Data Transfer Between WinNT and Linux Over a Non-Transparent Bridge

By

Ihab Bishara

Senior Project

COMPUTER ENGINEERING PROGRAM
College of Engineering
California Polytechnic State University
San Luis Obispo
2000

Advisor: Dr. James Harris

TABLE OF CONTENTS

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

ACKNOWLEDGEMENTS

I would like to thank 3Com corp. for funding this project, and making it possible for me and the rest of the group to learn about and work with the latest technologies. 3Com's support enabled us to have the hardware and software needed to make this project come to being.

I would like to thank Dr. James Harris for his continuous support, and his endless efforts to provide us with the right environment. I would also like to thank him for his time that he gave generously to direct, guide and support my work. Last but not least I would also like to thank him for all what he taught me throughout my undergraduate studies and for his assistance in the completion of this document.

I would like to thank Dr. Chris Scheiman for getting me to join this project. I would also like to thank him for the time he spent teaching me Systems and Computer Architecture and for his help throughout the whole project.

Many thanks, to the rest of the group members and all those who made my education an enjoyable experience. Special thanks, to Jim Fischer who was always willing to teach me, help me and work with me on many different issues throughout the whole project.

ABSTRACT

Data Transfer Between WinNT and Linux

Over a Non-Transparent Bridge

Ihab Bishara

An on going joint research project between Cal Poly and 3Com seeks to characterize network performance, identify networking bottlenecks and develop solutions. The group is currently working on an intelligent subsystem that will process the TCP/IP stack and perform web-cashing using Linux on a host co-processor PCI platform. The goal is to free the host from processing the network transactions, making it available for other applications and also to improve the latency and throughput of downloading files off the web. The subsystem is connected to the PCI bus through a non-transparent PCI bridge to avoid collision between the two operating systems. The host system is running on a Pentium® processor over a PCI architecture and the subsystem is running on a StrongARM® /21285 Evaluation Board (EBSA-285.) My contribution to the project was to establish data transfer across the non-transparent Bridge between the host system and the subsystem and to ensure the operation of the SCSI Disk under the ARM-Linux Operating System.

1 Introduction

1.1 Definitions and Glossary of Terms

I would like to start with defining some of the terms I will be using through out the document.

Linux

Linux is a Unix clone written from scratch by Linus Torvalds with assistance from a loosely knit team of hackers across the Net. [5]

ARM-Linux

ARM Linux is a port of the successful Linux Operating System to ARM processor-based machines mainly by Russell King with contributions from others. [6]

PCI

Peripheral Component Interconnect (PCI) is a specification that defines the PCI bus. This bus is intended to define the interconnect and bus transfer protocol between highly-integrated peripheral adapters that reside on a common local bus on the system board or add in expansion cards on the PCI bus. [1]

PCI-to-PCI Bridge

A device that connects one PCI bus to another but only places one electrical load on its host bus. The new PCI bus can then support a number of additional devices and/or PCI expansion connectors. PCI-to-PCI Bridges solve the problem of

the limited number of devices per bus. With PCI-to-PCI Bridges all of the devices on the secondary side are apparent to the host and the device drivers so no changes are needed to a device driver when a PCI peripheral is located behind a PCI-to-PCI Bridge. [1] [3]

Upstream

When a transition is initiated and is passed through the bridge flowing towards the host processor. [1]

Downstream

When a transition is initiated and is passed through the bridge flowing away from the host processor. [1]

Primary PCI Bus

It is the PCI bus on the upstream side of a bridge. [1]

Secondary PCI Bus

It is the PCI bus on the downstream side of a PCI-to-PCI bridge. [1]

Non-Transparent PCI-to-PCI Bridge

It is a bridge specifically designed to connect two processor domains. The host domain (on the primary side of the bridge) is controlled by the host processor and the local domain (on the secondary side of the bridge) is controlled by the local processor. [3]

SCSI

Small Computer System Interface, is a set of evolving ANSI standard electronic interfaces that allow personal computers to communicate with peripheral hardware such as disk drives, tape drives, CD-ROM drives, printers, and scanners faster and more flexibly than previous interfaces. [8]

EBSA-285

This is an evaluation board for the 21285, which comes as a PCI card. The PCI card can be configured to be a host or an add-in card. In host mode, it can be plugged into a stand-alone back plane and get a PCI system built up. In add-in mode, it can be plugged into a PCI slot in a standard PC. [2]

Core Logic

A core logic chip of a certain processor is its interface to its main memory, ROM, and the different kinds of busses available to its system.

NIC

A Network Interface Card (NIC) is a computer circuit board or card that is installed in a computer so that it can be connected to a network.

Intelligent Subsystem

It is a system connected to the host that has its own processing power to perform certain jobs for the host system.

Configuration type Zero

It is the configuration for all PCI devices except for PCI-to-PCI and PCI-to CardBus bridges. Those devices have a header type Zero, which is similar to the 21554 primary or secondary headers in Table 3 and Table 4. [1]

Configuration type One

It is a configuration for PCI-to-PCI bridges [1]. Transparent PCI-to-PCI bridges have header type one. Keep in mind that although the 21554 is a PCI-to-PCI bridge it is a nontransparent one. The 21554 does not have a header type one and does not answer to any of the configuration transactions of that type.

1.2 An overview of the architecture

As seen in Figure 1 the system consists of two parts, the host system and the intelligent subsystem. The subsystem is connected to the host through the Primary PCI bus.

The Intelligent subsystem consists of four boards:

1- The Nontransparent PCI-to-PCI bridge evaluation board, which contains the nontransparent PCI-to-PCI bridge chip (21554) and the secondary PCI bus. The secondary PCI bus has four available PCI option card slots one of which is capable of becoming the local processor with the insertion of a single board computer. The board is inserted into an empty PCI slot on the Primary bus. The bridge and its functions will be discussed in details in section *Intel Nontransparent PCI-to-PCI Bridge (21554)*.

2- The **E**valuation **B**oard with a **S**trong**A**RM processor and the 21**285** chip as its core logic (EBSA285.) This board is inserted into the local processor slot on the secondary PCI bus. The board is configured as the host of the local system and will be running ARM-Linux. The board, its configuration and its architecture will be discussed in section 3.0.

3- The Network Interface Card (NIC), which connects the subsystem to the network.

4- The Adaptec 2940U2W SCSI adapter board, which is the interface between the subsystem and the SCSI disk. This adapter is capable of handling up to 15 SCSI peripherals. The SCSI Interface will be discussed in details in section 4.

We are currently using a 19GB Ultra 2 Wide SCSI disk. Both the primary and the secondary buses are 32-bit wide and running at 33 MHz. The strong arm is running at 228 MHz with the capability to run at 287 MHz. There are 16 MB of memory on the EBSA board and it is expandable to 128 MB.

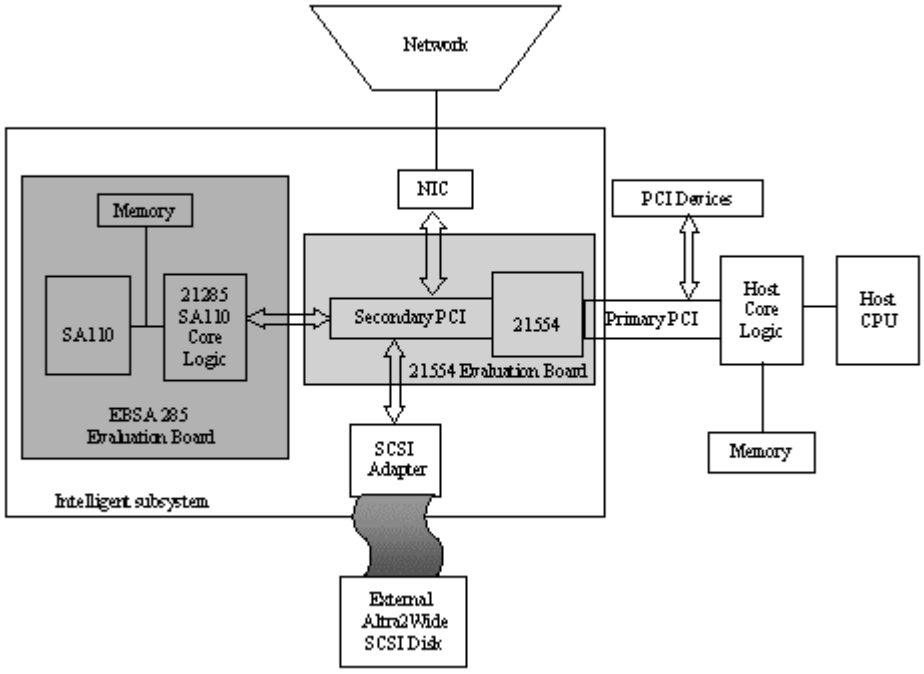

Figure 1 Project Architecture

1.3 The 3Com Group

The 3Com group consists of four faculty members, six students, and an administrator. In this section I will describe briefly the role of each of the members of the group.

