Design and Development of a Secure Compartmentalised 8-bit Architecture

By Timothy Kirby

Bachelor of Engineering (Robotics), Master of Engineering (Electronics)

Supervisor: Paul Gardner-Stephen

October 2017

Submitted to the School of Computer Science, Engineering, and Mathematics in the Faculty of Science and Engineering in partial fulfilment of the requirements for the degree of Bachelor of Engineering (Robotics), Master of Engineering (Electronics) at Flinders University – Adelaide, Australia

Declaration of Academic Integrity

I certify that this thesis does not incorporate without acknowledgment any material previously submitted for a degree or diploma in any university; and that to the best of my knowledge and belief it does not contain any material previously published or written by another person except where due reference is made in the text.

Timothy Kirby

Date: 11/10/17

Acknowledgements

To my family and friends, thank you for your support throughout my years at Flinders University. I would also like to thank Benjamin Gerblich for his help in understanding the MEGA65, and my supervisor, Paul Gardner-Stephen, for the opportunity to undertake this project and for his advice and support throughout.

Contents

List of Tables

List of Figures

Abstract

This thesis describes an architecture of a system for isolating sensitive tasks within the context of the 8-bit retro revival project, the MEGA65. This system has the potential to provide users with a means of secure communication on a platform which is simple enough to be understood by the user: A boon for the supporters of privacy and free speech.

In an attempt to reach the completed system, a sub-system was successfully designed and developed that provides the ability for the user to inspect the state of the machine without an external host computer. Architectural changes were conceptualised and designed to allow for secure exfiltration-resistant compartments to be constructed and used. Furthermore, the development and implementation of a rudimentary method of context switching lays a foundation for the future development of a multitasking system on this 8-bit platform.

Chapter 1

Introduction

Modern computers are complex. While this increasing complexity drives the performance we have come to depend upon, as the complexity of a system increases so too does the probability of flaws. Some of these flaws can potentially be a security risk. Bruce Schneier, Founder and CTO of Counterpane Internet Security, Inc., mentioned that “the future of digital systems is complexity, and complexity is the worst enemy of security” (Schneier, 2000).

Major processor architecture changes see an approximate fourfold increase in transistor count and a fourfold increase in design bugs (Gelsinger, et al., 2010). Some processor bugs could potentially weaken the security of the system by allowing unprivileged code to control the privileged state of the processor (Matthew Hicks, 2015). K. Yang, et al. showed that such a flaw even has the potential to be maliciously placed by unscrupulous fabrication facilities. (K. Yang, et al., 2016).

Recently, it was revealed that the CIA have been hoarding zero-day vulnerabilities; that is, vulnerabilities which were previously unknown to the vendor (WikiLeaks, 2017; US Govt., 2012). These vulnerabilities exist in major consumer platforms such as Android, iOS, Windows, and even consumer Wi-Fi routers. A user may wish to protect themselves from state-sponsored hackers who may exploit these vulnerabilities. Perhaps they could do this by moving to a much simpler platform.

Such a platform is the MEGA65 project. Developed by Paul Gardner-Stephen in collaboration with the M.E.G.A. (Museum of Electronic Games and Art), developers and commodore enthusiasts from all over the world, as well as students at Flinders University. The aim of the MEGA65 project is to recreate and enhance the popular 8-bit computer, the Commodore 64 and its never-released successor the Commodore 65 prototype.

Using a dedicated 8-bit machine to communicate, rather than the ubiquitous smart phones we are accustomed to, may be a major regression in convenience. But the increased security could provide peace of mind to some users.

1.1 Motivation

Spooked by terrorist attacks worldwide, much of the western world has used this opportunity to push for greater surveillance. In early 2015 UK Prime Minister at the time, David Cameron, and his team proposed the ban of strong encryption, such as end-to-end encryption used in many messaging applications or 256-bit AES used to secure phones, citing national security concerns (Griffin, 2015; Morris, 2015; Cowan, 2015). While this is a boon for police investigators, it leaves citizens potentially vulnerable. In this proposal officials requested a company be able to remove any electronic protections to access the required information. In late 2016 the UK Parliament passed Investigatory Powers Act 2016 of which section 254 provided them with such a power (UK Parliament, 2016). While companies can still technically employ encryption, they must also be able to somehow supply any requested data, coercing them into implementing surveillance backdoors, or weakening encryption using techniques such as key escrow.

At a press conference in July 2017, Australian Prime Minister Malcom Turnbull reiterated the push for greater powers to police the internet. He argued that the internet should not be used for bad people to hide their criminal activities from the law (Turnbull, 2017). It is becoming increasingly difficult to find out what terrorists, drug traffickers and paedophile rings are doing, and it appears the blame is on encryption. At the same time, recent new anti-terror laws in Queensland have provided the police with the authority to hack consumer electronics to provide surveillance.

Once the knowledge of strong encryption is disseminated and the technology to implement such a system is widely available, anti-encryption regulation would only serve to make the general public less safe (Bankston, et al., 2015). Key recovery systems, where the third party holds the keys or any other backdoor system in place to allow quick access to the plain text for law enforcement, have the possibility to be hacked and used by others.

If the communication service used by the criminals becomes compromised or weakens their encryption, it is logical to assume that they will seek out other means of secure communication. Whether it is open source projects like this or the development of their own software using open source implementations, it would render any regulations moot. This forces criminals into the deep web, that is, not using mainstream services, and will only make it harder for law enforcement as they will no longer receive any metadata. Metadata is any data other than the

message content, such as the sender/recipient and the time it was sent, which could be useful information to the police.

If granted, these new powers could be a slippery slope. Much like the NSA's existing dragnet mass surveillance system, it has the potential to be abused. The increasing surveillance could potentially be used to crackdown on protesters or political opponents, especially in the repressive regimes of the Middle East.

1.2 Problem Statement

For the rightfully paranoid, perhaps existing mainstream closed source platforms cannot be completely trusted given that western governments are attempting to boost their surveillance powers. If the user were a person of interest, whether it's because of political dissent, journalism, or whistleblowing, flaws or backdoors hidden in complex hardware and software could potentially be exploited by a state-sponsored hacker. Using, instead, a secure simple open platform may reduce the likelihood of a security vulnerability, but at the cost of convenience and functionality.

1.3 Objectives

The primary objectives of this project are:

- To explore compartmentalisation requirements
- To design and implement an out-of-band machine state inspection mechanism
- To design and implement a system to isolate tasks into a secure compartment
- To design and implement a rudimentary task switcher

1.4 Scope

The scope of this project includes the design and implementation of sub-systems that work together to complete the system to enable the compartmentalisation of secure tasks. Development of sub-systems which might eventually be a part of the secure task system, such as the implementation of a completed task switching system with UI are out of scope of the project. Additionally, implementing the requisite hashing and encryption algorithms is not within the scope of this project.

1.5 Research Questions

With the truly paranoid in mind, how could the secure compartmentalisation of tasks be implemented in the simple 8-bit architecture of the MEGA65? And to what extent is it possible to protect against physical access attacks, and which use cases can be protected?

And finally, is it possible to create a functional user-interface that allows for the transparent inspection of the machine's state such that a user can reasonably verify that the machine is running the correct software, or similarly, that it is not exfiltrating sensitive data from a secure compartment?

The main criteria for success is whether the functionality of the implemented systems meet the requirements of the inspection mechanism and the secure compartments, and whether these systems can protect from physical access attacks.

1.6 Thesis Organisation

Chapter 1: Introduction

Gives an overview of the background of the project, including motivation, problem statement, objectives, and scope of the project.

Chapter 2: Literature Review

Chapter 2 presents background information and related work. The literature review contains synopses on several academic papers concerned with various aspects of computer security, including backdoors and physical access attacks, attacks this architecture is aiming to protect against, and others which may still be outstanding vulnerabilities.

Chapter 3: Design of Secure System

Chapter 3 starts with the software tools and methods used to perform the research, and introduces the functionality requirements of the system. Followed by a proposed design fulfilling those requirements.

Chapter 4: Results and Discussion

Chapter 4 presents the progress made towards implementing the proposed designs on the MEGA65. It describes the modules which were developed, an overview of how they were implemented and tested, and whether the functionality requirements were met. Then ending with a discussion and analysis of how the overall system, when fully implemented, could protect against physical access attacks and which cases could be protected.

Chapter 5: Conclusion and Future Work

Chapter 5 presents a summary of the project. The chapter concludes with a discussion of the future work which could build upon the foundation created by this project.

Chapter 2

Literature Review

There have been attempts in the past to create secure systems, in both software design and in hardware design. The following chapter presents some background information on various aspects of computer security which relate to the nature of this project and an overview of related work.

2.1 The MEGA65 Project

The MEGA65 is an open source project that aims to create an understandable computer which is near 100% compatible with the 8-bit Commodore 64 and the prototype Commodore 65. And in doing so, supplementing it with newer technology such as support for higher resolution display, networking and MicroSD card storage (MEGA65, 2017).

The device runs on an off-the-shelf field programmable gate array (FPGA) development board. The development board was chosen as it was cheap relative to the performance and capacity of the included FPGA. Additionally it includes on-board peripherals, such as Ethernet, VGA output, USB keyboard input, and audio output (Gardner-Stephens, et al., 2016). Efforts to build a custom PCB are underway, with support for the real commodore 64 peripherals, such as the keyboard and joysticks.

As the MEGA65 project has an emphasis on being understandable, it has a potential future in education teaching children about the basics of computers at a very low level of abstraction. Being taught *how* to use a modern computer in the 21st century doesn't impart anything about the computer itself, it remains a mysterious box. However, in the early 80s as 8-bit home computers were becoming very popular, anyone with an interest could learn to program whether it was at school, clubs, or at home. There were difficulties with software distribution at the time, so to do much with the machine typically meant typing in programs found in PC magazines, or creating one's own software.

The machines of this era were inherently simplistic, a limitation of technology at the time, which may have promoted a deeper understanding of the underlying hardware.

2.2 Complexity and Security

There's a wide consensus in the computer security community that complexity is the enemy of security, meaning as complexity increases so do the number of potential security flaws. This notion is supported by notable security experts such as A.Yoran (Krebs, 2014), B. Schneier (Chan, 2012) and McCabe (McCabe Software, INC., 2012). This sentiment also lies at the heart of this project.

According to McCabe, complex systems potentially have more security bugs as they have more lines of code and more interactions. Complex systems are harder to fully test, and may have some portions untested. Complex systems are harder to design, implement, configure and use securely. Lastly and importantly, complex systems are harder for users to understand. Reducing the complexity of a computer system has the potential to make it more secure (McCabe Software, INC., 2012).

It is difficult to test for security, functional testing cannot tell us whether or not the device is secure as these security features are useful only to prevent things from happening (Schneier, 1999). The system could be working as intended and a vulnerability elsewhere in the system could undermine the feature. However, by keeping the overall system simple, we can reduce the effort required to analyse and evaluate the security of the system.

The CPU in the Commodore 64 has approximately 3510 transistors (Cox, 2011), whereas a 2017 desktop class CPU from AMD has almost 5 billion transistors. It can easily be seen that an 8-bit system, such as a Commodore 64, is definitely not a complex system when compared to modern computers.

2.3 Security by isolation

The idea of security by isolation is to segregate different sections of a computer system such that if one of these sections becomes compromised or malfunctions, the rest of the system is unaffected (Rutkowska, 2008). In 1975, Saltzer and Schroeder proposed eight design principles for the protection of information in computer systems. One of these is related to isolation: the principle of least privilege. It states that every module must only be able to access the information and resources it requires to complete the job (Saltzer & Schroeder, 1975). This is analogous to the military "need to know" security rule, where limiting the number of people with access to classified information lowers the risk of that information being compromised.

By isolating sections of the MEGA65 during a special mode, even if sections, such as the hypervisor or cellular modem, become compromised it should not compromise the security of activity during secure mode. These ideas will be explored in more depth in later chapters.

2.4 Electromagnetic Emanation Attacks

All electronic devices emit some electromagnetic radiation when current passes through their circuits. In 1985 Wim Van Eck found that it was possible to pick up and decode these electromagnetic emissions from devices such as CRT monitors (Van Eck, 1985). As cathode ray tube (CRT) monitors amplify the control signals to a high voltage, it was found that these video signals were a dominant component of the EM radiation emitted. The signal showed resemblance to a TV broadcast signal. Van Eck demonstrated that eavesdropping was possible using off the shelf equipment of the time, a television receiver. Attack of this nature were therefore dubbed Van Eck phreaking.

Attacks targeting electromagnetic radiation aren't limited to the analog realm. A recent paper by experts at Tel Aviv University showed that it was possible to acquire decryption keys from laptop computers using off the shelf hardware from a distance of 50cm. (Genkin, et al., 2015). Initial attempts using a cheap software defined radio (SDR) device as an acquisition device to listen in on specific frequencies were successful. These electromagnetic analysis attacks are a type of side-channel attack which exploit the physical implementation of a cryptographic system.

Since 2012 it is been known that encrypted FPGAs from Xilinx series 4 and 5 are susceptible to side-channel analysis attacks (Moradi, et al., 2012). By measuring and analysing from a single power-up, they were able to extract the AES key. With the key they can decrypt the bitstream to enable cloning or reverse engineering of the device. In 2016 another team improved the attack, targeting series 6 and 7 devices (Moradi & Schneider, 2016). These attacks are especially worrying, considering that the MEGA65 projects use a series 7 Xilinx FPGA in the design.

2.5 Backdoors

The use of undocumented closed source hardware and software has the potential to hide intentional backdoors, or unintentional security vulnerabilities. A backdoor is a method which circumvents authentication in a system and may allow unauthorised users to access the system.

There are many places a backdoor can be planted in, from the silicon in the underlying CPU, the operating system software, user software, and even computer peripherals.

One such example, which sparked many conspiracy theories, is a system known as the Intel Management Engine (IME). The IME system is a closed source embedded microcontroller inside recent Intel processors, which has unfettered access to hardware, including a dedicated connection to the network interface and memory access (Rutkowska, 2015). By employing security-by-obscurity Intel has made the system unable to be easily audited by the public, leading to speculation over potential security flaws. Such a flaw was found in mid-February 2017, which allowed an unprivileged remote attacker to gain control of the features provided by the AMT, a module included in the IME (CVE, 2017), affecting a range of products sold between 2010 and 2011 (Embedi, 2017). A recent submission to the Black Hat Briefings suggests a vulnerability was found in newer versions of the IME, enabling unsigned code to execute in the Platform Controller Hub (PCH) of any sixth-generation or above Intel CPU (Mark & Goryachy, 2017).

While Intel may not have been acting maliciously, there are growing concerns due to the globalisation of hardware supply chains, specifically hardware vulnerabilities which may be implanted during the design stages, unbeknownst to the original designer, by dishonest fabrication facilities. As designs get more complex, often third party intellectual property (IP) blocks are utilised in a design, and because of this the hardware may contain unspecified functionality. These undocumented modifications may introduce a hidden channel to exfiltrate data, or an intentionally placed backdoor (Hu, et al., 2016).

As mention in the introduction, Yang, et al. demonstrated that a hardware backdoor has the potential to be maliciously placed by a single dishonest worker in a fabrication facility (Yang, et al., 2016). They constructed a circuit that uses nearby wires to charge a capacitor. When the wire toggles frequently the capacitor charges, and when the capacitor is charged above a certain point, the payload is deployed forcing the targeted flip-flop to change to the desired value. The targeted flip-flop would ideally hold the bit for escalated privileges for the processor.

In March 2012, a team from the UK discovered the first real world backdoor found in military grade FPGA. A backdoor exists in the Actel/Microsemi ProASIC3 chip, by providing a key to the JTAG interface, extended debugging features were unlocked. By fuzzing the JTAG port they were able to identify that a function was requesting a 128-bit key. By using Pipeline Emission Analysis (PEA), a type of side-channel attack, similar to differential power analysis,

they were able to extract the key providing access to this interface. With this level of access they were able to disable all the security, access crypto keys and the unencrypted bitstream. This could lead to attackers reverse engineering the design, and re-programming the device having modified it with other backdoors or Trojans (Skorobogatov & Woods, 2012). Considering fuzzing the JTAG interface on FPGAs is a rather new technique, perhaps it is likely similar backdoors will be found in other products. Especially if it were a third party JTAG IP block the FPGA manufacturer may have integrated into the design.

2.6 Evil Maid Attacks

An evil maid attack is a type of physical access attack, coined by security researcher Joanna Rutkowska. It is an exploit that targets devices which have been left unattended and shutdown. The name comes from a hypothetical situation where the attacker could be a hotel maid where the owner of the device leaves it unattended in the room, giving them physical access to the device for a short time, on multiple occasions (Rouse, 2013).

An example of an evil maid attack could be an attack targeting full disk encrypted (FDE) hard drives: A user leaves their laptop unattended in their room, thinking because of FDE their important information will be safe. An attacker could come in and install a malicious bootloader, once the user unlocks the disk the bootloader may install malware to capture the key and send it over the internet. With the key obtained, the attacker could return to copy the data and erase any evidence of their attack. A potential solution to these type of attacks could be to implement some sort of two-factor authentication, and keeping the token generator with you at all times (Schneier, 2009). Or alternatively only store the bootloader you trust on a read only USB device, and keep that device on you at all times. The possibility of similar physical access attacks on the system introduced in this thesis will be discussed in later chapters.

2.7 Secure Operating Systems

While there have not been any systems like the MEGA65 strictly designed with the aim of providing security by *simplicity*, there exist some operating systems designed with security and privacy in mind. These include Tails OS (Tails, 2017), Qubes OS (The Qubes OS Project, 2017) and certain versions of Windows 10. Each have a different approach to how they secure the system.

The operating system of choice of whistle blower Edward Snowden, Tails OS, or **T**he **A**mnesic **I**ncognito **L**ive **S**ystem, is a Unix-like operating system bundled with tools to help preserve

privacy and anonymity online. Tails is designed as a stateless system, leaving no trace of activity on local storage. Tools such as Tor Browser, a browser based on Firefox with modifications to protect your anonymity, and Thunderbird email client with Enigmail for OpenPGP (a widely used email encryption standard) support. All communication is routed through TOR, **T**he **O**nion **R**outer. TOR itself is a special network that aims to anonymise the user to protect their privacy. While the MEGA65 will not be performing such tasks such as web browsing, the ability to anonymise communication using the TOR network or a VPN is worth considering.

The problem with many mainstream operating systems, including Tails OS, is that they use monolithic kernels, a kernel running entirely in a single address space. According to Joanna Rutkowska, project lead of Qubes OS, this could potentially be a security flaw as a single kernel exploit can be used to take over the entire system. (J. Rutkowska 2012). We can't ever be certain that no malicious code can get into the kernel, there's always going to be a security vulnerability that someone finds.

Qubes OS is a secure operating system designed from the ground up to utilise security by compartmentalisation. Qubes utilises the open source virtualisation software, Xen hypervisor, to create Xen domains it calls qubes. These domains provide a way to isolate different sessions, services or applications. You could have a personal qube to run sensitive applications or web services which require login credentials, and a different qube to run general browsing or work-related tasks.

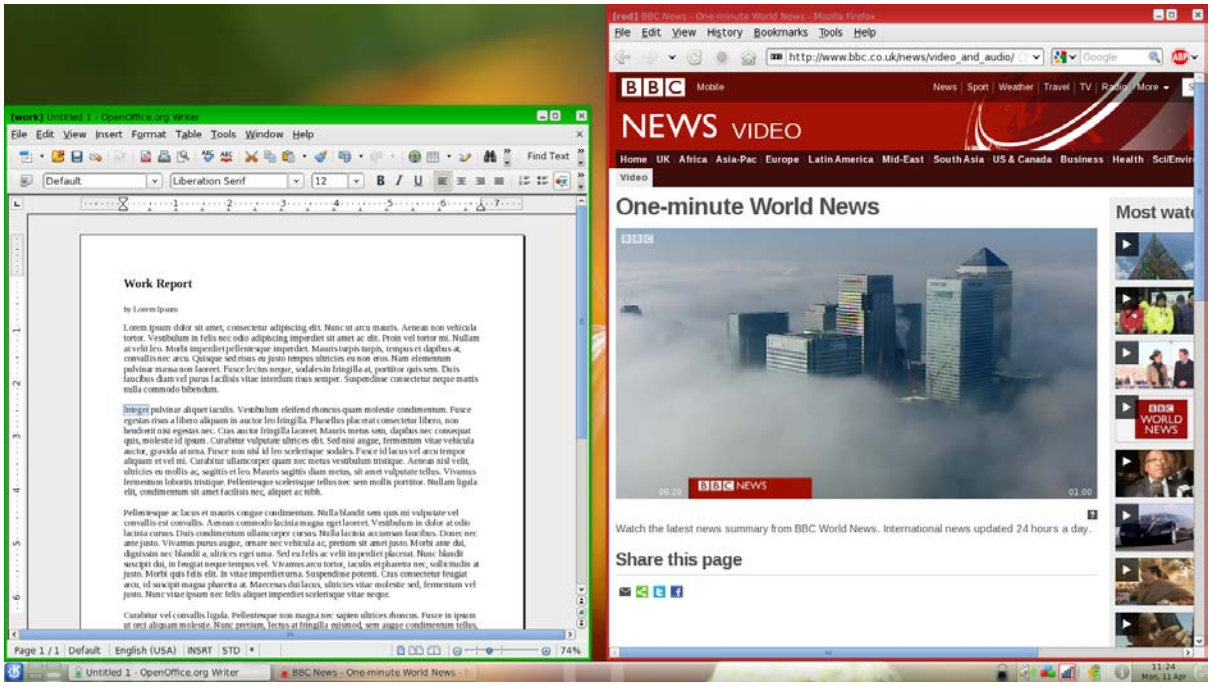

Figure 1. Qubes OS User Interface

While each domain has its own micro kernel, the desktop environment itself is unified providing a single interface to manipulate windows. The interface of Qubes OS, shown in Figure 1, shows two different domains, differentiated by the allegedly unforgeable coloured borders (Qubes OS Project, 2017). However a vulnerability was found in the GUI component which allowed applications to disable the drawing of the coloured border, potentially providing a way for malicious applications to spoof the border (Edge, 2016). In a system with advanced graphics capabilities, it would be difficult to ensure that special on-screen graphics symbolizing privilege are produced by the legitimate software system, given that there is the possibility of bugs like this.

Using a similar method as Qubes, recent enterprise and sever versions of Microsoft's Windows operating system have introduced security features leveraging their existing Hyper-V hypervisor. Virtual Secure Mode is a feature using security by isolation to solve the problem of kernel mode malware and device-based attacks. Even if the kernel is compromised, there are certain things, like credentials, that can be put behind a walled off section so that the kernel doesn't have access.

