Sharing external memory resources between strongly isolated domains in high-end security applications

THESIS

submitted in partial fulfilment of the
requirements for the degree of

MASTER OF SCIENCE

in

EMBEDDED SYSTEMS

by

J.H.A. Kruijsse, BSc

Embedded Systems
Department of Electrical Engineering
Faculty of Electrical engineering, Mathematics and Computer Science
Delft University of Technology

Abstract

Nowadays, the society strongly depends on computer networks and systems as a means of reliable communication and data storage. In order to maintain absolute security of the networks and thus the society, one would need to separate everything, but this is not feasible. Consequently, sharing of resources is inevitable. There are **security products** that rely on an FPGA to create **domain separation**. The domain separation is required to prevent leakage of confidential information and manipulating of critical processes.

A modern FPGA has enough resources to have multiple soft-cores initiated on it- each of them working in a different domain. However, due to the limited amount of IO pins on an FPGA, using multiple DRAM chips is not an option. Therefore a single DRAM is shared between multiple soft-cores, threatening the domain separation.

The main **threats** when using a **shared DRAM** are communication channels due to **latency deviations**, data corruption due to **rowhammering** and **direct access** to unauthorized data due to the data being available on shared addresses. Research has been done to determine what causes the latency deviation and how to mitigate it. The results of the research are that the only fundamental solution to mitigate the latency deviation is to have a **fixed latency** when accessing the DRAM. A **fixed time arbiter** is designed and tested. The fixed time arbiter is using a deterministic delay after each DRAM access in order to mitigate the latency deviation.

Before mitigating the rowhammer vulnerability it is shown that rowhammering causes bitflips not only in the adjacent rows, but also in non-adjacent rows. To mitigate the rowhammer vulnerability for adjacent rows, a **row refresher** is created that tracks the rows that are accessed and refreshes the adjacent rows when accessed more than the bitflip threshold. To mitigate the vulnerability for non adjacent rows a test is created to give an overview of all non adjacent rows that contain bitflips so that those rows can be be dedicated as unused guard rows.

The last part that is implemented is an **address mapper** to be sure that no soft-core can access the addresses of another soft-core.

The fixed time arbiter, row refresher and address mapper are combined into the **memory domain protector**. The consequence on the bandwidth of the DRAM is that the bandwidth is halved compared to the benchmark design. The memory domain protector also uses 23× more logic than a standard arbiter.

Acknowledgments

This thesis marks the completion of my embedded systems studies at the TU Delft. I would like to express gratitude to my thesis advisers, Stephan Wong and Sjoerd op 't Land, for their support, guidance, and insights throughout the entire research process. Their encouragement, motivation, and constructive feedback were instrumental in shaping this thesis.

I would also like to thank the members of my thesis committee, Dr. ir. J.S.S.M. Wong, Dr. ir. T.G.R.M. van Leuken, and Dr. ir. S.T. Op 't Land, for their constructive criticism, suggestions, and for taking the time to review my work.

I am grateful to my colleagues, friends, and family who have provided me with continuous support, encouragement, and motivation throughout my academic journey. Their support was invaluable in keeping me motivated during the challenging moments of the research process.

I would like to thank the staff and administration of TU Delft for providing me with an exceptional academic experience, excellent resources, and research opportunities that have contributed to my professional development.

Thank you all for your support and contributions to this project.

Contents

Acknowledgments **B**

List of Figures **E**

List of Tables **F**

1 Introduction **1**
- 1.1 Problem statements and methodology . . . 1
- 1.2 Design considerations . . . 2
- 1.3 Overview . . . 3

2 Background **4**
- 2.1 Memory hierarchy . . . 4
- 2.2 AXI4 . . . 5
  - 2.2.1 AXI4 transfers . . . 5
  - 2.2.2 AXI4 package . . . 6
- 2.3 Side-channels . . . 6
  - 2.3.1 Rowhammer attack . . . 7
  - 2.3.2 Cache attack . . . 9
  - 2.3.3 Row buffer attack . . . 10
- 2.4 Conclusion . . . 10

3 Implementation **12**
- 3.1 Overview memory domain protector . . . 12
- 3.2 Fixed time arbiter . . . 13
  - 3.2.1 Desired behavior . . . 13
  - 3.2.2 Timing research . . . 14
  - 3.2.3 Design . . . 15
  - 3.2.4 Verification . . . 18
  - 3.2.5 Conclusion . . . 19
- 3.3 Rowhammer bitflip characteristics . . . 19
  - 3.3.1 Methodology . . . 19
  - 3.3.2 Results . . . 19
  - 3.3.3 Conclusion . . . 20
- 3.4 Row refresher . . . 21
  - 3.4.1 Addressing . . . 21
  - 3.4.2 Tracking accesses . . . 21
  - 3.4.3 Validation . . . 21
  - 3.4.4 Conclusion . . . 22
- 3.5 Address mapper . . . 22
  - 3.5.1 Implementation . . . 23
  - 3.5.2 Validation . . . 24
  - 3.5.3 Conclusion . . . 24
- 3.6 Conclusion . . . 24

4 Results **26**
- 4.1 Methodology . . . 26
- 4.2 Synthesis . . . 26
- 4.3 Performance . . . 26
- 4.4 Conclusion . . . 27

5 Conclusion **30**
5.1 Summary . . . 30
5.2 Main contributions . . . 31
5.3 Future work . . . 32

List of Figures

2.1 Typical memory structure of a dual-core processor connected to external DRAM. The CPU consists of two cores, each of them has their own private L1 and L2 caches. The Last level (L3) Cache and DRAM memory are shared by all cores [3]. 4
2.2 Overview of a DDR3 DIMM [6]. 5
2.3 This figure shows the direction of each individual AXI4 channel, with the manager being the Master and the Subordinate being the slave [8]. 5
2.4 An overview of the AXI4 channel handshake. However, the ready signal can be high only if the destination can accept the transaction [9]. 6
2.5 Overview of a write and read burst transfer [9]. 6
2.6 Attack tree for DRAM chip 8

3.1 Overview of the memory domain protector. Each arrow is an AXI4 bus and the left side is the slave and the right side is the master of each individual block. The rowhammer protector monitors the AXI4 bus between the fixed time arbiter and the DRAM controller. 12
3.2 This figure shows how each core has its own time slot to access the DRAM. During the *accept transfers* stage it can accept transfers. During the *time buffer* stage it can not accept transfers but transfers could still be finished and during the *block transfer* stage all transfers to that core are blocked. 13
3.3 Two transfers when there is no security on the latency yet. The first transfer starts within the *accept transfers* stage and is accepted immediately. The response from the DRAM can contain information about the other cores since the response time varies, depending on the state of the DRAM and the DRAM controller. The second transfer starts in the *block transfers* stage and only get accepted after some time. This adds extra latency, but this latency does not contain any information since the time that is added is already known before the transfer request is created. The duration of the transfer after it has been accepted can contain information about the other cores since the response time varies, depending on the state of the DRAM and the DRAM controller. 13
3.4 Creating a fixed duration of the transfer that only depends on the transfer length mitigates the timing side channel. Since all transfers are determined before the transfer has started. 13
3.5 Design used to measure all signals between the traffic generator an the DRAM controller. 14
3.6 Overview of the design for the Fixed Time Arbiter. 16
3.7 The seven states of the AXI4 channel controller 18
3.8 Address mapping on a DRAM 21
3.9 Overview of the implementation for the rowhammer refresher. It consists of 2 parts: a pipelined tracker and Block RAM. 22
3.10 Overview of the pipeline within the rowhammer refresher. There are 3 stages. Stage 1 sends a read request to the Block RAM. Stage 2 receives the data from the block RAM, checks if the threshold has been exceeded and then it increases the counter or set the counter on zero, before writing it back to the Block RAM. The last stage generates the corresponding AXI read requests in order to refresh the targeted rows in the DRAM. . . 22
3.11 Overview of the address mapping with a separate control logic for each of the 5 AXI4 channels. 23

4.1 Flowchart of methodology of the bandwidth measurements. 27
4.2 Bandwidth for the standard round robin arbiter. The x-axis shows the burst transfer length and no the y-axis shows the bandwidth. 28
4.3 Bandwidth for the secure shared DRAM. The x-axis shows the burst transfer length and no the y-axis shows the bandwidth. 29

List of Tables

3.1 AXI4 latencies . . . 15
3.2 Every row in the DRAM is hammered and it is measured in which rows it caused bitflips. The rows with bitflips describes where the bitflips are after a single row has been hammered. The Nr. of hammered rows are the amount of rows that when hammered they caused bitflips as described in the column rows with bitflips. . . . 20
3.3 List of rows that got bitflips by hammering non of its adjacent rows. . . . 20

4.1 FPGA resources after synthesis. The available resources are the available resources on the Arty A7-100t. The Round Robin arbiter is the benchmark for the resources used. The total is the total of the rowhammer protecion, fixed time arbiter and the address map. . 27

1 Introduction

Nowadays, the society strongly depends on computer networks and systems as a means of reliable communication and data storage. To keep those networks and thus the society secure, the computer networks and systems depend on security products.

These security products depends on domain separation to make sure that there is no interference between different communication channels and none of the communication can leak to unauthorized domains.

To make the domain separation possible, the security products uses several soft-cores. A soft-core is a digital circuit that can be wholly implemented on e.g. FPGA, ASIC, CPLD by using logic synthesis. Each soft-core is running in a different security domain. The number of soft-cores scale well with the amount of resources available on an FPGA. However, the due to the limited amount of IO pins a single DRAM is shared between the soft-cores.

Sharing a single DRAM between multiple soft-cores threatens the domain separation since data might leak through the DRAM from one soft-core to another.

A solution is needed to mitigate this threat. This solution should consist of allocating an address space to each of the security domains that can not be accessed by other security domains. It should also provide security against malicious activities that try to leak or change data from one security domain to another on purpose.

1.1. Problem statements and methodology

The main research question is:

How can a single DRAM chip be shared between multiple soft-cores without leaking information from one soft-core to another soft-core?

Within this project, it is assumed that an attacker deliberately tries to access data on critical security products. Even if the attacker has software root access to one or multiple soft-cores within these security products, there should not be any form of data transfer possible from one core to another.

Some of these security products run in a physically protected environment. In this project it is assumed that an attacker has no physical access to the security product. Therefore, any hardware related attacks are out of the scope of this project. Furthermore, it is assumed that the attacker has no access to the reconfigurable part of the FPGA either, limiting the focus of this project to software attacks only.

From within software there are two ways that unauthorized data accesses can happen through the DRAM: directly access the address on the DRAM that contains the data, and indirect communication.

To make indirect communication happen, the attacker monitors physical quantities that could contain information about what is happening on the system in another security domain. For example, when the DRAM access is occupied by another soft-core and the attacker tries to access the DRAM, it will take longer, since the attacker needs to wait for the DRAM to be available. This waiting time carries information about the other soft-core: it is accessing the DRAM. This extra information that can be gathered because of the fundamental way a computer protocol or algorithm is implemented is called a side-channel.

Side-channels can also be used to setup a communication channel between different soft-cores that are not allowed to communicate with each other. In this case, it is called a covert-channel. Taking the previous example, when the waiting time to access the DRAM takes longer than usual, a ’1’ is communicated. If the waiting time is normal, a ’0’ is communicated makes a covert-channel using a side-channel.

For covert-channels, the attacker needs his software running on both soft-cores to be able to transmit data from one soft-core to another. By contrast, for side-channel attacks the attacker only needs access to a single soft-core.

For crucial security products, it is important that both side-channels and covert-channels are mitigated. Since covert-channels use side-channels as communication channel, it is sufficient to mitigate the side-channels.

Different side-channels are possible in different situations. Therefore the side-channels need to be identified first. After that the identified side-channels needs to be mitigated. On top of this the direct unauthorized data accesses should also be mitigated.

This leads to the following underlying sub questions and utilized methodology to answer the question:

1. **How to prevent direct unauthorized data access on a shared DRAM?**

   - A literature study to determine current existing methods to share DRAM between soft-cores.
   - Discuss the existing methods to share DRAM.
   - Implement a suitable design to share DRAM for this project.
   - Evaluate the functionality of the design.

   The implemented design should be able to guarantee that a core can only access its allocated addresses. After the direct unauthorized data access is mitigated, the focus is on the indirect communication, resulting in the question and approach:

2. **What side-channels are introduced that can be exploited through software, when sharing a DRAM chip between multiple soft-cores?**

   - A broad literature study will determine what classes of side-channels exists.
   - The findings of the literature study will be discussed with security experts to minimize the chance that any side-channel is missed.
   - Create a list with side-channels that need to be mitigated within this project.

   The following question follows from the list of side-channels:

3. **How can DRAM side-channel attacks be mitigated?**

   - Discuss countermeasures found in literature.
   - Design countermeasures for each side-channel in the previously created list.
   - Test the functionality of the countermeasures.

   After every aspect of this project is implemented, the influence on the performance needs to be determined resulting in the last question and chosen approach:

4. **What are the consequences of the implemented countermeasures on the performance?**

   - Create a benchmark test and run this test on a system with and without the mitigation.
   - Discuss the comparison of the results.

1.2. Design considerations

For this project, security is the most important aspect for the final design. Furthermore, the design should be easy to understand, so that when used in future projects the security can be proven by inspection. Therefore two design principles are considered: Security over performance and simplicity over complexity.

It is also assumed that each softcore runs its own operating system and there for it does not rely on any data or instruction sharing between different softcores.

1.3. Overview

The remainder of this report is divided into four chapters. Chapter 2 gives an overview of the required background information to understand the rest of the report. It describes the fundamental hardware architecture of a DRAM chip and how a typical memory hierarchy looks like. This is followed by a short introduction of the AXI4 bus that is used for within every hardware design in this project. Following that, the chapter gives an overview of the currently known side-channels of DRAM and explains which side-channels are mitigated in this project.

Chapter 3 contains a description of the complete design that is implemented and tested to safely share DRAM between multiple soft-cores. Chapter 4 analyzes the performance tradeoff of the implemented design. It compares the new design with a consisting benchmark design. Finally, Chapter 5 summarized the work that is done and gives an overview of the main contributions. The report then finishes with an overview of potential future work.

2
Background

This chapter will provide an overview of the security vulnerabilities of DRAM and the current solutions against those vulnerabilities. It will also show where current solutions are lacking.

