3.22 Description of pixel address calculation . . . 39
3.23 Description of the rendering procedure of the image primitive . . . 40
3.24 Description of how the string memory should be used . . . 41
3.25 Description of the rendering procedure of the string primitive . . . 42
3.26 Explanation of how the primitives are stored in the primitive memory 43
3.27 Explanation of the primitive selection process . . . 44
3.28 Explanation of the primitive sequencing process . . . 44
3.29 Explanation of the insert procedure for the primitive buffer . . . 46
3.30 Explanation of the delete procedure for the primitive buffer . . . 46
3.31 Description of when the synchronous component operations are performed . . . 49
3.32 Conceptual timing diagram of the synchronous component operations that are performed during a canvas row period . . . 50
3.33 Conceptual explanation of a canvas segment . . . 50
3.34 Conceptual timing diagram detailing the render stages . . . 51

4.1 Overview of firmware implementation . . . 52
4.2 Architecture of MicroGE . . . 54
4.3 Block diagram of the multiplier component . . . 57
4.4 Block diagram of the DPRAM component . . . 58
4.5 Block diagram of the timing controller . . . 58
4.6 Block diagram of the control interface unit . . . 60
4.7 Block diagram of the primitive processor . . . 64
4.8 Conceptual timing diagram explaining the behaviour of the PrimSequence process . . . 68
4.9 Description of how the primitives to provide to the render units are retrieved from the primitive memory during a grid period . . . 69
4.10 Block diagram of the vector render unit . . . 70
4.11 Block diagram of the comparator component . . . 70
4.12 Conceptual timing diagram explaining the behaviour of the VectorRender process . . . 71
4.13 Block diagram of the raster render unit . . . 73
4.14 Conceptual timing diagram explaining the behaviour of the RasterRender process . . . 75
4.15 Block diagram of the line buffer controller . . . 76

4.16 Conceptual timing diagram explaining the behaviour of the WriteLine process . . . 77
4.17 Block diagram of the clock generator which includes a Xilinx PLL_BASE component [28] . . . 78
4.18 Block diagram of the video transmitter which includes a Xilinx TMDS transmitter [29] . . . 80
4.19 Block diagram of the EDID reader which includes the I^2C master component created by [30] . . . 82
4.20 Block diagram of how MGAPI should be utilised . . . 84

5.1 System design of test board . . . 88
5.2 Block diagram of the power distribution scheme of the hardware design 89
5.3 Schematic of the power connector circuitry . . . 90
5.4 Schematic of the 3.3 V DC-DC voltage regulator . . . 91
5.5 Schematic of the 5 V LDO voltage regulator . . . 92
5.6 Spartan-6 LX25 FPGA IO banks and pin types [33] . . . 93
5.7 Block diagram of the JTAG interface . . . 94
5.8 Schematic of the FPGA's configuration circuitry . . . 95
5.9 Schematic of the PROM's configuration circuitry . . . 96
5.10 Summary of the signals of the HDMI interface . . . 97
5.11 Schematic of the HDMI connector and external components . . . 97
5.12 Schematic of the HDMI port protection device . . . 98
5.13 Schematic of the reference clock generation circuitry . . . 99
5.14 Schematic of the external memory circuitry . . . 100
5.15 Planning of the placement of all the main components . . . 102
5.16 Screenshots of a footprint that was created, a), and how that footprint was inspected, b) . . . 103
5.17 Screenshot of the fanout technique that was used to provide access to the FPGA's pins . . . 104
5.18 Screenshot of how length matching of the high-speed TMDS signal traces was performed . . . 106
5.19 Screenshot of the top side of the PCB layout . . . 106
5.20 Screenshot of the bottom side of the PCB layout . . . 107
5.21 The top side of a 3D rendered image of the test board . . . 108
5.22 The bottom side of a 3D rendered image of the test board . . . 108
5.23 Photo of the top side of the test board . . . 110
5.24 Photo of the bottom side of the test board . . . 110

6.1 Block diagram of the test bench that was used to verify the behaviour of the VHDL design . . . 114
6.2 Screenshot of how ISim's waveform editor was used to inspect the behaviour of some of the DUT's signals . . . 115
6.3 Screenshot of how ISim's memory editor was used to inspect the contents of one of the DUT's memory components . . . 116
6.4 Block diagram describing the automated verification method . . . 116
6.5 Block diagram of the test system that was used to verify the behaviour of the firmware developed in this study . . . 118
6.6 Photo of the test system that was used to verify the behaviour of the firmware developed in this study . . . 118

6.7 Photos of parts of the first vector primitive test pattern 121
6.8 Photos of parts of the second vector primitive test pattern 121
6.9 Photos of parts of the third vector primitive test pattern 122
6.10 Photo of a part of the depth buffer test pattern 122
6.11 Photo of the first image primitive test pattern 123
6.12 Photos of the second image primitive test pattern 123
6.13 Photos of parts of the string primitive test pattern 124
6.14 Photo of a potential application of MicroGE 125
6.15 Performance results of the raster memory update speed test 127
6.16 Performance results of the string memory update speed test 128
6.17 Performance results of the primitive memory update speed test 128

List of Tables

3.1 Description of the primitive attributes 23
3.2 The derived render range equations and their modified forms 38

4.1 Frequencies of the clocks generated by the clock generator component, where $M = 8$, $D = 1$ and $f_{in} = 50$ MHz 79
4.2 The video modes supported by the video transmitter [19], [20] 81
4.3 List of the registers read from the EDID/E-EDID ROM by the EDIDRead process [20], [21] 83

5.1 Power requirements of the hardware design 89

6.1 Summary of the resource utilisation of the VHDL design that was implemented on the Spartan-6 LX25 FPGA 126

List of Acronyms

2D Two-Dimensional

3D Three-Dimensional

ASCII American Standard Code for Information Interchange

ATM Automated Teller Machine

BADGE BitSim Accelerated Display Graphics Engine

BGA Ball Grid Array

BOM Bill of Materials

CEA Consumer Electronics Association

CEC Consumer Electronics Control

CML Current-Mode Logic

COTS Commercial Off-The-Shelf

CPU Central Processing Unit

DC Direct Current

DDC Display Data Channel

DNP Do not populate

DPRAM Dual-Port RAM

DUT Device Under Test

DVI Digital Visual Interface

E-DDC Enhanced DDC

E-EDID Enhanced EDID

EDID Extended Display Identification Data

ESD Electrostatic Discharge

ESL Equivalent Series Inductance

ESR Equivalent Series Resistance

FPGA Field Programmable Gate Array

GPI General-Purpose Input

GPIO General-Purpose Input/Output

GPO General-Purpose Output

GPU Graphics Processing Unit

HDMI High-Definition Multimedia Interface

HPD Hot Plug Detect

I^{2}C Inter-Integrated Circuit

IC Integrated Circuit

IDC Insulation-Displacement Contact

IDE Integrated Development Environment

ILA Integrated Logic Analyzer

IP Intellectual Property

ISim ISE Simulator

JTAG Joint Test Action Group

LCD Liquid Crystal Display

LDO Low-Dropout

LED Light-Emitting Diode

LHS Left-Hand Side

LUT Lookup Table

MCU Microcontroller

MGAPI MicroGE Application Programming Interface

MicroGE Micro Graphics Engine

MOSFET Metal Oxide Semiconductor Field-Effect Transistor

PCB Printed Circuit Board

PLB Processor Local Bus

PLL Phase-Locked Loop

PROM Programmable Read-Only Memory

RAM Random Access Memory

RGB Red, Green and Blue

RHS Right-Hand Side

ROM Read-Only Memory

SDRAM Synchronous Dynamic Random Access Memory

SPI Serial Peripheral Interface

TMDS Transition-Minimized Differential Signaling

TVS Transient Voltage Suppressor

VCO Voltage-Controlled Oscillator

VESA Video Electronics Standards Association

VHDL VHSIC Hardware Description Language

Chapter 1

Introduction

1.1 Background to the Study

As Field Programmable Gate Array (FPGA) technology has evolved throughout the last decades, FPGAs have become more suitable for graphics rendering applications. There are many advantages in using FPGAs, as opposed to commercial off-the-shelf (COTS) solutions, for these applications. Where COTS solutions go obsolete in just a few years, the lifetime of FPGAs is much longer. This makes them suitable for graphics rendering products that need to be supported over long time periods. Also, since a single HDL design can be implemented on different FPGAs, a single graphics rendering system can be implemented on different FPGAs that are released throughout the years. On the other hand, multiple graphics rendering systems can be implemented on a single FPGA, allowing the same hardware to support different graphics rendering applications. Finally, in applications that use expensive PCs to produce simple two-dimensional (2D) graphics, a low-cost FPGA, loaded with a graphics rendering system, can be used as an inexpensive alternative.

It is because of these advantages that many graphics rendering systems have been implemented on FPGAs. Most of these systems make use of the same main components. There is a component to which information, specifying what should be shown on a display monitor, is provided. This component, often called a graphics engine, collaborates with other components to render an image, called a frame buffer, and then store it in a memory component. This memory component is usually an integrated circuit (IC), such as synchronous dynamic random-access memory (SDRAM), that is external to the FPGA. Another component then reads the data from the frame buffer and provides it to a display monitor, such as a High-Definition Multimedia Interface (HDMI) monitor.

Even though this approach has been proven to work well, there is one way in which it can be improved for applications that focus on low FPGA resource utilisation. The memory required to implement a single frame buffer for a 1920 $\times$ 1080 video resolution is approximately 6 MB. This is small when considering storing the frame buffer in external memory, such as SDRAM. On the other hand, when considering storing all of this data in an FPGA itself, the memory requirement is too high for most FPGAs. Modern high-resource FPGAs do have enough memory resources available to store a frame buffer, but when these FPGAs are used in hardware designs, these memory resources are generally required for other tasks.

1.2 Problem to Investigate

The research problem that will be investigated in this study can be formulated as follows:
Current graphics rendering systems that are designed for FPGAs require too much memory to be implemented on low-resource FPGAs without external memory components. Also, even if these graphics rendering systems could be implemented on modern high-resource FPGAs without external memory components, they would still use too much of those FPGAs' resources. Finally, the implementation of a graphics rendering system on an FPGA usually requires either a high-resource FPGA or a combination of a low-resource FPGA and external components. This may unnecessarily increase the overall cost of the system.

The research problem will be solved by designing and implementing a low-resource 2D graphics engine for an FPGA which can be used as the main component of a graphics rendering system. This graphics engine should also allow the resource usage of the overall graphics rendering system that it is used in to be low. This graphics engine is named: Micro Graphics Engine (MicroGE). This name emphasises the main goal of the graphics engine, which is to use a minimal amount of FPGA resources. A software library, MicroGE Application Programming Interface (MGAPI), will also be developed. This software library will allow a device, such as a microcontroller (MCU), to control MicroGE.

1.3 Plan of Development

The study consists of the following aspects:

- Research of existing graphics engines to identify the most significant contributors to their resource utilisation
- Research of the video transmission technologies and 2D graphics rendering techniques that would be used by MicroGE
- Research of the hardware design theory that would be used to develop hardware for verification of MicroGE
- Performing a theoretical design of a graphics engine architecture for MicroGE that should allow MicroGE to be implemented, as part of a graphics rendering system, within a minimal amount of FPGA resources
- Implementation of this theoretical design in VHSIC Hardware Description Language (VHDL); this VHDL design should contain MicroGE, and the other components that are required to provide a complete graphics rendering system
- Development of hardware, which contains a low-resource Spartan-6 LX25 FPGA, for verification of MicroGE
- Development of MGAPI, and implementation of it on an Arduino Due MCU board; MGAPI should allow the Arduino Due to control MicroGE
- Development of verification methods to confirm the functionality of MicroGE

- Demonstration of the rendering capabilities of MicroGE
- Measurement and analysis of the resource usage of MicroGE
- Measurement and analysis of the performance of MicroGE

1.4 Requirements of MicroGE

The requirements of MicroGE are as follows:

- It may not use more than 1 Mb of FPGA random-access memory (RAM) resources
- It may not depend on any external components, such as SDRAM, to perform its rendering operations
- It should allow a device with low computing power, such as an Arduino Due, to render graphics to a display monitor
- It should be able to provide its video output to HDMI and Digital Visual Interface (DVI) display monitors; it does not have to include any video transmission components
- It should be able to support video resolutions of up to 1920 $\times$ 1080
- It should have a minimum frame update rate of one frame per second
- It should be able to scale its video output to different video resolutions
- It should be able to fit on a low-resource FPGA, such as the XC6SLX4, XC6SLX9, XC6SLX16, XC6SLX25 and XC6SLX45 Xilinx Spartan-6 FPGAs
- It should use a minimal amount of a modern high-resource FPGA's resources; less than 1 Mb of block RAM memory, 50000 logic cells, and 60 DSP slices
- It should be simple to add it to an existing FPGA design; thus, its interfaces should be simple
- It should be designed with portability in mind so that it may be implemented on many different FPGA platforms
- It should be controllable from an MCU or a soft-core processor with the aid of MGAPI; MGAPI should also be developed with portability in mind

The typical applications that MicroGE may be used for are:

- Point of sale terminals
- Digital signs
- Automated teller machine (ATM) displays
- Command line interfaces

1.5 Limitations

The following aspects are excluded from this study:

- The research of this study does not include investigation of any three-dimensional (3D) graphics engines or rendering; the research was based on simple 2D graphics only
- MicroGE was only implemented on one FPGA platform, a low-resource Spartan-6 LX25 FPGA, even though it was implemented with portability in mind
- MGAPI was only implemented on one MCU platform, an Arduino Due, even though MGAPI was implemented to be ported to other MCUs and soft-core processors

1.6 Dissertation Overview

The **Literature Review** chapter discusses all of the theory and previous work that apply to this study. The basics of 2D graphics and the rendering methods that are used in this study are discussed. Existing graphics rendering systems and graphics engines are discussed. Methods used to transmit video to HDMI and DVI monitors are discussed. Lastly, the hardware design theory that was used in this study is discussed.

The **Theory of Operation** chapter discusses the differences between MicroGE and the researched graphics engines, presents MicroGE's graphics rendering capabilities, and discusses MicroGE's graphics rendering algorithms.

The **Firmware Implementation** chapter discusses how MicroGE was implemented in VHDL. The implementation of other VHDL components, that were required to provide a graphics rendering system for MicroGE to be tested in, are also discussed. Finally, the implementation of MGAPI, which was used by the Arduino Due to control MicroGE, is discussed.

The **Hardware Development** chapter discusses the development of a test circuit board that was used for the verification of MicroGE. This discussion details the schematic design of the board, how the printed circuit board (PCB) layout of the board was implemented, how the PCB was fabricated, how the PCB was assembled, and how the functionality of the board was tested.

The **Verification and Results** chapter discusses how MicroGE was verified, demonstrates its rendering capabilities, and presents its performance and resource utilisation results.

The **Conclusions and Recommendations** chapter discusses how well the objectives of the study were met and provides recommendations for the further development of both MicroGE and MGAPI.

Chapter 2

Literature Review

2.1 2D Graphics

2.1.1 The Frame Buffer

A digital display monitor contains a set of pixels which allows many different images to be displayed. The display monitor shows a specific image by setting the colour of each pixel according to a provided colour value, for example, a red, green and blue (RGB) value. These colour values are stored in memory locations, called frame buffers. Both the display monitor and the controller of the display monitor, "display controller", can have frame buffers. A display controller produces an image to show on a display monitor, referred to in this thesis as a "video frame", in its frame buffer. This video frame consists of a set of colour values that should be provided to the pixels of the display monitor. After this video frame is transferred to the frame buffer of the display monitor, the display monitor can update the colour of each of its pixels accordingly so that the video frame can be shown. Throughout this thesis, the colour values that are provided to pixels of a display monitor are also referred to as "pixels". [1]

2.1.2 Raster and Vector Graphics

A display controller can create a video frame from different types of information. In this thesis, this information will be referred to as "render information". The render information that will be focussed on in this thesis includes 2D raster graphics and 2D vector graphics. Raster graphics are combinations of pixel values, usually organised in a rectangular 2D array, which represents an image. These images are stored in memory locations within the display controller. A video frame can be created by copying these images to a frame buffer. Vector graphics consist of sets of information which describe how an image should be created. They may contain a set of mathematical equations or algorithms which describe how pixels within a 2D array should be coloured to create an image. The process of creating a 2D array of pixels from a vector graphic is called rasterisation. A video frame can be created by rasterising vector graphics to a frame buffer. An advantage of vector graphics over raster graphics is that the same vector graphic can be used to produce images of different sizes. Raster graphics can be changed to different sizes, but this requires scaling of the original raster graphic to a new one which may contain

aliasing artefacts. [1], [2]

2.1.3 Rasterisation of 2D Vector Graphics

Rasterisation Overview

2D vector graphics are rasterised by colouring a set of pixels, according to some form of render information, such as mathematical equations. These equations can describe graphics primitives, such as lines, rectangles, triangles or ellipses, which are positioned within a 2D Cartesian coordinate system. To rasterise a primitive to a 2D array of pixels, each pixel needs to be assigned with a position from that primitive's coordinate system. The primitive can then be rasterised by evaluating the distances between the assigned positions and the primitive. If, for example, it is required to rasterise a solid primitive, the pixels whose positions fall within the primitive can be provided with colour values while the pixels whose positions fall outside the primitive can be provided with blank colours. These blank colours may be a background or transparent colour. The result of this process is a 2D array of pixels of which some pixels are coloured to represent the mathematically described primitive. [3]

Rasterisation of Lines

Figure 2.1 demonstrates how to rasterise a straight line to a 2D array of pixels. Each pixel within the array is provided with a position from the coordinate system that the line is within. The distance between each pixel's position and the line is then calculated to determine if that pixel should be coloured or not. If the calculated distance is smaller than a specified value, that pixel is coloured; otherwise, it is left blank. The result of this procedure is a 2D array of pixels of which some pixels are coloured to approximate the mathematically described line. [3]

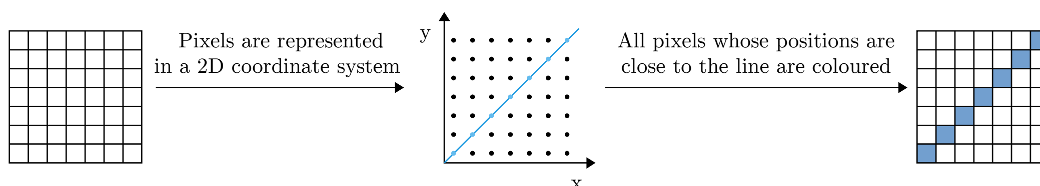


Figure 2.1: Explanation of 2D line rasterisation

Rasterisation of Solid Triangles

Rasterising solid triangles to a 2D array of pixels can be done by colouring all pixels that are contained within a mathematically described triangle. The studies performed in [4] and [5] used edge functions to describe triangles that should be rasterised to a 2D array of pixels. An edge function, as represented by Equation 2.1, is used to determine if a position is on one side of a straight line, the other side of it, or on top of it [5]. As shown in Figure 2.2 a), if the position is in the area that is in the direction of $\vec{n}$, the outcome of the edge function will be larger than zero. If the position is on the line itself or the other side of the line, the outcome

will be zero or smaller than zero, respectively. As shown in Figure 2.2 b), if three lines intersect to form a triangle, the three edge functions corresponding with those lines can be evaluated to determine if a position is within the triangle or not. If for a specific position, the outcome of all three of those edge functions is larger than zero, it can be determined that the position is within the triangle.

Figure 2.2 c) shows how the method that [4] and [5] used to describe triangles for rasterisation can be simplified to describe right-angled triangles. Since the edge functions corresponding with the vertical and horizontal edges of the triangle are in parallel with the vertical and horizontal axes of the coordinate system, respectively, those do not need to be evaluated. Thus, only one edge function, the one that corresponds with the diagonal line of the triangle, needs to be evaluated to determine if a position is within the triangle or not. However, only positions that are within a specific horizontal range, dx, and vertical range, dy, can be evaluated. This method can be used to reduce the number of calculations that needs to be performed when only required to rasterise right-angled triangles.

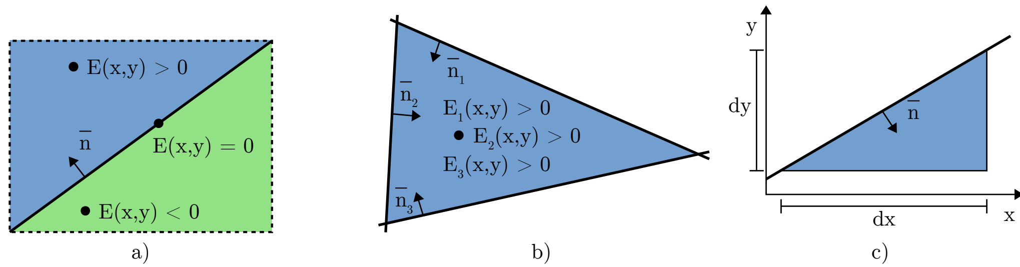


Figure 2.2: Explanation of an edge function, a), a triangle formed by three edge functions, b), and a right-angled triangle formed by one edge function

$$E(x, y) = \vec{n}.(x, y) + c \tag{2.1}$$

Where:
(x, y) = position that is evaluated
n = vector indicating a direction perpendicular to the line
c = constant determined by line properties

Figure 2.3 shows how a solid triangle, that is described by edge functions, is rasterised to a 2D array of pixels. The pixels are assigned with positions of the coordinate system that the edge functions are in. Each pixel's position can then be evaluated by means of the edge functions to determine if that position falls within the triangle or not. If a pixel's position falls within the triangle, that pixel should be coloured; otherwise, it should be left blank. The result is a 2D array of pixels of which some pixels are coloured to approximate the triangle that is described by the edge functions. [4], [5]

Rasterisation of Solid Ellipses

Figure 2.4 shows how a solid ellipse is rasterised to a 2D array of pixels. This ellipse is located in a 2D coordinate system and is described by Equation 2.2 [3]. Each pixel is assigned with a position that can be evaluated by this equation to determine if that position is within the ellipse or not. If for a specific pixel's position, the

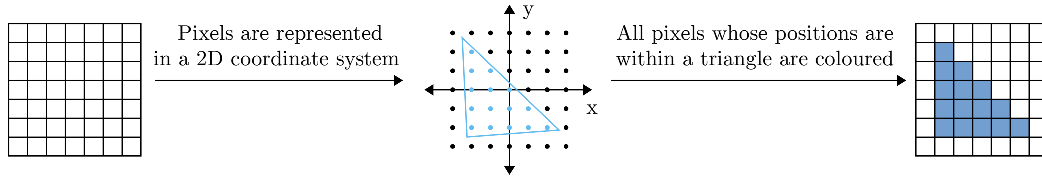


Figure 2.3: Explanation of 2D solid triangle rasterisation

outcome of the equation is smaller than or equal to zero, it can be determined that the position is within the ellipse or on its boundary, respectively. Thus, that pixel is provided with a colour. If the outcome of the equation is larger than zero, it can be determined that the position is outside of the ellipse, and thus, that pixel is left blank. The result is a 2D array of pixels of which some pixels are coloured to approximate the mathematically described ellipse. [3]

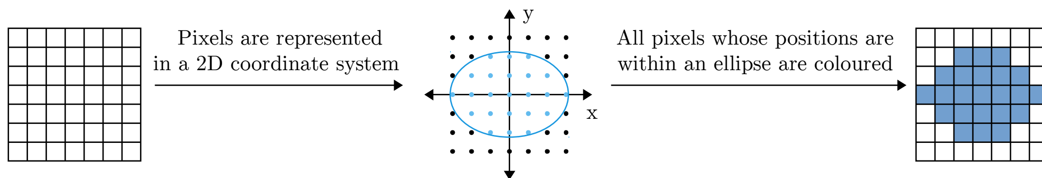


Figure 2.4: Explanation of 2D solid ellipse rasterisation

$$f(x, y) = x^2b^2 + y^2a^2 - a^2b^2 \tag{2.2}$$

Where:
(x, y) = position that is evaluated
a = horizontal radius of the ellipse
b = vertical radius of the ellipse

2.2 Graphics Rendering on FPGAs

2.2.1 Software-Intensive Graphics Rendering

A software-intensive graphics rendering approach is demonstrated by the example design in Figure 2.5 [6]. [6] implemented a graphics rendering system on an Intel, formerly Altera, Cyclone FPGA. The main component of this approach is the NIOS II embedded processor which renders an image, to display on a liquid crystal display (LCD) module, to external memory via an SDR/DDR interface component. The rendered image is then accessed via the Graphics/LCD controller component so that it can be provided to the LCD module. This graphics rendering system can be controlled from an external central processing unit (CPU) via the host interface component.

The first problem with this approach is that external memory is required to store frame buffer data, or the frame buffer data needs to be stored inside the FPGA, and thus, limits the use of MicroGE to high-resource FPGAs. The second problem is that, since soft-core processors execute instructions sequentially, all rendering

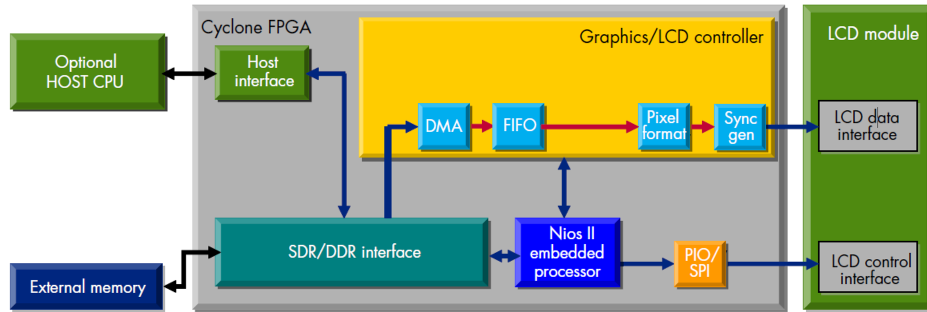


Figure 2.5: Example of how a graphics rendering system was implemented on an Intel (Altera) FPGA [6]

operations need to be performed by a sequence of sequential instructions. In the rendering algorithms discussed in Section 2.1.3, each pixel within a 2D array needs a specific number of operations to be performed to determine if that pixel should be provided with a colour or not. Even though these operations are simple and can be implemented by a small number of instructions, when rendering a large 2D array of pixels, a long sequence of sequential instructions needs to be performed to render a single graphics primitive.

2.2.2 Hardware Accelerated Graphics Rendering

An alternative to software-intensive graphics rendering is a hardware accelerated graphics rendering approach, such as the one shown in Figure 2.6 [7]. [7] implemented a graphics rendering system on a Xilinx Virtex-II Pro FPGA. The main component of this system is a hard CPU intellectual property (IP) core, the PowerPC IP Core, which interfaces with a set of peripheral components via the Processor Local Bus (PLB). The CPU collaborates with its peripheral components to render an image and store it in external system memory via the DDR Memory Controller IP Core. This image is then provided to a display monitor via the Xilinx LCD/VGA IP Core. Instead of relying on the CPU to perform all rendering operations, like [6], the CPU offloads the rendering operations to the Bresenham IP Core and the BitBLT IP Core. These components are hardware modules which were designed to accelerate graphics rendering. The CPU simply provides information, specifying what to render, to these hardware modules so that they can execute rendering operations in a much shorter time period. This not only increases the speed of rendering operations, but it also allows the CPU to perform other computations while the rendering operations are taking place.

The studies performed in [8] and [9] both emphasised the advantages of hardware accelerated graphics rendering over software-intensive graphics rendering. They replaced existing software rendering methods with hardware accelerated rendering methods and then analysed the differences in performance. [8]'s goal was to improve the rendering functions of a Linux distribution. This Linux distribution, called SnapGear, was implemented on an FPGA design that incorporated a soft-core processor. It was found that all of SnapGear's rendering functions were performed by software algorithms which only made use of the soft-core processor. Hardware components capable of accelerating these software algorithms were then developed and added to the FPGA design. These hardware components not only increased

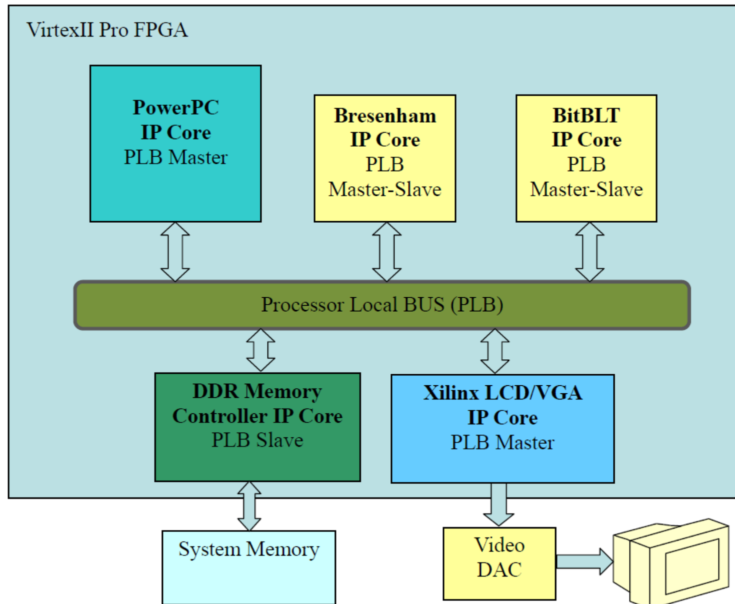


Figure 2.6: Example of how a graphics rendering system was implemented on a Xilinx FPGA [7]

the performance of the rendering functions, but also relieved the processor of other computations, and thus, increased the performance of the overall system. [9]'s goal was to develop a graphics acceleration component on an FPGA which increased the speed of vector rendering algorithms. These algorithms were previously performed by a compiler-optimised software component. They measured the performance of these two components by the number of paths, such as lines, curves and arcs, which could be processed per second. Their results, as shown in Figure 2.7, demonstrate how much hardware accelerated graphics rendering components can improve the performance of a graphics rendering system.

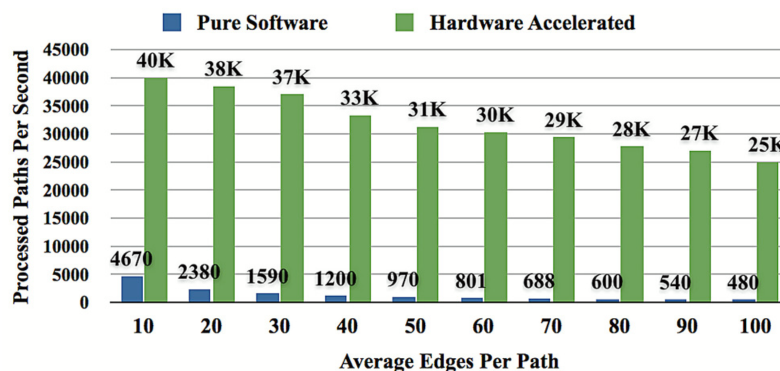


Figure 2.7: Comparison of hardware accelerated graphics rendering to pure software rendering in the study performed by [9]

Hardware accelerated graphics rendering was pursued since many studies, such as [7], [8] and [9], have shown that a significant amount of graphics rendering processing can be offloaded to dedicated graphics acceleration components. If a large enough amount, potentially all, of the graphics rendering processing could be offloaded to MicroGE, it would allow even a low-performance MCU, such as an Arduino Due, to

render graphics to HDMI and DVI display monitors, as required by this study.

2.2.3 Parallel Processing in Hardware Accelerated Graphics Rendering

Graphics rendering can be accelerated by hardware components because those components can be designed to perform a specific rendering operation very efficiently. This efficiency can be achieved by parallel processing techniques. Instead of performing a set of rendering operations sequentially, they can all be executed at the same time. In the rendering algorithms discussed in Section 2.1.3, the computations that need to be performed for each pixel within a 2D array can be performed at the same time, instead of sequentially. This can lead to a very significant decrease in the time required to render graphics primitives. However, the implementation of parallel processing techniques requires more FPGA resources and may increase the cost of a graphics rendering system. Thus, when taking the cost and FPGA resource restrictions of a design into account, there is a limit to how much parallelism will be able to improve the performance of a graphics rendering system.

An example of a graphics rendering system that uses parallel processing techniques to render graphics is shown in Figure 2.8 [10]. This graphics rendering system contains a 2D graphics acceleration core named BitSim Accelerated Display Graphics Engine (BADGE). BADGE is controlled by a host component such as a soft-core processor. It renders an image to an external memory component, such as SDRAM, via a memory interface component. This image is then provided to a display monitor by the display controller component. BADGE implements parallel processing techniques by rendering this image by means of several Graphics Processing Units (GPUs). Each of these GPU components is designed to perform a different rendering operation. Since a GPU only needs to perform a specific rendering operation, it can be designed to perform that rendering operation as fast as possible. When different types of rendering operations need to be performed to produce an image, each of these types can be provided to its dedicated GPU component. These different rendering operations can then be performed efficiently, and in parallel, to accelerate the overall graphics rendering process.

This concept would be useful in the design of MicroGE since different graphics acceleration components could be used to render different types of graphics primitives. Each of these components could then be optimised to render their graphics primitives as fast as possible. Also, multiple graphics acceleration components allow multiple graphics primitives to be rendered at the same time instead of having to use the same components to render a set of primitives sequentially. This would lead to a very significant improvement in the performance of MicroGE. A problem with increasing the number of graphics acceleration components is that the resource usage of MicroGE would increase as well. Thus, when considering the incorporation of parallelism in MicroGE, a trade-off would need to be made between MicroGE's performance and its low resource utilisation.

Parallel processing can also be achieved by pipelined architectures such as the one of the D/AVE graphics engine, [11]. [11]'s pipeline, as shown in Figure 2.9, consists of multiple graphics acceleration components. All of these components can operate on different parts of the graphics rendering process at the same time. Each of these components performs a specific operation based on information that was provided

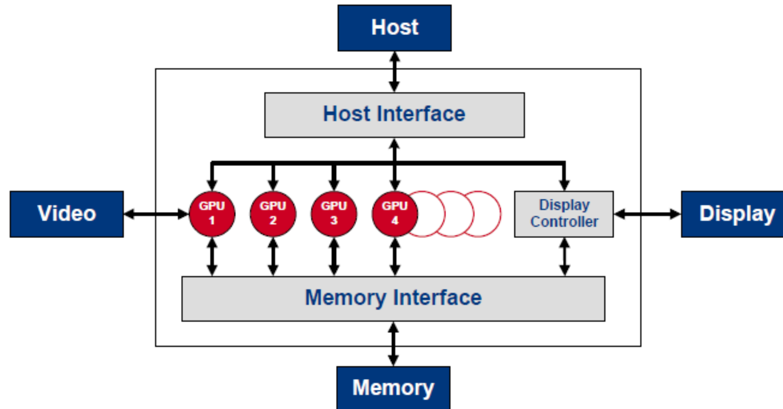


Figure 2.8: Block diagram of BADGE [10]

to it and then provides the output of that operation to another component in the pipeline shortly after. In the first part of [11]'s pipeline, the Display List Reader and Config/Status Controller components are provided with information specifying what should be rendered by the pipeline. This information is then provided to the following components of the pipeline, such as the Pixel Selection, Texture Unit and Color Unit, which collaborate to render parts of the video frame. A memory interface is provided to the Texture Unit so that it can read texture data that should be included in the video frame. There are also FIFO buffers between some of the components for synchronisation purposes. As the different parts of the video frame are rendered, they are eventually stored in a frame buffer by the Framebuffer Write component. Finally, the Blend Unit, Framebuffer Read and Framebuffer Cache components provide additionally rendering capabilities and performance improvements to the pipeline.

An advantage of this pipelined architecture over the architecture of [10] is that the number of hardware resources of the architecture can be decreased. Instead of providing an entire set of dedicated graphics acceleration components for each different rendering operation, some components can be shared among them. These different rendering operations can then access the same graphics acceleration components at different times. Thus, a pipelined architecture would allow MicroGE to achieve the best balance between performance and low resource utilisation.

2.2.4 Control of Graphics Rendering Systems

Since MicroGE may be controlled from a device with low processing power, such as an Arduino Due, MicroGE cannot rely on that device to contribute to any of the rendering operations. Instead, this device should only specify what MicroGE should render, by transferring instructions, while MicroGE uses hardware accelerated graphics rendering components to render the specified graphics. The faster these instructions can be transferred, the better the performance of MicroGE can be. However, since MicroGE's controller is a low-performance device, the controller will not be able to transfer these instructions quickly. Thus, even if MicroGE can render graphics quickly, this controller will limit the overall performance of MicroGE. One solutions to this problem is to keep the number of instructions that needs to be transferred to render a specific graphics primitive as small as possible. Another

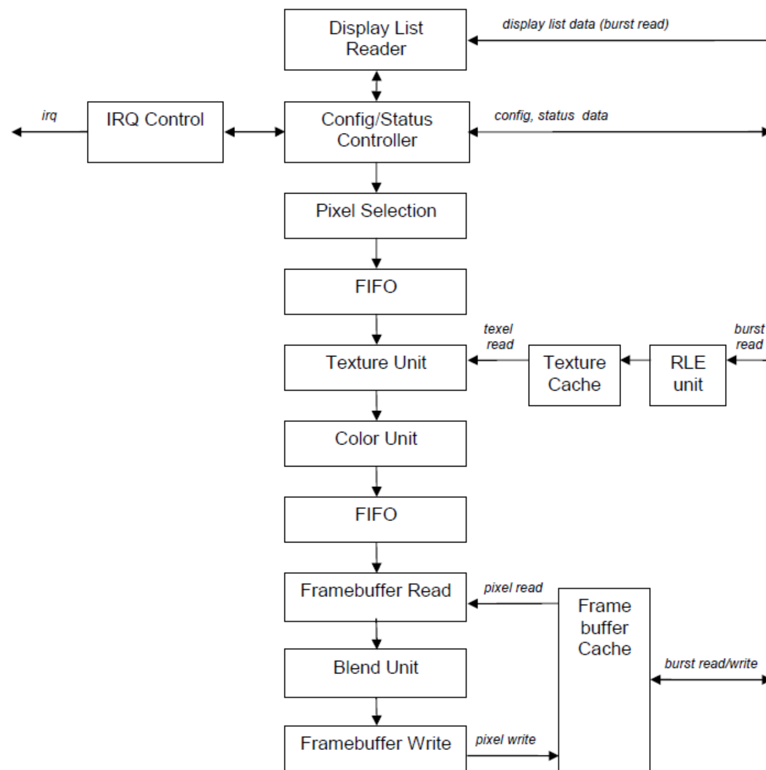


Figure 2.9: Block diagram of D/AVE 2D [11]

solution is to keep the size of each instruction a short as possible. This would decrease the time required for the controller to transfer instructions and would still allow MicroGE to achieve reasonable performance.

The study performed by [12] was able to implement a graphics rendering system on an Intel, formerly Altera, Cyclone II FPGA which could be controlled by a low-performance device such as an MCU. Even though this graphics rendering system was controlled by an MCU, it was still able to provide video output, with a video resolution of up to 640 × 480, to a VGA display monitor. This is a much lower video resolution than required by this study, but the same concept that [12] used to control this graphics rendering system can still be used by MicroGE. This graphics rendering system was controlled by 80-bit instructions which were transferred via a simple serial interface. Since these instructions are so short, they can be transferred to the graphics rendering system quickly. Also, since a simple serial interface is used, the MCU does not have to spend a significant amount of processing time to format data for transmission. Finally, these instructions only specify what should be rendered by the graphics rendering system. After an instruction is transferred, the graphics rendering system performs all of the required rendering operations by means of its hardware accelerated graphics rendering components. No software-intensive graphics rendering is performed by the MCU.

A concept used in the control of the Prevas graphics controller core can be used by MicroGE to reduce the length of the instructions that are transferred to control rendering operations [13]. Prevas is controlled by a set of high-level instructions of which one can, for example, indicate that a line should be rendered from one point on a screen to another. To render this line, only the high-level instruction needs

to be transferred. No information specifying the details of the graphics rendering process is provided. Since the high-level instruction does not need to contain a lot of information, it does not have to consist of a lot of data, and thus, can be transferred very quickly. After the instruction is transferred, Prevas uses a FIFO buffer to queue the instruction. This allows the controller of Prevas to send more instructions while Prevas is busy executing previous instructions. After the previous instructions are executed, Prevas will execute the new instruction by performing all of the functions required to render the line according to that instruction.

2.2.5 Graphics Rendering Without a Frame Buffer

Each of the graphics rendering systems that were discussed up to this point incorporates a memory component to store frame buffer data. The amount of memory required to store a frame buffer for a 1920 $\times$ 1080 video resolution is approximately 6 MB. Low-resource FPGAs, such as the Spartan-6 FPGAs, do not have enough memory resources to store a frame buffer for this video resolution [14]. Since it should be possible to implement MicroGE on a low-resource FPGA, without any external memory components, it is not possible for MicroGE to incorporate a frame buffer.

[15] designed a graphics rendering system that could provide graphics to a VGA display monitor without the need for any memory components to store frame buffer data. This graphics rendering system is integrated with the component that transmits video to the VGA monitor. This allows the rendering system to determine when each pixel of the video frame to display on the VGA monitor needs to be transmitted. Instead of storing an array of pixels in a frame buffer, the colour value of each pixel is generated right before transmission. As shown in Figure 2.10, [15] provides the horizontal and vertical coordinates of the pixels that should be transmitted to several hardware components. These hardware components, the Area Selector, Line Draw, Circle Draw and Text Draw components, collaborate to provide the colour value of the current pixel as output based on the current pixel coordinates and parameter values that are provided to them. Also, they all perform rendering operations in parallel by means of pure combinatorial logic. The Object Mixer and Colour Mapper evaluate the output of these components, also by means of combinatorial logic, and provide the final colour value of the pixel as output to the Video DAC so that it can be transmitted to a VGA monitor.

This type of rendering approach would allow MicroGE to be implemented within the required memory restrictions imposed by this study. On the other hand, since graphics would be rendered by means of combinatorial logic, it may be difficult for an FPGA design to be implemented within timing constraints of the Spartan-6 LX25 FPGA. Although, if some of the concepts of this rendering approach can be combined with a pipelined architecture, it may still be possible for MicroGE to be implemented within the imposed memory restrictions while meeting these timing constraints.

2.3 Video Transmission

Video transmission technologies are used to provide the data of rendered video frames to a display monitor so that those video frames can be seen by a user. One

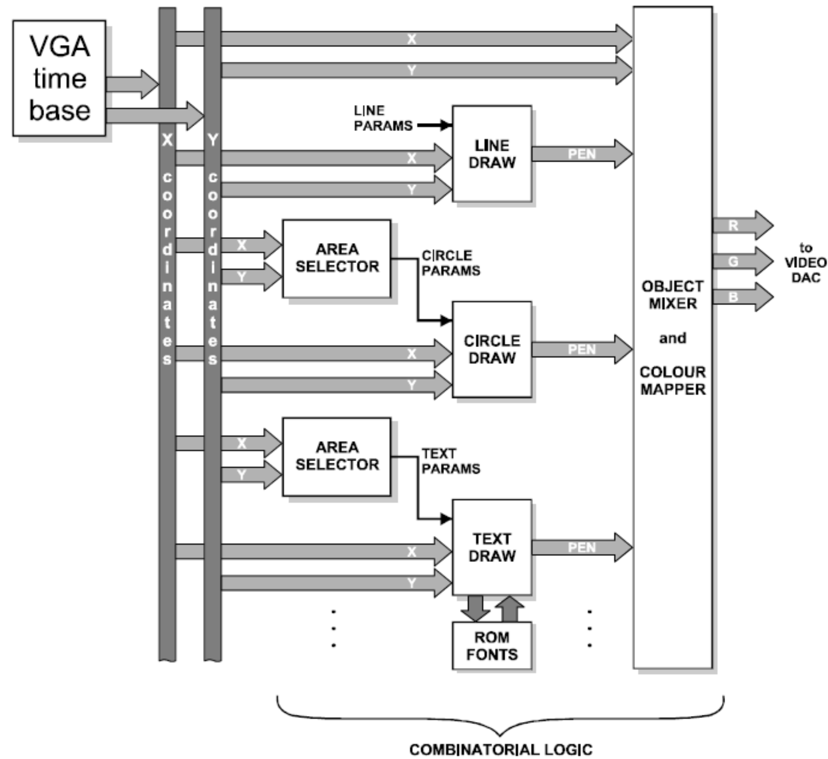


Figure 2.10: Block diagram of a graphics rendering system which does not incorporate a frame buffer [15]

of the first popular video transmission technologies was VGA, which used an analogue signalling standard to transfer video data to a display monitor. The use of this signalling standard greatly limited the maximum video resolution that could be achieved by VGA. DVI, which was later introduced, makes use of a digital signalling standard to transfer video data to a display monitor. This digital signalling standard allows much larger video resolutions to be achieved. HDMI uses the same digital signalling standard as DVI, but also allows additional information to be transferred along with video data, such as audio. DisplayPort, which was introduced after HDMI, is a royalty-free standard which also makes use of a digital signalling standard to transmit high-resolution video data along with additional information, such as audio. Even though DisplayPort offers advantages over other video transmission technologies, only HDMI and DVI were focused on in this study due to their popularity in the consumer market. [16], [17], [18]

2.3.1 TMDS

Both HDMI and DVI video transmitters transmit video data to display monitors via multiple Transition-Minimized Differential Signaling (TMDS) signal pairs. TMDS is a standard that specifies how data should be encoded and what physical interface to use when transmitting the encoded data. Octets are converted to 10-bit codewords which are serialised and then transmitted across twisted pair cables. The encoding is performed in a manner which minimises the number of transitions between ones and zeroes and maintains direct current (DC) balance in the transmitted signal content.

Minimising the number of transitions between ones and zeroes lowers the maximum frequency of the transmitted signal content, while maintaining DC balance in the transmitted signal content allows AC coupling of TMDS signals. Also, the Current-Mode Logic (CML) IO standard is used for transmission of the serialised data. [17]

2.3.2 HDMI and DVI Video Transmission

Figure 2.11 shows how the data of an HDMI video frame is encoded, how the encoded data is transmitted across multiple TMDS channels, and how the transmitted data is decoded. There are three data channels and one clock channel. Each data channel is used to transmit one of three colour components, such as RGB colour components, of a pixel. Control and auxiliary data are also transmitted across the data channels. The control data includes video synchronisation signals, while the auxiliary data includes packet headers and audio data. An HDMI video transmitter, or HDMI source, includes Encoder/Serialiser components which convert the pixel, control and auxiliary data to 10-bit codewords. These codewords are serialised and then transmitted to the Recovery/Decoder components in the display monitor, or HDMI sink, along with the clock signal, so that the original data can be recovered. DVI's video transmission and encoding/decoding scheme is similar to the one of HDMI, but no auxiliary data is included in the transmitted data. [17], [16]

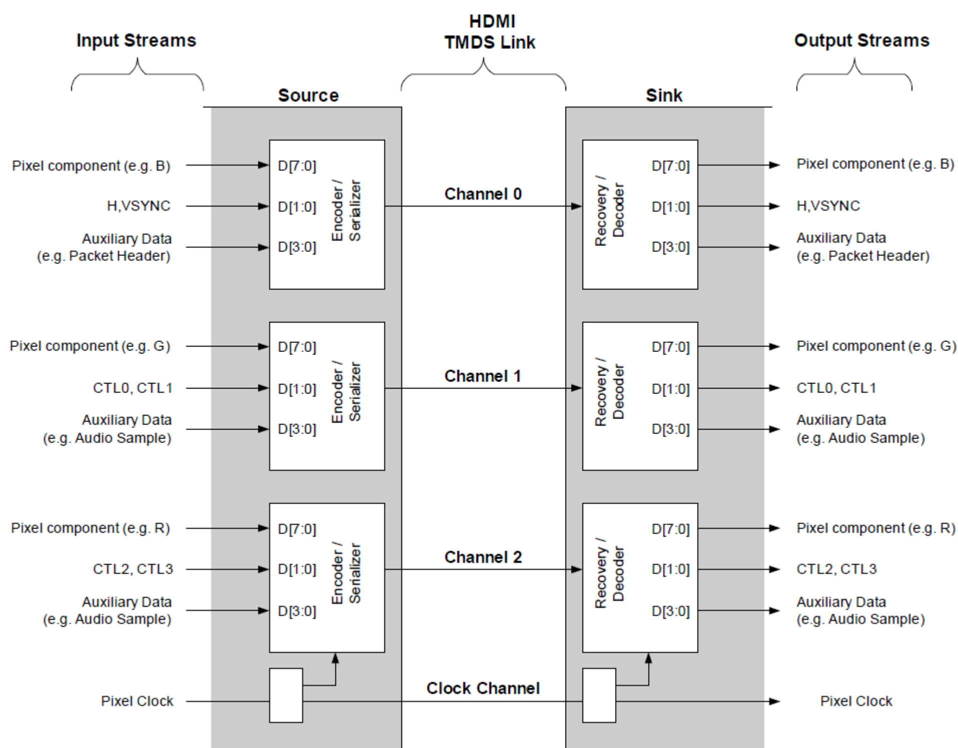


Figure 2.11: Block diagram describing the encoding, transmission and decoding of an HDMI video frame [17]

Figure 2.12 shows the structure of an HDMI video frame. The structure of a DVI video frame is similar except that no auxiliary data is contained within the frame. A display controller continuously transmits these frames to a display monitor, even if the frame content did not change. When a video mode has a 60 Hz refresh rate,

60 of these frames will be transmitted, consecutively, to the monitor each second. In the remaining part of this thesis, the time duration in which an HDMI video frame is transmitted will be referred to as an “HDMI video frame period”. [17], [16]

Each frame is divided into multiple lines which are transmitted consecutively, starting from the top line and progressing towards the bottom line. In the remaining part of this thesis, the time duration in which a line is transmitted will be referred to as a “line period”. Each line consists of a series of 10-bit codewords, per data channel, which can represent pixel characters, auxiliary characters or control characters. The characters of each data channel are transmitted one at a time, starting from the leftmost character and progressing towards the rightmost character. The lines can be classified as either vertical blanking lines or active lines. The vertical blanking lines can only contain auxiliary and control characters while the active lines can contain pixel, auxiliary and control characters. Auxiliary characters, as represented by the dark blue segments of the line periods in Figure 2.12, provide additional information along with the frame, such as audio data. Control characters are transmitted when the hsync and vsync signals, as shown in Figure 2.12, are asserted to the video transmitter. When the hsync signal is asserted, specific control characters, which indicate the beginning of a new line period, are transmitted. When the vsync signal is asserted, specific control characters, which indicate the beginning of a new HDMI video frame period, are transmitted. Furthermore, throughout the active line periods, the rows of pixels that should be shown on the display monitor are transmitted consecutively, starting from the top row and progressing towards the bottom row. Also, the pixels of a these rows are transmitted consecutively, starting from the leftmost pixel and progressing towards the rightmost pixel. In the remaining part of this thesis, this entire frame will be referred to as the “HDMI video frame” while the active part containing the actual video frame to show on the display monitor will be referred to as just the “video frame”. [17], [16]

2.3.3 Video Modes

The structure of the HDMI video frame in Figure 2.12 can be described by a set of parameters. These parameters specify the number of lines of the frame, the number of active and vertical blanking lines, the number of character to transmit during each line period, the number of pixels to transmit during an active line period, and when the hsync and vsync signals should be asserted. This set of parameters, along with the pixel clock frequency used to transmit the characters of the video frame, is referred to as a video mode. Different video modes provide different video resolutions as well as different frame update rates. The video modes that were used in this study are standardised by the Video Electronics Standards Association (VESA) and the Consumer Electronics Association (CEA). CEA video modes have names such as “1080p”, “1080i” and “720p”, while VESA video modes have names such as “800 × 600 @ 60Hz” and “1024 × 768 @ 60Hz”. [19], [20]

2.3.4 EDID and DDC

HDMI and DVI display monitors follow the Extended Display Identification Data (EDID) or Enhanced EDID (E-EDID) standards to provide their display capabili-

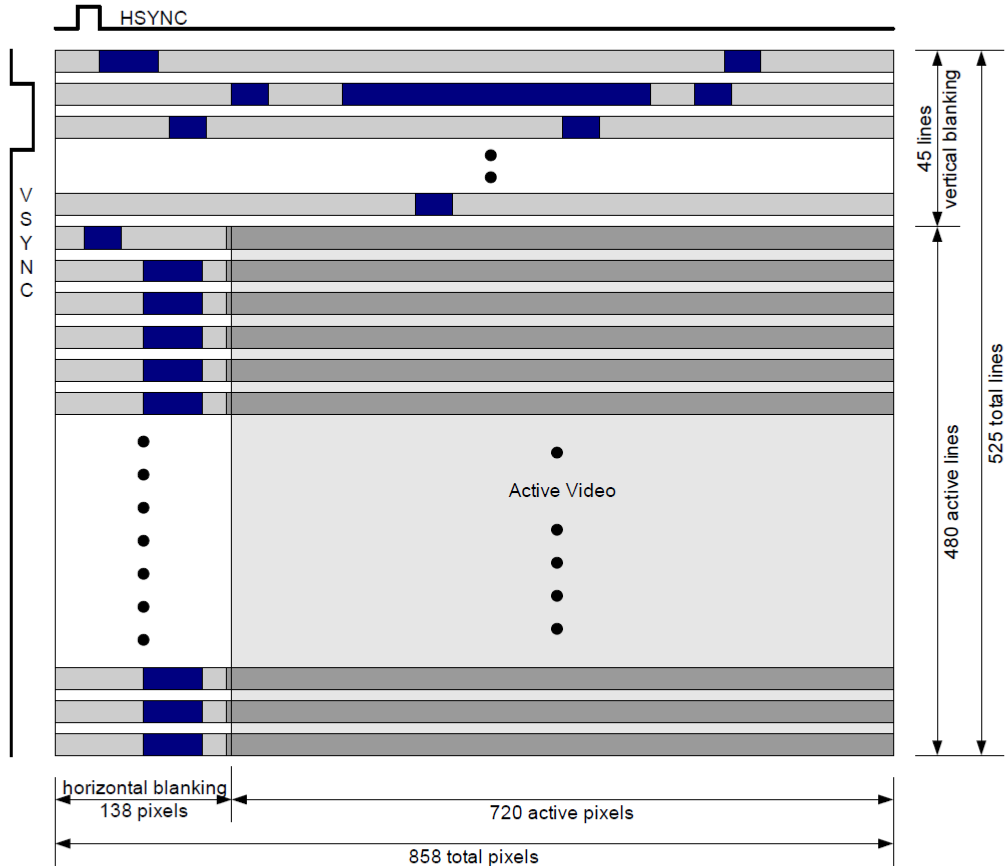


Figure 2.12: Example of the TMDS periods in the video frame of one of the HDMI video modes [17]

ties to display controllers. These standards specify a data structure in which display capability data should be stored and also specifies that this data should be stored in a read-only memory (ROM) IC, within the display monitor, so that it can be accessed via an HDMI or DVI interface. Once a display monitor is plugged into a display controller, the display controller reads the ROM data within the monitor to determine its display capabilities. An EDID ROM stores a monitor's display capabilities as a 128-byte data structure, called a data block, which describes capabilities such as supported VESA video modes. An E-EDID ROM also contains this 128-byte data structure, but includes additional storage capacity to store multiple 128-byte data block extensions. This allows a monitor to provide more information about its display capabilities. A common 128-byte data block extension is the CEA extension which allows a monitor to indicate support for CEA video modes, such as 1920p, 1920i and 720p. [20], [21]

A display controller needs to follow the Display Data Channel (DDC) or Enhanced DDC (E-DDC) standards to read a monitor's EDID or E-EDID ROM data, respectively. These standards specify the use of an Inter-Integrated Circuit (I^2C) bus for transferring data from an I^2C slave, either an EDID or E-EDID ROM, to an I^2C master within the display controller. With DDC, the I^2C master transmits an 8-bit address to the slave, the EDID ROM, so that the slave can reply with the data stored at that address. This 8-bit address space only allows 256 bytes to be

read, and thus, does not allow more than two consecutive 128-byte data blocks to be accessed. E-DDC is similar to DDC but uses an E-EDID ROM instead which can store more than two 128-byte data blocks. A segment pointer register within the E-EDID ROM, which can also be accessed via the I^2C bus, is written to for specifying which of these 128-byte data blocks to access via the 8-bit address space. [21], [22]

2.4 Hardware Design Theory

2.4.1 PCB Trace Length Matching

An electric signal can be analysed as an electric field that propagates along two conductors, such as a PCB trace and a ground plane. The speed at which the signal travels, calculated by Equation 2.3, is determined by the speed of light as well as the dielectric constant of the space around the two conductors [23]. With a microstrip, this space includes the air around and the PCB material between the microstrip and the ground plane. The air and PCB material have different dielectric constants. Equation 2.4 represents these two dielectric constants as a single effective dielectric constant [23].

$$v = \frac{c}{\sqrt{\epsilon_r}} \tag{2.3}$$

Where:
v = propagation speed of an electric signal
c = speed of light in a vacuum
ϵ_r = dielectric constant

$$\epsilon_{eff} = \frac{\epsilon_r + 1}{2} + \frac{\epsilon_r - 1}{2} \times \frac{1}{\sqrt{1 + 12\frac{h}{W}}}, \qquad when \quad \frac{W}{h} > 1 \tag{2.4}$$

Where:
W = width of microstrip
ϵ_{eff} = effective dielectric constant
ϵ_r = dielectric constant of material between microstrip and ground plane
h = height of material between microstrip and ground plane

Furthermore, since an electric signal travels from one point to another at a finite speed, there is a delay, called the propagation delay, between when the signal is provided from a source until when the signal reaches its destination. On a PCB that is fabricated with a specific dielectric material, the propagation delay is mainly dependant on the length of the PCB trace that the signal is propagating along. Each trace of a PCB has a resulting signal propagation delay from one component to another. When components on a PCB transfer data to each other by means of multiple PCB traces, the difference in the propagation delay among the transmitted signals, the timing skew, should be kept under a specified limit to ensure data integrity. This is achieved by keeping the difference in length among the PCB traces under a corresponding limit.

