Università degli Studi di Pisa
Facoltà di Ingegneria
Corso di Laurea Specialistica in Ingegneria Elettronica

VLSI-design and FPGA-based prototyping of a buffered serial port for audio applications

Relatori:

Primo relatore	Prof. Luca Fanucci
Secondo relatore	Prof. Pierangelo Terreni
Terzo relatore	Ing. Sergio Saponara

Candidato:
Andrea Marini

Ringraziamenti

Questa tesi é stata svolta durante uno stage di cinque mesi presso StarCore Llc, nella sede di Austin, in Texas. I primi ringraziamenti vanno quindi al Professor Fanucci e all'Ingegner Saponara per aver reso possibile questa esperienza ed avermi assistito da oltreoceano.

La mia gratitudine va poi a chi mi ha seguito indefessamente durante questo lavoro, il mio mentore Andrew Temple: da lui ho potuto imparare molto in quanto a metodo e dedizione.

Una menzione speciale va alla truppa italiana che mi ha accompagnato in questa speciale avventura; grazie per tutto l'apporto dato ad un disgraziato stampellante, per la pazienza alle mie domande e per non avermi fatto morir di fame.

Non ci sono proprio parole per tutte le persone che mi sono state vicine in questi anni, a Scuola, a ingegneria, a ISF, in Erasmus, a Brescia e a Pisa. Se sono arrivato fino a questo punto é soprattutto grazie a tutti voi.

Non dimentico certo la mia famiglia che mi ha sempre sostenuto, accolto e stimolato nelle mie peripezie e nei miei irrequieti ritorni sotto il tetto materno.

un grazie immenso a tutti quanti

Andrea

Contents

List of Figures

Introduction

The present market of semiconductor is very competitive; on one side consumers ask for always increasing performance and new possibilities, on the other companies have to offer low prices in order to be successful. For what concerns performance just think of the wide range of mobile applications, such as PDAs, cellular phones, and laptops : quality of services, duration of the battery and computational power are always taken into account when buying new devices. On the other side, due to the competition, costs have to be very low; this means that both recursive and non-recursive engineering costs have to be kept under control.

Time is another important concern: it is usually true that the earlier a product is presented to the market, the wider share of the market it will gain. This leads modern semiconductor companies to look for viable ways to design improved products in a short time. Because of the complexity of the new electronic systems, this is not an easy task to be accomplished; even tough electronic design automation (EDA) tools have greatly improved in the recent years, a gap still exists between the rate foundries can produce chips and the rate these chips can be designed.

A very common approach to deal with complexity and performance requirements is to integrate as many functions as possible on a single chip (System-On-Chip); this allows higher clock frequency and lower costs. In connection to this also design reuse has spread in a great part of semiconductor world. This means using in your system modules that others have already designed and tested. This allows you to skip some steps in the design flow (at least for those modules) and saving a significant amount of time.

In this framework lies the work of my thesis, developed at the StarCore, a company headquartered in Austin, Texas. StarCore designs and licences Digital Signal Processors as intellectual property; this is basically one of the companies that offer its product to be used in other electronic systems, avoiding licensees to spend time in designing it by themselves.

A Digital Signal Processor is a special kind of processor, designed to execute calculus-intensive applications: encoding and decoding of information, voice synthesis and recognition, compression and decompression of data, Fourier Transform are just some examples. In many systems, thanks to its programmability and its limited cost it is the suitable solution. For example most mobile phones employs a DSP processor to perform base band operation on the signal.

In these kind of systems, it is important that very few cycles are spent doing other than signal processing, such as dealing with peripherals. In the case of an audio signal it is important that the audio port asks for the fewer cycle it is possible. For this reason at StarCore my activity was to design and develop an audio port controller aiming to reduce at least the cycles asked to the processor in case that the algorithm run is frame based.

For this purpose I designed hardware to be mapped into an FPGA, and wrote some software for the DSP; I worked mainly with the Development Board, used to prototype applications based on the StarCore processor.

The first part of my thesis introduces some concepts of the modern semiconductor world that I consider important in order to understand the environment my work was developed. They are System-On-Chip, Design Reuse and Intellectual Property. Besides those concepts I added an introduction to the important technology of Digital Signal Processor.

In the second part of this report there is an introduction to my activity at StarCore: the problem I faced, the resources available, the constraints, an overview of the solution, and the design flow I used.

In the third part there are all the details of the implementation: the architecture of the User FPGA, the details of the audio port controller and some final considerations on the achievements.

In the appendix I considered useful to provide further details on the AHB standard for On-Chip Buses and to show some meaningful part of the Verilog Code I wrote.

Part I

First part

Chapter 1

The System-on-Chip and the Design Reuse

Nowadays System-On-Chip and Design Reuse are two of the dominant paradigms in the digital electronic world; in the chapter I would like to introduce both of them trying to focus on their importance in the present semiconductor world.

1.1 From the Printed Circuit Board to the System on Chip

Integration is a key concept to understand the evolution of electronic system from the very beginning up to now. In the early days every single device was produced by itself and to create a simple system every component was soldered on a board and connected to the others by wires. On the contrary nowadays very large systems are created on a single chip, that is millions of transistors on the same little piece of silicon. This is the result of the integration, and I'll try to explain this process in order to understand the present panorama of the semiconductor world.

The first Integrated Circuit was born approximately in the late 50's, in Fairchild and Texas Instruments where they succeeded in producing resistors, capacitors, transistors and diodes on the same slice of silicon. Thanks to this technological improvement the concurrence of vacuum tubes was beaten and the semiconductor industry stopped to be an uncertain but promising reality to became the ever-growing industry that we know today.

In the 70's Large Scale of Integration (LSI) allowed to produce complex systems such as memories, micro-controllers and microprocessors; the production spread from the U.S. to Europe and other country in East Asia, such as Japan, South Korea and Taiwan.

During the 80's the driving applications were the Personal Computer and other home electronic appliances. The former is important because to it applies the paradigm called Printed Circuit Board (PCB). In fact the best example of PCB is the PC's motherboard: it is a board made of plastic material where the processor, the memory and all the other modules can be soldered and interconnected.

So the Printed Circuit Boards were the ground for electronic systems; they allowed the integration of chips very different from each other, with ad-hoc multi-layered connections. In this way many chips from different vendors and with different functions could be assembled together in a single system.

In the 90's finally came a new degree of integration: the System-On-Chip. As the name suggests, a SoC is a complete system that stands on a single slice of silicon in a single package. Many parts of an electronic system have been integrated, thanks to both the miniaturization of devices and the improvement of design tools. On the one hand this means higher allowed frequencies, lower consumption and less silicon area, but on the other hand there is an increased complexity that causes risks and costs.

To face this problem and to keep the complexity to a manageable size, design reuse techniques were introduced; system designers were given the possibility to integrate previously designed modules. This is the subject of the next section.

1.2 Intellectual Property Cells in Digital Design

The design of a System-On-Chip is one of the most challenging activity in the modern era of semiconductor. Since SoCs offer so many positive features, many industry and research groups are doing many efforts in order to face this problem; in particular one of their goals is to provide designers with automatic tools that guarantee a successful output in a short time and at low cost.

As introduced before, one of the main issues in SoC design is the great complexity of such systems. Just imagine you have to deal with a CPU, a memory, and a number of peripheral in the same project; you could desire to describe them in HDL, simulate them, produce their logic synthesis, always considering them as a single unit. This is clearly demanding and it takes a lot of time.

When SoCs started to spread in the semiconductor world, it was introduced the idea of including in SoC project some parts that had been already designed and verified. For example instead of describing from scratch a micro-controller that you would need in your system you could use a micro-controller already designed by a third-party company. This latter sells you the Intellectual Property of that module, that is usually a RTL model fully tested and verified. By paying this company you get the right of employing their knowledge and you are quite sure that you don't have to worry about the implementation of that module but just on its integration in the system.

One of the most successful example of Intellectual Property design production was ARM; this English company in the early 90's offered to the market a soft micro-controller, that was easily integrated in many embedded SoCs. After this example proved that design reuse was a viable way in the semiconductor industry, many other companies started to provide different blocks with different features.

Intellectual Property vendors basically design their own products, test them, and try to synthesize them on different fabrication technologies. One of the key concepts (and strength) of design reuse and intellectual property is their technology independence; notwithstanding for SoC designers any aid in any phase of the design is welcome, in particular in the logic synthesis.

So nowadays design reuse is very wide spread, there are many IP vendors and many designers integrates more and more massively soft modules in their chips; far from being a direct solution to the SoC design it is anyway a very important help in keeping the complexity low and the prices affordable.

Chapter 2

Digital Signal Processors

This chapter introduces the technology I worked with: Digital Signal Processors. A DSP processor was the core of the system I developed so an overview on this kind of devices is necessary. In the first section I describe the architecture and the feature of the DSP processors; in the second section I provide some methods of evaluating the performance of such processors, while the last section deals with the application design flow, another important key in my work.

2.1 An introduction to Digital Signal Processors

The function of a great part electronic systems is to process the information that could come from the environment or from other similar systems. One possible way to classify them is by their degree of programmability; at the bottom of this classification we find dedicated systems such as Application Specific Integrated Circuits (ASIC). These are designed to perform a single task; it could also be very complex but it will be the same for the whole device's life. Instead on the top we find the processors that with the aid of a memory, could perform any desired task.

At the very top there are the so called General Purpose Processors (GPP) that are the hearth of every PC and laptop. If we go down from there, we immediately find two other kinds of processor: microcontrollers and Digital Signal Processors (DSP). They both are generally simpler than GPPs, but nevertheless they can still do almost everything. If we accept the general division of processor instructions in *control* instructions and *arithmetic-logical* instructions, we could say that the former is specialized in control instructions, while the second is focused on AL instructions.

Both kinds of processor are used in a number of applications, either because the high performance of GPPs would be unused or because the available resources (namely money,power and frequency) are not enough for a GPP. In systems where there are many peripherals and the task are computationally easy the most suitable processor is the microcontroller; if on the other side we require high calculus capacity the straightforward choice is the DSP.

As the name suggests the DSP is particularly suitable for the elaboration of signals, like audio and video signals. DSPs appeared for the first time in the early 80's, when Texas Instruments designed a processor specialized for operations like Finite Impulse Response (FIR) filter, and Fourier Transform (FT). The core of both these operations is the so-called *tap* that consists of a multiply and an accumulate operation linked together. These could have been completed without any concerns by a GPP but it would have taken a huge number of instructions, maybe failing to follow the signal and hence loosing information; consider that at that time no GPP had a multiply unit and most of them had a single access to memory. Multiplication was performed by adding and shifting as many times as the width of the data! Also take into account that a FIR algorithm requires two operands (the signal sample and the response coefficient) each time; this means two accesses in the memory for the data (besides program memory accesses), and given the intrinsic slowness of memories all this leaded a big loss of cycles.

For these reasons two were the main features introduced:

new memory architecture the so-called **Harvard architecture** was deployed, where the processor could access separately and at the same time the program memory and the data memory;

multiply unit in the execution unit a module was dedicated to the multiplication.

The price to pay was an increased complexity and more hardware (more area and more power consumption), but this made a single-cycle multiplication possible and an on-line execution of those algorithms reliable. Notwithstanding those drawbacks, this new system proved to be extremely cost-effective, and DSP's use rapidly spread in the semiconductor world.

In late 80's and in the 90's many other innovations were introduced in the DSPs' architecture: among them the introduction of the Very Long Instruction Word (VLIW), of the Single Instruction Multiple Data (SIMD) and an high degree of parallelism. All these improvements lead to the wide range of DSPs that nowadays we can find in the semiconductor market: from the DSP for high fidelity home audio systems where high precision is required, to the DSP for mobile applications where the limited power and cost are the most important requirements.

In the following I would like to introduce the innovative concepts of VLIW and SIMD that are commonly used in the design of high-performance DSPs.

The Very Long Instruction Word architecture One way to dramatically increase the performance of a DSP is to use multiple execution units (EU) in parallel; this means that you can at first fetch and then execute multiple instructions *at the same time*. This obviously leads to an increased complexity and there are different techniques to deal with it, that are VLIW and superscalar architecture.

The basic problem is to choose which instruction is executed in which EUs; this choice could be done either on-line by the processor itself or by an ad-hoc compiler. The former solution is adopted in the superscalar processors and the price to pay is an increased complexity of the execution part. The opposite approach is found in the VLIW DSP; in this case the load of the choice is carried by the compiler that groups the instructions in *execution* *sets* according to some rules. The positive feature of this approach is that notwithstanding an high degree of parallelism (up to eight instructions in parallel) the complexity (and so delay and power consumption) keeps low.

The Single Instruction Multiple Data SIMD is an architectural technique that could be used within any class of architectures. SIMD improves performance of DSP processor, by allowing it to execute multiple instances of the same operation in parallel using different data.

SIMD operations could be implemented in hardware with two basic techniques; the first is by adding a second set of execution units that exactly duplicates the original set. The second is based on the idea of splitting the DSP execution units into multiple sub-units that process narrower operands.

Both this solutions are used extensively in the TigerSHARC DSP architecture from Analog Devices. This processor for example is capable of executing eight 16-bit multiplications per cycle.

After this overview of the short history and major developments in the DSP field, in the last part of this chapter I am going to point out and other interesting features to provide a complete picture.

efficient memory access As I mentioned earlier the memory architecture is as important as the DSP core; digital processing requires very high memory bandwidth but offers advantages like predictability and very intense code repetition. Therefore while on one side it requires at least two buses, one for instruction the other for data, on the other side it allows intensive use of program cache and particular addressing modes.

data format Most DSP processors use a fixed-point numeric data type instead of the floating-point format most commonly used in scientific application. This means that the binary point is located at a fixed location in in the data word. Floating-point formats allow a much wider range of values to be represented, and highly reduce the risk of overflow; moreover DSP applications require a high numeric fidelity that is difficult to be maintained in fixed-point representation. The reason why the fixed-point is so common is that there are other more important constraints in DSP systems such as cost and energy efficiency. For these goals fixed-point format is far better than its counterpart, being its hardware implementation simpler and cheaper. Another point in data format is the data word width: DSP processors tend to use the shortest data word width that provides the adequate accuracy in their target applications. So most fixed-point DSP processors use 16-bit data words. In such cases in order to ensure adequate signal quality some hardware is added, like wider accumulator register to hold the result of summing several multiplication products.

zero-overhead looping DSP algorithms usually spend most of the processing time executing a short part of the code, that are run continuously, i.e. loops. To improve the performance DSP processors provide special hardware to support looping, in updating and checking the loop counter and in the branching back to the top of the loop.

specialized instruction set DSP processors instruction sets usually present two features: they make the maximum use of the underlying hardware by being highly parallel, and they have quite short instructions. This results in wide and complicated sets of short and specialized instructions. The shortness of each instruction is very important in keeping the amount of the program memory low, but it has heavy drawbacks such as an increased number of instruction and a reduced sets of embedded registers. Thus it is easy to understand that programming such DSP processors is not a trivial task; moreover very rarely high-level language compilers like C compilers produce an optimized assembly code that deploys all the feature of the processors. This is why most of the code of in DSP applications (or at least the most calculus intensive parts) are written directly in assembly language by DSP programmers.

2.2 How to evaluate a DSP processor performance

After this short overview of the architecture and of the features of the Digital Signal Processors I consider important to talk about the way of evaluating the performance of such systems. This is useful for designers in order to choose to most suitable devices given some constraints. DSP processor performance can be measured in many ways; the first is surely the execution time, that is the time taken to accomplish a defined task. Others are the power consumption and the memory usage. The following sections describe different metrics, trying to point out the strength and the weakness of every method.

2.2.1 Traditional Approaches: MIPS, MOPS and MACS

The metric that is traditionally used is MIPS, that stands for Millions of Instructions Per Second. This is a simple counting of the number of instructions executed in a definite amount of time. This kind of measurement is misleading because it strongly depends on the architecture of the DSP; for example a device with powerful execution units could perform a complex mathematical operation in a single instruction. This would cause a low MIPS, because the execution of that instruction could take longer than execution of simpler ones.On the other side a DSP with very simple instructions could score a very high MIPS, even if the actual mathematical operation are very few. Anyway MIPS is valid within the context of a single known processor architecture.

