Politecnico di Torino

Politecnico di Torino

Master of Science in Electronic Engineering

A.y. 2024/2025

Graduation Session : December 2025

Development of a programmable hardware command in the context of high speed serial links

Supervisor:

Maurizio Zamboni

Candidate:

Giusy Pascale

s318920

Abstract

In the era of technological innovation and increasing performance demands, input/output (I/O) systems play a significant role in data exchange between the components of electronic systems. Applications such as artificial intelligence, 5G networks, high-performance computing (HPC), and advanced automotive systems (ADAS) require faster and more efficient solutions for transmitting and receiving large amounts of data.
In this context, high-speed SerDes (High-Speed Serializer/Deserializer) systems have become a key technology, they currently represent the predominant implementation of I/O interfaces capable of supporting data transmission rates exceeding 100 Gbps.
A SerDes system is mainly composed of two functional blocks: the transmitter and the receiver. The transmitter converts parallel data into a high-speed serial stream, whereas the receiver deserializes the incoming stream to restore the original parallel format.
The SerDes includes a Physical Medium Dependent (PMD) section, responsible for the overall control of the components through multiple finite state machines (FSMs). It manages power-up and power-down requests, rate change operations, provides the interface with the firmware and performs limited data conditioning functions.

The focus of this project, in collaboration with Synopsys, is on the power-state management within the PMD. There is a Look-Up Table (LUT) storing the instructions and configuration data required to handle power state transitions efficiently. Nowadays each command is mapped to a specific signal, that is conditioned by some FSMs. This approach is highly rigid, as any modification of a command requires rewriting the hardware description or, in the worst case, re-fabricating the silicon.

The main goal is having a programmable hardware command in order to have more flexibility.
Firstly, a new module was created. It manages both the existing hardware commands and new ones, with the exception of one that has a dedicated FSM.
Then the module was instantiated in one of the modules in the PMD that handles the power up/down sequences. The old and the new implementations were simulated to actually check that the behavior was correct and equal. Finally, the synthesis of both has been done in order to be able mainly to compare the differences in area and power.

Table of Contents

List of Tables III

List of Figures IV

1 Introduction and Theoretical Background 1
- 1.1 High-Speed SerDes . . . 1
  - 1.1.1 Types of SerDes . . . 3
  - 1.1.2 SerDes Architecture . . . 3
- 1.2 Finite State Machine (FSM) . . . 5
  - 1.2.1 Bubble Diagram . . . 6
- 1.3 Digital Design Flow . . . 7
- 1.4 Thesis project: Programmable Hardware Command . . . 8

2 Methodology 10
- 2.1 Project specifications and design . . . 10
  - 2.1.1 Interface of the module . . . 11
  - 2.1.2 Base assign . . . 14
  - 2.1.3 Conditional assign . . . 15
  - 2.1.4 Handshake . . . 16
  - 2.1.5 Time wait . . . 19
  - 2.1.6 Signal pulse . . . 20
- 2.2 Design Verification . . . 21
  - 2.2.1 Design Checks: Lint, CDC and RDC . . . 21
  - 2.2.2 Functional Verification: Testbench and Verdi . . . 23
- 2.3 Integration in product . . . 24
- 2.4 RTL Synthesis and Optimization in Design Compiler NXT . . . 25
  - 2.4.1 Constraints . . . 27
  - 2.4.2 Formality . . . 30
- 2.5 Power estimation: PrimePower . . . 31

3 Results 32
3.1 Simulation with Verdi 32
3.1.1 PMD_HW_CMD_FSM module 32
3.1.2 Integration in product 34
3.2 Synthesis report 36
3.2.1 Report timing and clock 37
3.2.2 Report area 39
3.2.3 Report power from Design Compiler NXT 40
3.3 Power estimation report 41

4 Conclusions and future perspectives 45

Bibliography 47

List of Tables

2.1 Actual organization 10
2.2 Possible commands 11
2.3 Future Organization 11
2.4 Complete set of parameters. 13
2.5 arg field for `BASE ASSIGN` command 14
2.6 arg field for `COND ASSIGN` command 15
2.7 arg field for `HANDSHAKE` command 16
2.8 arg field for `GENERIC HANDSHAKE` command 17
2.9 arg field for `CALIBRATION HANDSHAKE` command 18
2.10 arg field for `TIME WAIT` command 19
2.11 arg field for `SIG PULSE` command 20

3.1 Clock information 38
3.2 Clock Gating Summary - Original `PMD_TX_PWR_CTL` 38
3.3 Clock Gating Summary - New implementation of `PMD_TX_PWR_CTL` . 38
3.4 Area report - Original `PMD_TX_PWR_CTL` 39
3.5 Area report - New implementation of `PMD_TX_PWR_CTL` 39
3.6 Power report from DC - Original `PMD_TX_PWR_CTL` 40
3.7 Power report from DC - New implementation of `PMD_TX_PWR_CTL` . 40
3.8 Power report - Original `PMD_TX_PWR_CTL` 41
3.9 Power report - New implementation of `PMD_TX_PWR_CTL` 42
3.10 Report threshold voltage - Original `PMD_TX_PWR_CTL` 43
3.11 Report threshold voltage - New implementation of `PMD_TX_PWR_CTL` 44

List of Figures

1.1 SerDes implementation. Source: Synopsys, “What is SerDes?”, [1]. 2
1.2 Parallel vs Serial. Source: Synopsys, “What is SerDes?”, Synopsys Glossary, [1]. 2
1.3 PMD Scheme 5
1.4 Moore vs. Mealy FSM block diagrams. Source: Figure 1.3,1.4 of [4]. . . 6
1.5 State Diagrams for (a) Moore and (b) Mealy Models Source: Figure 9.5 of [6]. 7

2.1 Interface of the module 12
2.2 Timing Diagram of the `BASE ASSIGN` Command 15
2.3 Timing Diagram of the `COND ASSIGN` Command 16
2.4 Timing Diagram of the `GENERIC HANDSHAKE` Command 17
2.5 Timing Diagram of the `CALIBRATION HANDSHAKE` Command 19
2.6 Timing Diagram of the `TIME WAIT` Command 20
2.7 Timing Diagram of the `SIG PULSE` Command 21

3.1 Waveform of the `BASE ASSIGN` command 32
3.2 Waveform of the `CONDITIONAL ASSIGN` command 33
3.3 Waveform of the `GENERIC HANDSHAKE` command 33
3.4 Waveform of the `CALIBRATION HANDSHAKE` command 33
3.5 Waveforms of the `TIME WAIT` command 34
3.6 Waveform of the `SIGNAL PULSE` command 34
3.7 A frame of the integration waveforms 35
3.8 A frame of the integration waveforms 35

Chapter 1

Introduction and Theoretical Background

In the era of technological innovation and increasing performance demands, input/output (I/O) systems play a significant role in data exchange between the components of electronic systems. In this context, high-speed SerDes (*High-Speed Serializer/Deserializer*) systems have become a key technology.
This introduction chapter addresses four main topics: **High-Speed SerDes**, **Finite State Machines (FSMs)**, **Digital Design Flow** and the **Thesis project**, which builds upon these concepts.

1.1 High-Speed SerDes

As described on the Synopsys website, a "SerDes is a functional block that Serializes and Deserializes digital data used in high-speed chip-to-chip communication. Modern SoCs for high-performance computing (HPC), artificial intelligence (AI), automotive, mobile, and Internet-of-Things (IoT) applications implement SerDes that can support multiple data rates and standards like PCI Express (PCIe), MIPI, Ethernet, USB, USR/XSR." [1]
A SerDes system is mainly composed of two functional blocks: the transmitter and the receiver. The transmitter converts parallel data into a high-speed serial stream, whereas the receiver deserializes the incoming stream to restore the original parallel format. Figure 1.1 shows how the parallel data stream is turned into a serial stream thanks to the use of a SerDes.

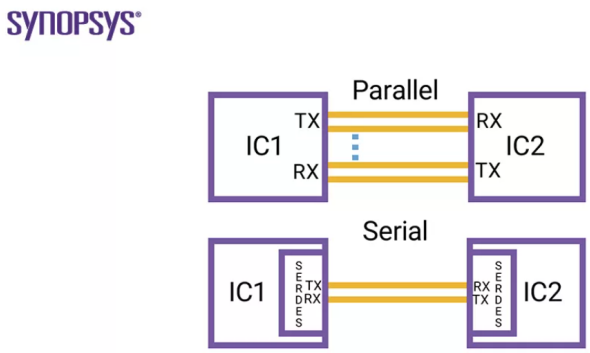


Figure 1.1: SerDes implementation. Source: Synopsys, "What is SerDes?", [1].

In the past, parallel data transmission was the most common approach. However, as data rates increased, this method started to show significant limitations.
One issue is skew, which occurs because each signal line has slightly different delays, making synchronization more difficult at higher speeds. Another problem is crosstalk, where adjacent lines interfere with each other, degrading the overall signal integrity.
This data conversion is also necessary because "In a parallel transfer, all bits move at once from source to destination. In a serial transfer, the bits are sent one at a time. That makes a parallel transfer faster but also requires multiple lanes or paths, one per bit. Parallel transfers are more expensive as they require more hardware.[...] Serial data transfers require only a single path or cable so less circuitry is needed." [2]. For these reasons, the main goal of SerDes is to minimize the number of interconnections.

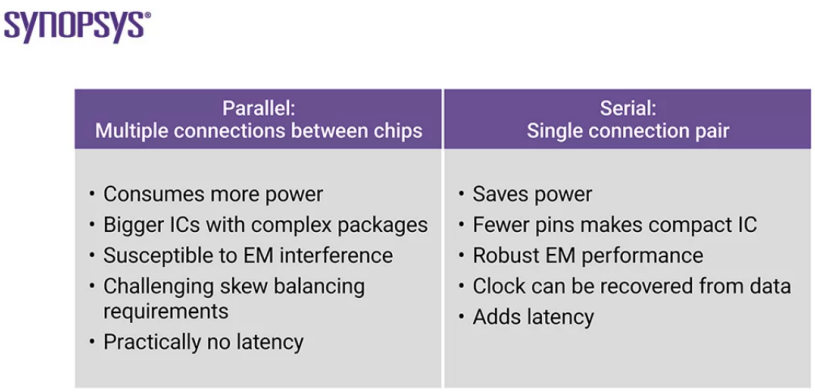


Figure 1.2: Parallel vs Serial. Source: Synopsys, "What is SerDes?", Synopsys Glossary, [1].

Figure 1.2 shows a table summarizing the main differences between the two options to transfer data between chips. Thus, the serial data transfer offers advantages including reduced power consumption, good resistance to electromagnetic interference and simplified packaging.

1.1.1 Types of SerDes

SerDes systems are often designed to group multiple transmission and/or reception channels in a single device. These channels are known as lanes, each one operates independently. The grouping of channels allows to share some circuits (e.g. the PLL), and therefore the resulting block is more efficient in terms of chip area, cost and power.
There are three different modes of communication: simplex, half-duplex and full-duplex. For this reason, it is possible to distinguish three designs of the SerDes.

- SerDes Simplex: it has a one-way connection that works either for transmission or reception.
- SerDes Half Duplex: it has a channel that allows transmission in both directions, but alternately over time.
- SerDes Full Duplex: it has a bidirectional electrical interface, which supports transmission and reception simultaneously and independently.

The choice to use one type rather than another depends on the application context, based on the communication protocol (Ethernet, USB, HDMI, etc.), modulation technology (NRZ, PAM4), etc. This directly influences the complexity of the SerDes design.

1.1.2 SerDes Architecture

In high-speed serial links, SerDes is the fundamental building block of a physical layer (PHY) and works together with the Physical Coding Sublayer (PCS) [1]:

$$\text{SerDes} + \text{PCS} = \text{PHY}$$

The Physical Coding Sublayer is a digital logic block that interfaces with the SerDes circuits. Its main functions are to prepare data for transmission by encoding and scrambling it, and to process received data by descrambling and decoding it. So the PCS makes sure that the digital signal is ready to be sent over the physical medium. (The PCS is not the focus of this work)

Different protocols recommend a variety of abstraction division approaches for a PHY.

