Curtin University

Department Of Electrical And Computer Engineering

Design and Implementation of an FPGA Based Thin Client

by

Liam Davey

A thesis submitted for the degree of

Bachelor of Engineering in Computer Systems Engineering

Curtin University

Department Of Electrical And Computer Engineering

Title: Design and Implementation of an FPGA Based Thin Client	
Author: Family Name: Davey Given Name: Liam	
Date: 05/10/11	Supervisor: Cesar Ortega Sanchez
Degree: Bachelor of Engineering in Computer Systems Engineering	Option:

Abstract:

An FPGA based Thin Client solution to the problem of compute resource consolidation was designed, implemented and verified. A prototype FPGA Thin Client was created that communicated with a remote server using raw Ethernet frames and uncompressed RGB image data. The aim was to create a Thin Client device that was as simple and inexpensive as possible, and to reduce the computational overhead at the server by trading a reduction in processor usage for an increase in network bandwidth required.

Indexing Terms:

FPGA, Thin Client, Virtualization, Ethernet, HDMI

	Good	Average	Poor
Technical Work			
Report Presentation			

Examiner:	Co-Examiner:

Synopsis

An FPGA based Thin Client solution to the problem of compute resource consolidation was designed, implemented and verified. A prototype FPGA Thin Client was created that communicated with a remote server using raw Ethernet frames and uncompressed RGB image data. The aim was to create a Thin Client device that was as simple and inexpensive as possible, and to reduce the computational overhead at the server by trading a reduction in processor usage for an increase in network bandwidth required.

Dear Prof Syed Islam,

I, Liam Davey, offer this thesis as partially satisfying the requirements of my Bachelor of Engineering in Computer Systems Engineering degree. The thesis is composed entirely of my own work except for the references quoted.

Yours sincerely,

Liam Davey

Signature:

Acknowledgments

Cesar Ortega Sanchez for supervising the project.

Curtin University of Technology for providing a Partial Engineering Scholarship.

TABLE OF CONTENTS

Table of Figures

1.0 Introduction

Computers are becoming more powerful as each year passes. Often users do not fully utilize the computational power provided by their compute stations. Desktop virtualization is one way of consolidating multiple user desktops onto one powerful computer, such as a server in a data center. However desktop virtualization requires a way for individual users to access their virtualized desktops. Often the user is situated at a physically distant location to the server so that some method of transmitting the video data to the users monitor is required. Current solutions compress the video data at the server before transmitting it via a network to a client device at the users station. The client device decompresses the video data and displays it on the users monitor. The problem with this current solution however is the high computational overhead at the server caused by compressing video data for multiple users remote users.

The solution proposed is to forgo the compression and instead transmit uncompressed RGB data to the user stations. This requires more communication bandwidth, but reduces the computational overhead at the server. To decode this data and display it on the users monitor an FPGA based Thin Client device was proposed. The FPGA based solution allowed high bandwidth communications to be used (such as Gigabit Ethernet) at minimal cost and design complexity.

A prototype of the FPGA based Thin Client was designed, implemented and verified. It was tested and found to provide an adequate solution to the problem, although a few drawbacks and disadvantages of the proposed FPGA based solution were discovered in the process.

In chapter 2.0, Review of Current Thin Client Solutions, several current solutions to the problem of compute resource consolidation are outlined. The advantages and disadvantages of each are analyzed. The reasons for choosing the FPGA based Thin Client solution are outlined in chapter 3.0, Problem Solution. This chapter is followed by chapter 4.0, Implementation, which covers the selection of the hardware platform, the top level design, the design and implementation of all FPGA modules, and the design and implementation of the server software. Chapter 5.0, Verification, outlines how each module was simulated and tested, how the software was tested, and how the functionality of the complete system was tested. The final part is chapter 6.0, Conclusions, which analyzes the results achieved, future improvements of the design, and future developments to the solution.

2.0 Review of Current Thin Client Solutions

2.1 The Thin Client Model

2.1.1 Overview

The Thin Client Model is a computing model which allows multiple users to access and utilize a single host computer (server). Each user operates a Thin Client device, which is usually a relatively low powered computer that communicates with the host computer via a network interface, drives a graphical display, and handles input devices (such as a keyboard and mouse). Figure 2.1 shows how multiple Thin Clients would connect to a Server.

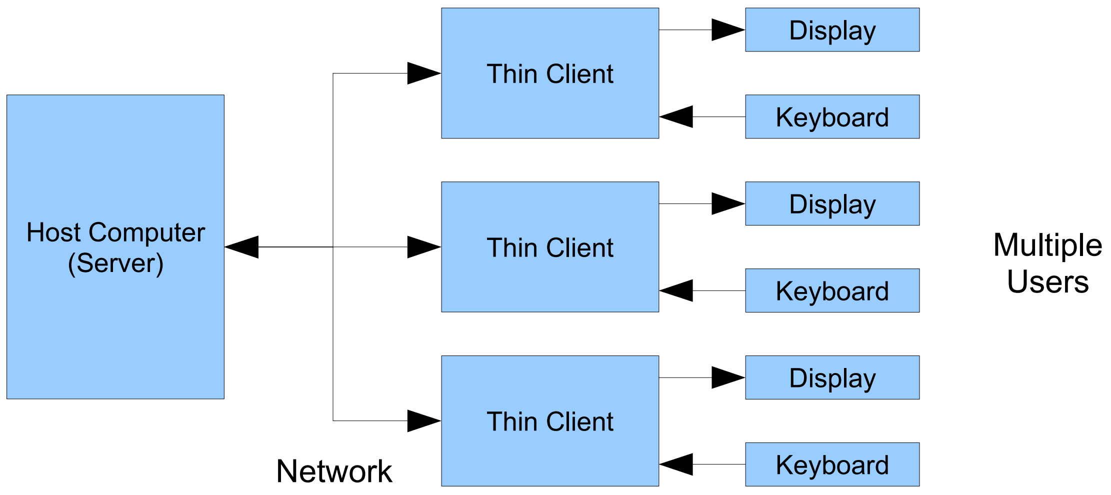


Figure 2.1: The Thin Client model

Depending on the specific configuration, the host computer can be used to provide storage and/or computing power. A few broad categories can be defined based on the amount of work performed on the server compared to the amount of work performed on the client.

2.1.2 Thick Client

Thick Clients usually only use the server for secondary storage, not for computing. An example would be a diskless client which uses PXE (Preboot Execution Environment) to load a kernel into memory, and then boot using an NFS (Network File System) share as the root file system [1]. This allows the storage systems (hard disk drives) to be consolidated on the server, easing administration and increasing reliability, as servers often use reliable storage technology such as RAID (Redundant Array of Independent Disks). The server need not be very powerful in terms of processing capability as the clients perform most of the processing required themselves.

2.1.3 Thin Client

Thin Clients use the server not only for storage, but also for computing power. An example would be a diskless Linux client that uses X forwarding to run graphical applications on the server with the graphical output data being sent to the client for display [2]. The server must be powerful enough and have enough memory to run the applications of multiple clients at once. However the clients can be less powerful (old or inexpensive hardware) as they only need to be able to decode the network protocol and drive a display. All intensive processing is done on the server.

2.1.4 Zero Client

Zero Clients are similar to Thin clients but even more minimal. They do not run a full Operating System, only the minimum amount of software needed to connect to the server using the chosen network protocol (for example Virtual Network Computing (VNC) or Remote Desktop Protocol (RDP)). They are often embedded devices with custom hardware and software (or firmware). An example would be an ARM based System on Chip platform running software whose only purpose is to initiate VNC connections, display the compressed video data and send input data to the server [3].

2.1.5 FPGA Based Zero Client

A alternate form of Zero Client is one implemented using a Field Programmable Gate Array (FPGA) rather than the more common embedded processor or SoC platform. At a low level an FPGA device is a programmable array of simple logic elements. These simple logic elements may be linked together to form more complex logic structures by programming the interconnect 'fabric' or 'mesh'. In this way logic structures such as arithmetic units, memory blocks, and even entire processors can be created within an FPGA. By linking together these logic structures, or modules, all functionality needed in a Zero Client can be implemented using an FPGA.

The main advantage of an FPGA implementation over a SoC implementation is the flexibility: the hardware can be configured to exactly meet the requirements. This means that modules can be written to accelerate tasks that would otherwise need to be performed slowly in software on the SoC sequential processor. Indeed an FPGA based zero client might not use a programmable processor at all. All tasks such as decoding and displaying data from the server could be entirely performed using a pipeline of hardware modules.

Pano Logic produces an FPGA based zero client called the Pano Zero Client [4]. The Pano Zero Client device connects to a server via a 10/100 Fast Ethernet Port, and displays decoded frames on a DVI or VGA connected monitor. It also has 4 USB ports for a keyboard and mouse and an audio port. The Pano Zero Client is interesting because it does not use a programmable processor. All processing, even the display drivers, audio drivers and USB host controller drivers run on the server.

Although details are scarce, to achieve this level of functionality it appears that the Pano Zero Client emulates standard PC audio and video adapters, such that unmodified drivers can be used within a virtualized guest machine running on the server (see section 2.2 for more details on desktop virtualization). Then any time the guest system accesses the system bus (PCI bus on x86 PCs) the request is translated to a network protocol and forwarded over Ethernet to the client. In this way the video and audio adapters appear to the virtualized guest to be connected locally via PCI, but in reality are remote devices connected via Ethernet.

2.1.6 Client Server Communication

In all categories the thin client communicates with the server over a network interface. In most cases the bandwidth available on the network interface is less than the raw bit rate of the RGB video data the server needs to output. In these cases some form of encoding or compression is necessary. Some examples of simple encoding methods are: only transmitting portions of the screen that change between frames, run length encoding regions of the same colour, or lowering the frame rate by not sending all of the frames generated by the server.

JPEG and MPEG compression algorithms are often used on lower bandwidth links where higher compression is needed. The higher compression reduces the network bandwidth required to transmit video data. However, more processing power is needed at both the server to compress the data and at the client for decompression, when compared to transmitting uncompressed data. The increased processing power required for decompression limits how “thin” the client can be.

2.1.7 Advantages of the Thin Client model

- Reduction of hardware costs for workstations. Each user only needs a low powered computer to access the shared resources of the server.
- Easier administration. System wide changes and updates easier to propagate to clients as all software.
- Reduction of total storage needed as shared software and data does not need to be duplicated for each client.
- Reduction of total computing resources needed based on the assumption it is unlikely for all users to need full computing power at the same time.
- Computing hardware can be moved to a central location, reducing infrastructure costs.
- Reduced power usage due to less total computing resources.

2.1.8 Disadvantages of the Thin Client model

- Poor implementations can be frustrating to use, because of network latency, low screen refresh rates, and low picture quality.
- Depending on available network bandwidth and video compression method used some software, such as graphically intense games, may not be usable from a Thin Client.
- Server becomes a single point of failure. If the server fails then all connected Thin Clients also fail.
- Issues with software licensing. Some software licensing, especially per user licensing, may be incompatible with the shared Thin Client model.

2.2 The Virtualized Desktop Model

A Virtualized Desktop computing model is where each client accesses a separate ‘Desktop’ or virtualized Operating System (OS) on a server [5]. This can be easier to implement than a non-virtualized Thin Client model as each client’s ‘Desktop’ can be a disk image of a completely standard OS. Figure 2.2 shows the separate guest operating systems on a single host, each guest OS being dedicated to a single user.

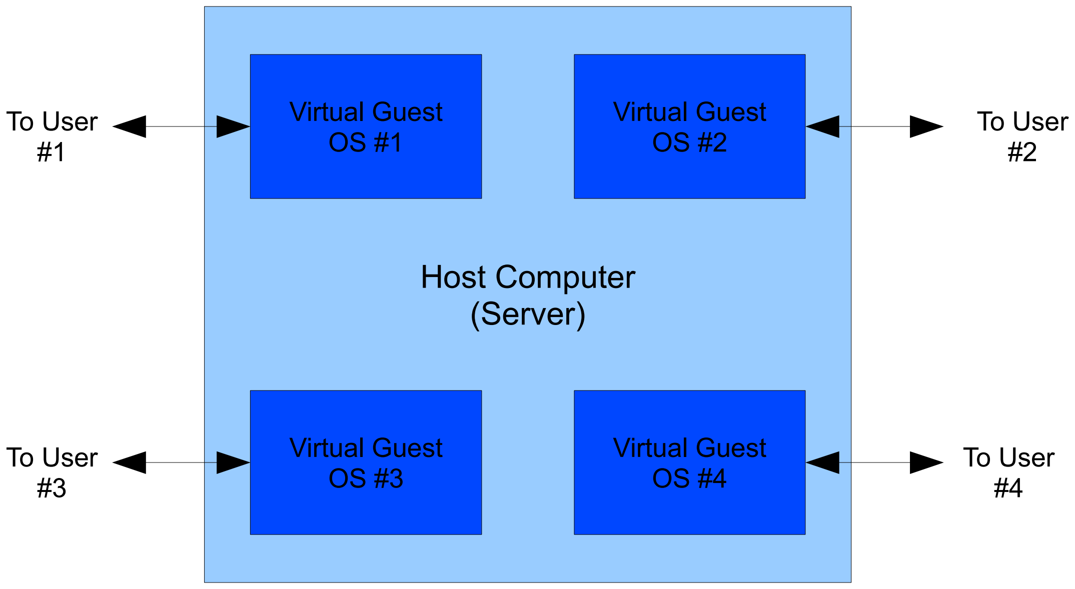


Figure 2.2: Virtualized Desktop Model

This virtualized desktop model allows storage and processing hardware to be consolidated while maintaining the familiarity and compatibility of a standard operating system. The drawback is that as opposed to a non-virtualized thin client model, where only the applications are duplicated for each client, with a virtualized desktop model, the operating system as well as the applications need to be duplicated for each client. This increases the amount of computing and memory resources needed when compared with the non-virtualized model.

2.3 Multiseat Desktop Configuration

The Multiseat Desktop configuration is similar in purpose to the Thin Client model, in that it allows multiple users to access a single powerful computer. However unlike the Thin Client model the video and input device data is not transported over a network link, instead multiple input devices (keyboards and mice), and multiple output devices (displays), are directly connected to the computer using the native interface for the device (Digital Video Interface (DVI) or Universal Serial Bus (USB) for example). Figure 2.3 shows how multiple input and display devices are connected to a single host computer.

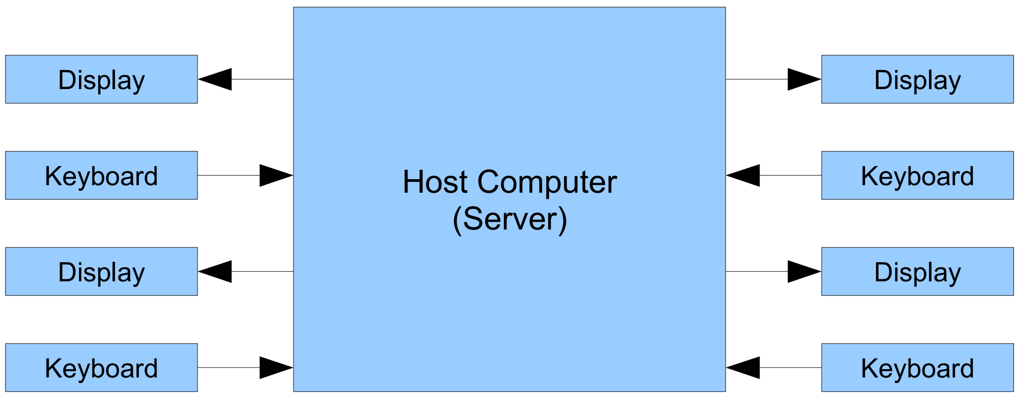


Figure 2.3: Multiseat Desktop Configuration

An advantage of this is that the processing overhead per user is reduced as no translation to a network protocol needs to be performed, nor any video compression. Also the hardware costs are reduced compared to the thin client model as each user can use a standard monitor, keyboard and mouse with no additional hardware, compared to the thin client model where an endpoint device (albeit low powered) is needed for each client to decode the network protocol and decompress the video data.

There are two main ways two main ways to implement a Multiseat Desktop configuration, one being a specialised operating system capable of handling multiple displays and input devices. The second is to use Desktop Virtualization to provide each user with a separate virtualized desktop using a standard operating system. Note that the underlying virtualization ‘shell’ or ‘hypervisor’ must still be capable of driving multiple displays.

The main disadvantage is that interfaces such as DVI and USB cannot be used for communication over as large distances as network interfaces, such as Ethernet. The maximum recommended cable length for both DVI and USB is 5m [6][7]. This means that the host computer must be located nearer to the input and display devices than if a network interface was used, for example, in the same room rather than in a separate building. This limits the amount of users that can connect to a single computer and thus the level of hardware consolidation possible.

Another issue with Multiseat Desktop configurations is that multiple graphics cards may be needed to drive the connected displays. This issue is less of a problem recently however with new graphics cards being able to drive up to 6 displays at the same time.

2.4 KVM over IP

KVM is in a sense the counterpart to the Multiseat Desktop model, in that it allows a single user to control and access multiple computers, rather than giving multiple users access to a shared computer. Standard KVM uses the native display and input device interfaces (DVI, VGA, USB, PS2) and as such can only be used at a distance limited by those links. When the computer that the KVM operator needs to connect to is more distant than the standard links allow for KVM over IP is used instead. In KVM over IP implementations the display data produced by the computer the user wishes to connect to is translated to IP packets by a separate device connected to the computer. These IP packets are then sent over standard network links to the user where the IP packets are translated back and used to drive a display. In a similar way data from the input device translated into IP packets and sent to the host computer. The layout of an example KVM over IP implementation is shown in Figure 2.4.

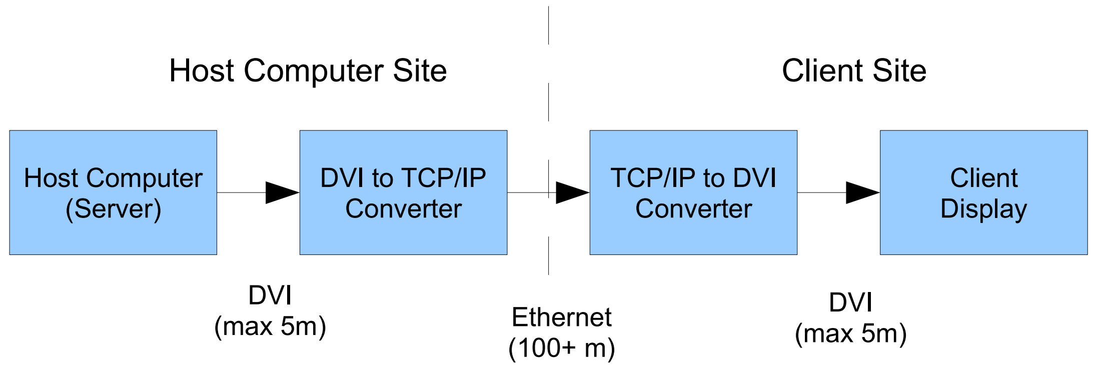


Figure 2.4: KVM over IP video data communication protocol conversions

The devices that translate between display signals and IP packets can be implemented as either external hardware or internal add-on cards in the host computer. When implemented as external hardware the device must convert the display interface (DVI, HDMI or VGA for example) to IP packets transmitted over a network interface, such as Ethernet. As with the Thin Client model the video data can be compressed to reduce bandwidth requirements.

The second option is to use an add on card, similar to a standard video card, that directly outputs IP packets rather than video data on a standard video interface. If the add on card appears to the host computer as a standard video card then no custom drivers would be necessary for the Operating System. An advantage of this option is that it reduces the number of signaling conversions needed as the server directly outputs correctly encoded TCP/IP data. A disadvantage is that the add-on card is unlikely to be as powerful as a normal video card for the price (due to low demand).

3.0 Problem Solution

3.1 Overall Requirements

There are many different ways the problem of compute resource consolidation can be solved, even if only partially. A solution needed to be chosen that best satisfied the requirements. The main, or overall, requirements of the solution were as follows:

- The solution must consolidate compute resources traditionally supplied by multiple workstations into a single server or data center.
- The solution must allow users to access the compute resources at a distance of up to 50 meters from the server.

Other desired aspects were that the solution should be:

- As open as possible. The solution should not use proprietary protocols or software and should work with open source software running on the server (for example, a Linux host operating system running virtualized guests under QEMU/KVM).
- As inexpensive possible. The solution should use existing infrastructure where possible, for example, it should use standard CAT5 or CAT6 Unshielded Twisted Pair (UTP) cabling between the client and server rather than needing new cabling to be installed.
- As consolidated as possible. There should be as little as possible processing occurring at the client. All storage, memory, and compute resources should be physically located at the server.
- As unobtrusive as possible. If a client device is used it should have a small

physical footprint, be electrically efficient, and operate quietly.

Each different method of resource consolidation (thin client, thick client, zero client, multiseat desktop) offered different strengths and weaknesses. The FPGA based zero client solution was selected after carefully considering each of the possible solutions researched. The following sections justify why each possible solution was chosen or not chosen.

3.2 Multiseat Desktop

The multiseat desktop configuration offers the lowest latency and closest experience to using a traditional workstation of all the solutions considered. As the graphical output is only transmitted over the native display interface, no computationally expensive translation to a network protocol is needed. Another advantage is that the multiseat desktop configuration is truly 'zero client' in that no client device is needed at all. The only device at the users station is a display, keyboard and mouse. This saves costs per user when compared to Zero, Thin, and Thick Client solutions.

However, the multiseat configuration was deemed not to be a suitable solution to the problem because it could not satisfy the requirement of up to 50m distance between the server and client station. The greatest strength of the multiseat desktop, that of using the native display interface, is also the greatest weakness, as the native display interface is limited to short cable lengths. Even if the native display interface could function at up to 50m distances, it would be costly and inefficient to install separate point to point cabling for every station connecting to the server (compared to Ethernet which uses a star topology to fan out connections from the server to each station).

3.3 Traditional Thick and Thin Clients

Thick and Thin Clients both communicate with the server via a network connection, commonly Ethernet, which satisfies the requirement of up to 50m distance between the client and server. Using Ethernet would also be inexpensive for most organisations as existing UTP cabling and network infrastructure could be used.

However neither a traditional Thick or Thin Client solution was found to be suitable because: the cost per client was too high, too much processing was needed at the client, and the client device was more obtrusive than necessary.

3.4 FPGA Based Thin Client

The chosen solution was an FPGA based Thin Client, which might also be called a 'Zero' Client solution. The term 'Zero Client' is misleading however as it implies that no client device is necessary, so henceforth only the term 'FPGA Based Thin Client' will be used.

An FPGA based thin client solution was chosen because it had the advantages of the traditional Thin and Thick Client solutions, including the possibility of using existing network infrastructure and the required 50m distance. The FPGA based solution also offered lower cost per client device, less processing overhead on the server, greater flexibility, lower power requirements and lower physical footprint.

4.0 Implementation

4.1 Overview

The first step after the FPGA based thin client solution was selected was to identify the external interfaces the thin client would need. This would allow an appropriate hardware platform to be selected for development.

The thin client needed 3 basic external interfaces: a network interface to communicate with the server, a display interface to drive the graphical display (LCD monitor for example), and an input device interface to receive keyboard scan codes and mouse movement data. These external interfaces are shown in Figure 4.1.

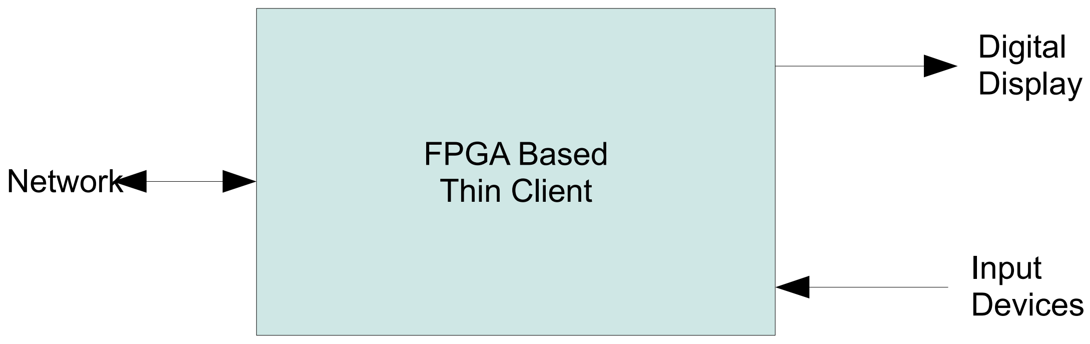


Figure 4.1: External interfaces necessary for the thin client

The next step was to select the specific standard to use for each of the external interfaces.

4.2 External Interface Standard Selection

4.2.1 Network Interface

The network interface needed to be inexpensive, readily available and use common existing network infrastructure. This limited the selection to Fast Ethernet, Gigabit Ethernet, or 802.11g/n Wireless Ethernet.

802.11g/n Wireless Ethernet has the advantage over Fast Ethernet and Gigabit Ethernet of not needing any cabling between the client and server, which would allow the client device to be somewhat portable. However, Wireless Ethernet has the disadvantages of lower bandwidth, higher cost, and reduced reliability compared to a wired solution. Another disadvantage is that Wireless Ethernet uses a shared medium for communication. This means that the 54Mbit/s for 802.11g and 300Mbit/s for 802.11n maximum theoretical bandwidths are shared between all devices on the network, further lowering the practically attainable bandwidth on a busy network.

This is less of a problem for wired Ethernet, where a faster upstream link can be fanned out using network switches to supply each client with the full bandwidth available. This is shown in Figure 4.2.