To avoid that confusion, other metrics have been used such as Millions of Operations Per Second (MOPS) and Multiply-Accumulates Per Second (MACS). Nevertheless they both proved to be unfair. The problem with MOPS is related to the difficulty of defining an operation, everyone using its own concepts. For what is concerning the MACS, the point is that even if the MAC is an important operation, in DSP algorithms involve many other operations than this. This makes it not reliable as an overall evaluation parameter.

Anyway neither MIPS,MOPS nor MACS address other fundamental issues in DSP performance evaluation like memory usage and power consumption. Just think of a very high MIPS DSP processor that has slow memory access or of a system with high consumption of memory requiring the use of slower external memory; in both cases the MIPS of the processor would never been deployed since the memory bandwidth would a serious bottleneck. Also the evaluation of power consumption could be an issue since it is strongly affected by the frequency and by the kind of instructions executed.

For these reasons other metrics have been developed in order to provide DSP system designers with more reliable metrics.

2.2.2 Application Benchmarking and Algorithm Kernel Benchmarking

As I said above, in order to take account of every issue, it is more useful to consider a system perspective. In this case a possible approach is the*application benchmarking*; this means running on the DSP system an application or even a suite of applications and measuring time, memory usage and energy consumption. This method has its own limits too; in fact applications are usually written in high-level language like C, and the benchmarking output will also include an evaluation of the compiler or of the programmer's skills.

Hence it's necessary to make one step down to reach the most suitable DSP performance evaluation method, that is algorithm the *algorithm kernel benchmarking*. Far from being perfect and complete, it is nevertheless the most common method used to compare different DSP devices and architectures.

Algorithm kernels are the building blocks of most signal processing systems and include functions such as fast Fourier transforms, vector additions and multiplications, and filters. These have some compelling advantages such as ease of specification, optimization and implementation; this means that they could be identified easily, implemented in a short time in assembly code and you can be quite sure that your implementation is close to the optimum one. Thus a set of algorithm kernels has been selected and it is commonly used to evaluate the performance of the each DSP.

Typical examples of benchmark suites are offered by the EDN Embedded Micro-processor Benchmark Consortium (EEMBC), that could also be used for DSP Processors. EEMBC develops different suites of benchmarks for different fields, such as consumer, networking, telecom, automotive and office automation. Each suite is composed of a range (five to sixteen) of individual benchmark tasks representative of that product category; for example in the consumer category you could find algorithms for image compression, image filtering, color conversion and other tasks common to consumer digital imaging products.

In this way companies can have a standard evaluation of their processor, that could be compared with those of their competitors.

2.3 How to Develop an Application for a DSP Processor

After this introduction of the DSP processor, the next step is to describe how to develop an application on a DSP platform. This is more closely related to my activity at StarCore; I will underline the role of the Development Board that is the platform I worked with.

As you can see in picture 3.1 the beginning step is the definition of the requirements of the system, and the definition of the algorithm to be implemented. These greatly affects the choice of the processor and the architecture of the system.

Once the device has been chosen, the designer is provided by the DSP vendor with a suite of software tools that are really helpful in the development; these includes a compiler, a linker, a DSP processor simulator and a profiler. The latter is used to understand which part of the program is more

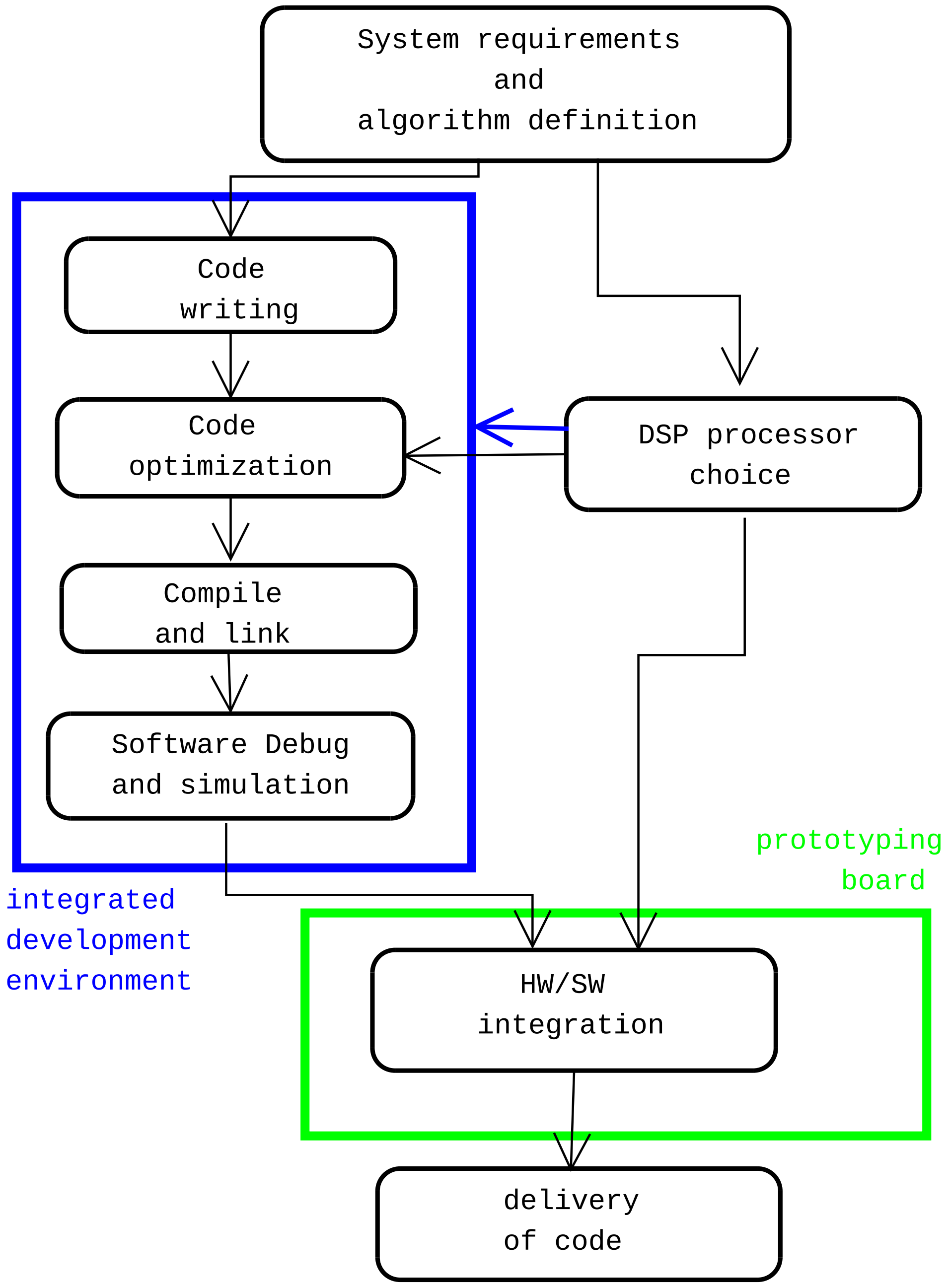

Figure 2.1: DSP application design flow

intensively run, that is the part the programmer should optimize best.

The processor simulator is a great tool too; it allows the program to be run and performance evaluated. It has some drawbacks: first of all it's very expensive to be built and secondly it is easily overcome by an hardware simulation in terms of speed and computation resources.

For these reasons -where possible- the hardware simulation is preferred; it could be done thanks to a Prototyping Board, where a system similar to the target one could be created. It is a Board delivered by the DSP vendor that could be customized by the DSP programmer in order to test the program and the whole system in the proper conditions.

On this kind of boards there is the DSP processor, so the code could be downloaded from the PC where it has been developed and made it run on the actual platform. An important feature that is usually on the same chip as the processor is the Emulator, that allows the programmer to control the flow of the program from the IDE, doing an efficient debugging. In general using the Prototyping Board allows to test in a quite easy way not only the DSP core but also all the systems resources like memories, caches, buses and peripherals. For these reasons this is a necessary step in the development of applications especially for such a particular systems like DSP systems.

This is an overview of the design flow for a DSP application; in my work at StarCore I mainly focused on this las part, since I worked on the Development Board DB1000 that is the Prototyping Board used for the StarCore processor. A deeper description of this board is in the next chapter, together with a complete introduction to my work in StarCore.

Part II

Second part

Chapter 3

Introduction to the system

The activity is basically the design and the implementation of a controller for an audio port. An audio port is a simple device to collect and play audio signals; on one side you have a microphone or a speaker, on the other you have digitally represented samples. This kind of device is used in every applications where sound is involved; for example a mobile phone has an audio port to record and reproduce the voice in a conversation. Like many other peripherals the audio port communicates to the CPU of the system with the interrupt mechanism. This is a typical method of communication but not always it is suitable for the correct deployment of the system resources. This is particularly true in a DSP system, where every cycle should be spent on algorithm processing, and where algorithms usually are frame-based. Hence in my work at StarCore was the need for a module - a *controller* - between the audio port and the DSP processor, that implements a buffer mechanism, in order to interrupt the core just once per frame instead of every sample.

So the first section of this chapter tries to provide an overview on possible solutions in order to deal with peripherals in a DSP environement. On the other hand the section after describes the actual system I worked with, that is the Development Board, an important tool already named in the previous chapter. Then as a conclusion of this chapter there is a section dedicated to the constraints I had in the implementation of the controller.

3.1 Analysis of the Problem

As briefly introduced above the communication between a DSP processor and the peripherals of the system could be an issue for the designers and the users. In particular there is the risk of loosing all the advantages of a DSP processor if the peripherals are not managed in an appropriate manner.

This could happen when the processor elaborates sets of values all together and not single samples; this is the case of some audio algorithms and most video and picture processing operations. If you think at the video case it is easily understandable that the elaboration is usually done on the complete frame and not on the single pixels.

So if the processor is asked to stop at every incoming samples just to transfer it in the internal memory, this would be a great lost of cycles. Every cycle spent by the processor in tasks other than signal processing is considered *overhead*, a necessary price that you try to minimize; so the main goal of my activity is to reduce the overhead in the communication between the DSP and the audio port.

Even if the interrupt mechanism is a very used one and it works very well in most of cases, sometimes other solutions can guarantee far better performance; the Direct Memory Access (DMA) is one of them: it is a kind of controller that is capable of transferring data in and out between memory and peripherals interrupting the processor just once.

With the picture 4.1 I would like to provide a comparison between these two possible solution, showing that for what concerns the overhead reduction the DMA proves very effective. This picture describes the number of cycles necessary in different configurations to perform an algorithm on a buffer of data, and all the IO operations to feed the buffer; it's worth noting that the total number of cycle spent in the actual algorithm (the light blue section) is always the same, no matter which configuration you consider. That number in the picture is 10k, so whatever is beyond the 10k tag is to be considered overhead.

The first case described is the simplest one; the communication method is the interrupt mechanism: basically when a sample in the peripheral is ready a signal is sent to the processor. This signal stops the processor and make the Interrupt Service Routine (ISR) run; the ISR transfers the data from the peripheral registers to the internal register. Once this transfer is finished the DSP goes back to the execution of the algorithm. As you can see in this case the overhead added by the IO operations is very high.

The second case considers the implementation of a buffer in a module near the peripheral; the processor is interrupted just once per frame, but in that case the ISR (clearly different from the first case) would be very cycle hungry cause it would be in charge of transferring all the data. In this case the total overhead introduced is slightly fewer than the previous case cause the transfer of a big amount of data could be somehow made faster.

The last case is the system which deploys the DMA. A controller is near the peripheral and it sends every incoming samples to the internal memory using the Direct Memory Access feature. Hence none of the transfer operations is performed by the processor and for this reason the overhead is very low.

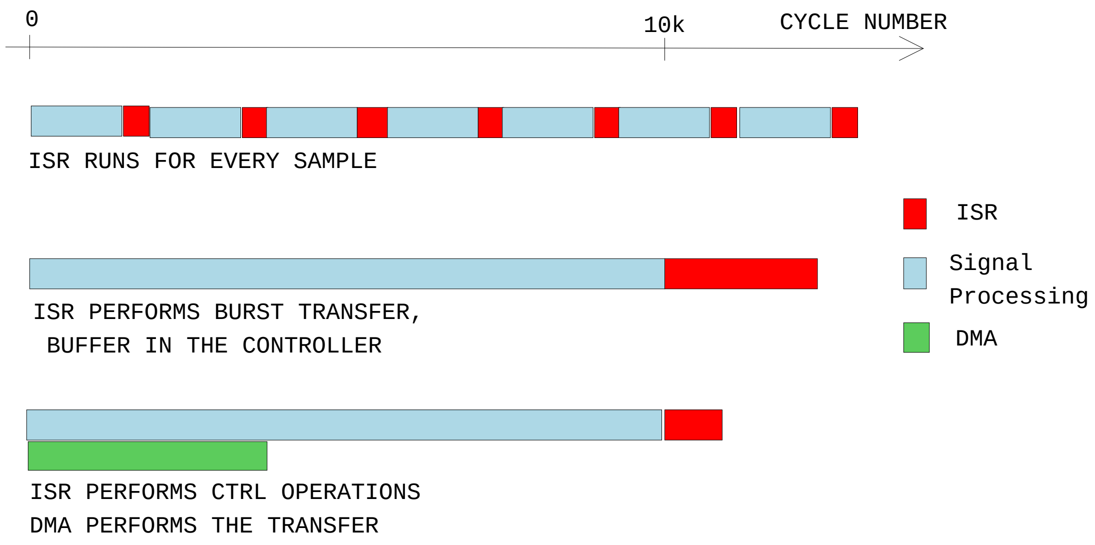
0
10k
CYCLE NUMBER
ISR RUNS FOR EVERY SAMPLE
ISR
Signal Processing
DMA
ISR PERFORMS BURST TRANSFER,
BUFFER IN THE CONTROLLER
ISR PERFORMS CTRL OPERATIONS
DMA PERFORMS THE TRANSFER

Figure 3.1: Overhead introduced in a) sample based ISR system ; b)frame based ISR system ; c) DMA based system.

Being my goal the reduction of the overhead the best solution is clearly the last one that involves the use of the Direct Memory Access; this is the actual solution implemented during my activity.

Before stepping to the description of the DB1000 it's also important to point out that DMA and DSP operations could be not always completely parallel, as it may seem at first sight. The DSP and the DMA share the memory, and they can access it only one at a time. In the StarCore architecture the shared memory (the SRAM) is dual port; considering that the areas in the memory accessed at the same time by the DMI and the DSP are different (different buffers) , this make the two operations completely parallel.

3.2 The Resources

The environment -and basic resource of my project- is the Development Board, that, as described in the previous chapter, is an important tool in every design flow based on a DSP platform. I will describe the architecture of this Development Board, and all the functions and possibilities that it offers.

In this kind of environment there are basically two kinds of resources: the first is software, that is the code that is run on the processing unit, the second is the hardware that is mapped on the programmable logic (FPGA) available in the system. Every design activity involves the use of one of them or both; in particular mine is mainly focused on the hardware side, being the software almost untouchable due to constraints that are explained below.

The Development Board is a Printed Circuit Board that helps customers in testing their applications on the StarCore platform; in particular the DB1000 hosts the Test Chip TC1000, with the one version of the Core and the Subsystem.

On the board there are the most common communication plugs like the USB2.0, the Uart, JTAG, and an ARM Integrator Connector that would allow a connection to an ARM processor. Moreover there is a Flash memory for the beginning configuration and a 256MB SDRAM bank plus the on-chip SRAM. You can see an overview in picture 4.2.

A further degree of customization is provided by the programmable logic with the form of Field Programmable Gate Arrays (FPGA). These are digital chips, that contains a number of logic gates that could be organized to perform any desirable function. There are two FPGAs on the board; the first is the *Control FPGA*, that hosts all the controllers of the peripherals and it is entirely programmed by StarCore before shipping the board. The other is the *User FPGA* that is delivered blank for the customer to use it and enrich the board as he desires. This latter has the connection to almost all the peripheral, so every customer could decide to change or implement other controllers. For example a third-party company developed a multimedia demonstration employing the User FPGA.

Third-party softwares are provided to develop the applications; once the program is compiled in the desired way, it is downloaded via USB in the internal memory where it runs controlled by the JTAG On-Chip Emulator.

