Hes·so
Haute Ecole Spécialisée de Suisse occidentale
Fachhochschule Westschweiz
University of Applied Sciences and Arts Western Switzerland

Master of Science HES-SO in Engineering

Orientation : Technologies industrielles (TIN)

WATCHMAN

at the University of Hawaii

UNIVERSITY of HAWAI'I MĀNOA

Realized by
Jonathan James Hendriks

Professor
Marco Mazza, HEIA-FR
marco.mazza@hefr.ch

External Expert
Gary S. Varner, University of Hawai'i at Manoa
varner@phys.hawaii.edu

Kurtis Nishimura, University of Hawai'i at Manoa
kurtisn@phys.hawaii.edu

Honolulu, University of Hawai'i, February 2019

Accepted by the HES-SO Master (Switzerland, Lausanne) on the proposal of
Prof. Marco Mazza, supervisor of the Master Thesis

Gary S. Varner, main External

Honolulu, February the 8th 2019

Prof. Marco Mazza
Supervisor and professor at the HEIA-FR

Prof. Gary S. Varner
Main Expert

Mahalo

"*Boston or Hawaii?...HAWAII!*"

I would like to thank, my professor Marco Mazza for giving me the opportunity to fly over to Hawaii and to finish my Master studies on this beautiful island.

"*Do you have 5 seconds ?*"

To Professor Gary Varner and Kurtis Nishimura a great thank for their help and wisdom along this journey.

"*Hey Burrito!*"

Special thanks go to Anthony for his trust and good company all along this project.

And a finally big thank you to all the people who supported me on this continent or 12 hours away!

Abbreviation

Term	Description
aFIFO	Asynchronous FIFO
AIT	Advanced Instrumentation Testbed
ASCII	American Standard Code for Information Interchange
ASIC	Application-specific integrated circuit
AXI	Advanced eXtensible Interface
CPU	Control Processing Unit
CVS	Comma-seperated values
DAC	Digital to Analog Converter
DC	Direct Current
DFF	D-Type Flip-Flop
DLL	Delay Lock Loop
DMA	Direct Memory Access
FIFO	First-In Firs-Out
FPGA	Field-Programmable Gate Array
GCC	Gray Code Counter
GUI	Graphical User Interface
IACT	Imaging Atmospheric Cherenkov Telescopes
ILA	Integrated Logic Analyzer
LE	Leading Edge
LSB	Least Significant Bit
LUT	Look Up Table
LVDS	Low-Voltage Differential Signaling
MMCM	Mixed-Mode Clock Manager
MSB	Most Significant Bit
OBUFDS	Differential signals buffers
PL	Programmable Logic
PLL	Phase Lock Loop
PS	Processing System
RTL	Register-Transfer Level
SRB	Small Round Buffer
TARGET	TeV Array Readout Gigasample-per-second Electronics with Trigger
TCL	Tool Command Script
TE	Trailing Edge
TPG	Test Pattern Generator
UART	Universal Asynchronous Receiver-Transmitter
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
WATCHMAN	WATer CHerenkov Monitor for Anti-Neutrinos

Contents

1 Introduction

1.1 WATCHMAN

The WATer CHerenkov Monitor for Anti-Neutrinos (WATCHMAN) is the first experiment planned for the Advanced Instrumentation Testbed (AIT) facility, which has just began construction at the Boulby Underground Lab in Great Britain. The goal of WATCHMAN is to develop technology and data analysis techniques to demonstrate the ability to monitor nuclear reactors from tens of kilometers. This proof of principle is to be incorporated into the future Nuclear Non-Proliferation Treaties. For this study, WATCHMAN will be able to use the Hartlepool Nuclear Power Station (a distance of 25 kilometres away from the Boulby Lab) as the reactor source of anti-neutrinos. [1]

1.1.1 Description of the Watchman Neutrino detector

The fission reactions in a nuclear plant produce anti-matter neutrinos. The interaction with a hydrogen atom results in a prompt flash of light. This special interaction has only tiny chances to be observed, even though nuclear reactors produces large numbers of neutrinos about, i.e 10^{20} per seconds. Water contains a hydrogen atom, the ground surrounding the reactor is filled with water, but only a very small quantity. The probability that a neutrino hits one of these elements is extremely small. In consequence, it takes lots of water volume to be able to see such an event. In addition to this big volume, the water can be doped with a chemical that enhances the light (Gadolinum as a capture agent).

Figure 1.1: 3D Rendering of the WATCHMAN Neutrino detector

1.1.2 Collaboration

WATCHMAN is a collaboration between universities in the United States and United Kingdom. The data acquisition system is undefined, but an evaluation of 3 systems from 3 different universities is ongoing : CAEN, ANNIE and TARGET C. The collaboration contains group for Photomultiplier tubes, testing, cabling and so on. The universities enrolled in this collaboration are :

- UC Davis
- UC Berkeley
- Brookhaven National Lab
- University of Hawaii - TARGET C
- Iowa State
- Penn State
- University of Pennsylvania
- University of Sheffield

1.1.2.1 CAEN System

CAEN system is an industrial ADC from Renesas. The resolution is 14 bits and sampling speed is up to 500 MSPS. Data collection is managed by a Cyclone IV FPGA from Intel. The system is ready to go, but the price is very high, a single channel (one PMT) is around 1200$. Another issue with this system, it is trigger-less. The ADC is continuously digitizing, software has to automatically detect PMT pulses and as consequence the system is running on full capacity. Therefore the power consumption is relatively high 15kW/PMT. However CAEN has rack support for their boards, so the system overall is built and ready to use.

1.1.2.2 ANNIE System

ANNIE system is based on an ADC from Texas Instrument (TI), the ADS5407 which has a resolution of 12 bits and sampling speed of 50MHz. Readout of the ADC data is done on a FPGA Arria V also from Intel. A second board is used in complement to the first one, it will transmit the data to the computer using 18 SFP (small form-factor pluggable) optical module transceiver ports connected to the multigigabit transceivers of the Arria V FPGA. This system is similar to the CAEN system, it is easily implemented, but power consumption is extremely high again and the costs are also very high.

1.1.2.3 Target C System

The Hawaii WATCHMAN readout module is a system that can handle a high density of PMTs, while digitizing the signals at a high sampling rate using TARGET-based ASIC technology developed by the University of Hawaii. The TARGETC's Wilkinson ADC architecture allows a sampling rate of 1 Giga-samples per second with relatively low power for 16 channels simultaneously. Employing commercial System-On-Module (SOM) with a ZYNQ-based FPGA, the system offers a trigger-less, dead-time less (at 10kHz rate), low-cost solution for the ADC readout requirements of modern High Energy Physics detectors.

1.1.2.3.1 System Overview

The prototype board is using a MicroZed with a Zynq 7010 plugged in a FMC Carrier board on which is plugged the FMC prototype TARGET C board.

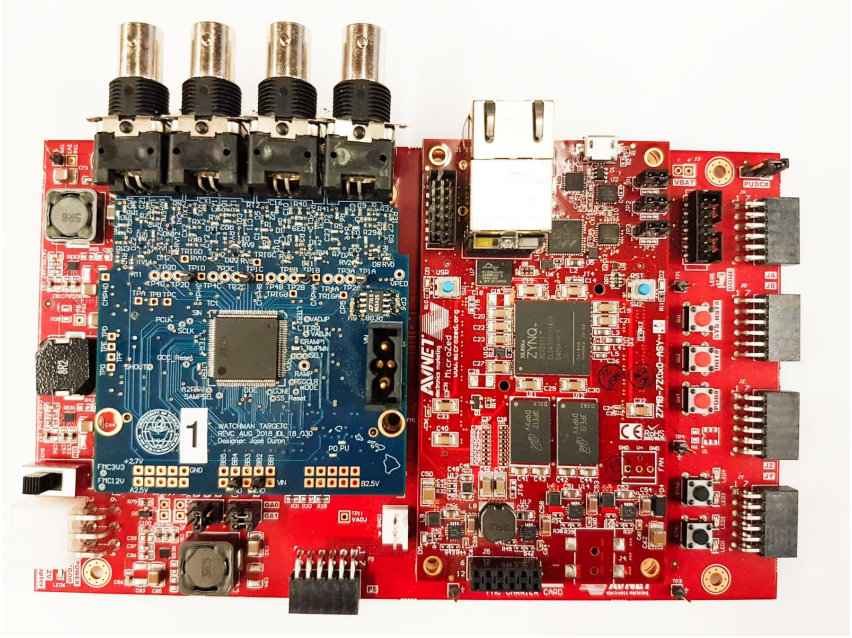


Figure 1.2: Target C FMC Prototype Board

1.2 Objectives and Management

The objectives set before coming to Hawaii were the following :
"The current plan is that all single photon (or larger) signals will be digitized. Feature extraction of the raw waveforms (time and charge) need to be done in Firmware to provide adequate throughput. These feature extracted packets are then collected in a master event-building, which also will require efficient Firmware to sort the hits from throughout the detector and implement a windowing algorithm to determine when a pattern of hits is of interest to record or not. To reduce background data, for instance, it may be necessary to veto, or at least pre-scale cosmic ray muon events, which are a background to the neutrino events of interest. Initial work will involve extensive simulation, while the hardware is being developed. Initial prototyping can also be verified using FPGA evaluation boards. Documentation of the algorithms implemented will be vital to ensure they can be maintained in the future.", pulled from the actual objectives set by Prof. Marco Mazza, in TM2018-19_Hawaii_SinglePhotonDigitalization.

Upon arrival at Hawai'i University, it became clear that these objectives were not realistic as the prototype was not yet built. Thus, my objectives changed to first developing the prototype and then testing it.

1.2.1 Project Structure

The Watchman TARGETC project is supervised by Dr. Gary Varner and Dr. Kurtis Nishimira. It is then divided into Hardware, Firmware and software. Each section is affiliated to a student, in my case, I am responsible of the firmware, see figure 1.3.

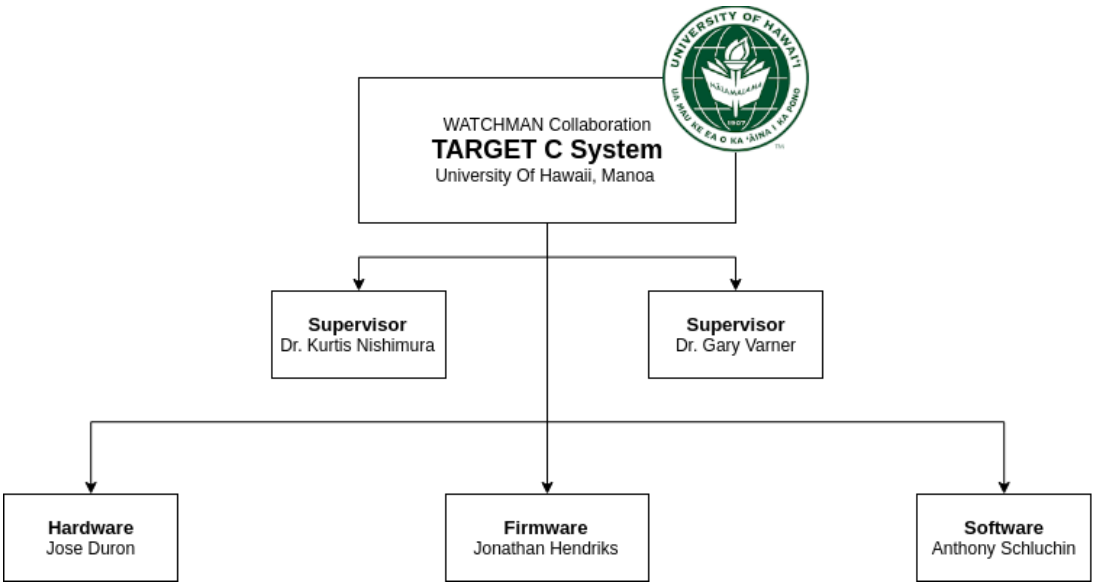


Figure 1.3: Organization and team

1.2.2 Objectives

My primary objectives after redefining the goals of the project and discussion with my supervisors are:

- ☐ Development of the data interface between the PS and PL (AXI-Lite and AXI-Stream).
- ☐ Data readout from TARGETC
- ☐ Elaborate the trigger system
- ☐ Round Buffer algorithm to deal with new events and previous event still being digitized

The secondary objectives derived from the primary ones are the following:

- ☐ Setting up an AXI-Lite Register Module
- ☐ VHDL Test component for AXI-Stream
- ☐ VHDL Project for AXI-Stream DMA with PL-PS
- ☐ Rough understanding of TARGETC ASIC
- ☐ Elaborate Datasheet for TARGETC ASIC
- ☐ Register Communication with TARGETC
- ☐ Sampling and storage of data with TARGETC
- ☐ Wilkinson digitization process
- ☐ Readout Data from Target C
- ☐ Round Buffer elaboration in VHDL
- ☐ Trigger system based on trigger information
- ☐ Analyze the received data

1.2.3 Planing

The Watchman project is a long term project which started in 2016 and runs at least until 2020. Since the prototype stage was not even started, the new objectives were to get a first system set up. The various different projects in the Instrumentation Development Laboratory (IDLAB) were using a previous design, the TARGETX on which the new TARGETC was based on. The TargetC's only existing documentation is the ASIC schematic layout composed of roughly 200 pages.

The only milestone was the Watchman Collaboration meeting on the **10th and 11th of January 2019**, at which time the different Data Acquisition System from the different Universities were to be evaluated and presented with good advancement and if possible demonstration.

1.3 Document Structure

This document is separated into several parts :

TARGET C

The TARGET C is one of the critical points of the project, the ASIC has never been used in projects. Therefore it is important to understand the component and document it. I will cover the essential of the Target C architecture and how to interface it with the information pulled out of the ASIC's schematics and the global explanations of Dr. Gary Varner.

PL : Programmable Logic

This section will cover the firmware structure and difficulties to interface the ASIC from the PL side of the Zynq. The main point in this chapter is the Round buffer concept which is a new feature of the Target C compared to its predecessors.

PS : Processing System

The small portion of code written to test the functionality of programmable logic with the processing system.

Further Development

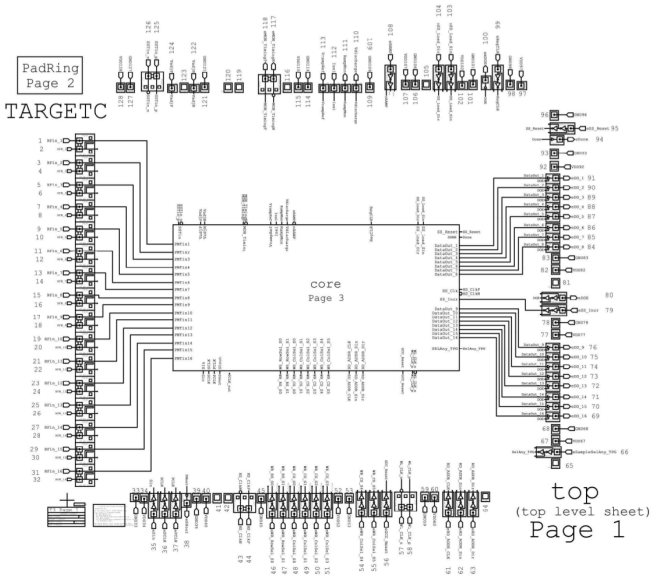
After simulating a complete system, some improvements were made, some issues were resolved and others were investigated.

Conclusion

Finally all results concerning the objectives will be discussed in this section.

2 TARGET C

(a) Prototype board



(b) Top level Schematic

Figure 2.1: TARGETC

TARGET C is a new application-specific integrated circuit (ASIC) of the TARGET family, which stands for TeV Array Readout Gigasample-per-second Electronics without Trigger (TARGET). This circuit is designed by Dr. Gary Varner at the University of Manoa Hawai'i. This ASIC is designed for the readout of signals from photosensors in the cameras of imaging atmospheric Cherenkov telescopes (IACTs) for ground-based gamma-ray astronomy. The TARGET family is composed of :

- TARGET 5
- TARGET 7
- TARGET X
- TARGET C

The main difference from the TARGETX to C is the control over storage cells and the triggering system, these details are explained later in this document.

2.1 Overall Architecture

The TARGETC can sample 2 x 32 samples x 16 channels at once. To do so a capacitor array composed of 64 sampling capacitors (SaC0 to SaC63) which are charged by a switch command signal generated by the SSTIN provided by the FPGA and SSPIN clock internally generated, see figure 2.2. For simplicity, this illustration only shows the channel RFIN_1 but there is 16 of them in parallel, consequently all sampling on the 16 channels is done at the same time. The sampling frequency depends on the SSTIN signal, generally the clock period is set to 64 ns. Each switch command is delayed by 1ns, but the width of the pulse is larger than 1ns giving the time for the capacitor to charge up. The delay circuit is composed of two inverters, one fixed delay and the other is adjustable (coarse and fine tune). Adjustment value is determined by a DLL (Delay Lock Loop) or can be written in a register. The signal SW0 is formed by logic combination of the superposition of two clocks and is enabled 1ns earlier than SW1 signal and 2ns earlier than SW2 signal, see figure 2.3. Assuming that each capacitor is switched ON and OFF at 1ns of interval, the input signal is sampled at a frequency of 1 Giga-Sample per second (1GS/s).

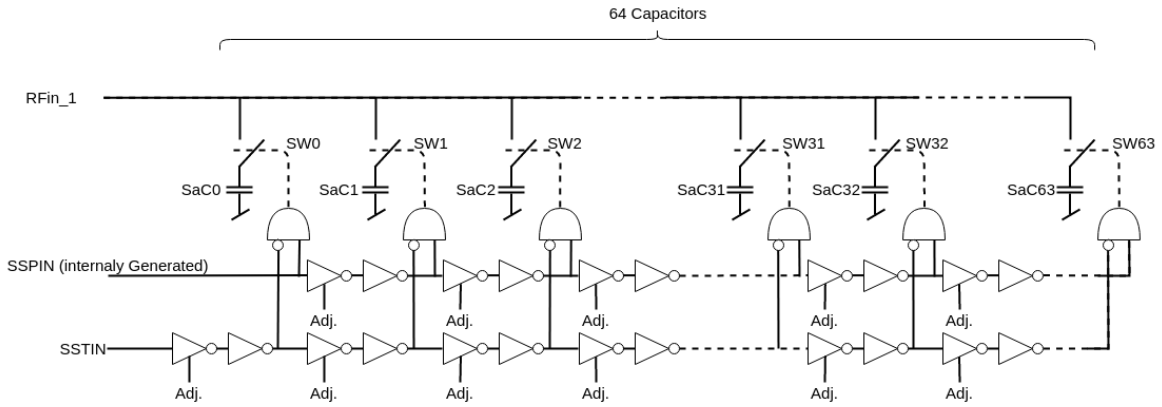


Figure 2.2: Capacitor array for sampling analog input

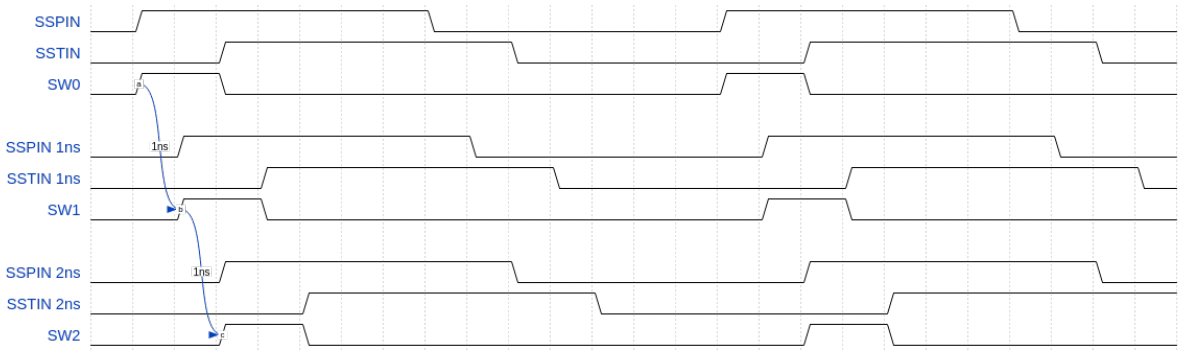


Figure 2.3: Switch command Pulse Generator for 1ns delay between samples

The storage address is commanded by the FPGA side using the ASIC input pins, WR_COL_SEL(5:0) and WR_ROW_SEL(1:0). The storage location is formed of capacitors (StC0 to StC63). The location addressed is latched onto the internal address bus by a write address sync (internal signal). Once the address is stable inside the ASIC, a second internal signal is enabled (write strobe). This signal switches ON and OFF the transmission lines to store the analog value from SaC0-SaC63 into the storage cells StC0-StC63. The 64 samples are stored in two different parts

of the analog memory. The ASIC separates the 64 samples into 2 windows of 32 samples (EVEN and ODD part). The LSB bit for row selection is this EVEN and ODD window select. The total analog memory is 32 samples x 8 rows x 64 columns = 16K samples. The FPGA must keep track which part of memory is occupied and which is free. This write command is done in parallel for the 16 channels of the TARGETC.

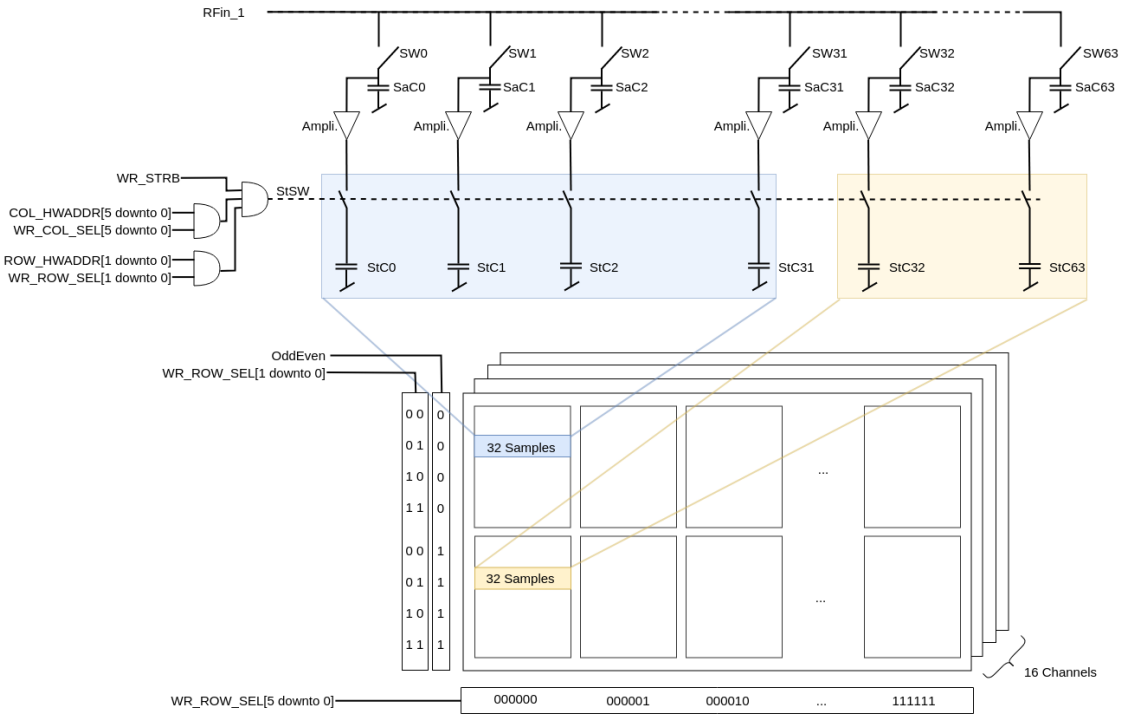


Figure 2.4: Analog storage area of 16K cells per channel

The digitalization is performed in parallel for the 32 samples per channel. It is triggered by sending the address through a serial communication (RDAD_clk, RDAD_sin and RDAD_dir). The address is decoded and triggers a read command which turns on the Wilkinson comparators of the memory location (normally off for power consumption).

The Wilkinson ramp is started by enabling the ramp signal. The ramp parameters are defined by a simple linear function $y = a*x+b$. The argument a is the slope of the ramp commanded by the ISEL register. The second argument b is defining the offset voltage of the function, this offset is the voltage discharge of the wilkinson capacity defined by a register VDISH. The Wilkinson ADC compares the voltage of a capacitor to a digital counter, when the comparison is equal the analog value is represented by the digital value inside the counter. Therefore the frequency at which this counter is increment must be linked to charge speed of the capacitor. So the Wilkinson clock period multiplied by the counter's length defines the total time of charge. The gray code is used for changing only 1 bit at the time and avoiding dangerous transition levels and meta-stabilities, like for example the transition from 127 to 128 (01111111_2 to 10000000_2). The figure 2.5 shows the different actors for varying the charge parameters of the capacitor.

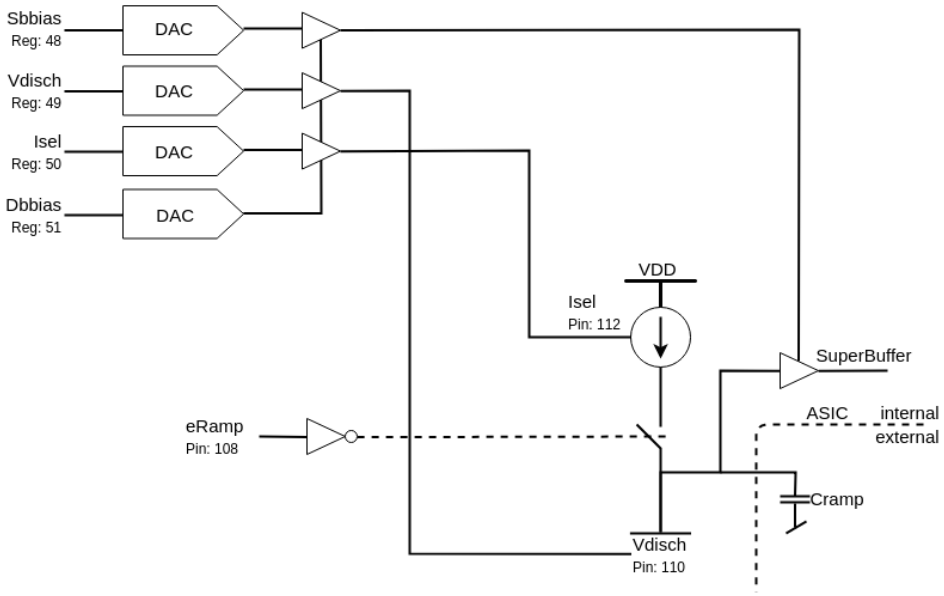


Figure 2.5: Principle scheme for Vramp

Once the voltage of the charge capacitor is above the storage cell voltage, the comparator switches from LOW to HIGH. This transaction latches the content of the Gray counter into a sample register. This digital value is the representation of the analog value in memory. The digitization process is finished. The eDONE signal is pulled high when all conversions are over. To put in another way, all comparators have switched from LOW to HIGH, when the 32 samples of each 16 channels are digitized. Only then is the eDONE signal pulled HIGH, see Figure 2.6.

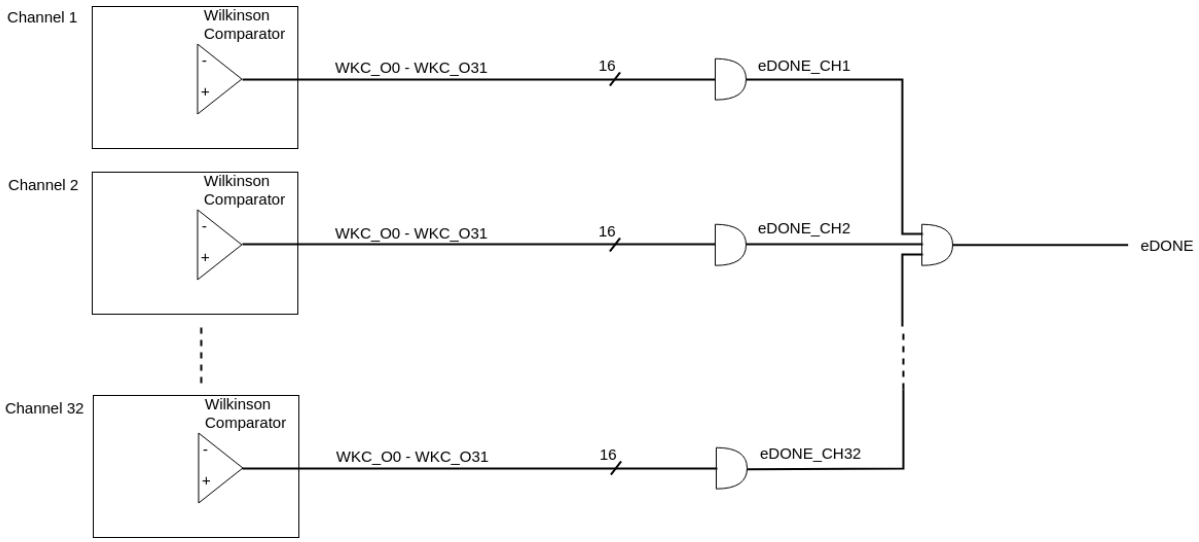


Figure 2.6: Logic circuit for eDONE signal

All samples of the 16 channels are serialized on the Data output pins DO0 to DO15. The SS_INCR signal pulled HIGH loads a sample register onto the data line, on the HIGH to LOW transition the data line is latched into the shift register. This generated pulse on SS_INCR will also increase the sample counter, on the next HIGH transition the next sample will be loaded and so on.