Section 2.1 describes the typical memory hierarchy of a computer and it explains the architecture of a DRAM chip. Section 2.2 describes the AXI4 protocol that is used for the communication between the softcores and the DRAM controller. Section 2.3 describes the different side-channels found in a computer. This section also gives an overview of the related work of each of the relevant side-channels found. It will explain where the existing solutions are relevant and where they are lacking for the purpose of this work. At last, Section 2.4 gives a summarizing conclusion of this chapter.

2.1. Memory hierarchy

To understand the side-channels caused by the DRAM, it is important to understand how the DRAM works and how the interaction is within the memory hierarchy.

A typical memory architecture of a multi-core system is shown in Figure 2.1. The memory architecture of a multi-core system has typically one or two private caches per core that communicate with a shared last level cache between all cores [1]. The last level cache is connected with the DRAM.

The data in the L1 cache is coherent with the data in the L2 cache, which is coherent with the data in the last level cache, which is coherent with the data in the DRAM [2]. This architecture is created to optimize the latency and bandwidth for frequently accessed data, because smaller caches close to the core have a better latency than DRAM, which is further away from the core.

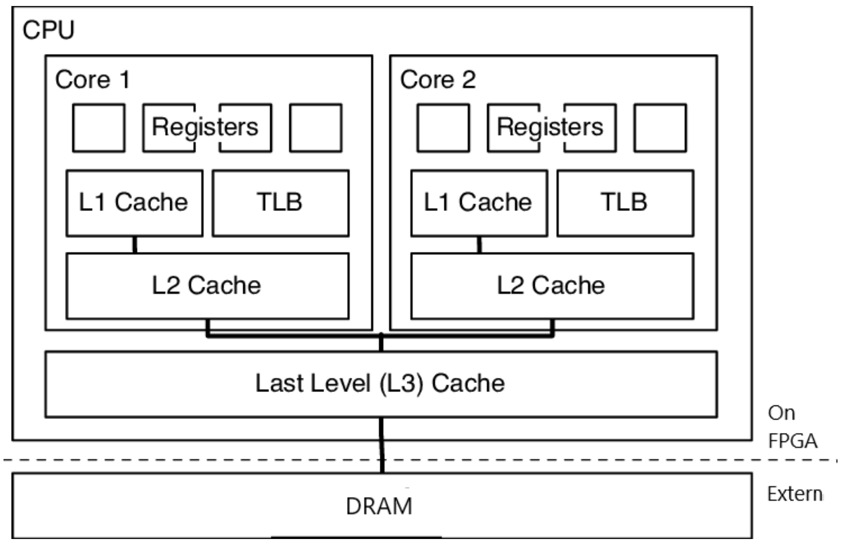


Figure 2.1: Typical memory structure of a dual-core processor connected to external DRAM. The CPU consists of two cores, each of them has their own private L1 and L2 caches. The Last level (L3) Cache and DRAM memory are shared by all cores [3].

Figure 2.2 shows the typical structure of the DDR (double data rate) DRAM structure. In most computers the DDR DRAM structure consists of a DDR DIMM. On the DIMM, there are several ICs with several banks. A bank consists of a cells that are connected through a wordline and a bitline. Each cell consists of a transistor and a capacitor. For this project, a single DRAM IC is connected to an FPGA, meaning that the address only contains information about the bank, row and column.

In DRAM, data is stored in the charge over the capacitor (e.g. charged is a ’1’, discharged is a ’0’). The charge over a capacitor will always change to its default state over time due to charge leakage of the capacitor. The default can either be charged or discharged depending on if the capacitor is connected to a pull up or pull down circuit. To prevent data loss due to this charge leakage, the DRAM specifications require a refresh every 64 ms for each individual row [4, 5].

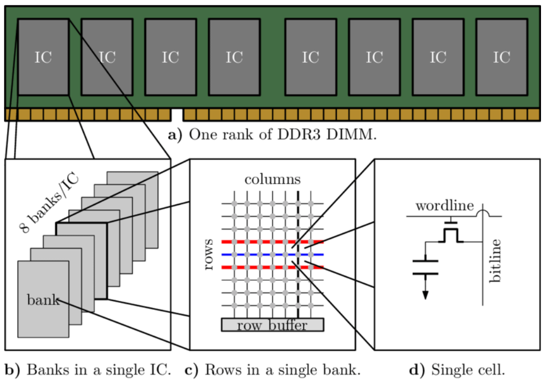


Figure 2.2: Overview of a DDR3 DIMM [6].

2.2. AXI4

The softcores that are used for this project uses the AXI4 bus to access all the peripherals including the DRAM. In this section, the basics of the AXI4 are explained based on the official AXI4 documentation [7]. After this explanation, an implementation is shown that is used later on in this project.

The AXI4 protocol is widely used for micro controllers and SoCs. The AXI4 specifications describe a point-to-point protocol between two interfaces: a master and a slave. It has 5 channels: read address (ar), write address (aw), read data (r), write data (w) and write response (b), see Figure 2.3.

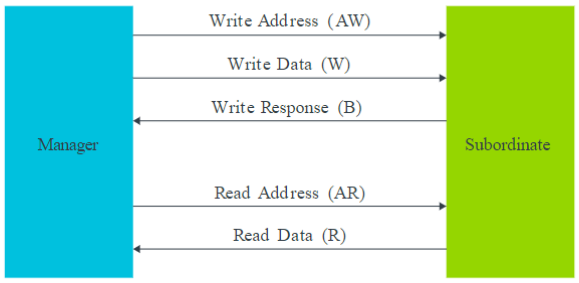
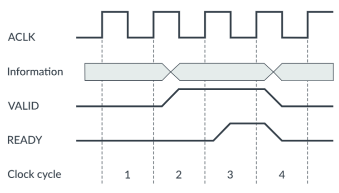


Figure 2.3: This figure shows the direction of each individual AXI4 channel, with the manager being the Master and the Subordinate being the slave [8].

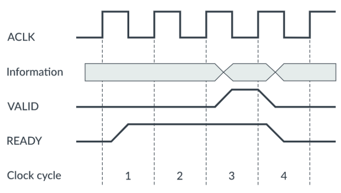
Each channel uses a two-way handshake to send data from source to destination. The handshake happens on the rising edge of the clock when both the *ready* and *valid* signals of a channel are high, see Figure 2.4. When a handshake happens the transaction is complete.

2.2.1. AXI4 transfers

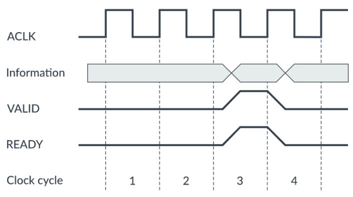
There are two types of transfers, a write transfer and a read transfer, see Figure 2.5. Both transfers consists of multiple transactions over multiple channels. A read transfer starts with a transaction over the ar channel and contains information about the address, burst type, length and size of the requested read transfer. After the ar transaction the r channel will be used to send the read data to the master.



(a) Valid is asserted before the ready.



(b) Ready is asserted before the valid.

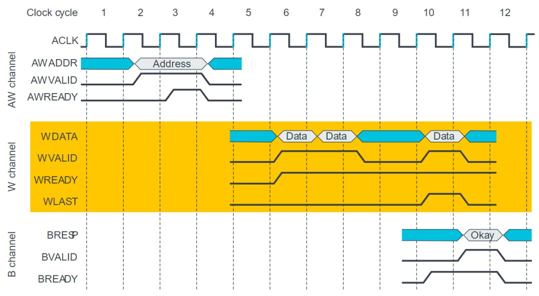


(c) Ready and valid are asserted at the same time.

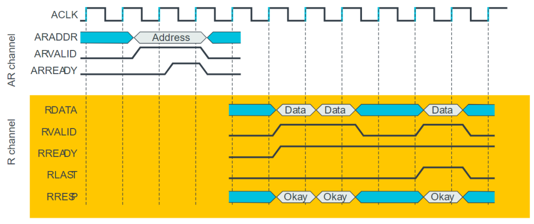
Figure 2.4: An overview of the AXI4 channel handshake. However, the ready signal can be high only if the destination can accept the transaction [9].

It can consist of multiple transactions on the r channel and each transaction contains a part of the requested data and information about whether or not the address was valid.

A write transfer starts with a transaction on the aw channel or on the w channel. The aw channel contains information about the address, burst type, length and size of the requested write transfer. The w channel contains the actual data and information about which bytes of the data are valid and information about whether it is the last data transaction. After all the transactions of the aw channel and the w channel have been finished, there will be a transaction on the b channel with information of a successful transfer or not.



(a) Write transfer



(b) Read transfer

Figure 2.5: Overview of a write and read burst transfer [9].

2.2.2. AXI4 package

To make the use of AXI4 a bit easier a VHDL package is used that combines all AXI4 signals from the master into a single type *req* and all signals from the slave into a single type *resp*. Both the *req* and the *resp* have as subtypes the *aw, ar, w, b* and *r* channels which contain all signals of the AXI4 bus. From now on in this report *req* is used for all signals from the AXI4 master to the AXI4 slave and *resp* is used for all signals from the AXI4 slave to the AXI4 master. When talking about a single channel *req.channel_name* and *resp.channel_name* are used and for specific signals *req.channel_name.signal_name* and *resp.channel_name.signal_name* are used. For example *req.r.ready* is used to access the ready signal of the r channel and *resp.r.data* is used to access the data from the r channel.

2.3. Side-channels

Any unintended information that can be gathered by the way a computer algorithm, protocol or hardware is implemented is defined as a side-channel and thus a potential threat to the security [10, 11, 12, 13, 14, 15]. Side-channels found in literature and after discussion with security experts at Technolution's are:

- Power: The voltage on or current through supply rails contain information about the systems and its application [16].
- Electromagnetic: The electromagnetic radiation contains information about the system and its application [17].
- Timing: Latency and duration of an instruction can hold information [18].

- Visual: For example, status LEDs, UART LEDs, LEDs on the hard drive etc. can contain information about the data that is being transferred.
- Sound: The vibration of electrical components, usage of keyboards, etc. creates sounds that contain information [19].
- Data remanence: Data that is left even after attempts at erasing the data [20].
- Photon emission: Photon emissions from switching transistor can be related to the program running on the chip [21].
- Thermal: The temperature can contain information about the program running on the chip [22].

There are many types of side-channels, but as stted earlier, this project is constrained to side-channels that can be exploited through software only. The attack tree of Figure 2.6 is created to find all possible ways an attacker can exploit the side-channels through software. It is import to note that after the attack tree branches to an attack that requires physical access or that requires an extra sensor attached to any of the soft-cores the research stopped, because it is assumed that the attacker has no physical access and there are no sensors attached to any of the softcores. The only physical quantity that can be measured by the software running on the core is elapsed time in passed clock cycles. For the attack tree, it is also assumed that the attacker has root access to the soft-core and therefore the attacker can access any memory address that is reachable from the soft-core.

The starting point is that an attacker wants to gather unauthorized data from another security domain through the memory. This can be done directly by accessing the data stored in the memory or indirectly by measuring side-channels. Direct memory access should be mitigated. The side-channels can be measured in two ways, directly by using probes or specific hardware on the FPGA, and indirectly by manipulating the different IPs connected to the soft-core. The probes and specific hardware are not in the scope of this project since it is assumed that the attacker has no physical access or access to the reconfigurable parts of the FPGA.

The indirect manipulation and measurements consist of two categories: voltage changes in the DRAM, which cause bitflips and manipulation of the timing of an application in another security domain due to the limited amount of resources.

2.3.1. Rowhammer attack

As shown in Section 2.1, the actual data is stored as voltage over a capacitor and needs to be refreshed every 64 ms. However, when the wordline is being opened repetitively using the activate command from the DRAM controller the voltage changes on the wordline influence the charge over the capacitor of cells in adjacent rows, adding a probability of bitflips in those cells. If the adjacent rows have not been refreshed or activated recently, the charge of the repetitively activated row leaks into the the dormant adjacent rows and causes a bit to flip. This exploit is called rowhammering [23, 24, 25, 26, 27, 28, 29].

The first documentation about rowhammering came from Kim et al. in 2014 [25]. They found out that almost every DRAM module (110 out of 129 of 3 different manufacturers) are sensitive to rowhammering. It took them as little as 139k activations to cause a bit flip in an adjacent row. However, it worsened after a similar study was done a few years later with the more modern DDR4 DRAM modules [26]. These new studies showed that modern DDR4 DRAM modules are even more sensitive for rowhammering than the previous generation: 300 DRAM modules were tested and bitflips where introduced after as few as 9.6k activations.

The attacker can use rowhammering to manipulate the data stored in the DRAM that belongs to another security domain. This manipulation can even be used to setup a covert-channel, since the attacker could create a protocol that if the data is manipulated it transmits a 1 otherwise it transmits a 0 or vice versa. Therefore, it is crucial to mitigate this threat.

Fundamental solutions against rowhammering

Since the data is stored as voltage over a capacitor within a DRAM cell, the only way to prevent bitflips is to refresh the charge before the voltage drops below a certain threshold. In normal usage the refresh frequency is 64ms, which is more than sufficient to prevent bitflips. However, due to rowhammering the voltage drops faster and might drop too much before it is refreshed. Therefore all mitigation techniques against rowhammering in DRAM consists of either adding extra refreshes, blocking repeating memory accesses, or detect bitflips and correcting them.

