KTH Information and Communication Technology

Implementing a SATA disk controller on a Virtex5 FPGA

Author: Berk Tuncer

Supervisor: Jan Andersson

Examiner: Professor Ahmed Hemani

Master of Science Thesis 2012

KTH Information and Communication Technologies

System on Chip Design

SE-100 44 STOCKHOLM

SAMMANFATTNING

Syftet med detta examensarbete var att utveckla en SATA-disk controller som följer med Intel AHCI-standarden. Den huvudsakliga funktionen hos AHCI standarden är att göra användaren kan interagera med den bifogade SATA hårddisken. Därför kan användaren skriva och läsa sektorer i disken genom mjukvara.

Den AHCI controller har kodats i VHDL och testades på en Xilinx ML505 FPGA. För att testa beteende mönstret har ett C-program kodats som sätter användaren specifika kommandon i systemminnet och meddelar användaren om svaren från SATA disken. Designen är testad med både SATA-1 och SATA-2-standarder och har visat sig att arbeta med var och en av dessa.

Regulatorn består av två olika delar som är främre regionen och AMBA AHB regionen. Den främre änden gränssnitt SATA hårddisk till AMBA domänen där den huvudsakliga statsmaskineriet finns. Operationen frekvenserna för dessa två domäner är också annorlunda. Den främre änden arbetar vid 150 MHz, medan AMBA domänen fungerar vid 80 MHz. Så dessa två domäner behöver klocksynkronisering att fungera varandra.

Designen har lindats till en enda kärna och kan förlängas med ytterligare funktioner i AHCI standarden som har lämnat ut till förmån för projektets genomförande.

ABSTRACT

The aim of this master thesis was to develop an SATA disk controller that complies with the Intel AHCI standard. The main operation of the AHCI standard is to make the user capable of interacting with the attached SATA hard disk. Hence the user can write and read sectors in the disk through software.

The AHCI controller has been coded in VHDL and tested on a Xilinx ML505 FPGA. To test the behavior of the design, a C program has been coded which puts the user specific commands into the system memory and notifies the user of the responses of the SATA disk. The design is tested with both SATA-1 and SATA-2 standards and is proven to be working with each of these.

The controller consists of two different parts which are the front end region and the AMBA AHB region. The front end interfaces the SATA disk to the AMBA domain where the main state machine resides. The operation frequencies of these two domains are also different. The front end operates at 150 MHz, whereas the AMBA domain is functioning at 80 MHz. So, these two domains need clock synchronization to operate mutually.

The design has been wrapped into a single core and can be extended with additional features of the AHCI standard that have been left out for the sake of completion of the project.

FOREWORD

The master thesis was proposed by Jiri Gaisler and it was conducted in Aeroflex Gaisler in Gothenburg. This project report was written for the Information and Communication Technologies Departmant of KTH.

I would like to thank to Jiri Gaisler for making it possible for me to perform this project. Specially, I want to thank Jan Andersson for aiding me thoroughly with great patience throughout the project. Last but not least, I would like to thank my mother father and aunt, Zerrin, Nural and Nurten Tuncer, for their most valuable support during the entire project.

Berk Tuncer

Gothenburg, October, 2012

NOMENCLATURE

Abbreviations

AHCI

It is the acronym for “Advanced Host Controller Interface”. This is the Intel standard that specifies the flow of communication between the controller and the SATA device.

Command list

Commands located in the system memory that the controller processes. It has 32 free slots.

Command slot

It contains the command to execute in the command list.

D2H

It is the acronym for “Device to HBA”. D2H is used for stating that the direction of the transfer of a FIS is from the SATA device to the controller.

Device

This keyword is used to describe a physical device like a SATA hard disk which is attached to a port on the FPGA.

FIS

It is the acronym for “Frame Information Structure”. It is a packet of information that is transferred between the controller and the device.

FPGA

It is the acronym for “Field Programmable Gate Array”. This is the device on which the design is implemented and tested.

H2D

It is the acronym for “HBA to device”. H2D is used for stating that the direction of the transfer of a FIS is from the controller to the SATA device.

HBA

It is the acronym for “Host Bus Adapter”. HBA refers to the silicon that implements the AHCI specification to communicate between system memory and SATA devices.

PRD

It is the acronym for “Physical Region Descriptor”. PRD entries describe the location and length of data to be transferred.

System memory

This is the main memory which is used by the controller. Commands and received data from the SATA device are being stored here.

DW

This is the abbreviation for double word. It is a structure of 32 bits.

TABLE OF CONTENTS

TABLE OF FIGURES

TABLE OF TABLES

1 INTRODUCTION

1.1 Aeroflex Gaisler

Aeroflex Gaisler develops and supports IP cores coded in VHDL. These IP s are integrated into a library called GRLIB, which is available as open-source. The library includes the LEON3 SPARC V8 processor, on which the software designed in this project, runs also. Other than that, the library includes IP cores such as Ethernet and DDR memory controller interfaces and PCI, USB host/device controllers. The communication between the GRLIB IP cores is obtained according to the AMBA bus specification [1].

1.2 Master Thesis Proposal

The project was proposed by Aeroflex Gaisler and it should include the development of a SATA controller which is to be generated in VHDL following the AHCI standard developed by Intel [5]. The proposal also includes that this controller has to be verified with a software application and then to be integrated as an IP core to GRLIB.

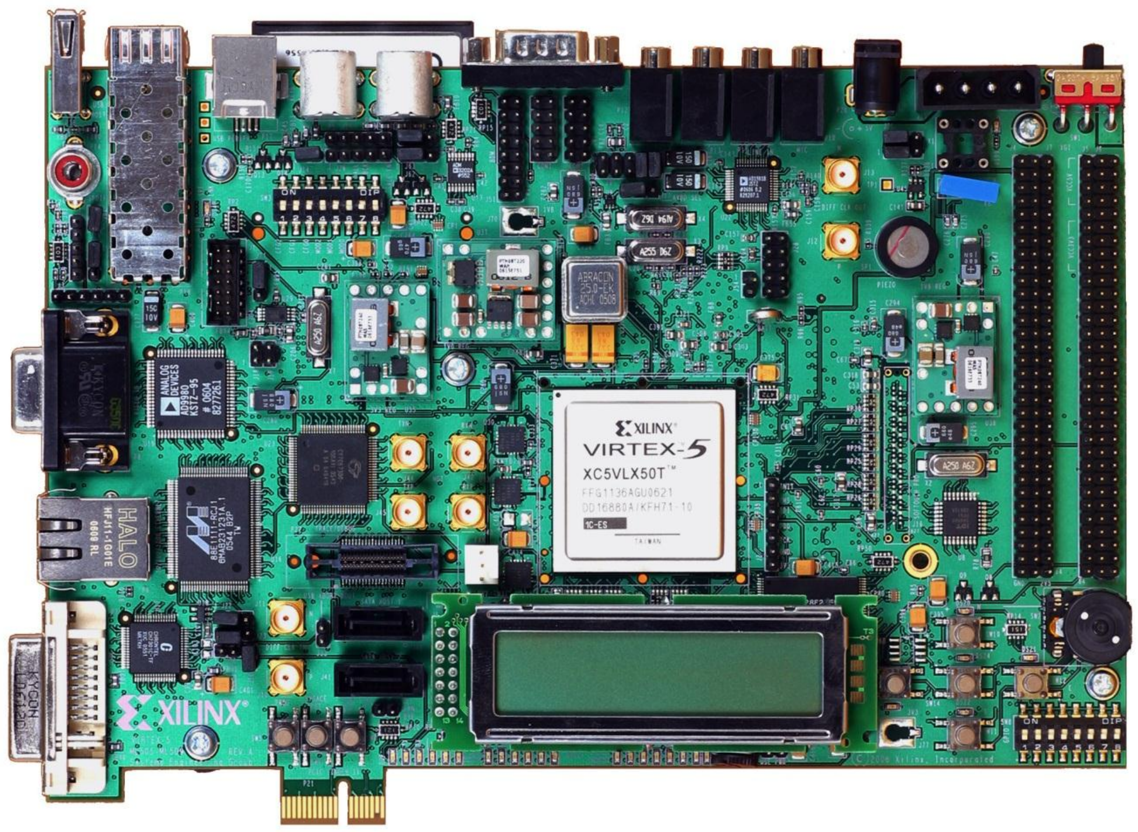


Figure 1: Xilinx ML505 board

The SATA controller will utilize the AMBA bus specification for interconnecting the designed components. The whole design has to be realized on a Xilinx ML505 board which can be seen in Figure 1. The final design has to be in a working shape so that drivers for Linux 2.6 can be generated on top of it.

The ML505 board can realize a full Leon3-based system including the required peripherals. It also has two connector ports to which SATA disks can be attached using a SATA cable. However these are just for carrying data and the power for the SATA devices has to be provided by external supplies.

1.3 Goal and structure of the project

The purpose of this project is to realize the AHCI Controller on the Xilinx ML505 board and test the communication with a SATA device through a C program. The AHCI Controller is realized in VHDL and simulation is carried out in Modelsim. Later on, synthesis is performed in Xilinx XST tool and the design is tested using GRMON, which is developed by Aeroflex Gaisler. This tool allows the user to monitor the values of the registers and the memory of the design. Hence it could be monitored whether a FIS is correctly received from the device.

As far as the timing goals are concerned, the design is expected to run at 80 MHz, which is the speed of the AHB domain. The SATA device domain operates at 150 MHz. However since for the completion of a transfer both domains have to be involved, the whole system could only operate at 80 MHz. Hence the AHB domain is the bottleneck of the system in terms of the speed.

Regarding the area goals, no solid specification has been set apart from that the design has to fit into the FPGA of course. Other than that, an efficient coding style of VHDL was tried to be pursued in order not to infer unnecessary components.

The system has 6 main components which are namely the Leon3 processor, DMA controller, AHCI Registers, AHCI controller, DDR2 memory and the front end interface. DMA controller, AHCI controller and AHCI Registers have been developed in this project. Leon3 processor and DDR2 memory controller was designed by Aeroflex Gaisler. Front end interface was designed by the previous master thesis work. The project utilizes the AMBA AHB and APB buses for connecting the components. There are two masters in the system. First one is Leon3 processor. It is the component on which the C program runs. The second master is the DMA controller, which handles the writes/reads to/from system memory. DDR2 memory acts as the system memory which is interfaced to the design through a memory controller component. AHCI Registers are flags which indicate the status of the controller in different times during the operation. These Registers are both utilized by the hardware (AHCI controller) and the software (running on the Leon3 processor). Finally, the AHCI controller is the component where the main state machine resides. It can be depicted as the brain of the whole system.

1.4 Demarcations

Since the project had too many features to be implemented, some limitations had to be set in order for a timely completion. The demarcation phase was decided to be initiated after the research for the project was completed. In this way, a better grasp of the AHCI standard could be obtained. Hence a more realistic plan for the achievements on the project could be outlined.

After researching the AHCI standard, it was decided that the priorities should be focused on three components. These are the AHCI Registers, the DMA Controller and the AHCI Controller. As moved on the next phases of the project, it was going to be discovered that the Front end, which was the product of a previous master thesis, also needed some modification to be integrated to the project. However in this phase, the key components were the three mentioned above. No demarcations have been made on the DMA Controller and the AHCI Registers, since the operations of these should have been carried out flawlessly for the sake of the project. In the AHCI Controller however, states which were responsible from interrupt generation and error recovery had been left out. These were additional features and the project could still function correctly in the absence of these. Taken this into account, there was no harm in constraining the goals of the project towards these states. Apart from that, Linux driver support had been left out but could be implemented any time later, since it also does not prevent the controller from working.

Luckily, a front end which encapsulates the necessary layers was designed by previous master thesis works. These layers were utilized to interface the SATA disk to the AHCI controller. These three layers, combined under the framework of the front end, are depicted in Figure 2 and are namely the Transport layer, Link layer and Physical layer. The link and physical layers were the production of a master thesis work in 2008, whereas the Transport layer and the front end top level module which encapsulates these three were designed in a master thesis work in 2010.

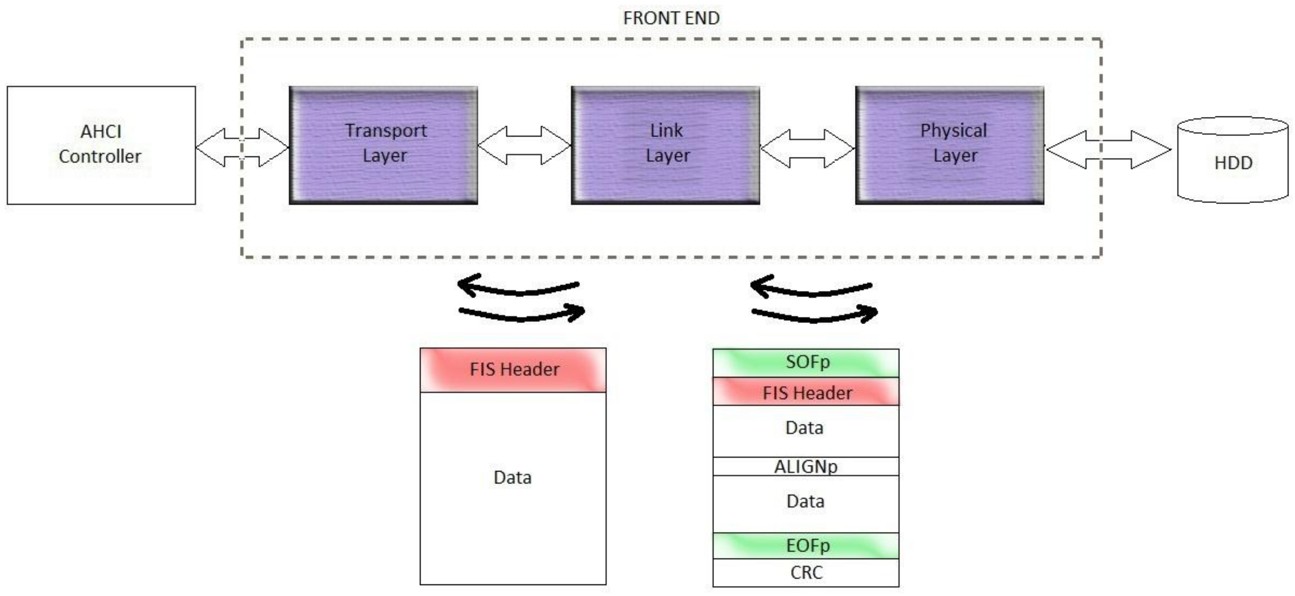


Figure 2: Front End

Although some changes had to be performed in the transport layer and the front end top module to meet the timing requirements and to obtain a flawless interface to the AHCI controller, the previous works made a great contribution to this thesis.

The physical layer is a Xilinx generated component and is the lowest layer of communication. The physical layer is significant because of two aspects. First, it is the layer that communicates directly with the SATA disk. Second, it handles the clock generation for the SATA domain and makes the SATA clock (150 MHz) available for the usage of transport and link layers. The physical layer uses a SERDES (Serialize – Deserialize) component to interact with the SATA disk. This component is also called GTP dual transceiver and handles data transfers using differential pulses. The GTP dual component is instantiated in the physical layer has both a receiver port and a transmitter port. These ports are mapped to the SATA connectors on the Xilinx ML505 board and provide the communication with the SATA disk. Regarding the clocking circuitry, the physical layer uses IBUF components from Xilinx library. Through this component, the 150 MHz clock coming from the board is distributed to the GTP dual component and to the rest of the design including the transport and link layers.

The ML505 board is capable of producing a 75 MHz clock also, but since the SATA disk supports 150 MHz, this was the chosen frequency.

The link layer communicates with the physical layer using the so called primitives. A list of the primitives can be seen in Table 1. Each of these primitives has different meanings, but they are mainly responsible to flag the start and end of the transfers. The data payload starts to be transferred after the link or the SATA device issues a SOF primitive depending on the direction of the transfer. This primitive marks the start of a transfer frame. A frame ends when an EOF primitive is issued by the sending side. These primitives are not to be confused with the FIS es, which are the main transfer blocks in this project. The FIS es also have different types and in this hierarchy, all of the FIS es are the data payloads, which are being sent between the SOF and EOF primitives.

Primitive Name	Hex Value	Meaning
ALIGNp	7B4A4ABC	Receiver has to align its data reception
CONTp	9999AA7C	Continue to receive the previous primitive
DMATp	3636B57C	Request the DMA to terminate a continuing transfer early
EOFp	D5D5B57C	Indicates the end of a transfer
HOLDp	D5D5AA7C	Request for a pause in a running transmission
HOLDAp	9595AA7C	Issued while HOLDp is received
R_ERRp	5656B57C	Received if errors occur in the last transmission
R_IPp	5555B57C	Reception is ongoing
R_OKp	3535B57C	Data transfer is completed without errors
R_RDYp	4A4A957C	Ready to receive data
SOFp	3737B57C	Indicates the start of a transfer
SYNCp	B5B5957C	Synchronization primitive
WTRMp	5858B57C	Sent after the reception of the EOFp while waiting for reception status from the receiver
X_RDYp	5757B57C	Ready to start data transfer

Table 1: Primitive Types

The transport layer's main responsibility is to act as a bridge between the AHCI controller and the SATA device. It collects FIS es from both sides to its internal FIFOs and transfers these. Different FIS types can be seen in Table 2. This layer is extremely significant since clock synchronization is also handled here. The FIFOs are asynchronous and work with both clock domains. For control signals which are needed for the initiation and finishing of the transfers, two flip-flop method is used. This layer had to be modified in this thesis work in order to perform a correct interfacing to the AHCI controller.

FIS Type Hex Value	Explanation
27h	Register FIS – Host to Device
34h	Register FIS – Device to Host
39h	DMA Active FIS – Device to Host
41h	DMA Setup FIS – Bidirectional
46h	Data FIS – Bidirectional
58h	BIST Activate FIS – Bidirectional
5Fh	PIO Setup FIS – Device to Host
A1h	Set Device Bits FIS – Device to Host

Table 2: FIS Types

Finally, the front end top module was the level where these three layers were instantiated. It provides a complete black box where it could be taken and connected to the AHCI controller to perform the necessary communication with the SATA disk. However in its given shape it did not meet timing. Hence some modifications had to be done to obtain error free traffic between the AHCI controller and the SATA disk.

