MULTIGIGABIT MULTIMEDIA PROCESSOR FOR 60GHZ WPAN: A HARDWARE SOFTWARE CODESIGN IMPLEMENTATION

A Thesis
Presented to
The Academic Faculty

by

Nicolas Dudebout

In Partial Fulfillment
of the Requirements for the Degree
Master of Science in the
School of Electrical and Computer Engineering

Georgia Institute of Technology
December 2008

MULTIGIGABIT MULTIMEDIA PROCESSOR FOR 60GHZ WPAN: A HARDWARE SOFTWARE CODESIGN IMPLEMENTATION

Approved by:

Joy Laskar, Advisor
School of Electrical and Computer Engineering
Georgia Institute of Technology

Gee-Kung Chang
School of Electrical and Computer Engineering
Georgia Institute of Technology

Paul Hasler
School of Electrical and Computer Engineering
Georgia Institute of Technology

Date Approved: 16 November 2008

This thesis is dedicated to Laura Catherine Thomas who supports me in all my choices.

ACKNOWLEDGEMENTS

I would like to thank Dr. Joy Laskar, Dr. Stephane Pinel and all of my colleagues who I have had the pleasure to engage with while working on this project.

A special thanks to Dr. Patrick Melet and John Harding who have offered guidance and insightful feedback. Their assistance has helped me discover solutions.

I would also like to thank Dr. Gee-Kung Chang and Dr. Paul Hasler for serving on my thesis reading committee.

TABLE OF CONTENTS

DEDICATION iii

ACKNOWLEDGEMENTS iv

LIST OF TABLES vii

LIST OF FIGURES viii

SUMMARY ix

I INTRODUCTION 1

1.1 60GHz Multi Gigabit Wireless 1

1.2 Motivation to work at 60GHz 1

II A MULTIMEDIA PROCESSOR 5

2.1 General setup 6

2.2 Software setup 9

2.3 Hardware setup 11

2.4 Hardware and software used 13

III HARDWARE DETAILS 15

3.1 Design flow and software used 15

3.2 System on Chip 17

3.2.1 Using an embedded PowerPC... 17

3.2.2 ...to achieve a precise goal 18

3.2.3 Composition of the system 20

3.2.4 Views in Xilinx EDK 22

3.3 Custom hardware 24

3.3.1 Interface layer 26

3.3.2 DMA controller 28

3.3.3 MAC layer 28

3.3.4 PHY layer 30

IV SOFTWARE DETAILS 33

4.1 Standard installation... 33

4.1.1 Cross compilation 33
4.1.2 GNU/Linux kernel 34
4.1.3 UNIX utilities: BusyBox 35
4.2 ...customized to our needs 36
4.2.1 GNU/Linux driver 36
4.2.2 C executables 37
4.2.3 Network topology 38
4.2.4 JavaScript GUI 39
4.2.5 Simple bitmap toolbox 40
V CONCLUSION 42
APPENDIX A MICROPROCESSOR PERIPHERAL DEFINITION FILE 43
APPENDIX B GNU/LINUX KERNEL CONFIGURATION 45
APPENDIX C BUSYBOX CONFIGURATION 47
APPENDIX D GNU/LINUX KERNEL DRIVER 49
REFERENCES 54

LIST OF TABLES

1 Summary of the hardware used 14

2 Summary of the software used 14

3 DMA Controller - Master signals used to request the bus 31

4 DMA Controller - Master signals used to specify a read operation 31

5 DMA Controller - Master signals used to specify a write operation 32

LIST OF FIGURES

1 Schematic view of the demonstration setup 2
2 Mockup GUI on the transmitter side 3
3 Mockup GUI on the receiver side 3
4 Schematic view of the multimedia processor 5
5 The XUPV2P board 7
6 One radio transceiver plugged on its DC board 8
7 The actual demonstration setup 9
8 An actual GUI screenshot 10
9 Localization of the software stack 12
10 High level view of the system on chip design 19
11 *Bus interfaces* tab of EDK 24
12 *Ports* tab of EDK 25
13 *Addresses* tab of EDK 25
14 *Block diagram* generated by EDK 26
15 Structure of the MAC layer 29
16 Network setup 39

SUMMARY

The emergence of a multitude of bandwidth hungry multimedia applications has exacerbated the need for multi-gigabit wireless solutions and made it out of the reach of conventional WLAN technology (802.11a, b and g).

This thesis presents a system on chip which demonstrates the potential of 60GHz transceivers. This system is based on an FPGA board on which a GNU/Linux kernel has been run. This document will give some insight on the design process as well as on the finished product. Both the hardware and the software parts of the design are presented.

This document is organized as follow. Chapter I presents an overview of the problem to be solved and some insight on the motivation to work at 60GHz. Chapter II gives a high level view of the multimedia processor that has been designed and implemented. Chapters III and IV respectively give more detail on the hardware parts and on the software components of the project. Finally, Chapter V draws the conclusion of this work and presents the future of the work that has been started to enhance this multimedia processor.

CHAPTER I

INTRODUCTION

1.1 60GHz Multi Gigabit Wireless

The Multi Gigabit Wireless research being conducted at the Georgia Electronic Design Center under the leadership of Prof. Joy Laskar involves the transfer of data over relatively short distances at very high speeds. The goal of the project is to achieve ultra high data rates for data transfer between devices such as laptops, handheld devices and hard disks. The frequency of operation is chosen to be 60 GHz due to numerous advantages discussed in Section 1.2.

The group led by Dr. Stephane Pinel has focused on the development of CMOS fully integrated 60GHz multi-gigabit radio chips. These chips have reached a certain level of maturity and I have been put in charge of designing a demonstration setup that shows the potential of it.

My goal was to realize a multimedia processor for 60GHz radio chips. This multimedia processor was to include a whole hardware setup able to physically control the radio chip. A high level view of this setup is depicted in Figure 1. It also was to include a software stack able to control the hardware setup. And, last but not least, it was to include a user friendly graphical interface. Figures 2 and 3 show the first mockups designed for this interface. Given the commercial purpose of this demonstration, this Graphical User Interface (GUI) was very important.

1.2 Motivation to work at 60GHz

In 2001, the Federal Communications Commission (FCC) set aside a continuous block of 7GHz of spectrum between 57 and 64GHz for wireless use. A major factor in this allocation was that it was unlicensed. The lack of requirement of a license to operate in this band attracted its use. Secondly, energy propagation in the 60GHz range is unique. The oxygen molecule absorbs a lot of energy in the 60GHz band. This absorption effect occurs much

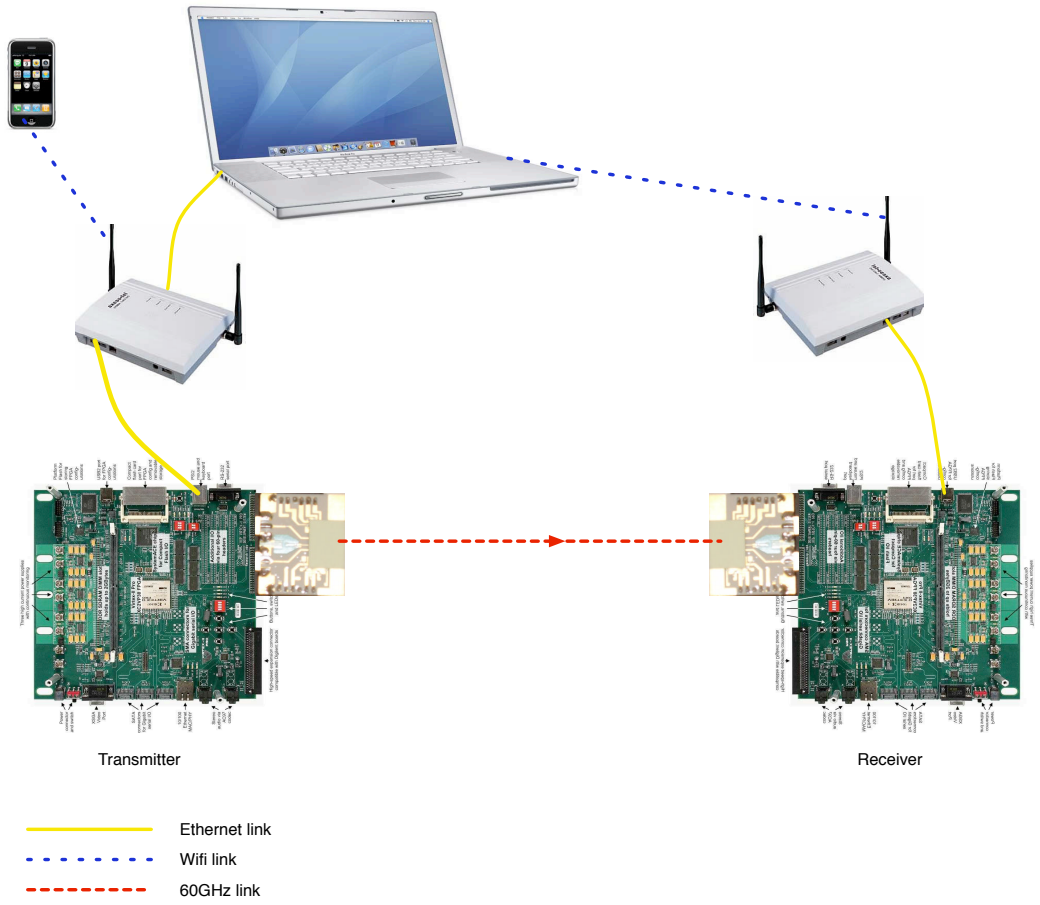

Figure 1: Schematic view of the demonstration setup

more at 60GHz than at lower frequencies used for wireless communication. This absorption weakens the 60GHz signal so that it cannot travel far beyond its intended receiver. As a result of this, radiation from one 60GHz link will not interfere with another 60GHz link in the vicinity. This reduction enables higher frequency reuse. Also, the limitation of their range makes the 60GHz signal far more secure. Thirdly, the propagation losses are proportional to the frequency of operation. As the frequency is increased, the propagation losses also increases. Therefore at 60GHz, high gain antenna array need to be used to compensate for these losses, and as a result the beam width of the antenna is far narrower than at the lower frequency unlicensed bands. The combined effects of oxygen absorption and narrow beam spread result in high frequency reuse, high security and low interference for 60GHz. In addition to this, point to point wireless systems operating at 60GHz have

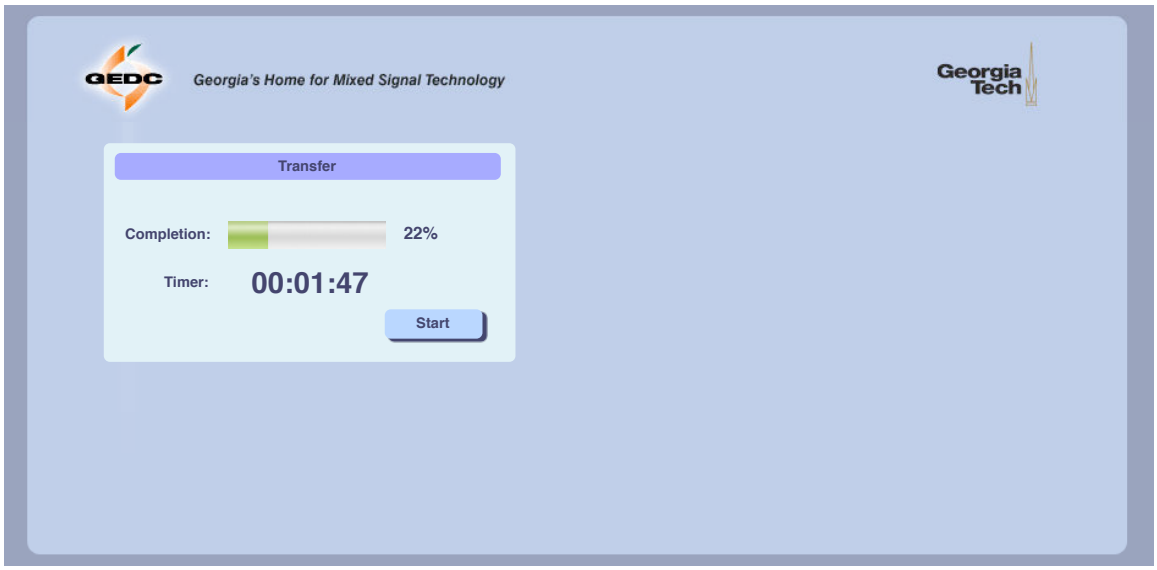

Figure 2: Mockup GUI on the transmitter side

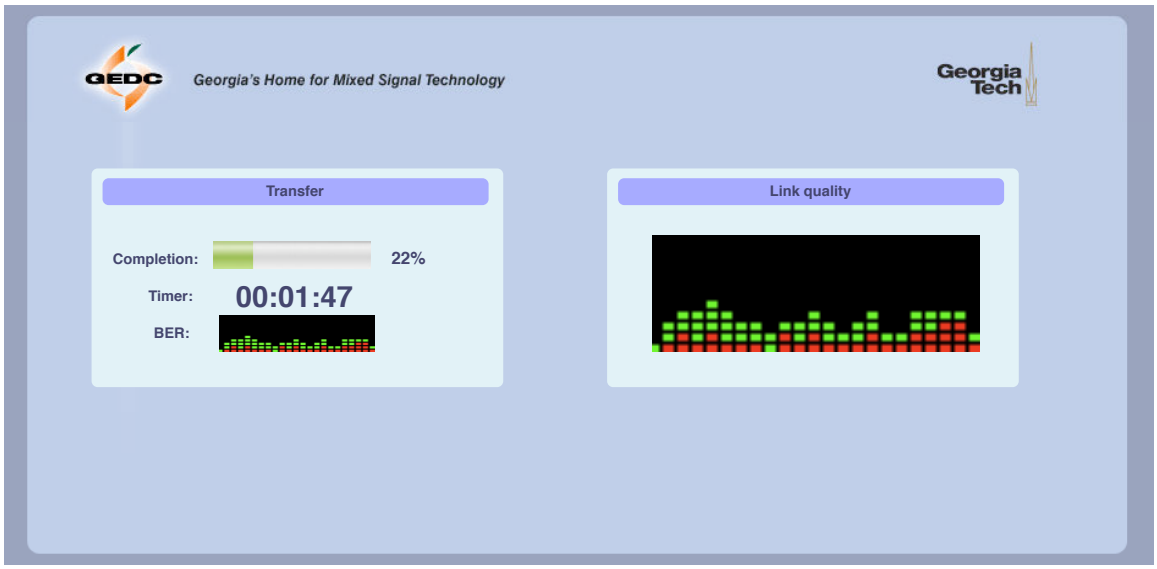

Figure 3: Mockup GUI on the receiver side

been used for many years by the military for highly secure communication. Because of this, a wide variety of components for 60GHz products are available today. Lastly, the data transfer rates achievable by using the 60GHz can be compared to those achieved by fiber optic technology.

