IP-core Manager for FPGA-based design on SEcube™

Project Documentation

Release: May 2021

Proprietary Notice

Authors

Maurizio DI LORENZO maurizdl@gmail.com
Simone MACHETTI simonemachetti@gmail.com
Alessandro MONACO alessandro.monaco.94@gmail.com
Flavio PONZINA flavio.ponzina@gmail.com
Paolo PRINETTO *(Director, CINI Cybersecurity National Lab)* paolo.prinetto@polito.it
Gianluca ROASCIO *(PhD candidate, Politecnico di Torino)* gianluca.roascio@polito.it
Antonio VARRIALE *(Managing Director, Blu5 Labs Ltd)* av@blu5labs.eu

Trademarks

Disclaimer

Revision History

Release	Date	Changes
005	April 2019	First public release (revisions prior to rel. 005 are not public).
006	October 2019	Minor update: restyling, rephrasing, update of broken links.
007	February 2021	General rephrasing. Removed redundant part of user guide, already present in the general documentation.
008	May 2021	Adaptation to firmware integration of the minimal FPGA library.

Contents

1 Introduction

Among its hardware resources, the **SE***cube*™ offers a powerful and flexible FPGA by Lattice Semiconductor.
The presence of the FPGA inside the **SE***cube*™ Chip is certainly a plus for the programmer's possibilities. The CPU-FPGA system can be seen as a *processor-coprocessor* system, where the programmable logic is entrusted with a series of frequently used routines, among which encryption and hashing certainly stand out, while the control and decision-making part remains to the microcontroller. Besides the advantages brought by this parallelism, there is also the guarantee of reaching the necessary levels of determinism and performance, in the case of time-critical applications. The flexibility the FPGAs dispose of allows to keep up with innovation in communication standards: in fact, converters and controllers can be implemented for external interfaces which may be not natively present on the processor.
From an operational point of view, despite these advantages, an issue immediately arises: how to reuse the FPGA when a new function is required? The FPGA can be a very powerful coprocessing tool, but time and power required for its complete reconfiguration can nullify the benefits it brings. Furthermore, the storage of information to program the entire device is rather expensive, and packing more than one of these long configuration arrays in the firmware is certainly not an optimal solution with respect to a constrained program memory, as is usual in embedded systems.
Hence the idea of driving the **SE***cube*™ FPGA to support more than one IP core at the same time. Such cores implement a set of functions, as fast computing of dedicated algorithms or enabling the device to expand its interfacing capabilities. The aim of the present work is to create a flexible channel between the CPU and a set of IP cores accommodated inside the FPGA of **SE***cube*™ , with the introduction of a *central manager block* to redirect CPU requests to the correct core, and to ensure that each communication with the CPU is exclusive with a single component. Particular attention is given to performance, parallelism and hardware design.
The cooperation between the two actors of the communication is intended to follow these specifications:

- The CPU must communicate with the IP cores through *transactions*, i.e., sets of data exchanged from an initial packet which opens the connection to a final one which closes it
- Transactions are exclusive: the CPU cannot handle a transaction with more than a single IP core at a time
- Inside the FPGA, several IP cores must have the possibility of running and producing results simultaneously
- Each core is not aware of the presence and the status of other cores inside the FPGA
- The typical transactions between the CPU and a core is the following: the CPU opens a transaction with a core and writes on a portion of *dual-side-accessed shared memory* its inputs, which are read and processed by the core that then writes results again on the same block of memory
- The FPGA should be efficiently designed, whatever the number of cores inside
- The cores can be designed in whatever way, but they must present a *standard interface* for the communication
- The cores can be implemented in such a way to support an error-signalling system to the central manager and therefore to the CPU.

2 System Architecture

2.1 Global Architecture

The core of the **SE***cube*™ Hardware device family is a 3D multi-module SoC (System-on-Chip), integrated in a 9mm x 9mm BGA package. The single chip embeds three hardware components: a powerful processor, a flexible FPGA, and an EAL5+ certified smart card. In this section, we want to give an overview of the global system we are dealing with, in order to have clear in mind the communication possibilities between the CPU and the FPGA: for such reason, we are going to present with more details the CPU and the FPGA blocks and their relative interface.

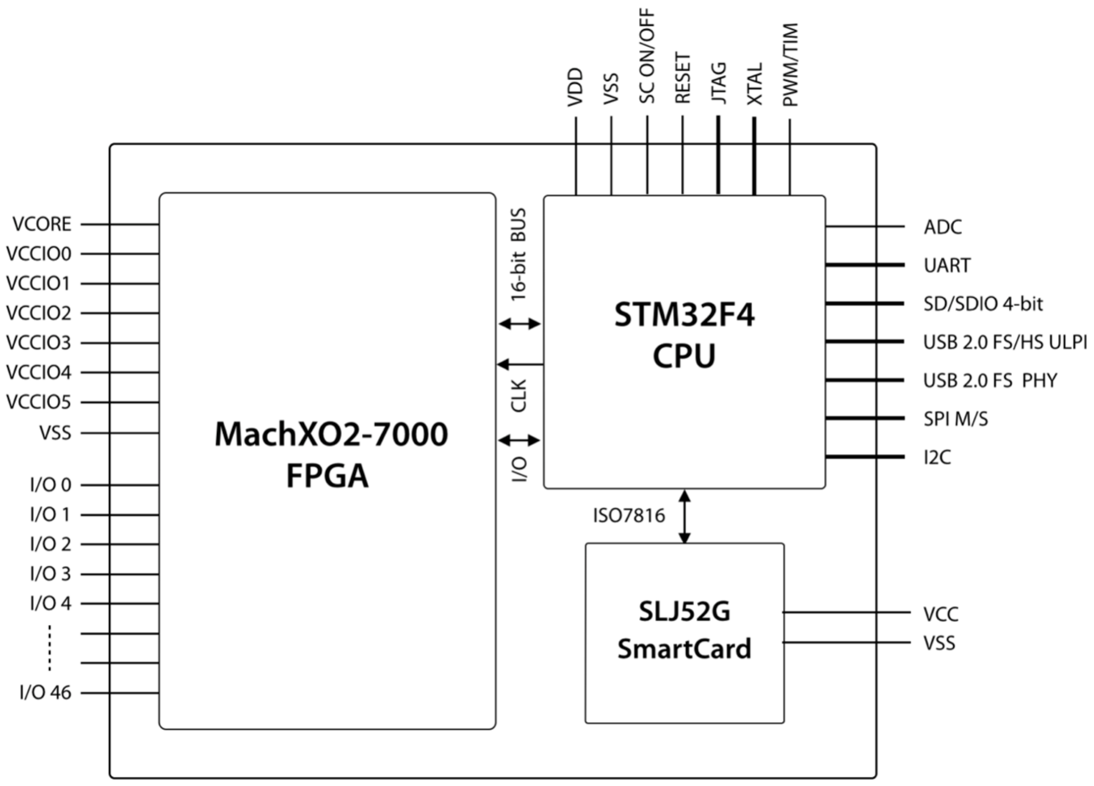


Figure 1: The **SE***cube*™ Hardware Architecture

2.1.1 The Processor

The processor adopted within the **SE***cube*™ is the STM32F429, produced by ST Microelectronics™ [1], which includes a high-performance ARM Cortex M4 RISC core and provides the following features:

- 2 MB of Flash memory
- 256 KB of SRAM
- 32-bit parallelism
- Operating frequency of 180 MHz
- Low power consumption

This CPU has been selected among many ARM-based microcontrollers, since it offers several features that make it suitable for high-performance and security-oriented solutions. For example, it supports the Cortex CMSIS implementation that provides, among the others, the CMSIS-DSP libraries: a collection with over 60 DSP functions for various data types. The CMSIS-DSP library allows developers to implement complex, real time operations using the embedded hardware Floating Point Unit.
In addition, the CPU provides several peripherals such as SPI, UART, USB2.0 and SD/MMC, which ease the hardware integration in diverse devices. For example, a secure USB device can be easily implemented using the USB2.0 and the SD card interfaces, respectively.
On the security side, a TRNG (True Random Noise Generator) embedded unit, hardware mechanisms like MPU (Memory Protection Unit), and privileged execution modes allow implementing the security strategies required by a certified secure controller (e.g., privileged memory areas, key generation, etc.).
For programming, debug, and testing operations, the CPU provides a standard JTAG interface that can be permanently disabled once the development cycle is over, protecting all the sensitive information through a physical hardware lock.

2.1.2 The FPGA

The FPGA element, a Lattice MachXO2-7000 device[2], is based on a fast, non-volatile logic array providing the following main features:

- 7,000 LUTs
- 240 Kbits of embedded block RAM
- 256 Kbits of user Flash memory
- Ultra low-power device

The FPGA exposes 47 general-purpose I/Os which may be used as a 32-bit external bus able to transfer data at 3.2 Gb/s.
As outlined in Figure 1, within the **SE***cube*™ Chip the FPGA is connected to the CPU through a 16-bit internal bus, providing a data transfer rate of up to 1.6 Gb/s.
A CPU-FPGA clock line is provided to simplify the clock domains synchronization.
To limit the number of pins and the BGA package size, the FPGA JTAG is connected just to the

[1] http://www.st.com/en/microcontrollers/stm32f4-series.html?querycriteria=productId=SS1577
[2] http://www.latticesemi.com/view_document?document_id=38834

embedded CPU, which manages both the debug and the programming operations. Therefore, the FPGA configuration can be implemented by means of customized, high-security techniques. For example, the programming bitstream can be encrypted and signed through dedicated algorithms. The CPU and/or the smartcard elements can then be used to decrypt and verify it before its injection into the FPGA.

2.2 FPGA-CPU connection

Here is presented the pin-to-pin interface between the CPU and the FPGA, which represent a fundamental point for our work.
The set of interconnections is used for exchanging data, control, and status signals, including:

- Clock, reset and interrupt signals
- JTAG interface for programming the FPGA
- Other control lines

The configuration within the **SE***cube*™ treats the FPGA as an external memory device (PSRAM), leveraging the Flexible Memory Controller (FMC) available on the processor. The FMC is an interfacing peripheral of the microcontroller used to connect external memories such as NOR Flash, NAND Flash, SRAM, and PSRAM[3], as in this case. With this configuration, each pin assumes a specific behaviour, its value and transitions being directly managed by the FMC.

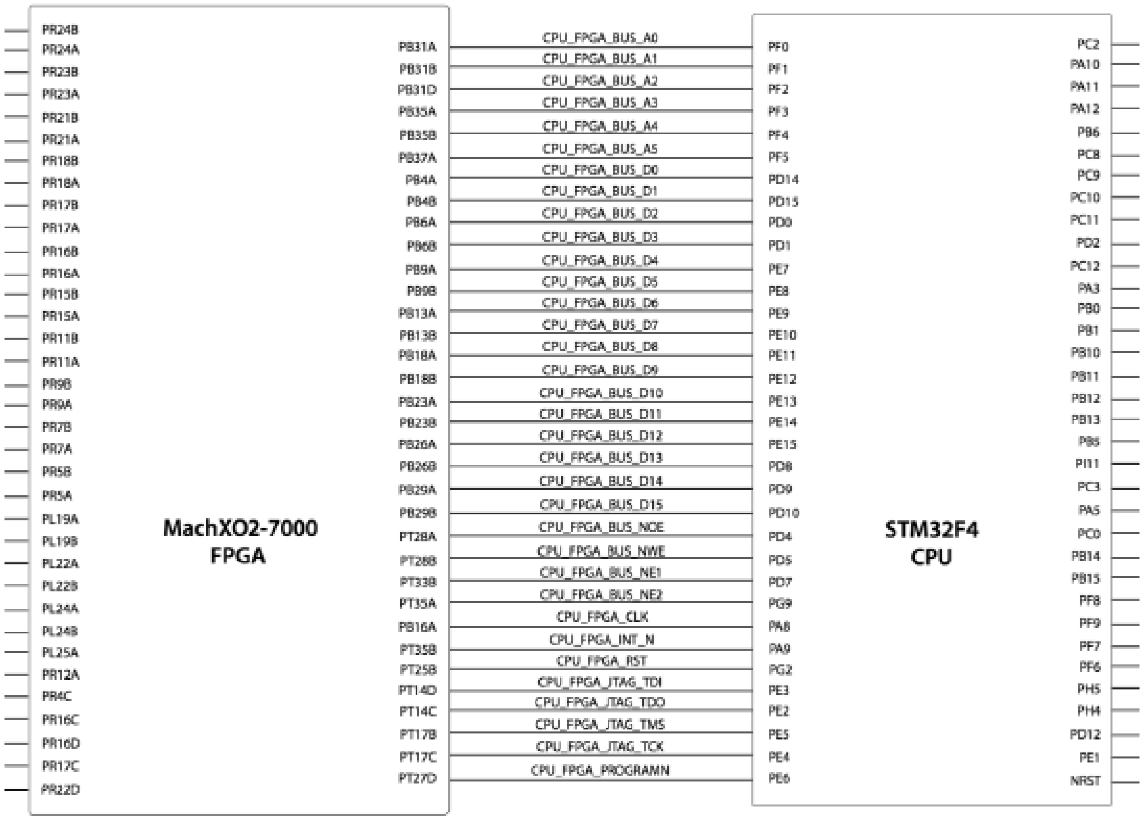


Figure 2: FPGA-CPU pin configuration within **SE***cube*™

[3]For additional details, please refer to STM32F405/415, STM32F407/417, STM32F427/437 and STM32F429/439 advanced Arm-based 32-bit MCUs Reference Manual, Chapter 37: https://www.st.com/resource/en/reference_manual/dm00031020-stm32f405-415-stm32f407-417-stm32f427-437-and-stm32f429-439-advanced-arm-based-32-bit-mcus-stmicroelectronics.pdf

The available pins within the bus are:

- Address – 6 pins (CPU_FPGA_BUS_A0:5)
- Data – 16 pins (CPU_FPGA_BUS_D0:15)
- Chip select – 2 pins (CPU_FPGA_BUS_NE1:2)
- Clock – 1 pin (CPU_FPGA_CLK)
- Controls – 2 pins (CPU_FPGA_{INT_N, RST})
- JTAG – 5 pins (CPU_FPGA_JTAG_{TDI, TDO, TMS, TCK}, CPU_FPGA_PROGRAMN).