3 THEORY OF OPERATION

3.1 Working Principle of AHCI

AHCI is a standard for controlling SATA hard disks through software applications. The general theory of operation is that the software produces commands and puts these into the system memory. The designed AHCI controller fetches and sends these over a communication link to the attached hard disk device. Based on the type of the command issued, the disk responds accordingly. A DMA engine is vital for the communication with the system memory. Also, registers specific to the AHCI standard have to be implemented for being able to communicate with the software.

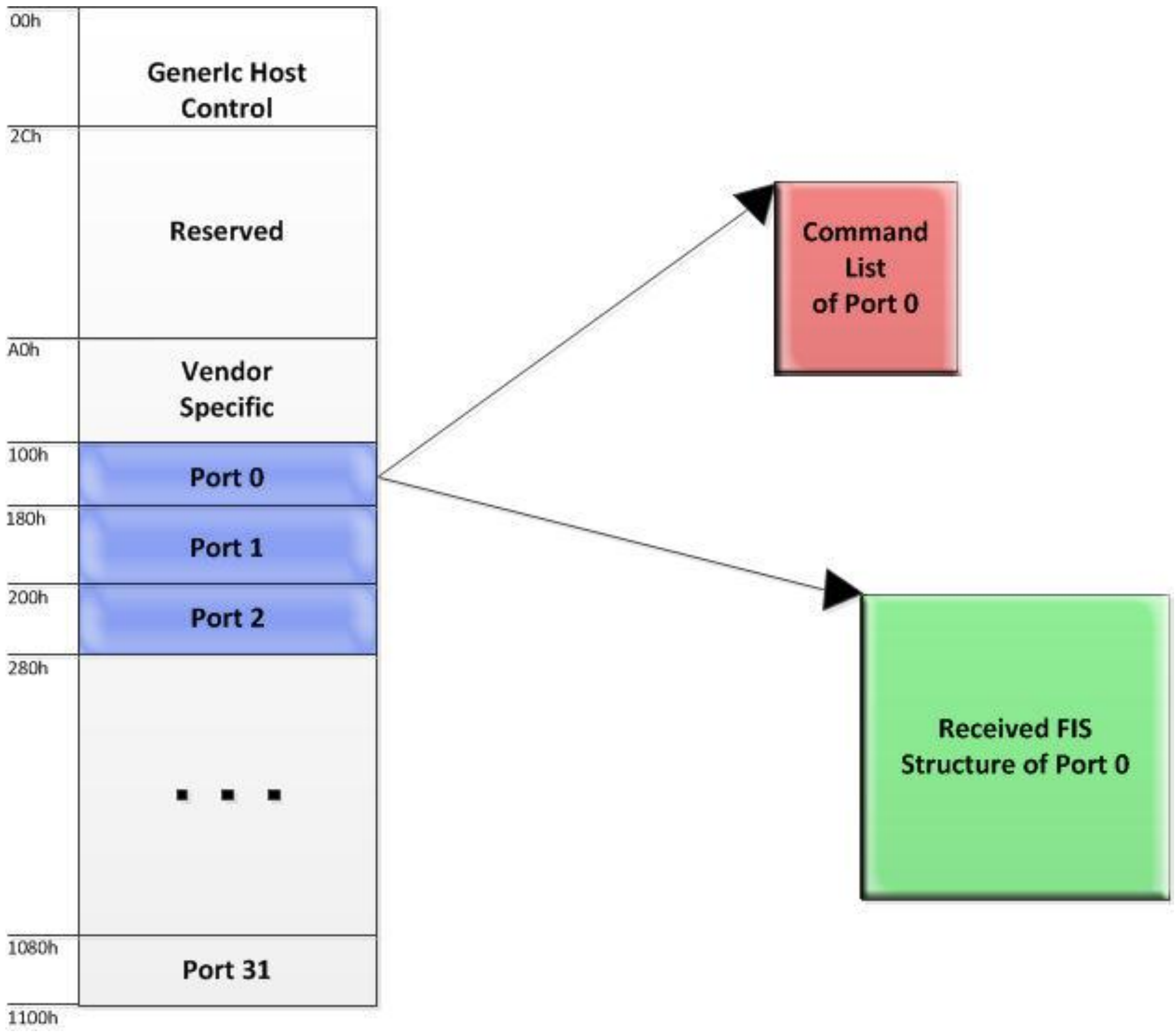


Figure 3: HBA Registers

Figure 5 shows the HBA registers. These are specified by the AHCI standard. The registers can be grouped into two major groups which are namely the global and the port registers. The global registers identify the general properties of the system, whereas the port registers are related to the attached SATA disks. Each port register is responsible from a single hard disk. Detailed map information about the global and port registers can be found in Tables 3 and 4 respectively.

Address Range	Abbreviation	Explanation
00h – 03h	CAP	Host Capabilities
04h – 07h	GHC	Global Host Control
08h – 0Bh	IS	Interrupt Status
0Ch – 0Fh	PI	Ports Implemented
10h – 13h	VS	Version
14h – 17h	CCC_CTL	Command Completion Coalescing Control
18h – 1Bh	CCC_PORTS	Command Completion Coalescing Ports
1Ch – 1Fh	EM_LOC	Enclosure Management Location
20h – 23h	EM_CTL	Enclosure Management Control
24h – 27h	CAP2	Host Capabilities Extended
28h – 2Bh	BOHC	BIOS / OS Handoff Control and Status

Table 3: Global Registers

It is visible in the figure that each port points to two different locations in the system memory. The one highlighted in green in Figure 5 is the Received FIS region, where the register FIS es are put after a successful reception from the SATA Disk. As an example to the types of FIS es copied to this region, “Device to Host” FIS can be stated. The end of each successful transfer between the AHCI Controller and the SATA disk is marked with a Device to Host Register FIS. This FIS is then copied to the system memory by the AHCI Controller using the DMA Engine. Also during the transfer, FIS es like “DMA Activate” are being received from the SATA disk, which are also copied to this portion of the system memory. For a full list of FIS es related to this section, please consult the Serial ATA AHCI manual [5].

Address Range	Abbreviation	Explanation
00h – 03h	PxCLB	Port x Command List Base Address
04h – 07h	PxCLBU	Port x Command List Base Address Upper 32-Bits
08h – 0Bh	PxFB	Port x FIS Base Address
0Ch – 0Fh	PxFBU	Port x FIS Base Address Upper 32-Bits
10h – 13h	PxIS	Port x Interrupt Status

14h – 17h	PxIE	Port x Interrupt Enable
18h – 1Bh	PxCMD	Port x Command and Status
1Ch – 1Fh	Reserved	Reserved
20h – 23h	PxTFD	Port x Task File Data
24h – 27h	PxSIG	Port x Signature
28h – 2Bh	PxSSTS	Port x Serial ATA Status (SCR0: SStatus)
2Ch – 2Fh	PxSCTL	Port x Serial ATA Control (SCR2: SControl)
30h – 33h	PxSERR	Port x Serial ATA Error (SCR1: SError)
34h – 37h	PxSACT	Port x Serial ATA Active (SCR3: SActive)
38h – 3Bh	PxCI	Port x Command Issue
3Ch – 3Fh	PxSNTF	Port x Serial ATA Notification (SCR4: SNotification)
40h – 43h	PxFBS	Port x FIS-based Switching Control
44h – 6Fh	Reserved	Reserved
70h – 7Fh	PxVS	Port x Vendor Specific

Table 4: Port Registers

The one highlighted in red in Figure 5 is the command list, where the command headers reside. These headers hold the base address of the command tables in system memory. The addresses contained by these headers are given to the DMA engine, which fetches the Command Tables from the system memory. Figure 8 shows a Command Table. Each command list can host 32 command headers and hence 32 command tables. For each command, there exists exactly one command table. As a result of this, 32 commands can be executed during the operation of the AHCI Controller.

Command FIS is highlighted in green in Figure 8. This is the most vital section in the command table, since the SATA disk reacts upon the reception of this from the AHCI Controller. The CFIS indicates the type of command, base address of the sector in SATA disk, from/to which the transfer to be issued and the number of sectors which are involved in the transaction. A list of the commands which are used in this project can be seen in Table 5. For a full list of commands that are usable depending on the features of the AHCI Controller, please consult to the ATA / ATAPI Command Set manual [6].

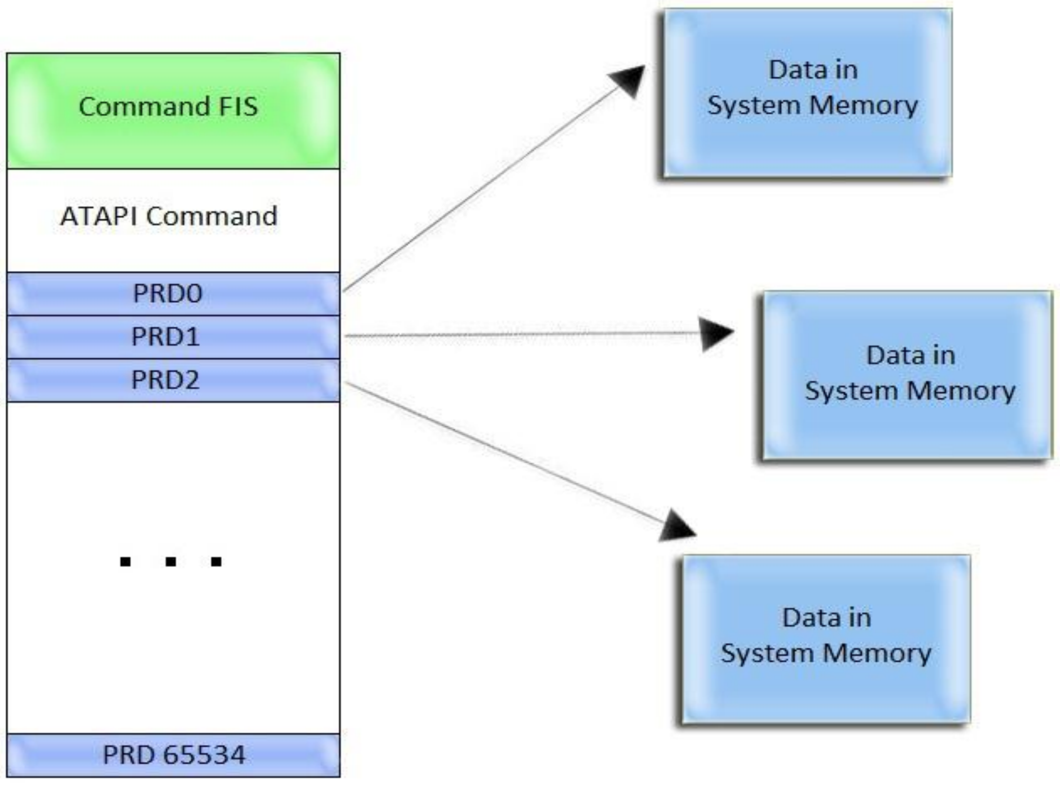


Figure 4: CFIS and PRD Table

Command Name	Explanation
IDENTIFY DEVICE	Returns the Model Number, Serial Number and the Firmware Version of the SATA disk
WRITE DMA EXT	Writes data from system memory to the SATA disk. The amount and disk address of data are also specified with this command.
READ DMA EXT	Reads data from SATA disk to the system memory. The amount and disk address of data are also specified with this command.

Table 5: Various Command Types

PRDs are residing inside the command table and are highlighted in blue in Figure 8. As CFIS es serve to the SATA disk for understanding the amount and the address of the transfer, PRDs are fulfilling the same purpose for the AHCI controller. During a "READ DMA EXT" command, the data read from the SATA disk according to the CFIS goes to the location in the system memory according to the address specified in these PRD s. Similarly during a "WRITE DMA EXT" command, the data to be written is fetched from the system memory according to the base address stated in these PRD s. Then it is transferred to the SATA disk, which writes it to the locations as specified in the CFIS.

3.2 Transfer Examples

This section plots how various transfers evolve during the operation of the software on the AHCI Controller. Each section assumes that the required command is built in system memory through the software with all of the required components such as Command Headers, Command Tables and PRD s.

The key fragments of the discussed transfers are FIS es, since every piece of information, including data, are encapsulated and transferred between the two sides utilizing these structures. Different types of these FIS es can be seen in Table 1 outlined in the related work section.

3.2.1 Identify Device

The flow of the "Identify Device" Command can be seen in Figure 10. This command is issued by the AHCI Controller in order to obtain the serial number, the firmware revision and the model number of the attached SATA disk to the board. It is a vital command for debugging also since the obtained values can be compared against the label on the SATA device. Hence it can be understood whether the board to SATA interface is functioning correctly and a healthy link could be established between the AHCI Controller and the SATA disk.

The "IDENTIFY DEVICE" command resides in the CFIS region in the system memory as shown in Figure 8. The AHCI controller fetches this command and issues it over the communication link to the SATA disk. Upon reception of this command, the SATA disk looks at the FIS type section of the command and sorts it as a "IDENTIFY DEVICE" command. According to the SATA standard, the device has to send two FIS es in succession upon reception of this command. The first FIS issued by the disk is a PIO Setup FIS. The length of this FIS is 5 DW and it also provides information about the size of the successive Data FIS. The 4th DW of the PIO Setup FIS indicates the length of the following Data FIS as 128 DW s. This corresponds to one sector of data as far as the SATA disk is concerned.

Indeed, the next FIS sent from the SATA disk is a Data FIS, which contains the model number, the serial number and the firmware revision of the attached SATA device. After this FIS has been received, the AHCI controller returns to an idle state and is ready to fetch and issue the next command header present in the command list. A command usually ends when the AHCI received a "Device to Host Register FIS" from the SATA device. However the "Identify Device" command is a special case, where the transfer of this last FIS is absent.

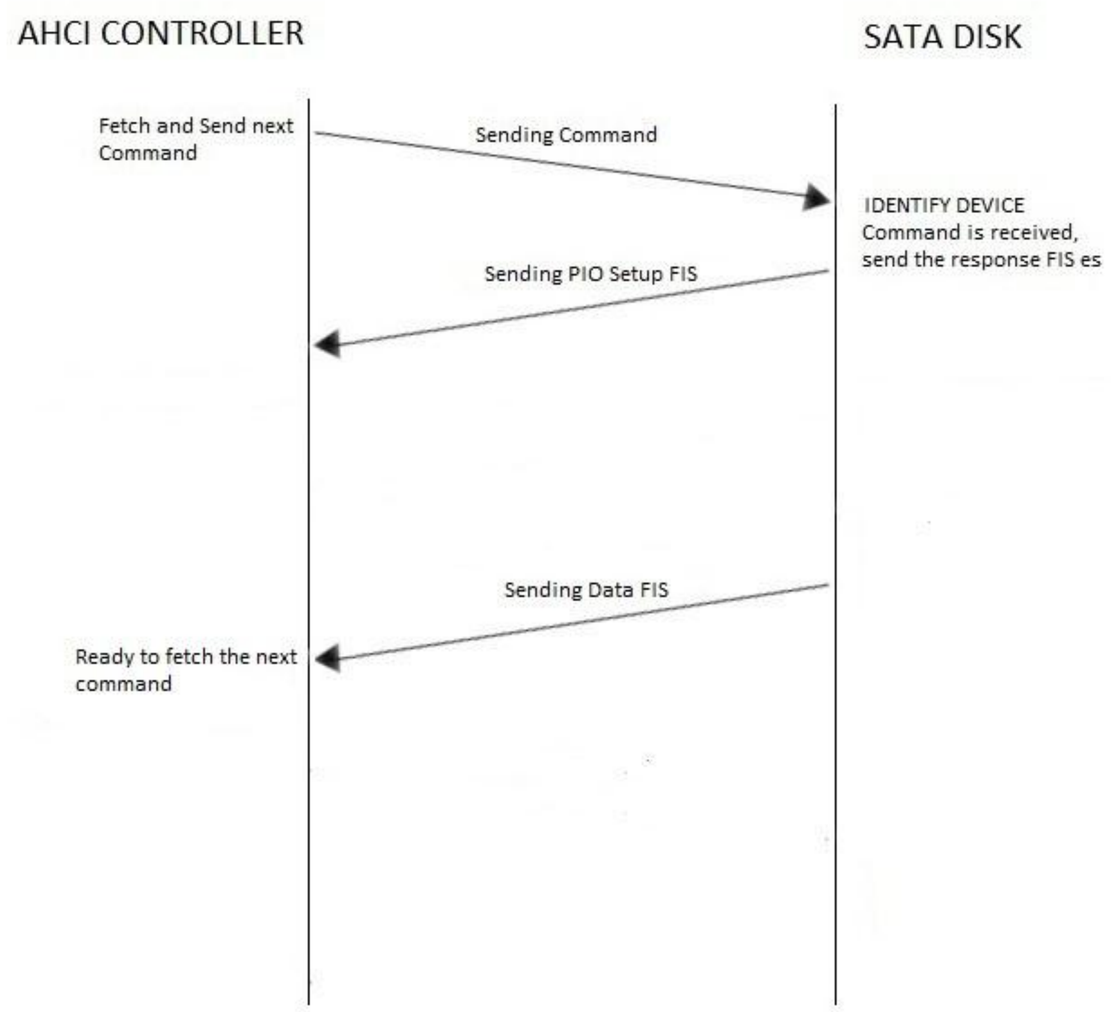

AHCI CONTROLLER
SATA DISK
Fetch and Send next Command
Sending Command
IDENTIFY DEVICE Command is received, send the response FIS es
Sending PIO Setup FIS
Sending Data FIS
Ready to fetch the next command

Figure 5: Identify Device

3.2.2 Write Transfer

The outline of a writing operation to the SATA device can be observed in Figure 11. The type of command needs to be sent to the device is “WRITE DMA EXT” for this operation. This command resides in the CFIS segment of the system memory as shown in Figure 8. The AHCI Controller fetches this command and issues it to the device over the communication link. When the SATA disk receives it and understands that it is “WRITE DMA EXT” command, it responds by sending a “DMA Activate FIS” back to the AHCI Controller. After receiving this FIS, the AHCI Controller starts to fetch the required data from the address stated in the next available PRD. The controller then encapsulates this chunk of data so that it forms a Data FIS and sends this structure to the disk. Initially in the Command FIS, the length of this transfer has been defined for the SATA disk. Hence, if the amount of data sent by the controller satisfies this length, the device issues a “D2H Register FIS” back to the controller, indicating that the transfer has been completed successfully. If not, the disk issues another

"DMA Activate FIS", requesting another Data FIS from the controller. Upon reception of the "D2H Register FIS", the AHCI Controller return to idle state and prepares for the execution of the next command.