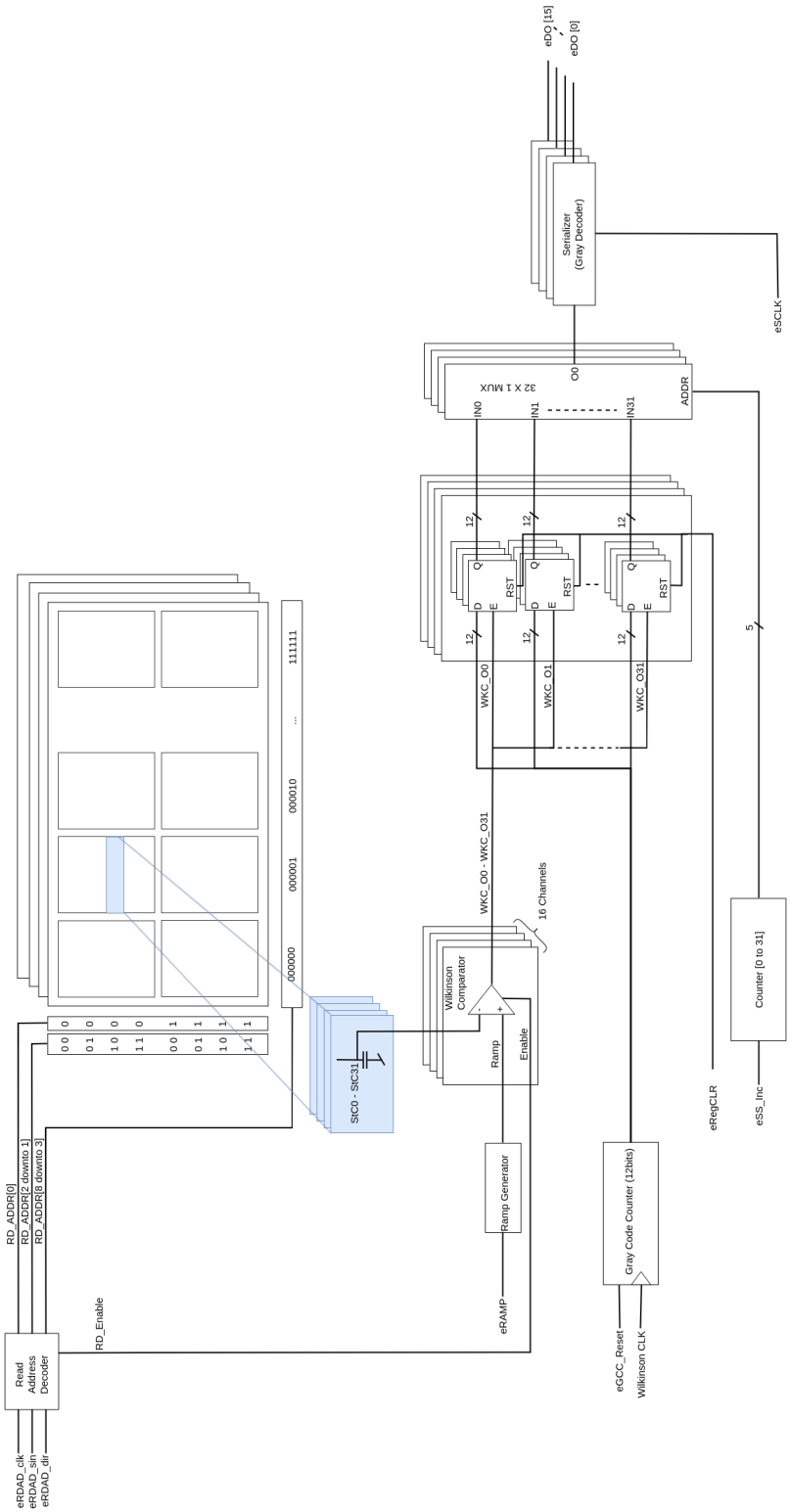


Figure 2.7: Full Readout and Digitalization

The ASIC has a built-in Delay Lock Loop, that once enabled and stable, will adjust the timing to temperature fluctuation. The figure 2.8 illustrates the DLL's different parameters.

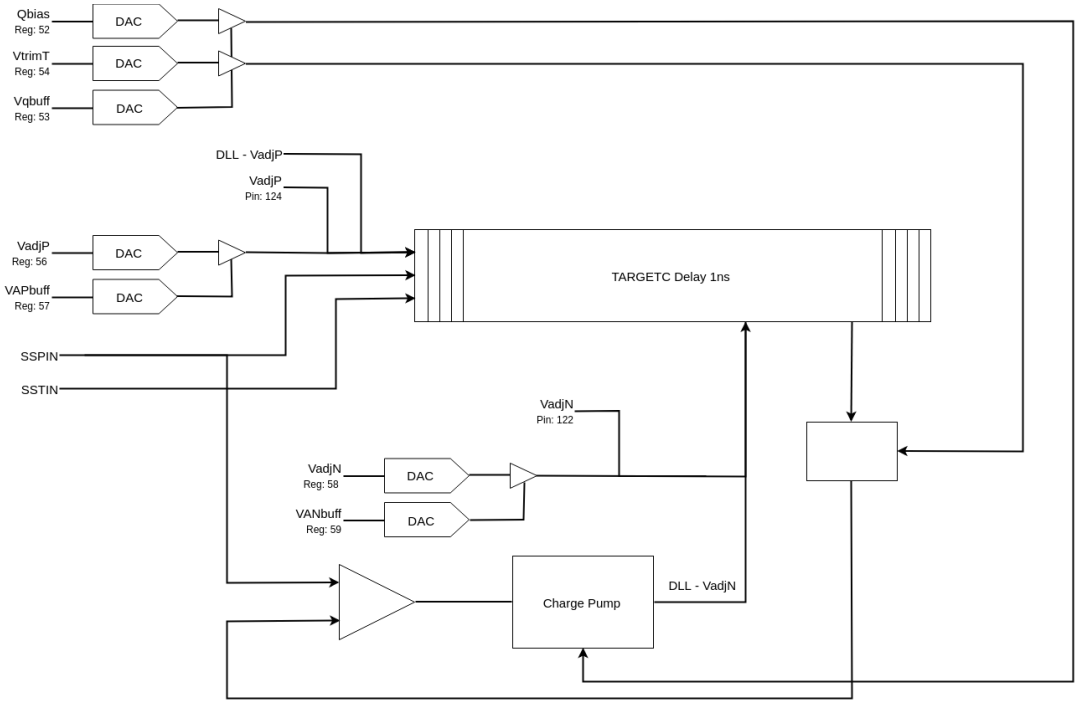


Figure 2.8: Principle scheme for DLL

2.2 New Features

2.2.1 External Trigger

The first difference from previous design like TARGETX, which is used in Belle II detector (over 20k channels), is the triggering system. On the TARGET X, the trigger circuit is internally managed and has the capabilities to trigger on any of the 16 channels. In TARGETC design the trigger system is removed, which means that the full control comes back to the FPGA or the interfacing device. As a consequence, the WATCHMAN TARGETC prototype board uses four operational amplifiers used as comparators. Four independent and adjustable threshold voltages are compared the highest gain lines (x10), which are Channel input 3, 7, 11 and 15 (see figure 2.9).

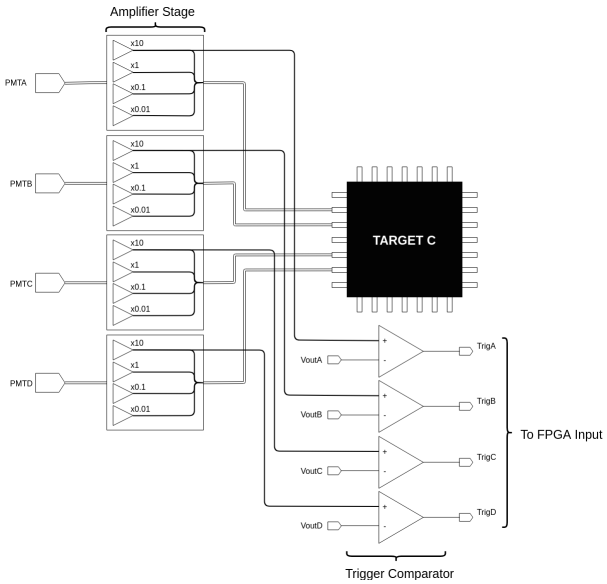


Figure 2.9: Principle scheme for External Trigger Circuit on the TARGET C Prototype

2.2.2 External Storage Control

The second difference from previous designs is the control over the analog memory storage. Earlier versions used an internal counter with a write enable, the parameter was turned OFF if the cell was not to be re-written. Unfortunately this also means that the data at that exact moment is not written in any memory or anywhere else, it is simply lost. In order to avoid this issue, the control circuit is brought out of the TARGET and left to the user to implement any algorithm to deal with storage control.

2.3 Communication with the ASIC

2.3.1 Register Write Interface

The ASIC contains a set of registers to enable internal DACs, which have different purposes such as fine tune of current charge, enable/disable Test Pattern Register, and so on. These register can only be accessed through the register interface pins, see table 2.1. This type of communication is a 19-Bit wide word followed by a signal transaction between SIN and PCLK,

see figure 2.10. The SIN signal must be shifted out on each rising edge of the SCLK clock, the ASIC samples the SIN input on the falling edge.

Pinout(s)

Pin#	Pin Name	Pin Type	Description
35	SIN	Digital Input	Serial Input data
36	SCLK	Digital Input	Serial clock advance
37	PCLK	Digital Input	Parallel clock load
38	SHout	Digital Output	Serial Shift Out

Table 2.1: Register interface Pins

Requirement(s)

Req#	Description
1	PCLK Pulse shall be at least 100 ns wide
2	SIN shall start before PCLK last pulse
3	MSB First
4	ASIC samples on falling edge, data is shifted out on the rising edge

Table 2.2: Requirement for the register write interface

Payload

Bits	Description
18 - 12	Register Address
11 - 0	12-Bit Data

Table 2.3: TARGETC 19-Bit Register payload

Interface sequence

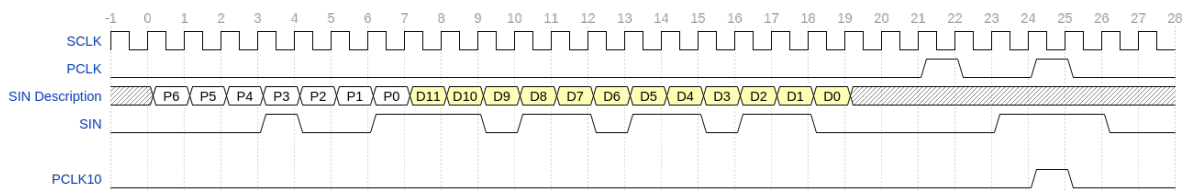


Figure 2.10: Register Write Sequence for regsiter 10 with the binary data 110110110110_2

2.3.2 Delay Lock Loop

The Delay Lock Loop is one of the main reasons that the Target C can sample at 1GS/s and it is very important to initialize it correctly. To work with the DLL a few parameters must be written.

- Qbias controls the charge pump of the DLL.
- VQBUFF enables the DAC Qbias
- VADJP controls the Trailing edge (TE) of the signals
- VADJN controls the Leading edge (LE) of the signals
- VAPBUFF enables the DAC VADJP
- VANBUFF enables the DAC VADJN

Requirement(s)

Req#	Description
1	VadjN must be stable in order to consider the DLL locked.

Table 2.4: Requirement for DLL

Interface Sequence

1. At initialization, VQBUFF and VANBUFF are set to 0
2. VADJN set to approximative value close to the locked one $y = 4095 * x/2.5$
3. VANBUFF set 1100, kisck-start the DLL
4. VQBUFF set to 1062, VADJN value are fighting
5. VANBUFF set to 0, DLL is going to stabilize itself
6. Wait until VADJN is stable
7. Control the MonTiming signals, the LE and TE parameters have to be set correctly for signals to perform as expected.

2.3.3 Storage Control

UPDATE, 28th of January 2019 : The storage address update sequence was wrongly understood and implemented due to lack of documentation. The previous project implementation will not be discussed and only the actual design is discussed, the correct storage address sequence is illustrate in figure 2.11.

The real address is the current window being sampled, the sampling value is however only valid a small amount of time after the sampling process due to the capacitor. A rising edge on the write address sync signal will latch the address in the storage memory lines to select the correct storage cell. This signal is initiated prior to a write strobe sequence, because , these previous

lines must stabilize themselves. The write strobe signal is level sensitive, a HIGH level will transfer the voltage from sampling to storage cells.

In summary the storage address is updated on each falling edge of SSTIN, in order to prepare a new memory location for the next sample cycle. The storage address is 8-bit wide bus. The TARGETC has a total of 512 windows and two of which are sampled sequentially during a SSTIN period, as a result the storage address is only 8-bit long.

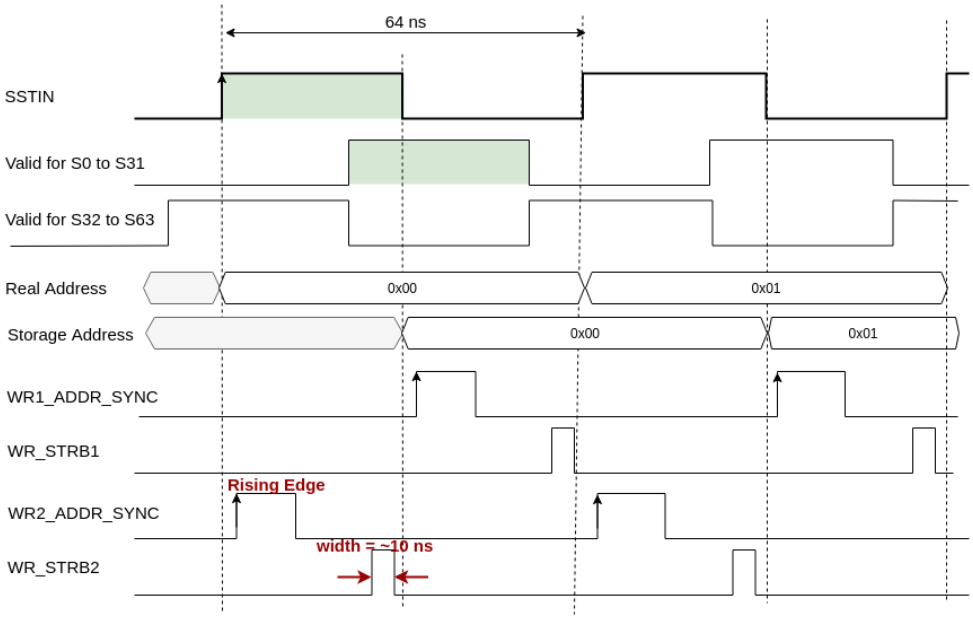


Figure 2.11: Timing principle for write address and write strobe signals

Pinout(s)

WR_RS_S are the lower bits and WR_CS_S are the upper bits of the storage address.

Pin#	Pin Name	Pin Type	Description
46	WR_RS_S0	Digital Input	WR Row Select Addr. 0
47	WR_RS_S1	Digital Input	WR Row Select Addr. 1
48	WR_CS_S0	Digital Input	WR Column Select Addr. 0
49	WR_CS_S1	Digital Input	WR Column Select Addr. 1
50	WR_CS_S2	Digital Input	WR Column Select Addr. 2
51	WR_CS_S3	Digital Input	WR Column Select Addr. 3
54	WR_CS_S4	Digital Input	WR Column Select Addr. 4
55	WR_CS_S5	Digital Input	WR Column Select Addr. 5

Table 2.5: Storage address interface Pins

Requirement(s)

Req#	Description
1	WR_RS_S(1-0) and WR_CS_S(5-0) shall be updated on the falling edge.

Table 2.6: Requirement for the register write interface (Update : 28th of January 2019)

Interface sequence

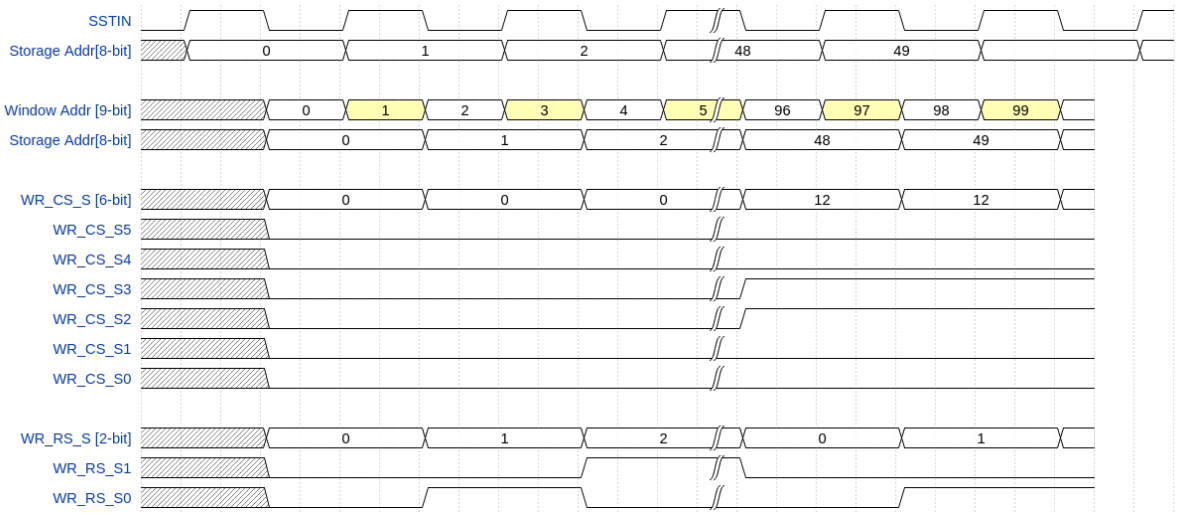


Figure 2.12: Storage address update sequence (Update : 28th of January 2019)

2.3.4 Window Readout

As mentioned the storage address is a 8-Bit wide address. The readout is a 9-Bit address, the extra bit enables the selection between the 1st or 2nd window sampled. In the TARGET world, we speak of EVEN or ODD window, the 1st window being the EVEN and respectively the 2nd the ODD window.

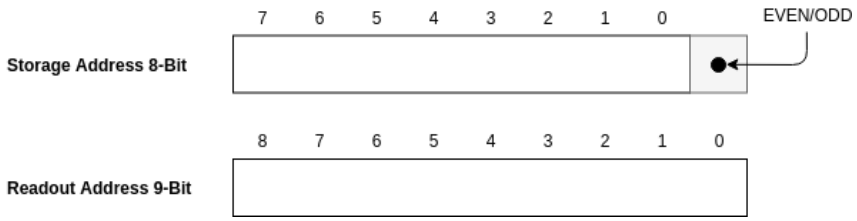


Figure 2.13: Storage VS Readout address

From previous TARGET designs, the readout address is swapped and the extra bit is placed in different places. This should not be the case for TARGETC.

Pinout(s)

Pin#	Pin Name	Pin Type	Description
61	RDAD_clk	Digital Input	Read Address Set SCLK
62	RDAD_sin	Digital Input	Read Address Set Serial Input
63	RDAD_dir	Digital Input	Read Address Set DIR

Table 2.7: Read address interface Pins

Requirement(s)

Req#	Description
1	TARGETC samples on the rising edge of RDAD_CLK. Data shall be shifted during the LOW period of the clock on SIN.
2	MSB is shifted out first on RDAD_SIN

Table 2.8: Requirement for the readout interface

Interface sequence

Figure 2.14: Example of Readout Address sequence for window 52_{10}

2.3.5 Wilkinson Digitization

The read module inside the ASIC decodes the address entered on the readout interface and the corresponding internal switches are powered on, connecting the window's cells comparator outputs to the latch inputs of the sample registers (Output shift register). The Wilkinson counter is reset using the signal GCC_Reset (11-Bit Gray Code Counter). The ramp signal (eRamp) is enabled, which enables the charge of the Wilkinson capacitor. It is important to synchronize eRamp signal with the release of GCC_reset signal, because the counter is incrementing on every rising edge of the Wilkinson clock. Once the comparator stored value is equal to the capacitor's value, the comparator's output switches from LOW to HIGH, this has the effect of latching the counter's value in the sample register. This latched value corresponds to the gray coded digital value of the stored sample. After the digitization is completed the eRamp signal must be kept HIGH during all the sample readout. Finally the Wilkinson capacitor should have enough time to discharge completely.

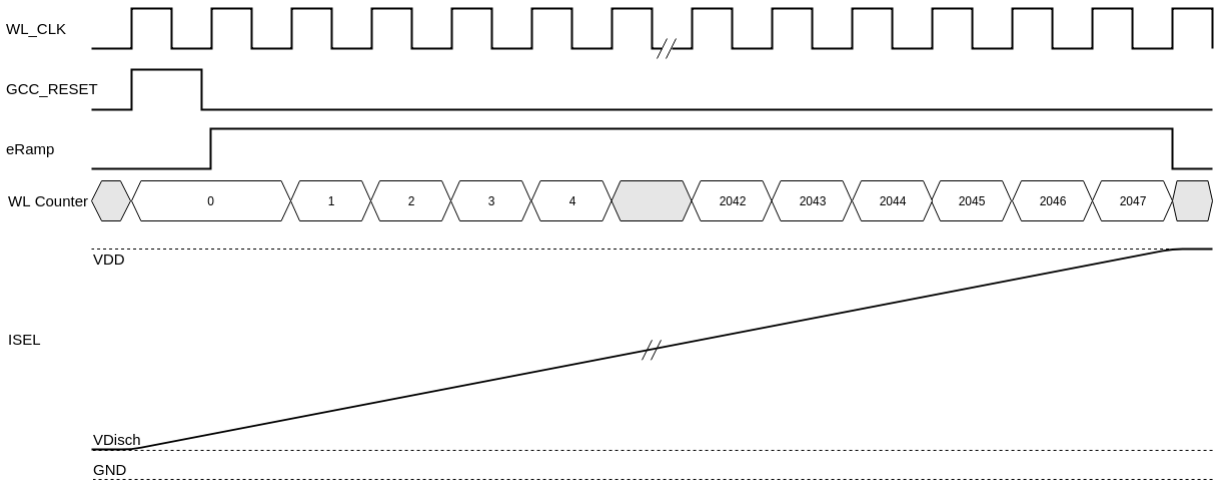


Figure 2.15: Wilkinson conversion signals for full range (VDD=2047)

Pinout(s)

Pin#	Pin Name	Pin Type	Description
56	GCC_Reset	Digital Input	Gray Code Counter Reset
57	WL_CLK_n	LVDS Input	Wilkinson Clock LVDS
58	WL_CLK_p	LVDS Input	Wilkinson Clock LVDS
108	eRamp	Digital Input	Wilkinson Ramp control
110	Vdischarge	Analog Output	Wilkinson Ramp Start voltage
111	RampMon	Analog Output	Buffered copy of Wilk Ramp
112	ISEL	Analog Output	Monitor for Wilk Ramp I (V out)
113	VrampRef	Analog Output	Charging node

Table 2.9: Storage address interface Pins

Requirement(s)

Req#	Description
1	The Wilkinson frequency must correspond with the charge of the capacitor defined by the DAC register ISEL.
2	The counter must be reset before starting a new conversion.
3	The eRamp signal is held HIGH during all sample readout.

Table 2.10: Requirement for the wilkinson interface

Interface sequence

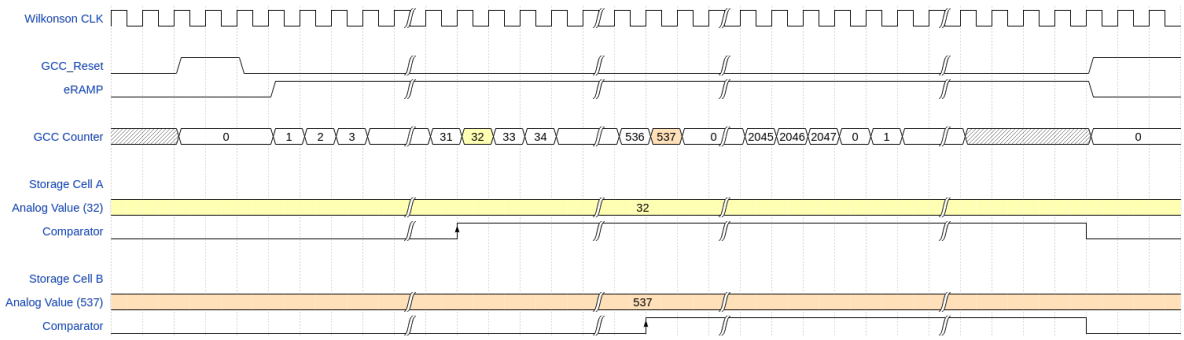


Figure 2.16: Wilkinson sequence for the digitization of a selected window

2.3.6 Sample Readout

The sample readout is an internal counter that can be incremented on the rising edge of SS_Incr or reset using eSS_Reset (Active HIGH). Once the 32 samples from all channels have been read, the counter goes back to sample 0 by applying a pulse on SS_Incr. The falling edge of SS_Incr charges the values into the registers. This is why the pulse width is important, because of the disposition of the registers in the ASIC, some lines take longer than others to arrive. The pulse should be long enough for the data to be stable before latching it into the output registers.

Pinout(s)

Pin#	Pin Name	Pin Type	Description
43	HSclkN	LVDS Input	Data Shift-out Clock
44	HSclkP	LVDS Input	LVDS
66	SmplSl_Any	Digital Input	Select between TPG or sample value
79	SS_Incr	Digital Input	Sample Select increment
80	DOE	Digital Input	Data Output Enable
69	DO_16	Digital Output	Serial Data Out Ch. 16
70	DO_15	Digital Output	Serial Data Out Ch. 15
71	DO_14	Digital Output	Serial Data Out Ch. 14
72	DO_13	Digital Output	Serial Data Out Ch. 13
73	DO_12	Digital Output	Serial Data Out Ch. 12
74	DO_11	Digital Output	Serial Data Out Ch. 11
75	DO_10	Digital Output	Serial Data Out Ch. 10
76	DO_9	Digital Output	Serial Data Out Ch. 9
84	DO_8	Digital Output	Serial Data Out Ch. 8
85	DO_7	Digital Output	Serial Data Out Ch. 7
86	DO_6	Digital Output	Serial Data Out Ch. 6
87	DO_5	Digital Output	Serial Data Out Ch. 5
88	DO_4	Digital Output	Serial Data Out Ch. 4
89	DO_3	Digital Output	Serial Data Out Ch. 3
90	DO_2	Digital Output	Serial Data Out Ch. 2
91	DO_1	Digital Output	Serial Data Out Ch. 1
95	eSS_Reset	Digital input	Sample Select reset

Table 2.11: Sample Readout interface Pins

Requirement(s)

Req#	Description
1	SS_Incr pulse shall be wide enough (latency in ASIC).

Table 2.12: Requirement for the sample select interface

Interface sequence

The interface sequence is very particular for the first sample. The SS_Reset is brought LOW and after the SS_INCR is brought HIGH. Internally this effect will load the register of sample 0 on the bus. After a stabilization time, SS_Reset is disabled and then SS_INCR is set LOW. The data of the register will be latched into the output shift register.

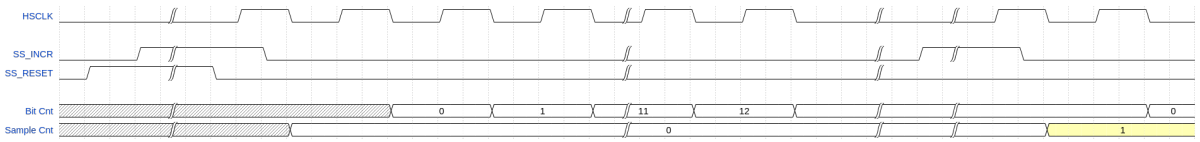


Figure 2.17: Sample select readout sequence

3 PL : Programmable Logic

This chapter covers the firmware architecture to interface with the TARGETC and acquire the data to send it back to the PS (processing system).

3.1 Clock Management

A clock management circuit is necessary to interface each external clock and derive a reference clock for the circuit. The TARGETC has 5 different clocks inputs:

- SSTIN : fixed period clock with 64 ns period (Frequency = 15.625 MHz, Duty cycle = 50%, LVDV signal).
- HSCLK : slow clock for data ouput (LVDS signal)
- WL_CLK : wilkinson clock, no particular frequency but should be able to change the frequency (LVDS signal)
- RDAD_CLK : Read Address clock
- SCLK : Serial clock for registers, slow clock

At the beginning of the project, the Vivado clock wizard IP Core generator was employed. However as the project evolved into a complexer system, Vivado's IP was becoming more and more difficult to change configuration at which point the decision was made to migrate to Xilinx's primitives (MMCM and PLL).[4]

My advisor here in Hawaii, Kurtis Nishimura, suggested using the MMCME2_ADV which he was familiar with and previously used on several projects. The solution proposed is easily implementable with only minor changes, the configuration parameters where taken from the previous IP core and re-entered in the primitive. The reference clock (FCLK_CLK0) for this modules is set up in the Zynq IP core. This clock of 100MHz served as reference to the primitive from which was derived only two frequencies, a 100 MHz and 250 MHz clock. The reason for this limitation is to accelerate development time by avoiding, as much as possible, clock domain crossing and metastabilities in the design. Even if the design is based on two clocks generated from the same clock, the AXI-Lite and Stream are generated from another source clock. Thus clock crossing is inevitable. The two output clocks are raw clocks and must be sent through a buffer (BUFG) before being used in the different processes.
The different clocks are then using the 100MHz or 250MHz to generate the final version of the needed clocks. Some of those clocks required a special LVDS buffer (OBUFDS).

3.2 Control PL and TARGETC

The programmable logic is controlled through the AXI-Lite interface, which is accessed like an array from the PS side. Vivado's IP for AXI interfaces was used and modified to fit it to the project's needs.