Providing a detailed overview of the typical structure of a SerDes, the transmitter path includes several functional blocks. The Parallel-to-Serial Converter changes the parallel data into a serial bitstream. A Data Encoder or Scrambler makes sure there are enough signal transitions for reliable clock recovery. Pre-emphasis and equalization circuits are used to compensate for high-frequency losses and channel attenuation before transmission. Finally, the Clock Multiplier Unit (CMU) generates the high-speed serial clock from a lower-frequency reference clock.
On the receiver path, the Clock and Data Recovery (CDR) circuit extracts the clock in the incoming serial stream, making sure the data is sampled correctly. The Equalizer, which can be used as a Continuous-Time Linear Equalizer (CTLE) or a Decision Feedback Equalizer (DFE), reduces the effects of signal distortion and improves signal quality. The Data Decoder or Descrambler puts the original bit sequence back together, and the Serial-to-Parallel Converter restores the original parallel form.

In addition to the main transmission and reception chains, a SerDes architecture incorporates several common blocks that facilitate its overall operation. These include the Phase-Locked Loop (PLL), which generates stable, synchronised internal clocks, and the Clock Distribution Network, which ensures these clocks are properly aligned across the different functional units. The control logic manages configuration, calibration and monitoring tasks. Finally, test and loopback circuits are incorporated to verify link performance. [3]

The focus of this work is another important block: the Physical Medium Dependent (PMD). This section is responsible for controlling the components through multiple finite state machines (FSMs). Its functions include managing power-up and power-down requests and rate change operations, providing an interface with the firmware, and performing limited data conditioning.
The following picture illustrates only a subset of the blocks contained within the previously described component. A request reaches the PMD and is processed by some logic that ensures that the signals go to activate the right blocks.

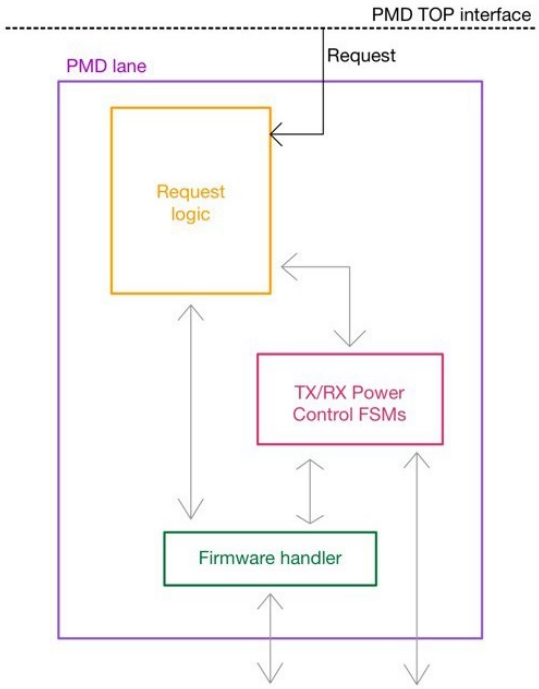
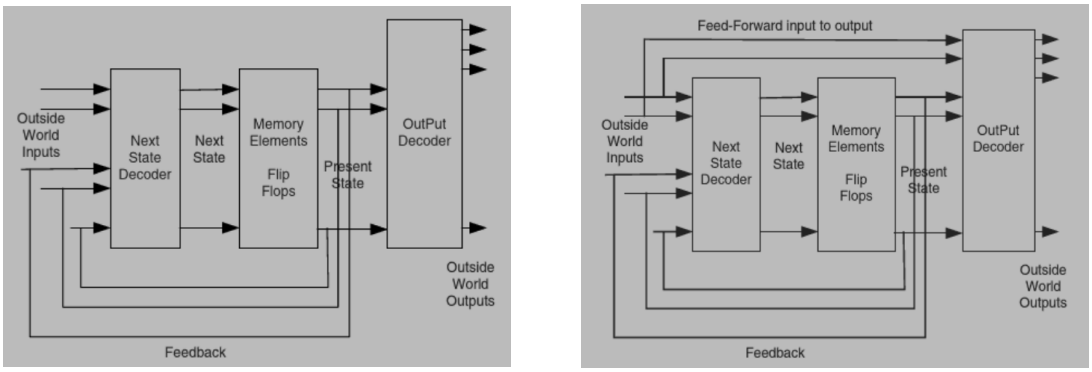


Figure 1.3: PMD Scheme

1.2 Finite State Machine (FSM)

"An FSM is a digital sequential circuit that can follow a number of predefined states under the control of one or more inputs. Each state is a stable entity that the machine can occupy. It can move from this state to another state under the control of an outside-world input." [4]
It can be classified as synchronous if its state transitions are controlled or synchronized by a clock signal. A machine that functions independently of a clock signal is designated as asynchronous.
Two main models of finite state machine can be distinguished: the Moore machine and the Mealy machine. In a Moore machine, the output depends only on the present state, whereas in a Mealy machine, the output is determined by both the current inputs and the present state. Hybrid architectures may also exist, where some outputs follow the Moore model and others follow the Mealy model.
Below are two schemes illustrating the two different types of FSMs described above.



(a) Block diagram of a Moore FSM **(b)** Block diagram of a Mealy FSM

Figure 1.4: Moore vs. Mealy FSM block diagrams.
Source: Figure 1.3,1.4 of [4].

In general, an FSM has a limited number of states. For N variables, there are between 2 and 2^N possible states. The present state of a state machine is defined by the values stored in the flip-flops within its sequential section, while the next state is determined by the combinational logic that controls the state transitions. A machine is typically defined by the number of inputs and outputs, the initial state and the relationship between the present and next states. [5]

1.2.1 Bubble Diagram

The *bubble diagram* is an effective graphical representation of a finite state machine. It can be described as a directed graph, where the nodes (represented by circles containing state identifiers) correspond to machine states and the edges denote transitions between them. Each transition is labeled with the input conditions that cause a change of state. In the case of a Moore machine, the outputs are indicated within the circles, immediately following the state label. In the case of a Mealy machine, they are indicated alongside the transition lines, following the inputs.
The Figure below shows this representation method for both Moore machines and Mealy machines.

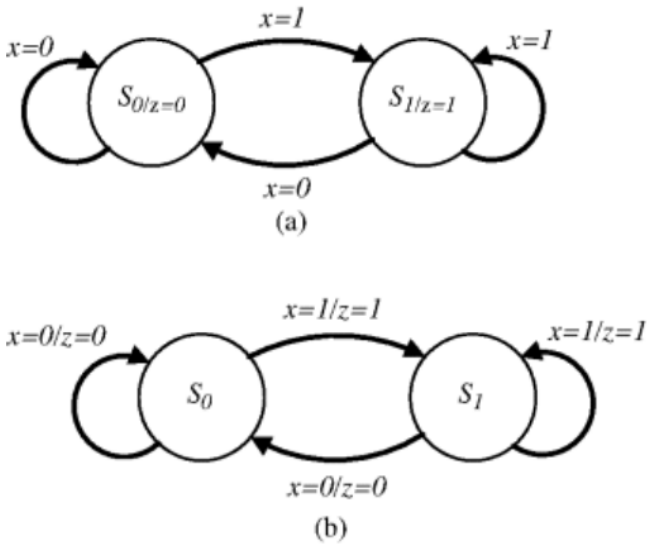


Figure 1.5: State Diagrams for (a) Moore and (b) Mealy Models
Source: Figure 9.5 of [6].

1.3 Digital Design Flow

The design of a digital circuit is typically characterized by a well-defined sequence of steps. The main stages of this process are described below. [7]

- **Pen & Paper design**: analysis of the problem, definition of the architecture and of the expected behavior.
- **RTL description**: the design is implemented at the register-transfer level (RTL) using Verilog or VHDL languages.
- **Linting, CDC and RDC static checks**: static analysis tools are used to detect issues relating to coding style, potential functional inconsistencies, and structural design problems. This stage usually involves linting and checks for Clock Domain Crossings (CDCs) and Reset Domain Crossings (RDCs). The aim is to detect risks of metastability, missing or incorrect synchronizers, and improper interactions within the reset domain.
- **RTL functional simulation**: a dedicated testbench is used to test the design functionality.
- **Synthesis**: transformation of the RTL description into a gate-level netlist.
- **Post-synthesis simulation**: the synthesized netlist is verified in accordance with the original RTL design.

- **Power estimation**: accurate power analysis is performed, based on the gate-level netlist and switching activity information extracted from post-synthesis simulations (e.g., FSDB files) to estimate both dynamic and static power.
- **Floorplanning**: definition of the physical layout regions for the main functional blocks, with the aim of optimizing performance and routing.
- **Place & Route**: the logic cells are placed and the connections routed to create the final layout.

1.4 Thesis project: Programmable Hardware Command

My thesis project, conducted in collaboration with Synopsys, focuses on power-state management within the PMD, a block of the SerDes.
A Look-Up Table (LUT) stores the instructions and configuration data required to handle power state transitions efficiently. Currently, each command is mapped to a specific signal that is conditioned by some finite state machines (FSMs).
This approach is highly rigid, as modifying a command requires either rewriting the hardware description or, in the worst case, re-fabricating the silicon.

The main goal is to have programmable hardware commands to allow for greater flexibility.

Firstly, a new module was created, following the steps described in Section 1.3. This module is an FSM that manages both existing and new hardware commands, except for one that has dedicated management. The module was then instantiated in one of the PMD modules that handles the power-up/down sequences. Looking at Figure 1.3, the block is the TX/RX power control FSMs, specifically the part related to the transmitter, which will be referred to as `PMD_TX_PWR_CTL`.
The old and new implementations were simulated to verify that the behavior was correct and consistent.
Finally, a synthesis of both versions of the `PMD_TX_PWR_CTL` was performed to make a comparison of the differences in area and power.
The final two steps described in Section 1.3, i.e. Floorplanning and Place & Route, were not included in this study.

The content presented in this chapter provides the essential background for understanding the development of the thesis work. There are three additional units.
The **Methodology** Chapter [2] provides a detailed description of the project specifications, the expected system behavior, the tools adopted during the development

process, and the analyses carried out to support the design choices.
The **Results** Chapter [3] presents and discusses the outcomes obtained through the applied methodology.
The **Conclusions and future perspectives** Chapter [4] summarises the main results of the work and outlines potential future developments.

Chapter 2

Methodology

This chapter outlines the methodology used to develop and validate the thesis project. The process is presented following the main stages of a standard digital design flow: project specification and design, design checks, simulation, integration within the target product, synthesis and optimizations, and finally power estimation.

2.1 Project specifications and design

The central focus of this project is the management of power states within the PMD in the block `PMD_TX_PWR_CTL`. The instructions and configuration data required to handle power state transitions efficiently are stored in a Look-Up Table (LUT). Nowadays each entry of power-up/down LUTs is programmed with a 16-bit data bus.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cmd[5:0]						arg[2:0]			FW	skip0	skip1	skip2	skip3	RESERVED	

Table 2.1: Actual organization

The command (cmd) field of 6 bits allows to encode up to 64 different HW commands: each command is mapped to a specific signal (or a group of signals, in some scenarios) that is conditioned by the `PMD_TX_PWR_CTL` FSMs.
The idea is to implement a programmable hardware command in order to remove the dependency on internal FSMs and achieve a more flexible and customizable behavior.

The `PMD_HW_CMD_FSM` is a new module with a new programmable FSM, that can handle both the existing HW commands (with the only exception of DELAY TOKEN which is already managed by a dedicated FSM) and new ones.

The order of the bits in the Table 2.1 are rearranged, particularly those of the command and argument (arg) fields.
Therefore the new programmable command is configured using a new set of registers that are selected using some of the 9 bits of the arg field. The arg field has different meanings depending on the command being executed (this will be analyzed later).

