Master Cycle Semester Project

Modelling of the DMA of an IEEE 802.11 MAC in SystemC

Adaptation of Design Flow to New Synthesis Tool and Architecture

Author: Christoph Eggimann
Supervision: Serge Lopez, Olivier Croset
Associated Professor: Prof. Yusuf Leblebici

Swiss Federal Institute of Technology, Lausanne (EPFL)

Lausanne, 19.6.2006
Abstract

An existing, complete hardware model of the direct memory access block of a IEEE 802.11 medium access controller, written in SystemC, was adapted to incorporate a subset of the original functionality for specific applications. The proposed model requires less than half the area of the original design on an FPGA and has been verified using simulation. Additionally, an adapted design flow from Register Transfer Level models towards an FPGA implementation using a new synthesis tool is proposed. The new synthesis flow could be verified for the complex design of the entire hardware medium access controller of concern in this project.
List of Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMBA</td>
<td>Advanced Microcontroller Bus Architecture</td>
</tr>
<tr>
<td>ASIC</td>
<td>Application Specific Integrated Circuit</td>
</tr>
<tr>
<td>BBIF</td>
<td>Baseband Interface</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>DC</td>
<td>Design Compiler</td>
</tr>
<tr>
<td>DC FPGA</td>
<td>Design Compiler FPGA</td>
</tr>
<tr>
<td>DMA</td>
<td>Direct Memory Access</td>
</tr>
<tr>
<td>DSP</td>
<td>Digital Signal Processor</td>
</tr>
<tr>
<td>EDA</td>
<td>Electronic Design Automation</td>
</tr>
<tr>
<td>EDIF</td>
<td>Electronic Design Interchange Format</td>
</tr>
<tr>
<td>EPFL</td>
<td>Swiss Federal Institute of Technology</td>
</tr>
<tr>
<td>FIFO</td>
<td>First In First Out Queue</td>
</tr>
<tr>
<td>FF</td>
<td>Flip-Flop</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field Programmable Gate Array</td>
</tr>
<tr>
<td>HDL</td>
<td>Hardware Description Language</td>
</tr>
<tr>
<td>HMAC</td>
<td>Hardware Medium Access Controller</td>
</tr>
<tr>
<td>IMEC</td>
<td>Interuniversity Microelectronics Centre</td>
</tr>
<tr>
<td>IP</td>
<td>Intellectual property</td>
</tr>
<tr>
<td>LAN</td>
<td>Local Area Network</td>
</tr>
<tr>
<td>LLC</td>
<td>Logical Link Control</td>
</tr>
<tr>
<td>LSM</td>
<td>Microelectronic Systems Laboratory</td>
</tr>
<tr>
<td>LOG</td>
<td>Log directory</td>
</tr>
<tr>
<td>LUT</td>
<td>Look-Up Table</td>
</tr>
<tr>
<td>MAC</td>
<td>Medium Access Controller</td>
</tr>
<tr>
<td>MIC</td>
<td>MAC Interface Controller</td>
</tr>
<tr>
<td>MLB</td>
<td>Master Local Bus</td>
</tr>
<tr>
<td>MSDU</td>
<td>Medium Access Controller (MAC) Service Data Unit</td>
</tr>
<tr>
<td>NCF</td>
<td>Netlist Constraint File</td>
</tr>
<tr>
<td>NGD</td>
<td>Native Generic Database</td>
</tr>
</tbody>
</table>


**OSCI** Open SystemC Initiative

**PCI** Peripheral Component Interconnect

**P&R** Place & Route

**QoS** Quality of Service

**RBI** Register Block Interface

**RF** Radio Frequency

**R&D** Research & Development

**RT** Register Transfer

**RTL** Register Transfer Level

**Rx** Receive

**SoC** System on Chip

**SNAP** Subnetwork Access Protocol

**SM** State Machine

**SRAM** Static Random Access Memory

**ST** STMicroelectronics

**STA** Client Station

**Tcl** Tool Command Language

**Tx** Transmit

**VHDL** Very High Speed Integrated Circuits Hardware Description Language

**VHSIC** Very High Speed Integrated Circuit

**VLSI** Very Large Scale Integration

**WiFi** Wireless Fidelity

## Typesetting Conventions in this Document

The following typesetting conventions are maintained throughout this document if they appear in normal text.

- *Signal names*
- *Command names*
- *File names*
- *Product names*
- *Company names*
- *Acronyms*
1 Introduction

This is the final report about a semester project effectuated at the Microelectronic Systems Laboratory (LSM) at the Swiss Federal Institute of Technology (EPFL). The work consisted of adapting an existing hardware description model of the direct memory access of an IEEE 802.11 wireless medium access controller in a first part and then the adaptation of the overall design flow to a new synthesis tool in a second part.

This project report will first introduce the reader to some basic background information needed to understand the circumstances of the present project. In section 2 will therefore quickly summarize the IEEE 802.11 standard and talk about hardware description languages, in particular SystemC which was extensively used in this project. To conclude this section, a discussion about the prototyping of ASICs using FPGAs follows.

Section 3 will introduce the Micmac project in general first and then enter with some detail into the building blocks that are concerned by the modifications done during this project. In section 4, I will discuss the first part of this semester project related to the direct memory access hardware model. The existing, complete model will be explained and the specifications for the simplified model debated. To conclude this section, the chosen implementation and results will be discussed.

Section 5 is focusing on the second part effectuated during this project, namely the adaptation of an existing synthesis flow to a new tool. To do this, I first need to explain the old synthesis flow and then explain the new file structure, all new instructions and the adapted design flow in detail.

Last but not least, a perspective upon future tasks following this project will be given, discussing further possible improvements of the current device architecture. I also decided to include the most important source code files as an appendix in this report as a quick reference. Furthermore, parts of this report are compiled to form a self containing document serving as a documentation for the new design flow [20].

Readers not familiar with the background of the present project best read through the introductory sections 2-3. Persons already familiar with the technological background in general and the Micmac project in particular can skip these parts and directly begin with section 4 where the discussion of the main work done during this project is described.
2 Background

2.1 Hardware Description Languages

During the 1980s, companies in the hardware design segment were confronted with increasingly complex designs. There was a need for powerful software that was able to simulate the behaviour of the software at a higher level of abstraction in a way that complex designs could be modelled and their functionality verified using behavioural models. The programming languages used to fulfill this tasks are commonly called Hardware Description Languages (HDLs).

2.1.1 VHDL and Verilog

The two predominant HDLs today are the Very High Speed Integrated Circuits Hardware Description Language (VHDL), and Verilog. VHDL was initiated in the 1980s by the American department of defense to document the functionality of advances chips of their Very High Speed Integrated Circuits (VHSIC) program [19]. Modularity and hierarchy allowing different levels of abstraction were defining characteristics of VHDL. The language was based on the programming language Ada and therefore intended to be used in close conjunction with software development. Another intention of VHDL’s promoters was to develop a standard for the evolving Electronic Design Automation (EDA) industry for their hardware descriptions.

The competing language, Verilog, has its roots in 1985 at a company called Gateway Design Automation, which was later acquired by Cadence Design Systems. Verilog is closely related to the C programming language and was very successful in the beginning although being proprietary. When VHDL became largely available, Cadence no longer kept the proprietary rights and opened Verilog in 1990.

Today’s use of HDLs is not only in system modelling/simulation, but also in synthesis of designs towards a description using standard cells to build semi-custom Application Specific Integrated Circuits (ASICs) or, more recently, mappings on Field Programmable Gate Arrays (FPGAs). Interestingly, synthesis was not an issue in the beginning. On the verilog.com homepage, it can be read: "Verilog was invented as simulation language. Use of Verilog for synthesis was a complete afterthought"[10]. This explains why only a subset of the language is today synthesizable and another big part is used for simulation only.

Many vendors offer tools that address specific issues in the design automation flow, from pure simulation tools to synthesis tools in all varieties. The major players are Cadence and Synopsys offering complete design suites, but there are a lot of smaller companies that are specialized in specific segments of the design flow. VHDL and Verilog are widely used today for complex designs modelling, verification and synthesis. Therefore, hardware design would not be possible at all without those powerful languages. Yet, these two dominant languages have got another competitor in recent years: SystemC.

2.1.2 SystemC

Both VHDL and Verilog are in use already for quite a long time. The languages have evolved a lot since then, and while they were first used for simulation, synthesizable subsets have later been defined and are now heavily used to design
2.1 Hardware Description Languages

**Figure 1:** System on Chip (SoC) vision. The chip shown includes CPU, on chip buses and drivers for input/output, dedicated memory, a specialized DSP, a MAC and an analog RF frontend. The Micmac block as presented in section 3 could be used in such a system.

complex systems that cannot be implemented by a custom Very Large Scale Integration (VLSI) designer anymore. Much of the work when designing VHDL models is done at a Register Transfer (RT) level, allowing cycle accurate simulations. For many applications, this is exactly what is needed: The system can well be decomposed into smaller subblocks which can then be implemented according to well defined interfaces between the blocks. This synthesis flow was working well as long as the overall system was a system-on-board, meaning that the system engineer was basically putting together several chips on a board, including Central Processing Units (CPUs), memory, Digital Signal Processors (DSPs), Analog devices, complex bus drivers and many more. Today, the System on Chip (SoC) has become a reality [22]. The components are not connected on a board anymore, but put on a single silicon die. A vision of a complete SoC for wireless broadcasting is visualized in figure 1. The designs are quickly becoming heavily complex. While a division into subblocks is still the most feasible way to cope with the complexity, there is a need of powerful software support that helps the chip designers fulfill their tasks. Often, a simulation of the entire system at an Register Transfer Level (RTL) level is not possible anymore, simulation would become too slow and often it is not even necessary to simulate a design at such a low level. Modelling allowing changing degrees of detail is needed, including high level models and abstract definitions down to RTL level in a single hardware description language. Intellectual property (IP) is another issue that needs to be discussed. While formerly a chip simply contained a single design by one design company, SoCs today incorporate often many different building blocks that are IP of different companies. An example for this are entire processor cores that can be bought on the market for implementation on an FPGA.

**SystemC** arose much motivated due to the reasons outlined above. Different commercial companies (Synopsys, Infineon, Frontier Design) in collaboration with universities (Interuniversity Microelectronics Centre (IMEC) from Leuven, Belgium, University of California, Irvine) and an open source initiative called Open SystemC Initiative (OSCI) were responsible for pushing forward the development of SystemC in the early years of the new millennium.
The language addresses, besides the issues discussed above, also ease in implementation on a software level and therefore software/hardware codesign [30]. It was because of the good experience with object oriented programming languages that SystemC was based on the C/C++ language, which is used extensively by computer scientists, but also by system designers in leading companies. SystemC comes as a C++ library enhancing the capabilities of the language, but finally the simulation is based on an executable C file. Other than the interpreted languages VHDL and Verilog, SystemC can therefore achieve better simulation times once it has been compiled as it can be run natively on any processor for which a C compiler exists.

Since SystemC is designed to allow the use of different levels of abstraction, obviously also the most detailed level must be supported, namely RTL [29]. This fact is important since the major work of this semester project was writing a model at RT level. However, the way SystemC RTL code is written does not differ substantially from coding styles in VHDL or Verilog [14]. There are some particularities in the syntax of SystemC, which mainly result of the fact that the language really is a C++ library, satisfying the most basic C++ language syntax. Architecture and Entity as known from VHDL are now replaced by macro definitions called Modules. In reality, the macros make use of the object oriented nature of C++ and derive an abstract data type of the basic SystemC module, inheriting all its properties. Instantiation as known in other HDLs is readily available in C++, making it possible to derive many internal implementations of a given module.

Vendors from the EDA industries have taken notice of the new upcoming language, and tools providing all kinds of functionality are now available. First, there are many tools allowing design simulation and verification, others offering translation from SystemC to VHDL/Verilog or the opposite direction. While the translation to VHDL/Verilog allows the SystemC based models to be directly used with standard synthesis tools such as Design Compiler of Synopsys for ASIC or tools by Synplicity for FPGA implementations, there now exists commercial tools offering direct synthesis from SystemC RTL or behavioural models. Examples thereof include Synthesizer by Forte Design System, Triton Builder by Poseidon Design Systems, CoCentric by Synopsys or Agility Compiler by Celoxica. An up-to-date list of available tools can be found on the official OSCI SystemC webpage.[9] Although RT level modelling is possible using SystemC, most designers still stick on classical HDLs for low level modelling and see the advantage of SystemC solely on an architectural level, as regular surveys by DeepChip’s John Cooley reveal[1]

2.2 FPGA prototyping of ASIC designs

Since this project’s work will also include a block dealing with the synthesis of an entire design to be mapped onto an FPGA, I will need to explain to some extent the interest in doing so and thereby address the issue of FPGA prototyping of ASIC implementations. ASICs are today still the working horse of the microelectronic industry in terms of revenue generated, which is also reflected in sales of EDA vendors. However, more recently, programmable devices

---

1DeepChip [4] is a website specialized in offering information about the EDA industry. Check the SystemC surveys [3, 2].
have become a viable alternative for many applications needing more flexibility and adaptability. As the modern FPGAs offer an extreme amount of logic gates, Flip-Flops (FFs) and more complex blocks such as adders, they can be used to implement a large amount of ASIC designs. Historically, the design software for programming the devices was offered directly by the FPGA vendor. The classical EDA companies were focussing on ASICs, which were generating more revenue. For the FPGA designs, Synplicity is claimed to have the most powerful tools achieving better timing and area results than the competitors. For ASIC design synthesis on the other hand, Synopsys is the dominant leader offering the industry approved product Design Compiler.

Although having made large steps forward, FPGAs will still not replace ASICS. There is still a factor of about 20 to 40 that is lost in area if using FPGAs over ASICs, depending on the design [24]. Similarly, more power is consumed and the timing is slower for the same process technology. While FPGA vendors are trying to tackle those problems and further close the gap to ASICS, more designers are beginning to use FPGAs for prototyping and an additional verification step before porting the design on an ASIC. Until recently, this led to severe difficulty in porting FPGA designs to an ASIC since different tools were involved. There is little interest from the FPGA vendor’s side to offer tools making such transitions easier as they would loose on FPGA sales. Recently, SYNOPSYS has approached the FPGA prototyping strategy as the company has realized the powerful possibilities such a design flow can offer [28].

Today, FPGA prototyping has become a reality. The present Micmac project that will be discussed in the next section, is an excellent example of such a design flow. STMicroelectronics typically uses Research & Development (R&D) engineers that experiment on new designs and realize FPGA demonstrators to verify the working principle. At the same time, a fully functional chip running close to the actual ASIC implementation speed can be used and design flaws can be detected in time. The designs that are approved for ASIC production are
then adapted to such an implementation. The design flow has of course to be changed, but since the designs have already been verified using the FPGA, the time to market from this point on can be quite fast. For the future, it can be expected that even more design companies are using FPGA prototypes with a design flow based on the same tool for both FPGA and ASIC implementation. SYNOPSYS' DESIGN COMPILER is the most promising candidate at this moment to be universally used. It has the huge advantage that it is already widely accepted for ASIC designs and now offers the FPGA synthesizer DC FPGA that is claimed to have caught up with SYNPLICITY's tools by SYNOPSYS. DC FPGA will be used in this project for the synthesis of complex VERILOG and VHDL RTL models for FPGA.

2.3 The IEEE 802.11 Standard

Wireless networks are today of great importance and the numbers of computers connecting to such wireless networks is steadily increasing. There exist several standards defining wireless communication, but by far the most important one for Local Area Networks (LANs) is the IEEE 802.11 standard, commonly known as Wireless Fidelity (WiFi). The standard is available online [5] and defines extensively the way a system accesses the medium (air), what the network structures are and so on.

The main layers defined by the standard are the PHY, meaning the physical level, and the medium access control (MAC) level. The PHY basically defines the way the information is coded which is sent through the air. The frequencies at which wireless networks are allowed to work depend on the country, but the bands are mainly in the 2.4 GHz range. The 802.11 standard knows several sub-standards, identified by added letters. The upper throughput limit of the connection is set to 11 MBit/s for the 802.11b standard and 54 MBit/s for 802.11g. The MAC level is responsible for properly setting up the packets that are sent for identification, data transmission and control information exchange. The Micmac project this semester work is part of and which is discussed in section 3 is one possible realisation of the MAC layer. For more information about the IEEE 802.11 standard, the interested reader is referred to the IEEE website [5] or the reference books such as [21].
3 The Micmac Project

This section discusses the overall project this semester work is part of. The group working at the Swiss Federal Institute of Technology (EPFL) in this project is just a part of a collaborative project group. The basic idea and main part of the project stems from STMicroelectronics (ST), a Grenoble, France, based microelectronics giant. Several groups of ST have been involved in the project which already started in 2002, the pioneering work being done by Pascal Delamotte of ST Geneva. Groups ST Grenoble and EPFL have since contributed to the entire project, which will now be explained in some detail to understand the background of the present semester project.

3.1 Basic Idea

The medium access in a 802.11 network is very complex and has to satisfy a large number of constraints, collected in a thick specification booklet [23]. The protocol was not designed with a possible hardware implementation in mind, but mainly driven by functionality and features. As a consequence, in common implementations of 802.11 MACs, many tasks are done by software running on either the host processor or a dedicated coprocessor. One feasible implementation, offered commercially by Ittiam Systems [6], is to build the baseband interface in hardware, next to a dedicated processor on a SoC. The actual medium access controller is then implemented in software running on this processor. Some blocks of the MACs are directly feasible for hardware acceleration. The whole package is then sold as an IP block for use in custom designed SoCs. Other similar implementations can be contracted from NewLogic [7] or from Duolog [4].

