SORBONNE UNIVERSITÉ

Master 1 SESI

(Electronic and Computer Systems)

RiVer Project :
Design and implementation of a pipelined RISC-V processor

Students:
M. Timothée Le Berre
M. Louis Geoffroy Pitailler
M. Kevin Lastra
M. Samy Attal

We would like to thank
Ms. Roselyne Chotin, Ms. Daniela Genius,
Mr. Pirouz Bazargan Sabet, Mr. Franck Wajsbürt and Mr. Damien Fruleux
who helped us a lot during the realisation of this project.

Table of contents

1 Introduction

1.1 History of RISC architecture :

In 1971, Intel released its first microprocessor, the Intel 4004 based on a 4-bit CISC (Complex Instruction Set Computing) architecture.

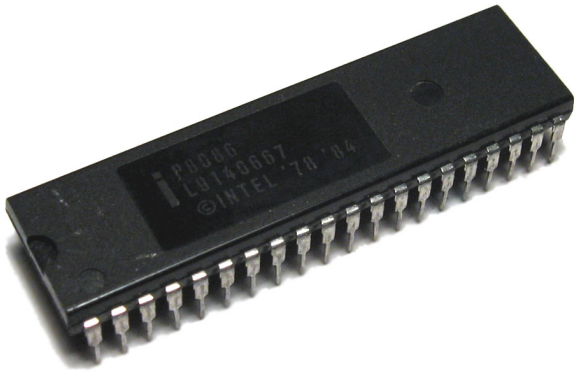


Figure 1: Intel 8086 [1]

CISC processors largely dominated the market until the 1980s when a new architecture appeared: the RISC architecture.
CISC architectures are much more complex than RISC, as the latter implement very complex functions in hardware. One can for example quote the Intel 8086 [1] which implements hardware instructions allowing comparisons between strings of characters.
RISC architectures on the contrary implement only basic and simple functions from a hardware point of view, the philosophy of RISC being indeed to leave the complex tasks to the compiler.
RISC was originally a project led by David Patterson at the University of Berkeley in California between 1980 and 1984.
This architecture will quickly show big advantages compared to the CISC architecture and many projects will be developed based on it.
In 1981, the MIPS (Microprocessor without Interlocked Pipeline Stages) -which is based on a RISC type architecture- made its appearance at Stanford University.
The MIPS technology was commercialised from 1984 and its first implementation, the R2000, became one of the most widely used processors for designing embedded circuits.

At the end of the 1980s, at Sorbonne University (formerly Pierre et Marie Curie) all the courses requiring a processor architecture used different architectures. This is why Professor Alain Greiner decided to homogenise all this by looking for an architecture that was both pedagogical and powerful as a basis for teaching. At first, he turned to the DLX -developed and used by Stanford- but he soon realised that the architecture was little used and little known outside the school environment. This is why he turned instead to MIPS.
This architecture being relatively simple, it allows to present the basic principles of the processor architecture, while being powerful enough to be really implemented. In addition, the architecture can support a multitasking operating system such as UNIX, since it supports two modes of operation :
- A user mode: certain areas of the memory and certain registers of the processor reserved for the operating

system are protected and therefore inaccessible
- A supervisor mode: all resources are accessible.
Nevertheless, this architecture is a commercial architecture and its implementation is controlled, which is why Lip6 and more particularly the SESI Master wishes to change its architecture to RISC-V, an open source architecture. Moreover, RISC-V is quite close to MIPS which makes it easier to understand and implement.
That's why 3 projects, including ours, have been built around this instruction set. In addition to our project, there is a project aiming at taking a synthesizable RISC-V to implement it on an FPGA board in order to run a Linux on it. Finally, another project aims to take a VHDL description of a MIPS 32 made by Lip6, and adapt it to the RISC-V instruction set.

1.2 Origin of the project

This project was initially carried out during the PSESI course of the 1st year of the SESI Master's degree at Sorbonne University. It then gave rise to an internship that allowed us to improve the initial model.
During our first semester of M1 SESI, we had the opportunity to implement a 4-stage scalar core based on an ARMv2a architecture in VHDL.
When Mrs. Genius proposed to implement a RISC-V architecture in SystemC[4], we immediately applied to participate in this project.
This allowed us to familiarise ourselves with the RISC-V instruction set and with the SystemC language. As RISC-V is one of the most widely used implementations of RISC - alongside the ARM architecture - it is very interesting to study how it works.
Not wanting to simply do a scalar architecture and wanting to go further than what we had already had the opportunity to do in VLSI with the ARM architecture, we decided to move quickly on the project with the aim of finishing the scalar implementation at the beginning of March and then being able to concentrate on a 5-stage pipelined SS2 superscalar implementation.

SS2 refers to a 2-stage super-scalar where the EXE, MEM and WBK stages are duplicated. It is an architecture designed by Mr. Pirouz Bazargan. We studied it during our ARCHI course in the 1st semester and we wanted to implement it in our design.

During the semester, within the framework of the PSESI UE, 3 projects had been launched on the RISC-V architecture, ours, a project aiming at implementing an OS on FPGA in which a RISC-V core would run and finally a VHDL implementation starting from a MIPS core.
As the project on FPGA aimed at implementing an OS, it was decided, after discussion with our supervisor, to start first by adding a Kernel part to our design and then move on to an SS2 implementation if there was time left.

We have therefore chosen to implement the RV32IZiscr instruction set with a User and Machine mode. RV32I is the basic 32-bit RISC-V instruction set, and Zicsr is the extension adding status and processor control registers, which are essential for system functions. These extensions were chosen in order to get closer to the MIPS studied at Lip6.

In May, after the project defense, our supervisor accepted to grant us a 3 months internship during the summer of 2022. This was the occasion for us to welcome an additional member, Mr. Samy Attal.
As the other two projects did not progress much during the semester, the VHDL implementation and the FPGA implementation will not be taken over and Mr. Attal had the responsibility of redoing this port based on our SystemC model.
The additional objectives we have set ourselves will be detailed in section 1.3.

1.3 Objectives

The primary objective of our project was to implement a 5-stage 32-bit pipelined RISC-V architecture without extension.
The goal was to replace the MIPS architecture of Lip6. We chose to use the same stages as those present in the MIPS to realize our implementation, namely :

- IFETCH
- DECODE
- EXECUTE
- MEMORY
- WRITE-BACK

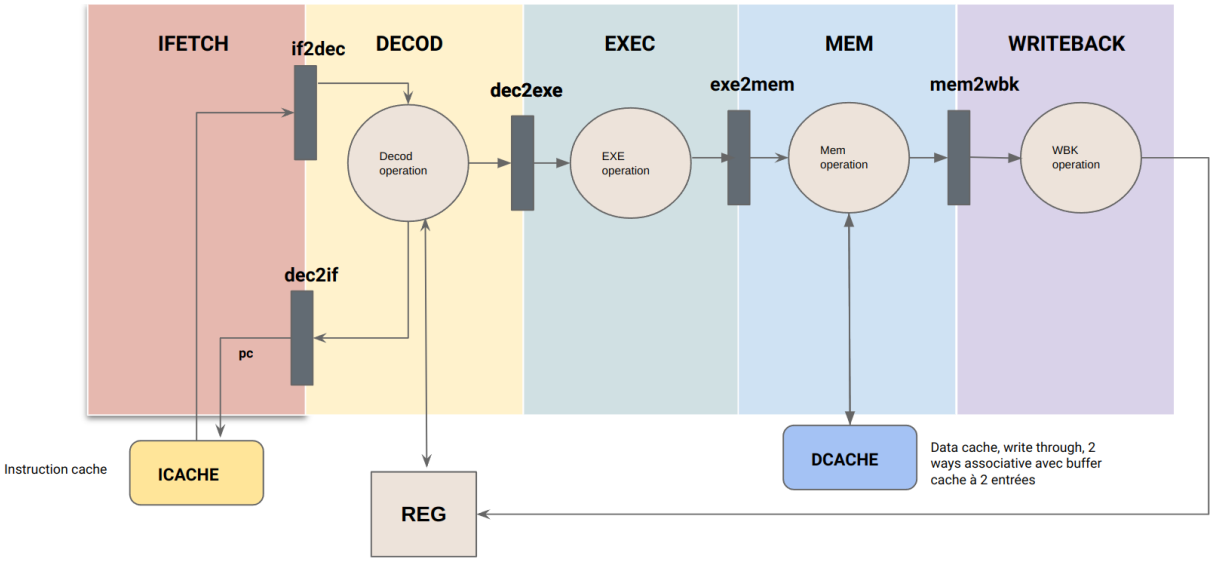


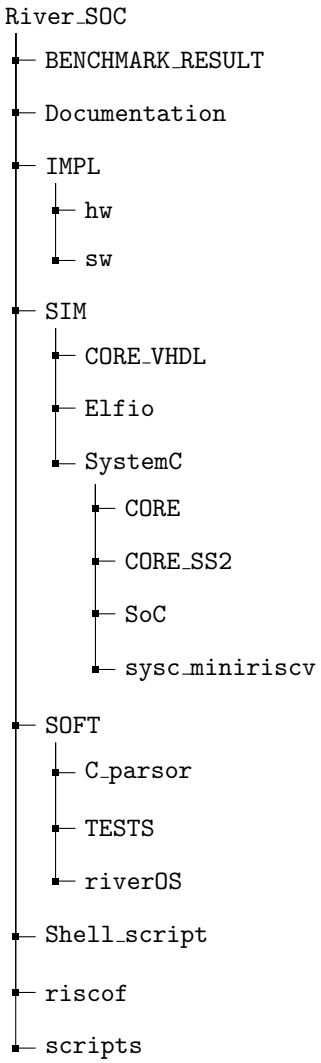
Figure 2: Pipeline diagram, the blocks between each stage designate fifos and the name assigned to them

The imposed language for this realization is SystemC, indeed, it is the language used for the UE MASSOC of M2 from where this choice. Our implementation must be as close as possible to the MIPS R3000 mentioned above so as not to make the transition too rough for the teaching.
We then implemented the Zicsr extension as well as privilege modes and interrupts, as well as instruction and data caches, still to get closer to the MIPS processor of the TME.
Finally, during our 3-month internship, we developed a total of 4 different cores:

- A SystemC RV32IMZiscr core with a user mode, a machine mode and with a branch prediction mechanism that we called **RiVer**,
- The same core, but this time translated into VHDL,
- A superscalar RV32IZiscr core with user and machine mode,
- A mini-riscv for the TP platform which implements only the RV32I instruction set

Finally a Whishbone-based system bus, a Snoopy mechanism for caching and peripherals will also have been developed.

1.4 Github Structure



The git contains the whole project, both the hardware implementation and the software part. The readme on Github contains more information on the hierarchy of folders, if you want more information about them, you can consult our git.
The 3 most important folders are :

- **IMPL** : contains the source code of the IP FPGAs and the drivers developed
- **SIM** : contains the source code of all the cores (VHDL and SytemC) as well as the ELFIO library used to parse the elf files in the case of SytemC
- **SOFT** : contains many assembly and C tests used to debug the cores. Also contains the reset code, the exception handler code and an OS prototype in Rust. The C_pasor folder contains a modification of the code used for the GHDL/C interface, this modified code is used to extract an area of memory into a text file.

To use one of the project's cores, simply go to the desired CORE, CORE_SS2 or CORE_VHDL folder, compile the project with the command **make**.
Finally, you just have to pass an assembler file, a c file or an ELF file as an argument to the executable and the core will execute the code.

2 Work done and roadmap

All the work was carried out between January 2022 and August 2022. As the project progressed, we established a roadmap enabling us to determine deadlines and thus to plan the progress of the project in advance. The table 2 summarises all the tasks carried out, lists the date on which they were finalised, and which project members contributed to them.

Timothée Le Berre	
Louis Geoffroy Pitailler	
Kevin Lastra	
Samy Attal	

	January	February	March	April	May	June	July	August				
RiscV Documentation	X											
Mips R3000												
MIPS30000 documentation		X										
RiscV CORE												
IFETCH stage		X										
DECODE stage		X										
EXEC stage		X										
MEMORY stage		X										
WRITEBACK stage		X										
CORE Debugging		X										
Bypass			X									
Basic RiscV test suite				X								
Test extension I "Riscof"					X							
Git Infrastructure						X						
Branch prediction							X					
RAS							X					
RiscV Extension												
Ziscr				X								
Exception mechanism						X						
M (multiplication/division)							X					
SoC												
ICache				X								
DCache				X								
Caches-core interface					X							
Wishbone Bus								X				
Dual core								X				
SS2												
Stage interface							X					
Bypass							X					
Debeug								X				
Benchmark								X				
VHDL/FPGA												
C/GHDL interface						X						
RV32I						X						
M							X					
Zicsr (kernel)							X					
Branch prediction							X					
Simple FPGA implementation							X					
Final FPGA implementation								X				
IP development								X				
Drivers development								X				
TPs, OS												
Cleaning, code comments							X					
Adaptation of TP topics								X				
Adaptation for OS (timer & TTY)							X					
Start of adaptation of k06							X					

X : Represents the month in which the task was completed

3 SystemC implementation : RiVer core

3.1 Introduction

The core we have implemented is a scalar core with 5 pipeline stages with a user mode and a machine mode. We have implemented the I, M and Zicsr extensions.
The objective of this section is to detail in depth the architectural choices that have been made for the realisation of the processor.

3.2 Pipeline Structure

3.2.1 IFETCH

IFETCH is the stage responsible for interfacing with the instruction cache. It makes a request to the cache to ask for the instruction related to the value of the PC it receives from Decod or to the value sent by the branch predictor which will be detailed in the section 3.5. Once the instruction is received, it will transmit it to the IF2DEC fifo which interfaces with the Decod stage.
This stage has an Interface with the instruction cache which will be detailed in section 4.1.1.

3.2.2 Decod

Decod is one of the most complex stages of the architecture as its role is to retrieve the instruction from IFECTH and decode it to inform the rest of the pipeline of the tasks to be performed.
To decode the instructions, we used the RISCV specification [7] and we wrote a synthesis of the instructions we used, accessible on our GitHub [13].
We have classified the instructions into 8 distinct categories:

- R-TYPE : Les instructions de type R sont des instructions arithmétiques qui utilisent uniquement des registres,
- I-TYPE : Instructions où l'opérande 2 est de type immédiat,
- B-type : Instructions de branchement conditionnel,
- U-type : lui et auipc (add upper immediat to pc),
- J-type : Branchements inconditionnels
- S-TYPE : Instructions de type Store
- CSR-TYPE : Instructions CSR (control and status register), cf section 3.3.1
- System-type : Instructions système (ecall/ebreak)

This distribution allowed us to facilitate the decoding, we then used the documentation to recover the opcode corresponding to each instruction.

RV32I Base Instruction Set

imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND

Figure 3: Opcode, from RISCV documentation [7]

Decod is composed of several parts, it includes two fifos **DEC2IF** and **DEC2EXE** allowing to transmit the information to the IFETCH and EXE stages respectively. It then includes an adder which performs the incrementation of PC. In fact, in order to respect what was done in MIPS, we have made sure that the address calculation is done in this stage. In the case of a branch, the necessary comparisons (equal, lower, higher...) are carried out directly in Decod and a signal inc_pc_sd is generated which indicates whether the standard incrementation of the address is stopped (+4) or whether the offset of the branch is added to PC.
. We have removed the delayed slots after the connections and added a protocol for flushing the fifo in the case of a successful connection. Once all the signals have been decoded, Decod sends all the necessary information to EXE using the fifo **DEC2EXE** and sends the next PC value to IFETCH using **DEC2IF**.

3.2.2.1 M extension

Finally, at the end of April, we started adding the M extension, which allows us to add a multiplier and a divider in hardware. Indeed, not having these two entities, it is impossible for us to efficiently perform multiplication, division or modulo operations in a C program. It is therefore necessary to redefine these functions for each program.
A processor that does not have these hardware components can emulate the operations normally performed by the component using the exception handler. But the execution of a multiplication, for example, will be less optimised and will take more cycles than with a multiplier. The addition of these components therefore reduces the number of cycles required to perform these operations and therefore increases the performance of the processor.

RV32M Standard Extension

0000001	rs2	rs1	000	rd	0110011	MUL
0000001	rs2	rs1	001	rd	0110011	MULH
0000001	rs2	rs1	010	rd	0110011	MULHSU
0000001	rs2	rs1	011	rd	0110011	MULHU
0000001	rs2	rs1	100	rd	0110011	DIV
0000001	rs2	rs1	101	rd	0110011	DIVU
0000001	rs2	rs1	110	rd	0110011	REM
0000001	rs2	rs1	111	rd	0110011	REMU

Figure 4: Opcode, from RISCV spec, extension M : [7]

3.2.3 EXEC

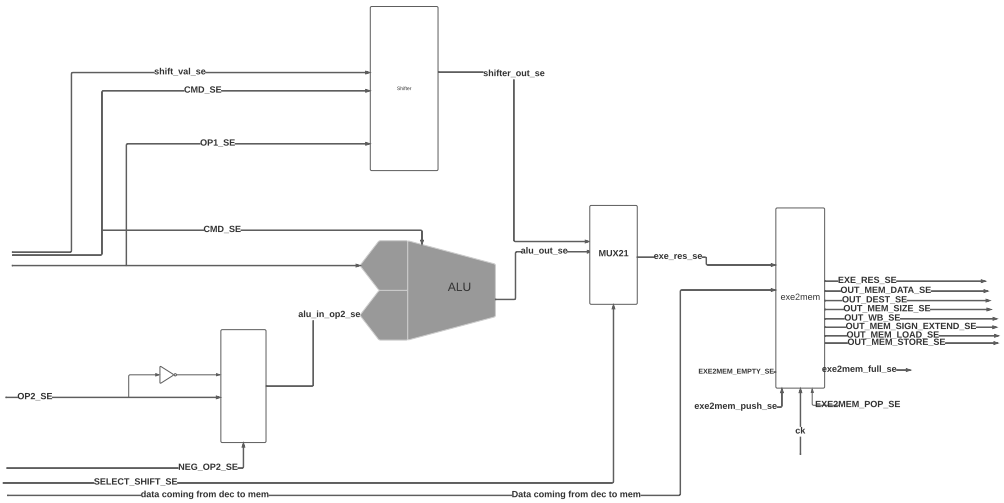


Figure 5: EXEC internal micro architecture

EXEC is made up of 2 essential parts: the ALU (*arithmetic logic unit*) and the shifter.
To select the entity to be used, Decod sends a SELECT_SHIFT_SE signal to EXEC which indicates whether the shifter or the ALU is selected. A 2-bit command then indicates the operation to be performed, logic operations

for the ALU and offset for the shifter. To carry out the subtractions, we placed a controlled inverter in EXEC which makes it possible to make the complement with two of a signal and thus to carry out a subtraction. The result of EXEC is then sent to the fifo **EXE2MEM** which allows to transmit all the necessary data to the MEM stage.

