Bootloader for a RISC-V processor that uses Flash Memory

José Heraldo Furtado Fernandes

Thesis to obtain the Master of Science Degree in

Electrical and Computer Engineering

Supervisor(s): Prof. José João Henriques Teixeira de Sousa

Examination Committee

Chairperson: Prof. Francisco André Corrêa Alegria
Supervisor: Prof. José João Henriques Teixeira de Sousa
Member of the Committee: Prof. Marcelino Bicho dos Santos

September 2021

Declaration

I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

Acknowledgments

I would like to thank my family and my uncles and aunts families. And I would also like to address a special thanks to my supervisor, Professor José T. de Sousa, for the always useful insight, and to thank my colleague at IObundle, João Lopes, for the precious help on technical implementation details.

Resumo

Este trabalho propõe um novo subsistema IP de controlador de memória flash SPI que pode ser integrado num Sistema-num-Chip (SnC) para executar programas diretamente na flash, e servir como um meio de armazenamento permanente de dados de uso geral.

O novo controlador apresenta parâmetros flexíveis de síntese e de tempo de execução, e suporta o protocolo QSPI e o modo Executar-no-Sítio. O subsistema faz interface ao barramento de instruções do CPU e pode ser acessado como um periférico por programas firmware. Um driver de software escrito em C também foi desenvolvido.

O subsistema foi desenvolvido e integrado na plataforma IOb-SoC, um template de SnC RISC-V de código aberto disponibilizado pela empresa de IP, IObundle, sediada em Lisboa. IOb-SoC facilita o design de SnCs automatizando o processo de inclusão de subsistemas IP adicionais e software.

O programa bootloader do IOb-SoC foi melhorado para fazer uso do novo driver de software; o programa configura o controlador, carrega o programa firmware nele e reinicia o CPU para executar o programa. O bootloader suporta funcionalidades adicionais tais como apagar sectores de memória flash e inspecionar localizações de memória flash.

Os resultados de implementação em FPGA e ASIC são comparados com um muito conhecido subsistema IP comercial e são mostrados serem competitivos dado que os recursos de hardware são muito similares. Em termos de performance, o subsistema é quatro vezes mais lento mas, em compensação, dispensa o recurso a dois domínios de clocks, o que reduz a sua complexidade e traz benefícios de integração.

Palavras-chave: Bootloader, controlador de memória flash SPI, Barramento de Instruções do CPU, Executar programas diretamente na flash, armazenamento permanente de dados de uso geral

Abstract

This work proposes a new SPI flash memory controller IP core that can be integrated into a System-on-Chip (SoC) for running programs directly on the flash, and serving as a general-purpose permanent data storage available to firmware programs.

The new controller features flexible synthesis and run-time parameters and supports the QSPI protocol and the Execute-In-Place mode. The core interfaces the CPU instruction bus and can be accessed as peripheral by firmware programs. A software driver written in C has also been developed.

The core has been developed and integrated into the IOb-SoC platform, an open-source RISC-V SoC template made available by the Lisbon-based IP company IObundle, Lda. IOb-SoC facilitates the design of SoCs by automating the process of adding additional IP cores and software.

The IOb-SoC bootloader program has been upgraded to make use of the new software driver; it configures the controller, loads the firmware program onto it and restarts the CPU to run the program. The bootloader supports additional functionalities such as erasing flash memory sectors and inspecting flash memory locations.

The FPGA and ASIC implementation results are compared to a well-known commercial IP core and are shown to be competitive as the hardware resources are quite similar. In terms of performance, the developed IP core is four times slower but, in compensation, it dispenses with the use of two clock domains, which reduces its complexity and brings integration benefits.

Keywords: Bootloader, SPI Flash memory controller, CPU Instructions bus, Running programs directly on the flash, General-purpose permanent data storage

Contents

Declaration of originality . . . iii
Acknowledgments . . . v
Resumo . . . vii
Abstract . . . ix
List of Tables . . . xv
List of Figures . . . xvii
List of Acronyms . . . xix

1 Introduction 1
1.1 Topic Overview . . . 1
1.2 Motivation . . . 2
1.3 Objectives and Deliverables . . . 2
1.4 Thesis Outline . . . 2

2 Background 5
2.1 IOb-SoC Architecture . . . 5
2.1.1 Architecture Components . . . 5
2.1.1.1 RISC-V CPU . . . 5
2.1.1.2 Memory Subsystem . . . 6
2.1.1.3 Peripheral Cores . . . 6
2.1.1.4 Interconnect and Buses . . . 6
2.1.1.4.1 Native Interface . . . 6
2.1.2 File Structure . . . 7
2.1.3 Operating Modes . . . 8
2.1.3.1 Memory Operating Modes . . . 8
2.1.4 Memory Map . . . 9
2.1.5 Development Flow . . . 10
2.2 Development Board . . . 10
2.2.1 Xilinx KU040 board features . . . 10
2.2.1.1 User Code/Data QSPI Flash . . . 10
2.3 Flash memory . . . 11

2.3.1 NOR vs NAND Flash 11
2.3.2 NOR flash interfaces 12
2.4 SPI communications protocol 12
2.4.1 SPI signals description 12
2.4.2 SPI Transaction 13
2.4.3 SPI Clocking modes 13
2.4.4 xSPI standard 13
2.4.5 Quad SPI 14
2.4.5.1 Signals description 14
2.4.6 Serial Flash Operation Modes 14
2.4.7 QSPI Transaction Format 15
2.4.7.1 Command Phase 15
2.4.7.2 Address Phase 16
2.4.7.3 Latency Phase 16
2.4.7.4 Data Phase 16
2.4.8 Command Transaction Waveforms 17
2.4.8.1 Write Enable on mode (1-1-1) 17
2.4.8.2 Reset Enable on mode (2-2-2) 17
2.4.8.3 Read Status Register mode (4-4-4) 17
2.4.8.4 Page Program mode (1-1-1) 18
2.4.8.5 Fast Read (1-2-2) 18
2.4.8.6 Fast Read (1-4-4) 19
2.4.9 Flash Device Functionalities (SFDP) 19
2.5 SPI Flash Memory Controllers 20
2.5.1 Comparative Flash Controller Model 20

3 On-Board Flash Memory **23**
3.1 N25Q256A Features Introduction 23
3.2 Memory Configuration 24
3.3 Status and Configuration Registers 24

4 Core Development **26**
4.1 Core functionality features 26
4.2 Files Directory Structure 27
4.3 Top Module 27
4.3.1 Blocks Description 27
4.3.1.1 Peripheral Interface Description of Signals 27
4.3.1.2 Instructions Interface Description of Signals 28
4.3.1.3 SPI master interface 28
4.3.1.4 SW Accessible Registers 28

4.4 Central Core . . . 30
4.4.1 Central Core Inputs and Outputs . . . 30
4.4.2 Global State Machine . . . 30
4.4.3 Component Modules . . . 32
4.4.3.1 Sclk_gen module description . . . 32
4.4.3.2 Configdecoder module descrition . . . 33
4.4.3.3 Latchspi module description . . . 33
4.4.4 Configuration Parameters Encoding . . . 35
4.4.4.1 Command type . . . 35
4.4.4.2 SPI Data Lanes modes . . . 35
4.4.4.3 Transaction Frame Struct . . . 36
4.4.4.4 DTR mode enable . . . 37
4.4.4.5 4-byte address mode . . . 37
4.4.4.6 XIP mode . . . 38
4.4.4.7 Number of data bits . . . 38
4.4.4.8 Number of dummy cycles . . . 38
4.4.4.9 Serial Clock Edges . . . 38
4.4.5 Transaction Control Registers . . . 38
4.5 Software Driver . . . 39
4.5.1 Software Driver Global Variables and High Level Functions . . . 42

5 Bootloader and Core Integration to SoC 45
5.1 Core Integration to SoC as Peripheral . . . 45
5.2 Core Integration to SoC as an Instruction Memory Interface . . . 46
5.2.1 Cache Integration . . . 46
5.2.2 Modifications to the CPU module . . . 47
5.3 Bootloader with Flash Functionality . . . 48
5.3.1 Bootloader RUN_FLASH function . . . 48
5.3.2 Bootloader PROGRAM_FLASH function . . . 49
5.3.3 Bootloader CHECK_FLASH function . . . 49
5.3.4 Bootloader SECTOR_CLEAR function . . . 51
5.4 Simulation and Board Run . . . 52
5.4.1 Core Simulation . . . 52
5.4.1.1 Testbench file code segment . . . 52
5.4.2 Core FPGA Synthesis and Implementation . . . 53
5.4.3 IOb-SoC Simulation and FPGA Board Running . . . 53

6 Results 55
6.1 Flash Core Comparative Results . . . 55
6.2 Performance Results . . . 56

6.2.1 Experimental Setup 56
6.2.2 Results Analysis 57
6.2.3 Experimentation Setup 2 57
6.2.4 Results Analysis 2 60
6.3 FPGA Implementation Results 60
6.4 ASIC Implementation Results 61

7 Conclusions **63**
7.1 Achievements 63
7.2 Future Work 64

Bibliography **65**

A IOb-SPI Core Directory Tree **67**

List of Tables

2.1 Native Interface Bus Signals. 7

2.2 Memory map for USE_DDR=0, RUN_DDR=0 mode. 9

2.3 Memory map for USE_DDR=1, RUN_DDR=1 mode. 9

2.4 Board Flash Pins Assignments. 11

2.5 SPI Modes according to Clock Polarity and Clock Phase 13

2.6 Commands Characterisation Details. 16

4.1 IOb Wrapper native slave interface signals. 28

4.2 IOb Wrapper native slave instruction interface signals. 28

4.3 IOb Wrapper SPI interface. 29

4.4 Software Accessible Registers. 29

4.5 FL_COMMAND SW Register Configuration Fields. 29

4.6 FL_COMMANDTP SW Register Configuration Fields. 30

4.7 SPI Core Ports. 31

4.8 SCLK_GEN Module Ports. 32

4.9 ConfigDecoder Module Ports. 34

4.10 Latchspi Module Ports. 35

4.11 Command types Encoding Description. 36

4.12 SPI data lane modes Encoding. 36

4.13 Frame Structure Bit Fields. 37

4.14 Frame Structure Encoding Description. 37

4.15 DTR Mode Enable Encoding. 37

4.16 4-byte Address Mode Enable. 37

4.17 XIP Mode Enable Encoding. 38

5.1 Ext_flash Module Ports. 47

6.1 Performance Results. 57

6.2 Performance Results. 60

6.3 Xilinx FPGA Resource Utilization Results. 60

6.4 Intel FPGA Resource Utilization Results. 60

6.5 UMC Asic Resource Utilization Results. 61

List of Figures

2.1 IOb-SoC architecture diagram. 6
2.2 Native Interface Protocols 7
2.3 SPI Signals Master-Slave Connection. 12
2.4 QSPI transaction phases 15
2.5 Write Enable Transaction Waveform. 17
2.6 Reset Enable Transaction Waveform. 18
2.7 Read Status Register Waveform. 18
2.8 Page Program Transaction Waveform. 18
2.9 Fast Read Dual IO Transaction Waveform. 18
2.10 Fast Read Quad IO Transaction Waveform. 19
2.11 Traditional SPI Flash Controller Block Diagram. 20
2.12 CAST xSPI memory controller. 21

3.1 User flash QSPI interface. 24

4.1 SPI controller core IOb Wrapper. 27
4.2 SPI Core Block Diagram. 30
4.3 Global Core State Machine. 31

List of Acronyms

AHB Advanced High-performance Bus.

ASIC Application-Specific Integrated Circuit.

AXI Advanced eXtensible Interface.

BRAM Block RAM.

CPU Central Processing Unit.

DDR Double Data Rate.

DMA Direct Memory Access.

DTR Double Transfer Rate.

ECC Error Correcting Code.

EEPROM Electrically Erasable Programmable Read-Only Memory.

FFT Fast Fourier Transform.

FIFO First-In Fist-Out.

FPGA Field Programmable Gate Array.

Gb Gigabit.

HDL Hardware Description Language.

I2C DInter-Integrated Circuit.

Kb Kilobit.

LUT Look-Up Table.

Mb Megabit.

MOSFET Metal Oxide Semiconductor Field Effect Transistor.

OTP One Time Programmable.

PCB Printed Circuit Board.

PLL Phase-Locked Loop.

QSPI Quad Serial Peripheral Interface.

ROM Read-Only Memory.

RTL Register Transfer Level.

SoC Sistem-on-a-Chip.

SPI Serial Peripheral Interface.

SRAM Static Random-Access Memory.

SSD Solid-State Drive.

STR Single Transfer Rate.

UART Universal Asynchronous Receiver-Transmitter.

XIP eXecute-In-Place.

Chapter 1

Introduction

1.1 Topic Overview

In recent years, there have been many vital advances in open-source and free of charge tools for hardware and embedded software design.

One of the crucial developments in this area is the RISC-V ISA, which is an open Instruction Set Architecture (ISA), aimed at standardising a RISC instruction set while still delivering excellent performance in a diverse application space, ranging from embedded programs to supercomputer applications. A RISC instruction set has become a commodity and all related patents have expired; there is no point in investing money, time and resources in something that does not add value in itself. This way organisations and individuals can focus on creating new things that add value (software and specific hardware engines), without having to "pay a rent" for using a RISC-V CPU.

These developments allow for accelerated production of Intellectual Property (IP) cores and software, standardisation, System-on-Chip (SoC) integration and prototyping in general by a larger community of developers.

IObundle, a Lisbon-based system design company, has developed IOb-SoC, an open-source 32-bit SoC, which uses the PicoRV32 RISC-V CPU, with the goal to serve as a platform for further System-on-a-Chip developments such as accelerator IP cores and other functionalities.

The IOb-SoC hardware components are written in Verilog, and their software drivers are written in C/C++. Numerous auxiliary files and configuration files make up the project's development environment, which helps automate the simulation, synthesis and implementation flows. IOb-SoC supports a range of simulators, open-source (Icarus Verilog and Verilator) and commercial (Modelsim and Xcelium), FPGA boards (Cyclone V GT, Kintex UltraScale, Spartan-6 SP605 and others), and supports one ASIC technology (UMC 130 nm).

IOb-SoC features a simple Bootloader Program. A bootloader program is usually a small program that is the first program run by the computer system on boot. The bootloader is responsible for the initialisation of system components and loading and giving control to the main program, which can be a bare-metal firmware program or an Operating System program for more complex embedded applica-

tions. The program must be fetched from permanent storage (disk, flash, or network, for example) and loaded to temporary storage (RAM) for execution.

Flash memory is a widely used non-volatile memory solution that is cheap and offers comparatively low power consumption. These are great advantage points, especially for embedded system applications.

1.2 Motivation

Presently, the IOb-SoC system allows for the execution of a bootloader program loaded from internal ROM, which can receive a firmware program from its Universal Asynchronous Receive Transmit (UART) communications module, and store it in internal RAM or external memory such as DDR for execution.

Although the present IOb-SoC functionality helps verify and test applications, in the production of an embedded system, one often needs a non-volatile memory to permanently store the firmware and eventually some data.

The motivation of this work is thus to create an interface to an external non-volatile memory and modify the IOb-SoC bootloader program so it can also load and run firmware programs from this memory.

1.3 Objectives and Deliverables

The primary objective of the present work is to design and integrate flash memory controller into IOb-SoC's memory subsystem, and modify the bootloader program so that program instructions can be stored onto flash storage and executed directly from it.

As a second objective, the flash memory controller can also be accessed as an IOb-SoC peripheral to serve as an unspecified data storage medium for firmware programs. This way, programs can store data to be used later after a power cycle.

In order to successfully attain the above objectives, the following list of deliverables are expected:

- a flash controller core, which includes a peripheral interface and an instruction memory interface.
- an IOb-SoC instance, with the necessary modifications to integrate the flash memory controller, with access to the instruction and peripheral buses.
- an upgraded bootloader program, run programs from the flash, update the cache with new programs, or both.

This work is developed on the Xilinx Kintex UltraScale KU040 FPGA development board, which houses an SPI NOR-flash memory chip. Synthesis results for a low-cost Intel Cyclone V GT device are also presented. The work is easily extensible to other boards.

