

Politecnico di Torino

Politecnico di Torino

Master's Degree in Electronic Engineering

Academic Year 2025-2026

Graduation Session October 2026

Design and Implementation of an AXI Memory-Mapped Flash Controller for the X-ALP RISC-V SoC Platform

Relators:

Guido Masera

Maurizio Martina

Luigi Giuffrida

Candidate:

Alessandro Barocci

Acknowledgements

I would like to extend my heartfelt thanks to my thesis supervisor, Luigi Giuffrida, for his continuous support, insightful guidance, and patience throughout the development of this thesis. My deepest gratitude also goes to my brother, my parents, and my friends, whose encouragement and presence have been an invaluable source of motivation during this final academic year. Finally, I would like to thank all the members of the Electronics Department at Politecnico di Torino whom I had the pleasure of interacting with, for contributing to such a stimulating, welcoming, and inspiring environment.

Table of Contents

List of Tables VI

List of Figures VII

1 Introduction 1
- 1.1 Subject . . . 1
- 1.2 X-ALP . . . 1
- 1.3 AXI Protocol . . . 3
  - 1.3.1 Overview . . . 3
  - 1.3.2 Signals of AXI . . . 4
- 1.4 SPI protocol . . . 7
  - 1.4.1 Signals of SPI . . . 7
  - 1.4.2 Data transfers in SPI . . . 7
  - 1.4.3 Advantages . . . 8
  - 1.4.4 Limitations . . . 8
- 1.5 SPI subsystem and AXI Flash Controller introduction . . . 9

2 SPI subsystem 10
- 2.1 The SPI subsystem, its features and internal actors . . . 10
  - 2.1.1 Configurable Data Size . . . 10
  - 2.1.2 SPI Controller . . . 10
  - 2.1.3 Flash Controllers . . . 10
- 2.2 RTL Modules and Architecture . . . 11
- 2.3 Configurability of the SPI subsystem . . . 14
- 2.4 SPI Subordinate: Flash Memory . . . 17
  - 2.4.1 Opcodes . . . 17
- 2.5 HDL Source Code . . . 19

3 AXI Flash Controller 20
- 3.1 Overview . . . 20
- 3.2 Control Unit . . . 21

3.2.1 Structure of the FSM 21
3.2.2 Flash Write Complexity 23
3.3 The states of the FSM: shortly outlined 23
3.4 Structure of each sub-FSM 25
3.4.1 top-FSM state: TOP_IDLE 25
3.4.2 top-FSM state: TOP_AXIREQ 25
3.4.3 Interlude: Address and byte lanes mutual alignment 26
3.4.4 top-FSM state: TOP_AXIREQ (cont.) 31
3.4.5 top-FSM state: TOP_POWERON 31
3.4.6 top-FSM state: TOP_READ 32
3.4.7 top-FSM state: TOP_FWAIT 40
3.4.8 top-FSM state: TOP_ERASE 41
3.4.9 top-FSM state: TOP_MODIFY 44
3.4.10 top-FSM state: TOP_WRITE 47
3.4.11 top-FSM state: TOP_AXIRESP 48
3.5 HDL Source Code 49

4 Results 50
4.1 Simulation 50
4.1.1 Simulation Program 50
4.1.2 Simulation Snapshots 52
4.1.3 Effect of Performance Enhancement Measure: Sector READ-/WRITE Bypass 60
4.2 Comparison between flash controllers 62

5 Conclusion 65
5.1 Advantages 65
5.2 Future improvements 65

Bibliography 67

List of Tables

2.1 Flash command opcodes 18

4.1 Performance comparison between the two SPI subsystem flash controllers. 63

List of Figures

1.1 The channels of AXI 5
1.2 Timing diagram of a AXI read transaction 6
1.3 Timing diagram of a AXI write transaction 6
1.4 SPI schematic highlighting how the actors are interconnected by the interface 8

2.1 Block diagram of the SPI subsystem 12
2.2 Schematic of SPI subsystem, all flash controllers omitted 16
2.3 Schematic of SPI subsystem, axi_to_flash_controller omitted . . . 16
2.4 Schematic of SPI subsystem, w25q128jw_controller omitted 17
2.5 Flash SPI protocol timing diagram: WE opcode (0x06) transfer . . 19

3.1 Block diagram depicting how all interested data actors are interconnected to exchange data during the AXI transaction 21
3.2 State transition chart as representation of the final state machine's top level 22
3.3 State transition diagram of TOP_IDLE 25
3.4 State transition diagram of TOP_AXIREQ 26
3.5 Sampling of W_DATA, example 0 27
3.6 Sampling of W_DATA, example 1 28
3.7 Sampling of W_DATA, example 2 28
3.8 Sampling of W_DATA, example 3 29
3.9 State transition diagram of TOP_POWERON 31
3.10 State transition diagram of TOP_READ 33
3.11 State transition diagram of TOP_FWAIT 40
3.12 State transition diagram of TOP_ERASE 41
3.13 State transition diagram of TOP_MODIFY 44
3.14 Sector data transfer from flash to buffer, weighting on address . . . 47
3.15 State transition diagram of TOP_WRITE 47
3.16 State transition diagram of TOP_AXIRESP 49

4.1 Simulation snapshot 0 : AXI WRITE - Sampling and pushing of write data beats from AXI bus W_DATA into wr_beat_queue, during AXI request handling by the flash controller. The active byte lanes shift and wrap around every write beat from the AXI manager according to the address. Notice how the first beat 0x3210 is truncated to 0x32 due to address nonalignment as it is stored into wr_beat_queue. 54
4.2 Simulation snapshot 1 : AXI WRITE - Sector data being stored in the sector buffer. Content from flash sector 0x021 with bytes initialized to 00, 01, 02, ... , FE, FF, 00, 11, ... 0x021 is the sector of only the first beat transfer of the AXI transaction. 55
4.3 Simulation snapshot 2 : AXI WRITE - Sector data being erased in the flash. After the sector erase, the flash (not)ready bit is polled from the flash status register repeatedly until it becomes '0' (see rdata[0] from reg_resp_i), only then the FSM moves from TOP_FWAIT to TOP_WRITE. 56
4.4 Simulation snapshot 3 : AXI WRITE - Sector buffer being modified with AXI beat data. Buffered sector is 0x021, the first write beat modifies its content with the value 0x32 at the last position. 57
4.5 Simulation snapshot 4 : AXI WRITE - Sector buffer data being written in the flash. The first page of the sector 0x021 is written, its first 8 bytes are shown. 58
4.6 Simulation snapshot 5 : AXI READ - AXI beat data being read from flash and pushed into rd_data_queue. All 5 the beats are read, the first one from sector 0x021 and others from sector 0x022, then stored into rd_beat_queue. 59
4.7 Simulation snapshot 6 : Dequeuing and driving of read data beats from rd_data_queue into AXI bus R_DATA, during AXI response. The active byte lanes shift and wrap around every write beat from the AXI manager according to the address. Notice how the first beat 0x3210 is truncated to 0x32 due to address nonalignment as it is stored into wr_beat_queue. 60
4.8 Full Simulation snapshot , Notice by looking at beat_count how the write beats 2,3 are much shorter than the others to process thanks to the sector processing bypass mechanism. 61
4.9 Chart profiling the w25q128jw_controller stages during the simulation 64

Chapter 1

Introduction

1.1 Subject

This thesis illustrates the setting, behavior, innerworkings, logic structure and points of merit of a AXI driven flash controller, allocated inside the SPI subsystem of the configurable microcontroller generating platform X-ALP.

1.2 X-ALP

X-ALP (eXtendable Application Level Platform) is an open-source, configurable, and extensible RISC-V System-on-Chip generator designed to support the development of heterogeneous computing systems targeting edge computing, embedded artificial intelligence, and accelerator-based applications. The platform provides a complete hardware and software environment that allows designers to integrate custom hardware accelerators while relying on a mature processor subsystem, memory hierarchy, peripheral infrastructure, and software stack. Its modular architecture significantly reduces integration effort and enables rapid prototyping of application-specific systems. [1]

At the core of X-ALP lies the CVA6 processor, a 64-bit RISC-V application-class core that provides substantially higher computational capabilities than traditional microcontroller-oriented solutions. The processor adopts a Harvard architecture with independent instruction and data paths and communicates with the rest of the system through AXI interfaces, enabling scalable integration with high-performance memory systems and hardware accelerators. The use of AXI facilitates interoperability with a large ecosystem of IP blocks and allows the platform to support multiple concurrent transactions and increased bandwidth requirements. Adapted from the original X-HEEP architecture.

The platform is organized into several functional domains that can be independently configured according to application requirements. The processor subsystem constitutes the main computation engine and can be coupled with dedicated accelerator blocks through standard AXI interconnects. This approach enables heterogeneous execution models in which software running on the CVA6 orchestrates and supervises specialized hardware accelerators responsible for computationally intensive kernels. The architecture is particularly suited to machine learning, signal processing, cryptographic, and data analytics workloads.

The memory subsystem is composed of multiple independently accessible memory banks that can be configured according to the target application. The banking organization allows parallel memory accesses, reducing contention and improving overall throughput. Individual memory banks can be selectively clock-gated or power-managed to minimize dynamic and leakage power consumption, providing a flexible trade-off between performance and energy efficiency.

The peripheral subsystem integrates the essential components required by modern embedded systems, including communication interfaces, interrupt controllers, timers, direct memory access (DMA) engines, and general-purpose input/output peripherals. These components are connected through the AXI interconnect infrastructure, enabling efficient communication between software, accelerators, and external devices. The DMA subsystem plays a crucial role in reducing processor overhead by supporting autonomous data transfers between memories, peripherals, and accelerator modules.

A key design objective of X-ALP is extensibility. The platform exposes standardized interfaces that allow custom accelerators to be integrated without requiring modifications to the processor subsystem or the software infrastructure. Consequently, developers can focus on accelerator design while inheriting a complete execution environment that includes boot support, hardware abstraction layers, software development kits, simulation frameworks, and FPGA prototyping flows. This methodology enables rapid exploration of hardware/software co-design solutions and simplifies the transition from simulation to FPGA and ASIC implementations.

The development ecosystem surrounding X-ALP supports multiple verification and deployment flows. RTL simulation can be performed using industry-standard simulators such as Verilator, QuestaSim, VCS, and Xcelium, while software applications can be developed using GCC- or Clang-based toolchains through a CMake-based build infrastructure. The platform is suitable for FPGA prototyping as well as full ASIC implementation, making it an effective research and development vehicle for next-generation heterogeneous computing systems.

By combining a 64-bit CVA6 processor, AXI-based communication infrastructure,

configurable memory subsystems, and native support for accelerator integration, X-ALP provides a flexible foundation for the exploration and deployment of heterogeneous RISC-V systems that require both high performance and energy-efficient operation.

1.3 AXI Protocol

1.3.1 Overview

The Advanced eXtensible Interface (AXI) protocol is a communication standard suited for high-frequency, high-performance, non-serial on-chip interfaces. [2]

The protocol is designed to facilitate efficient communication between multiple master and slave components while supporting high operating frequencies, pipelined transactions, and parallel data transfers. Its scalability and flexibility make it suitable for applications ranging from embedded microcontrollers to high-performance computing systems.

Its employment in the system doesn't demand the presence of complex bridging efforts, making it well suited for simple single stage interfaces like memory controllers with high initial access latency, as the one described in this thesis.

AXI's performance and flexibility is achieved thanks to key architectural features such as:

- Separation of address-control and data phases, allowing their independent handling and improving pipeline efficiency.
- Support of address-unaligned data transfers using byte strobes, enhancing internal robustness and external flexibility.
- Provides burst based transactions, increasing system autonomy, minimizing Manager time overhead, being the initial address the only one required to be issued for consecutive transfers.
- Distinct Write and Read channels, good for efficient DMA implementation.
- Multi-Subordinate, out-of-order, multiple outstanding transactions support, granting the system performance bottleneck resilience in case one Subordinate imposes more latency than the others, maximizing throughput.

In terms of applicability rather than technicality, AXI is unbound by timing constraints as it enables the straightforward insertion of register stages, thus being

a fit candidate as protocol for high speed systems, or rather low power if such frequency increase is instead exchanged for a supply voltage reduction.

1.3.2 Signals of AXI

The signal wires of the protocol are grouped in a total of 5 channels, 3 dedicated to WRITE transactions and 2 dedicated to READ transactions. Next, each channel is described with its own signals, but only the ones relevant to the project will be mentioned, as most of them are optional.

WRITE Channels

- Address Write channel **(AW)** : Manager driven, transmits the AXI write transaction's address (AW_ADDR) and control attributes to the Subordinate, such as burst length (AW_LEN), data size (AW_SIZE) and "valid" signal (AW_VALID). There is also a Subordinate driven "ready" signal used to complete the handshake exchange with the Manager (AW_READY).

- Write channel **(W)** : Manager driven, transmits the AXI write transaction's data to the Subordinate (W_DATA) complemented with byte strobe signals (W_STRB), "last data item" signal (W_LAST) and "valid" signal (W_VALID). There is also a Subordinate driven "ready" signal used to complete the handshake exchange with the Manager (W_READY).

- Write Response channel **(B)** : Subordinate driven, signals to the Manager that the write AXI transaction is completed with a "valid" signal (B_VALID). There is also a Master driven "ready" signal used to complete the handshake exchange with the Manager (B_READY).

READ Channels

- Address Read channel **(AR)** : Manager driven, transmits the AXI read transaction's address (AR_ADDR) and control attributes to the Subordinate, such as burst length (AR_LEN), data size (AR_SIZE) and "valid" signal (AR_VALID). There is also a Subordinate driven "ready" signal used to complete the handshake exchange with the Manager (AR_READY).

- Read Response channel **(R)** : Subordinate driven, transmits the AXI read transaction's data to the Manager (R_DATA) complemented with "last data item" signal (R_LAST) and "valid" signal (R_VALID). There is also a Manager

driven "ready" signal used to complete the handshake exchange with the Subordinate (R_READY).

There is no third read channel as this one handles both data and response.

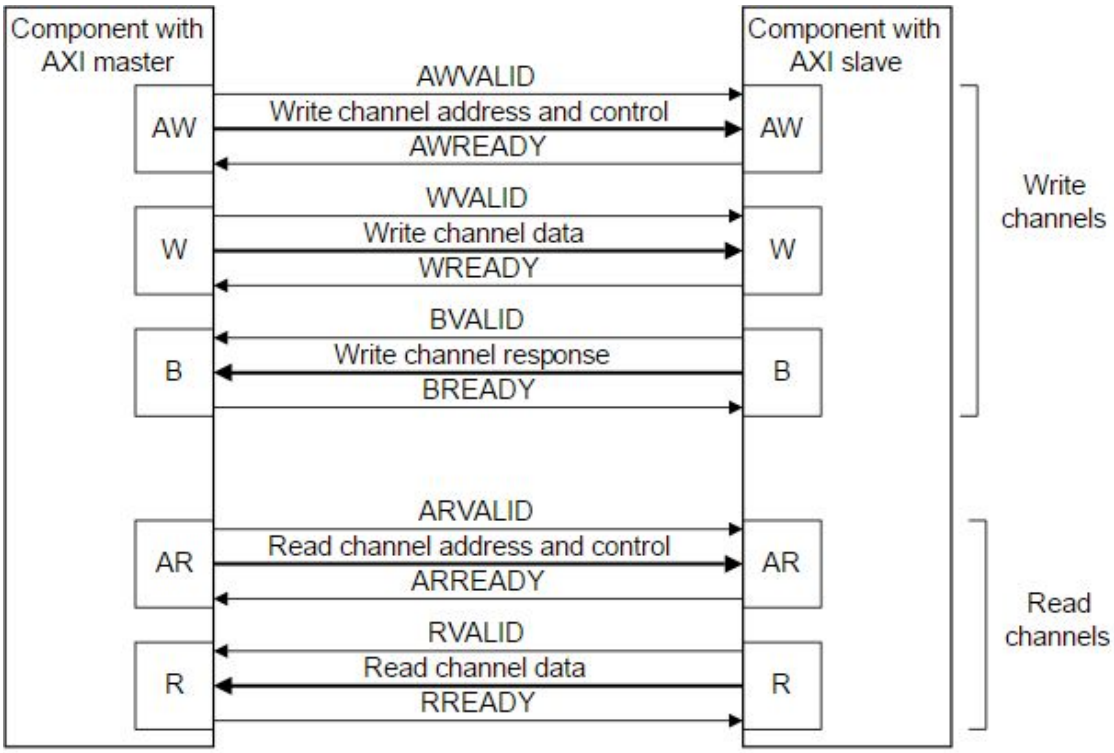


Figure 1.1: The channels of AXI

The AXI controller that is subject of this paper operates in a subset of AXI's possible operative conditions, that is:

- Single Subordinate
- Single Outstanding Transaction
- Always incremental burst type (AxBURST = INC)

Two examples of AXI transactions, respectively read and write, are displayed in form of timing diagrams in the figures 1.2 and 1.3.

