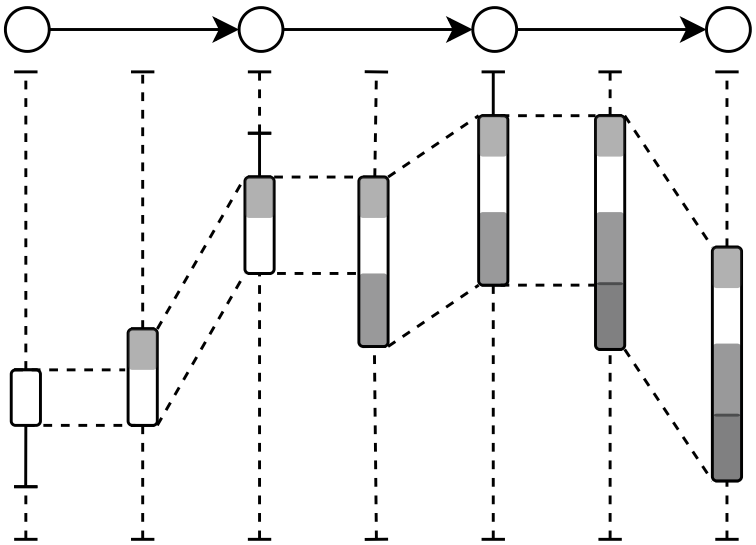
Modeling and Analysis of System-on-Chip Address Maps

Master of Science Thesis

Delft University of Technology



Niels Mook

Modeling and Analysis of System-on-Chip Address Maps

July 17th, 2024

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

EMBEDDED SYSTEMS

by

Niels Mook
born in Leiden, the Netherlands

under supervision of

Arie van Deursen
Soham Chakraborty

Erwin de Kock
Bas Arts

TU Delft
Programming Languages Group
Department of Software Technology
Faculty EEMCS
Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

NXP
System Design & Verification Group
Department of Design Enablement
Chief Technology Office
NXP Semiconductors
Eindhoven, the Netherlands
www.nxp.com

Cover picture: Simplified address-axis diagram example with three shifts and clips.

Modeling and Analysis of System-on-Chip Address Maps

Author: Niels Mook
Student id: 4607279
Email: n.l.c.mook@student.tudelft.nl

Abstract

This thesis presents a methodology for the formal verification of memory organizations in System-on-Chip (SoC) designs described in IP-XACT. The approach involves modeling the address map structures of the design's IP-XACT description and its spreadsheet-based global address map specification into a unified graph model we developed, called an Address Map Graph (AMG). Additionally, it introduces an analysis of AMGs to determine the equivalence of their mapped addresses, called Graph Bitmapping Equivalence (GBE). The methodology was implemented through a series of modular programs integrated into a solution flow. These programs process the spreadsheet memory specifications and IP-XACT design representations into AMGs and perform efficient GBE calculation and detailed reporting. The solution was evaluated using a state-of-the-art mid-size SoC design as a case study. Verification of results was performed using commercially available design analysis tools. The results demonstrated the developed solution was effective to analyze and verify the memory organization of complex SoC designs and to assist in identifying the causes of discrepancies.

Thesis Committee:

Responsible Professor:	A. van Deursen, Faculty EEMCS, TU Delft
Company Supervisor:	E. de Kock, Chief Technology Office, NXP Semiconductors
Company Supervisor:	B. Arts, Chief Technology Office, NXP Semiconductors
University Supervisor:	S. S. Chakraborty, Faculty EEMCS, TU Delft

Preface

This thesis not only marks an end to a nine-month research and internship, but also to my master's degree and education at TU Delft, as to my formal education at large starting in Gouda. At each step, I have strived to reach the maximum of my potential, which made the road challenging but also exciting and rewarding.

I want to thank my daily supervisors from NXP, Erwin de Kock and Bas Arts, for your fantastic onboarding, patience, and support for the entire duration of the internship. You have been my mentors but also closest colleagues for the past nine months. I want to thank you for your critical feedback, which I feel has refined both the thesis as well as myself into something better.

I want to thank my daily supervisor from TU Delft, Soham Chakraborty, for your support, valuable advice and feedback throughout the thesis. I also want to thank Arie van Deursen from TU Delft for your feedback and enablement of this opportunity.

My time at NXP Semiconductors in Eindhoven has been inspirational, educational, and I am grateful for the warm welcome and support I received from Bram Kruseman and the rest of the team. Thank you for your good company in and outside the office.

Finally, I want to express my deepest gratitude to my parents and my brothers, who have encouraged and supported me throughout my studies. Last but not least, I am thankful for the love and presence of my dogs through this period. With fulfillment and this gratitude, I close this chapter of my life with this work and look forward to grow and apply my knowledge further in the coming years.

Niels Mook
Delft, the Netherlands
July 12, 2024

Contents

List of Figures

Chapter 1

Introduction

1.1 System-on-Chips

At the heart of system-on-chip (SoC) design lies the process of integrating diverse intellectual property (IP) blocks – ranging from processors and peripheral interfaces to memory controllers and connectivity modules – into one system. SoC integration enables the creation of highly specialized and optimized systems to meet the high demands of modern embedded applications. Central to the integration of an SoC is its global address map, which specifies the address space allocated to peripheral IP blocks within the system. The implementation of the global address map through mappings of IP memory facilitates the routing of data and control signals across the SoC. Initiator IP blocks initiate data transactions in the SoC. The implemented address maps enable initiator IP blocks in the SoC to access and control peripheral IP blocks by reading from or writing to its designated addresses. Address maps thereby connect individual IPs into a cohesive system.

The timeline of SoC development has strict deadlines to minimize its time-to-market, enforcing software to be developed in parallel with hardware. This requires architects to define the SoC global address map at an early stage in the design process. The process of defining an global address map specification is often ad-hoc in nature. In consequence, the use of unconstrained, non-standardized text documents and spreadsheets for describing the global address map is prevalent among companies [1]. Consequently, the address map must be redefined in standardized formats at a later points in the design flow, in order to realize, verify, and document the address map. While designers and developers try to adhere to the prescribed global address map, its implementation in other formats is currently prone to human error.

1.2 Motivation

During SoC integration, the description of address map implementations plays a critical role. To formalize these descriptions, a standardized format is used, such as IP-XACT (IEEE Standard 1685) [2]. IP-XACT is an eXtensible Markup Language (XML) standard that is widely used to describe the aforementioned SoC integration. Once expressed in IP-XACT, the address maps implemented by the IP-XACT description together form its memory structure, which represents a possible implementation of the global address map specification. A specification describes how peripheral memory should be addressable by the SoC, typically as a single address map. However, the memory structures that implement the specification can be complex. One address map in the specification may be realized by multiple address maps in the implementation, allowing for intermediate address maps to take any form and paths through the design, as long as the addresses are mapped according to specification.

To avoid bugs in SoC products, it is essential to thoroughly verify the IP-XACT descriptions for SoC designs, especially given the complexity and often manual construction. Typical verification methodologies today construct complex testbenches and run elaborate simulations to perform the functional verification of an integrated SoC design. Such methodologies take a significant amount of time, while not verifying the correctness of implemented address maps directly. Instead, the correctness of the design is verified based on its correct functioning. A direct and formal verification method of address maps implemented by IP-XACT descriptions against their global address map specification could improve efficiency in this context by providing immediate verification of implemented address maps before functional verification or even during IP-XACT development.

1.3 Objectives

This thesis aims to find a methodology for the formal verification of address map implementations in IP-XACT descriptions (implementations) against their global address map spreadsheet (specification) in order to improve the efficiency of the current verification methodologies. To do this, this thesis addresses the following four research questions:

- **RQ0**: What unified data model represents both a memory implementation and specification, and enables the comparison of their address maps?
- **RQ1**: What algorithms process a memory specification XLS file into the unified data model?
- **RQ2**: What algorithms process the IP-XACT files of an SoC into the unified data model?
- **RQ3**: What algorithms check any two unified data model instantiations for equivalence in address maps?

1.4 Contributions

The main contributions of this thesis are:

- Definition of a unified data model, called the *Address Map Graph* (AMG) to represent address maps of both memory implementations described by IP-XACT and memory specifications outlined in XLS files. This includes the definition of the *Address-Axis Diagram* (AAD) format to illustrate the path of an AMG; a *bitmapping* representing the mapping of contiguous bit addresses between two nodes in the AMG; and the equivalence in mapped addresses between two AMGs, called *Graph Bitmapping Equivalence* (GBE).
- Implementation of a parser to process memory specification XLS files into the AMG data model.
- Development and implementation of an algorithm to transform hierarchical IP-XACT descriptions into the AMG data model.
- Development and implementation of an algorithm to determine GBE between an implementation and specification AMG and to generate a text-based report of found bitmappings equivalences and non-equivalences for debugging purposes.
- Integration of all algorithm implementations into a solution flow that is modular for extension purposes to other specification and implementation standards.

1.5 Related Work

Only a few studies have been conducted on the topic of formal memory map description, including its modeling and visualization. Besides these, we will discuss the standards that have been established for functional SoC verification and formal memory description.

1.5.1 UVM

The Universal Verification Methodology (UVM) [3], [4] is currently the most popular [5] methodology for the functional verification of SoC designs [6], [7]. However, UVM suffers from two drawbacks regarding address map verification:

Firstly, the verification of address map implementations against their specifications using UVM takes significant time, involving the construction of complex testbenches and elaborate simulations. Furthermore, the verification consists of development-simulation-debugging-coverage cycles [5], which can take substantial time in the SoC design process. Reducing the duration of these verification cycles is crucial to achieve efficiency in this methodology.

Secondly, the debugging process is slowed down due to the ambiguity of the origin of the bugs it finds. To perform the verification, UVM generates a register model from the design's IP-XACT description that models the memory state of the SoC under test. This model is then used as reference to perform tests and verify its state under different circumstances. However, a found bug may be caused by an error in the design behavior, but it may also be caused by an error in the generated register model. An incorrectly generated register model may be caused by discrepancies between the global address map specification and the implementation by the IP-XACT description.

Our proposed formal verification could shorten the verification cycles, by indicating inconsistencies between specified and implemented address maps instantly compared to UVM. As such, bugs could be found before testbench construction or even during IP-XACT development. This could result in less bugs found in the verification cycles, and thus shorter debugging steps and shorter cycles.

Additionally, our proposed formal verification provides assurance of the correctness of the generated UVM register model. This is achieved by the verification of the memory organization implemented by the IP-XACT description from which the register model is generated. When the implemented memory organization conforms to the specified global address map, then there is a higher assurance that the register model generate from it contains less bugs, and thus that any bugs found by UVM are not due to errors in the register model, but errors in the IP-XACT functionality.

1.5.2 Kactus2

Pekkarinen, Teuho, and Hämäläinen have developed a method that is most relevant and comparable to the modeling aspect of this research; they have developed a graph-based method for analyzing the memory layout in IP XACT design hierarchies. They have built upon the foundations laid by Kamppi et al., who have developed a graphical electronic design automation (EDA) tool for intuitive use of the IP-XACT standard, called Kactus2 [8], [9]. Pekkarinen et al. [1] have developed a method of interpretation, representation, and visualization of the memory map information of an IP-XACT design into a graph model to analyze and edit the memory organization. This included an algorithm to traverse the entire design and generate the corresponding graph elements and properties. In the developed graph model, nodes represent interfaces of the IP block, while edges represent their connections through various IP-XACT constructs. Our graph model developed in this work differs fundamentally in that nodes represent the memory elements of the IP blocks rather than their interfaces. Furthermore, our research focuses on the automated and formal verification of IP-XACT design

memory organization, rather than on visualization and editing. Finally, we hope to provide our solution as a verification tool that can be integrated in various workflows beside Kactus2.

The visualization of the resulting model has been improved further by their continued effort of creating a method of visual data compression [10]. This has resulted in a compressed version of the original visualization, to improve the clarity and ease of viewing and editing the memory structures in Kactus2. In contrast, our research visualizes memory structures in the form of directed graphs using the DOT format with GraphViz.

1.5.3 SystemRDL

For the purpose of formally describing an SoC global memory map, SystemRDL has emerged as a capable framework and language [11]. It is able to describe and manage the top-level IP global memory map, including a comprehensive register hierarchy, specifying details from individual bitfields to larger constructs like contiguous memory blocks. Additionally, SystemRDL allows the definition of access policies and constraints. As such, the capabilities of SystemRDL to describe the global memory map is more powerful than that of IP-XACT[1].

It is possible to specify an HDL path for a memory description in order for the SystemRDL environment to access and potentially verify it. No public implementation of this verification process is known. Furthermore, the verification is performed against the HDL, while the focus of this research lies on IP-XACT.

For further integration into the SoC design flow, SystemRDL compilers and parsers are introduced. PeakRDL [12], [13] is an open-source implementation of such a SystemRDL transpiler to and from SystemVerilog, C header files, and IP-XACT. The PeakRDL implementation shows, however, that this translation to and from IP-XACT only deals with a single IP-XACT component comprising of multiple addressBlock elements. This level of implementation is insufficient for complex or hierarchical IP-XACT designs involving multiple IP instantiations.

1.5.4 CMSIS-SVD

Another relevant standard to describe SoC memory maps, particularly for ARM Cortex microcontrollers, is CMSIS-SVD [14]. It is an XML standard that provides a standardized way to describe the system and peripheral registers of a microcontroller design, akin to SystemRDL and IP-XACT for general SoC designs. Similar to SystemRDL, it is able to define the global memory map from course grained device or peripheral regions down to bitfield level. Regarding its integration into the SoC design flow, it is able to generate C header files to aid in software development and could be used to perform verification. However, similar to UVM, this involves a dynamic functional verification which requires significant effort to setup. Instead, our approach aims for an automated and formal verification.

1.6 Overview

In the following Chapter 2 we give an introduction to the aspects of IP-XACT relevant to this research, after which we explain the data model and its properties we created. Finally, we use this data model to describe the formal problem statement at the end of the chapter.

Chapter 3 will provide the solution that we found to the formal problem. We explain how this solution was implemented, including its algorithms, structure, and general workflow.

Chapter 4 will then provide the methodology and results of evaluation of the solution against a complex SoC design. Alongside this evaluation, we explain the structure of the solution's generated report, and causes to the types of results we identified in this evaluation.

[1] https://peakrdl-ipxact.readthedocs.io/en/latest/exporter.html#limitations

Finally, Chapter 5 will summarize the thesis project and provide a detailed discussion of our found results. It will evaluate the effectiveness and shortcomings of the model, the solution, and its implementation to realize the asserted objectives. Furthermore, it will discuss research directions for future work.

Chapter 2

Problem Statement

Verification can be defined as checking whether an implementation adheres to a specification. In the context of this research, a *memory specification* is defined as the layout of a component's memory that is predefined at the early stages of IP design flow. Its source file is a spreadsheet that maps address blocks and peripheral memory to an address offset and range for one or more root components. After the IP-XACT description of an IP design is completed, it has its own memory layout dispersed through its components, called the *memory implementation*. With these two address map descriptions, we define verification as the matching of a memory implementation against the corresponding memory specification.

The comparison of a memory implementation with its specification is simplified when both are represented in the same data structure. A network of address maps lends itself well to be represented in a directed graph $G = (V, E)$ named a address map graph (AMG). When the AMG represents a memory specification, it is called a *specification graph*. When it represents a memory implementation, it is called an *implementation graph*. The definition and construction of both AMG types will be elaborated first. Then the general operations and methodology of comparison and verification will be explained. Finally, code of the generators that implement these operations are discussed.

2.1 IP-XACT (IEEE Std 1685)

This section provides an overview of the general organization of the IP-XACT standard [15] which enables designers to describe reusable IP blocks and system designs in a format that is agnostic to vendor, implementation language, and used tooling. To realize these system designs, IP-XACT facilitates the description address maps between IP instantiations. In turn, these designs can be assembled into new IP blocks, or complete products if forwarded further along the design flow.

Given that IP-XACT is an extensive standard, this section focuses specifically on aspects related to address mapping. We begin with the general structure and objects of an IP-XACT design implementation, then explain the memory structures that can be created using IP-XACT, and finally, we explain the addressing details of these memory structures.

2.1.1 General Design Structure

The IP-XACT standard abstracts the integration of SoC design into a structure of interacting objects. This section outlines the general structure of IP-XACT and how the objects and their elements realize a hierarchical design description. Some of the discussed objects are shown in Figure 2.1 [15, p. 4], which indicates the reference of one object to another with an arrow. We will focus on objects that are relevant to creating IP-XACT designs and to the research conducted in this work.

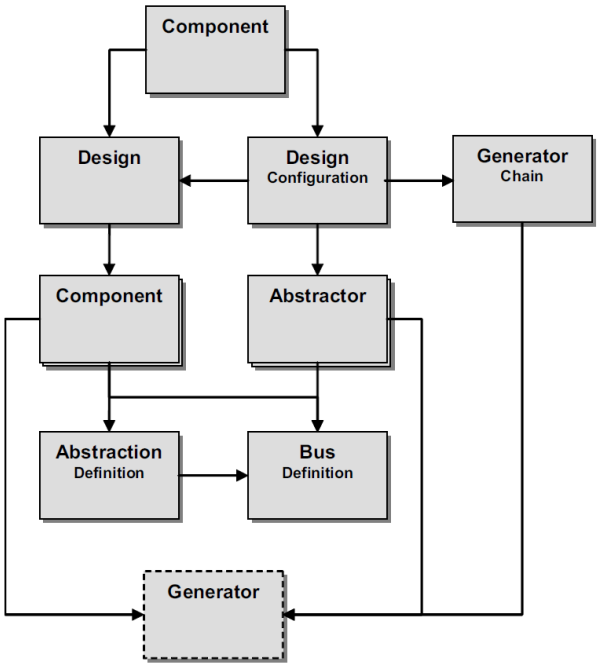


Figure 2.1: IP-XACT object references

2.1.1.1 Component Objects and Views

An IP block is abstracted into a `component` object, also called "component" in this work. It describes the metadata of an IP block such that it can be integrated into larger designs. A `component`contains `view` elements that define the implementation of the component. For a behavioral component, its register-transfer level (RTL) `view` elements contain the HDL files that implement its IP. For a structural component, its hierarchical `view` contains the instantiation of a `design` and `designConfiguration` object.

2.1.1.2 Bus Interface Objects and Interaction

Components may define bus interfaces, also called "interfaces" in this work. Interfaces are used to interact with other components. These interactions can involve communication protocols (e.g. I^2S), interrupt signals, hierarchical implementations, and address maps. As such, the interfaces have properties and attributes that adhere to their communication requirements. These properties and attributes are defined in their referenced *bus definition* and *abstraction definition* elements. Furthermore, the *interface modes* determines the role each interface has in the interaction, such as initiator or the target of communication.

2.1.1.3 Design Objects

A `design` object defines the internal structure of a structural component; it defines the configuration of and interconnection between `component` instances, later to be packaged into a new `component`. Essentially, they describes `component` instances and the interconnections between their bus interfaces.

2.1.1.4 Design Configuration Objects

A `designConfiguration` object defines the active views for each `component` instance within the `design`. This object defines the active view for each `component` instance in a `design`. Note that the `design` and `designConfiguration` are referenced by a view of the enclosing `component`,

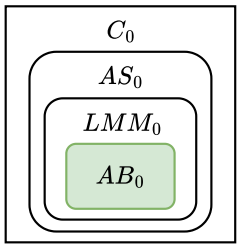


Figure 2.2: Diagram of a local memory map in IP-XACT

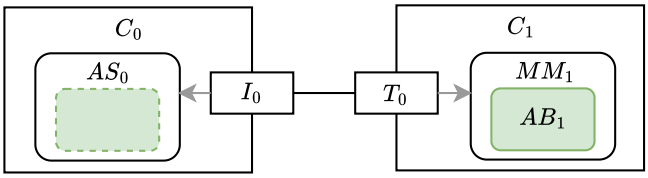


Figure 2.3: Diagram of a memory map in IP-XACT

such that different `designs` are possible per `component`. This flexibility enables the creation of various configurations and hierarchical structures of the same `component` within a single design.

2.1.1.5 Integration

The hierarchical integration of `component` instantiations is realized through the combined use of `design` and `designConfiguration` object instantiations. A top-level `component` references a `design` that describes how multiple lower-level `component` instantiations are interconnected through their bus interfaces. In turn, each lower-level structural `component` may also reference a `design` and `designConfiguration`, creating a multi-level hierarchy, and enabling the creation of complex SoC designs.

2.1.2 Memory Structures

In the following section, we identify and explain the memory structures that can be defined in IP-XACT in order to realize an address map. All the memory that is addressable by a component must be mapped to its address space, defined by an `addressSpace` element. A component may use zero, one, or multiple `addressSpace` elements, describing an address range with addresses starting at 0. Physical memory blocks of components are defined by `addressBlock` elements, which possibly start at a base address offset. A component cannot reference the contents of an `addressBlock` directly. Instead, it must be mapped to one of its `addressSpaces`. IP-XACT offers several memory structures to realize this address mapping: local address maps, (regular) address maps, bridges, and channels. Each will be elaborated in detail.

2.1.2.1 Local address map

By mapping a locally defined `addressBlock` to one of the component's `addressSpaces`, the component may address memory in the `addressBlock`. In this case, the `addressBlock` must be contained in a `localMemoryMap` element, which in turn is contained in one of its `addressSpaces`. This will create a local address map to the `addressSpace`, as illustrated in Figure 2.2, where component C_0 contains an `addressSpace` AS_0. This `addressSpace` contains a `localMemoryMap` LMM_0, with LMM_0 containing one `addressBlock` AB_0. Note that the AB_0 is mapped to its own base address inside AS_0.

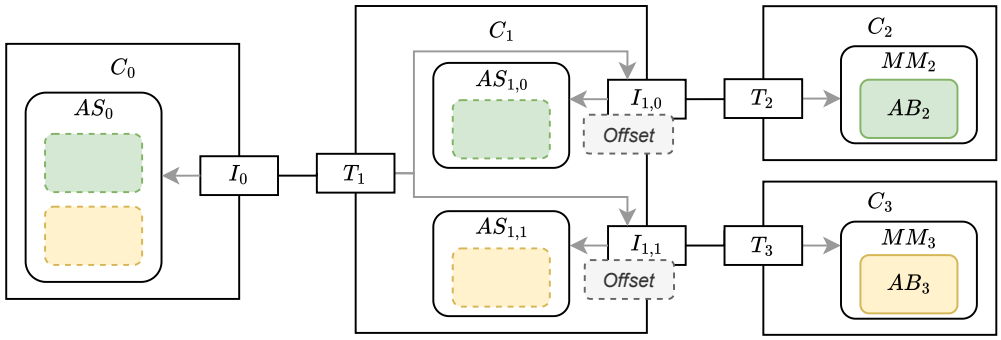


Figure 2.4: Diagram of an IP-XACT construct using a transparent bridge

2.1.2.2 Address map

A component may address memory in the `addressBlock` of another component, such that the `addressBlock` is mapped to one of its `addressSpaces`. In this case, the `addressBlock` must be contained in a `memoryMap` element of an external component. An address map is established implicitly via the connection of bus interfaces. The initiator and target interface modes are most important to create address maps. An initiator interface initiates transactions, while a target interface receives transactions. The establishment of an address map will now be explained according to the example showed in Figure 2.3. An element reference is denoted as a gray arrow. In order to realize the address map, firstly, an initiator interface must reference an `addressSpace`. Secondly, a target interface in another component must reference a `memoryMap`. When the initiator interface and target interface are connected by an IP-XACT interconnect element, indicated by the black line, then the `addressBlocks` of the `memoryMap` become addressable from the `addressSpace`, again under their own base address like for `localMemoryMaps`. This results in the mapping of `addressBlock` AB_1 to `addressSpace` AS_0. The green box with the dashed outline in AS_0 denotes the mapped AB_1 that is now accessible from AS_0.

