Bootloader for a RISC-V processor that uses Flash Memory

José Heraldo Furtado Fernandes
joseffernandes@tecnico.ulisboa.pt

Instituto Superior Técnico, Lisboa, Portugal

September 2021

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 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

1. Introduction

IObSoC is a System on a Chip which incorporates a RISC-V CPU, a memory system comprised by internal SRAM and external DDR memory and external peripheral IP cores integration. The system boots into a bootloader program which is able to initialize the memory resources with a firmware program. It then launches the firmware program. The objective of this work is to develop support for flash memory and its utilization into the bootloader program. The integration of flash memory into IOb-SoC should allow for direct firmware instructions execution from flash. The development of a flash memory controller IP core is required, and it must implement the IOb-SoC specific native interface. This work is developed on the Kintex UltraScale KU040 FPGA prototyping board. The prototyping board houses a User Code SPI NOR flash memory device.

The next sections comprise information concerning: Background, presenting relevant information for a general understanding of the development environment and system architecture, alongside a general characterization of flash memory and associated communications protocols; Implementation, explaining the SPI flash memory controller core development details and its integration into the IOb-SoC architecture; Results, evaluating the developed controller supported features and discussing the code execution performance and board FPGA implementation results; Conclusions, reporting the achievements and future work perspectives.

2. Background

2.1. IOb-SoC Architecture

The basic IOb-SoC system comprises the following components:

- RISC-V processor, based on the PicoRV32 processor implementation
- Memory Subsystem, consisting of a boot ROM and internal SRAM with additional support for external DDR memory
- Peripheral Cores (UART), comprising the cores added to the system directories and peripheral list. The UART communications peripheral is present by default.

- Interconnect, Instructions and Data Bus, connecting the system components. It implements the Native interface protocol signals for communication.

2.1.1 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 1. The signal direction is represented with respect to the CPU.

Table 1: Native Interface Bus Signals.

Signals	Direction	Width	Function
valid	output	1	Request
address	output	32	Request
wdata	output	32	Request
wstrb	output	4	Request
rdata	input	32	Response
ready	input	1	Response

A master read request to a slave device based on the Native interface protocol is illustrated in Figure 1.

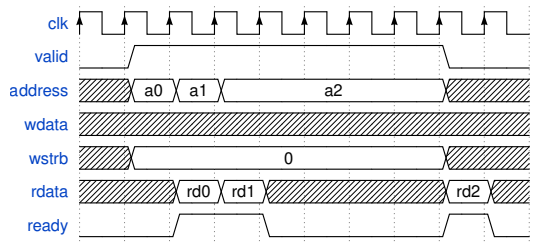

Figure 1: Native Interface Read

2.1.2 Development Environment

The IOb-SoC project development files are separated into the following main directories: **hardware**, which contains RTL description files, simulation testbench files, simulator files and board specific files and scripts for synthesis/implementation; **software**, which contains the bootloader and firmware C source files, headers and helper scripts;**document**, which contains documentation about IOb-SoC and components; **submodules**, which gathers complementary repositories for important components and peripheral cores. The project components' compilation, synthesis, board implementation and simulation processes are automated through the use of a recursive Makefiles tree structure.

The system supports synthesis and board implementation on particular ASIC and FPGA devices. Simulation is supported for a range of simulation tools, namely: icarus, verilator, ncsim, modelsim. The selected simulator for this project is icarus.

The IOb-SoC is a memory-mapped system which assigns an address range to each connected peripheral and memory device.

2.1.3 Operation

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.

2.2. Development Board Features
The Kintex UltraScale KU040 board (from here on, referred only as KU040) features 32MB SPI NOR flash memory with a QSPI interface. The flash memory device is a Micron N25Q256A family device. Figure 2 depicts the flash device signals connections to the board FPGA [1].

2.3. NOR Flash Memory
Flash memory is based on EEPROM. It is a reprogrammable, cheap and non-volatile data storage medium. There are two main types of flash memory: NAND flash and NOR flash memory. NAND flash has more advantages for large files storage and access of large memory blocks, while NOR flash is more reliable and has higher access performance for

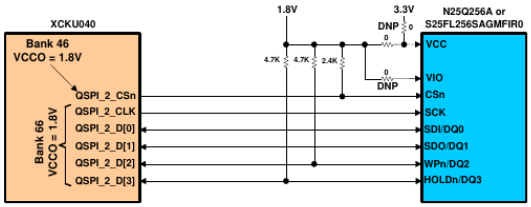

Figure 2: User flash QSPI interface.

individual bytes. NOR flash is the best one suited for running instructions code in embedded systems. A flash memory limitation is the limited number of program /erase cycles that can be performed on a block before it becomes unreliable.