The Test Chip Due to the kind of business carried by StarCore there is usually no chip designed or sold to customers; the only exception is the Test Chip, that contains the DSP core and other modules, and it is used by customers only for prototyping new applications. On the DB1000 the Test Chip is the heart of this system; besides the core other modules that are important to be very close to the core itself. In particular we can find:

224 KB SRAM it can be accessed at the same time by the Core and by the DMI;

8KB Data Cache between the Core and the external memory;

8KB Program Cache between the Core and the external memory;

DMI DMA Interface, that stands amidst the DMA bus and the SRAM

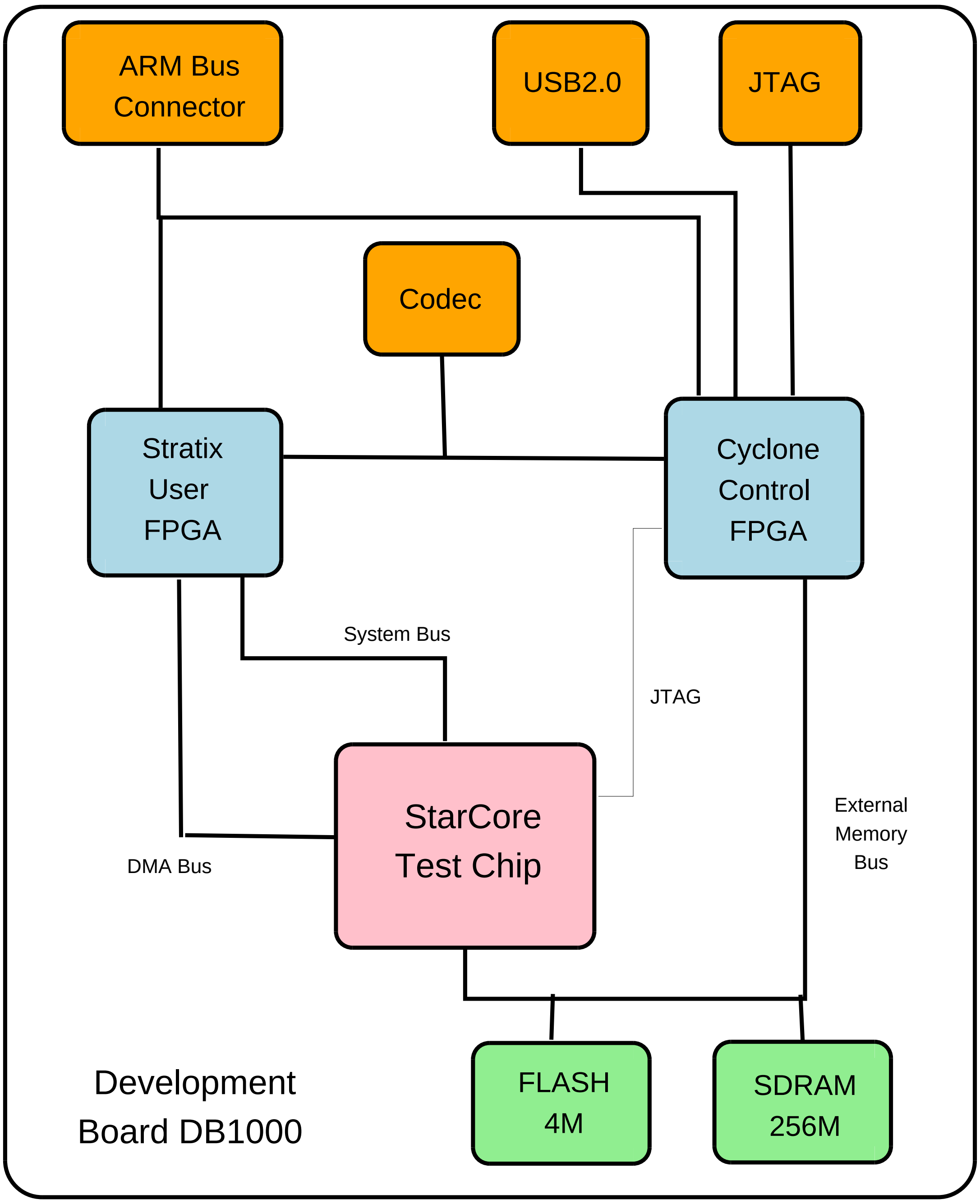

Figure 3.2: Overview of the DB1000

OCE On Chip Emulator; it communicates via the JTAG to the external world and make it possible to debug the system.

SBI System Bus Interface; it is the interface to the System Bus that is extended off-chip.

This board could act as a stand alone system, with its own operating system or as a single threaded system; it could also act in cooperation with other elaboration systems such as a Personal Computer or other micro-controllers. In these different cases, different solutions are provided for the communication; for example in the PC case, a dual port RAM implemented on the Control FPGA over the USB2.0.

Let me underline the importance and the connection of the buses that stand between the core and the FPGAs; these are the three buses: Off-Chip Memory Bus, the Off-Chip System Bus, and the Off-Chip DMA Bus. The OMB and the OSB are basically off-chip extensions of the internal buses so they share the same 4-Gbyte memory space. The OMB connects the chip with the external memories and with the Control FPGA, that has its own range of addresses. A similar role has the OSB that connects the TestChip with the User FPGA.

On the other side there is the ODB, used to access from the User FPGA the internal SRAM, via the DMA Interface. Both the OMB and OSB are Advanced High-Performance (AHB) buses, a standard for on-chip buses developed by ARM that is described in detail in the appendix A.

3.3 The Constraints: the Software Compatibility

To complete this introduction to my work I am going to speak about the constraints I was given. These basically come from the software side of the system.

If you develop a software application and you need to use any peripheral, you'll surely use the driver of this peripheral that is provided to you by the dealer of the platform. This driver usually comes as a set of functions that you have to run at the setting time of the system, and as another set of functions that you have to use in order to communicate with the peripheral.

All these functions are called in the application and have to be included as library files in the project, already compiled or as high-level language code.

In the case of the audio port peripheral there was already a controller and its driver and they were used by many applications such as demonstrations for customers. From this fact comes that even if the controller was changed, applications should still work. This means that the modifications should be transparent to the application. In other words the implementations of the functions composing the driver could even change, but their definitions (that is the interface toward the upper level) must remain the same.

This constraint poses some limits to the design possibility of my activity; in particular I had to keep all the memory variables as they were and could basically change just some initializing functions and part of the Interrupt Service Routine. Details about these modifications are in the software chapter.

The next chapter will deal with the design flow that I followed in order to complete the work. Some further details on the implementation of the new controller will be given.

Chapter 4

The Design Flow and the Solution Description

After the overview of the problem provided in the previous chapter, it's time to describe the solution implemented with further details and the method used to design it.

4.1 The Design Flow

Let's start with a description of the design flow. With the aid of the picture 5.1 I try to provide a clear picture of the whole work.

Analysis of the pre-existing solution The first step of the work is surely the analysis that was previously adopted for the audio port controller. This means a study of the audio port integrated circuit and of the pre-existing controller, provided as a Verilog described module, and all the software driver. This is the important to understand what could be still used (if anything) and what should be discarded.

Analysis of the available resources In parallel to this I studied the system in order to understand which where the resources available for the new design, in particular the architecture of the Test Chip and of the Development Board

System Solution Thanks to both the previous steps I was able to propose a solution involving all the aspects of the final project. It was an high level solution where the main concerns was to partition the tasks to be implemented among hardware on the User FPGA and software running on the DSP. Once this division was clear I was able to work on both side almost independently.

Hardware design After this partition I designed the architecture of the User FPGA, that was previously blank. For this work I used a top-down approach, from the overall architecture down to the implementation of the single modules. A particular attention was paid to the design of one of these modules, the audio port controller, as described in the last section of the chapter 7.

Software design By knowing the definitions of the hardware in the User FPGA I was able to modify or to write from scratch the functions that were to be changed for the new controller.

Test of the integrated solution Particular attention was given to the test phase, that involved both hardware and software sometimes at the same time. Some tricks were used in the hardware design in order to make this phase easier. As described by the red lines in picture 5.1, a constant feedback was necessary on both hardware and software design.

In all this work I was aided by powerful hardware and software tools, that proved necessary for the overall success.

On the software side I used the DSP Development Environment CodeWarrior by Metrowerks; with it I wrote the code (both C and assembly), downloaded it on the Board and debugged it thanks to the On-Chip-Emulator.

On the hardware side I used the Quartus Environment by Altera, that allowed the development of HDL code to be mapped on the Stratix User FPGA. Thanks to this suite I was able to follow the typical design flow for Hardware FPGA programming that is:

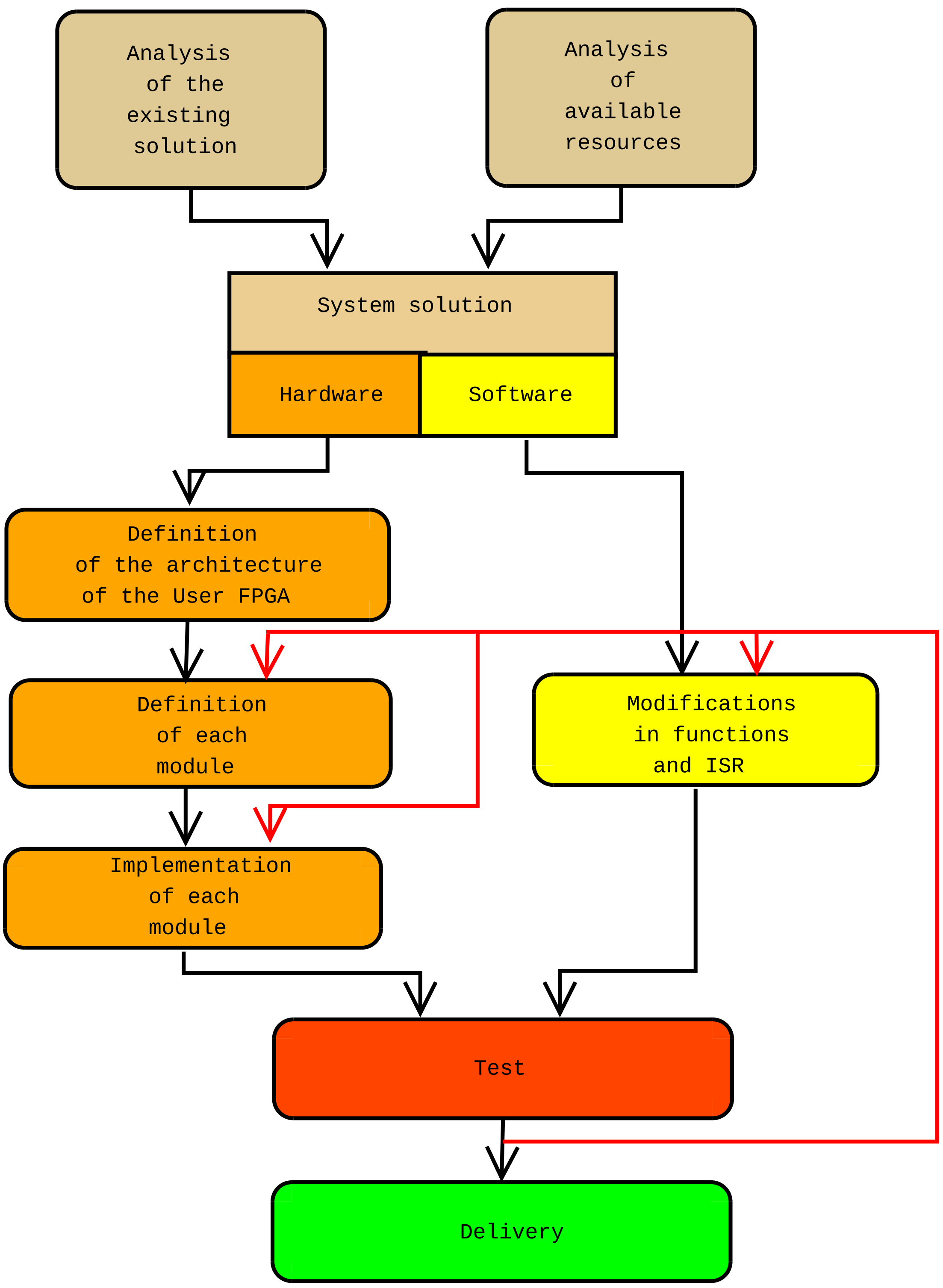
Analysis of the existing solution
Analysis of available resources
System solution
Hardware
Software
Definition of the architecture of the User FPGA
Definition of each module
Modifications in functions and ISR
Implementation of each module
Test
Delivery

Figure 4.1: Design flow block diagram

1. description of the hardware with HDL
2. functional simulation
3. fitting on the actual FPGA
4. timing simulation
5. programming of the FPGA
6. hardware testing of the system (aided by Signal TAP).

Until step five the design flow is quite straightforward; notwithstanding it's worth noting that as the complexity of the system designed increased the ease of simulation decreased strongly. In my case it was mainly due to the presence of multiple communication channels (like buses) between my system on the FPGA and the outer world. Hence I preferred when possible to test the validity of my design directly on the FPGA; in this activity I was greatly supported by the Signal TAP controller that allows the recording and visualization of a number of signals in the design. Moreover testing this FPGA was done in great part by running some ad-hoc code on the DSP and verifying the expected behavior.

In some steps of the actual implementation I also had to make use of the oscilloscope to test some signals directly on the Printed Circuit Board.

After the introduction of the methods I used in this work I am going to speak about what I actually designed.

4.2 Description of the solution

As described in the previous section, one of the main steps in my design was the definition of a system solution, that provided a global picture. In this section I describe the results of this step, and the main ideas that lie underneath it.

The main problem, as already introduced, is to reduce the overhead due to the communication between the DSP and the audio port, by deploying the feature of the audio algorithms of being frame based. Hence some changes in the architecture or in the deployment of system resources are to be made.

The first straightforward idea was the use of the Direct Memory Access bus, that connects the Test Chip to the User FPGA; as the name suggests, this bus allows a direct access to the memory without bothering the processor. Hence the samples from the Audio Port Integrated Circuit could be transferred to the internal memory via the User FPGA.

At this point, it would have been simple if the sample was to be written in the memory always at the same location. This is clearly wrong: the samples are to be stored in a vector one after the other. Moreover for the stereo signal there must be two vectors, and all this doubled for the outgoing samples.

So every time an audio value is sampled, there should be a transfer in the right place and the updating of some variables that describe the buffer. For this purpose some kind of controller is required that could access the internal memory and register of the audio port peripheral.

Moreover when the vector is full, the processor should be interrupted, to make it understand that a new frame of data is ready to be processed.

All these features come in the architecture described in picture 5.2; the main logical modules of this solution are the audio port peripheral and the audio port controller. These modules are both configured by the DSP processor via the Off-Chip System Bus; the peripheral is in charge of make the serial data coming from the audio port appear parallel, like a register. The controller on the other hand controls the transfer.

The whole system is timed by the interrupt sent at a frequency equal to the sampling rate by the audio port peripheral. This interrupt awakes the controller and enables the transfer. Depending on the configuration of the latter, when needed it sends another interrupt to the processor, that will eventually work on the data stored in the SRAM.