The cmd field is 3 bits: eight different commands can be configured.

Code	Command	Description
0	BASE_ASSIGN	This command implements the programmable version of the basic assignment category
1	COND_ASSIGN	This command implements the programmable version of the conditional assignment category
2	HANDSHAKE	This command implements the programmable version of both handshake categories
3	TIME_WAIT	This command implements the programmable version of the time waiting category
4	SIG_PULSE	This command implements a programmable pulse (not currently implemented by main HW FSM)
5	RESERVED	This coding is currently spare
6	DELAY_TOKEN	Not implemented by programmable HW command
7	FW_COMMAND	Not implemented by programmable HW command

Table 2.2: Possible commands

A possible organization of each entry of LUTs is:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
arg[8:0]									cmd[2:0]			skip0	skip1	skip2	skip3

Table 2.3: Future Organization

The bits[15:6] and the bits[1:0] of the old configuration are rearranged (FW command is merged in the new configuration). The skip bits are just translated.

The following provides an overview of the module interface and a detailed description of each command.

2.1.1 Interface of the module

This section outlines the interface of the module and the signals involved, as illustrated also in the next figure.

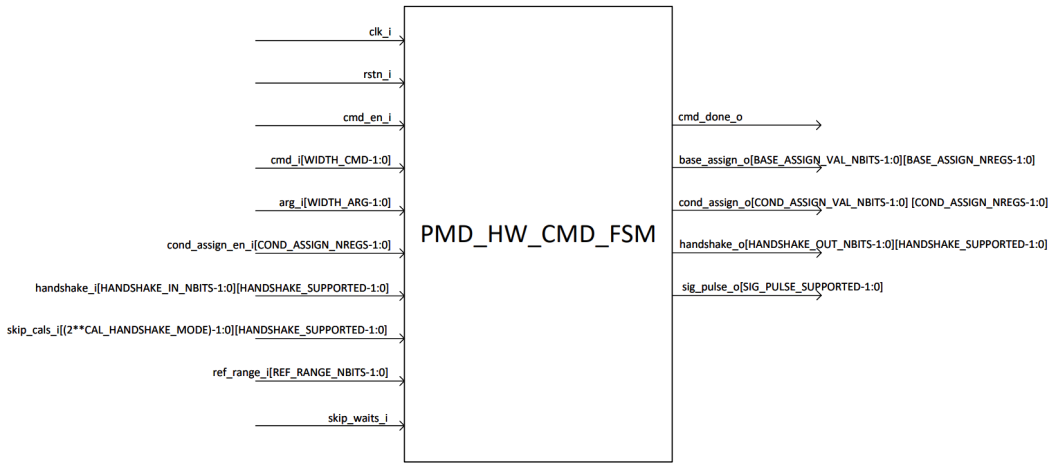


Figure 2.1: Interface of the module

The input and output signals that define the module interface are listed below. The direction, width and description are provided for each signal.

clk_i (input, 1 bit) Clock signal of the module.

rstn_i (input, 1 bit) Asynchronous active-low reset.

cmd_en_i (input, 1 bit) Command enable signal: when set to 1 the command is processed.

cmd_i (input, `WIDTH_CMD`) Command type identifier.

arg_i (input, `WIDTH_ARG`) Argument associated with the selected command.

cond_assign_en_i
(input, `COND_ASSIGN_NREGS`) Enables conditional assignment to selected registers: 0 = no assignment; 1 = assignment allowed.

handshake_i
(input, `HANDSHAKE_IN_NBITS` × `HANDSHAKE_SUPPORTED`) Input handshake bus. The handshake is completed when all bits are equal to 1.

skip_cals_i
(input, $2^{\texttt{CAL\_HANDSHAKE\_MODE}} \times$ `HANDSHAKE_SUPPORTED`) Indicates whether the calibration handshake should be performed (0) or skipped (1).

ref_range_i
(input, `REF_RANGE_NBITS`) Reference range of the clock.

skip_waits_i
: (input, 1 bit) Indicates whether the TIME WAIT command should be executed (0) or skipped (1).

cmd_done_o
: (output, 1 bit) Asserted when the command execution is completed.

base_assign_o
: (output, `BASE_ASSIGN_VAL_NBITS` × `BASE_ASSIGN_NREGS`) Output registers of the base assignment command.

cond_assign_o
: (output, `COND_ASSIGN_VAL_NBITS` × `COND_ASSIGN_NREGS`) Output registers of the conditional assignment command.

handshake_o
: (output, `HANDSHAKE_OUT_NBITS` × `HANDSHAKE_SUPPORTED`) Output registers related to the handshake command.

sig_pulse_o
: (output, `SIG_PULSE_SUPPORTED`) Output register for the signal pulse command.

It can be seen that the widths of some input and output signals are parametric. This offers an additional degree of flexibility, as it allows these widths to be defined when the `PMD_HW_CMD_FSM` is instantiated into other modules, adapting them to specific requirements and optimizing the area. The following table shows the allowed values for these parameters.

Name	Min	Max	Default
WIDTH_CMD	1	3	3
WIDTH_ARG	1	9	9
BASE_ASSIGN_VAL_NBITS	1	3	3
BASE_ASSIGN_NREGS	2	64	64
COND_ASSIGN_VAL_NBITS	1	2	2
COND_ASSIGN_NREGS	2	64	64
HANDSHAKE_IN_NBITS	1	3	3
CAL_HANDSHAKE_MODE	1	2	2
HANDSHAKE_OUT_NBITS	2	4	4
HANDSHAKE_SUPPORTED	2	32	32
REF_RANGE_NBITS	1	4	4
SIG_PULSE_SUPPORTED	2	8	8

Table 2.4: Complete set of parameters.

There are two additional parameters that must be defined when the module is instantiated:

- **RESET_BASE_ASSIGN**: this is a two-dimensional parameter whose size is `BASE_ASSIGN_VAL_NBITS` × `BASE_ASSIGN_NREGS`. It is used to reset the base-assign command registers to the desired initial values. (Default value = `'0`)
- **SP_PHASE_NBITS**: number of bits used for the pulse phases. It can be either 1 or 2. (Max value = 2, Default value = 2)

The analysis of each command provides more details about the choice of all of these parameters.

2.1.2 Base assign

This command implements the assignment of the value to the selected register. The arg field is divided in two sections :

8	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Select						Value		

Table 2.5: arg field for `BASE ASSIGN` command

The value field is the value to assign to the selected register.
The select field is the selected register to update.

It is possible to assign values of 1,2 or 3 bits and this information can be provided to the module through `BASE_ASSIGN_VAL_NBITS`. The value will always be placed in the LSBs of the argument.
`BASE_ASSIGN_NREGS` is set to indicate how many registers need to be instantiated. The information of which register should be written is encoded in bits 3 to 8 of the argument.
The following timing diagram is to be expected :

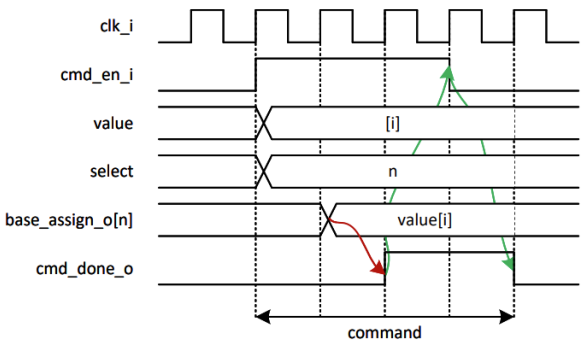


Figure 2.2: Timing Diagram of the `BASE ASSIGN` Command

Red arrow: the cmd_done_o is asserted after the value has been assigned.
Green arrow: highlights the handshake between cmd_done_o and cmd_en_i.

2.1.3 Conditional assign

This command implements the assignment of the value to the selected register if the cond_assign_en_i corresponding to that register is equal to 1, or if the force bit is set to 1. If neither of these conditions is met, the assigned value is 0.
The arg field is divided in three sections :

8	7	6	5	4	3	2	1	0
Select						Force	Value	

Table 2.6: arg field for `COND ASSIGN` command

The value field is the value to assign to the selected register.
The force field is a single bit which specifies that the value has to be assigned, neglecting the input signal cond_assign_en_i.
The select field is the selected register to update.

It is possible to assign values of 1 or 2 bits and this information can be provided to the module through `COND_ASSIGN_VAL_NBITS`. The value will always be placed in the LSBs of the argument.
`COND_ASSIGN_NREGS` is set to indicate how many registers need to be instantiated. The information of which register should be written is encoded in bits 3 to 8 of the argument.
The following timing diagram is to be expected :

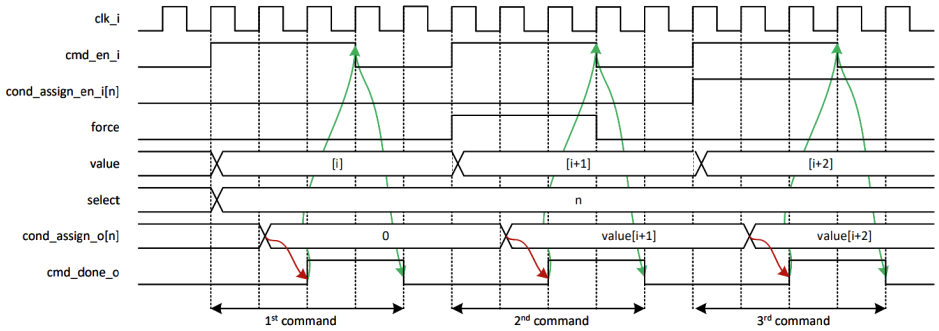


Figure 2.3: Timing Diagram of the `COND ASSIGN` Command

Red arrow: the cmd_done_o is asserted after the value has been assigned.
Green arrow: highlights the handshake between cmd_done_o and cmd_en_i.

2.1.4 Handshake

The module implements two different types of handshake mechanism. The MSB of the arg field, named *'cal'*, determines if the command is a *calibration hanshake* (=1) or a *generic handshake* (=0).

8	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
cal								

Table 2.7: arg field for `HANDSHAKE` command

The meaning of the 8 LSBs is different depending on the type of handshake.

The values of the parameters related to this command must be defined in a way that is consistent with both handshake types. The parameter `HANDSHAKE_IN_NBITS` depends on the *generic handshake* command. It can be 1,2 or 3 and must correspond to the bit-width of the value to be written to the output register. This parameter represents the size of the external feedback signal that indicates the handshake can be concluded.
The parameters `HANDSHAKE_OUT_NBITS` and `HANDSHAKE_SUPPORTED` define the output size related to the handshake command. The parameter `HANDSHAKE_OUT_NBITS` must consider the size of the value to be written for the generic handshake, but it mainly depends on the outputs expected from the calibration handshake (see subsection below 2.1.4).
`HANDSHAKE_SUPPORTED` is set to indicate how many registers need to be instantiated for both handshake types. The information of which register should be written is

encoded in bits 4 to 7 of the argument for the generic handshake, instead from bits 3 to 7 for the calibration handshake.

Generic Handshake

This command performs the value assignment to the selected register, and waits for every bit of the corresponding handshake_i to go high before asserting cmd_done_o.
If the bypass bit is set 1, a basic assignment is executed.
The arg field is divided in three sections :

7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Select				Bypass	Value		

Table 2.8: arg field for `GENERIC HANDSHAKE` command

The value field is the value to assign to the selected register.
The bypass field is a single bit which specifies that the value has to be assigned without performing the handshake.
The select field is the selected register to update.