2.1.2.3 Bridges

It is possible to combine and reorder multiple address mappings into one address map through the use of a bridge. A bridge component contains an element configuration that (implicitly) map multiple `addressSpaces` to a single `memoryMap`. IP-XACT defines two types of bridges – transparent and opaque – which will be explained in further detail.

Unlike a `memoryMap` in an address map, a mapped `addressSpace` defines no base address offset. Instead, this offset is provided by the bridge component for each mapped `addressSpace` individually. This offset causes inbound address accesses to the corresponding `addressSpace` to shift positively or negatively. As a result, the contents of the `addressSpace` are mapped with an offset. This process is called the *shifting* of an address mapping.

2.1.2.3.1 Transparent Bridge

A transparent bridge describes the mapping of (the contents of) multiple `addressSpaces` to one `memoryMap`. Each `addressSpace` is mapped with a unique offset defined in its referencing initiator interfaces. In the element configuration of a transparent bridge partake one target interface and one or more initiator interfaces. The target interface references each of the partaking initiator interfaces in the bridge component. For each of the initiator interfaces, if it references an `addressSpace` then it will be mapped to the `memoryMap`. Note that the transparent bridge does not actually contain a `memoryMap` element. Instead, its `memoryMap` is only *implied* at the target interface. In other words, when this target interface is connected to a component's initiator interface referencing an `addressSpace` then this implied `memoryMap` is mapped to the `addressSpace`.

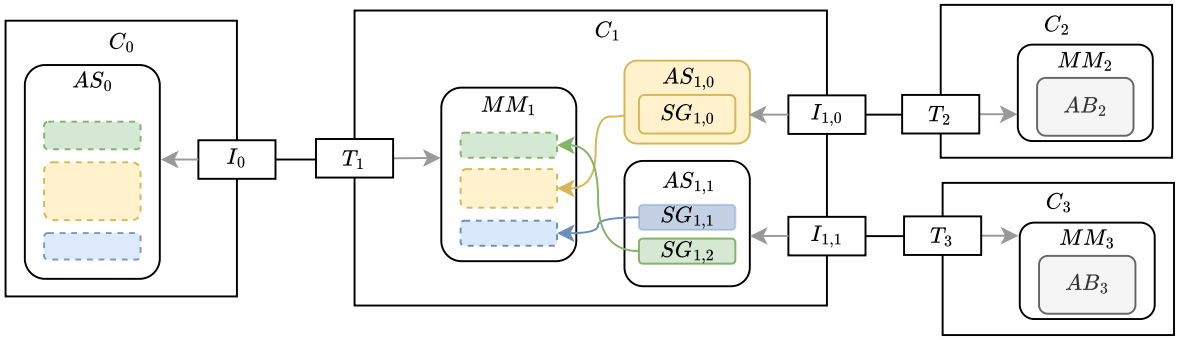


Figure 2.5: Simplified diagram of an IP-XACT construct using an opaque bridge

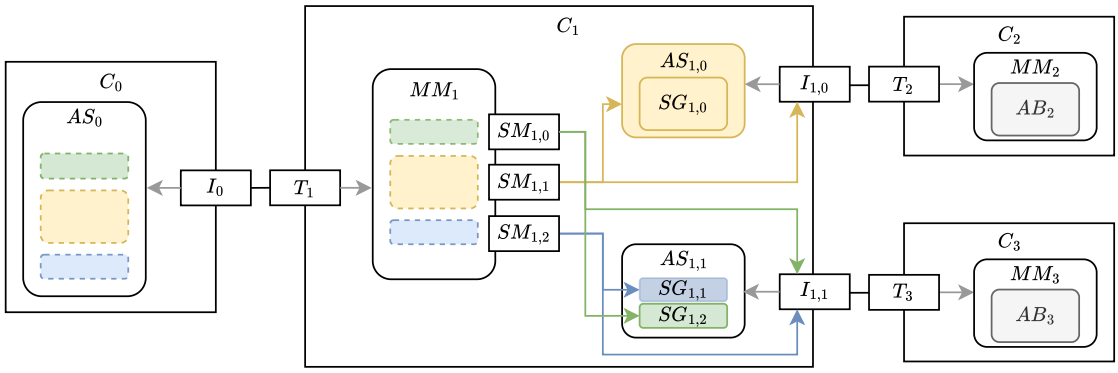


Figure 2.6: Detailed diagram of an IP-XACT construct using an opaque bridge

Figure 2.4 shows an example configuration that implements a transparent bridge C_1, where each initiator interface references an `addressSpace` $AS_{1,i}$, and the implied `memoryMap` at T_1 is mapped to `addressSpace` AS_0 in C_0. As a result, incoming transactions will be able to access their address spaces directly. This direct access implicitly maps shifted contents of the bridge component's `addressSpaces` $AS_{1,i}$ to the destination `addressSpace` AS_0.

Note that when multiple initiator interfaces reference the same `addressSpace`, then each interface maps to its own instantiation of this `addressSpace`, instead of mapping to the same `addressSpace`.

2.1.2.3.2 Opaque Bridge

An opaque bridge can reorganize a set of address mappings even further than its transparent counterpart. This is possible, because an `addressSpace` can define `segment` elements. These are segments of the `addressSpace` that are to be mapped separately from the containing `addressSpace`. They define their own base address offset and address range. See Section 2.2 for more information on their addressing. Figure 2.5 is a simplified illustration of the mapping structure. Via regular address maps, the `addressBlocks` AB_2 and AB_3 are mapped to $AS_{1,0}$ and $AS_{1,1}$ respectively. In component C_1, it shows the mapping of `addressSpace` $AS_{1,0}$ and `segments` $SG_{1,1}$ and $SG_{1,2}$ to `memoryMap` MM_1 by the colored arrows. Finally, the `memoryMap` is mapped to the `addressSpace` AS_0 in component C_0 via an address map.

Contrary to the transparent bridge, the offset at which a segment is mapped is not defined in the initiator interface, but in a `subspaceMap` element defined and contained in the `memoryMap` to which it must be mapped. A `subspaceMap` defines a reference to an initiator interface, a reference to a `segment` or `addressSpace`, and a base address offset. This is illustrated in the more detailed diagram of Figure 2.6 which denotes `subspaceMaps` as $SM_{1,i}$. To establish the mapping, the `subspaceMap` must first reference an initiator interface. For a `segment` mapping, the `subspaceMap` must also reference the `segment` contained in the `addressSpace` referenced

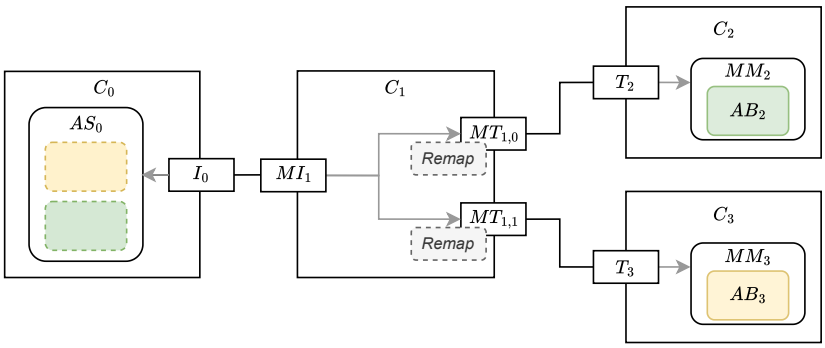


Figure 2.7: Diagram of an IP-XACT construct using a channel

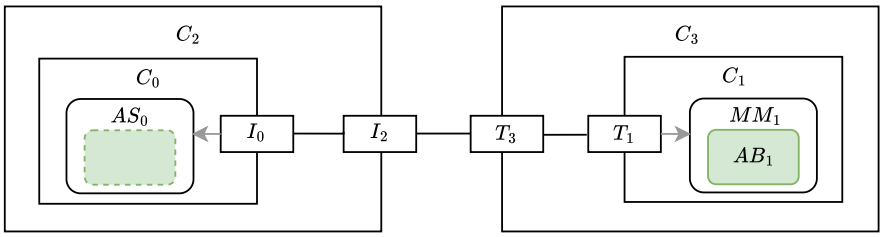


Figure 2.8: Simple address map structure with two hierarchical interconnects

by the initiator interface, as is done by $SM_{1,0}$ and $SM_{1,2}$. For an `addressSpace` mapping, the `subspaceMap` must reference the `addressSpace` referenced by the initiator interface, as is done by $SM_{1,1}$.

Via the opaque bridge, an address map's `memoryMap` can contain `addressBlocks`, but also `subspaceMaps` that map a `segment` or `addressSpace`. In all cases, the `memoryMap`'s contents are mapped to the `addressSpace` referenced by the initiator interface. Therefore, an address map can map `addressBlocks`, `segments`, and `addressSpaces`.

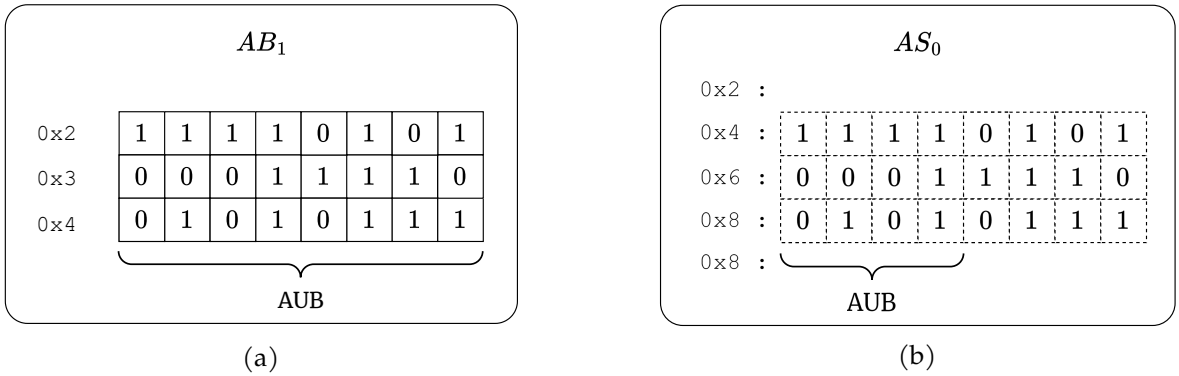
2.1.2.4 Channel

In the context of previous constructs, a *channel* component is essentially a transparent bridge, but without any `addressSpaces` associated with its initiator interface(s). Instead of regular initiator and target `interfaceModes`, they use the *mirroredInitiator* and *mirroredTarget*. Interfaces in these modes may only be connected to non-mirrored initiator and target interfaces respectively. An example configuration that implements a channel is illustrated in Figure 2.7. Inside the component, the mirroredInitiators reference the mirroredTargets, for example $MT_{1,0}$ referenced by MI_1. Incoming memory operations are forwarded from the mirroredInterface to the mirroredTargets. This causes `addressSpaces` and `addressBlocks` accessible through the connected target interface to be mapped to the `addressSpace` referenced by the connected initiator interface.

The mirroredTarget interfaces can provide a remap address. This remap address offsets the address of the incoming memory accesses. Consequently, the `addressSpaces` and `addressBlocks` that are accessible through the mirroredTarget are mapped with the offset defined by its remap address.

2.1.3 Hierarchical Interconnects

An IP-XACT design is a hierarchical description of an IP-block. This means that it consists of IP-blocks, which in turn may also be hierarchical. For each instantiation of a hierarchical component, its child components are also instantiated. The hierarchy is implemented by *hierarchical interconnects*, which connect an interface of a child component to an interface of



(a) (b)

Figure 2.9: Example contents of (a) an `addressBlock` and (b) an `addressSpace`

the parent component of the same type. For example, a hierarchical connection may connect the initiator interface of a child component to the initiator interface of the parent component.

Any interconnection used in previously explained constructs may be followed or lead by any number of hierarchical interconnects. This is illustrated in Figure 2.8. The source and destination memory elements of a mapping construct may reside in components at different hierarchical levels.

2.2 Addressing

The addressing of `addressSpaces` and memory in `addressBlocks` depends on their properties and their context. Memory addressing is understood as the way (groups of) bits are accessed by an address in an `addressBlock`, `addressSpace`, or `segment`; how many bits per address; under what address offset; starting from which address; for how many addresses. The context of an IP-XACT element is defined by the properties of its parent element and (in)directly referenced elements. The addressing may be further delimited by a semantic consistency rule (SCR) as defined by the IP-XACT standard [2, Annex B]. This section explores the fundamentals of this addressing for the memory structures provided by Section 2.1.2.

2.2.1 Addressing Properties

The addressing of `addressBlocks` and `addressSpaces` in address mappings depends on several of their properties and of their parent element.

Each `addressBlock` defines a row width, base address offset, and address range. It makes use of an address unit bits (AUB) value, which defines the number of bits which are addressed by one bit increments of the address. The AUB may be defined by its containing `memoryMap` for an address map, or by its containing `addressSpace` for a local address map. If it is not defined, it is presumed to be 8 [2, p. 222]. The width is a multiple of the AUB in accordance with SCR 8.1 [2]. The range is defined in AUBs. An example `addressBlock` is illustrated in Figure 2.9a, where the AUB is 8, the width is 8, base address is `0x2`, and the range is 3. In accordance with SCR 8.4 [2], `addressBlocks` in the same `memoryMap` may not overlap.

Each `addressSpace` defines a row width and an address range. It may also define an AUB value, else it is presumed to be 8. Again, its width is a multiple of its AUB, and its range is defined in AUBs. An example `addressSpace` is illustrated by Figure 2.9b, where the AUB is 4, the width is 8, and the range is 10. Let us assume that the `addressBlock` AB_1 of Figure 2.9a is mapped to `addressSpace` AS_0, via the memory map of Figure 2.3. The mapped memory bits of AB_1 are shown in AS_0 with a dashed outline. Any defined `segments` must fall within the range of the containing `addressSpace` in accordance with SCR 9.8 [15].

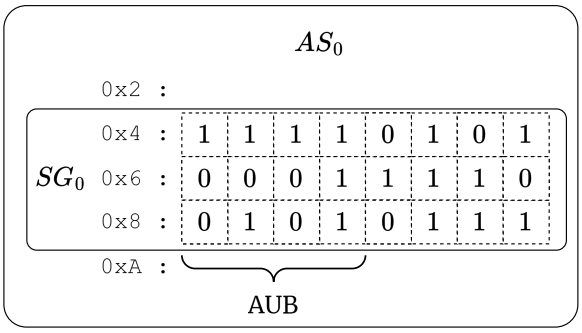


Figure 2.10: Example contents of a `segment` defined in AS_0 of Figure 2.9b

Each `segment` defines a base address offset and an address range, while it inherits the address width and AUB from its containing `addressSpace`. An example `segment` is illustrated by Figure 2.10. Note that the boundaries of a `segment` are irrespective of the boundaries of mapped `addressBlocks`. Therefore, (parts of) multiple `addressBlocks` may be mapped to one `segment`.

2.2.2 Address Resolution

The contents of memory may not remain at the same address when they are mapped to an `addressSpace`. Beside offsets introduced by bridges and channels, the AUB of a mapped `addressBlock`, `addressSpace`, or `segment` may differ from that of the destination `addressSpace`. The resolution of an address to its mapped address therefore possibly involves a translation of addresses. In this translation, the range and width of the destination `addressSpace` will always take precedence over those of the `addressBlock` `addressSpace` or `segment` being mapped. Each scenario of altered addressing will now be explained.

Firstly, the width may become smaller or larger. In both cases, it causes the bits to be wrapped to the new width and the content of addresses to shift. This happens because an `addressBlock` is an array of contiguous memory bits that can be addressed in any addressing width. This, however, only affects the number of bits transferred per transaction. A width change does not affect the addressing of mapped memory, for this is solely determined by the AUB and any induced offsets.

Secondly, the AUB may become smaller or larger. With exception of address `0x0`, a smaller AUB increases all memory addresses, while a larger AUB decreases them. For example, in the mapping of AB_1 to AS_0 in Figure 2.9, their widths remain the same, but the AUB is halved in AS_0. Consequently, the base address offset doubles from `0x2` to `0x4`, and the bits originally at address `0x3` are now mapped to address `0x6` and `0x7`.

After memory has been mapped to its new address, it is possible for it to fall outside the accessible addresses of the destination `addressSpace`. Alternatively, it may fall outside the boundaries of a mapped `segment`. One example is the situation where the range of an `addressSpace` is smaller than the range of the mapped `addressBlock`. The addresses mapped to such out-of-range addresses remains inaccessible, which is called the clipping of memory. Note that neither `memoryMaps` nor their `subspaceMaps` define a range. Therefore, they cannot clip memory that is mapped to them. Clipping will be further elaborated in the following chapter.

2.2.3 Bit Addressing

The following chapters use bit addressing to simplify the aforementioned situations of differing AUBs. A bit address in an `addressBlock` or `addressSpace` is calculated by multiplying an address by the appropriate AUB value. The starting address of an `addressBlock`and `subspace-`

`Map`in bit-addressing is given by Equation (2.1) and Equation (2.2) respectively [2, p. 223]. In a similar fashion, the bit addressed range of memory elements is obtained by multiplying its range with the appropriate AUB.

$$\begin{aligned} \text{addressBlock\_bit\_address\_offset} = {} & \text{addressBlock.baseAddress} \\ & \times \text{addressUnitBits} \end{aligned} \tag{2.1}$$

$$\begin{aligned} \text{subspaceMap\_bit\_address\_offset} = {} & \text{subspaceMap.baseAddress} \\ & \times \text{memoryMap.addressUnitBits} \end{aligned} \tag{2.2}$$

The address at the target interface for a transparent bridge is given by Equation (2.3) [2, p. 225]. The address at the target interface for an opaque bridge mapping an `addressSpace` and a `segment` is given by Equation (2.4) and Equation (2.5) respectively [2, p. 225]. Note that the term "initiator" in the equations refers to the initiator interface.

$$\begin{aligned} \text{target\_bit\_address\_offset} = {} & \text{initiator\_bit\_address\_offset} \\ & + \text{initiator.addressSpaceRef.baseAddress} \\ & \times \text{addressSpace.addressUnitBits} \end{aligned} \tag{2.3}$$

$$\begin{aligned} \text{target\_bit\_address\_offset} = {} & \text{initiator\_bit\_address\_offset} \\ & + \text{initiator.subspaceMap.baseAddress} \\ & \times \text{memoryMap.addressUnitBits} \end{aligned} \tag{2.4}$$

$$\begin{aligned} \text{target\_bit\_address\_offset} = {} & \text{initiator\_bit\_address\_offset} \\ & - \text{segment.addressOffset} \\ & \times \text{addressSpace.addressUnitBits} \\ & + \text{initiator.subspaceMap.baseAddress} \\ & \times \text{memoryMap.addressUnitBits} \end{aligned} \tag{2.5}$$

The address at the mirrored target interface of a channel can be derived from Equation (2.6) [2, p. 225]. The `bitsInLau` is an AUB value defined specifically by a `mirroredTarget` interface. The interface's remap address is defined in this `bitsInLau`.

$$\begin{aligned} \text{mirrored\_initiator\_bit\_address\_offset} = {} & \text{mirrored\_target\_bus\_bit\_address\_offset} \\ & + \text{mirroredTarget.baseAddress.remapAddress} \\ & \times \text{mirroredTarget.bitsInLau} \end{aligned} \tag{2.6}$$

2.3 Global Address Map Specification

The memory organization of an SoC design is specified in a global address map, which defines the address space and layout of memory elements within the system. The global address map ensures that all memory elements are correctly addressed and accessible by the initiator IP blocks within the SoC design. In the rest of this research, the organization specified by a global address map is referred to as a "specification".

This research focuses on specifications defined in spreadsheets, which are commonly used because they are intuitive to read and edit during the design process. To integrate this tool into such workflows and utilize existing designs with spreadsheet specifications as study cases for verification, XLS files are used as specification format in this research.

A spreadsheet specification file defines how peripherals are mapped to the address space of initiator IP blocks, such as the CPU, DMA controller, and others. An example spreadsheet

Address	Identifier	Attached Unit	Purpose	Spec Size [kB]	Implemented [kB]
0x00000000	Boot Code	ROM	System Boot Code	524288	524288
0x20000000	RAM	RAM1	Data	524288	524288
0x40000000	IO	FLEXCOMM	Debug	4	0.5
0x40001000		SPI	SPI Interface	1	1

CPU	DMA
ROM CODE	
RAM	RAM
FLEXCOMM	FLEXCOMM
SPI	SPI

Figure 2.11: Example specification spreadsheet in default layout

layout is illustrated in Figure 2.11. This layout defines peripherals row-wise on the left and initiator IP blocks column-wise on the right, specifying the mapped starting address, the mapped range, and to which initiator IP blocks each peripheral should be mapped.

The default layout, as defined by the configuration parameters, is shown in Figure 2.11. On the left side, the layout specifies all peripherals and their potential mappings to the address space of initiator IP blocks. The first column displays the hexadecimal byte-address, while the sixth column indicates the kilobyte-addressed range mapped for each peripheral. The right side columns define the initiator IP blocks, where each is specified by a name in the header, followed by a cell for each peripheral row. A non-empty cell signifies that the peripheral in that row should be mapped to the address space of the initiator IP block. For example, in Figure 2.11, all peripherals are mapped to both initiator IP blocks, except for the first ROM peripheral, which is not mapped to the ”DMA”.

2.4 Graph Definition

In this section, we present the unified graph model to describe the memory organization defined by both the IP-XACT implementation and the spreadsheet specification of the SoC design. This allows us to compare the models of implementations and specifications with each other.

This section starts by identifying the the different mapping types to be modeled, as well as how the specification memory specification is handled. Then we explain the properties of the nodes and edges in our model, as well as properties of the graph as a whole.