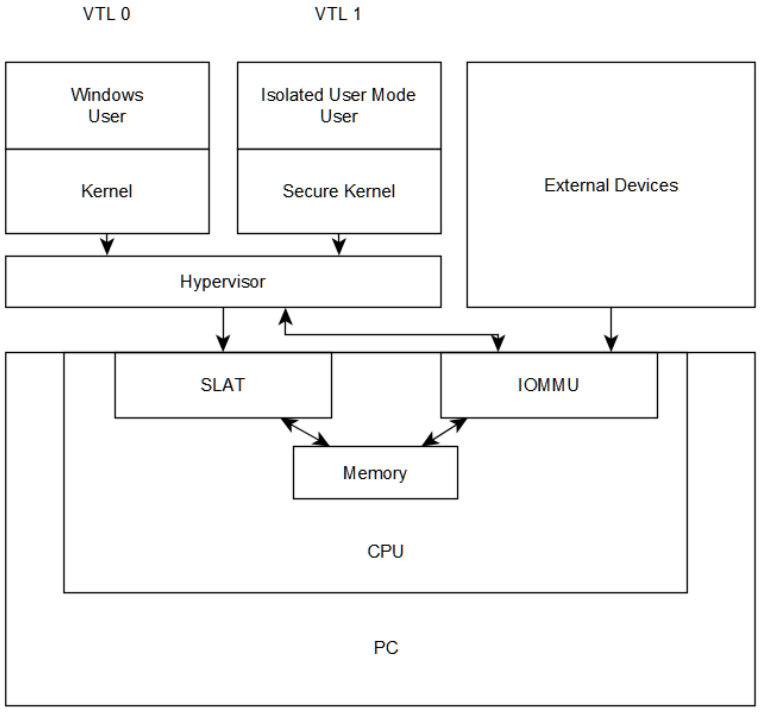

Figure 2. Block diagram of Microsoft's Virtual Secure Mode

Microsoft's Hyper-V hypervisor sits between the hardware and the kernel. A block diagram of the system is shown in Figure 2. The hypervisor can control which portions of memory that the guests can access, using a hardware feature called second level address translation (SLAT). Virtual Trust Levels (VTL) are VMs with various trust levels. VTL1 is the most privileged, containing a minimal kernel, and is used to contain secure applications. If the normal kernel becomes compromised, the SLAT feature is used to contain the potential damage by limiting memory access.

The operating system has device drivers in the kernel, which usually have direct access to physical devices and direct memory access (DMA). However, the hypervisor takes advantage of another hardware feature called IOMMU. The IOMMU has a page table with memory addresses and access permissions. When memory is request from a device the IOMMU checks whether that device has permission to access that portion of memory. This is such that a compromised device driver cannot access memory outside its allocated memory (Juarez, 2015). The two hardware technologies described here work with Microsoft's Hyper-V virtualisation technology to effectively compartmentalise access to memory to protect against rogue device drivers.

2.8 Physically Secure Hardware

The ORWL (pronounced Orwell) is purported to be the first physically secure computing device (Design Shift, 2016). The device looks like a standard Intel-based small form factor PC running a Windows operating system, however it is loaded with security features. The idea is to make it safe from someone trying to physically break into the device. A wire mesh lines the inside of the glass case; when the circuit detects a break, or when pressure switches are relieved by opening the case, the encryption keys are wiped and the device is shut off ensuring any information is destroyed rather than stolen.

There are other features like an NFC (Near Field Communication) key fob required to use the device. If the key gets more than 10 meters away the device will automatically lock, preventing unauthorised access to the unlocked machine (Design Shift, 2016).

However, the root of trust lies in a proprietary microcontroller based secure controller. Presumably this device relies on security-by-obscurity as no source code for the firmware of the microcontroller, nor the NFC or BIOS are supplied, despite claims of open source software and hardware, putting this into a similar situation to the Intel's IME. Figure 3 shows the architecture of the security system used in the ORWL computer.

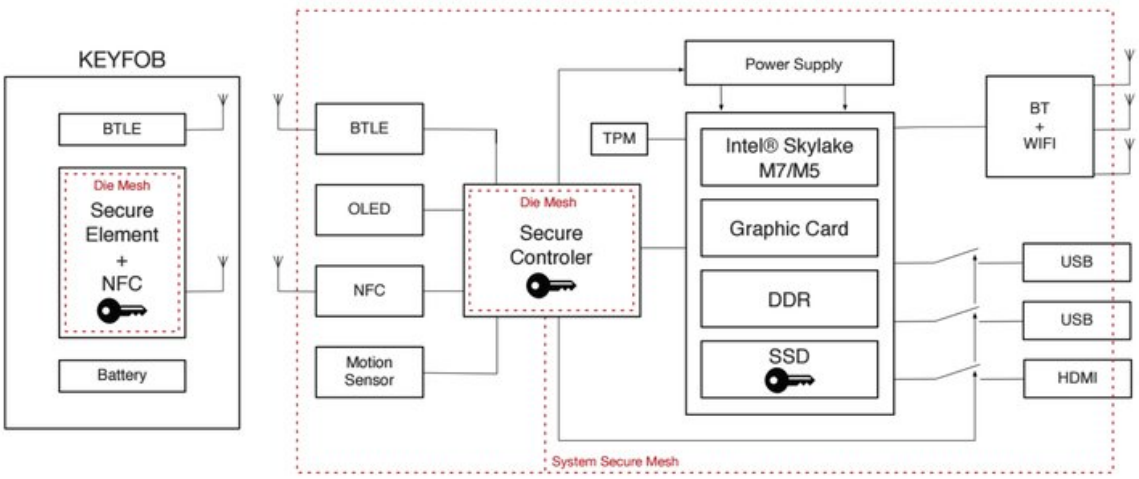

Figure 3. Architecture of the ORWL computer's security system

While the system may stop nosey roommates or co-workers from snooping around your computer, and basic physical access attacks, it may not withstand more sophisticated attacks. Depending on how the inertial sensor is utilised, i.e. if physically relocating the device doesn't

trigger a key wipe, the NFC authentication may be susceptible to a relay attack. The attacker replaces the entire unit with an identical copy and uses it to relay communication between the key fob and the original device (Rutkowska, 2016).

Additionally, if the device is running a standard operating system, advertised as running windows or Linux, the device would still susceptible to any attacks targeting that operating system. Interestingly the buyer also has the choice of preloading with Qubes OS.

2.9 Summary

This chapter presented an overview of various aspects of computer security relating to the project, and a review of similar attempts of creating a secure system. The next chapter introduces the design of the secure system.

Chapter 3

Design of Secure System

As the MEGA65 is a rapidly developing open source project it is not well-documented, and source code needed to be analysed to learn about the major systems functions. Additionally, problem solving by trial and error were employed to find and fix bugs. The project utilised an iterative development cycle. Requirements for the system were developed, the existing source code was analysed, and new additions which met these requirements were devised and implemented. Designs were synthesised and deployed on FPGA hardware then the functionality was evaluated. These steps were repeated as necessary.

The development was performed primarily in the software package Xilinx ISE 14.7 using VHDL (**V**ery High Speed Integrated Circuit **H**ardware **D**escription **L**anguage), and Ophis as the 6502 assembler. The operating system used was Ubuntu 15.04. An existing compile script, setup for the MEGA65 project, was used to compile, build and integrate the necessary external files (Gerblich, 2017).

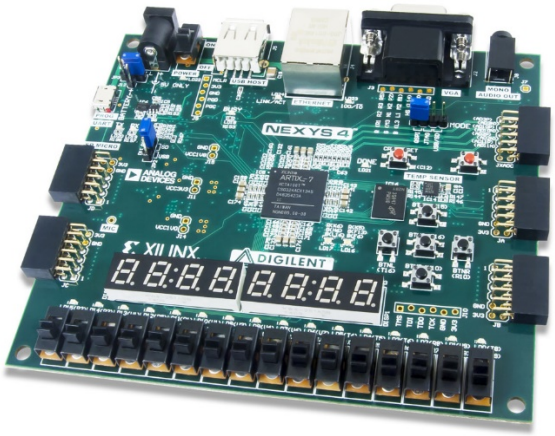

Figure 4. Digilent Nexys 4 DDR FPGA Development Board

The FPGA development board, Digilent Nexys 4 DDR, shown in Figure 4, was used as the target platform. A 512MB microSD card was used to store the configuration bitstream, C65 ROM, and other necessary files. The Nexys 4 board was connected to a monitor via VGA and powered through a USB cable connected to the development PC.

3.1 Out-of-band machine state inspection

In order for a user to reasonably verify that the machine is running the correct software, or similarly that it is not exfiltrating sensitive data, a hardware mechanism independent of the CPU is required to allow the user to inspect the state of the machine and contents of memory. Machine state is defined by contents of memory, RAM, ROM, and the CPU and I/O registers. The machine's CPU must not know that it being inspected or otherwise be able to be manipulated by software; therefore, such an inspection mechanism will need a dedicated interface to the memory, i.e., out-of-band. Furthermore, this feature must be somehow uniquely identifiable to the user such that it cannot possibly be spoofed by software. The basic user functionality required for inspection of machine state is the ability to read and write contents of RAM, ROM and the CPU registers and be able to halt the CPU such that memory doesn't change during inspection.

3.1.1 Remote Serial Monitor

A method of inspecting the machine state is already implemented in the MEGA65 with a feature called the remote serial monitor. The remote serial monitor is a rudimentary command line interface implemented in hardware, i.e. not software running on the CPU, originally designed to help developers debug their programs.

By connecting a USB cable from the FPGA development board to a PC, the device is identified as a virtual serial port. Using a terminal emulator program, such as PuTTY on Windows, and connecting at 230400 bps, the user is presented with a text-based interface. This interface can be used for displaying the contents of memory, writing to memory, inspecting CPU flags and registers, freezing the CPU, single-stepping the CPU and setting break points.

This already meets some of the requirements mentioned in the previous section; however, security wise, there are some drawbacks of external monitoring. To connect to the serial monitor requires the device to be physically connected to another computer. This computer might be targeted in an attack, allowing remote control of the serial monitor. As the serial monitor can be used to dump or fill the entirety of memory it could potentially be used to exfiltrate information or introduce a malicious program. The remote nature of this feature makes this a potential vulnerability. Modifications need to be made to remove this vulnerability.

3.1.2 Integrated Serial Monitor

An internal user interface which moves this functionality from an external computer to the VGA monitor directly connected to the MEGA65, would provide users with a more secure and convenient method of inspecting the machine state. This mode could be transparently overlayed on top of the existing video output. This alpha-blending cannot be reproduced by the VIC-IV in software, due to the limitations of that video controller, meaning it cannot be spoofed. Other unique indicators could be to set an LED to be a certain colour when this mode is active.

To modify the current system to provide this functionality would require the development of multiple subsystems:

- A terminal emulator to show the existing output of the serial monitor component on the MEGA65's display.
- A keyboard-to-serial converter to interface the MEGA65 keyboard with the existing input of the serial monitor.
- A video generator to composite an overlay over the existing video output.

Chapter 4 gives an overview of the implementation of these systems in more detail.

This system is colloquially referred to as "Matrix Mode", as the feature is giving users a look into "the matrix", from the 1999 Wachowski movie, *The Matrix*. This relates to viewing what is happening inside the machine, in a similar way that the digital rain of Matrix code represents the activity of the virtual reality environment.

Matrix mode alone presents some issues. Users will need to understand their machine and software competently to fully benefit from this feature, and there may be a steep learning curve for those new to the platform or without a background in computer science. However, those motivated enough will be able to learn to use this tool.

Simply identifying that a potential issue is present will not help if your data has already been exfiltrated. To fix this the next section presents a design for a system that combines this inspection mechanism with the necessary architectural extensions required to allow secure exfiltration-resistant compartments to be constructed and used.

3.2 Secure Compartmentalisation

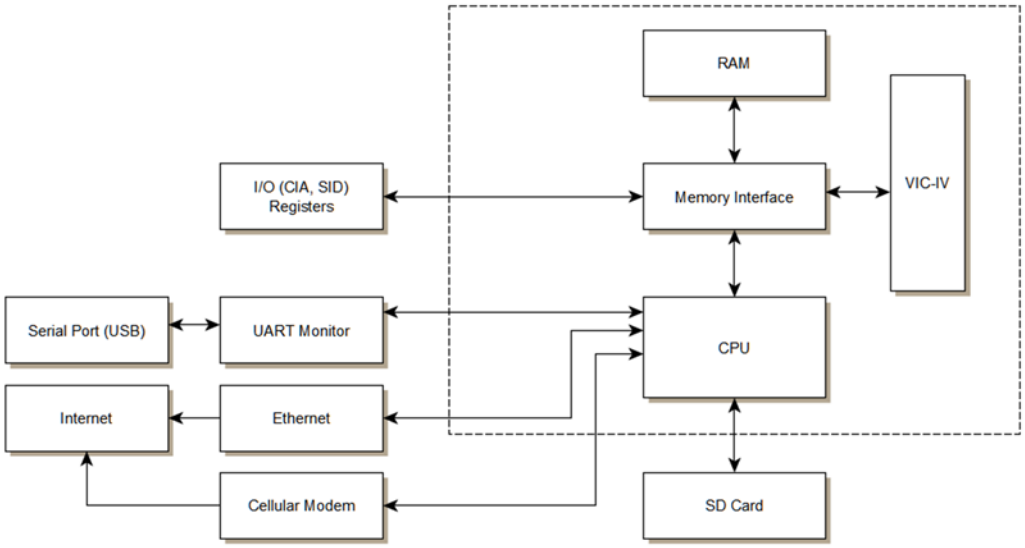

Figure 5. Potential methods of data exfiltration

In order to define an exfiltration-resistant secure compartment the possible ways of exfiltration need to be identified. This section describes some of the components which will need to be excluded from the secure compartment as they may be able to exfiltrate information. Figure 5 shows a block diagram showing potential methods of data exfiltration in the MEGA65.

While in Hypervisor mode, which is a privileged execution mode, the CPU has access to ethernet and eventually features like a baseband processor for mobile connectivity. Ethernet and mobile connectivity both present an issue as data can easily be pushed out over the internet. Therefore, access to the hypervisor and all outside communication must be disabled while in the secure container.

As previously mentioned the existing remote serial monitor has commands to modify or view contents of memory without the CPU knowing. If the device were connected to a PC to power the device, malicious software could exfiltrate data from the MEGA65. Likewise, if someone were to replace hardware inside a USB power brick with a compatible serial interface data could be exfiltrated. A similar vulnerability has been seen recently with smart phones and public charging stations and has been coined juice jacking (Wiggers, 2017). Therefore, we can say that the remote serial monitor should be excluded from the secure container.

The SD card or other bulk persistent storage can also be a channel for data exfiltration. Content can be transferred to the card and later someone can come and collect that data as part of an evil maid attack. Access to the SD Card, the configuration registers and buffer will need to be limited while in the secure container.

There could be other things in future development to consider placing limits on while in the secure compartment, such as expansion RAM, floppy disk emulation, and emulation of the original C65's UART serial interface. Support for external USB keyboards may also be dropped on the final hardware as there will be an integrated keyboard present.

Lastly anything within the secure container which has any persistent memory will need to be cleared or reset to a known state upon existing the secure compartment such that sensitive information cannot possibly remain there.

3.2.1 Hypervisor

In the MEGA65's CPU there are two main modes of operation: user mode and hypervisor mode. A program in user mode can perform a system call which temporarily switches the CPU into hypervisor mode, also known as *trapping*.

If "supervisor" is what an operating system kernel is typically labelled, then "hypervisor" is one step above that. Trapping to the hypervisor is akin to system calls to the kernel (or for the C64, the Commodore KERNAL). But where a system call allows a program to request services from the kernel of the operating system, trapping to the hypervisor runs routines from the hypervisor ROM independent from the Commodore KERNAL, and has full access to the extended hardware.

When the hypervisor is called it saves all CPU registers to their respective shadow registers such that when the CPU returns from the hypervisor these can be restored and the user program can be quickly resumed. These registers are shown in Figure 6.

```
when TrapToHypervisor =>
  -- Save all registers
  hyper_iomode(1 downto 0) <= unsigned(viciii_iomode);
  hyper_dmagic_list_addr <= reg_dmagic_addr;
  hyper_dmagic_src_mb <= reg_dmagic_src_mb;
  hyper_dmagic_dst_mb <= reg_dmagic_dst_mb;
  hyper_a <= reg_a; hyper_x <= reg_x;
  hyper_y <= reg_y; hyper_z <= reg_z;
  hyper_b <= reg_b; hyper_sp <= reg_sp;
  hyper_sph <= reg_sph; hyper_pc <= reg_pc;
  hyper_mb_low <= reg_mb_low; hyper_mb_high <= reg_mb_high;
  hyper_map_low <= reg_map_low; hyper_map_high <= reg_map_high;
  hyper_map_offset_low <= reg_offset_low;
  hyper_map_offset_high <= reg_offset_high;
  hyper_port_00 <= cpuport_ddr; hyper_port_01 <= cpuport_value;
  hyper_p <= unsigned(virtual_reg_p);
```

Figure 6. Hypervisor saving of CPU registers

The hypervisor ROM holds a set of routines which can perform certain tasks. There are registers between $D640 and $D67F. A user can write to one of these registers to initiate a trap to the hypervisor, which corresponds to a specific service. Additionally, there are some system generated traps, which cannot be called by the user, used for things like reset, and page faults.

The hypervisor ROM is integrated with the kickstart ROM, which is a bootstrap to load the original Commodore 65 ROM and the Character ROM, as for legal reasons these cannot be included in the bitstream. The source for the hypervisor routines are found in the file kickstart_task.a65, this is compiled as part of the kickstart ROM and integrated into the bitstream. Or alternatively it is compiled as a separate file, kickup.m65, which will load as an updated kickstart ROM. Given that the hypervisor has access to all of the hardware and its own memory space, access to hypervisor must be limited while in the secure compartment.

3.2.2 Secure Mode

With the secure compartment defined, a method of securely constructing this container and transferring the output data from the container is to be developed. The proposed method is referred to as “Secure mode”. Secure mode for the MEGA65 should allow the user to request for a secure compartment where the user can execute a single secure task without any other part of the system knowing anything other than the specified output of the task.

3.2.2.1 Guiding Use-Case

An example of an application where this would be beneficial is in an email messaging system, the user software would request a secure container for the user to type up a message. The secure mode program will allow the user to write the message, then encrypt it and only return the

encrypted message, and all other state is wiped before control is returned to the user program. Once outside of the secure container the user program is allowed to send this encrypted message via Ethernet or some other means. No access to the plain text is provided outside of the secure container. The next section gives a deeper look at the design of the secure mode system.

3.2.2.2 Design

A flowchart of the secure mode, including the memory contents at each major step, is shown in Figure 7. A secure service can be requested by the user program by trapping to the hypervisor. The hypervisor will then save the state of the task to the SD card. The hypervisor will initiate the clearing of all memory, except for the transfer area, allowing the user program to configure the secure service if necessary, or transferring encrypted data to be decrypted in the container. The hypervisor loads in the secure service program, preserving the transfer area and then sends a request to enter secure mode to the CPU and exits hypervisor mode.

In hardware, the matrix mode overlay is activated, the serial buffer is cleared, the CPU is halted, and a banner message is displayed asking the user to accept or reject entering secure mode. While this prompt for secure mode is active the user can also be checking memory to see whether the transfer area is as expected. Additionally, information such as the name of the secure service, and a cryptographic hash of the service code is to be provided to the user to ensure that the secure service is legitimate.

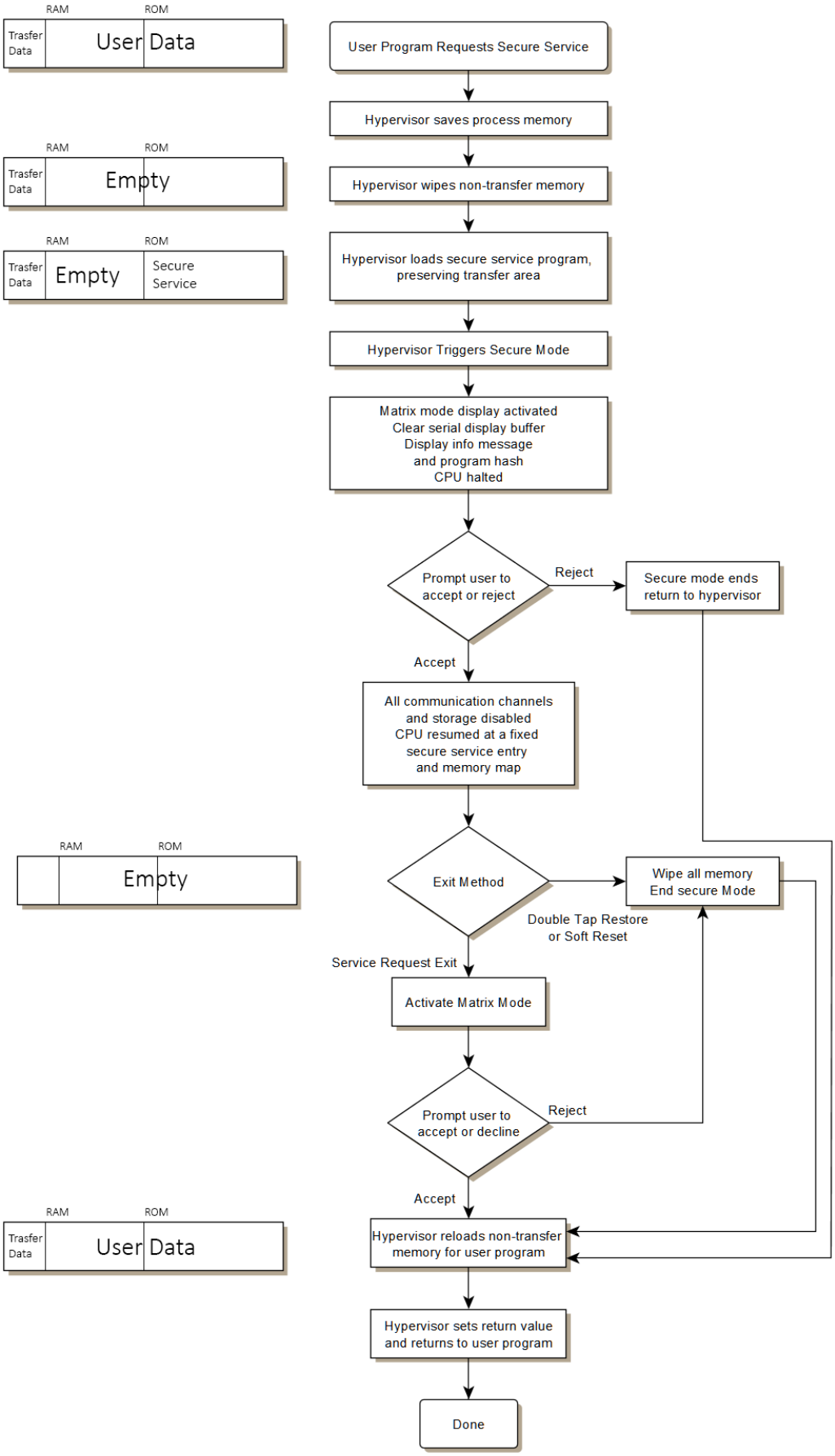

Figure 7. Secure mode flowchart

If the user is not happy with the state of the machine, they can type, in full, “reject” to abort the secure mode request. Control then returns to the hypervisor which reloads the state of the user program, a return value indicating an abort is set and control returns to the user program. If the user is happy with the state of the machine, they can type “accept” to continue. After this, the machine is confined to the secure compartment, access to communication channels and bulk storage is revoked, and trapping to the hypervisor is disabled. The program counter is set to a fixed secure service entry point, and the CPU is resumed, to start executing the secure program.