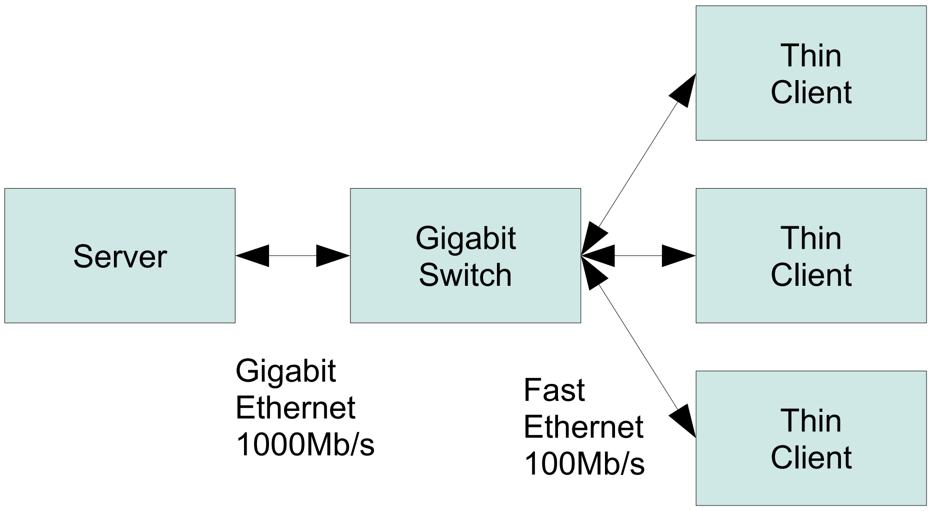


Figure 4.2: Ethernet fan out with Fast Ethernet clients

The main difference between Fast Ethernet and Gigabit Ethernet is that Gigabit Ethernet offers 10 times the bandwidth (100Mbit/s for Fast Ethernet, 1000Mbit/s for Gigabit Ethernet) for a slight increase in transceiver cost. If uncompressed frames were transmitted to each client the bandwidth required would be 843.75Mbit/s for 1280x720 resolution frames at 30 Hz with 32 bits per pixel. While this is a worst case scenario (usually only the updated regions of a frame would be transmitted), it illustrates the large amount of bandwidth required for uncompressed video. Uncompressed video is desirable because it reduces the processing overhead at both the server and client.

Gigabit Ethernet is rapidly replacing Fast Ethernet as the standard networking interface, and the cost of a Gigabit transceiver over a Fast Ethernet transceiver at the client would be at most a few dollars per client. Hence Gigabit Ethernet was the standard chosen for the client external network interface.

4.2.2 Digital Display Interface

Three different display interface standards were considered: VGA, DVI and HDMI. VGA is an analog standard mainly used on older CRT monitors. One drawback of VGA is that to implement it using an FPGA would require a fast Digital to Analog Converter (DAC) chip to produce the analog Red, Green and Blue signals, adding to the design cost and complexity. Another drawback of VGA is that the conversion from digital to analog at the FPGA, and then from analog to digital to drive an LCD monitor, results in a reduction of image quality.

DVI has neither of these drawbacks. It does not need any digital to analog conversion to drive a display, and the physical layer can be implemented entirely within an FPGA (using no external transceivers or encoders).

The other option was to use HDMI. HDMI is electrically compatible with the DVI connector, meaning that the same FPGA configuration could support either DVI or HDMI simply by changing the connector. HDMI does however extend the DVI specification somewhat to allow audio data to be transmitted along with video data over the same link. This capability would be lost if a DVI connection was used.

The main reason DVI was chosen over HDMI was that HDMI requires a royalty fee to be paid for each connector on a device. Also it was decided the audio transmission capability of HDMI would not be of any use in thin client applications (as most commodity LCD monitors do not decode the audio data).

4.2.3 Input Device Interface

The input device interface standard was a choice between USB and PS/2. PS/2 is a simple standard that is easy to implement on an FPGA with no external components. However it is a legacy standard, and less devices are being produced implementing it as it is gradually superseded by USB. Compared to PS/2, USB is a relatively complex protocol. This is partly because it supports many different classes of device, ranging from low speed Human Interface Devices (for example keyboards, mice, and joysticks), to high speed mass storage devices, cameras, and network interfaces.

It is technically possible to implement a USB host supporting a small range of USB devices entirely in hardware (using a very elaborate stack machine for example), however this is rarely done in practice. It is much more common to have a hardware Host Controller that manages the physical layer and low level USB transactions, and software drivers for each device or class of device that will be connected to the host controller [8]. A software driver for the host controller is also needed. One advantage of this approach is that all low level USB operations are performed by the fully tested hardware host controller, abstracting the physical and transaction layer complexities, saving development time. Another advantage of this approach is that supporting new devices is as easy as adding another software driver. The number of devices supported is then only limited by the program memory size.

However, needing a processor and program memory storage at the Thin Client to support USB devices goes against the design goal of maximum compute resource consolidation. One way around this is to have the host controller driver and all device drivers run remotely on the server, similar to the Pano Zero Client [4]. This approach is shown compared to the traditional approach in Figure 4.3.

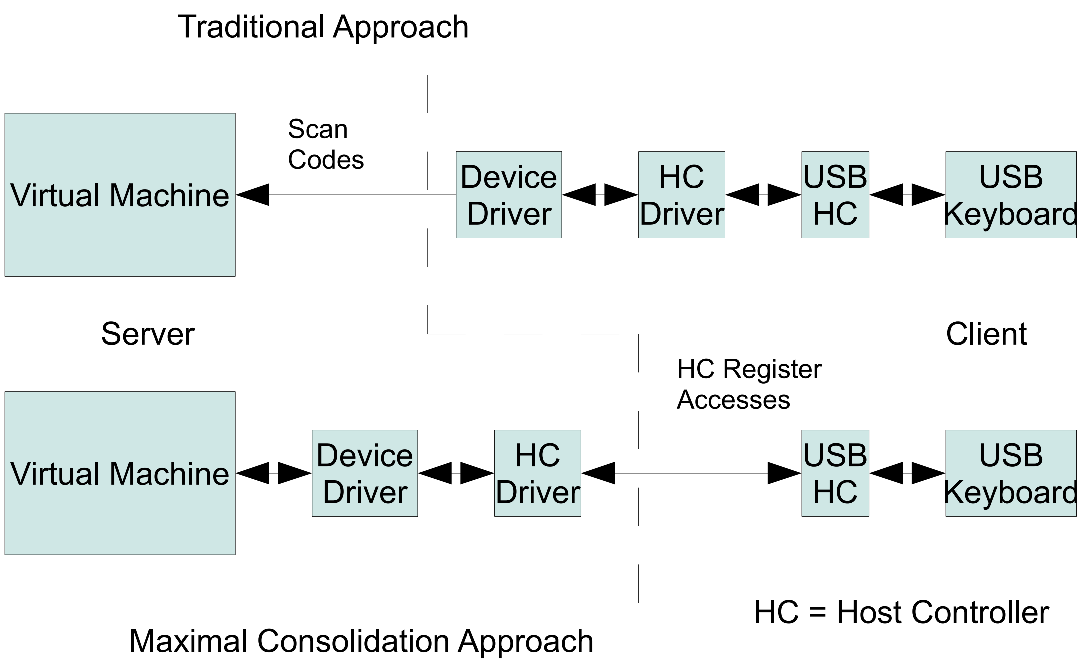


Figure 4.3: USB Host processing consolidation

As can be seen the software device drivers and associated processing are performed at the Server rather than at the Client with the second approach. Rather than the scan codes produced by the device driver (in the specific case of a USB keyboard), host controller register accesses generated by the host controller driver are transmitted over the network link.

If the FPGA based thin client was to be extended beyond the proof of concept stage, a USB interface would have been chosen over a PS/2 interface to support a greater range of devices (for example USB storage devices and USB cameras) and allow maximum future flexibility. The additional complexity and expense of attaching a USB Host Controller would be well worth the features a USB Host port would provide compared to a PS/2 interface. However, due to the lack of availability of FPGA development kits with USB host controllers or PHY chips, a PS/2 interface was selected instead.

4.3 Choosing the Hardware Platform

After the external interface standards were chosen the next step was to decide on a hardware platform to implement the solution on. Based on the decisions in the previous section, the hardware platform needed to have a Gigabit Ethernet interface, a DVI interface, and a PS/2 interface. These external interfaces are shown in Figure 4.4.

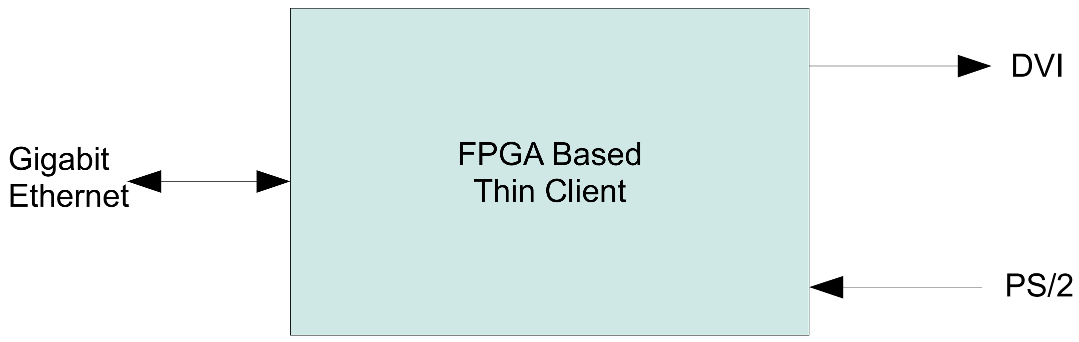


Figure 4.4: Protocols chosen for external interfaces

Finding an inexpensive FPGA development board with each of the required interfaces was difficult. The best match was the Digilent Atlys FPGA Development Board [9]. The Atlys board has the following relevant features:

- Xilinx Spartan 6 LX45 FPGA
- 128MB DDR2 Memory

- Marvell 88E1111 Gigabit Ethernet PHY
- One buffered and one unbuffered HDMI output ports
- USB HID Host microcontroller which communicates with the FPGA over two PS/2 interfaces.

The Atlys board uses HDMI connectors rather than the desired DVI connectors, but as the two interfaces are electrically compatible, this was deemed an acceptable compromise. The other compromise was that instead of a USB PHY or PS/2 connector, the Atlys board uses a 16 bit PIC microcontroller acting as an embedded USB Host. The microcontroller only supports one connected USB HID device at a time (either a keyboard or mouse, but not both at the same time). If a USB keyboard is connected the microcontroller sends keyboard scan codes over one of the PS/2 interfaces to the FPGA. Likewise if a USB mouse is connected it sends mouse movement data over the other PS/2 interface. The Atlys development board is pictured in Figure 4.5.

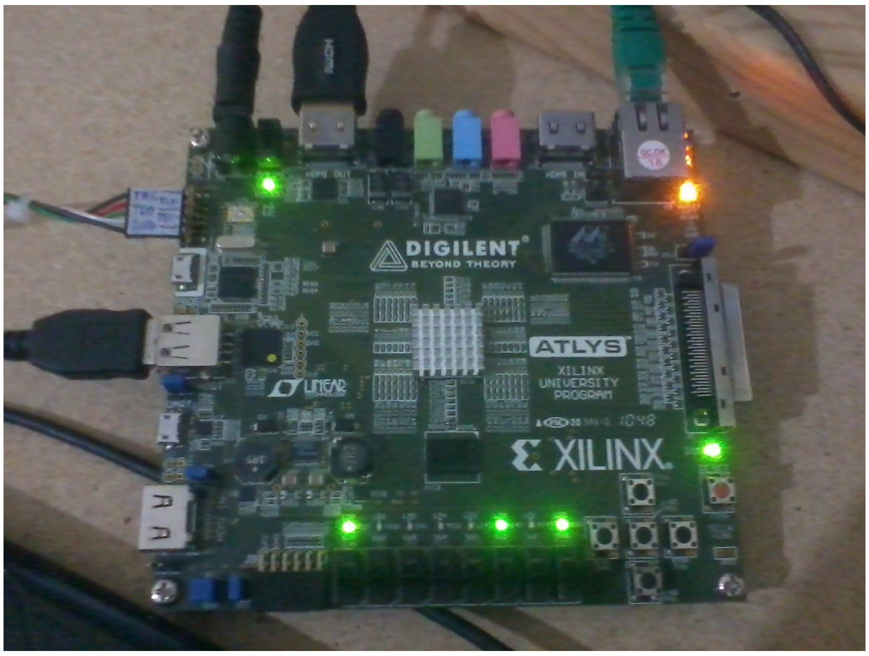


Figure 4.5: The Atlys Spartan 6 Development Board

4.4 Top Level FPGA Design

After the hardware platform was chosen the next step was to design the top level of the FPGA design. The requirements are as follows:

- The design must be able to send and receive Ethernet frames via the GMII connection to the Marvell PHY. This should be implemented with an Ethernet MAC module.
- The design must be able to decode the RGB pixel data contained within received Ethernet frames and must be able to encode PS/2 data into Ethernet frames to be sent. This should be implemented with an Ethernet Decode and Ethernet Encode module.
- The Ethernet Encode module needs to write decoded RGB pixel data to the frame buffer held in DDR2 DRAM. External DDR2 memory needed to be used for the frame buffer as a full uncompressed frame would not fit in the internal block RAM available on the FPGA.
- All DDR2 accesses must be made through a DDR2 controller module.
- The DDR2 controller module should provide at least one read and one write port to access the DRAM.
- The design must be able to read data from the PS/2 interface and send it to the Ethernet encode module.
- The design must be able to read RGB pixel data from the frame buffer held in DDR2 DRAM.
- The design must be able to generate the required video signals (for example, HSYNC, VSYNC, and DE), synchronised with the RGB pixel data read from

the framebuffer.

- The design must encode the video signals and RGB pixel data for transmission over the HDMI output port.

The design shown in Figure 4.6 was developed from these requirements.

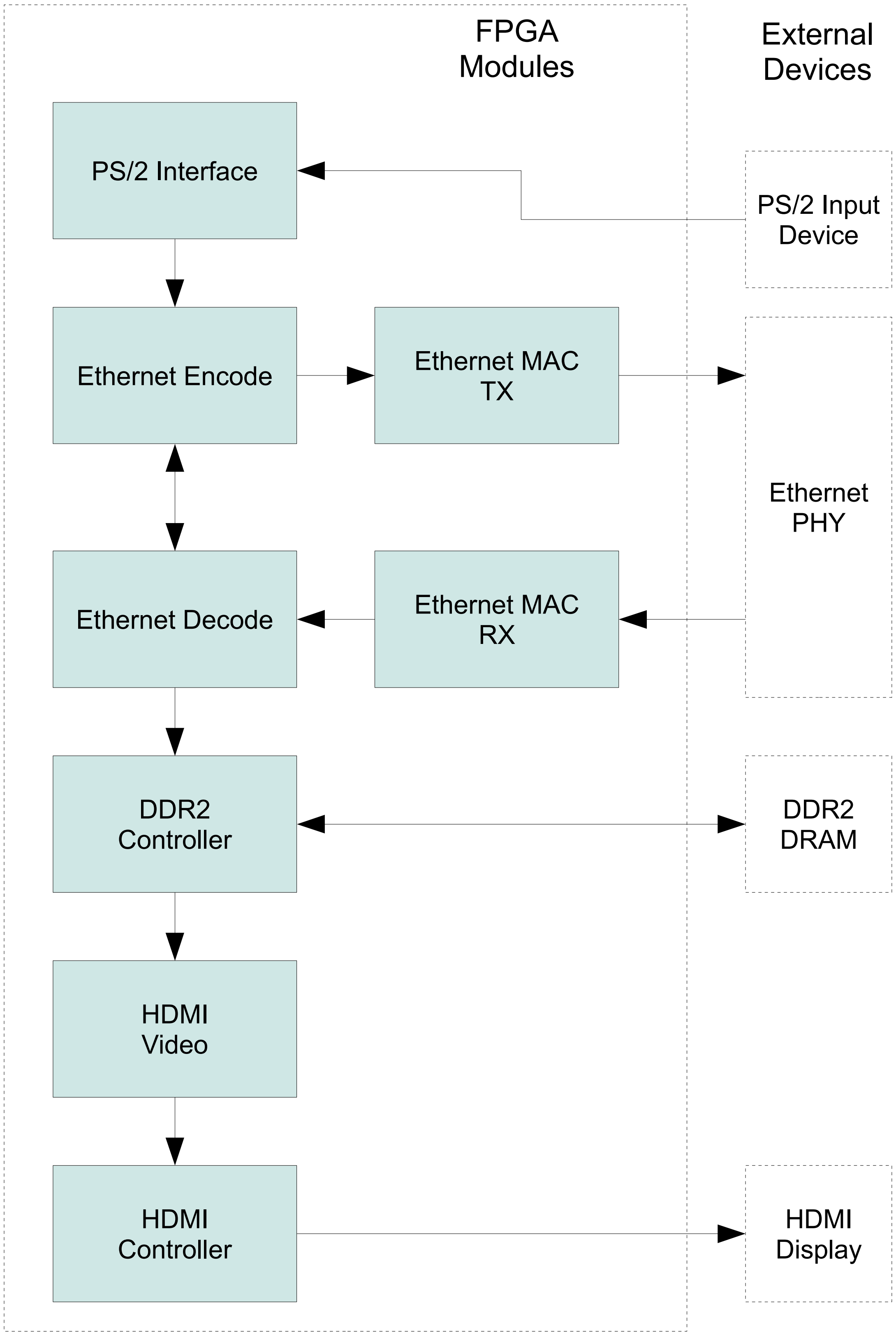


Figure 4.6: Top Level Modules

As can be seen in Figure 4.6 the design consists of several top level modules. The main function of each module is summarized in Table 4.1.

Module Name	**Function**
Ethernet MAC RX	Receives Ethernet frames via the GMII to the external PHY. Checks frame CRC. Passes frames without errors to the Ethernet Decode Module.
Ethernet Decode	Checks Ethernet frame MAC address, TYPE field, and passes valid RGB pixel data to the DDR2 Controller write port.
DDR2 Controller	Provides a FIFO interface to the external DDR2 memory. DDR2 physical layer handled by the Spartan 6 MCB hard core.
HDMI Video	Reads RGB data from the DDR2 Controller. Generates HSYNC, VSYNC, and DATA ENABLE signals.
HDMI Controller	Encodes the RGB data, HSYNC, VSYNC, DATA ENABLE and pixel clock signals using the Transition Minimized Differential Signalling (TMDS) algorithm. Serialization and differential signalling handled by Spartan 6 blocks.
PS/2 Interface	Reads keyboard scan codes or mouse movement data over a PS/2 interface and sends it to the Ethernet Encode module.
Ethernet Encode	Creates Ethernet frames containing PS/2 data and passes the frames to the Ethernet MAC TX module.
Ethernet MAC TX	Generates CRC for input Ethernet Frames and outputs the frames over the GMII to the external PHY.

Table 4.1: The functions performed by each FPGA module

Another decision was which HDL (Hardware Design Language) to use for the implementation of the design. The Xilinx ISE tool suite supported either VHDL or Verilog. The author had some previous experience with VHDL but none with Verilog. However Verilog was the language chosen because the syntax and semantics seemed more intuitive. Another reason Verilog was chosen over VHDL was for the learning experience of designing with a new language.

4.5 Alternative Design Using Microcontroller Core

An alternative top level design to the one outlined in the previous section was also considered. In the alternative design the Ethernet decode and encode modules were replaced with a microcontroller core implemented in FPGA logic. The microcontroller would have performed the functions of both the decode and encode modules, and also arbitrated interaction between all other modules.

The alternative top level design using a microcontroller core is shown in Figure 4.7.

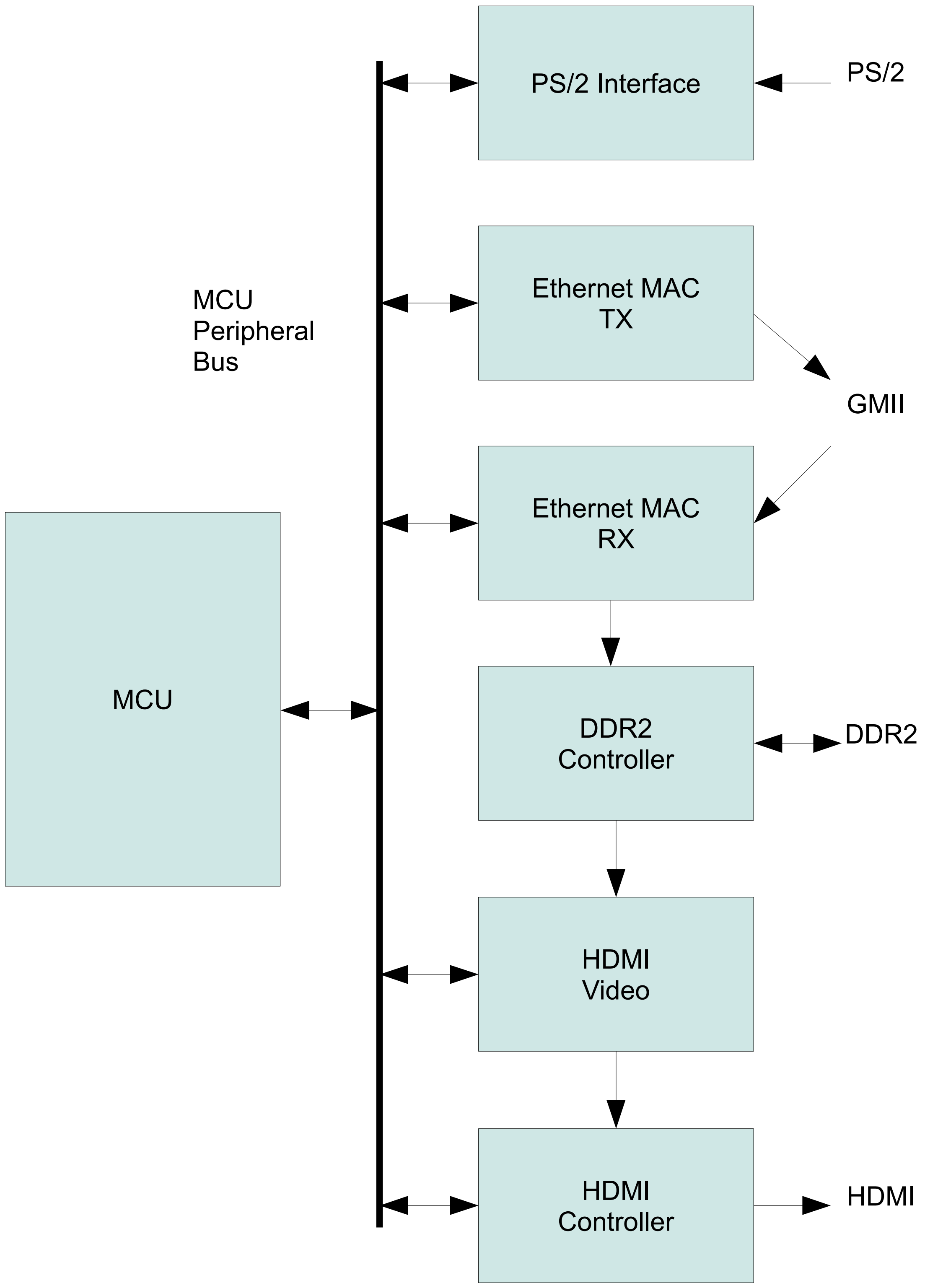


Figure 4.7: Alternative top level design with MCU

A few different processor cores were considered including the OpenRISC 32 bit RISC core, the Amber 32 bit ARM compatible core, and the OpenMSP 16 bit MSP compatible core. The more complex Amber and OpenRISC cores offered greater performance than the simpler OpenMSP core, and more features, such as instruction and data caching, external DRAM as main memory, a Wishbone peripheral bus, and a few ready made peripherals (including an Ethernet MAC peripheral). The OpenMSP core, being a simpler design, used less FPGA resources and was easier to modify for specific purposes. For more details about these processing cores and the Wishbone peripheral bus see the OpenCores website [10].

As each of these processing cores only operated at a relatively low frequency, some processing would still be needed to be offloaded to hardware. In particular the transfer of RGB pixel data from the Ethernet MAC receive buffers to DDR2 DRAM would be very inefficient to perform purely in software. It would be much more efficient to use a hardware DMA peripheral to copy the data instead. Using a DMA peripheral the microcontroller would only need to set the start address and how many bytes to copy, and the DMA peripheral would perform the actual transfer. This would free the processor core to perform other tasks while the transfer was taking place. The DMA peripheral would then trigger an interrupt to let the processor know when the transfer had completed.

The processor core would then only need to decode the received frame headers, write frame headers for frames to be sent, and respond to interrupts generated by the peripherals. For example, the PS2 module might generate an interrupt when a scan code had been received on the PS2 interface. The processor core would then read the scan code from a special memory location mapped to the PS2 peripherals registers and create an Ethernet frame to be sent to the server containing the read scan code.

The advantages of using a processor core over an implementation entirely in hardware are:

- Complex network protocols such as TCP/IP and DHCP would be easier to implement in software than in hardware.
- Existing software from other projects (such as a TCP/IP stack) could be used, saving development time.
- The modules and peripherals written for this project would be more easily used in other future projects if written to use a standard bus interface.
- Software written for the processor core in this project might be reusable in other future projects.

Some disadvantages of using a processor core are:

- Added complexity, more that could go wrong in implementation and verification.
- A toolchain would need to be set up to compile software. This might be problematic for a non-standard configuration.
- Design and implementation of the software/hardware interface would be difficult. It is easier to interface HW to HW or SW to SW than SW to HW.
- The processing core would bottleneck the Gigabit Ethernet interface if not designed and optimized carefully.
- One aim of a Thin Client is to move as much processing as possible to the server. Having a processor core at the client device when not absolutely necessary is counter to this aim.