The following timing diagram is to be expected :

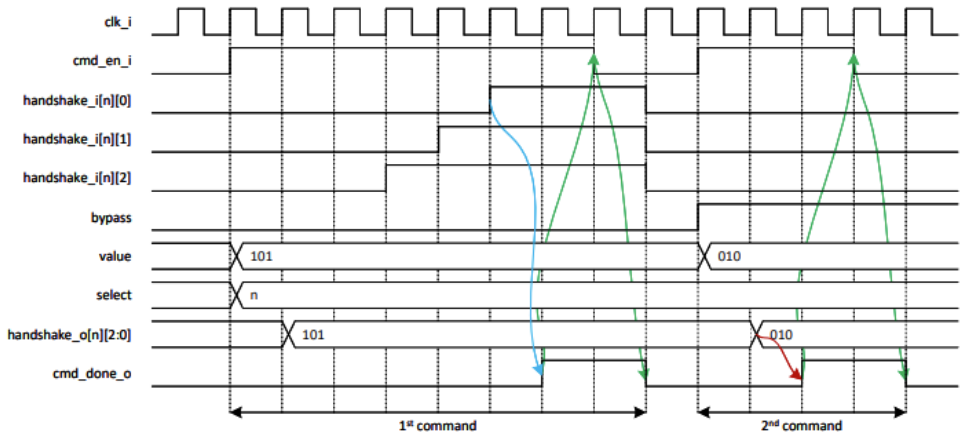


Figure 2.4: Timing Diagram of the `GENERIC HANDSHAKE` Command

Blue arrow: the cmd_done_o is asserted after the handshake is completed.
Red arrow: the cmd_done_o is asserted after the value has been assigned.
Green arrow : highlights the handshake between cmd_done_o and cmd_en_i.

Calibration handshake

This command performs the value assignment to the selected register and sets its LSB to 1. Then it monitors the corresponding handshake_i[0] : when the signal goes high the LSB is deasserted, when the signal returns low the cmd_done_o is asserted .
If the correspondig skip_cals_i[cal_mode] is equal to 1, the value of cal_mode and cal_sel are updated without enabling the calibration.
The arg field is divided in three sections :

7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Select					cal_sel	cal_mode	

Table 2.9: arg field for `CALIBRATION HANDSHAKE` command

The cal_mode field and the cal_sel field are the values to assign to the selected register (both of these are not necessarily required).
The select field is the selected register to update.
Table 2.4 shows that, in addition to the width that defines the output for this command, one of the parameters is the width of *cal_mode*. So four different scenario exist for the width of `HANDSHAKE_OUT_NBITS`, depending on both the type of information to be provided as output and the size of parameter `CAL_HANDSHAKE_MODE`. The LSB of the output is always occupied by the bit that interacts with the external modules. It can represents an enable or a request.
The different cases are :

1. `HANDSHAKE_OUT_NBITS`=2 with `CAL_HANDSHAKE_MODE`=1

cal_mode	fixed LSB

2. `HANDSHAKE_OUT_NBITS`=3 with `CAL_HANDSHAKE_MODE`=1

cal_sel	cal_mode	fixed LSB

3. `HANDSHAKE_OUT_NBITS`=3 with `CAL_HANDSHAKE_MODE`=2

cal_mode[1]	cal_mode[0]	fixed LSB

4. `HANDSHAKE_OUT_NBITS`=4 with `CAL_HANDSHAKE_MODE`=2

cal_sel	cal_mode[1]	cal_mode[0]	fixed LSB

The following timing diagram is to be expected :

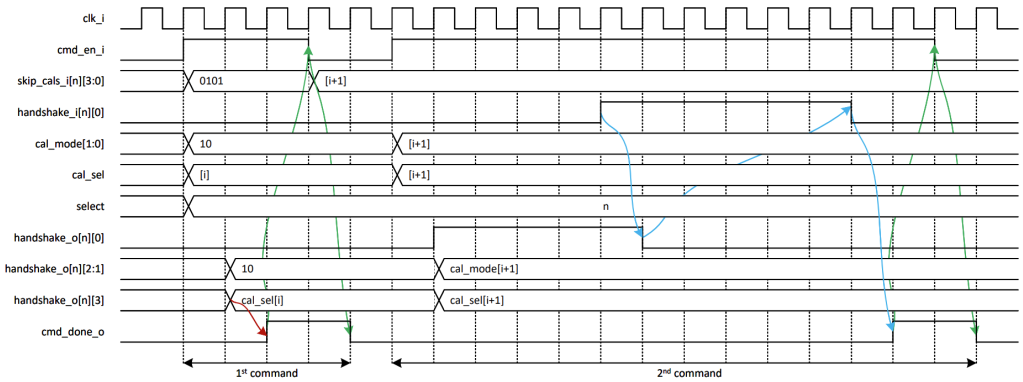


Figure 2.5: Timing Diagram of the `CALIBRATION HANDSHAKE` Command

Blue arrow: depending on the value of handshake_i[select][0] the handshake_o[select][0] is deasserted and the cmd_done_o is asserted.
Red arrow: the cmd_done_o is asserted after the value has been assigned.
Green arrow: highlights the handshake between cmd_done_o and cmd_en_i.

2.1.5 Time wait

This command simulates a waiting time.
The arg field is divided in two sections :

8	7	6	5	4	3	2	1	0
Count							Timescale	

Table 2.10: arg field for `TIME WAIT` command

The timescale field selects which timescale to use for this command:

- 00: 40ns
- 01: 1us
- 10: 10us
- 11: 100us

The count field is the amount of timescale units to count before issuing the signal cmd_done_o.
The actual clock periods between cmd_en_i assertion and cmd_done_o rising edge will be timescale*(count+1). In this way also the 0 is meaningful.
The time to be waited is independent of the reference clock frequency that is

provided to the module, so the input ref_range_i is used to adjust the internal counter to keep the waiting time consistent. The ref_range_i can be:

- 0000: 25MHz
- 0001: 50MHz
- 0010: 75MHz
- 0011: 100MHz
- ...
- 1111: 400MHz

The following timing diagram is to be expected :

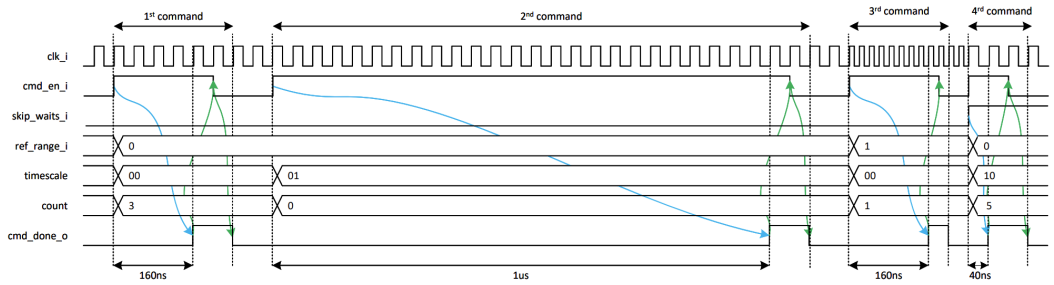


Figure 2.6: Timing Diagram of the `TIME WAIT` Command

Blue arrow: the elapsed time matches the duration required by the command, the cmd_done_o can be asserted.
Green arrow: highlights the handshake between cmd_done_o and cmd_en_i.

2.1.6 Signal pulse

This is a new command designed to generate a pulse with a programmable duration. The arg field is divided in four sections :

8	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Select			t3		t2		t1	

Table 2.11: arg field for `SIG PULSE` command

The t1, t2 and t3 fields are the duration of the three pulse phases, as follow :

- 00: 40ns
- 01: 80ns
- 10: 120ns
- 11: 160ns

The time to be waited is independent of the reference clock frequency that is provided to the module, so the input ref_range_i is used to adjust the internal counter to keep the waiting time consistent (same of TIME WAIT command).
The select field is the selected register to update.

The parameter `SIG_PULSE_SUPPORTED` defines how many registers need to be instantiated for the sig pulse command. Its maximum value defines the fixed size of the select field. The information of which register should be written is encoded in bits 6 to 8 of the argument.
The parameter `SP_PHASE_NBITS` defines the size of the three phases of the pulse and its maximum value defines the fixed size of the t1, t2 and t3 fields.
The following timing diagram is to be expected :

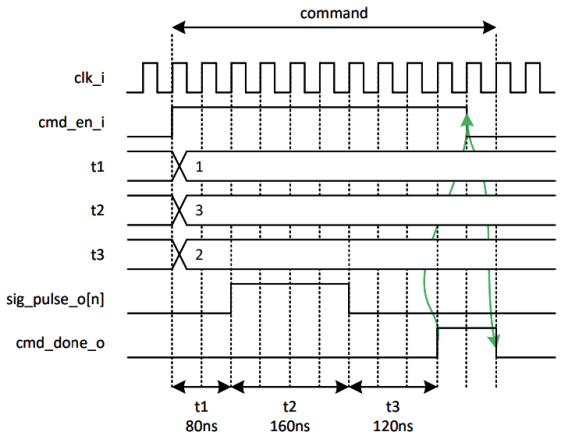


Figure 2.7: Timing Diagram of the `SIG PULSE` Command

Green arrow: highlights the handshake between cmd_done_o and cmd_en_i.

2.2 Design Verification

According to the digital design flow (1.3), after the project specifications are defined, the expected behaviour is outlined and the RTL description of the module is implemented, static checks and simulation are carried out.

2.2.1 Design Checks: Lint, CDC and RDC

As system-on-a-chip (SoC) systems become more complex, it is important to verify that RTL, clock domain crossing (CDC) and reset domain crossing (RDC) are constructed correctly. This should be done early in the RTL phase of development.

Synopsys VC SpyGlass is an advanced algorithm- and analysis-based tool that provides designers with detailed information and insights about their design much earlier in the RTL phase.[8]

The first level of verification consists of linting, which allows to identify stylistic problems, syntactic errors and potential coding bugs, improving the overall quality of the code. RTL linting detects several types of problems: syntactically, it reports errors or warnings when code doesn't adhere to the reference language; semantically, it identifies incomplete cases, missing default conditions or else clauses, and uninitialized ports. At the synthesis level, RTL linting identifies non-synthesizable constructs such as delays or unrecorded input/output blocks; for simulation, it verifies incomplete sensitivity lists and incorrect use of blocking/non-blocking assignments. Structural analysis enables the identification of issues that could affect functionality or performance, such as combinational loops, clock, select, and enable management, multiple drivers, unmanned signals, unconnected nets and floating pins.
VC SpyGlass Lint, which is part of VC Spyglasss, detects coding issues and enhances code quality, leading to faster and more efficient verification cycles. [9]

In order to start the linting analysis, the tool requires a set of inputs: the RTL design to be analyzed and a TCL file containing the tool configuration, including the inclusion of standard cell libraries and the list of rules or goals to be verified. It also requires the constraints applied to the design and finally the waivers list, which collects the exceptions to be excluded during the checks.
The following instructions outline some of the `PMD_HW_CMD_FSM`'s constraints.

- *create_reset "rstn_i" -async -type reset -value low*: this command defines an active low asynchronous reset signal.
- *create_clock -name clk_i -period 10 clk_i*: this command defines a clock signal with a period of 10ns.
- *set_input_delay 0.0 -clock clk_i cmd_en_i*: specifies that the input signal *cmd_en_i* is considered synchronous with respect to the clock *clk_i*.

As previously mentioned, some waivers are applied to exclude reports that are not relevant to the functional correctness of the design. In particular, the following waiver is used:

```
//spyglass disable ImproperRangeIndex-ML
```

which is applied in cases where the width of the selection signal is found to have a maximum value higher than the one actually required, generating a warning that

is not significant for the implementation.
The linting process contributed to the production of well-structured and error-free RTL code, preparing it for the subsequent stages of the design flow.

The second verification step is dedicated to identifying problems related to Clock Domain Crossing (CDC). In complex designs, the presence of multiple clock domains with different frequencies and phases is common, which can generate various critical issues.
The main problems are: metastability, compliance with synchronous reset conventions, correct clock assignment to all flip-flops, verification that no flip-flop is controlled by more than one clock, and verification that no more clocks have been defined on the same clock path.
A typical issue arises when data is transferred between flip-flops controlled by asynchronous clocks, that is, whenever a signal crosses different clock domains.
VC SpyGlass CDC correlates control and data signals resulting in a good understanding of the design intent for the lowest possible noise. It also integrates structural and functional CDC analysis.[10]

The clock is unique in the case of the developed module. After doing this analysis, the tool did not report any errors.

