6/18/2017

Project 13: IP-core Manager for FPGA-Based Design (HLS)

Final Report and User Manual

Authors: A. Floridia, R. Margelli, L. Mocerino, L. Ngongang
POLITECNICO DI TORINO, A.Y. 2016/2017

Index

Introduction

The end-goal of this project was the integration of different HLS (High-Level Synthesis) tools, namely Stratus HLS and Vivado HLS with the toolchain provided with the SEcube platform.

The following document is structured in three sections (User Manual, Implementation Details, Future Works) and 2 appendices in which we underlined the project methodology, deliverables, milestones and some modelling guidelines.

With this document we aim at providing useful information on how to use and improve the On-Board IP Manager (hereinafter OBIPM) for the SEcube platform. In Section 1 (User Manual), we provide an exhaustive view of the whole system from the user point of view. Note that the user can be either a programmer, in charge for writing an application program to be run on the CPU available on the SEcube SoC or an IP provider (i.e. a hardware designer) in charge of delivering a new IP-core for the on-board FPGA.

In Section 2 (Implementation details), we provide useful information on our design, from the SystemC models to the RTL validation after the high-level synthesis. Detailed how-to chapters are devoted to how the tools have been used and insights on the performed analyses are provided.

Section 3 describes which could be the possible next steps starting from the result of this project.

Finally, the appendices are devoted to listing some of the modelling guidelines we adopted in order to obtain a generic SystemC, which is tool-independent as much as possible. We also provide information on the applied project methodology and milestones reached to this date.

Section 1: User Manual

1. CPU-FPGA communication protocol

1.1 Introduction

The following is a description of the proposed communication protocol between the CPU and the FPGA. The main design goals were the simplicity of the hardware (in terms of area) and the performance. For achieving these two objectives, we divided part of the effort between the hardware implementation and the software tools.

With the present document, we will mainly focus on the hardware side, while briefly detailing which software functionalities are in in turn needed.

The protocol specification is organized as follows: section 2 illustrates the interface of the internal buffer and describes the overall functionalities. With section 3, we provide an overall description of the IP Manager interfaces. Section 4 is devoted to the pins configuration and finally sections 5, 6 describe CPU transactions and interrupt requests respectively.

1.2 Buffer Interface

In order to simplify the design, we decided to implement (by means of a synthesizable HDL design) the buffer as a full custom dual-port register file. Figure 1 shows the interface. `PORT_0` is dedicated for the communication with CPU, whereas the IP manager controls `PORT_1`. It is worth nothing that there is an additional output port, called `ROW_0`. The purpose of this port is to enable an easy access to the row 0 by the IP manager, which (as discussed in details in section 4) uses this row to perform the correct route for allowing interaction with the selected IP. This port reflects any change in the row 0 of the buffer to whichever device is connected to (in this case, the IP Manager).

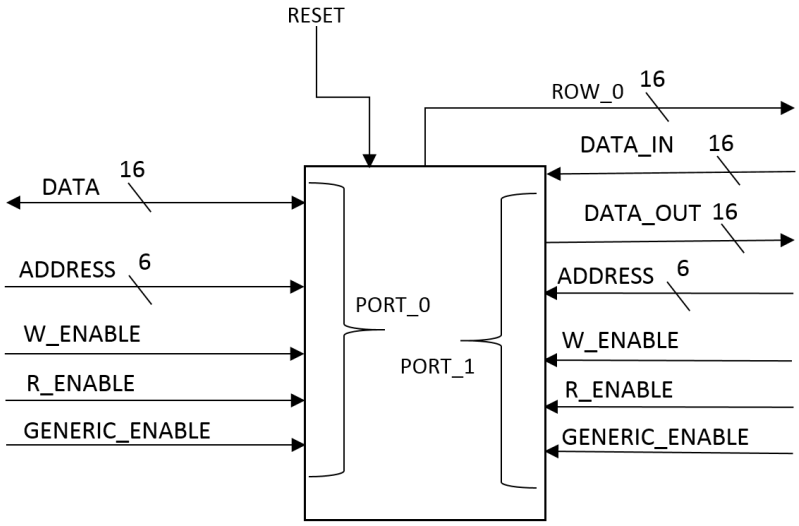


Figure 1 - Buffer interface

Another important aspect is that, as it happens with regular register files, the module is completely asynchronous, as a clocked register file would degrade the overall performance. On the other hand, the IP manager itself is synchronous and is fed with a clock signal from the CPU. In order to read/write on one of the two ports, the generic enable signal must be asserted as well as the required control signal (`W_ENABLE/R_ENABLE`). Then the correct values must be placed on data and address lines. In details, each signal (for both `PORT_0` and `PORT_1`) has the following function:

- `DATA`: input/output port, used for transferring data to/from the buffer.
- `ADDRESS`: input port, used for selecting the right row within the buffer to write/read data.
- `W_ENABLE`: input port, must be asserted in order to enable write operation.
- `R_ENABLE`: input port, must be asserted in order to enable a read operation.
- `GENERIC_ENABLE`: input port, must be asserted in order to start any operation on that port.
- `RESET`: input port, it brings the buffer to the reset state.
- `ROW_0`: output port, it reflects any change in the row 0 of the buffer.

Allowed configurations for each port are shown in table 1:

OPERATION	READ_ENABLE	WRITE_ENABLE	GENERIC_ENABLE
READ	1	0	1
WRITE	0	1	1

Table 1 - Possible configurations of the three signals

Any other configuration will not produce results (e.g., all outputs ports will be at 0).

As final remark, it's worth analysing the `DATA` port belonging to the CPU-FPGA interface. It is both input and output port, hence its behaviour must be compliant with a digital port driven by a tristate logic buffer. That is, when there is a READ operation (i.e. data flows out of the buffer) the tristate buffer must be activated by appropriate control signals, whereas in case of WRITE operation (i.e. data flows in the buffer) the tristate buffer should be in high impedance. Figure 2 describes a simplified hardware implementation:

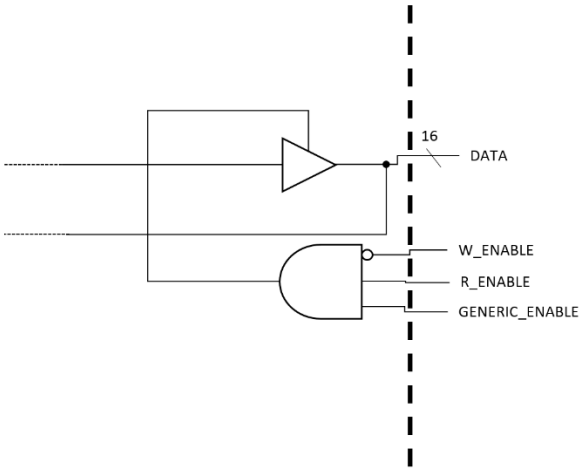


Figure 2 - Simplified hardware implementation of bidirectional port

1.3 IP Manager interfaces

The main tasks of the IP Manager are the correct selection of the target IP and the interrupt handling. Figure 3 shows the overall architecture of the whole system.