The suitability of using a processing core was thoroughly investigated. The 16 bit OpenMSP core was customized to run programs entirely from internal block RAM, a cross compiling toolchain was set up on a Linux host PC, a HDMI peripheral core written to display a text console on a digital monitor and a JTAG debug core written to upload programs and dump memory. It was decided however that a client device implemented entirely in hardware (no processing core) would be a better solution, as overall it was less complex, would not bottleneck the Ethernet interface, and better satisfied the aim of moving all processing from the client to the server.

Another reason for not using a processing core was that TCP/IP and DHCP were decided to be unnecessary features for the solution. The solution was only ever intended to work over Ethernet LANs, not to be routed to different subnets or over the internet. Hence all communication between the client and server could be performed using only raw Ethernet frames and MAC addresses without the overhead of TCP/IP. Note that users can still access the internet as per usual even without the client devices being allocated individual IP addresses, as it is the virtualized desktop running on the server that will access the internet, not the client device. The client device is only the video display and input terminal for the virtualized desktop.

4.6 Alternative Ethernet Module Design

An alternative design of the Ethernet MAC and Ethernet decode section was considered with the frame decode logic integrated with the MAC module. The alternative design also used an intermediate module between the Ethernet decode module and the DDR2 controller. The alternative design is shown compared with the final chosen design in Figure 4.8:

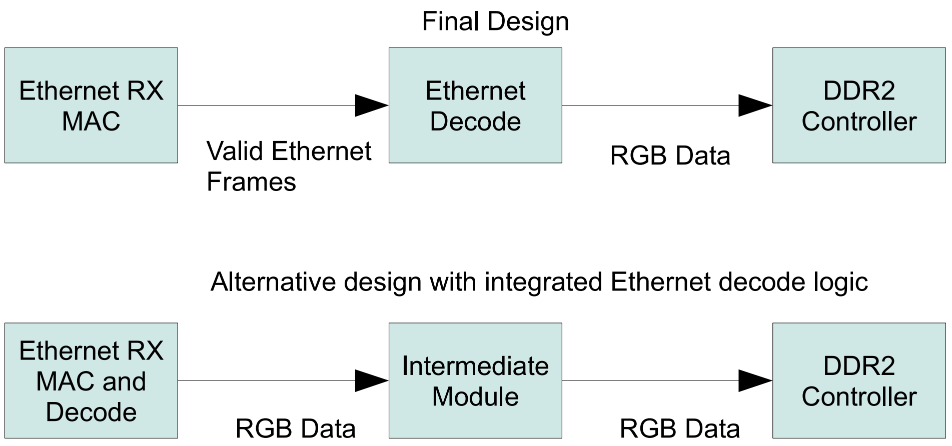


Figure 4.8: Alternative Ethernet section design

There were a number of reasons why the design with a separate MAC and decode module was chosen. The first was that by necessity the MAC module needed to use the relatively fast clock frequency of 125MHz. It was decided that it was better to have as little logic as possible running at this high frequency, and pass frame data to the decode module 64 bits at a time so that a lower frequency of 50MHz could be used for frame decoding. Decoding the Ethernet frames was also found to be easier processing 64 bits at a time compared to 8 bits at a time because, among other reasons, the 48 bit MAC address fields could be checked in one clock cycle rather than 6.

Another reason for having separate MAC and decode modules was that each had clearly defined functionality and could be developed and tested individually. This made simulating the modules for verification easier.

The main reason that no intermediate module was used between the decode module and the DDR2 controller was that the interface to the DDR2 controller turned out to be much simpler than initially anticipated. A separate module was used during early development because at first the DDR2 controller interface was unfamiliar and did not work as expected (see the Verification chapter for problems encountered). However, once the details of the interface were learned, integrating the functions of the intermediate module into the decode module turned out to be less complex than having separate modules would have been.

4.7 Ethernet MAC

4.7.1 Gigabit Media Independent Interface

Gigabit Media Independent Interface, or GMII, is a standard interface for the connection of a Gigabit Ethernet PHY to a host device or processor [11]. It uses 8 data signals, 1 clock signal, and two control signals in both the transmit and receive directions. These signals are shown in Figure 4.9. The signal timing is shown in Figure 4.10.

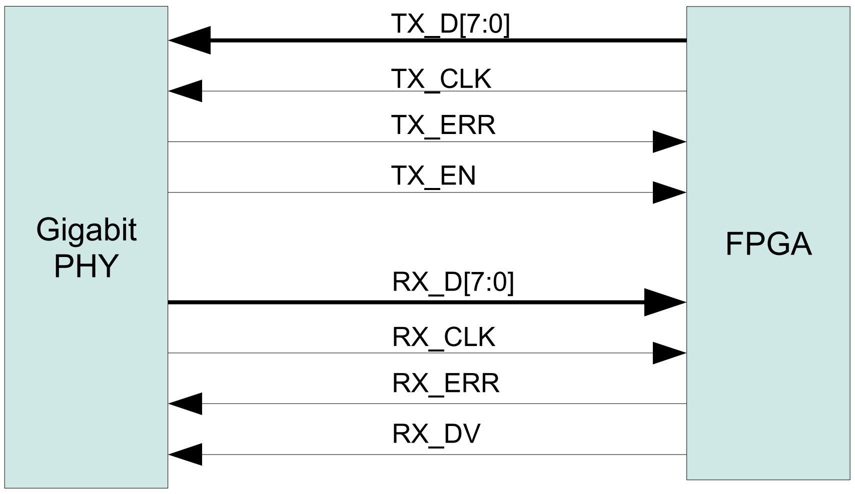


Figure 4.9: GMII signals

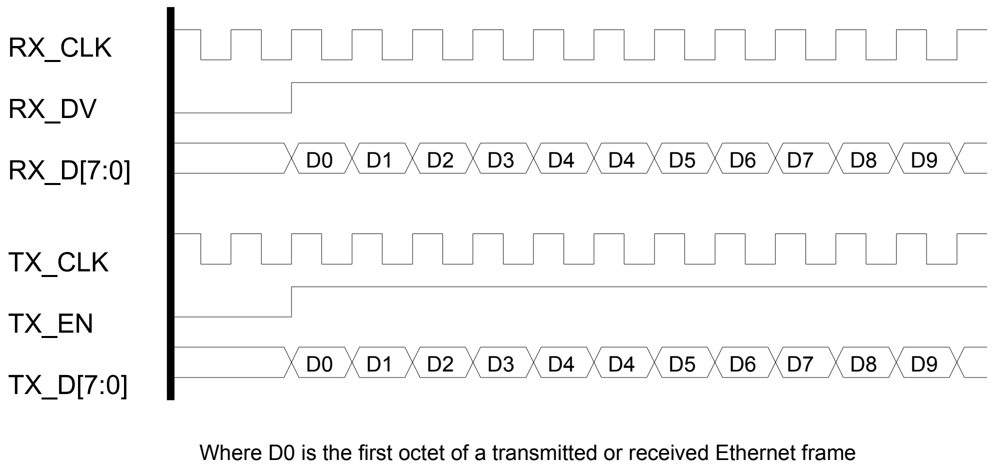

RX_CLK
RX_DV
RX_D[7:0]
D0 D1 D2 D3 D4 D4 D5 D6 D7 D8 D9
TX_CLK
TX_EN
TX_D[7:0]
D0 D1 D2 D3 D4 D4 D5 D6 D7 D8 D9
Where D0 is the first octet of a transmitted or received Ethernet frame

Figure 4.10: GMII signal timing diagram

Table 4.2 describes each of the GMII signals:

GMII Signal Name	**Direction**	**Description**
TX_D[7:0]	Output	8-bit transmit data port. Ethernet frame data is output one byte at a time on this port on the rising edge of TX_CLK when TX_DV is asserted.
TX_CLK	Output	125MHz transmit clock. Generated by the host device.
TX_EN	Output	Transmit Enable. Must be asserted to transmit frame data on TX_D.
TX_ERR	Output	Transmit Error. Can be asserted at any time during transmission to force a transmit error.
RX_D[7:0]	Input	8-bit receive data port. The host device receives Ethernet frame data one byte at a time on this port on the rising edge of RX_CLK.
RX_CLK	Input	125MHz receive clock. Generated by the PHY.
RX_DV	Input	Receive Data Valid. Data is valid on the RX_D port only when this signal is asserted.
RX_ERR	Input	Receive Error. If the PHY asserts this signal during frame reception it means a receive error has occurred and the current frame is invalid.

Table 4.2: Description of GMII signals

The Marvell 88E1111 PHY does not need any initial configuration (other than a 1ms reset pulse) before frames can be sent and received via GMII. The PHY automatically detects the Ethernet carrier and negotiates the maximum link speed when an Ethernet cable is connected. To receive an Ethernet frame the MAC module must simply monitor the RX_DV signal at rising clock edges. When RX_DV is asserted then there is valid data on RX_D[7:0]. The format of an Ethernet frame is described in Table 4.3.

Field Name	**Length in Bytes**	**Description**
Preamble	8	7 bytes of 0x55 for PHY clock synchronization, followed by a 1 byte 0x5D Start Of Frame indicator
MAC Destination Address	6	The MAC address of the intended recipient.
MAC Source Address	6	The MAC address of the sender.
Type/Length	2	If less than or equal to 0x0600 then this field indicates the length of the data field. If more than 0x0600 this field indicates the type of packet. For example 0x0800 indicates an IPv4 datagrams, and 0x0806 indicates an ARP packet.
Data	48 - 1462	The frame data. Ethernet packets must be at least 64 and not more than 1500 bytes in length (not including the IFG).
FCS	4	Frame Check Sequence. A CRC32 checksum of the MAC destination and source adresses, the type field, and the data. If a frame is received with an incorrect check sum it should be silently ignored.
IFG	>= 12	Inter Frame Gap. There must be at least 12 bytes of idle line state between frames. RX_DV is not asserted for the IFG.

Table 4.3: Ethernet Frame Data Structure

The GMII specification also includes a two wire management interface to access the configuration and status registers of the MAC. The registers provide information about the negotiated link speed, and may be used to configure the negotiation process. It was decided that it would not be necessary to implement this management interface as the default configuration of auto-negotiating the highest possible link speed did not need to be changed. Also the FPGA thin client was only intended to work at Gigabit link speeds, not at Fast Ethernet speeds. If only a Fast Ethernet link speed could be automatically negotiated, then it was not a requirement that the Thin Client function correctly.

4.7.2 Ethernet MAC RX Module

The Ethernet MAC RX (Receiver) module was required to:

- Read incoming Ethernet frames from the GMII port.
- Validate the received frames CRC32 checksum.
- Pass frames to the Ethernet Decode module, indicating whether the frame checksum was correct or not.

It was decided that the Ethernet subsystem would not implement Ethernet Flow Control. This is where a slow Ethernet device broadcasts a PAUSE frame to all other Ethernet devices on the network when it receives frames at too great a rate to sustain (without dropping frames). The PAUSE frame causes all Ethernet transmitters that implement Flow control to slow down their transmission rate. The receive logic of the FPGA Thin Client was designed to receive frames at greater than the full Gigabit Ethernet bandwidth, so it would not need Flow Control to slow down other transmitters. Also Flow Control for the transmit side of the Thin Client was not necessary as the amount of data transmitted (acknowledgment frames, keyboard data, mouse data) was very small relative to the Gigabit Ethernet bandwidth. Another reason for not including flow control was that the benefits it provided were far outweighed by the cost in time designing, implementing and verifying it.

The interface to the external PHY and to the Ethernet Decode module is shown for the MAC RX module in Figure 4.11.

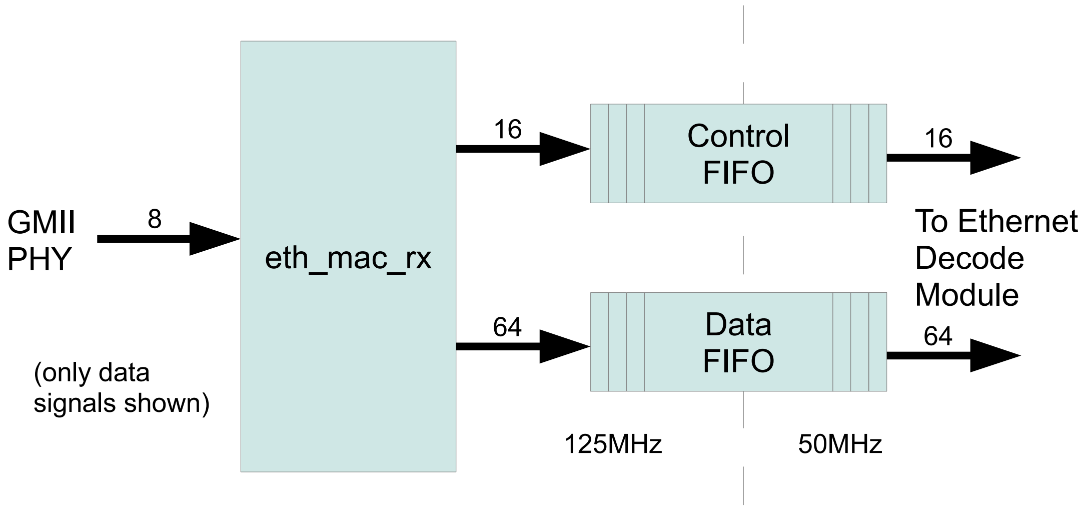


Figure 4.11: Ethernet MAC RX module interfaces

The design of the interface between the RX module and the Decode module is outlined in Section 4.8. A state machine was designed to read data from the PHY using GMII and to write the data to the control and data FIFO buffers. A simplified version of the state machine design is shown in Figure 4.12.

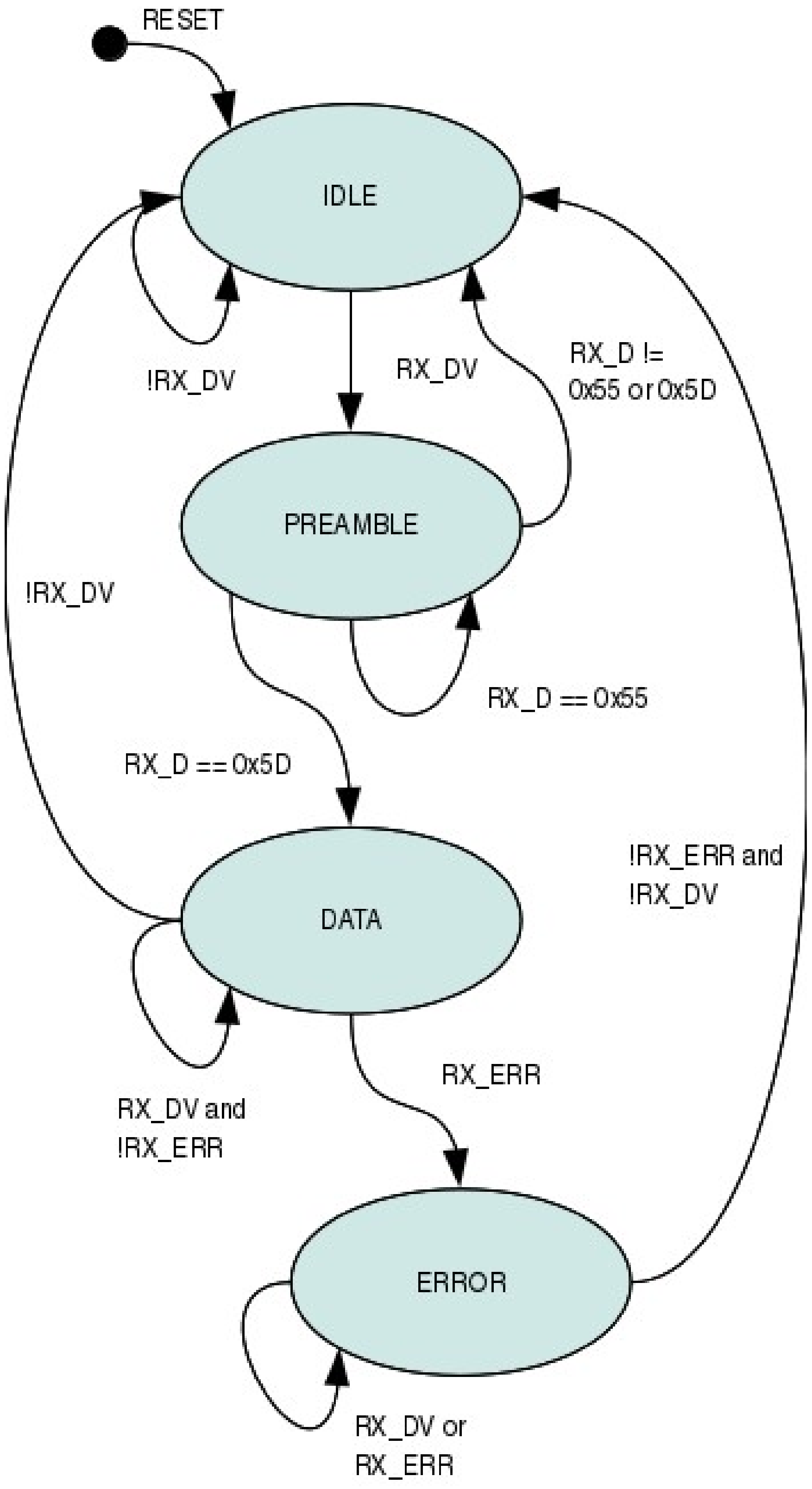


Figure 4.12: Ethernet MAC RX State Machine

The IDLE state simply waits for the RX_DV signal to be asserted by the PHY and 0x55 to appear on the RX_D[7:0] lines. When this happens the PREAMBLE state is entered. The PREAMBLE state enters the DATA state if RX_D == 0x5D, remains in the PREAMBLE state if RX_D == 0x55, or goes back to the IDLE state if anything but 0x55 or 0x5D is received.

The DATA state clocks in data while RX_DV is high and RX_ERR is not asserted. When RX_DV goes low the IDLE state is returned to. If RX_ERR goes high at any point the ERROR state is entered. The ERROR state loops until both RX_DV and RX_ERR are clear, at which point the IDLE state is entered.

Figure 4.12 does not contain all details present in the actual implementation. In particular the actual implementation needed to count the number of bytes received to make sure received frames were valid (more than 64 bytes and less than 1500 in length). Also not shown in the diagram is the Frame Check Sequence validation at the end of the DATA state.

To validate the Frame Check Sequence of received frames the CRC32 checksum of all bytes received after the preamble needed to be calculated. This was achieved using a simple CRC32 module with an 8-bit input data port, a CRC_OK output signal, and a 32-bit output containing the running checksum of the data input so far. If CRC_OK is asserted at the end of the DATA state (when RX_DV is deasserted) then the received frame is valid. If CRC_OK is low then the frame is invalid.

A feature not shown in the state diagram were two four bit counters that counted the number of good and bad frames received. The counters would be useful during testing to determine whether frames were being received and rejected, or not received at all. The counters were displayed in binary using the eight LEDs on the FPGA development board.

4.7.3 Ethernet MAC TX Module

The design of the Ethernet MAC TX module was similar to the design of the RX module. The main differences were:

- Instead of checking the FCS at the end of the frame data, the four bytes of the calculated CRC32 were written instead.
- A 12 byte Inter Frame Gap (IFG) needed to be output after each frame. More accurately, the IFG was not output, the TX_D, TX_DV and TX_ERR signals were simply cleared for 12 TX_CLK cycles.
- Instead of checking the ERR signal, it needed to be asserted if an error occurred reading the frame data from the encode module.

The interfaces of the MAC TX module are shown in Figure 4.13.

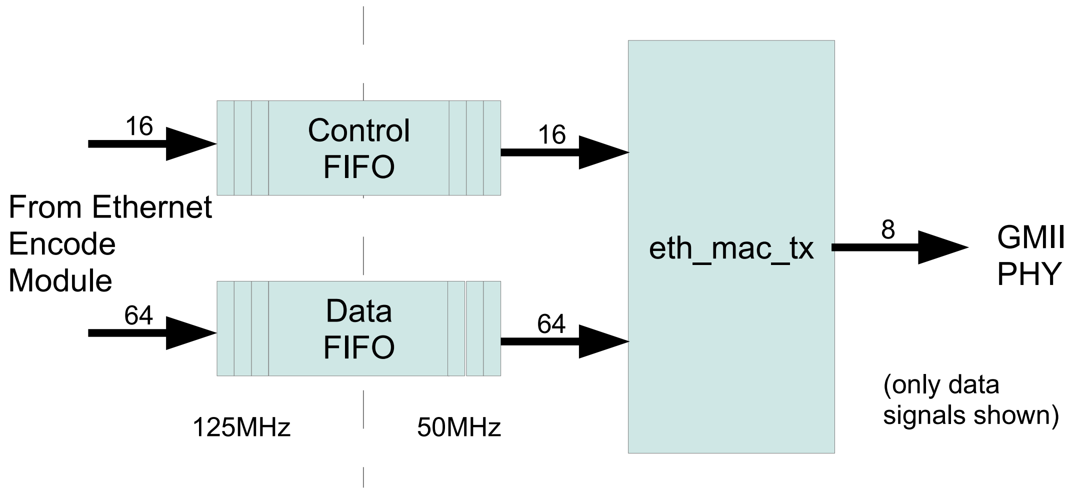


Figure 4.13: Ethernet MAC TX module interfaces

A state machine was designed to read frame data from the Ethernet Encode module and write it to the PHY using GMII. A diagram representing the state machine design is shown in Figure 4.14.

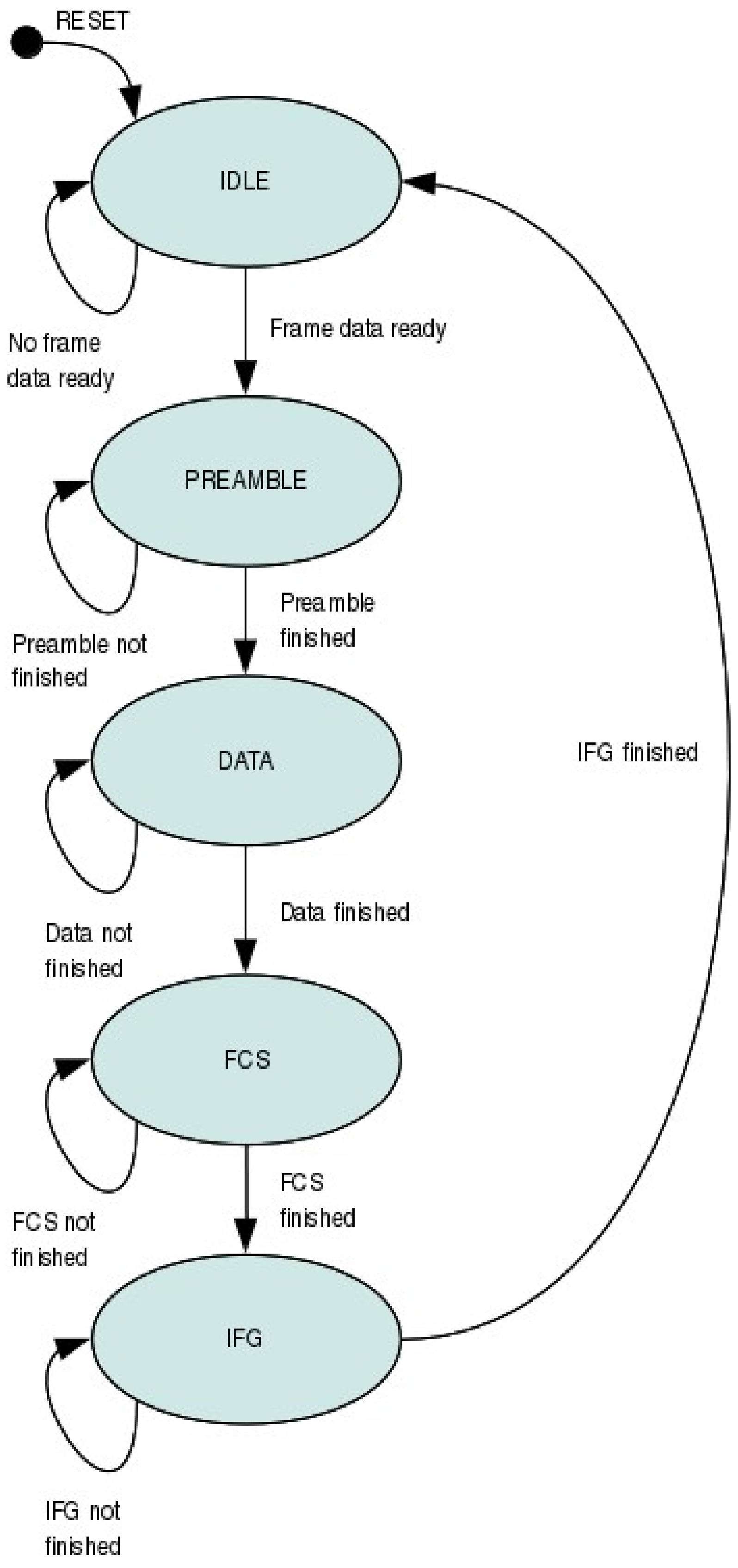


Figure 4.14: Ethernet MAC TX State Machine Design

The TX state machine design was less complex than the RX state machine as it did not need to perform as many frame validity checks (for example, checking the ERR and DV signals). The TX state machine waits in the IDLE state for frame data to be sent from the Ethernet Encode module. When data is received it moves through the PREAMBLE and DATA states, outputting data on the TX_D bus. All data bytes output are also written to a CRC32 module which calculates the FCS that is written in the FCS state. Finally the IFG state ensures that at least 12 bytes of idle line state are present between transmitted frames.