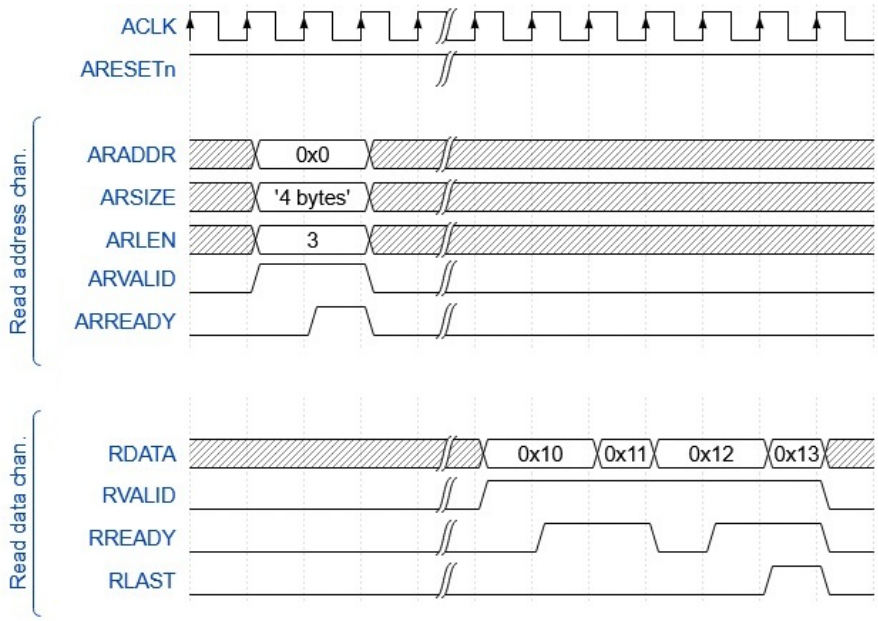


Figure 1.2: Timing diagram of a AXI read transaction

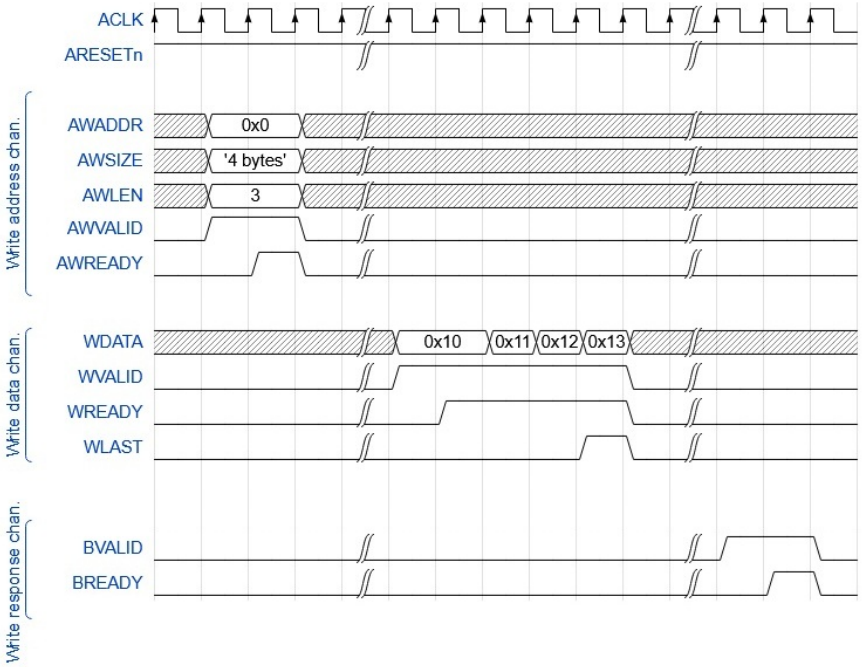


Figure 1.3: Timing diagram of a AXI write transaction

1.4 SPI protocol

While the AXI communication standard and interconnections are ideal and used for high performance on-chip interfaces, the SPI protocol is instead used for low power off-chip interfaces; the Serial Peripheral Interface (SPI) is a synchronous, full-duplex communication protocol common in embedded systems for short distance communication between a single manager and one or multiple subordinates thanks to its simplicity and hardware efficiency. [3]

It is based on a "one master to multiple peripherals" architecture, where the former controls the communications by generating the clock signal and the chip select signal for each subordinate peripheral, Unlike I2C where each peripheral is memory mapped and addressed through the data bus SDA.

1.4.1 Signals of SPI

The SPI protocol is equipped by (at least) these wires:

- SCK (Serial Clock): Generated by the master and used to synchronize data transmission.
- MOSI (Master Out Slave In): Carries data from the master to the slave.
- MISO (Master In Slave Out): Carries data from the slave to the master.
- CS / SS (Chip Select / Slave Select): Active-low signal used to enable a specific slave device.

Figure 1.4 shows how manager and subordinates are interconnected.

1.4.2 Data transfers in SPI

All SPI devices, at the boundary with the interface, require a shift register to turn word bundled output data into a serial stream or viceversa for input bits. The transfer of a data stream starts with the assertion of the CS line of target subordinate from the manager (typically driving it low), then, always the manager, will begin sending periodic pulses on SCK. MISO or MOSI are changed by the driver every SCK edge of one type, while the other edge is used for sampling by the other party; the sampling edge is defined by the parameter CPHA (Clock Phase), datasheet parameter of the subordinate peripheral. The transfer ends with the de-assertion of SCK by the manager, and then SCK will return to its idle value, which is defined by the parameter CPOL (Clock Polarity), also a datasheet parameter of the subordinate peripheral. A data transfer can be composed by a stream of 8, 16, 32 bits.

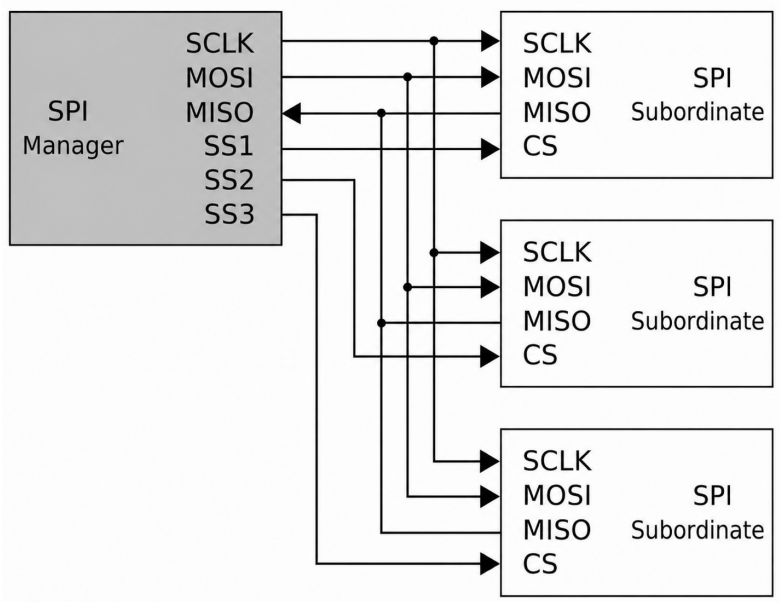


Figure 1.4: SPI schematic highlighting how the actors are interconnected by the interface

1.4.3 Advantages

The advantages of SPI over other serial protocols are:

- High data throughput: No arbitration overhead or acknowledgment phases
- Full-duplex communication: Simultaneous transmission and reception.
- Simple hardware implementation: Requires minimal logic compared to packet-based protocols.
- Flexible data size: stream length is configurable depending on implementation.

These characteristics make SPI particularly suitable for high-speed peripherals such as flash memories, ADCs/DACs, sensors, and display controllers.

1.4.4 Limitations

Despite its advantages, SPI is encumbered by these limitations:

- Low scalability: adding one subordinate means adding one physical terminal, rather than just reserving one additional address like in I2C.

- Short distance only: susceptible to crosstalk causing signal integrity errors; but mostly because data (MISO) doesn't always travel the same direction of the clock, being only manager driven. This means that skew due to wire extension can add artificial delay and time-shift the data out of its expected sampling window.
- No formal standardization of higher-level protocol: Device-specific command sets are required.
- No built-in addressing or acknowledgment: Reliability mechanisms must be implemented at higher layers.

A timing example of the SPI protocol in action is present in Chapter 2, in figure 2.5, showing the transfer of the "Writing Enable" opcode (0x06) from the the SPI Master to a SPI flash Slave.

1.5 SPI subsystem and AXI Flash Controller

X-ALP contains a SPI subsystem that allows the microprocessor to communicate with external SPI subordinate peripherals, as manager. Among these, a flash is present.

To speak with the flash, the subsystem is equipped with a AXI subordinate flash controller. Any AXI manager external to the SPI subsystem can use its AXI ports from the manager side to access the flash in a simple manner, also taking advantage of a SPI controller which directly drives the SPI terminals.

See Chapter 2 for a detailed description of the SPI subsystem and Chapter 3 regarding the AXI flash controller.

Chapter 2

SPI subsystem

2.1 The SPI subsystem, its features and internal actors

The subsystem handles the SPI data transactions between the microcontroller and external subordinates, among which a flash.

2.1.1 Configurable Data Size

Because X-ALP is based on the 64 bit architecture of the cva6 microprocessor and the intention to keep the SPI subsystem flexible, it is designed to have a configurable data size of 32 or 64 bits; as for the AXI flash controller. The SPI controller however is stuck to a 32 bits data size, regardless of the systems'.

2.1.2 SPI Controller

To trigger and define the SPI serial transfers (drive data to MOSI, acquire data from MISO, switch CS, pulse SCK) via software register interface, a SPI controller in employed in the subsystem, acting as SPI manager in the interface, being the module in direct contact with it. The next section holds a more in detail description of the SPI controller hosted by the SPI subsystem.

2.1.3 Flash Controllers

To streamline the usage of the flash by the microprocessor, two SPI controllers with different properties are leveraged, multiplexed into the SPI controller according to the necessities of the user.

To execute a simple SPI flash read or write, a complex sequence of many SPI command transfers are required to happen; to define and trigger each one of these, the SPI controller needs to be instructed via register interface each time.

The flash controller softens the workload of the microprocessor by allowing this complex program of SPI transfers to be triggered by a single flash access request, only specifying mode, address, data size and burst length.

Once the request is made, a FSM inside the flash controller will trigger and work autonomously to send all the required instructions to the SPI controller to execute the flash access according to the attributes specified by the microprocessor's request.

Basically, the flash controller serves the purpose to treat the flash like a RAM by the microprocessor. Therefore, it means that the software doesn't need to be conscious of these complexities:

- SPI complexity : data serialization, Chip Select and Serial Clock generation and protocol transfer policies.
- SPI power-on : After reset, the flash needs to be powered on by issuing the correct command before any data transaction.
- Flash latency : A Correct approach to a flash needs to be conscious of the long operational times that come with erase and write operations, hence the manager needs to verify the status of the flash before triggering a new operation.
- Flash model specific complexity : knowledge of the opcodes and how to issue them correctly with the correct timing in order to execute internal operations correctly.
- Flash byte write operation complexity : flash content cannot be written with byte precision, hence sector buffering followed by page programming is required; this concept is explained more in detail in chapter 3.

The microprocessor is released from the responsibility to handle these issues as they are passed upon the flash controller to be dealt with.

2.2 RTL Modules and Architecture

The block diagram of X-ALP's SPI subsystem is pictured in the following figure (2.1), also highlighting the different types of interfaces for inter-block communication.

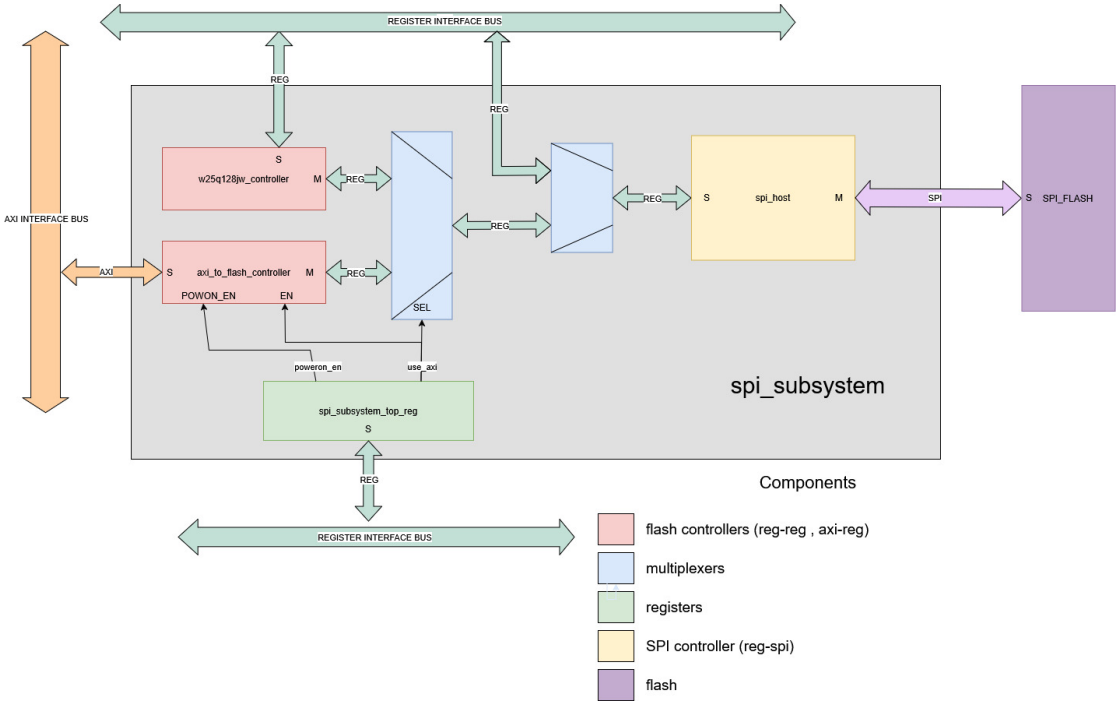


Figure 2.1: Block diagram of the SPI subsystem

The subsystem is made out of the following components:

- *spi_host* : SPI controller by OpenTitan, provided with:
  - A first FIFO to store SPI data to transmit (tx_fifo)
  - A second FIFO to store SPI received data (rx_fifo)
  - A queue to store commands that define and trigger the SPI data transfers
  - Status and control registers, descriptive of the environment.

  These actors are memory mapped and accessible through the software register interface of spi_host acting as subordinates.

  Other than these actors, within the core of spi_host resides its FSM, which handles the SPI transfers according to the command and the control register, and a shift register that turns data words from tx_fifo into a serial bit stream (MOSI) or turns the serial stream from flash (MISO) into a data word into rx_fifo.

  While the SPI subsystem can be generated with a variable data size of 32 or 64 bits, the paralleism of spi_host is fixed to 32 bits, such like its components.

- *Multiplexers* : one multiplexer selects spi_host's manager : system bus or any flash controller; the other selects the active flash controller, between the two that follow.

- *w25q128jw_controller* : flash controller with external register interface bus as manager and spi_host as subordinate. The microprocessor can issue flash accesses by changing the values of the control register inside this module.

  Write data, read data and flash sector content for write operations are buffered in the RAM, for this purpose the DMA is exploited and connected to this module (though these connections are neglected in the previous figure).

  This flash controller can work with only 32 bits data size.

- *axi_to_flash_controller* : flash controller with external AXI bus as manager and spi_host as subordinate. The microprocessor can issue commands by AXI interface and by changing the values of the SPI sybsystem control register spi_subsystem_reg_top.

  AXI data beats to be written in the flash are buffered first inside a queue (wr_beat_queue) in the module as a valid request is detected in the AW and W Channels, so that each beat can individually be written in the flash, in order of arrival.

  AXI data beats read from the flash also are buffered first in a different queue (rd_beat_queue) as they are each individually read from the flash, before being all together sent to the manager as response in the R channel, in order of arrival.

  Regarding write operations, the sector is buffered in a storage that is also locally placed within the module, a small SRAM memory: sector_buffer. The necessity of a sector buffer will be explained in Chapter 3.

  Since the buffer elements are all locally situated in the module, RAM and DMA are not involved, this detail will be important for later in Chapter 4, where both flash controllers are compared in clock cycles of latency, this feature being the difference maker.

  This flash controller can work with both 32 and 64 bits data sizes.

- *spi_subsystem_reg_top* : control register, three bits set by external register interface bus. These bits are:

- *use_axi* : selects which one of the two flash controllers is currently active: if set, the AXI driven controller is, otherwise the other; also, if de-asserted, completely freezes the FSM inside axi_to_flash_controller in IDLE state to avoid spurious transitions.
- *a2f_ctr_poweron_en* : enables the AXI flash controller to issue a preliminary power-on command to the flash.
- *a2f_ctr_quadspi_en* : enables the AXI flash controller to work in quad spi speed, rather than standard speed.

2.3 Configurability of the SPI subsystem

The SPI subsystem flexibility doesn't end with its configurable data size, but it extends further: it is possible to include or exclude hardware components given the specific parameters that are passed to the terminal as command line arguments at build time, when calling the Makefile target reserved for hardware generation.

Generating hardware means to generate the HDL source files in Systemverilog, this is done by calling a python script and feeding it a HDL template file: this file holds the HDL source code of all the possible HW blocks that can be included, these blocks each put inside a python interpretable if conditional scope that enables or not that HW piece to be finally present in the SPI subsystem; the conditions of these if scopes are function of the TCL paramenters passed to the Makefile at build time, as mentioned before.

So, in X-ALP the SPI subsystem can be configured to include or not any of the flash controllers; if none is included inside the final architecture, the SPI controller will receive requests only via software through the external register interface bus, so directly from the microprocessor.

