Politecnico di Torino

Master's Degree Course in Electronic Engineering

Master's Degree Thesis

Integration of LEN5, a RISC-V Out-of-Order Microprocessor, Inside a Low-Power, Heterogeneous System on Chip

Supervisors

prof. Maurizio MARTINA

prof. Guido MASERA

Co-Supervisors

eng. Michele CAON

eng. Mattia MIRIGALDI

Candidate

Ivan BIUNDO

ACADEMIC YEAR 2023-2024

Abstract

Processors supporting Out-of-Order (OoO) execution are widely used today, forming the basis of modern processors. The ability to reorder instruction execution, makes these processors, already widely used in consumer and High-Performance Computing, attractive in the realm of modern Low-Power Heterogeneous Systems, where a vast number of peripherals such as Coprocessors, Accelerators, GPUs, etc., need to be coordinated with arbitrary latencies. Being able to execute instructions at the time when operands are available and not in the order specified by the user, allows masking the latencies of these peripherals and at the same time increasing Instruction Level Parallelism (ILP), which benefits performance. In this context, architectures based on modular ISAs, such as RISC-V, are particularly suitable for application-specific contexts, where it is critical to support custom instructions to accelerate and increase the efficiency of critical parts of selected applications.

The work described in this thesis aims at the development of a module capable of connecting a 64-bit out-of-order RISC-V core, LEN5, to a 32-bit microcontroller platform, X-HEEP, and the development of the core's internal modules to support the debug system present on the platform itself and on similar microcontrollers. The ultimate goal is to have available a complete system of all the components of a Low Power, general purpose SoC, as are memories and peripherals, that supports not only the development of the processor, but also functions as an environment for debugging hardware and software, allowing tests to be performed on a system consistent with real applications.

The first part of this thesis will illustrate the development of a Bridge, which deals with the conversion of 64-bit requests from the core to 32-bit requests for the system bus. It will begin with an introduction to the architectures in question, LEN5 and X-HEEP, introduce the RISC paradigm, including relevant historical hints, and explain the reasons for its widespread adoption. It will then proceed to one of the two cornerstones of this thesis, the description of the development process of the bridge module, analyzing its functional requirements, motivating its architectural choices, and accurately describing its internal architecture. It will continue with the testing phase, in which the proper functioning of the module and the results obtained will be analyzed. It will finally conclude with the connection of LEN5 within X-HEEP, evaluating the impact and performance of the bridge.

The second part of the thesis will detail the process of developing the components needed to embed the debugging system within LEN5. It will begin with the development of the modules needed to support the JTAG protocol, and then focus on the second cornerstone of this thesis, which is the modifications to the architecture of LEN5 needed for compatibility with the debugging module present within the X-HEEP platform.

Acknowledgements

Ripensando agli anni del mio percorso universitario, le persone che hanno ricoperto un ruolo fondamentale e che hanno lasciato il segno nella mia vita, sono molte.

A partire da mia Madre e mio Padre, i miei punti di riferimento, i quali mi hanno sostenuto e spronato nei momenti difficili. Senza di loro tutto questo non sarebbe stato possibile.

Mio Fratello, divenuto coinquilino e compagno universitario nelle ultime fasi della mia carriera, sempre pronto ad ascoltarmi e a sollevarmi il morale.

I miei Nonni, in ogni istante sempre pronti a incoraggiarmi e a spendere una parole dolce nei miei confronti.

I miei coinquilini Salvo e Andrea, che più che coinquilini definirei fratelli acquisiti, con cui ho condivisio gioie e dolori fin dall'inizio di questa avventura.

Tutti i miei amici di su, Marta, Fede, Raf, Luca e Lore, su cui posso sempre contare e che ci sono sempre nel momento del bisogno.

Tutti i miei amici di giù, Pino, Peppe, Andrea, Angelo, Oliver, Dario, Alessio, Daniele, Matteo e Ciccio, con cui sono cresciuto e con cui ho passato i momenti più divertenti della mia vita.

A tutti voi voglio dire Grazie, perchè senza di voi, non sarei ciò che sono oggi.

In ambito accademico voglio ringraziare il professor Martina e il professor Masera, per avermi dato la possibilità di lavorare su questo progetto e accrescere conoscenze fondamentali che rimarranno parte del mio bagaglio per sempre. Inoltre ringrazio i miei Correlatori, Michele e Mattia, per avermi supportato durante tutto il lavoro.

Ivan

Contents

List of Tables IV

List of Figures V

I Development of the Bridge Module 1

1 General Introduction 2

1.1 LEN5 - Out of Order Processor . . . 2

1.2 X-HEEP - eXtendable Heterogeneous Energy-Efficient Platform . . . 7

1.3 Purpose of Integrating an OoO Processor in a Low-Power Heterogeneous SoC 9

2 LEN5 and X-HEEP Interfaces 10

2.1 OBI Protocol . . . 10

2.2 Addressing Method and Memory Alignment . . . 14

2.3 Bridge Motivation . . . 15

3 Bridge Design 17

3.1 Bridge Top . . . 17

3.2 Instruction Module . . . 21

3.2.1 Instruction CU . . . 23

3.2.2 Instruction Buffer and Tag FIFO . . . 27

3.2.3 Instruction Bus MUX and Additional Signals . . . 28

3.3 LOAD Module . . . 29

3.3.1 Grant Control Unit . . . 31

3.3.2 Address Splitter . . . 35

3.3.3 Rvalid Control Unit . . . 36

3.3.4 Data Buffer and Load FIFO . . . 43

3.3.5 Byte Selector . . . 43

3.3.6 Additional Signals . . . 46

3.4 STORE Module 47
3.4.1 Data Aligner 49

4 Experimental Results 52
4.1 Functional Verification 52
4.1.1 Instruction Module Simulation 53
4.1.2 LOAD Module Simulation 55
4.1.3 STORE Module Simulation 57
4.2 Synthesis 59
4.2.1 Bridge 60
4.2.2 Instruction Module 61
4.2.3 LOAD Module and STORE Module 62
4.2.4 Case Condition Analysis 62

5 Integration of the LEN5 processor within X-HEEP 64
5.1 LEN5 Instantiation 65
5.2 X-HEEP Modifications to Ensure Compatibility with LEN5 66
5.3 Simulation and Benchmark 67

II Development of the Debug System within the LEN5 processor 70

6 Debug System Design 71
6.1 JTAG 71
6.1.1 TAP Controller and Registers 72
6.2 Adaptation of LEN5 for X-HEEP's Debug System Support 80
6.2.1 Control Status Registers (CSRs) 81
6.2.2 Issue Stage Modification 82
6.2.3 Commit Stage Modification 85

7 Concluding remarks 90
7.1 Further Improvements 90

Bibliography 92

Acronyms 94

Glossary 95

List of Tables

2.1 OBI Port List 11
2.2 RISC-V most important Instruction Set and Extensions 16
3.1 Instruction CU - Signal Generated and Generation Methods 25
3.2 Grant CU - Signal Generated and Generation Methods 34
3.3 Rvalid CU - Signal Generated and Generation Methods - IDLE STATE . . 40
3.4 Rvalid CU - Signal Generated and Generation Methods - WAIT STATES . 41
4.1 Cycle Time and Max Clock Frequency for the Bridge and the three Principal Modules 62
4.2 Total Area, Combinational Area and Non-combinational Area for the Bridge and the three Principal Modules 62
4.3 Cycle Time, Max Clock Frequency and Area of the Bridge, as a Function of Case Conditions 63
5.1 Comparison between X-HEEP Supported CPU Cycle Time 69

List of Figures

2.1 OBI Read Transaction . . . 13
2.2 OBI Write Transaction . . . 13
3.1 Bridge TOP . . . 20
3.2 Bridge Instruction Module . . . 23
3.3 Normal Instruction Transaction . . . 26
3.4 Rready fluctuation during Instruction Transaction . . . 27
3.5 Bridge LOAD Module . . . 31
3.6 State Diagram of the Grant CU for a DOUBLEWORD Request . . . 34
3.7 State Diagram of the Rvalid CU for a DOUBLEWORD Request . . . 40
3.8 LOAD DOUBLEWORD - Normal Execution and Execution with Exception Present . . . 42
3.9 Byte Selector - 64 bits Selection . . . 44
3.10 Byte Selector - 32 bits Selection . . . 46
3.11 Bridge STORE Module . . . 49
3.12 Data Aligner - 32 bits Section . . . 51
4.1 Execution of an Instruction Request and Response . . . 54
4.2 Execution of an Instruction Request and Response with Rready Signal Deassertion . . . 54
4.3 Load BYTE and Load HALFWORD Execution . . . 56
4.4 Load WORD and Load DOUBLEWORD Execution . . . 57
4.5 Store BYTE and Store HALFWORD Execution . . . 58
4.6 Store WORD and Store DOUBLEWORD Execution . . . 58
5.1 LEN5 and Bridge Connection to XBAR and System BUS . . . 66
5.2 Output of the Hello World! Application . . . 68
5.3 Comparison in terms of execution cycles among processors supported by the X-HEEP platform. The tests performed were: printing a string, sum, multiplication, and division with relative printing of the result for each operation. 69
6.1 TAP Module . . . 74
6.2 TAP Controller State Diagram . . . 77

Part I

Development of the Bridge Module

Chapter 1

General Introduction

The first part of this thesis, aims at the realization of a module capable of connecting the LEN5 processor, to the X-HEEP microcontroller platform. The goal is to be able to take advantage of an Out-of-Order processor within the aforementioned platform, in order to optimize its performance by masking any latencies of the peripherals, and at the same time realize a system capable of being the development and testing platform for the core itself and any additional modules. It is therefore necessary to provide a brief introduction to the two architectures on which the work of this thesis will be based, the OoO LEN5 processor and the X-HEEP microcontroller platform.

1.1 LEN5 - Out of Order Processor

LEN5 is a processor based on RISC-V architecture, single in-order Issue, Out-of-Order execution, developed during master thesis work, by engineers Marco Andorno [1], Matteo Perotti [2] and Michele Caon [3].

The development of this processor, which took place in a purely academic setting, aims at the analysis and development of microprocessor architectures addressed during the master's and bachelor's degree courses, exploiting an open-ended ISA such as RISC-V [10], also enabling research in the same field.

RISC-V was born at the University of Berkley, California.

Conceived by a group of researchers with a fundamental goal, to develop a completely open source architecture, without any limitations imposed by private companies [4].

Based on this concept, RISC-V becomes the basis on which, whether professionals or students, can experiment or build, without having to submit to license fees or such to the multinational companies in the market. The ISA is maintained by the RISC-V Foundation, which has also found extensive support from leading companies in the field, ensuring its

continued development.

The new approach of RISC-V, Reduced Instruction Set Computer, is beginning to spread like wildfire over the years. The latter involves simplifying the instructions executed by the processor, reducing them to only those capable of being processed in one clock cycle; in addition, all arithmetic instructions are executed with operands on internal registers, the only instructions that can access memory, which is known to be very slow, are the LOAD and the STORE. This further speeds up computation for the benefit of the end user.

The reasoning, contrary to common logic, actually leads to accelerated instruction execution with significant energy savings, unfortunately paying with an expansion in the amount of instructions necessary to execute a task, weighing more on memory.

The emergence of this approach is due to the necessary contrast to CISC [5], Complex Instruction Set Computer, typically used in x86 architectures, which instead involves the use of more complex instructions, capable of operating either on registers or directly in memory, not necessarily executable in one clock cycle but reducing the amount of instructions necessary for a task.

Why then prefer the RISC paradigm to the CISC paradigm?

The main motivation is its incredible modularity. The presence of a reduced ISA and the support for custom extensions allows it to be used in a wide range of applications, from building application-specific processors, through the support of specific instructions for handling add-on modules, to being included as a core in microcontroller systems, as stated in [5].

Since LEN5 is an expression of RISC-V, this makes it a de facto entry point for in-depth study of RISC-V by students interested in the world of digital design or more specifically in microprocessor architectures, either by continuing its development in all its components or by using it as the core of more complex and generic systems, X-HEEP being one example, which this thesis deals with.

The internal structure of the processor is divided into two basic parts, Front-end and Back-end. The Front-end deals with the management of the Fetch phase of instructions, and the *speculation*, by means of a *branch predictor* of the *gshare* type. For more details regarding the processor front-end, refer to [1].

The structure of the processor Front-end, which implements the Fetch Stage, consists of four main components: the Branch Prediction Unit, the Program Counter Generator, the Memory Interface, and the Early Jump Unit.

- **Branch Prediction Unit**, module responsible for predicting the address of the next instruction, in the case of Branch. Composed in turn of a Branch Predictor called `Gshare` and a Branch Target Buffer (BTB). The `Gshare` is a branch predictor of

dynamic type, based on the type of predictor called two-level. This type of predictor has a shift register called Branch History Table (BHT), containing the outcome of the last executed branches. It is used as a pointer to a table called the Pattern History table (PHT) in which are the probabilities of a Branch being Taken, as a function of the sequence of Branches already executed, present in the BHT. The probabilities are expressed using 2 bits, so as to improve the prediction results. The Branch Predictor performs predictions on any address coming from the PC, which is why a Branch Targer Buffer is used, a cache containing the addresses of all Branches resolved as `taken`. This cache has two main functions, to provide the taken branch address immediately and to ensure that the prediction made by `Gshare` is taken into account only if it matches a branch address.

- **Program Counter Generator**, Program Counter Address Generator, selects the address of the next instruction to be executed, selecting it in order of priority from: Exception Handler, Mispredicted Branch, Branch Prediction, Early Jump and Sequential Execution.
- **Memory Interface**, saves within it the predictions made by the BPU waiting for the requested instruction to arrive, after which it forwards the instruction to the Backend.
- **Early Jump Unit**, a module that can precompute the jump address of instructions such as RET, JAL and CALL, providing it to the PC generator. Inside it is a Return Address Register where the return address is stored, to be supplied to the PC generator upon execution of a RET.

The second part of the processor, the Back-end, implements the execution pipeline, the foundation of which is based on Tomasulo's algorithm, an alternative to the better-known Scoreboarding. The latter allows the implementation of dynamic scheduling and Out-of-Order execution.

The fundamental difference between the two approaches lies mainly in the complexity of Scoreboarding compared to Tomasulo's algorithm. In fact, this allows dynamic scheduling, which is the reordering of instructions, avoiding stalls due to data-dependencies, by implementing complex data structures that keep within them information about all phases of an instruction's execution. In addition, Scoreboarding handles hazards (WAR and WAW) between instructions by stalling the processor pipeline, slowing execution. The problem is overcome by adding additional data structures, such as Reorder Buffer (ROB) and register renaming tables, further complicating instruction handling. In contrast, Tomasulo's algorithm drastically simplifies dynamic scheduling without making complex changes to the structure of the processor. Three data structures are used, the Reservation Station (RS), the Common Data Bus (CDB) and the Register Status. The Reservation Station, instantiated for each functional unit of the processor, presents within it various information about

the instructions to be executed on a given functional unit. Upon completion of the execution of an instruction, initiated when the functional unit and source registers are available, the result is saved in the Reservation Station itself, effectively going on to implement a temporary register from which the result can be retrieved if necessary to other instructions. This automatically implements register renaming. The second structure necessary is the Common Data Bus. As can be understood from the name, it is a common bus on which the results of completed instructions, present within an RS, are made available so that other instructions present in other RSs can access them. To have the information pair, result - RS where it is present, and to know when to update the data within the Register File, avoiding WAW, a third structure called Register Status is introduced. The latter is a table, with a number of entries equal to the number of registers in the architecture, in which is saved, for a given register, the index of the corresponding Entry in the RS where the result of the instruction using that precise register as a destination will be present. In addition there is a busy bit, to indicate whether the entry in question is used by an instruction being executed. This allows the system to know which RS contains the data an instruction needs.

For support of speculation, introduced through the branch predictor present in the frontend, and the *precise exception model*, a mechanism for maintaining the instruction fetch order is necessary. In the case of misprediction, it must be possible, in fact, to flush erroneously fetched instructions from the pipeline before they are committed, in the Commit phase, and start again. For this purpose, the structures already implemented in the backend are enriched with the presence of a ROB, in which instructions in the order of fetch are saved. Within the RS we find the index of the ROB entry of a given instruction, so that the ROB becomes the auxiliary register implementing register renaming. Consequently, in the Register Status the ROB entry is inserted instead of the RS entry.

What has been presented is available in Section 2.3.2 of [3] and Chapter 3 of [9].

We now turn to a general description from the execution pipeline architecture. Refer to Chapter 3 of [3] for detailed information.

The execution Pipeline consists of 3 basic blocks, Issue Stage, Execution Stage, and Commit Stage, each of which deals with a specific phase of instruction execution.

The issue stage has within it various modules necessary for the proper execution of the issue stage of the instruction:

- **Issue Queue**, a circular FIFO used to allocate instructions sent from the Fetch Stage, to be then decoded and assigned to the Functional Units of the processor. Having a depth greater than one, it allows backend stalls due to ROB or RS saturation to be obviated, allowing the fetch stage to continue fetching instructions.
- **Issue Logic**, Issue Stage control logic, which in turn consists of the Issue CU and

the Issue Decoder. The Issue Decoder is responsible for decoding the incoming instruction from the Issue Queue and, depending on the type of the latter, selects the RS corresponding to the necessary functional unit and reports any critical memory or execution order issues. It also checks the fetch of the operands of the instruction itself and notifies the CU of the type of instruction in issue. In the case of unrecognized instructions, special instructions such as ECALL, DRET, EBREAK, and MRET, or instructions not supported by the current processor configuration, it generates an exception that is later entered into the ROB along with the corresponding exception code. The Issue CU, on the other hand, is responsible for handling the sending of the instruction to the execution stage and the ROB. Depending on the code provided by the issue decoder, it checks whether the facilities in question, both or only the ROB, have an available entry and if so, it handles the pop of the instruction present in the Issue Queue. It also enables register status update, allowing the insertion of the ROB entry to which the instruction has been assigned and flagging which target register is in use. Finally, it flushes the fetch stage in the case of mispredictions reported by the branch unit and handles any exceptions coming from the Issue Queue. Externally to the two modules we find access to register status to control where to fetch the source operands (CDB or ROB) and routing of the instruction to ROB and Execution Unit.

The Execution Stage has within it the various functional units necessary to calculate instruction results and performing memory accesses. We will not go into a lengthy explanation of the various functional units. Within the unit are:

- **ALU**, assigned to operations on integers such as shifts, sums, subtractions, and Boolean logic operations.
- **Multiplier**, dedicated to the execution of MUL instructions found in the RV64M extension. At its core, LEN5 supports the instantiation of a serial or pipelined multiplier, depending on the initial configuration.
- **Divisor**, dedicated to the execution of DIV instructions, present in the RV64M extension. A serial divider is present within LEN5.
- **Branch Unit**, responsible for verifying the condition of a branch instruction. In case the verification is negative and there has been a misprediction by the Frontend, it notifies the issue stage of the need to flush the pipeline.
- **Load-Store Unit**, unit used to handle LOAD/STORE requests to the memory connected to the processor. Within the module we find two fundamental components, the Load Buffer and the Store Buffer, both of which can be likened to Reservation Stations in which the data for the corresponding instructions are stored, before performing memory accesses. In this version of LEN5, the Virtual Address Adder is not included; instead, a normal adder is used to calculate the final address from a base

address and an offset, making it incompatible with Virtual Memory support.

Last, the Commit Stage is used to manage the instruction commit stages. Inside we find: the ROB, the Commit Decoder, the Commit Cu and the Forward logic of operands.

- **Commit Decoder**, is in charge of recognizing the instruction contained in the Head of the ROB, reporting its type to the Commit CU. In addition, in the case of CSR-modifying instructions, it specifies which operation they perform.
- **Commit CU**, used to manage the entire commit phase of an instruction. Depending on the type of instruction to be committed, specified by the decoder, the CU checks whether the result is available in the ROB, for instructions that generate one, or whether a misprediction has occurred, in the case of branch or jump. Depending on the state evolution, it will take care of:
  1. Accept the commit of the instruction present in the ROB head and pop it.
  2. Enable the commit register so that it saves the instruction just deleted from the ROB.
  3. Control the CSR update, in case of instructions that require it.
  4. Update the register status and register file, making the destination register available again and writing the result of the instruction to the register file.
  5. Signal the Frontend when an exception arrives, for updating the PC with the address of the corresponding exception handler.
  6. Flush the pipeline and/or frontend in the case of misprediction and exception.
  7. Release the issue stage in case it is stalled due to execution of instructions that change the processor state.
- **Logic operand forward control**, is responsible for providing the issue stage with the operands necessary for the instruction being issued, if these are available in the CDB, ROB, or one of the Commit Stage buffers.

Finally, inside the processor are the CSRs, registers used to save the state and configuration of the processor. LEN5 supports the `zicsr` extension, so it can execute CSR-modifying instructions when privileges permit.

1.2 X-HEEP - eXtendable Heterogeneous Energy-Efficient Platform

X-HEEP is a microcontroller platform whose foundation lies in: configurability, to adapt its structure to different uses; expandability, to ensure that non-native modules can be

included, supporting their use; and heterogeneity, presenting within it a multitude of peripherals that can be used for different purposes. The above expands the platform's fields of use, being able to be employed from very low-power applications to complex systems comprising a variety of add-on modules and external peripherals. The architecture of X-HEEP is based on the use of a RISC-V core selectable between the cv32e20 and the cv32e40 family, providing the option of choosing the most suitable for the domain in which the platform will be used. The cv32e20 is optimized for low power applications, while the cv32e40s are better suited for high performance applications.

The interconnection between the core, memories, and peripherals is via a system bus, taking advantage of the OBI protocol. Like the other components of the platform, the bus has a configurable implementation, allowing for onetoM instantiation, where a single Master at a time can control the bus, or NtoM instantiation, where simultaneous access to multiple Slaves is possible. OnetoM instantiation is less wasteful in terms of area occupancy but is also less performant, necessarily having to wait for the completion of one transaction on the bus before another can be initiated. On the other hand, NtoM instantiation turns out to be very performant, being able to take advantage of a larger number of Master/Slave ports and consequently perform simultaneous transactions, at the cost of higher area occupancy and higher power consumption.

In the latter bus configuration, it is also possible to select the method of memory access, whose number and size of banks is user-configurable. In fact, two memory access methods are available: contiguous or interleaved. The first reduces bandwidth against the possibility of using power gating to limit power consumption. The second, on the other hand, provides higher performance, but having to keep all memory banks constantly active drastically increases consumption.

The platform natively features a variety of peripherals suitable for various application areas, such as: Timer, Platform Level Interrupt Controller, GPIO, SPI, etc., organized on different peripheral domains. In fact, X-HEEP supports two types of peripheral domains: a standard one, in which it is possible to disable or physically remove peripherals, and an always on one, in which we find peripherals that are fundamental to the operation and management of the platform itself, such as, for example: a Boot Rom, necessary for the configuration and choice of firmware execution mode phases; a Power Manager, responsible for managing the system's energy use through clock gating, power gating and similar techniques; a DMA, responsible for memory accesses performed by peripherals.

The strength of the X-HEEP platform is its extensibility, through the use of the XAIF configurable interface. Through the latter it is in fact possible to connect external modules, of particular interest are accelerators, whose interfacing requirements are met through the interface itself.

An overview of the platform build process itself can be found in Chapter 5.

What has been introduced can be found in [6], refer to it for more details on the architecture of the platform and how it works.

1.3 Purpose of Integrating an OoO Processor in a Low-Power Heterogeneous SoC

The motivations for including LEN5 within X-HEEP are several.

The support for Out-of-Order execution makes LEN5 an attractive processor as a core of Low-Power Heterogenous SoCs, in which the management and coordination of a large number of modules and peripherals, presenting arbitrary latencies, is critical. The ability to reorganize instructions, whose execution depends on the availability of operands and functional units and no longer on the order arranged by the user, makes it possible to mask delays due to peripherals and achieve a not inconsiderable speed-up of the entire platform.

At the same time, it is important to remember that LEN5 is a processor developed primarily for academic purposes, allowing researchers to study RISC-V-based architectures. It is therefore of interest for the purposes of developing and maintaining the architecture, whether one wants to optimize the already existing parts of the processor or to modify it by adding new modules, to have a complete platform available that can serve as a development environment. In fact, inclusion in a complete architecture makes it possible to expand the amount of tests that can be performed on the processor itself, being able to exploit commonly used scenarios, which differ greatly from the benchmarks normally performed as tests and which do not allow an exhaustive debugging phase on a complex processor.

For this reason it was decided to integrate LEN5 inside X-HEEP, making it officially a supported CPU.

Chapter 2

LEN5 and X-HEEP Interfaces

Before explaining the bridge architecture, it is useful to give some background on the operation of the LEN5 and X-HEEP interfaces, regarding both instruction requests and LOAD and STORE requests to memories. This will introduce the motivation for the implementation of the bridge module, which is concerned not only with handling LOAD and STORE requests, but also with instruction requests and their proper delivery.

2.1 OBI Protocol

The X-HEEP platform is based on an interface exploiting the OBI protocol, open bus interface, used for point-to-point connections via bus. Based on the request-grant mechanism, which is very similar to ARM's AMBA AXI protocol, each transaction via the OBI protocol consists of two parts, a request and a response, sent via the appropriate channels called Address Channel and Response Channel, as outlined in the protocol specification [8].

Specific, standardized signals, listed in table 2.1, are required for transaction execution:

Name	Source	Destination	Description
Global Signals			
clk	Clock Generator	Any	Reference Clock for transactions
reset_n	Reset Control Unit	Any	Bus and system reset signal, active low.
Address Channel Signals			
req	Master	Slave	Request signal. Indicates the validity of the signals present in the Address Channel.
gnt	Slave	Master	Grant signal. Indicates the availability to accept a transaction. Transaction accepted with req=1 and gnt=1.
addr[]	Master	Slave	Address Bus
we	Master	Slave	Write Enable. Write if high, Read if low.
be[]	Master	Slave	Byte Enable. Select bytes to read or write.
wdata[]	Master	Slave	Data to be written. Valid only for writings, not defined for readings.
Response Channel Signals			
rvalid	Slave	Master	Valid Signal. Indicates the validity of the signals present in the Response Channel.
rready	Master	Slave	Ready Signal. Indicates the availability to accept the Response. Response accepted if rvalid=1 and rready=1
rdata	Slave	Master	Read data. Valid only for reads, not defined for writes.

Table 2.1: OBI Port List

The operation of the protocol is given below for both read or write requests, as specified in section 3.3 of the protocol specification [8].

From now on, the request phase will be referred to as Address Phase and the response phase as Response Phase.

In the Address Phase, the Master indicates the validity of the signals in the Address Channel, specified in table 2.1, by the assertion of its request signal (`req`). The Slave indicates its readiness to accept signals on the Address Channel by asserting its grant signal (`gnt`). Address Phase begins in the clock cycle in which the request signal (`req`) is asserted and ends when both the request signal (`req`) and the grant signal (`gnt`) are asserted.

Once the Address Phase is completed, the Slave reports the validity of the signals on the Response Channel by asserting its `rvalid` signal. Simultaneously, the Master communicates its readiness to accept the above signals by asserting its `rready` signal. The Response Phase begins in the clock cycle when the `rvalid` signal is at high value and ends when both the `rvalid` and `rready` signals are at high value.