The MAC TX module also implements a frame counter for testing that counts the number of frames transmitted. The counter can be displayed using the LEDs on the development board. Whether to display the received frame counters or the transmitted frame counter was selectable using a switch on the development board.

4.8 Ethernet MAC and Decode Module Communication

4.8.1 Frame Buffering

One of the major problems encountered with the design of the Ethernet section was how to handle receiving frames when it was only known whether a received frame was valid after the entire frame had been received. There needed to be some way of buffering the frame before passing it to the Ethernet Decode module, so that the decode module only received valid frames. If received frames were not buffered the decode module would begin decoding frames before they could be checked for validity.

4.8.2 Control and Data FIFO Buffers

Several different approaches were considered for how best to handle the buffering. The first approach, and perhaps the simplest conceptually, was to use two FIFO buffers, a data FIFO and a control FIFO. The MAC module would write received frame data directly to the data FIFO. If the entire frame was received without errors, and the FCS was valid, then the number of bytes of data written to the data FIFO would be written to the control FIFO. If on the other hand an error occurred and the frame data written to the data FIFO was invalid, the MAC module would write the number of invalid bytes to the control FIFO with an error flag set. The error flag would be an unused upper bit of the control FIFO word.

On the other side of the FIFO buffers the decode module would wait for data to be present in the control FIFO. When control data is received, it indicates that a number of bytes are ready to be read from the data FIFO. The error flag indicates whether the frame data in the data FIFO is valid or not.

A diagram of this approach is shown in Figure 4.15. Note that the MAC lies in the 125MHz RX_CLK domain while the Decode module uses a 50MHz clock. Both FIFO buffers shown are asynchronous.

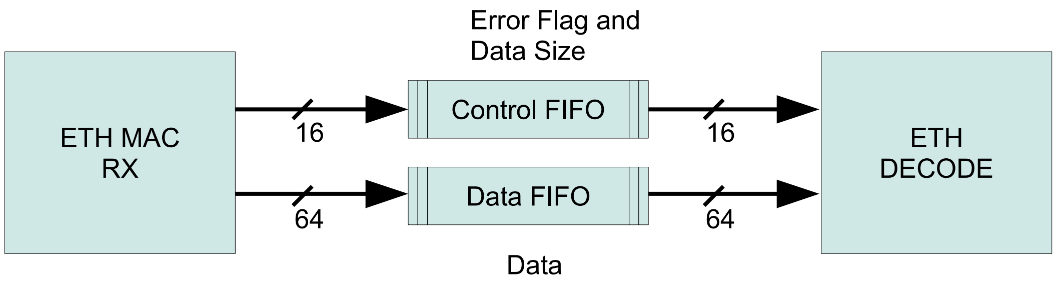


Figure 4.15: Ethernet module interface with Control and Data FIFO buffers

The advantage of this approach was the simplicity. Only standard FIFO buffers were used, and it was easy to implement in the MAC and decode state machines. One disadvantage is that if an unanticipated fault occurred that caused an incorrect size to be written to the control FIFO or an extra byte to the data FIFO, the FIFO buffers would be out of 'alignment' for all subsequent frames. The chance of a fault is not negligible given that the FIFO buffers cross clock domains. This introduces issues such as signal metastability caused by setup and hold violations.

An example of the problem is if an extra byte was accidentally written to the data FIFO, the decode module would still read the number of bytes indicated by the control FIFO. Then for the next frame received the decode module would read one byte of the previous frame along with the data from the current frame. This would probably cause the destination MAC address to be incorrect and the frame discarded. Thus all frames after the single extra byte was written would be discarded until a system reset was applied.

Another disadvantage is that if an invalid frame is received all the invalid data must be read from the FIFO one entry at a time by the decode module. There is no way to 'flush' data from the FIFO without reading it sequentially. This wastes clock cycles. This is not of particular concern however because as long as the decode module can read data at a faster rate than it can be received over Ethernet, there will be no loss of throughput.

4.8.3 Block RAM

Another approach considered in an attempt to fix the problems of using two FIFO buffers was to use a dual port block RAM to buffer received frames. A diagram of the approach is shown in Figure 4.16.

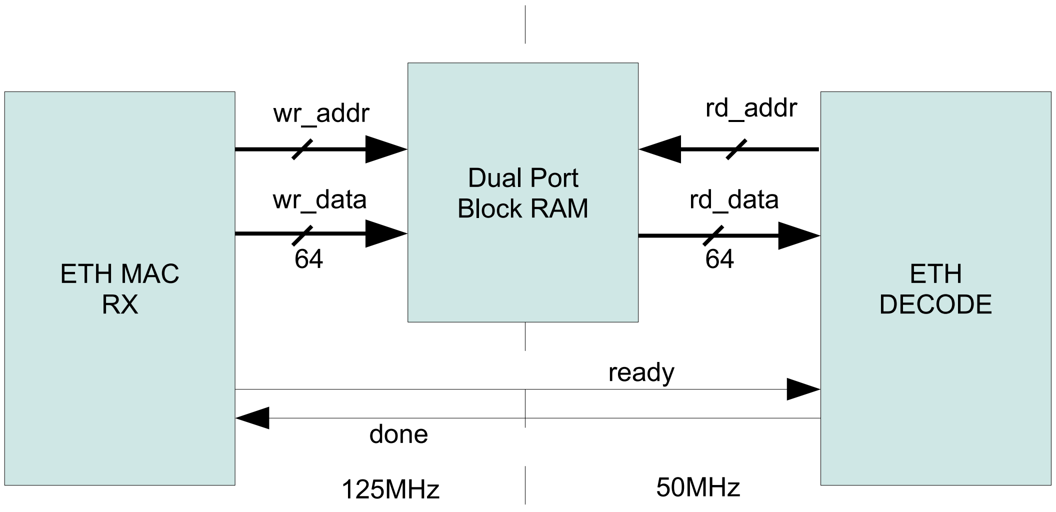


Figure 4.16: Block RAM Ethernet Module interface

The block RAM approach would work as follows:

1. The MAC RX module would write received Ethernet frame data to the block ram using the write port starting at address one.
2. After an entire frame was written the MAC module would write the length of data received to address zero.
3. After checking if the received frame was valid the MAC module would assert the 'ready' signal. If the frame was invalid then 'ready' would not be asserted, and the next frame received would overwrite the invalid frame data in block

RAM without the decode module needing to take any action.

4. The decode module after receiving this ready signal would start reading data from the block RAM.
5. After the decode module finished reading data it would assert the 'done' signal.
6. The MAC module would receive this 'done' signal, deassert 'ready' and wait for the next Ethernet frame to be received.

The advantage of this approach is that the decode module only has to ever read valid frame data, invalid frames are simply overwritten rather than having to be read out as with the dual FIFO approach. Also the block RAM approach does not suffer from the problem of the data and size becoming unaligned as with the FIFO approach. If an extra byte was written accidentally then the problem would only affect a single frame, no subsequent frames would be affected.

The main disadvantage is that the MAC module cannot write to the block RAM while the decode module is reading from it (as this would cause data conflicts where the MAC module would overwrite data that the decode module had not read yet). This problem could be solved by using more complicated control logic to make sure that only different sections of block RAM were being accessed at the same time, or by using multiple block RAM modules. This is the basis for the next approach.

4.8.4 FIFO with Checkpoints

The third approach considered was to extend on the idea of a dual port block RAM with multiple sections, each section being accessed by only one module at a time, to create an interface similar to that of a standard FIFO with the addition of 'checkpoints'. A standard FIFO uses a 'head' pointer to keep track of where the next write access will be written in RAM, and a 'tail' pointer to keep track of where the next read access will be read from. The 'full' and 'empty' flags of a standard FIFO are also generated from these pointers.

The difference between the checkpoint FIFO and a standard FIFO is that the checkpoint FIFO uses two 'head' pointers. One head pointer, called 'head', is used as per usual to point to the location where the next write will occur. The second head pointer, called 'head_chk', is used to generate the 'empty' signal. The head_chk pointer is usually some amount behind the head pointer, such that the FIFO will report that it is empty (when the tail pointer catches up to the head_chk pointer) even when there is still data in the FIFO (between the head_chk and head pointer). This is illustrated in the diagram below (Figure 4.17):

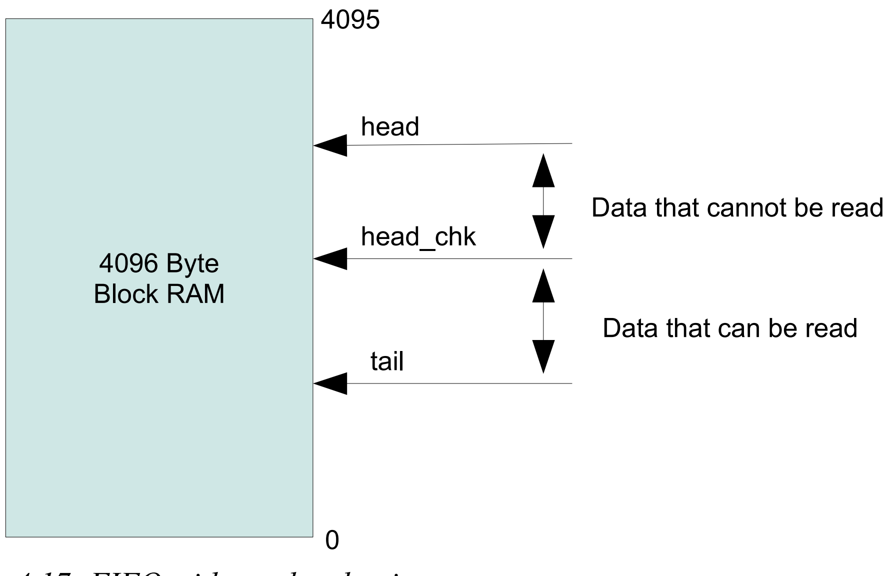


Figure 4.17: FIFO with two head pointers

Setting a checkpoint sets the head_chk pointer to the value of the head pointer. When a checkpoint is set all data up to that checkpoint can be read, but no data after. Another operation, called a 'reset', sets the value of the head pointer to that of the head_chk pointer. This effectively causes all data written since the last checkpoint was set to be discarded. The purpose of the checkpoint and reset operations can best be explained with an example of the MAC module writing a frame to the FIFO and the decode module reading it:

1. The head, head_chk and tail pointers start off after reset set to zero.
2. The MAC module writes the frame data as per usual to the FIFO. The FIFO reports it is empty as the head_chk pointer remains at zero, equal to the tail pointer.
3. At the end of the frame, if the frame is found to be valid, the MAC module sets a checkpoint. This causes the head_chk pointer to be set to the value of head. If the frame is invalid the MAC module 'resets' to the last checkpoint, effectively clearing the invalid frame from the FIFO.
4. If the frame was valid, tail will be behind the head_chk pointer, meaning that the FIFO will report it is not empty. The decode module can read data until the tail pointer reaches the head_chk pointer again.
5. At the same time as the decode module is reading data from the tail pointer, the MAC module can be writing data from the next frame at the head pointer. The decode module will not be able to read this data until the MAC module sets another checkpoint.

The checkpoint FIFO approach offered all the advantages the block RAM approach had over the dual FIFO approach, but without the drawback of not being able to write while the decode module was reading. In addition the checkpoint FIFO approach was more reusable, and was simpler to implement in the state machines than the block RAM approach.

4.8.5 Communication Method Selection

The checkpoint FIFO approach seemed to be the best option in terms of reusability, features provided, and simplicity of use. However, it was found that it was difficult to implement and had unanticipated problems when used between different clock domains. With more time to debug and fine tune the checkpoint FIFO would have been the best solution, however, the simpler dual FIFO approach was selected in the end because it worked where the checkpoint FIFO approach did not. It was found that the anticipated problem with the dual FIFO of the data and size becoming unaligned because of spurious writes did not occur in practice.

A FIFO width of 64 bits was chosen for the data FIFO and 16 bits for the control FIFO. The data FIFO width needed to be at least 32 bits so that it could support the full Gigabit Ethernet bandwidth at a 50MHz clock speed (need at least 1000Mbit/s, 16 bits * 50MHz = 800Mbit/s, 32 bits * 50MHz = 1600Mbit/s). 64 bits was chosen over 32 bits because it allowed the FIFO to be accessed only every second cycle and still support the full bandwidth. This extra cycle leeway greatly simplified the design when transferring data read from the RX MAC directly to the DDR2 controller FIFO.

4.9 Ethernet Decode Module

The Ethernet decode module was required to:

- Receive valid Ethernet frame data from the RX MAC module.
- Check the destination MAC address and the TYPE field.
- Decode received frames (frame format is defined below).
- Write RGB pixel data to the DDR2 Controller module.
- Write the received Source MAC address to a FIFO for use by the Ethernet encode module.

Only one frame format was defined. It contains RGB data and a frame buffer memory address to start writing the data at. The format is described in Table 4.4.

Field Name	**Size (Bytes)**	**Description**
DATA_LENGTH	2	The number of 64 bit data values to write to the frame buffer.
FLAGS	4	Flags are used to select the frame buffer start address (for double buffering), the bits per pixel (16 or 32 bits), and were used for testing and debugging.
ADDRESS	4	The 32 bit start address to start writing data at in the DDR2 frame buffer. Should be 64 bit aligned, that is, a multiple of 8.
DATA	8 * (DATA LENGTH)	The RGB data to write to the frame buffer. Should be a multiple of 8 bytes in length.

Table 4.4: RGB Frame Format

The TYPE field of the Ethernet frame should be set to 0x8B55. All other received frames with a TYPE not equal to 0x8B55 should be ignored, even if the destination MAC matches.

The interfaces of the Ethernet Decode module are shown in Figure 4.18.

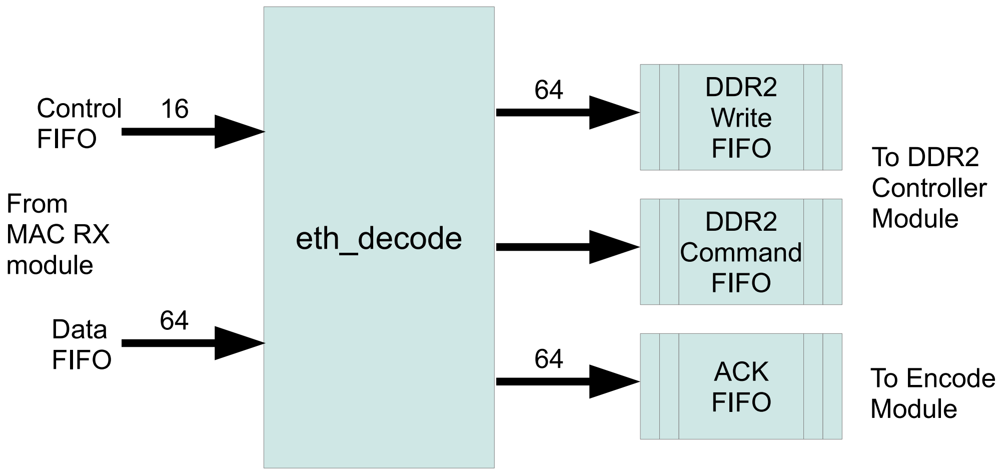


Figure 4.18: Ethernet Encode module interfaces

The communication interface between the MAC and the decode module was implemented with a 64 bit wide data FIFO. The DDR2 controller was also implemented with a 64 bit wide write port. It was decided that the data length should also be given in the number of 64 bit words rather than in the number of bytes. This was so that the decode state machine could simply count the number of 64 bit words written to the DDR2 controller to decide when the last data had been written. Requiring that the data be a multiple of 8 bytes simplified the design because the decode logic did not have to handle using a write mask for writes of less than 8 bytes.

To implement the requirements and decode the frame format, the state machine shown in Figure 4.19 was designed for the decode module.

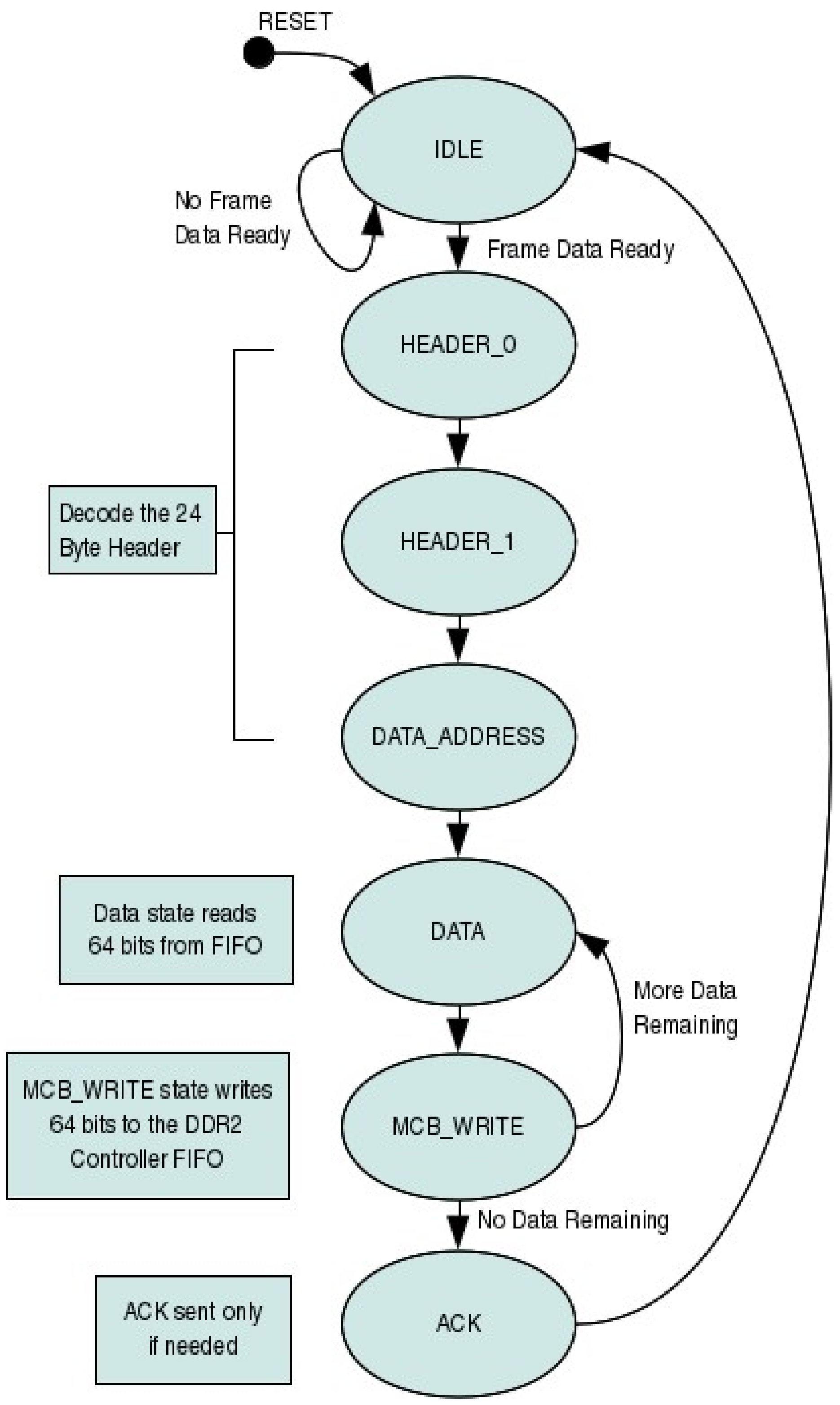


Figure 4.19: Ethernet Decode module state machine design

As can be seen in the state machine diagram the decode module state machine consisted of seven states. The IDLE state waits for the FIFO from the MAC module to become not empty before entering the HEADER_0 state. The HEADER_0 and HEADER_1 states each read a 64 bit word from the FIFO and check the destination MAC address and TYPE field, the source MAC address is stored in a register. The HEADER_1 state also reads and stores the DATA_LENGTH field. The DATA_ADDRESS state reads the FLAGS and DATA_ADDRESS fields.

After the DATA_ADDRESS state the decode module enters the DATA state. The DATA state waits for the data FIFO to become not empty, and then reads a 64 bit entry containing RGB pixel data to a register. Under normal operating conditions the FIFO should never actually be empty in this state as the entire frame should be present in the FIFO before the decode state machine even leaves the IDLE state. In the next state, the MCB_WRITE state, the data read in the DATA state is written to the DDR2 controller write FIFO, providing it is not full. The state machine oscillates between the DATA and the MCB_WRITE states until DATA_LENGTH writes have occurred.

If there are padding entries remaining in the FIFO after DATA_LENGTH writes, such as occurs when the frame is padded to reach the minimum frame length requirement, the remaining FIFO entries are still read in the DATA state but are not written in the MCB_WRITE state. After the entire frame has been read the state machine enters the ACK state, in which the source MAC address is written to the ACK FIFO to trigger the encode module to send an acknowledgment frame.

The reason that two separate states were used, one reading data from the FIFO and the other writing data to the DDR2 controller was to keep the design as simple as possible. The DATA state waits for the data FIFO to become not empty, and the

MCB_WRITE state waits for the DDR2 controller FIFO to become not full. The alternative was a more efficient, but more complicated, pipelined approach where data was both read and written every cycle. The main difficulty encountered in implementing this was how to handle the DDR2 FIFO becoming full. Logic would have been needed to both save the data previously read but not yet written, and to write this data before reading more data upon the DDR2 FIFO becoming not full. It was decided that the simpler approach of reading and writing on alternate cycles would save development time and not impact performance (if a sufficiently wide data path of 64 bits was used).

A design for test feature implemented was a test mode where the Decode module would write to the ACK FIFO every time a frame was decoded, rather than only for ping frames. This would cause an acknowledgment frame to be sent in reply to every frame received. This would be useful during testing to determine whether frames containing RGB data were actually being received.

4.10 Ethernet Encode Module

The main tasks the Ethernet Encode module was required to perform were:

- To generate acknowledgment frames given the MAC address of the source.
- To encode PS2 data into valid Ethernet frames.

The MAC address to send data to and the PS2 data to encode is received via two separate FIFO buffers, from the Encode module and the PS2 module respectively. Figure 4.20 shows the interfaces of the Ethernet Encode module:

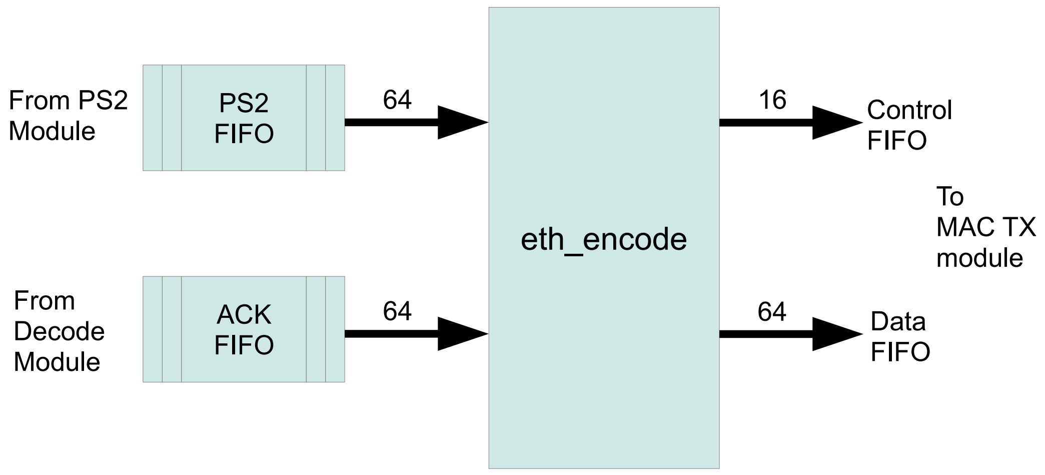


Figure 4.20: Ethernet Encode module interfaces

A Round Robin type system was used to schedule which FIFO to encode data from, as it was simple to implement and more 'fair' than reading each FIFO until it was empty before switching to the other.

Figure 4.21 represents the state machine designed for the Encode module.