The third step of the verification concerns Reset Domain Crossing (RDC) issues, which occur when asynchronous resets are asserted within the same clock domain. Similar to CDC problems, these scenarios can cause metastability and generate unpredictable behavior in the circuit. In particular, traversing asynchronous reset signals can lead to conditions where some flip-flops are not reset correctly or receive undefined intermediate values, compromising the consistency of the design state. Testing RDC issues, therefore, focuses on the correct use of asynchronous resets, ensuring that they do not generate conflicts within the same clock domain and that all affected flip-flops are initialized consistently and reliably.

This analysis is also performed using the VCSpyglass tool and no errors are generated.

2.2.2 Functional Verification: Testbench and Verdi

A testbench is developed for the module under study and simulations are performed using Synopsys' Verdi platform.
Verdi is a comprehensive solution for debugging and managing the verification of digital designs. It enables the organisation, execution and monitoring of simulations, and provides advanced tools for understanding and analysing design behaviour.[11]

The simulation environment is designed to cover a wide range of scenarios and ensure the correct functionality of the module. To this purpose, randomized inputs are applied and variable time intervals are introduced between consecutive commands. The values of the parameters that define the widths of the module's input and output interfaces are also assigned randomly.

The result of the simulations are shown in Section 3.1.1.

2.3 Integration in product

Once the `PMD_HW_CMD_FSM` module is described and implemented in SystemVerilog and subsequently tested, the study proceeds with its integration into the company product. As previously indicated, the part of the product involved concerns the management of power states in the PMD for the transmitter, namely `PMD_TX_PWR_CTL`. The module is instantiated through a macro so that, when the macro is defined, the `PMD_TX_PWR_CTL` uses the newly integrated FSM, while when it is not defined, the original behavior can be restored. In this way, it is possible to analyze both operating modes of the `PMD_TX_PWR_CTL` and observe the effects of the new implementation.

To ensure a correct integration, an additional module called `bridge_tx` is created. Its function is simply to reorganize the input signals to the `PMD_HW_CMD_FSM`, moving from the structure shown in Table 2.1 to that of Table 2.3. This approach is adopted because the work is still in the project phase, and before rewriting the firmware, it is necessary to verify the efficiency of the new implementation.

As in the previous case, the design flow steps described in Section 1.3 are followed. After drafting the RTL description in SystemVerilog, static checks are performed using the VC Spyglass environment in line with the company's workflow. These checks did not reveal any errors.
A testbench is then created to simulate the behaviour of the `PMD_TX_PWR_CTL` module with the macro both defined and undefined. Following the simulations performed on Verdi, it is observed that the module exhibits identical behavior in both implementations.

The result of the simulations are shown in Section 3.1.2.

2.4 RTL Synthesis and Optimization in Design Compiler NXT

The next step in the digital design flow 1.3 is the synthesis, which is performed using Synopsys Design Compiler NXT. This is an RTL synthesis tool that translates hardware descriptions written in languages such as Verilog or VHDL into an optimized gate-level netlist. During this phase, the RTL description of the module is analyzed and transformed, applying optimizations to meet timing, area, and power constraints, while ensuring that the functional behavior of the design is preserved. The resulting netlist is then ready to be used in the subsequent stages of integrated circuit design.[12]

The synthesis of a digital module involves a series of well-defined steps: the RTL description is analyzed, elaborated, linked to the technology libraries, and finally optimized to generate a gate-level netlist.
The main commands used in this flow are listed and described below.

- **source -echo -verbose $rtl_tcl_path** : read all RTL design files composing the entire design to be synthesized.
- **analyze -format sverilog -lib work `$ALL_FILES` -define `$SYN_DEFINE`**: analyzes the specified SystemVerilog files and stores their intermediate representation into the designated library, allowing the tool to create linkable design units.
  The variable `SYN_DEFINE` contains a list of macros and parameters used to select whether the synthesis should be performed on the original module or the version integrating the newly developed functionality.
- **elaborate `$DESIGN` -lib work** : builds the design hierarchy from its intermediate representation, applying the parameters and configurations previously defined.
- **link**: resolves all design references by connecting the current design to the library elements it depends on. A design is considered complete only when every referenced cell or module is connected to the appropriate component within the technology libraries. The purpose of this command is therefore to locate and bind all referenced elements, ensuring functional completeness and consistency of the design hierarchy.

Before compilation, all required constraints must be applied (these constraints are detailed in Section 2.4.1).
Next, the technology node associated with the process must be specified:

- **set_technology -node `$TECH_NODE`**

After defining the technology, several **"set_app_var"** commands are issued to configure internal application variables that control the synthesis behavior. Additional optimization directives are also applied to prepare the environment for design compilation.

At this stage of the flow, the design is ready to be compiled, and different compilation strategies can be adopted depending on the desired level of optimization.
The simplest option is the **compile** command, which optimizes the logic and gates in the current design. Its behavior is driven by the user-specified constraints that express objectives such as minimizing area or meeting timing requirements. The optimization process of **compile** evaluates timing–area trade-offs to produce the smallest possible circuit that still satisfies the imposed timing constraints. These constraints are generally classified as design-rule constraints, which are mandatory technology-dependent rules, or optimization constraints, which express desirable but non-critical design objectives.

In this work, a higher-effort compilation strategy has been used:

- **compile_ultra -no_autoungroup -no_seq_output_inversion -gate_clock -retime**

The **compile_ultra** command enables an advanced optimization process designed to enhance performance and quality of result (QoR). This process is particularly well-suited to designs with tight timing constraints.
The options of this command operate as follows :

- **-no_autoungroup**: disables automatic hierarchy ungrouping, preserving the module structure unless explicitly specified.
- **-no_seq_output_inversion**: prevents the tool from inverting sequential outputs, ensuring consistency between RTL and gate-level sequential behavior.
- **-gate_clock**: enables the clock-gating technique for optimization, allowing the tool to automatically insert or remove clock-gating cells.
- **-retime**: activates adaptive retiming to reduce critical path delays through register repositioning.

A second compilation step is then executed:

- **compile_ultra -incremental**

Running the command in incremental mode enables the tool to refine only those parts of the design that could still benefit from optimization, without having to run a full mapping again.

Before writing the final netlist, the following command is performed:

- **change_names -rules `CASEINS` -hierarchy -verbose**

This command adjusts the names of ports, cells (including physical-only cells) and nets, so that they comply with the specified naming rules. The use of **change_names** ensures that the object names within the tool are aligned with those that will appear in the generated design files, resulting in consistent naming across reports and netlists. It also enables the implementation of naming conventions required by the target environment.

The gate-level netlist is then generated through:

- **write -format verilog -hier -output $netlist_path/$DESIGN.v**

The `-hier` flag ensures that all hierarchical submodules are also written to the output file.

After the netlist generation, additional structural checks and synthesis reports can be produced to validate and evaluate the quality of the implementation.
The corresponding commands and results are directly analyzed in Section 3.2.

Finally, equivalence checking is performed using Formality (see Section 2.4.2).

All the steps in this section are done twice. This is to synthesize the `PMD_TX_PWR_CTL` module, both in its original and new implementation.
Once the two netlists are generated, a simulation is performed to verify the behavior's correspondence with the RTL description.
Since the results confirmed functional equivalence, waveforms are not reported, as they are considered redundant for analysis purposes.

2.4.1 Constraints

In the synthesis flow, it is necessary to define the set of constraints applied to the design. In the following, the constraints related to the clock signals and to the input/output interfaces are reported.

Clock

The first step in the constraint definition is setting the units of measurement used throughout the synthesis process:

- set_units -time ns
- set_units -capacitance pF

The design operates with two clock signals: *pmd_refclk_i*, with a frequency of 400 MHz, and *pmd_pclk_i*, with a frequency of 500 MHz.

Subsequently, several variables are defined to parameterize the characteristics of these clocks, including period, duty-cycle error and waveform generation.

- set `JITTER_FACTOR 0.90`
- set `PMD_REF_PERIOD [expr $JITTER_FACTOR*2.50 ]` ;
- set `PMD_P_PERIOD [expr $JITTER_FACTOR*2.00]` ;
- set `PMD_REF_DC_ERROR [expr $PMD_REF_PERIOD*0.00]`
- set `PMD_P_DC_ERROR [expr $PMD_P_PERIOD*0.00]`
- set `PMD_REF_HALF_PERIOD [expr ($PMD_REF_PERIOD/2) +($DUTYCYCLE_ERROR_FACTOR*$PMD_REF_DC_ERROR)]`
- set `PMD_P_HALF_PERIOD [expr ($PMD_P_PERIOD/2) +($DUTYCYCLE_ERROR_FACTOR*$PMD_P_DC_ERROR)]`
- set `wave_PMD_REF_HALF_PERIOD [list 0 $PMD_REF_HALF_PERIOD]`
- set `wave_PMD_P_HALF_PERIOD [list 0 $PMD_P_HALF_PERIOD]`

Once these parameters are defined, the actual clock objects are created.
The **-waveform** option specifies the rise and fall edges over one full clock period. If omitted, a default waveform with a rising edge at 0 and a falling edge at half the period would be assumed. Here, the waveform is explicitly defined:

- create_clock -name `PMD_REF_CLK` -period `$PMD_REF_PERIOD` -waveform `$wave_PMD_REF_HALF_PERIOD` [get_port pmd_refclk_i]
- create_clock -name `PMD_P_CLK` -period `$PMD_P_PERIOD` -waveform `$wave_PMD_P_HALF_PERIOD` [get_port pmd_pclk_i]

Finally, it is necessary to specify that the two clocks are asynchronous. Declaring asynchronous clock groups prevents the timing engine from analyzing paths between them:

- set_clock_groups -asynchronous -group [list `PMD_REF_CLK`]
  -group [list `PMD_P_CLK`]

The **-asynchronous** option indicates that the clocks have no phase relationship, therefore no timing paths should be evaluated between the two domains.

Input-output

To properly constrain inputs and outputs, a set of timing margins is first defined:

- set `PMD_REF_CLK_INPUT_MARGIN_MIN [expr 0]`
  set `PMD_REF_CLK_INPUT_MARGIN_MAX [expr 0.10*$PMD_REF_PERIOD]`
- set `PMD_REF_CLK_OUTPUT_MARGIN_MIN [expr 0]`
  set `PMD_REF_CLK_OUTPUT_MARGIN_MAX [expr 0.10*$PMD_REF_PERIOD]`
- set `PMD_P_CLK_OUTPUT_MARGIN_MIN [expr 0]`
- set `PMD_P_CLK_OUTPUT_MARGIN_MAX [expr 0.10*$PMD_P_PERIOD]`

Two lists containing all synchronous inputs and outputs are created:

- set synch_input_list[...] : this command creates the input list.
- set synch_output_list[...] : this command creates the output list.