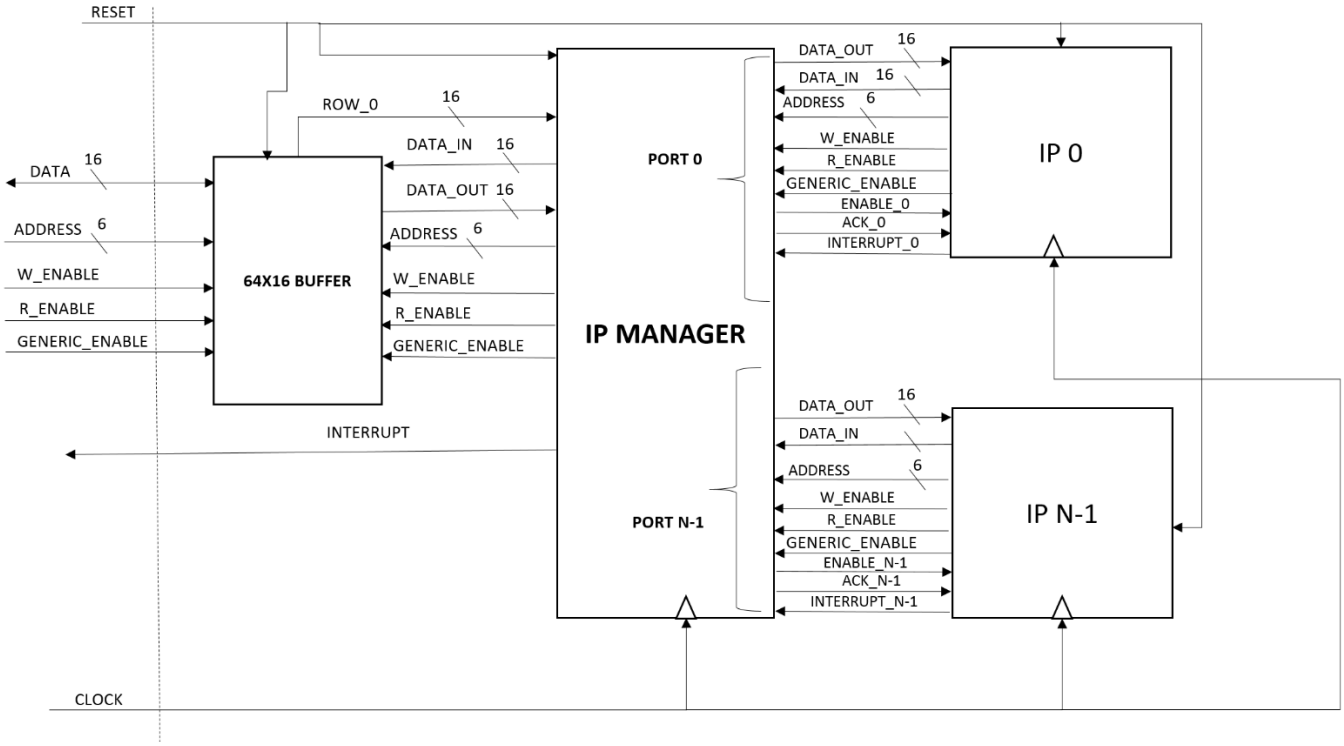


Figure 3 - System architecture

As shown in the figure, the IP manager has a standard interface with the buffer. This interface is used to redirect data to/from the target IP and to access the buffer itself (e.g. to write the row 0). However, it is

provided with a direct access (read-only) to the internal element to speed-up the routing process whenever a new transaction begins. It is important to underline that for each instantiated IP there are a set of standard signals and some protocol-specific signals. The former are the very same signals used for accessing the buffer, whilst the latter are used for handling interrupt requests. For the sake of clarity, we report the function of the IP-specific signals for the generic IP x:

- `ENABLE_x`: input port for IP x, output port for IP Manager. The IP Manager set this signal at the beginning of a transaction only. The signal is cleared at the end of a transaction. For the whole duration of the transaction, the signal must be maintained asserted. From the IP x viewpoint, as long as the signal is zero the IP itself is not activated and all the outputs are set to 0.
- `ACK_x`: input port for IP x, output port for IP Manager. Asserted by the IP Manager to acknowledge the interrupt request of IP x. It must be maintained asserted for the whole duration of the transaction. The IP x clears on the reception the `INTERRUPT_0` signal.
- `INTERRUPT_x`: output port for IP x, input port for IP Manager. Asserted by IP x to raise an interrupt request. It must be maintained asserted until it receives `ACK_x` from IP Manager.
- `INTERRUPT`: output for the IP Manager, used for signalling that an interrupt arrived from one of the IPs.

For the remaining signals, refer to previous section.

1.4 Pin Configuration

The present section provides a short description of which pins of the CPU-FPGA connection[1] are used and how they should be programmed by the CPU. The signals that should be connected to such bus are depicted in figure 3.

Specifically, they are those crossed by a vertical dotted straight line, in the left-hand side of the figure. In table 2 there are shown such associations:

Signal Design Name	FPGA-CPU Connection Name	Programmed
RESET	CPU_FPGA_RST	OUTPUT
DATA	CPU_FPGA_BUS_D0:15	INPUT/OUTPUT
ADDRESS	CPU_FPGA_BUS_A0:5	OUTPUT
W_ENABLE	CPU_FPGA_BUS_NWE	OUTPUT
R_ENABLE	CPU_FPGA_BUS_NOE	OUTPUT
GENERIC_ENABLE	CPU_FPGA_BUS_NE	OUTPUT
INTERRUPT	CPU_FPGA_INT_N	INPUT
CLOCK	CPU_FPGA_CLK	OUTPUT

Table 2 - CPU-FPGA connections

[1] CPU-FPGA connection is detailed in this document (page 8, section 2 FPGA-CPU connection): https://www.dropbox.com/s/4fysjgntr7qx3uh/FP1 - The SEcube™ FPGA Configuration - Getting Started - rel 1.4.pdf?dl=0

It is worth to be noticed that CPU_FPGA_BUS_D0:15 must be programmed as input during a read operation, while as output during a write operation.

1.5 CPU Transaction

Whenever the CPU wants to begin a transaction with one of the IPs present on the FPGA, it writes at address 0 of the buffer a data packet compliant with the following specifications:

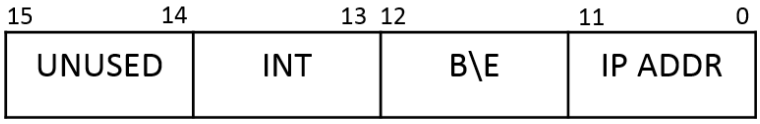


Figure 4 – Buffer row 0 fields

Bit(s)	Purpose	Value(s)
Bit 15	Unused	Unused
Bit 14	Unused	Unused
Bit 13	Interrupt ACK from the CPU	Interrupt = 1, Normal = 0
Bit 12	Signals the begin/end of a transaction	Begin = 1, End = 0
Bit 11-0	The physical address of the target IP.	From 0 up to N-1

Table 4 – Buffer row 0 fields

In the following, we explain what happens during a read or write transaction.
The CPU writes at address 0 the control word for the IP manager. In particular, the IP manager will always assume that the IP address that is written from bit 11 up to bit 0 is already the physical address of the IP (i.e. one of the output port) and it will not perform any further translation. Hence, it is up to the software environment to properly connect the right IP to the right port and to configure correctly the device drivers. Figure 5 shows the association that the software tool is supposed to do:

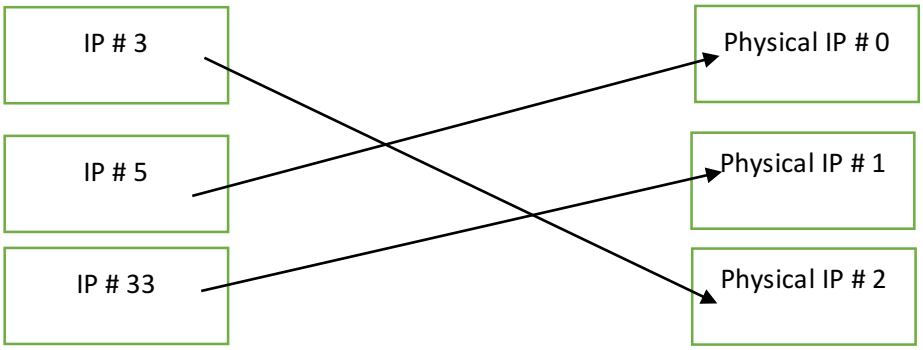


Figure 5 - Association IP number with physical IP

If the physical address of the control world is at all 0s, it means that the CPU wants to interact directly with the IP Manager for some kind of configurations (exploiting bits 15-14, left unused).

It is important to signal the start and the end of a transaction because this information will be exploited by the IP manager to understand whether it can signal to the CPU that an interrupt arose (more details in section 6).

Of course, both CPU and IP have to drive the buffer control signals according to the specifications of section 2 in order to access the buffer.

Form the viewpoint of the IP manager, the behaviour is also the same whether it is a read or a write transaction. Depending on the IP address, it will enable one of the IPs (asserting the `ENABLE_x` signal of Figure 3). The IP manager does not care about how many packets have to be written or read, nor whether is a read/write transaction, since it is IP-dependent. It must only keep the port associated to the target IP open as long as the value written in the IP address field do not change. The target IP will be enabled if the bit 12 is set (signalling the beginning of a transaction) and it will be not enabled as long as bit 12 is at zero (even if there is a change in the bits 11-0). Of course, even if not enabled for the current transaction, each IP has the capability of raising interrupt request if it has been programmed (during a previous transaction) for doing so.

