

GRENOBLE-INP PHELMA

NANOTECHNOLOGIES FOR ICTs

Development of a generic patch controller hardware

Student :

Supervisor :

BARON Paul
pub@melexis.com

MANSOUR Bassam
bassam.mansour@phelma.grenoble-inp.fr

TUTOR: ANGHEL Lorena
lorena.angel@phelma.grenoble-inp.fr



Politecnico
di Torino



February 13th - August 11th

Master thesis: 2023

Acknowledgments

I would like to express my sincere gratitude to Melexis DCC team, starting with Paul BARON, who guided me throughout this internship, imparting invaluable knowledge and advice, and showing exceptional kindness towards me.

I would also like to thank the rest of the DCC team for always being available to answer my questions and offer assistance, in particular Ludwig CRON and Michel NOE for their class on CPU architecture, memories and ATPG and Gabriel CRABBE for his very useful course on Gitlab.

I would also like to thank the other interns especially Aziz BADOUI and Martin ARTEAGA for filling these 6 months with unforgettable moments.

Lastly, I want to express my gratitude to everyone at the Melexis Paris site for providing me with the opportunity to undertake this internship and for their warm welcome to the company.

List of Abbreviations

ATPG Automatic Test Pattern Generation

CPU Central Processing Unit

DFT Design For Test

DMA Direct Memory Access

DUT Design Under Test

EEPROM Electrically Erasable Programmable Read-Only Memory

HDL Hardware Description Language

IC Integrated Circuit

IP Intellectual Property

ISA Instruction Set Architecture

LSB Least Significant Bit

MSB Most Significant Bit

PMU Port Management Unit

RAM Random Access Memory

ROM Read Only Memory

RTL Register Transfer Level

Contents

1	Introduction	1
1.1	About Melexis	1
1.2	About the digital work of Melexis - DCC	2
1.3	Internship progress	4
2	Background on CPU architecture and digital flow	5
2.1	Digital flow description	5
2.2	CPU architecture	5
2.2.1	Memory	5
2.2.2	The instruction cycle	7
2.2.3	Bus communication	8
3	Patch controller module	11
3.1	Patches principle	11
3.2	State of the art of patches in the platforms	13
3.2.1	Patches in <i>Ganymede</i> platform	13
3.2.2	Milestones for patches in <i>Callisto</i> platform	15
3.3	Subsequent objectives for the patch module	15
4	Implementation of the patch controller	16
4.1	Self-testable test cases	16
4.2	Wishbone compatible patch controller	17
4.3	Parameterize the patch count	18
4.4	Patch 16-bits and 32-bits instructions	21
4.4.1	Divide patch controller into ports and core	21
4.4.2	Description of the core - 16 and 32 bits patch	21
4.4.3	Patch safety	23
4.5	Patch both CPU and DMA access - Mlx16-EX	23
4.6	Patch multiple bus simultaneously - Mlx16-FX	24
4.7	Future steps	26
5	Conclusion	27
5.1	Conclusion on the patch controller	27
5.2	Personal experience	27

6	ANNEX	28
6.1	List of figures	28
6.2	References	28
6.3	Complementary documentation	29
7	Abstracts	32
8	Archive Card	34

1 Introduction

The target of this 6 months internship is to design a component handling and controlling patches, which are hardware blocks sensitive and important in a CPU architecture. This patch controller will have to be compatible both for the 3 custom 16 bits CPUs platform (*Ganymede* platform^{1,2}) and within the next generation 32 bits platform (*Callisto* platform^{1,2}). The mission I was given was to design, test and integrate a patch controller and to make it usable in combination with the other IPs^{1,2} of the platforms.

1.1 About Melexis

Melexis is a global supplier of micro-electronic semiconductor with a strong focus on automotive and industrial applications. Melexis designs, manufactures, and markets advanced integrated circuits and semiconductor devices that enable smarter, more efficient, and sustainable technologies. Although their main application areas are automotive, this company covers various sensor technologies, drivers, and transceivers to be utilized to power other various applications in the industries they serve, such as electric bikes, health functions or marine applications.

As an intern in the hardware digital team at Melexis Paris - La Defense (figure 1), I had the opportunity to work alongside digital hardware designer engineers and contribute to the development of digital hardware solutions for the sensors developed by Melexis. The hardware digital team plays a crucial role in designing and optimizing digital circuits, ensuring seamless integration with other components, and enabling the efficient execution of complex functionalities.



Figure 1: La Defense, Paris - Melexis DCC

Most of Melexis' turnover concerns automotive sensors. Every new car contains in average 18 Melexis' sensors. The automotive sensors developed are either safety and comfort related or directly implemented in the powertrain of the car [1].

From Tire Pressure Monitoring System (TPMS) to Stop lights and Turn signals and on to

Seat belt buckle sensor, Melexis sensors cover a very wide region of applications as it can be seen in the figure 2.

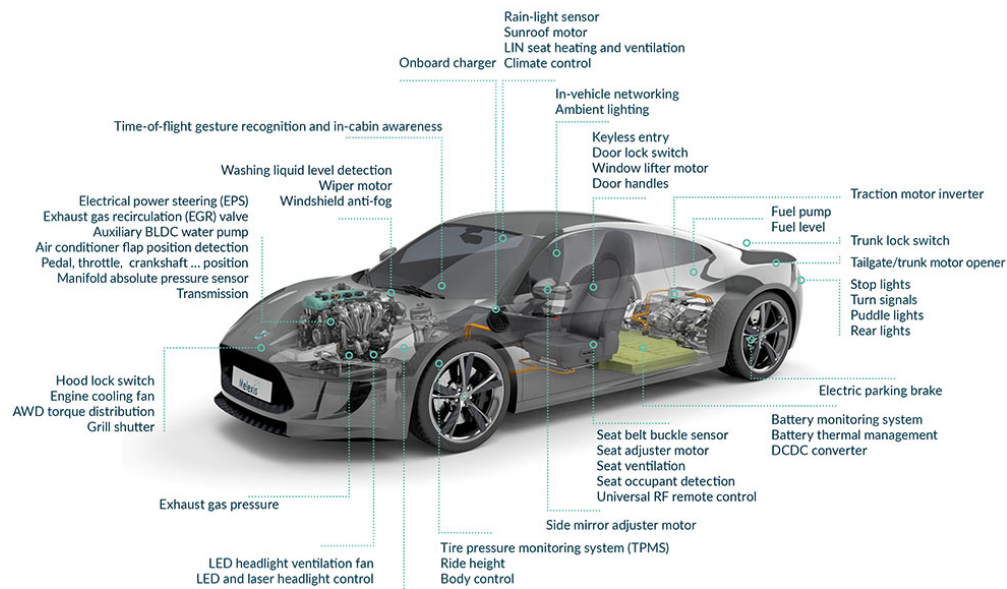


Figure 2: Car applications [1]

Except automobile, Melexis is also present in other markets. Either it is alternative mobility such as electric bikes, smart appliances, smart buildings, robotics or digital health, Melexis is developing sensors and providing sound solutions and achieving breakthroughs in these new domains.



Figure 3: Various sensors developed by Melexis [1]

1.2 About the digital work of Melexis - DCC

The digital team of Melexis (DCC : Digital Competence Center) is based in Paris and Grasse in the south of France. In Paris, it is mainly the front-end engineers while the back-end designers are located in Grasse. Recently, a division was made in the digital team to ease the work flow: one group of digital designers is only dedicated to the existing projects that is to say the one currently used by customers, while another group of designers work on an out-of-context basis named platform. The digital design can be separated in 2 categories.

Project team

The first is project specific design where custom logic is designed for a specific project. For instance the MLX91804 project is a tire pressure sensor used in electric bikes [1]; the project team designed the digital part of the sensor and at the end of the day aims to insure that the integration of the chip by the client is done properly.

Platform team

On the other hand the platform team is similar to R&D in the sense that it aims to anticipate the requests of customers in terms of performances and features for the digital part of the sensors and applications they use. A platform consists of a set of Intellectual Properties (IPs) to be used in different projects. IPs refer to pre-designed and pre-verified functional blocks or components that can be integrated into a larger hardware system or design. It enables a user to quickly generate a top level digital design with the modules required for the application. The objective of a platform structure is to avoid having to redesign the same IPs again to reduce the time to market, and improve IP quality. In simpler terms, a platform allows to receive a CPU and peripherals (such as communication interfaces, ADCs and memories) and create a structure adapted to this proper CPU and the bus protocol it is related to. There are several platforms, each of them having proper characteristics.

- *Ganymede* Platform

The only CPUs integrated into the *Ganymede* platform are custom made CPUs with their own Instruction Set Architectures (ISAs); it can only support a single CPU shell, being custom designed by Melexis and being either an MLX16-E8, a MLX16-EX or MLX16-FX [3].

Platform *Ganymede* is a highly configurable digital hardware allowing to realize a flexible 16 bits micro-controller; it can also be used for CPU-less projects (state machine only) which represents about half of the projects made. Its backbone is a customized Wishbone bus (see Wishbone bus documentation [2]). However, the need for a new, more flexible platform appeared because working with only Melexis custom CPUs (MLX16-E8, a MLX16-EX or MLX16-FX) is restrictive. Indeed, most of Melexis clients want to work with CPUs using standard ISA such as RISC-V. Another motivation to create a new platform using different CPUs is to increase the address space by going from a 16-bits CPU to a 32-bits CPU. The new CPU should also be faster and improve the overall achievable performances of the platform.

- *Callisto* Platform

Thus, Platform *Callisto* was introduced and can support 32-bits CPUs. It is multi-CPU, meaning that it can adapt with different types of CPU depending on the requirement of the customer. More particularly, this platform aims at developing an environment for RISC-V based CPUs and ARM architecture-based systems [3].

1.3 Internship progress

I have included a Gantt diagram to provide a visual representation of the internship's timeline and progress. The Gantt diagram illustrates the various tasks and milestones undertaken throughout the internship duration; these tasks will be furtherly discussed in the following report.