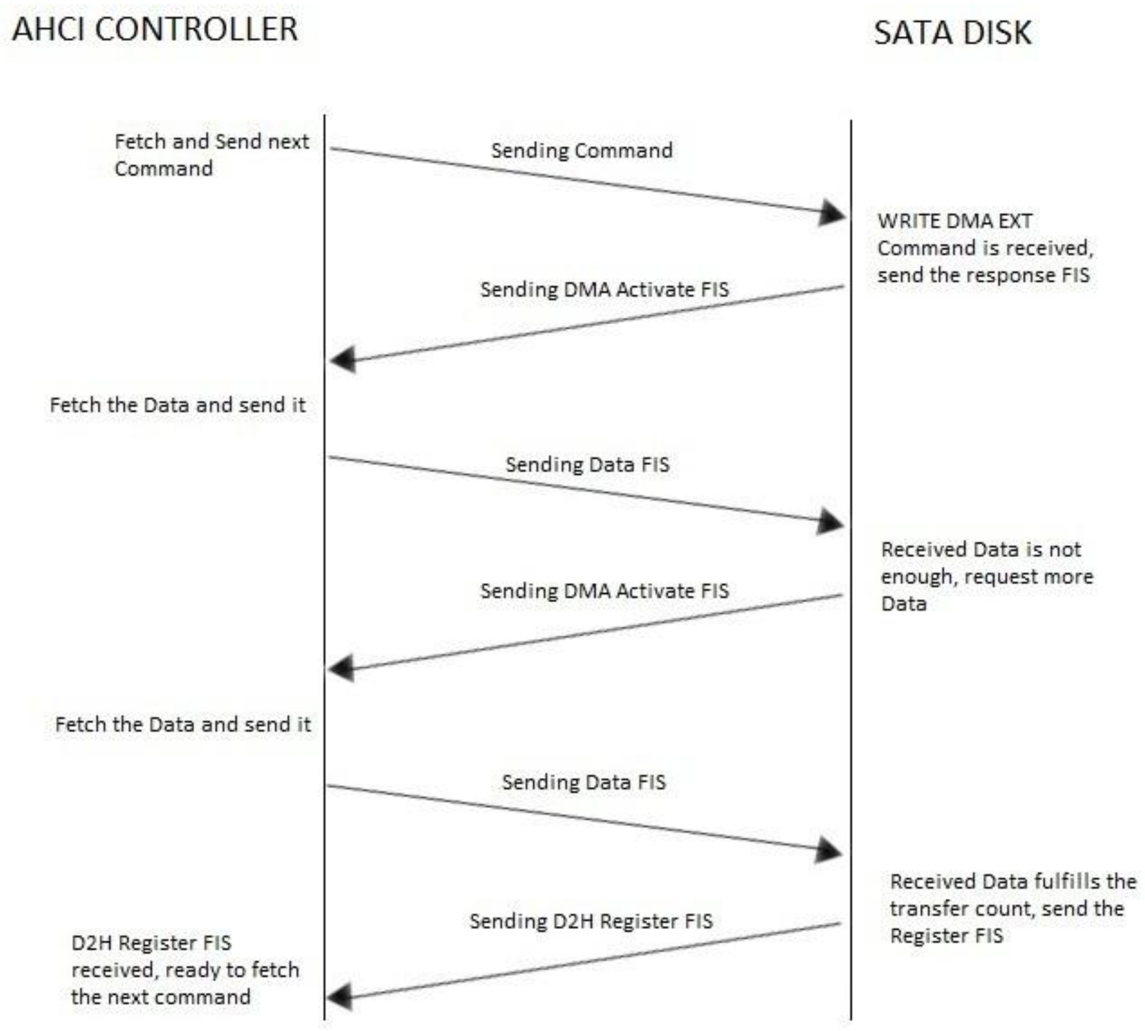

AHCI CONTROLLER
SATA DISK
Fetch and Send next Command
Sending Command
WRITE DMA EXT Command is received, send the response FIS
Sending DMA Activate FIS
Fetch the Data and send it
Sending Data FIS
Received Data is not enough, request more Data
Sending DMA Activate FIS
Fetch the Data and send it
Sending Data FIS
Received Data fulfills the transfer count, send the Register FIS
Sending D2H Register FIS
D2H Register FIS received, ready to fetch the next command

Figure 6: Write Transfer to the SATA Disk

3.2.3 Read Transfer

The flow of a read operation from the SATA disk can be examined in Figure 12. The type of the command required for this operation is "READ DMA EXT". This command resides in the CFIS section of the system memory as outlined in Figure 8. The AHCI Controller fetches and transfers this command to the SATA device. Once the disk understands that it is a "READ DMA EXT" command, it sends the data from the disk sector which is stated in the command. The length of the transfer is also stated in the command. Hence if the initiated Data FIS transfer does not satisfy the required data amount, a new chunk of data is fetched from the disk and sent to the AHCI Controller over the communication link. The successive Data FIS is fetched from the next available sector. For example if the controller wants 1024 bytes of data starting from the first sector of the disk, the SATA device transmits the first sector and then then the second sector. Each sector contains 512 bytes of data. Once the transfer count is satisfied, a “D2H Register FIS” is issued by the SATA disk. The AHCI Controller receives this and interprets it as a successful transfer. Then it is ready to fetch and execute the next available command header in the command list.

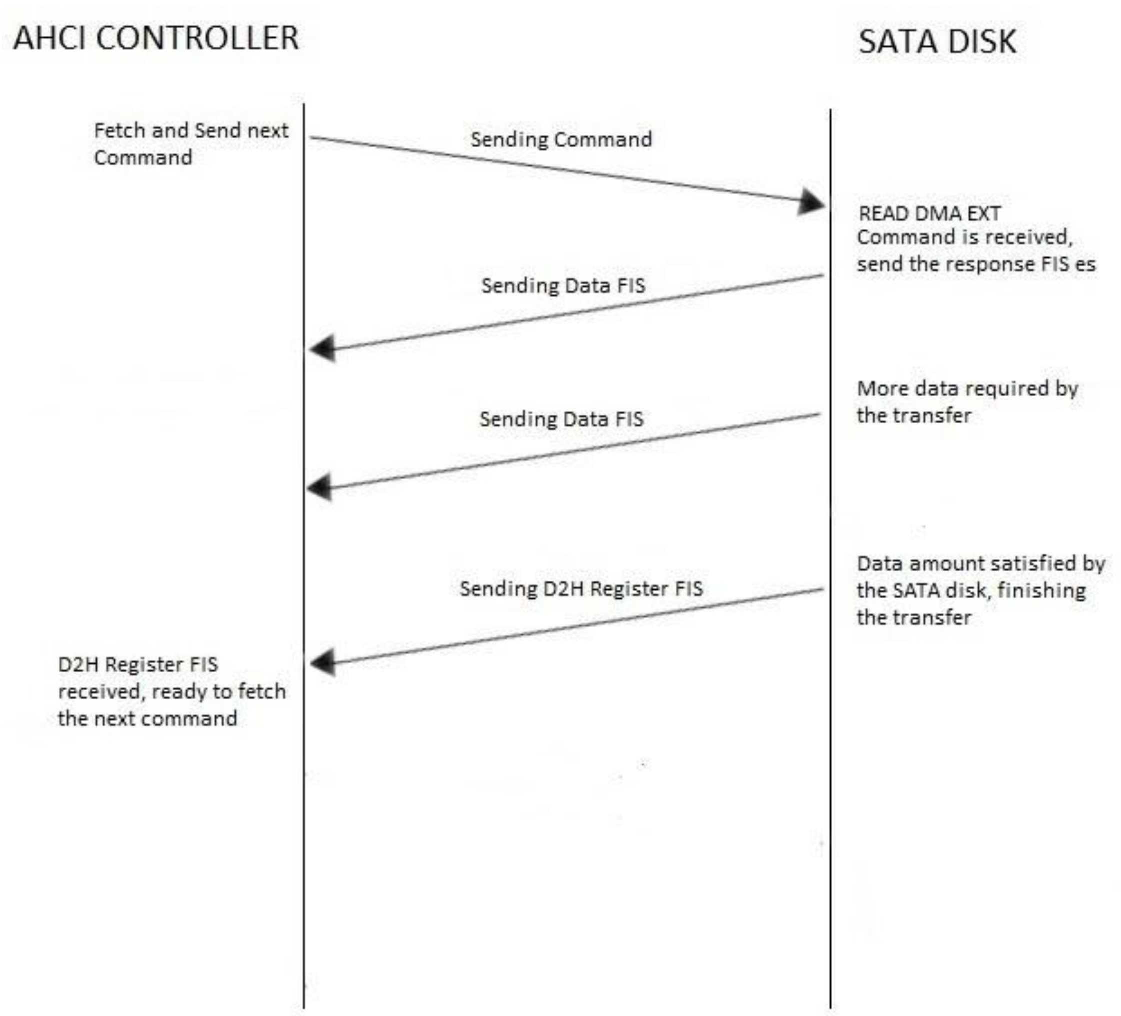

AHCI CONTROLLER
SATA DISK
Fetch and Send next Command
Sending Command
READ DMA EXT Command is received, send the response FIS es
Sending Data FIS
More data required by the transfer
Sending Data FIS
Data amount satisfied by the SATA disk, finishing the transfer
Sending D2H Register FIS
D2H Register FIS received, ready to fetch the next command

Figure 7: Read DMA Command

4 IMPLEMENTATION

4.1 Planning Phase

4.1.1 Overall System

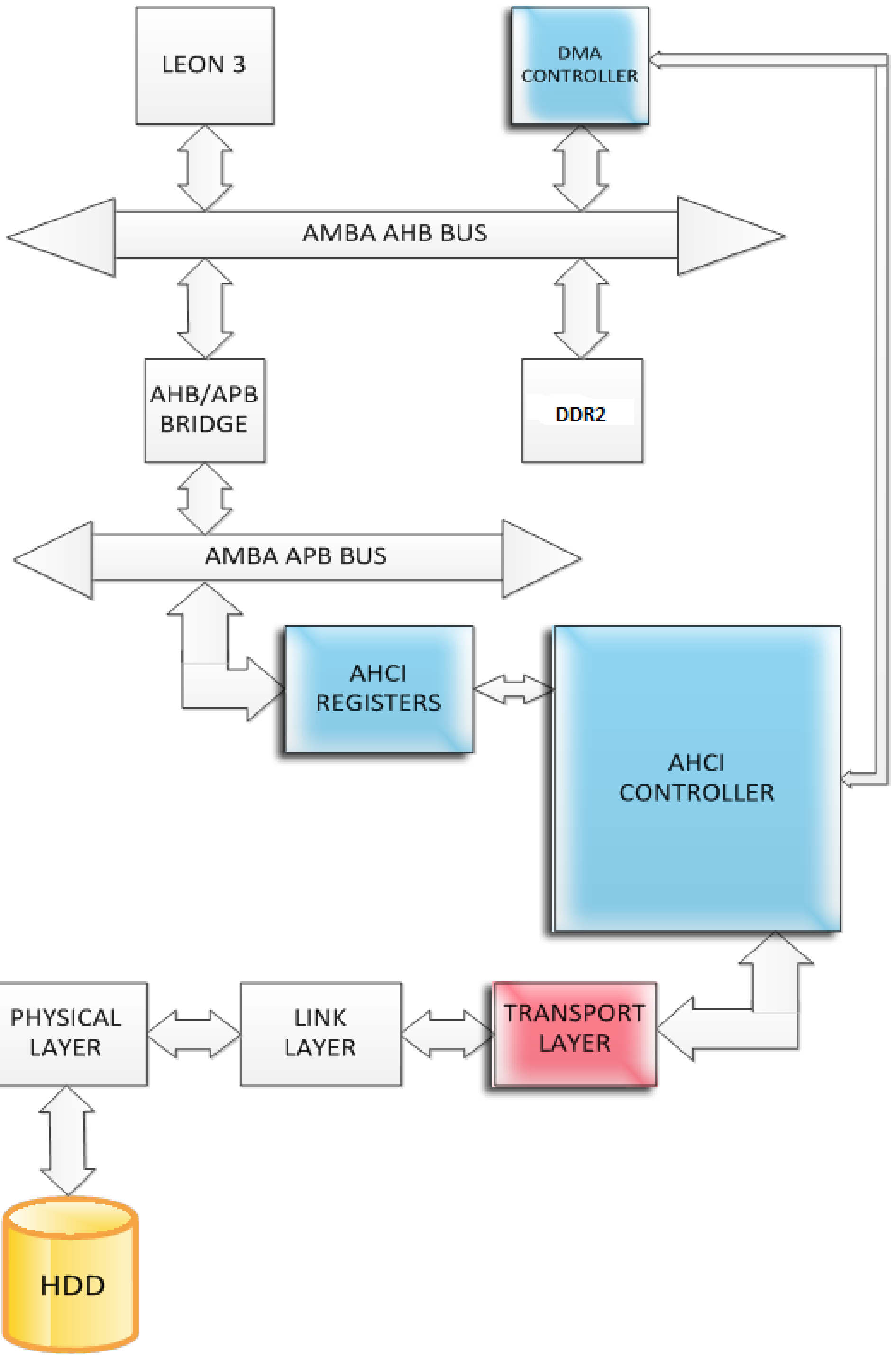


Figure 8: Overall System

The overall system is depicted in Figure 13. The modules designed from scratch are highlighted in blue and the one which had to be modified is highlighted in red. The SATA disk is highlighted in yellow. The system utilizes two buses for connecting the components, where the AHB bus is the main interconnection line and the APB bus is used to interface the AHCI Registers. Two masters and two slaves are present on the system. Leon3 is the first master, on which the C application runs. Leon3 accesses both slaves, which are the DDR2 memory and the AHCI Registers. The second master is the DMA controller. It only accesses the DDR2 memory. The AHCI Controller and the Front end are neither masters nor slaves on the bus. The reason of that is the Front end is related to SATA domain and does not need to be in interaction with the AMBA buses. The AHCI Controller is not involved in the bus, since it controls the DMA Controller and AHCI Registers with external control and data signals.

4.1.2 Leon 3

Leon3 is the processor that is used by this project. This processor is generated by Aeroflex Gaisler and it connects as a master to the AHB Bus. It executes the software which is written in C for the whole AHCI system. The software creates the necessary commands and puts these into the DDR2 memory. It then sets the required AHCI Registers which signal the AHCI Controller to start its operation.

4.1.3 DMA Controller

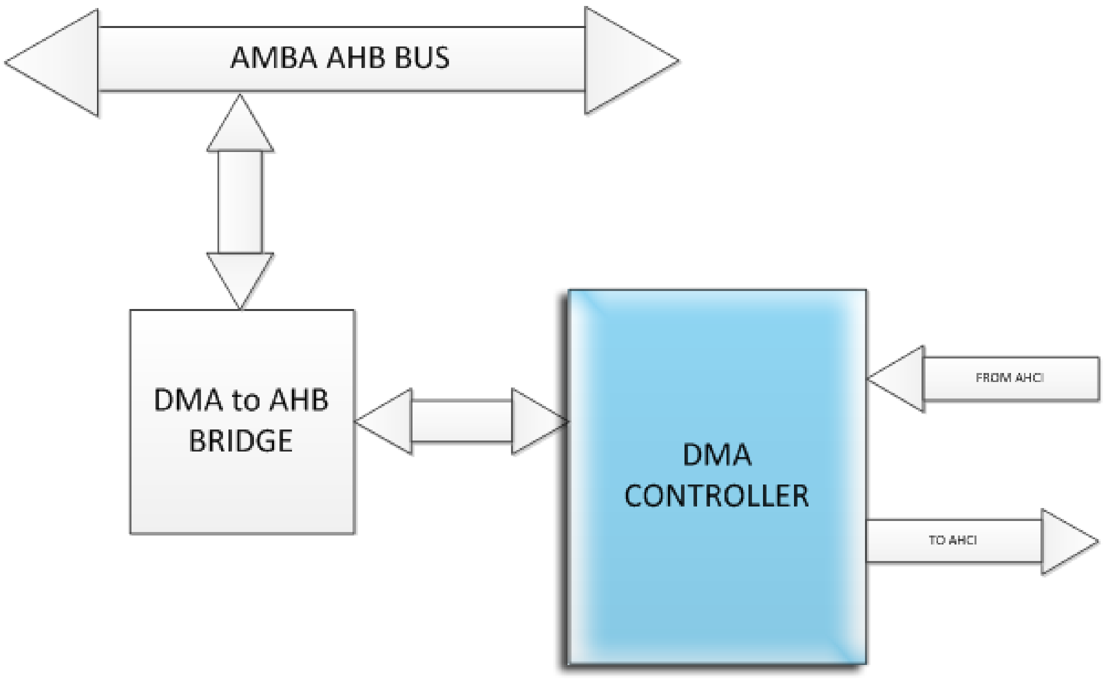


Figure 9: DMA Controller

The DMA Controller can be seen as highlighted in blue in Figure 14. It is responsible for the main data traffic between the system memory and the AHCI Controller. Hence its operation is vital for the whole system. Therefore, it is interfaced to the AHB bus through a previously designed and tested entity which is namely the DMA to AHB Bridge. The bridge takes the address, data and the type of operation from the DMA Controller and interacts with the AHB Bus. When the bus responds to the bridge, it conveys the information back to the DMA Controller.

4.1.4 AHB Bus

The AHB Bus is the main means of communication in the overall system. Leon3 and the DMA Controller are connected to it as masters, whereas the system memory and the AHCI Registers are presented as slaves. The communication among the masters and slaves is executed according to the AMBA specification, where a data phase follows an address phase, iterating for each clock cycle in a pipelined fashion.

4.1.5 DDR2

The DDR2 serves as the system memory. It is interfaced to the rest of the system through the AHB bus. The DDR2 which is present on the board ML505 is controlled with a memory controller generated by the Aeroflex Gaisler. This component runs at 190 MHz. Since the clock speeds are different for this controller than the rest of the AMBA domain, it’s operation had to be verified on hardware very carefully. The purpose of this component is to store the commands and the data which are built by the software prior to the operation of the AHCI Controller. During the operation, the commands are fetched by the DMA Controller from this memory. It interacts with the DMA Controller to fetch the necessary data for writes to the SATA disk and to store the data on reads from the SATA disk. It also stores the response FIS es that are sent from the SATA device.

