Internship report

BADOUI Aziz

Micro and Nanotechnologies for integrated systems
2022-2023

Advanced High-performance Bus (AHB) implementation

Melexis NV
4 Pl. des Vosges, 92400 Courbevoie
from 13/02/2023 to 13/08/2023

Under the supervision of:

Company Supervisor : CHEREL Louis , loc@melexis.com
PHELMA Tutor : ANGHEL Lorena , lorena.anghel@phelma.grenoble-inp.fr
Confidential : Yes

Politecnico di Torino

Phelma

EPFL

Acknowledgments

I would like to express my sincere appreciation to the entire Digital Component Center (DCC) at Melexis Paris for their warm welcome and invaluable contributions that made my internship and the completion of this report possible. First and foremost, I want to thank Louis Cherel, my supervisor, whose guidance, encouragement, and patience were instrumental throughout the six-month internship. I am also grateful to Paul Baron, a senior digital designer who generously shared his knowledge and experience with all the interns, enriching our learning journey. Lastly, I cannot conclude this segment without acknowledging the unending support of my family and friends, whose encouragement was a constant pillar of strength.

This internship experience at Melexis Paris has been nothing short of transformative. It has solidified my decision to continue working with the company as an Associate Digital Engineer, and I eagerly await the new journey that lies ahead.

Contents

List of Figures

List of Tables

Abstract

This report presents an enriching internship experience at Melexis Paris, centered on implementing the AMBA AHB 5 bus protocol for the innovative Callisto platform. The internship involved the creation of both slave and master models adhering to the protocol's specifications and developing a bridge enabling seamless communication between a RISC-V CPU and the new platform. Additionally, an efficient interconnect system was designed, facilitating smooth communication among multiple masters and slaves. This successful integration promises reduced access time and flexible chip utilization, establishing Callisto as an advanced platform to develop cutting-edge products for our company's expanding client portfolio.

Ce rapport présente une expérience enrichissante de stage chez Melexis Paris, axée sur l'implémentation du protocole de bus AMBA AHB 5 pour la plateforme innovante Callisto. Le stage a impliqué la création de modèles maîtres et esclaves conformes aux spécifications du protocole, ainsi que le développement d'un pont permettant une communication fluide entre un processeur RISC-V et la nouvelle plateforme. De plus, un système d'interconnexion efficace a été conçu pour faciliter la communication harmonieuse entre plusieurs maîtres et subordonnés. Cette intégration réussie promet un temps d'accès réduit et une utilisation flexible des puces, établissant ainsi Callisto comme une plateforme avancée pour le développement de produits de pointe destinés à notre portefeuille grandissant de clients.

Questa tesi presenta un'esperienza arricchente di stage presso Melexis Paris, incentrata sull'implementazione del protocollo di bus AMBA AHB 5 per la piattaforma innovativa Callisto. Lo stage ha coinvolto la creazione di modelli master e slave conformi alle specifiche del protocollo, nonché lo sviluppo di un ponte che consente una comunicazione fluida tra un processore RISC-V e la nuova piattaforma. Inoltre, è stato progettato un sistema di interconnessione efficiente per agevolare una comunicazione armoniosa tra vari master e sottoposti. Questa integrazione di successo promette un tempo di accesso ridotto e un'utilizzo flessibile dei chip, stabilendo così Callisto come una piattaforma avanzata per lo sviluppo di prodotti all'avanguardia destinati al nostro crescente portafoglio di clienti

Acronyms

ADC Analog to Digital Converter. 10

AHB Advanced High Performance Bus. 2–4, 6–8, 10, 11, 13, 14, 16–18, 21–23, 26, 27, 29, 31, 34, 38, 47

ALU Arithmetic Logic Unit. 17

AMBA Advanced Microcontroller Bus Architecture. 2, 4, 7, 8

ARM Advanced RISC Machines. 4

ASIC Application Specific Integrated Cicuit. 1

CPU Central Processing Unit. 2, 17, 19

DCC Digital Component Center. 1

DMA Direct Memory Access. 10

ICs Integrated Cicuits. 1

IP Intellectual property. 4

ISA Instruction Set Architecture. 2, 17

LED Light Emitting Diode. 1

RAM Random Access Memory. 10, 17, 20

RISC Reduced Instruction Set Computer. 2, 3, 17

ROM Read Only Memory. 10, 17, 19, 20

RTL Register Transfer Level. 2, 26

SOC System On Chip. 4, 8

1 Introduction

1.1 Company overview

This report outlines my internship experience at Melexis Paris, a distinguished division of Melexis, a medium-sized semiconductor company operating globally. Melexis specializes in the design, production, and distribution of Application Specific Integrated Cicuit (ASIC) tailored for the automotive industry, where modern vehicles integrate diverse Integrated Cicuits (ICs) to enhance intelligence and performance as shown in fig 1. The company offers a wide array of sensor solutions, including pressure sensors, hall effect motion sensors, and current sensors. Additionally, they provide electric motor drivers and actuators for various applications like pumps, electric seats, and windows, as well as Light Emitting Diode (LED) drivers and controllers used in high-end cars for features like stylish blinkers.

During my internship, I had the privilege of working within the Digital Component Center (DCC) at Melexis Paris, located in the vibrant business district of La Defense, ranked as the 4th most attractive business neighborhood worldwide. This district is home to several prestigious international corporations, creating a stimulating environment for innovation and growth. Focusing solely on the design and implementation of advanced digital circuits, the Parisian division stands as a hub of expertise in this vital field. Leveraging its technical know-how and extensive experience, Melexis consistently delivers high-performance and reliable solutions to its global clientele.

Throughout this internship, I had the privilege of collaborating with a highly skilled team of 20 members, including four interns. The atmosphere at Melexis Paris fostered a culture of collaboration and support, creating a dynamic learning environment where knowledge-sharing and teamwork were actively encouraged. This experience has been invaluable in shaping my understanding of digital design and its practical applications in the semiconductor industry.

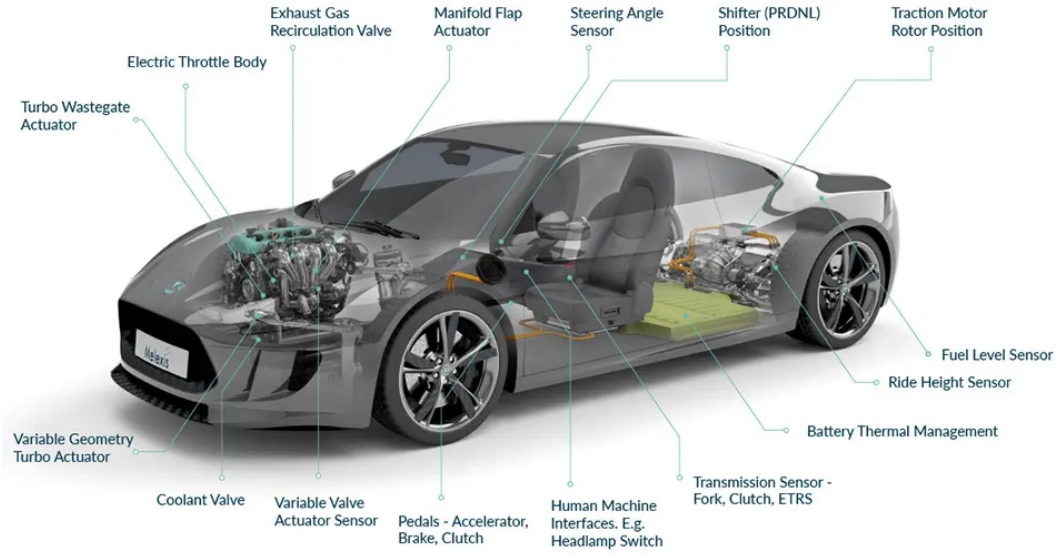


Figure 1: ICs in Modern Vehicles [2]

1.2 Topic Overview

At Melexis Paris, my internship focused on Register Transfer Level (RTL) digital design, specifically the design and implementation of the Advanced Microcontroller Bus Architecture (AMBA) Advanced High Performance Bus (AHB) 5 protocol. This initiative is part of the company's ambitious project, Callisto, aimed at integrating an ARM and/or Reduced Instruction Set Computer (RISC) V CPU into their new platform. By doing so, Melexis aims to enhance computational power and frequency, aligning with its commitment to providing cutting-edge solutions to its customers. Additionally, the project aimed to offer a standard Instruction Set Architecture (ISA) instead of the custom one developed by Melexis and used in previous projects, allowing clients greater flexibility in utilizing our chips.

The design process for the AMBA AHB 5 bus protocol was carried out using the hardware description language SystemVerilog, which offers powerful capabilities for modeling and simulating digital systems. An example of the written code is provided in Annex. The Cadence tool suite was utilized for verification and simulation, providing a comprehensive and efficient environment for developing and validating the protocol implementation.

1.3 Organisation of the internship

The organization of an internship plays a pivotal role in shaping its success and ensuring the accomplishment of desired objectives. This section aims to provide an overview of how the internship program is structured, including the milestones, plan, and objectives.

The initial step involved comprehensively understanding the AHB protocol, completely detailed and explained in the official specification document provided by ARM [6], and comparing it with the older Wishbone bus protocol used in previous platforms. This analysis provided valuable insights into the enhancements and benefits offered by the AHB protocol. Subsequently, we proceeded with the implementation process by creating a slave model specifically designed to comply with the new protocol. Thorough test benches were developed to verify its functionality and alignment with the expected behavior. Similar efforts were dedicated to the development of the master model, establishing a robust foundation for the overall project.

With functional slave and master models in place, the next focus shifted to the interconnect between them. This interconnect was divided into two parts: the single master interconnect, allowing a master to communicate with multiple slaves, and the single slave interconnect, enabling multiple masters to communicate with a single slave. The ultimate goal was to merge these interconnects into a single system that facilitates communication between multiple masters and multiple slaves. This milestone represents the successful implementation of the AHB bus protocol and establishes a solid foundation for future work on the Callisto platform..

Additionally, we developed a bridge to establish seamless communication between a RISC-V Central Processing Unit (CPU), utilized at Melexis, and the AHB bus. This bridge plays a vital role in enabling the CPU to be compatible with the Callisto platform, facilitating its smooth integration into the overall system.

So, summarizing the essential milestones or foundational elements of my internship in order, we have:

- Understanding the AHB protocol and comparing it to the previously used protocol.
- Creating a slave model that adheres to the AHB protocol.
- Creating a bridge to facilitate communication between a RISC-V CPU and the new Callisto Platform.
- Developing the master model following the same protocol
- Establishing an interconnect system between the slave models and the masters.

Finally, it is worth noting that the organizational structure outlined above serves as a guiding framework for the subsequent sections of this report.

Figure 2: Melexis World Map [2]

2 Advanced High Performance Bus(AHB) protocol

2.1 AHB as the Fifth Major Revision of the AMBA Protocol

In the context of computer systems and chip designs, a bus refers to a communication pathway that allows different components to exchange data and signals. It serves as a fundamental mechanism for interconnecting various Intellectual property (IP) components within a System On Chip (SOC) design. To facilitate standardized and efficient communication, bus protocols are developed. These protocols define the rules, formats, and behaviors that govern the interaction between components connected to the bus.

One such protocol is AHB within the AMBA family, developed by Advanced RISC Machines (ARM). AMBA provides specifications for on-chip communication and interconnection of IP components in SoC designs, including various bus protocols like AHB, APB, AXI, and ACE, tailored for different performance levels and system requirements.

AHB, or AMBA AHB 5 (fifth revision of AHB), is designed for high-performance on-chip communication and includes advancements and features compared to earlier versions. It offers a standardized interface for connecting IP components within a SoC, enabling efficient data transfers for high-performance communication.

The AHB protocol supports multiple bus masters and slaves, creating a flexible and scalable architecture. It accommodates single-cycle and multi-cycle data transfers, catering to different transaction types and data requirements.

In the following sections, we will explore the AHB bus protocol in-depth, including the signals used, pipeline stages involved in communication, and the unique characteristics of burst operations.

2.2 AHB Signal Definitions and Functions

Within the AHB protocol, several key features and concepts contribute to its effectiveness. One of the fundamental aspects is the presence of a well-defined set of signals. These signals serve as the communication interface between AHB masters and slaves, facilitating the exchange of control and data information.

All the signals below are well defined in the official AHB protocol document [6]. It is worth mentioning that all the signals start with the letter "h." This naming convention is significant as it signifies their association with the AHB protocol. Moreover, a Word = 32 bits, half_word = 16 bits, and byte = 8 bits.

AHB signal	Description	Values
hclk	1-bit signal: The clock input for the bus. It provides the timing reference for coordinating the operations and data transfers within the AHB protocol.	0 (Low level), 1 (High level)
hresetn	1-bit signal: An active-low reset used in the AHB protocol. It initializes the AHB system and brings it to a well-defined state by resetting the internal registers and logic, ensuring a known starting point.	0 (Reset active), 1 (Reset inactive)
hsel	1-bit signal: Input for the slave. This signal is used to indicate that the slave is selected according to the decoded address	0(not selected) 1 (selected)
htrans	2-bit signal: The type of transfer being performed within the AHB protocol. It differentiates between different types of transfers, such as nonsequential, sequential, or idle cycles.	00 (Nonsequential), 01 (Sequential), 10 (Idle), 11 (Busy)
hsize	3-bit signal: The size or width of the data transfer in the AHB protocol	000 (Byte), 001 (Half-word), 010 (Word)
hburst	4-bit signal: The type of burst transfer being performed in the AHB protocol. It defines the sequence of data transfers within a burst, such as fixed, incrementing, wrapping, or undefined bursts.	000 (Single), 001 (Increment), ... 2.4
hmastlock	1-bit signal: Indicates a locked transaction in the AHB protocol. It allows a master to gain exclusive access to a slave component for a series of transfers.	0 (Not locked), 1 (Locked)
haddress	A 32-bit signal: Carries the address information for a data transfer in the AHB protocol. It specifies the location in memory or the target register for reading from or writing to the selected slave component.	Depends on the memory address definition
hwrite	1-bit signal: Indicates the type of transfer being performed within the AHB protocol	0 (Read), 1 (Write)

hprot	4-bit signal: Provides protection information for data transfers in the AHB protocol. It includes attributes such as privileged, non-privileged, secure, or non-secure access.	Depends on the protection protocol implemented
hwstrb	4-bit signal: Specifies the active byte lanes during a Write data transfer	Depends on the intended request example: 0011 (Write bottom half word only)
hready	1-bit signal: It indicates the readiness of the slave component to accept a new request and the readiness of the master to provide a new request.	0 (Not ready), 1 (Ready)
hreadyout	1-bit signal: Serves as an acknowledgment from the slave component indicating the end of a request and the readiness of the slave to provide the output data	0 (Not ready), 1 (Ready)
hrdata	32-bit signal: Carries the data read from the slave component during a read operation in the AHB protocol	Depends on the data saved on the memory and the address
hresp	1-bit signal: Indicates the response or status of a transaction in the AHB protocol.It communicates the outcome of the data transfer	0 (Okay), 1 (Error)

Table 1: Implemented AHB signals

In addition to these signals, there are unimplemented AHB signals available as optional additions that can be incorporated at a later stage (see 7), once the Callisto platform has matured. As we are currently at the beginning of the platform's development, we have focused on implementing the signals mentioned earlier that ensure the full functionality of the bus protocol. The optional signals introduce new properties that enhance the bus's capability. These properties include Extended Memory Types, Secure Transfers, Exclusive Transfers, ...