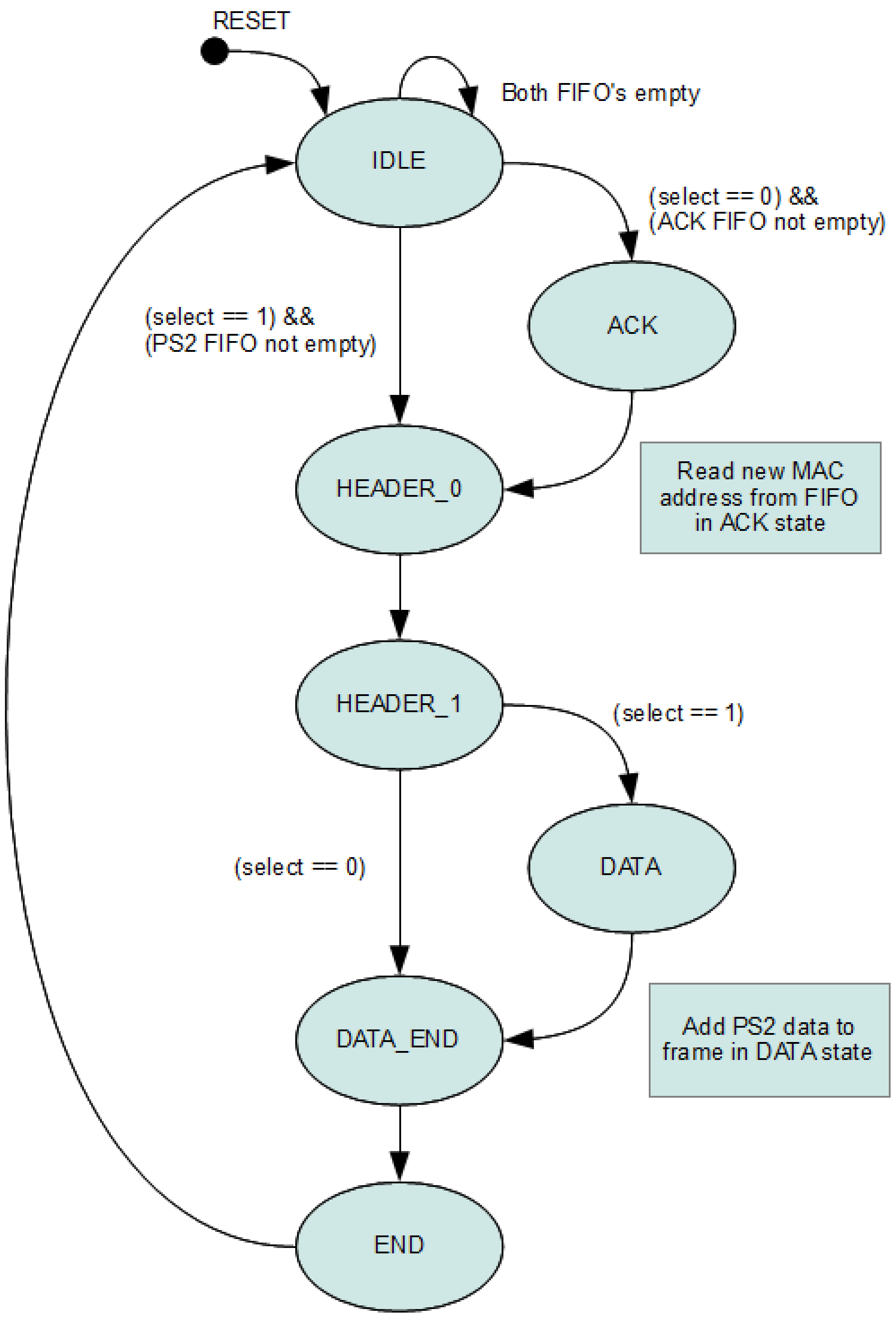


Figure 4.21: The Encode state machine design

The encode state machine consists of seven states. A single bit register called 'select' is used to decide whether to check the ACK FIFO or the PS2 FIFO for data in the IDLE state. The select bit is toggled if the selected FIFO is empty. If the ACK FIFO is selected and not empty the ACK state is entered. If the PS2 FIFO is selected and is not empty the HEADER_0 state is entered. In the ACK state the source MAC address of a valid Ethernet frame received by the Encode module is read from the ACK FIFO. This MAC address is then used as the destination MAC address for all subsequent frames sent until a different address is read from the FIFO. In the HEADER_0 and HEADER_1 states the destination MAC address, the source MAC address, and the Ethernet TYPE field are written to the TX data FIFO.

If the frame being encoded contains PS2 data ('select' is 1), the DATA state is entered. If the frame is an ACK frame ('select' is 0), the DATA state is skipped and the state machine goes directly to the DATA_END state. In the DATA state the 64 bit PS2 data entry is read from the PS2 FIFO and written to the data FIFO. In the DATA_END state padding entries are written to the data FIFO until the frame is greater than the minimum required length.

4.11 PS2 Interface Module

The PS2 module reads keyboard scan codes and mouse movement data from the external PS2 clock and data input pins. PS2 data is transmitted from the external device (keyboard or mouse) serially on the PS2 data line on the rising edge of the PS2 clock signal. The external device provides the clock signal which should be between 10KHz and 20KHz in frequency, but may stop when no data is being transmitted. Figure 4.22 shows the interfaces of the PS2 interface module.

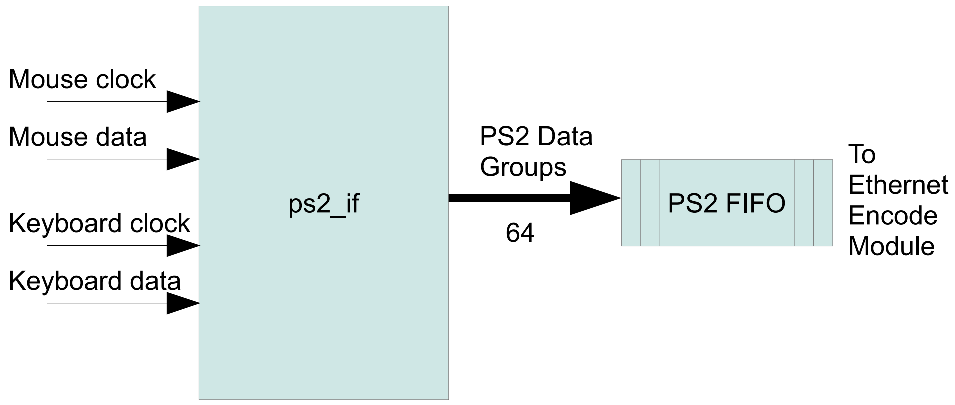


Figure 4.22: PS2 module interfaces

Data is transmitted in 11 bit frames consisting of one start bit, eight data bits, one odd parity bit, and one stop bit. Keyboard scan codes are transmitted in groups of one to four frames, with only the last data byte in each group having the MSB clear. The preceding data bytes will all have the MSB set. Mouse movement data is always transmitted in groups of three frames, the first frame indicating button status and movement direction, the second frame indicating magnitude of movement in the y axis, and the third frame indicating the magnitude of movement in the x direction.

The PS2 module did not need to decode the contents of the PS2 frames, it only needed to send the raw data bytes to the Ethernet Encode module so that the data could be sent to the server. The FIFO width was chosen to be 64 bits wide so that a whole keyboard data group of up to 32 bits could be transmitted at once, along with control signals to tell the server whether the PS2 data was read from the keyboard or the mouse interface. Another reason was that other FIFO buffers in the design had already been decided to use 64 bit wide ports, and it was simpler to re-use the 64-bit wide FIFO block than to create another FIFO with a different port width.

4.12 HDMI Display Subsystem

4.12.1 Overview

The HDMI display portion of the design was implemented using two modules, the HDMI Video module and the HDMI Controller module. The HDMI Video module generated the video control signals and read RGB pixel data from the DDR2 controller. The HDMI Controller module received the video control signals and RGB pixel data from the HDMI Video module and encoded it using the TMDS algorithm, serialized the encoded data, and output the serialized data on four differential pairs for transmission to the external display. The interfaces of the HDMI Video and HDMI Controller modules are shown in Figure 4.23. Both modules operate at the frequency of the Pixel Clock, which for 1280x720 resolution video at 60 frames per second was 74.25MHz.

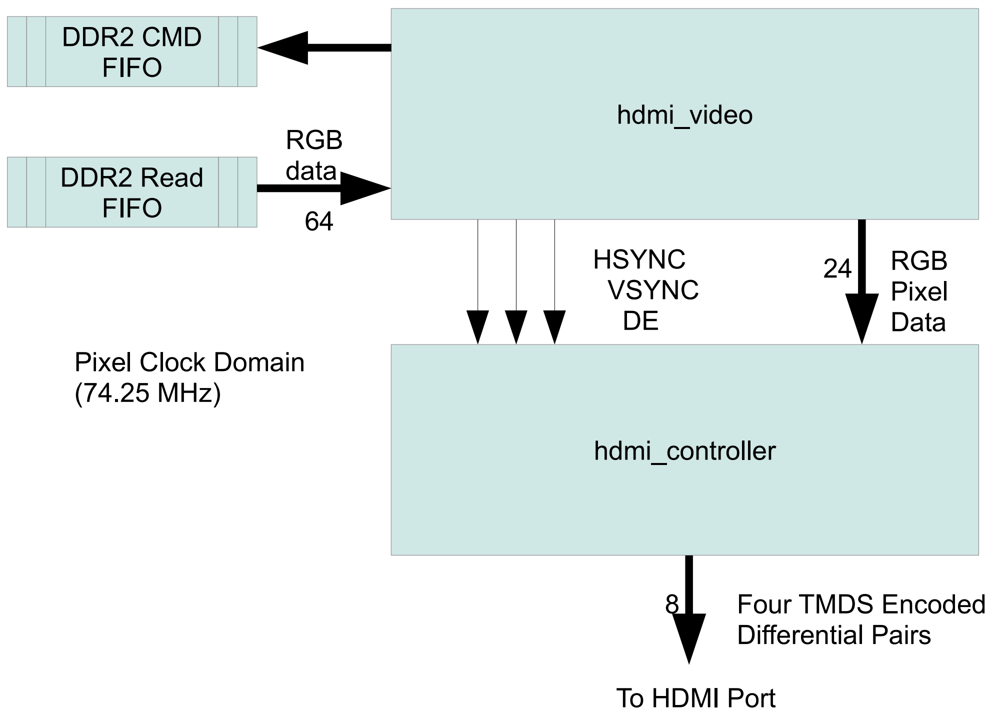


Figure 4.23: HDMI Video and HDMI Controller interfaces

4.12.2 HDMI Video Module

The HDMI Video module needed to generate the video control signals HSYNC, VSYNC and DE (Data Enable) given the timing parameters of a particular output resolution. Video mode timing parameters can be specified using a 'Modeline'. An example Modeline for a 1280x720 video mode is as follows:

```
 74.25 1280 1390 1430 1650 720 725 730 750 +HSync +VSync
```

These values can best be explained with a diagram. Figure 4.24 shows how the Modeline values relate to a graphical display.

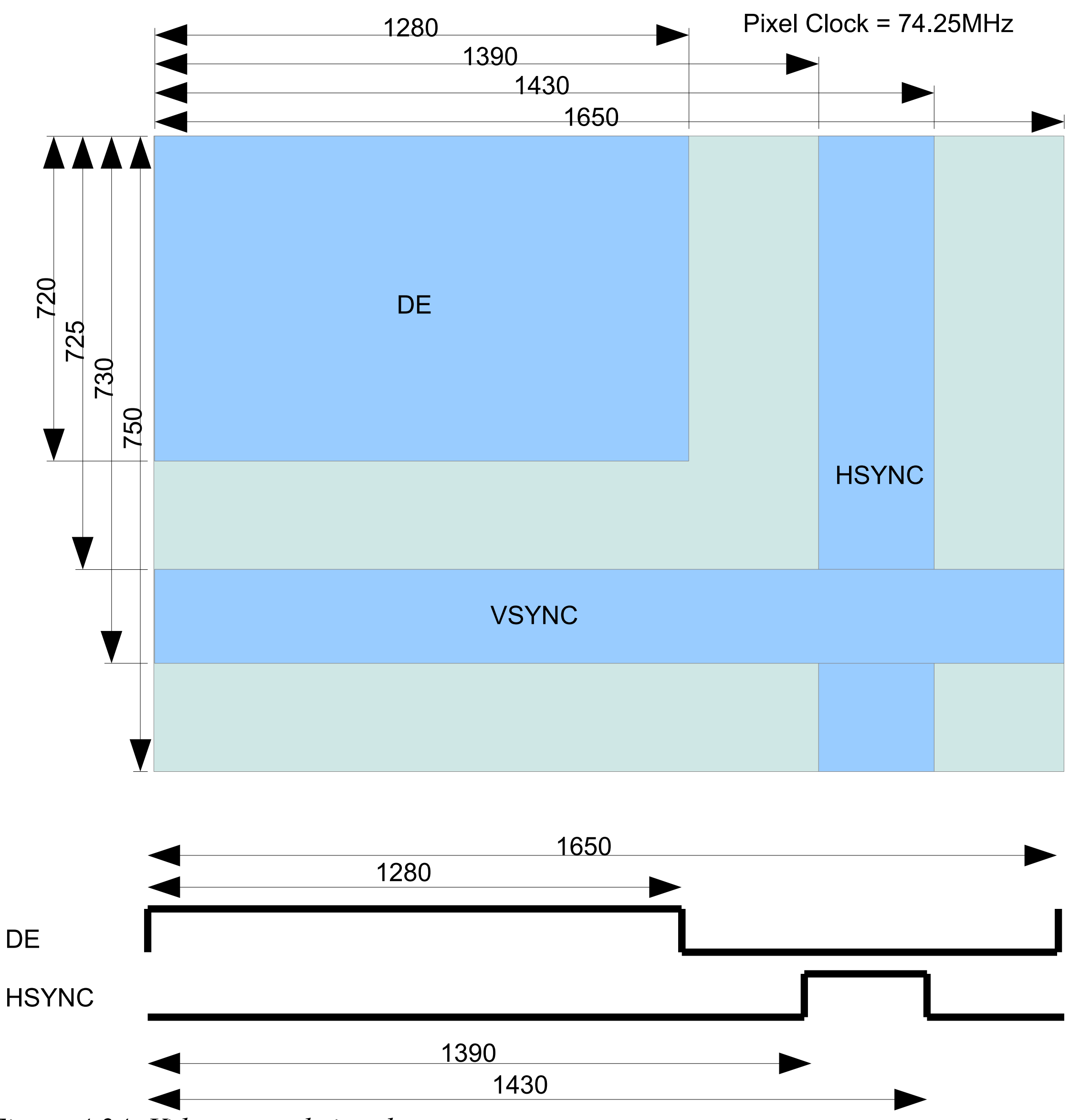


Figure 4.24: Video control signals

As can be seen the values of the Modeline specify the Pixel Clock frequency and the start and stop positions for the DE, HSYNC, and VSYNC signals for each horizontal scan line and for each frame. The HDMI Video module uses X and Y position counters to generate these signals. The X counter is incremented at the rate of the Pixel Clock, and the Y counter incremented once every time the X counter reaches the end of a horizontal scan line (1650 in this case). Logic is used to test whether each of the HSYNC, VSYNC or DE signals should be asserted based on the value of the X and Y counters.

The X and Y counters are also used to calculate which pixel should next be fetched from the framebuffer via the DDR2 controller. There were two main choices for mapping the framebuffer to memory locations. The first was to map each pixel such that:

```
  memory address = y * (width) + x * (bytes per pixel)
```

Where width is the horizontal resolution (1280 pixels). Using this scheme no memory space would be wasted as the start of each horizontal line immediately follows the end of another. The main problem is that an integer multiplication is needed to calculate each address. It was better to make the 'width' equal to a power of 2 so that only a logical shift was needed.

The 'width' value was chosen to be 2048 as this was the smallest power of two larger than 1280. No multiplication was needed for the 'bytes per pixel' value as this was either two or four (16 or 32 bits per pixel). The equation then became:

```
  memory address = (y << 11) | (x << (1 or 2))
```

Where '<<' is logical shift left and '|' is bitwise logical OR. By using a power of 2 number for the 'width' logic resources were saved (adders and multipliers), at the cost of some wasted memory space in DDR2 DRAM. It was decided that the inefficient memory use was worth the reduction in complexity and logic resource usage.

Reading the pixel data for each horizontal line from the DDR2 controller turned out to be more problematic than initially expected. The most efficient way to access DDR2 DRAM is in large bursts that read or write multiple consecutive memory locations. This worked well with the framebuffer memory map as horizontal lines were stored in consecutive memory locations. However, one of the problems with the DDR2 controller was that when a read was scheduled, there was an unpredictable and highly variable delay between the requested data being returned. The delay depended not only on the specific latencies of DDR2, but also on the amount of traffic on other ports (for example the Ethernet Decode module writing frame data).

The logical solution was to read the data for an entire line well before it was needed and buffer it in internal static RAM. The functional layout of this solution is shown in Figure 4.25.

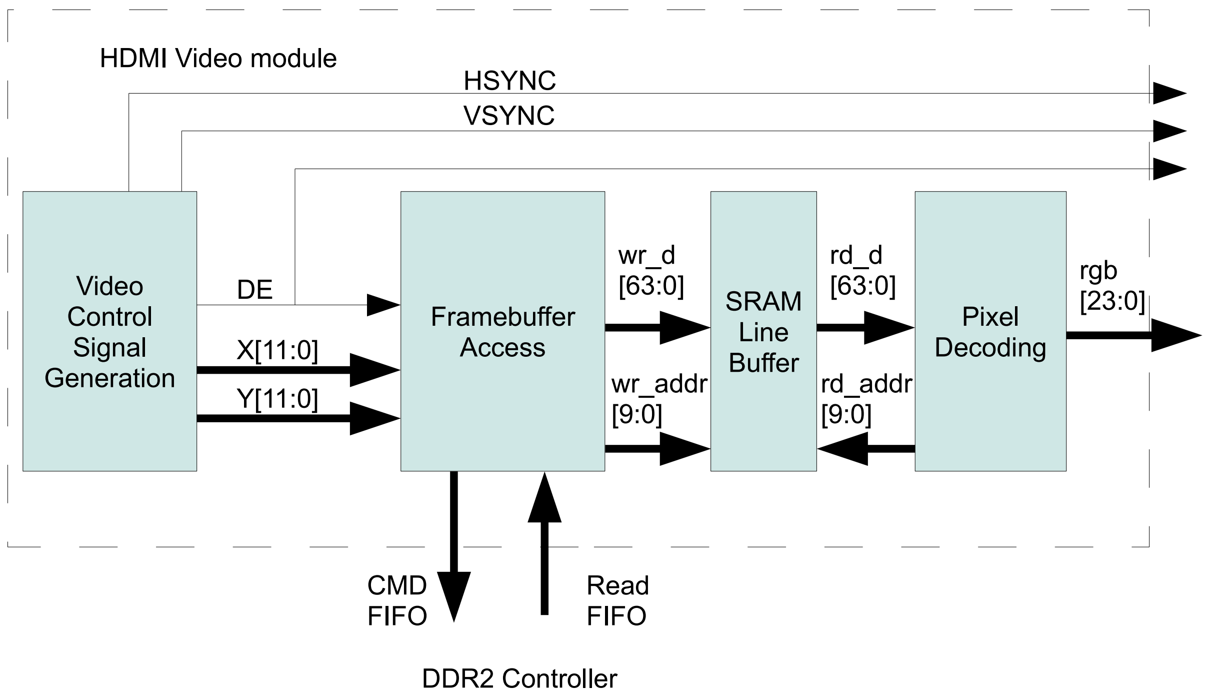


Figure 4.25: HDMI Video module functional layout

The start of the inactive period of each horizontal scan line was used to schedule the memory read to begin. This provided the DDR2 Controller the maximum number of cycles to complete the read operation before the data was actually needed. The read FIFO of the DDR2 controller could only hold 64 entries, so to read an entire horizontal line multiple chunks needed to be read. Each chunk was scheduled to be read a fixed number of clock cycles after the last, such that the read FIFO would not overflow and so that the last pixel in the line would be read before it was needed to be displayed. This number was determined by calculating the number of cycles each chunk would need to be displayed (depends on the chunk size and the bytes per pixel).

A test mode was also implemented that when activated (by an external switch) would ignore the RGB data read from the framebuffer and instead generate a test pattern based only on the X and Y counters. This would be useful during testing to determine whether a problem with the display was due to the HDMI Controller module not encoding the RGB signal correctly, or if the problem was due to incorrect RGB data being supplied.

4.12.3 HDMI Controller module

The main purpose of the HDMI controller need was to encode and serialize the video control signals and 24-bit RGB signal generated by the HDMI Video module for transmission to the external HDMI display. The encoding and serialization process can be described by the following steps:

1. The three 8 bit colour components of the 24 bit RGB signal are TMDS encoded separately. The DE signal is encoded with each colour component, while the HSYNC and VSYNC signals are only encoded with the Blue channel. The encoding is performed by three separate TMDS encoder modules.
2. The three TMDS modules output 10-bits each of TMDS encoded data per pixel clock cycle. The TMDS modules were designed and implemented according to the TMDS algorithm outlined in the DVI Specification [12]. The 10-bit outputs need to be converted to 5-bit outputs at double the pixel clock frequency. This is achieved by a 30-to-15 block that clocks out the top 5 bits of each channel, followed by the lower 5 bits of each channel, at twice the pixel clock rate.
3. The three 5 bit outputs at double the pixel clock rate are each serialized using hardware SERDES (SERialization DESerialization) blocks to form three serialized data streams at 10 times the pixel clock rate. A fourth channel is used to serialize the pixel clock.
4. The four serialized channels are output on four differential output channels using dedicated hardware blocks.

Figure 4.26 is a graphical depiction of the encoding process.

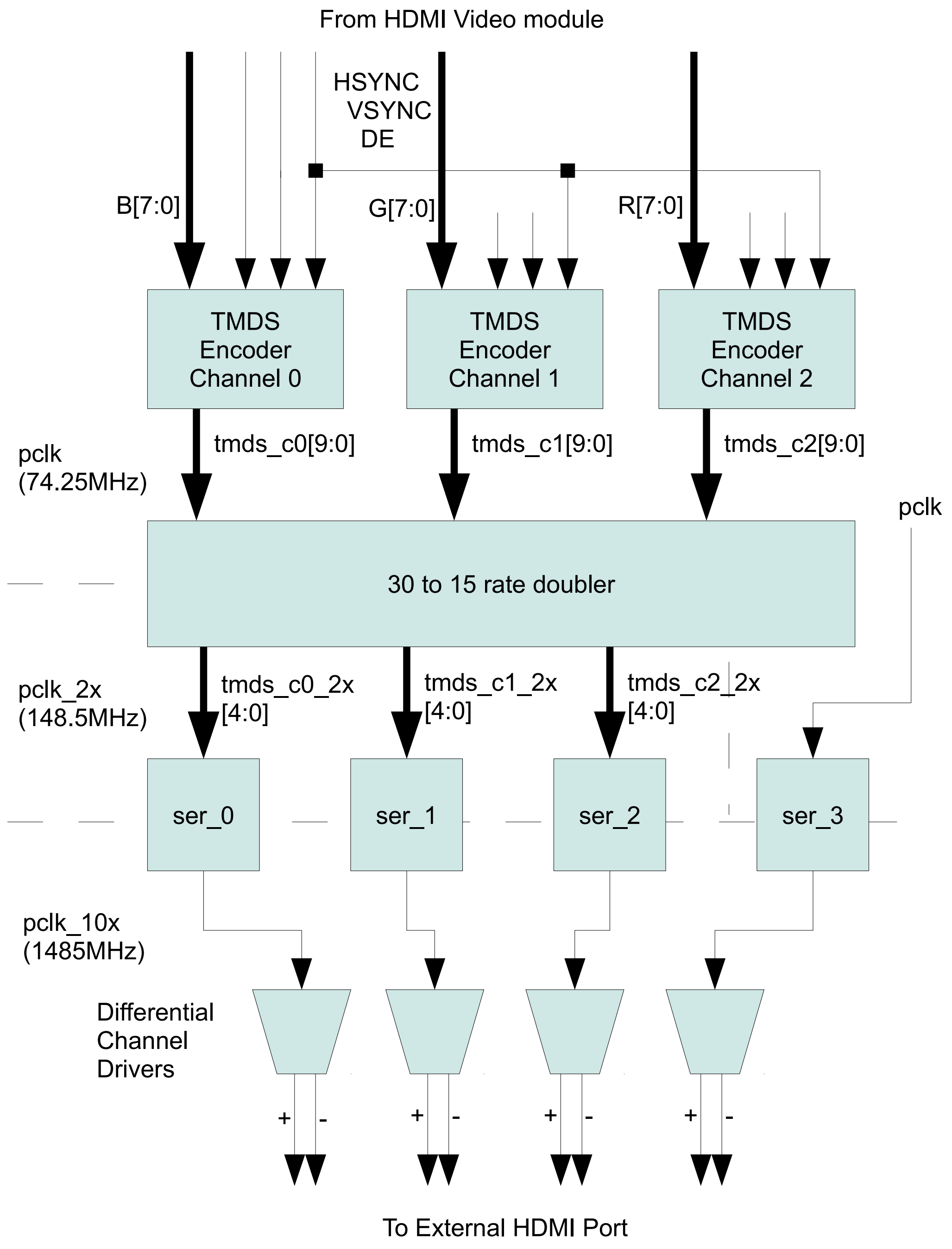


Figure 4.26: HDMI Controller encoding process

The TMDS encoder modules for each channel were designed according to the TMDS encoding algorithm outlined in the DVI specification (Digital Display Working Group, 1999). The TMDS algorithm encodes 8 input bits to 10 output bits. The algorithm attempts to encode the input data such that the number of zero and one bits output are equal. This is to reduce signal integrity issues caused by DC shift on the differential lines. The algorithm also reduces the number of zero to one and one to zero transitions that occur. This is in contrast to normal 8b to 10b encoding (used by Gigabit Ethernet, SATA, USB 3.0) which tries to increase the number of transitions to make clock recovery easier at the receiver. The TMDS algorithm reduces the number of transitions because clock recovery from the data stream is not needed, as the clock is transmitted in a separate differential channel.

The Spartan 6 series of FPGA has a high speed dedicated SERDES block attached to each Input Output Block (IOB). The hardware SERDES blocks were used as higher serialization rates could be achieved using the hardware SERDES blocks than performing the serialization in soft logic. Each SERDES block can serialize up to 4 bits of parallel data per clock cycle. The SERDES blocks can be cascaded with neighboring SERDES blocks to serialize more data per clock cycle, however, this means that the neighboring IOB will not be able to use the SERDES block. For this reason only two SERDES blocks were cascaded together rather than three which would have allowed full 10-bit serialization per cycle.

With two cascaded SERDES blocks up to 8 bits can be serialized per cycle, however, a factor of 30 needed to be chosen to allow for easy splitting of the data. Five bits was the only reasonable choice. This is the reason the 30 to 15 converter was needed, to split the three 10 bit channels to three 5 bit channels at twice the rate.