2.4.2 PCB Trace Impedance Matching

When transmitting high-speed signals across a relatively long PCB trace from a source to a load, transmission line effects need to be taken into consideration. To avoid signal distortion due to transmission line effects, the characteristic impedance of the PCB trace should be matched to the impedance of the load. Consider the cross-sectional view of a differential pair of PCB traces in Figure 2.13 which are used to carry high-speed signals from a source to a load. The differential and single-ended characteristic impedances of those two traces need to be designed so that they are equal to the differential and single-ended impedances of the load, respectively. This is done by selecting an appropriate dielectric constant for the material that is between the traces and the ground plane, and by selecting specific trace geometries. Equation 2.5 can be used to calculate the single-ended characteristic impedance of each of the two traces relative to the ground plane based on a specific dielectric constant and trace geometries [24]. Also, Equation 2.6 can be used to calculate the differential characteristic impedance of the pair of traces based on a specific dielectric constant and trace geometries [25]. When designing a PCB on which high-speed signals will be transmitted across traces, the parameters of these equations should be taken into consideration to ensure that the characteristic impedances of those traces are chosen correctly. [25]

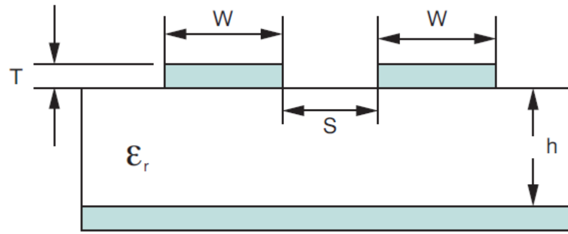


Figure 2.13: Cross-sectional view of differential PCB traces [25]

$$Z_0 = \frac{87}{\sqrt{\epsilon_r + 1.41}} \times ln\left(\frac{5.98h}{0.8W + T}\right) \tag{2.5}$$

Where:
Z_0 = single-ended characteristic impedance
T = trace thickness
W = trace width
ϵ_r = dielectric constant of PCB dielectric material
h = height of PCB dielectric material

$$Z_{DIFF} = 2Z_0(1 - 0.48 \times e^{-0.96\frac{s}{h}}) \tag{2.6}$$

Where:
Z_{DIFF} = differential characteristic impedance
Z_0 = single-ended characteristic impedance
s = distance between traces
h = height of PCB dielectric material

Chapter 3

Theory of Operation

3.1 Overview of MicroGE

3.1.1 Rendering Approach of MicroGE

The graphics rendering approaches of the researched graphics rendering systems, discussed in Section 2.2, are summarised in Figure 3.1 a) to c). All of these approaches use some type of controller to render a video frame that is shown on a display monitor. The differences between these approaches is in how the video frame is created. In the first approach, a), the controller renders the video frame to a frame buffer by means of sequential instructions, without any graphics accelerator. The video transmitter then transmits the video frame stored in the frame buffer to the display monitor. In the second approach, b), the controller provides instructions to a graphics accelerator that renders graphics to the frame buffer much faster than the controller alone would be able to. Similar to approach a), the video transmitter then transmits the video frame stored in the frame buffer to the display monitor. The last approach, c), differs from the first two approaches in that it does not use a frame buffer. Instead, its graphics accelerator uses combinatorial logic to generate each pixel of the video frame before it is transmitted to the display monitor.

Because of the requirements listed in Section 1.4, these approaches cannot be used for the design of MicroGE. Approaches a) and b) both require the incorporation of a frame buffer. The memory capacity required to store a frame buffer for a 1920 $\times$ 1080 video resolution is approximately 6 MB. This greatly exceeds the memory restriction, 1 Mb of RAM resources, imposed on the design of MicroGE. Additionally, since approach a) uses sequential instructions for all rendering operations, it would greatly limit MicroGE's performance. Even though approach c) would allow MicroGE to be implemented within the imposed memory restriction, the use of combinatorial logic to implement rendering operations would make it difficult to implement MicroGE within the timing restrictions of an FPGA.

As shown in Figure 3.2, MicroGE uses a different rendering approach that allows it to be implemented within all of the restrictions listed in Section 1.4. This approach also contains a controller that sends instructions to a graphics accelerator capable of performing rendering operations much faster than the controller alone would be able to. This approach also contains a video transmitter that reads pixels of a video frame and then transmits them to a display monitor. On the other hand, instead of using a frame buffer that stores the entire video frame, like the approaches in Figure

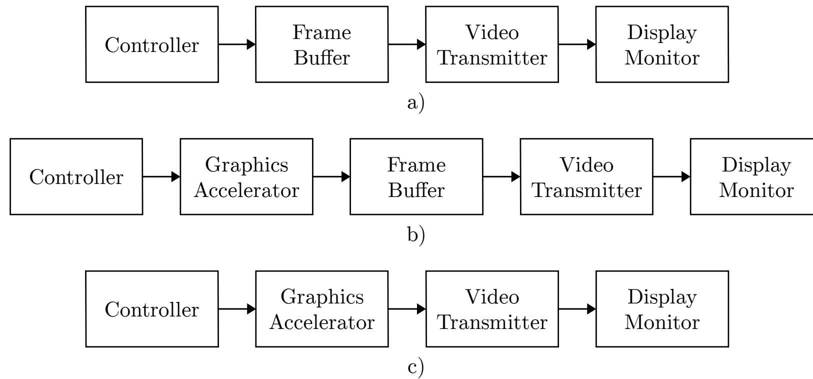


Figure 3.1: Block diagrams summarising the rendering approaches of the researched graphics engines

3.1 a) and b), it uses two line buffers. These line buffers are used to store lines of pixels of the video frame before the video transmitter transmits those lines to the display monitor. Also, unlike the approach in Figure 3.1 c), the graphics accelerator does not use combinatorial logic to render single pixels of the video frame before they need to be transmitted. Instead, pipeline mechanisms are used to render single lines of the video frame before the pixels of those lines need to be transmitted to the display monitor.

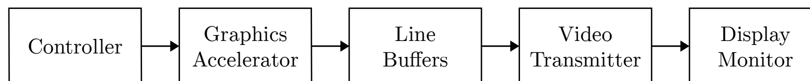


Figure 3.2: Block diagram summarising the rendering approach of MicroGE

3.1.2 Implementation of MicroGE's Rendering Approach

Figure 3.3 shows how the different roles of MicroGE's rendering approach are fulfilled. The role of the controller is fulfilled by the Arduino Due, which uses functions of MGAPI, to control MicroGE. The roles of the graphics accelerator and line buffers are fulfilled by MicroGE itself. Thus, one way of describing MicroGE is a graphics accelerator, or graphics engine, which also incorporates two line buffers. MicroGE does not include the video transmitter component, but rather interfaces with an external video transmitter that is implemented on the same FPGA as MicroGE. This video transmitter can provide video output to both HDMI and DVI display monitors. The reason for excluding the video transmitter will be motivated in Section 4.1.

3.2 Primitives

3.2.1 Primitive Attributes

MicroGE's rendered output is produced from a variety of basic shapes and images, called "primitives", which may be combined to create more detailed graphics. 1000 of these primitives can be rendered at the same time. Each primitive's position,

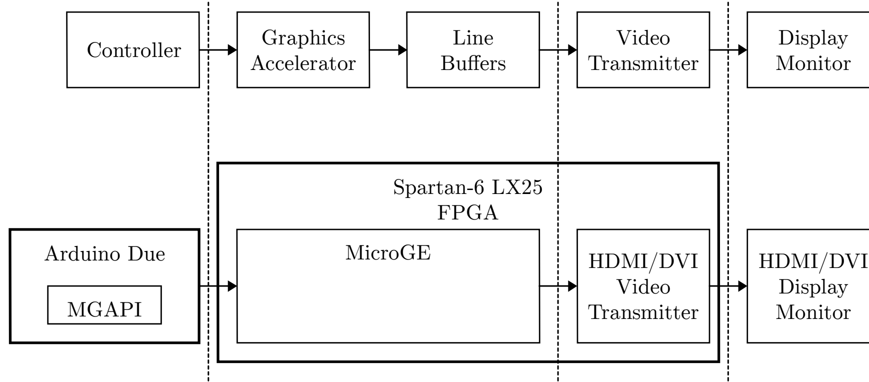


Figure 3.3: Block diagram describing how the different roles of MicroGE's rendering approach are fulfilled

size and pixel content are described by a set of attributes, which are listed in Table 3.1. The *id* attribute is used to provide a unique value to a primitive so that it can be identified among all others primitives. Figure 3.4 describes how a primitive, with an *id* attribute of n, is rendered from its attributes. It is positioned within a rectangular bounding box, referred to as the "primitive boundary", or just the "boundary", that has initial, (x_i, y_i), and final, (x_f, y_f), positions in a 2D Cartesian coordinate system. How a primitive is rendered within its boundary is dependent on the primitive's type, t, address, *adr*, and colour, *clr*, attributes. Finally, the depth layer, z, attribute specifies whether the primitive should be rendered in front or behind other primitives that may overlap it.

Attribute	Description
id	Value used to locate the primitive
x_i	Horizontal component of the initial position of the primitive
x_f	Horizontal component of the final position of the primitive
y_i	Vertical component of the initial position of the primitive
y_f	Vertical component of the final position of the primitive
z	Depth layer of the primitive
t	The primitive type
adr	Address of the primitive's pixel data
clr	Colour of the primitive (described by 8-bit RGB colour values)

Table 3.1: Description of the primitive attributes

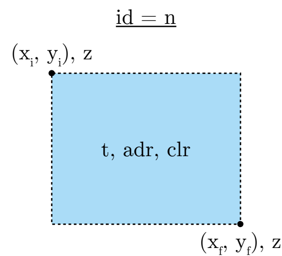


Figure 3.4: Description of how a primitive is rendered from its attributes

3.2.2 Vector and Raster Primitives

MicroGE's primitives are classified as either raster or vector primitives. Raster primitives' boundaries are filled with pixels that are retrieved from memory within MicroGE. These primitives use the *adr* attribute to indicate the locations of these pixels. Vector primitives are rasterised according to mathematical equations. The parameters of these equations are determined from the type, t, and position, x_i, y_i, x_f, y_f, attributes. Also, the colour of the vector primitives is determined by the *clr* attribute.

3.2.3 Vector Primitive Types

MicroGE has eight vector primitives. Figure 3.5 shows how these primitives are rendered. The first two primitives, a) and b), are lines that are rendered between two opposite corners of the primitive boundary. These primitives are called the ascending and descending line primitives, respectively. The next four primitives, c) to f), are right-angled triangles whose diagonal edges pass through opposite corners of the primitive boundaries. These primitives are called the top-left, top-right, bottom-left and bottom-right fill triangle primitives, respectively. Even though only right-angled triangles can be rendered, any type of triangle can be formed by combining a set of these right-angled triangles. The next primitive, g), is a filled ellipse which is positioned in the centre of the primitive boundary. The last primitive, h), is a rectangle that simply fills the entire primitive boundary.

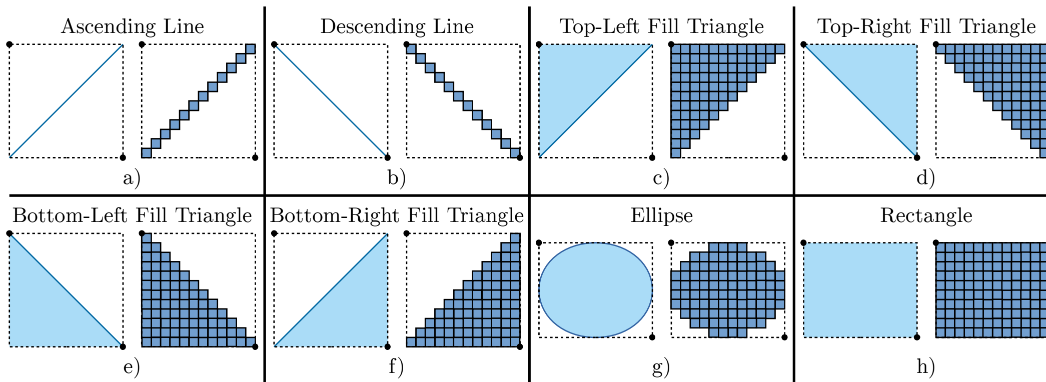


Figure 3.5: Illustration of the vector primitives

3.2.4 Raster Primitive Types

Figure 3.6 shows how MicroGE's two raster primitives are rendered. Both of their boundaries are filled with pixels retrieved from memory within MicroGE, called the raster memory. These pixels are grouped in n × n arrays of pixels called raster blocks. The first raster primitive, the image primitive, is rendered by filling its boundary with an image. This image is stored in the raster memory as a merged set of raster blocks. The string primitive is rendered from a set of raster blocks that are stored at arbitrary locations in the raster memory. When these raster blocks represent text characters, the string primitive may be used to render a string of text.

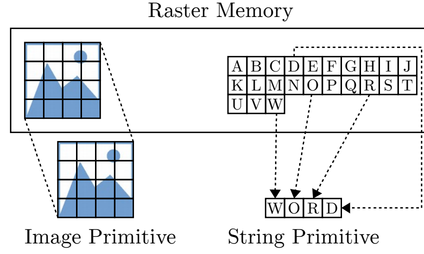


Figure 3.6: Illustration of the raster primitives

3.2.5 Depth Layers

MicroGE provides eight depth layers for rendering primitives over each other. The depth layers allow the creation of more detailed graphics since parts of primitives can overlap and smaller primitives can be placed on larger ones. The depth layer that a primitive should be rendered on is specified by its z attribute. As shown in Figure 3.7, pixels that are rendered on higher depth layers appear in front of pixels that are rendered on lower depth layers. MicroGE does not support transparency, and thus, pixels on the highest depth layer are chosen as part of MicroGE's final rendered output while pixels that fall below the highest ones are simply ignored.

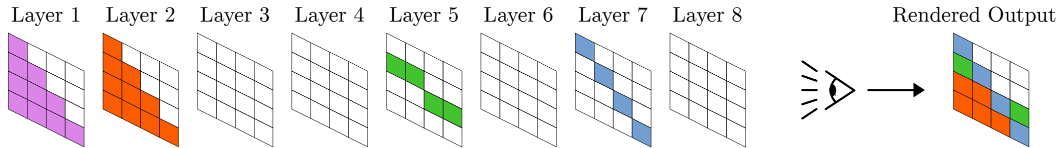


Figure 3.7: Description of the depth layers

3.3 Video Frame Organisation

3.3.1 Canvas

MicroGE renders the primitives to a 2D array of pixels, referred to as the "canvas". As shown in Figure 3.8, the canvas is transmitted to a display monitor as part of the active video part, referred to in this thesis as just the "video frame", of the overall HDMI video frame. The canvas can be positioned at different locations in the video frame and can also be set to eight different sizes. The aspect ratio of the canvas was chosen as 16:9 to match aspect ratios of popular video modes such as 1080i and 720p. For video resolutions with other aspect ratios, the canvas can be scaled and centred within the video frame to fill the video frame as well as possible.

3.3.2 Grid System

The canvas is rendered from primitives that are positioned in a 2D Cartesian coordinate system, referred to as the "grid system". As shown in Figure 3.9, the grid system is used to position the corners of the primitive boundaries. The origin of the grid system, (0, 0), is mapped to the top left corner of the canvas while the

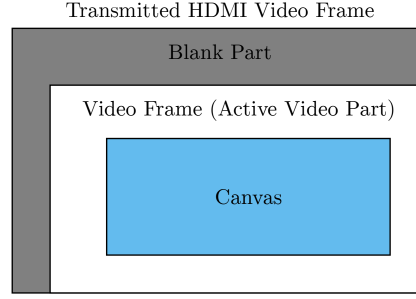


Figure 3.8: Explanation of how the canvas is transmitted along with the HDMI video frame

horizontal and vertical axes are mapped to the top and left edges of the canvas, respectively. The horizontal axis has 161 discrete positions, ranging from zero to 160, while the vertical axis has 91, ranging from zero to 90. The distance between these positions is referred to as "grid units". Also, a position within the grid system, (x_g, y_g), is referred to as a "grid position".

A primitive boundary has its own coordinate system, referred to as its "local grid system". Similarly, the origin of the local grid system, (0, 0), is mapped to the top left corner of the primitive boundary while the horizontal and vertical axes are mapped to the top and left edges of the boundary, respectively. A grid position within a primitive's local grid system is referred to as a "local grid position", (x_l, y_l), and is calculated by Equation 3.1. The length of the horizontal and vertical axes, in terms of grid units, is equal to the width, dx_g, and height, dy_g, of the primitive boundary, respectively, and is calculated by Equation 3.2.

To assist the discussion throughout the rest of the thesis, the following terms are defined to refer to areas within the global and local grid systems:

- A "global grid column" is an area with a width of one grid unit and a height of 90 grid units
- A "global grid row" is an area with a height of one grid unit and a width of 160 grid units
- A "local grid column" of a primitive's local coordinate system is an area with a width of one grid unit and a height of dy_g
- A "local grid row" of a primitive's local coordinate system is an area with a height of one grid unit and a width of dx_g
- A "grid block" is the area contained within four adjacent grid positions

$$\begin{bmatrix} x_l \\ y_l \end{bmatrix} = \begin{bmatrix} x_g \\ y_g \end{bmatrix} - \begin{bmatrix} x_\text{i} \\ y_\text{i} \end{bmatrix} \tag{3.1}$$

Where:

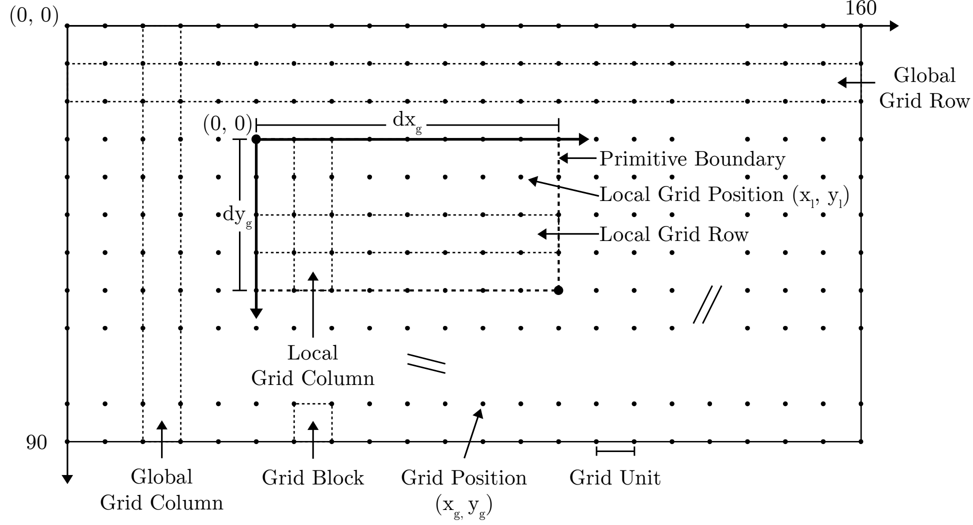


Figure 3.9: Description of the global and local grid systems

x_l = horizontal component of local grid position
y_l = vertical component of local grid position
x_g = horizontal component of global grid position
y_g = vertical component of global grid position
x_i = horizontal component of initial position of primitive
y_i = vertical component of initial position of primitive

$$\begin{bmatrix} dx_g \\ dy_g \end{bmatrix} = \begin{bmatrix} x_\text{f} \\ y_\text{f} \end{bmatrix} - \begin{bmatrix} x_\text{i} \\ y_\text{i} \end{bmatrix} \tag{3.2}$$

Where:
dx_g = width of primitive boundary
dy_g = height of primitive boundary
x_i = horizontal component of initial position of primitive
y_i = vertical component of initial position of primitive
x_f = horizontal component of final position of primitive
y_f = vertical component of final position of primitive

3.3.3 Scaling

MicroGE is able to render its canvas to different sizes depending on the value provided to its “PixelsPerGrid” property. This property determines how many pixels are rendered between parallel edges of a grid block. Figure 3.10 a) and b) shows how the ascending line primitive is rendered for PixelsPerGrid values of four and twelve, respectively. The two primitive boundaries have the same width and height in terms of grid units, but the one line is rendered with a higher resolution than the other. All primitives can be rendered to different resolutions without having to change their attributes. This allows the canvas to be rendered to different sizes, from the same set of primitives, without having to change the attributes of those primitives. Only the value of the PixelsPerGrid property has to be changed. Since

the lengths, in terms of grid units, of the horizontal and vertical axes of the canvas are 160 and 90, respectively, for a PixelsPerGrid property of eight, the resolution of the canvas will be 1280 × 720, that of the 720p video mode. Similarly, for a PixelsPerGrid property of 12, the resolution will be 1920 × 1080, that of the 1080p and 1080i video modes.

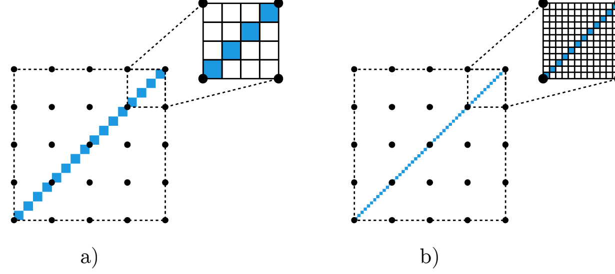


Figure 3.10: Explanation of the PixelsPerGrid property

3.3.4 Local Pixel Organisation

Figure 3.11 describes how a pixel's position is determined and how certain groups of pixels are addressed within a local coordinate system. The horizontal and vertical pixel positions of a pixel, x_{px} and y_{px}, are the horizontal and vertical distances between the top left corner of that pixel and the top left corner of the primitive boundary, respectively. These distances, calculated by Equation 3.3, are represented in pixel units, which is equal to the length of a pixel's edge. Also, the horizontal and vertical pixel offsets, dx_{px} and dy_{px}, are the horizontal and vertical distances, in pixel units, between the top left corner of that pixel and the top left corner of the grid block the pixel is within, respectively.

To assist the discussion throughout the rest of the thesis, the following terms are defined to refer to specific pixel groups:

- A "pixel row" is an adjacent set of pixels, which all have the same vertical pixel positions, spanning from the left to right edge of the primitive boundary
- A "pixel column" is an adjacent set of pixels, which all have the same horizontal pixel positions, spanning from the top to bottom edge of the primitive boundary

$$\begin{bmatrix} x_{px} \\ y_{px} \end{bmatrix} = P \times \begin{bmatrix} x_l \\ y_l \end{bmatrix} + \begin{bmatrix} dx_{px} \\ dy_{px} \end{bmatrix} \tag{3.3}$$

Where:

x_{px} = horizontal component of pixel position
y_{px} = vertical component of pixel position
x_l = horizontal component of local grid position
y_l = vertical component of local grid position
dx_{px} = horizontal pixel offset from top left corner of grid block
dy_{px} = vertical pixel offset from top left corner of grid block
P = value of PixelsPerGrid property

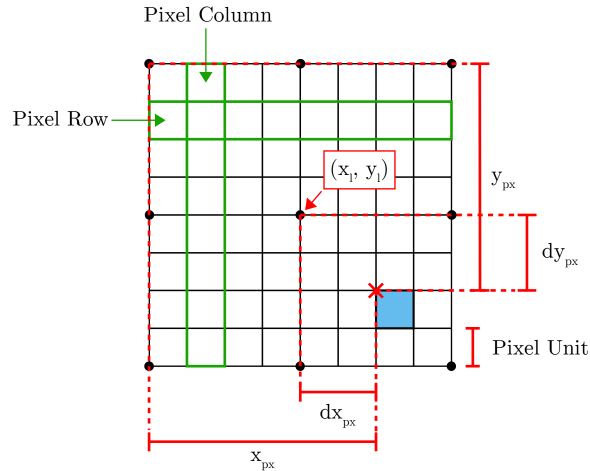


Figure 3.11: Description of the local pixel organisation

3.4 Rendering Procedure Overview

3.4.1 Line Buffers

MicroGE uses an unusual approach to produce its rendered output. Most of the graphics rendering systems discussed in Section 2.2 make use of frame buffers to store an entire video frame. In contrast, MicroGE only stores two rows of pixels of the canvas at a time, right before it is transmitted to the display monitor by the video transmitter component. These rows of pixels are referred to as "canvas rows". The advantage of storing only two canvas rows, as opposed to the entire canvas, is that much less memory is required to implement MicroGE. However, not storing the entire canvas requires each of the canvas rows to be reproduced every time the canvas is transmitted to the display monitor by the video transmitter component.

As shown in Figure 3.12, MicroGE uses a double buffering approach, which incorporates two line buffers, to store the rendered canvas rows before those rows are transmitted to the display monitor by the video transmitter component. Each canvas row is rendered and stored in one of the line buffers during one of the line periods of a video frame period. During the line period after a canvas row is rendered and stored in a line buffer, the canvas row is accessed from that line buffer, by the video transmitter component, and transmitted to a display monitor. MicroGE's rendering operations change their write access from one line buffer to another after each line period. Similarly, the video transmitter component changes its read access from one line buffer to another after each line period. This allows a canvas row to be read and transmitted from one line buffer while the following canvas row is rendered and stored in the other line buffer.

3.4.2 Rendering Canvas Rows

Figure 3.13 gives a simplified explanation of how one of the canvas rows is rendered so that it can be stored in one of the line buffers. Initially, the line buffer is empty. In the remaining part of this thesis, when no pixels of a canvas row are rendered yet, the canvas row will be referred to as a "blank" canvas row that contains a series

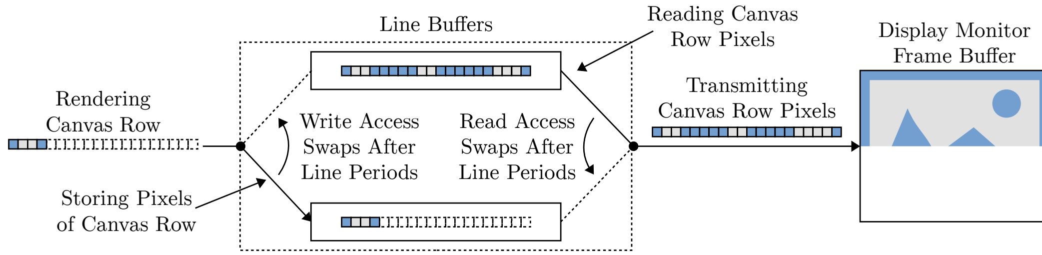


Figure 3.12: Simplified explanation of how the two line buffers are utilised

of "blank" pixels. To render the canvas row, each blank pixel of that canvas row is "visited", starting from the leftmost pixel and progressing towards the rightmost pixel. In the remaining part of this thesis, the process of visiting each blank pixel will be referred to as the "render position" that travels along the canvas row, starting from the leftmost pixel and ending at the rightmost pixel. When the render position visits a blank pixel, that pixel is provided with a colour. If the visited blank pixel is not within any primitive boundary, that pixel is provided with the background colour, black. On the other hand, if the visited pixel is within a primitive's boundary, that primitive's attributes are evaluated to determine the colour of that pixel.

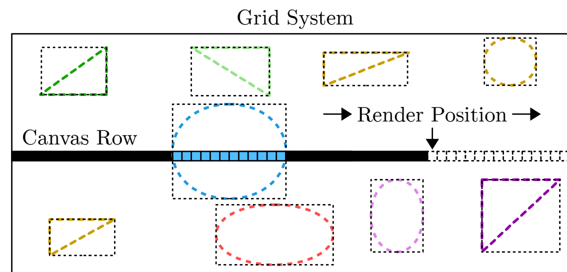


Figure 3.13: Simplified explanation of how a canvas row is rendered

3.5 Vector Rendering

3.5.1 Vector Primitives Rendering Procedures Overview

When the render position visits a blank pixel that is within a vector primitive's boundary, a series of operations are performed to determine that pixel's colour. These operations are based on the theory discussed in Section 2.1.3. First, the pixel position within the primitive boundary is calculated by means of Equation 3.3. The pixel position is then translated to a position in a different coordinate system, referred to as the primitive's "render coordinate system". The primitives have different types of render coordinate systems and also different coordinate transformation procedures. Each render coordinate system contains a mathematically expressed region, called the "render range". If the translated position, referred to as the "assessment position", falls within the render range, the visited pixel is provided with the colour specified by the primitive's *clr* attribute. On the other hand, if the assessment position falls outside of the render range, the visited pixel is provided with the background colour.

Figure 3.14 shows the render ranges and render coordinate systems of the vector primitives. An example of an assessment position that is mapped to each render coordinate system is also shown. Each primitive's render range is described by means of a mathematical equation. The two line primitives' render range is the area around a straight diagonal line, passing through the origin, that has a positive gradient. The four triangle primitives' render range is the area around, and under, a straight diagonal line, passing through the origin, that has a positive gradient. Finally, the ellipse primitive's render range is the area within an ellipse that is centred around the origin.

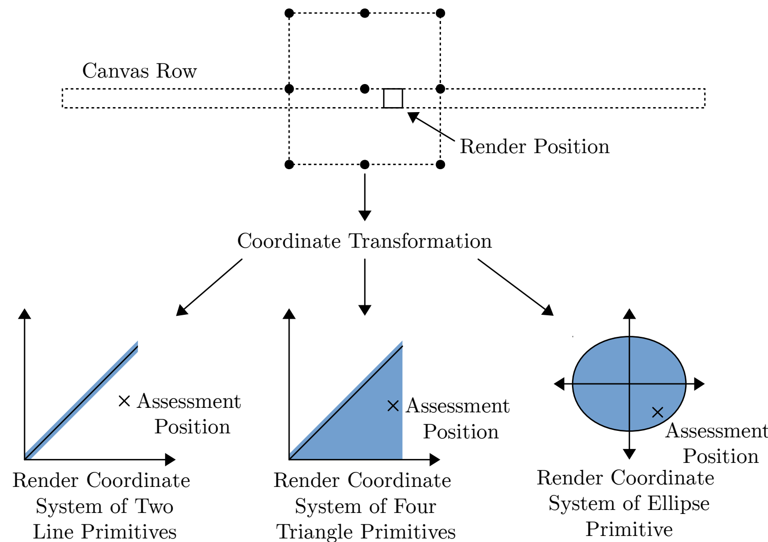


Figure 3.14: Description of the render ranges and render coordinate systems of the vector primitives

3.5.2 Line Primitives Rendering Procedure

Coordinate Transformation Procedures for Line Primitives

There are four different coordinate transformation procedures for the two line primitives. Since the gradient of the ascending line primitive is positive, while the gradient of the descending line primitive is negative, different coordinate transformation procedures are required to map those lines to a line with a positive gradient. Also, different coordinate transformation procedures are required depending on whether the absolute values of the lines' gradients are larger than one or not.

For each of these coordinate transformation procedures, different parts of the primitive boundary are mapped to different parts of the render coordinate system. Figure 3.15 shows an example of how the descending line primitive is mapped to its render coordinate system. One of the four corners of the primitive boundary is mapped to the origin of the render coordinate system while the two perpendicular edges, at that corner, are mapped to the horizontal and vertical axes of the render coordinate system. Furthermore, the lengths of pixel edges are represented by a length of one unit in the render coordinate system. Finally, the centre of the currently visited pixel is mapped to a corresponding assessment position.

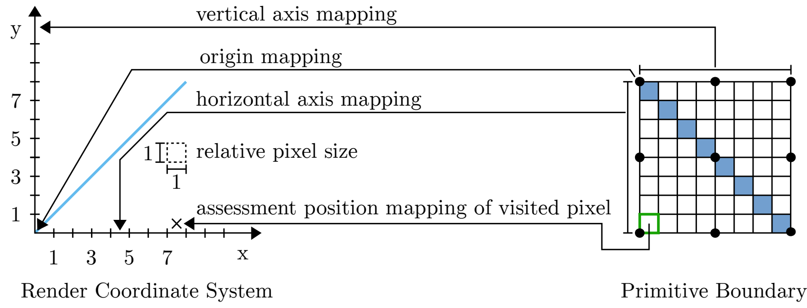


Figure 3.15: Example of one of the line primitives' coordinate transformation procedures

Figure 3.16 a) to d) shows the four different coordinate transformation procedures for the line primitives. The first two procedures, a) and b), are for two cases of the descending line primitive. In the first case, a), the absolute value of the gradient of the line to render within the primitive boundary should either be smaller than or equal to one while in the second case, b), it should be larger than one. The second two procedures, c) and d), are for two cases of the ascending line primitive. Similarly, in the first case, c), the absolute value of the gradient of the line to render within the primitive boundary should either be smaller than or equal to one while in the second case, d), it should be larger than one.

In each of these coordinate transformation procedures, a corner and two perpendicular edges of the primitive boundary are mapped to the origin and axes of the render coordinate system, respectively, as follows:

- In a), the top left corner is mapped to the origin while the top and left edges are mapped to the horizontal and vertical axes, respectively
- In b), the top left corner is mapped to the origin while the left and top edges are mapped to the horizontal and vertical axes, respectively
- In c), the top right corner is mapped to the origin while the top and right edges are mapped to the horizontal and vertical axes, respectively
- In d), the top right corner is mapped to the origin while the right and top edges are mapped to the horizontal and vertical axes, respectively

Equations 3.4, 3.5, 3.6 and 3.7 are used to perform the four different coordinate transformations procedures, a), b), c) and d), respectively, to translate the line primitives' pixel positions to assessment positions within the line primitives' render coordinate system.

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x_{px} \\ y_{px} \end{bmatrix} + \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix} \tag{3.4}$$

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} y_{px} \\ x_{px} \end{bmatrix} + \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix} \tag{3.5}$$

$$\begin{bmatrix} x \\ y \end{bmatrix} = \left(\begin{bmatrix} P \times dx_g \\ y_{px} \end{bmatrix} - \begin{bmatrix} x_{px} \\ 0 \end{bmatrix} \right) + \begin{bmatrix} -0.5 \\ 0.5 \end{bmatrix} \tag{3.6}$$

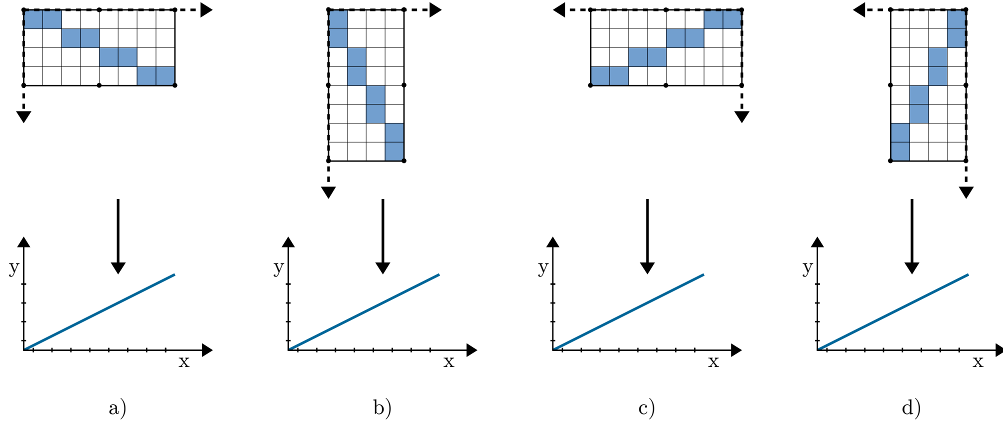


Figure 3.16: Description of the coordinate transformation procedures for the line primitives

$$\begin{bmatrix} x \\ y \end{bmatrix} = \left(\begin{bmatrix} y_{px} \\ P \times dx_g \end{bmatrix} - \begin{bmatrix} 0 \\ x_{px} \end{bmatrix} \right) + \begin{bmatrix} 0.5 \\ -0.5 \end{bmatrix} \tag{3.7}$$

Where:
x = horizontal component of assessment position
y = vertical component of assessment position
dx_g = primitive boundary width
x_{px} = horizontal component of pixel position
y_{px} = vertical component of pixel position
P = value of PixelsPerGrid property

Render Range of Line Primitives

Figure 3.17 shows the line primitives' render coordinate system and also shows examples of how the assessment positions are evaluated to determine if they fall within the render range or not. For purposes of explaining the render range, the visited pixels that were coloured as a result of these evaluations are overlaid on the render coordinate system. The render range is the area around an ascending diagonal line, passing through the origin, with a gradient smaller than or equal to one. This line is expressed by Equation 3.8. The gradient of the line is determined by the division of the lengths of the two primitive boundary edges that are mapped to the vertical and horizontal axes of the render coordinate system. The render range, expressed by Equation 3.9, includes all positions that are within a vertical range of 0.5 below and 0.5 above the line. The render range also includes positions that are exactly a distance of 0.5 above the line, since the line may pass exactly between two assessment positions. In such a case, the vertical distances between the line and each of the two assessment positions are both 0.5. Thus, to avoid rendering a discontinuous line, one of the assessment positions needs to be included in the render range.

$$y = \frac{dy}{dx}x \tag{3.8}$$

Where:
x = horizontal component of assessment position
y = vertical component of assessment position
$\frac{dy}{dx}$ = line gradient

$$-0.5 < y - \frac{dy}{dx}x \leq 0.5 \tag{3.9}$$

Where:
x = horizontal component of assessment position
y = vertical component of assessment position
$\frac{dy}{dx}$ = line gradient

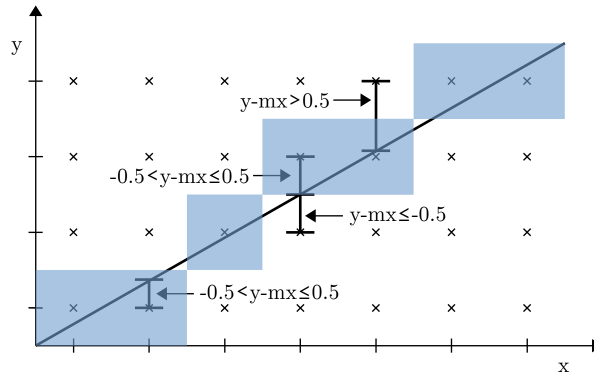


Figure 3.17: Description of the render range and render coordinate system of the line primitives, where $m = \frac{dy}{dx}$

3.5.3 Triangle Primitives Rendering Procedure

Coordinate Transformation Procedures of Triangle Primitives

The coordinate transformation procedures of the triangle primitives are based on the coordinate transformation procedures of the line primitives. The triangle primitives whose diagonal edges are descending, the top-right and bottom-left fill triangle primitives, use the coordinate transformation procedures, Figure 3.16 a) and b), of the descending line primitive. If either one of these triangle primitives' diagonal edge's gradient has an absolute value smaller than or equal to one, a) is used, otherwise, b) is used. The triangle primitives whose diagonal edges are ascending, the top-left and bottom-right fill triangle primitives, use the coordinate transformation procedures, Figure 3.16 c) and d), of the ascending line primitive. Similarly, if either one of these triangle primitives' diagonal edge's gradient has an absolute value smaller than or equal to one, c) is used, otherwise, d) is used.

The coordinate transformation procedures for both cases of each triangle primitive are shown in Figure 3.18. In each of these cases, a corner and two perpendicular edges of the primitive boundary are mapped to the origin and axes of the render coordinate system, respectively, as follows:

- In a), the bottom-right corner is mapped to the origin while the bottom and right edges are mapped to the horizontal and vertical axes, respectively

- In b), the bottom-right corner is mapped to the origin while the right and bottom edges are mapped to the horizontal and vertical axes, respectively
- In c), the bottom-left corner is mapped to the origin while the bottom and left edges are mapped to the horizontal and vertical axes, respectively
- In d), the bottom-left corner is mapped to the origin while the left and bottom edges are mapped to the horizontal and vertical axes, respectively
- In e), the top-left corner is mapped to the origin while the top and left edges are mapped to the horizontal and vertical axes, respectively
- In f), the top-left corner is mapped to the origin while the left and top edges are mapped to the horizontal and vertical axes, respectively
- In g), the top-right corner is mapped to the origin while the top and right edges are mapped to the horizontal and vertical axes, respectively
- In h), the top-right corner is mapped to the origin while the right and top edges are mapped to the horizontal and vertical axes, respectively

Render Range of Triangle Primitives

Figure 3.19 shows the triangle primitives' render coordinate system and also shows examples of how the assessment positions are evaluated to determine if they fall within the render range or not. Similar to Figure 3.17, the visited pixels that were coloured as a result of these evaluations are overlaid on the render coordinate system. The render range is the area around and under an ascending diagonal line, also expressed by Equation 3.8, that passes through the origin and has a positive gradient that is smaller than or equal to one. This area around and under the line, expressed by Equation 3.10, includes all positions that are within a vertical distance ranging from 0.5 above this line down to the horizontal axis of the render coordinate system.

$$y - \frac{dy}{dx}x \leq 0.5 \tag{3.10}$$

Where:
x = horizontal component of assessment position
y = vertical component of assessment position
$\frac{dy}{dx}$ = line gradient

3.5.4 Ellipse Primitive Rendering Procedure

Coordinate Transformation Procedure of Ellipse Primitive

In contrast to the line and triangle primitives, the ellipse primitive only uses one coordinate transformation procedure. This procedure maps parts of the primitive boundary to parts of the render coordinate system. As shown in Figure 3.20, the centre of the primitive boundary is mapped to the origin of the render coordinate system. The horizontal and vertical lines that pass through the centre of the primitive

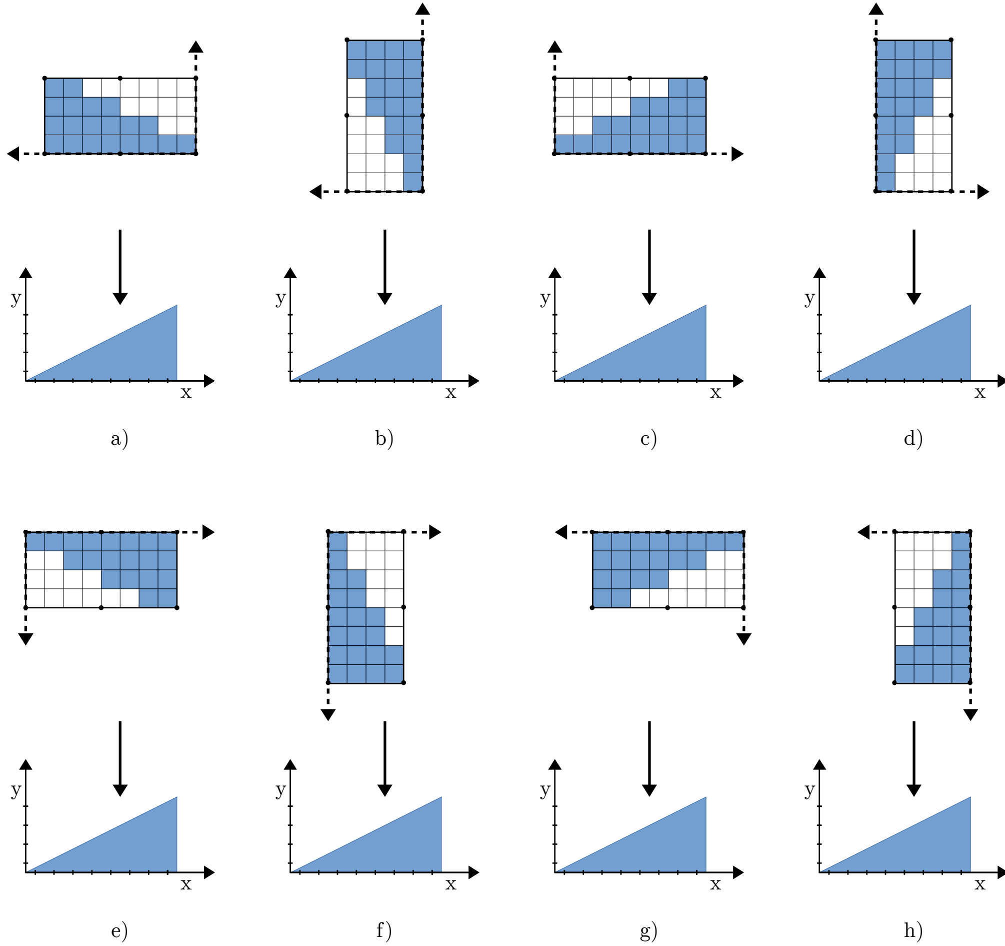


Figure 3.18: Description of the coordinate transformation procedures for the triangle primitives

boundary are mapped to the horizontal and vertical axes of the render coordinate system, respectively. Furthermore, the lengths of pixel edges are represented by a length of one unit in the render coordinate system. Finally, the centre of the currently visited pixel is mapped to a corresponding assessment position. Equation 3.11 is used to translate the pixel positions within the ellipse primitive's boundary to assessment positions within the render coordinate system.

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x_{px} \\ y_{px} \end{bmatrix} - \frac{P}{2} \begin{bmatrix} dx_g \\ dy_g \end{bmatrix} + \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix} \tag{3.11}$$

Where:
x = horizontal component of assessment position
y = vertical component of assessment position
dx_g = primitive boundary width
dy_g = primitive boundary height
x_{px} = horizontal component of pixel position
y_{px} = vertical component of pixel position
P = value of PixelsPerGrid property

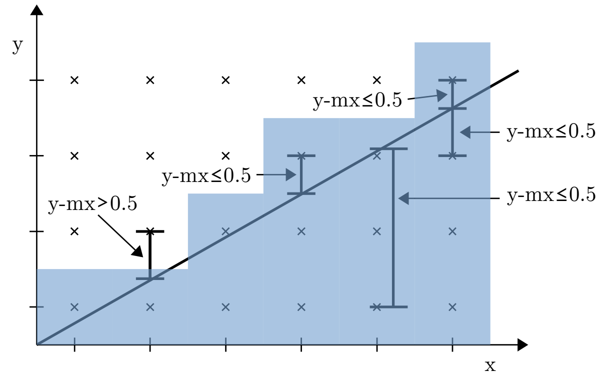


Figure 3.19: Description of the render range and render coordinate system of the triangle primitives, where $m = \frac{dy}{dx}$

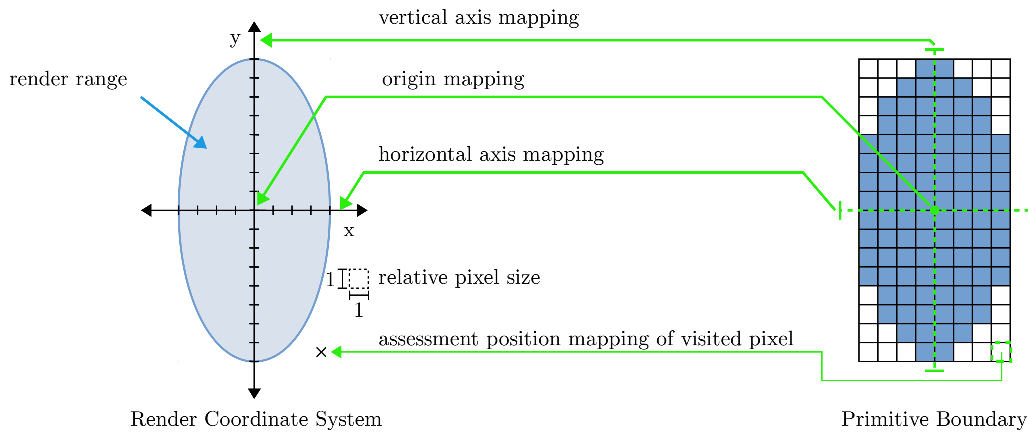


Figure 3.20: Description of the coordinate transformation procedure, render coordinate system and render range of the ellipse primitive

Render Range of Ellipse Primitive

Figure 3.20 shows the render range of the ellipse primitive. The render range, expressed by Equation 3.12, is the area within an ellipse centred around the origin. The horizontal, a, and vertical, b, radii of the ellipse are equal to half the lengths, in pixel units, of the horizontal and vertical edges of the primitive boundary, respectively.

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} \leq 1 \tag{3.12}$$

Where:
a = horizontal radius of ellipse
b = vertical radius of ellipse
x = horizontal component of assessment position
y = vertical component of assessment position

3.5.5 Rectangle Primitive Rendering Procedure

Since all pixels within the rectangle primitive's boundary should be provided with the colour specified by the primitive's *clr* attribute, no operations need to be performed to determine the colour of those pixels. Thus, when the assessment position visits any pixel within a rectangle primitive's boundary, that pixel is simply coloured according to the primitive's *clr* attribute without performing any of the procedures that are performed for the line, triangle and ellipse primitives.

3.5.6 Render Range Equation Optimisation

The render range equations of the line, triangle and ellipse primitives were modified to simplify the implementation of those equations in digital logic. All fractions were removed by multiplying all terms in the equations by the product of all fraction denominators. This allows the equations to be implemented without any division logic circuits, which reduces the overall digital logic required to implement MicroGE's rendering algorithms. Table 3.2 lists the render range equations of the line, triangle and ellipse primitives along with their modified forms.

Primitive	Derived Equations	Modified Equations
Line	$-0.5 < y - \frac{dy}{dx}x \leq 0.5$	$-dx < 2ydx - 2xdy \ and \ 2ydx - 2xdy \leq dx$
Triangle	$y - \frac{dy}{dx}x \leq 0.5$	$2ydx - 2xdy \leq dx$
Ellipse	$\frac{x^2}{a^2} + \frac{y^2}{b^2} \leq 1$	$x^2b^2 + y^2a^2 \leq a^2b^2$

Table 3.2: The derived render range equations and their modified forms

3.6 Raster Rendering

3.6.1 Raster Memory

MicroGE's raster memory stores 2D arrays of pixel data, which may represent icons, text characters or parts of images, that are used to render the two raster primitives. This pixel data is not generated by MicroGE itself, but transferred from MicroGE's controller. As shown in Figure 3.21, the transferred pixel data is grouped into n × n arrays, called "raster blocks", whose dimensions, n, are equal to the value of the PixelsPerGrid property. Since the pixel array dimensions of MicroGE's grid blocks are also determined by the value of PixelsPerGrid, the array dimensions of the raster blocks and grid blocks are both n × n. Thus, raster primitives can be rendered by providing the colours of pixels within raster blocks to pixels within the grid blocks that are within raster primitive boundaries.

Furthermore, the raster blocks are located within the raster memory by means of a base address. When assigning a base address of a raster block to a grid block, the colours of the pixels within that raster block, "raster block pixels", will be provided to the pixels within that grid block, "grid block pixels". Both the image and string primitives' rendering procedures involve assigning base addresses to the grid blocks within their primitive boundaries so that the colours of raster block pixels can be provided grid block pixels. The only difference in their rendering procedures is in how the values of the base addresses to assign to their grid blocks are determined.

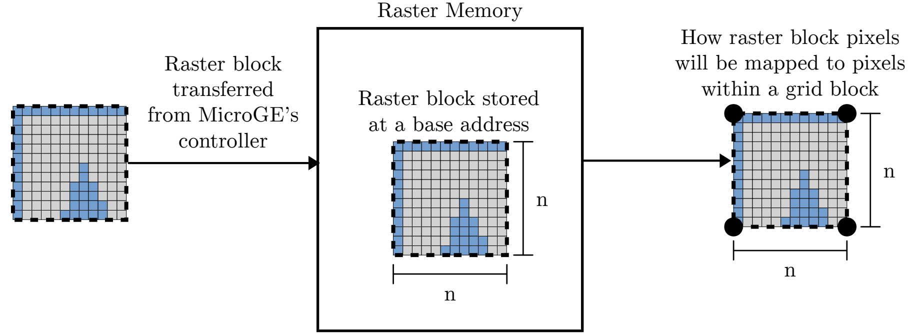


Figure 3.21: Description of MicroGE's raster memory

3.6.2 Pixel Addressing

Each raster block pixel can be located by means of a pixel address. These pixel addresses are used to map raster block pixels to grid block pixels. Figure 3.22 shows how the pixel addresses of pixels within a raster block, that has a base address of m, are determined. The pixel address of the top left raster block pixel is calculated by the value of the base address multiplied by the square of PixelsPerGrid while the remaining pixel addresses are determined by incrementing that calculated address. The incremented addresses are provided to the pixels in the top pixel row first, from left to right, then to the pixels of the proceeding rows, from left to right. When a base address of a raster block is assigned to a grid block, the pixel addresses of the corresponding raster block pixels, whose colour values need to be provided to the grid block pixels, can be determined by Equation 3.13.

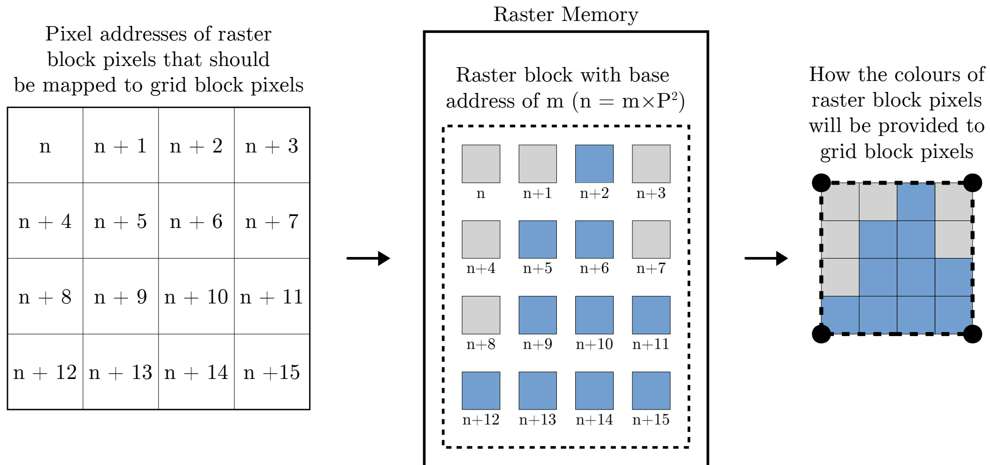


Figure 3.22: Description of pixel address calculation

$$adr_{px} = P^2 \times adr_b + P \times dy_{px} + dx_{px} \tag{3.13}$$

Where:
adr_{px} = pixel address
adr_b = base address of raster block
dx_{px} = horizontal pixel offset from top left corner of grid block
dy_{px} = vertical pixel offset from top left corner of grid block
P = value of PixelsPerGrid property

3.6.3 Image Primitive Rendering Procedure

The rendering procedure of the image primitive involves assigning base addresses to the grid blocks within primitive boundaries so that a corresponding set of raster blocks can be retrieved and rendered within those grid blocks. As shown in Figure 3.23, raster blocks constituting a single image should be stored at adjacent locations in the raster memory. If the first raster block has a base address of n, the proceeding raster blocks should have base addresses of n + 1, n + 2 and so forth. To render raster blocks of an image within the grid blocks of an image primitive's boundary, the base addresses of the raster blocks need to be calculated and then assigned to the grid blocks. The base address that should be assigned to the top left grid block within the image primitive's boundary is equal to the *adr* attribute of that image primitive. The values of the remaining base addresses are determined by incrementing the value of *adr*. The incremented values should be assigned to each grid block within the grid rows, from left to right, starting with the first grid row and progressing towards the bottom one. Equation 3.14 is used to calculate the base addresses that should be assigned to the grid blocks within an image primitive.

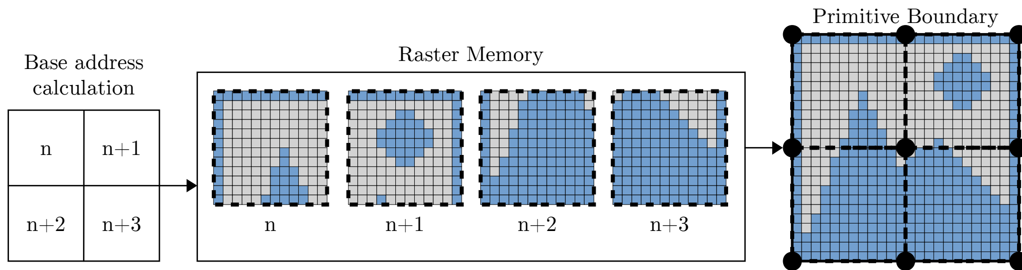


Figure 3.23: Description of the rendering procedure of the image primitive

$$adr_b = adr + x_l + y_l \times dx_g \tag{3.14}$$

Where:
adr_b = base address of raster block
adr = address attribute of image primitive
dx_g = width of primitive boundary
x_l = horizontal component of local grid position
y_l = vertical component of local grid position

3.6.4 String Memory

The string memory allows MicroGE to render a string of text characters, such as American Standard Code for Information Interchange (ASCII) characters, within the boundary of the string primitive. Each of these characters is depicted by a 2D array of pixels which are stored in the raster memory as raster blocks. These raster blocks, referred to as “character raster blocks”, do not need to be stored at adjacent locations in the raster memory. Character raster blocks are located in the raster memory by means of base addresses, referred to as “character pointers”, that are stored in MicroGE’s string memory. Figure 3.24 shows how a series of character pointers stored in the string memory is used to retrieve specific character raster blocks from the raster memory. The character raster blocks were transferred to the raster memory from MicroGE’s controller, and thus, the controller knows what the character pointers, or base addresses, of each character raster block is. This allows the controller to transfer a set of character pointers to the string memory which can be used to specify which character raster blocks to render within a string primitive. Furthermore, when the controller transfers character raster blocks, depicting ASCII characters, to the raster memory and also maps the corresponding character pointers to ASCII character codes, it allows the controller to render strings of ASCII characters within the boundary of the string primitive.