1.3.1 Faculty

Dr. James Harris

Dr. Harris is a professor in the Electrical Engineering department and leader of this project. He is also my senior project advisor.

Dr. Mei-Ling Liu

Dr. Liu is a professor in the Computer Science department who worked extensively on the web caching functionality.

Dr. Chris Scheiman

Dr. Scheiman is a professor in the Computer Science department in the Computer Engineering Program who worked on the Architecture of the system and time stamping for analysis of performance.

Dr. Hugh Smith

Dr. Smith is a professor in the Computer Science department worked on the networking part of the project and on the future possibilities and uses of the project.

1.3.2 Students

Jim Fisher

Jim is a graduate EE student. He is working on TCP/IP performance analysis and porting Linux to the subsystem.

Ying-Lin Chen

Ying-Lin is a graduate CSC student. She is working on the web caching functionality.

Bo Wu

Bo is a graduate EE student. He is working on the interface between the subsystem and the host from the subsystem side.

Peter Huang

Peter is a graduate CSC student. He is working on the interface between the subsystem and the host from the Host side.

Eric Engstrom

Eric is an undergraduate CSC student. He is working on porting Linux down to the Subsystem.

Ihab Bishara

I am working on the data transfer between both systems over the nontransparent bridge and the interface with SCSI. A more detailed description of my role is explained in the following section

1.3.3 Administrator

Shayna Leib

Shayna is the administrator who takes care of all the paper work and the administrative needs of all the members of the project.

1.4 My Role in the project

I started in this project by working on the SCSI Interface. The goal at that time was to write a driver that runs on the EBSA and controls a SCSI adapter. The SCSI disk is the storage for the web caching performed by the system.

After having difficulty porting the TCP/IP stack down to the EBSA-285 it was decided to port ARM-Linux down to the EBSA board. Since Linux has drivers and

can process the TCPI/IP stack both of those problems were solved, but another issue was raised concerning the two operating systems running on the same bus. My role was then changed to working with the rest of the group to consider the consequences of that issue. After some research we found out about the non-transparent PCI-to-PCI bridges designed specially to isolate subsystems from the main host system. I then worked on understanding the non-transparent bridge architecture and how to integrate it into our system without having to rework what was accomplished to this point. I worked with the rest of the group and was able to configure both the bridge and the secondary PCI bus and transfer data across the bridge using the same protocol other members of the team have developed for communication between the host system and the subsystem.

2 Intel Nontransparent PCI-to-PCI Bridge (21554)

2.1 Introduction

The 21554 is a bridge chip that is specially designed to enable developers to design a whole system that would look like one device to the host. It has a 64-bit primary and secondary interface, both running at 33 MHz. The primary interface is also called the host interface and the secondary interface is also called the subsystem or the local interface. By hiding all the devices on the secondary bus from the host the 21554 allows the local processor to initialize and control the devices on the secondary bus and also solves any resource conflicts between the local and host processor. The 21554 is capable of isolating both systems by having two different PCI address spaces, one for each system. To forward a transaction from one side of the bridge to the other, the 21554 translates one address to the other using its translation registers.

The 21554 also contains an arbiter function capable of supporting up to nine devices on the secondary PCI bus. The 21554 is configured through a serial Rom interface and through both primary and secondary PCI Interface. The 21554 has four primary interface base address registers for downstream data forwarding one of which is programmable to either memory or I/O and the other three are for forwarding memory transactions. On the secondary side there are three Base Address registers for

upstream data forwarding: one of which is programmable to either memory or I/O transactions, another is for memory transactions, and the third is also for memory transactions but, uses a look up table for translation between the two address domains instead of the translation registers.

2.2 Comparison with transparent PCI-to-PCI Bridge

The main difference between the 21554 and a transparent PCI-to-PCI Bridge is that with a transparent bridge the secondary bus and the devices on it are apparent to the host while with a 21554 the secondary bus and the devices on it are hidden from the host. Table 1 presented in the 21554 Hardware Reference Manual compares the main features of both the 21554 and a transparent PCI-to-PCI bridge.

Table 1 Comparison with transparent PCI-to-PCI Bridge [3]

Feature	**21554**	**PCI to PCI Bridge**
Transaction forwarding	-Adheres to PPB ordering rules. -Uses posted writes and delayed transactions. -Adheres to PPB transaction error and parity error guidelines, although some errors may be reported differently.	-Adheres to PPB ordering rules. -Uses posted writes and delayed transactions. -Adheres to PPB transaction error and parity error guidelines.
Address decoding	-Base address registers are used to define independent downstream and upstream forwarding windows. -Inverse decoding is only used for upstream transactions above the 4GB boundary.	-PPB base and limit address registers are used to define downstream forwarding windows. -Inverse decoding for all I/O and memory upstream forwarding.
Address translation	Supported for both memory and I/O transactions	None. Flat address model is assumed.

Feature	21554	PCI to PCI Bridge
Run-time resources	-Includes features such as doorbell interrupts, I_2O message unit, and so on, that must be managed by the device driver.	-Typically has only configuration registers; no device driver is required.
Clocks	-Generates secondary bus clock output. Asynchronous secondary clock input is also supported.	-Generates one or more secondary bus clock outputs.
Configuration	-Downstream devices are not visible to host. -Does not require hierarchical configuration code (Type 0 configuration header). -Does not respond to Type 1 configuration transactions. -Supports configuration access from the secondary bus. Implements separate set of configuration registers for the secondary interface.	-Downstream devices are visible to host. -Requires hierarchical configuration code (Type 1 configuration header). -Forwards and converts Type 1 configuration transactions. -Does not support configuration access from the secondary bus. Same set of configuration registers is used to control both primary and secondary interfaces.
Secondary bus central functions	-Implements secondary bus arbiter. This function can be disabled. -Drives secondary bus AD, C/BE#, and PAR during reset. This function can be disabled.	-Implements secondary bus arbiter. This function can be disabled. -Drives secondary bus AD, C/BE#, and PAR during reset.

2.3 Architecture Overview

As Shown in Figure 2 the 21554 consists of three types of functional blocks: data buffers, registers and control logic

2.3.1 Data Buffers

They are used to enable the 21554 to handle multiple transactions on each interface and burst transactions.

2.3.2 Registers

The 21554 includes the device configuration registers, the primary PCI configuration registers and the Secondary PCI configuration registers all of which are accessible from both Primary and secondary interfaces.

2.3.3 Control Logic

The 21554's logic includes the Secondary bus arbitration logic, JTAG control logic, ROM interface control logic and the Primary and secondary target/master control logic.

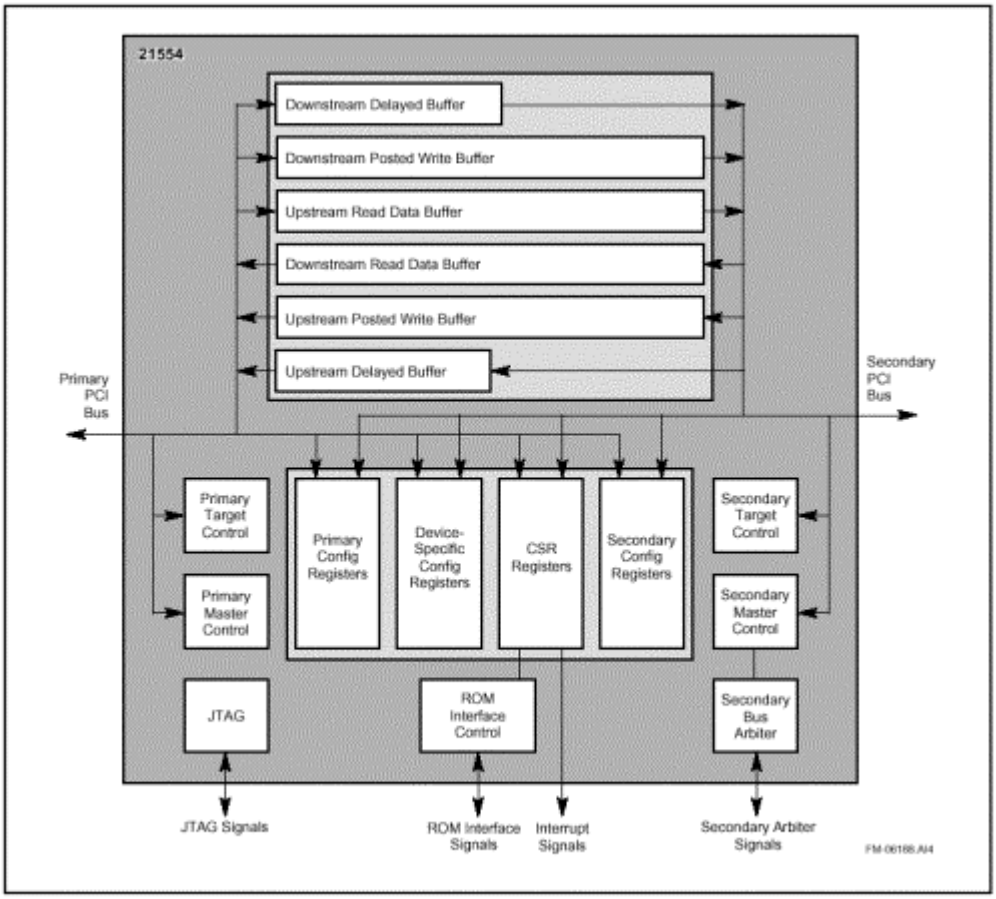

Figure 2 The 21554 Microarchitecture [3]

2.4 Why 21554?

After the decision was made to port Linux down to the EBSA-285 the problem of having two operating systems on the same bus was raised. To solve this problem I started looking for similar projects with similar problems. I found a lot of add in cards that had a whole computer on a card. These cards were to be plugged in a PCI interface of a computer (host) using the 21554 chip. From the Intel developer site

I downloaded the manuals on the chip, and presented the idea to the rest of the group in our weekly meeting and to 3Com in our monthly videoconference. After some discussions we came to the conclusion that the idea of a nontransparent bridge would give us a much cleaner environment to work with in our development. The bridge will isolate the two operating systems giving the subsystem the capability to initialize and control its own resources preventing any resource conflicts.