Each property needs careful adjustments if chosen. For example, a 3-bit extension of the HPROT signal is necessary if the Extended Memory Types property is True.

These signals, along with their respective number of bits and value ranges, form the core communication infrastructure of the AHB protocol. Understanding their definitions, functions, bit

widths, and allowed values is crucial for effectively implementing and utilizing the AHB protocol in system-on-chip designs.

2.3 Pipeline architecture

One of the key reasons for the popularity of the AHB protocol is its pipelined architecture, which allows for higher frequency and efficiency. The pipelined architecture of the AMBA 5 AHB bus protocol enables concurrent processing of address and data phases, resulting in improved overall system performance. Let's explore the address and data phases in more detail and understand how the one-stage pipeline contributes to higher frequency and efficiency:

1-Address Phase: During the address phase, the master initiates the transaction by sending the control signals: the requested address (haddress), transfer size (hsize), transfer type (htrans), write indication (hwrite), protection signal (hprot), lock signal (hmastlock) and other control signals if needed. The address phase primarily focuses on establishing the target slave and the specific operation to be performed.

2-Data Phase: Once the address phase is over, the bus protocol proceeds to the data phase. In this phase, the slave responds with the requested data (hrdata), response status (hresp), and readiness indication (hreadyout). Additionally, the master can send the write data (hwdta) and write strobes (hwstrb) if it is a write operation.

Now, let's discuss how the one-stage pipeline in the AMBA 5 AHB bus protocol contributes to higher frequency and efficiency:

Bandwidth refers to the maximum rate at which data can be transferred through a communication channel or bus. It represents the capacity or the amount of data that can be transmitted within a given time frame. In the AMBA 5 AHB bus protocol context, bandwidth refers to the maximum data transfer rate the bus can support. A higher bandwidth means more data can be transferred per unit of time, resulting in faster communication and data transfer between components within a SoC.

In the case of the AMBA 5 AHB protocol, the pipelined architecture and overlapping phases allow for efficient utilization of the available bus bandwidth. By minimizing idle bus cycles and enabling concurrent processing of address and data phases, the system can initiate subsequent transactions without waiting to complete previous transactions. Therefore the protocol can operate at higher frequencies by effectively utilizing the bus bandwidth, facilitating faster data transfers and overall system performance and throughput.

Based on the figure 3, by the end of cycle 2, the first request has been completed, and simultaneously, the second request has been initiated. Furthermore, by the end of the 4th clock cycle, three requests have been finished, allowing the initiation of a fourth request. This feature greatly enhances performance compared to protocols like the Wishbone bus used in previous Melexis platforms. Without this pipeline stage, only two requests would have been completed within the same timeframe, necessitating additional clock cycles to achieve the performance level of the AHB.

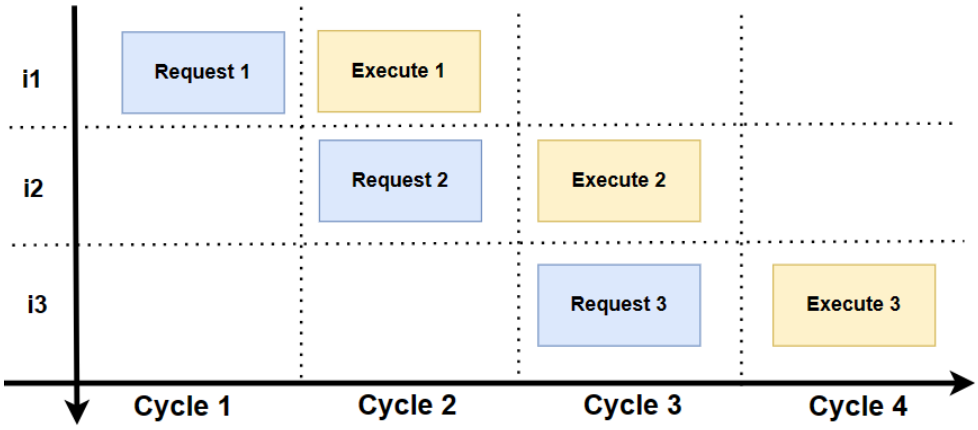


Figure 3: AHB pipelined architecture

2.4 Burst operation

The AMBA 5 AHB protocol supports burst operations, which further contributes to its attractiveness and widespread use in modern SOC designs

In the AHB protocol, a burst operation entails the capacity to convey a continuous sequence of data elements while knowing the upcoming operations from the start. Unlike executing separate read or write operations for each data item, burst transfers empower the simultaneous transfer of a data block. The burst length designates the number of data items intended for transfer within the burst itself.

Burst operations in the AHB protocol offer several benefits and advantages. Firstly, bursts empower slave units to anticipate forthcoming accesses. This anticipation enables slaves to precompute the subsequent address, initiate its retrieval, and thereby prepare for the ensuing access. This proactive approach significantly enhances the overall system's performance.

Secondly, burst transfers enable efficient movement of large data blocks or continuous data streams, maximizing the utilization of available bus bandwidth. This mainly benefits applications involving data-intensive tasks like high-speed data processing, memory transfers, or video streaming.

Lastly, burst transfers are particularly advantageous for memory access patterns that exhibit sequential or burst-like behavior. Instead of accessing memory locations one at a time, the burst operation allows for consecutive data fetches or writes, which aligns well with the underlying memory organization and cache architectures. This improves memory system efficiency and reduces access delays associated with non-sequential memory accesses.

Using the hburst signal defined in the first section, multiple types of bursts can be initiated, as explained in the table below 2.4.

hburst value	Description
0x000	Single Transfer: Indicates a single data transfer (non-burst).
0x001	Incrementing Burst of undefined length: the address for each transfer is incremented by the hsize value.
0x010	4 beat Wrapping Burst: the address wraps back to the starting address after reaching a 16-byte boundary for a 32 bits transfer size as an example: 0X48, 0X4C, 0X40, 0x44
0x011	4-beat incrementing burst
0x100	8-beat wrapping burst
0x101	8-beat incrementing burst
0x110	16-beat wrapping burst
0x111	16-beat incrementing burst

Table 2: Burst Types

In the figure below, we can see an example of a burst operation. The control signals, except the haddress, do not change, and the address is incremented automatically, respecting the burst type and the transfer size. In this example, the master requests a 4-beat incrementing burst starting from address x40 with a size of 32 bits. Since each address corresponds to 4 bytes than, the next addresses are 44, 48, and 4C, as shown in the figure below.

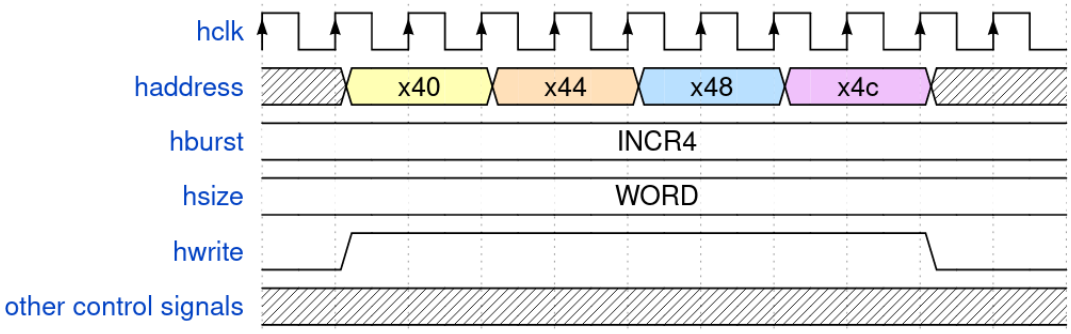


Figure 4: 4 beats incrementing burst

3 Slave Model and Implementation in the AHB Protocol

Within a bus protocol, devices are categorized as either masters or slaves. The master (manager) devices initiate and control bus transactions, while the slave (subordinate) devices respond to these transactions and provide the requested data or perform specific tasks.

In this section, we will delve into the slave model utilized in the AHB specification, highlighting its significance and explaining how it functions within the protocol. Additionally, we will explore the specific implementation employed during this internship and discuss the distinctive features incorporated into the developed slave model.

3.1 Slave Model Definition

In order to comprehend the slave model within the AHB protocol, let's start by visualizing it as a black box with inputs and outputs, as seen in the figure 5. The role of the slave, as mentioned earlier, is to receive control signals from the master as inputs and generate the appropriate outputs. Importantly, the slave model can additionally accommodate waitstates. These waitstates simulate the behavior of a physical slave that might necessitate extended clock cycles to respond effectively to the master's commands.

At Melexis, the slave entity typically corresponds to a memory module, which can be either a Read Only Memory (ROM) or a Random Access Memory (RAM). On the other hand, the master corresponds generally to a CPU or a Direct Memory Access (DMA) like an Analog to Digital Converter (ADC).

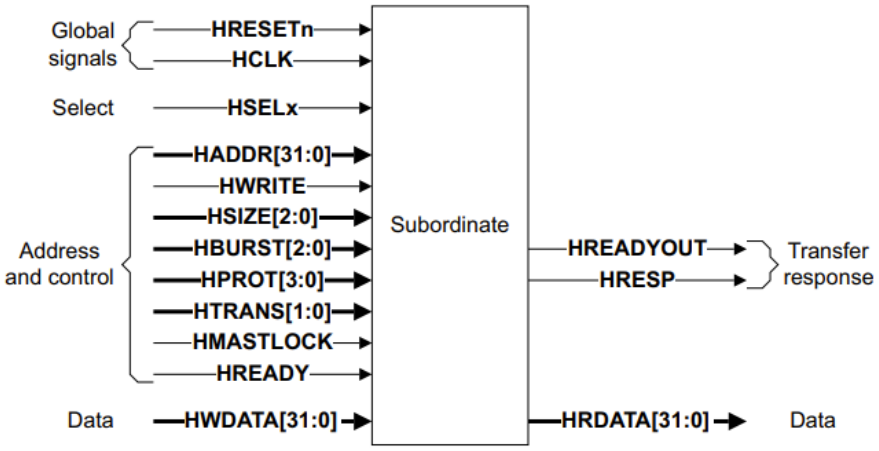


Figure 5: AHB slave model [6]

To illustrate a simple transaction, let's consider the scenario where the master initiates a read or write request to a specific memory address in the RAM. We will focus on a straightforward read operation with no waitstates for the purpose of this example.

During a read operation, the master initiates the process by sending the necessary control signals to the slave. These control signals typically include the address of the desired memory location and the command to read data. Upon receiving the read request, the slave locates the requested memory address within its internal memory space and then retrieves the corresponding data stored at the specified address. The retrieved data is then sent back as an output from the slave to the master.

The master, upon receiving the data from the slave, can further process it as needed. This concludes the simple read operation, demonstrating the basic interaction between the master and the slave in the AHB protocol. For clarity purposes, in the example shown in figure 6, we only represent some control signals sent by the master and read data and ready signal sent by the slave.

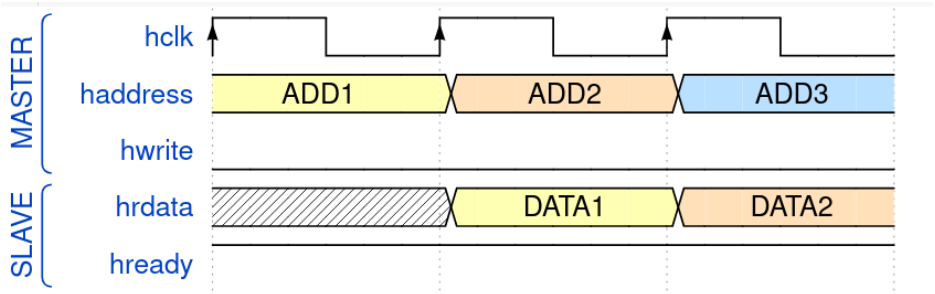


Figure 6: Straightforward Read Operations

In the depicted example, each color represents a request, and as discussed earlier, each request consists of two phases: the address phase and the data phase. Let's walk through the sequence of events:

In the initial address phase, the master commences a read operation (hwrite = 0) targeting the memory address ADD1 by transmitting requisite control signals. The model's representation of the slave, which is the memory in this context, duly receives and acknowledges these control signals. Subsequently, in the next clock cycle, the slave responds with the corresponding DATA1 for ADD1.

Simultaneously, the master proceeds to request another read operation, this time targeting the memory address ADD2. In the third clock cycle, the slave promptly responds to this second request by providing DATA2.

This process repeats continuously as the master continues to issue requests. By understanding this fundamental concept, we can proceed to explore the implementation details of the slave model and its accompanying features within the AHB bus protocol.

3.2 Implementation Details and State Machine

To implement the slave model, the Melexis team aimed to keep it generic and basic, ensuring its usability for various projects or purposes. Additional advanced functionalities will be discussed in

the subsequent subsection.

The chosen implementation for the slave model involved a Moore state machine comprising five states: IDLE, READ, WRITE, and two error states: ERROR1 and ERROR2. According to the AHB official specification document, there is no obligation that the slave must generate error states, but this was added as a feature that will be discussed further.

The IDLE state serves as the default state of the slave. Upon receiving control signals, the slave can transition to either the READ or WRITE state, depending on the nature of the request. If the request encounters an error during processing, the slave can enter the appropriate error state accordingly.

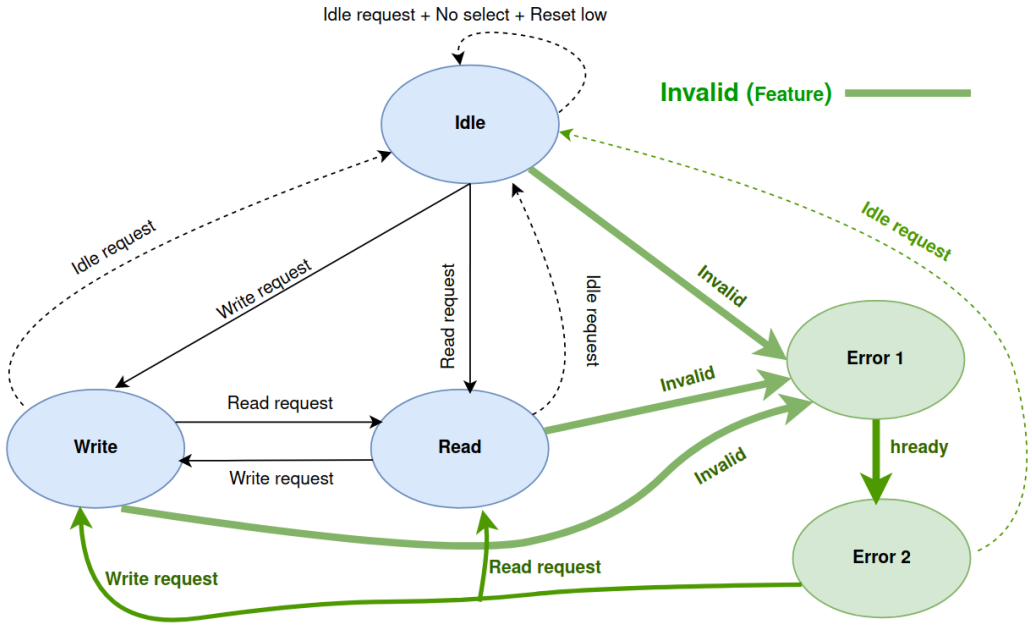


Figure 7: Slave model state machine

To gain a deeper understanding of this state machine, let's delve into its components:
The blue states in the state machine diagram represent the basic implementation of the slave, without any error handling features. In contrast, the green states, which we will discuss in the next subsection, represent the states that handle errors.