3.2.4 Multiplier

In order to be able to use multiplications it is necessary that we implement a multiplier. The implementation we chose was initially the Array Multiplier.

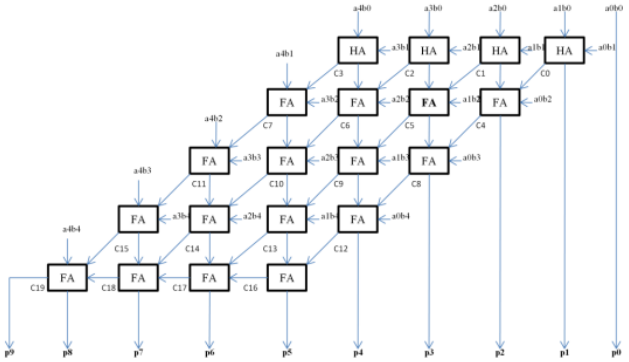


Figure 6: Array multiplier[14]

However, after discussion with Mr. Bazargan Sabet, we decided to change the implementation, as the latter is not very optimized and requires very long propagation times. The idea put forward by Mr. Bazargan Sabet is to use a pipelined multiplier on 3 stages (x0, x1 and x2) based on the "Wallace Tree multiplier" algorithm [6]. The idea is that in the event of a multiplication, Decod will propagate a signal to inform the EXE, MEM and WKB stages that such an operation is about to be initiated, which will block the bypass and write to memory. The instruction is then sent to the x0 stage which will start calculating the rows of partial products, following this calculation the rows will be propagated to the first stage of the wallace tree, the few will extend to the x1 stage, and finally the output of the tree will be summed in the x2 stage.

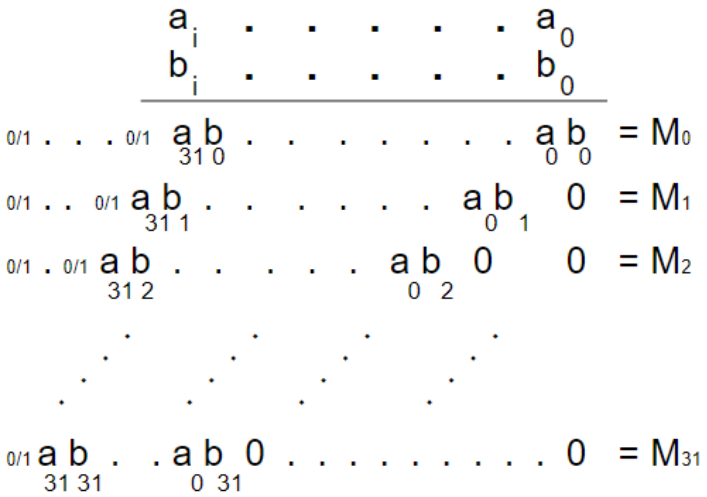


Figure 7: Multiplication(a_i binary representation of a, b_i binary representation of b)

M_i : corresponds to a partial product row of $\sum_{n=0}^{i} a_n * b_i$

Signed multiplication

3 cases to be taken into account:
1. if A and B are negative, we must only apply the complement A2.
2. if B is negative, the most significant bits of our partial product will be equal to 1.
3. if A is negative, we invert A and B and thus apply case number 2.

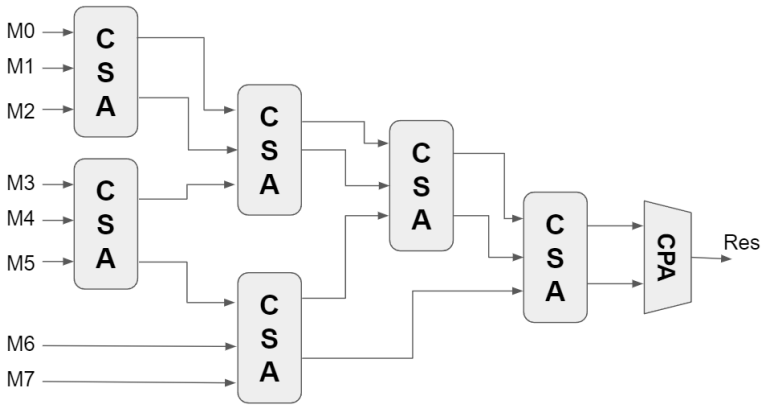


Figure 8: Wallace tree multiplier

CSA : Carry save adder
CPA : Carry propagation adder

To implement the multiplier, it is therefore necessary to implement it in the EXE, MEM and WBK stages, which are renamed X0, X1 and X2 respectively. The latter will thus be responsible for carrying out the calculation necessary to execute the multiplication instruction.

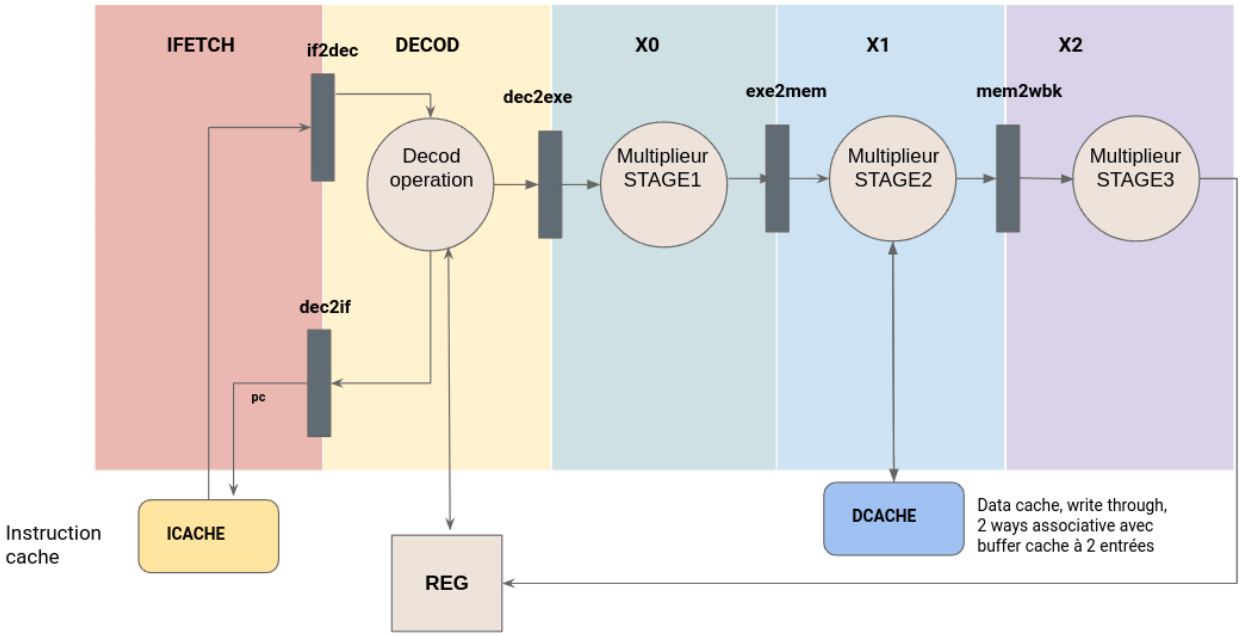


Figure 9: Schéma du Multiplieur intégré au pipeline

3.2.5 Divider

Our implementation of the divider consists of 3 shift registers (2 of 64bits and 1 of 32bits), a subtractor and a state machine.

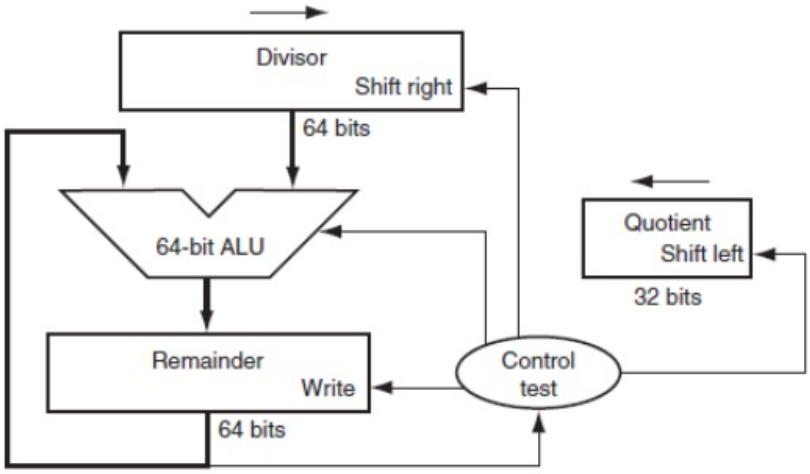


Figure 10: Diagram of the division algorithm [16]

Algorithm 1 Division algorithm

```
cpt ← 0
while cpt <= 31 do
    if divider >= remains then
        reste ← reste − divider
    end if
    quotient ← quotient << 1                    ▷ logic shift left
    quotient ← quotient + 1
    divider ← divider >> 1                      ▷ logic shift right
    cpt ← cpt + 1
end while
```

Division by zero

The result of a division can vary depending on the instruction used :
For REM and REMU the dividend is returned, for DIV and DIVU -1 is returned.

3.2.6 MEM

MEM performs load and store memory accesses. It receives an EXE_MEM_SIZE_SM signal which indicates whether the access is in bytes, half-word or word. Thus, in the case of a store, it indicates which bits are stored in memory and in the case of a load which bits are to be kept in the destination register.
In our implementation, we have allowed accesses to be aligned with the type of access. That is to say that in the case of a lb, it is possible to make byte-aligned accesses, in the case of an lh half-word-aligned.
In the case of a load, a masking will be done in the MEM stage allowing to load only the bits related to the type of access. The bits are always loaded into the low-order bits of the destination register.

In the case of a blind, the accesses can also be type-aligned, but this time the masking is not done in the MEM stage. Indeed, we had not initially thought about the problem of sb and sh which only need to modify the part

of the register they wish to store. This is why the address we sent to our core was always on 32 bits.
The problem being that our memory is simulated by a c++ map which is indexed by address, but in the case of a store byte at address 0xF000C121 for example, the map must not allocate address 0xF000C121, it must write the 2nd byte of address 0xF000C120.
To overcome the problem, we sent the access size to the core which performs a masking on the received address and replaces only the necessary bits in the c++ map.
Finally, the processed data is sent to WBK using the fifo **MEM2WBK**.
All the exception handling is also found in this stage, but this will be developed in the 3.3 section.

3.2.7 WBK

WBK receives signals from MEM and will write to the register bank if necessary (store instructions do not write anything). A REG_WB_SW signal is sent to REG indicating whether or not the received data should be stored.

3.2.8 REG

Finally, the register bank contains 33 registers, the 33rd being PC. We have chosen to place PC in REG in order to use the same signals for the data instructions.
Indeed, REG is placed directly in the core, it does not belong to a specific stage since both Decod and WBK can access it, Decod doing read accesses and WBK doing write accesses.
Thus placing PC in the register bank allows us to use the same interfaces for branch instructions, for example jal.
Note that the read address is on 6 bits in REG despite the fact that the opcode addresses are only on 5 bits.
In fact, in the case of branches, the value of PC is read, hence the interest in passing it on 6 bits.

3.3 CSR extension, exceptions and interruptions

Once our scalar processor with the base extension I (Integer) was implemented, we moved on to the implementation of the Kernel part which consisted of 3 essential points:

- CSR extension,
- Interruption and exception handling,
- Addition of a user mode and a kernel mode

3.3.1 Extension CSR :

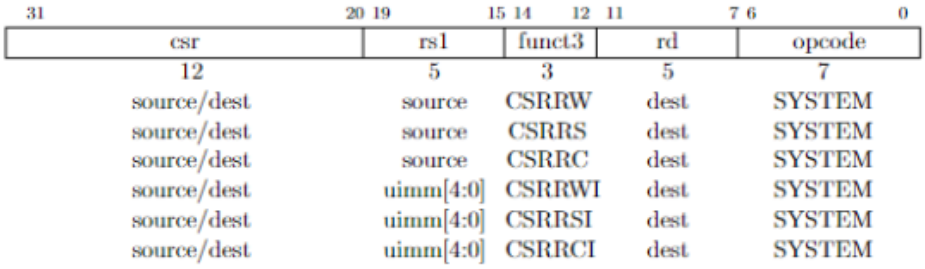


Figure 12: RISC-V CSR Instruction

Instruction	funct3	opcode
CSRRW	001	1110011
CSRRS	010	1110011
CSRRC	011	1110011
CSRRWI	101	1110011
CSRRSI	110	1110011
CSRRCI	111	1110011

Figure 11: Opcode CSR instructions

This extension is necessary for the addition of a Machine/Supervisor mode since it allows the addition of instructions that manage CSR (control status registers). CSRs are the equivalent of coprocessor-0 registers in MIPS and CSR instructions are the analogue of mfc0/mtc0 type instructions. Thus, there are instructions to read/write these status registers.
To implement them, we have therefore added the CSR register bank. This bank stores information about the implemented architecture and the current state of the pipeline. The Book 2 specification states [8] that the RISC-V architecture provides for up to 4096 CSRs, but not all of them are defined or required. That is why we have only implemented those necessary for our architecture, namely :

- mvendorid : CPU vendor ID,
- marchid : gives information about the base used for the architecture, 32 in our case,
- mimpid : CPU version,
- mhartid : CPU ID,

- mstatus : it is the status register of the processor, it keeps many information about the processor for example the current mode of the processor, the activation or not of interrupts...
- misa : gives the extensions implemented in the architecture
- mie : Contains information about activated machine interrupts
- mtvec : contains the base address of the trap functions, allowing an exception/interrupt to be handled, supports both **vectorize mode** and **direct mode**
- mstatush : same as mstatus
- mepc : stores the address that caused the interrupt/exception
- mcause : contains a code identifying the cause of the exception/interruption that occurred
- mtval : When a trap is taken in machine mode, mtval stores information about the exception
- mip : contains information about upcoming interruptions
- mscratch : Contains the address of a stack allowing the program to save the values of the csr registers
- kernel : Custom CSR register, placed at address **0x800**, which we have added to manage the supervisor address in software

To add instruction decoding to the pipeline, it was necessary to modify almost all the stages. Indeed, CSR instructions are atomic operations, i.e. when there is a CSR instruction in the pipeline, another instruction manipulating the same register cannot write or read the CSR that is being processed.
To overcome this problem, we have therefore propagated a csr_enable signal in each stage in order to tell whether a CSR instruction is in progress in that stage. This signal is then redirected to Decod to inform it that it cannot perform an instruction manipulating the same CSR as the one already present in the pipeline and that it must therefore freeze if this were the case.
Furthermore, as explained in the previous section, we have defined a boundary between EXE and MEM for exception handling, so the CSRs are written at the end of MEM i.e. at the moment when we look at what caused the exception. Once this processing is complete, the CSRs are updated at the end of the MEM cycle.

At the end of the project we also decided to add a custom register to manage the privileged mode address. This register is initialized to 0xFFFFFFFF during the reset, this allows to avoid problems during tests with riscof. Indeed, programs that do not take this feature into account could fail to run and generate exceptions since they run in user mode. If the value of this register is left at 0 when the processor is reset and the program code does not initialise it, this will generate numerous illegal access exceptions.

To validate our model, we used the riscof framework [15] and adapted our architecture where necessary to fit the test suite.
This is why we chose to write the PC value responsible for a misaligned or access-fault exception as well as the value of a store/load doing the same type of error in mtval. This is not mandatory if we follow the specification, but in practice it is useful for exception handling.

3.3.2 Exceptions handled in our architecture :

In order to implement the machine mode, we had to make it possible to detect and handle exceptions and interrupts in our processor.
Mr. Pirouz Bazargan Sabet explained to us how he had implemented and handled exceptions in the MIPS pipeline. Thanks to his explanations, we had a basis allowing us to implement the latter.
In order to set up exception/interrupt handling on our RISC-V processor, we first listed all the exceptions in the specification we were going to implement. Here is an exhaustive list of these exceptions :

- Instruction address misaligned : Adress of the next instruction in decod is miss-aligned

- Instruction access fault : Trying to access a memory area without the necessary privileges
- Illegal instruction : The instruction does not exist or is not implemented
- Load address misaligned
- Load access fault : Access to an area with wrong privilege
- Store address misaligned
- Store access fault : Access to an area with wrong privilege
- Environment call from U-mode : syscall from user mode
- Environment call from M-mode : syscall from machine mode
- Environment call from wrong mode : mret in another mode than machine one
- Mret : mret instruction is handled as an exception

Once the exceptions to be implemented were listed, we had to determine in which stage we were going to handle them, and for this we had to place a "barrier" in the pipeline.
Indeed, when an instruction triggers an exception, the following instructions must not be taken into account. From a micro-architectural point of view, this means that these instructions must not modify memory or the REG/CSR register bank.
This means that any instructions that arrive in MEM or WBK after an exception is caught must not modify memory or register banks. To do this, memory access and the signal to write to the register bank in WBK must be disabled when an instruction is detected. Note that the disabling in MEM is done before the memory access.