1.4 Thesis Outline

This document comprises seven chapters, as detailed below:

- Chapter 1 - Introduction
- Chapter 2 - Background: examine the relevant information for developing this work, including the IOb-SoC architecture, flash memory and flash memory controllers
- Chapter 3 - On-Board Flash Memory: present the on-board flash memory device main features
- Chapter 4 - Core Development: detail the implementation specifications for the controller core
- Chapter 5 - Bootloader and Core Integration to SoC: describe the core integration into IOb-SoC and the modified bootloader program
- Chapter 6 - Results: performance and implementation results are discussed and compared with a competitive solution
- Chapter 7 - Conclusions: summary, achievements and future work perspectives.

Chapter 2

Background

In this chapter, the relevant background information for the development of the SPI controller core is introduced. It is divided into four main sections, namely: IOb-SoC Architecture, Development Board Characterisation, SPI protocol details, Flash Memory and SPI Flash Memory Controllers.

2.1 IOb-SoC Architecture

IOb-SoC is a System on a Chip architecture based on the RISC-V CPU architecture. IOb-SoC serves as a platform for the development of complex applications by adding relevant IP cores and software. The hardware source components are written in the Verilog HDL, and the software source components are written in the C/C++ languages. IOb-SoC uses the RISC-V toolchain and many other free and open-source tools in a completely open and free of charge development environment. A general presentation of the various elements that are part of the IOb-SoC hardware/software platform and its development environment is given next.

2.1.1 Architecture Components

The IOb-SoC block diagram is shown in Figure 2.1 and illustrates the system's organisation. The base architecture of the system is formed by the following components:

- **RISC-V CPU**
- **Memory Subsystem**
- **Peripherals** (IP cores)
- **Interconnect, Instruction and Data Buses**

2.1.1.1 RISC-V CPU

IOb-SoC targets small embedded systems, using a small RISC-V CPU, which consumes reduced hardware resources. The processor implementation currently at use is the PicoRV32 [1] architecture, but others can also be used such as the DarkRV [2] implementation. The platform is CPU-agnostic.

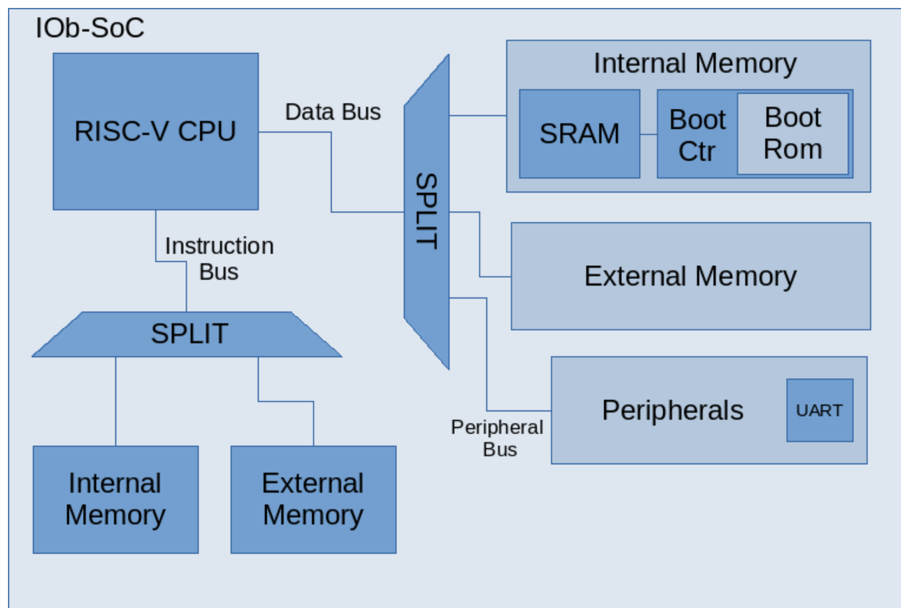


Figure 2.1: IOb-SoC architecture diagram.

2.1.1.2 Memory Subsystem

The memory subsystem consists of configurable SRAM, Boot ROM and optional DDR and cache memory integration. A boot controller system is coupled with the Boot ROM.

2.1.1.3 Peripheral Cores

Peripheral cores can be added to the system to implement additional functionalities. One of the important peripheral cores, used for basic system functions, is the UART core.

The UART communications peripheral is the only peripheral component included in the basic IOb-SoC configuration. The UART core is responsible for the communication with the host computer, sending and receiving runtime information such as commands, status data, firmware and other data.

2.1.1.4 Interconnect and Buses

The processor is connected to the memory components and other peripherals through the data and instruction buses. An interconnect sub-module implements the hardware interfaces used by the components. Although the interconnect component makes available standard hardware interfaces such as the AXI4 [3] bus, most interfaces use a native, memory-like and simpler hardware handshake protocol.

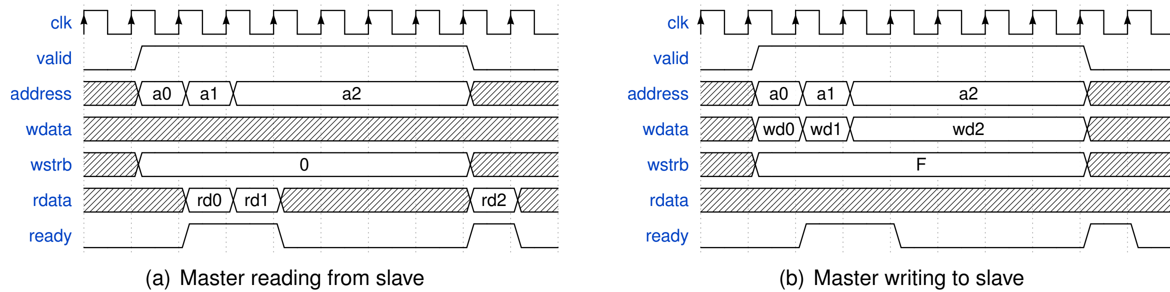
2.1.1.4.1 Native Interface

The Native Interface is the main bus protocol for interfacing with the IOb-SoC's processor bus. The signals which comprise the Native Interface protocol are summarised in the Table 2.1. The signal direction is represented with respect to the CPU.

Figure 2.2 illustrates the Native Interface signals and protocol. The left waveform describes a master core reading from a slave core, and the right waveform describes a master writing to a slave.

Table 2.1: Native Interface Bus Signals.

Signals	Direction	Width	Description
valid	output	1	Slave operation request trigger
address	output	32	Slave memory address requested for access
wdata	output	32	Data to write to slave
wstrb	output	4	Byte to write (0000 for read)
rdata	input	32	Slave response data
ready	input	1	Slave ready state



(a) Master reading from slave

(b) Master writing to slave

Figure 2.2: Native Interface Protocols

IOb-SoC makes use of Harvard architecture [4] where there are separate address spaces for program code and data. The CPU is connected to the other components via the instruction bus and the data bus. The data bus is further split into external data memory, internal data memory and peripheral data memory buses. The instruction bus can be split into external instructions memory and internal instructions memory buses.

2.1.2 File Structure

The IOb-SoC repository is hosted on Github [5]. The project files consist of source and header files, in the Verilog and C languages for hardware and software, respectively, makefiles and other scripts (Python, bash, tcl, etc). The project root directory is divided into the following subdirectories:

- **hardware**: contains the hardware components RTL description files, simulation testbench files, simulator specific files in directories with the simulator's name (for example, the icarus directory) and, board specific synthesis/implementation source and script files (ASIC and FPGA implementations) in the directories with the board's name (for example, the AES-KU040-DB-G directory).
- **software**: contains the bootloader and firmware C source files plus header files, helper python scripts, macros defining top header files and linker script files.
- **submodules**: gathers complementary repositories for important components and peripheral cores, for instance, the CPU or cache sub-modules.
- **document**: contains documentation about IOb-SoC core and components.

Other important files are:

- **system.mk**: found at the root directory, this makefile segment contains user defined variables/-macros for the makefiles that include this segment. These variables define the overall operation modes of the system and influence the synthesis/compilation processes.

- **Makefiles**: the project adopts a recursive makefile tree structure for automating the compilation/synthesis processes. There are Makefiles distributed in many of the project directories. For a component compilation process, its respective makefile is called (sub-makefile). The sub-makefile defines the dependencies required uniquely for this specific component, and includes a makefile segment in some ancestor directory. The makefile segment defines the dependencies which are common to the children directories. At the root directory, there is the main Makefile, which presents the overarching system tools interface, triggering the components' compilation processes.

2.1.3 Operating Modes

The IOb-SoC general operation flow can be described through the following steps:

1. After power-up, the Boot Controller hardware component loads the bootloader code from the internal Boot ROM to the internal SRAM memory space while keeping the CPU in reset state.

2. After the bootloader is copied to the SRAM, the CPU starts running the bootloader code from the SRAM.

3. The bootloader program running on the prototyping board connects to a console application running on the host computer by means of the UART interface. The console application is able to serve requests from the bootloader program to send or receive data and binary files. For example, if the INIT_MEM macro is reset, the bootloader asks the host to send the firmware binary file and loads it onto the memory (SRAM or DDR); else, if INIT_MEM=1, the firmware is already pre-initialised into the memory.

4. When the bootloader program finishes executing, it restarts the system to run the firmware from the 0x00000000 initial address.

5. The connection to the host console is terminated when the firmware finishes running.

2.1.3.1 Memory Operating Modes

IOb-SoC features several operating modes depending on the memory resources put to use when running the firmware [5]. IOb-SoC can make use of internal SRAM or external DDR memory for running code or accessing data. The operation modes are established by the variables USE_DDR, RUN_DDR and INIT_MEM, present in the root configuration file (system.mk). The project establishes two memory classifications: main memory and extra memory. Main memory is the memory resource from which the code is executed, while an extra memory is only accessed for data starting from a certain address. The system determines the following memory operating modes:

- USE_DDR=0 RUN_DDR=0: the internal SRAM memory is the main system memory used both as the data and instruction memory.
- USE_DDR=1 RUN_DDR=0: the internal SRAM is used as the main memory but the DDR is also available as an extra data memory space.
- USE_DDR=1 RUN_DDR=1: the DDR is the main memory for data and instructions. The SRAM stands as an extra usable memory.

2.1.4 Memory Map

IOb-SoC uses a 32-bit address space. Three main derived variables are used to determine and derive the components memory mappings, depending on the operating mode. They are: **E** (for Extra Bit), **P** (for Peripherals) and **B** (for Bootloader). These variables are found in the system.mk file. Next, the memory maps for two setup configurations are presented:

- USE_DDR=0, RUN_DDR=0: In this mode, the SRAM is used as the main memory and there is no extra memory source. For this configuration, the above mentioned variables are derived as E=31, P=31 and B=30. These configuration values result in a system memory map illustrated in Table 2.2. UART, SPI and TIMER represent peripheral submodules added to the system.

Table 2.2: Memory map for USE_DDR=0, RUN_DDR=0 mode.

Memory Component	Address Space
Main Memory (SRAM)	0x00000000 - 0x3FFFFFFF
BOOTCTR	0x40000000 - 0x7FFFFFFF
UART	0x80000000 - 0x9FFFFFFF
SPI	0xA0000000 - 0xBFFFFFFF
TIMER	0xC0000000 - 0xFFFFFFFF

- USE_DDR=1, RUN_DDR=1: In this mode, the DDR memory is used as the main memory and the SRAM as an extra memory. The configuration variables are set as E=31, P=30 and B=29. The resulting memory map is shown in Table 2.3.

Table 2.3: Memory map for USE_DDR=1, RUN_DDR=1 mode.

Memory Component	Address Space
Main Memory (DDR)	0x00000000 - 0x1FFFFFFF
BOOTCTR	0x20000000 - 0x3FFFFFFF
UART	0x40000000 - 0x4FFFFFFF
SPI	0x50000000 - 0x5FFFFFFF
TIMER	0x60000000 - 0x7FFFFFFF
Extra Memory (SRAM)	0x80000000 - 0xFFFFFFFF

2.1.5 Development Flow

When developing a new core for IOb-SoC, the system provides facilitated ways and automated processes to integrate and test the new core into the system architecture. To test the new functionalities provided by a core, the core's independent development repository is added as an IOb-SoC git submodule repository, placed in the submodules directory. Then, the core must be added to the IOb-SoC peripherals list in the system.mk file. By providing certain auxiliary files in the core directories, IOb-SoC can automatically integrate the core into the system for simulation and FPGA implementation purposes.

System description source files incorporating the core instances are automatically produced. At the current version, IOb-SoC integrates several simulation tools, namely: **icarus** [6], **ncsim**, **verilator** and **modelsim**.

2.2 Development Board

IOb-SoC can be implemented into FPGA chips and ASICs. At the current IOb-SoC codebase version [5], the following development boards are supported: **Kintex UltraScale KU040** Development Board [7], **Basys 3 Artix-7** board, **Cyclone V GT FPGA** development board, Terasic **DE10-Lite** development board, **Spartan-6 SP605 Evaluation Kit**; plus, the **UMC 130** nm ASIC. The present work was developed on the **Kintex UltraScale KU040** board, hereafter referred as the KU040 board, the target FPGA prototyping board.

2.2.1 Xilinx KU040 board features

The KU040 board houses a Xilinx XCKU040-1FBVA676 -1 speed grade FPGA device. The pins that are connected to the FPGA device are grouped into pin banks (6 I/O banks and 4 GTH banks). The board presents several power, programming, clocking, interfaces and memory resources [7], which can be utilized according to each project's needs. Particularly, the board presents the following memory resources:

- 1GB DDR4 SDRAM
- 32MB of Configuration QSPI Flash
- 32MB of User Code/Data QSPI Flash

2.2.1.1 User Code/Data QSPI Flash

The board can house two different flash memory devices, depending on availability: the **Micron N25Q256** or the **Cypress S25FL256SAGMFIR0**. Unlike the Configuration flash, which is used for the FPGA configuration bitstream, the User Code/Data flash can be used for storing used code and data in a persistent manner. The flash device is interfaced through the QSPI protocol, which is an evolution of the SPI protocol. QSPI uses 4 bidirectional data lanes, in contrast with the 2 unidirectional data lanes

used by SPI. Both use the traditional clock and chip select control signals. The following table exhibits the User Code/Data flash device pins connections:

Table 2.4: Board Flash Pins Assignments.

User QSPI interface	FPGA Bank	FPGA Pin
QSPI_2_DQ3	66	H12
QSPI_2_DQ2	66	J11
QSPI_2_DQ1	66	H11
QSPI_2_DQ0	66	G11
QSPI_2_CS	66	D19
QSPI_2_CLK	66	F10

2.3 Flash memory

Flash memory is a non-volatile memory technology based on EEPROM, which is both electrically erasable and reprogrammable. It is usually significantly cheaper than EEPROM, and is preferred when many rewrites are expected. The technology was initially developed at Toshiba in 1980. Since then, it has become a ubiquitous storage medium, found on a wide range of applications. The flash memory cell design is based on floating gate MOSFETs. The memory cells interconnection scheme defines two dominating types of flash memory: NAND flash and NOR flash.

2.3.1 NOR vs NAND Flash

NAND flash is arranged in blocks. Write, read and erase operations are also performed in blocks. While NAND flash provides whole memory page access (block access), NOR flash is able to provide random memory access for individual bytes. Due to the memory cells configuration, NAND flash cells occupy approximately 40% less silicon area than the equivalent NOR flash cell for a similar process technology. NAND flash is cheaper.

NOR flash provides for faster read access and has higher storage reliability and retention. NAND flash [8] provides much higher memory densities but requires ECC management to ensure better reliability. NAND flash usually presents a number of bad memory blocks which must be managed.

These characteristics help define more appropriate application areas for each of the flash memory types. NOR flash are more appropriate for systems where faster random access is required, and are commonly used for code storage and execution. This type of flash memory is commonly found on microcontroller boards in embedded applications. NAND flash is more commonly used in file storage applications such as USB flash drives, smartphones or solid-state drives (SSDs).

A severe limitation of flash memory is the limited number of program and erase cycles of memory blocks; the memory block becomes unreliable after this number is exceeded. Typically, for NOR flash devices, the maximum number of program/erase cycles is around 100.000. To circumvent this limitation and extend a flash device lifespan, wear-levelling mechanisms are employed. These mechanisms track the memory blocks usage level and relocate data to less used or unused memory block addresses [9].

