technische universität
dortmund

Bachelorthesis

Configurable FPGA-based Access Latency Emulation for Non-Volatile Main Memory

Dennis Morczinek

18th July 2020

Supervisors:
Prof. Dr. Jian-Jia Chen
M. Sc. Christian Hakert

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl Informatik 12 (Eingebettete Systeme)
http://ls12-www.cs.tu-dortmund.de

Abstract

Since the drawbacks of using non-volatile memory (NVM) technologies as main memory are being addressed by researchers, its use as an energy e cient alternative to traditional DRAM is more interesting than ever. The impact of the greater memory access latencies of NVM compared to DRAM on a system can be investigated in systems that utilize at least one type of NVM as main memory. In many cases those systems do not exist yet, so the research has to be conducted on non-volatile main memory (NVMM) emulators or simulators. Yu Omori et al. developed such an emulator on an SoC-FPGA. Their emulator injects additional read and write delays to memory accesses that are con gurable by the user. However, the con gurations are applied to the whole memory of the emulator, so emulating hybrid systems with more than one NVMM type is not possible.

This thesis extends the FPGA emulator design of Yu Omori et al. to allow the emulation of more than one NVMM type by making it possible to de ne areas in the main memory with di erent access latencies. The underlying emulator architecture is adapted so that the corresponding latency for each area can be stored and the delay injection logic receives the appropriate value when a memory access is performed. The count of de nable areas and the utilization of the FPGA are kept in balance to allow for future modi cations of the emulator design.

Contents

1 Introduction

1.1 Motivation and Contribution

Because of its fast access time and robustness, the main memory of a system is typically based on DRAM. Recently emerging non-volatile memories (NVMs) such as Phase-Change Memory (PCM), Ferroelectric RAM (FeRAM) and Magnetoresistive RAM (MRAM) show almost no leakage power in the memory cells [20], increase energy e ciency of a system compared to DRAM [20] and o er the possibility to realize durable data structures [16]. Due to these properties, NVMs are considered as a substitute to DRAM for use as main memory. Overcoming their drawbacks, like slower access time and limited endurance compared to DRAM, and examining their impact on a system when used as main memory are ongoing research topics [14] [19].

Emulating the behaviour of non-volatile main memory (NVMM) enables in-depth research of its impact on a system. In fact, for some types of non-volatile memory either emulation or simulation is required since no real system that utilizes this type as main memory exists. Various emulators or simulators that mimic the behaviour of NVMM have already been developed [15] [16] [18].

Because of the distinct advantages and disadvantages of di erent NVMs, some systems utilize more than one type of NVM as main memory. Moreover, there are types of NVM that can be set to operate in a low energy mode for certain memory regions to reduce energy consumption. Accordingly, an emulator of such systems requires a con gurable count of memory sections, each mimicking a certain type of NVM or energy setting.

In this thesis an existing NVMM emulator design that emulates the read and write latencies of a single NVM for the entire available main memory is extended. This emulator is based on an FPGA, an integrated circuit that can be con gured to realize any kind of digital logic within its resource limitations [13]. The extension will allow the user to de ne areas in the main memory with di erent read and write latencies. As a result, systems with more than one type of NVMM, as well as systems with one type of NVMM that has con gurable energy settings, can be emulated regarding their memory access latencies.

1.2 Structure of the Thesis

Different implementation approaches of memory access simulators and emulators, including the one used in this thesis, are at first introduced and briefly compared in Chapter 2 using concrete examples of available implementations.
Thereafter, in Chapter 3, the concepts of the technologies relevant to this thesis are explained. A concise overview of configurable hardware in general is provided before explaining FPGAs and associated terms as well as their use in system environments in greater detail. The architecture and specification of the hardware environment used in this thesis is presented thereon. Its adopted on-chip communication protocol named Advanced eXtensible Interface is introduced in the following.
Chapter 4 covers the extensions made to the FPGA-based NVMM emulator after initially demonstrating its actual state. The proposed idea is presented on a high level of abstraction at first, followed by detailed explanations of each added component. A program to access the emulator in order to change the emulated latencies is introduced at the end of this chapter.
At last, Chapter 5 evaluates how the design extensions affect system performance and FPGA resource utilization. The impact on system performance is measured in the nanosecond scope and analyzed subsequently. The resource utilization changes depending on how many NVM types can be emulated in the emulator's main memory. The correlation between memory section count, i.e. the number of emulatable NVMs, and resource utilization is analyzed and functions to determine the approximated utilization by section count are provided.

2 Related Work

2.1 Memory Access Simulators

The simulation of memory access characteristics of different memory types allows the exploration of its behaviour and impact on a system without requiring physical access to the type of memory to be examined. Different approaches have been made that allow to either simulate or emulate NVMMs. In general, emulators try to mimic the visible behavior of the target memory to the best of their ability, whereas simulators reconstruct the underlying architecture, which will consequently result in an emulation of the target. One possible classification is to differentiate between software-based simulators, external hardware emulators and SoC-FPGA-based emulators

Software-based simulators like NVMain 2.0 [18] are able to accurately simulate NVMs and save the information of the memory accesses in trace files. This however comes at the cost of long simulation times, especially for full featured operating systems.

External hardware emulators do not suffer from this problem. They are placed in-between the system memory and the system itself. Memory requests have to pass the external simulator, which is able to e.g. modify the latency of the requests to match that of a certain NVM type. HMTT is a memory accesses trace system implemented on an external board that is connected to the monitored system via a DIMM slot [12] . HMTT itself is only a monitoring system, but it is easy to imagine an extension where the memory accesses are not only snooped but also modified.

System-on-a-Chip-FPGAs (SoC-FPGAs) are, as the name indicates, FPGAs that are strongly linked to a processing system (cf. Chapter 3.1.3). The approach of SoC-FPGA-based emulators is not too different from that of external hardware emulators. A dedicated hardware, only this time implemented in an FPGA, is responsible for modifying the memory request issued by the processing system to match that of NVMs. So unlike external emulators, the dedicated emulation hardware is on the same board as the system. Therefore, the emulation hardware has access to much more system resources than just the memory interface, which potentially allows for precise measurements of the full-system performance impact of NVMs. Examples for SoC-FPGA-based emulators are TUNA [15] and the NVMM emulator by Yu Omori et al. [16] that is extended in this thesis.

3 Technical Background

3.1 Configurable System Hardware

Custom hardware can be built in various ways. Complex digital circuits that need to be manufactured in large numbers are often designed as application-specific integrated circuits (ASICs) [13]. Once an ASIC design reaches production, it is not possible to subsequently modify it. If a design does not need to be produced in large numbers or needs to be changed frequently (for example for prototyping or if the requirements are not well defined yet), reconfigurable digital circuits can be used. The relatively simple Programmable Logic Array (PLA) is capable of implementing combinational circuits, however, for sequential designs, mainly more advanced technologies like Complex Programmable Logic Devices (CPLD) or Field Programmable Gate Arrays (FPGA) are used [13, p. 24 ff.]. Ultimately, it depends on the use case, which device fits the target design best. The simpler architecture of CPLDs compared to FPGAs can usually be beneficial in terms of cost, while FPGAs offer a vast amount of logic, storage elements and sometimes additional atomic building blocks for common circuit elements like multipliers and random-access memory [1] [8].

Because, as mentioned before, the emulator of this thesis is designed in an FPGA, its basic concept is explained first. Hereafter the concrete system architecture used in this thesis is presented.

3.1.1 FPGAs

The concept of an FPGA consists of a homogeneous field of Configurable Logic Blocks (CLB), IO-Blocks and Switch Matrices [13, p. 28]. The CLBs implement the desired logic whereas IO-Blocks drive in- and outputs of the FPGA. The configuration of the switch matrices determines the interconnection between the FPGAs components. While the exact architecture of those components differs depending on the manufacturer and product line [2] [8], the basic concept can be described by a generic model as presented in Figure 3.1.

Figure 3.1: Concept of a Generic FPGA Structure

A basic CLB (cf. Figure 3.2) consists of three main components: a lookup-table (LUT), a ip- op and a multiplexer. LUTs are often labeled n-LUTs where n indicates the number of inputs. A n-LUT has 2^n rows and $n + 1$ columns. Therefore, one n-LUT can be used to implement any binary function $f0; 1g^n ! f 0; 1g$ as a truth table. When an FPGA gets con gured, the LUTs are lled and the selection line of the multiplexer is set to a xed value so that one of its inputs gets passed through continuously. If the circuit that needs to be implemented is combinational, the output of the LUT is selected; otherwise, for sequential circuits, the output of the ip op is selected.

To reduce the utilization of CLBs for common arithmetic computations, full adders are added to many CLB designs (cf. Figure 3.3). Depending on the con guration of the CLB, an extra multiplexer (depicted as MUX 1) controls, whether the output of the full adder or the LUT gets passed through. The 4-LUT is divided into two 3-LUTs, each of which is additionally connected to the full adder. Another multiplexer (depicted as MUX 2) then combines the two 3-LUTs into one 4-LUT. Numerous other adaptions are made to

Figure 3.2: Basic Con gurable Logic Block

Figure 3.3: Extended Con gurable Logic Block

CLBs by their manufacturers for resource- and thus space- and cost-saving reasons and for performance reasons [7].
A single CLB itself is too small to implement complex circuit designs. The interconnection between multiple CLBs, also called routing, is done by switch matrices and allows large designs that are only bound by the number of available ressources.[1] A simpli ed switch matrix is depicted in Figure 3.4. A memory unit, for example a register, contains a bit string that varies depending on the con guration of the switch matrix. The individual bits are connected to transistors which controls their resistance and thus the permeability of signals between the four wires.
The con guration, also called programming, of an FPGA is usually done with software designed for this purpose that follows a stringent work ow (cf. Figure 3.5). The software used in this thesis isVivado by Xilinx since the FPGA is produced by the same company. A hardware description language (HDL) such as Verilog or VHDL is used to describe the target hardware design. This description can be used to simulate the design behaviour

Figure 3.4: Simpli ed Switch Matrix

[1] Assuming that all timing constraints can be met.

since HDLs were originally designed to simulate circuits on a register-transfer level. When no information about timing behaviour is given explicitly in the simulation, the signals experience no delays and arrive instantaneously at their targets. Therefore, a correct behavioural simulation does not guarantee a functional physical circuit. In the following step, a tool translates the HDL description on register-transfer level into a textual netlist that can be displayed as a gate level schematic { this step is called synthesis. The resulting schematic depends on the optimization goals of the synthesis tool. The functional gate level simulation o ers insights into the synthesized design but still relies on timing behaviour descriptions provided by the developer. Before the nal implementation step, the developer can decide to do manual oorplanning, where a con ned physical space on the chip can be set for selected circuit parts. This can be bene cial in terms of route lengths and therefore timing delays. When the oorplanning has nished or has been skipped, the implementation tool is used to place the synthesized design into the FPGA. The main focus of the implementation tool is to nd a placement into the FPGA that does not violate any timing constraints or exceeds the resource capacity. If those conditions are met, other optimizations can be taken into consideration. This step results into a bitstream that gets loaded into the FPGA and sets all a ected CLBs, IO-Blocks and switch matrices. Because the FPGA loses its con guration when powered o , it has to be reprogrammed using the bitstream when powered on again.

