UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola Tècnica Superior d'Enginyeria
de Telecomunicació de Barcelona

DESIGN OF AN AXI-SDRAM INTERFACE IP IN A RISC-V PROCESSOR

A Master's Thesis
Submitted to the Faculty of the
Escola Tècnica d'Enginyeria de Telecomunicació de Barcelona
Universitat Politècnica de Catalunya
by

Joan Marimon Illana

In partial fulfilment
of the requirements for the degree of
MASTER IN ELECTRONIC ENGINEERING

Advisor: Francesc Moll Echeto

Barcelona, May 2020

Abstract

PreDRAC is a RISC-V based SoC developed with the collaboration of the BSC, CIC-IPN, IMB-CNM (CSIC) and UPC. On its first version, sent to fabricate on May 2019, it used a custom interface to access main memory through an FPGA. Access to memory is critical to the performance of a processor and a AXI-SDRAM interface IP to be integrated into a future revision of the chip has been designed.

No specific area, power or performance constraints are defined for AXI-SDRAM interface as the first step is to obtain a functional design with the required verification setup to ensure its proper operation once fabricated on silicon. The design of the IP covers different aspects in the ASIC design flow: the initial RTL implementation, synthesis, verification at RTL and gate-level simulations and a final power analysis. Final results show that this IP can successfully be integrated with the preDRAC SoC, replacing the custom interface, and obtaining better performance. However, the AXI-SDRAM interface IP can be further improved both in terms of performance and power.

Acknowledgements

I would like to express my gratitude to my master thesis advisor Francesc Moll for his support and to give me the opportunity to participate in this project.

I would like to make my gratiture extensive to all people from BSC, CNM and UPC who collaborated in the preDRAC tapeout with who I had the pleasure to work with.

Last but not least, I want to thank my family who have supported and motivated me the whole time.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In May 2019, a RISC-V based processor, the preDRAC, was sent to tapeout [1]. PreDRAC is the first RISC-V processor from a Spanish academic institution. It was developed with the collaboration of Barcelona Supercomputing Center (BSC), Centro de Investigación en Computación del Instituto Politécnico Nacional de México (CIC-IPN), Institut de Microelectrònica de Barcelona (IMB-CNM(CSIC)) and Universitat Politècnica de Catalunya (UPC). This was a first step in the development of the DRAC project (Designing RISC-V based Accelerators for the next generation Computers).

The preDRAC includes a single RISC-V core as well as different levels of cache hierarchy and additional peripherals such as UART, JTAG, or an SD card controller. During initial tests, the preDRAC was implemented on an FPGA board, the Xilink Kintex KC705, and used the on-board DDR3 memory as the main memory. Memory was connected to the core through an AXI bus.

Memories such as DDR, DDR2, DDR3, and above work at high frequencies ($>$200 MHz). At these frequencies, wires can not be considered ideal and some conditioning is needed to ensure data is transmitted correctly. A physical interface with the right pin terminations is needed to ensure such conditioning. This physical interface was not available so it was not possible to include the main memory interface within the chip.

The solution in the preDRAC was making the AXI interface available outside the chip. From there, connect it to the FPGA to access the KC705 on-board DDR3 memory. Since the AXI interface had too many signals and there were not enough available pins, a custom interface was designed to split the AXI signals into multiple packets and send them to the FPGA. Then, inside the FPGA, reconstruct the original AXI request and send it to memory.

1.1 Motivation

Access to the main memory has a significant impact on the performance of a processor. The current setup, while it is functional, has several limitations in terms of performance. For the next chip, the custom interface will be replaced with an SDR SDRAM controller that will be integrated into the chip. Then, the main memory will be an SDR SDRAM placed on the same board as the chip and and the auxiliary FPGA board will not be required anymore.

SDR SDRAM work at lower frequencies than DDR, typically the maximum frequency ranges from 100 MHz to 166 MHz depending on the memory chip, with some recent models reaching up to 200 MHz. In this case, it does not require a specific memory interface as it uses standard CMOS digital output buffers.

The purpose of this project is to design and implement an AXI-SDRAM interface IP that will be used in a future DRAC chip. The main goal is to obtain a functional design and provide the setup required for the verification to ensure everything will work once built on silicon. The project involves different steps in the ASIC design flow, starting from the design of the RTL itself, the synthesis, and verifications both at an initial RTL level and after synthesis. Additionally, even if power consumption is not a major concern for this particular project, a power analysis is included since it should be taken into consideration as part of the verification process.

1.2 Project organization

This work is divided into several parts. Chapter 2 gives an overview of the relevant concepts involved in this project. First, section 2.1 gives a brief introduction to RISC-V that is followed with a description of the preDRAC chip. Section 2.2 starts with the basics of memory organization. Then, the main operation of an SDRAM is described to have a better understanding of which are the tasks that should be performed by a memory controller. After that, section 2.3 gives an introduction to the AXI protocol. Finally, chapter 2 ends with an overview of the different steps involved in the ASIC design required to go from an initial RTL to the final layout that can be tapeout.

Chapter 3 contains the actual implementation of the design. It starts with a description of the main blocks in the design and how they work. Then, section 3.2 explains the methodology used for the verification of the design, and section 3.4 describes the steps that have been done to perform a power estimation of the design.

Results are presented in chapter 4, which include the area and power results obtained after synthesis and power analysis as well as some performance evaluation.

Finally, the project concludes with chapter 5, which includes some conclusions and an overview of the future work to be done.

Chapter 2

Background

2.1 RISC-V

RISC-V is an instruction set architecture (ISA) created in 2010 at UC Berkeley [2], initially designed to support research and educational projects. It is defined as a base integer ISA with two main variants of 32 and 64 bits. Moreover, RISC-V is designed so the base ISA can be extended to provide support for additional instructions.

The main interest behind RISC-V is that it defines an open and free ISA standard, available for both academia and industry. Anyone can develop their hardware that supports this common standard to run the software that was ported or developed for it.

Many RISC-V based processors have been fabricated both in academic and industrial environments. A special focus will be put on the preDRAC processor.

2.1.1 PreDRAC processor

The preDRAC is a 64-bit single-core in-order processor that implements the 64-bit RV64IMA scalar RISC-V ISA. It is the first RISC-V based processor from an spanish academic institution developed in 2019 with the collaboration of the BSC, CIC-IPN, IMB-CNM and UPC. It is based on the 0.2 version of the lowRISC chip [3].

lowRISC is a non-profit organization with the goal to produce a fully open source SoC. The lowRISC chip is a SoC built on top of the Rocket Chip and adds support for additional peripherals in order to make a fully functional SoC that can be implemented on a FPGA and is able to boot Linux. Moreover, the lowRISC chip also provides the required setup to verify the basic functionality of the SoC with different simulation tools.

The Rocket Chip is built with the Rocket Chip generator, a tool developed by UC Berkeley to generate custom SoC based on RISC-V cores [4]. The generator has a collection of libraries that can be used to generate SoCs with different paramaters, for instance different number of cores or cache sizes.

Then the preDRAC makes use of the lowRISC infrastructure but replaces the Rocket core with the Lagarto core, which is a custom core based on the RISC-V ISA. The main blocks of the design are shown in figure 2.1. The design is split into two parts: the preDRAC chip itself (SoC), and an auxiliary FPGA board.

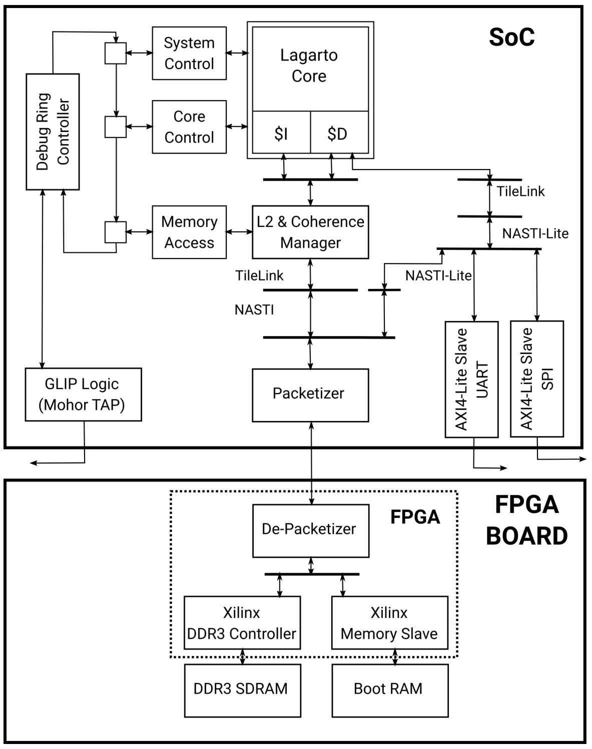


Figure 2.1: PreDRAC processor block diagram.

The SoC contains the Lagarto core, two L1 caches of 16 KB, one for instructions and the other for data, and a single L2 cache of 64 KB, additional logic for debugging and peripherals to connect the processor with external devices like JTAG, UART or SPI.

Due to the lack of a physical interface, it was not possible to integrate a memory controller into the SoC. The memory controller was implemented in an external FPGA board, the Xilinx Kintex KC705, which had an on-board DDR3 memory that was used as the main memory of the preDRAC.

An AXI bus is used to interconnect the memory controller and the core. Due to the limited amount of I/O pins, it was not possible to make the whole AXI interface available at chip level. To solve that, and a custom interface, the packetizer, was used to send the AXI requests from the core to the memory controller in the FPGA. What is done inside the packetizer, is to serialize the AXI interface so a single AXI transfer is split into multiple transfers with less width. The FPGA board was connected to the main board, the one containing the preDRAC SoC, with an FMC connector.

This setup has some drawbacks. For one side, the FMC connector only supports signals up to 50 MHz. This made the clock speed of the transfers to be reduced from 200 MHz to 50 MHz. Moreover, since the AXI transfers were split in 4, it takes 4 times more to send each transfer.

The main performance bottleneck is the access to main memory due to the lack of a memory controller. To improve the performance of the system, a memory controller will be designed to provide an on-chip memory interface.

2.2 Memory

Memory is an essential part of any data processing system since it is where instructions and data are stored. Access to memory has an significant impact and limits the performance of the system.

This section provides an overview of how memory is organized in a processing system and why a memory controller is required. Then the principle of operation of DRAM (Dynamic Random Access Memory) is reviewed in order to get a better understanding of which are the tasks that should be performed by the controller.

After the DRAM overview, special focus will be put on SDRAM (Synchronous Dynamic Random Access Memory), in particular the SDR (Single Data Rate) SDRAM, which is the target of this project.

2.2.1 Memory hierarchy

As applications become more complex larger amounts of data need to be processed. To use more memory while keeping low the time required to access it processors are organized in multiple levels of memory hierarchy [5]: smaller but faster memories are placed close to the processor while larger but slower ones go to subsequent levels. While it is not the scope of this work to analyze the memory hierarchy, a brief overview is done to provide some context and have a better understanding of the final purpose of this project.

The main memory, where the program data and instructions are stored, is external to the processor since it is too big to be integrated on-chip. Inside the processor, a smaller memory, the cache memory, is used to buffer the data from the main memory so it can be accessed faster by the processor. Both the main and cache memories are volatile, which means they lose the data when they are powered off. Usually, there is a secondary non-volatile memory in a more external level of the hierarchy that keeps the data even if powered off.

Different technologies are used to implement the memories in the different levels of the memory hierarchy. Secondary memories are made of non-volatile flash memories, or magnet disks when larger and cheaper storage is required. Higher levels in the hierarchy are implemented with volatile memories: the main memory is implemented with DRAM and the caches with SRAM (Static Random Access Memory).

SRAMs typically use six transistors to store each bit of data. They are made with standard CMOS technology, can be placed on the same silicon as the processor, and have faster access time compared to the other types of memories. This is why SRAMs are used to implement cache memories. On the other hand, DRAMs only require a single transistor plus an additional capacitor to store each bit of data, so they have higher memory density, but also higher latency. The higher density makes DRAM large enough to be used as main memory and store the program data. DRAMs are fabricated with a different process than standard CMOS and are typically used in an external chip.

A processor has to access the different layers of memory. It has direct access to the cache, which is made of SRAMs on the same silicon, but it does not has direct access to the main memory, which is typically made of DRAM and external to the chip.

A memory controller that acts as an interface between the processor and the main memory is needed. The controller has to manage the memory requests from the processor and read or write data to the external DRAM while also control the refreshing of the DRAM to ensure data is not lost.

2.2.2 DRAM

As previously mentioned, DRAM is a volatile memory that use capactitors as storage elements [6]. Since capacitors are not ideal, the charge is progressively lost due to leakage currents. To prevent the loss of information, DRAM needs to be periodically refreshed. This is the reason why they are called dynamic.

In a reliable memory, not a single bit of data should be lost. Every single DRAM cell must be refreshed before it loses its stored charge. In a DRAM, all cells are typically refreshed every 32 or 64 ms.

Figure 2.2 shows the basic structure of a DRAM cell, which is composed of one transistor and one capacitor (1T1C) and can store 1 bit of data.

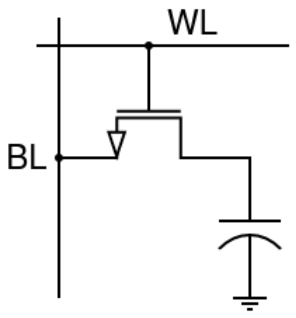


Figure 2.2: Schematic of a 1T1C memory cell.

Principle of operation

The main operation of a DRAM is based on the 1T1C cell previously mentioned. First, to understand how DRAM works, the operation of the basic storage cell will be reviewed. As seen before, the structure of a basic DRAM memory cell is composed of one transistor and one capacitor. The capacitor is the storage element while the transistor is used to control the access to the data.

The gate of the transistor is connected to the word-line. The word-line enables or disables the access to data. When the word-line is enabled, the gate of the transistor is activated and the storage capacitor is connected to the bit-line so it can be accessed for reading or writing. Otherwise, if the word-line is disabled, the transistor is off and the capacitor is disconnected from the bitline, keeping its current value.

To write data into the capacitor then the bit-line must drive the value to be written. Then, when the word-line is enabled, the capacitor will be charged or discharged to the voltage of the bit-line. On the other hand, to read data from the capacitor the bit-line should be floating, that way when the word-line is enabled the bit-line will be charged with the curent value of the capacitor.

Since the capacitor is not ideal, the charge stored in the capacitor will eventually leak and it will be discharged even if it is not accessed. This means the value of the capacitor must be constantly read and written in order to keep the data. Additionally, in this kind of cell, the reading is destructive, which means that reading a stored value implies discharging the capacitor so, after a read, the value must be written again to not lose the data.

Internal Architecture

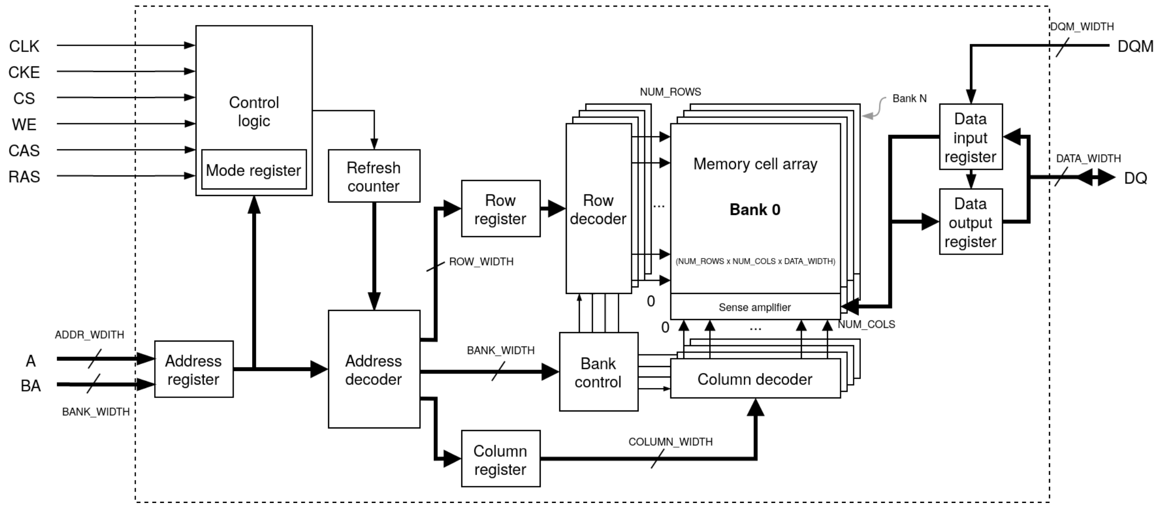


Figure 2.3: Functional block diagram of a SDR SDRAM.

Internally, DRAM is organized in memory arrays where each cell can be accessed by its row and column. DRAM does not operate with single bits, actually multiple memory arrays are put together and operate as a single unit, typically with word sizes of 4, 8, 16 or even 32 bits in modern memories. This defines the width of the data bus in the DRAM chips which is denoted as x4, x8, x16 or x32.

Additionally, memory arrays are grouped into banks. Each bank acts as an independent unit and it can be activated, precharged, read, or written while other operations are performed on other banks. There are some limitations, for example since the output data bus is shared for all banks, it is not possible to simultaneously read from two different banks at the same time.

Conventional DRAMs are asynchronous while SDR SDRAMs have a synchronous interface and all operations are controlled by a reference clock. Internally, the basic structure of a DRAM and a SDR SDRAM is the same. The main difference is that SDR SDRAMs add input and output registers and the control logic is different. Figure 2.3 shows the diagram of the main blocks in a SDR SDRAM chip. There is a control logic that decodes the input signals to determine which operation should be performed. Similarly, the input address is decoded to obtain the row, column, and bank to be accessed. Each bank has its own sense amplifier that is connected to the output data bus.

Sense amplifiers are used to detect the value stored in the capacitors [7]. The operation is performed as follow. First, bit-lines are initially precharged at VDD/2. Then, the storage capacitors are connected to the bitline through the access transistor and the capacitors charge or discharge the bit-lines producing a small variation on its voltage. The sense amplifier compares the voltage of the bit-line with a reference and the small differences are amplified to full VDD or GND.

The process to read from the DRAM requires to first precharge the bit-lines, read the values and then rewrite them since the read is destructive.

2.2.3 SDR SDRAM

After the overview of DRAM in the previous section, this section will give a a more specific review of the timing requirements and relevant commands involved in the operation of a DRAM. In this case, the focus will be specifically the SDR SDRAM. Table 2.1 shows the different signals in a SDR SDRAM chip.

Table 2.1: SDR SDRAM interface signals

Name	Width	Type	Description
CLK	1	Input	Reference clock. All SDRAM input signals are sampled at the positive edge of the clock.
CKE	1	Input	Clock enable, active high. When low, disables the CLK signal. It is used to control the low power operation modes of the SDRAM.
CS	1	Input	Chip select, active low. Enables or disables the command decoder.
WE	1	Input	Write enable. Command bit in the SDRAM, in a DRAM it was used to indicate the type of operation (read or write).
CAS	1	Input	Column Address Strobe. Command bit in the SDRAM, in a DRAM it was used to indicate the address was a valid column address.
RAS	1	Input	Row column strobe. Command bit in the SDRAM, in DRAM it was used to indicate the address was a valid row address.
A	≥ 11	Input	Address bits. Width depends on the size of the SDRAM, at least it has 11 bits. The 11th bit is used to control the operation of some commands.
BA	1, 2, 3	Input	Bank selection bits. Width depends on the number of banks in the SDRAM. SDRAMs can have 2, 4 or 8 banks.
DQ	4, 8, 16, 32	Inout	Bidirectional data bus, it can have 4, 8, 16 or 32 bits.
DQM	1, 2, 4	Input	Data mask. There is one mask bit for each byte in the data bus.

SDRAM modules are provided with different configurations of capacity, element size and maximum frequency [8] [9].

Internally the modules are configured as quad-bank DRAM with a synchronous interface: the accesses are synchronous to the reference clock and the data is registered at the positive edge of the clock. Accesses to the SDRAM are burst oriented: each request can be made up of multiple transfers. Read and write operations start at a given address and continue for a programmed number of locations, which can be of 1, 2, 4 and, 8 locations or without limit. In case of unlimited burst, refered to as a full-page burst, an additional command must be sent to terminate the operation.