2.4.1 Address Mapping Types

To help in further defining the graph data structure, we identify the mapping types of both the IP-XACT implementation and spreadsheet specification of the memory organization.

2.4.1.1 Implementation Mapping Types

The following five types of address mappings to `addressSpaces` are identified for the memory structures introduced in Section 2.1.2:

1. A local address map that maps `addressBlocks` from a `localMemoryMap` to an `addressSpace` without any shifting.

2. An address map that maps the contents of a `memoryMap` to an `addressSpace` without any shifting. The contents can be `addressBlocks`, but also `segments` and `addressSpaces` from bridges.

3. An address map with a transparent bridge, that maps one or more `addressSpaces` to an `addressSpace`, possibly with an address offset defined by the channel’s initiator interfaces under the AUB of the mapped `addressSpace`.

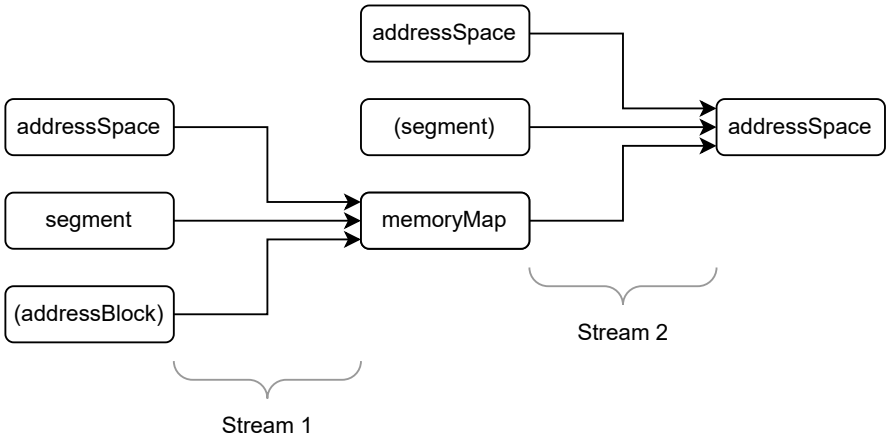


Figure 2.12: Identified memory element mapping streams

4. An address map with an opaque bridge, that maps one or more `addressSpaces` or `segments` to a `memoryMap`, possibly with an address offset defined by the `subspaceMap` under the AUB of the containing `memoryMap`.

5. Mapping type 2, 3, or 4, interjected by a channel. It may map the contents of one or more `memoryMaps` to an `addressSpace` in the case of type 2 and 4. Or, it may map `addressSpaces` to an `addressSpace` in the case of type 3.

It is important to observe that a memory element that is explicitly mapped to an `addressSpace` may end up in one of its `segments`. Whenever any memory element is mapped to an `addressSpace`, then is it also mapped to the `segment` within which it falls, if such a `segment` is defined. Furthermore, whenever an `addressSpace` is mapped, we must assume that all its `segments` are mapped as well. In short, any mappings to or from an AS are also performed to and from its `segments`. In the latter case, note that the segment base offset is not considered, thus not subtracted, so Equation (2.4) applies instead of Equation (2.5).

At first glance, these mapping types show that there are two streams of address mappings shown in Figure 2.12; firstly, from `addressSpaces`, their `segments` and contained `addressBlocks` to `memoryMaps` and secondly, from `memoryMaps` `addressSpaces`, and contained `segments` to `addressSpaces`. Note that child items are in parentheses, as their mapping to the parent memory element is implicit by their child-parent relation. It is observed, however, that `memoryMaps` have significantly different behavior from `addressSpaces` and `segments` as mapping destinations. To elaborate, a `memoryMap` has no clipping behavior like `addressSpaces` and `segments` have, for `memoryMaps` have neither a defined base nor range. The clipping of a `memoryMap`'s contents only happens once it is mapped to an `addressSpace` or `segment` and clipped by its boundaries. It is part of the mapping process, in which `addressBlocks`, `addressSpaces`, and `segments` are aggregated by the `memoryMap` as an implied `addressSpace` before finally being mapped to an actual `addressSpace` or `segment`. In other words, a `memoryMap` can be regarded as a transitory mapping element. Therefore, only `addressSpaces`, their `segments`, or `addressBlocks` are modeled as nodes in the graph, while `memoryMaps` are not. This results in one stream remaining, where addresses in `addressSpaces` can resolve to addresses in either other `addressSpaces`, their `segments`, or `addressBlocks`, shown in Figure 2.13.

2.4.1.2 Specification Mapping Types

As explained in Section 2.3, the spreadsheet specifies how the `addressBlocks` of peripheral IP blocks should be mapped to the `addressSpaces` of initiator IP blocks. This specification

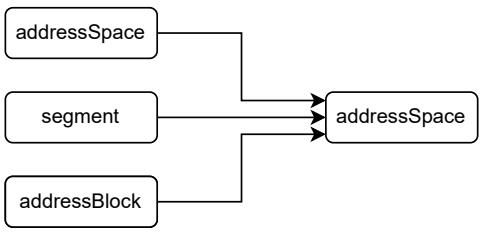


Figure 2.13: Memory element mapping stream processed into the model

includes both the address range and the destination address within the initiator IP blocks. As such, the spreadsheet defines regular address maps analogous to implementation mapping types 1 and 2. Therefore, our model can represent both the implementation and the specification effectively.

2.4.2 Node Attributes

A node $v \in V$ is defined to represent a memory element, which is either an `addressSpace`, `segment`, or `addressBlock`. Therefore, For each memory element, there exists a node and vice versa. As explained in Section 2.1, every memory element has several IP-XACT properties related to their addressing: a base address and an address range, both of which under a certain AUB. This results in a window of addresses that are accessibly by the memory element. Its base address is the first address in its window. Its address range is the number of addresses in its window. Given an IP-XACT element m, assume that its defined base address, range, and AUB – if they exist – can be retrieved through ipBase$(m) \in \mathbb{N}$, ipRange$(m) \in \mathbb{Z}^+$, aub$(m) \in \mathbb{N}$, respectively.

The graph model must also handle the situation where the two memory elements of an address mapping have different AUBs. Given a mapped memory element, this translation may increase or decrease, and thereby shift, the addresses to which this memory is mapped. Meanwhile, the mapping offsets must be applied while dealing with different AUBs, as explained in mapping types. For the purpose of uniformity and simplicity of calculation, all addressing in the graph will therefore be using bit addressing.

$$\text{bBase}(m_0, m_1) = \text{ipBase}(m_0) \cdot \text{aub}(m_1) \tag{2.7}$$

$$\text{bRange}(m_0, m_1) = \text{ipRange}(m_0) \cdot \text{aub}(m_1) \tag{2.8}$$

The processing of a memory element into a node $v \in V$ then involves the calculation of two bit-addressed attributes base$(v) \in \mathbb{N}$ and range$(v) \in \mathbb{Z}^+$. Their calculation involves the multiplication of their non-bit-addressed properties with their corresponding AUB, according to functions bBase(m_0, m_1) and bRange(m_0, m_1) defined by Equation (2.7) and Equation (2.8) respectively. The IP-XACT element arguments m_0 and m_1 depend on the memory element type – `addressSpace`, `segment`, or `addressBlock`:

- Any `addressBlock` AB_0 in `memoryMap` MM_0 is processed into a node $v \in V$ with base$(v) =$ bBase(AB_0, MM_0) and range$(v) =$ bRange(AB_0, MM_0).
- Any `segment` SG_0 contained in `addressSpace` AS_0 is processed into a node $v \in V$ with base$(v) =$ bBase(SG_0, AS_0) and range$(v) =$ bRange(SG_0, AS_0).
- Any `addressSpace` AS_1 is processed into a node v with base$(v) =$ bBase(AS_1, AS_1) and range$(v) =$ bRange(AS_1, AS_1).

Together, base(v) and range(v) define the window $W(v)$, which is a range of bit addresses available to the node. As defined in Equation (2.9), the base is the first bit address in the window, and the range is the number of bit addresses in the window.

$$W(v) = [\text{base}(v), \text{base}(v) + \text{range}(v)) \tag{2.9}$$

2.4.3 Edge Attributes

With the memory elements processed into nodes, the address mappings between them can be processed into edges. Any address mapping between two nodes $u, v \in V$ is processed into an edge $e \in E$. Such an edge is an ordered pair $e = (u, v)$, also shortly denoted as uv. We define the edge source $u = \text{s}(e)$ as the addressing memory element, and the edge target $v = \text{t}(e)$ as the mapped memory element. Each edge has the attribute offset(v) $\in \mathbb{Z}$, which is the address offset it may induce.

The graph edge direction has been chosen to follow address evaluation, instead of the direction of address mapping. Although the direction should not matter in the final verification, this direction was chosen because the mappings in the memory implementation should resolve to the same addresses as in the specification, while the opposite does not hold. In other words, address resolution is a central aspect of the verification process.

The interpretation of IP-XACT address mappings into graph edges depends on the mapping type. As is examined in Section 2.1.2, the mapping type is determined by the constellation of IP-XACT components, their interface interconnections, their interface references, and their contained IP-XACT elements, configurations and types. Given an IP-XACT element m, assume that its defined offset – if it has one – can be retrieved through ipOffset(m). For transparent bridges and channels, this means that the remap address defined in their initiator interface I is retrievable via ipOffset(I). For opaque bridges, this means that the base address defined in their `subspaceMap` SM is retrievable via ipOffset(SM).

$$\text{bOffset}(m_0, m_1) = \text{ipOffset}(m_0) \cdot \text{aub}(m_1) \tag{2.10}$$

Then, the bit-addressed offset(e) property is calculated for each edge $e \in E$ according to Equation (2.10). Mappings of type 5 that involve channels are ignored for now, because channels are rarely used in practice. From the four remaining types of address mappings identified in Section 2.4, we look at how combinations of the first three types are processed into edges. This results in the following scenarios:

- Consider a local address map (type 1), an example of which is shown in Figure 2.2. Assume that the `addressSpace` of a component C_0 contains a `localMemoryMap`, which in turn contains an `addressBlock`. Then, the `addressSpace` and `addressBlock` are processed into nodes u and v respectively. Consequently, edge uv is created with offset(uv) $= 0$.

- Consider simple address map (type 2) where the `memoryMap` contains only `addressBlocks`, an example of which is shown in Figure 2.3. Then, the `addressSpace` is processed into node u, and the `addressBlock` in the `memoryMap` is processed into node v. Then, edge uv is created with offset(uv) $= 0$.

- Consider an address map with a transparent bridge (type 3), an example of which is shown in Figure 2.4. Assume that an `addressSpace` $AS_{1,0}$ – referenced by initiator interface $I_{1,0}$ in component C_1 – is mapped to `addressSpace` AS_0 in component C_0. AS_0 and $AS_{1,0}$ are processed into nodes u and v respectively. Then, edge uv is created with offset(uv) $=$ bOffset($I_{1,0}, AS_{1,0}$).

- Consider an address map with an opaque bridge (type 2 & 4), an example of which is shown in Figure 2.6. Assume that it maps a `segment` $SG_{1,1}$ to a `subspaceMap` $SM_{1,0}$

contained in a `memoryMap` MM_1, all in component C_1. Then, assume that MM_1 in turn is mapped to `addressSpace` AS_0 in component C_0. Then, AS_0 is processed into node u, and $SG_{1,0}$ or $AS_{1,0}$ is processed into node v. Consequently, edge uv is created with $\text{offset}(uv) = \text{bOffset}(SM_{1,1}, MM_1) - \text{bBase}(SG_{1,0}, AS_0)$. The `segment` base must be subtracted because the opaque bridge will map the segment directly to the subspace map's offset, disregarding the `segment` base. Meanwhile, the node that represents the `segment` does contain the base. As such, the base must be subtracted from the edge offset.

- Consider an address map with an opaque bridge (type 2 & 4), an example of which is shown in Figure 2.6. Assume that it maps `addressSpace` $AS_{1,0}$ to `subspaceMap` $SM_{1,1}$ contained in `memoryMap` MM_1, all in component C_1. Then, assume that MM_1 in turn is mapped to `addressSpace` AS_0 in component C_0. Then, AS_0 is processed into node u, and $SG_{1,0}$ or $AS_{1,0}$ is processed into node v. Consequently, edge uv is created with $\text{offset}(uv) = \text{bOffset}(SM_{1,1}, MM_1)$.

2.4.4 Graph Properties

The set of nodes V and edges E, that result from processing all address mappings in an IP-XACT design, together form its graph $G = (V, E)$, called the Address Mapping Graph (AMG). An AMG is directed, possibly a non-tree, and possibly disconnected. When the AMG represents the memory organization of an IP-XACT implementation, it is called an implementation AMG. When it represents the memory organization of a spreadsheet specification, it is called a specification AMG. An implementation AMG may contain cycles, the base case of which is a bridge initiator interface connected to its own target interface. In practice, cycles rarely occur. For this reason, we exclude cyclic AMGs from the scope of this research. Specification AMGs cannot contain cycles in this research. This means that an IP block cannot be specified as both a initiator IP block as well as a mapped peripheral in the spreadsheet. As such, specification AMGs have a height of 1.

AMGs have a set of roots $R \subset V$ without incoming edges, and a set of leaves $L \subset V$ without outgoing edges, as defined by Equation (2.11) and Equation (2.12). A root node always represents an `addressSpace` or `segment`, because an `addressBlock` cannot address itself. On the other hand, a leaf node always represents an `addressBlock`, because `addressSpaces` and `segments` have nothing to address as leaves. As such, R and L are disjunct, such that $R \cap L = \varnothing$. Components may contain multiple `addressSpaces` that are processed into root nodes. Such a component is typically a initiator component, such as a CPU.

$$R = \{r \in V \mid \delta^+(r) = 0\} \tag{2.11}$$

$$L = \{l \in V \mid \delta^-(l) = 0\} \tag{2.12}$$

A path $p = (e_0, e_1, ..., e_{n-1})$ is a sequence of n consecutive edges, where $p(i) = e_i$, such that for each $0 \leqslant i < n - 1$, $\mathsf{t}(e_i) = \mathsf{s}(e_{i+1})$. The set of all possible paths in G, that start in R and end in L, is denoted P, as defined by Equation (2.13). When we take a specific root $r \in R$ and leaf $l \in L$, then we define a root-to-leaf path $p_{r \to l}$ as a path that starts in r and ends in l. Multiple paths may exist from the same root to leaf. As such, the set of all root-to-leaf paths from r to l is denoted $P_{r \to l}$, as defined by Equation (2.14).

$$P = \{p \in \mathscr{P} \mid \mathsf{s}(e_0) \in R \wedge \mathsf{t}(e_{n-1}) \in L\} \tag{2.13}$$

$$P_{r \to l} = \{p \in P \mid \mathsf{s}(e_0) = r \wedge \mathsf{t}(e_{n-1}) = l\} \tag{2.14}$$

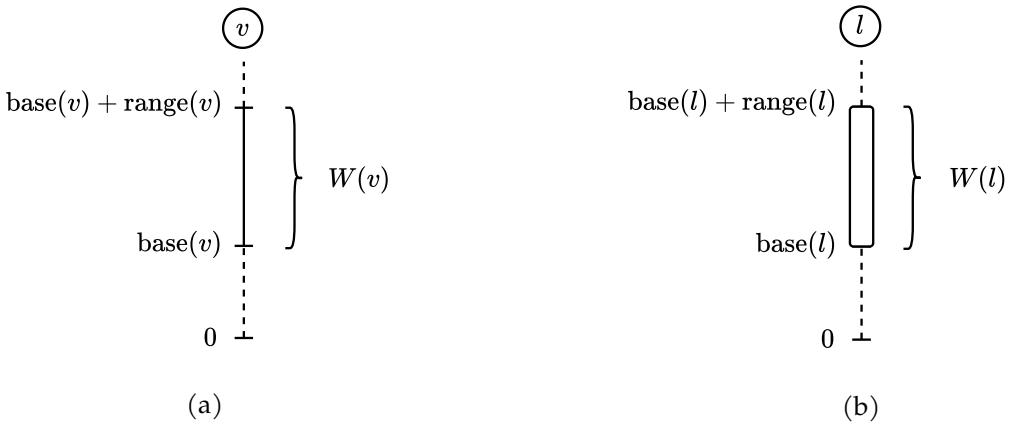


Figure 2.14: AAD of (a) a non-leaf node and (b) a leaf node

2.5 Bitmapping Calculation

With the definition of implementation and specification AMGs completed, we can now move forward to defining their equivalence in terms of address mappings. To achieve this, we first need to analyze and define the concepts of shifting and clipping of a path, as introduced in Section 2.1, in more detail. Next, we will explain the concept of the addressable window of a path. This will allow us to formalize the cumulative clippings along a path. Finally, we will use this formalization to define the concept of a bitmapping, on which we will base the equivalence definition.

2.5.1 Shifting and Clipping

An address mapping can have a clipping effect, shifting effect, or both effects on the mapped addresses of a memory element. A shift occurs when the edge $uv \in E$ of the address mapping has a non-zero offset(uv). Without channels, shifting is only possible between non-leaf nodes, however, shifting of leaf node memory will still be considered in the graph model and the following analysis. This is done for completeness of the model, and possible future inclusion of channels. Mapped addresses are clipped after shifting when they fall outside the window of addressing node u.

2.5.1.1 Address-Axis Diagrams

To aid more detailed explanation of address mappings, and their the shifting and clipping phenomena, they will be visualized using an address-axis diagram (AAD). Figure 2.14a shows the address axis of a non-leaf node v. The dashed line indicates the address axis. A solid line indicates its address window $W(v)$. Figure 2.14b shows the address axis of a leaf node l. Addresses that resolve to an `addressBlock` are visualized as a block. Because l is a leaf node and represents an `addressBlock`, its entire window $W(l)$ visualized as a block.

Figure 2.15 shows an AAD where the window of leaf node l is mapped to node v by an edge e_0. At this point, the AAD must be read from right-to-left, in the direction of address mapping. Later on, we will read the AADs from left-to-right, in the direction of address resolution. The address axis of leaf l is shown on the right. The middle axis is a transitory address axis, unto which the shifted memory of l is projected under offset(e_0). Finally, the shifted memory is mapped to the address axis of v on the left. Only addresses that fall inside the window $W(v)$ are kept, while addresses that fall outside of it are clipped. In this case. In this case, all addresses in $W(l)$ remain addressable.

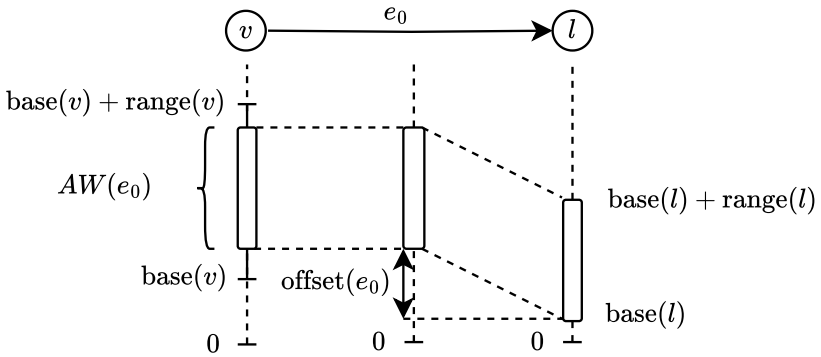


Figure 2.15: AAD of the address mapping of a leaf node

2.5.1.2 Addressable Windows

The concept of addressable windows is first introduced using Figure 2.15, after which the format definition is explained.

An address in an `addressSpace` or `segments` is only addressable when an `addressBlock` address is mapped to it; addresses that do not resolve to an `addressBlock` address are non-addressable. Therefore, v's window in Figure 2.15 has an addressable (sub-)window, denoted $AW(e_0)$.

Leaf nodes represent the `addressBlock`. addressable, however, as leaf nodes cannot address their own memory. Instead, the leaf window must be mapped to the window a non-leaf node u. All addresses in $W(v)$ to which it directly maps will become addressable. This results in a range within window $W(v)$ that is addressable, called the addressable window $AW(e_i)$. Because any addressable window $AW(e_i)$ resolves to `addressBlock` addresses, it is also visualized as a block in AADs.

For any edge e along a path p, the unclipped portion of an `addressBlock` that is mapped to the window of node $s(e)$ is the addressable window $AW(e)$ of that node. For any path p starting in r and ending in leaf l, take edge e_i with $v = s(e_i) \in V$. Then, the addressable window $AW(e_i)$ is the range of addresses of the leaf window $W(l)$ that are mapped to window $W(v)$. This is illustrated by Figure 2.15 for a single edge path.

2.5.1.3 Final Edge Clipping Conditions

To form an intuition in the clipping conditions, consider of the final edge $e_{n-1} = (v, l)$ of a path p, such that $t(e_{n-1}) = l$ and $l \in L$. Then, the bottom mapped addresses are clipped when the condition in Equation (2.15) holds, while the top mapped addresses are clipped when the condition in Equation (2.16) holds.

$$\text{base}(v) > \text{base}(l) + \text{offset}(e_{n-1}) \text{ where } e_i = p(i) \tag{2.15}$$

$$\text{base}(v) + \text{range}(v) < \text{base}(l) + \text{range}(l) + \text{offset}(e_{n-1}) \tag{2.16}$$

An example of such a final edge is illustrated in Figure 2.15. For this example, no clipping occurs because both equations do not hold.

2.5.1.4 Addressable window boundaries

Once an address has been clipped, it will no longer be mapped to further nodes up along the path. Figure 2.16 shows an AAD where the memory of previously used non-leaf node v maps to a node u by an edge e_i. Note how only the addressable window $AW(e)$ is mapped to the window $W(u)$. This is done, because we do not care about the mapping of non-addressable

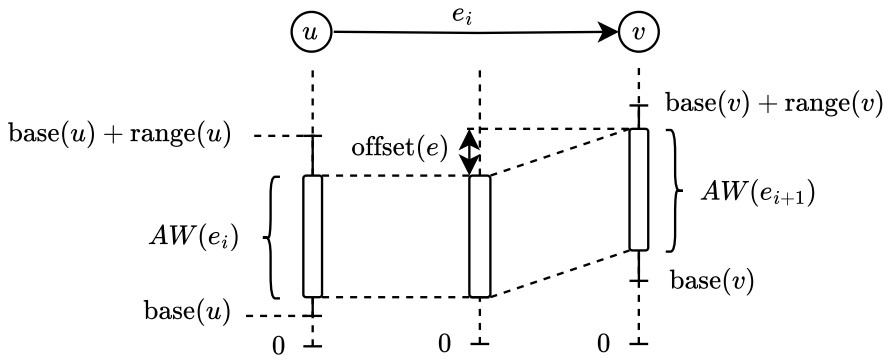


Figure 2.16: AAD of the address map of a non-leaf node

addresses. As such, for any leaf $l \in V$, the size of the addressable portion of $W(l)$ along a path is only able to be clipped or remain equal. The bounds of an addressable window of an edge depend on the bounds of the addressable window of all edges towards the leaf node. Therefore, the definition of the bounds of the addressable window is recursive.