An alternative solution to this would be to implement a buffer in hardware, for example on the User FPGA. It would be necessary to store all the data in it and when it is full to send them in the internal SRAM via the

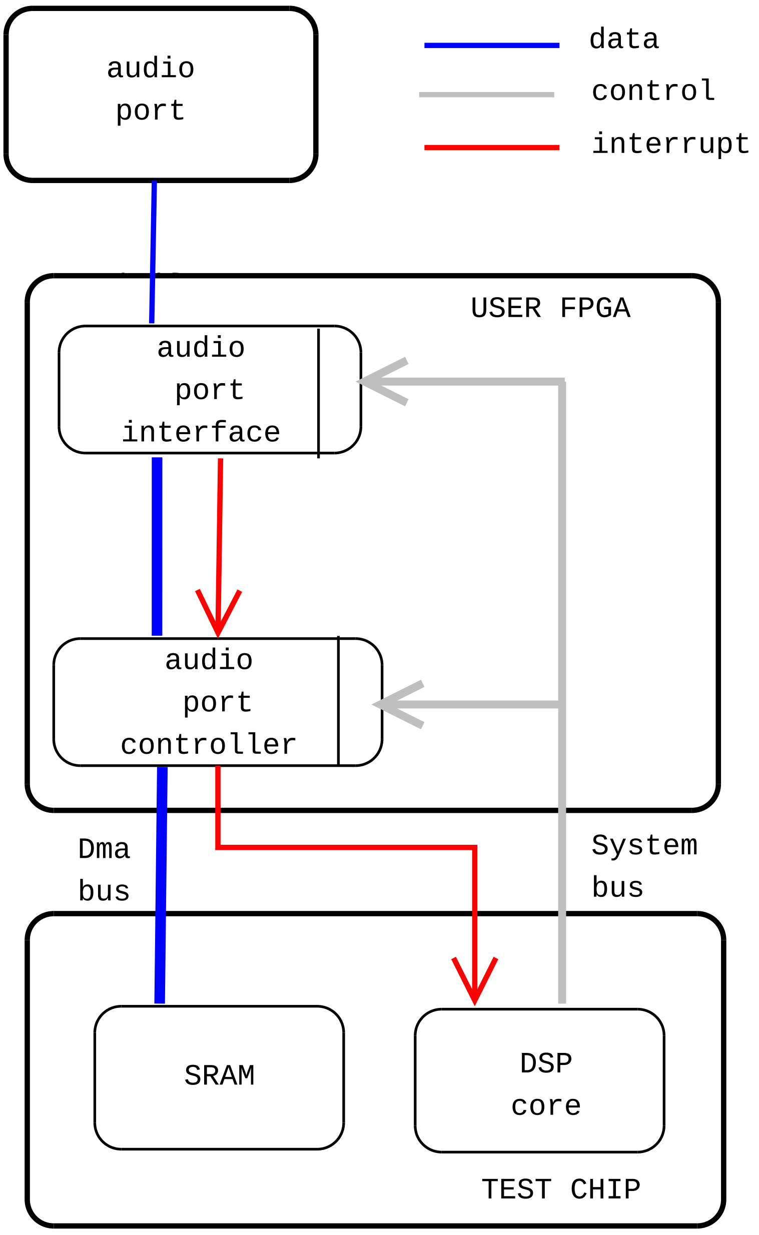

Figure 4.2: System solution overview

Off-Chip DMA Bus. This would not save any overhead, but the complexity would grow due to the presence of the buffer to be controlled. Another disadvantage of this solution is that the variable would not be updated at every value, as it should be.

Part III

Third part

Chapter 5

The User FPGA System Design and Implementation

The User FPGA is one of the two FPGAs on the Development Board, and it is where all the custom hardware I designed is hosted. This device, that is part of the Altera Stratix family, offers a huge amount of resources that are gates, memory and also specialized circuits. It has been added to the board to provide the user (i.e. the customer) with a further degree of freedom; in fact the user could custom the board by adding for example new controllers of peripherals, or a DMA controller. For this reason the board is usually delivered with the user FPGA blank, while all the basic functionalities and controllers are stored in the Control FPGA.

In this chapter I am going to describe the architecture of the digital system in this FPGA, and every single module that is part of it. There is a Control Register that contains configuration information, there are two bus interface, the audio port interface module and the audio port controller module.

To avoid confusion it's necessary to specify the difference between audio port interface and audio port controller. The former is a digital block already existing that translate the serial communication of the audio port to a parallel register-like shape. The latter, that is the main part of my work, is a kind of wrapper of the audio port interface and controls the transfer of data between the registers into the internal memory. To the description of this module is dedicated the whole chapter 7.

So after some introductive details about the architecture and the design flow, all the other modules are described.

5.1 The FPGA system architecture and flow of work

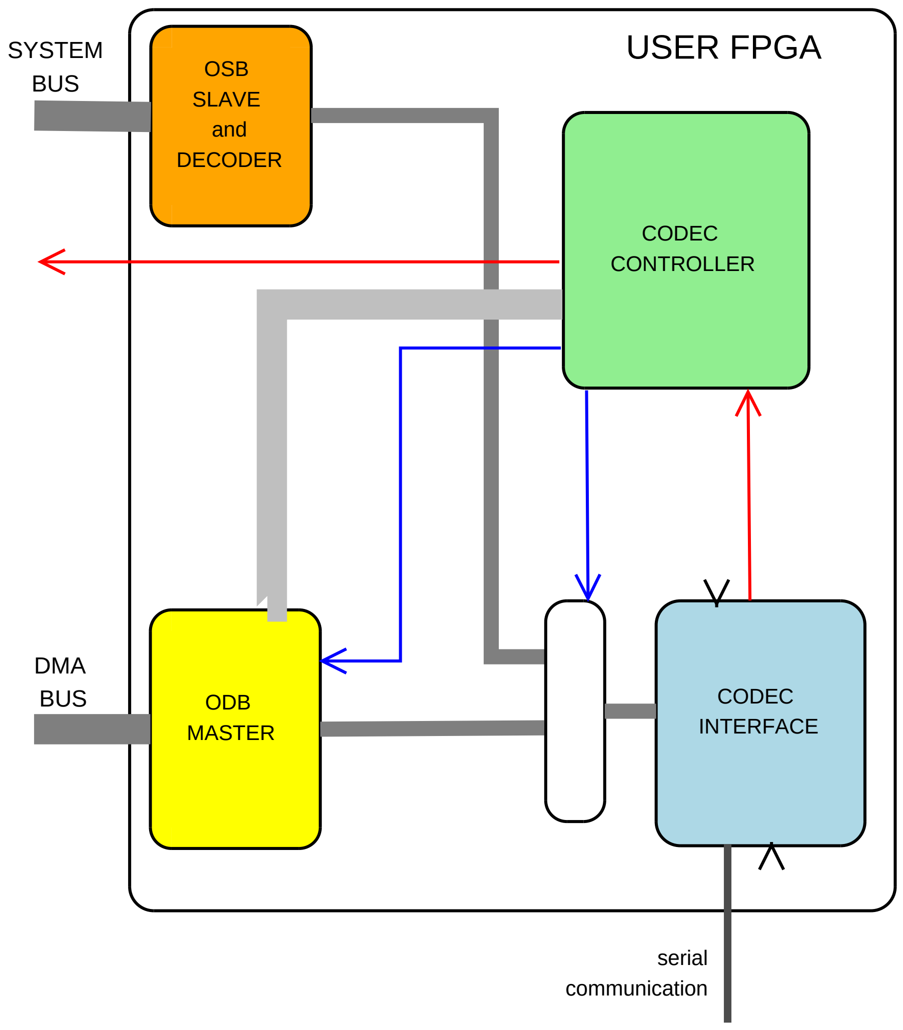

Figure 5.1: Description of the FPGA system

In this description of the architecture of the FPGA system I would start by considering which are the connections of this FPGA with the outer world. As you can see in the picture 6.1, there are two buses and the serial connection with audio port.

The first bus is the Off-Chip System Bus (OSB), the second is the Off-Chip DMA Bus (ODB). The OSB is used to initialize both the audio port interface and the audio port controller, while the other is used just by the audio port controller to transfer data between the test chip memory and the audio port interface and to read and write data in the same memory.

It should be clear that since both of them are buses, they must have at least one master and at least one slave. Due to the configuration on the board the OSB require a slave that could understand the signal coming on the Test Chip while the ODB requires a master. Both this OSB slave and ODB have been implemented and could be seen in the picture 6.1.

Besides these buses there are three wires that are very important between User FPGA and TestChip: three interrupt wires. One of them is used to communicate an interrupt to the Test Chip from the audio port Controller.

On the audio port side there is the serial communication that is described with further details at page B.

From the picture 7.1 you can also see the connection of the audio port interface and controller, with the other modules. As a legend for that picture the wide gray lines are buses, the red lines are interrupt wires, and the blue ones are control signals.

5.1.1 The implementation flow

Besides the design flow I consider in this case interesting to dedicate some lines to the flow I had in the implementation of this hard module.

As at the beginning of my owrk the FPGA was really empty, the first thing that I had to design was the System Bus Slave; this is the only way to start having a look inside the FPGA: without this whichever other modules in the FPGA would have been stand alone, with no communication at all with the TC, and with no possibility for me to understand whether they were working or not.

The second step was the porting of the Codec Interface from the Control FPGA to the User FPGA. This is the first layer of abstraction toward the Core; I made the Codec Interface registers accessible from the System Bus, and this allowed me to run a simple audio application with this new controller. The only difference with the original version was the FPGA in which it worked, in the original case the Control FPGA, and in my case the User FPGA.

Then I designed the Audio Port Controller, widely described in the next chapter.

Finally I designed the DMA bus master. Thanks to this the Test Chip memory could be accessed and data read and written from it. It is controlled by the Codec Controller, via a simple interface. This was the last hardware piece of my system, and it allowed to simulate and tune my whole system.

For all the duration of this implementation the size and definition of the User FPGA Control Register were continuously updated and reviewed, since new features were added during the work.

In the following sections I'll describe the modules that compose the User FPGA, starting from the OSB Slave and Decoder.

5.2 The System Bus Slave and Decoder

The Off-Chip System Bus is a 32-bit AHB bus that extends the On-Chip System Bus. In the architecture of the board, there is a 256-Mbyte memory space reserved to the User FPGA, accessible via the Off-Chip System Bus, in the range between address $0xD0000000$ and address $0xE0000000$. This means that when in the software there is a read or write operation to a location in this range, the System Bus master on the Test Chip will interrogate the User FPGA.

Hence the function of the OSB Slave and Decoder is to make it possible

Name	Width	Address	Access
FPGA Control Register	32	0xD0000000	RW
Controller Control Register	32	0xD0000004	RW
Controller Address Register	32	0xD0000008	RW
Controller Status Register	32	0xD000000C	R
Codec TX Left Channel Register	16	0xD0000010	RW
Codec TX Right Channel Register	16	0xD0000014	RW
Codec TX Command High Register	16	0xD0000018	RW
Codec TX Command Low Register	16	0xD000001C	RW
Codec RX Left Channel Register	16	0xD0000020	R
Codec RX Right Channel Register	16	0xD0000024	R
Codec RX Command High Register	16	0xD0000028	R
Codec RX Command Low Register	16	0xD000002C	R
Controller Program Memory	16 x 256	0xD0001000	RW

Table 5.1: Memory Space in the User FPGA

for the Test Chip to really access that memory area, complying the Standard AHB.

In my system I employed just part of the memory space available, and it is described in the table 6.1. There are mainly three different part of the memory:

32-bit registers used for different functions

audio port peripheral 16-bit registers used to store the values sampled by the audio port

512 byte memory to store the program of the controller

Given this memory space the role of this module is twofold; first it has to interpret the signals from the master and provide it with the correct signals according the standard. Then it has to make sure that the data on the bus are stored in the proper location (write case), or are loaded from the desired address (read case).

There is a couple of things I had to pay attention to in the design of this module; first of all I had to comply with the timing described in the AHB standard. As usual in this case a state machine was developed for the handshaking between master and slave.

A second important issue was dealing with the *endianness* of the system. In every elaboration system the memory stores data in two ways different ways, namely *big endian* and *little endian*; so in implementing a memory you have to take care of the endianness.

So in the case of 16-bit access in the User FPGA, according to the endianness value the meaningful bits would change their positions with respect to the 32 bits of the bus, and the bus slave should take this into account. Further details about this are in the appendix A, dedicated to AHB standard.

5.3 The User FPGA Control Register

The User FPGA Control Register is a 32 bit register at the address 0xD0000000, that is the first available on the FPGA; this register is accessed by the core to control the behavior of the FPGA, in particular of the audio port interface and of the audio port controller.

Table 6.2 contains a summary of all the values in that register. To understand the architecture and the behavior of the FPGA is necessary to focus on many of these bits, one by one.

DmaEnable This bit enables the Controller to work. When it is zero the Audio Port Interface is connected directly to the System Bus Slave, and its interrupt request goes directly to the core. Instead, when this bit is set, the Controller stands amidst the Codec Interface and the DMA Bus Master. The Codec registers in this case are accessible from the OSB in read only mode. This is the typical running mode, where

number	name	function
0	CodecIntAck	Acknowledge an interrupt from the CODEC
1	CodecIntEnable	Enable the Codec to send interrupt
2	CodecIntResetN	Reset the Codec
4 to 7	DipSwitches	Set the Codec clock value in Control mode
8	endianness	specify the endianness of the board
9	codecDC	set the MODE of the codec; 1 Control, 0 Data
10	MasterSlave	set the MODE of the codec I/F; 1 Master, 0 Slave
11	Loopback	1 Loopback Mode, 0 regular Mode
12	DmaEnable	enable the Codec Controller
13	DmaIntEnable	enable the interrupt from the Codec Controller
14		

Table 5.2: Control Register bit table

the Audio Port Controller is awake and generate, when it is necessary, the interrupt request to the core.

Loopback This makes it possible to test the Interface. In this configuration the *Codec Rx* wire of the serial communication, is shorted to the *Codec Tx*; for the Interface to work properly you expect the Codec RX registers to reply exactly what you write in the Codec TX registers[1].

endianness This bit indicates the endianness of the board. Please consider that endianness doesn't affect the 32-bit operations; thanks to this it is possible for the Core to communicate this information to the FPGA by making a 32-bit write operation in the Control Register. This configuration must be the first done by the software at power on ; then this bit is used by the two bus controllers to work in the correct way.

codecDC This bit is one of the Audio Port Serial Interface. It is changed by the ISR during its control section (see page 73), to change the mode of the codec from *Config* to *data*.

[1]For the function of these registers see the Audio Port Interface section

CodecIntEnable Enable the Interface to send interrupt request; depending on the *DmaEnable* bit, the IRQ could reach either the DSP or the Coded Controller;

DmaIntEnable Enable the Codec Controller to send an interrupt request to the DSP.

5.4 AHB Off-Chip DMA Bus Master

Another important module in the User FPGA system is the Off-Chip DMA Bus Master. Its role is to provide the right signal to the bus wires always complying with the AHB-standard. Moreover it has to provide the Controller with a simple interface that lets the Controller easily use the DMA bus.

The signals of the interface I designed are described in the table 6.3; besides the data and address wires it is worth speaking about the control signals. The audio port controller controls *write enable*, *read enable* and *size*; when it wants to perform an operation it raises one among *re* and *we* and put the right value on *size*. The Master, if ready to accept, will drop the *ok* signal, meaning that it became busy; the controller knows that the operation is finished when *ok* is raised again, and if it is a read operation it will read the proper value on the *RDATA* signal.

Due to possible problems in the communication I also implemented a time-out mechanism that forces each operation to end after 20 cycles, if not already finished.

5.5 The Audio Port Interface

The last block to describe before the audio port controller is the audio port interface, the digital block that stands between the audio port IC and the DSP, to make the two work together. As previously said it is a pre-existing module, and this section I am just going to describe it. In the picture 6.2 there is an overview of it. You can see the five signals that compose the

name	direction	function
WDATA[31:0]	IN	write data
RDATA[31:0]	OUT	read data
ADDRESS[31:0]	IN	on-chip SRAM address to access
WE	IN	write enable
RE	IN	read enable
SIZE[1:0]	IN	data size
OK	OUT	answer from the master

Table 5.3: DMA bus signals

serial communication to the integrated circuit; they are *CodecClock*, *Codec-Sync*, *CodecTx*, *CodecRx* and*ControlData* and they are connected to the corresponding pins of the integrated circuit.

The data wires are connected to a 64-bit shift register each; this is then connected to a group of four 16-bit registers. This simple systems makes the translation from serial to parallel communication. Both the dma bus and the system bus can access those registers, with the mode described in table 6.5.

The *CodecClock* and *CodecSync* are bidirectional pins, that could be driven by the interface. This happens during the Audio Port configuration phase[2]: the *State Machine 1* generates a a clock and a sync signal for the Audio Port, from the *UserFPGAClock*. The *ControlData* pin, set to zero, let the signals generated to drive the actual wires of the interface through the tristate buffer.

As soon as the *data* mode is entered (and the *ControlData* is set), the clock and the sync will be provided by the Audio Port, at the frequency configured. In both mode the *State Machine 2* is fed with sync and clock, in order to generate an interrupt to the core, or to the controller.

The interrupt is clearly at the same frequency of the sync signal, that is basically the sampling frequency. The interrupt means that there are data ready for the application in the RX registers, and that the TX registers are

[2]see 8.3 for explanation

empty and they could be filled with new data.