Figure 3.5: FPGA Design Work ow

3.1.2 IP Cores

Intellectual property cores (IP cores) approximately are to FPGAs what libraries are to programming languages. Instead of rewriting commonly used designs, they can be packed into IP cores for future use. There are two types of IP cores: hard cores and soft cores. Hard IP cores are built into the FPGA fabric and as a result cannot be modified. If a functionality is needed in many FPGA use cases, it is beneficial for resource usage and timing when the FPGA offers a corresponding hard IP core. Dedicated RAM called Block RAM (BRAM) and arithmetic units are examples that are commonly found in many FPGA architectures. Soft IP cores on the other hand are placed into the FPGA via synthesis and implementation, following the regular design workflow (cf. 3.5). They are either delivered as HDL code on a register-transfer level or as a netlist on gate level.

3.1.3 SoC-FPGA Architecture

To receive their inputs and produce outputs, FPGAs are always embedded into a circuitry. A relatively new approach of using FPGAs is the combination of an FPGA fabric and a processor on one chip. The in- and outputs of the FPGA are connected to the systems components, which enables flexible and efficient hardware extensions to them or the full system. This architecture, called SoC-FPGA, is used to implement the emulator in this thesis. More specifically, the Xilinx Zynq-7000 SoC ZC706 Evaluation Board that is populated with the Zynq-7000 XC7Z045 SoC is used here and since architectural characteristics differ for each product, its architecture is presented from now on.

Xilinx Zynq-7000 XC7Z045 SoC

As shown in Figure 3.6, the architecture is divided into a Processing System (PS) and a Programmable Logic (PL) on a single die. The PS contains an ARM Cortex-A9 dual-core CPU, interfaces for external memory, I/O peripherals and has access to 1 GB DDR3 on-chip memory [11]. The PL consists of an FPGA that, inter alia, can access the memory interfaces on the PS. Moreover, the PL can utilize an additional 1 GB of dedicated DRAM through an IP core called Memory Interface Generator (MIG), that acts as an memory controller. If used in a design, this memory is mapped into the address space of the PS. Other memory structures, either provided as IP cores like BRAM or realized with CLBs, can also be accessed by the PS via reserved memory spaces. BRAM is a common resource of FPGAs that can store large amounts of data while still allowing high performance accesses. This SoC-FPGA offers 19.1 MiB of BRAM.
The FPGA is also connected to the processor and a central interconnect that provides access to I/O peripherals. The PL is clocked by a 200 MHz oscillator which leads to a clock period of $T = (200\,MHz)^{-1} = 5$ ns. The MIG supports clock frequencies up to 300 MHz [5], however, in this thesis the aforementioned 200 MHz oscillator either directly

Figure 3.6: Zynq-7000 Architecture Overview

provides the clock signal for the components or drives clock generators to generate di erent clock domains.

3.2 Advanced eXtensible Interface

Advanced eXtensible Interface(AXI), part of the Advanced Microcontroller Bus Architecture (AMBA), is a protocol designed for on-chip communication. It has been adopted by Xilinx for communication between IP cores within their FPGAs. As presented in Chapter 3.1.3, the SoC-FPGA used in this thesis is by Xilinx and thus also implements AXI.
The protocol implements a master/slave model of communication. The interconnection between a master and a slave is seperated into ve channels with each channel using its own distinct signals (cf. Figures 3.7 and 3.8): read address (AR), read data (R), write address (AW), write data (W) and response (B).

Figure 3.7: AXI Read Channel Architecture

Figure 3.8: AXI Write Channel Architecture

The read and write address channels carry control information about the ensuing data transfers. The data is then transmitted in bursts via the read data or the write data channel depending on whether reading or writing is to take place. An additional response channel is used for writes, so that the master is informed by the slave whether the write was successful or not. Each channel is clocked by the same global clock signal, ACLK, and all components trigger on its rising edge. The active-low ARESETn is another shared global signal that is used to (asynchronously) cancel transactions.

Every channel has two dedicated signals, VALID and READY, that implement a handshake mechanism. The source provides an address, data or control information and asserts its VALID signal to inform the destination that all corresponding signal lines are valid. The destination asserts its READY signal as soon as it is ready to accept the information. Only after a handshake has taken place, the data transmission can begin. The order of assertion of the VALID and READY signals does not matter as long as the source does not assert the VALID signal before the information is provided. Furthermore, the information must be kept stable the whole time the VALID signal is asserted. Figure 3.9 presents all three feasible handshake situations.

Figure 3.9: Feasible AXI Handshakes

Figure 3.10 depicts an example of an AXI read transaction. For simplicity reasons, signals that are not relevant for every read transaction, such as memory type or quality of service indicators, are omitted. ARADDR describes the address from which to read. ARLEN=1 indicates the read burst length, or in other words the amount of data transfers (also called beats) of a single read access. The size in bytes of one beat is encoded by ARSIZE. Also encoded, ARBURST defines the read addresses for successive beats. The interpretation of encoded signals can be obtained from Table 3.1.

The burst type FIXED accesses the same address provided by ARADDR repeatedly for every beat. INCR accesses consecutive memory locations. Useful for cache line accesses. WRAP is similar to INCR, except that an upper address boundary is determined. If the address reaches this boundary, it is wrapped around to the lowest address of the burst.

In this specific example, the read burst first accesses the data saved on address 0xA0. After a successful data transfer, the address is incremented by the size of a beat internally to access the following four bytes. Since a total of four data transfers of four bytes are performed, 16 bytes on the consecutive memory locations 0xA0, 0xA4, 0xA8 and 0xAC are read by this transaction. The memory contents are put on the data line RDATA subsequently for the master to read. RRESP (cf. Table 3.1) indicates the status of the access. OKAY for a normal access success, EXOKAY for a succeeding exclusive access[2], SLVERR for accesses that reached the slave but returned an error, and lastly DECERR, if there is no slave for the given transaction address. The last beat of a burst must be flagged by RLAST.

Figure 3.10: Example Timing Diagram for AXI Read

[2] Exclusive accesses are used to ensure that only one master can access a certain slave at a time. Additional logic is needed by the slave to support this functionality. They are not used in this thesis.

Table 3.1: Encoded Signals for AXI

Burst Size		Burst Type		Response	
AxSIZE	Bytes	AxBURST	Type	xRESP	Status
0b000	1	0b00	FIXED	0b00	OKAY
0b001	2	0b01	INCR	0b01	EXOKAY
0b010	4	0b10	WRAP	0b10	SVLERR
0b011	8	0b11	Reserved	0b11	DECERR
0b100	16				
0b101	32				
0b110	64				
0b111	128				

An example of a write transaction is presented in Figure 3.11. The signals of the write address channel work analogously to the read address channel signals. However, the data channels di er for reads and writes. The data to write is put on WDATA, for which WSTRB indicates which bytes on WDATA hold valid data { one bit for every data byte. Thus, in this example the two bytes of 0xD5 and 0xD6 (equivalent to 0x00D5 and 0x00D6) are marked as valid for the write. For 0xD7E7, only the rst byte 0xD7 is marked as valid, hence only this byte lane is updated to the memory. The signal BRESP of the write response channel indicates the status of the nished write burst. The interpretations of all encoded signals are analog for read and writes transactions (cf. Table 3.1).

Figure 3.11: Example Timing Diagram for AXI Write

4 Development

In order to emulate the memory access latencies of systems with multiple main memory types or energy-adjustable NVMs, this thesis extends the NVMM emulator based on a SoC-FPGA implemented by Yu Omori et al. [16], which is called original emulator from now on. This chapter first introduces the design and features of this emulator and thereafter presents the development of the extensions and adjustments made to it, resulting in the extended emulator

4.1 NVMM Emulator Design

Chapter 3.1.3 presents the architecture of the Zynq-7000 XC7Z045 SoC-FPGA on which the original emulator is built on. The 1 GB of DRAM connected to the PL is taken as the emulated NVMM. Two models of a delay injection logic, a fine-grain and a coarse-grain one, are implemented in the PL. The coarse-grain model injects an additional delay for memory requests between the last level cache (LLC) of the CPU and the MIG. The fine-grain model injects the additional delay in the memory itself by increasing the Row Address to Column Address Delay (t_{RCD}) and the Row Precharge Time (t_{RP})[1]. This is done by modifying the MIG and allows for a better capture of the effects of bank parallelism and row-buffer access locality [16]. For this thesis, only the coarse-grain model is provided and will therefore be discussed from now on. Figure 4.1 shows an architectural overview of the emulator.

[1] For DRAM, t_{RCD} describes the minimum delay necessary before accessing a column in an active word line. t_{RP} describes the minimum delay necessary between disabling a selected word line and selecting a different one.