While having the advantage of being flexible and trading off hardware and software according to customer’s needs, these designs consume a fair amount of power as there is an entire dedicated processor running fulfilling the MAC functionality. The idea of the Micmac project is to implement the basic MAC functionalities in a hardware design, leading to reduced power consumption and area needs, implying cost reduction as compared to a software based design. There won’t be any local memory and the data to be sent is directly read in host memory using direct memory access. This way, no host processor time is needed and the MAC can function completely autonomously apart from the initializations and setup processes done by the main CPU. The fulfillment of the core functionality in hardware is for the moment only done for 802.11 Client Stations (STAs) as the protocol for access points is a lot more complicated to implement. Furthermore, the power, area and cost reduction is first of all of interest for mobile devices and less critical for access points.

The Micmac is not designed to be used as a single chip, but for integration on a SoC together with other components. ST aims at selling the design as an IP block to chip designers. The design will allow to change some important parameters such as the size of the transmit First In First Out Queue (FIFO), but will otherwise be seen as a black box from the customer’s point of view. The key features of the Micmac are summarized in the following list, taken from [18].

- Hardware optimized for supporting IEEE 802.11 protocol as Client.
- Lower MAC protocol hardwired: no local dedicated processor.
- Direct Memory Access (DMA) access to host memory: no local shared memory.
- One transmit FIFO and one receive FIFO with programmable size tuned at synthesis level depending on application and system constraints.
- Reduced driver buffer copy before transmit.
- Supports Access Point functionality by software running on host processor.
- Hardware independent of the Host Bus (the Peripheral Component Interconnect (PCI) bus could be replaced by an Advanced Microcontroller Bus Architecture (AMBA) bus for example).
- Designed for SoC integration (not as stand-alone IP).

Additionally to these features, the Micmac in its basic configuration is able to transmit with four different priority channels and an additional channel supporting beacon frames (for broadcast management). The multiple channels are part of the Quality of Service (QoS) draft definition of the IEEE 802.11e standard. Fragmentation in transmission and defragmentation in reception is supported in hardware and does not need any host CPU support. If there is an error in a transmitted packet (no acknowledge packet is received), the packet is retransmitted automatically by the hardware (Retry functionality fulfilled without host CPU support). Overviews of the Micmac project are found in [18, 11, 17], for example.

### 3.1.1 Reduced Micmac

For specific applications, there is no need to support the whole 802.11 STA functionality. An example is a camera, which is broadcasting the recorded material directly using a wireless network. The camera only uses one or two (image and sound broadcast) transmission channels and not four. Additionally, constraints on the alignment of data in the local memory can be imposed further reducing the complexity of the medium access controller. This is where the first part of this project comes into the picture: The reduced DMA developed is to be implemented for a reduced Micmac architecture.

### 3.2 Building Blocks

The Micmac hardware is structurally divided in three main building blocks: the Hardware medium access controller (with Baseband Interface (BBIF)), the MAC interface controller and the host interface. The Hardware Medium Access Controller (HMAC) implements the core functionality needed for the 802.11 protocol implementation. The second building block is the MAC Interface Controller (MIC). The MIC allows the HMAC to access the host memory through a DMA. The third and last building block is the host interface, composed of a PCI controller and a ST proprietary Register Block Interface (RBI) and Master Local Bus (MLB) interface. Because of the cascading of MIC and MAC, the project name (and the acronym designating the entire design) has been chosen to be the concatenation of the both: Micmac. The global building blocks involved are shown in figure 3 and explained in more detail in what follows.
3.2 Building Blocks

3.2.1 The HMAC

The HMAC controls the data flow from a Transmit (Tx) FIFO to the medium and from the medium to a Receive (Rx) FIFO and thus implements the whole functionality of an IEEE 802.11 MAC layer. The HMAC is capable of setting up the whole communication needed for identification of a STA in a wireless system. Once this is accomplished, the data packets to be sent are collected from the Tx FIFO and prepared properly according to the 802.11 protocol. The HMAC is also capable of generating status information that is then written back to the host memory. Fragmented transmission and retries in case of a corrupted segment are also included in this block. The interfacing to the medium (air) is accomplished with a BBIF block, which then controls the analog front-end of the chip. The entire HMAC is today a very complex model and won’t be discussed in more detail as this project only focusses on the MIC part of the Micmac. Further information about the HMAC functionality and architecture can be found in documents [16, 18, 17].

3.2.2 The MIC

As the name implies, the MAC Interface Controller (MIC) controls the interface between the MAC and the host, handling memory access requests from the HMAC and requesting itself the transmission of packets by the host. The two FIFOs situated between MIC and HMAC, one for Tx and one for Rx. The Tx FIFO is filled by the MIC with data to be transmitted that is obtained by a direct memory access to the host memory, and read out by the HMAC. If a packet is successfully transmitted, the MIC stores a status message generated by the HMAC in the host memory. On the reception side, a similar functionality is implemented except that the Rx FIFO is this time filled by the HMAC and the data is subsequently written to the host memory by the MIC receive DMA controller. The MIC has a number of registers used to control its functioning. A view of the MIC including Tx- and Rx-parts as well as the MIC registers and the MLB interface controller are shown in figure 4. Not shown is a loopback block, which was designed for test purposes and which mimicks the HMAC behaviour, and which will also be included in the final design in a modified form for testing. The work of this semester project deals with the DMA of the transmit block of the MIC. I will therefore focus on the Tx block in the future discussion and not deal with the Rx block nor all the MIC registers in more detail. The further partitioning of the transmit block is described in section 4.1. Detailed descriptions of the MIC
blocks can be found in [27, 15].

![Figure 4: MIC building blocks](image)

### 3.2.3 Host Interface

As the Micmac is designed to be used as an IP block to be included in different SoCs and not as a stand-alone chip, the design must be adapted for use with different host buses. The solution chosen by ST is to design an interfacing bus which can deal with common buses such as AMBA, PCI or STBus. This interfacing block thus needs to be designed just once and different IP designs can use it to access the host bus. ST divided the interfacing bus block into a part that deals with the IP registers, called RBI, and one part, called MLB, which allows the IP to access the host memory. The IP designer does not need to deal with the various system buses in such a way and only needs to know the MLB and RBI protocol. The MLB protocol, which will be of concern in this project for the direct memory access allows memory reads/writes in both single block and burst mode. The bus is arbitrated using a request signal and a subsequent acknowledge signal shows the availability. In the case of a read, a data ready signal indicates that the data line can be read in. Precise specifications of the MLB and RBI bus can be found in [8]. The Micmac project currently works with a PCI host bus used on PCs and consequently a PCI interface IP block of ST is included in the synthesized design.
4 DMA Implementation

This section presents the first part that has been effectuated for this semester project, namely the modelling in SystemC of a DMA block with reduced functionality decreasing the area needs of this constitutive block of the MIC. First, the transmit block of the MIC will be further explained and the functionality of the DMA as it has already been designed will be discussed. Next, the specifications for the reduced DMA are summarized and I will then explain the chosen implementation in detail. As a final part in this section, the results of simulation and synthesis will be shown and commented.

4.1 Transmit Block of the MIC

The constitutive blocks of the MIC were shown in the last section. As this project focuses on the DMA of the transmit block, I will explain this part in more detail. A simplified schematic of the transmit block of the MIC is shown in figure 5. The major components are a packet manager, the DMA block, and the transmit FIFO. To ensure the handling of the difference in clock frequency between HMAC (22 MHz) and MIC (33 MHz), a Tx sync block is used. The Tx block also contains a number of registers that can be accessed from outside, which are regrouped in the Tx Regs block. The interfacing towards the rest of the MIC and the host is done by accessing the MIC registers or the host interface (PCI) by using the MLB interface.

The packet manager consists of not only one, but five instances, used for offering priority in the sending of packets. The outputs of the different packet managers are multiplexed, so that only one is commanding the Tx DMA at a time. The Tx DMA block has an internal architecture using a DMA State Machine (SM) block, a FIFO filler and an address generator. Those blocks will be further discussed below. The Tx FIFO finally is a hardwired stack of registers of a size that is determined at synthesis time.

![Figure 5: Transmit Block of the MIC. The block interactions are simplified to show the main working principle](image-url)
Transmission of a packet  A FIFO containing pointers to the beginning of all prepared packets to be sent is implemented in software, thus located in the host memory. Each entry of this FIFO is a pointer pointing to the address of the first byte of a packet that had been prepared by the host for transmission in the host memory, representing the beginning of the 802.11 header. After the header, the following 8 bytes contain information about the next segment. The first bit indicates whether the next segment will be the last one (if the bit is set to a 1) or if more segments will follow thereafter (if the bit is set to a 0). The remaining bits contain the size of the following segment as well as the address where it begins as can be seen in figure 6. In this fashion, a packet can in principle be extended to an arbitrary number of segments, each of which is containing two bytes identifying its successor. However, in practice the number of segments is reduced to three, namely the header segment, a segment containing an 802.11 Logical Link Control (LLC)/Subnetwork Access Protocol (SNAP) header and the third being the actual payload.

The read- and write-positions of the packet pointer FIFO are stored in form of two pointers in registers of the MIC, called `tx_pkt_fifo_rd` and `tx_pkt_fifo_wr`, respectively. The value pointed by `tx_pkt_fifo_rd` is therefore a pointer to the address of the first byte of the next packet to be transmitted. Once a difference in the two pointer addresses `tx_pkt_fifo_wr` and `tx_pkt_fifo_rd` has been detected, meaning that the host has prepared a packet for transmission and advanced the read pointer, the packet manager of the Tx block of the MIC initiates the transmission of this packet, storing the current packet pointer in the MIC signal `tx_pkt_ptr`. The packet manager, itself commanded by the HMAC is connected directly to the Tx DMA block and can put the DMA to either of its three operating modes, namely header transmission, payload transmission and status writeback using the three signals `PM_start_header`, `PM_start_payload` and `PM_start_status`, respectively.

As a first step upon detection of the host intention to send a packet, the packet manager thus commands the DMA to transmit the header to the HMAC, applying a rising edge to the `PM_start_header` signal. The DMA is then in the first mode of operation, the transmission of the header. As a consequence, the DMA then begins to read the header, directly accessing the host memory using the MLB interface. The data read is directly written to the Tx FIFO.

The second step in the transmission of the packet, after the header has been sent successfully, is the transmission of the payload. The packet manager is commanded by the HMAC to assert the `PM_begin_payload` signal, putting the DMA in the second mode of operation. The DMA then reads the payload from the host memory and puts it into the Tx FIFO until the whole packet has been sent.

The third and last mode of operation of the Tx DMA is writing a status word indicating the successful or failed transmission back to the host memory. This status information is generated by the HMAC and written on a software FIFO containing the transmission status information, using the `tx_status_fifo_wr` pointer. The host can then analyze this status information and initiate appropriate measures.
4.2 Complete DMA

So far, the detailed structure of the packets has not been described. In fact, the host memory is accessed in blocks of 4 bytes or 32 bits at a time through the MLB interface. According to the 802.11 standard however, this alignment on four bytes is not necessarily satisfied in the host memory and payloads can well begin and end at any integer numbered byte address. The DMA as implemented in the old version of the Micmac was well considering this fact and allowed the treatment of any payload imaginable. The handling of the problem was solved by reading in data by four-byte blocks and then adding a 4 bit array indicating the validity of each of the bytes to the HMAC. Obviously, handling all the cases of alignment became quite complicated, as can be seen in [27], p.26ff.

4.3 Reduced DMA

The Micmac with reduced functionality has been discussed already in section 3.1.1. The goal of such an implementation is to further reduce the area and power needs of the design. When looking at the different block sizes, it can be seen that the Tx DMA is the largest of the MIC blocks and therefore the most feasible attack point for a redesign. The specifications of the DMA have been adapted as described next.
4.3 Reduced DMA

4.3.1 Specifications

The specifications of the reduced DMA can be found in the document [25]. This document covers most points in the transmission of a single fragment, however fragmented transmission and retry are not covered. As a consequence, there have been several points that have changed upon agreement with Serge Lopez and lead to the actual implementation done in this project. The main specification changes as compared to the old DMA are the following.

- Aligned payload: The payload content is not anymore aligned arbitrarily in the host memory, but is now aligned either on byte one or byte three of a four byte block.

- Aligned fragments: All fragments are similarly aligned, leading to fragment lengths that are multiples of two bytes.

- Alignment bits omission: As a consequence of the above restrictions, the alignment bits can now be omitted.

- One segment payload: The payload is defined to be contained in one single segment. In such a way, the overall number of segments is three (header, LLC/SNAP header, payload).

- Fragmentation handling: The fragmentation is handled entirely by the DMA without talking back to the packet manager.

- State machine changes: The specification document [25] shows a proposed state machine. As a consequence of the complete redesign, the state machine has been slightly adapted.

4.3.2 Implementation

This section will describe the chosen implementation. The DMA consists of three main components, being the DMA SM, the address generator and the FIFO filler. This global structure has not been changed as such, while some signals are not used anymore and can be removed in the future. The redesign mainly addresses the DMA SM block, which is at the same time the largest of the three constitutive blocks of the DMA.

To obtain an overview of the external signalling of the DMA, consider figure 7. The structure shown also includes the internal signalling of the block. Note that signals that had been included in the original DMA and which are not used anymore are shaded in gray whereas signals still needed are highlighted in black. The meaning of each individual signal can either be found in the DMA specifications [25] or directly in the commented source code as far as the DMA SM is concerned.
4.3 Reduced DMA

Figure 7: External signals of the Tx DMA, together with the internal signalling between the DMA Statemachine (SM), address generator and FIFO filler. Signals not used anymore in the new (reduced) implementation are shaded in grey. Signals representing buses use bold lines. A few additional register block signals are not shown.
The first step in the adaptation of the reduced DMA is a redesign of the DMA SM. The new SM is shown in figure 8 with signals indicating the transitions between states. To simplify the interpretation of this graph, figure 9 shows essentially the same graph, this time giving annotations of the transitions. While the header transmission part and the status write have been maintained as they were, slight changes and regrouping have been done for the write payload part. Each transition between states is now associated to a clear event that can be identified easily and captured in a descriptive sentence as has been done in figure 9.

The first step in the implementation was the reading of the header. There have not been many changes in the way of dealing with this read. There are
4.3 Reduced DMA

DMA IMPLEMENTATION

Certainly changes in the naming of internal signals, but basically the working principle has been kept the same as in the original implementation.

As for the transmission of the payload, there have been done the main design changes and therefore this mode of operation will be discussed in more detail. The first two cases are the transmission of packet in a single fragment or in multiple fragments, the third is an unsuccessful transmission requiring one or several retries.

There are three parameters defining how the Tx FIFO is filled during a payload transmission:

- **tx_threshold**, defines the available entry number in the Tx FIFO to trigger
4.3 Reduced DMA

a new Tx DMA operation.

- $tx_{max\_burst\_size}$, defines the maximum size (word number) of the burst on the MLB Bus

- $tx_{gap\_delay}$, defines the Tx DMA minimum space between 2 accesses on the MLB Bus, in clock period number.

While the two last conditions are directly handled by the address generator, the third condition, namely the availability of space on the FIFO, is treated by the DMA SM block. The signal $tx_{mic\_fifo\_size}$ contains the overall size of the FIFO. The signal $MAC_{txu\_frag\_size}$ finally contains the user defined size of one fragment.

![Figure 10: DMA internal signals](image)

**Single Fragment Transmission** If the size of the packet is smaller than the size of one fragment, $MAC_{txu\_frag\_size}$, the entire packet can be sent in one fragment. That this is the case cannot be detected from the beginning by the DMA since the next segment parameters of one segment only contain the size of the next segment and not of the overall packet. The internal structure of this case is therefore discussed together with the multiple fragment transmission case.

**Multiple Fragment Transmission** The 802.11 protocol allows transmission of a packet in multiple fragments. The idea is to divide the packet in multiple smaller blocks that can be sent and acknowledged individually reducing the risk that a fragment is corrupted. The transmission of each single fragment is acknowledged by the receiving station, which makes the overall transmission somewhat slower. In practice, an adapted fragment size is chosen to obtain optimum speed. This fragment size is of course dependent on the quality of the transmission as well as on the number of stations in the network that can cause possible interference.

As has been mentioned before, the packet manager commands the DMA to begin the transmission of the payload when the signal $PM_{begin\_payload}$ is
asserted. Once such a payload transmission is initiated, the DMA SM block does
the following (only the most important settings are detailed):

- Initialize fragment counter: Set \( \text{fragment\_count} \) to zero (the fragment
counter always keeps the number of the current fragment of the current
segment, beginning with a zero).

- Reset the fragment data counter: Set \( \text{rem\_fragdata\_count} \) to the maximum
fragment size given in \( \text{MAC\_txu\_frag\_size} \).

- Reset the segment data counter: Set \( \text{rem\_data\_count} \) to the size of the
current segment, given in the next segment parameters of each segment.

- Save the current fragment characteristics. In case that a retry will occur,
each time a new fragment is begun, the address, initial segment size and
information about whether the current is the last segment are stored in the
signals \( \text{cur\_frag\_ptr} \), \( \text{cur\_frag\_data\_count} \) and \( \text{cur\_frag\_lastseg} \), respectively.

After the initialization, the transmission of the payload data begins. During
the \( \text{addr\_gen\_calc} \) state of the SM, all major signals are updated. Using the
information about how much data is still remaining in the current segment, how
long the fragment can be and how much space is available on the transmit FIFO,
the signals for the address generator are formed. The important information
the address generator needs is the following:

- Address where to begin a burst read, indicated by \( \text{DMA\_addr} \).

- Number of bytes to be read, stored in \( \text{DMA\_number} \). Note that this num-
ber can be well larger than the maximum burst size, which is handled by
the address generator itself. The number is mainly limited by the available
space in the FIFO.

- Mode of the MLB access: The three signals \( \text{dma\_data\_rd} \), \( \text{dma\_ctrl\_rd} \) and
\( \text{DMA\_wr} \) indicate whether the MLB bus needs to be read and the data
sent to the FIFO, whether the read is control information that does not
need to be written to the FIFO or if status information needs to be written
onto the MLB bus, respectively.