Register name	function	access mode	system bus address
LchTxReg	Left Channel Transmit Register	write-only	0xD0000010
RchTxReg	Right Channel Transmit Register	write-only	0xD0000014
CmhTxReg	Command High Transmit Register	write-only	0xD0000018
CmlTxReg	Command Low Transmit Register	write-only	0xD000001C
LchRxReg	Left Channel Receive Register	read-only	0xD0000020
RchRxReg	Right Channel Receive Register	read-only	0xD0000024
CmhRxReg	Command High Receive Register	read-only	0xD0000028
CmlRxReg	Command High Receive Register	read-only	0xD000002C

Codec register table

5.6 The issue of synchronous design

The whole system is basically composed by the Test Chip, the User FPGA and the audio port IC. During the ordinary activity the audio port is configured to be the master of the serial communication, so this adds an asynchronous component to the system; since I am going to use the Codec I/F as it is, the problem should have been already faced and I am not going to deal with it. I would rather focus on the communication between the Test Chip and the FPGA; it is done by two buses and the clock is provided in both cases by the Clock Generator in Test Chip, that is software programmable. The System Bus clock is the one used in the DMA system. The default configuration is 45 MHz for both, but no software control is given on their phases; it could have been an important issue if there was a phase delay.

Luckily the results of measurement gave a positive answer, meaning that the two signals could be considered really synchronous, letting me consider all this part a complete single clock part.

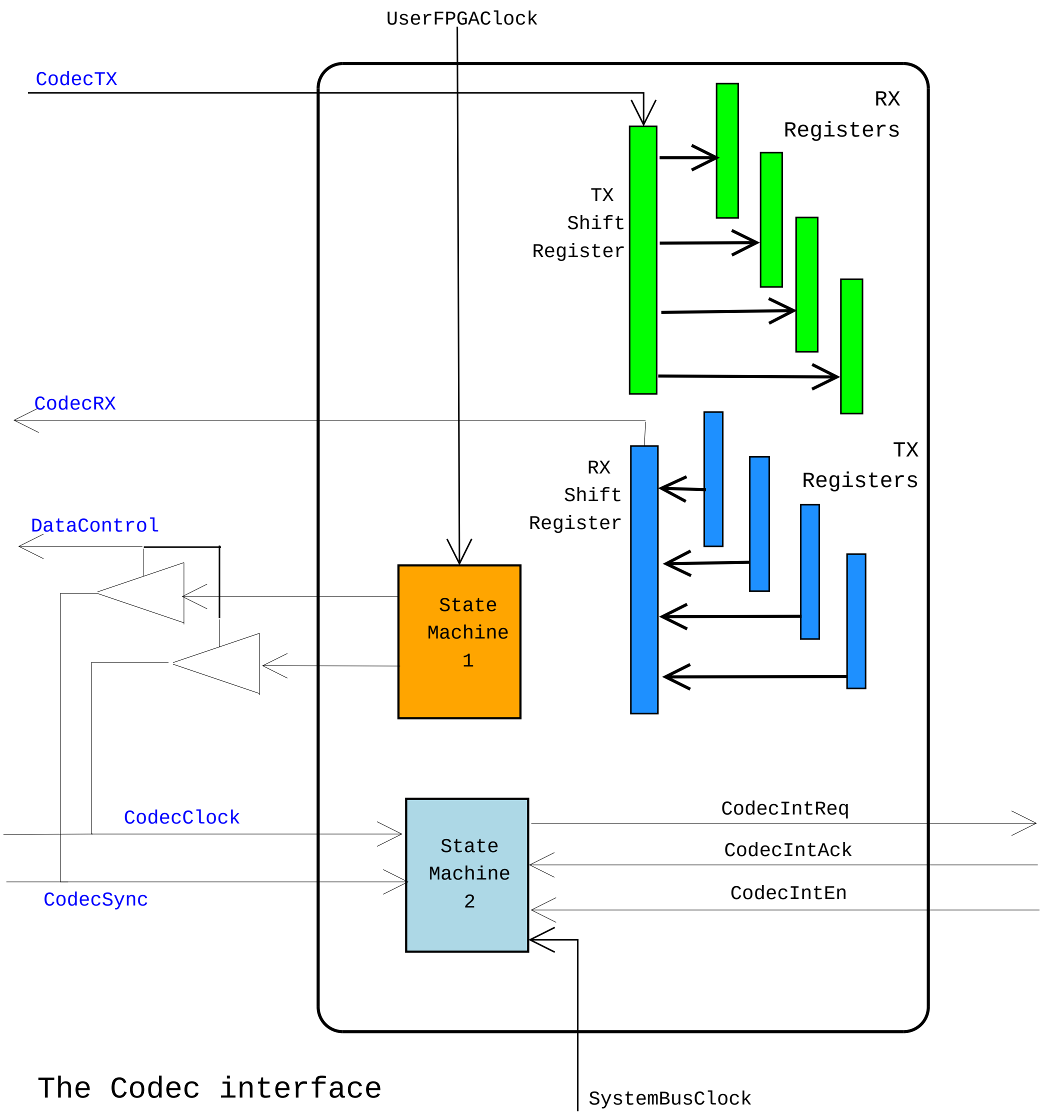

Figure 5.2: Audio Port Interface

Chapter 6

The Audio Port Controller

This chapter deals with the audio port[1] controller, its design and its architecture. To implement this module among the possible choices I decided to design a kind of micro-controller; the reason for this choice and some details on its features are in the first section. In the remainder of this chapter there is the description of all the modules that compose the micro-controller.

6.1 The Audio Port Controller Design Flow

In this first section I am going to explain the choice in the implementation of this module. It all starts with the tasks it has to accomplish; it is described in the section 8.4, but at this stage it's enough to say that it is quite complex and not a priori easily and exactly definable.

Since that complexity looked to be very high I decided to use a programmable module, a kind of CPU with its program memory. It is true that during the ordinary life of the final product there is no need of programmability, but I thought that this would have been a great advantage in the development of this module. This causes a little increase in the amount of resources taken for the design, but since the User FPGA was really huge, I estimated that this wouldn't have been an issue. As you can see in the

[1]in this document instead of the name "audio port" you could find the name "codec"

section 7.7, this proved to be right.

For what concerns the design I analyzed the task to be performed and decided a set of instructions to be implemented. The definition of the instruction set is important because it represents a kind of abstraction over the processing unit. There are many important features that come with the choice of the proper IS: the number of instructions, the width of each instruction, the width of the address space, which instruction to implement. In order to make the right choice I had to consider requirements from both the top level and the bottom level, for example readability of code, functionality desired, complexity of the underlining hardware, speed.

The outcome of this activity is described in details in the section 7.4.1.

Once I finished the implementation of this micro-controller, I tested with simulation as far as I could and then programmed the FPGA. At this stage I started also the controller software development, described in the Software chapter, section 8.4.

A great part of the design was also dedicated to debug tasks; in the design I had to foresee some tricks that would make the development of the application easier.

6.2 The Architecture of the Controller

The internal architecture of the audio port controller is described in figure 7.1. As in any processing unit it can be divided into two parts: the *control part* and the *operating part*. The operating part is mainly composed by a register bank and an arithmetic logic unit (ALU), and its task is to perform operations on the data stored in the registers.

The program flow and the flow of each instruction is regulated by the control part that sends the correct signals to the operating part (the red signals in picture 7.1).

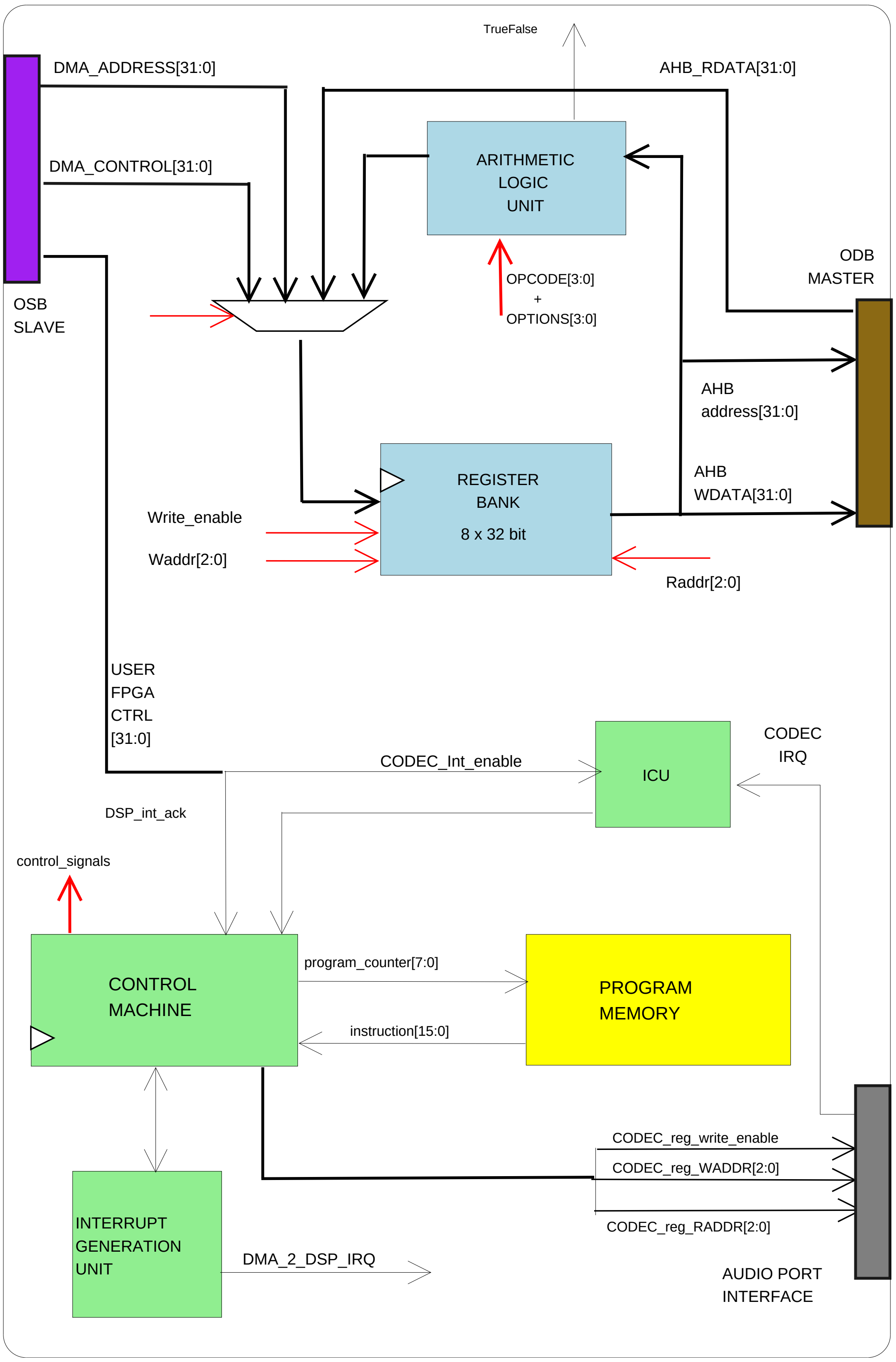

Figure 6.1: Internal architecture of the audio port controller

6.3 Input-output model

To provide some more details on the controller I would like to introduce all the inputs and all the outputs. Some of them are already in the picture 7.1, but the their complete list is in table 7.1.

The controller basically interfaces with the OSB bus, the ODB bus, the audio port interface, the core and the FPGA control register. It is an almost completely synchronous system regulated by a single clock and a single reset signal; the only exception is the Interrupt Control Unit.

Via the OSB, the Test Chip can communicate to the controller some information like a pointer to a SRAM location (in *dma_address* register), or configuration bits (in the *dma_control* register); on the other side the controller provides the TC with some details of his internal behavior. This is particularly useful in the *debug mode*.

To perform transfer instructions from and to the internal SRAM, the controller has to deal with the ODB master; with *re*, *we* and *size* it commands the master what to do. At the same time address is driven, and interface registers are enabled if the operation requires it.

There are also four bits from the *FPGA control register*: two of them (*endianness* and *dma_enable*) configure the behavior of the Controller, while the other two are used for interrupts.

6.4 The control part

As in any other processing unit, the control part is in charge of dealing with the flow of each instruction through all the stages, and of supervising the program flow. In this section I want to describe the whole control part, that is the instruction set, the flow of each instruction and the modules that build it up.

name	direction	width	description
system signals			
clk	IN	1	module clock
reset_n	IN	1	negative reset
OSB slave interface			
dma_control	IN	32	provide ctrl from TC to DMA
dma_address	IN	32	provide a starting address to DMA
dma_status	OUT	32	provide the TC with the status of the DMA
OSB interface to program memory			
rom_address	IN	8	address to program memory
rom_we	IN	1	write enable from OSB decoder
rom_data	IN	16	instruction to program memory
signals from FPGA control register			
dma _ enable	IN	1	enable DMA
endianness	IN	1	select endianness
codec_int_ en	IN	1	enable codec interrupt
codec_int_ ack	IN	1	acknowledge codec interrupt
TC signals			
dma_irq	OUT	1	interrupt request to TC
ODB master interface			
re	OUT	1	read request
we	OUT	1	write request
size	OUT	2	specify size of request
ok	IN	1	feedback signal from ODB master
odb_address	OUT	32	address for the ODB master
wdata	OUT	32	data to write
rdata	OUT	32	data to read
audio port interface			
codec_reg_sel	OUT	3	select one among codec registers
codec_reg_we	OUT	1	codec register write enable
Codec2DmaIrq	IN	1	interrupt request from codec
Dma2CodecIntAck	OUT	1	acknowledge codec irq

Table 6.1: Input-output description of the DMA module.

6.4.1 The Instruction Set

The instruction set is composed by 15 instruction, encoded in 16 bits, and divided into four 4-bit fields. In tables 7.2 and 7.3 you can see a detailed description of the whole instructions set; in the table are assumed the following meanings:

- *value* and *mask* are items in two sets of 16 elements;
- *Source*, *Dest*, *Source address*, *Dest address* are registers in the bank; they can be pointers to on-chip SRAM locations;
- *address* is a program ROM address ;
- *condition* is one among *IFTRUE*, *IFFALSE*, *ANYCASE.*

Eight instructions among these involve the use of the ALU; *ADD*, *SUB*, *OR*, and *AND* are the usual mathematical or logical operations. The first operand is in the register indicated by *REG1* field, while the second operand is carried encoded in the *options* field. It means that for anyone of these instructions the second operand could be chosen in a set of 16 items.

The *MOVE*,*MOVE_CTRL* and *MOVE_ADDRESS* involve the ALU but none operation is performed on data. The same could be said for the *TESTEQZERO* ; but in this case bit *TrueFalse* is set or reset if the operand is equal to zero. This bit is used by the address generation unit.

In all these operations the *REG2* field indicates the destination register.

Transfers of data between SRAM and internal registers are performed thanks to *READ* and *WRITE*; in both cases the *REG2* field contains a pointer to the SRAM location that should be read or written. The same thing happens with *WRITE2CODEC* and *READFROMCODEC* instructions, which respectively write and read from codec interface registers. One important difference that is worth noting is that the SRAM to DMA transfer is 32 bit wide every shot, while the SRAM to CODEC transfer is 16 bit wide.

The *JUMP* instructions with the options *IFTRUE*, *IFFALSE* and *ANYCASE*, are powerful means to control the program flow. The *REG1* and

instruction	OPCODE	OPTIONS	REG1	REG2
ALU instructions				
ADD	0000	value	Dest	Dest
SUB	0001	value	Source	Dest
AND	0010	mask	Source	Dest
OR	0011	mask	Source	Dest
TESTEQZERO	0100	none	Source	Dest
MOVE	0101	none	Source	Dest
MOVE_ADDR	0110	none	none	Dest
MOVE_CTRL	0111	none	none	Dest
transfer instructions				
READ	1000	none	Dest	Source address
WRITE	1001	none	Source	Dest address
WRITE2CODEC	1010	none	Dest	Source address
READFFROMCODEC	1011	none	Source	Dest address
control instructions				
JUMP	1100	condition	address[7:4]	address[3:0]
ClearCodecIrq	1101	none	none	none
SendIrq2Dsp	1110	none	none	none

Table 6.2: Instruction set description table, part I

instruction	descriptions
ALU instructions	
ADD	Dest = Source + value
SUB	Dest = Source + value
AND	Dest = Source AND mask
OR	Dest = Source OR mask
TESTEQZERO	set flag if Source == 0
MOVE	Dest = Source
MOVE_ADDR	Dest = Address
MOVE_CTRL	Dest = Control
transfer instructions	
READ	Dest = * Source address
WRITE	* Dest address = Source
WRITE2CODEC	Dest = * Source address
READFFROMCODEC	* Dest address = Source
control instructions	
JUMP	Jump to address if condition is true
ClearCodecIrq	clear the CODEC Int Request
SendIrq2Dsp	send an interrupt to DSP

Table 6.3: Instruction set description table, part II

REG2 field together provide the program address (8 bit long) where to jump if the condition is verified.