Figure 4.1: Architectural Overview of the Original Emulator

The delay injection logic consists of two IP core blocks called LatSet (latency set) and LatGen (latency generate). LatSet acts as an AXI slave, with the PS being the master. As shown by Figure 4.2 it is not connected to the master directly but to another IP core (shortened to IP henceforth) called AXI SmartConnect. AXI SmartConnect is an IP provided by Xilinx that enables the connection of m masters to n slaves and, inter alia, distributes the signals coming from a master to the corresponding slave[2]. LatSet implements two 32 bit wide registers to separately store the additional read and write latencies named rlat and wlat. They can be written to by the PS through a general purpose AXI master port GP0AXI. This port is mapped into the address space 0x40000000 to 0x7FFFFFFF and the registers are addressable at 0x43C00008 (rlat) and 0x43C0000C (wlat). The latencies are stored as the numbers of additional clock cycles that must elapse before the memory request is forwarded.

Figure 4.2: Partial Emulator Block Diagram Showing LatSet

Another general purpose AXI master port, GP1AXI, is connected to the MIG via a further AXI SmartConnect. Since GP1AXI is assigned to the address space 0x80000000 to 0xBFFFFFFF, the emulated NVM can be accessed at precisely these addresses. However, instead of connecting all AXI signals directly to the MIG, the AXI address channel ready

[2] AXI SmartConnect is not necessary for one-to-one connections but simplifies possible future extensions to the design.

and valid signals are passed through LatGen. The outputs of the rlat- and wlat-registers coming from LatSet are connected to LatGen as well, so that the additional memory request delays can be generated. It shall be noted here in advance that GP0_AXI and GP1_AXI are parts of different clock domains, which will be revisited in Chapter 4.2.2. The clock of GP0_AXI is generated by the PS whereas GP1_AXI is driven by a clock generated by the MIG (called UI_CLK in the design but simply referred to as memory clock henceforth). Figure 4.3 shows the complete block design for the emulator.

Figure 4.3: Complete Block Diagram for the Original Emulator Design

The concept of LatGen is to delay the handshake signals of the AXI read and write address channels before passing them to either the MIG or the PS (through the AXI SmartConnect). For read requests, this concerns the arvalid signal of the master (M0_AXI_arvalid) and the aready signal of the slave (S0_AXI_aready). The signals are asserted if, in addition to the state of the source signals, either rlat is zero (and hence no additional delay is required) or the signal ready_ar is asserted (cf. Listing 4.1).

Listing 4.1: Read Address Handshake Signal Assertion in Module LatGen

```
1 wire zero_ar = (rlat == 32'h0);
2 assign M0_AXI_arvalid = (zero_ar | ready_ar) & S0_AXI_arvalid;
3 assign S0_AXI_arready = (zero_ar | ready_ar) & M0_AXI_arready;
```

The signal ready_ar is asserted for the length of one clock cycle if a counter (that gets incremented every clock signal) reaches the delay value rlat . The following code listing 4.2 will cover the mechanism that asserts ready_ar in greater detail.

Listing 4.2: Assertion of ready_ar in Module LatGen

```
1  reg          ready_ar;
2  reg          busy_ar;
3  reg [31:0]   cnt_ar;
4
5  always@(posedge AXI_ACLK)
6  begin
7      if (  AXI_ARESETN)
8      begin
9          cnt_ar    <= 32'h1;
10         ready_ar <= 1'b0;
11         busy_ar   <= 1'b0;
12     end
13
14     if (  busy_ar & S0_AXI_arvalid &   ready_ar)
15     begin
16         busy_ar <= 1'b1;
17         cnt_ar   <= 32'h1;
18     end
19
20     if (busy_ar)
21         cnt_ar <= cnt_ar + 32'h1;
22
23     if (busy_ar & cnt_ar >= rlat)
24     begin
25         busy_ar   <= 1'b0;
26         ready_ar <= 1'b1;
27     end
28
29     if (ready_ar)
30         ready_ar <= 1'b0;
31 end
```

The assertion of S0_AXI_arvalid implies that the PS is requesting a read transaction and has already provided the read address and control information. Since S0_AXI_arvalid

is held high until the handshake completes, an signal called busy_ar controls, whether the counter has to be initialized or is already active, thus delaying the memory request. If busy_ar is deasserted and S0_AXI_arvalid is asserted (cf. lines 14{18, Listing 4.2), this means that a new read request has been initiated (the check for ready_ar prevents an error that is explained further below). The counter is reset and busy_ar is asserted. Because this process already takes one clock cycle, the counter is set to one. For the following clock cycles, the counter cnt_ar is incremented (cf. lines 20{21, Listing 4.2) until it reaches the value of rlat (cf. lines 23{27, Listing 4.2). In this case, the counter is deactivated by deasserting busy_ar, and ready_ar is set high until the next clock signal (cf. lines 23{30, Listing 4.2). If the condition of ready_ar had not been checked in line 14, the counter would be activated again in the following clock cycle that deasserts ready_ar, because busy_ar is already zero and S0_AXI_arvalid is not deasserted by the source until the handshake completes. Lastly, in case the PS sends an AXI reset signal, all values are reset (cf. lines 7{12, Listing 4.2).

This mechanism is analogous for write transaction delays. S0_AXI_arvalid is replaced with S0_AXI_awvalid, ready_ar is called ready_aw and instead of comparing the counter value against rlat, wlat is used. The continuous assignment statement that controls the assertion of the handshake signals therefore looks similar (cf. Listing 4.3).

Listing 4.3: Write Address Handshake Signal Assertion in Module LatGen

```
1 wire zero_aw = (wlat == 32'h0);
2 assign M0_AXI_awvalid = (zero_aw | ready_aw) & S0_AXI_awvalid;
3 assign S0_AXI_awready = (zero_aw | ready_aw) & M0_AXI_awready;
```

Now that the delay injection mechanism is known, the reason why the minimum realizable additional latency is 10 ns can also be explained. When either rlat or wlat is zero, zero_ar or respectively zero_aw are asserted and the handshake signals are immediately passed through. If on the other hand the delay value is set so a value greater than zero, the busy signal is asserted as soon as the handshake signal of the source arrives at time t. In the best case scenario { where the delay value is set to one { the ready signal is asserted in the following clock cycle (after t+1) because cnt_ar or cnt_aw are initialized with the value one. It takes at least until the end of this cycle (t+2) for the receiver to see the handshake signals. Figure 4.4 illustrates this situation for a write situation where wlat is set to one. The arrows indicate the conditions that led to assertion of the signals they are pointing to.

Figure 4.4: Example Showing Minimum Realizable Latency

4.2 Design Extensions

The previously presented original design emulates one type of NVM for the whole 1 GB PL-DRAM. Once rlat and wlat are set, all memory requests experience the same additional latency. The fundamental idea of the extension is to divide the PL-DRAM into sections of the same size which implement different, user-adjustable read and write latencies, so that each section corresponds to the behaviour of one main memory type. Depending on the use case that shall be emulated and thereupon evaluated with this emulator, the amount of required sections within the DRAM can vary vastly. Therefore, the greatest amount of sections possible that meets all timing requirements and still allows for future extensions to the design (i.e. that does not exhaust one or more resources of the FPGA) is implemented in the approach presented hereby. The correlation between resource utilization and section quantity will be examined later in Chapter 5.

The first design draft continues the already implemented approach of using a register pair to store the read and write latencies. Each memory section is associated with its own register pair instead of using just one pair for the whole DRAM. The outputs of all read latency registers, as well as the outputs of all write latency registers, are connected to one of two multiplexers, which are controlled by the address of the current memory request. The outputs of the multiplexers are connected to the rlat and wlat inputs of LatGen, so that the latency generation logic always receives the proper values for the current memory request. This design, called *Register Module* hereafter, is expected to behave equal to the former one in terms of latency generation and timing, albeit realizing the desired memory sections. However, beyond a certain size, multiplexers face issues meeting timing constraints in FPGAs [6]. As this limits the number of achievable memory sections to 64 in this case (cf. Chapter 4.2.1) and more or smaller sections might be needed for certain emulation use cases, the emulator is expanded with another module called *BRAM Module*.

The BRAM of the FPGA is used to store further latencies and again, depending on the memory request address, the corresponding latency values are send to LatGen. Utilizing the BRAM enables the storage of many latency values and thus, compared to the Register Module approach, more and smaller sections. As a drawback, reading the values from the BRAM takes a fixed amount of additional clock cycles to elapse before rlat and wlat are sent to LatGen, resulting in slower memory access times if no additional latency is desired, which also results in a in a greater feasible minimum latency for the memory sections whose latency values are stored in the BRAM.

To let the user decide which underlying conditions (section size, section count and timing behaviour) of the two modules fits their use case best, the PL-DRAM is divided into two 500 MB domains, each of which implements one of the ideas (cf. Figure 4.5). The module used is eponymous for the respective domain (Register Domain and BRAM Domain). The Register Module and the BRAM Module will continuously output a latency value, even if the memory access address is not intended for the domain they manage (cf. Chapter 4.2.3 for more details). Therefore, a multiplexer, called Module MUX, is controlled by the current memory request address. If the address lies in the first half of the DRAM (between 0x80000000 and 0x9FFFFFFF) the output of the Register Module is passed to LatGen; if it lies in the second half of the DRAM (between 0xA0000000 and 0xBFFFFFFF), the output of the BRAM Module is passed on. Taken together, by controlling the input of LatGen,

Figure 4.5: Architectural Overview of the Extended Emulator

the Module MUX is responsible for the realization of the split memory domains, whereas the Register Module and the BRAM Module store the latency values for each memory section within their domains.
Similar to LatSet in the original emulator, both modules are addressable through the AXI general purpose portGP0AXI to change the content of the register pairs, respectively the BRAM registers. The total picture of the architectural concept is visualized in Figure 4.5. The connections of the GP1AXI addresses toLatGen are new compared to the original emulator design. The necessity of those connections is revisited in Chapter 4.2.2.