In the figures 2.1 and 2.2, the two types of possible transactions via OBI protocol are depicted, READ to perform a read from a register or memory location and WRITE to write to a register or memory location. Note how in the READ transaction, the Master sets the signal `req` to a high value while on the bus `addr` is present the address needed to access a register or memory location. In this case the Slave is always ready to accept an Address Phase, so it keeps the `gnt` signal at high value constantly.

Consider this as a particular case; in fact, it is not necessarily true that the `gnt` signal is kept constantly asserted; rather, there will be cases, such as in bridge and X-HEEP, in which this signal is kept at a low value and asserted only when a transaction request is actually received. The implementation of this behavior is due to energy saving and also limits possible errors in transactions between Master and Slave.

Special attention should be paid to the signals `we` and bus `be`. These make possible to distinguish the type of transaction request, READ or WRITE, according to the state of the signal `we`, also the bus `be` allows to select up to the single byte to be read or written, depending on how its bits are set. The `wdata` bus is ignored during READ transactions.

Finally, note the `rvalid` signal, asserted by the Slave only when the transaction is complete and the read data is present on the `rdata` bus. At the same time, the Master is always ready to receive data and keeps the `rready` signal constantly at a high value.

Again, keeping the signal `rready` constant is a special case. It will be seen, in the case of LEN5 and the bridge, that this is not always true due to the possible unavailability of LEN5 to receive incoming instructions.

In the case of a WRITE transaction, the behavior is specular, except for the signal `we`, which will be at high value, signaling a WRITE transaction, and for the bus `rdata`, which

will be the ignored one this time.

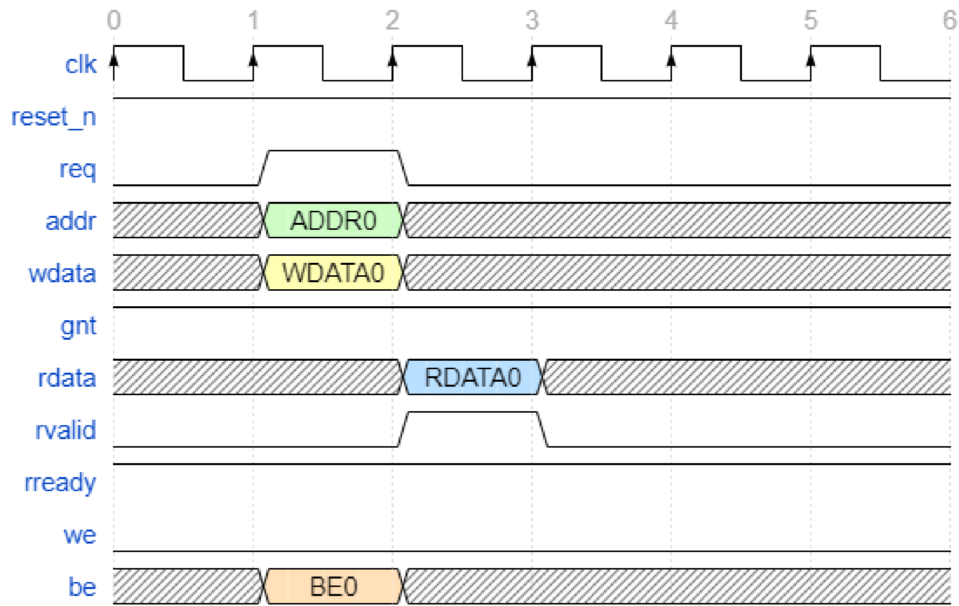


Figure 2.1: OBI Read Transaction

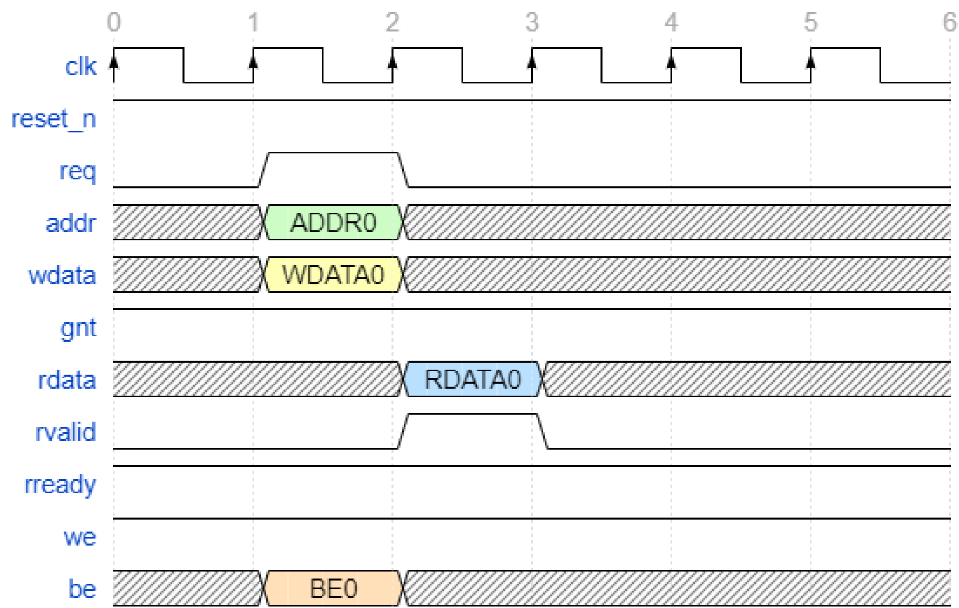


Figure 2.2: OBI Write Transaction

The LEN5 interface was then modified to support the OBI protocol, both in the instruction request and in the read and write requests related to LOAD and STORE instructions.

In order to speed up the load-store unit, it was necessary to separate the ports related to LOAD requests from STORE requests, resulting in three ports on the LEN5 interface, also considering the instruction port.

2.2 Addressing Method and Memory Alignment

The architecture of LEN5, being a newly developed processor, is based on 64-bit addressing. This implies that every request made, whether for instructions or for LOAD/STORE, is executed through a 64-bit address.

Also, it is important to keep in mind that for LEN5, a request for *Word* corresponds to a request for a 32-bit data from the memories connected to it, consequently a *Half Word* corresponds to 16 bits. Having 64-bit registers, LEN5 can also request and process 64-bit data through *Double Word* requests.

In contrast, X-HEEP is a 32-bit architecture, 32-bit addressing and 32-bit registers, with a memory alignment that is also 32 bits. Memory Alignment means that the address where the data is located within memory, is a multiple of its size in bytes, as defined in section A.3 of [9]. The minimum unit that a processor can interpret is a single byte. Data in memory must therefore be aligned to at least one byte, and a processor can access it by generating addresses that vary by an amount equal to 1. In the case of the architecture in question, the 32-bit alignment means that the data is arranged occupying 4 consecutive bytes in memory. This results in the distance between WORDs in memory, in terms of address, being equal to 4, as 32 bits equals 4 bytes. However, the processor can perform accesses with any address value, taking specific rules into account:

- Each address value can be used to perform BYTE type accesses (8 bits), addressing each of the 4 bytes that constitute a Word.
- Addresses with a multiple value of 2 can also be used to perform HALFWORD type accesses (16 bits), addressing each of the two halfwords that constitute a Word.
- Addresses with a multiple value of 4 can also be used for WORD type accesses (32 bits).

The alignment[1] mechanism is therefore fundamental; in fact, memories are commonly designed to work with groups of bytes. By ensuring, as far as possible, an aligned arrangement of data within a single group, the processor is prevented from accessing memory more than once for a single data.

Accesses to misaligned data, called *Misaligned Access*, as opposed to accesses to aligned data, called *Aligned Access*, are very time-consuming in terms of access time, as the core

[1] For more details on the arrangement of data in memory, refer to Appendix A.3 of [9]

has to make multiple accesses to memory and then reconstruct the data through shift operations or similar [9]. For this reason, many processors do not support misaligned access, and report its occurrence by generating *exception*.

2.3 Bridge Motivation

Upon analyzing the interfaces, two problems can be observed: the difference in addressing method and the maximum data size supported. LEN5 is a processor supporting 64-bit addressing, having been developed for modern systems, while the X-HEEP architecture supports 32-bit addressing. The difference in addressing is not in itself a problem; since the linker takes care of providing the correct addresses during firmware compilation, by which to perform memory access, these will necessarily fall within X-HEEP's address space and will therefore be containable in 32 bits. The address generated by LEN5 will present the useful address on the 32 LSBs of the 64 of which it is composed, thus being able to connect only the latter to the address bus of X-HEEP. Taking over support for different Instruction Sets as well, the problem becomes more complicated. In fact, the architecture of X-HEEP is based on the use of RISC-V cores supporting the Instruction Set called RV32, while LEN5 uses the RV64 Instruction Set.

Normally present as a component of the specification Istruction Set Architecture (ISA), the Instruction Set defines the instructions supported by the processor, the size of the architecture's registers, and its addressing method, as visible in [12]. The RISC-V Foundation defines two Instruction Sets as minimal, RV32I and RV64I, which as visible from the name, support an address space of 32 bits or 64 bits and a fixed instruction length of 32 bits. Using only type "I" instruction sets is highly limiting for a processor, consequently the two versions have been widely extended to support a variety of operation types, defining what we call *standard* and *non-standard* instruction sets. While standard instruction sets contain common instructions, nonstandard sets may contain instructions that can be used for very specific applications [12]. Table 2.2 shows the various types of construction sets and their most common extensions:

Name	Description
RV32I	Basic instruction set with only operations between integers, 32 bits
RV32E	Basic instruction set for embedded systems with only operations between integers, 32 bits
RV64I	Basic instruction set with only operations between integers, 64 bits
RV64E	Basic instruction set for embedded systems with only operations between integers, 64 bits
RV128I	Basic instruction set with only operations between integers, 128 bits, still under development
Extensions	
M	Extension with support for Multiplications and Divisions between integers
A	Extension with support for Atomic operations
F	Extension with Floating Point Single Precision Support
D	Extension with Floating Point Double Precision Support
Zicsr	Extension with support for instructions to access CSRs

Table 2.2: RISC-V most important Instruction Set and Extensions

Back to the architectures in question, the difference between the supported instruction sets causes the size of the internal registers of LEN5 and those of X-HEEP to differ, along with the size of the buses. In addition to the physical incompatibility of the interfaces, there arises the problem of data requests that LEN5 is capable of executing but that the X-HEEP platform cannot satisfy. In fact, having 32-bit buses and registers and not having implemented a multi-port bus for the inherent data interface, X-HEEP is only able to provide one 32-bit data at a time, limiting the LOAD and STORE instructions to request at most one WORD-sized data. LEN5, in contrast, can perform DOUBLEWORD requests, thus being able to work even with 64-bit data. This problem makes LEN5's instruction set unsupportable by X-HEEP and makes it necessary to introduce an intermediate module between LEN5 and X-HEEP's bus that can handle the processor's LOAD and STORE requests, split the DOUBLEWORD requests into two WORD requests, ensure the correct address size, and align the data consistently with the type of request made.

Chapter 3

Bridge Design

This chapter will proceed to analyze the design of the Bridge module. The layout of the module will be explained in a general way, after which each component will be described in detail. In case we refer to a particular component, it will be defined using *italic*, so that it stands out from the view.

3.1 Bridge Top

The Bridge consists of 3 main modules: *Instruction Module, LOAD Module and STORE Module*, visible in the figure 3.1. For ease of reference, the signals have been divided into colors. In red those related to LEN5 and in blue those related to the X-HEEP platform.

As the name suggests, they are responsible for handling instruction requests, LOAD data requests, and STORE data requests, respectively. The *Instruction Module* has 4 ports: one input and one output port connected to the processor and two others, input and output respectively, connected to the bus. In contrast, the *LOAD Module* and the *STORE Module* have 6 ports each, divided into: one input-output port pair connected to the processor and the remaining two input-output pairs, necessary for splitting 64-bit requests, connected to the bus.

In fact, LEN5 has a different load-store unit interface from that of common processors, having the port related to LOAD requests separate from that for STORE requests. Due to the split, the processor is able to execute the two types of requests simultaneously, speeding up the execution of instructions. The interface upgrade turns out to be an incipit for the future development of LEN5 as a multiple issue processor.

The bridge accordingly supports the processor interface, ensuring that LOAD and STORE requests can be executed simultaneously. Note that, both the LOAD Module and the STORE Module, in turn, have two ports to the X-HEEP bus, for 64-bit request

splitting. This makes the total number of ports related to data requests, to be connected to the bus, equal to four. For the reasons already introduced in Section 2.3, the single 32-bit port implementation of the X-HEEP bus makes it impossible to execute simultaneous requests, limiting the Bridge to executing only one WORD-type request at a time. This prevents exploiting the speed-up achieved by port splitting, not allowing the LOAD Module and the STORE Module to send requests simultaneously and at the same time makes it impossible to simultaneously send split requests generated by a DOUBLEWORD type request received by either module. The feature remains available, however, should the bridge be connected to a more advanced bus capable of supporting more than one simultaneous request. The connection of the bridge with the bus will then be made through the use of a 4:1 crossbar, capable of sequencing the requests sent by the two modules, making it possible to send them to the single bus port. More detailed information on the crossbar can be found in Section 5.1.

The signals present on the bridge interface are those described in table 2.1, being necessary the compatibility with the OBI protocol. We also find additional signals that we will divide into global and specific.

The global signals are as follows:

- **CLK** → Clock signal, shared with the processor.
- **Rst_n** → Reset signal, needed by specification to completely reset the bridge during boot and for any errors.
- **Flush** → Flush signal, needed to flush instructions contained in the bridge and not yet completed, in case there have been Mispredictions in the execution of a Branch or other types of problems that force the flushing of instructions in the processor.

On the other hand, as far as specific signals are concerned, we can note the presence in each port of a signal called TAG. The tagging mechanism used by LEN5 is very similar to that used within caches, in which each piece of data is associated with a label necessary for its recognition. Within LEN5, the TAG bus size is generated from the configuration parameter that defines the depth of the LOAD Buffer or the STORE Buffer, depending on which is deeper. In the case under analysis, the size of this value is 4 bits. LEN5 uses this value to store which instruction and LOAD/STORE requests have been made and, once it receives a response from the bridge, these are discarded.

The bridge receives the tags of the requests made by LEN5, whether Instructions or LOAD/STORE, and handles their proper resend to LEN5 when the requests have been fulfilled. The mechanism will be analyzed in the sections devoted to the Instructions and LOAD/STORE modules.

Last, there are signals called exceptions in all modules. As defined by RISC-V specifications, LEN5 supports the exception mechanism for instructions and data. There are various reasons why an exception may be generated, such as problems with the fetch of the instruction, problems arising from the decoding of the instruction, unrecognized instructions or read/write errors of data in memory. In the case under analysis, the exceptions that can be delivered to the processor are from an instruction read error in memory or a read/write errors of data in memory, which clearly affects whether the instruction can be executed. The X-HEEP bus does not yet support the exception mechanism, but the bridge architecture is fully compatible with them. In fact, it is able to receive them, store them and process them according to the requests received by LEN5. We will discuss the exception handling mechanism in more detail in the sections devoted to the LOAD and STORE modules.

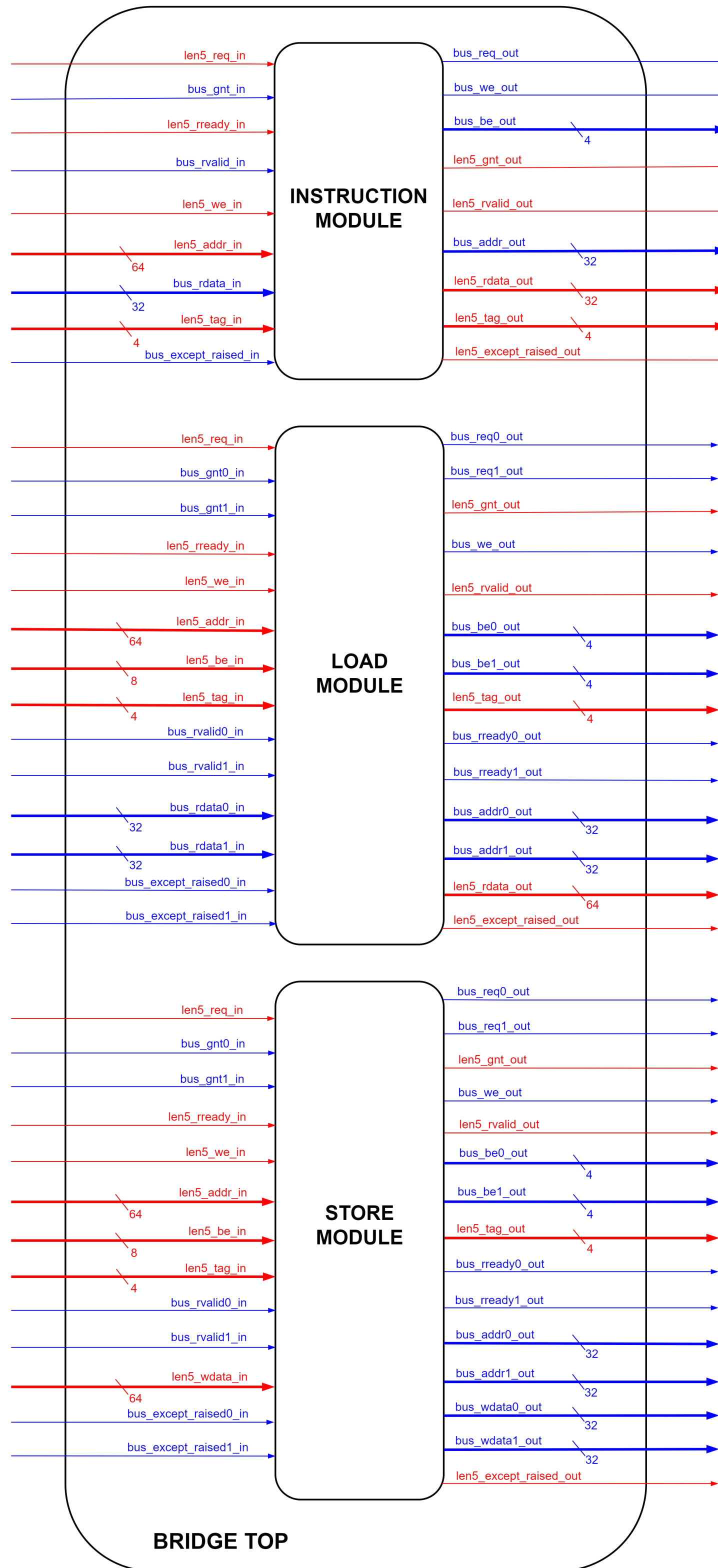


Figure 3.1: Bridge TOP

3.2 Instruction Module

Let us now introduce one of the three basic modules of the bridge, the *Instruction Module*, visible in its structure in figure 3.2. In this schematic we have shown the signals in different colors for ease of reading. We find in red the signals related to LEN5, in blue the signals related to the bus, in violet the internal signals, and in green the signals from the memories.

Responsible for handling instruction requests in both the Address Phase and Response Phase of the OBI protocol, it is itself composed of three main components: a Control Unit (CU), a Register (Buffer), and a FIFO queue, plus various multiplexers to handle the instruction bus.

The question may arise as to why handle instruction requests, since these do not present the previously described data requests problem. Indeed, IS RV64 provides a fixed instruction size of 32 bits, so no handling of double requests or data alignment would apparently be needed.

A second problem arises, however, related to the way LEN5 handles instructions internally and the operation of its instruction interface. Before executing an instruction request, LEN5 saves the contents of the PC within a register called the Request Register. The mechanism in question is necessary because instructions of the Branch and Jump type compute destination addresses, with which to modify the PC in a nonsequential manner, from the address corresponding to their location in memory, which coincides with the PC at the time they are in the fetch phase.

During the instruction request, the PC, along with the data generated by the Branch Prediction Unit, are pushed into a FIFO queue called the Prediction Buffer, holding it constant until the requested instruction arrives.

Once the requested instruction is received, it is stored in a second register called Answer Register along with the contents of the Prediction Buffer, so that the PC and the instruction can be propagated into the pipeline, together. As explained in section 2.1, the LEN5 interface takes advantage of the OBI protocol, which provides in the Response Phase a signal called `rready`, refer to the 2.1 table for operation. Unlike what was seen in the example and appropriately specified again in section 2.1, the `rready` signal is not kept fixed at the high value, signaling the constant acceptance of the input instruction; rather, its value depends on the state of the processor. Indeed, there may be cases when LEN5 is not available to receive the instruction and signals this by deasserting the `rready`. The X-HEEP bus, by contrast, does not support the `rready` signal, assuming that the processor is always ready to receive the incoming instruction.

The non-symmetry of the two interfaces causes the occurrence of instruction loss if a request is made but LEN5 is not available to receive the instruction at the time X-HEEP delivers it. This problem is particularly dangerous because in LEN5 there is no mechanism

for retrieving the lost instruction, simply because the processor cannot detect that it has been lost.

Going into detail, LEN5 switches the `rready` signal to 0 when the backend of the processor is currently unable to process an instruction in the Answer Register of the frontend. The instruction is then held within the Answer Register, along with the PC in the Prediction Buffer, waiting for the `rready` from the backend. At the same time, however, the Fetch Stage continues to execute requests for instructions, which, since they cannot be saved in the Answer Register, are lost, and places the corresponding PCs in the Request Register, subsequently passing them to the Prediction Buffer. The moment the backend asserts its `rready`, allowing instructions to be accepted again, the Answer Register samples the instruction present at its entry and the PC present in the Prediction Buffer. Since, however, the previous PCs have never been popped, the instruction - PC pair is incorrect, this causes the LEN5 pipe to desynchronize, not allowing it to function properly.

It was therefore necessary to implement the *Instruction Module* to handle this eventuality. In the event that LEN5 becomes unavailable upon receiving an instruction, the module saves the instruction within a buffer, keeping it available to LEN5 until it is again able to accept it. It also stalls requests to the bus and grants to LEN5, so as to prevent further requests until the rready signal returns to a high value. Its components are described below.

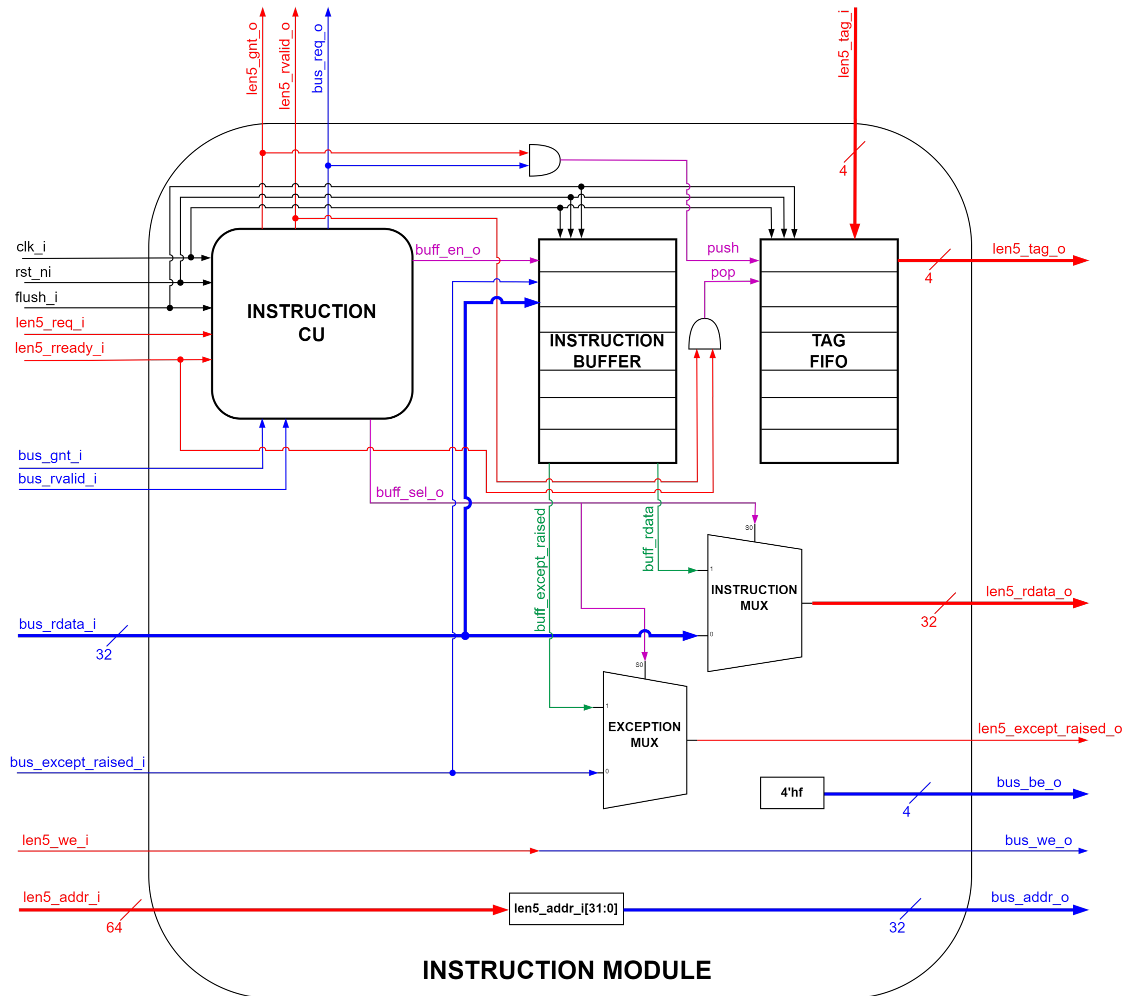


Figure 3.2: Bridge Instruction Module

3.2.1 Instruction CU

The first component that will be analyzed is the Control Unit, the component responsible for generating the control signals. In the case under analysis, the CU has an internal FSM, for the sequential generation of part of the control signals, and an independent combinational logic part for the generation of the remaining part of the control signals. A brief introduction is necessary before discussing the actual implementation. An FSM is theoretically composed of a combinational logic that deals with the generation of the future state, a combinational logic that deals with the generation of the control signals, and a register needed to save the value of the state.

FSMs can be implemented using two different methods, making what are called Moore's or Mealy's FSMs. The fundamental difference between the two types of FSMs is whether

or not the combinational logic of the control signals, is dependent on the inputs to the FSM itself. Notoriously, it is preferable to always use Moore's paradigm, as this is safer; in fact, since there is no direct connection between the logic of the control signals and the inputs, it is impossible to create combinatorial loops or critical paths that pass through the FSM. That said, it is important to point out that Moore's FSMs are slower in generating signals than Mealy's, necessarily having to perform the state transition first.

The Bridge structure should in no way slow down the processor more than necessary, which is why it was preferred to implement a Mealy's FSM, with combinatorial generation of control signals. Because of this, the control signals are already available before the FSM performs the state transition, saving a clock cycle.

The Control Unit works in 2 possible states: IDLE and BUFFER. For simplicity we will divide the description into two parts, one related to the next state logic and one related to the generation of control signals.

During normal execution, the FSM is in the IDLE state, in which incoming instructions are directly sent to LEN5 from the incoming instruction bus. In the case that LEN5 were to deassert its `len5_rready` signal, indicating that it is not ready to receive instructions, and at the same time a valid instruction arrives, signaled by the incoming `bus_rvalid` signal from the bus, the FSM would switch its state to that of BUFFER, where it would remain until LEN5 again signals its readiness to receive instructions, asserting the signal `len5_rready` and returning the FSM to the IDLE state. In addition to the `reset` signal, the `flush` signal is also capable of returning the FSM to the IDLE state if there have been flushes of the LEN5 pipe due to misprediction.