NOR flash can be offered with parallel interface or serial interface. Parallel interface provides better access performance at the cost of high pin count. A common serial interface for NOR flash is SPI (Serial Peripheral Interface) which traditionally features a minimum of 4 signals: 2 control signals, CSn (Chip Select) and SCLK (Serial Clock), and 2 data signals, MOSI/SDI (Master Out Slave In/Slave Data In) and MISO/SDO (Master In Slave Out/Slave Data Out). A range of SPI evolution protocols have been developed featuring an increased number of data signals which results in greater access performance. For instance, the QSPI (Quad SPI) protocol which features up-to 4 data signals.

2.3.1 General SPI operation

An SPI operation is processed in the following manner: the master device which may be connected to multiple slave devices each with its independent CSn line, asserts to 0 a selected slave device's corresponding CSn line signal (this slave device is the only one that can successfully make transfers with the master); next, the master device starts toggling the SCLK line signal and data is sent synchronously to the SCLK edges. Data may be simultaneously sent and received by both devices, though the MOSI - SDI and MISO - SDO connections. The communication is interrupted by setting CSn back to 1.

In a SPI operation, 4 different SPI clocking modes can be defined depending on the edge used for transmitting and sampling, clock phase (CPHA), and on the serial clock edge idle state, clock polarity (CPOL), namely the modes, format (CPOL, CPHA): (0,0), (0,1), (1,0) and (1,1). In particular, the most common modes for flash memory applications are SPI clocking modes (0,0) and (1,1), values for (CPOL, CPHA), where data is transmitted on the falling edge and sampled on the rising edge, the difference being in the clock idle state: for CPOL=0 the clock is idle at 0 and for CPOL=1 the clock is idle at 1. Modes (0,0) and (1,1) may also be referred as SPI clocking modes 0 and 3, respectively.

2.4. QSPI Protocol

The QSPI (Quad SPI) protocol makes use of a subset of the specifications for the xSPI (eXpanded SPI) standard defined in [4]. The xSPI standard defines protocols for transactions (read and write) involving compliant low signal count high speed serial devices. It is defined for a maximum of 8 data signals.

The QSPI protocol defines 4 data signals (bidirectional), plus the CSn and SCLK control signals, instead of the traditional 2 unidirectional data signals for SPI. The signals are: CSn (Chip Select), SCLK (Serial Clock), DQ0 (MOSI), DQ1(MISO), DQ2 and DQ3. The extra data signals (DQ2 and DQ3) may have added functionality depending on the flash device.

2.4.1 QSPI Transaction Format

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

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

Figure 3 depicts the sequence of transaction phases for a general read or program transaction. The displayed values refer typical values for the flash chip [5] on board the KU040 prototyping board.

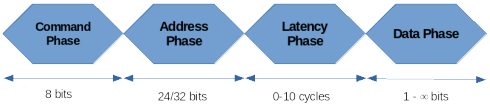

Figure 3: QSPI transaction phases

The **command phase** is normally the first transaction phase and consists of a 8 bit command code. This command opcode determines the following transaction phases characteristics, with some transaction phases omitted and others required.

In XIP (eXecute-In-Place) mode the command phase is dropped, thus saving a few transaction clock cycles for memory read operations.

The **address phase** specifies the memory location on the flash device requested for access. Current common flash chips allow for 24 bits or 32 bits addresses for larger memory capacities. Examples of operations requiring this phase are memory read and memory program operations.

The **latency phase** executes a number of required wait (dummy) cycles before the flash device

is able to correctly output requested data. On this phase, the data lines may be left unasserted, but can be set (mode bits) to relay certain information about the following transactions.

On the **data phase**, 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 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.

Basic commands supported by SPI NOR flash devices include (typical command size, address size, latency and data size found in [5]):

- Read SFDP : command code 5Ah, address size 3 bytes, latency cycles 8, data size 1 to ∞ (bytes)
- Write Enable : command code 06h (only transaction segment)
- Fast Read : command code 0Bh, address size 3/4 bytes, latency cycles 8, data size 1 to ∞ (bytes)
- Page Program : command code 02h, address size 3/4, data size 1 to 256 (bytes)

2.4.2 Transaction Waveforms

Figure 4 depicts the **Write Enable** command (command code 06h) in mode single mode (1 data lane).