4.2.1 Register Module

As indicated in Chapter 4, the architecture of the Register Module is based on the module LatSet of the original emulator. Two registers, that form a register pair, store the values for rlat and wlat for a speci ed memory section. The original emulator uses one register pair whose outputs are directly connected to LatGen, thus implementing a single memory section (the whole PL-DRAM). The Register Module implements 64 register pairs to divide the Register Domain of the DRAM into the same amount of sections. Each register pair is responsible for savingrlat and wlat of one section. Unlike before, each memory request might a ect a di erent section, so a logic unit has to determine which register pair contents to pass to LatGen, more precisely, to the Module MUX beforehand. This logic unit is realized by two 64-to-1 multiplexers, where each data line is 32 bit wide. As shown in Figure 4.6, all rlat -registers of a register pair are connected to the input ports of one multiplexer and all wlat -registers to the input ports of another multiplexer.Their selection lines are connected to either the current read or write address. Since those addresses are latched, the most recent addresses are present constantly. So if for instance a read request is issued,rlat changes as soon as the read address updates whilewlat keeps the value of the last issued write request. Before explaining how the selection of the correct input is realized by the multiplexers, the size of a memory section must be known. With# sect denoting the count of memory sections andmemsize being the total size of the memory that shall be divided, the size of a section,sectsize, is given as follows:

$$\text{sectsize} = \frac{\text{memsize}}{\#\,\text{sect}} = \frac{512\,\text{MiB}}{64} = 8\,\text{MiB} = 8388608\,\text{B}: \tag{4.1}$$

Accordingly, every register pair is responsible for providing the read and write latencies for an address space of 8388608 B (0x00800000 addresses[3]). The read and write addresses are 30 bits wide, but because the Register Domain is half the size of the PL-DRAM, only the rst 29 bits are of interest and the 30th bit is omitted. Since $\log_2(8388608) = 23$, only the 23 low-order bits will change when the addresses of di erent memory requests stay in the same memory section. Conversely, checking the remaining six high-order bits of the

[3] The PL-DRAM is byte-addressable.

Figure 4.6: Register Module Architecture

address will yield the number of the memory section (starting at zero) that is currently accessed as shown in the example below (cf. 4.2).

$$
\begin{array}{lll}
\text{Section } 0 & \begin{cases} 000000\,0000000000000000000000_{|_{0}} & 0x00000000 \\ 000000\,1111111111111111111111_{|_{0}} & 0x007FFFFF \end{cases} \\
\text{Section } 1 & \begin{cases} 000001\,0000000000000000000000_{|_{1}} & 0x00800000 \\ 000001\,1111111111111111111111_{|_{1}} & 0x00FFFFFF \end{cases} \\
\text{Section } 63 & \begin{cases} 111111\,0000000000000000000000_{|_{63}} & 0x1F7FFFFF \\ 111111\,1111111111111111111111_{|_{63}} & 0x1FFFFFFF \end{cases}
\end{array}
\tag{4.2}
$$

Therefore, only the six high-order bits of the address are connected to the selection lines of each multiplexer, automatically realizing the desired selection behaviour. The corresponding Verilog description of the multiplexers is done in a single Verilog module as depicted in Listing 4.4. At any change of the six high-order address bits or one of the rlat- or wlat-registers, the value of the address is compared against all possible cases (0 to 63) and the outgoing signals are updated. These always-blocks are synthesized into combinational logics that are functionally equivalent to multiplexers. Since every possible case is covered in the case-statement, the default case is redundant and could be omitted.

Listing 4.4: rlat -Register and wlat -Register Multiplexers

```
1  always@(*)
2  begin
3      case(rd_request_addr[28:23])
4          0  : rlat_out = rlat_s0;
5          1  : rlat_out = rlat_s1;
6          ...
7          63 : rlat_out = rlat_s63;
8          default : rlat_out = 32'h0;
9      endcase
10 end
11
12 always@(*)
13 begin
14     case(wr_request_addr[28:23])
15         0  : wlat_out = wlat_s0;
16         1  : wlat_out = wlat_s1;
17         ...
18         63 : wlat_out = wlat_s63;
19         default : wlat_out = 32'h0;
20     endcase
21 end
```

Similar to the original emulator, the rlat - and wlat -registers can be accessed through the general purpose AXI master port GP0 AXI of the PS. Each register is 4 byte wide (32 bit word length), so the registers realize a storage space of 4 B (# sect 1) = 2 4 B (64 1) = 504 B. An address space of 0x1F8 addresses (0x1F8 = 504) is required. The addresses from 0x42000000 to 0x420001F8 are reserved for this purpose[4]. The addresses of the registers for one section follow one another, which is why rlat - and wlat -registers alternate. The address of the rlat -register for section $i \in \{0;:::;63\}$ is given by

$$rlat\_addr_i = 0x42000000+(8i) \tag{4.3}$$

[4] As mentioned in Chapter 4.1, the address space from 0x40000000 to 0x7FFFFFFF is reserved for memory structures connected to GP0 AXI. These structures are synthesized to be byte-addressable as well.

and the address of the wlat -register for section i by

$$wlat\_addr_i = 0x42000000+(8i+4) : \quad (4.4)$$

The reason why 64 sections were chosen was already touched in Chapter 4.2. Unlike the BRAM Domain, the Register Domain should show the same timing behaviour as the PL-DRAM of the original emulator. This means that as soon as a memory request is issued, the rlat or wlat value has to be stable at LatGen at the rising edge of the next clock cycle (of the memory clock domain). Hence, the propagation delay of the signals between the Register Module and LatGen must not exceed the length of one clock cycle. The propagation delay is a ected by the design of the Register Module itself, the Module MUX, which is on the route between the Register Module and LatGen, and the total route length through the FPGA fabric. The synthesis and implementation tools of the FPGA design software optimize the described design to guarantee that all timing (as well as logical and physical) constraints are met [9] [10].
For the presented design, 64 sections within the Register Domain are the achievable upper limit. Raising the count of sections requires the usage of an additional bit for the selection lines of the multiplexers. The count of attainable sections would increase to $2^{6+1} = 128$ and the size of one section would decrease to half the size, which is 4 MiB (cf. Equation 4.1). A multiplexer with one additional selection bit is twice as large, which requires double the amount of CLBs to be used. This issue is illustrated in example implementations (cf. Figure 4.7) of a 4-to-1 and an 8-to-1 multiplexer within an FPGA, where the dedicated multiplexers within the CLBs are utilized and each LUT implements a 2-to-1 multiplexer. The increased usage of CLBs leads to longer data routes and ultimately the propagation

(a) 4-to-1 Multiplexer Implementation

Figure 4.7: Example of Multiplexer Implementations with CLBs [3]

(a) 8-to-1 Multiplexer Implementation

Figure 4.7: Example of Multiplexer Implementations with CLBs (cont.) [3]

delay of the data signals exceeds the length of one clock cycle. In this particular case, the memory request address lines are connected to such a high count of CLBs, that the propagation of their signals cannot meet the timing constraints. If an implementation fails due to strict constraints, using a di erent implementation strategy that focuses on solving the resource problem (e.g. time, area or power) can help [9]. However, even with a strategy that runs \timing-driven optimizations to potentially improve overall timing slack" [9, p. 181], the design with 64 sections is the largest one that meets all timing constraints.

4.2.2 BRAM Module

The BRAM Domain is implemented for designs that require a ner granularity of the memory sections compared to the Register Domain. The BRAM is an unused resource so far and it has the ability to store a great amount of quickly accessible data, which makes it a good candidate for this goal. The latency values are stored in the BRAM where the appropriate value rst has to be read from before it is passed to LatGen through the Module MUX. Because the BRAM is realized inside of the FPGA fabric as a hard IP core, it is able to handle read and write operations in one clock cycle [1, p. 12]. If a high routing delay of the output is expected, an optional output register stage that adds one additional clock cycle to reading operations is o ered to relax the timing constraints. It is enabled in this design which increases the read latency to two clock cycles (write operations are still handled in one clock cycle). Memory accesses in the BRAM Domain without additional delay set are therefore equivalent to memory accesses in the Register Domain with a set delay of 10ns = 2 T (with T denoting the period of the clock signal). A total of 131072

memory sections (the reason for this number is explained at the end of this chapter) are implemented with the size of

$$sectsize = \frac{memsize}{\#sect} = \frac{512\,MiB}{131072} = 4\,KiB = 4096\,B\,: \tag{4.5}$$

each. The BRAM is inferred by using the Block Memory Generator IP from Xilinx (BRAM Generator henceforth) that allows the con guration of various settings like the memory width and depth, the usage output registers or the type of memory. The available two memory types are single port and dual port random access memories.[5] Single port memories have to handle read and write operations sequentially. The address line is shared and a write-enable-signal switches between reading and writing mode. Dual port memories allow simultaneous read and write operations on separate ports. Port A handles write accesses whereas port B is responsible for reading accesses. Each port has their own address line, relevant control signal lines and clock signal. The BRAM Module makes use of the dual port BRAM con guration, the reason for this will be explained later in this chapter.
As in the Register Module, the AXI master port GP0 AXI of the PS is used to address the memory structure and set the latency values of the sections { only this time no dedicated registers are addressed but the BRAM. The address space 0x40000000 to 0x4007FFFF is reserved for rlat values, 0x41000000 to 0x4107FFFF for wlat values. The addresses for the latency values of section $i \in \{0, \ldots, 131071\}$ are given by

$$\begin{aligned} rlat\_addr_i &= 0x40000000+(4i) \\ wlat\_addr_i &= 0x41000000+(4i)\,: \end{aligned} \tag{4.6}$$

Xilinx o ers an AXI slave IP called AXI BRAM Controller that enables AXI compliant access to the BRAM and is used here to connect the GP0 AXI master interface of the PS to the BRAM via the controllers. This allows the modi cation of the BRAM content, however, reading the latency values of the current PL-DRAM memory access for LatGen requires additional logic. This logic controls the access to the BRAM Generator and is called Access Controller in Figure 4.8, that shows the block design of the BRAM Module. All architectural elements are realized twice: once for rlat (top) and once for wlat (bottom). This is also the reason why rlat and wlat values have their own address spaces.

[5] Actually, the random access memories can also be con gured to operate like read-only memories, the contents of which are loaded from an initialization le.

Figure 4.8: BRAM Module Block Diagram

If the BRAM needs to be accessed by the PS to update the section latencies, the signals of the AXI BRAM Controller must be forwarded to the BRAM Generator. At any other time, the system may try to access the emulated NVMM which is only indicated by a changing address in the GP1 AXI interface.

The AXI BRAM Controller asserts its output signal bram_en if the PS wants to issue a read or write access to the BRAM via GP0 AXI and deasserts it after finishing the transaction. This fact is used by the Access Controller to multiplex the right signals to the BRAM as shown in Listing 4.5.