3.2.1 Control Interface - Reg:129 TC_CONTROL_REG

The register TC_CONTROL_REG is the interface controlling all the operations of the ASIC in terms of starting a window sampling and digitization process, testing different features of the TARGETC, writing to an internal register of TARGETC and test functions.

Bit Name	Bit Nbr	Mask
C_WRITE_MASK	0	00000001_{16}
C_REGCLR_MASK	5	00000020_{16}
C_SS_TPG_MASK	7	00000080_{16}
C_WINDOW_MASK	10	00000400_{16}
C_SWRESET_MASK	12	00001000_{16}
C_SMODE_MASK	13	00002000_{16}
C_TESTSTREAM_MASK	14	00004000_{16}
C_TESTFIFO_MASK	15	00008000_{16}
C_PS_BUSY_MASK	16	00010000_{16}
C_CPUMODE_MASK	17	00020000_{16}

Table 3.1: TC_CONTROL_REG Bits/Mask

C_WRITE_MASK : This register is built using the code in 3.2. Setting this bit to HIGH will automatically generate a pulse, which will initiate the appropriate register write sequence, seen in 2.10. Double setting this bit HIGH will have no effect. This particular bit must be reset (LOW) before any other write command can be performed.

C_REGCLR_MASK : setting this bit to HIGH, will reset all registers from TARGETC, including the parameters and samples. The TARGETC is back to initialization stage.

C_SS_TPG_MASK : setting this bit to LOW will enable the Test Pattern generator function on the ASIC and output the value held in the TPG register inside the ASIC, like shown in the figure 3.30. Setting it to HIGH will readout normal sampled and stored value.

C_WINDOW_MASK : this bit enables the user to ask for any window and number of window by first setting the following registers:

- TC_FSTWINDOW_REG : a window number between 0 and 511, which will be the first window
- TC_NBRWINDOW_REG : the number of consecutive windows to read. MAXIMUM is 15 windows, limited by the asynchronous FIFO length. The length 15 is only for debugging and calibrating purpose, in the event of a trigger, the pulse should not take longer than 2 to 4 windows.

C_SWRESET_MASK : performs a software reset to the PL side of the Zynq. Every component inside the FPGA will be set back to initial state and outputs will go back to their initial values. (SWRESET = 0, reset is active)

C_SMODE_MASK : Development bit which switches between AXI-DMA interrupt or normal line interrupt. Previously employed when the AXI-DMA was not integrated into the design. A interruption is raised from the PL side indicating a new sample, the CPU has to acknowledge it and readout the data from the AXI-Lite registers (see code 3.1).

Code 3.1: AXI-Lite sample readout using SMODE

```
1  constant TC_eDO_CH0_REG:    integer := 133;
2  constant TC_eDO_CH1_REG:    integer := 134;
3  constant TC_eDO_CH2_REG:    integer := 135;
4  constant TC_eDO_CH3_REG:    integer := 136;
5
6  constant TC_eDO_CH4_REG:    integer := 137;
7  constant TC_eDO_CH5_REG:    integer := 138;
8  constant TC_eDO_CH6_REG:    integer := 139;
9  constant TC_eDO_CH7_REG:    integer := 140;
10
11 constant TC_eDO_CH8_REG:    integer := 141;
12 constant TC_eDO_CH9_REG:    integer := 142;
13 constant TC_eDO_CH10_REG:   integer := 143;
14 constant TC_eDO_CH11_REG:   integer := 144;
15
16 constant TC_eDO_CH12_REG:   integer := 145;
17 constant TC_eDO_CH13_REG:   integer := 146;
18 constant TC_eDO_CH14_REG:   integer := 147;
19 constant TC_eDO_CH15_REG:   integer := 148;
```

C_TESTSTREAM_MASK : Development bit used to self-test the AXI-Stream component. A pulse is sent to the component, which triggers a dummy write sequence to the AXI-DMA (see figure 3.36).

C_TESTFIFO_MASK : Development bit used to self-test the FIFO Manager component. A pulse is sent to the component, which triggers a complete dummy sample readout sequence, filling up the FIFOs with the dummy data. When all FIFOs are full, a window transfer is initiated through the AXI-Stream component.(see figure 3.35).

C_PS_BUSY_MASK : The processing system needs to feed the AXI-DMA a new memory location for the potential coming transfers, in order for the PL to continuously send data to memory. This bit is the way for the PS side to retain the PL from sending any more packets, before it can allocate new memory space. Upon an interruption for the AXI-DMA, this bit must be set HIGH to acknowledge the reception of a new packet, the ARM then disables this bit when the AXI-DMA has been configured with a new memory location.

C_CPUMODE_MASK : The Round buffer can work in two different ways, the normal and trigger mode. In the normal, it waits for a user command (C_WINDOW_MASK) to readout any number of windows. In the trigger mode, the user has no control, the trigger inputs dictate which window must be digitized.

Code 3.2: VHDL Process for pulse signal generation

```
process(nRST, CLK) is
begin
  if (nRST = '0') then
    statemachine_s <= IDLE;
  else
    if rising_edge(CLK) then
      case statemachine_s is
        when IDLE =>
          if ((TCReg(TC_CONTROL_REG) and C_SIGNAL_MASK) = C_SIGNAL_MASK) then
            statemachine_s <= PULSE;
          else
            statemachine_s <= IDLE;
          end if;
        when PULSE =>
          statemachine_s <= RESET;
        when RESET =>   -- Wait for user PS clear register
          if ((TCReg(TC_CONTROL_REG) and C_SIGNAL_MASK) = C_SIGNAL_MASK)then
            statemachine_s <= RESET;
          else
            statemachine_s <= IDLE;
          end if;
      end case;
    end if;
  end if;
end process;

signal_s    <= '1' when statemachine_s = PULSE else '0';
```

3.2.2 Status Interface - Reg:130 TC_STATUS_REG

The status register is a read-only register which the PL uses to send out information status of the PL side. The following table resumes the different information found in this register.

Bit Name	Bit Nbr	Mask
C_BUSY_MASK	0	00000001_{16}
C_LOCKED_MASK	1	00000002_{16}
C_STORAGE_MASK	2	00000004_{16}
C_SSVALID_MASK	3	00000008_{16}
C_WINDOWBUSY_MASK	4	00000010_{16}
C_FIFOBUSY_MASK	5	00000020_{16}

Table 3.2: TC_STATUS_REG Bits/Mask

C_BUSY_MASK : Indicates if the PL is finished writing the register to the TARGETC.

C_LOCKED_MASK: Monitors the lock signal from the MMCM used for the PL clock generation. The Control Unit is running on the AXI-CLK and is not touched by the MMCM restarting. The software must check if the MMCM is locked before continuing.

C_STORAGE_MASK: Direct write control for the bit 0 of the C_CONTROL_REG. The CPU can check whether the PL has received the command or not.

C_SSVALID_MASK: Readout of the sample valid signal.

C_WINDOWBUSY_MASK: After a window is stored and a trigger event or user command has evaluated this specific window is of importance. The three next operation are ran sequentially.

1. Readout Window
2. Wilkinson Digitization
3. 32 Samples Readout

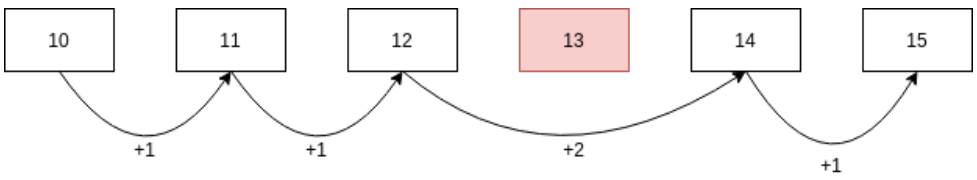
This bit indicate the busy state during these three phases, like shown in the code below.

Code 3.3: C_WINDOWBUSY_BIT

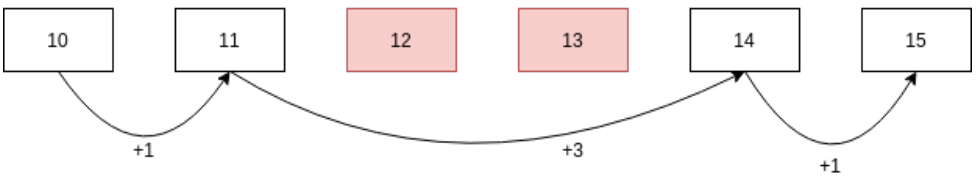
```
1 WindowBusy <=    '1' when RDAD.busy = '1' else
2                          '1' when WL.busy = '1' else
3                          '1' when SS.busy = '1' else
4                          '0';
```

3.3 Round Buffer

The Round Buffer is the most complicated and complex hardware design of the system. The previous TARGET designs were incapable of selecting the next storage location. The data was stored round and round in the same memory places. When a window should not be overwritten, the user deactivated the write enable line and therefore the window was untouched, this also meant that the 64 ns sampled in the slice of time was lost.
The TARGETC does not include this system, the storage address is managed externally. This feature is a great advantage compared to previous designs, because the next address can always be a free storage area, meaning the possibility of no dead time. When a window has data of interest, this address should be avoided till the window is digitized. This is illustrated in figure 4.3.



(a) Next by skip one



(b) Next by skip two

Figure 3.1: Next address problematic

The idea is to know in advance the next address, by adding an offset (1 + skip) or by an intelligent system advising the system of the next address. Different architecture can be derived from there, but all have their pros and cons.

3.3.1 Option and System evaluation

3.3.1.1 Small Round Buffer

The idea is to have some smaller static round buffer (SRB) with fixed length, for example 8 storage cells, so 64 small round buffer in total. At the beginning, only one round buffer is active, the system parses through the 8 addresses. A while later an event occurs the round buffer saves the addresses until the event is finished and moves to the next round buffer. The round buffer which encountered the hit, is dealt off-line, while a second buffer is running, having no dead time.

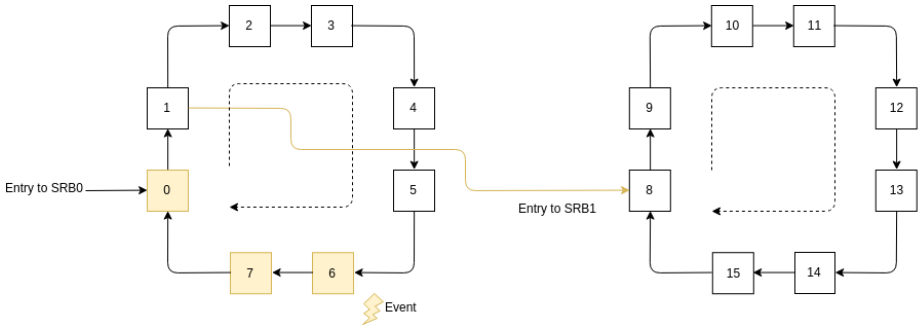


Figure 3.2: Small Round Buffer principle

Of course, the first problem that could be encountered is if the event is longer than the buffer size. A simple solution is to have the smaller round buffer size increase from 8 to 16 or 32 cells. This technique has however a weak spot, this starts by looking at the physical line of the trigger signal, refer to figure 2.9. The PMT signal goes through the amplifier stage, the trigger signal is taken from the amplifier x10 line and passed through a comparator and sent to the FPGA, while all the amplifier line are connected to the ASIC. The comparator introduces a delay on the trigger signal line compared to the data collected.

The worst scenario is having a trigger on PMTA which will start storing data in the SRB0. At the end of this trigger on PMTA, the system switches on SRB1, at that exact moment an event is spotted by trigB on PMTB. Like the trigger line has a long delay, the actual corresponding window is for the previous window 1 and not the current window 8.

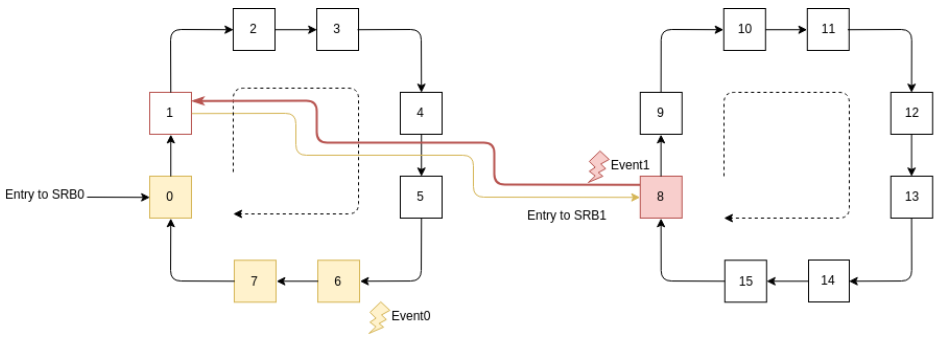


Figure 3.3: Small Round Buffer principle

3.3.1.2 On the fly address calculator

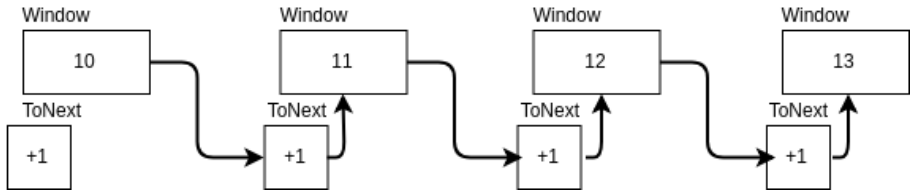
The first system implemented was a single component evaluating the next address on the fly. This is to say it ran a while loop until it found an available window, which meant it found the next address. This method is only working for proof of concept and cannot be implemented in synthesizable VHDL. Reminder, the windows are sampling by pairs, this is why the wr1 enable is for the first window (EVEN) and wr2 enable respectively the second (ODD). For one to be disabled, the two windows are disabled. The code 3.5 bellow shows a perfectly working code for simulating this next address behavior and this is exactly what must be implemented, but again this is a NON-synthesizable code.

Code 3.4: While loop evaluation of next free window in non-synthesizable code

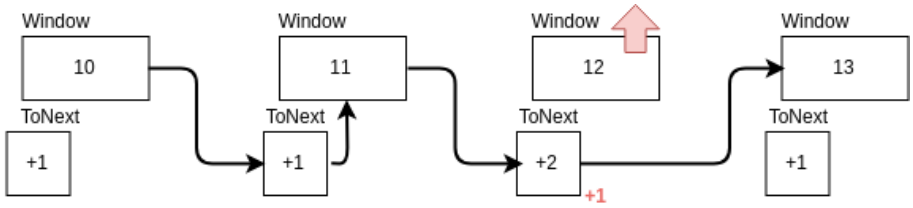
```
1 while(((Window64(Address).wr1_en = '0') or (Window64(Address).wr2_en = '0'))loop
2   if Address = 255 then
3     Address := 0;
4   else
5     Address := Address + 1;
6   end if;
7 end loop;
```

3.3.1.3 Jump Over

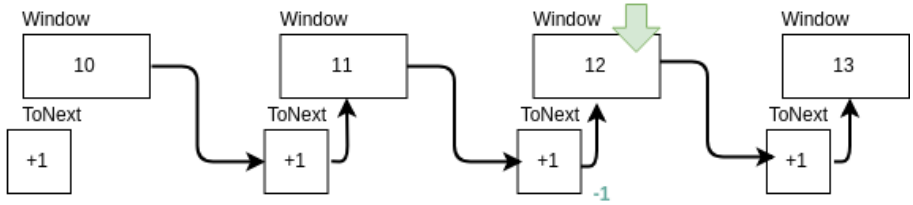
The previous method showed that the next address cannot be a multiple iterations search (while loop). The following assumption is the base of this research around a jump system in which every element must know its address, the previous and next one. The figure **??** shows the different steps of this process using a register indicating the amount to add to the current address window for having the next one. The system works fine for a single address being digitized, but if an event extends to two windows the system cannot deal with it. It should save all the address related to the event and add the corresponding amount of jumps to reach the next address.



(a) Normal jump sequence



(b) Element 12 is removed



(c) Element 12 is re-inserted

Figure 3.4: Jump over sequence for single element removal/insertion

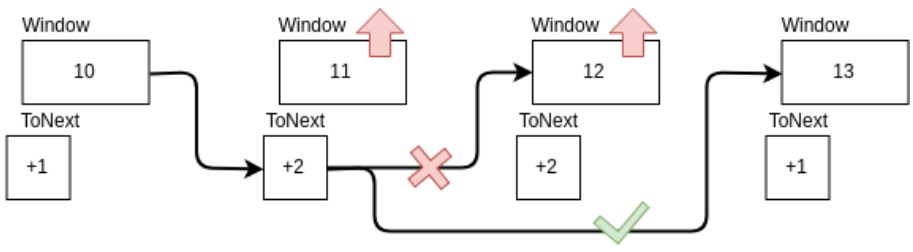


Figure 3.5: Jump over with multiple windows removed

3.3.1.4 Linked List

In C code a linked list is schemed by an element pointing to another and this element point to another and so on. Each of these elements knows the next and previous address. Removing an element from such a list implies that the next and previous elements of this element shall be updated in order to keep the list linked.

$$(element.prev).next = element.next$$
$$(element.next).prev = element.prev$$

At the re-insertion of an element in the linked list, this element needs to take back the previous and next address in the list, but the only knowledge the re-inserted element has is his return location in the list, which means his own previous and next addresses.

$$element.next = (element.prev).next$$
$$((element.prev).next).prev = element$$
$$(element.prev).next = element$$

In the case of the round buffer the element should be reinserted after digitization and the same problem occurs as with the jump over principle. The problem is illustrated in figure 3.6.

Subfigure 3.6a shows the initial setup of the elements in the list.

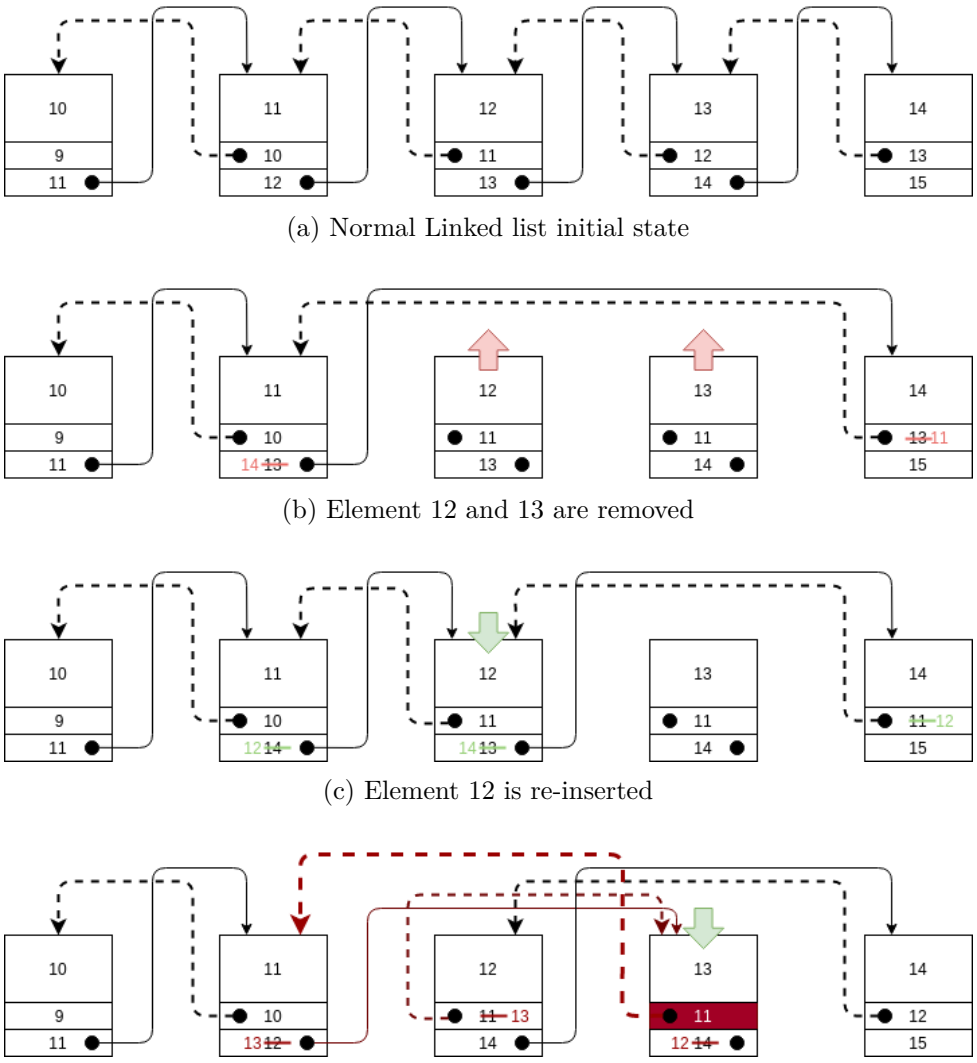
Subfigure 3.6b illustrate the removal of element 12, according to the equation:

1. Element 11's next becomes 13.
2. Element 13's prev becomes 11.
3. Now element 13 is removed from the list and so element 11's next becomes 14.
4. Element 14's prev becomes 11.

Subfigure 3.6c shows the insertion of element 12, following the equations.

1. Element 12's next becomes 14 (12's previous = 11, so 11's next = 14)
2. Element 14's previous becomes 12.
3. finally element 11's next becomes 12. The linked list is still working.

Subfigure 3.6d shows the crash part. Like element 13 has only his previous and next address to trust to re-enter the list. It will automatically take the last previous element which is 11 but element 12 was added before, which will result in an erroneous linked list.



(a) Normal Linked list initial state

(b) Element 12 and 13 are removed

(c) Element 12 is re-inserted

(d) Element 13 is re-inserted

Figure 3.6: Linked list failure sequence

3.3.1.5 Next and Previous Bus

The linked list design shows that having a single register containing the information, is not sufficient, because of the re-insertion of elements can lead to hazardous situations. Although the element is inactive in the round buffer, it should still be acknowledging the changes occurring in the round buffer. This is to say the change of previous and next addresses should be seen by the element. The principle implemented is transferring the information, if an element is unavailable.

Like explained earlier in this document, the sampling circuit samples over 64 ns, this means that every sampling process 2 windows of 32 samples are taken, therefore only 256 storage addresses are possible. Each of these address must keep track of window overwrite, this can be seen as write enable line, Write enable 1 for the first window and respectively write enable 2 for the second window. The window can be overwritten if the logical condition (wr1_en $\cap$ wr2_en) is respected.

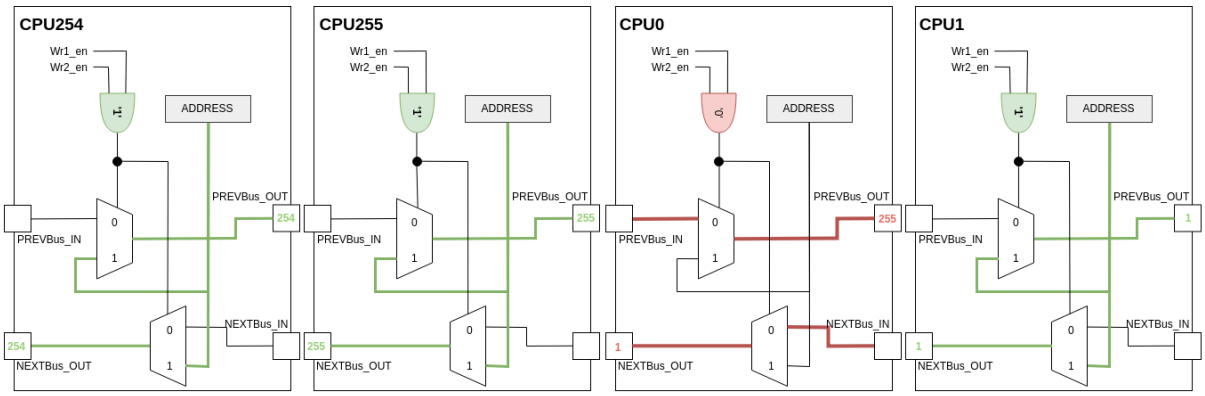


Figure 3.7: Next and previous bus using logic

In figure 3.7, CPU0 is of interest so the wr1_en or wr2_en is disabled causing the logic to switch from displaying his address to just pass an address. CPU1 knows the previous address from CPU255 because CPU0 is inactive but passes the information. This solution removes the address 0 of the bus and at the same time keeps the CPU0 active. When the CPU0 is enable again, the multiplexers re-wire themselves and CPU0 is back online. The logic behind this figure is very transparent and efficient.

Code 3.5: While loop evaluation of next free window

```
1 PREVBus_Out <= std_logic_vector(to_unsigned(ADDRESS,PREVBus_out'length)) when ((wr1_en↩
      ='1') and (wr2_en='1')) else PREVBus_In;
2 NEXTBus_Out <= std_logic_vector(to_unsigned(ADDRESS,NEXTBus_out'length)) when ((wr1_en↩
      ='1') and (wr2_en='1')) else NEXTBus_In;
```

3.3.2 Overall System

The system is composed of 256 CPUs each controlling the status of two windows of 32 samples. The control unit interacts with the CPUs to tell which window is of interest using a command bus.

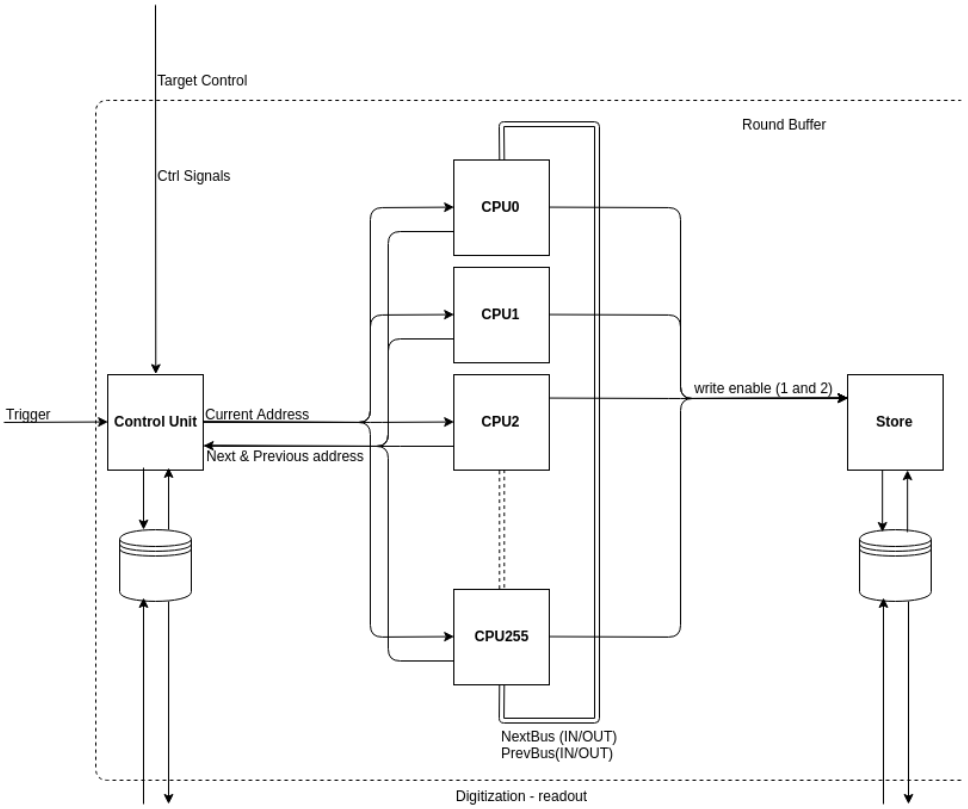


Figure 3.8: Overall principle schematic for the Round buffer

3.3.2.1 CPUs

Two ways are possible to have the CPU removed from the chain. The first is for the control unit to specify to the CPU, one or two of his windows are of importance using the command bus. The second option is trigger mode. Each CPU is responsible of knowing the trigger information and react accordingly. Four triggers are possible like the board is designed.

- **TrigA**: Channel 0 to channel 3
- **TrigB**: Channel 4 to channel 7
- **TrigC**: Channel 8 to channel 11
- **TrigD**: Channel 12 to channel 15

A special bus is dedicated for each trigger, because each of them is independent to each other. In other words, for the same window it is possible the triggers react differently or are delayed and so even if the last window for trigA is noticed, maybe trigC is still active and sampling should continue.

Code 3.6: Entity for the CPU

```
component WindowCPUV2 is
    generic(
        ADDRESS : integer := 0
    );
    Port (
    nrst :      in  std_Logic;
    CLK:            in  std_logic;

    --Window Part
    BusA :          in t_CommandBus;  -- Command Bus for trigger A and normal op.
    BusB :          in t_CommandBus;  -- Command Bus for trigger B
    BusC :          in t_CommandBus;  -- Command Bus for trigger C
    BusD :          in t_CommandBus;  -- Command Bus for trigger D

  --Trigger Information IN/OUT
    TrigInfo_IN:  in  t_triggerinformation;  -- Trigger information from Control
    TrigInfo_OUT:   out std_logic_vector(11 downto 0);

    wr1_en :        out std_logic;
    wr2_en :        out std_logic;
    valid_o :       out std_logic;

    CurAddr:          in    std_logic_vector(7 downto 0);  -- Current Address for ↩
        storage
    PREVBus_IN :    in  std_logic_vector(7 downto 0);
    PREVBus_OUT :   out std_logic_vector(7 downto 0);
    NEXTBus_IN :    in  std_logic_vector(7 downto 0);
    NEXTBus_OUT :   out std_logic_vector(7 downto 0));
end component WindowCPUV2;
```

Each CPU is aware of the current address and react depending on the previous and next address (PREVBus_IN and NEXTBus_IN) to update his outputs (PREVBus_OUT and NEXTBus_OUT). Also it must be aware of these elements at re-entering the chain. The re-enter can only happen if the current address has nothing in common with the CPU's address. As a result the re-enter condition is the CPU should neither be the next nor current nor previous address, when these three conditions are meet, the CPU can re-enter the round buffer without any problems.