To better clarify the transaction mechanism, it is reported a sequence diagram (figure 6) describing a normal transaction and the format of the control word:

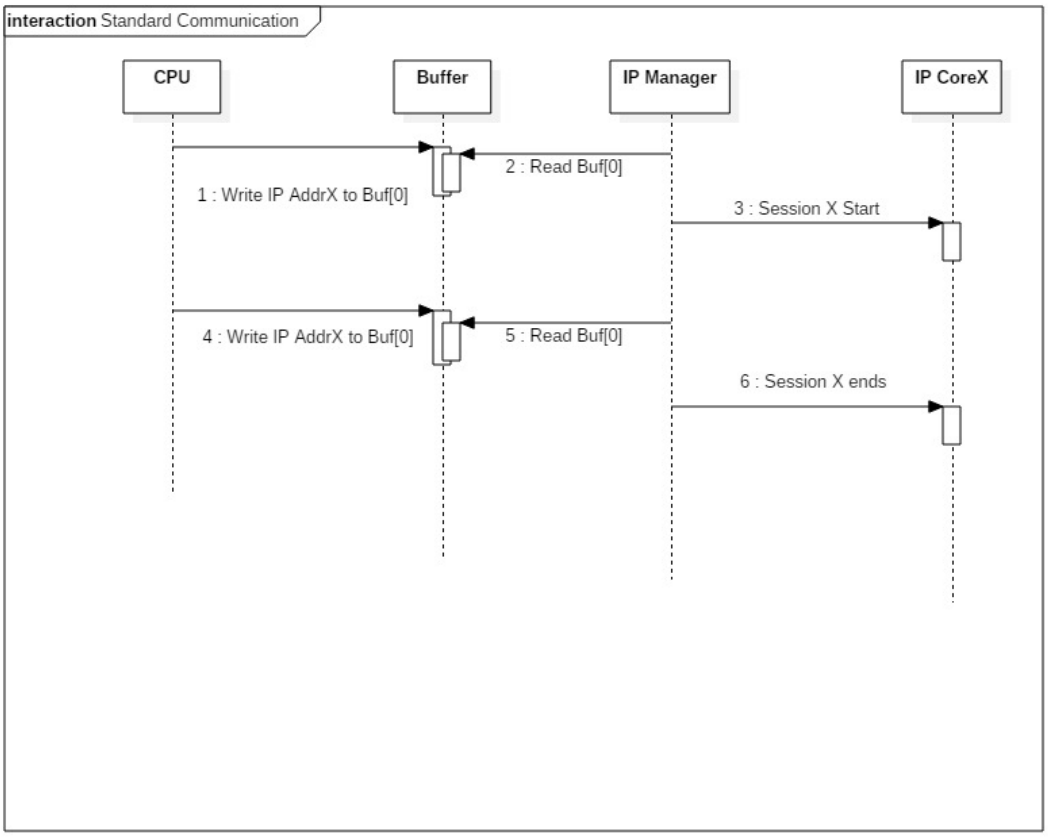


Figure 6 - Normal transaction

The format for the previous transactions (assuming that the target IP is the one connected to port 0):

1) Write IP AddrX to buffer 0(begin):
    - Bit 15,14: x (unused)
    - Bit 13: 0
    - Bit 12: 1
    - Bit 11-0: 001 (in hexadecimal, IP ADDR 0 is dedicated to the IP Manager)

2) Write IP Addrx to buffer 0(end):
    - Bit 15, 14: x (unused)
    - Bit 13: 0
    - Bit 12: 0
    - Bit 11-0: 001

1.6 Interrupt Handling

The association depicted in Figure 3 will be driven by priority of the IP (specified by the user). Highest priority will be always associated to IP manager port 0 (properly connected to the right IP), and the lowest to port N-1. The IP manager will always assume that IP connected to port 0 is the one with the higher priority.

Since the architecture is a master (CPU) slave (FPGA) architecture, the IP manager must guarantee that an interrupt request from one or more of IPs do not interrupt a transaction started by the master. For doing so, the manager leverages bit 12. Only when it is set to zero it signals to the CPU, through the `INTERRUPT` signal, that an interrupt service routine has to start. At the very same time, it writes to the address 0 of the buffer the IP address of the IP requesting the interrupt.

Before starting any transaction, the CPU reads the content of address 0 so that it knows which is the requesting IP. At this point, the CPU starts a typical transaction, with the exception that bit 13 now is set. Doing so the manager knows that the interrupt has been received, and it can clear `INTERRUPT` signal and sets the `ACK_x` signal to the requesting IP. This signal is cleared once the requesting IP clears its `INT_x` signal (done if and only if it receives the `ACK_x`).

It is worth noting that, if an interrupt request arrives while there is still an ongoing transaction, the manager must maintain the interrupt request alive. Indeed, the requesting IP will not clear the `INT_x` signal until it receives the `ACK_x` signal. The manager will raise this signal if and only if there is not any other ongoing transaction.

As final remark, since there could be several interrupts requests at the same time (with possibly different priorities), the IP Manager must guarantee that all requests are served. In particular, the IP Manager continuously rises the `INTERRUPT` signal as long as there are still pending requests. Hence, the `ACK_x` signal must be sent to only one IP at a time (according to the priority level).

A sequence diagram (figure 7) along with the formats of the control word are shown here (assuming that the requesting IP is connected to port 0):

1) Write IP AddrX to buffer 0(begin):

- Bit 15,14: x (unused)
- Bit 13: 1
- Bit 12: 1
- Bit 11-0: 001 (in hexadecimal, IP ADDR 0 is dedicated to the IP Manager)

2) Write IP Addrx to buffer 0(end):
   - Bit 15, 14: x (unused)
   - Bit 13: 1
   - Bit 12: 0
   - Bit 11-0: 001

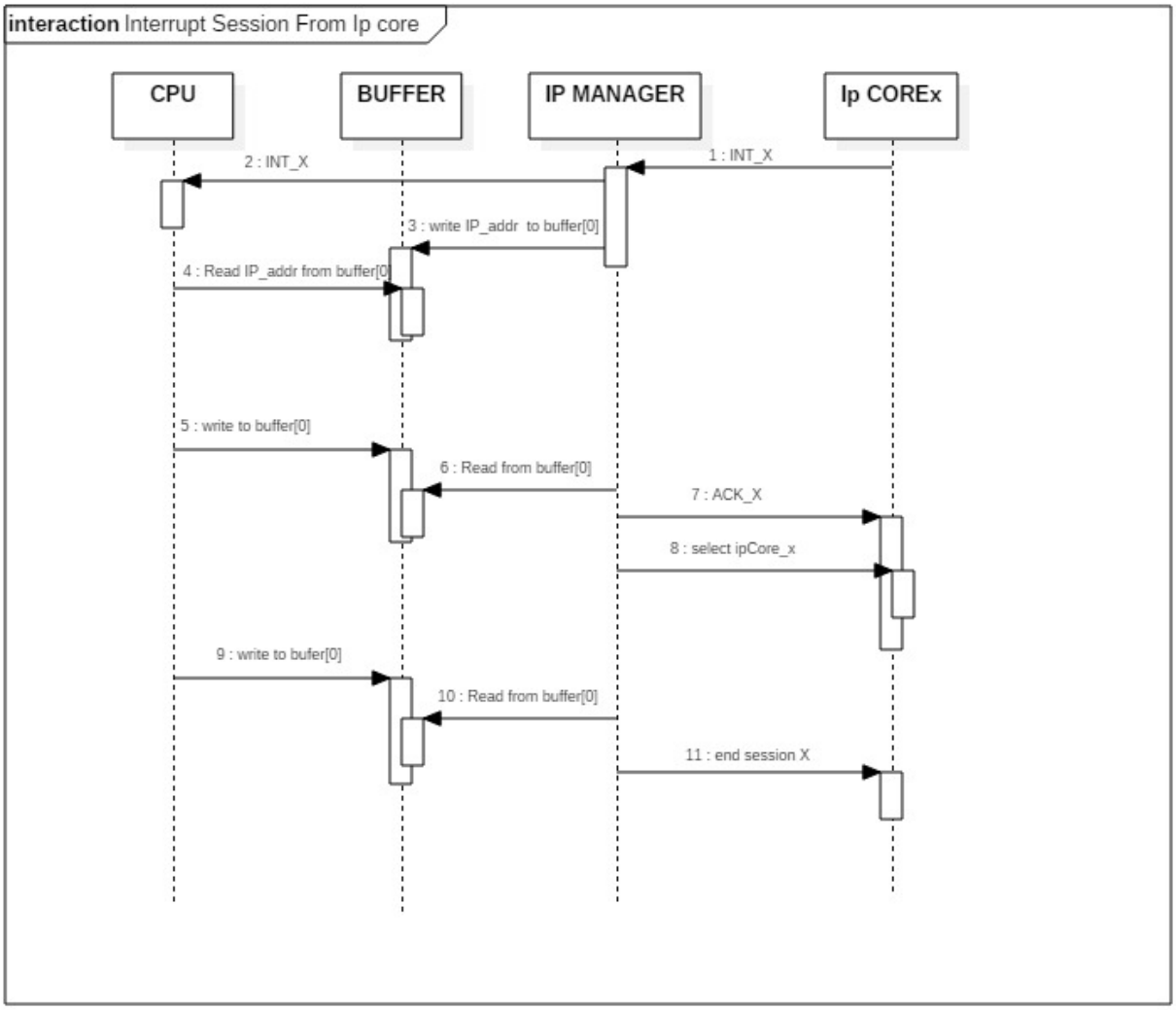


Figure 7 - Sequence diagram for interrupt transaction

Section 2: Implementation Details

2. SystemC Design

2.1 Introduction

The main purpose of this chapter is to provide a detailed view of the internal architecture of the OBIPM. For each module, interactions with other modules are highlighted as well as implementation decision. We also provide motivations to design decisions, such as why we chose SystemC as the modelling language.