Listing 4.5: Access Controller Logic

```
1 always@(*)
2 begin
3     if(bram_en)
4     begin
5         // AXI_GP0 issued a transaction
6         addr_o  = gp0_addr;
7         data_o  = gp0_wrdata;
```

```
8           wr_en_o = gp0_wr_en;
9       end else
10      begin
11          // Forward NVMM transaction address
12          addr_o  = adjusted_nvmm_access_addr;
13          data_o  = 32'b0;
14          wr_en_o = 4'b0;
15      end
16  end
```

Once all section latencies are set by the user, bram_en will stay deasserted and the Access Controller forwards the adjusted nvmm_access_addr to the BRAM and sets the other signals to zero. For every change of the NVMM access address an adjusted version of it is passed to the BRAM directly, where the appropriate latency value is output after two clock cycles. The original NVMM access address must be adjusted, such that the address for the latency value of the section in which the original address is located, is created. With a section size of 4096 B (cf. 4.6), each section has an address range of 0x1000 addresses (0x1000 = 4096). The adjustment of the address works similar to the procedure shown in the example 4.2 for the Register Module: the 30th bit of the original NVMM address is omitted because the BRAM Domain is only half the size of the PL-DRAM. Since $\log_2(4096) = 12$, only the 12 low-order bits will change when the address of a different memory request stays in the same memory section. So checking the remaining 17 high-order bits of the address will yield the number of the accessed memory section, starting at zero. Lastly, to obtain the address from the section number, it has to be multiplied by four, which is done by left-shifting the address twice (cf. Listing 4.6). The high-order bits are filled with zeros because the BRAM generator expects an address with a length of 32 bit.

Listing 4.6: NVMM Address Conversion

```
1  adjusted_nvmm_access_addr =
2      {15'b0, nvmm_access_address[28:12]} << 2;
```

The way the Access Controller works (connecting a single address line to both address ports of the BRAM) means that the advantage of simultaneously reads and writes of a dual port BRAM configuration is lost. The reason a dual port BRAM configuration was used is because the GP0 AXI and memory clock domains are asynchronous. In a single port configuration one of the clock signals would have to be chosen as the BRAM clock which could lead to errors in the other clock domain due to phase-shifted clock signals. The GP0 AXI domain is responsible for writing new values to the BRAM whereas the memory clock domain only reads from it. This is exploited by the dual port configuration by connecting the GP0 AXI clock to port A (writing port) and the memory clock to port B (reading port). This way writing new latency values is guaranteed to succeed and the

section latencies are guaranteed to arrive at LatGen in exactly two clock cycles of the memory clock.
A remaining problem is that a user cannot reliably check the content of the BRAM and readbacks from write accesses may also show incorrect data, since reads from the GP0AXI domain are not guaranteed to succeed. This problem is solved by setting the minimum read latency of the AXI BRAM Controller to three and implementing Delayed Registers (cf. Figure 4.8). The Delayed Registers are clocked by the memory clock. As mentioned, the valid read data appears on the output of the BRAM after two clock cycles in this clock domain. As soon as the bram_en signal is asserted, the Delayed Registers wait for two clock cycles before reading the value on their inputs and latching them. The AXI BRAM Controller waits one more clock cycle in the GP0AXI clock domain to ensure that even in the worst case phase alignment of both clock signals, valid data can be read from the Delayed Registers.
The BRAM read latency of two clock cycles has to be incorporated in LatGen aswell. The NVMM read and write addresses are connected to LatGen and the Verilog description is extended as shown for read accesses in Listing 4.7 (analogous for write accesses).

Listing 4.7: Introducing Delay Variables to LatGen

```
always@(*)
begin
    if(nvmm_rdrequest_addr < 30'h2000_0000)
    begin
        // Register Domain access
        delay_ar = 32'd0;
        zero_ar  = (rlat == 32'h0);
    end else
    begin
        // BRAM Domain access
        delay_ar = 32'd2;
        zero_ar  = 1'b0;
    end
end
```

The values of zero_ar and zero_aw are explicitly set to zero if the memory access is within the BRAM Domain. New values delay_ar and delay_aw are introduced to delay those accesses by two clock cycles. Accesses within the Register Domain will not be delayed any further because for those delay_ar and delay_aw are set to zero. The if statement that decides whether ready_ar or ready_aw has to be asserted is extended by also comparing the counter value against delay_ar or delay_aw (cf. Listing 4.8 as an example for read accesses).

Listing 4.8: Additional Check for Assertion of the Ready Signal

```
1 if(busy_ar & cnt_ar >= rlat & cnt_ar >= delay_ar)
2 begin
3     busy_ar  <= 1'b0;
4     ready_ar <= 1'b1;
5 end
```

The implementation of 131072 memory sections requires approximately 47 % of the available BRAM primitives. To get memory sections of equal size, the number of implemented sections has to be a divider of 512 MiB (as one byte is the smallest addressable unit), or in other words a multiple of 2 (smaller or equal to 512 MiB). Hence, the next bigger section size would be 262144. The BRAM primitives are equally distributed on the FPGA and a design with more than 131072 fails the timing requirements due to long routes that produce high net delays, again with the implementation strategy to run timing-driven optimizations.

4.2.3 Module MUX

In their calculations to determine the accessed memory section, both the Register Module and the BRAM Module omit the 30th bit of the NVMM read and write addresses. These modules are responsible for 512 MiB of the 1 GiB NVMM memory each, so 29 bits are sufficient to address every byte in their domain since $\log_2(512\text{Mi}) = 29$. As a consequence, the modules cannot determine whether they are responsible for the current NVMM access or not. Both modules output latency values and the Module MUX has to determine which output to pass to LatGen. This is done by checking the value of the NVMM address: addresses greater than or equal to 0x20000000 indicate that the BRAM Domain is addressed, values below this boundary are intended for the Register Domain. This behaviour can be realized by two 2-to-1 multiplexers, whose select signals are the 30th bit of either the NVMM read or write address. Checking the value of this bit is sufficient since values greater than or equal to 0x20000000 require the 30th bit to be set. The Module MUX Verilog description without input and output port declaration is presented in Listing 4.9.

Listing 4.9: Module MUX

```
1 assign rlat = nvmm_rdrequest_addr[29] ?
2     BRAM_rlat : register_rlat;
3 assign wlat = nvmm_wrrequest_addr[29] ?
4     BRAM_wlat : register_wlat;
```

4.2.4 Setting the Latencies

To set the read and write latencies in the original emulator, a program called latset is provided. latset uses the Unix system call mmap to map parts of the device file /dev/mem into its address space. /dev/mem contains an image of the main memory of the system where the byte addresses are interpreted as physical memory addresses. It can be used to examine or change the content of the main memory [4]. A sufficiently large portion of this file gets mapped into the address space of latset by the program itself to access the physical addresses 0x43C00008 (rlat register) and 0x43C0000C (wlat register). The arguments of the program determine the latencies in nanoseconds, so the usage is as follows: `latset <rlat> <wlat>`. Since the counter in the delay generation logic dictates the number of clock cycles by which the memory request shall be delayed, the arguments are divided by five, as the period of the memory clock is 5 ns. This also means that arguments that are not multiples of five are rounded down.
The same approach of using /dev/mem and mmap is adapted for the extended emulator. Since the amount of sections has changed, latset has to access different addresses (cf. Equations 4.3, 4.4 and 4.6). Moreover, because the smallest realizable additional latency is 10 ns, smaller arguments are adjusted accordingly. The fact that a value of 1 for rlat or wlat already produces a delay of 10 ns (cf. Figure 4.4) is taken into account. If for example a delay of 15 ns is desired, latset will store the number 2 instead of 3. Apart from that, the modifications of latset only affect the usability. For example, it is possible to provide an address range in order to set the latencies of all affected sections. Table 4.1 presents the usage of the extended version of latset.

Table 4.1: Usage of latset

Parameter Options	Description
`latset <rlat> <wlat>`	Sets all section latencies to values provided by <rlat> and <wlat>.
`latset -reg <rlat> <wlat>`	Sets all section latencies of the Register Domain to values provided by <rlat> and <wlat>.
`latset -bram <rlat> <wlat>`	Sets all section latencies of the BRAM Domain to values provided by <rlat> and <wlat>.
`latset -reg --read <sect>`	Reads the latencies of the section provided by <sect> of the Register Domain.

`latset -bram --read <sect>`	Reads the latencies of the section provided by <sect> of the BRAM Domain.
`latset -reg <sect> <rlat> <wlat>`	Sets the latencies of section <sect> of the Register Domain to values provided by <rlat> and <wlat>.
`latset -bram <sect> <rlat> <wlat>`	Sets the latencies of section <sect> of the BRAM Domain to values provided by <rlat> and <wlat>.
`latset -range <from> <to> <rlat> <wlat>`	Sets the latencies of sections in address range from <from> to <to> to values provided by <rlat> and <wlat>.

5 Evaluation

5.1 Resource Utilization

When a nished design is successfully implemented into the FPGA fabric, Vivado o ers reports of various circuit characteristics like for example resource utilization, estimated power consumption or clock interaction for design evaluation. Chapter 4 revealed that the e ects of the design extension on signal timing were the main concern of the design, which ultimately set an upper limit for the size of the design and thus the number of implemented memory sections. The relatively large number of FPGA resources of the Zynq-7000 XC7Z045 SoC is the reason why the resource utilization was not as signi cant and therefore not mentioned as of yet. Nevertheless, while converging to the design limit by enlarging the design, i.e. increasing the register count in the Register Module and utilizing more BRAM primitives in the BRAM Module, the e ect on resource utilization could be recorded[1]. Figure 5.1 shows the percentage utilization of the LUTs, the LUTRAM (LUTs implemented as RAM within a CLB) and ip- ops (FF) depending on the number of implemented sections in the Register Module. The number and distribution of used resources is indeed dependant on the HDL description of the design, but the used implementation strategy has a great impact aswell [9]. Every measurement in this chapter uses the same implementation strategy in Vivado to guarantee comparable results: Performance_NetDelay_high. The BRAM Module is unmodi ed in the shown measurement series and con gured to implement the upper limit of 131072 sections. The utilization of the BRAM is constantly at 47.16 % and excluded in this gure for readability reasons. With an constant value of 6.26 %, the LUTRAM utilization is not in uenced by the number of sections. The measurements of the LUT and ip- op utilization however indicate a linear increase with an increasing number of sections. Since changing the emulator design and running an implementation to estimate the resource utilization can be time consuming, linear regression is used to calculate functions that provide the approximate number of used resources (LUT and FF) for the number of sections that were not sampled. An

[1] To conduct the measurements of the resource utilization by section count for both the Register Module and BRAM Module, the supported section count of the modules has to be adjusted in both designs. Since these designs have to be created anyway for the measurements, they are provided with this thesis. Please note that these designs were not tested and that latset will not work for them. The contents of the rlat and wlat registers, respectively BRAM registers, can still be changed by manually accessing /dev/mem.