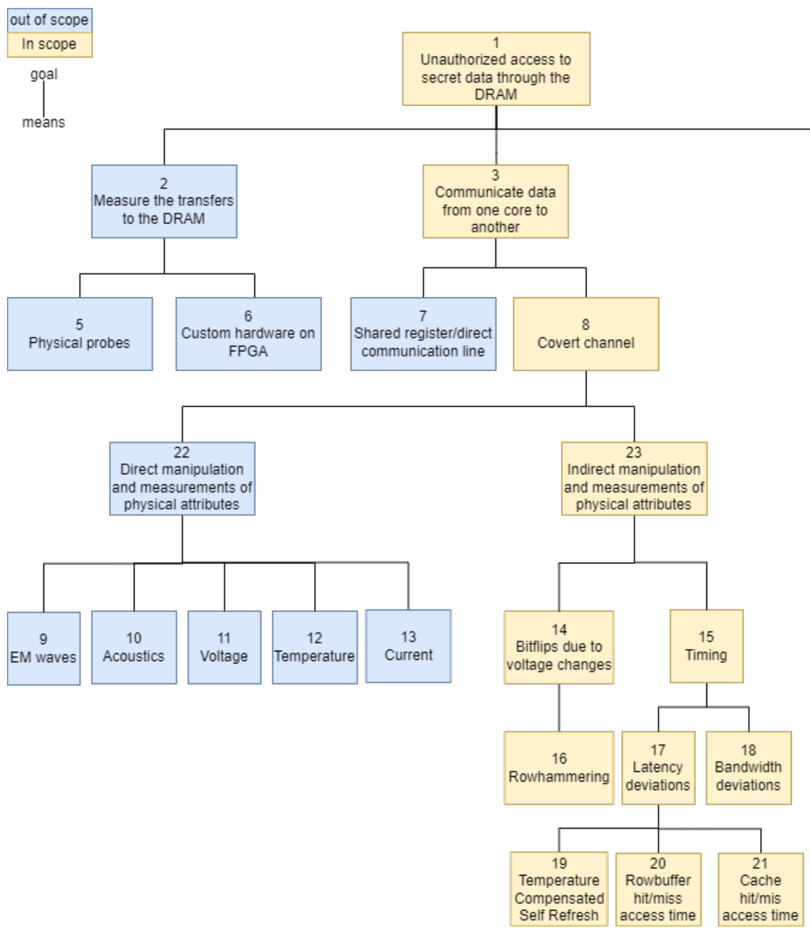

out of scope
In scope
goal
means
1 Unauthorized access to secret data through the DRAM
2 Measure the transfers to the DRAM
3 Communicate data from one core to another
4 Directly access the data from DRAM
5 Physical probes
6 Custom hardware on FPGA
7 Shared register/direct communication line
8 Covert channel
22 Direct manipulation and measurements of physical attributes
23 Indirect manipulation and measurements of physical attributes
9 EM waves
10 Acoustics
11 Voltage
12 Temperature
13 Current
14 Bitflips due to voltage changes
15 Timing
16 Rowhammering
17 Latency deviations
18 Bandwidth deviations
19 Temperature Compensated Self Refresh
20 Rowbuffer hit/miss access time
21 Cache hit/mis access time

Figure 2.6: Attack tree for DRAM chip

In modern DRAM chips an Error Correction Code (ECC) is used to detect and correct bitflips. However, it can only correct a single bitflip, if there are multiple bitflips in the data it can not correct them properly [27]. There are other error detection and correction algorithms but none of them are a 100% error free and they all have a significant impact on the performance [30, 31, 32]. Therefore preventing bitflips is preferred over correcting them.

The only way to prevent bitflips due to rowhammering is to make sure that each row is refreshed before the voltage within a DRAM cell is dropped below a certain threshold. It is also possible to accept that bitflips can happen within a security domain and add some empty rows between the different security domains. This gives four methods to prevent bitflips from happening in one security domain due to hammering in the other:

- Block frequently accessed rows until a refresh of its neighbors has been established [33].
- Targeted refreshes: Keep track of how often each row is accessed and if its more than a predetermined threshold refresh its neighbors [25, 34, 35].
- Refresh one of the neighbor rows with a probability every time a row is accessed [25, 36, 37, 38, 39, 40].
- Add guard rows between each security domain, this will prevent that rowhammering in one security domain causes bitflips in another security domain.

Tracking frequently accessed rows

Blocking frequently accessed rows and targeted refreshes both requires a table to keep track on which rows are accessed a lot. Blocking the access of a row can cost performance bandwidth and latency wise, since instructions of the softcore might wait on the blocked access to finish. Therefore it is preferred to refresh the neighbor rows instead of blocking the frequently accessed row.

Keeping track on how often each row is accessed requires area on a FPGA . For example, a DRAM chip with 2^3 = 8 banks, 2^{14} = 16384 rows and 2^{10} = 1024 columns will need $16384 * 8 = 131072$ entries. A bitflip can occur after 10000 accesses of a neighbor row before a refresh happened [26]. Therefore, each counter needs to be able to count to 10000/2 = 5000. The division by 2 is required, because the number of accesses are divided between both neighbors. Therefore, the total size needed for the table is $131072 * 2$ bytes = 256 kB.

Graphene [35] and TWiCe [34] reduced the total size of the table by implementing algorithms that guarantee that the most accessed rows are kept in the table. Graphene uses significantly less area compared to TWiCe, while it offers the same amount of protection and only has slightly more performance overhead [35]. Graphene uses the Misra-Gries algorithm [41] to guarantee that there are no bitflips due to more than 10000 accesses of all neighbor rows. This algorithm reduces the table size by 7.7 times, compared to the example above.

Probability based refreshes
Another approach is to refresh neighbor rows with a certain probability after every DRAM access [25, 36, 37, 38, 39, 40]. This will reduce the amount of bitflips due to rowhammering significantly. However, it also means that there will always be a chance that the neighboring row of the hammered row is not being refreshed in time. Although this chance is small, it is still not sufficient for the purpose of this research.

Guard rows
The last solution is putting guard rows between security domains. Guard rows are unused rows that none of the security domain can access. The idea behind guard rows is that if an attacker hammers a row in one security domain it can only cause bitflips within the same security domain or in the empty guard rows next to the allocated addresses in the DRAM. This will prevent bitflips from happening across security domains due to rowhammering.

However, there is a problem: In some DRAM chips the neighboring physical row might not be the same as the neighboring address row due to a remap of the addresses of the manufacturer[25]. Therefore to successfully use guard rows against rowhammering it is required to know which row influences which row.

2.3.2. Cache attack

The last level cache is typically shared between the different softcores and is located between the private caches and the DRAM. However, this typical design causes various vulnerabilities. The most obvious vulnerability is that another softcore can directly access private data from another softcore (nr. 4 in Figure 2.6) and even if the direct access was not possible it leaves a timing side-channel open.

There are several ways the timing side-channel can be exploited. One way the timing side-channel can be exploited is the Flush + Reload attack [15]. Flush + Reload attack use shared pages between the victim process and the spy process. The spy can ensure that a specific memory line is evicted from the whole cache hierarchy and then this is used by the spy to monitor the accesses to the memory line.

This attack consists of three phases. First, the monitored memory line is being flushed by using the *clflush* instruction. In the second phase the spy waits to allow the victim to access the memory line. Then in the last phase, the spy reloads the memory line, measuring the time it takes for the spy to load the data into the cache. Since loading the data from the DRAM will take significant more time compared to when the data is available in the cache, the spy knows whether or not the cache line has been accessed.

If the memory line is mapped to an instruction within an executable file, a probe is created that triggers every time the linked instruction that is being executed. The attacker could even chose to probe multiple memory lines each mapped to a different instruction. This gives the attacker information about what instruction is being executed at what time. If the executable is depending on a secret key and the instructions for a '0' are different than for a '1' then the probes created by the Flush + Reload attack can steal the key that is being used in the executable file.

The Prime + Probe attack exploits the cache hit and miss timings without having shared memory between the victim and the attacker [42]. Here there is no shared memory between the victim and the attacker. The attacker fills the memory with his own data from the memory, then waits so the victim has time to access his data and instructions, then the attacker accesses all his memory addresses again while measuring the load time.

Now the attacker knows that the victim has accessed the memory if the load time was slow, due to a cache miss, or did not access the memory when the load time was fast, due to a cache hit.

The principle of Flush + Reload and Prime + Probe is similar: get information by exploiting the cache hit and cache miss timings. There are more variations on these types of attacks, like Flush + Flush, Flush + Evict, Evict + Time[43, 44, 45, 46], but it all uses the same principle of the cache hit/miss latency.

All cache attacks rely on a shared cache between different softcores and most of the attacks make use of the cache attacks rely on the *clflush* instruction [43, 15, 11, 12]. The restriction of the *clflush* command is suggested as possible countermeasure against cache attacks [15, 47]. However, Gruss et al. showed that it is possible to execute cache attacks even without the *clflush* instruction available [48]. Therefore restricting the *clflush* is not sufficient to secure the memory hierarchy. They also stated that preventing cache-line sharing is not an option due to the fact that an operating system uses a lot of shared data. However, in the case of this study each softcore runs its own operating system and therefore preventing cache-line sharing is a possibility. Hence, it is even possible to not share any cache between the different cores, which would make a cache attack impossible.

2.3.3. Row buffer attack

Something similar to cache attacks can be exploited in the DRAM. When an address on the DRAM is accessed it loads the data stored on the cells of the DRAM into the row buffer, refer to Figure 2.2. The data will stay in the row buffer until an address is requested from another row or if the DRAM controller actively closes the row. Each bank has its own row buffer that can contain data independently from row buffers in other banks.

If the data is already available in the row buffer, the latency of the response will be lower than if the data is not available in the row buffer yet. This behavior can be exploited in a similar way as with the cache attacks. Liu et al. found 24 different kinds of DRAM timing-based attacks using computation tree logic [49]. Even though they found 24 different attack types, the root of each of the vulnerabilities is the row buffer hit/miss latencies.

It is even possible to reverse engineer which addresses are on each row and what rows are on the same bank using the latency differences [50]. Knowing the exact architecture of the DRAM even helps an attacker with a rowhammer attack because when the layout of the DRAM is known, it is possible to target specific rows during the rowhammer attack.

There are several papers describing how to exploit the row buffer hit/miss timing [51, 52, 49]. However, none of those papers came up with a mitigation technique. The only security solution came from Pessl et al., stating that the only counter-measure to mitigate the threat is to make the duration of corresponding DRAM operations constant [50]. Pessl et al. also states that the performance loss due to this solution is unacceptable. However, although this last statement might be true for most of the applications, for some specific applications the DRAM performance might still be sufficient with the constant timing.

2.4. Conclusion

This chapter gave an overview on how a typical memory hierarchy looks like. It explained how the architecture of DRAM consists of banks, rows and columns and that a cell consists of a capacitor and a transistor. The data is stored as electric charge in the capacitor and the data can be loaded in to the row buffer by activating the corresponding row.

An overview is given about the basic functionality of the AXI4 protocol. For the complete specifications it is recommended to read the official documentation of the AXI4 protocol [53].

It is explained what side-channels and covert-channels are and how attackers can exploit them. There are a lot of different types of DRAM attacks but there are only three relevant: Rowhammering, cache-timing attacks and rowbuffer timing attacks.

Because the data is stored as electric charge over a capacitor, the charge needs to be refreshed every 64ms to prevent data loss. Rowhammering causes the charge in adjacent cells to leak below a threshold causing bitflips before the periodic refresh on 64ms. To prevent this, it is possible to block the access to the hammered row until its neighbors have been refreshed, detect the hammered row and refresh its neighbors, on every row access refresh its neighbors with a small probability or use guard rows. Detection of the hammered row(s) and refresh its neighbors in combination with guard rows is the most effective against rowhammering. Although the downside is that it costs a lot of resources to keep track of all row activations, in this research it is most suitable because of its simplicity and effectiveness against bitflips compared to probabilistic refreshes and blocking frequently accessed rows.

Since there is no shared operating system, data or instructions between any softcore, shared cache is not required and therefore the cache attack can be avoided. Each individual softcore can still have its own private cache if required.

Currently there is no defense against the row buffer timing attack. In the literature it is suggested that the only potential solution is to make the DRAM operations a constant time, but that would cost too much performance. For this research security is the most important feature and therefore it should be researched what the performance loss is and if the performance is still sufficient to be used in the security products.

3
Implementation

This chapter will describe the implementation of the memory domain protector, an hardware block written in VHDL to secure the usage of DRAM when shared between multiple softcores. Section 3.1 describes the top level of the memory domain protector which consists of a fixed time arbiter, a row refresher and an address mapper. The fixed time arbiter is described in Section 3.2, the row refresher is described in Section 3.4 and the address mapper is described in Section 3.5.

For the fixed time arbiter an experiment is executed to determine the parameters for the implementation. This experiment is described in Section 3.2.2. Another experiment is done to determine which rows needs to be tracked and refreshed for the implementation of the row refresher. This experiment is described in Section 3.3.

3.1. Overview memory domain protector

Figure 3.1 shows the overview of the memory domain protector. The memory domain protector uses a fixed time arbiter, row refresher and address mappers to share the DRAM in a secured manner. The row refresher protects the DRAM from rowhammering, the fixed time arbiter mitigates the DRAM timing attacks and the address mappers make sure each softcore can only access its own memory domain.

The amount of AXI4 slaves of the fixed time arbiter is one AXI4 slave more than the amount of softcores using the memory domain protector due to the rowhammer protector. This adds an extra time slot reserved for the rowhammer protector and therefore will cause extra performance loss.

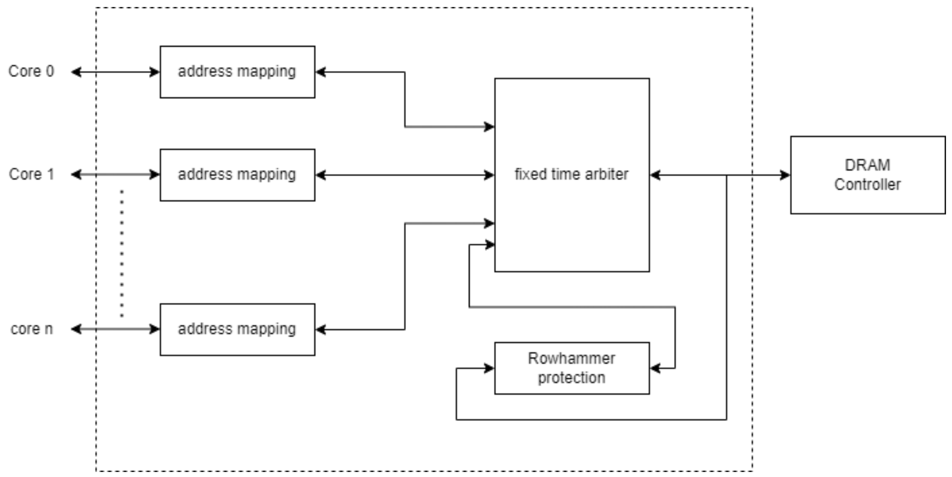


Figure 3.1: Overview of the memory domain protector. Each arrow is an AXI4 bus and the left side is the slave and the right side is the master of each individual block. The rowhammer protector monitors the AXI4 bus between the fixed time arbiter and the DRAM controller.

3.2. Fixed time arbiter

The solution to mitigate the timing side-channel of the rowbuffer is the fixed time arbiter (FTA). The goal of the FTA is to have a predetermined bandwidth and a predetermined latency for each of the softcores that does not contain any information about the other softcores that share the same DRAM. The required behavior to reach this goal is described in Section 3.2.1. In order to have a predetermined latency it is required to know the latency deviation of the DRAM. The experiment to determine the latency deviations of the DRAM is described in Section 3.2.2.