The secure program lets the user do their task, whether it’s typing up an email or document, or simply decrypting and viewing an encrypted email they have already received. When the user is ready to exit, the secure program will request to exit secure mode by writing to a hypervisor trap. The matrix mode overlay will once again be used to prompt the use to accept or reject the exiting of the secure container. The user can check all of memory to ensure that the transfer area is as expected. If the user accepts, secure mode ends and the hypervisor reloads the user program with the transfer area intact and sets a return value indicating successful exit. However, if the user decides to reject at this point, all memory is wiped before the user program is reloaded such that no sensitive data is leaked. Similarly, at any point while in secure mode if a soft reset is triggered by either the reset button, or double tapping the restore button, all memory is wiped before control is given back to the user program.

Changes need to be made to a variety of different parts of the machine in order to implement this feature. Implementation of some of these changes are discussed in the next chapter.

3.3 Summary

This chapter described the design and functionality of a method of inspecting machine state transparently and discussed the potential ways data could be exfiltrated from the system. Having defined an area which is to be contained, a method of transferring control to this secure compartment was conceptualised. The next chapter describes the attempt at implementing the individual modules that make up these systems, ending with discussion of the security of the system.

Chapter 4

Results and Discussion

As the goals of the project were largely involved with the implementation, this chapter presents the progress made towards implementing the matrix mode and secure mode on the MEGA65. It describes the modules which were developed, an overview of how they were implemented and tested, and whether the functionality requirements were met. Then ending with a discussion and analysis of how the overall system, when fully implemented, could protect against physical access attacks and which cases could be protected.

4.1 Matrix Mode

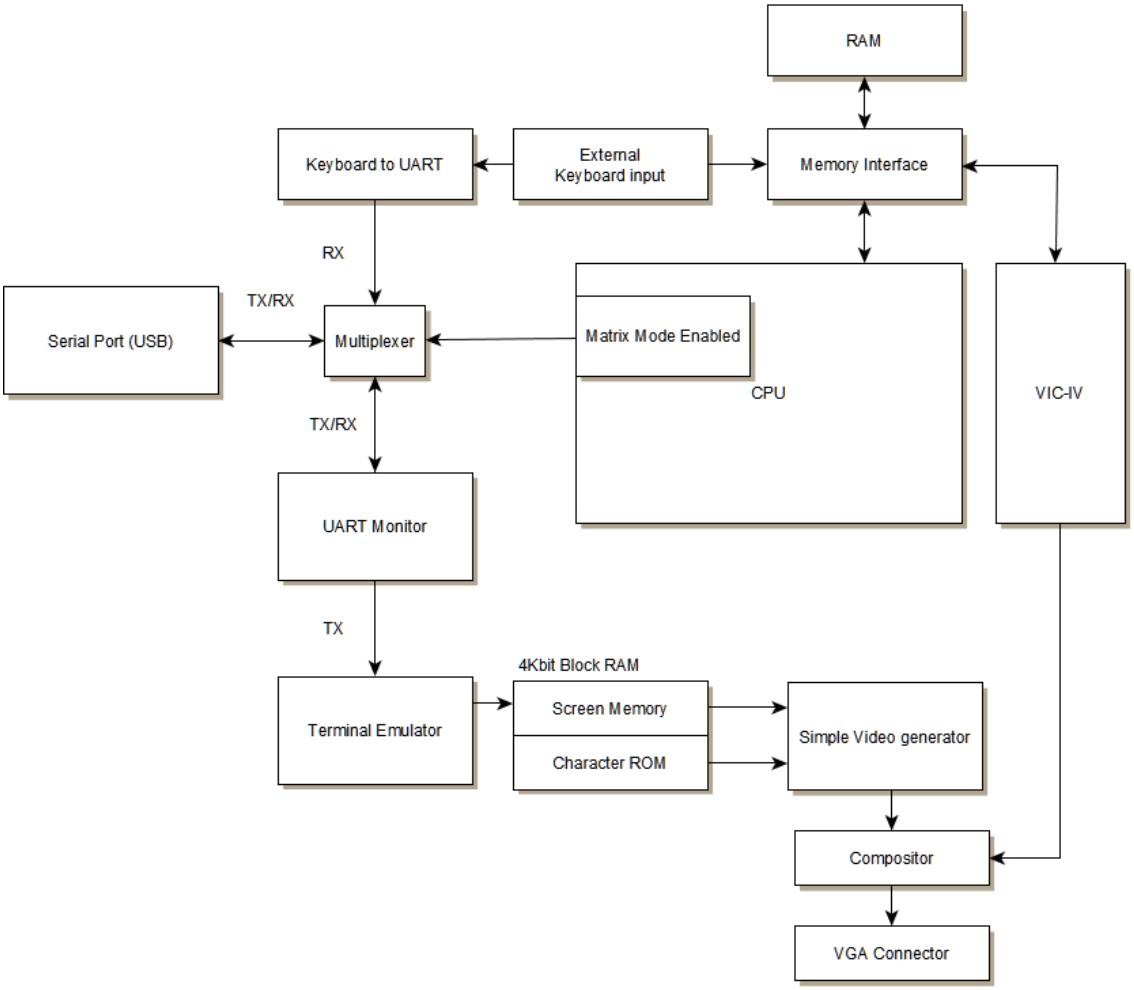

Figure 8. MEGA65 component block diagram

The matrix mode was envisioned as a method of on-board debugging and review. When a user presses a button combination, or a user program requests it, the screen is overlayed with a terminal monitor. This system gives the user the ability to review the data in the secure area before confirming the exit of secure mode.

The matrix mode is comprised of some new modules, a keyboard-to-serial converter, a terminal emulator, a 4K RAM block for the screen buffer and character ROM, and finally a video generator and compositor. These components can be seen in the block diagram in Figure 8.

Keyboard input will be redirected to a module which will convert the scan codes into serial output and then routed to the existing UART (**u**niversal **a**synchronous **r**eceiver/**t**ransmitter) receiver module of the serial monitor in lieu of the physical serial port. The UART monitor's transmit data will be redirected to a hardware-based terminal emulator, as opposed to running the terminal in software. The terminal emulator will convert the serial transmitted by the UART monitor module back into characters and stored in memory. The video generator will look up the character memory and output the associated character data to the display. The video generator will produce this overlay independent of the configuration of the existing video generator, the VIC-IV module, so that this type of output cannot be easily spoofed through malicious software. Output of both the simple video generator and the VIC-IV will be alpha-blended by the compositor and sent out to the monitor through the VGA connector or HDMI.

4.1.1 Keyboard to UART

The keyboard to UART is a module to convert PS/2 keyboard input into serial data that can be read by the existing serial monitor interface. The keyboard input for the MEGA65 project supports USB keyboards. To implement a USB host controller in the FPGA itself is not trivial and takes up logic space. To avoid this, the manufacturer of this particular development board, Digilent, opted to use a microcontroller with a built in USB host controller to convert the input into the simpler PS/2 interface. Figure 9 shows the connection from the PIC microcontroller to the Artix-7 FPGA on the development board.

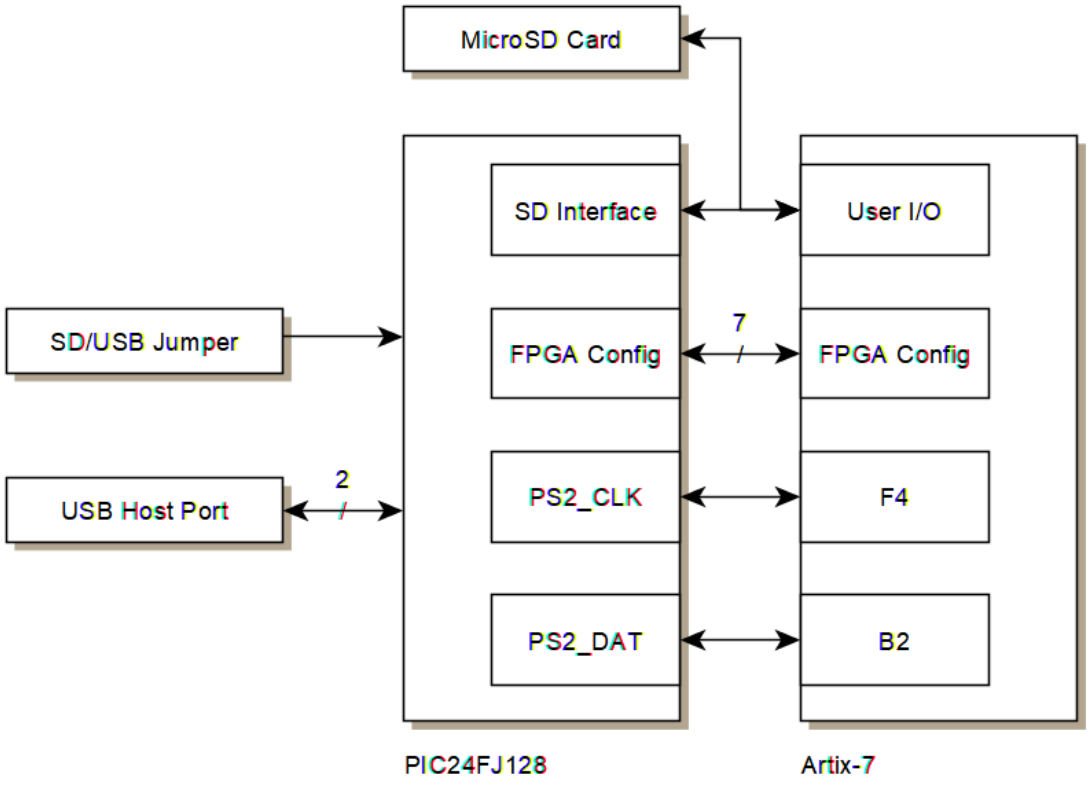

Figure 9. Nexys4 DDR PIC24 connections showing PS/2 connection

However, the PIC24FJ128 microcontroller is used for more than just the keyboard. It has an SD card interface and is capable of reading a configuration file from either the SD or a USB flash drive and configuring the FPGA with a bitstream.

After powering the device on, the PIC microcontroller will check whether the jumper is set to SD or USB, it will then look for the first file with the ".bit" extension on the selected device and configures the FPGA. If this microcontroller is reprogrammable it could present a potential security risk as it has direct access to configuration of the FPGA. For example it could be reprogrammed to configure the FPGA with a specific bitstream hidden elsewhere on the SD card, rather than the one the user wants. This method of configuration will likely not be present on the real MEGA65 hardware.

The MEGA65 also has, under development, support for real C64/C65 keyboards. By forcing the device to use only an internal keyboard in secure mode and not the USB keyboards could prevent any attacks using modifications to the keyboard such as embedded key logger hardware.

4.1.1.1 Implementation

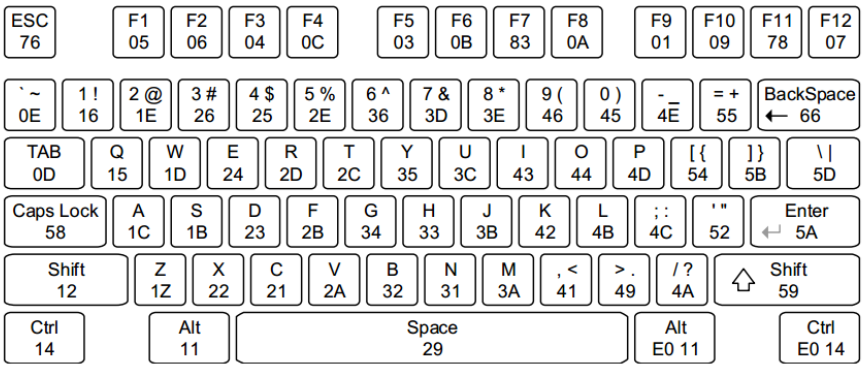

Figure 10. PS/2 Scan codes and the corresponding keys

The PS/2 keyboard interface sends a scan code when a key is pressed or released. “Make” codes are sent when a key is pressed. Make codes and the corresponding keys can be seen in Figure 10. When a key is released the “break” code is sent which is, in hexadecimal, 0xF0 and then the corresponding make code. Some scan codes are reused, such as the left and right control keys. To distinguish these keys an additional code is sent, E0, the extend code.

As the keyboard sends only the change in key state the host must keep track of what keys are down, compared to generic USB HID (**H**uman **I**nterface **D**evice) class devices where the host continuously polls to see what keys are held down. This allows any number of keys to be held down simultaneously depending on the implementation of the host.

An existing module, keymapper.vhdl, contains the state machine interfacing with the PS/2 data port. The module kept track of the last full scan code it received, taking into account extra commands such as whether or not it was a break, and whether it was extended.

Normally after reading the scan code from the PS/2 keyboard the module would update the CIA (**C**omplex **I**nterface **A**dapter). The CIA is the original chip used for I/O (**I**nput / **O**utput) in the Commodore 64 and includes support for the keyboard, joystick and internal timer. However when the keyboard is being used for the matrix mode we do not want input to also go to the CPU such that we don’t modify the state of the machine we’re trying to inspect. A disable flag was set such that it would not update the CIA with key presses while in matrix mode.

The last_scan_code was originally broken out to the CPU for debugging the keyboard layout. But this is all the information from the keyboard we need to implement the ps2_to_uart module, so this was routed through to the new module.

```
entity UART_TX_CTRL is
    Port ( SEND : in  STD_LOGIC;
           DATA : in  STD_LOGIC_VECTOR (7 downto 0);
           CLK : in  STD_LOGIC;
           READY : out  STD_LOGIC;
           UART_TX : out  STD_LOGIC);
end UART_TX_CTRL;
```

Figure 11. Interface of UART_TX module

An existing UART transmit module, UART_TX, was re-used. The interface can be seen in Figure 11. The module READY signal is asserted when the module is idle, data can be put on the DATA signal, and when the SEND signal is asserted it will initiate the serial transfer.

A simple state machine was setup for controlling the flow between waiting for a key, and outputting said key. While waiting for a key, the last_scan_code is checked to see if bit 12 is low, indicating a make code.

On any mainstream operating system, after holding down a key the key will start repeating itself. To maintain a familiar feel to the user it was decided that this should be implemented. If the make key remains the last_scan_code, it indicates that the key is held down. After around 200 milliseconds the key is repeated, if the key remains held the key will again be repeated after around 30 milliseconds. This helps in debugging scenarios where the user wants to step the CPU one instruction at a time, holding the enter key in single step mode, until something occurs on screen.

The full code for the implementation can be seen in Appendix A.

4.1.2 Terminal Emulator

A computer terminal is a physical hardware device, a keyboard and monitor, used to input data to, and display data from, a backend computer mainframe system or server. For non-graphical use physical hardware terminals are seldom used today, instead terminal emulators are used. These are software programs which emulate a terminal. A connected PC has a serial connection to the MEGA65 and using a serial terminal emulator we are able to display the monitor's command line console on the PC.

In a similar way this new module interfaces with the serial output of the MEGA65 to display the command line console, albeit directly on the MEGA65's display. This module is a program implemented in FPGA logic to behave like a terminal, hence in essence it is a hardware terminal emulator. Our terminal is used to control character value and position on the screen from data fed to it from the UART monitor module.

4.1.2.1 Implementation

Series 7 FPGAs from Xilinx have an amount, depending on the model, of dual-port block RAM, each capable of storing 32 kilobits (4 kilobytes) of storage (Mehta, 2012). Only a single 4 kilobyte block of memory was used for both the character ROM and the screen memory, to save on the already scare memory resources on the FPGA. The 4 kilobytes of memory are byte addressable such that a 12-bit address is required to address the 4096 elements. The character ROM is preloaded with pixel patterns representing each of the displayable characters. The screen memory contains information about which character is in each position on the screen.

To display the 96 different characters, each at 8x8 pixels, would require 8 bytes per character. Therefore, the character ROM takes up 768 bytes, leaving 3328 bytes left for screen memory. At 80 characters per line, this gives a horizontal resolution of 640 pixels which can divide the maximum horizontal resolution, 1920 pixels, evenly such that it can be scaled up exactly.

$$96 * 8B = 768B\ for\ Character\ ROM$$

$$4KB - 768B = 3328B\ left\ for\ screen\ memory$$

$$\frac{3328B}{80B/line} = 41.6\ lines\ \approx\ 40\ lines$$

Each character is mapped to the character rom with an 8-bit value, therefore each line requires 80 bytes. Given the 3328 bytes left at 80 bytes per line gives a possible 41.6 lines. However, to keep the numbers round, this was left at 40 lines, giving a 320 pixel vertical resolution, or 960 pixels when scaled. Screen memory is mapped to the bottom of the block, leaving 128 bytes free for additional characters if necessary.

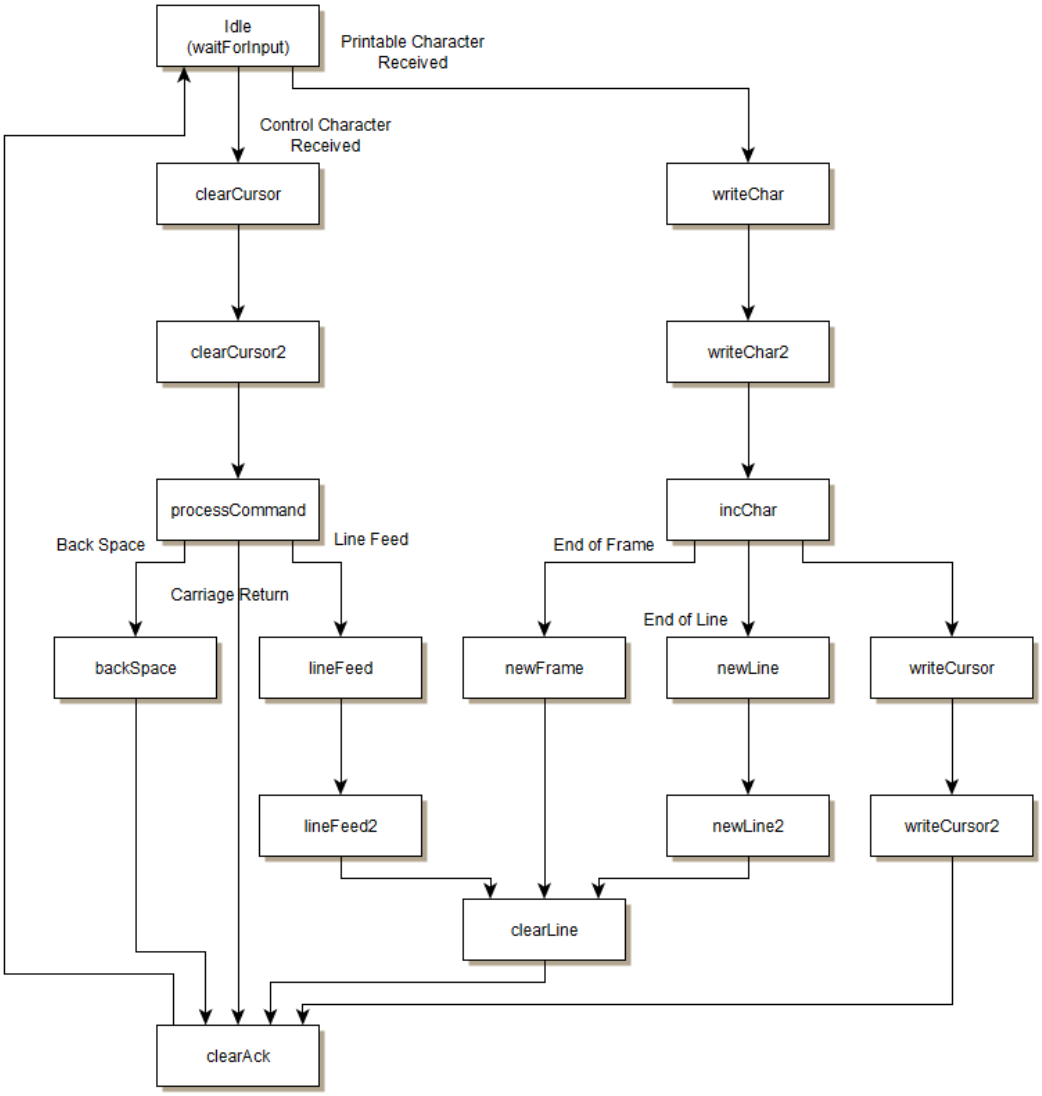

Figure 12. State diagram of the terminal emulator

The serial monitor does not output characters beyond the 96 standard ASCII characters, and control characters other than carriage return and linefeed. Since as less than 128 characters are represented, the 8th bit is not used, this bit was used to indicate whether the pixels should be inverted, providing a method of highlighting for a cursor.

The terminal emulator was implemented roughly as a finite state machine, as seen in Figure 12, attempting to be as simple as possible. While in its idle state, waitForInput, it waits until an input from the UART module is received. If the input is under 0x20, it indicates that it is a control character and is processed as such. If a printable character is received between 0x20 and 0x7E the character is written to screen memory.

Additionally, rather than shifting the entire contents of screen memory up every time the terminal scrolls, requiring many read and write operations, a pointer to the top of the frame in memory is provided to the output module, like a circular buffer. This means that no reading of memory is required for this module to function. Given that there is only a single read and a single write port, this is beneficial as this method would not impact reads of the character ROM in the video generator. However, as read operations are not available to the terminal emulator the cursor feature is limited and cannot be moved back over characters without losing them. This limitation is acceptable for now as the serial monitor module does not currently have any advanced cursor control.

The full code for the implementation can be seen in Appendix B.

4.1.3 Video Generator / Compositor

The purpose of the video generator is to overlay the terminal emulator on top of the existing output from the MEGA65. To do this the module needs to read the screen memory generated by the terminal emulator, and retrieve the associated pixel data from the character ROM. This is then alpha blended with the output with the original output from the MEGA65.

Additionally, there are three different modes of pixel scaling available to the user: 1x, 2x, and 3x scaling modes. Scaling mode 3x will fill the screen with the overlay, and modes 1x and 2x will provide a smaller window. This window can be moved around with the arrow keys. This provides flexibility to the user if they need to access any information behind the overlay.

4.1.3.1 Implementation

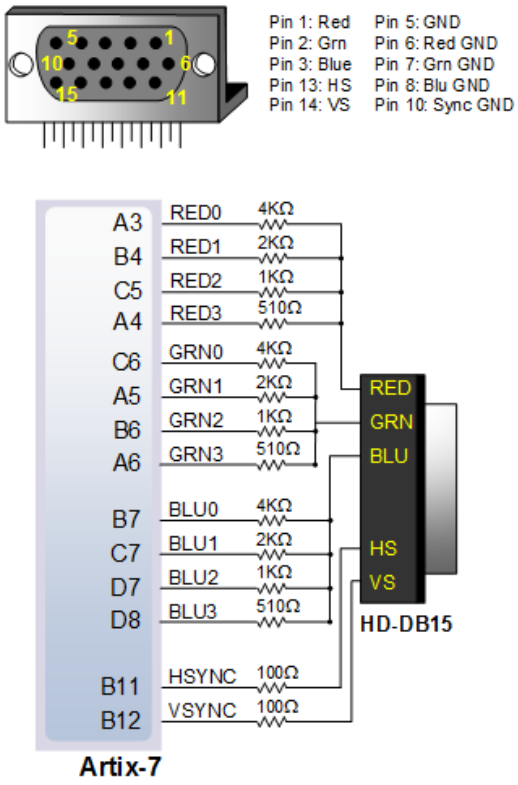

Figure 13. Nexys 4 DDR VGA interface (Digilent inc., 2016)

The VGA port on the FPGA development board provides a 12-bit (4-bit per colour) analog video output using a resistor DAC (**D**igital to **A**nalog **C**onverter) as shown in Figure 13. The VGA standard was developed back when cathode ray tubes were common, when the RGB signals controlled the deflection of electrons. This legacy format is still widely supported today. For more information about the VGA standard refer to (Digilent Inc., 2016).

