Bus Extraction From RTL

Ahmed Medhat

Thesis presented for a Master Degree in Electronics

Electronic System Design

Norges teknisk-naturvitenskaplige universitet

Norway

July 2019

ACKNOWLEDGEMENTS

I would first like to thank my thesis advisor Prof: Kjetil Svarstad of the Electronics at NTNU. The door to Prof. Svarstad office was always open whenever I ran into a trouble spot or had a question about my thesis or writing. He consistently steered me in the right the direction whenever he thought I needed it.

I would also like to acknowledge Berend Dekens from Nordic-Semi-Conductor as a supervisor, and I am gratefully indebted to his for his very valuable guidance and comments on this thesis. His guidance was a huge help to understanding and implementing the work

Finally, I must express my very profound gratitude to Eivind Fylkesnes from Nordic-Semi-Conductor as a co_supervisor, he was always helpful. His input was essential in every step.

This accomplishment would not have been possible without them. Thank you

Contents

1 Introduction **5**

1.1 Problem Description . . . 5

1.2 Steps . . . 6

1.3 Methodology . . . 8

2 Background **9**

2.1 Terminology . . . 9

2.2 Bus Signal Types . . . 11

2.3 Clocking . . . 11

2.4 Decoder . . . 13

2.5 Arbitration . . . 14

2.5.1 Static priority (SP) . . . 14

2.5.2 Round robin (RR) . . . 15

2.5.3 Time division multiple access (TDMA) . . . 15

2.6 DATA transfer modes . . . 16

2.6.1 Single non_pipe_lined transfer . . . 16

2.6.2 Pipelined transfer . . . 16

2.6.3 Burst Transfer . . . 16

2.6.4 Split transfer . . . 18

2.6.5 Out of order transfer (OoO) . . . 18

2.7 Bus topology . . . 19

3 Approaches **20**

3.1 Bus Protocols . . . 20

3.1.1 advanced peripheral bus (APB) . . . 20

3.2 Advanced high-speed bus (AHB) . . . 22

3.3 System . . . 23

4 Implementation of extraction algorithm **24**

4.1 Dump . . . 24

4.2 Extraction . . . 26

4.3 AHBLite-Multi-Layer-Interconnect Connection Table 28
4.4 Signal Force 32
4.5 Text Parsing 34
4.6 Graph Parsing 36
4.7 Address Mapping 38

5 Derivations & Conclusion **39**
5.1 Evaluation & Results 39
5.2 Achieved Goals 40
5.3 Further Work / Possible Improvements 40

List of Figures

1 SoC with a bus-based communication architecture [1] 10
2 Classification of bus signals [1] 11
3 Synchronous bus [1] 12
4 Asynchronous bus [1] 12
5 centralized decoder [1] 13
6 Distributed Decoder [1] 14
7 Single non-pipelined data transfer mode [1] 16
8 Pipelined data transfer [1] 17
9 Burst pipelined & non-pipelined data transfers [1] 17
10 Single bus [1] 19
11 Hierarchical bus [1] 19
12 APB signals descriptions [5] 21
13 APB state machine [5] 21
14 Master interface [6] 22
15 Slave interface [6] 22
16 M4 processor with M33 cortex 24
17 Cache_Sub_System Dump 26
18 AMLI Sub_System Connection table 30
19 AMLI Sub_System Simulation 33
20 AMLI Sub_System Print_out_file 34
21 AMLI connection to Cache 36
22 AMLI connection to Syn_up_bridge 36
23 AMLI Connections 36

1 Introduction

1.1 Problem Description

As the complexity of multi-core architectures grow, it becomes harder and harder to design, test and verify the bus interconnects that tie all the components in a chip together. Even though the internal architectural design documents provide an abstract and segmented view of the design, this does usually not reflect the actual RTL. In this project, the goal is to augment the RTL description in such a way that meta-information about the bus system and where it connects to, can be extracted automatically. This information can then be used to:

- Visualize the bus architecture under test
- Automatically generate an address map per master peripheral in the system
- Verify the correctness of this address map with the architectural specification
- Detect design flaws: bus loops, mapped-unmapped ranges, aliased ranges, etc.
- As a stand-alone project, this task is to create an overview of and evaluate the requirements and start implementation.

As a master thesis, background research should be performed on existing techniques and approaches. Results from this initial research can then be used to steer the expected implementation requirements as listed above. Additional investigation on the potential use of this system enhancement can further drive the requirements and provide options towards formal verification for example.

1.2 Steps

- Since the thesis is not a continuation so no pre-study was made a background is needed in order to understand the theory of bus-based communication.

- Understanding how the Nordic system IPs and design structure by completing a project in system- Verilog.

- Start looking into ARM Cortex-M4 processor and find a way to extract information.

- After the extraction of the information, a graphical representation should be implemented.

- Compare between the abstract design and the implemented design.

- Think of a solution on how to improve the structure and Detect design flaws.

Abstract

This thesis gives a comprehensive overview of the extraction of bus information from RTL. Since this thesis is not a continuation, a pre_study had to be made. The difference between looking at the design and the RTL code is quite substantial. Hence a detailed design was implemented based on the information extracted from the RTL to create a new design for ARM-M4-processor. The idea was put into action by changing the RTL code with_out changing the characteristics of the system. The RTL augmentation took place on various modules. The extracted information was then used to create a graphical representation in python which was then compared to the existing design. The next step was to look into the address map and from there create a list of which master can access which slave, not only that but also create a situation where the ranges overlap multiple of times. Finally explaining how loops are created and ways minimize it. This thesis starts from scratch and builds_up to the implementation.

1.3 Methodology

Over the years, System-on-Chip (SoC) designs have evolved from a single-processor unit and single-memory designs to multi-processor multi-processor systems with multiple chip memories, standard peripherals, and ASIC blocks. More and more components are being integrated into these designs to share the ever-increasing processing load, and there is a corresponding increase in communication between these components. Communications between components are often in the critical path of a SOC design A very common source of performance bottlenecks. It is, therefore, necessary for system designers to focus on exploring the communication design.

Bus-based Communications architecture based on a common bus such as AMBA is a common choice to connect the components inside the chip in the current SOC designs. The bus structures can be configured in several different ways thus, a great effort to re-engineering had to be made because of the very complex nature Of these systems. Accordingly the focused shifted to the analysis of bus-based System-on-Chip (SOC).

Extracting bus information from a complex system is not an easy task. Understanding the function of the signals and how they come together is essential. This was the stepping stone for the exploration of the system. In order to create a more detailed design rather than the existing one, the extraction was divided into manual and automatic. The manual represents the static information opposite to the automatic. After the extraction is completed the next step was to create a graphical representation.

In the upcoming sections the methodology and contribution are outlined. The background section gives a detailed overview on how buses work. The next section focuses on the protocols that are used in the M4-ARM system. Right after that it jumps to the implantation and how the problems are tackled. Finally come the evaluation and conclusion part.