3.3.2.2 Control Unit

The control system is responsible of identifying a trigger event (TrigA to TrigD) and interpreting a user command sent form the PS side (FSTWINDOW and NBRWINDOW).

3.3.2.2.1 User Command

The round buffer is continuously updating windows, therefore the storage address is changing. The control unit has to wait until the current address corresponds to the FSTWINDOW parameter. From there, the control unit has to evaluate which command to send out. The two parameters (FSTWINDOW and NBRWINDOW) are specified for 512 windows. Although the round buffer is on 256 because each storage cell has 2 windows. After evaluating the correct window to store, it starts counting and sending the appropriate command to the CPUS until the counter reaches the value of NBRWINDOW at this point the control unit is finished.

3.3.2.2.2 Trigger Event

The trigger is a very special case of event, because a lot of parameters need to be taken into account. At first lets look at figure 3.9, the trigger signal concerns only the window 0. Once the threshold value is reached the comparator will set its output to HIGH. The output of the comparator is routed to an input pin of the FPGA. Internally a fast clock (250MHz) samples the line. The resulting signal will be delayed by a few nano-seconds depending on the speed of

the comparator (Delay Comparator) and the sampling clock (Delay Sampling). An overall delay on the trigger line is an inevitable consequence of external circuit and input sampling.

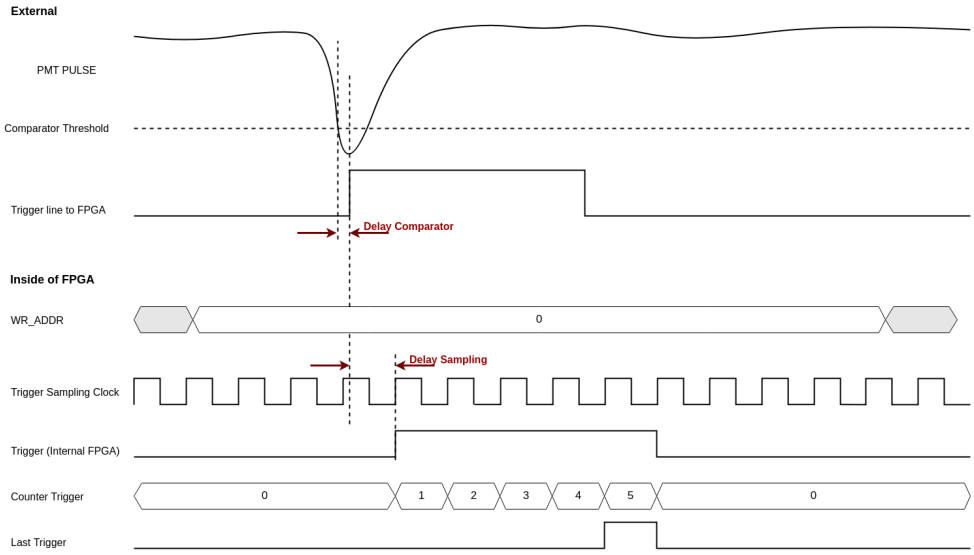


Figure 3.9: Sampling of the external trigger to internal trigger signal

The trigger information collected from the pulse are :

- TRIG: the sampled trigger of a certain group
- LAST: last clock cycle before end of trigger event
- LONG: counter trigger is incremented during the period of time the trigger is active, when this signal stays longer on than a maximum value the signal passes high, indicating a long trigger.

All these information are useful for the decision making of the window selection and CPUs control. They also serve another purpose in the PS side, they are relevant for the analysis and data processing which are performed on the waveforms, as an example of the feature extraction [1].

3.3.2.2.3 Asynchronous FIFO - Digitization Back

All window are treated (Readout-Wilkinson Digitization- Sample Readout) outside the round buffer. Once the window is dealt with, the Wilkinson process stores it into an asynchronous FIFO. (First-In-First-Out). The control unit is responsible to read it, retrieve the CPU's address and send the appropriate command on BusA. The CPU, like explained earlier can only re-enter the chain under specific conditions.

3.3.2.3 Store

Most importantly, when a window has been ordered to remove itself from the round buffer, this particular window changes its write enable signals. The store unit supervises the write enables of all CPUs. The change on the lines automatically initiates a write procedure to an asynchronous FIFO with the readout process. The information written to the FIFO are the time at which the storage cell (2 window) was sampled, the CPU's address, the write enable signals and a spare

[1] develop and explained in the Master thesis of Anthony Schluchin

32 bits register for development and debugging purpose. The store unit decodes the window address (9-Bit) from the CPU address (8-bit) and the write signals. The system is such that the readout and digitization processes know if they have to handle one window or two windows.

3.3.3 Development

This section only covers a small portion of a previous code that used the idea of the next and previous bus explained earlier.

3.3.3.1 Round Buffer

The system under test is a round buffer of length 16, which means that 16 CPUs are linked together. The testbench used in this simulation is a round buffer of 16 CPUs linked together. The window 4 out of the 32 has valuable information. The system removes CPU2 from the chain. Refer to figure 3.10, the command is sent on BusA (Addr = 2_{16}, command = CMD_WR1_MARKED) followed by an enable pulse on the bus. The write 1 enable lines from all CPUs have changed, line(2) is now at 0 indicating the window 4 is unavailable.

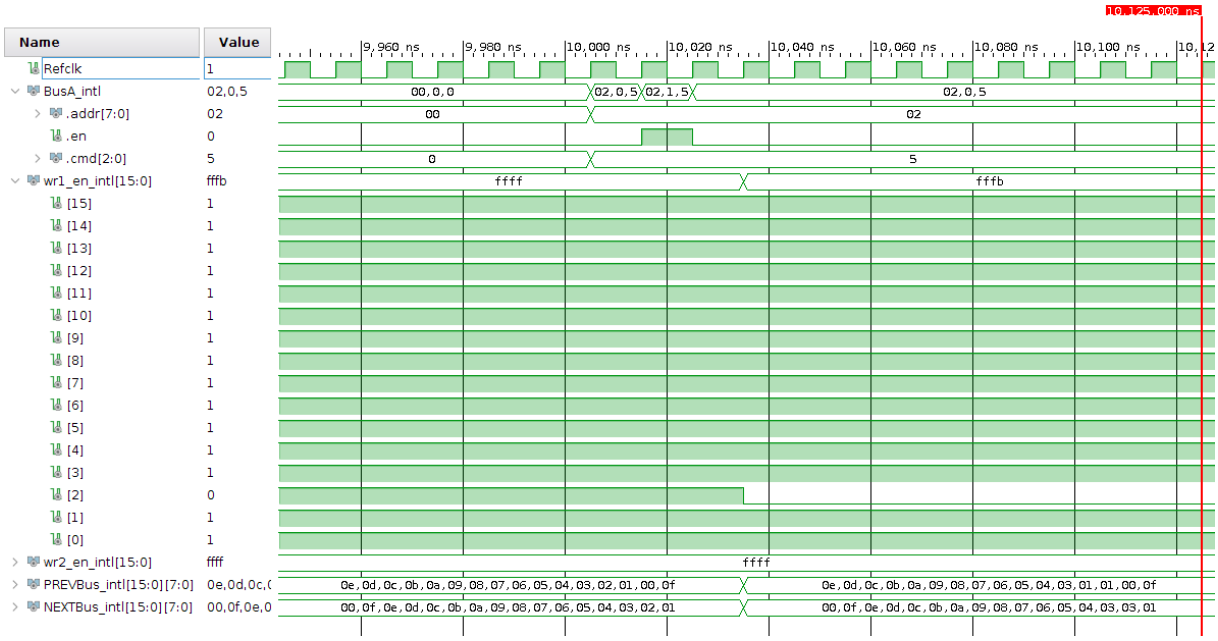


Figure 3.10: Simulation : Window 4 out of 512, write enable

Figure 3.11 shows the same simulation for the signals PREVBus and NEXTBus (between the CPUs). For instance on PREVBus(3), the signal has changed from 2_{16} to 1_{16}, meaning the previous address for CPU3 is indeed CPU1, like CPU2 is removed. The same applies for the NEXTBus(1) which indicates the next CPU, in this case the value changed from 2_{16} to 3_{16}.

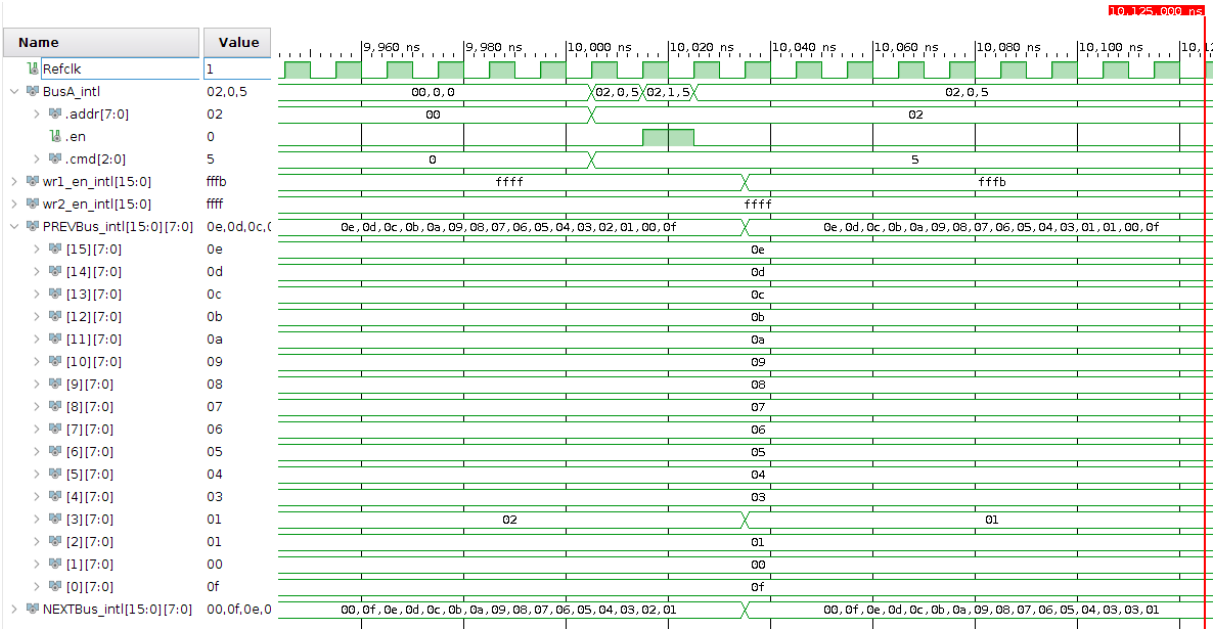


Figure 3.11: Simulation : Window 4 out of 512, next and previous bus

Window 5 and 6 are removed from the chain, this means CPU2 and CPU3 are removed (one after the other). The simulation shows after the first removal of CPU2 the write enable 2 lines change, the NEXTBus and PREVBus are updated. The new previous address for CPU3 is CPU1. When CPU3 is removed, CPU4 sees the CPU1 as the previous CPU.

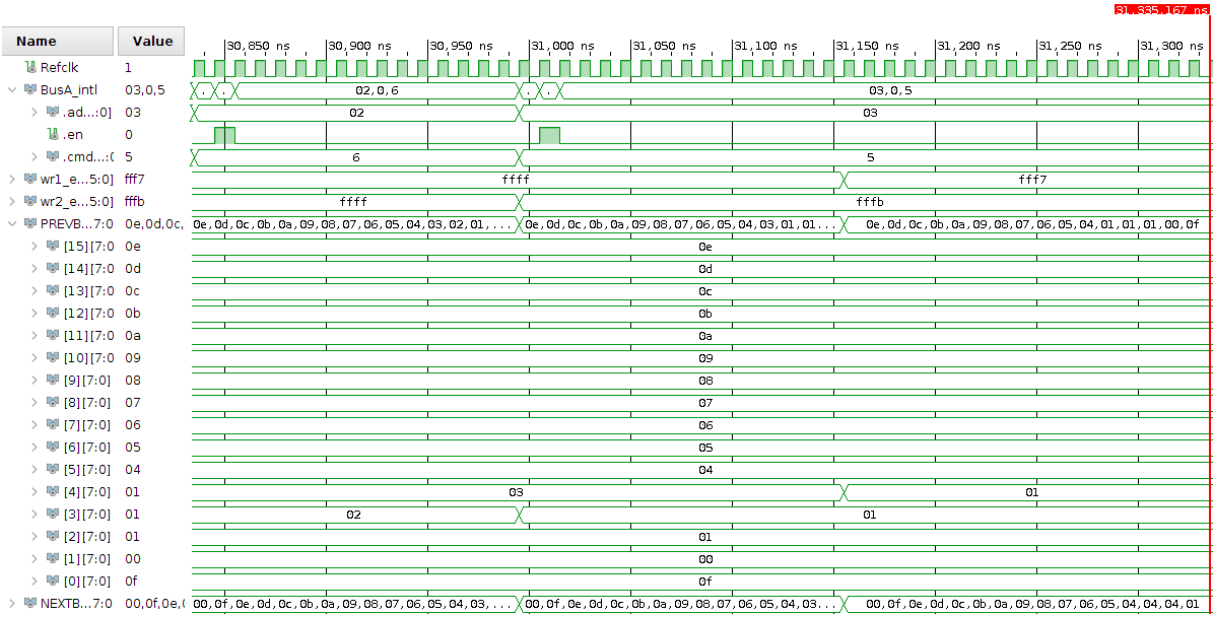


Figure 3.12: Simulation : Window 5 and 6 are out of 512

3.3.3.2 Asynchronous FIFO

In order to verify the functionality of the asynchronous FIFO and how it reacts to the input transactions, a brief test bench was performed. Two separate processes are running (Write and Read). The write process runs at 250MHz and the read at 100MHz. The write fills in the

memory and the read process reads it back and checks if the readout data is coherent with the written value. The signals full and empty are well updated, so a write on a full FIFO will not occur, like a read on an empty FIFO. The simulation report (below) shows that all writes and reads occurred correctly. In addition to the correctness of the read and write operations, the simulation waveform also shows that the data is shifted out on each rising edge.

```
aFIFO TestBench Simulation End
Success : 256/256 = 100 %
Error : 0/256 = 0 %
```

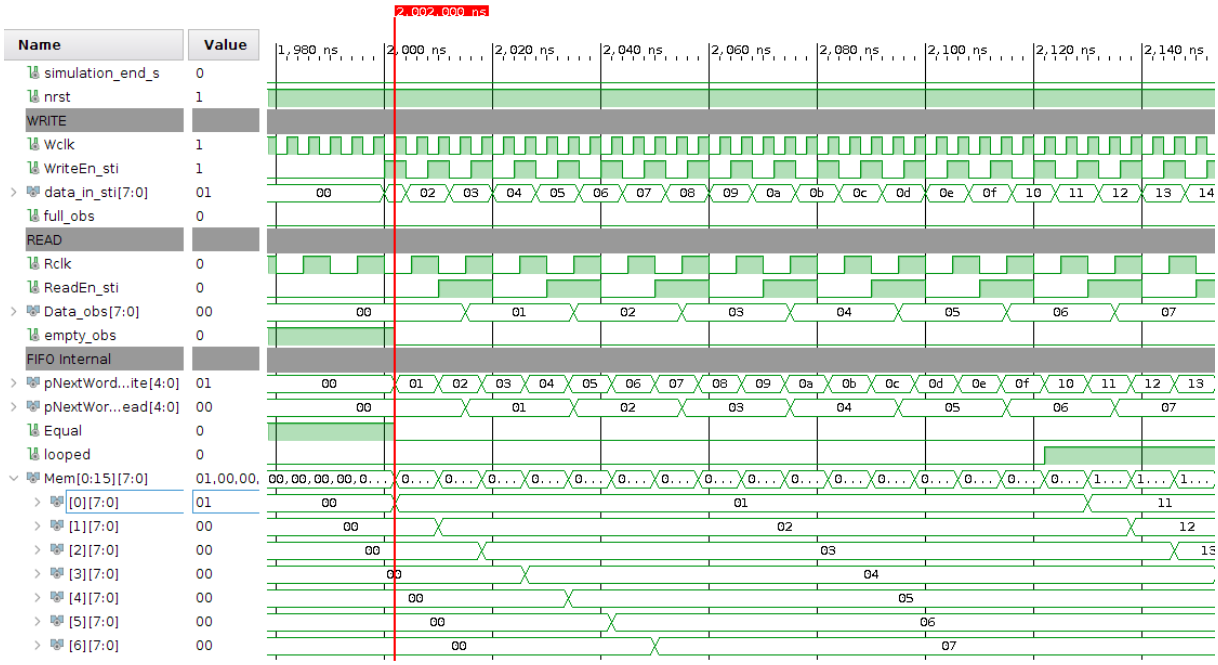


Figure 3.13: Simulation : Asynchronous FIFO

3.3.4 Intermediate results

After many simulations which were not mentioned in the section to avoid overwhelming the content of this report, they demonstrated a round buffer capable of offering good performance. The synthesize was performed, even though the result showed that the idea of using the linked bus between CPUs is a success. The current FPGA could not handle it, the resources are insufficient. Indeed the FPGA Zynq Z7010 on the MicroZed is one of the smallest of the Zynq family with 17600 LUTs. The system used over 28000 LUTs, over-utilization of 64%. The design had to be modified to fit in the actual FPGA. Furthermore the later perspective are to use a Microzed for two TARGETC. Hence a higher performance FPGA is probably necessary on the long run. The Microzed comes equipped with the Zynq Z7020, the board is actually 180 USD and equipped with the Z7020, the board is 220 USD.

3.4 Interface to TARGETC

The TARGETC has 5 different communications types, like seen earlier. Each of these communication have a specific sequence, therefore each communication has its own process or VHDL module. As a consequence each of these modules could be separately tested and verified without affecting the others. Nevertheless at some point in time they must be able to synchronize at some point. They must communicate information regarding when to start or wait for the previous to finish and so on. This transactional protocol use the fact that both parties have to acknowledge one another to continue. The handshake technique is a bidirectional synchronization process used in this firmware.

3.4.1 Handshake - V1.0

The handshake is a way to synchronize two processes or systems. Their is always a sender and a receiver, the sender waits on the receiver, it checks the busy signal. Once the receiver is available, the sender can initiate the handshake transaction illustrated in figure **??**.

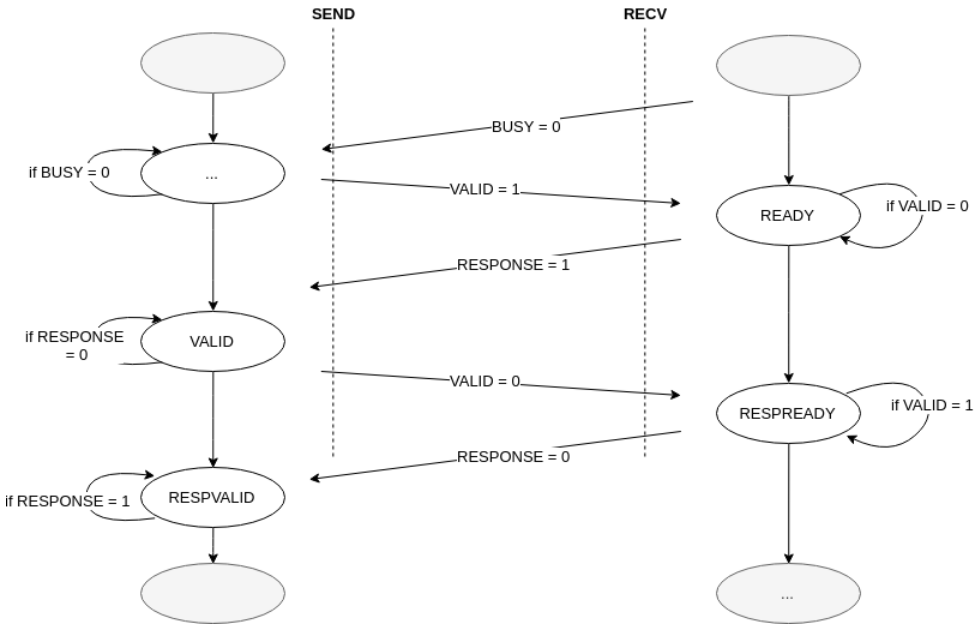


Figure 3.14: Handshake V1.0

3.4.2 Register Process

The write register interface has been unchanged for several TARGET versions now. The firmware for this function is taken from a previous project using TARGETX. However some modification were added to run the requirements elaborated in chapter 2.3.1, these changes concerned the setup and HIGH time of the PCLK line. The simulation in Vivado shows a correct waveform, the write register sequence is very long and stretches over a few micro-second.

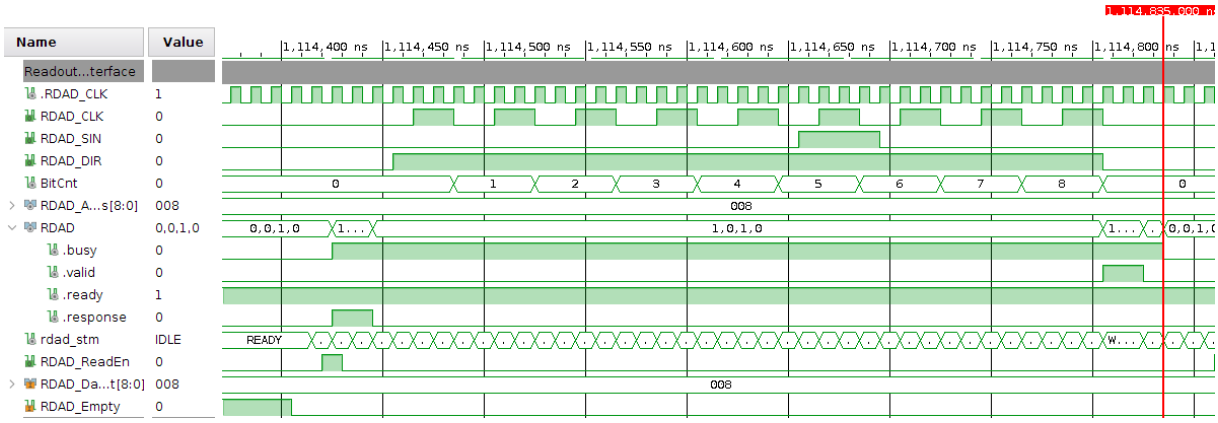


Figure 3.15: Sim : Write register interface

Verification on hardware was as expected, the PCLK pulse are 80 ns wide each (shorter than specified by the requirement), nevertheless the ASIC shows nominal behavior.. The write is successful. The measurement shows the write for the test pattern register number 128, when sent over the bus, this address is decremented by one. The real address is 1111111_2, the data transmitted is 12-Bit wide 00010010011_2 which is equal to the number 123_{16}

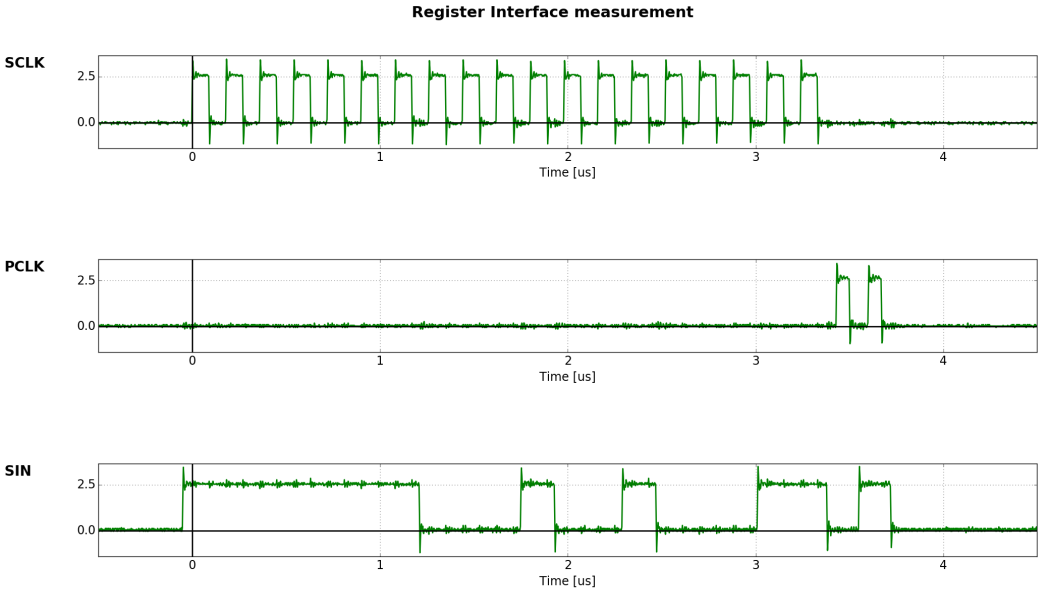


Figure 3.16: Write register interface measurement, for the TPG with the value 123_{16}

At first the TARGETC was considered identical to the TARGETX and so was the register map. Unfortunately, the ASICs schematics show that a different pattern was used to map out the internal PCLK signals to the registers.As a reminder the following table resumes the register map addressing for TARGETC.

Reg #	Reg Name	Description/Comment
1 .. 64	VdlyTune	Delay samples 1 - 64 Default to zero (max on)
65	SSToutFB	Timebase 8 bit time
66	SSPin LE	Timebase 8 bit time
67	SSPin TE	Timebase 8 bit time
68	WR_STRB2 LE	Timebase 8 bit time
69	WR_STRB2 TE	Timebase 8 bit time
70	WR_ADDR_Incr2 LE	Timebase 8 bit time
71	WR_ADDR_Incr2 TE	Timebase 8 bit time
72	WR_STRB1 LE	Timebase 8 bit time
73	WR_STRB1 TE	Timebase 8 bit time
74	WR_ADDR_Incr1 LE	Timebase 8 bit time
75	WR_ADDR_Incr1 TE	Timebase 8 bit time
76	MonTiming SEL	Timebase
77	Vqbuff	PLL
78	Qbias	PLL Vqbuff
79	VtrimT	PLL Vqbuff
80	Vbias	sampling Vqbuff
81	VAPbuff	Timebase
82	VadjP	Timebase VAPbuff
83	VANbuff	Timebase
84	VadjN	Timebase VANbuff
85	Sbbias	Vramp Dbbias
86	Vdisch	Vramp Dbbias
87	Isel	Vramp Dbbias
88	Dbbias	Vramp
89	CMPbias2	Wilk Sbbias
90	Pubias	Wilk Sbbias
91	CMPbias	Wilk Sbbias
92	Misc Digital Reg	5-bit == at right
93 .. 127	(Unused)	
128	TPGreg	12 bit pattern

Table 3.3: Register Map for TARGETC

3.4.3 Storage Process

The storage process represents the address change depending on the SSTIN clock, this process has changed many times, especially during development of the round buffer. The storage address is a 8-Bit wide bus decoded into 2 rows and 6 columns.

Code 3.7: TARGETC Storage pins attribution

```
1  -- Process Update
2  WR_RS_S <= StorageAddr(1 downto 0); -- Lower address
3  WR_CS_S <= StorageAddr(7 downto 2); -- Upper Address
4  -- ...
5
6  -- TARGETC PIN MAPPING/Update
7  WR_RS_S0    <= WR_RS_S(0); -- ROW Update
8  WR_RS_S1    <= WR_RS_S(1);
9
10 WR_CS_S0    <= WR_CS_S(0); -- COLUMN Update
```

```
11 WR_CS_S1    <= WR_CS_S(1);
12 WR_CS_S2    <= WR_CS_S(2);
13 WR_CS_S3    <= WR_CS_S(3);
14 WR_CS_S4    <= WR_CS_S(4);
15 WR_CS_S5    <= WR_CS_S(5);
```

The simulation of the storage address demonstrates the right signal sequence. The address 64_{16} is represented correctly on the WR_CS_S and WR_RS_S buses.

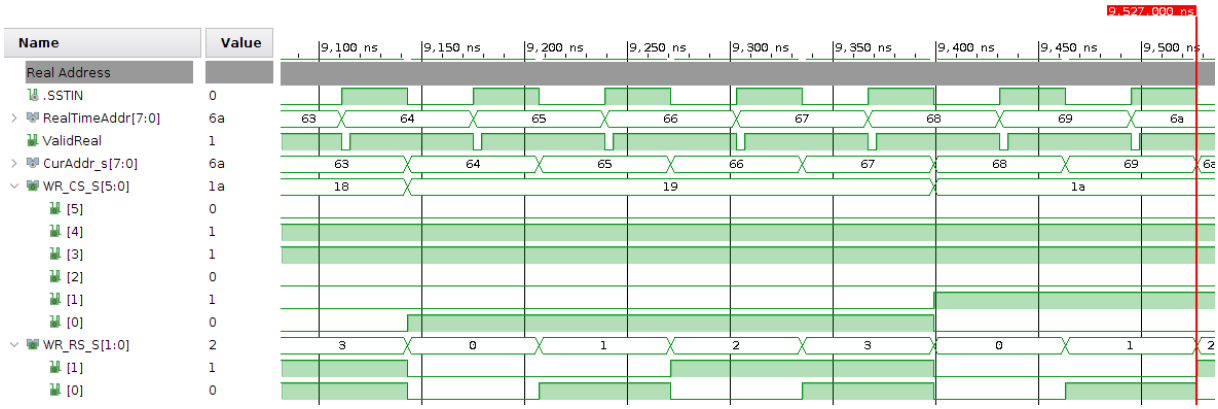


Figure 3.17: Storage Address Update Simulation

3.4.4 Readout Process

The readout processes interacts with the RDAD pins of the TARGETC. It runs on a state machine, the first state after IDLE is checking whether the FIFO from the round buffer contains any window to readout. If the FIFO is not empty, the window address is read and sent to the TARGETC using the serial communication for window readout. The full state machine is illustrated in figure 3.18.