4.13 DDR2 Controller

The DDR2 Controller core was responsible for interfacing to the external DDR2 DRAM. The Xilinx Spartan 6 series of FPGAs have a built in Memory Controller Block (MCB) core which can be used to access external DDR/DDR2/DDR3 memory chips. To use the MCB in designs wrapper logic needed to be generated using the Coregen tool available with the Xilinx ISE Webpack. Coregen allowed the wrapper to be configured to have up to six independent ports between 32 and 128 bits in width (greater width per port means less total ports available however). For this design the wrapper was configured to have two 64-bit read-write ports. This was so that both the Ethernet subsystem and the HDMI display subsystem could each access the DDR2 DRAM using a separate 64 bit port. The Coregen generated wrapper was named the DDR2 Controller.

Each port of the DDR2 Controller had a FIFO like interface with a command FIFO, a read FIFO, and a write FIFO. To read from external DRAM the address, burst length and read instruction needed to be written to the command FIFO. Then the read FIFO 'empty' flag needed to be monitored until data was present. To perform a write operation the write FIFO needed to be filled with the data to write, then the write instruction, write mask, burst length, and address written to the command FIFO. The read and write FIFO buffers each had a 64 entry capacity so care needed to be taken to design the logic such that the read FIFO would not overflow and the write FIFO would not underflow.

Figure 4.27 shows the DDR2 Controller ports and how they connect to other modules. Note that the read FIFO was not connected on the Ethernet side as received RGB data was only ever written to the framebuffer, not read. Likewise the HDMI subsystem only needed to read RGB data from the framebuffer, not write it, so the write port on the HDMI side was left unconnected.

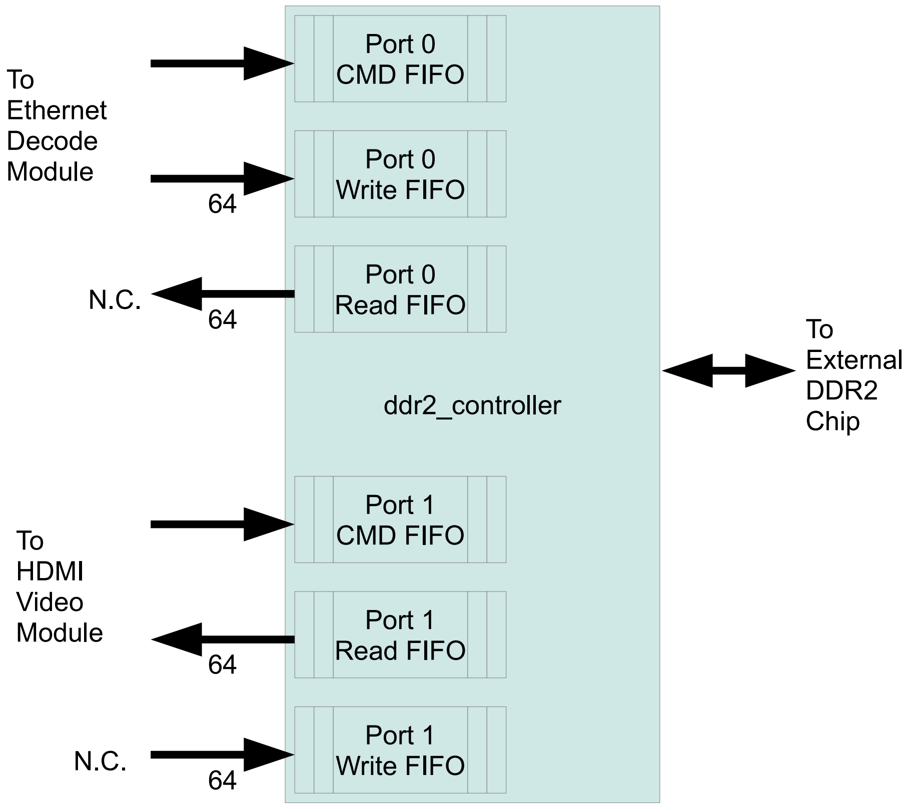


Figure 4.27: DDR2 Controller module

4.14 Clock and Reset Signals

Four main clock domains were used in the FPGA thin client design. These clock domains are listed below:

- The 50MHz main clock. Used by the Ethernet Decode, Ethernet Encode and PS/2 interface modules. This was generated by a PLL block in the DDR2 Controller module.
- The 125MHz Ethernet Receiver clock. This clock was received from the external PHY and needed to be buffered onto the FPGA global clock network.
- The 125MHz Ethernet Transmitter clock. Generated using a PLL block from the 50MHz main clock (multiply by 50, divide by 20).
- The 74.25MHz HDMI pixel clock. This was generated from the 50MHz main clock using a DCM_CLKGEN block. DCM_CLKGEN blocks allow a wider range of frequencies to be generated and support a simple serial control interface that allows the clock rate to be configured during normal operation.

Two other notable clock signals that needed to be generated were the twice and ten times pixel clocks for use by the HDMI Controller module serialization blocks. These clock signals were generated using a PLL block with the 74.25MHz pixel clock from the DCM_CLKGEN as input.

The global reset signal for the Thin Client design came from an external push button. This was directly connected to the DDR2 Controller module. The DDR2 Controller took some time to initialize the DDR2 chip after a reset, and the reset logic was designed so that all other modules stayed in their reset condition until the DDR2 chip had been initialized. This was to stop the modules accessing the DDR2 Controller before it was initialized. Reset signals for all modules needed to be buffered when originating from a different clock domain to prevent metastability issues and to prevent Synthesis warnings.

4.15 Host Side Software

So far only the design of the FPGA Thin Client device has been addressed. The Thin Client device was designed to use raw Ethernet frames for communication with the server. As no virtualization software could be found that was designed to use raw Ethernet frames for communication, an existing virtualization software solution needed to be modified to add support for this method. The chosen virtualization software to modify was QEMU [13]. QEMU is an open source machine virtualizer that performs well running under Linux using the KVM (Kernel Virtual Machine) framework.

By default QEMU will output the virtualized guest's display in a window on the host machine using SDL (Simple Direct media Layer). Table 4.5 summarizes some of the important functions for setting up a display, displaying video data, and receiving keyboard data:

Function name	**Description**
sdl_display_init	Called once when a new virtual machine is started. Is expected to set up a window or display so that the guest's virtual screen can be viewed.
sdl_update	Called every time the guest modifies its virtual screen. The rectangular region that changed since the last update is passed as an argument. The function is expected to update the modified region on the guest's virtual screen on the real screen or window.
sdl_refresh	Called at fixed intervals, about one hundred times per second. Is expected to collect keyboard or mouse events and send them to the virtual machine.
sdl_cleanup	Called when the virtual machine is shut down. Is expected to clean up any memory allocated by the sdl_display_init function and close any opened sockets or file handles.

Table 4.5: Description of QEMU SDL related functions

The functions were modified to perform the actions described in Table 4.6.

Function name	**Description**
sdl_display_init	Initialize raw Ethernet socket. Bind to specific interface. Allocate memory for guest virtual screen framebuffer.
sdl_update	Encode each horizontal line of updated rectangular region to raw Ethernet frames. Make sure that the frame size is within the maximum and minimum frame size allowance. Send the frames to the FPGA Thin Client.
sdl_refresh	Check if any raw Ethernet frames have been received from the Thin Client. If one or more has, decode the frames to get the PS/2 keyboard scan codes. Convert the PS/2 scan codes to XT scan codes (used by QEMU) and call the kbd_put_keycode function to send the scan code to the guest machine.
sdl_cleanup	Close raw Ethernet socket. Deinitialize guest frame buffer.

Table 4.6: Modified SDL related functions

5.0 Verification

5.1 Ethernet Subsystem

5.1.1 Frame Transmission Simulation

The first step of verifying the functionality of the Ethernet subsystem was to simulate the transmission of a frame. Frame transmission begins with the PS/2 module writing keyboard scan codes to the PS/2 FIFO. The FIFO is read by the Ethernet Encode module, which encodes the keyboard scan code into an Ethernet frame and transmits it to the MAC TX module via the TX data FIFO. The MAC TX module reads the frame data, appends the correct CRC32 FCS, and transmits it via GMII to the external Gigabit PHY.

There were two options for simulation, one was to write an individual test bench for each of the MAC TX, Encode, and PS/2 modules, synthesizing the expected inputs and checking that each modules outputs were correct. The other option was to write a test bench that synthesized inputs for the PS/2 module and verify that the outputs were correct. Then instead of writing a new test bench that synthesized input signals for the Encode module, the other choice was to simply attach the Encode module to the PS/2 module in the same way that it was attached in the full design, and run a simulation using the output of the verified PS/2 module as input to the unverified Encode module. The Unit Under Test (UUT) then becomes the combined PS/2 and Encode module.

This incremental simulation method is shown in Figure 5.1.

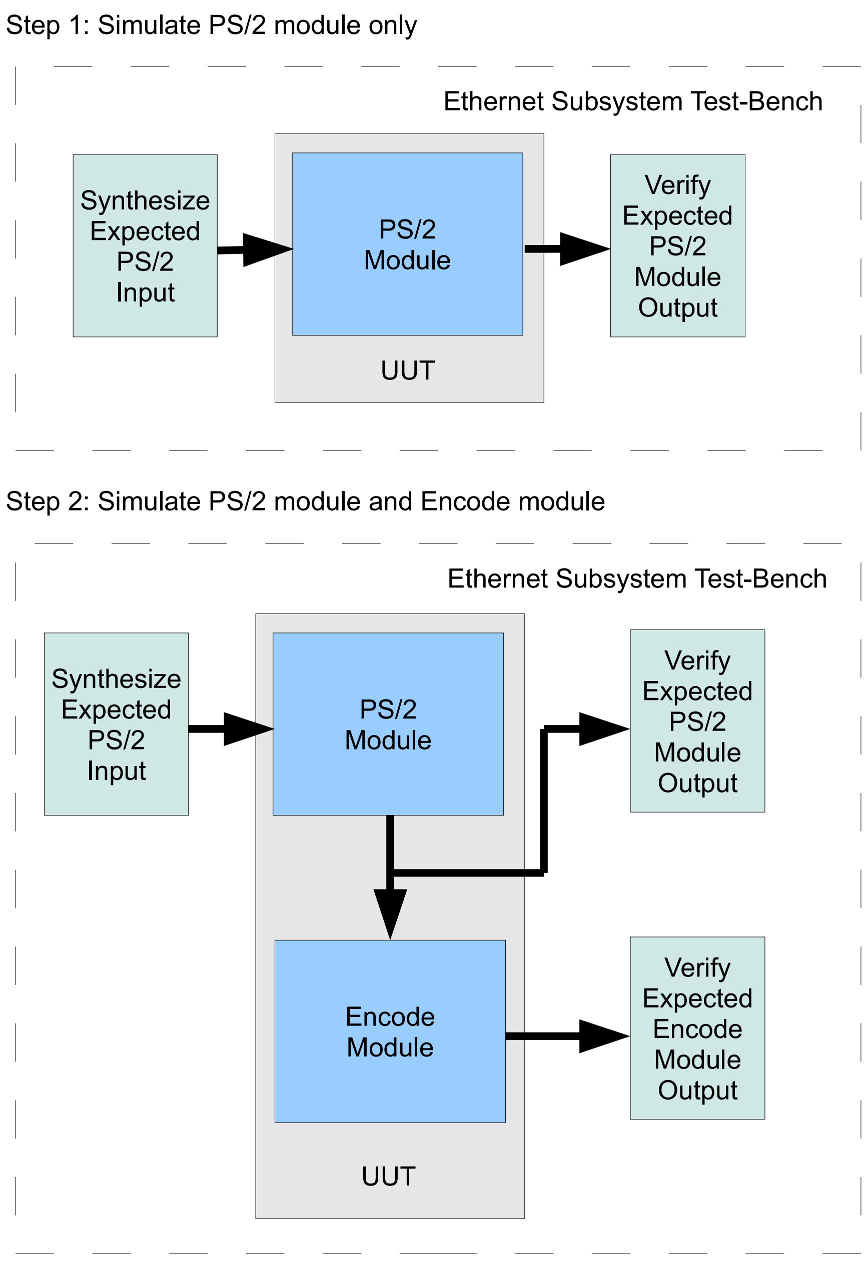

Step 1: Simulate PS/2 module only
Ethernet Subsystem Test-Bench
Synthesize Expected PS/2 Input
PS/2 Module
UUT
Verify Expected PS/2 Module Output
Step 2: Simulate PS/2 module and Encode module
Ethernet Subsystem Test-Bench
Synthesize Expected PS/2 Input
PS/2 Module
Verify Expected PS/2 Module Output
Encode Module
Verify Expected Encode Module Output
UUT

Figure 5.1: Incremental approach to simulation

The advantage of this approach to simulation was that only the external input signals (external to the FPGA) needed to be synthesized in the test bench, and not the intermodule communications. The disadvantage is that as more modules are added the simulation takes longer and longer to run. Another disadvantage is that using this approach it is difficult to verify how individual modules react to unexpected or invalid input signals. However, as the modules in the Thin Client design were relatively simple, the time taken to run simulations was small, and it was decided that as the design was only a prototype, testing module functionality with invalid input signals was not necessary.

The simulations of the Ethernet subsystem were performed using Xilinx ISIM, the simulator included in the Xilinx ISE Webpack. Other modules in the design were simulated using the open source Icarus Verilog simulator. However ISIM was used for the Ethernet subsystem as several FIFO buffers generated using Xilinx Coregen needed to be simulated. Icarus Verilog could not simulate these buffers directly as Coregen produced files in an incompatible format.

The simulations were run using the incremental method described. First the PS/2 module was simulated and its output verified. Then the Ethernet Encode module was added, and then the MAC TX module.

Expected results of frame transmission simulation:

- The PS/2 module should output a waveform that would write a 64-bit value representing the received keyboard scan code to the PS/2 FIFO.
- The Encode module should leave the IDLE state when the PS/2 module writes data to the PS/2 FIFO.
- The Encode module should output Ethernet Frame data as it passes through

the HEADER_0, HEADER_1, and DATA states. The data received from the PS/2 FIFO should be written in the DATA state.

- The Encode module should output padding data in the DATA_END state before returning to the IDLE state.
- The MAC TX module should only begin reading the frame data FIFO when the Encode module leaves the DATA_END state.
- The MAC TX module should read the correct amount of frame data from the data FIFO and output it on the GMII port in the correct order.
- The MAC TX module should append the correct CRC32 checksum to the frame.

It was discovered by analyzing the output waveform of the PS/2 module that the received scan codes were written in the reverse bit order to what was expected. This was fixed by reversing the direction of a shift register in the PS/2 module Verilog.

The CRC32 checksum appended to frames by the MAC TX module was found to be incorrect. It was discovered by analyzing the internal signals of the MAC TX module that this was caused by multiple problems. Firstly the last byte of frame data was not included in the checksum calculation. Secondly the frame data was passed to the CRC32 calculation module with the wrong bit order, and lastly the CRC32 checksum produced was appended to the frame in the wrong byte order.

Ethernet frames are transmitted one byte at a time starting with the leftmost byte. However the individual bytes are transmitted rightmost bit first (the least significant bit). The CRC32 checksum needs to be calculated in the same way, and appended such that the least significant bit of the checksum is the first bit of the checksum received. This is achieved by appending the least significant byte of the checksum first. In summary the solution was to reverse the bit order of the 8-bit input of the CRC32 module, reverse the bit order of the 32-bit output, and then reverse the byte order the 32 bit checksum was appended to the frame.

5.1.2 Frame Transmission Test on FPGA

After the problems discovered in simulation were fixed the design was synthesized and downloaded to the FPGA for a real world frame transmission test. The FPGA development board was connected via Ethernet cable to a Gigabit Ethernet Switch that was connected to a PC running Linux. The development board and Gigabit Switch are shown in Figure 5.2. A program called Wireshark [14] was used to capture and view any raw Ethernet frames the PC received.

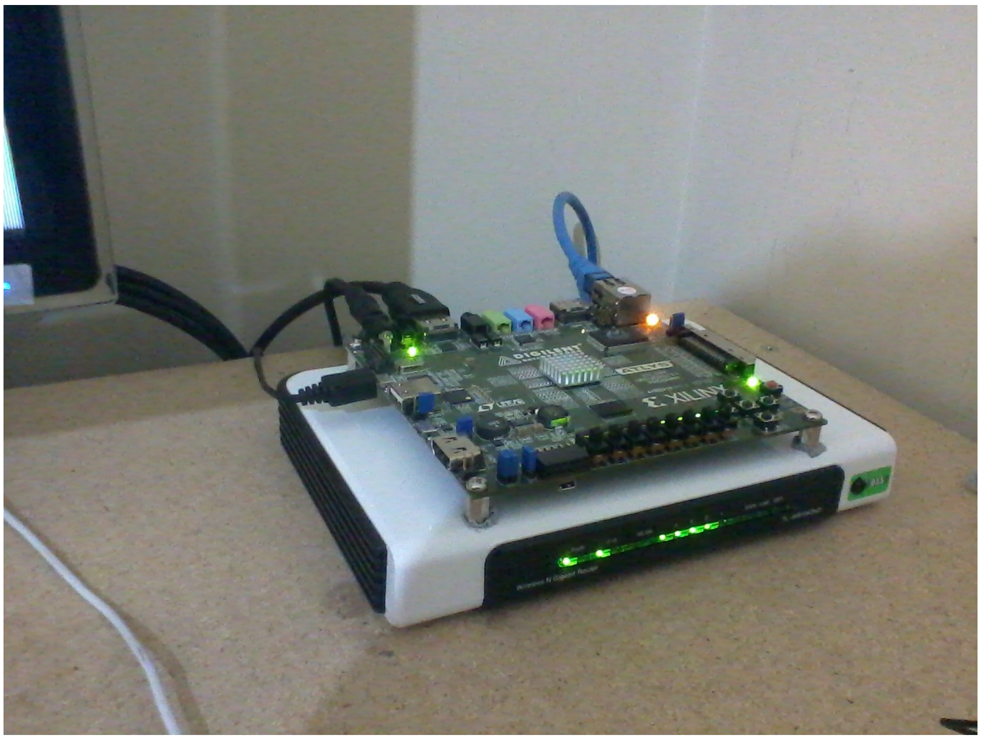


Figure 5.2: The Atlys development board connected to a Gigabit Switch

Expected results of real world frame transmission test:

- An Ethernet frame with the correct source address, destination address (FF:FF:FF:FF:FF:FF for broadcast), and type to be received upon pushing a key on the keyboard connected to the FPGA development board.
- The frame received should contain the PS/2 scan code representing the keyboard key that was pressed.

At first no frames were received at all when a keyboard key was pressed. The TX LED connected to the PHY showed activity, indicating that a frame was being transmitted onto the Ethernet link. However the frame was not captured by Wireshark at the PC. This pointed to a problem with the frame structure or contents, as invalid frames are dropped silently by Ethernet receivers. The most likely causes were a bad frame length, bad preamble, or bad checksum. An incorrect checksum could be caused by either a problem with the calculation or a problem with the frame data transmitted.

It was difficult to find the exact cause of this problem as there was no way to view the communication occurring between the FPGA and the PHY to analyze the data actually being transmitted. However it was believed that the problem was most likely with incorrect bit ordering of frame data. There was some uncertainty with the GMII port about whether the D0 pin or the D7 pin was the MSB. Also there was uncertainty about whether the frame preamble should be 0x55 or 0xAA. It was discovered after reversing each of these and testing that the problem was with the GMII port being connected with the wrong bit order. The D7 pin of the GMII port is the MSB, and the preamble should be seven bytes of 0x55 followed by one byte of 0xd5.

After the bit ordering problem was fixed valid frames could be captured at the host PC using Wireshark, however, the PS/2 scan codes received were incorrect. It seemed that some bits were being clocked in multiple times. The problem turned out to be with the PS/2 modules clock edge detection logic. Originally it would clock in a bit from the PS/2 data line every cycle that the PS/2 clock signal changed from zero to one. This approach did not take into account a noisy clock signal. The solution was to only clock in new data bits when the PS/2 clock was low for four cycles then high for four cycles. This is similar to the way that mechanical button presses are debounced when used as inputs to an FPGA or microcontroller.

Problems such as these show that functional correctness in simulation does not mean that the design will necessarily work in the real world. It is difficult to anticipate problems caused by connections to real external devices as opposed to simulated inputs.

5.1.3 Frame Reception Simulation

Frame reception was verified in a similar way to frame transmission. The modules were simulated starting with the MAC RX module, then adding the Ethernet Decode module. Simulating the input for the MAC RX module was a little more complex than simulating the input for the PS/2 module as the GMII signals for the reception of an entire valid frame needed to be created. This was achieved by storing a complete valid frame (along with preamble and correct FCS) in an array and writing the frame one byte per clock cycle to the MAC RX module.

Expected results were the Decode module writing the correct sequence of RGB data contained within the simulated frame to the DDR2 write FIFO and writing the Source MAC address from the frame to the ACK FIFO. No major problems were encountered in the simulation, and the output looked correct.

5.1.4 Frame Reception Test on FPGA

The next step was to synthesize the design and test it on the development board. The frame transmission logic had already been verified so the entire Ethernet subsystem was synthesized for the test. This included the TX and RX MAC modules, the Ethernet Encode and Decode modules, and the PS/2 module. To generate raw Ethernet frames for testing a small script in Python was written using the 'socket' library. The script generated raw frames such as would be generated by the Server in the final system.

Expected results of Ethernet reception test on real hardware:

- The received frame counter (displayed on development board LEDs) to increment once for each frame received.
- An acknowledgement frame should be received for each frame sent to the FPGA (using the test mode of the Ethernet Decode module).

Whether or not the RGB data sent in the frames was written to the DDR2 Controller could not be tested when the frame reception test was performed as the HDMI subsystem had not yet been completed.

The result of the first test was that only the first eight or so frames sent to the FPGA were acknowledged. After that no acknowledgement frames were sent and the receive counter would not increment. Frames were still transmitted when buttons on the keyboard were pressed however. This pointed to problems with either the MAC RX module, the Decode module, or the FIFO interface connecting the two. The problem turned out to be that when the DDR2 Controller write FIFO became full, frame data that had already been read from the RX data FIFO was not written when space became available again. Instead new data was read from the RX data FIFO and the unwritten data was lost. This had the overall effect of too much data being read from the RX data FIFO, such that part of the succeeding frame was read as the end of the first frame. This caused the decode logic to start decoding new frames from the wrong position and reject them because of an incorrect MAC address, causing acknowledgment frames not to be sent.

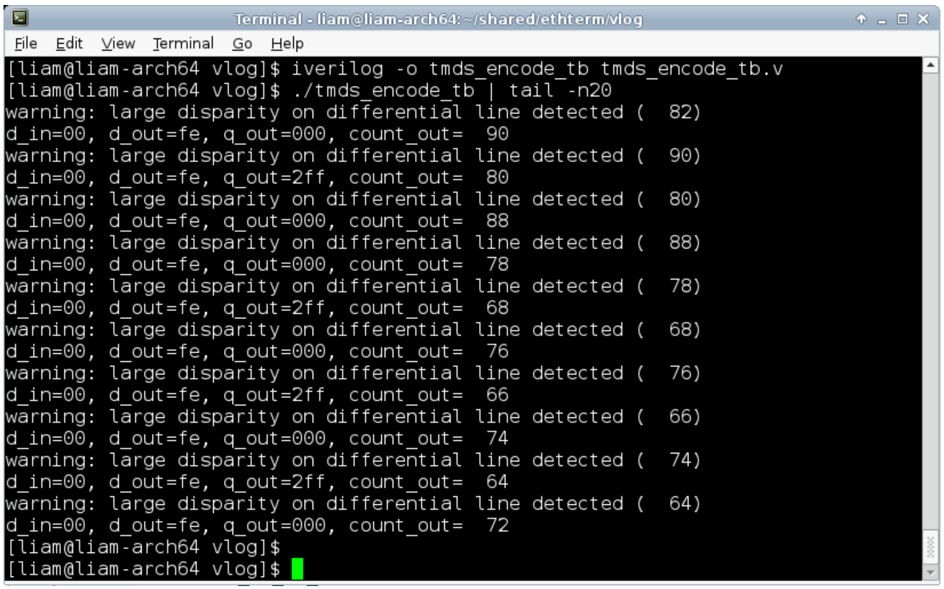
This was a difficult problem to find the cause of, because it did not appear in simulation (as the DDR2 Controller was not simulated), and because it only occurred under certain conditions (the write FIFO becoming full). The solution to the problem was to account for the data already read when the write FIFO becomes full. This was achieved by reading and writing data in different states within the Decode module state machine. To avoid such problems in the future special care should be taken to account for FIFO full and empty conditions.

5.2 HDMI Subsystem

5.2.1 TMDS Encoding Simulation

The first step for verifying the functionality of the HDMI subsystem was to simulate the TMDS encoders and verify that the output TMDS encoded data could be decoded to get the original data. To achieve this a simple TMDS decoder module was created. The TMDS decoder was simpler to write than the encoder as it was only meant to be simulated, so the design did not need to be optimized to reduce logic usage or increase performance. A Verilog Test Bench was then written that attached the TMDS Encoder and Decoder modules together, and generated a random sequence of 8 bit integers as input to the Encoder. The Test Bench displayed the input data to the encoder, the TMDS encoded data, the decoded output, and the disparity (count of how many more ones than zeroes had been transmitted). The test bench printed warning messages if a large disparity was detected (which would indicate problems with the encoder).