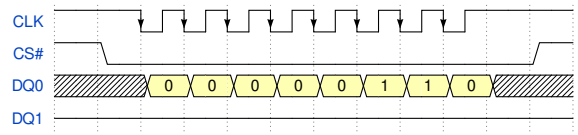

Figure 4: Write Enable Transaction Waveform.

Figure 5 illustrates the **Page Program** command (02h) in single mode .

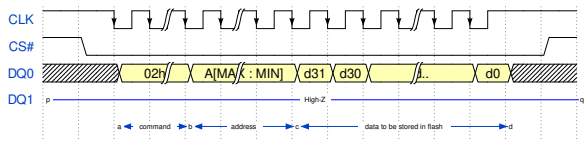

Figure 5: Page Program Transaction Waveform.

Figure 6 displays the transaction waveform for a **Fast Read** command (BBh) in mode (1-2-2), 1 data lane for command segment, 2 for address and 2 for data, with one mode bit (at 0).

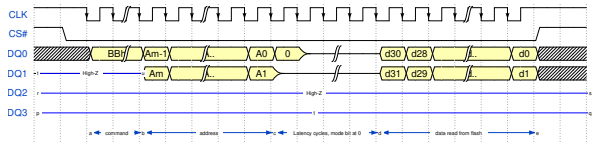

Figure 6: Fast Read Transaction Waveform Dual.

2.5. Flash Controller Comparative Model

The CAST xSPI-MC (memory controller) [2] core IP is a flash memory controller core compliant with the xSPI standard [4] which supports several important features, including: multi-data lanes modes support (single, dual, quad, octal, etc), STR (Single Transfer Rate) and DTR (Double Transfer Rate), XIP mode support, configurable transaction segments. Complete characterization information in [2]. This model is used as a comparative model for the developed flash memory controller core.

3. Implementation

3.1. Flash Memory On Board

The N25Q256A flash memory device on board the KU040 prototyping board features: 256 megabits memory density (32MB) organized into 512 sectors of 64KB representing 8192 subsectors of 4KB and a total of 131,072 pages of 256 bytes; multiple I/O data lanes; DTR mode; XIP mode support.

The flash device hosts several configuration registers that define the device's behaviour, including: the status register, the non-volatile and volatile configuration registers, flag status register, etc. At the initial delivery state all memory array bits are 1 (FFh bytes), status register set to 00h and non-volatile configuration register bits set to 1s (FFFFh).

Full characterization in [5].

3.2. Core Developement

The developed IP core implements support for a range of features including: multi I/O data lanes (single, dual, quad), XIP mode, DTR mode, CPU peripheral bus interface and CPU instructions bus interface, configurable serial clock with synchronous clock design. The project files repository is hosted on Github at `https://github.com/IObundle/iob-spi`.

The IP core is represented by a wrapper module (iob_spi_master_fl.v) which incorporates CPU peripheral bus and instructions bus interfaces, a SPI protocol interface, software accessible registers (SW) and the central SPI master flash controller core.

The Peripheral interface is used to configure the core behaviour through the SW accessible registers and perform atomic command transactions. The Instructions interface is used to connect to the

CPU instructions memory bus (via cache). The Instructions interface should be used after an initial configuration through the SW accessible registers. Both interfaces implement the CPU's native interface protocol.

3.2.1 SW Accessible Registers

The SW accessible registers comprise the registers described in Table 2.

Table 2: Software Accessible Registers.

Name	Register Type	Size
FL_RESET	Write	1
FL_DATAIN	Write	32
FL_ADDRESS	Write	32
FL_COMMAND	Write	32
FL_COMMANDTP	Write	32
FL_VALIDFLG	Write	1
FL_DATAOUT	Read	32
FL_READY	Read	1

The FL_COMMAND and FL_COMMANDTP registers hold special bit fields for particular configuration parameters. Table 3 describes the parameter bit fields in the FL_COMMAND register while Table 4 indicates the ones contained in the FL_COMMANDTP register.

Table 3: FL_COMMAND SW Register Configuration Fields.

FL_COMMAND	Size	Description
7 : 0	8	**command code**
14 : 8	7	**data bits**
15	1	**4-byte mode**
19 : 16	4	**latency cycles**
29 : 20	10	**frame structure**
31 : 30	2	**xip phase**

3.3. Central Core

The central core is responsible for: registering the inputs from the upper level modules, building the transaction frame according to desired configuration from 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 outputting this response data to the upper modules. The central core

Table 4: FL_COMMANDTP SW Register Configuration Fields.