Given any $e_i \in p$, the boundaries of $AW(e_i)$ can be resolved with respect to the remainder of the path down the to the root. In other words, the lower bound and upper bound of the addressable window $AW(e_i)$ must be calculated recursively down to e_0. Given a $p = (e_0, e_1, ..., e_{n-1})$ with edge index $i \in \mathbb{Z}^+$ such that $p(i) = e_i$, the lower bound $\text{lb}(e_i)$ and upper bound $\text{ub}(e_i)$ of $AW(e_i)$ can be calculated recursively according to Equation (2.17) and Equation (2.18) respectively. For each function, the first line shows the base case where $i = n$ and the second line gives the recursive case where $i < n$. Note that no shifting can occur in the base case, for it must be a address map, such that $\text{offset}(e_n) = 0$. Then an addressable window can be defined as in Equation (2.19).

$$\begin{aligned} \text{lb}(e_{n-1}) &= \max\left(\text{base}(\text{s}(e_{n-1})), \text{base}(\text{t}(e_{n-1})) + \text{offset}(e_{n-1})\right) \\ \text{lb}(e_i) &= \max\left(\text{base}(\text{s}(e_i)), \text{lb}(e_{i+1}) + \text{offset}(e_i)\right) \\ &\quad \text{where } e_i = p(i) \end{aligned} \tag{2.17}$$

$$\begin{aligned} \text{ub}(e_{n-1}) = \ &\min\left(\text{base}(\text{s}(e_{n-1})) + \text{range}(\text{s}(e_{n-1})),\right. \\ &\left.\text{base}(\text{t}(e_{n-1})) + \text{range}(\text{t}(e_{n-1})) + \text{offset}(e_{n-1})\right) \\ \text{ub}(e_i) = \ &\min\left(\text{base}(\text{s}(e_i)) + \text{range}(\text{s}(e_i)), \text{ub}(e_{i+1}) + \text{offset}(e_i)\right) \\ &\quad \text{where } e_i = p(i) \end{aligned} \tag{2.18}$$

$$AW(e_i) = [\text{lb}(e_i), \text{ub}(e_i)) \text{ where } e_i = p(i) \tag{2.19}$$

2.5.2 Bitmappings

2.5.2.1 Cumulative Path Clipping

Any $AW(e_i)$ addresses an equal sized region in the leaf node window $W(l)$. It is necessary to calculate the offset this region has from the base address offset of the leaf. The offset can be viewed from the bottom boundary of $W(l)$ called the bottom delta $\text{bd}(e_i)$, or from the top boundary of $W(l)$ called the top delta $\text{td}(e_i)$. The bottom and top deltas are equal to the cumulative clippings of the bottom and top addresses, respectively. For any path $p = (e_0, e_1, ..., e_{n-1})$, they are calculated recursively according to Equation (2.20) and Equation (2.21) respectively. Both deltas are visualized in Figure 2.17 for the final edge e_{n-1} of a path.

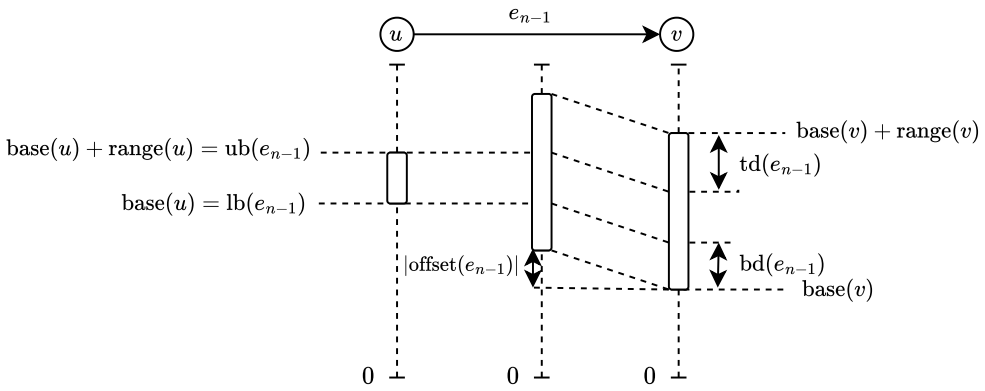


Figure 2.17: AAD a final edge that clips top and bottom addresses

$$\begin{aligned} \text{bd}(e_{n-1}) &= \max(0, \text{lb}(e_{n-1}) - \text{base}(\text{t}(e_{n-1})) - \text{offset}(e_{n-1})) \\ \text{bd}(e_i) &= \text{bd}(e_{i+1}) + \max(0, \text{lb}(e_i) - \text{lb}(e_{i+1}) - \text{offset}(e_i)) \\ &\text{where } e_i = p(i) \end{aligned} \tag{2.20}$$

$$\begin{aligned} \text{td}(e_{n-1}) &= \max(0, \text{base}(\text{t}(e_{n-1})) + \text{range}(\text{t}(e_{n-1})) + \text{offset}(e_{n-1}) - \text{ub}(e_{n-1})) \\ \text{td}(e_i) &= \text{td}(e_{i+1}) + \max(0, \text{ub}(e_{i+1}) + \text{offset}(e_i) - \text{ub}(e_i) \\ &\text{where } e_i = p(i) \end{aligned} \tag{2.21}$$

Symbolic definition of lb in terms of bd and vice versa is difficult, due to the max and min functions. These require further information, or assumption on their resolution (e.g. they always resolve to their second term). Instead, to demonstrate their correctness, we express the codomain upper bound in terms of td and bd, resulting in Equation (2.22). All formulas and this equations are applied an example path in Appendix A.1, which demonstrates their correct representation of the cumulative clippings of top and bottom address.

$$\begin{aligned} \text{base}(t(e_{n-1})) + \text{range}(t(e_{n-1})) - \text{td}(e_0) = \text{base}(t(e_{n-1})) + \text{bd}(e_0) + \text{ub}(e_0) - \text{lb}(e_0) \\ \text{where } e_i = p(i) \end{aligned} \tag{2.22}$$

2.5.2.2 Domains and Codomains

For any root-to-leaf path $p_{r\to l} = (e_0, e_1, ..., e_{n-1})$, the recursive calculations of lb(e_0) and ub(e_0) together define the addressable window $AW(e_0)$, called the *domain* of path $p_{r\to l}$. This domain maps to an equally sized range of contiguous bit addresses of the leaf window, called the *codomain* of path $p_{r\to l}$. The codomain is defined by the recursive calculation of bd(e_0), the leaf node e_{n-1} base base(e_{n-1}), and the range of the domain ub(e_0) − lb(e_0). The lower bound of the codomain is defined as base(e_{n-1}) + bd(e_0), and its upper bound is defined as base(e_{n-1})+bd(e_0)+ub(e_0)−lb(e_0). Thus, the domain and codomain represent a contiguous range of bit addresses in the root and leaf node, respectively.

2.5.2.3 Bitmapping Definition

A bitmapping, denoted as $m(p)$, is formed by the domain and codomain of a path within an AMG. As defined in Equation (2.23) with $e_i = p(i)$, it is a tuple of four values. The first two values lb(e_0) and ub(e_0) are the lower and upper bounds of the path recursively calculated for the path root, which together define the domain of the path. The second two values bd(e_0) and base(e_{n-1}) are the bottom delta recursively calculated for the path root, and the base

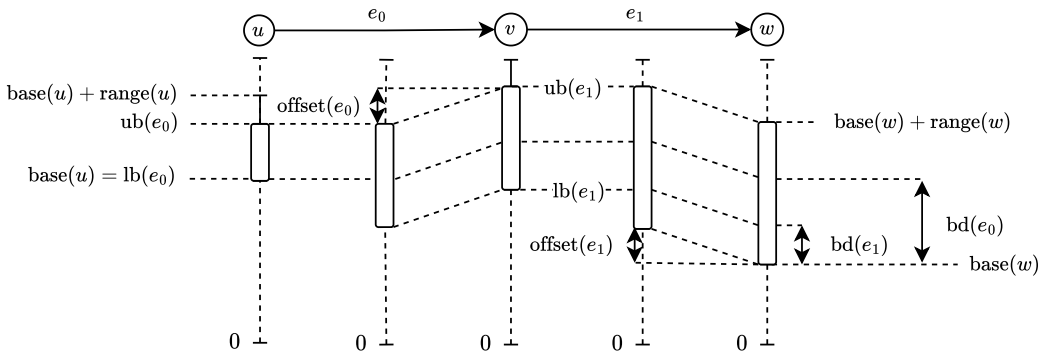


Figure 2.18: AAD of a bitmapping along two edges

address offset of the path leaf, which together with the domain range define the codomain of the path. A bitmapping describes the address map from the domain in the root to the codomain in the leaf of a path. This way, a bitmapping describes the mapping of equally sized contiguous memory regions realized by the path within the AMG. The bitmapping $m(p_{r\to l})$ defines the range of bits in the window of the path root $W(r)$ map, and the range of bits in the window of the path leaf $W(l)$ that map to each other. The domain of a bitmapping is denoted $D(m(p))$. The codomain of a bitmapping is denoted $C(m(p))$.

$$m(p) = (\text{lb}(e_0), \text{ub}(e_0), \text{bd}(e_0), \text{base}(t(e_{n-1}))) \text{ where } e_i = p(i) \tag{2.23}$$

An example two-edge path $p = (e_0, e_1)$ is visualized by Figure 2.18. Both edges introduce only bottom address clippings, and no top address clippings. The resulting bitmapping is $m(p) = (\text{lb}(e_0), \text{ub}(e_0), \text{bd}(e_0), \text{base}(w))$, with its domain $D(m(p)) = [\text{lb}(e_0), \text{ub}(e_0))$ and its codomain equal to $C(m(p)) = [\text{base}(w) + \text{bd}(e_0), \text{base}(w) + \text{range}(w))$.

2.5.2.4 Bitmapping Sets

A graph G can have one or more edges defined in its edge set E. The edges form a network of root-to-leaf paths P throughout the graph. A bitmapping $m(p)$ exists for each path $p \in P$. The set of bitmappings of all paths in P is denoted $M(G)$.

A bitmapping set is also defined for the bitmappings of all paths from any root $r \in R$ to any leaf $l \in L$, denoted $M(G, r, l)$. As such, Equation (2.24) holds.

$$M(G) = \bigcup_{r \in R, l \in L} M(G, r, l) \tag{2.24}$$

2.6 Graph Bitmapping Equivalence (GBE)

In this section, we define Graph Bitmapping Equivalence (GBE) by building on the concepts of bitmappings introduced earlier. We begin by explaining the bitmapping address functions and node mappings. Afterwards, we use these two concepts to define GBE.

2.6.1 Bitmapping Address Function

A bitmapping address function $A(m) : D(m) \to C(m)$ exists for any bitmapping $m(p)$, as defined in Equation (2.25) with $e_i = p(i)$. According to the bitmapping m, its address function accepts a bit address b in its domain and returns the mapped bit address in its codomain. The calculation involves subtracting the domain lower bound from the absolute bit address b to obtain the bit address relative to the domain. Then, addition of the codomain lower bound

results in the mapped address in the codomain. Then, applying $A(m(p))$ to all $b \in D(m)$ results in the codomain $[\text{base}(l) + \text{bd}(e_0), \text{base}(l) + \text{bd}(e_0) + \text{ub}(e_0) - \text{lb}(e_0)) = C(m)$, where $p(0) = e_0$. The mapped address is undefined for bit addresses that fall outside the bitmapping's domain, which conforms to our assumed behavior of memory implementations and specifications.

$$A(m(p))(b) = b - \text{lb}(e_0) + \text{base}(t(e_{n-1})) + \text{bd}(e_0) \tag{2.25}$$

2.6.2 Bitmapping Set Address Function

A bitmapping set address function $A(M(G, r, l)) : W(r) \rightarrow \{N\}$ is constructed for a bitmapping set $M(G, r, l)$. It accepts a bit address in the root's window $W(r)$ and returns a set of mapped bit addresses. As shown by Equation (2.26), the returned set contains the results from the application of all bitmapping address functions of each bitmapping $m \in M(G, r, l)$ for which $b \in D(m)$.

$$A(M)(b) = \bigcup_{m \in M(G,r,l)} A(m)(b) \text{ if } b \in D(m) \tag{2.26}$$

2.6.3 Bitmapping Set Equivalence

Two sets of bitmappings M_1 and M_2 are equivalent when their set address functions are equal, such that $A(M_1) = A(M_2)$. Note that the address function requires the $base$ property of each bitmapping path leaf. Hence, the context of the graph is considered. Bitmapping set equivalence means that two sets of bitmappings implement the same address mappings; from the same bit address in the domain to the same bit address in the codomain.

2.6.4 Node Mapping

Given two graphs G and H, then there exists a node mapping function $B(V_1, V_2)(u) : V_1 \rightarrow V_2$ where $V_1 \subset V_G$ and $V_2 \subset V_H$. As such, multiple nodes in G may map to the same node in H. Typically, this means that graph G is the implementation graph, while graph H is the specification graph, such that multiple implementation nodes may be merged to correspond with one specification node.

2.6.5 Graph Bitmapping Equivalence

Given are two graphs G and H and two bijective maps: $B_R : R_G \rightarrow R_H$ between the roots and $B_L : L_G \rightarrow L_H$ between the leaves. Graph bitmapping equivalence holds between G and H if and only if for all $r \in R_G$ and all $l \in L_G$ Equation (2.27) holds.

$$A\big(M(G, r, l)\big) = A\big(M(H, B_R(r), B_L(l))\big) \tag{2.27}$$

2.7 Formal Problem Statement

The aforementioned definitions allow us to formulate a format problem statement by refining our last three research questions. This problem statement guides the development of the algorithms presented in the next chapter.

1. What algorithms process a memory specification XLS file into an AMG?
2. What algorithms process the IP-XACT files of an SoC into an AMG?
3. What algorithms check any two AMGs for Graph Bitmapping Equivalence (GBE)?

Chapter 3

Solution

This chapter discusses our solution to the aforementioned problem statements and its implementation. The next Section 3.1 provides an overview of our created solution flow and its substeps. Afterwards, each of the following five sections discusses one step in the solution flow. Section 3.2 discusses the method developed to process a global address map specification into an AMG. Section 3.3 discusses the method developed to process the IP-XACT implementation of an SoC design into an AMG. Section 3.4 discusses the program and its algorithms that process two AMGs into an explicit node mapping. Section 3.5 discusses the program and its algorithms that analyzes two AMGs and their node map on graph bitmapping equivalence (GBE). Finally, Section 3.6 discusses the structure and tools used in creating an automated workflow for this solution using a TCL scripting environment.

3.1 Overview

Figure 3.1 shows the typical solution flow with the programs implemented by each section, interjected with the file type of each input and output. A flow performs the modeling and analysis of an SoC design's memory organization. Since the SoC memory organization is described using IP-XACT XML documents it is convenient to implement the flow with IP-XACT generators using the Tight Generator Interface (TGI) API. Generators are programs which may be executed in the IP-XACT design environment (DE), which streamlines the consecutive execution of these generators while simultaneously granting them access to the IP-XACT metadata. As such, the solution programs can be executed consecutively as well as separately.

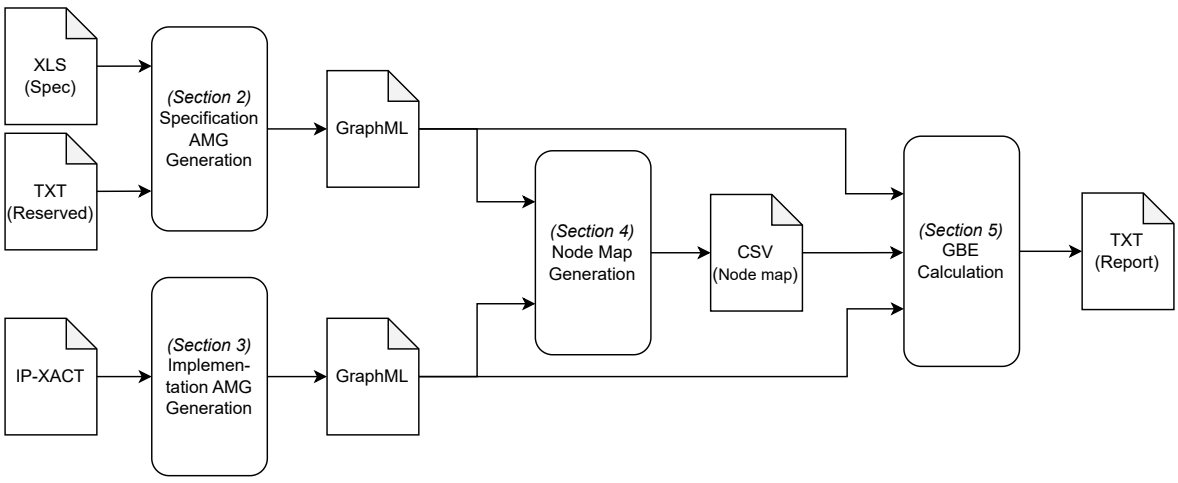


Figure 3.1: Overview of the implemented solution flow

In the explanation of the solution, the following sections use several terminologies. Firstly, unless stated otherwise, you may assume that graph G refers to the implementation AMG, and graph H refers to the specification AMG. The term "graph" is used interchangeably with AMG. Secondly, bitmappings originating from paths in an implementation AMG may be referred to as "implementation bitmappings", the same holds for specification bitmappings.

3.1.1 Tools

We implement all programs in Java 8 to integrate in the IP-XACT workflow using the TGI provided by a vendor Java 8 API. This API follows the IEEE 1685-2009 standard of IP-XACT [15, p. 273]. We parse Excel files using the JXL[1] library version 2.6.10 which was already imported in the IP-XACT environment. We implement graph modeling using the JGraphT [16] library version 1.5.2. We altered it to support JDK 8 from JDK 11. The same modification was made for the JHeaps dependency of JGraphT. The DOT [17], [18] language from GraphViz version 2.30.1 is used for graph visualization. LibreOffice [19] version 7.5.0.3 is used for XLS editing and conversion to and from other spreadsheet formats (XLSM and CSV).

3.1.2 AMG Model Representation and Serialization

AMGs exist inside programs to be constructed, altered, and analyzed, while they are also output from and input to programs in this solution flow. Therefore, AMGs have an in-memory format directly used by the programs, as well as a serialized format used for storage and transmission.

An in-memory AMG is represented by a JGraphT `DirectedPseudograph`. Among JGraphT's other supported directed graph structures, this structure is chosen because it supports cycles, unlike `DirectedAcyclic` graphs or `DirectedMultiGraph`, and does not use weighted edges, unlike `DirectedWeighted`.

We use GraphML as the serialized format for AMGs. It is a generic XML standard supported by JGraphT for export and import. We select GraphML for its ability to define custom node and edge attributes and because it is widely used compared to other supported standards.

JGraphT graph structures can be customized with specific Node and Edge classes to define necessary properties. In accordance with Section 2.4, the node class includes base and range properties, while the edge class includes an offset property. Additionally, the node class has a *name* property for distinguishing the IP-XACT memory element or specification entry. Furthermore, we extend the node class with several identification properties to either store IP-XACT identifiers or GraphML identifiers. We do not extend the edge class with unique identifier because we use JGraphT's default edge retrieval mechanism through their connected nodes.

3.1.3 Test Cases During Development

We developed the solution under continuous testing with a set of test cases. Each test case consisted of a small design for which we had the IP-XACT description and constructed the specification spreadsheet. Some of the designs were created to cover corner cases of the IP-XACT processing into AMGs. These corner cases are the hierarchical instantiation of components and the cloning of `addressSpaces` of transparent bridges. Additionally, the small size of these cases allowed for quick testing of the functionality of each program and the overall solution flow. We provide example results for each program using the largest test case design.

[1] `https://jexcelapi.sourceforge.net/`

3.2 AMG Construction from Global Address Map Specification

One of the initial steps in the solution flow, as illustrated in Figure 3.1, is the modeling of the memory specification as an AMG. The memory specification, described by an XLS spreadsheet, needs to be transformed into an AMG to facilitate further analysis. This section outlines the implementation of a program developed to perform this transformation, detailing the program's operation, configuration, and processing steps.

The program developed for this task primarily consists of an XLS parser. In general, the parser reads the memory specification from an XLS spreadsheet and converts it into nodes and edges, forming a new specification graph H. The following subsections detail the program's functionality, input requirements, and the process of transforming the spreadsheet data into an AMG.

3.2.1 Program Arguments

The program accepts the following arguments:

- *Specification XLS file path*: The path to the XLS file to be processed into an AMG.
- *Reserved words TXT file path*: The path to the TXT file containing all keywords (separated by a newline) that indicate an unmapped entry to be ignored by the parser. The reserved words are required in order to detect reserved memory regions in the spreadsheet, as the words to denote such regions may differ between SoC design. For example, one sheet may use "Reserved", while another may use "-", and yet another may use both.
- *Output GraphML file path*: The path at which the GraphML export should be stored.
- *Spreadsheet layout configuration parameters*: These optional parameters override the default layout by which the supplied XLS file will be parsed. In other words, the program allows customization of its layout configuration through parameters that can be adjusted to match the specific structure of the provided XLS file. These are not described in the thesis.

3.2.2 Configurable Spreadsheet Layout

Our parser processes the spreadsheet layout explained in Section 2.3. We assume the layout to use byte-addresses for all addressing. The layout specifies two ranges per peripheral: a specified range and an implemented range, both defined in kilobytes. The specified range defines the maximum number of byte addresses available to the peripheral before the next peripheral is mapped. The implemented range defines the number of kilobytes that should be mapped by the implementation and should be at most equal to the specified range. The specified range column must be located directly to the left of the implemented range column. Note that no leaf base address offset can be specified in this format, which means that all leaf nodes l have $\text{base}(l) = 0$.

While the parser adheres to this default spreadsheet layout, it remains configurable for deviating layouts to save time in adjusting existing spreadsheets to one layout. We provide nine layout parameters for this layout configuration. These includes the sheet index in the XLS workbook, the header row index defining the column names, the address column index defining the destination addresses of peripherals, the description column index defining the name of the peripheral, the range column index defining the address range mapped for each peripheral, the column index of the first defined root, and the number of roots that follow. This flexibility ensures the parser can handle a variety of spreadsheet layouts.