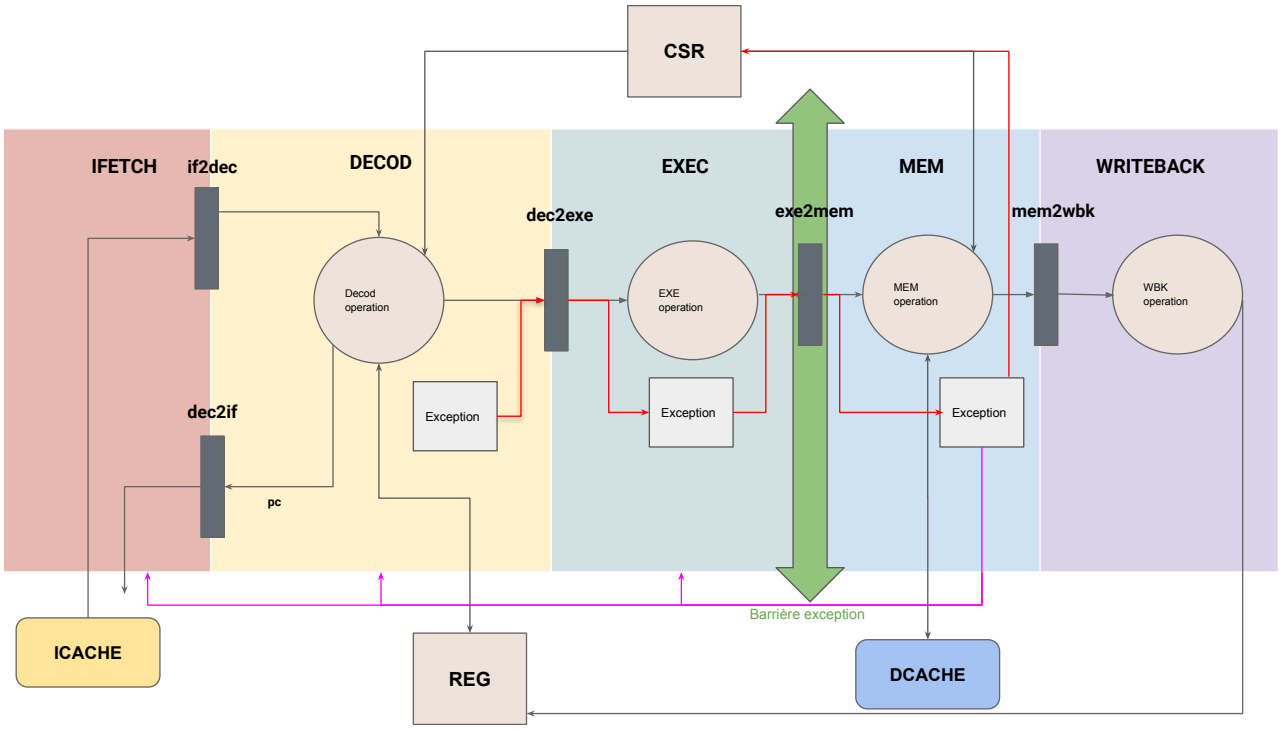


Figure 12: Diagram of the pipeline with exception handling

The pipeline is therefore the same as before with a few differences. Each stage detects its own exceptions, which can be broken down as follows:

- Instruction address misaligned : Decod
- Instruction access fault : EXE
- Illegal instruction : DEC
- Load address misaligned : EXE
- Load access fault : EXE
- Store address misaligned : EXE
- Store access fault : EXE
- Environment call from U-mode : DEC
- Environment call from M-mode : DEC
- Environment call wrong mode : DEC (custom)
- Mret : DEC

In each stage we perform a logical "or" between all the exceptions that we propagate to the next stage. For example, in Decode we will perform :

illegal_instruction **or** adress_misaligned **or** syscall_u_mode **or** syscall_s_mode

Then, we will send the result of this **or** to EXE where we will make a new **or** with all the exceptions of this stage.

This allows us to relieve the MEM stage, in fact we could simply send all the signals in MEM and do the **or** in this stage, but this would increase the propagation time of the signals in MEM which is already long because of the memory accesses.
In addition, we have modified the pipeline so that the PC of each instruction is passed from one stage to another. Thus, when an instruction arrives in MEM, if an exception is detected it is sufficient to write the PC value to the mepc.

3.3.3 Exception handling and reset mechanism:

3.3.4 Reset mechanism:

Our processor starts in machine mode, so all the reset instructions will be done with the highest privilege mode. This last one ending with a mret instruction it will leave the machine mode to pass in the user mode and thus pass to the main.
The reset code initializes the status registers necessary to the operation of the core, we must initialize
- the address of main in mepc
- the address of the exception handler in mtvec
- an address, allowing an OS to temporarily save csr values in memory, in mscratch,
- the starting address of the privileged area in the kernel csr register (**0x800**)

Once the csr registers are initialized, the address of the exception handler functions must be saved in the isr vector.
Finally it is necessary to initialize the stack pointer sp.

3.3.5 Exception handling mechanism :

When one or more exceptions arrive in MEM, they will have to be handled and the program in progress in the pipeline will be interrupted.
If several exceptions arrive at the same cycle in the MEM stage, the processor will process them with an order of priority defined in the RISCV spec.
Once MEM detects an exception it will perform the following steps:

- Change the mstatus register to change the pipeline mode to a higher privilege (from user word to machine mode in most cases)
- Write the address of the instruction responsible for the exception to the csr mepc register
- Write the value responsible for the exception if there is one in mtval (e.g. misaligned load value)
- Write the exception number in the csr mcause register
- Send a signal to Ifetch, Decod and Exec indicating that an exception has occurred

In response to the received signal, the Ifetch, Dec and Exec stages will flush their fifos by inserting nop instructions. Decod will then load the value of mtvec into PC.
Our implementation supports the two modes proposed by mtvec, i.e. a direct mode and a vectorised mode.
Finally, Ifetch will retrieve the value of the Interrupt handler and it will execute the code of our Interrupt handler.

3.4 Interruptions handling:

3.4.1 Used of Csr registers to implement a timer

In the absence of a real standard on timers in RISC-V processors, we decided to use the Csr as a control register for a "Timer" component, allowing to fill the time and timeh registers giving the system time, as well as to generate interrupts at regular intervals, useful in particular for operating system type programs.
The specification provides for free-use register numbers decided by the processor designers. We have chosen two of these registers, 5C0 and 5C1, to control the timer.
The first is the configuration register, so the first bit decides whether to enable or disable the interrupts, and the second decides whether the timer interrupt "resets" automatically once processed, or whether to do so from within the program.
The second register is the timer's "divider", which divides the frequency of interrupts (in practice, it is the number of "ticks" of the timer between two interrupts).

These registers are "write only", i.e. you can write into them, but a read always returns 0.

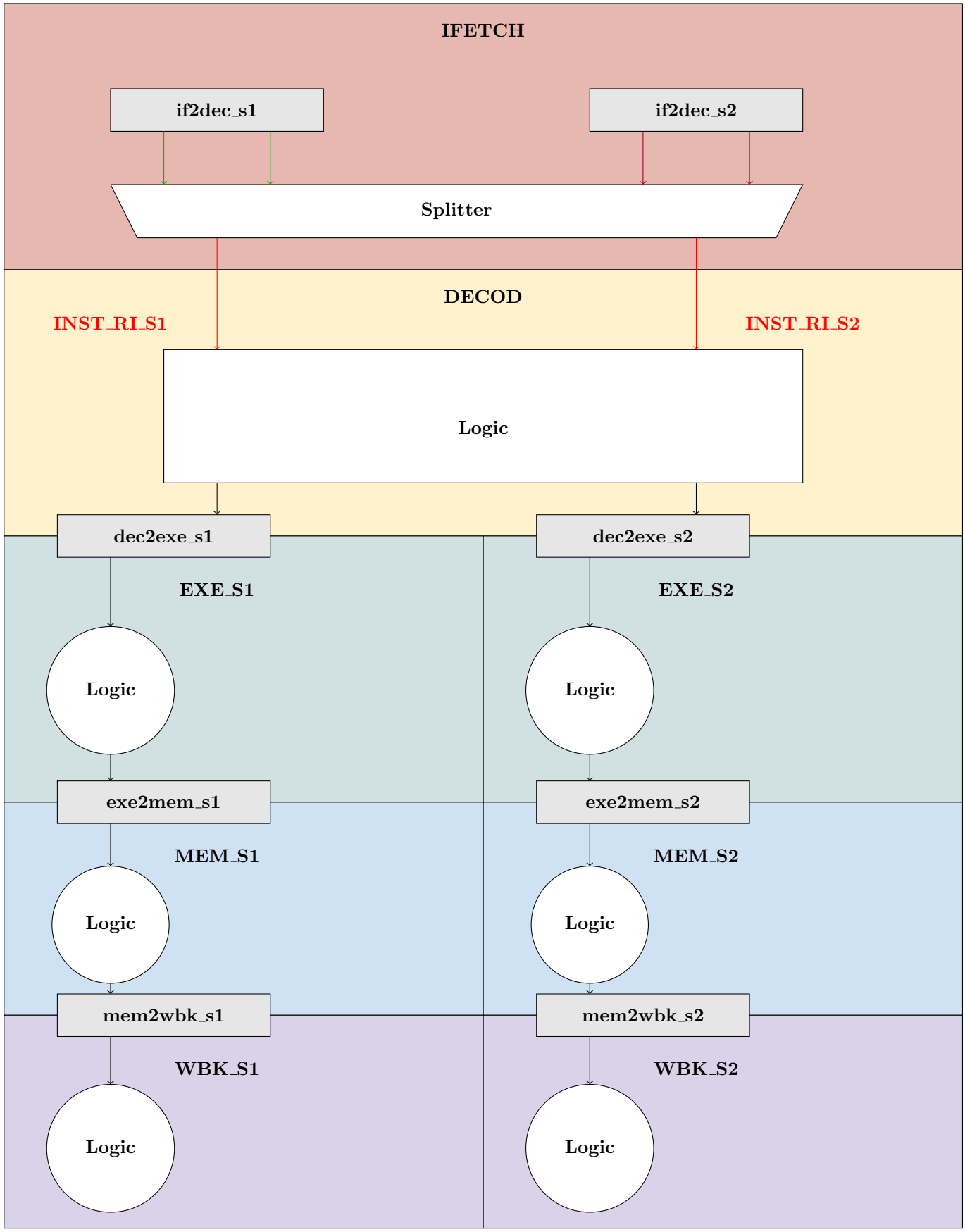
3.5 Core optimisation

After having completed the implementation of our core with the I, M extensions and the User/Machine mode, we wanted to improve some mechanisms in order to increase the performance of the core. To do so, we chose to implement a branch prediction mechanism and to design another superscalar core.

3.5.1 Super-scalar 2 ways :

Once the scalar with the kernel part was complete, we moved on to making the dual pipeline for the super-scalar.
We chose to duplicate the EXEC, MEM and WBK stages while leaving the multiplier in a separate pipeline and we modified the DECOD and IFETCH stages so that they would be able to process two instructions/addresses in a single cycle.
To realize SS2 we divided its creation into two stages :
- preparatory work aimed at doing the bulk of the work, i.e. renaming the signals and preparing the interfaces between all the stages - implementing the logic to manage data dependencies and the implementation of all bypasses.

We have chosen to keep the priority of stage S1 over stage S2, as in the case of data dependency on DECOD, the inversion of instruction loading will be done directly in IFETCH. This will be explained in more detail in the section 3.5.1.1.


IFETCH
if2dec_s1
if2dec_s2
Splitter
DECOD
INST_RI_S1
INST_RI_S2
Logic
dec2exe_s1
dec2exe_s2
EXE_S1
EXE_S2
Logic
Logic
exe2mem_s1
exe2mem_s2
MEM_S1
MEM_S2
Logic
Logic
mem2wbk_s1
mem2wbk_s2
WBK_S1
WBK_S2
Logic
Logic

3.5.1.1 IFETCH

We modified IFETCH so that it could request two addresses from the cache in a single cycle. We chose to keep two separate addresses rather than using a single 8-aligned address. This choice was made because we have implemented a branch prediction mechanism on our scalar core and it would have been easier for the scalar core to retrieve two addresses from DECOD. However, due to time constraints, branch prediction was not implemented on SS2.
In order for IFETCH to be able to transmit these addresses to DECOD, we have replaced the fifo **if2dec** by two circular fifos of two places, **if2dec_s1** et **if2dec_s2**.
In order to manage data dependencies in DECOD, the latter generates a signal **PRIO_PIPELINE_RD** which indicates how the instructions should be loaded.
Indeed, IFETCH has two outputs **INSTR_RI_S1** et **INSTR_RI_S2** which allow to send the instruction to be decoded to the DECOD stage.
In the case where PRIO_PIPELINE_RD is 0, we are in the so-called "normal" case where S1 has priority over S2. If this same signal is 1, we are in the case where S2 has priority over S1. In this case, the PRIO_PIPELINE_RD signal will indicate to IFETCH to invert the loading of instructions on the INSTR1 and INSTR2 output ports using the component that we have called **splitter** and which is actually very close to a multiplexer.

The truth table of the splitter is as follows :

if2dec_s1	if2dec_s2	PRIO_PIPELINE_RD	INST_R1_S1	INST_R1_S2
0	0	0	0	0
0	0	1	0	0
0	1	0	0	1
0	1	1	1	0
1	0	0	1	0
1	0	1	0	1
1	1	0	1	1
1	1	1	1	1

The truth table can thus be translated by the following relation:

$INST\_R1\_S1 = PRIO\_PIPELINE\_RD.if2dec\_s2 + !(PRIO\_PIPELINE\_RD).if2dec\_s1$
$INST\_R1\_S2 = PRIO\_PIPELINE\_RD.if2dec\_s1 + !(PRIO\_PIPELINE\_RD).if2dec\_s2$

3.5.1.2 DECOD

The DECOD stage, which is now able to decode two instructions in a single cycle, must be able to handle RAW (read after write) data dependencies. Indeed, if one of the source addresses of the S2 instruction is the same as the destination register of the S1 instruction, the S2 instruction should not be loaded.

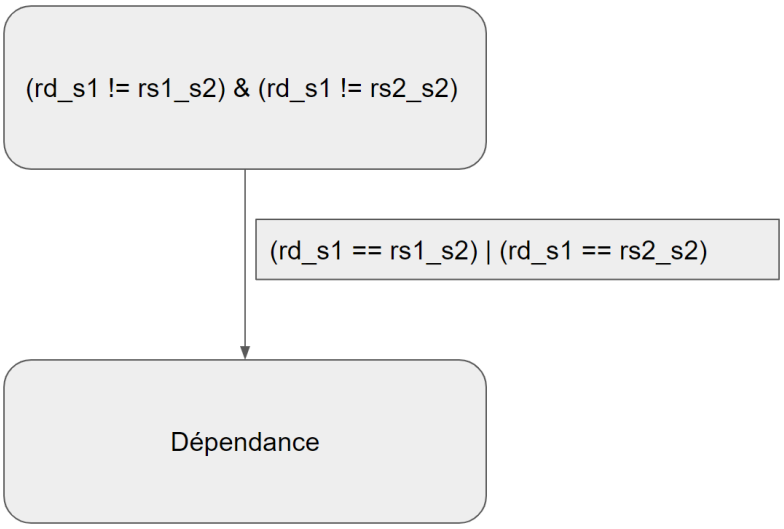


Figure 13: Data dependencies handling in decod

If an addiction is detected, it will be necessary to :

- loading a nop in the fifo dec2exe_s2,
- Pop only when the current priority is on if2dec_s1, else pop only if2dec_s2
- Reverse the instruction loading priority in IFETCH by changing the value of PRIO_PIPELINE_RD

In practice, with a program like this one:

```
1    add x2,x0,x1
2    add x3,x2,x0
3    sll x8,x0,x1
```

We'll have:

	cycle i	**cycle i+1**
S1	add x2,x0,x1	add x3,x2,x0
S2	NOP	sll x8,x0,x1

We have extended this mechanism of inversion of instruction loading in case of data dependency in the case of csr instructions, in fact in the case where two CSR instructions are decoded at the same cycle, the 2nd one will not be loaded and will be treated exactly as if a data dependency had been detected. We did this in order to avoid any kind of problem on CSRs handling the same register address, thus guaranteeing the atomicity of these instructions.

Finally, the PC incrementation mechanism will also have been modified since DECOD no longer generates a single address, but two, so it generates an address incremented by 4 with respect to the relative PC of the 1st instruction and a PC incremented by 8 with respect to the 1st instruction (in other words PC+4 and PC+8).

3.5.1.3 Memory access and others stages

The EXE and WBK stages are almost similar to those of the scalar except for the bypass part which will be detailed in the section 3.5.1.4.
In order to optimise memory accesses, we have chosen to **make simultaneous memory accesses possible** on S1 and S2.
Nevertheless, doing this poses a major problem, because when we do
- a store on A1 in S1
- a load on A1 in S2
You have to make sure that the processor processes the store request first, then the load request. If caches are used, they must be able to process the highest priority request (S1) before processing the lowest priority request (S2). In practice, the resolution of this problem has not been dealt with since the caches were not modified for SS2 as Kevin who was in charge of the caches was working on branch prediction at the time Louis was dealing with SS2.

However, this problem does not appear in simulation when the caches are disabled. Indeed without caches, the memory simulated by the core_tb via a c++ map, which is a perfect memory since it responds in a single cycle. Moreover, the core_tb has been built so that it always responds first to requests from MEM_S1 before responding to those from MEM_S2 and as S1 always has priority over S2 this avoids the problem mentioned above.

However, this problem would have to be handled differently in an ASIC/FPGA implementation since the memory has a latency.

Finally, the MEM stage is also in charge of exception handling.
We chose to create the skeleton necessary to disable simultaneous memory accesses and thus to invert the priority between MEM_S1 and MEM_S2, nevertheless the signals exist, but are never used, so S1 will always take priority over S2 in our implementation.
Thus, if an exception is detected in S1, it will have priority over the one in S2. In order to avoid having an independent handling of exceptions, we have chosen to transmit all the signals of an exception detected in S1 directly to S2. It is then the latter that will be responsible for accessing the CSR register banks and updating the register values with the values calculated in S1 or S2, depending on where the exception occurred. Finally, it is also M2 that will transmit a signal to the other stages informing them that an exception has occurred, although they are not aware of the stage in which it has occurred.

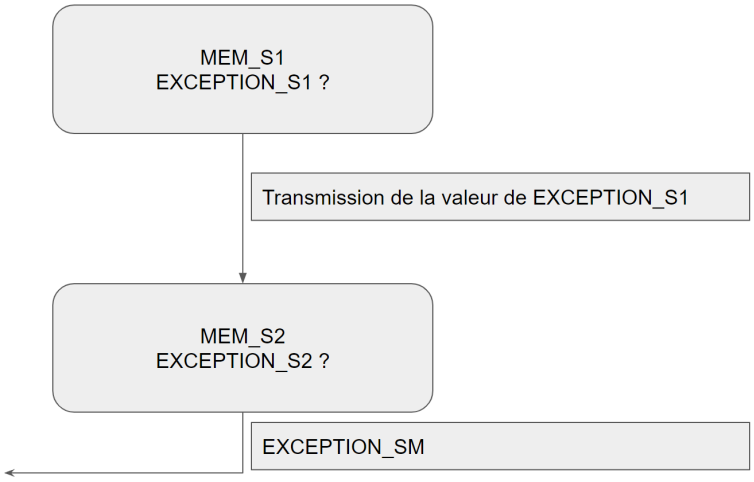


Figure 14: Exception signal transmission between MEM_S1 and MEM_S2

3.5.1.4 Bypass

To ensure optimal operation of the SS2, the total number of bypasses must be increased to 12.