2.2 Internal Architecture

Figure 8 depicts the internal architecture of OBIPM.

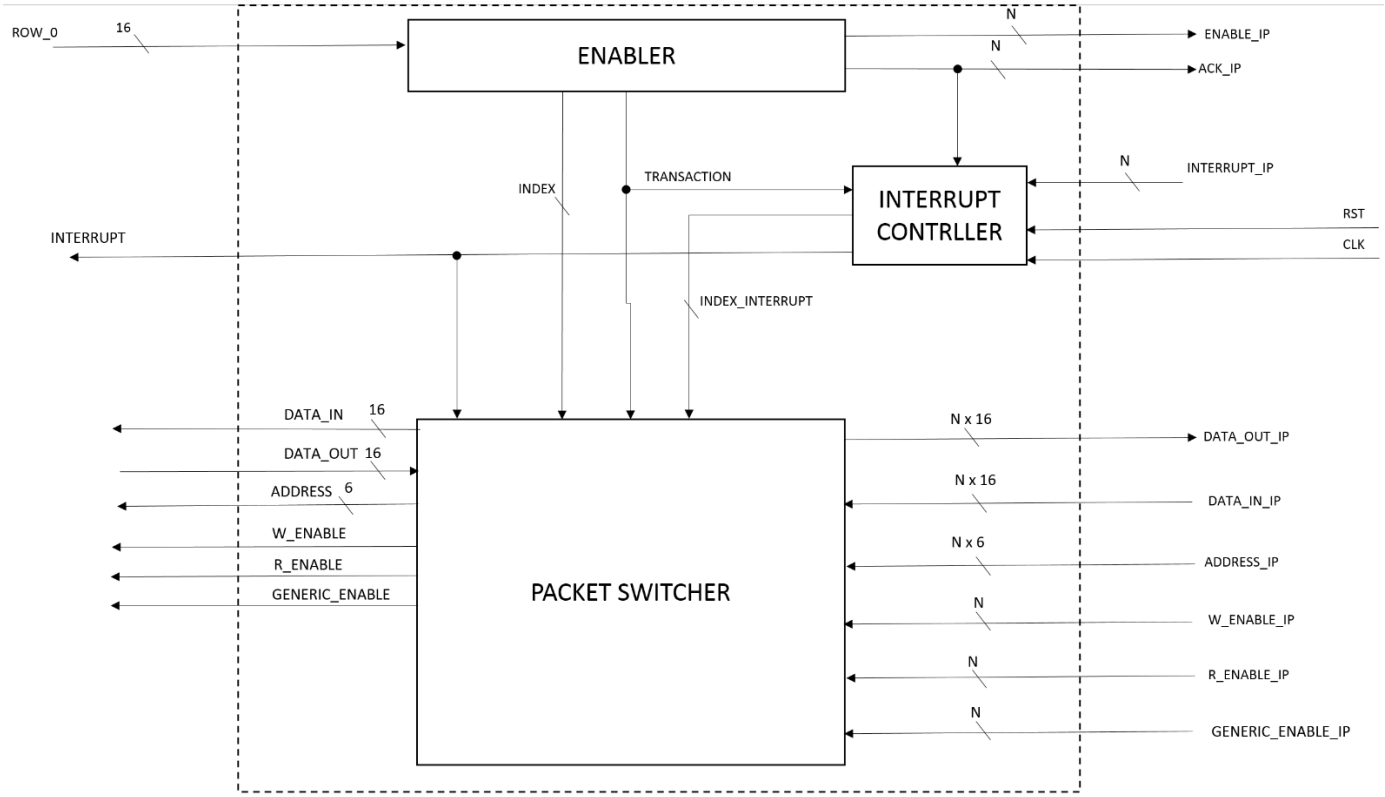


Figure 8 - OBIMP Internal architecture

There are 3 submodules:

- ENABLER: depending on the control word on `ROW_0`, it enables the selected IP (one-hot encoding) or sends an `ACK`. It raises the signal `TRANSACTION` for the whole duration of a transaction and it indicates to the PACKET SWITCHER the target IP for the transaction (via `INDEX` signal). It behaves as a decoder.
- PACKET SWITCHER: it is in charge for forwarding the signals coming from the target IP to the buffer and vice-versa. It is also in charge for writing to the buffer the address of the IP requesting an interrupt. This behaviour is controlled by the signal `INTERRUPT` and `TRANSACTION`. It can be thought as a switching matrix with added functionalities.
- INTERRUPT CONTROLLER: it detects and signals interrupts coming from IPs only when there is not an active transaction.

From the previous picture is worth noting that the Interrupt Controller is the only sequential block present in the design. Enabler and Packet Switcher are fully combinational circuit, since we need to guarantee a fast response from the OBIPM as soon as the CPU starts a transaction.

2.3 SystemC implementation

Several high-level languages fit well in the context of the high-level synthesis. Commercial HLS tools mainly support C/C++ and SystemC.

For this design we selected SystemC, since it offers the possibility to model at a cycle-accurate level whereas C/C++ are more suitable to model generic algorithms and map them to a hardware implementation. Furthermore, SystemC offers some constructs that resemble those of the other HDLs, simplifying the modelling of combinational or sequential behaviour. Finally, the SystemC library provides a simulation kernel that allows an earlier validation of the system at the behavioural level. Hence, the language allows rapidly refining the model until it satisfies the requirements.

From the design point of view, we decided to implement the three modules depicted in Figure 8 as two `SC_METHOD` (Enabler and Packer Switcher) and one `SC_CTHREAD` (Interrupt Controller). Since the language does not support concurrent statement as VHDL does, we introduced a further `SC_METHOD` called `Support_Logic`. This entity is in charge for routing internal signals to the module and performs routing of some signals that are useful for enabling some modules.

All these entities are enclosed in the top-level module, OBIMP. Thus, the main design files are obimp.cpp and obimp.h in according to the SystemC conventions.

We decided to implement the modules composing the internal architecture as `SC_METHOD/CTHREAD` in order to obtain a flexible structure and avoid too many files to deal with and to ease the expansion of the functionalities.

All the modules are parametric with respect to the number of IPs that are instantiated for a given design. It is up to the designer to fix that value before starting any synthesis or simulation, by properly tuning the define `NUM_IP` (found in `define.h`).

Internally to each module, we leveraged variables in order to maintain the implementation as much as generic as possible and to solve some limitations of the language (e.g. range selection on design ports). Note that the usage of variables it is completely safe, from both simulation and synthesis point of view. However,

in order to avoid unwanted behaviour in simulation it is advisable to use them only internally to each `SC_METHOD/CTHREAD`. For synthesis, they have to be coherent with the module's behaviour (combinational or sequential).

The Enabler performs a sort of "address translation" in order to identify the target IP given a control word written to `ROW_0`. The translation is performed by subtracting -1 to the address written in the control word, since OBIPM ports are numbered from `0 to MAX`−1, while IPs start from 1 (IP 0 is the manager itself).

The interrupt controller is organized as a high-level FSM, and serves the interrupt one by one according to the priority policy described in chapter 1. Ideally, it evolves through these sequences of states:

1) Check for active transaction: if there is an active transaction, wait.
2) Check for an interrupt request and serve the first request.
3) Wait for the ACK.
4) Serve the next request, according to the priority.

Actions 2 and 4 are implemented by means of a for-loop. The crucial aspect is that, since the priority must be always respected, the loop is broken once a request is *acked*. This guarantees that, if a higher priority request arrives while a lower one is being served, the latter is served as soon as the lower priority is *acked*.

As final remark, the ports of the top level are parametrized with respect to the number of IPs. We have decided to group ports that perform the same functionality, but on different IPs (e.g. `DATA_IP_IN` ports). This allows writing the interface of the top-level once and letting the synthesis tool infer the necessary control logic over these ports. Once the RTL version is obtained, each port can be divided into ranges, and each range is assigned to a specific.

As a matter of example, consider port `DATA_IP_IN`. Normally this port is 16 bits wide. In our implementation there are `16 x NUM_IP` bits. Assuming two IPs actually instantiated, the width will be 32 bits. Bits 31 to 16 are assigned to IP 2, while 15 to 0 to IP 1. Similar reasoning applies to the other ports.

3. SystemC Test-bench

3.1 Introduction

The design was tested extensively by resorting to a dedicated test-bench, which is here covered in details. None of the modules presented in the following are needed in the deployment on the FPGA and are used only during design validation. The test-bench definition includes many calls to the standard output stream, which enables the designer to quickly test functionalities. A more in depth debugging can then be performed by using waveform viewers at both the behavioural (TLM or PIN abstractions) and register-transfer levels (strictly PIN).