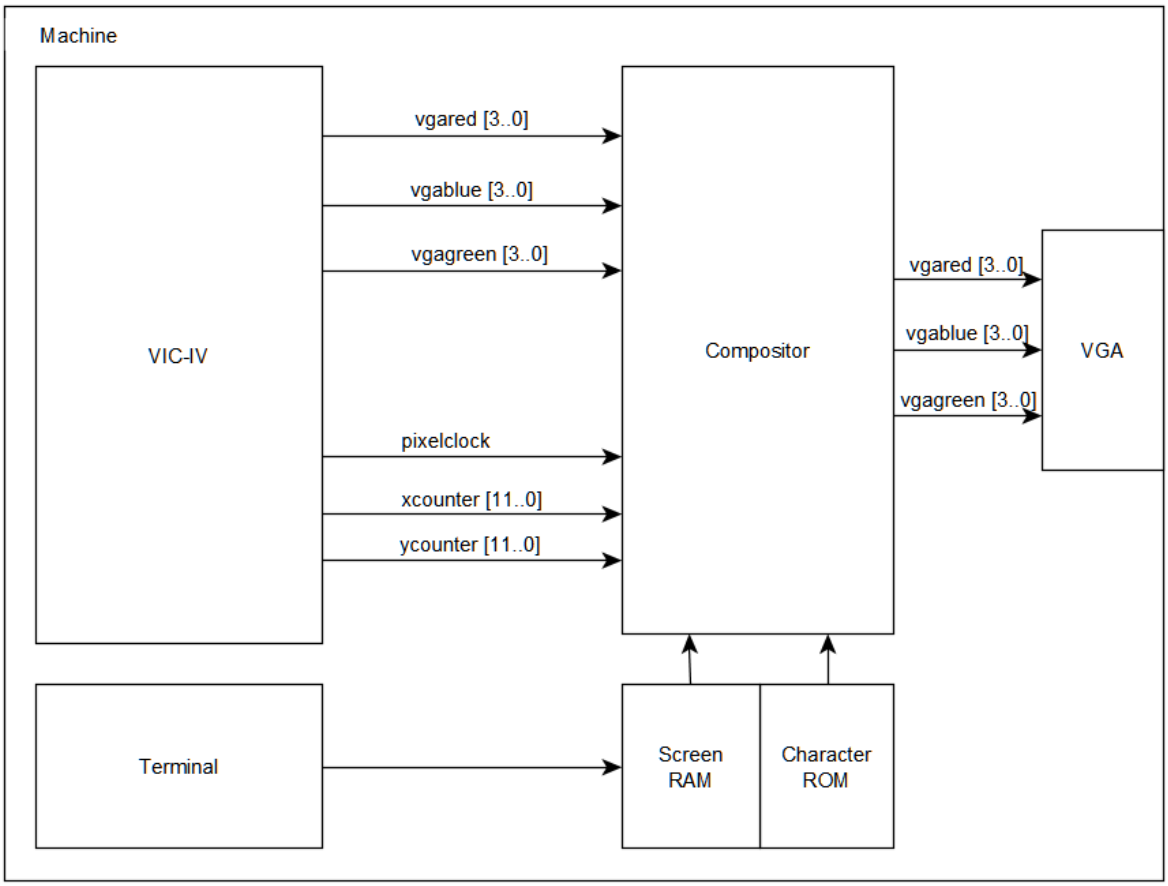

Figure 14. Signals between VIC-IV and the compositor module

The VIC-IV signals were routed through to the compositor module, and the output RGB signals from the compositor were routed to the VGA output, as shown in Figure 14. To synchronise existing output with the new video generator a 200MHz pixel clock and timing counters, xcounter and ycounter, were routed to the module. These counters provide the location of the current output pixel.

The main output generation process consists of two separate circuits, one which loads the next character data while another circuit sets the output pixel, taking into account the display mode.

```
case eightCounter is
  when b"00001" =>
    readAddress_rom<=charCount;
  when b"00011" =>
    charAddr<=dataOutRead_rom;
    invert<=dataOutRead_rom(7); --bit 7 is whether to invert or not.
  when b"00101" =>
    readAddress_rom<=(b"00" & charAddr(6 downto 0) & b"000")+charline;
  when b"00111" =>
    if charCount=CharMemEnd then
      charCount<=CharMemStart;
    else --otherwise increase
      charCount<=charCount+1;
    end if;
  when others =>
--do nothing;
end case;
```

Figure 15. Code snippet: Prepare next character data

The first circuit starts by retrieving the data from the screen memory in the position of the *next* character we want to read. From this, the bottom 7-bits concatenated with a line offset value becomes the address for the pixel data, read from the character ROM. If the 8th bit was set, this indicates that the character should be inverted, and an invert flag is set. The character counter is incremented after retrieving the character data. This is show in the code snippet in Figure 15.

```
--If it hasnt just loaded new data,
if eightCounter/=end_of_char then
  eightCounter<=eightCounter+1; --increment counter
  if bufferCounter=mm_displayMode then
    data_buffer<=data_buffer(6 downto 0)&'0';
    bufferCounter<=b"00";
  else
    bufferCounter<=bufferCounter+1;
  end if;

elsif eightCounter=end_of_char then
  --clear end of frame flags anywhere before end of frame
  doneEndOfFrame<='0';
  doneEndOfFrame1<='0';
  doneEndOfFrame2<='0';
  eightCounter<=b"00001"; --Reset counter
  if invert='1' then --invert flag, negate data.
    data_buffer<= not dataOutRead_rom; -- grab new data
  else
    data_buffer<= dataOutRead_rom; -- grab new data
  end if;
end if;
```

Figure 16. Code snippet: Shifting new data in buffer

The second circuit checks whether the current character has finished being output, and if it has, moves the new data to the buffer, and if it hasn't then the buffer is left-shifted depending on the display mode. Each pixel output is either held for 1, 2 or 3 pixelclock cycles to scale up the size. This is shown in the code snippet in Figure 16.

```
if data_buffer(7) = '1' then
  redOutput_all <=   b"00"&vgared_in(7 downto 2);
  greenOutput_all <= data_buffer(7)&data_buffer(7)&data_buffer(7)&vgagreen_in(4 downto 0);
  blueOutput_all <=  b"00"&vgablue_in(7 downto 2);
else
  redOutput_all <= b"00"&vgared_in(7 downto 2);
  greenOutput_all <= data_buffer(7)&data_buffer(7) &vgagreen_in(5 downto 0);
  blueOutput_all <= b"00"&vgablue_in(7 downto 2);
end if;
```

Figure 17. Code snippet: RGB outputs of the compositor

The top three bits of the green output are set to the value in the 8th bit of the data_buffer, as shown in Figure 17. Much of the rest of the code is related to timing, defining the area for the windowed modes, and line repeating for vertical scaling. The timing and display issues were fixed mainly using trial and error.

The full code for the implementation can be seen in Appendix C.

4.1.4 Testing

Testing was performed at many stages of development. The approach was to start with the components where the output could be viewed such that there was a feedback loop. For example, the Keyboard to UART module was implemented first as the keyboard output could be easily monitored by the existing remote serial monitor. Similarly, the video generator was developed before the terminal emulator as the video overlay could be seen directly on the connected display.

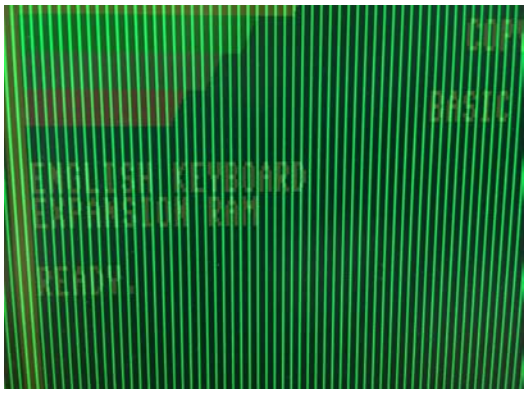

Figure 18. Screenshot of initial testing of generated output over existing video output

Figure 18 shows the initial testing of the compositor with green vertical lines overlaying the existing video output. This was only testing pass-through of the video output.

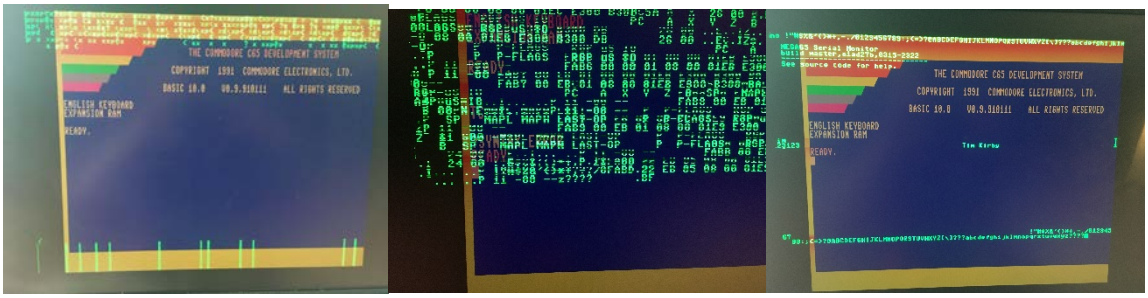

Figure 19. Screenshots of initial testing of the terminal emulator / compositor

Initially there were some minor problems with the video generator getting the correct characters, issues with drawing the characters to the screen and issues with timing and alignment of the video generator with the existing video output, as seen in Figure 19. Through trial and error, these issues were fixed over time.

With the components which make up the matrix mode substantially completed, the full functionality of the matrix mode can be shown.

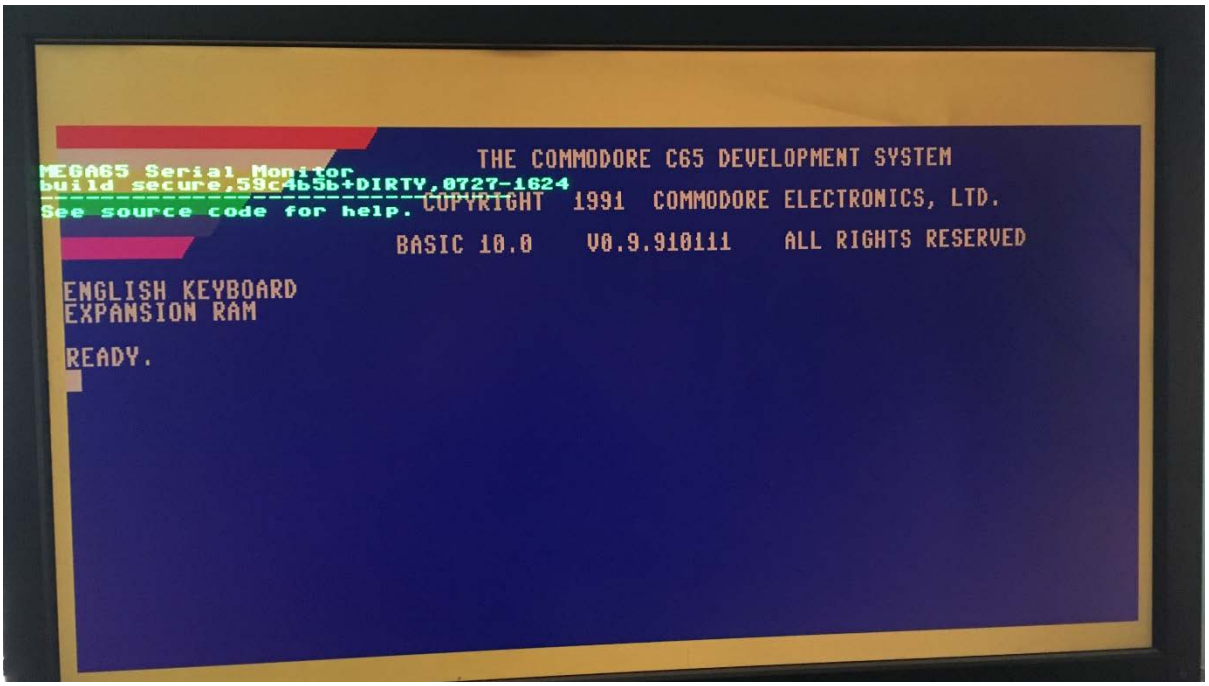

Figure 20. Matrix Mode, Full screen, 3x Scaling

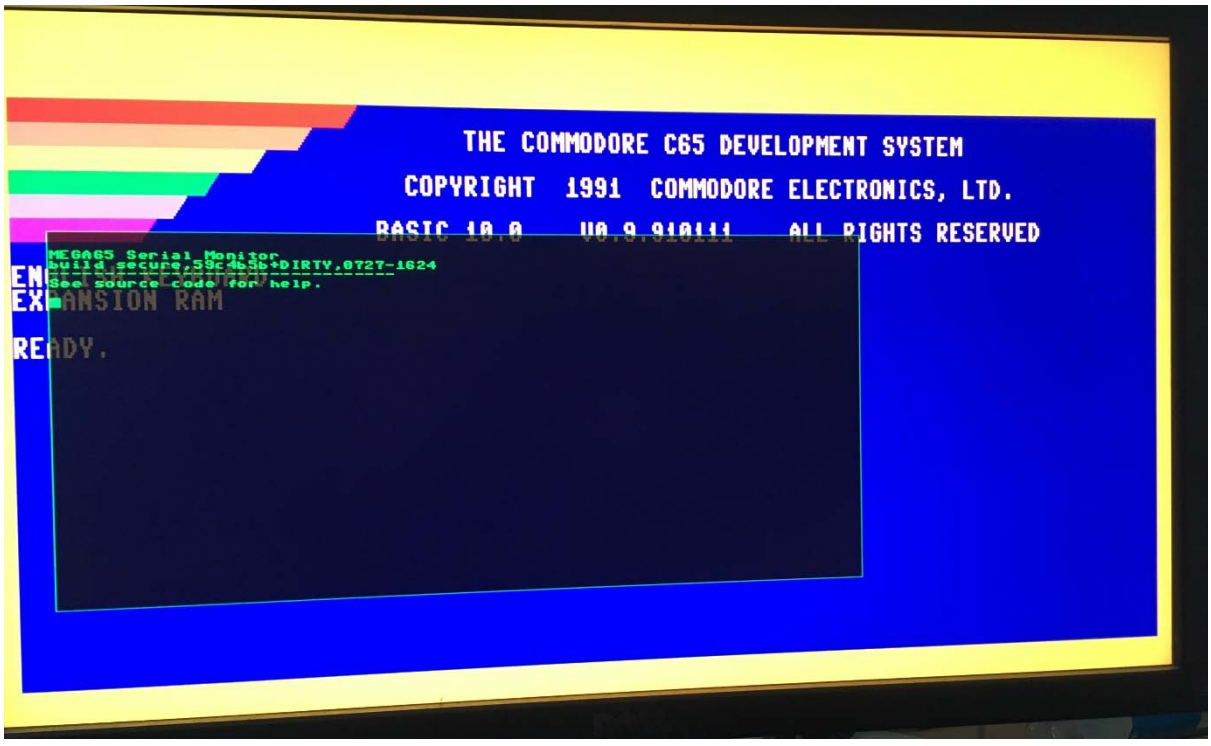

Figure 21. Matrix Mode, 2x Scaling

Figure 22. Matrix Mode, 1x Scaling

Figures 20 through 21 show the different scaling modes which were implemented. The 2x and 1x windowed modes can be moved around using the directional arrow keys.

Figure 23. Matrix Mode, memory read and write

In the existing serial monitor there are some useful commands which allow the user to interact with memory or control the state of the CPU. There are four commands to read from memory: “D”, “d”, “M” and “m”, and two commands to write to memory: “s” and “S”. There are three commands to control the CPU state: “t0”, “t1” and “tc”.

The “d” command, followed by a 16-bit address, displays the memory in that location, as seen from the current CPU memory map. The “m” command displays the memory with a 28-bit memory address. Similarly, the “s” command sets the value of memory, using up to a 28-bit memory address. By putting a space between each hexadecimal input, it writes to successive memory addresses starting at the location specified.

For example, by entering the string “s400 14 9 D 20 B 9 12 2 19” the authors name was printed in the top left-hand corner of the screen, show in Figure 23. $0400 is the location of screen RAM in C64 mode, the following hexadecimal values are related to the PETSCII characters, the character set used in the Commodore line of computers.

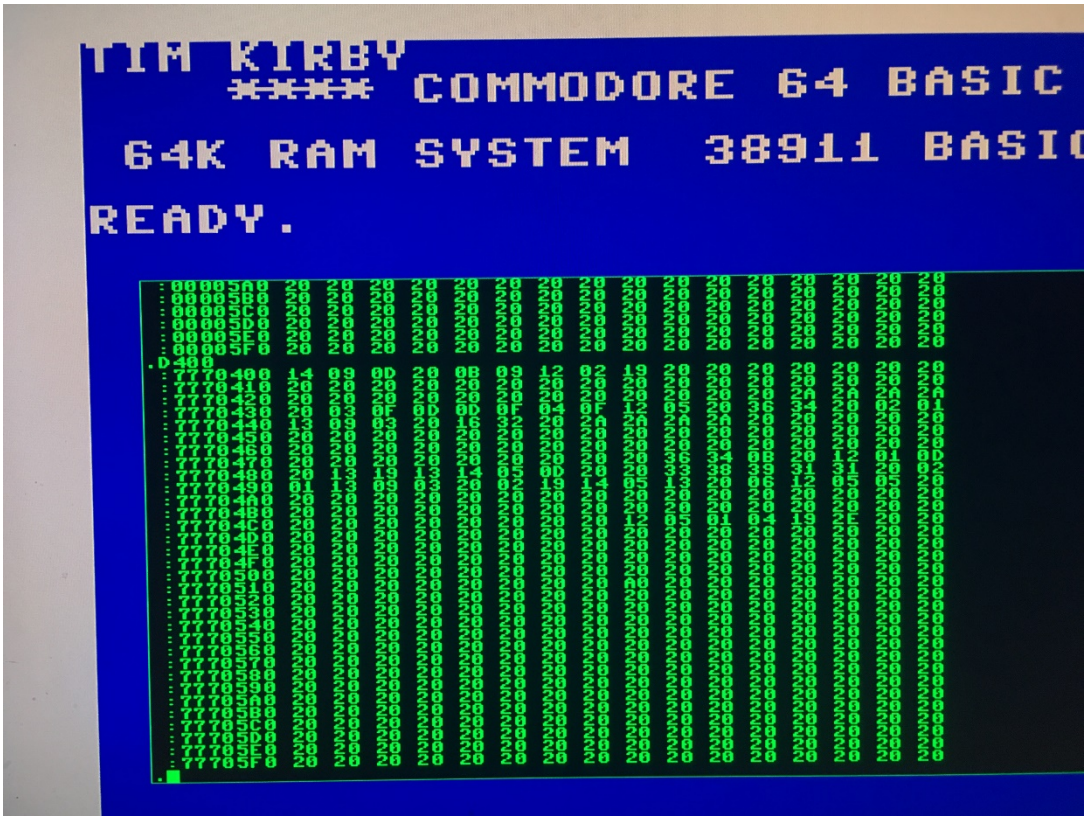

Figure 24. Matrix Mode, "D" command

The “D” or “M” command can be used to display 512 bytes of memory from a given address, this can be used to quickly view areas of memory. Issuing the command again without an address will show the next 512 bytes. The example shown in Figure 24 uses the “D” command to display 512 bytes of the screen RAM starting at $0400.

By pressing enter or entering the “r” command will print the contents of the machines registers, flags, current memory map, and the last operation. This is shown in Figure 25.

The commands “t0”, “t1” and “tc”, refer to CPU running state, where “t0” is free running, “t1” is freeze CPU, and “tc” is continuous trace. In continuous trace mode the system reports the status of the CPU, similar to the “r” command, for every cycle of the CPU. However, this adversely affects the systems performance.

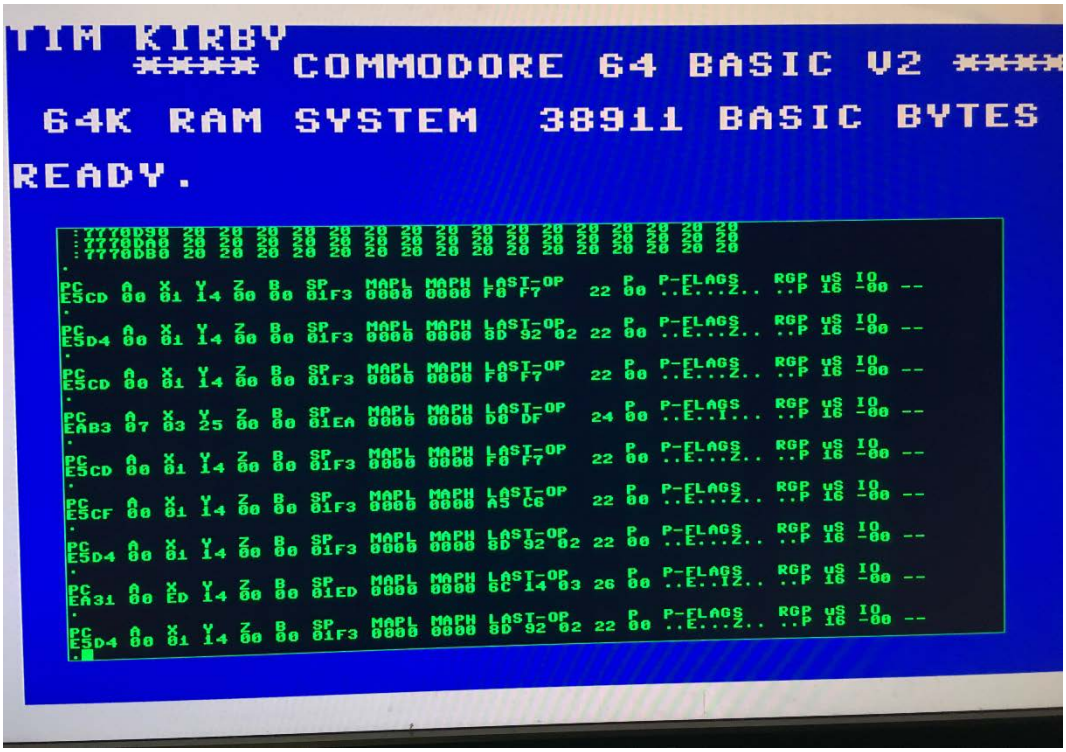

Figure 25. Matrix Mode, "r" command, showing CPU registers, operations

While most of these features were already present in the existing remote serial monitor, the ability to utilise the terminal with the internal display allows this to meet our requirements for the out-of-band machine state inspection mechanism. Additionally, the use of a transparent overlay makes it impossible to spoof using software, as there is no such graphic mode that supports this in the MEGA65. In the final hardware there will be an indication LED to remove any residual risk of spoofing the Matrix Mode from software.