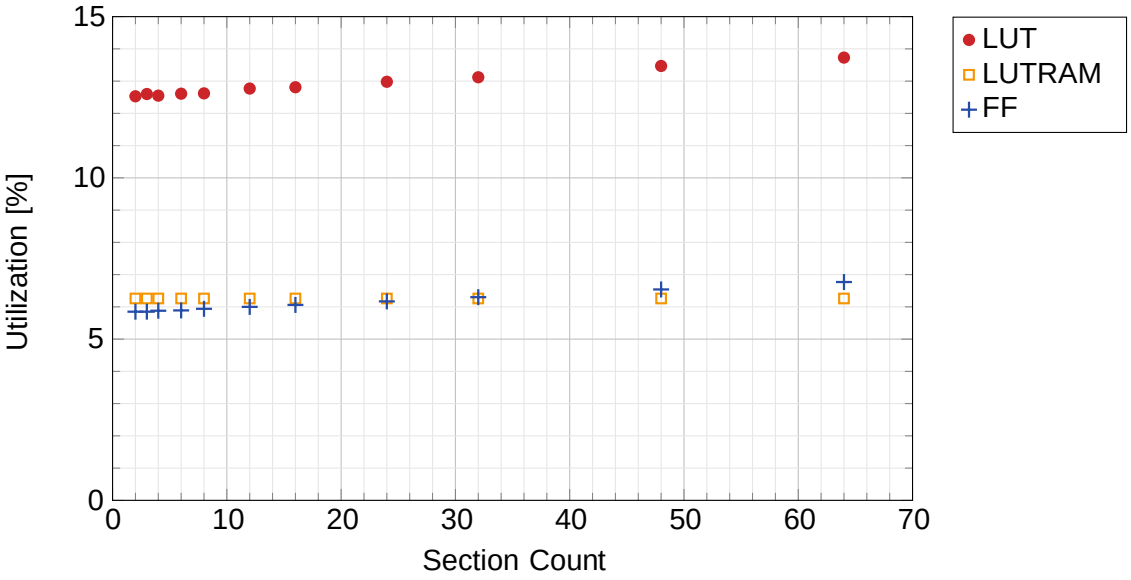

Figure 5.1: FPGA Utilization by Section Count for the Register Module

simple linear regression model $y = c + dx$ is determined using the least-squares approach as follows:

$$y = c + dx = \underbrace{\bar{y} - \frac{s_{xy}}{s_x^2}\bar{x}}_{c} + \underbrace{\frac{s_{xy}}{s_x^2}}_{d} x \tag{5.1}$$

with covariance

$$s_{xy} = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y}) \tag{5.2}$$

and variance

$$s_x^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2 \tag{5.3}$$

for all n measured data pairs $(x_i, y_i),\ i \in \{1, \ldots, n\}$.

Using the data pairs of the LUT and FF measurements, the regression yields the utilization models

$$\begin{aligned} \text{util}_{\text{LUT}_{\text{rm}}}(x) &\quad 0{:}0002x + 0{:}125 \\ \text{util}_{\text{FF}_{\text{rm}}}(x) &\quad 0{:}00015x + 0{:}05814 \end{aligned}$$

for the estimated percentage utilization of the LUTs and FFs by section number x realized by the Register Module (denoted by rm). The Pearson correlation coe cient

$$r_{xy} = \frac{s_{xy}}{s_x s_y} \tag{5.4}$$

yields $r_{xy}^{LUT} = 0{:}99787$ for the LUT model and $r_{xy}^{FF} = 0{:}99971$ for the FF model. These models are applicable for the section range 2 to 64, the validity for a section number above 64 is uncertain because Vivado might try to duplicate logic to shorten route lengths [17] since meeting the timing constraints was not possible beyond 64 sections with a fixed BRAM section number of 131072.

The percentage resource utilization with an increasing number of sections was also recorded for the BRAM Module (denoted by bm in the equations) and is presented in Figure 5.2. Especially for the section numbers smaller than 4098 it is visible in the figure that the BRAM utilization is constant for a certain section range before abruptly rising. The reason for that is the fact that one of the 545 available BRAM primitives is capable of storing 4096 words with a width of 32 bit [1].[2] Because the number of realized BRAM sections is identical to the number of words to be saved and the size of an atomic BRAM block (i.e. one primitive) is known, the utilization of the BRAM can be determined exactly instead of approximately. The realization of one section requires storage for two 32 bit words, hence one BRAM primitive is capable of storing enough data words for $4096\,B = 2 \cdot 4\,B = 512$ sections. One BRAM primitive that is already used in the MIG is taken into account in the calculation. The percentage utilization of the BRAM for a section number of x is therefore given by

$$\text{util}_{\text{BRAM}_{bm}}(x) = \frac{1}{545} \left\lfloor \frac{x}{512} \right\rfloor + 1 \quad :$$

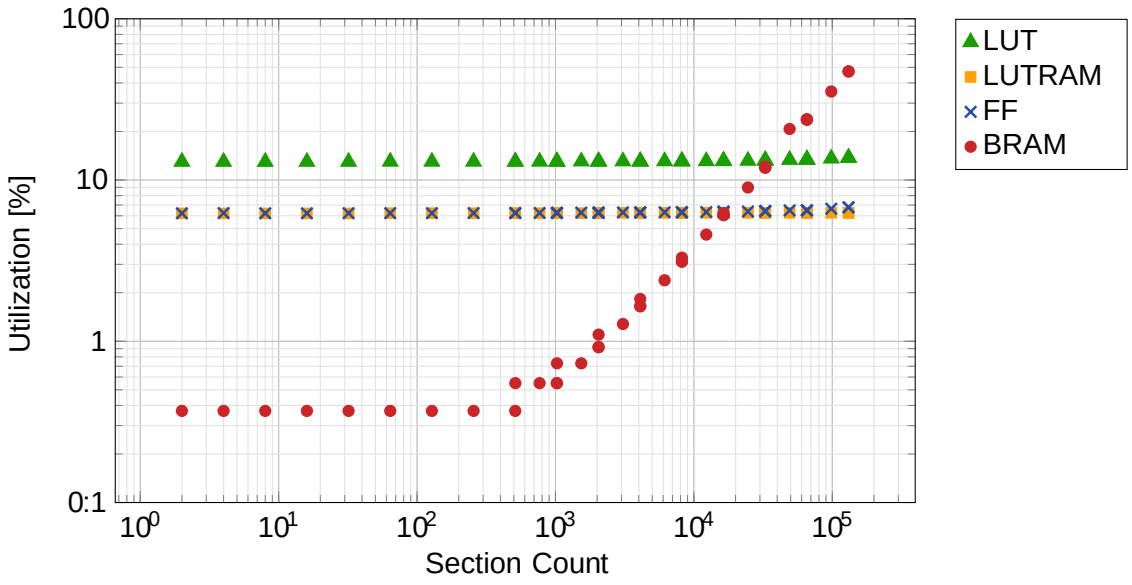

Figure 5.2: FPGA Utilization by Section Count for the BRAM Module

[2] The exact number of storable words for each primitive varies with different BRAM configurations and word widths [1]. The denoted value is true for the configuration used in the design of the extended emulator.

The measurements accurately capture the limits at which the LUTRAM utilization increased abrubtly, so their utilization can be determined precisely aswell in the range from 2 to 131072 by

$$\text{util}_{\text{LUTRAM}_{bm}}(x) = \begin{cases} 8 & \\ 6{:}21 & \text{for } 2 \quad i \quad 32 \\ 6{:}23 & \text{for } 33 \quad i \quad 2048 \\ 6{:}26 & \text{for } 2049 \quad i \quad 131072 \end{cases}$$

The utilization by section number of the two remaining resources for the BRAM Module, LUTs and FFs, again show a linear correlation. Equation 5.1 yields the approximation functions

$$\begin{aligned} \text{util}_{\text{LUT}_{bm}}(x) &\quad 5{:}74 \;\; 10^{-8} x + 0{:}13021 \\ \text{util}_{\text{FF}_{bm}}(x) &\quad 3{:}9045 \;\; 10^{-7} x + 0{:}6261 \end{aligned}$$

with Pearson correlation coe cients (cf. Equation 5.4) $r_{xy}^{LUT} = 0{:}98713$ and $r_{xy}^{FF} = 0{:}98045$.

Ultimately, the resource utilization of the extended design is compared to that of the original design in Figure 5.3.