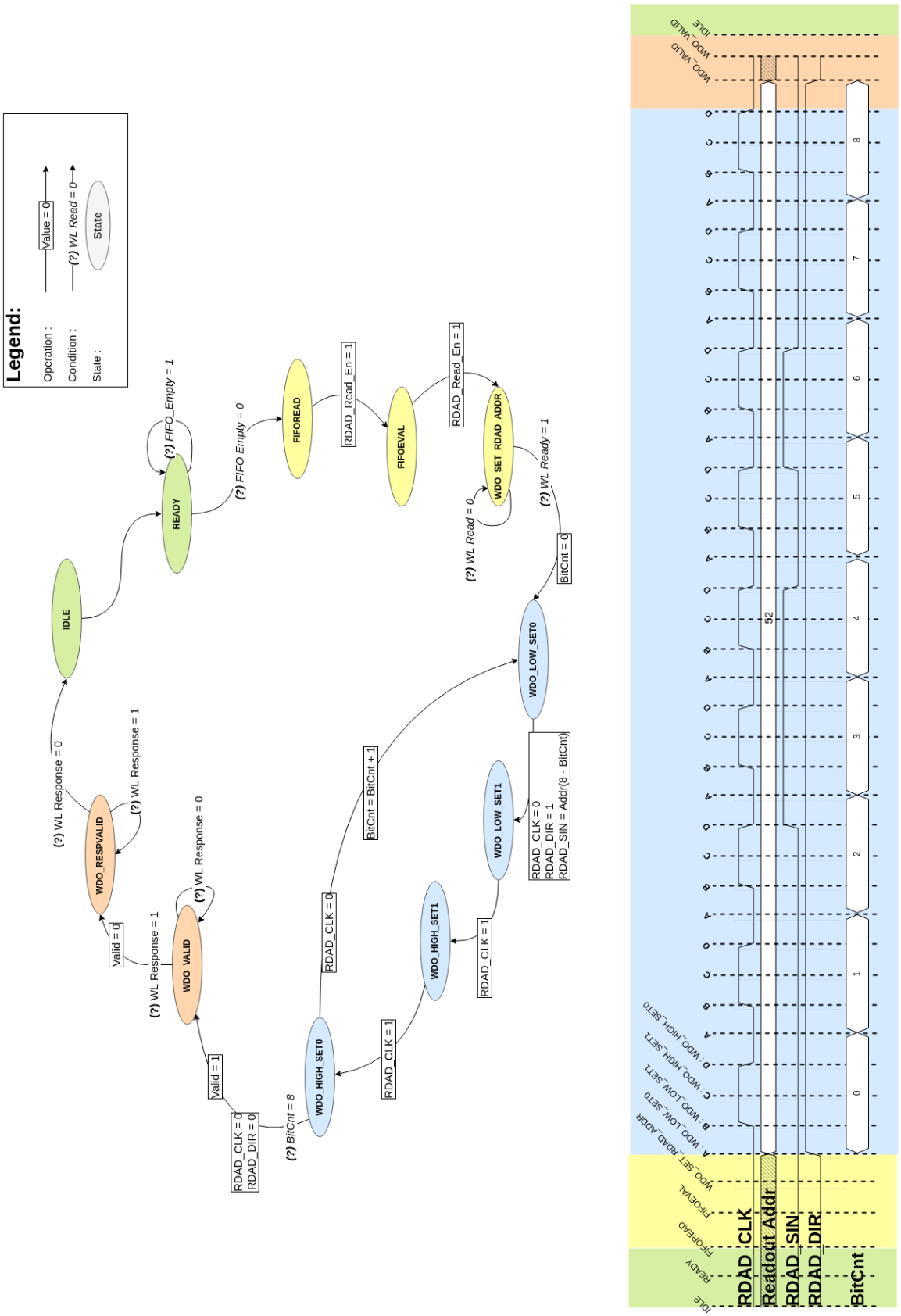


Figure 3.18: State Machine for the Readout process according to the specification

The state machine's implementation is rather straight forward, some improvement were added to the read sequence of the FIFO. It was simulated and tested that a read enable on the falling edge is more effective (timing constraints) in order to prepare a read instruction. The reference clock used by this process is the 100MHz clock from which RDAD_CLK is built.

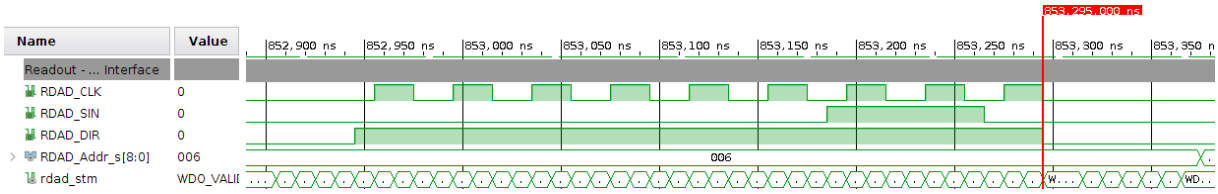


Figure 3.19: Sim : Readout interface sequence

The oscilloscope measurement shows the signals responding correctly to clock edges. The window asked is number 52_{10}. If the ASIC was to sample RDAD_SIN on every rising edge. It should decode, starting with the MSB (Left), 000110100_2, which is equal to $32 + 16 + 4 = 52$.

Figure 3.20: Readout interface pins measurement

Intermediate result and discussion

The process also implements the 1st handshake version between processes using a response and valid signal. In a near future, the goal is to replace this protocol by the 2nd version, (explained later in this chapter), which is much more efficient and robust. For the moment, the basic elements are in place for a same clock domain.

3.4.5 Wilkinson Process

Like for most of the processes in the firmware, this module has to wait on the readout process to finish to start the digitization (RDAD.Valid=1). This interface is the 1st version handshake, like previously explained. At first the Gray Code counter inside the ASIC is reset, by pulling down the GCC_Reset pin. Secondly, the ramp signal is enabled and the GCC_Reset is pulled HIGH,

the digitization process is underway. The Gray counter is 11-bit wide, therefore the counter's range is from 0_{10} to 2047_{10}. When the maximum value is reached, all the comparators should have been triggered, the digitization is finished. The information is transfered further in the chain to the sample readout processes using the Valid-response handshake. Once the samples have all been read, the Wilkinson process can discharge the capacitor and reset the Gray counter (eRamp = 0 and GCC_Reset = 1). Finally when the window has been dealt with, it can be transferred back to the round buffer system to enable it (DIG_WriteEn).

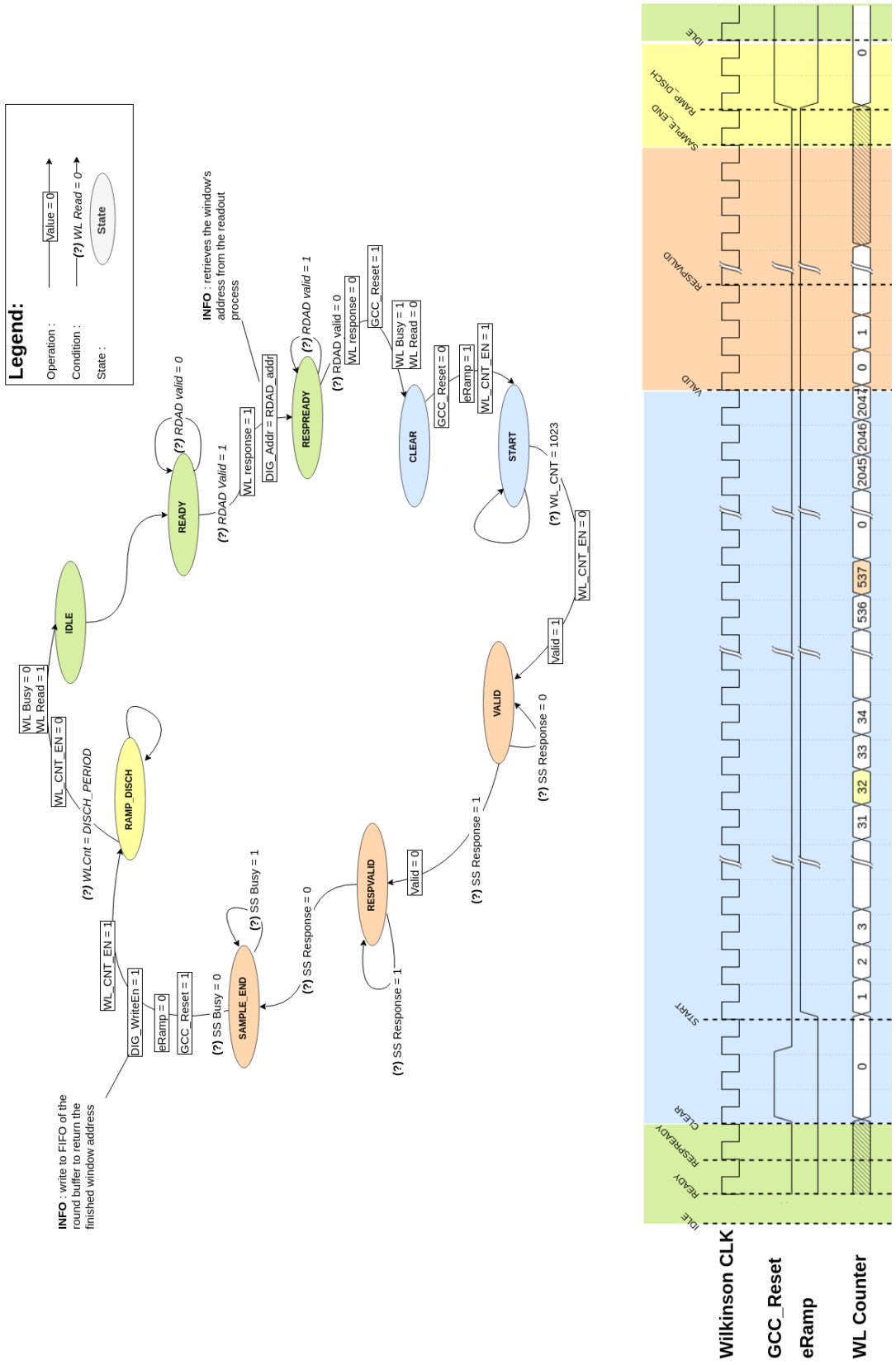


Figure 3.21: State Machine for the Wilkinson process according to the specification

The simulation shows the interaction between the different processes. Once the readout process (RDAD) is executed, it initiates a handshake transaction, which starts the digitization process. The Wilkinson counter is increment until it reaches $3FF_{16}$. A handshake transaction between WL and SS (sample select process) is established and the sample readout process is started like shown by the SS.valid signal and explained in the next section. The digitization process is temporarily put on hold for the time to read all 32 samples. Only after which it is resumed. The last operations to perform are to disable the ramp signal and to write back the window's address to the the round buffer, in this case address 0.

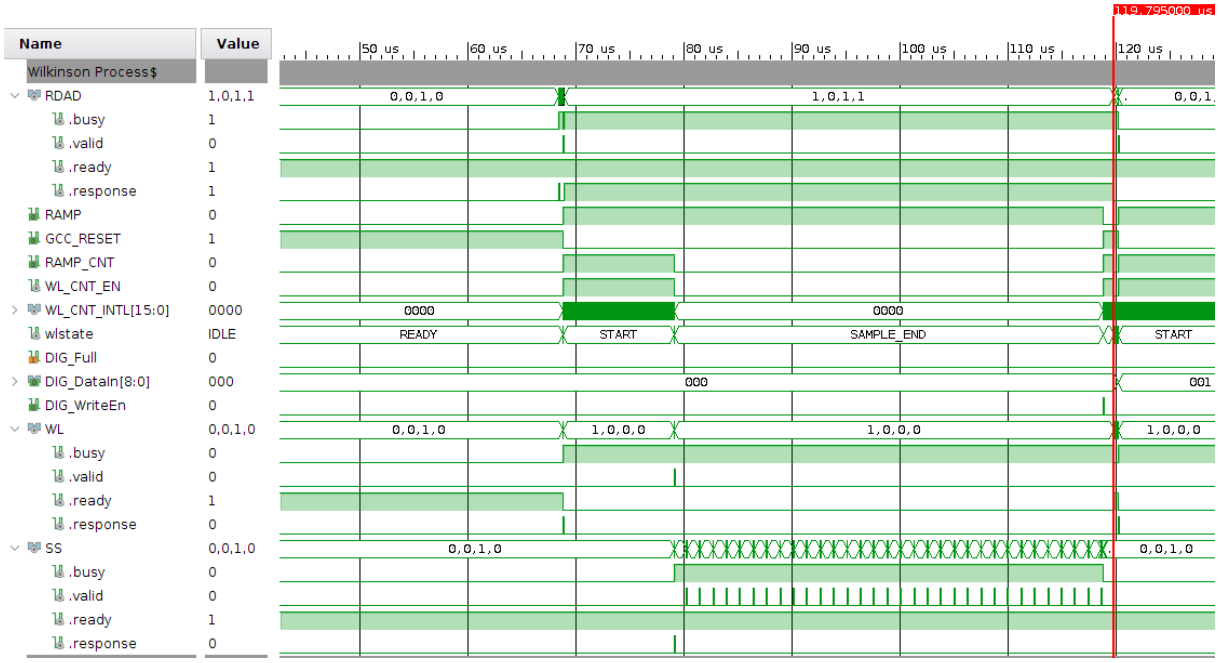


Figure 3.22: Sim : Wilkinson process with the interaction of window readout and sample readout

3.4.6 Sample Process

The Sample select process asks a good comprehension of the TARGETC internal architecture for generating the SS_INCR and SS_Reset signals. The first sample is affected by the SS_Reset signal. It is important to have the right numbers of clock cycles and sample the output on the falling edge of HSCLK.

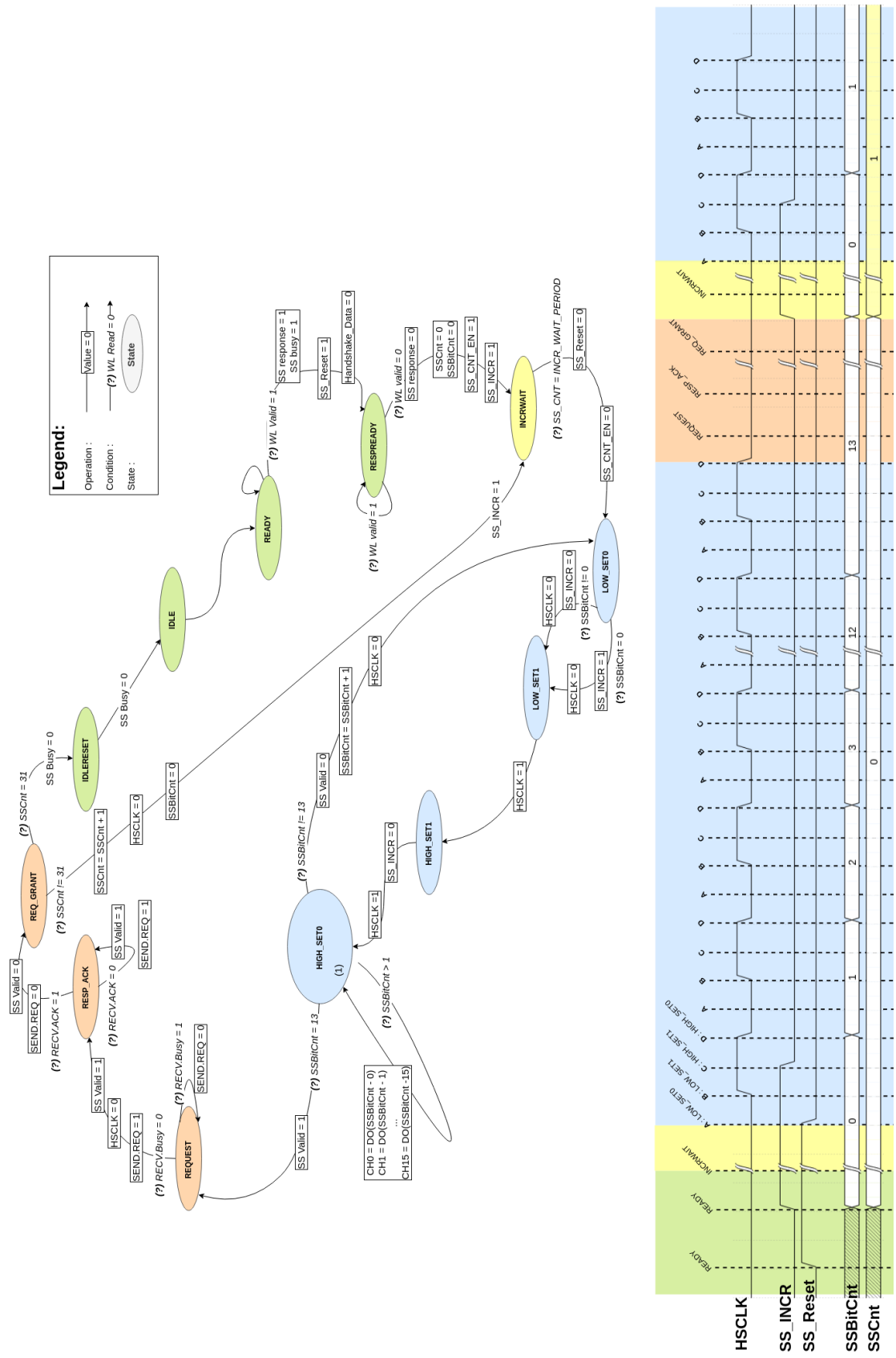


Figure 3.23: State Machine for the sample select process according to the specification

The simulations show the correct sequence between processes. The figure 3.24 demonstrates the SS_INCR setup time and the number of clock cycle to readout data from the TARGETC. At the end of the first sample readout, a handshake transaction takes place between this process and the FIFO manager. Indeed the data acquired must be saved into the 16 FIFOs. Once the write completed, the process re-initiates a sample readout and so on. Figure 3.25 validate the readout sequence for all 32 samples and write into the FIFOs.

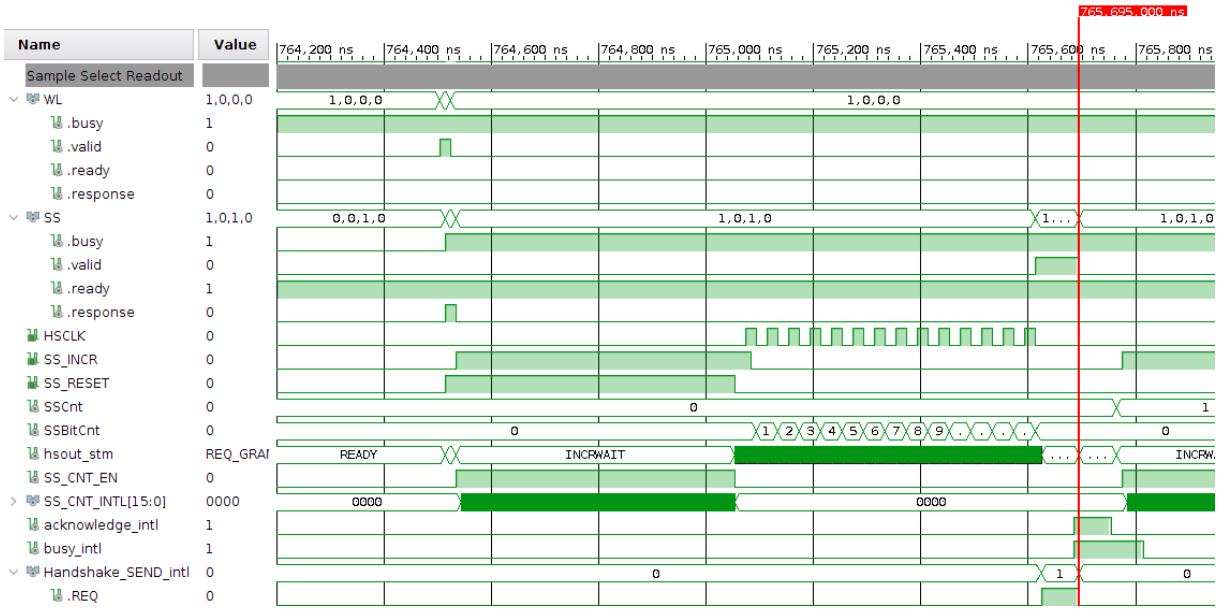


Figure 3.24: Sim : Sample Select process with the interaction of Wilkinson conversion and FIFO manager, Sample 0

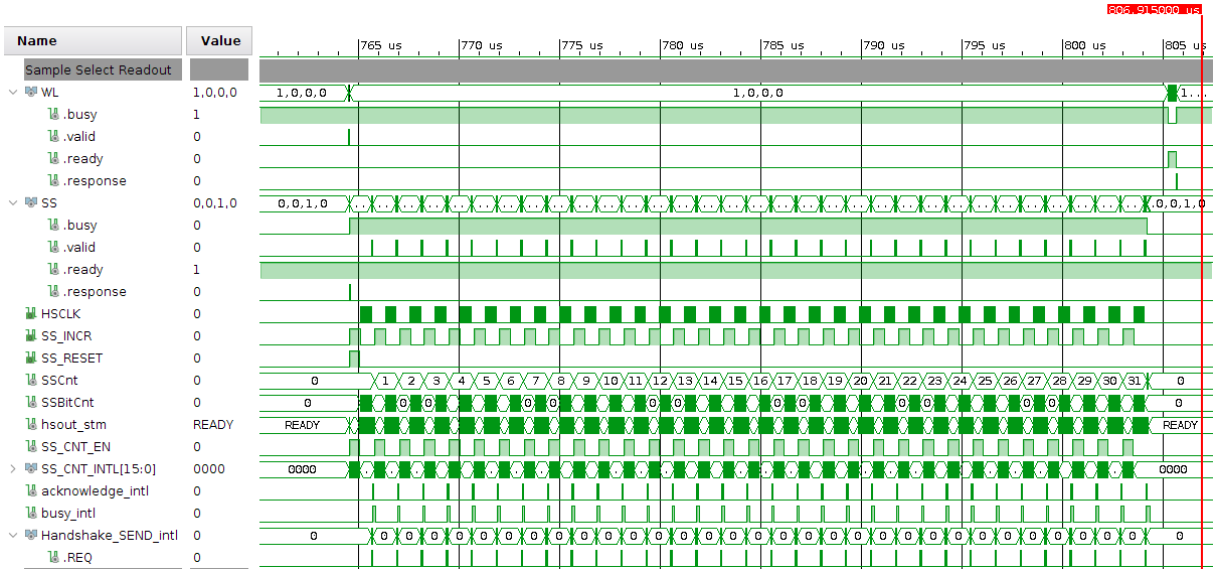


Figure 3.25: Sim : Sample Select process with the interaction of Wilkinson conversion and FIFO manager, Sample 0 to 31

In order to verify the signal transitions, an ILA core is implemented. This Integrate Logic analyzer will take 1024 samples over a small period of time, enough to see the complete first readout. The MSB is shifted out in last position.

	CH0	CH1	CH2	CH3	CH4	CH5	CH6	CH7	CH8	CH9	CH10	CH11	CH12	CH13	CH14	CH15
S0 :	559	560	541	551	543	569	559	569	558	550	556	568	555	558	575	572

Figure 3.26: Sample 0 readout all channels from the PL side

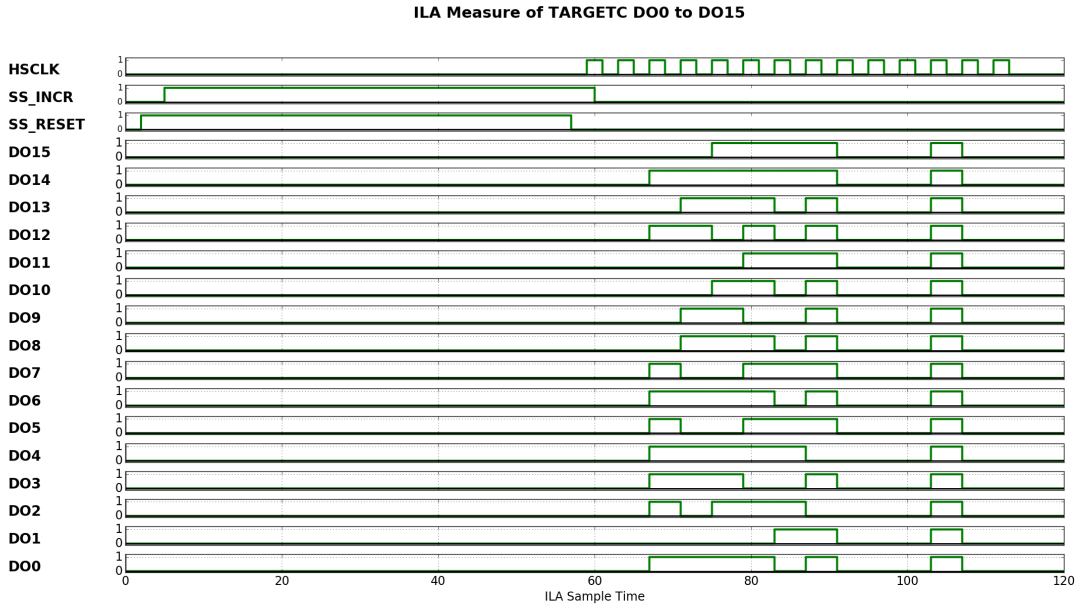


Figure 3.27: Sample 0 readout all channels with ILA

CH0 :	$559_{10} = 1000101111_2$	CH8 :	$558_{10} = 1000101110_2$
CH1 :	$560_{10} = 1000110000_2$	CH9 :	$550_{10} = 1000100110_2$
CH2 :	$541_{10} = 1000011101_2$	CH10 :	$556_{10} = 1000101100_2$
CH3 :	$551_{10} = 1000100111_2$	CH11 :	$568_{10} = 1000101100_2$
CH4 :	$543_{10} = 1000011111_2$	CH12 :	$555_{10} = 1000101011_2$
CH5 :	$569_{10} = 1000111001_2$	CH13 :	$558_{10} = 1000101110_2$
CH6 :	$559_{10} = 1000101111_2$	CH14 :	$575_{10} = 1000111111_2$
CH7 :	$569_{10} = 1000111001_2$	CH15 :	$572_{10} = 1000111100_2$

The same test is ran by adjusting the Vped voltage to 0 Volt (DC Offset on the RFIN inputs of the TARGETC). The results show that the ASIC for 0 Volt input is still sending out data, internally the ASIC is at negative saturation. The transistors are at lowest voltage possible.

	CH0	CH1	CH2	CH3	CH4	CH5	CH6	CH7	CH8	CH9	CH10	CH11	CH12	CH13	CH14	CH15
S0 :	125	136	108	126	119	136	134	132	128	124	136	138	119	131	136	132

Figure 3.28: Sample 0 readout all channels from the PL side (with Vped = 0)

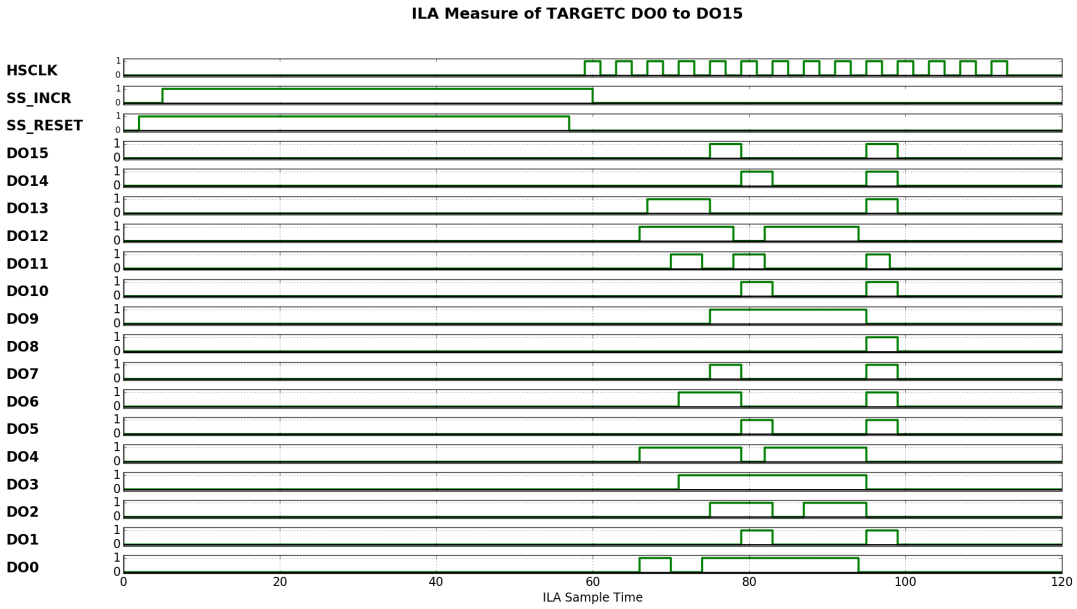


Figure 3.29: Sample 0 readout all channels with ILA (with Vped = 0)

CH0 : $125_{10} = 000001111101_2$
CH1 : $136_{10} = 000010001000_2$
CH2 : $108_{10} = 000001101100_2$
CH3 : $126_{10} = 000001111110_2$
CH4 : $119_{10} = 000001110111_2$
CH5 : $136_{10} = 000010001000_2$
CH6 : $134_{10} = 000010000110_2$
CH7 : $132_{10} = 000010000100_2$

CH8 : $128_{10} = 000010000000_2$
CH9 : $124_{10} = 000001111100_2$
CH10 : $136_{10} = 000010001000_2$
CH11 : $138_{10} = 000010001010_2$
CH12 : $119_{10} = 000001110111_2$
CH13 : $131_{10} = 000010000011_2$
CH14 : $136_{10} = 000010001000_2$
CH15 : $132_{10} = 000010000100_2$

(TPG) The test pattern generator verifies the write to TARGETC Register and correct sampling of the outcoming data. It is a very conclusive test, see figure 3.30. After reading out the 32 samples from the 16 channels, all values are equal to the TPG register set prior to 12316.

```
CH0    CH1    CH2    CH3    CH4    CH5    CH6    CH7    CH8    CH9    CH10   CH11   CH12   CH13   CH14   CH15
----   ----   ----   ----   ----   ----   ----   ----   ----   ----   ----   ----   ----   ----   ----   ----
>>>
0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123
0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123
0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123
0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123
0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123
0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123
0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123
0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123
0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123
0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123
0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123
0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123
0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123
0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123
0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123
0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123
0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123
0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123
0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123
0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123
0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123
0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123
0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123
0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123
0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123
0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123
0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123
0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123
0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123
0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123
0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123
0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123  0x123
<<<
```

Figure 3.30: Readout from TARGETC, for all 32 samples of the 16 channels, TPG Register = 123_{16}

(Normal Readout) The normal readout verifies the correct sample order. This is to say that sample 0 is actually sample 0 and not 31 or 1. TARGET design over the last years have shown the same characteristics which is that sample 31 is always very different (lower value) from the others sample, by plotting all the data, sample 31 is indeed different (see figure 3.31).