We, at GEDC, hope to achieve broad bandwidth and high data rate applications over short distances. The scope of applications for such technology is not limited. These wireless applications could encompass both the office and home environment. The demand for high data transfer using the wireless medium has grown due to the emergence of a number of multimedia applications. Devices such as laptops, external hard drives, MP3 players, cell phones, commercial kiosks and others could transfer large volumes of data in a matter of seconds. A wide domain of video applications can be developed using this technology. For example, a DVD quality movie can be transfer between a laptop and a cell phone in less than 10 seconds. Another potential application could involve wireless transfer of a movie to cell phone from a DVD rental store. Wireless high definition video could also be the result of this technology. Users could keep the DVD player by the side while transmitting wirelessly to a screen five to ten meters away. The innovation of this technology would render USB chords and other such cables obsolete. Photographs from a camera can be sent and stored on the computer using a wireless link. Music can be transferred to cell phones and MP3 players without using any wired connection. Another important area of application would be data centers. Data centers could get rid of the cluster of wires normally associated with servers by the use of such wireless technology. Thus, such applications have been the driving force behind the research and development of this project. Once completed, the range of applications that will be developed is unforeseeable.

CHAPTER II

A MULTIMEDIA PROCESSOR

The solution given to this problem is based on a FPGA board: XUPV2P (Xilinx University Program Virtex 2 Pro) manufactured by Digilent, Inc. [5]. Two of these boards have been used as a transmitter and a receiver board respectively. Each of these boards contains a system centered around an embedded PowerPC processor. This processor runs a GNU/Linux kernel powering among other things a web server. This web server has been used as the base to control and monitor the status of the boards. Figure 4 gives a schematic view of this multimedia processor.

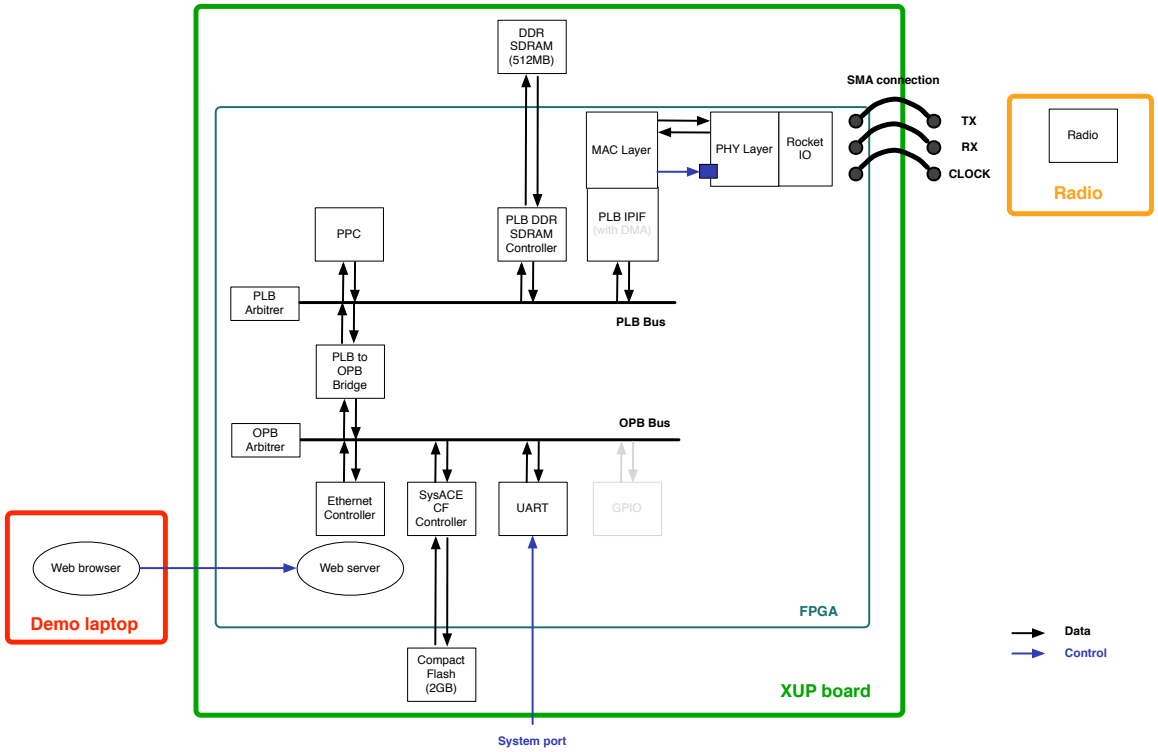

Figure 4: Schematic view of the multimedia processor

This chapter, will give a description of the whole design achieved. It will focus on the process of creation that lead to the solution. The different steps will be described briefly to show the logical progression in the project. This chapter is split in four sections. First,

I will describe the general setup. Then I will give a high level view of the software and the hardware respectively. Finally, I will summarize all the material and software that I have been using during the development.

2.1 General setup

The goal of this demonstration is to prove the concept of a reliable file transfer using 60GHz radio chips. Therefore it is based around two radio transceivers. All the other layers had to be designed and implemented. Given the speed achieved by these transceivers, a software only implementation was not imaginable. Given the wide variety of tasks to be performed it would also be unconceivable to realize it totally in analog. Only two principal ways of implementing it digitally remained: ASIC or FPGA. Being the first project of the Baseband part of the group, and to have a short term demo ready to investors and visitors, we decided to implement it first in a FPGA and then to translate it in an ASIC. This is the classical approach for validating a design on a FPGA before going to the costly and much more constraining ASIC. We therefore decided to implement the lower layers (Physical and Data Link layers) of the OSI stack in a FPGA and the higher levels in software.

When I began work on the project, the FPGA board had already been chosen. I had to try and optimize its usage. This board is the Xilinx University Program Virtex 2 Pro board sold by Digilent, Inc. The board is shown in Figure 5.

This board has many interfaces allowing you to design a system that best suits your needs. It has among others, VGA, PS/2, RS232, SATA and Ethernet ports. It also has slots to put in an SDRAM chip and a CompactFlash. It has onboard an Virtex 2 Pro FPGA manufactured by Xilinx. I discovered that the Virtex 2 Pro has embedded processors that could be used to simplify the implementation of the highest layers of the OSI model. Further investigation led me to the project directed by Dr. Brent Nelson at Brigham Young University [3] focused on installing Linux on this platform. I also found work by Jonathon Donaldson [8] that was inspired by Nelson's project [3]. Donaldson's work provided me with insight when additional material was needed.

In the early stage of development, I had used the Ethernet port. I then began to have new

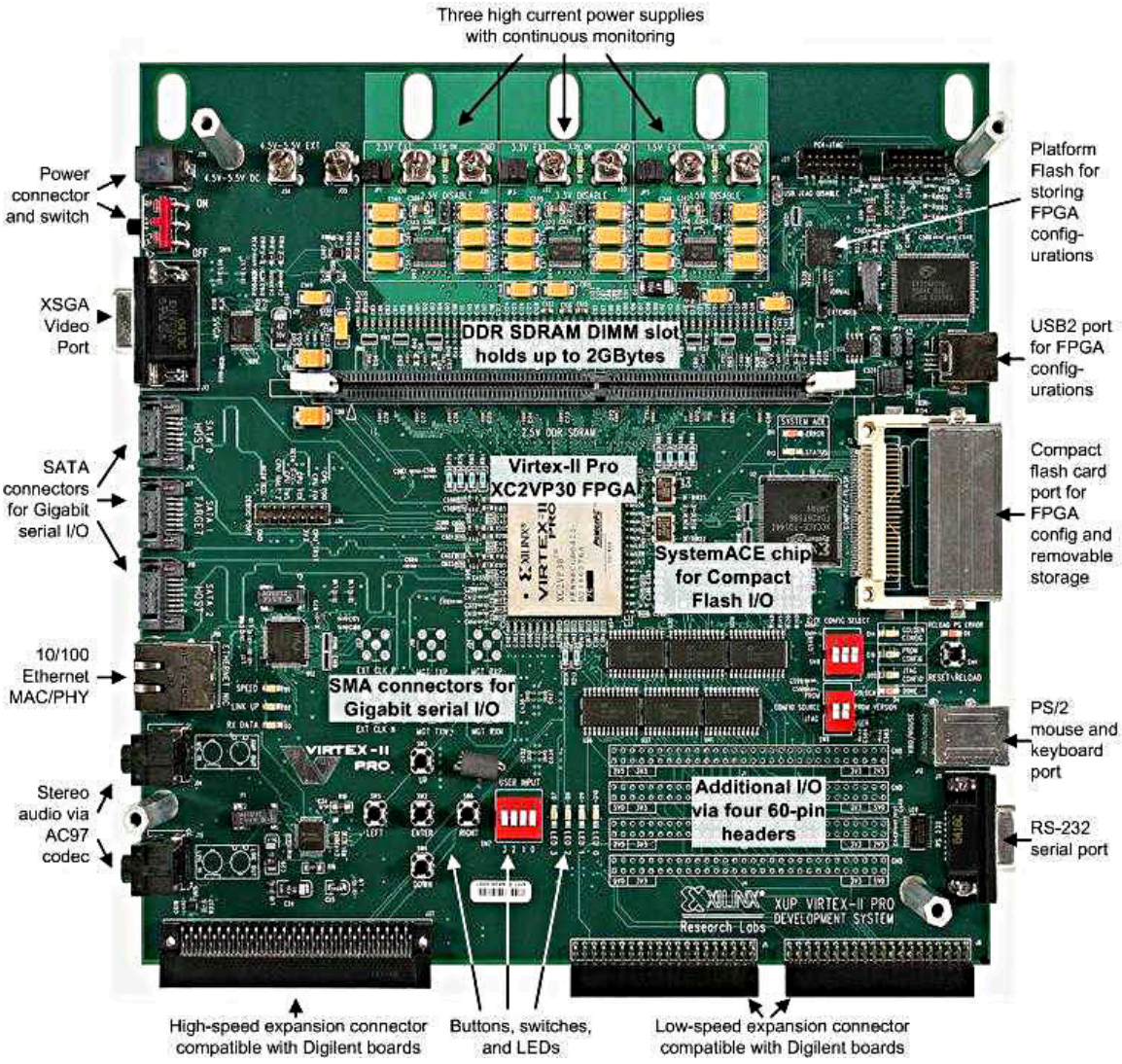

Figure 5: The XUPV2P board

ideas for making a more interesting demonstration. We needed to have some kind of display to show what was happening in the demonstration during the transfers. The first idea that came to my mind, was to have a program running on the board displaying through one of its interfaces such as VGA or RS232 for example, the status of the demonstration. However this solution was not very impressive and therefore not really appealing. Then I had the idea of putting the interface totally outside of the boards. The underlying concept was to separate the information from the display as done in the Model View Controller paradigm of programming. We wanted to have a nice way of displaying the information and controlling the demonstration that could be easily transported over an Ethernet network. I then got

the idea to use some kind of web application and I chose JavaScript as a starting point.

After having thought about different ways of connecting our two boards through a LAN, we came up with the idea of putting them in a WLAN. This dramatically increased the visual effect. The two boards were no longer physically connected. This setup was accomplished by using two WiFi routers and a laptop (MacBook Pro). The routers were used more like wireless switches since their goal was only to provide wireless capabilities to our FPGA boards. The two routers had to be interconnected. We used for that the *Internet Sharing* functionality of Mac OS X. As explained in Chapter IV, the MacBook Pro played here the role of a router by connecting the two subnets.

The last element of the setting was two 60GHz transceivers, one of which can be seen in Figure 6. The modules we used are running on batteries. They are connected to the FPGA boards through SMA connectors.