Using these signals, the address generator can handle the MLB accesses in-
dependently. Since the address generator is driving control signals of the FIFO
filler itself, it is the task of the DMA SM block to make sure the Tx FIFO is not
overflown by giving a read order of well calculated size to the address generator.
The transmission cycle of the DMA is therefore a succession of read instructions
to the address generator. After each such instruction, the remaining data of the
segment, of the fragment and the available space on the FIFO are considered to
initiate the next read instruction. Case distinctions allow the proper detection
of the end of a fragment and the end of a segment as becomes evident when
looking at the state machine transitions in figure 9. Once the entire packet has
been sent to the FIFO, the DMA enters the state \( \text{wait\_end\_msdu} \). The HMAC uses
three signals to indicate whether a transmission had been successful (an affirma-
tive acknowledge has been received from the addressed station): \( \text{MAC\_txm\_done}\),
\( \text{MAC\_txm\_frag\_ok} \) and \( \text{MAC\_txm\_msdu\_end} \). The meaning of all combinations
is shown in table 1.
4.3 Reduced DMA

<table>
<thead>
<tr>
<th>done</th>
<th>frag_ok</th>
<th>msdu_end</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>φ</td>
<td>φ</td>
<td>Payload transmission</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Fragment end with error</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Fragment end without error</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Msdu end</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Reserved, never generated by HMAC</td>
</tr>
</tbody>
</table>

Table 1: Transmission status according to signals MAC_txm_done, MAC_txm_frag_ok and MAC_txm_msdu_end

If the entire MAC Service Data Unit (MSDU) has been sent successfully, the DMA jumps back to the idle state. If the sent fragment was successfully transmitted, but not the last one, the current fragment number is increased and the filling of the transmit FIFO is continued. If however there had something gone wrong and a packet could not be transmitted properly (Corruption of the packet, receiver not responding etc.), the 802.11 protocol asks for a retry of the transmission. The implemented retry functionality will be discussed next.

Transmission Requiring Retry The packet manager indicates that a packet can be tried to be resent if a next PM_start_payload pulse follows after the failed transmission. The packet manager also manages the maximum number of retries. If this number is exceeded, the retry routine is aborted. Since the initial settings of each fragment are stored locally in the DMA, the retries are easily done by just restoring the values of the fragment counter, remaining data counter and so on accordingly. The following signals are restored:

- `dma_address` jumps to the address stored in `cur_frag_ptr`
- `rem_data_count` obtains its value from `cur_frag_data_count`
- `lastseg` obtains its value from `cur_frag_lastseg`
- `fragment_count` is reduced by one in case of a retry

In this manner, a retry can be handled entirely individually from other blocks by the DMA. There are some issues related to the alignment of the data in the case of retries that will be addressed next.

Alignment Issues As was described above, the complexity of the DMA could be heavily reduced by the introduction of constraints on the data alignment. However, there are still some issues that result from the fact that the data is not aligned on 4 byte blocks that can be directly accessed on the MLB bus, but merely on 2 byte blocks only.

The DMA handles the alignment question without using the alignment bits of the old implementation. Several signals are used to manage the proper transmission of data onto the FIFO:

- `frags_unaligned` indicates whether the fragment size is a multiple of four (frags_unaligned set to false) or only a multiple of two (frags_unaligned set to true).
The DMA calculates the number of 4 byte blocks to put on the FIFO by taking $MAC_{txu\_frag\_size}$ modulo four. Every second fragment then adds another 4 byte block to this basis length to be conformous with the way the HMAC reads in the data. The boolean signal $address\_addon$ indicates whether this additional block needs to be included or not.

The HMAC keeps the last byte of a fragment on the Tx FIFO if the current fragment ends unaligned, meaning that the two first bytes of the next fragment are already written on the FIFO. This working principle simplifies the implementation on the DMA side as there is no need to rewrite any 4 byte block onto the FIFO.

A problem however arises when a fragment could not be transmitted successfully. The HMAC now needs the very first byte of the fragment to be rewritten onto the FIFO. Therefore, the two signals used to store the retry conditions of the present fragment transmission, $cur\_frag\_ptr$ and $cur\_frag\_data\_count$, do not contain the actual initial values of the present fragment if this one is unaligned. While if there is no transmission error, no 4 byte block needs to be written twice onto the FIFO, this becomes a necessity for unaligned fragment retries.

This case is currently treated by decreasing the address $cur\_frag\_data\_count$ by four and increasing the counter $cur\_frag\_ptr$ by one. This works perfectly fine if the specifications are satisfied and the payload consists of a single segment.

One goal of the reduced DMA was to completely omit the four alignment bits. Currently, the DMA SM block does not affect the bits anymore. However it turned out that the HMAC unexpectedly still needs those bits for proper functioning. Therefore, the whole structure including the four alignment bits has been maintained. Once the HMAC dependence will have been resolved, the alignment bits can be quickly removed from the MIC architecture. There will be additional area gains as the structure of the FIFO filler can be considerably simplified, as discussed in section 6.

### 4.3.3 Results

The results of the DMA modelling can be divided into two parts, the first being simulation and the second being synthesis results.

#### Simulation

The group at ST working on the Micmac project has developed a test environment in SystemC which allows to simulate the developed model in a cycle accurate manner. The test system allows to design various test cases that can then be analyzed using the Micmac as the unit under test. The test-benches allow to write different settings for the MIC and HMAC registers, allow to mimick a host’s intention and even allows to work with several different instances of the Micmac that are communicating among each other. The whole test system is therefore fairly complex and won’t be discussed in detail here. The important thing to retain is however that the system allows to severely test the working behaviour of the developed block. In particular, the correct transmission of packets using fragmented transmission and also corrupted data and retried fragment transmission can be simulated.

The developed reduced DMA was constantly tested using this testbench architecture. In fact, the development of the functionality was enhanced step by step and after each step the functionality was validated using adapted test
cases. In the end, the proper functioning of the whole transmission functionality could be shown if the restrictions of the new DMA specifications are fulfilled. Extended tests were run using worst case scenarios with multiple non-aligned fragment transmission involving retries. The proposed final solution of the reduced DMA has passed all the test cases. For the test cases done, the reduced DMA is behaviourally equivalent with the old full DMA, however due to changes on the DMA SM, the two are not equivalent on a clock cycle scale, which was however not needed. An examples of a simulated testcase is shown in figure 11. More sample simulation results are put into the appendix A of this document.

**Synthesis** The new, reduced size DMA was synthesized to obtain an estimate of the area gains that could be obtained. To do so, two different tools were used. The first one is a tool by Celoxica, which compiles directly from SystemC to an Electronic Design Interchange Format (EDIF) description. The second tool is DC FPGA by Synopsys, which was also used for the second part of this semester project, the adaptation of the synthesis flow.

The Celoxica tool Agility Compiler has some specific SystemC coding rules which made it necessary to adapt the code for this synthesis tool. While DC FPGA allows output ports to be read in the same entity, this is not possible with Celoxica and intermediate signals thus needed to be inserted. Additionally, there need to be default cases and initial values for all signals dealt with in a switch statement. This coding style produces warnings if synthesized with DC FPGA that default branches cannot be reached. The current code is maintained as Celoxica-synthesizable, but can be easily modified back to code not producing any warnings in DC FPGA.

The results of the synthesis is summarized in tables 2 and 3. It can be clearly seen that the reduced DMA actually needs less FFs and Look-Up Tables (LUTs) tables than the original one. The new design reduces FF needs to 55% and LUT needs to 35% if only the DMA SM is considered in the case of DC FPGA. If the entire DMA SM is considered, these numbers are still 66% for FFs and 41% for LUTs. The percentual gains are about the same for Celoxica and DC FPGA tools.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Count</th>
<th>Old DMA SM</th>
<th>New DMA SM</th>
<th>New/Old (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Celoxica</td>
<td>FF</td>
<td>1811</td>
<td>841</td>
<td>46%</td>
</tr>
<tr>
<td>Celoxica</td>
<td>LUT</td>
<td>8940</td>
<td>3160</td>
<td>35%</td>
</tr>
<tr>
<td>DC FPGA</td>
<td>FF</td>
<td>522</td>
<td>289</td>
<td>55%</td>
</tr>
<tr>
<td>DC FPGA</td>
<td>LUT</td>
<td>1898</td>
<td>668</td>
<td>35%</td>
</tr>
</tbody>
</table>

*Table 2: Comparison of the new (reduced) to the old (complete) DMA SM after synthesis using Celoxica and DC FPGA in terms of FF and LUT counts.*

The two tools of Celoxica and Synopsys were used with the same technology mapping. Therefore, it is surprising to see the substantial difference in size the two tools need to implement the exactly same design. One possible reason might be that Agility Compiler is mainly designed for behavioural synthesis. Finding the exact reasons for this difference must however be postponed to a subsequent study involving both simple and complex designs to quantify the
### 4.3 Reduced DMA

<table>
<thead>
<tr>
<th>Tool</th>
<th>Count</th>
<th>Old DMA</th>
<th>New DMA</th>
<th>New/Old (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Celoxica</td>
<td>FF</td>
<td>2604</td>
<td>1592</td>
<td>61%</td>
</tr>
<tr>
<td>Celoxica</td>
<td>LUT</td>
<td>11240</td>
<td>5133</td>
<td>46%</td>
</tr>
<tr>
<td>DC FPGA</td>
<td>FF</td>
<td>778</td>
<td>511</td>
<td>66%</td>
</tr>
<tr>
<td>DC FPGA</td>
<td>LUT</td>
<td>2340</td>
<td>966</td>
<td>41%</td>
</tr>
</tbody>
</table>

**Table 3:** Comparison of the new (reduced) to the old (complete) DMA block (containing DMA statemachine, address generator and FIFO filler) after synthesis using Celoxica and DC FPGA in terms of FF and LUT count.

**Figure 11:** (Next page) Visualized simulation results using DINOTRACE. The signal names are shown on the left side. If space permits, the values of buses are included. The time scale on the top is in nanoseconds. The fourth fragment is corrupted and is resent. The fragment size is not aligned on four bytes, but on two only. The transmission of the entire packet is finally successful and the status is written. The meaning of the signals is explained in table 4 in the appendix A.

advantages of one tool over the other.
4.3 Reduced DMA

4 DMA IMPLEMENTATION

<table>
<thead>
<tr>
<th>pmd_wm_data[7:0]</th>
<th>mac1-&gt;mic1-&gt;MLB_rwN</th>
<th>mac1-&gt;mic1-&gt;MLB_addr[29:0]</th>
<th>mac1-&gt;mic1-&gt;MLB_idata[31:0]</th>
<th>mac1-&gt;mic1-&gt;MLB_odata[31:0]</th>
<th>mac1-&gt;mic1-&gt;MLB_drdyN</th>
</tr>
</thead>
</table>
5 Micmac Synthesis Flow

5.1 Introduction

This chapter gives a summary of how the synthesis and implementation of the Micmac is realized. It is shown how to obtain from RTL models written in SystemC the final bitfile that can be downloaded on the FPGA. The design flow is based on the current Micmac distribution and is linked to its file structure. This document is based on the first version of the design flow as described in [26]. The major changes include the adaptation to the new Micmac file structure as well as the change from the fpga COMPILER II tool to the next generation tool named DESIGN COMPILER FPGA. The implementation flow has not changed from the preceding document and can be used as it is.

5.1.1 Design Flow

An overview over the design flow is given in figure 12. The initial models are written in SystemC on a RT level. The first part of the design flow consists in a synthesis towards an EDIF netlist. These netlists are then passed to the Place & Route (P&R) tool and the final bit file is generated in the implementation flow, which can then be downloaded onto the FPGA.

![Figure 12: Design Flow Overview](image)

5.1.2 File Structure

As opposed to the Microelectronic Systems Laboratory (LSM) file structure used in [26], the current design flow is again based on the original distribution by ST Geneva. The file structure is detailed in figure 13. The root directory contains folders with the SystemC code for the major building blocks. Everything related to synthesis is contained in the synth folder. It follows a description of the content of all folders.
5.1 Introduction

**MICMAC SYNTHESIS FLOW**

![Diagram of MICMAC project directory structure]

**Figure 13:** Micmac project directory structure

- **Micmac** Root directory of the Micmac project. On this hierarchical top level, the script `project_setup.source` can be found, which contains important environmental variables (for now, only environmental variables concerning synthesis are included). It is important that the `MICMAC_ROOT` variable be set to the location of the root directory. In such a way, the synthesis scripts can run on several systems without referencing problems.

- **MIC** Directory containing the MIC source code, written in SystemC

- **HMAC** Directory containing the HMAC source code, written in SystemC

- **BBIF** Directory that contains the BBIF code, available directly in Verilog

- **IB2F.32** Directory with the PCI interface models, written in Verilog

- **sim** The sim directory contains all files related to the simulation of the design, including test cases, trace file definitions and the simulation outputs.

- **software** The software directory hosts all driver modules that will ultimately fulfill the communication between the host CPU and the fully implemented Micmac.

- **synth** This is the root directory for synthesis. It contains the `.synopsys_dc.setup` scripts which sets the paths to the SYNOPSYS libraries.

- **hdl_src** All HDL source files are located in the `hdl_src` directory. There are subdirectories for the MIC, HMAC, BBIF and PCI (`ib2f.32`), containing the VERILOG source files. The `hdl_src` directory also contains VHDL descriptions of the two modules `dcm22.vhd` and `dcm44.vhd` (clock dividers), the top level model `top_avnet`, also written in VHDL, and the `rbi_if.v` VERILOG model describing the interface to the RBI bus.

- **edif** Contains both EDIF sources that are needed for the top-level synthesis and the output EDIF files resulting from the subblock synthesis. The files `icon0.edn` and `Ila.edn` are also located here.
5.2 Synthesis Flow

5.2.1 Overview

The overall architecture of the Micmac is a complex collection of interdependent modules, written in different hardware description languages (SystemC, Verilog, VHDL). The main architecture is visualized in figure 14.

The top level module of the current Micmac edition is the top_avnet.vhd module, which instantiates all other subblocks. The main subblocks are the HMAC, the MIC and the IB2F (PCI) module. The Micmac toplevel architecture is situated in the micmac.v module, present as VERILOG source. The dcm22 and dcm44 modules are Digital Clock Managers, generated by XILINX’ Architecture.
5.2 Synthesis Flow

Wizard, written in VHDL. The rbi_if.v is an interface between the PCI and the MIC module. For further details about the building blocks, refer to the Micmac documentation by ST., [17].

The synthesis flow of the Micmac consists of synthesizing all the main sub-blocks separately, namely the MIC, the HMAC and the IB2F. Once all these modules have been compiled into EDIF netlists, the top level design can be synthesized, which reads in all the aforementioned sub blocks, as well as the additional EDIF blocks missing so far, Ha and Icon0. The RBI interface (rbi_if.v) and micmac.v are compiled together with the top level model, top_avnet.vhd.

The synthesis flow from RTL models to a netlist description using EDIF and Netlist Constraint Files (NCFs) is shown in figure 15. While the VHDL and Verilog RTL models can directly be synthesized using DC FPGA, the SystemC models need to be translated. The translation is done by a dc_shell script and results in a Verilog RTL description.

5.2.2 Translation from SystemC to Verilog

The synthesis tool, DC FPGA, does not support direct compilation of SystemC models. Therefore, all models written using this HDL need to be translated to either VERILOG or VHDL which are both supported by DC FPGA. The translation is achieved by a Design Compiler script. The translation has not been modified since the last version of this document. The reader is referred to the reference [26] for an extensive discussion of the translation step from SystemC to Verilog. It is to be noted that the tool is not available at EPFL and the translation step needs to be done at ST.
5.2 Synthesis Flow

5.2.3 DC FPGA Synthesis Walkthrough

In the previous version of the synthesis flow, FPGA Compiler II was used for synthesis. Although being the successor of this tool, FPGA Compiler introduced some noticeable changes which made it necessary to rewrite the synthesis scripts. However, the file structure and guiding idea of the scripts used has been widely adapted from the previous FPGA Compiler scripts. An overview of the synthesis flow using DC FPGA is shown in figure 16. For details about DC FPGA, consult [13] and [12].

The synthesis begins with setting all environmental variables according to the present location / organization of the project. Make sure that the global reference to the Micmac root directory in the project_setup.source file is set to the present root directory location. To adjust the environmental variables, go to the Micmac root directory and type

```
user-Micmac> source project_setup.source
```

Next, change to the scripts directory

```
user-Micmac> cd synth/scripts
```
All synthesis scripts are sampled here. There are several ways of running a script. Either you can run the `main_synthesis.csh` script with one of the following arguments: `-mic`, `-hmac`, `-pci`, `-top`, `-all`

```bash
user−scripts> main_synthesis.csh −mic
```

While the first three options synthesize the indicated module individually, the `-top` attribute needs all those three modules having been compiled before. The `-all` option will synthesize all blocks in succession. Once finished, DC FPGA puts the report files containing information about area, timing, resource usage and so on into a newly created report directory. To access it, type

```bash
user−scripts> cd ../LOG/LOG_yy_mm_dd hh_mm_ss
```

where the directory naming is using a notation incorporating the date when the script was run in a format `year_month_day_hour_minute_second`. The reports can be read using any text editor.

```bash
user−LOG_date> emacs mic_timing.rpt
```

The next section will discuss the synthesis procedure and the working principle of the synthesis scripts in more details.

### 5.2.4 Synthesis Scripts

All synthesis scripts used for the synthesis of the entire Micmac are shown in figure 17. The script `main_synthesis.csh`, which was already discussed, calls either of the four subscripts `mic_syn.csh`, `hmac_syn.csh`, `pci_syn.csh` or `top_syn.csh`.

![Figure 17: Overview of the synthesis scripts](image)

These four main synthesis scripts begin with setting up the environment for a new compilation, i.e. the script cleans old work directories and creates a new log file directory. As an example, the script for the MIC is shown in listing 1.