To send interrupt request to the core there is the *SendIrq2Dsp* interrupt; the most important feature of this instruction is that it will not terminate until it has been acknowledged by the core. And finally the *ClearCodecIrq* instruction clears the interrupt request sent by the codec to the DMA. These last instructions have their own modules called respectively *Interrupt Generator Unit* and *Interrupt Control Unit*.

6.4.2 The phases of each instruction

A simple state machine - *ctrl_sm*- is responsible for the flow of each instruction; here are the three steps: *fetch*,*execution* and *address generation*.

fetch phase given the correct address, the next instruction is fetched by accessing the *program memory*; this takes one cycle.

execution phase during this phase the actual operation is performed; please notice that the duration of this phase is not a priori determined, cause in transfer operation cases the Test Chip SRAM is involved. If it is required, the *TrueFalse* bit is evaluated and appropriate signals for the AGU are generated.

address generation phase according to the signals provided during the execution phase, the address of next instruction is produced. During this phase the Control Part interfaces the Interrupt Control Unit to check whether any interrupt has occurred .

The control part also provides the internal bus controller, the ALU, and the ODB bus controller with the appropriate commands.

6.4.3 The execution phase

The execution phase begins when the *instruction register* is loaded with the new instruction and an *enable* command is given; at this moment one among the following state machines runs :

- *alu_instr_sm* controls the instructions that involve the ALU
- *spec_instr_sm* controls *SendIrq2Dsp* and *ClearCodecIrq*
- *trans_instr_sm* controls the transfer instructions
- *jump_instr_sm* controls the *JUMP* instruction;

according to which acknowledges its own opcode pattern. This machine sets a *busy flag*, that will be cleared only when it will finish its operation. The flow doesn't proceed until all the *busy flags* are zero; this kind of semaphore is needed cause the execution time of the transfer instructions is unknown.

During this phase control signals to the datapath are generated; this is the only moment in the execution flow that it is possible. Each state machine generates a different configuration of these signals, and every time the correct configuration is selected with a mux, controlled by the OPCODE value. In other words, while the *alu_instr_sm* is running the mux selects its configuration to be forwarded to the ALU and to the register bank.

At the same time a binary information is generated for the Address Generator Unit, such as whether or not a jump is to be performed.

6.4.4 The Address Generation Unit

The AGU is responsible for the generation of the next instruction address; it either increments the present address, or jump to an absolute address. In table 7.4 you find a list of inputs and outputs. In the case that the Controller receives an interrupt request, the AGU jumps to the address provided by the ICU ; on the other hand a jump could occur if the condition in the JUMP instruction has been evaluated true.

The AGU interfaces directly with the ICU via two wires: it receives the *int_req* signal and output the *int_ack*; the acknowledgment is set when the interrupt is served, that is when the instruction execution reaches the address

name	size	direction	description
clk	IN	1	
Nreset	IN	1	
new_address	IN	8	address from the JUMP instruction
int_address	IN	8	address from the ICU
int_req	IN	1	from ICU; request of interrupt
int_ack	OUT	1	to ICU; interrupt acknowledged
LoadInc	IN	1	from *ctrl_sm*; increment or load new address
enable	IN	1	from *ctrl_sm*;
address_out	OUT	8	new program counter

Table 6.4: AGU input output description table

generation phase. At this moment the address provided by the ICU via the *int_address* is loaded as new address.

If the *int_req* is not asserted, the AGU checks whether the *JUMP* instruction condition has been evaluated true; in that case *new_address* will be loaded as the next one. If any of these possibility has occurred, the present address is increment by one and put on the output.

6.4.5 The Interrupt Control Unit

The ICU is the module in charge of receiving the interrupt and synchronizing it with the control part; when an interrupt occurs it signals it to the AGU and at the same time it provides it with the ROM address where the routine is. When the *ClearCodecIrq* instruction is run, the ICU deals with the codec interrupt acknowledge signal. This is a simple behavior that could be easily used for multiple interrupts. In this case there was no need for interrupt masking mechanism, so it was not implemented.

6.4.6 The Interrupt Generator Unit

The IGU is responsible for sending the interrupt request to the core via the dedicated FPGA pin. It is triggered via software by the instruction *SendIrq2Dsp* and reset by the core, by writing in User FPGA Control Register.

6.5 The program memory

The program memory is where all the instructions are stored; it is organized in 16 bit wide words and the maximum length of the program is 256 instructions. These size has been chosen as a trade off between the address width, and the program length. An eight bit width matches with the sum of two fields in an instruction; this makes it ideal to fit in *REG1* and *REG2* fields of the *JUMP* instruction, as described above. On the other side the final program length is under 100 instructions, making it reasonable the size chosen.

Deploying a positive feature of Stratix technology I designed it as a dual port memory RAM. The first port is a read only access used by the Controller, and it behaves exactly as a ROM. The other port is used in both write and read directions by the Core, that accesses it via the system bus. Thanks to this, it is possible to load the program at run-time, just before making it run. This feature was very useful in developing the correct program and in the discovery of bugs in design. The writing in this memory is protected by a *dma_memory_write_enable*, of the *Dma_Control_Register*; writing is allowed only when this bit is set.

As you can see later in this chapter the memory space is organized in two main part. The first part is at the beginning (address *h0*) and it will be executed at the start and whenever the Controller is idle. The second part starts at the address *hC* and it will be executed every time an interrupt from the CODEC arrives. At the end of this program the execution is back to the first part.

6.6 The DEBUG mode

To reduce the developing time I added a Debug mode to the Controller; in this mode the flow of the program could be controlled by an external programmer. I added two bits to the *Dma_Control_Register*; here they are with their function:

debug enable : this bit force the Controller is the debug mode when set. It means that the *ctrl_sm* is stuck just before beginning the execution phase.

debug this bit is used when the Controller is the debug mode. It allows the *ctrl_sm* to go ahead, but just for one instruction. This signal is edge triggered, to avoid problems due to asynchrony between core and Controller.

This allowed me to write simple *C* functions like *dma_debug_single_step()* that proved to be very useful in the debugging activity. It it important to say in this moment that in the whole work, the greatest important has been assigned to test and debug tools.

6.7 The Controller Mapped in the FPGA

In table 7.5 you find a list of the resources used by the Controller in the FPGA. Among those there are Logic Elements that could both registered and combinatorial, and memory bits, in the dedicated memory blocks. The device that is referred to is an ALTERA STRATIX EP1S30. A percentage value is also reported to give an idea of how many gates are required.

module	logic elements	memory bits
ICU	1	0
IGU	2	0
ctrl_sm	19	0
jump_instr_sm	6	0
spec_instr_sm	12	0
alu_instr_sm	12	0
trans_instr_sm	15	0
AGU	18	0
control part	121	0
ALU	210	0
status_register	30	0
mux	64	0
register bank	0	256
program memory	0	4096
overall	428	4352
amount available	32470	3317184
percentage occupied	1.31%	1.29 %

Table 6.5: FPGA resources used by the DMA system

Chapter 7

The Software

7.1 Introduction

This chapter deals with the software side of the system, that is the code that runs on the DSP, plus the code for the Controller. It is related to the hardware in a twofold way; on one side the pre-existing driver contributed to the specifications of the new controller that I designed. This is due to the transparency required to the new controller. On the other side the hidden part of the software and few other functions, had to be reshaped to fit the new hardware. Moreover some of this software has been changed, some has been written from scratch, some never touched.

In the software you can recognize three different parts; starting from the lowest level they are :

the interrupt service routine it is called by the controller, and it has been changed; in particular it has been decreased in size and functionalities;

the audio port driver that is all the data structures and functions used by the application. This was slightly changed

the main program this is the top level; the data are transformed with library algorithms and the FPGA is configured properly.

Trying to follow the way I worked on this issue, I'll start describing the *CODEC driver*; then I'll deal the *ISR* and finally I'll describe the program running on the audio port controller.

7.2 The Audio Port Driver

7.2.1 Data structures

The variables are allocated in the SRAM, that is the internal memory of the DSP system. This memory can be accessed by both the core and the DMA interface; the sharing of the these variable is one of the points, without which this work wouldn't have been possible. The data variables are eight buffers, half for transmission and half for reception; those for transmission are filled by the application and emptied by the ISR, and viceversa for the reception ones. In each of these two groups, two buffers are for the left channel and the other two are for the right channel.

The buffers allocated seem to be twice as many as required; to focus on this point consider that two parallel "threads" will work on the same data. To avoid that they interfere with each other, double data structures are needed; in particular when the application reads *RxBuffer1* and writes the transformed data in *TxBuffer1*, the *ISR* reads the samples from *TxBuffer2* and stores the incoming samples in *RxBuffer2*.

Let's introduce now the buffer descriptors called *db_codec_rx_bd* and *db_codec_tx_bd*; these and all the other variables are described in the picture 8.1. Due to the complexity I only showed the half referring to reception. Anyway those buffer descriptors are composed by the following four fields:

lSamplePtr a 32-bit unsigned int, that points to the first sample of the left channel;

rSamplePtr a 32-bit unsigned int, that points to the first sample of the right channel;

status a 32-bit int that contains information about the status of the buffer; it is the only field that is modified at run time;

length 16-bit int, that provides the size of the buffer.

Each buffer (left and right channel buffers considered as a whole) has its own descriptors. Every moment just one buffer could be active for the application and this is written in its buffer descriptors. The *status* field is modified and checked by the application, the *ISR* and the controller program. The following table shows the meaning of its parts :

name	bit	meaning
act	4	set if the buffer is active
rdy	3	buffer ready
lst	2	set if the buffer is the last
lEn	1	left channel enabled
rEn	0	right channel enabled
count	31:16	number of valid samples

Let's focus on the *rdy* bit. It behaves differently in the receiver and transmitter case. In the receiver buffer descriptors, it is set by the controller when this has filled the whole buffer; it is then cleared by the application when uses those values. In the transmitter case, it is the application that, once filled the buffer, sets the *rdy* bit while the controller should clear it as soon as it empties the buffer.

In addition the two receiver buffer descriptors may not have any variable interleaved between them; this is to allow an easy access to the *ISR*. The same may happen for transmitter ones.

Besides the buffer descriptors you find a configuration variable called *db_codec_config*; the following fields are there kept:

mode states the audio port operation mode, as far as data rate, sample size, behavior;

firstTxBdPtr points to the first transmitter buffer descriptor;

firstRxBdPtr points to the first receiver buffer descriptor;

callback points to the code where the *callback()* function is stored.

This variable is initialized at the beginning of the program, and it is never modified. The *callback()*function is very important; it is called at the end of *ISR* and it just switches a global variable. This allows the *main* function to enter in data operation segment. It basically acts as a one to one semaphore.

7.2.2 Interrupt Service Routine Data Variables

So far we have described the variables seen by the application; there is another important variable that is used and modified at run time by both the *ISR* and the controller program: it is the *DbCodecCurData*. It is hidden to the application, but it points to all the buffers and all the buffer descriptors, allowing the controller and the *ISR* to work on them.

Here are the details:

name	function
lTxPtr	pointer to the next left sample to be sent
rTxPtr	pointer to the next left sample to be sent
curTxBdPtr	pointer to the current tx buffer descriptor
firstTxBdPtr	pointer to the first tx buffer descriptor
txCallback	pointer to the tx callback function
lRxBdPtr	pointer to the location where to store the next left sample
rRxBdPtr	pointer to the location where to store the next right sample
curRxBdPtr	pointer to the current rx buffer descriptor
firstRxBdPtr	pointer to the first rx buffer descriptor
rxCallback	pointer to the rx callback function

Through a pointer to this variable, every data of the driver could be accessed. This is the way actually done by the controller. An overview of what I just described is in the picture 8.1.

7.2.3 Audio Port Driver Functions

The main functions provided for the application are the following:

DbCodecInit() This routine initializes the Audio Port driver: in particular it does the following operations:

- the Audio Port Interface and IC are reset
- the transmit and receive buffer descriptor are initialized
- the driver's internal pointers are configured to point the first transmit and receive buffer descriptors
- the Audio Port hardware is programmed according to the specified configuration mode
- the Audio Port FPGA registers are programmed
- the Audio Port ICU interface is programmed but not enabled
- the Audio Port hardware is taken out of reset

DbCodecEnable() This function enables the Audio Port interrupt with a specified priority.

These functions were changed to fit the new hardware; in particular the address of the Audio Port Interface registers changed, and the interrupt in the new controller was a different one. This modifications required to set in a different way some parameters in the internal definition of these functions, but the goals and structures kept exactly the same.

7.3 The Interrupt Service Routine

The interrupt service routine is the part that I most modified; many of the task originally in charge of it, are now performed by the controller. In this section I want to make a picture on how was the *ISR* before my work, and then make clear which changes were done.

The *ISR* is basically divided into three parts.

the Audio Port configuration this is responsible for the correct initialization of the audio port IC. As described in the CODEC chapter, to switch from *control* to *data* mode, it requires a precise timing. It is done at the very beginning and it takes about 200 interrupts.

sample transfer This transfers samples from audio port interface to internal memory. Some control has to be done, and the buffer descriptors are updated each time. This part of the interrupt runs for every sample, that at the sampling frequency.

control this part is executed just once per frame, and its goal is to update the *DbCodecCurData* variable with the current buffer descriptors.

Even if the configuration takes a long time, it is executed just once, and it is not taken into account when considering the performances of the system. As you can see the most cycle expensive part is the second one; it has the longest code and it is executed far more times than the third. Therefore this part has been taken away from the *ISR* and the same tasks have been assigned to the new controller.

For example let's consider an audio algorithm that works on 160 sample frames; let's also assume that the sampling frequency is 48 KHz. In this case the *transfer* part will run at 48 KHz while the *control* part at only 300 Hz.

In the following I am going to describe the details of this two part.

7.3.1 The Audio Port Configuration

As described in B, the Audio Port IC requires a precise procedure in order to enter the *data* mode. At the end of this procedure the Audio Port will be configured with the current wanted values of sampling frequency, data mode, internal behavior. To understand the following three phases keep in mind that the CODEC in the *control* mode will reply each input on the output. These are the three phases :

first interlock the control word is written with the DCB bit low; when a control word with the DCB bit low is replied, it jumps to the next phase;

second interlock the same as the first interlock, but the DCB bit is now high;

autocalibration reads and writes the control word for 195 times.

7.3.2 The Transfer Section of the ISR

This part is split into two equivalent sections, the first for the transmission, the second for the reception. Here are the steps executed by each part:

1. the *DbCodecCurData* is accessed and the current buffer descriptor is fetched;
2. the status field of the current buffer descriptor is read, in particular the *rEn* and *lEn* bits
3. if enabled the transfer operations between the current buffers and the audio port registers are performed
4. if enabled the *count* field of the current buffer descriptor is decreased by one,
5. if the *count* field is equal to zero, the *control* part is called, otherwise the *ISR* ends.

7.3.3 The Control Section of the ISR

In the control section the *DbCodecCurData* is updated; in particular the *CurTxBdPtr* and *CurTxBdPtr* are changed. In addition each buffer descriptors status is modified; the *count* field is initialized to the size value, and the *ready* bit is toggled to communicate to the application that the buffer is either full or empty.

Finally the *callback* function is called, just before ending the *ISR*. This will give the control of the flow back to the application.

7.3.4 Changes made in the ISR

The final version of the ISR for the new controller was basically the original one without the second part, which is in charge of the data transfer. In fact in the system I developed this function is performed by the audio port controller.

This change makes the Interrupt Service Routine much shorter in terms of cycles, and it is the main contribution to the overall overhead reduction. Quantitative details are in the 9 chapter.