2.3.2 NOR flash interfaces

Typical NOR flash memory densities range from 512 Kb to 2Gb. There are parallel and serial NOR flash memory interface solutions provided. Parallel flash interfaces tend to provide higher access performance but have much higher pin-counts. Serial NOR flash interfaces have much lower pin-counts, and therefore smaller PCB footprints, and they tend to present comparatively lower access performance. Traditionally, the interface used for serial flash is SPI. An SPI serial NOR flash device typically has four required signal pins, though for recent extended SPI protocols, it is common to find devices with ten or eleven pins [10] [11]; a parallel flash device may present 40+ pins [12].

2.4 SPI communications protocol

Numerous hardware protocols are employed for communication between electronic devices, each with a different set of advantages and constraints. Among the most widely used are the following: I2C (Inter-Integrated Circuit), UART (Universal Synchronous Receiver/Transmitter) and SPI (Serial Peripheral Interface).

SPI is a serial communication protocol widely utilized in several applications, including memory chip interfaces, microcontroller interfaces, Analog to Digital Converters and various sensor device interfaces. It was originally specified by Motorola [13].

2.4.1 SPI signals description

Figure 2.3 illustrates a master and multi slave configuration and the involved signals.

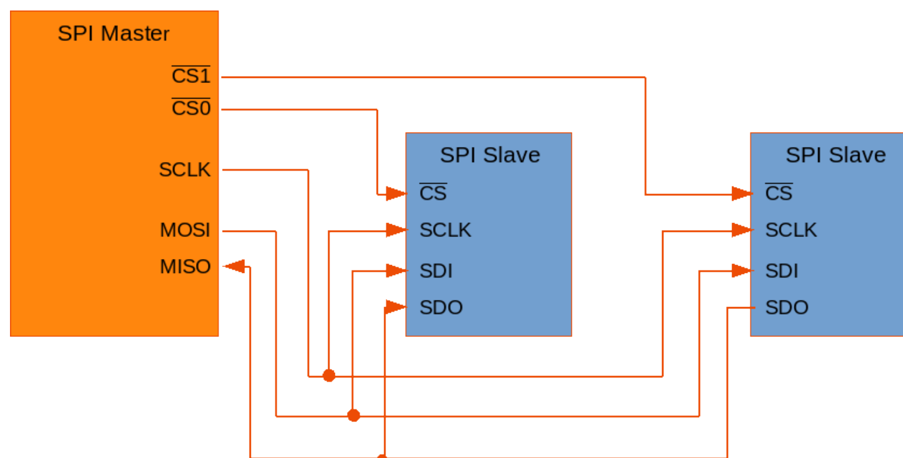


Figure 2.3: SPI Signals Master-Slave Connection.

The SPI protocol communications are based on the signals listed next:

- Serial Clock (SCLK)
- Chip Select / Slave Select (CSn/SSn)
- Master Out, Slave In (MOSI); or Slave Data In (SDI)

- Master In, Slave Out (MISO); or Slave Data Out (SDO)

2.4.2 SPI Transaction

An SPI transaction is performed as described next: The master generates the clock signal (SCLK). The transmitted and received data between the master and a slave is synchronized to this clock.

A chip select or Slave Select (SS) signal selects the corresponding target communication slave device. The SPI protocol allows for a multiple slave configuration, each with its independent SS signal.

The master then transmits data through the MOSI data line, and the slave device receives it through the SDI data line.

The master device receives data through the MISO data line from the slave. The slave device transmits data through the SDO data line.

The communication transaction is started by generating the clock signal by the master and asserting (normally low) the chip select signal of the target slave. Data on the data lines can be simultaneously transmitted and received by the master and the slave devices (full-duplex).

2.4.3 SPI Clocking modes

As the data is transacted (shifted-out and sampled) synchronously to the clock, the SPI protocol can define four operation modes according to different clock polarity (CPOL) and clock phase (CPHA) configurations. Table 2.5 outlines the possible CPOL and CPHA configurations.

Table 2.5: SPI Modes according to Clock Polarity and Clock Phase

CPOL	CPHA	SPI Mode	Description
0	0	0	Clock idle at logic low. Data latched out at falling edge and sampled at rising edge
0	1	1	Clock idle at logic low. Data latched out at rising edge and sampled at falling edge
1	0	2	Clock idle at logic high. Data latched out at rising edge and sampled at falling edge
1	1	3	Clock idle at logic high. Data latched out at falling edge and sampled at rising edge

2.4.4 xSPI standard

Due to access performance limitations for the traditional SPI protocol, newer SPI-based protocols were developed. Usually the new protocols add additional data pins for increased transmission bandwidth in a single serial clock cycle. The eXpanded Serial Protocol Interface (xSPI) has eight data lines, and is a JEDEC [14] standard, a body that specifies a series of protocols for communication between devices. The xSPI standard specifies a low pin-count interface with multiple data lanes for high data transfer bandwidths, bypassing the traditional SPI transfer bandwidth limitations. The standard is defined in the JEDEC JESD251A document [15].

The standard specifies commands for read and write operations for communication with compliant peripherals, and commands for specific functions for non-volatile memory devices. It also includes limited backward compatibility with SPI master controllers. It defines commands involving 1-bit wide and 8-bit wide data signals. The standard establishes the following signals: serial clock, chip select, a maximum of 8 data signals, and data strobe signal for timing reference. HyperBus [10] and Xccela [11] are some of the proprietary standards compliant with xSPI.

2.4.5 Quad SPI

QSPI (Quad SPI) is a single master multiple slave interface protocol that extends and has backward compatibility with the traditional SPI standard. It uses a subset of signals and commands for read and write operations from the xSPI standard and follows the xSPI standard transaction frame format.

2.4.5.1 Signals description

QSPI is specified based on the following signals:

- Serial Clock (Sclk)
- Chip Select (CSn)
- DQ0, DQ1, DQ2 and DQ3

The DQ0-3 signals are bidirectional data lines that can function as both inputs and outputs. Usually, flash chips that support QSPI also support the simpler SPI or Dual SPI protocols. On the simpler SPI protocol, one data line is used as an input to the flash chip (DQ0) and another as an output (DQ1), while on the Dual SPI protocol both data lines are used as inputs and outputs, thus increasing the data throughput capacity.

Some data lines may carry other functionalities offered commonly by many vendors. These functionalities include: a Word Protect (WPn) signal (usually on DQ2), a Reset (Rstn) signal or a Hold (Holdn) signal (DQ3).

Some manufacturers that offer NOR flash chip products that support QSPI, or, recently, the newer 8-bit SPI evolution protocols (Octal SPI), include: Micron Technology, Cypress Semiconductor, Macronix International, Winbound Electronics and GigaDevice.

2.4.6 Serial Flash Operation Modes

A communication transaction is started by the master device by activating the CSn (LOW) signal connected to the corresponding target slave device. The signal is held active while the transaction is taking place, and it is deactivated by the master at the end of the transaction or to interrupt it.

While the CSn signal is active, the serial clock signal is kept toggling during the transaction, taking into account the flash chip's supported SPI modes defined by CPOL and CPHA. It is not required that the clock signal toggles when the slave device is not selected (CSn HIGH).

The data is transmitted and sampled by the involved devices at the positive or negative edges of the clock, depending on the SPI mode in use. Usually, NOR flash chips support the SPI modes (0,0) and (1,1), where the first number refers to the clock polarity and the other to the clock phase. For instance, on mode (1,1), the first data bits are sent by the master on the first negative edge of the serial clock and sampled by the slave device on the following positive clock edge. The data is transmitted in 1-bit, 2-bit or 4-bit widths, depending on the current mode (simple, Dual or Quad), or the operation opcode.

Along with the normal SDR (Single Data Rate) mode, many flash chips offer support for DDR (Double Data Rate) mode. In DDR mode, the data is sent twice a clock cycle resulting in double the throughput. In DDR SPI mode (1,1) for instance, the data is sampled by the devices on both the negative serial clock edge and again on the following positive clock edge, which unfavourably results in smaller time windows for successfully reading the data.

2.4.7 QSPI Transaction Format

A QSPI transaction normally involves a number of the following phases (frame segments):

- Command Phase
- Address Phase
- Latency Phase
- Data Phase

Figure 2.4 depicts the sequence of transaction phases for a general read operation or program transaction. The displayed values refer to typical values for the code flash chip family [16] on the KU040 prototyping board.

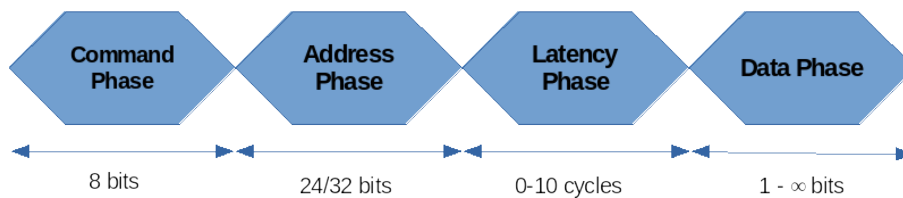


Figure 2.4: QSPI transaction phases

2.4.7.1 Command Phase

For QSPI transactions, the command phase is normally the first phase. It is usually represented by an 8-bit opcode. The opcode determines the following transaction phases, with some transaction phases omitted and others required. The command opcode can be transmitted at one bit, two bits or four bits per cycle, depending on the operating mode (simple, dual or quad), requiring 8, 4 or 2 cycles, respectively, for a complete transfer.

Many serial NOR flash devices support a special mode where the command phase is not required for memory read operations, thus saving a few clock cycles. This mode is commonly designated as the

XIP (eXecute-In-Place) mode. Commonly, transactions requiring command, address, latency and a data phases are memory read operations. Other operations such as reading or writing to registers may only require a command and a data phase.

2.4.7.2 Address Phase

This phase segment specifies the address in the flash memory requested for access. Current common flash chips allow for 24-bit or 32-bit addresses for larger memory capacities. Some commands do not require this phase. Examples of operations requiring this phase are memory read and memory program operations.

2.4.7.3 Latency Phase

This phase executes a number of required wait (dummy) cycles before the flash device is able to correctly output requested data. Some commands don't require this phase and the length of this phase has a default value that may be adjustable depending on the operating serial clock frequency on some flash devices.
On this phase, the data lines may be left unasserted, but can be asserted to relay certain information about the following transactions.

2.4.7.4 Data Phase

In this phase, the data to be stored on flash is transmitted from the master and sampled by the slave device on write operations, while on memory read requests the data is shifted out by the slave device to the master device. The flash memory device characteristics and configuration determine the maximum number of bits that may be read or written in a single transaction while the CSn signal is active.

The flash memory device datasheet should be referred to, for the complete characterisation of the transactions [16].

Table 2.6 lists the basic commands supported by SPI NOR flash devices.

Table 2.6: Commands Characterisation Details.

Function	Command Code	Address Bytes	Latency Cycles	Data Bytes
Read SFDP	5A	3	8	1 to ∞
Read	02	3/4	0	1 to ∞
Write Enable	06	0	0	0
Page Program	02	3/4	0	1 to 256
Read Status Register	05	0	0	1 to ∞
Fast Read	0B	3/4	8	1 to ∞
Enter Deep Power Down	B9	0	0	0
Exit Deep Power Down	AB	0	0	0
Reset Enable	66	0	0	0
Reset Mem	99	0	0	0

For Table 2.6, the following should be noted:

- The reference values are relative to the flash device model (N25Q256A) available on the KU040 board. The command nomenclature is the one used in the datasheet [16]. The nomenclature is consonant to the nomenclature used in the standard [15].
- The number of latency cycles are the default values and may be adjustable for higher or lower values (depending on the operation frequency).
- Some commands require that the Write Enable command be issued beforehand, for instance, the Program and Erase commands.
- The Read command (02h) does not require latency cycles and, as a consequence, has limited operation frequency.
- The 256 bytes limit for the Page Program command refers to the flash memory page size limit [16]. Only one page can be programmed at a time by a single command.

2.4.8 Command Transaction Waveforms

Below, a series of waveforms for some of the commands and the data lane usage modes introduced above is presented. The adopted terminology for some commands is the one presented in the datasheet in [16]. The clocking scheme is the SPI mode (1,1). The referred modes follow a (X-X-X) framing where X indicates the number of data lanes used during a particular phase. For multi data lanes transfers the most significant bit is sent on the DQ1 or DQ3 data lanes for dual or quad modes, respectively.

2.4.8.1 Write Enable on mode (1-1-1)

The Write Enable command (06h) is transmitted in a single data line. The command is often required before issuing a program or erase command as a memory protection mechanism. The waveform in Figure 2.5 displays the behaviour of the involved signals for the selected transmission mode.

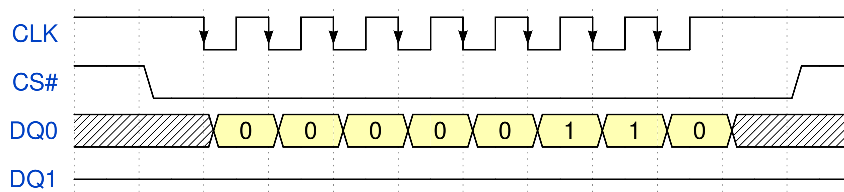


Figure 2.5: Write Enable Transaction Waveform.

2.4.8.2 Reset Enable on mode (2-2-2)

The Reset Enable command (66h) is often required before issuing a Reset Memory command. The figure 2.6 depicts its waveform.

2.4.8.3 Read Status Register mode (4-4-4)

The Read Status Register command (05h) in quad mode, where both the command and data phases utilise four data lanes, is illustrated in Figure 2.7.

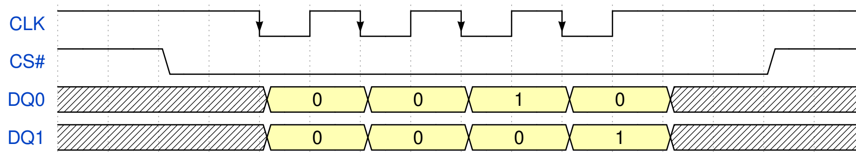


Figure 2.6: Reset Enable Transaction Waveform.

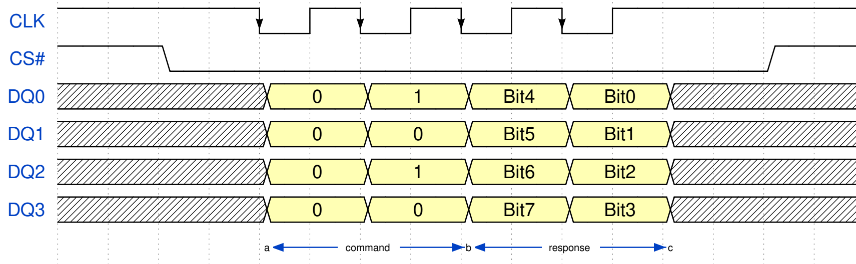


Figure 2.7: Read Status Register Waveform.

2.4.8.4 Page Program mode (1-1-1)

The Page Program (02h) command in simple mode is illustrated in Figure 2.8, which shows the signals waveform for a 32 data bits phase and a variable number of address bits where A[max : min] can refer to a 3-byte or 4-byte address.

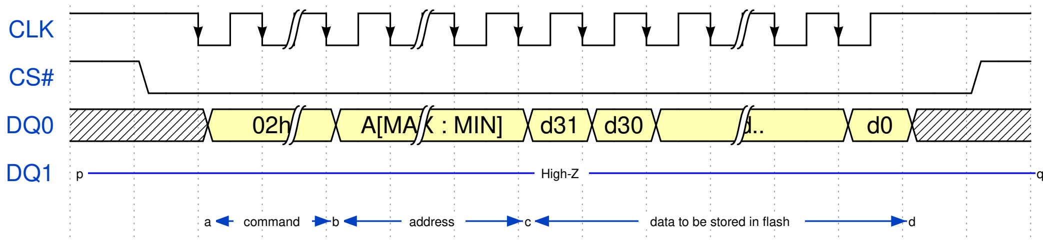


Figure 2.8: Page Program Transaction Waveform.