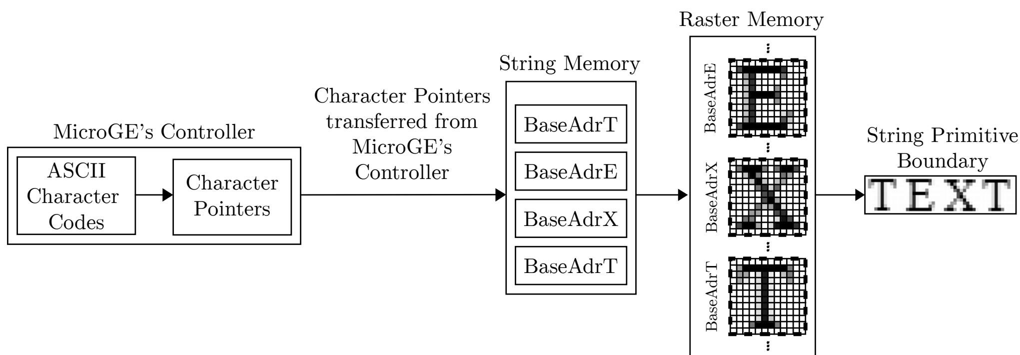


Figure 3.24: Description of how the string memory should be used

3.6.5 String Primitive Rendering Procedure

A string primitive is rendered by mapping a set of character raster blocks to the grid blocks within the string primitive’s boundary. Contrary to the other primitives, the height of the string primitive’s boundary is restricted to one grid unit since it is intended to render a single line of text. Each one of the grid blocks within this boundary is assigned with a character pointer so that a specific character raster block can be rendered within that grid block. The character pointer to assign to each grid block is located from the string memory by means of an address. As shown in Figure 3.25, the leftmost grid block is assigned with a character pointer located at an address determined by the value of the *adr* attribute of the string primitive, n. The proceeding grid blocks are assigned with character pointers that are located at addresses following the value of the *adr* attribute, n + 1, n + 2 and so forth.

Equation 3.15 is used to calculate the address of a character pointer that should be assigned to a specific grid block within the string primitive's boundary.

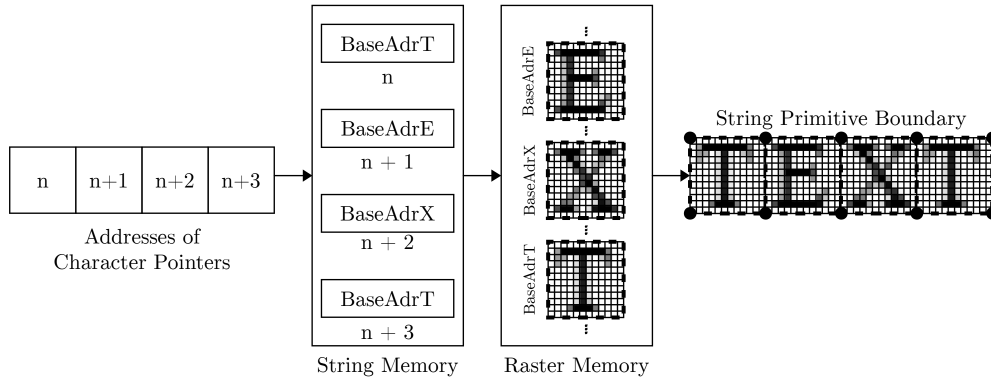


Figure 3.25: Description of the rendering procedure of the string primitive

$$adr_{ch} = adr + x_l \tag{3.15}$$

Where:
adr_{ch} = address of character pointer to assign to grid block
adr = address attribute of string primitive
x_l = horizontal component of local grid position

3.7 Primitive Processing

3.7.1 Primitive Memory and Pointer Memories

MicroGE uses a primitive memory and two pointer memories to process the primitives that should be rendered on the canvas. The primitive memory stores all of the primitives that should be rendered. Before the pixels of a canvas row are produced, the primitives whose boundaries intersect that canvas row are selected from the entire set of primitives stored in the primitive memory. They are selected by storing their primitive memory addresses as pointers in one of the two pointer memories. Then, when the pixels of the canvas row should be produced, the pointers are used to locate the primitives that intersect the canvas row. Then, as the render position travels along the canvas row, the primitive boundaries are positioned on the canvas row so that once the render position falls within one of those boundaries, the pixels within can be provided with the appropriate colour.

3.7.2 Primitive Memory Organisation

For MicroGE to perform all of the processing required to position the primitive boundaries on a canvas row in the required time, MicroGE manages the order of the primitives in the primitive memory. All primitives are stored in the primitive memory as their boundaries are positioned on the canvas from left to right. This order is independent of their depth layers and the vertical positions of their boundaries. Figure 3.26 shows the locations of the boundaries of 1000 primitives that are

stored in the primitive memory. Each of these primitives is accessed in the primitive memory using a corresponding address. As their addresses increase from zero to 999, the initial horizontal positions, x_i, of their boundaries increase as well. No primitive stored in a specific location in the primitive memory is allowed to have an initial horizontal position smaller than the one of the primitive stored in the location preceding it.

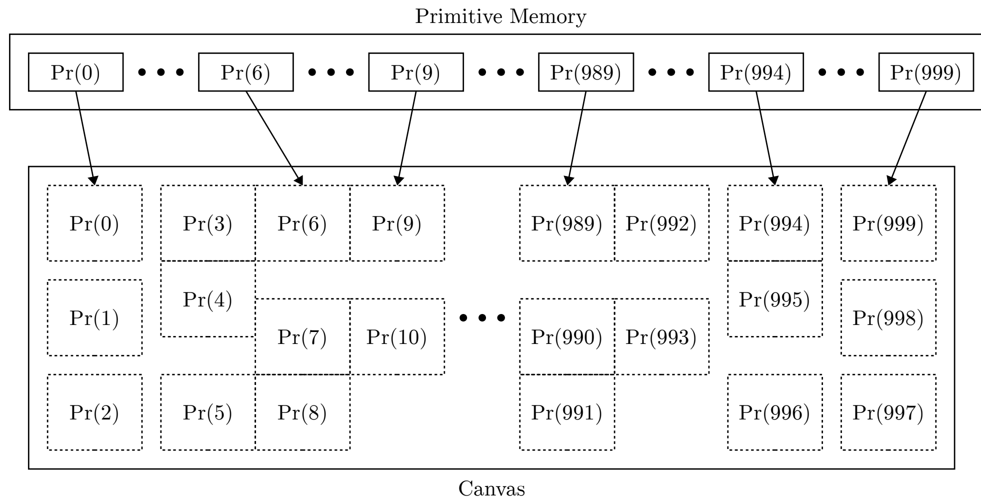


Figure 3.26: Explanation of how the primitives are stored in the primitive memory

3.7.3 Primitive Selection

With the primitives stored in the primitive memory as described in Section 3.7.2, they can be selected in the order in which they should be rendered on the canvas row by following the process described in Figure 3.27. All of the primitives in the primitive memory are iterated through, starting from the one located at address zero up to the one located at the highest address, in this case, 999. Each primitive's attributes are inspected to determine if that primitive's boundary is positioned on the specific canvas row to produce. If a primitive's boundary is positioned on the canvas row, that primitive's address is stored as a pointer in one of the pointer memories. The address, m, of the first primitive whose boundary was found to be positioned on the canvas row is stored in the first location of the pointer memory. Each time another primitive's boundary is found to be positioned on the canvas row, that primitive's address is stored in the proceeding location of the pointer memory. As a result, when the pointers are now provided to the primitive memory, starting from the first pointer and progressing towards the last one, all primitives whose boundaries are positioned on the canvas row will be provided from the primitive memory as output. Furthermore, those primitives will be provided as output in the order in which their boundaries are positioned on the canvas row from left to right.

3.7.4 Primitive Sequencing

Figure 3.28 shows how pixels of a canvas row are provided with colours by sequencing the primitive boundaries on the canvas row as the render position travels along

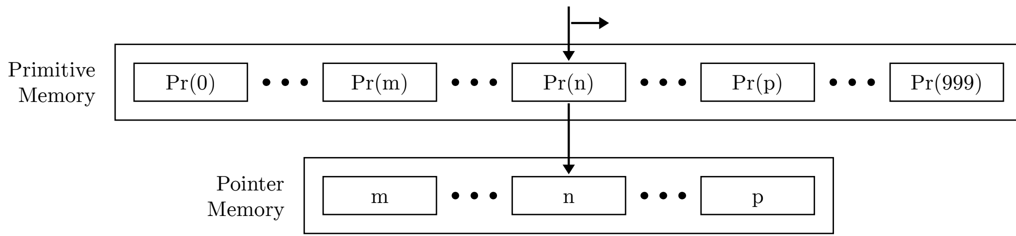


Figure 3.27: Explanation of the primitive selection process

it. As the stored pointers are provided to the primitive memory as input, the primitives are provided as output. The primitives are then provided to MicroGE's rendering processes so that their boundaries can be derived and then positioned on the canvas row. Before the render position visits the first pixel of the canvas row, the first pointer, q, is provided to the primitive memory to select the primitive whose boundary should be positioned on the leftmost position of the canvas row. Once the render position visits the pixels within this boundary, the pixels are coloured according to that primitive's attributes. When the last pixel within the boundary is coloured, the primitive whose boundary needs to be positioned on the proceeding position of the canvas row is located from the primitive memory. It is located by simply providing the next pointer, k, to the primitive memory. Every time the render position is done visiting the last pixel within a primitive boundary, the primitive required to derive the proceeding primitive boundary is retrieved using the following pointer. This process repeats until all primitive boundaries have been positioned where required and all of the pixels of the canvas row are provided with the required colours. Lastly, since MicroGE has eight depth layers, and primitives may overlap, this primitive sequencing process is performed for the primitives of each depth layer separately using pipelined techniques. These pipelined techniques are detailed in Section 4.2.5.

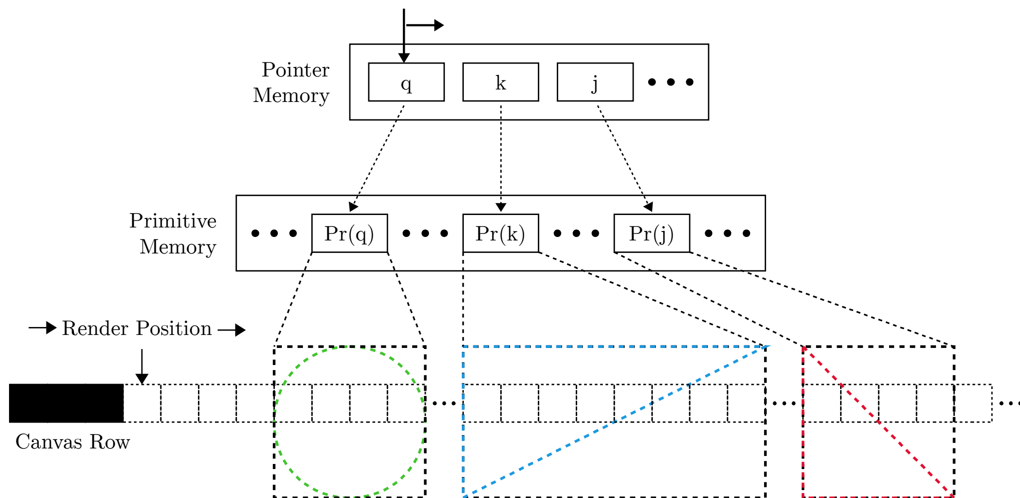


Figure 3.28: Explanation of the primitive sequencing process

3.7.5 Primitive Buffer

The primitive memory is almost constantly being accessed by MicroGE's rendering processes, throughout the HDMI video frame period, to render the canvas. Thus, not a lot of time remains to insert new primitives into or delete primitives from the primitive memory. Thus, the primitive buffer, which is a replica of the primitive memory, is provided to store the primitives that should be rendered on the canvas, before they are transferred to the primitive memory. The primitive buffer can be accessed during any time throughout the HDMI video frame period so that all of the required updates to the primitives, that should be rendered on the canvas, can be made. Then, after all of the updates are made, the updated set of primitives can quickly be copied to the primitive memory during one of the brief time periods of the HDMI video frame period when the primitive memory is not accessed by MicroGE's rendering processes.

Furthermore, since the primitives need to be stored in the primitive memory in a specific order, the primitives in the primitive buffer need to be stored in this order as well. When MicroGE's controller inserts or removes a primitive into or from the primitive buffer, respectively, this order needs to be maintained. MicroGE provides two procedures for inserting and deleting primitives into and from the primitive buffer, respectively, while still maintaining this order. Since these procedures will keep the primitives, stored in the primitive buffer, in the correct order, those primitives will be in the correct order after they are copied to the primitive memory. These two procedures are discussed in the following two sections.

3.7.6 Insert Procedure

Figure 3.29 describes how the insert procedure identifies the location where a primitive should be inserted, reorganises the primitive buffer, and then inserts a new primitive. For the first part of this procedure, the primitives are iterated through, and their initial horizontal positions are inspected. If an inspected primitive has a larger initial horizontal position than that of the primitive to be inserted, the address of that inspected primitive is stored, and the first part of the procedure is complete. Then, each one of the primitives, starting from the one whose address was stored up to the last one, is rewritten to the proceeding memory location. This is equivalent to shifting those primitives upwards by one memory location. Then, the primitive to insert is written to the location at the stored address. Thus, the primitive is inserted at the required memory location, and the primitive buffer order is maintained.

3.7.7 Delete Procedure

Figure 3.30 describes how the delete procedure identifies a primitive in the primitive buffer and then reorganises the primitive buffer to remove that primitive. First, the primitive buffer is iterated through to find the location of the primitive to delete. The one to delete is specified by the *id* attribute. Once the primitive with that *id* attribute is located, each one of the primitives, starting from the located one up to the last, is rewritten to the preceding memory location. This is equivalent to shifting all of those primitives downwards by one memory location, and thus, overwriting

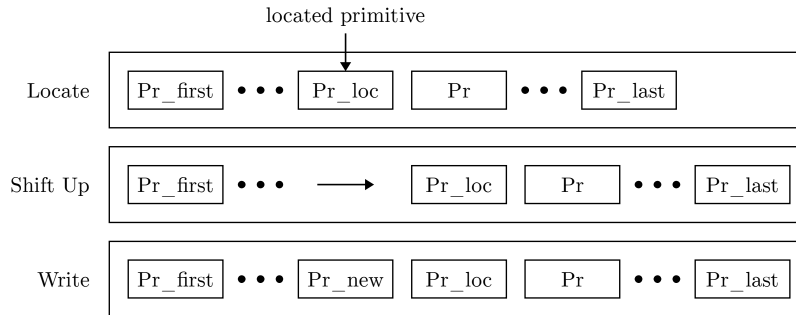


Figure 3.29: Explanation of the insert procedure for the primitive buffer

the primitive that should be deleted. Thus, the required primitive is removed, and the primitive buffer order is maintained.

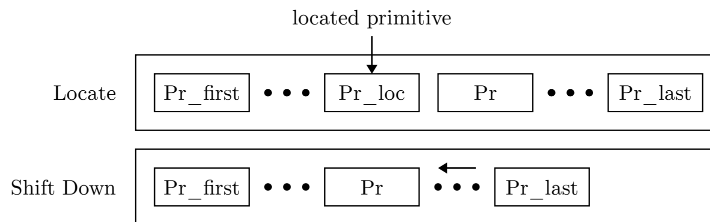


Figure 3.30: Explanation of the delete procedure for the primitive buffer

3.8 Pipeline Mechanisms

3.8.1 Video Transmitter Synchronisation

MicroGE renders the pixels of canvas rows and provides those pixels to a display monitor by synchronising with the video transmission process of the video transmitter component. To perform this synchronisation, MicroGE aligns its “row periods” with the line periods of the HDMI video frame period. A “row period” is defined as a period in which MicroGE may perform an operation which contributes to the generation and transmission of the canvas. Each of these row periods occurs during a specific line period of the HDMI video frame period.

3.8.2 Component Operations

MicroGE performs “component operations” throughout the HDMI video frame period to create the canvas rows and to provide the pixels of those canvas rows to a display monitor. The “component operations” are defined as functions that MicroGE’s components perform to transfer render information to memory components, render canvas rows from render information, and provide pixels of canvas rows to the video transmitter for transmission to a display monitor. Some of the component operations are performed within specific row periods and are referred to as the “synchronous component operations”. Other component operations can be performed during any time throughout the HDMI video frame period and are referred

to as the “asynchronous component operations”.

The synchronous component operations are defined as follows:

- CopyPrim copies the primitives stored in the primitive buffer to the primitive memory so that those primitives can be rendered on the canvas
- PrimSelect performs the procedure described in Section 3.7.3 to select the primitives that should be rendered on a specific canvas row
- PrimSequence performs the procedure described in Section 3.7.4 to sequence the selected primitives as they should be rendered on a canvas row
- VectorRender renders vector primitives, which are provided from the PrimSequence component operation, based on the equations and procedures discussed in Section 3.5
- RasterRender renders raster primitives, which are provided from the PrimSequence component operation, based on the equations and procedures discussed in Section 3.6
- WriteLine selects the pixels that are rendered by the VectorRender and RasterRender component operations and stores those pixels in one of the line buffers
- ReadLine reads the pixels that are stored in one of the line buffers and provides those pixels to the video transmitter for transmission to a display monitor

The PrimSequence, VectorRender, RasterRender and WriteLine component operations are performed together as a set within a row period since information is transferred between them. Together, they produce the pixels of a canvas row from selected primitives and store those pixels in one of the line buffers. This set of component operations are referred to as the “RenderSet” component operations.

Furthermore, the RenderSet component operations include eight VectorRender component operations but only one RasterRender component operation. Eight VectorRender component operations are required since eight vector primitives may be rendered at the same time, one for each depth layer. On the other hand, only one RasterRender component operation is required. If multiple RasterRender component operations were used to render raster primitives on different depth layers, only the raster primitive on the highest depth layer would be visible since MicroGE does not support colour transparency. To reduce the resource usage of MicroGE, only one RasterRender component operation is used, which only renders the raster primitive on the highest depth layer.

Furthermore, the asynchronous component operations are defined as follows:

- InsertPrim inserts a new primitive into the primitive buffer according to the procedure described in Section 3.7.6
- DeletePrim removes a primitive from the primitive buffer according to the procedure described in Section 3.7.7
- ClearPrim clears the primitive buffer

- StorePixels stores a set of pixels in the raster memory
- StoreChars stores a set of character pointers in the string memory

3.8.3 Synchronous Component Operation Scheduling

Figure 3.31 shows which synchronous component operations are performed during each row period. During row period two, the CopyPrim component operation is performed to update the primitive memory with the latest set of primitives from the primitive buffer, but only if this operation is requested by MicroGE's controller. During row period three, PrimSelect is performed to select the primitives that should be rendered on canvas row one. During row period four, RenderSet is performed to produce canvas row one from these selected primitives. Additionally, PrimSelect is also performed during row period four to select the primitives that should be rendered on canvas row two. During row period five, PrimSelect and RenderSet are performed to select the primitives that should be rendered on canvas row three and to produce canvas row two from the primitives that were selected during row period four, respectively. Then, during the following row periods, no component operations are performed unless those row periods are canvas row periods. A "canvas row period" is defined as a row period that occurs during an active line period of the HDMI video frame period in which a corresponding canvas row is provided to the video transmitter using the ReadLine component operation. Additionally, the PrimSelect and RenderSet component operations are also performed during most of the canvas row periods. During the first canvas row period, canvas row one is provided to the video transmitter, using ReadLine. Then, during canvas row periods two up to $90 \times P - 2$, ReadLine, RenderSet and PrimSelect are performed, where P is the value of the PixelsPerGrid property. During a specific one, n, of these canvas row periods, canvas row n is provided to the video transmitter, using ReadLine, canvas row $n + 1$ is produced, using RenderSet, and the primitives are selected for canvas row $n + 2$, using PrimSelect. Then, during canvas row period $90 \times P - 1$, PrimSelect is not performed any more since PrimSelect was performed for the last canvas row during canvas row period $90 \times P - 2$. Also, during canvas row period $90 \times P$, only ReadLine is performed since there are no more canvas rows left to produce.

3.8.4 Parallel Execution of Component Operations

Figure 3.32 shows how the PrimSelect, RenderSet and ReadLine component operations are performed in parallel during canvas row period n. PrimSelect selects the primitives from the primitive memory that should be rendered on canvas row $n + 2$ by storing the addresses of those primitives as pointers in pointer memory one. PrimSequence uses the pointers, which were selected by PrimSelect during canvas row period $n - 1$, from pointer memory two and then retrieves the primitives from the primitive memory that should be rendered on canvas row $n + 1$. PrimSequence provides the primitives it retrieves to the eight VectorRender component operations and the one RasterRender component operation so that those operations can render pixels of those primitives to the eight depth layers. WriteLine selects the pixels that are on the highest of these depth layers and writes the selected pixels to line buffer one to produce canvas row $n + 1$. ReadLine reads canvas row n from line buffer two

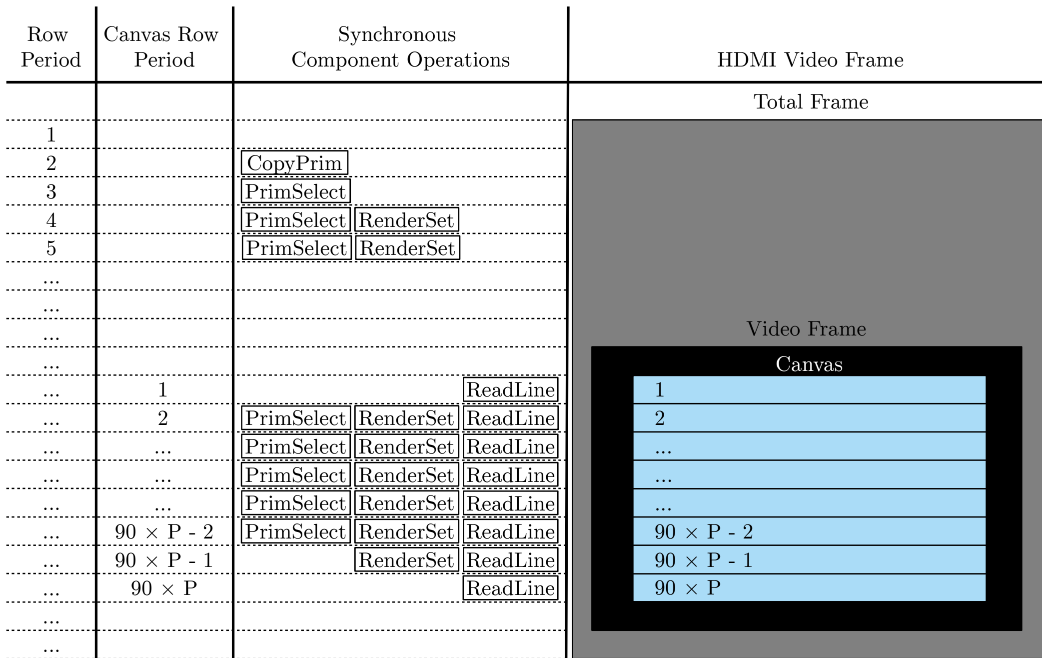


Figure 3.31: Description of when the synchronous component operations are performed

and provides the pixels of that canvas row to the video transmitter for transmission to a display monitor. Canvas row n was rendered during canvas row period n - 1, by the RenderSet component operations, while the primitives whose pixels were rendered on that canvas row were selected during canvas row period n - 2, by the PrimSelect component operation.

3.8.5 Canvas Segment

Figure 3.33 shows how MicroGE produces a single segment of the canvas row at a time, which is referred to as a "canvas segment". Each canvas row consists of 160 canvas segments, and the number of pixels that a canvas segment consists of is equal to the PixelsPerGrid property. The canvas segments are produced from left to right as time progresses. Additionally, each pixel within the canvas segments is also produced from left to right as time progresses. In Section 3.4.2, this was explained as the render position that travels along the canvas row, from left to right, and colouring each blank pixel along the way. The RenderSet component operations produce a specific canvas segment of a specific canvas row based on the values of the global grid positions, (x_g, y_g), and the pixel offset values, (dx_g, dy_g). The RenderSet component operations generate the horizontal component of the global grid positions, x_g, and the horizontal pixel offset values, dx_g, themselves, to specify which canvas segment and pixel within the canvas segment to produce, respectively. On the other hand, another component within MicroGE, the Timing Controller, which will be discussed in Section 4.2.3, provides the RenderSet and PrimSelect component operations with the vertical component of the global grid positions, y_g, and the vertical pixel offset values, dy_g, to specify which of the 90 × P canvas rows need to be produced.

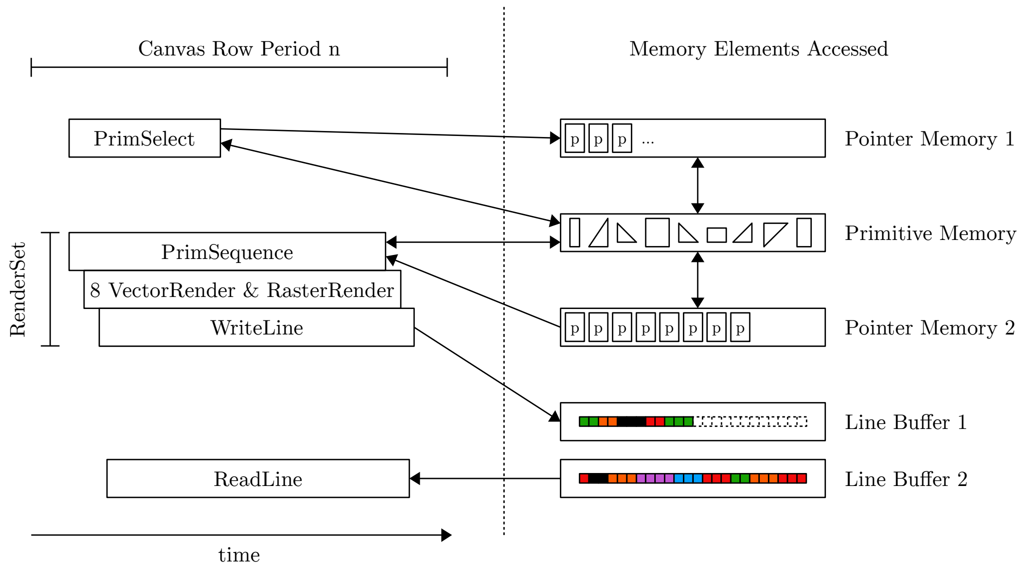


Figure 3.32: Conceptual timing diagram of the synchronous component operations that are performed during a canvas row period

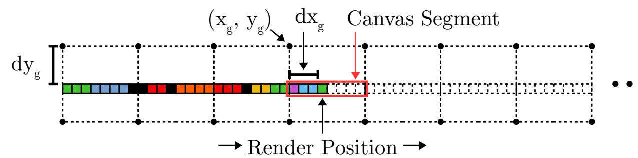


Figure 3.33: Conceptual explanation of a canvas segment

3.8.6 Render Stages

The RenderSet component operations consist of multiple stages which each perform parts of the computations required to produce the canvas segments. These stages are referred to as "render stages" which each consists of 160 "grid periods". A "grid period" is a twelve clock cycle period in which a computation is performed which contributes to the production of a specific canvas segment. Figure 3.34 shows the render stages of the RenderSet component operations and when a specific grid period, n, of each render stage takes place. The results of the computations of grid period n in one stage are provided to grid period n of the subsequent stages so that canvas segment n can eventually be produced. Furthermore, all of the n'th grid periods among all of the render stages are referred to as the n'th "grid period set".

Grid period n of the render stages are out of phase so that the computations of grid period n of one render stage can be completed before it is required by grid period n of one of the proceeding render stages. Grid period n of the render stages of the PrimSequence component operation is performed first, to retrieve the primitives from the primitive memory that should be rendered on canvas segment n. This set of retrieved primitives is referred to as the n'th "primitive set". A maximum of eight primitives, one for each depth layer, can be retrieved. These primitives may then be provided to grid period n of the first render stage of the one RasterRender component operation and to grid period n of the first render stage of the eight VectorRender

component operations. The RasterRender and VectorRender component operations then perform computations during grid period n of their several render stages to provide a set of “render segments” as output. “Render segments” are defined as a set of pixels, with a cardinality equal to the PixelsPerGrid property, which is produced for a specific depth layer, by either the RasterRender or VectorRender component operations, during a specific grid period. Then, during grid period n of the first and second render stages of the WriteLine component operation, the pixels within all of the n’th render segments are evaluated to choose the pixels on the highest depth layers so that those pixels can be stored in one of the line buffers as part of the n’th canvas segment.

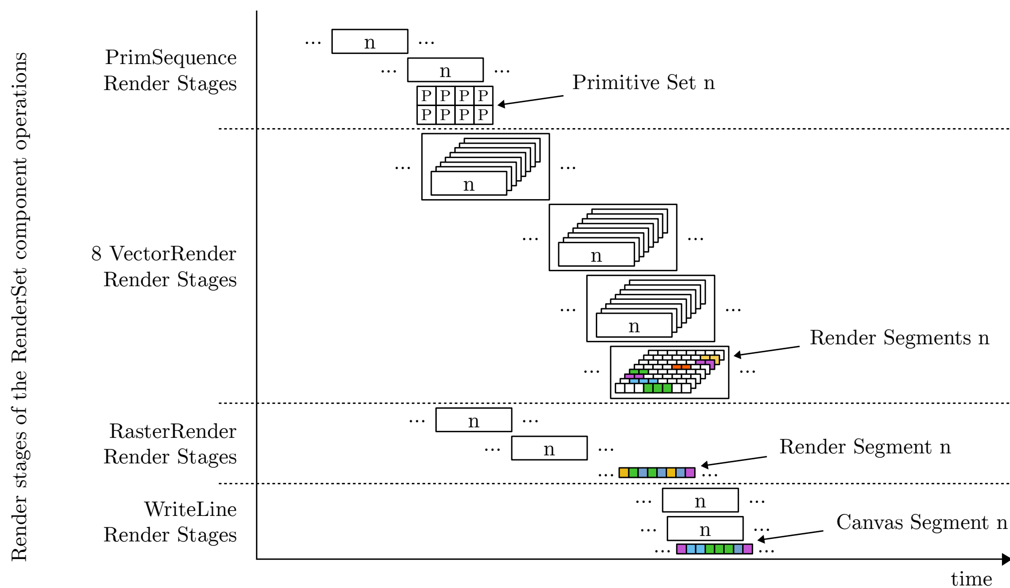


Figure 3.34: Conceptual timing diagram detailing the render stages

Chapter 4

Firmware Implementation

4.1 Implementation Overview

Figure 4.1 shows the firmware components that were created for this study, as well as the devices that they were implemented on. MicroGE, along with three other components, was implemented as a single VHDL design on a Spartan-6 LX25 FPGA, using ISE Design Suite 14.7. The three other components that were implemented include a video transmitter, clock generator and EDID reader. MGAPI, a C software library that allows a device to control MicroGE, was implemented on an Arduino Due, using the Arduino integrated development environment (IDE). This software library allows a device, which is then referred to as MicroGE's "controller", to send control, configuration and render information to MicroGE.

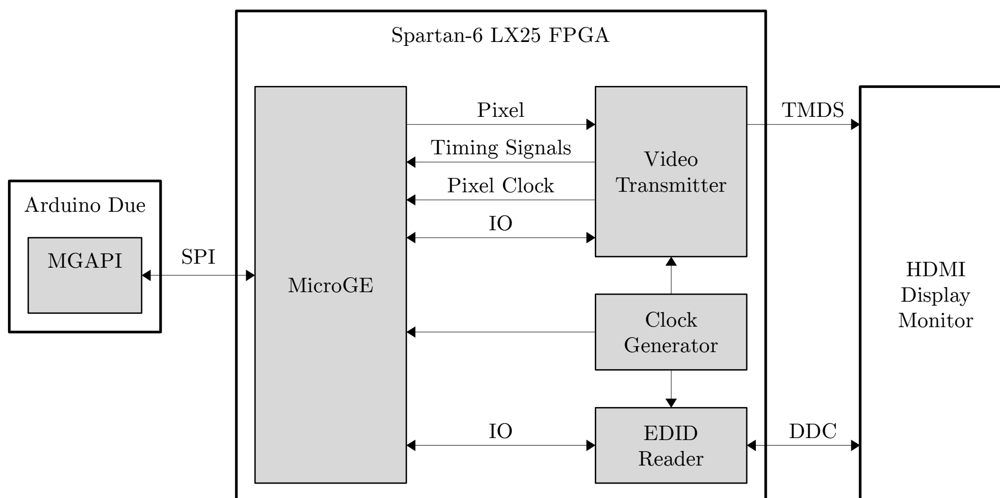


Figure 4.1: Overview of firmware implementation

MicroGE was implemented with portability in mind, so that it may be used on other FPGA platforms in the future without making significant changes to its VHDL. One way in which MicroGE achieves portability is by excluding clock generation and video transmission components from itself. These components are usually implemented using device-specific resources of an FPGA. When porting these components to other FPGA platforms, significant changes would have to be made to them. Most FPGAs are already provided with example designs containing clock generation and video transmission components which use those FPGAs' platform

specific hardware resources. Thus, it is not required to recreate and include these components within MicroGE. Instead, MicroGE was designed to receive a clock from an external clock generator component, was designed to interface with an external video transmitter component and was provided with a set of IO ports for additional control over external components. This set of IO ports consists of separate input and output ports which can receive signal values from and provide signal values to external components, respectively. Additionally, MicroGE interfaces with the external video transmitter by synchronising with the video transmitter's video timing signals and pixel clock. Once synchronised, MicroGE provides the pixels of the canvas to the video transmitter so that the canvas can be transmitted along with the video frame.

Apart from the advantage of portability, keeping the video transmitter separate from MicroGE also has other advantages. MicroGE does not have to be configured with video mode parameters since it derives them automatically by reading the video timing signals of the video transmitter. This also allows MicroGE to support a variety of different video modes without storing any video mode configuration information. Also, both HDMI and DVI video transmission protocols can be supported since MicroGE can interface with both HDMI and DVI video transmitters, respectively.

Since MicroGE requires external components, they were implemented as part of the VHDL design as well. The first of these, the video transmitter, supports seven different video modes, which are supported by HDMI and DVI video monitors. The video transmitter is able to switch between these video modes dynamically based on input received from MicroGE's IO ports. Also, the video transmitter uses a 50 MHz external reference clock to generate all of the clocks and signals required to transmit video data via a TMDS interface. The second external component, the EDID reader, reads the EDID/E-EDID ROM of a video monitor, via a DDC interface, to determine the monitor's display capabilities. The EDID reader is controlled via MicroGE's IO ports. The last external component is a clock generator component which generates a 100 MHz clock for MicroGE, a 50 MHz clock for the EDID reader and a 50 MHz clock for the video mode reconfiguration logic within the video transmitter.

Lastly, a Serial Peripheral Interface (SPI) interface is used to transfer control, configuration and render information from MicroGE's controller, the Arduino Due running MGAPI, to MicroGE itself. This interface was chosen since it is simple, and since it is also supported by most MCUs. Additionally, if in the future, MicroGE's controller is implemented as a soft-core processor, on the same FPGA as MicroGE, it will be simple to implement an SPI interface on that FPGA.

4.2 MicroGE VHDL Implementation

4.2.1 Architecture

MicroGE was implemented as a VHDL component according to the architecture in Figure 4.2. All of the components in this architecture were also implemented as VHDL components which are used within MicroGE's VHDL component.

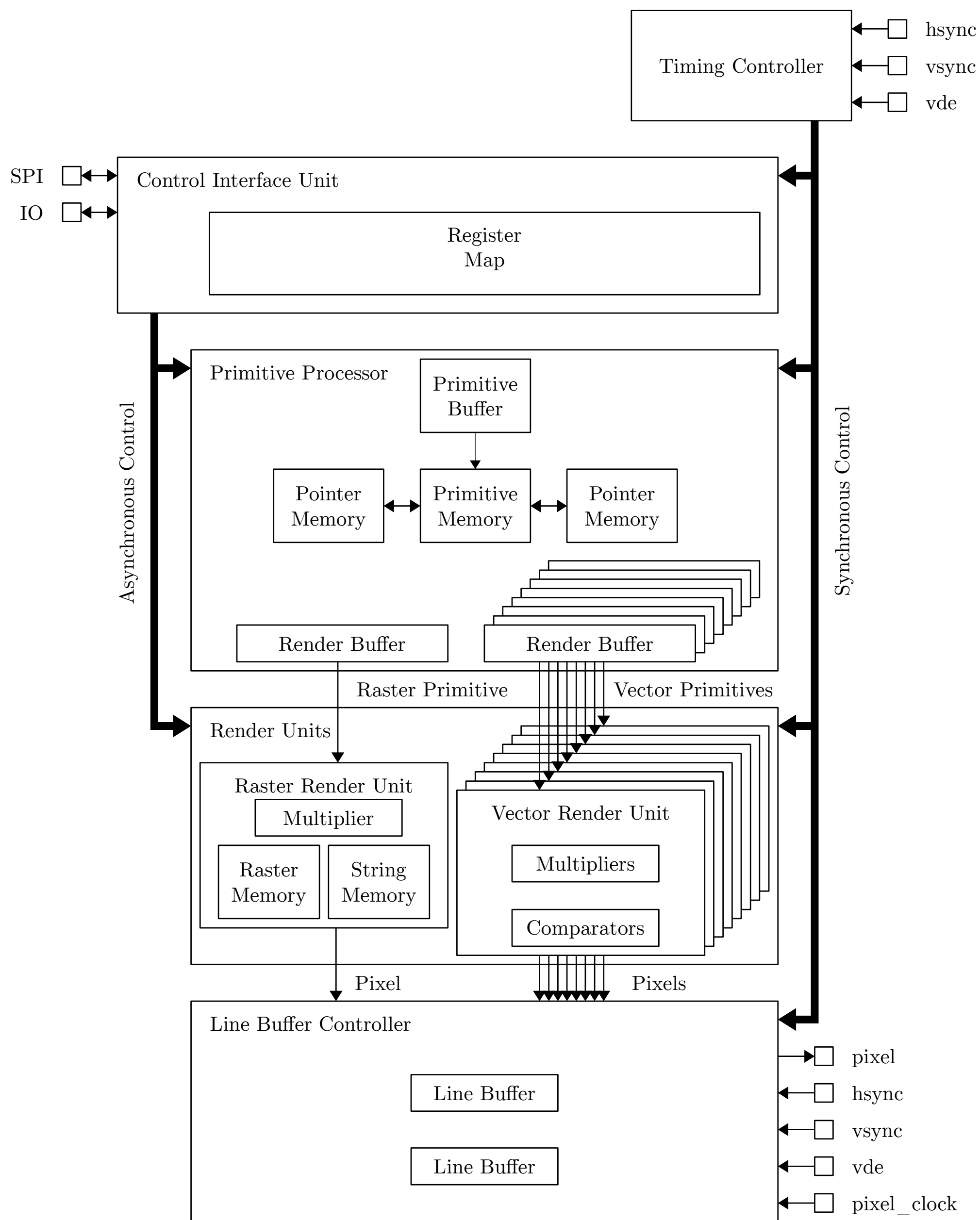


Figure 4.2: Architecture of MicroGE

Timing Controller

The timing controller synchronises MicroGE's row periods with the line periods of the HDMI video frame period, as discussed in Section 3.8.1, and then sequences the synchronous component operations according to Section 3.8.3. It reads the video timing signals, hsync, vsync and vde, which are provided from the video transmitter, to perform this synchronisation. Then, the timing controller provides control and data signals to other components of MicroGE so that those components can perform the required synchronous component operations during the row periods.

Control Interface Unit

The control interface unit provides an interface between MicroGE's controller and MicroGE itself. It interprets SPI instructions, sent from MicroGE's controller, and then either transfers information to the "register map" or sequences the asynchronous component operations. The register map is a set of registers which is used to transfer data to and read data from MicroGE's components. Since these registers can be accessed by MicroGE's controller, via the SPI interface, the register map provides MicroGE's controller with a means of accessing MicroGE's components so that control, configuration and render information can be transferred to them. Also, some of these registers are connected to MicroGE's IO ports so that those ports can be controlled from MicroGE's controller.

Primitive Processor

The primitive processor performs the procedures discussed in Section 3.7 to transfer primitives to the primitive buffer, to update the primitive memory with the contents of the primitive buffer, and to render the canvas with the assistance of the primitive memory and the two pointer memories. It performs these procedures using the InsertPrim, DeletePrim and ClearPrim asynchronous component operations and the CopyPrim, PrimSelect and PrimSequence synchronous component operations. It also contains a set of render buffers each of which is connected to a render unit. These buffers are used to store primitives during specific grid periods so that the render units can produce the corresponding render segments from those primitives.

Render Units

The raster render unit and eight vector render units perform the procedures discussed in Section 3.6 and Section 3.5, respectively, to produce render segments from primitives that are provided from the render buffers. The raster render unit performs the RasterRender synchronous component operation to produce render segments of raster primitives. The pixels within these render segments are produced by retrieving pixels and character pointers, with the aid of a multiplier component, from the raster and string memories, respectively. The raster render unit also performs the StorePixels and StoreChars asynchronous component operations to transfer new pixels and character pointers to the raster and string memories, respectively. Each of the vector render units performs the VectorRender synchronous component operation to produce render segments of vector primitives. The pixels within the render segments are produced by evaluating, with the aid of multiplier and comparator components, the render range equations for these pixels. Finally, the raster render

unit and vector render units provide their render segments as output to the line buffer controller.

Line Buffer Controller

The line buffer controller performs the WriteLine and ReadLine synchronous component operations to control the storage of canvas rows in the line buffers and to control the video transmission of pixels from canvas rows, respectively. The WriteLine component operation evaluates the pixels of render segments, which are produced by the render units, to produce canvas segments that can be stored in one of the line buffers as part of a canvas row. The ReadLine component operation synchronises with the hsync, vsync and vde signals provided from the video transmitter so that the pixels of a specific canvas row can be transmitted to a display monitor, by the video transmitter, during the required times.

Portability of MicroGE

One way in which MicroGE was designed to achieve portability was by keeping certain VHDL components, which require FPGA platform-specific resources, external from itself. Unfortunately, not all components that require platform-specific resources can be kept external. These components include the multipliers used by the render units and the dual-port RAM (DPRAM) components, which will be discussed in Section 4.2.2, which are used by all of MicroGE's memory components. Instead of attempting to keep these two components external from MicroGE, they were implemented as "VHDL wrapper components" that are used within MicroGE. Currently, these VHDL wrapper components use platform-specific resources of the Spartan-6 LX25 FPGA, but the contents within these components can be modified to port MicroGE to other FPGA platforms. Thus, when porting MicroGE to other FPGA platforms, only the contents within these two VHDL wrapper components need to be modified.

Scalability of MicroGE

MicroGE was implemented with scalability in mind. The raster memory and string memory can be implemented in different sizes on an FPGA depending on the values provided to two VHDL generic statements. The first of these generic statements is RAST_MEM_DPRAM_NUM which determines the number of DPRAM components that will be included within the raster memory. The second of these generic statements is STR_MEM_DPRAM_NUM which determines the number of DPRAM components that will be included within the string memory. Increasing the values of these generic statements will provide more storage capacity for the raster and string memories, but will increase the FPGA resource usage of MicroGE.

4.2.2 VHDL Wrapper Components

Multiplier

A block diagram of the multiplier component is shown in Figure 4.3. It has three input ports, mult_a, mult_b and mult_c, and one output port, mult_y. The values provided to mult_a and mult_b are multiplied with each other and then added to

the value provided to mult_c. The result of this operation is then provided as output via mult_y. In this study, the multiplier was implemented using a device-specific hardware component of the Spartan-6 LX25 FPGA, the DSP48A1 [26]. The DSP48A1 contains several inputs of which three can be used to perform the multiplication and addition operations required by MicroGE's rendering operations. The DSP48A1 performs pipelined multiplication and addition on a set of values provided at its inputs and then provides the calculated result as output three clock cycles later.

When porting MicroGE to another FPGA platform, the hardware component used to provide the functionalities of the multiplier should meet the following requirements:

- The latency of its calculations should be three clock cycles
- It should be pipelined in order to provide a calculated result with each clock cycle
- It should allow mult_a and mult_b to have data widths of 18-bit or larger
- It should allow mult_c and mult_y to have data widths of 36-bit or larger

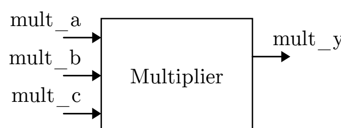


Figure 4.3: Block diagram of the multiplier component

DPRAM

A block diagram, which includes all main signal interfaces, of the DPRAM component is shown in Figure 4.4. Each of MicroGE's memory components consists of either one or a set of DPRAMs. The DPRAM has two sets of interfaces which can access the same memory contents simultaneously. Each interface set has an address port, mem_adr, which is used to select a location to which data should be written to or read from. When providing an address to mem_adr, the data located at the corresponding memory location is provided via mem_rd_data. When providing an address to mem_adr, along with data to mem_wr_data, while asserting mem_we, that data is written to the memory location specified by that address.

In this study, MicroGE's DPRAM component was implemented using a device-specific hardware component of the Spartan-6 LX25 FPGA, the RAMB16BWER component [27]. The RAMB16BWER component provides a memory capacity of 16 kb along with 2 kb of additional parity bit memory. The data port width of the RAMB16BWER component can be configured to 8-bit or 16-bit which then provides 2048 or 1024 memory locations, respectively.

When porting MicroGE to another FPGA platform, the hardware component used to provide the functionalities of the DPRAM component should meet the following requirements:

- It should provide a memory capacity of 16 kb or larger
- The widths of its data ports should be configurable to both 8-bit and 16-bit
- It should meet a read data latency of two clock cycles
- It should meet a write data latency of one clock cycle
- It should have two sets of interfaces to the same memory contents

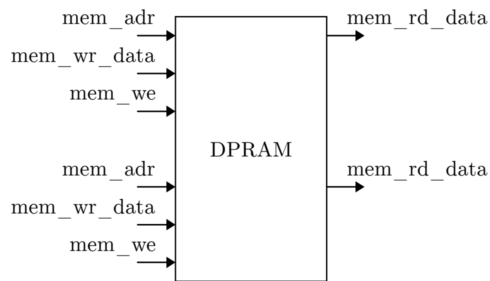


Figure 4.4: Block diagram of the DPRAM component

4.2.3 Timing Controller

A block diagram, which includes all main signal interfaces, of the timing controller is shown in Figure 4.5. The timing controller contains the following two processes:

- The VideoSync process synchronises the row periods with the line periods of the HDMI video frame period
- The CompControl process controls the synchronous component operations by means of control and data signals

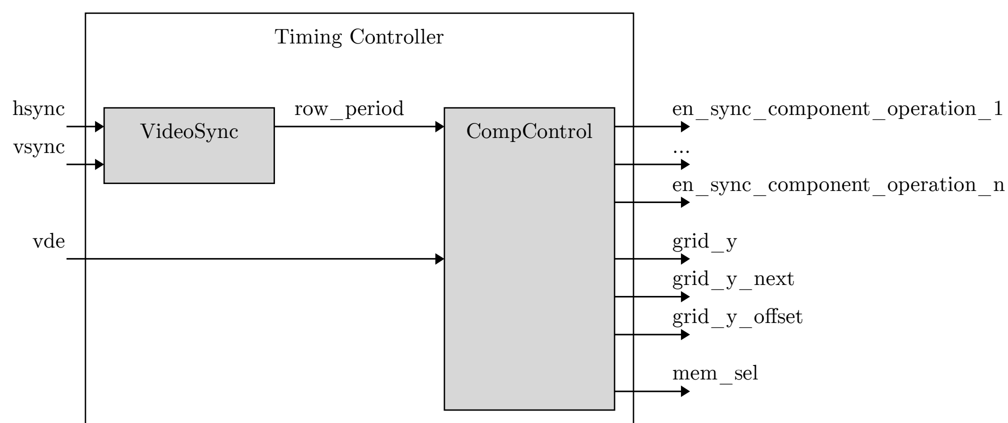


Figure 4.5: Block diagram of the timing controller

VideoSync Process

This process synchronises the row periods with the line periods of the HDMI video frame period. It does so by reading the vsync and hsync signals, which are provided from the video transmitter, to derive a "row period value" that indicates the number of the current row period. This row period value is provided to the CompControl process via row_period. When vsync is asserted, indicating the beginning of a new HDMI video frame period, the row period value is reset to zero. Then, after vsync is de-asserted, hsync will be asserted repeatedly to indicate the beginnings of new line periods. Each time the hsync signal is asserted, the row period value is incremented. Thus, a new row period begins each time a new line period begins, and thus, the row periods are synchronised with the line periods.

CompControl Process

The CompControl process controls other components of MicroGE by providing a set of control and data signals to them during row periods. These signals include:

- Enable signals, with "en_" prefixes, which control the synchronous component operations
- The grid_y, grid_y_next and grid_y_offset signals which specify which one of the canvas rows to produce
- A mem_sel signal which controls the read and write interfaces of the line buffers and the pointer memories

The CompControl process sequences the synchronous component operations by asserting "synchronous enable" signals during row periods. Each synchronous component operation has one corresponding synchronous enable signal. Throughout the remaining part of this thesis, the names of these signals will have "en_" prefixes. When one of these signals is asserted, the corresponding synchronous component operation is performed until the signal is de-asserted. The CompControl process asserts these signals during the row periods so that the synchronous component operations can be performed according to Figure 3.31. Also, since the RenderSet synchronous component operations consist of multiple stages, which exchange information between each other during specific times, as shown in Figure 3.32 and Figure 3.34, the CompControl process asserts the synchronous enable signals of these component operations with specific timing offsets between each other to align the information exchanges.

The CompControl process provides other components with the vertical global grid positions, y_g, and the vertical pixel offset values, dy_{px}, to specify which one of the 90 × P canvas rows to produce, where P is the value of the PixelsPerGrid property. The values of y_g and dy_{px} that specify a specific canvas row, n, to produce are provided to the render units via grid_y and grid_y_offset, respectively, while the values of y_g that specify which primitives to select to render on canvas row n + 1 are provided to the primitive processor via grid_y_next. When the value of row_period, provided from the VideoSync process, is set to zero, y_g and dy_{px} are set to zero as well. Then, during the row periods in which the RenderSet and PrimSelect synchronous component operations should be performed to produce canvas rows, as shown in Figure 3.31, y_g and dy_{px} are incremented. During row periods three to five

and canvas row periods two up to 90 × P - 2, y_g is incremented after every P number of canvas rows that are rendered while dy_{px} is continuously incremented, after each canvas row that is rendered, up to P - 1 and reset back to zero. Additionally, the vde signal is monitored to determine when the row periods should be interpreted as canvas row periods.

The mem_sel signal is provided to the two pointer memories and to the two line buffers to control the read and write accesses to them. During a specific row period, write accesses should be provided to the first pointer memory and the first line buffer while read accesses should be provided to the second pointer memory and the second line buffer. During the following row period, write accesses should be provided to the second pointer memory and the second line buffer while read accesses should be provided to the first pointer memory and the first line buffer. The CompControl process toggles the mem_sel signal between a logic zero and logic one after each row period to control digital logic that alternates these read and write accesses.

4.2.4 Control Interface Unit

A block diagram, which includes all main signal interfaces, of the control interface unit is shown in Figure 4.6. It contains the following component and process:

- The register map provides an interface between MicroGE's controller and MicroGE's components
- The InstructionControl process sequences the asynchronous component operations and transfers data to or from the register map based on instructions received from MicroGE's controller

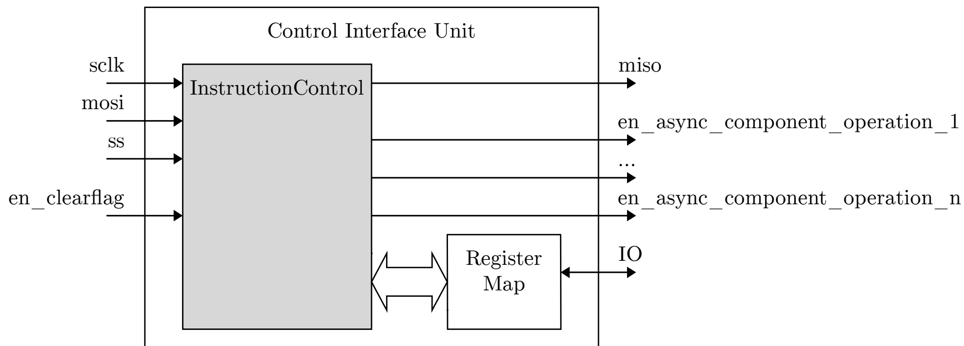


Figure 4.6: Block diagram of the control interface unit

Register Map

MicroGE's register map, as shown in Appendix D, is a set of registers which is connected to MicroGE's components so that those components can be accessed. MicroGE's controller can write data to these registers, and thus, can transfer control, configuration and render information to MicroGE's components. Each of these registers is eight bits wide and is accessed by MicroGE's controller, via an SPI interface, by means of a 7-bit address. Furthermore, these registers are divided

into seven categories, namely: configuration, IO, synchronous control, asynchronous control, primitive data, pixel data and character pointer data.

The registers within the configuration register category can be described as follows:

- The CanvasHorizontalOffset register specifies the horizontal offset of the canvas from the left of the video frame
- The CanvasVerticalOffset register specifies the vertical offset of the canvas from the top of the video frame
- The PixelsPerGrid register specifies the value of the PixelsPerGrid property
- The InterlacedVideo register is a single-bit register that should be set when MicroGE should synchronise with interlaced HDMI video frames

The registers within the IO register category can be described as follows:

- The general-purpose output (GPO) registers are connected to MicroGE's output ports so that those ports can be driven
- The general-purpose input (GPI) registers are connected to MicroGE's input ports so that the values of signals that are provided to those ports can be read

The synchronous control register category consists of a single-bit register, the PrimitiveMemoryUpdate register. When setting the PrimitiveMemoryUpdate bit, the CopyPrim synchronous component operation will be performed during the second row period, as shown in Figure 3.31. After the CopyPrim synchronous component operation is performed, the timing controller asserts the en_clearflag signal to the InstructionControl process so that the PrimitiveMemoryUpdate bit can be cleared.

The asynchronous control register category consists of multiple single-bit registers that are grouped into one 8-bit register. These registers are used to control the asynchronous component operations. The registers within this category can be described as follows:

- If the InsertPrim bit is set, the InsertPrim component operation inserts a primitive into the primitive buffer
- If the DeletePrim bit is set, the DeletePrim component operation removes a primitive from the primitive buffer
- If the ClearPrim bit is set, the ClearPrim component operation clears the primitive buffer
- If the StorePixels bit is set, the StorePixels component operation transfers pixels to the raster memory
- If the StoreChars bit is set, the StoreChars component operation transfers character pointers to the string memory

The primitive data register category consists of a set of registers which is used to store a set of primitive attributes temporarily. When a set of attributes, representing a primitive, is stored in these registers, that primitive can be inserted into the primitive buffer. Also, when an *id* attribute is stored in a specific one of these registers, the primitive within the primitive buffer that has that *id* attribute can be removed.

The registers within the pixel data register category can be described as follows:

- The set of PixelSet registers is used to store a set of pixels, with a cardinality equal to the value of PixelsPerGrid, that should be transferred to the raster memory
- The PixelAddress register specifies to which memory locations of the raster memory this set of pixels should be transferred to

The registers within the character pointer data register category can be described as follows:

- The set of CharacterPointerSet registers is used to store a set of character pointers that should be transferred to the string memory
- The CharacterPointerLength register specifies how many character pointers there are in this set
- The CharacterPointerAddress register specifies to which memory locations of the string memory this set of character pointers should be transferred to

InstructionControl Process

The InstructionControl process acts as an SPI slave that reacts to commands sent from an SPI master, which is MicroGE's controller. After a command is received, the process either updates the register map data or it sequences the asynchronous component operations. The signals of the SPI interface include a source synchronous clock from the master, sclk, a slave select signal from the master, ss, and two signals that are used to transfer serial data to or from the master, miso and mosi, respectively.

During an SPI transaction, the SPI master first transfers a 7-bit address followed by a control bit, via mosi. The 7-bit address specifies the location of the register that should be written to or read from, while the control bit indicates whether a read or write operation should be performed. When a write operation is requested, the master will also transfer a series of data octets directly after it transferred the address. These data octets are temporarily stored in a shift register as the transaction takes place. After the last octet is received, each octet is written to sequential locations in the register map starting from the location at the specified address. On the other hand, when a read operation is requested, the process provides the data located at the specified address as output, via miso, directly after the address was transferred by the master.

When data is written to one of the registers in the asynchronous control register category, one of the "asynchronous enable" signals, with "en_" prefixes, is asserted for a specific duration to sequence a corresponding asynchronous component operation. Each asynchronous component operation has one corresponding asynchronous

enable signal. When one of these signals is asserted, the corresponding asynchronous component operation is performed until the signal is de-asserted. As opposed to the synchronous component operations, the asynchronous components operations can be performed during any time throughout the HDMI video frame period and are not restricted to specific row periods. Thus, MicroGE's controller does not have to meet any timing requirements, relative to the row periods, when requesting the execution of the asynchronous component operations. However, MicroGE's controller needs to provide the appropriate delays between SPI transactions to ensure that an asynchronous component operation is completed before starting with the next SPI transaction. When MicroGE's main clock domain is clocked by a 100 MHz clock, each one of these component operations are performed within 21 μs. MGAPI, which will be discussed in Section 4.4, provides the required delays after SPI transactions to ensure that an asynchronous component operation is completed before the next SPI transaction is started.