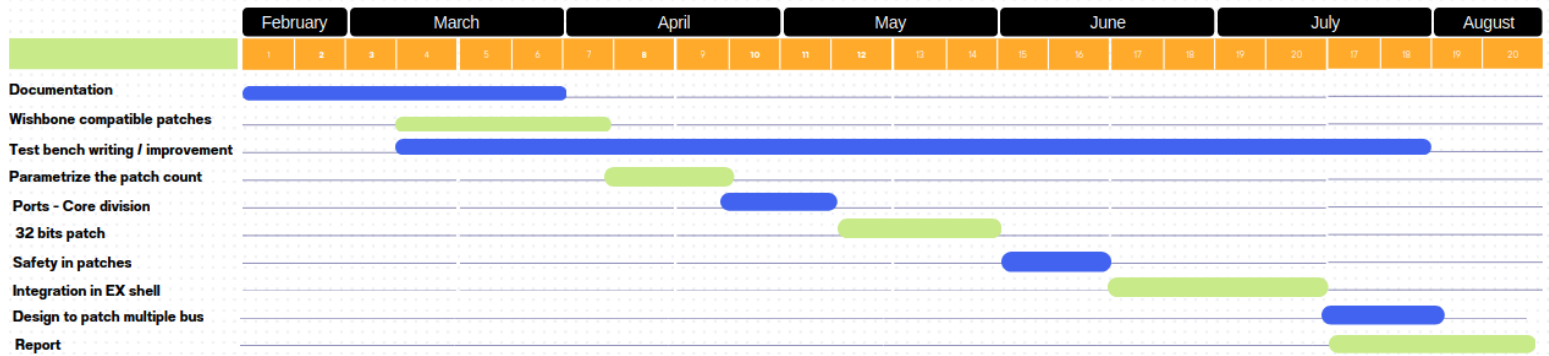


Figure 4: Gantt Diagram

2 Background on CPU architecture and digital flow

In this introductory chapter, we will explore the fundamental concepts related to digital flow in a first part, before discussing CPUs and their architecture, providing a solid background for understanding the specific module I worked on during my internship that is to say the patch controller.

2.1 Digital flow description

In the field of digital hardware design, the digital flow encompasses a series of steps involved in the creation of digital integrated circuits. In this paragraph, I will describe the important points to remember when it comes to the digital flow.

The front-end digital design phase typically begins with specification gathering, where the requirements and functionality of the circuit are defined. The system architect of Melexis is in charge of defining these specifications. This step is then followed by architectural design, where high-level decisions regarding the circuit's structure and functionality are made. The following step involves register transfer level (RTL) coding, where hardware description languages such as VHDL, Verilog or SystemVerilog are used to describe the circuit's behavior at an abstract level; at Melexis, SystemVerilog is used. Once the RTL code is written, it undergoes functional verification through simulation and testbench development to ensure its correctness. After successful verification, synthesis is performed, transforming the RTL code into a gate-level netlist, which represents the circuit using basic logic gates and flip-flops. Melexis is closely working with X-Fab which provides the library for most of the circuits designed by Melexis.

Moving on to the back-end digital design, the gate-level netlist is subjected to physical synthesis, where the netlist is optimized for area, power, and timing. Placement and routing algorithms are employed to determine the optimal positions for each gate and to establish the connections between them. Cadence software is used for this step at Melexis. This step is crucial for achieving proper timing constraints and meeting performance targets. Post-layout simulation is conducted to validate the circuit's functionality and timing after the physical design stage. The final step involves tape-out, where the design data is prepared for fabrication by generating manufacturing files. These files are sent to a semiconductor foundry for the production of the physical integrated circuit.

2.2 CPU architecture

In the realm of modern computing, the central processing unit (CPU) serves as the brain of a computer system. It is responsible for executing instructions, performing calculations, and managing data flow. To accomplish these tasks, CPUs employ a complex architecture consisting of various components and communication channels.

2.2.1 Memory

At the core of any CPU architecture lies the concept of memory. Computers rely on memory to store and retrieve data and instructions for processing.

One crucial component of memory is the Read-Only Memory (ROM). ROM is a non-volatile storage that contains instructions that are permanently written during manufacturing. The instructions stored in ROM serve as a foundation for the CPU's operation. Constants can also be stored in ROM (which can also be patched). Melexis Software team is in charge of writing the code and instructions that will be the basis for the CPU actions.

On the other hand, Random-Access Memory (RAM) is volatile; it allows fast Read and Write operations (see part 2.2.2) but requires a continuous power supply to retain data. In practice, RAM contains data to be used in all CPU's computations.

EEPROM (Electrically Erasable Programmable Read-Only Memory) is non-volatile and is suitable for applications where data persistence is crucial. The main difference between EEPROM and ROM is that one can write in EEPROM while it is impossible for ROM. EEPROM is usually useful to store calibration values or customer configuration. The fact that EEPROM is a non-volatile writable memory is used to store the addresses to patch and the patch instruction/sequence (see part. 3.1).

Ultimately, registers serve as temporary storage unit for critical data during processing. One must mention that at Melexis, peripheral registers (or memory-mapped registers) are referred as ports while CPU registers are still referred as registers, so from now on, when referring to peripheral registers we will instead use the term ports. The PMU (Port Management Unit) is the unit in charge of handling the ports.

In terms of dimensions, a ROM is smaller than a EEPROM and registers are much bigger than RAM which is of a medium size. These dimensions considerations explain architectural choices in terms of memory in a design.

The memory space is divided in 3 areas: the peripheral area, the variable area and the program area as described in figure 5. The address of input accesses is decoded with respect to this mapping. Then the memory arbiter latches an access onto one of the three output buses (program, peripheral or variable). In the most simple case, the memories can be connected directly on these 3 buses, the code memory (ROM or Flash for instance) on the program bus, the RAM on the variable bus and both the PMU and the EEPROM on the peripheral bus while ports of all peripherals are connected to the Wishbone bus PER_BUS through the PMU (see figure 8).

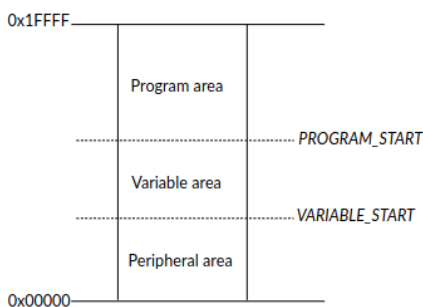


Figure 5: Basic principle of memory mapping

In practice, CPU is not the only one to have access to memory; it is relevant now to

introduce what a DMA is since it will be used in the following parts of the report (in particular in part 4.5 on DMA access).

DMA stands for Direct Memory Access. In simple terms, DMA's aim is to efficiently transfer data between devices without involving the main processor in every step of the process. The DMA is for example involved when copying one memory part in another memory location; the DMA controller directly moves the data between these locations in RAM without involving the CPU in every data transfer operation. This leaves the CPU free to handle other tasks, making the data transfer faster and more efficient. So this master DMA has a direct access to the memory as its name suggests.

2.2.2 The instruction cycle

The CPU operates by following a series of steps, commonly referred to as the instruction cycle. This cycle begins with the fetching of instructions from memory and more particularly from the ROM. The CPU retrieves the instructions one at a time and stores them in a temporary storage called the instruction register. The process is known as instruction fetching. The CPU then proceeds to decode the fetched instructions, determining the necessary actions to be performed.

One fundamental instruction that the CPU performs is the Read instruction. The Read instruction is essential for accessing data from memory, which enables the CPU to access and manipulate information required for executing programs and performing various tasks. In this section, we will give details on the Read instruction, explaining its purpose, steps, and significance since this operation will be used in the testbench written in part 4.1. The primary purpose of the Read instruction is to retrieve data from the main memory and transfer it to the CPU for further processing. This data can include program instructions, variables, constants, or any other relevant information necessary for the program's execution. Reading data from memory is crucial because it enables the CPU to access the necessary information to perform calculations, make decisions, and produce the desired output. This Read operation involves several steps that must be described. First, the instruction fetch: the CPU fetches the Read instruction from the current program's memory address. This instruction informs the CPU that it needs to Read data from a specific memory location. Then the CPU calculates the memory address from which it needs to Read data. The CPU initiates a memory access cycle to retrieve the data from the calculated memory address and once the data is accessed, it is transferred to the CPU's internal registers. After the data is successfully transferred to the CPU, the processor can proceed to execute further instructions. Depending on the following instructions, the CPU may perform arithmetic, logical, or control operations.

During the internship I focused particularly on the EX and FX CPUs and it is useful to provide some context to understand the following report. Mlx16-EX and Mlx16-FX are two CPUs used in the *Ganymede* platform and are developed by Melexis. The main difference between the EX and FX is that the FX is a pipelined CPU. A pipelined CPU is a crucial architectural design that optimizes instruction execution by dividing it into multiple stages. This allows the CPU to process multiple instructions concurrently, leading to increased throughput and improved performance. However, while pipelining offers substantial benefits, it also introduces challenges related to hazards and dependency handling.

2.2.3 Bus communication

To facilitate communication between various components of a computer system, including the CPU, memory, and input/output devices, a system bus is utilized. A bus acts as a communication pathway, enabling the transfer of data and instructions between different parts of the system. It serves as a vital link, allowing the CPU to access memory, transfer data between registers, and interact with other hardware components. Once the CPU has fetched an instruction from memory and decoded it, it moves on to the execution phase. During this phase, the CPU performs the specific operations dictated by the instruction. This might involve mathematical calculations, logical operations, or data manipulation. The execution phase is where the true processing and computation occur, utilizing the CPU's arithmetic logic unit (ALU) and other functional units.

Having this definition in mind, it is now easier to understand the crossbar arbiter present in the shell of the FX as can be seen in figure 12. This arbiter dispatches the accesses coming from the two Mlx16-FX bus: Code and Data, to the memory buses: per, var and prog covering the whole addressing space. Code bus is reserved to fetch instructions in memory which can be either in ROM or RAM while Data bus only access data that can also be either in ROM or RAM. This is the crossbar arbiter: two accesses can be done simultaneously if they use a different memory bus.

Wishbone bus

Wishbone bus is commonly used in *Ganymede* platform and a description of the behavior of the relevant signals and how they are related to each other is useful for the well understanding of the following parts [2].

- ADDRESS signal

The ADDRESS bus is used to specify the memory or port address for the transaction. It typically carries the address of the data being accessed. In Ganymede platform, the ADDRESS signal is 20 bits wide.

- WIDTH signal

If this signal is 0, the slave return 16 bits, if it is 1 slave returns 32 bits

- RDATA signal

The data bus used for data transfer between master and slave. It is from slave to master. For instance, as we can see in figure 21, RDATA is an output for the slave memory and an input for the master CPU. This signal carries the data being processed. A master can request either a Byte or a Word access to a particular slave and the outcome of this access passes through RDATA signal. RDATA can basically be either 16 or 32 bits, and depending on its size the data is arranged in different ways as we can see in figure 6. On the left, a slave sets WIDTH=0 when it sends back only 16 bits and in this case it must arrange its data as shown on the figure. A slave sets WIDTH=1 when it sends back 32 bits and in this case it must arrange its data as shown on the right figure.