2 Background

Buses are critical and vital components of any architecture. They act as a communication channel connecting distinct component together [1]. The set of paths connecting different components/modules are labeled interconnection structure. Busses are universally used means of communication between peripherals and components in the system_on_chip (SoC). A signal shared channel can connect multiple components together. The channel can be realized in the form of a wire. Originally buses are a broadcast medium, in the majority of the situation, the transmitted data is meant for a precise component and is discarded by the remainder of the components. The data transfers start to form the output pins in the source through the wire until it is received by the input pins in the destination. Upon receiving the data an acknowledgment is sent to the source to indicate the data is received correctly. A set of protocols are implemented to specifically define the communication characteristics and is divided into:

- Temporal defines the time frame and sequence order
- Spatial defines the message size

2.1 Terminology

System on chips (SoCs) has a different set of components. Components which start and manage the read and write data transfers are called masters. A processor is one example of master components that read/write data from/to different components using the bus as the communication medium [1]. Every master component is linked to the bus using a series of communicating signals. Slaves are a set of components that only reply to transfer requests originated from the masters. Slaves are not able to trigger a data transfer. One example of a slave component is a memory. The components don´t have to be either master or slave some components can act as both master and slave. Direct memory access(DMA) is a perfect example of a hybrid component. The slave port in the DMA permits the master to read_from/write_into. The master port in the DMA starts data transfers between various blocks. Bus_based_communication is

established on more than just the master and slave component it also includes logic components like:

- Decoder: A logic component that translates the destination address during data transfer. It pinpoints which slave the data transfer is intended to. The decoder can be centralized or distributed.

- Arbiter: A logic component that has the power to choose which master to grant access to the bus. Arbiter have different schemes to prevent starvation and make sure critical data is delivered.

- Bridge: A logic component that has the responsibility of connecting buses together. Buses have different protocols and clock frequencies that´s why bridges are essential.

Fig.1 shows SoC with a bus-based communication architecture displaying communication components.

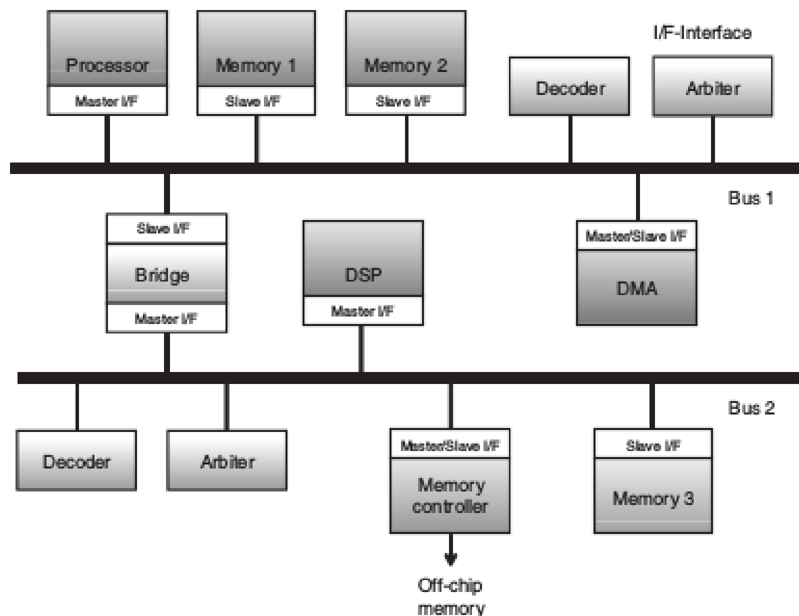

Figure 1. SoC with a bus-based communication architecture [1]

2.2 Bus Signal Types

Buses comprise of multiple pathways/lines[2], each line is able to transmit data, Fig.2 demonstrate signal types. There are three types of lines:

- Address line: The address lines determine the maximum capacity of the system. They are also responsible for naming the source and destination of the data.
- Data lines: The data lines provide a pathway between system components. They also hold a number of detached lines, each line can hold one bit at any given time. The number of lines controls the overall performance of the system.
- Control line: The control lines are used to pass on instructions/commands (example: Memory read/write signal — Interrupt requests — Clock signals) to coordinate and manage the activities.

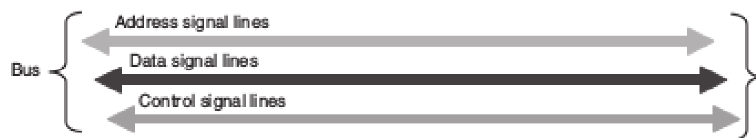

Figure 2. Classification of bus signals [1]

2.3 Clocking

Buses use different types of clocking for data transfer. The choice will be based on the function and application.

- Synchronous bus: A bus that incorporates a clock signal to control the transfer. Fig.3 represents a synchronous bus. The master initiates the transfer by sending the address

ADDR and asserting the write control signal at the first clock cycle. The data is then will be written at the second clock cycle

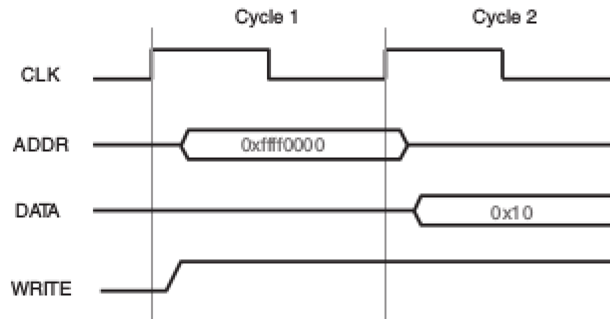

Figure 3. Synchronous bus [1]

- Asynchronous bus: Clock signal does not exist in the control signal of the bus. In this case, synchronization will take place with the help of acknowledgment signals (handshaking protocols) to ensure that the data transfer was concluded. Fig.4 represents the asynchronous bus where Ack signal is added for synchronization.

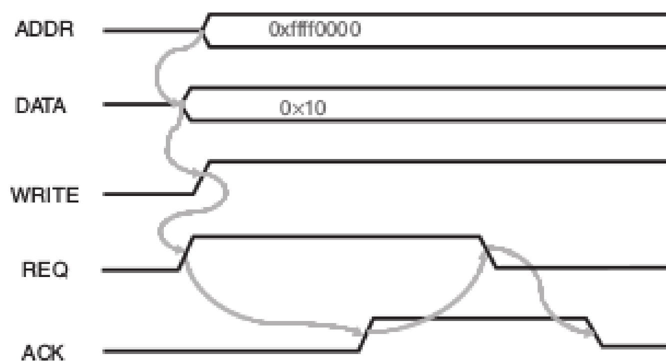

Figure 4. Asynchronous bus [1]

2.4 Decoder

The master(source) initiates the data transfer by transmitting the address to the slave(destination) [2]. Every component in the SoC design is assigned an address map (range of addresses). The function of the decoder is to translate the address and select the correct slave. There are two types of decoding:

- Centralized: As the name suggests there is one decoder for all the slaves. A select signal is constructed to determine the appropriate slave to read/write data. The advantage of this scheme is that it simplifies the design makes it easily extensible. Fig.5 shows the centralized design.

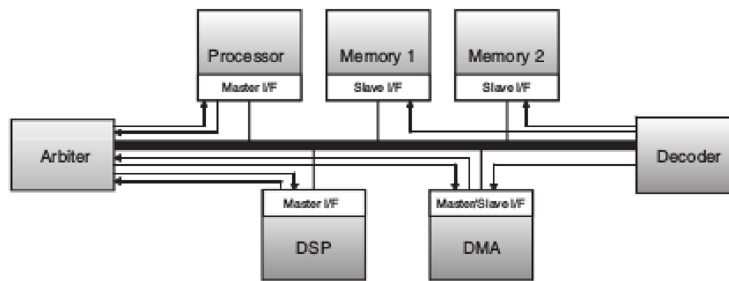

Figure 5. centralized decoder [1]

- Distributed: Each slave has its own separate decoder. The data will be transmitted to every slave during the data transfer. Every slave will decode the address. Only the slave that corresponds to the address will get to keep the data unlike the rest of the slaves will discard the message. The distributed scheme is more complex since it will lead to more hardware. Fig.6 shows the distributed design.