4.2.5 Primitive Processor

A block diagram, which includes all main signal interfaces, of the primitive processor is shown in Figure 4.7. It contains the following components and processes:

- The primitive memory stores the primitives that should be rendered on the canvas
- The primitive buffer temporarily stores the primitives before they are transferred to the primitive memory
- The pointer memories store the addresses of primitives that should be rendered on a specific canvas row
- The UpdatePrim process updates the contents of the primitive buffer
- The CopyPrim process transfers the contents of the primitive buffer to the primitive memory
- The PrimSelect process identifies primitives that should be rendered on a specific canvas row
- The PrimSequence process provides primitives, which should be rendered on specific render segments, as output to the render units

Primitive Memory

The primitive memory stores the primitives that should be rendered on the canvas. It consists of six of the DPRAM components, discussed in Section 4.2.2, which each stores one or more of the primitive attributes. The data width of all the DPRAMs are configured to 16-bit, but the entire data width of each is not fully utilised because some of these bits are reserved for future development of MicroGE. Also, since the DPRAM component has a memory capacity of 16 kb, a data width configuration of 16-bit provides 1024 memory locations to store the primitive attributes, which is more than required, 1000.

The primitive attributes are distributed among the six DPRAMs as follows:

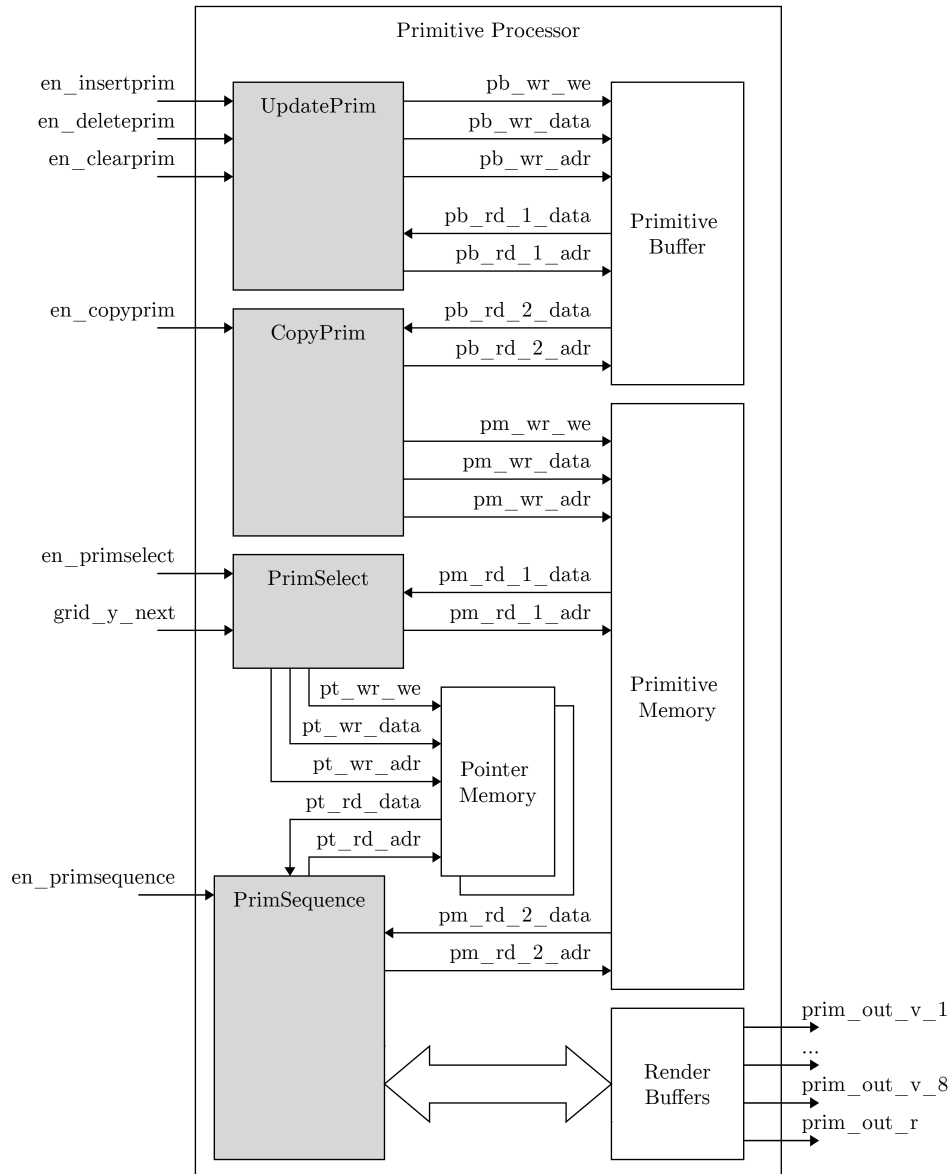


Figure 4.7: Block diagram of the primitive processor

- The first DPRAM stores the *id* attribute as 10-bit values
- The second DPRAM stores the x_i and x_f attributes as two 8-bit values
- The third DPRAM stores the y_i and y_f attributes as two 8-bit values
- The fourth DPRAM stores the z and t attributes as 3-bit and 4-bit values, respectively
- The fifth DPRAM stores the red and green *clr* attributes as two 8-bit values or the *adr* attribute as 16-bit values depending on whether a vector or raster primitive is stored, respectively
- The last DPRAM stores the blue *clr* attribute as 8-bit values

Furthermore, the primitive memory has two read interface sets and one write interface set. Two of the primitive memory's three interface sets, which are never used

at the same time, share one of the DPRAMs' interface sets since each DPRAM only has two interface sets.

The primitive memory's interface sets can be described as follows:

- In the write interface set, pm_wr, used by the CopyPrim process, the primitive provided to pm_wr_data is written to a memory location specified by the address provided to pm_wr_adr if pm_wr_we is asserted
- In the first read interface set, pm_rd_1, used by the PrimSelect process, the primitive stored at the memory location specified by the address provided to pm_rd_1_adr is provided as output via pm_rd_1_data
- In the second read interface set, pm_rd_2, used by the PrimSequence process, the primitive stored at the memory location specified by the address provided to pm_rd_2_adr is provided as output via pm_rd_2_data

Primitive Buffer

The primitive buffer temporarily stores the primitives before they are transferred to the primitive memory. It consists of six DPRAMs which each stores one or more of the primitive attributes. The primitive attributes are distributed among these DPRAMs in the same way the primitive attributes are distributed among the DPRAMs of the primitive memory. Also, the primitive buffer has two read interface sets and one write interface set. Two of these three interface sets, which are never used at the same time, share one of the DPRAMs' interface sets since each DPRAM only has two interface sets.

The primitive buffer's interface sets can be described as follows:

- In the write interface set, pb_wr, used by the UpdatePrim process, the primitive provided to pb_wr_data is written to a memory location specified by the address provided to pb_wr_adr if pb_wr_we is asserted
- In the first read interface set, pb_rd_1, used by the UpdatePrim process, the primitive stored at the memory location specified by the address provided to pb_rd_1_adr is provided as output via pb_rd_1_data
- In the second read interface set, pb_rd_2, used by the CopyPrim process, the primitive stored at the memory location specified by the address provided to pb_rd_2_adr is provided as output via pb_rd_2_data

Pointer Memories

The pointer memories store the addresses of the primitives that should be rendered on a specific canvas row as pointers. There are two pointer memories so that pointers can be written to one, by the PrimSelect process, while pointers are read from the other, by the PrimSequence process. The mem_sel signal is used to alternate the write access between the PrimSelect process and the two pointer memories after each row period. Similarly, the mem_sel signal is also used to alternate the read access between the PrimSequence process and the two pointer memories after each row period. Also, each of the pointer memories consists of one DPRAM component. The data width of the DPRAM component is configured to 16-bit which provides 1024

memory locations for storing pointers. Such a large number of memory locations is required since all of the 1000 primitives may have to be rendered on a single canvas row requiring all of their addresses to be stored in a single pointer memory.

Furthermore, each pointer memory has one read interface set and one write interface set. These interface sets can be described as follows:

- In the write interface set, pt_wr, used by the PrimSelect process, the pointer provided to pt_wr_data is written to a memory location specified by the address provided to pt_wr_adr if pt_wr_we is asserted

- In the read interface set, pt_rd, used by the PrimSequence process, the pointer stored at the memory location specified by the address provided to pt_rd_adr is provided as output via pt_rd_data

UpdatePrim Process

This process performs the InsertPrim, DeletePrim or ClearPrim asynchronous component operations when the en_insertprim, en_deleteprim or en_clearprim enable signals are asserted, respectively. These component operations can update the contents of the primitive buffer by inserting new primitives into memory locations and by shifting primitives that are stored at specific memory locations to different memory locations.

The InsertPrim component operation performs the procedure described in Section 3.7.6 to insert a new primitive, which is stored in the register map, into the primitive buffer. First, each of the primitives stored in the primitive buffer, starting from the one at the first memory location and progressing towards the last one, is read via the pb_rd_1 interface set. As each primitive is read, its x_i attribute is compared with the x_i attribute of the primitive stored in the register map. If one of the primitives read from the primitive buffer has a larger x_i attribute than the x_i attribute stored in the register map, the address of that primitive is stored, and the comparisons end. Then, each of the primitives starting from the one whose address was stored up to the last one is read from the primitive buffer via the pb_rd_1 interface set. Each of the primitives that are read from the primitive buffer is written to the proceeding memory location in the primitive buffer via the pb_wr interface set. This shifts each of those primitives upwards by one memory location. Finally, the primitive stored in the register map is written to the location whose address was stored, via the pb_wr interface set, to complete the procedure of inserting a new primitive into the primitive buffer.

The DeletePrim component operation performs the procedure described in Section 3.7.7, to delete a primitive from the primitive buffer that has an *id* attribute equal to the *id* attribute stored in the register map. First, each of the primitives stored in the primitive buffer is read, starting from the one at the first memory location and progressing towards the last one, via the pb_rd_1 interface set. As each primitive is read, its *id* attribute is compared with the *id* attribute stored in the register map. If the *id* attribute stored in the register map matches that of one of the primitives read from the primitive buffer, the address of that primitive is stored. Then, each of the primitives starting from the one whose address was stored up to the last one is read from the primitive buffer, via the pb_rd_1 interface set, and written to the preceding memory location in the primitive buffer, via the pb_wr

interface set. This shifts all of those primitives down by one memory location, and thus, overwrites the primitive that should be deleted.

The ClearPrim component operation simply sets the value of a register that keeps count of the number of primitives in the primitive buffer to zero. Thus, it seems like all of the primitives were removed. This is more efficient, in terms of processing time, than performing a series of DeletePrim component operations.

CopyPrim Process

This process performs the CopyPrim component operation, upon assertion of the en_copyprim enable signal, to copy all of the primitives stored in the primitive buffer to the primitive memory. The en_copyprim signal is asserted during the second row period if the PrimitiveMemoryUpdate bit, in the register map, is set. During this process, each of the primitives stored in the primitive buffer is read via the pb_rd_2 interface set. Each primitive that is read is written to a corresponding memory location in the primitive memory via the pm_wr interface set.

PrimSelect Process

When the en_primselect enable signal is asserted, this process performs the PrimSelect component operation, as described in Section 3.7.3, to write the addresses of the primitives, that should be rendered on a specific canvas row, as pointers to one of the pointer memories. The specific one of the pointer memories to write to is selected by the mem_sel signal. During this component operation, each primitive stored in the primitive memory is read, starting from the first primitive and progressing towards the last one, via the pm_rd_1 interface set. As each primitive is read, its y_i and y_f attributes are compared with the current value of y_g, provided via grid_y_next, to determine if that primitive is located on the specific canvas row to render. If it is, its address is written, as a pointer, to the selected pointer memory, via the pt_wr interface set.

PrimSequence Process

This process performs the PrimSequence component operation upon assertion of en_primsequence. Throughout this process, the primitives whose render segments should be produced by the render units throughout a grid period set are provided as output. As shown in Figure 4.8, this process consists of two render stages. During grid periods of the first render stage, the primitives whose render segments should be produced are retrieved from the primitive memory using pointers stored in one of the pointer memories. During grid periods of the second render stage, the retrieved primitives are held in the render buffers so that the attributes of those primitives can be provided to the render units. The render units will then use the provided attributes to produce render segments.

During a specific grid period, n, of the first render stage, the primitives whose render segments should be produced by the render units are retrieved from the primitive memory. They are retrieved using the primitives' addresses that were stored as pointers in one of the pointer memories during a preceding row period. As discussed in Section 3.7, these pointers allow the primitives to be retrieved in the order in which they need to be rendered on a canvas row from left to right.

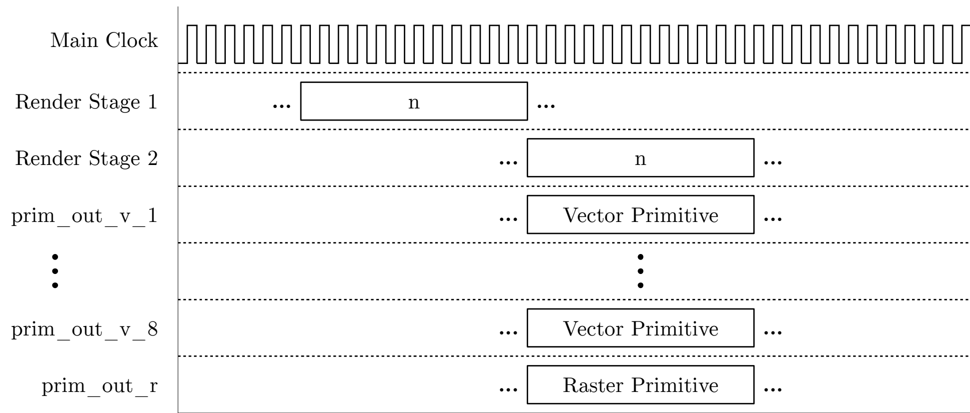


Figure 4.8: Conceptual timing diagram explaining the behaviour of the PrimSequence process

Since the canvas segments of a canvas row are also produced from left to right, each primitive can be retrieved in the order in which they need to be rendered on the canvas segments. Also, since a canvas segment is created from a set of render segments, the primitives can be retrieved in the order in which their render segments need to be produced. As shown in Figure 4.9, during grid period n, eight sequential pointers are retrieved from the selected pointer memory, via the pt_rd interface set, of which the first is located by the "base address". Only eight pointers are retrieved, each to retrieve a corresponding primitive, since at most eight render segments can be produced throughout a grid period set, one for each depth layer, to produce a canvas segment that will be stored in one of the line buffers. The eight pointers are then provided to the primitive memory, via the pm_rd_2 interface set, to retrieve a corresponding set of primitives. The render segments of some of the primitives that are retrieved may have to be produced during grid period set n while the render segments of others of those primitives may have to be produced during proceeding grid period sets. The primitives whose render segments should be produced during grid period set n are provided to grid period n of the second render stage. Then, during grid period n of the second render stage, those primitives are simply held in the render buffers so that their attributes can be provided to the render units via the prim_out_v_1 to prim_out_v_8 ports and the prim_out_r port.

Furthermore, of the eight retrieved primitives whose render segments are produced during grid period set n, more render segments may have to be produced for some of them during proceeding grid period sets while the last render segments of others were produced. If the last render segment of a primitive was produced, that primitive should not be retrieved from the primitive memory again. Before grid period n + 1 of the first render stage, the base address is incremented by the value of the number of primitives that should not be retrieved again. Then, during grid period set n + 1, these primitive will be excluded from the set of eight retrieved primitives and will be replaced by primitives whose render segments may have to be produced during grid period set n + 1 and following grid period sets.

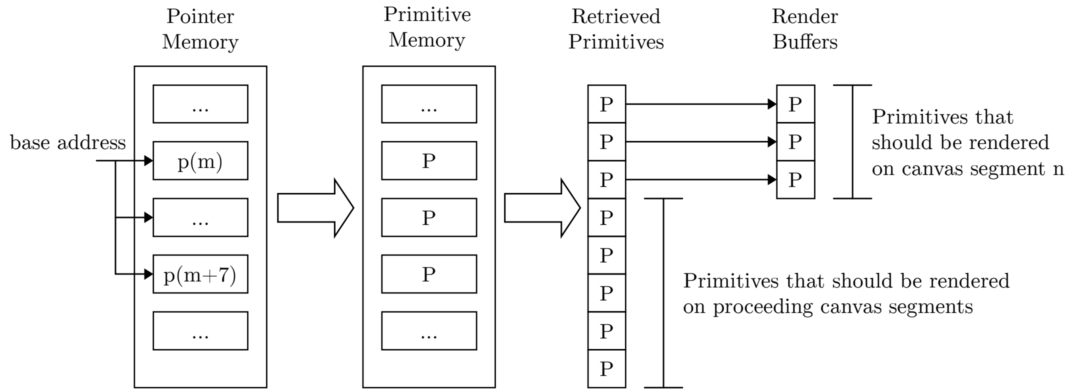


Figure 4.9: Description of how the primitives to provide to the render units are retrieved from the primitive memory during a grid period

4.2.6 Vector Render Unit

A block diagram, which includes all main signal interfaces, of the vector render unit is shown in Figure 4.10. It contains the following components and process:

- The multiplier components, as discussed in Section 4.2.2, are used to perform the multiplication operations required to calculate the values of the render range equations listed in Table 3.2
- The comparator components perform the comparison operations of the render range equations
- The VectorRender process produces render segments of vector primitives

Comparator

The comparator component, shown in Figure 4.11, takes two signal values as input, compares those values, and then provides the result of the comparison as output. It can be configured to perform "less than or equal to" and "greater than or equal to" comparisons. When the output of the comparator is inverted, the comparator can also be used to perform "less than" and "greater than" comparisons. The two values to compare are provided to the comp_a and comp_b inputs while the result of the comparison is provided via comp_out. Also, after the values to compare are provided as input, it takes three clock cycles for the result of the comparison to be provided as output.

VectorRender Process

This process performs the VectorRender component operation upon assertion of the en_vectorrender enable signal. Throughout this process, render segments of vector primitives are produced and are provided as output to the line buffer controller. These primitives are provided from one of the primitive processor's render buffers via prim_in. The vertical position of the canvas row for which render segments should be produced is specified by y_g, via grid_y, and dy_{px}, via grid_y_offset. These

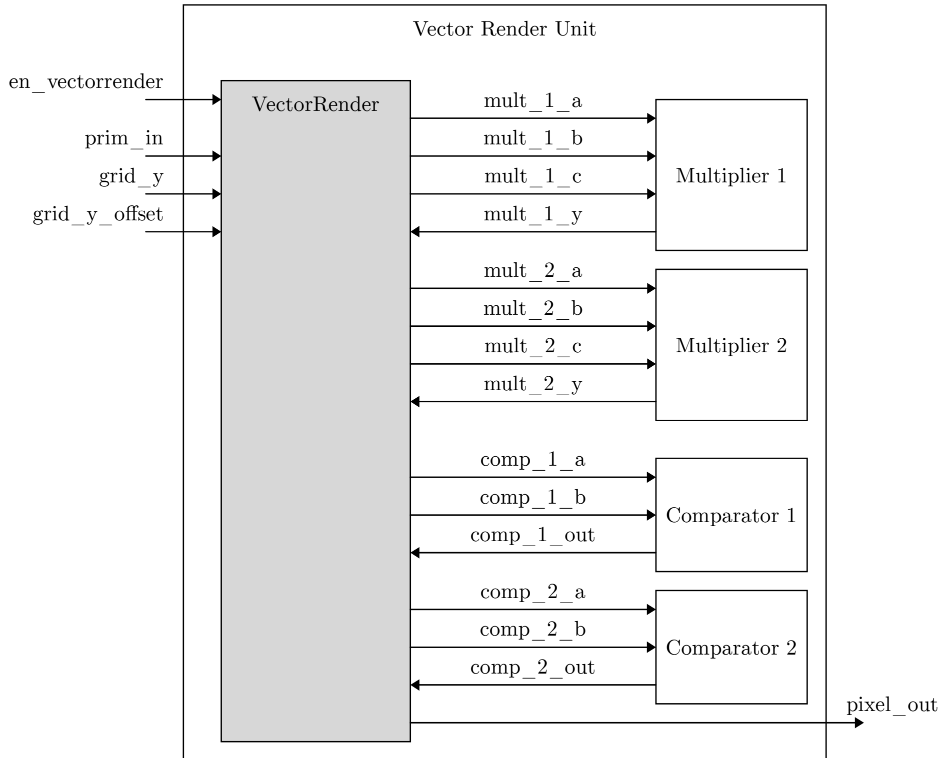


Figure 4.10: Block diagram of the vector render unit

Figure 4.11: Block diagram of the comparator component

signals are provided from the timing controller. As shown in Figure 4.12, this process consists of three render stages. During a grid period of the first render stage, the values of the render range equation of the first pixel within the render segment to produce are determined. During a grid period of the second render stage, the values of the render range equations of the remaining pixels within the render segment to produce are determined. Additionally, the left-hand side (LHS) and right-hand side (RHS) of the render range equations are provided to the comparators as input. During a grid period of the third render stage, the outputs of the comparators are evaluated to determine if the pixels of the render segment to produce should be provided with a colour according to the vector primitive's *clr* attributes or be provided with a blank colour. A blank colour indicates to the line buffer controller that a pixel should be ignored when pixels on the different depth layers are evaluated to choose the highest pixel that should be stored in one of the line buffers as part of the canvas row.

During grid period n of the first render stage, as shown in Figure 4.12, the values, terms, of the render range equation of the first pixel within the render segment to produce are calculated. The type of render range equation whose terms needs to be calculated depends on the type of vector primitive that is provided via prim_in.

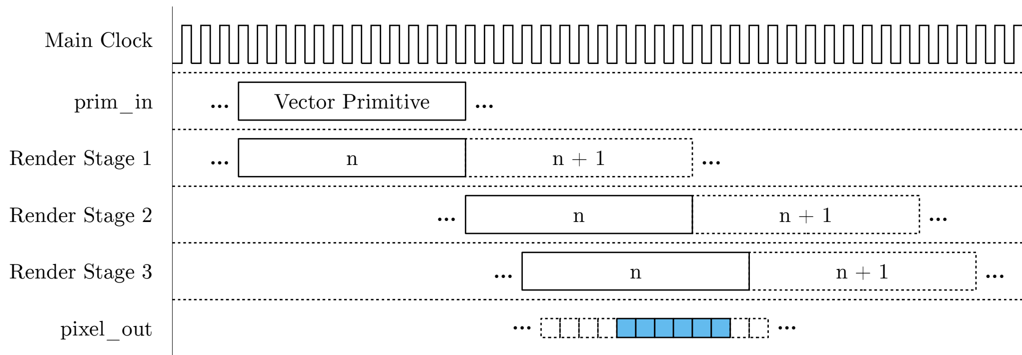


Figure 4.12: Conceptual timing diagram explaining the behaviour of the VectorRender process

Each render range equation consists of multiple terms. The values of these terms are calculated by means of several addition, subtraction and multiplication operations. These operations are based on the procedures discussed in Section 3.3 and Section 3.5. For each multiplication operation, the operands are provided to the mult_1_a and mult_1_b inputs of the mult_1 multiplier so that the calculated product can be provided via mult_1_y three clock cycles later. If direct addition to the product is required, the value to add is provided via mult_1_c. Also, the render range equation of the ellipse primitive requires more multiplication and addition operations to be performed than the render range equations of the other vector primitives, and thus, the second multiplier, mult_2, is provided. Similarly, the operands are provided to the mult_2_a, mult_2_b and mult_2_c inputs so that the calculated product can be provided via mult_2_y three clock cycles later. Other subtraction and addition operations are also performed to calculate the terms of the render range equations, but these operations are performed without the use of the multiplier components. At the end of grid period n of the first render stage, all terms of the render range equation for the first pixel within the render segment to be produced are calculated. These calculated results are provided to grid period n of the second render stage.

During grid period n of the second render stage, the LHS and RHS of the render range equation for each pixel within the render segment to produce are provided to the comparators as inputs so that the outputs of the comparators can be provided to grid period n of the third render stage. While the terms of the render range equation of the first pixel within the render segment to produce are already calculated, the terms of the render range equations of the other pixels still need to be calculated. Instead of calculating all of the terms of each one of the other pixels' render range equations, only the terms that differ from the render range equation of the first pixel are calculated. These include all terms that contain horizontal components of the assessment positions, x, since these differ among each pixel within the render segment. For the line and triangle primitives, $2xdy$ is the only term that is recalculated, while for the ellipse primitive, x^2b^2 is the only term that is recalculated. Instead of performing all of the procedures discussed in Section 3.3 and Section 3.5 to calculate these terms, only part of these procedures are performed to reduce the number multiplication, addition and subtraction operations required. This allows the terms of all the remaining pixels' render range equations to be calculated during grid period n of the second render stage. Finally, as the terms of the render range

equation of each of the pixels within the render segment to produce are calculated, the LHS and RHS of that equation are provided to the inputs of the comparators so that the outputs of the comparators can be provided to grid period n of the third render stage three clock cycles later.

During grid period n of the third render stage, the outputs of the comparators are received and then evaluated to produce the colours of the pixels within the render segment. If the result of a comparison is true, the corresponding pixel within the render segment is provided with a colour according to the vector primitive's *clr* attributes. On the other hand, if the output of the comparison is false, that pixel is provided with a blank colour. As each pixel of the render segment is provided with the appropriate colour, that pixel is provided via pixel_out so that it can be evaluated by the line buffer controller.

4.2.7 Raster Render Unit

A block diagram, which includes all main signal interfaces, of the raster render unit is shown in Figure 4.13. It includes the following components and processes:

- The raster memory stores raster blocks that should be rendered within raster primitives
- The string memory stores character pointers that may be used to locate raster blocks so that those raster blocks can be rendered within string primitives
- The StorePixels process transfers a new set of pixels to the raster memory
- The StoreChars process transfers a new set of character pointers to the string memory
- The RasterRender process produces render segments of raster primitives

Raster Memory

The raster memory stores the raster blocks that are used to render the raster primitives. It can consist of different numbers of DPRAMs, ranging from three to 96, allowing a maximum of 256 $\times$ 256 pixels to be stored. The number of DPRAMs to include within the raster memory is determined by the value of the RAST_MEM_DPRAM_NUM generic statement. The red, green and blue components of the pixels within the raster blocks are stored in separate sets of DPRAMs. The data width of each of these DPRAMs is configured to 8-bit allowing 2048 pixels to be stored in a set of three DPRAMs. The value of RAST_MEM_DPRAM_NUM was set to 24 in this study which provided enough memory capacity to store an image with a resolution of 128 $\times$ 128 pixels.

Furthermore, the raster memory has one write interface set and one read interface set. These interface sets can be described as follows:

- In the write interface set, rm_wr, used by the StorePixels process, the pixel provided to rm_wr_data is written to a memory location specified by the address provided to rm_wr_adr if rm_wr_we is asserted

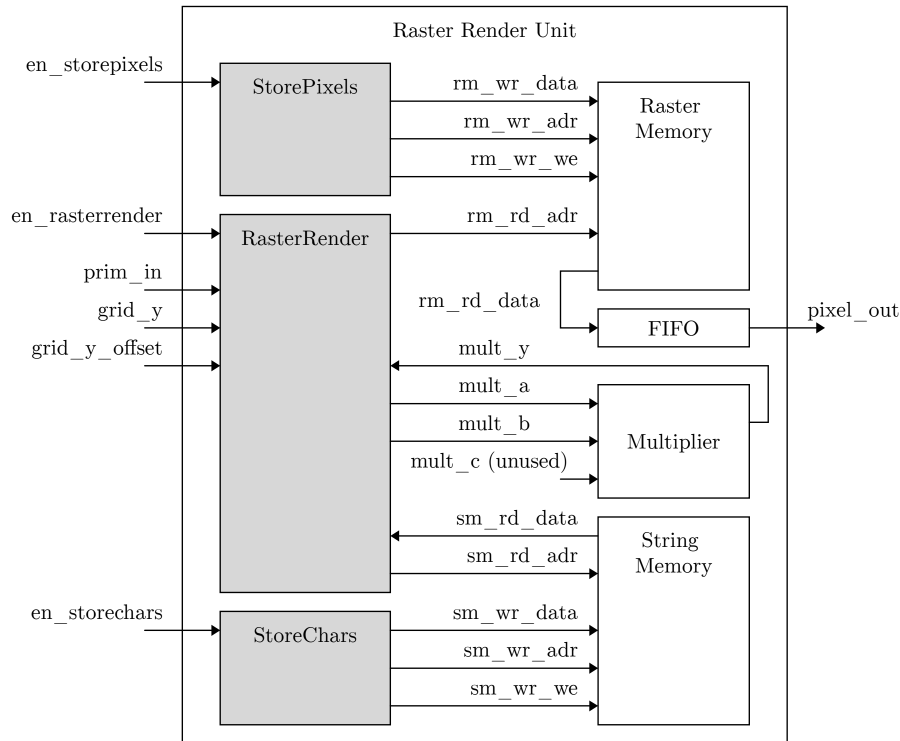


Figure 4.13: Block diagram of the raster render unit

- In the read interface set, rm_rd, used by the RasterRender process, the pixel stored at the memory location specified by the address provided to rm_rd_adr is provided as output via rm_rd_data

StorePixels Process

This process performs the StorePixels asynchronous component operation upon assertion of the en_storepixels enable signal. During this process, a set of pixels stored in the register map is transferred to the raster memory. The number of pixels within this set is determined by the value of PixelsPerGrid. Before this component operation is performed, the set of pixels that should be transferred to the raster memory and the pixel address of the memory locations to which these pixels should be transferred to should be stored in the register map. Then, when the en_storepixels signal is asserted, each pixel stored in the register map is read and then written, via the rm_wr interface set, to adjacent memory locations in the raster memory. The pixel address of the first of these memory locations is specified by the pixel address stored in the register map.

String Memory

The string memory stores character pointers that should be used to locate character raster blocks in the raster memory. It can consist of different numbers of DPRAMs, ranging from one to 64, allowing a maximum of 65536 character pointers to be stored. The number of DPRAMs to include within the string memory is determined by the value of the STR_MEM_DPRAM_NUM generic statement. The data width of the DPRAMs is configured to 16-bit allowing 1024 character pointers to be stored

in each of them. In this study, the value of STR_MEM_DPRAM_NUM was set to one providing enough memory capacity to store 1024 character pointers.

Furthermore, the string memory has one write interface set and one read interface set. These interface sets can be described as follows:

- In the write interface set, sm_wr, used by the StoreChars process, the character pointer provided to sm_wr_data is written to a memory location specified by the address provided to sm_wr_adr if sm_wr_we is asserted
- In the read interface set, sm_rd, used by the RasterRender process, the character pointer stored at the memory location specified by the address provided to sm_rd_adr is provided as output via sm_rd_data

StoreChars Process

This process performs the StoreChars asynchronous component operation upon assertion of the en_storechars enable signal. During this process, a set of character pointers stored in the register map is transferred to the string memory. Before this component operation is performed, the set of character pointers to transfer to the string memory, the number of character pointers in this set, and the address of the memory locations to which these character pointers should be transferred to should be stored in the register map. Then, when the en_storechars signal is asserted, each character pointer stored in the register map is read and then written, via the sm_wr interface set, to adjacent memory locations in the string memory. The address of the first of these memory locations is specified by the address stored in the register map.

RasterRender Process

This process performs the RasterRender component operation upon assertion of the en_rasterrender enable signal. It produces render segments of raster primitives by retrieving a set of pixels from the raster memory and providing those pixels as output to the line buffer controller. These raster primitives are provided from one of the primitive processor's render buffers via prim_in. The vertical position of the canvas row for which render segments should be produced is specified by y_g, via grid_y, and dy_{px}, via grid_y_offset. As shown in Figure 4.14, this process consists of two render stages. During a grid period of the first render stage, the pixel address of the first pixel to produce within the render segment is calculated. During a grid period of the second render stage, the pixel addresses of the remaining pixels to produce within the render segment are calculated. Also, as the pixel addresses are calculated, they are provided to the raster memory so that the corresponding pixels can be retrieved.

During grid period n of the first render stage, as shown in Figure 4.14, the pixel address of the first pixel to produce within the render segment is calculated. This calculation requires several addition and multiplication operations to be performed which are based on the procedures discussed in Section 3.6. Each multiplication operation is performed by means of the multiplier component, while the addition operations are performed without any dedicated components. The operands that should be multiplied are provided to the mult_a and mult_b inputs so that the calculated product can be received via mult_y three clock cycles later. At the end

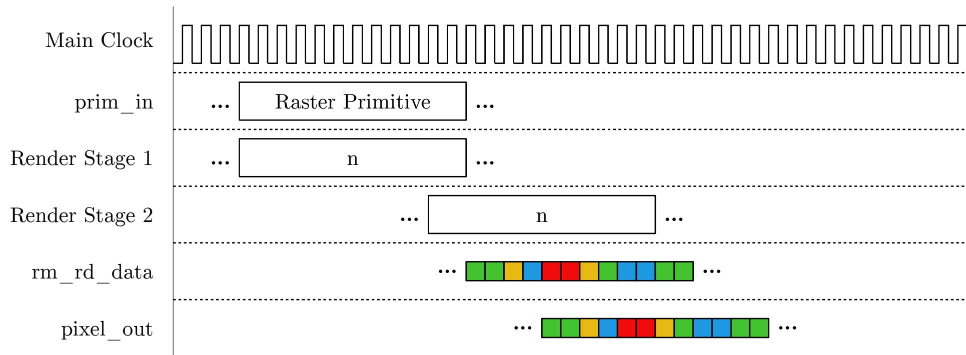


Figure 4.14: Conceptual timing diagram explaining the behaviour of the RasterRender process

of grid period n of the first render stage, the pixel address of the first pixel to produce within the render segment is calculated and is provided to grid period n of the second render stage.

During grid period n of the second render stage, the pixel addresses of all the pixels to produce within the render segment are provided to the raster memory. While the pixel address of the first pixel to produce is already calculated, the pixel addresses of the remaining pixels still need to be calculated. Instead of performing all of the procedures discussed in Section 3.6 to calculate each of the remaining pixel addresses, these pixel addresses are simply determined from the first calculated pixel address. Since the pixels within the render segments are adjacent to each other, the remaining pixel addresses can be determined by incrementing the first calculated pixel address by one for each pixel within the render segment. As each of the pixel addresses is determined, it is used to retrieve a corresponding pixel from the raster memory via the rm_rd interface set. Before the retrieved pixels are provided to the line buffer controller as output, via pixel_out, they are provided to a FIFO buffer so that their output can be delayed. This delay is required to allow render segments of the raster render unit and vector render units to be provided to the line buffer controller at the same time.

4.2.8 Line Buffer Controller

A block diagram, which includes all main signal interfaces of the line buffer controller, is shown in Figure 4.15. It contains the following components and processes:

- The line buffers are used to store canvas rows whose pixels should be transmitted to a display monitor
- The WriteLine process produces canvas segments and stores those canvas segments, as part of a canvas row, in one of the line buffers
- The ReadLine process provides the pixels of a canvas row to a video transmitter so that those pixels can be transmitted to a display monitor

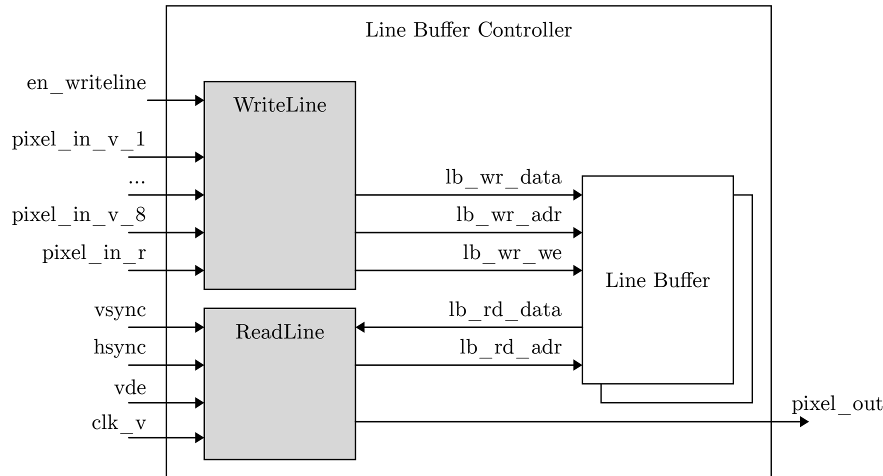


Figure 4.15: Block diagram of the line buffer controller

Line Buffers

The line buffers are used to store canvas rows so that the pixels of those canvas rows can be transmitted to a display monitor by the video transmitter. There are two line buffers so that pixels can be written to one, by the WriteLine process, while pixels are read from the other, by the ReadLine process. The mem_sel signal is used to alternate the write access between the WriteLine process and the two line buffers after each row period. Similarly, the mem_sel signal is also used to alternate the read access between the ReadLine process and the two line buffers after each row period. Each line buffer consists of three DPRAM components which each stores either the red, green or blue colour components of the pixels of a canvas row. The data width of the DPRAMs is configured to 8-bit providing each line buffer with 2048 memory locations for storing pixels. This is more than the number of memory locations required, 1920, to store canvas rows for the highest video resolution that MicroGE should be able to support, 1920 × 1080, as specified in Section 1.4.

Furthermore, each line buffer has one read interface set and one write interface set. These interface sets can be described as follows:

- In the write interface set, lb_wr, used by the WriteLine process, the pixel provided to lb_wr_data is written to a memory location specified by the address provided to lb_wr_adr if lb_wr_we is asserted

- In the read interface set, lb_rd, used by the ReadLine process, the pixel stored at the memory location specified by the address provided to lb_rd_adr is provided as output via lb_rd_data

WriteLine Process

This process performs the WriteLine synchronous component operation upon assertion of en_writeline. Throughout this process, the render segments provided by the render units are evaluated to produce canvas segments that can be stored in one of the line buffers as part of the canvas row whose pixels will be transmitted

by the video transmitter. As shown in Figure 4.16, this process consists of two render stages. During grid period n of the first render stage, the pixels within the render segments that are provided from all of the render units are evaluated. These pixels are provided from the raster render unit, and the vector render units via the pixel_in_r port and the multiple pixel_in_v ports, respectively. As these pixels are provided, they are evaluated to identify the pixels on the highest depth layer. The highest pixels are provided to grid period n of the second render stage. During grid period n of the second render stage, the provided pixels are stored in the line buffer, via the lb_wr interface set, as part of the canvas segment that forms part of the entire canvas row whose pixels will be transmitted by the video transmitter.

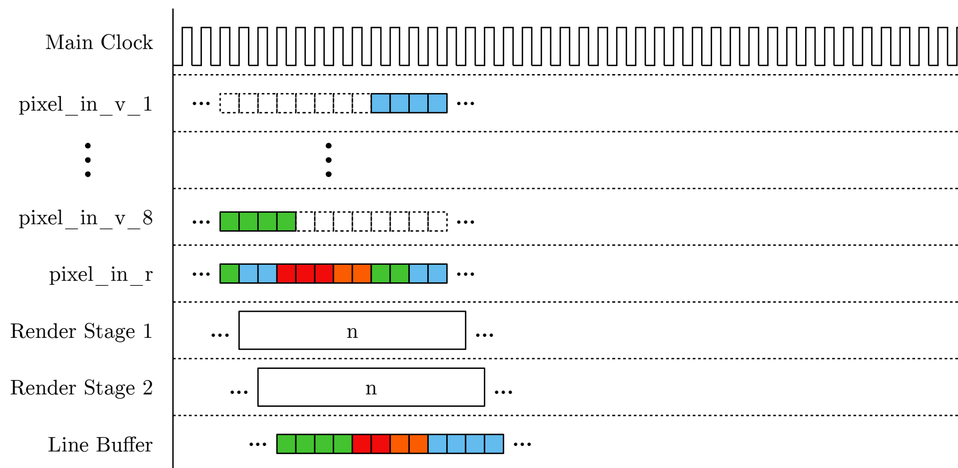


Figure 4.16: Conceptual timing diagram explaining the behaviour of the WriteLine process

ReadLine Process

This process performs the ReadLine synchronous component operation to provide the pixels of a canvas row to the video transmitter, during canvas row periods, for transmission to a display monitor. This process uses the pixel clock that is provided to the video transmitter, clk_v, as its operating clock so that it can perform all of its operations within the video transmitter's clock domain. It reads the pixels of the canvas row, via the lb_rd interface set, stored in the line buffer selected by mem_sel. While other processes are controlled by enable signals, the ReadLine process is controlled by the vsync, hsync and vde signals provided from the video transmitter to control the transmission of the video frame. Reading the vsync and hsync signals allows the ReadLine process to synchronise the row periods with the video transmitter's line periods in the same way that the VideoSync process does, as discussed in Section 4.2.3. Reading the vde signal allows ReadLine to determine if a line period is an active line period, in which canvas row periods may occur. ReadLine uses the value provided to the CanvasVerticalOffset register to determine in which of the active line periods canvas row periods should occur. The larger the value of CanvasVerticalOffset, the more the canvas should be shifted down relative to the top of the video frame, and thus, the more initial active line periods are skipped before the first canvas row period occurs within an active line period.

At the beginning of a canvas row period, the first part of the canvas row pixels is read from the line buffer and stored in registers which provide faster read access than the line buffer itself. This is simply to compensate for the delay of the line buffer later on. Then, if vde is asserted, and the value of the CanvasHorizontalOffset register is zero, the pixels from these registers are immediately provided as output, via pixel_out, so that these pixels can be transmitted. When the last of these pixels are provided from these registers, the remaining pixels of the canvas row are provided as output from the line buffer allowing the entire canvas row to be transmitted to the display monitor. If the CanvasHorizontalOffset register is set to a value larger than zero, the pixels are not provided as output for a corresponding number of pixel clock cycles which results in the canvas to be shifted to the right relative to the left edge of the video frame.

4.3 Additional VHDL Implementation

4.3.1 Clock Generator

The clock generator component generates three clock signals from a single 50 MHz reference clock. A block diagram, which includes all main signal interfaces, of the clock generator is shown in Figure 4.17. The clock signals generated by the clock generator include clk_microge, clk_edid and clk_dcm. They are provided to MicroGE, the EDID reader, and a dynamic clock frequency reconfiguration component within the video transmitter, respectively. The clock generator uses the PLL_BASE component, which uses a phase-locked loop (PLL) circuit inside the Spartan-6 LX25 FPGA, to generate these clock signals [28]. This PLL circuit tunes the output clock frequency of a voltage-controlled oscillator (VCO) to a multiple of the reference clock frequency, clk_ref. The relationship between the reference clock frequency and the VCO clock frequency is determined by the PLL's reference, D, and feedback, M, frequency dividers, as shown in Equation 4.1 [28]. The VCO clock is provided as output via multiple clock channels which each also has a frequency divider, O_n. These dividers allow the frequencies of clk_microge, clk_edid and clk_dcm to be chosen as a submultiple of the VCO clock frequency. Table 4.1 lists the frequencies of the generated clocks along with the corresponding O_n values.

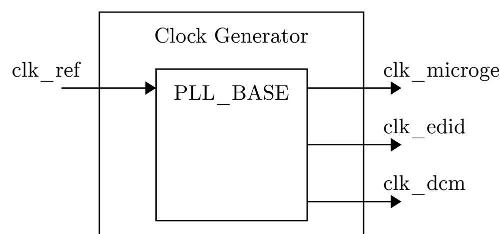


Figure 4.17: Block diagram of the clock generator which includes a Xilinx PLL_BASE component [28]

$$f_n = \frac{f_{in} \times M}{D \times O_n} \tag{4.1}$$

Where:
f_n = channel n clock frequency
f_{in} = reference clock frequency
M = feedback divider
D = reference divider
O_n = channel n divider

Clock Name	Channel n Clock Frequency (f_n)	Channel n Divider (O_n)
clk_microge	100 MHz	4
clk_edid	50 MHz	8
clk_dcm	50 MHz	8

Table 4.1: Frequencies of the clocks generated by the clock generator component, where $M = 8$, $D = 1$ and $f_{in} = 50$ MHz

4.3.2 Video Transmitter

The video transmitter component can transmit video frames of seven different video modes. These video modes are supported by HDMI and DVI display monitors. A block diagram, which includes all main signal interfaces, of the video transmitter is shown in Figure 4.18. The video transmitter consists of the following components:

- The Xilinx TMDS transmitter produces the TMDS signals required to transmit video frame data to a display monitor [29]
- The video clock generator produces the clock signals required to transmit video frame data to a display monitor
- The video mode lookup table (LUT) contains the video mode parameters of the seven supported video modes
- The video timing generator produces the video timing signals required to transmit the video frames of the seven supported video modes

Video Mode LUT

The video mode LUT component contains video mode parameters which are used to configure the video transmitter to transmit the video frames of a specific one of the supported video modes. The video mode whose video frames should be transmitted is specified by the value provided via vid_mode_sel. When a value is provided via vid_mode_sel, the corresponding video mode parameters are selected from the video mode LUT. Those parameters are then provided as output, via vm_params, to the video clock generator and video timing generator so that those components can produce the signals required to transmit the video frames of the selected video mode. In this study, vid_mode_sel is controlled from one of MicroGE's output ports.

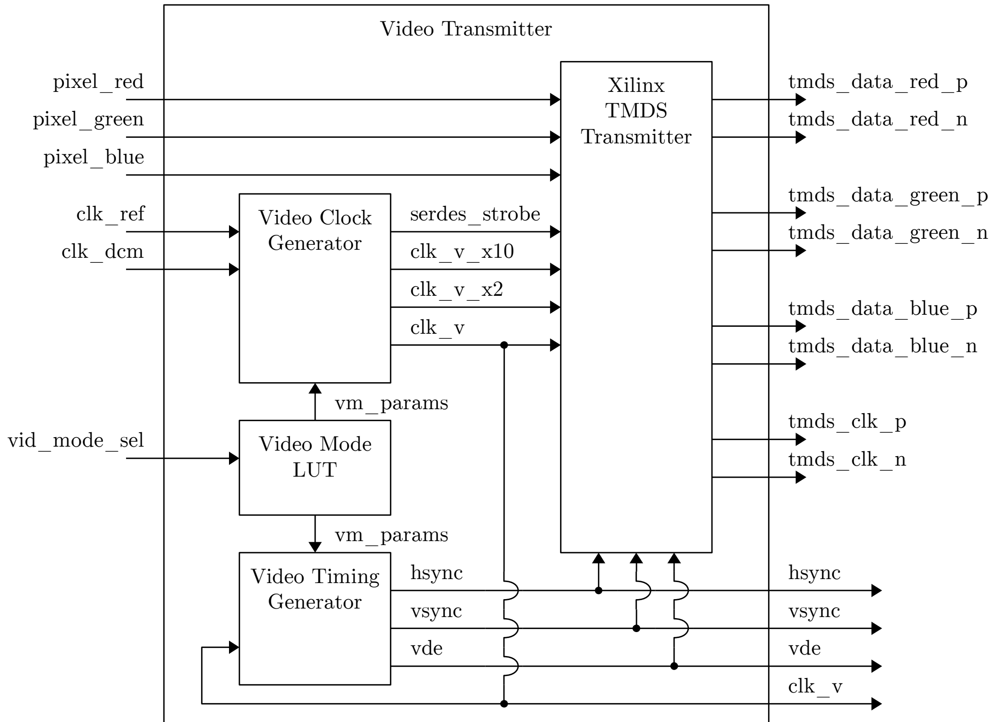


Figure 4.18: Block diagram of the video transmitter which includes a Xilinx TMDS transmitter [29]

Video Clock Generator

The video clock generator component produces three clock signals, clk_v, clk_v_x2 and clk_v_x10, and a strobe signal, serdes_strobe, which are provided to the Xilinx TMDS transmitter. The pixel clock, clk_v, is also provided as output so that MicroGE can use it as a reference to synchronise with the video transmitter. The frequency of clk_v can be set to any of the pixel clock frequencies of the supported video modes, listed in Table 4.2. The specific one of these frequencies that is chosen for clk_v is determined by the video mode parameters provided via vm_params. The other two clock signals, clk_v_x2 and clk_v_x10, have two and ten times the frequency of clk_v, respectively. The strobe signal, serdes_strobe, is a periodic pulse signal with a frequency equal to that of clk_v. These clock and strobe signals are generated from an external 50 MHz reference clock, provided via clk_ref, using clocking resources of the Spartan-6 LX25 FPGA. One of these clocking resources, DCM_CLKGEN, allows the frequencies of the clock and strobe signals to be adjusted during runtime [28]. This allows the video transmitter to change between video modes during runtime. This reconfiguration capability requires a separate 50 MHz reference clock which is provided via clk_dcm.

Video Timing Generator

The video timing generator produces the hsync, vsync and vde timing signals to indicate the active and synchronisation periods of the HDMI video frame periods. It produces these timing signals according to the video mode parameters provided from the video mode LUT component via vm_params. It has two counters, called

Video Mode	Pixel Clock Frequency (MHz)	Standard
800 × 600 at 60 Hz	40.00	VESA
1024 × 768 at 60 Hz	65.00	VESA
1280 × 768 at 60 Hz	79.50	VESA
1280 × 800 at 60 Hz	83.50	VESA
1360 × 768 at 60 Hz	85.50	VESA
1280 × 720p at 60 Hz	74.25	CEA
1920 × 1080i at 60 Hz	74.25	CEA

Table 4.2: The video modes supported by the video transmitter [19], [20]

the "vertical" and "horizontal" counters, that repeatedly increment from zero up to maximum specified values. The horizontal counter increments up to a value equal to the HDMI video frame width and then resets to zero. Every time the horizontal counter is reset, the vertical counter is updated. Similarly, the vertical counter increments up to a value equal to the HDMI video frame height and then resets to zero. When the values of these two counters are within specific ranges, the hsync, vsync and vde timing signals are asserted to indicate the horizontal synchronisation periods, vertical synchronisation periods, and active periods of the HDMI video frame periods, respectively.

Xilinx TMDS Transmitter

This component transmits the HDMI video frame data to a display monitor via four differential TMDS channels using clock signals, control signals, and pixel data as inputs. The pixel data includes the red, green and blue colour components, pixel_red, pixel_green and pixel_blue, respectively, of the pixels to transmit to the display monitor. The control signals include hsync, vsync and vde. Upon the rising edge of the pixel clock, clk_v, the control signal values and the pixel data are sampled and then encoded to 10-bit characters which are transmitted across the tmds_data_red, tmds_data_green and tmds_data_blue signal pairs. Also, a clock signal whose frequency is equal to that of clk_v is transmitted across the tmds_clk signal pair so that the TMDS receiver within the display monitor can recover the HDMI video frame data. These transmission processes also require the clock signals provided via clk_v_x2 and clk_v_x10, and the periodic strobe signal provided via serdes_strobe. [29]

4.3.3 EDID Reader

The EDID reader reads EDID/E-EDID ROM data of a display monitor, via a DDC interface, to determine that monitor's display capabilities. It then provides data to MicroGE, via one of MicroGE's input ports, that indicates which of the video modes supported by the video transmitter, discussed in Section 4.3.2, is supported by that display monitor as well. A block diagram, which includes all main signal interfaces, of the EDID reader is shown in Figure 4.19. The EDID reader contains the following component and process:

- The I^2C master component, created by [30], produces the I^2C signals required to transfer data across an I^2C bus

- The EDIDRead process controls the I^{2}C master so that a specific set of the display capability information stored within an EDID/E-EDID ROM can be retrieved and then be provided to MicroGE

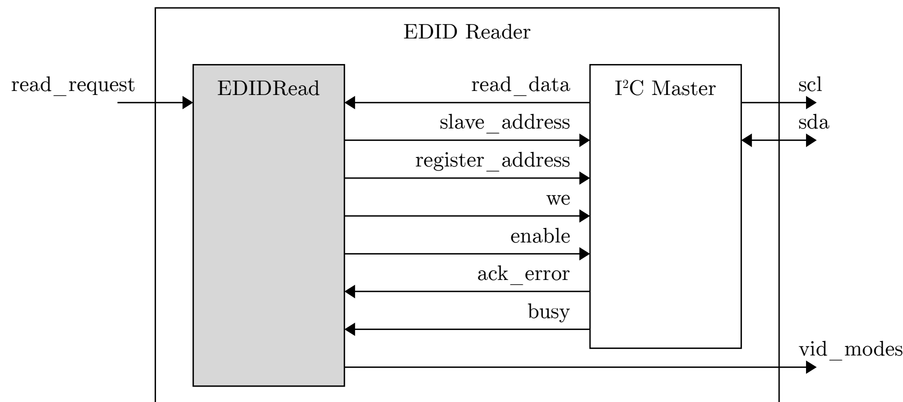


Figure 4.19: Block diagram of the EDID reader which includes the I^{2}C master component created by [30]

I^{2}C Master

The I^{2}C master is an open source VHDL core, created by [30], that can communicate with I^{2}C slaves across an I^{2}C bus. It is controlled via the slave_address, register_address, we and enable signals, to provide corresponding I^{2}C clock and data signals via scl and sda, respectively. The address of the I^{2}C slave device that should be accessed is specified by slave_address while the address of the register that should be accessed within the slave is specified by register_address. When the enable signal is asserted, a read or write data transaction is performed depending on the value provided to we. During a transaction, the busy status signal is asserted to indicate to the component controlling the I^{2}C master that the transaction is still taking place. When the transaction is complete, the data read from the register is provided via read_data, if a read transaction was requested. Also, if there was an error during the transaction, the ack_error flag is asserted.

EDIDRead Process

The EDIDRead process reads the data of a specific set of registers within a display monitor's EDID/E-EDID ROM, using the I^{2}C master, to determine which of the video modes supported by the video transmitter component, discussed in Section 4.3.2, is supported by the display monitor as well. When read_request is asserted, the data of the EDID/E-EDID registers that are listed in Table 4.3 are read and then evaluated. The data read from the Established Timings registers, Standard Timing Identification registers, and Detailed Timing Description registers are evaluated to determine if any of the five VESA video modes are supported. If the EDID/E-EDID ROM contains a single CEA extension, the data read from the CEA Data Block Collection registers are evaluated to determine if any of the two CEA video modes are supported. At the end of the process, a value is provided to MicroGE's input

ports, via vid_modes, so that MicroGE's controller can read this value and then determine which of the video modes supported by the video transmitter component is supported by the display monitor as well. [20], [21]

Address	Description
35 to 36	Established Timings registers 1 to 2
38 to 53	Standard Timing Identification registers 1 to 8
54 to 125	Detailed Timing Description registers 1 to 4
126	Extension flag
128	CEA tag
129	CEA revision number
130	Address of register following the CEA Data Block Collection (n)
132	First CEA Data Block Collection register
...	One of multiple CEA Data Block Collection registers
n - 1	Last CEA Data Block Collection register

Table 4.3: List of the registers read from the EDID/E-EDID ROM by the EDIDRead process [20], [21]

4.4 MGAPI C Implementation

4.4.1 MGAPI Overview

MGAPI is a C software library that enables a device such as an MCU or a soft-core processor to control MicroGE. A block diagram of how MGAPI should be utilised on a device, in this case, the Arduino Due, is shown in Figure 4.20. MGAPI provides a set of high-level functions that can be used to create a graphics rendering application. These high-level functions allow MicroGE to be controlled without having to understand the details behind the SPI transactions necessary to control MicroGE. Each of the high-level functions consists of a set of low-level functions. The low-level functions control the platform-specific hardware resources of the device MGAPI is running on to produce the SPI transactions required to control MicroGE. MGAPI can be ported to another device by simply changing the code within the low-level functions to support the platform-specific hardware resources of that device.

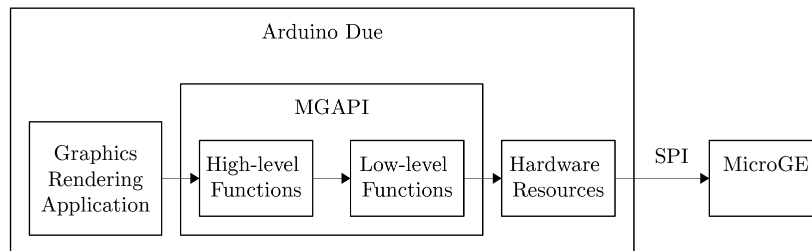


Figure 4.20: Block diagram of how MGAPI should be utilised

MGAPI was implemented on an Arduino Due using the Arduino IDE. An Arduino Due was chosen so that MGAPI can be used on a wide variety of other Arduino boards as well without having to make significant changes to the low-level functions. Also, since an Arduino Due does not nearly have enough processing power to produce graphics to an HDMI monitor on its own, it is a good demonstration of the graphics rending capabilities of MicroGE.