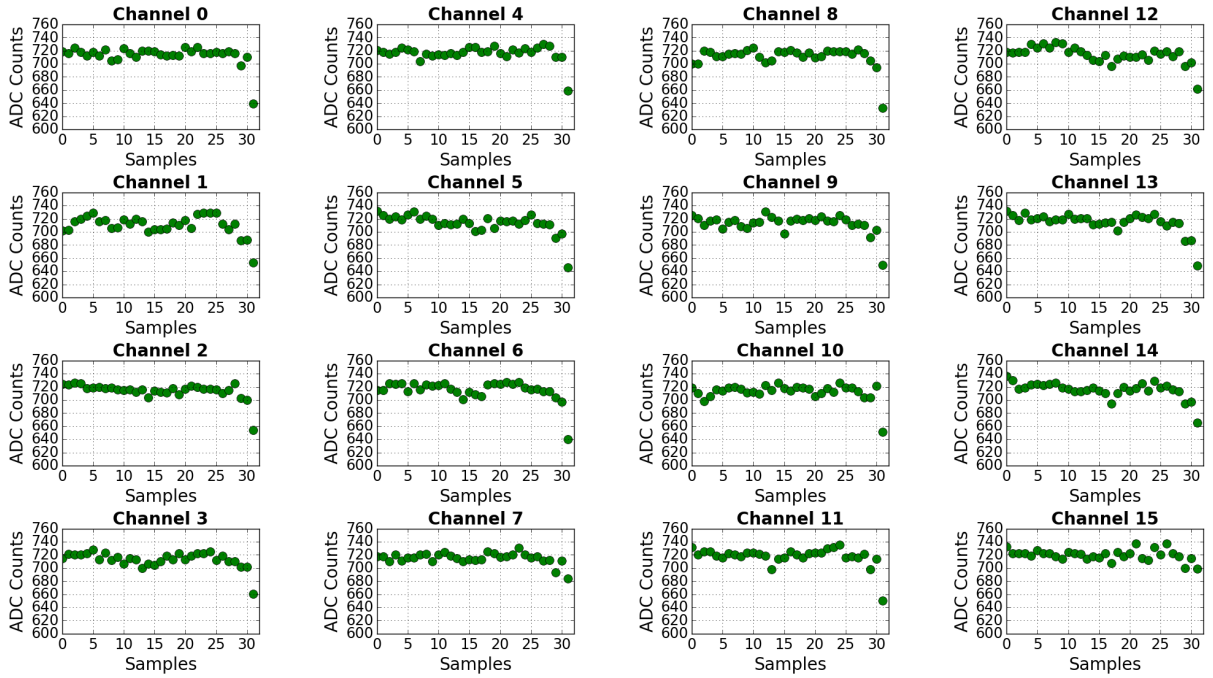


Figure 3.31: Readout from TARGETC, for all 32 samples of the 16 channels

3.5 FIFO Manager

The FIFO Manager is the data concentrator, it accumulates the 32 samples by 16 channels into 16 synchronous FIFOs. When all of them are full, it sends them through the AXI-Stream. Figure 3.32 shows the overview of the system, there should be a module able to select which FIFO to read first depending on the number of data sent out. The data structure is as follows.

Word (32-Bit)	Description	Test FIFO Values
WINDOW HEADER		
0	Time of Sampling 64-Bit (LSB)	$-1_{10} = FFFFFFFF_{16}$
1	Time of sampling 64-Bit (MSB)	0_{10}
2	Spare	0_{10}
3	Spare	$-1_{10} = FFFFFFFF_{16}$
4	Trigger Information	$305419896_{10} = 12345678_{16}$
5	Window ID	$438_{10} = 1B6_{16} = 110110110_2$
WINDOW DATA		
6 - 37	Channel 0 (Sample 0 to 31)	$16 * SampleID$
38 - 69	Channel 1	$16 * SampleID + 1$
70 - 101	Channel 2	$16 * SampleID + 2$
102 - 133	Channel 3	$16 * SampleID + 3$
134 - 165	Channel 4	$16 * SampleID + 4$
166 - 197	Channel 5	$16 * SampleID + 5$
198 - 229	Channel 6	$16 * SampleID + 6$
230 - 261	Channel 7	$16 * SampleID + 7$
262 - 293	Channel 8	$16 * SampleID + 8$
294 - 325	Channel 9	$16 * SampleID + 9$
326 - 357	Channel 10	$16 * SampleID + 10$
358 - 389	Channel 11	$16 * SampleID + 11$
390 - 421	Channel 12	$16 * SampleID + 12$
422 - 453	Channel 13	$16 * SampleID + 13$
454 - 485	Channel 14	$16 * SampleID + 14$
486 - 517	Channel 15	$16 * SampleID + 15$

Table 3.4: AXI-Stream Window packet

The FIFO manager has two processes, Write and Read. The write process is receiving the data from the sample readout, when it received 32 samples it initiates a handshake with the read process. The read starts transmitting the header information to the AXI-Stream component. It continues by sending the data of the first FIFO followed by the second and so on until all FIFOs are empty.

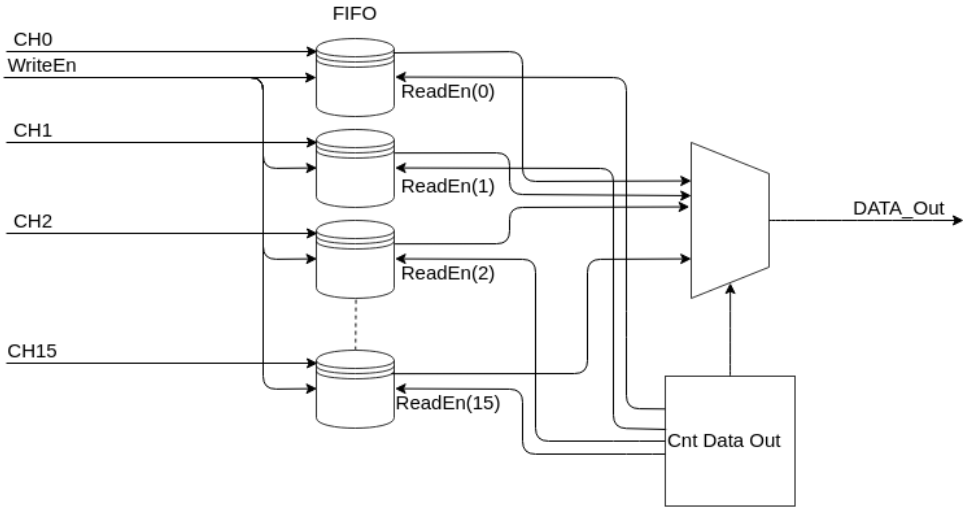


Figure 3.32: FIFO Manager Overview

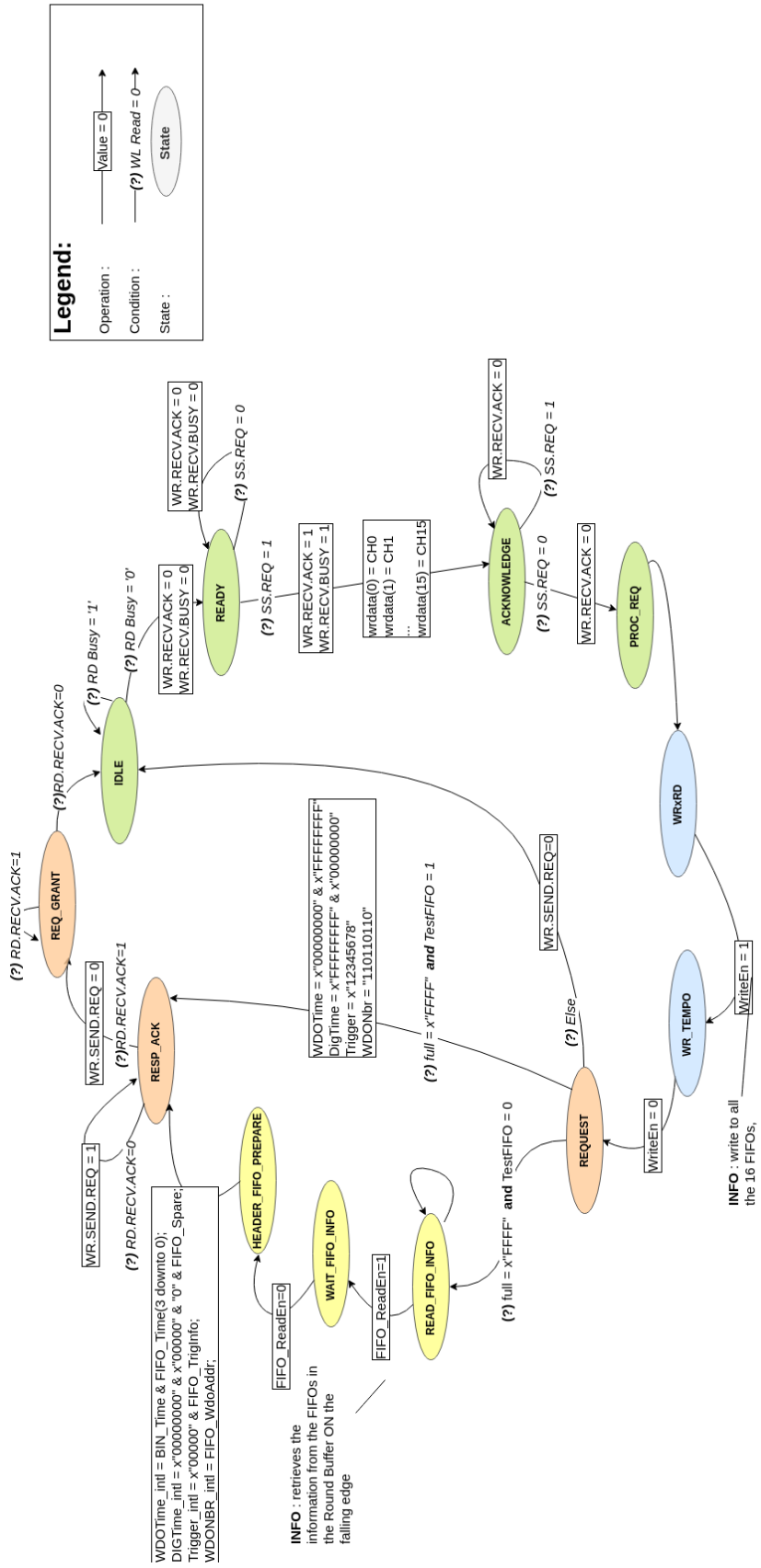


Figure 3.33: State Machine for the write process of process according to the specification

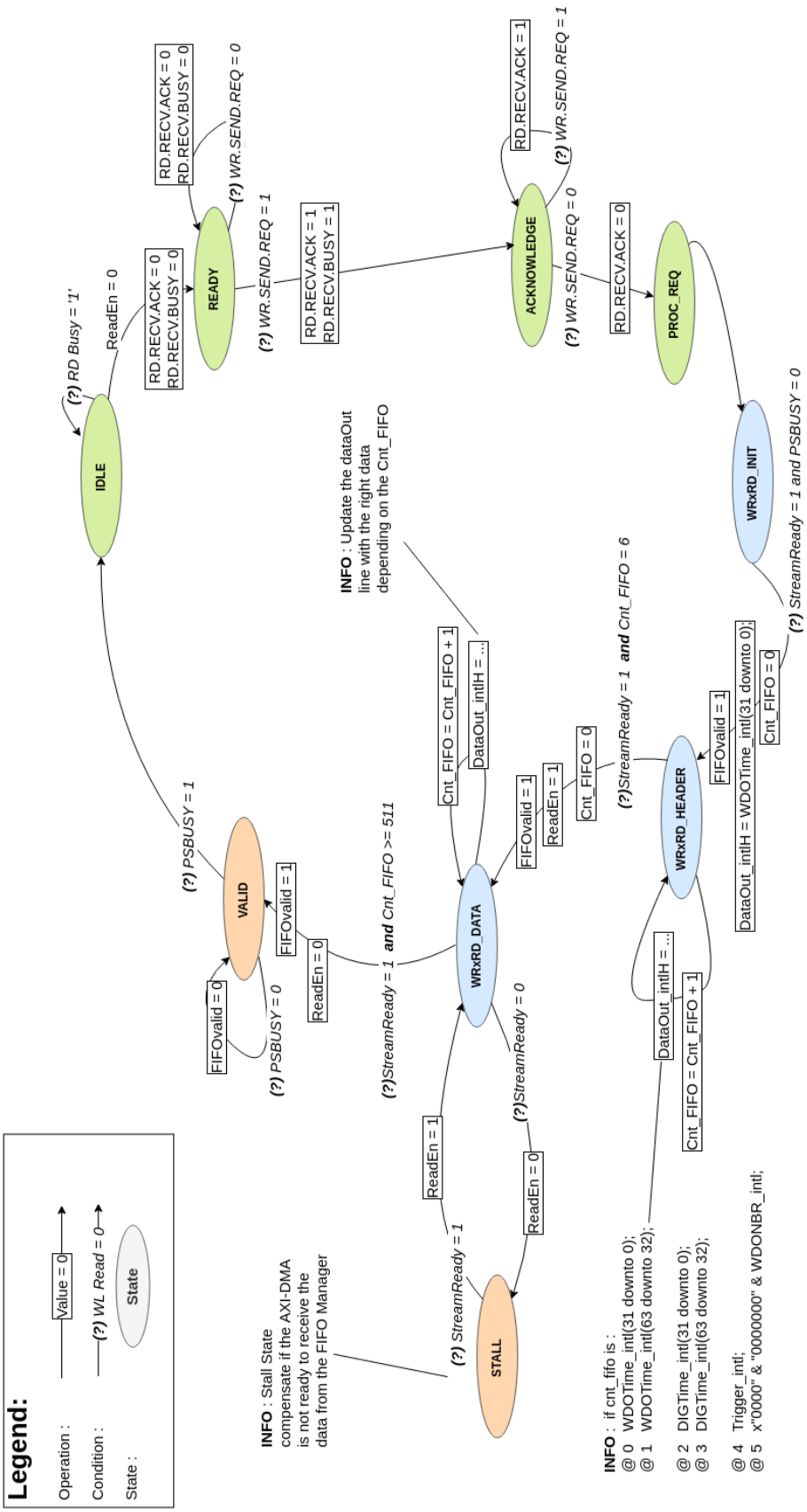


Figure 3.34: State Machine for the read process of process according to the specification

A built-in test function in the PL verifies if the FIFO Manager is correctly responding. The bit C_TESTFIFO_BIT from the TC_CONTROL_REG enables the sample readout process to output a set of known values on the channels data bus and triggers the valid signal which will cause the FIFO Manager to save the values into the FIFOs. When these ones are full, the read process will send them through the AXI-Stream. The outputted data should look like in the table 3.4.

```
*** Test FIFO ***
...
HEADER
------------
Header0:        -1
Header1:        0
Header2:        0
Header3:        -1
Header4:        305419896
Header5:        438
CH0     CH1     CH2     CH3     CH4     CH5     CH6     CH7     CH8     CH9     CH10    CH11    CH12    CH13    CH14 CH15
----    ----    ----    ----    ----    ----    ----    ----    ----    ----    ----    ----    ----    ----    ---- ----
>>>
0       1       2       3       4       5       6       7       8       9       10      11      12      13      14   15
16      17      18      19      20      21      22      23      24      25      26      27      28      29      30   31
32      33      34      35      36      37      38      39      40      41      42      43      44      45      46   47
48      49      50      51      52      53      54      55      56      57      58      59      60      61      62   63
64      65      66      67      68      69      70      71      72      73      74      75      76      77      78   79
80      81      82      83      84      85      86      87      88      89      90      91      92      93      94   95
96      97      98      99      100     101     102     103     104     105     106     107     108     109     110  111
112     113     114     115     116     117     118     119     120     121     122     123     124     125     126  127
128     129     130     131     132     133     134     135     136     137     138     139     140     141     142  143
144     145     146     147     148     149     150     151     152     153     154     155     156     157     158  159
160     161     162     163     164     165     166     167     168     169     170     171     172     173     174  175
176     177     178     179     180     181     182     183     184     185     186     187     188     189     190  191
192     193     194     195     196     197     198     199     200     201     202     203     204     205     206  207
208     209     210     211     212     213     214     215     216     217     218     219     220     221     222  223
224     225     226     227     228     229     230     231     232     233     234     235     236     237     238  239
240     241     242     243     244     245     246     247     248     249     250     251     252     253     254  255
256     257     258     259     260     261     262     263     264     265     266     267     268     269     270  271
272     273     274     275     276     277     278     279     280     281     282     283     284     285     286  287
288     289     290     291     292     293     294     295     296     297     298     299     300     301     302  303
304     305     306     307     308     309     310     311     312     313     314     315     316     317     318  319
320     321     322     323     324     325     326     327     328     329     330     331     332     333     334  335
336     337     338     339     340     341     342     343     344     345     346     347     348     349     350  351
352     353     354     355     356     357     358     359     360     361     362     363     364     365     366  367
368     369     370     371     372     373     374     375     376     377     378     379     380     381     382  383
384     385     386     387     388     389     390     391     392     393     394     395     396     397     398  399
400     401     402     403     404     405     406     407     408     409     410     411     412     413     414  415
416     417     418     419     420     421     422     423     424     425     426     427     428     429     430  431
432     433     434     435     436     437     438     439     440     441     442     443     444     445     446  447
448     449     450     451     452     453     454     455     456     457     458     459     460     461     462  463
464     465     466     467     468     469     470     471     472     473     474     475     476     477     478  479
480     481     482     483     484     485     486     487     488     489     490     491     492     493     494  495
496     497     498     499     500     501     502     503     504     505     506     507     508     509     510  511
<<<
```

Figure 3.35: Terminal output of built-in test function for the FIFO Manager

3.6 AXI-Stream

The AXI-Stream component takes the data from the FIFO Manager and shifts it out onto the AXI-Stream bus using the correct AXI sequence. Vivado provides, like for the AXI-Lite, an example that was modified for the needs of the project. An additional feature is added like the STALL state, which deals with the M_AXIS_TREADY going LOW. The AXI-Stream enters a STALL state, in which it must stop the FIFO Manager from sending more data, because the AXI-Slave is not ready to receive. Once this signal goes HIGH, the AXI-Stream component exits the STALL state and starts streaming again. This component has a built-in test function which verifies its behavior. The test sequence is sending the data from 0 to 517.

```
*** Test Stream ***
...
HEADER
------------
Header0:        0
Header1:        1
Header2:        2
Header3:        3
Header4:        4
Header5:        5
CH0     CH1     CH2     CH3     CH4     CH5     CH6     CH7     CH8     CH9     CH10    CH11    CH12    CH13    CH14    CH15
----    ----    ----    ----    ----    ----    ----    ----    ----    ----    ----    ----    ----    ----    ----    ----
>>>
6       38      70      102     134     166     198     230     262     294     326     358     390     422     454     486
7       39      71      103     135     167     199     231     263     295     327     359     391     423     455     487
8       40      72      104     136     168     200     232     264     296     328     360     392     424     456     488
9       41      73      105     137     169     201     233     265     297     329     361     393     425     457     489
10      42      74      106     138     170     202     234     266     298     330     362     394     426     458     490
11      43      75      107     139     171     203     235     267     299     331     363     395     427     459     491
12      44      76      108     140     172     204     236     268     300     332     364     396     428     460     492
13      45      77      109     141     173     205     237     269     301     333     365     397     429     461     493
14      46      78      110     142     174     206     238     270     302     334     366     398     430     462     494
15      47      79      111     143     175     207     239     271     303     335     367     399     431     463     495
16      48      80      112     144     176     208     240     272     304     336     368     400     432     464     496
17      49      81      113     145     177     209     241     273     305     337     369     401     433     465     497
18      50      82      114     146     178     210     242     274     306     338     370     402     434     466     498
19      51      83      115     147     179     211     243     275     307     339     371     403     435     467     499
20      52      84      116     148     180     212     244     276     308     340     372     404     436     468     500
21      53      85      117     149     181     213     245     277     309     341     373     405     437     469     501
22      54      86      118     150     182     214     246     278     310     342     374     406     438     470     502
23      55      87      119     151     183     215     247     279     311     343     375     407     439     471     503
24      56      88      120     152     184     216     248     280     312     344     376     408     440     472     504
25      57      89      121     153     185     217     249     281     313     345     377     409     441     473     505
26      58      90      122     154     186     218     250     282     314     346     378     410     442     474     506
27      59      91      123     155     187     219     251     283     315     347     379     411     443     475     507
28      60      92      124     156     188     220     252     284     316     348     380     412     444     476     508
29      61      93      125     157     189     221     253     285     317     349     381     413     445     477     509
30      62      94      126     158     190     222     254     286     318     350     382     414     446     478     510
31      63      95      127     159     191     223     255     287     319     351     383     415     447     479     511
32      64      96      128     160     192     224     256     288     320     352     384     416     448     480     512
33      65      97      129     161     193     225     257     289     321     353     385     417     449     481     513
34      66      98      130     162     194     226     258     290     322     354     386     418     450     482     514
35      67      99      131     163     195     227     259     291     323     355     387     419     451     483     515
36      68      100     132     164     196     228     260     292     324     356     388     420     452     484     516
37      69      101     133     165     197     229     261     293     325     357     389     421     453     485     517
<<<
```

Figure 3.36: Terminal output of built-in test function for the AXI-Stream

4 PS : Processing System

DISCLOSURE : The main functions and applications were develop and explained in the Master Thesis of Anthony Schluchin, colleague on this WATCHMAN collaboration. However all functions interfacing the programmable logic (PL) were investigated by myself before passing the knowledge further on. The main purpose of this chapter is to give the necessary information on the PS side interfacing with the PL and TARGETC.

4.1 Library Target C

4.1.1 Write to Register

One of the most important function to interface the TARGETC is the necessity to write to its register. The AXI-Lite is seen as a bidirectional array from the PS side. The BASE_ADDRESS parameter in the "xparametes.h" file links the hardware to the board support package, this pointer refers to the array's first address. For reminder the TARGETC has several registers from 1 to 128 but not all addresses are used. However to keep the code as simple as possible an array of 128 is reserved. The addresses are the register's ID. To initiate a write to TARGETC register, the following steps are performed.

1. The register's ID is written into the TC_ADDR_REG.
2. The data is updated in the register needed to be transfered to the TARGETC.
3. WRITE_BIT in the TC_CONTROL_REG is enabled. (Set to HIGH) This will trigger a rising edge and start the write register process. The principle is that this will cause the PL side to fetch the data in the register pointed by TC_ADDR_REG and send it to the TARGETC, previously explained.
4. The status register (TC_STATUS_REG) is checked to see if the transaction is finished.
5. WRITE_BIT in the TC_CONTROL_REG is disabled (Set to LOW)

Code 4.1: TC_CONTROL_REG

```
1 int WriteRegister(int regID, int regData){
2
3     ControlRegisterWrite(WRITE_MASK,DISABLE);
4
5     regptr[regID] = regData;
```

```
6       regptr[TC_ADDR_REG] = regID;
7
8       ControlRegisterWrite(WRITE_MASK ,ENABLE);
9       while(regptr[TC_STATUS_REG] & BUSY_MASK){
10          usleep(100); //sleep 100ms
11      }
12      ControlRegisterWrite(WRITE_MASK,DISABLE);
13  }
```

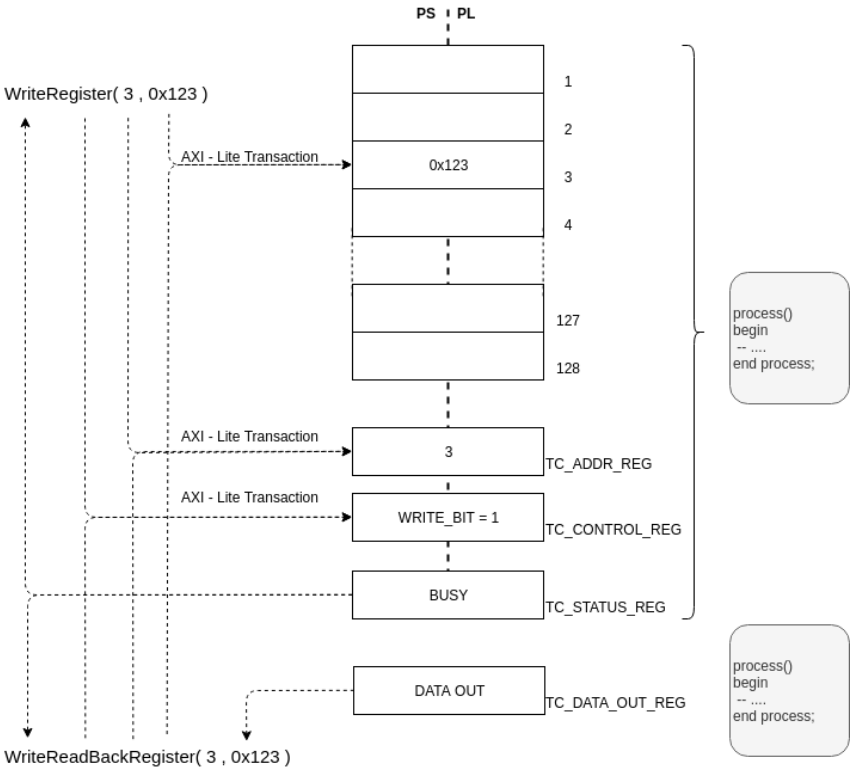


Figure 4.1: Principle of write to the Target C Regsiter

The TARGETC is write only, the system uses a shift register system. The input data is shifted into some D type Flip-Flops (DFF) on every clock cycle and the last DFF output is redirected to an output pin. As a consequence, the last register written can be read back by flushing all the DFF, this is simply implement by sending a new write register. However the outputted data is not directly the register's value but the shifted data. An earlier version of this function WriteReadBackRegister was used to check every transaction between the FPGA and ASIC. The output was sampled at the same time the second write was shifted out. The returned value was read on TC_DATA_OUT_REG.

Test Pattern Register Write
W: Reg: 80 Data: 123
RB: 7F123

The TARGETC has this PCLK address which is different from the real address of the register. The small test above shows the write to register 80_{16} (TPG address) with the value 123_{16}. Once the two transactions are finished, the register TC_DATA_OUT_REG returns the shifted data. The readout is equal to the PLCK address followed by the value 0x123. Like previously mentioned this test only verifies that the interface is good not the data inside the ASIC.

4.2 UART Communication

For debugging purpose and development, the UART was setup to communicate both ways. The bare metal application does not have the function scanf, which enable the user to enter a number or string value. After pressing the enter key this value is saved into a variable. The application only checks if a byte was received on the RX line. The function XUartPs_IsReceiveData returns a boolean value, if the return value is true, the function XUartPs_ReadReg(STDIN_BASEADDRESS, XUARTPS_FIFO_OFFSET) is called. This will read the character in the reception FIFO. The UART uses ASCII table to transmit data, therefore the input number on the terminal are ASCII number and must be translated into an integer. This function is really for debugging and development purposes, so no further development was done on the function.

4.3 Cache Coherence with AXI-DMA

The AXI-DMA raises an interrupt every time a AXI-Stream transfer is complete. The processor has no clue that a transaction modified the memory, because the HP (High performance) ports were used and not the ACP (Accelerator Coherency port) which include a cache coherence system, but is more difficult to setup. The system was kept as simple as possible. Therefore some cache coherence were possible. Looking at the internal architecture of the Zynq device, the memory controller for the DDR shares a direct bus with the ARMs and a second one with the HP ports.