**Listing 1: mic_syn.csh - root script for MIC synthesis**

```bash
1 #!/bin/csh −f
3 # create the report directory if not an overall compilation
5 if (OVERALL_COMPILATION == 0) then
```
5.2 Synthesis Flow

```bash
set LOG_DIR=$SYN_LOG/LOG
mkdir -p $LOG_DIR
echo "Create \ac{MIC} report directory"

# remove test directory — later to be changed for the dc_fpga directory
/bin/rm -rf $SYN_DCFPGA_WORK/mic

# remove old analysis file, will be immediately replaced by next line
/bin/rm $SYN_SCRIPTS/mic_sources.tcl

# create the analysis file containing all MIC verilog sources
$SYN_SCRIPTS/mic_create_analysis_file.csh

# launch the DC FPGA shell in command line. Stdout and Stderr will be stored in the current log directory.
/fpga_shell -t -f $SYN_SCRIPTS/mic_syn_dcfpga.tcl > $LOG_DIR/mic_dcfpga.log

# move reports
mv $SYN_LOG/mic_fpga.rpt $LOG_DIR/mic_fpga.rpt
mv $SYN_LOG/mic_timing.rpt $LOG_DIR/mic_timing.rpt
mv $SYN_LOG/mic_res.rpt $LOG_DIR/mic_res.rpt
mv $SYN_LOG/mic_ref.rpt $LOG_DIR/mic_ref.rpt
```

The .csh script creating the analysis file is run next. This script, shown in listing 2, searches the corresponding hdl_src directory and looks for files ending with .v, corresponding to Verilog RTL models. This way, the included models for synthesis are automatically updated and if a new instance is added to the model, the script detects and adds it automatically. Once done with setting up the environment for synthesis, DC FPGA is started in shell mode with the appropriate command file as an argument. The MIC_syn_dcfpga.tcl, shown in listing 3 file is a Tool Command Language (Tcl) script that is directly interpreted by the DC FPGA shell. First, the UNIX environmental variables are gathered and stocked into Tcl variables.

**Listing 2:** *mic_create_analysis_file.csh* - script creating a list of files for read-in to DC FPGA

```bash
#!/bin/csh -f

# looking for the Verilog files of the MIC, put list of all verilog files in file LOG1.txt
find $MIC_HDL_SRC | grep '.*\.v$' > LOG1.txt

# give all rights to LOG1.txt
chmod 777 LOG1.txt

# create a list of analyze commands and write it in the file \ac{MIC} syn_dcfpga.tcl
```
Finally, all files previously assembled by the mic_create_addfile.csh script are now read in using the analyze command. To do this, the created mic_addfile.tcl script is run from within the DC FPGA shell. What follows are specific steps required by the synthesis tool, such as elaborating the design, setting the current design to the top level model, setting the clocks and compilation options. The ungroup command allows a flattening of the design.

Once all options are entered, the design can finally be compiled. The compilation results are stored in form of EDIF netlists and .db files to the indicated folders, the simulation reports are first put into the Log directory (LOG) root directory. At this point, the DC FPGA shell is quit and the remaining task of the mic_syn.csh script is executed, which consists of moving all report files into the current log directory. This moving step has been maintained because it allows to easily change the mv command to a cp command, which will leave the latest reports easily accessible at the LOG root.

The synthesis flow is then finished for the discussed modules. As for the top_syn.csh script, not only RTL models are included, but also EDIFs from the aforementioned subblocks are added, and IP blocks such as IIa or Icon0 are assembled in this ultimate compilation step. The EDIF files can be read into DC FPGA using the read_edif command. The final compilation is then run equivalently to the other constitutive blocks and the resulting overall reports deposited similarly in the appropriate LOG directory. The synthesis flow of the Micmac design is then finished and we pass to the next design step, which is the implementation flow.

### Listing 3: mic_syn_dcfpga.tcl - DC FPGA script for MIC synthesis

```
synroot [get_unix_variable SYNROOT]
syn_db [get_unix_variable SYNDB]
syn_edif [get_unix_variable SYNEDIF]
syn_log [get_unix_variable SYNLOG]
syn_scripts [get_unix_variable SYNSCRIPTS]
device 2V4000FF1152

# Setup the tool. The synopsys setup file needs to be adapted to
# the FPGA in use. See ‘synlibs’ command
source $SYNROOT/./synopsys.dc.setup

# NOTE: to do: get the output of the source into the log file.
# -echo is not working.

# Define the WORK directory
```
5.2 Synthesis Flow

define_design.lib WORK -path $SYNROOT/dfpfa/mic

# Pragmas in code
set hdlin_translate_off_skip_text true

# Very important: Will perform basic variables setting depending on
# the FPGA targeted
set fpga_defaults xilinx_virtex2 -verbose

# Enhance LUT compacting (area driven)
set fpga_patch_luts true

# analyze RTL files. The script mic_sources.tcl contains a list of
# the analyze commands. It can be created using
# mic_create_analysis_file.csh
source $SYNSCRIPTS/mic_sources.tcl

# elaborate the analyzed source files. Only the toplevel instance
# needs to be called explicitly, without any extension.
elaborate mic

write -hier -f db -o $SYN_DB/mic.elab.db

link

current_design mic

# Set the FPGA device to be used. A list of all available FPGAs can
# be obtained using the command 'set_fpga_target_device
# -show_all'. This command needs to be typed into the DC FPGA
# shell after the libraries had been properly set (see also
# synlibs, .synopsys_dc.setup)
set_fpga_target_device $DEVICE

# Create Clock, false path, multicycle path etc.
create_clock [get_port host_clk] -period 30
create_clock [get_port MAC_clk] -period 45
create_clock [get_port mem_clk] -period 30
set_false_path -from [get_clock host_clk] -to [get_clock MAC_clk]
set_false_path -from [get_clock MAC_clk] -to [get_clock host_clk]
set_false_path -from [get_clock host_clk] -to [get_clock mem_clk]
set_false_path -from [get_clock mem_clk] -to [get_clock host_clk]
set_false_path -from [get_clock mem_clk] -to [get_clock MAC_clk]
set_false_path -from [get_clock MAC_clk] -to [get_clock mem_clk]
current_design mic

ungroup -all -flatten

# Compile the MIC design
compile -boundary_optimization

write -hier -f edif -o $SYN_EDIF/mic.edf
write -hier -f db -o $SYN_DB/mic.db
5.3 Implementation Flow

The implementation flow consists of the generation of bit files that can be readily downloaded to the FPGA. The input to the implementation flow is, as shown in figure 12, the EDIF description of the model. Additionally, a NCF is needed by the tool, which was also generated by DC FPGA. Since the exact implementation flow has basically not changed since the previous version of the synthesis flow document [26], its content will not be repeated here. All steps involving an adaptation to a XILINX specific logic database Native Generic Database (NGD), a FPGA mapping and finally the P&R are also discussed in detail in [26].
6 Probing Further

The design of the Micmac is a complex project involving different groups and many different people have been working on it since the beginning. The implementation of the reduced DMA and the adaptation of the design flow have only been two elements in this global project. I will still name some future realizations that can be done to enhance the product as it is now.

The first enhancement was already mentioned before: The HMAC is still dependent on the four alignment bits. The size of the DMA and of some buses can be further reduced once those four alignment bits are completely eliminated. A version of the transmit DMA FIFO filler is included in appendix B on page 70 to show the further possible reductions. Most of the current code could be commented out as it is not needed anymore once the alignment bits are gone. An estimated area reduction to about 30% of the original FIFO filler block is viable.

The next steps of the project would certainly be to test the whole Micmac design with an FPGA demonstrator and build an application showing the proper working of the design. The group at EPFL is currently working towards such a demonstrator. Of course, it is not sufficient to validate the models only using simulation, the Micmac needs to be thoroughly tested in a real life setup demanding the wireless communication in a standard IEEE 802.11 WiFi network.

It was mentioned that the design was intended to be used as an Intellectual property block for customer’s SoCs. Additional adaptations such as ASIC technology mappings, working with different system buses and analog front-ends etc. will need to be developed and tested. Finally, the design needs to be proven to be superior to existing commercial designs using other implementations in terms of area needs, speed and power consumption. The introduction of the IP product Micmac will only be successful if it actually shows an advantage at this stage.
7 Conclusion

In the first part of this semester project, I re-designed the direct memory access block of an IEEE 802.11 medium access controller according to adapted specifications, aiming at a reduced area need. The simulation results showed the proper functioning of the developed model and synthesis proved the reduced size of the design over the complete DMA block.

In the second part, I adapted an existing synthesis flow to a new synthesis tool and a changed project hierarchy and architecture. The transition has successfully been done and the new synthesis flow is now readily available for the Micmac project.

I was happy to work during this semester on a project that has an industrial background and which involved different laboratories. It was extremely interesting to see how such a large project is managed and how tasks are divided between the different groups. It is impressive how the various hardware models written in VHDL, VERILOG and SYSTEMC work together, how they can be simulated and how they interact with an entire driver system implemented on LINUX to work together with the FPGA demonstrator. It was very fruitful to see the hierarchy and interfacing between modules, the problems arising and necessity of proper specifications.

I definitely learned a lot during this semester project. It was not easy to dive into this project. Hundreds of signal names and header files made it a hard thing to get the overall view needed to begin constructive work. A wall of my room served as a large drawboard for connecting inputs to outputs and sketch the block dependences. Once the initial hurdles were taken, the work was still challenging, but intermediate results kept me motivated to tackle the next problem and the final results are very satisfactory for me. The second part of the project allowed me to gain even more insight of the structure of the whole Micmac project as I needed to synthesize the entire architecture. Again, this was very interesting and fruitful. To conclude, I think the main thing I’ve learned in this project is how a large development project is organized and managed in different blocks, how responsibilities and specifications are defined and how finally all the blocks are put together to a working whole.
8 Acknowledgements

I would like to thank my assistants, Serge Lopez and Olivier Croset for their assistance. They were always available for help and motivating words. They were competent persons to contact when things did not want to work and would most probably come up quickly with a solution.

I would also like to thank Dr. Alain Vachoux. He helped me to set up the computer account and the DC FPGA environment for synthesis. I very much appreciate his experience in the field of hardware modelling.

A word of thank also goes to Pascal Delamotte of STMicroelectronics. During his stay at EPFL he was very helpful and happy to explain the particularities of the MIC structure.

Finally, I would like to thank Prof. Yusuf Leblebici for letting me work in his laboratory during this semester project. I appreciated it very much.

Lausanne, 20.6.2006

Christoph Eggimann
A Simulation Results Trace Files

Some sample simulation results using Dinotrace to draw the generated traces from the simulations are shown in this appendix. The views show the overall behaviour of the developed reduced DMA block as well as some particularities in the zoomed views. For all the following figures, the signal names are shown on the left side. The arrows show the hierarchy of the instances the signal is taken from. Table 4 shows the meaning of each of the signals. The time ticks on top of the graphs represent nanoseconds and also the cursor distances shown on the bottom of the views are in nanoseconds. Please note that the time scales are not always the same from one view to another. To get a profound understanding of the working principle, however, the signals need to be investigated on a computer, having the possibility to zoom in and return particular signal values.
### Table 4: Meaning of the signals shown in the tracefiles

<table>
<thead>
<tr>
<th>Signal</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>pm__data</td>
<td>Signal currently in the air</td>
</tr>
<tr>
<td>MLB__writeN</td>
<td>Read/NotWrite access to the MLB bus</td>
</tr>
<tr>
<td>MLB__addr</td>
<td>Address lines of MLB bus</td>
</tr>
<tr>
<td>MLB__idata</td>
<td>Data coming in from the host on the MLB bus</td>
</tr>
<tr>
<td>MLB__odata</td>
<td>Data written to the host on the MLB bus</td>
</tr>
<tr>
<td>MLB__drtyN</td>
<td>Host signalling that data on MLB bus is available</td>
</tr>
<tr>
<td>dina</td>
<td>Signal showing the entrance to the Tx Static Random Access Memory (SRAM)</td>
</tr>
<tr>
<td>doutb</td>
<td>Signal showing the output of the Tx SRAM</td>
</tr>
<tr>
<td>pm0__state</td>
<td>State of the transmitting packet manager</td>
</tr>
<tr>
<td>dma__sm__state</td>
<td>State of the Tx DMA SM</td>
</tr>
<tr>
<td>addr__gen__state</td>
<td>State of the Tx address generator</td>
</tr>
<tr>
<td>addr__generator_end</td>
<td>Indicates the end of an address generator cycle</td>
</tr>
<tr>
<td>PM__start_header</td>
<td>Orders the Tx DMA to begin reading the header from the host and putting it onto the Tx FIFO</td>
</tr>
<tr>
<td>PM__start_payload</td>
<td>Orders the Tx DMA to begin sending the payload to the HMAC</td>
</tr>
<tr>
<td>PM__start_status</td>
<td>Orders the Tx DMA to write back the status information to the host</td>
</tr>
<tr>
<td>DMA__addr</td>
<td>Read address passed by the DMA to the address generator</td>
</tr>
<tr>
<td>DMA__number</td>
<td>Number of accesses on the MLB bus to be effectuated, passed by the DMA to the address generator</td>
</tr>
<tr>
<td>DMA__ctrl_rd</td>
<td>Indicates a control read to the address generator</td>
</tr>
<tr>
<td>DMA__data_rd</td>
<td>Indicates a data read to the address generator</td>
</tr>
<tr>
<td>dma__address</td>
<td>The internal address counter of the Tx DMA</td>
</tr>
<tr>
<td>filler__write_fifo</td>
<td>The data that is put by the FIFO filler onto the Tx FIFO</td>
</tr>
<tr>
<td>filler__write_fifo</td>
<td>Signal indicating that the filler data is actually passed to the FIFO</td>
</tr>
<tr>
<td>num__in_fifo_mic</td>
<td>Amount of data that is stored currently on the Tx FIFO</td>
</tr>
<tr>
<td>MAC__txm_msdu_end</td>
<td>First of three pulses indicating the transmission status of fragments/packets</td>
</tr>
<tr>
<td>MAC__txm_frag_ok</td>
<td>Second of three pulses indicating the transmission status of fragments/packets</td>
</tr>
<tr>
<td>MAC__txm_done</td>
<td>Last of three pulses indicating the transmission status of fragments/packets</td>
</tr>
<tr>
<td>cur__frag_ptr</td>
<td>Points to the beginning of the current frag in case a retry needs this address</td>
</tr>
<tr>
<td>cur__frag_data_count</td>
<td>Stores remaining number of bytes in segment for the beginning of each new fragment</td>
</tr>
<tr>
<td>rem__data_count</td>
<td>Remaining data to be sent of the current segment</td>
</tr>
<tr>
<td>rem__frag_data_count</td>
<td>Remaining data that can be put into the current fragment</td>
</tr>
<tr>
<td>address__addon</td>
<td>Indicates whether an additional 4 byte block is put on the FIFO for the current fragment</td>
</tr>
<tr>
<td>fragment__count</td>
<td>Counts the transmitted fragments</td>
</tr>
</tbody>
</table>
Figure 18: Simulation results: Test case fourth frag, retry o, overall view.
Figure 19: Simulation results: Test case fourth-frag-rty-4-header view.
Figure 20: Simulation results: Test case fourth frag retry, view of a part where the actual signal is transmitted in the air (pmd_wm_data signal).
A SIMULATION RESULTS TRACE FILES

Figure 21: Simulation results: Test case fourth-fragretry, view of a part where the retry is initiated.
Figure 22: Simulation results: Test case of non-aligned fragments and the first and third frags are corrupted.