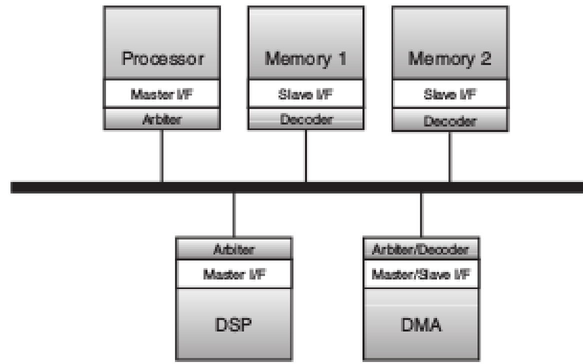

Figure 6. Distributed Decoder [1]

2.5 Arbitration

It is possible that two or more masters request access to the bus at the same time [3]. This is a huge problem since starvation and critical data delay is a possibility which can very well occur. Shared Bus can only manage individual data transfer at any given time. The arbitration scheme is required in order to decide which master will gain access. Arbitration can also centralized Fig.5 or distributed Fig.6.

There are various schemes used in bus-based communication. Each scheme is based on the set of requirement needed. All schemes share the criteria of fairness.

2.5.1 Static priority (SP)

SP is based on assigning fixed priority values to the masters [1]. The master with the highest priority will be the one who will gain access to the bus first. The SP scheme can be realized in pre_emptive or non_pri_emptive.

•**pre_emptive**: lower priority is terminated instantly without finishing the data transfer when higher priority request access to the bus.

•**non_pri_emptive**: lower priority gets to complete the data transfer even if a higher priority request access to the bus. After the data transfer is completed the bus access is given the high priority master.

SP is an elementary straightforward scheme that can ensure crucial data transfers are completed in time. On the other hand, it could lead to starvation for the lower priority masters.

2.5.2 Round robin (RR)

Round robin is based on the idea of granting access to the bus is a circular manner. The scheme is fair but can lead to delay of critical data transfers.

2.5.3 Time division multiple access (TDMA)

The scheme allocates time slots (frames) of different lengths, depending on the bandwidth requirements o the master [1]. Choosing the number of time slots is extremely crucial. The length must be sufficient to complete an individual data transfer, but not too long for starvation to take place.

2.6 DATA transfer modes

2.6.1 Single non_pipe_lined transfer

The elementary form of the transfer of data. The master initiates the transfer by requesting bus access from the arbiter. After the arbiter grant access to the master, the address is sent on the next cycle then, write data on the next cycle Fig.7.

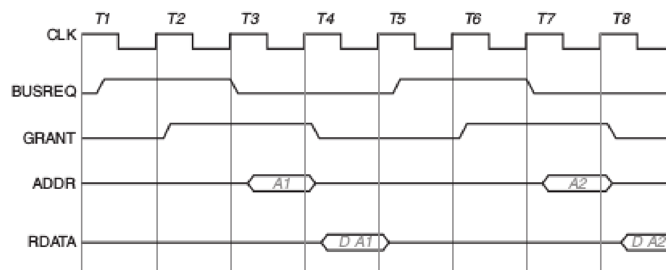

Figure 7. Single non-pipelined data transfer mode [1]

2.6.2 Pipelined transfer

The address and the data overlap for multiple data transfer to improve the performance [4]. Fig.8 represents a pipeline transfer where two write transfers started by two different masters. Master one and two request bus access. The arbiter decides to give access to master one. Master one sends the address in the first cycle then data to write at the second. During the write of master one the arbiter grant master two access to the bus when it sends the address. Pipeline transfer is more complex but has better performance.

2.6.3 Burst Transfer

Commonly any master with multiple data transfers will require multiple arbitrations for each one of the data transfer. Burst transfer boosts the performance by eliminating the need to request for multiple arbitrations. Burst mode can be implemented in pipelined or non_piplined Fig.9.

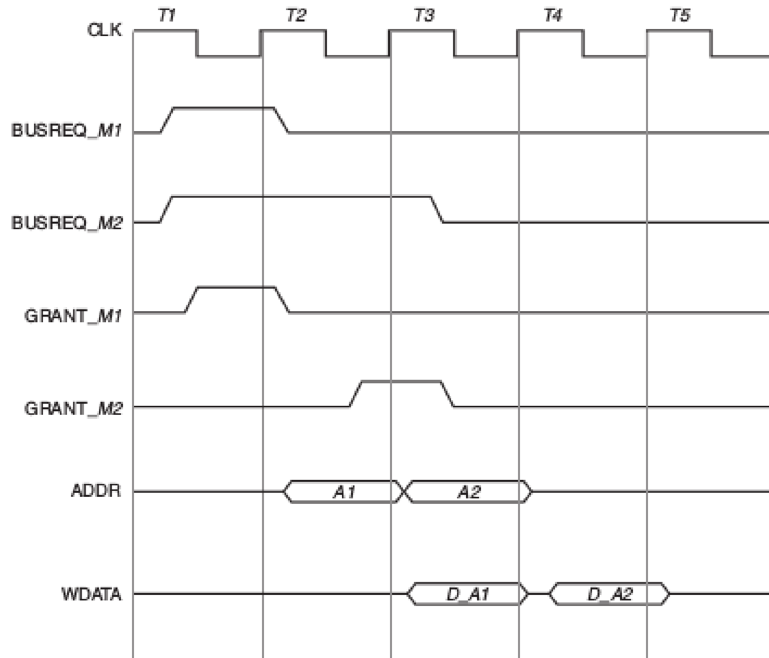

Figure 8. Pipelined data transfer [1]

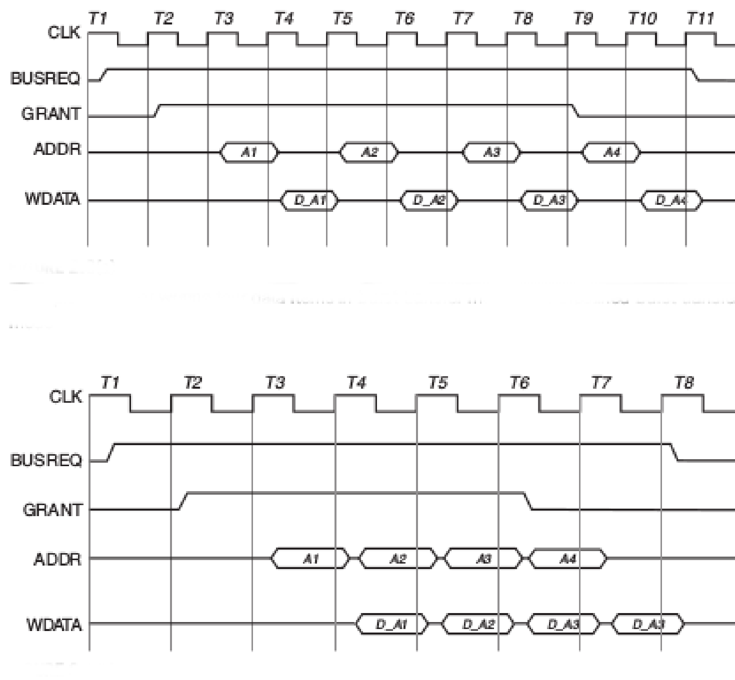

Figure 9. Burst pipelined & non-pipelined data transfers [1]

2.6.4 Split transfer

It is possible that the slave need multiple cycles to read/write data during data transfer. The bus can only be controlled by a single master [2]. The bus will be in a state of idle until the slave complete the transfer. In this case, the bus will be under_preforming. The split mode is used to grant another master access to the bus during the idle cycles. The bus initiates a spilled command where the bus will be under the control of another master. After the initial slave complete the transaction the bus will issue a un_split command to return the bus to the initial master. Split mode improves the performance drastically without risking any delay in critical transfers.