3.2 Test-bench structure

The test-bench is comprised of 4 submodules. 3 `SC_MODULES`:

- Dummy_IP1: a sample IP which can be configured as a `READER`, `WRITER` or `INACTIVE`;
- Dummy_IP2: duplicate of the above;
- Bridge: also referred as Wrapper, this module connects the IPs with the OBIPM and vice-versa.

And one `SC_THREAD`:

- Buffer: the 64-word buffer used in the CPU-FPGA communications (modelled as an `SC_THREAD` behaving like a register file);

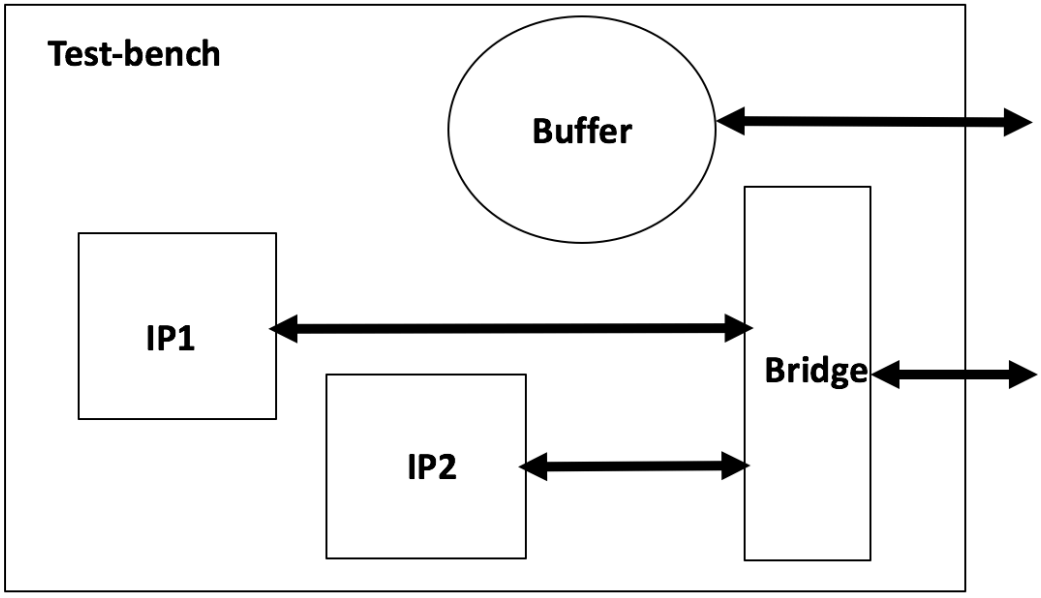


Figure 9 – Testbench structure, external IOs are connected to the OBIPM at the system level.

There are 3 main modes tests can be conducted. The designer can select which one before running simulation by defining one at a time in the `defines.h` file. Namely they are:

- `WRITE_TEST`: performs a simulation where an active IP writes 3 values to the buffer at each clock cycle;
- `READ_TEST`: an active IP performs 3 reads operations from the buffer at each clock cycle;
- `INTERRUPT_TEST`: The interrupt controller is tested by changing the value of the first word of the buffer over time i.e. by changing its value from `CW_INTERRUPT_TEST_BEGIN` to `CW_INTERRUPT_TEST_END`. In simulation runs, we can see how the interrupt controller changes from not being able to accept new requests to being available and running normally. Refer to section 3.3 for a dedicated interrupt controller testing.

In addition, the testbench is responsible for driving the active-low reset signal for all modules (both, its own and external ones, such as the OBIPM). For such reason it auto-samples its `rst` signal into `rst_in`. At simulation start, a dedicated `SC_CTHREAD` called `resetter` asserts such signal for 2 clock cycles before de-asserting it and letting the other modules run in their normal state.

3.3 The interrupt controller test-bench

A dedicated test infrastructure has been released to assess the correct functionality of the interrupt controller (IC). The reasoning behind this is related to the complex nature of such module. Hence, a more compact test-bench was devised and can be found under the `tb_IC` folder.

This project does not include the OBIPM, the buffer, the IPs, nor the Wrapper. In fact, the testbench interfaces directly with the IC.
After two reset cycles, two separate interrupt requests are sent to the IC and each change in signal is traced immediately to standard out with its timestamp. New tests can be deployed by simply changing the body of the `mc SC_THREAD` within the test-bench `tb` module.

3.4 Full system deployment

The testbench and OBIPM modules are instantiated by `System` (see `system.h` and `system.cpp`), which represents the highest block in the design hierarchy. Its purpose is to simply interconnect the two blocks. In turn, `System` is created in `main.cpp` where we have the `sc_main`, that is SystemC's equivalent to a C++ `main` function. Finally, the SystemC kernel is invoked by calling the `sc_start` function with a fixed end-time (in our case 1000 ns) as no other module in the hierarchy calls the `sc_stop` (which would force simulation end).

This approach, along with a well-structured Makefile transforms simulation into an effortless and repeatable task, which can be migrated to any machine hosting the SystemC library.

4. Stratus HLS

4.1 Introduction

Stratus HLS is the state-of-the-art tool for high-level synthesis (HLS). It has been in use in industry and academia for over a decade, delivering high-end solutions for hardware design. It is in constant development and new major updates are regularly released (to this date, the most recent version, that is 17.10.100, was released June 12th 2017).
In this project, Stratus was used to initially implement our design starting from the specifications (*Chapter 1 CPU-FPGA Communication Protocol*) and obtain the first set of RTL solutions. Subsequently, the design was easily migrated to Xilinx Vivado HLS with little to no effort.

4.2 The Stratus Approach

Stratus provides an integrated development environment (IDE) to get new users to quickly familiarize with the tool. There are 3 main sections:

- `Edit`: for viewing and editing the source and project files;
- `Analysis`: for analysing and studying a synthesized design under all aspects and for performing comparison among different synthesis runs;
- `Help`: to gain access to the User Manual and Reference Guide (which are both of great importance as there is no available information online).

A project is managed through the `project.tcl` and the `Makefile` files. This is a really easy yet powerful way to manage even complex designs and migrate a project from one machine to another.
In `project.tcl`, we define among other things, the DUT to be synthesized, how to synthesize it and which compiler to use. The `Makefile` makes it possible to automate the whole compilation and simulation process. Stratus then creates `Makefile.prj` which is a much more complex Makefile that should never be modified manually.

Besides using the IDE, more experienced designers may want to resort to using the tool via terminal. This is possible and highly encouraged as scripts can easily invoke the various tool features.

One of the most powerful features of Stratus is HW/SW co-simulation. The same SystemC test-bench, main and system modules (see *Chapter 3 SystemC Test-bench*) can be used to test the design under test (DUT) at both the behavioral (SystemC implementation) and lower levels (such as RTL or gate-level). In fact, Stratus will, in the compilation step, create a SystemC "wrapper" which encapsulates a Verilog description, making it possible to perform co-simulation.

Additionally, simulations at the behavioral level can be distinguished as being TLM or PIN-level with a simple one-liner (`define_io_config * TLM` directive and `–io_config PIN` option in `project.tcl`).

4.3 The project.tcl file

We here give a brief outline of the structure and reasoning behind the `project.tcl` file used for this implementation. New users should be able to get up and running with the whole project by simply opening such file in Stratus (`Open->Existing Project->/path/to/project.tcl`) or by calling Stratus from the command line in the folder where the project resides.

As a first step, technology library files should be defined:

```
set LIB_PATH "[get_install_path]/share/stratus/techlibs/GPDK045/gsclib045_svt_v4.4/gsclib045/timing"
set LIB_LEAF "slow_vdd1v2_basicCells.lib"
use_tech_lib   "$LIB_PATH/$LIB_LEAF"
```

Global synthesis attributes such as clock period and input delays:

```
set_attr clock_period           25.0
set_attr message_detail         3
set_attr default_input_delay    0.1
set_attr output_style_reset_all on
```

Simulation options, like SystemC version or simulator:

```
set_attr cc_options             " -DCLOCK_PERIOD=25.0 -g"
enable_waveform_logging -vcd
set_attr end_of_sim_command "make saySimPassed"
set_systemc_options -version 2.3 -gcc 4.1
# use_systemc_simulator incisive
```

Then, test-bench and system modules should be defined (i.e. modules that shall not be synthesized):

```
systemModule main.cpp
systemModule system.cpp
systemModule tb.cpp
systemModule wrapper.cpp
systemModule ip.cpp
```