2.5 21554 Evaluation Board

The 21554 came as an evaluation board with a 64 bit interface that plugs into a host PCI bus and provides a secondary bus with four 64-bit PCI optional card slots one of which could be used for a host bridge card as shown in Figure 3. The board also has a serial Rom that stores code for initialization of the 21554 on system power up. Also, the initialization switches on the board can control some of the bridge's settings.

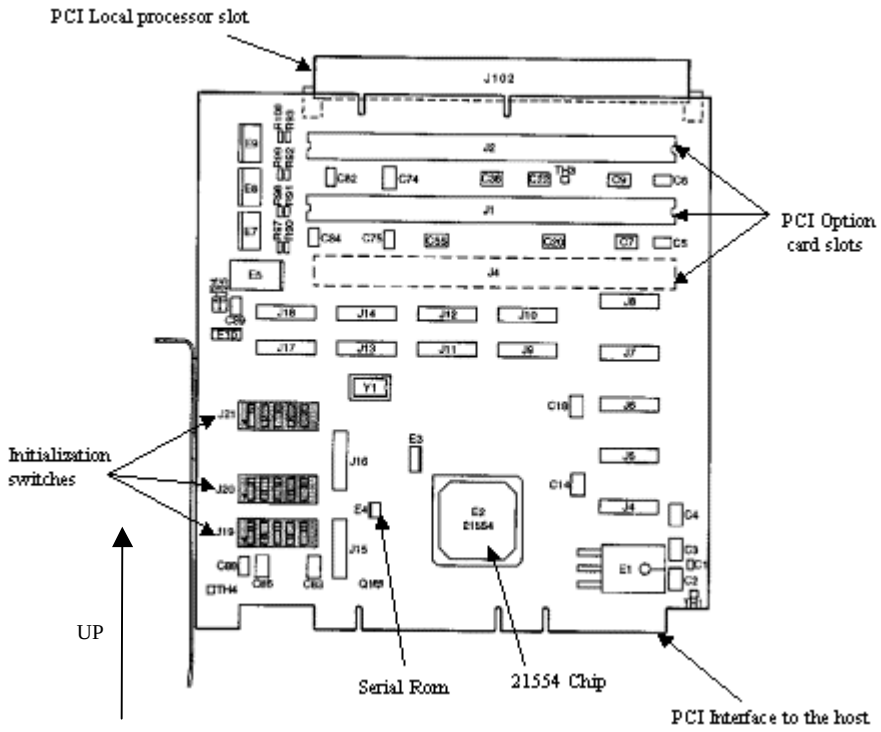

Figure 3 -21554 Evaluation Board [4]

Table 2 describes the operation of the three switches and the last column includes the setup I have for the project. Looking at Figure 3 SW1 is the left most

switch while SW5 is the right most one. J21 is the upper switches set with J20 in the middle and J19 in the bottom. Up is towards the system slot (J102).

Table 2 Initialization Options and Project Setting [4]

Location	DIP Switch	Description	Switch Down	Switch Up	Project setting
J19	SW1	Arbitration control	System slot (J102) as arbiter	21554 as arbiter	UP
	SW2	Arbitration control	System slot (J102) as arbiter	21554 as arbiter	UP
	SW3	MDE ROM Control	21554 **pr_cs**	MDE ROM test	UP
	SW4	MDE Socket Control	21554 **pr_cs**	MDE Socket test	UP
	SW5	**Cfg<1>** Control	Local Clock Divide Disabled	Local Clock Divide Enabled	DOWN
J20	SW1	SROM **pr_ad<2>:sr_do**	Serial ROM Output Disabled	Serial ROM Output Enabled	UP
	SW2	Lockout Bit Reset Value **pr_ad<3>**	No lockout (debug)	Lockout (normal operation)	UP
	SW3	Synchronous/ Asynchronous Operation **pr_ad<4>**	Asynchronous host and local clock domains	Synchronous host and local clock domains	UP
	SW4	**s_clk_o pr_ad<5>**	Disable 21554 **s_clk_o**	Enable 21554 **s_clk_o**	UP
	SW5	Central Function Mode **pr_ad<6>**	21554 as Central Function	System slot (J102) as Central Function	UP
J21	SW1	Arbiter Function **pr_ad<7>**	Disable 21554 arbiter	System slot (J102) as external arbiter.	UP
	SW2	Arbitration control	System slot (J102) as arbiter	21554 as arbiter	UP

Location	DIP Switch	Description	Switch Down	Switch Up	Project setting
	SW3	Arbitration control	System slot (J102) as arbiter	21554 as arbiter	UP
	SW4	Arbitration control	System slot (J102) as arbiter	21554 as arbiter	UP
	SW5	Arbitration control	System slot (J102) as arbiter	21554 as arbiter	UP

2.6 Configuration Space Registers

The 21554 Configuration space is divided in three parts: the primary interface configuration registers (Table 3), the secondary interface configuration registers (Table 4), and the device specific configuration registers (Table 5). All three parts can be accessed from either the primary or the secondary interface and the offsets to access those registers are specified in the following tables. Keep in mind that the offsets to access the primary configuration registers from the primary interface are different than the offsets to access those same registers from the secondary interface and the same is true for the secondary configuration registers.

The primary and secondary configuration headers consist of the basic 64 bytes of a type zero configuration header. One of the differences between the two is that the primary configuration header has four programmable downstream address ranges while the secondary configuration header has three programmable upstream address ranges. Those address ranges are used to transfer data downstream from the primary interface and upstream from the secondary interface. Each address range is specified in a Base Address Registers (BAR.)

The four downstream address ranges are:

- Downstream memory 0 at primary offset 10h
- Downstream I/O or memory 1 at primary offset 18h
- Downstream memory 2 at primary offset 1Ch
- Downstream memory 3 at primary offset 20h

The three upstream address ranges are:

- Upstream I/O or memory 0 at secondary offset 18h
- Upstream memory 1 at secondary offset 1Ch
- Upstream memory 2 at secondary offset 20h

The device specific registers contain setup registers, translation registers and configuration and control registers. Setup registers are used to mask the BAR to determine the size and type of the memory requested on the PCI bus. A setup register has to be programmed before the corresponding BAR is accessed for configuration. Translation registers are used in the address decoding between the primary and the secondary interfaces. The control and configuration registers set the functionality of the bridge.

Table 3 Primary Interface Configuration Registers [3]

Byte 3	Byte 2	Byte 1	Byte 0	Primary Offset	Secondary Offset
Device ID[1]		Vendor ID[1]		40h	00h
Secondary Status		Secondary Command		44h	04h
Secondary Class Code[2]			RevID[1]	48h	08h
BiST[1,2]	Header Type[1]	Secondary MLT	Secondary CLS	4Ch	0Ch
Secondary CSR Memory BAR				50h	10h
Secondary CSR I/O BAR				54h	14h
Upstream I/O or Memory 0 BAR				58h	18h
Upstream Memory 1 BAR				5Ch	1Ch
Upstream Memory 2 BAR				60h	20h
Reserved				64h	24h
Reserved				68h	28h
Subsystem ID[1,2]		Subsystem Vendor ID[1,2]		6Ch	2Ch
Reserved				70h	30h
Reserved			Capabilities Pointer[1]	74h	34h
Reserved				78h	38h
Secondary MAX_LAT[2]	Secondary MIN_GNT[2]	Secondary Interrupt Pin	Secondary Interrupt Line	7Ch	3Ch

Table 4 Secondary Interface Configuration Registers [3]