3.2.1. Desired behavior

To have a constant bandwidth for each individual softcore it is required to set time slots in which each softcore can access the DRAM, see Figure 3.2. After each time slot a time buffer is required to make sure all transfers are finished before the next core has access to the DRAM. However, the time slots on their own are insufficient to prevent any type of latency deviation. In Figure 3.3 is shown what stage of the transfer time could be manipulated to contain information about the other cores. To prevent this, extra time can be added to the transfer time, see Figure 3.4. The extra time should be chosen in such a way that the total transfer time consists of a fixed latency plus the minimal transfer time required. The fixed latency should be big enough to encompass any latency deviation from the DRAM and DRAM controller. The *time buffer*, see Figure 3.2, Figure 3.3 and Figure 3.4, should be equal to bigger than the fixed latency so the transfer is finished before switching to the next core. However, this only works when the fixed duration is bigger or equal to the maximum duration possible.

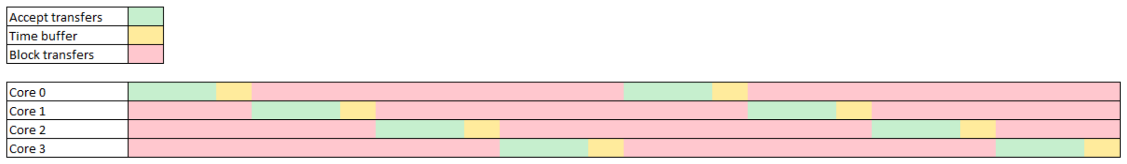


Figure 3.2: This figure shows how each core has its own time slot to access the DRAM. During the *accept transfers* stage it can accept transfers. During the *time buffer* stage it can not accept transfers but transfers could still be finished and during the *block transfer* stage all transfers to that core are blocked.

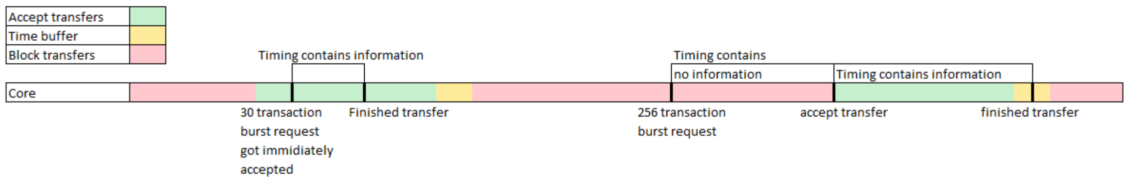

Accept transfers
Time buffer
Block transfers
Timing contains information
Timing contains no information
Timing contains information
Core
30 transaction burst request got immidiately accepted
Finished transfer
256 transaction burst request
accept transfer
finished transfer

Figure 3.3: Two transfers when there is no security on the latency yet. The first transfer starts within the *accept transfers* stage and is accepted immediately. The response from the DRAM can contain information about the other cores since the response time varies, depending on the state of the DRAM and the DRAM controller. The second transfer starts in the *block transfers* stage and only get accepted after some time. This adds extra latency, but this latency does not contain any information since the time that is added is already known before the transfer request is created. The duration of the transfer after it has been accepted can contain information about the other cores since the response time varies, depending on the state of the DRAM and the DRAM controller.

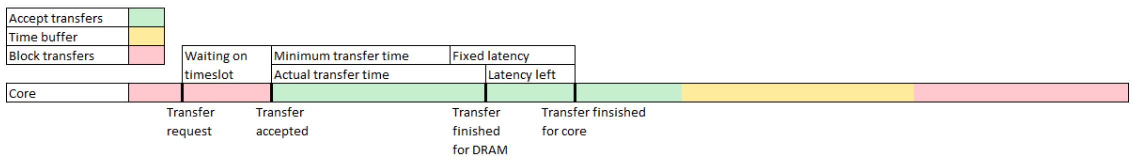


Figure 3.4: Creating a fixed duration of the transfer that only depends on the transfer length mitigates the timing side channel. Since all transfers are determined before the transfer has started.

3.2.2. Timing research

The goals of this experiment are to find the latency deviation of a DRAM access and to find the time buffer required for the fixed time arbiter. The reason to determine the latency through experimentation is because the DRAM and DRAM controller are from the manufacturer and it is not clearly documented what the maximum latency is.

For this experiment a simple design is created consisting of a softcore, a traffic generator, DRAM controller, DRAM, and an Integrated Logic Analyzer (ILA), see Figure 3.5. The softcore has a direct connection to the arbiter and the traffic generator. The arbiter connects the traffic generator and the softcore to the DRAM controller. The DRAM controller translates the requests from the arbiter to the interface of the DRAM chip. The ILA captures all signals between the traffic generator and the DRAM controller. This experiment focuses on the 128 bit AXI4 bus between the traffic generator and the DRAM controller, because every thing after this bus is shared by multiple cores in the final system.

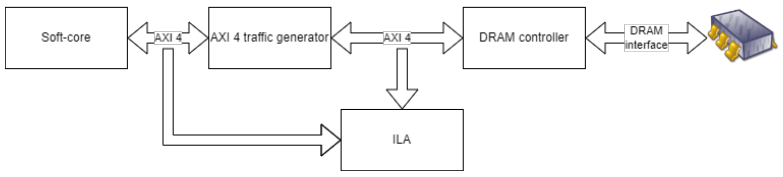


Figure 3.5: Design used to measure all signals between the traffic generator an the DRAM controller.

Methodology

There are several types of AXI4 transfers: single read, burst read, single write, and burst write. Each type transfer has its own latency properties and therefore needs its own measurement. On top of these transfers it is also possible to have a burst transfer that request to access addresses over multiple rows or banks.

To determine the latency of each type of transfer the following experiments are executed:

1. 64 single read request on the same row.
2. 64 single read request with each request on a different row.
3. 16 burst read requests of each 128 transactions each on the same row.
4. 16 burst read requests of each 256 transactions that crosses 2 different rows.
5. 64 single write requests on the same row.
6. 64 single write requests with each request on a different row.
7. 16 burst write requests of each 128 transactions each on the same row.
8. 16 burst write requests of each 256 transactions that crosses 2 different rows.

The hardware used for this experiment is an Arty a7-100t development board with a micron MT41K128M16 DRAM chip [54][5]. It is important to note that the latencies are depending on the DRAM and DRAM controller, therefore these results are only useful for the Arty a7-100t with a micron MT41K128M16 DRAM chip and standard DRAM controller of Xilinx. Also, the experiment is executed on room temperature and on a single copy of the Arty a7-100t.

Results and discussion

During each experiment the ILA is used to measure the behavior of the AXI4 bus. The latencies of each transfer are defined by the time difference between the moment the transfer is accepted and the last transaction of the transfer. For both read and write transactions this means the the start point is the AXI4 handshake of the address channel. However, the end point of the transaction is the handshake on the write response channel for a write transaction and the last read data transaction for the read channel. The results of the measurements are shown in Table 3.1.

The total latency for transfers that consist of more than 1 data transaction should be proportional to the amount of data transactions within the transaction. For example, a burst transfer that consists of 123 data transactions should have a latency of 122 clock cycles more than a transfer with only a single data transaction. Therefore the effective latency is equal to the measured latency minus the amount of data transactions. The maximum effective latency, shown in the last column of Table 3.1, is 57 clock cycles.

Experiment nr.	Min latency	Max latency	Number of transactions per transfer	Max effective latency
1.	24	26	1	25
2.	25	32	1	31
3.	162	173	128	45
4.	286	306	256	50
5.	6	7	1	6
6.	6	8	1	7
7.	139	145	128	17
8.	286	313	256	57

Table 3.1: AXI4 latencies

Conclusion
To determine what the constant latency response of the DRAM should be, an experiment is executed and the latency deviation is measured. To mitigate the timing side-channel the latency should be constant and equal to or bigger than the maximum latency of the DRAM chip. For the Arty a7-100t development board, the maximum effective latency measured is 57 clock cycles. Since there are a limited amount of measurements the chances are high that the latency of the DRAM might be higher than 57 clock cycles at one point in time. Therefore it is safer to take a 15% safety margin and take 65 clock cycles as fixed constant latency of the DRAM. Since the time buffer should be equal or bigger than the fixed latency, it is therefore set to 65 clock cycles as well.

3.2.3. Design

The overview of the design for the FTA is shown in Figure 3.6. The FTA consists of one AXI mux, one time slot tracker, per AXI4 input bus a deterministic delay and one AXI4 channel controller per AXI4 input bus.

Time slot tracker
The counter is responsible for generating an *input_sel* signal that controls which AXI input bus connects to the AXI bus to the DRAM for a certain time slot. The period of the time slot is set with two generics, the allocated time per transfer and a time buffer. The counter will also send information about the amount of time that has passed within the current time slot. This information will be used by the controllers to determine whether the next AXI transfer fits within the time slot or whether it should wait for the next time slot when it is its turn.

AXI mux
The AXI mux takes the state of each AXI4 channel controller and the *input_sel* from the counter to connect the correct AXI input bus to the AXI output bus.

The AXI mux has the following inputs and outputs:

- req_vector: Input AXI request vector.
- resp_vector: Output AXI response vector.
- req: Output AXI request.
- resp: Input AXI response.
- state_vector: State information vector.
- input_sel: Input select.
- clock: Input clock.
- reset: Input reset.

The AXI mux uses the *input_sel* as an index to select an AXI bus from the *req_vector* and *resp_vector* and it selects the corresponding state from the *state_vector*. Depending on the selected state it connects one of the channels of the selected AXI bus to the AXI output bus. The VHDL description can be found in Listing 3.1. The AXI mux is not clocked which means that the input and output AXI signals should be generated and captured by clocked registers. Another potential weakness of this mux is the fact that it does not check whether or not a transfer is finished before connecting the next input bus to the output bus. However, this should not be a problem as long as the time buffer after the time slot is big enough.

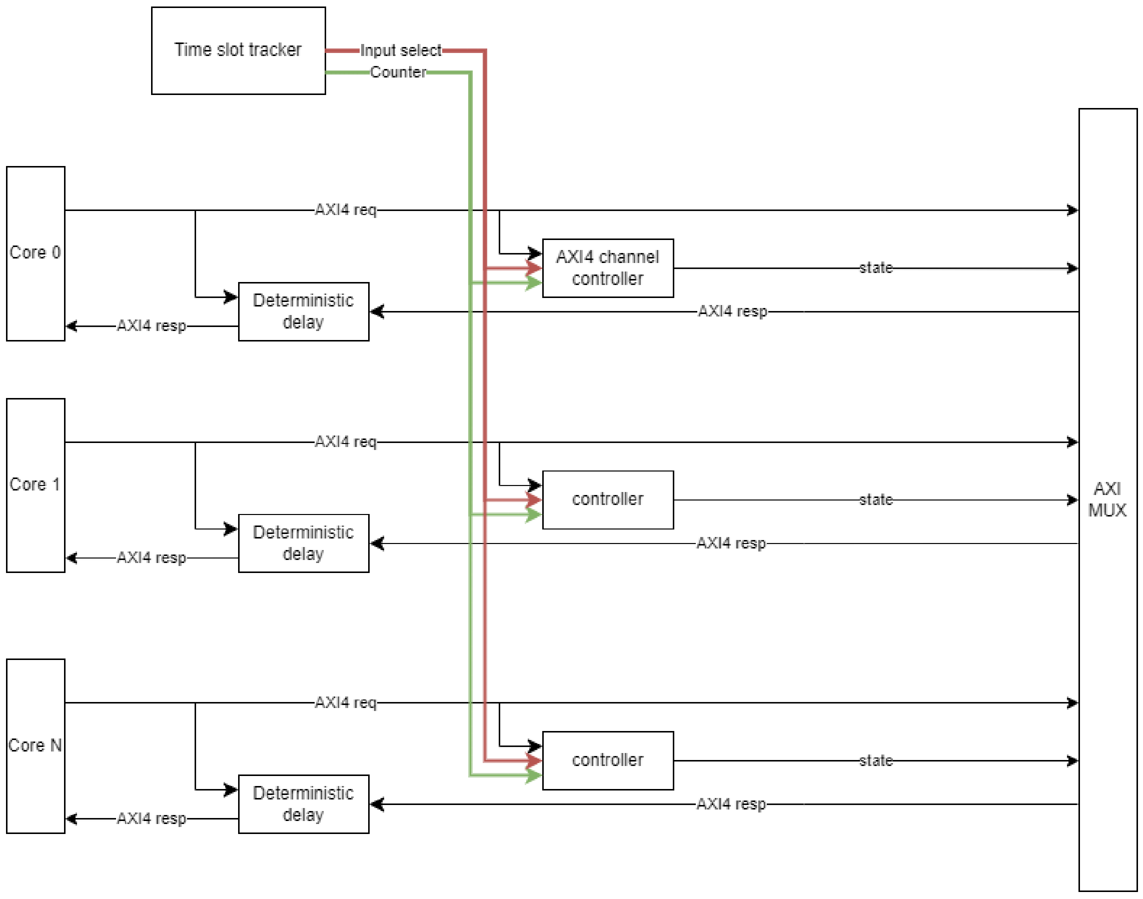


Figure 3.6: Overview of the design for the Fixed Time Arbiter.

Listing 3.1: VHDL code within the AXI mux to connect the AXI output to the correct AXI input.

```vhdl

req.ar <=   req_vector(input_sel).ar when state_vector(input_sel) = rd_accept else
                                              c_axi4_128_address_req_init;

req.aw <=   req_vector(input_sel).aw when state_vector(input_sel) = wr_accept else
                                              c_axi4_128_address_req_init;

req.r <=    req_vector(input_sel).r when state_vector(input_sel) = rd_resp else
                                              c_axi4_any_resp_init;

req.w <=    req_vector(input_sel).w when state_vector(input_sel) = wr_data else
                                              c_axi4_128_wdata_req_init;

req.b <=    req_vector(input_sel).b when state_vector(input_sel) = wr_resp or state_vector(input_sel) = wr_data else
                                              c_axi4_any_resp_init;

select_resp_vector: for inputs in 0 to g_number_of_ports-1 generate
        resp_vector(inputs).ar <=   resp.ar when input_sel = inputs and state_vector(inputs) = rd_accept else


        resp_vector(inputs).r <=    resp.r when input_sel = inputs and state_vector(inputs) = rd_resp else


        resp_vector(inputs).aw <=   resp.aw when input_sel = inputs and state_vector(inputs) = wr_accept else


        resp_vector(inputs).w <=    resp.w when input_sel = inputs and state_vector(inputs) = wr_data else


        resp_vector(inputs).b <=    resp.b when input_sel = inputs and (state_vector(inputs) = wr_resp  or state_vector(inputs) = wr_data) else


end generate select_resp_vector;
```

AXI4 channel controller

Each input AXI bus has its own AXI4 channel controller. The goal of the AXI4 channel controller is to guarantee the correct behavior of the AXI 4 protocol for each individual input AXI bus and to help the AXI mux to connect the correct input to the output AXI bus. Each controller has its own unique id number *g_port_nr* to identify the instance of each controller.