The `define_hls_module` directive defines modules to be synthesized, whereas `define_hls_config` sets the constraints and optimizations that the HLS engine shall take into considerations (we here report just a subset).

```
define_hls_module obipm obipm.cpp
define_hls_config obipm BASIC
define_hls_config obipm DPA_1       --dpopt_auto=op
define_hls_config obipm DPA_2       --dpopt_auto=expr
```

```
define_hls_config obipm DPA_3       --dpopt_auto=array
define_hls_config obipm DPA_4       --dpopt_auto=op,expr
define_hls_config obipm DPA_5       --dpopt_auto=op,array
define_hls_config obipm DPA_6       --dpopt_auto=array,expr
define_hls_config obipm DPA_ALL       --dpopt_auto=all
define_hls_config obipm CSE         --comm_subexp_elim=on
define_hls_config obipm FLAT        --flatten_arrays=all
define_hls_config obipm ST_BIN      --global_state_encoding=binary
define_hls_config obipm ST_1HOT     --global_state_encoding=one_hot
```

Verilog-level simulation and power configurations for can be set next.

```
define_sim_config B
foreach config [find -hls_config *] {
  set cname [get_attr name $config]
  define_sim_config ${cname}_V "obipm RTL_V $cname"
  define_power_config P_${cname} ${cname}_V -module obipm -command bdw_runjoules
}
```

Finally, options for gate-level simulation are set.

```
define_logic_synthesis_config RC {obipm -all} \
    -command "bdw_runrc" \
    -options \
        [list BDW_LS_TECHLIB "$LIB_PATH/$LIB_LEAF"] \
        {BDW_LS_CLK_GATING normal} \
        {BDW_LS_NOGATES 1}
```

5. Vivado HLS

5.1 Introduction

Vivado HLS is a tool for high-level synthesis provided by Xilinx. It allows synthesis from C/C++ and SystemC sources and provides the VHDL/Verilog and modified SystemC as output. As previously mentioned, we have chosen SystemC as design language as it allows to easily handle a cycle-accurate design.

The following picture shows the Vivado design flow:

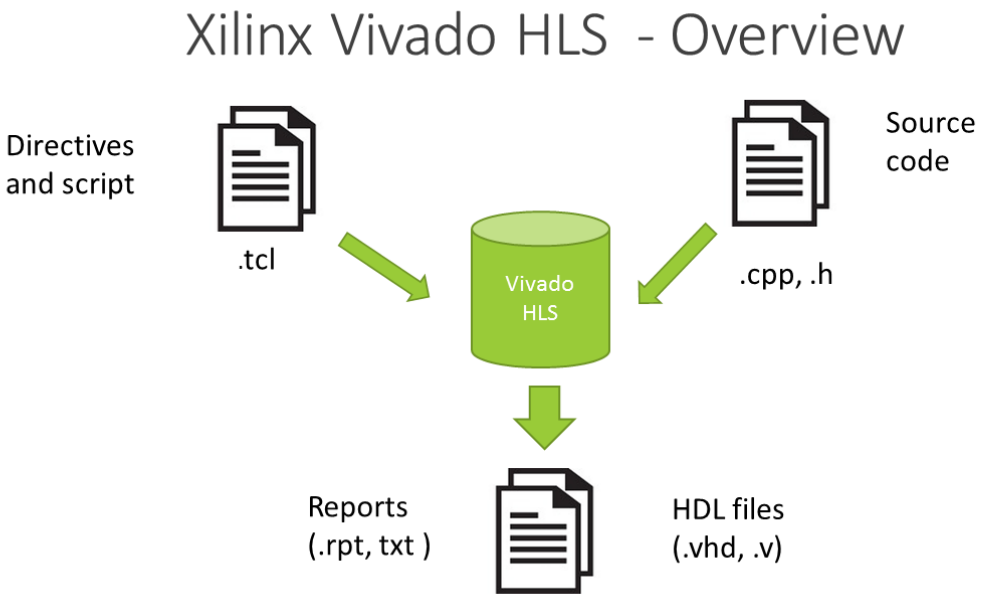


Figure 10 – Vivado HLS design flow

5.3 Directives and optimizations

The developed design is *generic* (in strictly HDL terms), so it allows to have several different HLS syntheses with different numbers of IPs. In order to obtain the desired design Vivado HLS allows to set a group of directives. In our case the most relevant ones are related to the interfaces. If there are not user definitions, the tool optimizes the interfaces, in most of the cases, changing the desired behavior. Moreover, other directives on the clock and reset signals are necessary to achieve any result.

An example of some directives applied during development:

```
############################################################
## This file is generated automatically by Vivado HLS.
## Please DO NOT edit it.
## Copyright (C) 1986-2017 Xilinx, Inc. All Rights Reserved.
############################################################
set_directive_interface -mode ap_ctrl_none "obipm"
set_directive_interface -mode ap_ctrl_none "obipm::enabler"
set_directive_interface -mode ap_ctrl_none "obipm::packet_switcher"
set_directive_interface -mode ap_ctrl_none "obipm::interrupt_controller"
set_directive_interface -mode ap_ctrl_none "obipm::support_logic"
```

Figure 11 – Vivado HLS directives

Other settings such as the ones for the clock and the reset signals are *inlined* within the SystemC code.

N.B. Specifying directives is crucial, not only to optimize your design, but because whenever Vivado HLS, does not find any related to a port, it changes the interfaces implementation. This may cause inconsistencies and misbehaviors in the resulting RTL.

5.4 Design Validation

Vivado HLS offers two main validation approaches:

- High-level validation using C/C++ co-simulation.
- RTL level validation.

The two approaches are different in terms of effort in developing test-benches and in results.
The former, co-simulation, allows to test the design using high-level languages, so developing a test-bench requires less effort. Moreover, the results are often inaccurate since the resulting HDL will not always reflect the specifications (due to Vivado-inferred interface protocols). The second approach, requires a significant design effort for developing the test-bench since a deeper knowledge of the resulting RTL is required. The main advantage of this approach is in terms of results: a deeper knowledge of the produced RTL code allows to build a complete test-bench, which yields more accurate results. In terms of simulation time, in our design the two approaches are comparable. We decided to adopte the second approach.

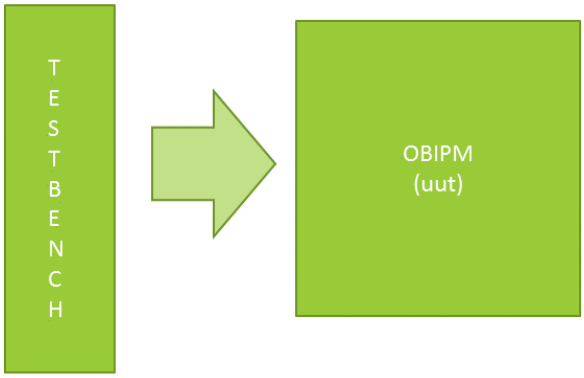


Figure 12 – Simple validation OBIPM validation in Vivado HLS

Starting from the protocol definition (Section 1), which constitutes our specification, we observed all signals.

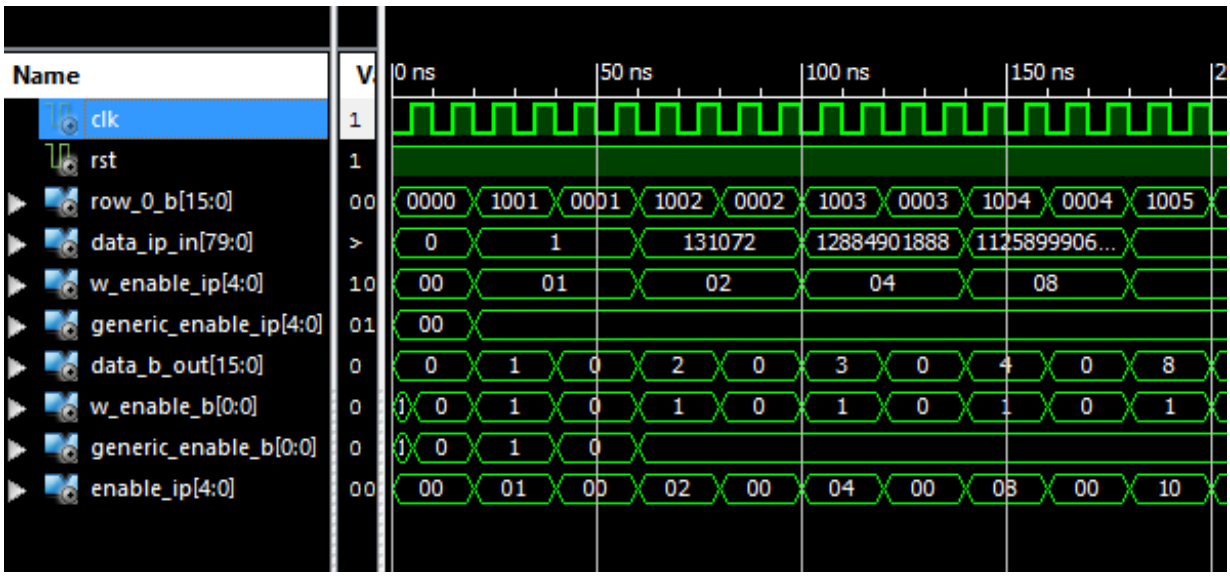


Figure 13 – RTL validation waveform with 5 IPs.