The commands of the SDRAM will be described in more detail on the following lines, along with the main timing requirements. Additionally, some examples of the basic operations will be provided.

Commands

To control the operation of the SDRAM a set of commands are sent through the CS, RAS, CAS, and WE pins, then the mask (DQM) bank (BA) and address (A) bits provide additional configuration for some commands. Table 2.2 summarizes the commands used to operate with an SDRAM. Command names can differ from one vendor to another, but their functionality is the same. The 11th bit of the address (A[10]) is used to control the precharge of the SDRAM. Since it has a special functionality it is given a dedicated column separated from the rest of the address.

On the following lines, each command will be described with a bit more detail to have a better understanding of their functionality and how it relates to the internal operation of the SDRAM.

Table 2.2: SDRAM commands summary.

Command	**CS**	**RAS**	**CAS**	**WE**	**DQM**	**BA**	**A**	**A[10]**
No Operation	L	H	H	H	x	x	x	x
Active	L	L	H	H	x	Bank	Row	x
Read	L	H	L	H	Mask	Bank	Col	Precharge
Write	L	H	L	L	Mask	Bank	Col	Precharge
Burst Terminate	L	H	H	L	x	x	x	x
Precharge	L	L	H	L	x	Bank	x	All banks
Auto refresh	L	L	L	H	x	x	x	x
Load Mode Register	L	L	L	L	x	Opcode	Opcode	x

No operation
: The *No Operation* is an auxiliary command that is used when no other command should be sent to the SDRAM, either because it is in idle or waiting for the completion of another command. It is equivalent to disabling the chip select.

Precharge
: The *Precharge* command deactivates an active row, which is done by resetting the sense amplifiers and precharging the bit-lines to the reference voltage so they are ready for another row access. There is a minimum time, t_{RP}, between a *Precharge* command and the next *Active* command.

Auto Refresh
: The *Auto Refresh* command must be sent during the normal operation of the SDRAM to refresh its contents. The SDRAM automatically generates the address that needs to be refreshed with an internal counter, the refresh counter shown in 2.3, which is updated each time the *Auto Refresh* command is sent. All the rows in the SDRAM must be refreshed within the refresh period, t_{REF}. Commands can be distributed along the period or sent all of them in a burst.

  After a *Precharge* command a minimum time, t_{RP} must be respected before sending the *Auto Refresh* command. There is also a minimum time, t_{RFC}, between two consecutive *Auto Refresh* commands.

Load mode register

The *Load mode register* command is used to configure different parameters of the related to the operation of the SDRAM, such as the burst length or the CAS latency.

The *Load mode register* can only be sent when all banks are precharged. After sending it some time, t_{MRD}, must be waited for the SDRAM to be configured and no other commands can be sent.

Active

The *Active* command is used to open (activate) a row in a particular bank so it can be further accessed with a *Read* or *Write* command. When using the *Active* command, the bank bits select the bank while the address bits determine the row. Internally this operation moves the data from the memory cells of the selected row and bank in the SDRAM array to the sense amplifiers, then restores the values of the memory cells so data is not lost.

The whole row of data is moved into the sense amplifier. This means that multiple accesses to read or write data can be done on different columns of the same row without having to send the *Active* command each time.

There is a minimum time, t_{RCD}, before *Read* or *Write* commands can be sent after having sent an *Active* command. An *Active* command to another row of the same bank can only be sent after precharging the bank. The sense amplifiers of the bank have to be reset before reading new data. On the other hand, it is possible to send an *Active* command to a different bank, since each bank has its own set of sense amplifiers. There is a minimum time, t_{RRD}, between consecutive *Active* commands to different banks.

Read

The *Read* command is used to start a read access in a previously opened row. The bank bits select the bank while the address bits select the column to be read. Additionally, the 11th bit of the address determines if an auto precharge will be performed by the SDRAM after data has been read. Internally, this command moves the data from the sense amplifier of the given bank to the output data bus. Once the command is sent, the read data is available on the data bus after t_{CAS}. There is a minimum time, t_{CCD}, between two consecutive *Read* or *Write* commands.

Write

The *Write* command starts a write access to a previously opened row. As with the *Read* command, the bank bits select the bank and the address bits the column with the 11th bit enabling auto precharge. Internally the *Write* command copies the data from the data bus into the sense amplifiers of the given bank. There is a minimum time, t_{WR}, between a *Write* and a *Precharge* command to ensure the sense amplifiers are not precharged until data is written into memory.

Burst terminate

In an SDRAM, reads and writes are burst oriented, which means each *Read* or *Write* command performs several accesses to memory. The *Burst terminate* command is used to truncate the burst started with the last *Read* or *Write* command.

Operation

Initialization

Before normal operation, the SDRAM must follow a set of initialization steps. Once power is applied and the clock is stable, SDRAM requires a delay of 100 μs. During this period, only the *No Operation* command can be sent. Alternatively, the chip select can be disabled.

After the initial delay, the *Precharge* command must be sent to all banks to ensure there is not any open row. Then at least two refresh cycles must be performed by sending the *Auto Refresh* command. After that the SDRAM is ready.

Since the state of the mode register will be undetermined after power up, the last step previous to use the SDRAM is to load the mode register with the right configuration using the *Load Mode Register* command.

Write operation

Figure 2.4 shows an example of a basic write request to SDRAM memory with a burst size of 2. Initially, the bank is precharged. Since the 11th bit of the address (A[10]) is low, only the selected bank will be precharged. Then the *Active* command is sent to open the selected row in the selected bank.

After waiting for the required time, the *Write* command is sent to write the data into the specified column of the previously opened row in the selected bank. Since the 11th bit of the address is not enabled, the SDRAM will not do an automatic precharge after writting. In this example, a burst of 2 transfers is sent. The data of the first transfer is sent on the same cycle as the *Write* command, the data of the next transfer is sent on the following cycle.

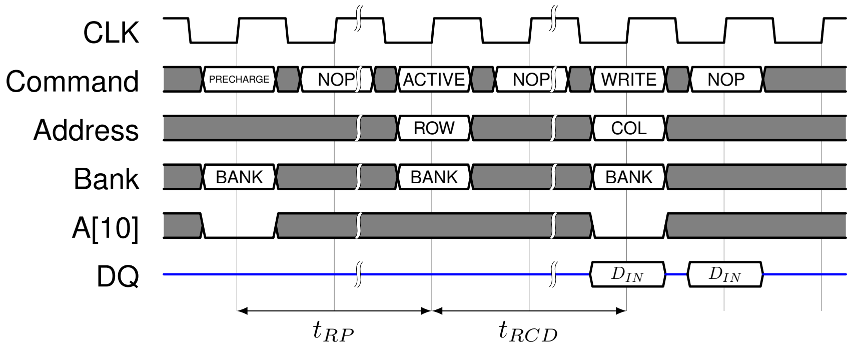


Figure 2.4: Timing diagram of a write operation in a SDRAM.

Read operation

Read operation is similar as the write operation. In this case, as shown in figure 2.5, after the *Read* command the data will be ready after the number of cycles defined by the CAS latency. In case of a burst operation, subsequent data will be updated on next cycles.

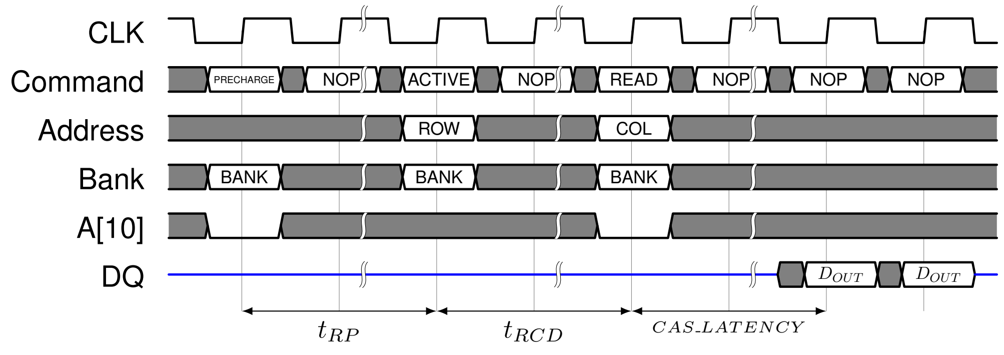


Figure 2.5: Timing diagram of a read operation in a SDRAM.

Timing parameters

Table 2.3: Summary of SDRAM timing parameters

Parameter	Description
t_{RCD}	Row to Column Delay, it is the time it takes for the *Active* command to move the data from the SDRAM cell array to the sense amplifier.
t_{RRD}	Row activation to Row activation delay, it is the minimum time between two *Active* commands.
t_{RAS}	Row Access Strobe latency, is the time it takes for the *Active* command to discharge and restore the data from the row of SDRAM cells. After this time the sense amplifiers have completed teh resotarion of data to the SDRAM cells and the sense amplifiers can be precharged to access another row.
t_{CAS}	Column Access Strobe latency, it is the time it takes to the SDRAM to put the the data on the data bus after receiving the *Read* command.
t_{CCD}	Column to Column Delay, it corresponds to the minimum burst duration and it is determined by the internal prefetch length of the SDRAM.
t_{RP}	Time needed before the bitlines and sense amplifiers are properly precharged after a *Precharge* command.
t_{WR}	Write recovery time, is the time needed for the write data to be written into the internal SDRAM arrays. It the minimum time between a *Write* and a *Precharge* command.
t_{RFC}	Required time between two consecutive *Auto Refresh* commands.
t_{MRD}	Required time between an *Load Mode Register* and the next *Active* or *Auto Refresh* command.
t_{REF}	Refresh period within all the rows of the SDRAM should be refreshed.

2.3 AMBA AXI

The ARM Advanced Microcontroller Bus Architecture (AMBA) is an open-standard, on-chip interconnect specification for the connection and management of functional blocks in system-on-chip (SoC) designs [10]. AMBA specifications provides the interface standards that enable IP re-use and compatibilty between different IPs, which is essential to reduce development costs. Additionally, AMBA provides a variety of interface specifications optimized for different requirements in terms of performance, power and area. Because of that, AMBA is flexible and widely used in multiple ASIC and SoC parts. AMBA provides a set of prototols that define how functional blocks communicate with each other, which has been constantly updated [11] [12]:

- The first generation of AMBA was introduced by ARM in 1997 and defined two specifications: the Advanced System Bus (ASB) and the Advanced Peripheral Bus (APB).
- The second generation, AMBA 2, appeared in 1999 and added the AMBA High-performance Bus (AHB).
- The third generation was introduced in 2003. AMBA 3 included the Advanced Trace Bus (ATB) and the Advanced eXtensbile Interface (AXI) targeted at high performance and high clock frequency designs and is designed to maximize bandwidth and reduce the latency of data transfers.
- In 2010 AMBA 4 specification added a revision of the AXI protocol, the AXI4, and included the AXI coherency Extensions (ACE) to provide system wide coherency which allows multiple processes to share the same memory resources.

Two metrics are used to determine the performance of a bus interface:

- Bandwidth: Rate of the data across the interface. In a syncrhonous system, the maximum bandwidth is limited by the product of the clock speed and the width of the data bus.
- Latency: Delay between the initiation and the end of the transfer. In a burst transfer, latency can also be defined as the delay between the start of the transaction and the end of the first transfer rather than the whole burst.

AXI is commonly used by many IP. It is the main interface to access the peripherals in the lowRISC SoC and therefore in the preDRAC SoC. AXI specification defines the interface between IP blocks. There are only two types of interfaces, master and slave, which are symmetrical. This section will give a summary of the main characteristics and signals of the AMBA AXI4 protocol [13].

2.3.1 Signals description

Table 2.4: AXI global signals.

Signal	Required	Description
ACLK	Required	Global clock signal.
ARESETn	Required	Global reset signal, active low.

Table 2.5: AXI Address Write channel signals.

Signal	Source	Required	Description
AWID	Master	Required	Identification tag of the write transaction.
AWADDR	Master	Required	Address of the first transfer in the write transaction.
AWLEN	Master	Required	Number of data transfers in the write transaction.
AWSIZE	Master	Required	Number of bytes in each transfer in the write transaction.
AWBURST	Master	Required	Burst type. Indicates how the address should be updated for after each transfer.
AWLOCK	Master	Optional	Provides additional information for atomic transfers.
AWCACHE	Master	Optional	Indicates how the transaction should progress through the system.
AWPROT	Master	Optional	Additional protection attributes for different priviledges and access levels.
AWQOS	Master	Optional	Quality of Service identifier.
AWREGION	Master	Optional	Additional region indicator.
AWUSER	Master	Optional	Additional user-defined extension.
AWVALID	Master	Required	Indicates when the write address channel signals are valid.
AWREADY	Slave	Required	Indicates that a transfer on the write address channel can be accepted.

Table 2.6: AXI Write Data channel signals.

Signal	Source	Required	Description
WDATA	Master	Required	Write data
WSTRB	Master	Required	Write strobes, indicate which bytes are valid.
WLAST	Master	Optional	Indicates which is the last transfer in a write transaction.
WUSER	Master	Optional	Additional user-defined extension.
WVALID	Master	Required	Indicates the write data channel signals are valid.
WREADY	Slave	Required	Indicates that a transfer on the write data channel can be accepted.

Table 2.7: AXI Response channel signals.

Signal	Source	Required	Description
BID	Slave	Required	Identification tag of a write response.
BRESP	Slave	Optional	Write response indicating the status of the write transaction.
BUSER	Slave	Optional	Additional user-defined signals.
BVALID	Slave	Required	Indicates the write response channgel signals are valid.
BREADY	Master	Required	Indicates the transfer of the write response channel can be accepted.

Table 2.8: AXI Read Address channel signals.

Signal	Source	Required	Description
ARID	Master	Required	Identification tag for a read transaction.
ARADDR	Master	Required	Address of the first data transfer in a read transaction.
ARLEN	Master	Required	Number of data transfers in a read transaction.
ARSIZE	Master	Required	Number of bytes in each data transfer in a read transaction.
ARBURST	Master	Required	Burst type, indicates how address is updated after each transfer.
ARLOCK	Master	Optional	Additional signals for atomic transfers.
ARCACHE	Master	Optional	Additional signals to specify how signals progress through the system.
ARPROT	Master	Optional	Additional signal for different privilege and access type.
ARQOS	Master	Optional	Quality of Service additional signals.
ARREGION	Master	Optional	Region indicator for a read transaction.
ARUSER	Master	Optional	Additional user-defined signals.
ARVALID	Master	Required	Indicates that read address channel signals are ready.
ARREADY	Slave	Required	Indicates that the transfer on the read address channel can be accepted.

Table 2.9: AXI Read channel signals.

Signal	Source	Required	Description
RID	Slave	Required	Identification tag for read data.
RDATA	Slave	Required	Read data.
RRESP	Slave	Optional	Read response indicating the status of the transfer.
RLAST	Slave	Required	Indicates the last transfer in a read transaction.
RUSER	Slave	Optional	Additional user-defined signals.
RVALID	Slave	Required	Indicates that the read data channel signals are valid.
RREADY	Master	Required	Indicates that a transfer on the read data channel can be accepted.

2.3.2 AXI architecture

Clock

Each AXI component uses a single clock signal. Input signals are sampled on the rising edge of the clock and all output signals changes must occur after the rising edge of the clock. Between master and slave devices there must not be any combinational logic between inputs and outputs.

Reset

The AXI protocol uses as single active low reset signal. It can be asserted asynchronously but it must be deasserted synchrously with the rising edge of the clock. During reset, the master must keep the signals ARVALID, AWVALID and WVALID low and the slave must keep RVALID and BVALID signals low. All the other signals can have any value. The master can assert ARVALID, AWVALID or WVALID at the rising edge of ACLK after ARESTETn is high.

AXI channels

AXI transactions are made of five independent channels:

- Read address channel (AR).
- Read data channel (R).
- Write address channel (AW).
- Write data channel (W).
- Write response channel (B).

The read address and read data channel are used in read transacions while the write address, write data and write response channels are used in write transactions. Since read and write channels are independant from each ohter, read and write operations can happen at the same time.

Channels are unidirectional, which means they go from master to slave or the other way around. The address channels go from master to the slave and contain information about the transfer, including the address that will be accesses. On a read transfer, the read data channel contains the response from the slave, including the read data. In a write tranfer, the write data channel contains the data that has to be written into the slave. The additional response channel is needed in write transfers so the slave can send a response to the master.

Handshake

The AXI protocol uses a hanshake mechanism so each channel has a pair of signals, VALID and READY. The source uses the VALID signal to indicate the channel contains valid data while the destination uses the READY signal to indicate it can accept new data. Once both VALID and READY of a given channel are asserted the handshake occurs at the rising edge of the clock signal and the transaction is completed. Additionally, write and data channels also have a LAST signal to indicate if the last item of the transaction is being transfered.

In the handshake process, the source must assert the VALID signal when it has valid data, it must not wait for the READY signal to be asserted. Once asserted the VALID signal must remain high until the handshake occurs. The destination can wait for the VALID signal before asserting the correspoding READY and if READY is asserted, it can be deasserted before VALID is asserted. To prevent a deadlock situation the VALID signal must not depend on the READY signal.

AXI transactions

AXI allows multiple outstanding addresses, which means a master can issue transactions without waiting previous transactions to complete. Each channel has a field used to send the identifier (ID) of the transfer. Transfers that have the same ID must be processed in order, but out-of-order transactions can be done by using different transaction IDs.

AXI is a burst-based protocol and each transaction can be made of multiple transfers. The address channel contains the information that defines the size and the number of transfers as well as the type of burst. The address is sent only once per transaction, then the slave has to generate subsequent addresses according to the type of transfer specified by the master.

2.3.3 Operation

Write transaction

Figure 2.6 shows an example of a write transaction. The transaction starts when both the write address channel and write data channel have done the handshake. In a burst transfer, the write data channel will be updated for each transfer. On the last transfer, the WLAST signal is enabled. Once all the data has been sent a response is returned. The transaction has completed after the handshake on the response channel.

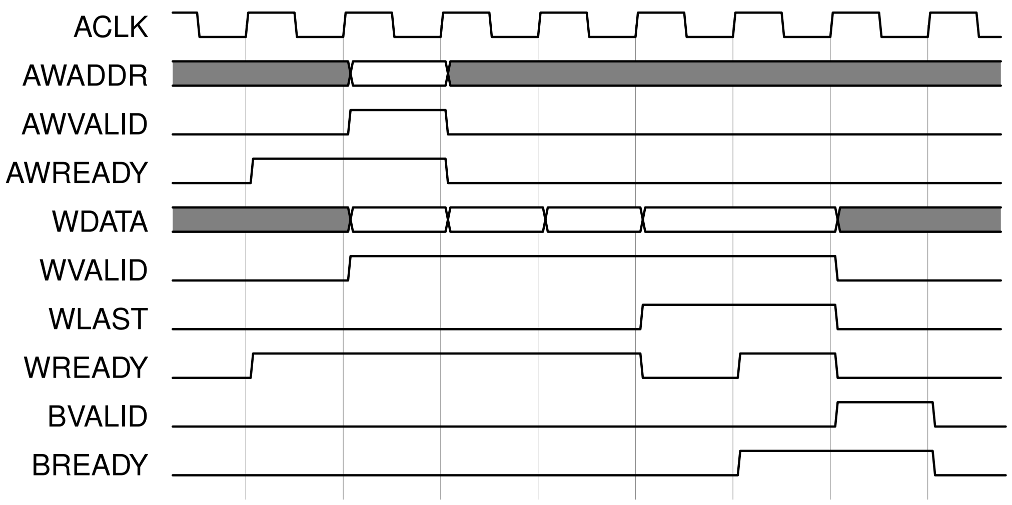


Figure 2.6: Timing diagram of an AXI write transaction.

Read transaction

Figure 2.7 shows an AXI read transaction which is similar to the write transaction. In this case the transaction starts once the handshake of the read address channel is produced. The slave returns the read data and enables the RLAST signal on the last transfer.