The code relevant to the generation of the spi_subsystem hardware as HDL source code is reported in the following snippet:

```
RTL_DIR ?= rtl

#[...]

SPI_SUBSYS_PERIPH_GEN ?= axi

#[...]

# output directory
OUT_DIR ?= $(RTL_DIR)

#[...]

## Generate spi_subsystem.sv
.PHONY: periph-gen
periph-gen:
    python3 $(RTL_DIR)/spi_subsystem_gen.py
    ↪ $(RTL_DIR)/spi_subsystem.sv.tpl $(OUT_DIR)/spi_subsystem.sv
    ↪ $(SPI_SUBSYS_PERIPH_GEN)
```

What follows are four figures depicting the possible versions that can be acquired by spi_subsystem.sv inside X-ALP, according to the configuration parameters that are passed to the python script and so the Makefile target when building the hardware structure:

- Figure 2.1 - SPI subsystem with both flash controllers
  - SPI_SUBSYS_PERIPH_GEN="axi w25_ctr"
- Figure 2.2 - SPI subsystem without flash controllers
  - SPI_SUBSYS_PERIPH_GEN=""
- Figure 2.3 - SPI subsystem without axi_to_flash_controller
  - SPI_SUBSYS_PERIPH_GEN="w25_ctr"

- Figure 2.4 - SPI subsystem without w25q128jw_controller
  - SPI_SUBSYS_PERIPH_GEN="axi" (default)

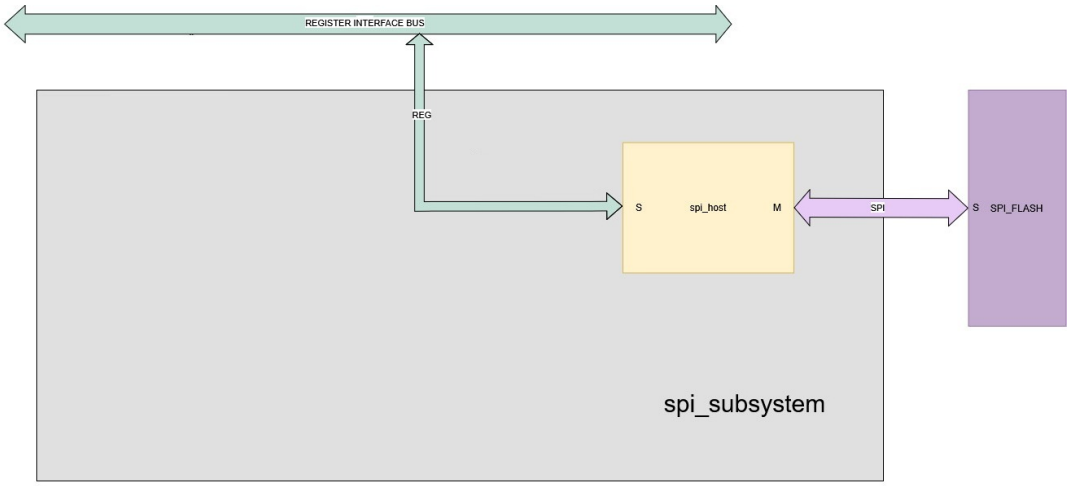


Figure 2.2: Schematic of SPI subsystem, all flash controllers omitted

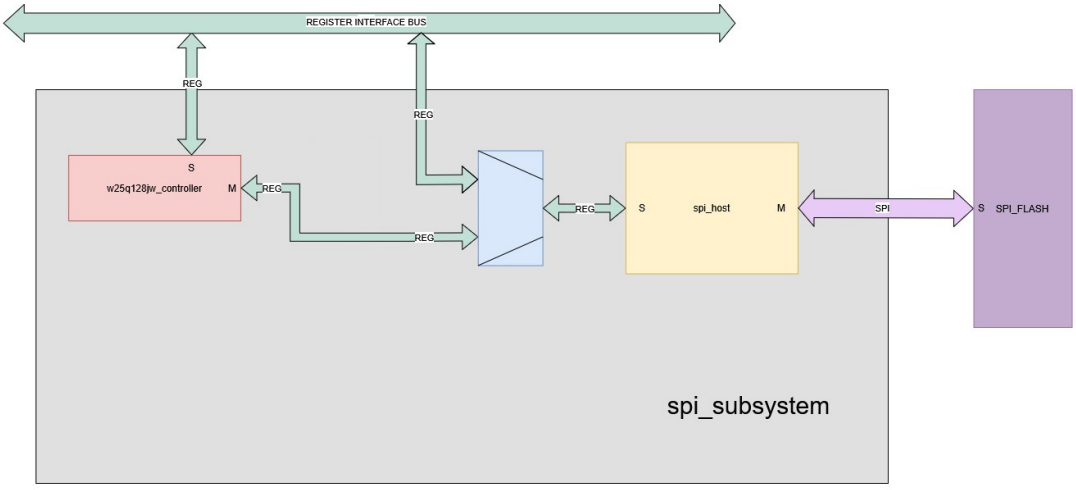


Figure 2.3: Schematic of SPI subsystem, axi_to_flash_controller omitted

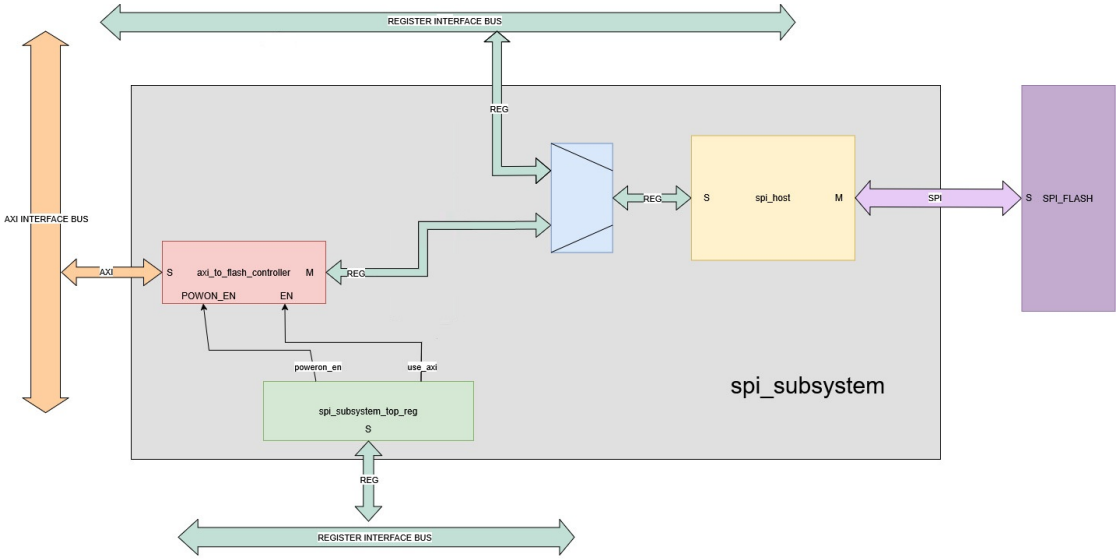


Figure 2.4: Schematic of SPI subsystem, w25q128jw_controller omitted

2.4 SPI Subordinate: Flash Memory

As previously stated in this chapter, the external SPI subordinate attached to the managing SPI subsystem is a flash holding the firmware for the microcontroller and weights for CNN edge AI applications.

The currently employed flash model is Windbond's w25q128jw [4] , a 16 MB SPI flash supporting standard and quad SPI speeds, SCK up to 133 MHz, so an equivalent throughput of up to 532 MHz.

The flash can be erased with a resolution of 4 KB (sector) and programmed with a resolution of 256 B (page, hence 16 pages in a sector). Since the size is 16 MB and being each byte addressable individually, the flash address is 24 bits long, with the least significant half being the offset of the sector, and the remaining being the byte address relative to the sector.

2.4.1 Opcodes

To issue instructions to the flash, commands must be sent in SPI in form of 1 Byte opcodes. To send the opcodes.

The SPI commands recognized by the flash to trigger its operations that are relevant to its application by the AXI flash controller are listed in table 2.1:

Action	**Identifier**	**Opcode Byte**
Power On	PO	0xAB
Read data (standard speed)	RD	0x03
Read data (quad speed)	RDQIO	0xEB
Read Status Register 1 (flash ready/busy)	RSR1	0x05
Write Enable	WE	0x06
Sector Erase 4KB	SE	0x20
Page Program (standard speed)	PP	0x02
Page Program (quad speed)	PPQ	0x32

Table 2.1: Flash command opcodes

To issue commands via opcodes, the correct timing must be respected between MOSI, CS and SCK as illustrated in the datasheet, also according to the flash model specific SPI environment parameters: csnidle, csntrail, csnlead, fullcyc, cpha, cpol.

A timing example of the SPI protocol in action inside the interface between the w25q128jw flash and its manager is shown in figure 2.5; in this context the flash works with SPI parameters cpha = 0 and cpol = 0, although these are not the only values supported by this model. What is being transferred is the "Write Enable" opcode (0x06), from user to flash, as required to do before every page program and sector erase (see Chapter 3).

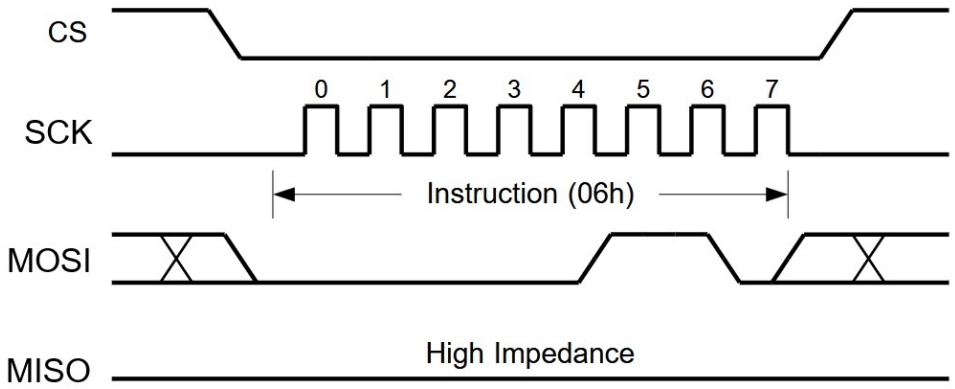


Figure 2.5: Flash SPI protocol timing diagram: WE opcode (0x06) transfer

2.5 HDL Source Code

The GitHub repository holding the source files of the SPI subsystem and its internal modules can be found online at this link: `https://github.com/alebaroc/xhee p_spi/tree/first_impl/rtl`.

Chapter 3

AXI Flash Controller

3.1 Overview

The module converts the AXI flash access request into a sequence of software instruction towoards the SPI controller that, accordingly to the AXI attributes, will act as SPI manager to access the flash, SPI subordinate.

The flash controller is essentially a Finite State Machine, whose program begins with the AXI request polling and ends with the AXI response.

For future reference, a block diagram of the SPI subsystem and flash is drawn in figure 3.1; it highlights the path traveled by data as it is directed by the AXI flash controller.

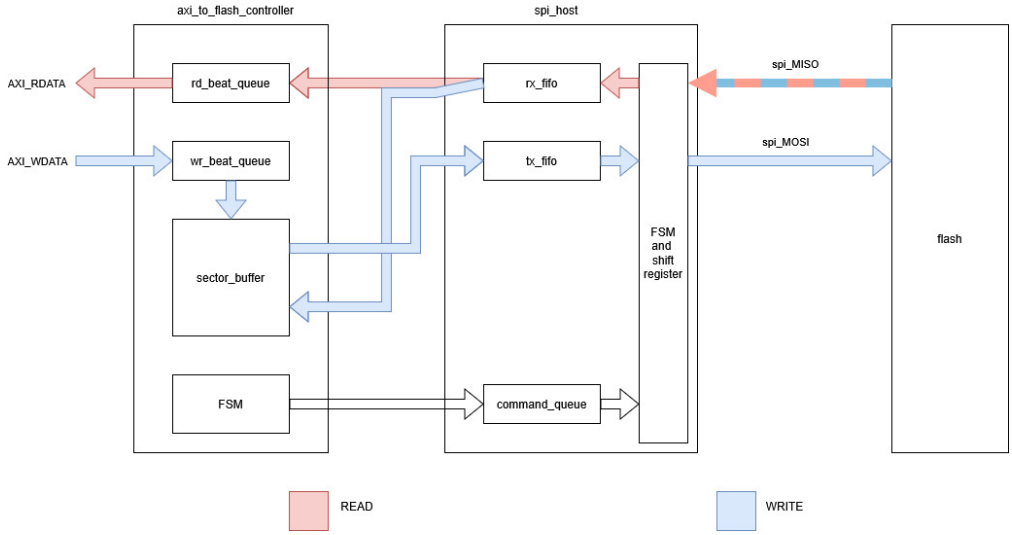


Figure 3.1: Block diagram depicting how all interested data actors are interconnected to exchange data during the AXI transaction

3.2 Control Unit

3.2.1 structure of the FSM

Each state of the FSM is associated to a sub-FSM. When a top-state is true, its relative sub-FSM will execute its sub-states from first to last. The top-fsm state diagram is displayed in figure 3.2.

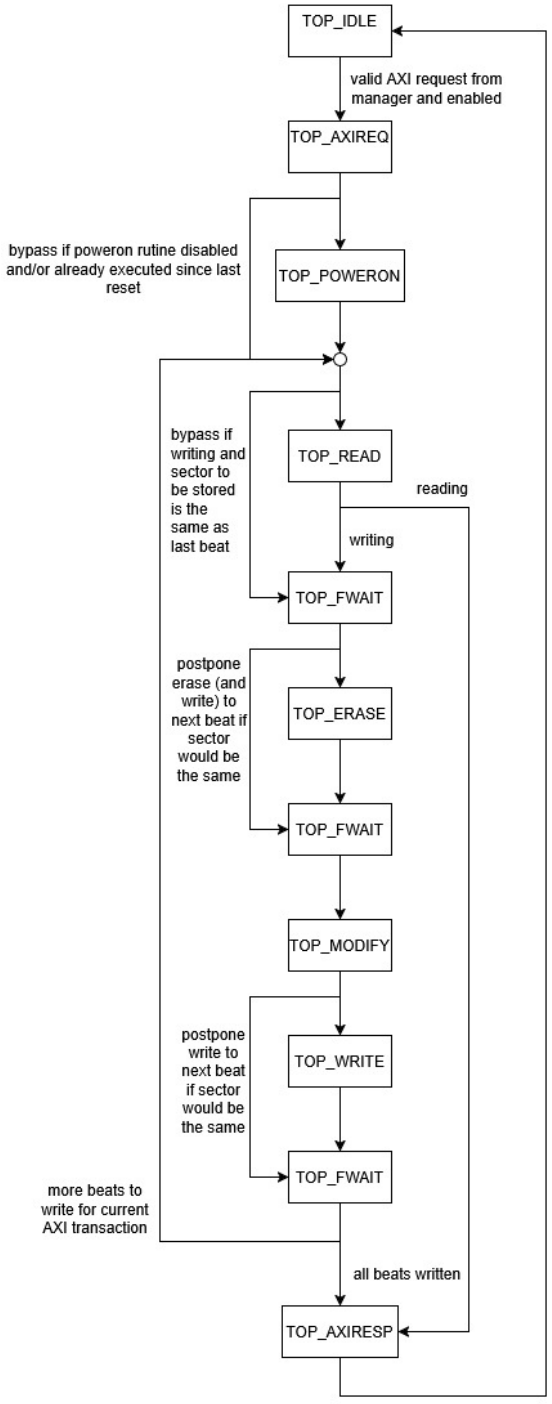


Figure 3.2: State transition chart as representation of the final state machine's top level

Two different programs are executed as sequence of top-states according to the AXI transaction type: write or read.

(TOP_*)

- **READ** : IDLE , AXIREQ , POWERON , READ , AXIRESP .
  - all AXI read data beats are read from the flash in a single READ state iteration.

- **WRITE** : IDLE , AXIREQ , POWERON , READ , FWAIT , ERASE , FWAIT , MODIFY , WRITE , FWAIT , AXIRESP .
  - MODIFY brings only the current AXI write data beat into the sector buffer, so the whole FSM except for AXIREQ and AXIRESP needs to be crossed for each beat. Once all beats are written AXIRESP completes the program.

3.2.2 Flash Write Complexity

The additional complexity of the WRITE operation is due to the fact that, while the flash is capable to be read with byte precision, when writing, the bits of the flash can be affected with a much lower precision: '1' can be written only with sector resolution.

So it is required first to buffer the sector data into a second memory to not lose its content after erasing it, the sector being the one that contains the location of the data to be written, then erase said sector in the flash (turn it into '1's), modify the sector buffer with the write data and only then program the flash sector with the updated information, page by page.

As stated in Chapter 2, the sector is the erase resolution (4 KB) and a page is the program resolution (256 B, a 16th of the sector).

To program a flash page means to turn some of its bits into '0's, these bits defined by the write page data that is fed.

Also, intuitively, the sector buffer must be a RAM-like memory item that allows bit precise value modification; in this case it is a local SRAM memory.

3.3 The states of the FSM: shortly outlined

The states of the top-FSM are:

- **TOP_IDLE** : IDLE state of the top-FSM. This is the only top-state that doesn't yeld a sub-FSM. The FSM in this state waits for a AXI request from the manager.

- **TOP_AXIREQ** : state dedicated to react to the AXI request from manager. The transaction attributes and possible write data are stored respectively in registers and the write beat queue (wr_beat_queue).

- **TOP_POWERON** : state dedicated to the SPI transmission of the power-on command to the flash and wait for its reset preamble.