To enhance the clarity of the state machine graph, each condition is comprehensively explained in the table 3.2 provided below. This table clearly shows the conditions associated with each state transition. For the same reason, table 3.2 includes the corresponding output for every Moore machine state. In the context of the table, when an "X" is present for a particular signal, it signifies that the signal can assume any value without impacting the final outcome.

Transition	Inputs			
	htrans	**hwrite**	**Invalid**	**hselx**
Idle request	00 or 01	x	x	x
Read request	11 or 10	0	0	1
Write request	11 or 10	1	0	1
No select	x	x	x	0

Table 3: Slave state machine transitions

It is crucial to emphasize that the transition from one state to another strictly abides by the condition that the ***"hready"*** signal must be equal to one. In the event that the slave requires additional time to process the request, it can introduce wait states, temporarily setting the ***"hready-out"*** signal to zero for a specific number of clock cycles. Once the slave is prepared to provide the output response to the master, the ***"hreadyout"*** signal is reset to one; otherwise, the slave remains in the current state.

State	Outputs		
	hrdata	**hready**	**hresp**
IDLE	X	1	0
READ	Address-dependent.	waitstates == 0	0
WRITE	X	waitstates == 0	0
ERROR1	X	0	1
ERROR2	X	1	1

Table 4: Slave state machine outputs

The expression ***"waitstaes == 0"*** indicates that if the parameter for the slave, known as ***"waitstates"*** is not equal to zero, the ***"hready"*** signal will be set to zero. By utilizing a counter that progressively decreases the value of ***"waitstates"***, we can reduce it until the slave is finally ready. This approach ensures proper synchronization and allows the slave to take the necessary time to fulfill the master's request, using waitstates when required and signaling readiness by setting the ready signal accordingly.

Lastly, we conducted a comparison between our implementation and the intended slave behavior outlined in the AHB specification document [6]. In order to perform this evaluation, first, We initialized our slave model as a memory, starting at address 0x40, with a capacity of 16 WORDS. Each WORD is composed of 4 bytes, resulting in a total size of 16 * 4 BYTES. Next, we configured each memory case i with the word: "FACECAFi," where i corresponds to the case number, as depicted in the figure 8.

This memory configuration will also be used for the test benches to follow in order to verify the subsequent designed blocks and to keep the tests and simulations coherent.

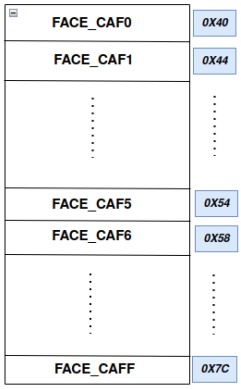


Figure 8: Slave Memory

Finally, we simulated a read request to the memory address x54, which was previously configured to store the data "FACECAF5", which requires 2 wait states. The figure below illustrates the expected behavior from the slave (on top) and the wave forms generated by our implemented slave after simulation (on bottom). Remarkably, our implemented slave demonstrates full compliance with the AHB protocol, precisely adhering to the specifications outlined in the documentation.

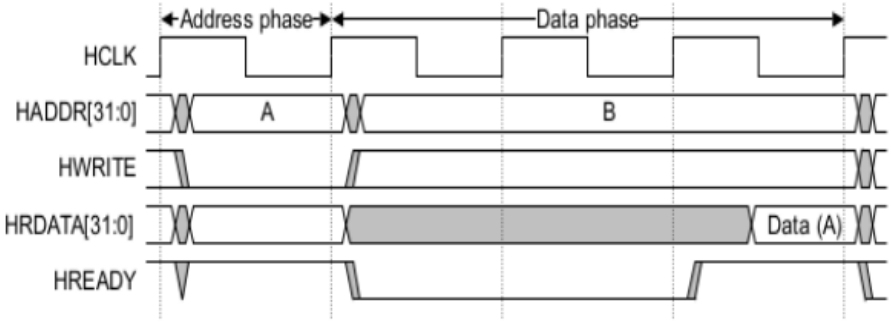


Figure 9: Expected behavior for a 2 wait states read request

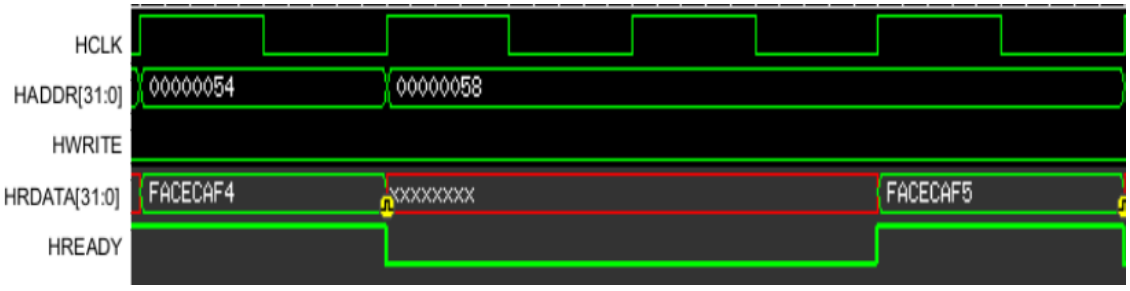


Figure 10: Simulation of the implemented slave for a 2 wait states read request

3.3 Features: Self-Checking and Invalid Response Handling

3.3.1 Error response generation

One significant feature of the slave model is its ability to handle error responses, as detailed in the specification document. Although the protocol does not mandate that the slave issues an error response for protocol violations, this feature was added to ensure the model covers all possible scenarios. To accomplish this, we introduced a signal called ***"Invalid"*** that enables the slave to enter error states. Table 3.2 of the AHB protocol outlines the two states an error response must traverse. The first state, referred to as ***"Error1"*** is characterized by ***"hready = 0"*** and ***"hresp = 1"***. The second state, ***"Error2"*** occurs when both ***hready*** and ***hresp*** are set to 1.

To provide further flexibility, the ***"invalid"*** signal in the slave model can be configured based on various options. In the Melexis model, the following options were implemented:

- "Always invalid": Setting the invalid signal to 1 guarantees that the slave consistently generates error responses for every access, regardless of the nature of the access itself.
- "Always valid": When the invalid signal is set to 0, the slave never enters error states and performs standard read and write operations without generating error responses.
- "Read-only": If a write operation is requested, the invalid signal is set to 1, restricting the slave to read-only operations, similar to a read-only memory (ROM).
- "Write-only": When a read operation is requested, the invalid signal is set to 1, allowing the slave to perform write operations exclusively.

To validate the integrity of this feature, we conducted a simulation where the invalid option was set to "Readonly" despite the request being a write operation (***hwrite = 1***) to address x40. As anticipated, the figure below exhibits the precise and expected behavior, indicating the occurrence of 2 error states, as specified in the protocol documentation.

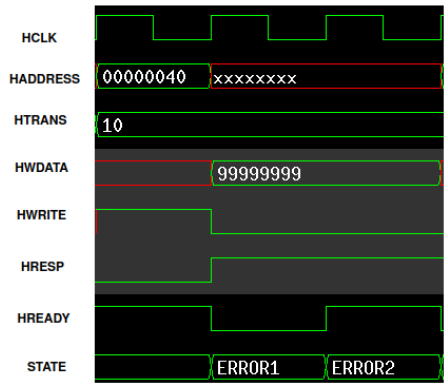


Figure 11: Implemented Slave Error response

The incorporation of a two-cycle response in the AHB bus protocol introduces a crucial mechanism that empowers the Manager to efficiently handle subsequent accesses. This response window

spanning two cycles grants the Manager sufficient time to cancel the upcoming access by transitioning ***HTRANS*** to the IDLE state before the commencement of the next transfer. While the AHB protocol does not mandate request cancellation, this feature has been added to the implemented master model to enhance its capabilities and detailed in a subsequent section.

Furthermore, in cases where the invalid feature is not activated, and the slave receives erroneous requests, such as accessing non-existent memory addresses or control signals that violate the protocol, it was discussed within the team to respond with undefined outputs. In such cases, the data returned or written would be represented as 'X,' denoting the undefined nature of this behavior.

3.3.2 Self-Checking

The implemented slave model's second feature is its self-checking capability, streamlining the verification process and ensuring proper functioning. Using SystemVerilog, multiple tasks were implemented to validate the control signals received by the slave from the master. For each master request, a corresponding slave task was created to compare the bus's control signal values with the expected ones specified in the testbench design. If the values match, the check is considered valid, indicating the slave received the intended request. Any discrepancies are easily identified during simulation execution in the terminal, eliminating the need for manual waveform inspection.

For instance, when the master initiates a read request, the ***"slave.read"*** task verifies the expected control signals. The desired result is achieved if the request transfers correctly (as shown in figure 12). Otherwise, the unexpected results are shown in red Annex B(7), and further debugging may be required by analyzing the simulation waveforms.

Incorporating self-checking capabilities into the slave model enhances the efficiency of the verification process, saving time, and facilitating the detection of potential issues or deviations from expected behavior.

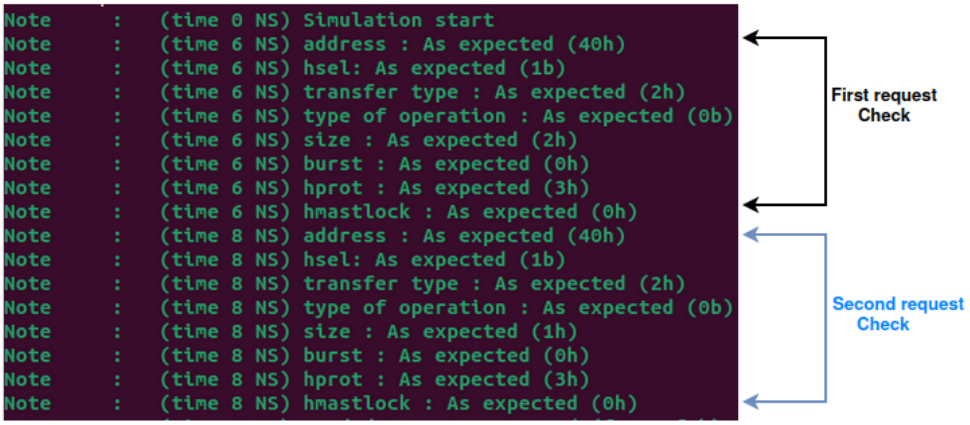


Figure 12: Slave Self Checks

4 Ibex to AHB bridge

After developing a fully functional slave model, the next step was to create a master model capable of driving input stimuli. Initially, our digital design team at Melexis had an available RISC-V CPU that could serve as the master to generate requests on the slave. However, a challenge arose as the RISC-V CPU used a different bus protocol than the AHB. To address this, we needed to develop a bridge to enable the RISC-V CPU to interact with the AHB slave model.

The following sections will introduce the IBEX CPU and its operation. Then, we will discuss the implementation of the bridge, facilitating communication between the IBEX CPU and the AHB slave model. Lastly, we will present the results obtained from this interconnected communication system.

4.1 RISC-V IBEX CPU

The IBEX CPU is an open-source, 32-bit RISC-V processor written in SystemVerilog and developed by the lowRISC project. It aims to provide a high-performance and efficient processor core for various applications [1]. The IBEX CPU is designed with a focus on simplicity, modularity, and configurability, allowing users to tailor it to their specific needs.

One notable aspect of the IBEX CPU is its compliance with the RISC-V open instruction set architecture (ISA). It is highly configurable, allowing users to choose the desired features and optimizations for their specific use cases. This is why the Melexis digital team decided to use it in future projects.

This processor incorporates essential components, including 32 registers for data storage and operations, a fetch stage for instruction retrieval, an Arithmetic Logic Unit (ALU), a memory unit for data transfers with external memory, a branch unit for evaluating branch instructions, and a two-stage pipeline (fetch and execute stages) for efficient instruction execution. These components contribute to the IBEX CPU's ability to execute RISC-V instructions accurately and efficiently.

The IBEX CPU is classified as a Harvard architecture type CPU, which means it consists of two distinct groups of buses: an instruction bus for fetching instructions (as an example from the ROM) and a data bus for retrieving data (as an example from the RAM). A comprehensive description of the IBEX protocol can be found in the following reference [1]. However, the protocol itself is not the main focus for our purposes. What is of interest is how to integrate and link this protocol with the AHB protocol. The subsequent section aims to outline the implementation details of achieving this integration, bridging the communication between the IBEX protocol and the AHB protocol.

In order to enable the usage of the IBEX on the AHB bus for other designers, integration of the bridge with the IBEX core is essential. This integration creates a shell that can be readily employed by designers for their specific requirements. The figure 13 illustrates the architectural setup depicting the integration of the IBEX core with the bridge on the AHB bus.

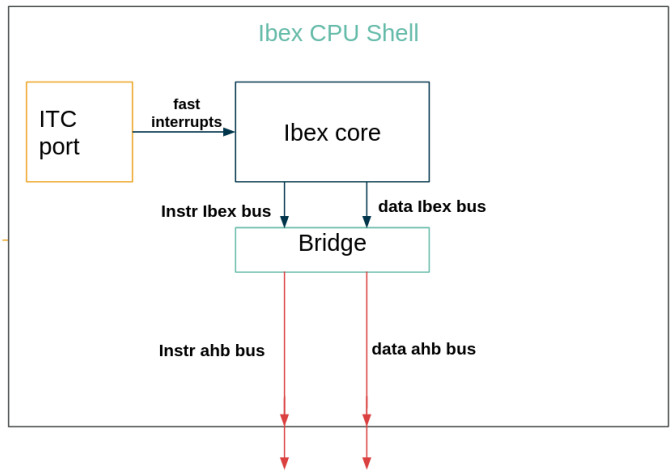


Figure 13: Ibex Shell

4.2 Bridge implementation

A comprehensive understanding of both protocols is crucial to establishing a robust communication bridge between the IBEX and AHB protocols. This understanding serves as the foundation for initiating the link between them as shown in fig 14. The table 4.2 outlines the correspondence between each signal type in the IBEX protocol and its equivalent in the AHB protocol, facilitating the mapping process between the two.

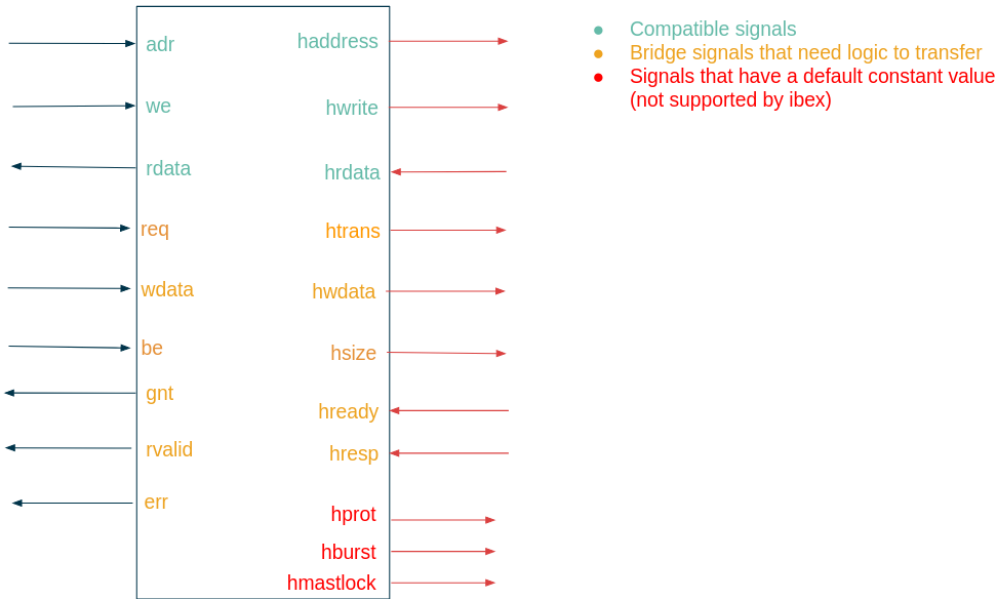


Figure 14: Ibex to AHB bridge

During the mapping process, it is important to note that certain signals, such as address and rdata, remain unchanged as they have similar definitions and functions in both the IBEX and AHB

protocols. However, additional logic is required for other signals to enable their smooth transfer from one bus protocol to the other.