Byte 3	Byte 2	Byte 1	Byte 0	Primary Offset	Secondary Offset
Device ID[1]		Vendor ID[1]		00h	40h
Primary Status		Primary Command		04h	44h
Primary Class Code[2]			RevID[1]	08h	48h
BiST[1,2]	Header Type[1]	Primary MLT	Primary CLS	0Ch	4Ch
Primary CSR and Downstream Memory 0 BAR				10h	50h
Primary CSR I/O BAR				14h	54h
Downstream I/O or Memory 1 BAR				18h	58h
Downstream Memory 2 BAR				1Ch	5Ch
Downstream Memory 3 BAR				20h	60h
Upper 32 Bits Downstream Memory 3 BAR				24h	64h
Reserved				28h	68h
Subsystem ID[1,2]		SubsystemVendor ID[1,2]		2Ch	6Ch
Primary Expansion ROM Base Address				30h	70h
Reserved			Capabilities Pointer[1]	34h	74h
Reserved				38h	78h
Primary MAX_LAT[2]	Primary MIN_GNT[2]	Primary Interrupt Pin	Primary Interrupt Line	3Ch	7Ch

Table 5 Device Specific Configuration Registers [3]

Byte 3	Byte 2	Byte 1	Byte 0	Primary Offset	Secondary Offset
Downstream Configuration Address[1]				80h	80h
Downstream Configuration Data[1]				84h	84h
Upstream Configuration Address[1]				88h	88h
Upstream Configuration Data[1]				8Ch	8Ch
Configuration Control and Status[1]		Configuration Own Bits[1]		90h	90h
Downstream Memory 0 Translated Base[1]				94h	94h
Downstream I/O or Memory 1 Translated Base[1]				98h	98h
Downstream Memory 2 Translated Base[1]				9Ch	9Ch
Downstream Memory 3 Translated Base[1]				A0h	A0h
Upstream I/O or Memory 0 Translated Base[1]				A4h	A4h
Upstream Memory 1 Translated Base[1]				A8h	A8h
Downstream Memory 0 Setup Register[2]				ACh	ACh
Downstream I/O or Memory 1 Setup Register[2]				B0h	B0h
Downstream Memory 2 Setup Register[2]				B4h	B4h
Downstream Memory 3 Setup Register[2]				B8h	B8h
Upper 32 Bits Downstream Memory 3 Setup Register[2]				BCh	BCh
Primary Expansion ROM Setup Register[2]				C0h	C0h
Upstream I/O or Memory 0 Setup Register[2]				C4h	C4h
Upstream Memory 1 Setup Register[2]				C8h	C8h
Chip Control 1[2]		Chip Control 0[2]		CCh	CCh
Arbiter Control[2]		Chip Status		D0h	D0h
Reserved		Secondary SERR# Disables[2]	Primary SERR# Disables[2]	D4h	D4h
Reset Control				D8h	D8h
Power Management Capabilities[2]		Next Item Ptr	Capability ID	DCh	DCh
PM Data[2]	PMCSR BSE	Power Management CSR[2]		E0h	E0h
VPD Address		Next Item Ptr	Capability ID	E4h	E4h
VPD Data				E8h	E8h
Reserved	Hot Swap Control	Next Item Ptr	Capability ID[2]	ECh	ECh
Reserved				FF:F0h	FF:F0h

2.7 Serial ROM

Some of the device specific registers can be accessed through the Serial ROM. The process of writing to some of the registers from the Serial ROM is called preloading. Before the Primary or the secondary interfaces get to configure the bridge

the preloading process takes place. Some of the registers that the serial ROM preloads are the setup registers and the control registers. A copy of the configuration file of the Serial ROM that I used in the project is in Appendix B.

2.8 Initialization

At power up both the secondary and primary might try to configure the 21554 bridge. There are several ways to initialize the bridge using a combination of the Serial Rom, the primary interface, and the secondary interface. Unless the Serial ROM is disabled (J20 SW1 is down on the evaluation board) the 21554 will start with the preloading process from the serial ROM. During this process the 21554 returns a target retry to any configuration transaction from either interface. It is recommended in the manual to let the secondary interface initialize the device and configure its PCI space before the primary interface does so. The reason for that is if the secondary interface will access some of the primary interface registers for setting and configuration purposes it should be done before the primary interface starts its configuration. After preloading is completed the secondary lock out bit is cleared allowing the secondary interface to configure the bridge. If the lock out bit for the Primary interface is not cleared by hardware or by the secondary interface the 21554 will continue to send target retry to any configuration access from the primary interface.

It is possible to let both the primary and the secondary interface configure the device at the same time as long neither interface will be initializing registers that will

be used by the other interface in their initialization process. In the project I let both primary and secondary interfaces configure the device simultaneously. I chose that setting because all the initialization needed for configuring either PCI interface could be done during the Serial Rom preload at this stage of the project except for the translation registers. The translation registers are then initialized by the secondary interface for downstream data transfer and by the primary interface for upstream data transfer.

2.9 Configuration

To initially test for the functionality of the bridge I configured only one downstream address range (downstream memory 1) and one upstream address range (upstream memory 1) and disabled the other address ranges through the serial ROM preloading. I then let both primary and secondary interfaces configure the device each from its own interface and allocate the memory for those address ranges. After the memory was allocated the secondary interface sets up the translation register to the required address. I will now go into more details on how these configurations are performed.

2.9.1 Serial Rom configuration

Using the software provided with the evaluation board the serial ROM is loaded using a text file that contains an address and a value for each of the registers to be preloaded. I used that software to disable all the address ranges except for downstream memory 1 and upstream memory 1. The format of the configuration file

to be downloaded to the SROM is shown in Appendix B and a quick explanation of the software used to download it to the serial Rom is also in Appendix B.

2.9.2 Primary interface configuration

At startup the system probes the PCI bus and allocates resources for each device. During configuring the PCI bus, the host side sees the bridge as a one device and starts sending configuration type zero reads and writes to its different configuration registers. To allocate the right type (I/O or memory) and size address space to each device the host has to access the BAR of each device. Each BAR has a setup register that masks it. If a bit in the setup register is 1 then the corresponding bit in the BAR is readable and writable. If a bit in the setup register is 0 then the corresponding bit in the BAR is readable only and will return 0 whenever read. Setup registers must be programmed before any access to the BAR registers. First the host writes all ones to the BAR and then reads it. By writing ones to the BAR the host detects which bits have ones and which have 0's in the setup registers and allocates the right type of address space with the right size to that device. The host then generates a boundary-aligned starting address in the PCI address space for the address range and writes it back to the BAR.

If data was to be transferred upstream to a device on the primary bus the host should write the PCI address of that device to one of the upstream translation registers corresponding to one of the upstream address ranges on the secondary bus.

2.9.3 Secondary interface configuration

The secondary bus is configured in the same way the primary bus is. After both sides of the device are configured, the secondary host (SA110) writes the address of the target device on the secondary bus where we want the data to be transferred (in this case to the SDRAM on the EBSA-285) into the translation register of the downstream memory chosen for the transfer (in this project I chose downstream memory 1 and wrote to its translation register). After this, a path is put together from one device or the host on the primary side of the bridge to another device on the secondary side of the bridge through address decoding.

2.10 Address decoding

Since the bridge has two different address spaces, one for each interface, there has to be a way to transfer data between these two interfaces. The goal is to be able to write to one of the bridge's address ranges on the secondary bus and have the data get to a device on the primary bus and vice versa. To be able to link these two address domains there has to be a way to translate the addresses on the secondary bus address map to those on the primary bus address map.

The 21554 uses three ways to communicate between the two interface: Direct offset translation, a Look-up table used by upstream memory two and Flat addressing utilizing Dual Address Cycles (DAC.)

For Direct offset translation Figure 4 shows an example of an upstream memory transaction and Figure 5 shows how the address is being translated using the translation register. For a data transfer, the host on either side of the bridge reads the

BAR of a target device on its bus and copies that address to the translation register of the particular memory address on the other side of the bridge. For example: through this mechanism by writing to an address in the downstream memory 1 address range on the primary side the base (high bits in the address depending on the size of the address range) will be replaced by the same bits in the downstream memory 1 translation register and sent out on the secondary PCI interface.