4.2 Secure Mode

To implement the design of the secure mode as discussed in Chapter 3, changes to the architecture need to be defined. This system would require the following architectural changes:

- Changes in the Hypervisor ROM
    - Trap to enter and exit from the secure container
    - Routines to save and load the tasks from SD card (Task Switcher)
- Changes to the serial monitor
    - A command to clear the screen
    - Write the banner prompt to screen
    - Handling of user input for accepting and rejecting
    - Interface with CPU to confirm entry/exit from secure mode
- Changes in the CPU
    - Interface with monitor to confirm entry/exit from secure mode
    - New CPU states to handle secure mode events:
        - Erasing all memory on rejection or reset
        - Erasing non-transfer memory on exit
        - Triggering the matrix mode on entry/exit
    - Disable trapping to hypervisor
    - Disabling access to I/O (SD, Ethernet, UART) (Secure Compartments)
- Miscellaneous Additions
    - A hashing algorithm, SHA-3
    - An encryption algorithm, AES

Due to the time limitations of this project, not all of these features were able to be implemented. This section summarises the progress that was made.

4.2.1 Task Switcher

Due to the memory (256KB) and processor constraints (50Mhz, 8-bit), the MEGA65 would not be able to run many independent processes at the same time. The task switcher was proposed as a way of switching between whichever recent tasks the user has been using, while only ever actively running a single task at a time. The secure mode concept required the ability for the machine to switch in and out tasks to and from the SD card. As the MEGA65 can only run a single process at a time, a single task can be defined by the entire machine state. As the complete task switching interface is quite a large undertaking, only the preliminary development was within the scope of this project. The aim of the preliminary development for task switching was to try and get machine state to save and load from SD card.

To simplify the moving of bulk data the machine state was mapped to a contiguous address space in memory. The machine state consists of all the memory, CPU registers, and I/O configuration data. There are some intricacies in the current design of the CPU which make exposing this machine state difficult. For example, a certain configuration would be set if a register was written to with certain values twice (like a secret knock), or when an I/O register has different functions when it is read than when it is written, as well as write-only and read-only registers. These configurations are not possible to restore easily unless the underlying configuration registers are exposed. To find and fix all of these internal registers was out of scope for this project.

4.2.1.1 Implementation

The MEGA65 CPU has an enhanced 28-bit address space, compared to the 20-bit C65 or the 16-bit C64, therefore a total of 256MB of memory can be addressed. The address chosen was $7F00000 to $7FFFFFF, this provided us with 1MB of data. Enough to map the memory and later additional data such as a thumbnail image of the task.

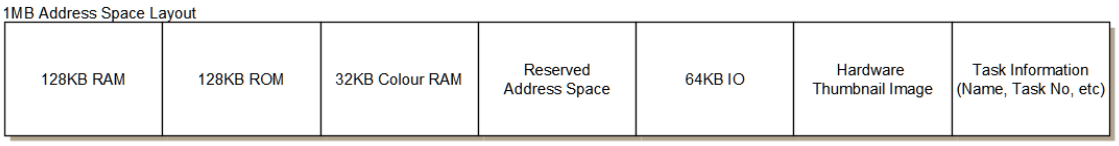

Figure 8. Proposed layout of the machine state in memory

To re-map the already exposed registers, the addresses were intercepted in the read_long_address procedure in the CPU. There is an intermediate variable real_long_address, and if this is within the range of $7F0 0000-$7F1 FFFF, for example, it would modify the long_address to $000 0000 - $001 FFFF, and therefore pointing to the actual RAM address.

Initially these were mapped to the following:

Type	From Address	To Address	Amount
RAM	$7F0 0000 - $7F1 FFFF	$000 0000 - $001 FFFF	128K
ROM	$7F2 0000 - $7F3 FFFF	$002 0000 - $003 FFFF	128K
Colour RAM	$7F4 0000 - $7F4 8000	$FF8 0000 - $FF8 FFFF	32K

This was done to remap and exclude registers as necessary. Some important I/O registers were also mapped, such as the CIA1/2 chips. This method only works if the memory is already mapped to an existing address.

4.2.1.1.1 SD Card Reading and Writing

The original SD card specification calls for a fixed sector sized of 512 bytes. The SD card that the author used was a 512MB MicroSD card, giving a capacity of 510,132,224 bytes or 996,352 sectors. There is an existing generic VHDL SD card controller module (sd.vhdl) as well as module to interface the CPU with the SD controller (sdcardio.vhdl). Currently the interface only supports single sector reads or writes.

To read or write to the SD card there are some registers that need to be set. There is a 512 byte SD card buffer mapped at $FFD3E00, this holds the data to be written to, or read from, the SD card. There are also four registers at $FFD3681 to $FFD3684 which determine which sector to read or write to.

The control register is mapped to $FFD3680. With the address registers filed in correctly, writing to the control register with a value of 0x03 will initiate a write to the SD card. Similarly, writing 0x02 to the control register will initiate a read from the SD card. Reading from the control register provides the status information. These registers and their associated addresses are listed in Table 1.

Table 1. SD Card interface registers

Registers / Memory	Absolute Address	Relative Address
SD Buffer (512B)	$FFD3E00	$DE00
Control Register	$FFD3680	$D680
Address Byte 0	$FFD3681	$D681
Address Byte 1	$FFD3682	$D682
Address Byte 2	$FFD3683	$D683
Address Byte 3	$FFD3684	$D684

4.2.1.1.2 DMAgic DMA Controller

We wish to copy data from the remapped address range to the SD card. Using the CPU to copy a large amount of data would take many cycles and program instructions. To avoid this the C65 introduced a hardware DMA controller called the DMAgic (Bowen, et al., 1991). The DMA in the MEGA65 has access to the full 28-bit address space, allowing the movement of data independent of the current memory map.

The DMAgic controller uses a list-based method of fetching DMA command sequences. The list contains the command, the source and destination addresses, and the number of bytes to process. The list is simply sequence of bytes written anywhere in RAM, the program tells the DMA where to look for this list. There are three main commands, copying: copies a block from one area to another, swapping: exchanges the contents of two blocks of memory, and filling: fills a block with a source byte. Table 2 shows the DMA list bytes and a description of their function.

Table 2. DMA list description

DMA List byte number	Description
Byte 0	Command
Byte 1	Lower byte of copy size
Byte 2	Upper byte of copy size
Byte 3	Lower byte of source address (bits 0-7)
Byte 4	Upper byte of source address (bits 8-15)
Byte 5	Bank byte of source address (bits 16-19)
Byte 6	Lower byte of destination address (bits 0-7)
Byte 7	Upper byte of destination address (bits 8-15)
Byte 8	Bank byte of destination address (bits 16-19)
Byte 9	Modulo Byte Lower (not used)
Byte 10	Modulo Byte Upper (not used)

To use the DMA controller, in a similar way to the SD card interface, specific registers are to be written to. There are six main registers, 4 bytes to locate the DMA list, and two to provide the upper 8 bits for the source and destination address. Table 3 shows these registers and their relative address while in the hypervisor.

Table 3. DMAgic hardware registers

Register	Description	Relative Address
DMA Source MB	Upper 8-bits of source address, Write Only	$D705
DMA Destination MB	Upper 8-bits of destination address, Write Only	$D706
List Address Low	Absolute location of list, bit 0-7, Write Only (Writing initiates the DMA transfer)	$D700
List Address High	Absolute location of list, bit 8-15, Write Only	$D701
List Address Bank	Absolute location of list, bit 16-23, Write Only	$D702
List Address MB	Absolute location of list, bit 24-27, Write Only	$D704

4.2.1.1.3 Hypervisor Routines

There were some hypervisor routines developed to test the reading and writing of machine state to the SD card. For ease of development the reading and writing of SD were temporarily assigned to the traps already in place for double tapping the restore key, and the alt-tab button combination. When the alt-tab button combination is pressed, the state is written to SD, when the restore key is tapped twice, the state is read from the SD.

The general structure of these routines is given as pseudocode in Figures 26 and 27.

Start

1. Reset SD card
2. Reset sector counter
3. Initialise addresses for list, source and destination
4. Initialise DMA address list location
5. While sector counter < 512
    a. Initiate DMA to transfer 512 bytes from machine state to SD buffer
    b. Write sector number to SD control register
    c. Initiate write to SD
        i. While SD is busy
            1. Wait
    d. Increment destination address
    e. Increment bytes counter by 512
    f. Increment sector counter
6. Reset counters

End

Figure 26. Pseudocode overview of reading from SD card

Start

1. Reset SD card
2. Reset sector counter, byte counter
3. Initialise addresses for list, source and destination
4. Initialise DMA address list location.
5. While sector counter < 512
    a. Initiate read from SD
        i. While SD is busy
            1. Wait
    b. Initiate DMA to transfer 512 bytes from SD buffer to machine state
    c. Write next sector number to SD control register
    d. Increment destination address
    e. Increment byte counter by 512
    f. Increment sector counter
6. Reset counters

End

Figure 27. Pseudocode overview of writing from SD card

The full code can be seen in Appendix D.

4.2.1.1.4 Testing and Issues

The SD card writes were tested by taking an image of the SD card with a card reader, and inspecting it with a hex editor. This machine state was compared to the memory as dumped using the serial monitor. By comparing these we can tell whether there were any issues when writing to SD card. Similarly, when testing reading, the memory was checked from the serial monitor to see whether it matched what was on the SD card.

The sector on the SD card to write to was set to start at sector 7816,250, or at around 400MB, this is seen in Figure 28.

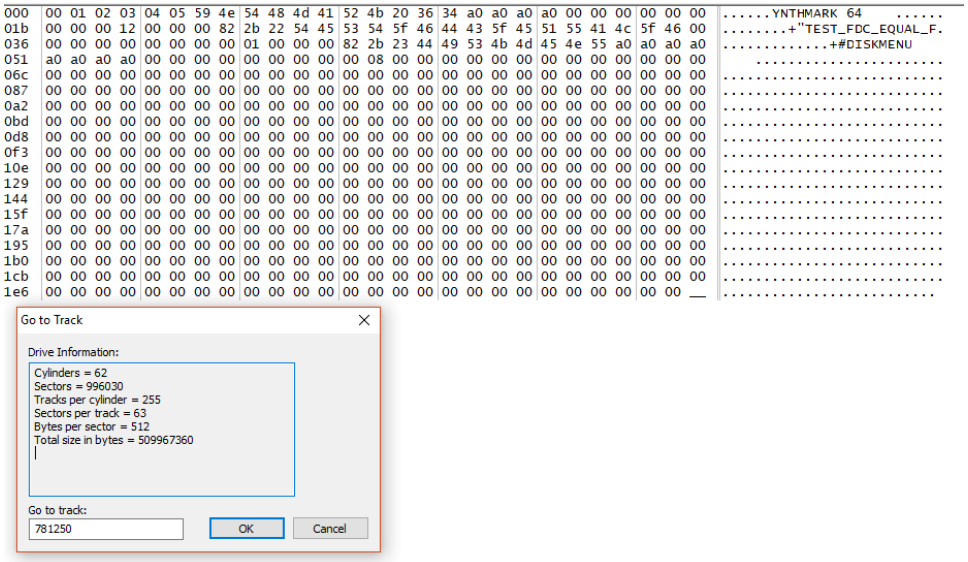

Figure 28. State written to sector 781,250 on SD card

It was found that the machine state, as mapped, was saved successfully. But the mapping was still missing some internal configuration states. The time taken to save state was also quite long, 16 seconds to save state, and 3 seconds to restore state, this was likely because the SD card interface is only implemented using single-sector reads and writes.

There were some other issues encountered, for example when the state was loaded the screen would appear a garbled mess. This was found to be caused by I/O registers which performed different actions when written to than the value they give when read.

There were many instances of this in the CIA modules such that anything that relied on timers or interrupts would break, such as the cursor blink. For example, in the original C64, the CIA 1 registers consist of 16 bytes mapped at $DC00-$DCFF, with the registers mirrored every 16 bytes. However, in the MEGA65 implementation, each mirrored set of registers was 32 bytes, consisting of the normal 16 bytes plus 16 extended registers originally for debugging. These

extended registers were only implemented for reading data but were still mapped to their original function when they were written. This meant that the registers were overwritten with whatever data was in the debugging registers. To fix this these issues, the debug registers where modified to expose the internal configuration, such that these could be directly read and written. Figure 29 shows the read function of these extended registers, which now expose the internal registers of Timer A and the interrupt mask.

```
------Read Extended Regs:
when x"10" => fastio_rdata <= reg_timera(7 downto 0);
when x"11" => fastio_rdata <= reg_timera_latch(7 downto 0);
when x"12" => fastio_rdata <= reg_timera(15 downto 8);
when x"13" => fastio_rdata <= reg_timera_latch(15 downto 8);

when x"14" => fastio_rdata <= reg_timerb(7 downto 0);
when x"15" => fastio_rdata <= reg_timerb_latch(7 downto 0);
when x"16" => fastio_rdata <= reg_timerb(15 downto 8);
when x"17" => fastio_rdata <= reg_timerb_latch(15 downto 8);

--Direct read interrupt mask
when x"18" =>
  fastio_rdata(7 downto 5) <= b"000";
  fastio_rdata(4) <= imask_flag;
  fastio_rdata(3) <= imask_serialport;
  fastio_rdata(2) <= imask_alarm;
  fastio_rdata(1) <= imask_tb;
  fastio_rdata(0) <= imask_ta;
```

Figure 29. Modified extended CIA registers

A similar issue was found in the CIA 2 module, which caused graphical artefacts when loading state, as shown in Figure 30.

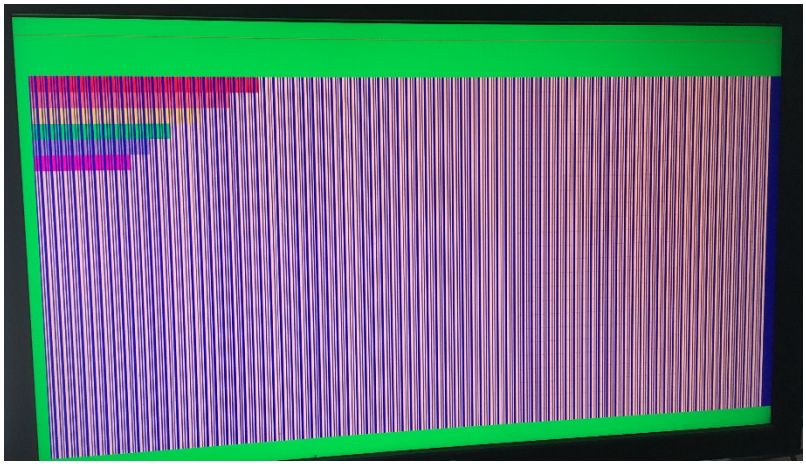

Figure 30. Graphical artefacts on loading state

The first two bits of the CIA 2 port A register controls the position of the memory for the VIC. Manually setting this register to the default value using the remote serial monitor was found to fix these display issues, but a proper solution is yet to be implemented. Debugging all of these issues was tedious and a lot of time was spent hunting down bugs of this nature.

4.2.2 CPU Modifications

There needs to be some modifications to the CPU core itself such that the various components can come together to create the secure container. Figure 31 shows the extra states created to facilitate secure mode. Due to the scattered nature of these changes, full code cannot be provided in this document, but is available on the GitHub repository. The URL can be seen in Appendix E.

Firstly, the trap entry point for the secure mode was selected to be $D672. When this is written to by the user program, the hypervisor routine will initiate the saving and loading of task state to SD, and then it will exit when $D672 is written to again.

```
--Secure Mode states
EnterSecureMode, EnterSecureMode1,
ReturnFromSecureMode, ReturnFromSecureMode1,ReturnFromSecureMode2,
SecureWipeAll, SecureWipeNonTransfer,SecureWipeNonTransfer1,
```

Figure 31. Code snippet, showing secure mode states in the CPU

The CPU was modified such that if $D672 is written to while in hypervisor mode it initiates a request to enter secure mode, using the flag secure_mode_pending, as seen in Figure 32.

```
if hypervisor_mode='1'
  and memory_access_address(5 downto 0) = "110010" then
  state<=ReturnFromHypervisor;
  secure_mode_pending<='1';
end if;
```

Figure 32. Code snippet, showing state change when writing to $D672 from Hypervisor

The hypervisor then exits normally, restoring the user mode CPU registers, then in the next cycle the state changes to the secure_mode state. While in this state, it enables the matrix mode overlay and sends a request to the serial monitor to display a prompt on the screen. It then moves to the next state secure_mode1 where it waits, effectively halting the CPU, until the user confirms that they want to accept or reject the pending secure mode request.

```
if secure_confirm_in = "01" then --accept
  reg_pc <= x"8000";
  reg_mb_low <= x"00";
  reg_mb_high <= x"00";
  reg_map_low <= "0000";
  reg_map_high <= "0001"; --Block 4 Mapped
  reg_offset_low <=x"000";
  reg_offset_high <= x"160"; --24000-8000 = 16000
  secure_request_out <= "00"; --de-assert request
  state<=normal_fetch_state;
elsif secure_confirm_in = "10" then --reject
  --End secure mode, hypervisor reloads non-transfer memory for user program
  hyper_protected_hardware(7)<='0';
  hypervisor_trap_port<="1000100"; --$44
  secure_request_out <= "00";
  state <= TrapToHypervisor;
end if;
```

Figure 33. Code snippet, showing the result of accept and reject

If the request is accepted, then the program counter register is set to $8000, which is mapped to the absolute address $24000, and the state is resumed to the normal_fetch_state. If the user rejects at this point control is given back to the hypervisor to reload the user program. This can be seen in Figure 33.

```
when ReturnFromSecureMode =>
  hyper_protected_hardware(6)<= '1'; --enter MM
  secure_request_out <= "10"; --request exit
  transfer_size <= reg_x;
  state<=ReturnFromSecureMode1;

when ReturnFromSecureMode1 =>
  if secure_confirm_in = "01" then --accept
    state<= SecureWipeNonTransfer;
  elsif secure_confirm_in = "10" then --reject
    state<= SecureWipeAll;
  end if;
```

Figure 34. Code snippet, Return from secure mode

When the user is ready to exit secure mode the program can do so by writing to $D672. It goes to a state called ReturnFromSecureMode, seen in Figure 34. The matrix mode overlay is enabled and a request to confirm exit is sent to the serial monitor. Additionally, the size of the transfer area can be defined by program by setting the CPU register reg_x, signifying the number of kilobytes to keep. Once the user is okay with the contents of the transfer area and they confirm, the machine goes to a state called SecureWipeNonTransfer. This state should wipe all memory, other than the transfer area, and exit secure mode then trap to the hypervisor to restore the user program. If the user rejects at this point the state SecureWipeAll will wipe all of memory, including the transfer area, and the hypervisor will then restore the user program.

4.2.2.1 Secure Compartments

The secure compartment itself is constructed by denying access to certain I/O during secure mode. Figure 35 shows where the iomapper modules fits in between the IO peripherals and the memory controller in the CPU. As its name suggests, the iomapper maps the I/O registers to memory addresses accessible to the CPU.

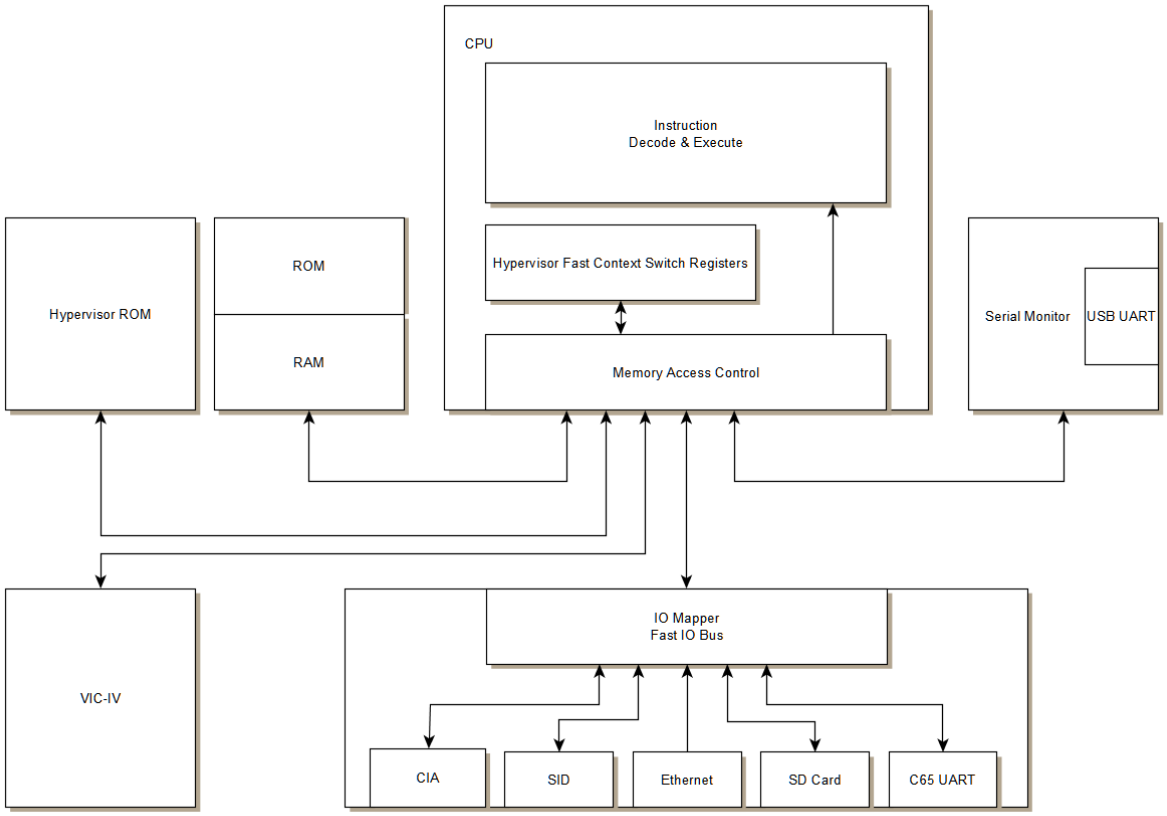

Figure 35. Connections to the memory access control

To restrict these I/O we can route out the secure mode flag to the iomapper module. A condition can be put on the chip select line such that the peripheral cannot be selected when secure mode is active. The example in Figure 36 shows that sectorbuffercs can only be asserted when secure_mode_active is disabled.

```
if address(19 downto 16) = x"D"
  and address(15 downto 14) = "00"
  and address(11 downto 9)&'0' = x"E"
  and sector_buffer_mapped_read = '1' and colourram_at_dc00 = '0'
  and secure_mode_active = '0' then
  sectorbuffercs <= '1';
  report "selecting SD card sector buffer" severity note;
end if;
-- Also map SD card sector buffer at $FFD6000 - $FFD61FF regardless of
-- VIC-IV IO mode and mapping of colour RAM
if (address(19 downto 8) = x"D60" or address(19 downto 8) = x"D61")
and secure_mode_active='0' then
  sectorbuffercs <= '1';
end if;
```

Figure 36. Chip Select lines for the SD card buffer

A similar technique can be used to restrict any of the I/O chips. The important features being SD card, and Ethernet. The CPU modifications and the secure compartments were not substantially completed in time to the point where further testing could be performed.

4.3 Discussion

The implementation of the matrix mode and related subsystems were substantially completed to the point where they meet the requirements set for the out-of-band machine state inspection mechanism. A rudimentary task switching system was completed to some extent, with outstanding issues with solutions which are out of the scope of this project. Because of this roadblock, implementation and testing on the rest of the architectural modifications and the overall secure mode system was not fully completed. The system can still be discussed assuming these components will be completed as designed.