I	D	E1	M1	W1					
		E2	M2	W2					
	I	D	E1	M1	W1				
			E2	M2	W2				
		I	D	E1	M1	W1			
				E2	M2	W2			
			I	D	E1	M1	W1		
					E2	M2	W2		
				I	D	E1	M1	W1	
						E2	M2	W2	

Figure 15:

- Bypass on EXE :
  - E1 → E1
  - E1 → E2
  - E1 → D
  - E2 → E1
  - E2 → E2
  - E2 → D

- Bypass on MEM :
  - M1 → E1
  - M1 → E2
  - M1 → D
  - M2 → E1
  - M2 → E2
  - M2 → D

We have not put a bypass at the input of MEM in order to avoid lengthening the critical path of this stage, which is already long, due to memory accesses.
The operation of the bypass is quite similar to the one we will find in the simple scalar. One point we had to add, however, is the following case If the destination address of EXE_S1/MEM_S1 is identical to the destination address of EXE_S2/MEM_S2 and we have to bypass a data, then we take the bypass of the S2 stage since this data will be the most recent. Once again, we remind you that the priority of our stages is fixed, so S2 will always have the most recent data, and if this constant priority is removed, we must check which pipeline has the most recent data in order to manage this case.

3.5.1.5 Benchmark and conclusion on SS2 implementation

The implementation of SS2 allowed us to detect certain problems that had escaped us on the scalar core, such as when we :

```
1    add x3,x2,x1
2    csrrc x2, csr_register, x3
```

Indeed, in this case the x3 register will be bypassed, nevertheless the csrrc instruction performs a logical and between the value of the status register and the inverse of the value in x3 (inverse in the sense of bit inversion). This is why $\bar{x}3$ must be bypassed and not simply the value in x3.

Finally, in order to measure the performance gain of the SS2 implementation and the scalar implementation, we automated the measurement of the number of cycles required to execute a program. It would have been interesting to measure CPI, but we did not have the time to automate its calculation. Indeed, the calculation of the number of instructions executed in a program is a little more complicated than a simple measurement of cycles.

We nevertheless carried out this measurement of the number of cycles with the help of a shell script generating a file which we then placed in Excel in order to obtain the following results :

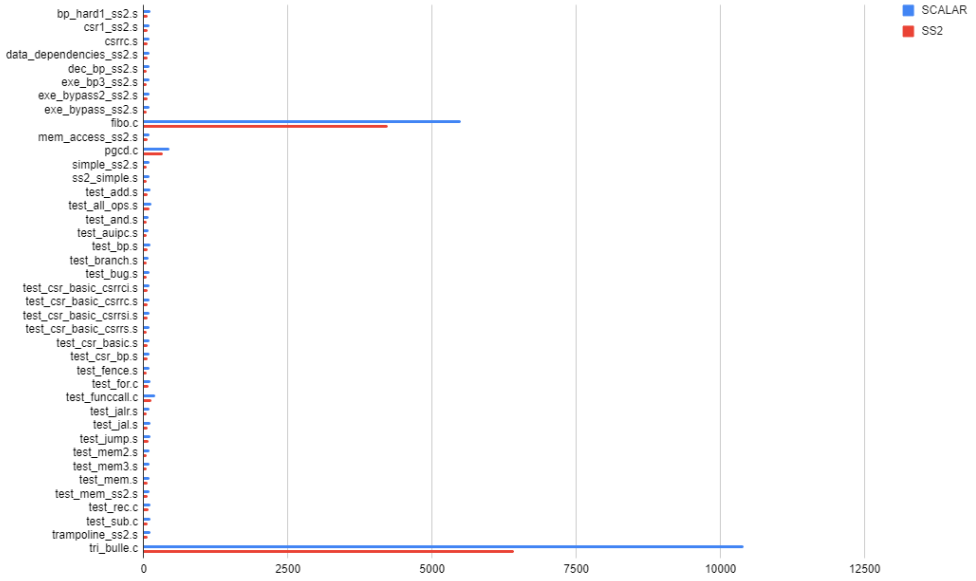


Figure 16: Comparison of the number of cycles per program on custom tests

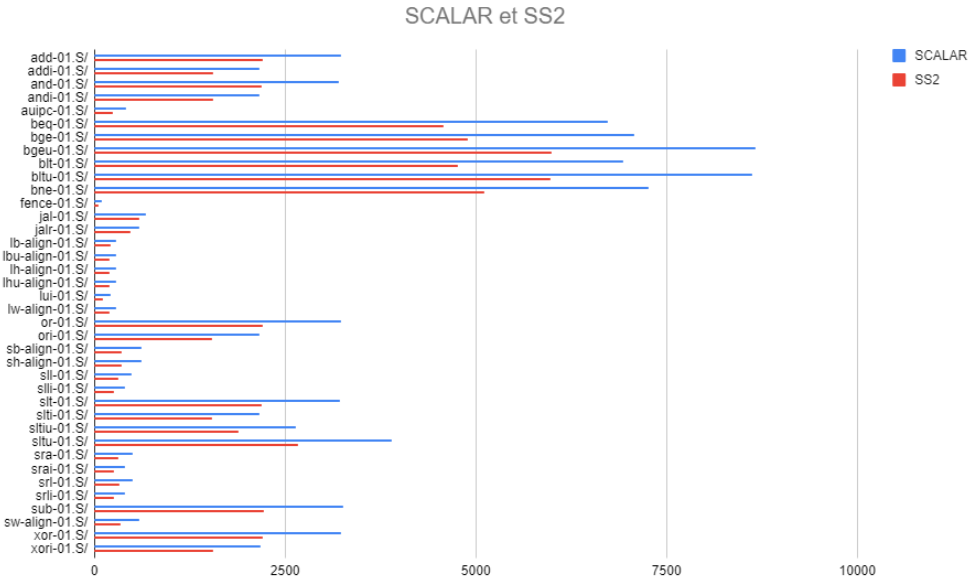


Figure 17: Comparison of the number of cycles per program on the riscof suite

On our custom tests, SS2 is on average 1.62 times faster, while on the riscof suite tests the gain drops to 1.50. This difference is quite logical as our custom assembly tests are quite short, while the c tests are a little longer. This difference is quite logical insofar as our custom assembly tests are quite short, the c tests being a little longer. The riscof tests are much longer and therefore contain much more data dependencies. Indeed, in our programs there is little data dependency, which will imply that SS2 tends to be twice as fast, as there will be no freeze on the 2nd instruction decoded in the DECOD cycle.

If we take the fibo.c test for example, we have a gain of 1.30, which is much lower than the average obtained. The compiler (gcc) does not take into account the fact that we are using a super-scalar architecture and the code is therefore not optimised for this kind of architecture, hence the fact that this gain is much smaller. Indeed, the code contains a lot of data dependencies which leads to many freezes on the loading of the 2nd instruction in DECOD.

We have not performed a critical path measurement on the SystemC model but it might be interesting to do so later to see if there is really a gain from the super-scalar implementation.

3.5.2 Branch prediction

When a branch occurs in DECOD and it succeeds, a clock cycle will be lost. Indeed, it will be necessary to flush the instruction which was loaded in the if2dec fifo and to fetch the instruction located at the address of the branch.
Branch prediction is a mechanism to avoid losing this cycle by anticipating whether the branch will succeed and thus loading the corresponding instructions.

Our branch prediction uses 2 different mechanisms[25]:
- **2-bit Saturating Counter**
- **Return address stack**

3.5.2.1 2-bit Saturating Counter

This mechanism uses an associative table that has the address of a branch as its key and an address and state as its value. The address is where the branch jumps to if it succeeds and the state is the result of the prediction, i.e. whether it failed or succeeded.

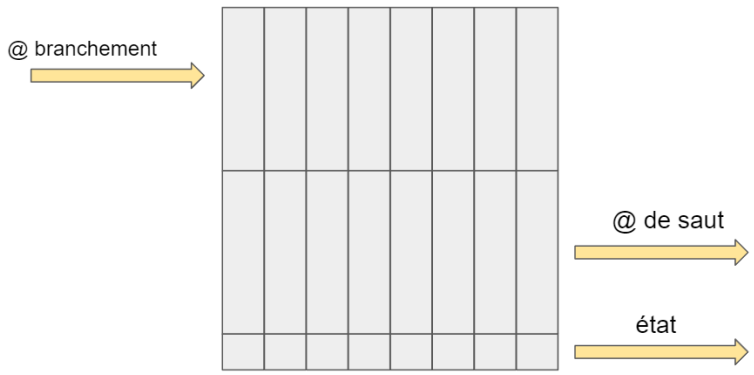


Figure 18: Associative table of the branch predictor

The state stored in the associative array corresponds to one of the states of the state machine [figure 19].
The state of this machine changes when the branch succeeds or fails. A branch will be considered taken when its state is **weakly taken** or **strongly taken**, while it will not be taken when it is in the state **strongly not taken** or **weakly not taken**.

Ifetch detects that the received instruction is a branch, it will be sent to Decod as usual. But before sending it to Decod, Ifetch "looks" to see if the instruction is stored in its associative array.
If it is, then once the instruction is sent to Decod, if the prediction indicates that the branch will succeed, Ifetch will ignore the value of PC contained in dec2if and will fetch the instruction at the address of the branch.

So if the branch also succeeds in Decod, there will be no need to flush the instruction in if2dec and the PC sent in dec2if will be that of branch + 4.

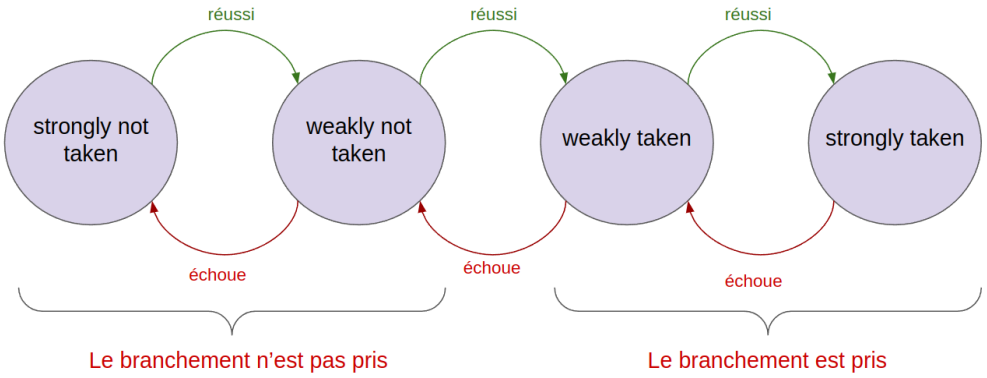


Figure 19: State graph of branch prediction

3.5.2.2 RAS(return address stack)

The branch prediction presented in the paragraph 3.5.2.1 does not make it possible to predict the returns of functions. Indeed, this return address can vary since by convention the RET instruction reads the jump address in the x1.register.

To overcome this problem, we have added a stack to Ifetch called **RAS(return addresse stack)** which will store the return addresses of the functions.

The spec [8] reads :
"A jal instruction should stack the return address on the return address stack only when rd=x1/x5. Jarl instruction should stack/unstack the RAS as shown in the table [figure 20]"

rd	*rs1*	*rs1=rd*	RAS action
!*link*	!*link*	-	none
!*link*	*link*	-	pop
link	!*link*	-	push
link	*link*	0	push and pop
link	*link*	1	push

Figure 20: Conditions for stacking and unstacking the RAS

This table lists the stacking and unstacking conditions of the RAS, we can see for example that in the case where we have a destination register and a source register used with the jalr instruction and that this source register is equal to the destination register and that it is 0, then it will be necessary to both stack and unstack.

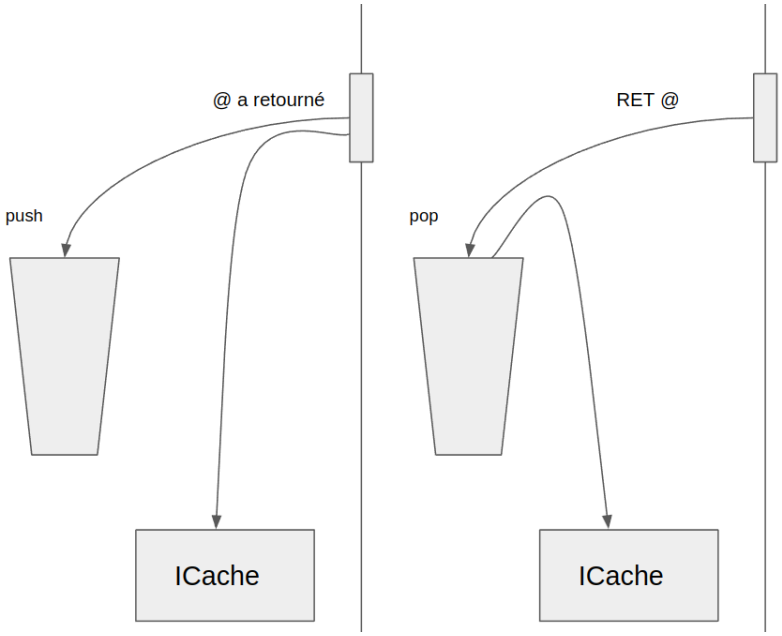


Figure 21: RAS implementation in IFETCH

3.5.2.3 Results obtained

In order to determine the gain from branch prediction, we made comparisons between the number of cycles required to run a program with and without branch prediction. The results are presented in table 3.5.2.3.

Test programm	Nb of cycles without opt.	Nb of cycles with branch pred. without RAS (PB)	% gain	Nb of cycles with PB and RAS	% gain	Nb of successful branch
div_2.c	403	378	6.2%	378	6.2%	31
fibo.c	5499	5223	5.0%	5047	8.2%	677
pgcd.c	49283	48396	1.8%	48396	1.8%	1783
factoriel.c	326	323	0.9%	323	0.9%	11
test_funcall.c	197	194	1.5%	192	2.5%	13
test_jump.s	120	119	0.8%	119	0.8%	9
tri_bulle.c	10405	10238	1.6%	10238	1.6%	456

∗The tests were carried out without the caches.

We can see that we gain about 2.6% of cycles with the branch prediction without the stack mechanism. With the stack mechanism, the gain this time is 3.2%. We notice however that we gain nearly 3% of cycles on the recursive fibonacci sequence with the stack mechanism. This seems coherent since the program is recursive, it makes a large number of function calls and therefore a large number of function returns.

3.6 Validation protocol

3.6.1 Basic tests

To validate our implementation, we worked as follows :
- Firstly, we carried out bench tests (cf GitHub) for each of the stages in order to verify their operation. These tests consisted of sending signals with random values to the tested stage and watching what we obtained in output.
- In a second step, we compiled some fairly simple assembly programs with one or two instructions, we then made the programs more complex and designed tests with all instructions of the same type. We have for example a test called test_all_ops which tests all the instructions of type I
- Finally, we wrote some C programs like the Fibonacci sequence or a PGCD algorithm that we compiled and we checked that everything worked correctly.

In order to carry out all our tests, we used Gtkwave to visualize the signals of our entities, we indeed created trace() functions in each of our files which trace all the signals of the same entity in a .vcd file which can be visualized in Gtkwave.

Finally, to compile code directly, we used the C++ ELFIO library which allows to parse an ELF file. To use this library, our core_tb.cpp file takes as argument a .s or .c file, it then calls the RISCV compiler and produces an objdump as well as an executable.
This executable is then parsed using ELFIO and we thus recover the instructions which are stored in our RAM, which is simulated by a map which forms a couple (address, instruction).
We also have thebad andgood segments on which we jump at the end of the program to see if everything went well.

Here is an example of one of our test programs, the recursive Fibonacci recursive:

```
extern void _bad();
extern void _good();

__asm__(".section .text") ;
__asm__(".global _start") ;

__asm__("_start:");
__asm__("addi x2,x0, 0x100");
__asm__("addi x1,x1, 4");
__asm__("sub x2, x2,x1 ");
__asm__("jal x5, main");


int fib(int n) {
    if (n == 0) {
        return 0;
    }
    else if (n == 1) {
        return 1;
    }
    else {
        return fib(n-1) + fib(n-2);
    }
}



int main() {
    if (fib(10) == 55) {
        _good();
    }
    else {
        _bad();
    }
}
__asm__("nop");
__asm__("_bad:");
__asm__("    add x0, x0, x0");
__asm__("_good :");
__asm__("    add x1, x1, x1");
```

3.6.2 Exception handler

When we started to implement the machine part, we had to modify our test codes in order to have a code acting as an exception handler. Indeed, when an exception is detected in MEM, we will load the value of MTVEC in PC, a register which contains the address where the exception handler is located. Wanting to get as close as possible to MIPS, we set this address to 0x8000 0000, the problem being that nothing was initially at this address.
To remedy this program, we added two linker scripts : **app.ld** and **kernel.ld** defining a text section and a Kernel section, it's indeed Mr. Franck Wajsbürt who advised us to do like that in order to make a distinction between user script and machine script. These two linker scripts are identical except that app.ld ends with an INPUT(kernel) instruction.
The interest of having two linker scripts is the following, the kernel.ld file allows to generate an object file named **kernel** which contains the reset code and the exception handler which we will link with the user file during the compilation of the test file.
When compiling the test or in other words the user file, we will use the app.ld file, the INPUT(kernel) instruction allowing to force the compiler to generate an elf file presenting both the user code and the machine code.

```
SECTIONS
{
    . = 0x10054 ;
    seg_text :
    {
        *(.text)
    }
    . = 0x80000000 ;
    seg_reset :
    {
        *(.reset)
    }
    . = 0x81000000 ;
    seg_kernel :
    {
        *(.kernel)
    }
}

INPUT(kernel)
```

3.7 Official riscof test suite

To better validate our processor, we wanted to use a more complete and external test suite (to avoid being biased in our tests). We chose to use the Riscv-V Compatibility Framework. This is a test program that compares a "signature" (a segment of memory) between two implementations of RISC-V on a whole battery of tests.
We made this choice, because the suite is very complete (several hundred unit tests for most instructions), and it has an "official" value. Indeed, although it is not developed by the risc-v foundation, but by a private company (incore), validating this test suite is a prerequisite to be approved by the risc-v foundation and to be able to use the "risc-v" trademark.
To make this test suite work, we had to make a "plugin" allowing to connect the test framework with our program, as well as a tool to dumper the memory in a file.
We also had to modify our test-bench, to read the sections and symbols defined in the tests, such as the beginning of the test, the end of the test, the beginning of the memory area to be dummied and the end of it.
We then chose the "spike" implementation as the reference implementation for the tests.
These tests allowed us to correct many bugs: many instructions with borderline cases that did not react correctly.