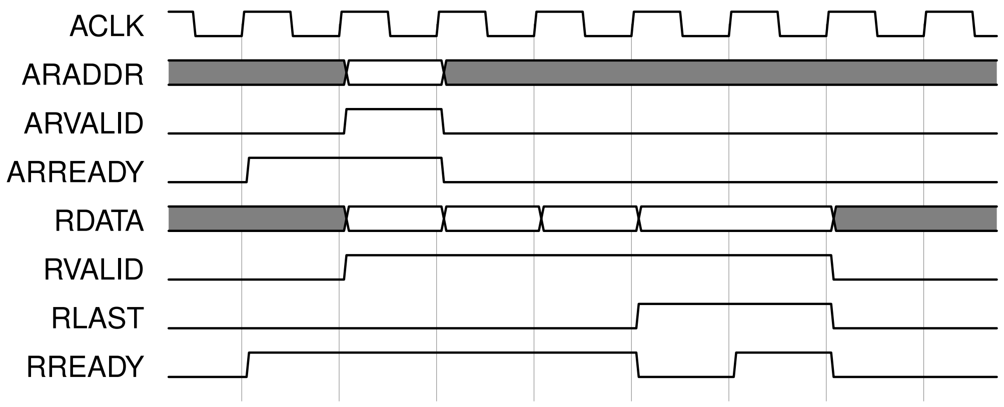


Figure 2.7: Timing diagram of an AXI read transaction.

2.4 ASIC design flow

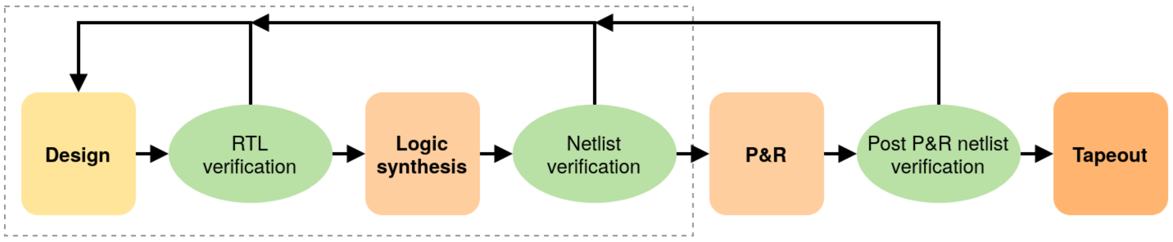


Figure 2.8: Different steps in the ASIC design flow

Before a design is ready to be put on a chip, several steps must be performed. ASIC design is a complex process which is summarized in figure 2.8.

The process starts with a set of specifications that define the functionality of the design, from there the initial RTL is developed. Additionally, a testbench is also developed in parallel with the RTL. At the end of the design phase, both an RTL and a testbench are obtained.

After the initial design stage the functionality of the RTL must be verified. If the verification is not successful, the design needs to be modified and then verified again. This is an iterative process that can take several iterations. The verification process is seen in more detail in section 2.4.3.

The next step is the logic synthesis. During this step, the high-level RTL description is mapped into a netlist made of logic gates. The gates are defined in a standard cell library, which contains area, power, and timing information. More about the synthesis process in section 2.4.1.

A post-synthesis verification is done to ensure the obtained netlist has the same behavior as the original RTL.

Finally, on the Place&Route (P&R) step the previous netlist is placed into the actual silicon area, the routing of the different cells is performed, and the clock tree is generated. In the end, another netlist and a physical layout are obtained.

A post-P&R verification is done to ensure this last netlist still behaves as expected. In this phase, there are additional checks to ensure the final layout meets the design rules so it can be manufactured, or that the design is within the power constraints.

Once the design is completed and verified, the obtained layout can be sent to tapeout.

This completes an overview of the full ASIC design flow. This project does not follow the full flow, but only the first steps of design and logic synthesis, with their respective verifications, are performed. The flow used in this project is the one within the dashed box in figure 2.8. In the following sections the main synthesis and verification steps are analyzed in more detail. Moreover, though it could be considered part of the verification, there is a separate section dedicated to power analysis.

2.4.1 Synthesis

To manufacture a chip what is needed is a layout that describes the shape and position of each transitor on the wafer. Since the number of transistors of digital designs is extremely high, it is not feasible to manually place all of them. Digital designers work at a higher level of abstaction and use RTL descriptions to define the behavior of the design.

Foundries still require the layout to manufacture the chip so the original RTL desription has to be converted into the final layout. This is what is done by the synthesis and Place&Route processes.

Logic synthesis is the process of converting the RTL desription into a gate-level netlist. At the same time, this process has different steps [14]:

- Generic synthesis: During this initial step, the original RTL is converted into a set of registers and combinational logic made of generic gates.
- Mapping: This step maps the design from generic gates to a specific technology library.
- Optimization: The final step performs incremental optimizations on the mapped design in order to meet the given design contraints.

Additionally, physical synthesis is a similar process but uses information from a floorplan so it has more accurate timing information based on the actual placement of the different cells and can optimize the design taking such information into consideration.

2.4.2 Place&Route

Place&Route is composed of several steps to convert the netlist obtained from synthesis into the final layout that can be sent to the foundry. The main steps are:

- Floorplanning: this initial step consist in designing the layout and how the different elements of the design will be placed on silicon.
- Placement: after having the floorplan, this step consist on placing the cells or other macros into the actual silicon area according to the floorplan previosly designed, which is automatically performed by the tool.
- Routing: in this step the tool perform the connections of the different nets in the design.
- Parasitic Extraction: After routing the parasitic resistors and capacitors of the wires can be extracted. This information can be used later to verify the proper behavior of the system, both in terms of timing but also power, noise or IR drop.
- Clock Tree Synthesis: This step generates the routing of the clock and places the required buffers so the clock reaches all the the elements in the design minimizing the skew.

2.4.3 Verification

Verification is an essential step in the ASIC design process, as it is crucial to ensure the proper operation of the ASIC before sending the design to the foundry and manufacture the chip. Verification consist of checking the design meets timing, power, and other constraints as well as testing the functionality to make sure the design will behave as expected once implemented on silicon. It is an essential part of the ASIC design process, which can take from 40% to 70% of the total development efforts [15].

Different tools and methodologies are used in in the industry in order to both provide more accurate results, which increase the reliability of the verification, and to speed up the process and accelerate the design flow. Such methodologies can be divided into the following groups:

- **Simulation:** use a testbench to provide stimulus to the design and evaluate its behavior.
- **Static verification:** includes linting and static timing analysis. Lint checking revises the syntaxis of the code to find possible errors such as uninitialized variables, unsupported constructs or port mismatches, and static timing verification checks if the design meets the timing requirements. Unlike simulation, it does not require a testbench.
- **Formal verification:** use of mathematical techniques to verify the correctness of the design. One type of formal verification is the equivalence check, which verifies the equivalence between two designs, for example between RTL and a netlist or between two netlists obtained at different stages, such as one after logical synthesis and the other after physical synthesis. It does not require testbench but only a functional reference design.
- **Physical verification:** check the integrity of the design to determine if it meets the design rules and other electrical or power requirements to ensure it can be manufactured.

Verification is not just a final step in the design flow, but a reiterative process that should be done at every stage. Previously, figure 2.8 showed here is a verification step after each development stage in the ASIC design flow. First, an initial verification of the RTL design, then another verification after the different synthesis to ensure both the netlist has the same functionality as the original design as well as timing requirements are met. In the following lines, the methodology and tools used to perform the verification of the design at the different stages of the ASIC design flow will be described in more detail.

RTL verification

The first step in the verification process is to simulate the RTL design to ensure it behaves accordingly to the desired functionality. At this stage both simulation and static analysis methods can be used to verify the design:

- **Behavioral simulation:** In RTL behavioral simulation a testbench is used to determine if the design under test (DUT) behaves as expeted or not [16]. The testbech can be seen as a wrapper around the DUT which generates the stimulus, applies such stimulus to the DUT, gets the response from the DUT and checks if the response coincides with the expected one. It is important to design an adequate testbench that applies the right stimulus to the DUT so the functionality is properly tested and a reliable verification is obtained. To generate the stimulus a testench use additional components, which can be non-synthetizatble modules used to emulate the behavior of external components and buses so the DUT can be properly tested.

  In a real synchronous system, due to the propagation delay, signals changes slightly after the clock edge. This is not true with RTL simulation, since there are no delays all signals change at the same simulation time. Depending on the simulator and how the different modules of the testbench are implemented, it can happen that some signals change before the clock edge and other signals after it but at end everything happens at the same simulation time. This problem is known as race condition. A possible workaround to avoid race condition is to add small delays so signals change after the clock edge like its expected from the real hardware. SystemVerilog offers more complex methodologies to prevent race condtition which are out of the scope of this project, more information about this topic can be found on C. Spear [16].

- **Static analysis:** Linting tools can be used to check the design and detect both syntax as well as possible design errors such as mismatch between signals, unused signals, combinational loops or inferred latches.

It is not needed to wait until the design is completely finished to start with the simulations. Starting with simulations at an early stage in the design phase is a good way to identify and fix errors, which can save a significant amount of time in later phases of the design.

Post-synthesis verification

After synthesis, the RTL is mapped into a gate-level netlist. The obtained netlist is closer to the final desing that will be implemented on silicon, as it is made of gates from the standard cell library that was provided during synthesis.

Since the technology and cell libraries provide timing information it is now possible to perform timing analysis on the design. After the logic synthesis both a gate-level netlist and a SDF (Standard Delay Format) file are obtained. The SDF file contains the delay information of each net in the design.

Again, it's important to verify the design and check that after synthesis the obtained netlist still performs as expected. At this stage, there are different methods that can be used to perform such verification. The main methods are:

- **Static timing analysis:** As opposed to simulation, static timing analysis [17] is a method used to verify the timing of a design which doesn't require a testbench that provides stimulus to it. On the other hand, what it does require is the timing constrains of the system (such as the definition of the different clocks).

  Static timing analysis performs an exhaustive verification, checking all the paths in the design, and validates if the design can operate at the desired speed. This is done with the same tool used for synthesis, Genus, which can generate detailed timing reports of the different paths in the design.

- **Equivalence check:** As previously mentioned, equivalence check is a way to verify that two different designs are functionally equivalent. This is a step that can be performed after synthesis to verify if the obtained netlist has the same functionality as the original RTL.

- **Gate-level simulation:** Verification methods such as static timing analysis and equivalence check can be used to verify the design but, for large and complex systems the combination of such methods might not be enough and another level of verification is required. Gate-Level simulation can help to find errors that static timing analysis or an equivalence check are not able to detect. For example it can help to verify the initialization and reset sequence.

Gate-level simulation consist in the simulation of a netlist. As with the RTL simulation, a testbench is needed to provide stimulus to the DUT. The testbench can be the same as the used in the initial RTL behavioral simulation. Since after synthesis timing information for the design is obtained, this information can be used to do a simulation taking into account such delays. At this point, different simulations can be performed:

- Zero-Delay simulation: This simulation does not contain delay information but it is still useful to validate the functionality of the netlist and verify that the initialization and reset sequence work as expected. The advantage of this kind of simulations is that are much faster than simulations with timing. On the other hand, since no timing checks are done they can not be used to completely verify the netlist.
- SDF simulation: In this case the delays defined in an SDF file are annotated into the corresponding nets of the design. After successfully reading and annotating the SDF file, it can be seen in a waveform viewer that the signals do not longer change with the clock but have some delay. This kind of simulations can be much slower than zero-delay simulations, but they are crucial to verify the actual behaviour of the system and check if the timing requirements, such as hold and setup times, are met for the different cells in the design.

Post-P&R verification

After the place and route (P&R) a new netlist and more accurate delay information are obtained. With this new information another static timing analysis, equivalence check and gate-level simulation can be performed. The verification process followed in this step is the same as in the post-synthesis verification seen above.

After this step a layout is generated, which is what is sent to the foundry to manufacture the chip. In this case, additional checks are required, such as design rule check (DRC) to ensure the design meets the requirements of the foundry and to ensure the layout is functionally equivalent to the given netlist.

2.4.4 Power analysis

For the last decades, following Moore's law, the number of transistors integrated on a single die has kept increasing. Technology scaling allowed a significant improvement of processors performance. At the same time with the increasing number of transistors as well as the operating frequency, power dissipation become a major concern in ASIC design.

Power dissipation depends on many factors such as the size of the design and how the architecture is implemented, the techonology and cells used in the fabrication process, the operating frequency and the actual task being performed. Often there is a tradeoff between power, area and performance. It is becoming more important to perform power analysis, not only as a final signoff verification step but also at intermediate verification to ensure the design stays within the desired constraints.

Power dissipation of an ASIC can be divided into two components, the static and dynamic power [18]. The source of the static power is the leakage current in the transistors, which depends for one side on the size and number of transistors in the design and also on the technology used to fabricate them:

$$P_{static} \propto I_{leakage} \cdot V_{DD} \tag{2.1}$$

On the other hand, the dynamic power depends mainly on activity of the ASIC:

$$P_{dynamic} = \alpha \cdot C_L \cdot V_{DD}^2 \cdot f \tag{2.2}$$

The final power is the combination of both:

$$P_{total} = P_{static} + P_{dynamic} \tag{2.3}$$

Static power

Static power is the power dissipated even when there is no activity, simply because the system is powered on.

Ideally, due to the complementary behavior of a CMOS cell there should not be any static power consumption since on steady state either the NMOS or the PMOS network are off and there is no direct path between V_{DD} and V_{SS}. With real transistors, there is some static power dissipation due to the leakage currents. The main source of static power disispation is the subthreshold leakage, which is the leakage current between drain and source because of the transistor not turning completely off when the gate voltage falls below the threshold voltage.

As mentioned, leakage depends on the actual technology and libraries used. This information is available on the cell libraries provided by the vendor, the same ones used during synthesis. In the tools used, Genus and Joules, the static power is reported as leakage power.

Dynamic power

Dynamic power is the power dissipited because of the activity of the design. It is produced when there is a transition in the state of the cells, eihter from high to low or low to high. The main source of dynamic power is the charging and discharging of load capacitances. An additional source of dynamic power is the short circuit that is produced during a transition where for an instant both NMOS and CMOS networks are on at the same time and the current flows from V_{DD} to V_{SS}.

To compute the dynamic power it's important to have an estimation of the activity of the design. Not every signal will change on every cycle so activity can bee seen as a factor (α in formula 2.2) that determines the rate of change of the signals. For every node, the capacitance and activity should be considered and the total power will be the sum of the power of all nodes.

The activity factor depends on the task being performed and sometimes it could be difficult to estimate which factor use. As an alternative, its also possible to obtain the activity factor from the switching information obtained with an RTL or gate-level simulations. Additionally, to compute the dynamic power are also needed the technology and cell libraries since they contain the information needed to obtain the capacitances of each node.

In the tools used, Genus and Joules, the dynamic power of each cell is divided into two: internal and switching power. The internal power is the power dissipated internally the cell, which includes the short circuit currents and the charging and discharging of parasitic capacitances within the cell. On the other hand, the switching power is the power dissipated because of the load capacitance on the output of the cell, which include the output capacitance and also the parasitic capacitance of the wire.

Chapter 3

Methodology

The purpose of this project is to design a memory controller that acts as an interface between the SoC and the external memory. For one side, the controller has to communicate with the main processor using the AXI protocol, previously described in section 2.3. On the other side, the controller has to send the required request to the external SDR SDRAM, described in section 2.2.3.

The design has been designed in Verilog, then synthetized, verified and finally a power analysis has been done. This chapter describes the different steps and methodologies used involved in the the design of the AXI-SDRAM controller.

3.1 Implementation

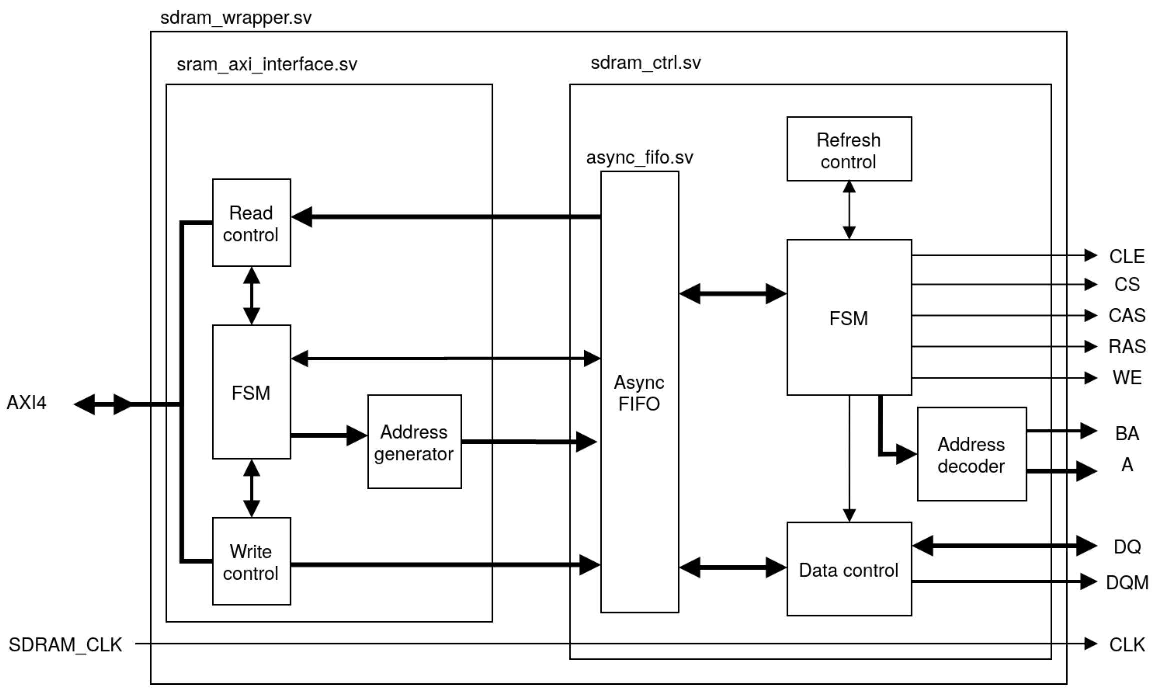


Figure 3.1: Organization of the design modules.

The purpose of the controller is to provide a simplied way to the accesses the memory. The proposed design is composed of two cascaded blocks as shown in figure 3.1:

- **AXI interface:** This module is used to adapt the previous SDRAM interface to the AXI4 specification, turning the design into an AXI4 slave that can communicate with the main processor through the AXI bus.
- **SDRAM controller:** This module interacts with the external SDRAM memory and provides a simple interface that makes the read and write operations transparent to the user. It contains the logic required to manage the accesses to the SDRAM, generate the appropiate commands and control the refresh sequence. Additionally it contains an asynchronous FIFO which is used to synchrnonize the data between different clock domains, from the main clock to the SDRAM clock and the other way around.

Both blocks are wrapped together to provide the final AXI-SDRAM interface. On the following lines the actual implementation of the different modules in the design will be described.

3.1.1 SDRAM controller

As previoulsy seen in section 2.2.3, SDRAM has a standard interface composed of several control signals (CAS, RAS, WE and CS) and an address, bank, data and mask buses. The SDRAM controller is built on top of that and provides a simpler interface which consist of a an address, both input and ouput data buses, a RW signal and a set of valid and ready signals to control the transactions, in a similar way as the handhsake in the AXI bus works. The whole set of signals is described in table 3.1.

The suffix *sdram_* is used for the signals that goes to the SDRAM, the prefix *_o* indicates the signal is an output of the SDRAM controller, similarly the prefix *_i* is used for inputs. An additional *_io* prefix is used to denote bidirectional signals. Moreover the previous table shows to which interface belongs each signal as well as its size. Some of the sizes are not fixed but depend on a paramater than can be set set when instantiating the design. This adds some flexibility as it allows the controller to be configured according to the specific needs of the project. Table 3.2 contains the modifiable parameters in the SDRAM controller.