2.3 FPGA internal structure

The internal architecture that this project intends to accommodate on the FPGA embeds three main blocks:

- a dual-ported data buffer where inputs and outputs are exchanged;
- the IP cores customized for executing a given task;
- a central IP manager for routing, arbitrating and configuring the communication.

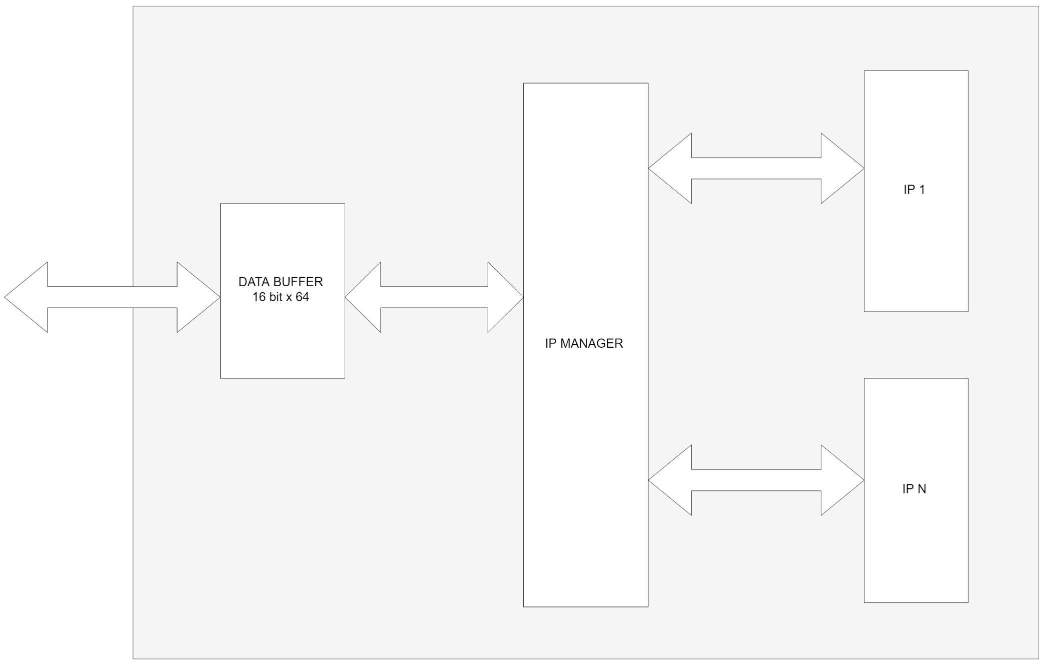


Figure 3: FPGA internal structure

2.3.1 The data buffer

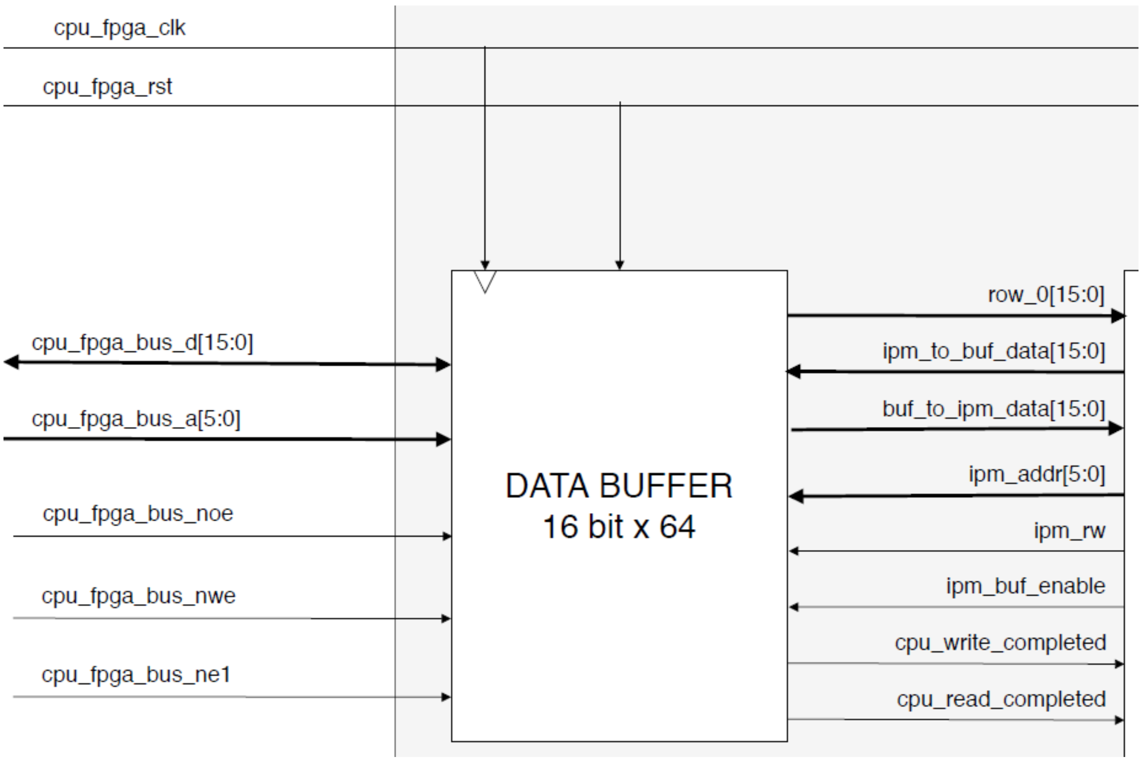


Figure 4: Data buffer schematic

The data buffer represents the memory module interfacing with the CPU, where commands and inputs for the FPGA are placed, and where outputs are stored at the end of the operations. It is a dual-ported portion of 128 B of shared synchronous SRAM, organized in 64 16-bit words. The buffer presents two different interfaces at the two sides through which is accessed. Hence the presence of a conspicuous logic that surrounds and controls the memory array. From the CPU point of view, because of the employ of the FMC, the entire FPGA is seen as a block of PSRAM. Therefore, signals and timings are accorded to this abstraction: besides the common data, address, clock and reset signals, the buffer presents 3 additional specific signals for the FMC protocol:

- the generic enable (NE1)
- the output enable (NOE)
- the write enable (NWE)

Below here, we report values and timing of the interfacing signals, in a timing diagram which shows the behaviour of the protocol followed by the Flexible Memory Controller (FMC) to interface the FPGA for reading and writing[4].

[4]STMicroelectronics, Reference Manual RM0090 for STM32F405/415, STM32F407/417, STM32F427/437 and STM32F429/439 advanced ARM-based 32-bit MCUs, Chapter 37, “Flexible Memory Controller (FMC)”: https://www.st.com/resource/en/reference_manual/dm00031020-stm32f405-415-stm32f407-417-stm32f427-437-and-stm32f429-439-advanced-arm-based-32-bit-mcus-stmicroelectronics.pdf

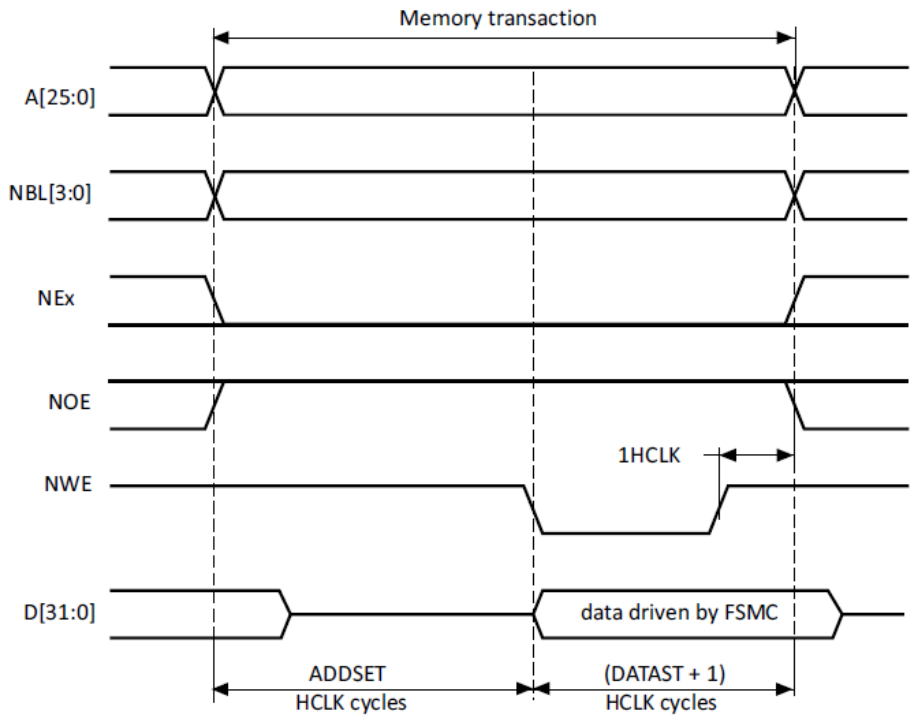


Figure 5: Mode A write access waveforms of the FMC protocol for STM32F4xx MCU

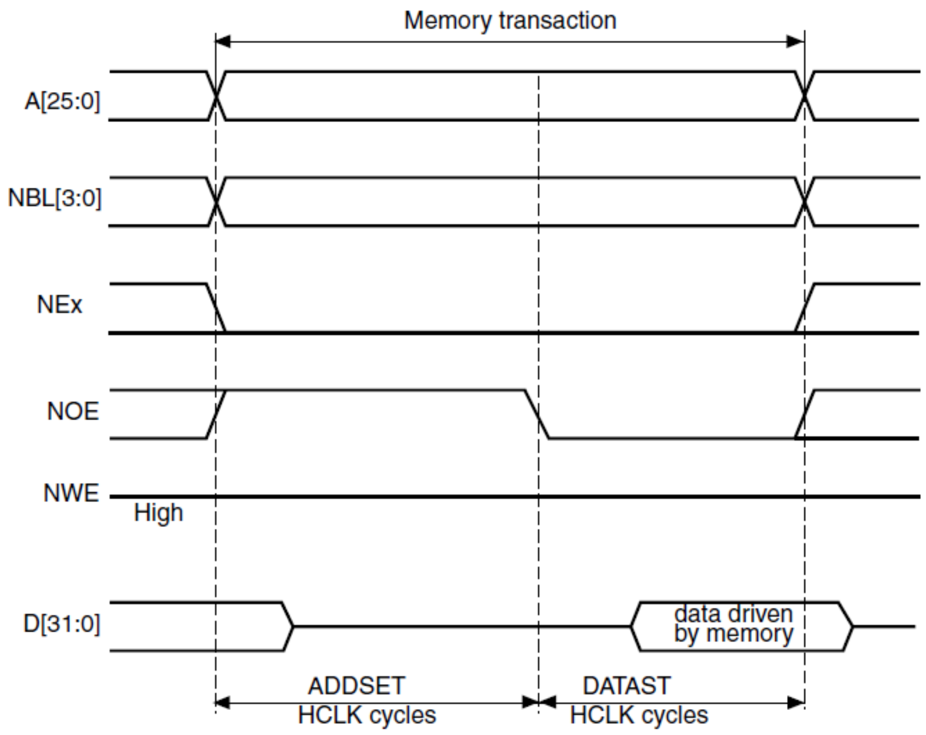


Figure 6: Mode A read access waveforms of the FMC protocol for STM32F4xx MCU

The Flexible Memory Controller is able to manage 4 different banks of memory within a specific address range. Bits [27:26] of the AHB address bus (HADDR) are interpreted by the FMC as the identifier for 1 of the 4 banks, and enable the activation of the corresponding NEx signal (NE1, NE2, NE3 or NE4), which is active low. The CPU-FPGA interconnection then allows to accommodate just 2 of these 4 banks of memory, presenting both NE1 and NE2 pin in the CPU interface.
Bits [25:0] of the HADDR bus are instead interpreted as the actual external memory address, but actually just the lowest 6 of them are physically pinned to the FPGA.
When NEx is low, a memory operation is taking place. The address is forwarded and a setup time for its stabilization is awaited. Then there is the data phase: if the NWE not asserted, the address is intended to be a read address, and the CPU waits for a data response from the memory for a given data setup time. If a write is to be performed, NWE is asserted and a word to be written is forwarded and maintained for the data setup time. These setup times are decided by the software configuration of the FMC and must be provided to the FPGA design as generic VHDL parameters. The machine within the buffer recognizes when the FMC enters the address setup time and the data setup time looking at the values of the interface signals, and once understood sets an internal *decrementing counter* with the generic parameter values and waits for the end of the different phases before acquiring a valid address and a valid data, or before providing it.
At the end of any CPU operations, the internal machine is able to send to the right-side interface two *strobes signal*, which indicate the completion of a read of or a write, as it is possible to see in Figure 4. These signals are useful to synchronize the activity of the internal modules, as we will see.
From the internal side, the buffer is accessed in a simpler way as a standard bank of synchronous fast memory, with an enable signal, two data busses for input and output, an address bus and a read/write signal. These signals are in control of the module which is enabled to communicate by the Manager.
The first row of the buffer is reserved to the *command word* for the Manager, as we will see in the following chapters. Since this word is to be continuously monitored by the Manager to sense any CPU command, the row is directly reflected to an output port.