4 SoC

After finishing the branching optimization through branching prediction, simultaneously with the development of the SS2 core, it was decided to model a complete SoC in SystemC and VHDL. Indeed, the addition of a system bus was necessary for the addition of peripherals for the implementation on FPGA boards.
The first step of this implementation was the choice of a BUS protocol.
After studying several types of bus (PiBus, AXI4, Avalon, etc.), we decided to go for the Wishbone B4 protocol that we discovered during the FSIC2022 conferences.
This protocol is open source, simple to implement and very well documented [17]. The specification offers us different choices for the type of interconnection, for example "Point-to-Point" or "crossbar switch". We chose "shared bus" because it was the closest to the PiBus that we studied in the 2nd semester of M1 in the "Multi" class.

We chose a system **round-robin** for the arbiter, which will help us to share the bus well between the cores. The arbiter will also, according to the address sent to the bus, choose the target which will process the memory accesses.

We have placed a total of two initiators (two RiVer cores) and one target (the RAM) on the bus.

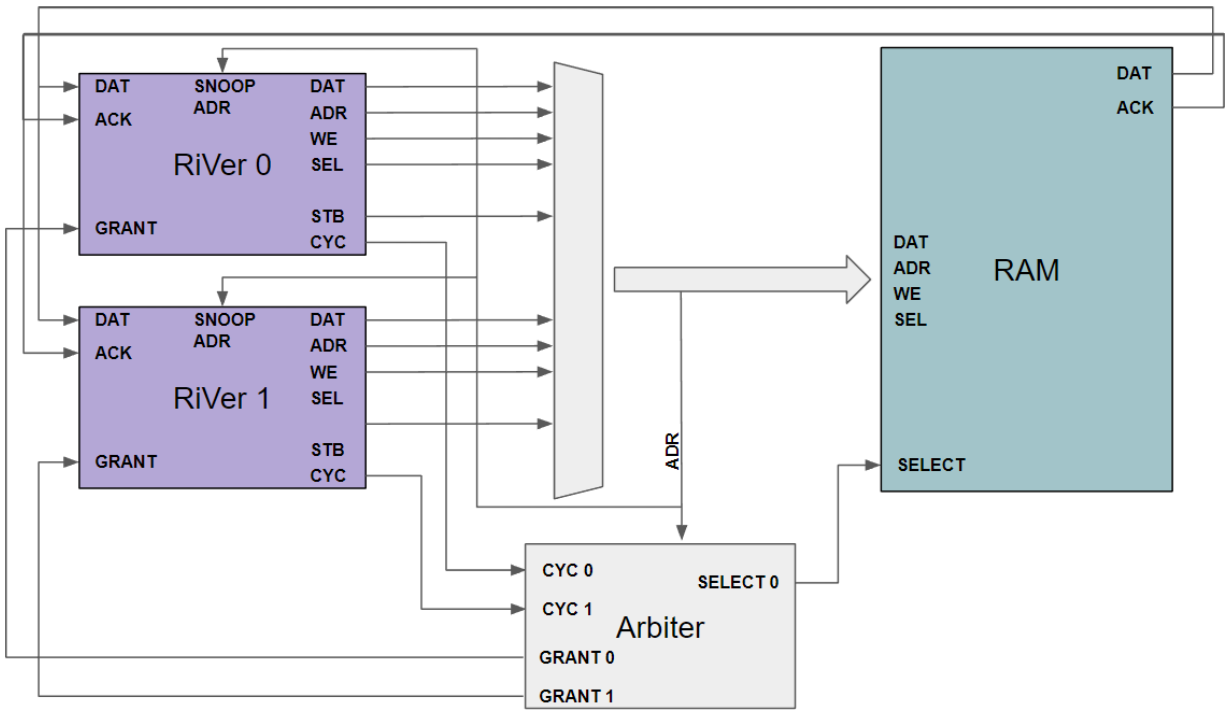


Figure 22: SoC diagram

4.1 Master RiVer

The RiVer initiator consists of the following elements:

1. A RiVer RV32IM core
2. 2 caches (instruction cache and data cache)
3. the wrapper

Caches contain a "snoop" mechanism, to ensure data coherency, which will be detailed in section 4.1.1.

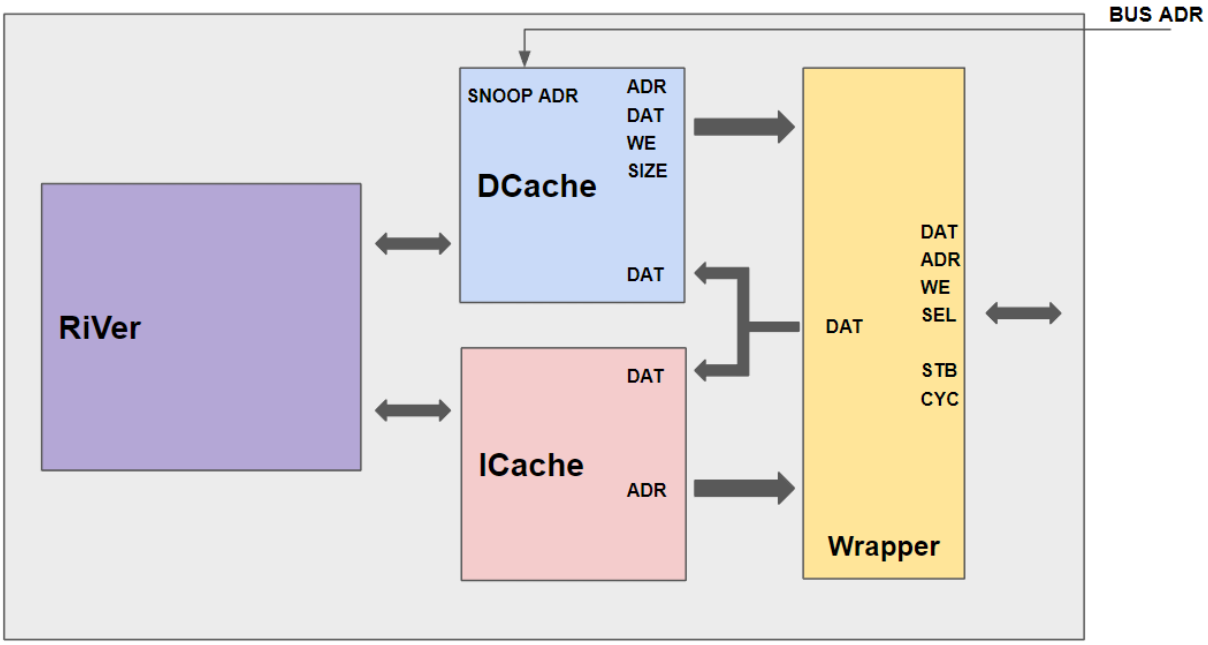


Figure 23: Diagram of the initiator RiVer

4.1.1 Caches

After much discussion among our group members, we decided to implement separate L1 caches. Indeed, caches allow us to reduce the CPI (cycle per instruction) by limiting memory accesses. Moreover, we have spent almost 2 semesters studying the functioning of caches, hence our interest in implementing them.

4.1.1.1 Instruction cache

The instruction cache is a cache composed of 256 rows of 4 columns. Each cell can contain one word (32 bits) so the total capacity of our cache is 32Kb.

To design the state machine, we defined 3 distinct states. Indeed, after several tests of different state machines, the most optimised one we managed to implement is composed of the following states:

- IDLE
  The default state that responds to processor requests and in case of MISS makes a request to the upper memory level.
- WAIT MEMORY
  This state waits for a response from memory.
- UPDATE
  This state is responsible for receiving data from the bus and loading it into the corresponding cache lines.

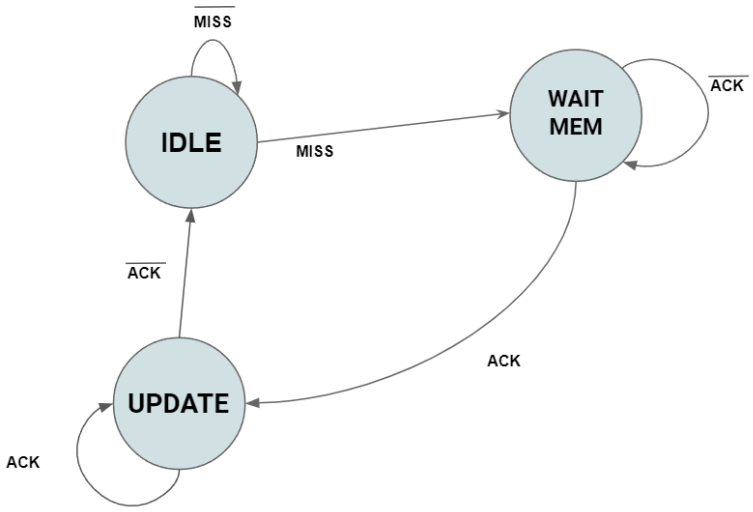


Figure 24: State machine of the insturction cache

4.1.1.2 Data cache

The data cache is a 2-way associative cache, each way containing 128 rows and 4 columns, for a total capacity of 16Kb per column. This cache has been designed in accordance with the write-through policy. This strategy will allow us to solve some memory coherence problems.
In order to improve the performance of the processor, we have added a 2 place buffer-cache, which allows us to reduce the number of stalls created by the cache in case of a write.

The data cache state machine is the same as the instruction cache state machine, except that the data cache state machine handles write and read requests :

1. IDLE
   Default state that responds to read requests from the processor. In case of a write, it writes the request to the cache buffer and in case of a read miss, it writes the request to the cache buffer and will switch to the WAIT_MEM state to wait for the RAM response.
2. WAIT MEM
   This state waits for the RAM response (ACKNOWLEDGE) and updates the first data it receives.
3. UPDATE
   This state is responsible for receiving the remaining data sent on the bus and putting it into the corresponding cache lines.

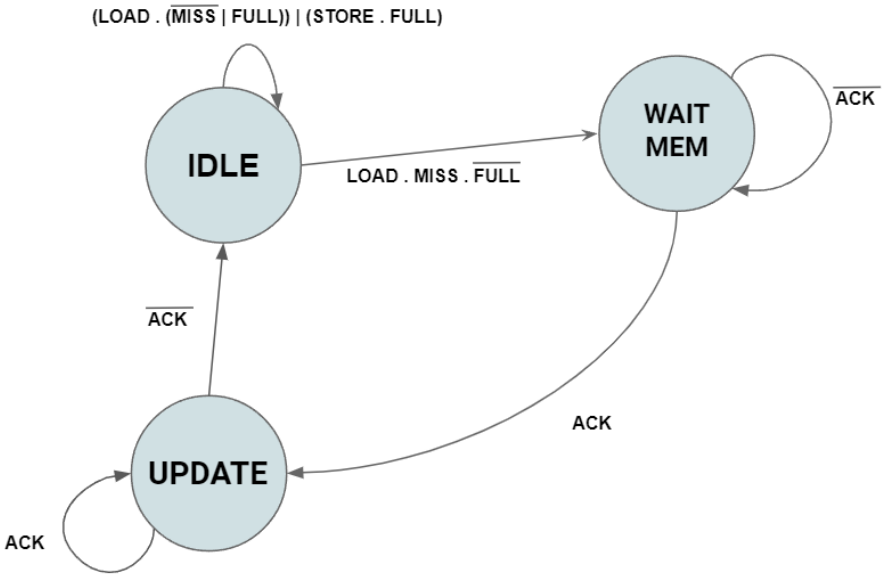


Figure 25: State machine of the data cache

4.1.1.3 Snoopy mechanism

In order to design a multi-core system, memory consistency must be ensured.
Our architecture uses write-through caches with a posted write buffer. The presence of these buffers disturbs

memory coherence. Indeed, if a core performs successive writes, the data will only be present in its L1 cache while it is being written to memory. The problem is that if another core tries to manipulate these addresses, it must be aware that the other core has performed a write.

This is why the presence of a monitoring mechanism is necessary. Snoopy ensures this consistency by reading the addresses on the bus. If a write is made and the data is present in the cache or buffer-cache, Snoopy will be in charge of invalidating the line or the data.

4.1.2 Wrapper RiVer-Wishbone B4

The role of the wrapper is twofold. Firstly, it translates the requests sent by the caches into requests that can be understood by the Wishbone protocol.
Secondly, it manages the transmission of data on the bus, through state machines.

The state machine takes into account the fact that the instruction cache and the data cache can make a request to the same cycle. Thus, this MAE favours requests from the data cache in order to raise a pipeline stall more quickly, thus freeing the MEM, EXE and DEC stages.

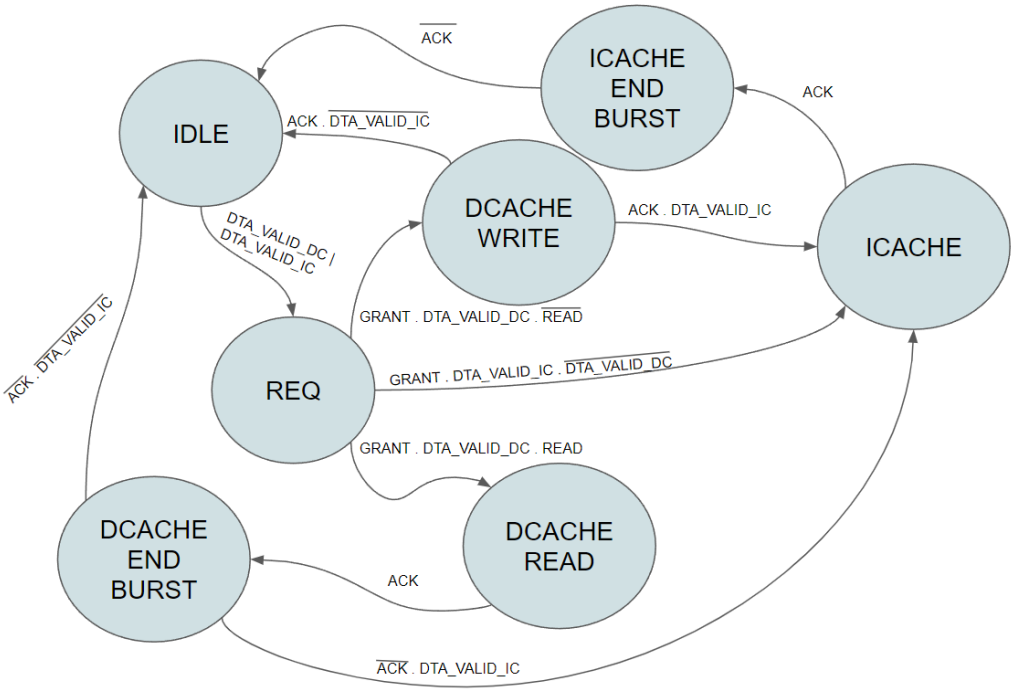


Figure 26: River wrapper state machine

The states of this FSM are as follows

- **IDLE** :
  Default state, waiting for the instruction cache or data cache to send valid data,
- **REQ** :
  Request for access to the bus by an initiator,

- **DCACHE WRITE** :
  Request to write the data cache. When the target has responded (**ACKNOWLEDGE**) and the data is valid, it passes into the **ICACHE** state, otherwise returns to the **IDLE** state,
- **DCACHE READ** :
  Request to read the data cache. Once the request is made, it waits for the target's response (**AKNOWLEDGE**). When the response arrives, it enters the state **DCACHE END BURST**
- **DCACHE END BURST** :
  The wrapper continues to read responses from the target as long as the **AKNOWLEDGE** signal is in the high state,
- **ICACHE** :
  Write the instruction cache request to the BUS and wait for the target to set the **ACKNOWLEDGE** signal high. When the signal is raised, it transmits the first response to the instruction cache which will go into the state **ICACHE END BURST**
- **ICACHE END BURST** : The wrapper continues to read responses from the SLAVE as long as the ACKNOWLEDGE signal is in the high state. When it returns to the low state, it returns to the **IDLE** state.

4.2 Targets

The SystemC bus has only a few targets, in fact, there is only the RAM which contains 2 elements:

1. An unordered map which acts as a ram
2. A wrapper which allows to translate the requests to make them understandable by the bus

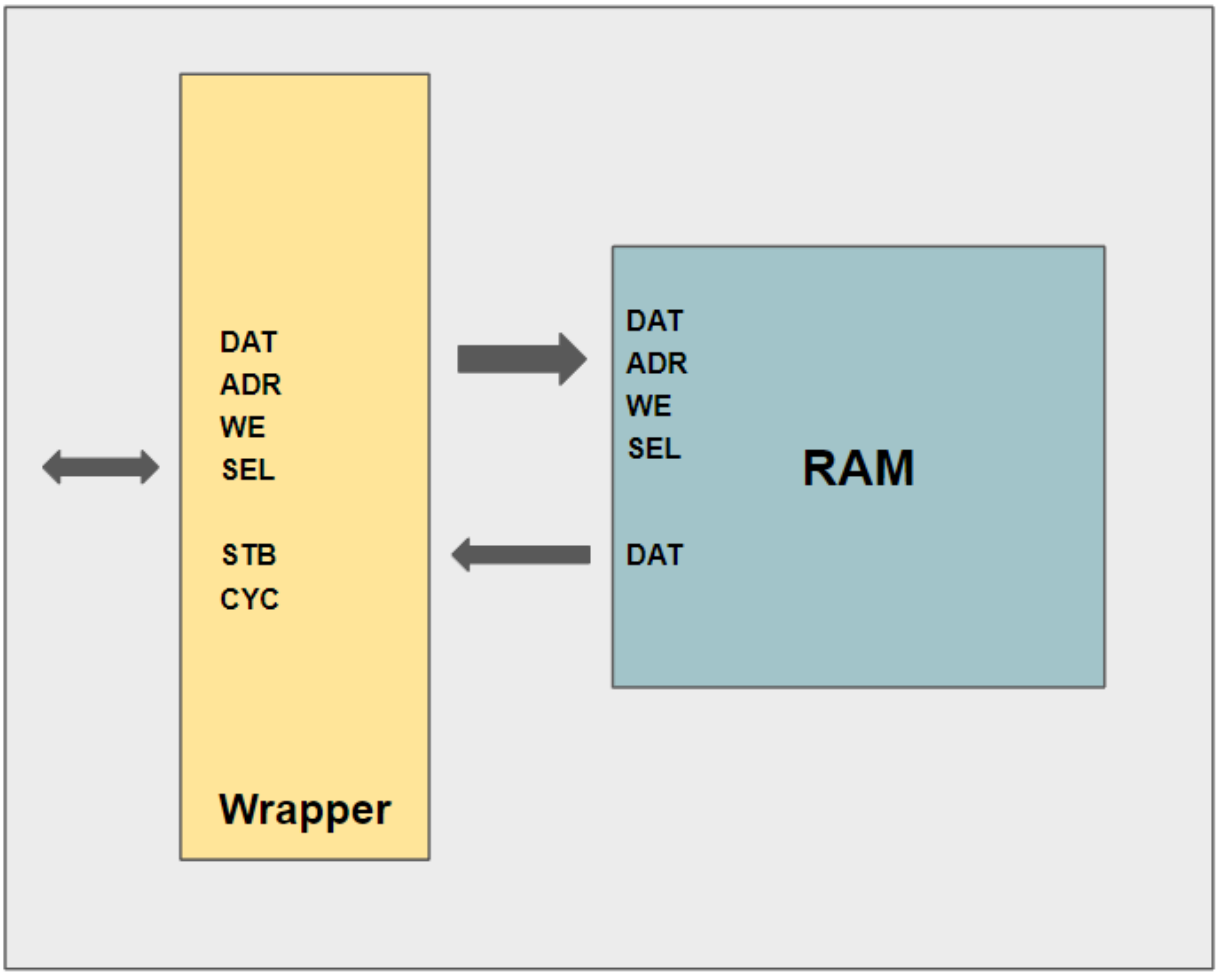


Figure 27: Diagram of the target RAM

The wrapper reads requests from the bus and translates them into one or more requests for RAM.