7.4 The Program Running on the Controller

After I introduced all the variables and all the software, the picture is complete and it is possible to show the program that runs on the audio port controller.

To program the controller in the FPGA I found very useful to develop a simple program that could translate a text file with the assembly language into the controller machine language. In this way I make it easier to develop the program that should run on the controller. It was also very useful in the debugging of the controller, where the assembler proved to be a user-friendly and easy tool.

It basically reads each line of the text file and translate it in a 16 bit word, that is an instruction. Every field is written and encoded separately. After the encoding it writes this word in a vector, in the location pointed by the row number. The program was made also to allow blank lines and comments at the end of each instruction.

Let me remind that the memory in the User FPGA is part of the memory space of the DSP; this makes it easily accessible via DSP software. Thanks to this the vector with the instructions is allocated directly in memory in the FPGA where the controller goes to read it. A protection bit has been planned; this protects this area of the memory from unwanted write operations.

So here is the program that actually runs on the DMA; this is written in the assembly language described below.

```
0 MOVE_ADDR R0      // dbCodecCurDatain R0
1 JUMP ANYCASE 0 1

12 ClearCodecIrq

13 MOVE R0 R1
14 ADD EIGHT R1 R2
15 READ R3 R2        // R3 <- curTxBdPtr
16 READ R4 R3        // R4 <- txBdStatus
17 AND  REnBdMask R4 R5
18 TESTEQZERO R5 R5
19 JUMP IFTRUE 1 6
20 SUB 0X100 R4 R4
21 WRITE R4 R3     // update txBdStatus

22 AND LEnBdMask R4 R5
23 TESTEQZERO R5 R5
24 JUMP IFTRUE 1 D
25 READ R2 R1        // R1 <- lTxPtr
26 WRITE2CODEC CodecTxLeft R2
27 ADD TWO R2 R2
28 WRITE R2 R1      // update lTxPtr

29 AND REnBdMask R4 R5
30 TESTEQZERO R5 R5
31 JUMP IFTRUE 2 5
```

```
32 ADD FOUR R1 R1
33 READ R2 R1        // R1 <- rTxPtr
34 WRITE2CODEC CodecTxRight R2
35 ADD TWO R2 R2
36 WRITE R2 R1       // update rTxPtr

37 AND StatusCountMask R4 R5
38 TESTEQZERO R5 R5
39 JUMP IFFALSE  2 A   // 42

40 AND TxReadyMask R4 R5
41 WRITE R5 R3      // clear TX ready bit

42 ADD SIXTEEN R1 R1 // *R1 = lRxPtr
43 ADD EIGHT R1 R2   // *R2 = CurRxBdptr
44 READ R3 R2        // R3 <- curRxBdPtr
45 READ R4 R3        // R4 <- rxBdStatus
46 AND RLEnBdMask R4 R5
47 TESTEQZERO R5 R5
48 JUMP IFTRUE 3 3
49 SUB OX100 R4 R4
50 WRITE R4 R3     // update rxBdStatus

51 AND LEnBdMask R4 R5
52 TESTEQZERO R5 R5
53 JUMP IFTRUE 3 A
54 READ R2 R1       // R2 <- lRxPtr
55 READFROMCODEC  CodecRxLeft R2
56 ADD TWO R2 R2
57 WRITE R2 R1      // update lRxPtr
```

```
58 AND REnBdMask R4 R5
59 TESTEQZERO R5 R5
60 JUMP IFTRUE 4 2
61 ADD FOUR R1 R1    // *R1 = rRxPtr
62 READ R2 R1        // R2 <- rRxPtr
63 READFROMCODEC  CodecRxRight R2
64 ADD TWO R2 R2
65 WRITE R2 R1      // update rRxPtr

66 AND StatusCountMask R4 R5
67 TESTEQZERO R5 R5
68 JUMP IFFALSE  0 1

69 OR RxReadyMask R4 R5
70 WRITE R5 R3    // set RX ready bit

71 SendIrq2Dsp
72 JUMP ANYCASE 0 1
```

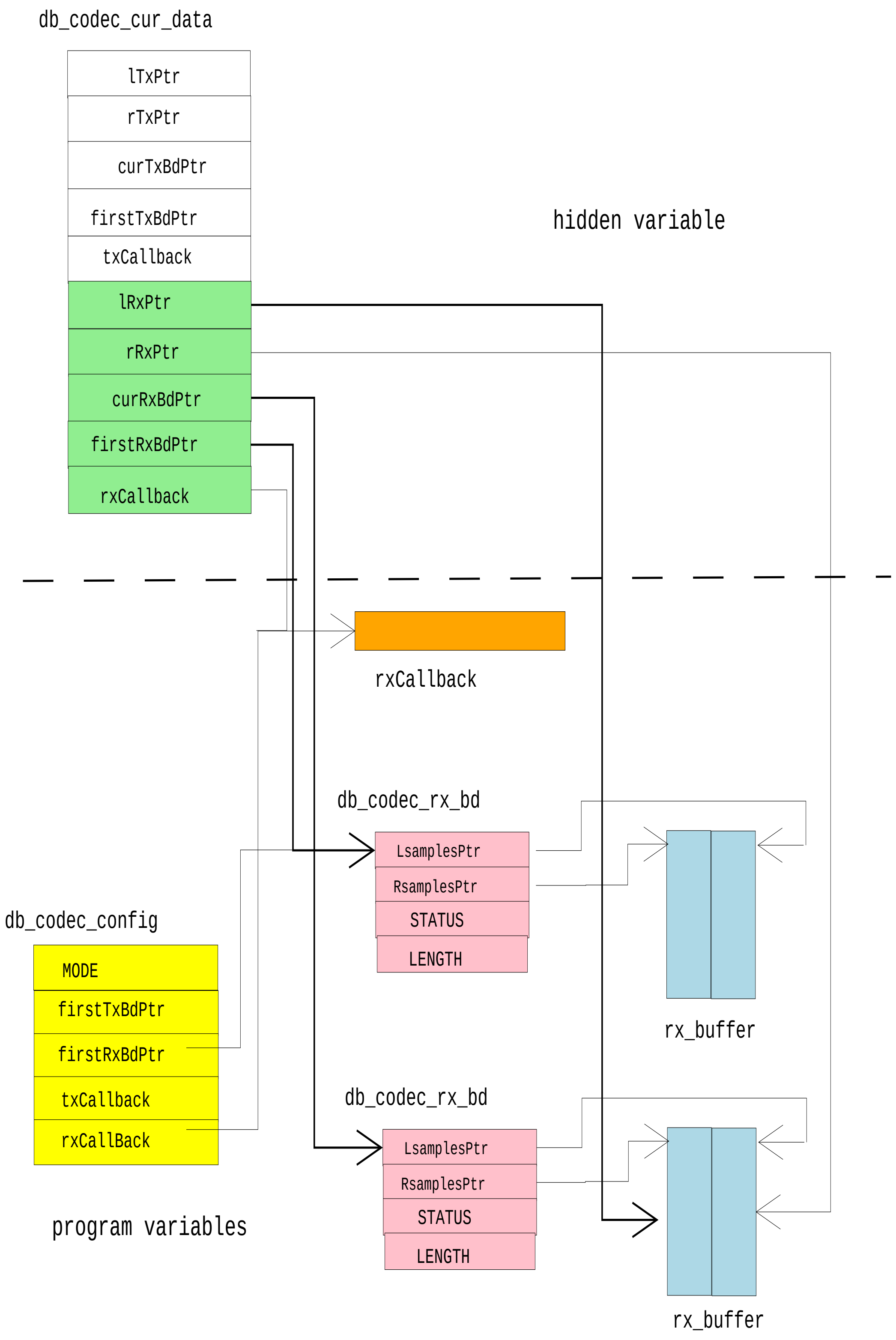

Figure 7.1: Variables in the internal memory

Chapter 8

Achievements

In this chapter I would like to present the outcome of my work in term of performance. To do so I first introduce the Adaptive-Multi Rate Vocoder that is a typical example of frame based audio processing algorithm.

Then referring to this algorithm it is possible to make some comparative evaluation of the final performance of my project.

8.1 The Adaptive Multi-Rate Vocoder

The AMR Vocoder is a standard procedure to compress and decompress the voice signal. It was standardized in the 1999 by the 3GPP, and it used in GSM, GPRS and EDGE mobiles. The word Vocoder stands for Voice Encoder and Decoder. It means that the voice coming from a microphone is compressed, transmitted and then decompressed to be played on a speaker, as shown in figure 9.1.

In this picture you can see a block diagram of a typical wireless communication. The voice compressed has to be encoded again before being sent in the channel; this is due to the characteristic of the channel that may introduce high noise and distortion.

In this kind of wireless communication system, the band is always an expensive resource. For this reason, before sending any kind of data, they

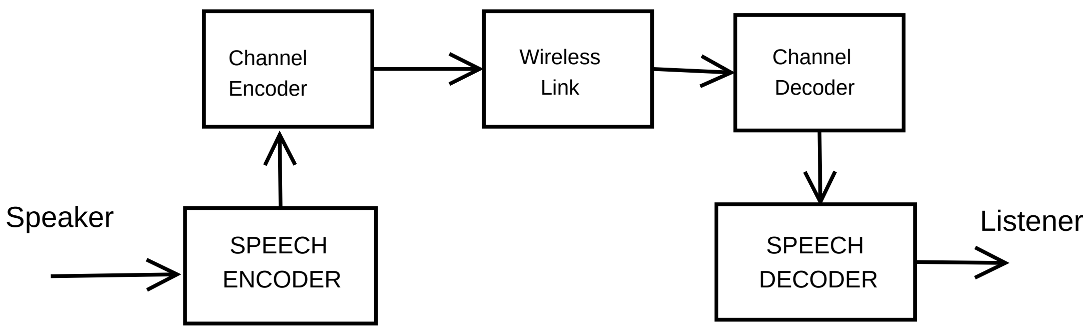

Figure 8.1: Wireless transmission block diagram

are compressed as much as it is possible; this is could be done in an effective way in the case of the voice, thanks to its particular propriety.

The AMR Vocoder employs a special model for the human speech as you can see in picture 9.2. This model is only valid for short periods of time from 2 to 40 milliseconds. According to this model the sounds generated by humans are either *voiced* or *fricative*; the first are produced by the vibration of the vocal cords and they could represented by a train impulse, where the pitch is the fundamental frequency. On the other side the *fricatives* originate as a random noise, not from vibration of vocal cords.

In both cases the sound passes through the oral cavities where it is modified; this operation is equivalent to a linear operation and therefore it could be modelled as a linear filter, with some parameters. The whole model is also considered in the research field of synthetic speech generation.

The power of this model is that, for every time frame of 20 milliseconds, just three parameters are sent; they are the voiced/unvoiced information, the pitch value, and the vocal tract response. These values are extract from every frame and sent; the decoder with this information tries to rebuild the original frame.

Another positive feature of the AMR is the ”multi-rate”; according to the quality of the channel, it can tune the degree of compression used, generating different bit rates. The standard sampling frequency is 8 KHz; being the time window 20 millisecond long, it makes every frame composed of 160

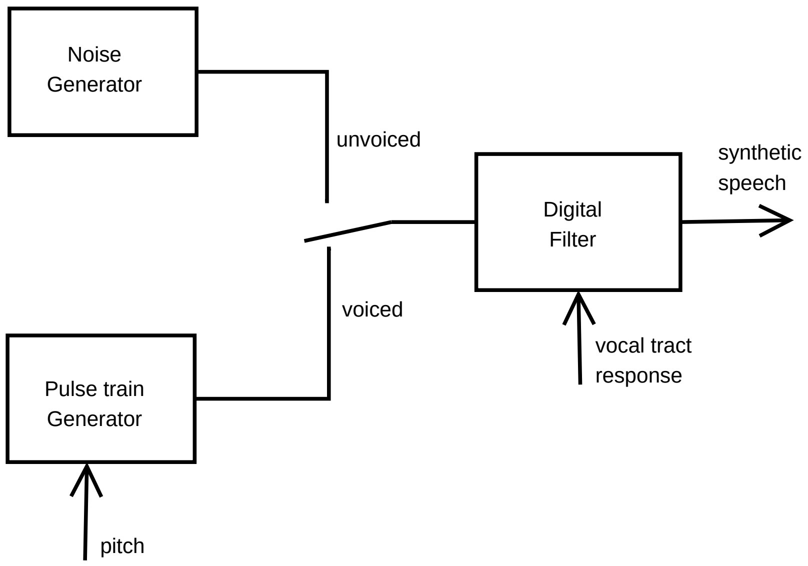

Figure 8.2: Human speech model

samples. The output sent to the channel encoder could be compressed down to a bit rate between 4.75 *Kbit/s* and 12.2 *Kbit/s*. If you consider that without the compression the bit rate would be 128 *Kbit/s*, this is a big improvement; that bit stream is transformed by the channel encoder and then sent in the channel. The actual bit rate, after channel encoding, for the only voice is then about 23 *kbit/s*; to a correct interpretation of this value take into account that the latest GSM network architectures support an about 500 *Kbit/s* channel per user.

8.2 The Evaluation of Performance

Due to many constraints, not last the lack of time, the evaluation of performance in my work was not performed as I planned at the beginning.

The original idea was to run the AMR vocoder on the Development Board with the new controller installed; in that case a measurement of the number of cycles could have been performed, and then compared to the same number with the old controller. In particular the number measured should have been the number of cycles taken to perform once the algorithm together with the IO operations.

Unfortunately to run the AMR vocoder proved to be a not straightforward task, taking time and efforts to manage issues such as memory organization and system configuration, that go beyond the purpose of my work.

Nevertheless is still possible with regard to AMR Vocoder to get some evaluation. In StarCore the AMR Vocoder was usually employed at the sampling frequency of 8 KHz and time window of 20 $msec$; this means that the buffer is made of 160 samples.

In general the overhead spent for IO operations would be the frequency that the interrupt is called multiplied for the length of the interrupt service routine that is:

$$OH = cycles_{ISR} * frequency_{interrupt} \quad [MIPS] \tag{8.1}$$

.

Consider that in this case I am going to make a comparison within the same architecture, so the measurement in MIPS is considered valid and meaningful.

In the case of the sample-based interrupt system it would be:

$$OH_{SAMPLE-BASED} = cycles_{ISR-SAMPLE-BASED} * f_{sample} \quad [MIPS] \tag{8.2}$$

On the other side, in the case of the frame-based interrupt based system the overhead would be:

$$OH_{FRAME-BASED} = cycles_{ISR-FRAME-BASED} * \frac{f_{sample}}{length_{buffer}} \quad [MIPS] \tag{8.3}$$

In these equations two values are algorithm dependent, namely the length of the buffer, and the sampling frequency. Moreover thanks to the implementation of the system we can also evaluate the length of the interrupt service routine in both cases. From the code of the two routines an estimation of their length is 250 cycles for the sample-based one and 30 for the frame-based one.

If we substitute the actual values for the AMR algorithm together with the lengths of the ISR in 9.2 and 9.3 we get:

$$OH_{SAMPLE-BASED} = 250 * 8KHz \quad = \quad 2 \quad [MIPS] \tag{8.4}$$

$$OH_{FRAME-BASED} = 30 * \frac{8KHz}{160} = 0.0015 \quad [MIPS] = \frac{2}{1300} \quad [MIPS] \tag{8.5}$$

So overall thanks to the controller I designed and implemented, the overhead was reduced by a factor 1300, down to a really negligible value. This means that the DSP processor really spends its time in an appropriate way, doing what it was designed for.

Part IV

Appendixes

Appendix A

AMBA AHB Bus description

In this appendix I want to provide an overview of the AMBA AHB bus standard. AHB stands for Advanced High-Performance Bus, and it is part of the AMBA family together with ASB (Advanced System Bus) and APB (Advanced Peripheral Bus).

The Advanced Micro-controller Bus Architecture (AMBA) is a set of specifications that define an on-chip communication standard for designing high-performance embedded micro-controller. The AMBA AHB is the most powerful standard in this family, being it suitable for high-performance, high clock frequency system modules, and it usually acts as the high-performance backbone bus.

A.1 The Architecture

In the picture A.1 there is a description of the usual architecture; there is the possibility to have in the same system multiple masters and multiple slaves. In that case an arbiter and a decoder are necessary. The arbiter deals with the requests of bus use coming from the masters; it assigns the right of using the bus to a single master at a time. On the other side the decoder selects which slave is active each time.