2.4.8.5 Fast Read (1-2-2)

Fast Read commands allow for greater serial clock speeds but require latency cycles. The Fast Read command (BBh) in mode (1-2-2) operates in simple mode but admits the address phase and data phase to be transmitted in dual mode. The figure 2.9 presents the transaction waveform and includes a mode bit (0) in the latency phase named XIP confirmation bit.

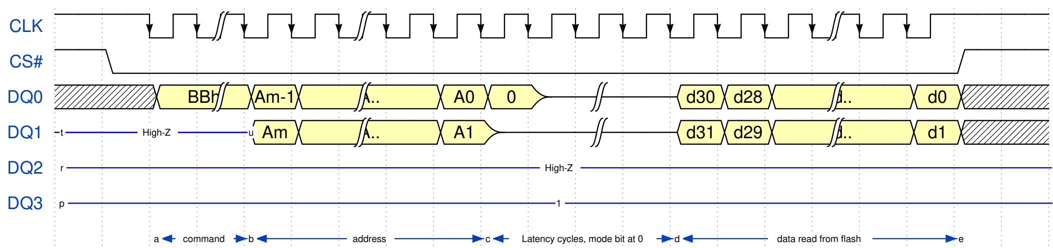


Figure 2.9: Fast Read Dual IO Transaction Waveform.

2.4.8.6 Fast Read (1-4-4)

The Fast Read command in mode (1-4-4), with command opcode EBh, admits an address phase and data phase in quad mode. The waveform in Figure 2.10 also includes a XIP confirmation bit set to 1.

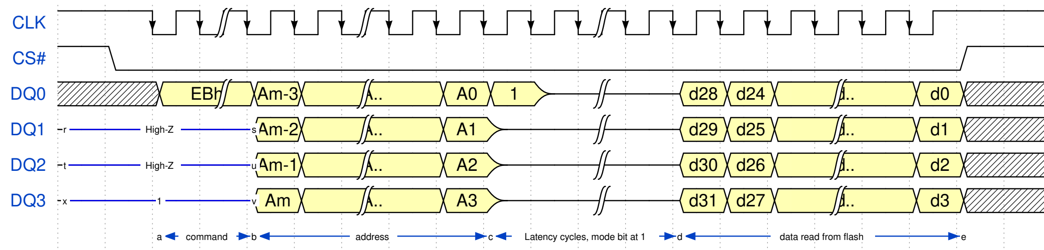


Figure 2.10: Fast Read Quad IO Transaction Waveform.

2.4.9 Flash Device Functionalities (SFDP)

One important JEDEC standard for flash vendors is the JEDEC Serial Flash Discoverable Parameters standard, defined in [17]. This standard establishes a method for organising and presenting flash device functionalities and features, and also establishes the access protocol for the host system controller. The standard defines the SFDP Database, a set of headers and tables stored in the flash device memory.

Below, the description of some of the SFDP Basic Flash Parameter Table fields is presented:

- **1st WORD**: Its value describes the following flash device features: Write Buffer Size, DTR support, uniform 4KB block erase, number of address bytes, fast read support for different data lanes use modes, etc.
- **2nd WORD**: Flash device memory density
- **3rd WORD**: Fast Read on modes (1-4-4) and (1-1-4) characterisation (number of latency cycles, mode bits, command code)
- **4th WORD**: Fast Read on modes (1-1-2) and (1-2-2) characterisation
- **5th WORD**: Fast Read on modes (2-2-2) and (4-4-4) support
- **6th WORD**: Fast Read on mode (2-2-2) characterisation
- **7th WORD**: Fast Read on mode (4-4-4) characterisation
- **8th WORD**: Erase Type 1 and 2 (block size and respective command code)

The JESD216D document [17] uses the (n1 - n2 - n3) terminology for indicating the number of data lanes used for each transaction phase segment, where n1 refers to the number of data lanes used for the command phase, n2 to the address phase and n3 to the data phase (1, 2, 4 or 8 data lanes possible for the newer flash devices). For the complete SFDP standard description, the JESD216D document [17]

should be referred. The SFDP standard compliant Parameter Table for the KU040 on-board flash device can be checked in [16]. The SFDP Headers and Tables information is accessed by the Read SFDP command described earlier.

2.5 SPI Flash Memory Controllers

An SPI Flash Memory Controller is a device which serves as an interface between the CPU and a flash memory device. It normally provides a slave interface implementation to communicate to the CPU, and additionally an SPI master interface to communicate with the flash memory device. It allows for the CPU to read from and write data to the flash device. Traditionally, SPI flash memory controllers comprise the logical blocks shown in Figure 2.11 below:

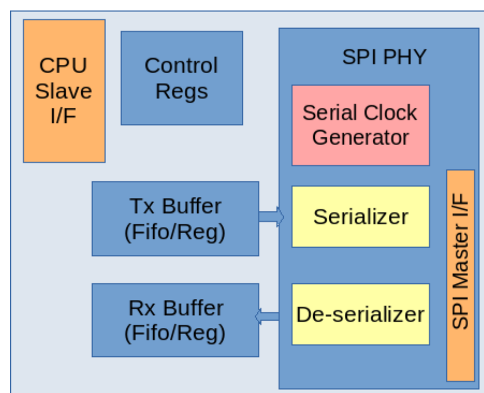


Figure 2.11: Traditional SPI Flash Controller Block Diagram.

Traditional SPI flash memory controllers have a minimum of two interfaces: the CPU interface and a master SPI interface connected to the flash device. The controller normally hosts a transmit (TX) and receive (RX) buffer which can be based on registers or FIFOs. The TX buffer stores the transmission bit sequence and the RX buffer stores the response data bits from the flash memory device. The SPI Phy is responsible for serially shifting out the TX bits and for sampling the response bits and storing them in the RX buffer.

2.5.1 Comparative Flash Controller Model

The xSPI-MC flash memory controller [18] from CAST is a flash memory controller that can be taken as a reference implementation. Figure 2.12 below illustrates the controller's block diagram (taken from [18]).

This memory controller supports several features, in particular the following, as specified in its product brief [18]:

- Support and compatibility for a number of proprietary SPI protocols and standards (in particular xSPI and Xccela).
- Multiple data lanes modes support (single, dual, quad, twin-quad and Octal).

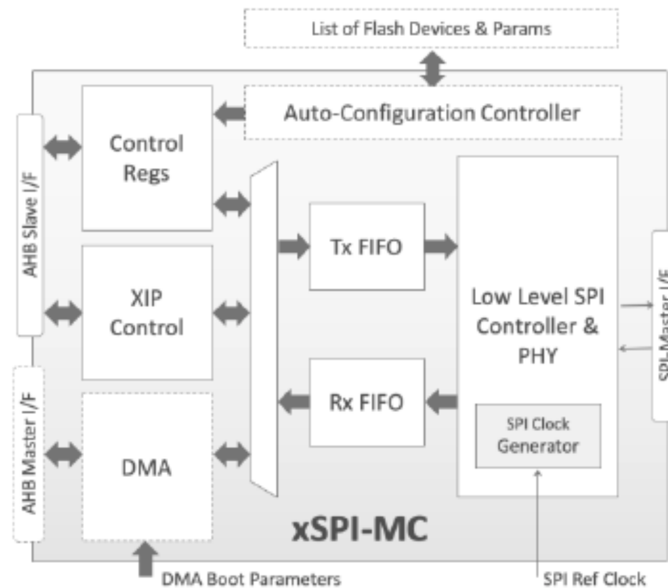


Figure 2.12: CAST xSPI memory controller.

- Single Data Rate (STR) and Dual Data Rate (DTR) modes support
- XIP (eXecute-In-Place) support
- DMA usage support
- AHB slave interface for communication with CPU; AHB master interface for DMA
- Auto-configuration for particular flash device capability
- Configurable reset values for configuration registers and configurable AHB interface bus widths.

The complete features description can referred at [18] datasheet. This implementation reports the following FPGA resources usage and performance metrics:

- Implementation device : Xilinx Kintex-7 7k420-ffg1156-2; Area : 1200 LUTs; Clock frequency (MHz): AHB:250, SPI:100. Implementation Configuration: XIP on, DMA off, auto-configuration off.

The CAST flash controller core hosts a set features commonly found on several commercially available flash controller IPs. Other flash controller IP designers include Digital Blocks, Xilinx, and numerous others.

Chapter 3

On-Board Flash Memory

This chapter introduces the main features of the flash device present on the prototyping FPGA board. The flash device is a flash memory device from Micron and the complete characterisation information can be found in [16].

3.1 N25Q256A Features Introduction

The flash device on the FPGA prototyping board is a flash memory device of the N25Q256A family.

The flash device has a density of 256 megabits, 1.8V supply voltage, multiple IO, 4 KB, Sector Erase, Serial NOR flash memory, and supports the following set of features:

- SPI-compatible serial bus interface
- Multiple IO data lanes
- Simple SPI (designated Extended mode), Dual and Quad modes protocols
- DTR mode support
- XIP mode support
- Configurable latency cycles
- Burst read support (continuous read)
- 3-byte/4-byte addresses support
- 4 KB sub-sector, 64 KB and full chip Sector Erase capability
- Write Protection functionalities
- OTP memory storage

The flash memory interface can be seen in Figure 3.1, taken from [7].

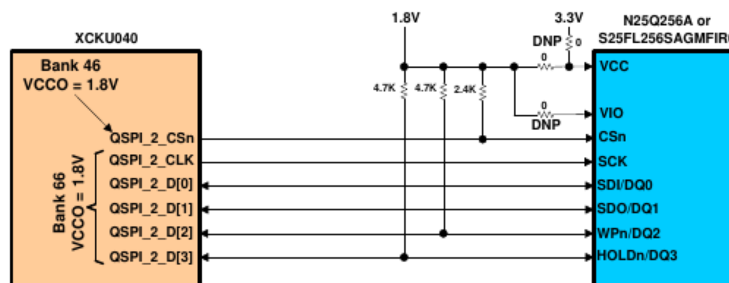


Figure 3.1: User flash QSPI interface.

3.2 Memory Configuration

The memory is composed of 33,554,432 bytes, 512 sectors (64 KB), 8192 sub-sectors (4KB each), and 131,072 pages (256 bytes each) and an additional 64 bytes of OTP memory.

The memory is divided into 2 memory segments: the upper memory segment (16 MB) and the lower memory segment (16 MB). The lower memory segment is accessible by the initial 24-bit (3 bytes) addresses, while accessing the upper memory requires an extra address bit (4-byte address mode).

Bits are programmed from the initial state 1 to 0, and are erased setting them back from 0 to 1.

The flash memory implements numerous write protection mechanisms including the write enable command, the write-protect hardware signal and the sector blocking configuration mechanism.

When reading from and writing to the flash memory array, the corresponding accessed byte addresses are incremented from the input base address. For instance, when trying to read more than one byte of information from the flash memory, while asserting the address segment to 0x100, the response bytes will be from the flash byte memory location 0x100, followed by 0x101, 0x102 and so on, contiguously, with a byte's most significant bit being shifted-out first.

3.3 Status and Configuration Registers

The flash memory device comprises several registers, namely:

- Status Register
- Flag Status Register
- Nonvolatile Configuration Register
- Volatile Configuration Registers
- Enhanced Volatile Configuration Register
- Extended Address Register
- Internal Configuration Register

The device behaviour parameters depend on the Internal Configuration Register. The Internal Register configuration is downloaded from the Nonvolatile Configuration Register or from the Volatile Configuration Register or Enhanced Volatile Configuration Register.

The flash memory device initial state from factory is the following: all memory array bits are set to 1 (FFh bytes), the status register is set to 00h, and the nonvolatile configuration register has all bits set to 1 (FFFFh).

Chapter 4

Core Development

In this chapter, the SPI flash controller core development is described, referring details about the core's components and the development environment. It comprises five main sections: core functionality features, files directory structure, top module, central core module and, lastly, software driver. The project files repository is hosted on Github at `https://github.com/IObundle/iob-spi`.

4.1 Core functionality features

The developed core provides the following features:

- Single, Dual, Quad SPI protocol modes.
- XIP mode support.
- DTR mode support.
- 3 bytes and 4 bytes addressing modes support.
- Facilitated transaction frame encoding through predefined frame types
- CPU Peripheral bus interface and CPU Instruction bus interface.
- Zero Overhead Read Operations interface (Instruction Bus interface).
- Configurable transaction frame parameters: command code, number of dummy cycles, number of data bytes/bits.
- Synthesis configurable parameters: serial clock SPI mode (polarity and phase), serial clock frequency ratio, data bus widths.
- Synchronous clock design (one clock domain).

4.2 Files Directory Structure

The core's directory file structure presents a clear separation of concerns for the components description modules. The core's files and subdirectories structure is presented in Appendix A.

4.3 Top Module

The core's top module is a wrapper module that integrates the SPI master controller components and the required interfaces for the CPU (software accessible registers). The top module is designated IOb-spimaster.

The IOb-spimaster module presents three different interfaces: two for communicating with the CPU, and one multi data lane SPI interface for connecting to the SPI flash memory device. The block diagram is shown in Figure 4.1:

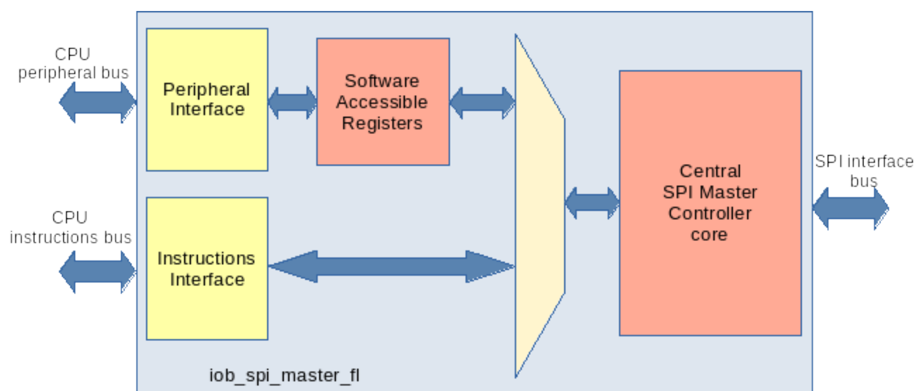


Figure 4.1: SPI controller core IOb Wrapper.

4.3.1 Blocks Description

The peripheral interface and the instructions interface implement the CPU's native slave interface protocol. The peripheral bus interface is utilized by the CPU for general read and write memory operations, while the instructions interface is designed for the purpose of zero overhead instruction code fetching. The Software (SW) Accessible Registers Block features a set of registers used to configure the core behaviour. When accessing the core through the instructions interface, the central core behaviour is defined by a fixed SW accessible registers pre-configuration. The component blocks are described next.

4.3.1.1 Peripheral Interface Description of Signals

The peripheral interface is connected to the CPU's peripherals bus, and is used for atomic transactions between the CPU and the flash controller. It incorporates the set of signals exhibited in Table 4.1.

Table 4.1: IOb Wrapper native slave interface signals.

Name	Type	Size	Description
valid	Input	1	Triggers the module to register wdata input to SW registers
address	Input	4 (ADDR_W)	Selects the target SW register
wdata	Input	32 (WDATA_W)	Carries the data from CPU to be registered
wstrb	Input	4 (DATA_W/8)	Indicates the operation direction: 1111 for writes, 0000 for reads from the SW registers
rdata	Output	32 (DATA_W)	Holds the response data to be read by the CPU
ready	Output	1	Indicates if the SW registers have accepted new data after a valid input pulse

4.3.1.2 Instructions Interface Description of Signals

This interface is connected to the CPU's instructions bus. It provides capabilities for successive memory read transactions. These transactions use a fixed configuration set through the SW accessible registers in the bootloader program. This interface is optional and depends on setting the RUN_FLASH variable to 1. It incorporates the set of signals indicated in Table 4.2 :

Table 4.2: IOb Wrapper native slave instruction interface signals.

Name	Type	Size	Description
valid_cache	Input	1	Triggers a flash memory read request
address_cache	Input	24 (FLASH_CACHE_ADDR_W)	Address for the flash memory location requested for access
wdata_cache	Input	32 (WDATA_W)	Not in use. Only reads from flash memory expected
wstrb_cache	Input	4 (DATA_W/8)	Only read operations expected (0000 binary value)
rdata_cache	Output	32 (DATA_W)	Outputs data from requested flash memory address
ready_cache	Output	1	Pulses for 1 clock cycle to indicate response read data available after a valid_cache pulse

This interface can be connected to a cache system to provide improved access performance for the CPU's instruction bus.