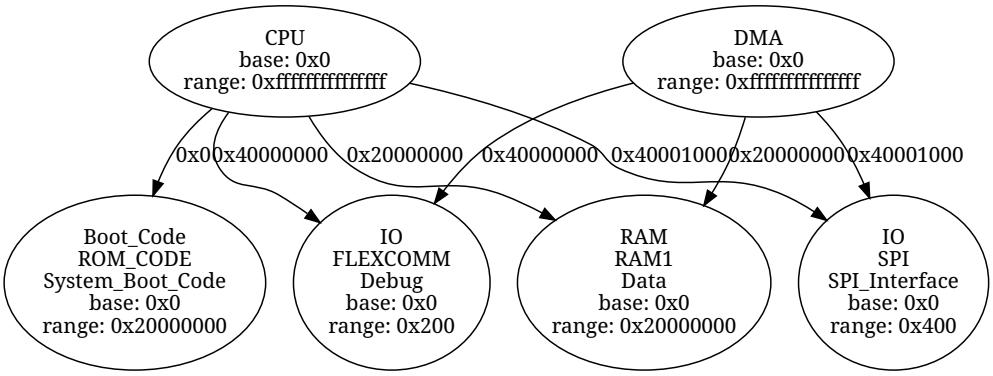


Figure 3.2: AMG graph visualization of an example specification

3.2.3 Construction Procedure

The following subsections describe the specific procedures for processing root nodes, leaf nodes, and edges, highlighting the considerations and calculations involved in each step.

For example, we generate the AMG for the global address map specification of Figure 2.11. This produces the AMG visualized in Figure 3.2. As mentioned before, the graph has a height of 1 for the specification AMG.

3.2.3.1 Root Node Processing

The root nodes are added starting at the specified root column index. For each root, a new node r is created with the specified name in the cell, and properties $\text{base}(r) = 0$ and $\text{range}(r) = \infty$. An infinite range is used for specification root nodes because the range of a specification root is not relevant in our definition of GBE. The root range can be understood as the final mapped peripheral address plus its specified range. Thus, an infinite range ensures no clipping occurs. In this implementation, the maximum value of a Java `long`, $2^{63}-1$, is used instead of infinity, which sufficed for the project duration.

3.2.3.2 Leaf Node Processing

Each row is processed into a leaf node and a set of incident (incoming) edges from the roots. The node name is the concatenated region identifier, unit identifier, and purpose description. If the region identifier is empty, the last defined region identifier is used. Similarly, if the unit identifier is empty, the last defined unit identifier is used. If only the unit identifier is empty while a new region identifier is defined, the empty unit identifier is used. Non-alphanumeric characters are replaced by hyphens to adhere to the naming standard of the DOT language, which will be used later.

To calculate the bit-addressed range of the leaf node, the kilobyte-addressed range in the implemented range column is multiplied by 8192 (the number of bits in a kilobyte). If no number can be parsed, the range from the specified range column is used instead.

For example, assume the last row of Figure 2.11 is processed into leaf node v. Then it will have properties $\text{name}(v) = \texttt{IO\_\_SPI\_\_SPI-Interface}, \text{base}(v) = 0, \text{range}(v) = 8192$.

3.2.3.3 Edge Processing

After the node is added to the graph, we process the specified edges. An edge is added for each root in the root columns with a non-empty cell for the current row. To calculate the bit-addressed offset of this edge, the byte-address in the address column is multiplied by 8.

For example, assume the last row of Figure 2.11 is processed into leaf node v. Then, it will have two incident edges from root CPU and root DMA, both with offset $8 \times 0x40001000$, which is the bit address of hexadecimal byte address 0x40001000.

3.2.3.4 Handling Inaddressable Addresses

When the root accesses an address that is either unmapped or falls outside its range, the behavior is undefined and may result in returning a 0, an error code, or something else. According to our definition of GBE, the root can access any address within its domains. Consequently, the behavior of addresses outside its domains —- whether unmapped or outside its range —- remains undefined in the context of GBE. Therefore, handling of unmapped address behavior is outside the scope of this research.

3.3 AMG Construction from IP-XACT Description

The other initial step of the solution flow depicted in Figure 3.1 is the modeling of the memory implementation as an AMG. Like the memory specification before, the memory implementation must be processed into an AMG, in order to compare them. Unlike the memory specification, which is derived from an XLS spreadsheet, the memory implementation is described by IP-XACT project files that detail the memory structure across various components of an SoC design. This section outlines the steps performed by the implemented program to realize this transformation.

To model the memory implementation, the program must traverse all components of the IP-XACT project that are relevant to the memory structure. In general, it converts all relevant components and their elements into nodes, and their memory constructs into edges. These nodes and edges collectively form a new implementation graph G. The following subsections explain the implementation details of this program, including its arguments, abstractions, and recursive design traversal algorithms.

3.3.1 Program Arguments

The program accepts the following arguments:

- *IP-XACT project identifier*: This identifier is used by the DE to retrieve the project files that describe the design.
- *Output GraphML file path*: The path at which the GraphML export should be stored.

Typically, IP-XACT project files are imported into the design environment (DE) at an earlier stage. We supply the identifier of the top component to the generator. The DE then makes the project metadata accessible to all programs via the Tight Generator Interface (TGI). Figure 3.1 only shows this section's program receiving the IP-XACT files as input. This indicates that this is the only program in the flow accessing the metadata through the supplied project identifier.

3.3.2 Extension of Node Fields with IP-XACT Identifiers

To correctly process the memory organization of an IP-XACT design into the graph model, the IP-XACT elements must be uniquely identifiable from their node representation. The TGI generates an identifier for each IP-XACT entity, also called a reference or ID. These are regenerated at each run, hence the identifier to the same entity may differ between runs. Although the concepts of configured versus unconfigured identifiers [2, p. 345] are important in this implementation, they are omitted from this explanation for simplicity. The TGI can

use an ID, or a combination of several, to retrieve an entity's metadata. This capability is frequently used in the implementations of the pseudocode in this section. For example, it is used to find the memory elements and interfaces in a component or to determine the interface endpoints of interconnections. The obtained metadata is processed into an AMG.

To facilitate this, the extra IP-XACT information is integrated into the same data structure that represents the graph model. This integration involves the following extra node properties introduced for constructing the implementation AMG:

1. `designInstanceID`: ID of the design instance in which the memory element exists.
2. `instanceID`: ID of the `component` instance in which the memory element exists
3. `memoryID`: ID of the `addressSpace` or the `memoryMap` that contains the processed memory element.
4. `elementID`: ID of the `addressSpace`, `segment` or the `addressBlock` that this node represents. When the node represents an `addressSpace`, both the `memoryID` and are the same.
5. `interfaceIDs`: Set of interfaces that link to this memory element.
6. `isSegment`: Boolean that explicitly indicates whether the node is a `segment`.

The `designInstanceID`, `instanceID`, `memoryID`, and `elementID` together are unique for all nodes during implementation AMG construction. These identifiers precisely identify the node's implementation in the IP-XACT design, whose metadata can be retrieved by supplying these identifiers. When the node represents an `addressSpace`, both the `memoryID` and `elementID` are the same. The `interfaceIDs` are unique within the contained `component` instance.

For clarity, the pseudocode algorithms abstract from the retrieval of metadata via IP-XACT identifiers, hence they will be absent in the pseudocode. Futhermore, future analyses on the resulting AMGs do not use these identifiers; they are only used in the AMG construction from the IP-XACT implementation. No identifier property is included in the GraphML as they relate only to the IP-XACT implementation, not the graph model. Instead, the automatic identifier index of GraphML nodes is used for node identification after export. The isSegment property is included in the GraphML because it is necessary in the programs of Section 3.4 and Section 3.5.

Algorithm 1: Implementation Graph Initiation

Input: C_{top}: Top component of design to be modeled
Output: G: Constructed AMG of this design's memory implementation

```
1 createGraph(C_top) {
2     G ← Empty graph
3     interfaceMap ← Empty map between interfaces
4     foreach C'_child instantiated in C_top do
5         createGraphRec(G, interfaceMap, C_top, C'_child)
6     end
7     addEdges(G, interfaceMap, C_top)
8     return G
9 }
```

3.3.3 Construction Procedure

To construct the implementation AMG, we developed an algorithm that recursively explores all components of the hierarchical IP-XACT designs. The pseudocode of this is shown in Algorithm 1. It starts with the top component, initiating a new graph and kicking off recursion

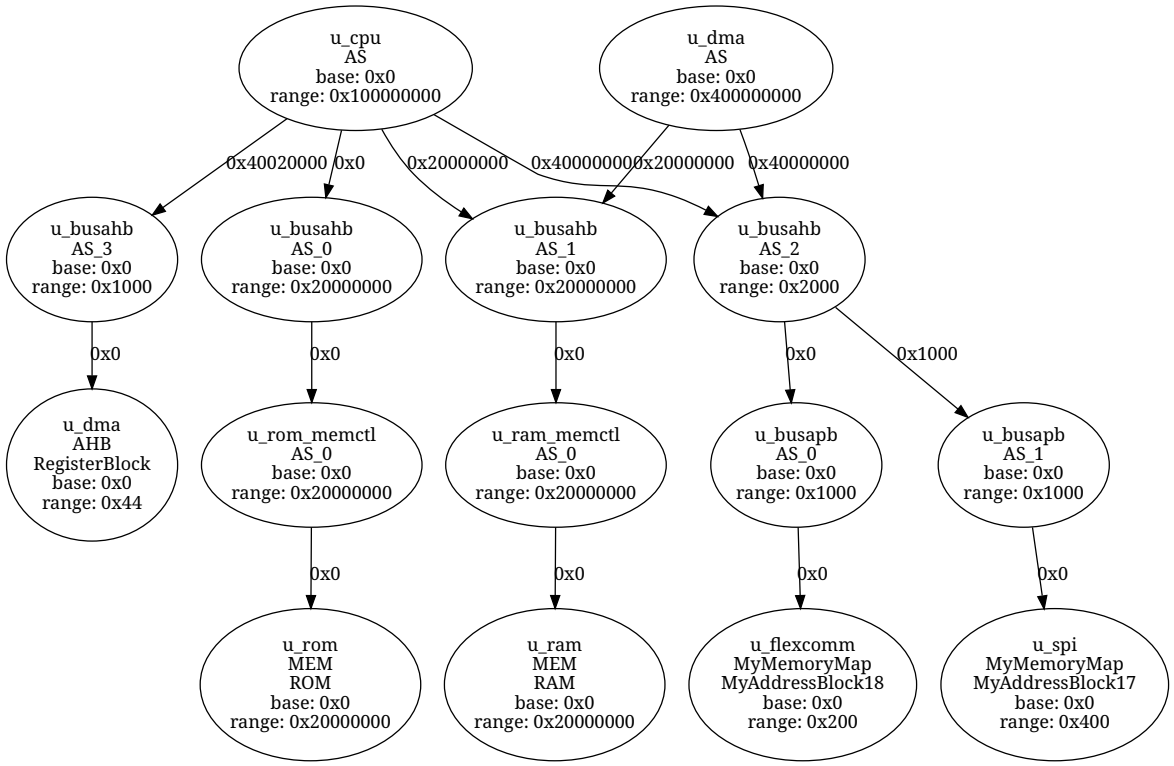


Figure 3.3: AMG graph visualization of an example implementation

for each contained component instance (line 4-6). Our approach is inspired by the C++ implementation of [1] discussed in Section 1.5.2, which uses a general recursive structure. Given a component instance, the algorithm extracts the graph nodes from the component, performs recursive calls for its child component instances, and finally extracts its graph edges. Our Java implementation follows the same recursive structure. As such, AMG nodes are added on the recursive way down the IP-XACT hierarchy, while AMG edges are added on the recursive way back up. The exact procedures to perform each of these steps is explained by the following subsections.

For example, we generate the AMG for the IP-XACT description implementing the specification of Figure 2.11. This produces the AMG visualized in Figure 3.3. As expected, this graph has a height larger than 1 unlike the AMG of the specification in Figure 3.2. The bitmappings are performed via the intermediate address maps.

The following subsections explain each step of the recursive Algorithm 2. Therefore, you may assume that all line references are for this algorithm, unless Algorithm 1 is mentioned explicitly.

3.3.3.1 Node Processing

First, all memory elements contained in the current component are added as nodes to the graph. These nodes can become root nodes, leaf nodes, or intermediate nodes. When an `addressSpace` is added, all its contained `segments` and `localMemoryMaps` are also added with a directed edge pointing towards them with an offset of 0 (line 2-15). All `addressBlocks` inside a `memoryMap` are processed into nodes (line 16-21).

To complete the definition of the nodes, the helper function `addInterfacesToNodes` adds the interface identifiers `interfaceIDs` to the nodes whose corresponding memory element they reference (line 22). This is necessary because each interconnect contains a pair of interface identifiers, so it must be known which nodes in the graph these interfaces belong to.

Algorithm 2: Recursive Implementation Graph Construction

```
   Input: G: the graph under construction
   Input: interfaceMap: map between interfaces
   Input: C_parent: parent component of currently processed component
   Input: C_current: currently processed component
   Output: G: The AMG under construction after processing C_current
1  createGraphRec(G, interfaceMap, C_parent, C_current) {
       /* Process all memory elements in C_current into graph nodes.                    */
2      foreach addressSpace AS in C_current do
3          process AS into node u
4          add u to V(G)
5          foreach segment SG in AS do
6              process SG into node v
7              add v to V(G)
8              add edge uv to E(G) with offset 0
9          end
10         foreach localMemoryMap LMM in AS do
11             process LMM into node v
12             add v to V(G)
13             add edge uv to E(G) with offset 0
14         end
15     end
16     foreach memoryMap MM in C_current do
17         foreach addressBlock AB in MM do
18             process AB into node u
19             add u to V(G)
20         end
21     end
       /* Add the interfaces of C_current to the previously added nodes.                */
22     addInterfacesToNodes(G, C_current)
       /* Map the two interfaces of hierarchical interconnects in interfaceMap.         */
23     foreach hierarchical interconnect l in C_current do
24         Inf_high ← interface of l in higher level component
25         Inf_low ← interface of l in lower level component
26         add map Inf_high ↦ Inf_low to interfaceMap
27     end
28     foreach C'_child instantiated in C_current do
29         createGraphRec(G, interfaceMap, C_current, C'_child)
30     end
31     addEdges(G, interfaceMap, C_current)
32 }
```

When a transparent bridges contains a target interface, each initiator interface on the other end of the bridge requires an additional processing step. When multiple bridge initiator interfaces reference the same `addressSpace`, they each reference their own new instantiation of this `addressSpace`. In other words, the address maps from multiple bridge initiator interfaces cannot be combined into the same referenced `addressSpace`. This is implemented in `addInterfacesToNodes` by creating a new `addressSpace` node clone for each transparent bridge initiator interface, while the original `addressSpaces` nodes are removed from the graph. Removal does not affect non-bridge connections, as their interfaces are added to the original node before cloning.

3.3.3.2 Registering Hierarchical Interconnects

Before the algorithm executes the recursive calls, all hierarchical interconnects defined in the current component are processed into an interface map (line 23-27), as explained in Section 2.1.2. In this map, the interface of the current component points to the interface of the

child component. For example, when the hierarchical interconnects of Figure 2.8 are processed, interface I_2 maps to I_0 while interface T_3 maps to T_1. This map of hierarchical interconnect interfaces is needed to resolve the non-hierarchical interconnects to which they connect.

3.3.3.3 Recursive Call

The recursive call is performed for each child component instance (line 28-30). This call passing along the current graph, the interface map, the current component, and the child component itself. The current component is needed in the recursive call to register the precise location of the node in the IP-XACT implementation by including the parent design instance identifier in its properties.

3.3.3.4 Edge Processing

After the algorithm has performed recursive call, all memory constructs defined in the current component are processed into edges by helper function `addEdges` (line 31). This step is also performed for the initiator Algorithm 1 (line 7). Because all recursive calls have been performed for this component, we know all memory elements defined by child components have also been processed into graph nodes. Additionally, we know all hierarchical interconnects defined by either child components or parent components have been added to the interface map. The rest of this subsection explains how helper function `addEdges` processes interconnects (hierarchical and non-hierarchical) into edges between these nodes.

All non-hierarchical interconnects defined by this component, which connect a target and initiator interface, can now be processed into edges. To do this, the endpoint memory elements of each interconnect must be identified. There may be several hierarchical interconnects between the non-hierarchical interface and the final interface that references the memory element. These hierarchical interconnects are resolved to their final interface using the interface map. The resolution involves exhaustively retrieving the hierarchically-connected, lower-level interface from the interface map until no further entries exist, indicating that the final interface has been found. This process results in the retrieval of the two final initiator and target interfaces for both endpoints of the interconnect chain. For the example in Figure 2.8, resolution of the non-hierarchical interconnect between I_2 and T_3 results in endpoints I_0 and T_1, respectively.

If a bridge does not contain the final target interface, the interconnect implements a simple address map structure that can be processed into an edge directly. An edge with zero offset is added to the graph from each source node to each target node. The source nodes are all nodes that contain the initiator interface identifier, while the target nodes are all nodes that contain the target interface identifier.

If a bridge does contain the final target interface, the interface will not directly reference a memory element represented in the graph. Instead, the bridge constructs are analyzed and processed into edges in the following manner:

- For each opaque bridge, retrieve its initiator interface and all `subspaceMaps` contained in its `memoryMap`. Then, for each `subspaceMap`, retrieve its referenced `addressSpace` or `segment`, and add an edge from each source node to the target node. The source nodes are all nodes that contain the identifier of the final non-hierarchical initiator interface. The target node is the node that represents either the `addressSpace` or `segment` referenced by the `subspaceMap`. The edge offset is calculated as discussed in Section 2.4.3.

- For each transparent bridge, retrieve its initiator interface. Add an edge from each source node to each target node. The source nodes are all nodes that contain the identi-

Graph H			*Graph G*		
Node ID	*Node Name*	*Map Index*	*Map Index*	*Node Name*	*Node ID*
ROOTS					
0	CPU	0	1	u_dma$AS	18
1	DMA	1	0	u_cpu$AS	6
LEAVES					
2	ROM	0	0	u_romMEMROM	5
3	RAM	1	1	u_ramMEMRAM	23
4	SPI	2	2	u_spi$MM1$AS17	4

Table 3.4: Example of the first seven entries of a node map CSV.

fier of the final non-hierarchical initiator interface, while the target nodes are all nodes that contain the bridge initiator interface.

3.4 Node Mapping

After constructing the Address Map Graphs (AMGs) for both memory specification and implementation, the next step in the solution flow, depicted in Figure 3.1, is constructing the node map. This node map is essential for determining the Graph Bitmapping Equivalence (GBE) between two AMGs as defined by Section 2.6. GBE requires two bijective node maps B_R and B_L between the implementation and specification roots and leaves, respectively. These mappings ensure that only the bitmappings of corresponding nodes are compared.

However, the raw state of the implementation AMG does not allow for a bijective map. The implementation AMG may represent a specification leaf by means of multiple leaves, making a bijective map impossible. To resolve this, the split implementation leaf nodes must be merged into one. The node map defines these merges by mapping multiple implementation leaves to the same specification leaf.

This section outlines the steps our implemented program takes to generate the node map. We start by explaining the handling of merged nodes and the CSV format used to define node mappings. Next, we introduce bitmapping maximization, a technique that implicitly implements the address function and enables direct comparison of bitmappings for equivalence. We then demonstrate the correctness and completeness of the algorithms used for bitmapping maximization. Finally, the procedure for generating the final node map CSV is discussed.

3.4.1 Program Arguments

The program accepts the following arguments:

1. *Specification AMG GraphML file path*: The path to the GraphML file describing the specification AMG.

2. *Implementation AMG GraphML file path*: The path to the GraphML file describing the implementation AMG.

3. *Output node map CSV file path*: The path at which the generated CSV file must be stored. It defines the node map between the two AMGs and specifies any leaf nodes to be merged in the implementation AMG.

3.4.2 CSV File Layout

We use a CSV file to provide a clear and structured way to define node mappings. Table 3.4 shows the first seven lines of the node map for one of the test designs, including two indicative headers. The CSV layout is split in two symmetric sides. The left side represents nodes from the specification AMG, referred to as the H-side, while the right side represents nodes from the implementation AMG, referred to as the G-side. Each row, excluding the "ROOTS" and "LEAVES" headers, may display a node entry for either side.

The "Node ID" columns, located at the far left and right, contain the unique identifiers of the nodes, which are automatically generated by the GraphML format. The "Map Index" columns in the middle indicate the mapping of nodes. When two nodes from both AMGs share the same map index, they are considered mapped. It's important to note that the map indices are independent of the node identifiers. If a node's map index cell is empty, it means the node is not mapped and will be excluded from further analysis in the solution flow.

To handle leaf node merging, multiple G-side leaf nodes can be merged if they share the same map index. Merging does not apply to H-side leaf nodes, because our focus lies on verification of implementations against specifications, and it is not possible in our specification model for multiple specification entries to specify a single implementation bitmapping. As such, this makes the merging unidirectional and specific to leaf nodes. Consequently, H-side map indices are always unique, such that H-side nodes may not merge. Although it would be possible, this solution also does not implement the merging mechanism for root nodes, because it was not seen as credible that multiple roots are specified as one root.

3.4.3 Node Mapping Procedure

This subsection explains the steps and their related algorithms for generating a node map. When handling larger designs, their implementation AMGs quickly get too many leaf nodes to map manually. An automated program was developed to complete the map indices for leaf nodes in Graph G, thereby automatically mapping leaf nodes to each other. To implement the automatic mapping of nodes, a preliminary comparison of bitmappings of all root-leaf pairs must be performed. In general, the steps consist of calculating all bitmappings in both AMGs, performing our developed method of bitmapping maximization on both sets, and finally traversing the nodes and generating a node map between their nodes based on bitmapping equivalence.

3.4.3.1 Bitmapping Calculation

Before explaining the method of bitmapping calculation, it is essential to understand the way bitmapping sets are stored throughout this solution. The set of bitmappings $M(G)$ of an AMG G is represented as a two-dimensional map, denoted M_G. Root nodes are in the first dimension, and leaf nodes are in the second dimension. For any root $r \in R(G)$ and leaf $l \in L(G)$, the set $M_G[r][l]$ contains the bitmappings of $M(G, r, l)$, such that $M_G[r][l] = M(G, r, l)$. Throughout this solution, all bitmappings $m \in M_G[r][l]$ are sorted in ascending order of $lb(m)$. This structure ensures that bitmappings are efficiently managed and accessed during the various processing steps.

Algorithm 3 shows the algorithm to calculate M_G. For each root node r and leaf node l pair, it calculates bitmapping $m(p)$ for each path p between them (line 2-8). The bitmapping is recursively calculated (line 8) for the AMG path as explained in Section 2.5. The bitmapping is then added to set $M_{r \to l}$ unique to this root-leaf pair (line 9). Each set $M_{r \to l}$ is then added to the two-dimensional set at $M_G[r][l]$ (line 11). Similarly, M_H is calculated for the specification AMG H.