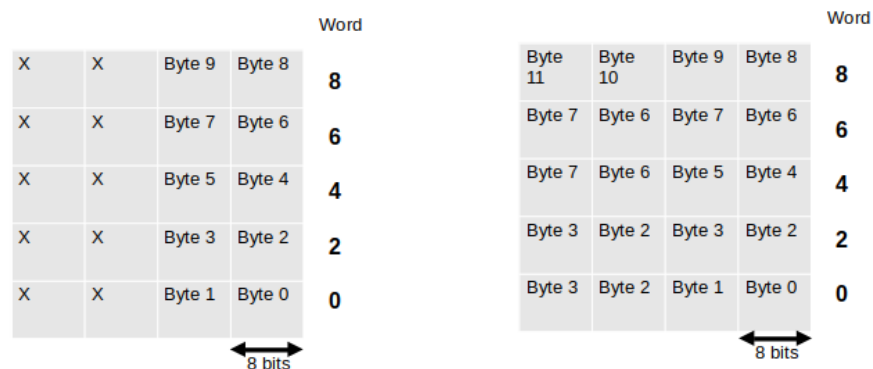


Figure 6: RDATA disposition 16 bits (left) and 32 bits (right)

- RPARITY signal

Safety related signal, it gives information on RDATA Parity. RPARITY signal is constructed with a defined table as it can be seen in table 1.

- WDATA signal

16 bits wide signal that contains data to be written in a slave. It is RDATA complementary signal as it is from master to slave.

IO bus

IO bus is the bus protocol used when communicating with ports. It relies on signals that are similar to the one described for the Wishbone with some particularities though. It is essential to present the structure of a port before actually describing the IO bus.

A port is basically a 16-bits register (as mentioned in part on memory 2.2.1) with an optional bit containing the parity of what is stored in the port (see figure 7).

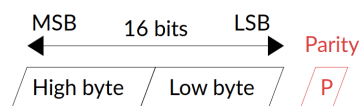


Figure 7: Port description

The port related inputs are:

- io_clk : Port clock
- io_init : Synchronous reset
- io_write : Write
- io_read : Read
- io_wdata : Write Data
- io_wparity : Parity of Write Data

- io_access : Selection of the port addressed

The port related outputs are:

- io_rdata : Read Data
- io_rparity : Parity of Read Data

3 Patch controller module

This chapter of the thesis aims to provide a comprehensive understanding of what is a patch, its underlying principles, and its implications for the design and performance of CPUs. A deeper description of the current module of patches in the different platforms will be given.

3.1 Patches principle

The patch controller acts as a critical component within the CPU's architecture, responsible for fixing erroneous instructions loaded into the ROM, without rewriting the code. A single patch consists of an address port and an instruction port. What happens in practice is that a synthesized circuit may not work as predicted and errors occur. These errors are software bugs and the detection and resolution of errors are crucial to ensure the production of high-quality ROM chips. These errors are ideally detected by Melexis software team before selling the circuit to the client, but if not, the chip is sent back to us and it is our duty to fix the error. We can obviously not modify what is written in the ROM so, ultimately, the role of the patch controller is to corrupt what is written in memory, because what is in memory is different from what was expected and thus leads to errors.

When errors in memory are defined and located by the software team, the addresses and correct instructions are loaded in the EEPROM (see part on memory 2.2.1). A space of the EEPROM is allocated to writing the addresses to be patched and the corrected instructions. The CPU can now retrieve the configuration of the patches in the EEPROM and load the ports of the patches correspondingly at circuit startup (red arrow in figure 8).

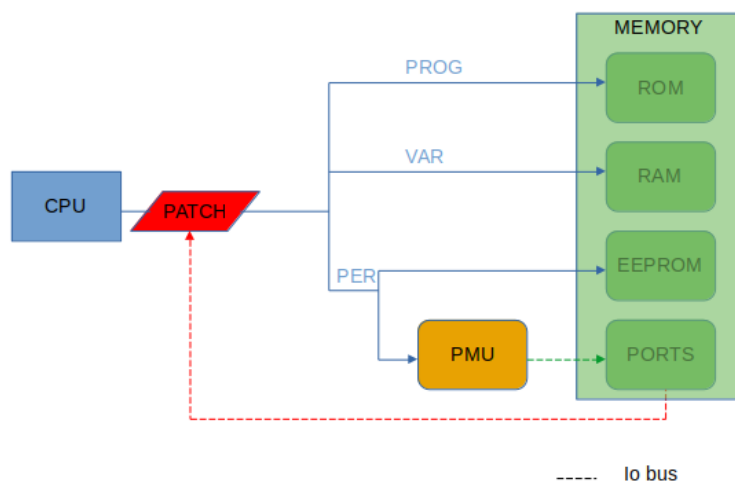


Figure 8: Basic principle of patches

There is predefined amount of address and instruction ports (working in pair) that can fix a limited amount of errors in the code. In figure 9, three ports are represented. The patch controller replaces the faulty instruction of the memory by the instruction in the instruction port. This enables the CPU to correct and modify the behavior of the program in real-time, enhancing its reliability and functionality.

So, the patch controller, a module within the CPU, plays a pivotal role in rectifying incorrect instructions of the memory by substituting them with user-provided instructions, at first written in the EEPROM, and transferred afterwards within the instruction ports of the patch.

The patch module is thus between the CPU and the memory, as seen in figure 9. When the CPU fetches at the address where the error was found, the patch replaces the erroneous instruction by the one in its instruction port.

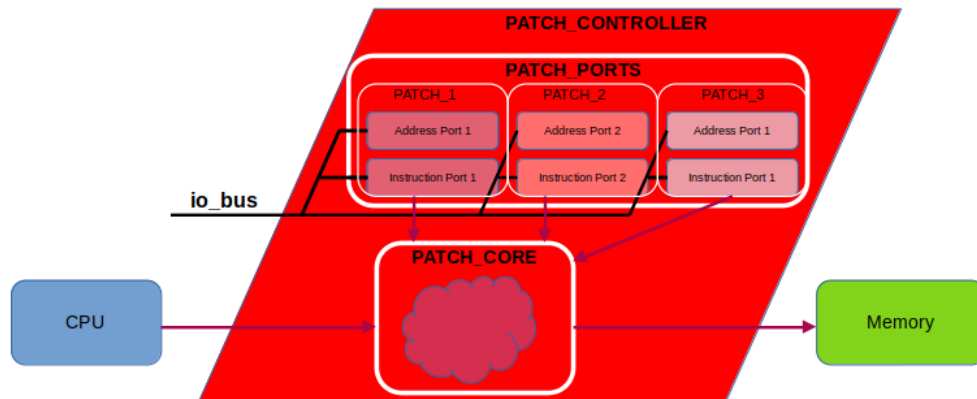


Figure 9: Patch simple description

3.2 State of the art of patches in the platforms

3.2.1 Patches in *Ganymede* platform

Patches in MLX16-EX shell

The patches have two 16-bits ports for each patch; one port for the address where the instruction has to be replaced, one for the instruction itself. The Mlx16-EX shell provides up to 4 patches and the complete shell is represented in figure 10.

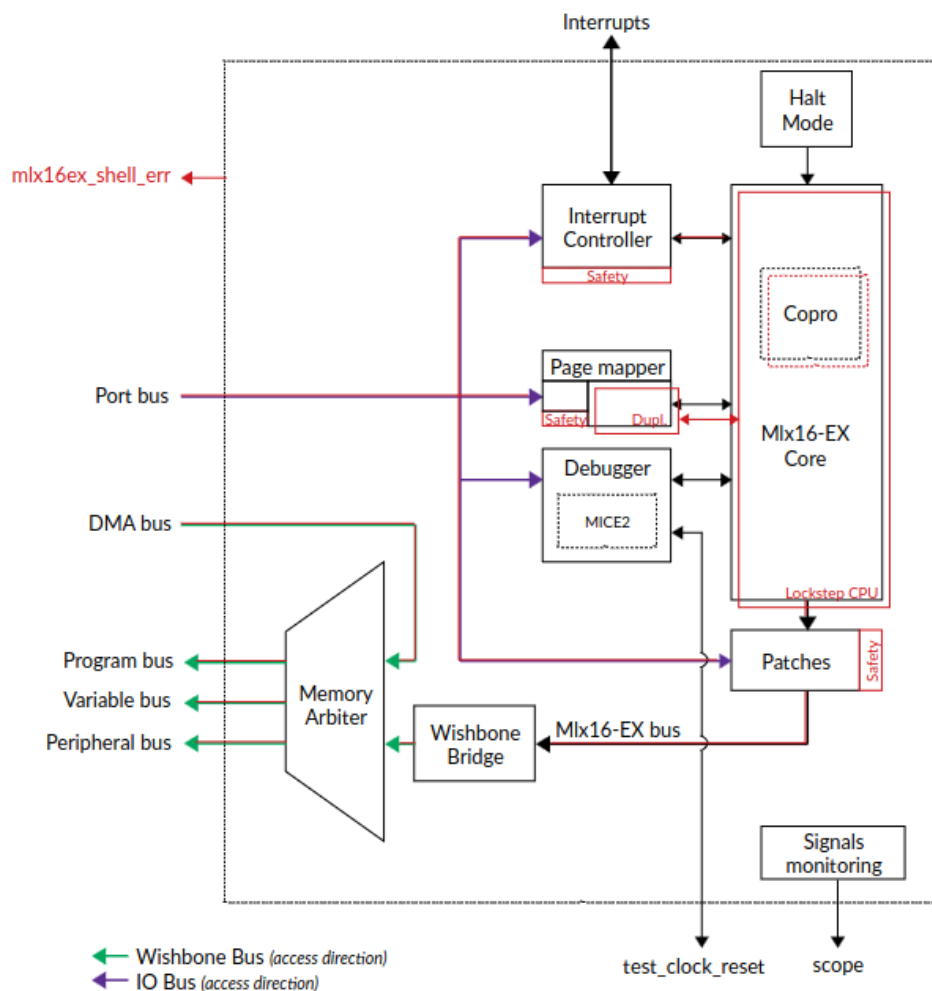


Figure 10: Patches in Mlx16-EX shell

The patches are enabled provided the 16-bit Word address is not 0x0000. The 128 KB of program space can be patched. When the CPU fetches at the address which is defined within the patch, it is the patch instruction that is latched on the CPU bus. If the fix is a single instruction, the patch instruction port is filled with that fixed instruction. Otherwise it must be filled with a jump instruction so that the program can jump to a part of the RAM

where the fix is stored. All the patches address and instruction ports can implement a parity mechanism (see part related on safety 4.4.3).

Figure 11 shows a simplified view of the complete EX-shell; it highlights what is relevant for our work on the patch controller module. The patch block is currently communicating through the Mlx16-EX CPU custom bus and the Wishbone bridge's goal is to convert this custom bus protocol to Wishbone bus protocol used by the memories and the DMA.