Some of the parameters depend on the internal configuration of the SDRAM chips or how many chips are used. For example, multiple chips could be used in parallel, commonly referred to as a rack, which will increase the memory data width. Similarly, multiple racks could be used, which would increase the number of chip selects. There are other parameters, such as the number of rows and columns, which depends on the internal organization of the SDRAM chip. Additionally, the timing configuration can vary from one chip to another, so some parameters related to the operation of the SDRAM can also be modified. Such parameters include the CAS latency or some critical delays that must be respected, in such case the parameter value refers to the number of cycles of delay.

From the user interface side, the parameters allow the modification of the data and address widths and also if the controller includes or not the asynchronous FIFO used for synchronization. If master operates at the same frequency as the SDRAM there is no need to synchronize between both domains, then by removing the overhead of the synchronization the latency can be reduced by some cycles.

Parameters can be modified to adapt the controller to different system requirements, but

Table 3.1: SDRAM controller interface.

Port	Interface	Size	Description
clk_i	User	1	Main clock of the master system.
clk_sdram_i	User	1	Internal clock of the controller, at the same frequency of operation as the SDRAM.
rstn_i	User	1	Asynchronoust active low reset.
addr_i	User	ADDR_WIDTH	Input address bus.
rw_i	User	1	Read/Write enable signal, 0 for read and 1 for write.
data_i	User	DATA_WIDTH	Data input bus. Contains the data to be written into the SDRAM.
data_o	User	DATA_WIDTH	Data output bus. Contains the data read from the SDRAM.
busy_o	User	1	Output busy signal. Indicates the controller is busy and can not accept more requests.
valid_i	User	1	Input valid signal. Indicates the data_i, addr_i and rw_i signals contain a valid request.
ready_i	User	1	Input ready signal. Indicates the master can accept the response.
valid_o	User	1	Output valid signal. Indicates the request has been completed, in a read request it also means data_o holds valid data.
sdram_clk_o	SDRAM	1	Output clock to the SDRAM.
sdram_cle_o	SDRAM	1	SDRAM clock enable.
sdram_cs_o	SDRAM	SDRAM_CS_WIDTH	SDRAM chip selects.
sdram_cas_o	SDRAM	1	SDRAM CAS pin.
sdram_ras_o	SDRAM	1	SDRAM RAS pin.
sdram_we_o	SDRAM	1	SDRAM write enable pin.
sdram_ba_o	SDRAM	SDRAM_BANK_WIDTH	SDRAM bank bits.
sdram_a_o	SDRAM	SDRAM_ADDR_WIDTH	SDRAM address bus.
sdram_dq_io	SDRAM	SDRAM_DATA_WIDTH	SDRAM bidirectional data bus.
sdram_dqm_o	SDRAM	SDRAM_DATA_WIDTH/8	SDRAM data mask.

the current implementation has some limitations and not all parameters can be set to any value.To ensure the controller works as expected, some conditions must be respected:

- DATA_WIDTH must be a multiple of the SDRAM_DATA_WIDTH.
- The maximum number of chip selects, SDRAM_CS_WIDTH, is 4.
- User address, addr_i, must be aligned to the width of user data, DATA_WIDTH.

As seen in figure 3.1 the SDRAM controller module is composed of various blocks: the refresh controller, the asynchronous FIFO, the address generator, the data generator and a finite state machine that controls the whole operation.

Table 3.2: Parameters of the SDRAM controller.

Parameter	Default	Description
SDRAM_DATA_WIDTH	16	Number of bits in the external SDRAM data bus.
SDRAM_ADDR_WIDTH	13	Number of bits in the external SDRAM address bus.
SDRAM_BANK_WIDTH	2	Number of bits in the external SDRAM bank.
SDRAM_CS_WIDTH	1	Number of chip selects in the external SDRAM bank.
SDRAM_ROW_BITS	13	Number of bits needed to access all the rows in the external SDRAM module.
SDRAM_COL_BITS	9	Number of bits needed to access all the columns in the external SDRAM module.
SDRAM_BANK_BITS	2	Number of bits needed to access all the banks in the external SDRAM module.
SDRAM_WORD_SIZE	2	Size, in bytes, of each element in the SDRAM module.
DATA_WIDTH	128	Number of bits of controller interface data bus.
ADDR_WIDTH	26	Number of bits of the controller address bus.
CAS_LATENCY	3	CAS latency of the SDRAM.
ASYNC_FIFO	1	Selects if asynchronous FIFO is used (1) or not (0).
DELAT_T_RP	4	Number of cycles to meet t_{RP}.
DELAT_T_RFC	12	Number of cycles to meet t_{RFC}.
DELAT_T_MRD	2	Number of cycles to meet t_{MRD}.
DELAT_T_RCD	4	Number of cycles to meet t_{RCD}.
DELAT_T_RC	11	Number of cycles to meet t_{RC}.
DELAT_T_REF	1300	Number of cycles to meet t_{REF}/NUM_ROWS since refreshes are distributed along the whole period.

FSM

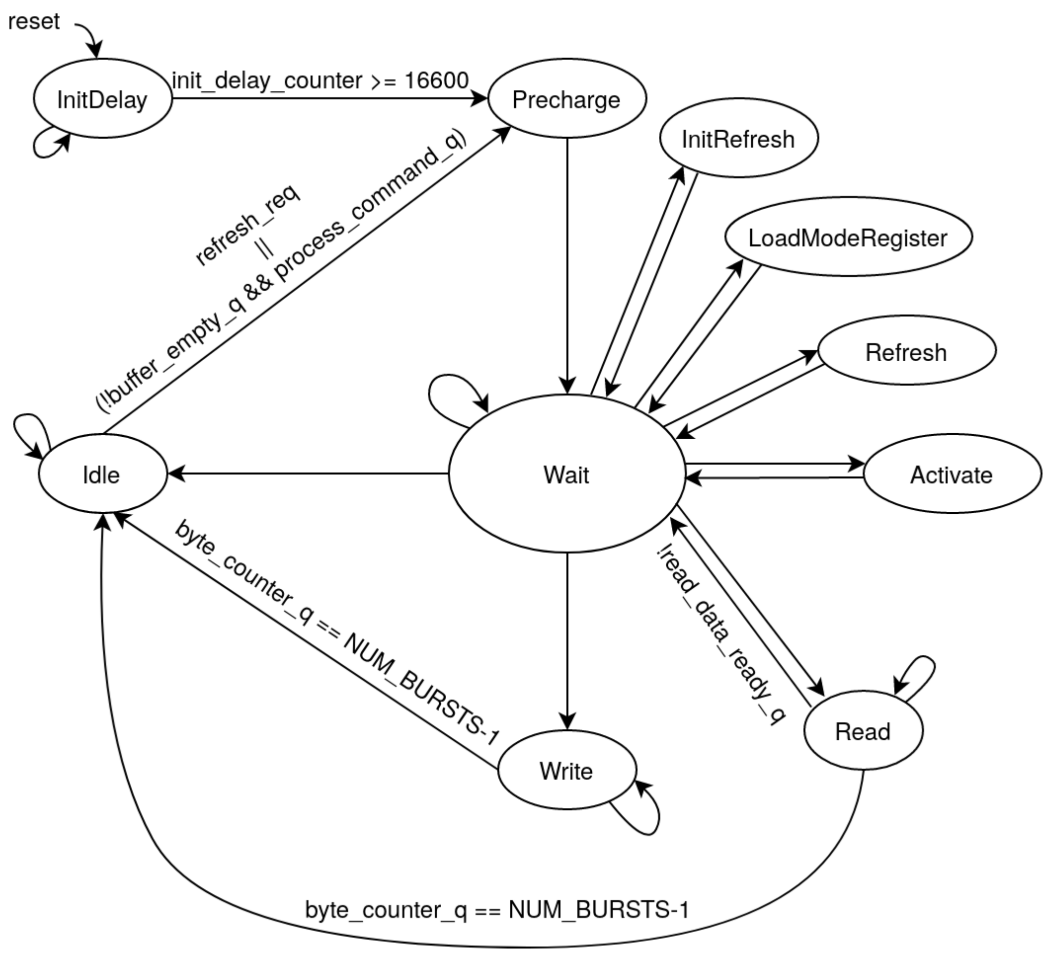
The operation of the SDRAM controller is controlled by a FSM, which is shown in figure 3.2. In total there are 10 states:

- InitDelay: In this state the controller waits for the initial delay of 100us required to initialize te SDRAM. This is done by means of an internal counter, once the counter limit it reached it jumps to the *Precharge* state. While in this state the controller keeps sending the *No Operation* command. This is the default state after a reset.

- Precharge: This state sends the Precharge command to the SDRAM, after that it goes into the *Wait* state to wait for the SDRAM to do complete the operation. From there it will resume to the *InitPrecharge*, *Refresh* or *Activate* state depending on the operation being performed by the controller.

- InitRefresh: During the initialization process the SDRAM needs to be refreshed 2 times. In this state the refresh command is sent to the SDRAM and then goes into *Wait* state. After waiting the state will be resumed back to *InitRefresh*. An internal counter keeps track of the number of refreshes.

  On the second refresh, it will jump again to *Wait* state, but this time it will resume in the *LoadModeRegister* state.



reset
InitDelay
init_delay_counter >= 16600
Precharge
InitRefresh
LoadModeRegister
Refresh
refresh_req || (!buffer_empty_q && process_command_q)
Idle
Wait
Activate
byte_counter_q == NUM_BURSTS-1
!read_data_ready_q
Read
Write
byte_counter_q == NUM_BURSTS-1

Figure 3.2: Diragram of the FSM in the SDRAM controller.

- LoadModeRegister: This state configures the mode register of the SDRAM. After that if goes to *Wait* state to wait for the required delay. After the delay it will resume to *Idle* state, and the SDRAM will be ready to take requests.
- Wait: It's an auxiliary state used to wait for the delays required by the different commands of the SDRAM. It has an internal counter whose limit is updated with the right value each time the controller jumps into this state, at the same time it is also specified the state to which the controller should jump after the waiting period. Then when the internal counter reaches the limit the controller resumes into the programmed state.
- Idle: This is the state where the controller will remain after initialization, it remains in this state until a refresh request or a user request are received. Once a request is received it jumps to the *Precharge* state.
- Refresh: In this state the *Refresh* command is sent to the SDRAM. After that it goes to *Wait* state and from there it will resume to *Idle* state.
- Activate: The controller reaches this state when a user request is being processed. In this state the *Activate* command is sent to the SDRAM. After that jumps to the *Wait* state and resumes to *Write* or *Read* state depending on the type of request being performed.

- Write: In this state the controller sends the *Write* command to the SDRAM. In a burst transfer an internal counter keeps track of the number of burst. The controller remains in this state until the last burst of data is sent, then goes to *Idle* state.
- Read: This state sends the *Read* command to the SDRAM. Then it goes into *Wait* state to wait for the CAS latency, then it will come back to the *Read* state. Similarly as with the *Write* state, an internal counter keeps track of the bursts. When the last burst is received, it goes into *Idle* state.

Refresh control

An essential function of the SDRAM controller is to send the refresh command to the SDRAM to ensure the data is not lost. The refresh control logic is shown in figure 3.3. The main element is the refresh counter, which is used to determine when the controller needs to send the refresh command to the SDRAM. The counter counts up to a configurable number of cycles, DELAY_T_REF, which is set to accomplish the specified refresh period of the SDRAM. In the current configuration the counter is limited to 780 cycles to meet the refresh period at a frequency of 166 MHz.

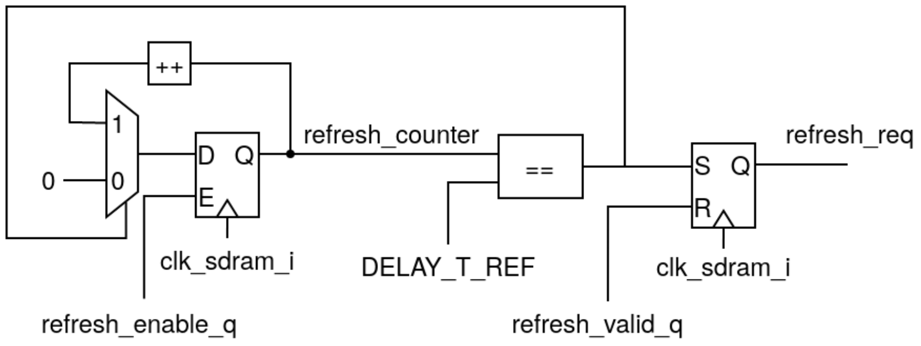


Figure 3.3: Schematic of the refresh control logic in the SDRAM controller.

The counter is enabled with the refresh_enable_q signal. Initially it is disabled to not generate refresh requests during the initialization of the SDRAM. After the initialization sequence, the counter is enabled to periodically make refresh requests. Each time the refresh counter reaches the counter limit a refresh request (refresh_reg) is issued. The refresh request keeps asserted until the controller sends the refresh signal, which is indicated with the refresh_valid_q signal.

Address generation

Figure 3.4 shows the decoding of the input address. Since the address can come from a different clock domain, an asynchronous FIFO is used to syncrhonize the address from the external clock (clk_i) to the clock of the controller (clk_sdram_clk_i), which operates at the same frequency as the SDRAM. When the input address is valid, indicated with the valid_i signal, and the FIFO is not full the address is written into the FIFO. The input address assumes words of 8 bits, so each postion refers to 1 byte of data, but on the SDRAM side the word size is not always of 1 byte and it depends on the architecture of each particular SDRAM chip. The input address is initially shifted to compensate for this mismatch. In

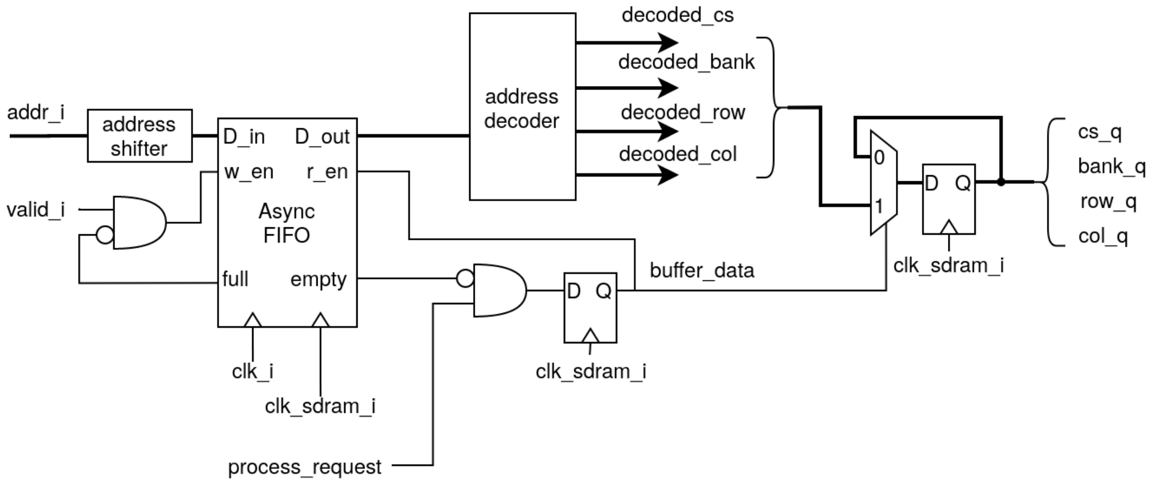


Figure 3.4: Schematic of the address processing logic in the SDRAM controller.

the current configuration of the controller only x8 and x16 SDRAM chips are supported. is shifted.

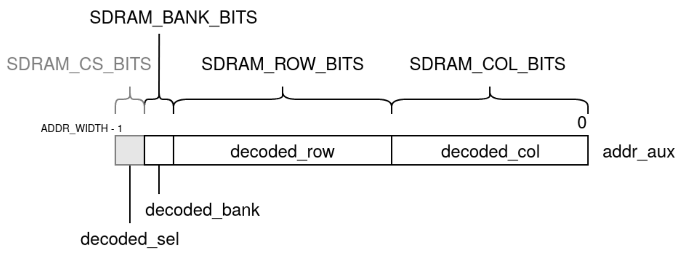


Figure 3.5: Address decoding in the SDRAM controller.

When request is ready to be processed, indicated with the process_request signal, and the FIFO is not empty then the address is read from the FIFO, decoded and stored into an internal buffer to be processed. The decoding is done by slicing the address as shown in figure 3.5, the lower bits of the address are assigned to the column, the following bits to the row, the next ones to the bank and the rest to the chip selects.

The number of bits assigned to the chip selects depends on the $log2(SDRAM\_CS\_WIDTH)$, which means that if there is only one chip select no bits of the address are assigned to it and this field is ignored.

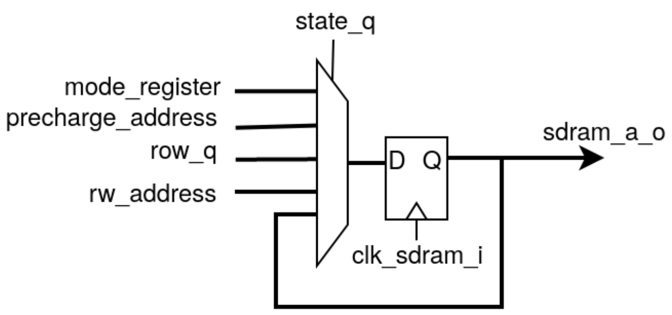


Figure 3.6: Schematic of the output address logic the SDRAM controller.

Figure 3.6 shows the address multiplexer that is used to select the right address depending on the current state of the FSM. The selected address is stored in an output buffer and sent to the SDRAM.

Data generation

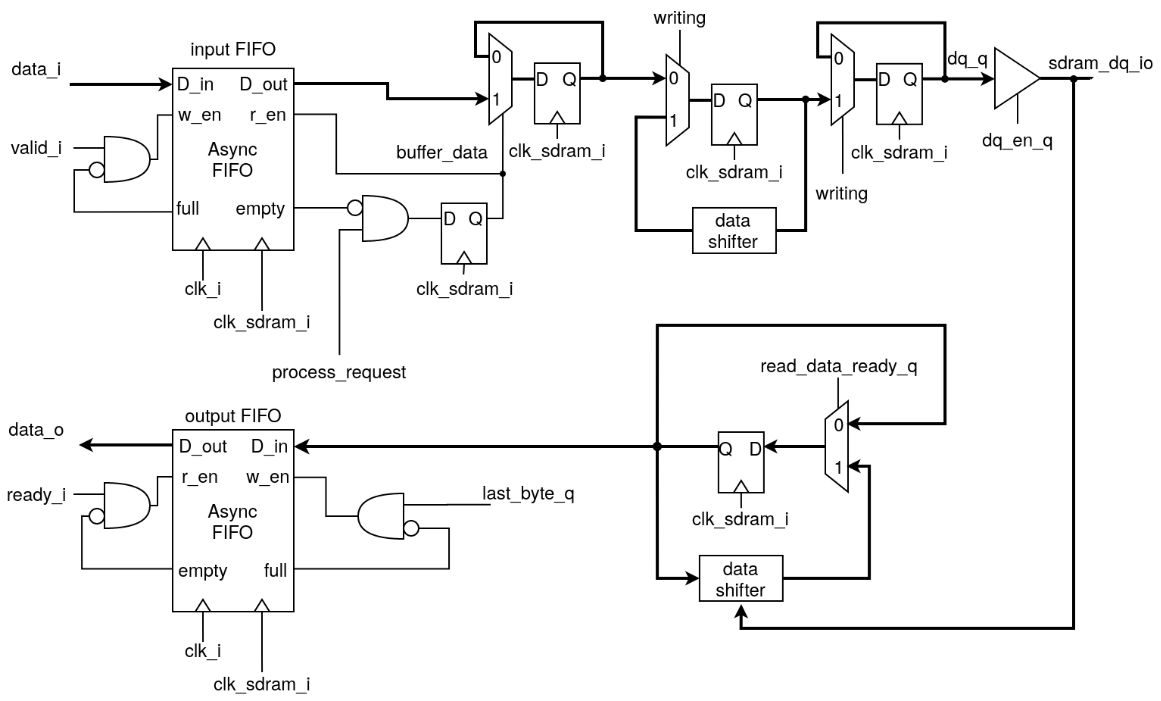


Figure 3.7: Data reading and writting control logic in SDRAM controller.