In this table, the numbers are represented in Verilog Standard [4]. As an example, when we use "4'b1100," it conveys that the data is encapsulated within 4 bits, and the corresponding binary value is "1100."

Bridge Outputs	**Assignment**
haddress	adr.
hwrite	we
htrans	req and 2'b10
hwdata	Instruction bus : X | Data bus : gnt ? wdata: hwdata
hsize	case statement (see below 4.2)
hprot	4'b0011 (Data bus) | 4'0010 (Instruction bus)
hburst	4'b0000 (Ibex not capable of doing Bursts)
hmastlock	0 (Ibex not capable of locking transfers)
hrdata	rdata
err	hready ? hresp : x
rvalid	hready and req

Table 5: Bridge Signals Mapping

- When be is 4'b1111, the value of hsize is set to 3'b010. When be is 4'b1100, 4'b0011, or 4'b0010, the value of hsize is set to 3'b001. When be is 4'b0001, 4'b0100, or 4'b1000, the value of hsize is set to 3'b000; otherwise, hsize= 3'b010, meaning a 32 bits as default transfer size.
- The expression ***"gnt ? wdata : hwdata"*** indicates that when ***"gnt"*** equals 1, the value of ***"hwdata"*** will be updated to the value of ***"wdata"*** and if ***"gnt"*** is not equal to 1, ***"hwdata"*** will retain its current value unchanged.
- the expression ***"hready ? hresp : x"*** means if ***"hready"*** is true, ***"hresp"*** will take the value of ***"hresp"*** itself. If ***"hready"*** is false, ***"hresp"*** will be set to the value of "x.

4.3 Bridge testing and Results

To ensure the proper functioning of the IBEX CPU, communication with it is established through the RISC-V assembly language, as defined in the official Instruction Set Architecture (ISA) [7]. By writing requests using the assembly language, the compiler transforms them into data that is stored in the ROM. Subsequently, the IBEX CPU fetches and executes these stored instructions sequentially.

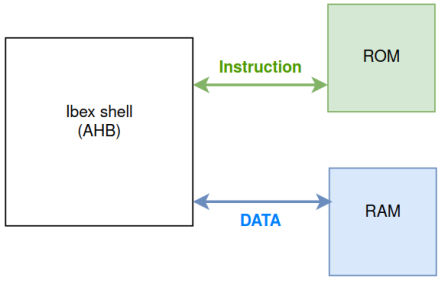


Figure 15: Ibex to AHB bridge testbench

A comprehensive set of requests in assembly language was compiled and stored in the ROM to validate the bridge implementation. The requests covered various scenarios, such as byte, half-word, and word store and load operations (Table 6 shows a subset of these requests). The testbench included instantiated slave models to simulate the ROM and RAM, allowing IBEX to perform its operations. Further details on the testbench configuration are explained in fig 15

Assembly language	**Intended Request**
lbu x2, 0x40(x0)	Load byte from memory address 0x40 to register x2
lhu x2, 0x48(x0)	Load half word from memory address 0x48 to register x2
sw x2, 0x4C(x0)	Store word found in register x2 to memory address 0x4C

Table 6: Assembly Language Requests

The CPU fetches requests previously stored on the ROM and executes them on the RAM. Through self-checking in the slave model and additional tests in the test bench, we simulate the bridge and verify if the observed behavior aligns with the expected behavior from the master's requests. The accompanying figure validates the successful execution of all requests, confirming accurate data reading and writing.

```
Note     :   (time 20 NS) first byte load: As expected (0b)
Note     :   (time 34 NS) second byte load: As expected (0b)
Note     :   (time 48 NS) third byte load: As expected (0b)
Note     :   (time 54 NS) first store operation with 3 bytes written: As expected (fa998877h)
Note     :   (time 66 NS) First half word load: As expected (0b)
Note     :   (time 82 NS) Second half word load: As expected (0b)
Note     :   (time 88 NS) second store operation with 2 half words written: As expected (aa998877h)
Note     :   (time 100 NS) word load: As expected (0b)
Note     :   (time 106 NS) third store operation with one word written: As expected (aa998877h)
Note     :   (time 116 NS) fourth store operation with one word written: As expected (ffffffffh)
Note     :   (time 133 NS) An error has been detected , next request is 0x100 : interrupt controller: As expected (100h)
Note     :   (time 153 NS) An error has been detected , next request is 0x100 : interrupt controller: As expected (100h)
Note     :   (time 173 NS) An error has been detected , next request is 0x100 : interrupt controller: As expected (100h)
Note     :   (time 193 NS) An error has been detected , next request is 0x100 : interrupt controller: As expected (100h)
Note     :   (time 304 NS) Simulation completed successfully
```

Figure 16: Ibex to AHB bridge simulation output

5 Master Model and Implementation in the AHB Protocol

As previously mentioned, the master, also known as the manager, is responsible for initiating and controlling bus transactions to which the slaves respond. This section explores the significance of the master model in facilitating efficient data exchanges. Additionally, we will discuss the specific implementation used during this internship, highlighting the unique features integrated into the master device for seamless communication across the AHB bus.

5.1 Master Model Definition

In the master model, we can envision it as the counterpart of the slave model. It can be visualized as a block that generates outputs representing the request information, which will serve as inputs for the slave. On the other hand, the master model's inputs correspond to the slave outputs, which contain the result information of the request.

To illustrate this relationship clearly, the figure below visually represents the master model with distinctly defined inputs and outputs.

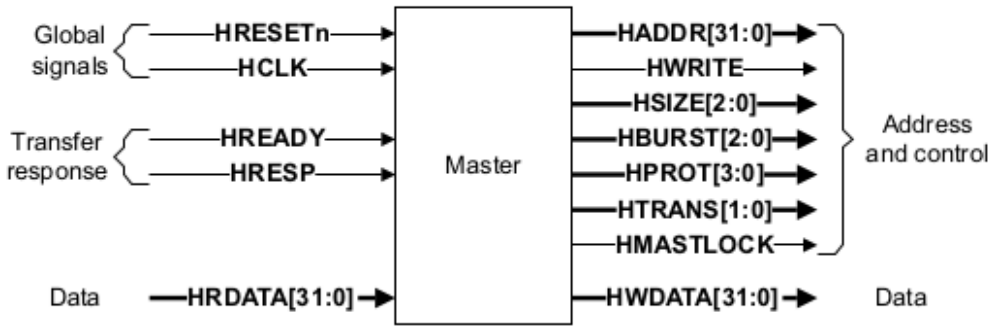


Figure 17: AHB Master Model[6]

The model must effectively synchronize the control signals, such as address, size, and hwrite, during the address phase while managing the data signals, including write data signal and strobe signals, to be transmitted during the data phase. Moreover, the model must adhere to the principle that no new request can be sent until the previous one has been received and processed by the slave.

To illustrate this synchronization, the figure below depicts how a write operation should be structured to ensure compliance with the AHB protocol. Only the important signals are shown for clarity.

In figure 18, we observe that the first request is directed to address A, incorporating all the control signals required during the address phase. Subsequently, the data phase commences in the next clock cycle, specifying the written data. Remarkably, during this data phase, it is possible to initiate another request simultaneously directed to address B. This subsequent request can

involve various operations, such as read, write, burst, or even an idle state, offering flexibility and versatility to the user's interactions with the system.

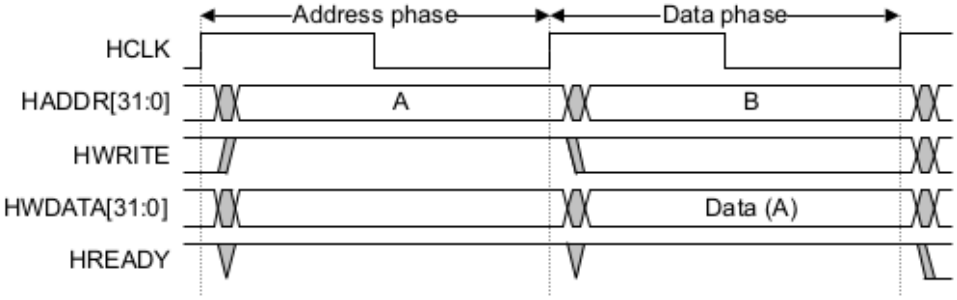


Figure 18: Write transfer

5.2 Implementation

The primary objective of this model is to facilitate the transfer of request information from the user to the slave, including the desired read and write addresses, along with their corresponding control signals. To achieve this, we leverage a feature provided by the SystemVerilog language called Tasks. These tasks function similarly to functions, taking arguments representing control signals, and when invoked, they drive the relevant signals to stimulate the slave. Consequently, the master consists of a collection of tasks that users can call to communicate their desired requests to the slave effectively.

For the user's convenience, a task has been created for each possible request:

- Tasks for reading from the slave with sizes of 8 bits, 16 bits, and 32 bits.
- Tasks for writing to the slave with sizes of 8 bits, 16 bits, and 32 bits.
- Tasks to generate an IDLE state in the slave.
- Tasks to create Bursts: All possible bursts specified in Table 2.4 can be requested, including incremental, wrapping, or with undefined lengths.

By employing these well-defined tasks, users can seamlessly interact with the slave model, allowing for smooth and versatile communication between the master and the slave for various read, write, and burst operations.

However, creating the tasks alone is insufficient, as the model must adhere to the AHB protocol by accurately driving the data phase signals at the appropriate time after the address phase concludes. Furthermore, the tasks cannot be executed independently; they must align with the master's readiness to initiate a new request. Consequently, a parallel logic component was meticulously designed alongside the task definitions to ensure a correct master model that remains compatible with the AHB protocol.

To thoroughly validate our model and ensure its compliance with the theoretical specifications, comprehensive test cases were crafted. These test cases encompass all possible conditions, including WRITE, READ, and BURST operations with varying sizes, such as BYTE, HALF WORD, and WORD, along with different wait states. As part of this confirmation process, we present an example of a simple write operation targeting the memory address x44 to illustrate the procedure 19.

In the testbench, the user initiates the task: `master.Write(.address[0x40], .size[0x010], .htrans[2'b10] , .hwrite[1], .hmastlock[1] , .hburst[3'b000] , .hprot[4'b0011], .hwdata[32'h0000] , .hwstrb[4'b1111])` . Upon analyzing the waveforms simulation, we observe that the master accurately provides the correct control signals and data signals (shown in blue). This successful outcome confirms the model's accuracy.

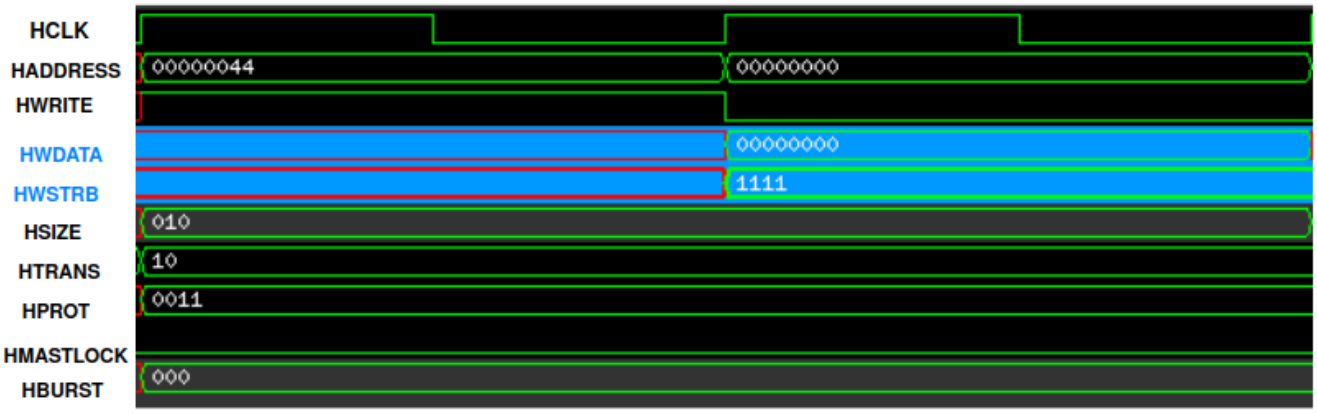


Figure 19: Implemented Master Write request to address x44

By conducting these rigorous tests covering a wide range of scenarios, we ensure the reliability and robustness of our master model, providing confidence in its practical implementation and adherence to the AHB protocol.

5.3 Features

5.3.1 Cancel access on error

One of the notable enhancements made to the master is its capability to cancel a future request upon receiving an error during the previous request. While not mandated in the AHB official specification document, this feature proves to be highly valuable, preventing the continuation of a sequence of requests if one of them encounters an error. Implementing this functionality allows the master to drive ***"HTRANS"*** to IDLE state, signifying that a request has been halted due to a previous error, effectively transitioning the slave to an IDLE state as well.

To enable this option, a parameter called **Cancel-access-on-error** was introduced. When this parameter is set to 1, the cancellation feature can be executed, and the transfer is terminated, as

illustrated in the figure below. This added flexibility provides greater control over the communication process, ensuring that any erroneous request does not propagate further, leading to a more robust and reliable system

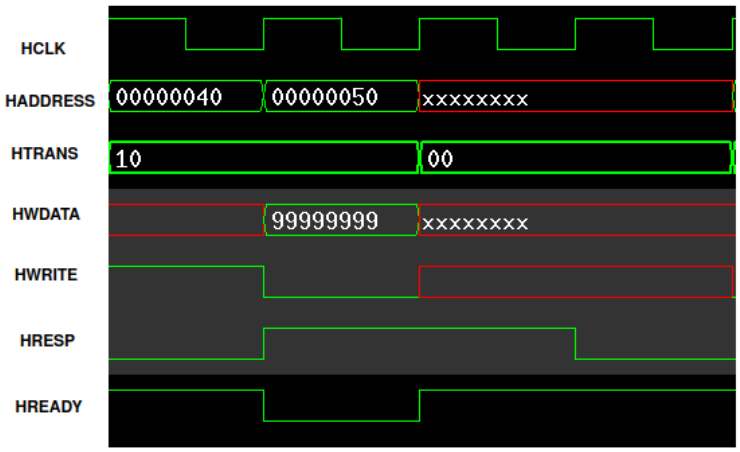


Figure 20: Cancel Access on Error Feature

To achieve this result, the error slave feature was activated by setting the option to alwaysinvalid, which ensures that the slave always generates an error response. Subsequently, the master sent two successive requests: a write operation with data 99999999 to address x40 and a read operation to address x50.