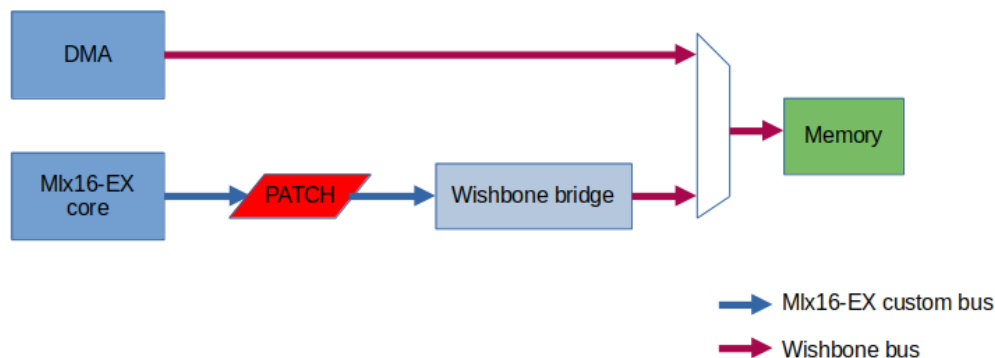


Figure 11: Simplified view of Mlx16-EX shell

Patches in MLX16-FX shell

When it comes to the FX shell, there is no clear module of patches as the same Verilog code is shared between breakpoints and patches within the debugger module (see figure 12). The debugger can be either in breakpoints mode or in patch mode, thus the need for a proper patch module. Currently for the FX, only the code bus is patched when the debugger is in patch mode.

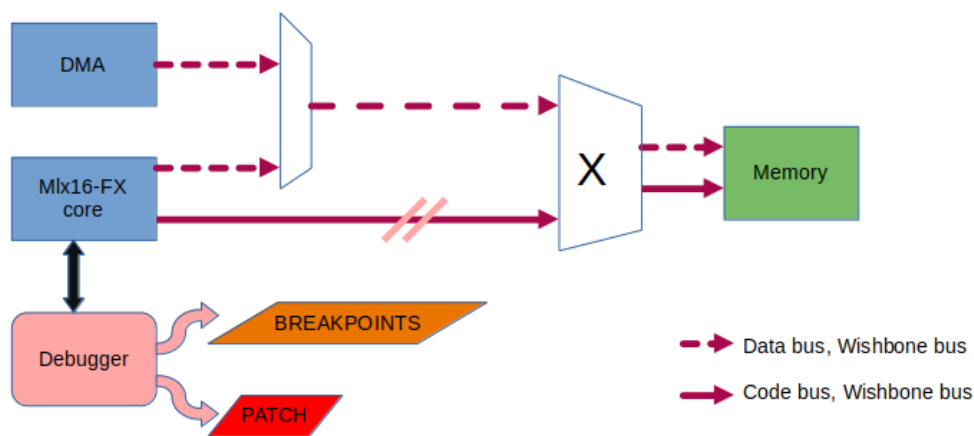


Figure 12: FX-shell

The figure 13 is a simplified view of the shell of the FX when the debugger is in patch mode.

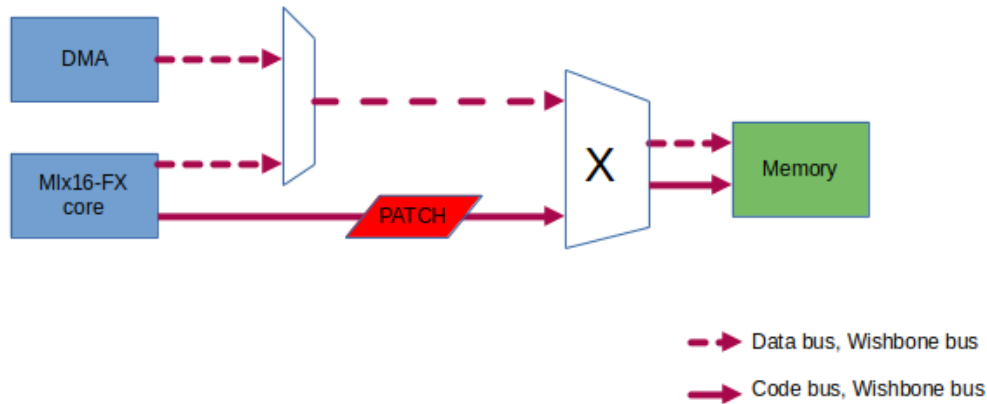


Figure 13: FX-shell simplified when debugger is in patch mode

3.2.2 Milestones for patches in *Callisto* platform

As for the *Callisto* platform, no patch controller exists yet. The patches should be compatible with AHB bus when an ARM CPU is used in the platform and Ibex bus in the case of a RISC-V CPU (see part on bus communication [2.2.3](#)).

3.3 Subsequent objectives for the patch module

The objective is thus to design and verify a patch controller module. This module has to be flexible enough to be easily integrated in both platforms. More freedom must be provided to the patches already existing. To be more precise, the hardware for the patches in *Ganymede* platform must be reworked to design a standardize module that can be reused both for EX and FX shell and create a clear distinction between breakpoints and patches for the FX. The patch controller module must be compatible with the *Ganymede*'s Wishbone bus, since this is the bus protocol supported by the platform. By controlling and parametrizing the patch count and the address and data widths, we could add a degree of freedom on the design, which is at the end of the day one of the objective of the internship. The hardware design must allow to patch not only CPU access but also DMA access, both for EX and FX. As mentioned in part [1.2](#), FX has a crossbar structure, thus, patch hardware should patch on both Wishbone Data and Code buses meaning that we have to support patching multiple bus simultaneously (1 and 2 typically). When it comes to the module of patch in the *Callisto* platform, the hardware should allow to be compatible with the AHB 32-bits bus. And finally, as *Callisto* platform is multi-CPU, the design provided should be standardize for both RISC-V and ARM.

On the whole, a piece of hardware design is to be written to fulfil all the above requirements and to be reused in any platform. Not only must the patch controller design be functionally tested, it also must be synthetisable and the testbench written have to fully cover the design. A formal proof of the correct behavior of the design is expected.

4 Implementation of the patch controller

4.1 Self-testable test cases

The first step of the internship was to write a set of self-testable test cases, easily reusable and that can be adapted to any new design of the patches. The aim of this newly created testbench is to avoid checking waves as much as we could and achieve functional testing only by looking at the terminal; this would save time and allow automatized non-regression test. The testbench that is used in the following parts of the internship is divided in several parts to test the patches in every conditions they will be placed in practice. Different colors in figure 14 show different steps that will be described subsequently.

```
Note : (time 943510 NS) Initialize address and instruction ports -----
Note : (time 943510 NS) [port_master_if] Writing Word[0] = 0x0054
Note : (time 943550 NS) [port_master_if] Writing Word[2] = 0x1ace
Note : (time 943590 NS) [port_master_if] Writing Word[4] = 0x0031
Note : (time 943630 NS) [port_master_if] Writing Word[6] = 0x43d8
Note : (time 943670 NS) [port_master_if] Writing Word[8] = 0x004a
Note : (time 943710 NS) [port_master_if] Writing Word[10] = 0xda36
Note : (time 943750 NS) [port_master_if] Writing Word[12] = 0x005f
Note : (time 943790 NS) [port_master_if] Writing Word[14] = 0x9ac2
Note : (time 943830 NS) Beginning of the controlled part of the tb -- Word access
Note : (time 943830 NS) !! patch number 1 is active !!
Note : (time 943850 NS) [bus_master] READ [0x000a8]=0x1ace
Note : (time 943870 NS) !! patch number 2 is active !!
Note : (time 943890 NS) [bus_master] READ [0x00062]=0x43d8
Note : (time 943910 NS) !! patch number 3 is active !!
Note : (time 943930 NS) [bus_master] READ [0x00094]=0xda36
Note : (time 943950 NS) !! patch number 4 is active !!
Note : (time 943970 NS) [bus_master] READ [0x000be]=0x9ac2
Note : (time 943990 NS) Beginning of the random part of the tb -- Word access
Note : (time 943990 NS) !! patch number 1 is active !!
Note : (time 944010 NS) [bus_master] READ [0x000a8]=0x1ace
Note : (time 944030 NS) !! patch number 3 is active !!
Note : (time 944030 NS) [bus_master] READ [0x0003c]=0x1af0
Note : (time 944050 NS) !! patch number 3 is active !!
Note : (time 944050 NS) [bus_master] READ [0x00094]=0xda36
Note : (time 944070 NS) [bus_master] READ [0x00094]=0xda36
Note : (time 944090 NS) [bus_master] READ [0x00002]=0xee28
Note : (time 944110 NS) [bus_master] READ [0x00022]=0x2256
Note : (time 944130 NS) [bus_master] READ [0x00024]=0x418d
Note : (time 944150 NS) Beginning of the controlled part of the tb -- Byte access
Note : (time 944150 NS) !! patch number 1 is active !!
Note : (time 944150 NS) [bus_master] READ [0x00048]=0x4613
Note : (time 944170 NS) !! patch number 1 is active !!
Note : (time 944170 NS) [bus_master] READ [0x000a8]=0x1ace
Note : (time 944190 NS) !! patch number 2 is active !!
Note : (time 944190 NS) [bus_master] READ [0x000a9]=0x1ace
Note : (time 944210 NS) !! patch number 2 is active !!
Note : (time 944210 NS) [bus_master] READ [0x00062]=0x43d8
Note : (time 944230 NS) !! patch number 3 is active !!
Note : (time 944230 NS) [bus_master] READ [0x00063]=0x43d8
Note : (time 944250 NS) !! patch number 3 is active !!
Note : (time 944250 NS) [bus_master] READ [0x00094]=0xda36
Note : (time 944270 NS) !! patch number 4 is active !!
Note : (time 944270 NS) [bus_master] READ [0x00095]=0xda36
Note : (time 944290 NS) !! patch number 4 is active !!
Note : (time 944290 NS) [bus_master] READ [0x000be]=0x9ac2
Note : (time 944310 NS) Beginning of the random part of the tb -- Byte access
Note : (time 944310 NS) [bus_master] READ [0x000bf]=0x9ac2
Note : (time 944330 NS) [bus_master] READ [0x00038]=0x6f74
Note : (time 944350 NS) [bus_master] READ [0x0005f]=0xa438
Note : (time 944370 NS) [bus_master] READ [0x00092]=0xff4f
Note : (time 944390 NS) [bus_master] READ [0x00056]=0xd86c
```

Figure 14: Terminal output of the newly created testbench

The first thing to do is to initialize all the ports of the patches(yellow part - first part); in our case with the patch block used in the EX-shell where only a maximum of 4 patches can be instantiated it means initialize 8 ports.

Then, we know from the part on Byte and Word access (2.2.3) that the CPU can do either Byte or Word access. The following step of the functional test is to perform a ReadWord access at all the addresses where patch is supposed to be active (red part - second part). To do so, one very important point of the testbench is to clearly identify at what addresses we want to patch (these addresses will be discussed in the part 4.4.2).