4.1.6 AHCI Registers

The AHCI Registers were vital with regards to two points. First, they were serving as control signals for the AHCI Controller. The operation of the AHCI Controller had to be initiated based on the values of some specific AHCI Registers. These also reported back the status of a particular transfer to the AHCI Controller. Second, they were serving as debug registers. Several debug registers were planned to be implemented in this component. First part of debug registers should show in which state a particular component is. The DMA Controller’s and the AHCI Controller’s state machines were intended to be mapped to these debug registers. The purpose of this was to be able to understand in which state the components get stuck in case of a failure. Second part of the debug registers were solely focused to the AHCI

Controller. The variables in the AHCI Controller were mapped to these registers. The reason for this mapping can be traced back to the complexity of the AHCI Controller. It is the far most complicated component in this project and it was not enough just to observe in which state the controller is, because multiple duties were to be carried out in each state. Hence the variable debug registers were to be used if a step is completed within a state.

In the beginning, it was thought that the AHCI Register component could be designed straight forward as a memory array. However the AHCI standard required a more complicated design for this component. Both the software and the AHCI Controller had to be able to access this component. In case of a clash, which is not very common during the operation but is still a possibility, some mechanism had to be developed to give privilege to one side. As the AHCI standard was carefully examined, it has been seen that the software has to finalize the value of a register if a conflict occurs. Hence the registers were decided to be implemented in a way that the software writes overwrite the AHCI Controller writes. The AHCI Registers and how these were interface to the rest of the system can be seen in Figure 15 as highlighted in blue.

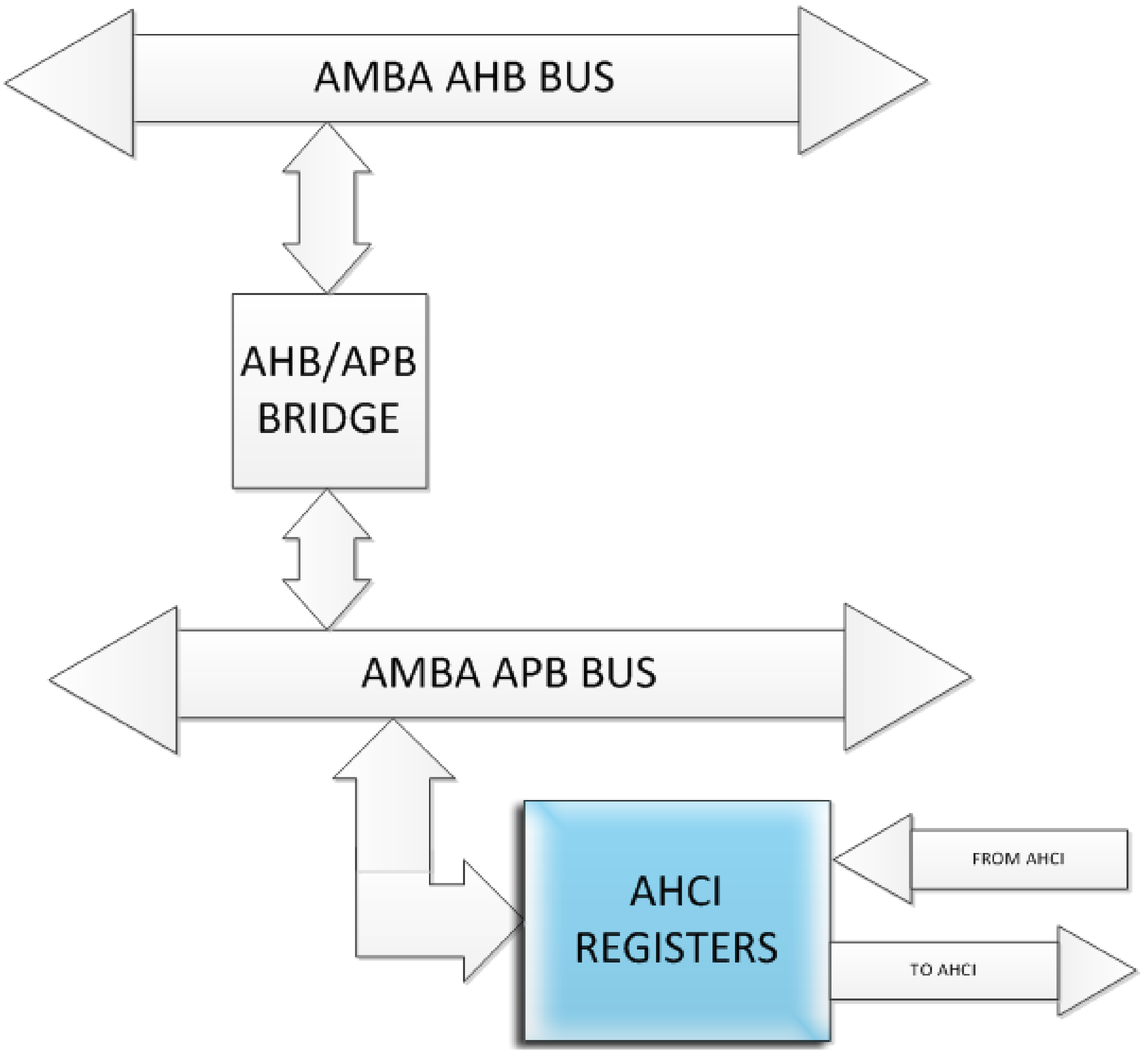


Figure 10: AHCI Registers

4.1.7 AHCI Controller

The AHCI Controller is the most complicated component, since it acts as a brain for the whole system. This duty is managed by a complicated state machine designed according to the specifications in the AHCI standard document [4]. As seen from Figure 16, it interacts with the AHCI Registers, SATA Front end and the DMA Controller. The main operation for the AHCI Controller is to fetch the commands from the system memory and execute these in order to get the correct responses from the SATA disk.

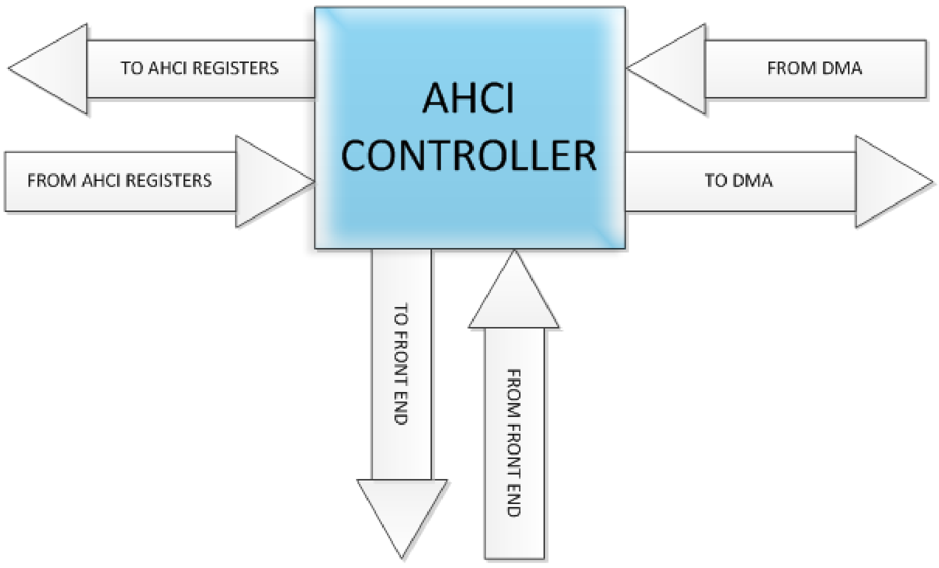


Figure 11: AHCI Controller

4.1.8 Front End

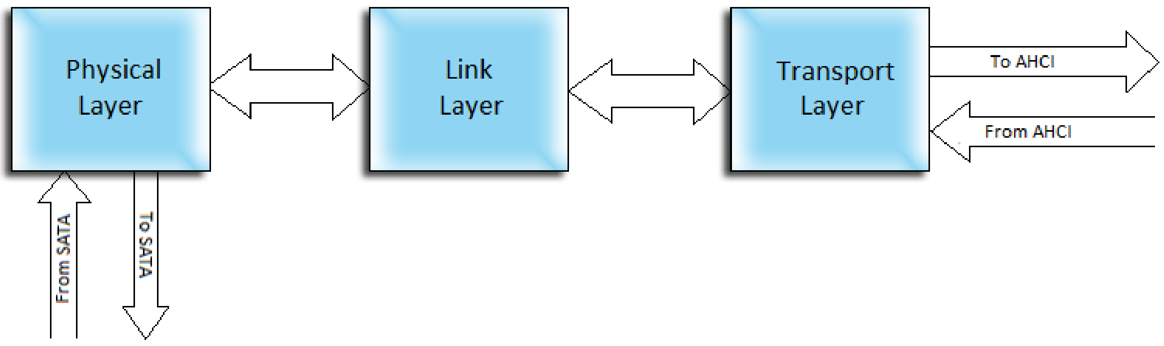


Figure 12: Front End

The Front End can be seen in Figure 17 and it is including the transport, link and physical layers. It was a result of a previous master thesis work and was planned to be taken as a black box to be integrated into this project. Unfortunately this was not exactly the case and the transport layer had to be modified to satisfy the needs of this project as will be discussed later.

The transport layers duty was to collect the FIS es in its asynchronous FIFO s and transmits these to the destination domain. It also provides clock synchronization for signals crossing clock domains. The link layer is between the transport layer and the physical layer and it handles the primitive traffic between these two which are the smallest amount of information transferred in this project. The physical layer handles the clock generation for the SATA domain. It also embodies the component GTP DUAL transceiver, through which it interacts with the SATA device.

4.2 Designing Phase

4.2.1 Leon3

The leon3 processor was already designed by Aeroflex Gaisler. Hence the only work regarding this component was the design of the software corresponding to the AHCI standard. The duties of the software were to create the commands and the data and place these to the system memory. Following this, it should set the corresponding registers so that the AHCI Controller can start its operation. The first step was to design a memory mapping that indicates the position of the commands and the data in the system memory. This mapping is shown in Figure 18. It consists of four different sections which are highlighted in different colors. The operation of the AHCI standard is pointer based. That is the command headers, which can be seen in the first region of Figure 18, are mainly pointers which indicate the location of the command tables and PRDs. Apart from being pointers, these headers also have some information about the command itself. For a full list of specifications, please consult the AHCI manual [4]. The second region contains the command tables and PRD s. The command tables have the vital piece of information for the SATA device, which is the command FIS. It is the structure upon which the SATA device responds. The PRD s depict the length and the location of the data in the system memory. In the software, a pointer to the base of this memory mapping has been defined, which is 0x41000000. Actually the DDR2 memory starts at 0x40000000. However, since the software application is also residing at this address, the first 16 Megabytes were left untouched to avoid collision with the C program. The above mentioned structures were built according to the memory mapping shown in Figure 18 through the C application. After that the necessary register values have been updated according to the AHCI standard and the start signal is given to the AHCI Controller. For a full list of updated registers, please consult the AHCI manual [4].

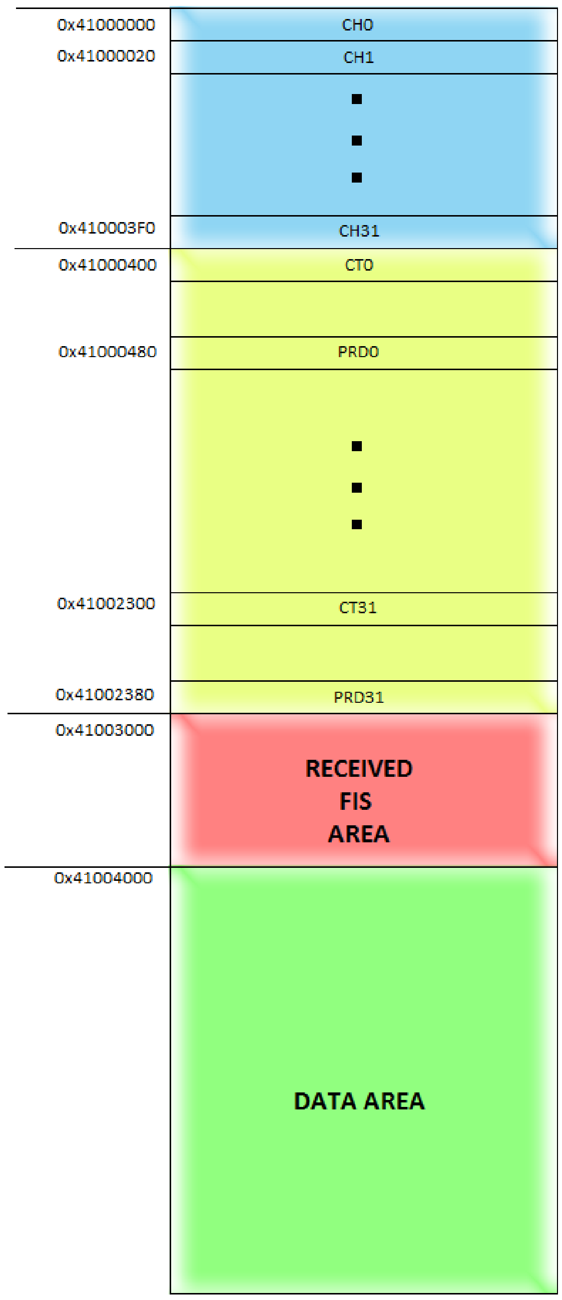


Figure 13: Memory Mapping

4.2.2 DMA Controller

The DMA Controller's inputs and outputs are visible in Figure 19. One side of the controller interacts with the AHB bus through a bridge. The other side is controlled by the AHCI Controller. The DMA Controller waits until it receives a start signal from the AHCI side. Once it recognizes the start signal, it's state machine is activated.

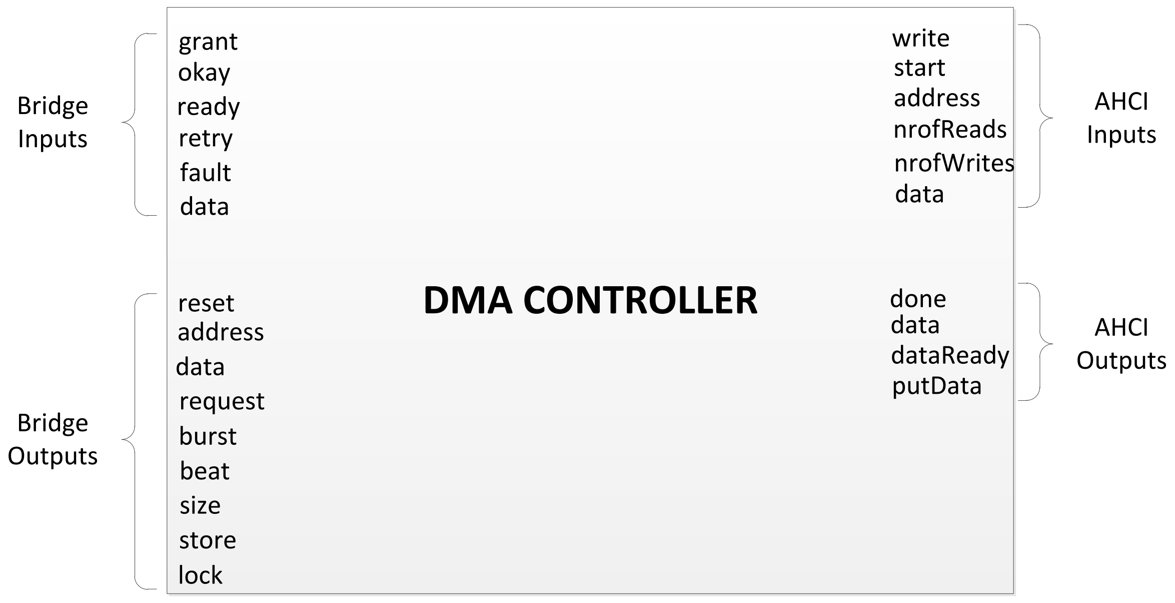


Figure 14: DMA Inputs & Outputs

The operation of the state diagram can be seen in Figure 20. Once the DMA is activated, it jumps to the Read or the Write state according to the type of the required operation. If it is a read operation, the DMA feeds the address to the AHB bus and notifies the bus that it requires to make a read. It also requests to access the bus. Once the given address is sampled by the bus and the requested data emerges, the DMA takes and conveys it back to the AHCI controller. The DMA stays in the Read state and continues this loop until the amount of data requested by the AHCI Controller initially has been transferred. The writing operation to the system memory gets executed in the same style. The only difference is that the AHCI Controller provides also the data additionally.