4.3.1.3 SPI master interface

This interface is used to connect the core to the flash memory device. It comprises the signals described in Table 4.3.

4.3.1.4 SW Accessible Registers

The SW accessible registers are accessed through the peripheral interface and are used to hold configuration data set by the CPU, and to hold data read from the flash memory array coming from the central controller. These registers also store values representing the current configuration state of

Table 4.3: IOb Wrapper SPI interface.

Name	Type	Size	Description
SS	Output	1	Connects to the CSn flash device pin
SCLK	Output	1	Connects to the serial clock pin
MOSI	Bidir	1	Connects to the DQ0 pin
MISO	Bidir	1	Connects to the DQ1 pin
WP_N	Bidir	1	Connects to the DQ2 pin
HOLD_N	Bidir	1	Connects to the DQ3 pin

the controller. The configuration data is used to modify the controller's behaviour. The SW accessible registers are comprised by the registers described in Table 4.4.

Table 4.4: Software Accessible Registers.

Name	Register Type	Size	Description
FL_RESET	Write	1	Set to 1 for a controller reset
FL_DATAIN	Write	32	Carries the data to be written to the flash device
FL_ADDRESS	Write	32	Holds the flash memory address requested for access
FL_COMMAND	Write	32	Holds the command and other transaction configurations
FL_COMMANDTP	Write	32	Holds the command type and other transaction configurations
FL_VALIDFLG	Write	1	Registers and triggers a CPU access request
FL_DATAOUT	Read	32	Holds the response data for the CPU
FL_READY	Read	1	Communicates if the core is busy running a transaction (1: not busy, 0: busy)

The FL_COMMAND and FL_COMMANDTP SW accessible registers feature special configuration bit fields. In Table 4.5 and Table 4.6 below, the configuration parameters fields are explained.

Table 4.5: FL_COMMAND SW Register Configuration Fields.

FL_COMMAND	Size	Description
7 : 0	8	**command code**: sets the command code to be used on the transaction frame
14 : 8	7	**data bits**: configures the number of bits in a transmit or receive transaction
15	1	**4-byte mode**: activates the 3-byte or 4-byte mode
19 : 16	4	**latency/dummy cycles**: specifies the number of dummy cycles for a transaction
29 : 20	10	**frame structure**: controls the number of data lanes to be used on each transaction phase
31 : 30	2	**xip phase**: enables xip mode bit phase

Table 4.6: FL_COMMANDTP SW Register Configuration Fields.

FL_COMMANDTP	Size	Description
2 : 0	3	**transaction type** : sets the command type
20	1	**dtr mode**: enables dtr transactions
21	1	**4-byte mode** : enables the 4-byte address mode
31 : 30	2	**SPI mode** : sets the SPI mode (single, dual or quad)

4.4 Central Core

The central core houses several functional components which represent the capabilities of the core. The core is responsible for: registering the inputs from the upper level modules, building the transaction frame according to desired configuration from the inputs, transferring the transaction frame through the SPI interface connected to the flash memory device, receiving eventual data sent back from the flash device and, finally, outputting the response data to the upper modules. The block diagram representing the logical components of the core is exhibited in Figure 4.2.

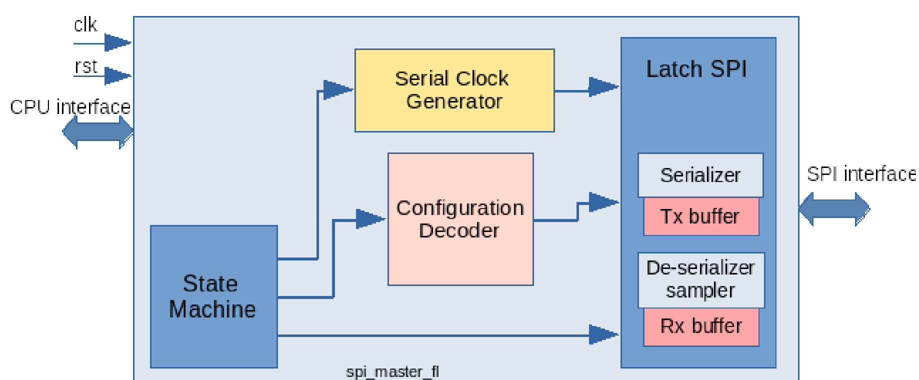


Figure 4.2: SPI Core Block Diagram.

4.4.1 Central Core Inputs and Outputs

The core comprises the input and output ports gathered in Table 4.7 .

The central core features 3 configurable module synthesis parameters, namely: CLKS_PER_HALF_SCLK predefined to 2, CPOL predefined to 1 and CPHA predefined to 1.

4.4.2 Global State Machine

The core bases its functioning on three main phases, being: the IDLE state, the SETUP state and the TRANSFER state. The diagram in Figure 4.3 illustrates the state machine through its states and state transition signal conditions.

The **IDLE** state is the phase where the core is not performing any action, no transaction is in course and no transaction setup registers are being set. In this phase, the core awaits for a transaction request

Table 4.7: SPI Core Ports.

Signal	Type	Size	Description
System Interface			
clk	Input	1	System clock signal
rst	Input	1	System reset signal
CPU Interface			
command	Input	8 (SPI_COM_W)	Command segment opcode
commtype	Input	3 (SPI_CTYP_W)	Transaction frame type selector (command type)
address	Input	32 (SPI_ADDR_W)	Transaction address segment bits
data_in	Input	32 (SPI_DATA_W)	Transaction Transmission data segment bits
validflag	Input	1	Transaction trigger signal
dtr_en	Input	1	Enable DTR mode transaction
ndata_bits	Input	7	Number of transmitted or received data bits
dummy_cycles	Input	4	Number of transaction dummy cycles
frame_struct	Input	10	Controls the number of data lanes used per transaction segment
xipbit_en	Input	2	Enables and sets the xip mode bit
spimode	Input	2	Selects the SPI data lanes mode (single, dual or quad)
fourbyteaddr_on	Input	1	Enables 4-byte addressing mode
data_out	Output Reg	32 (SPI_DATA_W)	Holds response data bits received from flash memory
tready	Output Reg	1	Indicates busy state
SPI Interface			
sclk	Output Reg	1	serial clock
ss	Output	1	slave select (chip select)
mosi_dq0	Inout	1	Connected to flash device's data lane 0
miso_dq1	Inout	1	Flash device's data lane 1
wp_n_dq2	Inout	1	Flash device's data lane 2
hold_dq3	Inout	1	Flash device's data lane 3

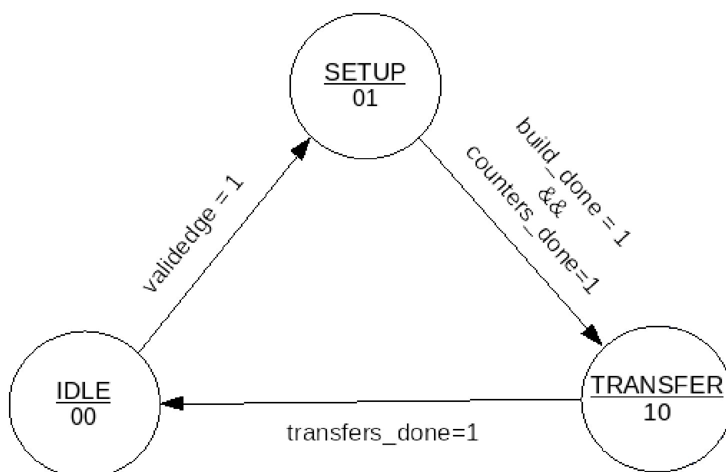


Figure 4.3: Global Core State Machine.

and the tready output port signal is set to 1.

In the **SETUP** phase, the core has received a transaction request signal and proceeds to set the necessary transaction configuration registers and to build the transaction frame. It launches the TRANSFER phase when the configuration is completed (build_done and counters_done registers both set to 1) and sets the r_transfer_start register to 1. A transaction request signal consists in receiving the validflag set to 1 signal in conjunction with having the tready signal set to 1 (IDLE state).

In the **TRANSFER** phase, the core proceeds to transmit the frame through the SPI interface to the flash device, and performs the sampling of eventual response data bits from the flash device. The transaction frame transfer occurs when the serial clock toggles, which is triggered by the r_transfer_start being set to 1. When the transaction is finished, the r_transfers_done signal is asserted and the core returns to the IDLE state.

The functionality described above is achieved by means of the internal integration of different component modules, namely: the **sclk_gen** module, the **configdecoder** module and the **latchspi** module. The component modules characterization is discussed hereafter.

4.4.3 Component Modules

The central core module, spi_master_fl module (spi_master_fl.v source file), includes the **sclk_gen**, **configdecoder** and **latchspi** modules. The component modules are described next.

4.4.3.1 Sclk_gen module description

The **sclk_gen** module is responsible for generating the serial clock signal and several serial clock synchronisation control signals. It accepts as configurable synthesis parameters the following parameters: CLKS_PER_HALF_SCLK predefined to 2, CPOL predefined to 1 and CPHA predefined to 1. This module comprises the ports described in Table 4.8.

Table 4.8: SCLK_GEN Module Ports.

Signal	Type	Size	Description
clk	Input	1	System clock
rst	Input	1	System reset
sclk_edges	Input	9	Total number of transaction serial clock edges
sclk_en	Input	1	Enables serial clock toggling
op_start	Input	1	Transaction frame transmission start
dtr_edge0	Output Reg	1	Serial clock edge synchronization for DTR mode
dtr_edge1	Output Reg	1	Serial clock edge synchronization for DTR mode
op_done	Output Reg	1	Total number of serial clock edges reached
sclk_leadedge	Output Reg	1	Serial clock edge synchronization
sclk_trailedge	Output Reg	1	Serial clock edge synchronization
sclk_int	Output Reg	1	Internal serial clock signal (one clock cycle in advance to core output sclk)

The sclk_gen module provides the signals to which the data transmission and sampling is synchronised. Namely, the module provides the following set of relevant signals for synchronisation: **se-**

rial clock leading edge (represented by sclk_leadedge), **serial clock trailing edge** (represented by sclk_trailedge), **dtr_edge0** and **dtr_edge1** and **sclk_int**. The serial clock leading edge and the serial clock trailing edge may represent a falling clock edge or a rising clock edge, respectively, depending on the CPOL and CPHA parameters. For instance, for CPOL=1 and CPHA=1, the leading edge pulses are falling edges and the trailing edge pulses are rising edges. The dtr_edge0 and dtr_edge1 signals are used for similar tasks, but are exclusive for DTR transactions. The sclk_int signal represents the serial clock signal. It toggles for every leading edge and trailing edge pulses and it is one system clock cycle in retard. It is used as clock-synchronous input for the SPI interface's sclk output register. This clock enable serial clock generation mechanism is robust and more portable than using PLL clocking resources, but has a comparatively limited maximum clocking frequency. For instance, for a 100Mhz system clock input, the maximum serial clock frequency is 25Mhz (the serial clock frequency can be adjusted by setting the module's CLKS_PER_HALF_SCLK parameter, but the minimum working parameter value is 2).

4.4.3.2 Configdecoder module descrition

The **configdecoder** module is responsible for decoding the configuration parameters, building the transaction frame and configuring the transaction control registers. The configdecoder module ports are listed in Table 4.9.

The configdecoder modules receives as input the transaction segments registers (command, address, data_in), and concatenates them to form the r_str2sendbuild register, which serves as the transaction transmission buffer. The data_in register bytes are concatenated into the r_str2sendbuild register in reversed order, so that the transmitted data segment bytes, when stored to the flash memory, match the IOb-SoC CPU's byte ordering. The module is also responsible for outputting the decoded control signals and configuring the transaction control registers (explained in Section 4.4.5).

4.4.3.3 Latchspi module description

The **latchspi** module is responsible for transmitting the transaction frame bits and sampling the response data bits from the flash device according to the configuration. It serializes the data to be transmitted and drives the SPI interface signals. It performs the latency dummy cycles synchronously to the serial clock and, finally, it samples the response data bits if any are expected. Table 4.10 summarizes the module ports.

The latchspi modules starts transmitting the transaction frame bits when it detects the serial clock edge toggling. The first bit is sent on the first leading clock edge pulse. When the bits transmission counter reaches the maximum pre-configured stop value, it signals completion by setting the sending_done and mosifinish signals to 1. This enables the performing of the pre-configured dummy cycles if they are not 0. When the dummy cycles counter reaches 0, the sampling of the response bits depend on the pre-configured command type, remaining expected number of serial clock edges, and number of expected response bits. Finally, the module outputs the received response bits, if available, through the read_data and read_datarev (with the sampled bytes in reverse order).

Table 4.9: ConfigDecoder Module Ports.

Signal	Type	Size	Description
clk	Input	1	System clock
rst	Input	1	System reset
command	Input	8 (SPI_COM_W)	command segment opcode
commtype	Input	3 (SPI_CTYP_W)	Transaction frame type selector
address	Input	32 (SPI_ADDR_W)	Transaction address segment bits
data_in	Input	32 (SPI_DATA_W)	Transaction Transmission data segment bits
spimode	Input	2	Selects the SPI data lanes mode (single, dual or quad)
nmiso_bits	Input	7	Number of received data bits
ndatatx_bits	Input	7	Number of transmitted data bits
frame_struct	Input	10	Controls the number of data lanes used per transaction segment
dummy_cycles	Input	4	Number of transaction dummy cycles
dtr_en	Input	1	Enables DTR mode transaction
fourbyteaddr_on	Input	1	Enables 4-byte addressing mode
setup_start	Input	1	Indicates entering SETUP phase
dualrx	Output	1	Receiving data bits in two data lanes enabled
quadrx	Output	1	Receiving data bits in two data lanes enabled
dualcommand	Output	1	Transmitting command segment in two data lanes enabled
quadcommand	Output	1	Tranmitting command segment in four data lanes enabled
dualaddr	Output	1	Transmitting address segment in two data lanes enabled
quadaddr	Output	1	Transmitting address segment in four data lanes enabled
dualdatatx	Output	1	Transmitting data segment in two data lanes enabled
quaddatatx	Output	1	Transmiitting data segment in four data lanes enabled
dualalt	Output	1	Transmitting data bits in two data lanes enabled (Not used)
quadalt	Output	1	Transmitting data bits in four data lanes enabled (Not used)
r_str2sendbuild	Output Reg	72	Transmission frame buffer
txcntmarks	Output Reg	30	Transaction transmission segments maximum transmitted bits count and segment SPI mode
r_build_done	Output	1	SETUP phase's transmission buffer formation complete
r_counters_done	Output	1	SETUP phase's transaction control registers configuration complete
r_sclk_edges	Output Reg	9	Total number of serial clock edges
r_counterstop	Output Reg	8	Total number of transmitted transaction bits count
r_misoctrstop	Output Reg	7	Total number of receiving bits count

Table 4.10: Latchspi Module Ports.

Signal	Type	Size	Description
clk	Input	1	System clock
rst	Input	1	System reset
sclk_en	Input	1	Serial clock toggling enable
latchin_en	Input	1	Serial clock sampling edge
latchout_en	Input	1	Serial clock transmission edge
latchout_dtr_en	Input	1	Serial clock transmission edges in DTR mode
dtr_en	Input	1	Enable DTR transaction modes
setup_rst	Input	1	Reset transaction control registers
loadtxdata_en	Input	1	Transmission buffer loading enable
mosistop_cnt	Input	8	Transmission bits maximum count
txstr	Input	72	Transmission buffer
dualrx	Input	1	Data bits received from flash in two data lanes
quadrx	Input	1	Data bits received from flash in four data lanes
dummy_cycles	Input	4	Dummy clock cycles count
misostop_cnt	Input	7	Received bits maximum count
xipbit_en	Input	2	Xip bit enable and set bit
txcntmarks	Input	30	Maximum bit counts for transmission segments and segment SPI mode
spimode	Input	2	SPI data lanes modes selector
numrxbits	Input	7	Number of received bits count
data_rx	Input	4	Concatenates the SPI data lanes for sampling bits
data_tx	Output	4	Concatenates the SPI data lanes for transmitting bits
dualtx_en	Output	1	Transmission in two data lanes enabled
quadtx_en	Output	1	Transmission in four data lanes enabled
xipbit_phase	Output	1	Xip bit transmission window
sending_done	Output	1	Transmitting bits from controller completed
mosifinish	Output	1	Transmitting bits from controller completed after flash device sampling edge
mosicounter	Output	8	Transaction transmission bits counter
read_data	Output	32	Sampled received bits
read_datarev	Output	32	Sampled received bits in reverse order

4.4.4 Configuration Parameters Encoding

The configuration registers encode information about the desired operation for the flash controller. This information is input to the configuration decoder module which translates it into transaction control values. In this section, the encoding of the configuration registers is described.