After reading at the addresses to patch, we generate random addresses and read the Word

at this address (white part - third part). If by any chance the randomly generated address happens to be one address to patch, then the instruction Read is the one in the instruction port of the patch; otherwise we should Read the value in memory at this particular address. We repeat the random Read-Word access several times and then follow along with the next part of the testbench, which consists of a Byte access at the addresses to patch (green part - fourth part).

Finally, similarly to what was done with the random Word Read access, we perform several random Byte Read access (blue part - last part). This core testbench structure was followed for every design that had to be tested but slight modifications were made for particular designs (in particular for the part on DMA and CPU patch 4.5).

The relevant signals used during the simulations are shown on figure 15. These particular waveforms call for Wishbone bus knowledge described in part 2.2.3 and are relative to the red step (second step) of the testbench described in figure 14; it consists of ReadWord accesses at all the addresses where patch is supposed to be active.

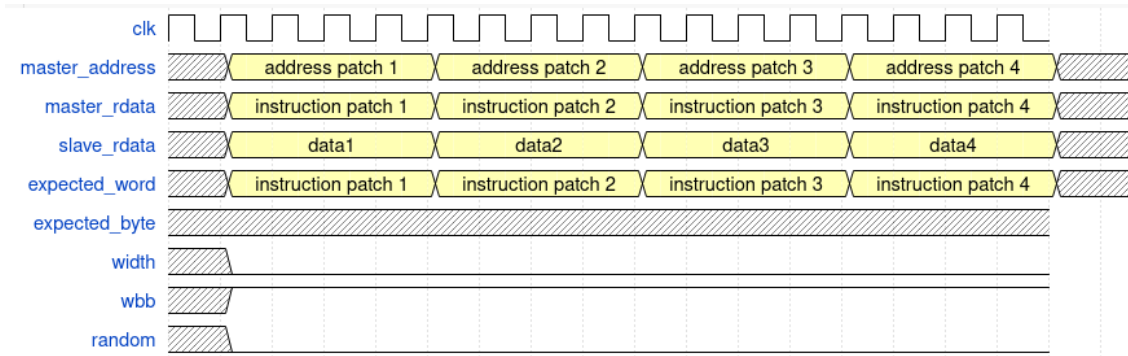


Figure 15: Waves of the ReadWord on addresses to patch for a 16 bits slave

The waveforms relative to all the steps described in this part describing the testbench core structure can be found in annex.

4.2 Wishbone compatible patch controller

The current patch module integrated in the EX-shell is only compatible with the custom bus of the Mlx16-EX and the patch is integrated as it is shown on figure 11. The fact that it is only compatible with this particular bus is something that has to be changed. Indeed, the objective is to provide a module that is the most flexible possible and the first step to achieve is to make the block of patches Wishbone compatible. By doing so, the module can be moved wherever the Wishbone bus is used, and ultimately the shell of the EX is modified as it is shown on figure 16.

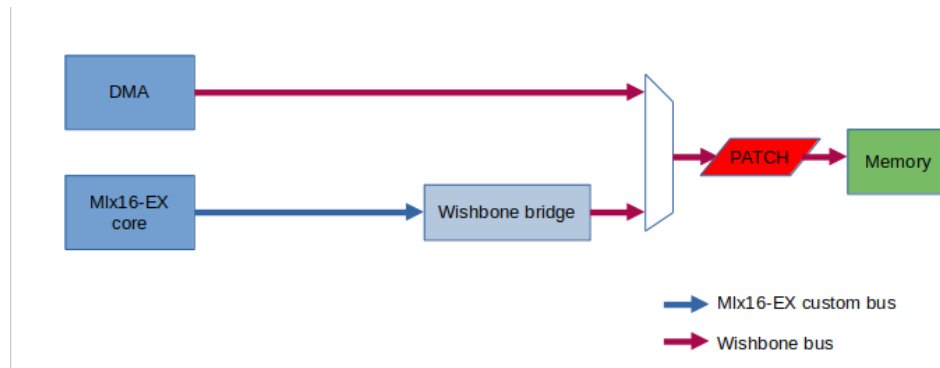


Figure 16: EX-shell with Wishbone compatible patches

The newly designed patch controller is tested by itself; it means that only the simple patch block was instantiated in the testbench and tested with the procedure described in part 4.1. After having verified the functional behavior of the patch controller, it is necessary to perform the code coverage of the testbench. The purpose of code coverage in hardware design is to indicate the percentage of code that has been exercised during testing. Higher code coverage suggests that more parts of the design have been tested and, thus, the design is more thoroughly verified. Code coverage is also used to validate the quality and effectiveness of the testbenches used in the verification process. It ensures that the testbench is capable of driving the design under test with sufficient stimulus to achieve good coverage. Toggle coverage tracks the activity of signals within the design to ensure that all parts of the circuit are being exercised and that, for instance, we verify every if and else conditions at least once. To validate a design and its corresponding testbench, a lint must be performed in addition to the code coverage previously discussed. Hardware design uses linting tools to ensure the correctness, consistency, and best practices adherence in hardware designs; we verify that our design is synthesizable.

4.3 Parameterize the patch count

The current patch module of the EX-shell (see figure 11) exhibits limitations. One of them concerns the maximum number of patches it can instantiate, this maximum being four. That means that with the patches currently at disposal we can fix only a maximum of 4 errors in the code. More than just fixing the code as previously discussed, the patch controller can also be used to tune the code after manufacturing, so these patches can provide more freedom in any case: this maximum number of patches is a serious restriction. The code describing the ports in the patch is very long and consists in instantiating several times the same port. Figure 17 shows the pair of address and instruction ports relative to the second patch (patch2); this code is repeated four times and is not efficient; it can be simplified.

```

if(PATCHES > 2) begin : patch2
    port #(
        .RW_BITS      (16'hffff >> (16-(CPU_ADD_WIDTH-1))),
        .RESET_VALUE  (16'h0000),
        .SAFETY_LEVEL (PORTS_PARITY),
        .TEST_PORT    (0)
    ) port_patch_addr2 (
        //outputs
        .q      (patch_addr2),
        .p_q    (patch_addr2_p_q),
        .rdata   (patch_addr2_io_rdata),
        .rparity (patch_addr2_io_rparity),
        .rwl_invalid (0),
        .key_en  (0),
        .w0l_en  (0),
        //inputs
        .io_clk   (patch_io_clk),
        .mcu_clk  (mcu_clk),
        .access   (patch_io_access[4]),
        .io_write (patch_io_write),
        .io_init  (patch_io_init),
        .io_wdata (patch_io_wdata),
        .io_wparity (patch_io_wparity),
        .ro_data_in (1'bx),
        .ro_parity_in (1'bx),
        .rwe_data_in (1'bx),
        .rwe_parity_in (1'bx),
        .rwe_en (1'bx),
        .rwlc_set (1'bx),
        .rwlc_clear (1'bx),
        .rwls_clear (1'bx),
        .rwls_set (1'bx),
        .rwl_unlock (1'bx),
        .async_rstb (1'b1),
        .io_read (1'bx)
    );
end

port #(
    .RW_BITS      (16'hffff),
    .RESET_VALUE  (16'h0000),
    .SAFETY_LEVEL (WISHBONE_BUS_SAFETY),
    .TEST_PORT    (0)
) port_patch_instr2 (
    //outputs
    .q      (patch_instr2),
    .p_q    (patch_instr2_p_q),
    .rdata   (patch_instr2_io_rdata),
    .rparity (patch_instr2_io_rparity),
    .rwl_invalid (0),
    .key_en  (0),
    .w0l_en  (0),
    //inputs
    .io_clk   (patch_io_clk),
    .mcu_clk  (mcu_clk),
    .access   (patch_io_access[5]),
    .io_write (patch_io_write),
    .io_init  (patch_io_init),
    .io_wdata (patch_io_wdata),
    .io_wparity (patch_io_wparity),
    .ro_data_in (1'bx),
    .ro_parity_in (1'bx),
    .rwe_data_in (1'bx),
    .rwe_parity_in (1'bx),
    .rwe_en (1'bx),
    .rwlc_set (1'bx),
    .rwlc_clear (1'bx),
    .rwls_clear (1'bx),
    .rwls_set (1'bx),
    .rwl_unlock (1'bx),
    .async_rstb (1'b1),
    .io_read (1'bx)
);

```

Figure 17: Ports instantiating in patches of the EX-shell

The objective is thus to parameterize the patch count to provide more flexibility to the module. We introduce the parameter `PATCHES` and clean up the code by using a for loop. This for loop generation implies the use of an interesting feature of SystemVerilog which is array port connection inside the generate loop instantiation. The code is on the whole simplified and cleaned-up and consists only on the generate block seen on figure 18, both for the address and for the instruction ports.


```

generate
genvar i;
for (i=0;i<PATCHES;i=i+1)
begin
port #(
    .RW_BITS      (16'hFFFF),
    .RESET_VALUE  (16'hFFFF),
    .SAFETY_LEVEL (SAFETY_LEVEL),
    .TEST_PORT    (0)
)
address_port(
    // Outputs
    .p_q      (patch_addr[i]),
    .p_q      (patch_addr_p_q[i]),
    .rdata     (patch_addr_io_rdata[i]),
    .rparity   (patch_addr_io_rparity[i]),
    .rwl_invalid(),
    .key_en    (),
    .w0l_en    (),
    // Inputs
    .io_clk     (patch_io_clk),
    .mcu_clk    (mcu_clk),
    .access     (patch_io_access[2*i]),
    .io_write   (patch_io_write),

    .io_init    (patch_io_init),
    .io_wdata   (patch_io_wdata),
    .io_wparity (patch_io_wparity),
    .ro_data_in (1'bX),
    .ro_parity_in (1'bX),
    .rwe_data_in (1'bX),
    .rwe_parity_in (1'bX),
    .rwe_en     (1'bX),
    .rwlc_set   (1'bX),
    .rwlc_clear (1'bX),
    .rwlsl_set  (1'bX),
    .rwlsl_clear (1'bX),
    .rwl_unlock (1'bX),
    .async_rstb (1'b1),
    .io_read    (1'bX)
);
end

```

Figure 18: New ports instantiating in EX-shell

To test the newly designed patches, the testbench follows the one described in part 4.1 and the different steps of the test procedure are highlighted with the same colors as the one described previously. The parametrization on the patch count is thus realised and can be seen in figure 19 where we have seven patches instantiated.