The generation of the `bus_req` and `len5_gnt` signals takes place outside the logic of control signals, so they do not depend on the state in which the FSM is, plus they are generated in a purely combinatorial manner to speed up requests to the bus, as explained earlier. The `bus_req` signal to the bus is generated by the logical AND between the `len5_req` signal input from LEN5 and the `len5_rready` signal, also input. Similarly, the `len5_gnt` signal to LEN5 is also generated by the logical AND of the corresponding input from the bus and the `len5_rready` signal. This solution is necessary to block any requests that would be made even when LEN5 is not ready to receive them.

Within the control signal logic, the signals `len5_rvalid`, `buff_en`, and `buff_sel` are generated. The signal `len5_rvalid` is used to signal the availability of a valid instruction on the bus, while the remaining two are needed: to enable the saving of the instruction in the buffer, in case LEN5 sets its `len5_rready` to a low value, and to select as module output, on input to the instruction bus of LEN5, the value present in the instruction buffer. In the IDLE state, instructions are sent directly to LEN5, so the signal `len5_rvalid` for LEN5 is directly connected to the signal `bus_rvalid` input from X-HEEP, and the instruction bus of LEN5 is directly connected to that of X-HEEP, `buff_sel`=0. However,

the buffer must be able to store within it the value of the instruction in the bus at the exact moment when LEN5 is no longer available to receive instructions. For this reason, the signal of `buff_en`=∼`len5_rready`, so that it can be asserted immediately when LEN5 deasserts the `len5_rready`, allowing the buffer to be enabled without waiting for the FSM state transition. In the BUFFER state, the instruction has already been saved within the instruction buffer and must be retained until the FSM transitions back to the IDLE state. The instruction buffer is then disabled, `buff_en`=0, simultaneously signaling to LEN5 that a valid instruction is present in the buffer by setting `len5_rvalid`=1 and no longer having it controlled by the incoming `bus_rvalid` from the bus, which returns to 0 after only one clock cycle. The instruction bus of LEN5 is connected to the output of the instruction buffer by setting `buff_sel`=1, so LEN5 will be able to sample the valid instruction as soon as it is available to do so. Table 3.1 shows the signals generated combinatorially and by the FSM along with their method of generation.

Signal	**Value**
bus_req	len5_req & len5_rready
len5_gnt	bus_gnt & len5_rready
State IDLE	
len5_rvalid	bus_rvalid
buff_sel	0
buff_en	∼len5_rready
State BUFFER	
len5_rvalid	1
buff_sel	1
buff_en	0

Table 3.1: Instruction CU - Signal Generated and Generation Methods

In the figures 3.3 and 3.4 can be seen correspondingly: a normal request of an instruction and an instruction request that has a fluctuation of the `rready` signal during the reception of the INSTR0 instruction. As can be seen. the signal `buff_en` is asserted to save the instruction in the buffer, after which there is switching of the state of the FSM and the signal `buff_sel` is set to 1, enabling the buffer output to LEN5. When LEN5 is available again, at cycle 6, the FSM returns to the IDLE state, deasserting the `buff_sel`. At the same time a second request for the INSTR1 instruction has been received, note how the request is blocked (`len5_req` maintained asserted) until the `len5_rready` signal is again asserted and the request is propagated to the bus (`bus_req` asserted).

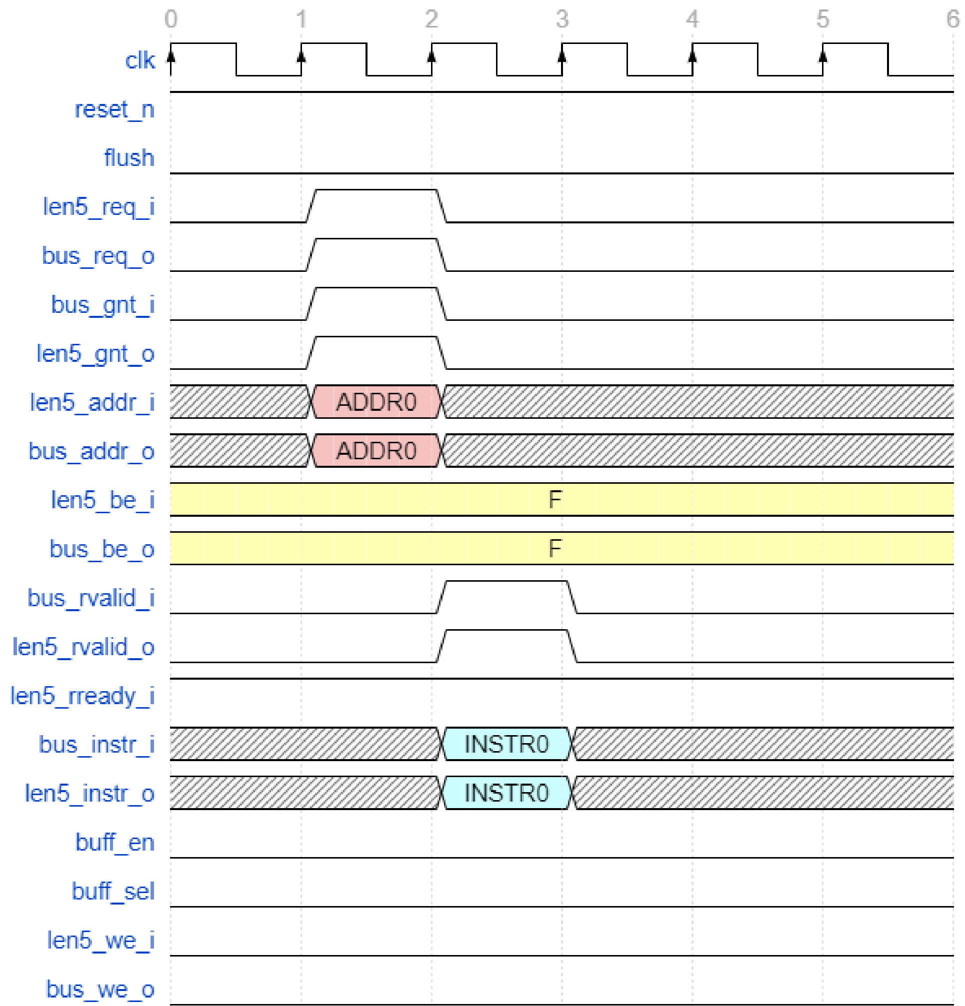


Figure 3.3: Normal Instruction Transaction

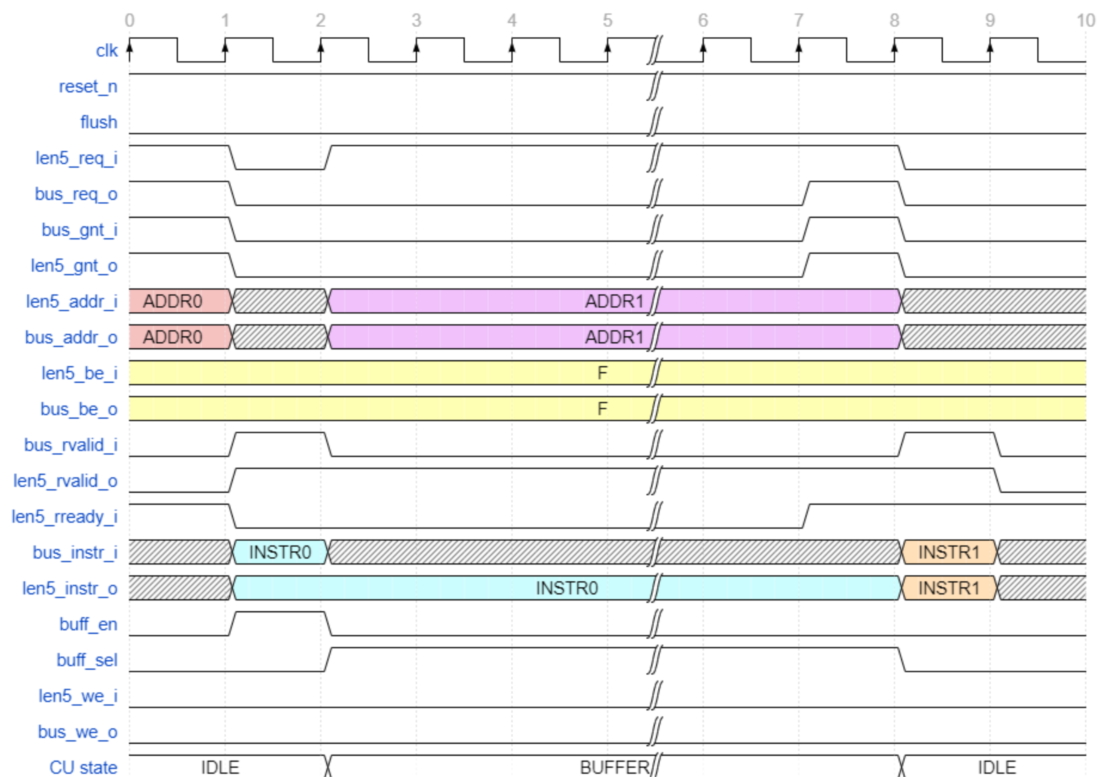


Figure 3.4: Rready fluctuation during Instruction Transaction

3.2.2 Instruction Buffer and Tag FIFO

The structure of the *Instruction Buffer* is that of a 33-bit register, divided into 32 bits for the instruction and 1 bit for the exception, if any, enabled by a `buff_en` signal. Also in the register is the possibility of flushing via the `flush` signal, in case LEN5 had flushed its pipe for misprediction. In the IDLE state, if LEN5 should be unavailable, `len5_rready`=0, the instruction would be saved within the register, along with the exception signal. The bit for the *exception* is necessary since, in the event that an *exception* was received for the instruction saved in the buffer, it must be delivered to LEN5 along with the instruction that generated it, in order to be propagated in the pipeline. X-HEEP's bus does not support any kind of *exception* for instructions, so the *Exception Code* on LEN5's interface was forced to E_I_ACCESS_FAULT. However, the bridge is compatible with the *exception* mechanism for future development.

The FIFO tag allows the instruction TAGs generated by LEN5 to be saved within it and to be supplied again when the corresponding instruction is supplied on the output, so that the processor can terminate the transaction by recognizing the TAG. The structure used is identical to those inside the processor. It consists of a circular buffer, with the

addition of a bit for each data item useful for validating the corresponding data item, and two counters in modulo N needed for counting the data within the queue and for generating the two signals `head_cnt` and `tail_cnt`. On the clock signal front, if the `push` signal is present, the queue saves the data at the position indicated by `tail_cnt`. On the other hand, if the signal `pop` is present, the queue deletes the data present at the position `head_cnt`. In the case of an instruction transaction, the queue must save the TAG at the time the transaction is accepted, end of Address Phase, and provide it to LEN5 at the time the instruction is available, end of Response Phase. To do this, the signal `push` is generated by the logical AND between the signal `bus_req` and the signal `len5_gnt_o`, when in fact both signals are available, the transaction has definitely been accepted and the Address Phase, concluded. The `pop` signal, on the other hand, is connected to the logical AND between the `len5_rvalid_o` signal and the `len5_rready_i` signal, which respectively signal the readiness of the instruction and the readiness of LEN5 to accept it, thus indicating the conclusion of the Response Phase.

Since it is a circular buffer, to avoid errors due to `push` operations when the queue is full or `pop` operations with an empty queue, there is also a system to control the filling or emptying of the queue by means of the signals `empty` and `full`, generated combinatorially from the counters. If the two counters were to have the same value but the `data_valid` bit corresponding to the position of `head_cnt` was 0, the queue would be empty and no `pop` operations would be allowed. Similarly with the two counters of identical value and the bit `data_valid` corresponding to the position of `head_cnt` was 1, the queue would be full and no `push` operations would be allowed. The structure also supports the possibility of `flush`, like any other presented, by counter reset and forced emptying of the FIFO queue.

LEN5 does not take advantage of TAG generation for instructions, and the FIFO is unused at present. However, it is introduced for an eventual evolution of LEN5 in multiple issues, where it will be necessary to include TAG support for instructions as well.

3.2.3 Instruction Bus MUX and Additional Signals

Finally, the *Instruction Module* has two MUX, which are necessary for routing to the LEN5 bus of instructions and exceptions, in the two states of the FSM. Both controlled by the `buff_sel` signal, in the IDLE state of the FSM, they directly connect the instruction bus and exception signal, incoming from X-HEEP with those of LEN5. In the BUFFER state, on the other hand, they connect the contents of the *Instruction Buffer* with LEN5.

Finally we find: the signal `len5_we`, responsible for the selection of READ or WRITE requests, connected directly to the output to X-HEEP, not used in the case of instruction requests, these being read-only, it is retained for protocol compatibility; the bus `len5_be`, necessary for the selection of the type of request, WORD, HALF-WORD or BYTE, fixed

at the value 4'hf in the case of instructions, these being necessarily WORDs; the bus `len5_addr`, containing the address of instructions, of which only the 32 LSB are connected directly to the bus of X-HEEP.

3.3 LOAD Module

The second main module of the Bridge is the *LOAD Module*, visible in the figure 3.5, used to handle LOAD-type requests to the X-HEEP bus. In the schematic, the signals have been divided by function, via color. In red the signals coming from or destined to LEN5, in blue the signals coming from or destined to the X-HEEP bus, in green the internally used data signals, and in violet the control signals.

As already introduced in section 2.3, the bus size of X-HEEP is only 32 bits, thus preventing LEN5 from executing DOUBLEWORD requests directly. It is therefore necessary for the LOAD Module to distinguish between the two possible types of data requests, 32 or 64 bits, coming from LEN5 and, taking advantage of the two available ports, to send a single 32-bit data request in the case of LOAD WORD, HALFWORD and BYTE or two simultaneous 32-bit data requests in the case of LOAD DOUBLEWORD.

As introduced earlier, the microcontroller bus does not allow more than one request to be executed at a time, so implementing a single port, executing two requests sequentially, would have sufficed. It was decided to integrate two ports and the possibility of simultaneous requests anyway, in the event that the bridge was connected to a more advanced multiport bus. The requests will later be sequenced by the crossbar, introduced in section 3.1.

Since the type of bus to which the architecture is connected cannot be known, it cannot be determined whether the 64-bit data will arrive at the same instant or at different times, so the module is able to save the first 32-bit input data, within a buffer, waiting for the arrival of the second part of the data, and then supplying the complete data to the LEN5 bus. A double buffer implementation could have been used at the input, so that both input data would be saved in each case, simplifying the CU. A single-buffer implementation was preferred, to speed up the delivery of the data to LEN5.

The operations performed by the module are as follows:

1. If a request is received from LEN5 regarding a LOAD instruction, the module distinguishes the type of request according to the bus byte enable. Requests can be for 64-bit data, thus DOUBLEWORD, otherwise for 32-bit data, thus WORD, HALFWORD or BYTE.
2. Generates `request` signals on one or both ports, depending on the type of request received, to be sent along with the address to the X-HEEP bus.

3. Waits for `grant` signals on one or both ports, depending on the request made in the previous step, and generates the `grant` signal for LEN5. Keep in mind that since we have separated the 64-bit data requests into two 32-bit data requests, the acceptance of the 64-bit data request is signaled to LEN5, through the assertion of `grant`, only when both 32-bit data requests are accepted by the bus.
4. Saves within the FIFO queue the signals required to complete the Response Phase.
5. Waits for the availability of the data on the X-HEEP bus and the `rvalid` signal, again on one or both ports, depending on the type of request.
6. Saves the first half of the input data into the buffer and waits for the second half, or sends the data directly to the alignment module, the behavior depending on the type of request.
7. Aligns the input data, loads it onto the LEN5 bus, and generates the `rvalid` signal for LEN5. As in the case of accepting the 64-bit data request, in the case of resolving the same request, the `rvalid` signal for LEN5 is asserted only when both 32-bit data are received.

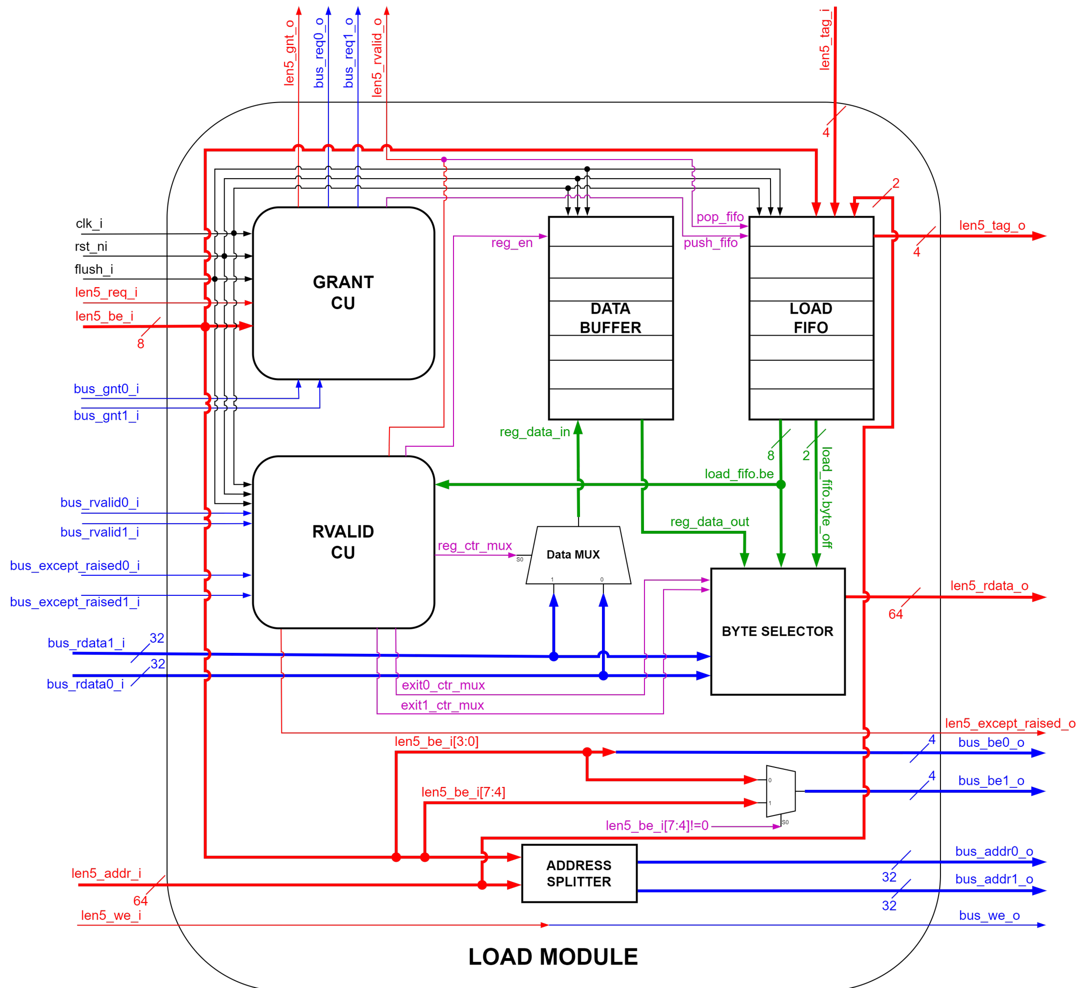


Figure 3.5: Bridge LOAD Module

The module consists of: two Control Units, an Address Splitter, a Data Buffer, a FIFO Queue, and a Byte Selector. The following is a detailed analysis of the components.

3.3.1 Grant Control Unit

The first Control Unit instantiated in the *LOAD Module* is called the Grant Control Unit. Necessary for handling the Address Phase of the OBI protocol, it is in charge of generating the `bus_req` signals to the X-HEEP bus and the `len5_gnt` signal to LEN5, as well as controlling the `push` of the information needed for the Response Phase of the OBI protocol, within the FIFO queue.

Again as with the *Instruction CU*, the approach used for implementation is Mealy's. In fact, memory accesses are one of the major causes of processor pipe stalls due to long

delays caused by memories. A Moore-type implementation would have introduced too many delays due to state-passing waits, so it was preferred to generate control signals combinatorially, speeding up the execution of LOADs.

Unlike the instruction interface of LEN5, which may not be ready to receive instructions, in the case of the LOAD interface, LEN5 is always ready to receive a data, which is why the signal `len5_rready`, although supported by protocol request, is not used within the CU.

In addition to the Clock and Reset signals, whose operation is obvious, there is a `flush` signal that works as a synchronous reset for the CU. In the case of the *LOAD Module* this signal is not used, unlike that in the *Instruction Module*, in fact no `flush` of the data interface has to be done even in the case of branch misprediction, since LOADs have to be executed in any case, the processor will later take care of flushing the pipeline in the case of misprediction.

The FSM within the Control Unit has 3 states: ISSUE, WAIT_GNT0 and WAIT_GNT1. Again we will divide the analysis of the FSM into two parts, future state logic and control signal logic.

The future states logic works starting from the ISSUE state, here the FSM checks if there is a request from LEN5, through the assertion of the signal `len5_req`. If the request is received, it checks its type via the value of the 8 bits of the `len5_be` bus, this in fact has specific values depending on the size of the requested data. There is no need to use all the combinations of the 8 bits of the bus `len5_be` to identify a subpart of the 64 bits, the bridge is able to do this from the address provided, this allows to simplify the generator of the byte enable inside LEN5. Of the four combinations of values generated by LEN5, only one has the first four bits at value 1. This corresponds to the DOUBLEWORD request.

- **8'b11111111** → DOUBLEWORD request, 64-bit data.
- **8'b00001111** → WORD request, 32-bit data.
- **8'b00000011** → HALFWORD request, 16-bit data.
- **8'b00000001** → BYTE request, 8-bit data.

If the request is of the DOUBLEWORD type, the FSM remains in the ISSUE state in the event that both requests sent to the bus are accepted simultaneously, signaled by assertion of both `bus_gnt` signals. Otherwise, if only one of the two requests is accepted, the FSM switches to one of the two WAIT_GNT states, waiting for the remaining second request to be accepted. In the case of a WORD, HALFWORD or BYTE type request, the FSM will always remain in the ISSUE state.

The states of WAIT_GNT0 and WAIT_GNT1 are identical, differing only in the `bus_gnt` signal that the FSM controls. Respectively, the FSM controls the `bus_gnt0`

signal in the former case or the `bus_gnt1` signal in the latter case, returning to the ISSUE state when the signal is asserted.

As introduced earlier, the control signal logic combinatorially generates the signals needed for transactions so that they are available in the same clock cycle in which LEN5 executes a request.

Starting from the state of ISSUE, the logic distinguishes the two types of requests from the value of the first 4 bits of the bus `len5_be`, as happens in the logic of future states. For 64-bit data requests, the FSM forwards the `len5_req` signal, input from LEN5, to both module ports, making two requests to the X-HEEP bus. In this way, the requests occur immediately and there is no need to change state. At the same time, the FSM waits for the assertion of one or both of the `bus_gnt`, signaling the acceptance of the requests. The `len5_gnt` signal is generated by the logical AND of the two `bus_gnt`, so that it would be asserted immediately if both requests were accepted. As we have already pointed out, this is an impossible occurrence in the case of the X-HEEP bus, since it is single-port. The last signal is the `push` for the FIFO queue, generated from the AND between the signal `len5_req` and the OR between the two `bus_gnt`. In fact, the queue must save within it various data needed by the Rvalid CU and the alignment module, before the bus asserts the `rvalid` for the first satisfied request, so that these are already available for Rvalid CU operation. This is done by exploiting the first received between the two input `bus_gnt` (OR operation), while ensuring that there has been a request from LEN5, (AND operation between `len5_req` and the result of the previous OR operation). In the case of 32-bit requests, by design choice, only port 1 of the module is used. The FSM forwards the `len5_req` signal, sent by LEN5, to the `bus_req1` and generates the `len5_gnt` signal from the `bus_gnt1`. The `push` for the FIFO queue is generated by the AND of `len5_req` and `bus_gnt1` alone.

The states of WAIT_GNT0 and WAIT_GNT1 are achievable only if the type of the detected request is DOUBLEWORD. In the WAIT_GNT0 state, the `bus_req0` is kept asserted, waiting for the bus to assert the `bus_gnt0` signal, which will be used to generate the `len5_gnt` signal. All this is necessary because LEN5 keeps the `len5_req` signal asserted for only one clock cycle. Similarly, the WAIT_GNT1 state works in the same way, with mirrored signals. Figure 3.6 shows the diagram of CU states, in the case of a DOUBLEWORD type transaction. The case of WORD, HALFWORD and BYTE is omitted, since the CU always remains in the ISSUE state. The table 3.2 shows the signals generated by the CU and the method of generation.

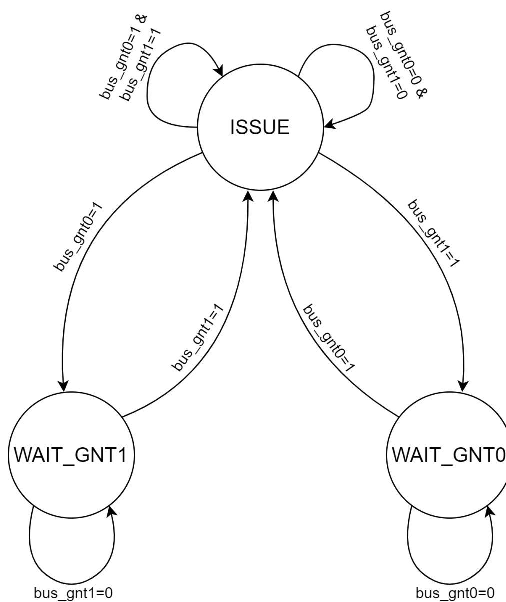


Figure 3.6: State Diagram of the Grant CU for a DOUBLEWORD Request

Signal	Value
State ISSUE - LOAD DOUBLEWORD	
bus_req0	len5_req
bus_req1	len5_req
len5_gnt	bus_gnt0 & bus_gnt1
push_fifo	len5_req & (bus_gnt0 | bus_gnt1)
State ISSUE - LOAD WORD, HALFWORD or BYTE	
bus_req0	0
bus_req1	len5_req
len5_gnt	bus_gnt1
push_fifo	len5_req & bus_gnt1
State WAIT_GNT0	
bus_req0	1
bus_req1	0
len5_gnt	bus_gnt0
push_fifo	0
State WAIT_GNT1	
bus_req0	0
bus_req1	1
len5_gnt	bus_gnt1
push_fifo	0

Table 3.2: Grant CU - Signal Generated and Generation Methods

3.3.2 Address Splitter

The *Address Splitter* module is responsible for generating addresses for LOAD requests to the X-HEEP bus, from the address provided by LEN5. It presents as input the 64-bit address bus, `len5_addr`, and the byte enable, `len5_be`. Instead, as output it presents two 32-bit address buses, `bus_addr0` and `bus_addr1`. It should be noted that, by design choice, 64-bit requests are divided as follows: port 0 responsible for the 32 LSBs, port 1 of the 32 MSB of the requested data.

Its operation is divided into two cases, also determined by the type of request made to the bus. The module is able to distinguish between the two types of requests, 32 bits or 64 bits, based on the value of the `len5_be`.