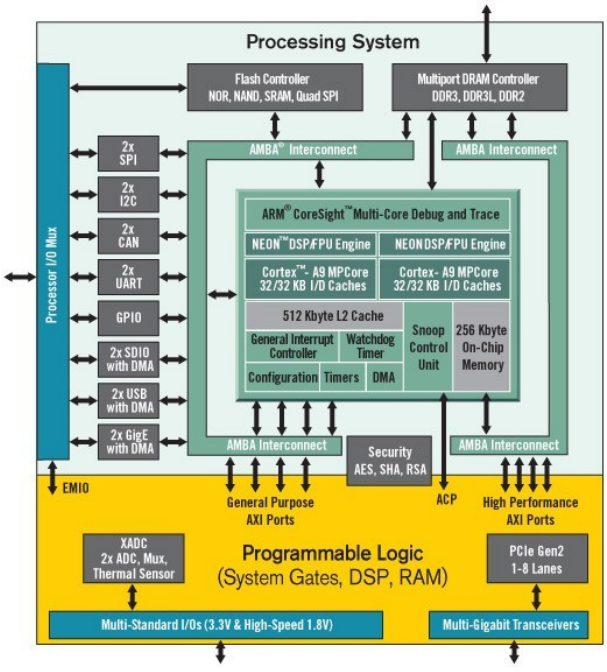
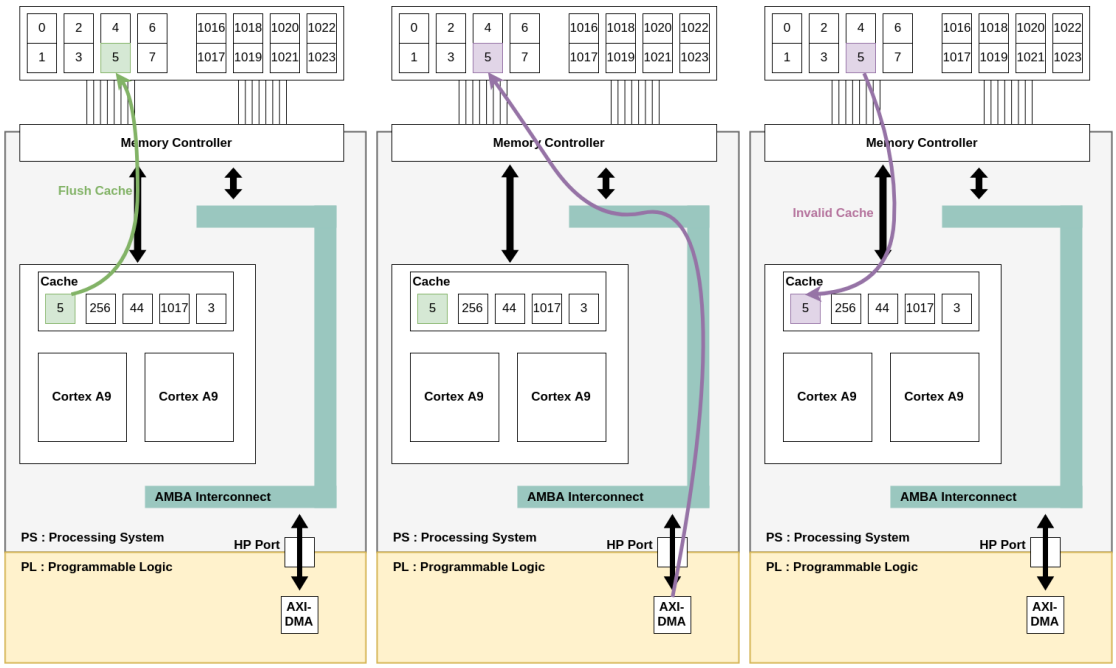


Figure 4.2: Zynq©-7000 Soc internal architecture

The cache coherence is an important aspect to take into account when working with double memory write access. The functions Xil_DCacheFlushRange() and Xil_DCacheInvalidateRange() both take, the address of the data as first parameter and the length to flush or invalidate out/in the cache. The example below shows the correct method on how to work with the cache.



(a) Flush the cache into memory

(b) AXI-DMA Transfer

(c) Invalidate cache after transfer

Figure 4.3: Next address problematic

In figure 4.3a, the processor flushes the cache into the memory. This step is important in case the cache refreshes itself and re-allocates this block of cache. In the case where the flush function was not called, this specific block is marked inconsistent with the memory. The cache system will replace the data in memory with the cache's values and so if a transfer is already underway, the new values will be replaced and overwritten with erroneous values. In consequence prior calling of this flush function avoids this issue.

In figure 4.3b, the AXI-DMA transfers the data from the PL side to the PS from this moment on, the memory and cache are incoherent, but the cache system has no idea of the situation. At the end of the data transfer and an AXI-DMA interrupt is raised.

Once the interruption occurs, the cache is invalidated to make it aware that memory and cache are incoherent. The cache will update by fetching the memory values, see figure 4.3c

5 Further development

This chapter will cover the actual firmware development which was a result of the resources reduction, timing issues and optimizations.

5.1 Resources Reduction

The first step is to have a design which can be implemented into the actual FPGA. To do so the entire system had to be modified and every line had to be evaluated, if it was necessary or not. Many ideas were elaborated using ”tri-state” bus inside the FPGA, but this equals to a big multiplexer. One of the best outcomes is the actual result.

NOTE: The resources which are over utilized are Slice LUTs, which can be LUTs as memory or as Logic. By reducing the number of lines, the gain of Slice Register is greatly improved. This is not directly linked to the LUTs as logic utilization. Overall each line is attached to a DFF (D-type Flip-Flop) if they react to a clock. Reducing these lines implies simplifying the logic behind, which makes it possible to fit the overall system in the FPGA.

The overall system is illustrated in figure 5.1. The system has enormously changed regarding bus width and also new signal have appeared. In other words every bus width and trigger information were reduced to minimum size, respectively resources.

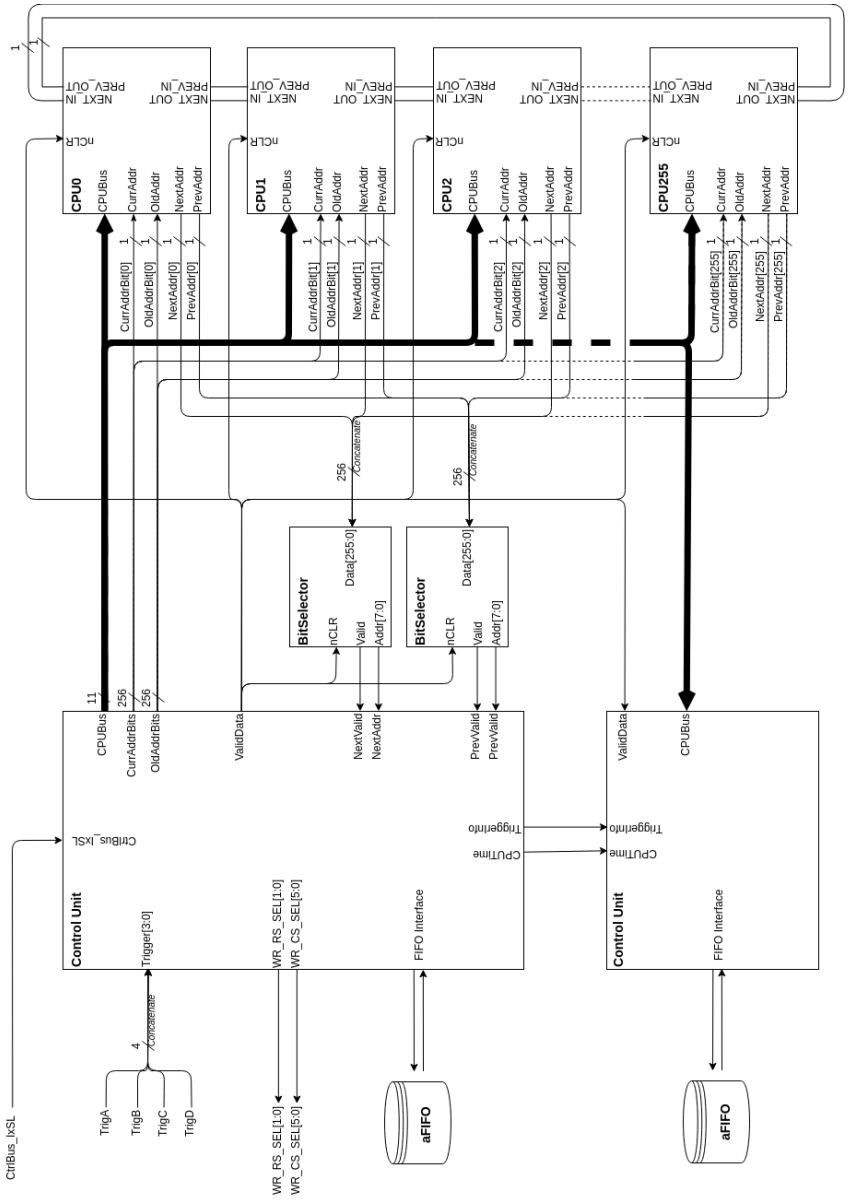


Figure 5.1: Round Buffer system architecture

5.1.1 Old and Current Address

While the current address is being sampled, no read operation can be performed on this same window. Only at the next SSTIN clock period will this window be eligible for readout and digitization. The signal Old address keeps track of the old window address. Like the information is old of a SSTIN period, the command is sent on the CPUBus and also to the store unit, which no longer keeps track of the CPUs, but receives the new command and treats it immediately. To keep the resources low, the 8-bit wide current and old address are not sent to each CPU ($2 * 256 * 8 = 4096$ `DFFs`). They are converted to a single bit connected to its CPU ($2 * 256 = 512$ `DFFs`).

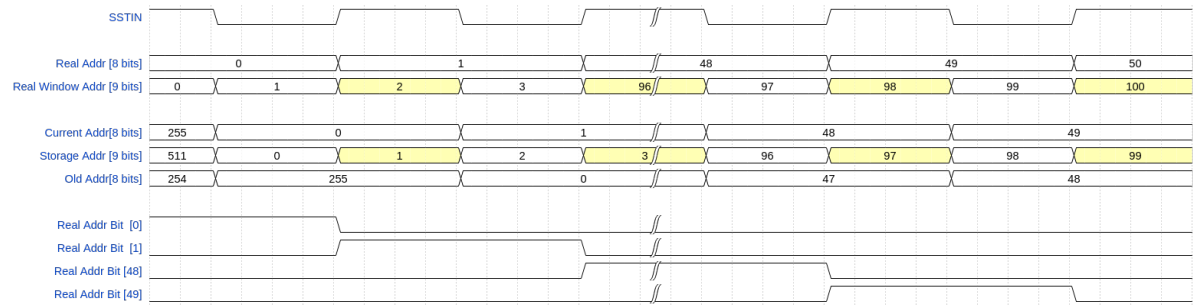


Figure 5.2: Chronogram of old and current address

The old address differs from the previous address, because the previous is always recalculated depending on the internal write enables. The old address is an exact copy of the current window one clock later.

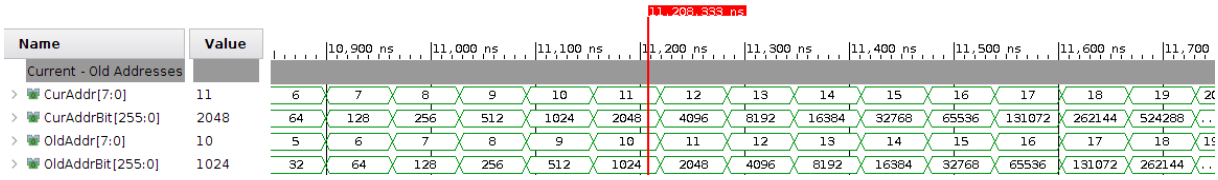


Figure 5.3: Simulation of the old and current address

In order to properly "Jump Start" the round buffer, the initial values are very important. The current address starts at 0_{16}", meaning CurAddrBit(0) is HIGH, the others are kept LOW. Respectively old address is FF_{16} and OldAddrBit(255) is HIGH.

Code 5.1: Initialization of Old and Current Address

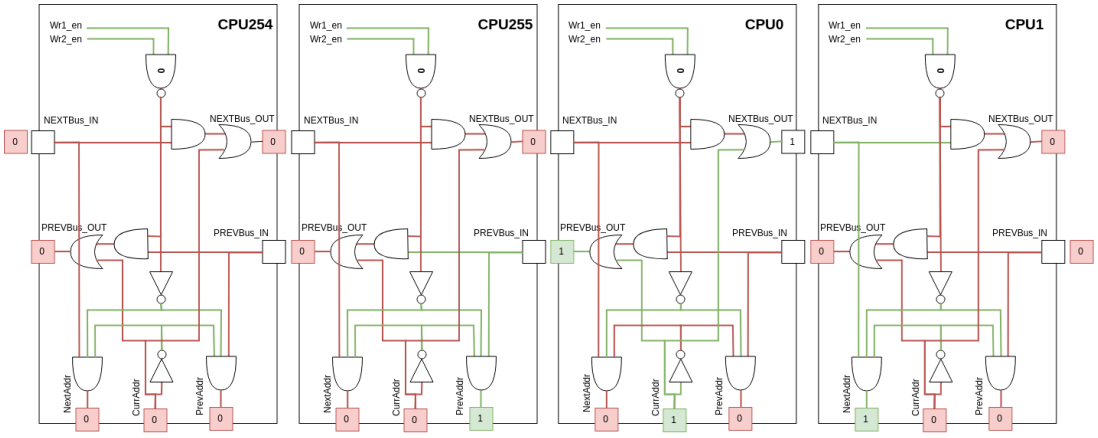
```
1  if nRST = '0' then
2      CurAddr_s    <= x"00";
3      CurAddrBit   <= (0 => '1', others => '0');
4
5      OldAddr_intl <= x"FF";
6      OldAddrBit   <= (255 => '1', others => '0');
7    --...
8  else
9    --...
```

5.1.2 Previous and Next bus

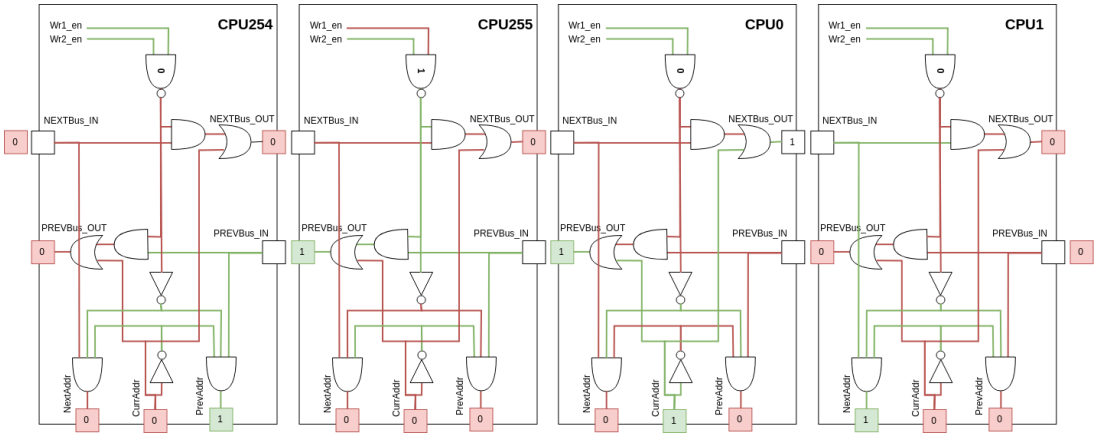
The next and previous address bus between CPUs is also reduced from $256 * 4 * 8 + 256 * 2 * 8 = 12288$ `DFFs` to $256 * 4 + 256 * 2 = 1536$ `DFFs`. Looking back at figure 3.7, the principle remains the same except, that an address is not flowing trough the CPU but a bit, indicating by HIGH logic level that it is searching for the next/previous CPU. In addition to these changes, two new outputs are added NextAddr and PrevAddr, which are used to indicate if the CPU is the next/previous CPU on the line depending on the bit shift on PREVBus and NEXTBus.

Figure 5.4 shows the propagation of the bit, if CPU0 is the current address (the one being sampled). In image 5.4a, all CPUs are free. Therefore, the expected previous CPU from CPU0 is 255. The logic inside the CPU processes that the write enables are active, the CPU is not the current address (CurAddr is LOW) and that it is the previous in line (PREVBus_IN is HIGH).

However in case 5.4b, CPU255 is not free, the previous available CPU is 254. This time the logic processes that at least one write enable is inactive and so PREVBus_IN is directly connected to PREVBus_OUT for the previous CPU. CPU254's logic processes the information and sets the PrevAddr to HIGH indicating it is the previous CPU on the line.



(a) All CPUs are available



(b) CPU255 is unavailable

Figure 5.4: Previous and Next bus on a single bit

The simulation of the bus is completed with some DFF for timing issues, the CPUs had to be setup with some pipeline stages. This is why the PrevAddr and NextAddr return to 0_{16} after each change in the current address. At CurAddr equals 0, CPU0 searches for the next and previous CPU by setting HIGH PREVBus and NEXTBus (Bus equal to 1). After a small amount of time, the NextAddrBus passes to 2, which is equal to a HIGH value on NextAddrBus(1), CPU1 is the next available CPU in the chain. PrevAddrBus passes to a large number ($5.789e+76 = 2^{255}$) which is equal to a HIGH value on PrevAddrBus(255), CPU255 is the previous available CPU.

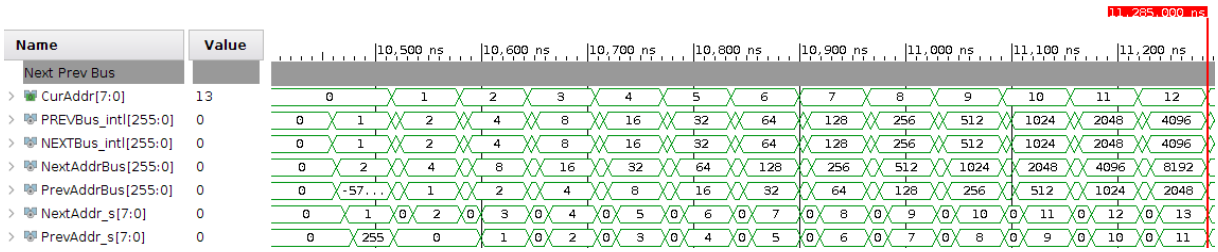


Figure 5.5: Sim : Previous and next bus

The next section will explain how to retrieve the previous or next address from the 256 lines, coming from the CPUs by using the Hamming code correction. This technique will avoid the need of using an enormous amount of multiplexer.

5.1.3 Bit Selector - Hamming Code

Hamming code is a set of error-correction codes that can be used in communication to detect errors in a frame. It uses redundant bits like the CRC (Cyclic-Redundant-Check). In short, the Hamming code is the use of extra bits in the data to identify errors. The redundant bits must be placed at powers of two, 2^n (1, 2, 4, 8, 16 , etc), the data frame is then equal to the figure 5.6. The correction is counting the number of 1 in the frame for each parity bit like shown in figure 5.7.

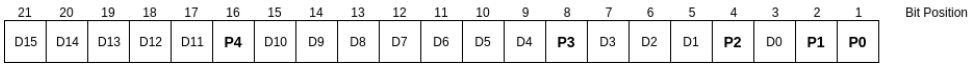


Figure 5.6: Hamming 16 bits data + parity

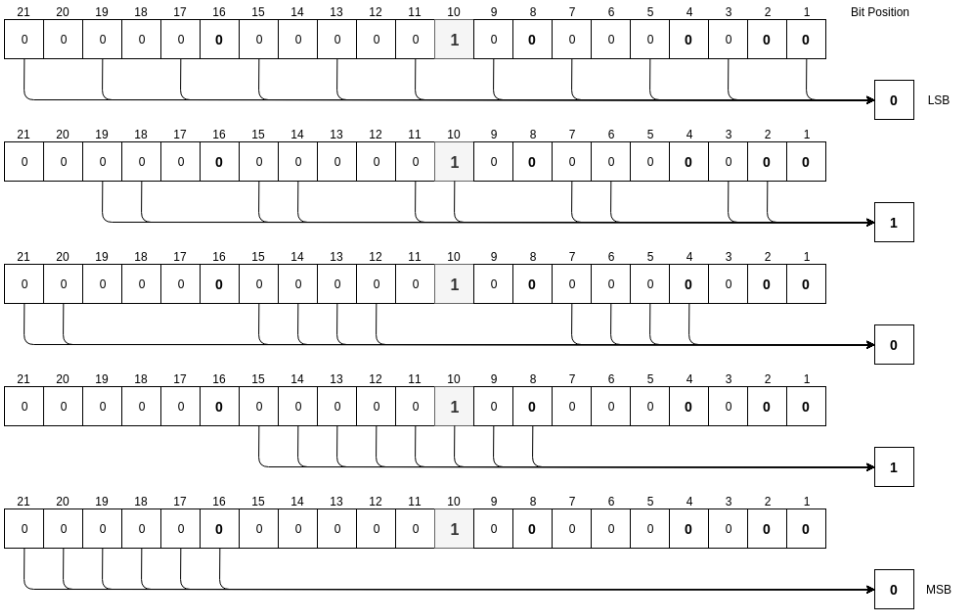


Figure 5.7: Hamming correction for bit position 10

"Why use the Hamming code ?" By using this algorithm, the position of the erroneous bit in the frame gives out the address of the line. This technique can be used to find out the address depending on bit change on a std_logic_vector signal. The CPU has no need to know the next or previous address, but the control unit needs the information. The buses NEXTBus and PREVBus are removed from the CPU and replaces with two std_logic lines (NextAddr and PrevAddr), this change reduces significantly the number of LUT registers in the design from 4096 LUTs to 512. 256 lines must now pass through the hamming code correction. The hamming component is developed for a 16 bits data bus, so a total of 16+1 hamming components are used to select the address from the 256 lines. A small correction has to be made on the 4 bit address, indeed hamming code gives the address of the bit position including the parity bits. They must be subtracted to have the real address of the system, a small lookup table is used.

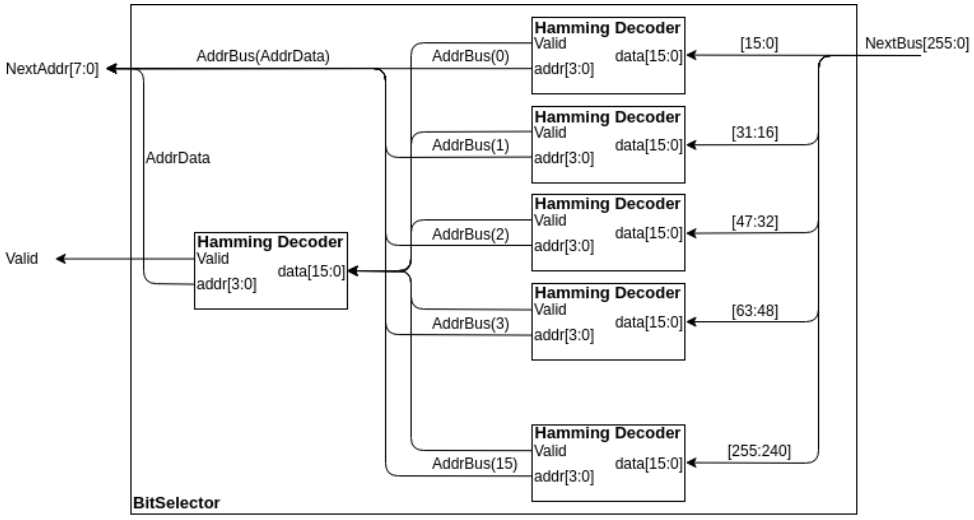


Figure 5.8: Bit to address using hamming code correction for 256 input lines

The code in 5.2 is the testbench used to to test the 256 data input possibilities. The Hamming decoder components are cleared like the CPUs after each rising edge of SSTIN clock. Thus faulty states induced by dual bit insertion are avoided. This feature was added later on in the project, for timing issues, but the logic behind the hamming code is still the same.

Code 5.2: Testbench code for Hamming Decoder Full test

```
1  WRITE(L,string'("Hamming TestBench:" & CR));
2    for I in 0 to 255 loop
3      data_sti <= (I => '1', others => '0');  -- Change the 256 input lines
4
5      while valid_s = '0' loop  -- Waiting on valid address
6        wait until rising_edge(RefClk);
7      end loop;
8
9      -- Checking address against test value
10     if FullAddr = std_logic_vector(to_unsigned(I,8)) then
11       WRITE(L,string'("OK: Addr ="& integer'image(to_integer(unsigned(FullAddr))) & " ↩
             Bit : " & integer'image(I) & CR));
12     else
13       WRITE(L,string'("FAIL: Addr ="& integer'image(to_integer(unsigned(FullAddr))) & "↩
             Bit : " & integer'image(I) & CR));
14     end if;
15     WRITELINE(fd,L);
16     wait for 1 us;
17
18     -- Clear Hamming bits
19     nrst <= '0';
20     data_sti <= (others => '0');
21     wait for 1 us;
22     nrst <= '1';
23   end loop;
```

The simulation report (below) shows that hamming decoder works for the 256 input lines and manages to output the correct address. The simulation waveform is also a good way to check the behavior of the system. The signal fullAddr is the exponent of two to the power to equal data_sti, $\texttt{data\_sti} = 2^{\texttt{FullAddr}}$.

```
Hamming TestBench Simulation End
Success : 256/256 = 100%
Error : 0/256 = 0%
```

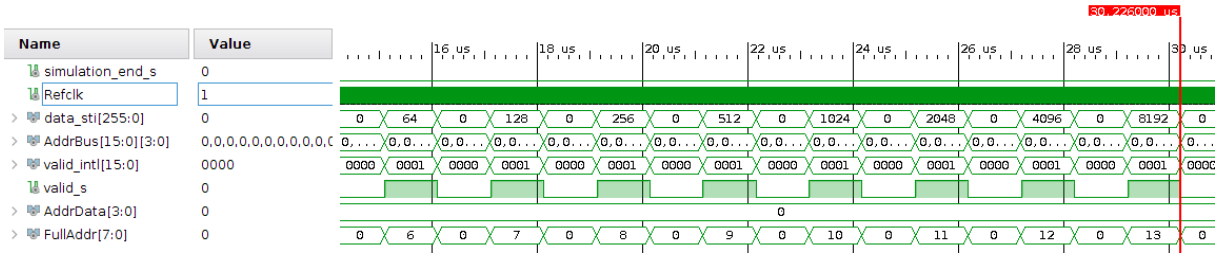


Figure 5.9: Waveform simulation of the Hamming Decoder for 256 input line

5.1.4 Control Unit

The control Unit is now much more flexible than before, the round buffer gives back the next and previous addresses. The OldAddr address keeps track of sampled addresses. The system is upgraded to an individual trigger information recovery unit and now all trigger information are processed together. Like the trigger information is sampled during the current address at the time of the old address, the information is exact for the current window.

5.1.4.1 Trigger System

The trigger controller evaluates each trigger separately and then processes them together when the information (TRIG, LAST and LONG) are found. These information are for the current address as they are real time. They need to be delayed of one SSTIN clock and stretched. This process is to keep the actual information constant for a least a clock period equal to SSTIN.

Each trigger is treated individually, the block single trigger will output in real time the information regarding the trigger. The LONG signal will only fire if the trigger has been HIGH for a longer period of time than a threshold value, represented by a generic parameter in VHDL. Therefore the single trigger block contains a 8-bit counter with enable. The figure 5.11 bellow shows the behavior of this component for two different trigger, a short and long one. The Write enable 1 process keeps track of this signal status depending on the sample count given by the time counter, same for the write enable 2 process. These signal serve to define the command which has to be sent to the CPUs and store unit. In a simplified manner, which window is of interest.