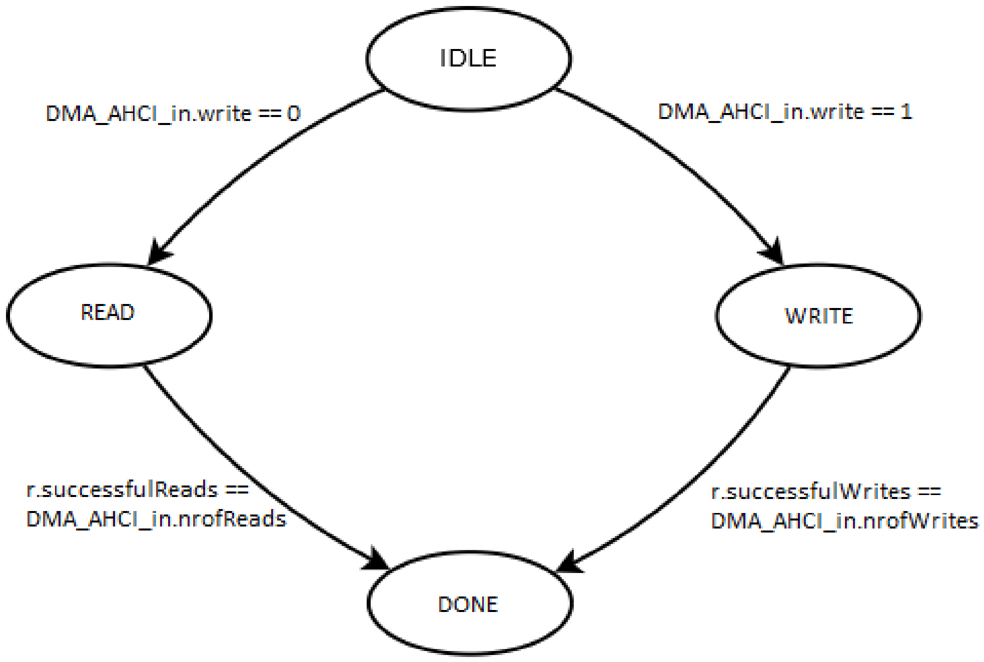

IDLE
DMA_AHCI_in.write == 0
DMA_AHCI_in.write == 1
READ
WRITE
r.successfulReads == DMA_AHCI_in.nrofReads
r.successfulWrites == DMA_AHCI_in.nrofWrites
DONE

Figure 15: State Diagram of the DMA Controller

4.2.3 AHCI Registers

The AHCI Registers are the flags of the system. The AHCI Controller shapes its operation based on the values present on these registers. Interfaced to the system through the APB bus, these register are writeable and readable both by the software and the AHCI Controller. The memory mapping of different registers residing in this component was designed based on the AHCI manual [4]. Each register in this component was realized using flip flops, which were inferred by the definition of a variable. The reading operation differs whether the AHCI Controller or the software wants to read. In case of software reads, the APB address is put on the bus and according to this address, the content of the required flip flops were obtained on each rising edge. In case of AHCI Controller however, a parallel read is required. The reason for this is that some states in the AHCI Controller need to read more than one register. This means that the content of multiple registers had to be revealed on one clock cycle. For this reason, the required registers had been wired out of the flip flop memory array and fed to the AHCI Controller. These signals can be seen in Figure 21 as AHCI outputs.

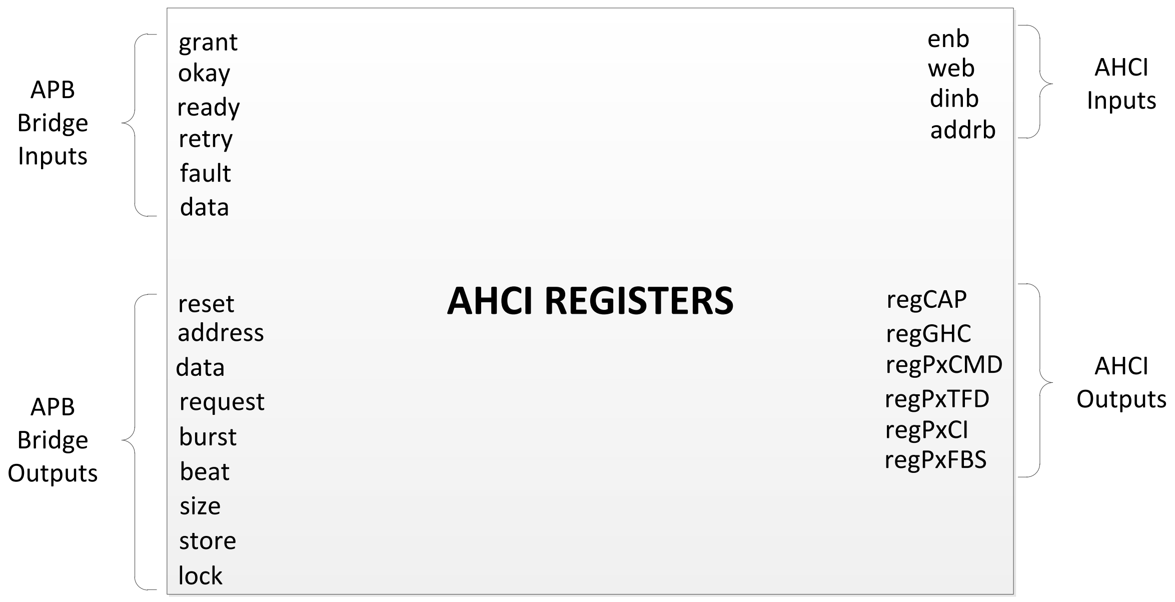


Figure 16: AHCI Regs Inputs & Outputs

Regarding the writing operation, software writes were handled according to the APB protocol. That is when the slave is selected; software has access to the flip flop array. AHCI Controller writes did not go through the bus, hence simple chip and write enable signals were enough. A challenge regarding the write operations was the issue of overwriting in the case of a clash between the two sides. As the AHCI manual was examined, the software should be the one which is updating the memory array in such a case. Hence in the code, it was assured that even if the AHCI Controller accesses the flip flop array in the same cycle as the software, the software will be the last one to update the array.

4.2.4 AHCI Controller

The AHCI Controller can be considered as the center of the whole design, since it is the main decision making mechanism. It has interactions with the DMA Controller, the Front end and the AHCI Registers. The main task of the controller is that the operation is run with respect to the AHCI standard. It fetches the commands by instructing the DMA Controller and then sends these over the Front end to the SATA device. Meanwhile it uses the AHCI Registers as flags and reads or updates these according to the phases of the operation. The input and output pins which provide the interaction of the AHCI Controller with the other components can be seen in Figure 22.

To explain the operation of the AHCI Controller more thoroughly, the partitions of its state machine during a specific command can be examined. The rest of this section addresses the traversing of the states during the operation of the "WRITE DMA EXT" command.

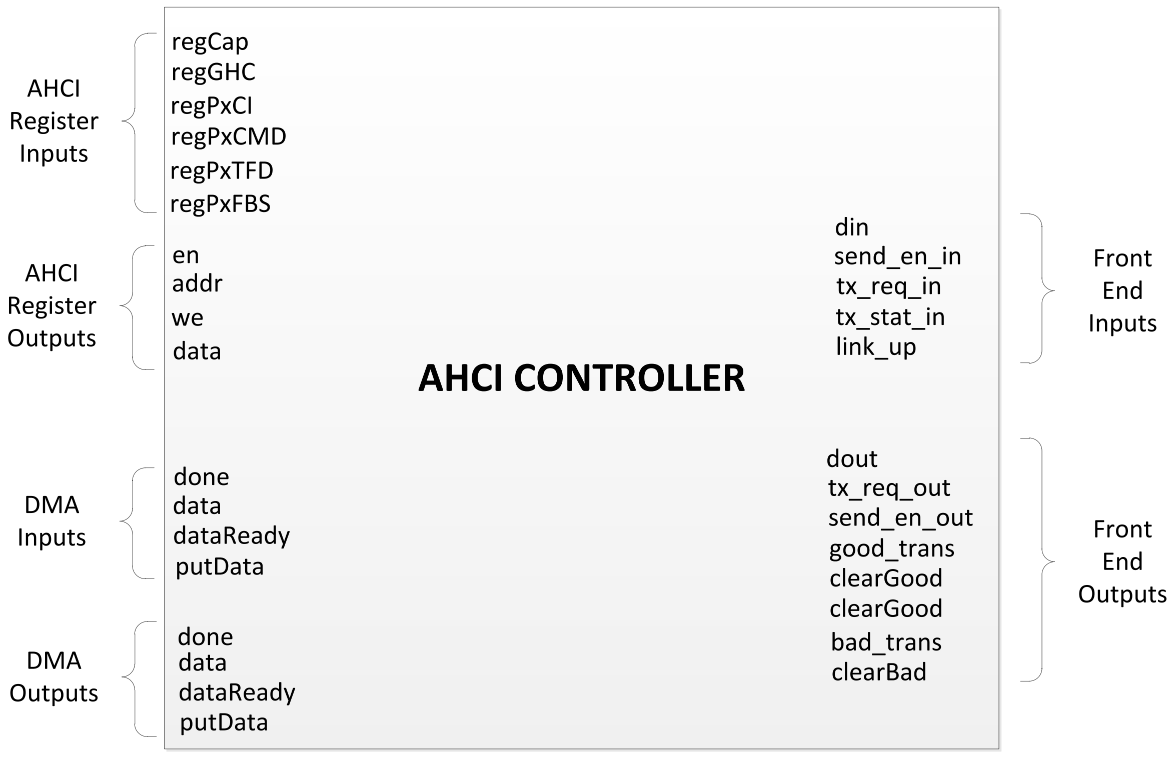


Figure 17: AHCI Controller Inputs & Outputs

When the AHCI Controller gets the start signal from the software through the AHCI Registers, it traverses some initial states to reset the defined variables. Then it lands on a state called "P_IDLE". This is the base state to which the controller returns after executing each phase of the required operation. After visiting this state for the first time, the controller looks for an executable command present in the command list. If it finds such a command, the controller traverses the states which are shown in Figure 23 so that the SATA device can receive the CFIS structure present in the command table of the command.

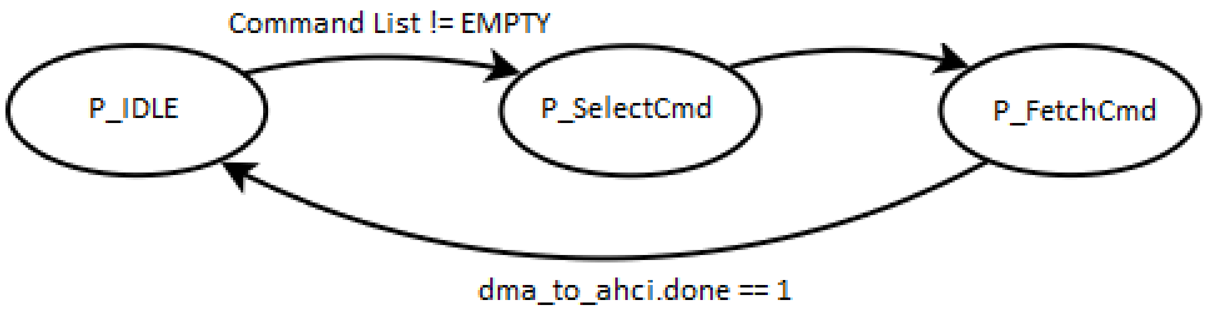


Figure 18: Issuing the command

When in P_IDLE, the Controller checks the related AHCI Registers to see if there are executable commands present in the command list. Once a command is found, it goes to the P_SelectCmd state.

Here it selects the next available command. From this state, it moves to P_FetchCmd unconditionally where it instructs the DMA Controller to fetch the command from the system memory. When the DMA Controller finishes its operation the AHCI Controller returns to P_IDLE.

Now that the "WRITE DMA EXT" command is fetched from the system memory, it needs to be transferred to the SATA device. This behavior can be traced in Figure 24. During the fetching of the command, the variable "pCmdToIssue" is set to 1. Hence the controller will jump from P_IDLE to CFIS_Xmit, where it instructs the Front end to send the fetched command to the SATA device. Once it gets back the response from the Front end suggesting that the SATA device has accepted the command, it jumps to the state CFIS_Success, where it clears the variable "pCmdToIssue", indicating that the transfer of the command has been completed successfully. From CFIS_Success, it returns to P_IDLE unconditionally.

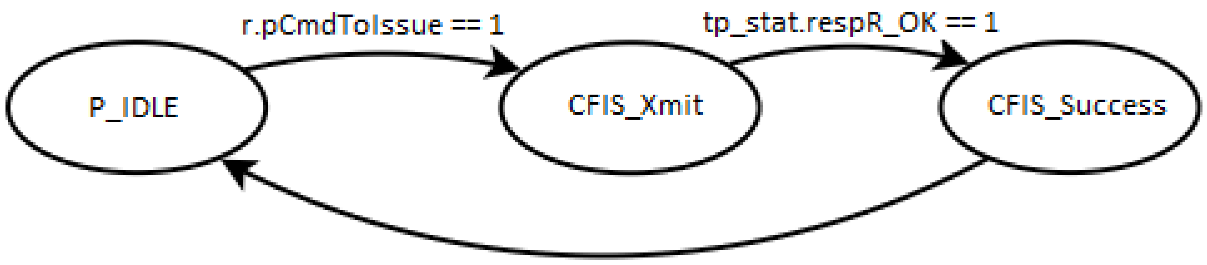


Figure 19: Transmitting the Command

The response from the SATA device is a “DMA Activate FIS”, since it was a “WRITE DMA EXT” command. When this FIS is received, the AHCI Controller goes to the NDR_Entry state, since it is a non-data FIS. There it updates the variable “FisType” according to the received FIS and travels to NDR_Entry where it accepts the incoming FIS and determines the next state based on the type of the FIS. Since it was a “DMA Activate FIS”, the next state to be traversed is DX_Entry. In this state, the controller fetches the PRD from the system memory which indicates the length and the address of the data in the system memory. After the PRD is fetched, it retrieves the data from the system memory accordingly and constructs the data FIS to be sent to the SATA device. When the data FIS is constructed the controller jumps to DX_Transmit. As the name of the state suggests, data transmission is occurring in this state. When the Front end approves this transfer the controller moves on to the DX_UpdateByteCount state. Here it updates the PRD’s byte count field in the system memory through the DMA Controller and returns to P_IDLE. This phase is outlined in Figure 25.

At this stage, the SATA device can send either another “DMA Activate FIS” or a “D2H Register FIS” as response. This depends on the initial command issued to the SATA disk, which indicated the number of sectors to be written. Hence if the command was stating that 5 sectors of data are to be written to the SATA disk, the device sends another “DMA Activate FIS”. Then the state machine shown in Figure 25 would have been repeated 5 times writing data to different sectors in the SATA disk. Once the AHCI has completed the writing operation to these 5 sectors, the SATA device issues back a “D2H Register FIS”, stating that the writing process has been completed successfully.

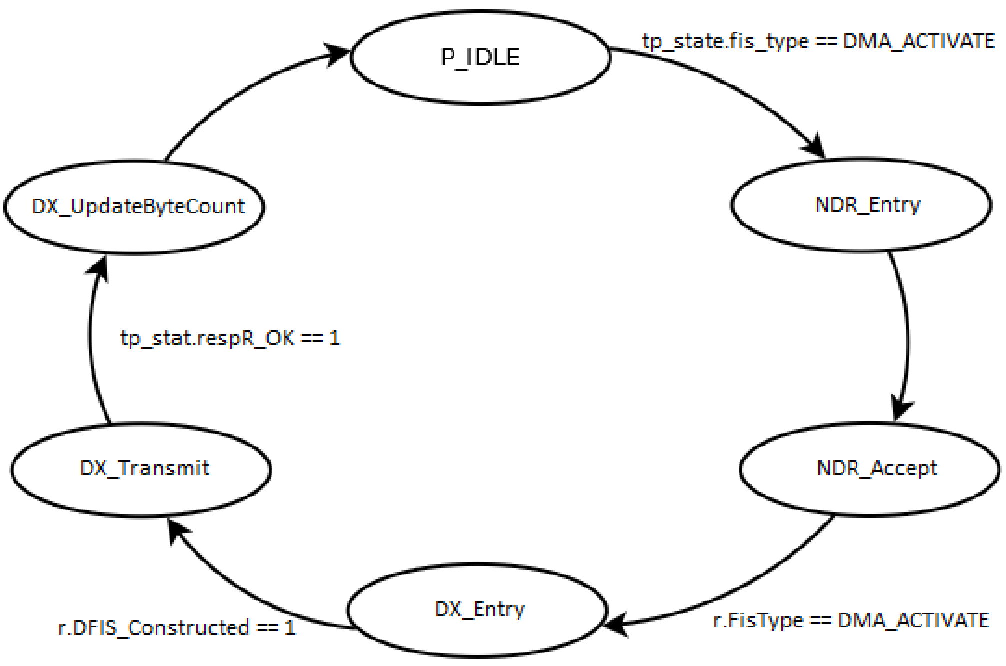


Figure 20: DFIS transmission to SATA device

Upon reception of the “D2H Register FIS”, the AHCI Controller traverses the states as shown in Figure 26, since this is a non-data FIS. First state to be visited is NDR_Entry, where the variable “FisType” is updated as “D2H_REGISTER”. From there, the controller jumps to NDR_Accept, where it receives the incoming FIS and determines the next state, which will be RegFIS_Entry. There, the received FIS is written back to the received FIS area of the system memory. RegFIS_ClearCl is the next state traversed by the AHCI Controller, where it clears the register that points to this command. This is done to prevent the AHCI Controller from fetching this command again, since it is handled.

The reading commands get executed in a similar fashion to the writing commands. The operation principle of the AHCI Controller is based on a handshaking protocol between the controller itself and the SATA device as described above. In general it can be summarized as follows. First the commands are being sent. Then FIS es are received as response. Finally the SATA device sends a registration FIS which marks the end of a transfer.

The states which handle the writing commands have been mentioned above. The reading commands are being handled through different states which are similar. Apart from these, the AHCI Controller has some states to boot up and handle the initial FIS received when the SATA device is turned on. In total, the AHCI Controller has 33 states in its finite state machine.