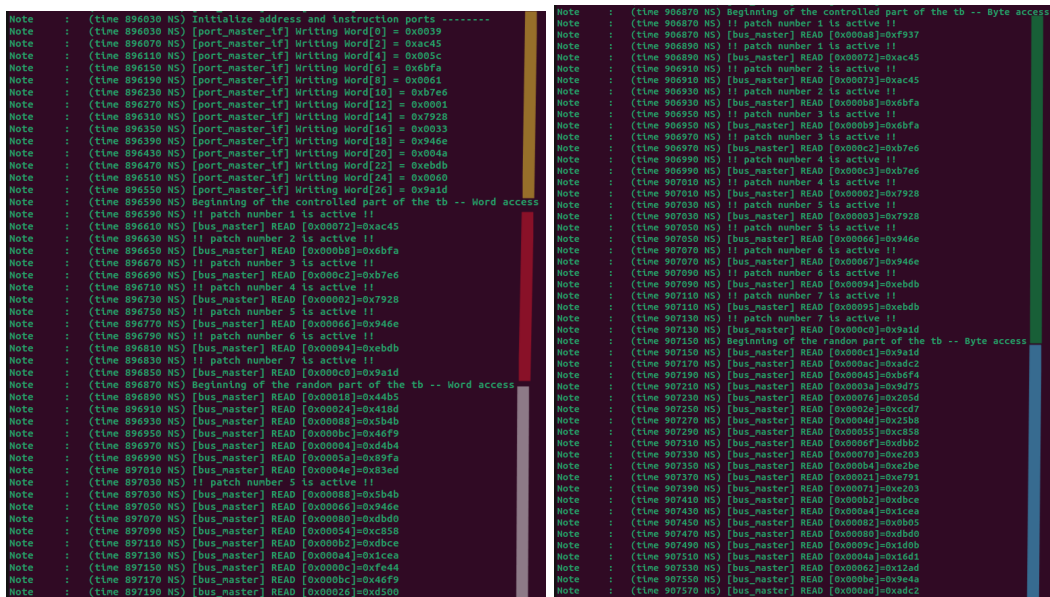


Figure 19: Testbench with 7 patches

4.4 Patch 16-bits and 32-bits instructions

4.4.1 Divide patch controller into ports and core

To prepare for the integration of the patch block in *Callisto* platform (see part on *Callisto* platform [1.2](#)) and to the fact that the ports won't be 16 bits anymore, it is useful to separate the ports of the patch to the core of the logic itself. The union of patch_ports and patch_core constitute on itself what the patch controller is (see figure 20).

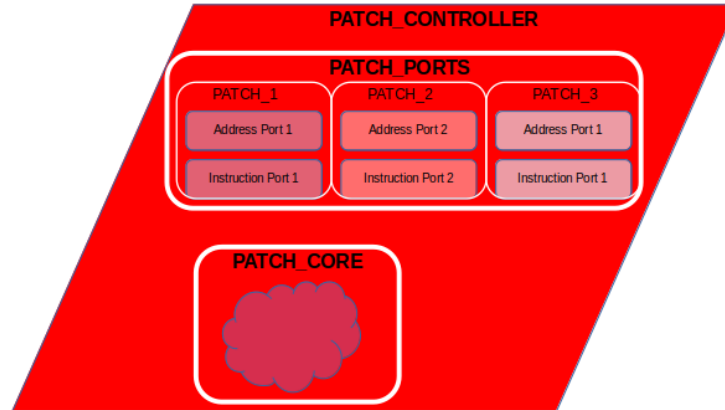


Figure 20: Separation core and ports

4.4.2 Description of the core - 16 and 32 bits patch

- 16-bits patch

For the Mlx16-EX, the ADDRESS signal is 17 bits wide. The address loaded in the address port of the patch is a 16 bits address. Thus we can't simply compare the ADDRESS signal getting out the CPU to the address in the patch controller address port. Instead, the logic implemented in the core of the patch controller consists in comparing the 16 MSB of the CPU ADDRESS signal (CPU_ADDRESS[16:1]) to the address loaded in the address port of the patch. If these two addresses are equal, we call it a match and we should actually corrupt what is at this address in memory and patch. For this case when slave RDATA signal is 16 bits there is no caution to take because the instruction port is 16 bits too: when there is a match, we replace the entire RDATA by the instruction loaded in the instruction port of the patch. It is different when the memory RDATA signal is 32 bits.

- 32-bits patch

When memory RDATA signal is 32 bits, a deeper comparison must be achieved to know when and where to patch. We compare the 15 MSB of the CPU ADDRESS signal which is CPU_ADDRESS[16:2] to the 15 MSB of the port address in the patch_ports meaning PATCH_ADDRESS[15:1]. If a match occur, we now have to patch. As mentioned earlier (part on io bus and port description [2.2.3](#)), the instruction port is only 16 bits while RDATA signal returning to the CPU must be 32 bits. So, one part of this RDATA is coming from

the instruction port of the patch, the other 16 bits is formed by the complementary memory RDATA signal. The last bit of the port address of the patch will give us whether the instruction should be the 16 MSB of the CPU RDATA signal (CPU_RDATA[31:16]) or the 16 LSB (CPU_RDATA[15:0]). To be more precise, in the case where parity is not considered, if the LSB of the address port of the patch is 1, then CPU_RDATA, meaning the "corrupted" RDATA, is constructed with its 16 MSB being the instruction in the patch, and the 16 LSB being the 16 LSB of memory RDATA. On the contrary, when the LSB of the address port of the patch is 0, CPU_RDATA is constructed the other way around.

Another precaution to take when it comes to 32 bits memory is the fact that when the CPU ADDRESS signal is either 2, 6, 10 and so on (that is to say when CPU_ADDRESS[1]=1) the disposition of the memory RDATA signal is particular and must be taken care of in the case of a match at this address (see the disposition of RDATA signal in part 2.2.3).

All the previous considerations regarding 16 and 32 bits RDATA signals are gathered in the block diagram on figure 21.

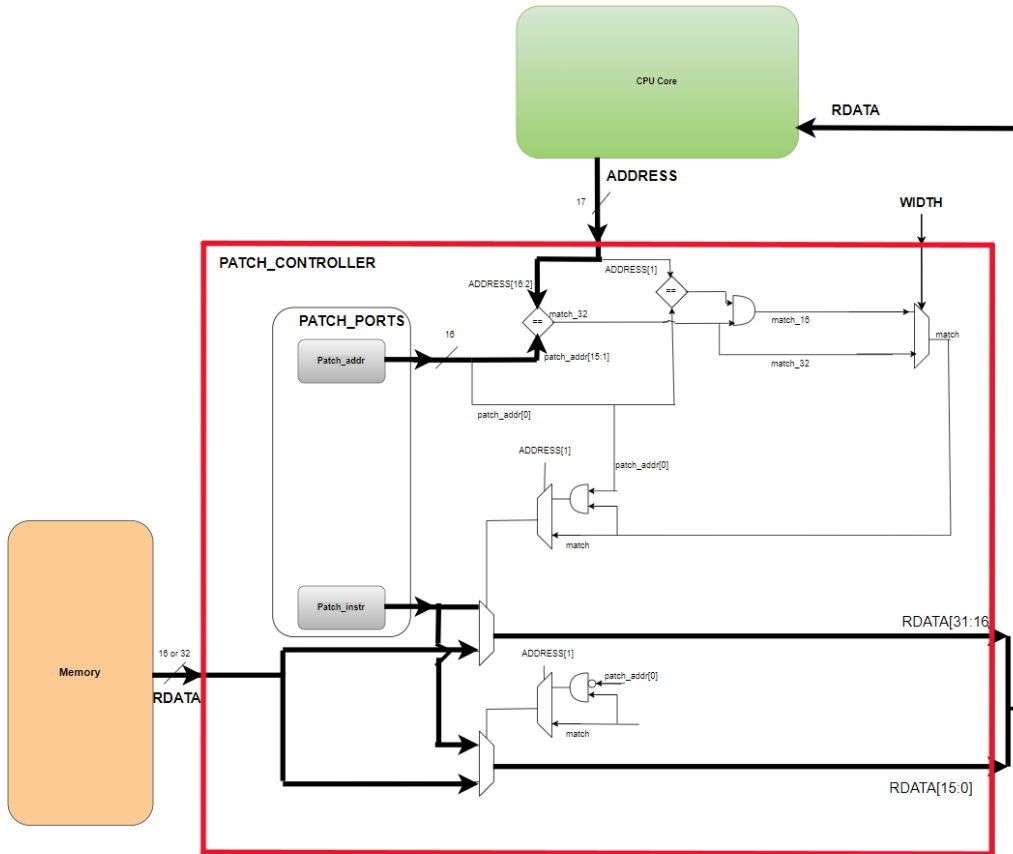


Figure 21: Logic implemented in the core

4.4.3 Patch safety

Until now, we haven't considered RPARITY signal. RPARITY corresponds to the parity of RDATA and this signal described in part 2.2.3 is a safety related signal. The primary purpose of the parity signal is to provide error detection capability. Parity is a simple form of error checking that helps detect single-bit errors in data transmission. The data receiver checks if the parity of the signal actually matches the expected parity of the data, and if not, an error is detected and it means that the data travelling through the RDATA signal is wrong and corrupted. The patch controller is important in the generation of RPARITY signal and a table describing the combinations to properly define RPARITY signal in the case of a 16-bits slave is constructed in table 1. When patches are inactive, the master RPARITY signal is kept identical to the slave RPARITY signal. On the contrary, when they are active, the parity signal at the input of the master depends on whether the instruction port contains a parity bit or not (see io bus description in part 2.2.3); SAFETY_LEVEL parameter informs us on the status of the parity of the instruction port.

Width	slave_RDATA[31:16]	slave_RDATA[15:0]	slave_RPARITY[2:0]	Match?	instr_p_q	master_RPARITY[2:0]
0	X	EVEN	110	1	0 1	110 011
0	X	ODD	011	1	0 1	110 011
0	X	NO PARITY	111	1	0 1	110 011
0	X	X	X	0	0 1	slave_RPARITY[2:0]

Table 1: RPARITY for a 16-bits slave

A more complex table but relying on a similar concept is drawn for a 32 bits slave and can be found in table 2 in the annex.

4.5 Patch both CPU and DMA access - Mlx16-EX

With all the previous modifications and changes made to the design of the patches and after having tested the consecutive designs of the patch controller without any context, it is important now to integrate the patch controller in the shell of the Mlx16-EX and model it in real conditions.

The memory arbiter connects the DMA and the CPU buses to the memory buses. If the CPU and the DMA provides simultaneously an access, the bus arbiter selects in priority the bus having the highest priority; if the CPU and the DMA have the same priority, the CPU is selected. In addition the address decoder decodes the access address in order to determine which memory area is addressed. The DUT is constituted by this memory arbiter containing the newly designed patch controller, two masters (one CPU and one DMA) and three slaves (PER, PROG and VAR as introduced in part 2.2.1). The DUT is shown on figure 22.