2.3.2 The IP cores

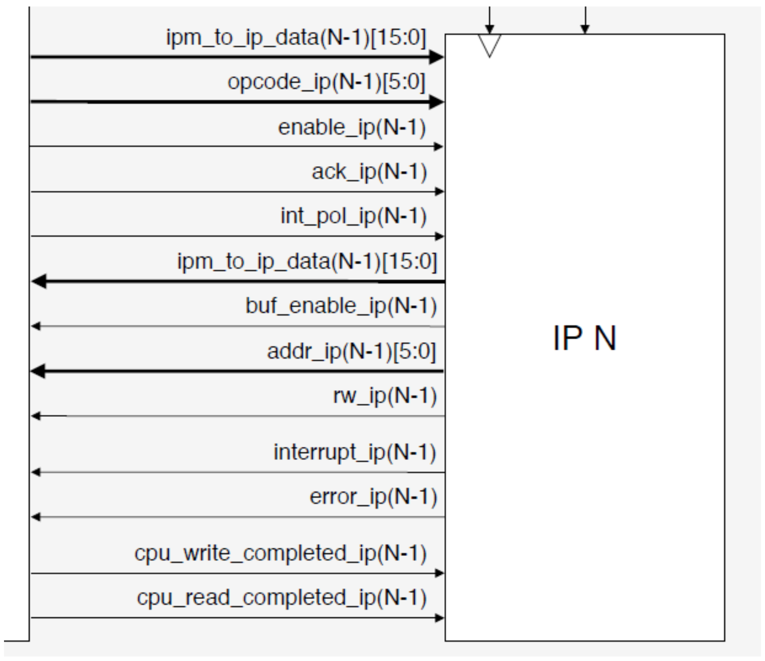


Figure 7: Example of IP core with its interface

The IP cores are the target modules of the CPU-FPGA communications, as they represent the hardware accelerators needed by the CPU to enhance speed in executing some dedicate algorithms and tasks. The designer is thus free to internally design the core as preferred, but must present at least the following mandatory ports (Figure 7):

- Clock and reset input signals
- An enable signal as starting strobe for the internal machine
- A dedicate 6-bit signal for the possible operative code
- A single-bit input for the communication mode (interrupt/polling)
- 2 separate 16-bit buses for data input and data output
- 6-bit address bus
- 2 command signals for the data buffer (enable, read/write)
- 2 dedicate interrupt and error output lines
- An acknowledgment (ACK) input line
- CPU read and write completion input strobes

A core remains inactive until the enable signal coming from the Manager is asserted: this must be the event that triggers its internal state machine. The enable signal ensures the core that the internal-side signals of the buffer corresponds one-to-one to its memory interfacing signals: the buffer answers now to its commands. Therefore, the IP core can start a pipelined communication with the master thank to the strobe signals which indicate the end of a read or of a write operation by the CPU.

The core can be designed to support more than one operation, and both *polling* and *interrupt* communication mode: the mode and the possible operative code are written by the CPU in the command word of start transaction and are forwarded to the module at the enable time by the manager through two dedicated signals.
When in polling mode, the CPU does not close the transaction until the output are returned on the buffer. It just awaits the completion of the computation polling a dedicate address or waiting for some predefined time necessary to the core to end its job.
When in interrupt mode, the core and the CPU have an input-transmitting transaction first, then the communication is closed, and the core starts computing. When done, the core asks to the manager the interruption of the CPU. When it is acknowledged, the core has a second transaction during which the outputs are written on the shared memory.
The system also provides support for error signalling from the IP cores, e.g., after an internal failure detected by a built-in self-test procedure. A dedicate error signal is present in the interface to advise the Manager, and consequently the CPU, of the problem.

2.4 The IP Manager

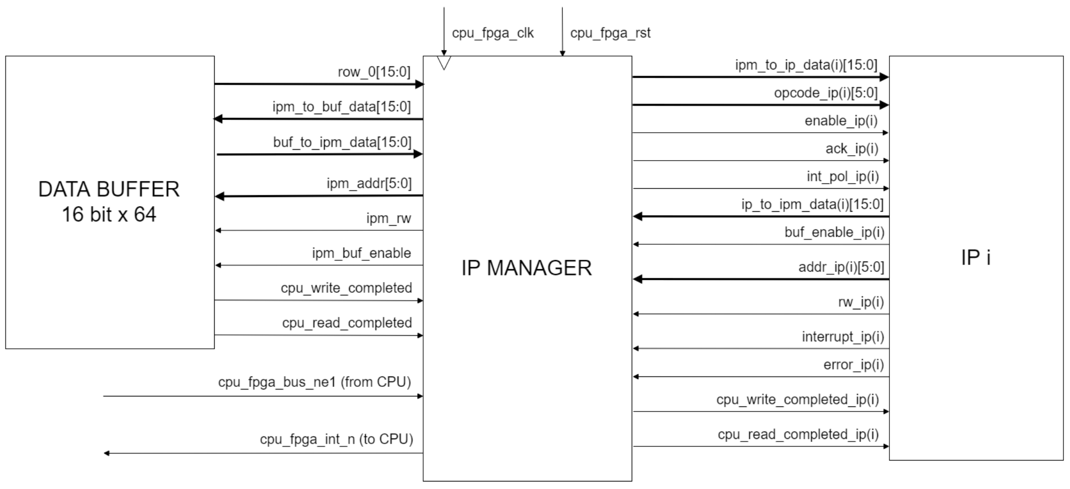


Figure 8: The IP Manager connections

The IP Manager is the central main block of the architecture. It is the intelligent multiplexer/de-multiplexer of the system. Its main intent is to put in communication the IP core addressed by the CPU with the buffer where data are exchanged. The IP Manager/IP core interface is replicated for each core placed in the design.
After the global reset, the module is in its IDLE state and waits for a request from the CPU. This is forwarded under the form of a control word written in the first row of the buffer, row which is immediately reflected towards the Manager not to lose time for addressing it before reading. At each clock cycle during the IDLE state, the IP Manager monitors this signal, detecting at some point the word of start transaction. As we will see in the next sections, this word contains the ID number of the IP core desired by the CPU plus an operative code and other configuration bits. One of them indicates the begin of a transaction. The IP Manager then enters in its MULTIPLEXING state, putting into direct contact the signals on its IP interface with the ones on its buffer interface, asserting the enable signal to the correct core, and forwarding to the corresponding interface the operative code and the communication mode (interrupt/polling). The Manager detaches this link only when the word of end transaction is written at row 0 of the buffer, disabling the core and taking back the control of the signals to the buffer. During the MULTIPLEXING state, the module

is blind to any other request from the cores not involved in the communication and even to CPU commands different from the end of transaction one, such that the exclusivity of the transactions is preserved.
Once exited the MULTIPLEXING state, the Manager returns IDLE. When IDLE, it is sensitive to any interrupt or error request coming from the IP cores. If the interrupt line of one IP core is high, the Manager raises the global interrupt signal of the FPGA towards the CPU, writing at address 0x00 of the buffer the ID of the core that desires to interrupt the CPU. The CPU, after having read it, possibly sends a control word of acknowledge for that request, which triggers the raising of the corresponding ACK line of the Manager/core interface along with the enable. An acknowledgment transaction thus begins, and the core probably communicates its results to the CPU and the Manager is in its MULTIPLEXING state, from which exits when the end of acknowledgment transaction command arrives, as usual. The CPU can also decide of not caring the interrupt from the FPGA, starting after some time another transaction without acknowledging the request. If this is the decision, of course, the CPU loses the right to know the ID of the interrupting IP and its results.
If an error line is raised by a core, the Manager forwards an interrupt request from the CPU by itself, i.e., writing at address 0x00 its own ID, which is 0. An acknowledge for IP 0 from the CPU equals the beginning of a transaction during which the Manager itself communicates on the buffer (at address 0x01) the ID of the IP which has signalled an internal fault. The faulty IP is not involved in this exchange of information as in the interrupt case, and its error line remains up until the problem is persistent. This cause no problem to the Manager, that once acknowledged by the CPU becomes blind to that line until its next rising edge.
The CPU sees the Manager as an actor of the communication in this case and in this case only. It is not possible, for the CPU, to start a transaction with the IP with ID 0 outside this situation, because that ID is normally considered out of range and, for this reason, ignored.
Although, thank to this ID reservation, future updates of the project could include a Manager that, for example, handles a transaction with the CPU to dynamically set interrupt priority and mask.

3 Communication Protocol

3.1 Overview

The following section is intended to describe the communication protocol that has been thought for the project. It is necessary to keep in mind that the goal is to allow a flexible communication channel between the CPU and the FPGA. The direct connection between these two entities seems to suggest that there is no need of a dedicated protocol. But here are more cores accommodated within the design. With a single core, the communication would be very straightforward, since the two sides would implement a point-to-point communication. In the common foreseen scenario, the CPU programs more than a single core, but the actual interface only allows the processor to read and write the buffer of the FPGA. Hence the need of an additional entity (the IP Manager) which has the aim of controlling the information exchange, and both the CPU and the cores need to follow a given protocol for their communications.
To manage exclusivity of communication between a single core at a time and the CPU, such protocol is based on *transactions*: the CPU must send a specific data packet to open or to close a transaction, which addresses one and one only IP. Once the transaction is open, the CPU and the selected core can perform the exchange of information. At the end, the CPU must send a packet to close the transaction.
As already mentioned before, the communication can be held in polling or in interrupt mode.

3.1.1 Polling

Polling is a very common way of communication between a master of a computing system and a slave such as a peripheral or a coprocessor. The master continuously checks the status of the slave to monitor its state (which can be “something to communicate”, or “nothing to communicate”). The monitoring is done classically with a continuous read of the status on a shared location, a portion of a memory or a control register.
In our case, the CPU can decide to communicate with a core in polling mode when the core computation is thought to take a short time and can be waited idly without impactive loss of performance. Inputs are then sent to the core, and the transaction is maintained opened until the results are ready.
When the communication is handled in this way, after the opening of the transaction, an handshaking phase with delivering of the inputs follows. Such inputs can be read by the core as soon as they are written thanks to the write strobe sent by the buffer. The core, once read the last input, starts working, while the CPU sets the polling location to a *locking value*, and keeps reading it. When the core is done, it writes the outputs in the buffer without waiting any enabling signal, since the transaction is open and it has rights to access the buffer. As last write, it writes the unlocking code in the polling location. The CPU is thus unlocked and starts reading the outputs, then closes the transaction. The driver of the core must be obviously aware of (i) the location in the buffer of the results, (ii) of the word to be polled and (iii) of the value to expect for the unlocking.

3.1.2 Interrupt

Interrupt is a good alternative to polling when working in a multitask environment. In this scenario, the master first programs the slave and then continues its own computation, while the slave reads the input, makes its job and at the end triggers an interrupt request to the master. This is typically the most common way of communication in modern systems, because it offers higher performance with respect of polling, because of the possibility for processor and accelerators of working in parallel. The hardware overhead consists of a single additional line for each slave (interrupt line).

In the IP-Manager-based architecture, the cores do not have a dedicate interrupt line directly towards the CPU, but the requests are all intermediated by the central IP Manager.
When the communication is held in interrupt mode, the CPU opens the transaction and writes the input as usual, while the core is enabled by the Manager and reads them in parallel as soon as they are written. Once written the last input, the CPU closes the transaction, disabling the core, which probably has already started its computation once read the last input. When finished, the core raises its interrupt line transmitting the request to the Manager first, then the Manager writes at location 0x00 the ID of the core and forwards its request to the interrupt output line of the FPGA. The CPU receives the interruption and, sooner or later, may decide to respond. First, it reads location 0x00 where it retrieves the ID of the interrupting IP, then opens an acknowledgment transaction with this IP. At this point, the interrupt lines are cleared, and the exchange of the output on the buffer starts. The CPU has the only limitation of leaving some time to the core to complete this write-out. Once completed this phase, the CPU closes the transaction.

3.2 The control word

The control word is the fundamental opening and closing word of any CPU-FPGA transaction. It is located at address 0x00 of the data buffer, which is immediately reflected to the IP Manager through a dedicate port. The structure of the control word is reported in Figure 9.