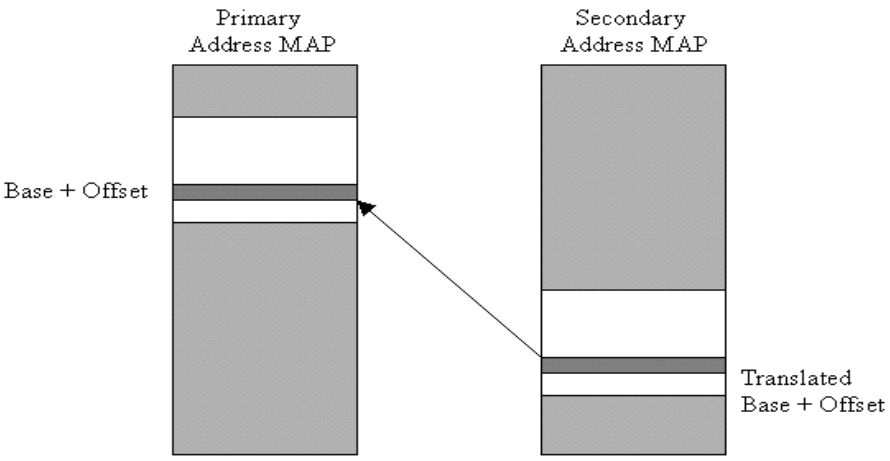

Figure 4 Address translation between the secondary and primary interfaces [3]

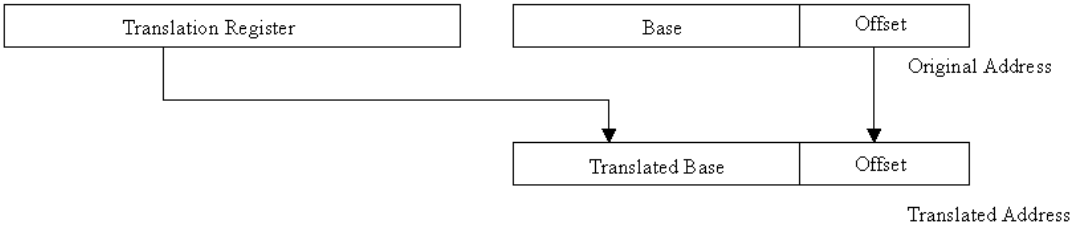

Figure 5 Register level translation of the address between the two interfaces [3]

Transferring data using the Lookup table can be only done on upstream data transfers only through upstream memory two. As illustrated in Figure 6 for an address

that is put on the secondary bus that lies in upstream memory two address range the index (bits [17:12]) and the base (bits [31:18]) of that address field are replaced with the entry in the table corresponding to that index number and then the transaction is put onto the primary bus. The lookup table has 64 entries and it is programmed by the secondary interface. Also the table is implemented in hardware so when reset the table will keep its values until reprogrammed through the secondary interface.

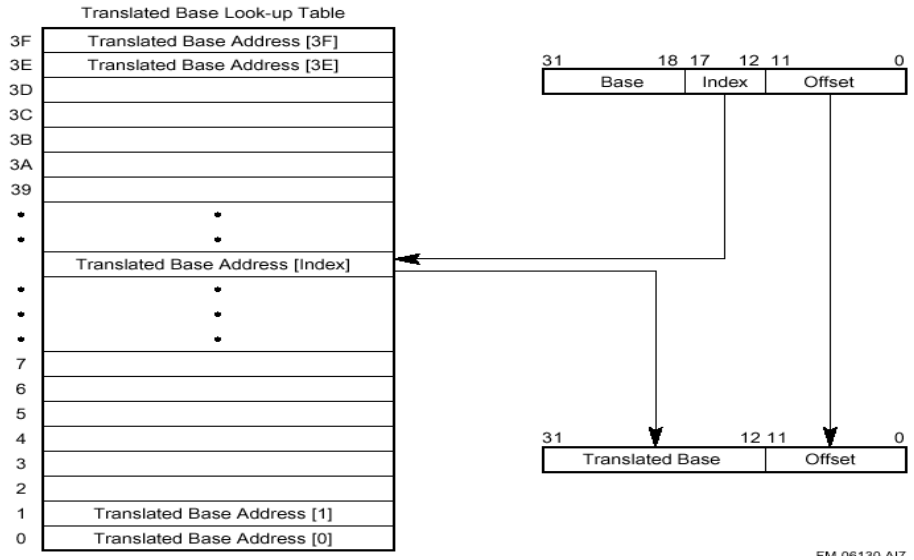

Figure 6 Address decoding using the lookup table [3]

Flat addressing using Dual Address Cycles is used to transfer data in both upstream and downstream data transactions. It utilizes the PCI 64-bit addressing. With this mechanism there is no address translation so the address written on the one interface is transferred to the other interface directly. On the primary interface Downstream memory 3 BAR could be setup for 64-bit transfers by preloading bit [31] to a one in the memory 3 upper 32 bit register (offset 24h from the primary interface). On the secondary interface any 64-bit address falling outside the 32 bits of

the downstream memory 3 BAR is transferred to the primary interface with no address translation. Figure 7 shows the flat addressing mode of the 21554.

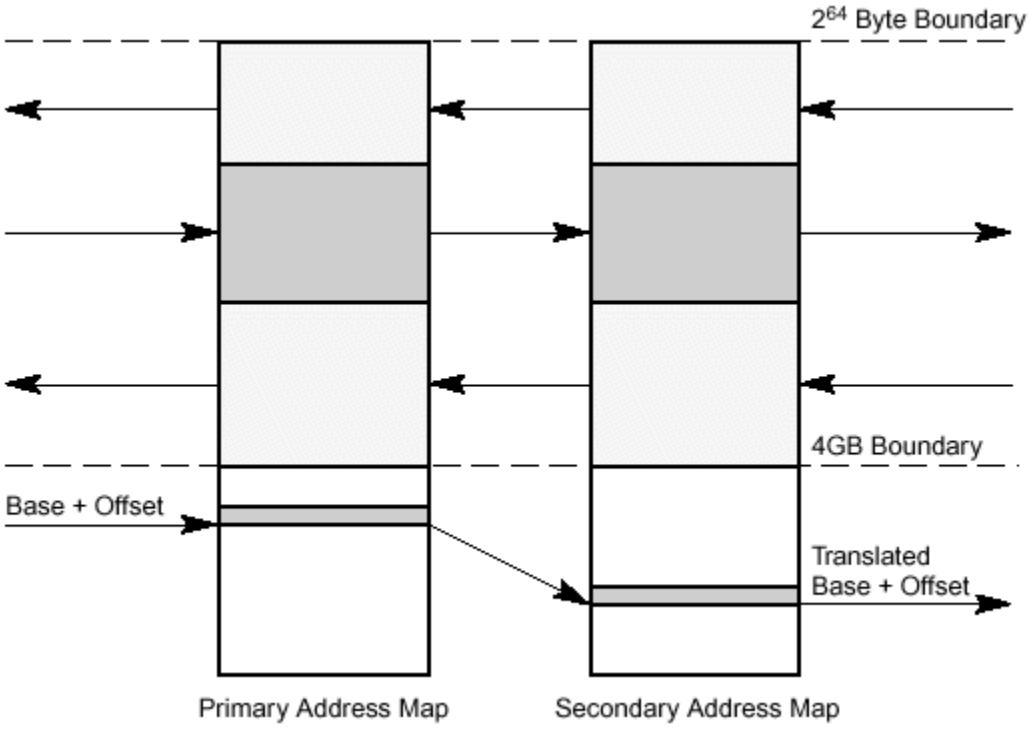

Figure 7 Flat Addressing Transactions Using DAC [3]

In the project so far we have only utilized the direct offset translation just to get a functional model working. The other two addressing modes are available but there were no reasons for using them in the project. Both the lookup table and the DAC addressing expand the bridge's capability to be able to transfer data between more devices than the number of BAR on its interfaces, which could be very beneficial in the progress of the project.

2.11 Software Available

The 21554 Evaluation board came with a few programs to expedite development. These programs are for programming the serial ROM and for reading and writing from and to configuration spaces of PCI devices on either bus. The DOS utility for programming the Serial ROM is called MKSROM.exe and the one for reading from and writing to PCI configuration registers is called pview.exe. The use of the MKSROM utility is explained in Appendix B. Both programs are explained in more details in the documentations provided with the board on the floppy diskettes that came with it.

3 EBSA-285

3.1 Architecture Overview

The EBSA-285 is the evaluation board for the Strong Arm 110 and its core logic 21285. The board has on it: a StrongARM 110 chip; a 21285 chip; two memory modules, one of which is used in the project and has 16 Meg RAM on a DIMM; 4 Meg in Flash ROM; a serial port; a socket for EPROM; and some header blocks for setting.

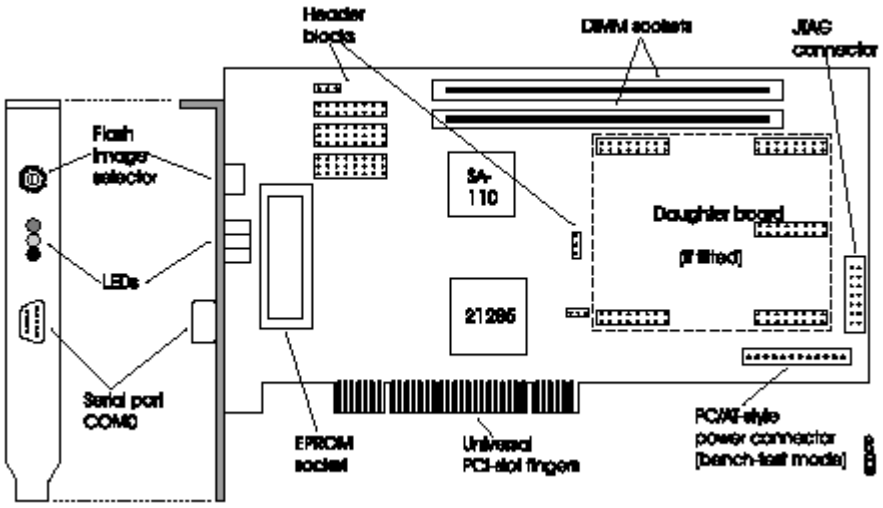

Figure 8 The EBSA 285 Board [7]

The EBSA board has two functional modes, as an add-in card and as a host bridge. In the add-in card mode the EBSA is just another device on the PCI bus and the system is responsible for configuring its PCI space and assigning it the requested

resources. On the other hand in the Host Bridge mode the EBSA is the host of the PCI bus that it is plugged into and is responsible for configuring all the devices and assigning them their requested resources. In this mode, the EBSA could also be the bus arbiter with some restriction on the data transfer on the board itself.