The concept of secure mode helps protect against unauthorised programs from being executed in the secure container. In a physical attack, the hypervisor, user task or secure service can easily be compromised as these are stored unprotected on the SD card. The system protects against attacks exploiting these files by allowing users to inspect memory, through a dedicated channel, prior to entering the container. The validity of the secure service can be checked by comparing the hash with the known good hash. If it is not what the user is expecting they can abort. Additionally, given that the user can inspect the transfer area before exiting the container, the user knows the content which could possibly be exfiltrated. These functions put some onus on the user to know the software and what to expect.

The hypervisor or secure service cannot deconstruct the secure container, without the user knowing, as this is controlled by the hardware. For example, the hypervisor cannot load the secure service without requesting secure mode, as the user would surely notice that the matrix mode is not activating, unless the user does not know what to expect. If this is the case then the system could be subverted by user ignorance.

The kickstart ROM, and therefore the hypervisor ROM, can be easily replaced by placing a file on the SD card. While this is good for development, it presents a potential vulnerability. As such, a method of safeguarding the hypervisor ROM could be devised, or simply this feature could be disabled by the user before bitstream synthesis if they decide security is more important.

Physical attacks targeting the software of the machine can be mitigated to some extent, however this system cannot protect against any attack which successfully replaces the bitstream. At the moment this an unresolved vulnerability as the bitstream is open source and unencrypted and is easily replaced on the SD card. If the bitstream were replaced and the secure compartment features were disabled, the machine could seemingly provide the “real” hash and memory contents. When the matrix mode appears, and everything looks correct, the user would be lead to believe that the software is legitimate while the machine could be keylogging or otherwise exfiltrating the user’s secure message.

It is realised that the bitstream may be the weakest link; given the re-programmable nature of FPGAs, there is potential for this to be spoofed. On Xilinx series 7 FPGA's it is possible to require an encrypted bitstream to be able to program the device. A 256-bit key is programmed by the user via JTAG and stored in either battery backed RAM (BBRAM) or eFUSE. The Xilinx bitstream writer on the PC encrypts the bitstream, and the device decrypts the incoming bitstream. However, as discussed in the literature review, there have been cases where the AES keys can be extracted from Xilinx devices though electromagnetic side-channel attacks. In contrast to proprietary designs, with the MEGA65 project, once the key is known there is no need to reverse engineer the extracted bitstream to be able to insert a backdoor or Trojan as the source is freely available. Furthermore, because it is open hardware even if the user’s FPGA is using an encrypted bitstream it is possible that an attacker could replace the entire PCB with a compromised version. Perhaps implementing some physical security features similar to the ORWL computer, discussed in the literature review, could protect against these types attacks which require access to the PCB.

Chapter 5

Conclusion

The design and implementation of a functional user-interface that allows for transparent inspection of the machine state was substantially completed. Design of a system which utilises this interface to allow for secure exfiltration-resistant compartments to be constructed and used was explored and implementation of parts of this system was discussed.

It was found that physical attacks or other vulnerabilities which target the software running on the MEGA65 could be mitigated to an extent, given the user understands what is expected of secure mode. Other physical attacks targeting the configuration bitstream cannot be prevented with this system. The use of encryption, as provided by the FPGA manufacturer, may reduce the chance of this kind of attack.

The contribution the author has made to the secure mode implementation on the MEGA65 provides a foundation to build upon such that the next student or open source contributor can come along and complete the implementation of this secure architecture. We’re not aware of anyone else making an architecture which has such strong security yet with a focus on being understandable.

5.1 Future Work

This thesis outlined an architecture for a secure mode on the MEGA65 project, however the implementation was not fully complete. There are a variety of tasks that should be completed in the future to complete the working system, and to extend the functionality of the overall platform.

Secure Mode Implementation

Primary future work would include the full realisation of the secure mode as designed. This would require the implementation of the hashing and encryption algorithms. After secure mode is complete, some secure services and user programs need to be written to take advantage of the system.

Functionality Enhancements

There could be work done on the task switcher to implement a full user interface which allows the explicit saving, loading and naming of tasks, as was originally envisioned for this feature. Machine state saving/loading must be sufficiently bug-free and tested to work with wide variety of software. But for task switching to be as seamless as possible, the speed at which the state is saved to the SD card must be improved. Multi-sector reads and writes must be implemented in the SD card interface. To do this may require modifications to the current DMA system. Furthermore, given that SD cards smaller than 4GB are hard to find these days, adding support for SDHC and SDXC protocols would be beneficial.

Ethernet functionality is very limited at the moment, so the development of a TCP/IP stack would enable the development of software to take advantage of internet connectivity. This would allow for the system to be used as a communications platform as originally envisioned.

There are also some enhancements which could be made to the matrix mode, such as optimising the interface to allow users to easily read memory. Two things could be improved here: Firstly, instead of requiring the user to explicitly input commands to read, when secure entry or exit is requested the memory interface could be navigated using a keyboard shortcut. Secondly, displaying the memory as characters alongside the hexadecimal output would allow the user to quickly spot plain text in memory. Such a feature is commonplace in hex editors.

Glossary

Backdoor: A method which circumvents authentication in a system and may allow unauthorised users to access the system.

Bitstream: An FPGA configuration file.

Cleartext: Readable data transmitted or stored “in the **clear**” (i.e. unencrypted).

Ciphertext: Output of an encryption algorithm.

Cryptographic Keys: String of bits used by a cryptographic algorithm to convert Cleartext into Ciphertext or vice versa.

DMA: Direct **M**emory **A**ccess, access to RAM independent of the CPU.

Exfiltration: Unauthorised transfer of information out of a system.

FPGA: Field **P**rogrammable **G**ate **A**rray, a developer-configurable integrated circuit.

Hypervisor: A manager of virtual machines. Or, in the MEGA65, a privileged CPU mode.

Keylogger: Malicious software or hardware device, which records keystrokes to steal credentials.

Out-of-Band: Outside of normal communication channel.

Risk: Product of the probability of an undesirable event and the severity of the event.

Register: Fast memory element inside a CPU.

Threat: Any danger which could exploit a vulnerability, leading to someone obtaining, or compromising an asset.

Trojan: A malicious computer program or hardware circuit pretending to be legitimate. Unlike a virus or worm, it does not spread by itself.

Virtualisation: An application, or a guest operation system which is abstracted away from the underlying hardware. Either through a software emulation layer or using hardware virtualisation features.

Vulnerability: Weakness or flaw in a system that can be exploited by threats.

References

Bankston, K., Schulman, R. & Laperruque, J., 2015. *New America.* [Online]
Available at: https://static.newamerica.org/attachments/12155-the-crypto-cat-is-out-of-the-bag/Crypto_Cat_Jan.0bea192f15424c9fa4859f78f1ad6b12.pdf

Bowen, F., Lassa, P., Gardei, B. & Andrade, V., 1991. *C64DX SYSTEM SPECIFICATION.* [Online]
Available at: http://www.zimmers.net/cbmpics/cbm/c65/c65manual.txt

Chan, C.-S., 2012. *Complexity the Worst Enemy of Security.* [Online]
Available at: https://www.schneier.com/news/archives/2012/12/complexity_the_worst.html

Cook, T., 2016. *A Message to Our Customers.* [Online]
Available at: http://www.apple.com/customer-letter/

Cowan, P., 2015. *UK PM wants to ban encrypted comms.* [Online]
Available at: https://www.itnews.com.au/news/uk-pm-wants-to-ban-encrypted-comms-399338

Cox, R., 2011. *The MOS 6502 and the Best Layout Guy in the World.* [Online]
Available at: https://research.swtch.com/6502

CVE, 2017. *Common Vulnerabilities and Exposures, CVE-2017-5689.* [Online]
Available at: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-5689

Design Shift, 2016. *Meet ORWL.* [Online]
Available at: https://orwl.org/about-orwl-first-open-source-physically-secure-computer/

Design Shift, 2016. *ORWL - The First Open Source, Physically Secure Computer.* [Online]
Available at: https://www.crowdsupply.com/design-shift/orwl

Digilent Inc., 2016. *Nexys 4 DDR Reference Manual.* [Online]
Available at: https://reference.digilentinc.com/reference/programmable-logic/nexys-4-ddr/reference-manual

Edge, J., 2016. *Qubes OS and colored-border spoofing.* [Online]
Available at: https://lwn.net/Articles/704287/

Embedi, 2017. *MythBusters: CVE-2017-5689.* [Online]
Available at: https://embedi.com/news/mythbusters-cve-2017-5689

Gardner-Stephens, P., Gerblich, B. & Gurcie, 2016. *MEGA65 FPGA Computer User Manual.* [Online]
Available at: https://github.com/MEGA65/mega65-core/blob/master/doc/usermanual0.md#11-purpose

Gelsinger, P., Kirkpatrick, D., Kolodny, A. & Singer, G., 2010. Such a CAD!. *IEEE Solid-State Circuits Magazine*, pp. 32-43.

Genkin, D., Pachmanov, L., Itamar, P. & Tromer, E., 2015. Stealing Keys from PCs using a Radio: Cheap Electromagnetic Attacks on Windowed Exponentiation. *Cryptographic Hardware and Embedded Systems -- CHES 2015: 17th International Workshop, Saint-Malo, France, September 13-16, 2015, Proceedings*, pp. 207-228.

Gerblich, B., 2017. *Build Documentation.* [Online]
Available at: https://github.com/MEGA65/mega65-core/blob/master/doc/build.md

Griffin, A., 2015. *Whatsapp and iMessage could be banned under new surveillance plans.* [Online]
Available at: https://www.independent.co.uk/life-style/gadgets-and-tech/news/whatsapp-and-snapchat-could-be-banned-under-new-surveillance-plans-9973035.html

Hall, J., D'Souza-Wiltshire, I., Lich, B. & Méndez, R. C., 2017. *Mitigate threats by using Windows 10 security features.* [Online]
Available at: https://docs.microsoft.com/en-us/windows/threat-protection/overview-of-threat-mitigations-in-windows-10

Hicks, M., Sturton, C., King, S. T. & Smith, J. M., 2015. SPECS: A Lightweight Runtime Mechanism for Protecting Software from Security-Critical Processor Bugs. *ACM SIGARCH Computer Architecture News,* 43(1), pp. 517-529.

Hu, W., Mao, Oberg, J. & Kastner, R., 2016. Detecting Hardware Trojans with Gate-Level Information-Flow Tracking. *Computer,* 15 August, 49(8), pp. 44-52.

Juarez, S., 2015. *Windows 10 Virtual Secure Mode with David Hepkin.* [Online]
Available at: https://channel9.msdn.com/Blogs/Seth-Juarez/Windows-10-Virtual-Secure-Mode-with-David-Hepkin

Krebs, B., 2014. *Complexity as the Enemy of Security.* [Online]
Available at: https://krebsonsecurity.com/2014/05/complexity-as-the-enemy-of-security/

Lord, N., 2017. *WHAT IS DATA EXFILTRATION?.* [Online]
Available at: https://digitalguardian.com/blog/what-data-exfiltration

Mark, E. & Goryachy, M., 2017. *Black Hat Briefings.* [Online]
Available at: https://www.blackhat.com/eu-17/briefings/schedule/#how-to-hack-a-turned-off-computer-or-running-unsigned-code-in-intel-management-engine-8668

McCabe Software, INC., 2012. *More Complex = Less Secure.* [Online]
Available at:
http://www.mccabe.com/pdf/More%20Complex%20Equals%20Less%20Secure-McCabe.pdf

MEGA65, 2017. *MEGA65 FACTS.* [Online]
Available at: http://mega65.org/#facts

Mehta, N., 2012. *Xilinx.* [Online]
Available at:
https://www.xilinx.com/support/documentation/white_papers/wp377_7Series_Embed_Mem_Advantages.pdf

Minev, P. B. & Kukenska, V. S., 2009. The Virtex-5 Routing and Logic Architecture. *Annual Journal of Electronics.*

Moradi, A., Kasper, M. & Paar, C., 2012. Black-Box Side-Channel Attacks Highlight the Importance of Countermeasures. In: O. Dunkelman, ed. *Topics in Cryptology -- CT-RSA 2012: The Cryptographers' Track at the RSA Conference 2012, San Francisco, CA, USA, February 27 -- March 2, 2012. Proceedings.* Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 1-18.

Moradi, A. & Schneider, T., 2016. *Improved Side-Channel Analysis Attacks on Xilinx Bitstream Encryption of 5, 6, and 7 Series.* In F. Standaert and E. Oswald, editors, COSADE 2016, volume 9689 of LNCS, pp. 71-87.

Morris, I., 2015. *WhatsApp And Snapchat Could Be Banned In The U.K. After Charlie Hebdo Murders.* [Online]
Available at: https://www.forbes.com/sites/ianmorris/2015/01/12/the-british-prime-minister-wants-to-ban-snapchat-and-whatsapp/#5911a9067a43

Qubes OS Project, 2017. *Getting Started.* [Online]
Available at: https://www.qubes-os.org/getting-started/

Rouse, M., 2013. *Definition: Evil Maid Attack.* [Online]
Available at: http://searchsecurity.techtarget.com/definition/evil-maid-attack

Rutkowska, J., 2008. *The three approaches to computer security.* [Online]
Available at: http://theinvisiblethings.blogspot.com.au/2008/09/three-approaches-to-computer-security.html

Rutkowska, J., 2015. *Intel x86 considered harmful.* [Online]
Available at: https://blog.invisiblethings.org/papers/2015/x86_harmful.pdf

Rutkowska, J., 2016. *Thoughts on ORWL.* [Online]
Available at: https://blog.invisiblethings.org/2016/09/03/thoughts-about-orwl.html

Saltzer, J. H. & Schroeder, M. D., 1975. *The Protection of Information in Computer Systems.* [Online]
Available at: http://www.cs.virginia.edu/~evans/cs551/saltzer/

Schneier, B., 1999. *A Please for Simplicity.* [Online]
Available at: https://www.schneier.com/essays/archives/1999/11/a_plea_for_simplicit.html

Schneier, B., 2000. *Crypto-Gram Newsletter, Article 8: Software Complexity and Security.* [Online]
Available at: https://www.schneier.com/crypto-gram/archives/2000/0315.html#8

Schneier, B., 2009. *"Evil Maid" Attacks on Encrypted Hard Drives.* [Online]
Available at: https://www.schneier.com/blog/archives/2009/10/evil_maid_attac.html

Skorobogatov, S. & Woods, C., 2012. *Breakthrough silicon scanning discovers backdoor in military chip.* Leuven, Belgium, CHES'12 Proceedings of the 14th international conference on Cryptographic Hardware and Embedded Systems, pp. 23-40.

Tails, 2017. *About.* [Online]
Available at: https://tails.boum.org/about/index.en.html
[Accessed 2017].

The Qubes OS Project, 2017. *Qubes OS: A reasonably secure operating system.* [Online]
Available at: https://www.qubes-os.org/
[Accessed 2017].

Turnbull, M., 2017. *Press Conference with Attorney-General and Acting Commissioner of the AFP - Sydney.* [Online]
Available at: https://www.malcolmturnbull.com.au/media/press-conference-with-attorney-general-and-acting-commissioner-of-the-afp-s

UK Parliament, 2016. *Investigatory Powers Act.* [Online]
Available at: https://publications.parliament.uk/pa/bills/lbill/2016-2017/0066/17066.pdf
[Accessed July 2017].

US Govt., 2012. *Cherry Blossom (CB) User's Manual.* [Online]
Available at: https://wikileaks.org/vault7/document/SRI-SLO-FF-2012-177-CherryBlossom_UsersManual_CDRL-12_SLO-FF-2012-171/SRI-SLO-FF-2012-177-CherryBlossom_UsersManual_CDRL-12_SLO-FF-2012-171.pdf

Wiggers, K., 2017. *Digital Trends.* [Online]
Available at: https://www.digitaltrends.com/mobile/public-charger-exploit/

WikiLeaks, 2017. *Vault 7: CIA Hacking Tools Revealed.* [Online]
Available at: https://wikileaks.org/ciav7p1/

Yang, K. et al., 2016. A2: Analog Malicious Hardware. *2016 IEEE Symposium on Security and Privacy*, pp. 18-37.

Appendices

Appendix A

Keyboard-to-Serial Implementation

(PS2_to_uart.vhdl)

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE ieee.numeric_std.ALL;
USE work.debugtools.ALL;
USE ieee.std_logic_unsigned.ALL;
ENTITY ps2_to_uart IS
 PORT (
  clk : IN STD_LOGIC; --48mhz
  reset : IN STD_LOGIC;
  enabled : IN std_logic;
  scan_code : IN std_logic_vector (12 DOWNTO 0);
  mm_displayMode_out : OUT std_logic_vector(1 DOWNTO 0);
  tx_ps2 : OUT STD_LOGIC;
  display_shift_out : OUT std_logic_vector(2 DOWNTO 0);
  shift_ready_out : OUT std_logic;
  matrix_trap_out : OUT std_logic := '0';
  shift_ack_in : IN std_logic
 );
END ps2_to_uart;

ARCHITECTURE Behavioral OF ps2_to_uart IS

 COMPONENT UART_TX_CTRL IS
  PORT (
   SEND : IN STD_LOGIC;
   DATA : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
   CLK : IN STD_LOGIC;
   READY : OUT STD_LOGIC;
   UART_TX : OUT STD_LOGIC
  );
 END COMPONENT;

 -- Determines which display scale mode to use for matrix mode
 -- 0=1:1 (640x320), 1=2:1 (1280x640), 2=3:1 (1920x960) (Fullscreen)
 -- CMD: Alt-1 , Alt-2 , Alt-3
 SIGNAL mm_displayMode : std_logic_vector(1 DOWNTO 0) := b"10";

 -- Pressing a cursor key will shift the matrix mode display by 8px
 -- Only on modes 0 and 1
 SIGNAL display_shift : std_logic_vector(2 DOWNTO 0) := b"000"; --direction
to shift. 1=up, 2=right, 3=down, 4=left
 SIGNAL shift_ack : std_logic; --Has the compositor shifted the display
already.
 SIGNAL shift_ready : std_logic;


 --UART Signals
 SIGNAL tx_data : std_logic_vector(7 DOWNTO 0);
 SIGNAL tx_ready : std_logic;
```

```
 SIGNAL tx_trigger : std_logic := '0';

 -- States in FSM
 TYPE ps2_to_uart_state IS (WaitForKey, Enter, KeyPress, Output);
 SIGNAL state : ps2_to_uart_state := WaitForKey;
 SIGNAL next_state : ps2_to_uart_state;
 SIGNAL timer : unsigned(27 DOWNTO 0) := (OTHERS => '0');

 --Keyboard Timers
 CONSTANT timer200ms : std_logic_vector(23 DOWNTO 0) :=
b"110100100111110010011001";
 CONSTANT timer30ms : std_logic_vector(20 DOWNTO 0) :=
b"11101111100100010111";
 SIGNAL repeatCounter1 : std_logic_vector(23 DOWNTO 0) := timer200ms;
 SIGNAL repeatCounter2 : std_logic_vector(20 DOWNTO 0) := timer30ms;
 --Keyboard signals
 SIGNAL caps : std_logic;
 SIGNAL previousScanCode : std_logic_vector (12 DOWNTO 0);
 SIGNAL firstPress : std_logic;
 SIGNAL inputKey : std_logic;
 SIGNAL firstRepeatDone : std_logic;
 SIGNAL altcode : std_logic;
BEGIN
 uart_tx1 : UART_TX_CTRL
 PORT MAP(
  send => tx_trigger,
  clk => clk,
  data => tx_data,
  ready => tx_ready,
  uart_tx => tx_ps2
 );

 uart_test : PROCESS (clk)
 BEGIN
  IF rising_edge(CLK) THEN
   IF enabled = '1' THEN --Input only run when matrix mode is enabled
    CASE state IS
     WHEN WaitForKey =>
      tx_trigger <= '0';

     WHEN KeyPress =>
      state <= Output;
      next_state <= WaitForKey;

     WHEN Enter =>
      tx_data <= x"0D"; --cr only
      next_state <= WaitForKey;
      state <= Output;

     WHEN Output =>
      IF tx_ready = '1' THEN
       tx_trigger <= '1';
       state <= next_state;
      END IF;
    END CASE;
   END IF;

   --If the scan code is Left Shift
   IF scan_code(7 DOWNTO 0) = x"12" THEN
    IF scan_code(12) = '0' THEN --and make code
     caps <= '1';
```

```
     ELSE --break code
      caps <= '0';
     END IF;
    END IF;



    --CursorKeys
    -- E0 75 up
    -- E0 6B left
    -- E0 72 down
    -- E0 74 right
    -- ScanCode(8) is extended code

    --Check alt-1/2/3 combos
    -- ALT: x"11" b"00010001"
    -- Don't care whats in bits 11-8
    --If the previous key was alt, and is still held.