- **TOP_READ** : state used to read the data from the flash. Essential for both read but also write operations given the necessity to buffer the sector before erasing it.

  Each beat is written independently in the flash, each requiring its own full top-FSM cycle, but with some measures to avoid read, erase and write operation redundancy; in fact, the sector read is avoided if the previous write beat of that same AXI transaction burst shared the same sector (because it would have been the same sector).

- **TOP_FWAIT** : state dedicated to wait for flash's availability to receive further directions. In this state the flash status register is read continuously polling the "ready" bit.

- **TOP_ERASE** : state dedicated to the erasure of the write beat's sector. All the bits of the sector are turned into '1's. To avoid non-consequential erasures, if the current write beat would share the same sector with the next one in the AXI transaction burst, then the erasure is postponed to that following write transfer.

- **TOP_MODIFY** : state dedicated to the modification of the sector loaded in the buffer with current write beat.

- **TOP_WRITE** : state used to page program the modified sector buffer data into the now erased sector in the flash, turning some bits into '0's. To avoid non-consequential programs, if the current write beat would share the same sector with the next one in the AXI transaction burst, then the write is postponed to that following write transfer (so, every time the erasure is skipped, so the write).

- **TOP_AXIRESP** : state dedicated to the final response to the manager, completing the AXI transaction. The "VALID" signal is set and potential read data is transferred.

3.4 Structure of each sub-FSM

3.4.1 top-FSM state: TOP_IDLE

The TOP_IDLE state is not really a sub-FSM as it is not composed by multiple sub-states, but just one, as it is displayed in figure 3.3.

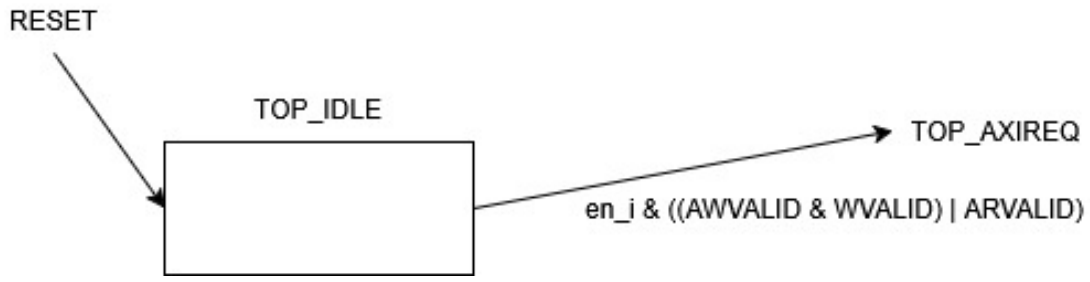


Figure 3.3: State transition diagram of TOP_IDLE

This state is reached after reset or after the completion of a AXI transaction. While in this state, all the sub-FSM also are in their respective X_IDLE state.

To allow the transition from this state, the module also needs to be externally enabled by the *use_axi* bit inside spi_subsystem_reg_top; this bit, other than just selecting the right flash controller, also freezes this module avoiding spurious transactions if not set.

The FSM transits from this state when a valid AXI request is detected: either ARVALID or both AWVALID and WVALID need to be asserted for this condition to be detected.

3.4.2 top-FSM state: TOP_AXIREQ

The state diagram of the AXIREQ sub-FSM is displayed in figure 3.4.

In this state, the control attributes defining the AXI transaction are extracted from the AR or AW channel and stored inside registers to be used later; these are: mode, address, burst length, and data size.

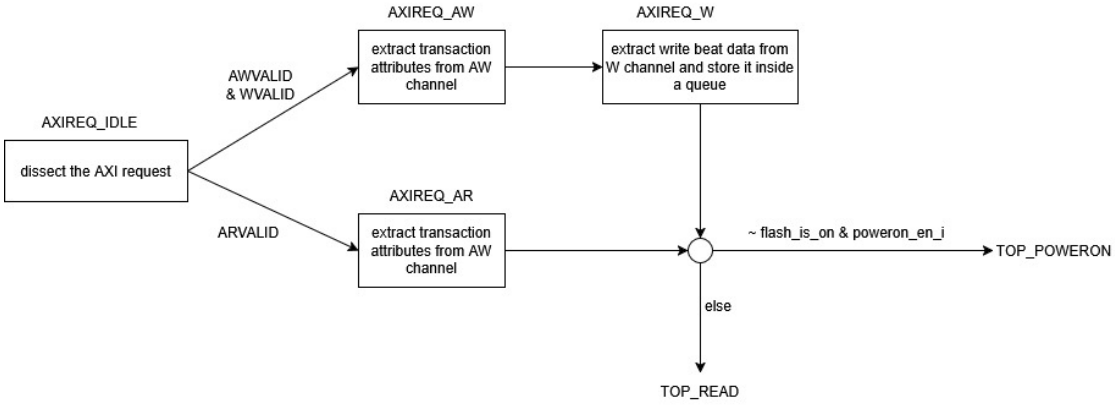


Figure 3.4: State transition diagram of TOP_AXIREQ

If the manager is writing, then there is an additional state after AXIREQ_AW to receive the write data beats from the W channel, the data is then pushed into the write beat queue (referred as wr_beat_queue in the HDL code).

3.4.3 Interlude: Address and byte lanes mutual alignment

When sampling data from the AXI bus W_DATA, the subordinate needs to be conscious of what precise byte lanes to sample for each write beat, as they follow a set of rules within the AXI protocol.

Each AXI address, so flash address, is associated to a byte lane of W_DATA.

Being lane 0 the least significant byte of W_DATA and lane 7 the most significant one (assuming 64 bits data width), the association works as following :

$$byte\ lane\,(byte\ address)\ =\ byte\ address\ \%\ 8; \tag{3.1}$$

with % being the remainder of the integer division.

So, when sampling the bytes of the AXI write beat W_DATA, the AXI subordinate needs to be conscious of which byte lanes within that bus are to be sampled.

According to data size, the byte lanes of W_DATA are grouped in subsets of N, where N is that data size in bytes, the AXI subordinate samples the write beat from one of these subsets. The address space as well, starting from 0x0, in divided into subsets of N, each being potentially assigned to the write beat of N bytes.

Since AXI supports unaligned addresses, the first write beat might be truncated if its address (AW_ADDR) is not the start of a address subset.

For instance, if the beat size is 4 bytes, any address equal to $0 + 4 \cdot N$ would be aligned, but on the contrary, if it is something in the middle of a subset like $AW_ADDR = 1$, that means the first beat would be truncated to 3 bytes of addresses 1, 2, 3, bypassing the first byte of address 0.

This truncation due to misalignment thankfully happens only in the first beat, as the address becomes aligned regardless from the second beat and on; in fact, in the previous example the second beat would be 4 bytes of addresses 4, 5, 6, 7.

The following figures visually explain the previous concept of alignment between byte lanes and addresses. The active byte lanes of W_DATA to be sampled are highlighted for each beat according to AW_ADDR and AW_SIZE:

- Figure 3.5 - aligned address
- Figure 3.6 - aligned address , sampling starts from a byte lane in the middle of W_DATA
- Figure 3.7 - unaligned address
- Figure 3.8 - unaligned address, already wrapped

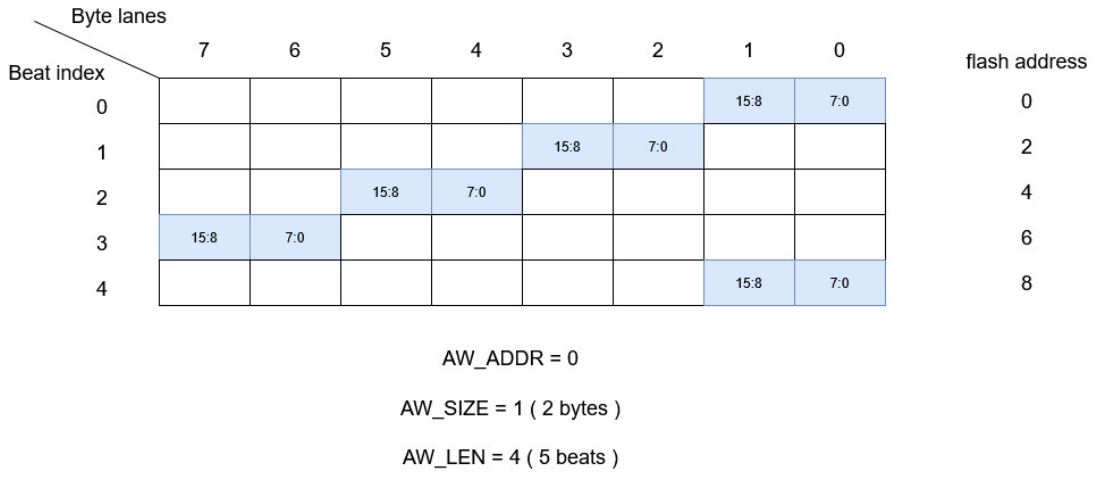


Figure 3.5: Sampling of W_DATA, example 0

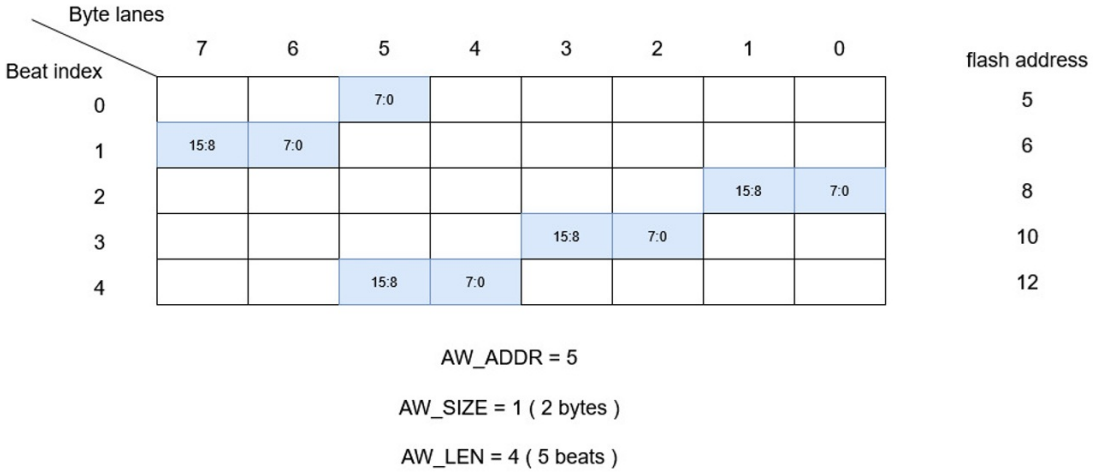


Figure 3.6: Sampling of W_DATA, example 1

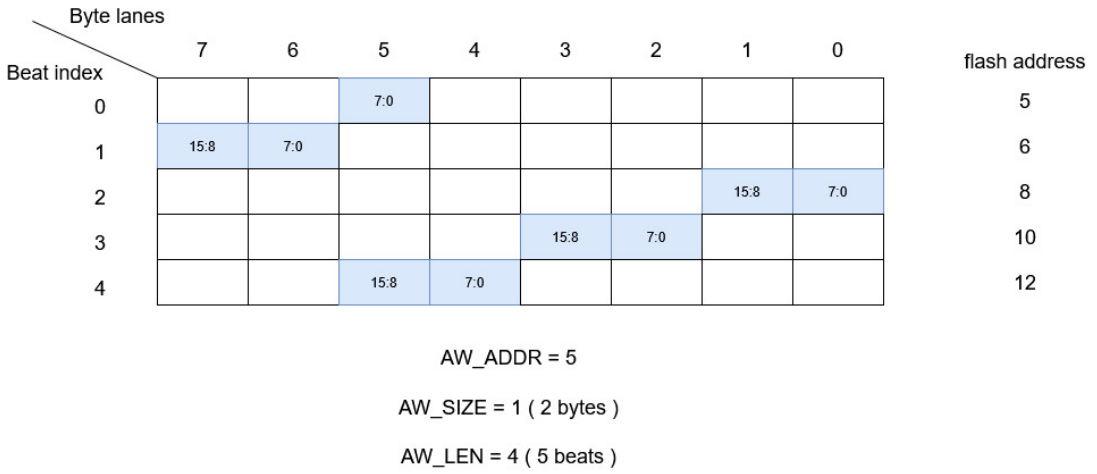


Figure 3.7: Sampling of W_DATA, example 2

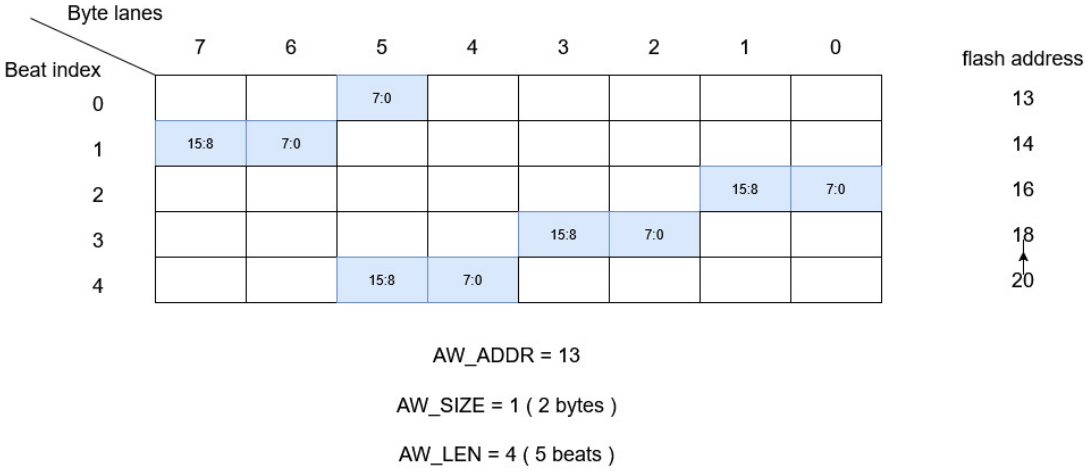


Figure 3.8: Sampling of W_DATA, example 3

Follows a snippet of the code of sub-state AXIREQ_W dedicated to the response to channel W and sampling of the write beats from the correct byte lanes of bus W_DATA, highlighting only the relevant code.

```
1   AXIREQ_W: begin
2     // Answer to the W channel
3     logic [AddrWidth-1:0]                 beat_addr_at_beat_count;
4     logic [sizeInBits(DataBytes-1)-1:0]   lower_byte_lane,
      ↪  upper_byte_lane;
5     logic [AddrWidth-1:0]                 alligned_beat_addr =
      ↪  (first_beat_addr_q / beat_size_q) * beat_size_q;
6     int n;
7
8     // Here write beats from AXI's w_data are put into the wr_beat_queue,
      ↪  however it's needed to be concious about which byte lanes in
      ↪  w_data are active for the current beat
9     // Bytelanes are beat size and adress alligned. The byte lanes to
      ↪  sample for the beat are function of the address of said beat
10    if (w_valid) begin
11      axi2fls_wr_valid = 1'b1;
12      // Define the active bytelanes
13      if (beat_count_q == 0) begin
14        lower_byte_lane = first_beat_addr_q - (first_beat_addr_q /
          ↪  DataBytes)*DataBytes;
15        upper_byte_lane = alligned_beat_addr - (first_beat_addr_q /
          ↪  DataBytes)*DataBytes + (beat_size_q - 1);
16      end
17      if (beat_count_q > 0) begin
18        beat_addr_at_beat_count = alligned_beat_addr +
          ↪  (beat_count_q)*beat_size_q;
19        lower_byte_lane = beat_addr_at_beat_count - (first_beat_addr_q /
          ↪  DataBytes)*DataBytes;
20        upper_byte_lane = lower_byte_lane + (beat_size_q - 1);
21      end
22      // Extract data from the correct byte lanes inside w_data
23      axi2fls_wr_data = '0;
24      axi2fls_wr_be = '0;
25      n = 0;
```

```
26        for (int i = lower_byte_lane ; i <= upper_byte_lane ; i++) begin
27          // axi2fls_wr_data[n*8+7:n*8] =  w_data[i*8+7:i*8];
28          axi2fls_wr_data[n*8 +: 8] = w_data[i*8 +: 8];
29          axi2fls_wr_be[n] = w_strb[i];
30          n++;
31        end
32        [...]
```

3.4.4 top-FSM state: TOP_AXIREQ (cont.)

After AXIERQ_W, the FSM will move to the next top-state: either TOP_READ or TOP_POWERON, that depends on whether the latter is enabled by the external control input poweron_en_i (which resides inside the SPI subsystem register) and only if that procedure has not once been executed since last reset; otherwise TOP_POWERON is bypassed and TOP_READ is executed instead.

3.4.5 top-FSM state: TOP_POWERON

The state diagram of the POWERON sub-FSM is displayed in figure 3.9.

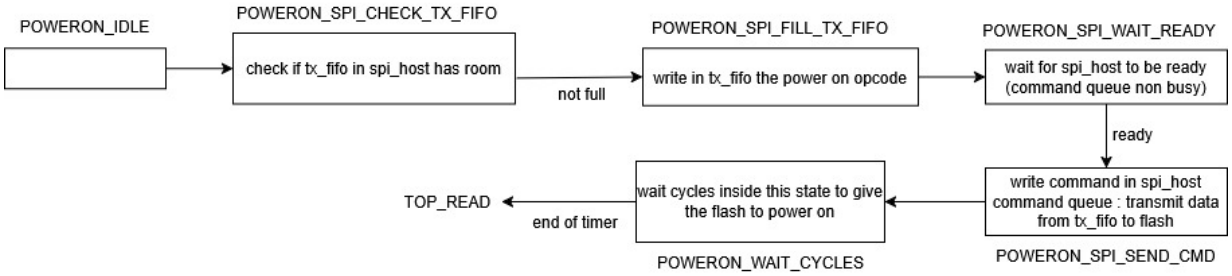


Figure 3.9: State transition diagram of TOP_POWERON

In This state, the FSM sends the flash a byte with the power-on opcode.