The 21285 on the EBSA Board is the traffic controller handling all the data transactions between all the other chips and interfaces on the board. The 21185 has a direct interface with the address and data lines of the SA processor and the memory. It also has a direct interface with the PCI bus and the X-Bus. The X-Bus connects the 21285 chip to the Flash ROM, the LEDs, and the X-Bus Expansion as seen in Figure 9.

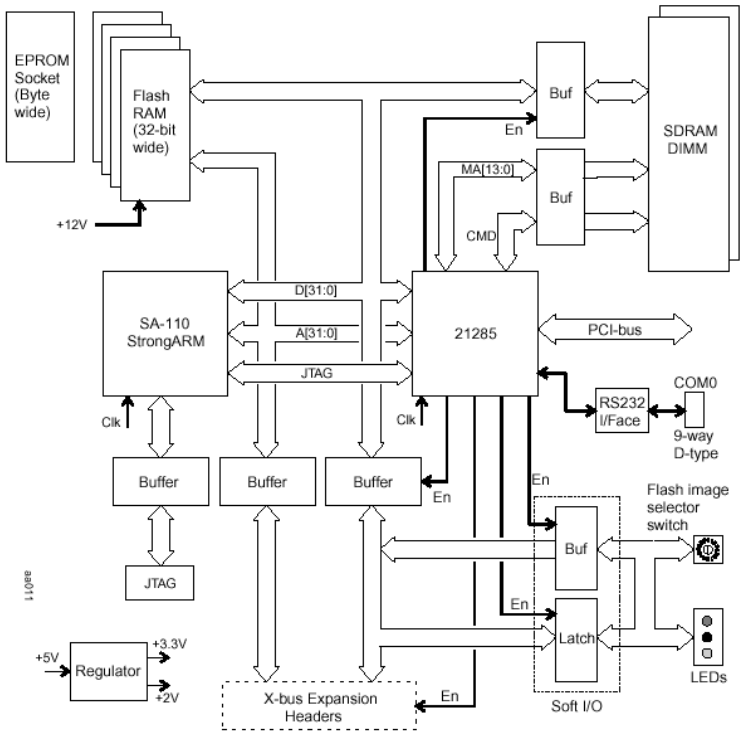

Figure 9 An Overview of the EBSA Board Architecture [7]

3.2 Board Configuration

In the project we have the EBSA in its second mode as a host bridge controlling the secondary PCI bus. Since the PCI and the X-Bus share some of the signal pins on the 21285 and those pins happen to be the ones used for arbitration functions on the PCI Interface the choice has to be made between the arbitration function and the functionality of the X-Bus. Although we download programs to the EBSA through the serial port, which is connected to the 21285 chip directly, we still need the flash ROM at boot time to load an image to the processor via the X-Bus. I decided to keep the arbitration function for the 21554 chip so we can use the X-Bus on board the EBSA.

The configuration is explained in details in the StrongARM EBSA-285 Evaluation Board Reference Manual section A1 [7] and Figure 10 shows the jumper setting for the host bridge mode with no arbitration function and an active X-Bus.

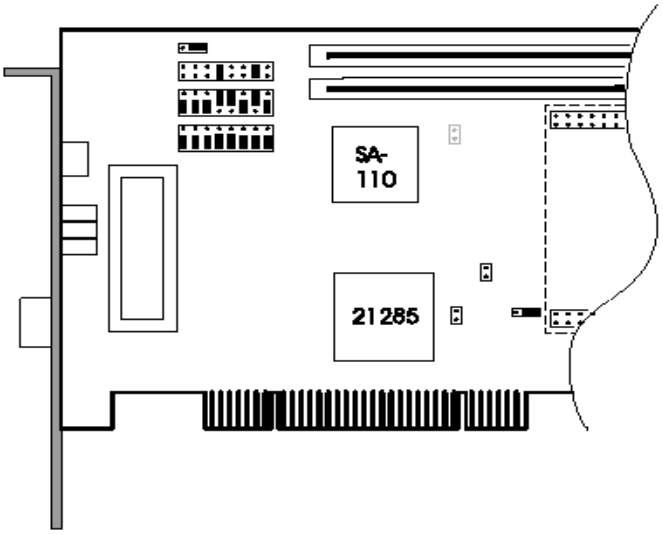

Figure 10 The EBSA board configured as a host bridge without arbitration [7]

3.3 Secondary PCI Bus Configuration

Intel provides a software package called uHAL Libraries and Angel Software Applications (SAAPPS) for StrongARM. They are both to help developers write code for access to the PCI via the 21285 core logic for the SA110. In this software package, code is provided to initialize the 21285 and to configure the PCI bus giving each device on the bus the resources requested.

3.3.1 UHAL and The SA Software Applications (SAAPPS)

The uHAL libraries and the SAAPPS can be downloaded from the Intel developer site at the following URL: http://developer.intel.com/design/strong/swsup/. Both provide example programs using those libraries and are well documented. I was able to download the example programs to the EBSA and run them using the STD and the Angel debugger.

3.3.2 Configuration Steps

To be able to configure the PCI, the bus host must build a device tree and then allocate resources requested to each device. To build a tree according to the uHAL library, the host generates configuration read cycles to the Vendor ID offset (configuration read with offset 00) of each possible PCI device using a range of device numbers, function numbers and Bus numbers. The bus number, device number and function number are sent to a function to check their validity against that range and are then used to create a device handle which is used to create the configuration address to read from the PCI device with those parameters. If a value was returned for the PCI configuration read, then a device with these particular numbers exists and its vendor ID is stored in a tree node along with its device handle.

After building the device tree, the PCI devices configuration takes place by reading the BAR of each device and allocating to it the requested address space as was explained in section 2.9.2.

3.3.3 21285 Configuration Write

In general to generate a PCI configuration address the high bits of the address are coupled with the IDSEL signal for each one of the devices on the board. By asserting only one bit at a time, the corresponding device will recognize it and complete the transaction. For example, device or slot number 0 will have its IDSEL on bit 11 of the configuration address and device number 1 is on bit 12 and so on. With this mechanism one might think you can put twenty devices on the same bus but due to the electrical loading, the maximum number of devices that could be placed on

one bus is ten or eleven devices. Keep in mind that a bridge places only one load on the system but can have more than one device behind it.

The configuration address generated by the StrongARM is different than the actual PCI address that goes on the bus. There are two mechanisms to generate PCI configuration addresses on the 21285 from the address sent to it by the StrongARM: Direct Mechanism and Decoding Mechanism.

The Direct Mechanism is used when the SA-110 address bits [23:22] are not equal to 11, or the SA-110 address is in PCI type 1 configuration space; then the PCI address is derived directly from the SA-110 address as shown in Table 6.

Table 6 Direct Mechanism for Generating PCI Addresses [2]

PCI Address Bits	Description
31:24	All bits are 0.
23:2	Equal to SA-110 address bits [23:2].
1:0	1:0 If the SA-110 address is in PCI type 0 configuration space, it equals 00. If the SA-110 address is in PCI type 1 configuration space, it equals 01.

The Decoding Mechanism is used when the SA-110 address bits [23:22] are equal to 11, and the SA-110 address is in PCI type 0 configuration space; then the PCI address is derived directly from the SA-110 address bits [15:11] as shown in Table 7.

Table 7 Decoding Mechanism for Generating PCI Addresses [2]

PCI Address Bits	Description	
31:11	Derived from the following decodes:	
	SA-110 A [15:11]	PCI ad [31:11]
	00000	00000000000000000001
	00001	00000000000000000010
	00010	00000000000000000100
	00011	00000000000000001000
	00100	00000000000000010000

PCI Address Bits	Description	
31:11	Derived from the following decodes:	
	SA-110 A [15:11]	PCI ad [31:11]
	00101	00000000000000100000
	00110	00000000000001000000
	00111	00000000000010000000
	01000	00000000000100000000
	01001	00000000001000000000
	01010	00000000010000000000
	01011	00000000100000000000
	01100	00000001000000000000
	01101	00000010000000000000
	01110	00000100000000000000
	01111	00001000000000000000
	10000	00010000000000000000
	10001	00100000000000000000
	10010	01000000000000000000
	10011	10000000000000000000
	10100	00000000000000000000
	10101 through 11111	00000000000000000000
10:2	Equal to SA-110 address bits [10:2]	
1:0	00.	