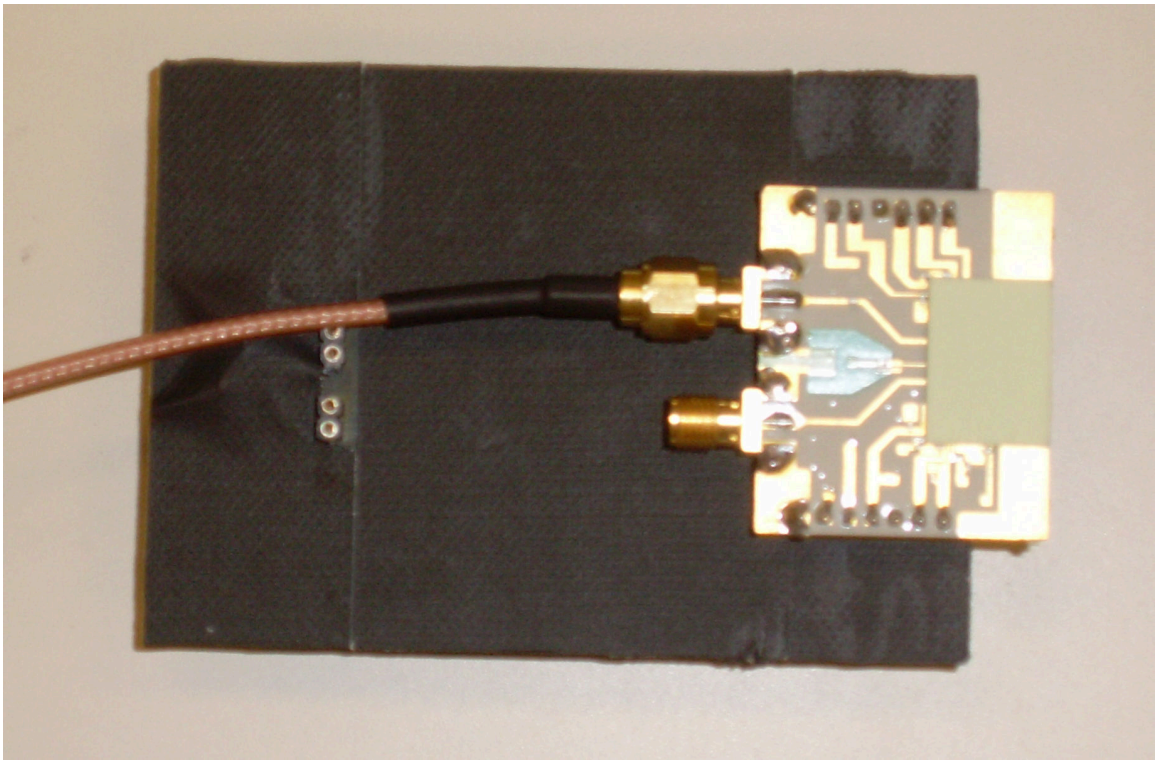

Figure 6: One radio transceiver plugged on its DC board

The whole setup described above can be seen in Figure 7.

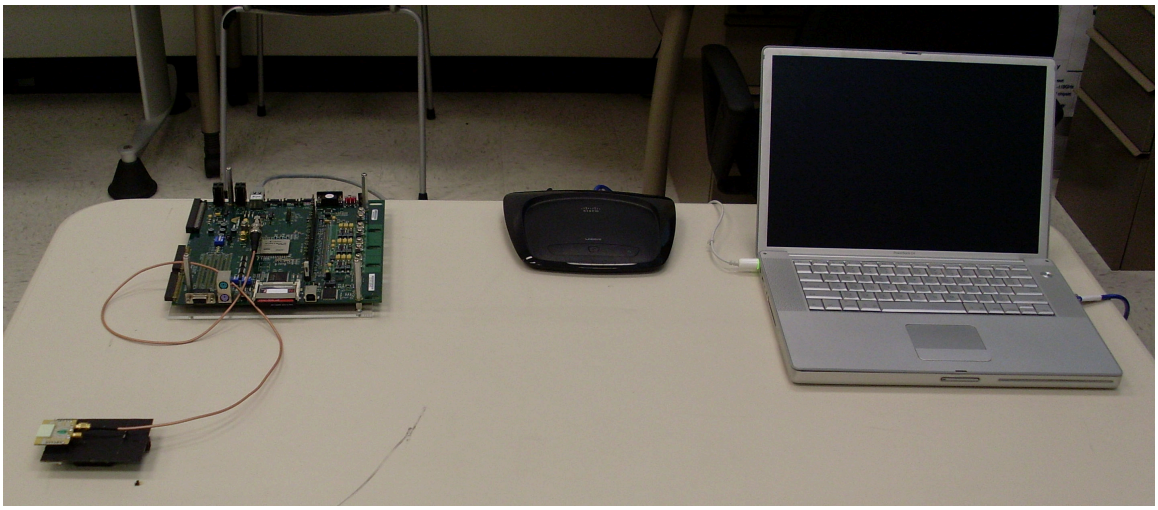

Figure 7: The actual demonstration setup

2.2 Software setup

On the software point of view, the visible part of this demo is the JavaScript based GUI (Graphical User Interface). This GUI is used to monitor the high level status of the two boards and to trigger events in the setup such as "Load file foo.jpg in RAM" or "Start the 60GHz transfer". A screenshot of the final version of the GUI can be seen in Figure 8.

In the first version of this demonstration we have been working in a simplex scheme. This meant that we had no way of retransmitting a corrupted file. For this reason we tried to find a file format that would be tolerant to errors. We thought of bitmap. This allowed us to give a visual idea of errors in transmission. You can easily spot a dropped packet by replacing all the pixels of the packet with a black color. Errors can also easily be discerned as wrong pixels. There was still a little problem with the headers of the bitmap files that are really sensitive to noise. For this reason I built a little suite of tools. It allowed me to strip the header from a group of bitmap files and to combine them. I combined at the same time the remaining part of the files. The headers were sent three times to allow some redundancy and the big part of the image was sent just once. This repetition was an attempt to include some protection of the data but it still did not work when the link was of very poor quality.

This JavaScript application is powered by a web server running on the TX (transmittter side) board. All the information from the RX (receiver side) board are collected by the TX

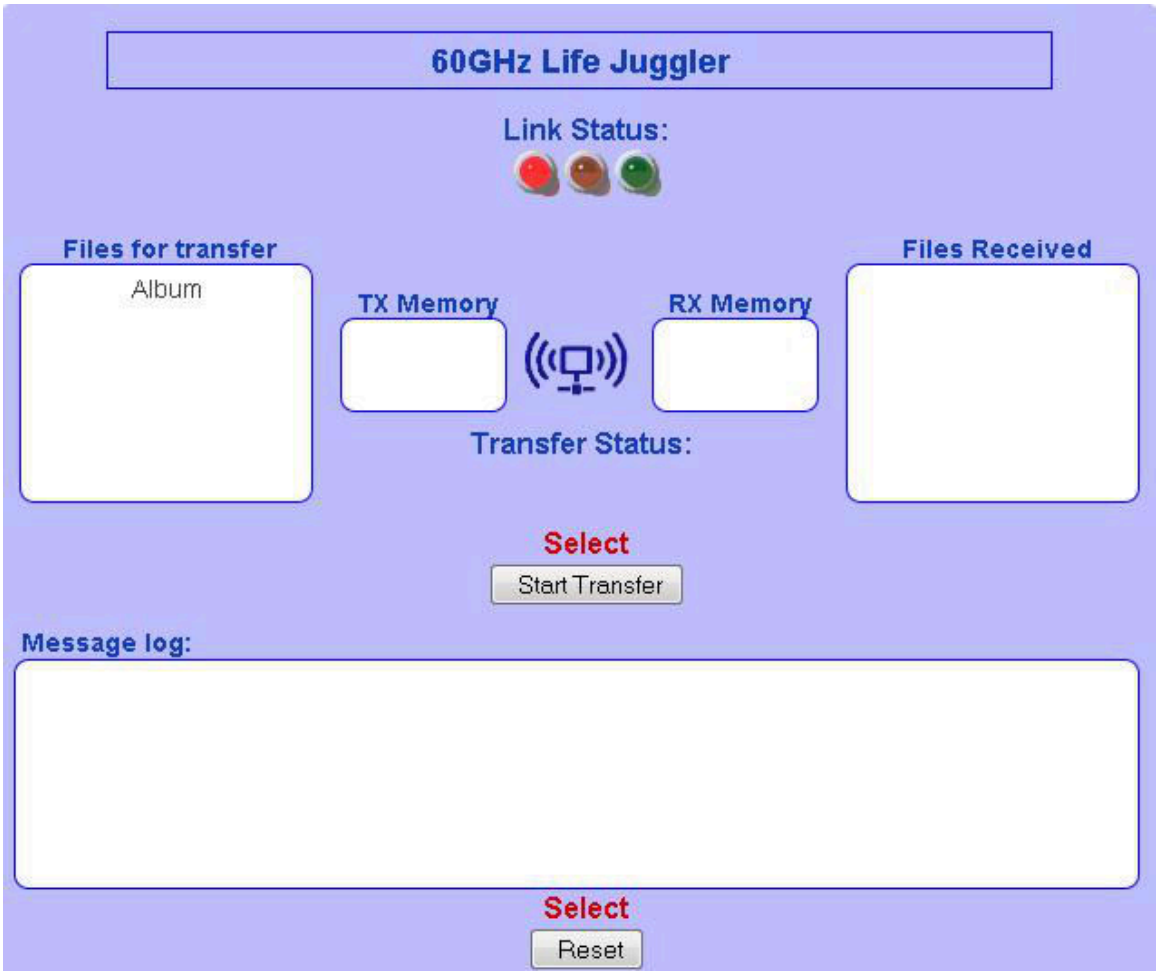

Figure 8: An actual GUI screenshot

board through another web server, running on the RX board.

The main purpose of the demonstration is to do a file transfer at very high speed. For this purpose we wanted the selected file to be located in the external SDRAM plugged on the board. We therefore needed a way to get the file from a persistent storage device to this volatile storage device. One consideration was to use the external CF (CompactFlash) card. We would have pre-loaded files on the card and make a choice at the time of the demo. The only problem with this configuration was the low reading speed of the CF card (4.20Mbps). By using an Ethernet accessed storage device we were able to get a $\times 7$ speed increase (30Mbps). The idea then came to use an NFS server on the MacBook Pro that was already in the setup. This way the MacBook Pro was used at the same time as a router and as a file server.

As seen in the above description, a certain number of high level services are being used to perfom the demonstration: file system, networking, etc. To be able to use these services we had to code them or to find them already coded. The easiest way we found to get this level of service was to install a GNU/Linux kernel on the boards. That is where we used the work done by [3]. The net result of the operation was that I had a GNU/Linux 2.4 kernel running on a PowerPC on top of which I had installed a BusyBox utility.

The kernel is mainly used to enable all the low level functionalities needed, such as use of the Ethernet NIC, standard interface for the hardware I designed. BusyBox, on the other hand, is mainly used to enable high level functionalities such as having a NFS client or a web server.

The aforementioned software stack is depicted in Figure 9.

2.3 Hardware setup

As described above, when I arrived in this project, the XUP board had already been purchased. I started from that to build a demonstration setup that would be able to support all the features required.

As I started to work, I immediately spotted a problem in the way the development was done in the team. There was nothing done to test the design produced. I therefore started writing what I called a debug module. This was nothing more than a message generator on the RS232 port. The purpose was for the design to send signals during its execution and for the module to send them to the user. This allowed the user to follow the execution flow. The limitations of such a design were soon found but it gave me a first insight in the digital hardware creation process.

I then entered the system on chip world. In this world the hardware is closely linked to a software stack. This allows for much greater flexibility. The idea behind it was similar to the one behind the debug module. I wanted to be able to follow what was happening in my complex design. It also allowed me to easily control this same design. This part is the one that took most of my time on this project. This is due to the complexity of the system built and that hardware and software were to be co-developed to reach the goal I targeted.

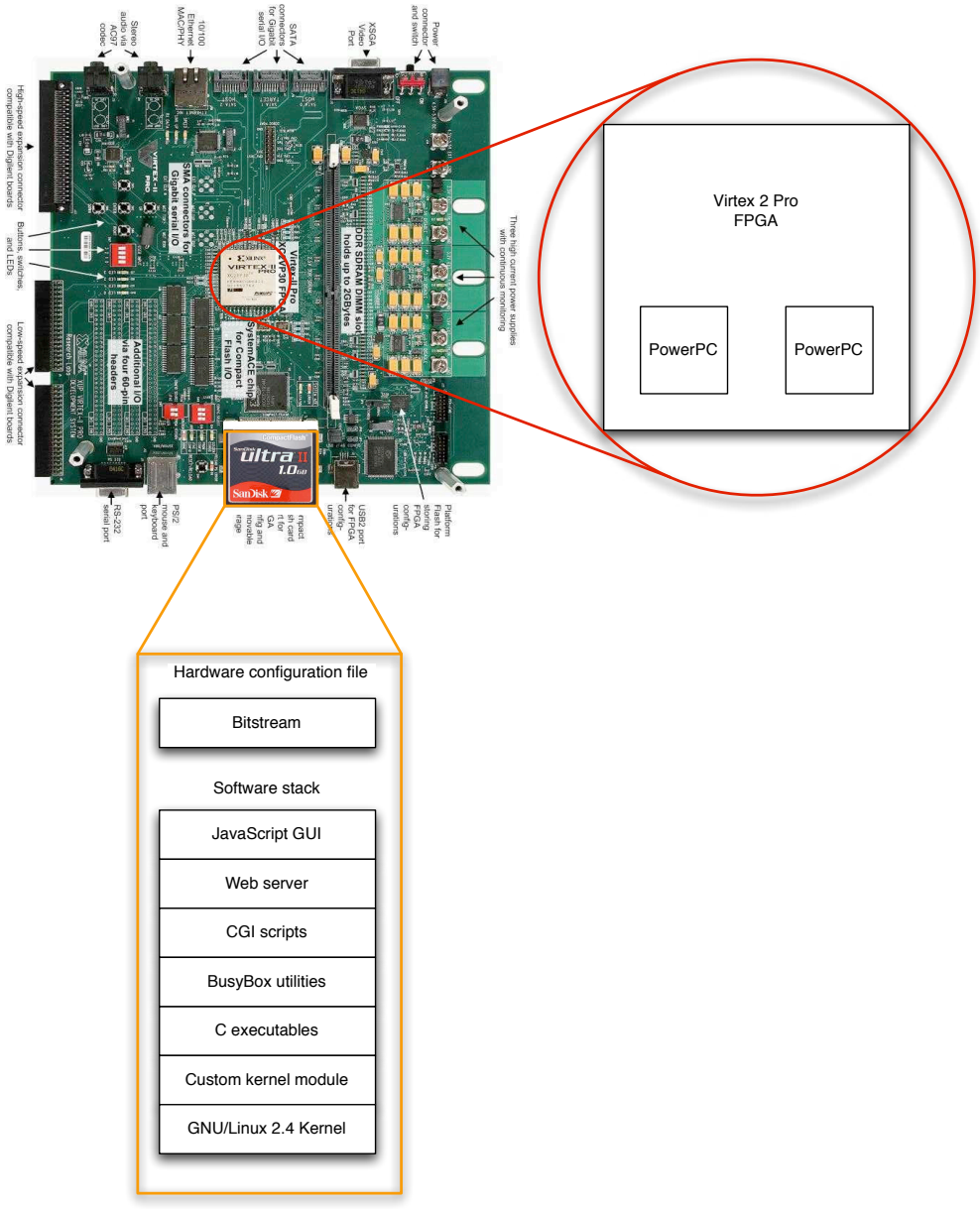

Figure 9: Localization of the software stack

My first steps as a system designer were to put together blocks provided by Xilinx to have a running system. I then started designing my own block that I plugged in this system. I started simple with a General Purpose Input Output (GPIO) device. This gave me some insight on the way to read and write from hardware registers from the GNU/Linux kernel. Given the throughput we aimed at reaching, a GPIO was not enough so I moved to a device located on the PLB (Processor Local Bus) instead.