The control of input and output data is shown in figure 3.7. Similarly to the address, the input data is stored into the input FIFO when a valid request is received. When the controller is ready to process the request it is stored into and internal buffer.

SDRAM data width might be lower than the actual input data width. The whole data is not sent in a single transaction but divided into multiple burst. During a write operation input data is shifted on every burst, then only the lower bits are stored on the output buffer and sent to the SDRAM.

The reading process is similar. In this case there the input data from the SDRAM is stored into the higher bits of an internal buffer which is shifted after each burst. Once the last burst is sent, the data is stored into the output FIFO. From there it is sent to the master when it is ready to accept new data.

3.1.2 AXI interface

To implement the final AXI SDRAM controller an extra layer of logic is needed in order to convert the previous SDRAM controller into an AXI4-slave compatible device. This module is placed between the SDRAM controller and the external AXI, which means it has to properly match with both interfaces.

In the AXI protocol the width of the data, address and user signals are not defined. Such signals can have a different widths depending on every implementation. In order to make the AXI SDRAM controller more flexible, the AXI iterface has been parametrized so the width of the different signals can be defined when instantiating the controller accordign to the needs of the design.

Table 3.3 contains the parameters that can be modified in the current implementation of the AXI interface and their defaults values. In the AXI protocol, the id, address, user and data width can have variable widths depending on the implementation, the rest of the fields defined on the AXI protocol either have a fixed size or depend on the data width. Therefore those are defined as internal parameters inside the AXI interface module and can not be modified in order to simplify the configuration and avoid possible mistakes.

Table 3.3: Configurable parameters of the AXI interface

Parameter	**Default**	**Description**
ID_WIDTH	8	Width of the ID field of the AW, B, AR and R channels.
ADDR_WIDTH	32	Width of the address field in the AW and AR channels.
USER_WIDTH	1	Width of the USER field, present in all channels.
DATA_WIDTH	128	Width of the data.
SDRAM_CTRL_ADDR	26	Width of the address bus of the SDRAM controller. It must coincide with the actual width of the address bus in the SDRAM controller.
SDRAM_CTRL_DATA	128	Width of the data bus of the SDRAM controller. It must coincide with the actual with of the data bus in the SDRAM controller.

In the current implementation the input data with (DATA_WIDTH) must be the same as the SDRAM controller data width (SDRAM_CTRL_DATA). Also the width of the address in the AXI bus (ADDR_WIDTH) must be larger or equal than the address bus width in the SDRAM controller (SDRAM_CTRL_ADDR).

As shown in figure 3.1 there are 4 main blocks, the read and write control logic, the address generation logic and the FSM.

FSM

A diagram of the FSM is shown in figure 3.8. On reset the AXI controller goes into *Ready* state. During reset, according to the AXI protocol, the ready signals of the different AXI channels are driven low, when the reset is released the ready signals are enabled, indicating that new data can be accepted.

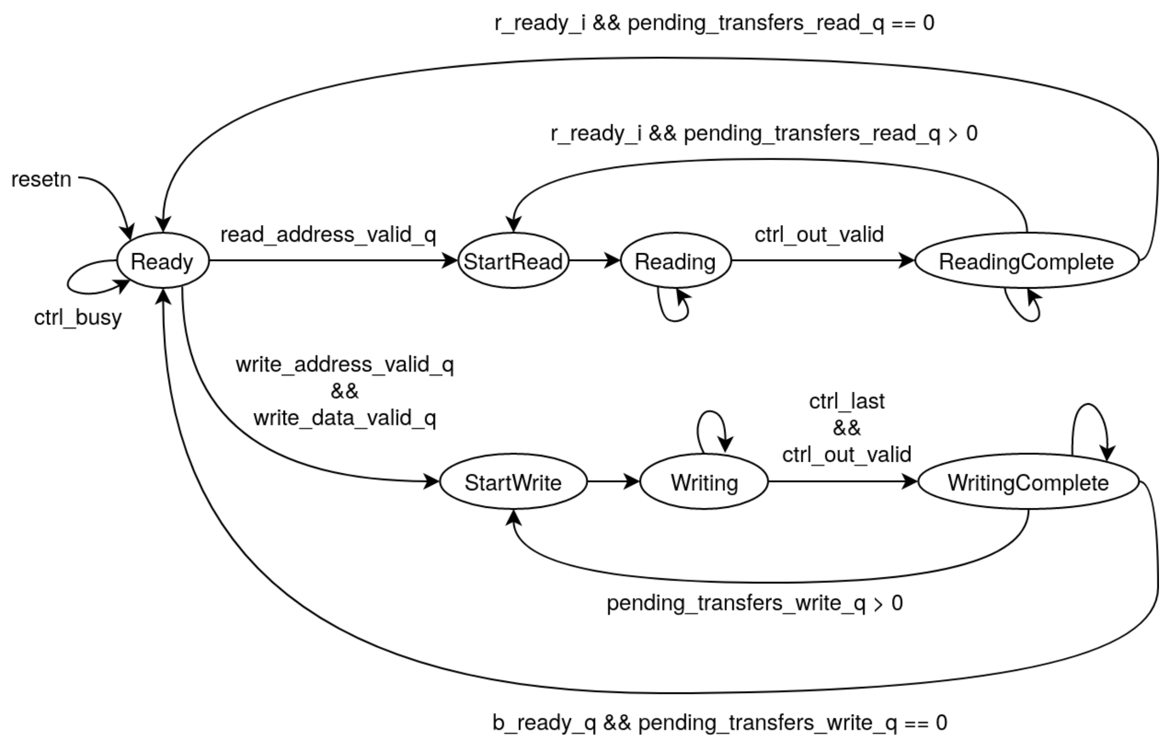

r_ready_i && pending_transfers_read_q == 0
r_ready_i && pending_transfers_read_q > 0
resetn
Ready
ctrl_busy
read_address_valid_q
StartRead
Reading
ctrl_out_valid
ReadingComplete
write_address_valid_q && write_data_valid_q
StartWrite
Writing
ctrl_last && ctrl_out_valid
WritingComplete
pending_transfers_write_q > 0
b_ready_q && pending_transfers_write_q == 0

Figure 3.8: Finite state machine diagram of the AXI interface module.

In *Ready* state the AXI controller waits for two conditions to be met:

- The SDRAM controller is ready to accept a request.
- A valid AXI request is recevied from the master.

When SDRAM controller is ready (ctrl_busy is low) and there is a valid read request (read_address_valid_q is high) then the state changes to *StartRead*. The *StartRead* state is used to send the request to the SDRAM controller, then immediately jump to *Reading* state.

In the *Reading* state, the AXI interface waits for a response from the SDRAM controller indicating the transfer has completed. Once the request is completed it jumps to the *ReadingComplete* state and remains there until the response is sent back to the AXI master. Then if the request contained multiple transfers it goes back to the *StartRead* state to process the next transfer, otherwise if the request is fully completed it goes to the *Ready* state.

Similarly, when in *Ready* state, the controller will jump to *StartWrite* state if a valid write request is received (both write_address_valid_q and write_data_q are high). With the current configuration read requests have preference over write requests.

Read control

Figure 3.9 shows the read control logic. It has an input buffer to store the request from the Address Read channel (ar_*). The request is stored as soon as it is received (ar_valid_i is asserted), as long as the ar_ready_o signal is enabled. As mentioned before, after reset ar_ready_o is enable so when a request is received it is stored in the input buffer, then

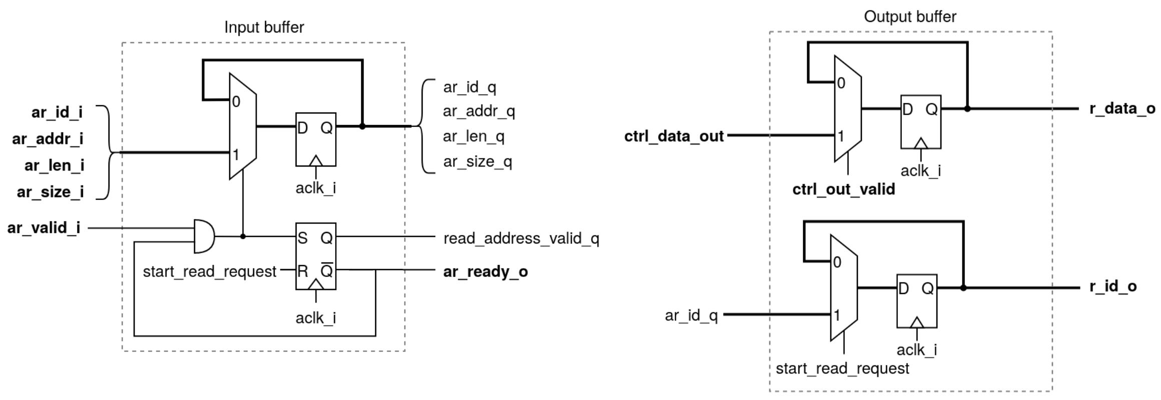


Figure 3.9: Schematic of input and output registers for a read operation in the AXI interface.

ar_ready_o is set to low so no more request can be accepted. This performs the handshake on the AR channel.

Since ready signals are already enabled before the valid signal is received, it takes only one cycle to do the handshake. Otherwise it would take at least two cycles, since an extra cycle would be required to enable the ready signal once the valid signal is detected.

When there is a valid request in the input buffer, the signal read_address_valid_q is enabled.

This configurations allows requests to be accepted at any time as long as the input buffer is empty, independently of the current state of the AXI interface or SDRAM controller. This way the handshake is produced as soon as possible which frees the channel so the master can perform other requests.

Once the request is sent to the SDRAM controller, indicated with the start_read_request signal, then the data is passed to the SDRAM controller. When this happens, the input buffer is emptied, ar_ready_o is enabled again and new requests can be accepted.

Figure 3.9 also shows the output buffer which stores the response that will be sent back to the AXI master. In a read transaction, the response is sent on the Read channel. Once the request is processed the id of the input request (ar_id_i) is passed to the id of the response (r_id_o). When the SDRAM controller has completed the request, the read data is also stored in the output buffer and the everything is ready to do the handshake with the master, which is managed by the FSM.

Write control

Figure 3.10 shows the write control logic. For one side the signals of the Address Write channel (aw_*) have the same behaviour as the ones from the Adress Read channel (ar_*) in the read control logic. Additionally there is the Write channel, which contains the data to be written.

The functionality of the Write channel is slightly different. As before, as soon as a valid request is received is is stored in the input buffer. The difference with respect to the Address Write channel is that this time the buffer is emptied after each transfer (when start_write_transfer is enabled) so, if a request contains multiple transfers all of them are registered and passed into the SDRAM controller.

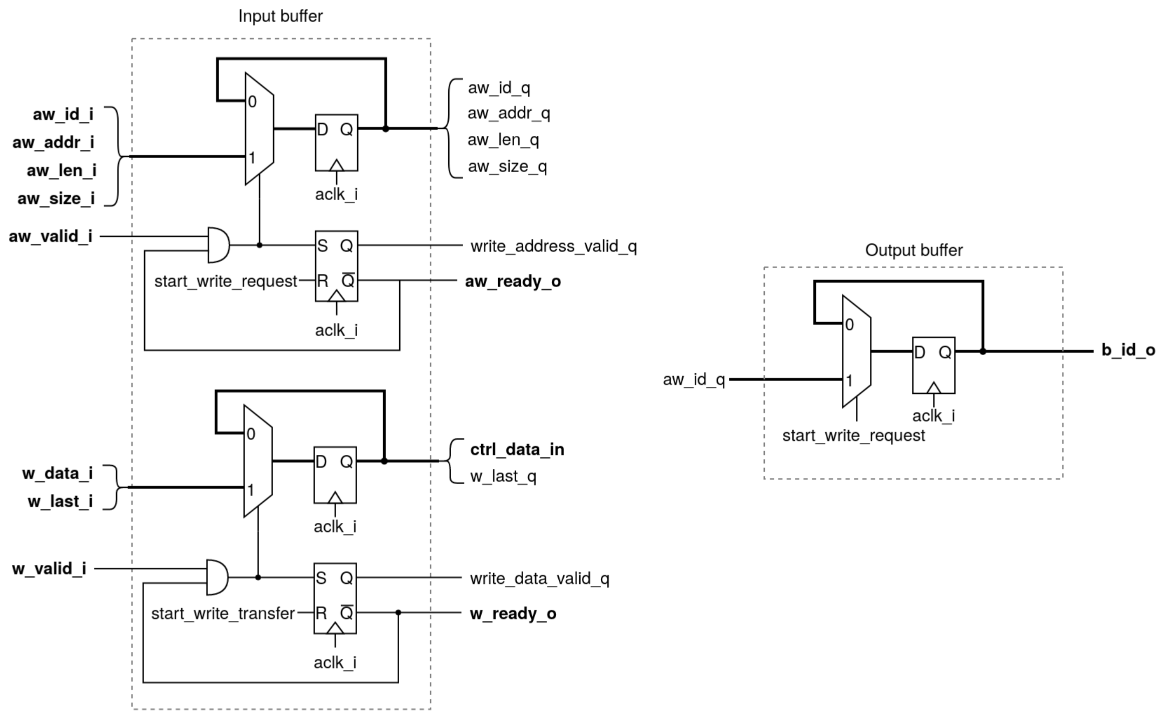


Figure 3.10: Schematic of input and output registers for a write operation in the AXI interface.

Address generation

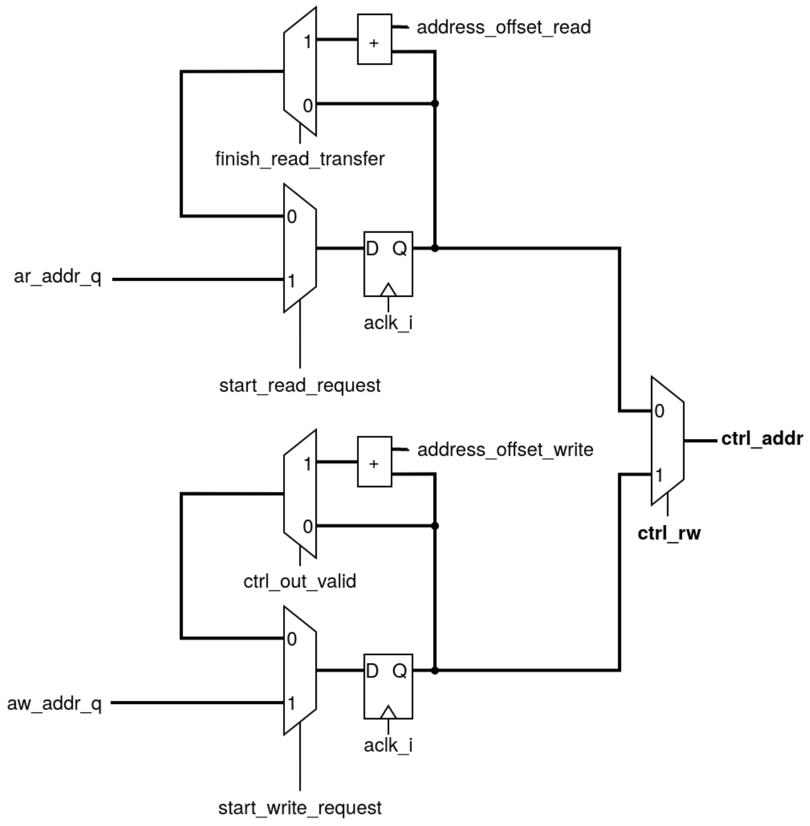


Figure 3.11: Schematic of address generation logic in the AXI interface.

A single AXI request can contain multiple transfers. For a read request this means each transfer must return a different read value while for a write request is means that each transfer will carry different data to write. On the other hand, addresses are sent only once per request. The AXI protocol specifies that the slave has to generate the addresses from the request data. Figure 3.11 shows the logic to generate the addresses. This addresses will be the ones sent to the SDRAM controller.

For one side in an AXI request it is specified the length of the transaction, which means the number of transfers to process, also specifies the size of each transfer and the type of transaction. Currently the controller only supports incremental bursts, which means that the address increments on each transfer.

To obtain the next address, an address offset is added to the current address. This offset is obtained from the size field in the AXI request, which specifies the number of bytes per transfer.

3.1.3 Synchronization

While the SDRAM memory will work at one frequency, the core that will be using the controller could work at another one. It is a common practice that processors work with different clock domains to adapt to different peripherals while allowing the main core to operate at a higher frequency. Such is the case with the DRAC chip, while the core is expected to work at a high frequency, other peripherals such as the external memory, will operate at a lower frequency. For instance, the SDR SDRAM can work at a maximum frequency of 200 MHz, depending on the model.. Since the core needs to communicate with the memory, and other peripherals working at different frequencies, this requires to pass from one clock domain to another, which is known as clock domain crossing (CDC).

With a CDC there is the risk to introduce metastability to the system. To avoid that proper synchonization mechanisms should be used. There are options available, from a simple synchronizer based on two flip-flops to more complex methods like asynchronous FIFOs or handshake mechanisms. Choosing one method or another can have some impact in system performance both in terms or reliability, latency, area and power [19].

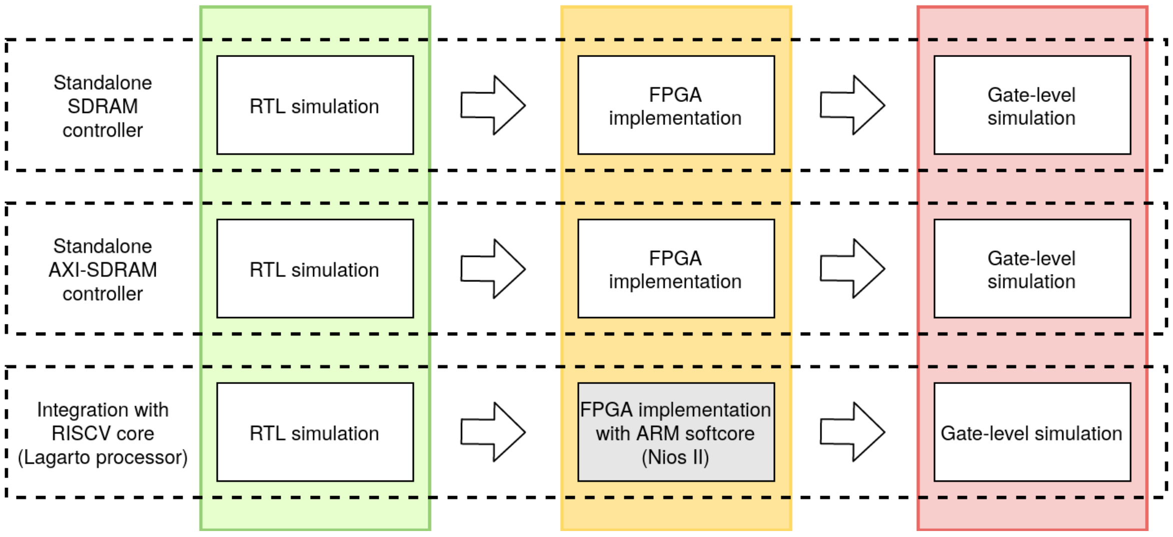
For this design, an asynchronous FIFO has been used to synchronize both domains. The implementation is based on the design from Cummings [20].

3.2 Verification

In the previous section the implementation of the design has been described, but that's just the first part of the project. The goal of this project is to design an AXI-SDRAM interface IP to be intergrated in a RISC-V core, which will later be built into a chip. Therefore, the design must be properly verified to ensure it will work once implemented on silicon.

The implementation of the design is divided into two parts: the SDRAM controller and the AXI interface. Similarly, the verification of the design has been divided into three different stages:

- **Verification of the standalone SDRAM controller:** The first stage in the design flow was the design on the SDRAM controller, without the AXI interface. In a similar way,



Standalone SDRAM controller
RTL simulation
FPGA implementation
Gate-level simulation
Standalone AXI-SDRAM controller
RTL simulation
FPGA implementation
Gate-level simulation
Integration with RISCV core (Lagarto processor)
RTL simulation
FPGA implementation with ARM softcore (Nios II)
Gate-level simulation

Figure 3.12: Stages of the verification flow.

the first step in the verification process it to test the functionality of the standalone SDRAM controller to be sure it is able to properly send requests to an SDR SDRAM memory.