As anticipated, upon accepting the control signals from the master, the slave responded with an error, transitioning to the first state ERROR1 and setting ***"HRESP=1"*** and ***"HREADY=0"***. Due to the active cancel on access error feature, the master detected the error response ***"HRESP=1"*** and promptly canceled the pending read request to address x50, even though it had started before. Next, the master sets ***"HTRANS"*** to IDLE, signifying the end of the canceled transfer. The slave enters in the second state ERROR2, characterized by ***"HREADY"*** and ***"HRESP"*** both being set to 1.

5.3.2 Self Checking

Another valuable addition to our master model is the capability to perform self-checks on its inputs, specifically the read data (RDATA) and the validity of the request (HRESP). Similar to the feature added to our slave model, this enhancement verifies the correctness of the design without having to look at the waveforms. By incorporating additional inputs, namely ***"expected_rdata"*** and ***"expected_valid"***, to the previously defined master tasks, users can now call the appropriate task corresponding to their intended request while providing the expected data that will be read in the case of a read operation and specifying whether the request is expected to encounter an error or not.

To better illustrate this feature, let's consider a word read request targeting address x48, which

was previously configured to contain the data FACECAF2 and the same for address 0x4C containing the data FACECAF3 as explained in figure 8. When creating the testbench, the user calls the task

- `master.read(0x48, .htrans[10], .expectedrdata[FACECAF2], .expectedvalid[1],...)`
- `master.read(0x4C, .htrans[10], .expectedrdata[FACECAF3], .expectedvalid[1],...)`

The result, as depicted in the figure below, combines the self-checking features of both the slave and the master. We can observe that the master indeed reads the correct value as expected during both requests , demonstrating the effectiveness of this self-validation mechanism.

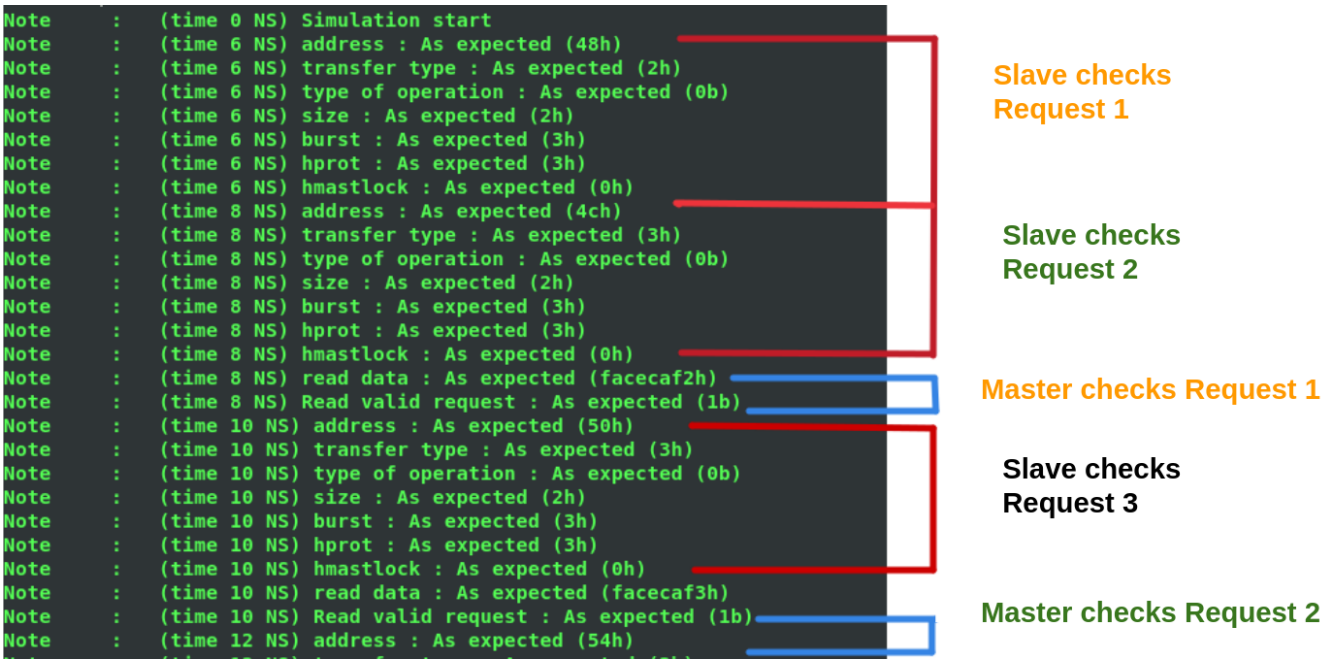


Figure 21: Combined Self Checks feature of the implemented master and slave model

By leveraging the self-checking features of both the master and slave models, we can thoroughly validate the correctness of our design without the need to investigate waveforms for each simulation. This significantly saves time and enhances efficiency for the designer.

6 Interconnect System

With verified AHB protocol-compliant master and slave models, we now focus on the most challenging part of the internship: creating the interconnect system. Our workflow is structured into four main parts, aiming to establish robust and efficient communication pathways for smooth data flow and effective collaboration between various masters and slaves.

During this internship stage, synthesizable designs are crucial. In fact, the slave and master components are non-synthesizable models, relying on Tasks and while loops, which are not synthesis tolerant. Ensuring synthesizability for the interconnect system blocks and the previously implemented bridge becomes paramount as we progress. We meticulously adhere to synthesis tool guidelines while writing RTL code, ensuring compatibility with the hardware description language and target hardware architecture.

The final RTL code undergoes a LINT test to ensure the design's physical implementation viability. This thorough examination assesses compliance with synthesis constraints and verifies effective synthesis for a physical chip. All digital blocks designed in the following section have passed the lint test, confirming that the coding guidelines have been respected.

6.1 Single Master Interconnect

In the Single Master Interconnect, our primary objective is to enable seamless communication between a single master and multiple slaves. We introduced two key components to achieve this goal: an **address decoder** and a **multiplexer** whose mechanism is detailed in the spec document.

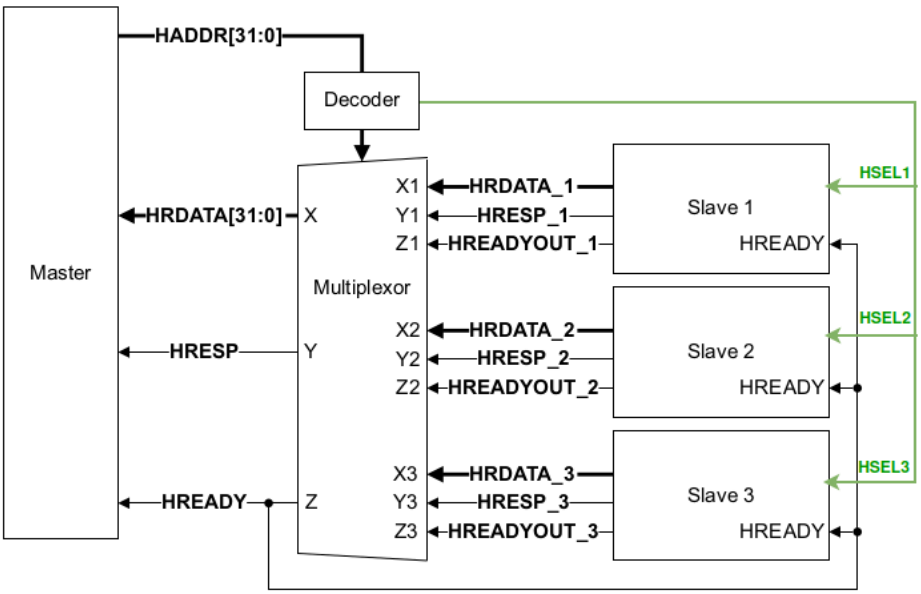


Figure 22: Single Master Interconnect

The figure 22 illustrates the configuration of the single master interconnect, showcasing both the address decoder and the multiplexer and their interconnections.

6.1.1 Address Decoder

The address decoder plays a critical role in the system by precisely interpreting the master's address signals and determining the target slave for the request. This decoding enables the system to accurately route the request to the designated slave module, ensuring seamless and accurate data exchanges.

Each slave possesses a specific address range which should be a multiple of 1 Kilobyte (Kb) blocks, and the address decoder uses the ***"HSEL"*** signal to selectively set it to 1 only for the intended slave module. Simultaneously, the other slave modules receive ***"HSEL=0"***, placing them in an IDLE state.

The design features a single input address that generates the ***"HSEL"*** signal for each slave. The ***"HSEL"*** signal is one hot decoded, indicating that for each request, only one bit is set to 1, corresponding to the selected slave (see fig 22). For instance, in a system with 3 slaves, the ***"HSEL"*** is a 3-bit signal, where the first bit represents the first slave, the second bit represents the second slave, and the last bit represents the third slave. So, if ***"HSEL = 010"***, the second slave was selected.

This intelligent routing mechanism ensures precise delivery of a master's request to the appropriate slave while preventing the slave from responding to requests not intended for it. This optimization enhances resource utilization and streamlines communication pathways.

6.1.2 Multiplexor

Working in conjunction with the address decoder, the multiplexer plays a vital role as an intermediary, facilitating efficient communication between the master and the selected slave module.

Each slave module has three outputs: ***HREADYOUT, HRDATA, and HRESP***. The multiplexer takes the selected slave from the address decoder as input, allowing only the chosen slave to transfer its outputs to the master. This selection ensures that the master receives data and responses solely from the intended slave, preventing interference and streamlining communication.

Additionally, the multiplexer generates the ***"HREADY"*** input signal for the slaves and the master, adhering to the AHB protocol's principle that no slave can be ready for a new request until the previous transfer is completed. This synchronization ensures precise coordination between the master and selected slave modules, meeting protocol requirements.

To handle the outputs from the slave, the multiplexer takes the one hot decoded signal from the decoder as input, identifying the active channel between the master and slaves. It addresses timing mismatches between the ***"HSEL"*** signal (address phase) and the slave outputs (data phase), using flip-flops to delay the ***"HSEL"*** signal and synchronize it with the slave outputs. The final ***"HREADY"*** signal is obtained by choosing the selected slave ***"HREADYOUT"*** signal, ensuring no new request is accepted until the previous one is completed. Figure 23 explains the topology of this Multiplexor.

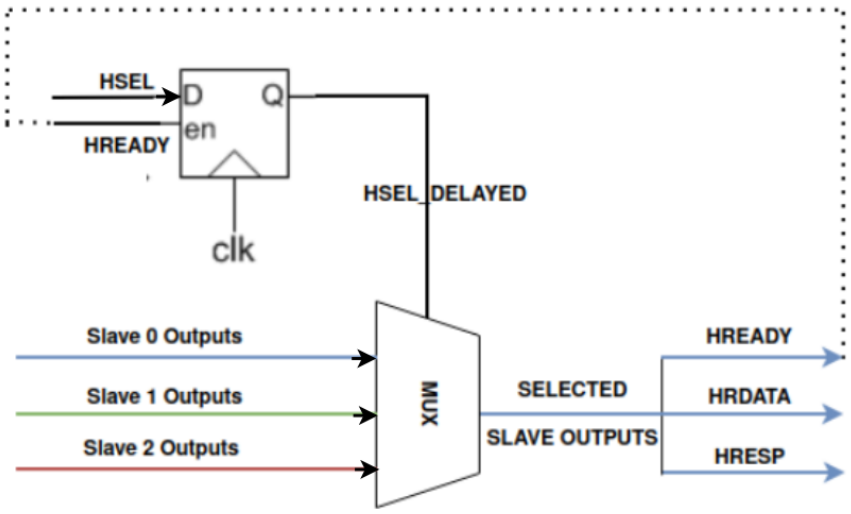


Figure 23: Multiplexor Topology

6.1.3 Testing

We used the previously developed models to validate our implementation as part of our comprehensive testbench. The testbench was designed very similar to fig 22 to accommodate one master, two slaves, and our single master interconnect system. Throughout the testing process, multiple requests were initiated on both slaves, while we also incorporated a check to ensure the correct slave was selected for each request.

Leveraging the self-checking capabilities of both models, we analyzed the test results. The outcome was promising, as the testbench successfully demonstrated that all requests were executed flawlessly. The interconnect system played a vital role in mediating the transfer between the master and slaves, efficiently managing data flow and ensuring smooth communication.

```
Note     :   (time 0 NS) Simulation start
Note     :   (time 4 NS) slave 1 is not selected : As expected (0b) ←
Note     :   (time 6 NS) address : As expected (48h)
Note     :   (time 6 NS) hsel: As expected (1b)
Note     :   (time 6 NS) transfer type : As expected (2h)
Note     :   (time 6 NS) type of operation : As expected (0b)
Note     :   (time 6 NS) size : As expected (2h)
Note     :   (time 6 NS) burst : As expected (0h)
Note     :   (time 6 NS) hprot : As expected (3h)
Note     :   (time 6 NS) hmastlock : As expected (0h)
Note     :   (time 6 NS) slave 1 is not selected : As expected (0b) ←
Note     :   (time 10 NS) read data : As expected (ffffffffh)
Note     :   (time 10 NS) Read valid request : As expected (1b)
Note     :   (time 10 NS) slave 0 is not selected : As expected (0b) ←
Note     :   (time 10 NS) address : As expected (90h)
Note     :   (time 10 NS) hsel: As expected (1b)
```

Figure 24: Single Master Interconnect Testbench Results

NOTE: If the master requests an address that is not supported by any of the slaves, the protocol requires sending an error response. To achieve this, the protocol mandates the use of a default slave that becomes active only when the master initiates such a request with a non-existent address. The design for this default slave is straightforward, as it closely resembles the previously designed slave model, but this time it is specialized to handle error responses exclusively when selected.

6.2 Single Slave Interconnect

In this subsection, we delve into the design of the single slave interconnect, which serves as the counterpart to the previously implemented interconnect. However, the primary objective here is to enable multiple masters to communicate with a single slave seamlessly.

To achieve this, we introduce three essential components: an **Input Stage**, an **Arbiter**, and a **Demultiplexer**.

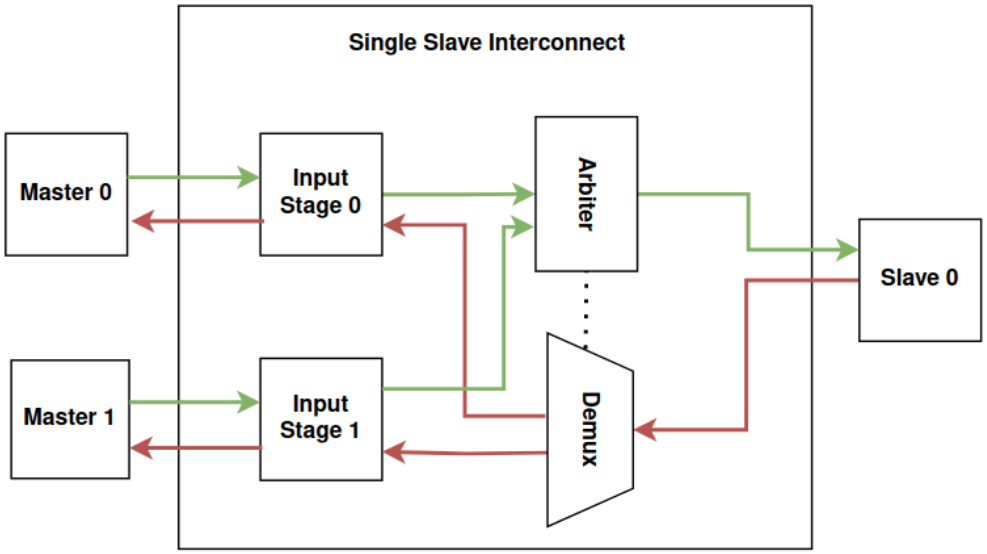


Figure 25: Single Slave Interconnect

The Green arrows illustrate the data flow of the request information, including ***HADDRESS, HTRANS, HWRITE ...*** On the other hand, the Red arrows depict the data flow of the slave outputs, representing the response to the request: ***HREADYOUT, HRDATA and HRESP***. To ensure clarity, this representation is simplified, omitting numerous internal signals between components.