The FSM of each AXI4 channel controller has 7 states, see Figure 3.7. The state transition conditions are described below:

- *idle*: It waits for the *input_sel* to select the corresponding AXI bus. If the *input_sel* is equal to the *g_port_nr* it checks if there is any read or write request on the input AXI bus. If there are any requests, it compares the length of the transfer to the time left in the time slot. If the thransfer fits in the remaining time of the time slot, the state changes to *rd_accept* or *wr_accept*.
- *wr_accept*: In this state the controller waits for the write address transaction to happen. If *aw.ready* and *aw.valid* are both '1' the state goes to *wr_data* otherwise it stays in *wr_accept*.
- *wr_data*: In this state the controller waits for all data to be send. If *w.last*, *w.valid* and *w.ready* are all '1' the state transitions to *wr_resp*, otherwise it stays in *wr_data*.

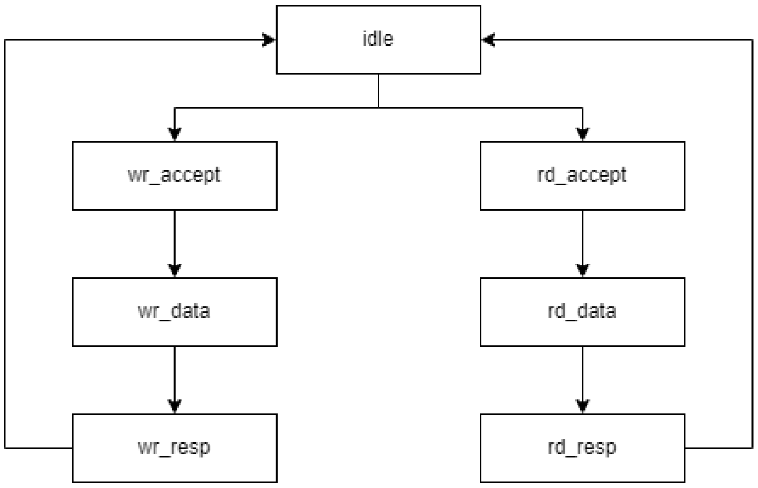


Figure 3.7: The seven states of the AXI4 channel controller

- *wr_resp*: In this state the controller waits for the response from the DRAM. When *b.valid* and *b.ready* are both '1' the state becomes *idle* again.
- *rd_accept*: In this state the controller waits for the read address transaction to happen. If *ar.ready* and *ar.valid* are both '1' the state goes to *rd_data* otherwise it stays in *rd_accept*.
- *rd_resp*: In this state the controller waits for all data to be send. If *r.last*, *r.valid* and *r.ready* are all '1' the state transitions to *idle*, otherwise it stays in *rd_resp*.

Deterministic delay
The goal of the deterministic delay is to delay the response from the r and b channels in such way that the latency is predetermined and does not contain any information, as described in Section 3.2.1. The deterministic delay consists of two parts: FIFOs and FIFO controllers. The inputs of the FIFOs are the b and r channel responses form the AXI mux. Together with the b and r ready signals from the input AXI bus, the FIFO controllers determine when the data stored in a FIFO is allowed to be send to the softcore. Each input AXI bus has two FIFO controllers, one for the r channel and one for the b channel. The controller consist of a counter and some control logic.

When the FIFO controller is used for a r channel it enables the counter when the ready and valid signals of the ar channel are both '1' and disables the counter when the ready, valid and last signals of the r channel are all '1'. When the FIFO controller is used for a b channel it enables the counter when the ready and valid signals of the aw channel are both '1' and disables the counter when both ready and valid of the b channel are '1'. The last signal is fixed to high, since the b channel does not have a last signal.

The control logic compares a generic *delay* to the value of the counter. If the counter is bigger than the delay the *fifo_en* is '1' otherwise the *fifo_en* is '0'. The *fifo_en* signal goes to an AND gate together with the r or b ready signal. The output of the AND gate goes into the FIFO enable port. This guarantees the compliance with the AXI protocol and a fixed latency for every transfer.

3.2.4. Verification

To verify the design Cocotb is used to simulate the fixed time arbiter. Cocotb is a Python based simulator for VHDL. The test cases created in Cocotb are:

1. Single transfer with burst length 1.
2. Multiple transfers with burst length 1.
3. Multiple transfers with burst length 1 when the slave is delayed.
4. Multiple transfers with burst length 128.
5. Multiple transfers with burst length 256.

This set of test are chosen because a single transfer is to check if the basic principle works and if the delay is as expected. Multiple single transfers is to make sure that when there is not enough time left in the current timeslot the next transfer will be executed in the next timeslot. The test with the delayed slave is used to simulate a memory access with a higher latency from the DRAM, the latency to the master should be the same as in the previous test. The 128 burst transfer is used to show that if there is not enough time left in the timeslot the transfer will be executed in the next timeslot. The last test is to make sure that a timeslot is big enough to execute a burst transfer with the maximum length of 256 transaction is possible.

The results are that each of the transfers is only executed when the fixed time arbiter is in the window of the selected time slot. Also when measuring the latency of each transfer is equal to the preset time buffer plus the transfer length after the corresponding timeslot has started. This is just as described in Section 3.2.1 and therefore the design is correct in simulation.

3.2.5. Conclusion

The fixed time arbiter gives a fixed predetermined time slot to each of the AXI4 input buses. To mitigate the latency deviations from the DRAM it uses FIFOs to create a predetermined latency for all transfers depending on a preset fixed latency and the burst length. This will make sure that the latency has no hidden information about the other softcores. Due to the predetermined time slots, the bandwidth side channel is also mitigated.

The behavior is tested with a simulation and the results are as expected.

3.3. Rowhammer bitflip characteristics

This experiment is required to determine which rows are suitable to be guard rows and which rows should be refreshed if a individual row is hammered. These properties are required to created a solution to protect the DRAM chip against rowhammering. A physical map of the DRAM is created that contains information about the electromagnetic interaction between the rows on the DRAM. The result of this experiment are specific for the individual chip on the Arty A7 100t that is used and should be done for any chip before allocating address space of each softcore.

3.3.1. Methodology

For all rows within the DRAM the following steps are executed:

1. Initialize the DRAM so all addresses contain the data 0x55.
2. Switch off the refreshes of the DRAM.
3. Hammer a row by writing data at the maximum bandwidth of the AXI4 bus for 1 second.
4. Switch on the refreshes of the DRAM.
5. Validate all data in the DRAM and return for each row the amount of bytes that have changed.
6. Repeat step 2 to 5 for each row in the DRAM.

3.3.2. Results

The hardware used is the same as in Section 3.3. However, it uses another application. The application hammers a single row at a time for a second. Then, it checks the full DRAM on flips and returns the amount of flips per row and then it hammers the next row until the full DRAM is hammered. After the test has run, the results are captured in a text file showing for each of the hammered rows, the rows it caused bitflips to and the amount of bitflips per row. After analyzing the results the following categories are created:

- No rows: There are no rows with bitflips after a particular row has been hammered.
- One row: There is one row with bitflips after a particular row has been hammered.
- Two rows: There are two rows with bitflips after a particular row has been hammered.
- More than two rows: More than two rows have bitflips after a particular row has been hammered.
- Single adjacent row: Only one of the adjacent rows has bitflips after a particular row has been hammered.
- Both adjacent rows: Both adjacent rows have bitflips after a particular row has been hammered.
- Non adjacent rows: Number of non adjacent with bitflips after a particular row has been hammered.

Each hammered row is analyzed and put into one or multiple categories depending on how many and in which rows it caused bitflips.

Table 3.2 shows the result of the experiment. Hammering any row caused bitflips in at least one other row. Most of the rowhammer attacks caused bitflips in at least both its adjacent rows. The bitflips that are found in non adjacent rows are only limited to the rows shown in Table 3.3. 99% of the hammered rows cause bitflips in both of their neighbor rows. When looked at the nonadjacent rows, 93% of the rows cause bitflips in nonadjacent rows. However, when looked at the where the nonadjacent flips took place, there are only 17 different rows that are sensitive for rowhammering in a nonadjacent row, see Table 3.3. Therefore, it is better to keep those 17 rows unused.

Rows with bitflips	Nr. of hammered rows that caused bitflips
No rows	0
One row	35
Two rows	9360
More than two rows	121677
Single adjacent row	512
Both adjacent rows	130560
Non adjacent rows	121990

Table 3.2: Every row in the DRAM is hammered and it is measured in which rows it caused bitflips. The rows with bitflips describes where the bitflips are after a single row has been hammered. The Nr. of hammered rows are the amount of rows that when hammered they caused bitflips as described in the column rows with bitflips.

Row	Bank	Nr. of unexpected flips
0x6C5B	1	121987
0x129A1	4	42005
0x7700	1	34010
0x1BF64	6	15157
0x4DEC	1	3886
0x3691	0	3205
0x2B8E	0	1660
0x49E5	1	653
0x111E8	4	275
0xCC96	3	309
0xC9D3	3	166
0xE3A9	3	66
0x66F7	1	87
0x76D7	1	52
0x1C089	7	29
0x8D4A	2	13
0xB535	2	2

Table 3.3: List of rows that got bitflips by hammering non of its adjacent rows.

3.3.3. Conclusion

The effects of rowhammering on a specific DRAM chip has been tested. If any row is hammered, there is at least one row that has bitflips in it. In 99% of the cases rowhammering causes bitflips in both of their adjacent rows. In 17 rows there were bitflips due to hammering in a nonadjacent row. With those two observations it is concluded that those 17 rows should be unused and that guard rows between two security domains are sufficient to prevent bitflips in one domain due to rowhammering in another domain. However, if it is required that there are no bitflips within a single security domain, refreshing of the adjacent rows should be implemented.

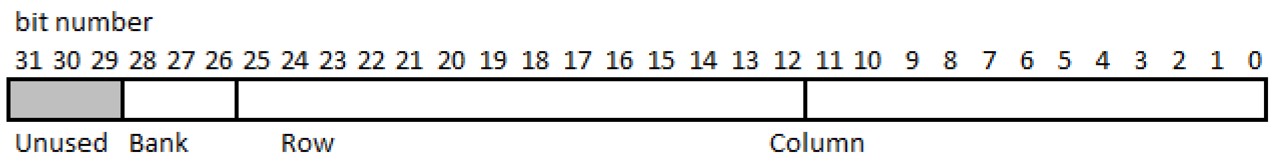


Figure 3.8: Address mapping on a DRAM

3.4. Row refresher

The goal of the rowhammer refresher is to refresh the adjacent rows before there is a chance on a bitflips in the adjacent rows. The implementation consists of block RAM and a controller. Two different implementations of the controller are created. One tracks all rows on all banks and refreshes only two the adjacent rows on one bank at the time. The second implementation only tracks the row bits of the address and does not check on which bank the row is. When the access threshold is exceeded it refreshes all corresponding adjacent rows on every single bank.

3.4.1. Addressing

In order to refresh the neighbors of an attacked row the amount of accesses needs to be tracked and therefore the mapping from AXI 4 addressing to the actual DRAM location is needed. The DRAM controller is responsible for the addressing remapping. In case of the Arty A7 development kit, 11 bits are used to select a column, 14 bits are used to select a row and 3 bits are used to select a bank, see Figure 3.8. The column bits are the LSB, the bank bits are the MSB and the row bits are in between.

3.4.2. Tracking accesses

The most obvious method to track the accesses of each row is to have a memory array that contains a counter for each row on each bank. The address of each counter is determined by the bank and row address bits. In case of the used DRAM this would mean that there are 2^{17} addresses each containing 2 bytes. This would add up to 256 KiB. The Arty A7-100T has 593 KiB in block RAM so it would fit on the FPGA.

To reduce the amount of block RAM required it is also possible to only use the row address bits to track the row accesses. This would reduce the amount of memory required to 32 KiB. The downside of tracking the rows only is that it is not known on which bank the rows are and therefore all rows with the same row address need to be refreshed on every bank. In case of the used DRAM this would mean 8 extra refreshes instead of 1.

Both methods are implemented and with a generic, so it is possible to switch between the two depending on the end user requirements.

Both implementations use block RAM and a tracker to protect against rowhammering, see Figure 3.9. The tracker works similarly with three pipeline stages. Stage one checks the address of the read request and send a read request to the block RAM. Stage two checks if the current row access count received form the block RAM exceeds the predetermined threshold and writes the new value of the count back to the block RAM. The last stage generates AXI read request if the row needs to be refreshed, see Figure 3.10. The difference between tracking all rows on all banks compared to only tracking the row bits is the amount of input address bits (17 vs. 14), and the amount of AXI read requests (2 vs. 16).

For the block RAM a two-port block ram is used with a read priority. Port a is used to read the access count and port b is used to write the updated access count back. The block RAM controller from Xilinx handles the read and write operations for a correct functionality. The advantage of having two ports is that both can write and read at the same time so it is easier to implement.

3.4.3. Validation

For the validation a testbench for the controller is created. The signals of the Block RAM are simulated in the testbench, the same holds for the AXI4 signals. There are two test cases created, test case one test if the pipeline is correct, test case two tests if the access counter is reset to zero when the threshold is reached.