2.6.5 Out of order transfer (OoO)

OoO mode makes t possible for the master to start a transfer without waiting for previous data transfers to complete, which boosts system performance by processing numerous data transfers. This is achievable by assigning an ID to all data transfers. If a master initiates two data transfer, it is possible for the second transfer to be complete before the first.

2.7 Bus topology

There is a considerable number of bus types and structures which influence area, power, complexity, cost, and performance. Single shared bus Fig.10 is the simplest scheme. The single shared bus is acceptable for simple Socs with few components but it can't be scaled to handle larger systems since it only permits single data transfer at any given time.

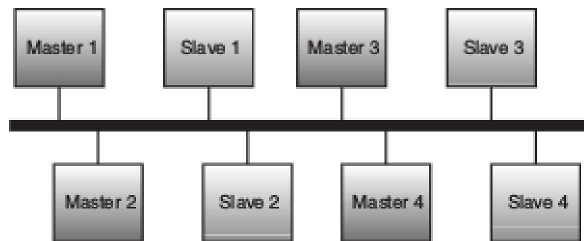

Figure 10. Single bus [1]

A more advanced topology is the hierarchical bus Fig.11. In this topology, the components are connected to multiple buses that connect to each other with a bridge. Concurrent data transfers are achievable with the hierarchical bus. hierarchical bus improves the performance but on the expense of complexity [4].

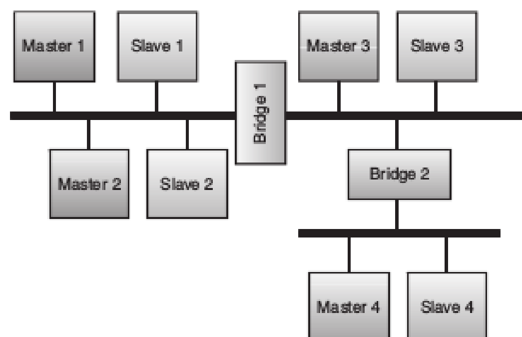

Figure 11. Hierarchical bus [1]

There are a numerous configuration of bus topology which will be mentioned but not explained. More examples of bus topology Split, Full bus crossbar, Partial bus crossbar, and Ring Bus.

3 Approaches

Looking for existing techniques and ways to implement the information extraction. The surprise was that there were no tool or research that could help. It was unexpected since there is a big difference between looking at the design and RTL source code. The only option was to examine the Nordic structure and figure out how their IPs are connected. Nordic uses a standard set of protocol for all communication.

3.1 Bus Protocols

Nordic semi conductor uses ARM communication architecture standards. Those standards are the guidelines for communication. Advanced Micro-controller Bus Architecture (AMBA) is one of the most comminly used standards for on-chip communication. This thesis focuses on bus extraction from ARM cortex-M4 system which uses an advanced peripheral bus (APB) and advanced high-speed bus (AHB) protocols. Those protocols will be the main core of the implementation.

3.1.1 advanced peripheral bus (APB)

APB is a part of the AMBA protocol family [5]. It describes a low-cost interface that is developed for low power consumption. The APB protocol is not pipe-lined, every data transfer will take at least two cycles. APB enclose a list of signals Fig.12. It also has two independent data buses, one for data read and the other for data write. The protocol uses PSLVERR to signify/flag the occurrence of an error either in reading/writing transaction.

Signal	Source	Description
PCLK	Clock source	Clock. The rising edge of **PCLK** times all transfers on the APB.
PRESETn	System bus equivalent	Reset. The APB reset signal is active LOW. This signal is normally connected directly to the system bus reset signal.
PADDR	APB bridge	Address. This is the APB address bus. It can be up to 32 bits wide and is driven by the peripheral bus bridge unit.
PPROT	APB bridge	Protection type. This signal indicates the normal, privileged, or secure protection level of the transaction and whether the transaction is a data access or an instruction access.
PSELx	APB bridge	Select. The APB bridge unit generates this signal to each peripheral bus slave. It indicates that the slave device is selected and that a data transfer is required. There is a **PSELx** signal for each slave.
PENABLE	APB bridge	Enable. This signal indicates the second and subsequent cycles of an APB transfer.
PWRITE	APB bridge	Direction. This signal indicates an APB write access when HIGH and an APB read access when LOW.
PWDATA	APB bridge	Write data. This bus is driven by the peripheral bus bridge unit during write cycles when **PWRITE** is HIGH. This bus can be up to 32 bits wide.
PSTRB	APB bridge	Write strobes. This signal indicates which byte lanes to update during a write transfer. There is one write strobe for each eight bits of the write data bus. Therefore, **PSTRB[n]** corresponds to **PWDATA[(8n + 7):(8n)]**. Write strobes must not be active during a read transfer.
PREADY	Slave interface	Ready. The slave uses this signal to extend an APB transfer.
PRDATA	Slave interface	Read Data. The selected slave drives this bus during read cycles when **PWRITE** is LOW. This bus can be up to 32-bits wide.
PSLVERR	Slave interface	This signal indicates a transfer failure. APB peripherals are not required to support the **PSLVERR** pin. This is true for both existing and new APB peripheral designs. Where a peripheral does not include this pin then the appropriate input to the APB bridge is tied LOW.

Figure 12. APB signals descriptions [5]

APB has three operating states Fig.13. Starting with the IDLE state which is the default state. Moving on to the SETUP state where a transfer is requested, PSELx signal is asserted high to indicate that the slave is selected. Finally comes the ACCESS state where the data transfer is carried-out and the state machine shift back to the idle state.