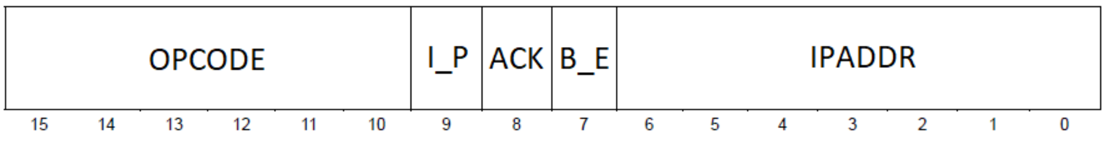


Figure 9: Bit fields of the control word

Bits 15:10	**OPCODE**	Operative Code	Hosts the possible operative code instructing the core on the task to be performed.
Bit 9	**I_P**	Communication Mode	0 : Polling Mode 1 : Interrupt Mode
Bit 8	**ACK**	Acknowledgment Transaction	0 : The transaction opened/-closed is a normal transaction 1 : The transaction opened/-closed acknowledges an interrupt request
Bit 7	**B_E**	Begin/End of transaction	0 : End Transaction 1 : Begin Transaction
Bits 6:0	**IPADDR**	Address of the IP core	Hosts the IP core identifier. Can assume any value from 0 to 127.

Whenever the Manager is IDLE and senses setting of bit 7, it enables the core specified by the field **IPADDR** and reflects on its interface the other fields. When bit 7 is cleared by the CPU, all the signals towards the cores are cleared consequently.
As will be presented in the following sections, a set of APIs has been developed to manage the communication via software, including also functions for opening and closing a transaction with given parameters that avoid the low-level bit-by-bit set of the control word.
In this format, the control word is always set and cleared via software. However, the location

0x00 can be used in a different format from the inside, when the IP Manager uses it to write the identifier of the core that requests to interrupt the CPU. In that case, bits 15 to 7 are cleared by the IP Manager.

0	0	0	0	0	0	0	0	0	IPADDR						
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Figure 10: Bit fields of the control word set via hardware during interrupt

3.3 Sequence diagrams

This section is intended to provide sequence diagrams of communication modes between the CPU and the cores.
Typical writing and reading operations are presented in Figures 11 and 12. As already mentioned, the communication is “pipelined”, in the sense that there is no need of waiting that an entire block of data has been written/read before making the following read/write operation. This is achieved through the write strobe asserted by the FSM present in the data buffer.

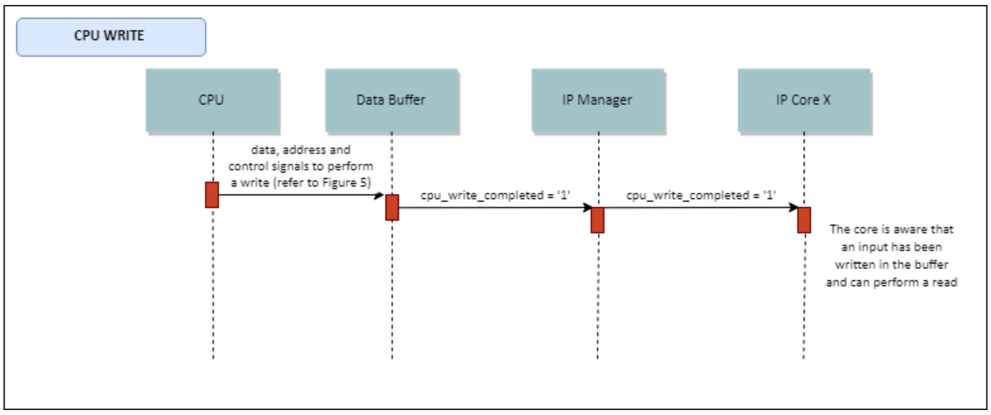

CPU WRITE
CPU
Data Buffer
IP Manager
IP Core X
data, address and control signals to perform a write (refer to Figure 5)
cpu_write_completed = '1'
cpu_write_completed = '1'
The core is aware that an input has been written in the buffer and can perform a read

Figure 11: Typical CPU write operation during a transaction with an IP core

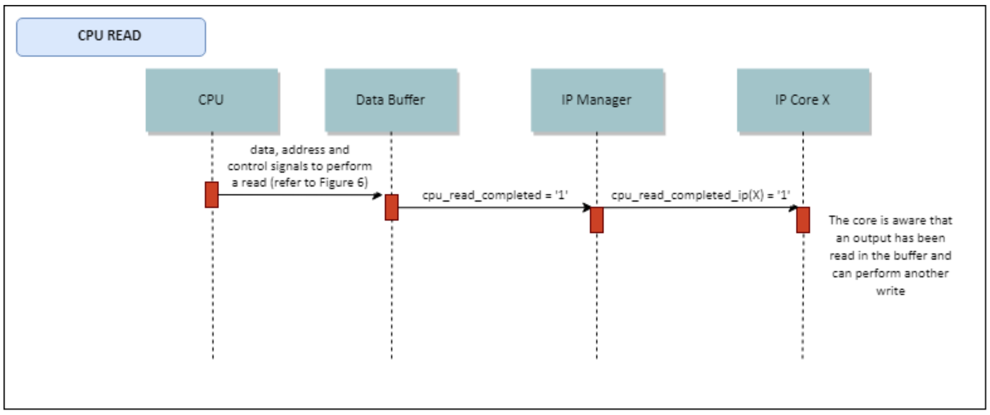

CPU READ
CPU
Data Buffer
IP Manager
IP Core X
data, address and control signals to perform a read (refer to Figure 6)
cpu_read_completed = '1'
cpu_read_completed_ip(X) = '1'
The core is aware that an output has been read in the buffer and can perform another write

Figure 12: Typical CPU read operation during a transaction with an IP core

The typical polling and interrupt transactions between the CPU and the FPGA are presented in Figures 13 and 14. In Figure 15, a typical IP error notification is outlined.

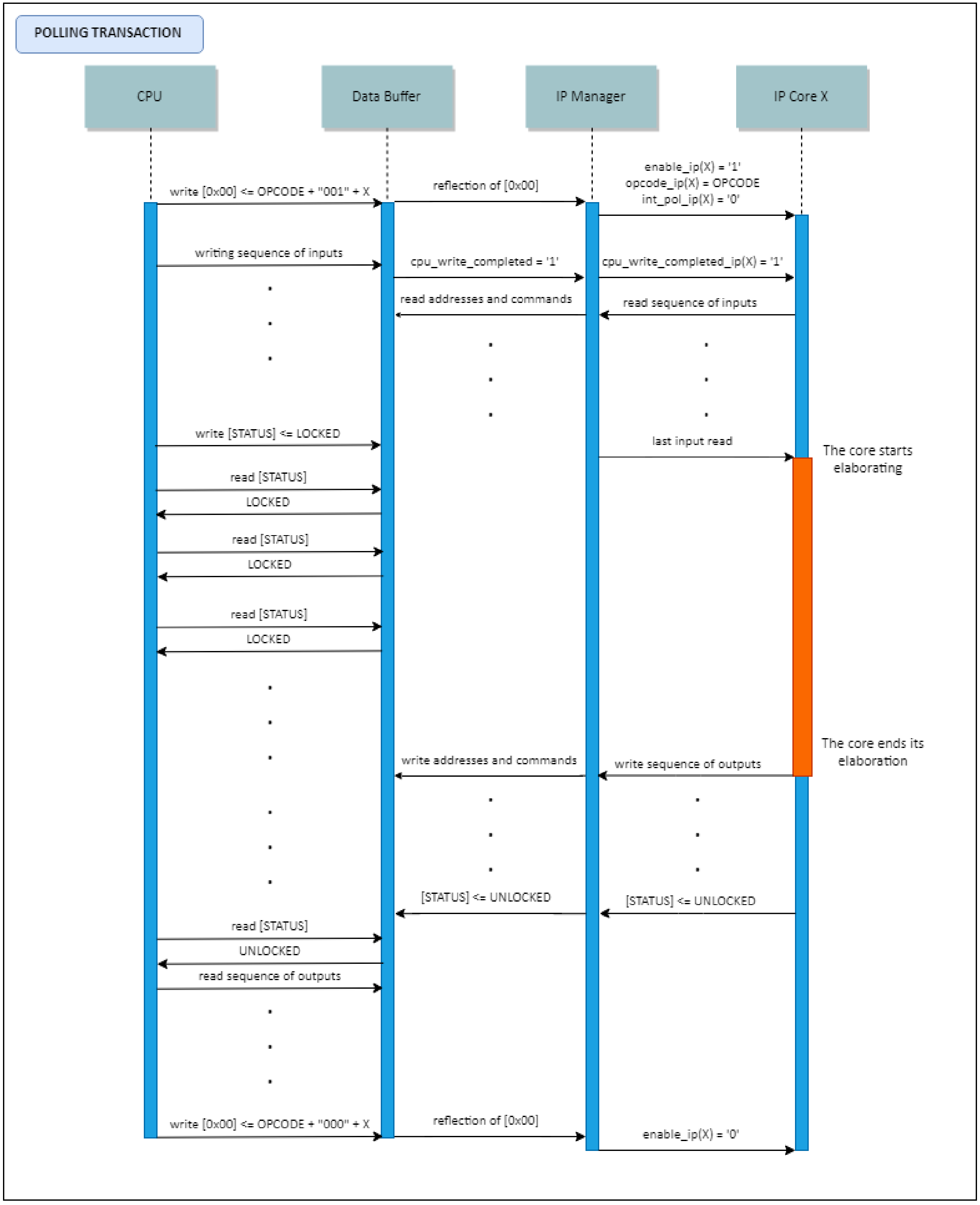


Figure 13: Polling transaction between the CPU and a core

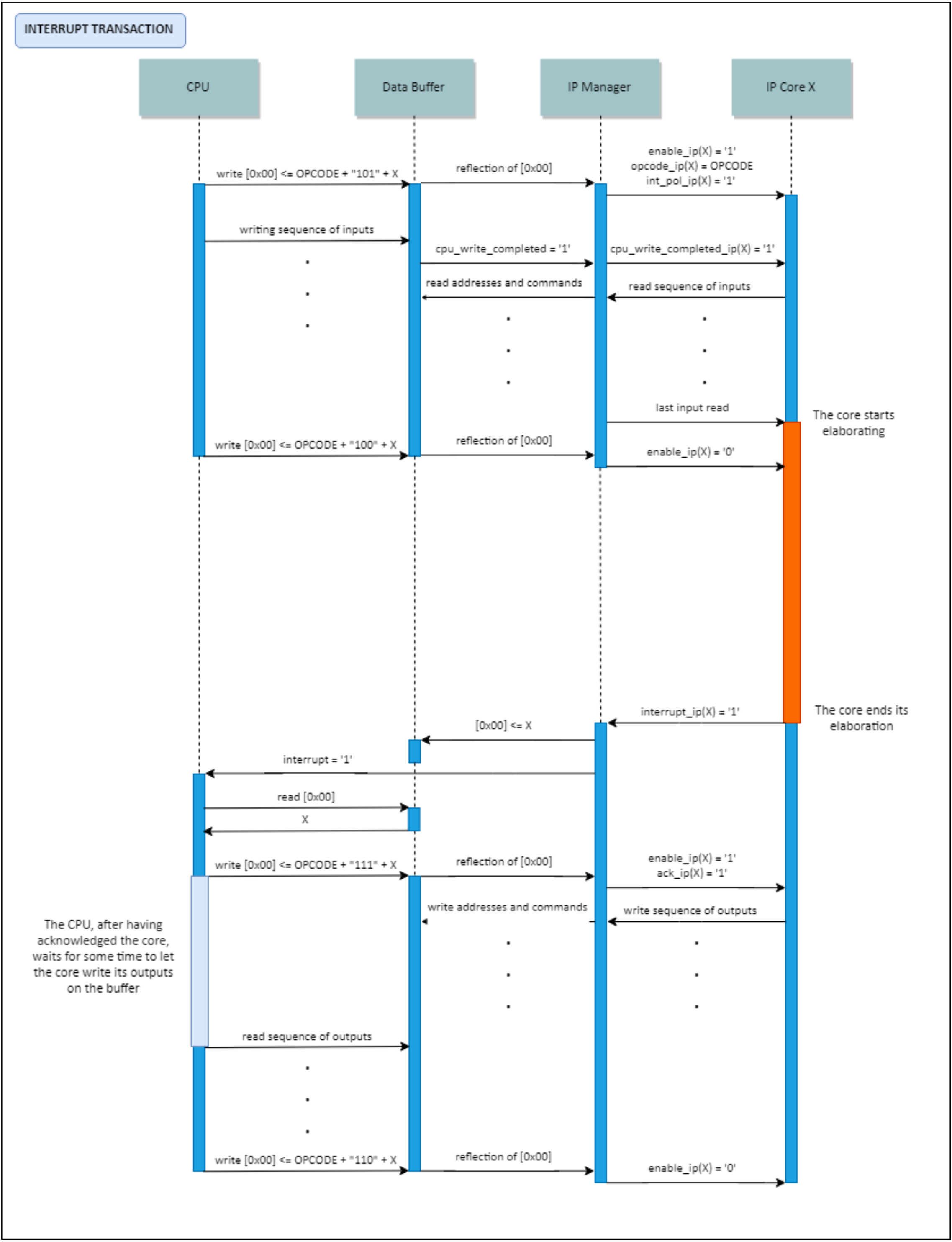

INTERRUPT TRANSACTION
CPU
Data Buffer
IP Manager
IP Core X
write [0x00] <= OPCODE + "101" + X
reflection of [0x00]
enable_ip(X) = '1'
opcode_ip(X) = OPCODE
int_pol_ip(X) = '1'
writing sequence of inputs
cpu_write_completed = '1'
cpu_write_completed_ip(X) = '1'
read addresses and commands
read sequence of inputs
last input read
The core starts elaborating
write [0x00] <= OPCODE + "100" + X
reflection of [0x00]
enable_ip(X) = '0'
interrupt_ip(X) = '1'
The core ends its elaboration
[0x00] <= X
interrupt = '1'
read [0x00]
X
write [0x00] <= OPCODE + "111" + X
reflection of [0x00]
enable_ip(X) = '1'
ack_ip(X) = '1'
write addresses and commands
write sequence of outputs
The CPU, after having acknowledged the core, waits for some time to let the core write its outputs on the buffer
read sequence of outputs
write [0x00] <= OPCODE + "110" + X
reflection of [0x00]
enable_ip(X) = '0'