3.4 Change in the uHAL Configuration Write Code

As I mentioned before, the examples on the software libraries worked flawlessly except that they could not find any of the devices on the bus. The reason was the way the 21554's evaluation board is setup. There are four PCI slots on the board and the bridge itself, so there should be 5 different bits in the PCI address that correspond to each IDSEL of each device. On the board those device numbers are 13 for the system slot, 17 for the 21554 bridge, 18, 19, and 20 for the other three slots. The bit numbers on the configuration address corresponding to those device numbers are 24, 28, 29, 30, and 31 respectively. With the Direct Mechanism for generating PCI configuration addresses on the 21285, the high six bits [31:24] of the PCI address were set to 0 so by using this mechanism we would not be able to access the devices

on the evaluation board. To reach the devices on those slots the Decoding Mechanism has to be used to generate the appropriate IDSEL signals. In the uHAL library the only mechanism used to generate the PCI configuration address is the direct mechanism, which was not able to reach to the devices on the 21554 evaluation board. To be able to reach those devices I made some changes in the following uHAL library files:

1- uHAL\lib\ebsa285\platform.c

2- uHAL\h\ebsa285\platform.h

3- uHAL\h\pci.h

4- uHAL\lib\pci.c

In Platform.c there is a function named SAir_PciMakeConfigAddress(U32 handle, U32 offset) that takes the handle generated from the device, bus and function numbers and an offset into the configuration address space as arguments, and generates the configuration address for the 21285. The 21285 then takes the address and uses one of the two mechanisms discussed above to generate the PCI address. The library was set to check for devices numbered 0 to 12 and for that they did not need any access to the higher bits of the PCI address. To generate the address to access the IDSEL coupled with the higher bits of the PCI address, the following must be done: bits [23:22] of the address generated by the SA must be equal to 11; bits [15:11] must have the device number; and the max device number must be increased to 20. To implement that, I added a define statement (DC21285_PCI_TYPE_0_ CONFIG_22_23) in platform.h, changed the max device number in both pci.c and

pci.h to 21 and added a couple of lines to the SAir_PciMakeConfigAddress(U32 handle, U32 offset) function in platform.c. In Appendix A, I attached the code of parts of these files before and after the changes.

4 SCSI Interface

After deciding to port Linux to the EBSA and since Linux has a set of drivers for SCSI disks, the main goal for the SCSI interface development was to find the right adapter that is supported by Linux, and find the right disk for it.

4.1 Adapter Compatibility

Since a lot of the adapter manufacturers do not write drivers for Linux machines, not all adapters can be installed in Linux machines. On the Red Hat site there is a list of the compatible SCSI adapters that I used to get the right adapter. I also checked on the e-mail lists to find any problems that others were experiencing with some of the adapters. After some search I came to the conclusion that the Adaptec 2940U2W SCSI adapter was the best choice since it supports different SCSI standards including the SCSI Ultra2 Wide with up to 80 MB/s speed, can support up to 15 devices and has an external connection.

4.2 Disk compatibility

With the Adaptec 2940U2W adapter, any disk that had the half Pitch 68 pin connector was compatible. Again after some research and checking with some of the email lists, we decided on a Seagate Barracuda 18.2 GB SCSI hard drive.

4.3 Installation and Testing

Since we had a lot of connections and many boards in the instrumented computer (the computer with the 21554 Evaluation Board plugged into its PCI) we installed the SCSI disk in an external case and hooked it to the external connection of the Adaptec card.

To initially test the disk and the card I installed it in a different PC running windows NT, partitioned it and wrote to it a few files. We then installed it on a Linux machine and tested it there too. Both the disk and the adapter functioned correctly and the only test left was to see if it works with the ARM Linux on the EBSA. Since the EBSA is not running Linux yet, we could not verify that files transferred from NT to the EBSA will be readable by Linux. To get one step closer to the real system, with the help of Jim Fischer, we tested for this functionality by installing the SCSI on a Linux system, and writing .htm files to the SCSI disk on the Linux system from a Windows NT system over the network. After writing the .htm files to the SCSI on the Linux system we were able to open and view them using Netscape on Linux.

5 System Testing and Results

Bo Wu and Peter Huang developed a protocol to communicate between the host on Windows NT and the EBSA board [10]. They verified the functionality of the protocol using the EBSA board as an add-in card plugged into the primary PCI bus, without the nontransparent bridge. The program they used simply uses the mapmem functionality to write to an address in the SDRAM on the EBSA card and then the StrongARM reads that same address to verify the data transfer.

The program on the NT side uses the vendor ID of the EBSA board to identify it on the PCI bus and after finding its address on the PCI, it writes to its SDRAM. By adding the bridge in the middle and implementing the subsystem architecture as in Figure 1, and by changing the vendor ID from that of the EBSA to that of the 21554 bridge, the NT side now will write to the bridge. I wrote a program on the EBSA side using the uHAL libraries that would configure the secondary PCI, writes the address of the SDRAM on the EBSA to the translation register of the downstream memory used by the NT, and then reads a certain address that is agreed upon to find any data changes. The program I wrote for the EBSA side is attached in Appendix C

The data was successfully transferred from NT, over the primary PCI, through the bridge, over the Secondary PCI and to the SDRAM on the EBSA. The only obstacle we have is the caching problem.

The caching problem occurs because each time the StrongARM reads from an address in memory the data gets transferred to the cache. The second time StrongARM tries to read from the same address it does not get the value in memory but the cached value. So, if NT updates that value in the SDRAM, unless the cache is flushed the StrongARM will not see it.

6 Conclusion

I was able to accomplish my goal of integrating the Nontransparent PCI-to-PCI Bridge into the system without any change of the protocol developed for the communication between the host system and the EBSA-285. The data transfer from the host on the primary bus to the EBSA on the secondary bus was accomplished through the 21554. The 21554 successfully isolated the subsystem (EBSA card and other devices on the secondary bus) from the host system on the primary bus. The idea of an isolated system made it possible for the EBSA (as a host bridge) to be able to configure the secondary bus and all the devices connected to it, giving it full control of the whole system on the secondary bus.

The SCSI adapter and disk are both running successfully on a different Linux machine. They were both tested by: transferring files to the disk over the network from a WinNT machine, reading the files on the Linux machine and vise versa. The driver for the SCSI adapter is included in the ARM-Linux and should be tested as soon as the ARM-Linux is running on the EBSA-285.

There is one more issue that needs to be addressed, which is the EBSA-285 Memory Management Unit (MMU).

The EBSA-285 MMU is configured to have all the pages of the SDRAM memory cacheable. This means if a part of memory was read once and was

transferred to the cache, any memory reads to an address that lies in that cached part of memory will be read from the cache and not from the actual address unless the cache was flushed. The problem here is if the WinNT transfers data to the memory on the EBSA-285, the ARM processor might not be able to read it unless the cache was flushed. If parts of the memory were used as shared memory, those parts need not be cached and should be read straight from the memory. The MMU should then be configured to allow a part of memory to be not cacheable, in order to implement the shared memory.

Further work on the 21554 bridge could include: using other addressing modes to transfer data between the two interfaces, transferring interrupts over the bridge since the goal is to use interrupts instead of polling, and optimizing the bridge setting for best performance.

Further work on the EBSA-285 could include configuring the MMU to make the shared memory not cacheable.

Further work on the SCSI interface could include testing the SCSI adapter and the disk using ARM-Linux and transferring files between the two systems (NT and ARM-Linux) after the porting of ARM-Linux to the EBSA-285 board is complete.

7 Bibliography

[1] Anderson, Don and Shanely, Tom; PCI System Architecture, MindShare, INC.