Once the lists are defined, input delays can be applied relative to the reference clock:

- set_input_delay -clock `PMD_REF_CLK` -min -add_delay [expr `$PMD_REF_CLK_INPUT_MARGIN_MIN`] [get_port $synch_input_list]
- set_input_delay -clock `PMD_REF_CLK` -max -add_delay [expr `$PMD_REF_CLK_INPUT_MARGIN_MAX`] [get_port $synch_input_list]

The same process is applied to the output signals, in this case associated with both clocks:

- set_output_delay -clock `PMD_REF_CLK` -min -add_delay [expr `$PMD_REF_CLK_OUTPUT_MARGIN_MIN`] [get_port $synch_output_list]
- set_output_delay -clock `PMD_REF_CLK` -max -add_delay [expr `$PMD_REF_CLK_OUTPUT_MARGIN_MAX`] [get_port $synch_output_list]

- set_output_delay -clock PMD_P_CLK -min -add_delay [expr $PMD_P_CLK_OUTPUT_MARGIN_MIN] [get_port sreg_fw_cmd_en_o]
- set_output_delay -clock PMD_P_CLK -max -add_delay [expr $PMD_P_CLK_OUTPUT_MARGIN_MAX] [get_port sreg_fw_cmd_en_o]

After constraining synchronous paths, asynchronous input and output lists must be generated by removing the synchronous elements:

- set false_inputs [remove_from_collection [all_inputs] [get_ports $synch_input_list]]
- set false_outputs [remove_from_collection [all_outputs] [get_ports $synch_output_list]]
- set false_outputs [remove_from_collection $false_outputs [get_ports sreg_fw_cmd_en_o]]

The instructions below are used to remove timing constraints and make these signals asynchronous :

- set_false_path -from [get_ports $false_inputs] -to [all_clocks]
- set_false_path -to $false_outputs -from [all_clocks]

Finally, it is necessary to specify the load attributes and to associate an external driving cell with the relevant ports and nets of the design. This is done to model external circuitry :

- set_load 0.005 [get_ports *]
- set_driving_cell -lib_cell $DRIVING_CELL [all_inputs]

2.4.2 Formality

After performing synthesis, a verification step is carried out to ensure that the generated gate-level netlist matches the intended behavior described in the RTL. This check, performed using the Synopsys' Formality tool, confirms that the synthesis process or any manual modifications have not introduced functional errors.
Formality uses formal methods to compare two designs and verify their functional equivalence, supporting RTL-to-RTL, RTL-to-gate, and gate-to-gate comparisons. This verification focuses only on functionality and does not consider timing, making it a static analysis process.

Both the netlists passed this test.

2.5 Power estimation: PrimePower

During the synthesis phase, the tool used also generated reports on power consumption. However, these reports are not sufficiently accurate. For this reason, the PrimePower tool is employed for detailed analysis.

PrimePower provides precise gate-level power analysis reports, enabling SoC designers to perform timely optimizations and achieve power targets during implementation and signoff. The supported analysis modes include average power, peak power, glitch power, clock network power, dynamic and leakage power, and multi-voltage power, using activity derived from RTL vectors, gate-level simulation and emulation vectors, or vectorless analysis.[13]

In order to perform this power analysis, the FSDB files generated from both the netlist simulation are required, as they contain the switching activity of the design.
Furthermore, it is necessary to define a specific time window in order to focus the analysis on the portion of interest within the simulation.

For both designs under examination, the supply voltage is set to 0.675 V, ensuring consistency in the comparison of power consumption results.

The results are reported in Section 3.3.

Chapter 3

Results

3.1 Simulation with Verdi

These are the results of the simulation performed as described in Section 2.2.2. Looking at them, it is possible to notice that the implementation is consistent with the specifications described in Section 2.1.

3.1.1 PMD_HW_CMD_FSM module

The figures below show the simulation of the `PMD_HW_CMD_FSM` module, highlighting in detail the execution of each command.

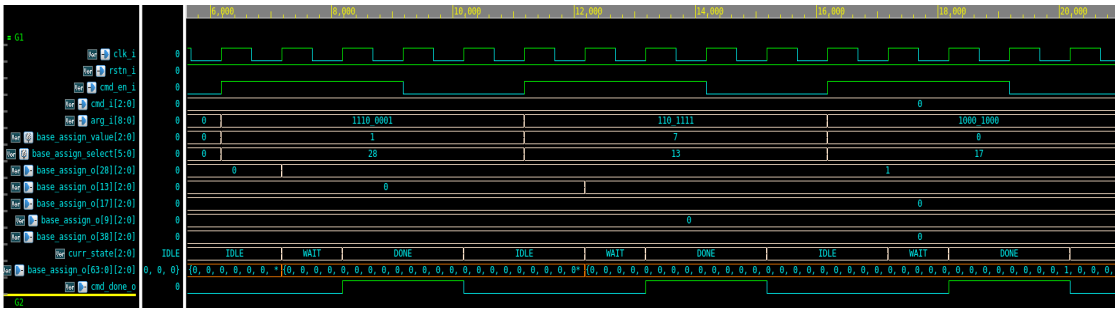


Figure 3.1: Waveform of the `BASE ASSIGN` command

In the figure above, the expected behavior of the `BASE ASSIGN` command is shown. The following figure instead illustrates three scenarios for the `COND ASSIGN` command: first, the case where the corresponding input signal *cond_assign_en_i* is set to 1, then the case where it is 0 and finally the situation in which the assignment is forced.

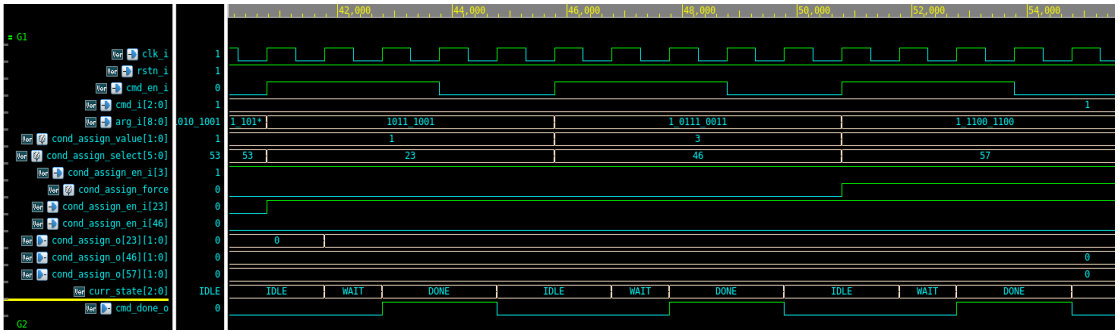


Figure 3.2: Waveform of the `CONDITIONAL ASSIGN` command

The two cases of the `HANDSHAKE` command are also shown in the two figures below, each exhibiting behavior consistent with the expected design operation.

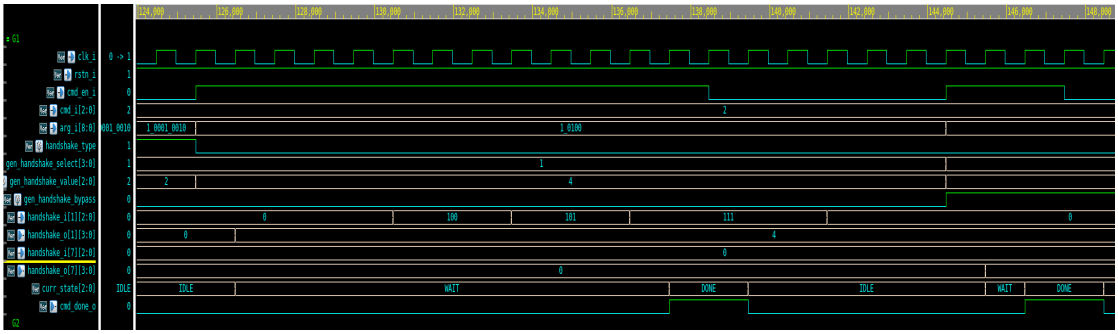


Figure 3.3: Waveform of the `GENERIC HANDSHAKE` command

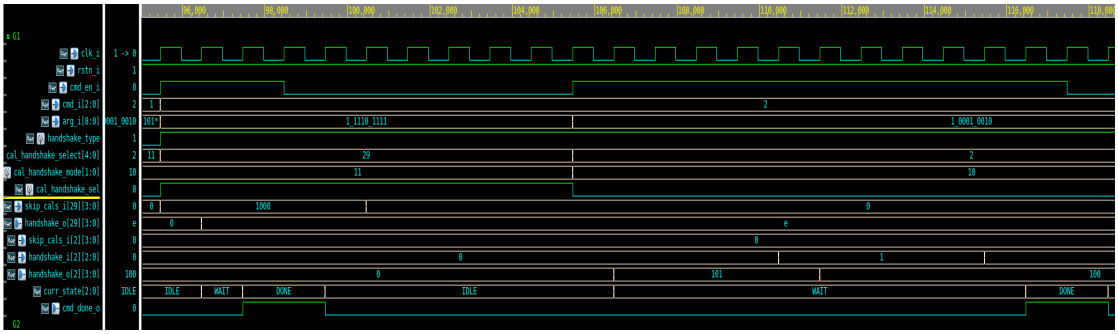
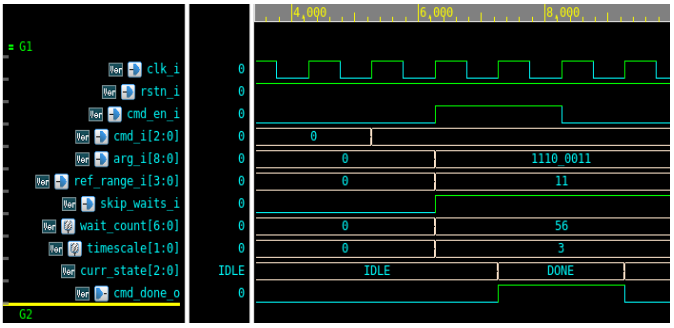
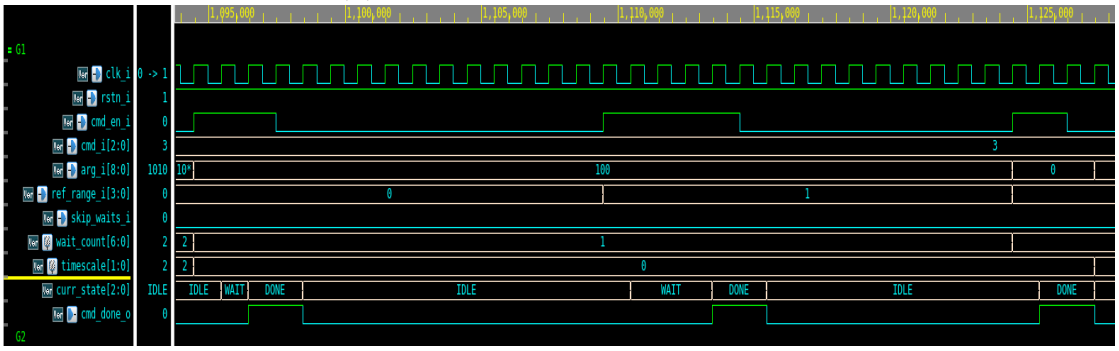


Figure 3.4: Waveform of the `CALIBRATION HANDSHAKE` command

Figure 3.5 illustrates the behavior of the `TIME WAIT` command in two situations: when the waiting phase is skipped, and when the clock has different reference range.



(a) Example of skip the command



(b) Example with two different reference range

Figure 3.5: Waveforms of the `TIME WAIT` command

An example of the generation of a pulse is shown in the following figure.

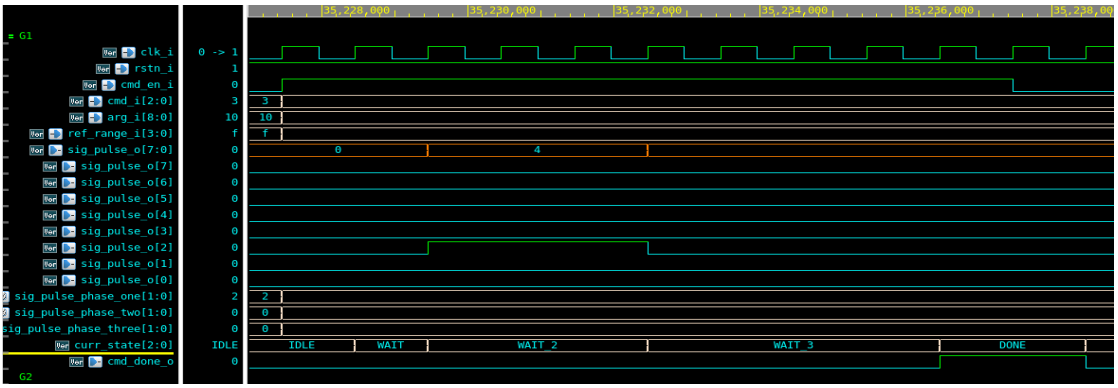


Figure 3.6: Waveform of the `SIGNAL PULSE` command

3.1.2 Integration in product

These results are related to the behavior of the module `PMD_HW_CMD_FSM` integrated within the company product.