250000 500000 750000 1000000 1250000 1500000 1750000 2000000 2250000 2500000 2750000 3000000
450330 2983230
2532900
B  Source Code of the adapted DMA

Listing 4: tx_dma_sm.h - Transmit DMA State Machine header file

```c
// *****************************************************
// Tx DMA SM header file
// *****************************************************

#define Transmit_DMA_SM

/* Name: Tx_dma_sm.h */
/* Date created: 18/09/03 */
/* Author: Julia Le Maitre */

/* Description: This module is controlled by one PM channel and 
communicates with the IMAC, the address generator for 
the memory accesses, the fifo filler for the access to 
the FIFO and the M&R data bus for the addresses read. */

/* Revision: 
0.1: initial 18/09/03 */
0.2: Pkt parameters are now part of the header 02/10/03
0.3: No fragment prefetch in the FIFO 08/10/03

04 01 06 - jgb - DMA_first_frag to fifo_filler
04 01 08 - jgb - frag_count_s1_top
04 01 19 - jgb - frag_begin/end_enable
04 01 20 - jgb - segfrag_begin/end_size
04 01 20b - jgb - newseg_frag_count, frag_begin_size
04 01 23 - jgb - newseg nxtfrag_end
04 01 26 - jgb - seg_ltrf
04 01 26b - jgb - next_frag_addr_2 sensitivity list
04 01 28 - jgb - seg_ltrf output to tx_fifo_filler
06 04 06 - ceg - Simplified DMA. Functionality without frag-
and Retry. Segments and Fragments must be aligned on 2bytes
05 05 06 - ceg - DMA completely rewritten, supports fragmentation
**********

#include "systemc.h"
#include "Tx_const.h"

1/ // MODULE(tx_dma_sm) { 
2/ sc_in<bool> mem_clk;
2/ sc_in<bool> resetN;
3/ // MIC_Registers block interface 
3/ sc_in<bool> MIC_Endianness; // NOT USED 
4/ sc_out<bool> MIC_txFrag_p; // NOT USED 
4/ // IMAC interface 
5/ sc_in<sc_uint<32>> MAC_txm_status; 
5/ sc_in<bool> MAC_txm_done; // End of fragment. can be ok or not 
5/ sc_in<bool> MAC_txm_nsdru_end; // Packet status available 
5/ sc_in<bool> MAC_txm_frag ok; // Fragment ok. Sending next fragment 
5/ sc_out<bool> MIC_dma_hd_rdy; // Header is in FIFO 
5/ // PM interface 
5/ sc_in<bool> PM_start_header; 
5/ sc_in<bool> PM_start_payload; 
5/ sc_in<bool> PM_start_status; 
5/ sc_out<bool> DMA_header ok; 
5/ sc_out<bool> DMA_frag ok; // NOT USED 
5/ sc_in<sc_uint<32>> PM_packetfifo_rd; 
5/ sc_in<sc_uint<32>> PM_packetpointer; // NOT USED 
5/ sc_in<sc_uint<32>> PM_fragseg_count; // NOT USED 
5/ sc_out<sc_uint<32>> DMA_packetpointer; 
```

53
B SOURCE CODE OF THE ADAPTED DMA

```c
sc_out<sc_uint<32>> DMA_fragment_pointer; // NOT USED
sc_out<sc_uint<32>> DMA_frag_seg_count;
// FIFO interface
sc_in<sc_uint<12>> num_in_fifo_mic;
sc_out<bool> DMA_flush;
// PCI interface
sc_in<bool> tx_MLB_drdyN;
sc_in<sc_uint<32>> tx_MLB_idata;
// Add generator and fifo filler interface
sc_in<bool> addr_generator_end;
sc_out<sc_uint<32>> DMA_addr;
sc_out<sc_uint<32>> DMA_pl_stat;
sc_out<sc_uint<10>> DMA_number;
sc_out<sc_uint<12>> DMA_seg_cnt; // NOT USED
sc_out<sc_uint<12>> DMA_frag_cnt; // NOT USED
sc_out<bool> DMA_wr;
sc_out<bool> DMA_ctl_rd;
sc_out<bool> DMA_data_rd;
sc_out<bool> DMA_data_mode;
sc_out<bool> DMA_init_seg; // NOT USED
sc_out<bool> DMA_frag_begin; // NOT USED
sc_out<bool> DMA_first_frag; // NOT USED
sc_out<bool> DMA_lid_begin;
sc_out<sc_uint<10>> DMA_size;
sc_out<sc_uint<12>> DMA_count_al; // NOT USED
sc_out<sc_uint<6>> DMA_count;
sc_out<sc_uint<6>> seg_begin_enable; // NOT USED
sc_out<sc_uint<6>> seg_mid_enable; // NOT USED
sc_out<sc_uint<6>> seg_end_enable; // NOT USED
sc_out<sc_uint<6>> frag_begin_enable; // NOT USED
sc_out<sc_uint<6>> frag_end_enable; // NOT USED
sc_out<bool> seg_itrfr; // NOT USED
// Reg block interface
sc_in<sc_uint<32>> tx_status_fifo_wr;
sc_in<sc_uint<16>> tx_mic_fifo_size;
sc_in<sc_uint<12>> tx_threshold;
sc_in<bool> tx_retry; // NOT USED

// DMA states
comm dma_states {
  idle, // 0
  rd_pkt_ptr, // 1
  rd_header, // 2
  rd_next_seg_params, // 3
  wait_fetch_hd, // 4
  addr_gen_calc, // 5 - ceg 6.4.06 renamed
  wait_rd_completed, // 6
  init_next_ptr_params, // 7 - ceg 6.4.06 added
  rd_next_ptr_params, // 8
  wait_fifo, // 9
  wait_end_msd, // A
  wr_transaction // B
};

//sc_signal<dma_states> state;
sc_signal<sc_uint<4>> > state;
// Communication between processes
sc_signal<sc_uint<4>> > fragment_count;
sc_signal<sc_uint<4>> > segment_addr;
sc_signal<sc_uint<12>> > MAC_txx_frag_size_al;
sc_signal<sc_uint<12>> > dma_address;
sc_signal<sc_uint<12>> > rem_data_count;
sc_signal<sc_uint<12>> > rem_fragdata_count;
sc_signal<sc_uint<12>> > cur_frag_ptr;
sc_signal<sc_uint<12>> > cur_frag_data_count;
sc_signal<bool> cur_frag_lastseg;
sc_signal<bool> frags_unaligned;
sc_signal<bool> lastseg;
sc_signal<bool> address_addon;
```

54
B SOURCE CODE OF THE ADAPTED DMA

sc_signal<bool> delay1_tx_MLB_drtyN;

// Processes
void nextstate();
void addr_gen_start();
void DMA_packet_pointer_gen();
void DMA_hd_begin_gen();
void MIC_header_rdy_gen();
void data_mode_gen();
void DMA_frag_params_gen();
void DMA_header_ok_gen();
void MLB_drty_gen();
void lastseg_gen();
void retry_parameters_gen();
void DMA_flush_gen();
void enable_bits_gen();

// Counters
void address_counter();
void rem_data_counter();
void rem_fragdata_counter();
void frag_counter();

// Combinational Signals Generation
void frags_unaligned_gen();
void address_addon_gen();

SCCTOR(tx_dma_sm) {
    SCMETHOD(nextstate);
    sensitive_pos << mem_clk;
    sensitive_neg << resetN;
    SCMETHOD(addr_gen_start);
    sensitive_pos << mem_clk;
    sensitive_neg << resetN;
    SCMETHOD(DMA_packet_pointer_gen);
    sensitive_pos << mem_clk;
    sensitive_neg << resetN;
    SCMETHOD(DMA_hd_begin_gen);
    sensitive_pos << mem_clk;
    sensitive_neg << resetN;
    SCMETHOD(MIC_header_rdy_gen);
    sensitive_pos << mem_clk;
    sensitive_neg << resetN;
    SCMETHOD(data_mode_gen);
    sensitive_pos << mem_clk;
    sensitive_neg << resetN;
    SCMETHOD(DMA_frag_params_gen);
    sensitive_pos << mem_clk;
    sensitive_neg << resetN;
    SCMETHOD(DMA_header_ok_gen);
    sensitive_pos << mem_clk;
    sensitive_neg << resetN;
    SCMETHOD(MLB_drty_gen);
    sensitive_pos << mem_clk;
    sensitive_neg << resetN;
    SCMETHOD(lastseg_gen);
    sensitive_pos << mem_clk;
    sensitive_neg << resetN;
    SCMETHOD(retry_parameters_gen);
    sensitive_pos << mem_clk;
    sensitive_neg << resetN;
Listing 5: tx_dma_sm.cc - Transmit DMA State Machine main file

```cpp
SCMETHOD(DMA_flush_gen);
sensitive_pos << mem_clk;
sensitive_neg << resetN;

SCMETHOD(enable_bits_gen);
sensitive_pos << mem_clk;

SCMETHOD(address_counter);
sensitive_pos << mem_clk;
sensitive_neg << resetN;

SCMETHOD(data_counter);
sensitive_pos << mem_clk;
sensitive_neg << resetN;

SCMETHOD(rem_fragdata_counter);
sensitive_pos << mem_clk;
sensitive_neg << resetN;

SCMETHOD(rem_data_counter);
sensitive_pos << mem_clk;
sensitive_neg << resetN;

SCMETHOD(frag_counter);
sensitive_pos << mem_clk;
sensitive_neg << resetN;

SCMETHOD(frag_unaligned_gen);
sensitive << MAC_txu_frag_size;

SCMETHOD(address_addon_gen);
sensitive <= fragment_count;
sensitive <= frags_unaligned;
```

---

**B  SOURCE CODE OF THE ADAPTED DMA**

```cpp
//**************************************************************************
*                      Tx DMA SM implementation                           *
**************************************************************************
* Name:  Tx_dma_sm.cpp                                                *
* Date created:  18/09/03                                              *
* Author: Julia Le Maitre                                             *
* Description: The DMA is in charge of launching the address geneator  *
* for the memory accesses. It can work in 3 modes:                    *
* - Header mode: transmit of the header                                *
* - Payload mode: transmit of the payload                              *
* - Status mode: Write status to host memory                          *
* He has to generate the enable bit to indicate to the                *
* serializer which bits are valid. He also has to flush                *
* the FIFO and the serializer in case of error.                       *
* Revision:                                                          *
* 0.1: initial 18/09/03                                              *
* 0.2: Pkt parameters are now part of the header 02/10/03             *
* 0.3: No fragment prefetch in the FIFO 08/10/03                      *
* WARNING: DMA_fragment_pointer line 456, supports only even fragment sizes, odd fragment sizes are forbidden
* 04 01 06 - jgb - DMA_first_frag to fifo_filler, frag_count,al
* 04 01 08 - jgb - frag_count,al_top
* 04 01 19 - jgb - frag_begin/end_enable
* 04 01 20 - jgb - seg/frag_begin/end_sz/size & concurrent end
* 04 01 21 - jgb - seg
* 04 01 23 - jgb - seg/frag
* 04 01 25 - jgb - newseg/frag_count, frag_begin_sz,old
* 04 01 26 - jgb - frag_count,al
* 04 01 27 - jgb - newseg/nextfrag_end
* 04 02 02 - jgb - assert DMA_first_frag when retry to avoid wrong
```
SOURCE CODE OF THE ADAPTED DMA

```c
#include "Tx_dma_sm.h"

// DMA state machine
// written signals:
// − state

void tx_dma_sm::nextstate() {
  if (!resetN.read()) {
    state.write(idle);
  } else {
    state.write(idle);
    switch (state.read()) {
      case idle:
        if (PM_start_header.read()) {
          state.write(rd_pkt_ptr);
          // DMA in header mode
        } else if (PM_start_payload.read()) {
          state.write(addr_gen_calc);
          // DMA in payload mode
        } else if (PM_start_status.read()) {
          state.write(addr_generator_end
          // DMA in status mode
        } else {
          // remain in idle state
          state.write(idle);
        } break;
      // HEADER MODE
      // State machine in header mode, started by PM with pulse
      // 'PM_start_header'
      //
      case rd_pkt_ptr:
        if (addr_generator_end.read()) {
          // address of the header ok, proceed to reading the header
          state.write(rd_header);
        } else {
          state.write(rd_pkt_ptr);
        } break;
      case rd_header:
        if (addr_generator_end.read()) {
          // header retrieved, obtain next segment parameters next
          state.write(rd_next_seg_params);
        } else {
          state.write(rd_header);
        } break;
      case rd_next_seg_params:
        if (addr_generator_end.read()) {
          // if finished reading the next segment parameters, wait until
          // header is read by HMA
          state.write(wait_fetch_hd);
        } else {
          state.write(rd_next_seg_params);
        }
    }
}
```
break;

case wait_fetch_hd:
    if (num_in_fifo_mic.read() == 0) {
        // if the fifo is emptied, return to idle state
        state.write(idle);
    } else {
        state.write(wait_fetch_hd);
    }
    break;

    // ______________________ PAYLOAD MODE ______________________
    // State machine in payload mode, started by PM with pulse
    // 'PM_start_payload'
    // ______________________

case addr_gen_calc:
    // This state dures only one clock cycle and is used to calculate
    // the parameters for the address generator (address, number,
    // data/control)
    state.write(wait_rd_completed);
    break;

case wait_rd_completed:
    if (addr_generator_end.read() & (rem_data_count.read() != 0) &
        (rem_fragdata_count.read() != 0)) {
        // there’s still data to be put on fifo and fragment not complete
        // yet, so wait till fifo emptied below threshold by HMAC
        state.write(wait_fifo);
    } else if (addr_generator_end.read() & (rem_data_count.read() == 0) &
        !lastseg.read()) {
        // All data of current segment written, read next segment
        // parameters
        state.write(init_next_ptr_params);
    } else if (addr_generator_end.read() & (rem_data_count.read() == 0) &
        !lastseg.read()) {
        // Current fragment complete, wait for HMAC signal
        state.write(wait_end_msdu);
    } else if (addr_generator_end.read() & (rem_data_count.read() == 0) &
        lastseg.read()) {
        // All data of last segment written, go to wait end msdu state
        state.write(wait_end_msdu);
    } else {
        state.write(wait_rd_completed);
    }
    break;

case init_next_ptr_params:
    // Another state during only one pulse, initializing the settings
    // for the next parameter retrieval
    state.write(rd_next_ptr_params);
    break;

case rd_next_ptr_params:
    if (addr_generator_end.read()) {
        // Next packet parameters read, wait for the fifo level to be below
        // threshold next
        state.write(wait_fifo);
    } else {
        state.write(rd_next_ptr_params);
    }
    break;

case wait_fifo:
    if (num_in_fifo_mic.read() <= tx_threshold.read()) {
        // Fifo level below threshold, proceed calculating the next address
        // generator settings
        state.write(addr_gen_calc);
    }
```c
B SOURCE CODE OF THE ADAPTED DMA

void tx_dma_sm::addr_gen_start() {

    // Starts the address_generator each time it needs a memory access
    // sending him an address where he will read or write (DMA_addr), the
    // number of bytes it wants to read or write (DMA_number) and the operation
    // type:
    // − DMA_data_rd : the data read out of memory will be filled into FIFO
    // − DMA_ctrl_rd : the data read out of memory will be "traitee" by the DMA
    // − DMA_wr : write operation
    // In write operation mode, it sends the status, given by HMAC
    // Ceg: Not supported yet are DMA_seg_cnt and DMA_frag_cnt
    // written signals:
    // − DMA_addr
    // − DMA_number
    // − DMA_data_rd
    // − DMA_ctrl_rd
    // − DMA_wr
    // − DMA_pl_stat
```
if (!resetN.read()) {  
    DMA_addr.write(0);  
    DMA_number.write(0);  
    DMA_data_rd.write(false);  
    DMA_ctrl_rd.write(false);  
    DMA_wr.write(false);  
    DMA_pl_stat.write(0);  
}  
else {  
    DMA_addr.write(0);  
    DMA_number.write(0);  
    DMA_data_rd.write(false);  
    DMA_ctrl_rd.write(false);  
    DMA_wr.write(false);  
    DMA_pl_stat.write(0);  
}

switch (state.read()) {  
    // case idle  
    //  
    case idle:  
    if (PM_start_header.read()) {  
        // IDLE -> RD_PKT_PTR : memory access to the packet pointer  
        DMA_addr.write(PM_packet_fifo_rd.read());  
        DMA_number.write(1);  
        DMA_ctrl_rd.write(true);  
        // additional signals, added for synthesizability purposes  
        DMA_data_rd.write(0);  
        DMA_wr.write(0);  
        DMA_pl_stat.write(0);  
    }  
    else if (PM_start_status.read()) {  
        // IDLE -> WITRANSACTION : writing packet pointer and status to memory  
        DMA_addr.write(tx_status_fifo_wr.read());  
        DMA_pl_stat.write(MAC_txm_status.read());  
        DMA_wr.write(true);  
        // additional signals, added for synthesizability purposes  
        DMA_number.write(0);  
        DMA_data_rd.write(0);  
        DMA_ctrl_rd.write(0);  
    }  
    else {  
        // else clause, for synthesizability purpose  
        DMA_addr.write(DMA_addr.read());  
        DMA_number.write(DMA_number.read());  
        DMA_data_rd.write(false);  
        DMA_ctrl_rd.write(false);  
        DMA_wr.write(false);  
        DMA_pl_stat.write(0);  
    }  
    break;  
}  
// case rd_pkt_ptr  
//  
    case rd_pkt_ptr:  
    // RD_PKT_PTR -> RD_PKT_PARAMS : memory access to the packet header (params included)  
    if (addr_generator_end.read()) {  
        DMA_addr.write(DMA_packet_pointer.read());  
        DMA_number.write(9);  
        DMA_data_rd.write(true);  
        // additional signals, added for synthesizability purposes  
        DMA_ctrl_rd.write(false);  
        DMA_wr.write(false);  
        DMA_pl_stat.write(0);  
    }  
    else {  
        // else clause, for synthesizability purpose  
        DMA_addr.write(DMA_addr.read());  
        DMA_number.write(DMA_number.read());  
        DMA_data_rd.write(false);  
    }  
}
 DMA_ctrl_rd.write(false);
    DMA_wr.write(false);
    DMA_pl.stat.write(0);
}
break;

// case rd_header
case rd_header:
    // RD_HEADER -> RD_NEXT_SEG_PARAMS : memory access to the next
    // segment address, size and type (always two reads of type control)
    if (addr_generator.end.read()) {
        DMA_addr_write(dma_addr.read());
        DMA_number.write(2);
        DMA_ctrl_rd.write(true);
        // additional signals added for synthesizability purposes
        DMA_data_rd.write(false);
        DMA_wr.write(false);
        DMA_pl.stat.write(0);
    }
    else {
        // else clause: for synthesizability purpose
        DMA_addr_write(DMA_addr.read());
        DMA_number.write(DMA_number.read());
        DMA_data_rd.write(false);
        DMA_ctrl_rd.write(false);
        DMA_wr.write(false);
        DMA_pl.stat.write(0);
    }
break;

// case addr_gen_calc
case addr_gen_calc:
    // Calculates the input to the address_generator in the payload mode.
    // It is always a data read, but the size depends on the fifo size
    // (tx_mic fifo size), the amount of data already in the fifo
    // (num_in_fifo_mic), the size of the remaining data in the present
    // segment rem_data_counter. The start address of the read is given
    // by dma_addr, which is passed to the address generator using
    // DMA_add.
    if (rem_fragdata_count.read() <= (tx_mic_fifo_size.read() -
        num_in_fifo_mic.read()) & &
        rem_fragdata_count.read() <= rem_data_count.read()) {
        DMA_addr_write(dma_addr.read());
        DMA_number.write(rem_fragdata_count.read());
        DMA_data_rd.write(true);
        // additional signals added for synthesizability purposes
        DMA_ctrl_rd.write(false);
        DMA_wr.write(false);
        DMA_pl.stat.write(0);
    }
else if (rem_data_count.read() <= (tx_mic_fifo_size.read() -
    num_in_fifo_mic.read()) & &
    rem_data_count.read() < rem_fragdata_count.read()) {
    // all remaining bytes of the segment can directly be written.
    DMA_addr_write(dma_addr.read());
    DMA_number.write(rem_data_count.read());
    DMA_data_rd.write(true);
    // additional signals added for synthesizability purposes
    DMA_ctrl_rd.write(false);
    DMA_wr.write(false);
    DMA_pl.stat.write(0);
}
else {
    // Size of segment/fragment does not fit into the Fifo.
    // -> fill the Fifo
}
DMA_addr.write(dma_address.read());
DMA_number.write(tx_mic_fifo_size.read() - num_in_fifo_mic.read());
DMA_data_rd.write(true);
// additional signals, added for synthesizability purposes
DMA_ctrl_rd.write(false);
DMA_wr.write(false);
DMA_pl_stat.write(0);
}
break;