4.4.4.1 Command type

This configuration parameter, represented by the input port ”commtype”, is a 3 bit sized parameter, which represents the target transaction frame format to be executed. The encoding values are described in Table 4.11.

4.4.4.2 SPI Data Lanes modes

This 2-bit input parameter (spimode input port) represents the desired transaction mode, which can be: single SPI mode, dual SPI mode or quad SPI mode. The SPI mode configuration overrides the frame_struct configuration. Table 4.12 shows the encoding values details.

Table 4.11: Command types Encoding Description.

Value	Mnemonic	Description
000	COMM	Refers to transaction frame with only the command segment. Examples Reset Enable command (command code: 66h), Write Enable command (06h)
001	COMMANS	Transaction frame with command segment and data bytes from flash (ANS, answer). Examples Read Status Register command (05h)
010	COMMADDR_ANS	Transaction frame including command segment, address segment and data received from flash (answer). May also include dummy cycles. Memory Read operations fall into this specification. Example Read (03h), DTR Fast Read (0Dh)
011	COMM_DTIN	Transaction frame including the command segment and the data for flash segment. Usually commands which refer writing specific values to the flash device registers fall into this specification. Examples Write Status Register (01h), Write Volatile Register (81h)
100	COMMADDR_DTIN	Transaction frame including command segment, address segment, transmit data segment. Memory program operations fall into this category. Example Page Program (02h), Extended Quad Input Fast Program (12h)
101	COMMADDR	Transaction frame including command segment and address segment. This is the command type configuration used for Erase operations. Examples Sector Erase (D8h), Bulk Erase (C7h)
110	XIP_ADDRANS	Transaction Frame including address frame, data received from flash segment. This mode is set for read operations in XIP mode
111	RECOVER_SEQ	Free format transaction frame. This configuration mode can be used for manually setting the frame bits with no predefined information. Can also be used for recover sequences support by many flash devices used to for instance, reset he flash device after a power failure.

Table 4.12: SPI data lane modes Encoding.

Value	Description
00	default mode (single mode): segments on one data lane or, command segment on one lane with subsequent segments on two/four data lanes
01	dual mode: all segments transmitted or received on two data lanes
10	quad mode: all segments transmitted or received on four data lanes
11	default mode (single mode)

4.4.4.3 Transaction Frame Struct

This configuration parameter is 10-bit sized input parameter, which gives information about the SPI mode to be used on each transaction frame segment. Several current NOR SPI flash memory devices support transaction frames where the segments are transmitted or received on different SPI modes. For instance, the N25Q256A Micron family flash devices [16] support the ”Fast Read Dual Output” (command

code: 3Bh) where in the default flash device configuration, the command segment is sent on the on the mosi (dq0) data lane, the address segment also on the mosi (dq0) data lane, but the received data bits segments is on both the dq0 and dq1 data lanes. Another example, is the ”Extended Quad Input Fast Program” (command code: 12h) command, where the command segment is trasmitted on the dq0 lane, the address segment on all the data lanes (dq0, dq1, dq2 and dq3), and the data bits also on all 4 data lanes (dq0, dq1, dq2 and dq3).

Table 4.13 gives the corresponding transaction frame segment for each 2-bit configuration interval and Table 4.14 gives the configuration value effects.

Table 4.13: Frame Structure Bit Fields.

Bit interval				
9 : 8	7 : 6	5 : 4	3 : 2	1 : 0
command	address	tx data	rx data	alt

Table 4.14: Frame Structure Encoding Description.

Value	**Description**
00	Segment in single mode
01	Segment in dual mode
10	Segment in quad mode
11	Segment in single mode

4.4.4.4 DTR mode enable

The DTR mode enable is a 1 bit input signal (input port dtr_en), which supports transaction frames for DTR, as illustrated in the N25Q256 datasheet [16] for DTR commands. The address and data segments are transferred in two serial clock cycles. Table 4.15 describes the possible values and effects.

Table 4.15: DTR Mode Enable Encoding.

Value	**Description**
0	DTR mode deactivated
1	DTR mode activated (address and data bits transfered twice a clock cycle)

4.4.4.5 4-byte address mode

The 1-bit fourbyteaddr_on input input signal indicates the address segment bytes size. The N25Q256A flash device family [16] supports 3-byte and 4-byte addressing modes. Table 4.16 describes the settings.

Table 4.16: 4-byte Address Mode Enable.

Value	**Description**
0	3-byte address mode
1	4-byte address mode

4.4.4.6 XIP mode

This 2-bit configuration input parameter (input port xipbit_en) enables XIP mode transactions (address segment, data segment) and also sets a mode bit. The most significant bit enables the XIP bit effect, and the least significant bit sets the value of the mode bit; in the N25Q256A flash devices family [16], it is called the XIP confirmation bit. Table 4.17 describes the possible configuration values and respective effects.

Table 4.17: XIP Mode Enable Encoding.

Value	Description
0x	xip mode bit not enabled
10	xip mode bit set to 0
11	xip mode bit set to 1

4.4.4.7 Number of data bits

This 7-bit configuration input parameter specifies the number of data bits for the data segment. It represents the number of data bits to be sent with a PROGRAM/WRITE command type, or the number of data bits to be received, for example, with READ commands.

4.4.4.8 Number of dummy cycles

The 4-bit dummy_cycles input signal gives the number of dummy cycles for the transaction frame. For the N25Q256A flash device family, the typical default dummy cycles value is 8 cycles for single and dual modes and 10 for quad modes.

4.4.4.9 Serial Clock Edges

The r_sclk_edges register is an internal configuration register loaded with double the sum of the number of expected bits for each frame segment present in the transaction. Its value depends on the different configuration parameters, in particular the command type setting.

4.4.5 Transaction Control Registers

At the Setup state, a number of transaction control registers must be set. These registers are the following: Transmit Counter Stop (r_counterstop), Total Number of Serial Clock Edges (r_sclk_edges), Transmit Segments Maximum Count register (txcntmarks) and Receive Counter Stop (r_misoctrstop). These registers are found in the configdecoder module.

The r_counterstop register is set to the value of the total number of bits the controller transmits to the flash device. In the Transfer state, while transmitting bits through the SPI interface, the bits are counted and, when the counter reaches the r_counterstop value, the transmission is stopped. The dummy cycles phase or the receiving phase may follow or not. The r_sclk_edges register is set to the number of total expected serial clock edges. When this value is reached, while transmitting or receiving data bits, the

transaction is terminated. The txcntmarks register is a concatenation of three 10-bit segments. Each 10-bit segment holds information for a particular transaction segment, where the 2 most significant bits hold information about the corresponding segment's SPI mode for transmission (simple, dual or quad). The remaining 8 bits tell the bit-count value when a particular transaction segment ends its transmission.

For instance, considering the COMMADDR_ANS command type, Fast Read command in mode (1-4-4), STR, 3-byte addressing, the Transaction Control Registers are set to the values described hereafter:

- The r_counterstop register is set to 32, as the command segment is 8 bits and the address segment 24 bits.
- The r_sclk_edges register is set to 60, as this represents the double of the total of the serial clock cycles for the command segment (8), plus the address segment (6), plus the number of dummy cycles (8) and lastly, plus the number of receiving bits cycles (8).
- The r_misoctrstop is set to 32.
- For the txcntmarks register, the 10-bit segment at index range 9 to 0 is set to 0x008 (most significant 2 bits encode the simple SPI mode, 8 for the maximum count for the command segment), the segment at index range 19 to 10 is set to 0x220 (quad SPI mode and 32 bits for the transmission counter) and the segment at index range 29 to 20 is set to 0 as there are no further transmit segments after the address segment.

4.5 Software Driver

The software driver is written in the C programming language. The driver abstracts away the low-level details of the communication with the flash controller hardware core.

The driver functions are divided into two groups: the basic lower level functions (platform functions) which communicate directly with the SW registers, and the higher levels functions built on top of the lower level functions and which allow for advanced behaviour. New functions can be built on top of the lower level functions that best suit the user requirements.

The base functions which directly get or set values to the software accessible registers and basic auxiliary functions are displayed below (content of software/iob_spiplatform.h):

```
//Functions

void spiflash_reset();
void spiflash_init(int base_address);

//Setters
void spiflash_setDATAIN(unsigned int datain);
void spiflash_setADDRESS(unsigned int address);
void spiflash_setCOMMAND(unsigned int command);
```

```
void spiflash_setCOMMTYPE(unsigned int commtype);
void spiflash_setVALIDIN(unsigned int validin);

//Getter
unsigned int spiflash_getDATAOUT();
unsigned int spiflash_getVALIDOUT();
unsigned int spiflash_getREADY();

//General command execution trigger function
void spiflash_executecommand(int typecode, unsigned int datain,
        unsigned int address, unsigned int command, unsigned *dataout);
```

The spiflash_executecommand triggers the execution of a command on the flash controller core, depending on the command type value, by setting FL_VALIDFLAG to 1. This function also checks the controller ready state (FL_READY) to verify if it is able to accept a new command request, performing wait cycles while the controller is not ready. It is useful for simplifying the code, so that new functions can support new command codes. The implementation code for the function is exhibited below:

```
void spiflash_executecommand(int typecode, unsigned int datain,
        unsigned int address, unsigned int command, unsigned *dataout)
{
        spiflash_setCOMMAND(command);
        spiflash_setCOMMTYPE(typecode);
        while((!spiflash_getREADY()));

        switch(typecode)
        {
                case COMM:
                                spiflash_setVALIDIN(1);
                                spiflash_setVALIDIN(0);
                                break;
                case COMMANS:
                                spiflash_setVALIDIN(1);
                                spiflash_setVALIDIN(0);
                                while(!spiflash_getREADY());
                                *dataout = spiflash_getDATAOUT();
                                break;
                case COMMADDR_ANS:
                                spiflash_setADDRESS(address);
                                spiflash_setVALIDIN(1);
```

```
                spiflash_setVALIDIN(0);
                while(!spiflash_getREADY());
                *dataout = spiflash_getDATAOUT();
                break;
        case COMM_DTIN:
                spiflash_setDATAIN(datain);
                spiflash_setVALIDIN(1);
                spiflash_setVALIDIN(0);
                break;
        case COMMADDR_DTIN:
                spiflash_setADDRESS(address);
                spiflash_setDATAIN(datain);
                spiflash_setVALIDIN(1);
                spiflash_setVALIDIN(0);
                break;
        case COMMADDR:
                spiflash_setADDRESS(address);
                spiflash_setVALIDIN(1);
                spiflash_setVALIDIN(0);
                break;
        case XIP_ADDRANS:
                spiflash_setADDRESS(address);
                spiflash_setVALIDIN(1);
                spiflash_setVALIDIN(0);
                while(!spiflash_getREADY());
                *dataout = spiflash_getDATAOUT();
                break;
        case RECOVER_SEQ:
                spiflash_setDATAIN(datain);
                spiflash_setVALIDIN(1);
                spiflash_setVALIDIN(0);
                break;
        default:
                spiflash_setVALIDIN(1);
                spiflash_setVALIDIN(0);

    }
}
```

The spiflash_executecommand function receives the required arguments and proceeds to set the FL_COMMAND and FL_COMMANDTP registers for configuration. It then awaits for the controller's ready signal (IDLE state) by reading the FL_READY register. Then, it triggers the flash controller operation by setting the FL_VALIDFLG to 1. For commands that result in response data from the controller, it reads the FL_DATAOUT register to get the response data after the controller has returned to the IDLE state (FL_READY set to 1).

4.5.1 Software Driver Global Variables and High Level Functions

The iob_spi.c source file defines a set of global variables and high level functions. The global variables are used to hold configuration information that is persistent for many functions calls, and reflect the flash device configuration state. The global variables are shown below:

```
static unsigned int base;
unsigned xipframestruct = 0;
unsigned commtypeReg = 0;
```

The **base** global variable is used to store the core's base address assigned by system's memory mapping. The **xipframestruct** global variable is set to express the defined frame structure for a previously activated XIP mode, otherwise it is reset. It can be updated through the numerous fast-read high-level functions defined in file iob_spi.c The **commtypeReg** variable holds the persistent configuration information for the FL_COMMANDTP register. These configuration fields are the SPI mode (simple, dual or quad) in the 2 most significant bits (bits 31 and 30), and the 4-byte address enable in bit position 20. The commtypeReg variable should be updated every time the mentioned internal flash device configuration fields are set.

The high-level functions (defined in iob_spi.c) provide support for particular command codes. These functions are implemented on top of a function call to spiflash_executecommand. Illustrative examples are given next:

- Reset Memory command (command code 99h). This command requires that a Reset Enable (66h) command is performed before the flash device can successfully accept a Reset Memory request. The code is displayed below:

```
void spiflash_resetmem()
{
  //execute RESET ENABLE
  spiflash_executecommand(commtypeReg|COMM, 0, 0,
  RESET_ENABLE, NULL);
  //execute RESET MEM
  spiflash_executecommand(commtypeReg|COMM, 0, 0,
  RESET_MEM, NULL);
}
```

- Fast Read Command in Dual Input and Output format (command code BBh), waveform illustrated in 2.9. The command segment is transmitted in one lane, while the address and data segments are sent in two lanes (the command segment is transmitted in two lanes if the dual mode is selected). The code is illustrated next:

```
unsigned int spiflash_readfastDualInOutput(unsigned address,
unsigned activateXip)
{
    unsigned misobytes = 4, data=0;
    unsigned frame_struct = 0x00000044;// uint8 later
        unsigned dummy_cycles = 8;
    unsigned xipbit = 1;

    // 2-> Activate/keep active, 3-> terminate Xip, others ignore
    if (activateXip == ACTIVEXIP || activateXip == TERMINATEXIP)
    {
        xipbit = activateXip;
        xipframestruct = (activateXip == ACTIVEXIP) ? frame_struct: 0;
    }
    else
        xipbit = 0;

    unsigned command = (xipbit << 30) | (frame_struct<<20)|
    (dummy_cycles<<16)|((misobytes*8)<<8)|READFAST_DUALINOUT;
    spiflash_executecommand(commtypeReg|COMMADDR_ANS, 0, address,
    command, &data);
        return data;
}
```

- Fast Extended Quad Memory Program (command code 12h). The address and data segments are transmited in four lanes. This command requires that a Write Enable is issued first. The code illustrated next:

```
void spiflash_programfastQuadInputExt(unsigned int word, unsigned address)
{
        //execute WRITE ENABLE
        spiflash_executecommand(commtypeReg|COMM, 0, 0, WRITE_ENABLE, NULL);
        //execute PAGE PROGRAM
        unsigned frame_struct = 0x000000a0;
        unsigned numbytes = 4;
        unsigned command = (frame_struct << 20) | (numbytes*8 << 8)
```

```
        | PROGRAMFAST_QUADINEXT;
        spiflash_executecommand(commtypeReg|COMMADDR_DTIN, word,
        address, command, NULL);
}
```

A firmware source code (software/test_firmware.c) is also provided in the core's software directory. The firmware can be run when the core is integrated into a SoC, and hosts a set of function calls that can test the system's interaction with the flash controller.

Chapter 5

Bootloader and Core Integration to SoC

In this chapter, the core integration to the SoC platform is detailed along with the bootloader program upgrades for handling the flash controller core. It comprises four main sections: core integration to SoC as peripheral, core integration as an instruction memory, bootloader with flash functionality and, lastly, simulation and board run.