- **Verification of the AXI SDRAM controller:** Once there is a functional SDRAM controller that can successfully communicate with an SDRAM, an AXI interface can be built on top of that. At this point the functionality of the whole AXI-SDRAM interface can be verified.

- **Verification of the integration with the RISC-V core:** The goal of this project is to implement an AXI-SDRAM interface IP to be used in a RISC-V core. Then, the last verification step is to integrate the AXI-SDRAM interface IP with the RISC-V core and verify the functionality of the whole SoC.

At the same time, each stage is divided into three steps: RTL simulation, FPGA implementation and gate-level simulation. RTL simulations has been done to verify the design while it's being designed. Once the design is completed and passes the simulation its implemented on FPGA to test it's behavior with an actual SDRAM chip. Then, if both RTL simulation and FPGA implementation tests are successful the design is synthetized. After synthesis a gate-level simulation is performed.

Figure 3.12 shows the different steps. The integration with the RISC-V core has not been tested on FPGA, instead what has been done is to test the AXI SDRAM controller with an ARM softcore both of them implemented on the same FPGA.

The next sections provide a more detailed view of the different steps followed to verify the design.

3.2.1 Behavioral RTL simulation

RTL simulation has been done using both Modelsim from Intel (formerly Altera) and Xcelium from Cadence. In the testbench an RTL behavioral model of an SDR SDRAM from Micron [21] has been used.

The same way SDRAMs come in multiple configurations, the behavioral model of the SDRAM can be configured to match different types of SDRAM. For this tests the model used has been configured to match the MT48LC16M16A2 part from Micron [8], which is a 256 Mb x16 SDRAM that can work up to 166 MHz.

For commodity, the default values of the parameters in the SDRAM controller, including the timing parameters, has been set to work with this memory configuration.

The development of the testbenches is done in parallel with the development of the RTL. RTL simulations are used to find errors and ensure the design operates accordingly to what is expected. Once a minimal functionality is implemented, simulations are also useful to test if the design still behaves as expeted when when additional features are implemented.

The testbench instantiates the DUT, the SDRAM model and the additional components to generate the stimulus. The DUT can be either the SDRAM controller, the full AXI-SDRAM controller or the whole RISC-V Soc, depending on the veritfication stage. While the concept is the same the testbench is slightly different for the different stages: SDRAM controller, AXI-SDRAM controller and integration with RISC-V. Each case will be reviewed separately.

Standalone SDRAM controller

SDRAMs have a set of timing requirements that must be met in order to ensure they proper operation. The same behavioral model of the SDRAM from Micron that is used in the test-bench contains timing checks that verify if the main timing requirements are met. Moreover, the testbench has to verify if the right data is written to the right address and also that the refresh sequence is properly performed, since the SDRAM model does not checks it.

To verify the functionality of the SDRAM multiple versions of the testbench has been used:

- Single write followed by a single read: basic test to verify the basic functionality of the controller and see if it can properly write and read data from memory.
- Multiple writes followed multiple reads: similar to the previous test but a bit more robust. Since multiple request are perfomed it can be tested if the SDRAM controller properly decodes the addresses and is able to read and write to different banks.

To verify if the test passes or not, the read values are compared with the written ones. The test passes if the read values are the same as the written ones.

The main purpose of the tests is to verify the functionality of the controller: see that it can write and read from memory. Also to ensure that the timing requirements are met and the refresh sequence works as expected. Additionally, RTL simulation can be used to obtain some metrics such as the latency and bandwidth of the SDRAM controller.

Standalone AXI-SDRAM controller

Similarly as with the SDRAM controller testbench, to verify the behaviour of the AXI SDRAM controller both model of an SDR SDRAM and a model of an AXI master device are used, figure 3.13 show the testbench setup.

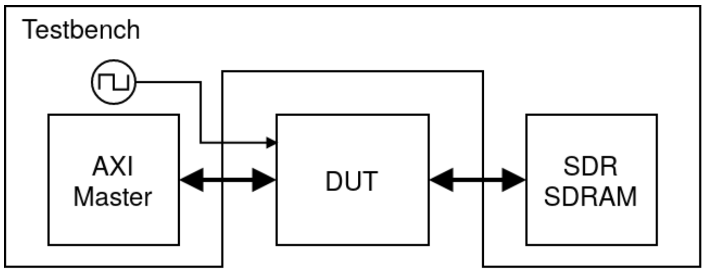


Figure 3.13: Testbench compnents in the AXI-SDRAM controller verification.

The AXI master module of the testbench has been implemented following the AXI4 specifications. It was implemented using SystemVerilog using non-synthetizable structures, since the purpose was to use it only for the simulation. In this case it showed different behaviour when simulated with the different simulators (ModelSim and Xcelium) due to the race condition issues mentioned in section 3.2. Such issues has been solved by adding delays on the signals of the AXI master module to ensure they behave the same way on both simulators.

As has been done with the SDRAM controller, there are multiple versions of the test-bench:

- Single write and read: used to test the proper communication between the AXI and SDRAM interfaces.
- Multiple writes and multiple reads: which is used both to test the functionality and also to measure the bandwidth and latency.
- Use multiple SDRAM models: this additional test is used to verify the operation of the chip select and check if the AXI SDRAM controller can write to multiple chips of memory.

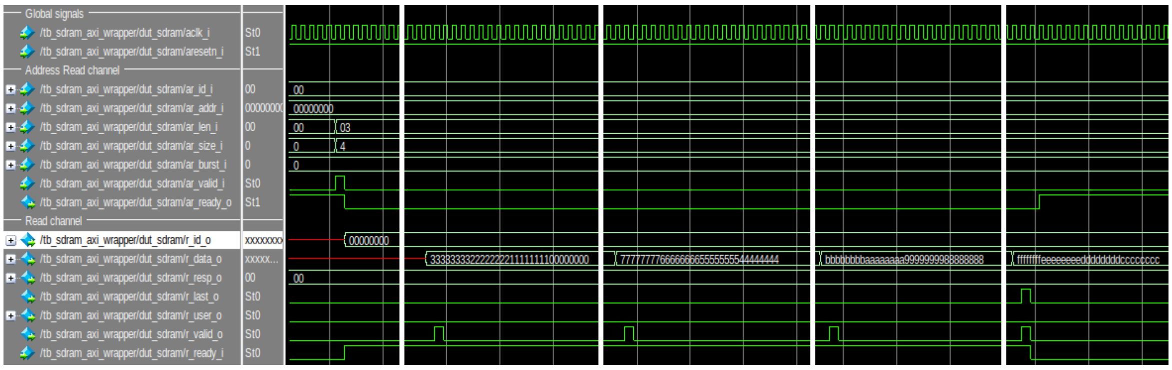

Global signals
/tb_sdram_axi_wrapper/dut_sdram/aclk_i
St0
/tb_sdram_axi_wrapper/dut_sdram/aresetn_i
St1
Address Read channel
/tb_sdram_axi_wrapper/dut_sdram/ar_id_i
00
/tb_sdram_axi_wrapper/dut_sdram/ar_addr_i
00000000
/tb_sdram_axi_wrapper/dut_sdram/ar_len_i
00
03
/tb_sdram_axi_wrapper/dut_sdram/ar_size_i
0
4
/tb_sdram_axi_wrapper/dut_sdram/ar_burst_i
0
/tb_sdram_axi_wrapper/dut_sdram/ar_valid_i
St0
/tb_sdram_axi_wrapper/dut_sdram/ar_ready_o
St1
Read channel
/tb_sdram_axi_wrapper/dut_sdram/r_id_o
xxxxxxxx
00000000
/tb_sdram_axi_wrapper/dut_sdram/r_data_o
xxxxx...
33333333222222221111111100000000
77777777666666665555555544444444
bbbbbbbbaaaaaaaa9999999988888888
ffffffffeeeeeeeeddddddddcccccccc
/tb_sdram_axi_wrapper/dut_sdram/r_resp_o
00
/tb_sdram_axi_wrapper/dut_sdram/r_last_o
St0
/tb_sdram_axi_wrapper/dut_sdram/r_user_o
St0
/tb_sdram_axi_wrapper/dut_sdram/r_valid_o
St0
/tb_sdram_axi_wrapper/dut_sdram/r_ready_i
St0

Figure 3.14: Capture of an AXI read transaction obtained by simulation.

Figure 3.14 shows a read transaction. The master sends the transfer information on the AR channel, including the number of elements per transfer and the size. The slave is ready (ARREADY is high) and the transfer starts when a valid request comes in (ARVALID is high) then. A single address is transmitted, the slave increments the address after each transfer and there is a handshake after each transfer, at the end, once the last element is transfered the WLAST bit is asserted.

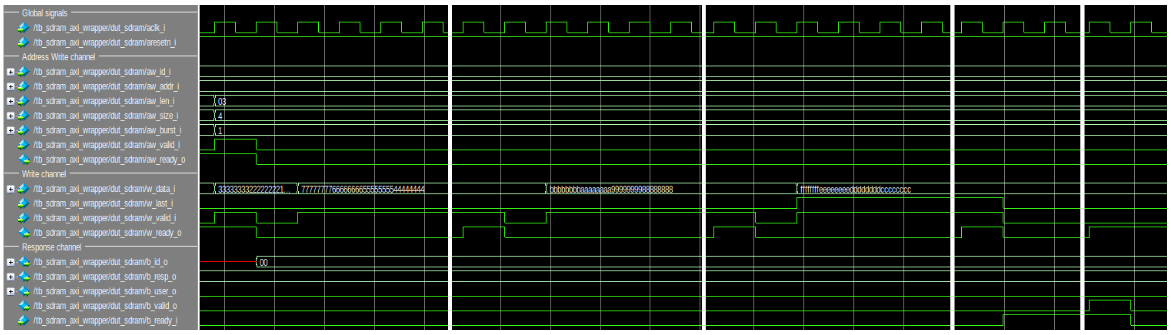

Global signals
/tb_sdram_axi_wrapper/dut_sdram/aclk_i
/tb_sdram_axi_wrapper/dut_sdram/aresetn_i
Address Write channel
/tb_sdram_axi_wrapper/dut_sdram/aw_id_i
/tb_sdram_axi_wrapper/dut_sdram/aw_addr_i
/tb_sdram_axi_wrapper/dut_sdram/aw_len_i
/tb_sdram_axi_wrapper/dut_sdram/aw_size_i
/tb_sdram_axi_wrapper/dut_sdram/aw_burst_i
/tb_sdram_axi_wrapper/dut_sdram/aw_valid_i
/tb_sdram_axi_wrapper/dut_sdram/aw_ready_o
Write channel
/tb_sdram_axi_wrapper/dut_sdram/w_data_i
/tb_sdram_axi_wrapper/dut_sdram/w_last_i
/tb_sdram_axi_wrapper/dut_sdram/w_valid_i
/tb_sdram_axi_wrapper/dut_sdram/w_ready_o
Response channel
/tb_sdram_axi_wrapper/dut_sdram/b_id_o
/tb_sdram_axi_wrapper/dut_sdram/b_resp_o
/tb_sdram_axi_wrapper/dut_sdram/b_user_o
/tb_sdram_axi_wrapper/dut_sdram/b_valid_o
/tb_sdram_axi_wrapper/dut_sdram/b_ready_i
03
4
1
33333333222222221...
77777777666666665555555544444444
bbbbbbbbaaaaaaaa9999999988888888
ffffffffeeeeeeeeddddddddcccccccc
00

Figure 3.15: Capture of an AXI write transaction obtained by simulation.

Figure 3.15 shows a write transaction which is part of the testbench used to verify the functionality of the design. In this case both AWVALID and WVALID signals arrive at the same time, although this is not required. The transaction starts after both the AW and W channels had performed the handshake. The AW channel contains the transaction information, in this case it's a burst transaction of 4 elements and each element has 128 bits and the address is incremented after each transfer. It can be seen how there is a total of 4 handhsakes in the W channel, one per transfer and that in the last one the WLAST signal is asserted. After the last transfer the controller sends the response through the response channel.

Integration with RISC-V

The testbench that is used to verify the functionality of the RISC-V core is based on the setup provided by lowRISC. The verification is based on the execution of different instructions of the ISA to test functionality of the RISC-V core, as well as the different memory hierarchies. LowRISC provided several versions of their testbench for different simulation environments, one of them for HDL simulators, such as Xcelium or ModelSim, which was written in SystemVerilog.

The verification setup consists of several tests, referred to as ISA-tests, each one of them evaluates an instruction of the RISC-V ISA. An ISA-test is a small piece of assembler code that first performs any required initialization needed by the core, then performs the operation and finally it evaluates if the operation completed successfully. Before the simulation, the ISA-test must be compiled to obtain the machine code that can be written into the main memory.

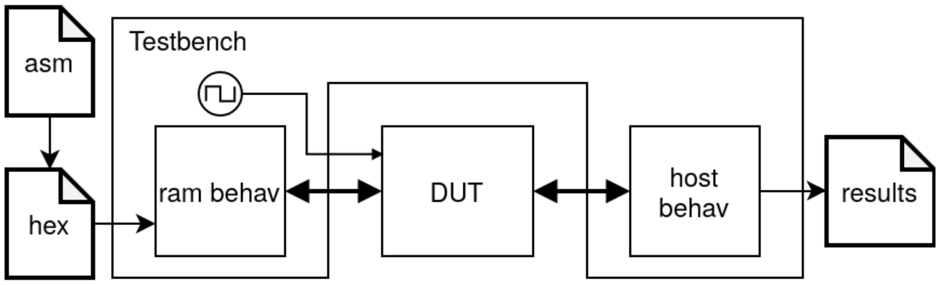


Figure 3.16: Initial testbench setup used to verify the preDRAC SoC.

The initial testbench setup used in the preDRAC is shown in figure 3.16. It is composed of three main elements:

- DUT (device under test), which corresponds to the ASIC of the preDRAC SoC.
- ram_behavioral, a model implemented in C++ which is used to emulate the main DDR memory. It is connected to the DUT through an AXI interface.
- host_behavioral, an interface also implemented in C++ that communicates with the core used to determine whether the test passes or not.

At the beginning of the simulation the ISA-test is loaded into the main DDR memory and from there once the core starts running it fetches the instructions from memory and executes the tests. ram_behavioral and host_behavioral are interfaces written in SystemVerilog which include Direct Programming Interface (DPI) calls that interact with the actual ram and host models that are the written in C++.

The original testbench was modified to properly load the memory when using the Xcelium simulator. To automatize the execution of the ISA-tests, a simulation script was created. This script performs the following operations for each ISA-test:

- reset the simulation
- load ISA-test into memory
- run simulation until it stops (either because it reaches the end or a predefined number of cycles, set to avoid the simulation stucks if there are errors in some of the test)
- write status (success or fail) to external file

Figure 3.19 shows the waveforms obtained with the simulation of one ISA using the initial preDRAC testbench. This is provided as reference since it shows the expected behavior of the system.

The AXI interface had too many pins and it couldn't be included in the final preDRAC chip, to get over this issue the AXI interface was divided into multiple packets and serialized to reduce the number of pins. This requires an extra logic to serialize and deserialize the requests on both ends. This has a negative impact on the performance since it slows the accesses to memory. The final testbench used in the preDRAC included this logic as shown in figure 3.17.

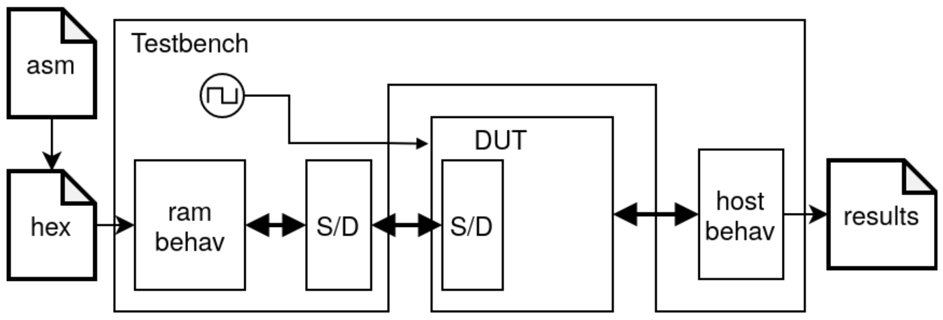


Figure 3.17: Final testbench setup used to verify the preDRAC SoC.

This was the original simulation environment used in the preDRAC chip. With the intergration of the AXI-SDRAM interface IP, the testbench had to be modified. The actual testbench setup used for the verification of the integration is shown in figure 3.18 The ram_behavioral_model has been replaced with the SDRAM behavioral from Micron model used in previous tests, which is connected to the preDRAC SoC through the AXI-SDRAM interface IP. The host_behavioral_model was left since it is used to determine the result of the test.

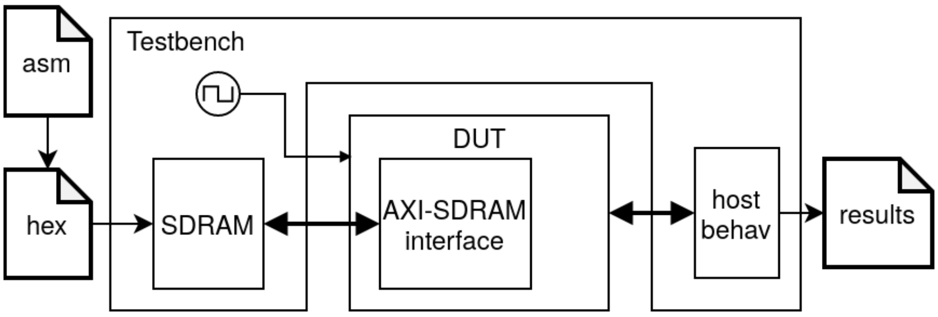


Figure 3.18: Testbench setup used to verify the intergration of the AXI-SDRAM interface IP with the preDRAC SoC.

The execution of the test had to be slightly modified. In this case the test is loaded into the internal registers of the SDRAM behavioral model using a simulator command. The rest of the test is executed as before.

Figure 3.20 shows the behavior of the preDRAC SoC, previous to the integration of the AXI-SDRAM interface. This particular test takes around 94 μs to complete. On the other hand, figure 3.21 shows the behavior of the preDRAC SoC after integrating the AXI-SDRAM interface IP. In this case the test takes around 124 μs, but as it can be seen, during the first 100 μs there is no operation since the controller is performing the initialization sequence.

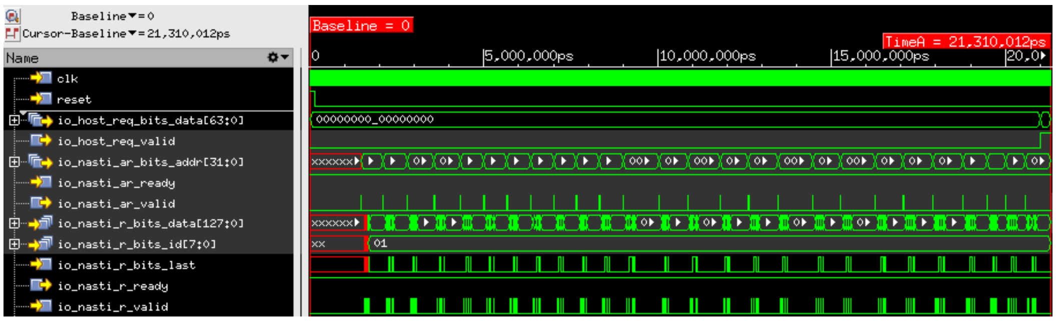


Figure 3.19: Extract of the simulation of the initial preDRAC SoC when running the rv64ui-p-addw ISA-test.