// case init_next_ptr_params
// Next pointer parameters are always two control reads:
DMA_addr.write(dma_address.read());
DMA_number.write(2);
DMA_ctrl_rd.write(true);
// additional signals, added for synthesizability purposes
DMA_data_rd.write(false);
DMA_wr.write(false);
DMA_pl_stat.write(0);
break;

default:
// default, assign signals for synthesizability purposes
DMA_addr.write(DMA_addr.read());
DMA_number.write(DMA_number.read());
DMA_data_rd.write(false);
DMA_ctrl_rd.write(false);
DMA_wr.write(false);
DMA_pl_stat.write(0);
break;

} 
// Header mode: reads packet pointer on MLI bus and sends it to the PM
// written signals:
// - DMA_packet_pointer

void tx_dma_sm::DMA_packet_pointer_gen()
{
if (!resetN.read()) {
DMA_packet_pointer.write(0);
} else {
if ((state.read() == rd_pkt_ptr) && !delay1_tx_MLB_drdyN.read()) {
DMA_packet_pointer.write(tx_MLB_idata.read());
}
else {
DMA_packet_pointer.write(DMA_packet_pointer.read());
}
}

}

// This signal indicates to fifo filler the header transfer beginning
// written signals:
// - DMA_hd_begin

void tx_dma_sm::DMA_hd_begin_gen()
{
if (!resetN.read()) {
DMA_hd_begin.write(false);
} else {
if ((state.read() == rd_pkt_ptr) && addr_generator_end.read()) {
DMA_hd_begin.write(true);
} else {
DMA_hd_begin.write(false);
}
}
// Generation of signal MIC_dma_hd_rdy for HMAC when header is in FIFO
// written signals:
void tx_dma_sm::MIC_header_rdy_gen() {  
  if (!resetN.read()) {  
    MIC_dma_hd_rdy.write(false);  
  } else {  
    if ((state.read() == rd_header) && addr_generator_end.read()) {  
      MIC_dma_hd_rdy.write(true);  
    } else if ((state.read() == wait_fetch_hd) && (num_in_fifo_mic.read() == 0)) {  
      MIC_dma_hd_rdy.write(false);  
    }  
  }  
}

// This signal indicates to fifo filler that data that is coming is data
// to put into fifo
void tx_dma_sm::data_mode_gen() {  
  if (!resetN.read()) {  
    DMA_data_mode.write(false);  
  } else {  
    if ((state.read() == rd_header) || (state.read() == wait_rd_completed)) {  
      DMA_data_mode.write(true);  
    } else {  
      DMA_data_mode.write(false);  
    }  
  }  
}

// In header mode:
// Reads first segment address and size on MLB bus
// same in payload mode
// written signals:
void tx_dma_sm::DMA_frag_params_gen() {  
  if (!resetN.read()) {  
    DMA_frag_params.write(0);  
    segment_addr.write(0);  
  } else {  
    if ((state.read() == rd_next_seg_params) && !delay1 tx_MLB_drtyN.read() &&  
      tx_MLB_drtyN.read()) {  
      DMA_frag_params.write(DMA_frag_params.read());  
      segment_addr.write(tx_MLB_idata.read());  
    } else if ((state.read() == rd_next_seg_params) && !delay1 tx_MLB_drtyN.read() &&  
      !tx_MLB_drtyN.read()) {  
      DMA_frag_params.write(DMA_frag_params.read());  
      segment_addr.write(tx_MLB_idata.read());  
    } else {  
      DMA_frag_params.write(DMA_frag_params.read());  
      segment_addr.write(segment_addr.read());  
    }  
  }  
}

// Generation of signal DMA_header_ok pulse for PM when DMA in header mode
// has returned to idle state
// signal num_in_fifo_mic is zero when the HMAC has emptied the fifo

// written signals:
void tx_dma_sm::DMA_header_ok_gen() {
    if (!resetN.read()) {
        DMA_header_ok.write(false);
    } else {
        if (state.read() == wait_fetch_hd && (num_in_fifoMic.read() == 0)) {
            DMA_header_ok.write(true);
        } else {
            DMA_header_ok.write(false);
        }
    }
}

void tx_dma_sm::MLB_drdyN_gen() {
    if (!resetN.read()) {
        delay1_tx_MLB_drdyN.write(true);
    } else {
        delay1_tx_MLB_drdyN.write(tx_MLB_drdyN.read());
    }
}

void tx_dma_sm::lastseg_gen() {
    if (!resetN.read()) {
        lastseg.write(0);
    } else {
        if (state.read() == rd_next_ptr_params && !delay1_tx_MLB_drdyN.read() && !tx_MLB_drdyN.read()) {
            // first bit of the next seg param (on MLB if the given drdyN
            // condition fulfilled)
            lastseg.write((bool)(tx_MLB_idata.read()][31]));
        } else if (state.read() == rd_next_seg_params) {
            lastseg.write(0);
        } else if (state.read() == wait_end_msdu && MAC_txm_done.read() && !MAC_txm_fragOk.read() && !MAC_txm_msdu_end.read()) {
            // retry: restore the lastseg flag of the beginning of the last frame
            lastseg.write(cur_frag_lastseg.read());
        } else {
            lastseg.write(lastseg.read());
        }
    }
}

// Generation of current fragment characteristics storage
// These values are generated each time a new fragment is begun and in the
// case of a retry, the values can simply be restored from those. The
// following information is needed:
// cur_frag_ptr: always points to the start address of the current fragment
// cur_frag_data_count: stores the value that is in DMA_frag_seg_count
void tx_dma_sm::retry_parameters_gen () {
    if (!resetN.read()) {
        cur_frag_ptr.write(0);
        cur_frag_data_count.write(0);
        cur_frag_lastseg.write(0);
    } else {
        // segment not aligned
        cur_frag_data_count.write((sc_uint<16>)(DMA_frag_seg_count.read() .range(11,2)) + 1);
        cur_frag_ptr.write(segment_addr.read());
        cur_frag_lastseg.write(0);
    } 
    else if (state.read() == wait_end_msdu && MAC_txm_done.read() && MAC_txm_frag_ok.read() && (rem_data_count.read() != 0)) {
        // wait_end_msdu -> addr_gen_calc: Next fragment, keep the data of
        // this next fragment, if the current fragment begins with unaligned
        // 2-bytes, the last word of the preceding frame needs to be rewritten
        // onto FIFO
        if (address_addon.read() && frags_unaligned.read()) {
            // if current frag has no addon and frags unaligned, then the
            // present frag must begin one word earlier.
            // NOTE: the address is simply decreased by 1, this will only work
            // as long as there are not multiple segments per packet except the
            // LLC-snap
            cur_frag_ptr.write(dma_address.read() -1);
            cur_frag_data_count.write(rem_data_count.read() +1);
            cur_frag_lastseg.write(lastseg.read());
        } else {
            cur_frag_ptr.write(dma_address.read());
            cur_frag_data_count.write(rem_data_count.read());
            cur_frag_lastseg.write(lastseg.read());
        }
    } else {
        cur_frag_ptr.write(cur_frag_ptr.read());
        cur_frag_data_count.write(cur_frag_data_count.read());
        cur_frag_lastseg.write(cur_frag_lastseg.read());
    }
}

} // Flush the fifo and the serializer on receiving either MAC_txm_done
// signal without MAC_txm_frag_ok signal or MAC_txm_msdu_end signal.
// ceg: modified from original version

// (the number of 4byte words remaining in the segment
// NOTE (Ceg): change the naming to cur_firstseg_count
// or similar, naming is misleading
// cur_frag_lastseg: stores whether the current segment is the last
// written signals:
// cur_frag_ptr
// cur_frag_data_count
// cur_frag_lastseg

B SOURCE CODE OF THE ADAPTED DMA
void tx_dma_sm::DMA_flush_gen() {
    if (!resetN.read()) {
        DMA_flush.write(false);
    } else {
        if (MAC_txm_done.read() && (MAC_txm_frag_ok.read() || MAC_txm_msdn_end.read())) {
            DMA_flush.write(true);
        } else {
            DMA_flush.write(false);
        }
    }
}

// Famous enable bits
// For the moment (2.5.2006), keep them all at 1, then the DMA is working fine. However those bits need to be removed in the future and the HMAC has to deal itself with unaligned fragments/segments
void tx_dma_sm::enable_bits_gen() {
    // set enable bits to 1
    seg_mid_enable.write(15); // 1111
}

// Generate a signal indicating if the fragment size (given by the MAC_txu_frag_size register) is unaligned. frags_unaligned is 0 if the fragments are aligned, 1 if they are unaligned
void tx_dma_sm::frags_unaligned_gen() {
    frags_unaligned.write(MAC_txu_frag_size.read()[1]);
}

// Contains the number of words needed to be added in case of an unaligned fragment (0 or 1)
// (Note: If the frames are unaligned, every second fragment either begins or ends with a two byte piece. In practice, the DMA needs to write an additional 4byte word to the FIFO for every second fragment, whose size is otherwise calculated taking MAC_txu_frag_size mod 4.
void tx_dma_sm::address_addon_gen() {
    if (frags_unaligned) {
        address_addon.write(!fragment_count.read()[0]);
    } else {
        address_addon.write(0);
    }
}

// Counters

// This counter is address counter that increments directly from the size of the reads made by DMA in order for it to know where it is in the segment independently of addr_generator address counter
// - dma_address
void tx_dma_sm::address_counter() {
    if (!resetN.read()) {
        dma_address.write(0);
    } else {
        switch (state.read()) {
        case idle:
            if (!PM_start_payload.read()) {
                dma_address.write(0);
            } else {
                // Further cases...
            }
        case transmit:
            // Further cases...
        }
    }
}
DMA 757

dma_address . write (cur_frag_ptr . read () ) ;
}
else {
    dma_address . write (dma_address . read () ) ;
}
break;

case rd_pkt_ptr :
    if (addr_generator_end . read () ) {
        // Initialization of the counter with pkt_ptr address + 36
        dma_address . write (DMA_packet_pointer . read () + 36 ) ;
    } else {
        dma_address . write (dma_address . read () ) ;
    }
break;

case rd_next_ptr_params :
    if ( ! delay1_txi_drdyN . read () & & tx_MLB_drdyN . read () ) {
        dma_address . write (tx_MLB_idata . read () ) ;
    } else {
        dma_address . write (dma_address . read () ) ;
    }
break;

case wait_rd_completed :
    if (addr_generator_end) {
        // Make sure that update of address counter happens only once for
        // each addr gen call.
        dma_address . write ((uint<32>)(dma_address . read () + DMA_number . read () + 4)) ;
    } else {
        dma_address . write (dma_address . read () ) ;
    }
break;

default :
    dma_address . write (dma_address . read () ) ;
break;

    }
}

// rem_data_count value represents the remaining data to be sent of one
// segment (units of 4 bytes)
// written signals:

// = rem_data_count

void tx_dma_sm::rem_data_counter () {
    if ( ! resetN . read () ) {
        rem_data_count . write (0) ;
    } else {
        switch (state . read () ) {
    case idle :
        if (PM_start_payload . read () ) {
            rem_data_count . write (cur_frag_data_count . read () ) ;
        } else {
            rem_data_count . write (rem_data_count . read () ) ;
        }
break;
    case rd_next_ptr_params :
        if ( ! delay1_txi_drdyN . read () & & ! tx_MLB_drdyN . read () ) {
            // Update the rem_data_count with the values read in the next
            // segment information
            // If segment is not aligned, an additional 4 byte word needs to
            // be written to the FIFO
            if (tx_MLB_idata . read () [1] == 1) {
            // segment not aligned
        ...
rem_data_count.write(((sc_uint<10>)(tx_MLB_idata.read()).range(11,2)) + 1);
}
else {
    rem_data_count.write((sc_uint<10>)(tx_MLB_idata.read()).range(11,2));
}
else {
    rem_data_count.write(rem_data_count.read());
}
break;

case addr_gen_calc:
    if (rem_fragdata_count <= (tx_mic_fifo_size.read() - num_in_fifo_mic.read()) &&
        rem_fragdata_count.read() <= rem_data_count.read()) {
        rem_data_count.write(rem_data_count.read());
    }
    else if (rem_data_count.read() <= (tx_mic_fifo_size.read() - num_in_fifo_mic.read()) &&
             rem_data_count.read() < rem_fragdata_count.read()) {
        rem_data_count.write(0);
    }
    else {
        rem_data_count.write(rem_data_count.read() - tx_mic_fifo_size.read() + num_in_fifo_mic.read());
    }
break;

default:
    rem_data_count.write(rem_data_count.read());
break;
}

// rem_fragdata_count value represents the remaining data to be sent of
// one fragment (units of 4 bytes)
// written signals:
// - rem_fragdata_count

void tx_dma_sm::rem_fragdata_counter() {
    if (!resetN.read()) {
        rem_fragdata_count.write(0);
    } else {
        switch (state.read()) {
            case rd_next_seg_params:
                // rem_fragdata_count is initialized after each header read:
                // MAC txm_frag_size
                // address addon takes into consideration that the fragment might
                // be unaligned and an additional 4 byte words needs to be written to
                // FIFO
                rem_fragdata_count.write((sc_uint<10>)(MAC_txu_frag_size.read().range(11,2)) + address_addon.read());
                break;
            case addr_gen_calc:
                if (rem_fragdata_count.read() <= (tx_mic_fifo_size.read() - num_in_fifo_mic.read()) &&
                    rem_fragdata_count.read() <= rem_data_count.read()) {
                    // in next address_generator call, the fragment will be entirely
                    // written, no data remaining
                    rem_fragdata_count.write(0);
                }
                else if (rem_data_count.read() <= (tx_mic_fifo_size.read() - num_in_fifo_mic.read()) &&
                         rem_data_count.read() < rem_fragdata_count.read()) {
                    // the remaining data of the current segment can be written
                    // entirely, no need to fill fragment
                    break;
                } else {
                    rem_data_count.write(rem_data_count.read());
                }
        }
    }
}
rem_fragdata_count.write(rem_fragdata_count.read() - rem_data_count.read());
}
else {
    // next write to FIFO is limited by space in FIFO, write as much
    // as possible (given by tx_mic_fifo_size - num_in_fifo мик)
    rem_fragdata_count.write(rem_fragdata_count.read() -
        tx_mic_fifo_size.read() + num_in_fifo мик.read());
}
break;

case wait_end_msdu:
    // reset the fragdata counter in the wait_end_msdu state
    if (MAC_txm_done.read() && !MAC_txm_frag_ok.read() && !
        MAC_txm_msdu_end.read()) {
        // retry
        if (frags_unaligned.read()) {
            // in unaligned case, fragments following retry are ALWAYS one
            // word longer because either the first or the last word needs
            // to be added to the pile
            rem_fragdata_count.write((sc_uint<10>)(MAC_txu_frag_size.read()).
                range(11,2)) + 1);
        } else {
            rem_fragdata_count.write((sc_uint<10>)(MAC_txu_frag_size.read()).
                range(11,2)) + address_addon.read());
        }
    } else {
        rem_fragdata_count.write((sc_uint<10>)(MAC_txu_frag_size.read()).
            range(11,2)) + address_addon.read());
    }
break;

    // no retry: every second frame is one word longer than the others
    if (fragment size unaligned)
        rem_fragdata_count.write((sc_uint<10>)(MAC_txu_frag_size.read()).
            range(11,2)) + address_addon.read());
break;

    default:
        rem_fragdata_count.write(rem_fragdata_count.read());
break;
}
}

// Generates the fragment counter.
// The fragments are counted beginning with zero (0)

void tx_dma_sm::frag_counter() {
if (!resetN.read()) {
    fragment_count.write(0);
} else {
    switch (state.read()) {
    case wait_fetch_hd:
        // reset the fragment counter to zero when reading a new header
        fragment_count.write(0);
        break;

    case wait_rd_completed:
        if (addr_generator_end.read() && (rem_data_count.read() != 0) && (!
            rem_fragdata_count.read() == 0)) {
            // increase the current fragment number
            fragment_count.write(fragment_count.read() + 1);
        } else {
            fragment_count.write(fragment_count.read());
        }
break;

    case wait_end_msdu:
        if (MAC_txm_done && !MAC_txm_frag_ok && !MAC_txm_msdu_end) {
            // retry: subtract one from the current frame-number
            fragment_count.write(fragment_count.read() - 1);
        } else {
            fragment_count.write(fragment_count.read());
        }
break;
}
B SOURCE CODE OF THE ADAPTED DMA

Listing 6: tx_fifo_filler_cleaned.cc - Transmit DMA FIFO Filler

```c
/* Source code of the adapted DMA*/

break;

default:
break;
}
```

---

```
#include "Tx_fifo_filler.h"

// Write pulse for FIFO
void tx_fifo_filler::write_gen() {
    if (!resetN.read()) {
        write_fifo.write(false);
    } else {
        // Data coming from MLB or // 0401015 remove 'or' => if ... else if ...
        if (!MLB_dioYN.read() && DMA_data_mode.read() && (data_cnt_delayed.read () > 0)) {
            write_fifo.write(true);
            // ... end of fragment: last fragment word must be repeated in FIFO ...
        }

        // ceg: exclude all the else clauses
        /*
        else if ( DMA_frag_begin.read() &&
            !DMA_first_frag.read() && // 04 01 15
            !seg_ltrf.read() &&
            // 040128 not(end of segment = end
            of frag)
            (frag_begin_enable.read() != 0) &&
            (frag_begin_enable.read() != 1)) {
            // ... and, if we are at end of segment, ... 
            if (seg_cnt.read() == 0) {  // 040115: segment end
                // ... only if end of fragment and end of
                segment aren't aligned
                if (frag_end_enOld.read() != seg_end_enable.read()) {
                    write_fifo.write(true);
                } else {
                    // If they are aligned, no word must be repeated
                    write_fifo.write(false);
                }
        } else {
```
// No end of segment, but end of fragment and its enable isn’t null
// => word must be repeated
// or data is coming from MLB => write of this word in FIFO
write_fifo.write(false);
}
*/
else {
  // No data coming from MLB and no end of fragment => no FIFO write
  write_fifo.write(false);
}
}