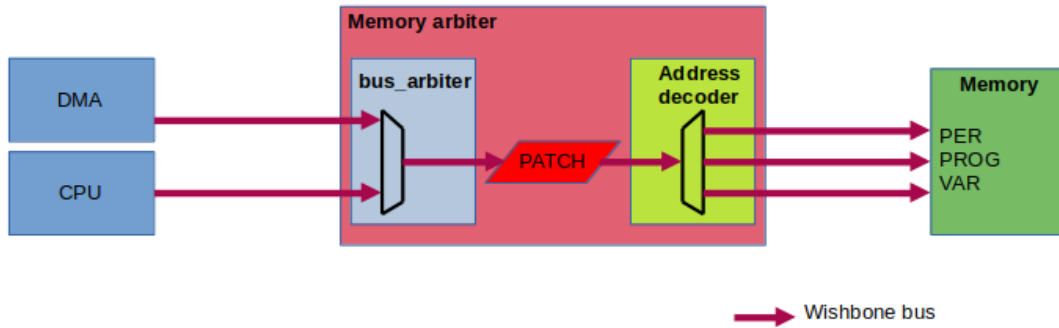


Figure 22: Integration of the patches in the memory arbiter

Several steps were added to the core testbench described in part 4.1. In particular, the parameter WIDTH which defines whether the memory is set to output 16 bits or 32 bits RDATA (see part on Wishbone signals 2.2.3) can vary during the steps of the testbench. Furthermore, the Word or Byte Read accesses achieved during test is performed either by the DMA master or the CPU. By doing so, we verify that the patch controller is operational to patch both CPU and DMA access and that it can handle both 16 bits and 32 bits slaves correctly.

4.6 Patch multiple bus simultaneously - Mlx16-FX

Until now, we only considered a patch controller working with a single bus. However, the FX uses two buses to access memory: one bus for Code and one bus for Data (see part on FX description in *Ganymede* platform 1.2). As explained in the state of the art of the patches in the FX (3.2.1), the current patch module of the FX is integrated in the debugger and is only working on the Code bus. However, sometimes the Data bus also accesses the ROM thus, it is important to make the patch controller flexible enough to be used with two buses when needed (see figure 23).

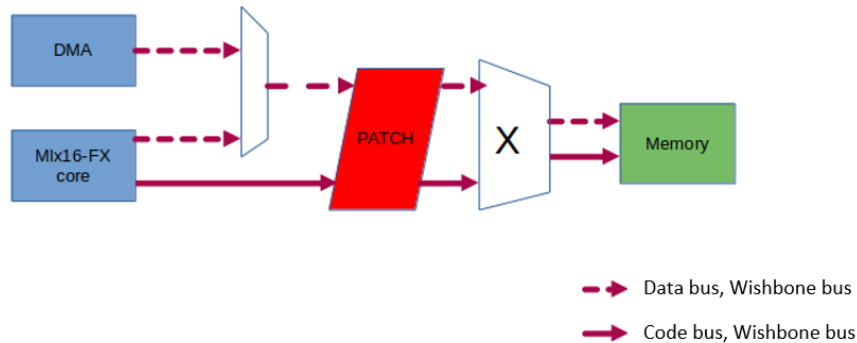


Figure 23: FX-shell with patch controller allowing to patch both Code and Data bus

This parametrization on the number of buses is done by duplicating the core described in part 4.4.2 and by introducing the NB_BUS parameter that is either 1 or 2 depending on whether a single bus or a double bus is used. In practice in the design we instantiate a second PATCH_CORE_UNIT only if NB_BUS is equal to 2 as it can be seen in figure 24.

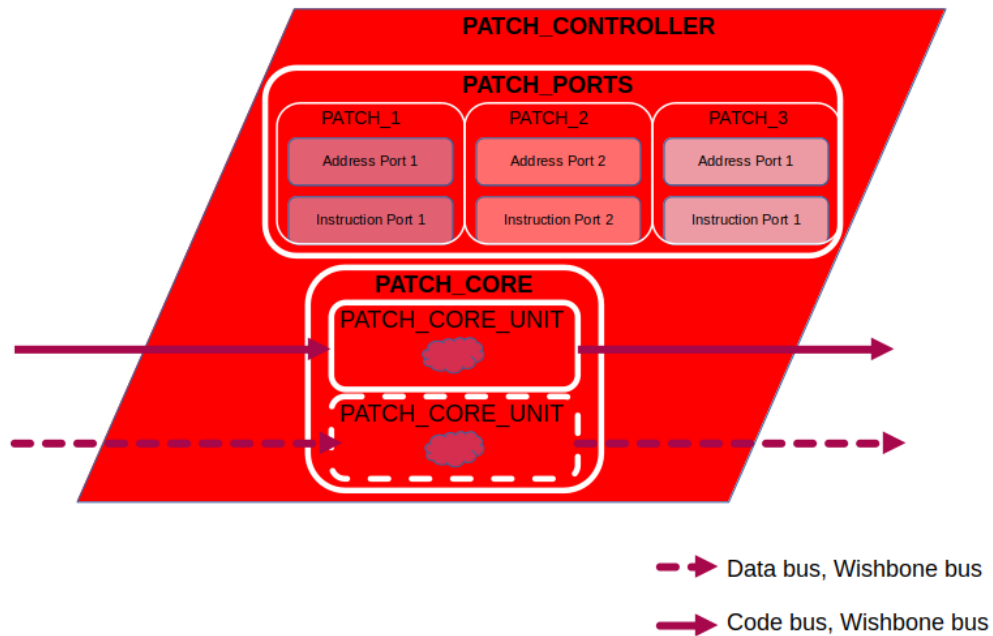


Figure 24: Patch controller allowing to patch both Code and Data bus

4.7 Future steps

The patch controller designed until now is adapted to *Ganymede* platform. Indeed it is Wishbone compatible (part 4.2), parametrized on the patch number (part 4.3), can work both with 16-bits and 32-bits slaves (4.4) and it can be used to patch both DMA and CPU access (part 4.5). Furthermore as discussed in part 4.6, the design is flexible enough to patch several bus simultaneously. However the integration of this new patch controller design in the FX shell has not been tested yet. Once this is done, the objective is to develop the patch controller in the *Callisto* platform. The first step would be to make the patch controller AHB compatible since it is the bus protocol used by ARM CPUs. The verification of the design would follow the same procedure as the one described previously that is to say use AHB masters and slaves freshly designed by another intern and reuse the testbench written in part 4.1.

5 Conclusion

5.1 Conclusion on the patch controller

During this 6 months internship I had the opportunity to gather knowledge on CPU architecture, to develop my skills in RTL design and to effectively test the design I coded. I was assigned to improve the design of a critical block - the patch controller. The already existing patch module has limitations, thus, the objective is to give more flexibility to the module and to overcome these design barriers in order to obtain, at the end of the day, a generic patch controller module that can be implemented wherever we want. The first step was to implement a new testbench and provide a set of self-testable test cases fully covering the future designs. This testbench was improved and features were added during the whole duration of the internship. The first designing step was to get the patch controller Wishbone compatible. This is essential since the Wishbone bus is the protocol used in the *Ganymede* platform. Then, by parameterizing the patch count we allow to patch as many addresses as we judge it is necessary to do, providing more freedom with regards to the previous patch module. The Wishbone compatible patch controller require to patch both 16-bits and 32-bits instructions; taking this into consideration is critical in the design since the patch controller must deal with these instructions in a different manner. Safety is a topic of a paramount importance here at Melexis and it is important that the patch controller take into account this safety aspect. Through the RPARITY signal generation introduced in the patch controller core logic, we make sure that the whole 'patch operation' is correct. The introduction of the module previously designed inside the memory arbiter is a way to verify in real conditions the behavior of the patch controller when both DMA and CPU can access memory. Finally, the last design modification I did was to implement a way to patch two bus simultaneously; this is particularly interesting for the case of the FX where there are two bus (Code and Data). The functional test of this final design wasn't realised; a simple waveforms check was done to confirm the correct behavior for one bus and two bus patch. The integration in the FX shell, similarly to what was done for the EX shell is to be done as the introduction of the patch controller in the *Callisto* platform.

5.2 Personal experience

This internship was for me the opportunity to learn a lot about CPU architecture; this topic is not taught in the MNIS program as we focus more on the process and theory, except for the semester at EPFL. For this master thesis, I wanted to improve my level in any HDL: designing a hardware block and testing it in SystemVerilog was the best practice to do so, thus I am very grateful for this experience. I learned how to use tools required for simulation and had the great occasion to discuss with experienced front-end and back-end digital design engineer. Thus I have a more clear view of the digital flow involved in chip manufacturing. As mentioned in the acknowledgments, DCC team organized several classes for the interns on CPU architecture, memories, DFT and ATPG and the importance of testing especially in the automotive industry and Gitlab for us to feel comfortable with the tools used and the topics discussed during our internships.

6 ANNEX

6.1 List of figures

List of Figures

1	La Defense, Paris - Melexis DCC	1
2	Car applications [1]	2
3	Various sensors developed by Melexis [1]	2
4	Gantt Diagram	4
5	Basic principle of memory mapping	6
6	RDATA disposition 16 bits (left) and 32 bits (right)	9
7	Port description	9
8	Basic principle of patches	11
9	Patch simple description	12
10	Patches in Mlx16-EX shell	13
11	Simplified view of Mlx16-EX shell	14
12	FX-shell	14
13	FX-shell simplified when debugger is in patch mode	15
14	Terminal output of the newly created testbench	16
15	Waves of the ReadWord on addresses to patch for a 16 bits slave	17
16	EX-shell with Wishbone compatible patches	18
17	Ports instantiating in patches of the EX-shell	19
18	New ports instantiating in EX-shell	20
19	Testbench with 7 patches	20
20	Separation core and ports	21
21	Logic implemented in the core	22
22	Integration of the patches in the memory arbiter	24
23	FX-shell with patch controller allowing to patch both Code and Data bus	24
24	Patch controller allowing to patch both Code and Data bus	25
25	Waves of the ReadWord on random addresses for 16 bits slave	29
26	Waves of the ReadByte on address to patch for 16 bits slave	29
27	Waves of the ReadByte on random addresses 16 bits slaves	29
28	Waves of the ReadByte on addresses to patch for 32 bits slaves	30
29	Waves of the ReadByte on random addresses for 32 bits slaves	30

6.2 References

- [1] Melexis website : <https://www.melexis.com/en>
- [2] Wishbone bus protocol: https://cdn.opencores.org/downloads/wbspec_b3.pdf
- [3] Melexis *Ganymede* and *Callisto* platforms documentation