The core can be used on the SoC platform as a peripheral module and/or as an interface to an external instruction memory. As a peripheral, the core is connected to the SoC through the CPU's peripheral bus, while as an instruction memory interface it is connected to the CPU's instructions bus.

As a peripheral, the flash controller core can be accessed by both the bootloader and the firmware programs. In this mode, the bootloader and the firmware can issue the usual read and write commands and the configuration setting for the flash controller.

As an instruction memory interface, the the bootloader initially loads the flash memory with the firmware program. Furthermore, the flash controller core is configured through the peripheral interface by the bootloader, so that the flash core is able to handle the instruction read requests from the CPU when running the firmware program.

5.1 Core Integration to SoC as Peripheral

In this section, the flash controller core integration details to the IOb-SoC platform as a peripheral are presented.

To add and utilize a core as a peripheral to the IOb-SoC, there are two required steps that must be done:

1. adding the core as a git submodule in the IOb-SoC's submodules directory

2. adding the core to the peripherals list on the IOb-SoC's system.mk file

The core can be added as a submodule to the IOb-SoC's submodules directory by issuing the following command:

```
$ git submodule add git@github.com:IObundle/IOb-spi.git submodules
```

With this command the core's directory is registered as a git submodule for IOb-SoC, and it is cloned into the submodules directory.

After the core has been added as a submodule, it can be referenced in the peripherals list on the system.mk file as illustrated below:

```
#PERIPHERAL LIST
#must match respective submodule or folder name in the submodules directory
#and CORE_NAME in the core.mk file of the submodule
PERIPHERALS ?=UART SPI TIMER
```

With these two steps completed, and provided that the core presents certain interfacing auxiliary files, the core can be integrated into IOb-SoC and its functionality tested both on simulation and on FPGA board runs.

The flash controller core's interfacing auxiliary files mentioned above are listed below:

hardware/include/inst.v : describes a Verilog instantiation of the IOb_spi_master_fl core to be included in the IOb-SoC's system module as a peripheral.

hardware/include/pio.v : presents the core pins that require instantiation in IOb-SoC's ports list for external interfacing.

software/iob_spidefs.h : defines constants that represent core configuration parameters, particularly, the set of encoding of the implemented commands.

software/iob_spi.h : presents a list of function headers for functionalities such as the READ and WRITE operations on the flash registers or on the flash memory array, among others.

software/iob_spiplatform.h : presents the set of basic low-level flash controller interfacing function headers.

5.2 Core Integration to SoC as an Instruction Memory Interface

This mode is activated by setting the RUN_FLASH variable to 1. The core is connected to IOb-SoC's instruction bus and the code is run directly from the flash memory. The data bus is connected to the SRAM. The interconnection between the CPU's instruction bus and the flash controller interface goes through the instruction cache, which improves the instruction memory access performance.

5.2.1 Cache Integration

The L1 instruction cache is instantiated in the ext_flash module. The ext_flash module interfaces with the CPU's instruction bus signals, and the flash controller core's instruction interface, a native slave

interface. The ext_flash module instantiation depends on the RUN_FLASH being set to 1. The ext_flash module exhibits the list of ports described in Table 5.1.

Table 5.1: Ext_flash Module Ports.

Signal	Type	Size
System Interface		
clk	Input	1
rst	Input	1
Instructions Bus Interface		
i_req	Input	54 (1+FIRM_ADDR_W+WRITE_W -2)
i_resp	Input	33 (RESP_W)
Flash controller interface		
mem_valid	Output	1
mem_addr	Output	24 (FLASH_ADDR_W)
mem_wdata	Output	32 (DATA_W)
mem_wstrb	Output	4 (DATA_W/8)
mem_rdata	Input	32 (DATA_W)
mem_ready	Input	1

The cache component is an instance of the iob_cache module from IObundle [19]. The iob_cache module features a set of configuration parameters, namely: **front-end address width**, **back-end address width**, **number of ways**, **line offset width**, **word offset width**, **FIFO's depth width**, **cache control access enable** and **cache control counter enable**. The instruction cache is instantiated with the following configuration values: **front-end address width** set to FIRM_ADDR_W, **back-end address width** set to FLASH_ADDR_W, **back-end address width** set to 2, **line offset width** set to 4 resulting in 16 lines for each way table, **word offset width** set to 4 resulting in 16 words for each word, **FIFO's depth width** set to 5 which enables BRAM FIFO implementation, **cache control access enable** set to 0 and **cache control counter enable** set to 0.

5.2.2 Modifications to the CPU module

To successfully implement the RUN_FLASH mode, additional lines of code were introduced in the CPU wrapper module Verilog file. These modifications allow the firmware code to be read from the flash controller's instruction interface when the RUN_FLASH is set to 1 and the CPU is not running bootloader code. The code lines introduced are the following:

```
ifdef RUN_DDR_USE_SRAM
   assign ibus_req = {CPU_i_req[`V_BIT], ~boot, CPU_i_req[`REQ_W-3:0]};
   assign dbus_req = {CPU_d_req[`V_BIT],
    (CPU_d_req[`E_BIT]^~boot)&~CPU_d_req[`P_BIT], CPU_d_req[`REQ_W-3:0]};
`elsif RUN_FLASH
   assign ibus_req = {CPU_i_req[`V_BIT], ~boot, CPU_i_req[`REQ_W-3:0]};
   assign dbus_req = CPU_d_req;
`else
```

```
    assign ibus_req = CPU_i_req;
    assign dbus_req = CPU_d_req;
`endif
```

5.3 Bootloader with Flash Functionality

The bootloader program runs from the internal SRAM memory. When completed it reboots the CPU. From this point, the CPU starts executing the firmware program from the internal memory or external memory. The additional flash functionalities are defined inside ifdef code guards and are enabled by setting to 1 the corresponding variable (in the system.mk file), namely: the **RUN_FLASH** variable, the **PROGRAM_FLASH** variable, the **CHECK_FLASH** variable or the **SECTOR_ERASE** variable. The concerned variables and functionalities are described next.

5.3.1 Bootloader RUN_FLASH function

When the RUN_FLASH variable is set to 1, the CPU is enabled to run the firmware from the flash memory, after the bootloader reboots the system. In this running mode, the CPU's instruction memory becomes the flash memory, while the CPU's data memory remains the internal SRAM. In this mode, the bootloader executes a code segment to configure the flash controller SW accessible registers for running READ commands and activating low-latency running modes. The code segment is presented next.

```
#ifdef RUN_FLASH
    //Configure flash for utilization
    uart_puts (PROGNAME);
    uart_puts (":␣configuring␣flash␣parameters\n");
    unsigned frame_struct = 0x00000088;
    unsigned numbytes = 4; //max 4
    unsigned dummy_cycles = 8;
    unsigned command = (frame_struct << 20) |
    (dummy_cycles << 16)|((numbytes*8) << 8) | READFAST_QUADINOUT;

    spiflash_init(SPI_BASE);
    spiflash_reset();
    //set Xip mode in Config. Register
    spiflash_XipEnable();
    //Fast Read confirmation bit 0 to activate XIP mode
    unsigned first_word = spiflash_readfastQuadInOutput(0, ACTIVEXIP);
    command |= (ACTIVEXIP << 30);
    spiflash_setCOMMAND(command);
```

```
    spiflash_setCOMMTYPE(XIP_ADDRANS);
    //spiflash_setCOMMTYPE((0x01 << 20)|COMMADDR_ANS);
    uart_puts (PROGNAME);
    uart_puts (": configuration complete\n");

#endif
```

The code segment above illustrates the configuration of the controller core for fast read quad IO with XIP enabled mode.

5.3.2 Bootloader PROGRAM_FLASH function

Setting the **PROGRAM_FLASH** variable to 1 enables the bootloader to program the firmware binary file received from the host computer via UART onto the flash memory. The firmware binary file is stored into the flash memory starting at address 0.

```
#ifdef PROGRAM_FLASH
   //Assumes firmware.bin received by UART
    spiflash_init(SPI_BASE);
    unsigned int flash_addr = 0;
    int pages_programmed = 0;
    int last_addr = 0;
    uart_puts (PROGNAME);
    uart_puts (": Programming firmware to flash...\n");
    last_addr = spiflash_memProgram(prog_start_addr[0], file_size, flash_addr);
    uart_puts (PROGNAME);
    printf("Initial address: %x\nLast address: %x\n", flash_addr, last_addr);
    uart_puts("Testing firmware program\n");
    uart_puts (PROGNAME);
    uart_puts (": Flash programming done...\n");
#endif
```

5.3.3 Bootloader CHECK_FLASH function

Setting the **CHECK_FLASH** to 1 enables the bootloader to read the previously stored firmware data content from the flash memory and send it back to the host computer via UART. The file sent back is named "flash_firm.bin". The file size read from the flash is the value $2^{FIRM_ADDR_W}$.

```
#ifdef CHECK_FLASH
    //init SPI
```

```
spiflash_init(SPI_BASE);
unsigned start_addr = 0;
uart_puts(UART_PROGNAME);
uart_puts(": requesting to send file\n");

//send file transmit command
uart_putc(FTX);

//send file name
char flash_file_name[] = "flash_firm.bin";
int j=0;
do
  uart_putc(flash_file_name[j]);
while (flash_file_name[j++]);

//calculate firmware size
int flash_file_size = 0;
int power_of2 = 0;
for(int i=0; i < FIRM_ADDR_W; i++)
{
     if(i==0) power_of2 = 1;
     else power_of2 *= 2;
}
flash_file_size = power_of2;

// send file size
uart_putc((char)(flash_file_size & 0x0ff));
uart_putc((char)((flash_file_size & 0x0ff00) >> 8));
uart_putc((char)((flash_file_size & 0x0ff0000) >> 16));
uart_putc((char)((flash_file_size & 0x0ff000000) >> 24));

// send file contents
unsigned mem_word = 0;
for (unsigned int i = 0; i < flash_file_size; i+=4)
{
    mem_word = spiflash_readfastQuadInOutput(i, ACTIVEXIP);
    uart_putc((char)(mem_word & 0x0ff));
    uart_putc((char)((mem_word & 0x0ff00) >> 8));
    uart_putc((char)((mem_word & 0x0ff0000) >> 16));
```

```
        uart_putc((char)((mem_word & 0x0ff000000) >> 24));
    }

    uart_puts(UART_PROGNAME);
    uart_puts(": file sent\n");


#endif
```

5.3.4 Bootloader SECTOR_CLEAR function

Setting the **SECTOR_CLEAR** variable to 1, allows the bootloader to erase 64KB flash memory sector, starting from address 0.

```
#ifdef SECTOR_CLEAR
  spiflash_init(SPI_BASE);
  uart_puts (PROGNAME);
  uart_puts (": Erasing flash sector 0\n");
  //Erase first sector (64Kb, from addr 0)
  unsigned int flash_addr0 = 0;
  unsigned int statusReg = 0;
  spiflash_erase_sector(flash_addr0);
  spiflash_readStatusReg(&statusReg);
  if(statusReg != 0){
      do{
          spiflash_readStatusReg(&statusReg);
      }while(statusReg != 0);
  }
  uart_puts (PROGNAME);
  uart_puts (": Flash sector erase complete\n");

#endif
```

The above code segment erases the first 64 KB flash memory sector with a spiflash_erase_sector function call, and proceeds to verify the completion of the erasing process by reading the flash status register state (spiflash_readStatusReg function call).

5.4 Simulation and Board Run

Among the several simulation tools directly supported by IOb-SoC for local and remote simulation, the Icarus Verilog simulation and synthesis tool has been selected for this work. The Icarus Verilog simulator produces a VCD-format signal dump file. Additionally, the Gtkwave waveform viewer tool has been selected for displaying and examining the simulation waveforms.

5.4.1 Core Simulation

Currently, the SPI flash controller core only supports the Icarus Verilog simulator and the Gtkwave waveform viewer. These tools are integrated into the core's Makefile. The following make commands can be used in the core repository for simulation purposes: **make sim**, which runs the synthesis and testbench simulation; **make sim-waves**, which can be used to view the produced VCD simulation waveform in Gtkwave; **make sim-clean**, used to clean the files produced in simulation.

The **hardware/testbench/spi_fl_tb.v** file can be edited to test the core's behaviour for specific operation commands. After setting the appropriate registers, the validflag input signal should be asserted and de-asserted shortly to create an input pulse. The valigflag input pulse and the core's internal ready signal being asserted triggers the core operation. After the operation is started, the user can issue a wait for ready call to detect if the core has finished executing (ready signal returning to 1).

A flash memory model from Micron [16] for the N25Q256 flash memory family can also be instantiated in the testbench file for simulation. To best use the model, the user should specify the particular flash device present on the prototyping board, by defining it in the UserData.h file. To simulate the core, it has been defined as **N25Q256A11E**. Additional memory initialisation files can be specified by defining the **FILENAME_mem** and **FILENAME_sfdp** constants. The **FILENAME_mem** constant indicates the flash memory initialisation file, read by a readmemh Verilog function call; **FILENAME_sfdp** refers to the SFDP table information initialisation file, read through a readmemb Verilog function call.

5.4.1.1 Testbench file code segment

A testbench file code segment, illustrating the assignment of configuration values to input registers and triggering the operation, is shown below.

```
//New command
spimode = 2'b11;
data_in=32'haabbccdd;
command=8'h6b;
address=24'h555555;
commtype = 3'b100;
frame_struct = 10'h260;
xipbit_en = 2'b00;
```

```
nmiso_bits = 7'd32;
dummy_cycles = 4'd0;
dtr_en = 0;
#50
validflag=1'b1;
#20
validflag=1'b0;
#100
wait(tready);
```

5.4.2 Core FPGA Synthesis and Implementation

The core can be synthesised for FPGA implementation locally or on a remote server. The relevant Makefile commands are: **make fpga** and **make fpga-clean**. Board synthesis configuration variables are set in the core.mk file. The target server hosts a Xilinx KU040 FPGA board. The spi.xdc is the implementation constraints file.

The core supports the USE_NETLIST board synthesis flag variable. An IOb-SoC instance integrating this core can set this flag active to use its netlist description instead of performing a complete re-compilation.

5.4.3 IOb-SoC Simulation and FPGA Board Running

The IOb-SoC instance integrating the SPI flash controller core, IOb-SoC-flash, can be simulated and run on an FPGA board to test the core functionalities.

To run the provided firmware file in the flash controller repository instead of the firmware file in IOb-SoC's software/firmware directory, the former should be listed in the software/firmware/Makefile file, and the latter commented out.

When using the flash model for simulation, a new python script, makehex_flash.py (based on makehex.py), is executed to generate the firmware_flash.hex file, which contains a formatted version of the firmware binary and can be be used to initialize the flash model.

Chapter 6

Results

In this chapter, the practical implementation results are discussed. The core's features, verification and validation have been done by simulation and by running it on an actual FPGA board.

Implementation results for the implemented flash controller core are discussed and compared to the CAST flash controller core [18], in terms of their common features successfully implemented.

Performance results after integration into IOb-SoC and running actual firmware are compared to SRAM-only performance. Lastly, resource utilisation results concerning FPGA and ASIC implementations are also presented.

6.1 Flash Core Comparative Results

Comparing the features of the CAST xSPI-MC core presented in [18], the core successfully implements an important feature-set, including:

- Support for multi-lane data transfers (simple, dual and quad modes)
- Support for DTR transfers
- Support for XIP mode
- Configurable lengths for transaction segments
- Configurable data widths through defines

The core does not support the 8 data lanes mode (Octal mode), as the flash memory device in the prototyping board supports only a maximum of four data lanes in quad mode. However this feature can easily be implemented following the implementation format for the already in place QSPI support.

The core expects a maximum data buffer width of 32 bits with special support for 8-bit and 16-bit widths. Data read transactions with different bit widths other than 8 and multiples can also be performed, but the user should be attentive of the output format.

Considering that numerous flash devices support continuous (burst) read modes, where long contiguous memory locations can be read in one transaction, implementing the buffers through FIFOs would

allow the core to take advantage of the additional storage capacity, and of the first-in-first-out mechanism, to make more data available with fewer transactions and less overhead.

Basic framework functions are available, on top of which the user can implement support for other command codes.