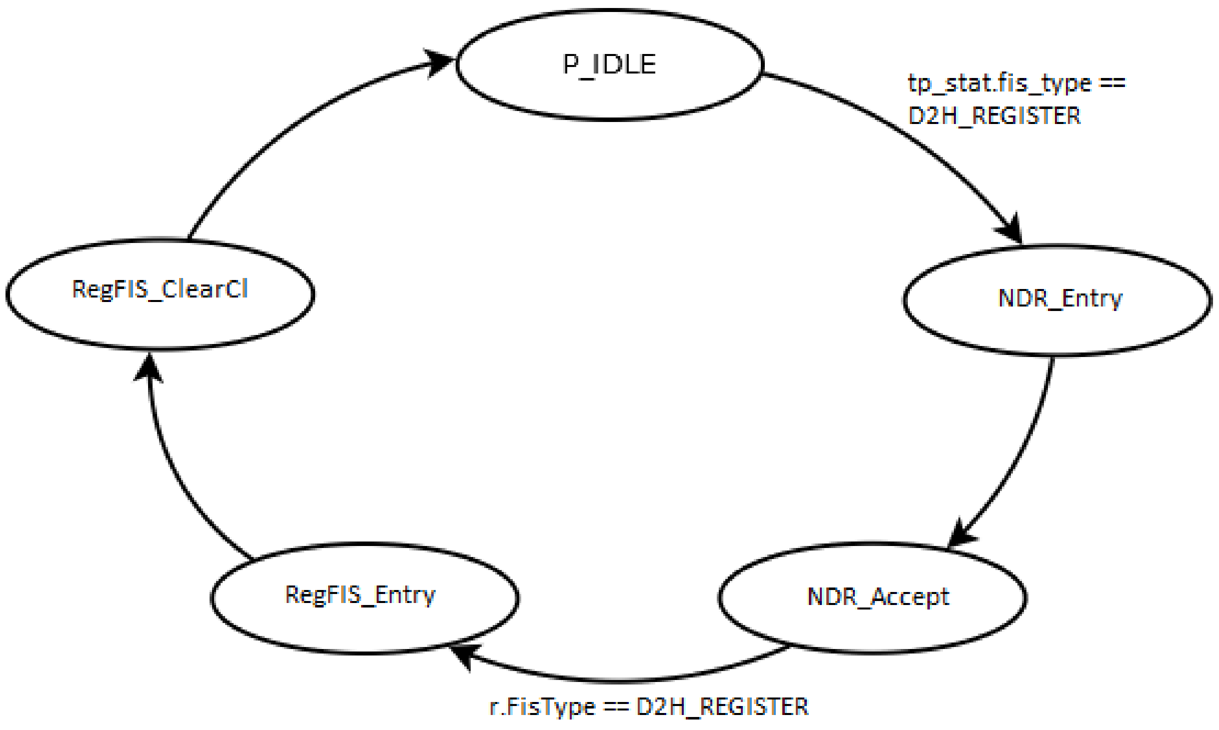


Figure 21: D2H FIS reception from SATA

4.2.5 Front End

From the start of the project, the Front end has been omitted as a black box which could be integrated together with the rest of the design to form the whole system. Unfortunately, this was not exactly true. Some modifications had to be performed in order to make it work with the rest of the design. First modification was related to the transport layer inside the Front end. It was designed to interface a DMA Controller. However the AHCI Controller had to interact with the Front end and not the DMA Controller. To fulfill this requirement, the transport layer had been modified accordingly.

Second modification was related to timing issues. After the synthesis was complete, the timing report suggested that some paths between the transport and link layers could not meet the timing. Hence, registers have been planted between these two layers to crack down the paths which had failing timing endpoints. After these fixes were realized, the Front end could be integrated successfully with the rest of the design.

5 RESULTS

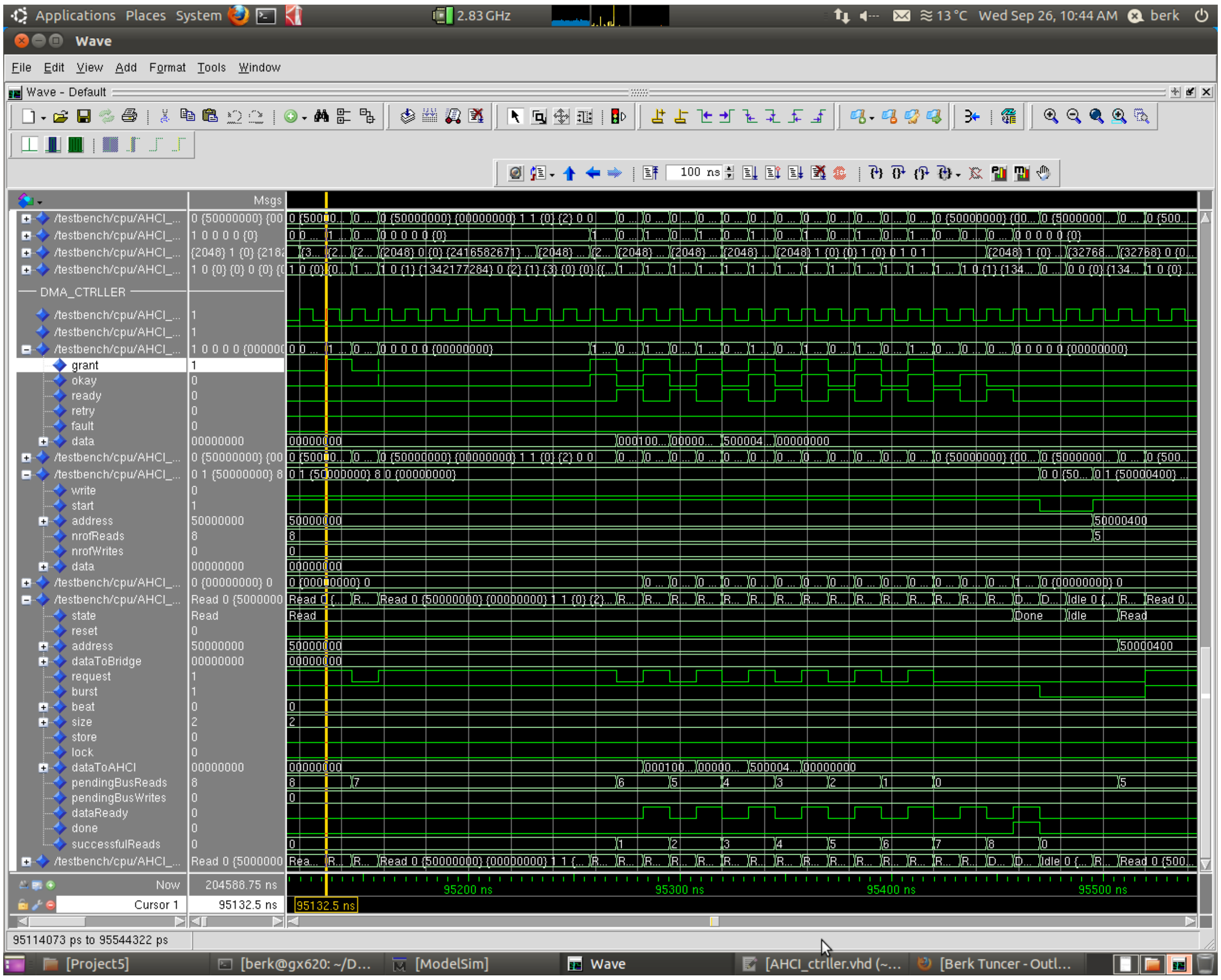
This section explains the simulation and synthesis results of the project, as well as the problems encountered during these phases. The project was broken down to two parts. First, the simulation part is handled through Modelsim. Then, the synthesis has been performed in Xilinx. Both in simulation and in synthesis, several problems had been encountered. Some of these problems were directly interfering with the functionality of the design. Others were caused inferring of unnecessary components and led to the waste of the area. This section elaborates how the results of the project have evolved in each phase. It also explains how the above mentioned problems have been resolved.

5.1 Simulation

After the research part had been completed, the next phase was the implementation and the simulation of the project. For that purpose, the plan was to design the necessary components and simulate these on their own before joining these to form the whole design. First component to be implemented was the AHCI Registers. These were APB interfaced registers and the implementation of these was straight forward. As tested in Modelsim after writing a simple C program, the registers could be both written and read through the software and through the testbench.

The next component to be designed was the DMA controller, which is the key component while interacting with the system memory. This component had to be examined very carefully for each corner case, since it could be very hard to debug once it has been integrated with the rest of the design. There was a built-in DDR2 memory controller for the ML505 board. Hence it has been chosen to serve as the system memory. The DMA controller had to be interfaced as a master to the AHB bus. Luckily, there was a bridge component designed by the company, which can be utilized to interface this component to the bus.

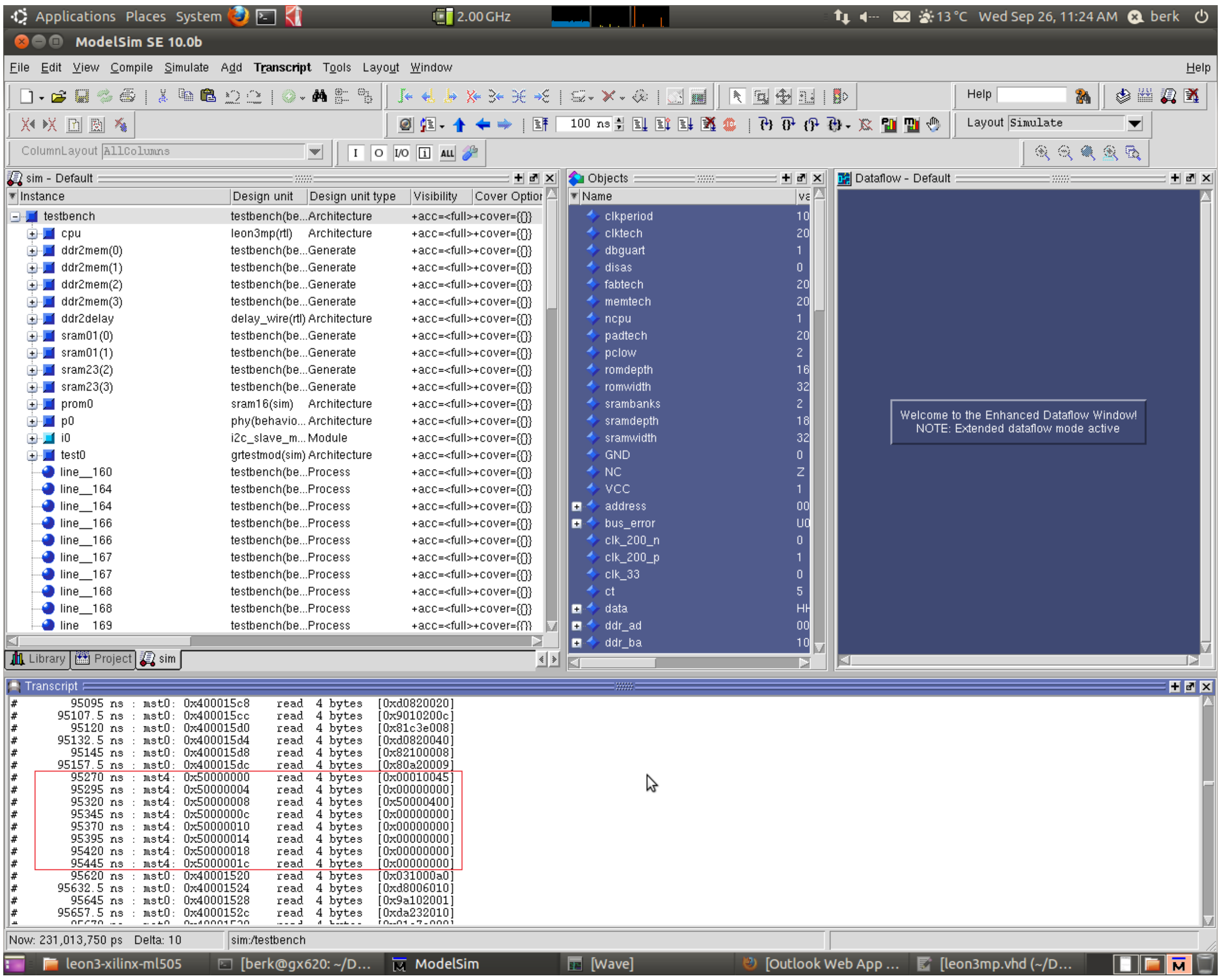
Despite of this simplification, the DMA controller was a tricky component due to the nature of the AMBA protocol. In Figure 27, some waveforms had been sketched to make the operation of the DMA controller more clear. In the figure, a read from the system memory by the DMA controller is shown. In this test, a simple state machine is designed that interacts both with the DMA controller and the AHCI registers. The C application creates and puts the data into the system memory first. Then it notifies the registers that the data has been planted into the system memory. The simple state machine checks the registers and once it sees that the data is ready to be fetched, it jumps to a state to interact with the DMA controller. Each transaction with the DMA begins with a start signal to DMA. The controller also states whether it is a read or write by flipping a control bit. It also specifies the address of the data and the number of reads or writes depending on the type of operation. To interact with the DMA, the controller uses three signals which are called grant, okay and ready. These are generated by the bridge as mentioned above and have the following meaning. Grant means that the bus is available for the use of the requesting master. Okay notifies the master that a transaction has finished on the bus and a new data can be fed. This signal is utilized for writes in the case of DMA. Ready means that the requested data can be sampled from the bus. This signal is utilized for reads in the case of DMA.



```
DMA_CTRLLER
grant                1
okay                 0
ready                0
retry                0
fault                0
data                 00000000
write                0
start                1
address              50000000
nrofReads            8
nrofWrites           0
data                 00000000
state                Read
reset                0
address              50000000
dataToBridge         00000000
request              1
burst                1
beat                 0
size                 2
store                0
lock                 0
dataToAHCI           00000000
pendingBusReads      8
pendingBusWrites     0
dataReady            0
done                 0
successfulReads      0
Now                  204588.75 ns
Cursor 1             95132.5 ns
95114073 ps to 95544322 ps
```

Figure 22: DMA Read

As can be seen from Figure 27, the controller instructs the DMA with 8 DW reads. The DMA request the access to the bus. The grant indicates that the DMA is allowed to access the bus. Then the okay signal is visible, which indicates that the DMA has read a DW from the system memory through the AHB. Following this, arrives the ready signal which means that the read data is available to the DMA and it can transfer it to the further modules. Since the number of reads has been indicated as 8 in this example, this sequence repeats itself 8 times. Once the DMA finishes its operation, it raises the done signal to notify the controller that it has finished the requested task. This signal has been added because some tasks in this project should only have been initiated after the DMA completes its operation. The writes succeeded in the same fashion, apart from that the write control signal to DMA controller had to be raised. Besides checking the waveform and trying to understand what was going on, the top file in the design had the property to switch on a property called “ahbtrace”. This made the monitoring of the transactions to the AHB possible. The address and the data which are involved in the AHB accesses could be observed through this feature as shown in Figure 28. This technique was also useful to check the software writes to system executed on Leon3 which was the master 0.



Instance	Design unit	Design unit type	Visibility	Cover Optior
testbench	testbench(be...	Architecture	+acc=<full>+cover={0}	
cpu	leon3mp(rtl)	Architecture	+acc=<full>+cover={0}	
ddr2mem(0)	testbench(be...	Generate	+acc=<full>+cover={0}	
ddr2mem(1)	testbench(be...	Generate	+acc=<full>+cover={0}	
ddr2mem(2)	testbench(be...	Generate	+acc=<full>+cover={0}	
ddr2mem(3)	testbench(be...	Generate	+acc=<full>+cover={0}	
ddr2delay	delay_wire(rtl)	Architecture	+acc=<full>+cover={0}	
sram01(0)	testbench(be...	Generate	+acc=<full>+cover={0}	
sram01(1)	testbench(be...	Generate	+acc=<full>+cover={0}	
sram23(2)	testbench(be...	Generate	+acc=<full>+cover={0}	
sram23(3)	testbench(be...	Generate	+acc=<full>+cover={0}	
prom0	sram16(sim)	Architecture	+acc=<full>+cover={0}	
p0	phy(behavio...	Architecture	+acc=<full>+cover={0}	
i0	i2c_slave_m...	Module	+acc=<full>+cover={0}	
test0	grtestmod(sim)	Architecture	+acc=<full>+cover={0}	
line__160	testbench(be...	Process	+acc=<full>+cover={0}	
line__164	testbench(be...	Process	+acc=<full>+cover={0}	
line__164	testbench(be...	Process	+acc=<full>+cover={0}	
line__166	testbench(be...	Process	+acc=<full>+cover={0}	
line__166	testbench(be...	Process	+acc=<full>+cover={0}	
line__167	testbench(be...	Process	+acc=<full>+cover={0}	
line__167	testbench(be...	Process	+acc=<full>+cover={0}	
line__168	testbench(be...	Process	+acc=<full>+cover={0}	
line__168	testbench(be...	Process	+acc=<full>+cover={0}	
line__169	testbench(be...	Process	+acc=<full>+cover={0}	

Name	Va
clkperiod	10
clktech	20
dbguart	1
disas	0
fabtech	20
memtech	20
ncpu	1
padtech	20
pclow	2
romdepth	16
romwidth	32
srambanks	2
sramdepth	18
sramwidth	32
GND	0
NC	Z
VCC	1
address	00
bus_error	U0
clk_200_n	0
clk_200_p	1
clk_33	0
ct	5
data	HH
ddr_ad	00
ddr_ba	10

Welcome to the Enhanced Dataflow Window!
NOTE: Extended dataflow mode active

```
#        95095 ns : mst0: 0x400015c8    read  4 bytes  [0xd0820020]
#      95107.5 ns : mst0: 0x400015cc    read  4 bytes  [0x9010200c]
#        95120 ns : mst0: 0x400015d0    read  4 bytes  [0x81c3e008]
#      95132.5 ns : mst0: 0x400015d4    read  4 bytes  [0xd0820040]
#        95145 ns : mst0: 0x400015d8    read  4 bytes  [0x82100008]
#      95157.5 ns : mst0: 0x400015dc    read  4 bytes  [0x80a20009]
#        95270 ns : mst4: 0x50000000    read  4 bytes  [0x00010045]
#        95295 ns : mst4: 0x50000004    read  4 bytes  [0x00000000]
#        95320 ns : mst4: 0x50000008    read  4 bytes  [0x50000400]
#        95345 ns : mst4: 0x5000000c    read  4 bytes  [0x00000000]
#        95370 ns : mst4: 0x50000010    read  4 bytes  [0x00000000]
#        95395 ns : mst4: 0x50000014    read  4 bytes  [0x00000000]
#        95420 ns : mst4: 0x50000018    read  4 bytes  [0x00000000]
#        95445 ns : mst4: 0x5000001c    read  4 bytes  [0x00000000]
#        95620 ns : mst0: 0x40001520    read  4 bytes  [0x031000a0]
#      95632.5 ns : mst0: 0x40001524    read  4 bytes  [0xd8006010]
#        95645 ns : mst0: 0x40001528    read  4 bytes  [0x9a102001]
#      95657.5 ns : mst0: 0x4000152c    read  4 bytes  [0xda232010]
```

Now: 231,013,750 ps Delta: 10 sim:/testbench

Figure 23: ahbtrace Feature

As can be seen in the transcript window of Modelsim, the 8 reads issued by the DMA were marked in red. The DMA is the master 4.