The expected result of the pipeline is that after an AXI4 request (write or read) the controller send a read request to the Block RAM on an address equal to the row number. In the next cycle when the data

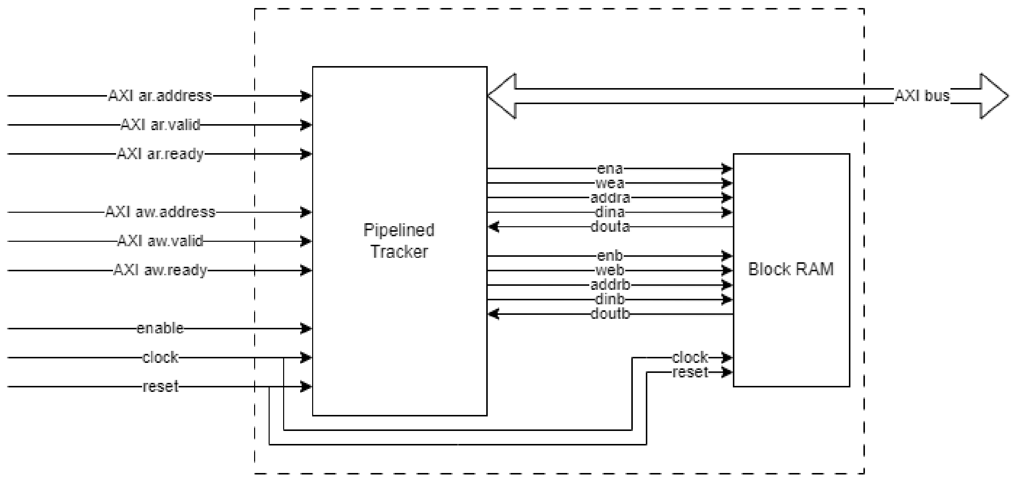


Figure 3.9: Overview of the implementation for the rowhammer refresher. It consists of 2 parts: a pipelined tracker and Block RAM.

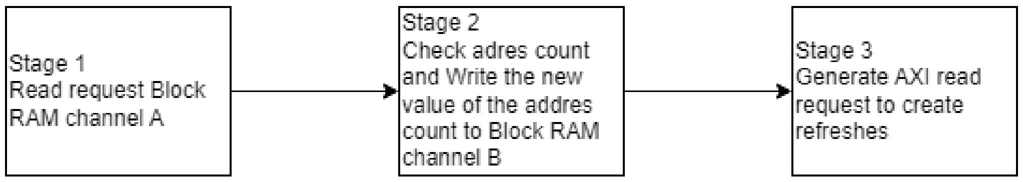

Stage 1
Read request Block RAM channel A
Stage 2
Check adres count and Write the new value of the addres count to Block RAM channel B
Stage 3
Generate AXI read request to create refreshes

Figure 3.10: Overview of the pipeline within the rowhammer refresher. There are 3 stages. Stage 1 sends a read request to the Block RAM. Stage 2 receives the data from the block RAM, checks if the threshold has been exceeded and then it increases the counter or set the counter on zero, before writing it back to the Block RAM. The last stage generates the corresponding AXI read requests in order to refresh the targeted rows in the DRAM.

from the Block RAM is available to the controller it increases the data by one and write the data back to the Block RAM.

The expected results of the reset of the counter is that the data written to the Block RAM is equal to zero and that two or sixteen AXI4 read requests are created for both of the adjacent rows of the AXI4 request from the softcore, depending on which one of the two implementations is used.

When running the simulation for both test cases the controller behaves as expected.

3.4.4. Conclusion

The row refresher is created and the end user can use a generic to choose between two different implementations: track all rows on all banks, or track only the row bits of the addressing. The trade-off between the two options is on area and performance. Tracking all rows on all banks uses 8 times more Block RAM than tracking the row bits only. However, when the counter in the Block RAM reaches the threshold it will also have eight 8 more refreshes, two refreshes on each bank.

In simulation the design behaves as expected. Increasing the counter of each row when accessed and refreshing the correct adjacent rows. When the threshold of the counter is reached two or sixteen refreshes are send to the DRAM, depending on which of the two implementations are used.

3.5. Address mapper

The goal of the address mapping is to allocate a fixed address space for each individual softcore implemented in the hardware. This will prevent the connected softcore from accessing unauthorized addresses directly. If the softcore tries to access an address that is not within the range of the allocated address space the address mapping will respond with a decode error.

The address mapping checks if the requested address is within the allocated size of the memory. It also remaps the incoming address to its allocated physical address in the DRAM by adding a preset offset to the incoming address. If an incoming address is outside of the allocated size the address mapper makes sure that no data is send of received from the DRAM. Instead it respond to the core with a decode error.

3.5.1. Implementation

Figure 3.1 shows how the address mapping should fit into the design. It has an AXI4 slave connected to the AXI4 master of the softcore and an AXI4 master connecting to the slave of an arbiter that is also connected to multiple other softcores. The other connections to the arbiter are not drawn in the figure.

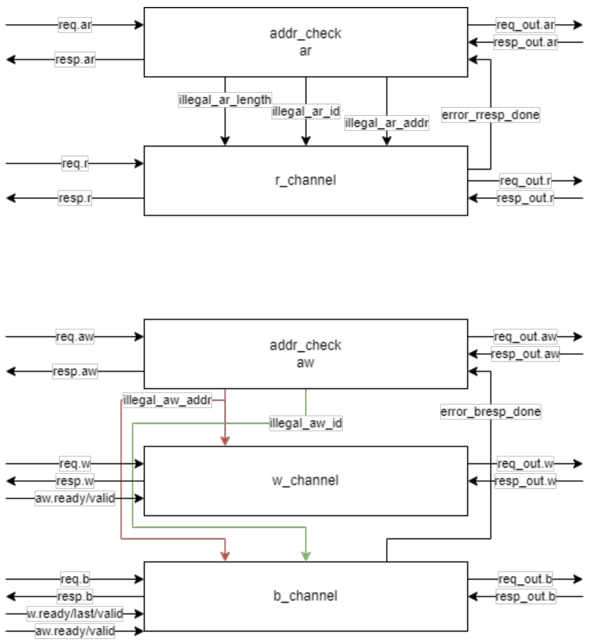


Figure 3.11: Overview of the address mapping with a separate control logic for each of the 5 AXI4 channels.

The overview of the the address mapping is shown in Figure 3.11. Each AXI4 channel has its own controller that makes sure that the AXI4 protocol is correctly followed. The allocated address offset and address size is set through two generics offset and size. The *addr_check* checks if the incoming transfer is within the allocated size and is used for both the aw channel and ar channel. It also adds the offset to the incoming address from the softcore before sending the remapped address to the DRAM controller. If an address is out of range of the allocated size, the *addr_check* waits for an *error_resp_done* signal before it can accept the next transaction.

The r channel controller send data from the DRAM controller to the softcore as long as the address is legal. If the address is illegal it sends a decode error to the softcore. Any open transactions before the illegal address are finished first before the error message.

The w channel controller only sends data to the DRAM controller if there is a transfer in progress with a legal address. The w channel controller gets the information on whether the address is legal or not from the *addr_check* of the aw channel. If there is an illegal address the w channel controller makes sure all pending transfers are finished and no data from the illegal transfer goes through to the arbiter.

The b channel controller sends the data from the DRAM controller to the softcore if the address is legal, otherwise it sends a decode error to the softcore as specified in the AXI4 documentation. After an illegal transfer is handled it sends an *error_resp_done* signal to the *addr_check* of the aw channel so it knows the illegal address has been processed.

The address mapper only supports increased burst transfers and single transfers, because they are the most common and cover the other transfers when checking if the requested transfer is within the allocated range. The other types of burst transfers are fixed and wrap. Fixed burst transfers only reads or writes data from a single address. Wrap transfers wraps around to a set address after it has accessed the highest address in its transfer. Both the fixed and the wrap burst transfer have a lower maximum address than an increased burst transfer. Therefore, if they are treated as a increased burst transfer they will be rejected if they are out of the allocated address space. The only problem with relying on the increased burst transfer check is that some of the fixed and wrap transfers are rejected even tho their actual transfer is within the allocated address space.

3.5.2. Validation

To validate the design several test cases are created and simulated. The address mapper is set to have an offset of 0x1000 and an size of 0x1000.

1. Read and write multiple single transfers on address 0x0 to check if the offset is applied correctly.
2. Read and write multiple single transfers on address 0x0 with a throttled ready to check if the AXI4 protocol is compliant.
3. Read and write increased 256 burst transfer with a constant ready on address 0x0 to check if the burst transfers are AXI4 compliant and the offset is correctly.
4. Read and write increased 256 burst transfer with a throttled ready on address 0x0 to check if the burst transfers are AXI4 compliant and the offset is correctly.
5. Read and write single transfer on address 0x5 to check if it is compliant with asynchronous addresses.
6. Read and write single transfer on the last address 0xFF0 and the first address that is out of the allocated range 0x1000 with a steady ready, to check if the preset size is handled correctly.
7. Read and write single transfer on address 0xFFFFF to check that it gives the correct error response when the address is out of the allocated range.
8. Read and write 256 burst transfer transfer on address 0xF00 to check if a burst transfer that is partially inside the allocated range and partially outside the allocated range is rejected and gives the correct error response.

The address mapper behaves as expected when the above test cases are simulated. It checks if the incoming address is within its address range and adds the preset offset to the address correctly. The test prove that it can handle increased burst transfers and that the last address is handled correctly. Other type burst transfers are not tested, since this is not fully supported in the address mapper.

3.5.3. Conclusion

The address mapping uses a size generic to check if the incoming address is within the allocated address space and it uses an offset generic to map the incoming address to the physical address on the DRAM. By making the data channels wait for the addresses, it guarantees that no out of bound data transfers can leak in to the wrong address domain. This design supports only increased burst transfers and single transfers. Other type of burst transfers can be falsely rejected by the address mapper.

The cost of this are 4 extra clock cycles before the response reaches the softcore. This hardware block is very flexible and can therefore be used to make sure the rows that are sensitive for rowhammering, see Section 3.3, can be left unused.

3.6. Conclusion

The latency side-channel has been mitigated. There are still latency deviations, but the deviations can not be used to transfer information from one core to another, since the latencies depend on how long a transfer needs to wait on its timeslot, which is predetermined and therefore contains zero information. There is a downside on the implementation, namely if the fixed latency is not set to the maximum latency possible, the implementation has no way to identify this. Therefore, before using this solution the experiments in Section 3.2.2 should be done to determine the maximum latency and set the parameters accordingly.

The bitflip characteristics of the DRAM chip is created for a specific DRAM chip. The bitflips are mostly in the adjacent rows of the row that is being hammered. There are a few non adjacent rows that also had bitflips due to rowhammering, see Table 3.3. The address mapper can be used to set the rows that have bitflips when a non-adjacent row is hammered on non active.

There are two implementations described to mitigate rowhammering, one tracks all rows on all banks, the other one only tracks the row address bits of the AXI4 address and refreshes all corresponding adjacent rows on all banks. The second option causes some performance loss, but it uses significantly less resources on the FPGA.

The address mapper makes sure that no softcores can access addresses outside of its domain. When a softcore tries to access an unauthorized address the address mapper responds with a decode error. It is up to the final user of the address mapper to set the offset and size correctly for its design. The address mapper can be used to add guard rows against rowhammering. It can also be used to keep the rows in Table 3.3 unused.

4 Results

This chapter describes the performance results and the amount of resources used for the complete design as described in Section 3.1 and compares it to a design that uses a standard arbiter. Section 4.1 describes the methodology used to measure the performance and the amount of resources used on the FPGA. Section 4.2 shows the amount of resources each of the components of the Memory Domain Protector uses on the FPGA. The measured performance is discussed in Section 4.3 and this chapter is finished with a conclusion in Section 4.4.

4.1. Methodology

The interfaces between the softcore and the secure arbiter and the secure arbiter and the DRAM controller are using the AXI4 protocol. Since the performance of the AXI4 protocol depends on the burst length of the transfer, the bandwidth is measured for all possible burst lengths. Figure 4.1 shows the algorithm used to measure the duration for each of the transfer lengths. The traffic generator is used to generate 256 request for all possible transfer lengths. Per transfer length the average bandwidth is calculated.

This experiment is repeated four times: all burst writes have the same start address, all burst reads have the same start address, all burst writes have alternating start addresses of 0x0 and 0x800 (row 0 and row 1) and the last experiment is all burst reads have alternating start addresses of 0x0 and 0x800 (row 0 and row 1).

The hardware used for the benchmark is a standard round robin based arbiter that has 2 AXI4 input buses, with only a single input bus is generating requests. The results of the benchmark are compared with the results of a similar design with the memory domain protector that uses the fixed time arbiter, address mapping and rowhammer refreshes as described in Section 3.1.

To determine the resources used by the secured shared DRAM solution the Vivado synthesis report is used. The secured shared DRAM solution is compared to a standard round robin arbiter with the same amount of input AXI4 buses.

4.2. Synthesis

To determine the resources used on the FPGA, Vivado's hardware reports are used. The hardware reports can show exactly how much Slice Look-Up-Tables (LUTs), Slice registers, F7 Muxes and Block RAM (BRAM). are used by each individual ip block. The results are shown in Table 4.1.

4.3. Performance

Figure 4.2 shows the bandwidth results of the standard round robin arbiter. All 4 experiments show a drop in bandwidth when the burst length is bigger than 128. The reason for this drop is that the DRAM on the Arty A7-100t has 1024 columns per row and each address contains two bytes. Therefore each row can contain 2048 bytes. This is equal to an AXI4 transfer with a burst length of 128 and 16 bytes per transaction. Therefore burst lengths bigger than 128 are accessing at least two rows on the DRAM and

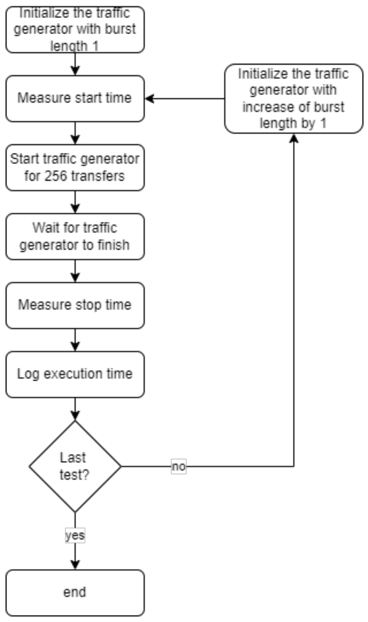

Initialize the traffic generator with burst length 1
Measure start time
Initialize the traffic generator with increase of burst length by 1
Start traffic generator for 256 transfers
Wait for traffic generator to finish
Measure stop time
Log execution time
Last test?
no
yes
end

Figure 4.1: Flowchart of methodology of the bandwidth measurements.