The idea I had when I started the project was to design my device as a slave on the PLB bus and to use a Xilinx designed module to act as a DMA controller to increase the achievable speed. After a few days of struggling and googling I realized that their module was not working and that it would be easier to design my own DMA controller. My peripheral then became a Master and a Slave on the bus and achieved a throughput of 6.4Gbps that was enough to feed the radio while keeping some bandwidth on the bus for the OS.

In parallel with the design and implementation of this interface layer I have been in charge of designing and testing the PHY (Physical Layer) and MAC (Medium Access Control: Lower half of the Data Link Layer) blocks of our architecture with Dr. Patrick Melet. The implementation has been done by two other Master students: Ankit Khanna and Hemant Sane.

2.4 Hardware and software used

Table 1 summarizes the hardware and Table 2 summarizes the software used for this project.

Table 1: Summary of the hardware used

Component (quantity used)	**Manufacturer**	**Role**
XUPV2P board (2)	Digilent Inc.	Main component of the setup
External SDRAM (2)	Kingston	Volatile storage used by the operating system and by the hardware
WiFi routers (2)	Linksys	Adding wireless capability
MacBook Pro	Apple	File server and router
iPhone	Apple	Display for the GUI
RS232 cable		Brought back debugging information
USB cable		Allowed to program the board and to read the output of the logic analyzer
Desktop	Gateway	Computer used in the development

Table 2: Summary of the software used

Software	**Developer**	**Role**
Emacs [7]	GNU Project	Text editor
ModelSim	Mentor Graphics [9]	Verilog simulator
ISE	Xilinx [15]	Xilinx specific FPGA synthesizer
Chipscope	Xilinx	Xilinx specific FPGA logic analyzer
EDK	Xilinx	Xilinx specific system on chip builder
minicom [1]	Adam Lackorzynski, Martin A. Godisch	A terminal emulator allowing me to read on the RS232 port
Crosstool [4]	Dan Kegel	A set of scripts that came in handy to setup a cross compile chain
GNU utilities	GNU Project	A set of useful tools to program on a UNIX machine
GNU/Linux kernel 2.4	GNU Project	The kernel running on the boards
BusyBox [6]	Erik Andersen	A set of UNIX utilities for embedded devices

CHAPTER III

HARDWARE DETAILS

In this chapter I will describe in more details the tools that I used to design the hardware and the hardware produced.

3.1 Design flow and software used

Any time I had to design a piece of hardware that I wanted included in my system I would follow this flow:

1. **Paper and pencil**: I would try first to write the documentation of the features I wanted for my piece of hardware. Then I would try to break it down in pieces that I could handle more easily. I then would give a functional description of each block newly created. I would usually add to this ideas I had about a possible implementation.

2. **GNU Emacs**[1]: Only then would I start coding. I was taking my time to write the code in the more parameterizable way I could. I was also thinking carefully about the naming of my signals, modules etc. to be able to remember things easily and to interact efficiently with other members in the team. I would then write a testbench for the module I had just written. After having completed this project, I do not think this was the best approach to take. However, given the information I had when I started to work on it, it was better than nothing. For my new projects I will try and adopt a test-driven approach in which the writing of the testbench will be part of my functional description of the block and should allow me to spot problems before the

[1] **I had been using Emacs before but this is in this project that I started using it as my main tool. A powerful text editor (such as Vi or Emacs), in which you are comfortable, is really a must in a designer's toolbox. It allowed me to get my code colored properly as Xilinx editor can but it also indented it. To do that I used verilog-mode [10]. For a simple task, such an editor may seem to be an overkill but you can reuse all that you learned for any text editing. The best example I can give of that is using it to write this thesis in LaTeX.**

implementation stage.

3. **ModelSim**: I would then use ModelSim to test the code I had just written against my testbench. It would generally take a few runs to get rid of the compilation errors and then the real debugging could start. To facilitate my work, to be more productive and more reproducible I started writing scripts to drive ModelSim. This is a process that is time consuming but in my opinion really worth it in the long run.

4. **ISE**: I then used ISE to implement my design in the Virtex 2 Pro. To do so I needed to add generally only a few elements. These elements were mainly used by the Placer and Router to locate a crystal, a certain output etc.. These were given in the form of a User Constraints File (UCF).

5. **ChipScope**: Once the design was implemented it was tested and if the results did not match the simulation ones I would start an investigation with the help of a logic analyzer. Xilinx provides such a tool: ChiScope. This software automates the generation of HDL code to implement a logic analyzer in the FPGA being used[2]. During this step, the results seen on the logic analyzer were compared to the results generated in simulation and the design would be modified until the bug had been found.

6. **EDK**: Then we would pass the design to EDK that is used to generate the final system on chip hardware. Its role is first to help in the design of an adaptation module between a hardware piece and a defined bus interface and then in generating a complete system from all the modules and the processor chosen.

7. **minicom**: As a final step I would connect a serial cable to the board. I would then be able to control the GNU/Linux system on the board through there. I designed a few programs that triggered our design and therefore I could generate test cases and see if I got the expected outputs.

[2]ChipScope's user interface is not adequate at all, as of version 9.2, but given the fast pace at which progress is made on all the other Xilinx software we know it will be improved shortly.

This flow was, of course, not followed linearly at all time. Especially during a debug phase I may have had to use different tools to find where the bug was lying.

3.2 System on Chip

3.2.1 Using an embedded PowerPC...

Let's now describe with a little more detail the system on chip system that we have built. The goal of this system was to enable a smooth and powerful interface between the user and the hardware layers designed by the rest of the team.

To do that we thought about using an OS. This represented a large overhead at the beginning but would pay back when reaching the point where we wanted to design the actual interface for the user. Indeed, once we had an OS installed we could easily choose the form our interface should have taken.

I then had to decide which OS to install. This choice was lead by the available processors in the FPGA. In the Virtex2 Pro XC2VP30 there are 2 hard implemented PowerPC and I could also have used a soft processor MicroBlaze. Having found the work done by Dr. Brent Nelson [3] to install GNU/Linux on the PowerPC I thought it would be a smart choice. This choice was also motivated by the huge amount of software and help available in the GNU/Linux community. Given that it was my first experience in creating hardware and installing an OS on top of it I decided to take a two steps approach:

1. Follow exactly what had been done in [3] and try and get their results on my board. Given that I wanted to expand on their research I however did not apply stupidly all that was written in this tutorial. I tried to understand all that was done and it, of course, paid in the long term.

2. When I had this starting point and this knowledge I started playing around with it to design the exact system that I needed.

As a result my system has a common base with the one described in [3]. However it has some very custom hardware designed just to tail our needs. In the next two sections I will describe the high level view of my system and then the smaller blocks constituting this system with a little insight on the tool used to assemble these blocks.

3.2.2 ...to achieve a precise goal

Before going into the details of the hardware implemented I will give a quick overview of the end goal of the product. This will help understanding all the blocks that have been actually used.

We had to have a hardware setup that could do a two steps process:

1. Fetch some data from a location external to the board. We first thought about using the CompactFlash but it has a really low data rate. In addition, it is not flexible and the card would have had to be reprogrammed to add any file. We then had the idea of using a setup that we had been using during the debugging, namely an NFS server[3]. Therefore we needed to have an Ethernet entry point leading to some internal storage.

2. This internal storage had to be really fast because we wanted it to give its data to the radio. This meant that we wanted a data rate in the order of 1Gbps. We thought about the external SDRAM that we were using on the board due to the GNU/Linux kernel[4]. From this SDRAM we had to transfer the data to the radio at a really high speed. This required three blocks. The first one is an interface on the processor bus including a DMA controller to have the speed required. The second one is a MAC layer that was handling smaller chunks of data to be passed to the PHY layer. The third and last one is a PHY layer doing some encoding/decoding to prepare the data to be transmitted over the air.

Figure10 shows this whole system.

[3]During the debugging process, this server was holding the whole filesystem of the board allowing to recompile it really easily. Indeed, there was no need for transferring the newly compiled binary to the board. We would just have to power it off and power it on again. The kernel was still located on the Compact Flash but once it had finished initializing it would go to the NFS server to find all its executables. This means that any new hardware configuration bitstream and kernel still required to be transferred to the Compact Flash but this happened less and less as the project went. This NFS server made us save an enormous amount of time.

[4]As you can see with the use of the NFS server and the SDRAM in the final product this project matured as it was developed. I used an incremental approach of having one system to work and then adding functionality. This allowed me at each step to see devices that I would maybe never have thought of from the beginning. This has really been an amazing source of learning.

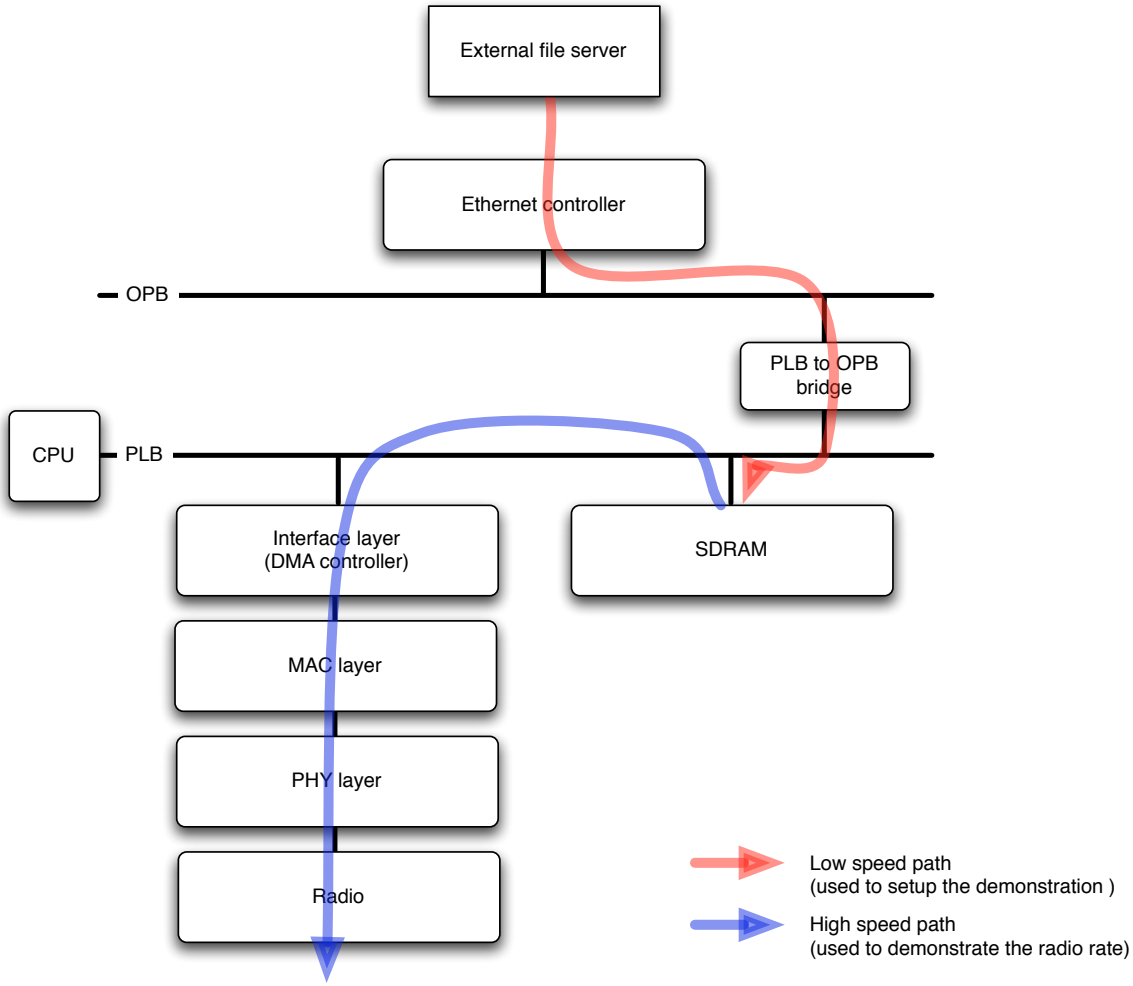

Figure 10: High level view of the system on chip design

3.2.3 Composition of the system

Here are, with a little more details, the blocks that compose our system and their role in the latter:

- **ppc405 (instance 1)**: the processor we use to run our OS;
- **ppc405 (instance 2)**: the second processor on the FPGA that has to be enabled when the JTAG is enabled. And we had to have JTAG enabled because it was the entry point that we used to transfer the root filesystem on the board. It was given from the external CompactFlash. This CompactFlash is connected to the FPGA through a chip designed by Xilinx named SystemACE that uses a JTAG bus to connect the FPGA and the various ports that can be used to configure it (USB, Xilinx specific JTAG port and CompactFlash). This processor is just included to be able to close the JTAG loop to reach the instance 1 of ppc405 but is not doing anything at all;
- **plb_v34**: the controller for the high speed bus on the ppc405, namely the Processor Local Bus (PLB). This bus is used for all the high speed operations but the more peripheral there are on the bus, the slower it becomes;
- **opb_v20**: the controller for the low speed bus on the ppc405, namely the On-chip Peripheral Bus (OPB). The use of this bus is one way of diminishing the congestion of the PLB bus.
- **opb_intc**: the interruption controller of the OPB bus;
- **plb2opb_bridge**: a bridge between the PLB bus that is physically attached to the ppc405 and the OPB bus that is not directly connected to it. It means that when accessing a peripheral on the OPB bus, a request is done on the PLB bus that is handled by the bridge. This bridge then creates another request that it transmits on the OPB bus. This request is handled by the desired peripheral. Its answer follows the reverse path;

- **jtagppc_cntlr**: the JTAG controller that interfaces with the processors. It is used during the initialization process that is as follows:

    1. the hardware design is loaded in the FPGA. The PowerPC is therefore connected to all its peripheral and its instruction side memory is initialized with the program that has been chosen by the user;
    2. in our case we put an infinite loop that maintains the processor in an idle state but not in an erroneous one. When the processor is running this way the JTAG controller can take over it and inject instructions directly [5];
    3. then the JTAG controller effectively sends its instruction that are in our case designed to move the kernel from the CompactFlash to the external SDRAM;
    4. to finish the JTAG controller puts the next instruction of the processor to be in the SDRAM where it has just put the kernel.