Two problems were encountered during the design of the DMA Controller. The first one was an addressing problem which caused overlapping in the system memory. That is if the DMA wanted to access the address 0x40000000 and then 0x41000000. It accesses two times the address 0x40000000. The reason of that was the size of the DDR2 memory controller. It was set to 16 Megabytes. Hence the location 0x41000000 did not exist and it was writing to 0x40000000 instead of 0x41000000.

The second problem required a more detailed analysis of the DMA controller module. The problem occurred because the controller was providing one less DW than the requested amount. As the waveforms were examined, it was clear that the controller was doing one less transaction than the required amount. The duration of the request signal generated by the DMA was determined with a variable called “pendingBusReads” which is dynamically altered during the runtime of the DMA. This check has been modified by counting the successful transactions on the bus. Once the number of successful transfers reaches the number of the initially requested transfers, the DMA completed its operation. This new approach solved the problem and the DMA started to function without any problems in both reads and writes.

The main focus of the research was the AHCI controller, which is outlined in the SATA AHCI manual [4]. It is a big, complicated state machine that interacts with the registers, the DMA controller and the front end interface. Until the states which had to be implemented in order to interact with the SATA disk, no big problems were encountered. However when these states had to be designed, two problems occurred. First, it became clear that the previous work could not be simulated since it was using a component as a black box, for which the simulation models were lacking. Second problem was more serious. Since the hard disk was a physical device, another way had to be found to simulate its behavior. The solution for the first problem was to bypass the transport layer, since it was a control layer which gave instructions to the link layer on how to operate. This could have been done from the AHCI controller itself. So, this approach was followed and a lot of time had been put into understanding how the link and physical layers operate. Finally, a controller was designed to control the link layer directly. However to test whether it was functioning correctly, a solution to the second problem had to be found, which was the simulation of the SATA disk’s behavior. In the master thesis report of 2008, it has been stated that two instances of GTP DUAL components could be connected in order to test the transmission and reception in simulation [2]. So it was a good idea to call two instances of the current front-end, which was consisting of link layer and physical layer only, and connect them for testing. Figure 29 exhibits how this setup was built.

In Figure 29, the front end which interfaces the AHCI Controller to the SATA disk is highlighted in blue. The other front end behaves like a SATA disk and is highlighted in green. To complete the behavior, a front end controller has been coded in VHDL. In the hardware, the SATA disk will send the necessary FIS es automatically. However in simulation, a controller for that purpose had to be implemented, since the front end is just an interface. As shown in figure, the two front ends are identical. The trick was that these were cross-connected through their transmission and reception ports. In this way, the two modules could communicate and the AHCI Controller was fully tested. Figure 30 shows an example transmission, where a “WRITE DMA EXT” command is executed and data has been transmitted from the AHCI Controller to the SATA disk over the bridge connection shown in Figure 29.

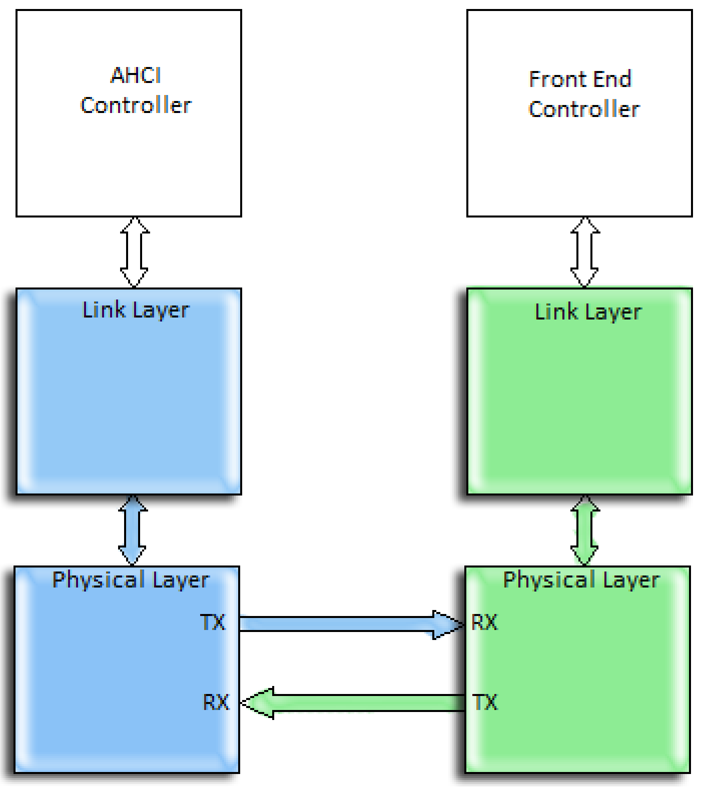


Figure 24: Front End Setup for Simulation

In Figure 30, the execution of the "WRITE DMA EXT" command has been shown. The data to be written to the SATA disk is "GAISLER" if interpreted in ASCII. This data and the command are first written to the DDR2 memory of the system. These are then fetched by the AHCI Controller and sent over the communication link to the SATA front end. The row which is highlighted in yellow in Figure 30 shows the DW s received by the SATA disk's front end. It is seen that just before the actual data transfer, "0x3737B57C" is transmitted. According to Table 1, this is the SOF primitive, which indicates the start of a data transmission. Then the type of the FIS is sent as "0x00000046", indicating that it is a data FIS. Then the actual data is sent word by word. After the data has been fully transmitted, the primitive EOF is sent.

In this stage, all of the design was simulated and verified. The next aim was the testing of the design on the board. However for this purpose, the transport layer had to be included as well, since in the absence of synchronization, the design would fail during clock domain crossing. This did not pose a problem in the simulation, since the same clock frequency could be fed to all of the modules, because there was no SATA disk present.

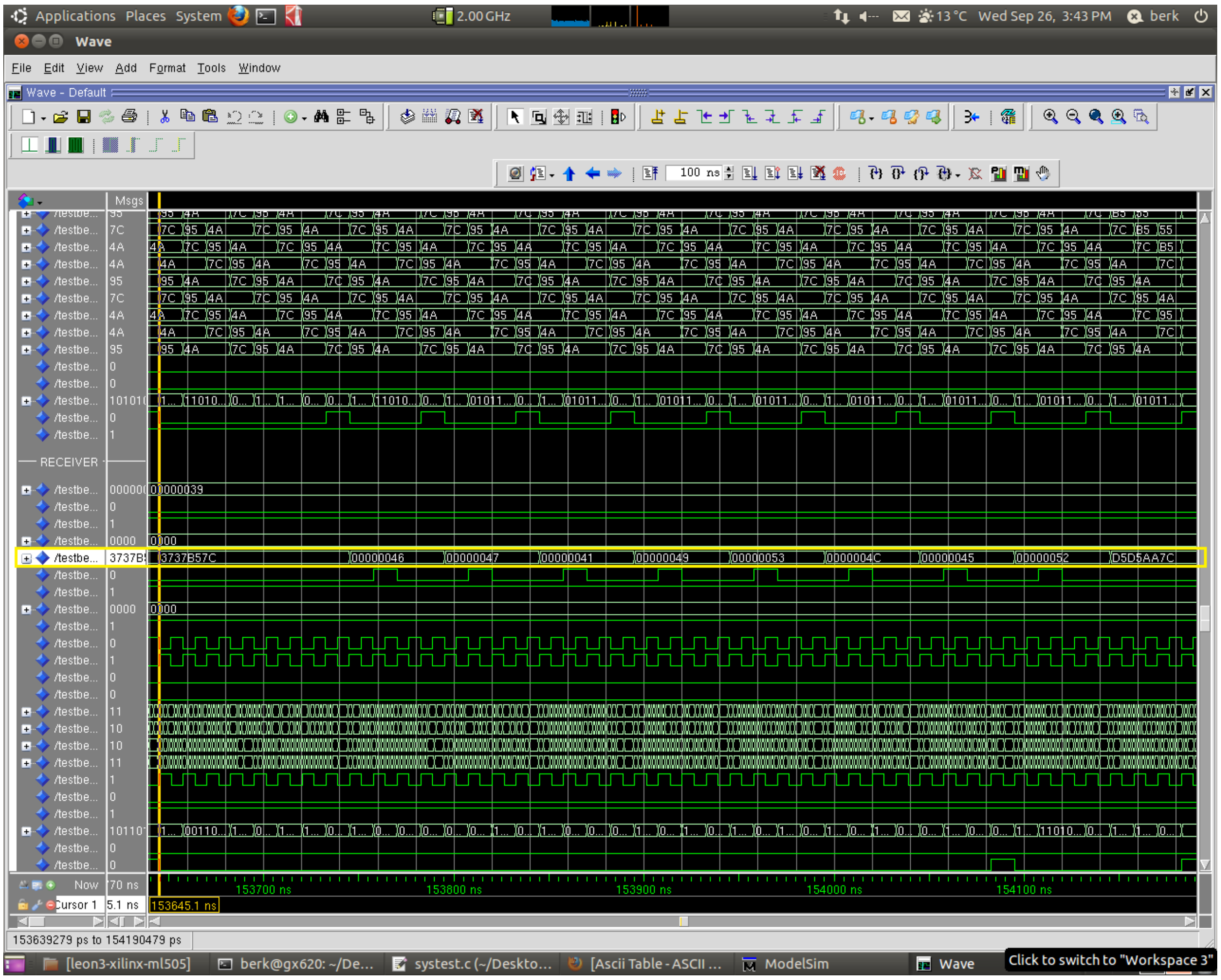

Applications Places System
Wave
File Edit View Add Format Tools Window
Wave - Default
RECEIVER
Now
Cursor 1
153700 ns
153800 ns
153900 ns
154000 ns
154100 ns
153639279 ps to 154190479 ps

Figure 25: Data reception on SATA side

5.2 Synthesis

Once the project reached the phase of synthesis, the method of partially testing the modules was consulted again. In order to see the outputs of the design, a software that connects to the ML505 board through the JTAG link was used. This software was called GRMON and was created by the company. Once initiated, it shows which components are present on the system, indicates the address space of each component and depicts also whether a AHB connected component is a master or a slave. To make the AHCI registers work and seeing the values written to these through GRMON was essential for testing the functionality of the rest of the design, since a part of the AHCI Registers were allocated to serve as debug registers. These debug registers were used for example to understand in which state the AHCI Controller hangs in case of malfunctioning.

After the AHCI Registers, the next component to be tested was the DMA Controller. At this stage, a problem occurred. The DDR2 memory controller started to malfunction. It was a complicated component which operated at 190 MHz and since the AMBA domain was operating at 80 MHz, the problem could have occurred because of that. However, since understanding the problem in the DDR2 memory controller was beyond the scope of this thesis work, some aid was required from the company. After making a few trials to fix the memory controller, it has been concluded that it is best to switch to a newer framework with a new DDR2 memory controller. However this did not happen immediately. While waiting for the framework, the system memory was replaced with the AHBRAM module, which was another module generated by the company. However it was a small component. Apart from that, when testing the DMA multiple reads and writes started to fail. That is when requesting a 8 DW read for example, the DMA was returning only the first DW and hanged. At this point, it was hard to judge whether the problem lied in the DMA controller or in AHBRAM, since it was working in simulation with the DDR2 memory controller. Soon it was found out that the AHBRAM did not support busy cycles in AMBA protocol during a transfer. The DMA was making a transfer and then confronting a busy cycle, indicating that there is still DW s to transfer, but could not be finished at the current time instance. Although this would have worked with the DDR2 memory controller, it was unnecessarily wasting cycles. Hence it has been decided to fix this problem. When the code of the DMA controller was examined in detail, it was found out that the problem was related to the burst control given to the AHB. The burst was asserted and de-asserted for each transfer of DW. Hence first DW was transferred, but when then second DW was about to be transferred, a busy cycle was confronted. This was the reason why AHBRAM was hung. Through keeping the burst control signal throughout the transfer, this issue got resolved.

Although the DMA issue was fixed, the AHBRAM was too small and the C program could not be run on such a small memory. Apart from that, the AHCI Controller needed to fetch commands and data from the system memory, for which the AHBRAM was too small again. Because of this reason, the data set has been shrunk to a more feasible amount and not all of the commands were put into the system memory. Luckily, GRMON had some features that enabled the user to write/read to/from memory. Hence, scripts had been generated instead of a C program to put the commands and the data into the AHBRAM. These scripts were also utilized to set the required AHCI Registers after the commands and the data had been planted into the system memory. By taking these measures, further testing of the synthesis of the remaining design could be continued.

The next challenge was to utilize the front end to interface the SATA disk to the board. However, the transport layer was continuing to pose problems since the Xilinx tool was failing to compile it as it is. Later on, it was going to be discovered that the problem was a missing ".ngc" file, which was utilized as a black box in the transport layer to realize an asynchronous FIFO. Unfortunately, this ".ngc" file overlooked at first and a decision had been made in the favor of coding a transport layer from scratch. After doing some research about clock synchronization, it was obvious that the methods required for this project were using an asynchronous FIFO for data transfer and double flip-flop synchronizing for control signals crossing the clock domains. However after designing a transport layer and connecting it together with the link and the physical layers, the communication with the SATA disk could not be established as expected. This was looking like a dead end and it was very hard to determine where the problem was occurring. It could be in the transport layer, the AHCI Controller or even in the link and the physical layers, since none of these was proven to be working on the board. As the most feasible solution, continuing with the front end designed by the master thesis work of 2010 was picked, since in its report it was stated that the communication with the SATA disk was successful. After careful examination of the previous master thesis work, the".ngc" file and its usage as a black box have been discovered. The author's lack of experience using the Xilinx XST tool made the overlooking of this file easier. After introducing this file to the Xilinx XST, the compilation of the front end was completed without any errors.

For integrating the front end with the designed AHCI Controller, AHCI Registers and the DMA Controller however, another problem had to be solved. The asynchronous FIFO s outputs in the transport layer were designed to interact with a DMA Controller. In this project however, a generic transport layer was needed that could function with the AHCI Controller, since the AHCI Controller was interacting with the DMA Controller. Hence the transport layer had to be modified to cooperate with the AHCI Controller. After all of these changes, the design was assembled and synthesized in the Xilinx XST without any problems.

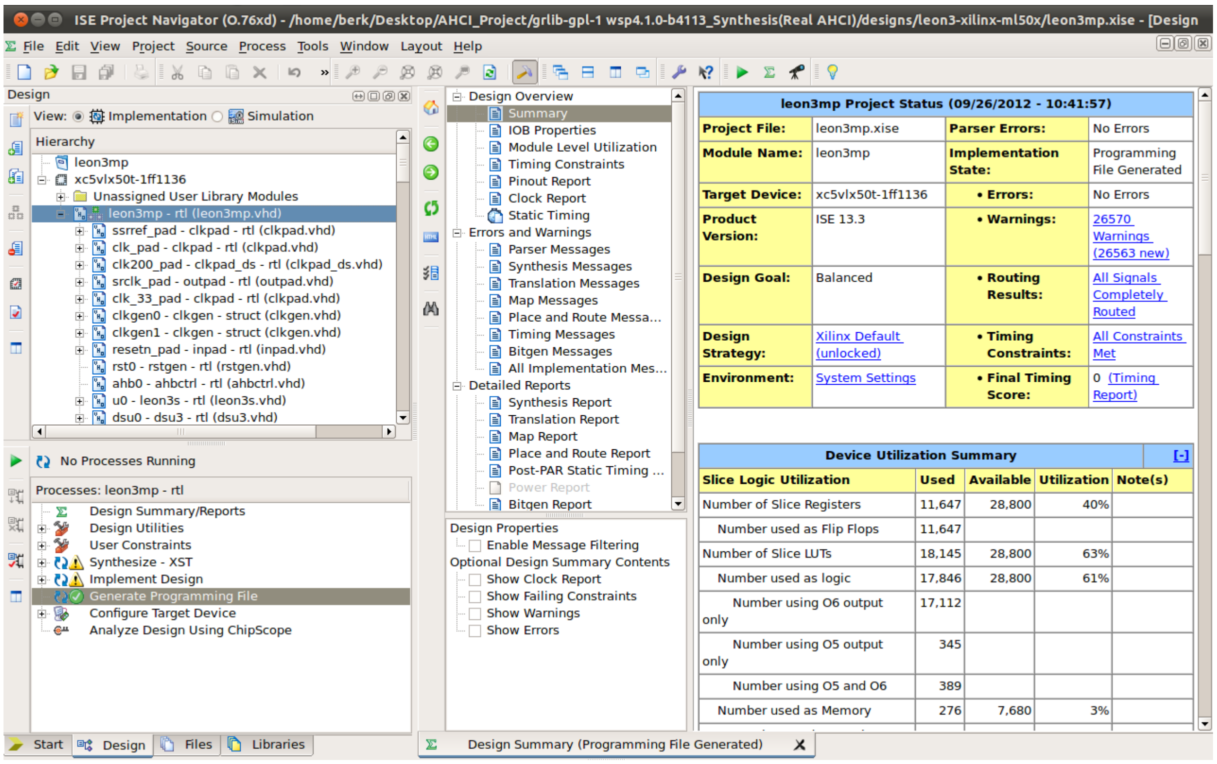
Meanwhile, the new framework with the DDR2 memory controller has arrived and the system memory was switched back from AHBRAM to DDR2. This eased the testing with the C application which was required for a complete design. Also the command and data amount that could be embedded to the system memory increased significantly.

First test was done issuing the "IDENTIFY DEVICE" command to the SATA disk. As stated in the master thesis report 2008, the SATA device should respond with a PIO Setup FIS and a data FIS to this command [2]. The PIO Setup FIS could be monitored in the registers using GRMON. However the data FIS could not be received. This could have a lot of reasons. However since the design was able to receive an FIS from the SATA device, the connection was successfully established. Hence the probability of a timing error was high. Indeed, when checking the timing report of the design, it was seen that the path between the link and the transport layers have failed to meet timing. Registers have been planted to crack down this path. After re-synthesizing the design, it was confirmed that all of the timing errors were eliminated. However when run in GRMON, the problem occurred again. Examining the state machine of the AHCI Controller, it became clear that it goes to another state directly after receiving the PIO Setup FIS. Hence a counter had been created to wait for the data FIS to arrive after the PIO Setup FIS. After doing this, it was confirmed in GRMON that the AHCI Controller was able to receive both FIS es.