// This register sends data from MLB data bus to FIFO data bus
void tx_fifo_filler::data_gen() {
  if (!resetN.read()) {
    data.write(0);
  } else {
    if (!MLB_drdyN.read() & & DMA_data_mode.read()) {
      data.write(tx_MLB_idata.read());
    }
  }
}

// Sends correct enable to FIFO corresponding to current data sent
void tx_fifo_filler::enable_gen() {
  if (!resetN.read()) {
    enable_in.write(0x0);
  } else {
    enable_in.write(0x0F); // 1111
  }
  /
  } else {
    if (!MLB_drdyN.read()) {
      if (seg_cnt.read() == 0) {
        // End of segment & end of fragment, take the shorter.
        enable_in.write(frag_end_enable.read() & seg_end_enable.read());
      } else if (seg_cnt.read() < (DMA_segsz.read() - 1)) {
        // End of fragment + (segment end or segment middle)
        enable_in.write(frag_end_enable.read());
      } else {
        // End of fragment + Segment begin
        enable_in.write(frag_end_enable.read() & seg_begin_enable.read());
      }
    }
  }
  if (seg_cnt.read() == 0) {
    // End of segment
    // 040205: if begin of fragment:
    // 040205b: take into account frag_begin_enable = 0 => all valid
    if ( !if_frag_begin & & frag_begin_enable.read() != 0 ) {
      enable_in.write(frag_begin_enable.read() & seg_end_enable.read());
    } else {
      enable_in.write(seg_end_enable.read());
    }
  } else if (seg_cnt.read() == (DMA_segsz.read() - 1)) {
    // Segement begin
    enable_in.write(seg_begin_enable.read());
  } else {
    // Middle of segment
    enable_in.write(seg_mid_enable.read());
  }
}
SOURCE CODE OF THE ADAPTED DMA

```c
127 }
128 }
129 } else if (DMA_frag_begin.read()) {
130  // End of fragment: last fragment word must be repeated in FIFO
131  // with appropriate byte enable
132  // 040115: if (seg_cnt.read() == DMA_segsz.size.read()) {
133  if (seg_cnt.read() == 0) { // 040115: segment end
134  enable_in.write(frag_begin_enable.read() & seg_end_enable.read());
135  }
136  } else {
137  enable_in.write(frag_begin_enable.read());
138  }
139 }
140 
141 /*
142 */
143 }
144 
145 // This signal is tx_MLB_drdyN shifted by one mem_clk period
146 // so that when it is asserted, data can be read on MLB data bus
147 void tx_fifo_filler::MLB_drdy_gen() {
148  if (!resetN.read()) {
149    MLB_drdyN.write(true);
150  } else {
151    MLB_drdyN.write(tx_MLB_drdyN.read());
152  }
153 }
154 
155 // Segsize counter (in words) used for sending correct enable
156 void tx_fifo_filler::segsize_counter() {
157  /*
158  if (!resetN.read()) {
159    seg_cnt.write(0);
160  } else {
161    if (DMA_init_seg.read()) {
162      // Initialization with DMA_segsz on receiving init signal
163      seg_cnt.write(DMA_segsz.size.read());
164    } else if ((tx_MLB_drdyN.read() && DMA_data_mode.read() && (data_cnt.read() > 0)) { // decrements by one word every time a data for FIFO is incoming
165      seg_cnt.write(seg_cnt.read() - 1); // and was a requested data (<= data_cnt > 0)
166    }
167  }
168  */
169 }
170 
171 // Counter of words in one DMA request
172 void tx_fifo_filler::burstsize_counter() {
173  /*
174  if (!resetN.read()) {
175    burst_cnt.write(0);
176  } else {
177    if (DMA_data_rd.read()) {
178      // Initialization with number of words in current DMA request
179      burst_cnt.write(DMA_number.read());
180    } else if ((tx_MLB_drdyN.read() && DMA_data_mode.read() && (data_cnt.read() > 0)) { // Decrement when data incoming and was a requested data
181      burst_cnt.write(burst_cnt.read() - 1); // (<=> data_cnt > 0)
182    }
183  */
184 }
185 
186 // Fragsize counter (in words) used for sending correct enable
187 void tx_fifo_filler::fragsize_counter() {
188  /*
189  if (!resetN.read()) {
190    frag_cnt.write(0);
191  */
192 ```
B SOURCE CODE OF THE ADAPTED DMA

```c
frag_end_enOld.write(0); // 040120
}
else {
    if (DMA_hd_begin.read()) {
        // Init with header size (in words)
        frag_cnt.write(9);
    } else if (DMA_frag_begin.read()) {
        // Init with aligned fragment size (in words = / 4)
        frag_cnt.write((frag_count_al.read()) range(11, 2));
        frag_end_enOld.write(frag_end_enable.read()); // 040120
    } else if (DMA_init_seg.read()) { // 040121b – for any new seg !
        // New seg in last fragment word, refresh value for un-aligned
        frag_cnt.write((frag_count_al.read()) range(11, 2));
    }
}

// Data counter: counts data coming out of FIFO.
// Fifo filler mustn’t take into account when more data
// is incoming than requested.
void tx_fifo_filler::data_counter()
{
    if (!rstN.read()) {
        data_cnt.write(0);
    } else {
        if (init_data_cnt.read()) {
            // Init by addr_generator with burst size value
            data_cnt.write(init_value_read());
        } else if (DMA_data_mode.read() & tx_MLB_drdyN.read() & (data_cnt.read() > 0)) {
            // Decrments when data incoming unless counter reached 0
            data_cnt.write(data_cnt.read() - 1);
        } else {
            data_cnt.write(data_cnt.read());
        }
    }
}

// Delayed data counter: same than data counter but reacts on MLB_drdyN
void tx_fifo_filler::data_counter_delayed()
{
    if (!rstN.read()) {
        data_cnt_delayed.write(0);
    } else {
        data_cnt_delayed.write(data_cnt.read());
    }
}

// 040205: First data of a fragment flag.
// Used for byte enable: 1rst word of frag = last word of seg.
void tx_fifo_filler::fif_frag_begin_gen()
{
    if (!rstN.read()) {
        fif_frag_begin.write(false);
    } else {
        // Set on first frag indicator:
        // First frag of msdu or after a retry
        if (DMA_frag_begin.read() & DMA_first_frag.read()) {
            fif_frag_begin.write(true);
        } else if (!MLB_drdyN.read() & DMA_data_mode.read()) {
            // Released by first data write.
            fif_frag_begin.write(false);
        }
    }
}
```

73
C Source Code of the Synthesis Part

Listing 7: mic_syn.csh

```bash
#!/bin/csh -f
#
# Written by Serge Lopez / EPFL-IMM-LSM / 01-11-2004
# This script is the main script for synthetizing the MIC module.
# Adapted 03-06-2006 by Christoph Eggimann / EPFL
#
# create the report directory if not an overall compilation
if ($OVERALL_COMPILATION == 0) then
    set LOG_DIR=$SYN_LOG/LOG: date +%y%M%d%H%M%S
    mkdir -p $LOG_DIR
    echo "Create MIC report directory"
    echo "directory path: $LOG_DIR"
endif

# Interrupt jump
onintr out

echo "Resynthesize MIC Files"

echo "----------------------"

# remove test directory — later to be changed for the dc_fpga directory
echo "Remove old DC FPGA Work directory for MIC"
/bin/rm -rf $SYN_SCRIPTS/DC_FPGA_WORK/mic

# remove old analysis file, will be immediately replaced by next line
echo "Remove old mic analysis source file (mic_sources.tcl)"
/bin/rm $SYN_SCRIPTS/mic_sources.tcl

# create the analysis file containing all mic verilog sources
echo "Create new analysis source file (mic_sources.tcl)"
$SYN_SCRIPTS/mic_create_analysis_file.csh

# launch the DC FPGA shell in command line. Stdout and Stderr will be
# stored in the current log directory.
echo "Begin DC FPGA Synthesis of mic"
/dc_fpga_shell=t -f $SYN_SCRIPTS/mic_syn_dcfpga.tcl > $LOG_DIR/mic_dcfpga.log
echo "Finished DC FPGA Synthesis of mic"

mv $SYN_LOG/mic_dcfpga.rpt $LOG_DIR/mic_dcfpga.rpt
mv $SYN_LOG/mic_dcfpga.timing.rpt $LOG_DIR/mic_dcfpga.timing.rpt
mv $SYN_LOG/mic_dcfpga.ref.rpt $LOG_DIR/mic_dcfpga.ref.rpt

exit
```

Listing 8: mic_create_analysis_file.csh

```bash
#!/bin/csh -f
#
# Written by Christoph Eggimann, EPFL, 3.6.2006, following the script
#
```

74
The script creates the file mic_sources.tcl which contains all the analysis commands to be executed by DC FPGA. mic_sources.tcl is called from the main synthesis script, mic_syn_dcfpga.tcl.

Listing 9: mic_syn_dcfpga.tcl
C SOURCE CODE OF THE SYNTHESIS PART

```c
# working.
source $SYN_SCRIPTS/mic_sources.tcl

# elaborate the analyzed source files. Only the toplevel instance needs to be
# called expliciteley, without any extension.
elaborate mic
write -hier -f db -o $SYN_DB/mic.elab.db
link

current_design mic
# Set the FPGA device to be used. A list of all available FPGAs can be obtained
# using the command 'set_fpga_target_device -show_all'. This command needs to
# be typed into the DC FPGA shell after the libraries had been properly set
(set_fpga_target_device $DEVICE)
# Create Clock, false path, multicycle path etc.
create_clock [get_port host_clk] -period 30
create_clock [get_port mem_clk] -period 45
set_false_path -from [get_clock host_clk] -to [get_clock mem_clk]
set_false_path -from [get_clock mem_clk] -to [get_clock host_clk]
set_false_path -from [get_clock host_clk] -to [get_clock mem_clk]
set_false_path -from [get_clock mem_clk] -to [get_clock host_clk]
set_false_path -from [get_clock mem_clk] -to [get_clock MAC_clk]
set_false_path -from [get_clock MAC_clk] -to [get_clock mem_clk]
current_design mic
ungroup -all -flatten

# Compile the mic design
compile -boundary_optimization
write -hier -f edif -o $SYN_EDIF/mic.edf
write -hier -f db -o $SYN_DB/mic.db
report_fpga > $SYN_LOG/mic_fpga.rpt
report_timing > $SYN_LOG/mic_timing.rpt
report_resources -hier > $SYN_LOG/mic_res.rpt
report_reference > $SYN_LOG/mic_ref.rpt
quit
```

Listing 10: hmac_syn.csh

```bash
#!/bin/csh -f
###
#
# Adapted 03–06–2006 by Christoph Eggimann / EPFL from Serge Lopez’
# Files. This file synthesizes the HMAC.
#
#
# set the other environmental variables
#
# create the report directory if not in overall compilation
if ($OVERALL_COMPILATION == 0) then
    LOG_DIR=$SYN_LOG/LOG_DATE +$SYN_EDIF/$SYN_DB/$SYN_RES/$SYN_REF
    mkdir -p $LOG_DIR
    mkdir -p $LOG_DIR
    echo "Create HMAC report directory"
    echo " directory path: $LOG_DIR"
```
# Interrupt jump onintr out

```c
#endif
```

```c
# Interrupt jump

```

```c
interrupt

```

```c
onintr

```

```c
out

```

```c
#echo "Resynthesize HMAC Files"

```

```c
#echo "----------------------"

```

```c
# go to BBIF source directory (so DCFPGA finds the included .vh files)

```

```c
cd $BBIF_HDL_SRC

```

```c
# remove test directory — later to be changed for the dc_fpga directory

```

```c
/bin/rm -rf $SYN_DCFPGA_WORK/hmac

```

```c
# remove old analysis file, will be immediately replaced by next line

```

```c
#echo "Remove old DC FPGA Work directory for HMAC"

```

```c
/bin/rm -f $SYN_SCRIPTS/hmac_sources.tcl

```

```c
#echo "Create new analysis source file ( hmac_sources.tcl)"

```

```c
$SYN_SCRIPTS/hmac_create_analysis_file.csh

```

```c
# launch the DC FPGA shell in command line. Stdout and Stderr will be

```

```c
# stored in the current log directory.

```

```c
echo "Begin DC FPGA Synthesis of HMAC"

```

```c
fpga-shell -t -f $SYN_SCRIPTS/hmac_syn_dcfpga.tcl > $LOG_DIR/hmac_dcfpga.log

```

```c
echo "Finished DC FPGA Synthesis of HMAC"

```

```c
mv $SYN_LOG/hmac_fpga.rpt $LOG_DIR/hmac_fpga.rpt

```

```c
mv $SYN_LOG/hmac_syn_dcfpga.log $LOG_DIR/hmac_dcfpga.rpt

```

```c
mv $SYN_LOG/hmac_timing.rpt $LOG_DIR/hmacTiming.rpt

```

```c
mv $SYN_LOG/hmac_res.rpt $LOG_DIR/hmac_res.rpt

```

```c
mv $SYN_LOG/hmac_ref.rpt $LOG_DIR/hmac_ref.rpt

```

```c
## ***************

```

```c
exit

```

```c
## ***************

```

```c
# --- 'C interrupt

```

```c
out:

```

```c
#echo "Synthesis interrupted in script hmac_syn.csh"

```

```
```

```
Listing 11: hmac_create_analysis_file.csh

```c
#!/bin/csh -f

```

```c
#

```

```c
# Written by Christoph Eggimann, EPFL, 3.6.2006, following the script

```

```c
# The script creates the file hmac_sources.tcl which in contains all

```

```c
# the analysis commands to be executed by DC FPGA. hmac_sources.tcl is

```

```c
# called from the main synthesis script, hmac_syn_dcfpga.tcl

```

```c
##

```

```c
# looking for the Verilog files of the HMAC, put list of all verilog files in

```

```c
# file LOGI.txt

```

```c
find HMAC_HDL_SRC | grep '.\v$' > LOGI.txt

```

```c
chmod 777 LOGI.txt

```

```c
# create a list of analyze commands and write it in the file hmac_sources.tcl

```

```c
# the order in which the files are written does not matter for synthesis

```

```c
foreach f ('less LOGI.txt')

```

```c
echo analyze -format verilog $f >> $SYN_SCRIPTS/hmac_sources.tcl

```

```c
end

```

```c
echo "" >> $SYN_SCRIPTS/hmac_sources.tcl

```

```c
```
```
C SOURCE CODE OF THE SYNTHESIS PART

```c
# remove the temporary file
rm -rf LOG1.txt

# looking for the Verilog files of the HMAC, put list of all verilog files in
# file LOG1.txt
find $BBIF_HDL_SRC | grep ‘\.v$’ > LOG1.txt
# give all rights to LOG1.txt
chmod 777 LOG1.txt

echo analyze -format verilog $f >> $SYN_SCRIPTS/bbif_sources.tcl
end

echo "" >> $SYN_SCRIPTS/bbif_sources.tcl

# create a list of analyze commands and write it in the file bbif_sources.tcl
# the order in which the files are written does not matter for synthesis
foreach f (‘less LOG1.txt’)
    echo analyze -format verilog $f >> $SYN_SCRIPTS/bbif_sources.tcl
end

echo "" >> $SYN_SCRIPTS/bbif_sources.tcl

# remove the temporary file
rm -rf LOG1.txt

# interrupt
out:
    echo "Synthesis interrupted in script hmac_create_analysis_file.csh"
```

Listing 12: hmac_syn_dfpga.tcl

```
# Written by Christoph Eggimann, EPFL, 3.6.2006, following the script
dc_fpga_hmac.scr by Pascal Delamotte (?)
The script is passed to the DC FPGA shell and uses the file
hmac_sources.tcl as an input containing references to all verilog files
# to be analyzed.
#
set SYN_ROOT [get_unix_variable SYN_ROOT]
set SYN_DB [get_unix_variable SYN_DB]
set SYN_EDIF [get_unix_variable SYN_EDIF]
set SYN_LOG [get_unix_variable SYN_LOG]
set SYN_SCRIPTS [get_unix_variable SYN_SCRIPTS]
set DEVICE 2V4000FF1152

# Setup the tool. The synopsys setup file needs to be adapted to the
# FPGA in use. See ‘synlibs’ command
source $SYN_ROOT/synopsys_dc_setup
# NOTE: to do get the output of the source into the log file. —echo is not
# working.