    -- When keyboard has no keys pressed its last code is a break code
    -- When a key is pressed scan_code(12) will be '1' i.e. make code
    -- Input first keystroke
    -- Wait ~250ms
    -- firstRepeatDone will be '1'
    -- When first repeat is done second counter
    -- will count down ~50msish before inputting keys again
    -- If breakcode or new code is sent, reset everything.
    -- If the previous scan code wasnt ALT, do normal keys. (this is so
1/2/3 arent input when switching modes)
    IF scan_code(12) = '0' AND firstPress = '0' THEN --and
(previousScanCode(12)&previousScanCode(7 downto 0) /= b"000010001") then
     inputKey <= '1'; --input key when first instance of key is pressed
     firstPress <= '1';
     previousScanCode <= scan_code;
    ELSIF scan_code(12) = '0' AND firstPress = '1' THEN -- if it has been
held down
     IF repeatCounter1 = x"000000" THEN
      IF firstRepeatDone = '0' THEN
       firstRepeatDone <= '1';
      END IF;
     ELSE
      repeatCounter1 <= repeatCounter1 - 1;
     END IF;
    END IF;

    IF firstRepeatDone = '1' THEN
     IF repeatCounter2 = b"00000000000000000000" THEN
      repeatCounter2 <= timer30ms; --reload timer
      inputKey <= '1'; --input character
     ELSE --otherwise decrement the timer
      repeatCounter2 <= repeatCounter2 - 1;
     END IF;
    END IF;

    IF previousScanCode(12) & previousScanCode(7 DOWNTO 0) = b"000010001"
AND altcode = '1' THEN
     --1= x"16", 2=x"1E", 3="26"
     CASE scan_code(12) & scan_code(7 DOWNTO 0) IS --make sure its a MAKE
code.
      WHEN '0' & x"16" => --1
       mm_displayMode <= b"00";
```

```
      WHEN '0' & x"1E" => --2
       mm_displayMode <= b"01";
      WHEN '0' & x"26" => --3
       mm_displayMode <= b"10";
      WHEN '0' & x"0D" => --Alt-tab, activates matrix mode. These are
probably temporary commands
       matrix_trap_out <= '1'; --setup trap.
      WHEN OTHERS => --Alt anything
       --Do Nothing
       --altcode<='0'; --Why is this here.
     END CASE;
    ELSE
     matrix_trap_out <= '0'; --reset trap
    END IF;

    --when any key is released or a new key is down, reset repeat timers
    IF scan_code(12) = '1' OR scan_code /= previousScanCode THEN
     repeatCounter1 <= timer200ms;
     repeatCounter2 <= timer30ms;
     firstRepeatDone <= '0';
     inputKey <= '0'; --stop any input;
     firstPress <= '0';
    END IF;

    --If a key is lefted, no longer an alt-code (keys need to be down at
same time)
    IF scan_code(12) = '1' THEN
     altcode <= '0';
    END IF;

    IF shift_ack = '1' THEN
     shift_ready <= '0';
    END IF;


    IF inputKey = '1' THEN
     inputKey <= '0'; --Disable input character.
     CASE scan_code(7 DOWNTO 0) IS
      WHEN x"11" =>
       altcode <= '1';
       --up / numpad 8 --Don't bother checking for extended code, as numpad
isn't implemented
      WHEN x"75" =>
       display_shift <= b"001";
       shift_ready <= '1';
       --left / numpad 4
      WHEN x"6B" =>
       display_shift <= b"100";
       shift_ready <= '1';
       --down / numpad 2
      WHEN x"72" =>
       display_shift <= b"011";
       shift_ready <= '1';
       --right / numpad 6
      WHEN x"74" =>
       display_shift <= b"010";
       shift_ready <= '1';

       -- 3, W, A, 4, Z, S, E, left-SHIFT
      WHEN x"26" =>
       IF altcode = '0' THEN
```

```
   IF caps = '0' THEN --3/#
    tx_data <= x"33";
   ELSE
    tx_data <= x"23";
   END IF;
   state <= KeyPress;
  END IF;

 WHEN x"1D" => --W
  IF caps = '0' THEN
   tx_data <= x"77";
  ELSE
   tx_data <= x"57";
  END IF;
  state <= KeyPress;

 WHEN x"1C" => --A
  IF caps = '0' THEN
   tx_data <= x"61";
  ELSE tx_data <= x"41";
  END IF;
  state <= KeyPress;

 WHEN x"25" => --4/$
  IF caps = '0' THEN
   tx_data <= x"34";
  ELSE tx_data <= x"24";
  END IF;
  state <= KeyPress;

 WHEN x"1A" => --Z
  IF caps = '0' THEN
   tx_data <= x"7A";
  ELSE tx_data <= x"5A";
  END IF;
  state <= KeyPress;

 WHEN x"1B" => --S
  IF caps = '0' THEN
   tx_data <= x"73";
  ELSE tx_data <= x"53";
  END IF;
  state <= KeyPress;

 WHEN x"24" => --E
  IF caps = '0' THEN
   tx_data <= x"65";
  ELSE tx_data <= x"45";
  END IF;
  state <= KeyPress;


  -- 5, R, D, 6, C, F, T, X
 WHEN x"2E" =>
  IF caps = '0' THEN
   tx_data <= x"35";
  ELSE
   tx_data <= x"25";
  END IF;
  state <= KeyPress; --5
```

```
WHEN x"2D" =>
 IF caps = '0' THEN
  tx_data <= x"72";
 ELSE tx_data <= x"52";
 END IF;
 state <= KeyPress;--R

WHEN x"23" =>
 IF caps = '0' THEN
  tx_data <= x"64";
 ELSE tx_data <= x"44";
 END IF;
 state <= KeyPress;--D

WHEN x"36" =>
 IF caps = '0' THEN
  tx_data <= x"36";
 ELSE tx_data <= x"5E";
 END IF;
 state <= KeyPress;--6

WHEN x"21" =>
 IF caps = '0' THEN
  tx_data <= x"63";
 ELSE tx_data <= x"43";
 END IF;
 state <= KeyPress;--C

WHEN x"2B" =>
 IF caps = '0' THEN
  tx_data <= x"66";
 ELSE tx_data <= x"46";
 END IF;
 state <= KeyPress;--F

WHEN x"2C" =>
 IF caps = '0' THEN
  tx_data <= x"74";
 ELSE tx_data <= x"54";
 END IF;
 state <= KeyPress;--T

WHEN x"22" =>
 IF caps = '0' THEN
  tx_data <= x"78";
 ELSE tx_data <= x"58";
 END IF;
 state <= KeyPress;--X

 -- 7, Y, G, 8, B, H, U, V
WHEN x"3D" =>
 IF caps = '0' THEN
  tx_data <= x"37";
 ELSE tx_data <= x"26";
 END IF;
 state <= KeyPress; --7/&

WHEN x"35" =>
 IF caps = '0' THEN
  tx_data <= x"79";
 ELSE tx_data <= x"59";
```

```
 END IF;
 state <= KeyPress;--Y

WHEN x"34" =>
 IF caps = '0' THEN
  tx_data <= x"67";
 ELSE tx_data <= x"47";
 END IF;
 state <= KeyPress;--G

WHEN x"3E" =>
 IF caps = '0' THEN
  tx_data <= x"38";
 ELSE
  tx_data <= x"2A"; --8/*
 END IF;
 state <= KeyPress;

WHEN x"32" =>
 IF caps = '0' THEN
  tx_data <= x"62";
 ELSE tx_data <= x"42";
 END IF;
 state <= KeyPress;--B

WHEN x"33" =>
 IF caps = '0' THEN
  tx_data <= x"68";
 ELSE tx_data <= x"48";
 END IF;
 state <= KeyPress;--H

WHEN x"3C" =>
 IF caps = '0' THEN
  tx_data <= x"75";
 ELSE tx_data <= x"55";
 END IF;
 state <= KeyPress;--U

WHEN x"2A" =>
 IF caps = '0' THEN
  tx_data <= x"76";
 ELSE tx_data <= x"56";
 END IF;
 state <= KeyPress;--V

 -- 9, I, J, 0, M, K, O, N
WHEN x"46" =>
 IF caps = '0' THEN
  tx_data <= x"39";
 ELSE
  tx_data <= x"28";
 END IF;
 state <= KeyPress;--9/(

WHEN x"43" =>
 IF caps = '0' THEN
  tx_data <= x"69";
 ELSE tx_data <= x"49";
 END IF;
 state <= KeyPress;--I
```

```
WHEN x"3B" =>
 IF caps = '0' THEN
  tx_data <= x"6A";
 ELSE tx_data <= x"4A";
 END IF;
 state <= KeyPress;--J

WHEN x"45" =>
 IF caps = '0' THEN
  tx_data <= x"30";
 ELSE
  tx_data <= x"29";
 END IF;
 state <= KeyPress; --0/)

WHEN x"3A" =>
 IF caps = '0' THEN
  tx_data <= x"6D";
 ELSE tx_data <= x"4D";
 END IF;
 state <= KeyPress;--M

WHEN x"42" =>
 IF caps = '0' THEN
  tx_data <= x"6B";
 ELSE tx_data <= x"4B";
 END IF;
 state <= KeyPress;--K

WHEN x"44" =>
 IF caps = '0' THEN
  tx_data <= x"6F";
 ELSE tx_data <= x"4F";
 END IF;
 state <= KeyPress; --O

WHEN x"31" =>
 IF caps = '0' THEN
  tx_data <= x"6E";
 ELSE tx_data <= x"4E";
 END IF;
 state <= KeyPress; --N

 -- +, P, L, -, ., :, @, COMMA
WHEN x"4E" =>
 IF caps = '0' THEN
  tx_data <= x"2D";
 ELSE tx_data <= x"5F";
 END IF;
 state <= KeyPress; ---_

WHEN x"4D" =>
 IF caps = '0' THEN
  tx_data <= x"70";
 ELSE tx_data <= x"50";
 END IF;
 state <= KeyPress; --P

WHEN x"4B" =>
 IF caps = '0' THEN
```

```
   tx_data <= x"6C";
  ELSE tx_data <= x"4C";
  END IF;
  state <= KeyPress; --L

 WHEN x"55" =>
  IF caps = '0' THEN
   tx_data <= x"3D";
  ELSE
   tx_data <= x"2B";
  END IF;
  state <= KeyPress; --=/+

 WHEN x"49" =>
  IF caps = '0' THEN
   tx_data <= x"2E";
  ELSE
   tx_data <= x"3E";
  END IF;
  state <= KeyPress;--./>

 WHEN x"4C" =>
  IF caps = '0' THEN
   tx_data <= x"3B";
  ELSE
   tx_data <= x"3A";
  END IF;
  state <= KeyPress;--;/:

 WHEN x"54" =>
  tx_data <= x"5B";
  state <= KeyPress;--[

 WHEN x"41" =>
  IF caps = '0' THEN
   tx_data <= x"2C";
  ELSE
   tx_data <= x"3C";
  END IF;
  state <= KeyPress;--,/<

 WHEN x"16" =>
  IF altcode = '0' THEN
   IF caps = '0' THEN
    tx_data <= x"31";
   ELSE
    tx_data <= x"21";
   END IF;
   state <= KeyPress; --1/!
  END IF;


 WHEN x"1E" =>
  IF altcode = '0' THEN
   IF caps = '0' THEN
    tx_data <= x"32";
   ELSE
    tx_data <= x"40";
   END IF;
   state <= KeyPress; --2/@
  END IF;
```

```
      WHEN x"15" =>
       IF caps = '0' THEN
        tx_data <= x"71";
       ELSE
        tx_data <= x"51";
       END IF;
       state <= KeyPress; --Q

      WHEN x"5A" =>
       tx_data <= x"2C";
       state <= Enter; --ENTER

      WHEN x"66" => --del
       tx_data <= x"08";
       state <= KeyPress;

      WHEN x"29" => --space
       tx_data <= x"20";
       state <= KeyPress;

      WHEN OTHERS => state <= WaitForKey;
     END CASE;
    END IF;
   END IF;
  END PROCESS uart_test;

  mm_displayMode_out <= mm_displayMode;
  shift_ready_out <= shift_ready;
  shift_ack <= shift_ack_in;
  display_shift_out <= display_shift;
END Behavioral;
```

Appendix B

Terminal Emulator (terminalemulator.vhdl)

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE ieee.numeric_std.ALL;
USE work.debugtools.ALL;
USE ieee.std_logic_unsigned.ALL;

ENTITY terminalemulator IS
 PORT (
  clk : IN STD_LOGIC; --200Mhz?
  uart_clk : IN std_logic; --48MHz
  uart_in : IN STD_LOGIC;
  topofframe_out : OUT std_logic_vector(11 DOWNTO 0);
  wel_out : OUT STD_LOGIC_VECTOR(0 DOWNTO 0);
  addrl_out : OUT STD_LOGIC_VECTOR(11 DOWNTO 0);
  dinl_out : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
 );
END terminalemulator;

ARCHITECTURE Behavioral OF terminalemulator IS

 COMPONENT uart_rx IS
  PORT (
   clk : IN std_logic;
   UART_RX : IN std_logic;
   data : OUT std_logic_vector(7 DOWNTO 0);
   data_ready : OUT std_logic;
   data_acknowledge : IN std_logic
  );
 END COMPONENT;

 TYPE terminal_emulator_state IS (clearAck,
 incChar,
 writeChar, writeChar2,
 waitforinput,
 processCommand,
 newLine, newLine2,
 newFrame,
 clearLine,
 linefeed, linefeed2,
 backspace,
 writeCursor, writeCursor2,
 clearCursor, clearCursor2);
 SIGNAL state : terminal_emulator_state := waitforinput;
 SIGNAL next_state : terminal_emulator_state;

 CONSTANT CharMemStart : std_logic_vector(11 DOWNTO 0) := x"302";
 CONSTANT CharMemEnd : std_logic_vector(11 DOWNTO 0) := x"F81";

 SIGNAL rx_data : std_logic_vector(7 DOWNTO 0);
 SIGNAL rx_ready : std_logic;
 SIGNAL rx_acknowledge : std_logic;
 SIGNAL dataToWrite : std_logic_vector(7 DOWNTO 0);
 SIGNAL charCursor : std_logic_vector(11 DOWNTO 0) := CharMemStart;
 SIGNAL charX : std_logic_vector(7 DOWNTO 0) := x"00";
 SIGNAL lastLineStart : std_logic_vector(11 DOWNTO 0) := CharMemStart;
 SIGNAL clearLineStart : std_logic_vector(11 DOWNTO 0) := CharMemStart;
```

```
SIGNAL clearLineEnd : std_logic_vector(11 DOWNTO 0) := CharMemStart + 80;
 SIGNAL topofframe : std_logic_vector(11 DOWNTO 0) := CharMemStart
 SIGNAL hasHitEoF : std_logic := '0';
BEGIN
 uart_rx0 : uart_rx
 PORT MAP(
  clk => uart_clk,
  UART_RX => uart_in,
  data => rx_data,
  data_ready => rx_ready,
  data_acknowledge => rx_acknowledge
 );

 topofframe_out <= topOfFrame;

 uart_receive : PROCESS (clk)
 BEGIN
  IF rising_edge(uart_clk) THEN
   CASE state IS
    WHEN waitforinput =>
     IF rx_ready = '1' AND rx_acknowledge = '0' THEN
      rx_acknowledge <= '1';
      IF rx_data < x"20" AND rx_data /= x"0A" THEN
       state <= clearCursor;
       next_state <= processCommand;
      ELSIF rx_data = x"0A" THEN
       IF charX = x"00" THEN
        state <= processCommand;
       ELSE
        state <= clearCursor;
        next_state <= processCommand;
       END IF;

      ELSE
       state <= writeChar;
       dataToWrite <= rx_data - 32;
      END IF;
     END IF;
    WHEN writeCursor =>
     dinl_out <= b"10000000";
     addrl_out <= charCursor;
     wel_out <= b"1";
     state <= writeCursor2;

    WHEN writeCursor2 =>
     wel_out <= b"0";
     state <= next_state;

    WHEN clearCursor =>
     dinl_out <= b"00000000";
     addrl_out <= charCursor;
     wel_out <= b"1";
     state <= clearCursor2;

    WHEN clearCursor2 =>
     wel_out <= b"0";
     state <= next_state;

    WHEN writeChar =>
     addrl_out <= charCursor; --latch address
     dinl_out <= dataToWrite; --latch output data
```

```
 wel_out <= b"1"; --enable write
 state <= WriteChar2;

WHEN writeChar2 =>
 rx_acknowledge <= '0'; --clear acknowledged
 wel_out <= b"0";
 state <= incChar;


 --Increase char position by 1
WHEN incChar =>
 --Check boundaries
 IF charX >= x"4F" THEN --if its at the end of a line
  charX <= (OTHERS => '0');
  IF charCursor >= CharMemEnd THEN
   state <= newFrame;
   charCursor <= CharMemStart;--(others=>'0');
   lastLineStart <= CharMemStart;--others=>'0');
   hasHitEoF <= '1';
  ELSE
   charCursor <= charCursor + 1;
   state <= newLine;
  END IF;

 ELSE
  charCursor <= charCursor + 1;
  charX <= charX + 1;
  --state<=clearAck;
  state <= writeCursor;
  next_state <= clearAck;
 END IF;

WHEN newFrame =>
 charCursor <= CharMemStart;
 lastLineStart <= CharMemStart;
 charX <= (OTHERS => '0');
 hasHitEoF <= '1';
 clearLineStart <= CharMemStart;
 clearLineEnd <= CharMemStart + 80;
 state <= ClearLine;

WHEN newLine =>
 lastLineStart <= charCursor;
 clearLineStart <= charCursor;
 clearLineEnd <= charCursor + 80;
 --Write new cursor whenever charCursor moves
 state <= newLine2;
WHEN newLine2 =>
 next_state <= clearAck;
 state <= clearLine;

WHEN processCommand =>
 IF rx_data = x"0D" THEN --CR carriage return
  charCursor <= lastLineStart; --go back to start of line?
  charX <= (OTHERS => '0');
  state <= clearAck;
 ELSIF rx_data = x"0A" THEN --LF line feed
  charCursor <= charCursor + 80;
  lastLineStart <= lastLineStart + 80;
  state <= linefeed;
 ELSIF rx_data = x"08" THEN --BS
```

```
    charCursor <= charCursor - 1;
    charX <= charX - 1;
    dataToWrite <= x"00";
    wel_out <= b"1";
    --state<=backspace;
    state <= writeCursor;
    next_state <= clearAck;

   ELSE
    state <= clearAck;
   END IF;

  WHEN backspace =>
   addrl_out <= charCursor;
   dinl_out <= dataToWrite;
   state <= clearAck;

   --Clear acknowledge, ready for next Char
  WHEN clearAck =>
   wel_out <= b"0";
   rx_acknowledge <= '0';
   state <= waitforinput;


  WHEN linefeed =>
   IF charCursor > CharMemEnd THEN
    charCursor <= charCursor - 3200;
    hasHitEoF <= '1';
   END IF;
   -- >3120 (the last line start)
   IF lastLineStart > CharMemEnd - 79 THEN
    lastLineStart <= CharMemStart;
    hasHitEoF <= '1';
   END IF;

   state <= linefeed2;

  WHEN linefeed2 =>
   clearLineStart <= lastLineStart;
   clearLineEnd <= lastLineStart + 80;
   state <= clearLine;

  WHEN clearLine =>
   IF hasHitEoF = '1' THEN
    IF topOfFrame >= CharMemEnd - 79 THEN
     topOfFrame <= CharMemStart;
    ELSE --otherwise increase
     topOfFrame <= topOfFrame + 80;
    END IF;
   END IF;

   wel_out <= b"1";
   addrl_out <= clearLineStart;
   dinl_out <= (OTHERS => '0');
   clearLineStart <= clearLineStart + 1;

   IF (clearLineStart = clearLineEnd) THEN
    wel_out <= b"0";
    state <= clearAck;
   END IF;
 END CASE;
```

```
  END IF;
 END PROCESS;
END Behavioral;
```

Appendix C

Video Generator (compositor.vhdl)

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE ieee.numeric_std.ALL;
USE work.debugtools.ALL;
USE ieee.std_logic_unsigned.ALL;

ENTITY compositor IS
 PORT (
  display_shift_in : IN std_logic_vector(2 DOWNTO 0);
  shift_ready_in : IN std_logic;
  shift_ack_out : OUT std_logic;
  mm_displayMode_in : IN std_logic_vector(1 DOWNTO 0);
  uart_in : IN std_logic;
  xcounter_in : IN unsigned(11 DOWNTO 0);
  ycounter_in : IN unsigned(10 DOWNTO 0);
  clk : IN std_logic; --48Mhz
  pixelclock : IN std_logic; --200Mhz
  matrix_mode_enable : IN STD_LOGIC;
  vgared_in : IN unsigned (3 DOWNTO 0);
  vgagreen_in : IN unsigned (3 DOWNTO 0);
  vgablue_in : IN unsigned (3 DOWNTO 0);
  vgared_out : OUT unsigned (3 DOWNTO 0);
  vgagreen_out : OUT unsigned (3 DOWNTO 0);
  vgablue_out : OUT unsigned (3 DOWNTO 0)
 );
END compositor;

ARCHITECTURE Behavioral OF compositor IS

 COMPONENT uart_charrom IS
  PORT (
   clkl : IN STD_LOGIC;
   wel : IN STD_LOGIC_VECTOR(0 DOWNTO 0);
   addrl : IN STD_LOGIC_VECTOR(11 DOWNTO 0);
   dinl : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
   clkr : IN STD_LOGIC;
   addrr : IN STD_LOGIC_VECTOR(11 DOWNTO 0);
   doutr : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
  );
 END COMPONENT;

 COMPONENT terminalemulator IS
  PORT (
   clk : IN STD_LOGIC;
   uart_clk : IN std_logic;
   uart_in : IN STD_LOGIC;
   topofframe_out : OUT std_logic_vector(11 DOWNTO 0);
   wel_out : OUT STD_LOGIC_VECTOR(0 DOWNTO 0);
   addrl_out : OUT STD_LOGIC_VECTOR(11 DOWNTO 0);
```

```
  dinl_out : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
 );
END COMPONENT;

--Location of start of character memory
CONSTANT CharMemStart : std_logic_vector(11 DOWNTO 0) := x"302";
--Location of end of character memory
CONSTANT CharMemEnd : std_logic_vector(11 DOWNTO 0) := x"F81";
--Character Map Memory Interface
SIGNAL writeEnable : std_logic_vector(0 DOWNTO 0);
SIGNAL writeAddress : std_logic_vector (11 DOWNTO 0);
SIGNAL dataInWrite : std_logic_vector(7 DOWNTO 0);
SIGNAL charAddr : std_logic_vector (7 DOWNTO 0);
SIGNAL readAddress_rom : std_logic_vector(11 DOWNTO 0) := CharMemStart;
SIGNAL dataOutRead_rom : std_logic_vector (7 DOWNTO 0);
-- Frame boundaries
SIGNAL startx : unsigned(11 DOWNTO 0) := x"079";
SIGNAL endx : unsigned(11 DOWNTO 0) := x"814";
SIGNAL starty : unsigned(11 DOWNTO 0) := x"07C";
SIGNAL endy : unsigned(11 DOWNTO 0) := x"43C";

--Mode0 Frame
--640x320
CONSTANT mode0_startx : unsigned(11 DOWNTO 0) := x"096";
CONSTANT mode0_starty : unsigned(11 DOWNTO 0) := x"07C";
CONSTANT mode0_endx : unsigned(11 DOWNTO 0) := mode0_startx + 647;
CONSTANT mode0_endy : unsigned(11 DOWNTO 0) := x"1BB";
CONSTANT mode0_garbage_end_offset : unsigned(11 DOWNTO 0) := x"008";
--Mode1 Frame
--1280x640
CONSTANT mode1_startx : unsigned(11 DOWNTO 0) := x"096";
CONSTANT mode1_starty : unsigned(11 DOWNTO 0) := x"07C";
CONSTANT mode1_endx : unsigned(11 DOWNTO 0) := mode1_startx + 1295;
CONSTANT mode1_endy : unsigned(11 DOWNTO 0) := mode1_starty + 640;
CONSTANT mode1_garbage_end_offset : unsigned(11 DOWNTO 0) := x"00F";

--Mode2 Frame
--1920x960
CONSTANT mode2_startx : unsigned(11 DOWNTO 0) := x"079";
CONSTANT mode2_starty : unsigned(11 DOWNTO 0) := x"07C";
CONSTANT mode2_endx : unsigned(11 DOWNTO 0) := x"813";
CONSTANT mode2_endy : unsigned(11 DOWNTO 0) := x"43B";
CONSTANT mode2_garbage_end_offset : unsigned(11 DOWNTO 0) := x"01D";

SIGNAL xOffset : unsigned(11 DOWNTO 0) := x"000";
SIGNAL yOffset : unsigned(11 DOWNTO 0) := x"000";
SIGNAL shift_ack : std_logic := '0';
SIGNAL garbage_end : unsigned(11 DOWNTO 0) := x"000";
SIGNAL garbage_end_offset : unsigned(11 DOWNTO 0) := x"000";

--Character signals
SIGNAL charCount : std_logic_vector(11 DOWNTO 0) := CharMemStart;
SIGNAL charline : std_logic_vector(3 DOWNTO 0);
SIGNAL eightCounter : std_logic_vector(4 DOWNTO 0) := (OTHERS => '0');
SIGNAL bufferCounter : std_logic_vector(1 DOWNTO 0) := (OTHERS => '0');
SIGNAL invert : std_logic;

--Outputs
SIGNAL greenOutput : std_logic := '0';
SIGNAL redOutput : std_logic := '0';
SIGNAL blueOutput : std_logic := '0';
```

```
 --4-bit Outputs
 SIGNAL greenOutput_all : unsigned(3 DOWNTO 0);
 SIGNAL redOutput_all : unsigned(3 DOWNTO 0);
 SIGNAL blueOutput_all : unsigned(3 DOWNTO 0);