Figure 14: Interrupt transaction between the CPU and a core

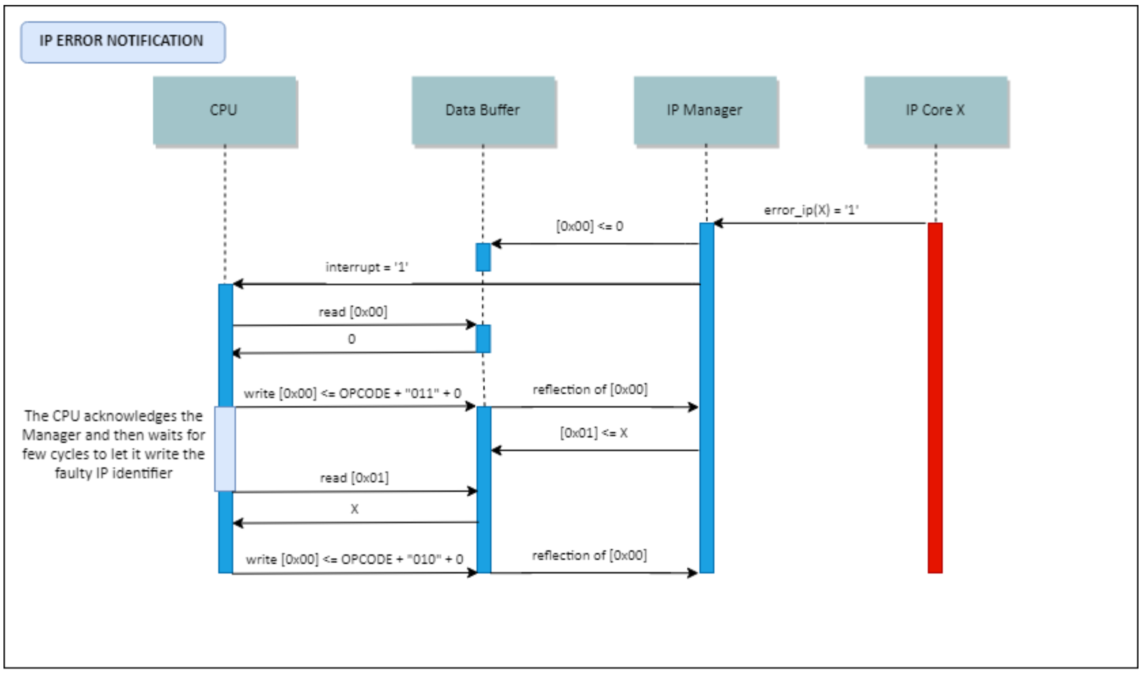


Figure 15: An IP core signals an internal error

4 Driver

This Section describes the communication software driver developed for interfacing the FPGA and the IP cores within it.
For a correct communication, the complete driver should be composed of two layers:

- a high-level layer, composed of the specific functions for managing the task of each IP core;
- a low-level layer, containing the low-level functionalities for the communication with the FPGA, as opening and closing the transaction, or reading and writing a word on the buffer, but also managing the locking of the hardware resource.

Our work focused on providing the driver programmer a reliable low-level layer, over which it is possible to implement a functionality layer to interact with the cores.

4.1 Low-level APIs

The low-level communication with the FPGA is handled by the functions declared in the header file "Fpgaipm.h", included in the released files of this project. Such functions are supposed to be called by the high-level driver code, without the need of any knowledge of the specific hardware implementations and of the details about the microcontroller and the FPGA provided by **SE*cube*™**
.

```
FPGA_IPM_BOOLEAN FPGA_IPM_open(FPGA_IPM_CORE coreID,
   FPGA_IPM_OPCODE opcode, FPGA_IPM_BOOLEAN interruptMode,
   FPGA_IPM_BOOLEAN ack)
```

This function allows to open a transaction with a core on the FPGA, writing at address 0x00 of the buffer the control word with the specified parameters.
The following parameters are required:

- `coreID`: the ID of the required core that will be used. The ID 0 refers to the IP Manager itself
- `opcode`: the possible operative code. Depends on the used core
- `interruptMode`: a boolean indicating the type of transaction: polling (0) or interrupt (1)
- `ack`: a boolean that is positive when a transaction is answering to the interrupt request from a core.

Opening a transaction is possible only if there are no active transactions. Any other transaction is blocked by the software semaphore. The procedure to open a transaction follows these steps:

1. Check if the transaction can be established: if not, the request is rejected
2. Lock the resource (FPGA) by decrementing the semaphore
3. Update control variables to the new values
4. Perform a write operation at address `0x00` of the buffer
5. Send a positive response to the calling function if everything went fine.

```
FPGA_IPM_BOOLEAN FPGA_IPM_read(FPGA_IPM_CORE coreID,
   FPGA_IPM_ADDRESS address, FPGA_IPM_DATA *dataPtr)
```

This function is used to read a word from the FPGA buffer.
The following parameters are required:

- `coreID`: this parameter permits to check if the request is compliant with the current transaction (i.e., the active core ID is the same). If there are no transactions opened, the function returns a negative value
- `address`: the row of the data buffer from which data is read. The address must be in the range from 0x01 (1) to 0x3F (63)
- `dataPtr`: the main memory pointer where the read word is saved.

The function returns a boolean that notifies if the operation was correctly performed.
This procedure is implemented using internally the native function `HAL_SRAM_Read_16b()`, which is declared in the library file “stm32f4xx_hal_sram.h”. The function only reads a piece of SRAM organized in 2-byte words. The automatic interaction with the FMC is possible thank to the FMC initial settings.

```
FPGA_IPM_BOOLEAN FPGA_IPM_write(FPGA_IPM_CORE coreID,
   FPGA_IPM_ADDRESS address, FPGA_IPM_DATA *dataPtr)
```

This function is used to write a word into the FPGA buffer.
The required parameters are:

- `coreID`: as the previous function, this parameter is used to check whether the request can be performed in base of the presence of an open transaction or not
- `address`: the memory offset of the FPGA data buffer in which the word is intended to be stored. Must be in the range from `0x01` (1) to `0x3F` (63)
- `dataPtr`: the main memory pointer that contains the word that is going to be stored in the FPGA data buffer.

The function returns a value that notifies whether the operation was correctly performed or not.

```
FPGA_IPM_BOOLEAN FPGA_IPM_close(FPGA_IPM_CORE coreID)
```

This function is used to close a transaction with the FPGA by just complementing the B_E bit in the control word written at address `0x00`. The only parameter that has to be specified is the corresponding IP core ID, that is used to verify if the closing request is compliant with the previous actions.
Closing a transaction releases the FPGA resources for the transactions.

4.2 FPGA interrupt handler

The firmware call `se3_FPGA_Init()` of the library “se3_fpga.h”, to be inserted to work with the FPGA of the **SE***cube*™ , also inserts in the interrupt sensitivity list of the processor the pin PA9, which is attached to the global interrupt line of the FPGA. The function `void EXTI9_5_IRQHandler (void)` present in “Fpgaipm.c” is the interrupt service routine automatically called when the FPGA interrupts the CPU. This function overrides the `__weak` function declared in “se3_fpga.c”.
Once entered, the CPU effectively controls if the interrupt flag is set, then performs its operations and finally clears such flag. The body of this routine is to be customized depending on the cores present in the FPGA, as it is discussed in Chapter 6.

4.3 Resource lock issues

The FPGA physical resource, when in use, must correctly appear as such to the software layers, to avoid errors coming from inaccuracies in programming or from modules that enter in action in a non-deterministic way over time, in case an embedded OS solution is adopted for **SE***cube*™ . The issue is solved through the implementation of a semaphore inside the driver that allows the execution of one and one only transaction at the time. The semaphore is managed by:

- the `static` variable `FPGA_IPM_SEM sem`, initialized at 0
- `FPGA_IPM_open()`, that checks the value of the semaphore. In case the resource is unlocked, the function zeroes the semaphore and allows the beginning of the current transaction. Otherwise, the function immediately returns with an error
- `FPGA_IPM_close()`, that increments the semaphore releasing the resource if and only if there is an active transaction and the caller of the function is the caller that has opened the active transaction.

5 User Manual

This Chapter is intended to explain you how to build your own application to be run on **SE***cube*™ DevKit exploiting the communication between the CPU and the possible FPGA hardware accelerators with a step-by-step description of the required actions. A complete project involves the customization of both sides: by the microcontroller side, an opportune firmware must be written, that makes use of a combination of the STMicroelectronics™ standard APIs and of the dedicate APIs for interacting with the customized hardware; by the FPGA side, the HDL description of the IP cores must be deployed and synthetized, paying attention to the respect of the protocol and of the physical interface restrictions already described.

5.1 The SE*cube*™ System Setup

The set of both hardware and software resources you need to set up the **SE***cube*™ DevKit, as well as the step-by-step explanation on how to install and connect the different component can be found in the general project documentation, at https://www.secube.eu/resources/open-sources-sdk/, inside latest version of the SDK, in the **wiki** folder.
At the end of the Section 9 of that document, you will have acquired a clear overview of the prerequisites to set up the environment and to create a Device-Side project using one of the supported IDEs (Eclipse, ARM Keil, STM32CubeIDE).

5.1.1 FPGA_LED (Device Side)

The procedure shown in this paragraph guides you to a first example of how to use the Open Source Libraries with the FPGA; it programs the FPGA embedded in the **SE***cube*™ chip to make the LED print a custom byte.
Hereby we list a step-by-step guide to run this program:

1. Import the project as described in Section **??**
2. Edit the code in "main.c" file, adding the following include at the beginning:
```
#include "se3_fpga.h"
```
3. Be sure that functions `MX_GPIO_Init()` and `MX_FMC_Init()` are inserted in the `main()`
4. After the already present initialization functions, in the `main()` add a call to function `se3_FPGA_Init(RCC_MCODIV_2)`. This is used to program the FPGA and feed it with a 90 MHz clock input[5]
5. In the `main()` body, after the `device_init()` call, be sure that the FPGA is brought to a defined state through the `se3_FPGA_Reset()` call
6. From here on, before the `device_loop()` call, edit the function `main()` by inserting as many `se3_FPGA_Write()` that you prefer. Remember to insert a given delay between calls through `HAL_Delay()` to better visualize[6]
7. Save the changes to all files and build the project
8. Connect the DevKit as described in previous section

[5]The example design is encoded into a bitstream in the "se3_fpga_bitstream.h" source file of the firmware, and comes from a Lattice Diamond project fromed by the files located in the "/FPGA" folder of the firmware sources: IP_BLINKER.

[6]For the sake of this project, the `address` parameter of the function `se3_FPGA_Write()` is completely useless: the design is not address-dependent.

9. Run the project following the IDE-specific steps that are described in Section **??**

At the end of the programming, you should notice all LEDs turning on with a blue low-intensity light. This indicates that the FPGA is under programming. The programming process might require some time (1-2 minutes) to be completed.
After that, the LEDs are switched off, and then, depending on the code, the lower 8 bits passed through `*dataPtr` are printed on the LEDs, i.e., bits to 0 are translated into an off LED and bits to 1 are translated to an on LED. The on state is characterized by a strong blue light.

5.2 How to import your own project

At this point, you should have understood how the CPU-FPGA communication system should work. Apart from the minimal library for the FPGA (composed by the files "se3_fpga.h" and "se3_fpga_bitstream.h" included in "Project/Inc" and by the file "se3_fpga.c" in "Project/Src") and the correct GPIO settings for the FPGA, you need a call to the FPGA initialization function in the `main()`. What is to be substituted are the two huge byte arrays present in the file "se3_fpga_bitstream.h", containing the information about the pin interface and for programming internally the FPGA through the JTAG. Such file is generated automatically from your own HDL description of the FPGA by the Lattice Diamond® Deployment Tool after the synthesis steps. What you should do is nothing else than replace within the file these two arrays with the ones generated by this tool, but this will be explained in detail the following.
Another important setup that you must do in order to interface the FPGA design described by this project is the import of our API library with the general managing class we created.