The simulation was run using the Open Source Verilog simulator Icarus Verilog. The expected results were that no disparity messages would be displayed and that the output data stream would match the input data stream (although would be delayed by a number of cycles). However the first time the simulation was run the output was as shown in Figure 5.3.


Terminal - liam@liam-arch64:~/shared/ethterm/vlog
File Edit View Terminal Go Help

```
[liam@liam-arch64 vlog]$ iverilog -o tmds_encode_tb tmds_encode_tb.v
[liam@liam-arch64 vlog]$ ./tmds_encode_tb | tail -n20
warning: large disparity on differential line detected (  82)
d_in=00, d_out=fe, q_out=000, count_out=  90
warning: large disparity on differential line detected (  90)
d_in=00, d_out=fe, q_out=2ff, count_out=  80
warning: large disparity on differential line detected (  80)
d_in=00, d_out=fe, q_out=000, count_out=  88
warning: large disparity on differential line detected (  88)
d_in=00, d_out=fe, q_out=000, count_out=  78
warning: large disparity on differential line detected (  78)
d_in=00, d_out=fe, q_out=2ff, count_out=  68
warning: large disparity on differential line detected (  68)
d_in=00, d_out=fe, q_out=000, count_out=  76
warning: large disparity on differential line detected (  76)
d_in=00, d_out=fe, q_out=2ff, count_out=  66
warning: large disparity on differential line detected (  66)
d_in=00, d_out=fe, q_out=000, count_out=  74
warning: large disparity on differential line detected (  74)
d_in=00, d_out=fe, q_out=2ff, count_out=  64
warning: large disparity on differential line detected (  64)
d_in=00, d_out=fe, q_out=000, count_out=  72
[liam@liam-arch64 vlog]$
[liam@liam-arch64 vlog]$
```

Figure 5.3: TMDS Encoder simulation output showing warnings

The VCD (Value Change Dump) waveform file was then viewed to discover what the problem was. An example of the waveform for the TMDS Encoder Test Bench is shown below in Figure 5.4.

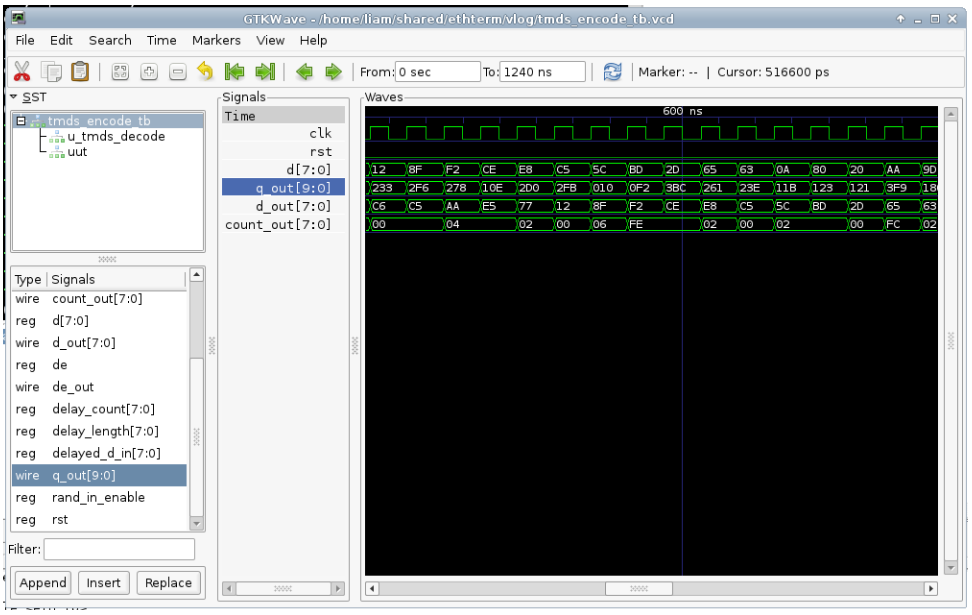
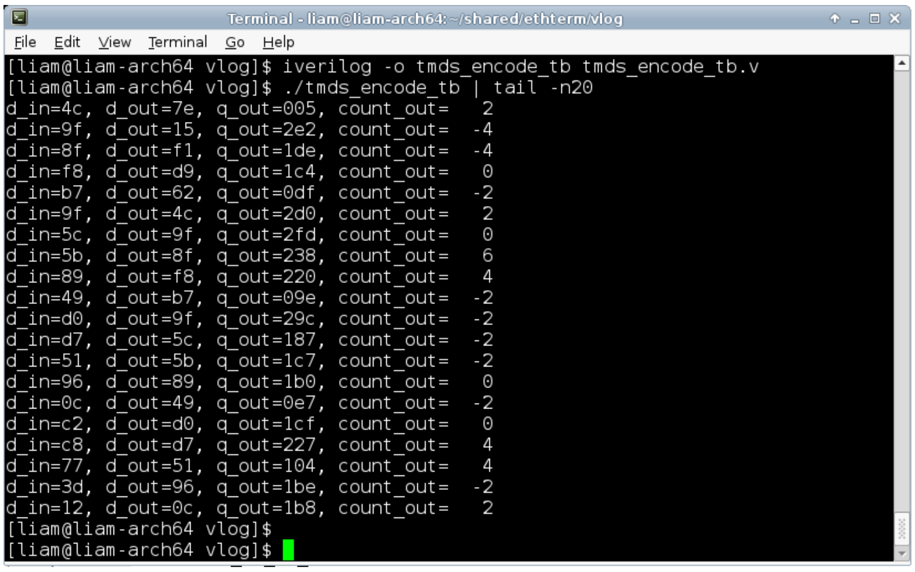


Figure 5.4: GTKWave screen capture of working TMDS Encoder

After viewing the waveform, it was discovered that one of the registers had an 'undefined' value. The cause was a typo using a register from the wrong pipeline stage (s1_c0 used instead of s2_c0). After fixing this error the output messages shown in Figure 5.5 were produced.


Terminal - liam@liam-arch64:~/shared/ethterm/vlog
File Edit View Terminal Go Help

```
[liam@liam-arch64 vlog]$ iverilog -o tmds_encode_tb tmds_encode_tb.v
[liam@liam-arch64 vlog]$ ./tmds_encode_tb | tail -n20
d_in=4c, d_out=7e, q_out=005, count_out=   2
d_in=9f, d_out=15, q_out=2e2, count_out=  -4
d_in=8f, d_out=f1, q_out=1de, count_out=  -4
d_in=f8, d_out=d9, q_out=1c4, count_out=   0
d_in=b7, d_out=62, q_out=0df, count_out=  -2
d_in=9f, d_out=4c, q_out=2d0, count_out=   2
d_in=5c, d_out=9f, q_out=2fd, count_out=   0
d_in=5b, d_out=8f, q_out=238, count_out=   6
d_in=89, d_out=f8, q_out=220, count_out=   4
d_in=49, d_out=b7, q_out=09e, count_out=  -2
d_in=d0, d_out=9f, q_out=29c, count_out=  -2
d_in=d7, d_out=5c, q_out=187, count_out=  -2
d_in=51, d_out=5b, q_out=1c7, count_out=  -2
d_in=96, d_out=89, q_out=1b0, count_out=   0
d_in=0c, d_out=49, q_out=0e7, count_out=  -2
d_in=c2, d_out=d0, q_out=1cf, count_out=   0
d_in=c8, d_out=d7, q_out=227, count_out=   4
d_in=77, d_out=51, q_out=104, count_out=   4
d_in=3d, d_out=96, q_out=1be, count_out=  -2
d_in=12, d_out=0c, q_out=1b8, count_out=   2
[liam@liam-arch64 vlog]$
[liam@liam-arch64 vlog]$
```
Figure 5.5: Simulation output from working TMDS encoder

As can be seen the disparity count stays low in magnitude and the decoded data 'd_out' matches the input data 'd_in' after a five cycle delay. The delay is mostly due to the pipelined design of the TMDS Encoder module.

5.2.2 Video Signal Generation

The HDMI video module was simulated individually with a Verilog Test Bench. The Test Bench simply generated an incrementing 64-bit integer on the read FIFO data port (where the HDMI Video module would normally read data from the DDR2 Controller module). The output waveforms were analyzed to verify that the correct VSYNC, HSYNC, DE and RGB signals were being generated. Also the commands and addresses written to the CMD FIFO were checked to see that the correct addresses were being accessed and the correct read instruction supplied. No major problems were discovered in simulation.

The next step was to test the HDMI subsystem on the development board. To eliminate any possible problems caused by interaction with the Ethernet subsystem, a simple module was written in the Ethernet subsystems place. The module generated an RGB test pattern and wrote it to the framebuffer via the DDR2 Controller. The design was synthesized, the development board connected to an HDMI monitor, and the bitstream downloaded to the FPGA.

The expected result was that the test pattern be displayed on the monitor. However this was not the case. The first step in finding the cause of the problem was to activate the 'test mode' of the HDMI Video module using a switch on the board. This bypasses the normal reading of the framebuffer from DDR2 and generates a test pattern using the video signals directly. No test pattern was displayed on the monitor however. This narrowed the problem down to the HDMI Controller module. As the

HDMI Controller module mostly consisted of hardware blocks for clock generation and signal serialization, not very much could be simulated, and without a high speed oscilloscope the HDMI output signals could not be checked physically. The only way of finding the problem was to search for mistakes in the Verilog design. The problem turned out to be that the system clock (50MHz) was accidentally used as the clock for the HDMI subsystem rather than the pixel clock (74.25MHz).

5.2.3 Framebuffer Access

After the problem with the incorrect HDMI subsystem clock was solved the test mode of the HDMI Video module displayed a perfect test pattern. However when normal mode was activated, where RGB data was read from the framebuffer via the DDR2 controller, the image on the display was heavily distorted. This distortion is shown in Figure 5.6.

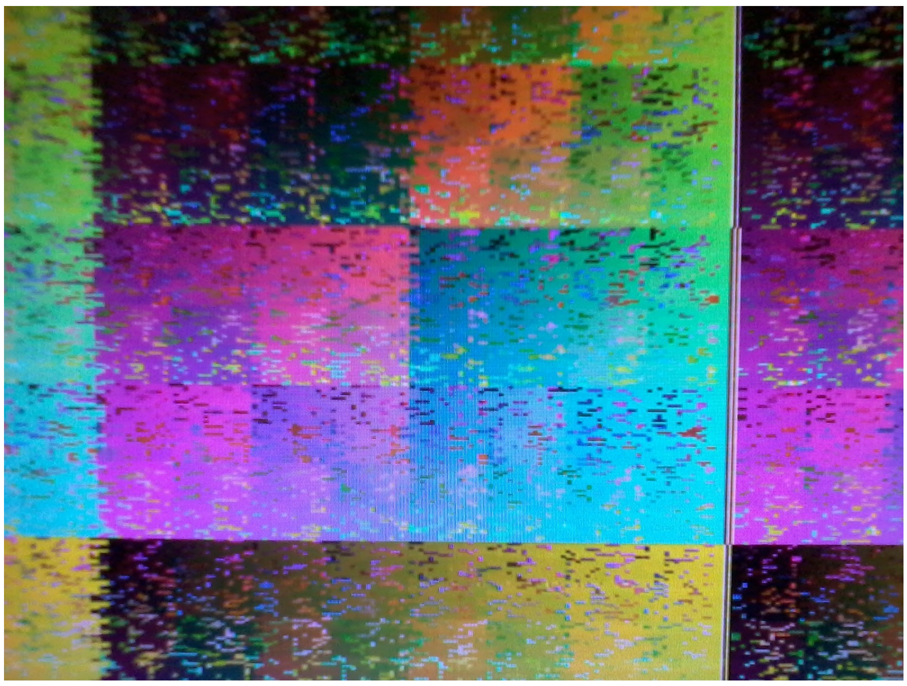


Figure 5.6: Distorted image during HDMI subsystem test

The cause of this problem was that the DDR2 controller read FIFO was overflowing intermittently. At first the HDMI Video module was designed to read data directly from the DDR2 read FIFO without using an intermediate line buffer. This problem demonstrated the need for the line buffer. After implementing the line buffer and scheduling framebuffer memory reads more effectively (only when needed rather than as fast as possible) the distortion disappeared. The test pattern read from the framebuffer was as clear as the test pattern generated in the test mode (as shown in Figure 5.7). This verified the operation of the HDMI subsystem.

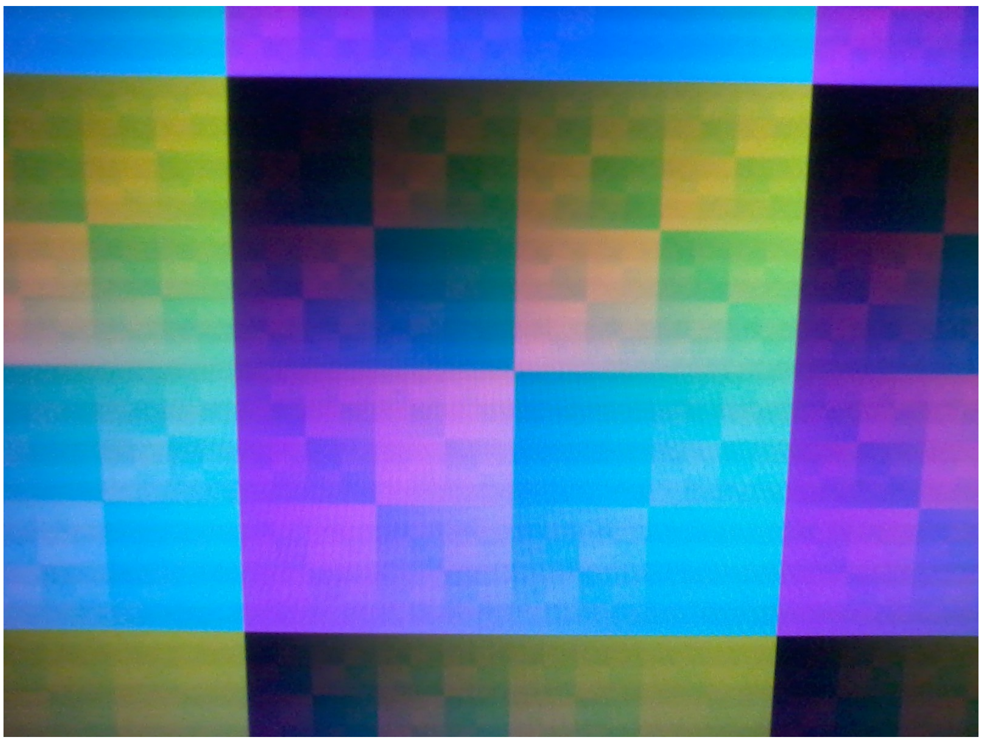


Figure 5.7: Test pattern without distortion

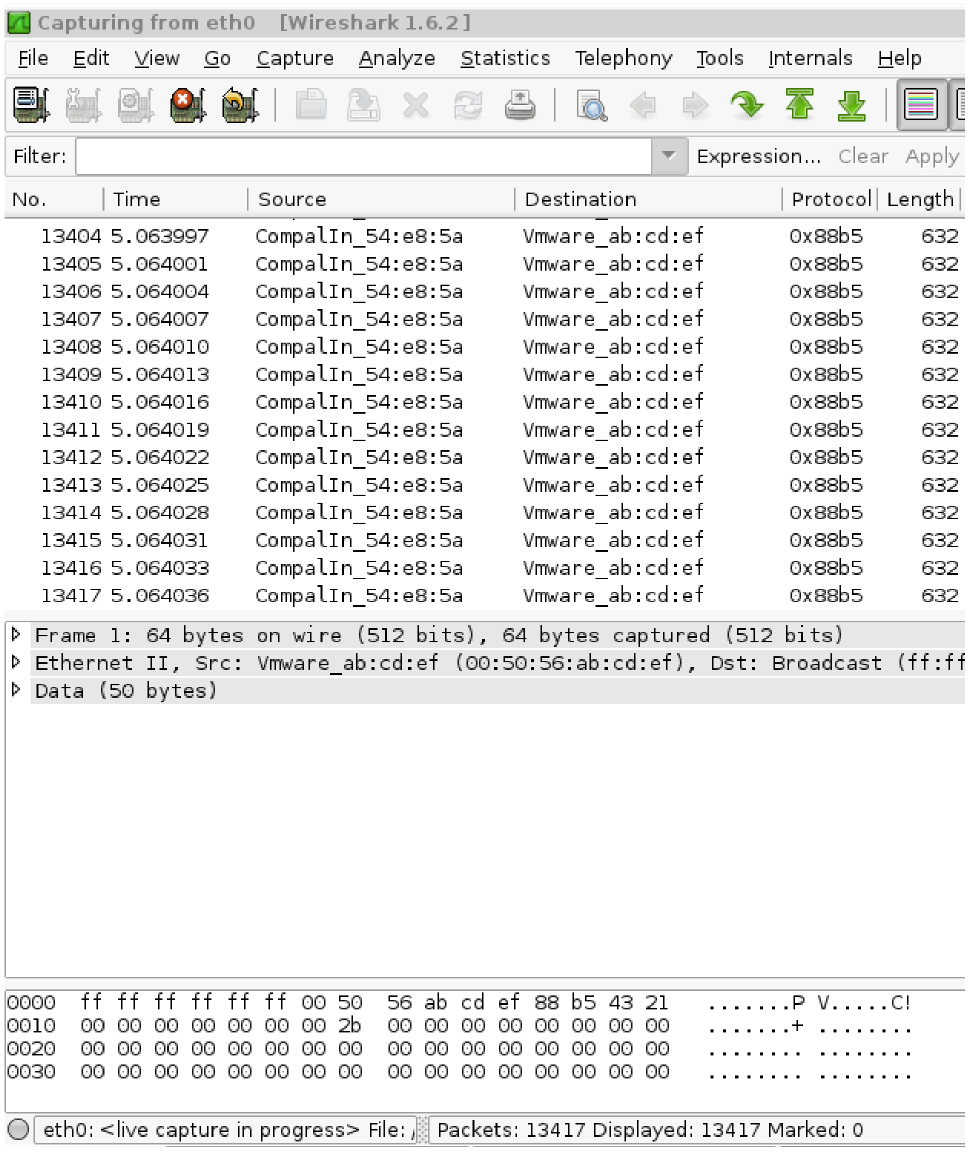
5.3 DDR2 Controller

As the DDR2 Controller was only really a wrapper around the dedicated hardware Memory Controller Block (MCB) of the Spartan 6 FPGA, no simulation was performed. The main problems anticipated were incorrect clock dividers and incorrect configuration of the MCB wrapper. The DDR2 controller was tested with a simple module that wrote an incrementing 8-bit value to address zero of memory, and attempted to read the value back. However, when this module was tested on the FPGA a different value was read than the one written. The value read did not change except when the entire development board was powered down and the restarted.

After carefully reading the data sheets for the Spartan 6 MCB it was found that the write mask was set incorrectly, such that the entire write was masked and not actually written to DRAM. It was initially believed that if a bit was set in the write mask it would keep the corresponding data byte from being written, however the opposite was true, the write mask acted as a logical AND so that only bytes with corresponding bits set would be written. Thus the solution was to set all the bits in the write mask, rather clearing them all. Following this change each value written to DRAM could be read back as expected.

5.4 Software

The functionality of the modified QEMU software was verified by analyzing the raw Ethernet frames transmitted using Wireshark. An example of a Wireshark frame capture is shown in Figure 5.8. It was expected that the transmitted frames were of the correct format and contained the correct RGB pixel data. A few minor problems were encountered, such as the starting memory address being incorrect and the frame size being smaller than desirable, but these were easily fixed.


Capturing from eth0 [Wireshark 1.6.2]
File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Filter:
Expression... Clear Apply

```
No.   Time     Source             Destination        Protocol Length
13404 5.063997 CompalIn_54:e8:5a  Vmware_ab:cd:ef    0x88b5   632
13405 5.064001 CompalIn_54:e8:5a  Vmware_ab:cd:ef    0x88b5   632
13406 5.064004 CompalIn_54:e8:5a  Vmware_ab:cd:ef    0x88b5   632
13407 5.064007 CompalIn_54:e8:5a  Vmware_ab:cd:ef    0x88b5   632
13408 5.064010 CompalIn_54:e8:5a  Vmware_ab:cd:ef    0x88b5   632
13409 5.064013 CompalIn_54:e8:5a  Vmware_ab:cd:ef    0x88b5   632
13410 5.064016 CompalIn_54:e8:5a  Vmware_ab:cd:ef    0x88b5   632
13411 5.064019 CompalIn_54:e8:5a  Vmware_ab:cd:ef    0x88b5   632
13412 5.064022 CompalIn_54:e8:5a  Vmware_ab:cd:ef    0x88b5   632
13413 5.064025 CompalIn_54:e8:5a  Vmware_ab:cd:ef    0x88b5   632
13414 5.064028 CompalIn_54:e8:5a  Vmware_ab:cd:ef    0x88b5   632
13415 5.064031 CompalIn_54:e8:5a  Vmware_ab:cd:ef    0x88b5   632
13416 5.064033 CompalIn_54:e8:5a  Vmware_ab:cd:ef    0x88b5   632
13417 5.064036 CompalIn_54:e8:5a  Vmware_ab:cd:ef    0x88b5   632

▷ Frame 1: 64 bytes on wire (512 bits), 64 bytes captured (512 bits)
▷ Ethernet II, Src: Vmware_ab:cd:ef (00:50:56:ab:cd:ef), Dst: Broadcast (ff:ff
▷ Data (50 bytes)

0000  ff ff ff ff ff ff 00 50  56 ab cd ef 88 b5 43 21   .......P V.....C!
0010  00 00 00 00 00 00 00 2b  00 00 00 00 00 00 00 00   .......+ ........
0020  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ........ ........
0030  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ........ ........

eth0: <live capture in progress> File: /   Packets: 13417 Displayed: 13417 Marked: 0
```

Figure 5.8: Example Wireshark Ethernet frame capture

A program (named simply 'raw') was written in C to read raw RGB data from a file (or piped via stdin), encode it to raw Ethernet frames, and transmit it as quickly as possible. The main reason for this was to test the maximum bandwidth attainable and to test if the FPGA Thin Client would work at full Gigabit Ethernet data rates. A secondary reason was to test the Thin Client with High Definition video data to make sure the display quality was acceptable. The program 'raw' was first tested without the FPGA Thin Client connected. This worked because 'raw' did not check for received frames, it only transmitted.

It was expected that 'raw' would transmit frames at near the full bandwidth of Gigabit Ethernet (125MB/s). However the rate achieved was only about 2MB/s. Wireshark was used to capture the frames transmitted and received to see what the problem could be. It transpired that a wireless router on the network was transmitting Ethernet Flow Control 'Pause' frames to slow down the transmission rate of the program. This was because the MAC address chosen at random for the FPGA Thin Client turned out by chance to have the broadcast bit set. The broadcast bit caused the frames generated to be received by all devices on the network, and as the wireless router could not sustain more than 2MB/s data rate it transmitted Pause frames to slow down the transmission rate.

The solution was simply to pick a different MAC address for the Thin Client without the broadcast bit set. However this problem brought to attention another possible problem. Ethernet switches learn which port to forward frames to from the Source MAC address of received frames. Without any frames received from the FPGA Thin Client device an Ethernet Switch would not learn which port the device was connected to. Without learning the port the Switch would simply broadcast to all ports rather than to the correct port. To prevent this the Thin Client device should be pinged so that it sends an acknowledgment frame before RGB data is transmitted. The acknowledgment frame would cause all Ethernet Switches in the path between the Thin Client and the server to learn which port frames addressed to the Thin Client should be forwarded to.

5.5 System Integration

After the FPGA Thin Client device and the server side software had been tested separately, the next step was to verify that the entire system functioned as expected. The first test was to verify that a stream of raw RGB frames from a High Definition video could be transmitted to the FPGA and displayed on the connected HDMI monitor. The raw RGB frames were generated using the open source video transcoding program Mencoder. The generated RGB frames were then transmitted via Ethernet to the Thin Client using the 'raw' program. The sustained bandwidth was measured using a small shell script. Figure 5.9 shows the video displayed on the HDMI monitor by the Thin Client.