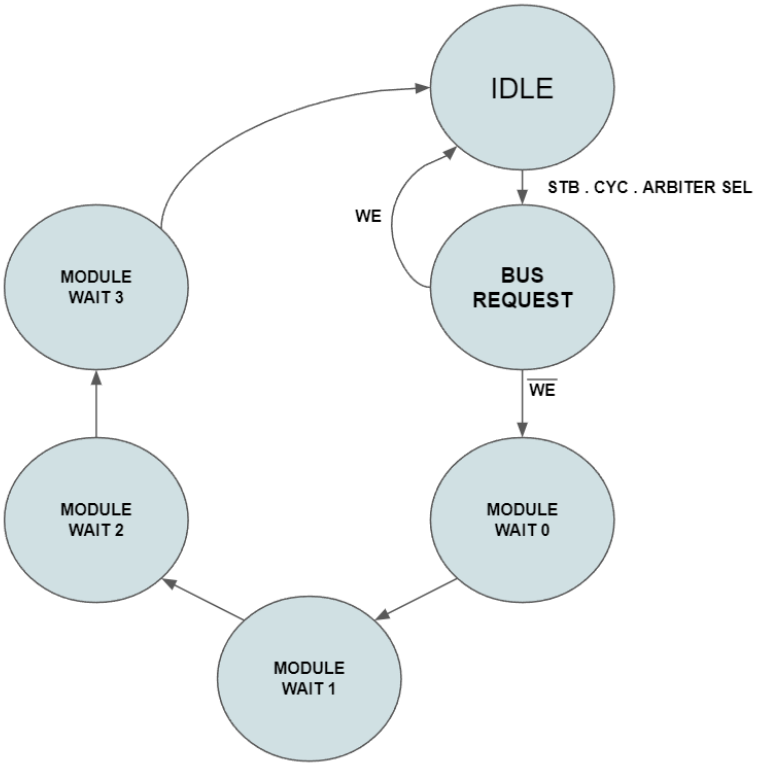


Figure 28: FSM of the wrapper RAM

The wrapper is controlled by a MAE, of which the following is a brief description:

- **IDLE** :
  Default state which waits for the address comparator (or in our case the arbiter) to choose it to respond to a bus request,
- **BUS REQUEST** :
  Processes the bus request, two cases are possible **Read** Sending of the first address (ADR+0) by the wrapper and passage into the state **MODULE WAIT 0**,ì **Write** Sends the data to be written as well as its address by the wrapper, then returns to the **IDLE** state,
- **MODULE WAIT 0**:
  Reads the RAM and writes the response to the bus, then sends the next address ADR+4,
- **MODULE WAIT 1**:
  Reads the RAM and writes the response to the bus, then sends the following address ADR+8,
- **MODULE WAIT 2**:
  Reads the RAM and writes the response to the bus, then sends the following address ADR+12,
- **MODULE WAIT 3**:
  Reads the RAM and writes the response to the bus, then returns to the Idle state,

5 FPGA implementation

We decided to implement our processor on FPGA in order to test it physically.
We chose the Nexys A7 [20] board from Xilinx with Vivado 2020.2 with which we are used to work in FPGA practical training.
But in order to carry out this implementation, we must first translate the SystemC model into VHDL. Indeed, SystemC being initially a language designed for high-level simulations and not RTL descriptions, we did not find a tool allowing us to translate our SystemC implementation into VHDL. This is why Mr. Samy Attal had the responsibility of translating the whole project manually.
In this chapter we will discuss the main differences between the VHDL and SystemC model and the FPGA implementation.

5.1 VHDL description

The core described in VHDL is almost similar, in terms of performance, to that in SystemC. We therefore have a RV32IM-Zicsr VHDL core with branch prediction which has been validated by the riscof framework.
We used GHDL [18] as a VHDL compiler and simulator as well as GTKwave [19] to visualise the signals.
The choice of these tools was made in order to stick to what could be done in SystemC, the names of the signals are almost identical, but obviously differences in their assignments are present.
The choice of GHDL is also justified by the fact that it is possible to link a C program to the VHDL testbench in order to simulate and fill a RAM in C from an executable (much easier to do than in VHDL).
The major difference being the divider state machine, in fact, it is a Moore machine in the VHDL implementation, whereas in SystemC it is a Mealy machine (faster) which is difficult to implement in the VHDL core (although functional alone). The problem in the implementation is related to the start/stop conditions of the MAE and the push/pop conditions of the fifo in post-synthesis.

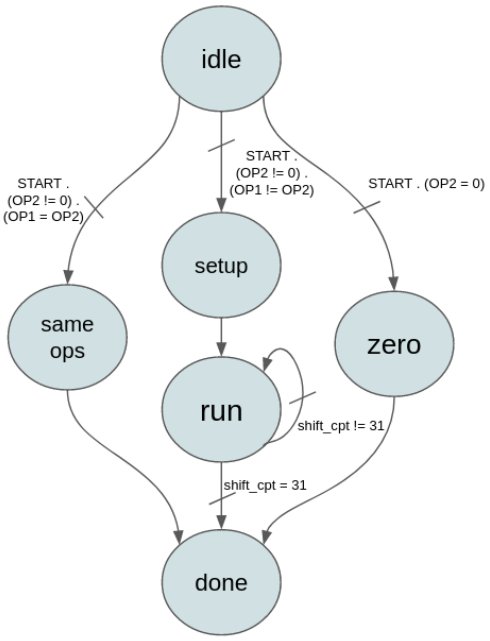


Figure 29: Moore FSM of the divider(VHDL)

So we have a first wait state, where when a divide instruction enters the EXE stage it will compare and check the values of the operands. We then have two special cases that we can separate from the standard case in order to save cycles:

- operand 2 (the denominator) is null, we then pass into the "zero" state which will give as result 0xFFFFFFFF (-1) as specified in the RISC-V documentation.

- If the two operands are identical (and the denominator is non-zero), then the "same ops" state is used, which will give the result as 1 if it is a division, or 0 if it is a modulo.)

Otherwise, we go into the "setup" state which will load the registers represented in figure 10 with the correct values, and the "run" state will allow the calculation to be performed (as in the SystemC model).

5.1.1 Interface C/VHDL

To create a complete testbench, close to the one found in SystemC, we have created an interface between GHDL and a C program. This interface defines functions that can be called in a VHDL program and that will allow to interact with a RAM simulated by an integer array.
In order to carry out this interface we took again a code which had been provided to us by Jean-Lou Desbardieux at the time of our 1st semester of M1 within the framework of the UE VLSI. This program allows to parse an elf file and to extract the instructions and address of this last one.
We were indeed not able to use the ELFIO library that we used in SystemC since the latter is written in C++ and we did not find a way to make a C++/GHDL interface.
On the other hand, the interest of the parser we used is that it is written in pure C and does not require anything apart from the **elf.h** library which is a basic linux library.
Once the parser was simplified and adapted for our use, we wrote the program **ram_sim.c** which defines the RAM, places the instructions in memory and defines functions, callable in VHDL, allowing to write or read the RAM.
We have also adapted this program so that it is able to run the riscof framework necessary for the validation of our implementation.

5.2 FPGA

5.2.1 Circuit design

Our FPGA implementation was carried out in several stages.
The first step consisted in testing the correct operation of the processor at the implementation. We had associated a memory array simulating the instruction cache (written in VHDL) which contained the instructions to be executed as well as a memory array simulating the data cache whose low-order bits of one of the memory cells were connected to LEDs present on the board in order to be able to debug.
This step is shown in figure 30.

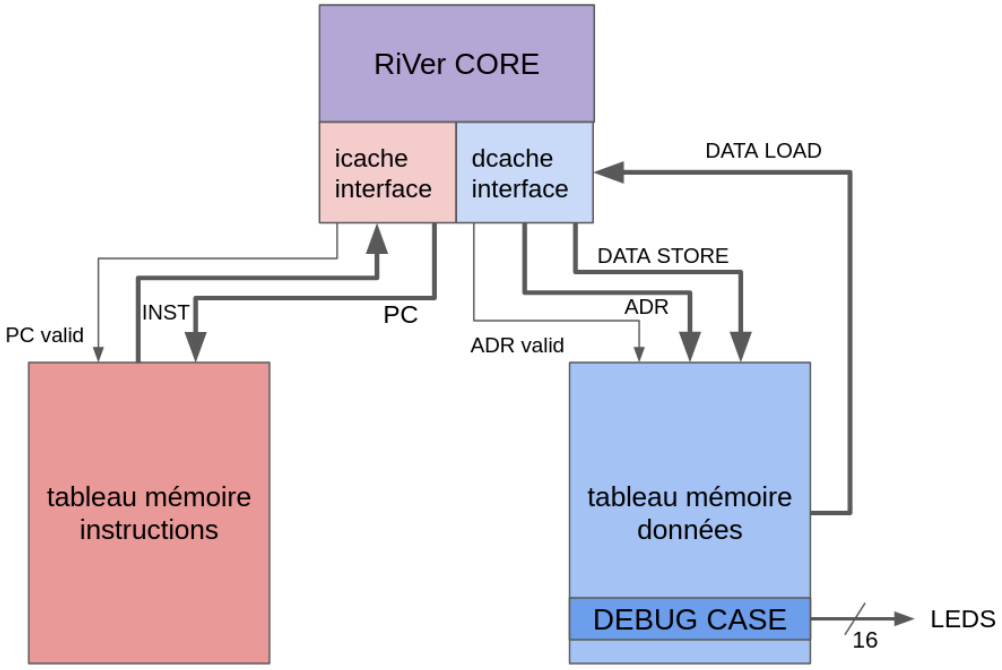


Figure 30: Basic implementation on FPGA

This is how we were able to test each version of the processor in post-synthesis up to the final version (RV32I, RV32I-Zicsr, RV32IM-Zicsr, RV32IM-Zicsr with connection prediction).

The second step, after having validated the correct operation of the core, consisted in integrating the processor on a bus with a RAM. We decided to implement the Wishbone bus, being open source and easier to implement. Knowing that it would be quite simple to convert the interfaces of our Wishbone-compatible components to another bus (like AXI4). This makes it easier to port our processor to different targets and FPGA families without fearing the lack of AXI4 bus support if our target is not from Xilinx, or Avalon if our target is not from Altera.

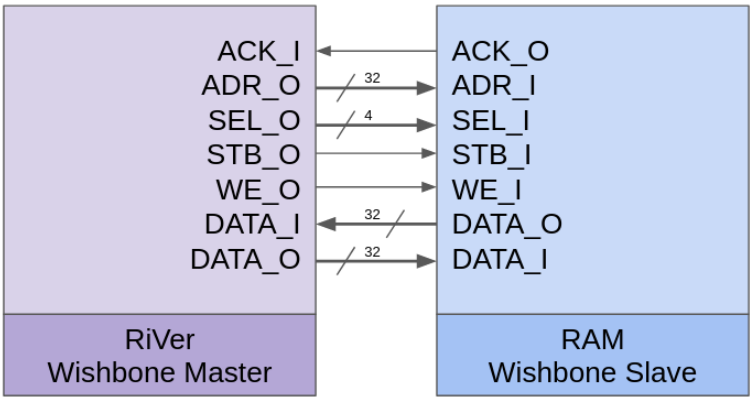


Figure 31: Wishbone CPU and RAM

The interfaces of the Wishbone wrappers used in our system are represented in figure 31. Note that in the context of our FPGA implementation, we have chosen for simplicity to implement only the single write / single read mode present in the Wishbone B4 specification. An improvement towards a pipelined or block transaction mode is quite possible in order to move towards what has been previously done in the SystemC model.
The RAM is at this stage simply an array described in VHDL (easier for debugging).

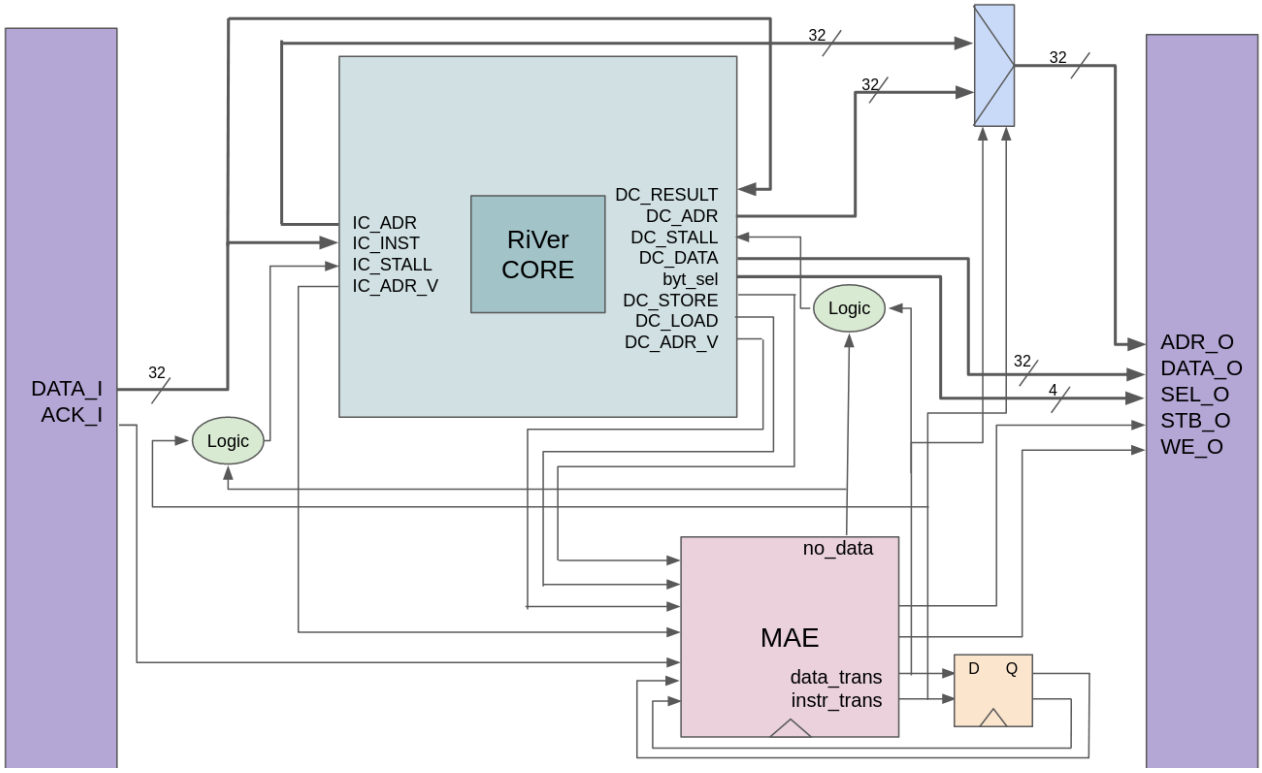


Figure 32: Interface core/Wishbone bus

Figure 32 represents how we interfaced our processor to the Wishbone bus. This is the wrapper.

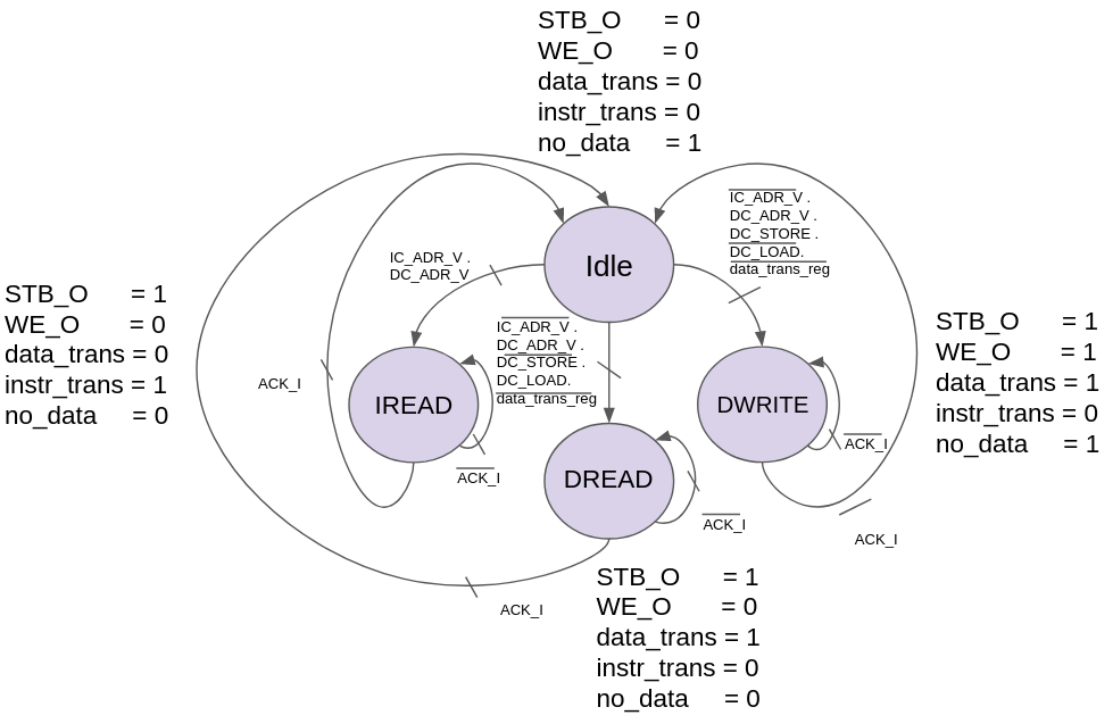


Figure 33: Moore FSM of the core Wishbone wrapper

The MAE, shown in figure 33, controls in addition to the STB_O (to initiate a transaction) and WE_O (to command a write) signals used in the Wishbone protocol. The no_data, data_trans and instr_trans signals which are used to tell the ifetch and mem interfaces whether or not they are recipients of the available data from the bus.