This is the first SPI data transmission from SPI manager spi_host to SPI subordinate flash.

From now on, When SPI data is to be sent, the sequence works as the following:

1. spi_host's status register is checked to make sure that the tx_fifo has room for one item.

2. If room is present, the SPI command word is pushed into said FIFO.
3. An additional check is done to make sure spi_host is not busy.
4. If ready, the command word is sent to the command queue of spi_host, which basically defines and triggers the next SPI serial data transfer, specifying the following attributes:
   - Direction - three values : reception, transmission, dummy reads.
   - Speed - two values : quad SPI or standard SPI.
   - CSAAT - indicates whether the Chip Select needs to return to its idle value '1' after the transfer.
   - Length - number of bytes.

The SPI word is supposed to be transferred any moment from the issue of the command, unless more transfers are queued up ahead of it.

After the power-on command has been issued, a number of clock cycles are waited with a counter, counting up to the quantity of microseconds required for the flash to power on after digesting the opcode. Said quantity t_{RES} is defined in the datasheet of the flash in microseconds, and its value in clock cycles is equal to the product between clock frequency in MHz and t_{RES} :

$$N_{wait\ cycles} = t_{RES}\,[\mu s] \cdot f_{CLK}\,[MHz] \tag{3.2}$$

3.4.6 top-FSM state: TOP_READ

The state diagram of the READ sub-FSM is displayed in figure 3.10.

This state is used to send the commands required to read the flash.

It is required for both reads and writes: if the manager is reading, what is read are all the beats from the flash, one by one pushed into a queue; if the manager is writing, what is read is the sector containing the write location of the current beat, byte by byte into the local sector buffer.

In the first state of READ_IDLE, two important things are done: the flash address is evaluated and a condition is tested to see whether, if writing, the whole sector read can be skipped.

The flash address is serially transmitted alongside the read opcode to the flash to trigger the addressed data to leave the flash.
This address points to the position in the flash to start reading, but it represents

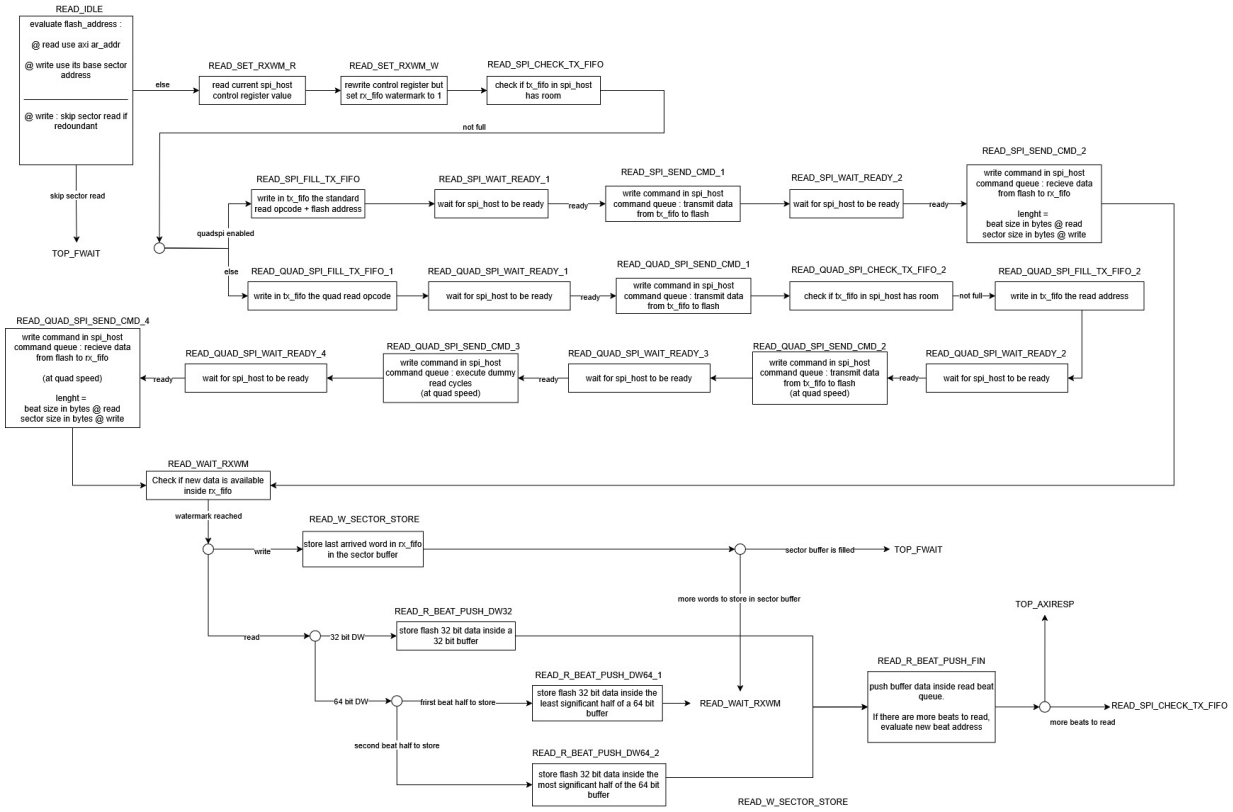


Figure 3.10: State transition diagram of TOP_READ

two different things according to the access mode:
if the manager is reading, it is the beat address; if it is writing, it is the offset of the sector to be containing the write beat, which essentially is the piecewise AND between the beat address and 0xFFF000, since the 12 MSB of the beat address indicate the sector offset.

Since reading a whole sector is a much more time expensive operation than just 8 bytes (max), some effort is paid to see if it can be avoided to be done.
Indeed, it is very possible that in a write burst, the current write beat is located in the same sector of the previous, which means that the flash address to be sent is the same.
Repeating the sector read would just overwrite the sector buffer with the exact content it already holds and was used in the previous write beat iteration to page program that same flash sector, hence it would be redundant: different beats, same buffer data.
So, sector read is avoided, retaining and reusing the content inside the sector buffer from last beat write iteration.

The conditions that are checked to skip the read are:

- "the flash is being written".
- "this is not the first beat of the burst".
- "the previous flash address is equal to the current one", which, when writing, it means "the previous sector is equal to the current one".

Across the whole AXI transaction, a result regarding this improvement is that, during a write burst, a sector read is done only once for each set of consecutive write beats that share the same sector, greatly reducing the number of total clock cycles.

Follows a snippet of the code of sub-state READ_IDLE explaining how the flash address and the condition to skip the sector read are evaluated.

```
1  READ_IDLE: begin
2    // Evaluate flash_addr to send to the flash and skip_sector_read
3    logic skip_sector_read;
4    logic [FlashAddrW-1:0] previous_flash_addr = flash_addr_q;
5    logic [AddrWidth-1:0] sector_address_of_beat = (beat_addr_q >> 12)
     ↪  << 12;
6    // Note : Flash address is byte-precise and 24 bits wide
7
8    // flash_addr evaluation
9    flash_addr_d = rnw_q ? beat_addr_q[FlashAddrW-1:0] :
     ↪  sector_address_of_beat[FlashAddrW-1:0];
10
11   // skip_sector_read evaluation
12
13   // For each write beat, the flash sector that contains it is store
     ↪  in the sector buffer
14   // Skip this operation in case it would be redoundant from last
     ↪  write beat
15
16   // Skip the sector read if :
17   // * Writing
18   // * The sector base address sent to the flash would be the equal to
     ↪  the one of previous write beat
19   // * it is not the first beat
20   skip_sector_read = ((flash_addr_d == previous_flash_addr) && ~rnw_q
     ↪  && beat_count_q > 0 ) ? 1'b1 : 1'b0;
21   // Next state evaluation
22   if (skip_sector_read) begin
23     read_state_d = READ_IDLE;
24     top_state_d = TOP_FWAIT;
25   end else begin
26     read_state_d = READ_SET_RXWM_R;
27   end
28 end
```

After READ_IDLE, what is done is the serial SPI transmission of flash address and correct read opcode, to do this they need to be pushed first into tx_fifo of spi_host, and once the SPI controller command is also pushed into its dedicated queue in the same module, the transmission will be triggered.
After a while, a slot in the rx_fifo of spi_host is expected to be filled with read flash data.

The AXI flash controller, to acquire flash data from rx_fifo as soon as it arrives, sets the watermark of that FIFO to 1 by changing that value in the control register of spi_host.

To send address and read opcode, two different branches can be taken according to the value of the external input quadspi_en_i, which is a bit of spi_subsystem_reg_top selecting the speed of SPI: quad or standard.

In case of quad spi, the read opcode is different, and the SPI controller also needs to be instructed to execute dummy read cycles to match the increase of throughput

the state of the quad spi branch have the READ_QUAD_SPI_ prefix in the state identifier.

Follows a code snippet containing the definitions of the READ sub-states dedicated to the SPI transmission of the flash read request, divided in two branches: standard SPI and quad SPI.

```
    // Standard spi branch
    READ_SPI_FILL_TX_FIFO,   // Write standard read command (FC_RD) +
    ↪  address to tx_fifo
    READ_SPI_WAIT_READY_1,   // Wait for spi_host to be ready
    READ_SPI_SEND_CMD_1,     // Send command to spi_host to release
    ↪  tx_fifo data to flash
    READ_SPI_WAIT_READY_2,   // Wait for spi_host to be ready
    READ_SPI_SEND_CMD_2,     // Send command to spi_host to receive data
    ↪  from flash into rx_fifo

    // Quad spi branch
    READ_QUAD_SPI_FILL_TX_FIFO_1,       // Write quad spi read command
    ↪  (FC_RDQIO) to tx_fifo
    READ_QUAD_SPI_WAIT_READY_1,         // Wait for spi_host to be ready
    READ_QUAD_SPI_SEND_CMD_1,           // Send command to spi_host to
    ↪  release tx_fifo data to flash
    READ_QUAD_SPI_CHECK_TX_FIFO_2,      // Check whether tx_fifo has
    ↪  room
    READ_QUAD_SPI_FILL_TX_FIFO_2,       // Write address to tx_fifo
    READ_QUAD_SPI_WAIT_READY_2,         // Wait for spi_host to be ready
    READ_QUAD_SPI_SEND_CMD_2,           // Send command to spi_host to
    ↪  release tx_fifo data to flash (at quad speed)
    READ_QUAD_SPI_WAIT_READY_3,         // Wait for spi_host to be ready
    READ_QUAD_SPI_SEND_CMD_3,           // Send command to spi_host to
    ↪  execute dummy flash read cycles (at quad speed)
    READ_QUAD_SPI_WAIT_READY_4,         // Wait for spi_host to be ready
    READ_QUAD_SPI_SEND_CMD_4,           // Send command to spi_host to
    ↪  receive data from flash into rx_fifo (at quad speed)
```

You can see that a SPI read request to be issued requires many SPI transfer to happen, and many states are involved to do so:

At standard speed the SPI transfers are:

1. TX standard speed read opcode + address
2. RX flash data

At quad speed the SPI transfers are:

1. TX quad speed read opcode
2. TX address - at quad speed
3. TX command to execute dummy reads - at quad speed
4. RX flash data - at quad speed

Now that the SPI read has been triggered, data needs to be brought into the correct buffer element according to the mode: rd_data_queue if reading, sector buffer if writing.

If reading, for each read beat, the watermark of rx_fifo is polled and then the beat is pushed from that FIFO of spi_host to rd_beat_queue of axi_to_flash_controller.

To support the variable data size, The push of the beat may take one or two individual FIFO transfers, as in the 64 bit case a read beat may be twice the size of a slot of rx_fifo of spi_host, fixed to 32 bits.

After each push in rd_beat_queue, a new read beat address is evaluated, always assuming that the burst type is incremental (AX_BURST = INC always).

Follows a code snippet containing the address evaluation in the read use case (identical to the write use case, but done in a different top-state).

```
1   READ_R_BEAT_PUSH_FIN: begin
2     // Push read beat into the rd_queue
3     spihost_rd_valid = 1'b1;
4     spihost_rd_data = rd_queue_buffer_q;
5     if (spihost_rd_ready) begin
6       beat_count_d = beat_count_q + 1;
7       // If there are more beats to read, beat_address is updated,
        ↪  otherwise TOP_AXIRESP is up
8       if (beat_count_d < beat_number_q) begin
9         // This controller assumes to work only in INCR bursts
10        // addr_0 = ax_addr
11        // addr_0_alligned = INT(addr_0 / size) * size
12        // addr_N = addr_0_alligned + size*N
13        beat_addr_d = ( (first_beat_addr_q / beat_size_q) * beat_size_q)
          ↪  + (beat_count_q + 1)*beat_size_q;
14        flash_addr_d = beat_addr_d; // It is enstablished that the
          ↪  intention is to read the flash
15      end
16    end
17    // Next state evaluation
18    if (spihost_rd_ready && beat_count_d <  beat_number_q) begin
19      read_state_d = READ_SPI_CHECK_TX_FIFO;
20    end
21    if (spihost_rd_ready && beat_count_d == beat_number_q) begin
22      read_state_d = READ_IDLE;
23      top_state_d = TOP_AXIRESP;
24    end
25  end
```

If writing, sector data is just copied in the sector buffer; the watermark of rx_fifo is always polled, and a request is sent to the local memory for each word as soon as present in the FIFO.

3.4.7 top-FSM state: TOP_FWAIT

The state diagram of the FWAIT sub-FSM is displayed in figure 3.11.

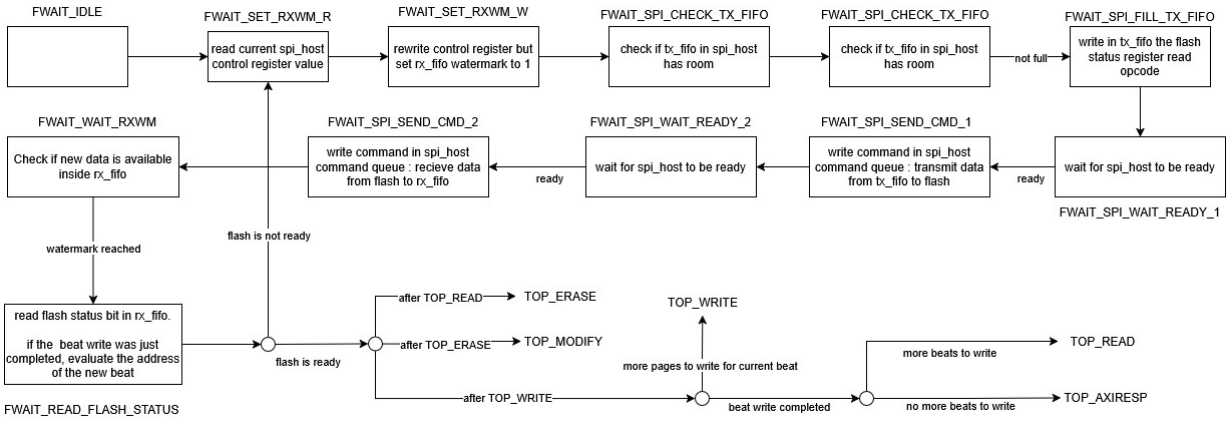


Figure 3.11: State transition diagram of TOP_FWAIT

This state is required to wait for the completion of time expensive flash operations: sector erase and page program; so TOP_FWAIT is executed after TOP_ERASE and TOP_WRITE, but also TOP_READ to make sure that there are no ongoing internal operation from previous iterations.
This state is unused while reading

To be aware of the state of the flash, a opcode is sent to to read Status register 1.
As in a polling process, this is repeated until, according to the value of a bit in the register, the flash appears to be ready for the next operation.

In the final state, the next top-state is decided according to the status that initially called TOP_FWAIT:

- TOP_READ called TOP_FWAIT : execute TOP_ERASE after TOP_FWAIT (erase flash sector)
- TOP_ERASE called TOP_FWAIT : execute TOP_MODIFY after TOP_FWAIT (modify sector buffer)
- TOP_WRITE called TOP_FWAIT , page_count < 16 : execute TOP_WRITE after TOP_FWAIT (program next page)
- TOP_WRITE called TOP_FWAIT , page_count $= 16$, beat_count $<$ beat_number : execute TOP_READ after TOP_FWAIT (begin next AXI transfer)

- TOP_WRITE called TOP_FWAIT , page_count = 16 , beat_count = beat_number : execute TOP_AXIRESP after TOP_FWAIT (complete AXI transaction)

In this state, when a AXI write beat transfer is completed, a new address is evaluated for the next one.

3.4.8 top-FSM state: TOP_ERASE

The state diagram of the ERASE sub-FSM is displayed in figure 3.12.