FL_COMMANDTP	Size	Description
2 : 0	3	**transaction type**
20	1	**dtr mode**
21	1	**4-byte mode**
31 : 30	2	**spi mode**

(spi_master_fl.v) functionalities are accomplished by the integrating the following submodules: the serial clock generator module (sclk_gen.v), the configuration decoder module (configdecoder.v) and serial transmitter/sampler module (latchspi.v).

The core's block diagram is presented in Figure 7.

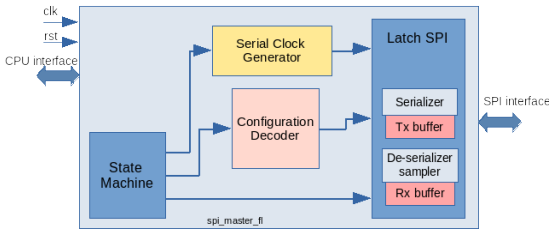

Figure 7: SPI Core Block Diagram.

The central core accepts the following module parameters: CLKS PER HALF SCLK predefined to 2, CPOL predefined to 1 and CPHA predefined to 1. It includes the input and output ports detailed in Table 5.

3.3.1 Global State Machine

The core bases its functioning on three main phases, being: the **IDLE** state, the **SETUP** state and the **TRANSFER** state. 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 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 building the transaction frame. It launches the TRANSFER phase when configuration is completed (build done and counters done registers both set to 1) and asserts the r_transfer_start register to 1. A transaction request signal consists in receiving a validflag set to 1 signal in conjunction to having the tready signal to 1 (IDLE state). In the **TRANSFER** phase the core proceeds to transmitting the frame through the SPI interface

Table 5: SPI Core Ports.

Signal	Type	Size
System Interface		
clk	Input	1
rst	Input	1
CPU Interface		
command	Input	8
commtype	Input	3
address	Input	32
data_in	Input	32
validflag	Input	1
dtr_en	Input	1
ndata_bits	Input	7
dummy_cycles	Input	4
frame_struct	Input	10
xipbit_en	Input	2
spimode	Input	2
fourbyteaddr_on	Input	1
data_out	Output Reg	32
tready	Output Reg	1
SPI Interface		
sclk	Output Reg	1
ss	Output	1
mosi_dq0	Inout	1
miso_dq1	Inout	1
wp_n_dq2	Inout	1
hold_dq3	Inout	1

to the flash device, and performs the sampling of eventual response data bits from the flash device. The transaction frame transfer occurs when detecting serial clock edges toggling which is triggered by the r_transfer_start being set to 1. When the transaction is finished, the r transfers done signal is asserted to 1 and the core returns to the IDLE state.

3.3.2 Component Modules

The **sclk_gen** module is responsible for generating the serial clock signal and several serial clock synchronization control signals. It accepts as configurable module level parameters the following parameters: CLKS PER HALF SCLK predefined to 2, CPOL predefined to 1 and CPHA predefined to 1. It generates the serial clock signal and serial clock edge enable signals to which the transaction bits transmission and bits sampling can be synchronized, particularly, the serial clock leading edge and trailing edge signals.

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

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 samples the response data bits if any expected.

3.3.3 Command Type Configuration Encoding

This configuration parameter, represented by the input port "commtype", is a 3 bit sized parameter which represents the targeted transaction frame format to be executed. The encoding values are described in Table 6.

Table 6: Command types Encoding Description.

Value	**Mnemonic**
000	COMM
001	COMMANS
010	COMMADDR_ANS
011	COMM_DTIN
100	COMMADDR_DTIN
101	COMMADDR
110	XIP_ADDRANS
111	RECOVER_SEQ

3.3.4 Transaction Control Registers

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

The r_counterstop register is set to value of the total of bits for transmission by the controller to the

flash device. In the Transfer state, while transmiting bits through the SPI interface, the bits transmited 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 register 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) while the remainder 8 bits refer the bits count value where 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 (8).
- The r_misoctrstop is set to 32.
- The txcntmarks register for bit range 9 to 0 is set to 0x008, for bit range 19 to 10 is set to 0x220 and for the segment for bit range 29 to 20 is set to 0.

3.4. Software Driver

The driver software is written in the C programming language. The software driver provides two sets of functions: basic lower level functions (platform functions) which communicate directly with the SW accessible registers, and the higher level functions built on top of the lower functions which allow for advanced behaviour.

The platform spiflash_executecommand driver function abstracts writing to SW registers and triggers the start of a transaction on the controller depending on the transaction command type.