6.2 Performance Results

In this section, the code execution performance comparative results are presented. The performance results for the code running from flash (with an L1 cache) are compared against the code running performance on SRAM. The core supports flexible initial flash instructions for the interface configuration, done by the bootloader program, which affects the instructions read performance of the firmware words. The experiments are run on the KU040 board.

6.2.1 Experimental Setup

The code running performance is tested against the internal SRAM running performance baseline. A number of possible initial configurations of the flash controller interface are set for experimentation, namely: quad input output fast read mode with XIP disabled and enabled, and dual input output fast read mode with XIP activated.

All the mentioned modes are configured for 8 dummy cycles, except the first. Also, the modes are entered from the simple SPI mode.

The instructions cache configuration is set to 2 ways, 16 lines and 16 words (32 bits) per line.

The performance is measured by using the TIMER peripheral to track the time duration for the firmware execution. The serial clock frequency is 25 MHz, based on a 100 MHz system clock. The firmware code used for the experiments is listed below:

```
int main()
{
  //init uart
  uart_init(UART_BASE,FREQ/BAUD);
  timer_init(TIMER_BASE);

  uart_puts("\n\n\nHello world!\n\n\n");
  int a = 11;
  printf("\nValue of this is %d\n\n", a);
  printf("\n\nValue of Pi = %f\n\n", 3.1415);

  printf("\nExecution time: %d clock cycles\n",
  (unsigned int) timer_get_count());
  printf("\nExecution time: %dus @%dMHz\n", timer_time_us(), FREQ/1000000);
  uart_finish();
```

```
}
```

6.2.2 Results Analysis

Table 6.1 summarises the performance results obtained for the configurations presented above and fixed cache parameters. The results are reported for board implementation runs.

Table 6.1: Performance Results.

Configuration	Execution time		Performance Comparison	
	clock cycles	duration(us)	clock cycles	duration
SRAM	568235	8916	–	–
Quad IO Fast Read XIP enabled	768514	11178	+35.2%	+25.3%
Quad IO Fast Read XIP disabled	840129	12022	+47.8%	+34.8%
Dual IO Fast Read XIP enabled	876801	12453	+54.3%	+39.6%

As expected, running code from the flash memory is considerably slower than from the SRAM. The fastest running configuration mode is Quad IO Fast Read with XIP enabled as expected. The reason is that in this mode the flash controller uses the maximum 4-bit data width for the address and data segments, and the command segment is dropped as a result of enabling the XIP mode. Disabling the XIP mode resulted in a 12.6% increase in clock cycles, which reflects the substantial impact of the additional 8 clock cycles for the command segment. Nevertheless, the extra 8 clock cycles for the command segment in quad IO still results in better efficiency than for dual IO with XIP enabled.

The performance results can be improved by decreasing the number of dummy cycles. The flash memory device on the prototyping board supports adjusting the number of dummy cycles depending on the operational serial clock frequency. Running the serial clock at 25MHz allows decreasing the default number of dummy cycles (8 and 10) to 1. Adjusting the number of dummy cycles can be done through the flash device's Non-volatile Configuration Register and the Volatile Configuration Register.

The performance results could also be improved by supporting the continuous word (32 bits) read mode (burst reads), which would allow the controller core to provide the cache with more memory words for fewer controller transactions. The cache configuration also affects the performance.

The fastest mode is theoretically the quad IO fast read in DTR mode. It can spare a few clock cycles compared to quad IO fast read mode with XIP enabled. The DTR mode was not used for the experiments as it currently only accurately works in simulation.

6.2.3 Experimentation Setup 2

The code running performance of a FFT kernel implementation is tested. The FFT kernel is a more normal code execution case than the one featured in the previous experimental setup and can largely benefit from cache memory utilization.

The performance is tested against the SRAM performance for two cache memory configurations, namely: 2 KB cache (2 ways, 16 lines, 16 words), and 16 KB cache (4 ways, 64 lines, 16 words). The

flash controller configuration is set to quad input output fast read mode with XIP enabled, 10 dummy cycles.

The performance is measured by using the TIMER peripheral. The serial clock frequency is 25 MHz. The FFT kernel firmware code FFT and main functions are listed below:

```
void FFT(void){
  short iter, blocks, butterflys;
  short b_max, bf_max;
  short index, index_2, index_ROM;
  int re_1, im_1, re_2, im_2, mult_1, mult_2;

  bf_max = 1;
  b_max = N>>1;

  for(iter = 0; iter < i_max; iter++){
    for(blocks = 0, index = 0; blocks < b_max; blocks++){
      for(butterflys = 0, index_ROM = 0; butterflys < bf_max;
      butterflys++, index++){
                index_2 = index + (blocks << iter);

                if(iter == 0){
                  re_1 = x[2*reversed((int)index_2)];
                  im_1 = x[2*reversed((int)index_2)+1];

                  re_2 = x[2*reversed((int)(index_2+bf_max))];
                  im_2 = x[2*reversed((int)(index_2+bf_max))+1];
                }else{
                  re_1 = X[2*index_2];
                  im_1 = X[2*index_2+1];

                  re_2 = X[2*(index_2+bf_max)];
                  im_2 = X[2*(index_2+bf_max)+1];
                }

                mult_1 = mult(re_2,w[2*index_ROM]);
                mult_2 = mult(im_2,w[2*index_ROM+1]);

                /*Real part*/
                X[2*index_2] = mult_1 - mult_2 + re_1;
```

```
                X[2*(index_2+bf_max)] = re_1 - (mult_1 - mult_2);

                mult_1 = mult(im_2,w[2*index_ROM]);
                mult_2 = mult(re_2,w[2*index_ROM+1]);

                /*Imaginary part*/
                X[2*index_2+1] = mult_1 + mult_2 + im_1;

                X[2*(index_2+bf_max)+1] = im_1 - (mult_1 + mult_2);

                index_ROM = index_ROM + b_max;
            }
        }

        bf_max = bf_max<<1;
        b_max = b_max>>1;
    }

    return;
}

int main(void){

    // init uart
    uart_init(UART_BASE, FREQ/BAUD);
    timer_init(TIMER_BASE);

    printf("\nCalculate FFT\n");
    FFT();

    printf("\nExecution time: %d clock cycles\n",
    (unsigned int) timer_get_count());

    printf("\nExecution time: %dus @%dMHz\n", timer_time_us(), FREQ/1000000);

    uart_finish();
    return 0;
}
```

6.2.4 Results Analysis 2

Table 6.2 summarises the performance results obtained for the configurations presented.

Table 6.2: Performance Results.

Configuration	Execution time		Performance Comparison	
	clock cycles	duration(us)	clock cycles	duration
SRAM	2239800	25637	–	–
Cache 2KB	2270865	26305	+1.386%	+2.6%
Cache 16KB	2267595	26191	+1.24%	+2.16%

As expected, running a cache-intensive FFT kernel from flash reports marginal performance degradation compared to the SRAM performance for sufficiently large cache. The minimal performance degradation is due to cache filling which once completed reveals very high hit rate. The 16 KB cache is more adequate than the 2 KB cache presenting an important performance increase.

The average 32-bit word access overhead for the flash memory is around 100 clock cycles in the selected mode (quad input output fast read mode with XIP enabled). While SRAM has 4 clock cycles access latency. Thus, cache memory utilization is essential for approximating the SRAM and flash memory code running performance levels.

6.3 FPGA Implementation Results

Tables 6.3 and 6.4 present the implementation results for the Kintex Ultrascale KU040 board FPGA and the Cyclone V GT FPGA, respectively. The instructions interface is enabled.

Table 6.3: Xilinx FPGA Resource Utilization Results.

Resource	Utilization
LUTs	565
Registers	519
DSPs	0
BRAM	0

Table 6.4: Intel FPGA Resource Utilization Results.

Resource	Utilization
ALM	375
FF	561
DSPs	0
BRAM blocks	0

The Xilinx FPGA implementation could reach 384.6 MHz operation frequency. The Cyclone V FPGA implementation can guarantee at least 152.53 MHz of operation frequency.

Comparing the resource utilization results from the Xilinx and Intel FPGAs to the ones reported in [18] and [20] from CAST, respectively, it can be observed that the core consumes close to half the

LUT resources for the Xilinx FPGA and almost 40% of the ALM resources for the Intel FPGA. This is due to the fact that the CAST flash controller implements a more complex CPU interface (AHB) and a more complex SPI interface that uses up-to 8 data lanes (Octal), and for being compatible with many proprietary NOR SPI protocols.

6.4 ASIC Implementation Results

Table 6.5 presents the implementation results for the UMC 130 nm ASIC, with the instruction interface enabled.

Table 6.5: UMC Asic Resource Utilization Results.

Type	**Instances**	**Area(um^2)**	**Area** %
sequential	518	16928.000	60.3
inverter	84	328.960	1.2
tristate	4	25.600	0.1
logic	996	10769.920	38.4
total	1602	28052.480	100.0

The CAST ASIC implementation [21] reports 11,500 Gates (Area) utilisation. To obtain comparable values, the UMC ASIC total area (28052.480) should be divided by the area of NAND2 gate ($5.12\mu m^2$) which results in 5,479 Gates. The developed core ASIC implementation consumes nearly half the resources compared to the CAST IP core, which is consistent with the FPGA results above and can be explained in the same way.

The above results have been obtained for a 200 MHz system clock constraint. The maximum achievable clock frequency has not been obtained but it should be much higher than this.

Chapter 7

Conclusions

In the present dissertation, a new flash memory controller design is proposed and implemented, and the bootloader program of IOb-SoC, an open-source RISC-V-based platform, is upgraded to accommodate the flash memory core for running code and permanent data storage. Software driver functions have been developed for the controller, which is called in the bootloader program.

Flash memory is a cheap, reprogrammable and non-volatile memory solution, which is very useful for embedded applications. Recent SPI NOR flash memory devices offer high-speed multi-bit access and implement support for low latency read modes specially designed for direct code execution (execute-in-place mode), avoiding the need for RAM code shadowing.

7.1 Achievements

Upon completion of this work, which reached all the initially defined objectives, the following main achievements can be enumerated:

An SPI master flash controller core has been developed and can perform the basic memory-read and memory-write operations with acceptable performance. The controller core can be dynamically configured in several operating modes using configuration registers. The controller supports multiple commands, multi-bit access, XIP mode, DTR mode and multi-length transaction segments.

The controller can be connected to the processor instruction bus and run code directly from the flash. With a reasonably sized instruction cache, only a few per cent degradation in performance is observed compared to running the code from SRAM. Without a cache or a small one, the performance penalty can be significant (up to 25x slower), but it is still effective.

The bootloader program has been upgraded to load code to the flash and restart the system to run it. By using the developed flash controller core driver functions, the bootloader features four main additional features, namely: programming firmware to flash, erasing flash memory blocks, flash memory inspection (useful for debugging) and flash controller pre-configuration for firmware execution.

The developed controller core has been successfully integrated into IObundle's IOb-SoC platform, which can now execute code from a flash device, and use the same device for permanent storage. This

feature is vital for IOb-SoC to target stand-alone embedded systems that can boot from their own non-volatile memory. The integration required the development of two IOb-SoC specific native interfaces, one that connects to the L1 instruction cache for running code, and another that connects the controller as a peripheral so that programs can use flash to save and retrieve non-volatile data.

Comparing the core's performance to a commercially available implementation from CAST [18] in terms of supported features, it can be said that both cores implement a similar set of essential features for multi-data lane modes (simple, Dual and Quad), multi-speed modes (STR and DTR) and XIP mode.

The commercial core can also offer other supplementary features such as DMA support, auto-configuration support and eight data-lane modes (Octal), unsupported in the developed core. Accordingly, the resource usage of the commercial IP core is twice that of the developed flash controller.

In terms of the maximum serial clock frequency, the commercial core can achieve 100 MHz. The developed core achieves a maximum frequency of one-fourth of the system clock frequency. It has been tested with a 25 MHz serial clock speed for a 100 MHz system clock.

7.2 Future Work

The present work can be further expanded in several ways, particularly in terms of developing extra features for the flash controller core.

The core does not support simultaneous use of the peripheral interface and instructions interface because of the shared SW registers configuration. This limitation makes it impossible for the core to respond to read or write requests while running firmware from the flash. The core can be extended to implement independent instruction and peripheral interfaces.

A significant development for the core is to implement FIFO based transmit and receive buffers which would allow for extended continuous read mode support, and generally less access latency for contiguous memory locations. The added support for DMA and interrupts frees the CPU for other tasks, providing overall better efficiency.

Another important missing feature is the implementation of an asynchronous serial clock solution, which would allow higher serial clock frequencies but would incur in added complexity and resource usage for dealing with clock domain crossings.

At the IOb-SoC level, a wear-levelling mechanism to increase the flash device lifespan is an interesting future development. Flash memories specify a limited number of Program or Erase operations over a particular memory block before the block becomes unreliable. Such a mechanism is accomplished by distributing the Program or Erase operations into multiple memory sectors. By spreading the usage into many sectors, the quick wearing of a particular memory sector is avoided.

The development of a parallel flash memory controller is also an interesting line of work, enabling the prototyping with boards having parallel flash memory devices, which offer considerably higher access speeds.

Bibliography

[1] J. T. de Sousa and et. al. Iob-picorv32. https://github.com/IObundle/iob-picorv32, February 2021. GitHub repository.

[2] J. T. de Sousa and et. al. Iob-darkrv. https://github.com/IObundle/iob-darkrv, April 2021. GitHub repository.

[3] *AXI Reference Guide*. Xilinx, March 2011.

[4] Harvard architecture. https://teachcomputerscience.com/harvard-architecture/. Accessed July 2021.

[5] J. T. de Sousa and et. al. Iob-soc. https://github.com/IObundle/iob-soc, May 2021. GitHub repository.

[6] Icarus verilog. http://iverilog.icarus.com/. Website Accessed on May 2021.

[7] *Kintex UltraScale KU040 Development Board*. Avnet, Inc., 12 2015. Version 1.0.

[8] *NAND Flash 101: An Introduction to NAND Flash and How to Design It in to Your Next Product*. Micron, 2006. URL https://web.archive.org/web/20160604054353/https://www.micron.com/~/media/Documents/Products/Technical%20Note/NAND%20Flash/tn2919_nand_101.pdf. Rev. B 4/10 EN, Archived.

[9] *Wear Leveling*. Cypress, 3 2021. URL https://www.cypress.com/file/202541/download. Rev. *E.

[10] *HyperBus™ Specification*. Cypress, 2 2019. Rev. *H.

[11] X. Consortium. Xccela bus. https://www.xccela.org/technology/. Accessed in May 2021.

[12] *S29GL01GT/S29GL512T*. Cypress, 4 2019. Rev. *L.

[13] *SPI Block Guide*. Freescale Semiconductor, Inc., 7 2004. V04.01.

[14] Jedec. https://www.jedec.org/about-jedec. Accessed in July 2021.

[15] *Expanded Serial Peripheral Interface (xSPI) for Non Volatile Memory Devices*. JEDEC, 2 2020.

[16] *1.8V, 256Mb: Multiple I/O Serial Flash Memory Features*. Micron, 2012. Rev. O 11/16 EN.

[17] *Serial Flash Discoverable Parameters (SFDP)*. JEDEC, 8 2019.

[18] *xSPI Flash Memory Controller*. CAST, Inc, . Xilinx FPGA.

[19] J. T. de Sousa and et. al. Iob-cache. `https://github.com/IObundle/iob-cache`, May 2021. GitHub repository.

[20] *xSPI Flash Memory Controller*. CAST, Inc, . Intel FPGA.

[21] *xSPI Flash Memory Controller*. CAST, Inc, . ASIC.

Appendix A

IOb-SPI Core Directory Tree

```
iob-spi
├─ document/
│  ├─ pb/
│  ├─ ug/
│  ├─ figures/
│  └─ .tex files
├─ hardware/
│  ├─ src/
│  ├─ testbench/
│  ├─ simulation/
│  ├─ fpga/
│  ├─ include/
│  └─ harware.mk
├─ software/
│  ├─ embedded/
│  ├─ pc/
│  ├─ fpga/
│  ├─ software.mk
│  ├─ test_firmware.c
│  └─ .h hearder files
├─ submodules/
│  ├─ INTERCON/
│  ├─ LIB/
│  └─ TEX/
├─ core.mk
├─ Makefile
└─ Readme.md
```