 SIGNAL data_buffer : std_logic_vector(7 DOWNTO 0) := x"00";
 SIGNAL lineStartAddr : std_logic_vector(11 DOWNTO 0) := CharMemStart;
 SIGNAL lineCounter : std_logic_vector(2 DOWNTO 0) := b"000";
 SIGNAL topOfFrame : std_logic_vector(11 DOWNTO 0) := CharMemStart;
 SIGNAL doneEndOfFrame : std_logic := '0';
 SIGNAL doneEndOfFrame1 : std_logic := '0';
 SIGNAL doneEndOfFrame2 : std_logic := '0';
 --Display Mode signals
 SIGNAL mm_displayMode : std_logic_vector(1 DOWNTO 0) := b"10";
 SIGNAL end_of_char : std_logic_vector(4 DOWNTO 0) := b"11000";
 CONSTANT mode0_end_of_char : std_logic_vector(4 DOWNTO 0) := b"01000"; --8
 CONSTANT mode1_end_of_char : std_logic_vector(4 DOWNTO 0) := b"10000"; --
16
 CONSTANT mode2_end_of_char : std_logic_vector(4 DOWNTO 0) := b"11000"; --
24

BEGIN
 uart_charrom1 : uart_charrom
 PORT MAP(
  clkl => pixelclock,
  clkr => pixelclock,
  wel => writeEnable,
  addrl => writeAddress,
  addrr => readAddress_rom,
  dinl => dataInWrite,
  doutr => dataOutRead_rom
 );

 terminalemulator0 : terminalemulator
 PORT MAP(
  clk => pixelclock,
  uart_clk => clk,
  uart_in => uart_in,
  topofframe_out => topOfFrame,
  wel_out => writeEnable,
  addrl_out => writeAddress,
  dinl_out => dataInWrite
 );

 vgared_out <= vgared_in WHEN matrix_mode_enable = '0' ELSE
               redOutput_all;
 vgagreen_out <= vgagreen_in WHEN matrix_mode_enable = '0' ELSE
greenOutput_all;
 vgablue_out <= vgablue_in WHEN matrix_mode_enable = '0' ELSE
blueOutput_all;

 ram_test : PROCESS (pixelclock)
 BEGIN
  IF rising_edge(pixelclock) THEN
   --End of line 2100
   IF xcounter_in = 2400 AND ycounter_in >= starty AND ycounter_in < endy
THEN
    IF lineCounter = mm_displayMode THEN
     lineCounter <= b"000";
     IF charline = b"0111" THEN
```

```
    charline <= b"0000";
    --Boundary check
    IF lineStartAddr = CharMemEnd - 79 THEN
     lineStartAddr <= CharMemStart;
    ELSE
     lineStartAddr <= lineStartAddr + 80;
    END IF;
   ELSE
    charline <= charline + 1;
   END IF;
  ELSE
   lineCounter <= lineCounter + 1;
  END IF;
END IF;

IF xcounter_in = 2401 AND ycounter_in < endy THEN
 charCount <= lineStartAddr;
 eightCounter <= (OTHERS => '0');
 bufferCounter <= (OTHERS => '0');
END IF;

--End of Frame, reset counters
IF ycounter_in = b"10010110000" THEN
 IF doneEndOfFrame = '0' THEN
  mm_displayMode <= mm_displayMode_in;
  doneEndOfFrame <= '1';
  lineCounter <= (OTHERS => '0');
  charline <= (OTHERS => '0');
  charCount <= topOfFrame;
  lineStartAddr <= topOfFrame;
  eightCounter <= (OTHERS => '0');

  IF shift_ack = '0' AND shift_ready_in = '1' THEN
   CASE display_shift_in IS
    WHEN b"001" => --up
     IF starty > 25 THEN
      yoffset <= yoffset - 8;
     END IF;
    WHEN b"010" => --right
     IF endx < x"7F8" THEN
      xoffset <= xoffset + 8;
     END IF;
    WHEN b"011" => --down
     IF endy < 1200 THEN
      yoffset <= yoffset + 8;
     END IF;
    WHEN b"100" => --left
     IF garbage_end > 150 THEN
      xoffset <= xoffset - 8;
     END IF;
    WHEN OTHERS =>
   END CASE;
   shift_ack <= '1';
  ELSE
   shift_ack <= '0'; --reset ack
  END IF;


  CASE mm_displayMode_in IS
   WHEN b"00" =>
    end_of_char <= mode0_end_of_char;
```

```
        startx <= mode0_startx + xoffset;
        starty <= mode0_starty + yoffset;
        endx <= mode0_endx + xoffset;
        endy <= mode0_endy + yoffset;
        garbage_end_offset <= mode0_garbage_end_offset;
       WHEN b"01" =>
        end_of_char <= mode1_end_of_char;
        startx <= mode1_startx + xoffset;
        starty <= mode1_starty + yoffset;
        endx <= mode1_endx + xoffset;
        endy <= mode1_endy + yoffset;
        garbage_end_offset <= mode1_garbage_end_offset;
       WHEN b"10" =>
        end_of_char <= mode2_end_of_char;
        startx <= mode2_startx;
        starty <= mode2_starty;
        endx <= mode2_endx;
        endy <= mode2_endy;
        garbage_end_offset <= mode2_garbage_end_offset;
       WHEN OTHERS =>
        end_of_char <= mode2_end_of_char;
        startx <= mode2_startx;
        starty <= mode2_starty;
        endx <= mode2_endx;
        endy <= mode2_endy;
        garbage_end_offset <= mode2_garbage_end_offset;
      END CASE;
     END IF;
    END IF;


    IF ycounter_in = b"10010110100" THEN
     IF doneEndOfFrame1 = '0' THEN
      garbage_end <= startx + garbage_end_offset;
     END IF;
    END IF;

    IF xcounter_in >= startx AND xcounter_in <= endx AND ycounter_in >=
starty AND ycounter_in <= endy THEN

     --====================
     -- Generate Outputs:
     --====================

     --Green Outline on modes 0 and 1 Only
     IF xcounter_in >= garbage_end THEN
      IF mm_displayMode /= b"10" AND (xcounter_in = garbage_end OR
xcounter_in = endx OR ycounter_in = starty OR ycounter_in = endy) THEN
       redOutput_all <= b"00" & vgared_in(1 DOWNTO 0);
       greenOutput_all <= b"111" & vgagreen_in(0);
       blueOutput_all <= b"00" & vgablue_in(1 DOWNTO 0);
      ELSE
       IF data_buffer(7) = '1' THEN
        redOutput_all <= b"00" & vgared_in(1 DOWNTO 0);
        greenOutput_all <= data_buffer(7) & data_buffer(7) &
data_buffer(7) & vgagreen_in(0);
        blueOutput_all <= b"00" & vgablue_in(1 DOWNTO 0);
       ELSE
        redOutput_all <= b"00" & vgared_in(1 DOWNTO 0);
        greenOutput_all <= data_buffer(7) & data_buffer(7) & vgagreen_in(1
DOWNTO 0);
```

```
        blueOutput_all <= b"00" & vgablue_in(1 DOWNTO 0);
       END IF;
      END IF;

     ELSE --If its in garbage display background.
      IF mm_displayMode = b"10" THEN
       redOutput_all <= b"00" & vgared_in(1 DOWNTO 0);
       greenOutput_all <= b"00" & vgagreen_in(1 DOWNTO 0);
       blueOutput_all <= b"00" & vgablue_in(1 DOWNTO 0);
      ELSE
       redOutput_all <= vgared_in;
       greenOutput_all <= vgagreen_in;
       blueOutput_all <= vgablue_in;
      END IF;
     END IF;

     --======================
     --Timing and memory
     --======================

     -- We've got 8 clocks to:
     -- Load read address for next screen Memory
     -- Save the output into CharAddr
     -- Load the address of the character in charrom
     -- Increment charCount
     -- Save new data into buffer
     -- Case for first ~8 counts of eightCounter
     -- End of character count dependent on display mode

     CASE eightCounter IS
      WHEN b"00001" =>
       readAddress_rom <= charCount;
      WHEN b"00011" =>
       charAddr <= dataOutRead_rom;
       invert <= dataOutRead_rom(7); --bit 7 is whether to invert or not.
      WHEN b"00101" =>
       readAddress_rom <= (b"00" & charAddr(6 DOWNTO 0) & b"000") +
charline;
      WHEN b"00111" =>
       IF charCount = CharMemEnd THEN
        charCount <= CharMemStart;
       ELSE --otherwise increase
        charCount <= charCount + 1;
       END IF;
      WHEN OTHERS =>
       --do nothing;
     END CASE;

     --If it hasnt just loaded new data,
     IF eightCounter /= end_of_char THEN
      eightCounter <= eightCounter + 1; --increment counter
      IF bufferCounter = mm_displayMode THEN
       data_buffer <= data_buffer(6 DOWNTO 0) & '0';
       bufferCounter <= b"00";
      ELSE
       bufferCounter <= bufferCounter + 1;
      END IF;
     ELSIF eightCounter = end_of_char THEN
      --clear end of frame flags anywhere before end of frame
      doneEndOfFrame <= '0';
      doneEndOfFrame1 <= '0';
```

```
      doneEndOfFrame2 <= '0';
      eightCounter <= b"00001"; --Reset counter
      IF invert = '1' THEN --invert flag, negate data.
       data_buffer <= NOT dataOutRead_rom; -- grab new data
      ELSE
       data_buffer <= dataOutRead_rom; -- grab new data
      END IF;
     END IF;
    ELSE
     --If its out of visible area, display background
     IF ycounter_in > endy THEN
      lineCounter <= (OTHERS => '0');
      charline <= (OTHERS => '0');
      charCount <= topOfFrame;
      lineStartAddr <= topOfFrame;
      eightCounter <= (OTHERS => '0');
     END IF;

     IF mm_displayMode = b"10" THEN
      redOutput_all <= b"00" & vgared_in(1 DOWNTO 0);
      greenOutput_all <= b"00" & vgagreen_in(1 DOWNTO 0);
      blueOutput_all <= b"00" & vgablue_in(1 DOWNTO 0);
     ELSE
      redOutput_all <= vgared_in;
      greenOutput_all <= vgagreen_in;
      blueOutput_all <= vgablue_in;
     END IF;
    END IF;

   END IF;

  END PROCESS;

  shift_ack_out <= shift_ack;

END Behavioral;
```

Appendix D

Truncated kickstart.a64, showing relevant parts only

```
 ; variables for task switching

 .space ts_current_sector_byte 4
 .space ts_dmalist 10
 .space delayCounter 1
 .space ts_sector_counter 2

 ; make sure that we don't go past the 256 byte page reserved for
hypervisor scratch space
 ;
 .checkpc $CEFF
```

Truncated kickstart_task.a65, showing relevant parts only

```
double_restore_trap:
 ; Double tapping restore button triggers this trap
 ; bump border colour so that we know something has happened
 ;
 lda $D020
 inc
 and #$0f
 sta $D020
 jsr sd_unmap_sectorbuffer ;so we can write to cia regs.

 jsr ts_read_from_sd

 ; return from hypervisor
 sta hypervisor_enterexit_trigger

;=======================

protected_hardware_config:

 ; store config info passed from register a
 lda hypervisor_a
 sta $D672

 ; bump border colour so that we know something has happened
 ;

 lda $D020
 inc
 and #$0f
 sta $D020

 sta hypervisor_enterexit_trigger
```

```
;========================

matrix_mode_toggle:


; Originally used to toggle Matrix Mode
; Repurposed as 'save state' button for now.
; Alt-tab to trigger this trap

; bump border colour so that we know something has happened
  lda $D020
  inc
  and #$0f
  sta $D020

  jsr sd_unmap_sectorbuffer
  jsr ts_write_to_sd

  sta hypervisor_enterexit_trigger

; ========================

; Writes the next sector to SD card.

write_next_to_sd:


; bump border colour
 lda $D020
 inc
 and #$0f
 sta $D020

; enable enhanced registers, idk if this is needed?
 lda #$47
 sta $d02f
 lda #$53
 sta $d02f

; Next Sector
 jsr next_sector
 jsr set_sector
; Next Address
 jsr inc_address_write


; Make sure SD is ready
;jsr wait_for_ready

; Start DMA transfer
 jsr ts_initiate_dma

; Start SD card transfer
 jsr write_to_buffer

; Increment the sector counter
 jsr inc_sector_counter

rts

; ======================
```

```
read_next_from_sd:
; bump border colour
 lda $D020
 inc
 and #$0f
 sta $D020

; Next Sector
 jsr next_sector
 jsr set_sector

; Next Address
 jsr inc_address_read
 jsr inc_sector_counter

; Start SD card transfer
 jsr read_from_buffer

; Start DMA transfer
 jsr ts_initiate_dma

rts


; ======================

ts_read_from_sd:

; zero sector counter
jsr reset_sector_counter

; reset SD card
jsr sdreset
jsr sd_unmap_sectorbuffer

; set inital DMA values
jsr ts_setup_read_address ; initalises DMA list for reading

; read from SD card sector (write into the buffer)
jsr read_from_buffer

; start DMA transfer from buffer
 jsr ts_initiate_dma

; increment the sector counter
 jsr inc_sector_counter

tsr_loop:
; if upper byte of sector counter is zero
; check the lower byte of sector counter for sector 102 (103 skip) to skip
lda ts_sector_counter+1
and #$01
beq checkLowerSector


tsr_cont:

jsr read_next_from_sd
lda ts_sector_counter+1
;RAM+ROM = ?262143? = 512 sectors
```

```
and #$08 ;stop @ sector ~400 768 512-1
bne retr
jmp tsr_loop

;stop on exit
pause_loop:
jmp pause_loop;

retr:
jsr reset_sector_counter
rts

checkLowerSector:
lda ts_sector_counter
cmp #$67 ;Sector 102 (103)
beq tsr_skip ;on zero go to skip
jmp tsr_cont;

tsr_skip:
;Skips a single sector
;increment sector, adress, etc.
 jsr next_sector
 jsr set_sector
 jsr inc_address_read
 jsr inc_sector_counter
;jmp pause_loop
jmp tsr_cont;



; ======================

ts_write_to_sd:
jsr reset_sector_counter
jsr sdreset
jsr sd_unmap_sectorbuffer
jsr ts_setup_write_address ; initalise DMA list for writing
jsr ts_initiate_dma ; initiate the DMA transfer
jsr write_to_buffer ; initiate write to sd
jsr inc_sector_counter

;write loop
tsw_loop:
jsr write_next_to_sd
lda ts_sector_counter+1
and #$08
bne retw ;branch if upper byte of counter reaches 8 (0000 1000 0000 0000 is
2048) or maybe OBO error?
jmp tsw_loop

retw:
jsr reset_sector_counter
rts

; ======================

init:
jsr reset_sector_counter
jsr sdreset
jsr sd_unmap_sectorbuffer
jsr ts_store_dmalist_init;
```

```
rts

; ==================
inc_sector_counter:

lda ts_sector_counter
clc      ;clear carry
adc #$01 ;add one to counter low byte
sta ts_sector_counter

lda ts_sector_counter+1
adc #$00 ;add the carry to low byte
sta ts_sector_counter+1

rts
; ==================

reset_sector_counter:
lda #$00
sta ts_sector_counter
sta ts_sector_counter+1
rts

;====================

wait_for_ready:
  lda $d680
  and #$01 ;Bit test bit 0
  bne wait_for_ready;
rts

; ======================

next_sector:
;increment current sector bytes by 512:

lda ts_current_sector_byte+1
clc
adc #$02
sta ts_current_sector_byte+1

lda ts_current_sector_byte+2
adc #$00
sta ts_current_sector_byte+2

lda ts_current_sector_byte+3
adc #$00
sta ts_current_sector_byte+3

rts

; ========================

set_sector:
  lda ts_current_sector_byte
  sta sd_address_byte0 ; is $D681 --sd_sector(7 downto 0)
  lda ts_current_sector_byte+1
  sta sd_address_byte1 ; is $d682 --sd_sector(15 downto 8)
  lda ts_current_sector_byte+2
  sta sd_address_byte2 ; is $d683 --sd_sector(23 downto 16)
  lda ts_current_sector_byte+3
```

```
  sta sd_address_byte3 ; is $d684 --sd_sector(31 downto 24)
  rts

; ========================

ts_setup_write_address:

lda #$00
sta ts_current_sector_byte
sta sd_address_byte0
lda #$84
sta ts_current_sector_byte+1
sta sd_address_byte1
lda #$D7
sta ts_current_sector_byte+2
sta sd_address_byte2
lda #$17
sta ts_current_sector_byte+3
sta sd_address_byte3

lda #$00 ;DMA list bank (22-16 value(6 downto 0), zeroes addr(27 downto 23)
sta $d702

; Kickstart ROM is at $FFFE000 - $FFFFFFF, so DMA list is at $FF
lda #$00 ;DMA list MB (27-20)
sta $d704

; Copy from MB $7F (7F00000-$00001FF First 512B of RAM)
lda #$7F
sta $d705 ;DMA src MB

; Destination MB, also $FF because enhanced io RAM space ?
lda #$FF
sta $d706 ;DMA dst MB

; Source Address |
lda #$00
sta ts_dmalist+3 ; lower byte
lda #$00
sta ts_dmalist+4 ; upper byte
; Destination Bank
lda #$00 ;
sta ts_dmalist+5


; map to static SD buffer @ FFD6000-FFD61FFF so mapping of colour ram
; doesnt matter

; Destination Address | ... to sd buffer at FF|D|3E00 ($DE00-$DFFF)
;lda #$00
;sta ts_dmalist+6 ; lower byte
;lda #$3E
;sta ts_dmalist+7 ; upper byte

; Destination Address | ... to sd buffer at FF|D|6000
lda #$00
sta ts_dmalist+6 ; lower byte
lda #$60
sta ts_dmalist+7 ; upper byte

; Destination Bank
```

```
lda #$0D ;
sta ts_dmalist+8

; copy + last request in chain
lda #$00
sta ts_dmalist

lda #$00
sta ts_dmalist+1 ; lower byte of copy size
lda #$02
sta ts_dmalist+2 ; upper byte of copy size

; Modulo (not used, but I think needs to be zeroed)
lda #$00
sta ts_dmalist+9
lda #$00
sta ts_dmalist+10


rts

; ======================

ts_setup_read_address:

lda #$00
sta ts_current_sector_byte
sta sd_address_byte0
lda #$84
sta ts_current_sector_byte+1
sta sd_address_byte1
lda #$D7
sta ts_current_sector_byte+2
sta sd_address_byte2
lda #$17
sta ts_current_sector_byte+3
sta sd_address_byte3


lda #$00 ;DMA list bank (22-16 value(6 downto 0), zeroes addr(27 downto 23)
sta $d702

; Kickstart ROM is at $FFFE000 - $FFFFFFF, so DMA list is at $FF
lda #$00 ;DMA list MB (27-20)
sta $d704

; Source MB, also $FF because enhanced io RAM space ?
lda #$FF
sta $d705 ;DMA dst MB

; Destination
lda #$7F
sta $d706 ;DMA src MB

; Source Address | ... from sd buffer at FF|D|3E00 ($DE00-$DFFF)
;lda #$00
;sta ts_dmalist+3 ; lower byte
;lda #$3E
;sta ts_dmalist+4 ; upper byte

; Source Address | ... from sd buffer at FF|D|6000
```

```
lda #$00
sta ts_dmalist+3 ; lower byte
lda #$60
sta ts_dmalist+4 ; upper byte

; Destination Bank
lda #$0D ;
sta ts_dmalist+5

; Destination Address | ... to 7F|0|0000
lda #$00
sta ts_dmalist+6 ; lower byte
lda #$00
sta ts_dmalist+7 ; upper byte
; Destination Bank
lda #$00 ;
sta ts_dmalist+8

; copy + last reqest in chain
lda #$00
sta ts_dmalist

lda #$00
sta ts_dmalist+1 ; lower byte of copy size
lda #$02
sta ts_dmalist+2 ; upper byte of copy size

; Modulo (not used, but I think needs to be zeroed)
lda #$00
sta ts_dmalist+9
lda #$00
sta ts_dmalist+10


rts


; =========================


ts_initiate_dma:

lda #>ts_dmalist
sta $d701
; set bottom 8 bits of address and trigger DMA.
lda #<ts_dmalist
sta $d700

rts


; ========================

write_to_buffer:
;Assuming buffer has data to be written
;and D681-D684 is set to correct sector

lda #$03
sta $d680
jsr wait_for_ready
```

```
rts

; ========================

read_from_buffer:

lda #$02
sta $d680
jsr wait_for_ready
rts

; ========================

;re-write this like the inc_address_read below. but this still works.

inc_address_write:
; Increment Source Address, byte 1, byte 2, and bank
; Byte 1: ts_dmalist+3 - Lower Byte
; Byte 2: ts_dmalist+4 - Upper Byte
; Byte 3: ts_dmalist+5 - Bank

;increment byte 2
addw_inc_b2:
inc ts_dmalist+4
inc ts_dmalist+4
beq addw_inc_b3 ;if byte2 result is zero (overflow)
rts

;increment byte 3
addw_inc_b3:
inc ts_dmalist+5
rts




; ======================

inc_address_read:
; Increment Dest Address, byte 1, byte 2, and bank
; Byte 1: ts_dmalist+6 - Lower Byte
; Byte 2: ts_dmalist+7 - Upper Byte
; Byte 3: ts_dmalist+8 - Bank

lda ts_dmalist+7
clc
adc #$02
sta ts_dmalist+7

lda ts_dmalist+8
adc #$00
sta ts_dmalist+8

rts

; ======================
ts_store_dmalist_init:
;Address set to scratch space @400M
;Initialise SD controller sector bytes

lda #$00
```

```
sta ts_current_sector_byte
sta sd_address_byte0
lda #$84
sta ts_current_sector_byte+1
sta sd_address_byte1
lda #$D7
sta ts_current_sector_byte+2
sta sd_address_byte2
lda #$17
sta ts_current_sector_byte+3
sta sd_address_byte3

;Initalise DMA list

lda #$00 ;DMA list bank (22-16 value(6 downto 0), zeroes addr(27 downto 23)
sta $d702

; Kickstart ROM is at $FFFE000 - $FFFFFFF, so DMA list is at $FF
lda #$00 ;DMA list MB (27-20)
sta $d704

; Copy from MB $7F (7F00000-$00001FF First 512B of RAM)
lda #$7F
sta $d705 ;DMA src MB

; Destination MB, also $FF because enhanced io RAM space ?
lda #$FF
sta $d706 ;DMA dst MB

; copy + last reqest in chain
lda #$00
sta ts_dmalist
lda #$00
sta ts_dmalist+1 ; lower byte of copy size
lda #$02
sta ts_dmalist+2 ; upper byte of copy size
; Source Address | copy from start of RAM at 7F|0|0000
lda #$00
sta ts_dmalist+3 ; lower byte
lda #$00
sta ts_dmalist+4 ; upper byte
; Source Bank
lda #$00
sta ts_dmalist+5
; Destination Address | ... to sd buffer at FF|D|3E00 ($DE00-$DFFF)
lda #$00
sta ts_dmalist+6 ; lower byte
lda #$3E
sta ts_dmalist+7 ; upper byte
; Destination Bank
lda #$0D ;
sta ts_dmalist+8
; Modulo (not used, but I think needs to be zeroed)
lda #$00
sta ts_dmalist+9
lda #$00
sta ts_dmalist+10

rts

;====================
```

Appendix E

Source code is available at: https://github.com/Kirb0031/mega65-core