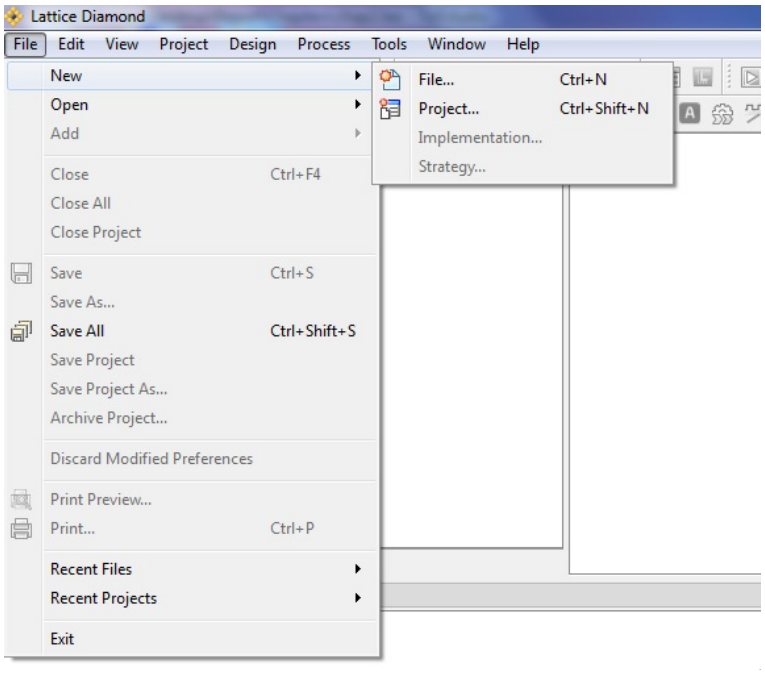
5.2.1 HDL project

Hereby are listed the instructions you need to follow to setup your own set of IP cores and to import them within the project.
First, you need to download the HDL source files for the data buffer, the IP Manager, the package containing the constants and the top-entity structural description of the FPGA available in this project folder. For a project that works as specified in this document, these sources must be altered only in minimal part as it will be indicated, while all the rest of the code must be maintained untouched. The group of available files also comprehends an example core performing SHA256 algorithm and a relative generic testbench usable for simulation purposes, but this will be explained in the last Section.

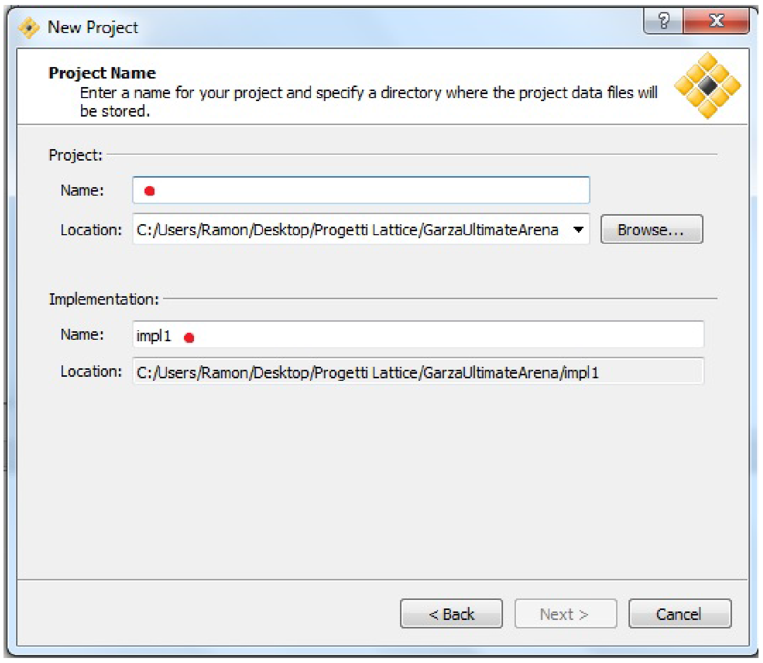
5.2.2 How to create a Lattice Project

Once your IP cores have been developed, tested and validated as working with whatever simulation tool you prefer, the synthesis process must begin. Open Lattice Diamond® and follow these steps.

1. Create a new project



2. Browse to the location of your HDL project folder and insert an implementation name

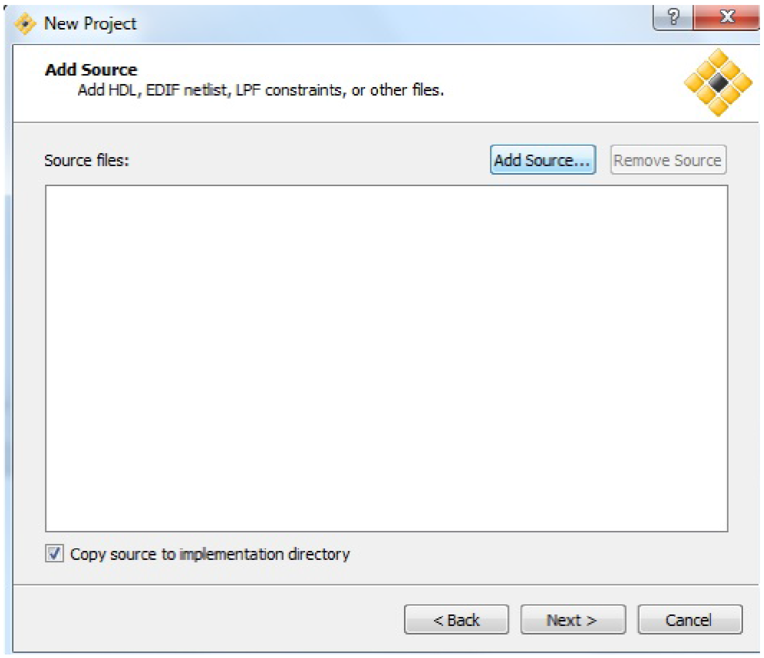

New Project

Project Name
Enter a name for your project and specify a directory where the project data files will be stored.

Project:
Name:
Location: C:/Users/Ramon/Desktop/Progetti Lattice/GarzaUltimateArena

Implementation:
Name: impl1
Location: C:/Users/Ramon/Desktop/Progetti Lattice/GarzaUltimateArena/impl1

3. Add VHDL source files in this step or inside the project by right clicking on the input files folder. Remember that files named “CONSTANTS.vhd”, “DATA_BUFFER.vhd”, “IP_MANAGER.vhd”, “TOP_ENTITY.vhd” must be included, plus your own custom IP cores source files

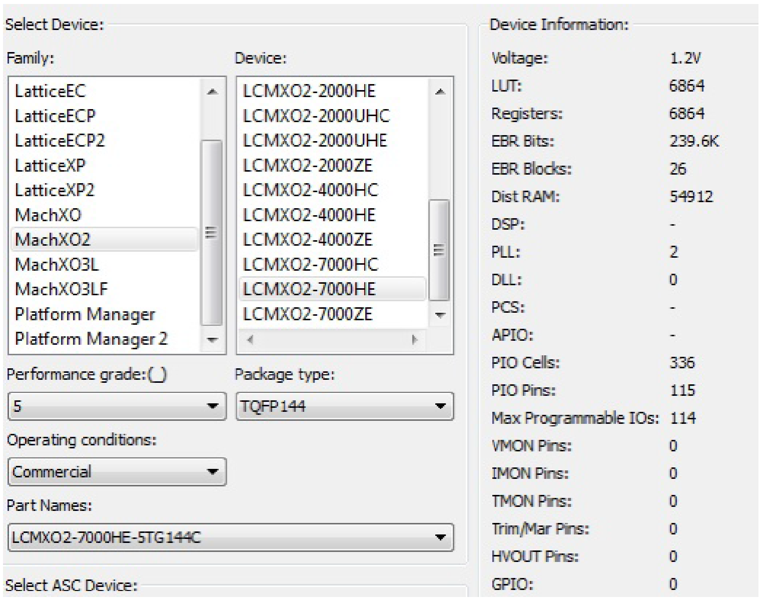

New Project

Add Source
Add HDL, EDIF netlist, LPF constraints, or other files.

Source files:

☑ Copy source to implementation directory

4. Select the correct FPGA model. In this case, the **SE***cube*™ FPGA correct version is «MachXO2-LCMXO2-7000HE-5TG144C»


Select Device:

Family:	Device:
LatticeEC	LCMXO2-2000HE
LatticeECP	LCMXO2-2000UHC
LatticeECP2	LCMXO2-2000UHE
LatticeXP	LCMXO2-2000ZE
LatticeXP2	LCMXO2-4000HC
MachXO	LCMXO2-4000HE
MachXO2	LCMXO2-4000ZE
MachXO3L	LCMXO2-7000HC
MachXO3LF	LCMXO2-7000HE
Platform Manager	LCMXO2-7000ZE
Platform Manager 2	

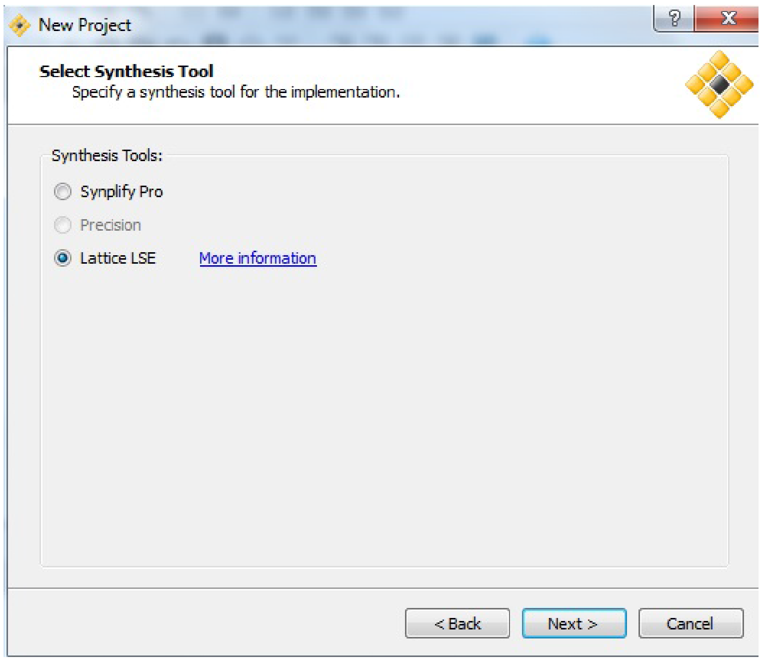
Performance grade:(_): 5
Package type: TQFP144
Operating conditions: Commercial
Part Names: LCMXO2-7000HE-5TG144C

Select ASC Device:

Device Information:

Voltage:	1.2V
LUT:	6864
Registers:	6864
EBR Bits:	239.6K
EBR Blocks:	26
Dist RAM:	54912
DSP:	-
PLL:	2
DLL:	0
PCS:	-
APIO:	-
PIO Cells:	336
PIO Pins:	115
Max Programmable IOs:	114
VMON Pins:	0
IMON Pins:	0
TMON Pins:	0
Trim/Mar Pins:	0
HVOUT Pins:	0
GPIO:	0

5. In the following window you have to select the Synthesis tool used. Select «Lattice LSE» and click «Next»



Select Synthesis Tool
Specify a synthesis tool for the implementation.

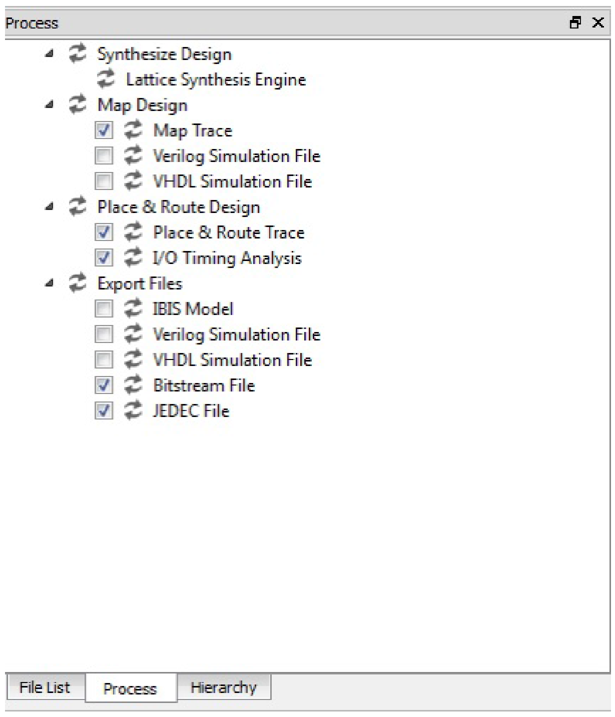
Synthesis Tools:
- ( ) Synplify Pro
- ( ) Precision
- [✓ Lattice LSE] More information

6. Open and edit the LPF source file in the section «LPF Constraint File». This file is really important, as it is used for mapping I/O signals to pins and for configuring parameters as the target clock frequency. The file is structured as a set of non-sequential commands. The command `FREQUENCY` is used to set the target speed of the design. The command `LOCATE COMP` is used to map ports of the top entity to I/O pins of the FPGA. Refer to our predefined file "FPGA_IPM.lpf" that you find along with the other VHDL source files for the correspondence between the pinout of the FPGA and the top entity ports. You can anyway customize your own LPF file. Anyway, make sure that the frequency constraint must be lower or equal than 60 MHz, not to collide with the software driver settings

7. Save all current changes

5.2.3 Synthesis Procedure

The synthesis procedure requires to complete all the process steps that are selected in the screenshot below here:



- Synthesize Design
  - Lattice Synthesis Engine
- Map Design
  - [✓] Map Trace
  - [ ] Verilog Simulation File
  - [ ] VHDL Simulation File
- Place & Route Design
  - [✓] Place & Route Trace
  - [✓] I/O Timing Analysis
- Export Files
  - [ ] IBIS Model
  - [ ] Verilog Simulation File
  - [ ] VHDL Simulation File
  - [✓] Bitstream File
  - [✓] JEDEC File

Then, it should be necessary to manually convert the static C source files generated to be compliant with the firmware. To skip all these steps, it is just sufficient to import in the working folder the “run.tcl” file that you can find under the “HDL source code” folder. This is a TCL script already containing all necessary commands to complete all steps.
Once copied this file in the directory where Lattice Diamond is working, open the «Console» tab of Lattice Diamond, and type:

```
> source run.tcl
```

This will generate a custom version of the file “se3_fpga_bitstream.h”.