Once the correct operation of the bus was verified with a master (the processor) and a slave (the RAM) we decided to create and connect to the bus a peripheral that would allow us to control the LEDs directly present on the card (the Nexys A7 contains sixteen LEDs as well as two RGB LEDs). This IP could also be customisable (by the number and type of LEDs it controls) in order to be more easily adapted to any FPGA development board target.
Figure 34 illustrates the implementation of this device.

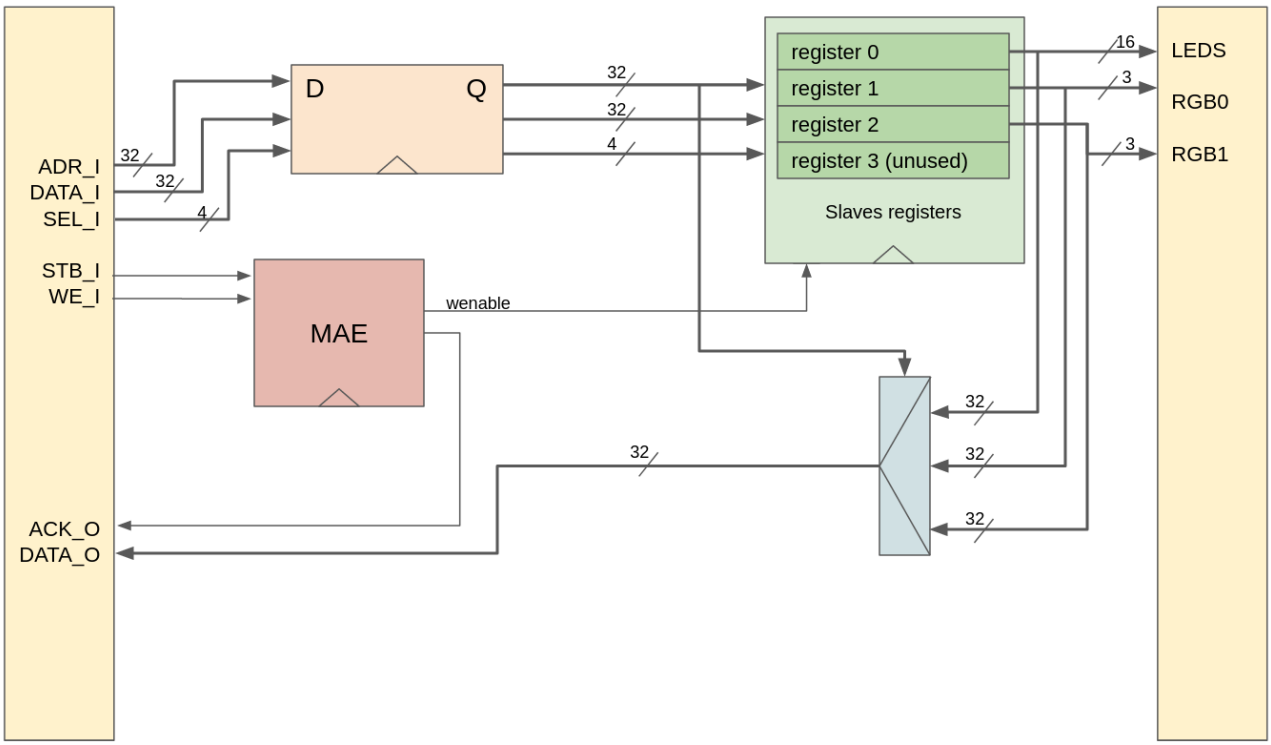


Figure 34: Peripheral LEDs

We have here 4 addressable registers (minimum number of registers to ensure alignment in the addressing table) on the bus accessible in read and write (initially the base address was chosen to be 0x40000000 but this is quite configurable). The last register is not used.
We also find a D flip-flop to save the address values, data and byte selection. In order to synchronise the master request with the MAE signals.
This MAE will then be used to authorize or not the writing in a register (in case of a memory access in writing) and to put at the high state the ACK_O signal in order to inform the master that its request was treated.

In the figure 35 we find the transition graph of the LEDs wrapper MAE as well as its generation function.

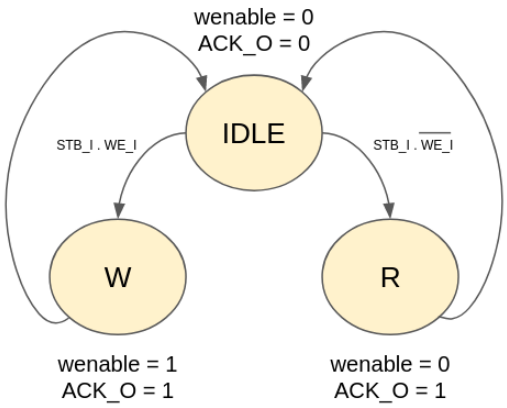


Figure 35: Moore's FSM of the Wishbone wrapper of the LEDS device

Here is a schematic of the complete implementation on FPGA. Of course, thanks to the bus, it is possible to add as many Wishbone compatible peripherals as one wishes, the single read/write mode being normally natively compatible with all Wishbone slaves (One would then have a system containing one master and several slaves).
The current system does not allow for multiple masters on the bus. You would need to add ports GNT_I and CYC_O, and a bus arbiter for this purpose. However, it is still possible to add several cores in the same master with an arbitration which will be done internally of the component and not on the bus.

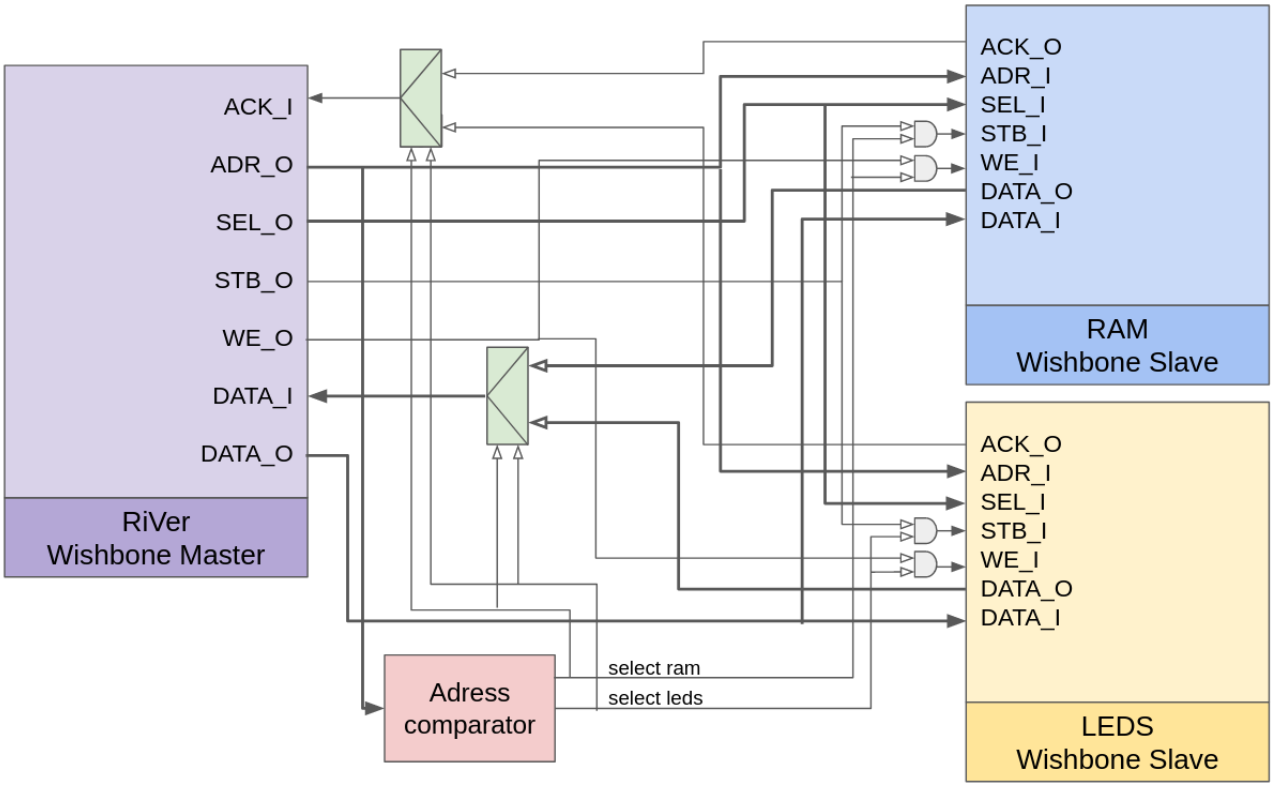


Figure 36: Wishbone CPU, RAM and LEDS

The "Address comparator" block allows, as shown on the diagram, to select the slave according to the address issued by the master (each slave having a base address).
As the RAM is still at this stage a memory array described in VHDL, it is quite complicated to pre-fill it with instructions or data (unless you do it by hand, which is not necessarily practical). In the next section, we will explain how we were able to overcome these problems and optimise the use of memory resources on FPGAs.

5.2.2 RAM configuration

We have chosen to use an IP from Xilinx : Block Mem Generator [21] which will allow us to use in an optimal way the BRAM resources (Block RAM) present in the Xilinx boards, as well as to initialize it via a "Coefficient file" which contains the values in hexadecimal that we want (namely the data and instructions).
In order to achieve this we need to slightly modify the Wishbone interface of the RAM, which thus becomes a RAM controller.

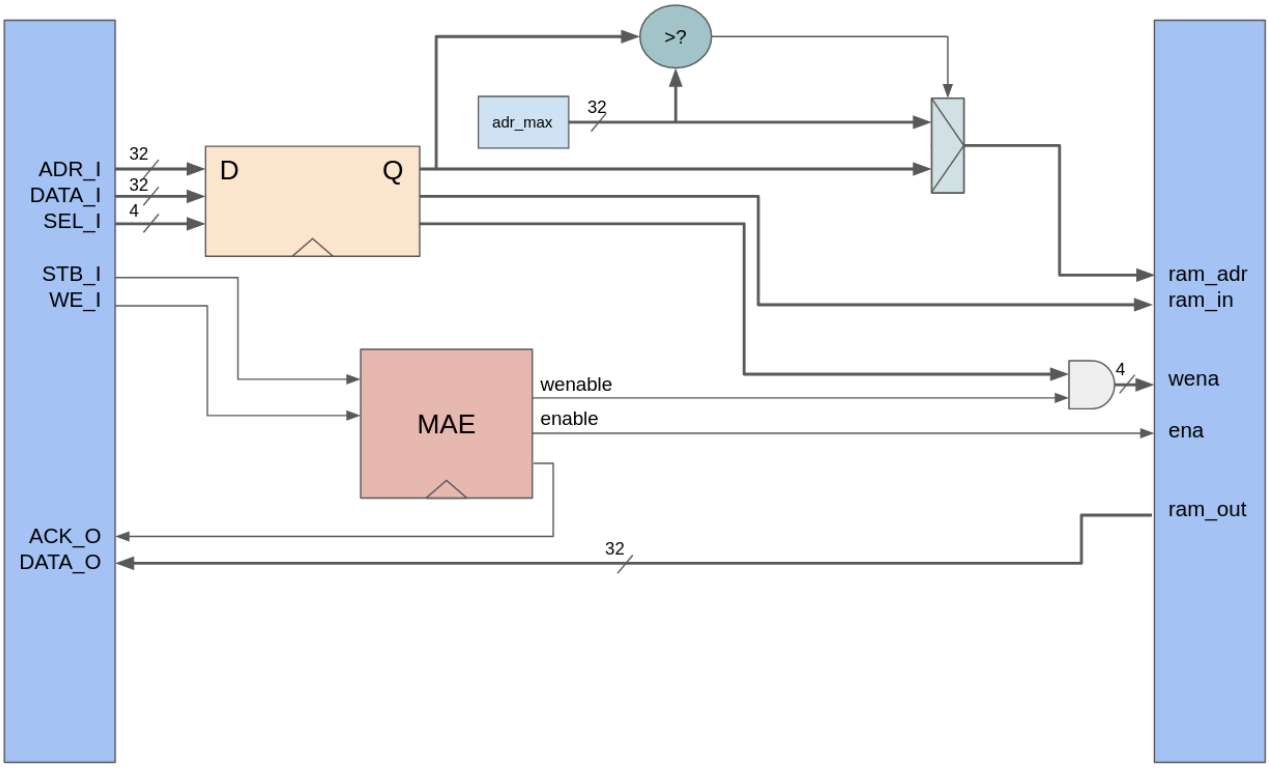


Figure 37: Wishbone RAM controller

We have on the figure 37 the diagram of the RAM controller which consists of a D flip-flop which will save the request of the master and allow the synchronisation between the request (address, data, selection of bytes) and the MAE.
We have in addition to the address, the input and output of the RAM (ram_adr, ram_in, ram_out) two signals:

- wena : write enable, on 4 bits which will allow to select the bytes to write.
- ena : enable, which will activate the RAM for either writing or reading.

adr_max being a constant defined with the choice of the size of the RAM (given that it is not possible with the resources on our card to have the 4 Gb addressable by 32 bits of the RAM) and thus to avoid addressing a memory zone which does not exist.

Figure 38 represents the transition graph and the generation of the outputs of the RAM controller MAE. There are 4 states:

- IDLE : waiting state
- r : read from RAM
- w : write to RAM
- done : end of the operation

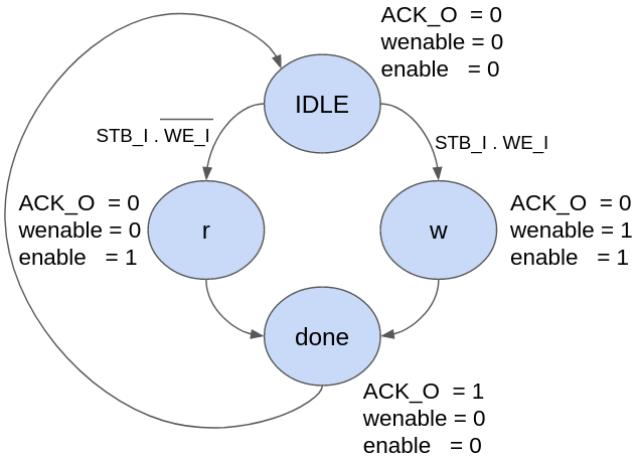


Figure 38: Moore's MAE of the RAM controller

Our FPGA implementation is shown in Figure 39, so we have connected our processor to the Wisbbone bus along with two peripherals: a RAM and LEDS.

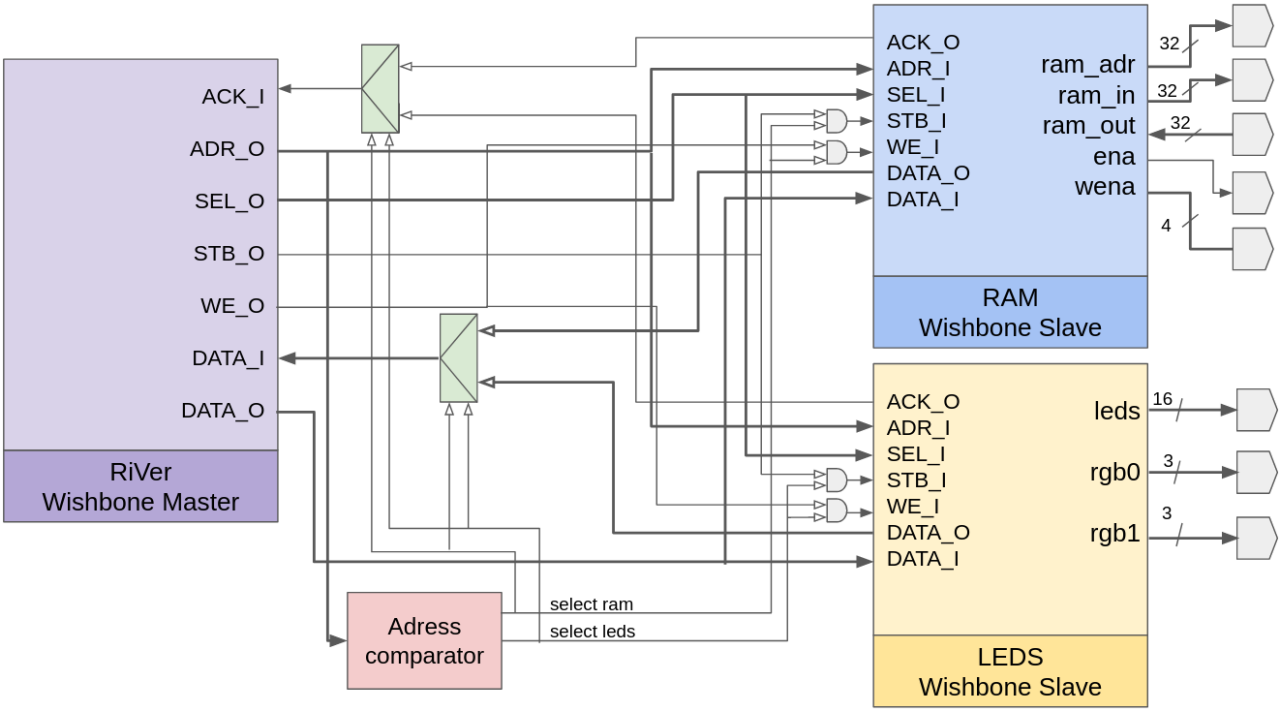


Figure 39: Final implementation on FPGA

We observe on the figure 40 the block design on Vivado. We have the top, which is the RTL (VHDL) description of the whole previous figure, and the IP Block Memory Generator which we can configure in size as well as many parameters which are not of interest to us in the context of our implementation (such as the addition of registers in order to make pipelined accesses to memory)