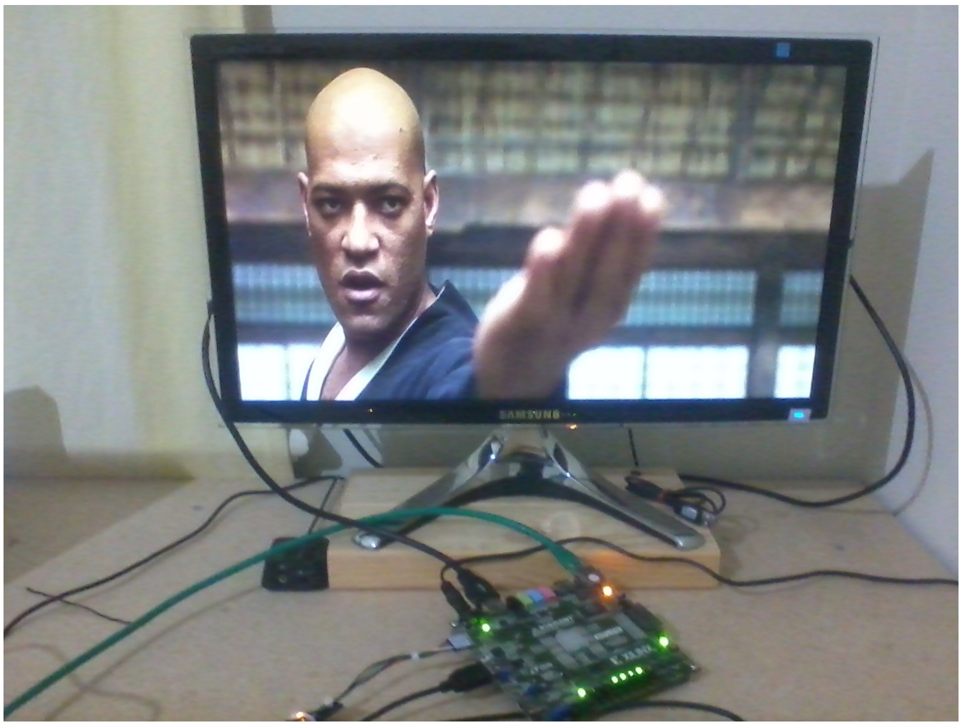
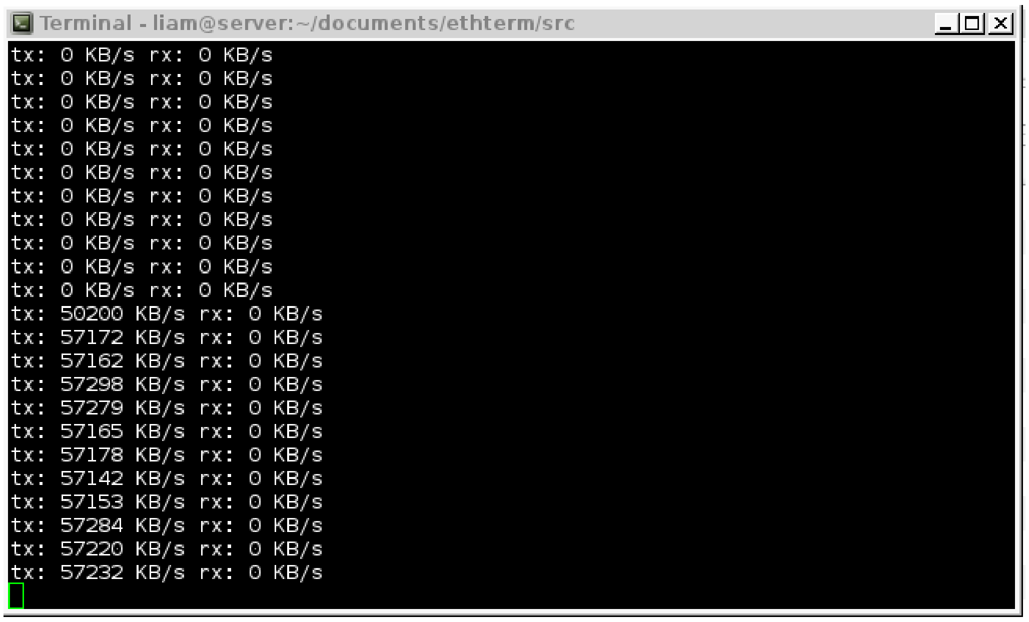


Figure 5.9: A HD video displayed using the FPGA Thin Client

Figure 5.10 shows the bandwidth sustained during the video playback test.



```
tx: 0 KB/s rx: 0 KB/s
tx: 0 KB/s rx: 0 KB/s
tx: 0 KB/s rx: 0 KB/s
tx: 0 KB/s rx: 0 KB/s
tx: 0 KB/s rx: 0 KB/s
tx: 0 KB/s rx: 0 KB/s
tx: 0 KB/s rx: 0 KB/s
tx: 0 KB/s rx: 0 KB/s
tx: 0 KB/s rx: 0 KB/s
tx: 0 KB/s rx: 0 KB/s
tx: 0 KB/s rx: 0 KB/s
tx: 50200 KB/s rx: 0 KB/s
tx: 57172 KB/s rx: 0 KB/s
tx: 57162 KB/s rx: 0 KB/s
tx: 57298 KB/s rx: 0 KB/s
tx: 57279 KB/s rx: 0 KB/s
tx: 57165 KB/s rx: 0 KB/s
tx: 57178 KB/s rx: 0 KB/s
tx: 57142 KB/s rx: 0 KB/s
tx: 57153 KB/s rx: 0 KB/s
tx: 57284 KB/s rx: 0 KB/s
tx: 57220 KB/s rx: 0 KB/s
tx: 57232 KB/s rx: 0 KB/s
```

Figure 5.10: Screen capture of the bandwidth monitoring script output

The 57MB/s bandwidth utilization was a limitation of the test PC rather than the Thin Client. The same bandwidth was used regardless of whether the Thin Client was connected or not. Greater data rates would be achievable using a more powerful test PC with a more capable Network Interface Card (perhaps an Intel Gigabit adapter rather than the Realtek 8169 used in the test).

A frame rate of up to 30 frames per second was seen in the test, although this varied between 20 and 30 as no frame rate control was implemented in either the Thin Client or the 'raw' program. This test verified that the FPGA Thin Client design could receive RGB data via Ethernet and display the RGB data on a HDMI monitor. It also verified that the design could operate at high data rates.

The next test performed was to verify that the FPGA Thin Client would function correctly with the modified QEMU virtualization software. An Ubuntu Live CD was loaded as the virtualized guest under QEMU.

Figure 5.11 shows the Live CD startup screen being displayed by the FPGA Thin Client.

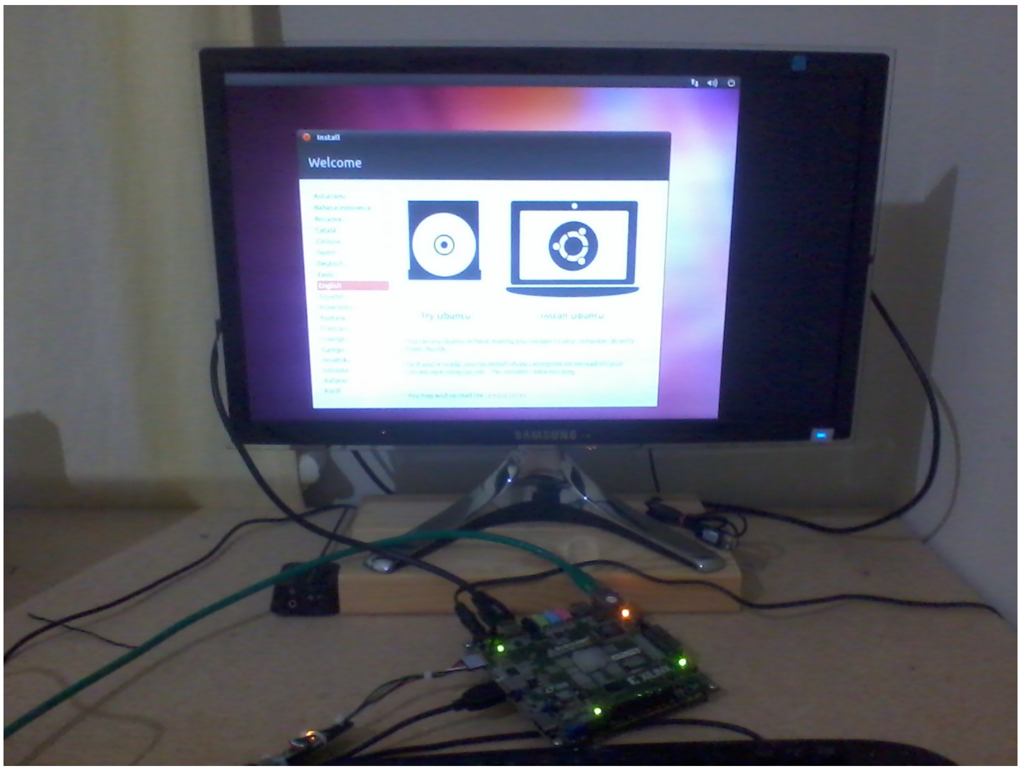
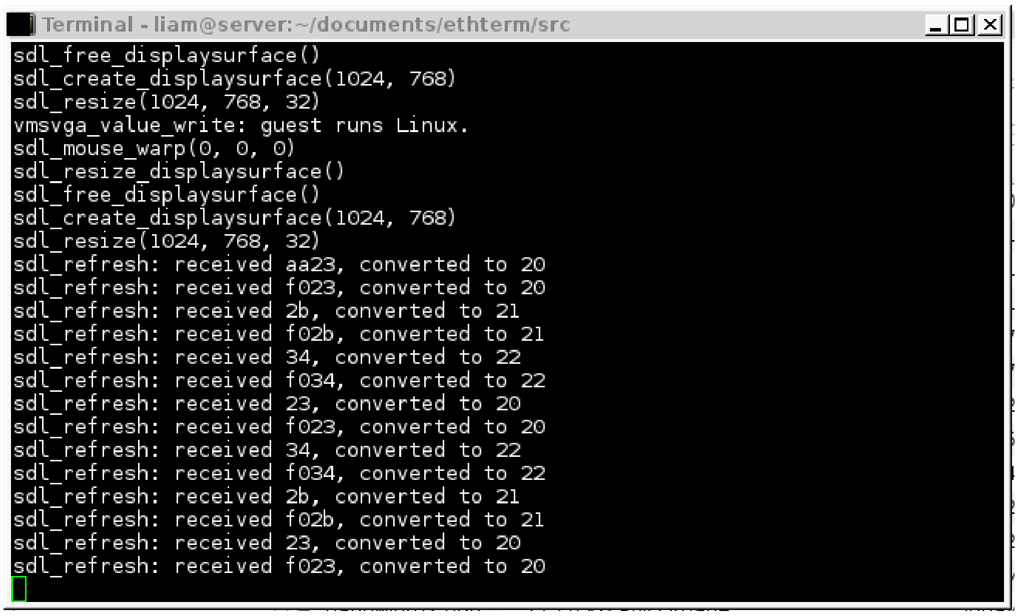


Figure 5.11: Testing the FPGA Thin Client with the Modified QEMU software

The keyboard connected to the FPGA worked, although initially some keys such as the arrow keys did not seem to function. The problem was that the PS/2 to XT key code conversion table implemented in the modified QEMU software did not include conversions for these keys. The solution was to manually add the keys that did not work to the conversion table, recompile, and restart the test.

Figure 5.12 shows the output of the modified QEMU. Note the 'sdl_refresh' lines showing the PS/2 scan codes received in the raw Ethernet frames from the client and the XT key codes they were converted to.

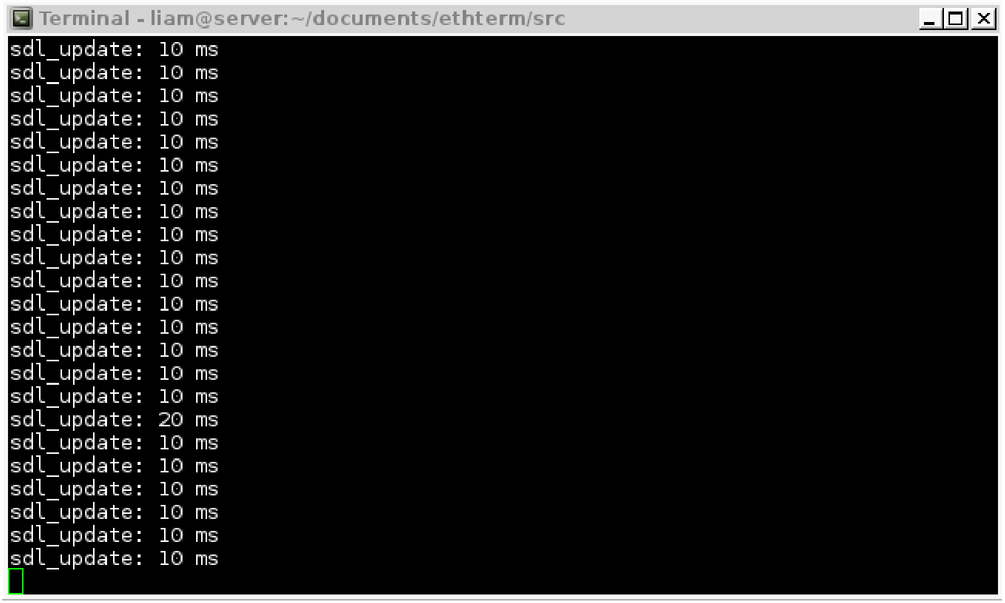

Terminal - liam@server:~/documents/ethterm/src

```
sdl_free_displaysurface()
sdl_create_displaysurface(1024, 768)
sdl_resize(1024, 768, 32)
vmsvga_value_write: guest runs Linux.
sdl_mouse_warp(0, 0, 0)
sdl_resize_displaysurface()
sdl_free_displaysurface()
sdl_create_displaysurface(1024, 768)
sdl_resize(1024, 768, 32)
sdl_refresh: received aa23, converted to 20
sdl_refresh: received f023, converted to 20
sdl_refresh: received 2b, converted to 21
sdl_refresh: received f02b, converted to 21
sdl_refresh: received 34, converted to 22
sdl_refresh: received f034, converted to 22
sdl_refresh: received 23, converted to 20
sdl_refresh: received f023, converted to 20
sdl_refresh: received 34, converted to 22
sdl_refresh: received f034, converted to 22
sdl_refresh: received 2b, converted to 21
sdl_refresh: received f02b, converted to 21
sdl_refresh: received 23, converted to 20
sdl_refresh: received f023, converted to 20
```

Figure 5.12: Modified QEMU software output

Using the keyboard to control the virtualized machine was responsive and no latency was noted between keyboard button presses and the screen being updated. Scrolling in a web browser was also tested, a task which is often very frustrating because of the slow update speeds when using when using VNC. However with the FPGA Thin Client the web page scrolled fairly smoothly. This was a very subjective test, but it supported the theory that by reducing the computational power required at the server, even if the reduction was at the expense of network bandwidth, the reduction would reduce screen update latency.

A test was performed in an attempt to measure the time screen updates took to be transmitted to the Thin Client. The 'sdl_update' function of QEMU was modified to print the time elapsed between the function being called and exiting. Then a full screen graphical screen saver preview was run on the virtualized system (the 'Floating Flakes' screensaver preview on the Ubuntu Live CD. Figure 5.13 shows the results.



```
sdl_update: 10 ms
sdl_update: 10 ms
sdl_update: 10 ms
sdl_update: 10 ms
sdl_update: 10 ms
sdl_update: 10 ms
sdl_update: 10 ms
sdl_update: 10 ms
sdl_update: 10 ms
sdl_update: 10 ms
sdl_update: 10 ms
sdl_update: 10 ms
sdl_update: 10 ms
sdl_update: 10 ms
sdl_update: 10 ms
sdl_update: 10 ms
sdl_update: 20 ms
sdl_update: 10 ms
sdl_update: 10 ms
sdl_update: 10 ms
sdl_update: 10 ms
sdl_update: 10 ms
sdl_update: 10 ms
```

Figure 5.13: Screen update latency testing

The resolution of the timer was only 10ms, such that update times of less than 10ms appeared as 0ms. These 0 ms results were not printed by the 'sdl_update' function. This test was not a very accurate measure of screen update latency because it only measured the time taken for the 'sdl_update' function to write RGB data to the systems network transmit buffer in memory. It did not measure the actual time taken to transmit a frame.

A better way to measure the screen update times was by calculating the amount of data needed per frame, and dividing this by the measured bandwidth. For a 1280x720 frame at 32 bits per pixel, 3.69 MB of uncompressed data per frame would need to be transmitted. With a data transfer rate of 57MB/s this equates to a frame update latency of about 155 ms. Using 16 bits per pixel this would be halved to 78 ms. Note that these times are worst case scenarios, when the entire screen needs to be updated between frames. When only a small area of the screen needs to be updated between frames, such as when editing a document, the update time would be dramatically reduced.

Another test performed was an attempt to measure the processor usage of the Modified QEMU program transmitting frames to the FPGA Thin Client, and compare the result to the processor usage of an unmodified QEMU program sending frames to a VNC client. The processor usage was difficult to measure accurately as it varied so much with time. Figure 5.14 below shows a comparison between the processor usage of each solution.

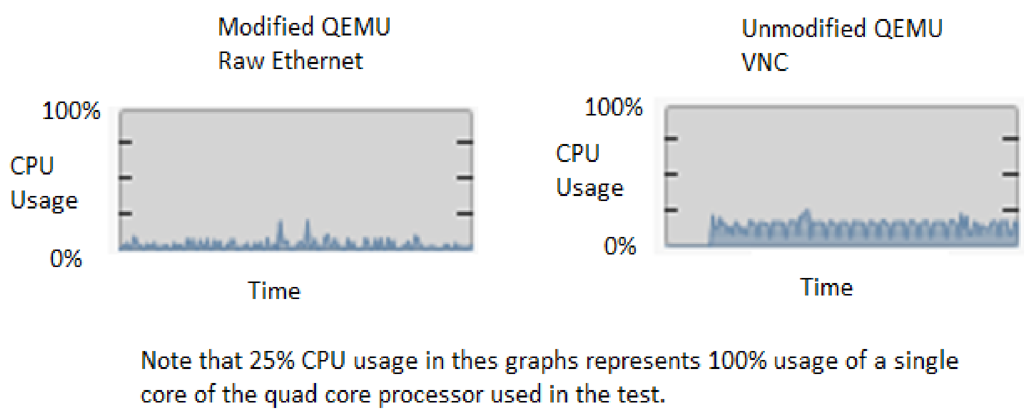


Figure 5.14: CPU usage comparison between FPGA Thin Client and VNC

The VNC test utilized between 90 and 100 percent of a single core on a quad core processor machine, while the Raw Ethernet test only utilized about 20 percent on average. The VNC frame rate was limited by the speed of the CPU while the Raw Ethernet frame rate was limited by the transfer rate of the network.

6.0 Conclusions

6.1 Analysis of Results

The main goal of this project was to show that an FPGA based Thin Client could be designed to operate a virtualized desktop on a remote server. This goal was achieved in that a prototype Thin Client was designed, implemented, and verified on a FPGA development board. The prototype Thin Client was capable of decoding and displaying 1280x720 resolution frames at 30 frames per second on a connected HDMI display. Data transfer rates of up to 57MB/s were measured on the Gigabit Ethernet interface. A keyboard connected to the FPGA could be used to control a remote virtualized desktop with full screen updates completing in less than 200ms, and normal screen updates completing in a fraction of that time.

When the FPGA Thin Client solution was compared to a more traditional VNC solution, the processor usage was reduced on the virtualization server by up to 70% using the FPGA Thin Client. This was because the VNC solution tested used JPEG compression to encode frames whereas the FPGA Thin Client solution only used uncompressed RGB frames.

However, the FPGA Thin Client solution had some drawbacks. One was that a Gigabit Ethernet connection was necessary between the Thin Client and the virtualization server. Wireless Ethernet or Fast Ethernet would not provide enough communication bandwidth to make the FPGA Thin Client solution worthwhile over other more traditional Thin Client solutions. Another drawback was that because raw Ethernet frames were used, the virtualization server and FPGA Thin Client needed to be on the same network. The raw Ethernet frames could not be routed to different sub-networks or over the internet, limiting the possible use cases for the solution.

Overall the FPGA Thin Client solution could best be applied to scenarios where Gigabit Ethernet networking infrastructure was already in place, and where reducing processing requirements of the virtualization server was a top priority.

6.2 Future Improvements

A number of improvements could be made to the design. One improvement would be to add USB device support at the FPGA Thin Client. The prototype designed uses the PS/2 protocol for communication with input devices. USB support would allow newer input devices to be connected, along with other types of device besides input devices. In particular USB support would allow users to connect USB mass storage devices to the Thin Client.

Another improvement would be to add support for some means of data encryption so that raw keyboard scan codes would not be transmitted via Ethernet. This would reduce the security risk of typed passwords being collected by capturing Ethernet frames.

Another improvement would be to implement dynamically switchable video output resolutions, such that the video resolution could be selected during normal operation. The prototype Thin Client only supports one resolution. To change the resolution of the prototype requires the HDL design to be modified and the FPGA reconfigured.

6.3 Future Developments

An alternative solution to reducing the processing overhead at the virtualization server would be to offload the image compression to a specialized device (such as an FPGA or GPU) at the server. This alternative solution would achieve a similar result to the FPGA Thin Client solution, however, the communication network would not be limited to Gigabit Ethernet. This would allow more versatile communication methods to be used to connect the Client device and the server, such as wireless Ethernet or even broadband internet connections. One caveat is that the Thin Client device would need to support image decompression, but with more powerful FPGA and SoC devices being manufactured all the time, this would not be too great a problem.

References

1. R. Baljeu. (2009). *Ubuntu Diskless Fat Client.* [Online]. Viewed 2011 June 28. Available: http://rjb.home.xs4all.nl/fatclient.html

2. Linux Terminal Server Project. (undated). *Linux Terminal Server Project.* [Online]. Viewed 2011 June 28. Available http://www.ltsp.org

3. Hewlett Packard. (2011). *HP t5335z Smart Client.* [Online]. Viewed 2011 June 28. Available: http://h10010.www1.hp.com/wwpc/pscmisc/vac/us/product_pdfs/HP_t5335z_Data_Sheet.pdf

4. Pano Logic, Inc. (2010). *Pano Zero Client Reference Architecture.* [Online]. Viewed 2011 August 20. Available: http://www.panologic.com/zero

5. K. Fogarty. (undated). *Desktop Virtualization.* [Online]. Viewed 2011 June 28. Available: http://www.cio.com/article/504348/Desktop_Virtualization_5_Most_Popular_Flavors_Explained

6. Datapro.net. (undated). *All About DVI.* [Online]. Viewed 2011 June 28. Available: http://www.datapro.net/techinfo/dvi_info.html

7. Usb.org. (undated). *USB FAQ.* [Online]. Viewed 2011 June 28. Available: http://www.usb.org/developers/usbfaq

8. Intel Corporation. (2002). *Enhanced Host Controller Interface Specification.* [Online]. Viewed 2011 September 5. Available: http://www.intel.com/technology/usb/ehcispec.htm

9. Digilent, Inc. (2010). *Atlys Board Reference Manual.* [Online]. Viewed 2011

January 10. Available: http://www.digilentinc.com/Data/Products/ATLYS/Atlys_rm.pdf

10. OpenCores. (undated). *OpenCores.* [Online]. Viewed 2011 January 10. Available: http://opencores.org/

11. D. Kim. (2004). *Gigabit Media Independent Interface.* [Online]. Viewed 2011 September 5. Available: http://hearlink.tripod.com/CandCDB/GMII_REPORT.pdf

12. Digital Display Working Group. (1999). *Digital Visual Interface.* [Online]. Viewed 2011 September 5. Available: http://www.ddwg.org/lib/dvi_10.pdf

13. Qemu.org. (undated). *QEMU Open Source Processor Emulator.* [Online]. Viewed 2011 September 5. Available: http://wiki.qemu.org/Main_Page

14. Wireshark.org. (undated). *Wireshark*. [Online]. Viewed 2011 September 5. Available: http://www.wireshark.org/

Appendix 1

Initial Project Plan

Stage	Tasks	Time Frame
Research	• Research HDMI, Ethernet and DDR2 interfaces. • Research possibility of using a soft processor core. • Learn basics of Verilog.	1 week
Module Development	• Design, implement, and simulate the Ethernet interface modules, the DDR2 Controller, the PS/2 interface, and the HDMI display modules. • Design, implement and verify the selected processor core. • Create JTAG debug interface for processor core.	6 weeks
System Development	• Integrate modules into a complete system. • Verify communication interfaces between modules. • Verify all hardware modules function correctly when linked together. • Revise and update design to solve integration problems. • Write processor core firmware to control modules and provide debugging information • Modify QEMU software to provide a VNC server compatible with the system • Verify and test modified QEMU software	4 weeks
System Verification and Testing	• Verify and test complete system. • Revise design to increase performance if necessary. • Add additional features.	1 week

Plan amendments:

- It was decided that a processor core would not be used in the final design, so the tasks relating to the design and implementation of the processor core, debug interface, and firmware code were removed from the plan.
- Other commitments caused the project to be put on temporary hold for four

weeks after the eighth week. This caused the project length to be increased from 12 weeks to 16 weeks, although the project was only worked on for 12 of these weeks.

- The DDR2 interface turned out to be much easier to implement than initially anticipated so this was completed in less time. However the Ethernet subsystem was more complicated than anticipated so the extra time gained was used for additional work on the Ethernet modules. The net effect was no increase or reduction of stage length.

Appendix 2

Contents of attached CD:

Directories	**Description of Directory Contents**
/	Contains 'thesis.pdf' which is this document in PDF format.
/thesis_working	Contains the working files used in creating this thesis (images, charts, diagrams).
/verification	Contains high resolution images taken and screen captures created during the verification process. May be viewed for greater clarity if images in this document are unclear.
/ethterm	The Xilinx ISE project directory for the FPGA Thin Client prototype. Contains various project files. The file 'ethterm.xise' may be opened with Xilinx ISE to view the project.
/ethterm/vlog	Contains the Verilog source code for each module used in the design. Also contains the Verilog test bench simulations (filenames suffixed with '_tb').
/ethterm/src	Contains the patched 'sdl.c' source code file for QEMU to support sending raw RGB Ethernet frames to the FPGA Thin Client. The full patched QEMU source tree is compressed and archived in the 'qemu-kvm-0.14.1-patched.tar.gz' file. This archive may be extracted and the source code compiled to produce the QEMU executable used during design verification. The 'qemu_demo.sh' shell script was used to start an Ubuntu Live CD virtual machine using the modified QEMU. The 'raw.c' file is the C source code for a program used in verification to send RGB video frames to the Thin Client to test bandwidth. The 'convert_video.sh' and 'video_demo.sh' shell scripts were used to play videos on the thin client (using 'raw'). The 'bandwidth' shell script was used to calculate the network bandwidth utilized during verification.