Algorithm 3: Graph Bitmapping Set Calculation

Input: G: AMG for which to calculate all bitmappings
Output: M_G: Two-dimensional map of all bitmappings of AMG G, with roots in first dimension and leaves in second dimension

```
1  calculateBitmappingSet(G) {
2      M_G ⟵ Empty two-dimensional map from nodes to sets
3      foreach r ∈ R(G) do
4          foreach l ∈ L(G) do
5              if r = l then continue
6              M_{r→l} ⟵ ∅
7              foreach p ∈ P_{r→l} do
8                  calculate bitmapping m(p)
9                  insert m(p) into M_{r→l} in order of ascending lb
10             end
11             add M_{r→l} to M_G[r][l]
12         end
13     end
14     return M_G
15 }
```

3.4.3.2 Address Function Implementation

Section 3.5 explained that GBE between two AMGs is determined by the equivalence of the address functions A of their bitmapping sets for corresponding root-leaf pairs. To determine GBE, it is necessary to determine the equivalent output of both functions for each bit address in all domains of the pairs. However, this approach is inefficient for larger designs, as it requires processing millions of bit addresses. We developed an alternative approach that is more efficient in GBE calculation, called *bitmapping maximization*, which we discuss later in this section.

3.4.3.2.1 Direct Bitmapping Comparison

The GBE definition in Section 2.6 relies on an address function because bitmappings, as they come out of previous step, cannot be directly compared immediately. Figure Figure 3.5 illustrates why direct bitmapping comparison fails. This figure depicts three bitmappings: m_H from specification graph H on the left, and $m_{G,1}$ and $m_{G,2}$ from graph G on the right. Assume there is a node mapping that maps r_G to r_H and l_G to l_H, and base(l_H) = base(l_G). Bitmappings are equal when they have the same values for their properties: lb, ub, bd, and $base$. Equal bitmappings have equivalent address functions. In Figure 3.5, neither $m_{G,1}$ nor $m_{G,2}$ is equal to m_H. This implies $A(m_{G,1}) \neq A(m_H)$ and $A(m_{G,2}) \neq A(m_H)$. However, using the set address function, $A(m_{G,1}, m_{G,2}) = A(m_H)$ holds because all address maps in m_H are covered by $m_{G,1}$ and $m_{G,2}$. This example shows that direct comparison of bitmappings can lead to a false negative, where a specification bitmapping appears to be missing in the implementation despite its address maps being present. The GBE definition avoids false negatives by using the address function A to retrieve and compare each individual address map implemented by a set of bitmappings.

3.4.3.2.2 Bitmapping Merging

To facilitate direct comparison, bitmappings m_1 and m_2 of the same root-leaf pair can be merged under the following two conditions:

1. The bit address offset between the domain and codomain must be the same for both bitmappings. This offset, also called the bitmapping address offset, is defined by Equation (3.1) and denoted addressOffset(m) for a bitmapping m. In essence, it is the offset by which addresses in the domain map to the codomain. Thus, it is calculated by

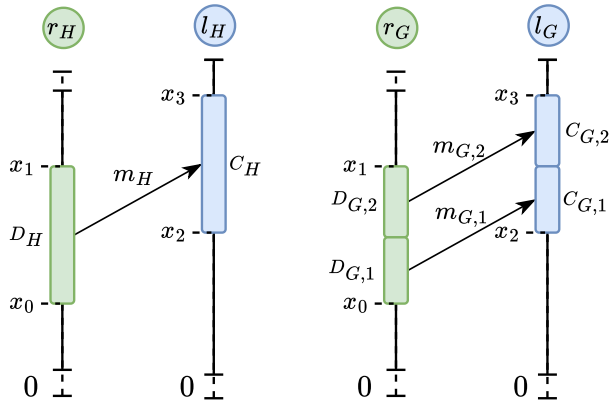


Figure 3.5: Bitmappings from two graphs that implement the same address function

subtracting the codomain's (calculated) lower bound from the domain's lower bound. Thus, addressOffset(m_1) = addressOffset(m_2) must hold.

$$\text{addressOffset}(m) = \text{lb}(m) - \text{bd}(m) - \text{base}(m) \tag{3.1}$$

2. The domains of the bitmappings must be contiguous or overlapping. Two bitmappings m_1 and m_2 are contiguous if $\text{ub}(m_1) = \text{lb}(m_2)$ or $\text{ub}(m_1) = \text{lb}(m_2)$ holds. Note that the upper bounds is exclusive and the lower bounds is inclusive. Two bitmappings m_1 and m_2 are overlapping if either $\text{lb}(m_2) < \text{ub}(m_1)$ and $\text{lb}(m_2) \geqslant \text{lb}(m_1)$ holds, or $\text{lb}(m_1) < \text{ub}(m_2)$ and $\text{lb}(m_1) \geqslant \text{lb}(m_2)$ holds.

When both conditions hold for two bitmappings m_1 and m_2 of the same root-leaf pair, they can be merged into one equivalent bitmapping that spans both and is contiguous. The merged bitmapping m_m will have the minimum lower bound $\text{lb}(m_\text{m}) = \min(\text{lb}(m_1), \text{lb}(m_2))$, the maximum upper bound $\text{ub}(m_\text{m}) = \max(\text{ub}(m_1), \text{ub}(m_2))$, and the minimum bottom delta $\text{bd}(m_\text{m}) = \min(\text{bd}(m_1), \text{bd}(m_2))$. The block base address offset is the same for both m_1 and m_2, because they have the same leaf node, such that $\text{base}(m_\text{m}) = \text{base}(m_1) = \text{base}(m_2)$ holds.

For the example in Figure 3.5, the contiguous bitmappings $m_{G,1}$ and $m_{G,2}$ from graph G could be merged into one equivalent bitmapping $m_{G,3}$, such that $A(m_{G,3}) = A(m_{G,1}, m_{G,2})$. Then, given that $\text{base}(l_H) = \text{base}(l_G)$, we get $m_{G,3} = m_H$, implying $A(m_{G,3}) = A(m_H)$. This method of establishing equivalence based on the address function is significantly simpler and more efficient than the brute force method of comparing all mapped addresses.

False positives can also occur when verifying an implementation against a specification. Imagine m_H in Figure 3.5 is as large as $m_{G,1}$, then $m_{G,2}$ would remain unmatched. However, the current specification format cannot define more than one bitmapping per leaf. Therefore, a second bitmapping in H that matches $m_{G,2}$ could never exist since it would be part of M_H. Hence, this is a false positive: a match in bitmappings is found while in reality, G implements a larger bitmapping than specified. By merging $m_{G,1}$ and $m_{G,2}$ into one, the false positive is prevented.

3.4.3.2.3 Bitmapping Maximization

The merging of all contiguous or overlapping bitmappings with the same bit address offset and from the same root-leaf pair is called *bitmapping maximization*. Figure 3.6 shows an example bitmapping maximization of a set of bitmappings to the window of a root r_G. Only the domains are shown with their range indicated above them. Assume that all have the same bitmapping address offset. The bitmappings are maximized into the one bitmapping

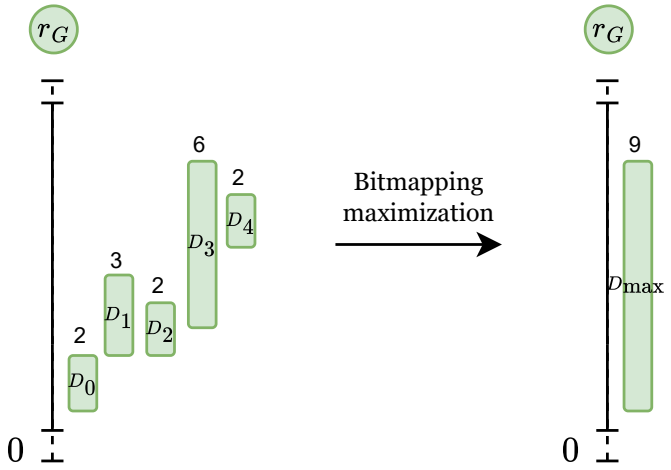


Figure 3.6: Example bitmapping maximization problem

with domain $D_{\max}$. For each of the contained bitmappings, the resulting bitmapping $m_{\max}$ has the maximum range of of all possible bitmapping merge combinations. Merged bitmappings are removed from their bitmapping set, since their exact address maps are also covered by the maximum combination. As a result, false negatives are not possible in the comparison of maximized bitmappings, for there are no bitmappings (with equal bit address offset) contiguous or overlapping with it, for they would have been merged. Furthermore, false positives are not possible in the comparison of maximized bitmappings, for there are no bitmappings (with equal bit address offset) contained in it, for they would have been merged. Therefore, there is no advantage to use address functions for comparison anymore, and we can instead perform direct comparison of maximized bitmappings.

Note that any implementation root-leaf pair with more than one maximized bitmapping must result in non-equivalence with its specification, as the specification defines only one bitmapping.

3.4.3.2.4 Algorithm

Bitmapping maximization is implemented as shown in Algorithm 4. The input is the two-dimensional set M_G calculated by Algorithm 3. For each root-leaf pair with paths in M_G, all bitmappings are added to a set M_{unmerged} (line 2-6), which is afterwards sorted in ascending order of lb (line 7). Maximization is then repeatedly performed for unmerged bitmappings. In each iteration, the `poll` function retrieves and removes the smallest lb bitmapping that is yet unmerged. The algorithm is initiated with this bitmapping as the running maximized bitmapping through (line 8-9) and an empty set of merged bitmapping of this iteration (line 10). Then, it checks each of the remaining unmerged bitmappings for the two merging conditions (line 11-12). When the conditions hold for an unmerged bitmapping, it is added to the list of merged bitmappings (line 13) and the upper bound of the maximized bitmapping is updated (line 13-16). After all unmerged bitmappings are checked, the merged bitmappings are removed from M_{unmerged} (line 19-21) and the maximized bitmapping is added to the result set M_{result} (line 22). When there are no unmerged bitmappings left, the set of maximized bitmappings replaces the original $M_{r\to l}$ (line 24).

3.4.3.3 Algorithm Analysis on Completeness and Correctness

To assess the robustness of the bitmapping maximization algorithm, we examine its completeness and correctness. This is necessary for this algorithm, because bitmapping maxi-

Algorithm 4: Graph Bitmapping Maximization

Input: M_G: Two-dimensional map of all bitmappings of AMG G, with roots in first dimensions and leaves in second dimension
Output: M_G: Two-dimensional map M_G after bitmapping maximization.

```
1  maximizeBitmappings(M_G) {
2      foreach root node r for which M(G) contains a path do
3          foreach leaf node l for which M(G) contains a path from r do
4              M_result ⟵ ∅
5              M_{r→l} ⟵ M_G [r][l]
6              M_unmerged ⟵ copy(M_{r→l})
7              sort M_unmerged in order of ascending lb.
8              while M_unmerged ≠ ∅ do
9                  m_max ← M_unmerged.poll()
10                 M_merged ⟵ ∅
11                 foreach m ∈ M_unmerged do
12                     if addressOffset(m_max) = addressOffset(m) and m_max is contiguous or overlaps with m
                       then
13                         add m to M_merged
14                         if m.ub > m_max.ub then
15                             m_max.ub ← m.ub
16                         end
17                     end
18                 end
19                 foreach m ∈ M_merged do
20                     remove m from M_unmerged
21                 end
22                 add m_max to M_result
23             end
24             replace M_{r→l} by M_result in M_G
25         end
26     end
27 }
```

mization drastically changes the manner in which GBE is determined from comparing address function outputs for each bit address to direct comparison of maximized bitmappings.

Completeness of the algorithm is demonstrated by the following four points resulting from the analysis of the algorithm. Firstly, the algorithm handles all root-leaf pairs for which paths exist. Secondly, it iterates through all bitmappings, ensuring that every bitmapping is considered. Thirdly, it ensures that all bitmappings are either merged into a maximized bitmapping or added to M_{result}, leaving no bitmappings unprocessed. Finally, completeness can be verified by ensuring that the unchecked set is fully processed, and all bitmappings are accounted for in the result set.

Correctness is demonstrated by checking whether it a collection of cases listed in Appendix A.2. These cases all verify an implementation AMG against a single specification AMG bitmapping. The cases have been constructed to cover corner cases in the merging process. Each of the cases was correctly processed.

3.4.3.4 Node Mapping Generation

With the bitmapping sets calculated and maximized for both input AMGs, the next step is to map their nodes. Root nodes are not automatically mapped; they are listed on both sides of the node map, with the right-hand side map index left empty. Even after manual mapping of root nodes, a medium to large design likely contains a significant number of unmapped implementation roots, cluttering the CSV file contents. To reduce this clutter, the graph is trimmed before analysis by removing segment roots. This process significantly reduces the number of unmapped implementation root nodes. This approach is effective because the

implementation of a specification root typically corresponds to an address space node rather than an implementation segment node.

Leaf nodes are mapped automatically with the following algorithm. Note that for a bitmapping m_1 to "fall within" a bitmapping m_2, this means that $lb(m_1) \geqslant lb(m_2)$ and $ub(m_1) \leqslant ub(m2)$ hold.

1. For both graphs G (right) and H (left), find the minimum lb bitmapping for each leaf among all its connected roots.
2. Sort both sets of minimum lb in ascending order.
3. Obtain the lowest lb bitmappings m_H and m_G for both graphs.
4. Repeat the following steps until all bitmappings have been traversed for both graphs:
   a) If m_G falls within m_H, their leaves are given the same map index and both are advanced to the next bitmapping.
   b) Else if $lb(m_G) < lb(m_H)$, the leaf of m_G is given the next available address and m_G is advanced to the next bitmapping.
   c) Else if $lb(m_H) < lb(m_G)$, the leaf of m_H is given the next available address and m_H is advanced to the next bitmapping.
   d) Else if $lb(m_G) = lb(m_H)$, the leaf of m_H is given the next available address and the leaf of m_G is left with empty map index. Both are advanced to the next bitmapping.
   e) If no nodes are left to traverse in either graph, the remaining nodes in the other graph are given no map index.

3.5 Graph Bitmapping Equivalence Assessment

The final step in the solution flow, depicted in Figure 3.1, is to determine the Graph Bitmapping Equivalence (GBE) between the two constructed Address Map Graphs (AMGs) using the generated node map. The details of the GBE assessment are recorded in a text-based report, which highlights any discrepancies found.

We implement a program to perform this assessment that follows the GBE definition provided in Section 2.6. Like the node map generator described in previous Section 3.4, it uses bitmapping maximization to implement address functions and facilitate direct bitmapping comparison. Additionally, the program merges nodes as defined in the node map and generates the text-based report highlighting any inequivalences.

3.5.1 Program Arguments

The program accepts the following arguments:

1. *Specification AMG GraphML file path*: The path to the GraphML file describing the specification AMG.
2. *Implementation AMG GraphML file path*: The path to the GraphML file describing the implementation AMG.
3. *Node map CSV file path*: The path to the CSV file describing the node map generated for the same two AMGs. It defines their node mappings and specifies any leaf nodes to be merged in the implementation AMG.
4. *Output GBE report TXT file path*: The path at which the text-based GBE report should be stored.

3.5.2 GBE Assessment Procedure

The following subsections explain the steps executed by the program to determine GBE between the provided AMGs. Generally, the program calculates the sets of all bitmappings for both AMGs, performs bitmapping maximization on these sets, merges leaf nodes according to the node map, and then determines GBE using the node map. The initial steps of bitmapping calculation and bitmapping maximization are also performed by the node map generator and have already been explained in the previous section, and hence will not be explained in detail. This section will focus on the steps that follow: leaf node merging, GBE calculation, and report generation.

3.5.2.1 Leaf Node Merging

When the node map specifies that multiple leaf nodes in graph G share the same map index as one node in graph H, these leaf nodes must be merged. Leaf nodes can be merged in two ways: either keeping internal merge gaps or removing them.

For nodes to merge correctly, all bounds of their maximized bitmappings must align with each other, to form one contiguous block. If they do not align, gaps of unmapped addresses exist between the bitmappings. From the implementor's perspective, the gaps may already be known and the nodes must be merged anyway, such that whatever gaps are present are not important and thus can be ignored. However, from the specifier's perspective, the gaps have not been specified, hence this discrepancy must be present in the equivalence report. Hence, the gaps may be handled in different ways.

The node map CSV file is parsed into a mapping function f_{node}, which maps nodes in implementation G to nodes in specification H. Nodes $u, v \in V(G)$ that are to be merged, must map to the same node in $V(H)$, such that $f_{\text{node}}(u) = f_{\text{node}}(v)$ holds. The merging process is performed on the two-dimensional map M_G maximized by Algorithm 4.

Algorithm 5: Node Merging, Keeping Gaps

Input: f_{node}: Map from nodes in G to nodes in H
Input: M_G: Two-dimensional map of all bitmappings of AMG G, with roots in first dimension and leaves in second dimension
Output: M_G: The two-dimensional map after bitmapping maximization with internal merge gaps kept

```
1  mergeLeaves(f_node, M_G) {
2      collect distinct H nodes that f_node maps to, in set V_{H,mapped}
3      foreach node v_H in V_{H,mapped} do
4          collect all distinct G nodes that f_node maps to v_H in set V_{G,same}
5          if |V_{G,same}| > 1 then
6              foreach root node r for which M(G) contains a path do
7                  M_result ← ∅
8                  foreach node l ∈V_{G,same} do
9                      if M(G) contains a path from r to l then
10                         add all bitmappings from M_G[r][l] to M_result in order ascending lb
11                     end
12                 end
13                 maximize the bitmappings in M_result
14                 foreach leaf node l ∈V_{G,same} do
15                     replace M(G, r, l) in M_G with M_result
16                 end
17             end
18         end
19     end
20 }
```

The implementation of node merging does not involve removing nodes from the AMG nor inserting a single merged node. Instead, when multiple leaves need to be merged, we

keep all of them in the AMG and refer to them as *merge siblings*. After the merging process, each merge sibling contains the set of maximized bitmappings between each root and the set of merge siblings. Thus, we perform merging at the bitmapping level, not the AMG level.

This approach avoids altering the AMG by removing or inserting nodes and edges, as merging would remove detail from the AMG. Instead, we focus on interpreting the AMG data. Additionally, inserting and removing nodes and edges would affect more bitmappings than those that need merging. Consequently, we would need to update all affected paths to reflect the new merged node's base and range. By performing the merge at the bitmapping level, we avoid these drastic effects and maintain simplicity in the analysis.

After completing bitmapping maximization and node merging, the surjective node map, which maps multiple implementation nodes to one specification node, has become a bijective node map. This bijective node map matches merged nodes to specification nodes, as required by the GBE definition.

3.5.2.1.1 Keeping Internal Merge Gaps

Algorithm 5 shows how nodes are merged while maintaining the gaps between bitmappings. First, we find the set of merge sibling $V_{G,\text{same}}$, which are the leaves in G that map to the same leaf in H according to f_{node} (line 2-5). Then, for each root r from which there is a path in $M(G)$, the bitmappings of each merge sibling $l \in V_{G,\text{same}}$ with a path to said root are added to a combined bitmapping set (line 6-12). Because bitmapping maximization was only performed on a root-leaf pair basis, the resulting merged bitmapping set M_{result} may be non-maximized. Hence, the maximization is again performed on the set, using the same algorithm as in Algorithm 4 (line 13). Then, each $M(G, r, l)$ is set to this maximized combined set M_{result}, such that each merge sibling has the same combined set of bitmappings for the same root (line 14-16).

Algorithm 6: Node Merging, Removing Gaps

Input: f_{node}: Map from nodes in G to nodes in H
Input: M_G: Two-dimensional map of all bitmappings of AMG G, with roots in first dimension and leaves in second dimension
Output: M_G: The two-dimensional map after bitmapping maximization with internal merge gaps removed

```
1  mergeLeaves(f_node, M_G) {
2      collect all distinct H nodes that f_node maps to, in set V_{H,mapped}
3      foreach node v_H in V_{H,mapped} do
4          collect all distinct G nodes that f_node maps to v_H in set V_{G,same}
5          if |V_{G,same}| > 1 then
6              foreach root node r for which M(G) contains a path do
7                  M_result ⟵ ∅
8                  foreach leaf node l ∈ V_{G,same} do
9                      add M(G, r, l) to M_result
10                 end
11                 check address alignment is equal across M_result
12                 find the lowest lb bitmapping m_first ∈ M_result
13                 find the highest ub bitmapping m_last ∈ M_result
14                 create a new bitmapping m_merged with the lb, bd, base of m_first and ub of m_last
15                 foreach node l ∈ V_{G,same} do
16                     replace M(G, r, l) in M_G with {m_merged}
17                 end
18             end
19         end
20     end
21 }
```

3.5.2.1.2 Removing Internal Merge Gaps

This algorithm, shown in Algorithm 6, fills any gaps between bitmappings to form one large bitmapping. Similar to the previous algorithm, we identify the set of merge sibling $V_{G,\text{same}}$, which are the leaves in G that map to the same leaf in H according to f_{node} (line 2-4). For each root r with a path in $M(G)$, we collects all bitmappings to the merge siblings (line 6-10). Because no maximization step is performed, we must verify manually whether the address alignment is equal across M_{result} (line 11). If this is not the case, the program outputs a warning message. Afterwards, we find the two bitmappings in M_{result} with lowest lb and highest ub, respectively (line 12-13). A new bitmapping m_{merged} is created from these two bitmappings according to line 14, such that the range of the new bitmapping spans all bitmappings to be merged. Finally, each $M(G, r, l)$ is replaced by m_{merged} (line 15-17) such that it is the only bitmapping from root r to each merge sibling $l \in V_{G,\text{same}}$.

3.5.2.2 GBE Assessment

We perform GBE calculation in two steps. First, we determine formal GBE by a simple algorithm that sticks to the GBE definition, but which thereby cannot give further details into inequivalences or bitmappings from unmapped nodes. If it is determined that GBE does not hold, then we perform a more detailed analysis through a more complicated algorithm which is able to generate a detailed report on inequivalences.

3.5.2.2.1 GBE Calculation

First, Algorithm 7 calculates GBE between supplied AMGs. According to the definition of GBE in Section 2.6, only the nodes mapped by the node map are handled. It comes down to checking for equivalence $M(G, r, l) = M(H, r', l')$ for all $r \in R(G)$ mapped by f_{node} such that $r' = f_{\text{node}}(r)$, and all leaves $l \in L(G)$ mapped by f_{node} such that $l' = f_{\text{node}}(l)$. As explained in Section 3.4.3, this is possible due to the merges performed by bitmapping maximization, which allows for direct bitmapping comparison while avoiding false negatives and false positives. The returned boolean declares whether AMG G and H are bitmapping equivalent according to its formal definition.