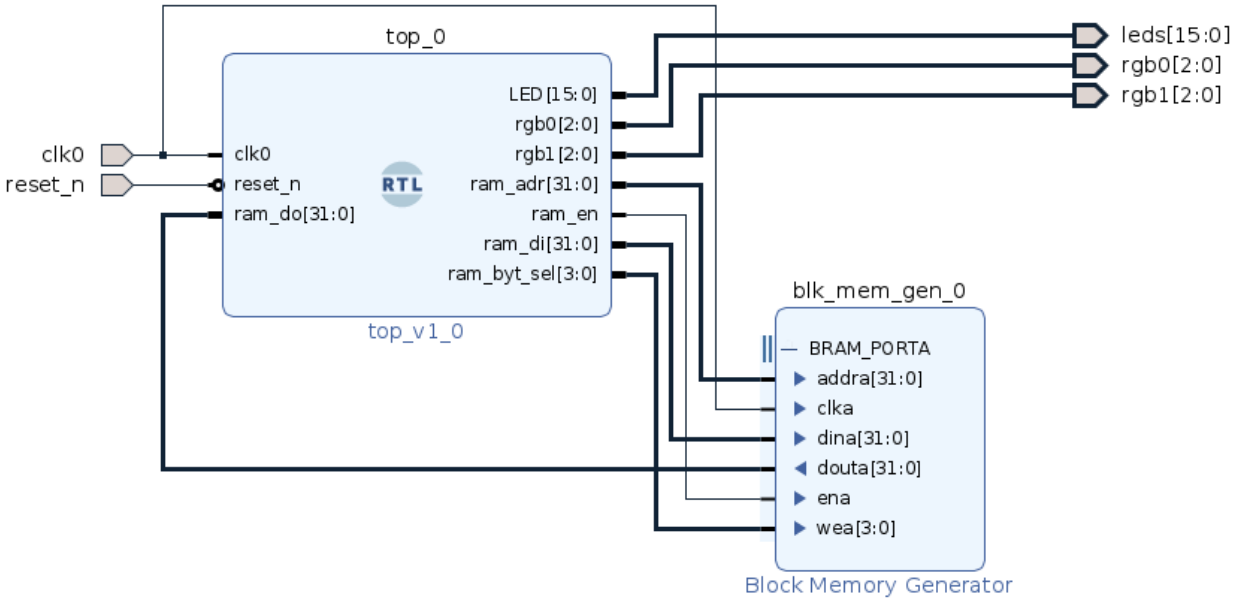


Figure 40: Conception de blocs Vivado

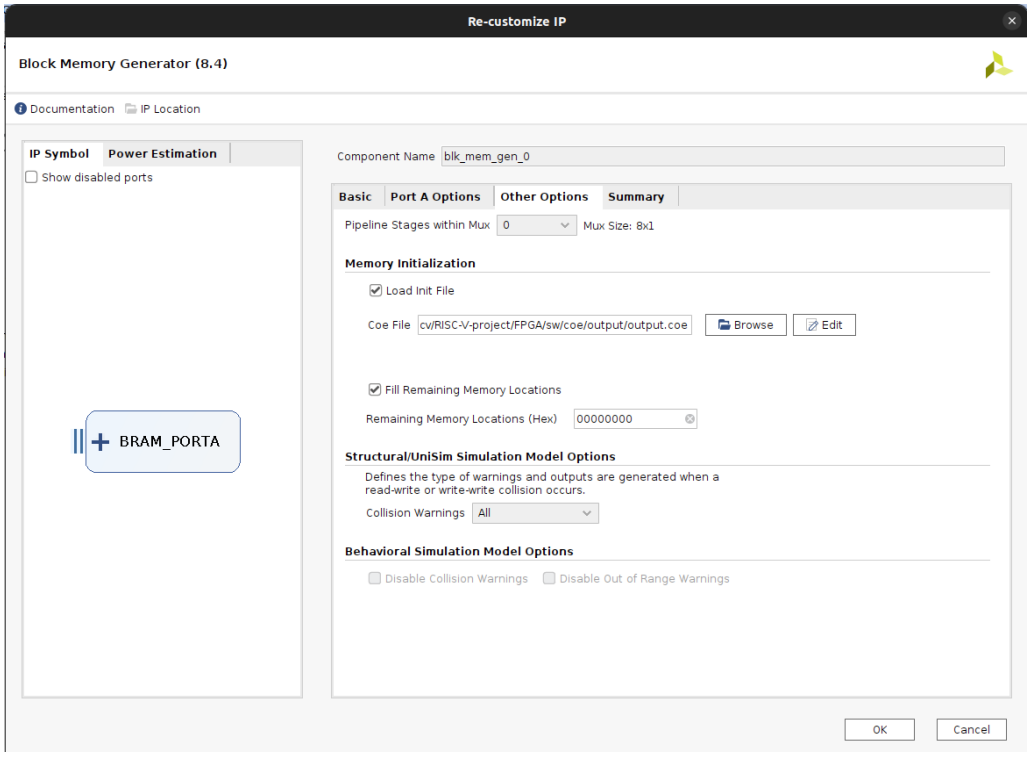


Figure 41: Initialisation de la RAM

The figure 41 presents the different options The RAM initialization file. The .coe file is generated from the executable that we wish to execute by our processor :
Thanks to a first script[22] found on Github, we will be able to transform the executable into a .hex or .txt file which will be in fact the sequence of instructions/data in the memory (from address 0x00000000). A second script [23] will allow us to transform the previous file into a .coe file requested by the RAM IP. All this is of course done automatically via a Makefile in the IMPL/sw directory present in the git of our project[24].

We also had fun writing C drivers (present in the IMPL/sw directory) to program our system and make LEDs blink, which is quite enjoyable to watch.

We could write two drivers :

- driver_wb : generic driver of our system containing functions to write and read registers, as well as a wait function and all the information of the system and the processor.
- driver_leds : driver to control the LEDs containing functions to turn on and off the LEDs.

5.2.3 Post-implementation analysis

In this section we will analyse our physical implementation of the system seen previously on FPGA.
We recall that our target for this project is a Nexys A7 from Xilinx, but it can of course be ported to other targets from other manufacturers. We have kept the default Vivado parameters for the synthesis and implementation.

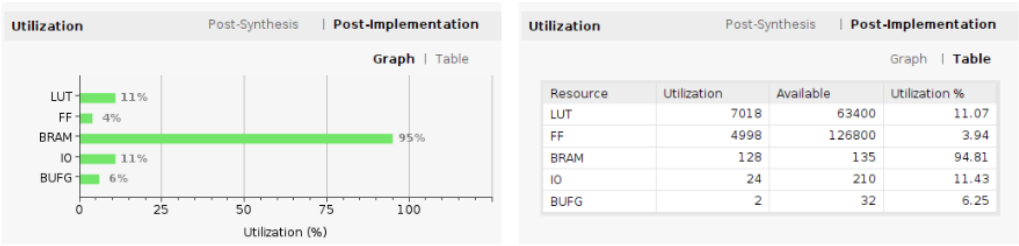


Figure 42: Use of FPGA resources

The figure shows the use of FPGA resources. We notice in particular that the LUT and flipflop are mainly for the processor. Their use by other components remains negligible (except RAM for BRAMs). Our RAM uses 95% of the BRAM present in the FPGA, i.e. about 4860 Kb.
We then notice that we are only using about 7000 LUTs (11% of the maximum capacity) of the board, but this should be compared to other similar RISC-V core implementations. But we note that it is feasible to implement multiple cores (a maximum of eight in the current configuration would be 88% of the LUTs used) without thinking about all their interfaces via the Wishbone bus.
It also seems important to note that in our current implementation, we have no caches. This is obviously not optimal in terms of performance, but in theory quite easy to implement as all the wrappers have already been implemented.
An implementation using all the resources of the FPGA could consist of 2-3 cores of our RiVer as well as an instruction and data cache (depending on their size) which will probably use LUTRAM.

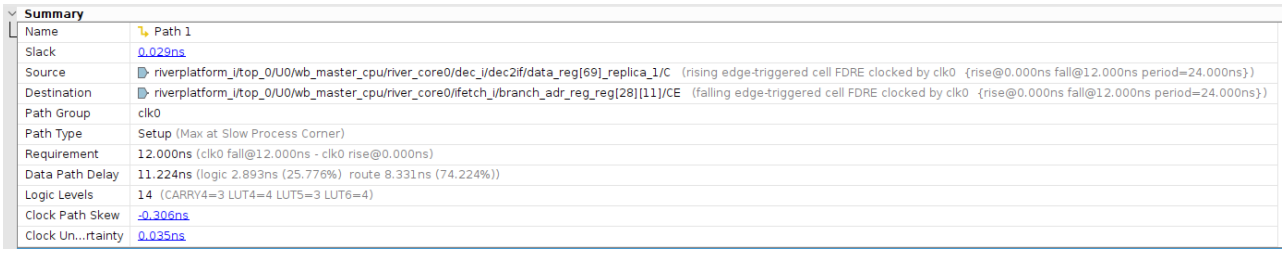


Figure 43: Critical path

In the figure 43 the critical path (longest propagation time between two D flip-flops) is in the ifetch stage and the branch prediction and fifo between ifetch and decod. The minimum period we could force is 24 ns (half period of 12 ns) which is a maximum frequency of 41.77 MHz or to simplify, we can impose a frequency of 40 MHz (on Nexys A7). This seems to us to be a fairly good frequency value given that we have not sought to optimise the critical path of our processor. This frequency remains quite acceptable, especially in the world of microcontrollers.
For that we can imagine solutions in order to maximize the frequency (minimize the critical path) to cut in two the ifetch stage or to optimize the branch prediction.

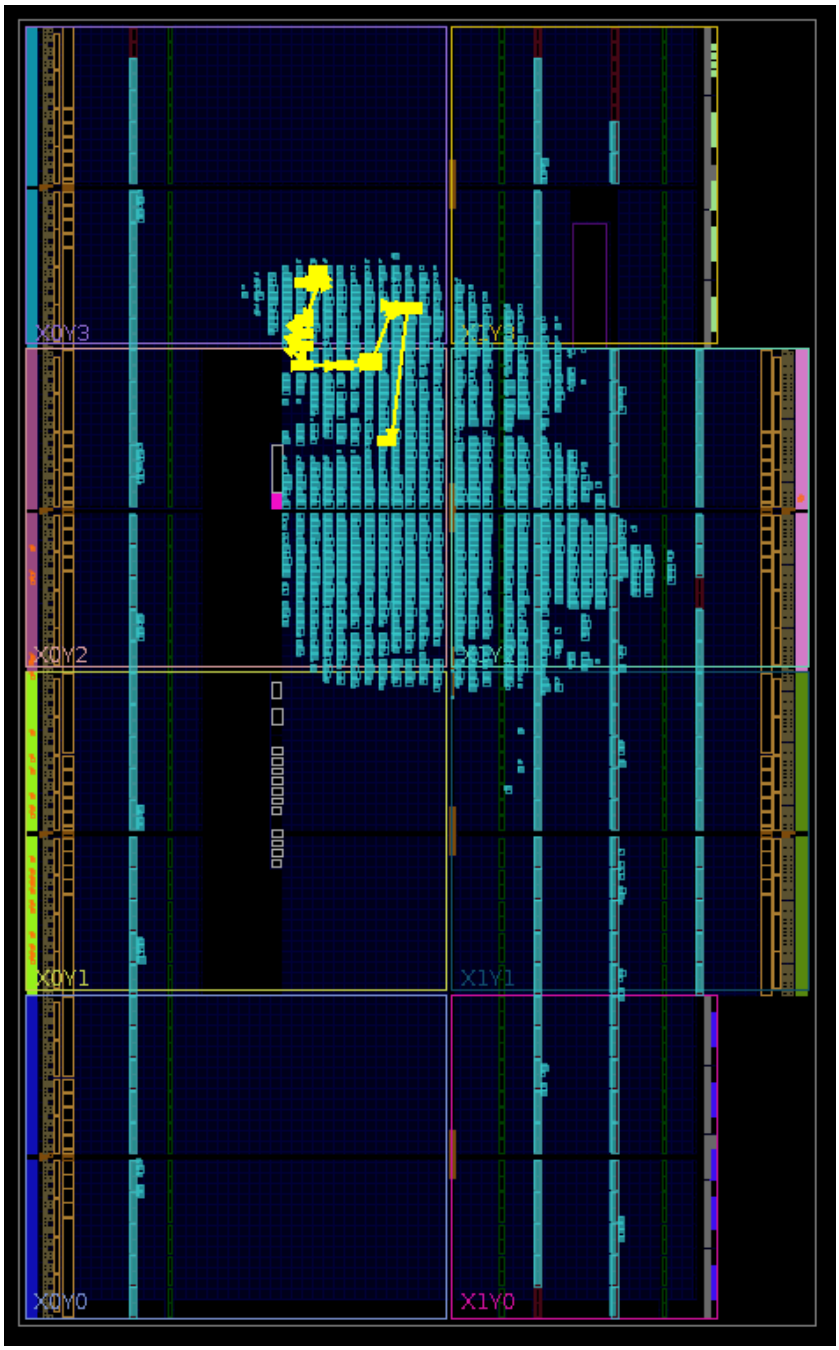


Figure 44: Post-implémentation critical path

We find on the figure 44 the critical path in post-implementation which could be highlighted (in yellow) thanks to Vivado.
This concludes this FPGA part. We were able to carry out several tests with several programs made by our care (flashing of leds, chase...) in order to confirm the good operation of our system. It is also possible to imagine to port our processor on an ASIC, in particular with the Alliance/Coriolis software suite of LIP6.

6 Miniriscv, a lighter heart for teaching

We have recovered a part of the processor (simple processor, without branch prediction, multiplication, superscalar, or privilege mode / function called "kernel" in the rest of this report), which we have reorganised to make it more understandable, and we have added a lot of comments.
The idea is that this code should be understandable by students who would put in the time, or at least readable and usable in a fairly short time.

At the time of writing, two tutorials are planning to use this processor: one to explain the systemC and the operation of the processor itself, where students have to execute instructions on the processor, and then add a simple component (which will have been removed beforehand), the Shifter.
The second tutorial goes into less detail about the use of the processor, but shows an example of simple communication between the processor and RAM using the VCI interface. The fact of using the exact same processor helps to understand the tutorial.

7 Conclusion

7.1 Problems encountered in the design

The first difficulty we encountered was the division of labour. Indeed, the project being quite large, we had to define who would be in charge of what in order to avoid stepping on each other's toes during the realisation of the project. Moreover, defining key stages as well as deadlines helped us a lot in the distribution of the work.
The second difficulty encountered was the synthesis of the RISCV specification, we had to sort out what we intended to implement and what we didn't, and we had to understand the instruction set well in order to implement it correctly.
Bypassing was also difficult to implement as it was the first time we implemented an architecture with bypassing. Indeed, there are many special cases that need to be taken care of, as they can cause many problems. On the other hand, the SS2 implementation doubles the amount of bypasses used and makes the data dependencies more complex, so the debugging of this architecture was quite long.
Finally, the implementation of a machine/user mode, what we call simply Kernel mode, took us a long time. Indeed, we have never studied this type of architecture before since it is in the M2 program. We had no idea initially of what to do and how to manage it.

Finally for the FPGA part we had some difficulties to find a simple and universal solution to program our implemented processor. In particular, we abandoned the idea of reprogramming dynamically by JTAG. The documentation on the use of JTAG being rather succinct (without going through AXI4 in the framework of Xilinx IP) to directly load a code. It is however possible to add this functionality with the current system and some modifications.

7.2 Prospects for improvement

During this project which lasted nearly 8 months, we had the opportunity to develop a lot of things and to experiment with many different implementations. The scalar core being fully operational, it could be able to support a Linux-like operating system, provided that virtual memory management is added and a supervisor mode is added for the kernel part. Finally, it would also be necessary to add an exception in the event that a user attempts to use a CSR instruction. We have not implemented this feature but if an OS is ported to the core and this exception is not generated, a big security hole could appear. If these features were provided it would be possible to run a very basic OS on the core containing just the standard library. Mr. Franck Wajsbürst has developed a very simplified Linux-like OS that Timothée Le Berre had begun to adapt for our processor, so his work could be taken up again with a view to running this code directly on our core.
On the other hand, during the implementation on FPGA Mr. Samy Attal discovered a bug in the RAS mechanism of the branch prediction that we did not have time to correct.
The dual core SoC has only been tested a little and many problems can still appear.
It could be interesting to add a latency on memory accesses and thus measure the real impact of the caches, indeed the memory being perfect in our simulator the caches have no interest if a latency is not introduced.
Finally the SS2 could be improved in order to tend more towards a synthesizable model and the prediction of branching could also be added on this implementation.
Finally, for the VHDL part, the code could be adapted in order to have it synthesized by the Alliance/Coriolis software suite of LIP6 and analyze a potential implementation on silicon.

References

[1] https://www.x86-guide.net/fr/cpu/Intel-8086-PDIP-cpu-no662.html

[2] https://www.cpushack.com/MIPSCPU.html

[3] https://en.wikipedia.org/wiki/IBM_System/370

[4] https://www.accellera.org/

[5] https://en.wikipedia.org/wiki/IBM_document_processorsIBM_801

[6] https://en.wikipedia.org/wiki/Wallace_tree

[7] Waterman, A., Lee, Y., Patterson, D., Asanovic, K., level Isa, V. I. U. (2014). The RISC-V instruction set manual. Volume I: User-Level ISA'.
https://riscv.org/technical/specifications/

[8] Waterman, A., Lee, Y., Patterson, D., Asanovic, K., level Isa, V. I. U. (2014). The RISC-V instruction set manual. Volume II: Privileged Architecture.
https://riscv.org/technical/specifications/

[9] Asanović, K., Patterson, D. A. (2014). Instruction sets should be free: The case for risc-v. EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146.

[10] Utting, M., Kearney, P. (1992). Pipeline specification of a MIPS R3000 CPU. Technical Report 92-6, Software Verification Research Centre, Department of Computer Science, University of Queensland.

[11] David A. Patterson John L. Hennessy (2021). Computer organization and design RISC-V edition, second edition.

[12] Jurij Šilc, Jurij Silc, Borut Robic, Theo Ungerer (1999). Processor architecture : From dataflow to superscalar and beyond.

[13] https://github.com/lovisXII/RISC-V-project/tree/main/Compte_rendu

[14] https://www.researchgate.net/figure/Conventional-Array-Multiplier_fig1_306034550

[15] https://github.com/riscv-software-src/riscof

[16] https://www.brainkart.com/article/Arithmetic-Operations–Division_8616/

[17] https://cdn.opencores.org/downloads/wbspec_b3.pdf

[18] http://ghdl.free.fr/

[19] http://gtkwave.sourceforge.net/

[20] https://digilent.com/reference/programmable-logic/nexys-a7/reference-manual

[21] https://docs.xilinx.com/v/u/en-US/pg058-blk-mem-gen

[22] https://github.com/irmo-de/xilinx-risc-v/blob/main/firmware/makehex.py

[23] https://github.com/kooltzh/xilinx-coe-generator

[24] https://github.com/lovisXII/RiVer_SoC/tree/main/IMPL/sw

[25] https://en.wikipedia.org/wiki/Branch_predictorStatic_branch_prediction