6.2.1 Input Stage

As the AHB protocol mandates, we must incorporate an additional block, the INPUT stage. This stage is of utmost importance as it serves two key purposes. Primarily, it maintains the request information from masters denied immediate bus access, safeguarding their requests for subsequent processing upon gaining access. This preservation is essential due to the protocol's prohibition against slaves introducing wait states during the address phase, which would otherwise cause these requests to vanish – a scenario we must avoid. This feature ensures that no data or

transaction requests are lost during communication.

Secondly, the INPUT stage acts as a safeguard, prohibiting masters from receiving the slave's output unless they have initiated a granted transfer with the slave previously. This mechanism helps maintain data integrity and prevents unauthorized access to the slave's outputs.

To design this block, the arbiter sends a 1-bit ***"Gnt"*** signal input to indicate whether the master is granted access or not. If the master is granted access, its outputs are directly sent to the arbiter. However, if the master's access is not granted and it has initiated a request, all the relevant request information is stored in a series of FlipFlops. The First MUX plays a critical role in selecting the appropriate path based on the request status, as illustrated in Figure 26.

Based on the same logic, when the master is granted access, the second MUX allows the outputs from the slave to directly reach the master. However, if access is not granted, only the default outputs are sent, representing a valid request with no wait states and irrelevant read data: ***"HRESP = 0"***, ***"HREADY = 1"***, and ***"HRDATA = XX"***. These inputs are provided to the master when no request is demanded.

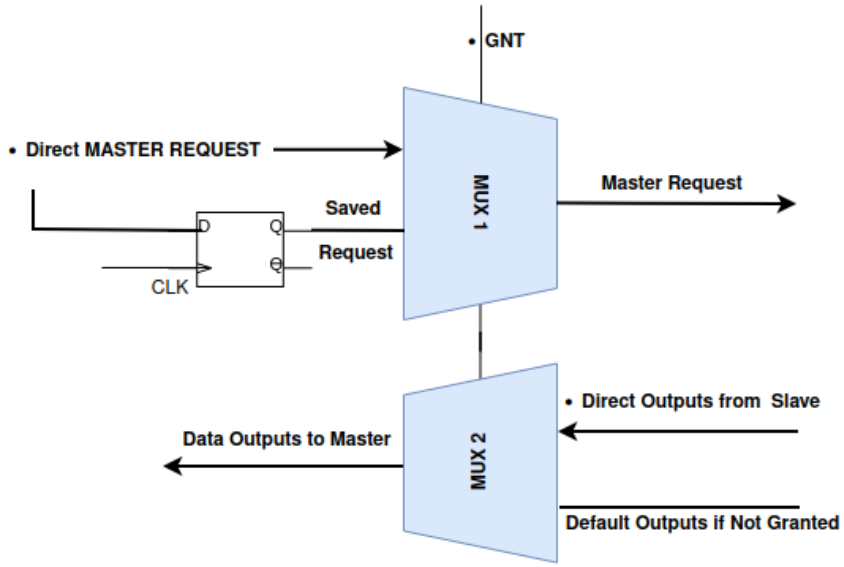


Figure 26: Input Stage Topology

6.2.2 Arbiter

This arbitration process ensures fair access and orderly communication between the masters and the slave. In this arbitration process, masters are assigned priorities, and the one with the highest priority gets constant access to the bus. If the highest priority master has no pending requests, the arbiter moves on to the master with the next highest priority, granting it access to the bus. This prioritization mechanism continues for subsequent masters based on their assigned priorities

[3].

To design this block, we adopted a priority scheme based on the decreasing order of the connected masters. For instance, if we have three masters - Master 0, Master 1, and Master 2 - Master 0 holds the highest priority.

To determine whether a master has initiated a request or not, we rely on the ***"HTRANS"*** signal as an indicator. When ***"HTRANS"*** is busy or idle, it signifies the master is not requesting a transaction. In particular, if the first bit of ***"HTRANS"*** is 0, the master is considered inactive, indicating no current request from that master. By utilizing ***"HTRANS"*** and the priority order, the process of selecting the master with the highest priority among the active masters becomes straightforward.

Additionally, we integrated the ***"GNT"*** (Grant) signal, which plays a crucial role in informing the input stage about which master is granted access. The GNT signal is one hot decoded, indicating the granted master's identity. For example, in a system with three masters, if ***"GNT = 100"***, it means that Master 0 has been granted access while the other two masters have not.

By utilizing ***"HTRANS"***, the priority order, and ***"GNT"*** signal, we have created a streamlined and efficient process for selecting and granting access to masters on the AHB bus. This implementation ensures effective communication and data transfer between the masters and the interconnect system, enhancing the overall performance and reliability of the system

6.2.3 Demultiplexor

The demultiplexer plays a crucial role in the system by receiving the slave's response data to the request (***"HRDATA"***), along with ***"HREADY"*** and ***"HRESP"*** signals, and directing to the specific master that initiated the request. For other inactive masters, it distributes default outputs, which are ***"HREADY = 1"***, ***"HRESP = 0"***, and ***"HRDATA = X"***. Essentially, the demux serves as a data distributor, guiding the output data through the appropriate channel to the corresponding master based on the control signal ***"sel_data"*** .

This ***"sel_data"*** signal, sent by the arbiter, serves as an indicator of the winning master in the arbiter battle, determining the recipient of the request-response. The demultiplexer ensures efficient communication between the masters and the slave, facilitating data exchange based on the outcome of the arbiter's decision

In the figure below, the green arrows depict the data flow of the request information, while the red arrows represent the data flow of the request output. The internal signal ***"sel_data"*** acts as a link between the arbiter and demux, facilitating communication. Lastly, the ***"GNT"*** signal is transmitted to the input stage, as explained earlier.

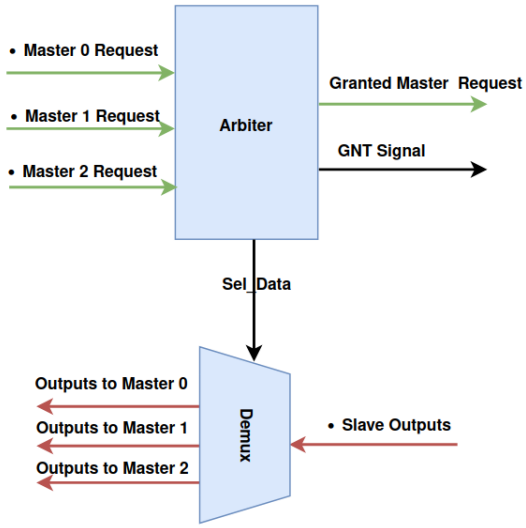


Figure 27: Arbiter and Demux Topology

6.2.4 Testing

To verify our implementation, a testbench was created similar to fig 25, incorporating 2 masters, 1 slave, and the single slave interconnect block. Multiple operations were performed on the slave from both masters, and an additional check was introduced to validate the expected master that initiated each request (blue arrows in simulation output).

By leveraging the self-checking capabilities, we successfully verified that the design effectively manages data flow between the masters and the slave as intended. This validation process confirms the interconnect's reliable and efficient functionality.

```
Note      :   (time 0 NS) Simulation start
Note      :   (time 6 NS) The correct master is chosen: As expected (2h)
Note      :   (time 6 NS) address : As expected (80h)
Note      :   (time 6 NS) hsel: As expected (1b)
Note      :   (time 6 NS) transfer type : As expected (2h)
Note      :   (time 6 NS) type of operation : As expected (0b)
Note      :   (time 6 NS) size : As expected (2h)
Note      :   (time 6 NS) burst : As expected (0h)
Note      :   (time 6 NS) hprot : As expected (3h)
Note      :   (time 6 NS) hmastlock : As expected (0h)
Note      :   (time 10 NS) The correct master is chosen: As expected (0h)
Note      :   (time 10 NS) read data : As expected (facecb00h)
Note      :   (time 10 NS) Read valid request : As expected (1b)
Note      :   (time 10 NS) address : As expected (40h)
Note      :   (time 10 NS) hsel: As expected (1b)
Note      :   (time 10 NS) transfer type : As expected (2h)
Note      :   (time 10 NS) type of operation : As expected (1b)
Note      :   (time 10 NS) size : As expected (2h)
Note      :   (time 10 NS) burst : As expected (0h)
Note      :   (time 10 NS) hprot : As expected (3h)
Note      :   (time 10 NS) hmastlock : As expected (0h)
Note      :   (time 14 NS) The correct master is chosen: As expected (0h)
```

Figure 28: Single Slave Interconnect Testbench Results

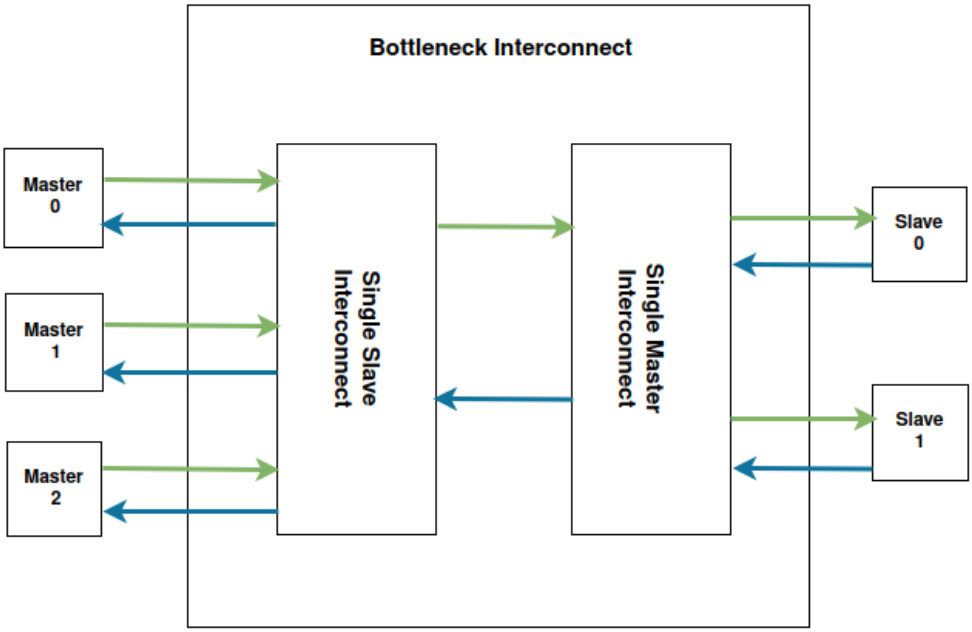


Figure 29: Bottleneck Configuration

6.3 Bottleneck Interconnect

6.3.1 Architecture

The Bottleneck Interconnect combines the previously implemented single slave interconnect and single master interconnect. It enables communication between multiple masters and multiple slaves. The design process was rather straightforward and involved connecting the ports of the single slave interconnect to those of the single master interconnect, as depicted in the figure above 29.

During the design process of the interconnect blocks, a key priority was to ensure simplicity and user-friendliness for other designers. To achieve this, we implemented a flexible approach where the user can easily specify the number of masters and slaves required in the bus configuration. Leveraging the **"generate"** feature offered by the SystemVerilog language, the design automatically generates all the necessary blocks to accommodate the requested configuration.

This dynamic generation of blocks based on the provided parameters offers unparalleled flexibility and adaptability. It allows our design to be easily utilized in various projects and scenarios without requiring manual adjustments or complex modifications. Designers can now seamlessly incorporate our interconnect solution into their projects, tailored to their specific requirements, thereby saving time and effort in the development process.

This interconnect type, unfortunately, comes with a significant limitation: it allows only one active communication channel at a time, preventing simultaneous communication between two masters and two slaves. This constraint significantly hampers the performance of Harvard ar-

chitecture CPUs. To surmount this limitation and achieve enhanced performance, an alternative configuration known as the Crossbar has been developed, which will be further explored in the final section.

6.3.2 Testing

To thoroughly test our design, we developed a comprehensive testbench featuring 3 masters, 2 slaves, and the bottleneck interconnect just as detailed in fig 29. Multiple requests were initiated from the various masters to the different slaves, and similar verification strategies, as used in the previous interconnects, were employed.

The test results were highly promising, as illustrated in the figure below. It confirms that all requests were successfully executed, showcasing the interconnect's capability to efficiently manage and transfer multiple data transfers within the AHB while adhering to its protocol.

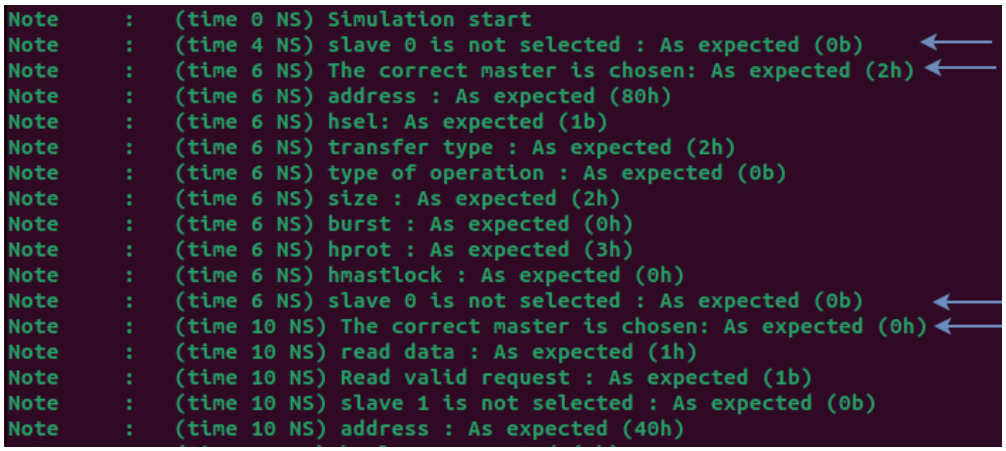


Figure 30: Bottleneck Testbench Results

6.4 Crossbar

6.4.1 Architecture

The Crossbar Architecture is designed to facilitate **simultaneous** communication between multiple masters and multiple slaves. To achieve this, we modified and managed the previously designed blocks, ensuring compatibility with this configuration. Additionally, we introduced new blocks to enable seamless communication within the bus.

3 main blocks were used to design the crossbar: the **Input stage**, previously implemented, an **Arbiter stage**, and A **Decoding stage** as seen in figure 31.

In this system, a layer corresponds to a single master communicating with multiple slaves [5]. Since the crossbar's purpose is to enable simultaneous communication between many masters and many slaves, it essentially functions as a multiplayer system, as illustrated in the figure above.