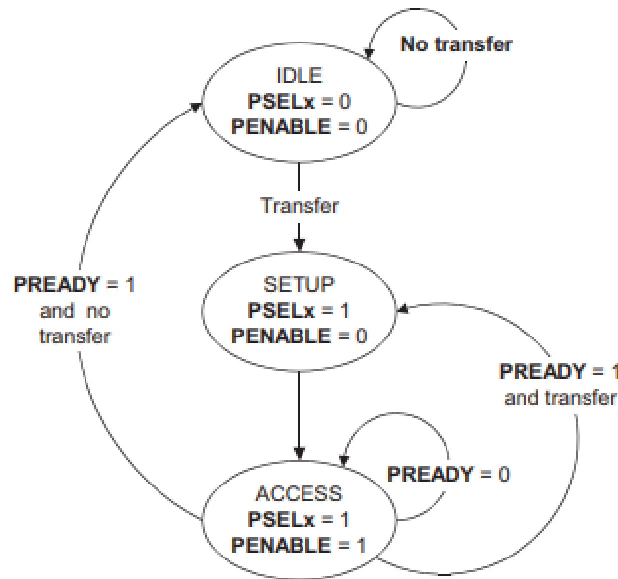

Figure 13. APB state machine [5]

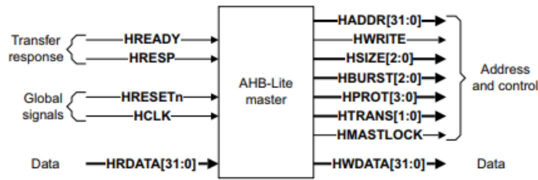

Figure 14. Master interface [6]

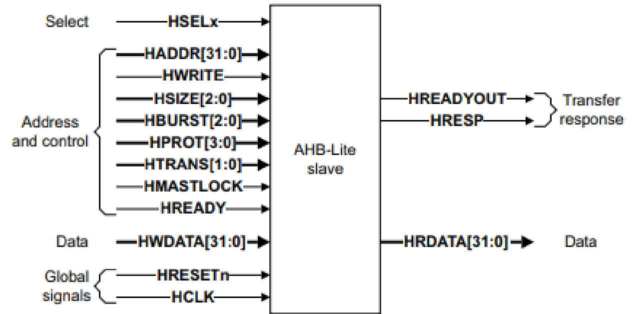

Figure 15. Slave interface [6]

3.2 Advanced high-speed bus (AHB)

The AHB bus protocol is designed for high performance/frequency systems [6]. As it was previously mentioned the master provides address and control information for reading/write transfer. The master interface can be examined Fig.13. Slaves reply to the transfer started by the master. The slaves use HSELx single to flag the decoder to identify the intended slave Fig.14 represent the slave interface.

3.3 System

The system from which the bus extraction will take place from Arm-M4 processor with M33 cortex. The design details are confidential so it can't be presented. Fig.16 shows a mock-up representation of the system, the design contains five main functional blocks. The Cortex-M4 CPU Subsystem is developed to support wide flexibility in how a CPU is connected to code/data memory blocks and peripherals. Two AMLI blocks which arbitrate accesses from both sources internal to this subsystem and external sources makes it possible at the system level to connect several subsystems, as well as other subsystems, into a wide variety of configurations.

The task of bus extraction from RTL will take place by augmenting the source code of a set of modules and the test-bench. The set of modules are :

- Cache
- CpuCoreCM33
- AHBLiteMultiLayerInterconnect is a module providing a single layer bus interconnect for the AHBLite protocol.
- Ahb2AhbAsyncBridge: The Ahb2AhbAsyncBridge implements an asynchronous AHB to AHB bridge between two clock and/or power domains.

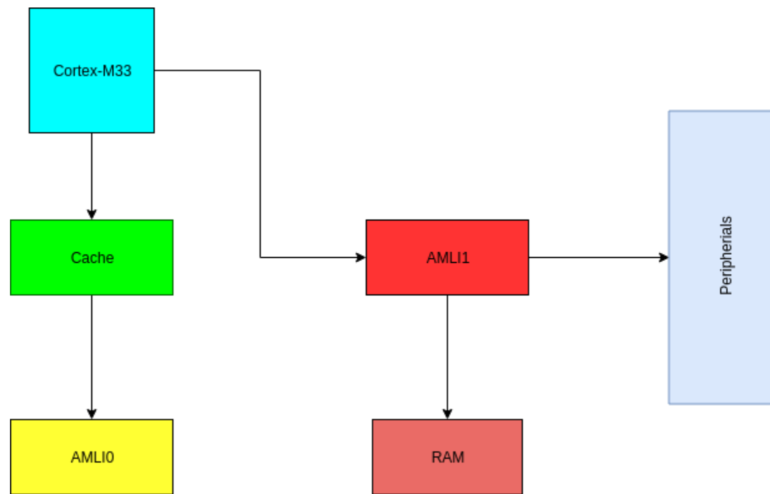

Figure 16. M4 processor with M33 cortex

4 Implementation of extraction algorithm

This chapter will focus on implementing a solution to the problem of having an abstract segmented overview that does not reflect the RTL. Using the insights gained in chapter 2, a plan was formalized in order to solve the problem.

4.1 Dump

Since no prior pre-study took place, and due to the complexity of the system and shortage of time some alternatives had to be found in order to extract the information from the system. The system is written in system_verilog. The dump file command is supported by system_verilog which is used in order to dump the changes in the values of nets and registers in a file that is named as its argument. The dump will be recorded in a file called VCD file that stands for value change dump.

Performing a wide system dump was proven difficult to handle since millions of lines of information were written in the recorded file. Another approach was to take individual modules and

perform the dump. Localizing the dump made it easier to understand the variables change and signal information and parameter. In order to improve on the output, a time frame was added in order to realize more concrete information. The code below shows a part of the dump command performed on the cache subsystem. Fig.17 shows a part of the output written to the text file

```
initial begin
file = $fopen(""Cache.vcd","w");
$dumpfile("Cache.vcd");
//property definitionslevel is set to 0 ,dumps ALL the variables of that module and all the
    variables in ALL lower level modules instantiated by this top module.
$dumpvars(0,Cache);
#100;
$dumpoff;
#1020;
$dumpon;
#1040;
$dumpoff;
$finish;
 end
```

```
$var parameter 1 ! INCLUDE_
$var parameter 32 " NUM_CL
$var parameter 32 # DFT_NUM_O
$var parameter 32 $ AHB_AW $end
$var parameter 32 % AHB_DW $end
$var parameter 32 & AHB_DW_WIDE $end
$var parameter 32 ' AHB_MW $end
$var parameter 32 ( APB_AW $end
$var parameter 32 ) APB_DW $end
$var parameter 12 * ID_
$var parameter 12 + ID_ end
$var parameter 12 , ID_ nd
$var parameter 12 - ID_ end
$var parameter 12 . ID_
$var parameter 12 / ID_
$var parameter 12 0 ID_ end
$var parameter 12 1 ID_ end
$var parameter 12 2 ID_
$var parameter 12 3 ID_ $end
$var parameter 12 4 ID_ $end
$var parameter 12 5 ID_ end
$var parameter 12 6 ID_
$var parameter 32 7 RV_
$var parameter 32 8 RV_ $end
$var parameter 32 9 RV_ $end
$var parameter 32 : RV_ $end
$var parameter 32 ; RV_CACH
$var wire 1 k! apbPAddr [11] $end
$var wire 1 l! apbPAddr [10] $end
$var wire 1 m! apbPAddr [9] $end
$var wire 1 n! apbPAddr [8] $end
$var wire 1 o! apbPAddr [7] $end
$var wire 1 p! apbPAddr [6] $end
$var wire 1 q! apbPAddr [5] $end
$var wire 1 r! apbPAddr [4] $end
$var wire 1 s! apbPAddr [3] $end
```

Figure 17. Cache_Sub_System Dump

4.2 Extraction

The next step to be taken after extracting all that information about the signals and their parameters is to to create a new file in the TestBench in order to extract information about the bus e.g. size, name of masters,name of slaves . Classes and function were used to optimize the reuse-ability and make it easier to follow.