- **opb_sysace**: this controller is used to read the data found on the CompactFlash. Its role is to read the design that is stored in the card that has to have been formatted in FAT16. After the Linux kernel has booted, other partitions with other filesystems can be accessed;

- **plb_ddr**: the controller for the external SDRAM;

- **plb_bram_if_cntlr**: a controller used for some RAM internal to the FPGA. This is necessary on top of the external SDRAM because we need a way for the CPU to stay in an idle state as mentioned above and when programming the FPGA only the RAM inside the FPGA is available;

- **bram_block**: the RAM used to implement the instruction side memory of the processor. It is some Block RAM as opposed to some Distributed RAM.

- **opb_ethernet**: the Ethernet controller managing the Ethernet transceiver that we can find on the XUP board. It is plugged on the OPB bus;

[5] This ability to take over the processor, stop it, examine the registers and inject code is heavily used in the debugger installed in EDK

- **opb_uart16550**: the controller for the RS232 port;
- **dcm_module (instance 1)**: a Digital Clock Manager (DCM) used to generate the different clocks used by the processor;
- **dcm_module (instance 2)**: another DCM used to generate the clock used by the external SDRAM;
- **util_vector_logic (instance 1)**: a core that takes two vector operands and bit wise applies a logic function to generate a single vector. It is used here to generate the negative version of the SDRAM clock. It is used instead of the DCM clock because it is working on the 90 degrees shifted version of the clock;
- **util_vector_logic (instance 2)**: same use here for the 90 degrees shifted version of the system clock;
- **util_vector_logic (instance 3)**: same use here again to invert the system clock;
- **proc_sys_reset**: a core used to manage the reset on the processor and to enable or disable certain features;
- **mac_phy**: our custom MAC and PHY layers described in Section 3.3;
- **custom_ctl_bram**: the interface between our abovementioned layers ant the PLB bus. It is described in Section 3.3 too.

3.2.4 Views in Xilinx EDK

These different blocks can be seen from various angles in EDK. This allows the designer to focus on one aspect at a time but to generate anyway a nice consistent system. The system can be seen mainly in four different views:

- **Bus interfaces view**: In this view, each block can be connected to buses. Xilinx provides a certain number of standard buses such as the aforementioned PLB and OPB buses. However, anybody can design its own bus that is just a set of lanes with predetermined names. The advantage of that is that you do not have, as a Verilog

writer, to worry about connecting each single line of your blocks to the bus. With the information it gets from the user, EDK is generating some HDL code to do the instantiation with the correct wire names. To be able to do its job, EDK needs more information from the user than just some HDL code for a module. This is why when you have finished designing a piece of hardware in HDL, you have to *Create a peripheral* in EDK. This step creates a Xilinx specific file named a MPD (Microprocessor Peripheral Definition) file. This file basically makes an HDL-independent (as of today, mainly Verilog or VHDL) representation of the inputs, outputs and parameter of the module. An example of a MPD file is available in Appendix A. Then when the synthesis is requested by the user this file is translated in an HDL (as of today, VHDL) and synthesized. This view for my project can be seen in Figure 11;

- **Ports view**: This view is complementary to the previous one. All of the signals that are not part of a bus can be connected in here. External signals can be assigned to a wire name. Interconnection signals between modules can be assigned here too. These signals are not routed through a bus because such a bus is not defined. It would be a loss of time to properly define a bus for each custom interface designed. This view for my project can be seen in Figure 12;

- **Addresses view**: In this third view the hardware is looked from a higher perspective. Interconnections have been taken care of in the previous two views. Now we have several devices on each bus and we need to assign each of them an address range. This is done in this view. These information will be used in the HDL generation so that a given piece of hardware only answers to requests made to its own address range. This is also used to generate header files that can be used by the software to access the hardware. This way the code written is not dependent on the location of the peripheral on the bus. If the address range changes for any reason, a simple recompilation, with the new headers, is sufficient to keep the code working. This view for my project can be seen in Figure 13;

- **Block diagram view**: This last view, as opposed to the previous ones, is not used

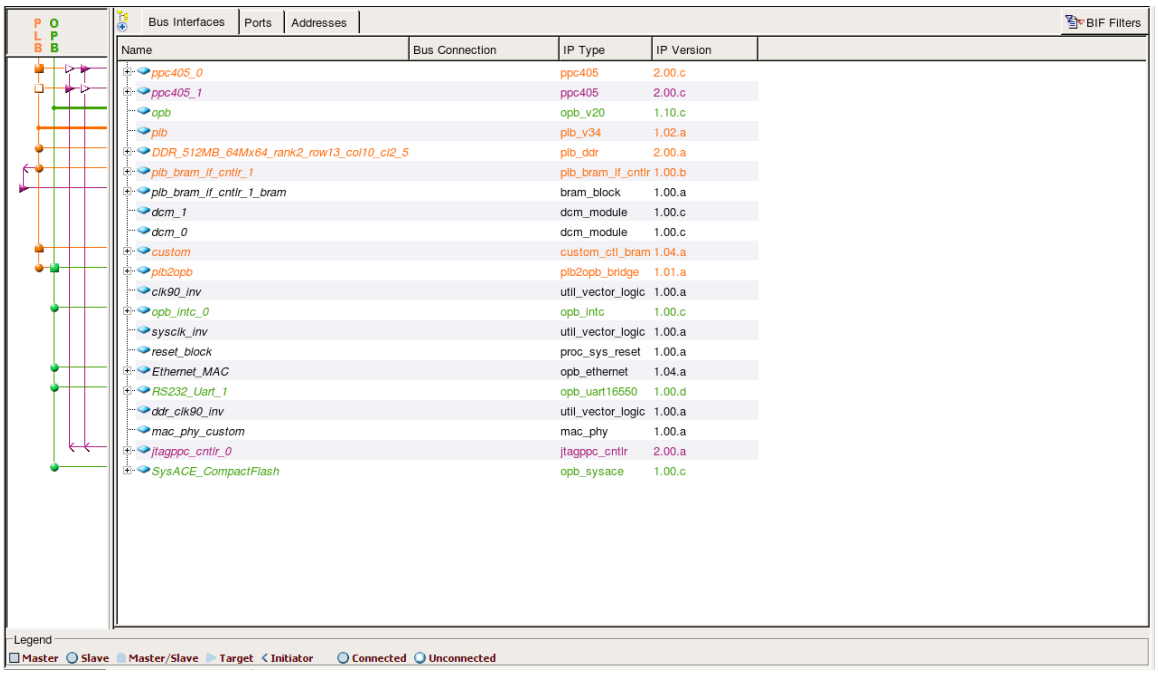

Figure 11: *Bus interfaces* tab of EDK

to design the system but just to have a view of it once it is finalized. This is just a PNG file generated by EDK. This view for my project can be seen in Figure 14.

3.3 Custom hardware

In this section I will describe in a little more detail the custom hardware that has been designed for this project. As explained earlier, it is composed of three parts, an interface layer, a MAC layer and a PHY layer.

I have entirely designed the interface layer between the PLB bus and the MAC layer. I have also worked a lot on the design of the two lower layers but not on the actual coding. While these layers were being coded I was working on the PowerPC side and the interface module. However, I worked once more on these layers during the debug process. This has been a really tough but valuable experience. I have then realized the integration of all the blocks. Most of the design problems that I faced were common design issues. I found a tremendous amount of information in [13] and [14] to cope with these issues.