	Slice LUTs	Slice Regs	F7 Muxes	Slices	BRAM
Available resources	*63400*	*126800*	*31700*	*15850*	*135 (4860 Kbits)*
Round Robin arbiter	*105*	*49*	*0*	*52*	*0*
Rowhammer protection	78	88	0	44	6.5 (234 Kbits)
fixed time arbiter	1200	569	4	551	6 (216 Kbits)
address map	586	1195	0	468	0
Total	**1864**	**1852**	**4**	**1063**	**12.5 (450 Kbits)**

Table 4.1: FPGA resources after synthesis. The available resources are the available resources on the Arty A7-100t. The Round Robin arbiter is the benchmark for the resources used. The total is the total of the rowhammer protecion, fixed time arbiter and the address map.

the bandwidth drop is due to the load time of the row into the rowbuffer as explained in Section 2.3.3. The maximum bandwidth is 1184 Mb/s for read transfers with burst length 256.

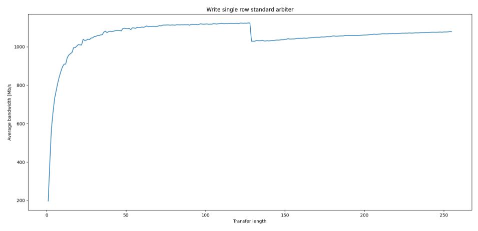
Figure 4.3 depicts the performance of the secure shared DRAM. The first thing that stands out is the sawtooth pattern. This is due to the use of fixed time slots in the fixed time arbiter. The length of each time slot is chosen in such way that a 256 burst transfer fits exactly in a single time slot. However, burst transfers with smaller burst lengths can fit multiple times within a time slot and therefore it cause jumps in the bandwidth when the burst length exceeds the threshold that fits within a single time slot.

When comparing the bandwidth with the standard round robin arbiter the performance is roughly 4 times worse. This is due to the fact that the secured arbiter uses fixed time slots for each AXI4 input bus for scheduling the DRAM accesses and therefore one core cannot use the time slot of another core even tho the other time slot is idle. On top of this the secured arbiter also has an extra time slot inside for the rowhammer protection. The last reason why the performance is 4 times worse than the standard arbiter is due to the fact that there is a fixed transfer time equal to the worst case scenario for each DRAM access and no new transfer can start until the old transfer is finished.

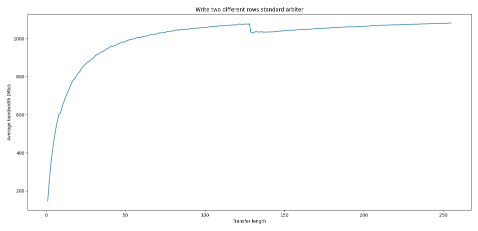
However, the performance gap will be smaller when multiple cores are accessing the DRAM at the same time, since the secured arbiter has a constant bandwidth no matter what the other cores do and the standard arbiter divides the maximum bandwidth over each core.

4.4. Conclusion

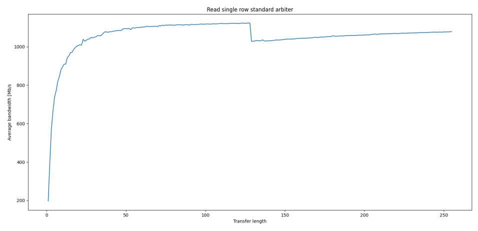
In this chapter the resources and performance of the memory domain protector are compared to a standard round robin arbiter. The amount of resources it takes to secure the DRAM is a lot and in most



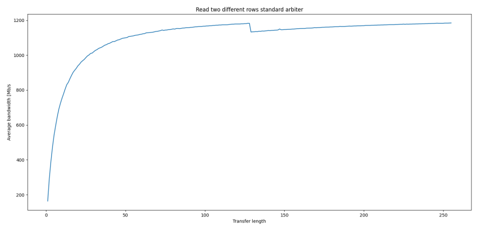
(a) Increasing write transfers with the same starting address for each transfer.



(b) Increasing write transfers with alternately address 0x0 and address 0x800 as starting address.



(c) Increasing read transfers with the same starting address for each transfer.

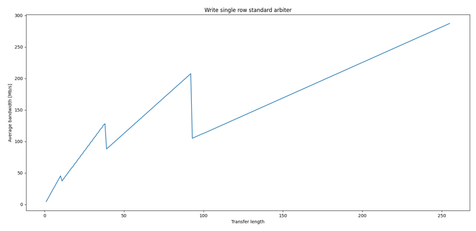


(d) Increasing read transfers with alternately address 0x0 and address 0x800 as starting address.

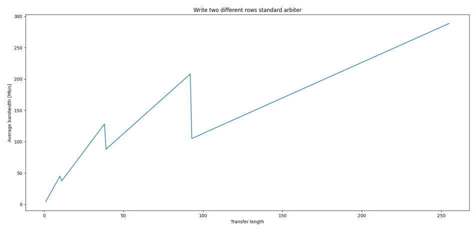
Figure 4.2: Bandwidth for the standard round robin arbiter. The x-axis shows the burst transfer length and no the y-axis shows the bandwidth.

user cases not worth the tradeoff. However, for high security products it is a must and therefore it is justified. The maximum bandwidth is also significantly worse compared to the round robin arbiter. But again it is application dependent whether or not it is sufficient.

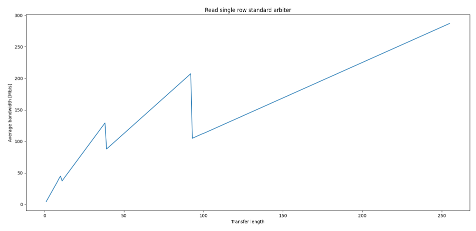
All in all if it is required for the product to have shared DRAM between multiple softcores the extra area and the performance loss is the cost that needs to be paid.



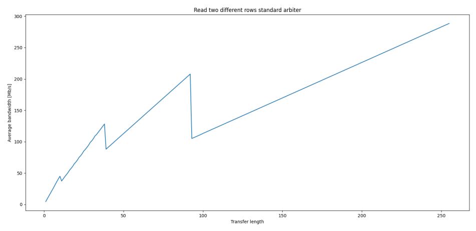
(a) Increasing write transfers with the same starting address for each transfer.



(b) Increasing write transfers with alternately address 0x0 and address 0x800 as starting address.



(c) Increasing read transfers with the same starting address for each transfer.



(d) Increasing read transfers with alternately address 0x0 and address 0x800 as starting address.

Figure 4.3: Bandwidth for the secure shared DRAM. The x-axis shows the burst transfer length and no the y-axis shows the bandwidth.

5 Conclusion

5.1. Summary

Security products in computer networks depend on domain separation. In order to create the domain separation multiple softcores are used within a product. The softcores scale well with the amount of resources available on an FPGA, but due to limited amount of IO pins a single DRAM is shared between the softcores. This thesis presented the memory domain separator in order to share a single DRAM without having any data leakage from one domain to another domain.

In Chapter 2 the necessary background knowledge about the memory hierarchy, DRAM and the AXI4 bus is explained. It also gave an overview of the currently known security vulnerabilities. The three main vulnerabilities for sharing DRAM are: direct access to another domain, timing deviations and rowhammering. The mitigation for direct access to another domain is trivial.

There exists several solutions that mitigate rowhammering. Those solutions rely on keeping track of the memory accesses and have targeted refreshes of the adjacent rows that are accessed more than a certain threshold. The other solution is probabilistically refreshing one of the adjacent rows every time the memory is accessed. Unfortunately, rowhammer can also cause bitflips in non-adjacent rows due to production faults or remapping by the producer and therefore more research was required to give an overview of which rows get influenced by other non adjacent rows.

There has not been a lot of research on latency deviations of DRAM accesses. There is only one solution named in the literature about mitigation of the latency deviations of DRAM, namely the implementation of a fixed latency. However, this was not implemented by anyone since the performance penalty is too high for them.

Chapter 3 gives an overview of the memory domain protector. In Section 3.2 it is described how the latency deviation is mitigated. There are still latency deviations, but the deviations do not contain any information about the other softcores, or can it be influenced by other softcores. Therefore, those latency deviations do not contain any information that could be used to setup a covert-channel. There is a downside on the implementation, namely if the fixed latency is not set to the maximum latency possible, the implementation has no way to identify this. Therefore before using this solution the experiments in Section 3.2.2 should be done to determine the maximum latency and set the parameters accordingly.

Section 3.3 explains the bitflip characteristics of the DRAM chip that is being used when it is under a rowhammer attack. The bitflips are mostly in the adjacent rows of the row that is being hammered. There are a few non adjacent rows that also had bitflips due to rowhammering, see Table 3.3.

Section 3.4 explains how the row refresher works to mitigate rowhammering. There are two implementations described, one tracks all rows on all banks, the other one only tracks the row address bits of the AXI4 address and refreshes all corresponding adjacent rows on all banks. The second option causes some performance loss, but it uses significantly less resources on the FPGA.

The address mapper is described in Section 3.5. It makes sure that no softcores can access addresses outside of its domain. When a softcore tries to access an unauthorized address the address mapper respond with a decode error. It is up to the final user of the address mapper to set the offset and size correctly for its design. The address mapper can be used to add guard rows against rowhammering. It can also be used to keep the rows in Table 3.3 unused.

In Chapter 4, the resources and performance of the memory domain protector are compared to a standard round robin arbiter. The amount of resources it takes to secure the DRAM is a lot and in most user cases not worth the trade-off. However, for high security products it is a must and therefore it is justified. The maximum bandwidth is also significantly worse compared to the round robin arbiter. Whether or not there is sufficient performance left depends on the application.

All in all if it is required for the product to have shared DRAM between multiple softcores the extra area and the performance loss is the cost that needs to be paid.

5.2. Main contributions

In this section the research questions that were introduced in Section 1.1 will be answered, after which the main contribution are listed.

The main research question was as follows:

- How can a single DRAM chip be shared between multiple soft-cores running in different security domains without leaking information from one soft-core to another soft-core?

The answer to this question is found in Chapter 2 and Chapter 3. To share a single DRAM between multiple soft-cores an arbiter is needed to connect several AXI4 masters to a single AXI4 slave. However, using a standard round robin arbiter will introduce security vulnerabilities in the form of bandwidth and latency deviations that can be exploited to create covert-channels. Also the DRAM itself has timing deviation that can be used to create a covert channel. Therefore to prevent any form of data leakage the arbiter should make sure the DRAM access has a constant latency and each soft-core gets its own allocated time slot so so the bandwidth is also fixed.

On top of that the address accesses need to be tracked in order to add extra refreshes against rowhammering.

- How to prevent direct unauthorized data access on a shared DRAM?

The answer is to add an address mapper that also checks whether or not the requested address is within the allocated address space of the soft-core. If the requested address is out of the allocated address space the address mapper returns a decode error.

- What side-channels are introduced that can be exploited through software when sharing a DRAM chip between multiple soft-cores?

There are two major side-channels found: latency deviations introduced by the rowbuffer and rowhammering.

- How can the DRAM side-channels be mitigated?

To mitigate the latency deviation side-channel it is required to add a deterministic delay after the response of the DRAM controller. This will make sure that the latency will not contain any information and therefore cannot be used to leak data from one soft-core to another.

To mitigate rowhammering extra refreshes need to be added to prevent bitflips in the adjacent rows of the frequently accessed row. To reach this extra hardware needs to be added to track how often each row is accessed and when a row is accessed more than the preset threshold it should refresh the adjacent rows. It is also possible that there are bitflips in nonadjacent rows due to faults introduced by the making of the chip. Often there are only a limited amount of nonadjacent rows that could have bitflips due to rowhammering and therefore it is possible to not use those rows without losing tons of memory. The address mapper could be used for this purpose.

- What are the consequences of the implemented side-channel countermeasures on the performance and the amount of resources required?

The consequences of the implemented side-channel countermeasures are significant. The maximum bandwidth per soft-core soft-core is around 280 Mb/s per when two soft-cores are used on the arty a7-100t development board. The benchmark had a maximum bandwidth of 1184 Mb/s. However, it is important to note that in both cases only single soft-core is accessing the DRAM. When multiple soft-cores are accessing the DRAM, the maximum bandwidth stays the same when the counter measures are used, on the other hand the maximum bandwidth of the benchmark design will be divided equally between the soft-cores. This means that when 2 softcores are used and both require the maximum bandwidth possible the performance are a lot closer to each other: 280 Mb/s for the design with the countermeasures and 592 Mb/s for the design without the countermeasures.

The amount of extra resources required for the implementation of the countermeasures is significant when comparing it to a standard round robin arbiter. However, when looking at the total available resources on the arty a7-100t it only uses 10% of its block RAM and only a few percent of all other resources.

The major contribution of this work consists of the following:

- The design and implementation of the fixed time arbiter.
- The research on the performance loss of having a fixed latency and bandwidth from the DRAM.
- Designing and implementing hardware to mitigate rowhammering based on only the row address bits.
- Designing and implementing a test to determine the influence of rowhammering on a specific DRAM chip.

5.3. Future work

The development of the memory domain protector has proven to secure the shared DRAM against rowbuffer latency attacks and rowhammering. However, there is room for improvement. The most interesting improvements are as follows:

- Currently there is no fail-safe mechanism for the fixed time arbiter. If an event happens that the latency from the DRAM is bigger than the measured maximum latency then there is no way to detect this event and correct for it. Therefore, a useful feature would be to keep track of the latency from the DRAM and adapt the deterministic delay of the fixed time arbiter to the new maximum delay. This will also help to initialize the fixed time arbiter to delay not more than the maximum delay possible.
- Currently the addresses needs to be allocated manually withing the VHDL code of the memory mapper for each individual soft-core. This requires a lot of understanding of the code and makes debugging of the address allocations hard. In the future it would be a good idea to create a file that contains all address allocations that can be an input of the memory domain protector to automatically generate the correct memory map for all soft-cores.
- The design of the address mapper currently supports only burst transfers that uses increased burst, meaning that all data transactions within the transfer have incremental addresses. The wrap and fixed burst transfer might get blocked when their base address is too close to the edge of the allocated address space. Therefore, full support for those two types of transfers should be implemented in the future.
- The most important work that needs to be done is to create better and safer DRAM chips that do not have latency deviations when accessing them and are not sensitive for rowhammering.