The code begins with setting up a package to store and share data that can be used in multiple other modules and interfaces. Moving on to the creation of queues for masters and slaves and

then using the push_back command in order to return the last element in the queue. There are two types of masters, one with a connection table and one without which will be discussed in detail later on. Creating functions for the names of the master, names of slaves, number of masters and slaves and finally giving an id to discover the connection which will be discussed separately. The code below is a small part of the actual code just to give an overview of what was augmented in the source code.

```
package busAnalyser;
logic [31:0] busID = 0;
class node;
    typedef struct {
    string    name;
        string    masters;
        string    slaves;
        int       size ;
        int       numberOfInstances;
        int       busSize ;
        int       slaveID;
        int       masterID;
        int       number;
// NEW Master with a connection table
string    masters1;
    } ty_Bus;
    ty_Bus  slaves [$];
    ty_Bus masters [$];
    ty_Bus masters1 [$];
function new(string name);
        this.name =  name;
    endfunction
function addMaster(string name);
        ty_Bus mBus;
        mBus.name = name;
        this.masters.push_back(mBus);
    endfunction
function numberofmasters(int number);
        this.numberM = number;
    endfunction
function addMaster1(string name, int id);
        int z ;
        ty_Bus mBus;
        mBus.masterID = id;
        mBus.name = name;
        this.masters1.push_back(mBus);
  endfunction
```

```
38 endclass
39 class BusStructureParser;
40     node nodes [$];  // empty queue
41 function addNode(node mNode);
42 nodes.push_back(mNode);     //removes and returns the last element of the queue
```

4.3 AHBLite-Multi-Layer-Interconnect Connection Table

Moving on to the AMLI. The AMLI module is the only module which contains a connection table. The connection table is represented in the form of a matrix. The matrix components are: master, fragment and slave. The fragment is an indication of whether there is a connection between the master and the slave and the fragment is represented by one bit [0, 1]. There is a matrix $\begin{bmatrix} Masters & Fragment & Slaves \end{bmatrix}$ for each slave-master pair indicating whether the master have access to the slave or not.

The code below shows how the connection table was derived from inside the TestBench. A function was constructed in order to add the connection for each master in the form of a queue and pushing back the last element.

```
1  function addConnectionsToMaster(int connectedTo [$]);
2    masterConnections.push_back(connectedTo);
3    endfunction
4
5  foreach(nodes[i].masters[j]) begin
6      int connectedTo[$] = nodes[i].masterConnections[j];
7      $fwrite(file, "MasterName: %s \n ", nodes[i].masters[j].name);
8      $fwrite(file, "Connected to slaves:");
9      foreach(connectedTo[k]) begin
10     $fwrite(file,"%d ,", connectedTo[k]);
11 end
12     $fwrite(file,"\n");
13 end
```

The matrix will be constructed inside the AMLI module. The number of master and slaves is not fixed it is created dynamically every time the simulation runs. Two for-loops were devised to loop through the masters and slaves and check whether the fragment is high or low. One of the problems that were encountered is that functions are static in nature, i.e., they use the same memory stack for the all function/task calls. There may be an ambiguity in these function calls, it can not ascertain that the calls will be working fine, as they use the same stack of memory. Which was the case in this situation during the simulation, the print was incomplete. Automatic in a pass by reference inside of pass by value. Pass by reference means the changes made to arguments of subroutines will be visible outside the subroutine, i.e., during its function calls also. As they use separate stack memory for each function call, pass by reference is made possible. The code below is a snippet from the AMLI module.

```
1 reg Matrix [3][3] ;
2 for(k=0;k<MASTERS;k=k+1)
3 begin
4     automatic int connectedTo [$];
5     for (l=0;l<SLAVES;l=l+1)
6     begin
7     if(CONNECTION[l][k])connectedTo.push-back(l);endend
```

After running the simulation the print-out will be as shown in Fig.18. The figure displays the number of masters and slaves and their connections. It can be noticed that one master is connected to every slave which can only mean that the master is the CPU. Another master is connected to eight slaves which represent the DMA. The print-out match the design documentations. There are two different AMLIs (AMLI0 - AMLI1). On one hand AMLI0 relay on APB protocols and on the other hand AMLI1 relay AHB protocols. The main focus is on AMLI1 since that´s the improved versions where all the connection passes through. The AMLI is the only module that possesses a connection table but the rest of the modules do not, so a new plan had to be devised in order to extract information from the rest of the modules.

```
Name : AHBLiteMultiLayerInterconnect Location: test_CM4SubSystem.u_dut.u_Amli1.u_AMLIWrapper1.u_AHBLiteMultiLayerInterconnect
 Size:          19
 Number of Masters:         19
 Number of salves:         17
MasterName: Name
 Connected to slaves:        0 ,        1 ,        2 ,        3 ,        4 ,        5 ,        6 ,        7 ,        8 ,        9 ,        10 ,        11 ,
12 ,        13 ,        14 ,        15 ,        16 ,
MasterName:
 Connected to slaves:        0 ,        1 ,        2 ,        3 ,        4 ,        5 ,        6 ,        7 ,
MasterName:
 Connected to slaves:
MasterName:
 Connected to slaves:
MasterName:
 Connected to slaves:
MasterName:
 Connected to slaves:
MasterName:
 Connected to slaves:
MasterName:
 Connected to slaves:
MasterName:
 Connected to slaves:
MasterName:
 Connected to slaves:
MasterName:
 Connected to slaves:
MasterName:
 Connected to slaves:
MasterName:
 Connected to slaves:
MasterName:
 Connected to slaves:
MasterName:
 Connected to slaves:
MasterName:
 Connected to slaves:
MasterName:
 Connected to slaves:
MasterName:
 Connected to slaves:
MasterName:
 Connected to slaves:
```

Figure 18. AMLI Sub_System Connection table

A set of different ideas were formulated in order to extract information from the rest of the modules. Starting with searching for tools that can extract the information automatically. There are no tools that can do that kind of extraction right now. Major companies like ARM working on it.

The next idea was to use the functions that were created in order to add the information manually. adding the information manually can for a range of different aspects like the size, number, and size of masters and slaves... etc. The manually added information had to be fixed and at no point, in the future, there will be a change in their values. The code below is a snippet that shows the fact that the cache has one master and two slaves.

```
node myNode= new("Cache");
initial begin
  myNode.location = $sformatf("%m");
    #5;
        myNode.addMaster("CpuMaster: \t AhbMasterCpu ");
        myNode.addSlave("CodeSlave: \t Slave0");
        myNode.addSlave("CodeSlave: \t Slave1");
        mBusStructure.addNode(myNode);
end
```

The hard part was to represent the rest of the connections. It was found that the best choice was to create a signal and force it to propagate through the system and give IDs to the masters and Slaves where the master's IDs will correspond to the slaves which they are connected to. The problem is that creating a new signal and initializing that signal across the modules and TestBench is not the best solution since there are great dependencies, It is not impossible but it requires a great knowledge of the system. A new improvement had to be made rather than creating a new signal we **hijack** an already existing signal and force it to get the connection.

4.4 Signal Force

Signal Hijacking is not an easy task since the signal to be picked has to be used in all the modules to get accurate results. In all the modules HWDATA signal is used. The write data signal propagate to all the modules which makes it a perfect choice.