The final problem encountered in the synthesis was an area problem. A variable was created to receive and send data. For example during a "WRITE DMA EXT" command, the AHCI Controller fetches the data from the system memory and puts it to this temporary variable. Then this variable is read and the data had been transferred to the SATA device. This has been done because it was the easiest way of coding. However it was very unwise, since it was inferring 4096 D-flip-flops. This was wasting the area unnecessarily, but also increasing the synthesizing time since the tool was struggling heavily during the optimization phase of the

AHCI Controller. To prevent this, the design of the AHCI Controller has been updated to map this variable to RAM instead of D-flip-flops. A component called “Syncram” was generated by the company already. This was utilized to minimize the area usage. After changing to this component, area ratio decreased from 139 to 96. The area constraint was defined as 105, hence the design met the all of the constraints.

As a result, Figure 31 summarizes last state for this project which is realized on board Xilinx ML505 in the right column.



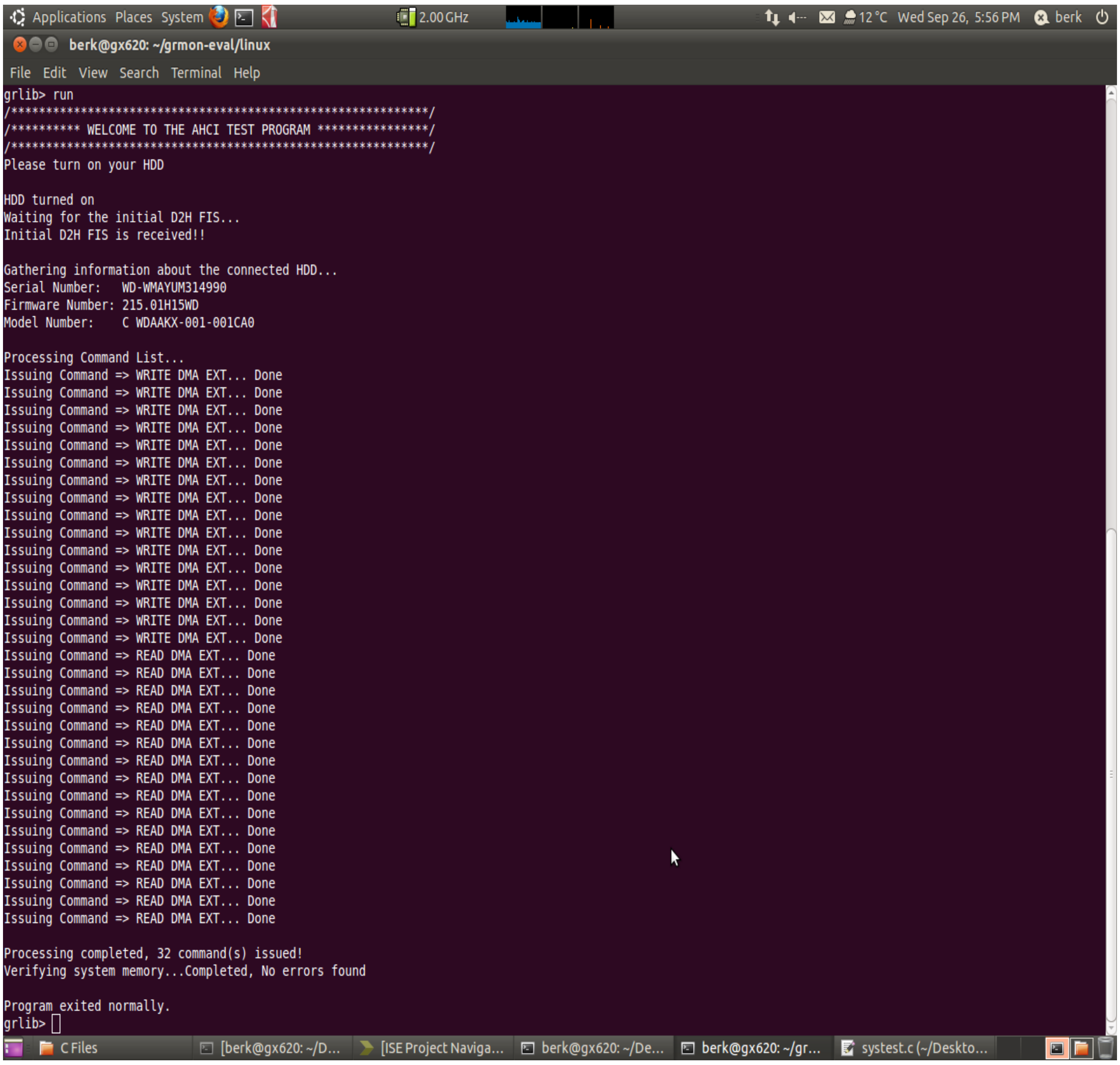
ISE Project Navigator (O.76xd) - /home/berk/Desktop/AHCI_Project/grlib-gpl-1 wsp4.1.0-b4113_Synthesis(Real AHCI)/designs/leon3-xilinx-ml50x/leon3mp.xise - [Design

leon3mp Project Status (09/26/2012 - 10:41:57)			
Project File:	leon3mp.xise	**Parser Errors:**	No Errors
Module Name:	leon3mp	**Implementation State:**	Programming File Generated
Target Device:	xc5vlx50t-1ff1136	• **Errors:**	No Errors
Product Version:	ISE 13.3	• **Warnings:**	26570 Warnings (26563 new)
Design Goal:	Balanced	• **Routing Results:**	All Signals Completely Routed
Design Strategy:	Xilinx Default (unlocked)	• **Timing Constraints:**	All Constraints Met
Environment:	System Settings	• **Final Timing Score:**	0 (Timing Report)

Device Utilization Summary				[-]
Slice Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of Slice Registers	11,647	28,800	40%	
Number used as Flip Flops	11,647			
Number of Slice LUTs	18,145	28,800	63%	
Number used as logic	17,846	28,800	61%	
Number using O6 output only	17,112			
Number using O5 output only	345			
Number using O5 and O6	389			
Number used as Memory	276	7,680	3%	

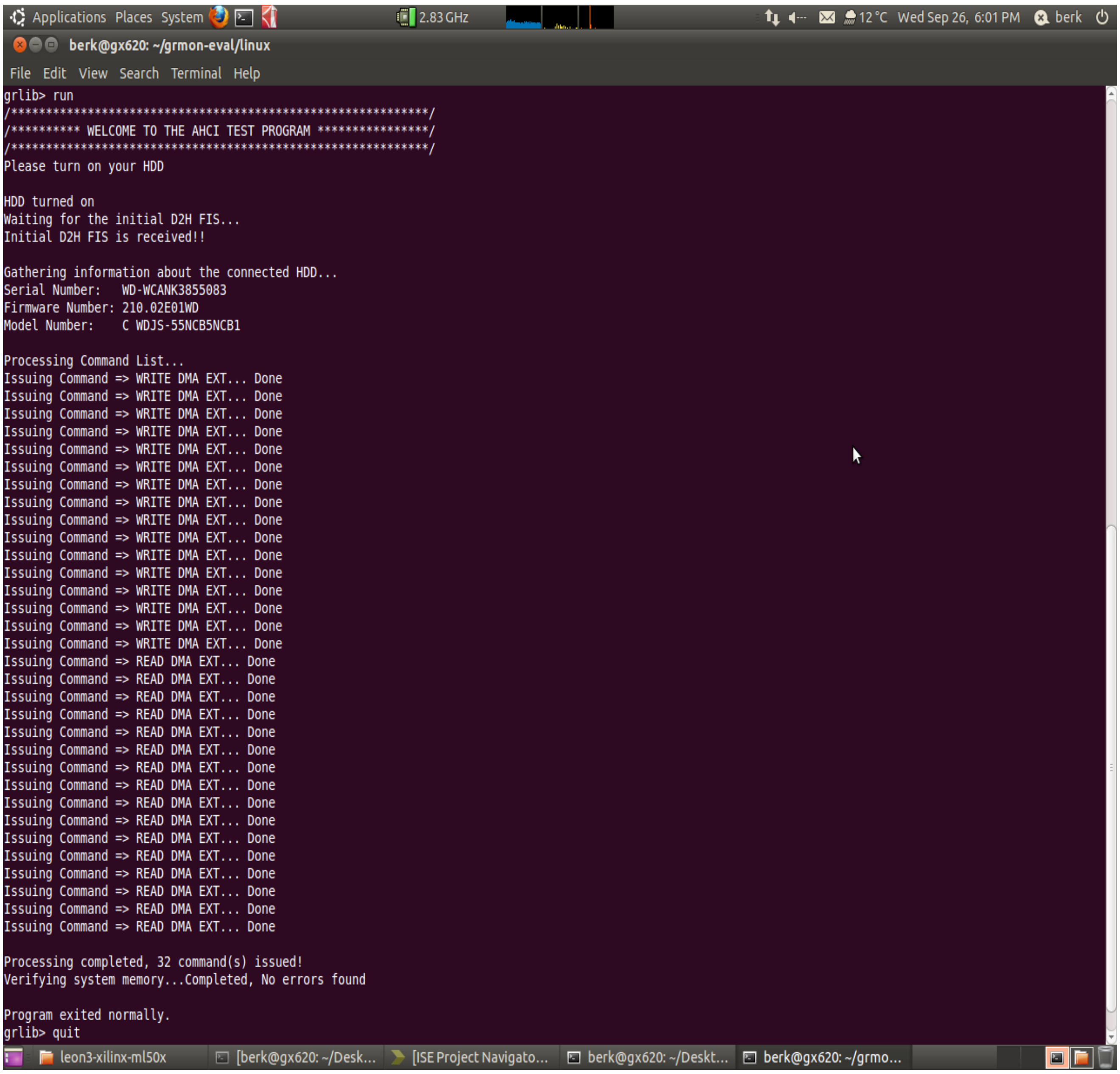
Figure 26: Synthesis Summary

Finally a test application in C has been written which produces 32 commands in total, where the AHCI Controller writes to 16 different locations in the SATA disk and then reads back from these for testing the results. Before issuing the write and read commands, it issues a “IDENTIFY DEVICE” command to understand and print the features of the attached SATA device. Figure 32 and 33 show the test applications run for two different Western Digital disks.


Applications Places System
2.00 GHz
12 °C Wed Sep 26, 5:56 PM berk
berk@gx620: ~/grmon-eval/linux
File Edit View Search Terminal Help
C Files
[berk@gx620: ~/D...
[ISE Project Naviga...
berk@gx620: ~/De...
berk@gx620: ~/gr...
systest.c (~/Deskto...

```
grlib> run
/***************************************************************/
/********** WELCOME TO THE AHCI TEST PROGRAM ****************/
/***************************************************************/
Please turn on your HDD

HDD turned on
Waiting for the initial D2H FIS...
Initial D2H FIS is received!!

Gathering information about the connected HDD...
Serial Number:   WD-WMAYUM314990
Firmware Number: 215.01H15WD
Model Number:    C WDAAKX-001-001CA0

Processing Command List...
Issuing Command => WRITE DMA EXT... Done
Issuing Command => WRITE DMA EXT... Done
Issuing Command => WRITE DMA EXT... Done
Issuing Command => WRITE DMA EXT... Done
Issuing Command => WRITE DMA EXT... Done
Issuing Command => WRITE DMA EXT... Done
Issuing Command => WRITE DMA EXT... Done
Issuing Command => WRITE DMA EXT... Done
Issuing Command => WRITE DMA EXT... Done
Issuing Command => WRITE DMA EXT... Done
Issuing Command => WRITE DMA EXT... Done
Issuing Command => WRITE DMA EXT... Done
Issuing Command => WRITE DMA EXT... Done
Issuing Command => WRITE DMA EXT... Done
Issuing Command => WRITE DMA EXT... Done
Issuing Command => WRITE DMA EXT... Done
Issuing Command => READ DMA EXT... Done
Issuing Command => READ DMA EXT... Done
Issuing Command => READ DMA EXT... Done
Issuing Command => READ DMA EXT... Done
Issuing Command => READ DMA EXT... Done
Issuing Command => READ DMA EXT... Done
Issuing Command => READ DMA EXT... Done
Issuing Command => READ DMA EXT... Done
Issuing Command => READ DMA EXT... Done
Issuing Command => READ DMA EXT... Done
Issuing Command => READ DMA EXT... Done
Issuing Command => READ DMA EXT... Done
Issuing Command => READ DMA EXT... Done
Issuing Command => READ DMA EXT... Done
Issuing Command => READ DMA EXT... Done
Issuing Command => READ DMA EXT... Done

Processing completed, 32 command(s) issued!
Verifying system memory...Completed, No errors found

Program exited normally.
grlib>
```

Figure 27: WD-WMAYUM314990 Results



Applications Places System
2.83 GHz
12 °C Wed Sep 26, 6:01 PM berk
berk@gx620: ~/grmon-eval/linux
File Edit View Search Terminal Help
leon3-xilinx-ml50x
[berk@gx620: ~/Desk...
[ISE Project Navigato...
berk@gx620: ~/Deskt...
berk@gx620: ~/grmo...

```
grlib> run
/***************************************************************/
/********** WELCOME TO THE AHCI TEST PROGRAM ****************/
/***************************************************************/
Please turn on your HDD

HDD turned on
Waiting for the initial D2H FIS...
Initial D2H FIS is received!!

Gathering information about the connected HDD...
Serial Number:    WD-WCANK3855083
Firmware Number: 210.02E01WD
Model Number:     C WDJS-55NCB5NCB1

Processing Command List...
Issuing Command => WRITE DMA EXT... Done
Issuing Command => WRITE DMA EXT... Done
Issuing Command => WRITE DMA EXT... Done
Issuing Command => WRITE DMA EXT... Done
Issuing Command => WRITE DMA EXT... Done
Issuing Command => WRITE DMA EXT... Done
Issuing Command => WRITE DMA EXT... Done
Issuing Command => WRITE DMA EXT... Done
Issuing Command => WRITE DMA EXT... Done
Issuing Command => WRITE DMA EXT... Done
Issuing Command => WRITE DMA EXT... Done
Issuing Command => WRITE DMA EXT... Done
Issuing Command => WRITE DMA EXT... Done
Issuing Command => WRITE DMA EXT... Done
Issuing Command => WRITE DMA EXT... Done
Issuing Command => WRITE DMA EXT... Done
Issuing Command => READ DMA EXT... Done
Issuing Command => READ DMA EXT... Done
Issuing Command => READ DMA EXT... Done
Issuing Command => READ DMA EXT... Done
Issuing Command => READ DMA EXT... Done
Issuing Command => READ DMA EXT... Done
Issuing Command => READ DMA EXT... Done
Issuing Command => READ DMA EXT... Done
Issuing Command => READ DMA EXT... Done
Issuing Command => READ DMA EXT... Done
Issuing Command => READ DMA EXT... Done
Issuing Command => READ DMA EXT... Done
Issuing Command => READ DMA EXT... Done
Issuing Command => READ DMA EXT... Done
Issuing Command => READ DMA EXT... Done
Issuing Command => READ DMA EXT... Done

Processing completed, 32 command(s) issued!
Verifying system memory...Completed, No errors found

Program exited normally.
grlib> quit
```

Figure 28: WD-WCANK3855883 Results

6 CONCLUSION

6.1 Conclusions

In general, the project was completed by designing the necessary cores in VHDL and testing these on the board through C. Regarding the project, the ultimate goal was to develop a fully featured AHCI Controller and integrate it to the GRLIB framework which is used by Aeroflex Gaisler. However both due to the complexity of the AHCI Controller itself and the not fully working front end, which had timing issues, the original goal had to be abandoned. This led to the emergence of some demarcations in the project. However in the end, an AHCI Controller was designed, which was able to write to a SATA disk and read back from it. It has been successfully tested with C codes and integrated into the company framework. The entire design is capable of being simulated in Modelsim, although many errors have been encountered and solved during the testing phase on FPGA. This was due to the fact that the SATA device could not be integrated to the simulation and a component had to be designed in VHDL to mimic its operation. Another reason was that the timing errors did not cause problems during the simulation, since the design is not sensitive to increased round-trip delay.

6.2 Future Work

Regarding the future work, the AHCI Controller could easily be extended with additional features. These are mainly not implemented states related to error recovery and interrupt generation. For now, the AHCI Controller does not recover from errors. That is, it will stall its operation and hang if some errors occur. However, there is debug support present through the registers to understand the reason of a possible crush during the operation. These error states can be implemented in a straight forward way following the AHCI specification [4].

Interrupts are required to be generated in case of the reception of some FIS es from the SATA disk during the transfer. The software will catch these during the operation and act accordingly. The generation of interrupts might sound a little bit tricky in the beginning; however it is easy to be implemented, since AMBA protocol has dedicated signals for interrupt generation. This control signal had to be raised for one clock cycle in order to generate an interrupt.

Other than that, Linux driver support has to be implemented and added to the project. But as a prerequisite of this step, interrupt generation has to be handled. As an optional feature, multiple disk support could be added, since the AHCI protocol suggests a way to handle multiple disks through port multiplier controls. Since there are already two SATA connection ports on the Xilinx ML505 board, two different SATA devices could be attached and this feature could be implemented.

Regarding the optimization of the design, the number of asynchronous FIFO s in the transport layer could be reduced to one. One is utilized for sending data to the SATA disk and the other for receiving. Hence these are not used simultaneously. A control logic could be implemented which adjust the usage of this FIFO for the AHCI Controller between the SATA device.

7 REFERENCES

[1] Aeroflex Gaisler AB website. [online] Available at: <www.gaisler.com> [Accessed 03 October 2012].

[2] Adrian Sparrenborn, *Designing a SATA-1 front-end interface in VHDL on a Xilinx Virtex-5 ML505 development board*, 2008

[3] Alexander Miks, *Implementing a SATA disk controller on a Virtex5 FPGA*, 2010

[4] Serial ATA International Organization, *SATA ATA Revision 2.6*, 15-February-2007

[5] James Boyd (Intel Corporation), *Serial ATA - Advanced Host Controller Interface (AHCI) 1.3*, 26th June 2008

[6] American National Standards Institute, *ATA/ATAPI Command Set*, December 11 2006

[7] ARM Limited, *AMBA Specification (Rev 2.0)*, 13th May 1999

[8] Xilinx, *ML505/ML506/ML507 Evaluation Platform User Guide (UG347 v3.1)*, 2008-11-10, http://www.xilinx.com/support/documentation/boards_and_kits/ug347.pdf