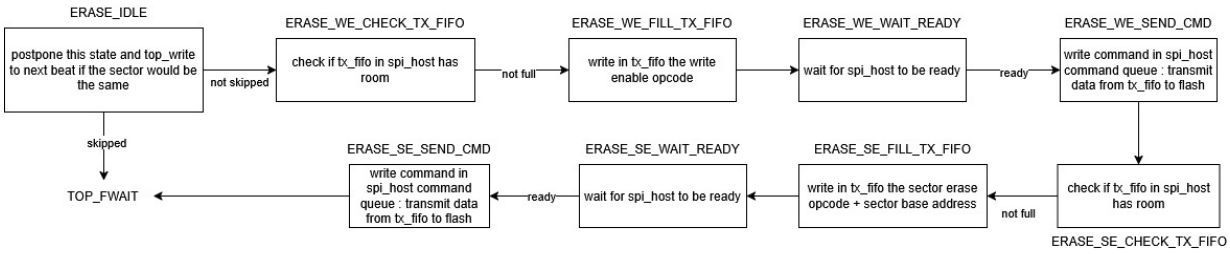


Figure 3.12: State transition diagram of TOP_ERASE

Once again, this top-state is relevant only for flash writes, it is where the sector erase opcode is issued to the flash.

In the first sub-state ERASE_IDLE a condition is tested to see whether it is possible to postpone the erase and program to the next write beat.

If the next write beat would share the same sector, it means that during its TOP_WRITE execution, the current write beat will be present in the modified buffer to be page programmed into the flash sector, making the current TOP_ERASE and TOP_WRITE states redundant.

What is done to evaluate the condition, the address of the next write beat is evaluated in advance to check whether it would share the same sector (upper 12 bits in the address).
This is similar to what is done in TOP_READ to check whether the sector read can be skipped, however the condition depends on the previous beat rather than the next one like here.

The condition that is tested here to bypass the erase, is also valid to skip the sector program for this same AXI write transfer iteration, this is why the outcome of the test is saved in a flip-flop to be checked later in TOP_WRITE.

Bypassing the three sector operations read, erase and program when possible, greatly improves performance; combining these features allows to execute only one of each operation for each subset of consecutive write beats in a burst that share the same sector: sector read is done only for the first beat, sector erase and program are done for the last one.
Because sector operations are the bottleneck of latency, the algorithm complexity would then go from O(N) to O(1) : the number of cycles no longer scales linearly with the length of the burst, but is constant.
This concept will the put to test and discussed in a simulation in Chapter 4.

Follows a code snippet highlighting the erase and write bypass condition

```
1       TOP_ERASE: begin
2         case (erase_state_q)
3           ERASE_IDLE: begin
4             logic [AddrWidth-1:0] next_beat_addr, next_sect_addr,
                ↪  curr_sect_addr;
5             // In this state, before erasing, it is considered the
                ↪  possibility to postpone flash erase and program in the
                ↪  next write beat cycle if the sector involved would be
                ↪  the same
6             // As these operations are the performance bottlenecks and
                ↪  it is better to share them between more beat write
                ↪  instances.
7
8             // This is done if :
9             // * The current sector is not the first of two being
                ↪  written for the current signle beat (sector would change
                ↪  after)
10            // * The current sector is not the last one (would be the
                ↪  last call to write)
11            // * The next sector address would be the same as current
12            if (beat_count_q < beat_number_q - 1) begin
13              next_beat_addr = ((first_beat_addr_q / beat_size_q) *
                  ↪  beat_size_q) + (beat_count_q + 1)*beat_size_q;
14              next_sect_addr = (next_beat_addr >> 12) << 12;
15              curr_sect_addr = '0 | flash_addr_q;   // flash_addr_q
                  ↪  during write transactions is the current sector
                  ↪  address, set in TOP_READ
16              skip_write_d = (next_sect_addr == curr_sect_addr) ? 1'b1 :
                  ↪  1'b0; // This value is reset after each beat write
17            end
18  [          // Next state evaluation
19            if(skip_write_d) begin
20              // skip
21              erase_state_d = ERASE_IDLE;
22              top_state_d = TOP_FWAIT;
```

```
23              end else begin
24                // no skip
25                erase_state_d = ERASE_WE_CHECK_TX_FIFO;
26              end
27            end
```

The following states of TOP_ERASE are dedicated to the SPI transmission of the opcodes required to trigger the sector erase in the flash: first the "write enable" opcode is transmitted, then the "sector erase" opcode, the latter concatenated to the offset of the sector to be erased.

3.4.9 top-FSM state: TOP_MODIFY

The state diagram of the MODIFY sub-FSM is displayed in figure 3.13.

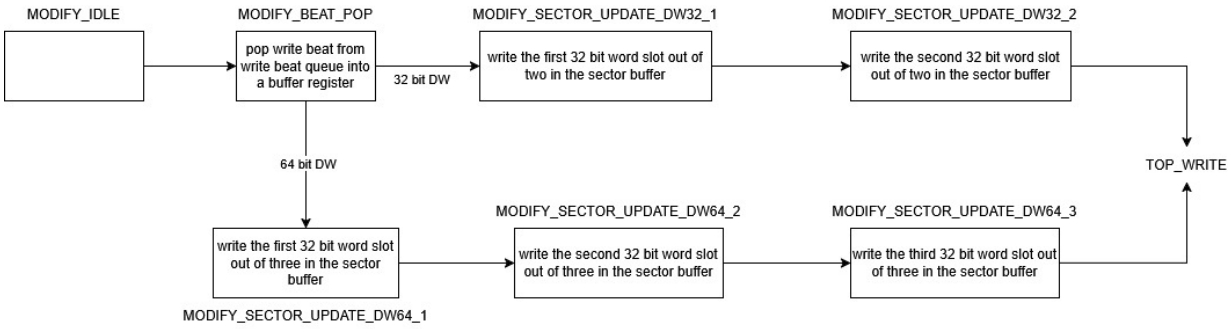


Figure 3.13: State transition diagram of TOP_MODIFY

This state is used to modify the sector content that has been stored in the sector buffer of the module, with the write beat dequeued from wr_beat_queue.

Modifying the sector buffer with the write beat is not as simple as making a single write request with address equal to its sector relative address (least 12 significant bits), additional care needs to be taken as the flash address is byte precise, while the sector buffer SRAM address is 32 bit word precise.

To extract the buffer address from the sector relative flash address, the 2 LSB are taken away, because the address precision moves from byte to 32 bit word. However, depending on the value of these two bits, the beat might not start from the byte position 0 of that word slot in the buffer: a misalignment between word slots and write beat is may happen.

Additionally, since a write beat can be up to 8 bytes long, it can take up to two words slots in the buffer, and if it is misaligned even three; for each slot a separated write request needs to be issued.
For each possible word slot, a state is present in the sub-fsm of TOP_MODIFY to issue a request to said slot in the buffer, executed one after the other.

Assuming 64 bits data width, to issue the correct 32 bits write data to the buffer during each state, a string of 96 bits resembling the three words slots of the buffer is constructed, the write beat from the wr_beat_queue is assigned to it and then left shifted 8*m times where m is the misalignment, equal to the 2 LSBs of the beat address, as mentioned before; then according to what slot is being written (which state we are in), a different slice of 32 bits is provided as write data to the buffer.

Follows a code snippet of the state writing the first slot of the sector buffet to be affected by the write beat, at 64 bits data width.

```
1     MODIFY_SECTOR_UPDATE_DW64_1: begin
2       // Sector_relative_addr = beat_addr & 000fff
3       sector_relative_addr[11:0] = beat_addr_q[11:0];
4       // Sector buffer is word precise, so remove the two LSB that
        ↪  indicate the 4 bytes inside the word
5       sect_buffer_addr = sector_relative_addr >> 2;
6       // Those two LSB indicate the miasslignment between flash address
        ↪  and sector buffer address
7       misallignment = sector_relative_addr[1:0];
8
9       // These two signals indicate how the beat data is distributed
        ↪  across the 96 bits of the three adiacent words
10      buffer_beat_space_data = wr_queue_buffer_data_q <<
        ↪  misallignment*8;
11      buffer_beat_space_be = wr_queue_buffer_be_q << misallignment;
12
13      // write word 1 of 3
14      sect_buffer_wdata = buffer_beat_space_data[31:0];
15      sect_buffer_be = buffer_beat_space_be[3:0];
16
17      sect_buffer_req = 1;
18      sect_buffer_we = 1;
19      // Next state evaluation
20      modify_state_d = MODIFY_SECTOR_UPDATE_DW64_2;
21    end
```

In the 32 bits data width, a separate branch is corssed, where the states are just two, being the maximum write beat size reduced.

In each state, additional conditional logic is present to end early TOP_MODIFY in case the sector relative address of the beat overflows, that is when the last byte of the sector is written; that always means that the final byte of the beat is also written, so there is no need overflow data in the next sector, this is due to how unaligned addresses are handled by AXI, as explained in the TOP_AXIREQ subsection.

Figure 3.14 shows a example of how a write beat is stored in the sector buffer according to the position it would have in the flash.

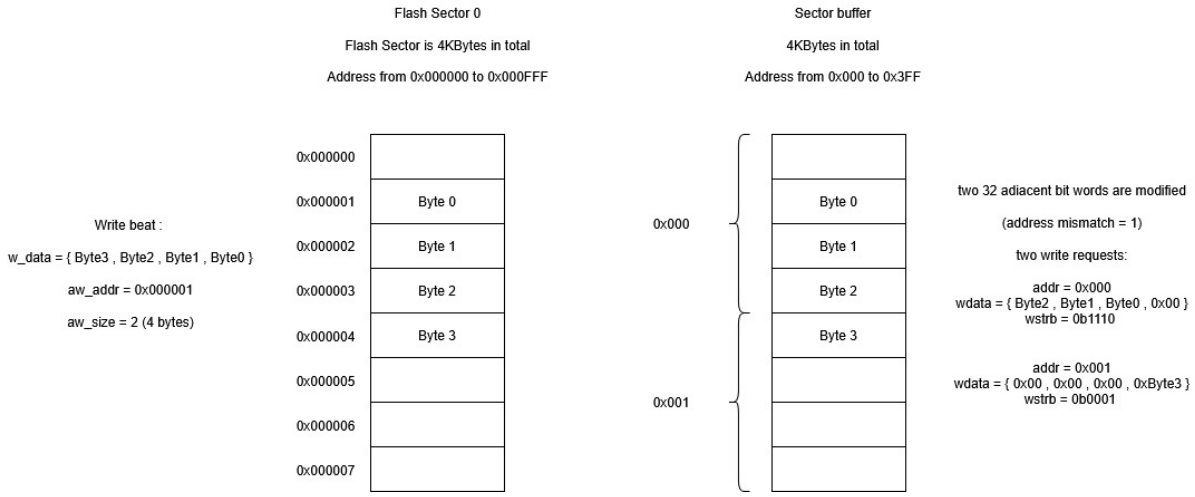


Figure 3.14: Sector data transfer from flash to buffer, weighting on address

TOP_WRITE is always executed after TOP_ERASE.

3.4.10 top-FSM state: TOP_WRITE

The state diagram of the WRITE sub-FSM is displayed in figure 3.15.

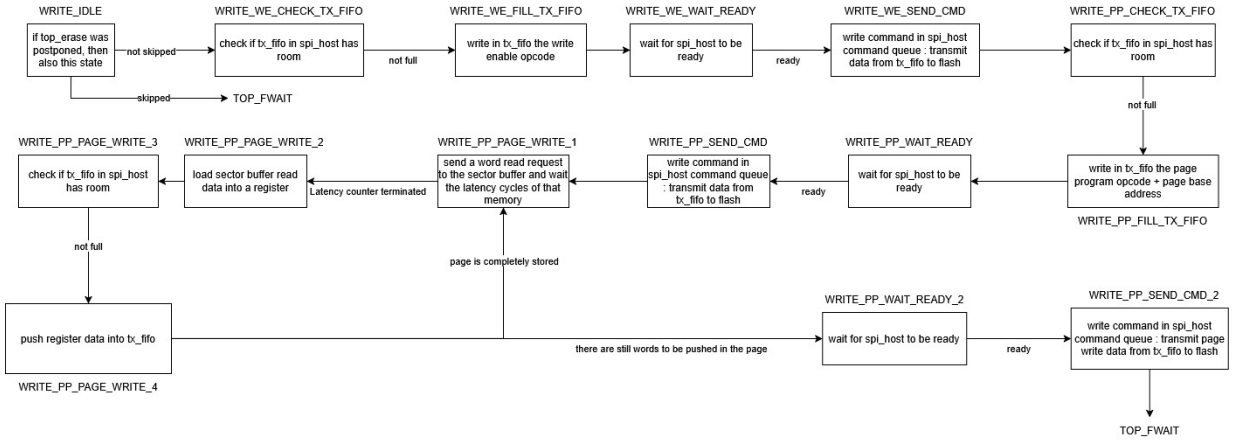


Figure 3.15: State transition diagram of TOP_WRITE

This state is used to page program the flash with the modified sector data from buffer.

It happens page by page, until the whole sector is programmed, so 16 times.

After every page program, top-FSM goes back to TOP_FWAIT.

As mentioned in the TOP_ERASE subsection, the first state WRITE_IDLE is used to check whether the sector program can be postponed to the next write beat iteration, in case it shares the same sector.

The same exact condition that was checked to postpone the sector erase is tested, hence sector erase and write are always skipped together, unlike sector read.
The condition was evaluated and saved in a flip-flop (skip_write) back in TOP_ERASE, and that bit is what is checked in TOP_WRITE.

After testing the skip condition, the steps of this state are:

1. Send opcode "Write Enable"
2. Send opcode "page Program" (one of two possible according to SPI speed) coupled with sector offset + page offset.
   SPI speed is defined by input quadspi_en_i, coming from the register spi_subsystem_reg_top.
3. Fill tx_fifo of spi_host with 256 B page data from sector buffer, word by word.
   For each of the 64 words, a read request is made to the sector buffer, and to sample the outgoing data, access latency cycles are waited with a counter in a separated state.
4. Push command in spi_host to trigger the release of the page data from tx_fifo.
5. Increment page counter (page offset).

3.4.11 top-FSM state: TOP_AXIRESP

The state diagram of the AXIRESP sub-FSM is displayed in figure 3.16.

The state is the one dedicated to the response back to the AXI manager.

In the write case, it is just required to set a "valid" signal and to wait for the "ready" from the manager to complete the handshake and the AXI transaction; in the read case it is also required to drive the AXI data bus R_DATA with the read data beats dequeued from rd_beat_queue, each one complemented with its own valid-ready handshake.

Like when sampling W_DATA, when beat data needs to be driven into R_DATA, the right byte lanes needs to be taken into consideration, these are evaluated the same way as they are in TOP_AXIREQ.

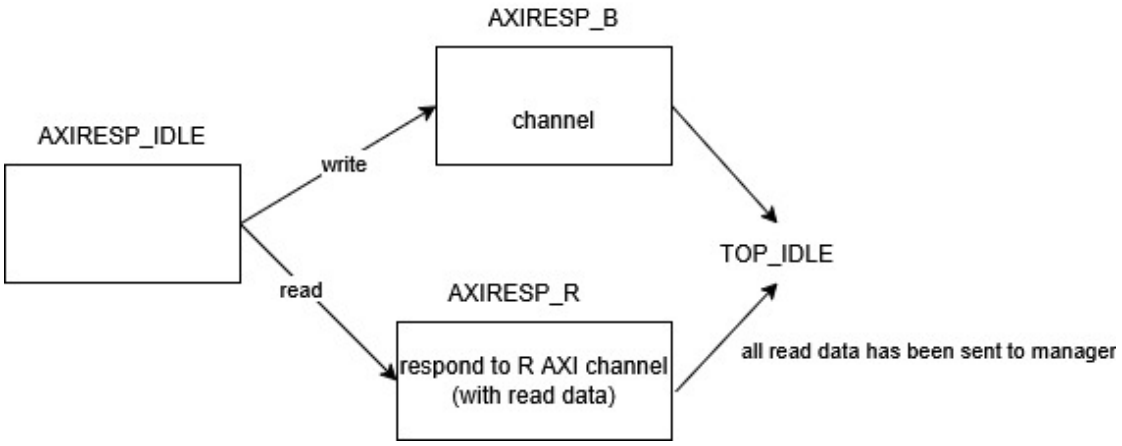


Figure 3.16: State transition diagram of TOP_AXIRESP

Once the response and this state are completed, the FSM goes back to TOP_IDLE hence the other sub-FSM's also are set to their idle state.

3.5 HDL Source Code

The source code in SystemVerilog of the axi_to_flash_controller module can be found at: `https://github.com/alebaroc/xheep_spi/blob/first_impl/rtl/axi_to_flash_controller.sv`.

Chapter 4

Results

4.1 Simulation

The simulation consists in a program inside the Verilator testbench spi_subsystem_tb.cpp, which can be found at `https://github.com/alebaroc/xheep_spi/tree/first_impl/tb`.

In this simulation, the DUT is implied to be a 64 bit system, however the testbench can also assume a 32 bit width.

4.1.1 Simulation Program

The first thing done by the program is to configure the DUT by writing the control registers of the SPI subsystem and SPI controller spi_host:

the bits of spi_subsystem_reg_top are set as following:

- **use_axi** is set to enable and select the AXI flash controller
- **a2f_ctr_poweron** is set to power on the flash before the first read, inside the state TOP_POWERON.
- **a2f_ctr_quadspi_en** is cleared to work with starndard SPI speed, since the flash model used in the simulation could not be used in quad spi speed.

Then, the bits of spi_host_reg_top are set as following:

- **sw_rst** is pulsed to issue a software reset of the whole SPI controller (memory elements, fsm).
- **csid** is set to 0 to enable the correct active SPI subordinate line.