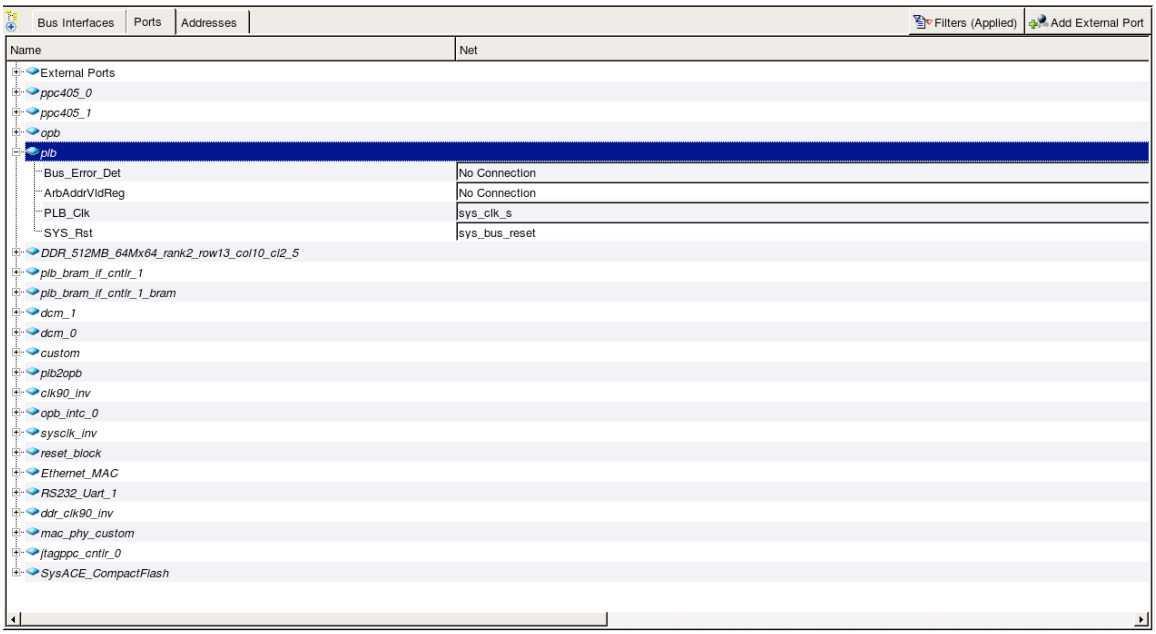

Figure 12: *Ports* tab of EDK

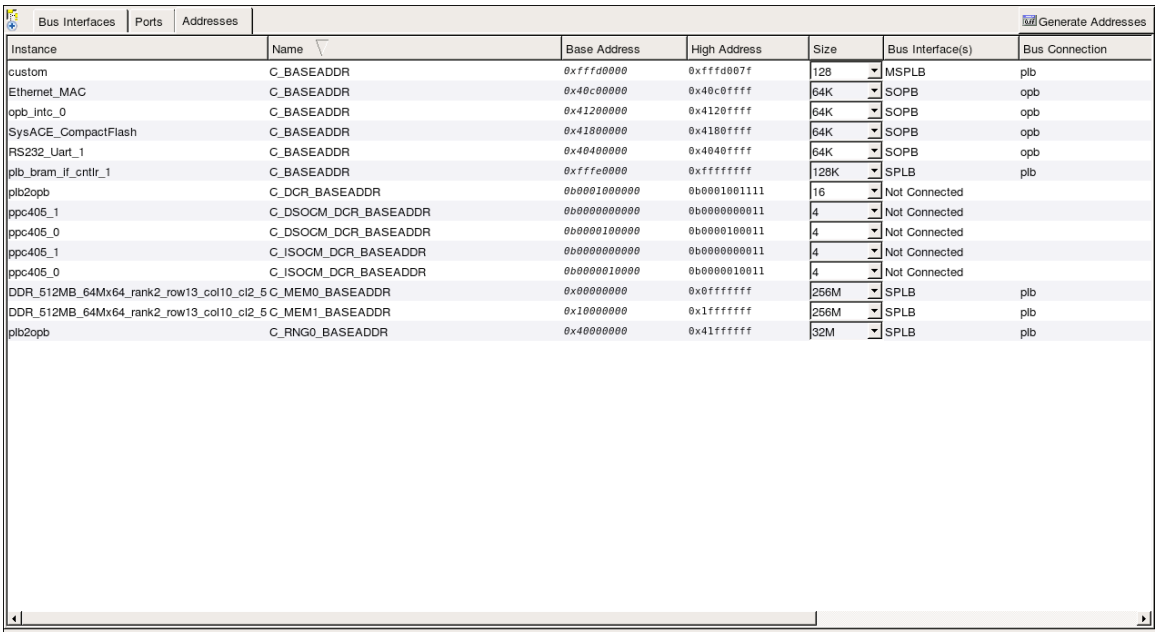

Figure 13: *Addresses* tab of EDK

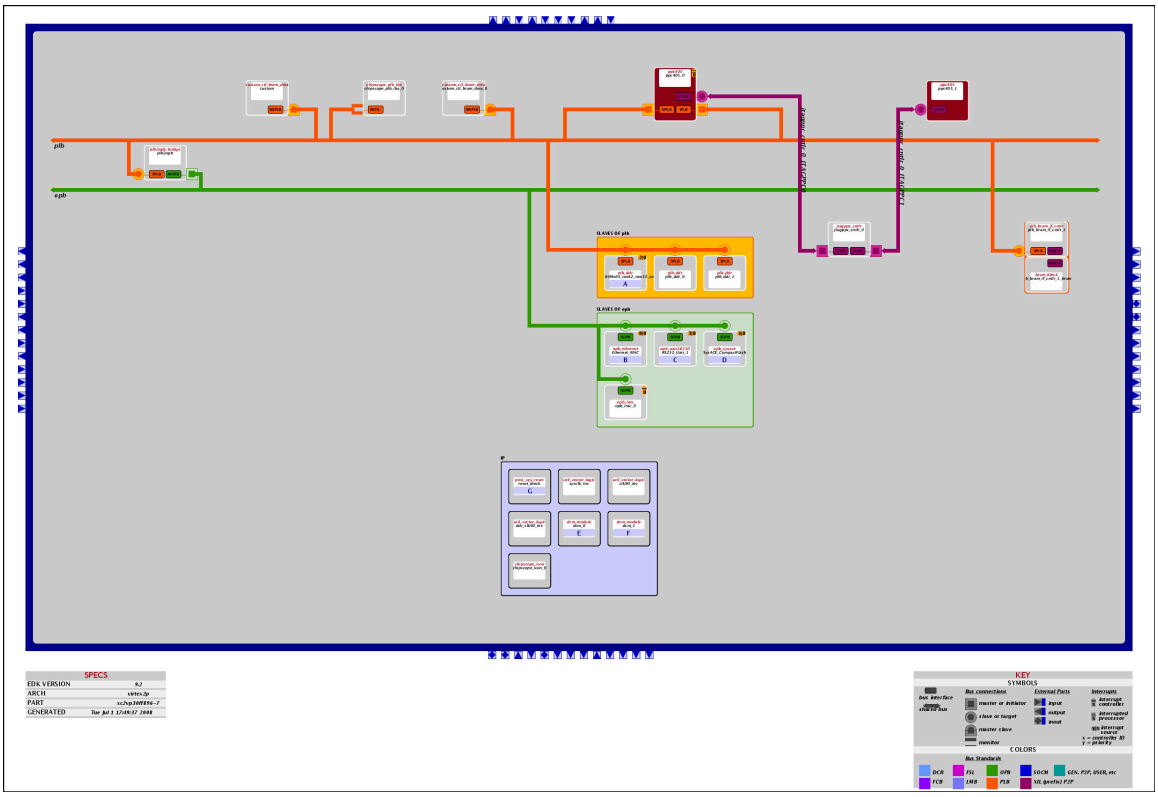

Figure 14: *Block diagram* generated by EDK

3.3.1 Interface layer

The first step was to get the data from the SDRAM. This could not be accessed directly for two reasons:

- the connection to a DDR SDRAM is pretty complicated and needs to be done with extra care and was not at all the purpose of my work. It would have been a loss of time to try and do that;
- this same SDRAM was still used by the GNU/Linux kernel. It had to be accessible to the PowerPC on a standard bus, namely the PLB.

Therefore, the easiest solution was to use what has already been implemented to plug the RAM on the bus. We just needed to get on that same bus and to request data from the RAM.

The first, simple but inefficient, approach has consisted in designing a slave on the PLB bus. This meant that it could not do a request on the bus but could receive some data.

Therefore the request had to be issued by the processor. There was a delay of about 10 clock cycles between the arrival of each 64-bit word. The clock on this bus being 100MHz, we could reach a throughput of 640Mbps that was hardly enough to feed the MAC layer.

We therefore had to use a DMA controller to speed up the process. Xilinx provides one but it seemed to not work properly. After struggling a little with it and having "googled" it, I found out that it was a known issue. The solution recommended by Eli Hughes on the Usenet newsgroup **comp.arch.fpga** on Tuesday, 17 Apr 2007 at 09:03:49 was to design my own DMA controller. It took me a week but it eventually worked. On top of that, I got a $\times 2$ improvement in the performance of the DMA controller by designing my own. If you use the Xilinx one, each data goes from the RAM to the DMA controller and then from the DMA controller to the peripheral, making it transit twice on the bus. [12] came in handy at this point in time where I needed to interact with Verilog code written by others.

To get a peripheral on the PLB bus with Xilinx tools you have two options:. IPIF module or directly plugging to the bus.

The IPIF (Intellectual Property Interface) is doing the translation between the bus of your choice: OPB or PLB, and a standard interface named IPIC (Intellectual Property Interconnect). This IPIC is bus agnostic allowing you to develop only once a peripheral that you can then plug either to the PLB or the OPB bus. For example, this model directly incorporates the concept of registers that you can read from and write to. On top of that you never have to worry about the bus being busy and things like that. Everything is transparent to the user.

This level of abstraction is really valuable as a starting point, but I reached its limit when I had to design the DMA controller. The IPIC helped me once more however because it is one example of HDL code interacting with the PLB bus. It was written in VHDL and not in Verilog. This gave me a hard time in the beginning but was a good first approach to it too. [11] helped me at this occasion to understand the major differences between VHDL and Verilog.

3.3.2 DMA controller

After having understood how it worked it was not that complicated to implement it. To simplify the implementation of this DMA controller, I assumed at first that there were no errors on the slave side or on the bus during the transactions. If this assumption would have revealed to be a problem the design would have been extended to address the error cases.

The signals used to request the bus are described in Table 3. The signals used to perform a read operation are described in Table 4. The signals used to perform a write operation are described in Table 5.

Note: The signal naming convention allows us to determine each signal's direction. The group of letter before the underscore (_) describes the driver of the signal. *M* means that the signal is driven by the master and is therefore an output of the DMA controller. *PLB* means that the signal is driven by the PLB core and is therefore an input of the DMA controller.

3.3.3 MAC layer

In this first version, the Medium Access control (MAC) layer was really thin. It was mostly in charge of doing the clock domain interface between the PowerPC world and the purely hardware world. Its inputs were two blocks of RAM numbered 0 and 1 respectively and control signals. The PowerPC would send a request to transmit to the DMA controller. The latter would then start filling RAM 0. When it would be done it would send a signal to the MAC telling it that a packet was ready and it would start filling RAM 1. When the MAC would receive this signal it would empty the RAM 0 and pass the data to the PHY layer. When RAM 1 would be full the DMA controller would inform the MAC layer of the event and would then go back to filling RAM 0 and the cycle would repeat. The structure of the MAC layer can be seen in Figure 15.

The MAC was therefore principally a state machine reading alternatively in two RAMs. It would also control the PHY layer with the specific signals needed.

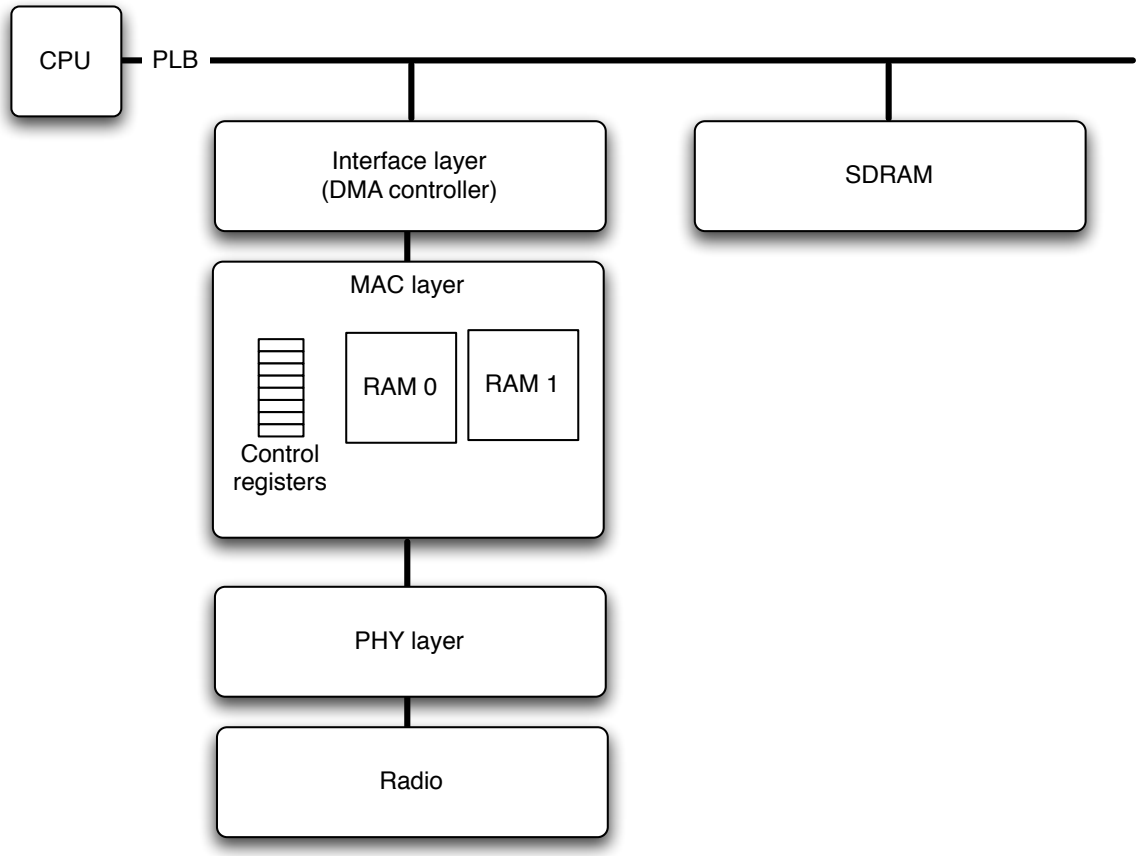

Figure 15: Structure of the MAC layer

3.3.4 PHY layer

The Physical (PHY) layer was slightly thicker in this version but was still pretty basic. It was constructing packets by doing the following operations:

1. a MAC header and a MAC payload are received from the MAC layer. A PHY header is constructed and appended to the beginning of the stream;

2. with this stream as an input, a parity check is built. Every 10 bytes, 2 bytes are added to the stream;

3. this new stream is put through a Reed-Solomon encoder, RS(3,5), that takes 3 bytes in and appends 2 bytes to them to make them 5. With the parity check and the Reed-Solomon encoding the throughput was divided by 2 but the data were protected.

4. before sending the data to the radio, a preamble is appended. This known sequence of data is used by the receiver to align its sequence decoder. Indeed, data are sent in a serial manner but all the computation is performed on symbols. This sequence allow the receiver to find exactly the beginning of the first symbol.

I described above the transmitter side functionality and obviously the symmetric operations happen on the receiver side.

Table 3: DMA Controller - Master signals used to request the bus

Signal name	Use	Value
M_request	This signal is asserted by the DMA controller to request the bus for one transfer. It is deasserted when **PLB_MAddrAck** goes high	n/a
M_ABus [0 : 31]	This signal gives the address at which data is read from or written to in the SDRAM	n/a
M_RNW	This signal sets the direction of the transaction	n/a
M_priority [0 : 1]	The priority of the transaction is always the highest possible	0b11
M_size [0 : 3]	The transaction is always a burst transfer of double words	0b1011
M_BE [0 : 7]	The burst length is never fixed	0x00
M_type [0 : 2]	The transaction is always a memory transfer	0b000
PLB_MAddrAck	This signal informs the master that the bus request succeeded	n/a
M_abort	The DMA controller never aborts a bus request	0b0
M_busLock	The DMA controller never locks the bus because it is only performing burst transfers	0b0
M_compress	The transaction is never compressed	0b0
M_guarded	The transaction is never guarded	0b0
M_ordered	The transaction is never ordered	0b0
M_lockErr	The slave is assumed to not encounter any error	0b0
M_mSize [0 : 1]	The master size is fixed on the V2P to 64-bit	0b01
PLB_MBusy	The slave is assumed to never be busy	n/a
PLB_MRdWdAddr	No line transfer is performed	n/a
PLB_MSSize [0 : 1]	The slave size is fixed on the V2P to 64-bit	n/a
PLB_MRearbitrate	It is assumed that no rearbitration will be asked by the slave	n/a
PLB_MErr	It is assumed that no error will occur	n/a
PLB_pendPri [0 : 1]	The pending requests are not taken into consideration	n/a
PLB_pendReq	The pending requests are not taken into consideration	n/a
PLB_reqPri [0 : 1]	The pending requests are not taken into consideration	n/a

Table 4: DMA Controller - Master signals used to specify a read operation

Signal name	Use	Value
M_rdBurst	Signal asserted when **PLB_MAddrAck** goes high	n/a
PLB_MRdDBus [0 : 63]	Signal sampled each time signal **PLB_MRdDAck** is high	n/a
PLB_MRdDAck	Signal indicating that a new data is available on **PLB_MRdDBus [0 : 63]**	n/a
PLB_MRdBTerm	This signal is not used because the slave is not supposed to request a termination of the burst transfer	n/a

Table 5: DMA Controller - Master signals used to specify a write operation

Signal name	**Use**	**Value**
M_wrBurst	Signal asserted at the same time as the **M_request** signal. It is deasserted during the cycle in which the last data is put on the bus	n/a
M_wrDBus [0 : 63]	Value changed each time the **PLB_MWrDAck** goes high	n/a
PLB_MWrDAck	Signal indicating that the data on **M_wrDBus [0 : 63]** has been read by the slave	n/a
PLB_MWrBTerm	This signal is not used because the slave is not supposed to request a termination of the burst transfer	n/a

CHAPTER IV

SOFTWARE DETAILS

In this chapter I will give a little more details on the software side of this project. I will describe it in two distinct steps. First I will describe the standard part of the install where I followed pretty closely [3]. As I will explain, this has not been as straightforward as it may seem for a big variety of reasons. Then I will focus more on the software that I have wrote to accommodate our specific needs.