Algorithm 7: Graph Bitmapping Equivalence Calculation

```
   Input: M_H: Two-dimensional map of all bitmappings of AMG H
   Input: M_G: Two-dimensional map of all bitmappings of AMG G
   Input: f_node: Map from V(G) to V(H)
   Output: Boolean whether GBE holds
 1 calculateSimpleEquivalence(M_H, M_G, f_node) {
 2     foreach root r_G mapped by f_node do
 3         r_H ← f_node (r_G)
 4         foreach leaf l_G mapped by f_node do
 5             l_H ← f_node (l_G)
 6             M_{G,r→l} = M_G[r_G][l_G]
 7             M_{H,r→l} = M_H[r_H][l_H]
 8             if M_{G,r→l} ≠ M_{H,r→l} then return false
 9         end
10     end
11     return true
12 }
```

3.5.2.2.2 Report Generation

In case the GBE calculation results in bitmapping inequivalence, we should find the discrepancies in either the specification or implementation that caused it. For this purpose, we perform an extended algorithm to provide insight on the exact equivalences and inequivalences

between the two AMGs, including for unmapped implementation leaves, in order to get a complete report on the comparison of the two AMGs.

To perform a more detailed GBE analysis and generate the report, Algorithm 8 takes a different approach and direct compares individual bitmappings in G and H. Its inputs are the two-dimensional maps M_H and M_G and node mapping function f_{node}.

Algorithm 8: Graph Bitmapping Equivalence Report Generation

```
   Input: M_H: Set of all bitmappings in H in order of ascending lb
   Input: M_G: Set of all bitmappings in G in order of ascending lb
   Input: G: Implementation AMG
   Input: f_node: Map from V(G) to V(H)
   Output: Text-based GBE report
 1 calculateEquivalence(M_H, M_G, G, f_node) {
 2     M_unmatched ← ∅
 3     foreach root node r_G mapped by f_node do
 4         r_H ← f_node[r_G]
 5         get next bitmapping m_G in M(G) with lowest lb and r_G as root
 6         get next bitmapping m_H in M(H) with lowest lb and r_H as root
 7         while not all bitmappings in M(G) and M(H) have been traversed do
 8             if m_G and m_H exist then
 9                 get leaf l_G of path p_G of m_G
10                 get leaf l_H of path p_H of m_H
11                 l_H,mapped ← f_node [l_G]
12                 if a sibling merged leaf of l_G has been traversed for r_G then continue
13                 if l_H,mapped = l_H and m_G equivalent to m_H then
14                     report both bitmappings as equivalent
15                     get the next lowest lb bitmappings for both m_G and m_H
16                 else
17                     if m_H.getUb() smaller or equal to m_G.getLb() then
18                         report m_H as non-equivalent
19                         get next bitmapping m_H in M(H) with lowest lb and r_H as root
20                     else
21                         report m_G as non-equivalent
22                         get next bitmapping m_G in M(G) with lowest lb and r_G as root
23                     end
24                 end
25             else if m_H is null then
26                 report m_H as non-equivalent
27                 get next bitmapping m_H in M(H) with lowest lb and r_H as root
28             else
29                 report m_G as non-equivalent
30                 get next bitmapping m_G in M(G) with lowest lb and r_G as root
31             end
32         end
33     end
34     return M_unmatched
35 }
```

For each root r_G in G that is mapped by f_{node}, the mapped root $r_H = f_{\text{node}}(r_G)$ is obtained as well as the first bitmappings m_G and m_H for any leaf in either graph (line 2-5). While there are bitmappings in either set, the current bitmappings m_G and m_H are compared and the next bitmappings are retrieved. When the leaf node of m_G is a merged node, and one of its merge siblings has already been traversed, then this bitmapping must be skipped in order to avoid duplicate processing (line 11). If the mapped leaf $l_{H,\text{mapped}}$ of m_G is equal to the leaf l_H of m_H, then their bitmapping may be compared (line 12). Then the bitmappings are compared and reported as equivalent or non-equivalent. Whether two bitmappings are reported as equivalent (line 13-15) depends on the strictness of the program, which can be configured to include the occasion where bitmapping $m_G \in G$ implement a smaller range

than its counterpart $m_H \in H$, while m_H totally overlaps m_G. This situation is called a *partial equivalence*. In such case, the report must indicate the partial equivalence, which will be visible when we discuss the structure of the report in the following chapter. If the leaf nodes are unmapped, or if the bitmappings are found non-equivalent, one of the lower lb bitmapping is printed (line 16-24). If either of the bitmappings is null, the other is deemed unmatched and printed (line 25-33).

3.6 Solution TCL Scripting Flow

Each individual program is wrapped into a TCL script. The solution flow is initiated through a single script, `run.tcl`, by which each step in the flow can be enabled or disabled, providing flexibility in the execution process. The inputs and parameters for all scripts are configured in a separate TCL file, `preamble.tcl`, which is run at the start of each individual program, such that its TCL script can be initiated on its own as well.

For the specification graph generator, its TCL script supports input files in XLSX, XLS, and CSV formats to define the specification spreadsheet. To ensure compatibility with the JXL library used for parsing, all input files are converted to CSV and then back to XLS. This conversion process is handled using the command-line interface of LibreOffice, specifically using its conversion functionality [20], [21].

Chapter 4

Evaluation

This chapter discusses the evaluation of the solution against a state-of-the-art mid-size SoC design for which specification and implementation files are provided as a case study. The chapter is structured as follows. First, we apply the solution flow to the design's memory specification and implementation, and discuss the result of each step from AMG generation to GBE calculation. Then, we discuss the generated GBE report, including its structure. Finally, we discuss the findings of the report.

4.1 Report Structure

To understand the contents of the GBE report, it is important to first explain its structure shown in Figure 4.1, which consists of a header, a detailed report table, and path traces. An example of a report header, along with the first few bitmappings of the report table. To provide context and traceability, the report also includes the analysis date and the source paths for the three input files: the GraphML files of the two compared AMGs and the node map CSV file. These are not shown by the figure.

The report table lists all bitmappings for both graphs in a format designed to facilitate manual comparison. The table contains two sections, with each section displaying the bitmappings of one graph, referred to as 'Graph H' and 'Graph G'. Each row in the table represents a bitmapping.

Bitmappings within the table facilitate the comparison between both graphs. The middle columns display the bitmapping domains as boxes, with their lower and upper bounds (lb and ub) shown at the top and bottom of each box, respectively. These bounds are formatted in byte-addressed hexadecimal format to maintain consistency with the specification and IP-XACT format. The domain boxes are positioned centrally in the row, making differences in their bounds more recognizable.

The domains point outward to their corresponding bitmapping codomains, which are also represented as boxes with lower and upper bounds. The codomain lower bound is calculated as $\text{base}(m) + \text{bd}(m)$, while the upper bound is calculated as the codomain lower bound plus the domain range. Each codomain box includes the unique identifier of its GraphML node, allowing for traceability back to the node map and differentiation from identically named leaves.

Each row also contains several identifiers to provide context and potential for further analysis to the user. The leaf name is displayed beside each codomain box to help users recognize the associated specification entry or implementation IP. Domain boxes also contain a report identifier, which can be used to retrieve the path trace, as shown in Figure 4.2. The path trace facilitates further interpretation of the report, especially in identifying the cause of any non-equivalence by showing the base and range of each node along the bitmapping

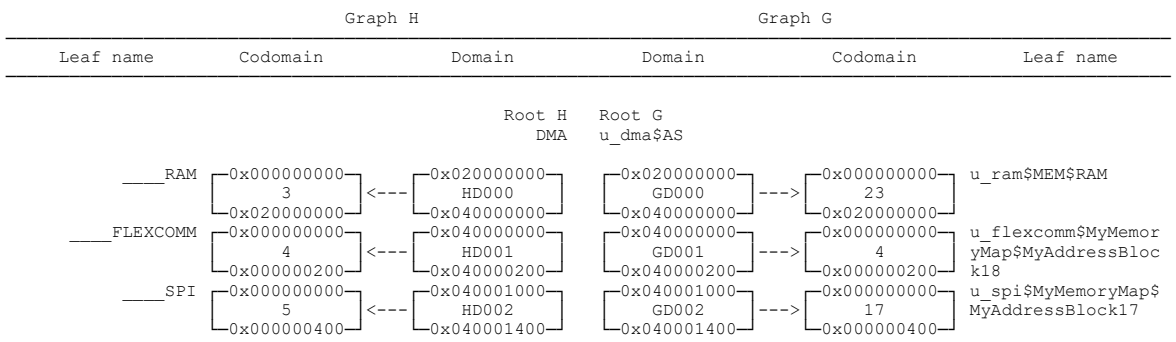


Figure 4.1: Header and first entries of a GBE report example

```
GD007 --> GC007
u_cpu$AS
        base: 0x0
        range: 0x100000000
        offset: 0x0
    --> u_cpu$AS$Peripheral
        base: 0x40000000
        range: 0x20000000
        offset: 0x40020000
    --> u_busahb$AS_3
        base: 0x0
        range: 0x1000
        offset: 0x0
    --> u_dma$AHB$RegisterBlock
        base: 0x0
        range: 0x44
```

Figure 4.2: Example of a path trace

path, as well as the offset induced by the edges. An offset displayed in a node indicates the shift by which the memory of the following node is mapped.

Finally, the table displays all bitmappings in order of ascending lb. In the simplest case, a bitmapping in the table represents a single bitmapping calculated from the AMG. Alternatively, it may represent the maximized bitmapping within a single root-leaf pair. In the more complex scenario, a maximized bitmapping may merge bitmappings across multiple root-leaf pairs if their leaves have been merged via the node map. In such case, the name of the first node leaf (the one with the bitmapping with the minimum lb) is used, prefixed with "MERGED_".

A user may want to view all bitmappings to leaf nodes in an unmerged state. In such case, the report generation can be configured without leaf node mappings, so that none leaves are merged. Additionally, bitmapping maximization can be disabled completely to view all bitmappings as they exist in the graphs, providing a complete and unmerged representation. However, without node merges or bitmapping maximization, correct GBE calculation cannot be achieved when these processes are necessary.

4.2 Bitmapping Comparison Scenarios and Report Results

A comparison of bitmappings from specification AMG H and implementation AMG G can result in the following three scenarios:

1. *Total Equivalence*: A pair of bitmappings in H and G have equal domain and codomain bounds. The bitmappings are printed side by side, as shown in Figure 4.3.

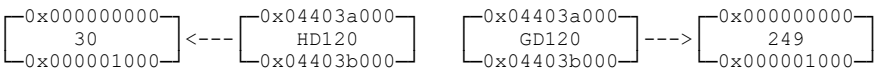


Figure 4.3: Two bitmappings with total equivalence

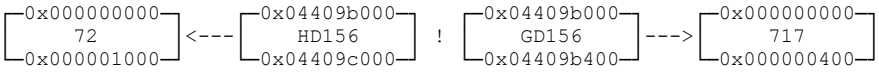


Figure 4.4: Two bitmappings with partial equivalence

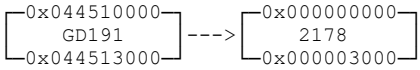


Figure 4.5: An implementation bitmapping without equivalent specification bitmapping

2. *Partial Equivalence*: The domain of the bitmapping in G falls within the bounds of the bitmapping in H, with equal address alignment to their codomains. This indicates only part of the specified bitmapping is implemented. The bitmappings are printed side-by-side with an exclamation mark between them, as shown in Figure 4.4.

3. *Non-Equivalence*: No totally or partially equivalent bitmapping is found in the other graph. The bitmapping is printed on one side only. Figure 4.5 shows an example where no equivalent bitmapping is found in H. This creates "holes" in the table, such that non-equivalence are easily identified by the user. Holes may occur on both sides.

We have chosen to report the results in this format in order to facilitate the identification of non-equivalences, for these are most likely the points of interest to the user.

4.3 Analysis and Interpretation of Non-Equivalences

Manual analysis was conducted to determine the causes of the 126 non-equivalences identified during the evaluation. This section discusses the methodology of this non-equivalence analysis, the causes identified, and the implications of these findings for interpreting the results.

4.3.1 Analysis Methodology

The analysis of non-equivalences generally adhered to the following approach to identify the cause. In this approach, we start with unmatched specification bitmappings. The steps are as follows:

1. *Locate non-equivalent bitmappings*: Identify a hole in the right side of the GBE report table, and identify the specification bitmapping m_1.

2. *Identify potential matches*: Locate the anticipated equivalent bitmapping m_2 in the right side of the GPE report table among neighboring bitmappings. The anticipated equivalent bitmapping often occurs directly before or directly after m_1 in the report, as all bitmappings are in ascending order of lb. Whether a bitmapping is anticipated to be equivalent can be derived from the similar leaf name and domain bounds that are close to the domain of m_1. The m_2 cannot be found, specification m_1 is determined not to be implemented at all.

3. *Compare domain bounds*: Compare the domain bounds of m_1 and m_2. If they do not correspond, it suggests that m_2 extends beyond the bounds of its specification m_1.

4. *Compare address alignments*: Compare the codomain address alignment of both m_1 and m_2. If these differ, it suggests that m_2 mapped its domain at an unspecified offset.

5. *Verify node map*: Ensure that the node map correctly maps the leaf nodes of m_2 to that of m_1. If unmapped, map them and regenerate the GBE report.

6. *Check for unmerged bitmappings*: If other unmatched implementation bitmappings are present before or after m_2, and together with m_2 form an equivalent bitmapping to m_1 when merged, then verify that the node map specifies the merging of their leaf nodes. If not, correct the node map and regenerate the GBE report.

When the non-equivalence persists after step 2 or step 5, it indicates that the implementation of the bitmappings does not adhere to its specification. In this case, the specification and implementation must be examined for errors. This often requires extensive knowledge on the workings of the SoC and is in this research performed with the help of the SoC architect or a design verification engineer. To aid this manual examination, two commercial tools were used for memory analysis of IP-XACT designs.

The remaining unmatched implementation bitmappings in this evaluation fall in two categories. Firstly, when their bounds are close to that over other implementation bitmappings, and their leaf names are equal, then this means that this bitmapping is part of a merged bitmapping. However, because there is a gap between their domains, they cannot be maximized to one bitmapping. The user can disable the reporting of internal merge gaps, but for this evaluation, it was enabled to maintain a strict evaluation. Secondly, the bitmapping implements and unspecified bitmapping, which must be examined with the SoC architect.

4.4 Solution Application

Starting with AMG generation, we parse the design's specification XLS file into an AMG comprising 134 edges and 120 nodes, including 5 roots and 115 leaves. AMG generation from the implementation IP-XACT files results in a significantly larger AMG with 6151 edges and 2018 nodes, consisting of 658 roots, 650 leaves, and numerous intermediate nodes. The generation process, performed in a virtual machine, takes 18.2 seconds, utilizing 2 cores running at 2.4 GHz and 16 GB of RAM.

Visualization of the implementation AMG produces a very large graph. Despite its size, each peripheral is easily identifiable by its small tree of nodes, hanging from a large intertwined collection of edges corresponding with the Network-on-Chip (NoC) component present in the design. This NoC seemingly always interjects the paths, which makes it impossible to find the exact roots of paths above the NoC. Ultimately, the visualization aids in debugging and further interpretation of the GBE report.

Node map generation results in 316 implementation leaves being automatically mapped to 73 specification leaves. Afterwards, we map 31 implementation leaves manually to 18 specification leaves. We verified the implementation completely for one root of the specification that addresses the majority of the peripherals. Other roots were not handled due to their similarity in mappings.

The final program first calculates all bitmappings of both the specification and implementation. For the specification, 107 bitmappings are calculated between the 115 leaves and the one root. For the implementation, 502 bitmappings are calculated between the 650 leaves and the one root, taking around 800 milliseconds. Next, node map interpretation cause 104 implementation bitmappings to be merged into other bitmappings, resulting in 212 implementation bitmappings remaining. Furthermore, 144 implementation leaf nodes are merged

into 24 leaf nodes, leaving 530 implementation leaf nodes and 184 bitmappings. This process takes around 10 milliseconds. The final result is that the specification and the implementation are not equivalent. The generated report shows the equivalent and non-equivalent bitmappings for further inspection.

4.4.1 Causes of Non-Equivalence

The analysis of non-equivalences resulted in two categories of non-equivalences: one-sided and two-sided. one-sided non-equivalences are bitmappings for which the anticipated other bitmapping could not be found in the GBE report table. When the anticipated bitmapping is found, the non-equivalence is two-sided.

For one-sided non-equivalences, the following causes were identified in this evaluation:

1. *Outdated specification XLS*: 4 specification and 2 implementation bitmapping are unmatched due to the specification XLS being out-of-date, and thereby not reflecting design changes made during implementation.

2. *Dead-end paths*: 1 specification bitmappings is unmatched due to the implementation path ending in an address block without connection to any leaf node, resulting in a dead-end path ending in an `addressSpace` or `segment` instead of `addressBlock`. In our case, this is caused by an incomplete IP-XACT description of the implementation, where implementation leaf interfaces have no reference to a memory element, neither direct nor through hierarchical interconnects.

3. *Internal merge gaps*: 4 implementation bitmappings are unmatched due to merged leaf nodes having gaps between their bitmappings. This causes the bitmapping maximization algorithm to merge into multiple bitmappings with gaps between them. The lowest *lb* bitmapping a partial matches with the specification, while the other bitmappings remain unmatched.

For two-sided non-equivalence, the following causes were identified in this evaluation:

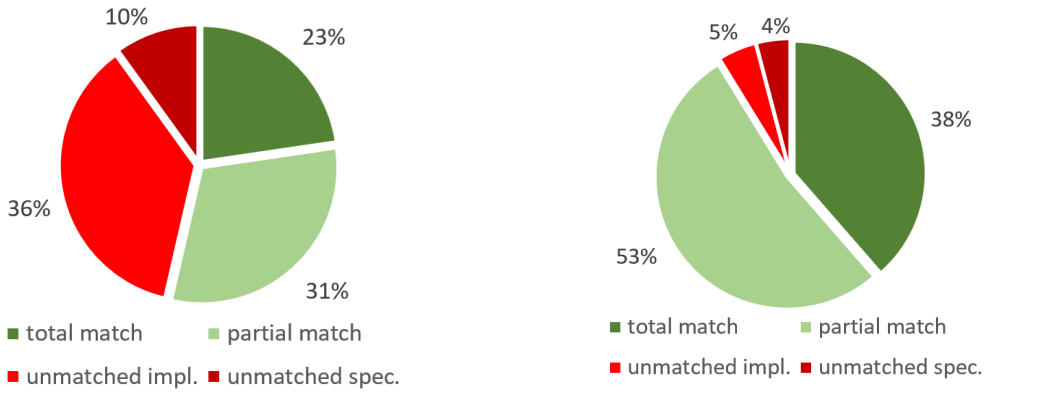
1. *Different bitmapping address offsets*: 2 specification and 2 implementation bitmappings are unmatched due to the bitmapping address offset of their codomains being different. As a result, they map their addresses with different offsets to their codomains. In this design, it was caused by the specification assuming relative addressing while the implementation employs absolute addressing to the codomain. As a result, the specification domain maps to the codomain with starting address 0, while the equivalent implementation domain maps to the codomain with starting address non-zero.

2. *RTL-based address handling*: IP-XACT implementation may not describe the complete handling of addresses in the RTL code. Therefore, their addressing in the implementation may align with the specification, but this cannot be derived from the IP-XACT. Meanwhile, RTL-based address handling may affect all aspects of a bitmapping, thereby creating a non-equivalence. Two examples of such non-equivalences are the following:

   a *Parallel ROM/RAM blocks*: 20 specification and 96 implementation bitmappings are unmatched due to the way in which ROM and RAM memory is described in the IP-XACT description of this design. They are constructed by multiple `addressBlocks` mapped in parallel to the same domain. This method of handling `addressBlocks` is not described in the IP-XACT standard, and is instead handled via RTL code.

Evaluation Property	Implementation AMG G	Specification AMG H
#Nodes	2018	120
- #Roots Nodes	658	5
- #Leaf Nodes	650	115
#Edges	6151	134
#Automatically Mapped Leaves	316	73
#Manually Mapped Leaves	31	18

Table 4.6: Evaluation properties of implementation and specification graphs



(a) Raw results

(b) Unsupported non-equivalences excluded

Figure 4.7: The distribution of evaluation bitmapping equivalence scenarios

b *Different memory organization between components*: 2 specification and 2 implementation bitmappings are unmatched due to the implementation bitmappings having significantly larger ranges their specification bitmappings. This occurs due to different memory organization used across components in the path. For example, the initiator IP may perform byte-aligned accesses on its 32-bit word memory, while this may access memory of a peripheral which performs word-aligned accesses on its 12-bit word memory. Hence, in each word access in the peripheral, 4 bits remain unread. Our solution converts all into a bit-addressed organization, such that it cannot take into account which bits will not be used due to word alignment. This involves mostly DSP peripherals whose RTL perform address modification for each memory access.

4.5 Results

Our solution processed the XLS address map specification and IP-XACT design description of a mid-size SoC into a specification AMG H and implementation AMG G, respectively, whose properties are summarized in Table 4.6. Application of the rest of our solution flow to H and G results in 156 equivalent bitmappings, of which 66 total and 90 partial. Furthermore, it also results in a significant 135 non-equivalent bitmappings, of which 106 in the implementation AMG and 29 in the specification AMG. The distribution of the raw results of this evaluation are shown in Figure 4.7a. Together, all scenarios represent 106 bitmappings in the specification, 184 bitmappings in the implementation, totaling to 291 bitmappings in both.

At first glance, it shows nearly half of the found bitmappings remain unmatched, which would make you assume that the solution might not work well, as the design surely cannot have this many incorrectly specified or implemented address maps. However, further analysis found several considerations to be made when interpreting these raw results. Afore-

Category	#Bitmappings Raw	#Bitmappings After Exclusion
Total Equivalence	66	66
Partial Equivalence	90	90
Non-Equivalence	135	15
- From Implementation	106	8
- From Specification	29	7
Total	291	171
- From Implementation	184	86
- From Specification	107	85

Table 4.8: Evaluation report results

mentioned causes of non-equivalences mention the case of non-equivalences caused by RTL-based address handling. The effect of RTL-based address handling on the bitmapping cannot be derived from IP-XACT analysis, thus not by our solution. Therefore, we decide to leave out these comparison from this evaluation.

Exclusion of unsupported non-equivalences has significant impact on the results, leaving 8 unmatched implementation bitmappings down from 106, and 7 unmatched specification bitmappings down from 29. The distribution of evaluation results are illustrated in a pie chart after excluding these errors in Figure 4.7b. The majority of the decrease in non-equivalences originates from the amount of excluded bitmappings to ROM and RAM peripherals. All results before and after exclusion of unsupported errors are summarized in Table 4.8.

Chapter 5

Conclusion

5.1 Summary

This thesis presents a methodology for formal verification of memory organizations of SoC designs described in IP-XACT. This is achieved by (i) modeling of the address map structures implemented by a design's IP-XACT description (memory implementation) and specified by its global address map spreadsheet specification (memory specification) into a unified graph model, called an Address Map Graph (AMG); and by (ii) analysis of AMGs on equivalence of their mapped addresses, called Graph Bitmapping Equivalence (GBE).