6.3 Complementary documentation

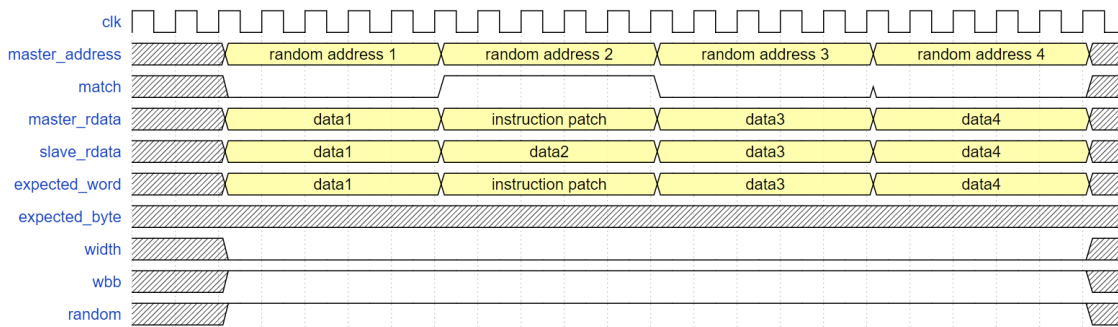


Figure 25: Waves of the ReadWord on random addresses for 16 bits slave

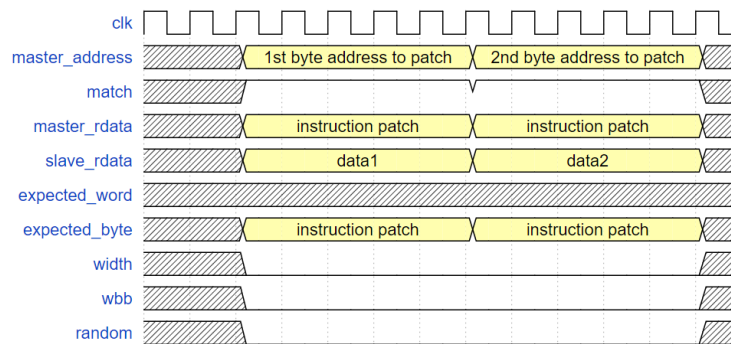


Figure 26: Waves of the ReadByte on address to patch for 16 bits slave

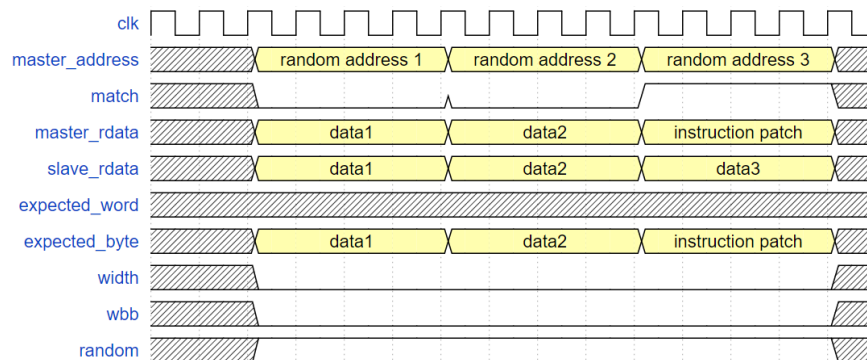


Figure 27: Waves of the ReadByte on random addresses 16 bits slaves

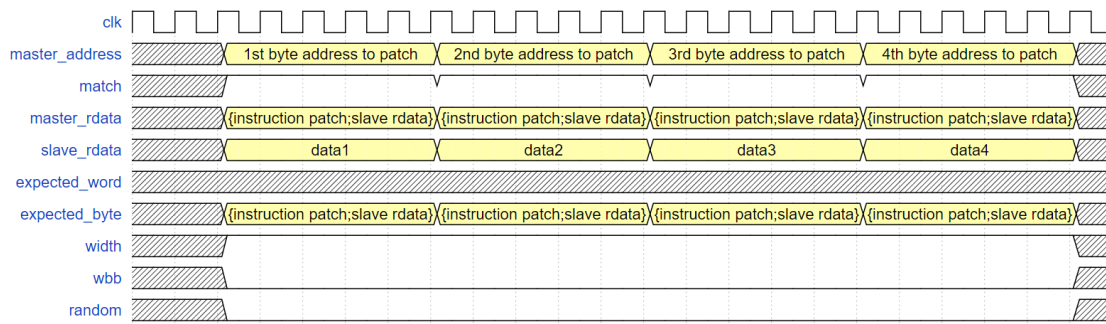


Figure 28: Waves of the ReadByte on addresses to patch for 32 bits slaves

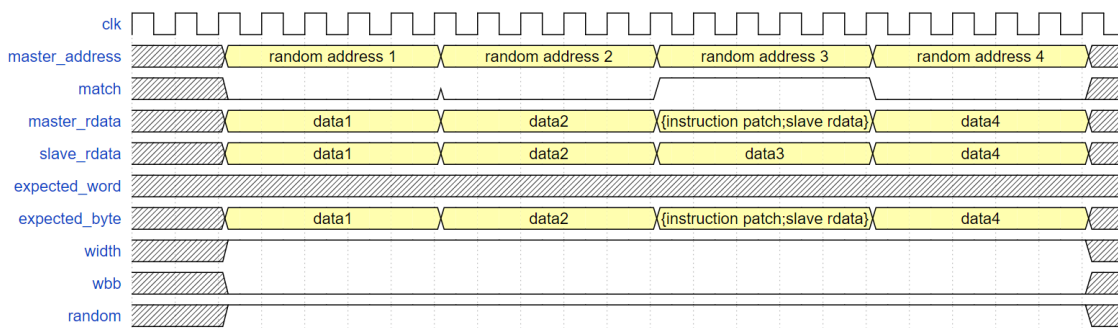


Figure 29: Waves of the ReadByte on random addresses for 32 bits slaves

N.B: The waveforms for the ReadWord with a 32 bits slave are similar to the ones of a ReadWord for a 16 bits slave (see figure 15).

Width	slave_RDATA[31:16]	slave_RDATA[15:0]	slave_RPARITY[2:0]
1	EVEN	EVEN	100
1	ODD	ODD	011
1	ODD	EVEN	110
1	EVEN	ODD	001
1	NO PARITY	NO PARITY	X

Table 2: RPARITY for a 32-bits slave

7 Abstracts

English

The aim of my internship was to create a flexible, versatile patch controller module to overcome previous design limitations. Initial steps involved creating a comprehensive self-testable set of testcases and ensuring Wishbone compatibility, crucial for the *Ganymede* platform. Parameterizing the patch count allowed patching multiple addresses, granting more flexibility than the previous design. Handling both 16-bit and 32-bit instructions in the Wishbone compatible patch controller was a significant challenge. By integrating the module into the memory arbiter, we assess its behavior during concurrent DMA and CPU memory access. Further advancement involved enabling dual-bus patching, useful for Mlx16-FX microcontroller. Although the functional test of the final design was not conducted, preliminary waveform analysis verified its accurate operation. Future work includes integrating the patch controller into the FX shell and its introduction within the *Callisto* platform. This internship was at the end of the day an opportunity for me to acquire knowledge in CPU architecture and improve my RTL design skills, with a focus on enhancing and testing a critical block - the patch controller.

French

L'objectif de mon stage était de créer un module de contrôleur de patch flexible et polyvalent permettant de surmonter les limitations de design précédentes. Les premières étapes consistaient à créer un testbench et à garantir la compatibilité Wishbone, cruciale pour la plate-forme *Ganymede*. Le paramétrage du nombre de patches permet en premier lieu de corriger autant d'adresses que l'on souhaite, offrant plus de flexibilité que le design précédent. La gestion des instructions 16 bits et 32 bits dans le contrôleur de patch compatible Wishbone représentait un défi de taille. En intégrant le module dans le memory arbiter, nous évaluons son comportement et son fonctionnement lors d'accès simultanés DMA et CPU. Une avancée supplémentaire impliquait l'activation du patch fonctionnant avec deux bus en parallèle, utile pour le microcontrôleur Mlx16-FX. Bien que le test fonctionnel du design final n'ait pas été effectué, une analyse préliminaire des waveforms a vérifié son fonctionnement. Les travaux futurs incluent l'intégration du contrôleur de patch dans le shell FX, ainsi que toute son intégration au sein de la plateforme *Callisto*. Ce stage a finalement été l'occasion pour moi d'acquérir des connaissances en architecture CPU et d'améliorer mes compétences en conception RTL, avec un focus sur l'amélioration et le test d'un bloc très important : le contrôleur de patch.

Italian

Lo scopo del mio tirocinio era creare un modulo controller patch flessibile e versatile per superare i limiti di progettazione precedenti. I passi iniziali hanno comportato la creazione di un set completo di casi di test auto-testabili e la garanzia della compatibilità con Wishbone, cruciale per la piattaforma *Ganymede*. La parametrizzazione del numero di patch ha consentito di applicare patch a più indirizzi, garantendo maggiore flessibilità rispetto al progetto precedente. Gestire le istruzioni sia a 16 che a 32 bit nel controller patch compatibile con Wishbone è stata una sfida significativa. Integrando il modulo nell'arbitro della memoria, valutiamo il suo comportamento durante l'accesso simultaneo alla memoria DMA e CPU. Ulteriori progressi hanno comportato l'abilitazione del patching a doppio bus, utile per il microcontroller Mlx16-FX. Sebbene non sia stato condotto il test funzionale del progetto finale,

l'analisi preliminare della forma d'onda ne ha verificato il corretto funzionamento. Il lavoro futuro include l'integrazione del controller patch nella shell FX e la sua introduzione all'interno della piattaforma *Callisto*. Questo stage è stato alla fine della giornata un'opportunità per me di acquisire conoscenze sull'architettura della CPU e migliorare le mie capacità di progettazione RTL, con particolare attenzione al miglioramento e al test di un blocco critico: il controller delle patch.

8 Archive Card

Nano- Microtechnologies for Integrated Systems
2022-2023

Internship done February 13th - August 11th 2023

Grenoble INP-Phelma Supervisor :

ANGHEL Lorena

lorena.anghel@phelma.grenoble-inp.fr

Student :

MANSOUR Bassam

42001948

bassam.mansour@phelma.grenoble-inp.fr

Melexis Supervisor :

BARON Paul

pub@melexis.com

Project Description

The target of this internship is to design and verify a component handling patches for our 3 custom 16 bits CPUs. It will also have to be compatible with our next gen. 32 bits platform (Risc-V based)

Responsibilities

- Design and simulate a patch controller block
- Gather knowledge on CPU architecture and software debugging
- Use GCC and Cadence tools for linting, code coverage and simulation
- Document the designed block for future integration into a complete system.

Expected Deliverables

- A SystemVerilog design database of the patch hardware
- A set of SystemVerilog self-testable test cases fully covering the design
- Formal proof of the correct behavior of the patch hardware

A complete access to the company's tool was granted and help provided by every employee when needed. Work space was provided as for employees.



Melexis NV/BO, Paris

4 Place des Vosges

92400 Courbevoie

FRANCE

<https://www.melexis.com>