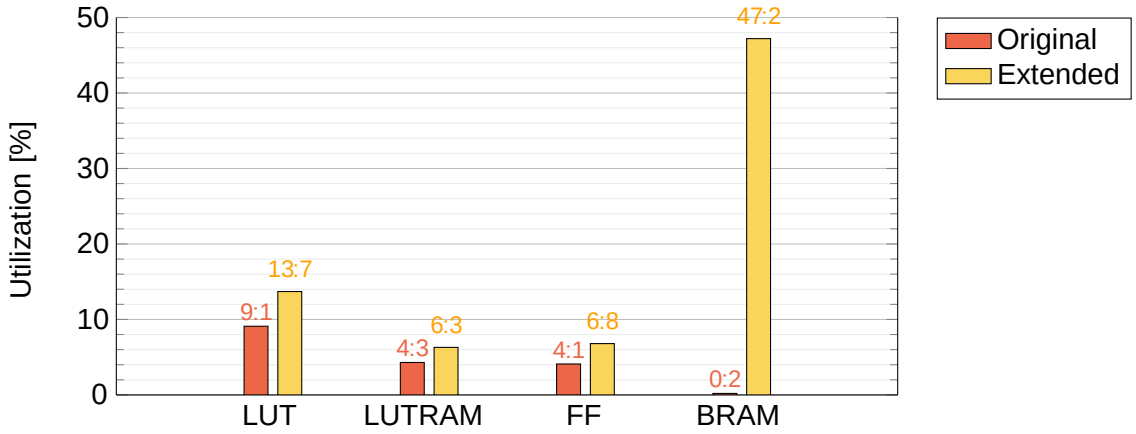

Figure 5.3: Comparison of FPGA Resource Utilization

5.2 Impact on Memory Latency

The e ects of the extended design in respect to the memory latencies without additional latency set have already been addressed in Chapter 4. Rather, it was presented that the implemented design extensions resulted from considerations regarding the timing behavior of the memory accesses. The timing constraints that were set for the design ensure that the route delay of a signal from one clocked component to another must not exceed the

length of one clock cycle. Since the extended design passes the timing analysis of the implementation tool, the effects of the design changes can easily be determined within the scope of the delay generation logic itself (i.e. LatSet and LatGen in the original emulator, Register Module/BRAM Module plus Module MUX and LatGen in the extended emulator). Chapter 4.2 describes how the emulator behaviour changes with the executed extensions: memory accesses to the Register Domain with rlat and wlat set to zero will behave the same way as they do in the original emulator, accesses to the BRAM Domain on the other hand will experience a delay because LatGen has to wait two clock cycles before rlat and wlat coming from the BRAM are present. Contrary to what one might first expect, this delay is three and not two clock cycles. As soon as the latency values are present to LatGen, it detects that their value is zero and the AXI handshake has to be forwarded immediately. The ready signal is asserted in the next clock cycle which triggers the forwarding of the handshake signals. It takes another clock signal at the receivers side to detect those handshake signals, ultimately resulting in a delay of three additional clock cycles in the BRAM Domain (see also Figure 4.4 where this behaviour was already covered).

Notwithstanding the fact that the timing behaviour can be determined precisely for the delay generation logic, the performance of the system as a whole might still be affected. So far, the propagation delay of the memory request between the PS and the delay generation logic (where the AXI SmartConnect IP can be found, cf. Figure 4.3) was not considered. Whether the duration of the AXI transactions that send the memory requests is affected by the design extensions or not is examined hereafter.

A series of measurements are performed on three designs: the original emulator, the extended emulator and an adjusted design where the delay generation logic is removed. An Xilinx IP core called ILA (Integrated Logic Analyzer) enables to capture signals of the FPGA that follow a predetermined trigger condition. In this case, the address lines coming from the PS, the address lines directly at the MIG and the BVALID signal sent by the MIG are captured using the ILA core. As the trigger condition a change of the memory request address coming directly from the PS is chosen. To determine the latencies on read accesses, randomized memory locations are read from on all three designs. A further distinction is made for the extended design, where the Register Domain and BRAM Domain are considered individually. The number of clock cycles between the change of the read address coming from the PS and the moment at that the output of the MIG changes is captured and presented in Figure 5.4.

The measurement results indicate that the impact on reading performance is indeed an additional delay of three clock cycles in the BRAM Domain of the extended emulator whereas the Register Domain behaves similar to the original emulator that shows no significant derivation from a design without a delay generation logic. It is also visible in

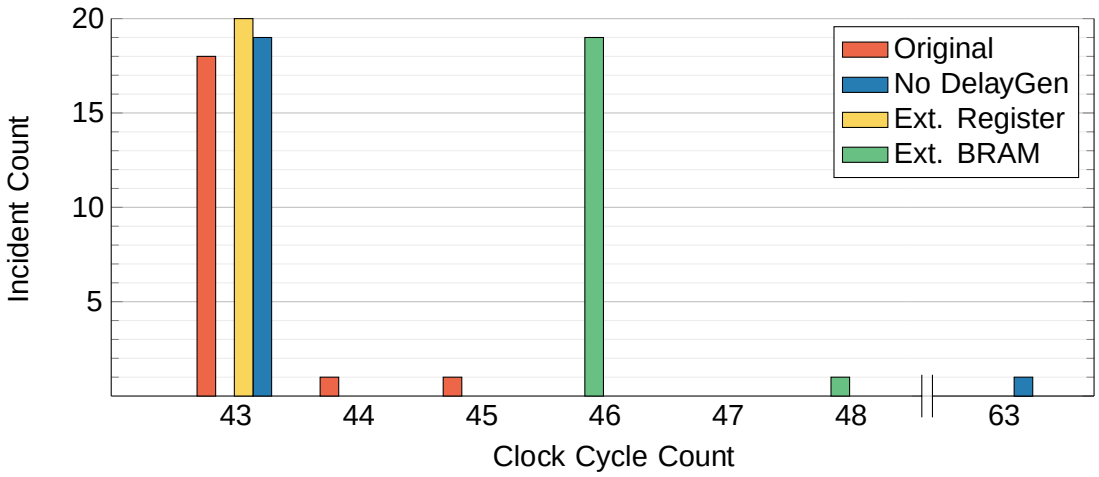

Figure 5.4: Read Access Latency Samples

the diagram that the read performance is not a xed number but may vary depending on signal line utilization.

For write accesses, again randomized locations are written to using random 32 bit words. This time, the elapsed clock cycles between the address coming from the PS and the BVALIDsignal of the MIG are measured. This outcome is depicted in Figure 5.5. Despite the delay of three clock cycles before the handshake signals appears at the MIG in the BRAM Domain, the diagram shows that most of the time the MIG asserts BVALIDonly one clock cycle later compared to every other case. The write address and write data are already present at the MIG and independent from the delay generation logic. Since it requires many connections to the ILA core to directly tap the content of the DRAM, a more precise measurement of when the data word is present in the memory registers is hard to realize, so this measurement has to rely on the validation of the write access by

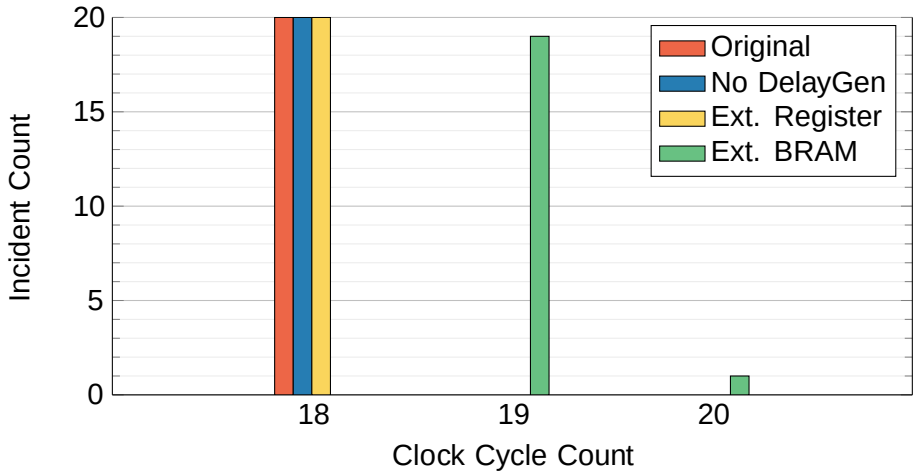

Figure 5.5: Write Access Latency Samples

the MIG. Whether or not the AXI logic is responsible for this (at least) one clock cycle delay of the valid signal can only be guessed at this point. It can be stated with high certainty however that write accesses to the BRAM Domain are validated at a later time by the MIG.
Using an ILA core for the measurements of potential effects on signal behaviour has the advantage that the effects are visible in the nanosecond scope. The ILA sees what has actually happened in the circuit during the captured time period. A more precise measurement is not possible, but this comes at a cost of measurement effort. The memory accesses have to be carried out manually before the signals can be observed in Vivado where then the number of elapsed clock cycles between the trigger and the expected event needs to be counted.

6 Conclusion

6.1 Summary of Results

The effects on a system due to specific memory access latencies of non volatile memories used as main memory can be investigated with the emulator provided by Yu Omori et al. The emulator is limited to a single main memory type; due to the specific characteristics of different NVMs however, hybrid systems that make use of several NVMs as main memory are subjects of research as well. This thesis extends the given emulator by the function to mimic the access latency behaviour of more than one NVM type. In order to be able to emulate the largest possible amount of system configurations and offer the flexibility to experiment with different configurations, the extension allows for a fine granularity of memory sections within the main memory if required, each of which corresponds to one NVM type. The access latencies for specified memory regions or the whole memory can be set by a program provided with this thesis.

Since the physical boundaries of a realizable section count and therefore emulatable NVMs are reached relatively fast in an approach to extend the design idea of the original emulator, the DRAM that emulates the NVMs is divided into two domains. The access latency emulation of the first half of the DRAM behaves similar to original emulator design. The number of emulatable NVMs is limited to 64 in this domain. The second half of the DRAM enables up to 131072 different NVM types with the drawback of overall slower access times, which increases the viable minimum access latency for this domain. The delay generation logic design of this domain takes a different approach compared to the original emulator by utilizing an hitherto almost unused resource (the BRAM) of the FPGA, in which the emulator is implemented in.

The evaluation of the extended design suggests that the impact on the system performance caused by the extensions are as expected. The design environment of the FPGA ensures that the described hardware works as stated (the signal timing is of particular interest here) within the FPGA, nonetheless the hardware changes can have an impact on the system as a whole (i.a. because the on-chip communication protocol allows deviations in the timing of signal exchanges). It should be noted however that the measurements had to be conducted manually and therefore the sample size is relatively small. Further research needs to be done to increase the significance of the findings.

6.2 Future Outlook

Additionally to the coarse-grained delay injection model, Yu Omori et al. implemented a ne-grained model where the additional memory latencies are generated within the DRAM itself [16]. This model could also be extended in future work in order to enable the emulation of hybrid systems.
Moreover, regular use of the emulator could reveal frequent con gurations of the emulator regarding the count of emulated NVMs. It is also possible that the division of the DRAM is more of a hindrance than a bene t. Future work could gather the most common use cases so that adapted versions of the emulator could be made available.