As for 32-bit requests, WORD, HALFWORD and BYTE, these do not require special handling; the module simply generates on both outputs the 32-bit address, consisting of the 32 LSBs of the `len5_addr` bus. It is important to remember that, as specified in section 3.3.1, regardless of the presence of the address on both output ports, the Grant CU will only generate the request on port 1.

Related to 64-bit requests, we have already introduced in section 2.2, the concept of Misaligned Access. The bridge is designed to be able to support aligned and misaligned accesses. In fact, there is a possibility that data may not necessarily be arranged on the same row, in order to optimize memory occupancy, think for example of embedded systems where ram banks are small and space cannot be wasted. That said, depending on the arrangement of the data, the processor may perform an Aligned access, which implies generating an address that is a multiple of the number of Bytes of which the data is composed. Considering a 64-bit, 8-byte array, the multiple addresses would be all those whose last 3 LSBs are 0, e.g., 0, 8, 16, etc. There could also be the case where the data is not on the 64 bits of one row, but, for example, on the last 32 bits of one row and the first 32 bits of the next. This would force the processor to perform a Misaligned access, generating an address that is not a multiple of the number of bytes of the data, e.g., 4, 12, etc., and to access memory multiple times to retrieve the data.

The module can distinguish between Aligned and Misaligned requests, based on the value of the third bit of the address provided by LEN5. If this is equal to 0, the address will necessarily be a multiple value of 8, so there will be aligned access. Otherwise, the access will be misaligned.

The above remains an incipit for future developments of the bridge, possibly making it compatible with 64-bit aligned ram connected via 32-bit multiport bus. The ram inside X-HEEP is in fact 32-bit aligned with read/write accesses to entire WORDs while the system bus has a single port dedicated to 32-bit data requests. Therefore, the bridge will always duplicate memory access requests in the case of LOAD/STORE DOUBLEWORD,

regardless of whether these are aligned or misaligned, since it is not possible to recover the data in a single access.

The generation of addresses on the two ports is done in two different ways. For aligned accesses, it is sufficient to connect to port 0 the last 32 LSBs of the address given by LEN5, while in port 1 the same portion of bits is connected but with the third one forced to 1. This will automatically result in an addition of 4 on the address given by LEN5. For Misaligned accesses, the 32 LSBs are connected to port 0, as in the previous case. Unfortunately, for the outgoing address from port 1, a value of 4 must be physically added to the address provided by LEN5 by instantiating an adder.

3.3.3 Rvalid Control Unit

The second Control Unit present within the module is named Rvalid Control Unit. It is used to manage the Response Phase of the OBI protocol, through: the generation of the `len5_rvalid` signal, the control of the Data Buffer through the `reg_en` and `req_ctr_mux` signals, for saving the input data in case the transactions were not executed at the same time, and the control of the muxes on the data bus, via the signals `exit0_ctr_mux` and `exit1_ctr_mux`, which are needed to align the partial input data, in case of 64-bit transactions. It is also responsible for generating the `len5_except_raised` signal, to be supplied to the processor in case homonymous signals are asserted from the bus.

Like all CUs within the Bridge, this one also features a Mealy approach with the aim of accelerating the delivery of data to the processor and not stalling the pipe for too long, while simultaneously reducing the number of states required.

We will leave out the obvious operation of the `clock` and `reset` signals; as for the `flush` signal, its operation was explained in Section 3.3.1.

The control unit has a 5-state FSM:

1. IDLE
2. WAIT_RVALID0
3. WAIT_RVALID1
4. WAIT_RVALID0_ERR
5. WAIT_RVALID1_ERR

We begin the analysis of the FSM from the future state generation logic. The initial state of the FSM is IDLE, here the two types of requests made in the previous step are distinguished from the first four bits of the `len5_be` bus. Note that at this time, the `len5_be` bus is no longer available, as the processor has already made another instruction request. The byte enable that the Rvalid CU uses is provided by the FIFO, which is controlled by the Grant CU.

Once the type of request is identified, in the event of a DOUBLEWORD, the FSM checks the signals `bus_rvalid0` and `bus_rvalid1`. If both are asserted, indicating that both requests have been satisfied and both data are present on the input bus, the FSM remains in the IDLE state. Otherwise, if only one of the two requests has been validated, the FSM checks for the presence of the exception signal associated with the received `bus_rvalid` signal. If no exception is present, the FSM switches to the WAIT_RVALID state opposite to that of the asserted `bus_rvalid` signal; if, on the other hand, the exception is present, it switches to the mirrored WAIT_RVALID_ERR state. Finally, if no `bus_rvalid` is asserted, the FSM remains in the IDLE state.

In the other eventuality that the request is for WORD, HALFWORD, and BYTE, the FSM will always remain in the IDLE state.

The WAIT_RVALID0 and WAIT_RVALID1 states allow the FSM to wait for the `bus_rvalid` signal that has not yet been received. In the WAIT_RVALID0 state, the FSM checks the assertion of the `bus_rvalid0` and, when this happens, returns to the IDLE state. Similarly, in the state of WAIT_RVALID1, the FSM checks the `bus_rvalid1` signal and upon its assertion, returns to the IDLE state.

Concerning the last two states, WAIT_RVALID0_ERR and WAIT_RVALID1_ERR, the logic of the future states has no difference from the WAIT_RVALID0 and WAIT_RVALID1 states. They turn out to be necessary to internally encode the exception received in the IDLE state when only one of the two requests has been satisfied. This avoids the implementation of registers and additional checks to sample the incoming `exception` signal.

Let proceed now to the analysis of the control signal generation logic. Starting from the IDLE state, the FSM selects the type of request according to the bus `len5_be`, provided to it by the FIFO. With a DOUBLEWORD request, the signal `len5_rvalid` is generated from the AND of the two signals `bus_rvalid0` and `bus_rvalid1`, in case the requests were fulfilled at the same time. The signal `reg_en`, which is necessary for enabling the *Data Register*, is generated by the XOR of the two signals `bus_rvalid`. In fact, the register must be enabled at the time the data is present on the bus and only in case one of the two parts of the data arrives before the next. The bus where the data is present is then connected to the register itself by means of a mux, controlled by the signal `reg_ctr_mux`, generated from the signal `bus_rvalid1`. If the data on port 0 were to arrive first, the `bus_rvalid1` would not be asserted, connecting the port 0 bus to the register input; conversely, the port 1 bus would be connected. Now follows the generation of the two control signals for the alignment module, `exit0_ctr_mux` and `exit1_ctr_mux`. The first of the two controls the multiplexer on whose inputs are the partial data pairs, `bus_data1` + *Data Buffer* or *Data Buffer* + `bus_data0`, in the default IDLE state of 0 because the FSM has not yet identified which of the two data pairs arrived first. The second signal controls the output of the alignment module, connecting either the previously selected pair

or the pair `bus_data1` + `bus_data0`. As mentioned in the section on future state logic, the FSM uses the IDLE state even if both requests are satisfied at the same time, so the signal is kept at 1, directly connecting the data buses to the input data bus at LEN5. The last signal is `len5_exception_raised`, generated by the OR between the two signals `bus_exception_raised0` and `bus_exception_raised1`. The presence of any one of the two `exception`, indicating the wrong data read, must in fact be forwarded to LEN5 regardless of which one is asserted.

For requests other than the DOUBLEWORD type, as specified above, only port 1 is used. The signal `len5_rvalid` is directly connected to the input signal `bus_rvalid1`. There is no need to use the *Data Buffer*, which will then be disabled by deasserting `reg_en`, also it does not matter which of the two inputs is connected to the Buffer, as it is disabled. The signal `reg_ctr_mux` is then also deasserted. The signals `exit0_ctr_mux` and `exit1_ctr_mux` are not used in this case for data alignment, the input address being used directly. They are therefore set to 0 by default. Finally, the signal `len5_except_raised` is directly connected to the signal `bus_except_raised1`.

Continuing with the pair of states WAIT_RVALID0 and WAIT_RVALID1, it is important to remember that the FSM can proceed in these states only if the requests have not been fulfilled at the same time and a `exception` has not been received. That said, in the WAIT_RVALID0 state, the FSM generates the signal `len5_rvalid` from the input `bus_rvalid0`. The partial data, made available in the IDLE state, has already been saved in the *Data Buffer* and must be preserved by disabling its update, deasserting the signal `reg_en`. The signal `reg_ctr_mux` is set to 0 by default, as it is no longer relevant which input bus is connected to the register, since it is disabled. The alignment module is driven, bringing to the output the pair *Data Buffer* + `bus_data0`, `exit0_ctr_mux`=0 and `exit1_ctr_mux`=0, being arrived first the partial data on port 1, which is now in the *Data Buffer*. The signal `len5_except_raised` is directly connected to `bus_except_raised0`, there having been no exception in the previous state. Similarly, the WAIT_RVALID1 state provides the same behavior. It differs only in the signal connections: `len5_rvalid`, now connected with `bus_rvalid1`, `len5_except_raised` connected directly to `bus_except_raised1`, and `exit0_ctr_mux`=1, which connects to the output of the alignment module, the pair `bus_data1` + *Data Buffer*.

The states required to encode an *exception* received during the IDLE state, WAIT_RVALID0_ERR and WAIT_RVALID1_ERR, are identical to the states of WAIT_RVALID0 and WAIT_RVALID1 described above. They have only one difference, the signal `len5_except_raised` is asserted directly, signaling to the processor the presence of an *exception*, regardless of what will happen during the reception phase of the second part of the requested data.

The diagram in figure 3.7 shows the progress of CU states, in the case of a DOUBLEWORD transaction. The WORD, HALFWORD, and BYTE cases are deliberately omitted since the Control Unit constantly remains in the IDLE state.

The tables 3.3 and 3.4 show the various signals generated by the CU and the methods of generation.

Figure 3.8 depicts two LOAD DOUBLEWORD transactions, during a standard execution and with the occurrence of an exception. In the first section, the signals and the data buses as well as address buses are shown during the execution of all steps of the OBI protocol, for a LOAD without exception. Several behaviors described in the previous sections can be seen. First, the signal `len5_req`, asserted by LEN5 for only one clock cycle, is forwarded to both ports at the same time it is detected. The first grant asserted by the bus is the `bus_gnt1`, note in fact how the request on port 1 is deasserted after only one cycle while the same request on port 0 is held by the bridge until the `bus_gnt0` is received. Only then the bridge asserts the `len5_gnt`. LEN5 guarantees the correctness of the incoming address and byte enable only until the grant is asserted, so these signals last exactly two cycles; however, it is not a given that, after the grant is asserted, they become invalid. Also important is the signal `push_fifo`, asserted the moment the first request is accepted, with `bus_gnt1` asserted. In the second phase of the protocol, it is crucial to pay attention to the `rvalid` signals; consequently to the acceptance of the first request on port 1, the bus responds with a partial data asserting the `bus_rvalid1`. Since the second `rvalid` is not present at the same time, the Rvalid CU asserts the `reg_en` and connects the input bus, via the `reg_ctr_mux`, to the *Data Buffer* register, saving the data inside, as visible in the `bus_data1_i` and `data_buffer` buses. In the clock cycle following the reception of `bus_rvalid1`, `bus_rvalid0` is also asserted. The Rvalid CU, at this time in the state of WAIT_RVALID0, drives the Byte Selector via the signals `exit0_ctr_mux`=0 and `exit1_ctr_mux`=0, bringing the sorted data to the output bus in the correct manner.

Instead, the second section of the image shows a LOAD transaction with, however, the assertion of the exception related to the first partial data. The pattern of Address Phase and Response Phase is essentially identical to the transaction illustrated above. It is important to note one major difference, however, at the time of the assertion of the signal `bus_exception_raised0` in conjunction with the arrival of the signal `bus_rvalid0`, this is first forworded directly to LEN5, thereafter the Rvalid CU switches its state to WAIT_RVALID1_ERR keeping the `len5_exception_raised` asserted even in the cycle following the one in which the exception was received, ensuring that LEN5 receives it along with `len5_rvalid`.

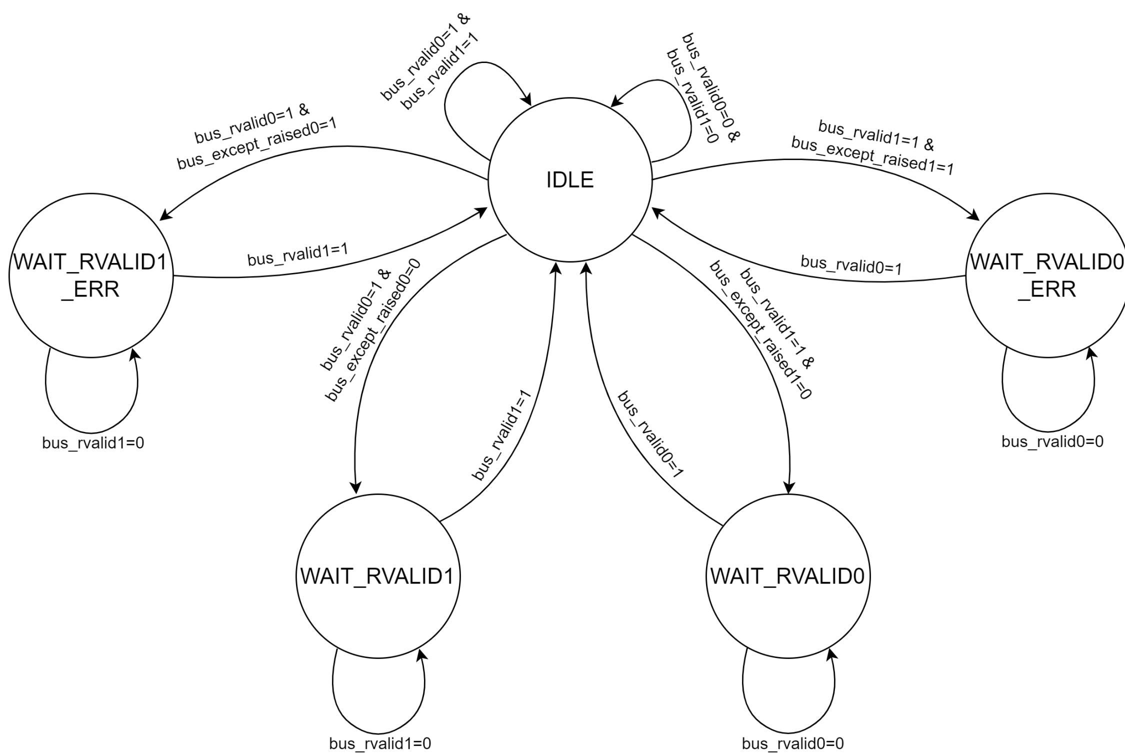


Figure 3.7: State Diagram of the Rvalid CU for a DOUBLEWORD Request

Signal	Value
State IDLE - LOAD DOUBLEWORD	
len5_rvalid	bus_rvalid0 & bus_rvalid1
reg_en	bus_rvalid0 ⊕ bus_rvalid1
reg_ctr_mux	bus_rvalid1
exit0_ctr_mux	0
exit1_ctr_mux	1
len5_except_raised	bus_except_raised0 | bus_except_raised1
State IDLE - LOAD WORD, HALFWORD or BYTE	
len5_rvalid	bus_rvalid1
reg_en	0
reg_ctr_mux	0
exit0_ctr_mux	0
exit1_ctr_mux	0
len5_except_raised	bus_except_raised1

Table 3.3: Rvalid CU - Signal Generated and Generation Methods - IDLE STATE

Signal	Value
State WAIT_RVALID0	
len5_rvalid	bus_rvalid0
reg_en	0
reg_ctr_mux	0
exit0_ctr_mux	0
exit1_ctr_mux	0
len5_except_raised	bus_except_raised0
State WAIT_RVALID1	
len5_rvalid	bus_rvalid1
reg_en	0
reg_ctr_mux	0
exit0_ctr_mux	1
exit1_ctr_mux	0
len5_except_raised	bus_except_raised1
State WAIT_RVALID0_ERR	
len5_rvalid	bus_rvalid0
reg_en	0
reg_ctr_mux	0
exit0_ctr_mux	0
exit1_ctr_mux	0
len5_except_raised	1
State WAIT_RVALID1_ERR	
len5_rvalid	bus_rvalid1
reg_en	0
reg_ctr_mux	0
exit0_ctr_mux	1
exit1_ctr_mux	0
len5_except_raised	1

Table 3.4: Rvalid CU - Signal Generated and Generation Methods - WAIT STATES

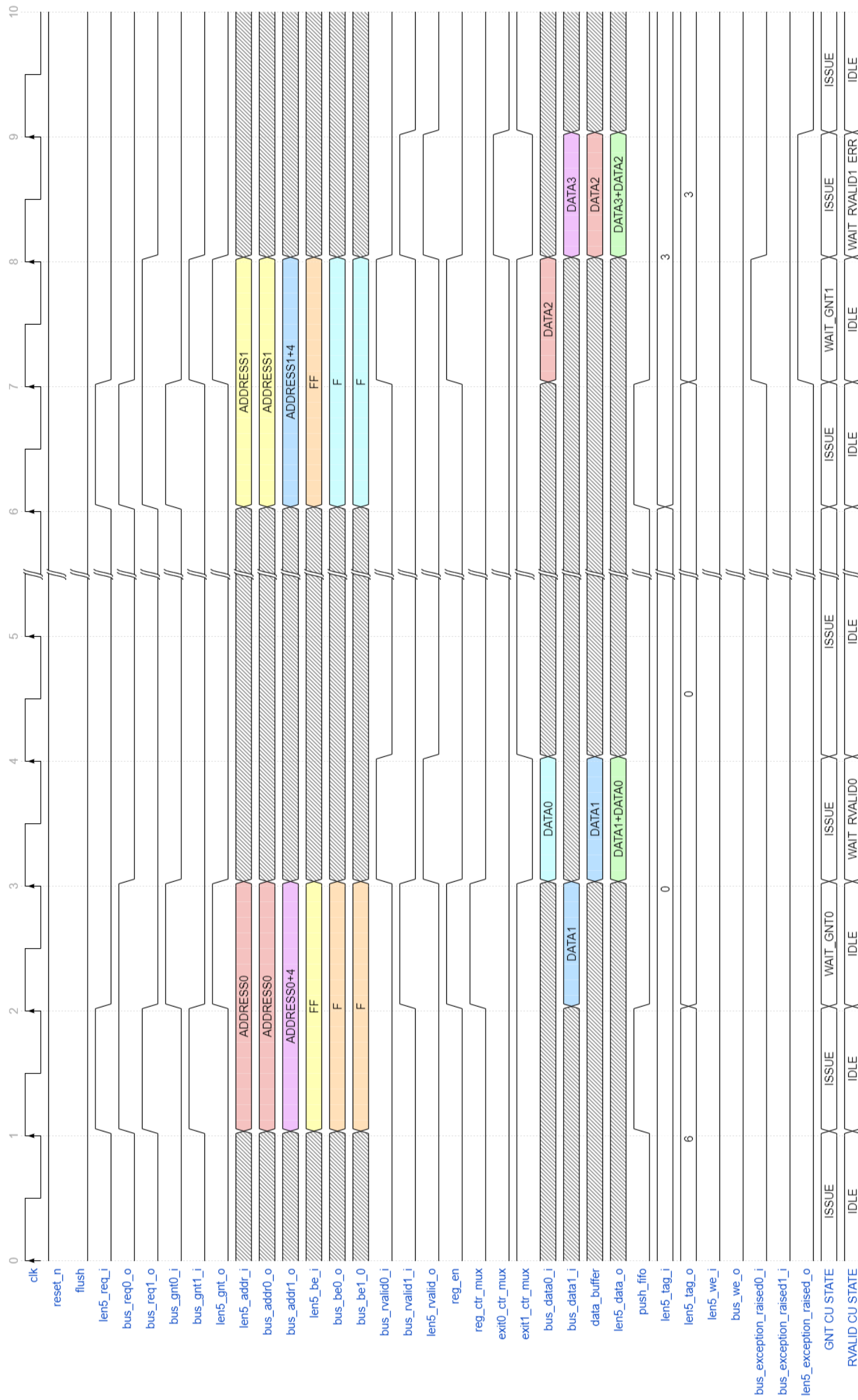


Figure 3.8: LOAD DOUBLEWORD - Normal Execution and Execution with Exception Present

3.3.4 Data Buffer and Load FIFO

The register needed to save the partial input data, in case of unfulfilled requests at the same time, is called *Data Buffer*. The implementation is that of a normal 32-bit register, so we will not dwell on in-depth explanations. In addition to the `clock` and `reset` signal, it has a `flush` signal, acting as a synchronous reset, and an enable signal, `reg_en`, controlled by the Rvalid CU, presented in section 3.3.3. As input, it receives a partial data from one of the two input buses to the LOAD Module, `bus_data0` or `bus_data1`, which of the two buses is controlled by the Rvalid CU, depending on which of the partial data is provided first. Finally, the output is provided to the alignment module.

The second data structure in the module is a FIFO queue. At the end of the Address Phase of the OBI protocol, once the `len5_gnt` signal is received, LEN5 no longer guarantees to maintain fixed signals on its interface. However, many of these signals are needed in the Response Phase: to distinguish between different types of requests, to align data, and to provide LEN5 with the tag of the completed LOAD instruction. The FIFO queue, identical to the one used in the *Instruction Module*, allows the following to be stored within it: the signal `len5_be`, which is needed for the identification of the request in both the Rvalid CU and the alignment module, the 2 LSBs of the bus `len5_addr`, which are needed within the alignment module for 32-bit data requests, and the TAG of the LOAD instruction being executed, so as to provide it to LEN5 when it is completed. As for the `push` signal, we have already defined how this is generated. The `pop` signal, on the other hand, is directly controlled by the `len5_rvalid` signal. In fact, once the data is received, signaled by `bus_rvalid`, it is no longer necessary to store the signals within the FIFO.

3.3.5 Byte Selector

At the time the requested data is received by the bridge, it is not yet in the correct form to be delivered and processed by the processor. In fact, it is necessary to select and align the received data, depending on the type of request made by LEN5. The module assigned to this task is called the Byte Selector. It consists of two main sections, each capable of handling a type of request, again identified from the `len5_be` bus, out of the FIFO.

We begin the analysis from the section responsible for DOUBLEWORD type requests, whose data must be aligned taking into account their arrangement within memory and the time at which the data is available on the bridge interface. The arrangement of the data is of the Little Endian type, that is, with the LSBs of the data corresponding to the lower value address. As presented in Section 3.3.2, by design choice it was decided to assign the lower value address to port 0 and the higher value address to port 1, so the LSB portion will be available at port 0 and the MSBs at port 1.

This section, visible in figure 3.9, is essentially composed of two cascaded multiplexers,

controlled through their selectors, by the signals `exit0_ctr_mux` and `exit1_ctr_mux` generated by the Rvalid Cu. The first multiplexer is used to compose the data, in case the two parts arrived at different times and one is saved within the *Data Buffer*. In the case of the X-HEEP bus, having only one port, this always happens. Through the switch driven by `exit0_ctr_mux`, it is possible to connect on the 64-bit output of the multiplexer, the union between the `bus_rdata1` bus and the contents of the *Data Buffer* or the contents of the *Data Buffer* and the `bus_rdata0` bus. Note the order of in which the buses are connected, which is critical for restoring positional notation from the Little Endian notation of the memory. The second multiplexer, placed between the first multiplexer and the output connected to the processor's data bus, uses the `exit1_ctr_mux` signal as a selector to connect the combination of the module's two input buses, `bus_rdata1` and `bus_rdata0` in that order, directly to the LEN5 bus, in case the requests are completed at the same time. Otherwise, it connects the output of the first multiplexer to the data bus of LEN5.

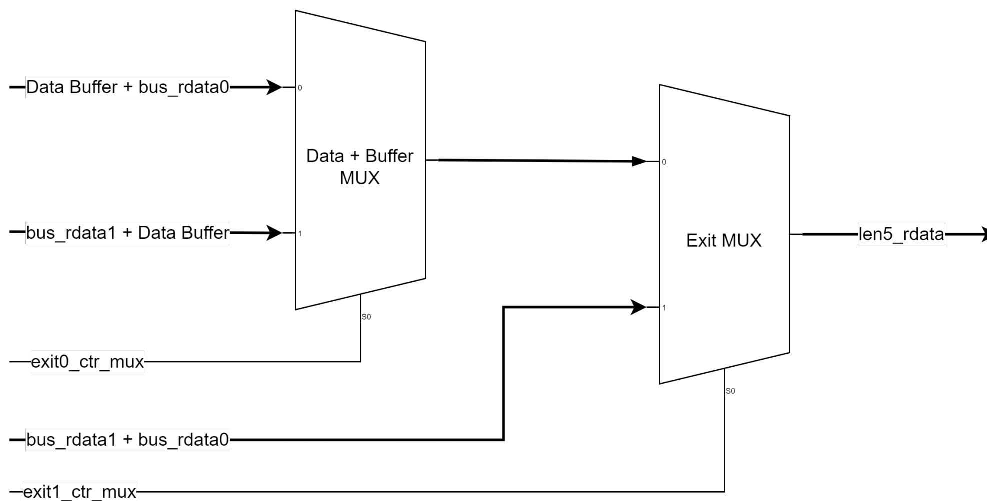


Figure 3.9: Byte Selector - 64 bits Selection

The second section, related to 32-bit requests, does not use the signals generated by the Rvalid CU for alignment, but instead takes advantage of the 2 LSBs of the address provided by LEN5, which are present within the FIFO queue. As mentioned earlier, the ram instantiated within X-HEEP ignores the last two bits of the input address, providing in read a 32-bit data, regardless of the request made. However, the processor could access the memory by making BYTE requests, with addresses of any value, or HALFWORD requests, with addresses multiple of 2, still receiving a WORD as a response. The *Byte Selector* solves this problem by exploiting two multiplexers, appropriately controlled by the 2 LSBs of the address given by LEN5, generating the HALFWORD and BYTE corresponding to the address of the request, from the WORD received, and then connecting the input bus to

the output bus, in the manner appropriate to the request made.

Going into more detail, a logical diagram of the operation of the *Byte Selector* can be seen in figure 3.10. Note that no additional registers have been used; what looks like registers in the diagram are actually the various combinations by which the input data bus can be connected to the output data bus.

Starting with the input WORD, depending on bit 1 of the address, the module either connects the 16 MSBs of the input bus with the 16 LSBs of the output bus, in the case where bit[1]=1, or keeps the 16 input LSBs in the same position on the output, in the case where bit[1]=0. In both cases, the 16 input MSBs are also connected to the 16 output MSBs. The problem of repeating MSBs does not arise since LEN5 will take care of the sign extension, eliminating repetitions. Similarly, starting from the combinations of signals generating the HALFWORD, depending on bit 0 of the address, the module either connects the 8 MSBs of the HALFWORD to the 8 LSBs of the output, in the case where bit[0]=1, or keeps the 8 LSBs in the same position, in the case where bit[0]=0. For this configuration, both the 16 MSBs of the WORD and the 8 MSBs of the HALFWORD are also copied to the same position. Finally, all three configurations are the input to a multiplexer controlled by the value of the 4 LSBs of the `len5_be` bus, which is used to select which of the configurations to connect to the output bus, `len5_rdata`.

1. **4'b1111** → LOAD WORD
2. **4'b0011** → LOAD HALFWORD
3. **4'b0001** → LOAD BYTE

The result will be to have the BYTE or HALFWORD selected in the manner described in 2.2, always occupying the bit portion on the right-hand side of the 64-bit output, so that LEN5 can extend the sign and create a usable WORD.