We have implemented the graph model and algorithms of this methodology into a modular chain of programs, integrated in the IP-XACT workflow. The programs process XLS memory specifications and IP-XACT design descriptions into the graph model and perform efficient calculation for GBE. Finally, we generate a report that highlights all correspondences and discrepancies between addresses mapped by specification and implementation.

The resulting solution flow has been evaluated against a state-of-the-art, mid-size, real-world SoC design. Our evaluation demonstrates the capability of the developed solution to analyze and verify the memory organization of complex SoC designs, and to assist in identifying the causes of discrepancies. As such, the algorithms developed for processing memory specifications and implementations into the graph model, along with the efficient GBE calculation methods, offer a solution for memory verification in SoC designs. The modularity of the solution facilitates adaption to various specification standards. Overall, this work represents another step in the advancement of formal SoC memory verification.

5.2 Discussion

This thesis addresses four research questions:

- **RQ0**: What unified data model represents both a memory implementation and specification, and enables the comparison of their address maps?
- **RQ1**: What algorithms process a memory specification XLS file into the unified data model?
- **RQ2**: What algorithms process the IP-XACT files of an SoC into the unified data model?
- **RQ3**: What algorithms check any two unified data model instantiations for equivalence in address maps?

Each question has been addressed through the development and implementation of specific algorithms, followed by their application to a real-world scenario.

5.2.1 Definition of the AMG model and GBE

To address RQ0, we have developed a unified graph model called the Address Map Graph (AMG). This model encapsulates the essential aspects of both memory implementations and specifications through its node, edge, and graph properties. In this definition, we focus on the typically used IP-XACT objects and constructs to realize address maps. Additionally, we have introduced the concept of a bitmapping, which describes the mapping of contiguous memory regions from root to leaf nodes along a path in the AMG. Finally, we have defined Graph Bitmapping Equivalence (GBE) to enable comparison and establish the equivalence of mapped addresses between two AMGs. The evaluation of the implementation solution with a mid-size SoC have shown the AMG accurately represents the implementation and specification.

5.2.2 Processing of Memory Specification XLS Files into AMGs

To address RQ1, we have developed an algorithm that traverses the rows and columns of the XLS spreadsheet and converts the contained information into nodes and edges, creating an Address Map Graph (AMG) of the memory specification. The algorithm can accommodate different spreadsheet layouts. After implementing this approach in a parser program, we have successfully applied it to the memory specification of the evaluated SoC design. Finally, the evaluation has shown that the parser is able to accurately form the two-level AMG.

5.2.3 Processing IP-XACT Designs into AMGs

To address RQ2, we have created an algorithm to recursively traverse the IP-XACT design, processing its structures and metadata into nodes and edges to create an AMG of the memory implementation. The algorithm integrates extra IP-XACT identifiers into the graph model by adding properties to each node, ensuring unique identification of each IP-XACT element. The process involves recursively processing each component, adding nodes for memory elements and edges for their various interconnection structures. Finally, we manually verify the algorithm's implementation with various small IP-XACT design cases containing corner cases. It has also been verified against the evaluated SoC IP-XACT design, for which inconsistencies have been verified with the results of a commercial memory analysis tool. All verifications again have shown that the resulting AMGs are accurate in their representation and the capability of the algorithm to process the memory organization of complex IP-XACT descriptions into an AMG.

5.2.4 Checking AMGs for GBE

For RQ3, we have developed a method to calculation Graph Bitmapping Equivalence (GBE) between a specification and implementation AMG efficiently and accurately. The found algorithm involves three steps. First, we have developed an algorithm to merge contiguous bitmappings into their maximum combination, called bitmapping maximization. Second, we have created an algorithm to construct a node map from two AMGs based on the correspondence of their bitmappings. Finally, we have developed two algorithms to determine GBE: one checks for equivalence between bitmapping sets, while the other traverses all maximized bitmappings individually and generates a report that provides a clear overview of any inequivalences. Our evaluation with the mid-size SoC showed the effectiveness of the report. 91% of the bitmappings were found equivalent, with the remaining non-equivalent bitmappings clearly reported. We have shown the report to the design architect, who was able to intuitively read the report, which enabled him to perform a quick analysis and identification of inconsistencies in the SoC.

5.2.5 Key Observations

Application of the solution to a mid-size SoC design and its specification has resulted in the distribution of bitmapping equivalences and inequivalences shown in Figure 4.7b. We can interpret partial matches as a negative or positive result for the equivalence of AMGs, but in this case study we have handled partial equivalences as a positive result. The reason for this is that the architect knows that the implementation of this design may implement address mappings that are smaller than specified. To elaborate for evaluated designs, given an IP-XACT `addressBlock` AB, then range(AB) typically is equal to the difference between the lowest and highest addressable register. On the other hand, in a specification, the SoC architect generally reserves a range in terms of whole kilobytes (1 Kbyte or 4Kbyte). As such, implementation bitmappings with unmapped addresses at the start or end of their domains. but that are totally overlapped by the specification bitmapping are deemed acceptable. If we do not expect partial matches for a design, the program can be configured to report them as non-equivalences.

As expected, the results show that the majority of the evaluated SoC design has been implemented according to its specification. We expected the significant amount of partial matches as they were deemed correct according to the designers of the IP-XACT description. Additionally, we expected the identified internal merge gaps and the resulting unmatched bitmappings. A commercial memory analysis tool also displayed the mapped `addressBlocks` with gaps of unmapped addresses, while the specification spreadsheet describes them as a single contiguous block of mapped peripheral memory. Thus, the `addressBlocks` are not contiguous with each other, though the specification describes otherwise. The removal of internal merge gaps is deemed incorrect for the evaluated SoC, because it would have assumed all addresses within the merged bitmapping to be addressable, while some of them are not mapped, thus not addressable.

Finally, this evaluation has resulted in several important findings for the SoC design. Firstly, we have found implemented bitmappings that are not specified or specified bitmappings that are not implemented. These findings directly show a discrepancy between specification and implementation for designers to review, either by updating the specification or extending the implementation.

Secondly, we have found different codomain address offsets between specification and implementation bitmappings caused by use of relative versus absolute addressing. We can resolve the first case of address misalignment by enhancing the specification format. This enhancement involves adding an extra column where the architect can define whether each peripheral is mapped using relative or absolute addressing.

The comparison against results from commercial tools, examination of the resulting equivalence distribution, detailed analysis of inequivalences, and their resulting key observations demonstrate that the implemented algorithms adequately address the research questions. The findings from equivalence checks, the subsequent reports, and their interpretation by the design architect highlight the effectiveness of the proposed methodologies in verifying the memory organization of SoC designs.

5.3 Future Work

5.3.1 GBE Analysis for AMGs of Equal Type

This thesis handled the verification of an implementation against a specification AMG. However, for the node mapping and GBE calculation procedures it is also possible to supply two AMGs of equal type: either two specification AMGs or two implementation AMGs. Currently, our node mapping process supports merging nodes only from the second AMG. We have not implemented bidirectional leaf merging because the primary focus has been on GBE

calculation between a specification and an implementation. Nonetheless, we can still apply the solution to AMGs of equal type under the limitation of unidirectional merging capability.

Extending this work to support bidirectional verification for equal AMG types would enhance the solution's versatility. To achieve this, the node mapping and GBE calculation procedures require extension to correctly process AMGs of the same type. This mainly involves handling of bidirectional leaf merging for the node mapping procedure, and handling of bidirectional merges in the GBE report generation. As a result, two implementations or specifications could be verified against each other for equivalence in address mappings.

5.3.2 Dynamic IP-XACT Address Maps

This thesis handled static implementation graphs as described in IP-XACT 1685-2009 [15]. The newest standard of IP-XACT 1685-2022 [3], however, enables element configurations to alter during runtime. This is achieved through elements called *modes of operation*. Consequently, address decoding inside a SoC may depend on its mode of operation. Examples of such modes of operations are boot mode, test mode, user mode, and supervisor mode. In different modes, access to certain memory regions may be restricted or memory regions may be remapped to other addresses. Only one mode can be active per element. Which mode is actives depends on whether its *mode condition* resolves to a value 1, and whether it has the highest *mode priority* in case this holds for multiple modes. The priority is fixed, while the condition is a symbolic expression that can contain dynamic elements, which are evaluated at runtime. These conditions can use the value of a port slice, a register field slice, and other mode condition values.

Many elements can define modes:

1. Modes are referenced by the `interfaceModes` of ubs interfaces. For initiator and target `interfaceModes`, the referenced mode will be the only mode in which their reference `addressSpace` or `memoryMap` is accessible. As a result, an address map can be available at one point in time, while it may be inaccessible at another. Given that the interfaces of a bridge component now have dynamic modes, the bridge can alter the structure of a design. This concept is often applied in designs in the form of a network-on-chip (NoC) IP.

2. Modes are also used in `memoryRemap` elements. The `memoryRemaps` appear as optional elements in a `memoryMap`. Just like regular `memoryMap`, they can contain `addressBlocks`, banks, and `subspaceMap`. All of these elements will be added to the original enclosing `memoryMap` when its mode is active. As such, modes affect the contents of mapped memory.

3. Modes are also used in `alternateRegister` elements, which are potential elements of regular registers. When an `alternateRegister`'s mode is active, it will replace the containing register's definition.

4. Modes are referenced by port `fieldMap` elements, which maps register field bits onto port bits. Which `fieldMap` is active is based on which has the highest active mode. Hence, modes can directly alter port slice values, and thus directly affect other mode condition values.

5. Modes are also used in `accessPolicy` elements, which are referenced by an `addressBlock`, `bank`, `register`, `alternateRegister`, `registerFile`, and register field element in the form of *fieldAccessPolicy*. Without a mode, the access policy functions as the default for modes without `accessPolicies` referring them. The default is read-write, other options are read-only, write-only, read-writeOnce, writeOnce, and no-access. When memory elements reside in other memory elements, then the most restrictive policy is used.

When the one of them has write-only and the other read-only, then they both become no-access. As such, modes affect the accessibility of memory and maps.

From the affected IP-XACT elements it can be concluded that mode changes can alter the entire layout and accessibility of memory. This, in turn, can change runtime values, which can affect mode conditions again. This results in a dynamic problem regarding the verification of memory implementations that make use of modes.

For the purpose of a future work, we propose the following objectives.

The first objective is to extend our static AMG model to include the mode information into a dynamic AMG model. For example, edges may contain the accessibility information of an address map. This information will alter the structure of the graph based on mode conditions that are formulated in symbolic expressions. As such, the combination of all symbolic expressions result in a dynamic system.

The second objective is to find the algorithms to verify this dynamic AMG against a given static memory specification, and consequently calculate the symbolic expressions under which GBE holds.

Bibliography

[1] E. Pekkarinen, M. Teuho, and T. D. Hamalainen, "Analysis and visualization of product memory layout in IP-XACT," in *2017 Euromicro Conference on Digital System Design (DSD)*, Vienna, Austria: IEEE, Aug. 2017, pp. 155–162, ISBN: 9781538621462. DOI: 10.1109/DSD.2017.57. [Online]. Available: http://ieeexplore.ieee.org/document/8049780/.

[2] IEEE, "IEEE standard for IP-XACT, standard structure for packaging, integrating, and reusing IP within tool flows," *IEEE Std 1685-2022 (Revision of IEEE Std 1685-2014)*, pp. 1–750, Feb. 2023. DOI: 10.1109/IEEESTD.2023.10054520. [Online]. Available: https://ieeexplore.ieee.org/document/10054520.

[3] IEEE, "IEEE standard for universal verification methodology language reference manual," *IEEE Std 1800.2-2020 (Revision of IEEE Std 1800.2-2017)*, pp. 1–458, Sep. 2020. DOI: 10.1109/IEEESTD.2020.9195920. [Online]. Available: https://ieeexplore.ieee.org/document/9195920.

[4] Accellera. "UVM (universal verification methodology)." (2024), [Online]. Available: https://accellera.org/downloads/standards/uvm.

[5] A. B. Mehta, *ASIC/SoC Functional Design Verification: A Comprehensive Guide to Technologies and Methodologies*, 1st. Springer Publishing Company, Incorporated, May 2017, ISBN: 9783319594170.

[6] S. Qamar, W. H. Butt, M. W. Anwar, F. Azam, and M. Q. Khan, "A comprehensive investigation of universal verification methodology (UVM) standard for design verification," in *Proceedings of the 2020 9th International Conference on Software and Computer Applications*, ser. ICSCA '20, New York, NY, USA: Association for Computing Machinery, Apr. 17, 2020, pp. 339–343, ISBN: 9781450376655. DOI: 10.1145/3384544.3384547. [Online]. Available: https://doi.org/10.1145/3384544.3384547.

[7] A. O. Naik, E. Kuruvilla, and A. P. Chavan, "Integration and verification of IP cores on SoC," in *2021 IEEE Mysore Sub Section International Conference (MysuruCon)*, Oct. 2021, pp. 120–125. DOI: 10.1109/MysuruCon52639.2021.9641547. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9641547.

[8] A. Kamppi, L. Matilainen, J.-M. Maatta, E. Salminen, T. D. Hamalainen, and M. Hannikainen, "Kactus2: Environment for embedded product development using IP-XACT and MCAPI," in *2011 14th Euromicro Conference on Digital System Design*, Aug. 2011, pp. 262–265. DOI: 10.1109/DSD.2011.36. [Online]. Available: https://ieeexplore.ieee.org/document/6037418.

[9] "Kactus2 source repository." (2024), [Online]. Available: https://github.com/kactus2/kactus2dev.

[10] M. Teuho, E. Pekkarinen, and T. Hamalainen, "Visualization of memory map information in embedded system design," in *2018 21st Euromicro Conference on Digital System Design (DSD)*, Prague: IEEE, Aug. 2018, pp. 163–166, ISBN: 9781538673775. DOI: 10.1109/DSD.2018.00040. [Online]. Available: https://ieeexplore.ieee.org/document/8491811/.

[11] SystemRDL Working Group. "SystemRDL." (Jan. 26, 2018), [Online]. Available: https://accellera.org/activities/working-groups/systemrdl.

[12] A. Mykyta. "PeakRDL documentation." (2023), [Online]. Available: https://peakrdl.readthedocs.io/en/latest/.

[13] "PeakRDL source repository." (2024), [Online]. Available: https://github.com/SystemRDL/PeakRDL.

[14] Arm Ltd. "CMSIS system view description." (May 2, 2022), [Online]. Available: https://www.keil.com/pack/doc/CMSIS/SVD/html/index.html.

[15] IEEE, "IEEE standard for IP-XACT, standard structure for packaging, integrating, and reusing IP within tool flows," *IEEE Std 1685-2009*, pp. 1–374, Feb. 2010. DOI: 10.1109/IEEESTD.2010.5417309. [Online]. Available: https://ieeexplore.ieee.org/document/5417309.

[16] D. Michail, J. Kinable, B. Naveh, and J. V. Sichi, "JGraphT—a java library for graph data structures and algorithms," *ACM Transactions on Mathematical Software*, vol. 46, no. 2, 16:1–16:29, May 19, 2020, ISSN: 0098-3500. DOI: 10.1145/3381449. [Online]. Available: https://doi.org/10.1145/3381449.

[17] The Graphviz Authors. "DOT language," Graphviz. (Oct. 4, 2022), [Online]. Available: https://graphviz.org/doc/info/lang.html.

[18] E. Gansner, E. Koutsofios, S. North, and K.-P. Vo, "A technique for drawing directed graphs," *IEEE Transactions on Software Engineering*, vol. 19, no. 3, pp. 214–230, Mar. 1993, ISSN: 1939-3520. DOI: 10.1109/32.221135. [Online]. Available: https://ieeexplore.ieee.org/document/221135.

[19] The Document Foundation. "LibreOffice." (2024), [Online]. Available: https://www.libreoffice.org/.

[20] The Document Foundation. "LibreOffice documentation - starting LibreOffice software with parameters," LibreOffice 24.2 Help. (2024), [Online]. Available: https://help.libreoffice.org/latest/en-US/text/shared/guide/start_parameters.html.

[21] The Document Foundation. "LibreOffice documentation - CSV filter parameters," LibreOffice. (2024), [Online]. Available: https://help.libreoffice.org/latest/en-US/text/shared/guide/csv_params.html.

Acronyms

AAD address-axis diagram

AB address block

AMG address map graph

API application programming interface

AS address space

AUB address unit bits

bd bottom delta

GBE graph bitmapping equivalence

IF interface

IP intellectual property

lb lower bound

LMM local memory map

MM memory map

NoC Network-on-Chip

RTL register-transfer level

SCR semantic consistency rule

SG segment

SM subspace map

SoC system-on-chip

td top delta

TGI Tight Generator Interface

ub upper bound

XML eXtensible Markup Language

Appendix A

A

A.1 Bitmapping Example Calculation

Given is the example path in Figure A.1, with nodes u, v, w and edges $e_0 = uv$ and $e_1 = vw$. Node w is a leaf node such that its entire window is addressable. This results in the following given properties:

$$\begin{aligned}
\text{base}(u) &= 12 \\
\text{range}(u) &= 5 \\
\text{base}(v) &= 0 \\
\text{range}(v) &= 20 \\
\text{base}(w) &= 9 \\
\text{range}(w) &= 16 \\
\text{offset}(e_0) &= 9 \\
\text{offset}(e_1) &= -15
\end{aligned}$$

Then, we calculate the lower bounds and upper bounds of the path edges:

$$\begin{aligned}
\text{lb}(e_1) &= \max(\text{base}(v), \text{base}(w) + \text{offset}(e_1)) \\
&= \max(0, 9 - 15) = 0 \\
\text{ub}(e_1) &= \min(\text{base}(v) + \text{range}(v), \text{base}(w) + \text{range}(w) + \text{offset}(e_1) \\
&= \min(20, 25 - 15) = 10 \\
\text{lb}(e_0) &= \max(\text{base}(u), \text{lb}(e_1) + \text{offset}(e_0)) \\
&= \max(12, 0 + 9) = 12 \\
\text{ub}(e_0) &= \min(\text{base}(u) + \text{range}(u), \text{ub}(e_1) + \text{offset}(e_0) \\
&= \min(17, 19) = 17
\end{aligned}$$

Next, we calculate the bottom deltas and top deltas of the path edges:

$$\begin{aligned}
\text{bd}(e_1) &= \max(0, \text{lb}(e_1) - \text{base}(w) + \text{offset}(e_1)) \\
&= \max(0, 0 - 9 + 15) = 6 \\
\text{bd}(e_0) &= \text{bd}(e_1) + \max(0, \text{lb}(e_0) - \text{lb}(e_1) - \text{offset}(e_0)) \\
&= 6 + \max(0, 12 - 0 - 9) = 9 \\
\text{td}(e_1) &= \max(0, \text{base}(w) + \text{range}(w) + \text{offset}(e_1) - \text{ub}(e_1)) \\
&= \max(0, 9 + 16 - 15 - 17) = 0 \\
\text{td}(e_0) &= \text{td}(e_1) + \max(0, \text{ub}(e_1) + \text{offset}(e_0) - \text{ub}(e_0) \\
&= 0 + \max(0, 10 + 9 - 17) = 2
\end{aligned}$$

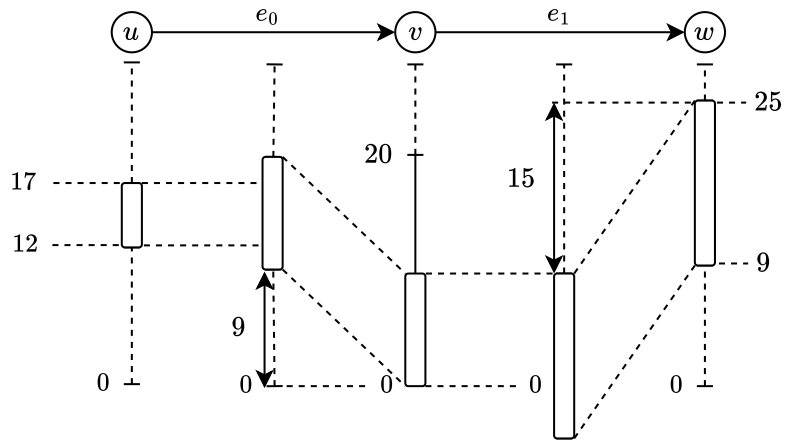


Figure A.1: Example path with two edges.

This path thus results in bitmapping $m = (12, 17, 9, 9)$. Finally, we calculate the codomain upper bound from using the bottom delta (right side) and top delta (left side):

$$\begin{aligned} \mathsf{base}(w) + \mathsf{range}(w) - \mathsf{td}(e_0) &= \mathsf{base}(w) + \mathsf{bd}(e_0) + \mathsf{ub}(e_0) - \mathsf{lb}(e_0) \\ 9 + 16 - 2 &= 9 + 9 + 17 - 12 \\ 23 &= 23 \end{aligned} \tag{A.1}$$

The equality holds, thus td and bd correctly represent the cumulative clippings of edges e_0 and e_1.

Test Case	Description	Expected Maximization	Actual Maximization
Single	One bitmapping.	Unchanged	Unchanged
Two disjunct	Two disjunct bitmappings.	Unchanged	Unchanged
Two partially overlapping, unequal codomain offsets	Two partially overlapping bitmappings with different codomain address offsets.	Unchanged	Unchanged
Two totally overlapping	Two totally overlapping bitmappings with equal codomain address offsets.	One remaining	One remaining
Two contiguous	Two bitmappings contiguous in domains and codomains, with equal codomain address offsets.	Merged into a single bitmapping	Correctly merged
Two contiguous in reverse	Two bitmappings with domains contiguous in one direction and codomains contiguous in reversed order	Unchanged	Unchanged
Double overlap	Four bitmappings. Two with the same domain bounds but disjunct codomains. Two with domains contiguous with the other overlapping domains from both sides, but the same codomains in the middle of and contiguous with the other disjunct codomains.	Two merged bitmappings, overlapping in the middle.	Correctly merged
2 contiguous options	Three bitmappings where the first is contiguous and has equal codomain offsets with other bitmappings, the last of which has largest range.	First and last bitmappings merged, second removed	Correctly merged
Multi choice	Five bitmappings. First bitmapping has 2 contiguous following, bitmappings, last of which is smallest, which has in turn two contiguous, first of which is largest. Where contiguous also the codomain address offsets are equal.	One bitmapping spanning all five and all five removed.	Correctly merged
Multi choice with noise	Same set as above, but with overlapping bitmappings between each contiguous bitmapping with a unique codomain address offsets.	Five bitmappings; one that spans all contiguous, and four unmerged with unique codomain address offsets.	Correctly merged

A.2 Bitmapping Maximization Correctness Test Cases