4.4.2 MGAPI High-Level Functions

MGAPI's high-level functions, as listed in Appendix E, are divided into configuration, IO and render categories. These categories can be described as follows:

- The configuration category contains functions that provide configuration information to MicroGE
- The IO category contains functions that control MicroGE's IO ports
- The render category contains functions that update the contents of MicroGE's primitive buffer, primitive memory, string memory and raster memory

Furthermore, the configuration functions can be described as follows:

- MgapiSetCanvasHorizontalOffset sets the value of the CanvasHorizontalOffset register to specify the horizontal offset of the canvas from the left of the video frame

- MgapiSetCanvasVerticalOffset sets the value of the CanvasVerticalOffset register to specify the vertical offset of the canvas from the top of the video frame
- MgapiSetPixelsPerGrid sets the value of the PixelsPerGrid register, and thus, the PixelsPerGrid property
- MgapiSetInterlacedVideo sets the value of the InterlacedVideo register to allow MicroGE to synchronise with interlaced HDMI video frames

The IO functions can be described as follows:

- MgapiSetGpoOutput sets the value of one of the GPO registers that are connected to MicroGE's output ports
- MgapiReadGpiInput reads the values of one of the GPI registers that are connected to MicroGE's input ports

The first four render functions are used to update the contents of the primitive buffer and to transfer the primitive buffer's contents to the primitive memory. The first function, MgapiInsertPrimitive, inserts a primitive into the primitive buffer by first transferring a set of primitive attributes to the register map. It then sets the InsertPrim bit so that MicroGE can perform the InsertPrim component operation to insert that primitive into the primitive buffer. The MgapiDeletePrimitive function removes a primitive from the primitive buffer by first transferring an *id* attribute to the register map. It then sets the DeletePrim bit so that the DeletePrim component operation can remove the primitive with that *id* from the primitive buffer. The MgapiClearPrimitiveBuffer function sets the ClearPrim bit so that the ClearPrim component operation can clear the primitive buffer. The MgapiUpdatePrimitive-Memory function updates the primitive memory with the primitive buffer's contents by setting the PrimitiveMemoryUpdate bit. The primitive buffer's contents are then copied to the primitive memory, during the second row period, by the CopyPrim component operation.

The next render function, MgapiTransferRasterBlock, transfers a raster block to the raster memory. If the raster block that should be transferred has to be scaled to fit the current raster block size of MicroGE, this resizing functionality should be implemented as part of the graphics rendering application code since MGAPI currently does not have a capability to resize raster blocks. Once the raster block is sized appropriately, the raster block and the base address specifying where the raster block should be stored in the raster memory should be provided to the MgapiTransferRasterBlock function. Each row of pixels within the raster block is then transferred to the raster memory individually. During each of these transfers, the row of pixels and a pixel address indicating where that row of pixels should be stored are first transferred to the PixelSet and PixelAddress registers, respectively. After each row of pixels is transferred to the PixelSet registers, the StorePixels bit is set so that the StorePixels component operation can transfer that row of pixels from the PixelSet registers to the raster memory.

The last render function, MgapiTransferCharacterPointers, transfers a set of character pointers to the string memory. The set of character pointers to transfer, the number of character pointers in this set, and an address specifying where this set should be stored in the string memory should be provided to the Mgapi-TransferCharacterPointers function. This set of character pointers is then divided

into subsets of character pointers that are each transferred to the string memory individually. During each transfer, the subset of character pointers, the number of character pointers in this subset, and an address specifying where this subset should be stored in the string memory are provided to the CharacterPointerSet, CharacterPointerLength and CharacterPointerAddress registers, respectively. After each subset of character pointers is transferred to the CharacterPointerSet registers, the StoreChars bit is set, so that the StoreChars component operation can transfer that subset of character pointers from the CharacterPointerSet registers to the string memory.

4.4.3 MGAPI Low-level Functions

All of the high-level functions contain low-level functions whose contents can be changed to port MGAPI to another platform. In this study, these low-level functions were implemented using functions provided by Arduino's software libraries.

Furthermore, these low-level functions, as listed in Appendix E, can be described as follows:

- MgapiWriteData writes a byte via the SPI interface
- MgapiReadData reads a byte via the SPI interface
- MgapiDelay provides a delay, specified in microseconds
- MgapiInit performs all initialisation required by the other low-level functions

Chapter 5

Hardware Development

5.1 Hardware Development Overview

The objective of the hardware development was to develop a test board that MicroGE could be verified on. This board allows MicroGE to provide video output to an HDMI display monitor from instructions received by an Arduino Due running MGAPI. Figure 5.1 shows a block diagram of the hardware design of the test board. The hardware design contains a Xilinx Spartan-6 LX25 FPGA which was used to test the VHDL design discussed in Section 4.1. This FPGA was chosen since it has a small number of resources compared to FPGAs of other Xilinx FPGA families. This forced the VHDL design to be implemented within limited resources, and thus, forced the main requirement of the design of MicroGE to be fulfilled. This FPGA also contains all of the hardware resources required to implement the video transmitter, discussed in Section 4.3.2, within the FPGA itself. This allowed external video transmission electronic components to be avoided during the hardware design, which simplified the design of the board.

Figure 5.1 also shows the other design components of the hardware design. The first design component is used for power distribution. The power distribution includes a power connector from which power is provided to the board, and four voltage regulators which derive different supply voltages for the other components on the board. The second design component is used as the interface between the Arduino Due, running MGAPI, and the FPGA that is configured with the FPGA design containing MicroGE. This interface contains an insulation-displacement contact (IDC) connector for the Arduino Due to connect to. The third design component is used for configuration of the FPGA. It consists of a Joint Test Action Group (JTAG) connector and a flash programmable read-only memory (PROM) that are connected to each other and to the FPGA. A Xilinx JTAG programmer can connect to the JTAG connector so that a bitstream can be uploaded onto either the FPGA or onto the PROM. The fourth design component is the HDMI interface which is used to provide MicroGE's video output to an HDMI display monitor and also to read that monitor's display capabilities. This interface contains an HDMI connector for an HDMI cable to connect to. It also contains a port protection device, between the HDMI connector and the FPGA, that protects the FPGA from ESDs and also provides voltage level translation to some of the HDMI signals. The fifth design component provides a reference clock to the FPGA so that the clocks required by the FPGA design can be derived. The sixth design component is exter-

nal memory which can be used to store frame buffer data. This was added to the hardware design for in case MicroGE's "frame-buffer-less" architecture could not be realised. Thus, it was not used. The last design component, "Other", contains all of the remaining electronic components of the hardware design, such as push-buttons, general-purpose input/output (GPIO) connectors and LEDs, that are required for debugging purposes.

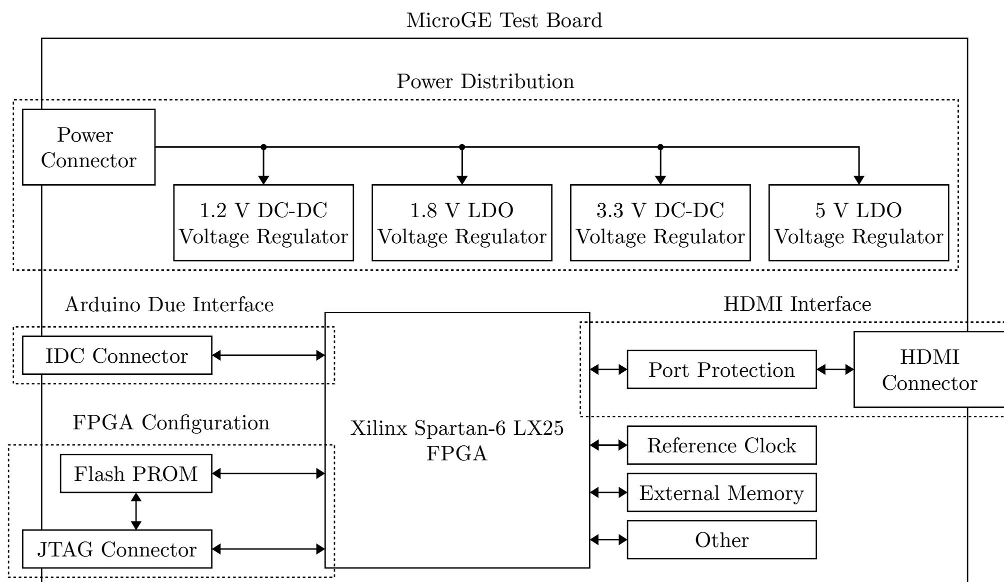


Figure 5.1: System design of test board

5.2 Circuit Design

5.2.1 Power Distribution

Power Requirements

The power requirements of all the main components of the hardware design are listed in Table 5.1. Some of these components require more than one supply voltage. The power requirements of the Spartan-6 LX25 FPGA were determined by the Xilinx Power Estimator tool. This tool calculates the power usage of the FPGA depending on the number of FPGA resources used. At the point of the study when the test board was designed, it was not possible to determine what resources the FPGA design would use, and thus, the worst-case power usage was estimated for. The power requirements of the other main components were determined from their datasheets as well as the power requirements of the passive components, such as bias resistor, that are connected to them.

Power Rails

The hardware design contains the following power rails to provide the required supply voltages to the components:

Component	Voltage (V)	Current (mA)	Power (mW)
Reference Clock	3.3	50	165
External Memory (1)	3.3	10	33
External Memory (2)	3.3	200	660
Flash PROM (1)	1.8	100	180
Flash PROM (2)	3.3	10	33
Port Protection Device (1)	5	50	250
Port Protection Device (2)	3.3	50	165
FPGA (1)	1.2	2000	2400
FPGA (2)	3.3	2000	6600

Table 5.1: Power requirements of the hardware design

- The 3V3 power rail provides 3.3 V to the reference clock, flash PROM, port protection device, FPGA, and external memory
- The 1V8 power rail provides 1.8 V to the flash PROM
- The 1V2 power rail provides 1.2 V to the FPGA
- The 5V power rail provides 5 V to the port protection device
- The 12V power rail is provided with 12 V from an external power supply so that the supply voltages of the other power rails can be derived

Power Distribution Scheme

The 1V2, 1V8, 3V3 and 5V power rails are powered from the 12V power rail according to the power distribution scheme shown in Figure 5.2. The 1V2 and 3V3 rails are powered from the 12V rail using step-down DC-DC voltage regulators since those rails have high power requirements. The 5V and 1V8 rails are powered by low-dropout (LDO) linear voltage regulators since those rails do not have high power requirements. Step-down DC-DC voltage regulators could have been used as well, but LDO regulators are simply cheaper. The one LDO provides 5 V directly from the 12V rail while the other LDO provides 1.8 V from the 3V3 rail so that the voltage drop across that LDO can be reduced.

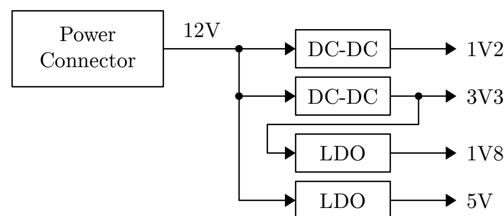


Figure 5.2: Block diagram of the power distribution scheme of the hardware design

Power Connector

Figure 5.3 shows the power connector, J8, along with other components that provide protection capabilities and status output. A light-emitting diode (LED), D11,

is connected to the positive terminal of the connector, via a 10 kΩ resistor, R31, for indicating when power is provided to the test board. The transient voltage suppressor (TVS) diode, D12, is connected in parallel with the connector terminals for ESD protection purposes. If there is a large electrical potential difference between the two terminals of the connector, the TVS diode will conduct, and thus, redirect ESD currents away from the rest of the circuit that should be protected. Furthermore, reverse polarity protection circuitry is provided using a p-channel enhancement mode metal oxide semiconductor field-effect transistor (MOSFET), Q2, along with two 10 kΩ resistors, R34 and R35, that form a voltage divider. If the polarity of the voltage applied to the connector is reversed, the voltage at the gate of the MOSFET will be positive with respect to voltage at its source. Thus, the MOSFET will not conduct, and the rest of the circuit will be protected. On the other hand, if the voltage is provided in the correct polarity, current will flow through the body diode of the MOSFET and through the resistors, and thus, a voltage will be established at the gate of Q2. Since the voltage at the gate will then be negative with respect to the voltage at the source, the MOSFET will conduct, and thus, power will be provided to the rest of the circuit. Finally, two 0 Ω jumper resistors are provided, both marked "do not populate" (DNP), to indicate that they should not be populated during the PCB assembly process. The first, R33, should be populated manually after the input voltage is verified to be 12 V while the second, R32, provides the option of bypassing the reverse polarity protection circuitry, if required.

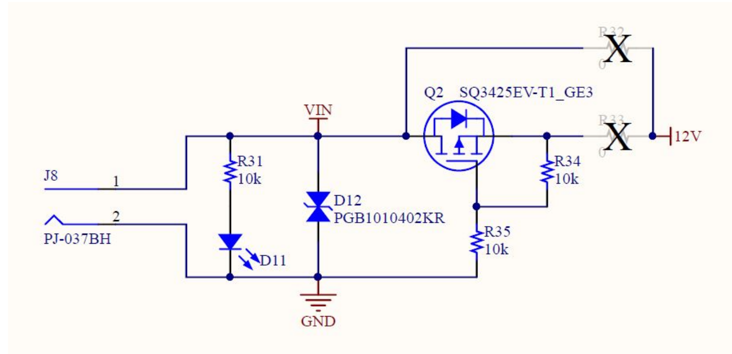


Figure 5.3: Schematic of the power connector circuitry

Step-down DC-DC Voltage Regulators

The two DC-DC step-down voltage regulators, U7 and U8, and their external components, are shown in schematic sheet 16 in Appendix A. One of them powers the 3V3 rail while the other powers the 1V2 rail. The same IC is used for both of the regulators, but some of their external components differ. Figure 5.4 shows the regulator, U8, that is used to power the 3V3 rail. The output voltage of the regulator is selected by choosing specific values for the resistors, R43, R44 and R45, that are connected to the FB pin. Equation 5.1 calculates the output voltage from the values of these resistors [31]. Furthermore, groups of low ESR decoupling capacitors, C67 to C71 and C63 to C66, are provided at the input and output of the regulator, respectively, as recommended by [31]. The capacitance values of the capacitors within these groups range from 0.047 μF to 220 μF so that efficient decoupling and stability can be provided at a range of different frequencies. The regulator also provides a

soft start functionality when connecting a 0.47 μF capacitor, C72, to the SS pin. This reduces the rate at which the output voltage ramps up when power is initially provided, which decreases surge currents. The functionality of the remaining pins are not required, and thus, those pins are left unconnected or connected directly to ground. A 0 Ω jumper resistor, R42, is added at the output of the voltage regulator. It is marked "DNP" so that it can be populated manually after verification of the output voltage of the regulator. Finally, the circuitry for the regulator that powers the 1V2 rail is the same except for the values of the resistors that are connected to the FB pin to set the output voltage.

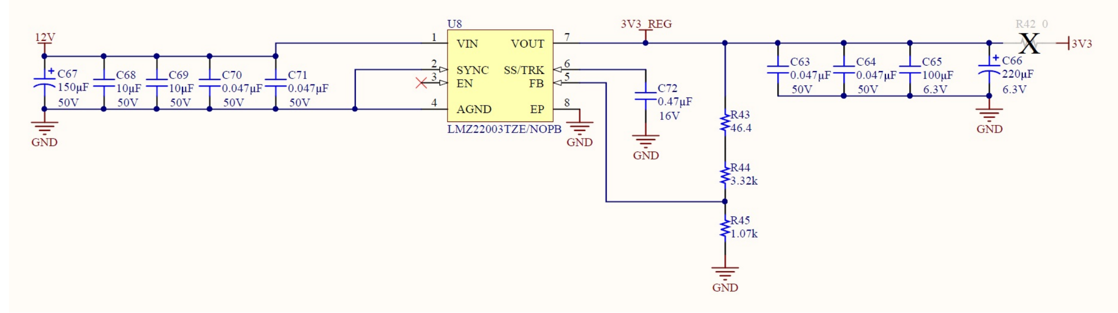


Figure 5.4: Schematic of the 3.3 V DC-DC voltage regulator

$$V_o = 0.796 \times \left(1 + \frac{R_a}{R_b}\right) \tag{5.1}$$

Where:
V_o = regulator output voltage
R_a = resistance between FB pin and VOUT pin
R_b = resistance between FB pin pin and ground

LDO Voltage Regulators

The two LDO voltage regulators, U5 and U6, and their external components, are shown in schematic sheet 15 in Appendix A. One of them powers the 1V8 rail while the other powers the 5V rail. The ICs that are used for these two regulators differ, but their principle of operation and the external components that they use are the same. Figure 5.5 shows the regulator, U6, that is used to power the 5V rail, along with its external components. The regulator provides a fixed 5 V voltage as output. To ensure good transient response and stable operation of the regulator, the 10 μF and 100 μF tantalum capacitors, C51 and C52, are provided at the regulator's inputs and outputs, respectively, as recommended by [32]. Also, a protection diode, D14, is provided between the IN and OUT pins of the regulator since the internal circuitry of the regulator is very sensitive to reverse currents. During cases when power will be removed from the test board, and the two terminals of the power supply connector are accidentally connected to each other, a reverse current will flow through the internal circuitry of the regulator. With a large output capacitance of 100 μF, the reverse current will last relatively long and may damage the regulator. The protection diodes will divert the reverse current away from the LDO, and thus, protect the LDO from damage. Also, a 0 Ω jumper resistor, R37, is added at the

output of the voltage regulator. It is marked “DNP” so that it can be populated manually after verification of the output voltage of the regulator. Finally, the external circuitry and principle of operation of U5 are the same except that 1.8 V is provided as output instead.

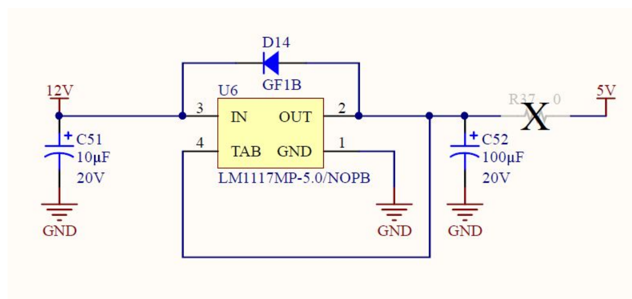


Figure 5.5: Schematic of the 5 V LDO voltage regulator

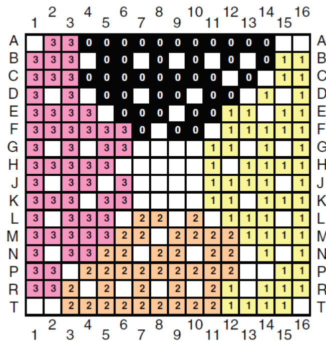
5.2.2 FPGA

Spartan-6 LX25 FPGA Overview

A Spartan-6 LX25 FPGA, U1, shown in schematic sheets two to seven in Appendix A, is used as the main component of the hardware design. It has 256 pins and uses a ball grid array (BGA) package. As shown in Figure 5.6, it has different pin types that are grouped into four banks. These banks can be powered by different supply voltages so that the pins within those banks can provide different functions. For this design, all banks are powered by 3.3 V. Furthermore, the FPGA pins are classified according to their functions. User IO pins are connected to the FPGA’s IO buffers which drive and receive signals to and from external components, respectively. They can be configured to use a variety of IO standards. Multifunction pins are similar to User IO pins, but they can also provide additional functionalities such as configuration. Also, some of them provide access to the internal clocking resources within the FPGA, such as the PLL component used by the clock generator in Section 4.3.1. The dedicated pins can only be used for specific functions, such as configuration, and cannot be configured to provide other functionalities. The remaining pins include power and ground pins which are used to power the FPGA’s internal logic and IO buffer resources. [33]

FPGA Power Supply

As shown in schematic sheet seven in Appendix A, the FPGA contains multiple supply pin groups. The first of these supply pin groups, VCCO_0, VCCO_1, VCCO_2 and VCCO_3, are used to provide power to the IO buffers of bank zero, one, two and three, respectively. The IO buffers support different IO standards depending on the voltage provided to their banks. Providing all of these supply pin groups with 3.3 V from the 3V3 power rail allows all IO standards required by the hardware design to be supported. The next supply pin group, VCCAUX, is used to provide power to the clocking and configuration resources of the FPGA, and is also powered by the 3V3 power rail. Finally, the VCCINT supply pin group is used to provide



IO Banks

User I/O Pins	Multi-Function Pins		Dedicated Pins	Other Pins
IO_LXXY_#	VREF	CCLK	PROGRAM_B_2	GND
	P_GCLK	CSI	TCK	VCCAUX
	N_GCLK	CSO	TDI	VCCINT
	D0 - D15	DIN	TDO	VCCO
	A0 - A25	DOUT_BUSY	TMS	NC
	FCS / FWE / FOE / HDC / LDC	HSWAPEN	DONE_2	
		INIT	SUSPEND	
	RDWR_B_VREF	M1, M0	CMPCS_B_2	
		AWAKE		

Pin Types

Figure 5.6: Spartan-6 LX25 FPGA IO banks and pin types [33]

power to the FPGA's internal logic resources, and is powered from the 1V2 power rail. [33]

5.2.3 FPGA Configuration

FPGA Configuration Overview

A VHDL design is implemented on an FPGA by converting it to configuration information, called a bitstream, and uploading that bitstream onto the FPGA. The FPGA of the test board can be configured with the bitstream using two methods. In the first method, the bitstream is manually uploaded to the FPGA using a Xilinx JTAG programmer. When power is removed from the test board, the configuration information will be lost, and the FPGA needs to be reconfigured. The second method allows automatic configuration of the FPGA using the Xilinx Platform Flash PROM, U2, shown in schematic sheet eight in Appendix A. Instead of uploading the bitstream to the FPGA, it is uploaded to the PROM which retains the bitstream when power is removed from the test board. During power-up, the FPGA automatically retrieves the bitstream from the PROM and configures itself with it.

JTAG Interface

Both the PROM and FPGA are accessed by the Xilinx JTAG programmer via a JTAG interface. Figure 5.7 summarises how the JTAG interface connects the JTAG connector, FPGA and PROM to each other. These components and their connections to the JTAG interface are shown in schematic sheets six, eight and 13 in Appendix A. The first signal of the JTAG interface is a source synchronous clock, TCK, which is provided to the FPGA and PROM so that the data provided to them can be sampled. The second JTAG signal, TMS, is also provided to both devices and is used to control the state of the JTAG interface. The JTAG data signals, TDO and TDI, of the three components are connected to each other to form a "chain". This "chain" allows data to be transferred from the JTAG connector to the FPGA, then from the FPGA to the PROM, and finally, from the PROM back to the connector. Thus, data can be written to and read from both the FPGA and PROM by a Xilinx JTAG programmer that is connected to the JTAG connector.

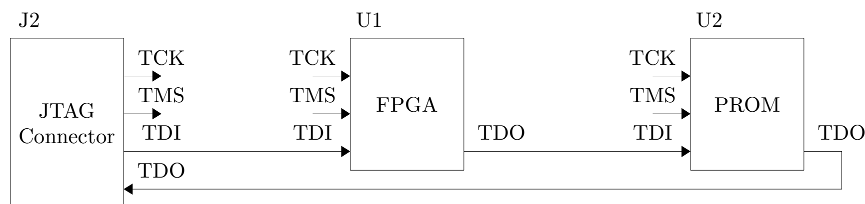


Figure 5.7: Block diagram of the JTAG interface

FPGA Configuration Circuitry

Figure 5.8 shows the FPGA's dedicated configuration pins along with external components. The first of the configuration pins, TDI, TDO, TCK and TMS, are used by the JTAG interface to upload the bitstream directly to the FPGA and to send instructions to the FPGA. The DONE_2 pin is used to indicate if the FPGA is

configured or not by providing a logic-high or logic-low voltage value as output, respectively. This pin is provided with a 330 Ω pull-up resistor, R1, as recommended by [34], and is connected to the gate of a MOSFET, Q1. When the FPGA is configured, the gate of Q1 will be positive with respect to the source, and thus, the transistor will be switched on causing current to flow through the resistor, R30, and the LED, D10. Thus, the LED will illuminate to indicate that the FPGA is configured. The PROGRAM_B_2 pin is used to initiate configuration of the FPGA from the bitstream stored in the PROM without having to perform a power cycle. This pin is provided with a 4.7 kΩ pull-up resistor, R28, as recommended by [34]. When the pin is momentarily pulled low, the configuration process will be initiated. It can either be pulled low by pressing the push-button, SW1, or from the PROM, via the FLASH_CF signal, when sending a specific instruction to the PROM via JTAG. The latter option requires the jumper resistor, R29, to be populated so that FLASH_CF can be connected to PROGRAM_B_2. This option is provided for debugging purposes. The functionality of the remaining pins are not required, and thus, those pins are left unconnected or are connected directly to ground.

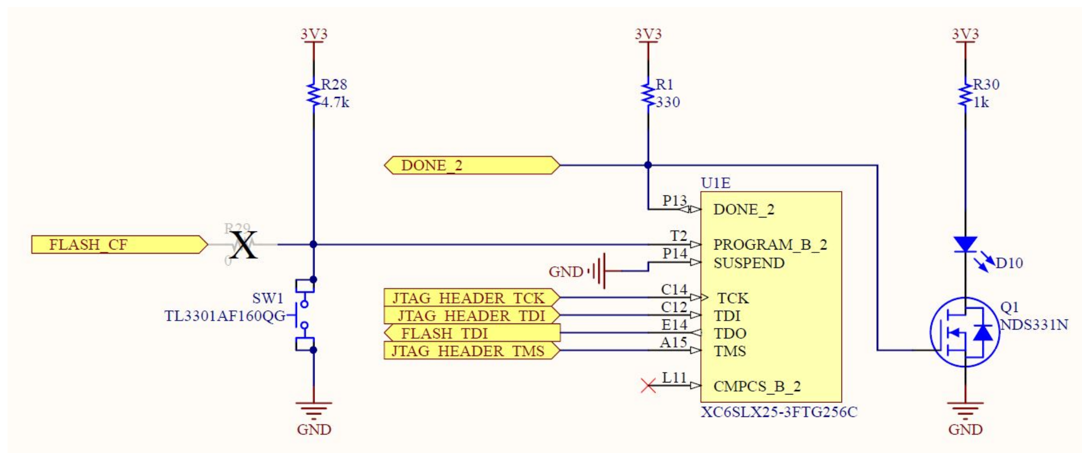


Figure 5.8: Schematic of the FPGA's configuration circuitry

PROM Configuration Circuitry

Figure 5.9 shows the PROM along with its external components. It has two power supply pin groups which are powered by the 1V8 and 3V3 power rails. Each of the power supply pins is provided with a 0.1 μF decoupling capacitor. The TDI, TDO, TCK and TMS pins are used by the JTAG interface to transfer the bitstream to the PROM and to send instructions to the PROM. The jumper resistors, R6 and R7, are provided for debugging purposes so that the PROM can be disconnected from the JTAG chain and bypassed with jumper wires if required. The CLK pin is provided with a clock signal from the FPGA during the configuration process. When this clock signal is received by the PROM, it will provide the configuration data back to the FPGA serially via the D0 pin. The Thévenin termination resistors at the CLK pin, R2 and R8, and the series termination resistor at the D0 pin, R5, are recommended by [34]. The CE pin is used to put the PROM in a low power mode if a logic-high voltage value is applied to it. It is connected to the DONE_2 pin of the FPGA which provides a logic-high voltage value after configuration, and

thus, puts the PROM in a low power mode after the FPGA is configured. The CF pin is used to send a configuration pulse to the PROGRAM_B_2 pin of the FPGA, when a specific instruction is sent via JTAG, so that the configuration process can be initiated without having to perform a power cycle. The OE_RESET pin is momentarily pulled low by the FPGA before configuration to reset the PROM to its initial state. Both the CF and OE_RESET pins are provided with 4.7 kΩ pull-up resistors, R3 and R4, as instructed by [34]. Finally, the functionality of the remaining pins are not required, and thus, those pins are left unconnected or are connected directly to ground.

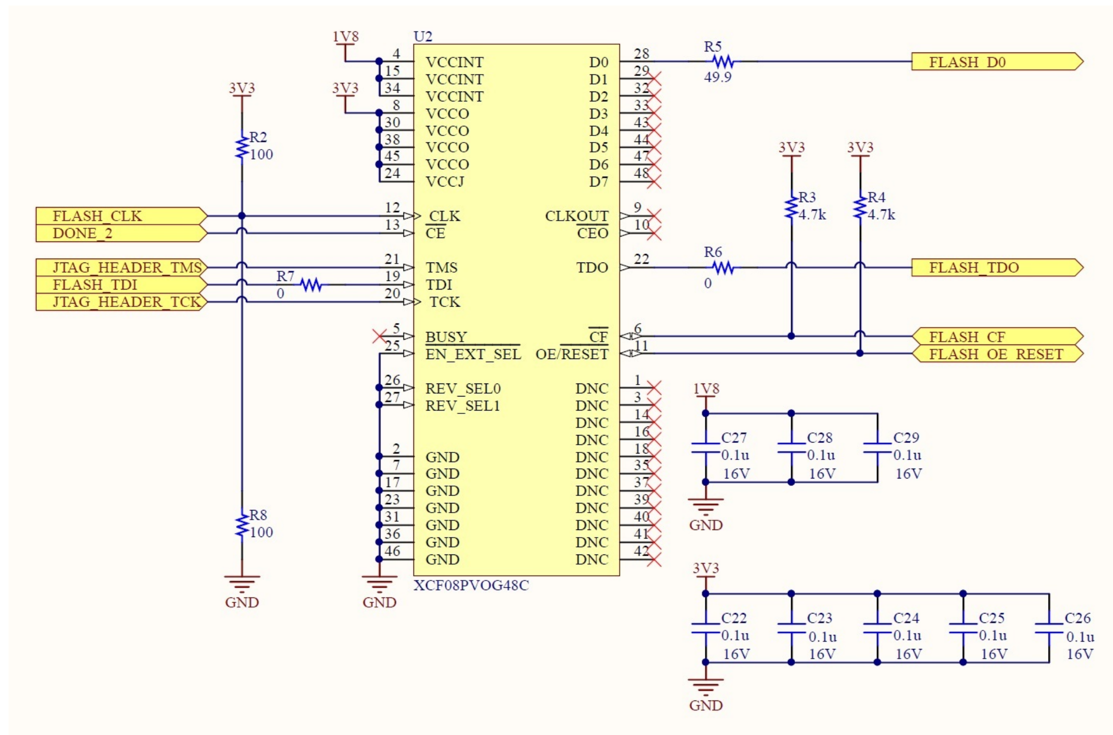


Figure 5.9: Schematic of the PROM's configuration circuitry

5.2.4 HDMI Interface

HDMI Signals

Figure 5.10 shows the four interfaces that were implemented to constitute the HDMI interface. The signals of these interfaces are provided to the HDMI connector, J1, from the FPGA, via the HDMI port protection device, U3. The first interface consists of four differential TMDS signal pairs. Three of these differential pairs are used to transfer HDMI video frame data while the other pair provides the pixel clock to the display monitor. These signals are used by the video transmitter component, discussed in Section 4.3.2, so that it can provide MicroGE's video output to an HDMI display monitor. The second interface, DDC, is used by the EDID reader, discussed in Section 4.3.3, to read the EDID/E-EDID ROM data within a display monitor using the SCL and SDA signals of the I^2C standard. The third interface is the hot plug detect (HPD) signal which indicates when the other end of the HDMI cable that is plugged into the test board is plugged into a display monitor. The

last interface is Consumer Electronics Control (CEC), which is used for high-level control of HDMI devices. CEC is not used in this study, but it was still implemented as part of the HDMI interface to provide additional functionalities to the test board.

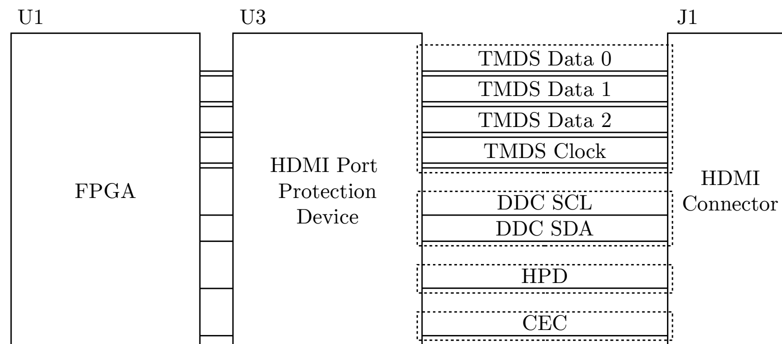


Figure 5.10: Summary of the signals of the HDMI interface

HDMI Connector

The HDMI connector, J1, shown in Figure 5.11, is used to connect the test board to an HDMI display monitor, via an HDMI cable, so that MicroGE's video output can be inspected. The chassis of the HDMI connector is connected to ground via a resistor, R19, and capacitor, C36, for ESD protection purposes. If there is an electrical potential difference between the chassis and ground, it will cause a slow discharge of current through the resistor, and thus, eliminate the potential difference. The capacitor, on the other hand, provides a low impedance path for sudden ESDs. Pin 14 of the connector is connected to ground via a 75 Ω resistor as recommended by [35]. Each of the remaining pins is either connected to an HDMI signal or directly to ground.

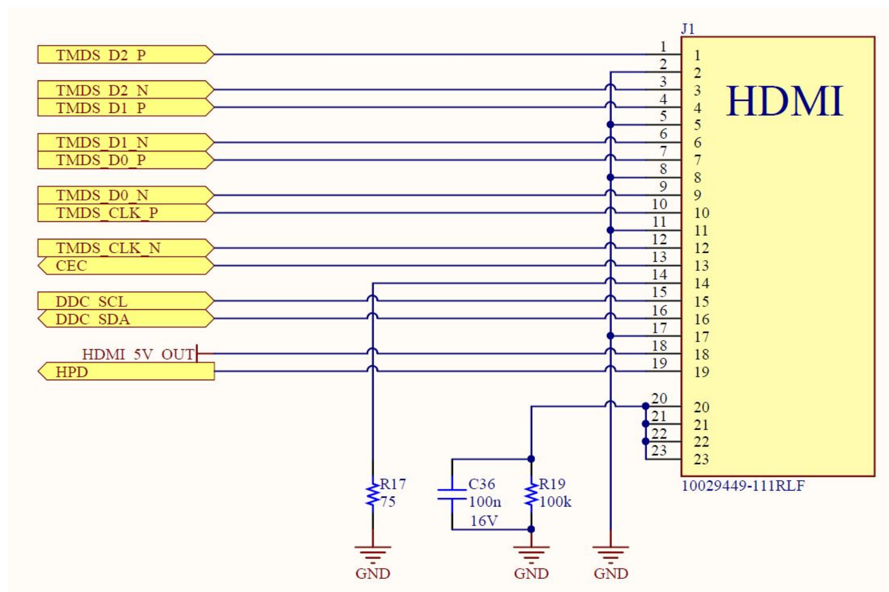


Figure 5.11: Schematic of the HDMI connector and external components

HDMI Port Protection Device

Figure 5.12 shows the HDMI port protection device along with passive components and HDMI signal traces. This device protects the FPGA from ESDs that may occur on any of the HDMI signal traces and also provides voltage level translation to some of the signals. Each of the TMDS signal traces is connected to two pins of the device as they are routed underneath. Each of the remaining HDMI signal traces is separated so that one part of the trace can be connected to one pin while the other part can be connected to another pin. The device then performs voltage level translation between those two pins. The voltage levels of the DDC and HPD signals, and the voltage levels of the CEC signal are translated to 5 V and 3.3 V levels, respectively, to make those signals compatible with the HDMI standard [17]. External pull-up and pull-down resistors, R9, R10, R11, R12, R14, R15, R16 and R18, are provided to some of the pins of the device to enable its voltage level translation capabilities. The values of these resistors are recommended by [36]. The Schottky diode, D1, is provided to protect the test board against backdrive current in case a 5 V signal is accidentally applied to the CEC pin of the HDMI connector.

Furthermore, the HDMI port protection device has two supply pins, 5V_SUPPLY and LV_SUPPLY. LV_SUPPLY should be powered by the supply rail that is powering the signals whose voltage levels should be translated, the 3V3 rail. The 5V_SUPPLY pin is powered with 5 V from the 5V rail. A resistor is provided between the 3V3 rail and the LV_SUPPLY pin to reduce current consumption of the device, as allowed by [36]. The device also provides a current-limited 5 V supply voltage to the HDMI monitor, via HDMI_5V_OUT, as required by the HDMI standard. Finally, a 0.1 μF ceramic decoupling capacitor is provided to each supply pin and also to the ESD_BYP pin, as recommended by [36].

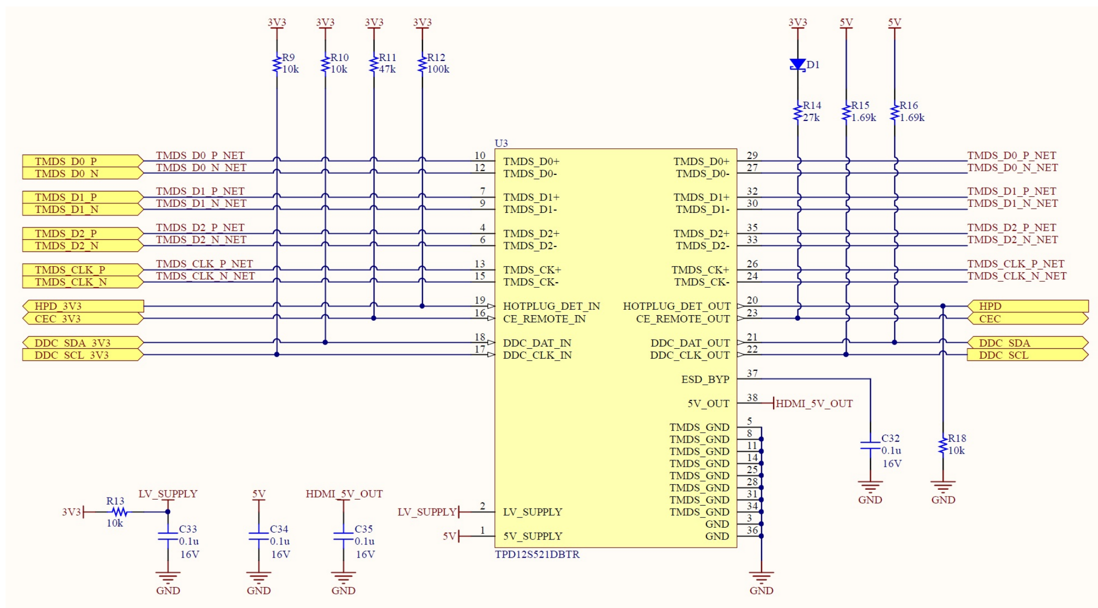


Figure 5.12: Schematic of the HDMI port protection device

5.2.5 Reference Clock

Figure 5.13 shows the reference clock generation circuitry. This circuitry provides the reference clock to the FPGA so that the clock generator component and the video transmitter component, discussed in Section 4.3.1 and Section 4.3.2, respectively, can derive their output clocks. A 50 MHz oscillator, Y1, was chosen to provide the reference clock since a 50 MHz frequency allows all of the output clock frequencies of these two components to be derived. The reference clock output pin of the oscillator, OUT, is connected directly to one of the FPGA's User IO pins that has access to the FPGA's internal clocking resources, such as PLLs. The oscillator's supply pin, VDD, is powered from the 3V3 rail and is provided with a 10 nF ceramic capacitor for power supply decoupling purposes. Also, the oscillator can be placed in a standby mode by providing a logic-low voltage value to its standby pin, STBY, but this functionality was not required, and thus, the pin was connected to the 3V3 power rail.

Also, another oscillator, Y2, along with a decoupling capacitor, was added to the schematic design, as shown in schematic sheet nine in Appendix A, but they were both marked DNP. The footprints of these two components can be used to add another reference clock to the test board. Having another reference clock, with a frequency different from 50 MHz, may allow the video transmitter component to support a wider range of pixel clock frequencies, and thus, support a wider range of video modes. This allows additional functionalities to be added to the test board.

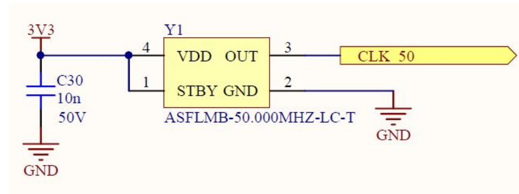


Figure 5.13: Schematic of the reference clock generation circuitry

5.2.6 Debugging Circuitry and SPI Interface

Schematic sheet 13 in Appendix A shows the debug circuitry and the SPI interface connector of the test board. First of the debug circuitry is a set of eight LEDs, D2 to D9, which is each connected to a User IO pin of the FPGA via a 1 kΩ resistor. These LEDs can be used to provide status output of the FPGA design during testing and debugging. Second of the debug circuitry is a set of IDC connectors, J3 to J6, whose pins are connected to some of the User IO pins of the FPGA. These connectors, all marked DNP, were added to the schematic design for in case additional control of the FPGA would be required during testing and debugging. The test point connected to ground, TP1, provides a convenient location for connection of the ground terminal of electronic test equipment. Lastly, the IDC connector, J7, is provided so that the Arduino Due can be connected to the test board via an IDC cable. The pins of this IDC connector are connected to some of the User IO pins of the FPGA so that the Arduino Due can send SPI instructions to the FPGA when the FPGA is configured with the FPGA design containing MicroGE.

5.2.7 External Memory

Figure 5.14 shows the external memory component, U4, along with decoupling capacitors. External memory was added to the hardware design to provide additional memory to store frame buffer data for in case MicroGE's "frame-buffer-less" architecture could not be realised. The memory component is a 256 Mb SDRAM IC with a maximum clock frequency of 166 MHz. This component provides a large enough memory capacity and bandwidth to store and transfer the frame buffer data, respectively, that MicroGE would have required. The SDRAM pins include address, data, clock and control pins which are all connected to some of the FPGA's User IO pins.

Furthermore, the SDRAM has two power supply pin groups, VDD and VDDQ, which are used to power the internal logic and IO buffers of the SDRAM, respectively. VDD is powered directly from the 3V3 power rail while VDDQ is powered from the 3V3 power rail via a ferrite bead, L1. The ferrite bead reduces the amount of noise that couples from VDDQ onto the 3V3 rail. This noise is due to toggling of signals that are provided from the IO buffers of the SDRAM to the FPGA. The ferrite bead is marked DNP so that it can be populated manually after the voltage of the 3V3 power rail is verified. Finally, a 0.1 μF ceramic capacitor is provided at each supply pin for power supply decoupling purposes.

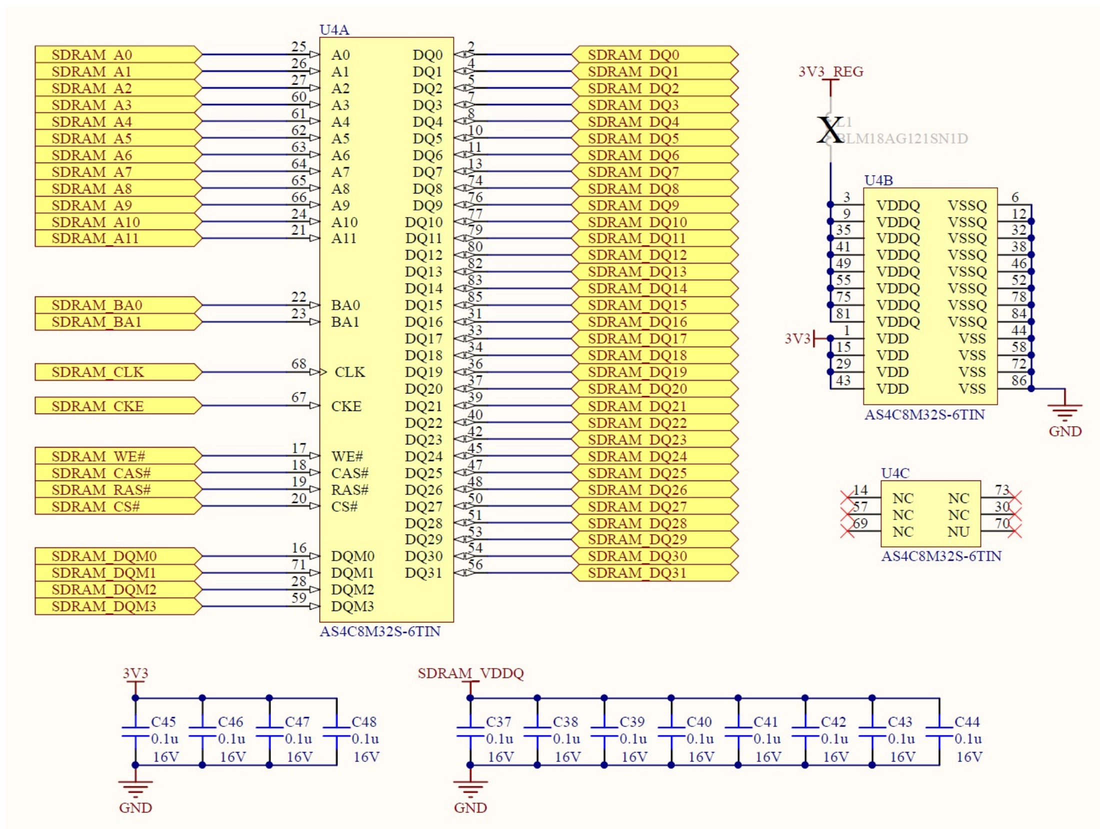


Figure 5.14: Schematic of the external memory circuitry

5.3 PCB Layout

5.3.1 PCB Layout Process Overview

The PCB layout of the test board was performed with Altium Designer. Before starting with the layout, the placement of all the main components was planned. After this planning, it was clear what the dimensions of the PCB had to be and how many copper layers were required for power planes and routing. Then, stackup information, that suited the needs of the envisioned layout, was acquired from Trax Interconnect. After this information was acquired, the PCB layout was implemented in Altium Designer. First, the footprints for all of the components were created. Then, these footprints were placed as initially planned. The copper planes of the PCB were then allocated to specific purposes so that power planes could be placed and routing could be performed. Finally, some thermal calculations were performed to ensure that the test board would not overheat during operation.

5.3.2 Component Placement Planning

The purpose of this process was to plan how all of the main components needed to be placed on the PCB so that routing among them could be as simple as possible. All of the main components were placed on the top side of the PCB. The block diagram in Figure 5.15 shows the areas that were reserved for all of the main components on the top side of the PCB. From this placement, it could be determined that the size of the PCB had to be 10 cm $\times$ 10 cm and that the PCB's layer count had to be six.

The FPGA was placed in the middle of the test board with the other components surrounding the FPGA. Devices that interface directly with the FPGA, such as the oscillators, HDMI port protection device, PROM, and SDRAM, were placed as close as possible to the FPGA pins that they had to be routed to and were oriented to simplify routing. Connectors were placed at the board edge to make the connection of cables to them more convenient. Each voltage regulator was placed close to the supply pins of the components to which it had to provide with power. Also, sufficient PCB area was provided around the DC-DC voltage regulators for placement of large decoupling capacitors. Finally, the placement of fiducials, standoff holes, debug LEDs, a push-button, and a layer key was also considered.

5.3.3 PCB Stackup

The PCB stackup that was used to fabricate the PCB of the test board is shown in Appendix B. Trax Interconnect provided this stackup and fabricated the PCB according to it. It is a six-layer stackup that consists of two power, two ground and two routing copper layers. The thickness of the copper layers was chosen as 35 micrometres since it is thick enough to carry all supply currents without causing significant temperature rises. The NY2150 dielectric material was used for the prepregs and cores of the stackup. This material was suggested by Trax Interconnect since it is inexpensive while it still allows sufficient impedance matching to be achieved for the high-speed TMDS signal traces. The thickness of the cores and prepregs were chosen to allow all signal traces to be matched to their required characteristic impedances with trace widths smaller than 0.25 mm. A width smaller than 0.25 mm is required since most of the signal traces needed to be routed between the BGA

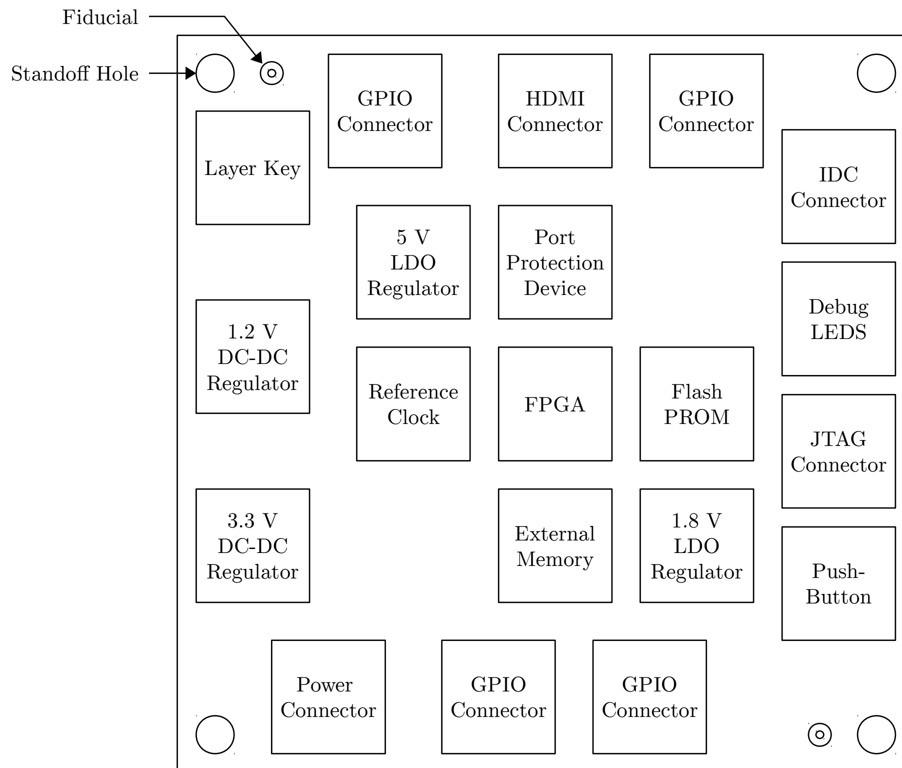


Figure 5.15: Planning of the placement of all the main components

pads of the Spartan-6 LX25 FPGA's footprint. Finally, a single through-hole drill pair was provided for vias.

5.3.4 Footprint Creation

The footprints of all the components were created and verified with Altium Designer. Before a footprint was created, the dimensions of its pads and solder mask clearances were determined. Most of the components' datasheets provided guidelines to determine this information. When a component's datasheet did not provide any guidelines, Altium Designer's IPC Compliant Footprint Wizard was used to calculate the dimensions of the pads and solder mask clearances. Once the required information was determined, a footprint was created in which the pads and solder mask clearances were defined accordingly. Each footprint was also provided with a silkscreen outline which encloses the area where the component should be mounted. The footprints of components that had to be mounted in a specific orientation were also provided with an indication of that orientation. These indications include the addition of a polarity dot in one of the corners of the footprint and modification of the silkscreen outline of the footprint to clarify the orientation. Once the creation of a footprint was complete, the footprint was inspected in 3D, along with a 3D model of the component that should be mounted on it, to verify that the footprint was correct. Figure 5.16 a) and b) shows a footprint that was created and how that footprint was inspected, respectively.

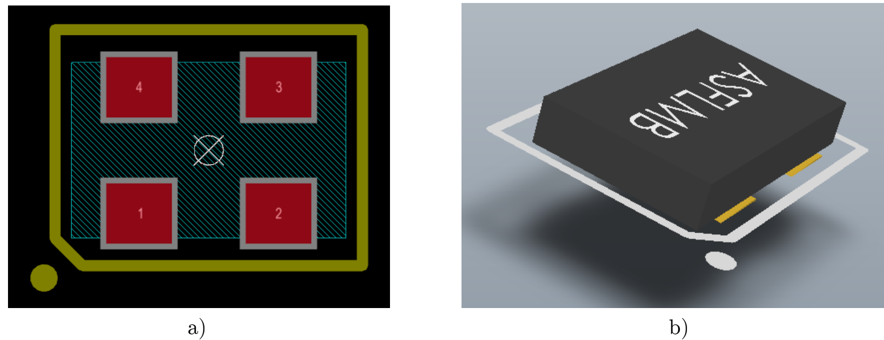


Figure 5.16: Screenshots of a footprint that was created, a), and how that footprint was inspected, b)

5.3.5 Component Placement

After the footprints were created, each one of them was placed on the PCB layout. The main components, whose placement was planned in Section 5.3.2, were placed first. They were all placed on the top PCB layer, as initially planned, within the areas that were reserved for them. The decoupling capacitors were placed next. They were placed as close as possible to the supply pins to which they had to be connected. During placement of capacitor groups, the low ESR, low ESL, ceramic capacitors were placed the closest to those supply pins while the bulk capacitors were placed directly after. Also, since some of the supply pins of the FPGA are in the centre of the BGA package, some of the decoupling capacitors were placed on the other side of the PCB, directly under the FPGA, so that those capacitors could be as close as possible to those supply pins. Those capacitors were then connected to those supply pins with vias. After all of the decoupling capacitors were placed, the resistors that had to be close to specific ICs, such as the termination and bias resistors, were placed. Then, the ESD protection components were placed. They were placed as close as possible to the connectors from which they should protect ESDs from so that if an ESD event occurs, the discharged current can be diverted from the rest of the circuit that should be protected. The remaining components did not have to be placed close to any specific location on the PCB and were simply placed wherever there was enough PCB area.

5.3.6 PCB Layer Allocation

The six copper layers of the PCB were dedicated to specific purposes. The top and bottom layers were used for all of the signal trace routing. Also, the areas of these layers that were not used by routing and were not occupied by component footprints were provided with large copper pours for ground and for the power rails. Layers two and five were used as dedicated ground planes for the signal traces on the top and bottom layers to reference to, respectively. Layers three and four were used as dedicated power planes. The entire area of layer three was provided to the 3V3 power rail since most components needed to be powered from 3V3. The area of layer four was split between the 12V, 1V2, 5V and 1V8 power rails. A part of layer four was also used to connect the VDDQ supply pin group of the SDRAM, U4, to the

3V3 power rail via the ferrite bead, L1. Having dedicated power planes on layers three and four is beneficial since the surface area of the copper plane between each voltage regulator and the components that are powered by that voltage regulator is very large.

5.3.7 Routing

Routing Overview

Routing was performed on the top and bottom layers of the PCB, as planned in Section 5.3.6. Some of the routed signal traces required special considerations. The high-speed TMDS signal traces required length and impedance matching to ensure sufficient signal integrity of the signals that would be transmitted across them. The signal traces of the external memory component were also length matched, but since the signals that would be transmitted across those traces have low frequencies, 166 MHz, it was not required. This length matching was performed so that experience could be gained with Altium Designer's length matching tools. The remaining traces did not require any special considerations regarding length and impedance matching since no high-speed signals would be transmitted across them. For consistency, all of them were routed as 50 Ω traces with 0.2 mm trace widths. Also, since the FPGA has a BGA package, a fanout technique was used, as shown in Figure 5.17, to provide access to the pins of the FPGA. Finally, when the routing process was complete, the PCB layout was simulated in HyperLynx V8.0 to confirm that there would not be any significant crosstalk among any of the signal traces.

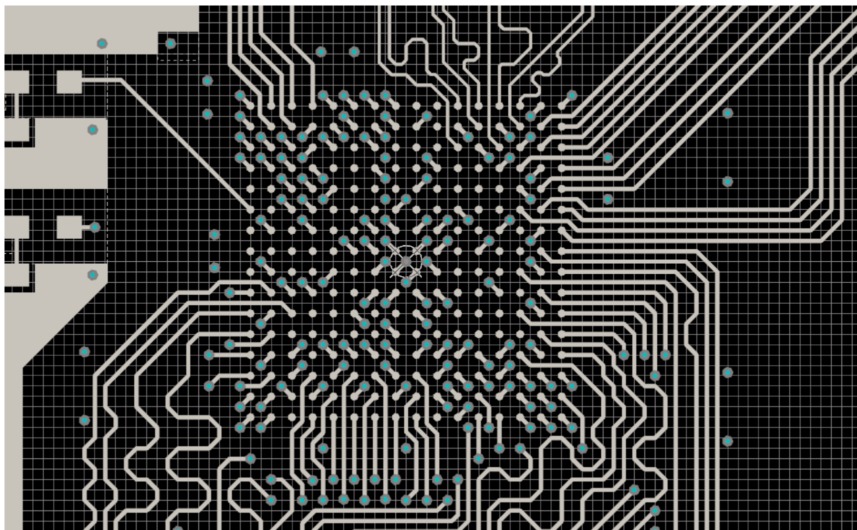


Figure 5.17: Screenshot of the fanout technique that was used to provide access to the FPGA's pins

Impedance Matching

The differential and single-ended characteristic impedances of the high-speed TMDS signal traces were matched to specific values to ensure sufficient signal integrity of the signals that would be transmitted across those traces. According to [17], each TMDS signal trace pair needs to have a differential characteristic impedance of 100

Ω. The equations provided in Section 2.4.2 were used to design the TMDS signal traces to have the required characteristic impedances. The thickness of the traces, 0.038 mm, the distance between the traces and the ground plane, 0.118 mm, and the dielectric constant of the material between the traces and the ground plane, 4.2, were chosen in Section 5.3.3. Thus, the required characteristic impedances were obtained by choosing the width of the TMDS signal traces and by choosing the distance between the two traces of each TMDS signal trace pair. Equation 2.5 was used to calculate the width of the TMDS signal traces as approximately 0.15 mm to obtain a single-ended characteristic impedance of 55 Ω. Equation 2.6 was then used to calculate the distance between the two traces of each TMDS signal trace pair as approximately 0.35 mm to obtain a differential characteristic impedance of 100 Ω.

Length Matching

The high-speed TMDS signal traces were length matched to minimise the intra-pair and inter-pair timing skew of the signals that would be transmitted across them. [17] allows the intra-pair skew of the TMDS signals transmitted from the FPGA to the HDMI source connector to be no longer than 0.15 of the current data bit period of the TMDS interface. Similarly, [17] allows the inter-pair skew of these signals to be no longer than twice the current data bit period of the TMDS interface. The shortest data bit period of the video modes that the video transmitter component, discussed in Section 4.3.2, supports is 1.17 ns. Thus, the maximum allowable intra-pair and inter-pair skews are 175.5 ps and 2.34 ns, respectively.