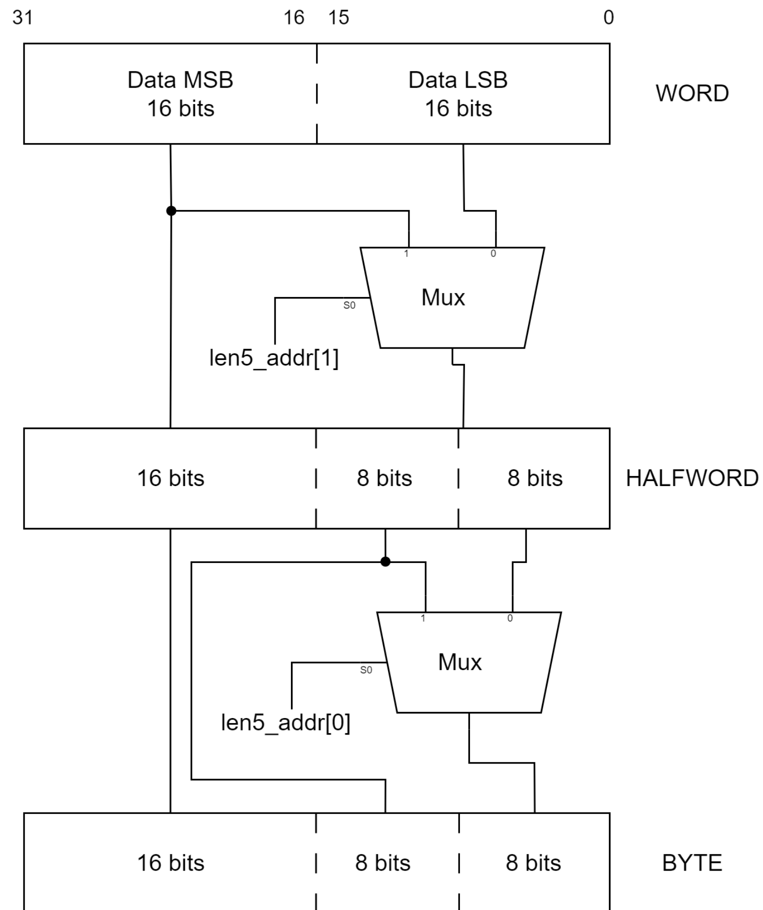


Figure 3.10: Byte Selector - 32 bits Selection

3.3.6 Additional Signals

The signal `bus_we`, which is necessary for the distinction between LOAD and STORE requests, is directly connected to the signal `len5_we`. With the splitting of the processor's LOAD/STORE interface, it becomes superfluous, being constantly at a fixed value. It is retained for protocol compatibility.

The values of the buses `bus_be0` and `bus_be1` are assigned from that of the bus `len5_be`. For the bus `bus_be0`, the value is always equal to that of the 4 LSBs of the bus `len5_be`, while for the bus `bus_be1`, the value can vary between: the 4 MSBs of the bus `len5_be` if the transaction is 64 bits or the 4 LSBs of the same bus in case of a 32-bit transaction.

3.4 STORE Module

The last of the 3 component modules of the Bridge is the *STORE Module*, visible in the figure 3.11, intended to handle STORE requests to the X-HEEP bus. For comprehensibility, in the schematic the various signals have been divided by function, via color. In red the signals coming from or destined for LEN5, in blue the signals coming from or destined for the X-HEEP bus, in green the data signals used internally, and in violet the control signals.

As already mentioned in the section on the *LOAD Module* and described in section 2.3, its implementation is due to the impossibility of executing a single STORE DOUBLEWORD request to the X-HEEP bus because of the size of the data bus itself. It is therefore necessary for the module to distinguish the type of input request from LEN5 and based on the type of the latter, send a single 32-bit data request to the bus in the case of STORE WORD, HALFWORD and BYTE or two simultaneous 32-bit data requests in the case of STORE DOUBLEWORD. Output requests from the two module ports will be made sequential by the crossbar to which the bridge connects so that the bus can handle them.

For this module, too, it would have been possible to implement a single port and directly serialize requests to the X-HEEP bus. However, the implementation of two ports and simultaneous requests was preferred, as in previous cases, to make the bridge compatible even with multiport buses supporting this type of request.

The operations performed by the module are given below:

1. Asserted the signal `len5_req` for a STORE, the module forwards the request to port 1 or both ports, depending on the type of request received, identified by the bus value `len5_be`.
2. Correctly aligns the data to be saved in memory according to the type of STORE request received.
3. Asserts the signal of `len5_gnt` when the request(s) have been accepted by the bus, signaled by the assertion of the signals `bus_gnt0` and `bus_gnt1`.
4. Saves within the FIFO queue the value of the signals required for the second phase of the OBI protocol.
5. Asserts the signal `len5_rvalid` when the request(s) have been completed.

The implementation of the module provides the use of: two Control Units dedicated to the management of OBI protocol requests, an Address Splitter for the generation of addresses, a FIFO queue used to save information useful in the second phase of the OBI protocol, and a Data Aligner necessary for the alignment of the output data to the X-HEEP bus and the generation of the correct bus `bus_be`. Given the use of the same communication protocol and the modularity of the components implemented, it was possible to use

the same Control Units, Address Splitter and FIFO queue already explained in the section on the *LOAD Module*. Therefore, the analysis of the components already illustrated in the previous paragraphs and the additional signals will be left out, focusing only on the differences present.

The signals saved within the FIFO queue differ from those saved in the internal FIFO of the *LOAD Module*. In fact, it is not necessary to save the 2 LSBs of the address sent by LEN5 since the alignment is performed in the Address Phase of the OBI protocol, when the address is still present at the input. There is also no need to implement a *Data Buffer* to save the data, in fact the data is kept as input by LEN5 until the module asserts the `len5_gnt` signal. At that time, the X-HEEP bus has already taken over the data to be written to memory, the `len5_wdata` bus being part of the Address Phase of the OBI protocol, as visible in the 2.1 table. The signals needed to control the Buffer and data routing mux, generated by the *Rvalid CU*, remain unused.

The alignment of data output from the module differs from that seen in the *LOAD Module*. In this case, an aligner called *Data Aligner* is implemented, which will be explained in detail.

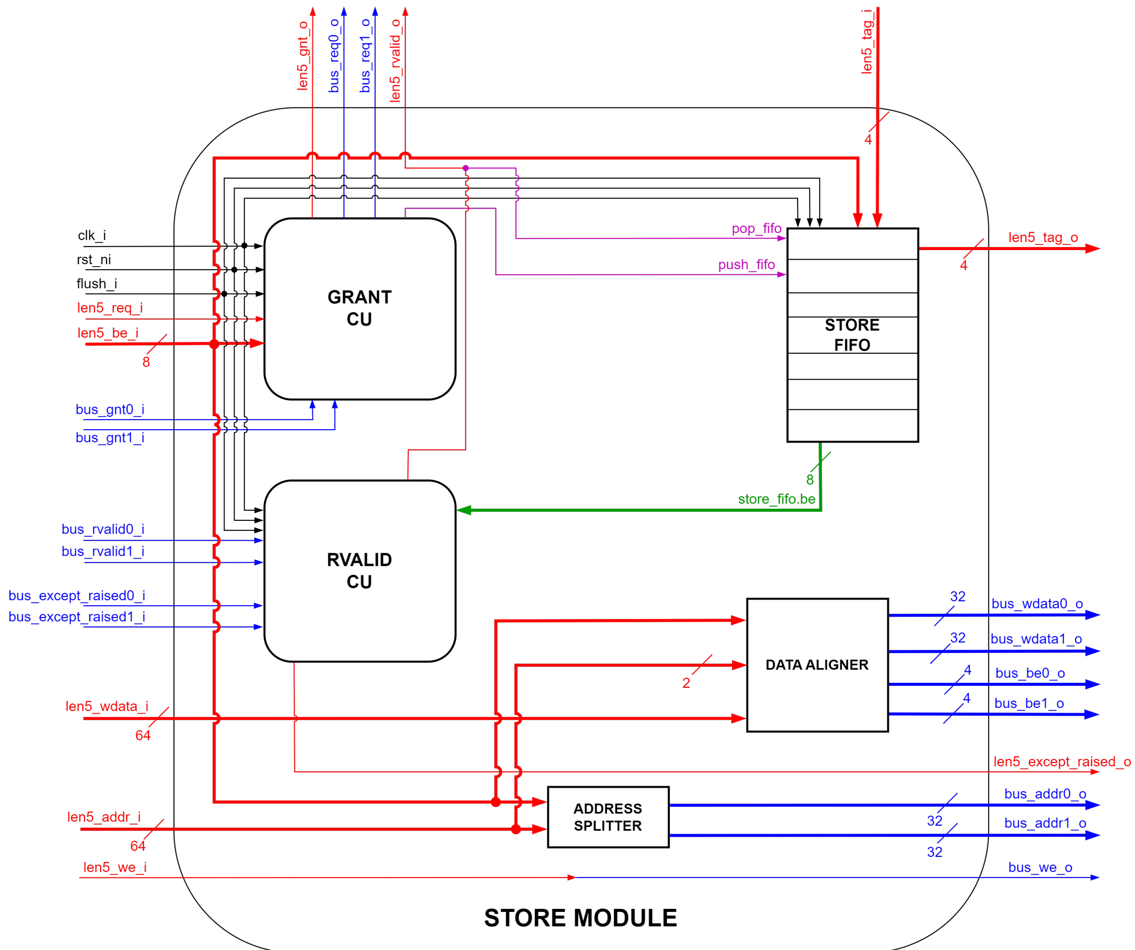


Figure 3.11: Bridge STORE Module

3.4.1 Data Aligner

The *Data Aligner* is responsible for aligning the data to be written to memory and for generating the `bus_be` signal needed to signal which portion of the supplied data is to be written. One of the features of the ram present within X-HEEP, which the *Data Aligner* exploits, is the selection of the portion of the supplied data to be updated. Through the `bus_be` bus, in fact, a single BYTE or a single HALFWORD of the WORD available on the input bus can be selected to be updated within memory without rewriting the entire WORD. Always keeping in mind, however, that the ram works with 32-bit data, although it can select a subset to write. This implies that the data provided will have to be a WORD in any case.

The distinction of transaction type, as in the other modules, is made through the first 4 MSBs of the `len5_be` bus, available in the FIFO. For 64-bit transactions, no alignment

is necessary outside the correct arrangement of the two partial data on the ports, so as to respect the Little Endian encoding of the memory. In this case, the two `bus_be0` and `bus_be1` correspond to the 4 LSB and 4 MSB of the `len5_be`, respectively. For 32-bit transactions, the *Data Aligner* generates the corresponding HALFWORD and BYTE from the WORD present in the input, via appropriate connections of the `len5_wdata` bus to the output bus.

It is important to note that LEN5 provides the data to be written to memory, in positional notation with extended sign. This implies that the data will be placed on the rightmost bits of the 64 of which the provided DOUBLEWORD is composed. We speak of WORD being present in the input even though LEN5 sends a DOUBLEWORD, since the *Data Aligner* ignores the 32 MSBs, which are only needed for the sign extension, and only works on the 32 LSBs.

The address of the location in memory to which the data is to be written will be compatible with the specification described in 2.2.

Leaving aside the simple case of WORD, in the other eventualities where we have data of type HALFWORD or BYTE, it should be shifted, bringing it to occupy the portion of the 32 bits corresponding to the address given.

However, by being able to use the partial data selection feature built into X-HEEP's ram, in the case of HALFWORD or BYTE, it is possible to replicate the supplied data by saturating the 32 bits of a WORD and provide this as the data to be written to memory along with a properly configured `bus_be` to select the correct portion of the data to be saved. This saves the instantiation of a barrel shifter and limits the inserted delay.

In particular, the HALFWORD is generated by replicating the 16 LSBs of the WORD provided by LEN5, resulting in a 32-bit data consisting of two repetitions of the 16 LSBs. Similarly, BYTE is generated by replicating the 8 LSBs of the input WORD, resulting in a 32-bit data consisting of four repetitions of the 8 LSBs. A logical diagram of the operation of the *Data Aligner* is shown in Figure 3.12. Note that none of the visible structures implements a register; what is represented are the various connections that starting from the input bus, generate WORD, HALFWORD and BYTE.

Once the possible outputs have been generated, the output of the *Data Aligner* is selected based on the three possible combinations of the 4 LSBs of the `len5_be` bus. At the same time the `bus_be1` is also generated, only port 1 being used for 32-bit requests. The `bus_be0` remains identical to the 64-bit case. The possible combinations of the 4 LSBs of the `len5_be` and their outputs are listed below:

1. **4'b1111** → STORE WORD, the selected output is the entire WORD and the bus `bus_be1` corresponds to the 4 LSBs of the bus `len5_be`.
2. **4'b0011** → STORE HALFWORD, the selected output is the HALFWORD and the

bus `bus_bel` must be appropriately configured according to the 2 LSBs of the address provided by LEN5, in order to save the HALFWORD in the correct location. Specifically, if the `len5_addr`[1]=1, the HALFWORD selected will be the one occupying the 16 MSB of the 32 bits of the WORD generated by the *Data Aligner* and the `bus_bel`=4'b1100. Conversely, the lower HALFWORD will be selected and `bus_bel`=4'b0011.

3. **4'b0001** → STORE BYTE, the selected output is the single BYTE. In this case, there are four possible `bus_bel` bus configurations, depending on the 2 LSBs of the memory access address.
   (a) **`len5_addr`[1]=1 and `len5_addr`[0]=1** → the fourth BYTE is selected and `bus_bel`=4'b1000.
   (b) **`len5_addr`[1]=1 and `len5_addr`[0]=0** → the third BYTE is selected and `bus_bel`=4'b0100.
   (c) **`len5_addr`[1]=0 and `len5_addr`[0]=1** → the second BYTE is selected and `bus_bel`=4'b0010.
   (d) **`len5_addr`[1]=0 and `len5_addr`[0]=0** → the first BYTE is selected and `bus_bel`=4'b0001.

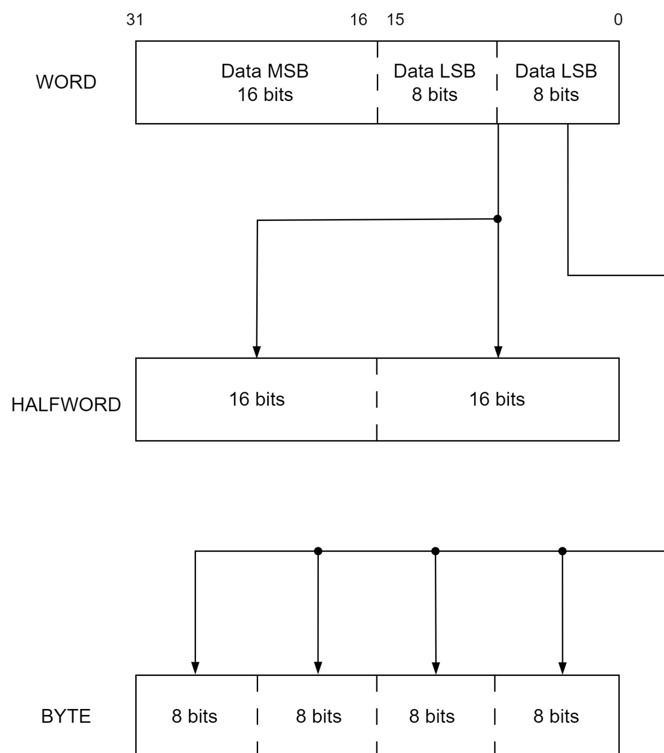


Figure 3.12: Data Aligner - 32 bits Section

Chapter 4

Experimental Results

This chapter will discuss the results obtained from the design of the architecture, through functional verification of module operation and its synthesis. Each component was simulated using the QuestaSim simulator, while the synthesis was performed using the Synopsys Design Compiler synthesizer.

4.1 Functional Verification

The testing phase of an architecture represents the critical part of the entire project. Building test environments capable of ensuring the proper functioning of complex architectures turns out to be one of the longest lasting parts of the entire development process. A popular testing methodology in the design verification landscape is Unit Testing, which is the testing of various standalone components to ensure their proper implementation. The use of this methodology is the obligatory choice at this stage of verification of the realized module. Subsequently, once the Bridge has been connected to LEN5 and the processor introduced within X-HEEP, it is the platform itself that will allow its final validation through a System Test, that is, testing the interfacing of the module with the various components of the system that talk through it.

The realized module is fully described in System Verilog. The use of the *Verification* features available in the language largely facilitated the development of a modular Testbench, including all possible relevant use cases, of the Bridge module.

The Testbench has 3 memories within it, one used for instructions and two used for data, each of which is populated with instructions or data randomly, through the use of the `std::randomize` function.

The testbench structure is organized into three sections: one related to the Instruction Module, one related to the LOAD Module, and one related to the STORE Module. In each,

a variable number of transactions of different types are executed, covering all case scenarios for each module. It is possible to select the test to be executed, by correctly setting the TEST_TYPE variable, declared as a testbench parameter. Two other parameters adjust the duration of the clock cycle and the memory depth.

Each test runs a transaction based on the OBI protocol, appropriately setting the Bridge inputs to simulate the same behavior as the X-HEEP interface. The module outputs are automatically checked by a series of **assert**, verifying the correctness of the results.

4.1.1 Instruction Module Simulation

Two tests are performed for the *Instruction Module*, covering the case of an instruction request during which LEN5 is ready in both phases of the OBI protocol and the case where LEN5 is not ready during the Response Phase.

In figure 4.1 the request and reception of the instruction can be seen. The testbench asserts the `len5_req` signal, being propagated to the bus, and in the same clock cycle it asserts the `bus_gnt`, accepting the request. At the next cycle the instruction is available on the `bus_rdata` and the `bus_rvalid` is asserted. The behavior is identical to that envisaged in 3.3.

More interestingly, the transaction shown in figure 4.2, where upon assertion of the signal `bus_rvalid` from the X-HEEP bus, LEN5 turns out not to be ready to receive the instruction, signaled by the low value of the signal `len5_rready`. This results in the state transition of the CU present in the instruction module, resulting in the `len5_rvalid` signal remaining at a high value, even in the clock cycles following the reception of the data, where X-HEEP has already deasserted the `bus_rvalid`. At the same time, the instruction received by the bridge is maintained on the output bus even with the presence of random data on the input bus, thanks to the module's internal register. With the assertion of the signal `len5_rready`, the processor is again able to receive instructions, so the module's CU returns to the IDLE state and `len5_rvalid` returns to 0, in the next clock cycle. The simulation performed coincides with the behavior projected in 3.4.

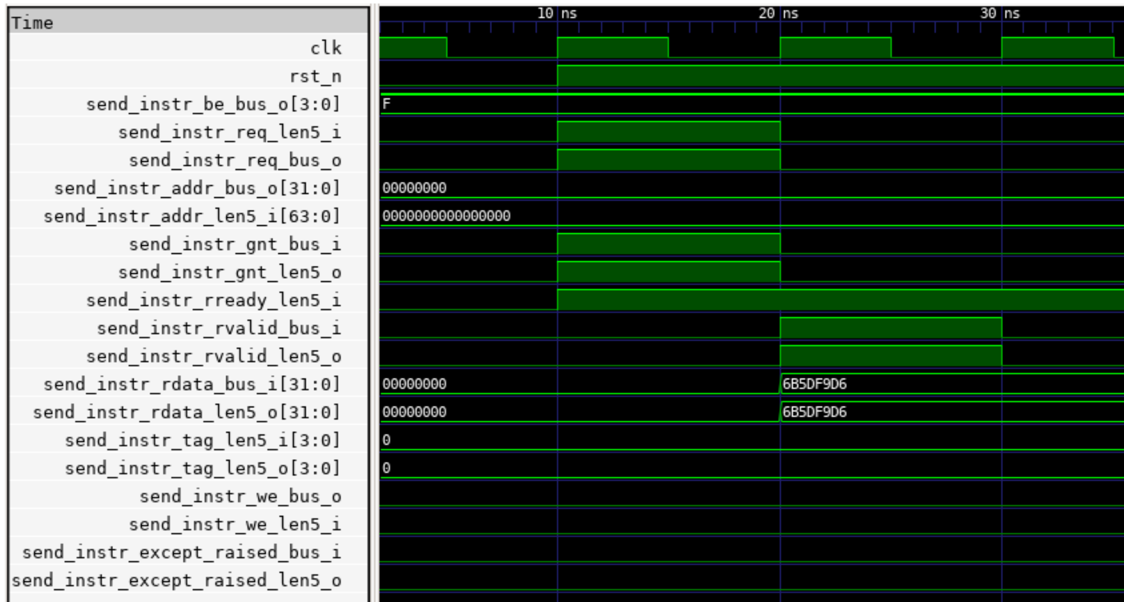


Figure 4.1: Execution of an Instruction Request and Response

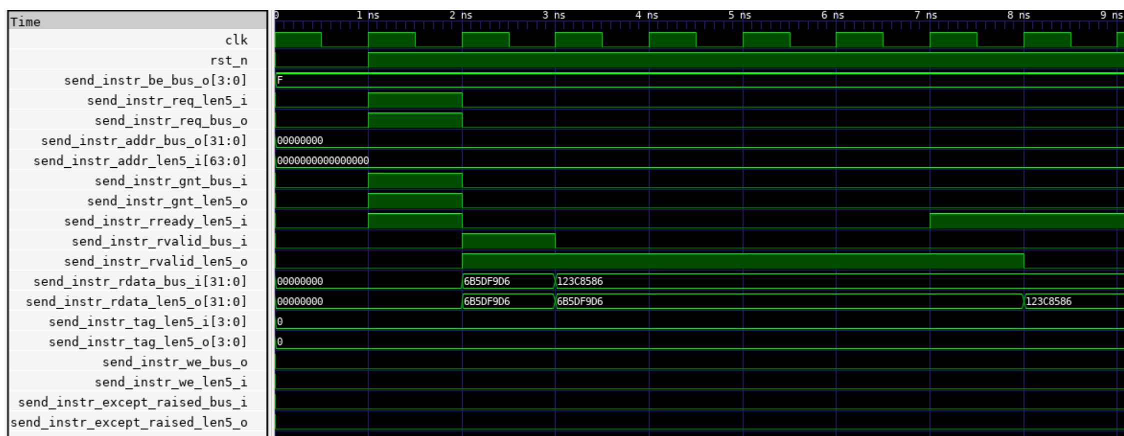


Figure 4.2: Execution of an Instruction Request and Response with Rready Signal Deassertion

4.1.2 LOAD Module Simulation

The section of tests dedicated to the LOAD Module includes six different types of tests, starting with transactions related to 32-bit data, such as BYTE, HALFWORD and WORD, and ending with transactions related to 64-bit data, DOUBLEWORD, executed on both dual-port and single-port buses. In addition, one of the tests is devoted to receiving exceptions during the execution of a DOUBLEWORD request.

We will avoid analyzing again the operation of 32-bit data requests, since they are similar in structure to instruction requests. What it is useful to dwell on is the generation of the output data to LEN5, verifying the operation of the Byte Selector. In the simulations visible in figures 4.3a, 4.3b and 4.4a, there are all possible variations of the WORD output from the bridge, depending on the byte enable.

For a LOAD Byte, the testbench generates a `len5_be` equal to 1, selecting the BYTE access type, and an address equal to 1, thus requesting the 2 BYTE of the WORD supplied from memory. Looking at the `bus_rdata1` and `len5_rdata` buses, port 1 being the only one on which the 32-bit data request is executed, as explained in previous chapters, note that the second BYTE of the data input is correctly shifted to the last 8 bits of the output DOUBLEWORD.

Similarly for a LOAD Halfword, the testbench generates a `len5_be` equal to 3, 0011, selecting the Halfword data and accesses the second Halfword of the input data, via address 2. The output data correctly presents the duplicated second Halfword in the 16 LSBs of the output DOUBLEWORD, ascertaining the correct operation of the Byte Selector.

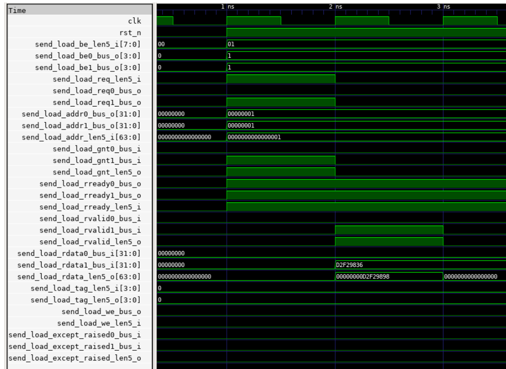
We then proceed to simulate the 64-bit transactions. These differ from the simulations analyzed previously; in fact, the requests can be executed on a multiport bus, thus being able to receive both responses at the same time, or on a single port bus, as in the case of X-HEEP, separating the reception of the halfwords. Both cases were simulated, along with the reception of an exception related to one of the two partial data.

Figure 4.4c shows the transaction on multiport bus. Unlike the previous transactions, in this one we note the assertion of the signals `bus_req` and `bus_gnt` on both ports, along with the addresses coming out of the *Address Splitter*, one of which is identical to the address given by the testbench, while the second is correctly added by a value of 4, for access to the next WORD in memory. In the second stage of the protocol, signals `bus_rvalid` and partial data are also provided on both ports, simultaneously, generating the complete DOUBLEWORD on the output.

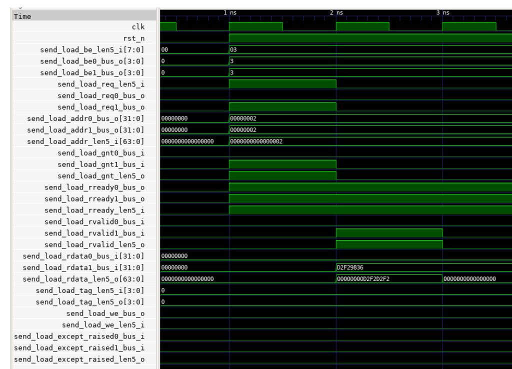
The X-HEEP bus is a single-port bus, so it will process requests separately, one at a time. A transaction on a bus of the same type is shown in figure 4.4b. The generation of the `request` and `grant` signals differs from the previous cases; the testbench asserts the `len5_req` which is correctly forwarded to both module ports, the `bus_gnt0` signal being

the first asserted, so that only the request coming out of port 0 is accepted. This causes the Grant CU to proceed in the WAIT_GNT1 state, as found by keeping `bus_req1` high, after deassertion of `len5_req` by the testbench. In the next clock cycle, the testbench also asserts the `bus_gnt1`, which results in the generation of the `len5_gnt`, and the `bus_rvalid0`. Here it is possible to see the operation of the *Data Register* present within the *Load Module*. The testbench initially loads a corrected partial data on `bus_rdata0` and, at the next cycle, loads a spurious data on the same bus along with the correct partial data on `bus_rdata1`. At the same time it asserts the `bus_rvalid1`, triggering the assertion of the `len5_rvalid` signal. The output bus `len5_rdata` still contains the 64-bit data composed of the two correct partial data, demonstrating the correct operation of the module.

Last, the operation of the logic responsible for the propagation of exceptions, received during the reception of a data, was verified. Visible in figure 4.4d, a second DOUBLE-WORD transaction was performed, with the assertion of the signal `len5_except_raised0` during the reception of the first partial data. As visible, the *exception* is propagated immediately to the processor, keeping it asserted even during the reception of the second partial data, at which time the processor receives the `len5_rvalid` signal and samples the complete data, along with the *exception* signal.