# Define the WORK directory
define_design_lib WORK –path $SYN_ROOT/dfpga/hmac

# Pragmas in code
set hdlin_translate_off,skip_text true

# Very important: Will perform basic variables setting depending on the
# FPGA targeted
set fpga_defaults xilinx_virtex2 –verbose

# Enhance LUT compacting (area driven)
set fpga_patch_luts true

# analyze RTL files. The script hmac_sources.tcl contains a list of the
# analyze commands. It can be created using hmac_create_analysis_file.csh
# NOTE: to do get the output of the source into the log file. —echo is not
# working.
```
source $SYN_SCRIPTS/hmac_sources.tcl
source $SYN_SCRIPTS/bbif_sources.tcl

# elaborate the analyzed source files. Only the toplevel instance needs to be
# called explicitly, without any extension.
elaborate hmac

write -hier -f db -o $SYN_DB/hmac.elab.db

link

current_design hmac

set fpga_target_device $DEVICE

create_clock [get_port host_clk] -period 30
create_clock [get_port mem_clk] -period 45
create_clock [get_port phy_clk] -period 45

set_false_path -from [get_clock host_clk] -to [get_clock mem_clk]
set_false_path -from [get_clock host_clk] -to [get_clock phy_clk]
set_false_path -from [get_clock mem_clk] -to [get_clock phy_clk]
set_false_path -from [get_clock phy_clk] -to [get_clock host_clk]
set_false_path -from [get_clock phy_clk] -to [get_clock mem_clk]

ungroup -all -flatten

compile -boundary_optimization

write -hier -f edif -o $SYN_EDIF/hmac.edf
write -hier -f db -o $SYN_DB/hmac.db

report_fpga > $SYN_LOG/hmac_fpga.rpt
report_timing > $SYN_LOG/hmac_timing.rpt
report_resources -hier > $SYN_LOG/hmac_res.rpt
report_reference > $SYN_LOG/hmac_ref.rpt

quit

#!/bin/csh -f
##
###
#
# Written by  Serge Lopez / EPFL-IMM-LSM  / 01–11-2004
# This script is the main script for synthetizing the PCI module.
# Adapted 06–06–2006 by Christoph Eggimann / EPFL
#
#
# create the report directory if not an overall compilation
if ($OVERALL_COMPILATION == 0) then
    set LOG_DIR=$SYN_LOG/LOG; date +"%m_%d_%Y\ %H\ %M\ %S"
    mkdir -p $LOG_DIR
    echo "Create PCI report directory"
    echo " directory path: "$LOG_DIR"
endif

# Interrupt jump
onintr out

Listing 13: pci_syn.csh
22 echo "Resynthesize PCI Files"  
24 # remove test directory — later to be changed for the dc.fpga directory  
26 echo "Remove old DC FPGA Work directory for PCI"  
28 /bin/rm -rf $SYNDCFPGA_WORK/pci  
30 # go to source directory  
32 cd $PCI_HDL_SRC  
34 # remove old analysis file, will be immediately replaced by next line  
36 echo "Remove old pci analysis source file (pci_sources.tcl)"  
38 /bin/rm $SYNSCRIPTS/pci_sources.tcl  
40 # create the analysis file containing all pci verilog sources  
42 echo "Create new analysis source file (pci_sources.tcl)"  
44 $SYNSCRIPTS/pci_create_analysis_file.csh  
46 # launch the DC FPGA shell in command line. Stdout and Stderr will be  
48 # stored in the current log directory.  
50 echo "Begin DC FPGA Synthesis of pci"  
52 fpga_shell -t -f $SYNSCRIPTS/pci_syn_dcfpga.tcl > $LOG_DIR/pci_dcfpga.log  
54 echo "Finished DC FPGA Synthesis of pci"  
56 mv $SYNLOG/pci_fpga.rpt $LOG_DIR/pci_fpga.rpt  
58 mv $SYNLOG/pci_timing.rpt $LOG_DIR/pci_timing.rpt  
60 mv $SYNLOG/pci_res.rpt $LOG_DIR/pci_res.rpt  
62 mv $SYNLOG/pci_ref.rpt $LOG_DIR/pci_ref.rpt  
64  
68 exit  
72 echo "Synthesis interrupted in script pci_syn.csh"  
74
Listing 14: pci_create_analysis_file.csh

#!/bin/csh -f  
#  
# # Written by Christoph Eggimann, EPFL 6.6.2006, following the script  
# create_add_file_fc2.csh of Serge Lopez / EPFL of 1.11.2004  
# The script creates the file pci_sources.tcl which in contains all  
# the analysis commands to be executed by DC FPGA pci_sources.tcl is  
# called from the main synthesis script, pci_syn_dcfpga.tcl  
#  
# looking for the Verilog files of the PCI, put list of all verilog files in  
# file LOG1.txt  
12 find $PCI_HDL_SRC | grep '.\..vh$' > LOG1.txt  
14 find $PCI_HDL_SRC | grep '.\..v$' >> LOG1.txt  
16 # give all rights to LOG1.txt  
18 chmod 777 LOG1.txt  
20 # create a list of analyze commands and write it in the file pci_sources.tcl  
# the order in which the files are written does not matter for synthesis  
22 foreach f ("less LOG1.txt")  
24 echo analyze -format verilog $f >> $SYNSCRIPTS/pci_sources.tcl  
26 end  
28 echo "" >> $SYNSCRIPTS/pci_sources.tcl  
30 # remove the temporary file  
32 rm -rf LOG1.txt  
34  
38 # #
Listing 15: pci_syn_defpga.tcl

```tcl
# Written by Christoph Eggimann, EPFL, 6.6.2006, following the script # dc_fpga мик.scr by Pascal Delamotte (?)
# The script is passed to the DC FPGA shell and uses the file # pci_sources.tcl as an input containing references to all verilog files # to be analyzed.
#
set SYNROOT [get_unix_variable SYNROOT]
set SYNDB [get_unix_variable SYNDB]
set SYNEDF [get_unix_variable SYNEDF]
set SYNLOG [get_unix_variable SYNLOG]
set SYNSOURCES [get_unix_variable SYNSOURCES]
set DEVICE 2V4000FF1152

# Setup the tool. The synopsys setup file needs to be adapted to the # FPGA in use. See 'synlibs' command
source $SYNROOT/synopsys_dc_setup

# NOTE: to do: get the output of the source into the log file. -echo is not # working.

# Define the WORK directory
define_design_lib WORK –path $SYNROOT/dcfpga/pci

# Pragmas in code
set hdlin_translate_off skip_text true

# Very important: Will perform basic variables setting depending on the # FPGA targeted
set fpga_defaults xilinx_virtex2 –verbose

# Enhance LUT compacting (area driven)
set fpga_patch_luts true

# analyze RTL files. The script pci_sources.tcl contains a list of the # analyze commands. It can be created using pci_create_analysis_file.csh # NOTE: to do: get the output of the source into the log file. -echo is not # working.
source $SYNSOURCES/pci_sources.tcl

# elaborate the analyzed source files. Only the toplevel instance needs to be # called explicitly, without any extension.
elaborate ib2f_32

write –hier –f db –o $SYNDB/pci.elab.db

link

# Set the FPGA device to be used. A list of all available FPGAs can be obtained # using the command ’set_fpga_target_device –show_all’. This command needs to be # typed into the DC FPGA shell after the libraries had been properly set # (see also synlibs, synopsys_dc_setup)
set_fpga_target_device $DEVICE

# Create Clock, false path, multicycle path etc.
create_clock [get_port pciClk] –period 30
create_clock [get_port mlbClk] –period 30
create_clock [get_port rbiClk] –period 30
```

exit
```
```
C SOURCE CODE OF THE SYNTHESIS PART

```c
set_false_path -from [get_clock pciClk] -to [get_clock mlbClk]
set_false_path -from [get_clock mlbClk] -to [get_clock pciClk]
set_false_path -from [get_clock rbiClk] -to [get_clock mlbClk]
set_false_path -from [get_clock mlbClk] -to [get_clock rbiClk]
set_false_path -from [get_clock rbiClk] -to [get_clock pciClk]
set_false_path -from [get_clock pciClk] -to [get_clock rbiClk]

current_design ib2f_32

Listing 16: top_syn.csh

#!/bin/csh -f
###
# Written by  Serge Lopez / EPFL-IMM-LSM / 01-11-2004
# This script is the main script for synthesizing the TOP MICMAC.
# Adapted 06-06-2006 by Christoph Eggimann / EPFL
#
# create the report directory if not an overall compilation
if ($OVERALL_COMPILE == 0) then
    set LOG_DIR = $SYN_LOG/LOG` date +%Y%m%d%H%M%S`
    mkdir -p $LOG_DIR
    echo "Create report directory"
    echo "  directory path: $LOG_DIR"
endif

# Interrupt jump
onintr out

# Remove test directory — later to be changed for the dc_fpga directory
/bin/rm -rf $SYNJCFPGA_WORK/top
/bin/mkdir $SYNJCFPGA_WORK/top
/bin/chmod 777 $SYNJCFPGA_WORK/top

# launch the DC FPGA shell in command line. Stdout and Stderr will be
# stored in the current log directory.
/fpga_shell -t -f $SYNJSCRIPTS/top_syn_dcfpga.tcl > $LOG_DIR/top_dcfpga.log

# Move Reports to the appropriate Log directory
mv $SYNJLOG/top_design.rpt $LOG_DIR/top_design.rpt
mv $SYNJLOG/top_area.rpt $LOG_DIR/top_area.rpt
mv $SYNJLOG/top_hier.rpt $LOG_DIR/top_hier.rpt
mv $SYNJLOG/top_area.rpt $LOG_DIR/top_area.rpt
mv $SYNJLOG/top_res.rpt $LOG_DIR/top_res.rpt
mv $SYNJLOG/top_fpga.rpt $LOG_DIR/top_fpga.rpt
mv $SYNJLOG/top_constr.rpt $LOG_DIR/top_constr.rpt
mv $SYNJLOG/top_timing.rpt $LOG_DIR/top_timing.rpt
```
source $SYN_ROOT/synopsys_dc.setup
# NOTE: to do: get the output of the source into the log file. -echo is not working.

define_work_design_lib WORK -path $SYN_ROOT/dcfpga/top
set hdlin_translate_off skip_text true
set_fpga_defaults xilinx_virtex2 -verbose
set fpga_patch_luts true

read_edif $SYN_EDIF/hmac.edf
read_edif $SYN_EDIF/pci.edf
read_edif $SYN_EDIF/mic.mac.edf
read_edif $SYN_EDIF/ila.edn
read_edif $SYN_EDIF/rx_dual_sram.edn
read_edif $SYN_EDIF/tx_dual_sram.edn

set bus_naming_style "%d<%d>"
set bus_inference_style "%d<%d>"
set bus_extraction_style "%d<%d:4>"

set DEVICE 2V4000FF1152

set SYN_ROOT [get_unix_variable SYN_ROOT]
set SYN_DB [get_unix_variable SYN_DB]
set SYN_EDIF [get_unix_variable SYN_EDIF]
set SYN_LOG [get_unix_variable SYN_LOG]
set SYN_SCRIPTS [get_unix_variable SYN_SCRIPTS]
set HDL_SRC [get_unix_variable HDL_SRC]

Listing 17: top_syn_dcfpga.tcl

# Written by Christoph Eggimann, EPFL, 6.6.2006, following the script
dcfpga.mic.scr by Pascal Delamotte (?)
The script is passed to the DC FPGA shell.

# for tool. The synopsys setup file needs to be adapted to the FPGA in use. See 'synlibs' command
source $SYN_ROOT/synopsys_dc.setup
# NOTE: to do: get the output of the source into the log file. -echo is not working.

# Define the WORK directory
define_design_lib WORK -path $SYN_ROOT/dcfpga/top
set hdlin_translate_off skip_text true

# Pragmas in code
set fpga_defaults xilinx_virtex2 -verbose
set fpga_patch_luts true

# analyze RTL files.
analyze -format vhdl $HDLSRC/dcm44.vhd
analyze -format vhdl $HDLSRC/dcm22.vhd
analyze -format verilog $HDLSRC/rbi_if.v
analyze -format verilog $HDLSRC/micmac.v
analyze -format vhdl $HDLSRC/top_avnet.vhd

read_edif $SYN_EDIF/hmac.edf
read_edif $SYN_EDIF/pci.edf
read_edif $SYN_EDIF/mic.mac.edf
read_edif $SYN_EDIF/ila.edn
read_edif $SYN_EDIF/rx_dual_sram.edn
read_edif $SYN_EDIF/tx_dual_sram.edn

# Adjust Bus naming of Xilinx Coregen Components, according to p.12.3 of
set bus_naming_style "%4<d>"
set bus_inference_style "%8<%d>"
set bus_extraction_style "%8<%d:4>"

# Adjust Bus naming of Xilinx Coregen Components, according to p.12.3 of
set bus_naming_style "%4<d>"
set bus_inference_style "%8<%d>"
set bus_extraction_style "%8<%d:4>"

# Adjust Bus naming of Xilinx Coregen Components, according to p.12.3 of
set bus_naming_style "%4<d>"
set bus_inference_style "%8<%d>"
set bus_extraction_style "%8<%d:4>"

# Adjust Bus naming of Xilinx Coregen Components, according to p.12.3 of
set bus_naming_style "%4<d>"
set bus_inference_style "%8<%d>"
set bus_extraction_style "%8<%d:4>"

# Adjust Bus naming of Xilinx Coregen Components, according to p.12.3 of
set bus_naming_style "%4<d>"
set bus_inference_style "%8<%d>"
set bus_extraction_style "%8<%d:4>"

# Adjust Bus naming of Xilinx Coregen Components, according to p.12.3 of
set bus_naming_style "%4<d>"
set bus_inference_style "%8<%d>"
set bus_extraction_style "%8<%d:4>"

# Adjust Bus naming of Xilinx Coregen Components, according to p.12.3 of
set bus_naming_style "%4<d>"
set bus_inference_style "%8<%d>"
set bus_extraction_style "%8<%d:4>"
elaborate the analyzed source files. Only the toplevel instance needs to be called explicitly, without any extension.

```bash
elaborate top_avnet
write -hier -f db -o $SYN_DB/top.elab.db
link
current_design top_avnet
```

Set the FPGA device to be used. A list of all available FPGAs can be obtained using the command `set_fpga_target_device -show all`. This command needs to be typed into the DC FPGA shell after the libraries had been properly set (see also synlibs, synopsys.dc.setup)

```bash
set_fpga_target_device $DEVICE
```

Create Clock, false path, multicycle path etc.

```bash
create_clock [get_port host_clk] -period 30
create_clock [get_port MAC_clk] -period 45
set_false_path -from [get_clock host_clk] -to [get_clock MAC_clk]
set_false_path -from [get_clock MAC_clk] -to [get_clock host_clk]
current_design top_avnet
ungroup -all -flatten
```

Compile the mic design

```bash
compile -boundary_optimization
write -hier -f edif -o $SYN_EDIF/top.edf
write_par_constraint $SYN_EDIF/top.ncf
write -hier -f db -o $SYN_DB/top.db
```

Reports

```bash
# Reports
# Design reports
report_design > $SYN_LOG/top_design.rpt
report_area > $SYN_LOG/top_area.rpt
report_hierarchy > $SYN_LOG/top_hierarchy.rpt
report_resources -hier > $SYN_LOG/top_res.rpt
report_fpga > $SYN_LOG/top_fpga.rpt
# Other reports
report_parts > $SYN_LOG/top_parts.rpt
report_timing > $SYN_LOG/top_timing.rpt
report_reference > $SYN_LOG/top_ref.rpt
quit
```

Listing 18: main_synthesis.csh

```bash
#!/bin/csh -f
### **
onintr out

if ($#argv == 0) then
goto help
else
  while ( "$!" != "" )
    switch ( "$1" )
      case -h:
        goto help
      case -mic:
        set mic_v = y
        shift
```

84
C  SOURCE CODE OF THE SYNTHESIS PART

```
breaksw
  case -hmac:
    set hmac_v = y
    shift
    breaksw
  case -pci:
    set pci_v = y
    shift
    breaksw
  case -top:
    set top_v = y
    shift
    breaksw
  case -all:
    set all_var = y
    shift
    breaksw
default:
  echo " $1 : not a valid module\n":
  shift
endsw
endif
if ($? all_var) then
  echo " ARE YOU SURE YOU WANT TO RESYNTHESIZE ALL THE HDL OF THE MicMac ?? y /n) ??"
  set answer = $<
  if ($answer = = "y") then
    set mic_v = y
    set hmac_v = y
    set pci_v = y
    set top_v = y
  else
    exit
  endif
endif
# set overall-compilation variable to one. This indicates to the subsequent
# scripts that they don't need to set the environmental variables again.
setenv OVERALL_COMPILE 1
#
# create a directory where all log files of the current compilation are stored
setenv LOG_DIR $SYN_LOG/LOG/%y_%m_%d_%H_%M_%S
mkdir -p $LOG_DIR
echo "Create Report directory"
  echo " directory path: $LOG_DIR"
if ($?mic_v) then
  # $MIC_SYN_DIR/sc2v.csh
  $SYN_SCRIPTS/mic_syn.csh
endif
if ($?hmac_v) then
  # $HMAC_SYN_DIR/sc2v.csh
  $SYN_SCRIPTS/hmac_syn.csh
endif
if ($?pci_v) then
  $SYN_SCRIPTS/pci_syn.csh
endif
if ($?top_v) then
  $SYN_SCRIPTS/top_syn.csh
endif
## ******************************************************************
exit
## ******************************************************************
```
C SOURCE CODE OF THE SYNTHESIS PART

help:
cat << LABEL
Options: 'basename $0' -hmac -mic -all
-mic : synthesize MIC
-hmac : synthesize HMAC
-pci : synthesize PCI
-all : synthesize all modules: MIC, HMAC
LABEL
## **********************
exit
## **********************

--- `C interrupt
out:
echo "Synthesis interrupted in script main_synthesis.csh"

Listing 19: project_setup.source

echo

setenv MIMACROOT /home/ceggiman/MIMAC

setenv SYNNROOT $MIMACROOT/synth
setenv SYNSRC $SYNNROOT/scripts
setenv SYNLORG $SYNNROOT/log
setenv SYNDFFPGAWORK $SYNNROOT/dcfpga
setenv SYNDDB $SYNNROOT/db
setenv SYNEDIF $SYNNROOT/edif
setenv HDL_SRC $SYNNROOT/hdl_src
setenv MIMACHDL_SRC $HDL_SRC/mic
setenv HMAC_HDL_SRC $HDL_SRC/hmac
setenv BBIF_HDL_SRC $HDL_SRC/bbif
setenv OVERALL_COMPILE 0

# Synthetization relevant paths
# synth directory location
# synthetization scripts location
# Log file location
# dcfpga work directory location
# .db file location
# .edif file location
# compiled source code root
# mic source
# hmac source
# pci source
# bbif source
# indicates if more than one block is compiled
# at once (1) or not (0) using main_synthesis.csh

86
References