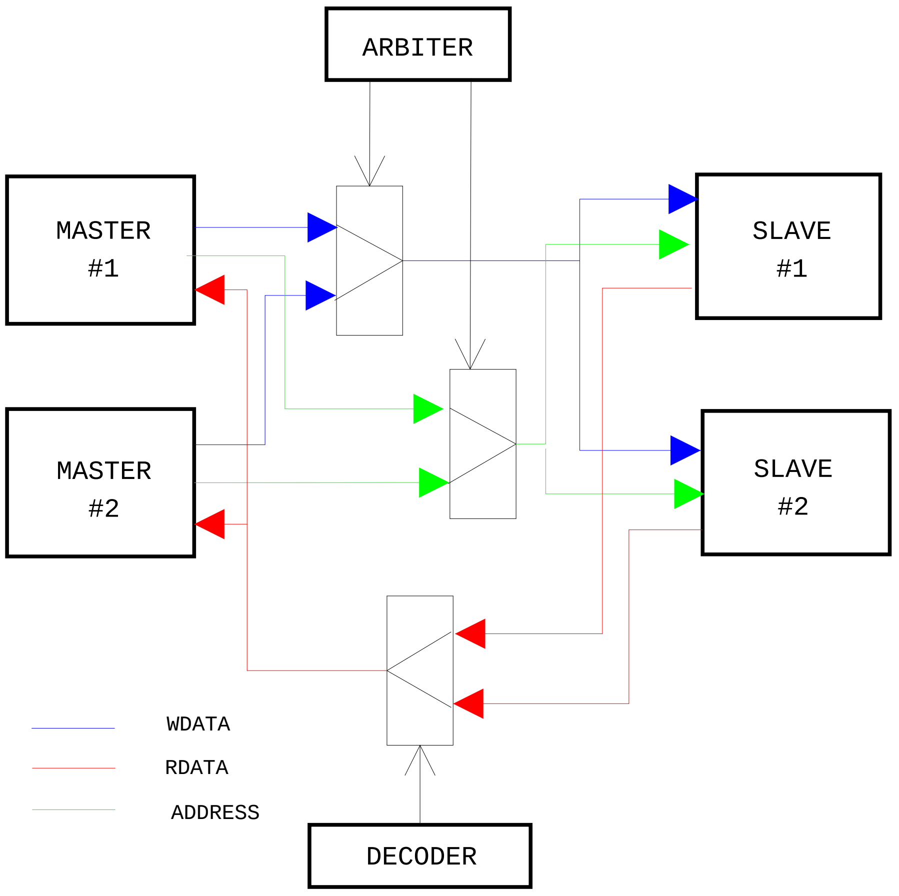

Figure A.1: Architecture of AHB bus system

A.2 The Signals and the Timing

The signals involved in the AHB standard are in the table A.2; the first three signals connect the masters with the arbiter, and their function is to assign the bus to only one master at a time. In particular, as you can see in the picture A.3, in the phase 1 and 2 the master raises *busreq*, and waits for *grant* to be raised before going on with the operation.

Once the master has the right on the bus, it starts with the proper signals for the communication; thanks to *write*, *burst*, *size* and *trans* it can specify the mature of the transfer.

Let's see in details what these signal mean:

write It specifies whether the operation is a *read* or a *write*;

size it indicates the size of the transfer, from 8 bits to 1024 bits

burst AHB offers the possibility to perform many transfers together, i.e. a burst; so this signal indicates whether the required operation is a burst operation or not and its width.

trans provides more information in the case of burst transfer, or in the case of no operation required.

After the master has provided the slave with these signals (phase 3),the slave has to give some feedback to the master, to let it know about the status of its request. This is done via the *ready* and *resp* signals together, phase 4. This phase could also be delayed by the slave if it requires time, just by keeping *ready* low.

A.2.1 The Endianness

As I explained in the COntroller chapter, endianness is an important issue in the design of bus controllers and interfaces. In the table A.2.1 and A.2.1, you can see the difference between the *little endian* and *big endian* case, in loading shirt data on the 32-bit wide bus.

name	direction	function
arbiter signals		
busreq	M to A	master request of the bus
grant	A to M	arbiter grants the bus to the master
lock	M to A	master locks the bus for its own use
slave signals		
write	M to S	direction of transfer
ready	S to M	answer of the slave
data[31:0]	bidir	data, read and write
address[31:0]	M to S	address
burst[2:0]	M to S	kind of transfer
trans[1:0]	M to S	kind of transfer
size[1:0]	M to S	size of the single transfer
sel	D to S	select the slave
clk	M to S	clock

Figure A.2: AHB signals overview, where M stands for Master, S for Slave, D for Decoder and A for Arbiter.

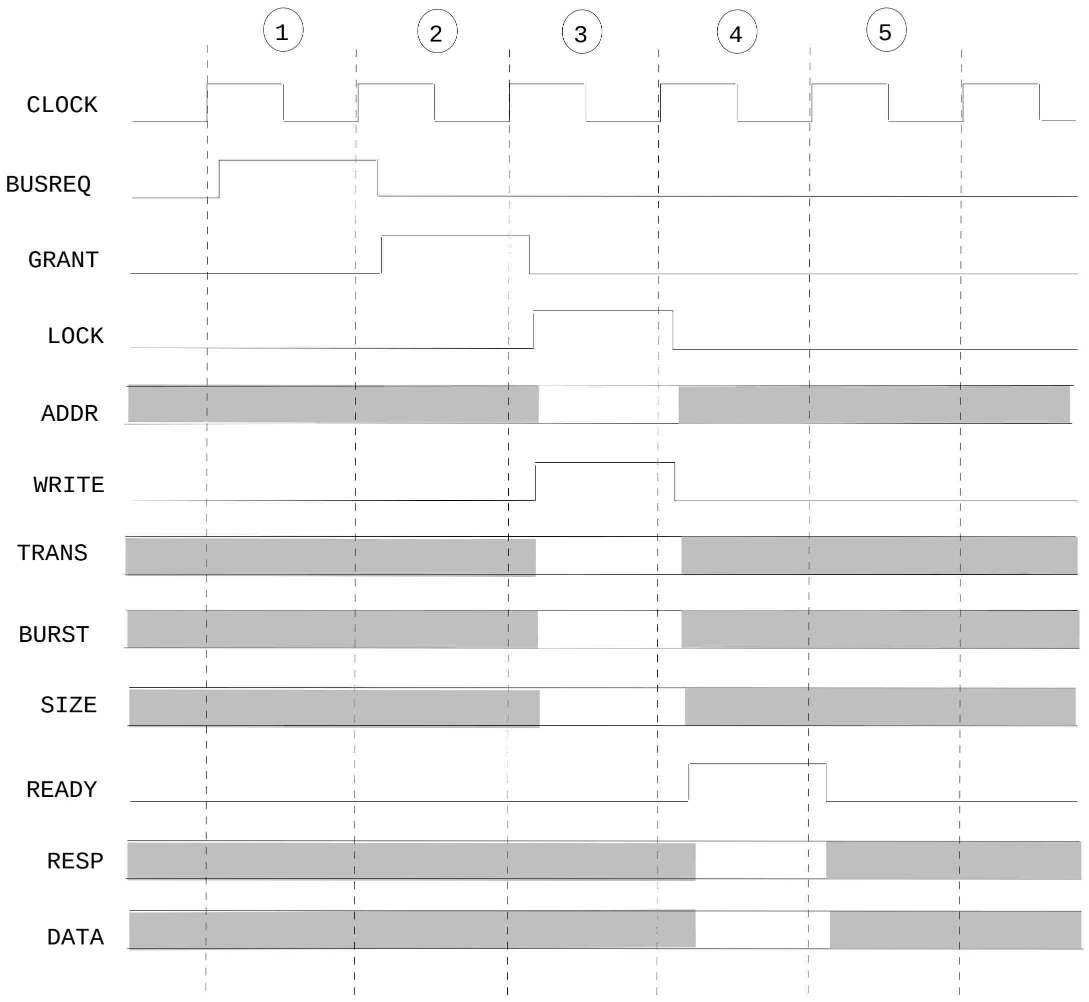

Figure A.3: Example of AHB timing for a write operation

Transfer size	address offset	Data [31:24]	Data [23:16]	Data [15:8]	Data [7:0]
word	0	x	x	x	x
halfword	0			x	x
halfword	2	x	x		
byte	0				x
byte	1			x	
byte	2		x		
byte	3	x			

Figure A.4: Active byte lanes for a 32-bit little endian bus

Transfer size	address offset	Data [31:24]	Data [23:16]	Data [15:8]	Data [7:0]
word	0	x	x	x	x
halfword	0	x	x		
halfword	2			x	x
byte	0	x			
byte	1		x		
byte	2			x	
byte	3				x

Figure A.5: Active byte lanes for a 32-bit big endian bus

Appendix B

The Audio Port Integrated Circuit

The device is an AD1849K SoundPort Stereo Codec produced by Analog Devices; it integrates the data conversion circuit with the control functions. The data conversion is composed by two stereo pairs of Sigma-Delta converters, at a frequency in the range from $5KHz$ to 48 KHz. The clock could be supplied either by external oscillators or by an external clock source, or by the serial interface.

The data are typically on 16 bit; the device support four different data types, that are the *16-bit twos-complement linear PCM*, the *8-bit unsigned linear PCM*, the *8-bit companded u-law*, and the *8-bit companded A-law*. In the system I developed the first one was always used. The choice among these types could be done during the configuration of the Codec as you will see in the next section.

To interface the outer world the Codec is provided with a six wire serial port. As you can see in table B.1, thanks to two different data wires the interface is bidirectional. Along with the clock signal, the *fsync* provides a synchronization, being high at the end of each word.

The *DC* pin is very important, since it decides in which mode the Codec must be working. There are two possible working modes: *config mode* and

wire	function	direction
SDTX	CODEC data transmitter	OUT
SDRX	CODEC data receiver	IN
SCLK	bit clock	INOUT
FSYNC	frame sync	INOUT
DC	DATA CONTROL mode	IN
RESET	reset	IN

Figure B.1: CODEC serial interface pins

data mode. In the former the Codec is always a slave in the serial communication, while in the latter it could be both slave and master.

When the audio port comes out from reset, it is usually put in the *config mode*, by clearing the *DC* bit. In this phase the desired configuration is loaded, and then it should be turned to data mode. This happens to be a complex operation, because it includes also the switch from slave to master configuration. In my system this operation is performed by the *ISR* and it described in details in the section 8.3.1.

To test the device two possible *loopback* modes are available. The first is completely digital, the input to the Codec, is replied back directly on the output. The second is also analog; the digital is converted in analog and then back in digital. These functionalities proved to be very useful and they were both employed in the test phase. The serial interface is directly connected to control or data registers; the word sent through the data wires is always 64-bit long and depending on the mode it could come either from the data registers or from the control registers.

When in *config mode* by writing in the control registers you can set the following configuration for the upcoming *data mode*:

DF1:0 data format selection

MS master slave select

MCK2:0 clock source for Codec internal operation

DCB data control bit, see 72

DFR2:0 data conversion frequency select

ST stereo mode select

ENL enable loopback testing

ADL loopback mode

On the other side when the Codec is in *data mode* the 64-bit word is composed by two 16-bit samples, plus other 32 online control bits. With these you can amplify the analog incoming signal, or attenuate the output signal. These functionalities were not actually used in my project.

Appendix C

Verilog appendix

In this appendix I show the Verilog code of the central state machine of the control part. The function of this module is to regulate the flow of each instruction, fetch, execution, new address generation.

```
//****************************************************************
//
//                      module:   ctrl_sm
//
//****************************************************************

module ctrl_sm (
    debug,
    clk,
    Nreset,
    enable_new_instr,
    enable_execution,
    addr_en_1,
    addr_en_2,
    busy_flag_reg,
    );
```

```
// input and output definition
input debug ;
input clk;
input Nreset;
input[3:0] busy_flag_reg ;
output addr_en_1 ;
output addr_en_2 ;
output enable_new_instr ;
output enable_execution ;

// declaration of internal signals
reg enable_new_instr_reg ;
reg goflag_reg ;
reg addr_en_1_reg ;
reg addr_en_2_reg ;
reg enable_execution_reg ;
reg[3:0] sm ;

// description of the internal architecture of the module
assign enable_execution = enable_execution_reg ;
assign addr_en_1 = addr_en_1_reg ;
assign addr_en_2 = addr_en_2_reg ;
assign enable_new_instr = enable_new_instr_reg ;

always @( posedge clk or negedge Nreset )
    begin
         if (Nreset == 1'b0) sm <= 4'b0000 ;
         else
        begin
             case (sm)
```

```
            0: sm <= 1 ;
            1: if (debug == 1'b1) sm <= 1 ; else sm <= 2 ;
            2: sm <= 3 ;   // FETCH
            3: sm <= 4 ;   //BEGIN EXECUTION
            4: if ( busy_flag_reg == 4'b0000 ) sm <= 4 ; else sm <= 5 ;
            5: if ( busy_flag_reg == 4'b0000 ) sm <= 6 ; else sm <= 5 ;
            6: sm <= 7 ;   // ADDRESS GENERATION
            7: sm <= 8 ;
            8: sm <= 0 ;
                     endcase
        end
    end

always @(sm)
    begin
      case (sm)
        0: begin
             enable_execution_reg <= 1'b0 ;
             enable_new_instr_reg <= 1'b0 ;
             addr_en_2_reg <= 1'b0 ;
             addr_en_1_reg <= 1'b0 ;
           end
        1: ;
        2: enable_new_instr_reg <=1'b1;
        3: begin
              enable_new_instr_reg <= 1'b0 ;
              enable_execution_reg <= 1'b1 ;
           end
        4: enable_execution_reg <= 1'b0 ;
        5: ;
        6: ;
        7: addr_en_1_reg <= 1’b1 ;
        8: begin
              addr_en_2_reg <= 1’b1 ;
              addr_en_1_reg <= 1’b0 ;
           end
        endcase
    end

endmodule
```

Appendix D

Glossary

Here you can find a glossary of terms I used in this document, in particular acronyms, with their explanation

AMR Adaptive Multi-Rate Vocoder; a voice compression and decompression algorithm used in GSM mobiles.

DB1000 Development Board 1000. It is a Development Board produced by Lyrtech, that hosts a Test Chip, many communication peripherals, like USB2.0, UART, JTAG, RS232, SDRAM memory banks.

DMI Direct Memory access Interface; this module is on the Test Chip and it stands amidst the ODB and the internal SRAM.

DRAM Dynamic RAM; RAM using capacitors (needs periodic refresh)

DSP Digital Signal Processor; processor specialized for signal elaboration, like audio and video algorithms, compression and decompression.

FIR Finite Impulse Response filter ; a kind of digital filter which uses a truncated impulse response.

FPGA Field Programmable Gate Array;

IP Intellectual Property

JTAG Joint Test Action Group; standard serial test interface.

ICU Interrupt Control Unit.

ISR Interrupt Service Routine.

ODB Off-Chip DMA Bus; it is on the DB1000, and it connects the DMI on the Test Chip with the User FPGA.

OSB Off-chip System Bus; it is on the DB1000, and it connects the Test Chip with the User FPGA.

SDRAM Synchronous Dynamic RAM.

SRAM Static RAM; RAM using active cells. Compatible with standard CMOS process.

TC Test Chip; it is an integrated circuit that contains the DSP core and other subsystem blocks like an internal SRAM, data and program caches, external memory interfaces, and the DMI. It uses a 130 nm CMOS technology, and it could speed up to 320 KHz.

VLIW Very Long Instruction Word

Bibliography

[1] ARM Ltd **AMBA Bus Specification** Rev 2.0

[2] StarCore Llc**Test Chip Specification**

[3] StarCore Llc **SC1000 Reference Manual**

[4] Lyrtech Signal Processing **Development Board DB1000**

[5] Margarida F. Jacome, Helvio P.Peixoto **A Survey of Digital Design Reuse** IEEE Design & Test of Computers

[6] Berkeley Design Technology Inc. **Evaluating DSP Processor Performance**

[7] Jennifer Eyre, Jeff Bier, Berkeley Design Technology Inc. **The evolution of DSP processors** IEEE Signal Processing Magazine

[8] Analog Devices **AD1849K SoundPort Stereo Datasheet**