Appendix

FPGA Utilization (%) Measurement Data for Figure 5.1

Section Count	LUT	LUTRAM	FF	BRAM
2	12.53	6.26	5.85	47.16
3	12.60	6.26	5.85	47.16
4	12.55	6.26	5.88	47.16
6	12.61	6.26	5.89	47.16
8	12.62	6.26	5.94	47.16
12	12.77	6.26	6.00	47.16
16	12.81	6.26	6.06	47.16
24	12.98	6.26	6.17	47.16
32	13.12	6.26	6.30	47.16
48	13.47	6.26	6.54	47.16
64	13.73	6.26	6.77	47.16

FPGA Utilization (%) Measurement Data for Figure 5.2

Section Count	LUT	LUTRAM	FF	BRAM
2	12.98	6.21	6.22	0.37
4	12.98	6.21	6.24	0.37
8	12.97	6.21	6.22	0.37
16	12.98	6.21	6.22	0.37
32	12.98	6.21	6.23	0.37
64	13.00	6.23	6.23	0.37
128	13.00	6.23	6.23	0.37
256	13.00	6.23	6.24	0.37
512	13.00	6.23	6.24	0.37
513	13.01	6.23	6.25	0.55
768	13.01	6.23	6.25	0.55
1023	13.01	6.23	6.25	0.55

1024	13.01	6.23	6.24	0.55
1025	13.05	6.23	6.28	0.73
1536	13.05	6.23	6.28	0.73
2047	13.02	6.23	6.25	0.92
2048	13.02	6.23	6.25	0.92
2049	13.08	6.26	6.31	1.10
3072	13.07	6.26	6.30	1.28
4095	13.04	6.26	6.28	1.65
4096	13.04	6.26	6.31	1.65
4097	13.08	6.26	6.31	1.83
6144	13.08	6.26	6.31	2.39
8191	13.07	6.26	6.31	3.12
8192	13.07	6.26	6.31	3.12
8193	13.10	6.26	6.33	3.30
12288	13.10	6.26	6.33	4.59
16383	13.12	6.26	6.34	6.06
16384	13.12	6.26	6.34	6.06
16385	13.18	6.26	6.37	6.24
24576	13.17	6.26	6.38	8.99
32767	13.21	6.26	6.41	11.93
32768	13.21	6.26	6.41	11.93
32769	13.30	6.26	6.43	12.11
49151	13.34	6.26	6.48	20.73
65535	13.37	6.26	6.48	23.67
65536	13.37	6.26	6.48	23.67
65537	13.45	6.26	6.50	23.85
98304	13.60	6.26	6.64	35.41
131071	13.74	6.26	6.77	47.16
131072	13.74	6.26	6.77	47.16

Access Latency (Clock Cycles) Measurement Data for Figures 5.4 and 5.5

Original		No DelayGen		Ext. Register		Ext. BRAM	
read	write	read	write	read	write	read	write
43	18	43	18	43	18	46	19
43	18	43	18	43	18	46	19
43	18	43	18	43	18	46	19
43	18	43	18	43	18	46	20
43	18	43	18	43	18	46	19
43	18	63	18	43	18	46	19
43	18	43	18	43	18	46	19
43	18	43	18	43	18	48	19
43	18	43	18	43	18	46	19
43	18	43	18	43	18	46	19
43	18	43	18	43	18	46	19
43	18	43	18	43	18	46	19
45	18	43	18	43	18	46	19
43	18	43	18	43	18	46	19
43	18	43	18	43	18	46	19
43	18	43	18	43	18	46	19
44	18	43	18	43	18	46	19
43	18	43	18	43	18	46	19
43	18	43	18	43	18	46	19
43	18	43	18	43	18	46	19
43	18	43	18	43	18	46	19

List of Figures

List of Source Codes

Bibliography

[1] 7 Series FPGAs Memory Resources User Guide (UG473). Version 1.14. https://www.xilinx.com/support/documentation/user_guides/ug473_7Series_Memory_Resources.pdf.

[2] Altera FPGA Architecture White Paper (WP-01003). Version 1.0. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/wp/wp-01003.pdf.

[3] Designing Large Multiplexers (UG002). Version 1.3. http://ebook.pldworld.com/_semiconductors/Xilinx/DataSource%20CD-ROM/Rev.6%20(Q1-2002)/userguides/V2_handbook/ug002_ch2_multiplexers.pdf.

[4] Linux manual page for mem(4). https://man7.org/linux/man-pages/man4/mem.4.html.

[5] Memory Interface Solutions User Guide (UG086). Version 3.6. https://www.xilinx.com/support/documentation/ip_documentation/ug086.pdf.

[6] Multiplexer Design Techniques for Datapath Performance with Minimized Routing Resources (XAPP522). Version 1.2. https://www.xilinx.com/support/documentation/application_notes/xapp522-mux-design-techniques.pdf.

[7] Spartan-7 FPGAs: Meeting the Cost-Sensitive Market Requirements (WP483). Version 1.1. https://www.xilinx.com/support/documentation/white_papers/wp483-spartan-7-intro.pdf.

[8] UltraScale Architecture Con gurable Logic Block User Guide (UG574). Version 1.5. https://www.xilinx.com/support/documentation/user_guides/ug574-ultrascale-clb.pdf.

[9] Vivado Design Suite User Guide { Implementation (UG904). Version 2018.3. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_3/ug904-vivado-implementation.pdf.

[10] Vivado Design Suite User Guide { Synthesis (UG901). Version 2017.1. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_1/ug901-vivado-synthesis.pdf.

[11] Zynq-7000 SoC Data Sheet: Overview (DS190). Version 1.11.1. `https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf`.

[12] Y. Bao, M. Chen, Y. Ruan, L. Liu, J. Fan, Q. Yuan, B. Song, and J. Xu. Hmtt: A platform independent full-system memory trace monitoring system. In *Proceedings of the 2008 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS 08, page 229240, New York, NY, USA, 2008. Association for Computing Machinery.

[13] I. Grout. *Digital Systems Design with FPGAs and CPLDs*. Newnes, USA, 2008.

[14] C. Hakert. Memory access analysis and endurance leveling approaches for non-volatile working memory systems. Master's thesis, 2019.

[15] T. Lee and S. Yoo. An fpga-based platform for non volatile memory emulation. In *2017 IEEE 6th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, pages 1{4, 2017.

[16] Y. Omori and K. Kimura. Performance evaluation on nvmm emulator employing negrain delay injection. In *2019 IEEE Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, pages 1{6, 2019.

[17] H. Patel. Synthesis and implementation strategies to accelerate design performance. *Xilinx White Paper*, 229, 2005.

[18] M. Poremba, T. Zhang, and Y. Xie. Nvmain 2.0: A user-friendly memory simulator to model (non-)volatile memory systems. *IEEE Computer Architecture Letters*, 14(2):140{143, 2015.

[19] Y. Zhang and S. Swanson. A study of application performance with non-volatile main memory. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1{10, 2015.

[20] P. Zhou, B. Zhao, J. Yang, and Y. Zhang. A durable and energy e cient main memory using phase change memory technology. In *ISCA '09*, 2009.

Eidesstattliche Versicherung
(Affidavit)

Morczinek, Dennis — 190883

Name, Vorname
(Last name, first name)

Matrikelnr.
(Enrollment number)

Ich versichere hiermit an Eides statt, dass ich die vorliegende Bachelorarbeit/~~Masterarbeit~~* mit dem folgenden Titel selbstständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate kenntlich gemacht. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

I declare in lieu of oath that I have completed the present Bachelor's/~~Master's~~* thesis with the following title independently and without any unauthorized assistance. I have not used any other sources or aids than the ones listed and have documented quotations and paraphrases as such. The thesis in its current or similar version has not been submitted to an auditing institution.

Titel der Bachelor-/~~Master~~arbeit*:
(Title of the Bachelor's/ ~~Master's~~* thesis):

Configurable FPGA-based Access Latency Emulation for Non-Volatile Main Memory

*Nichtzutreffendes bitte streichen
(Please choose the appropriate)

Dortmund, 18.07.2020

Ort, Datum
(Place, date)

Unterschrift
(Signature)

Belehrung:
Wer vorsätzlich gegen eine die Täuschung über Prüfungsleistungen betreffende Regelung einer Hochschulprüfungsordnung verstößt, handelt ordnungswidrig. Die Ordnungswidrigkeit kann mit einer Geldbuße von bis zu 50.000,00 € geahndet werden. Zuständige Verwaltungsbehörde für die Verfolgung und Ahndung von Ordnungswidrigkeiten ist der Kanzler/die Kanzlerin der Technischen Universität Dortmund. Im Falle eines mehrfachen oder sonstigen schwerwiegenden Täuschungsversuches kann der Prüfling zudem exmatrikuliert werden. (§ 63 Abs. 5 Hochschulgesetz - HG -).

Die Abgabe einer falschen Versicherung an Eides statt wird mit Freiheitsstrafe bis zu 3 Jahren oder mit Geldstrafe bestraft.

Die Technische Universität Dortmund wird ggf. elektronische Vergleichswerkzeuge (wie z.B. die Software „turnitin") zur Überprüfung von Ordnungswidrigkeiten in Prüfungsverfahren nutzen.

Die oben stehende Belehrung habe ich zur Kenntnis genommen:

Official notification:
Any person who intentionally breaches any regulation of university examination regulations relating to deception in examination performance is acting improperly. This offense can be punished with a fine of up to €50,000.00. The competent administrative authority for the pursuit and prosecution of offenses of this type is the chancellor of TU Dortmund University. In the case of multiple or other serious attempts at deception, the examinee can also be unenrolled, section 63, subsection 5 of the North Rhine-Westphalia Higher Education Act (*Hochschulgesetz*).

The submission of a false affidavit will be punished with a prison sentence of up to three years or a fine.

As may be necessary, TU Dortmund will make use of electronic plagiarism-prevention tools (e.g. the "turnitin" service) in order to monitor violations during the examination procedures.

I have taken note of the above official notification:**

Dortmund, 18.07.2020

Ort, Datum
(Place, date)

Unterschrift
(Signature)

**Please be aware that solely the German version of the affidavit ("Eidesstattliche Versicherung") for the Bachelor's/ Master's thesis is the official and legally binding version.