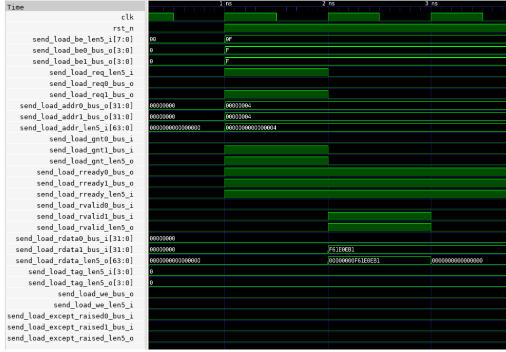


(a) Load BYTE Execution

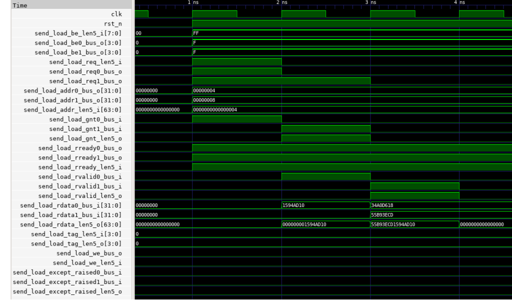


(b) Load HALFWORD Execution

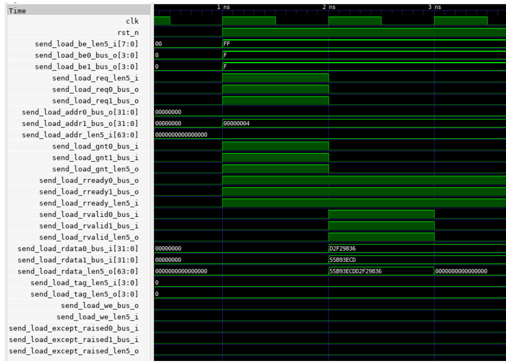
Figure 4.3: Load BYTE and Load HALFWORD Execution



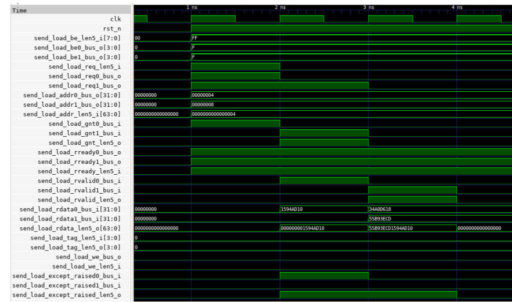
(a) Load WORD Execution



(b) Load DOUBLEWORD Execution with a Single Channel Response at a Time



(c) Load DOUBLEWORD Execution with Concurrently Channels Response



(d) Load DOUBLEWORD Execution with a Single Channel Response at a Time and an Exception Raised

Figure 4.4: Load WORD and Load DOUBLEWORD Execution

4.1.3 STORE Module Simulation

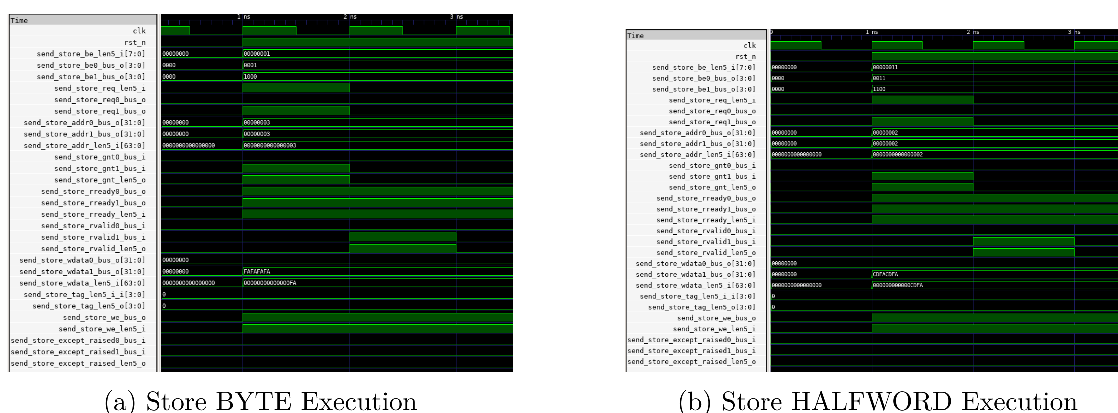
Last, the section on testing the *STORE Module* includes 5 tests, four necessary to cover all transaction case histories, both 32-bit and 64-bit, the last one additional, for 64-bit transactions executed on a multichannel bus.

Transactions for 32-bit data are structurally identical to instruction requests, with the only difference given the presence of two output ports instead of a single one, resulting in duplicate signals. Instead, it is useful to look at the data output to the bus. The *Data Aligner* present in the *Store Module*, correctly replicates the BYTE, in the case of Store Byte in the figure 4.5a, or the HALFWORD, in the case of Store Halfword in the figure 4.5b, saturating the Word provided at the output to the X-HEEP bus. At the same time, it generates the `bus_be1` depending on the type of request and the address provided by the processor. In the cases under consideration: 1000 for the Store Byte request, the address being 3, 1100 for the Store Halfword request, the address being 2, allowing only a portion of the Word in memory to be updated. The WORD transaction, in figure 4.6a, is of less interest, having the *Data Aligner* a marginal role on the generation of the data output and

the `bus_be1`.

Finally, for the DOUBLEWORD transaction, the request on a single port bus is simulated, visible in figure 4.6b. Since it is identical to a LOAD DOUBLEWORD request, as far as the generation of the signals `bus_req, len5_gnt` and `len5_rvalid`, we will not dwell on the analysis again. The only difference is that the data is immediately available on the outputs and arranged to respect the Little Endian disposition in memory.

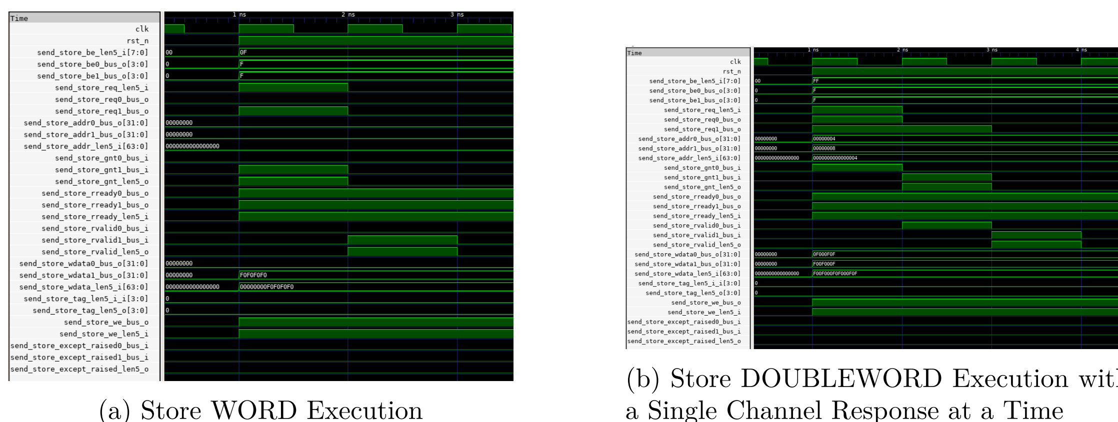
In all STORE transactions reported here, the update of the data in memory cannot be seen. The correctness of the same is checked by appropriate assert, which verifies the equality of the input data on `len5_wdata`, with the contents of memory at the locations indicated by the given address.



(a) Store BYTE Execution

(b) Store HALFWORD Execution

Figure 4.5: Store BYTE and Store HALFWORD Execution



(a) Store WORD Execution

(b) Store DOUBLEWORD Execution with a Single Channel Response at a Time

Figure 4.6: Store WORD and Store DOUBLEWORD Execution

4.2 Synthesis

Architecture synthesis was performed using TSMC's 65-nm libraries: *tcbn65lp* and *tcbn65lplvt*. Both libraries feature low-power components with different threshold voltages. In particular, the tcbn65lplvt library is of the *low voltage threshold* type, presenting low threshold voltage components within it, with higher speeds but higher leakage. Both libraries were provided to the Synopsys Design Compiler synthesizer, to allow the selection of components best suited to the design constraints and the various critical paths present in the layout. Each library has several different versions, taking into account the possible operating conditions of the components present. By operating conditions we mean that set of factors, such as *Process Corners*, *Temperature*, *Voltage Threshold* and *Supply Voltage*, capable of profoundly affecting the results of the synthesis. A synthesis will first be performed in the *Worst-case Condition*, setting the worst operating conditions. This is a mandatory constraint to obtain the minimum clock period of the system and ensure its operation in every usage scenario. Next, to better model the standard usage scenarios, two additional syntheses will be performed in the *Typical-case Condition* and *Best-case Condition*, comparing the results.

Given the amount of commands required for the synthesis, several scripts were produced that can automate the synthesis and make the results replicable. The configuration parameters used are listed below:

- Setting the Clock distribution tree as non-optimizable, via the command `set_dont_touch_network`, leaving this task to the place and routing tool. In fact, this step is critical, as the clock distribution must be fair throughout the device to avoid skew and consequently failure of certain components.
- Setting a `skew` time for the clock signal equal to 0.07 ns, using the `set_clock_uncertainty` command. Clock generators are not perfect elements; rather, they have nonidealities that cause variations in the duration of the clock period. By setting this parameter, any dissimilarities are taken into account, improving the accuracy in calculating the minimum clock period of the module.
- Setting the input and output signal delay equal to 0.5 ns, using the commands `set_input_delay` and `set_output_delay`. The library used already defines for each memory element within it, a standard setup time. An additional delay is added only on input and output of the module, to model any combinatorial delays external to the module under analysis.
- Setting a load capacity on all nodes in the generated netlist, via `set_load` command. The bridge is an interface module between LEN5 and X-HEEP. Having performed the synthesis of the module only, due to the excessive complexity of the complete architecture, it is impossible to estimate the load of each node and consequently the

fan-out of each gate. However, to make the synthesis meaningful, the input capacity of a buffer (BUFFD4) with a drive strength of 4, equal to 2 fF, was attached to each node. Its complete representation of all parameters can be found in [13].

- Synthesis performed via the directive `compile_ultra`, forcing the synthesizer to use the constraints of Ultra High Effort, minimizing the area of the module and maximizing its clock frequency.

4.2.1 Bridge

The initial stage of the synthesis process sees the processing of the various components of the bridge, using the `elaborate` command. The output was carefully analyzed for any `latch inferred` or other errors. Having ascertained that the processing was correct, the actual synthesis of the design was carried out using the command described above.

In order to estimate the maximum clock frequency at which the module is capable of operating, a clock signal was generated via the command `create_clock` with period equal to 0, forcing the synthesizer to perform all possible optimizations to allow the signals to propagate in a null period. It is, of course, impossible to obtain a positive result; what is of interest, however, is the `slack` time obtained after the analysis is completed. The `slack` time means the difference between the set clock period and the actual propagation time of the signals within the critical path. As given in Appendix A.11 of [11], the minimum clock period can be calculated as:

$$T_{clk} = T_{c \to q} + T_{comb} + T_{skew} + T_{sp}$$

where $T_{c \to q}$ is the propagation time within a memory element, T_{comb} is the propagation time within a combinational logic, and T_{sp} is the setup time. By forcing T_{clk} equal to 0, the resulting T_{slack} will be equal to the time it takes for the signals to propagate within the critical path of the module, thus obtaining the module's minimum clock period.

In practice, the results obtained do not exactly match the T_{slack} provided by the above analysis, as the synthesizer uses overly stringent optimizations trying to force a null period. Several syntheses were subsequently performed, calibrating the clock period, until `slack met` was obtained, i.e., the coincidence of the period set with the propagation time in the critical path.

Once the synthesis is completed, two types of reports are extracted, the `timing report` and the `area report`. The first illustrates the critical path present in the module, complete with all the elements the signal passes through, and the time it takes to propagate within it. The second describes the area occupancy of the module divided into `Combinational Area`, `Noncombinational Area (NC Area)` and `Buff/Inv Area`. As the names imply, the first is the area occupied by combinational logic, the second is the area occupied by

sequential components, such as registers, and the last is the portion of the area occupied by buffers/inverters.

In the timing analysis, a critical path was found to have a start point, the `len5_addr` bus, and an end point, the `bus_addr1` bus. This is not surprising at all, it has already been introduced in the section on *Address Splitter* how the generation of the address on port 1 related to the access case Misaligned, needs the instantiation of an adder. It is precisely the presence of this adder that generates the critical path, otherwise absent given the additional methods by which the *Address Splitter* generates addresses in the cases different to the Misaligned access. The cycle time and maximum clock frequency for the Bridge can be seen in table 4.1.

As for the area analysis, the values obtained are visible in table 4.2. Note how the area of the entire Bridge is occupied for 2/3 by combinational logic; the presence of 5 Control Units, containing the state generation and control logics, together with the data alignment modules, the *Address Splitter* and the counters internal to the FIFO queues, is preponderant over the state and data registers. Wanting to go into more detail, the three main modules of the bridge were summarized separately in order to assess their impact in terms of timing and area on the whole structure.

4.2.2 Instruction Module

The *Instruction Module* appears to be the least limiting from the point of view of timing and area. As can be seen in table 4.1, the cycle time of the module is the only one that is less than the others, given the absence of the address splitter present on both other modules instead. The critical path encountered in this case would be internal to the FIFO tag, terminating with the output tag to the processor. However, as already explained in the previous paragraphs, the FIFO tag is not used by LEN5, so it can be disabled or completely removed to gain area as well. Therefore, it was decided to discard the critical paths internal to the FIFOs, selecting the first critical path that included different signals. The new critical path turns out to be internal to the *Instruction CU*, with start point the status register and end point in the *Instruction Register*. This is most likely the path related to the generation of the `buff_en` signal, which is used to enable the register in case the processor is not available to receive the instruction.

The area of the *Instruction Module*, visible in table 4.2, has a completely opposite trend to that of the Bridge analyzed earlier, presenting a lower `Combinational Area` than the `Non-combinational Area`. This result was, however, to be expected; in fact, the module presents a Control Unit with simple logics, having only two possible states and few control signals; it also does not present alignment modules or the like. In contrast, the *Instruction Register*, the TAG FIFO and the CU status register are present, although of

little importance being only 1 bit.

4.2.3 LOAD Module and STORE Module

The results of the LOAD Module and STORE Module synthesis, given the above considerations, are easily deduced. By both presenting the *Address Splitter* component, the cycle time found, present in table 4.1, coincides with that presented in the Bridge section, thus making the *Address Splitter*, the limiting component in the Bridge performance.

Analysis of the LOAD Module area, visible in table 4.2, shows that the `Combinational Area` is in this case predominant over the `Non-combinational Area`. This can be explained given the presence of two Control Units, the Byte Selector and the Address Splitter, along with several multiplexers for addressing data. The registers present are less incidental than the combinational logic, although they still represent the largest contribution to the total `Non-combinational Area`. Similarly, the analysis of the STORE Module shows the same trend as the LOAD Module. It is important to note, however, the reduced `Non-combinational Area`, due to the presence of a single FIFO queue and only 2 state registers.

Module	**Cycle Time** [ns]	**Max Frequency** [MHz]
Bridge	1.41	709.2
Load Module	1.41	709.2
Store Module	1.41	709.2
Instruction Module	1.36	735.3

Table 4.1: Cycle Time and Max Clock Frequency for the Bridge and the three Principal Modules

Module	**Total Area** [μm^2]	**Comb. Area** [μm^2]	**NC Area** [μm^2]
Bridge	4770.72	3012.84	1757.88
Load Module	2461.68	1642.68	819
Store Module	1520.28	1163.16	357.12
Instruction Module	788.76	207	581.76

Table 4.2: Total Area, Combinational Area and Non-combinational Area for the Bridge and the three Principal Modules

4.2.4 Case Condition Analysis

To complete the Bridge analysis, two additional syntheses were performed, using the *typical-case* and *best-case* versions of the libraries provided to the synthesizer. The `Case`

`Conditions Analysis` allows the synthesis parameters of the Bridge to be verified even in the typical and best, as well as worst, varying cases of operation:

1. **Process Corners**, characterizing the deviation of MOS device parameters within the library, such as diffusion doping levels, gate oxide thickness, etc., from nominal values.
2. **Temperature**, capable of affecting leakage currents within the devices.
3. **Threshold Voltage**, affects the operating current of the devices and the switching speed.
4. **Supply Voltage**, controls the speed of signal propagation within the architecture.

What is expected from the variation of the devices used is mainly the reduction of the Cycle Time of the propagating signals within the critical path. As introduced at the beginning of this chapter, the use of the library in the *Worst-case Condition* is a mandatory constraint for calculating the minimum clock period. Nevertheless, in typical usage scenarios, estimating the maximum working frequency of the module with the *Worst-case Condition* is overly stringent, leading to a reductive evaluation of device performance. For this reason, analyses are also introduced in the *Typical-case Condition*, which models the standard use case, and in the *Best-case Condition*.

The results obtained, which can be seen in table 4.3, regarding the timing analysis, are as projected. By using devices with better technological parameters and favorable environmental conditions, it is possible to perceptibly reduce the Cycle Time and consequently increase the maximum clock frequency. Indeed, a reduction of 10% is obtained in the minimum clock period, between *Worst-case Condition* and *Typical-case Condition*, proceeding to 14% between *Worst-case Condition* and *Best-case Condition*.

The analysis of occupied area shows a downward trend. The decrease is most likely due to the choices of generic components during the processing phase. By using increasingly favorable conditions and higher-performance technologies, the synthesizer no longer needs complex implementations of the generic blocks in order to meet timing constraints; instead, basic implementations, which save area, are sufficient.

Bridge	**Cycle Time** [ns]	**Max Frequency** [MHz]	**Area** [μm^2]
Worst-case Condition	1.41	708.2	4770.72
Typical-case Condition	1.28	781.3	4504.32
Best-case Condition	1.22	819.7	4273.56

Table 4.3: Cycle Time, Max Clock Frequency and Area of the Bridge, as a Function of Case Conditions

Chapter 5

Integration of the LEN5 processor within X-HEEP

Having finished the examination of the Bridge as a stand-alone module, this chapter will look at the integration of the LEN5 processor within the X-HEEP platform through the use of the developed Bridge.

Before proceeding to the key topic of this chapter, it is useful to introduce the compilation and simulation process of the RTL and Software used by X-HEEP.

The platform is based on a fully automated mechanism, managed through a series of makefiles, which first set a series of environment variables needed by the simulator and the compiler (GCC), then a python script is called (`mcu_gen.py`) which, starting from templates[1] files, generates the RTL of the Microcontroller Unit (MCU), comprising all the components within which processors, peripherals, etc. are instantiated.

Having generated the RTL, we proceed with the architecture build by exploiting **Fusesoc**.

Fusesoc can be defined as a package manager and build system for digital hardware [14]. An extremely modular tool, supporting a variety of simulators, such as Verilator or Questasim, and synthesizers such as Design Compiler. It uses a single file, called `.core` file, where all the directives useful for compiling, simulating and synthesizing an entire architecture are defined, and then used in the generation of an output file, suitable for the tool in use, which will eventually be executed by the tool in question.

Once the architecture build is completed, the compilation of the software to be run on

[1]Files inside of which there is part of the essential Systemverilog code along with variables, in which the script `mcu_gen.py` saves the configurations described in the files `mcu_cfg.hjson` and `pad_cgf.hjson`, being able to dynamically generate the `.sv`

the platform is performed, through the use of CMake, to automate the compilation process and provide the correct parameters to the compiler **gcc**

Finally, the Verilator tool is used, to simulate the design and produce the `.vcd` files, which can be viewed on any *Waveform Viewer*.

LEN5 already includes within it the `.core` files necessary to ensure compatibility with Fusesoc, so two additional core files, `bridge.core` and `bridge_top.core`, were produced and added to the existing ones. The first one presents inside the declaration of all System Verilog files made for the Bridge together with the dependency on the core files of the LEN5 Packages. In the second we find the declaration of the `bridge_top.core` file and the dependencies on the previous core file and that of the LEN5 Packages.

5.1 LEN5 Instantiation

Before LEN5 was instantiated within X-HEEP, it was interconnected with the Bridge, through the creation of a Wrapper and its `.core` file. The interface of the Wrapper coincides with the interface of the Bridge, which will then be connected with the X-HEEP bus.

We have already introduced in section 2.3, the limitation due to the single 32-bit port bus present inside X-HEEP. Even supporting multiple LOAD and STORE requests in parallel, via the four output ports, the presence of a single port for the bus limits the Bridge to performing a single transaction at a time. For this reason, the Bridge's connection with the X-HEEP bus is made by instantiating a 4-to-1 Crossbar (XBAR), sequencing the outgoing requests from the Bridge's four ports and sending them to the single data port on the bus.

The crossbar used is a module designed to connect an N number of Masters to a single Slave. This allows all connected Masters to send simultaneous requests to the Slave; the crossbar takes care of collecting and forwarding them sequentially, ensuring that each Master correctly receives the `grant` and `rvalid`, along with the requested data.

Once the various modules were assembled, the wrapper and crossbar were instantiated within the X-HEEP `cpu_subsystem`, adding the ability to select LEN5 as the platform processor in the corresponding `.sv` file. In addition, LEN5 was added as an option of the `-cpu` parameter of the `mcu_gen.py` script, allowing its selection directly by setting the `mcu_cfg.hjson` file.

Finally, the `.core` file of the Wrapper was added within that of X-HEEP.

Figure 5.1 shows the schematic of the interconnections between the various components described above.

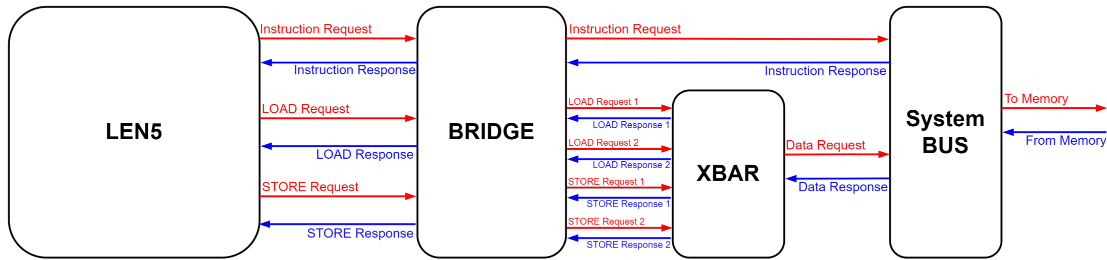


Figure 5.1: LEN5 and Bridge Connection to XBAR and System BUS

5.2 X-HEEP Modifications to Ensure Compatibility with LEN5

With the instantiation completed, the various files needed for X-HEEP to compile and build RTL and Software were adapted. Initially, the primary makefile of the X-HEEP platform was modified to include among the simulatable architectures, riscv64, adding it to the already supported riscv32. To do this, an environment variable was added, called `ARCH_TARGET`. Through this variable, the makefile can correctly set the compiler (gcc) to be used, specifying the contents of the variable itself as part of the compiler prefix present in the `COMPILER_PREFIX` variable. In the case under consideration, 64-bit compilation of the source code is required, LEN5 being a processor supporting an rv64 ISA, so it is necessary to set the variable `ARCH_TARGET`=riscv64. LEN5 does not yet support all ISA rv64 extensions, so it was necessary to specify to the compiler which extensions to use, via combination `-march` - `-mabi`[2]. Not all architecture-abi combinations are in fact supported, in the case under consideration an ISA `rv64im_zicsr`, supporting standard instructions along with multiplications/divisions and access to CSR, coupled with an ABI `lp64` is used. To do this, the file `CMakeLists.txt` was modified so that, based on the contents of the variable `ARCH_TARGET`, it sets the variable `COMPILER_LINKER_FLAGS` with the appropriate options to be specified to the compiler.

After the source files are compiled, the linking phase of the libraries, present inside X-HEEP, to the object file produced by the compiler, takes over. X-HEEP has within it a linker of reduced complexity compared to the standard gcc linker. The linker is responsible for resolving references to functions declared inside the object file, assigning definitive addresses to the variables in the data section of memory as well as adjusting the arrangement of program sections, containing instructions or data, inside the ram, including the

[2] It can be understood as a low-level version of an API, in which the methods of using functions and data structures are defined

crt0 and the interrupt vector table. To produce a LEN5-compatible `.elf` file, a second linker file was made, identical to the one already within X-HEEP, with the parameter `OUTPUT_FORMAT` modified, replacing `elf32littleriscv` with `elf64littleriscv`. In addition, the file `CMakeLists.txt` was again modified so that it selects the correct linker, depending on the contents of the variable `ARCH_TARGET`. The ELF file, Executable and Linking Format, is a type of executable file, containing, in addition to instructions and data, several Headers with inside the information needed to execute the compiled and linked file, once saved to disk. Each of these Headers provides information about different parts of the file itself; the ELF Header, for example, contains the type of ELF file, the ISA of the processor that will execute it, and the type of arrangement of the data in memory. The Program Header, on the other hand, provides useful information about the (`segments`), unions of various sections of the program, and their execution, as visible in [16]. For compatibility with a 64-bit architecture, it is mandatory to use the `elf64` format, which, among other things, sets the length of addresses equal to 64 bits. The suffixes `little` and `riscv` establish the convention of the arrangement of the data in memory (Little Endian) and the architecture on which the file will run. We will not dwell further on the ELF format, for further discussion refer to [15] together with chapter 4 and 5 of the update found in [16].

Firmware execution starts from a boot rom. At startup, the CPU is jumped to the start address of the boot rom, where the firmware execution mode is selected in addition to system initialization. X-HEEP supports 3 execution modes, execution via JTAG, execution via FLASH SPI, and firmware loading via FLASH SPI directly into the ram. The above is reported in [7]. For the purpose of this thesis, execution via JTAG is sufficient, of which the preloading of firmware in memory via testbench is exploited to speed up the simulation. In order to make the boot rom compatible with LEN5, the makefile with which the rom is generated was modified by selecting via the contents of the environment variable `RISCV`, the 32-bit or 64-bit compiler to compile the rom firmware.

5.3 Simulation and Benchmark

To conclude the inclusion of the LEN5 processor within X-HEEP, the output of the application `Hello World!`, already present within the platform, was simulated. The configuration chosen for the use of the processor with X-HEEP provides a trade-off between performance and power consumption, the bus is in fact instantiated in NtoM mode, so as to maximize the available bandwidth, the memories are instead instantiated in contiguous addressing mode so as to take advantage of power gating. The simulation waves will be omitted due to space constraints, being the execution of the entire boot rom combined with the firmware, extremely long. Through direct observation, it was ascertained that virtually every type

of LOAD and STORE request possible was executed correctly. At the end of execution, LEN5 was able to correctly execute the entire firmware and print the string `hello world!` on the screen.

The result can be seen in the screenshot in the figure 5.2.