High level functions such as spiflash_resetmem, spiflash_readfastDualInOutput, spiflash_programfastQuadInputExt implement a transaction request for the Reset Memory (command code: 99h), Read Fast Dual Input Output (BBh), Fast Program Quad Input Extended (12h), respectively (using nomenclature from [5]). The user can implement other functions on top of the platform functions to best suit requirements.

3.5. Bootloader and Core Integration to SoC

In this section the core integration to the SoC platform is detailed along with the bootloader program upgrades for handling the flash controller core.

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

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

3.5.1 Core Integration to SoC as Peripheral

In order to utilize the core as a peripheral the core repository must be added as a submodule to IOb-SoC's git root directory and added to the PERIPHERALS LIST variable in system.mk. IOb-SoC provides special mechanisms for autonomously integrating the core into its compilation and synthesis process.

3.5.2 Core Integration to SoC as Instructions Memory

In order to integrate the core into the system's instructions memory bus, it must be first added as a peripheral core and the RUN_FLASH variable set to 1. When added as instructions memory a L1 cache memory instance is used as intermediary.

The bootloader runs from SRAM and in this mode the firmware is executed from the flash memory, while the data bus main memory is the SRAM. For a successful utilization of this mode the bootloader must preconfigure the flash memory controller before running the firmware program.

3.5.3 Bootloader with Flash Functionality

The IOb-SoC bootloader program incorporating Flash Functionality can provide four different operations by enabling the respective variable. The flash functionality operations are: preconfiguring

the flash controller to correctly responding to the instructions memory bus requests (RUN_FLASH variable), programming the firmware to flash memory (PROGRAM_FLASH variable), inspecting the flash memory content (CHECK_FLASH) and erasing a flash memory sector (SECTOR_CLEAR).

The mentioned flash functionality operations are defined inside C ifdef code guards in order to save memory.

When the **RUN_FLASH** variable is defined, the bootloader program preconfigures the flash controller for a specific transaction command. The predefined command is fast read in quad mode for address and data segments with XIP mode enabled.

Defining the **PROGRAM_FLASH** variable allows for programming the firmware binary received by UART from the host computer to the flash memory starting from address 0.

The **CHECK_FLASH** variable's associated functionality allows for reading the flash memory bytes up-to the size defined by $2^{FIRM\_ADDR\_W}$. It then sends the received file through UART. It can thus serve for debugging purposes.

The **SECTOR_CLEAR** variable's associated functionality allows for erasing a 64KB flash memory sector from address 0. It is required when reprogramming the flash memory.

4. Results

In this section, 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 [2], 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 the FPGA implementation are also presented.

4.1. Flash Core Comparative Results

Comparing the features of the CAST xSPI-MC core presented in [2], 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.

4.2. Performance Results

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.

4.2.1 Experimental Setup

The code running performance is tested against the internal SRAM performance. 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 (10 dummy cycles). The modes are entered from the simple SPI mode.

The 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. The serial clock frequency is 25 MHz for 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();
}
```

4.2.2 Results Analysis

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. Running the serial clock at 25MHz allows decreasing the default number of dummy cycles (8 and 10) to 1. The theoretical fastest mode is fast Read in quad IO in DTR, but it was not used due to only accurately working in simulation.

4.2.3 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.

4.2.4 Results Analysis 2

As expected, running a cache-intensive FFT kernel from flash reports marginal performance degradation (around 2%) 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.

4.3. Fpga Implementation Results

Tables 7 and 8 present the implementation results for the Kintex Ultrascale KU040 board FPGA and the Cyclone V GT FPGA, respectively. The instructions interface is enabled.

Table 7: Xilinx FPGA Resource Utilization Results.

Resource	Utilization
LUTs	565
Registers	519
DSPs	0
BRAM	0

Table 8: 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 [2] and [3] 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 can be 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.

5. Conclusions

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

5.1. 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 is able to perform the basic memory read and memory write operations with acceptable performance. The controller core can be dynamically configured in several operating modes by means of 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 percent degradation in performance is observed compared to running the code from SRAM. Without a cache or with 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 now is able to execute code from a flash device, and use the same device for permanent storage. This is an important feature for IOb-SoC, which now can 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 the flash to save and retrieve data in a permanent basis.

Comparing the core's performance to a commercially available implementation from CAST [2] 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 lanes 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.

5.2. 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 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.

An important 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 future development could be the implementation of a wear leveling mechanism to increase the flash device lifespan.

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.

References

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

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

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

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

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