[2] 21285 Core Logic for SA-110 Microprocessor Datasheet, Intel, September 1998

[3] 21554 PCI-to-PCI Bridge for Embedded Application, Hardware Reference Manual, Intel, September 1998

[4] 21554 PCI-to-PCI Bridge Evaluation Board User's Guide, Intel, February 2000

[5] Linux Kernel Readme files

[6] http://www.arm.uk.linux.org

[7] EBSA-285 Evaluation Manual, Intel, October 1998

[8] http://www.whatis.com

[9] Using a serial ROM with the 21554 Embedded PCI-to-PCI Bridge, Application Note, Intel, July 1998

[10] Wu, Bo, Network Performance Enhancement, Master thesis, Cal Poly San Luis Obispo, June 2000

Appendix A uHAL Library code changes

This appendix includes the changes I made to some of the files in the uHAL library. All the changes are in **bold**

Platform .h

I added the second define statement, masking bits [23:22] to use later in platform.c

```
#define DC21285_PCI_TYPE_0_CONFIG       0x7B000000
#define DC21285_PCI_TYPE_0_CONFIG_22_23 0x7BC00000
```

Platform.c

Using the #define in platform.h to generate the configuration address as explained in Table 4, I commented out the code that generated the address before my changes. This code is part of the U32 SAir_PciMakeConfigAddress(U32 devHandle, U32 offset) function.

```
/* Generating address for type0 configuration */
/*Was:
 * address = (U32) DC21285_PCI_TYPE_0_CONFIG ;
 * address |= (U32)(1 << (device + 11)) ;
*/
address = (U32) DC21285_PCI_TYPE_0_CONFIG_22_23 ;
address |= (U32)((device) << 11) ;
```

PCI.h

I increased the MAX_PCI_SLOT from 11 to 21

```
/* #define MAX_PCI_SLOT    11*/
#define MAX_PCI_SLOT    21
```

PCI.c

There is a function SAir_CheckCfgParams(U32 devHandle, U32 offset) that checks the parameters before it generates the address. I increased validity of the device number to 20.

```
/*Was:
 *if (device >= 12)*/
  if (device >= 21)
          return(MSG_ERROR_BAD_DEVICE) ;
```

Appendix B The 21554 Serial ROM Data File

This file comes with the Evaluation board of the 21554 (Srom.dat). I edited some of the entries to suit the desired 21554 configuration for the project. I also added some comments to clarify the registers' names being programmed. All the changes I made to the addresses and their assigned values are in bold.

After editing the file, the file is loaded from a dos prompt on the host of the machine with the 21554 evaluation board using the MKSROM dos utility provided with the board

The MKSROM utility has two uses:

- If used in the following format it will load the specified filename into the SROM of the 21554 board:

  MKSROM <filename>

- If used without a filename it will list the addresses and values already loaded in the SROM.

  MKSROM

```
; SROM data file. This file will be loaded into the SROM
; on the 21554 evaluation board
;
; Refer to the Intel Application note: Using a serial ROM
; with the 21554 Embedded PCI-to-PCI Bridge.[9]
; The semicolon (;) specifies a comment
; The open bracket ([) specifies the start of the file
; The close bracket (]) specifies the end of the file
```

```
; The colon (:) specifies a line of code

[
; preload enable (sets bit seven to preload enable)
:0 80
:1 00
:2 00
:3 00
; Primary class code
:4 00
:5 80
:6 06
; Subvendor IDs
:7 46
:8 00
:9 11
:A 10
; Primary Min GNT, Max Lat
:B 00
:C 00
; Secondary Class Code
:D 00
:E 80
:F 06
; Secondary Min GNT, Max Lat
:10 00
:11 00
; Downstream Mem0 (set to a 4K window for CSR only)
:12 00
:13 F0
:14 FF
:15 FF
; Downstream Mem1 or I/O(set 2MB memory window size
; and disable I/O window
:16 08
:17 00
:18 E0
:19 FF
; Downstream Mem2 (disabled)
:1A 00
:1B 00
:1C 00
:1D 00
; Downstream Mem3 (disabled)
:1E 00
```

```
:1F 00
:20 00
:21 00
; Downstream Mem3 Upper 32 bits for Flat addressing (DAC)
; (disabled)
:22 00
:23 00
:24 00
:25 00
; Expansion ROM (Set 1MB expansion ROM size)
:26 01
:27 F0
; Upstream Mem0 or I/O (disabled)
:28 00
:29 00
:2A 00
:2B 00
; Upstream Mem1 (Set 2MB memory window size)
:2C 08
:2D 00
:2E E0
:2F FF
; Chip control0
:30 00
; clear lockout bit
:31 00
; Chip control1
:32 00
; LUT disable, i2o disabled
:33 00
; Arbiter control (Sets internal 21554 secondary bus
; request to high priority ring
:34 00
:35 02
; System error disable
:36 00
:37 00
Power management
:38 00
:39 00
:3A 00
:3B 00
:3C 00
:3D 00
:3E 00
```

```
:3F 00
:40 00
:41 00
:42 00
]
```

Appendix C Testing Code Running on the EBSA

The following code is the printout of the pcicom.c file. I generated this file using one of the demo projects of the uHAL software package. The demo projects are located at uHAL\demo\ebsa285. I simply opened one of the projects, changed the source code, and changed the names of the source file and the project to pcicom.

```
#include "system.h"

#define Down_Mem1_Trans_Reg    0x98

/*============================================================================
 *
 *  routine:     main()
 *
 *  parameters:  not used
 *
 *  description: this routine will initialize the EBSA PCI interface, Build and print
 *               the PCI devices tree, Configure the PCI devices, Write the address of
 *               that memory into the translation Register of Downstream Memory 1 at
 *               offset 98h on the 21554 secondary interface and it will then polls
 *               the shared memory location to check for any changes that might have
 *               been made from the NT side.
 *
 *  calls:       SAr_BuildDevNodeTree(), SAr_PrintDevNodeTree(),
 *               SAir_PciInit(memSize, memBase), SAr_ConfigurePciDevices()
 *
 *  returns:     void
 */

int main ()
{
 U32    data, temp;
 /*U32  Mem_Current_Value, Mem_Prev_Value;*/
 U32    offset = 0x00000018;
 U8     bus, slot, func = 0 ;
 U32    handle_285, handle_554;
 S32    status ;


        /*initialize the EBSA PCI interface and dedicate 2Meg for the shared memory
        starting at the 8meg point in the SDRAM */

        SAir_PciInit(SZ_2M, SZ_8M);

        /*Build and print the PCI devices tree of the Secondary PCI Bus*/

        SAr_BuildDevNodeTree();
        SAr_PrintDevNodeTree() ;

        /*Configure the PCI devices and give them the requested resources*/
```

```
SAr_ConfigurePciDevices() ;

/*make  and print out the handle of the 21285 bus = 0, slot = 0,
and func = 0 since it is the host bridge of the Bus */

handle_285 = MAKE_HANDLE(PCI_TYPE, bus, slot, func) ;
SAr_printf("21285 handle is:     %02xh\n",handle_285);

/*make  and print out the handle of the 21554 bus = 0, slot = 17,
and func = 0 since it is connected to slot 17 of the Bus*/

slot = 17;
handle_554 = MAKE_HANDLE(PCI_TYPE, bus, slot, func) ;
SAr_printf("21554 handle is:     %02xh\n",handle_554);

/*Read SDRAM BAR on the 21285 offset 18h (Secondary Bus) and store it in
data*/

data = SAr_PciCfgReadLong(handle_285, PCI_DRAM_BAR , &status);
SAr_printf("PCI DRAM BAR is:     %02xh\n",data);

/*Write 8Meg into the SDRAM Base Address Offset Register */

*(U32*)(DC21285_ARMCSR_BASE + DRAM_BASE_ADDR_OFF) = 0x00800000;

/*Read the SDRAM Base Address Offset Register */

temp = *(U32*)(DC21285_ARMCSR_BASE + DRAM_BASE_ADDR_OFF);
SAr_printf("SDRAM Base Address Offset Register is:     %02xh\n",temp);

/* Read the SDRAM Base Address mask Register */

temp = *(U32*)(DC21285_ARMCSR_BASE + DRAM_BASE_ADDR_MASK);
SAr_printf("SDRAM Base Address Mask Register is:     %02xh\n",temp);


/* Write the address of that memory into the translation Register of
Downstream Memory 1 offset 98h*/

/*Read the contents of the translation register*/

temp = SAr_PciCfgReadLong(handle_554, Down_Mem1_Trans_Reg , &status);
SAr_printf("Contents of trans reg before write:     %02xh\n",temp);

/*write to the translation register and read it again to verify*/

SAr_PciCfgWriteLong(handle_554, Down_Mem1_Trans_Reg, data, &status);
temp = SAr_PciCfgReadLong(handle_554, Down_Mem1_Trans_Reg , &status);
SAr_printf("Contents of trans reg after write :     %02xh\n",temp);


/*The following while loop polls the memory location at address 0x00800000 and
prints its contents if it was changed.
since the loop reads the true value of the memory only once and then reads the
cached value I adjusted the code to debug for that by only reading the memory
when I enter a character from the keyboard*/

/*Mem_Prev_Value = Mem_Current_Value = 0xFFFFFFF;*/

while (1)
{
/*Mem_Current_Value = *(volatile U32*)(0x00800000);
*if (Mem_Current_Value != Mem_Prev_Value)
*{
*       SAr_printf("Previous Contents of Memory at address 0x00800000:
*       %02xh\n",Mem_Prev_Value);
```

```
        */
                SAr_printf("Current Contents of Memory  at address 0x00800000:
                %02xh\n",*(U32*)(0x00800000));
        /*}
        *Mem_Prev_Value = Mem_Current_Value;
        */

        printf("Before getchar\n");
        getchar ();
        printf("After getchar\n");
        }
}
```