To evaluate the performance of the processor connected through the Bridge and the overhead of the Bridge itself, four simple types of tests were performed: string printing, sum, multiplication, and division. The number of execution cycles found within the CSR MCycle was then used to estimate the time required to execute each test. The CSR is accessed by editing the `main.c` of the `Hello World!` application, adding the command pair `asm volatile ("csrr %0, 0xb00" : "=r"(start))` and `asm volatile ("csrr %0, 0xb00" : "=r"(stop))`. Note that the execution of the two additional commands will partially perturb the results of the performance analysis of the single application `Hello World!`, it turns out, however, to be acceptable since the code executed is the same for all CPUs. On the other hand, there are no incompatibility issues; since the CSR `MCycle` is standard, all cpu's supported by the platform have it and can access it, allowing the performance evaluation to be done equally. The configuration settings of X-HEEP, for each cpu tested, are identical to those used for LEN5. The results can be seen in table 5.1 and figure 5.3.

```
[TESTBENCH]: No OpenOCD is used
[TESTBENCH]: loading firmware  ../../../sw/build/main.hex
[TESTBENCH]: Max Times is  200000
[TESTBENCH]: No Boot Option specified, using jtag (boot_sel=0)
                   0: the parameter COREV_PULP is 00000000
                   0: the parameter FPU is 00000000
                   0: the parameter ZFINX is 00000000
                   0: the parameter X_EXT is 00000000
                   0: the parameter ZFINX is 00000000
                   0: the parameter JTAG_DPI is 00000000
                   0: the parameter USE_EXTERNAL_DEVICE_EXAMPLE is 00000001
                   0: the parameter CLK_FREQUENCY is     100000 KHz
[X-HEEP]: NUM_BYTES =         64KB

UART: Created /dev/pts/11 for uart0. Connect to it with any terminal program, e.g.
$ screen /dev/pts/11
UART: Additionally writing all UART output to 'uart0.log'.
Reset Released
Set Exit Loop
Memory Loaded
Program Finished with value 0
hello world!
```

Figure 5.2: Output of the Hello World! Application

The benchmark results show that the bridge overhead is completely negligible against an average improvement of 7% in execution speed over the cv32e20 and 5.5% over the cv32e40

series. This is due to both Out-of-Order execution and the presence of dynamic branch prediction, as opposed to the static prediction found in other processors, which ensures accurate prediction during memset and function execution phases, where a large amount of branches are found. Keep in mind that the code executed by LEN5 was compiled with an ISA rv64, so within it there are also LOAD/STORE instructions of type DOUBLEWORD, requiring two consecutive memory accesses via the Bridge. While performing multiple memory accesses, LEN5 remains the fastest processor of all those supported by X-HEEP, resulting in the most appropriate choice for executing complex codes, where the Out-of-Order architecture makes the difference.

Cycle Time	LEN5	cv32e20	cv32e40x	cv32e40p	cv32e40px
printf("Hello World!")	52684	56690	55728	55731	55731
Sum + printf(...)	50008	54051	52915	52918	52918
Mul + printf(...)	58032	62113	60937	60940	60940
Div + printf(...)	50004	54051	53278	53313	53313

Table 5.1: Comparison between X-HEEP Supported CPU Cycle Time

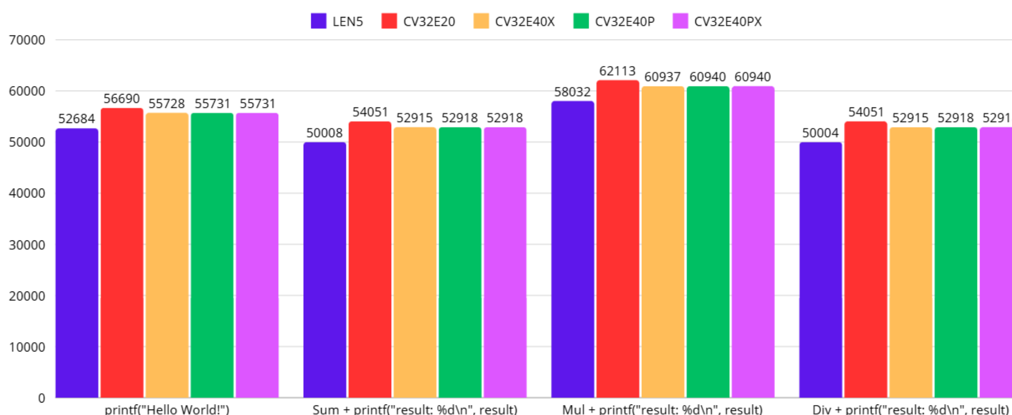


Figure 5.3: Comparison in terms of execution cycles among processors supported by the X-HEEP platform. The tests performed were: printing a string, sum, multiplication, and division with relative printing of the result for each operation.

Part II

Development of the Debug System within the LEN5 processor

Chapter 6

Debug System Design

The second part of this thesis aims to develop the internal debugging system for the LEN5 processor. It will initially introduce the JTAG standard and the implementation of the TAP module, which is necessary for the processor to support the standard. It will conclude by analyzing in detail the changes made on the architecture of LEN5, in order to take advantage of the Debug Module present within X-HEEP. The ultimate goal is to obtain a control and testing mechanism, exploitable during the development of the processor and possibly during the execution of software exploiting LEN5 as a core, that can provide information on the operation of the hardware.

6.1 JTAG

The JTAG standard, created by a consortium of leading companies also called the Joint Test Action Group (JTAG), hence the name of the standard, and known nowadays as the IEEE 1149.1 standard, was created to address a fundamental problem that has become widespread with the evolution of technology and the move to packages in the BGA form factor. In fact, the package structure, coupled with the increased complexity of the chip itself, makes it difficult to perform functional tests or physical tests, such as bed-of-nails [18]. The proposed solution to the above problem is the introduction within the architecture of various ad hoc registers, including Boundary Scan Registers (BSR), composed of single cells arranged on the device pins, and a control module, called Test Access Port Controller, responsible for controlling the BSCs and registers. Boundary scan cells (BSCs) have two modes of operation: they are transparent (functional mode) during normal operation of the component, while being able to isolate it (test mode) during the test phase. Once the component is isolated, it is possible to either read the values present on the pins during the test or force specific values on them to evaluate their behavior [17]. This makes it

possible to perform functional tests on the device, called Boundary Scan, either electrical, evaluating the correct connection of the various pins, or logical, evaluating the validity of the component.

Added to what has just been introduced is the possibility of using the JTAG interface, also called Test Access Port (TAP), as the debugging interface of the module itself. The Test Access Port Controller module and the registers needed for the JTAG protocol were then implemented, complying with the IEEE 1149.1 protocol specification [19]. The structure of the module can be seen in figure 6.1.

6.1.1 TAP Controller and Registers

The JTAG interface has five mandatory signals to be specified, described below:

- **TCK** → Clock signal of the TAP module, BSCs and additional registers.
- **TRST_NI** → Reset signal, active denied.
- **TMS** → Control signal needed by the FSM within the TAP module to evolve state.
- **TDI** → Sequential data input to the TAP module.
- **TDO** → Sequential data output from the TAP module.

In addition, there are additional signals, which are necessary for monitoring, sending and receiving data from the BSR.

- **bsr_data_o** → Data sequentially sent to the BSR.
- **bsr_data_i** → Data sequentially received by the BSR.
- **bsr_shift_o** → Control signal required for BSR shift.
- **bsr_capture_o** → Control signal needed for loading the values present on the device pins, into the BSR.
- **bsr_update_o** → Control signal necessary for loading the values contained in the BSR, onto the device pins.
- **bsr_enable_o** → Enable of the BSR.

The TAP consists of a TAP Controller, an Instruction Register, and two Data Registers, named IDCode Register and Bypass Register, respectively.

The instruction register is actually composed of two registers, a standard register and a shift register. In the shift register, the instruction necessary for the TAP to select the data register to be connected to the output of the module itself is entered by a shift operation. Once the shift register is filled, the instruction is saved in the standard register, the value of which selects which Data Register to connect to the multiplexer that handles the output of the TAP module and enables the Data Register corresponding to the instruction within it.

The IDCode register is itself a dual register, consisting of a shift register and a standard register, containing the JTAG-compatible device identification value. The shift register allows the device ID code to be loaded and shifted onto the signal `td_o`.

The Bypass register is a single Flip-Flop, used to connect input and output with as little delay as possible. In fact, modules supporting the JTAG interface can be connected in cascade, and by taking advantage of the bypass register, it is possible to communicate with a specific module by passing through those preceding it, with minimal delay.

Finally, the TAP controller is based on an FSM required to manage the multiplexer that controls module output and register control signals.

In order to be JTAG-compliant, the module must necessarily support three instructions, BYPASS, EXTEST, and SAMPLE/PRELOAD, which condition its operation, as visible in [19]. In the case of the module implemented in this thesis, support for the previous three plus an additional one has been included. The supported instructions are listed below[1]:

- **BYPASS** → signals `td_i` and `td_o` are connected through the Bypass register. The instruction allows bypassing the module in question, testing other modules connected in cascade, with minimal overhead.
- **EXTEST** → signals `td_i` and `td_o` are connected to the BSR. The instruction allows the signals present on the device pins at a given instant to be saved by the signal `capture_dr`, shift other values to the BSR by `shift_dr`, and load values to the device pins by `update_dr`.
- **SAMPLE/PRELOAD** → the signals `td_i` and `td_o` are connected to the BSR. This instruction maintains the normal operation of the device, allowing the values of the signals on the pins to be saved in real time, via the signal `capture_dr` or new values to be loaded on the device pins via the signal `update_dr`. It also allows the BSR to be preloaded for the EXTEST instruction, via `shift_dr`.
- **IDCODE** → signals `td_i` and `td_o` are connected to the IDCode register. The instruction allows the device ID code to be displayed by first loading it via the `capture_dr` signal and then shifting the contents of the register via the `shift_dr` signal.

[1]The instruction descriptions, and in general the descriptions of the specifications and protocol components given in this document, are those found in [19]. Refer to it for more details

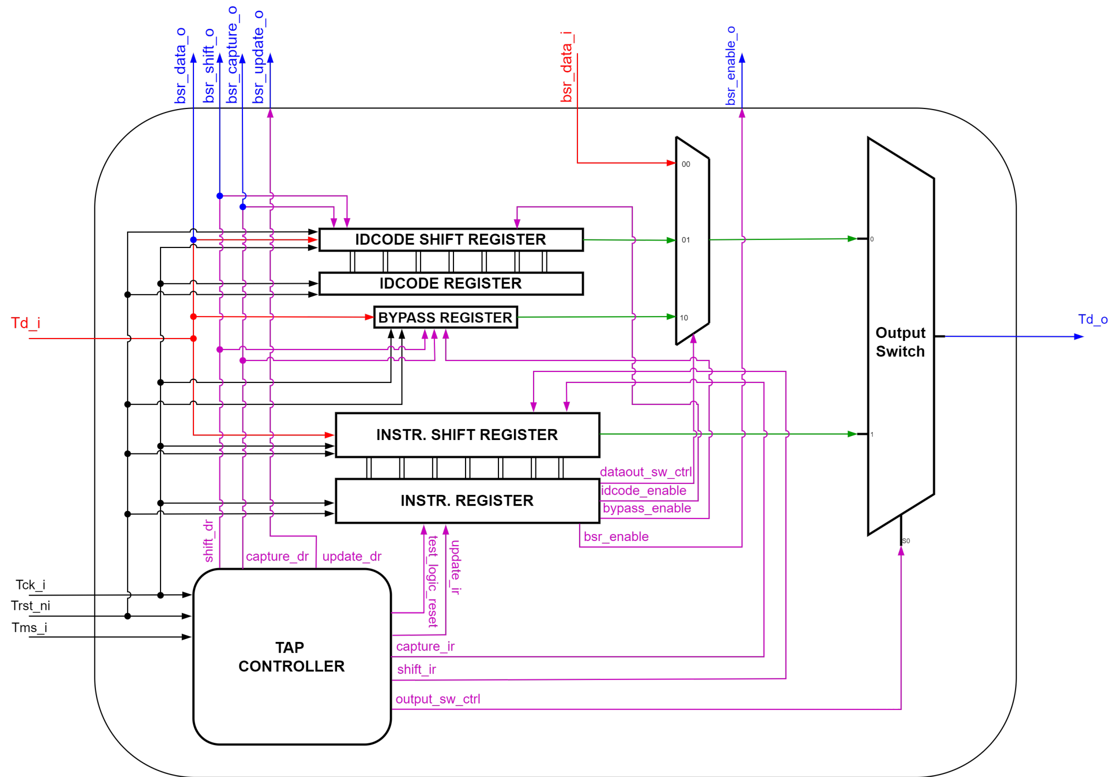


Figure 6.1: TAP Module

The implementation of the internal components of the TAP Module is described next.

TAP Controller

The TAP Controller is responsible for generating the register control signals and selecting the correct output of the module, via a multiplexer placed between the outputs of the Instruction Register and Data Register, and the output of the TAP module. The implementation is based on a 16-state FSM, described in the protocol specification, controlled by the `TMS` signal value only. Internally there are eight signals generated, we find:

1. The signal `test_logic_reset`, necessary to reset the Instruction Register by loading it with the IDCODE instruction.
2. The three control signals of the Instruction Register, `capture_ir, shift_ir` and `update_ir`, necessary correspondingly for the parallel loading on the shift register in question of a fixed value with the last 2 LSBs forced to 01, the shift of the value contained in the shift register to `td_o` and the parallel loading of the contents of the shift register within the standard register.
3. The three control signals of the Data Registers, `capture_dr, shift_dr` and `update_dr`,

whose functions vary according to the Data Register selected. In the case of the ID-Code Register, `capture_dr` enables the parallel loading within the shift register of the IDCode value present in the standard register, while `shift_dr` performs the shift of the shift register to the output `td_o`. The signal `update_dr` is not used. In the case of the Bypass Register, the signal `capture_dr` resets the Flip-Flop with value 0, the signal `shift_dr` loads the value present on `td_i` into the register, making it available on `td_o`. The signal `update_dr` is not used in this register either. Finally in the BSR, the signal `capture_dr` allows the parallel loading into the BSR of the values present on the outputs of the device, the signal `shift_dr` allows its shift to `td_o`, the signal `update_dr` allows the parallel loading of the value present in the BSR, on the inputs of the device.

4. The signal `output_sw_ctrl`, used to select the output of the TAP module, depending on the branch of the FSM being executed and consequently on the register to be connected to the output, Instruction or Data.

The possible states and their operation, visible in the state diagram in figure 6.2, are illustrated below. The control signals are set, as the default value, to the value 0, except for the signal `output_sw_ctrl`, which is set to the value 1. Should the value change, it will be specified in the corresponding state; otherwise, the value will be equal to the default value.

- **Test_Logic_Reset** → State in which test logic is disabled and the device can continue normal execution. The signal `test_logic_reset` is asserted, resetting the Instruction Register and leaving the device unaffected.
- **Run_Test_Idle** → In the current implementation, this state is empty. In fact, no operation is performed while the FSM is inside it. Its presence is exploited in case RUNBIST instruction is supported, which, in this state, performs a self-test on the device.
- **Select_Ir_Scan** → Temporary state in which it can be decided to continue in the instruction branch and use the Instruction Register or return to the Test_Logic_Reset state. No operations are performed while in this state.
- **Capture_Ir** → During this state, the shift register, part of the Instruction Register, is loaded by asserting the signal `capture_Ir`.
- **Shift_Ir** → State used for shifting the value present in the Instruction Register, by the assertion of the signal `shift_ir`.
- **Exit_1_Ir** → First temporary state of the instruction branch, in which the decision can be made to suspend the shift of the data in the Instruction register, passing into the `Pause_Ir` state, or to continue execution by updating the standard register contained in the Instruction Register, via the `Update_Ir` state.

- **Pause_Ir** → In this state, the shift of the input instruction, inside the Instruction Register, is suspended.
- **Exit_2_Ir** → Second temporary state of the instruction branch. Here you can decide whether to resume the instruction shift, switching back to the `Shift_Ir` state, or end the instruction branch by updating the standard Instruction Register by switching to the `Update_Ir` state.
- **Update_Ir** → Final state of the instruction branch, allows the value contained in the shift register to be loaded into the standard Instruction Register by asserting the signal `update_Ir`.
- **Select_Dr_Scan** → Temporary state used to select whether to continue in the data branch and use the Data Registers or to switch to the Select_Ir_Scan state in which it will be possible to select the instruction branch and use the Instruction Register. No operation on the registers is performed, however, the `output_sw_ctrl` signal is deasserted, setting the TAP output to the Data Registers.
- **Capture_Dr** → State used for parallel loading of the selected Data Registers, if it supports the function, by asserting the signal `capture_dr`. The signal `output_sw_ctrl` is kept deasserted.
- **Shift_Dr** → Mirror state to that introduced for the instruction branch. Here the shift of the data register, selected by the instruction in the Instruction Register, occurs by assertion of the signal `shift_dr`. The signal `output_sw_ctrl` is kept deasserted.
- **Exit_1_Dr** → First temporary state of the data branch, here you can decide whether to suspend the data shift within the selected register, switching to the `Pause_Dr` state or complete the execution by loading on the parallel output of the selected register, the data present within it, switching to the `Update_Dr` state. Also in this state, the signal `output_sw_ctrl` is kept deasserted.
- **Pause_Dr** → Pause state in which the data shift in the shift register selected by the Instruction Register is suspended. The signal `output_sw_ctrl` is kept deasserted.
- **Exit_2_Dr** → Second temporary state of the data branch, in which you can decide whether to resume the data shift in the selected shift register, returning to the `Shift_Dr` state, or terminate execution by switching to the `Update_Dr` state. Also in this state the signal `output_sw_ctrl` is kept deasserted.
- **Update_Dr** → End state of the data branch, here it is possible to load the data present in the selected shift register, on its parallel output, by assertion of the signal `update_dr`. The signal `output_sw_ctrl` is kept deasserted. In this implementation, the only register supporting the update will definitely be the BSR, having to make its contents available to the device pins. The IDCode Register and the Bypass Register, on the other hand, do not support update, the former since it is not allowed to overwrite the value of the device's IDCode, this being constant. The second, being

used only to bypass the TAP itself, has no second register where its contents can be saved.

Again, as with instructions, descriptions of the various states along with the signals they generate are visible in [19].

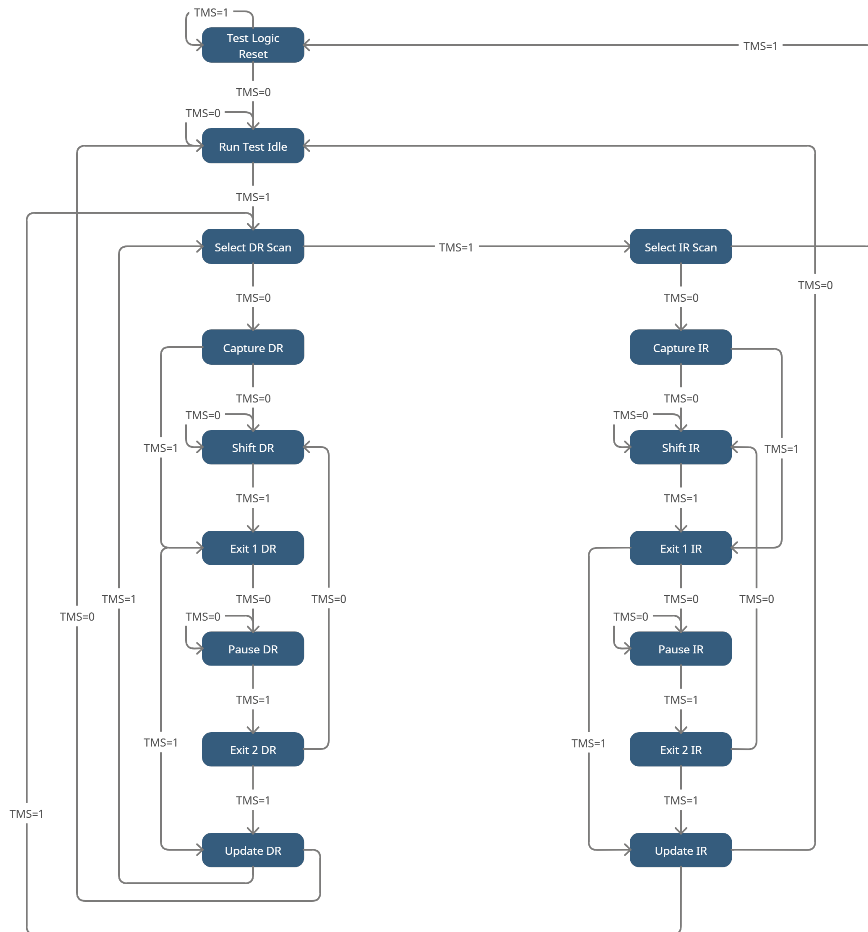


Figure 6.2: TAP Controller State Diagram

Instruction Register

The structure of the Instruction Register consists of two registers. The first is a normal register, the size of which is defined through a module configuration parameter, supporting the signals of: `clock`, as stated in the specification the register works on the falling edge of the clock, `reset`, and two additional signals, `test_logic_reset` and `update_ir`. The first signal is used in the test_reset phases of the module, in which the device to which it is connected is allowed to work under normal conditions. When asserted, it forces the loading

into the register of the IDCODE instruction. The second signal is actually the enable of the register itself, when asserted it enables the loading into the register of the data present on its input, coming from the second structure that makes up the Instruction Register, the shift register. In all other cases, the register holds the data within it constant.

The contents of the register are used as a selector for a multiplexer placed on the outputs of the Data Registers, selecting which of the registers to connect to the second multiplexer placed on the output of the TAP. It also serves as the generator of the `enable` signal of the Data Register corresponding to the instruction inside it. The supported instructions, as described earlier, are four: IDCODE, BYPASS, EXTEST, and SAMPLE/PRELOAD. In the case of the IDCODE instruction, the signal `idcode_enable`, enabling the IDCode Register, is asserted and the multiplexer is set on the register outputs by connecting the IDCode Register to the output. Similarly, for the BYPASS instruction, the signal `bypass_enable` is generated and the Bypass Register is connected to the module output. The last two instructions are related to the BSR, so the `bsr_enable` will be generated and connected the latter to the output.

The shift register is a more complex register than the previous one, supporting both parallel and serial inputs and outputs. The serial input is directly connected to the input of the TAP module, `td_i`, so that instructions are loaded into the register serially; the parallel input, on the other hand, is connected to a fixed value, presenting the last 2 LSBs fixed at 01, as given in the specification [19]. The serial output is connected directly to the mux present on the module output, controlled by the TAP controller, while the parallel output is connected to the standard register input. Its size is managed by a configuration parameter of the TAP module, in the same way as the previous register. In addition to the `clock` and `reset` signals, the register supports a `enable`, which is constantly connected to 1 in the case of the Instruction Register but its presence is important since the shift register module is also reused for one of the Data Registers. In addition, there are two other signals, `capture` and `shift`, connected to the signals `capture_ir` and `shift_ir`, respectively, which are needed to parallel load the register, in the `Capture_Ir` state of the controller, and to shift the contents of the register to the serial output, in the `Shift_Ir` state of the controller. In all other cases, the register holds the value within it, constant.

Data Registers

The Data Registers section consists of two registers, IDCode Register and Bypass Register. The former is itself composed of a normal register combined with a shift register, as in the case of the Instruction register, while the latter is a single Flip-Flop.

The standard register within the IDCode register supports only the signals `clock` and `reset`, storing within it the IDCode value of the device, in the reset phase of the module,

and then keeping it constant. In fact, the IDCode value cannot be changed and is provided as an external parameter of the TAP module.

The internal implementation of the shift register is identical to that already illustrated for the Instruction Register, the only differences lie in the signal, input and output connections. The serial input of the register is connected to the input of the module, `td_i`, the parallel is instead connected to the output of the standard register, to load the IDCode value of the device. The serial output is connected to the multiplexer controlled by the instruction in the Instruction register, being able to supply the IDCode via shift to the output of the TAP module. The parallel output is not used, as the standard register cannot be updated, the IDCode being constant. The signal `enable`, is connected to the signal `idcode_enable` generated by the instruction present inside the Instruction Register, enabling the register only in the case of IDCODE instruction. Instead, the signals `shift` and `capture` are connected to the signals `shift_dr` and `capture_dr`, being able to shift the value of the IDCode during the state of `Shift_Dr` or load it to the register in the state of `Capture_Dr`. In the remaining cases, the IDCode register keeps the value within it constant.

Finally, the Bypass register is implemented as a single-bit register with only serial input. Its output is directly connected to the multiplexer controlled by the Instruction register. In addition to the signals `clock` and `reset`, it supports the signal `bypass_enable`, asserted when the BYPASS instruction is present in the Instruction register, the signal `shift_dr`, which allows the loading of the value present on `td_i` inside the register, and the signal `capture_dr`, which from specification resets the value of the register to 0. In all other cases, the Bypass Register keeps the value present within it fixed.

Additional Signals

The last remaining signals are those related to the BSR, which is connected externally to the TAP module. The signals of `capture_dr`, `shift_dr`, and `update_dr` are propagated externally, together with the input data from the TAP module, via the `td_i` signal. The output of the BSR is fed back into the TAP module and connected to the multiplexer managed by the Instruction Register, like the other Data Registers.

In conclusion, the value assignment of the output `td_o` of the TAP module, is done on the falling edge of the clock, as stated in the specification.

After careful consideration of the inclusion in LEN5, of support for the JTAG protocol, it was decided to avoid deep structural changes to the core, preferring support for the approach called *Execution Based*, requiring minimal changes on the core, on which the debug module present inside X-HEEP is based. Therefore, a simulation phase of the TAP module was not performed and its correct operation cannot be guaranteed, although it is

compliant with the IEEE 1149.1 standard specification.

6.2 Adaptation of LEN5 for X-HEEP's Debug System Support

This section will outline the changes made to the architecture of LEN5, for debug mode support and to allow it to work in tandem with the Debug Module present within X-HEEP and with other Debug Modules of the same kind. Keep in mind, however, that at this stage of development, the purpose is not to integrate all debug mode components into the processor, but to introduce only its basic support.

The module present within X-HEEP conforms to the specifications outlined in [20]; the implementation used is called *Execution Based*, as the name suggests, it is based on using the processor pipeline to execute code placed at any location in memory, avoiding substantial changes to the processor architecture itself. For this reason, we will not dwell on the analysis of the debug module, the processor being completely transparent to its operation.

Before analyzing the changes made to the processor, it is necessary to introduce in general, the process of entering debug mode. What will be illustrated can be found in the RISC-V debug specification [20]. The LEN5 processor, called `hart` in the debug environment, supports two methods of entry at present. The first method is the reception of the `debug_req` signal, referred to as `halt request`, sent by the debug module, which acts as an interrupt for the processor. Upon receipt of the request, the processor will have to save the current PC and the cause of the debug mode entry within two CSRs, specifically introduced to store debug mode information. Next, the processor will have to modify its PC with a memory address provided to it externally, via the signal `dm_halt_address_i`, corresponding in the present case to the debug rom.

The second method involves the execution of the EBREAK instruction. Depending on the configuration of the processor, it is possible to enter debug mode, with the associated PC and debug mode entry cause saved within the CSRs, when an instruction of this type is executed outside debug mode.

Each debug mode entry method has a different priority, the EBREAK instruction has priority 3 while the `debug_req` has priority 1. In the event that there are multiple simultaneous requests for debug mode access, in the case under consideration the execution of an EBREAK with at the same time the arrival of a `debug_req`, the processor will have to access by setting the higher priority cause in the respective CSR.