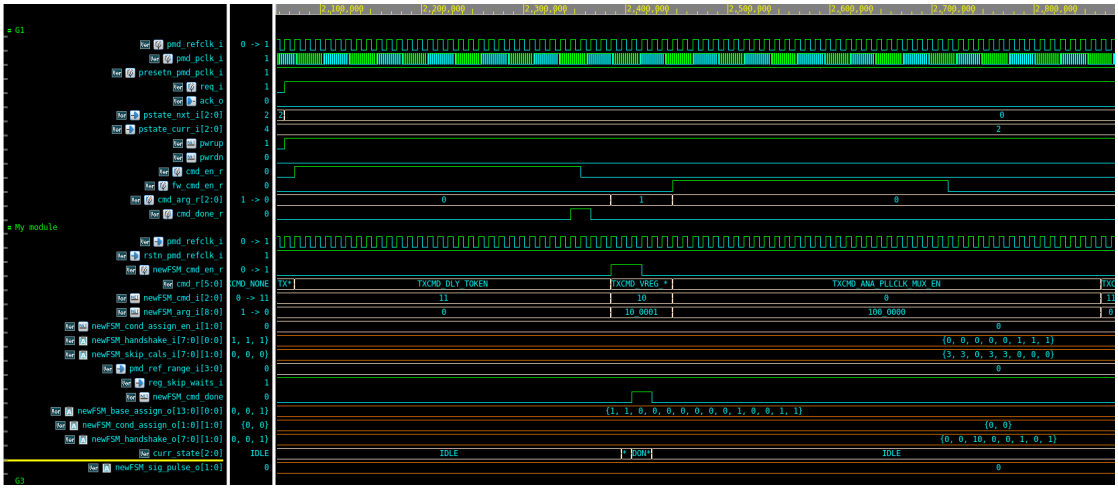


Figure 3.7: A frame of the integration waveforms

The Figure 3.7 shows that when there is a `FIRMWARE` command (like `TXCMD_ANA_PLLCLK_MUX_EN`) or the `DLY TOKEN` command, the enable signal **newFSM_cmd_en_r** of the new module is zero. This is consistent because these two commands are not managed by the `PMD_HW_CMD_FSM`.
There is also the `TX_VREG_WAIT` command, which is treated as a `HANDSHAKE` command. Unlike other handshake commands, it does not produce any output; it simply waits for a feedback signal before proceeding.

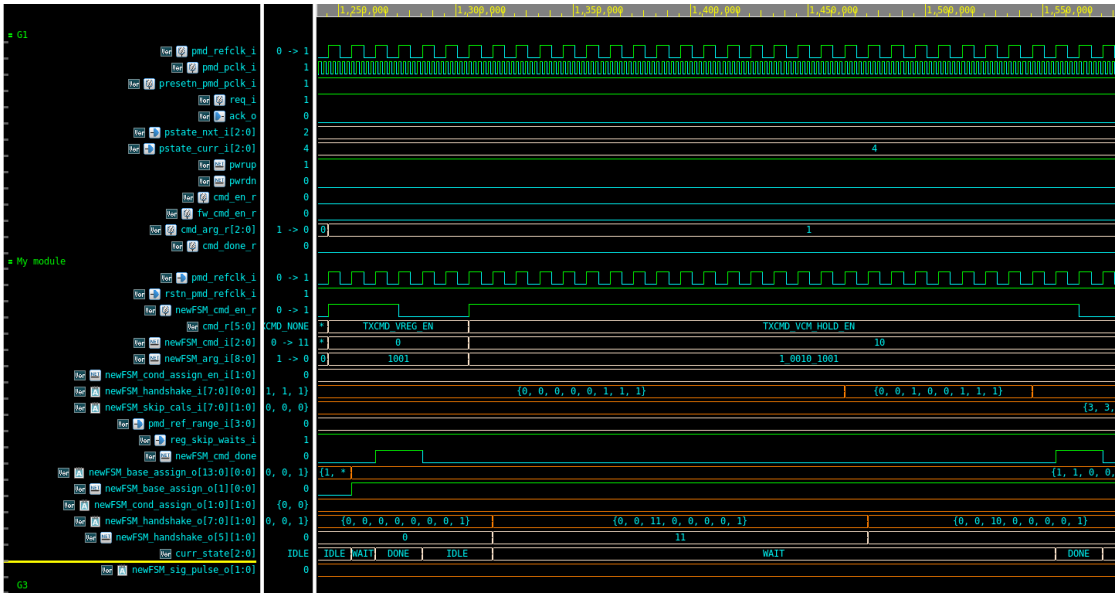


Figure 3.8: A frame of the integration waveforms

In the figure above, the `TXCMD_VREG_EN` corresponds to a `BASE ASSIGN` command. Whereas the `TXCMD_VCM_HOLD_EN` is a `HANDSHAKE` command. These are just some examples.

3.2 Synthesis report

The synthesis is performed with the constraints discussed in Section 2.4.1.
It must be recalled that the synthesis was performed twice on the company product module: the first on the original version and the second on the version that includes the instantiation of the new module.
Some reports are generated by Design Compiler NXT about timing, clock, area, constraints,etc. Let's analyze some of these data.

After having done the synthesis of a design, it is important to verify first the consistency of the work done. The following are important checks to do.

- **check_design**

The **check_design** command is used to verify the internal representation of the current design for consistency, issuing error or warning messages whenever potential issues are detected. It also highlights cases where a design is instantiated multiple times within a system, generating a warning if the same design appears in different instances. Among the issues that can be identified are unloaded input ports or undriven output ports, nets lacking drivers or loads or with multiple drivers, cells or designs without defined inputs or outputs, mismatches between instance and reference pin counts, tristate buses driven by non-tristate elements, and wire loops (timing loops without intervening cells) across hierarchical levels. Other similar structural inconsistencies can also be detected

The report regarding the original module gives these information:

Unconnected ports = 5
Warning: In design 'pmd_tx_pwr_ctl', port 'tx_vcm_lp_la_clk_i' is not connected to any nets.
Warning: In design 'pmd_tx_pwr_ctl', port 'reg_tx_disable_mask_i[4]' is not connected to any nets.
Warning: In design 'pmd_tx_pwr_ctl', port 'reg_tx_disable_mask_i[3]' is not connected to any nets.
Warning: In design 'pmd_tx_pwr_ctl', port 'reg_bg_rdy_mask_i[4]' is not connected to any nets.
Warning: In design 'pmd_tx_pwr_ctl', port 'reg_bg_rdy_mask_i[3]' is not connected to any nets.

This is what is expected, as those wires are not connected to anything.

Instead for what concerns the module with the integration: the **check_design** shows both the above warnings and in addition other unconnected signals. This is because the new module `PMD_HW_CMD_FSM` is able to implement more operations with respect to those present in pmd_tx_pwr_ctl.
An example is the output signal of the `SIG_PULSE` command, which is not among the stored commands in the LUT of the `PMD_TX_PWR_CTL`.
In addition, some input and output signals related to the `HANDSHAKE` command are reported as unconnected ports. This was also foreseen because it is remembered that this command performs both the generic and the calibration handshake. In this case the calibration forces the output port to be declared wider even for generic handshakes.

- ***check_timing***

This command checks the timing attributes placed on the current design and issues warning messages for possible problems.
For both cases under analysis, those inputs and outputs that do not have clock constraints are reported because they are asynchronous, consistently with that reported in the Section 2.4.1.

- ***report_constraint -all_violators -nosplit -max_delay***

This instruction displays whether the constraint was violated or met, by how much it was violated or met and the design object that was the worst violator.
For both cases, the outcome of this analysis is:

This design has no violated constraints.

After performing these standard checks, the synthesis reports of interest are produced and their contents are examined in detail in the following sections.

3.2.1 Report timing and clock

- ***report_clock***

This command displays all clock-related information for a design. Below is the result of the report, which is the same for both the original module and the module with integration.

Clock	Period	Waveform	Sources
PMD_P_CLK	1.8000	0 0.9	pmd_pclk_i
PMD_REF_CLK	2.2500	0 1.125	pmd_refclk_i

Table 3.1: Clock information

- ***report_timing $rpt_timing_common_options -slack_lesser_than 0***

The **report_timing** command provides a report of timing information for the design under analysis. By default, this command reports the single worst setup path in each clock group. Here is specified that only those paths with a slack less than 0 are to be reported.
The outcome for both cases is: "*No paths*". This means that all the critical paths are above the threshold.

- ***report_timing -loops -max_paths 10***

It reports only the timing loops in the design.
The outcome for both cases is: "*No loops*".

- ***report_clock_gating***

This command reports information about clock-gating cells and gated and ungated registers in the design. To generate the report, the report_clock_gating command uses clock-gating attributes added on the clock-gating cells, by the tool. Below are reported the results.

Number of Clock gating elements	6
Number of Gated registers	81 (79.41%)
Number of Ungated registers	21 (20.59%)
Total number of registers	102

Table 3.2: Clock Gating Summary - Original `PMD_TX_PWR_CTL`

Number of Clock gating elements	12
Number of Gated registers	85 (77.27%)
Number of Ungated registers	25 (22.73%)
Total number of registers	110

Table 3.3: Clock Gating Summary - New implementation of `PMD_TX_PWR_CTL`

3.2.2 Report area

The ***report_area*** command displays area information for the design. The unit of measurement is μm^2.

Number of ports	1973
Number of nets	4092
Number of cells	2226
Number of combinational cells	2067
Number of sequential cells	97
Number of macros/black boxes	11
Number of buf/inv	159
Number of references	59
Combinational area	488.954893
Buf/Inv area	17.003520
Noncombinational area	104.768638
Macro/Black Box area	30.222720
Net Interconnect area	undefined (Wire load has zero net area)
Total cell area	623.946251

Table 3.4: Area report - Original `PMD_TX_PWR_CTL`

Number of ports	2102
Number of nets	4248
Number of cells	2279
Number of combinational cells	2098
Number of sequential cells	111
Number of macros/black boxes	11
Number of buf/inv	181
Number of references	56
Combinational area	496.886413
Buf/Inv area	19.284480
Noncombinational area	118.765438
Macro/Black Box area	30.222720
Net Interconnect area	undefined (Wire load has zero net area)
Total cell area	645.874571

Table 3.5: Area report - New implementation of `PMD_TX_PWR_CTL`

According to these data, there is an increase of about 3.5% of the area with respect to the original implementation of `PMD_TX_PWR_CTL`.

It is also known that the area per Kgate, while for the original implementation is 2.407 Kgate, for the module with integration is 2.492 Kgate. To perform this calculation, the value of the area of a NAND with two inputs into the library technology used was taken as a reference.
The increase in area is expected, as the new module implements a larger number of commands and therefore requires additional combinational and sequential logic compared to the previous solution.
However, this is not a negative result, since the increase in area is negligible and is balanced by the significant advantages in terms of flexibility introduced by the new implementation.

3.2.3 Report power from Design Compiler NXT

This instruction generates the power consumption report. Results for both implementations are reported.

- *report_power*

Cell Internal Power	104.2374 uW (77%)
Net Switching Power	30.9000 uW (23%)
Total Dynamic Power	135.1375 uW (100%)
Cell Leakage Power	2.6251 nW

Table 3.6: Power report from DC - Original `PMD_TX_PWR_CTL`

Cell Internal Power	110.2987 uW (78%)
Net Switching Power	30.8221 uW (22%)
Total Dynamic Power	141.1208 uW (100%)
Cell Leakage Power	2.7419 nW

Table 3.7: Power report from DC - New implementation of `PMD_TX_PWR_CTL`

The increase in power consumption for the new implementation is primarily due to the additional logic introduced. A larger number of gates and sequential elements results in greater switching activity, which directly raises dynamic power. If the signals of the new implementation toggle more frequently or if the design now includes more complex datapaths, the overall activity factor also grows, reinforcing this effect. As a result, the modified module naturally shows higher power compared to the original implementation.
Although, as previously discussed, this power report is not fully accurate, it still consistently reflects an increase in consumption following the introduction of the

new module. This trend is meaningful and aligns with expectations, regardless of the limited precision of the estimation.