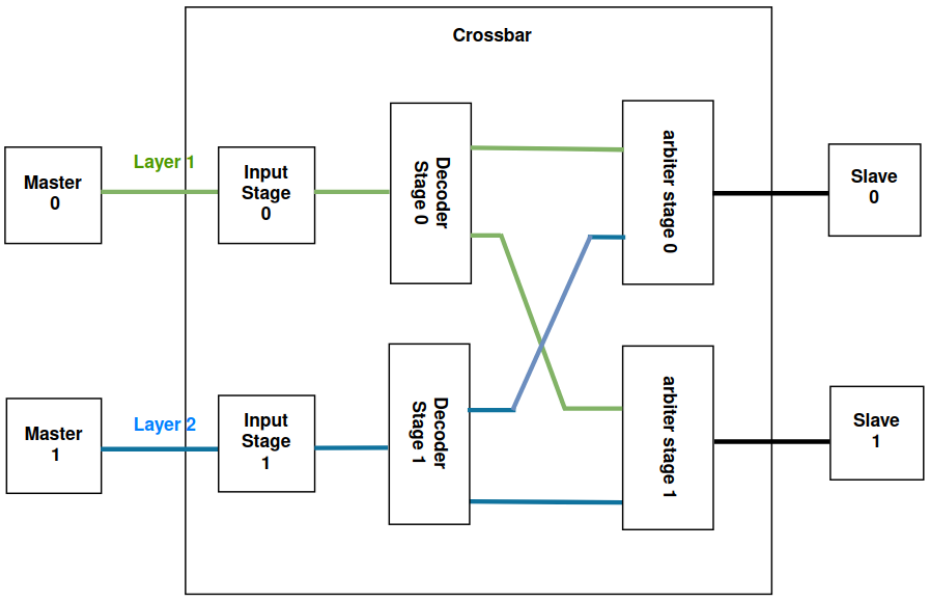


Figure 31: Crossbar Configuration

6.4.2 Arbiter Stage

The arbiter stage comprises two main blocks: the previously discussed unchanged arbiter block and Demultiplexer (Demux). In the Crossbar configuration, each slave is associated with a corresponding arbiter stage. The purpose of this stage is twofold: first, to choose among the masters competing to access the slave, and second, to demux the outputs of these slaves to the masters. Implementing this block was relatively straightforward, involving the integration of the previously designed arbiter and Demux without any modifications, as seen in the figure below, in contrast to the decoder stage.

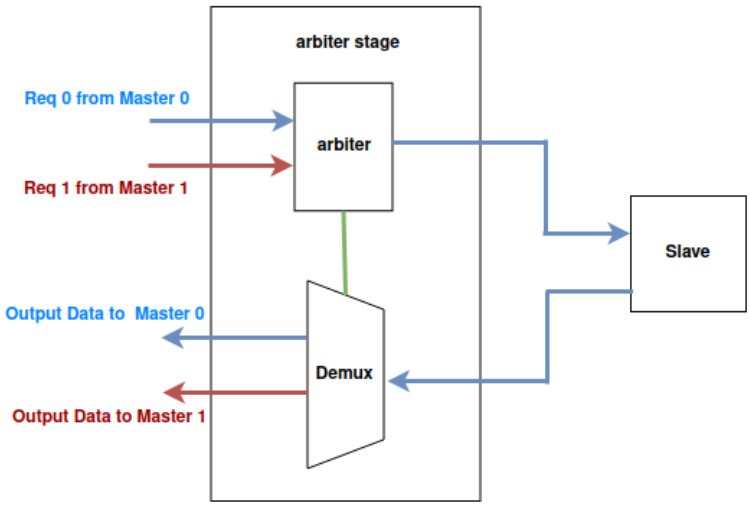


Figure 32: Arbiter Stage Topology

6.4.3 Decoder Stage

In accordance with the Crossbar Configuration, each master must be paired with a decoder stage. This stage incorporates the following building blocks :

- Address Decoder previously designed and unmodified .
- A multiplexer responsible for selecting the specific slave outputs to be transferred to the master. This multiplexer has been slightly modified to include an additional output called ***"ChangedSlave"***. This modification enables the multiplexer to indicate whether the HSEL output from the decoder is different between two successive requests.

  The multiplexer enhances its functionality by providing the ***"ChangedSlave"*** output, allowing for better tracking of changes in the ***"HSEL"*** signal. It serves as a useful indicator for the system, helping to identify cases where the master switches its target slave between consecutive requests.

  Additionally, two new blocks are introduced to complete the decoder stage.

- The Intercept Stage is essential for synchronizing requests when a master switches from one slave to another. For example, if Master 0 communicates with Slave 0 and then intends to communicate with Slave 1, the Intercept Stage ensures that the master can only initiate the new request after completing the previous one with Slave 0. This synchronization prevents interference between the two requests, ensuring orderly and synchronized data transfers.

  To achieve this, the Intercept Stage takes a signal called ***"ChangedSlave"*** as input. When this signal is high, the outputs ***"Intercepted HTRANS"*** for each slave are set to IDLE, making the master appear inactive for all slaves. However, when ***"ChangedSlave"*** is low, the Intercept Stage takes the selected slave from the address decoder as input and transfers only the value of ***"HTRANS"*** provided by the master to the selected slave. This ensures that ***"HTRANS"*** remains IDLE for the unselected slaves.

  By implementing this mechanism, the Intercept Stage effectively manages and synchronizes master requests, maintaining a well-ordered flow of transfers and promoting seamless communication between the masters and slaves.

- The Grant Multiplexor, or GNT Mux, serves the specific purpose of handling the ***"GNT"*** signals received from the arbiter stages. It takes these GNT signals as inputs and ensures that only the GNT signal from the arbiter stage associated with the targeted slave is transferred to the input stage. Similar to the logic employed in the slave output data multiplexor, the GNT Mux ensures the selection of the appropriate GNT signal.

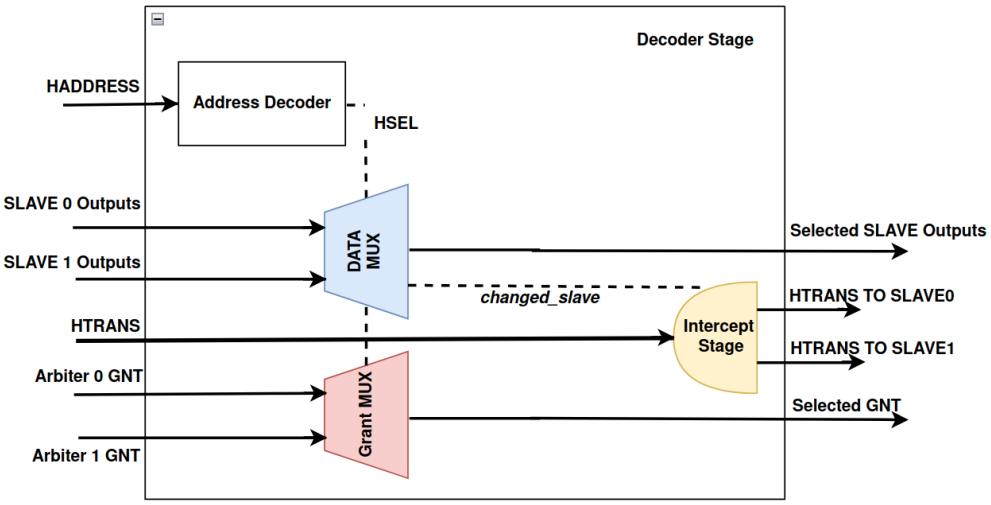


Figure 33: Decoder Stage Topology

6.4.4 Testing

We rigorously constructed the crossbar's components according to the specified setup. Using 2 masters and 2 slaves, we conducted various tests to encompass all scenarios. This involved simultaneous actions, like Master 0 communicating with Slave 1 while Slave 0 responded to Master 1. The simulation results (see figure below) validate the checks, confirming our crossbar's successful and expected behavior.

```
Note      :   (time 0 NS) Simulation start
Note      :   (time 6 NS) idle state requested: As expected (0h)
Note      :   (time 6 NS) address : As expected (44h)
Note      :   (time 6 NS) hsel: As expected (1b)
Note      :   (time 6 NS) transfer type : As expected (2h)
Note      :   (time 6 NS) type of operation : As expected (1b)
Note      :   (time 6 NS) size : As expected (2h)
Note      :   (time 6 NS) burst : As expected (0h)
Note      :   (time 6 NS) hprot : As expected (3h)
Note      :   (time 6 NS) hmastlock : As expected (0h)
Note      :   (time 8 NS) idle state requested: As expected (0h)
Note      :   (time 10 NS) Write valid request : As expected (1b)
Note      :   (time 10 NS) idle state requested: As expected (0h)
Note      :   (time 10 NS) address : As expected (40h)
Note      :   (time 10 NS) hsel: As expected (1b)
Note      :   (time 10 NS) transfer type : As expected (2h)
Note      :   (time 10 NS) type of operation : As expected (1b)
Note      :   (time 10 NS) size : As expected (2h)
Note      :   (time 10 NS) burst : As expected (0h)
Note      :   (time 10 NS) hprot : As expected (3h)
Note      :   (time 10 NS) hmastlock : As expected (0h)
Note      :   (time 10 NS) write data signal : As expected (aaaaaaaah)
```

Figure 34: Crossbar Testbench Results

While this setup notably enhances design performance through concurrent transfers, it introduces the trade-off of heightened area utilization. This demands M Decoders and M input stages for M masters, along with N arbiter stages for N slaves. Consequently, the resulting chip might be sizable and costly to manufacture. Therefore, prudent evaluation of these aspects is crucial for selecting the optimal design approach that aligns with project needs.

7 Conclusion

In conclusion, we proudly present the validation results of this internship work on implementing the AHB protocol. Throughout this report, we have diligently explored and developed various components, such as the slave model, master model, IBEX to AHB bridge, and interconnect systems. Now, as the crowning achievement, we seamlessly integrate all these individual blocks, showcasing the first mature stage of the Callisto platform.

The outcome of this work provides future designers with the essential tools to effectively utilize this platform, meeting the specific requirements of our valued clients. As we move forward, the next step after this internship is to replace the slave and master models with real IPs and rigorously ensure that the functionality remains intact.

The visual representation below showcases the capabilities achieved during this internship at Melexis, demonstrating what we can do with the Callisto platform today.

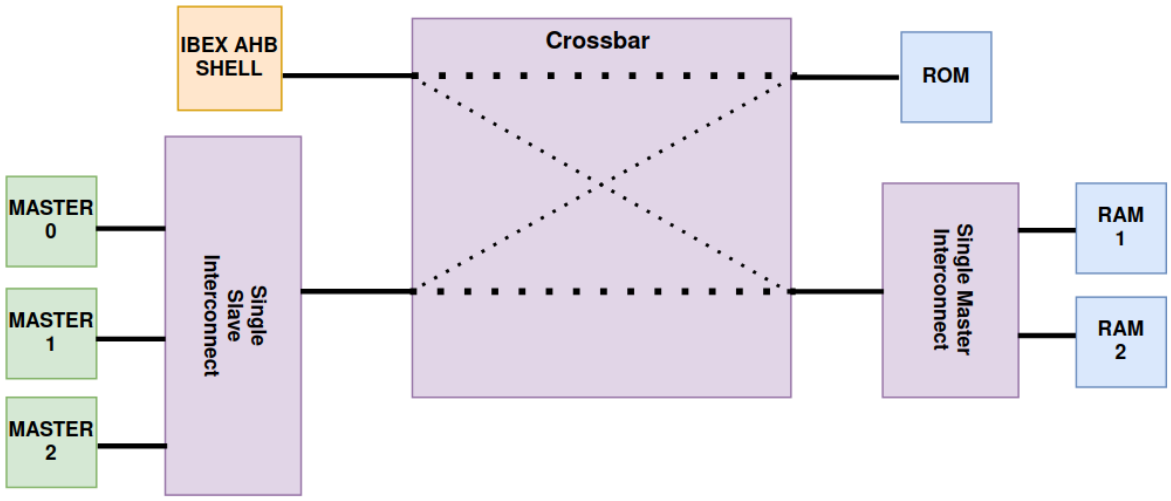


Figure 35: Conclusive Internship work

This internship has been an incredibly enriching experience, providing me with a comprehensive introduction to the digital design field. From understanding the bus protocol through the specification document to writing RTL code, creating test benches, running simulations, and rigorously verifying the code, I have been exposed to the entire digital design flow. This hands-on approach reaffirmed my passion for this field. I am thrilled to have been offered a long-term contract with the company as an Associate Digital Engineer before the internship concluded.

I am eager to delve deeper into this field, acquiring more knowledge and honing my skills further. Contributing to Melexis' success as a leading semiconductor company fills me with immense excitement and gratitude. I look forward to playing my part in the company's growth and innovation.

Bibliography

[1] Waterman Andrew et al. Ibex: An embedded 32-bit risc-v cpu core. https://ibex-core.readthedocs.io/en/latest/, 2019.

[2] Melexis Financial. Presentation results q2 2022. https://www.melexis.com/en/investors/results-and-presentations, 2022.

[3] Potladurthi Gouthami. *AHB Arbiter in AMBA bus With Effective Arbitration Logic for DMA*, volume 1. LAMBERT Academic Publishing, April 26, 2016.

[4] Will Green. Numbers in verilog. https://projectf.io/posts/numbers-in-verilog/, 2023.

[5] Arm limited. Multilayer ahb technical overview. https://developer.arm.com/documentation/dvi0045/latest/, 2004.

[6] Arm Limited. Amba ahb protocol specification. https://developer.arm.com/documentation/ihi0033/latest/, 2021.

[7] David Patterson and Andrew Waterman. *THE RISC-V READER: AN OPEN ARCHITECTURE ATLAS*. Strawberry Canyon; 1st edition, November 7, 2017.