The equations provided in Section 2.4.1 were used to calculate the corresponding maximum allowable length differences among the TMDS signal traces. The width of the traces, 0.15 mm, the dielectric constant of the material between the traces and the ground plane, 4.2, and the distance between the traces and the ground plane, 0.118 mm, were determined in the previous sections. Equation 2.4 can be used to calculate, from these three values, the effective dielectric constant of each TMDS signal trace as 3.1. Equation 2.3 can then be used to calculate, from the effective dielectric constant, the propagation speed of the TMDS signals as 170.27 $\times 10^6$ m/s. Finally, the calculated propagation speed can then be used to calculate the maximum allowable length difference between two traces of each TMDS signal trace pair as 2.99 cm and to calculate the maximum allowable length differences among the four TMDS signal trace pairs as 39.84 cm. The length differences among the TMDS signal traces were kept far below these calculated limits to increase the maximum allowable skew of the TMDS signals before they are provided from the FPGA. This reduces the timing constraints of those signals within the FPGA.

Figure 5.18 shows how the length differences among the TMDS signal traces were minimised. As two traces of a TMDS signal trace pair are routed from the pads of the FPGA's footprint, a large amount of length difference between those two traces is already introduced. This is because the pads that those two traces are routed from are not located next to each other. The length difference between those two traces is minimised immediately after they leave their pads by means of a serpentine routing technique. After the length difference between the two traces of each TMDS signal trace pair is minimised, the length differences among the four TMDS signal trace pairs are also minimised by means of a similar serpentine routing technique.

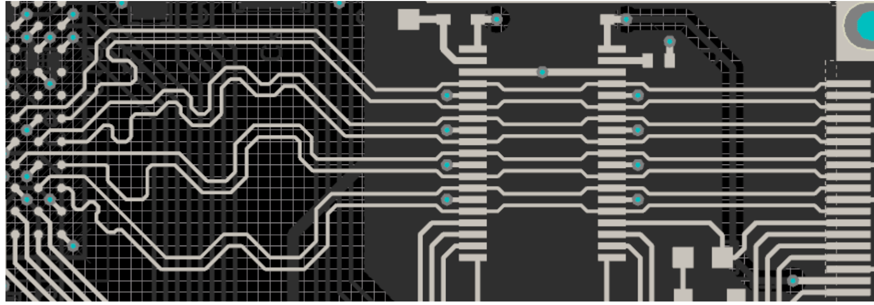


Figure 5.18: Screenshot of how length matching of the high-speed TMDS signal traces was performed

Thermal Verification

The tools offered by [37] were used to confirm that the temperature rises of all the conductors on the test board would stay below 10 °C. The calculated temperature rises of most conductors were far below this limit since the maximum current that would flow through those conductors are very small. On the other hand, the maximum current that would flow through the 3V3 and 1V2 power rails are very large. However, since these power rails were provided with dedicated power planes that have large surface areas, as discussed in Section 5.3.6, the calculated temperature rises of these power rails were still below 10 °C.

5.3.8 Final PCB Design

Screenshots of the top side and bottom side of the PCB layout are shown in Figure 5.19 and Figure 5.20, respectively.

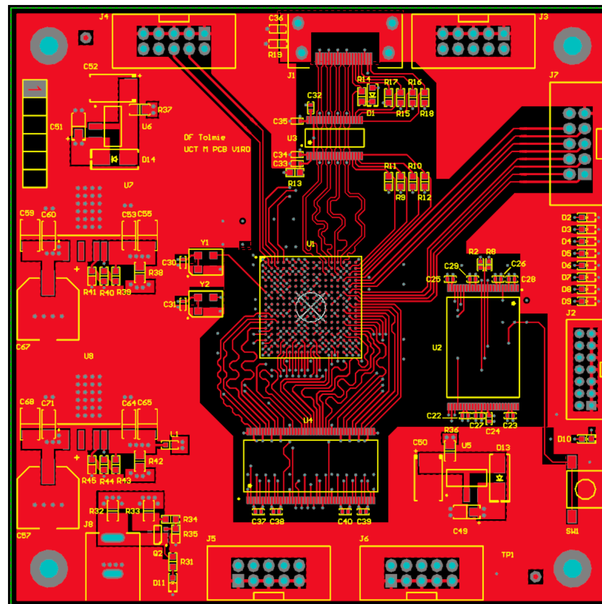


Figure 5.19: Screenshot of the top side of the PCB layout

Figure 5.20: Screenshot of the bottom side of the PCB layout

Simulated 3D images of the top side and bottom side of the test board, rendered by Altium Designer, are shown in Figure 5.21 and Figure 5.22, respectively.

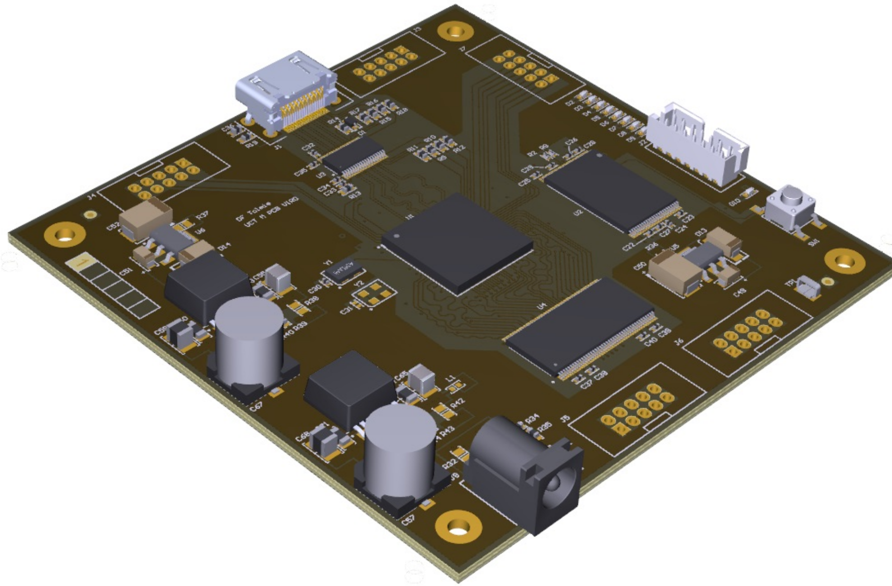


Figure 5.21: The top side of a 3D rendered image of the test board

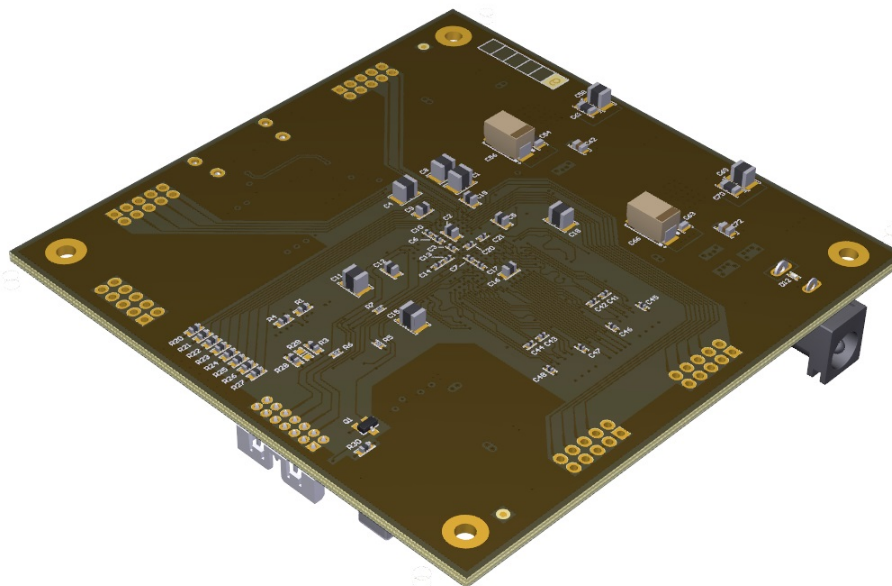


Figure 5.22: The bottom side of a 3D rendered image of the test board

5.4 Fabrication and Assembly

5.4.1 PCB Fabrication

Trax Interconnect manufactured the PCB of the test board using various design files exported from Altium Designer. Before starting with the manufacturing process, some preparations and verifications were required. First, a PCB stackup was requested from Trax. This was done before beginning with the PCB layout process since many aspects of the layout process, such as impedance matching, depend on the stackup. The requested stackup, included in Appendix B, was then used to complete the PCB layout process. Before proceeding with the PCB manufacturing process, it was confirmed that it would be possible to assemble all components on the PCB. A 3D step file of a simulated version of the assembled PCB was exported from Altium Designer and then sent to Rob Steltman, one of the owners of Barracuda Holdings. He verified that assembly could be done using their equipment. Once the step file was verified, the PCB's Gerber and NC drill files were exported from Altium Designer and sent to Trax for verification. These files contain information about the various layers of the PCB to fabricate and the holes to drill in the PCB, respectively. Finally, once these design files were verified, the PCB was manufactured and delivered to UCT.

5.4.2 PCB Assembly

After the PCB was fabricated, some preparations were made for completion of the assembly process. Barracuda Holdings was used for the assembly since Rob Steltman already verified that the assembly could be done by their equipment. Before assembly begun, Rob Steltman requested the PCB's assembly drawings and bill of materials (BOM) for verification purposes. The BOM, included in Appendix B, lists all of the components to assemble on the PCB along with their reference designators. The assembly drawing, also included in Appendix B, indicates where each component should be mounted by means of its reference designator. It also provides the PCB dimensions which are required to prepare the assembly equipment. After the assembly files were verified, the components of the test board were ordered from Digi-Key and delivered to UCT. The components were then inspected to verify that their part numbers and quantities correspond with the information in the BOM. Once this was verified, the components and PCB were sent to Barracuda for assembly. A week later, assembly was complete, and the assembled PCB, the test board, was ready for testing.

5.4.3 Assembled PCB

Photos of the top side and bottom side of the assembled PCB, the test board, are shown in Figure 5.23 and Figure 5.24, respectively.

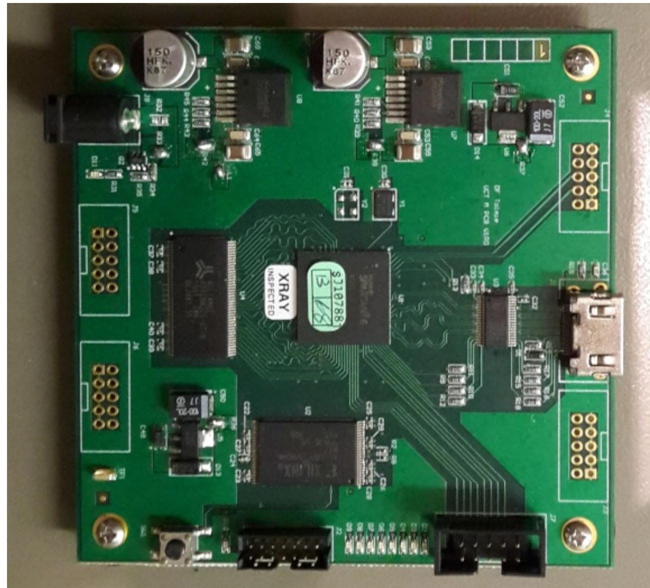


Figure 5.23: Photo of the top side of the test board

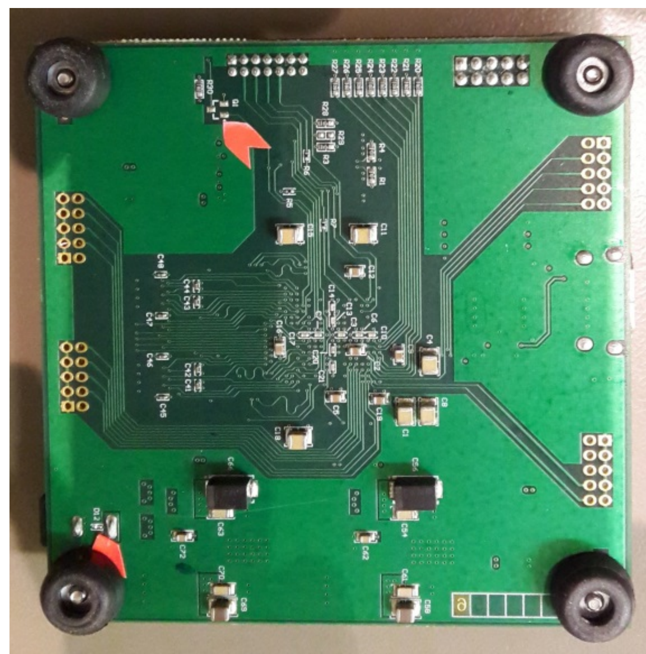


Figure 5.24: Photo of the bottom side of the test board

5.5 Hardware Verification

The PCB assembly, power supplies and functionality of the test board were verified using the bring-up plan included in Appendix B. This plan consists of a series of steps, each performed to verify different aspects of the test board. These steps are divided into three parts. The first part, visual inspection, was performed to confirm that the test board was assembled correctly. The second part, power supply verification, was performed to verify that the voltages of the power rails were within tolerance. The last part, functionality verification, was performed to verify the functionality of the test board.

The first part of the bring-up plan was performed to identify any possible assembly faults that could have caused the test board to get damaged during power-up. First, it was confirmed that the test board was clean and that there were no assembly residues. Then, it was confirmed that all components that should have been assembled during the PCB assembly process were mounted on the correct footprints and also mounted in the correct orientations. In contrast, components like the power jumper resistors at the output of the voltage regulators should not have been assembled during the PCB assembly process. They were soldered to their footprints manually, during part two of the bring-up plan, after the output voltages of those voltage regulators were verified. Finally, the pins of each component were inspected to confirm that they were soldered to their pads properly and to confirm that there were no solder bridges between these pins which may have resulted in short circuits.

The second part of the bring-up plan was used to verify that the voltages at the output of the voltage regulators were within tolerance. When power was provided to the test board for the first time, the power jumper resistors were not populated so that the output of the voltage regulators could be disconnected from the rest of the test board circuitry. This allowed the output voltages of the voltage regulators to be measured and verified before connecting the output of the regulators to the rest of the test board circuitry. Initially, two of the measured voltages did not correspond with the expected voltages. This was because two of the voltage regulators were swapped during the assembly process, and thus, were assembled onto the incorrect footprints. This mistake was not found during the visual inspection part of the bring-up plan since the packages of these two voltage regulators look similar. After these voltage regulators were swapped back to their correct footprints manually, all of the regulators' output voltages were within tolerance. Then, the power jumper resistors were populated so that power could be provided to the rest of the test board.

The last part of the bring-up plan was performed to verify the functionality of the test board. The functionalities of different parts of the test board were verified separately by uploading four different "test bitstreams" to the FPGA and evaluating the behaviour of the test board. Before a bitstream could be uploaded to the FPGA, the JTAG interface between the PC, containing the bitstreams, and the FPGA had to be established. Once this interface was established, it was confirmed that most aspects of the FPGA were functional. Then, each of the four test bitstreams was uploaded to the FPGA. The first bitstream was used to verify that the reference clock and debug LEDs work by toggling the debug LEDs every second based on timing derived from the reference clock. The second bitstream was used to verify that the DDC interface works by reading a monitor's EDID/E-EDID data and verifying

that data. The third bitstream was used to verify that the SPI interface works by transmitting data across the MOSI output and reading the data at the MISO input. The MOSI and MISO pins were connected to each other so that the transmitted and received data could be compared with each other. The last bitstream was used to verify that the TMDS interface works by transmitting a test pattern to an HDMI monitor. After these tests were performed, the functionality of the flash PROM component was verified. A bitstream was converted to the required format and then uploaded to the PROM via the JTAG interface. Then, it was confirmed that the FPGA could be configured from the PROM after power-up and after pressing the reset button, SW1.

Chapter 6

Verification and Results

6.1 Verification and Results Overview

To confirm that MicroGE was developed according to the specifications listed in Section 1.4, its behaviour had to be verified, and its resource utilisation and performance results had to be determined. First, MicroGE's behaviour was simulated using a VHDL test bench, as discussed in Section 6.2. After MicroGE's behaviour was successfully verified in simulation, the real behaviour of MicroGE, the behaviour of MicroGE when it was loaded onto the FPGA of the test board, was verified on a test system, as discussed in Section 6.3. After MicroGE's behaviour was verified on the test system, this test system was used to demonstrate MicroGE's rendering capabilities, as discussed in Section 6.4. Finally, MicroGE's resource utilisation and performance results were determined, as discussed in Section 6.5 and Section 6.6, respectively.

6.2 Test Bench Firmware Verification

6.2.1 VHDL Test Bench

A VHDL test bench was created, which models the system described in Figure 4.1, so that the behaviour of MicroGE could be verified in simulation. A block diagram of the main components of this test bench is shown in Figure 6.1. The device under test (DUT) component contains the VHDL design that was developed in this study. As discussed in Section 4.1, this VHDL design contains MicroGE, the clock generator, the EDID reader and the video transmitter. The MCU model, which models the behaviour of the Arduino Due, reads a list of instructions from a file and then produces SPI signals to the DUT, which contains MicroGE, so that MicroGE can be controlled. These simulated SPI signals correspond with the actual SPI signals that would be provided from the Arduino Due to the test board, as discussed in Section 6.3.1. The HDMI monitor model, which models the behaviour of an HDMI display monitor, receives the TMDS signals from the video transmitter component within the DUT, decodes the incoming video frame data and writes the video frame data to a file. This model recovers the video frame data in a similar manner that an actual HDMI display monitor would. The test bench also includes code that provides a 50 MHz reference clock to the DUT, which models the 50 MHz oscillator on the test board, so that all clocks required by the components within

the DUT can be derived. Additionally, the behaviours of the MCU model, HDMI monitor model and the reference clock generation code were verified using separate test benches before they were added to the main test bench. The main test bench was then used to verify the individual behaviours of the lower-level units within the DUT and to verify the behaviour of all the lower-level units in collaboration.

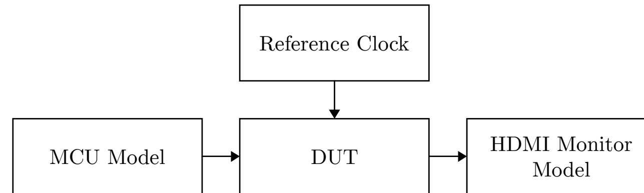


Figure 6.1: Block diagram of the test bench that was used to verify the behaviour of the VHDL design

6.2.2 Test Bench Simulator

The ISE Simulator (ISim) was used to simulate the test bench. ISim provides utilities that can be used to inspect the behaviour of the test bench during simulation. Among these utilities is a waveform editor which displays the values of signals as the test bench is simulated. The utilities also include a memory editor which displays the internal contents of memory components as the test bench is simulated. Additionally, ISim also allows data to be read from files to use as stimuli in the test bench, and it allows values of signals within the test bench to be written to files.

6.2.3 Verification Methods Overview

Three verification methods were used to verify the behaviour of the DUT. The first of these methods, discussed in Section 6.2.4, made use of ISim's waveform editor to verify the internal signal behaviour of the DUT. The second method, discussed in Section 6.2.5, made use of ISim's memory editor to verify that data was transferred to the DUT's memory components correctly. These two verification methods required manual visual inspection of the simulated results provided from ISim. These inspections were time-consuming, and thus, could not be used for detailed verification of the DUT's behaviour. The third method, discussed in Section 6.2.6, not only made use of ISim's tools but also used Python scripts to perform automated verification of the DUT's behaviour. This method was much less time-consuming, and thus, allowed more detailed verification of the DUT's behaviour.

6.2.4 Signal Waveform Verification

During this verification method, ISim's waveform editor was used to verify the internal signal behaviour of the DUT. The waveform editor was used to display the values of the DUT's signals, as shown in Figure 6.2, during simulation. By inspecting these signal values during the simulation process, the frequencies of signals and the timing offsets between related signal edges could be verified. This inspection process was very time-consuming, and thus, only a small part of the DUT's behaviour could be verified by means of this verification method.

Furthermore, the signals whose behaviour were verified by this inspection method include:

- Video timing signals of the video transmitter component
- Output clocks of the clock generator component
- Address, data and control signals of memory components such as the primitive memory and raster memory
- SPI signals that are transmitted from the MCU model to MicroGE
- TMDS clock and data signals that are transmitted from the video transmitter component to the HDMI monitor model

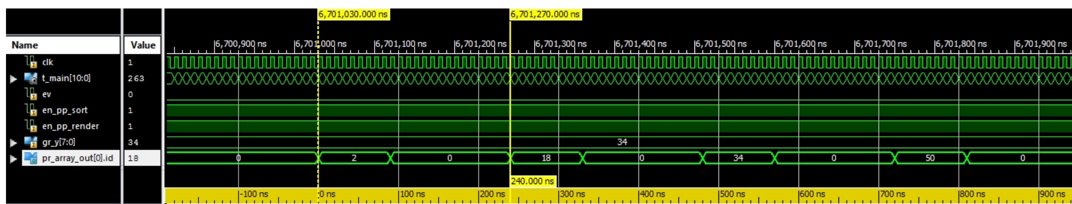


Figure 6.2: Screenshot of how ISim's waveform editor was used to inspect the behaviour of some of the DUT's signals

6.2.5 Memory Contents Verification

During this verification method, ISim's memory editor was used to display the contents of memory components of MicroGE, as shown in Figure 6.3, so that it could be verified that data was transferred to these memory components correctly. As the test bench was simulated, the MCU model transferred render information, such as raster blocks, character pointers, and primitives, to MicroGE. The memory editor was then used to inspect the contents of the raster memory, string memory, primitive buffer and primitive memory, to confirm that the correct render information was transferred to the correct memory locations. The contents of the pointer memories and line buffers were also inspected to confirm that the correct pointers and canvas row pixels were created, from that specific set of render information, and that those pointers and canvas row pixels were transferred to the correct memory locations. The inspection of the contents of these memory components was a very time-consuming process, and thus, this verification method could only be used to verify a small part of the DUT's behaviour.

6.2.6 Automated Verification

Figure 6.4 shows the verification method that was used for automated verification of the DUT's behaviour. Since this verification method is automated, it is much less time-consuming than the previous two verification methods, and thus, allowed more detailed verification of the DUT's behaviour. During this verification method, the DUT generates a video frame from render information provided by the MCU model. The verification method consists of three stages which each automatically produces

	0	1	2	3
0x37	55	54	53	52
0x33	51	50	49	48
0x2F	47	46	45	44
0x2B	43	42	41	40
0x27	39	38	37	36
0x23	35	34	33	32
0x1F	31	30	29	28
0x1B	27	26	25	24
0x17	23	22	21	20
0x13	19	18	17	16
0xF	15	14	13	12
0xB	11	10	9	8
0x7	7	6	5	4
0x3	3	2	1	0

Figure 6.3: Screenshot of how ISim's memory editor was used to inspect the contents of one of the DUT's memory components

output from provided input. During the first stage, a Python script uses four files, which describe a video frame, to produce a file containing a list of instructions. During the second stage, these instructions are used by the test bench so that the corresponding video frame data can be generated and stored in a file. Finally, during the third stage, a Python script verifies the generated video frame data.

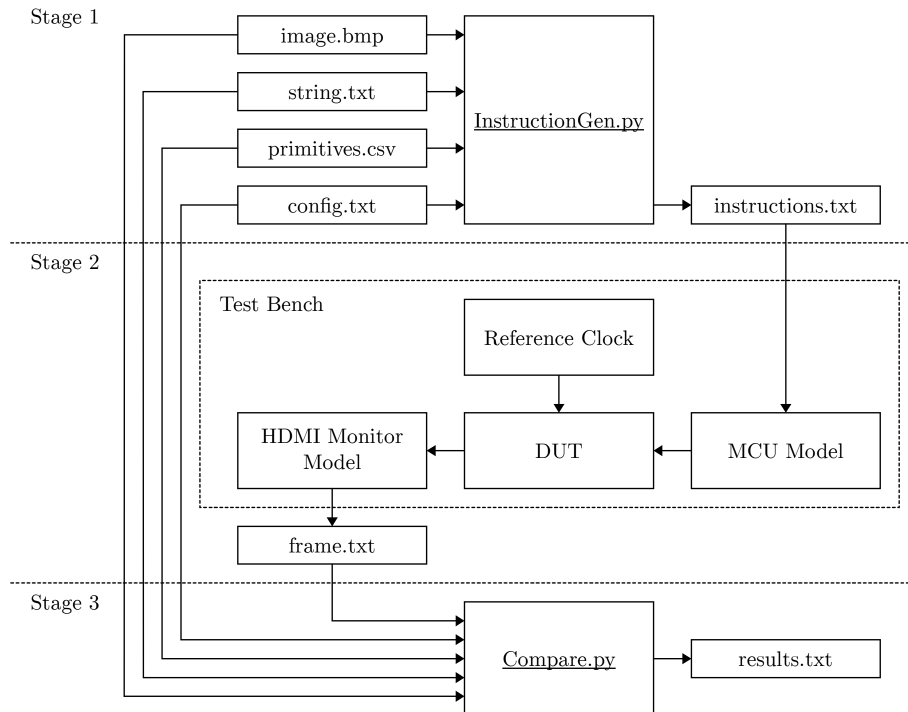


Figure 6.4: Block diagram describing the automated verification method

During the first stage, the InstructionGen.py Python script uses four files as input, which describe the contents of a video frame that should be generated by the DUT, to produce the instructions.txt file. The first of these four files is image.bmp which contains an array of pixels that should be transferred to MicroGE's raster memory. The second file is string.txt which contains a set of character pointers that should be transferred to MicroGE's string memory. The third file is primitives.csv

which contains the primitives that should be transferred to MicroGE's primitive buffer and primitive memory. The last file is config.txt which specifies the contents that should be provided to MicroGE's register map. The InstructionGen.py script uses these four files to produce the instructions.txt file. The instructions.txt file contains a list of instructions that can be provided to MicroGE, within the DUT, so that the video frame described by the image.bmp, string.txt, primitives.csv and config.txt files can be generated. Also, the instructions in the instructions.txt file are identical to the instructions that would be transferred from the Arduino Due to the test board, as discussed in Section 6.3.1.

During stage two, the test bench uses the instructions.txt file as input and generates the frame.txt file which describes a video frame generated by the DUT. The MCU model in the test bench reads the instructions from the instructions.txt file and transfers these instructions to MicroGE, within the DUT, by means of simulated SPI signals. MicroGE then generates the canvas according to the transferred instructions so that the canvas can be transmitted to the HDMI monitor model as part of the video frame. The video frame data is transmitted to the HDMI monitor model by means of simulated TMDS signals. The HDMI monitor model recovers the video frame data from these TMDS signals and then writes the video frame data to the frame.txt file.

During stage three, the Compare.py script uses five files as input to determine if the video frame data generated by the DUT during stage two is correct. The first four files, image.bmp, string.txt, primitives.csv and config.txt, are the files that are also provided to the InstructionGen.py script. The fifth file is the frame.txt file which is produced by the DUT during stage two. The Compare.py script uses the first four files as input to produce a video frame exactly like the DUT should have produced during stage two and then stores the pixel values of this video frame in a BMP image file. The Compare.py script then uses the frame.txt file as input, which contains the video frame that was actually generated by the DUT during stage two, and then creates another BMP image file containing the pixels of this video frame. The pixel colour values of these two BMP image files are then compared, and if any discrepancies are found, the locations of the pixels at which the discrepancies occur are written to the results.txt file. After completion of the script's execution, the results.txt file can be inspected to determine if MicroGE, within the DUT, produced the video frame from the render information specified in the image.bmp, string.txt, primitives.csv and config.txt files correctly.

6.3 Test System Firmware Verification

6.3.1 Tests System

A block diagram of the test system that was used to verify the behaviour of the firmware developed in this study is shown in Figure 6.5. This test system was created according to the system described in Figure 4.1. The main component of the test system, the test board, contains the Spartan-6 LX25 FPGA onto which the bitstream of the developed VHDL design was uploaded. As discussed in Section 4.1, this VHDL design contains MicroGE, the clock generator, the EDID reader and the video transmitter. The Arduino Due was loaded with firmware, from the laptop, which used MGAPI's functions to allow the Arduino Due to control MicroGE, on the

FPGA of the test board, via the SPI interface. MicroGE provided the corresponding graphics output to the HDMI display monitor via the HDMI interface. The Saleae logic analyser was used to measure signals of the SPI interface so that the waveforms of those signals could be inspected on the laptop. The Xilinx programmer was used to upload the bitstream of the VHDL design to the FPGA on the test board and to measure signals within the FPGA so that the waveforms of those signals could be inspected on the laptop.

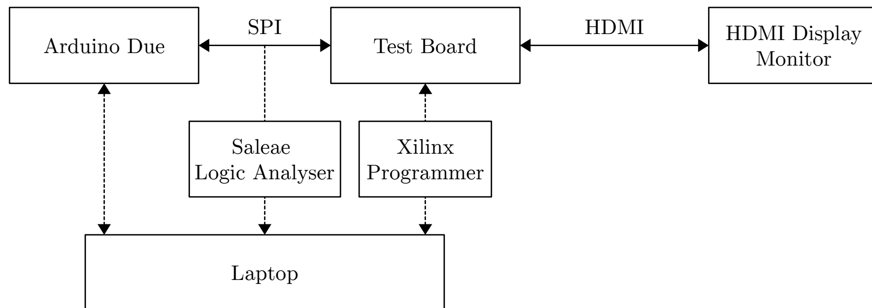


Figure 6.5: Block diagram of the test system that was used to verify the behaviour of the firmware developed in this study

A photo of the test system, excluding the Saleae logic analyser, is provided in Figure 6.6.

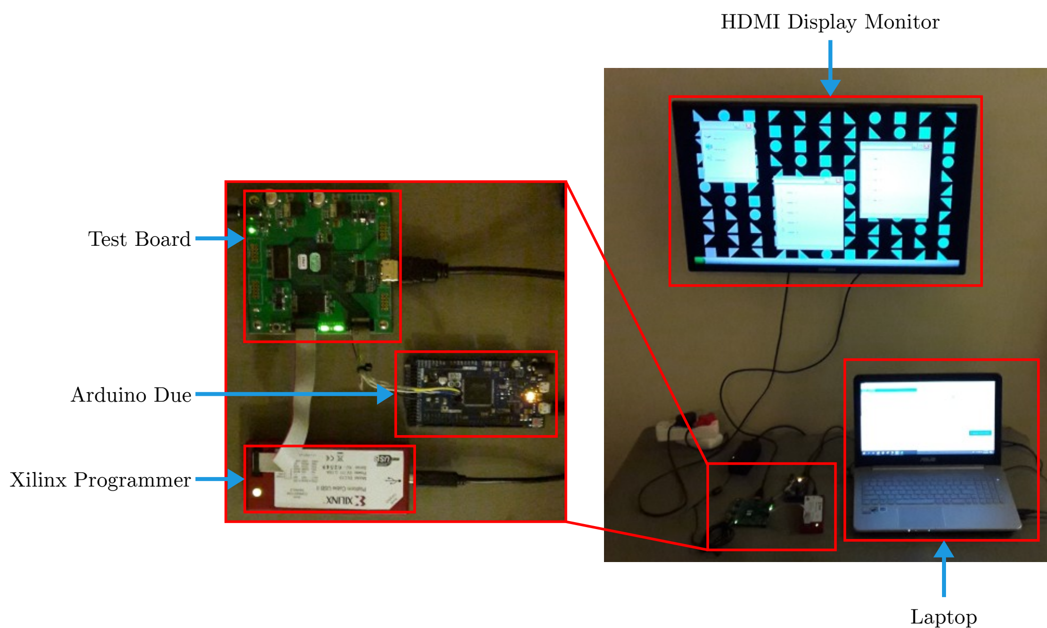


Figure 6.6: Photo of the test system that was used to verify the behaviour of the firmware developed in this study

Before the test system could be used to demonstrate MicroGE's performance and rendering capabilities, it had to be confirmed that MicroGE's real behaviour, the behaviour of MicroGE when it was loaded onto the FPGA of the test board, corresponded with MicroGE's simulated behaviour. First, as discussed in Section 6.3.2, the functionality of the SPI interface between the Arduino Due and MicroGE,

on the FPGA of the test board, was verified. After this verification, the Arduino Due could transfer render information to MicroGE. Then, as discussed in Section 6.3.3, MicroGE's graphics output could be displayed on the HDMI display monitor so that MicroGE's behaviour could be evaluated. Initially, this behaviour was incorrect, but after performing the procedure discussed in Section 6.3.4, the cause of this incorrect behaviour could be found. After making the appropriate adjustments to the VHDL design, the simulated and real behaviour of MicroGE corresponded.

6.3.2 SPI Interface Verification

This verification method made use of the Saleae logic analyser to verify the functionality of the SPI interface between the Arduino Due and the FPGA on the test board. The Saleae logic analyser has a maximum sample rate of 24 MSa/s which made it suitable for verification of the SPI signals that were transferred from the Arduino Due to the FPGA on the test board. Although, the speed of the SPI interface had to be decreased during this verification method to avoid aliasing of the measured SPI signals. The measured SPI signals were displayed on the laptop as signal waveforms so that those signals could be inspected. The functionality of the SPI interface was verified by instructing the Arduino Due to transfer a specific set of bytes across the SPI interface and then inspecting the signal waveforms to verify that those bytes were transferred correctly.

6.3.3 Display Monitor Output Verification

During this verification method, MicroGE's graphics output was inspected on the HDMI display monitor. Even though inspecting the graphics output on the display monitor did not provide detailed information on the internal behaviour of MicroGE, it provided an overview of MicroGE's behaviour. The verification method discussed in Section 6.2.6 provided detailed information on MicroGE's behaviour, but it required more time to perform. On the other hand, during this verification method, many different combinations of render information could be transferred to MicroGE in a short period, and thus, allowed the inspection of MicroGE's behaviour based on many different sets of render information in a short period.

6.3.4 Debugging of VHDL Design

The ChipScope Integrated Logic Analyzer (ILA) was used to debug the VHDL design when the real behaviour of the VHDL design did not correspond with the simulated behaviour of the VHDL design. The ILA is a customisable core that can be added to an existing VHDL design so that the internal signals of that VHDL design can be monitored when that VHDL design is loaded onto an FPGA [38]. The Xilinx programmer can be used to interface with the ILA so that real-time signal data can be captured and then be displayed. The ILA was added to the VHDL design of this study so that signals of the VHDL design could be measured and then be displayed on the laptop of the test system for inspection. An advantage of using an ILA for verification, as opposed to using the verification methods discussed in Section 6.2, is that the ILA provides the real behaviour of the VHDL design as opposed to the simulated behaviour of the VHDL design. On the other hand, the use of an ILA

requires the entire VHDL design to be rebuilt every time a change is made to the VHDL code and it only provides access to post-synthesis signals.

Initially, the real behaviour of the VHDL design did not correspond with the simulated behaviour of the VHDL design. During the visual inspection of the signal waveforms provided by the ILA, it was found that some of the measured signals had unpredictable behaviour. Upon further investigation, it was discovered that this unpredictable behaviour occurred due to signals that cross clock domains. In the VHDL design, these signals are the ones that are used to transfer data between MicroGE's clock domain and the video transmitter's clock domain. After performing research, it was discovered that when signals cross clock domains, it can cause metastability [39]. Metastability occurs when a signal that is provided to the input of a register does not meet timing relative to the clock signal that is provided to that register. This is what happens when signals cross clock domains. The problem was solved by using a simple solution proposed by [39]. By adding synchronisation registers to all of the signals that cross clock domains, the unpredictable signal behaviour caused by metastability was removed. This allowed the real behaviour of the VHDL design to correspond with the simulated behaviour of the VHDL design.

6.4 Test Patterns

Various "test patterns" were developed to demonstrate MicroGE's rendering capabilities. These test patterns were implemented on the Arduino Due, along with MGAPI, and demonstrated on the test system. During each test pattern, the Arduino Due transferred render information to MicroGE, within the FPGA of the test board, which then rendered the graphics accordingly so that these graphics could be displayed on the HDMI display monitor. Photos were taken of the test patterns and are shown in the following sections. Due to the difference between the scanning frequency of the HDMI display monitor and the scanning frequency of the camera used to take the photos, "flickering" and "banding" artefacts are visible in some of the photos. These artefacts were not visible to the naked eye.

Vector Primitive Test Pattern One

During the test pattern shown in Figure 6.7, 1000 primitives, positioned in a 25×40 array, were moved around the canvas. This array consisted of different vector primitives which each had a width and height of one grid unit. The Arduino Due produced this test pattern by repeating the procedure of clearing the primitive buffer and transferring a set of 1000 primitives to MicroGE. There was a delay of one second between these transfers. The same set of primitives was transferred every second, but the horizontal positions of the primitives were incremented before each transfer. This created the appearance of the same set of primitives moving to the right. This test pattern demonstrated how MicroGE is capable of rendering 1000 primitives on the canvas at the same time and is also capable of updating the primitives rendered on the canvas based on render information transferred from the Arduino Due.

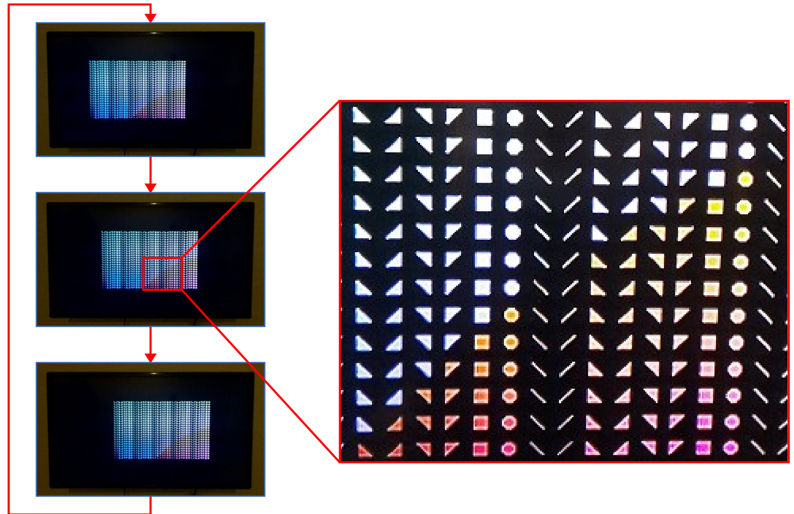


Figure 6.7: Photos of parts of the first vector primitive test pattern

Vector Primitive Test Pattern Two

During the test pattern shown in Figure 6.8, vector primitives, also positioned in a 2D array, were displayed on the canvas, but their sizes were increased as the test pattern progressed. The Arduino Due produced this test pattern by repeating the procedure of clearing the primitive buffer and transferring a different set of vector primitives, of the same type, to MicroGE. There was a delay of one second between these transfers. Before each transfer, half of the primitives were removed while the sizes of the remaining primitives were doubled. This process was repeated for each type of vector primitive. This test pattern demonstrated how MicroGE is capable of rendering multiple vector primitives on the canvas in different sizes.

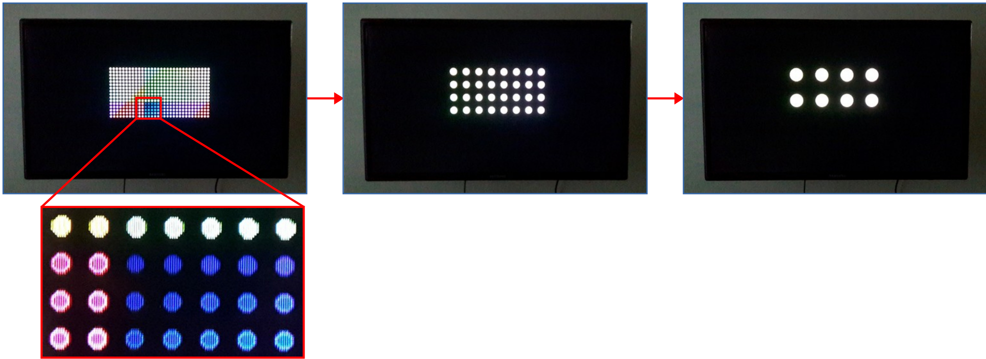


Figure 6.8: Photos of parts of the second vector primitive test pattern

Vector Primitive Test Pattern Three

During this test pattern, vector primitives were flattened in both the horizontal and vertical directions as the test pattern progressed. Figure 6.9 shows an example of how the ellipse primitive was flattened in the vertical direction. The Arduino Due produced this test pattern by repeating the procedure of clearing the primitive buffer

and then transferring a new vector primitive, of the same type but a different width or height, to MicroGE. There was a delay of one second between these transfers. Before each transfer, either the width or height of the primitive was decreased to create the appearance of the primitive being flattened in either the horizontal or vertical direction, respectively. This process was repeated for each type of vector primitive. The purpose of this test pattern was to demonstrate how MicroGE is capable of rendering vector primitives with different aspect ratios.

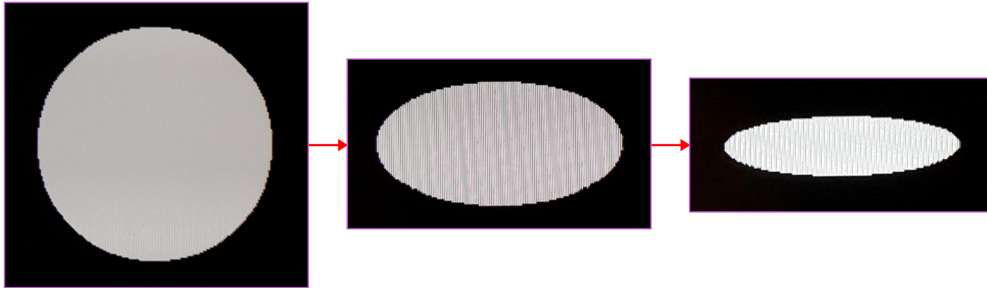


Figure 6.9: Photos of parts of the third vector primitive test pattern

Depth Buffer Test Pattern

This test pattern demonstrated the depth buffer functionality of MicroGE by rendering eight vector primitives of the same type but different sizes on top of each other. First, a single primitive, with a large width and height, was transferred to MicroGE. This primitive was located on the lowest depth layer. Then, after a delay of one second, a smaller primitive, which was located on a higher depth layer, was transferred to MicroGE. This process was repeated until there were eight primitives positioned on top of each other with decreased sizes as they were located on higher depth layers. Thus, it seemed like the primitives were added on top of each other throughout the test pattern. This process was repeated for each type of vector primitive. Figure 6.10 shows a part of this test pattern where eight ellipse primitives were rendered on top of each other.

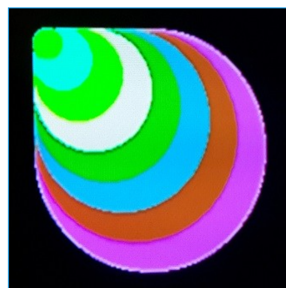


Figure 6.10: Photo of a part of the depth buffer test pattern

Image Primitive Test Patterns

In each of the test patterns shown in Figure 6.11 and Figure 6.12, a single image was rendered on multiple locations of the canvas to create the appearance of a

continuous image. In each of these test patterns, an image, with a resolution of 128 $\times$ 128 pixels, was transferred to MicroGE's raster memory. Then, multiple image primitives, all of the same size, were transferred to MicroGE. These primitives were positioned in a 2D array, and there were no empty spaces between them. The image stored in the raster memory was rendered within each of these image primitives. These test patterns demonstrated how MicroGE is capable of rendering the same raster memory contents on multiple locations of the canvas.

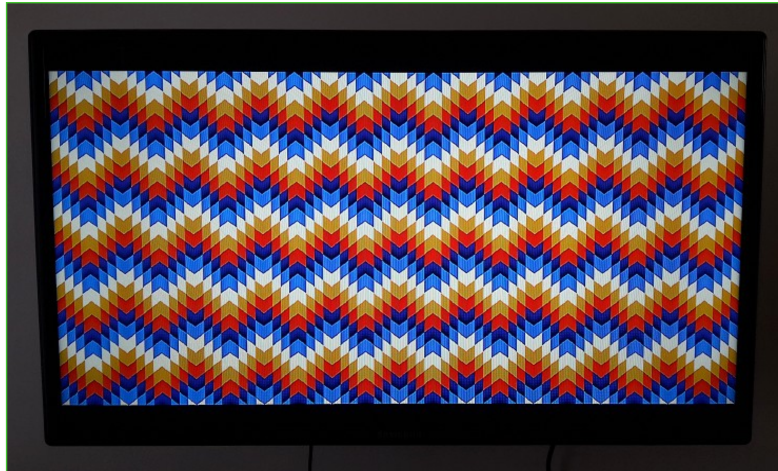


Figure 6.11: Photo of the first image primitive test pattern

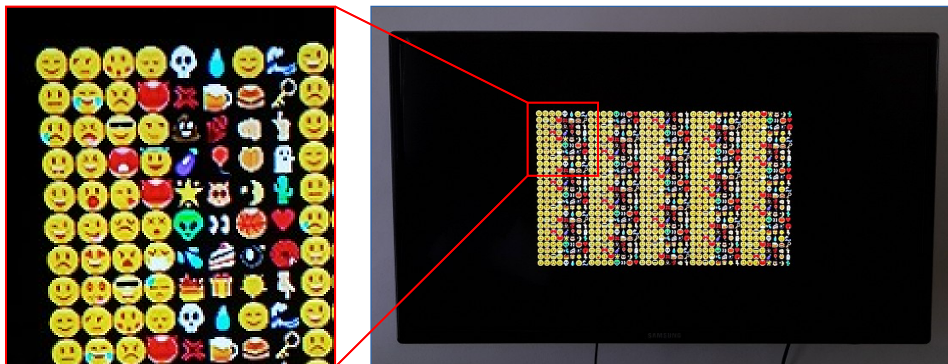


Figure 6.12: Photos of the second image primitive test pattern

String Primitive Test Pattern

During the test pattern shown in Figure 6.13, lines of ASCII characters scrolled down the canvas. First, character raster blocks which each depicted an ASCII character were transferred to the raster memory. Then, character pointers which each pointed to one of these character raster blocks were transferred to the string memory. Then, 90 string primitives which each had a width of 160 grid units were transferred to MicroGE. These primitives were positioned at subsequent vertical grid rows of the canvas. A specific subset of a set of character raster blocks was rendered within each string primitive so that it appeared that the character raster blocks within each primitive were shifted to the right by one grid unit relative to the character

raster blocks within the primitive above it. Every second, the primitive buffer was cleared, and the 90 string primitives were transferred to MicroGE again. Before each transfer, the *adr* attribute of each one of these primitives was updated so that the character raster blocks rendered within each one of the string primitives could be shifted further to the right by one grid unit. This created the appearance of lines of ASCII characters scrolling down the canvas. This test pattern demonstrated how MicroGE is capable of rendering large amounts of text on the canvas.

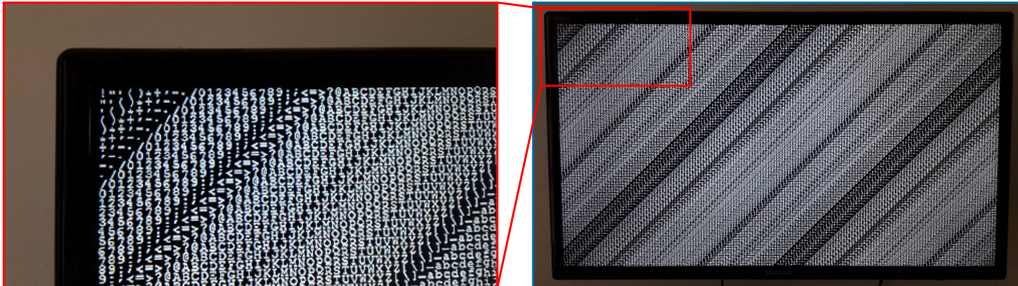


Figure 6.13: Photos of parts of the string primitive test pattern

Potential Application of MicroGE

Figure 6.14 demonstrates a potential application of MicroGE by combining some of MicroGE's rendering capabilities. A simple window system was rendered using a combination of primitives along with string and raster memory contents. The raster memory was filled with a combination of image segments and icons which were used to render windows, title bars and a taskbar. The button of the taskbar was created using an image primitive, while the rest of the taskbar was created by repeating image segments within string primitives. The three windows were created using a combination of vector and raster primitives. The white backgrounds of these windows were created using rectangular vector primitives while the boundaries and title bars of these windows were created by repeating image segments around the peripheries of the rectangular vector primitives. The icons and text within the windows were created using image primitives. Finally, the background consisted of an array of vector primitives which were located on the lowest depth layer so that they could appear behind all of the other primitives.

6.5 Resource Utilisation Results

MicroGE's resource usage was assessed by investigating to what extent MicroGE could be implemented within the memory restrictions listed in Section 1.4. One of these restrictions required MicroGE to be implemented on a low-resource FPGA. Since the FPGA that MicroGE was implemented on in this study is a low-resource FPGA, the Spartan-6 LX25 FPGA, it was already confirmed that MicroGE satisfies this requirement. The other restrictions required MicroGE to be implemented within a small number of resources of modern high-resource FPGAs and required MicroGE to use no more than 1 Mb of an FPGA's RAM resources. To verify that MicroGE

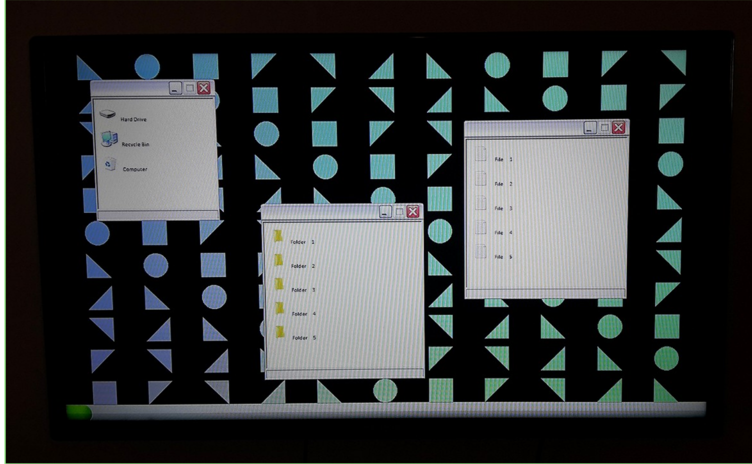


Figure 6.14: Photo of a potential application of MicroGE

could be implemented within these restrictions, the resource usage report of the VHDL design developed in this study was examined.

The resource usage report, provided by ISE Design Suite 14.7, of the VHDL design that was implemented on the Spartan-6 LX25 FPGA is provided in Appendix C. To compare the device-specific Spartan-6 LX25 FPGA resources that MicroGE utilised with the available resources of other Xilinx FPGAs, these device-specific Spartan-6 LX25 FPGA resources were converted to a generic form that is common across Xilinx FPGAs. The most important of these converted resources are listed in Table 6.1. The first listed resource, block RAM memory, was determined from the number of RAMB16BWER components that were utilised by MicroGE. In this study, MicroGE utilised a total of 44 RAMB16BWER components, each with a memory capacity of 18 kb. Thus, MicroGE's block RAM memory usage was calculated as 792 kb. This is significantly lower than the block RAM memory capacity available in the latest Xilinx UltraScale FPGAs, which range from 4.5 to 132.9 Mb [40]. Additionally, MicroGE's block RAM memory usage can be decreased even further by decreasing the values of the RAST_MEM_DPRAM_NUM and STR_MEM_DPRAM_NUM generic statements, which would decrease the number of RAMB16BWER components included in MicroGE's raster and string memories, respectively. The second listed resource, logic cells, was determined by multiplying the number of Spartan-6 LX25 FPGA slices utilised by MicroGE, 2970, with 6.4 to obtain 19008 [14]. This is also significantly lower than the number of logic cells available in the latest Xilinx UltraScale FPGAs, which range from 103000 to 5541000 [40]. Additionally, since the logic cell usage of the video transmitter, clock generator and EDID reader is included in the logic cell usage of 19008, the logic cell usage of MicroGE alone is slightly lower than 19008. The last listed resource, DSP slices, is 17. The latest Xilinx UltraScale FPGAs not only have much more of these components, 240 to 12288, but the components themselves are improved [40].

From these results, it is clear that MicroGE could be implemented within the FPGA resource restrictions that are listed in Section 1.4. MicroGE's block RAM memory usage is lower than the maximum specified limit, which is 1 Mb, and MicroGE could also be implemented without requiring external memory components, such as SDRAM. Since MicroGE was successfully implemented within the limited

Resource	Utilisation	Available
Block RAM memory	792 kb	936 kb
Logic cells	19008	24051
DSP slices	17	38

Table 6.1: Summary of the resource utilisation of the VHDL design that was implemented on the Spartan-6 LX25 FPGA

resources of a Spartan-6 LX25 FPGA, it was verified that MicroGE could fit on a low-resource FPGA. From the comparison of MicroGE's resource utilisation on the Spartan-6 LX25 FPGA with the amount of resources available in modern high-resource Xilinx UltraScale FPGAs, it was verified that MicroGE would use a minimal amount of those high-resource FPGAs' resources. Additionally, VHDL designs that are implemented on those high-resource FPGAs can incorporate MicroGE without causing a significant increase in the resource usage of those VHDL designs.

6.6 Performance Results

MicroGE's performance is mostly dependent on how fast its render information can be updated. This render information includes the primitives, raster blocks and character pointers that are stored in MicroGE's memory components. MicroGE renders the canvas at the same rate, which is the frame update rate of the current video mode, no matter what set of render information is stored in its memory components. Whether this set of render information includes only a few vector primitives or a combination of a large quantity of vector and raster primitives, the rate at which the canvas is rendered is equal to the frame update rate of the current video mode. However, when the contents that should be rendered on the canvas should be updated, this update rate is dependant on how fast render information can be transferred to MicroGE's memory components. This update rate is mostly dependant on the speed at which render information can be transferred from MicroGE's controller to MicroGE via the SPI interface. This speed is mostly dependant on the frequency of the SPI interface and the delays between SPI transactions. In this study, MicroGE's performance was tested using the Arduino Due as MicroGE's controller, which transferred render information to MicroGE at an SPI frequency of 8 MHz. The following three sections discuss the results of the tests that were performed.

6.6.1 Raster Memory Update Speed Test

The purpose of this test was to determine how fast raster blocks could be transferred to MicroGE's raster memory. During this test, the time periods required to transfer different quantities of raster blocks from the Arduino Due to MicroGE were measured using one of the Arduino Due's timer registers. Each raster block was 8 × 8 pixels in size, allowing the entire raster memory to be filled with 256 raster blocks. The Arduino Due used the MgapiTransferRasterBlock function, discussed in Section 4.4.2, to transfer each one of these raster blocks. Figure 6.15 shows the time periods that were required to transfer a range of one to 256 raster blocks to MicroGE's raster memory. The time periods required to transfer one, 32, 128 and 256 raster blocks were one, 40, 159 and 318 ms, respectively. There is a linear relationship between

the quantities of raster blocks transferred and the time periods required to transfer these raster blocks. If the size of the raster memory is increased, and more raster blocks can be transferred to the raster memory, this linear relationship should stay the same. 318 ms is a relatively long time period to transfer 256 raster blocks, or a 128 × 128 sized image, to the raster memory, but this delay will only be present during the transfer of the raster blocks. After the raster blocks are stored in the raster memory, they can be rendered within any of the raster primitives positioned on the canvas. These performance results are acceptable since the typical applications of MicroGE, as listed in Section 1.4, do not require the raster memory to be updated frequently.

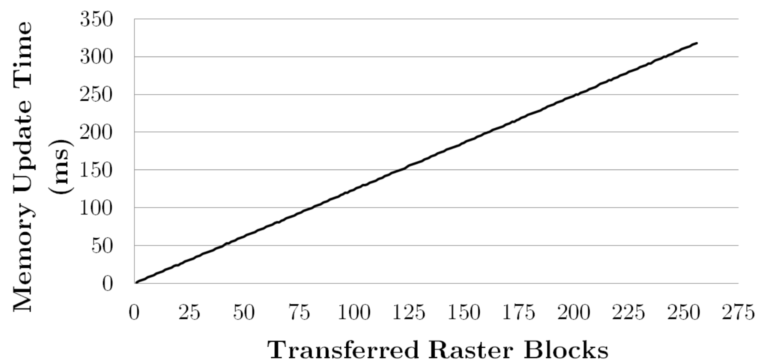


Figure 6.15: Performance results of the raster memory update speed test

6.6.2 String Memory Update Speed Test

The purpose of this test was to determine how fast character pointers could be transferred to MicroGE's string memory. During this test, the time periods required to transfer different quantities of character pointers from the Arduino Due to MicroGE were measured using one of the Arduino Due's timer registers. During each of these measurements, integer multiples of six character pointers were transferred to MicroGE, using the MgapiTransferCharacterPointers function, discussed in Section 4.4.2. To fill the string memory, 1024 character pointers had to be transferred, which required integer multiples of six character pointers to be transferred first followed by a single transfer of four character pointers. Figure 6.16 shows the time periods that were required to transfer these different quantities of character pointers to MicroGE's string memory. The time periods required to transfer six, 300, 600 and 1024 character pointers were one, eight, 15 and 25 ms, respectively. There is a linear relationship between the quantities of character pointers transferred and the time periods required to transfer these character pointers. If the size of the string memory is increased, and more character pointers can be transferred to the string memory, this linear relationship should stay the same. These performance results are satisfactory and allow MicroGE to update large quantities of text on the canvas very quickly. Thus, MicroGE can be used for command line interface applications, which is listed as one of the typical applications of MicroGE in Section 1.4.