4.1 Standard installation...

The install I have performed was basically the installation of GNU/Linux on an embedded platform. To do that, one needs a cross compiler, a GNU/Linux kernel and some tools to make it usable and useful.

4.1.1 Cross compilation

All the development for this project has been done on a GNU/Linux box. This box was a Gateway personal computer powered by an AMD Athlon(tm) 64 X2 Dual Core Processor 5000+. It was running Centos 5.0. The target of the software was the PowerPC 405 embedded in the FPGA. Therefore, there was a need for a cross compiler.

I used crosstool to easily setup this cross compile environment. Crosstool is a set of scripts that automates the construction of a full cross compile chain. I downloaded it and created the latest available cross compile chain. However, it would fail in the compilation of my GNU/Linux kernel. I started taking a look at the errors but could not figure out what the problem was. Indeed, I did not have much experience in the GNU/Linux kernel at the time. I however learned at this occasion that for a certain version of the kernel you need to have a certain version of gcc. These two pieces of software are designed in a really interleaved manner. This has been one of my first painful but truly interesting discoveries. I did not know, however at the time that the list would be so long, and therefore that I

would learn so much.

After having struggled a little and having realized this problem with the versions of gcc and the GNU/Linux kernel, I took an older version of gcc that was the one used in [3].

4.1.2 GNU/Linux kernel

I took, here too, the version from [3]. I used the base kernel they designed and added a few features. I will not detail here all that has been put in this kernel because it was not the focus of this thesis and it is already documented. I will briefly discuss interesting features that I have included in my kernel and that have brought something to the final demonstration. Lines from the *.config* file used to generate the kernel have been included in Appendix B. I will give details on interesting points that correspond to sections in the file:

- **Loadable module support:** This allowed for me to compile my driver as a module and not as a part of the kernel. This allowed me to make changes to the driver and to reload it without having to reboot my whole system. However, being part of the kernel space, this piece of code had to be clean from real bugs else the reboot would be unavoidable.

- **Platform support:** I turned on options allowing me to compile the right parts of the kernel to drive the specific hardware that I had on the XUP board.

- **General setup:** In this region, the address range of the memory layout is being configured. I did not configure that by hand. It was automated by Xilinx EDK but this reside equally in the *.config* file.

- **Parallel port support:** In this section the initial boot option is given. It allowed me to plug a console on the serial port of the XUP, to define the rate of this serial connection, to give the path to the root file system and to specify the amount of RAM that could be used by the kernel.

- **Network File Systems:** This part has been extremely helpful in the development phase and is still used in the final stage. It allowed me to mount an NFS partition on

the XUP board. In the first place I would place my root file system on it to allow me to change it really easily without having to reprogram a CompactFlash. In the final demonstration it mounts a folder residing on the hard drive of the laptop containing the files to be transferred over the air.

4.1.3 UNIX utilities: BusyBox

Having the kernel running on the board was a big step. This meant that all the hardware on the board could be accessed and controlled by software instead of having to write Verilog to interface it. However, a kernel by itself is not user friendly and does not give that big an abstraction to the end user.

This abstraction is usually given by the UNIX utilities that every UNIX, GNU/Linux, Solaris, etc. user has encountered an uncountable number of times. It usually comes "for free" as part of a distribution. In our case we used BusyBox that is an executable regrouping all these utilities mainly of embedded systems. It is called *The Swiss Army Knife of Embedded Linux* and it really acts like it. It is really what allowed us to use the XUP board as a normal GNU/Linux box.

Like the Linux kernel, BusyBox is configured through a *.config* file. I have included lines of it in Appendix C and will describe it here the same way:

- **Build Options:** In this part I informed BusyBox to use a cross compiler and gave him the location of the cross compiler chain.

- **Editors - Finding Utilities:** I installed awk, sed and grep utilities to simplify the writing of scripts to control the hardware at a very high level. The scripts I wrote represented mainly the glue code necessary to make the interface between the user interface and the driver.

- **Init - Login/Password Management Utilities:** These utilities are the one called at the end of the initialization of the kernel and allowed to grant access to a specific user. The format used by the configuration files of these utilities has to be respected really carefully else the system became unusable. I spent a couple of days trying to

figure what had gone wrong in my configuration after I misplaced a colon in *inittab*. This however has been an interesting mistake because it gave me my first opportunity to do some kernel level debugging/troubleshooting. It, in a sense, allowed me to demistify this "scary kernel". Debugging can indeed be done in a similar way as in user land using printk() statements.

- **Linux Module Utilities:** Utilities used to insert and remove easily my module to and from the kernel.

4.2 ...customized to our needs

The work described in the previous Section was a matter of adjusting some standard software to my hardware. It does not mean in any case that it was an easy job which needed no thinking. There was indeed a pretty steep learning curve since most of the information on the subject was given in the form of tutorials that did not explain everything. Instead the approach taken was to give the big steps used to go to the goal without explaining the "why" of each line.

In this Section, however, I will emphasize on the software that I have specifically written for this project. It is composed of:

- a GNU/Linux driver;
- some C executables built on top of this driver;
- the choice of a network topology[1];
- a JavaScript GUI;
- a simple bitmap toolbox.

4.2.1 GNU/Linux driver

The role of the Linux driver has been to do the direct interface to the hardware. I chose to implement it as a character device driver (as opposed to a block device driver) since I

[1] that is not software per se, but is often attached to software. I had to define where my servers and clients would be in order to write my code

wanted to mainly trigger events in the hardware and not read/write from it.

This driver is pretty small, as it should be. It contains the minimum amount of code to be located in the kernel. It is composed of a C file and a header. Snippets from these two files are included in Appendix D.

To give a little more information to the reader wanting to read the code, I will briefly describe the structure of it here. There are two main blocks in this driver:

- **a hardware control function**: this functionality is achieved through the use of *ioctl* to send commands to and get feedback from the hardware. This mechanism is based on the reading and writing of control registers as in most peripherals;

- **a buffer manager**: the role of this block is to put in SDRAM the file to be transferred over the air on the transmitter side and to get it out of the SDRAM on the receiver side. For this purpose, a part of the external SDRAM is left unused by the kernel. I access it directly from the driver by using addresses. These same addresses (not truly identical since addresses are remapped in the kernel) are used by the DMA controller to fetch from or put back to SDRAM the file.

The design of this driver made an heavy use of [2].

4.2.2 C executables

On top of this light layer of register reading/writing I had to implement the actual control of my hardware. The approach I took was to design a toolbox of functions that could be reused in different contexts and from that derive different executables for each task.

For example I will describe the several layers used to implement an executable named link_quality. This executable basically reads a register in hardware that contains a value indicating the quality of the wireless link at any given time and prints it to stdout. The layers look like that:

- link_quality.c: this file makes a call to a function defined in toolbox.c to read the register number LINK_QUALITY_REG (which is just a #define value);

- toolbox.c: this file implements a system call to the driver through an ioctl request to get the value in the register LINK_QUALITY_REG;
- custom.c (the driver): this file then translates this register number to an address and returns the value read at this address.

This decomposition allows a very efficient reusability of the code. Indeed, if I want to read another value in a register I just have to change the number of the register read in link_quality.c. I have no need to know the format of the ioctl request at this level.

This is at this stage of this project that I started using *make* more efficiently. I had heard about it and used it in installing programs but I had never really played with it. I did not use "autoconf" and "automake" at this time. My goal in using the Makefiles was to automate some processes not to make my work available on any imaginable platform. I realized at that time too that I should use Makefiles also in my hardware flow. These kind of files are used by the vendors but they hide it behind their GUIs.

4.2.3 Network topology

One last step was to be achieved to have the whole demo setup. We needed to put the boards in a network configuration that would allow to control all of them from a single interface. However, we did not want to create and LAN and put all the boards on it because it really would not have had the required look at all. It would have seemed that we were trying to cheat and that the transfer was not wireless but through Ethernet.

We therefore had the setup shown in Figure 16. In this setup, each board is on its own subnet and attached to a wireless router. Actually, the routers are not acting as routers per se. They are, in this configuration, just used as switches that can link the wired and the wireless devices. The real router, meaning the device doing the connection between the two subnets, is the MacBook Pro. It is connected through its Ethernet port to the Transmitter side subnet and to the Receiver side subnet through its WiFi card.

The only drawback of this setup is that the MacBook Pro cannot be easily configured as a real router. We have used its built-in Internet sharing capabilities to achieve our goal. It means that we had a unidirectional router. The TX board could see the RX board but the

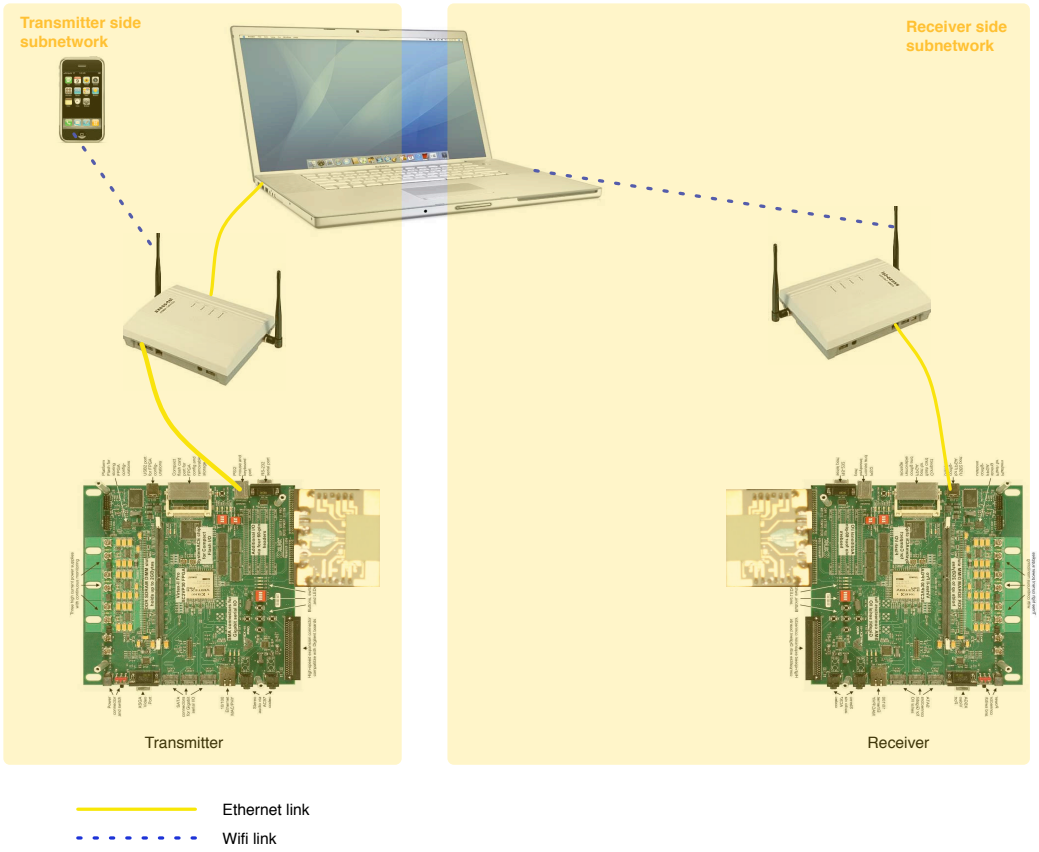

Figure 16: Network setup

opposite was not true. This just means that the TX board had to start all the transactions by sending a request. The RX board could then reply and give information back. This behavior is not the best because the RX board has to give some information back to the TX. The TX has to poll the RX until the information is received instead of the RX just sending a message when it is done.

4.2.4 JavaScript GUI

At this point, the whole demonstration was done from a technical point of view. Indeed, through the use of the command line a file could be transferred over the air received on the other side.

It was however lacking a serious "Wow factor". We needed to do this demo to show that the product could be actually used by anybody. We had to have a GUI to serve this

purpose.

Following the trend of the web applications I decided to do a GUI based on JavaScript. It had the advantage to be "totally" client independent. Any platform with a web browser and a way to connect to our demo could view our application.

The architecture for this application is the following:

- the boards are controlled and monitored by C-based executables;
- these are linked together through ash[2] scripts that trigger the right commands and store the monitored information in a *status.html* file;
- a CGI script is servicing the GUI and launching all the scripts;
- the GUI is "just" sending request to the CGI script and reading the information it needs from *status.html*

The communication between the two boards uses the same channel. It is based on CGI scripts too. The Receiver side that is not controlled directly by the GUI receives its instructions from the Transmitter side through the call of CGI scripts.

4.2.5 Simple bitmap toolbox

With the first version of the hardware that have been produced, some frames arrived damaged and some were missing. Our hardware setup was only a simplex at the time and therefore no retransmission was possible. To overcome this problem we had to transmit some information that was not sensitive to noise. We decided to go for bitmap images. Indeed, in bitmap alterations and frame losses are really easy to spot. It gives respectively funny pixels and black lines respectively.

To do that, we just needed to strip the bitmap header out of the images because this part was still pretty sensitive to noise. Any error in the header made the image look really wrong and it did not give to the user a good impression of the number of bits lost.

To do a file big enough to give a sense of the speed I concatenated 12 pictures of slightly over 4MB each after having stripped the headers off.

[2]ash is the Almquist shell. We used it because it is installed in BusyBox

Since I wanted to be able to change the pictures easily, I developed a small toolbox that can take a folder containing bitmaps (named an *album*) and create a single file with the bitmaps payload (named a *collection*). It can of course also do the opposite operation of producing a collection from an album.

CHAPTER V

CONCLUSION

This document describes a multimedia processor which demonstrates the capabilities of a 60GHz radio. A hardware software codesign approach was taken, as it seemed to be the most reasonable given the clock speed requirements. The solution presented in this document has been designed, fully implemented, and thoroughfully tested.

This system has met its goal. Indeed, it has been presented numerous times in Georgia Tech. It has also been presented in San Diego, in Finland and in France. This insured the reliability of the system. The system is also robust since the hardware was always carried in the baggage hold of the plane and no backup has ever been used for an actual demonstration.