Bibliography

[1] Swadhesh Kumar and P K Singh. "An overview of modern cache memory and performance analysis of replacement policies". In: *2016 IEEE International Conference on Engineering and Technology (ICETECH)*. Mar. 2016, pp. 210–214. DOI: 10.1109/ICETECH.2016.7569243.

[2] Robert C. Steinke and Gary J. Nutt. "A Unified Theory of Shared Memory Consistency". In: *J. ACM* 51.5 (Sept. 2004), pp. 800–849. ISSN: 0004-5411. DOI: 10.1145/1017460.1017464. URL: https://doi-org.tudelft.idm.oclc.org/10.1145/1017460.1017464.

[3] Holger Pirk et al. "CPU and cache efficient management of memory-resident databases". In: Apr. 2013. DOI: 10.1109/ICDE.2013.6544810.

[4] Young-Ho Gong and Sung Woo Chung. "Exploiting Refresh Effect of DRAM Read Operations: A Practical Approach to Low-Power Refresh". In: *IEEE Transactions on Computers* 65.5 (May 2016), pp. 1507–1517. ISSN: 1557-9956. DOI: 10.1109/TC.2015.2448079.

[5] micron. *micron MT41K128M16 datasheet*. 2020. URL: https://www.micron.com/-/media/client/global/documents/products/data-sheet/dram/ddr3/2gb_1_35v_ddr3l.pdf.

[6] Damian Poddebniak et al. "Attacking Deterministic Signature Schemes Using Fault Attacks". In: *2018 IEEE European Symposium on Security and Privacy (EuroS P)*. Apr. 2018, pp. 338–352. DOI: 10.1109/EuroSP.2018.00031.

[7] ARM. "AMBA AXI and ACE Protocol Specification". In: (2011).

[8] ARM. *AXI protocol overview*. Oct. 2022. URL: https://developer.arm.com/documentation/102202/0300/AXI-protocol-overview.

[9] ARM. *Channel transfers and transactions*. Oct. 2022. URL: https://developer.arm.com/documentation/102202/0300/Channel-transfers-and-transactions.

[10] Yi-Liang Hong, Yui-Kai Weng, and Shih-Hsu Huang. "Hardware Implementation for Fending off Side-Channel Attacks". In: *2021 IEEE International Conference on Consumer Electronics-Taiwan (ICCE-TW)*. 2021, pp. 1–2. DOI: 10.1109/ICCE-TW52618.2021.9603186.

[11] Keith Nyasha Bhebe, Jian Liu, and Wenyu Qu. "Cache Side-Channel Attacks: Flush+Flush and the Countermeasures Time Gap". In: *2019 IEEE 25th International Conference on Parallel and Distributed Systems (ICPADS)*. Dec. 2019, pp. 880–887. DOI: 10.1109/ICPADS47876.2019.00129.

[12] Minwoo Jang et al. "Defending Against Flush+Reload Attack With DRAM Cache by Bypassing Shared SRAM Cache". In: *IEEE Access* 8 (2020), pp. 179837–179844. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2020.3027946.

[13] Paul Kocher et al. "Spectre Attacks: Exploiting Speculative Execution". In: *2019 IEEE Symposium on Security and Privacy (SP)*. May 2019, pp. 1–19. DOI: 10.1109/SP.2019.00002.

[14] Mark Zhao and G. Edward Suh. "FPGA-Based Remote Power Side-Channel Attacks". In: *2018 IEEE Symposium on Security and Privacy (SP)*. 2018, pp. 229–244. DOI: 10.1109/SP.2018.00049.

[15] Yuval Yarom and Katrina Falkner. "FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack". In: *Proceedings of the 23rd USENIX Conference on Security Symposium*. SEC'14. San Diego, CA: USENIX Association, 2014, pp. 719–732. ISBN: 9781931971157. DOI: https://eprint.iacr.org/2013/448.pdf.

[16] Petr Socha, Jan Brejník, and Matěj Bartik. "Attacking AES implementations using correlation power analysis on ZYBO Zynq-7000 SoC board". In: *2018 7th Mediterranean Conference on Embedded Computing (MECO)*. 2018, pp. 1–4. DOI: 10.1109/MECO.2018.8406034.

[17] Dong-Hyun Seo et al. "Enhanced Detection Range for EM Side-channel Attack Probes utilizing Co-planar Capacitive Asymmetry Sensing". In: *2021 Design, Automation and Test in Europe Conference and Exhibition (DATE)*. 2021, pp. 1016–1019. DOI: 10.23919/DATE51398.2021.9474155.

[18] Tuba Yavuz et al. "ENCIDER: Detecting Timing and Cache Side Channels in SGX Enclaves and Cryptographic APIs". In: *IEEE Transactions on Dependable and Secure Computing* (2022), pp. 1–1. DOI: `10.1109/TDSC.2022.3160346`.

[19] Mehwish Shaikh, Qasim Ali Arain, and Salahuddin Saddar. "Paradigm Shift of Machine Learning to Deep Learning in Side Channel Attacks - A Survey". In: *2021 6th International Multi-Topic ICT Conference (IMTIC)*. 2021, pp. 1–6. DOI: `10.1109/IMTIC53841.2021.9719689`.

[20] Tim Tuan, Tom Strader, and Steve Trimberger. "Analysis of Data Remanence in a 90nm FPGA". In: *2007 IEEE Custom Integrated Circuits Conference*. 2007, pp. 93–96. DOI: `10.1109/CICC.2007.4405689`.

[21] Alexander Schlösser et al. "Simple Photonic Emission Analysis of AES". In: *Cryptographic Hardware and Embedded Systems – CHES 2012*. Ed. by Emmanuel Prouff and Patrick Schaumont. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 41–57. ISBN: 978-3-642-33027-8. URL: `https://www.iacr.org/archive/ches2012/74280037/74280037.pdf`.

[22] Jaya Dofe. "Thermal Side-channel Leakage Protection in Monolithic Three Dimensional Integrated Circuits". In: *2022 IEEE 35th International System-on-Chip Conference (SOCC)*. 2022, pp. 1–2. DOI: `10.1109/SOCC56010.2022.9908080`.

[23] Y. Konishi et al. "Analysis of coupling noise between adjacent bit lines in megabit DRAMs". In: *IEEE Journal of Solid-State Circuits* 24.1 (1989), pp. 35–42. DOI: `10.1109/4.16299`.

[24] Barbara Aichinger. "DDR memory errors caused by Row Hammer". In: *2015 IEEE High Performance Extreme Computing Conference (HPEC)*. Sept. 2015, pp. 1–5. DOI: `10.1109/HPEC.2015.7322462`.

[25] Yoongu Kim et al. "Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors". In: *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. June 2014, pp. 361–372. DOI: `10.1109/ISCA.2014.6853210`.

[26] Jeremie S. Kim et al. "Revisiting RowHammer: An Experimental Analysis of Modern DRAM Devices and Mitigation Techniques". In: *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. May 2020, pp. 638–651. DOI: `10.1109/ISCA45697.2020.00059`.

[27] Lucian Cojocar et al. "Exploiting Correcting Codes: On the Effectiveness of ECC Memory Against Rowhammer Attacks". In: *2019 IEEE Symposium on Security and Privacy (SP)*. May 2019, pp. 55–71. DOI: `10.1109/SP.2019.00089`.

[28] Lidia Pocero Fraile, Apostolos P. Fournaris, and Odysseas Koufopavlou. "Revisiting Rowhammer Attacks in Embedded Systems". In: *2019 14th International Conference on Design and Technology of Integrated Systems In Nanoscale Era (DTIS)*. 2019, pp. 1–6. DOI: `10.1109/DTIS.2019.8734936`.

[29] Onur Mutlu and Jeremie S. Kim. "RowHammer: A Retrospective". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39.8 (Aug. 2020), pp. 1555–1571. ISSN: 1937-4151. DOI: `10.1109/TCAD.2019.2915318`.

[30] Yanbin Zhang and Qi Yuan. "A multiple bits error correction method based on cyclic redundancy check codes". In: *2008 9th International Conference on Signal Processing*. 2008, pp. 1808–1810. DOI: `10.1109/ICOSP.2008.4697490`.

[31] Shahram Babaie et al. "Double Bits Error Correction Using CRC Method". In: *2009 Fifth International Conference on Semantics, Knowledge and Grid*. 2009, pp. 254–257. DOI: `10.1109/SKG.2009.77`.

[32] G. Manoj Sai et al. "Diagonal Hamming Based Multi-Bit Error Detection and Correction Technique for Memories". In: *2020 International Conference on Communication and Signal Processing (ICCSP)*. 2020, pp. 0746–0750. DOI: `10.1109/ICCSP48568.2020.9182249`.

[33] A. Giray Yağlikçi et al. "BlockHammer: Preventing RowHammer at Low Cost by Blacklisting Rapidly-Accessed DRAM Rows". In: *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. Feb. 2021, pp. 345–358. DOI: `10.1109/HPCA51647.2021.00037`.

[34] Eojin Lee et al. "TWiCe: Preventing Row-hammering by Exploiting Time Window Counters". In: *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*. 2019, pp. 385–396. DOI: `10.1145/3307650.3322232`.

[35] Yeonhong Park et al. "Graphene: Strong yet Lightweight Row Hammer Protection". In: *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Oct. 2020, pp. 1–13. DOI: `10.1109/MICRO50266.2020.00014`.

[36] Jung Min You and Joon-Sung Yang. "MRLoc: Mitigating Row-hammering based on memory Locality". In: *2019 56th ACM/IEEE Design Automation Conference (DAC)*. 2019, pp. 1–6.

[37] Mungyu Son et al. "Making DRAM stronger against row hammering". In: *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*. 2017, pp. 1–6. DOI: 10.1145/3061639.3062281.

[38] Yicheng Wang et al. "Discreet-PARA: Rowhammer Defense with Low Cost and High Efficiency". In: *2021 IEEE 39th International Conference on Computer Design (ICCD)*. Oct. 2021, pp. 433–441. DOI: 10.1109/ICCD53106.2021.00074.

[39] Mottaqiallah Taouil et al. "LightRoAD: Lightweight Rowhammer Attack Detector". In: *2021 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. July 2021, pp. 362–367. DOI: 10.1109/ISVLSI51109.2021.00072.

[40] Kwangrae Kim et al. "HammerFilter: Robust Protection and Low Hardware Overhead Method for RowHammer". In: *2021 IEEE 39th International Conference on Computer Design (ICCD)*. Oct. 2021, pp. 212–219. DOI: 10.1109/ICCD53106.2021.00043.

[41] J. Misra and David Gries. "Finding repeated elements". In: *Science of Computer Programming* 2.2 (1982), pp. 143–152. ISSN: 0167-6423. DOI: https://doi.org/10.1016/0167-6423(82)90012-0. URL: https://www.sciencedirect.com/science/article/pii/0167642382900120.

[42] Eran Tromer, Dag Arne Osvik, and Adi Shamir. "Efficient Cache Attacks on AES, and Countermeasures". In: *Journal of Cryptology* 23.1 (2010), pp. 37–71. ISSN: 1432-1378. DOI: 10.1007/s00145-009-9049-y. URL: https://doi.org/10.1007/s00145-009-9049-y.

[43] Daniel Gruss et al. "Flush+Flush: A Fast and Stealthy Cache Attack". In: July 2016, pp. 279–299. ISBN: 978-3-319-40666-4. DOI: 10.1007/978-3-319-40667-1_14.

[44] Zili Kou et al. "Load-Step: A Precise TrustZone Execution Control Framework for Exploring New Side-channel Attacks Like Flush+Evict". In: *2021 58th ACM/IEEE Design Automation Conference (DAC)*. 2021, pp. 979–984. DOI: 10.1109/DAC18074.2021.9586226.

[45] Himanshi Jain, D. Anthony Balaraju, and Chester Rebeiro. "Spy Cartel: Parallelizing Evict+Time-Based Cache Attacks on Last-Level Caches". In: *Journal of Hardware and Systems Security* 3.2 (2019), pp. 147–163. ISSN: 2509-3436. DOI: 10.1007/s41635-018-0062-1. URL: https://doi.org/10.1007/s41635-018-0062-1.

[46] Cezar Reinbrecht et al. "LiD-CAT: A Lightweight Detector for Cache ATtacks". In: *2020 IEEE European Test Symposium (ETS)*. 2020, pp. 1–6. DOI: 10.1109/ETS48528.2020.9131603.

[47] Yinqian Zhang et al. "Cross-Tenant Side-Channel Attacks in PaaS Clouds". In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. CCS '14. Scottsdale, Arizona, USA: Association for Computing Machinery, 2014, pp. 990–1003. ISBN: 9781450329576. DOI: 10.1145/2660267.2660356. URL: https://doi-org.tudelft.idm.oclc.org/10.1145/2660267.2660356.

[48] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. "Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches". In: *Proceedings of the 24th USENIX Conference on Security Symposium*. SEC'15. Washington, D.C.: USENIX Association, 2015, pp. 897–912. ISBN: 9781931971232. URL: https://www.usenix.org/system/files/conference/usenixsecurity15/sec15-paper-gruss.pdf.

[49] Yuxin Liu et al. "Analysis of DRAM Vulnerability Using Computation Tree Logic". In: *ICC 2022 - IEEE International Conference on Communications*. 2022, pp. 986–991. DOI: 10.1109/ICC45855.2022.9839097.

[50] Peter Pessl et al. "DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks". In: *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, Aug. 2016, pp. 565–581. ISBN: 978-1-931971-32-4. URL: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/pessl.

[51] Zhiyuan Lv et al. "DRAMD: Detect Advanced DRAM-based Stealthy Communication Channels with Neural Networks". In: *IEEE INFOCOM 2020 - IEEE Conference on Computer Communications*. 2020, pp. 1907–1916. DOI: 10.1109/INFOCOM41043.2020.9155515.

[52] Minghua Wang et al. “DRAMDig: A Knowledge-assisted Tool to Uncover DRAM Address Mapping”. In: *2020 57th ACM/IEEE Design Automation Conference (DAC)*. 2020, pp. 1–6. DOI: `10.1109/DAC18072.2020.9218599`.

[53] ARM. “AMBA AXI and ACE Protocol”. In: (2011). DOI: `http://www.gstitt.ece.ufl.edu/courses/fall15/eel4720_5721/labs/refs/AXI4_specification.pdf`.

[54] Digilent. *Arty A7 Reference Manual*. 2020. URL: `https://digilent.com/reference/programmable-logic/arty-a7/reference-manual`.