Commencing with TestBench and initializing a busID, masterID, and slaveID. The ID will be added to the already existing master and slave functions. Creating a for-loop that loops through the masters and slaves giving them IDs. The code below is added to the print function in the TestBench.

```
1   foreach(nodes[i].slaves[k]) begin
2                  $fwrite(file, "SlaveName: %s \n  ID: %d \n ", nodes[i].slaves[k].name, nodes[i
       ].slaves[k].slaveID);
3           end
4
5  foreach(nodes[i].masters1[m]) begin
6
7               $fwrite(file, "MasterName: %s \n ID: %d \n", nodes[i].masters1[m].name, nodes[i].
       masters1[m].masterID);
8            end
9  end
```

The tricky part was the configuration of the modules. Starting with importing the package to import the ID which was initialized in the TestBench. Afterward creating a one dimension array of a varying length depending on the size slaves. Then equalizing the array with busID and using the force statement to a variable to override a procedural assignment or procedural continuous assignment that takes place on the variable. The code below is a snippet from the AMLI module which gives a better understanding of the ID process.

```
1  myNode.location = $sformatf("%m");     //myNode.location = $psprintf("%m");
2   for(z=0;z<SLAVES;z=z+1)
3          begin
4          myId[z] = busAnalyser::busID;
5          myNode.addSlave("SlaveAMLI:     \t SlaveAHBLiteMultiLayerInterconnect", busAnalyser::
```

```
  busID++);
6
7         end
8         force  ahbHWDataSlave= myId  ;
9         #13;
10     for(z=0;z<MASTERS;z=z+1)
11             begin
12         myNode.addMaster1("MasterAMLI: \t  MasterAHBLiteMultiLayerInterconnect \n",
    ahbHWDataMaster[z]);
13 end
```

Subsequently running the simulation in Questa_Sim, it was found that forcing the signal was a success. Fig.19 shows the output from the simulation as it has shown the master have IDs 11, 13, 15, ... 31, which corresponds to the slaves of the Syn_Up_bridge. Which matches the output print file. Fig.20 represents a part of the AMLI connection output from the print file.

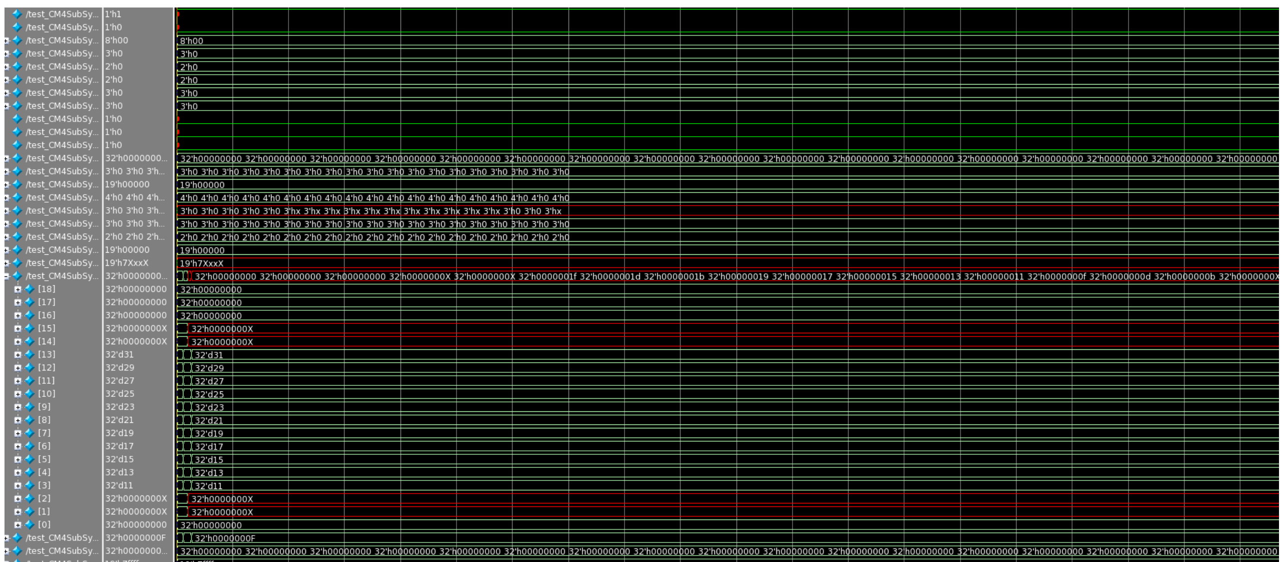

Figure 19. AMLI Sub_System Simulation

```
1538  ID:         11
1539 MasterName: MasterAMLI:      MasterAHBLiteMultiLayerInterconnect
1540
1541  ID:         13
1542 MasterName: MasterAMLI:      MasterAHBLiteMultiLayerInterconnect
1543
1544  ID:         15
1545 MasterName: MasterAMLI:      MasterAHBLiteMultiLayerInterconnect
1546
1547  ID:         17
1548 MasterName: MasterAMLI:      MasterAHBLiteMultiLayerInterconnect
1549
1550  ID:         19
1551 MasterName: MasterAMLI:      MasterAHBLiteMultiLayerInterconnect
1552
1553  ID:         21
1554 MasterName: MasterAMLI:      MasterAHBLiteMultiLayerInterconnect
1555
1556  ID:         23
1557 MasterName: MasterAMLI:      MasterAHBLiteMultiLayerInterconnect
1558
1559  ID:         25
1560 MasterName: MasterAMLI:      MasterAHBLiteMultiLayerInterconnect
1561
1562  ID:         27
1563 MasterName: MasterAMLI:      MasterAHBLiteMultiLayerInterconnect
1564
1565  ID:         29
1566 MasterName: MasterAMLI:      MasterAHBLiteMultiLayerInterconnect
1567
1568  ID:         31
1569 MasterName: MasterAMLI:      MasterAHBLiteMultiLayerInterconnect
1570
1571  ID:          0
1572 MasterName: MasterAMLI:      MasterAHBLiteMultiLayerInterconnect
1573
1574  ID:          1
1575 MasterName: MasterAMLI:      MasterAHBLiteMultiLayerInterconnect
```

Figure 20. AMLI Sub_System Print_out_file

4.5 Text Parsing

The rest of the modules are set in a similar way. All the modules will print to a file called "Analyzer.txt". After the simulation, the file will consist of more than 3000 lines of data extracted from the processor. The data is not unique, which is a result of modules running more than once. This will create a large amount of duplicates. This is redundant and makes it difficult to pinpoint vital data. Since there are a lot of dependencies it is challenging to stop the modules from running more than once. A simple solution is to use text parsing in python to remove all the duplicated data and make the file more structured. The code below removes the duplication. Now after removing the duplication, the file contains around 400 lines which are a big improvement from 3000.

```
1 import hashlib
2 output_file_path = "P:\Workspace\rtl"
3 input_file_path = "P:\Workspace\work\rtl"
4 completed_lines_hash = set()
```

```
output_file = open("outfile.txt", "w")
for line in open("Analyser.txt", "r"):
    hashValue = hashlib.md5(line.rstrip().encode('utf-8')).hexdigest
    ()
    if hashValue not in completed_lines_hash:
        output_file.write(line)
        completed_lines_hash.add(hashValue)
output_file.close()
```

The file still need more organization in order to have a clear format of the output data. Applying text parsing was needed in order to arrange the data. The code below shows how the parsing took place.

```
import csv
with open ('outfile.txt', 'r') as rf:
    with open ('outputsort.txt','w') as wf:
        for line in rf:
            wf.write(line)
bands = list()
with open ('outfile.txt')     as fin:
    for line in fin:
        bands.append(line.strip())
bands.sort()
filename = 'bands_sorted.txt'
with open ('outputsort.txt','w') as fout:
    for band in bands:
        fout.write(band + '\n')
```

4.6 Graph Parsing

Graphical representation was created based on the simulation output and text parsing . The graphical representation was implemented in python. Utilizing the environment for tree exploration (ETE), using python toolkit that assists in the automated manipulation, analysis, and visualization of hierarchical tree structures. The graphs was developed based on the IDs generated from modules during simulation which represents the connections.