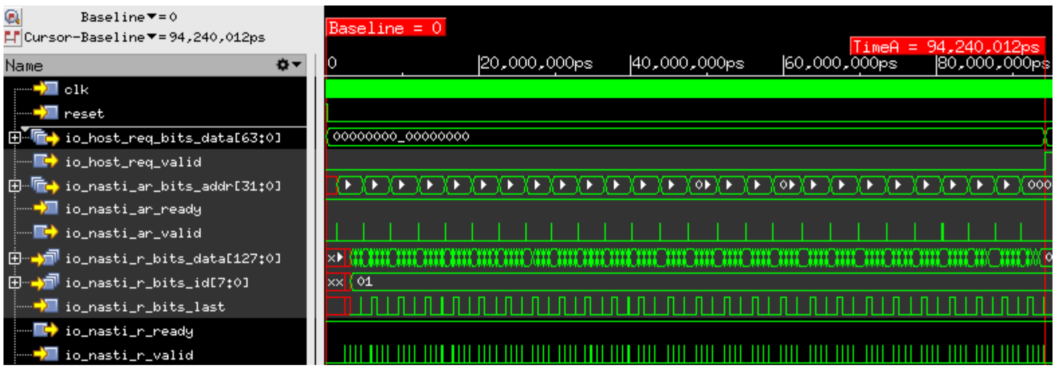


Figure 3.20: Extract of the simulation of the preDRAC version that was sent to tapeout running the rv64ui-p-addw ISA-test.

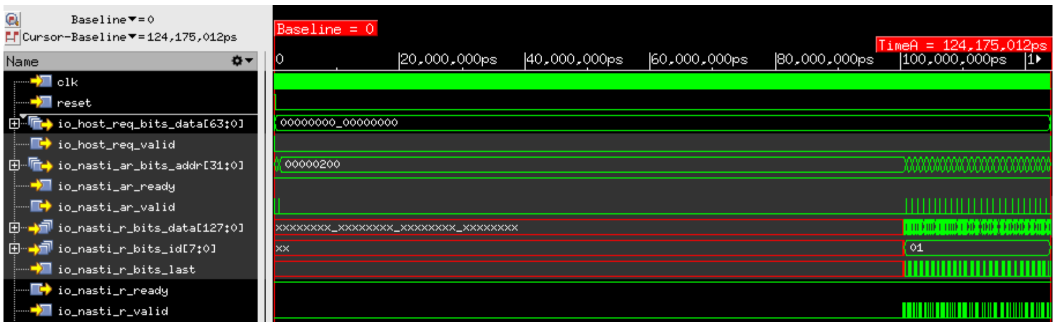


Figure 3.21: Extract of the simulation of the preDRAC SoC after the integration of the AXI-SDRAM interface IP running the rv64ui-p-addw ISA-test.

3.2.2 FPGA implementation

To emulate the design in hardware it has been used the DE0-Nano development board [22]. This board has a Cyclone IV FPGA and includes a SDR SDRAM module.

The memory included in the DE0-Nano board (IS42S16160G-7TLI [9] from ISSI) has 32 MB with a data bus of 16-bit data. Internally it is configured with 4 banks and each bank is organized in 8192 rows by 512 columns with elements of 16 bits.

This module can work up to 143 MHz with a CAS latency of 3 clock cycles, or up to 133 MHz with a CAS latency of 2 clock cycles. For this test, the SDRAM frequency has been set to 133 MHz, therefore a CAS latency of 2 has been selected. Table 3.4 shows how the parameters of the controller are configured to work with the SDRAM included in the DE0-Nano board. Those parameters that do not depend on the SDRAM are left with their default value.

Standalone SDRAM controller

To verify the functionality of the SDRAM controller in the FPGA a synthetizable test module has been written in verilog. This test has a simple state machine which generates the commands that are sent to the SDRAM controller. The test then writes data into memory and reads it. Both data and address are generated with an internal counter in a way that the data corresponds to the address. With this setup the read data can be put to the external

Table 3.4: Parameters of the SDRAM controller for the DE0-Nano board

Parameter	Value	Comment
SDRAM_DATA_WIDTH	16	Width of the data bus.
SDRAM_ADDR_WIDTH	13	Width of the address bus.
SDRAM_BANK_WIDTH	2	Number of bits for the external SDRAM bank.
SDRAM_ROW_BITS	13	Required to access the 8192 rows.
SDRAM_COL_BITS	9	Required to access the 512 columns.
SDRAM_BANK_BITS	2	Required to access the 4 banks.
SDRAM_ELEMENT_SIZE	2	Each element has 2 bytes.
DATA_WIDTH	-	Non relevant, independant of SDRAM module.
ADDR_WIDTH	26	Required number of bits to access all elements in the SDRAM.
CAS_LATENCY	2	Select CAS latency of 2 to work up 133 MHz.
ASYNC_FIFO	-	Independant of SDRAM module.

leds in the board to have visual feedback, since the data corresponds to the address the read values observable on the leds should reproduce a counter. This test in FPGA is meant to check the basic operation of the controller with a real SDRAM.

The Qsys tool (Project planner in newer verions of Quartus) has been used to instantiate and configure the PLL to generate a 133 MHz clock. Then a top wrapper has been written to instantiate both the PLL and the test module so it can be synthetized and programmed into the FGPA.

Standalone AXI-SDRAM controller

The setup used to verify the funcionality of the AXI controller in the FPGA is similar as the previously described. But in this case the state machine generates AXI requests.

An additional test has been done in FPGA using an embedded Nios II softcore. Nios II is a 32-bit RISC based core which can be instantiated using the Qsys tool of Quartus (similarly as the PLL is instantiated). A new project has been created which generates an instance of the Nios II core, the PLL, an instance of the AXI-SDRAM controller, a JTAG to communicate with the main PC and some RAM memory to store the program.

By default Quartus uses Avalon interfaces to interconnect the different IPs, but it is able to automatically generate the required logic to adapt the AXI4 interface to Avalon.

Once the project was built, a C++ program was written. This program write and read from the memory locations associated to the AXI-SDRAM controller to test its proper operation.

3.2.3 Gate-level simulation

The simulation setup for the gate-level simulations is similar as the described previosly in the Behavioral RTL simulation section (3.2.1), with some differences:

- The DUT has been replaced with the netlist obtained from synthesis. The netlist is made of standard cells. To simulate the design a library containing the definition of the standard cells has to be loaded into the simulator.
- The SDF file, also obtained from the synthesis, has been used to annotate the delays of the different cells.
- The standard cells models behave as the actual gates that will be implemented on silicon, they have propagation delays and also setup and hold time requirements. The rest of the testbench still has an ideal behavior, which means synchronous signals can change with the clock. If this happens on the signals that go into the netlist, it can cause timing violations.

  To emulate the behavior of a real system and prevent this timing violations, small delays are added on the testbench signals that go to the input ports of the DUT. It is not necessary to add delays on the outputs of the DUT as they will already be delayed since they come from the netlist.

Gate-level simulations has been run in the Xcelium simulator. Figure shows a waveform obtained from the gate-level simulation of the AXI-SDRAM controller, where are visible the previosly mentioned delays.

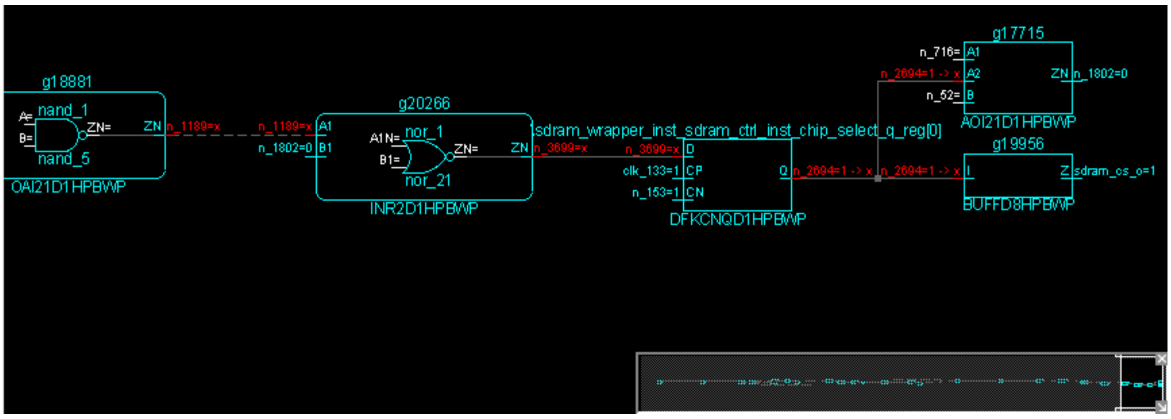


Figure 3.22: Capture of the schematic view in Xcelium while debuging an x-propagation issue.

Initial gate-level simulations were not successful as some signals were undetermined and that caused the simulation to fail. The source of this kind of problems is usually a register that is not properly reset and is treated as an 'X' by the simulation, from there the 'X' propagates through the system until it eventually reaches control signal and causes the simulation to fail.

The way to fix the errors derived from the x-propagation is to find the source of the indetermination and make sure it has a defined value after reset. It is an iterative process: first it must be found the first signal that gets undetermined, then get the drivers of that signal and keep tracking back the 'X' untill the original register is found. Figure 3.23 is an example of a waveform with an x-propagation error. Additionaly in figure 3.22 is shown the schematic view from Xcelium, which can be useful to trace back the drivers of a signal while seeing the gates it goes through. The image shows the last gates, while in the lower left corner it can be seen how those gates are only part of a much longer chain.

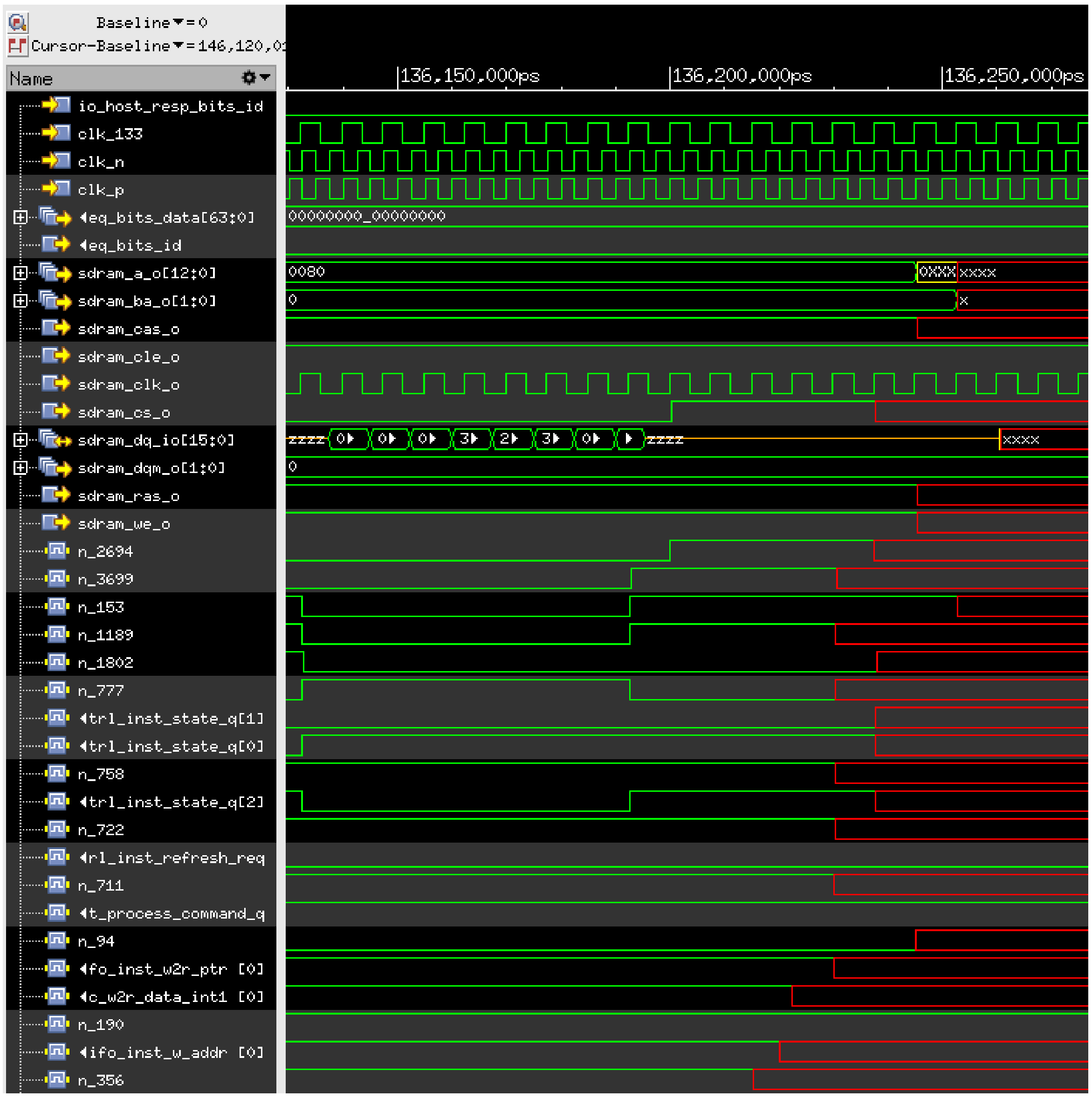


Figure 3.23: Capture of a waveform while debuging an x-propagation issue.

3.3 Synthesis

After implementation and to obtain a netlist to do the gate-level simulations, the design has to be synthetized. This has been done with Genus from Cadence.

The synthesis has been done using TSMC 65 nm technology. A multi-corner flow has been done using 10-track regular Vt cells, and three corners: typical, best and worst case. Table 3.5 summarizes the different conditions for each corner.

Table 3.5: Summary of standard cell corner conditions.

Corner	Conditions
Typical	1.2 V and 25C
Fast	1.32 V and 0C
Slow	1.08 V and 125C

In the synthesis two clocks has been specified, a clock of 600 MHz for the logic of the AXI interface logic and a clock of 200 MHz for the internal logic of the SDRAM controller.

3.4 Power analysis

To estimate the power of the design the Cadence toolflow has been used [23]. Cadence offers different tools to obtain analyze the power at different stages of the design:

- Joules is a power analysis tool which can estimate the power of the design on its early stages. it is primarly intered for RTL power analysis but it can also be used wth a gate-level netlist.
- Voltus is a power analysis tool used on the final stages of the design, after place and route, to validate the power requirements of the design a as sign-off step previous to tapeout.

This project only covers the initial stages in the ASIC design process, therefore Joules is the tool that has been used to perform the power analysis. Additionally, the Xcelium simulator has been used to obtain the switching activity before doing the actual power analysis.

The design that has been analyzed is the whole AXI-SDRAM interface IP. The following sections describe the conditions used.

3.4.1 Extracting activity data

There are different ways to obtain the switching activity information needed to perform an accurate power estimation. One way is to let the power analysis tool to estimate the switching activity based on the clock definition and a default switching probability. Since this estimation might not be realistic it is also possible to annotate switching activity obtained from a simulation, which offers a more accurate estimation based on actual activity of the design.

Switching activity information can be obtained from an RTL or gate-level simulation in different formats:

- **VCD** (Value Change Dump): contains full waveform data of the signals in the design and offers a complete representation of the design activity, which can also be used to analyze the quality of the switching activity information. Since this format contains many data the size of the file generated file is large, specially if there are many signals in the design.

- **SAIF** (Switching Activity Interchange Format): contains toggle information of the signals in the design and gives statistical information about the activity and state of the nets in the design: time spent on each state ('1', '0' or 'x') and the toggle count (number of transitions)
- **TCF** (Toggle Count Format): this is a Cadence proprietary format similar to SAIF.
- **SHM** (Simulation History Manager): this is a Cadence proprietary format similar to VCD.

The purpose of obtaining switching activity is to use it later to perform a more accurate power analysis. The stimulus used to generate the swithing activity should reflect the actual operation of the design.

In the testbench used the reset is initially applied, then a total of 50 write request are applied, followed by the same number or reads. Finally the simulation is run for 100 μs to capture the activity in idle state. Each request is a burst with 4 transfers.

In the stimulus two clocks are generated, a clock of 600 MHz that acts as a main clock an it is used in the logic of the AXI interface and a clock of 166 MHz for the logic of the SDRAM controller.

Figure 3.24 shows the waveforms of AXI-SDRAM interface ports that have been obtained when using the described stimuli.

After running the simulation the swithcing activity of the whole design, including internal registers is extracted in SHM format, which is a Cadence propietary format similar to VCD, so it contains both transitions and timing information.

3.4.2 Joules power analysis

As mentioned earlier, Joules is a power analysis tool inteneded for RTL power analysis [24]. Power analysis flow is divided into two stages, an initial setup and the power analysis itself.

Setup

During setup the RTL, libraries and design constraints are loaded into Joules so the design can be elaborated. This process is similar to the elaboration performed by Genus during the synthesis flow, in fact both elaborations take the same inputs (RTL, lib, lef, SDC) and can be done using the same script.

Once the design is loaded and elaborated it has to be mapped. Joules support all commands, options and attributes used by Genus for synthesis. On top of that, Joules has an additional command, power_map, to map the design. This additional command in the power analysis flow is used to perform a faster but still accurate synthesis of the design which can later be used to analyze the power of the design. The speedup obtained when using the power_map command is achieved by partitioning the design, parallelizing the synthesis of such partitions over multiple CPUs and relaxing the timing constraints in timing dense areas.

After mapping, and before the power analysis, the switching activity data can be annotated to the design. In order to have an accurate result it's important to check that a high

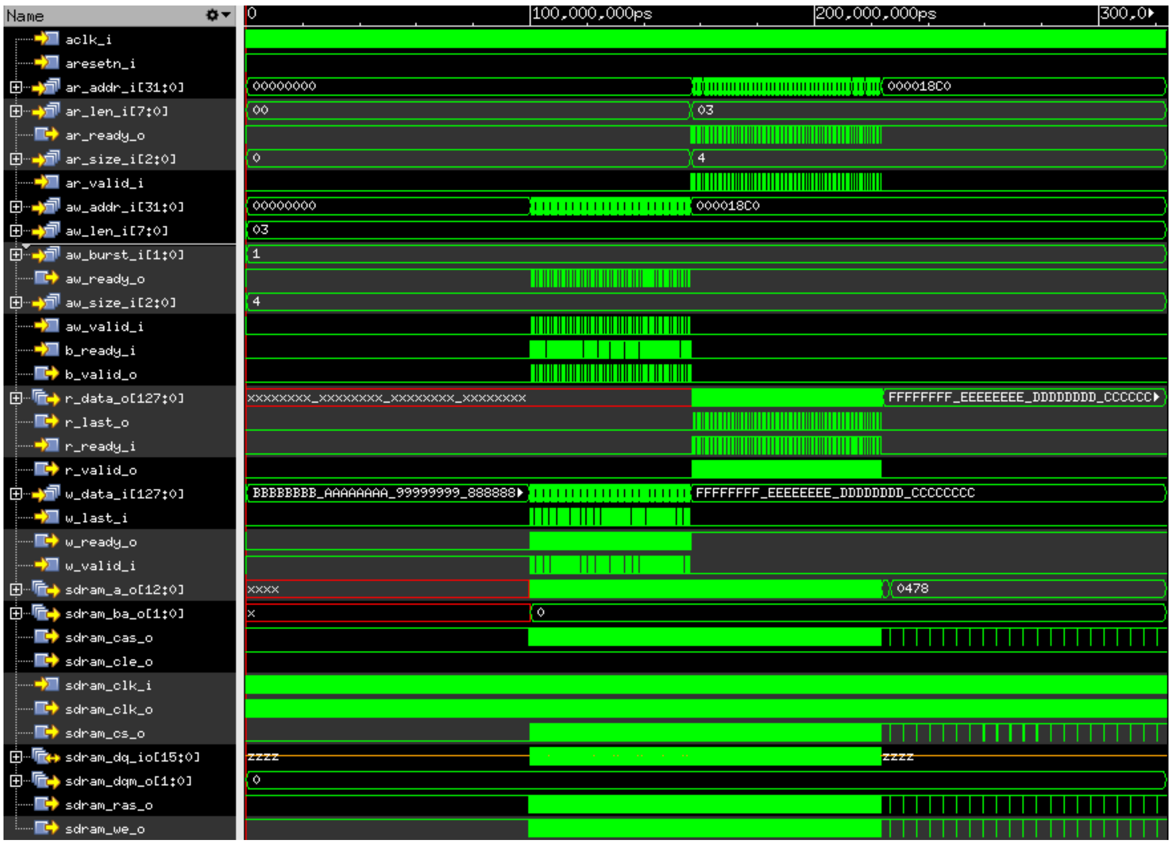


Figure 3.24: Waveforms obtained from the simulation of the AXI SDRAM controller.

annotation rate is obtained. This information is reported when stimulus is read into Joules, as shown in table 3.6. Once the activity is annotated, it must be propagated through the different nets in the design. This propagation of the activity is automatically done by Joules. After that the power of the design can be computed.