An example of steps which allowed to test our design:

1. Forcing `row_0_b` with LSB to '1' and the bit 12 to '1' in order to select IP1;
2. At the same time, forcing the `w_enable_ip` and the `generic_enable_ip` at one.

In output we can see, that the `enable_ip` is set to '1', that means IP1 is selected and the data from the buffer (`data_ip_in`) in input was forwarded to the output (`data_b_out`).

3. Writing the control word to stop the transaction and observing the output `data_b_out`, that is zero.

All operations are repeated for the other IP cores. Using these steps, the design was validated and behaved as expected.

5.5 Vivado HLS How to

In order to reproduce the adopted design flow with Vivado HLS, the following steps are required:

1. File > New Project.
2. Insert the project name and add the specification (C/C++, SystemC).
3. Insert the directives for the optimizations. Two options are available:
    a. In-code directives. Using the GUI and clicking on the desired statement in the Directive View.
    b. Using a .tcl file that is running with the synthesis script.
4. Run the synthesis and collect the resulting HLD.

Point 3 is the most crucial part. The most important directives used in our design are related to the interface. In particular, use `ap_ctrl_none` if you want Vivado to avoid adding other kind of interfaces[2].

In order to run the synthesis (point 4) another .tcl file is required. An example is showed in the following picture:

```
open_project SysC_try
set_top obipm
add_files Documents/AtlasRep/stratus/defines.h
add_files Documents/AtlasRep/stratus/obipm.cpp
add_files Documents/AtlasRep/stratus/obipm.h
open_solution "solution1"
set_part {xa7z030fbg484-1i} -tool vivado
create_clock -period 10 -name default
source "./SysC_try/solution1/directives.tcl"
#csim_design
csynth_design
#cosim_design
export design -format ip catalog
```

Figure 14 – Sample directives.tcl file

[2] https://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_1/ug902-vivado-high-level-synthesis.pdf

6. Lattice Diamond Integration

6.1 Introduction

The purpose of this chapter is to provide details on how the VHDL or Verilog code generated from the HLS tools were integrated into Lattice Diamond. We also briefly provide some analysis results.

6.2 Design Integration into Lattice Diamond

The software tool is organized into a project that consists of:

- HDL source files.
- EDIF netlist files.
- Synthesis constraint files.
- LPF constraint files.
- Script files for simulation.
- Analysis files for power calculation and timing analysis.

The project data is organized into implementations, which define the project structural elements and strategies (these are collections of tool settings and states on how the implemented design should be run). There can be multiple implementations of the project with the same associated active strategy file constraint or each with its own.

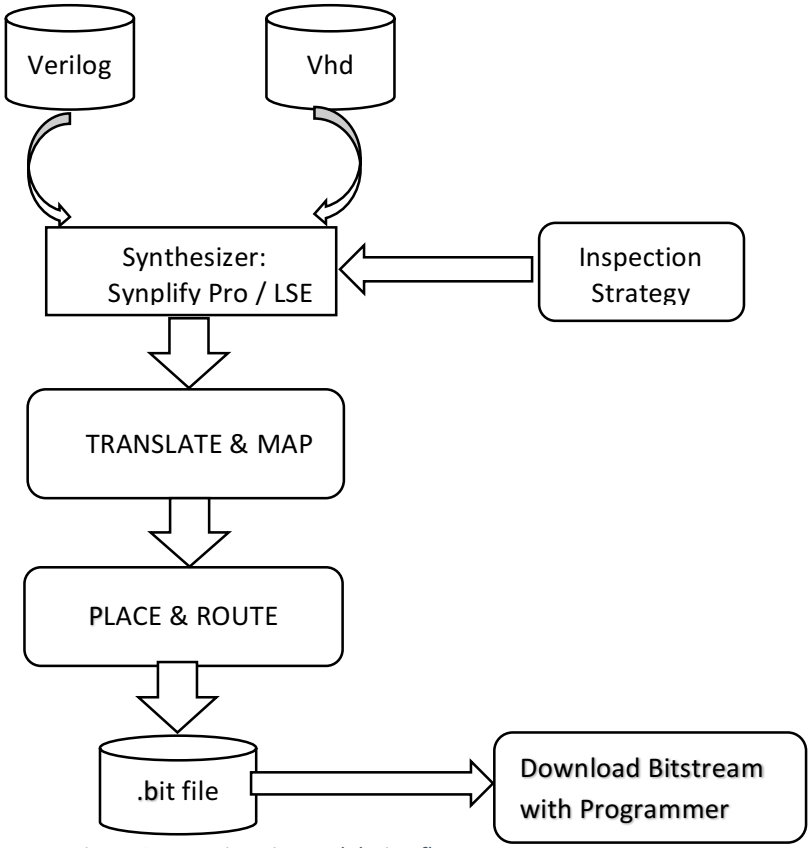


Figure 15 – Lattice Diamond design flow

The design integration from the designs produced with Stratus and Vivado are the first step in the design flow (Figure 15).

We then used two different synthesis tools to have better exploration of the design alternatives. The tools are:

- Lattice Synthesis Engine (LSE): from Lattice Semiconductors and so optimized for use with Lattice devices.
- Synplify Pro: (by Synopsys) for producing high-performance, cost-effective FPGA designs
  Which can also perform high-level optimizations before synthesizing the RTL code into a specific FPGA logic.

After the design has been synthesized the process of translating and mapping the design to the specific target FPGA takes place. It converts the EDIF file output from synthesis to NGD format and produces a mapped Native Circuit Description (.ncd) design file.
Then, the place and route process takes a mapped physical .ncd design file and creates a physical layout of the design implementation. Finally, the actual bitstream generation takes place. This file contains all the implementation's configuration information, which defines the internal logic and interconnections of the FPGA.

6.3 Pareto Analysis Results

The analysis reported in the figure 16 has been performed by comparing the RTL produced from Vivado with that from Stratus according to the performance point of view. We used Synplify Pro to perform synthesis only on the combinational part of the design (enabler and packet switcher blocks).
There are 3 designs per HLS tool, one with a different number of IPs (2, 3, 5). It is easily observable that the Vivado implementations perform better (higher clock frequency achievable).

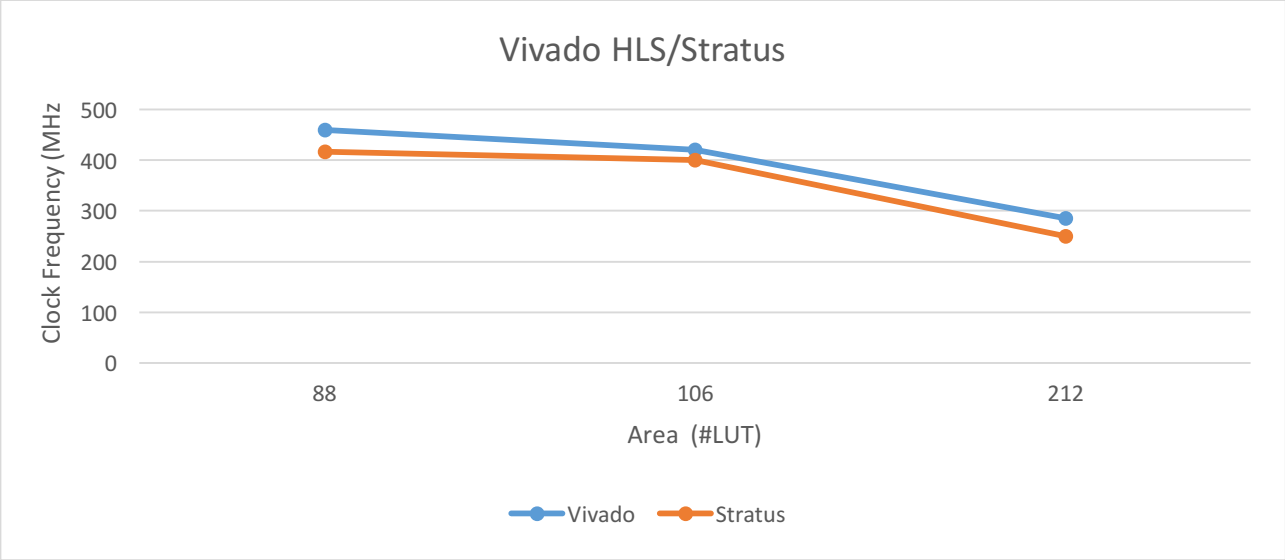


Figure 16 – Clock frequency vs Area plot

Section 3: Future Works

Future works

The most important conclusion from the performed experiment is that HLS can be integrated with the SEcube toolchain, but the process is not trivial. An in-depth knowledge of the HLS tool is required in order to model the system correctly and reach an RTL implementation with a relatively low effort.