5.3 Putting all together

Now that your HDL project is complete, you have to include it in a custom device-side project set up for working with the FPGA. In order to do that,

1. Import the project as described in the abovementioned general document
2. Inside the downloaded files for this project, browse to the “/C source code” folder, where the API libraries for the CPU-FPGA communication and the additional ST libraries for the CPU are located
3. Import all the files in that folder into your project, possibly resolving conflicts in favor of these new files
4. In the project settings, if not automatically supported, add the imported files to the build path of the project
5. Do these import steps also for your custom source files for interacting with your IP cores

6. Now edit the code in "main.c" file including the header files relative to the FPGA programming and communication:

   ```
   #include "se3_fpga.h"
   #include "Fpgaipm.h"
   ```

7. Possibly include your own header files relative to your own IP cores

8. Also in "main.c", add a call to `se3_FPGA_Init()` along with the `MX_` initialization functions for the microcontroller modules. Make sure that `MX_GPIO_Init()` and `MX_FMC_Init()` are called before it

9. Remember to pass to the `se3_FPGA_Init()` the correct clock prescaler in order to be compliant with your design[7]

10. Remember to correcly set setup/hold times for the FMC inside `MX_FMC_Init()`[8]

11. Before inserting calls to the APIs for interfacing with the FPGA, be sure that it is brought to a reset state by calling `se3_FPGA_Reset()` which have the aim of resetting the whole architecture inside the FPGA

12. If not present, import in your project the file "stm32f4xx_hal_sram.c", located in the downloaded firmware files of the SDK, under "/Drivers/STM32F4xx_HAL_Driver/Src"; if not automatically supported, add the imported file to the build path of the project

13. Save the changes to all files and build the project

14. Connect the DevKit as described in the abovementioned general document

15. Run the project

Remember that at startup, all the LEDs of the **SE***cube*™ DevKit are in a weak pullup state which indicates that the programming is advancing. After the programming (that may last up to 2 minutes), the LEDs are set on or off or left in the same state depending on what is stated by the HDL code and the connection done through the LPF file: all the LEDs are in control of the FPGA, as it will be explained in the next Section.

[7] cfr. Chapter 5 of the general wiki of the **SE***cube*™ SDK.
[8] cfr. Section 6.2

6 Technical Guidelines

This section wants to give some guidelines for future hardware/software developers within this project, with reference to technical details of our work.

6.1 Hardware design guidelines

As already mentioned, the IP cores can be designed in whatever way internally but must follow some fundamental rules.
First of all, since the communication protocol reckons on enable, acknowledgment and strobe signals for notifying the end of some activity by the CPU, and their activation is distributed over time, the block must be sequential. A combinational IP core is not feasible for this project.
The external interface of the cores as presented in Section 2.3.2 must not be modified in any way to guarantee the correct behaviour during the communication. The IP Manager is thought to control this and only this interface with the generic core, so if for example the opcode field or the interrupt/polling bit are deemed unnecessary, or the ack and the interrupt lines are not used since the core is thought to work only in polling mode, the ports must be only ignored or kept stuck at given values, but never dropped.
Modifications that do not involve any change in the communication interface with the CPU but, for example, are intended to add ports to drive other I/O pins of the FPGA, the LED (as explained soon) or to read the buttons are instead allowed. The information about the physical connection of the I/O pins of the FPGA are reported at the bottom of the general documentation of **SE*cube*™** [9].
All the cores are fed with the global FPGA clock and reset signals coming from the CPU. When the reset arrives, all the blocks of the design are reset, so the FPGA as a whole is reset, as well as when the clock is suspended by the CPU, the entire FPGA stops. There is no way to gate or control these two signals for the cores through the manager using some special command for the Manager or something. The signals can only be managed internally by the custom description of the core, but this is an unsafe and not recommended solution.
The typical IP core should be structured internally as an FSM with datapath (FSM-D) that performs a given algorithm. It usually remains in an IDLE state until the enable signal arrives. This signal unlocks the machine. The core acquires the information about the type of transaction (interrupt or polling) and the operative code transmitted by the CPU, and usually moves to a reading state, in which it waits for the write completed strobe signal before addressing the buffer and read the inputs. Once read the last input, the core starts doing its job.
The computation may take an undefined number of clock cycles. The CPU can wait a precise number of clock cycles knowing the clock frequency the FPGA is fed with, but this choice is always not recommended for the possible indeterministic delay that can affect the computation. Usually, either the CPU polls a register waiting for the end of the work or it closes the transaction and waits for an interrupt request continuing its operations.
During the computation phase in interrupt mode, the core is not enabled to read the buffer. It is instead enabled to do it in a polling transaction, but the idea is that the buffer should not be used as a RAM by the core, which implements its own internal block of storage if required. The buffer is only left for exchange of inputs and outputs.
Once the algorithm is executed, either the core is still enabled, and so it writes its input on the buffer before unlocking the CPU stuck on the polled register, or it signals the Manager with rising its interrupt line. The request is forwarded to the CPU when possible, as we said, and after a certain time it is acknowledged.

[9] At https://www.secube.eu/resources/open-sources-sdk/, download the Wiki relative to the latest version of the SDK, and go to **Appendix B - SE*cube*™ DevKit Schematics** for reference.

The IP core thus receives an enable signal along with the ACK, and its machine is unlocked. The access to the buffer is granted and now the core can start writing its outputs. Here a debate can be born on the CPU synchronization: after having started the ACK transaction, when is the CPU enabled to read the buffer? Some time is surely required to fill the buffer with the results, and there is no inverse strobe from FPGA to CPU to say that some write operation has been done.
In the normal case, a further polling solution is considered as philosophically incorrect, but the core can be designed to clear a row in the buffer and to oblige the CPU to poll that area until the writing is not ended. Considering the different temporization between the CPU and the FPGA this is a not recommended solution, which is preferably substituted by a sleep time after the ISR call, as the next section will explain.

For the limited dimensions of the FPGA mounted on the **SE***cube*™ DevKit and consequently of the data buffer implemented in the project, a core should be not designed for reading instructions as input, even if this solution is exploited in many coprocessor examples on FPGA. The field for the operative code left in the command word can somehow try to overcome this problem, allowing the possibility of designing a core in a flexible way, with the ability of executing more than one program on the inputs stored in the buffer. Although, it is to be remarked that the **SE***cube*™ FPGA is not a large device, 7000 LUTs are barely sufficient to host the buffer, the Manager and two or three simple cores, so mounting up a complex and flexible hardware may lead to have an extra-occupancy.
Alternatively, a little microcode memory can be inserted in the RTL of the core and changed with redesign process or even at runtime exploiting the limited resources available. However again, there may be no enough space for writing a program for a core using the LUTs or the distributed RAM as code memory.

The IP cores inserted in the design may be of whatever type, not only accelerators for common algorithms which require parallelism enhancements. Especially in the embedded domain, the FPGA can be exploited to implement memory controllers for external memories or even peripheral controllers. The 47 general-purpose pins present in the external interface of the FPGA have been made available right for this scope. In such scenarios, transactions are aimed at sending or receiving data from external devices. The CPU therefore waits for the end of a data transfer rather than for the results of an algorithm, and it is possible that transactions with different modes are wondered to control the upstream and the downstream. Taking the example of a controller for a local network interface, the CPU could write outgoing packets on the data buffer and could wait for the end of the upstream polling a control word, while it could be informed of packet arrival by the interrupt signal.
In such IP cores, more aimed at communication than at calculation, a certain number of preliminary settings preceding the the actual transfer of data is likely required (settings of transfer rate, preferred data width, window size, packet header and so on), so two different types of transactions could be foreseen by the designer, one for configuration and one for data.
Whatever the core implemented, the developer must take into account that the FPGA is the block that control the LEDs, numbered from 0 to 7, of the **SE***cube*™ DevKit. Referring to the pinout of the FPGA shown in the Getting Started manual and also in Figure 2 of this document, LEDs are controlled by the following pins:

LED	PIN
LED0	PR16C
LED1	PR16D
LED2	PR17C
LED3	PR17D
LED4	PR18C
LED5	PR18D
LED6	PR19A
LED7	PR19B

If not mapped in the LPF file, these LEDs are left in a weak pullup state. To be controlled by a core, they must be mapped in the LPF file following the table above. Remember that, being in pullup, they are lightened by setting at logic 0 their control pin and turned off vice versa. The usage of LEDs must be very useful for core's debugging purposes.

Similarly, the FPGA also controls the two push buttons (PB) placed under the LEDs on the DevKit. The switches control in pulldown (active low) the following pins:

PB	PIN
PB0 (SW4001)	PR19C
PB1 (SW4000)	PR19D

As for the LEDs, also the PBs must be mapped in the LPF file following the table above.

Due to the timing paths present in the data buffer and in the IP Manager entity, the FPGA must run with a clock 3 times slower than the CPU (which runs at 180 MHz), as only integer prescaler can be set and a half rate (90 MHz) is too fast for the two main blocks of the architecture. A 60 MHz (or lower) constraint must be thus set in the LPF file, so that if setup and hold times are violated (see 5.2.3), the designer is advised and is free to choose either to further slow down the clock frequency arriving from the CPU or to edit the design in order to meet the constraint. The chosen running frequency is related to the two generic parameters of the top entity VHDL file, ADDSET and DATAST, which must respect the software configuration stated by the low-level driver, as it will be presented in the following paragraph.

6.2 Software design guidelines

For correctly interacting with the FPGA, a dedicated low-level driver was developed. Functions for initializing the environment, opening and closing a transaction, writing and reading to or from a specified location in the buffer and a global FPGA interrupt service routine are present in this set. A semaphore then is encharged of blocking possible new transactions when one of them is still opened with a core. The APIs use recasts and wrappers for native types and relative addresses not to force the user to know lower details. Therefore, the first suggestion is to always and only use this API to control the CPU-FPGA communication.
In any case, it is strongly recommended to write a piece of software which is aware of the rules of the communication explained in Section 3 (e.g. when in polling mode, first open the transaction, then write inputs, then listen to a register and then read outputs before closing, when in interrupt mode do not poll the core if the interrupt is not raised yet, etc.)
When an interrupt request arrives, the global ISR of the FPGA (connected to pin PA9 of the CPU) is called (`EXTI9_5_IRQHandler()`). The body of this function must be actually customized by the user. Anyway, a trace is already present in the released file of the driver. After having checked that the logic level of the line is high, the CPU should read the address 0x00 of the buffer first, to identify the interrupting core. At that point, the private variable `row0` is updated and can be

switch-cased. For each allowed value, i.e., for each present core, the dedicate interrupt service routine has to be written there. Before returning from the global routine, the interrupt flag is cleared so that the CPU is ready to be interrupted another time.
The generic ISR of a core should open an acknowledgment transaction rather than a normal one, so that both the core and the manager are advised that the request has been accomplished. When received the ACK along with the enable, the core knows that the CPU wants to receive data from it, so the core can immediately start to fill the buffer. If the CPU starts reading the buffer immediately after the transaction opening, it may incur in timing errors, as it does not know what is happening and what are the words already written and the ones to be written yet. It is thus advisable for the CPU to wait a little time, depending on an estimation on what is the time required by the core to write its outputs.
The other functions of the driver can be trusted, left untouched and just used. Function `se3_FPGA_Init()` must be fed with a correct prescaler depending on the maximum clock frequency supported by the FPGA design. Parallely, the `MX_FMC_Init()` may be customized for the setup/hold times of the FMC, once fixed these frequencies.
If the design does not meet the 60 MHz constraint, the parameter of `se3_FPGA_Init()` can be changed to set another prescaler. The parameter is in the form RCC_MCODIV_X, where X can assume values from 1 to 5, i.e., the FPGA can run at frequencies from 180 MHz to 36 MHz. Once decided the operating frequency, it is also possible to change `AddressSetupTime` and `DataSetupTime` fields of the timing structures relative to read and write operations of the FMC, inside `MX_FMC_Init()`, to stretch them as preferred. Values are expressed in terms of CPU clock cycles. Changes are free, but the overall settings must be compliant each other. In fact, given X the value of the prescaler, software parameters `AddressSetupTime` and `DataSetupTime` and hardware parameters ADDSET and DATAST (VHDL generics of the top entity), the following formulas must be absolutely respected to make the system work:

```
ADDSET = AddressSetupTime / X
DATAST = DataSetupTime / X
```

Since the two hardware parameters are read by a counter internal to the FPGA, they must also be natural non-zero values.