The AMLI masters will be connected to Cache, Syn-up-brigde and halter. The figures (Fig.20, Fig.21, and fig.22) below represent the python code output from the ETE for the AMLI connection.

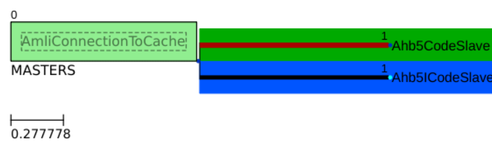

Figure 21. AMLI connection to Cache

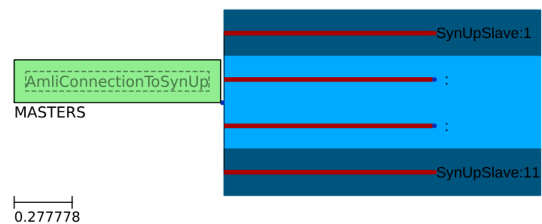

Figure 22. AMLI connection to Syn_up_bridge

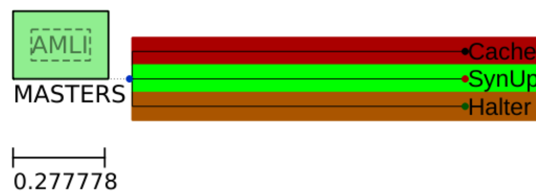

Figure 23. AMLI Connections

A new complete design was constructed based on the RTL extraction. Unfortunately it can't be presented since it contains trade secrets belonging to nordic-semi-conductor. The design shows aspects of the RTL, giving a better understanding about the design. The design was created in python using ETE tool and networkx package in python based on the output from the text file. The design shows the connections of masters and slaves as a substitute to just an abstract segmented. Which was the main focus of this thesis to have a new design that reflect the RTL. The design is based on tree structure where it starts from the cortex and move downwards.

During the simulation, there are some masters and slaves showing no connection which is partially true but since the simulation do not cover every possible scenario it's not accurate enough. thus an address map had to be implemented in order to create a table where it is clear which master can connect to which slave based on the overlapping addresses. This is a time-consuming process thus starting off with just creating an address map for a single particular master and build from there.

4.7 Address Mapping

Building up on the updated design, the next step was to examine a particular master and show all the salves connected to it. This can be achieved by looking at the address range. The overlapping of the ranges will signify that there is a connection. starting with taking a single master and extract/analyze all the overlapping ranges and print out the intersection using python. The code below is a snippet to map-out the intersection between two ranges and print out the intersection to a file.

Usually, during data transfers, the master jumps between different nodes before reaching the destination. Every node has its own set of ranges. Multiple ranges had to be added instead of just two ranges. In order to test that, a connection was established between AMLI0 and AMLI1 by connecting the slave bus signals. Furthermore, test the connection by changing the settings of address. There are some glitches during simulation, not all the slaves were shown. This is mainly because there are some gaps/flaws in the connection between the AMLI0 and AMLI1. Another solution is to create another AMLI from scratch so as to control all the connections and dependencies, but due to the lack of time, this wasn't explored.

```
1 import numpy
2 filename = "set.txt"
3 f = open("set.txt", "w")
4 x = numpy.arange(0, 5000, 1)
5 y = numpy.arange(2500, 4500, 1)
6 #x.intersection(y)
7 #print (intersection(x, y))
8 f.write(str(set(x).intersection(y)))
```

5 Derivations & Conclusion

This section will evaluate and summarize the work achieved and present the results and future work.

5.1 Evaluation & Results

There is a big difference between the design documents/Architectural design and the RTL code. The design only shows an abstracted segmented view of the design. Having only a segmented aspect of the design was the main problem. In order to tackle this problem some modifications were made on the RTL in order to extract the information and build-up and a new updated design that reflect both the design and the RTL. The extraction of the information was implemented in both manual and automatic manner. The manual is based on adding static information about the bus like size, names, and the number of masters and slaves. The automatic is based on the connection and the IDs. The extraction phase had one glitch that was an overflow in one ID belonging to the syn-up bridge module. The occurrence of the overflow was difficult to solve since it required augmenting the test bench. After the extraction phase was complected, the focus was shifted to using the extracted information to create a new design.

The next part was to explore the address range. Focusing on a single master showing all the slaves connection. This is based on analyzing the overlapping address ranges. Moving on to having more than one node, which means having different address ranges. Based on the exploring ranges, a small connection table was created to show all the connection for one particular master.

5.2 Achieved Goals

The set objective at the start of this thesis was to extract bus information from an ARM processor based design. Using the extracted information a new design structure was created. This was the main purpose of the thesis. Moving on to implementing an automatic generation of address maps per master and Verify the correctness of this address map with the architectural specification. The generation of the address mapping was carried out for only one master and based on that a connection table was created.

5.3 Further Work / Possible Improvements

Design flaws like bus looping can be further explored to find a solution and/or reduce the occurrence of loops. Loops occur when data transfer fails, instead of reaching the destination it moves in a circular manner. There is some precaution already in place to prevent the circular manner. One of the precautions is to establish a time frame. Once the time runs out the transfer will be restarted. This is not an ideal solution since the time wasted will impact efficiency. One of the solutions is to keep a list on the master side with all the visited node. Once looping occurs, the master can detect it since the master can already see that this node was visited before.

Since not all masters can contact all slave. It would come in handy to have a database with all the connection and range of addresses. This could be achieved by building on what was implemented for one master but for the entire system. This will come in handy to trace any errors. All the overlapped ranges will signify that a connection can be established between the master and slave.

References

[1] Sudeep Pasricha and Nikil Dutt. *On-Chip Communication Architectures: System on Chip Interconnect*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008. ISBN: 012373892X, 9780123738929.

[2] M. Sgroi et al. "Addressing the system-on-a-chip interconnect woes through communication-based design". In: *Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232)*. June 2001, pp. 667–672. DOI: 10.1145/378239.379045.

[3] Ather. *Buses and Interconnection*. Feb. 2019. URL: https://www.scribd.com/doc/14803250/Buses-and-Interconnection.

[4] F. Karim, A. Nguyen and S. Dey. "An interconnect architecture for networking systems on chips". In: *IEEE Micro* 22.5 (Sept. 2002), pp. 36–45. ISSN: 0272-1732. DOI: 10.1109/MM.2002.1044298.

[5] ARM. "ARM manual. APB-Lite Protocol". In: *Technical Reference Manual* (). URL: https://static.docs.arm.com/ihi0024/b/AMBA3apb.pdf.

[6] ARM. "ARM manual. AHB-Lite Protocol". In: *Technical Reference Manual* (). URL: http://www.eecs.umich.edu/courses/eecs373/readings/ARM_IHI0033A_AMBA_AHB-Lite_SPEC.pdf.