- **intr_enable** is set to 0b11 to enable all interrupts to be sent from the SPI controller: unused, but they can be used as a way to monitor the system (same for events and errors.
- **event_enable** is set to 0b111111 to enable all event sources to set their respective event flag.
- **error_enable** is set to 0b11111 to enable all error sources to set their respective error flag.

Then, the SPI controller control bits directly related to the SPI protocol policy need to be set accordingly to the necessity of the flash model that we are working with.

- **clkdiv** = 0 (ratio between local clock frequency and SCK)
- **csnidle** = 0 (amount of cycles between two adjacent transactions where CS needs to be deasserted)
- **csntrail** = 1 (amount of cycles between CS assertion and first SCK edge)
- **csnlead** = 1 (amount of cycles between last SCK edge and CS de-assertion)
- **fullcyc** = 1 (a full SCK cycle is waited before sampling, otherwise sampling is done after half cycle)
- **cpha** = 0 (data is sampled during falling edge of SCK)
- **cpol** = 0 (idle value of SCK)

Finally, FIFO watermarks are set to their reset value for safety reasons (will also be changed by the AXI flash controller to the working value) and the SPI interface is enabled:

- **output_en** set to 1.
- **spien** set to 1
- **rx_watermark** = 0x7F
- **tx_watermark** = 0x00

After the control registers configuration, the simulation program begins to simulate the SPI subsystem as it accesses a flash in a 64 bit system - a AXI write request is issued with the following attributes:

- **AW_ADDR** = 0x021FFF
- **AW_LEN** = 5
- **AW_SIZE** = 1 (2 bytes)
- **AW_BURST** = INC (implied)
- **W_DATA** = 0x3210000000000000, 0x0000000000007654, 0x00000000BA980000, 0x0000FEDC00000000, 0x1111000000000000.
  The beats are transmitted in this order.
  The useful 2 bytes shift and wrap as required from the AXI protocol because the active byte lanes change with the address of such bytes, as explained in Chapter 3, TOP_AXIREQ section.
- **W_STRB** = 0xFF (data byte lanes are all always enabled, but only some lanes are sampled)

This kind of AXI request implies:

- The least important half of the first beat is truncated having a unaligned address, this is done to assure alignment for the following beats.
- From the second beat a different sector is involved, changing from 0x021 to 0x022.

Then the write transaction is followed by a read transaction which intends to verify the former, addressing the same exact data, hence its attributes are the same.

4.1.2 Simulation Snapshots

Snapshots of waveforms generated from the simulation via Verilator are reported, with chronologic order:

- Figure 4.1 : AXI WRITE - Sampling and pushing of write data beats from AXI bus W_DATA into wr_beat_queue, during AXI request handling by the flash controller.
- Figure 4.2 : AXI WRITE - Sector data being stored in the sector buffer
- Figure 4.3 : AXI WRITE - Sector data being erased in the flash
- Figure 4.4 : AXI WRITE - Sector buffer being modified with AXI beat data

- Figure 4.5 : AXI WRITE - Sector buffer data being written in the flash
- Figure 4.6 : AXI READ - AXI beat data being read from flash and pushed into rd_data_queue.
- Figure 4.7 : AXI READ - Dequeuing and driving of read data beats from rd_data_queue into AXI bus R_DATA, during AXI response.

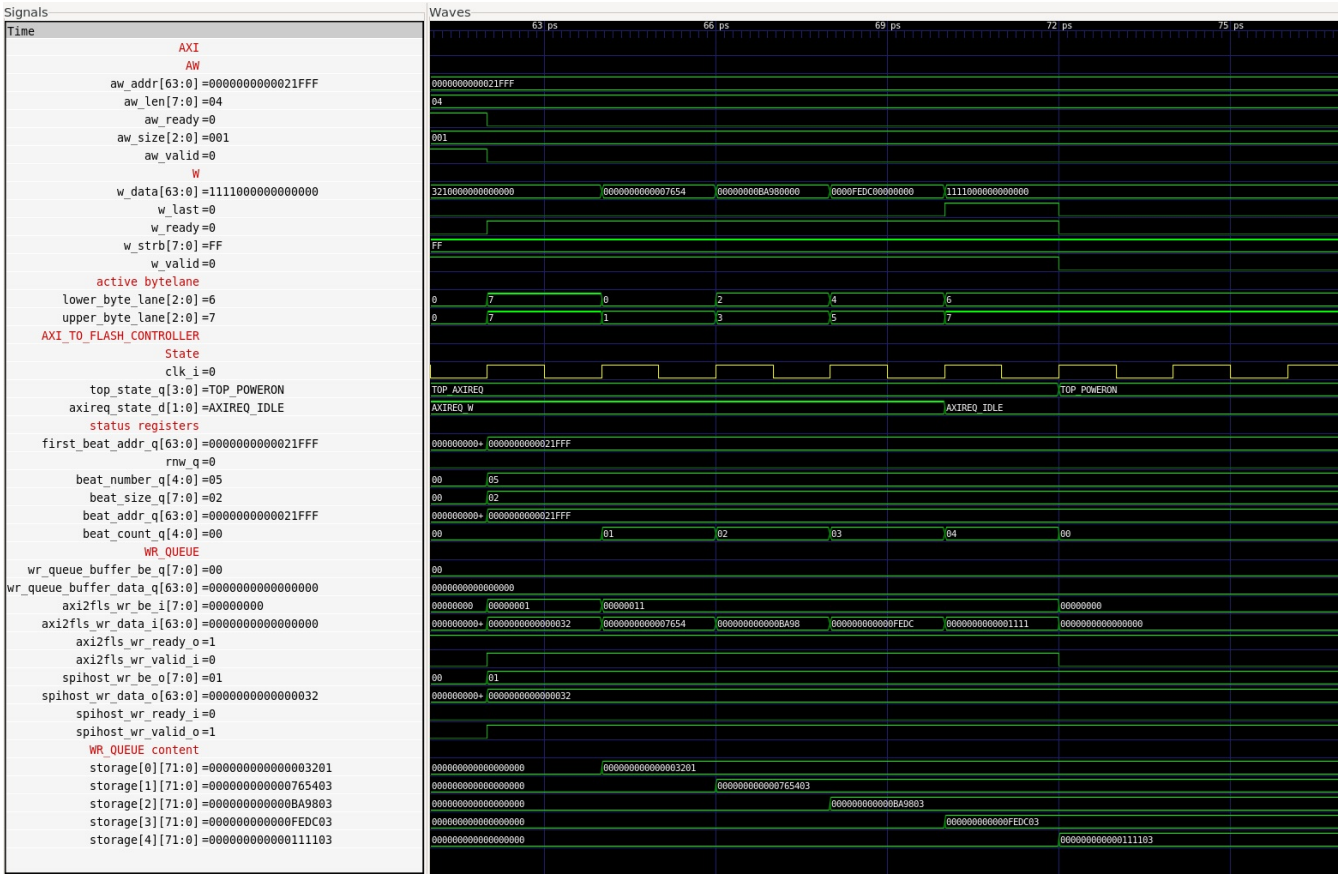


Figure 4.1: Simulation snapshot 0 : AXI WRITE - Sampling and pushing of write data beats from AXI bus W_DATA into wr_beat_queue, during AXI request handling by the flash controller. The active byte lanes shift and wrap around every write beat from the AXI manager according to the address. Notice how the first beat 0x3210 is truncated to 0x32 due to address nonalignment as it is stored into wr_beat_queue.

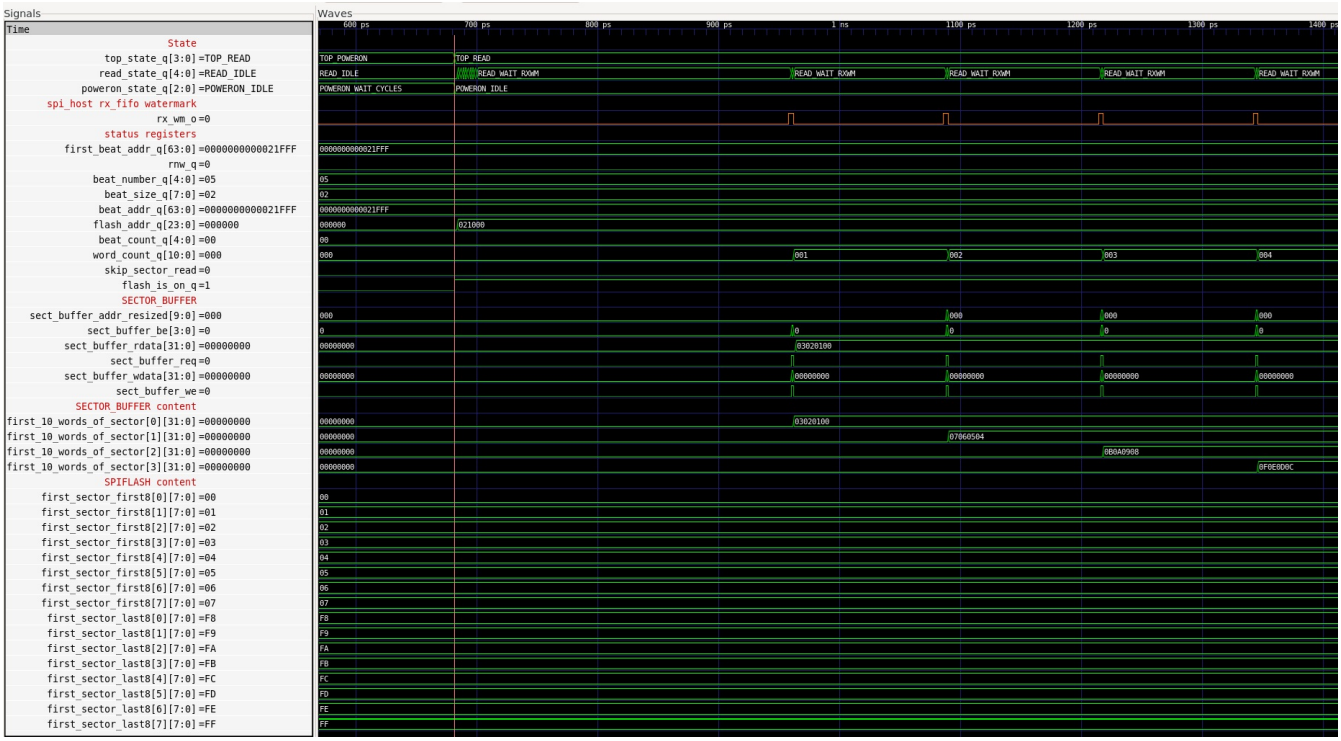


Figure 4.2: Simulation snapshot 1 : AXI WRITE - Sector data being stored in the sector buffer. Content from flash sector 0x021 with bytes initialized to 00, 01, 02, ... , FE, FF, 00, 11, ... 0x021 is the sector of only the first beat transfer of the AXI transaction.

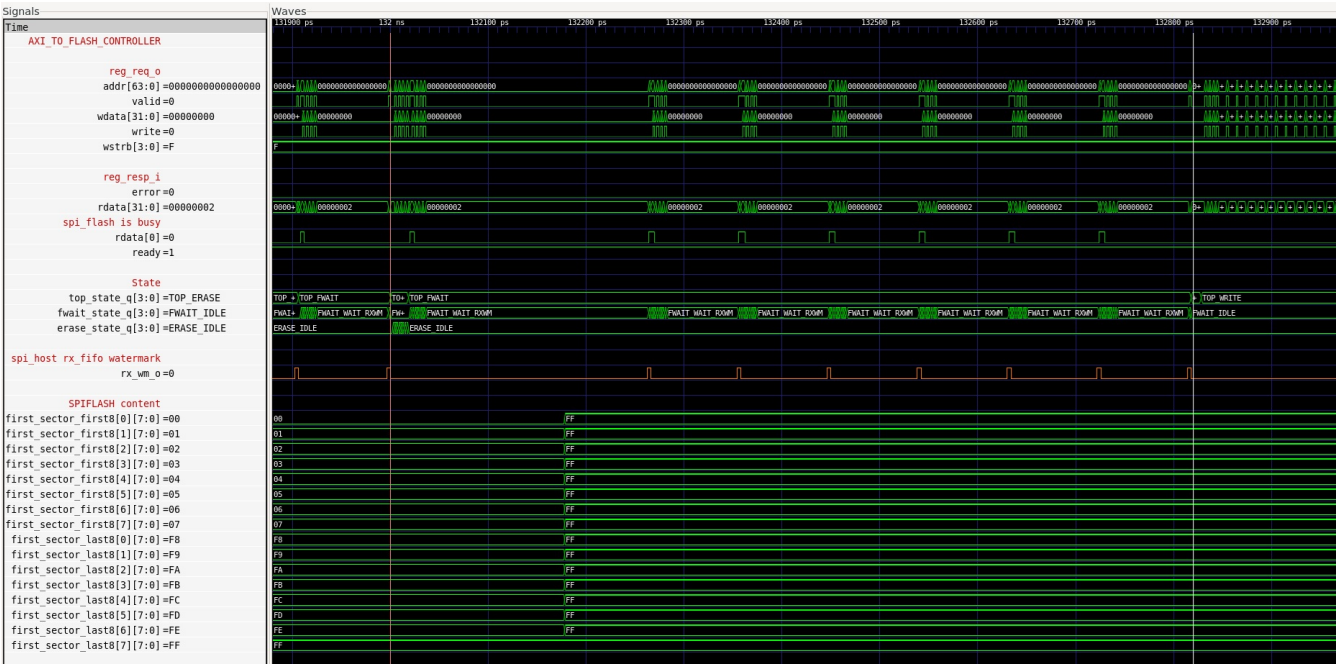


Figure 4.3: Simulation snapshot 2 : AXI WRITE - Sector data being erased in the flash. After the sector erase, the flash (not)ready bit is polled from the flash status register repeatedly until it becomes '0' (see rdata[0] from reg_resp_i), only then the FSM moves from TOP_FWAIT to TOP_WRITE.

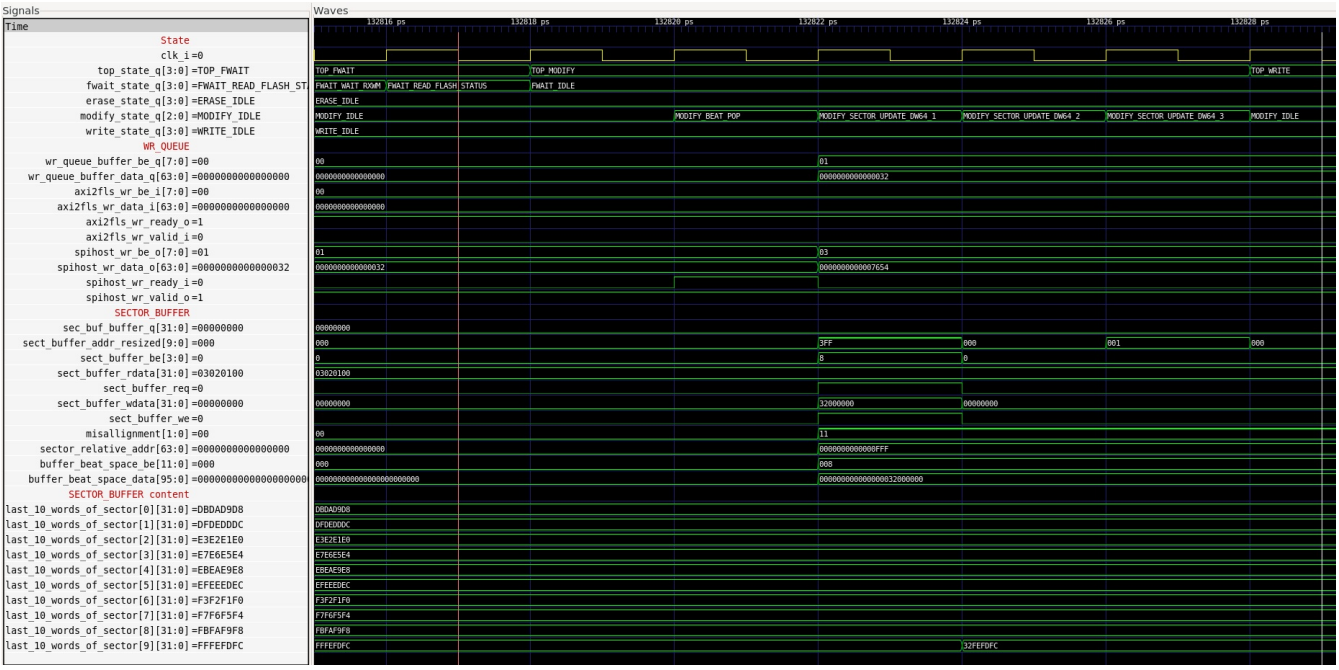


Figure 4.4: Simulation snapshot 3 : AXI WRITE - Sector buffer being modified with AXI beat data. Buffered sector is 0x021, the first write beat modifies its content with the value 0x32 at the last position.