6.3 Single-core applications

The FPGA architecture developed in this project is thought for a multicore environment, where multiple IPs coexist to perform different tasks, with the advantage of being able to address them individually without having to reconfigure the entire FPGA whenever you want to use a different service. However, in principle nothing forbids having an environment where there is a need to exploit the advantages of a single IP core. In this scenario, the IP Manager has no effective task to perform and becomes a useless intermediary between the buffer and the only IP present. For single-IP applications that require optimized timing and area constraints, the Manager could be dropped without the need for a radical redesign of the environment: the IP can maintain the standard interface defined by our work.
In fact, since the IP Manager works primarily as a dynamic connector of the core interface with the buffer interface, in case there is a single core this connection can be operated statically in the top entity of the VHDL design, without changing anything neither in the default interfaces of the buffer and of the core nor in their internal behavior.

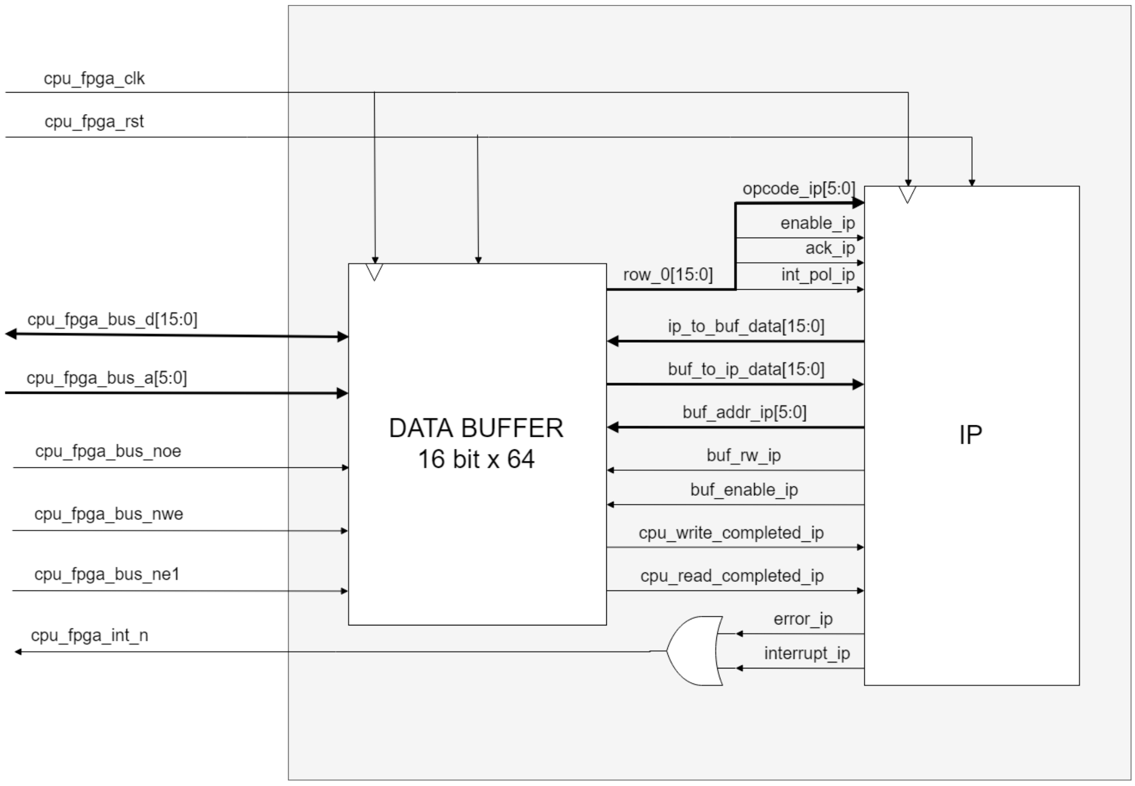


Figure 16: Example of the FPGA internal connections with a single IP core

The Figure 16 shows the internal connections of an FPGA package with a single core and without the IP Manager. As it is possible to see, there is no port modification in the entities. The first address of the buffer, containing the control word, could be simply split in its components and immediately reflected to the IP which then is informed of the start/end of the transaction and of the transaction parameters. Data, address, control and strobe signals are not multiplexed but statically assigned to the only IP present, which uses them in the exact same way than in the IP Manager scenario. The interrupt and error signals are not to be handled by any controller, since the request can only arrive from one IP, so both of them can be attached to the FPGA global interrupt signal. The distinction between these two cases could be overcome.
Obviously, the CPU must be aware of this situation, and the driver should be minimally changed to avoid useless controls. For example, it is no longer necessary to check at software level the identity of the active core, but only if there is already an open transaction or not, at most. Alike, setting the **IPADDR** field of the control word becomes pointless (that part of row 0 is neglected at buffer interface).
The new ISR for such an FPGA could skip the reading of row 0 and immediately start an acknowledgment transaction to get the results or to understand the problem, in case the interrupt was triggered in response of an error line assertion.

7 An example IP core: SHA256

After creating a reliable environment, the authors decided to include an example IP core in the project, a core that executes an algorithm of effective and wide use in the cryptographic domain: the SHA256 Secure Hash Algorithm. This section is intended to present the SHA256 IP core, first introducing it with a brief overview of the algorithm and the architecture, and then continuing with the presentation of its use modes.

7.1 Overview

Secure Hash Algorithm (SHA) is a label given to a family of cryptographic algorithms which have been developed and published since '90s by the U.S. Government. Given a message or a file or any sequence of information, the SHA produces a *digest* for it, a compact string (of 256 bits in the SHA256 case) which is unique and not reversible. These features make it a safe way to store sensitive information, preventing it from tracing back to the original message (think of the case of passwords stored on servers) or from modifying (think of a file whose hash represents its kind of digital certificate, since just a few bytes of difference completely change the digest).
The hashed message can be of whatever dimension but must be divided into *blocks* of 512 bits that one after the other contribute to transform the starting hash (always the same, defined by the standard) into the digest with a series of plain operations, mainly rotations and additions. At every round of the algorithm, one 32-bit word from the block (extended from 512 to 2048 bits) enters the main sequence along with the hash (called also state), and as a result a new state is produced. There are thus 64 rounds like this, then at the end the state is summed with the starting hash to obtain the digest. The algorithm is therefore mostly sequential and, more important, one block must wait the computation of the following[10].
The core we developed for the execution of the SHA works with one block and leaves the division of the message into blocks and the padding of the last one to the driver. In other words, every time a block is to be computed, the intermediate state and the block itself must be written on the data buffer and the production of a new state must be awaited. This was an almost obligatory choice for the reduced dimensions of the data buffer, which offers a memory space which is barely enough to contain the state (16 16-bit words) and a single block (32 16-bits words).

The Figure 17 shows the internal architecture of the core. The control unit is the main central entity of the core, and contains the microcode for controlling the datapath, but also is responsible of control signal interpretation and driving and address forwarding towards the buffer. Whenever the enable signal is sensed, the machine inside this unit awakens and starts reading the state putting it into the dedicate registers. After that, the reading of the block starts: the acquired words are inserted in a queue of registers called *block FIFO*. The transaction is then closed and the chain, now filled, releases its content with a first-in-first-out policy towards a first computational block called *R-pipeline*. This pipeline contains 3 stages and is used to extend the block from 512 to 2048 bits creating 48 additional 32-bit words needed for the computation. The work of the R-pipeline actually goes in parallel with the main sequence of the algorithm, represented by the *P-pipeline*. This is a 4-stage pipeline which makes the actual encryption of the block. The first word of the block is encrypted using the initial state stored in the dedicate registers, while the following are encrypted using the state coming from the previous cycle of the P-pipeline. Once every of the 64 32-bit words have passed through the 4-stage pipeline one after the other, the final sum between the obtained state and the initial one can be computed, and the output can be stored, with modes and times dictated by the transaction mode adopted (interrupt/polling).

[10]For additional details on SHA family of algorithms, please refer to https://web.archive.org/web/20130526224224/http://csrc.nist.gov/groups/STM/cavp/documents/shs/sha256-384-512.pdf

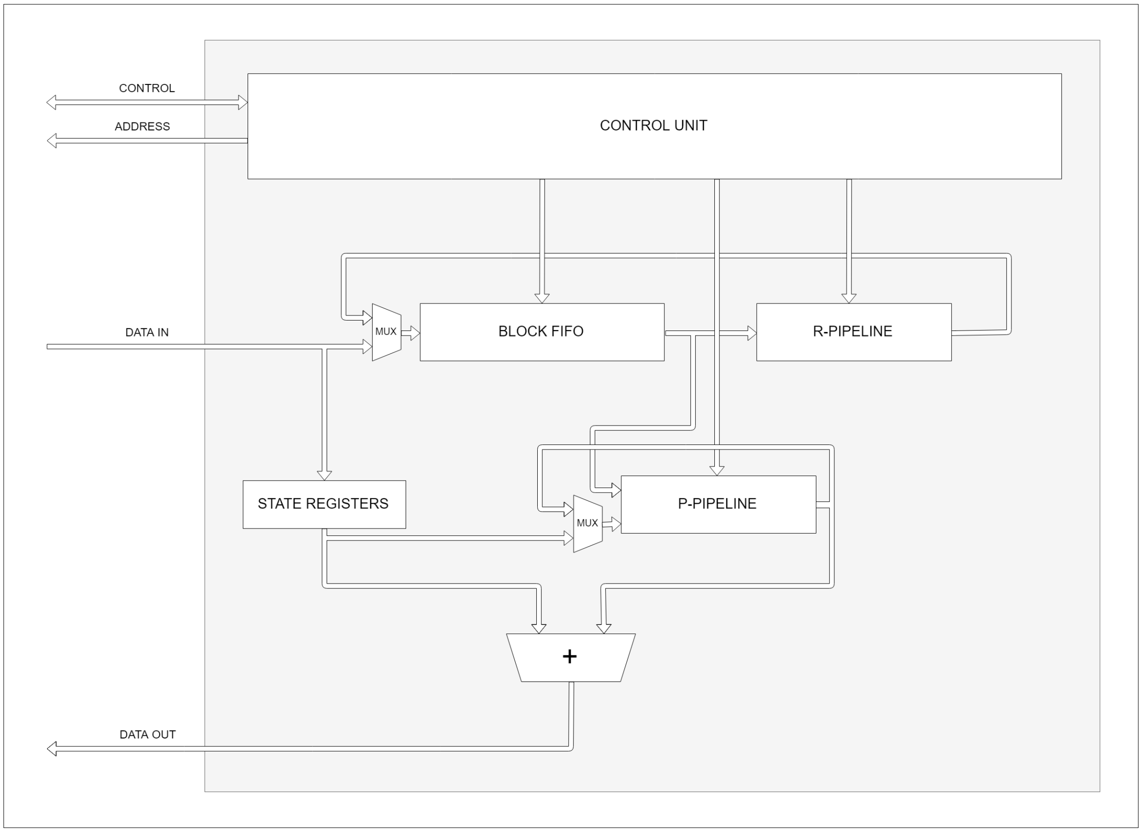


Figure 17: SHA256 IP core architecture overview

7.2 Testing the core via HDL simulation

In the downloadable material relative to this project, the folder containing the HDL code for the FPGA also contains a testbench ("FPGA_testbench.vhd") which can be simulated with whatever HDL simulation tool available to you. We anyway recommend the use of Altera™ ModelSim, for which a TCL script file is already provided in the same folder.
The testbench contains procedures which emulate the FMC behaviour during read and write operations and 2 benchmarks that show a polling and an interrupt transaction respectively, with the core computing the same block with the same starting hash.

7.3 Testing the synthesized core via high-level driver example

In the other folder of the downloadable material, the C source files are stored. These files are to be added to the project as explained in 5.3. The file "se3_fpga_bitstream.h" present here already contains the FPGA configuration with the IP Manager and the SHA256 core.
Besides the files named in Section 5.3, two additional files ("sha256_fpga.c" and "sha256_fpga.h") must be added with the same procedure in order to interact with the SHA256 IP core. Such files contain an example on how a high-level driver for controlling a core can be written.

7.3.1 SHA256 IP core usage example

This example is used to try the IP manager environment on your **SE***cube*™ , using the functionalities of the SHA256 core present inside the provided architecture. Here are the steps you need to follow:

1. Launch Eclipse

2. Setup a FPGA-related firmware project as described in Section 5.3

3. Inside the downloaded files for this project, browse to the "/C source code" folder, where the API for interacting with the SHA256 IP core are located ("sha256_fpga.h" and "sha256_fpga.c")

4. Import the files in that folder into your project

5. In the project settings, if not automatically supported, add the imported files to the build path of the project

6. Edit the code in "main.c" file, adding the following include at the beginning:

   ```
   #include "sha256_fpga.h"
   ```

7. Other than the fundamental calls for making the FPGA work as described in 5.3, before the `device_loop()`, add a call to the

   ```
   SHA256_FPGA_digest_message(const uint8_t *message, uint64_t
      dataLen, uint8_t *digest)
   ```

   using parameters of your preference. The function receives the pointer to the message, the pointer to the resulting digest and the number of bytes to be processed. Internally, it divides the message into blocks and pads the last block, and every time a block is ready to be computed, it is sent to a lower-level private function which handles reads and writes to establish a polling transaction with the core for performing the required computation

8. Save the changes to all files and build the project

9. Connect the DevKit as described in the abovementioned general document

10. Run the project

At the end of the programming phase, you should see all LEDs turning off, and after few instants, your digest is computed. Choose the mode you prefer (debug or trace print on UART/SDIO) to check whether the supposed digest is coherent with what you expected.