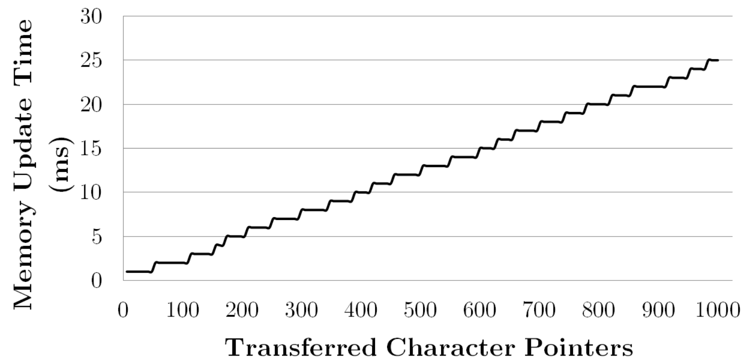


Figure 6.16: Performance results of the string memory update speed test

6.6.3 Primitive Memory Update Speed Test

This test determined how fast the contents of MicroGE's primitive memory could be updated, and thus, determined how fast the contents rendered on MicroGE's canvas could be updated. During this test, the time periods required to transfer different quantities of primitives from the Arduino Due to MicroGE were measured using one of the Arduino Due's timer registers. During each one of these transfers, a set of primitives was transferred to the primitive buffer first, using the MgapiInsert-Primitive function, so that those primitives could be transferred from the primitive buffer to the primitive memory, using the MgapiUpdatePrimitiveMemory function, directly after. Figure 6.17 shows the time periods that were required to transfer different quantities of primitives, ranging from one to 1000, to MicroGE's primitive memory. The time periods required to transfer 10, 100 and 1000 primitives were one, 11 and 111 ms, respectively. These time periods were independent of the attributes of the primitives that were transferred, which determine the type, position, size and colour of those primitives, since the amount of data that the MgapiInsertPrimitive function transfers for any set of primitive attributes is the same. These time periods were also independent of the size of the canvas that was rendered. Thus, the worst-case update speed of MicroGE's primitive memory was determined as 111 ms. This allows MicroGE to have a much faster frame update rate than specified in Section 1.4, which is one frame per second. To conclude, these results make MicroGE suitable for all of the typical applications listed in Section 1.4.

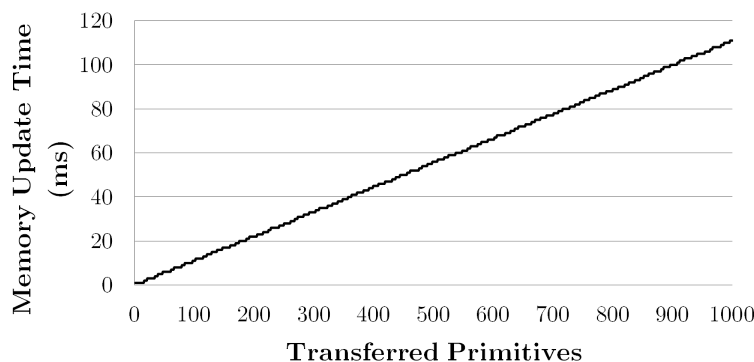


Figure 6.17: Performance results of the primitive memory update speed test

Chapter 7

Conclusions and Recommendations

7.1 Conclusions

This study focused on the design and implementation of a low-resource 2D graphics engine, MicroGE, which can be implemented on an FPGA. MicroGE's architecture was designed differently from those of existing graphics rendering systems so that MicroGE could be implemented entirely within an FPGA and also use minimal FPGA resources. To prove that MicroGE's architecture could be realised, it was implemented as part of a VHDL design. A VHDL test bench was developed so that MicroGE's behaviour could be verified in simulation. A test board, containing a low-resource Spartan-6 LX25 FPGA, was also developed so that MicroGE's behaviour could be verified on an actual FPGA platform. A software library, MGAPI, was developed which allowed an Arduino Due to control MicroGE when MicroGE was loaded onto the FPGA of the test board. Finally, MicroGE, MGAPI, the Arduino Due and the test board were all used as part of a test system to demonstrate MicroGE's rendering capabilities, to determine MicroGE's resource utilisation, and to determine MicroGE's performance.

The functionalities of the test board were verified using the procedure discussed in Section 5.5. At the beginning of this procedure, it was found that the supply voltages that were provided from two of the voltage regulators were out of tolerance. This was because these voltage regulators were swapped during the PCB assembly process and then mounted onto incorrect footprints. After this mistake was rectified, the remaining aspects of the test board could be tested, and all functionalities of the test board could be verified.

The VHDL design, containing MicroGE, was successfully implemented by means of ISE Design Suite 14.7. All aspects of MicroGE's architecture, discussed in Chapter 3, were possible to implement using VHDL code. However, even though it was possible to implement scalable raster and string memories, the full extent of this scalability could not be verified due to the limited resources of the Spartan-6 LX25 FPGA. Also, even though MicroGE was implemented with portability in mind, MicroGE has not been ported to other FPGA platforms yet.

The behaviour of MicroGE was successfully verified after performing the verification methods discussed in Section 6.2 and Section 6.3. After verifying MicroGE's behaviour in simulation, MicroGE was tested on the test board. Initially, MicroGE's

behaviour on the test board did not correspond with MicroGE's simulated behaviour. This incorrect behaviour was a result of metastability phenomena which occurred due to signals that crossed clock domains. This incorrect behaviour did not occur during simulation since metastability is not simulated in discrete time simulators. After synchronisation registers were added to all signals that crossed clock domains, the simulated behaviour and the real behaviour of MicroGE corresponded, and thus, the behaviour of MicroGE was verified both in simulation and on the test board.

MGAPI was successfully implemented on the Arduino Due by means of the Arduino IDE. The behaviour of MGAPI was verified by inspecting the SPI waveforms that were produced when calling specific functions of MGAPI. After the behaviour of MGAPI was verified, the Arduino Due could use MGAPI's functions to control MicroGE successfully. Also, even though MGAPI was implemented with portability in mind, MGAPI has not been ported to other platforms yet.

MicroGE was successfully implemented within the specified FPGA resource restrictions. MicroGE's block RAM memory usage, 792 kb, was found to be lower than the specified limit of 1 Mb. Also, MicroGE's logic cell usage, 19008, and DSP slice usage, 17, is also very low when considering how much of these resources there are in modern high-resource FPGAs. These resource usage results allow MicroGE to be implemented on low-resource FPGAs without requiring external memory components, such as SDRAM. Also, it allows MicroGE to be implemented on modern high-resource FPGAs without requiring a significant amount of those FPGAs' resources.

MicroGE's performance results were found to be satisfactory, even when MicroGE was controlled from a device with low computing power, the Arduino Due. The Arduino Due could update MicroGE's entire primitive memory, raster memory and string memory within 111 ms, 318 ms and 25 ms, respectively. The minimum update rate of the contents rendered on MicroGE's canvas, which is determined from the time required to update the entire primitive memory, is much faster, approximately nine frames per second, than the specified limit of one frame per second. The update speed of the raster memory is relatively low, and thus, does not allow MicroGE to update images very quickly. Even so, the typical applications of MicroGE, as listed in Section 1.4, do not require the raster memory to be updated frequently, but rather once, to load a set of images to use throughout the graphics rendering application. The fast update speed of the string memory allows MicroGE to be used for command line interface applications, which is also listed as one of the typical applications of MicroGE in Section 1.4.

MicroGE's canvas can be displayed on both HDMI and DVI display monitors. Since MicroGE was designed to interface with an external video transmitter, MicroGE's canvas can be transmitted using different video transmission protocols. MicroGE can also support a wide variety of video modes without storing any configuration information about those video modes. This is because MicroGE derives video mode parameters from the external video transmitter that it is interfacing with. The maximum supported video resolutions of the video transmitter that MicroGE interfaced with in this study is 1360 × 768 for the progressive video modes and 1920 × 1080 for the 1080i interlaced video mode. Thus, it could be verified that MicroGE could render its canvas up to a resolution of 1280 × 720 for progressive video modes and 1920 × 1080 for interlaced video modes. Also, because of how MicroGE uses the PixelsPerGrid property for its rendering algorithms, the canvas

could be scaled to various sizes.

To conclude, MicroGE was successfully designed and implemented within the restrictions that were imposed on it. The resource usage of MicroGE was found to be low enough to allow MicroGE to be implemented on low-resource FPGAs, and low enough to allow MicroGE to be implemented within minimal resources of modern high-resource FPGAs. The performance of MicroGE, even for the worst cases, was found to be much better than the required performance, as specified in Section 1.4. These performance results allow MicroGE to be used in all of the applications listed in Section 1.4. Additionally, since MicroGE and MGAPI were implemented with portability in mind, their development can continue on various other platforms.

7.2 Recommendations

Even though the design and implementation of MicroGE were successful, there are still many aspects of MicroGE that can be improved. These aspects are discussed in the following sections.

Improvements of Rendering Capabilities

Many improvements can be made to MicroGE's rendering capabilities. These improvements can be listed as follows:

- The maximum size of the vector primitives, 20×20 grid units, can be increased
- The resolution of MicroGE's grid system can be increased so that more detailed graphics can be rendered on the canvas
- The capacity of the primitive memory can be increased so that more primitives can be rendered on the canvas
- Support for colour transparency can be added
- The ability to render vector primitives with colour gradients can be added
- More types of vector primitives can be added
- The maximum memory capacity of the raster memory can be increased
- The maximum memory capacity of the string memory can be increased
- Support for scaling of string primitives can be added to allow larger text to be rendered
- Anti-aliasing support can be added to the vector rendering process to improve the appearance of vector graphics
- The addition of a colour palette functionality to the raster render unit can greatly reduce the memory requirements of the raster memory

Power Consumption Optimisation

Even though low power consumption was not a requirement of MicroGE, MicroGE's power consumption can still be optimised. Many of the computations that are performed by MicroGE's rendering processes are redundant. These redundant computations were allowed to keep the implementation of MicroGE as simple as possible. However, these redundant computations cause MicroGE to consume more power. Removing these redundant computations will decrease the power consumption of MicroGE without influencing MicroGE's performance.

Performance Improvements

Many improvements can be made to MicroGE's performance. Since MicroGE's performance is mostly dependent on the speed at which render information can be transferred to its primitive, raster and string memories, its performance can be improved by increasing this speed. Currently, the maximum supported SPI frequency of MicroGE is 10 MHz. This frequency can be increased to increase the speed at which render information can be transferred. Also, support for other interfaces, such as AXI and Wishbone buses, can be added to MicroGE. These buses will allow render information to be transferred to MicroGE at a much higher speed than what could be achieved with an SPI interface.

Addition of a Raster Block Scaling Functionality

A raster block scaling functionality should be added to either MicroGE or MGAPI. MicroGE does not have the capability to scale raster blocks to the appropriate sizes. Instead, it is assumed that raster blocks are transferred to MicroGE, from its controller, in the appropriate sizes. MGAPI does not provide the functionality to scale raster blocks to the appropriate sizes before they are transferred to MicroGE from the controller. Thus, the user of MicroGE is responsible for ensuring that the raster blocks are sized appropriately before they are transferred to MicroGE from the controller. If a raster block scaling functionality is added to either MicroGE or MGAPI, the user will not have to be responsible for this any more.

Porting of MicroGE and MGAPI to Other Platforms

MicroGE and MGAPI can be ported to other platforms to allow their development to continue on those platforms. So far they have only been used on the test system of this study. If MicroGE is ported to an FPGA that has more memory resources than the Spartan-6 LX25 FPGA, the scalability of MicroGE's raster and string memories can be tested to a larger extent. Also, if MicroGE is ported to an FPGA whose DSP slices are better than those of the Spartan-6 LX25 FPGA, many of MicroGE's rendering capabilities can be improved.

Addition of Error Checking Functionalities

Error checking functionalities still need to be added to MicroGE's SPI interface. Neither MicroGE nor MGAPI performs any error checking on data that is transmitted across the SPI interface. Even though an error checking functionality was not

required to transmit SPI data reliably in the test system of this study, it might be required when MicroGE and MGAPI are ported to other platforms.

Bibliography

[1] J. Chen, W. Cranton, and M. Fihn, *Handbook of Visual Display Technology*, 2nd ed. Springer International Publishing, 2016.

[2] T. C. Inglis and C. S. Kaplan, "Pixelating vector line art," in *Proceedings of the Symposium on Non-Photorealistic Animation and Rendering.* Eurographics Association, June 2012, pp. 21–28.

[3] D. Da Silva, "Raster algorithms for 2d primitives," Master's thesis, Brown University, May 1989.

[4] Y. Ma, X. Wang, M. Zhu, and W. Wan, "Rasterization of geometric primitive in graphics based on fpga," in *Proceedings of the International Conference on Audio, Language and Image Processing.* IEEE, November 2010, pp. 1211–1216.

[5] X. Wang, F. Guo, and M. Zhu, "A more efficient triangle rasterization algorithm implemented in fpga," in *Proceedings of the International Conference on Audio, Language and Image Processing.* IEEE, July 2012, pp. 1108–1113.

[6] S. Zammattio, "Using fpgas to render graphics and drive lcd interfaces," *Intel White Paper, Version 1.0*, April 2009.

[7] K. S. Ay and A. Doan, "Hardware/software co-design of a 2d graphics system on fpga," *International Journal of Embedded Systems and Applications*, vol. 3, no. 1, March 2013.

[8] B. Fagner and M. Gustafsson, "Design and implementation of a 2d acceleration engine for a video controller," Master's thesis, Chalmers University of Technology, November 2009.

[9] S. H. Chen, H. M. Lin, H. W. Wei, Y. C. Chen, C. T. Huang, and Y. C. Chung, "Hardware/software co-designed accelerator for vector graphics applications," in *Proceedings of the 9th Symposium on Application Specific Processors.* IEEE, 2011, pp. 108–114.

[10] BitSim, "Bitsim accelerated display graphics engine," *BitSim Product Brief*, 2017.

[11] TES Electronic Solutions, "D/ave 2d," *TES Electronic Solutions Datasheet TD-240201SH DS, Version 2.9*, August 2012.

[12] P. A. Greczner, "Two-dimensional graphics card (gpu) on an altera fpga," Master's thesis, Cornell University, May 2010.

[13] Prevas, "Graphics controller core with 2d acceleration functionalities," *Prevas Product Specification*, February 2010.

[14] Xilinx, "Spartan-6 family overview," *Xilinx Product Specification DS160, Version 2.0*, October 2011.

[15] V. Kasik, "Fpga-powered embedded vector graphics," in *Proceedings of the Second International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies*. IEEE, September 2008, pp. 419–421.

[16] Digital Display Working Group, "Digital visual interface," *DVI Specification, Revision 1.0*, April 1999.

[17] Hitachi, Ltd. et al., "High-definition multimedia interface specification," *HDMI Specification, Version 1.3a*, November 2006.

[18] Video Electronics Standards Association, "VESA DisplayPort standard," *VESA Standard, Version 1, Revision 1a*, January 2008.

[19] ——, "Vesa and industry standards and guidelines for computer display monitor timing," *VESA Standard, Version 1.0, Revision 11*, May 2007.

[20] Consumer Electronics Association, "A dtv profile for uncompressed high speed digital interfaces," *CEA Standard, CEA-861-D*, July 2006.

[21] Video Electronics Standards Association, "VESA enhanced extended display identification data standard," *VESA Standard, Release A, Revision 1*, February 2000.

[22] ——, "Enhanced display data channel standard," *VESA Standard, Version 1.1, Revision 1*, March 2004.

[23] H. Zhang, S. Krooswyk, and J. Ou, *High Speed Digital Design: Design of High Speed Interconnects and Signaling*. Elsevier, 2015.

[24] A. J. Burkhardt, C. S. Gregg, and J. A. Staniforth, "Calculation of pcb track impedance," *Circuit World*, vol. 26, no. 1, pp. 6–10, March 1999.

[25] D. Long, "Considerations for pcb layout and impedance matching design in optical modules," *Texas Instruments Application Report SLLA311*, February 2011.

[26] Xilinx, "Spartan-6 fpga dsp48a1 slice user guide," *Xilinx User Guide UG389, Version 1.2*, May 2014.

[27] ——, "Spartan-6 fpga block ram resources user guide," *Xilinx User Guide UG383, Version 1.5*, July 2011.

[28] ——, "Spartan-6 fpga clocking resources," *Xilinx User Guide UG382, Version 1.10*, June 2015.

[29] B. Feng, "Implementing a tmds video interface in the spartan-6 fpga," *Xilinx Application Note XAPP495, Version 1.0*, December 2010.

[30] S. Larson, "I2C Master (VHDL)," November, 2017. [Online] Available: https://www.digikey.com/eewiki/pages/viewpage.action?pageId=10125324. [Accessed on 2017-07-03].

[31] Texas Instruments, "Lmz22003 3-a simple switcher power module with 20-v maximum input voltage," *Texas Instruments Datasheet SNVS658I, Revision I*, August 2015.

[32] ——, "Lm1117 800-ma low-dropout linear regulator," *Texas Instruments Datasheet SNOS412N, Revision N*, January 2016.

[33] Xilinx, "Spartan-6 fpga packaging and pinouts," *Xilinx User Guide UG385, Version 2.3*, May 2014.

[34] ——, "Spartan-6 fpga configuration user guide," *Xilinx User Guide UG380, Version 2.10*, March 2017.

[35] G. Yater, "Preventing esd arcing in a connector," *Texas Instruments Application Report SLVA707*, June 2015.

[36] Texas Instruments, "Tpd12s521 single-chip hdmi transmitter port protection and interface device," *Texas Instruments Datasheet SLVS639F, Revision F*, February 2016.

[37] Digi-Key, "PCB Trace Width Conversion Calculator," 2017. [Online] Available: https://www.digikey.co.za/en/resources/conversion-calculators/conversion-calculator-pcb-trace-width. [Accessed on 2017-06-01].

[38] Xilinx, "Logicore ip chipscope pro integrated logic analyzer (ila)," *Product Specification DS299, Version 1.04a*, June 2011.

[39] J. Stephenson, D. Chen, R. Fung, and J. Chromczak, "Understanding metastability in fpgas," *Intel White Paper, Version 1.2*, July 2009.

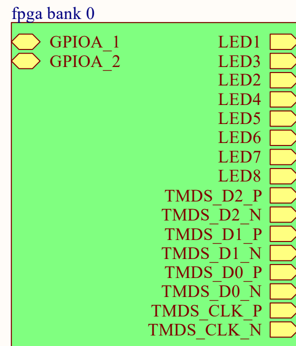
[40] Xilinx, "Ultrascale architecture and product data sheet: Overview," *Xilinx Product Specification DS890, Version 3.5*, August 2018.

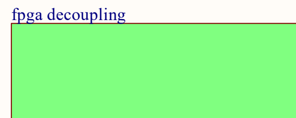
Appendix A

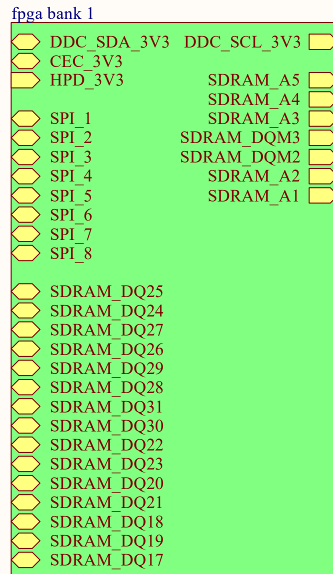
Hardware Design Schematics

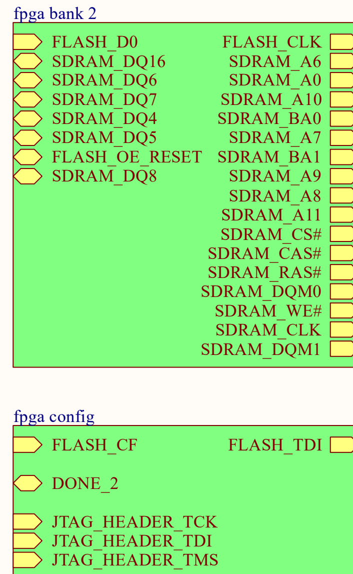
1 2 3 4

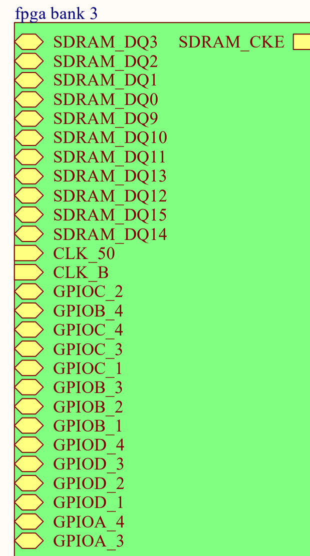
A B C D

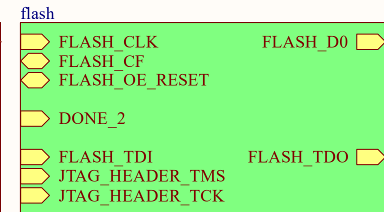


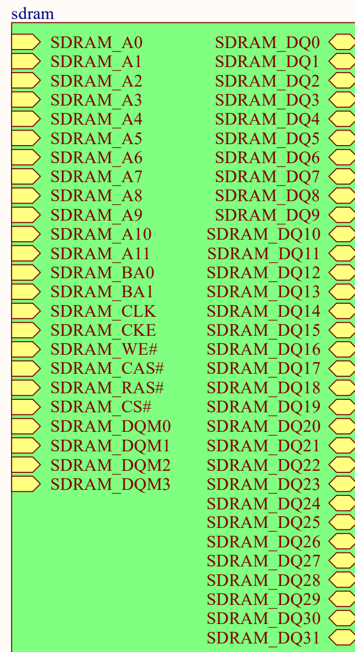


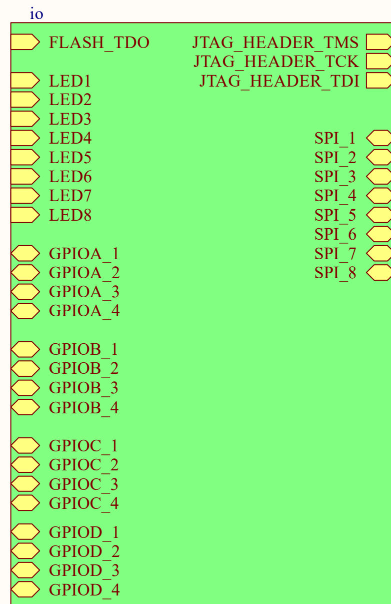


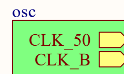


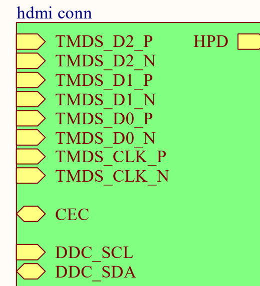


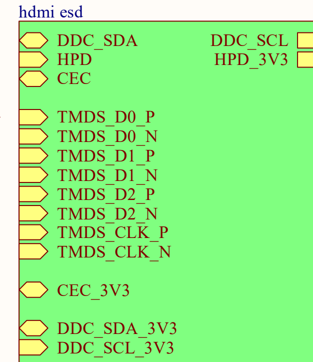








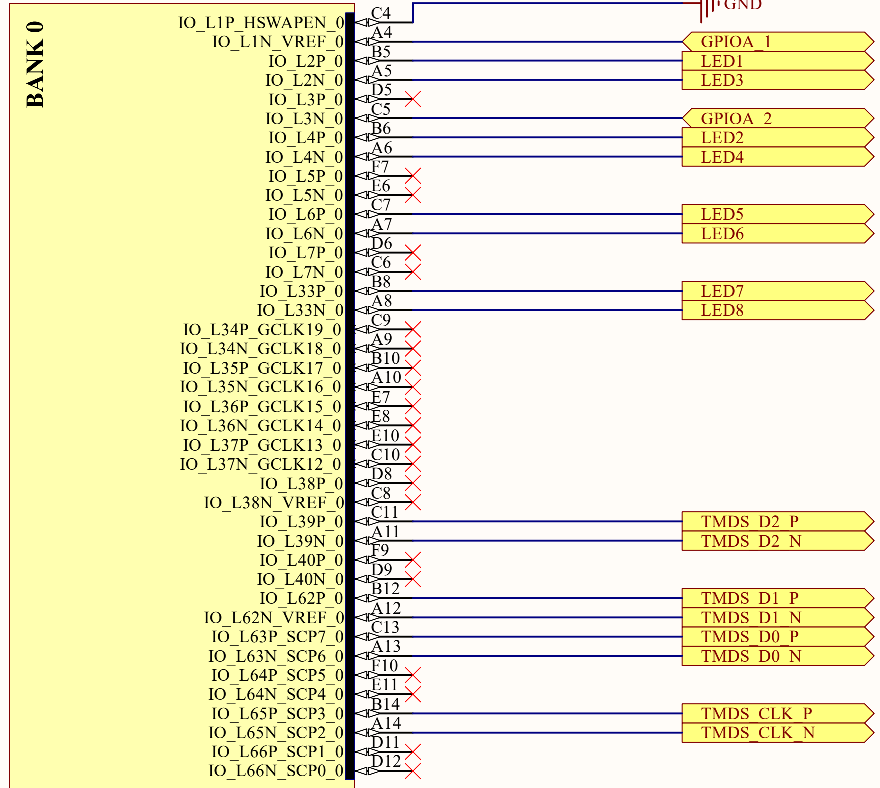




µGE verification PCB	
Title: µGE Schematic Summary	
Size: A4	Version: 1
Date: 6/24/2018	Revision: 0
Sheet 1 of 16	Engineer: Francois Tolmie

1 2 3 4

A B C D



µGE verification PCB	
Title: FPGA Bank 0	
Size: A4	Version: 1
Date: 6/24/2018	Revision: 0
Sheet 2 of 16	Engineer: Francois Tolmie

1 2 3 4

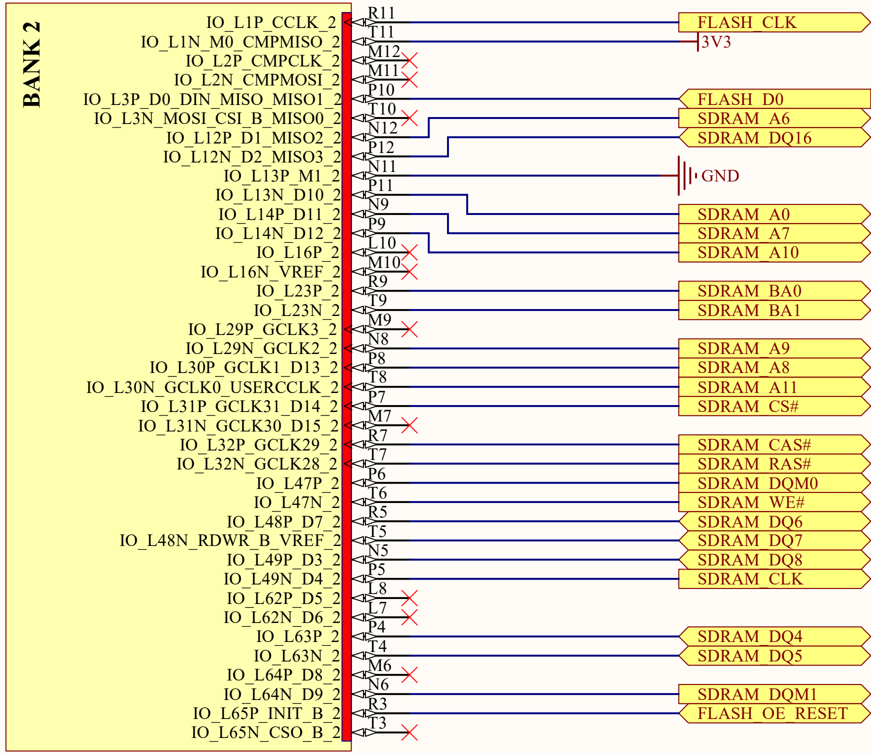
U1B

BANK 1

Pin name	Pin	Net
IO_L1P_A25_1	E13	
IO_L1N_A24_VREF_1	E12	
IO_L29P_A23_M1A13_1	B15	CEC_3V3
IO_L29N_A22_M1A14_1	B16	DDC_SCL_3V3
IO_L30P_A21_M1RESET_1	F12	
IO_L30N_A20_M1A11_1	G11	
IO_L31P_A19_M1CKE_1	D14	SPI_8
IO_L31N_A18_M1A12_1	D16	SPI_7
IO_L32P_A17_M1A8_1	F13	
IO_L32N_A16_M1A9_1	F14	SPI_2
IO_L33P_A15_M1A10_1	C15	DDC_SDA_3V3
IO_L33N_A14_M1A4_1	C16	HPD_3V3
IO_L34P_A13_M1WE_1	E15	SPI_6
IO_L34N_A12_M1BA2_1	E16	SPI_5
IO_L35P_A11_M1A7_1	F15	SPI_4
IO_L35N_A10_M1A2_1	F16	SPI_3
IO_L36P_A9_M1BA0_1	G14	
IO_L36N_A8_M1BA1_1	G16	SPI_1
IO_L37P_A7_M1A0_1	H15	SDRAM_DQ25
IO_L37N_A6_M1A1_1	H16	SDRAM_DQ24
IO_L38P_A5_M1CLK_1	G12	
IO_L38N_A4_M1CLKN_1	H11	
IO_L39P_M1A3_1	H13	
IO_L39N_M1ODT_1	H14	
IO_L40P_GCLK11_M1A5_1	J11	
IO_L40N_GCLK10_M1A6_1	J12	
IO_L41P_GCLK9_IRDY1_M1RASN_1	J13	
IO_L41N_GCLK8_M1CASN_1	K14	
IO_L42P_GCLK7_M1UDM_1	K12	
IO_L42N_GCLK6_TRDY1_M1LDM_1	K11	
IO_L43P_GCLK5_M1DQ4_1	J14	SDRAM_DQ27
IO_L43N_GCLK4_M1DQ5_1	J16	SDRAM_DQ26
IO_L44P_A3_M1DQ6_1	K15	SDRAM_DQ29
IO_L44N_A2_M1DQ7_1	K16	SDRAM_DQ28
IO_L45P_A1_M1LDQS_1	N14	SDRAM_A5
IO_L45N_A0_M1LDQSN_1	N16	SDRAM_A4
IO_L46P_FCS_B_M1DQ2_1	M15	SDRAM_A3
IO_L46N_FOE_B_M1DQ3_1	M16	SDRAM_DQM3
IO_L47P_FWE_B_M1DQ0_1	L14	SDRAM_DQ31
IO_L47N_LDC_M1DQ1_1	L16	SDRAM_DQ30
IO_L48P_HDC_M1DQ8_1	P15	SDRAM_DQ22
IO_L48N_M1DQ9_1	P16	SDRAM_DQ23
IO_L49P_M1DQ10_1	R15	SDRAM_DQ20
IO_L49N_M1DQ11_1	R16	SDRAM_DQ21
IO_L50P_M1UDQS_1	R14	SDRAM_DQ18
IO_L50N_M1UDQSN_1	T15	SDRAM_DQ19
IO_L51P_M1DQ12_1	T14	SDRAM_DQ17
IO_L51N_M1DQ13_1	T13	SDRAM_DQM2
IO_L52P_M1DQ14_1	R12	SDRAM_A2
IO_L52N_M1DQ15_1	T12	SDRAM_A1
IO_L53P_1	L12	
IO_L53N_VREF_1	L13	
IO_L74P_AWAKE_1	M13	
IO_L74N_DOUT_BUSY_1	M14	

XC6SLX25-3FTG256C

µGE verification PCB	
Title: FPGA Bank 1	
Size: A4	Version: 1
Date: 6/24/2018	Revision: 0
Sheet 3 of 16	Engineer: Francois Tolmie



µGE verification PCB	
Title: FPGA Bank 2	
Size: A4	Version: 1
Date: 6/24/2018	Revision: 0
Sheet 4 of 16	Engineer: Francois Tolmie

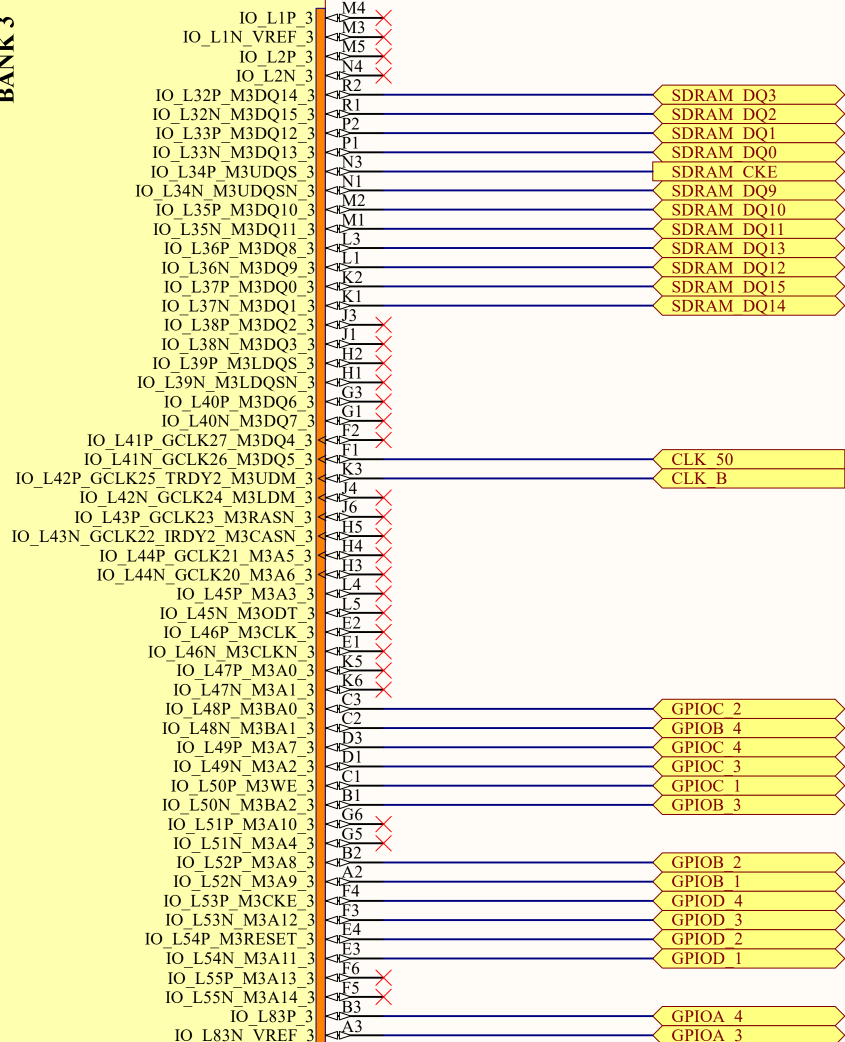
1	2	3	4

A

B

C

D



A

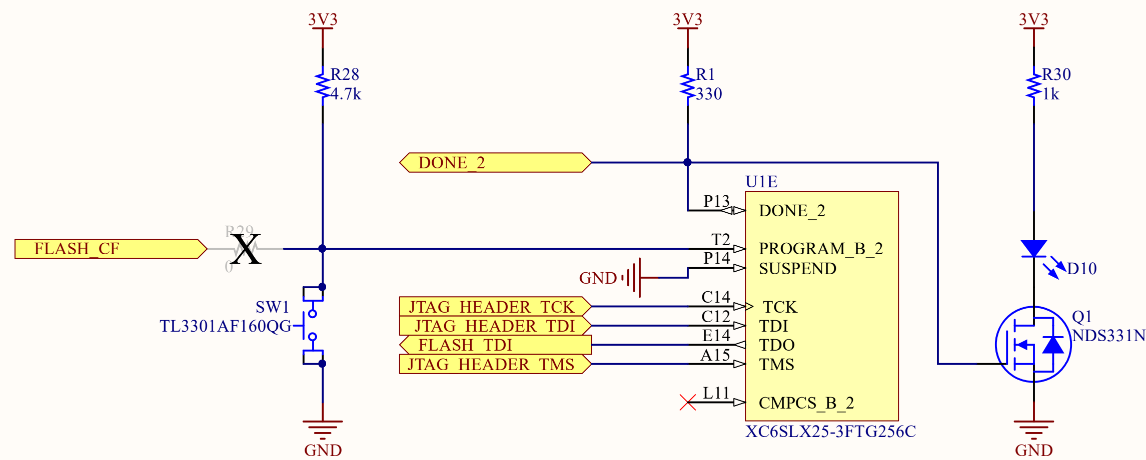
B

C

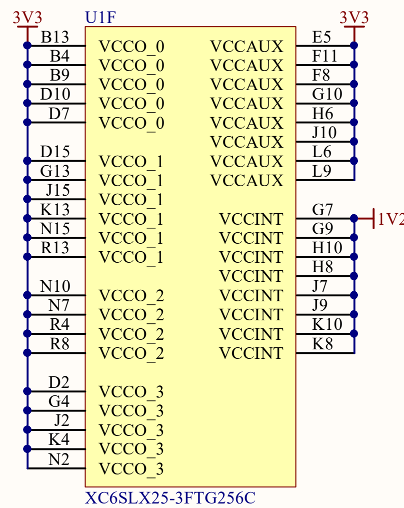
D

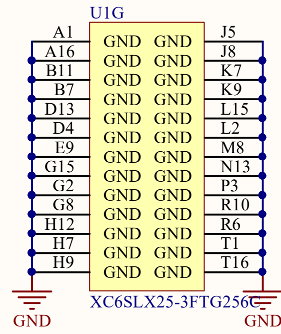
µGE verification PCB	
Title: FPGA Bank 3	
Size: A4	Version: 1
Date: 6/24/2018	Revision: 0
Sheet 5 of 16	Engineer: Francois Tolmie

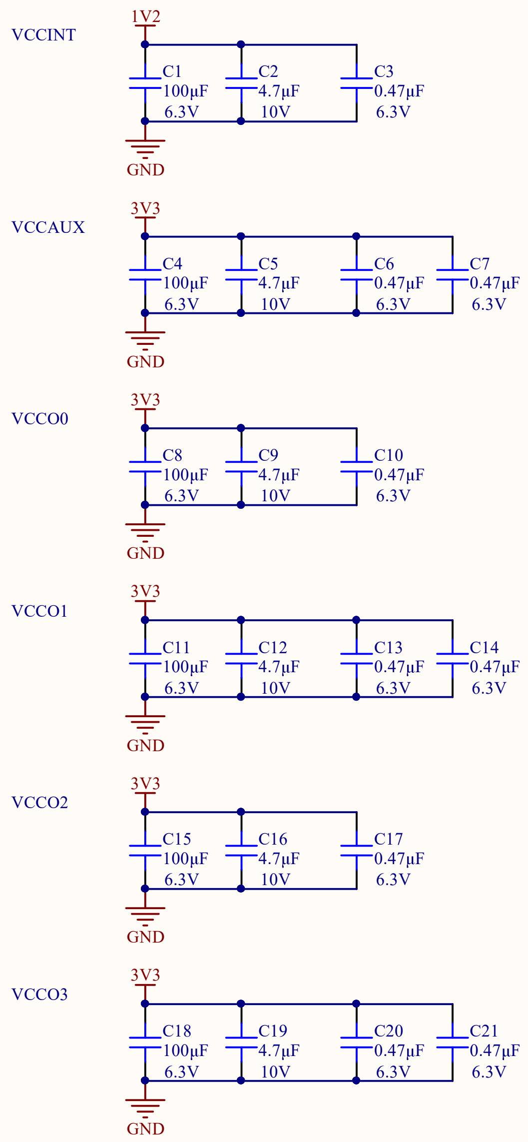
1	2	3	4



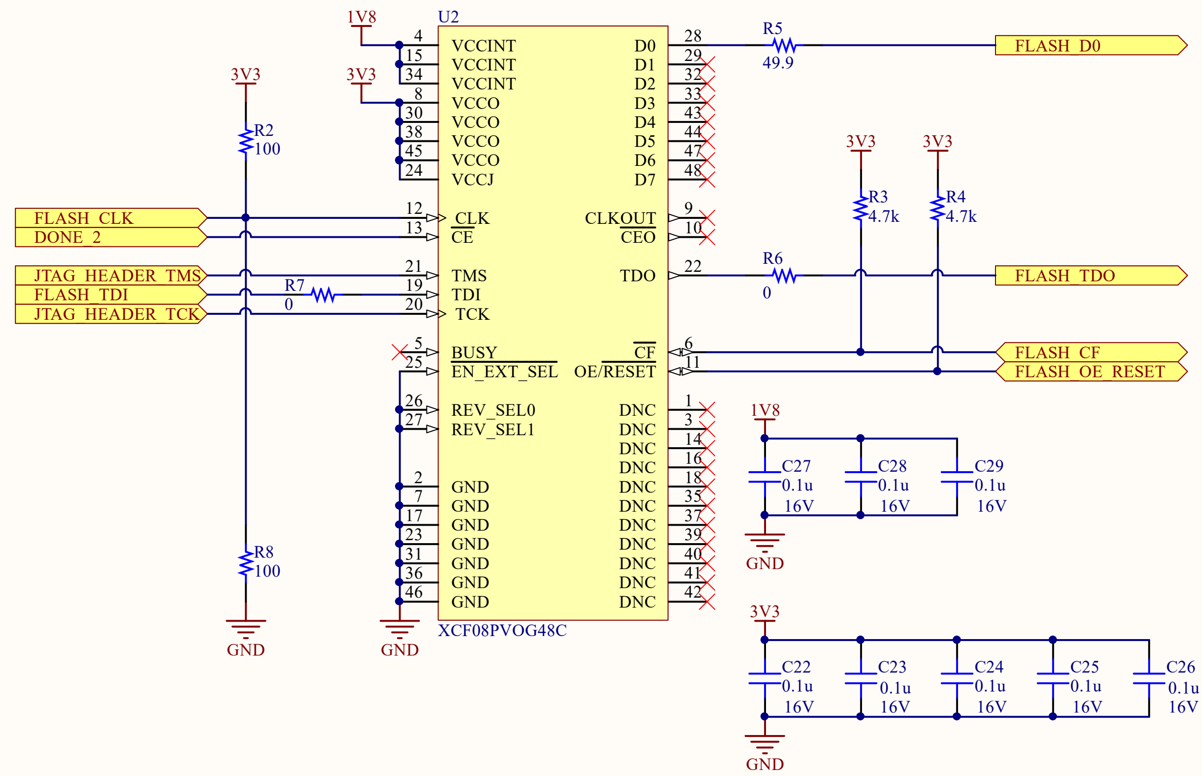
µGE verification PCB	
Title: FPGA Config	
Size: A4	Version: 1
Date: 6/24/2018	Revision: 0
Sheet 6 of 16	Engineer: Francois Tolmie



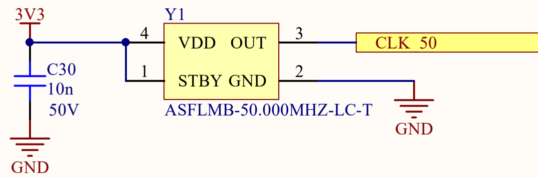


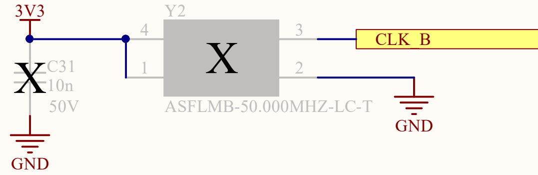


µGE verification PCB	
Title: FPGA Decoupling	
Size: A4	Version: 1
Date: 6/24/2018	Revision: 0
Sheet 7 of 16	Engineer: Francois Tolmie

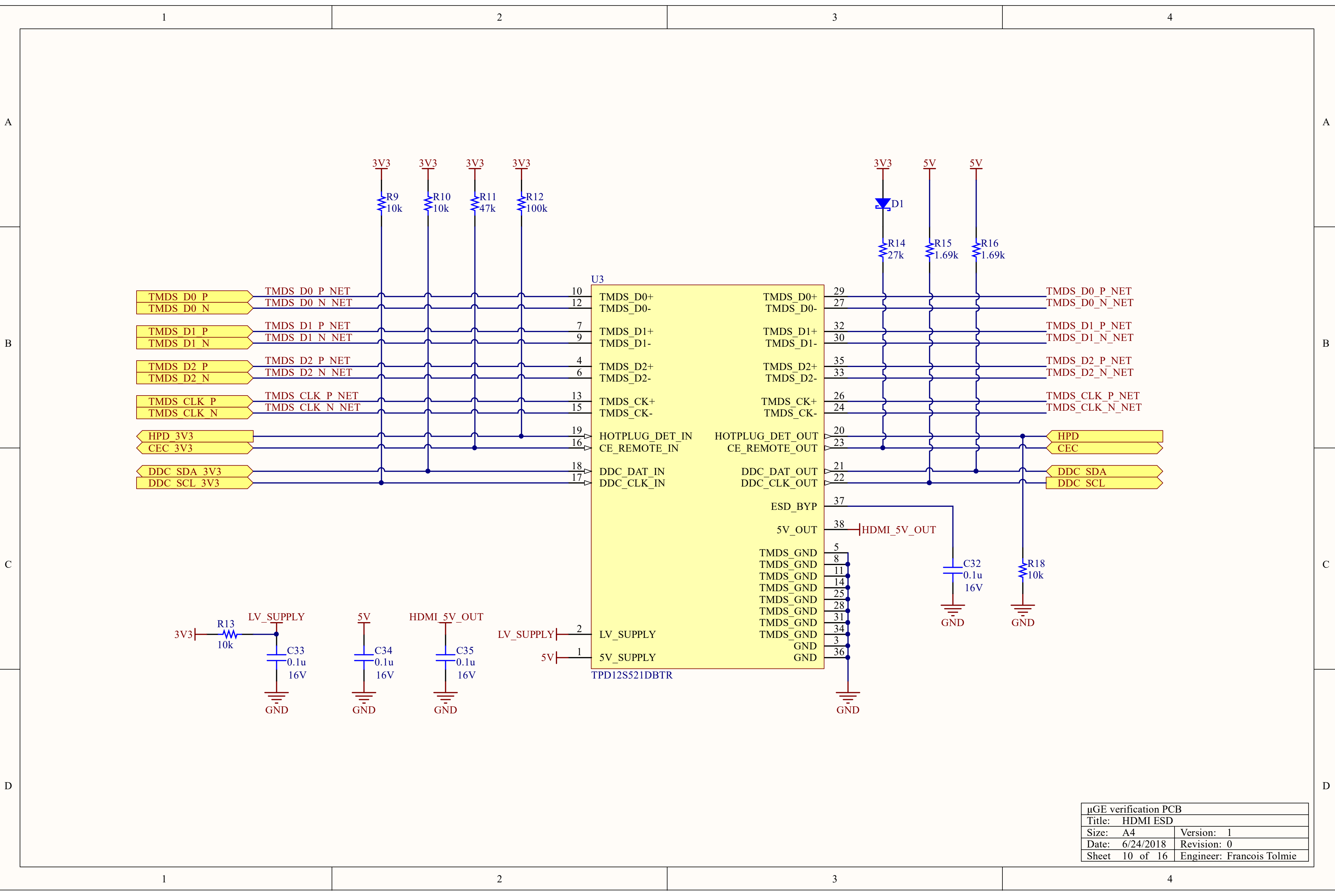


µGE verification PCB	
Title: Flash	
Size: A4	Version: 1
Date: 6/24/2018	Revision: 0
Sheet 8 of 16	Engineer: Francois Tolmie

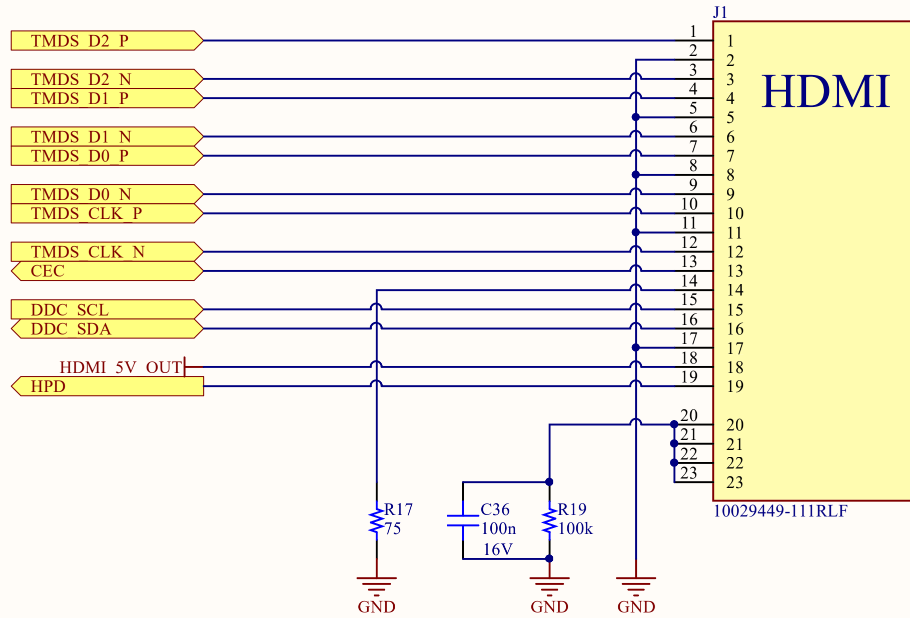




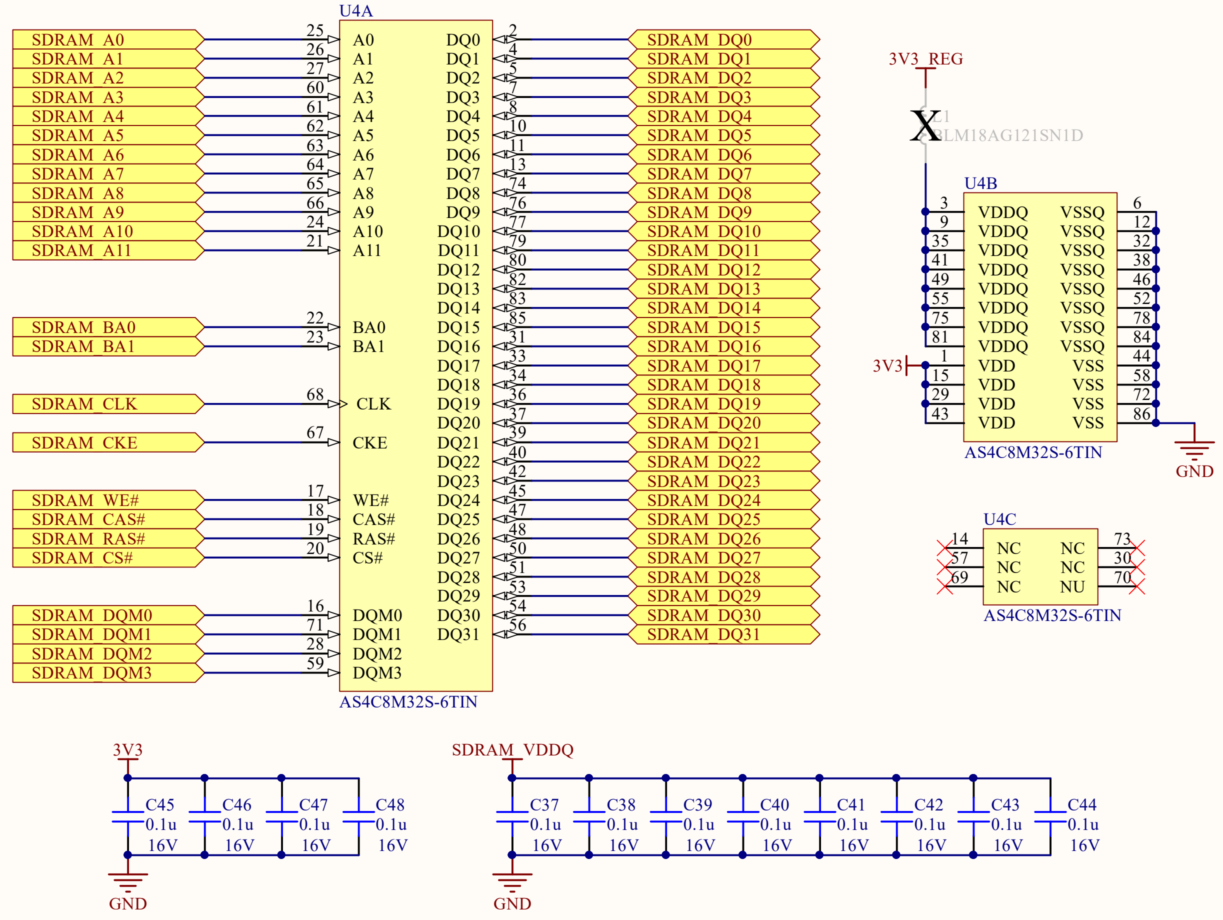
µGE verification PCB	
Title: Osc	
Size: A4	Version: 1
Date: 6/24/2018	Revision: 0
Sheet 9 of 16	Engineer: Francois Tolmie



3V3
R9
10k
R10
10k
R11
47k
R12
100k
D1
R14
27k
5V
R15
1.69k
R16
1.69k
U3
TMDS_D0_P
TMDS_D0_N
TMDS_D1_P
TMDS_D1_N
TMDS_D2_P
TMDS_D2_N
TMDS_CLK_P
TMDS_CLK_N
HPD_3V3
CEC_3V3
DDC_SDA_3V3
DDC_SCL_3V3
TMDS_D0_P_NET
TMDS_D0_N_NET
TMDS_D1_P_NET
TMDS_D1_N_NET
TMDS_D2_P_NET
TMDS_D2_N_NET
TMDS_CLK_P_NET
TMDS_CLK_N_NET
TMDS_D0+
TMDS_D0-
TMDS_D1+
TMDS_D1-
TMDS_D2+
TMDS_D2-
TMDS_CK+
TMDS_CK-
HOTPLUG_DET_IN
CE_REMOTE_IN
DDC_DAT_IN
DDC_CLK_IN
HOTPLUG_DET_OUT
CE_REMOTE_OUT
DDC_DAT_OUT
DDC_CLK_OUT
ESD_BYP
5V_OUT
TMDS_GND
GND
LV_SUPPLY
5V_SUPPLY
TPD12S521DBTR
HDMI_5V_OUT
HPD
CEC
DDC_SDA
DDC_SCL
C32
0.1u
16V
R18
10k
GND
R13
10k
C33
0.1u
16V
C34
0.1u
16V
C35
0.1u
16V
µGE verification PCB
Title: HDMI ESD
Size: A4
Version: 1
Date: 6/24/2018
Revision: 0
Sheet 10 of 16
Engineer: Francois Tolmie

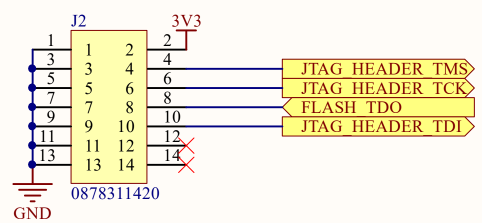


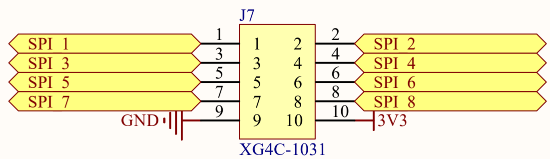
µGE verification PCB	
Title: HDMI Conn	
Size: A4	Version: 1
Date: 6/24/2018	Revision: 0
Sheet 11 of 16	Engineer: Francois Tolmie

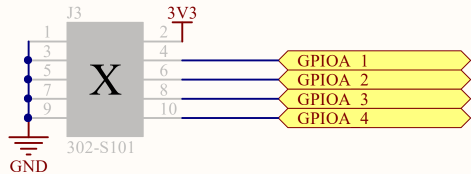


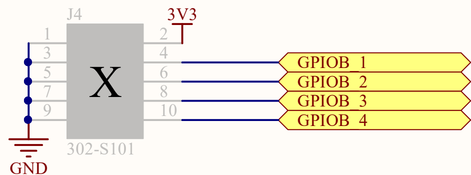
µGE verification PCB	
Title: SDRAM	
Size: A4	Version: 1
Date: 6/24/2018	Revision: 0
Sheet 12 of 16	Engineer: Francois Tolmie