Annex: A

AHB signal	Description	Values
hexokay	1 bit signal : Exclusive Okay is a signal that indicates whether an Exclusive Transfer has been successful or not. This signal is available when the AHB Exclusive Transfers property is set to True.	0 (OKAY), 1 (NOT OKAY)
hmaster	3 bit signal : The master identifier is created by a master that has multiple Exclusive-capable threads. To ensure unique identification of each master, the interconnect modifies this identifier. This signal is available when the AHB5 Exclusive Transfers property is enabled..	Depends on the number of active masters
hexcl	1 bit signal : The Exclusive Transfer signal signifies that the ongoing transfer is part of an Exclusive access sequence. Its support is contingent upon the AHB5 ExclusiveTransfers property being enabled.	0 (Not Exclusive), 1 (Exclusive)
hnonsec	1 bit signal : This signal indicates whether the current transfer is categorized as either a Non-secure transfer or a Secure transfer. Its availability depends on the AHB5 Secure Transfers property being enabled.	0 (Not secure), 1 (Secure)

Table 7: Optional AHB signals

Annex: B

```
Note     :   (time 0 NS) Simulation start
Error    :   (time 6 NS) address : Expected 40h, got 30h
Note     :   (time 6 NS) hsel: As expected (1b)
Note     :   (time 6 NS) transfer type : As expected (2h)
Note     :   (time 6 NS) type of operation : As expected (0b)
Note     :   (time 6 NS) size : As expected (2h)
Note     :   (time 6 NS) burst : As expected (0h)
Note     :   (time 6 NS) hprot : As expected (3h)
Note     :   (time 6 NS) hmastlock : As expected (0h)
Error    :   (time 8 NS) address : Expected 40h, got 46h
Note     :   (time 8 NS) hsel: As expected (1b)
Note     :   (time 8 NS) transfer type : As expected (2h)
Note     :   (time 8 NS) type of operation : As expected (0b)
Error    :   (time 8 NS) size : Expected 1h, got 2h
```

Figure 36: Error simulation example output

```
`timescale 1ns/10ps
`include "models/share/log/log.svh"
module testbench();
    crossbar_tb tb() ;
    import bus_ahb::* ;
    initial begin

        repeat (2) @(posedge tb.clk);

    fork begin
        repeat (1) @(posedge tb.clk);
        //normal requests
        tb.master0.Write(32'h0000_0040 , 32'h9999_9999 , 3'b010) ;
        tb.master0.Read(32'h0000_048,3'b010 , .expected_rdata(32'hFACE_CAF2) ) ;
        tb.master0.Read(32'h0000_094, 3'b010 , .expected_rdata(32'h0000_0006) ) ;
        tb.master0.Read(32'h0000_0050 , 3'b010 , .expected_rdata(32'hFACE_CAF4) , .expected_valid
    (1'b1) , .given_hmastlock(1'b1)) ;
        //Read what was written by the other master
        tb.master0.Read(32'h0000_080, 3'b010 , .expected_rdata(32'h1111_2222) ) ;
        //create error in slave 0 that will be detected by a request from the other master
        repeat (2) @(posedge tb.clk);
        tb.slave0.option <= alwaysinvalid ;
        //select no slave
        tb.master0.Read(32'h0000_194, 3'b010 , .expected_rdata(32'h0000_0000) , .expected_valid(0)
     ) ;
    end


    begin
        tb.master1.Write(32'h0000_0044 , 32'hAAAA_AAAA , 3'b010 ) ;
        tb.master1.WriteBurst_INCR4(32'h0000_080 , '{32'h1111_2222 , 32'h9999_9997 ,32'h9999_9998
    ,32'h9999_9999 }, 3'b010 , '{1,1,1,1}) ;
        tb.master1.Read(32'h0000_070,3'b010 , .expected_rdata(32'hxx) , .expected_valid(1'b0) ) ;
    end

    begin
        repeat(2) @(posedge tb.clk) ;
        tb.slave0.Write(32'h44 , 32'hAAAA_AAAA ) ;
        tb.slave0.Write(32'h40 , 32'h9999_9999 ) ;
        tb.slave0.Read(32'h48) ;
        tb.slave0.Idle();
        repeat(5) @(posedge tb.clk) tb.slave0.Idle();
        tb.slave0.Read(32'h50 , .expected_mastlock(1)) ;
        tb.slave0.Idle() ;
    end

    begin
        repeat(2) @(posedge tb.clk) ;
        tb.slave1.Idle();
        tb.slave1.Idle();
        tb.slave1.Idle();
        tb.slave1.Write(32'h80,32'h1111_2222 ,  .expected_burst(3'b011)) ;
        tb.slave1.Write(32'h84,32'h9999_9997 , .expected_trans(2'b11) , .expected_burst(3'b011)) ;
        tb.slave1.Write(32'h88,32'h9999_9998 , .expected_trans(2'b11) , .expected_burst(3'b011)) ;
        tb.slave1.Read(32'h94) ;
    end

    join
        repeat (10) @(posedge tb.clk);
        `log_Terminate ;

    end
endmodule
```

```
// Description:
//    Platform Callisto - Hardware Revision 1.0
//    Bus slave model
//
// Created: 20-Feb-2023
//      by: azb
//
// Copyright (c) Melexis Digital Competence Center
//
// ------------------------------------------------------------------------
//
//    bus_slave
//
// This model represent a slave on the ahb bus, by example a memory. By default
// the memory accept all address and all access. But you can change this using
// the defined tasks.
//
// You can use this model to check one access using the Read and Write tasks.

module ahb_bus_slave
#(
        parameter    START           = 0,
        parameter    SIZE            = 32'h1000  ,
        parameter    WAITSTATES      = 0
)(
    //data signal
        output    wire   [31:0] ahb_bus_slave_hrdata,     //read data,
    //Transfer response signals
        output    wire          ahb_bus_slave_hreadyout,  //ready,
        output    wire          ahb_bus_slave_hresp,      //valid transfer,
    //  output    wire          ahb_bus_slave_hexokay,    //used in Exlusive Transfer

    //select signal
        input     wire          ahb_bus_slave_hselx,      //access,
    //data signal
        input     wire   [31:0] ahb_bus_slave_hwdata,     //wdata,
    //Adress and control signals
        input     wire   [31:0] ahb_bus_slave_haddress,   //address,
        input     wire          ahb_bus_slave_hwrite,     //write,
        input     wire   [1:0]  ahb_bus_slave_htrans,     //transfer type : idle,busy,
    nonsequential,sequential,
        input     wire   [2:0]  ahb_bus_slave_hsize,      //transfer size byte , halfword or word
        input     wire   [3:0]  ahb_bus_slave_hwstrb,     //not supported yet
        input     wire          ahb_bus_slave_hready,     //the previous transfer is complete or
    the bus is free
        input     wire   [2:0]  ahb_bus_slave_hburst,     //burst type,
        input     wire          ahb_bus_slave_hmastlock,  //lock not really used only if we use
    slaves that can be accessed by more than one master,
        input     wire   [3:0]  ahb_bus_slave_hprot,      //protection control 3 bits for AHB-lite
    //Global signals
        input     wire          ahb_bus_slave_hrestn, //reset
        input     wire          ahb_bus_slave_hclk    //clock,
    );

    localparam      ARRAY_SIZE = SIZE / 4;

    event posedge_clk ;
    always_ff @(posedge ahb_bus_slave_hclk) begin
         -> posedge_clk;
    end

`include "models/share/log/log.svh"
    string          SlaveName = "bus_slave";
    localparam      IDLE = 3'b000, WRITE = 3'b001, READ = 3'b010  , ERROR1= 3'b011   , ERROR2= 3'
```

```
    b100  ;
    // Internal register
62  reg [31:0]       slave_start = START;
    reg [31:0]       slave_end = START + SIZE - 1;
64  reg [31:0]       slave_rdata , hrdataout ;
    reg              slave_ready ;
66  reg              slave_valid ;
    reg [31:0]       mem[0:ARRAY_SIZE-1]; //memory
68  reg [2:0]        state , next_state ;
    reg [31:0]       add_hold ;
70  reg [2:0]        size_hold ;
    reg              error , error_hold ;
72  reg [2:0]        cnt  ;
    reg [31:0]       mask ;
74  reg              valid_address , valid_size , valid_transfer  ;

76  int              waitstate = WAITSTATES; //internal waitstates that we can modify in the
    testbench
    reg              invalid  ;
78
    reg [1:0]        trans ;
80


82  // Shorthand AHB signals
    //   Outputs
84  logic    [31:0]      hrdata;
    logic                hreadyout;
86  logic                hresp;
    // logic                hexokay;
88  //   Inputs
    logic                hselx;
90  logic    [31:0]      hwdata;
    logic    [31:0]      haddress;
92  logic                hwrite;
    logic    [1:0]       htrans;
94  logic    [2:0]       hsize;
    logic    [3:0]       hwstrb;
96  logic                hready;
    logic    [2:0]       hburst;
98  logic                hmastlock;
    logic    [3:0]       hprot;
100 logic                hrestn;
    logic                hclk;
102
    assign ahb_bus_slave_hrdata     = hrdata;
104 assign ahb_bus_slave_hreadyout  = hreadyout;
    assign ahb_bus_slave_hresp      = hresp;
106 // assign ahb_bus_slave_hexokay    = hexokay;

108 assign hselx        = ahb_bus_slave_hselx;
    assign hwdata       = ahb_bus_slave_hwdata;
110 assign haddress     = ahb_bus_slave_haddress;
    assign hwrite       = ahb_bus_slave_hwrite;
112 assign htrans       = ahb_bus_slave_htrans;
    assign hsize        = ahb_bus_slave_hsize;
114 assign hwstrb       = ahb_bus_slave_hwstrb;
    assign hready       = ahb_bus_slave_hready;
116 assign hburst       = ahb_bus_slave_hburst;
    assign hmastlock    = ahb_bus_slave_hmastlock;
118 assign hprot        = ahb_bus_slave_hprot;
    assign hrestn       = ahb_bus_slave_hrestn;
120 assign hclk         = ahb_bus_slave_hclk;

122
```

```
124 import bus_ahb::* ;

126     // enum  {readonly , writeonly , alwaysinvalid , alwaysvalid , addr_size_invalid}  option ;
        option_invalid option ;
128
        //data to hold control signals of the previous access
130     always_ff @(posedge hclk or negedge hrestn) begin
            if (hrestn == 0) begin
132             add_hold   <=  32'hxxxx_xxxx ;
                size_hold  <=  3'bxxx ;
134             error_hold <=  0 ;
            end else if (hready) begin
136             add_hold   <=  haddress ;
                size_hold  <=  hsize ;
138             error_hold <=  error ;
            end
140     end
        //counter for wait states
142     always_ff @(posedge hclk or negedge hrestn) begin
            if (hrestn == 0 || state == IDLE) begin
144             cnt <= waitstate ;
            end else if (state == READ || state == WRITE || state == ERROR1) begin
146             cnt <= cnt == 0 ? waitstate : cnt - 1 ;
            end
148     end
        //state register
150     always_ff @(posedge hclk or negedge hrestn) begin
            if (hrestn == 0) begin
152             state <= IDLE;
            end else begin
154             state <= next_state;
            end
156     end
        //state transition FSM
158     always_comb begin
        if ((htrans == 2'b00 || htrans == 2'b01 || !hselx || !hrestn) )begin   //busy or Idle transfer
160         if (state == ERROR1)
                next_state = cnt == 0 ? ERROR2 : ERROR1 ;
162         else
                next_state = hready   ? IDLE   : state  ;
164     end
        else begin                                      //nonsequential or sequential transfer
166     case (state)
        IDLE: begin
168         if (hselx && hready && !invalid)
                next_state = hwrite ? WRITE : READ ;
170         else if (hselx && invalid && hready)
                next_state = ERROR1 ;
172         else
                next_state = IDLE ;
174     end

176     WRITE: begin
            if (hready && !invalid)
178             next_state = hwrite ? WRITE : READ;
            else if (hready && invalid)
180             next_state = ERROR1 ;
            else
182             next_state = WRITE ;
        end
184
        READ: begin
186         if (hready && !invalid)
                next_state = hwrite ? WRITE : READ;
188         else if (hready && invalid)
```

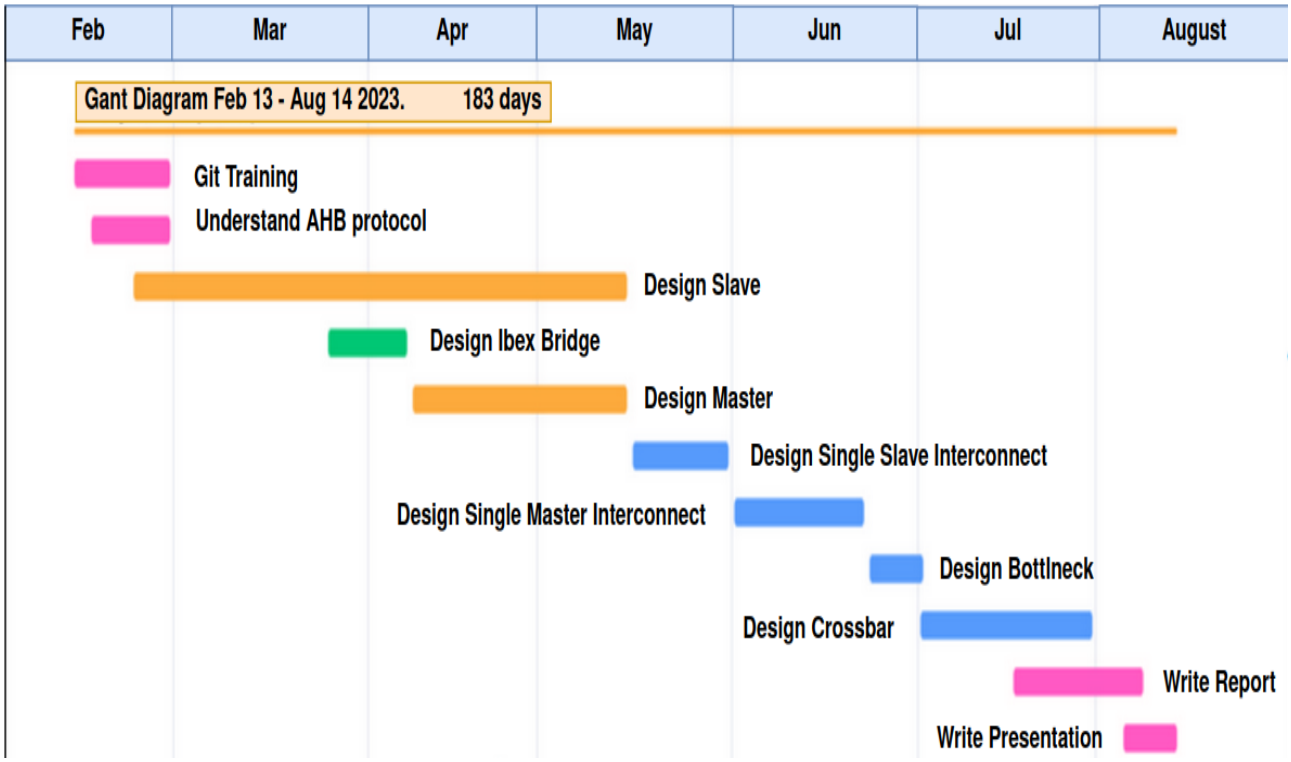
```
                next_state = ERROR1 ;
            else
                next_state = READ ;
        end

        ERROR1: begin
            next_state = cnt == 0 ? ERROR2 : ERROR1    ;
        end

        ERROR2: begin
            if (hready && !invalid)
                next_state = hwrite ? WRITE : READ;
            else if (hready && invalid)
                next_state = ERROR1 ;
            else
                next_state = ERROR2 ;
        end

        default: begin
            next_state = IDLE;
        end
        endcase
            end
        end
```

Gantt Diagram



Master in Micro and Nanotechnologies for integrated systems
Academic year : 2022/2023
Internship Duration : February 13th - August 14th

Company Supervisor :
Louis CHEREL
loc@melexis.com

Phelma Supervisor :
Lorena ANGHEL
lorena.anghel@phelma.grenoble-inp.fr

Student :
Aziz BADOUI
aziz.badoui@phelma.grenoble-inp.fr

Advanced High-performance Bus (AHB) implementation

Melexis Technologies SA
4 Pl. des Vosges, 92400
Courbevoie, France
https://www.melexis.com

Project Description

The target of this internship is to implement the Advanced High-Performance Bus (AHB), which entailed implementing and verifying generic bus arbiters and decoders for our next-gen CPU platform (Callisto).

Expected Results

A Verilog design database of the arbiter, slave, and master model, a set of SystemVerilog self-testable test cases fully covering the design and ensuring the correct behavior of the AHB blocks.

Available Resources and Guidance

Full access to the company's toolbox, including software like VsCode, Xcelium, and Cadence apps. Learning resources such as RISC-V ISA textbooks and SystemVerilog tutorials were provided. Continuous guidance came from my supervisor and colleagues, from juniors to team leads. Training sessions covered Git, CPU architecture, and ATPG tools. My workspace features an open space environment with a Linux-running Lenovo Laptop, dual large screens, and all the necessary peripherals