3.3 Power estimation report

This section shows the results relating to the Section 2.5.
Below the tables are generated by this command:

- ***report_power***

The summary power report displays internal, leakage, switching and total power. The peak power, peak time, glitching power and X transition power are reported below the table. The summary power report also reports power for seven predefined power group:

1. io_pad: Cells defined as part of the pad_cell group in the library.
2. memory: Cells defined as part of the memory group in the library.
3. black_box: Cells with no functional description in the library.
4. clock_network: Cells in the clock_network excluding io_pad cells.
5. register: Latches and flip-flops driven by the clock network excluding io_pads and black_boxes.
6. combinational: Nonsequential cells with a functional description.
7. sequential: Latches and flip-flops clocked by signals other than those in the clock network.

Power group	**Internal Power**	**Switching Power**	**Leakage Power**	**Total Power**	**(%)**
clock_network	1.969e-05	0.0000	2.410e-11	1.969e-05	92.78%
register	3.604e-07	9.693e-08	9.252e-10	4.582e-07	2.16%
combinational	4.605e-07	6.132e-07	1.647e-09	1.075e-06	5.07%
sequential	0.0000	0.0000	0.0000	0.0000	0.00%
memory	0.0000	0.0000	0.0000	0.0000	0.00%
io_pad	0.0000	0.0000	0.0000	0.	0.00%
black_box	0.0000	0.0000	0.0000	0.0000	0.00%

Table 3.8: Power report - Original `PMD_TX_PWR_CTL`

Net Switching Power = 7.101e-07 (3.35%)
Cell Internal Power = 2.051e-05 (96.64%)
Cell Leakage Power = 2.597e-09 (0.0/1%)

Total Power = 2.123e-05 (100.00%)

X Transition Power = 0.0000
CAPP Estimated Glitching Power = 0.0000
Peak Power = 1.033e-04
Peak Time = 5422.25

Power group	**Internal Power**	**Switching Power**	**Leakage Power**	**Total Power**	**(%)**
clock_network	2.189e-05	0.0000	4.817e-11	2.189e-05	93.64%
register	3.226e-07	1.057e-07	9.584e-10	4.292e-07	1.84%
combinational	4.843e-07	5.712e-07	1.704e-09	1.057e-06	4.52%
sequential	0.0000	0.0000	0.0000	0.0000	0.00%
memory	0.0000	0.0000	0.0000	0.0000	0.00%
io_pad	0.0000	0.0000	0.0000	0.	0.00%
black_box	0.0000	0.0000	0.0000	0.0000	0.00%

Table 3.9: Power report - New implementation of `PMD_TX_PWR_CTL`

Net Switching Power = 6.769e-07 (2.90%)
Cell Internal Power = 2.270e-05 (97.09%)
Cell Leakage Power = 2.711e-09 (0.01%)

Total Power = 2.338e-05 (100.00%)

X Transition Power = 0.0000
CAPP Estimated Glitching Power = 0.0000
Peak Power = 1.128e-04
Peak Time = 5404.25

Using the command ***report_power -nosplit -groups $power_groups*** it is possible to know the contribution of the power of each instance in the module. The total power of the instance *bridge_tx* is the 0.47% of the whole module, that is 1.092e-07 W. Since this block is temporary in the implementation of the integration, its contribution is subtracted to have a more realistic result.

From the data, it can be noticed an increase of about 9.7% of the power with

respect to the original implementation of the PMD_TX_PWR_CTL.
The reasons are the same explained in the Section 3.2.3
It is evident how this analysis is more accurate compared to the one reported in the mentioned section.

Adding some options to the previous command, more details on these power contributions for each cell can be analyzed. For example:

- ***report_power -hierarchy -sort_by cell_internal_power -leaf -power_greater_than 0***

The option **leaf** is used to indicate that the power report must traverse the hierarchy and report the nets or cells at lower-levels. **Hierarchy** option generates the hierarchy-based power report, while **sort_by sort_method** specifies the sorting mode for the net or cell order in the power report. The option **power_greater_than threshold** is used to report only the nets or cells with total power value equal to or greater than threshold value.
The results of this command are not reported since there are confidential company data.

- ***report_power -threshold_voltage_group***

This option is used to report the leakage power for each voltage threshold group. In this analysis, three groups are considered: LVT (Low Voltage Threshold), SVT (Standard Voltage Threshold) and UVT (Ultra-Low Voltage Threshold).

Power group	**LVT leakage (%)**	**SVT leakage (%)**	**Total leakage (%)**
memory	0.0000 (0.00%)	0.0000 (0.00%)	0.0000 (0.00%)
io_pad	0.0000 (0.00%)	0.0000 (0.00%)	0.0000 (0.00%)
clock_network	0.0000 (0.00%)	2.410e-11 (100.00%)	2.410e-11 (100.00%)
black_box	0.0000 (0.00%)	0.0000 (0.00%)	0.0000 (0.00%)
combinational	0.0000 (0.00%)	1.647e-09 (100.00%)	1.647e-09 (100.00%)
register	5.429e-10 (58.68%)	3.822e-10 (41.32%)	9.252e-10 (100.00%)
sequential	0.0000 (0.00%)	0.0000 (0.00%)	0.0000 (0.00%)
Total	5.429e-10 (20.91%)	2.054e-09 (79.09%)	2.597e-09 (100.00%)

Table 3.10: Report threshold voltage - Original PMD_TX_PWR_CTL

Power group	**LVT leakage (%)**	**SVT leakage (%)**	**Total leakage (%)**
memory	0.0000 (0.00%)	0.0000 (0.00%)	0.0000 (0.00%)
io_pad	0.0000 (0.00%)	0.0000 (0.00%)	0.0000 (0.00%)
clock_network	0.0000 (0.00%)	4.817e-11 (100.00%)	4.817e-11 (100.00%)
black_box	0.0000 (0.00%)	0.0000 (0.00%)	0.0000 (0.00%)
combinational	0.0000 (0.00%)	1.704e-09 (100.00%)	1.704e-09 (100.00%)
register	5.429e-10 (56.65%)	4.155e-10 (43.35%)	9.584e-10 (100.00%)
sequential	0.0000 (0.00%)	0.0000 (0.00%)	0.0000 (0.00%)
Total	5.429e-10 (20.03%)	2.168e-09 (79.97%)	2.711e-09 (100.00%)

Table 3.11: Report threshold voltage - New implementation of `PMD_TX_PWR_CTL`

Since there are no critical timing paths, the majority of cells used belong to the SVT group, while UVT cells are not employed.
There are no relevant differences between the two implementations because the timing is not altered by the new module.

Chapter 4

Conclusions and future perspectives

This thesis work, carried out in collaboration with Synopsys, focused on the management of power states within the PMD, aiming to overcome the current rigidity of hardware commands. In the original system, each command is associated with a dedicated signal and controlled by specific FSMs; this makes any functional modification particularly complex, potentially requiring, in the worst case, a complete re-fabrication of the silicon.

To overcome this issue, a new module, `PMD_HW_CMD_FSM`, was designed to handle both existing and new commands, thus providing a level of programmability in hardware behavior that was not present in the previous solution. Once defined, the module was integrated into the PMD block responsible for power state transitions of the transmitter, allowing verification of its functionality within the real system context.

The work followed the traditional digital design flow, including RTL description of the component, simulations, synthesis, and quantitative analysis. Comparative simulations between the original and updated versions confirmed the full functional equivalence of the two approaches. Subsequently, synthesis of both designs allowed evaluation of the differences in terms of area and power: the results show a 3.5% increase in area and a 9.7% increase in power consumption.
These values are entirely acceptable, as they represent a minor cost considering the significant benefits introduced in terms of system flexibility and programmability. Moving from a rigid command management approach to a configurable one indeed provides substantial advantages for the future evolution of the product.

Looking ahead, future developments could include extending the new module to the reception path as well, thereby fully unifying the control architecture. Firmware modifications will also be necessary to reconfigure inputs and fully exploit the configurability introduced in the PMD. Only after this complete integration will the system be able to fully benefit from the flexibility provided by this project.

In conclusion, the work demonstrates that it is possible to introduce programmability and adaptability into the PMD without compromising system behavior and with a minimal impact on design resources, laying the foundation for a more advanced and reconfigurable approach to power-state management in SerDes.

Bibliography

[1] Synopsys. *What is SerDes?* Synopsys Glossary. 2024. URL: https://www.synopsys.com/glossary/what-is-serdes.html (cit. on pp. 1–3).

[2] Louis E. Frenzel. *Handbook of Serial Communications Interfaces: A Comprehensive Compendium of Serial Digital Input/Output (I/O) Standards*. Available via ProQuest Ebook Central. Elsevier Science & Technology, 2015. URL: https://ebookcentral.proquest.com/lib/polito-ebooks/detail.action?docID=2189944 (cit. on p. 2).

[3] David R. Stauffer, James T. Mechler, Mark A. Sorna, Kerry Dramstad, Craig R. Ogilvie, Ahmed Mohammad, and John D. Rockrohr. *High Speed SerDes Devices and Applications*. Springer Science & Business Media, 2008 (cit. on p. 4).

[4] Peter D. Minns and Ian Elliott. *FSM-Based Digital Design Using Verilog HDL*. ProQuest Ebook Central. John Wiley & Sons, Incorporated, 2008. URL: https://ebookcentral.proquest.com/lib/polito-ebooks/detail.action?docID=470251 (cit. on pp. 5, 6).

[5] Tertulien Ndjountche. *Digital Electronics 3: Finite-State Machines*. ProQuest Ebook Central. John Wiley & Sons, Incorporated, 2016. URL: https://ebookcentral.proquest.com/lib/polito-ebooks/detail.action?docID=4722457 (cit. on p. 6).

[6] Mohammed Ferdjallah. *Introduction to Digital Systems: Modeling, Synthesis, and Simulation Using VHDL*. ProQuest Ebook Central. John Wiley & Sons, Incorporated, 2011. URL: https://ebookcentral.proquest.com/lib/polito-ebooks/detail.action?docID=69764 (cit. on p. 7).

[7] Ashok B. Mehta. *ASIC/SoC Functional Design Verification: A Comprehensive Guide to Technologies and Methodologies*. ProQuest Ebook Central. Springer International Publishing AG, 2017. URL: https://ebookcentral.proquest.com/lib/polito-ebooks/detail.action?docID=4890725 (cit. on p. 7).

[8] Inc. Synopsys. *VC SpyGlass: RTL Static Signoff Platform*. 2025. URL: https://www.synopsys.com/verification/static-and-formal-verification/vc-spyglass.html (cit. on p. 22).

[9] Inc. Synopsys. *VC SpyGlass Lint: RTL Design Analysis*. 2025. URL: https://www.synopsys.com/verification/static-and-formal-verification/vc-spyglass/vc-spyglass-lint.html (cit. on p. 22).

[10] Inc. Synopsys. *VC SpyGlass CDC: Clock Domain Verification*. 2025. URL: https://www.synopsys.com/verification/static-and-formal-verification/vc-spyglass/vc-spyglass-cdc.html (cit. on p. 23).

[11] Inc. Synopsys. *Verdi: Automated Debug System*. 2025. URL: https://www.synopsys.com/verification/debug/verdi.html (cit. on p. 23).

[12] Synopsys, Inc. *Synopsys Design Compiler® NXT*. 2025. URL: https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/design-compiler-nxt.html (cit. on p. 25).

[13] Synopsys, Inc. *Synopsys PrimePower: RTL to Signoff Power Analysis*. 2025. URL: https://www.synopsys.com/implementation-and-signoff/signoff/primepower.html (cit. on p. 31).