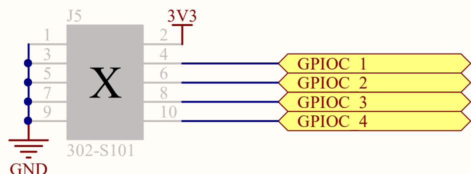
1	2	3	4

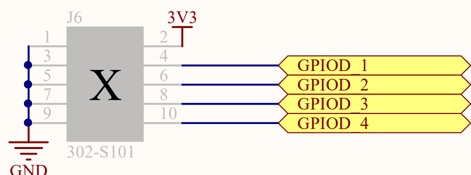


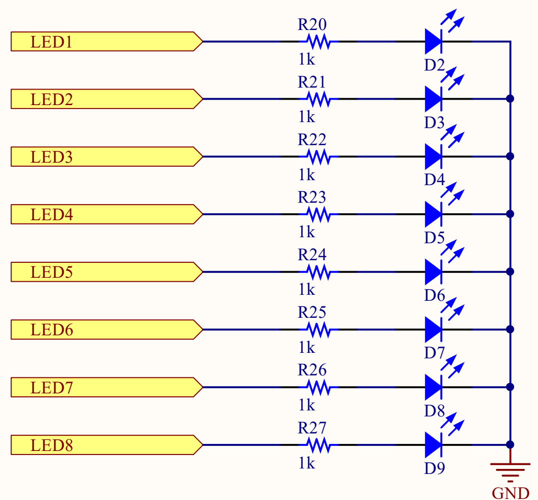


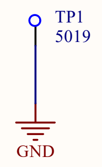






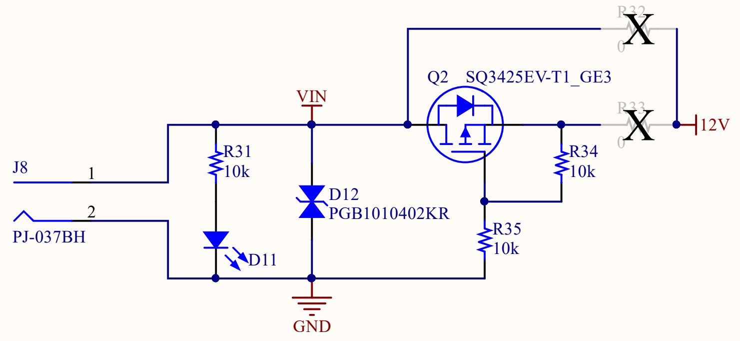




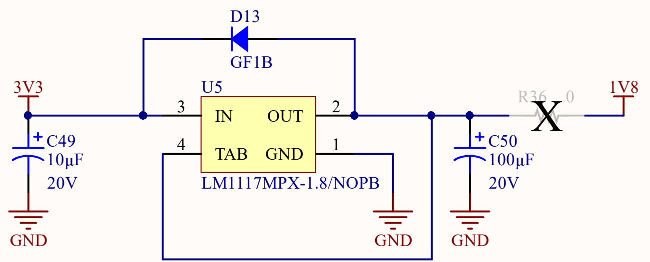


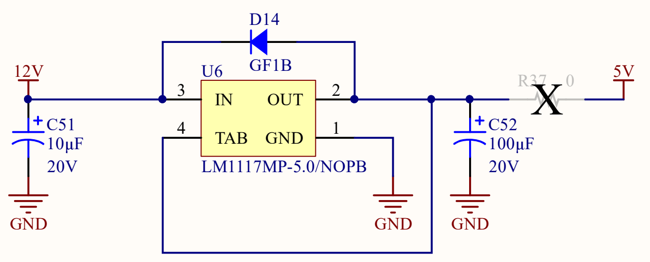
µGE verification PCB	
Title: IO	
Size: A4	Version: 1
Date: 6/24/2018	Revision: 0
Sheet 13 of 16	Engineer: Francois Tolmie

1	2	3	4

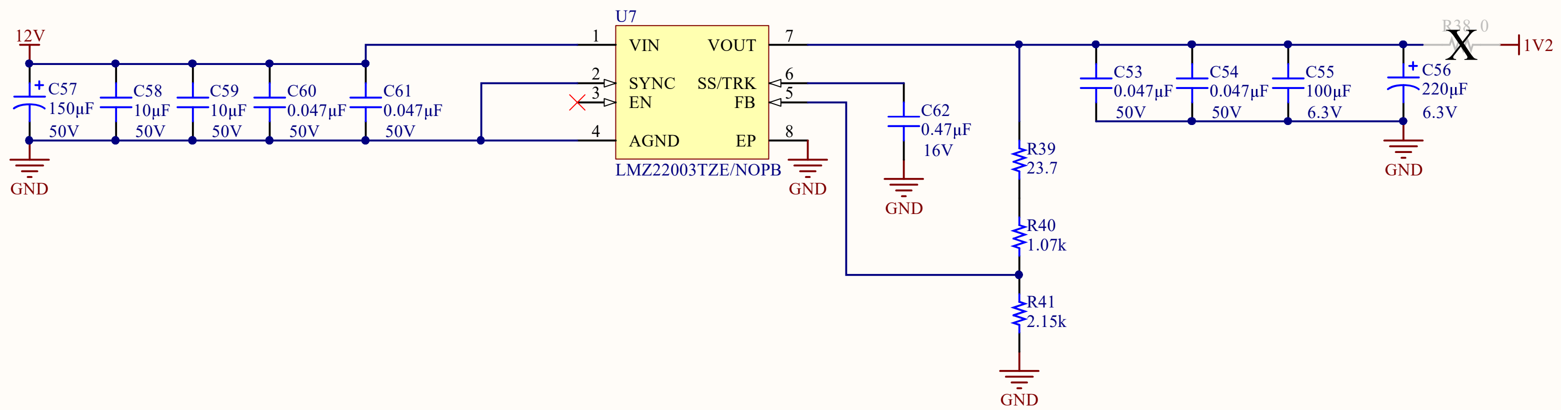


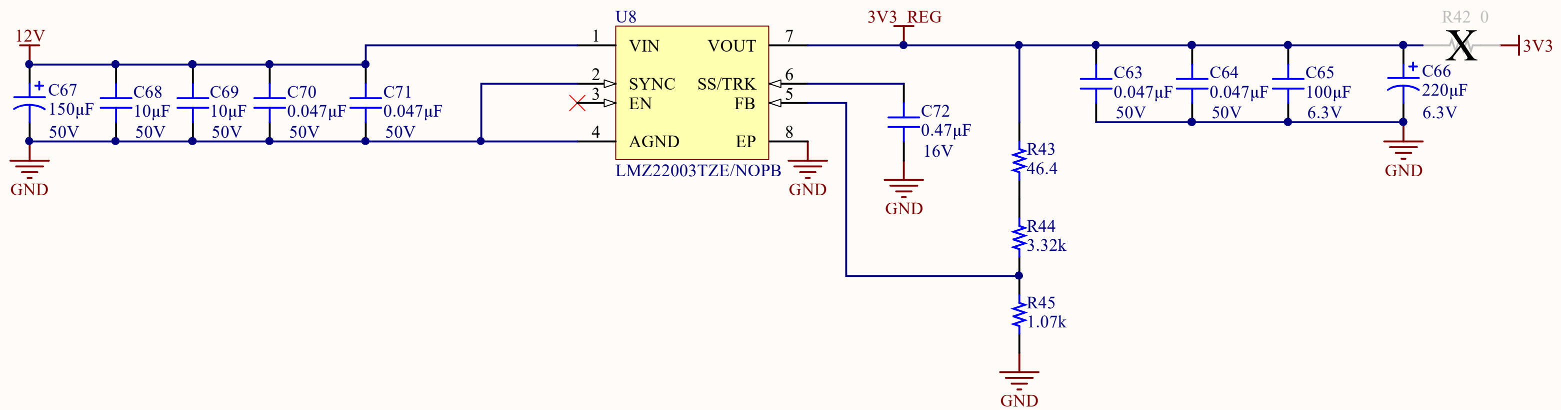
µGE verification PCB	
Title: Power In	
Size: A4	Version: 1
Date: 6/24/2018	Revision: 0
Sheet 14 of 16	Engineer: Francois Tolmie





µGE verification PCB	
Title: LDO	
Size: A4	Version: 1
Date: 6/24/2018	Revision: 0
Sheet 15 of 16	Engineer: Francois Tolmie





µGE verification PCB	
Title: DCDC	
Size: A4	Version: 1
Date: 6/24/2018	Revision: 0
Sheet 16 of 16	Engineer: Francois Tolmie

Appendix B

PCB Layout, Assembly and Testing

This appendix contains the following documents of the PCB layout, assembly and testing:

- PCB Stackup
- PCB Bill of Materials
- PCB Assembly Drawings
- PCB Layout Plots
- PCB Fabrication Quote
- PCB Assembly Sales Order
- PCB Bring-Up Plan

Polar

Layer	Stack up	Supplier	Supplier Description	Description	Type	Tg	εr	Base Thickness	Processed Thickness
		Electra Polymers		Liquid PhotoImageable Mask	SolderMask		4.000		
1		Circuitfoil		Copper Foil	Copper			0.018	0.038
		Shanghai Nanya	NY2150	PrePreg 2116	Dielectric	150.000	4.200	0.120	0.118
2								0.035	0.035
		Shanghai Nanya	NY2150	NY2150 Core	FR4	150.000	4.200	0.510	0.510
3								0.035	0.035
	1.59	Shanghai Nanya	NY2150	PrePreg 2116	Dielectric	150.000	4.200	0.120	0.117
4								0.035	0.035
		Shanghai Nanya	NY2150	NY2150 Core	FR4	150.000	4.200	0.510	0.510
5								0.035	0.035
		Shanghai Nanya	NY2150	PrePreg 2116	Dielectric	150.000	4.200	0.120	0.118
6		Circuitfoil		Copper Foil	Copper			0.018	0.038
		Electra Polymers		Liquid PhotoImageable Mask	SolderMask		4.000		

Copper Thickness = 0.216 | Dielectric Thickness = 1.373 | Solder Mask Thickness = 0.050 | Stack Up Thickness = 1.589 | Stack Up Thickness with Soldermask = 1.639 |
Stack Up Cost = 46.00 |

Structure Image	Impedance ID	Structure Name	Impedance Signal Layer	Substrate 1 Height (H1)	Lower Trace Width (W1)	Trace Separation (S1)	Lower Ground Strip Width (G1)	Ground Strip Separation (D1)	Trace Thickness (T1)	Calculated Impedance	Target Impedance	CI Notes-1
	1	Coated Microstrip 1B	1	0.118	0.200	0.000	0.000	0.000	0.038	48.360	50.000	
	2	Coated Microstrip 1B	6	0.118	0.200	0.000	0.000	0.000	0.038	48.360	50.000	

Drill Image	1st Layer	2nd Layer	Column Position
	1	6	1

Notes

StackName: Mlb6_FrancoisT_UCT_Enquiry	Version:	Revision:	Modification:	Date of Revision:	Editor	Page 1/1
Date: 2017/08/14	Associated Documents: e-Mail enquiry from Francois Tolmie at UCT for 6 layer having impedance control on Signal layers - details in e-mail.					
Author: Marc N						
Department: Tech						
Site: Diep River						

Copyright ©
www.polarinstruments.com

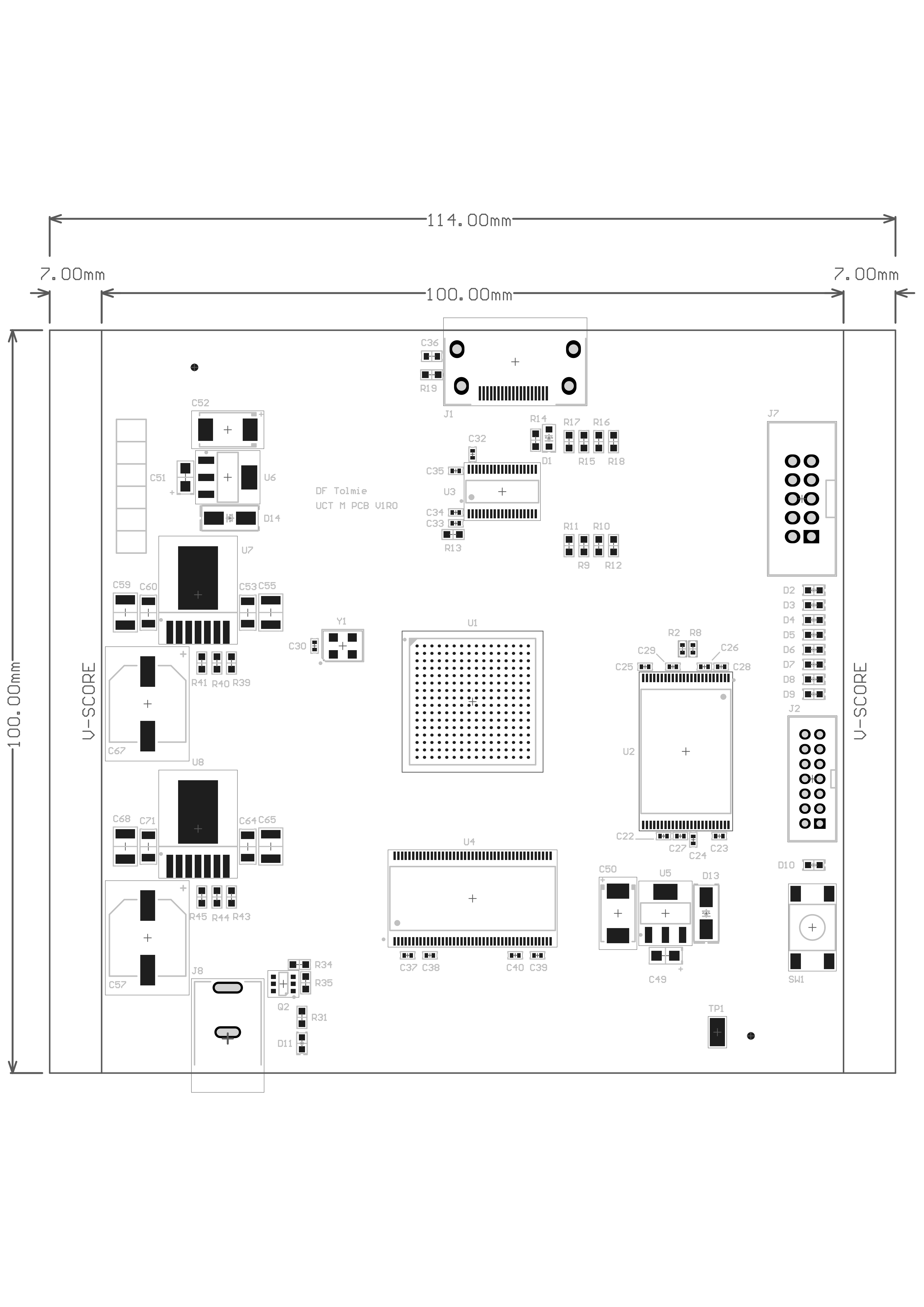
µGE V1R0 (19920420UGE0100) Bill of Materials **BOM Version: 1**

Quantity	Designator	Description	Manufacturer Part Number	Digi-Key Part Number
6	C1, C4, C8, C11, C15, C18	CAP CER 100UF 6.3V X5R 1210	GRM32ER60J107ME20L	490-3390-1-ND
6	C2, C5, C9, C12, C16, C19	CAP CER 4.7UF 10V X7R 0805	GRM21BR71A475KA73	490-6479-1-ND
9	C3, C6, C7, C10, C13, C14, C17, C20, C21	CAP CER 0.47UF 6.3V X5R 0402	GRM152R60J474ME15D	490-10007-1-ND
24	C22, C23, C24, C25, C26, C27, C28, C29, C32, C33, C34, C35, C37, C38, C39, C40, C41, C42, C43, C44, C45, C46, C47, C48	CAP CER 0.1UF 16V X7R 0402	GRM155R71C104KA88J	490-6328-1-ND
1	C30	CAP CER 10000PF 50V X7R 0402	GRM155R71H103KA88D	490-4516-1-ND
1	C36	CAP CER 0.1UF 16V X7R 0603	GRM188R71C104KA01D	490-1532-1-ND
2	C49, C51	CAP TANT 10UF 20V 20% 1206	F931D106MAA	478-8275-1-ND
2	C50, C52	CAP TANT 100UF 20V 10% 2917	TAJD107K020RNJ	478-1724-1-ND
8	C53, C54, C60, C61, C63, C64, C70, C71	CAP CER 0.047UF 50V X7R 1206	CC1206KRX7R9BB473	311-1178-1-ND
2	C55, C65	CAP CER 100UF 6.3V X5R 1210	C3225X5R0J107M	445-1437-1-ND
2	C56, C66	CAP ALUM POLY 220UF 20% 6.3V SMD	EEF-UE0J221LR	PCE4263CT-ND
2	C57, C67	CAP ALUM 150UF 20% 50V SMD	EEE-FK1H151P	PCE3810CT-ND
4	C58, C59, C68, C69	CAP CER 10UF 50V X5R 1210	UMK325BJ106MM-T	587-2225-1-ND
2	C62, C72	CAP CER 0.47UF 16V X7R 0805	0805YC474KAT2A	478-1403-1-ND
1	D1	DIODE SCHOTTKY 30V 800MA USC	CUS08F30,H3F	CUS08F30H3FCT-ND
10	D2, D3, D4, D5, D6, D7, D8, D9, D10, D11	LED GREEN CLEAR 0603 SMD	150060VS75000	732-4980-1-ND
1	D12	TVS DIODE 12VWM 250VC 0402	PGB1010402KR	F2862CT-ND
2	D13, D14	DIODE GEN PURP 100V 1A SMA	GF1B	GF1BFSCT-ND
1	J1	CONN HDMI RECPT 19POS SMT R/A	10029449-111RLF	609-4614-1-ND
1	J2	CONN HEADER 14POS 2MM VERT GOLD	0878311420	WM17469-ND
1	J7	CONN PLUG 10POS 3A 300V STRT DIP	XG4C-1031	OR896-ND
1	J8	CONN PWR JACK 2.5X5.5MM SOLDER	PJ-037BH	CP-037BH-ND
1	Q1	MOSFET N-CH 20V 1.3A SSOT3	NDS331N	NDS331NCT-ND
1	Q2	MOSFET P-CH 20V 7.4A SOT23-3	SQ3425EV-T1_GE3	SQ3425EV-T1_GE3CT-ND
1	R1	RES SMD 330 OHM 1% 1/10W 0603	RC0603FR-07330RL	311-330HRCT-ND
2	R2, R8	RES SMD 100 OHM 1% 1/5W 0402	RCA0402100RFKEDHP	541-3239-1-ND
3	R3, R4, R28	RES SMD 4.7K OHM 1% 1/10W 0603	RC0603FR-074K7L	311-4.70KHRCT-ND
1	R5	RES SMD 49.9 OHM 1% 1/16W 0402	RC0402FR-0749R9L	311-49.9LRCT-ND
2	R6, R7	RES SMD 0 OHM JUMPER 1/16W 0402	RC0402JR-070RL	311-0.0JRCT-ND
7	R9, R10, R13, R18, R31, R34, R35	RES SMD 10K OHM 1% 1/10W 0603	RC0603FR-0710KL	311-10.0KHRCT-ND

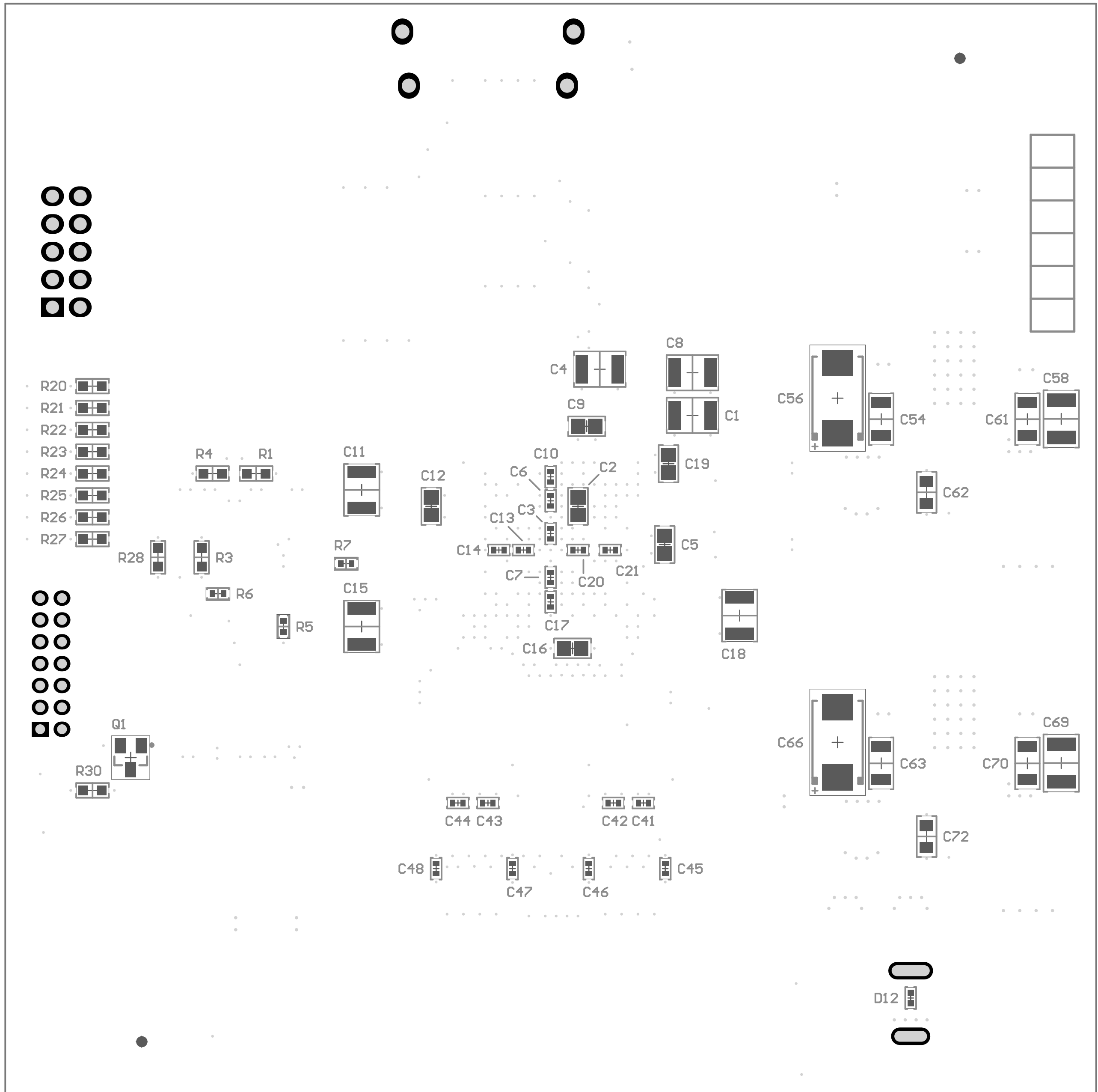
µGE V1R0 (19920420UGE0100) Bill of Materials **BOM Version: 1**

Quantity	Designator	Description	Manufacturer Part Number	Digi-Key Part Number
1	R11	RES SMD 47K OHM 1% 1/10W 0603	RC0603FR-0747KL	311-47.0KHRCT-ND
2	R12, R19	RES SMD 100K OHM 1% 1/10W 0603	RC0603FR-07100KL	311-100KHRCT-ND
1	R14	RES SMD 27K OHM 1% 1/10W 0603	RC0603FR-0727KL	311-27.0KHRCT-ND
2	R15, R16	RES SMD 1.69K OHM 1% 1/10W 0603	RC0603FR-071K69L	311-1.69KHRCT-ND
1	R17	RES SMD 75 OHM 1% 1/10W 0603	RC0603FR-0775RL	311-75.0HRCT-ND
9	R20, R21, R22, R23, R24, R25, R26, R27, R30	RES SMD 1K OHM 1% 1/10W 0603	RC0603FR-071KL	311-1.00KHRCT-ND
1	R39	RES SMD 23.7 OHM 1% 1/10W 0603	RC0603FR-0723R7L	311-23.7HRCT-ND
2	R40, R45	RES SMD 1.07K OHM 1% 1/10W 0603	RC0603FR-071K07L	311-1.07KHRCT-ND
1	R41	RES SMD 2.15K OHM 1% 1/10W 0603	RC0603FR-072K15L	311-2.15KHRCT-ND
1	R43	RES SMD 46.4 OHM 1% 1/10W 0603	RC0603FR-0746R4L	311-46.4HRCT-ND
1	R44	RES SMD 3.32K OHM 1% 1/10W 0603	RC0603FR-073K32L	311-3.32KHRCT-ND
1	SW1	SWITCH TACTILE SPST-NO 0.05A 12V	TL3301AF160QG	EG2526CT-ND
1	TP1	PC TEST POINT MINI SMD	5019	36-5019CT-ND
1	U1	IC FPGA 186 I/O 256FTBGA	XC6SLX25-3FTG256C	122-1956-ND
1	U2	IC PROM SRL 1.8V 8M GATE 48TSOP	XCF08PVOG48C	122-1454-ND
1	U3	IC HDMI RX PORT PROT 38-TSSOP	TPD12S521DBTR	296-24020-1-ND
1	U4	IC SDRAM 128MBIT 166MHZ 86TSOP	AS4C8M32S-6TIN	1450-1337-ND
1	U5	IC REG LINEAR 1.8V 800MA SOT223	LM1117MPX-1.8/NOPB	LM1117MPX-1.8/NOPBCT-ND
1	U6	IC REG LINEAR 5V 800MA SOT223	LM1117MP-5.0/NOPB	LM1117MP-5.0/NOPBCT-ND
2	U7, U8	IC BUCK SYNC 20V 3A TO-PMOD-7	LMZ22003TZ/NOPB	LMZ22003TZ/NOPB-ND
1	Y1	OSC MEMS 50.000MHZ CMOS SMD	ASFLMB-50.000MHZ-LC-T	535-11779-1-ND

Do Not Install
R32, R33, R36, R37, R38,R42, L1, C31, Y2, J3, J4, J5, J6, R29



114.00mm
7.00mm
7.00mm
100.00mm
100.00mm
V-SCORE
V-SCORE
DF Tolmie
UCT M PCB V1R0

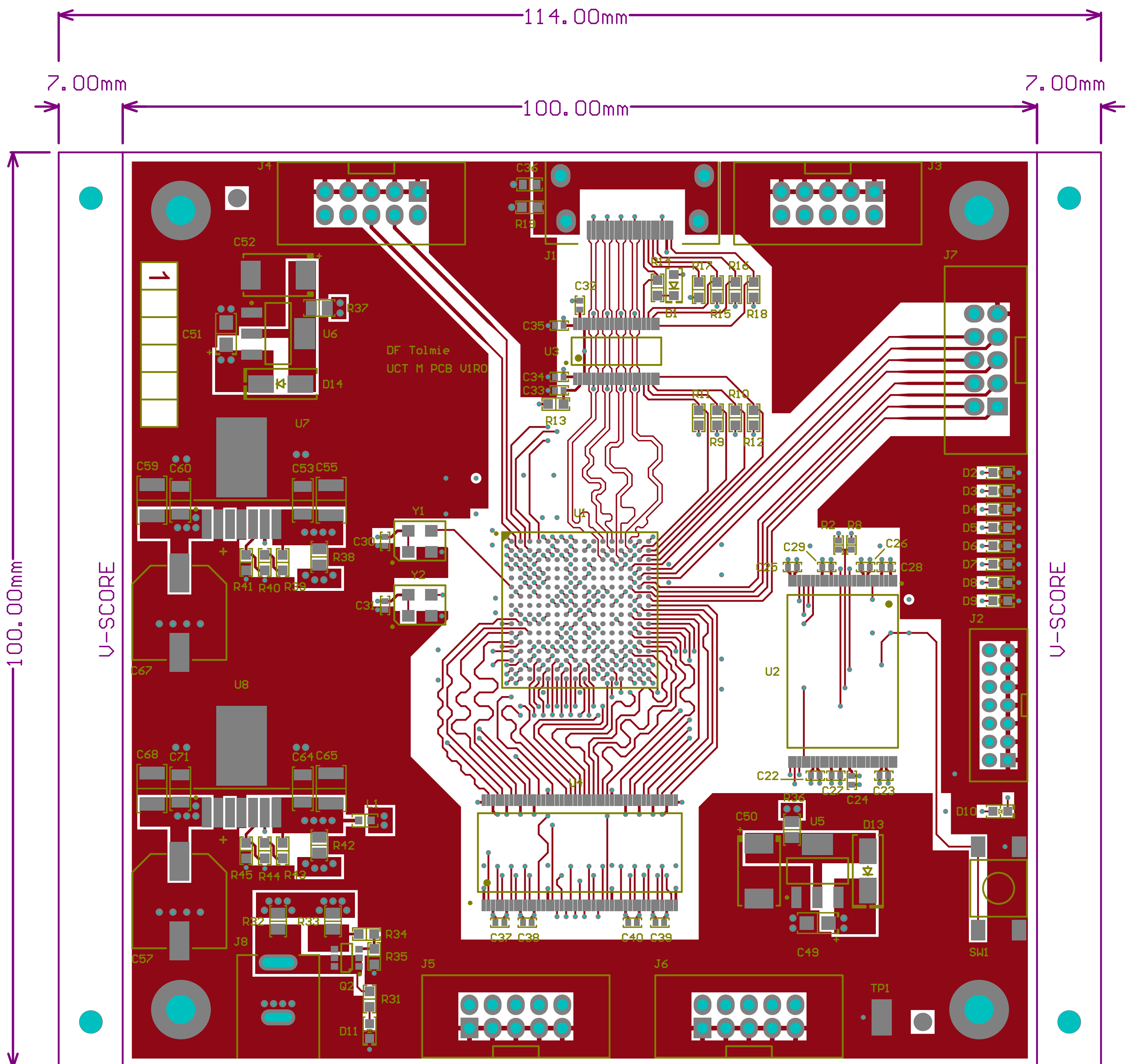


C4
C8
C9
C1
C56
C54
C58
C61
R20
R21
R22
R23
R24
R25
R26
R27
R4
R1
C11
C12
C10
C6
C2
C19
C62
C13
C3
C14
C20
C21
C5
R28
R3
R7
C7
R6
C15
C17
R5
C18
C16
C66
C63
C69
C70
Q1
R30
C44
C43
C42
C41
C72
C48
C47
C46
C45
D12

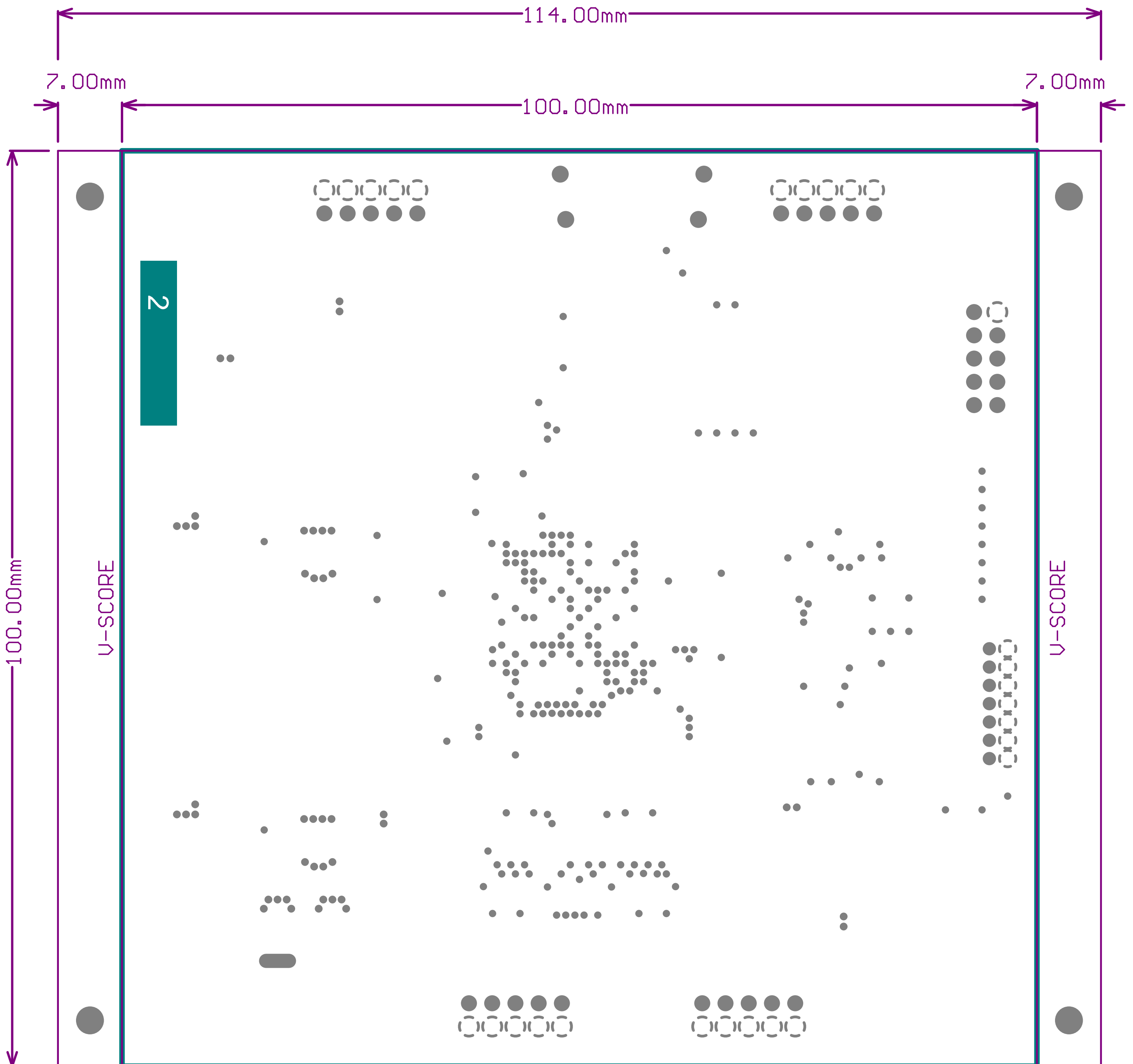
Top Overlay

Top Solder Mask

Layer 1: Signal Layer 1 (Top)



Layer 2: Ground Plane 1



Layer 3: Power Plane 1

114.00mm

7.00mm

7.00mm

100.00mm

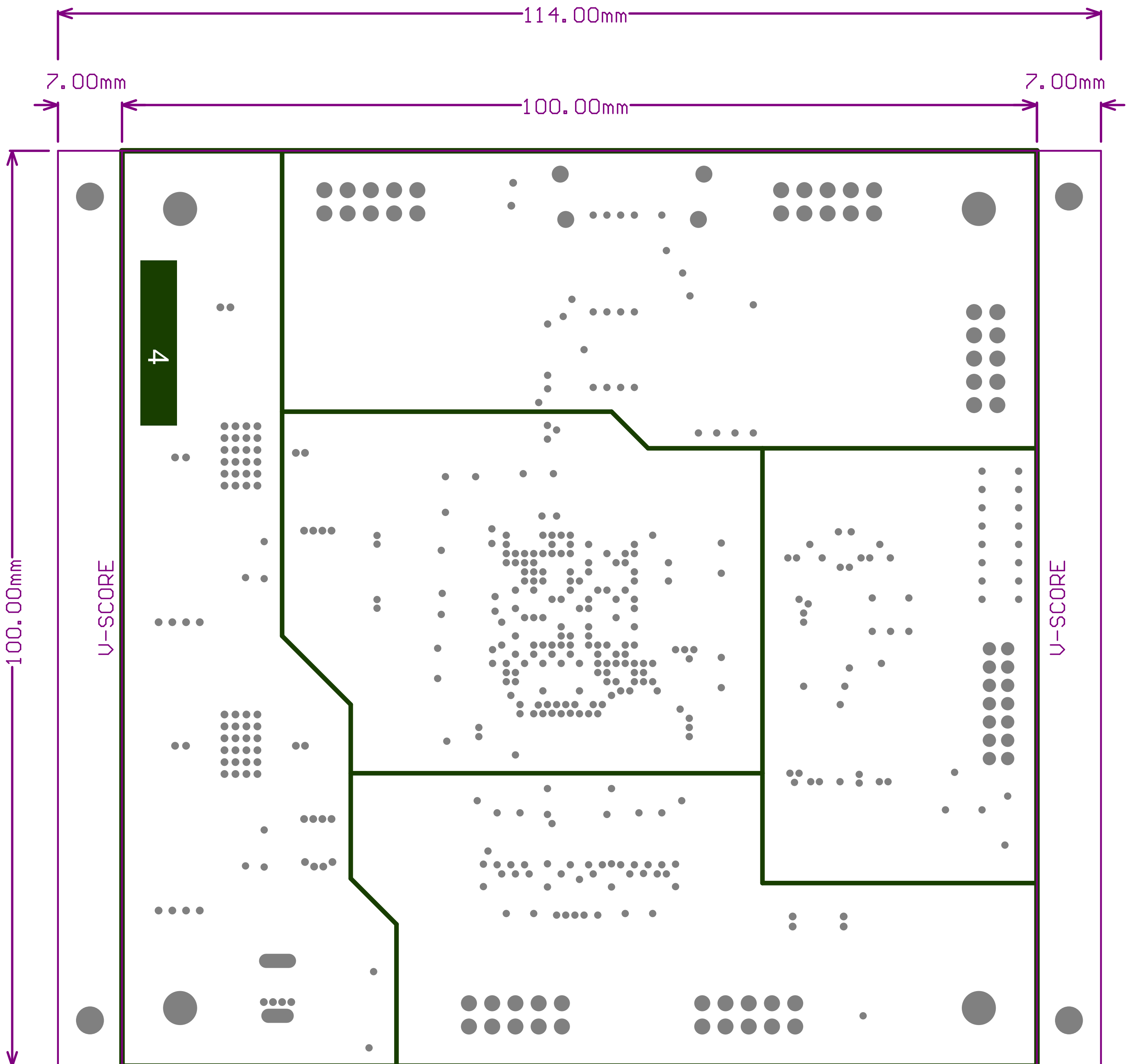
100.00mm

V-SCORE

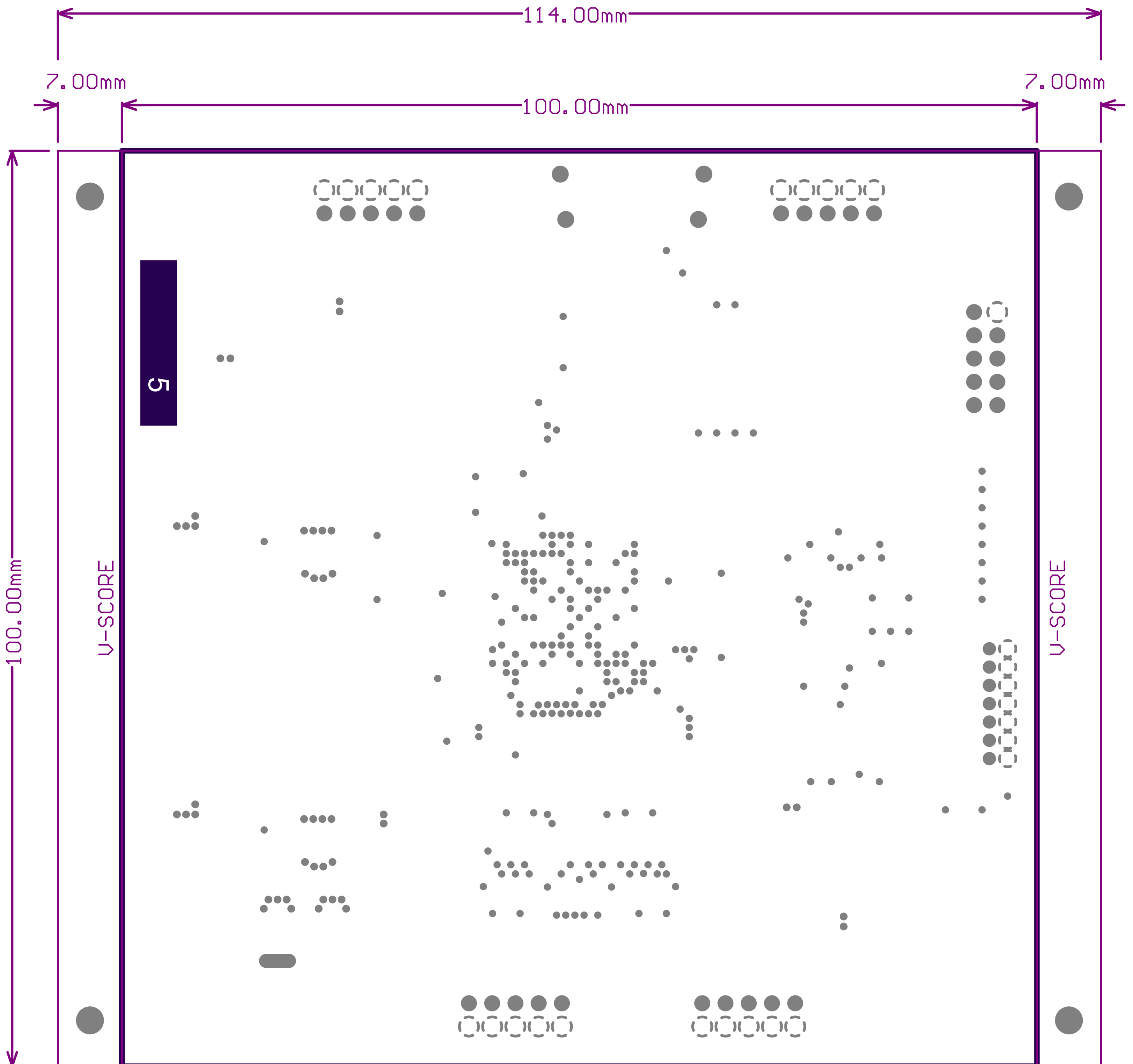
V-SCORE

3

Layer 4: Power Plane 2



Layer 5: Ground Plane 2



Bottom Overlay

Bottom Solder Mask

Layer 6: Signal Layer 2 (Bottom)

2 Estmil Road
Diep River
Cape Town 8000
Tel : (021) 712 5011
Fax : (021) 712 5798
P.O. Box 180
Plumstead 7801
South Africa

Vat Reg No. 4540118413 Reg. No. 1991/00696/07

Invoice to : UNIVERSITY OF CAPE TOWN
CREDITORS SECTION
FINANCE DEPARTMENT
PRIVATE BAG X3
RONDEBOSCH
7701

Deliver to :

28 September 2017

Your Ref :
Attention : FRANCOIS TOLMIE

QUOTATION FOR PRINTED CIRCUIT BOARDS (Quote #1048893)

Part Number:	MICRO_GE
Issue/Revision:	
Board Quantity:	1
Panel Quantity:	1
Boards per Panel:	1
Board Length x Width (mm):	100.0 x 100.0
Panel Length x Width (mm):	114.0 x 100.0
Panel Area (cm²):	114.0
V-Score:	Yes
Rout:	
Bare Board Test:	Flying Probe
Number of Layers:	6-Layer
Material Type:	FR4 (NY2150)
Board Thickness (mm):	1.639
Final Cu Thickness(µm):	35
Final Finish:	ENIG
No. of Gold Tabs:	0
No. of Holes/Board:	539
Component Side Legend:	White
Solder Side Legend:	White
Component Side Mask:	Green Matt
Solder Side Mask:	Green Matt
Peelable Ink/Conductive Ink:	No/No
Conformance Certificate:	No
Batch Coding:	No

BOARD COSTS

1 Board/s @	R 912.67 / board	
1 Panel/s @	R 912.67 / panel	
Total Board Price:		R 912.67

ONCE OFF SETUP

Origination:	R 1900.00
Bare Board Test Jig:	R 0.00
Flying Probe:	R 110.00
Paste Plot CS:	R 0.00
Paste Plot SS:	R 0.00
Paste Stencil CS:	R 0.00
Paste Stencil SS:	R 0.00

TESTING & SUNDRIES

Flying Probe @	R 91.27 / board	
Flying Probe Total:		R 91.27
Sundries:		R 0.00

TOTALS

Subtotal:	R 3013.94
VAT:	R 421.95
Invoice Total:	R 3435.89

Factory Completion Time:	13 working days
Terms of Payment:	30 days
	30 days net

Please Note:
* This quotation is valid for 4 Weeks.
* The price is subject to review should production data vary from information supplied for this quotation.
* Origination is a once off cost but becomes applicable for changes in production data.
* All sales are subject to our standard terms and conditions of sale.
* The Factory Completion Time is calculated from the date of order, provided finalised production files are received no later than 12h00 on that day.
* Please allow up to 2 days for delivery, unless arrangements have been made for collection.

Kind Regards
Anton Tait

Doc No. 7.2.1_1

Barracuda Holdings (Pty) Ltd

SALES ORDER

Unit 1, Bataleur Park
Olive Grove Industrial Estate
Ou Paardevlei Road

Telephone **+27(0)218513357**
Fax **+27(0)862106255**

PO Box 544
Somerset Mall
7137
Registration **1995/013557/07**
Vat Number **4730162346**

To:
UNIVERSITY OF CAPE TOWN
Upper Campus
University of Cape Town
Rondebosch
7700

Customer Vat No. 4540125707

Document No.	SO10101
	Page 1 of 1
Account	UCT000
Order Date	2017/11/09
Order No	1604367
Planned Due Date	2017/12/09

Item Code	Item Description	Quantity	Price (Ex)	Total (Excl)
1000	Hand PCB Assembly of Micro_ge PCB	1	1588.86	1 588.86
99-UCT-0008	PROTOTYPE LASER CUT STENCIL -MICRO GE	1	1622.00	1 622.00

MicroGE Test Board Bring-Up Plan

Visual Inspection	
PCB is clean (no flux residue, etc.)	✓
No components marked DNP are mounted (power jumper resistors, etc.)	✓
All components not marked DNP are mounted	✓
Components are mounted on correct footprints (and correct orientations)	✓
All component pins are soldered to pads properly	✓
No solder bridges between component pins	✓

Power Supply Verification	
Voltage at power connector is within specification	✓
Populate power jumper resistor at power connector	✓
Output voltages of voltage regulators are within specification	X✓
Populate power jumper resistors at voltage regulators	✓

Functionality Verification	
Can connect to FPGA via JTAG interface	✓
Debug LEDs and reference clock are working	✓
SPI interface is working	✓
DDC interface is working	✓
TMDS interface is working	✓
Can connect to PROM via JTAG	✓
FPGA can be configured from PROM	✓

Appendix C

FPGA Resource Utilisation Summary

fpga Project Status			
Project File:	uge.xise	**Parser Errors:**	No Errors
Module Name:	fpga	**Implementation State:**	Programming File Generated
Target Device:	xc6slx25-3ftg256	• **Errors:**	
Product Version:	ISE 14.7	• **Warnings:**	
Design Goal:	Balanced	• **Routing Results:**	All Signals Completely Routed
Design Strategy:	Xilinx Default (unlocked)	• **Timing Constraints:**	All Constraints Met
Environment:	System Settings	• **Final Timing Score:**	0 (Timing Report)

Device Utilization Summary				[-]
Slice Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of Slice Registers	7,743	30,064	25%	
Number used as Flip Flops	7,732			
Number used as Latches	0			
Number used as Latch-thrus	0			
Number used as AND/OR logics	11			
Number of Slice LUTs	7,934	15,032	52%	
Number used as logic	7,544	15,032	50%	
Number using O6 output only	5,657			
Number using O5 output only	493			
Number using O5 and O6	1,394			
Number used as ROM	0			
Number used as Memory	182	3,664	4%	
Number used as Dual Port RAM	32			
Number using O6 output only	4			
Number using O5 output only	0			
Number using O5 and O6	28			
Number used as Single Port RAM	0			
Number used as Shift Register	150			
Number using O6 output only	14			
Number using O5 output only	0			
Number using O5 and O6	136			
Number used exclusively as route-thrus	208			
Number with same-slice register load	161			
Number with same-slice carry load	47			
Number with other load	0			
Number of occupied Slices	2,970	3,758	79%	
Number of MUXCYs used	2,276	7,516	30%	
Number of LUT Flip Flop pairs used	9,940			
Number with an unused Flip Flop	3,004	9,940	30%	
Number with an unused LUT	2,006	9,940	20%	
Number of fully used LUT-FF pairs	4,930	9,940	49%	
Number of unique control sets	311			
Number of slice register sites lost to control set restrictions	690	30,064	2%	

Number of bonded IOBs	23	186	12%	
Number of LOCed IOBs	22	23	95%	
IOB Master Pads	4			
IOB Slave Pads	4			
Number of RAMB16BWERs	44	52	84%	
Number of RAMB8BWERs	0	104	0%	
Number of BUFIO2/BUFIO2_2CLKs	1	32	3%	
Number used as BUFIO2s	1			
Number used as BUFIO2_2CLKs	0			
Number of BUFIO2FB/BUFIO2FB_2CLKs	0	32	0%	
Number of BUFG/BUFGMUXs	5	16	31%	
Number used as BUFGs	5			
Number used as BUFGMUX	0			
Number of DCM/DCM_CLKGENs	1	4	25%	
Number used as DCMs	0			
Number used as DCM_CLKGENs	1			
Number of ILOGIC2/ISERDES2s	0	272	0%	
Number of IODELAY2/IODRP2/IODRP2_MCBs	0	272	0%	
Number of OLOGIC2/OSERDES2s	8	272	2%	
Number used as OLOGIC2s	0			
Number used as OSERDES2s	8			
Number of BSCANs	0	4	0%	
Number of BUFHs	0	160	0%	
Number of BUFPLLs	1	8	12%	
Number of BUFPLL_MCBs	0	4	0%	
Number of DSP48A1s	17	38	44%	
Number of ICAPs	0	1	0%	
Number of MCBs	0	2	0%	
Number of PCILOGICSEs	0	2	0%	
Number of PLL_ADVs	2	2	100%	
Number of PMVs	0	1	0%	
Number of STARTUPs	0	1	0%	
Number of SUSPEND_SYNCs	0	1	0%	
Average Fanout of Non-Clock Nets	3.55			

Performance Summary			[-]
Final Timing Score:	0 (Setup: 0, Hold: 0, Component Switching Limit: 0)	**Pinout Data:**	Pinout Report
Routing Results:	All Signals Completely Routed	**Clock Data:**	Clock Report
Timing Constraints:	All Constraints Met		

Detailed Reports					[-]
Report Name	**Status**	**Generated**	**Errors**	**Warnings**	**Infos**
Synthesis Report	Current	Tue May 8 17:54:02 2018			
Translation Report	Current	Tue May 8 17:54:18 2018	0	1 Warning (0 new)	7 Infos (0 new)

Map Report	Current	Tue May 8 17:58:15 2018	0	2 Warnings (0 new)	10 Infos (0 new)
Place and Route Report	Current	Tue May 8 18:00:07 2018	0	2 Warnings (0 new)	2 Infos (0 new)
Power Report					
Post-PAR Static Timing Report	Current	Tue May 8 18:00:30 2018	0	2 Warnings (0 new)	4 Infos (0 new)
Bitgen Report	Current	Tue May 8 18:01:14 2018	0	0	0

Secondary Reports		[-]
Report Name	**Status**	**Generated**
ISIM Simulator Log	Current	Tue May 8 18:37:44 2018
WebTalk Report	Current	Tue May 8 18:01:15 2018
WebTalk Log File	Current	Tue May 8 18:01:23 2018

Date Generated: 05/13/2018 - 09:16:52

Appendix D

MicroGE Register Map

MicroGE Register Map

Register Category	Register Name	Address	Bits	Description	Default
Configuration	CanvasHorizontalOffset	0x61	[7:0]	Horizontal offset of canvas from left of video frame	0x00
	CanvasVerticalOffset	0x62	[7:0]	Vertical offset of canvas from top of video frame	0x00
	PixelsPerGrid	0x02	[3:0]	Value of PixelsPerGrid property	0x8
	InterlacedVideo	0x60	0	Set to configure MicroGE for interlaced video modes	0x0
IO	GPO[4]	0x63 to 0x66	[7:0]	Signal values that should be provided to output ports	0x00
	GPI[4]	0x67 to 0x6A	[7:0]	Signal values that are provided to input ports	0xUU
Synchronous Control	PrimitiveMemoryUpdate	0x5F	0	Allows the CopyPrim operation to be performed; self clears to 0	0x0
Asynchronous Control	InsertPrim	0x03	0	Set to perform the InsertPrim operation; self clears to 0	0x0
	DeletePrim	0x03	1	Set to perform the DeletePrim operation; self clears to 0	0x0
	StorePixels	0x03	2	Set to perform the StorePixels operation; self clears to 0	0x0
	StoreChars	0x03	3	Set to perform the StoreChars operation; self clears to 0	0x0
	ClearPrim	0x03	4	Set to perform the ClearPrim operation; self clears to 0	0x0
Primitive Data	ID (L)	0x04	[7:0]	LSBs of id attribute to insert into primitive buffer	0x00
	ID (H)	0x05	[1:0]	MSBs of id attribute to insert into primitive buffer	0x0
	Xi	0x06	[7:0]	xi attribute to insert into primitive buffer	0x00
	Xf	0x07	[7:0]	xf attribute to insert into primitive buffer	0x00
	Yi	0x08	[7:0]	yi attribute to insert into primitive buffer	0x00
	Yf	0x09	[7:0]	yf attribute to insert into primitive buffer	0x00
	Z	0x0A	[7:4]	z attribute to insert into primitive buffer	0x0
	T	0x0A	[3:0]	t attribute to insert into primitive buffer	0x0
	ClrRed / Adr (L)	0x0B	[7:0]	Red clr/LSBs of adr attribute to insert into primitive buffer	0x00
	ClrGreen / Adr (H)	0x0C	[7:0]	Green clr/MSBs of adr attribute to insert into primitive buffer	0x00
	ClrBlue	0x0D	[7:0]	Blue clr attribute to insert into primitive buffer	0x00
Pixel Data	PixelSetRed[12]	0x20 to 0x2B	[7:0]	Red components of pixels to transfer to raster memory	0x00
	PixelSetGreen[12]	0x2C to 0x37	[7:0]	Green components of pixels to transfer to raster memory	0x00
	PixelSetBlue[12]	0x38 to 0x43	[7:0]	Blue components of pixels to transfer to raster memory	0x00
	PixelAddress (L)	0x50	[7:0]	LSBs of address where pixels should be transferred to	0x00
	PixelAddress (H)	0x51	[7:0]	MSBs of address where pixels should be transferred to	0x00
Character Pointer Data	CharacterPointerSet[12]	0x44 to 0x4F	[7:0]	Character pointers to transfer to string memory	0x00
	CharacterPointerLength	0x5D	[3:0]	Number of character pointers in CharacterPointerSet	0x0
	CharacterPointerAddress (L)	0x54	[7:0]	LSBs of address where character pointers should be transferred to	0x00
	CharacterPointerAddress (H)	0x55	[7:0]	MSBs of address where character pointers should be transferred to	0x00

All registers not shown are reserved and read 0

Appendix E

MGAPI Functions

MGAPI Functions

High-Level Functions				
Category	Function Name	Description	Arguments	Return
Configuration	MgapiSetCanvasHorizontalOffset	Sets canvas horizontal offset from left of video frame	CanvasHorizontalOffset	-
	MgapiSetCanvasVerticalOffset	Sets canvas vertical offset from top of video frame	CanvasVerticalOffset	-
	MgapiSetPixelsPerGrid	Sets value of PixelsPerGrid property	PixelsPerGrid	-
	MgapiSetInterlacedVideo	Configures MicroGE for interlaced video modes	InterlacedOn	-
IO	MgapiSetGpoOutput	Sets signal values that should be provided to output ports	GpoValue, GpoIndex	-
	MgapiReadGpiInput	Reads signal values that are provided to input ports	GpiIndex	GpiValue
Render	MgapiInsertPrimitive	Inserts a primitive into primitive buffer	AtrId AtrXi AtrXf AtrYi AtrYf AtrZ AtrT AtrClrRed AtrClrGreen AtrClrBlue AtrAdr	-
	MgapiDeletePrimitive	Removes a primitive from primitive buffer	AtrId	-
	MgapiClearPrimitiveBuffer	Clears primitive buffer	-	-
	MgapiUpdatePrimitiveMemory	Updates primitive memory with primitive buffer contents	-	-
	MgapiTransferRasterBlock	Transfers a raster block to raster memory	RasterBlock RasterBlockAdr PixelsPerGrid	-
	MgapiTransferCharacterPointers	Transfers a set of character pointers to string memory	CharPtrsSet CharPtrsSetAdr CharPtrsSetLength	-

Low-Level Functions			
Function Name	Description	Arguments	Return
MgapiWriteData	Writes a byte via SPI interface	WriteByte	-
MgapiReadData	Reads a byte via SPI interface	-	ReadByte
MgapiDelay	Provides a delay specified in microseconds	DelayMicroSec	-
MgapiInit	Performs all initialisation required by other low-level functions	-	-

Appendix F

Ethics Approval Form

Application for Approval of Ethics in Research (EiR) Projects
Faculty of Engineering and the Built Environment, University of Cape Town

APPLICATION FORM

Please Note:
Any person planning to undertake research in the Faculty of Engineering and the Built Environment (EBE) at the University of Cape Town is required to complete this form **before** collecting or analysing data. The objective of submitting this application *prior* to embarking on research is to ensure that the highest ethical standards in research, conducted under the auspices of the EBE Faculty, are met. Please ensure that you have read, and understood the **EBE Ethics in Research Handbook** (available from the UCT EBE, Research Ethics website) prior to completing this application form: http://www.ebe.uct.ac.za/usr/ebe/research/ethics.pdf

APPLICANT'S DETAILS		
Name of principal researcher, student or external applicant		Francois Tolmie
Department		Electrical Engineering
Preferred email address of applicant:		francoistolmie@gmail.com
If a Student	Your Degree: e.g., MSc, PhD, etc.,	MSc
	Name of Supervisor (if supervised):	S. Ginsberg
If this is a research contract, indicate the source of funding/sponsorship		—
Project Title		Design of an HDMI interface for MCUs

I hereby undertake to carry out my research in such a way that:

- there is no apparent legal objection to the nature or the method of research; and
- the research will not compromise staff or students or the other responsibilities of the University;
- the stated objective will be achieved, and the findings will have a high degree of validity;
- limitations and alternative interpretations will be considered;
- the findings could be subject to peer review and publicly available; and
- I will comply with the conventions of copyright and avoid any practice that would constitute plagiarism.

SIGNED BY	Full name	Signature	Date
Principal Researcher/ Student/External applicant	DF Tolmie	[signature]	17/01/2017

APPLICATION APPROVED BY	Full name	Signature	Date
Supervisor (where applicable)	S. Ginsberg	[signature]	16/01/2017
HOD (or delegated nominee) Final authority for all applicants who have answered NO to all questions in Section1; and for all Undergraduate research (Including Honours).	SUNETRA CHOWDHURY	Sunetra Chowdhury	14/1/17
Chair : Faculty EIR Committee For applicants other than undergraduate students who have answered YES to any of the above questions.	Click here to enter text.		Click here to enter a date.