Analysis

There are different ways to report the computed power. By default the power reported by Joules is computed using the average of the switching activity. Alternatively, if the switching activity has timing information (is in SHM or VCD format) then it can also be read into multiple intervals and a graph can be obtained showing the evolution of power over time.

As a example, the simulation shown in figure 3.24 has a total duration around 324 μs. When computing the power, it has been divided into 1000 intervals (the maximum allowed) of 324 ns. With this setup the graph shown in figure 3.25 has been obtained. Additionally power reports can be generated for the different hiearchies in the design or separated by source (switching, interal or leakage).

Table 3.6: Annotation report generated by Joules.

Object Type	Asserted	Unasserted	Total	Asserted
Primary Ports				
Inputs	310	0	310	100.00 %
Outputs	195	0	195	100.00 %
I/O	32	0	32	100.00 %
Sequential Outputs				
Memory	0	0	0	N/A
Flop	1530	0	1530	100.00 %
Latch	0	0	0	N/A
Drivers				
Driver nets	2230	446	2696	82.71 %
RTL Driver nets	1837	0	1837	100.00 %

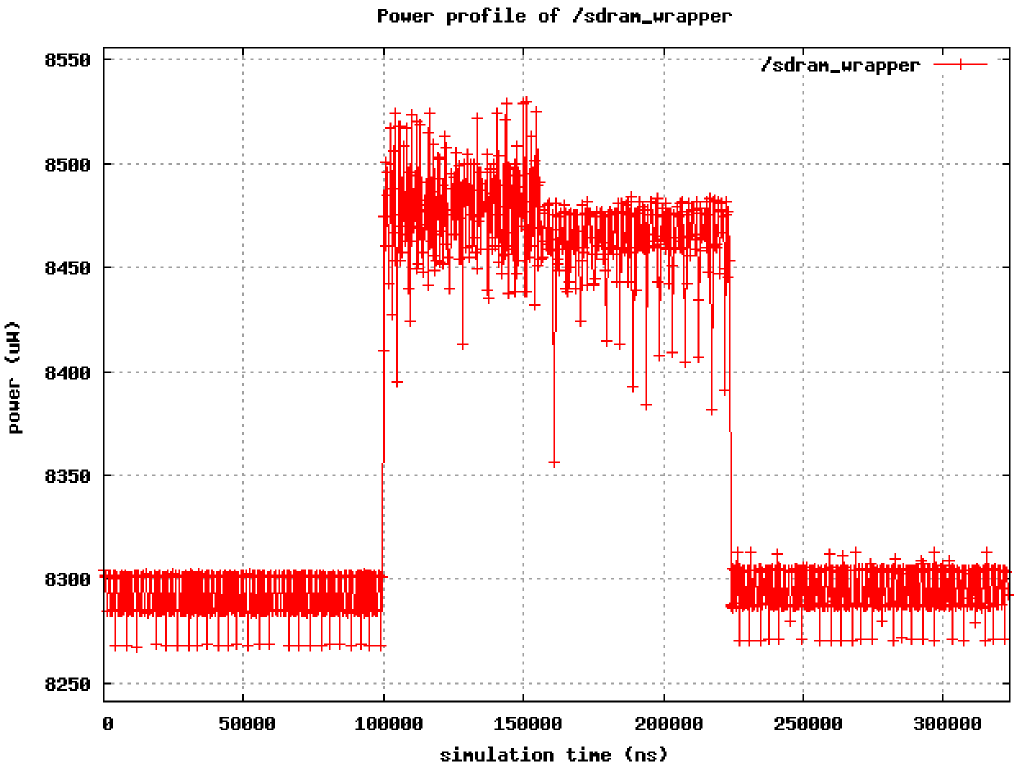


Figure 3.25: Graph of the power over time obtained with Joules.

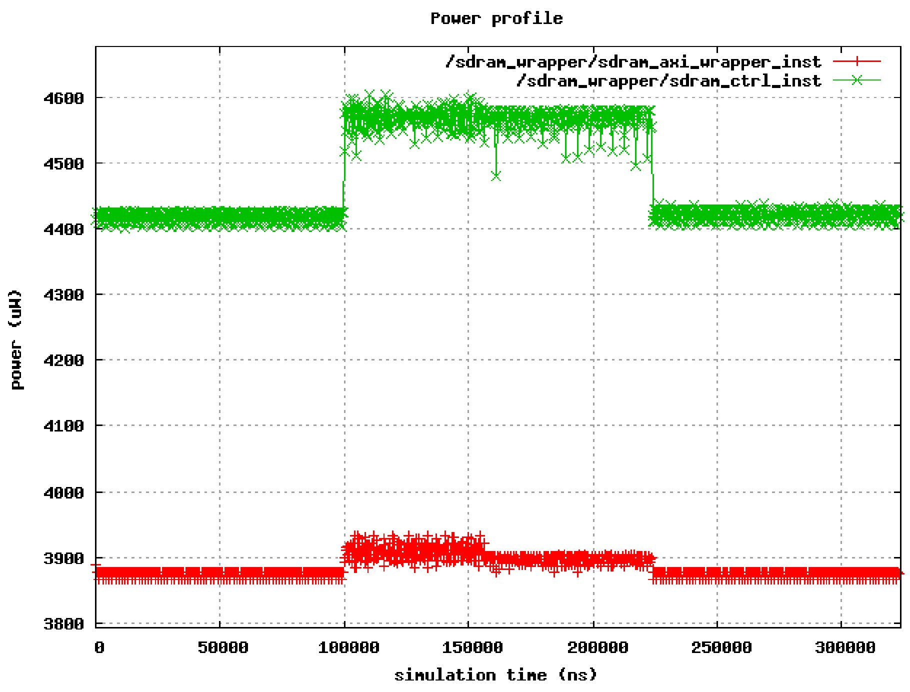


Figure 3.26: Graph of the power over time, divided by main blocks.

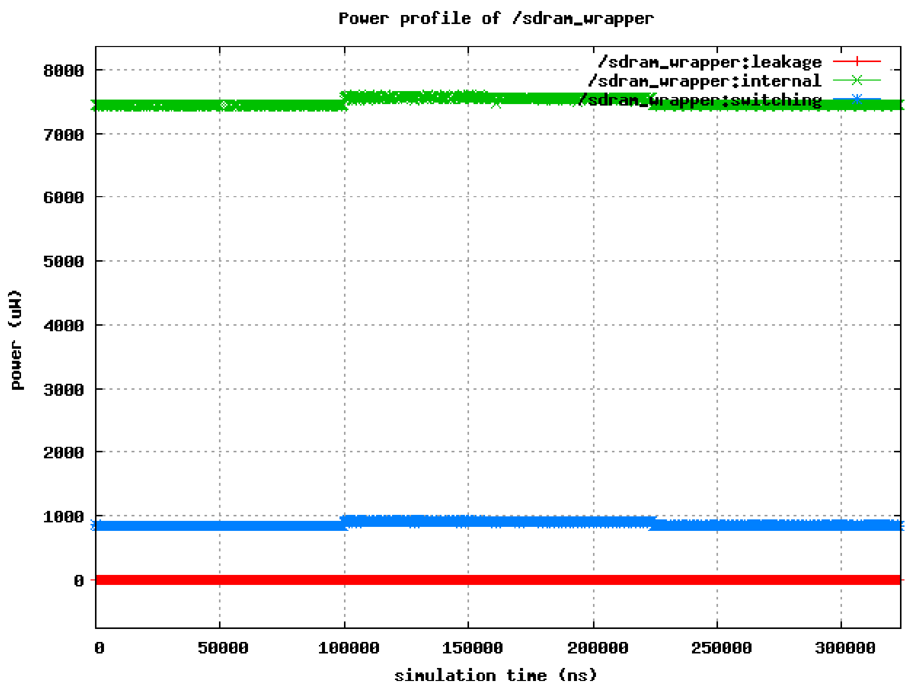


Figure 3.27: Graph of the power over time, divided by power source.

Chapter 4

Results

The current section presents and analyzes the results obtained. It is divided into three main sections. Firstly, simulation results provide some metrics that are used to evaluate the performance of the design. Secondly, synthesis results give an area estimation. Finally, power analysis results are presented.

4.1 Simulation

Table 4.1 shows the throughput measurements obtained by simulation with an SDRAM frequency of 166 MHz with different sizes for the data bus. The first column contains the maximum theoretical bandwidth, the following columns the measurements of write and read operations both for random and sequential accesses.

Table 4.1: Measured throughput of the SDRAM controller.

SDRAM data bus	Bandwidth (MB/s)	Write Random (MB/s)	Write Sequential (MB/s)	Read Random (MB/s)	Read Sequential (MB/s)
16 bit	332.00	124.09	195.98	104.29	149.79
64 bit	1328.00	172.76	359.25	136.44	232.19

When increasing the data bus by 4 the measured throughput is about 40 % and 30 % for random writes and reads respectively and 83 % and 55 % for sequential writes and reads, while the expected improvement is of 300 %. This is because in the current version the design is not optimized and each request to memory takes a fixed amount of time where the actual data transfer is only a small part of it. When increasing the data bus, the transfers of data takes less time, but it only represents a small percentage of the whole transfer. Throughput could be improved by pipelining the requests.

Table 4.2: Average simulation time of all the ISA-tests.

C++ behavioral model	preDRAC with packetizer	preDRAC with AXI-SDRAM interface
158.940 μs	344.769 μs	233.816 μs

Table 4.2 shows the average simulation time it took to complete all the different ISA-

test, it includes the design with the C++ memory model, the current design of the preDRAC with the packetizer and the the preDRAC integrated with the AXI-SDRAM interface IP. The ISA-test are meant to test the proper behavior of the system, they do not perform memory intensive operations so this results are used basically to verify that the memory controller effectively improves the performance compared with the current setup.

4.2 Synthesis

Synthesis reported a TNS (total negative slack) of 0 ps and an area of 23496 μm^2. Tables 4.3 and 4.4 contain a summary of the synthesis reports obtained with Genus. The area of the current version of the preDRAC chip is 2.3 mm^2, then current configuration of the AXI-SDRAM controller represents about 1 % of the total area of the preDRAC chip.

Table 4.3: Summary of the area report obtained by Genus.

Cell Area (μm^2)	**Net Area (μm^2)**	**Total Area (μm^2)**
19122.800	4374.103	23496.903

Table 4.4: Summary of the gates area report obtained by Genus.

Types	**Instances**	**Area** (μm^2)	**Area** (%)	**Leakage Power** (nW)	**Leakage Power** (%)
sequential	1533	16261.200	85.0	902.104	82.0
inverter	35	48.400	0.3	4.499	0.4
buffer	3	14.400	0.1	3.288	0.3
tristate	16	64.000	0.3	1.743	0.2
logic	794	2734.800	14.3	189.149	17.2
physical_cells	0	0.000	0.0	0.000	0.0
Total	2381	19122.8	100.0	1100.783	100.0

4.3 Power analysis

Table 4.5 shows a summary of the results of the power analysis. Two different experiments are reported: one with the default Joules flow and the other after doing a Clock Tree (CT) estimation. As it can be seen, due to the introduction of the routing of the clock the overall power increased about a 30 %.

For each experiment, power is reported for different activity levels. Init corresponds to the initialization sequence of the SDRAM, Write and Read to the power obtained while performing a set of writes and read operations respectively and Idle when no operations are being perfomed.

Init and Idle operations are equivalent since during the initialization of the SDRAM there is no operation. There is only 2.2 % of difference between the controller being idle and performing an actual operation. This may be due to the fact that the main contribution to the power is the clock signal, which is always present, and it suggest that the design could benefit from applying clock gating to reduce the power when no operations are being per-

Table 4.5: Power report results.

		Leakage (W)	**Internal (W)**	**Switching (W)**	**Total (W)**
Default	Init	1.04510e-6	7.44136e-3	8.49463e-4	8.29187e-3
	Write	1.05012e-6	7.56414e-3	9.13066e-4	8.47827e-3
	Read	1.05034e-6	7.55225e-3	9.09342e-4	8.46264e-3
	Idle	1.05439e-6	7.44248e-3	8.51572e-4	8.29510e-3
With CT	Init	1.14516e-6	8.64732e-3	2.02452e-3	1.06730e-2
	Write	1.15019e-6	8.76989e-3	2.08812e-3	1.08592e-2
	Read	1.15040e-6	8.75804e-3	2.08437e-3	1.08436e-2
	Idle	1.15445e-6	8.64843e-3	2.02660e-3	1.06762e-2

formed. There is a slightly difference of 0.2 % in power between write and read operations. This difference can be appreciated on figure 3.25.

Chapter 5

Conclusions and future work

The main goal of this project, the design of an AXI-SDRAM interface IP has been successully completed. A significant effort was put into the verification, since the first priority was to obtain a functional design that can actually be integrated into an ASIC as the final purpose is to use this IP in a future version of the DRAC SoC.

Once the standalone AXI-SDRAM controller has been successfully verified, it has been integrated with the preDRAC SoC to verify the whole system. After some iterations to fix design errors the design successfully passed all the tests. The designed AXI-SDRAM interface was meant to replace the current interface to external memory to improve the performance of the system. This has been achieved and a significant increase on the perforance of the access to memory has been obtained. But it has to be taken into account that the current tests performed are meant to verify the proper functionality of the design and are not meant to be used as performance metrics.

An extra step of power analysis has been done. At this point, there are no defined power requirements but this is usually one of the major concers. From the results obtained, there is little diffference between the power consumption while it is operating and when it is in idle. The main power contribution comes from the clock. As a further step, low power techniques could be applied to minimize the power, being clock gating a good possible option to reduce the power.

The current design is a first version. Future revisions will improve the performance of the controller, specially the bandwidth which in the current version is far from the maximum performance. Future work include:

- Implementation of pipeline in the SDRAM controller to increase the bandwidth.
- Making the AXI-SDRAM controller configurable. The current version the controller is parametrizable, which means the parameters can be set during instantiation. In future versions it is expected to make it configurable, so the internal timing parameters are configured with internal programmable registers.
- Revision and improvement of the verification setup. Expected additional features will require more testing. Then future work also include revising the current verification setup to test the new features.
- FPGA implementation of the whole preDRAC SoC with the AXI-SDRAM controller. This requires additional work since the design doesn't fit in the FPGA used in this

project, then another FPGA will have to be used. Higher end FPGA boards don't include SDR SDRAM, then an additional PCB with the required memory has to be previously designed.

- Analysis of low power techiques such as clock gating to reduce power consumption of the design. As it has been mentioned, power consumption could be probably reduced, but it requires further analysis and testing to ensure this is done without affecting the functionality of the design.

Bibliography

[1] BSC, "The BSC coordinates the manufacture of the first open source chip developed in Spain," December 2019. [Online]. Available: https://www.bsc.es/news/bsc-news/the-bsc-coordinates-the-manufacture-the-first-open-source-chip-developed-spain. Accessed: 2020-05-14.

[2] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanović, "The risc-v instruction set manual, volume i: User-level isa, version 2.1," Tech. Rep. UCB/EECS-2016-118, EECS Department, University of California, Berkeley, May 2016.

[3] W. Song, "Untethered lowRISC tutorial." [Online]. Available:https://www.lowrisc.org/docs/untether-v0.2/. Accessed: 2019-06-18.

[4] K. Asanović, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D. A. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo, and A. Waterman, "The rocket chip generator," Tech. Rep. UCB/EECS-2016-17, EECS Department, University of California, Berkeley, Apr 2016.

[5] J. L. Hennessy and D. A. Patterson, "Memory hierarchy design," in *Computer Architecture, Fourth Edition: A Quantitative Approach*, ch. 5, pp. 288–354, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2006.

[6] B. Jacob, S. W. Ng, and D. T. Wang, "Overview of DRAMs," in *Memory Systems* (B. Jacob, S. W. Ng, and D. T. Wang, eds.), ch. 7, pp. 315–351, San Francisco: Morgan Kaufmann, 2008.

[7] K. Itoh, M. Horiguchi, and H. Tanaka, "An Introduction to LSI Desig," in *Ultra-Low Voltage Nanoscale Memories (Series on Integrated Circuits and Systems)*, ch. 1, pp. 1–78, Berlin, Heidelberg: Springer-Verlag, 2007.

[8] Micron Technology Inc., "SDR SDRAM." MT48LC16M16A2 datasheet, 2015. Rev. W, https://www.micron.com/-/media/client/global/documents/products/data-sheet/dram/256mb_sdr.pdf, Accessed: 2020-05-04.

[9] Integrated Silicon Solution Inc., "256Mb SYNCHRONOUS DRAM." IS42S16160G datasheet, 2013. Rev. F, http://www.issi.com/WW/pdf/42-45S83200G-16160G.pdf, Accessed: 2020-05-04.

[10] "Amba overview." https://developer.arm.com/architectures/system-architectures/amba. Accessed: 2020-04-14.

[11] ARM, "What is AMBA, and why use it?," February 2020. [Online video]. Available: https://www.youtube.com/watch?v=CZlDTQzOfq4i. Accessed: 2020-05-05.

[12] ARM, “Evolution of the Arm AMBA specifications,” November 2019. [Online video]. Available: `https://www.youtube.com/watch?v=T-sNar6yBcg`. Accessed: 2020-05-05.

[13] ARM, *AMBA AXI and ACE Protocol Specification*, 2019. Rev. G.

[14] Cadence Design Systems Inc, *Genus User Guide*, 2019.

[15] P. Rashinkar, P. Paterson, and L. Singh, *System-on-a-chip Verification: Methodology and Techniques*. Norwell, MA, USA: Kluwer Academic Publishers, 2000.

[16] C. Spear, *SystemVerilog for Verification, Second Edition: A Guide to Learning the Testbench Language Features*. Marlboro, MA, USA: Springer Publishing Company, Incorporated, 2nd ed., 2008.

[17] J. Bhasker and R. Chadha, *Static timing analysis for nanometer designs: a practical approach*. Springer, 2009. OCLC: ocn297148333.

[18] N. Weste and D. Harris, “Power,” in *CMOS VLSI Design: A Circuits and Systems Perspective*, ch. 5, pp. 181–210, USA: Addison-Wesley Publishing Company, 4th ed., 2010.

[19] N. Sharif, N. Ramzan, F. K. Lodhi, O. Hasan, and S. R. Hasan, “Quantitative analysis of state-of-the-art synchronizers: Clock domain crossing perspective,” in *2011 7th International Conference on Emerging Technologies*, pp. 1–6, Sep. 2011.

[20] C. E. Cummings, “Multi-clock domain synchronizers,” in *Clock Domain Crossing (CDC) Design & Verification Techniques using SystemVerilog*, 2008.

[21] Micron Technology Inc, “64 mb - 512 mb sdram verilog model,” 2016. Revision 2.3 [Online Verilog model]. Available:`https://www.micron.com/-/media/client/global/documents/products/sim-model/dram/sdram/sdr_sdram.zip?rev=dd26a7b0db494f8ca96595bf8d0cdb6c`. Accessed: 2020-05-20.

[22] “De0-nano development and education board.” `http://de0-nano.terasic.com`. Accessed: 2020-04-17.

[23] Cadence Design Systems, Inc, *Power Analysis Flow from RTL Power through Signoff Power with Joules and Voltus*, 2019.

[24] Cadence Design Systems Inc, *Joules User Guide*, 2019.