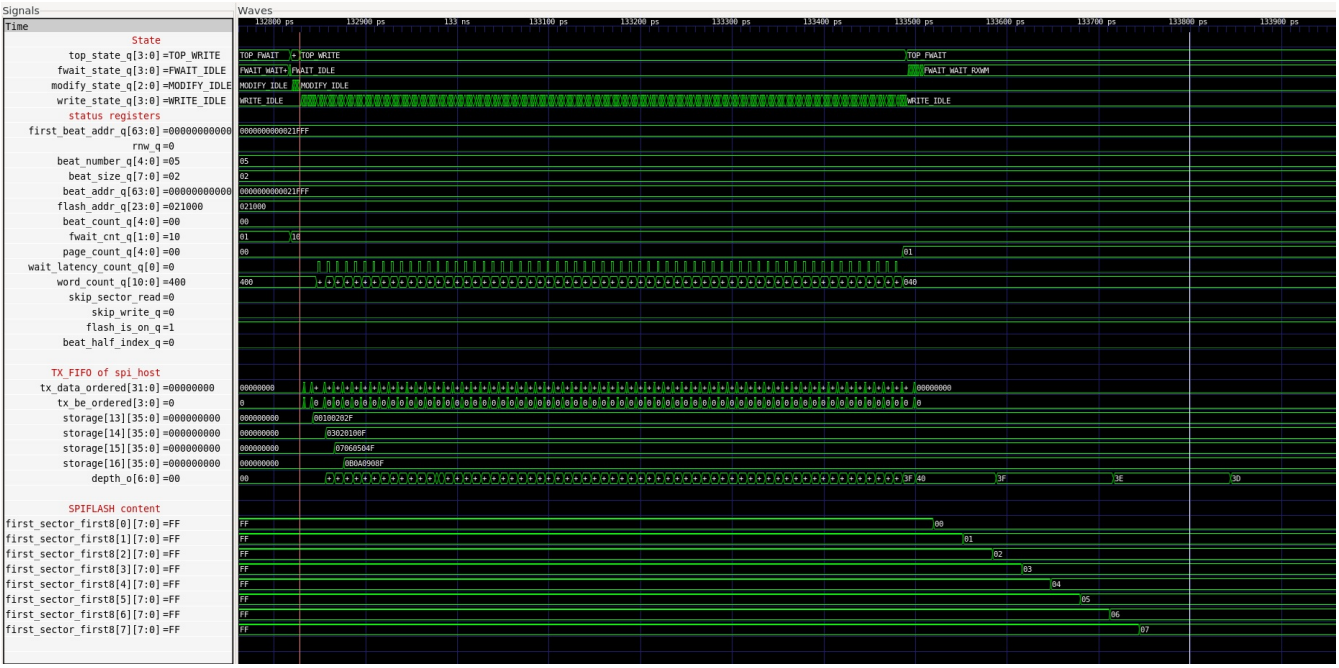


Figure 4.5: Simulation snapshot 4 : AXI WRITE - Sector buffer data being written in the flash. The first page of the sector 0x021 is written, its first 8 bytes are shown.



Figure 4.6: Simulation snapshot 5 : AXI READ - AXI beat data being read from flash and pushed into rd_data_queue. All 5 the beats are read, the first one from sector 0x021 and others from sector 0x022, then stored into rd_beat_queue.

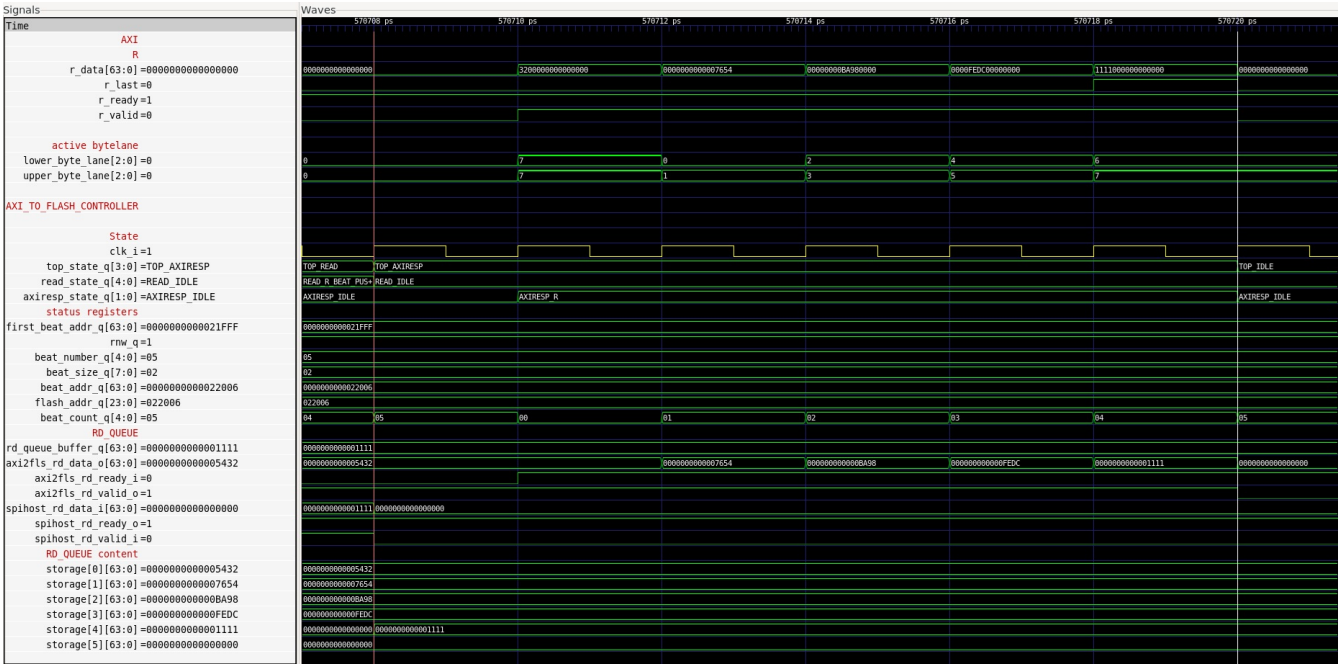


Figure 4.7: Simulation snapshot 6 : Dequeuing and driving of read data beats from rd_data_queue into AXI bus R_DATA, during AXI response. The active byte lanes shift and wrap around every write beat from the AXI manager according to the address. Notice how the first beat 0x3210 is truncated to 0x32 due to address nonalignment as it is stored into wr_beat_queue.

4.1.3 Effect of Performance Enhancement Measure: Sector READ/WRITE Bypass

Finally, a snapshot including the whole program is reported in figure 4.8; its purpose is to show the effects of the performance enhancing measures by bypassing sector operations: skipping redundant sector reads and postponing redundant erases and programs.

The concepts on how and why this is done are explained in detail in the respective FSM state sections in Chapter 3 : TOP_READ, TOP_ERASE, TOP_WRITE.

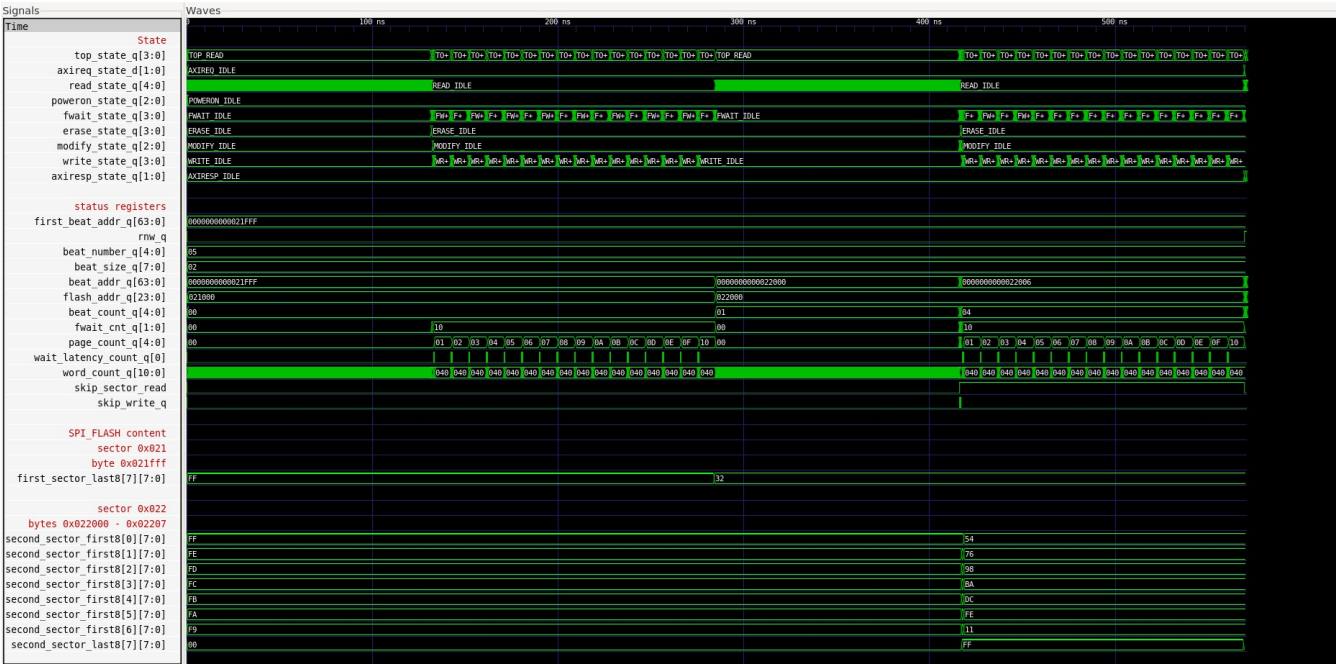


Figure 4.8: Full Simulation snapshot , Notice by looking at beat_count how the write beats 2,3 are much shorter than the others to process thanks to the sector processing bypass mechanism.

From the snapshot of figure 4.8 can be extrapolated that the vast majority of the clock cycles is dedicated to the AXI write transaction; more precisely, this is due to the time overhead of the sector read, erase and program operations. Indeed most of the time is either spent inside the state TOP_READ , where the whole sector is read from the flash into the buffer, or between TOP_WRITE and TOP_FWAIT, where each page is being programmed and consequently the status register 1 of the flash is polled until the "ready" value is detected.

The flash latency regarding these operations is simulated by its RTL model with a counter that counts up to a constant value, it's assumed that these wait periods are somewhat reflective of what can be expected from the physical flash.

For what regards the performance enhancement that comes with bypassing sector read and write, the focus needs to shift to the beat_count value: this value is the index of the current AXI beat that is subject of the ongoing flash write or read operation.
It can be seen that the beats 2,3 are very short in time duration compared to 0,1,4. That is because the burst subset composed of beats from 1 to 4 concerned the same sector: 0x022 ; therefore, regarding this set, despite being 4 beats, the sector read,

erase and write are done only once each to avoid redundancy:

- beat_count = 0 : one sector read, one sector erase and write are executed regarding sector 0x021, being the only beat concerning it in this AXI transaction.
- beat_count = 1 : sector read is executed : sector 0x022 is stored in the buffer in place of 0x021. Sector erase and write are skipped.
- beat_count = 2 : no sector operation is executed, the sector buffer is modified only. Sector read, erase and write are skipped.
- beat_count = 3 : no sector operation is executed, the sector buffer is modified only. Sector read, erase and write are skipped.
- beat_count = 4 : sector 0x022 erase and write are executed. Sector read, was skipped.

The total number of clock cycles of the AXI transaction is approximately $2.8{\cdot}10^5$; it is estimated that for each sector read-write pair $1.4{\cdot}10^5$ cycles are required, so, without this performance enhancement, the total would have been instead $(1.4 \cdot 10^5) \cdot 5 = 7 \cdot 10^5$;

4.2 Comparison between flash controllers

This section is dedicated to the comparison between w25q128jw_controller and axi_to_flash_controller in terms of performance, counted in clock cycles, the former is simulated by running an app in the X-HEEP platform.

Both are simulated by writing and then reading the same addresses in the flash to test the outcome of the write.

Two tests are made: one test features a AXI transaction with burst length of 10 consecutive 4 bytes data beats with incremental address, the other just one data item. In both tests a read transaction follows to verify the outcome of the write transaction. The address is such to allow all transfers to happen within the same sector, and the SPI speed is standard. The outcome is represented in table 4.1.

For what regards the performance difference as function of the burst length in any flash controllers, the cycle count doesn't increase a lot with the burst length because most cycles are dedicated to the flash sector read, erase and write latency and, if the involved sector doesn't change across the burst, these operations are

Table 4.1: Performance comparison between the two SPI subsystem flash controllers.

Sim context	w25q128jw_controller	axi_to_flash_controller
Single Access	241 702 Cycles	142 466 Cycles
Burst Access	242 802 Cycles	145 067 Cycles

not repeated each single data transfer (algorithm complexity doesn't scale linearly, but like a constant).

For what regards the difference of performance about the two at any burst length, the additional latency in w25q128jw_controller comes with the involvement of the DMA: write data, read data, and sector buffer data are all stored in the same external RAM.
These data exchanges involve the DMA and are more time-expensive exchanges than just driving or sampling from a local bus or memory element: the AXI data bus or the local smaller SRAM.
Although local memory elements come with a price in area.

X-HEEP, the platform used to test w25q128jw_controller, offers a tool called "riscv profiler" which generates a Flame chart plotting the distribution of all the clock cycles of the SPI write C++ application used to test the DUT across all the functions that are called, dissecting all the stages of operation of the flash controller and the performance bottlenecks.

This chart, improved with some annotations, is reported in figure 4.9 (burst length of 10 items of 4 bytes) .

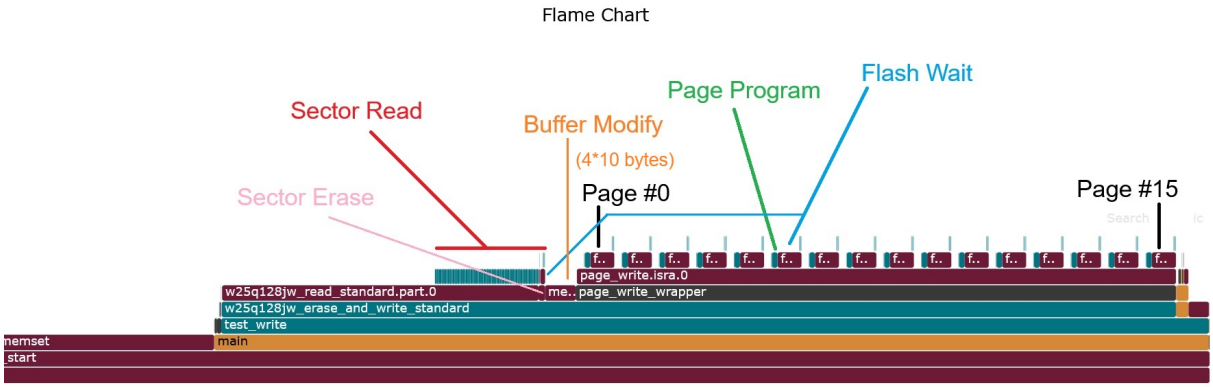


Figure 4.9: Chart profiling the w25q128jw_controller stages during the simulation

The involvement of the DMA makes short operations like Buffer Modify non negligible in duration: more than 6000 cycles, unlike axi_to_flash_controller, where it is just 5 clock cycles per write beat: 2 to extract the write beat from the queue and 3 to write each possible word slot in the sector buffer.

Chapter 5

Conclusion

5.1 Advantages

The AXI flash controller most outstanding points of merit are :

- Parallelism flexibility, making it flexible enough to be deployed in any 32 or 64 bit architecture equipped with a AXI peripheral interface.
- No DMA and RAM dependency, enabled by internal data and sector buffering, increasing performance and reducing interconnection complexity.
- High flash write performance, bypassing sector operations whether possible.
- Features the ability to autonomously power on the flash
- Burst flash accesses, minimizing processor overhead.
- Grants the Manager a RAM-like approach to exchange data with the Subordinate, completely unaware of any additional complexity involving the nature of SPI and flash handling, for both read and write operations.

5.2 Future improvements

To improve the performance of the flash controller, at the expense of the area, a cache like approach can be implemented by creating multiple copies of the sector buffer each containing a different sector that was perviously recently accessed.

For instance a queue of N sectors that is updated with LRU cache logic, exploiting the key concepts of temporal locality to regain access of a specific sector that was used recently without repeating the performance costly operations of sector read, erase, write.

However, any other policy can be used to update the cache other than LRU (for example MFU); or rather than temporal, spacial locality can be exploited, hence keeping memorized sectors that are adjacent in memory rather than sectors that have been accessed in adjacent temporal iterations.

Bibliography

[1] Simone Machetti, Pasquale Davide Schiavone, Giovanni Ansaloni, Miguel Peón-Quirós, and David Atienza. «X-HEEP: An Open-Source, Configurable and Extendible RISC-V Platform for TinyAI Applications». In: *2025 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. 2025. DOI: 10.1109/ISVLSI65124.2025.11130281 (cit. on p. 1).

[2] Arm Limited. *AMBA AXI Protocol Specification*. IHI 0022. Version L. Arm Limited. URL: https://developer.arm.com/documentation/ihi0022/latest (visited on 09/19/2026) (cit. on p. 3).

[3] A. Patra and L. M. Saini. «Analysis of Serial Peripheral Interface». In: *2023 2nd IEEE International Conference on Measurement, Instrumentation, Control and Automation (ICMICA)*. 2024. DOI: 10.1109/ICMICA61068.2024.10732859 (cit. on p. 7).

[4] Winbond Electronics Corporation. *W25Q128JW: 1.8V 128M-Bit Serial Flash Memory with Dual, Quad SPI*. W25Q128JW. Version B. Winbond Electronics Corporation. Nov. 2019. URL: https://www.mouser.com/datasheet/2/949/W25Q128JW_RevB_11042019-1761358.pdf (visited on 09/19/2026) (cit. on p. 17).