Let proceed then to the analysis of the changes made to LEN5 for debug mode support.

6.2.1 Control Status Registers (CSRs)

For debug mode support, the RISC-V specification, [20], calls for the introduction of specific CSRs within the processor, named *DPC, DCSR, Dscratch0 and Dscratch1.* Each of these registers is accessible only while in debug mode, otherwise a *exception* is propagated, signaling invalid access. In addition, a custom CSR, `dm`, has been placed within LEN5, used as a flag for storing the processor execution mode, debug mode or normal mode. Accessible in Read/Write, it is used for: blocking counters, having hardwired the value `stopcount=1` in the DCSR, recognizing the current state of the processor in the case of `debug_req`, handling the DRET instruction in the issue stage, handling the ECALL, EBREAK, MRET instructions and exceptions in the Commit Stage and for regulating access to the registers below.

The CSRs *Dscratch0 and Dscratch1* are normal 64-bit registers, the use or non-use of which depends on the connected debug module. From specification, implementation is optional, they are, however, inserted for compatibility, in case the processor is connected to a debug module that requires their use. Both are read/write accessible.

The CSR *DPC*, Debug Program Counter, is also a 64-bit register that is read/write accessible. It is used to save the current PC when the processor enters debug mode and as a source register from which to retrieve the PC to resume code execution when the processor exits debug mode.

The last of the mandatory CSRs to be specified is the *DCSR*, a 32-bit register used to configure the behavior of the processor while in debug mode, by means of the value present within its fields[2]. Each of these fields has different access modes, read-only, read/write, or WARL.

- **xdebugver** → Identifies the type of external debugging supported by the processor. Hardwired to value 4, indicating support for debugging present in the RISC-V specification. Read Only.
- **ebreakm** → Indicates whether or not the EBREAK instruction should force entry into debug mode when executed in M-Mode. Accessible in Read/Write.
- **ebreaks** → Indicates whether or not the EBREAK instruction should force entry into debug mode when executed in S-Mode. Accessible in Read/Write.
- **ebreaku** → Indicates whether or not the EBREAK instruction should force entry into debug mode when executed in U-Mode. Accessible in Read/Write.
- **stepie** → Indicates interrupt enable during single step execution in debug mode.

[2]The descriptions of the DCSR fields in this document are as given in [20], rely on it for further details.

Hardwired to 0 in order to disable receiving interrupts, as they are not currently supported by LEN5. Accessible in WARL, being hardwired the value read is necessarily legal.

- **stopcount** → Indicates the enabling of counters present in CSRs. Hardwired to 1 to disable counters during Debug Mode. Accessible in WARL.
- **stoptime** → Indicates the enabling of timers present in CSRs. Hardwired to 1 to disable timers, as they are not present in LEN5. Accessible in WARL.
- **cause** → Specifies the reason for entering debug mode. In the case of the LEN5 processor, two values can be found within it, 1 if the cause of entry is the EBREAK instruction or 3 if the cause is the receipt of the `debug_req`. Read Only from the outside, set only at the time of debug mode access via assertion of the `debug_csr_write` signal by the Commit CU.
- **mprven** → Specifies whether the Privilege Mode present in CSR `mstatus` also applies in debug mode or not. Hardwired to 1 to apply the contents of `mstatus`, currently in fact LEN5 only supports M-Mode. Accessible in WARL.
- **nmip** → If set, signals the presence of an unmaskable interrupt. Read Only from outside, set only at the time of debug mode access.
- **step** → Indicates the enablement of single step execution. This mode is not currently supported by LEN5. Accessible in Read/Write.
- **prv** → Specifies the Privilege Mode of the processor upon entering Debug Mode. Accessible in Read/Write.

6.2.2 Issue Stage Modification

As much as the signal `debug_req` is considered an interrupt request, there is no specification on how this should be interpreted by the processor for debug mode entry. Therefore, it was decided to implement the `debug_req` signal as an exception, placing it in the Issue Stage and propagating it in the processor pipe until it reaches the Commit Stage, within which debug mode entry will be handled.

The Issue Stage is responsible for receiving the incoming instruction from the Issue Queue, decoding it, and fetching the operands[3]. Having done so, it sends the data to the Reorder Buffer (ROB), which stores the instruction order for the next commit. The `debug_req` is received by the Issue CU, where it acts as an enable, along with the denied content of the CSR dm, for a new register called the Debug Sampler, whose operation is essentially that of a flag, necessary to keep the `debug_req` pending, until the Issue CU is able to process it. Debug mode entry is in fact second in priority to resolving any

[3]For more on the Issue Stage, refer to [3].

mispredictions, which would otherwise cause the instruction to be executed in the wrong order.

Checking the current state of the processor in the register enable is necessary to avoid sampling `debug_req` while it is in debug mode, otherwise there would be a risk of having the processor re-enter debug mode as soon as it exits a previous debug mode.

The Issue CU's internal future-state precomputation logic, which is present to simplify the future-state computation logic found in every FSM, was modified by adding a check for the presence of either an input `debug_req` or a pending `debug_req` (OR operation between the two requests). In case either one is present and not already in debug mode, signaled by the CSR DM, the state of the Issue CU will evolve into a new state called S_ISSUE_DEBUG.

Once the CU enters S_ISSUE_DEBUG, it asserts the `commit_valid`, signaling the ROB to place within it the instruction received along with the debug request. In fact, the instruction will never actually be entered into the ROB, but this does not generate any error since, with the entry into debug mode, the current PC is saved in the CSR DPC and once the PC is restored with the exit from debug mode, the current instruction will be re-executed. Along with the previous signal, the signals `debug_req_clr` and `issue_debug_sel_o` are asserted. The former resets the Debug Sampler, having the CU handled the `debug_req` and initiated the debug mode input process. The second is the selector of a multiplexer added on the Issue Stage outputs, connected to the ROB input. The multiplexer in question allows a dummy entry, called DEBUG_REQ_ROB_ENTRY, to be loaded into the ROB, within which are set:

- **order_crit** = 1, which is necessary to signal the inability to commit out-of-order at the commit stage. Debug mode entry must wait for the commit of instructions that arrived earlier in order to respect the correct order of execution. In fact, the execution of certain instructions has different routines when executed in debug mode, moreover, failure to comply with this constraint would cause the ROB to flush with still inside uncommitted instructions prior to the address saved in the CSR DPC, corresponding to the instruction with which the `debug_req` arrived and to which the processor will jump once it exits debug mode.
- **except_raised** = 1, saves in the ROB the arrival of an exception concurrently with the instruction.
- **except_code** = E_DEBUG, identifier of the type of exception received, in the case of `debug_req` a custom code named E_DEBUG has been integrated.
- **curr_pc** equal to the PC of the current instruction that will not be entered in the ROB.

The remaining fields of the entered entry are initialized to 0, as they are not needed to

handle the `debug_req`. Next, the CU proceeds to the S_STALL state, in which it remains stalled waiting for the `debug_req` to be handled by the commit stage. Once complete, the commit sends the `comm_resume` signal, unlocking the issue stage from stall.

The second debug mode entry is the execution of an EBREAK. As specified in the previous section, the processor can be configured, through the `ebreakm` field in the DCSR, to enter debug mode in case an EBREAK instruction is executed outside of it. The issue decoder was modified to signal the Issue CU, by sending a specific code (ISSUE_TYPE_EBREAK), that the instruction in question had been recognized. Upon receiving the code from the issue decoder, it will be the Issue CU that will check whether the EBREAK instruction should force entry into debug mode, via the `ebreakm` field of the DCSR, before checking for the presence of a `debug_req`, so as to respect the entry priority. In case the check is positive, the CU will evolve to a new specific state (S_ISSUE_EBREAK_DM), in which it resets the Debug Sampler and asserts the signal `commit_valid`, signaling the ROB to insert the instruction within one of its entries, otherwise the EBREAK will be treated as an exception. In this way, the instruction is actually entered into the ROB, along with the `order_crit` = 1 signal, the `skip_eu` = 1 signal, the exception signal, and the associated E_BREAKPOINT code, and is not replaced by the dummy entry, allowing the commit stage to handle it appropriately. Also in this case, the CU proceeds to the S_STALL state, waiting for the instruction commit and the `comm_resume` sent by the commit stage to unlock the issue stage.

The exit from debug mode is not governed by a signal like the entry; rather, an ad hoc instruction, found within the ISA's `rv_sdext` extension, called `dret`, is used. To support the instruction, the issue decoder was again modified by adding recognition of the `opcode` of the `dret`. When recognized, the issue decoder generates several signals:

- **skip_eu** = 1, is asserted to signal the bypass of the execution unit, no computation being necessary for the `dret`.
- **order_crit** = 1, as in the case of debug mode entry, the exit must also be executed in order, so that all instructions issued during debug mode and present in the ROB, are committed while it is still active.
- **opcode_except** = ∼dm, as described by the risc debug mode specification [20], the instruction `dret` must cause an exception, if executed outside the debug mode. Therefore, the signal is generated directly from the negated value of csr dm, which is used to indicate debug mode execution or not.
- **except_code** = E_ILLEGAL_INSTRUCTION, code of the exception propagated by the `dret`, in case it was executed outside debug mode.
- **issue_type** = ISSUE_TYPE_NONE, used to signal to the issue CU what type of instruction is being issued, so that it can generate the `ready` and `valid` signals needed

for the issue queue and commit stage. In this case, the CU signals the commit stage that the instruction is valid for insertion into the ROB, via the signal `comm_valid_o` and unlocks the issue queue at the moment the commit stage signals that it has inserted the instruction into the ROB, `iq_ready_o = comm_ready_i`.

6.2.3 Commit Stage Modification

Actual entry into debug mode is handled by the Commit Stage, which is responsible for committing instructions in the ROB, forwarding instruction operands to previous stages, and handling exceptions[4]. The first module to deal with the presence of the exception derived from the `debug_req` is the *Commit Decoder*, which is responsible for recognizing the instruction ready to commit, detects the assertion of the exception present in the dummy entry within the ROB, signaling its presence to the Commit CU by generating the code `comm_type = COMM_TYPE_EXCEPT`. The Commit CU was modified by adding support for four new states:

- COMMIT_DEBUG
- COMMIT_DEBUG_SAVE_PC
- DEBUG_WRITE_CODE
- DEBUG_LOAD_PC

In addition, an extra check has been inserted into the future state precomputation logic which, upon receiving the code sent by the decoder, checks for coincidence between the `except_code` propagated along with the exception and the E_DEBUG code. If so, it will generate as a future state, the new state COMMIT_DEBUG. Once it enters the COMMIT_DEBUG state, the Commit CU will finalize the entry into debug mode by executing the steps described in the opening paragraph. The following describes the state steps with the operations performed by the CU:

1. **COMMIT_DEBUG** → A write is performed in the CSR DM, setting the flag to signal entry into debug mode, also the execution pipeline flush is performed, having to subsequently update the PC with the address of the debug rom, from which to resume execution. Finally, the debug sampler present in the issue stage is reset, to prevent another `debug_req` from being saved in the time between stalling the issue and writing to the CSR DM.

2. **COMMIT_DEBUG_SAVE_PC** → A write to the CSR DPC is executed, saving within it the value of the PC received along with the `debug_req`.

[4]For more on the Commit Stage, refer to [3]

3. **DEBUG_WRITE_CODE** → A write is executed in the DCSR CSR, saving the cause of debug mode entry in the `cause` field of the DCSR, in this case `debug_req` (value 3), and the active execution mode, M-MODE. Note that the CU simultaneously asserts the `csr_debug_write` signal, which is mandatory to keep the `cause` and `nmip` fields of the DCSR in Read Only, preventing writes from outside, as visible in the specification [20]. In addition, front-end flushing is performed, so that the PC can be modified with that of the debug rom, provided externally via the `dm_halt_address` bus.

4. **DEBUG_LOAD_PC** → The address present on the `dm_halt_address` bus is sent to the fetch stage as input, to be loaded into the PC to access the debug rom.

5. **CLEAR_COMM_REG** → The commit register is reset, via the signal `comm_reg_clr`.

All writing within the CSRs is done through the use of a multiplexer, called the CSR Multiplexer, instantiated in the Commit Stage and controlled by the Commit CU, via the signal `comm_csr_sel`. It was necessary to expand the number of inputs present, to make it possible to write the debug mode access cause and the current execution mode, in the case of DCSR access, and to write a value of 1, in the case of CSR DM access, to signal that debug mode has been entered.

The forwarding to the fetch stage of the address contained in the `dm_halt_address` bus, on the other hand, is done by a second multiplexer, called exception multiplexer, instantiated on the output of the exception adder. Again, it was necessary to expand the number of multiplexer inputs to support the different instances of PC modification, due not only to the loading of the PC present in a given CSR and the execution of an exception handler, but also to the access in debug rom and the handling of an exception during debug mode.

In fact, the RISC-V debug specification requires that exception handling while in debug mode differs from standard handling. Specifically, the arrival of an exception must not cause any CSR to be updated, and the exception handler loaded must be the one provided via the address on the input `dm_exception_addr` bus. To do this, support for two additional states has been included in the Commit CU, named COMMIT_EXCEPT_DM and EXCEPT_LOAD_PC_DM. At the moment when the commit decoder signals the arrival of an exception, if it is not due to a `debug_req` and therefore the `except_code` is not E_DEBUG, the CU checks the value present inside the CSR DM and, if this signals debug mode execution, the state is updated to COMMIT_EXCEPT_DM. Otherwise, the state advances to COMMIT_EXCEPT, handling the exception in the standard way. In the new state, the CU deals with flushing the execution pipe and front-end, having to jump to another memory location. It then transitions to

the EXCEPT_LOAD_PC_DM state, in which it selects the exception multiplexer input corresponding to the bus `dm_exception_addr`, sending it to the fetch stage for PC update. Finally, the CU evolves into the CLEAR_COMM_REG state, emptying the commit register.

Once debug mode accessibility is built in, the RISC-V debug specification provides a set of mandatory behaviors in case the Debug Module, connected to the processor, supports execution from Program Buffer, ch.4 of [20]. This being the case for the Debug Module of X-HEEP, the following was incorporated:

- All debug mode operations should be performed in M-Mode, except that it is possible to override the value in `mprv` present in the CSR mstatus, via the value in the DCSR `mprven` field. → the `mprven` field is hardwired to 1, so the value in `mprv` is also applied in debug mode, but since LEN5 currently supports only M-Mode, the specification is automatically met.
- All interrupts are masked. → At the time of writing this thesis, LEN5 has no support for interrupts, so this specification is also respected automatically.
- Exceptions do not update any CSRs and terminate the execution of the Program Buffer. → As explained earlier, 2 states have been added within the Commit CU that take care of exception execution in debug mode, without updating the CSRs and loading the PC with the value present in the bus `dm_exception_addr`.
- No action is taken if there is a trigger match. → LEN5 has no Trigger Module inside it, so the specification is met.
- Counters and Timers can be blocked if specified in the `stopcount` and `stoptime` fields of the DCSR. → As introduced in the section on CSRs, these two fields are hardwired to value 1, locking both counts during debug mode by checking the contents of the `dm` flag.
- The instruction `wfi` behaves like a `nop`. → As specified earlier, LEN5 does not yet support the interrupt mechanism, which is why `wfi` instructions are already interpreted as `nop`.
- All instructions that change privilege levels have undefined behavior in debug mode, except for the EBREAK instruction, which causes the PC to be changed to that contained in the `dm_halt_address` bus, without changing any CSR. → For both instructions that modify privilege levels, such as ECALL and MRET, additional states have been inserted within the Commit CU, such as COMMIT_ECALL_DM and COMMIT_MRET_DM. In both cases, the CU commits the instruction without updating the CSRs, after which it switches to the EXCEPT_LOAD_PC_DM state, loading the value on the `dm_exception_addr` bus to the PC.
  The handling of the EBREAK instruction, however, is more complex. This, in fact,

can not only be executed with a different routine while in debug mode, as just described, but can also cause entry into debug mode if executed outside it, depending on the configuration contained in the DCSR, as explained in the introduction to this section and in the section on issue stage.
To comply with both specifications, the commit decoder was modified, adding an additional check in the case of exception. Until now, in fact, in case an exception was present in the ROB, the code sent to the CU would be *COMMIT_TYPE_EXCEPTION*, regardless of the type of instruction, forcing the Commit CU to execute the routine for exceptions, loading the address of the exception handler. Thanks to the added check, the decoder is now able to identify whether the instruction with which the exception arrived, is an EBREAK, and if so, send the code *COMMIT_TYPE_EBREAK* to the CU. The Commit CU can handle the instruction, through the introduction of four additional states, COMMIT_EBREAK_DM, COMMIT_EBREAK_FORCE_DM, EBREAK_SAVE_PC, and EBREAK_WRITE_CODE, respectively.
Upon receiving the EBREAK identifier from the decoder, the future state precomputation logic checks whether the processor is in debug mode; if so, the Commit CU evolves to COMMIT_EBREAK_DM state, committing the instruction and flushing the execution pipeline and front-end. Thereafter, the CU evolves to DEBUG_LOAD_PC state, loading the initial address of the debug rom contained in the `dm_halt_address` bus into the PC.
In the case, however, that the processor is not in debug mode, the CU checks the value in the `ebreakm` field of the DCSR and if equal to 1, signaling the requirement to switch to debug mode by executing an EBREAK, the state evolves to COMMIT_EBREAK_FORCE_DM. In this state, the CU accesses the CSR DM, setting the flag and signaling entry into debug mode, simultaneously flushes the execution pipeline and resets the debug sampler in the issue stage, to again prevent a `debug_req` from being received between the issue stage stall and the assertion of the CSR DM flag. The next state is EBREAK_SAVE_PC, in which the CU saves the PC of the EBREAK instruction within the CSR DPC and then proceeds to the EBREAK_WRITE_CODE state, in which the instruction is committed, the cause of debug mode entry is saved within the `cause` field of the DCSR, in this case it is EBREAK (value 1), the current execution mode (M-MODE) is saved in the `prv` field, and then the front-end is flushed. Once the EBREAK execution routine is completed, the CU evolves to DEBUG_LOAD_PC state, sending the address present on the `dm_halt_address` bus to the fetch-stage, to load it into the PC and complete the debug mode access.
Last, if the processor were not to be in debug mode and the DCSR configuration did not force entry into debug mode for EBREAKs, the instruction in question would be

executed as a normal exception, evolving the CU state to COMMIT_EXCEPTION, saving the PC in the MEPC CSR and the cause of the exception in the MCAUSE CSR, and then computing the exception handler address from the value present inside the MTVEC CSR.

- The completion of the Program Buffer is considered output for the instruction `fence`. → Unfortunately, this condition is not satisfiable, as LEN5 does not yet support the `fence` instruction.
- All instructions that modify the PC, either by pointing to a location inside the Program Buffer or outside, or that depend on the PC value, could be considered illegal. → By choice of implementation, it was decided to keep the standard operation of this type of instruction.

The exit from debug mode, as already introduced in the issue stage, is controlled by the instruction `dret`. Therefore, its support was introduced, within the commit decoder, which signals to the Commit CU the occurrence of the instruction via the code *COMMIT_TYPE_DRET*. Two additional states, DRET_LOAD_PC and COMMIT_DRET, have been introduced within the Commit CU. As soon as the CU receives code from the decoder, it evolves into the COMMIT_DRET state, where the instruction is committed, a write to the CSR DM is executed, deasserting the flag signaling debug mode execution, and the execution pipeline and front-end are flushed, having to load the PC to which to jump to exit debug mode. Following this, the CU proceeds to the DRET_LOAD_PC state, here it performs a read of the DPC CSR and through the selection of the output in the exception mux corresponding to the input data from the CSRs, sends the PC contained within it to the fetch stage, ending debug mode.

Two unfinished specifications remain. It is necessary to include support for the instruction `fence`, for sorting the memory accesses after the Program Buffer execution is terminated, also the single step execution mode must be integrated. Unfortunately, due to timing problems, the development of debug mode had to be stopped. Therefore, implementations of the two missing specifications and testing of the operation of the debug mode of LEN5 will be left for future development.

Chapter 7

Concluding remarks

The work illustrated in this thesis led to the integration of a processor based on an Out-of-Order architecture within a Low Power Heterogeneous SoC, through the development of an interfacing module capable of ensuring effective communication without impacting system performance and through the inclusion of debug mode support within the processor itself. The results obtained are very encouraging, leading to a tangible improvement of the SoC in terms of performance and, at the same time, opening concrete development scenarios for the processor itself, exploiting the platform as a development environment.

7.1 Further Improvements

Several aspects of the Bridge module can be improved, beyond a general optimization of the various components, the most important changes could be:

- Implementation of Outstanding Requests. The Bridge currently does not allow requests to be accumulated internally to be sent to the bus when it is available to accept them. More advanced buses may implement the feature and request support for it.
- Misaligned Access Support. Although the Bridge supports recognition of misaligned accesses, it is not yet possible to handle them in a consonant manner. It would be possible to modify the address splitter and CUs to support misaligned accesses on WORD and HALFWORD.

Regarding Debug mode, it has already been discussed in the related section regarding the still missing specifications:

- Support of the Fence instruction. LEN5 does not yet support this type of instruction, which is required at the end of the execution of the Program Buffer contained in X-HEEP.

- Single step mode. To be fully compliant with the RISC-V specification, it is necessary to introduce the possibility of single step execution of instructions, within the processor.
- Trigger Module. Required for trigger and breakpoint support.

Finally, I would like to join the collective wish of the creators of LEN5. That this step of integrating the processor within a SoC would increase even more its notoriety in the academic community, prompting more and more students and researchers to take an interest in its development and improvement.

Bibliography

[1] Marco- Andorno. *Design of the frontend for LEN5, a RISC-V Out-of-Order processor.* Dec. 2019

[2] Matteo Perotti. *Design of an OS compliant memory system for LEN5, a RISC-V Out-of-Order processor*. Dec. 2019

[3] Michele Caon. *Design of the execution pipeline for LEN5, a RISC-V Out-of-Order processor.* Dec. 2019

[4] *History of RISC-V* url: https://riscv.org/about/history/

[5] Abigail Opiah. *What is RISC-V and why is it important?* January 11, 2024 url: https://riscv.org/news/2024/01/what-is-risc-v-and-why-is-it-important/

[6] S. Machetti, P. D. Schiavone, T. C. Müller, M. Peón-Quirós, D. Atienza. *X-HEEP: An Open-Source, Configurable and Extendible RISC-V Microcontroller for the Exploration of Ultra-Low-Power Edge Accelerators.* arXiv preprint arXiv:2401.05548 (2024).

[7] *X-HEEP Documentation*. 2023 url: https://x-heep.readthedocs.io/en/latest/index.html

[8] Silicon Labs, Inc. *OBI-v1.2.* 2023

[9] David Patterson, John L. Hennessy. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Pub, 2017

[10] RISC-V Foundation. *RISC-V: The Free and Open RISC Instruction Set Architecture.* 2019 url: https://riscv.org/

[11] David Patterson, John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface: Risc-V Edition.* Morgan Kaufmann Pub, 2017

[12] Andrew Waterman and Krste Asanovic. *The RISC-V Instruction Set Manual. Volume I: User-Level ISA.* RISC-V Foundation, May. 2017

[13] *TCBN65LP TSMC 65nm Cole Library Data Book, version 1.4*. Sep. 2008

[14] Olof Kindgren. *FuseSoC Documentation, Release 0.0.0.* 2024 url: https://github.com/fusesoc/fusesoc.github.io

[15] TIS Committee. *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification, Version 1.2.* Linux Foundation. May 1995

[16] *System V Application Binary Interface - DRAFT.* Linux Foundation, 24 Apr. 2001 url: https://refspecs.linuxfoundation.org/elf/gabi4+/contents.html

[17] *What is JTAG and how can I make use of it?* url: https://www.xjtag.com/about-jtag/what-is-jtag/

[18] *Technical Guide to JTAG* url: https://www.xjtag.com/about-jtag/jtag-a-technical-overview/

[19] *IEEE Standard Test Access Port and Boundary-Scan Architecture.* 14 Jun. 2011

[20] *RISC-V External Debug Support. Version 0.13.2.* RISC-V Foundation, Mar. 2022

Acronyms

CISC Complex Istruction Set Computer. 3, 95

CPU Central Processing Unit. 9

CSR Control Status Register. 66

CU Control Unit. 23

FIFO First In First Out. 21, 27

FSM Finite State Machine. 23

ISA Istruction Set Architecture. 3, 15

LSB Least Significant Bit. 29

MSB Most Significant Bit. 35

MUX Multiplexer. 28

OoO Out-of-Order. 2

PC Program Counter. 21

RISC Reduced Istruction Set Computer. 3, 96

ROB Reorder Buffer. 4, 82

WAR Write after Read. 4

WARL Write-Any Read-Legal. 81

WAW Write after Write. 4

Glossary

Aligned Access Aligned Access refers to access to a memory to retrieve a data therein, which is arranged on sections of memory with addresses multiple of the number of bytes of which the data is composed. This allows single accesses to be made for a single piece of data. 14

BSR Boundary Scan Register, a register placed between the pins and logic of a component, allows its monitoring in the testing and debugging phases, as well as the possibility of forcing signals on its interface. 71

CISC Complex Istruction Set Computer. Architecture based on the execution of complex instructions that can read, modify, or save a data directly into memory. 3

Commit Final stage of execution of an instruction, the result is written to the Register File/Memory. 5

Fetch Reading the instruction present in instruction memory, pointed by the Program Counter. 3

GCC GNU Compiler Collection, cross-platform compiler supporting various languages, including C, C++, Java, and various architectures, such as x86, x86-64, ARM, RISC-V. 64

High-Performance Computing High Performance Computing, commonly referred to by the acronym HPC, is the set of technologies used to create computing systems with very high computing performance. i

ISA An acronym for Instruction Set Architecture, it defines the set of instructions that a computer can interpret and execute at the Hardware level. 2

Issue Instruction decoding and assignment to a Functional Unit. 2

JTAG An acronym for Joint Test Action Group, it is a consortium of 200 companies manufacturing integrated circuits and printed circuit boards in order to define a standard protocol for functional testing of these devices. i

LEN5 LEN5 is a single issue, out-of-order execution processor based on RISC-V architecture. i, 2

LOAD Instruction belonging to the basic RISC-V ISA. One of only two instructions that can access memory. It performs the read of a data. 3

Misaligned Access Misaligned Access refers to access to a memory to retrieve a data therein, which is arranged on sections of memory with addresses that are not multiples of the number of bytes of which the data itself is composed. This forces multiple accesses to be made for a single piece of data. 14

Out-of-Order Out-of-Order execution refers to the execution of instructions, issued and present in the ROB, not necessarily in the order defined by the user, but rather at the time when the source registers are available and there are no hazards with other previous instructions or the instruction appears order critical. 2

RISC-V RISC-V is an open-source instruction set architecture (ISA) operated by the nonprofit RISC-V Foundation organization. RISC stands for "Reduced Istruction Set Computer" (computers with a reduced instruction set), while "V" is the Roman numeral to signify the fifth generation of the architecture. 2

STORE Instruction belonging to the basic RISC-V ISA. One of only two instructions that can access memory. It performs the write of a data. 3

TAP Test Access Port, interface designation of JTAG. 72

X-HEEP X-HEEP is an eXtendable Heterogeneous Energy-Efficient Platform, a configurable, extensible and heterogeneous microcontroller platform, complete with every component: CPU, memories and peripherals. i, 2