At the time of print, a second version of this design has already been started. The second version brings improvement of several aspects. The main points that will be improved are the automation of the process through the use of a smart hierarchy and the use of Makefiles, the MAC layer to have a duplex communication following ECMA and IEEE standard specifications, and the application layer to illustrate a new generation of use cases enabled by the 60GHz multigigabit wireless technology.

APPENDIX A

MICROPROCESSOR PERIPHERAL DEFINITION FILE

Here are a few lines from a MPD (Microprocessor Peripheral Definition) to give an idea of the syntax being used.

```
##############################################################################
## Name     : custom_ctl_bram
## Desc     : Microprocessor Peripheral Description
##          : Automatically generated by PsfUtility
##############################################################################

BEGIN custom_ctl_bram

## Peripheral Options
OPTION IPTYPE = PERIPHERAL
OPTION IMP_NETLIST = TRUE
OPTION HDL = MIXED
OPTION IP_GROUP = PPC:USER
OPTION STYLE = MIX
OPTION RUN_NGCBUILD = TRUE
OPTION ARCH_SUPPORT_MAP = (virtex2p=DEVELOPMENT)

## Bus Interfaces
BUS_INTERFACE BUS = MSPLB, BUS_TYPE = MASTER_SLAVE, BUS_STD = PLB

## Generics for VHDL or Parameters for Verilog
PARAMETER C_DWIDTH = 64, DT = INTEGER
```

```
PARAMETER MAC_DATA_WIDTH = 8, DT = INTEGER
PARAMETER MAC_ADDR_WIDTH = 14, DT = INTEGER
PARAMETER C_BASEADDR = 0xffffffff, DT = std_logic_vector, BUS = MSPLB
PARAMETER C_HIGHADDR = 0x00000000, DT = std_logic_vector, BUS = MSPLB
PARAMETER C_PLB_AWIDTH = 32, DT = INTEGER, BUS = MSPLB
PARAMETER C_PLB_DWIDTH = 64, DT = INTEGER, BUS = MSPLB
PARAMETER C_PLB_NUM_MASTERS = 8, DT = INTEGER, BUS = MSPLB
PARAMETER C_PLB_MID_WIDTH = 3, DT = INTEGER, BUS = MSPLB
PARAMETER C_USER_ID_CODE = 3, DT = INTEGER
PARAMETER C_FAMILY = virtex2p, DT = STRING

## Ports
PORT tx_rx_mode = "", DIR = O
PORT bram0_full_tx = "", DIR = O
PORT bram1_full_tx = "", DIR = O
PORT bram0_empty_tx = "", DIR = I
PORT bram1_empty_tx = "", DIR = I
<...>
PORT mac_rffu_reg = "", DIR = I, VEC = [(C_DWIDTH-1):0]
PORT PLB_MRdWdAddr = PLB_MRdWdAddr, DIR = I, VEC = [0:3], BUS = MSPLB
PORT PLB_MRearbitrate = PLB_MRearbitrate, DIR = I, BUS = MSPLB
PORT PLB_MSSize = PLB_MSSize, DIR = I, VEC = [0:1], BUS = MSPLB

END
```

APPENDIX B

GNU/LINUX KERNEL CONFIGURATION

Here are a few selected lines from the *.config* file used to generate the kernel that we used on the board. I have not included the whole file because it was too detailed and did not facilitate understanding of the process.

```
# Code maturity level options
CONFIG_EXPERIMENTAL=y
# Loadable module support
CONFIG_MODULES=y
# Platform support
CONFIG_PPC=y
CONFIG_XILINX_ML300=y
CONFIG_UART0_TTYS0=y
CONFIG_EMBEDDEDBOOT=y
CONFIG_VIRTEX_II_PRO=y
CONFIG_XILINX_OCP=y
CONFIG_405=y
# General setup
CONFIG_HIGHMEM_START=0xfe000000
CONFIG_LOWMEM_SIZE=0x30000000
CONFIG_KERNEL_START=0xc0000000
CONFIG_NET=y
CONFIG_SYSCTL=y
# Parallel port support
CONFIG_CMDLINE_BOOL=y
CONFIG_CMDLINE="console=ttyS0,38400 root=/dev/xsysace/disc0/part2 mem=100M"
```

```
# Memory Technology Devices (MTD)
CONFIG_MTD=y
# RAM/ROM/Flash chip drivers
CONFIG_MTD_CFI=y
# Block devices
CONFIG_XILINX_SYSACE=y
CONFIG_BLK_DEV_RAM=y
CONFIG_BLK_DEV_RAM_SIZE=400000
# Networking options
CONFIG_UNIX=y
CONFIG_INET=y
# Network device support
CONFIG_NETDEVICES=y
# Ethernet (10 or 100Mbit)
CONFIG_XILINX_ENET=y
# Character devices
CONFIG_SERIAL_CONSOLE=y
CONFIG_XILINX_CUSTOM=m
# File systems
CONFIG_EXT2_FS=y
# Network File Systems
CONFIG_NFS_FS=y
# Partition Types
CONFIG_MSDOS_PARTITION=y
# Library routines
CONFIG_ZLIB_INFLATE=y
CONFIG_ZLIB_DEFLATE=y
# Kernel hacking
CONFIG_DEBUG_KERNEL=y
```

APPENDIX C

BUSYBOX CONFIGURATION

Here are a few selected lines from the *.config* file used to generate the BusyBox utility that we used on the board. As for the kernel, I have not included the whole file because it was too detailed and did not facilitate understanding of the process.

```
# General Configuration
CONFIG_BUSYBOX_EXEC_PATH="/proc/self/exe"
# Build Options
USING_CROSS_COMPILER=y
CROSS_COMPILER_PREFIX="powerpc-405-linux-gnu-"
# Installation Options
CONFIG_INSTALL_APPLET_SYMLINKS=y
# Busybox Library Tuning
CONFIG_MD5_SIZE_VS_SPEED=2
# Coreutils
CONFIG_CAT=y
CONFIG_CP=y
CONFIG_LS=y
CONFIG_MKDIR=y

# Common options for ls, more and telnet
CONFIG_FEATURE_AUTOWIDTH=y
# Editors
CONFIG_SED=y
# Finding Utilities
CONFIG_GREP=y
```

```
CONFIG_FEATURE_GREP_CONTEXT=y

# Init Utilities

CONFIG_INIT=y

# Login/Password Management Utilities

CONFIG_LOGIN=y

CONFIG_PASSWD=y

# Linux Ext2 FS Progs

CONFIG_E2FSCK=y

# Linux Module Utilities

CONFIG_MODPROBE=y

# Options common to multiple modutils

CONFIG_FEATURE_2_4_MODULES=y

# Linux System Utilities

CONFIG_MOUNT=y

# Networking Utilities

CONFIG_PING=y

# udhcp Server/Client

CONFIG_WGET=y

# Process Utilities

CONFIG_PS=y

# Shells

CONFIG_ASH=y

# Ash Shell Options

CONFIG_ASH_BUILTIN_ECHO=y

# Bourne Shell Options

CONFIG_FEATURE_COMMAND_TAB_COMPLETION=y

# System Logging Utilities

CONFIG_SYSLOGD=y

CONFIG_KLOGD=y
```

APPENDIX D

GNU/LINUX KERNEL DRIVER

Snippets of code from the driver are included here. The whole code has not been included for clarity. The full code is available upon request.

D.1 Header file (custom.h)

```
#ifndef _CUSTOM_H_
#define _CUSTOM_H_

#include <linux/ioctl.h>
#include <linux/types.h>

#define TOTAL_REG_NUMBER 0x8

#define CUSTOM_IOCTL_BASE 0xD0

struct custom_ioctl_data {
  __u32 reg_num;
  __u32 reg_data_lower;
  __u32 reg_data_upper;
  __u32 dma_sdram_file_size;
  __u32 dma_sdram_addr;
  __u32 dma_ctrl_mask;
  __u32 cfs_arg;
};
```

```
#define CUSTOM_MINOR 224

#define CUSTOM_READ_REGS _IOR(CUSTOM_IOCTL_BASE, 0, __u32)
#define CUSTOM_WRITE_REGS _IOW(CUSTOM_IOCTL_BASE, 1, __u32)
#define CUSTOM_CFS _IOW(CUSTOM_IOCTL_BASE, 2, __u32)

// CFS = Custom File System
#define CUSTOM_IOCTL_RESET_CFS 0
#define CUSTOM_IOCTL_TX_UPDATE_CFS 1

#define DMA_SDRAM_ADDR 0x10000000
#define DMA_SDRAM_LENGTH 0x10000000L
#define DMA_BUFFER_LENGTH 16384UL

#define FILENAME_MAX_LENGTH 128
struct file_desc_packet {
  char filename[FILENAME_MAX_LENGTH];
  int file_size;
  int file_packets_number;
  int last_packet_bytes;
};

// CUSTOM FILE SYSTEM
struct cfs_desc_packet {
  int cfs_count;
  loff_t cfs_array[CFS_MAX_FILE_NUM];
};

#endif
```

D.2 Implementation file (custom.c)

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/miscdevice.h>
#include <linux/init.h>
#include <linux/custom.h>
#include <asm/io.h>
#include <linux/fs.h>
#include <asm/uaccess.h>

MODULE_AUTHOR("Nicolas Dudebout");
MODULE_DESCRIPTION("Custom driver for XUPV2P");
MODULE_LICENSE("");

static int custom_open(struct inode *inode, struct file *filp);
static int custom_release(struct inode *inode, struct file *filp);
static ssize_t custom_read
(struct file *filp, char *buf, size_t count, loff_t *f_pos);
static ssize_t custom_write
(struct file *filp, const char *buf, size_t count, loff_t *f_pos);
static int custom_ioctl
(struct inode *inode, struct file *filp, unsigned int cmd, unsigned long arg);
static loff_t custom_llseek(struct file *filp, loff_t off, int whence);
static int custom_read64(__u32 address, __u32 *data_lower, __u32 *data_upper);
static int custom_write64(__u32 address, __u32 *data_lower, __u32 *data_upper);

static struct file_operations customfops = {
  owner:THIS_MODULE,
  open:custom_open,
```

```
  release:custom_release,
  read:custom_read,
  write:custom_write,
  ioctl:custom_ioctl,
  llseek:custom_llseek
};

static struct miscdevice miscdev = {
  minor:CUSTOM_MINOR,
  name:"xilinx_custom",
  fops:&customfops
};

static __u32 remapped_addr_regs;
const static long remap_size_regs = XPAR_CUSTOM_HIGHADDR - XPAR_CUSTOM_BASEADDR + 1;
static __u32 remapped_dma_sdram_addr;
static struct cfs_desc_packet *cdp;
static int
custom_release(struct inode *inode, struct file *filp)
{
  MOD_DEC_USE_COUNT;
  return 0;
}

static int
custom_init(void)
{
  int rtn;
```

```
  remapped_addr_regs      = (__u32)ioremap(XPAR_CUSTOM_BASEADDR, remap_size_regs);
  remapped_dma_sdram_addr = (__u32)ioremap(DMA_SDRAM_ADDR, DMA_SDRAM_LENGTH);

  cdp = (struct cfs_desc_packet*)remapped_dma_sdram_addr;
  cdp->cfs_count = 0;
  cdp->cfs_array[cdp->cfs_count] = BUF_SIZE;

  rtn = misc_register(&miscdev);
  if (rtn){
    printk(KERN_ERR "%s: Could not register driver.\n", miscdev.name);
    return rtn;
  }
  return 0;
}

static void
custom_cleanup(void)
{
  printk(KERN_INFO "%s unloaded\n", miscdev.name);
  iounmap((void *)remapped_dma_sdram_addr);
  iounmap((void *)remapped_addr_regs);
  misc_deregister(&miscdev);
}

EXPORT_NO_SYMBOLS;
module_init(custom_init);
module_exit(custom_cleanup);
```

REFERENCES

[1] Adam Lackorzynski, Martin A. Godisch, "Minicom website." http://alioth.debian.org/projects/minicom. (accessed on 11/2008).

[2] Alessandro Rubini, Jonathan Corbet, *Linux Device Drivers*. O'Reilly, 2nd ed., June 2001.

[3] Brent Nelson, "The BYU Linux on FPGA Project website." http://ccl.ee.byu.edu/projects/LinuxFPGA/. (accessed on 11/2008).

[4] Dan Kegel, "Crosstool website." http://www.kegel.com/crosstool. (accessed on 11/2008).

[5] Digilent, Inc., "Digilent website." http://www.digilentinc.com. (accessed on 11/2008).

[6] Erik Andersen, "BusyBox website." http://www.busybox.net. (accessed on 11/2008).

[7] GNU Project, "GNU Emacs website." http://www.gnu.org/software/emacs. (accessed on 11/2008).

[8] Jonathon W. Donaldson, "Porting MontaVista Linux to the XUP Virtex-II Pro Development Board," Master's thesis, Rochester Institute of Technology, August 2006.

[9] Mentor Graphics, Inc., "Mentor Graphics website." http://www.mentor.com. (accessed on 11/2008).

[10] Michael Mc Namara, "Verilog Mode website." http://www.verilog.com/verilog-mode.html. (accessed on 11/2008).

[11] Peter J. Ashenden, *The designer's guide to VHDL*. Morgan Kaufmann Publishers, 2nd ed., May 2002.

[12] Samir Palnitkar, *Verilog HDL - A guide to digital design and synthesis*. Prentice Hall PTR, 2nd ed., March 2003.

[13] Sunburnst Design, "Papers by Cliff Cummings." http://www.sunburst-design.com/papers/. (accessed on 11/2008).

[14] Sutherland HDL, "Papers by Stuart Sutherland." http://alioth.debian.org/projects/minicom. (accessed on 11/2008).

[15] Xilinx, Inc., "Xilinx website." http://www.xilinx.com. (accessed on 11/2008).