By analysing the two tools, it is clear that Stratus offers more features for optimizing the design while Vivado HLS mainly produces FPGA-dependent RTL. On the other hand, for simple designs, Vivado is more user-friendly than Stratus and it is relatively easier to get a working RTL.

From the gathered results it is not clear which of the two tools under analysis performs better. We believe that this is due to the design under analysis (OBIPM). The system modelling style is very close to RTL, giving very few degrees of freedom to the tools to optimize and derive different micro-architectures.

We were not able to fully synthesize the design, due to the complex FSM of the interrupt controller. Despite that, we believe the next step toward the full integration of HLS with SEcube should concern how different design style are actually handled by the two tools and how the resulting RTL can be integrated on Lattice logic synthesis tools. In our opinion, a larger set of input benchmark should be evaluated, both data-dominated and control-dominated circuits. In order to really take advantage of the features offered by HLS tools these circuit should be closer to algorithmic-level. By doing so, the differences between the two tool should become more evident.

Changing the input languages could be an interesting area of exploration. Most of our design choices were actually limited due to SystemC's characteristics. Indeed, some of the scheduling algorithms that are the core of the back-end of such tools might perform better (e.g. C language).

However, the idea of using SystemC should not be completely discarded. Indeed, by exploring different design styles, a hypothetical IP-provider could leverage SystemC in order to provide both an interface compatible with the protocol defined in this project and the actual algorithm that the IP should implement. This could be achieved by mixing cycle-accurate and algorithmic-level modelling style, a feature offered by SystemC only. The great advantage is that, if the aforementioned design flow would be successful, there is no need to develop a RTL wrapper and integrate it with the RTL IP from generated via HLS, but the whole process can be done in a single SystemC design.

Finally, most of the difficulties have been found in validating the resulting RTL, mainly due to improper usage of some constructs and tools limitations (poor support for SystemC, starting from the documentation). From this point of view it would be definitely a good choice to migrate to Catapult HLS from Mentor Graphics. The tool embeds some functionalities related to the pre-synthesis verification, via automatic formal checking techniques, which allows to find bugs in the design before starting the HLS process. It also allows to verify the final RTL against the C/C++/SystemC model by resorting to formal verification equivalence. Last but not least, the tool generates RTL which does not target any specific device (as Vivado HLS does). Hence, the probability of having a fully working design flow, compatible with SEcube, could be higher than with other tools.

Appendix A: Project Methodology, Deliverables and Milestones.

A.1 Introduction

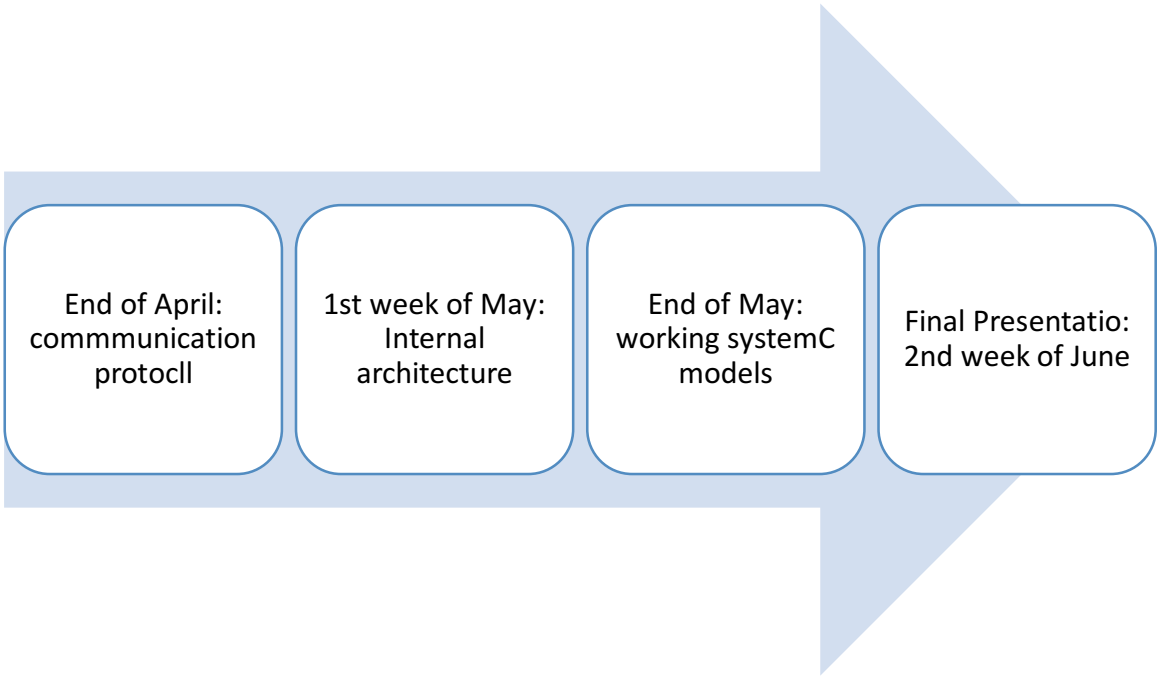
With the following appendix we aim at describing what our deliverables were, how we divided the task among us, and how we arranged our activities to meet the final deadline.

A.2 Project Methodology

The first part of the project was devoted to analysing the problem in order to find the best solution to develop the requested system. During this first phase, we worked in tight collaboration with the other teams in order to define the specifications of the system, characterizing the behaviour we were going to model. The result of this first phase is detailed in Section 1 of this report.

In parallel, we internally devised a strategy in order to meet the deadline, i.e. project presentation on 2nd or 3rd week of June. We applied the so called Backward design (or planning)[3], that is we started from defining our end-goal and then we backward planned our activities, setting up intermediate deadlines and checkpoints so that we could meet the final deadline.

For doing such kind of planning, we leveraged Gantt charts available on Redmine. The following picture summaries the planned activities:



[3] Backward design: https://en.wikipedia.org/wiki/Backward_design

We knew that (based on our previous experiences) it would have been better to have at least two full weeks for synthesizing the design, verifying the RTL and deriving some conclusions. Thus, once the main deadline was fixed, we set up intermediate deadlines for end of May (working implementation), 1st week of May (for internal architecture) and end of April (communication protocol).

We also divided the activities among us in order to reach those deadlines. We all collaborated for achieving the first two deadlines (communication protocol and internal architecture). Then we divided the workload as follow:

- Andrea Floridia (team leader): SystemC implementation and test-bench development.
- Robert Margelli: SystemC implementation, test-bench development, Stratus HLS.
- Luca Mocerino: Vivado HLS, Lattice Diamond integration, RTL verification.
- Leonel Ngongang: Vivado HLS, Lattice Diamond integration, RTL verification.

Since one of the member of the team was abroad for pursuing his master's thesis, we extensively used GIT and Dropbox, this made it possible to collaborate efficiently despite the different time zones.

We arranged Skype meetings in order to exchange ideas and kept in touch by also using text messaging applications such as WhatsApp and Telegram.

A.3 Milestones

April 24th: Communication protocol CPU-FGPA.

May 7th: Definition of the internal architecture of OBIPM.

May 25th: Fully working SystemC models.

June 8th: Last RTL verified.

A.4 Deliverables

We successfully implemented the initial specifications in SystemC, and a fully working simulation environment has been deployed. Hence, it is possible to extend and evaluate the functionalities of the OBIPM without the need of working directly with the RTL. It is possible to extend the model in order to perform hardware/software co-design.

Along with the source file of the design, we provide a modular test bench that allows testing the functionalities individually.

We successfully verified a portion of the final RTL of the OBIPM (namely Enabler and Packet Switcher). Due to synthesis issues we could not verify the Interrupt Controller. An initial design flow is also provided, which embrace both HLS and logic synthesis with Lattice Diamond.

Appendix B: Modelling guidelines

With the following we aim at providing some useful guidelines that allows to write SystemC code for both Vivado HLS and Stratus HLS.

Vivado HLS:

- Avoid using arrays within a module, they are synthesized as SRAM blocks.
- For-loops should have as first statement a `wait()` call (due to the internal scheduling algorithm).
- Complex mathematical functions are synthesized using DSP blocks available on Xilinx FPGA.

Stratus HLS:

- `sc_lv` are not synthesizable, use `sc_bv` instead.
- Use `sc_bv` even for single bit signals.
- Resort to the `HLS_DEFINE_PROTOCOL` directive when `wait()` statements shall be preserved in the RTL description as clock cycles. The main reasoning is to apply a protocol block at the interface of an `SC_CTHREAD` and break the protocol in its data-path region.
- Prints to standard output are not reliable in Verilog-level simulations.
- As the project progresses, run synthesis regularly to check for possible errors due to erroneous coding styles.

Common issues:

- Reset issue: when modelling with `SC_CTHREAD` be sure that reset statements are separated by a `wait()` statement from the operational statements (i.e. the while-loop).