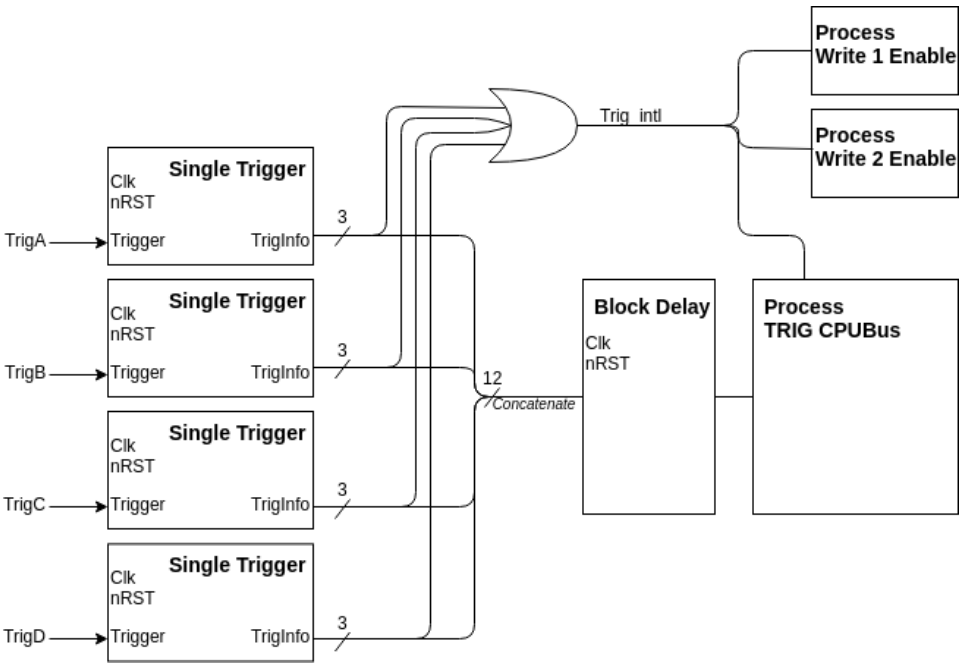
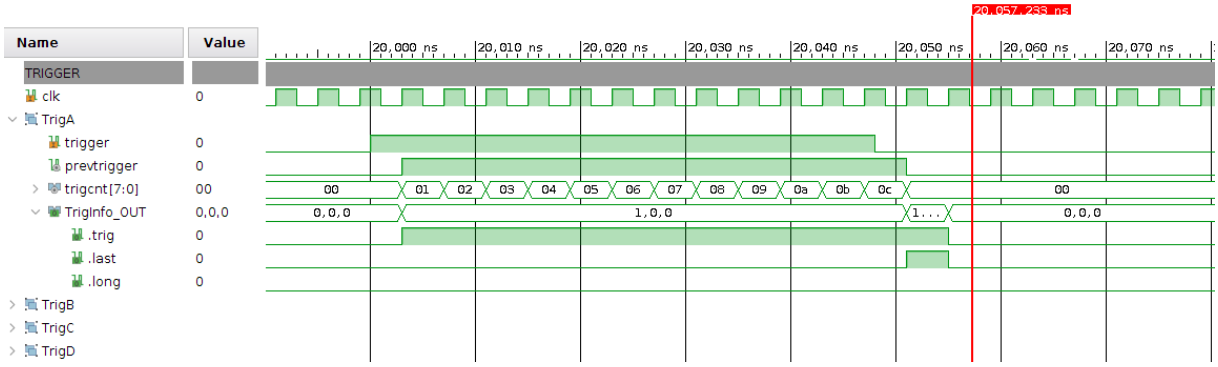
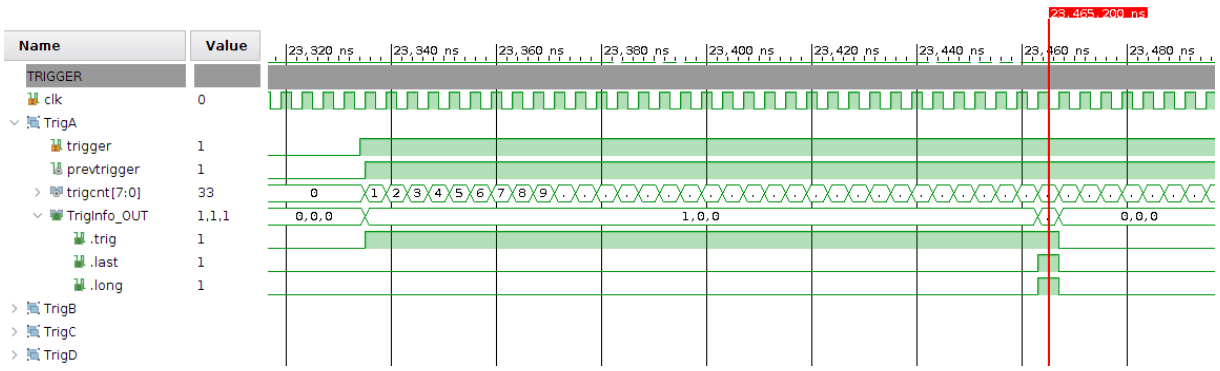


Figure 5.10: General overview of trigger system



(a) Normal trigger



(b) Trigger A is too long

Figure 5.11: Sim : Trigger A normal and too long pulse

The trigger information is collected with the write signals and the process defines which command it has to send. The package *WINDOW_PKG.vhdl* contains all information related to data types and records used.

Code 5.3: Window_pkg.vhdl CPUBus definitions

```
1  -- Command used to interface CPUs and store unit
2  constant CMD_NOP :          std_logic_vector(2 downto 0) := "100";
3  constant CMD_WR1_MARKED :   std_logic_vector(2 downto 0) := "010";
4  constant CMD_WR2_MARKED :   std_logic_vector(2 downto 0) := "001";
5  constant CMD_BOTH_MARKED :  std_logic_vector(2 downto 0) := "000";
6  constant CMD_WR1_EN_DIS :   std_logic_vector(2 downto 0) := "101";
7  constant CMD_WR2_EN_DIS :   std_logic_vector(2 downto 0) := "110";
8
9  -- Type for CPUBus 11 bits bus wide
10 type t_CommandBus is record
11     cmd :   std_logic_vector(2 downto 0); --CPUBUS(10 downto 8)     <= cmd
12     addr:   std_logic_vector(7 downto 0); --CPUBUS(7 downto 0)      <= addr
13 end record;
```

In this simulation, the internal trigger signal (Trig_intl) goes HIGH on the current window 95_{16}. The information from the single trigger block (TrigInfo_intl) comes out of the delay block on TrigInfo_intl_dly, it can be observed that the signal is stretched across a full SSTIN clock cycle marked by the change in window. The trigger bus is updated, all CPUs and store unit receive the new command and update one clock cycle later (4ns for 250MHz) after window change. The bus is by default 400_{16}, which means the command is CMD_NOP (nothing will happen) and for debugging reason the address for no command is set to 0_{16}. The command sent to the store unit is saved into the aFIFO (Asynchronous FIFO) and is read back by the testbench (RDAD_DataOut_obs). The commands are saved into a report file.

```
RDAD_DataOut_obs: 149 WR_ADDR: 149
WR1: '0'
WR2: '0'
RDAD_DataOut_obs: 662 WR_ADDR: 150
WR1: '0'
WR2: '1'
```

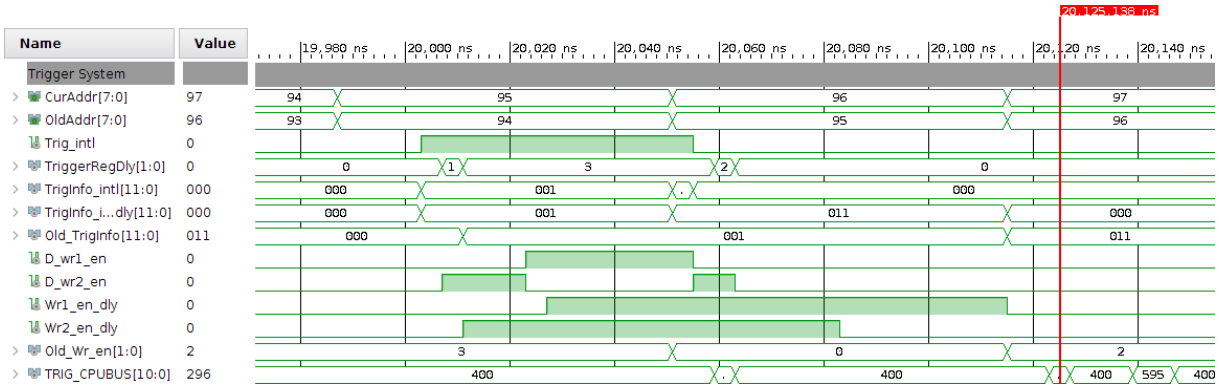


Figure 5.12: Sim : Trigger information and command generation

Near the time of sample 0, if the internal trigger rises from LOW to HIGH, because of the delay in the trigger line, the system knows the trigger definitively happened in the previous window. It is very important to distinguish these events, because they are the main reasons why the

data acquired is useful or not. Like the delays are not well known, most of the trigger system is generically mapped to easily run different configuration. In reference to figure 5.13, The PMT pulse is right on the edge of changing storage address, sampling only the address 0 will result on an incomplete trigger signal and the feature extraction will not have been performed correctly.

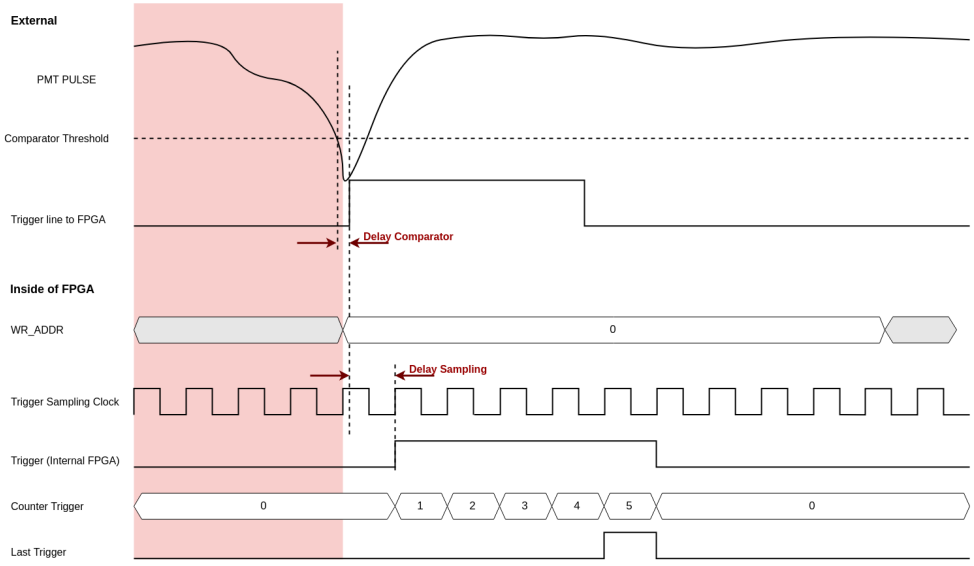


Figure 5.13: Delay issue with internal trigger signal

The delay between the real trigger event and the internal signal can be calculated. This delay will indicate after how many sample count the control unit must also digitize the previous window. The simulation shows the internal trigger signal and a leading edge detection behavior, the min delay parameter is set to 3 sample count for a sampling clock of 250 MHz, the previous window is still sampled if the min delay of 12 ns is not met. The TriggerRegDly line will trigger the leading edge detection.

Code 5.4: Leading edge detection

```
1  if rising_edge(ClockBus.CLK250MHz) then
2      --Code detection state
3      -- 00 : no trigger
4      -- 01 : trigger's LE
5      -- 11 : trigger
6      -- 01 : trigger's TE (not evaluated, like comparator is slower than the PMT pulse)
7      TriggerRegDly <= TriggerRegDly(0) & Trig_intl;
8
9      if valid_1dly = '1' then
10         --Send the command
11         -- ....
12     else
13         -- Leading Edge detection by compararing internal signal and LE lookup table
14         if LE_intl = '1' and TriggerRegDly = "01" then
15             TRIG_CPUBUS <= CMD_WR2_MARKED & OldAddr_intl;
16         else
17             TRIG_CPUBUS <= DIGI_CPUBUS;
18         end if;
19     end if;
20 end if;
```

The simulation shows that if the condition leading edge signal is HIGH and that a leading edge is detected with the delay line, the previous window is digitized ($1B6_{16}$ is equal to write 2 enable for CPU182, window 365).

Figure 5.14: Sim : Leading edge detection simulation

5.1.4.2 User mode

The trigger system is autonomous and standalone from the user mode, they both work separately. The user (PS) is master to switch from one to another using a bit in the control register ((Register : *TC_CONTROL_REG*, Bit : *C_CPUMODE_MASK*, '1' Trigger mode, '0' User mode). This last mode is used to setup the system. The PS decides which window is of interest or needed, the test pattern generator check can be checked, the Initialization of all pedestals and development are all done in this mode. The window demand was introduced earlier in the document, please refer to section 3.3.2.2.1.

5.1.4.3 CPUBus

The CPUBus connects all CPUs and store unit to the control unit. The bus structure and command format was mentioned earlier in this chapter. The same bus is shared for the various process in the control unit, which includes the user process, the trigger process and the return of digitization process. This last process controls the asynchronous FIFO between the control unit and the Wilkinson digitization process. The data recovered from the aFIFO is a window address that was digitized and so it can enabled again.
The Bus is not always busy, very often the information is sent in one clock cycle and the remaining clocks are for not doing anything. The digitization process uses this waster time to take control of the bus.

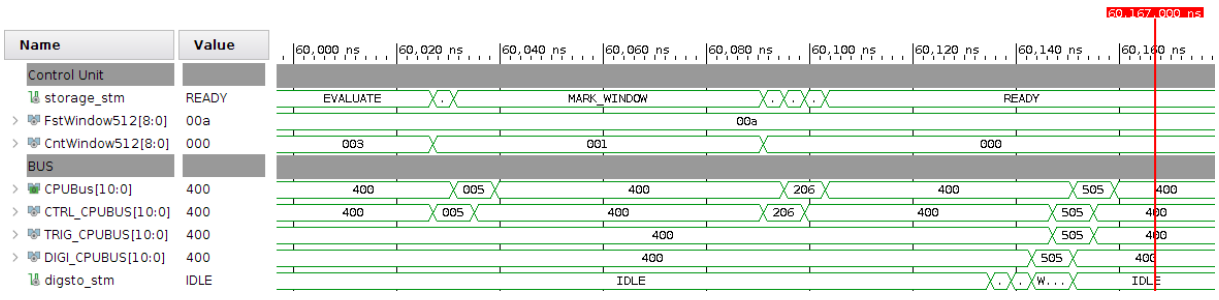


Figure 5.15: Sim : Control Bus operation

Like observed in the previous simulation run, the control unit has received a command for sampling window 10 to 12 included. This is translated by the bus going from 400_{16} (No operation) to 005_{16} (Both window for CPU5) and 206_{16} (EVEN window for CPU6). The control unit is finished and just waiting for the next clock cycles. However the return of digitization process receives an information, a window has been processed. It decodes the appropriate command and update the DIGI_CPUBUS, which can be seen on the simulation goes from 400_{16} to 505_{16} (Window 10 can be enabled again). The control unit (who is in charge, CPUMODE = 0) will send this command, if it is free and the digitization process will check the bus and stop sending this command after it sees it on the main CPUBUS.

5.2 Timing constraints

A complex and fast system will have some timing constraints violation, but here are a few techniques used to avoid some reported in the design.

5.2.1 Clock stages implementation

The next-previous logical equation takes some time to come to a solution, indeed the propagation delays inside are respectively high. Meta-stabilities are most likely possible. As a result the design was safer by using some DFF to pipeline the stages including a valid signal for confirmation after the Hamming stage. The simulation (figure 5.17) shows the different signal paths on a new address. As CurAddr changes from 3_{16} to 4_{16}, the valid signal is set to LOW, leaving time for the address lines to stabilize. This same valid signal is used like nCLR signal to reset all the next and previous buses and addresses of all the CPUs. At the same time the current CPU will start the next-previous search. CPU4 outputs a set of ones, NEXTBus and PREVBus are equal to $16 = 2^4$, the CPU's number is equal to the power of 2. These bus are logic, so some delay is to be expected, unfortunately this is not seen in the simulation. The NextAddrBit and PrevAddrBit update to $32 = 2^5$ respectively $8 = 2^3$, CPU5 and CPU3 are the next and previous addresses. These bits are propagated into the Bit Decoder, which takes 2 clocks cycles to output the correct data. Each bit decoder has two hamming decoder stages which each take one clock cycles to pass the information, a total of 4 clock cycles for propagation of the next and previous address into the chip.

NOTE : These next simulation contains the code for correcting some timing issues regarding the storage addressing of the Target C, however these changes do not alter the simulation result here.

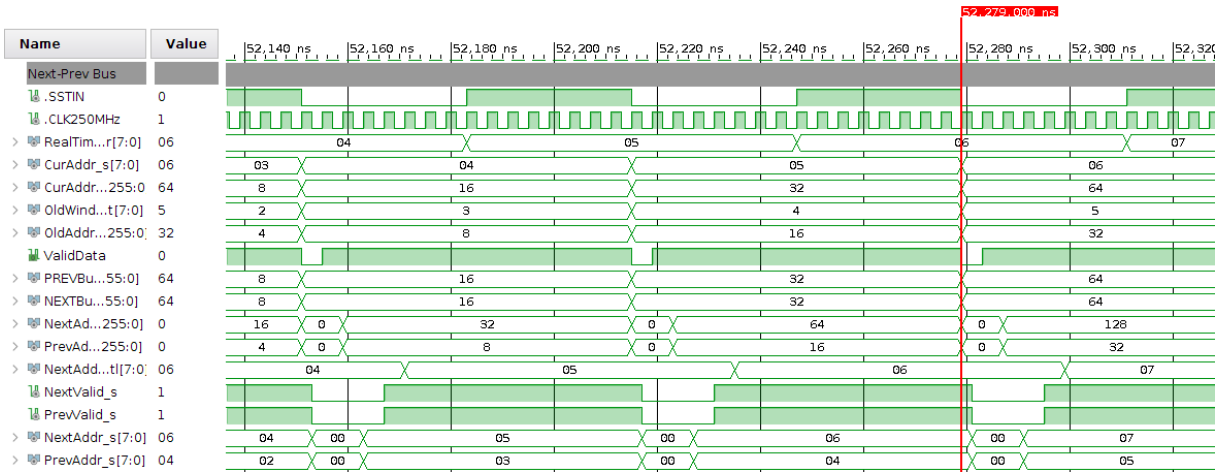


Figure 5.16: Sim : Clock stages in Round Buffer, normal case

An old command has not been processed yet and CPU0 and CPU1 are not available. At CurAddr change from FE_{16} to FF_{16}, the next and previous address are found to be 02_{16} and FF_{16} which is the correct sequence (CPU0 and CPU1 are ignored).

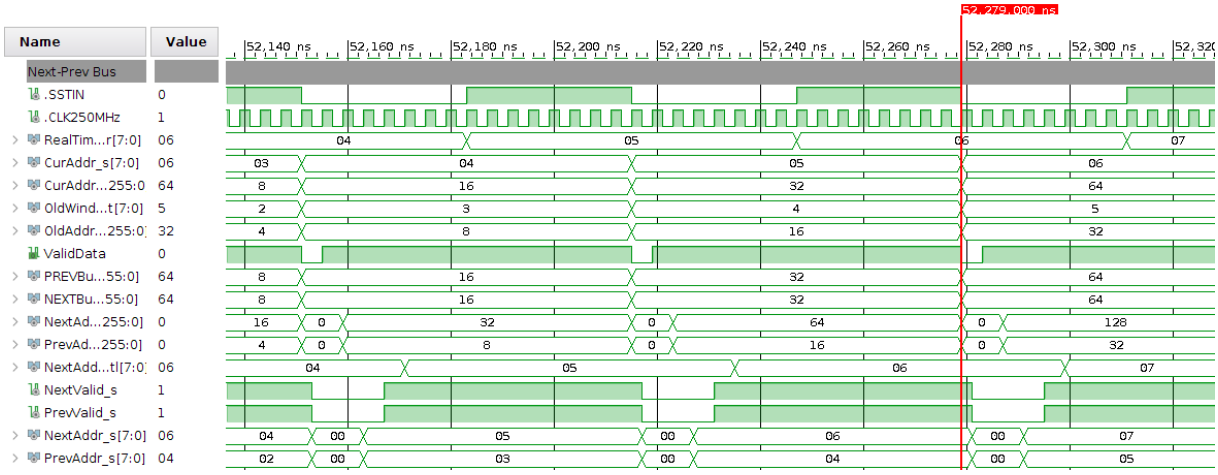


Figure 5.17: Sim : Clock stages in Round Buffer, CPU1 and CPU2 are skipped

5.2.2 Clock domain crossing

The problem which happens when data is transfered to a new clock domain, is the violation of setup or hold time. Indeed if the signal A changes to closely to the ClkB, the output of the flip flop may oscillate from some period of time before the next clock cycle. To ensure the signal is properly transmitted to the other clock domain, the signal shall be present for a least for 2 clock periods of the slower clock. The difficult path is always from the faster clock to the slower clock. A simple circuit presented in figure 5.18.

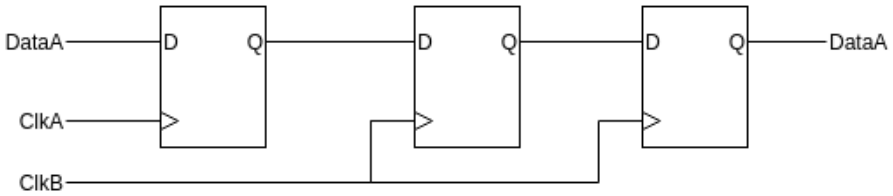


Figure 5.18: Multi fli-flop synchronizer

5.2.3 Handshake - V2.0

Like previously described the handshake is a way to synchronize two processes or systems. Their is always a sender and a receiver, the sender waits on the receiver, it checks the busy signal. Once the receiver is available, the sender can initiate a request, that the receiver must acknowledge. The sender once certain that the receiver is aware of the request, responds to the acknowledgement and so does the receiver. The figure 5.19 demonstrates the steps to establish such a communication. An important fact is the clock domain crossing, the signals must be processed accordingly, please refer to the chapter 5.2.2. This is not shown in this transaction diagram. Once both parties are synced, data can be passed from one side to another, in most cases from sender to receiver.

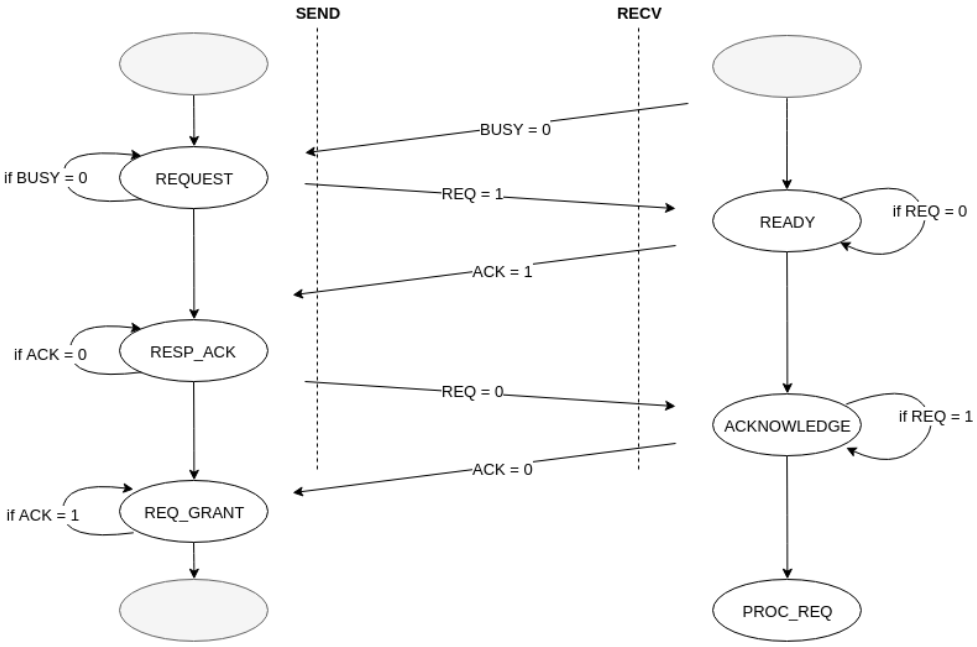


Figure 5.19: Transaction and state machine of the handshake

A package definition of the handshake elements was implemented to facilitate the handling of those signals, including clocks for the cross domain component.

Code 5.5: Type and subtype definition for the handshake

```
type T_Handshake_IxRECV is record
  REQ:    std_logic;
  RCLK:   std_logic;
end record;

type T_Handshake_OxRECV is record
  ACK:    std_logic;
  BUSY:   std_logic;
  ACLK:   std_logic;
end record;

subtype T_Handshake_IxSEND is T_Handshake_OxRECV;
subtype T_Handshake_OxSEND is T_Handshake_IxRECV;

type T_Handshake_SEND_INTL is record
  REQ:    std_logic;
end record;

type T_Handshake_RECV_INTL is record
  ACK:    std_logic;
  BUSY:   std_logic;
end record;

type T_Handshake_signal is Record
  recv : T_Handshake_RECV_INTL;
  send : T_Handshake_SEND_INTL;
end record;
```

The hardware describing this handshake uses the three DFFs solution mentioned previously. One for each critical signal (REQ,ACK and BUSY). The data line (DATA_IN and DATA_OUT) must be setup prior to the transaction, thus when the receiver or the sender samples the data line it is already stable for several clock cycles.

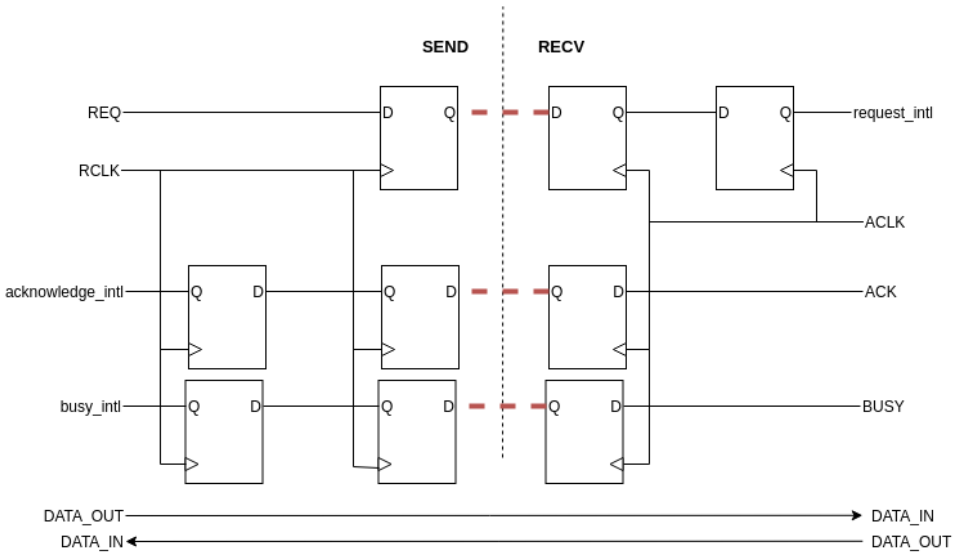


Figure 5.20: Handshake V2.0 - Logic Elements

The simulation below demonstrates the establishment of a handshake transaction between an AXI clock domain at 100MHz and the Readout-Digitization at 100MHz. Simulation shows the same clock because it is the case, however in the design they come from two different sources. The state machine is not so different from the Valid-Response handshake discussed earlier, except that this method REQ-ACK is implemented with clock crossing domain. The handshake_data line is the information needed to be sent from one end to the other.

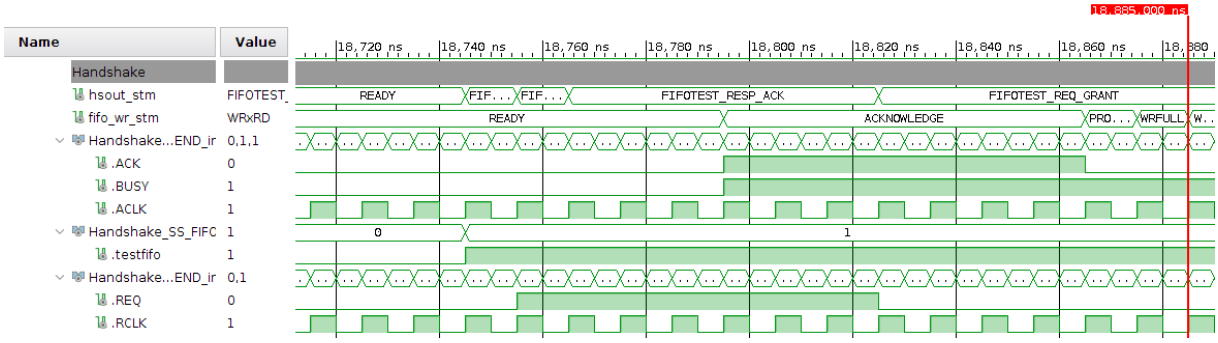


Figure 5.21: Sim : Handshake transaction for a test FIFO event

5.2.4 Asynchronous FIFO

The asynchronous FIFO is a great example of timing constraints violation. The pointers for the write and read were not synchronized for the full and empty signal, which gave place to unexpected behavior. These pointers are a 5-bit address line, the previous case for clock crossing was for a single bit, for data or address buses the problem is different. The metastability can occur at a terrible moment like the change from 11111_2 to 00000_2. In this case all bits are in a metastable state and can drift to HIGH or LOW, meaning on the next clock cycle the address should have been 11111_2 or 00000_2, but instead it is actual an erroneous value 01101_2.

A few mechanism exist to avoid such bus problem, the uses of MUX recirculation technique [6], FIFOs or handshakes (already implemented) and Gray Code encoding-decoding technique [7]. This last technique is the one used to customize my aFIFO component.

5.2.4.1 Gray Code

The gray code has the ability to count up and down by only changing 1 bit at a time. The address plays on this property, indeed the address will now only changing 1 bit at the time, the metastability concerns again only one bit and not the entire bus anymore.

Decimal	Binary	Gray
0	0000	0000
1	0001	0001
2	0010	0011
3	0011	0010
4	0100	0110
5	0101	0111
6	0110	0101
7	0111	0100
8	1000	1100
9	1001	1101
10	1010	1111
11	1011	1110
12	1100	1010
13	1101	1011
14	1110	1001
15	1111	1000

Table 5.1: 4 bits Binaray to Gray code conversion

(a) 4 bits Gray code encoder (b) 4 bits Gray code decoder

Figure 5.22: Gray Code using XOR logic gates

Code 5.6: VHDL Gray Code Encoding

```
1 GRAY_OUT(NBITS-1) <= BIN_IN(NBITS-1);
2
3 GEN_XOR : for I in 0 to NBITS-2 generate
4     GRAY_OUT(I) <= BIN_IN(I+1) xor BIN_IN(I);
5 end generate;
```

Code 5.7: VHDL Gray Code Encoding

```
1 XOR_intl(NBITS-1) <= GRAY_IN(NBITS-1);
2
3 GEN_XOR : for I in 0 to NBITS-2 generate
4     XOR_intl(I) <= GRAY_IN(I) xor XOR_intl(I+1);
5 end generate;
6
7 BIN_OUT <= XOR_intl;
```

The aFIFO is fitted with the new elements described earlier and the empty and full signals are adjusted to their right clock domain. The design interfacing the aFIFO is untouched, because only one command can arrive per 64 ns, this excluded the need to wait 2 cycles to check the full and empty signals before writing the next value.

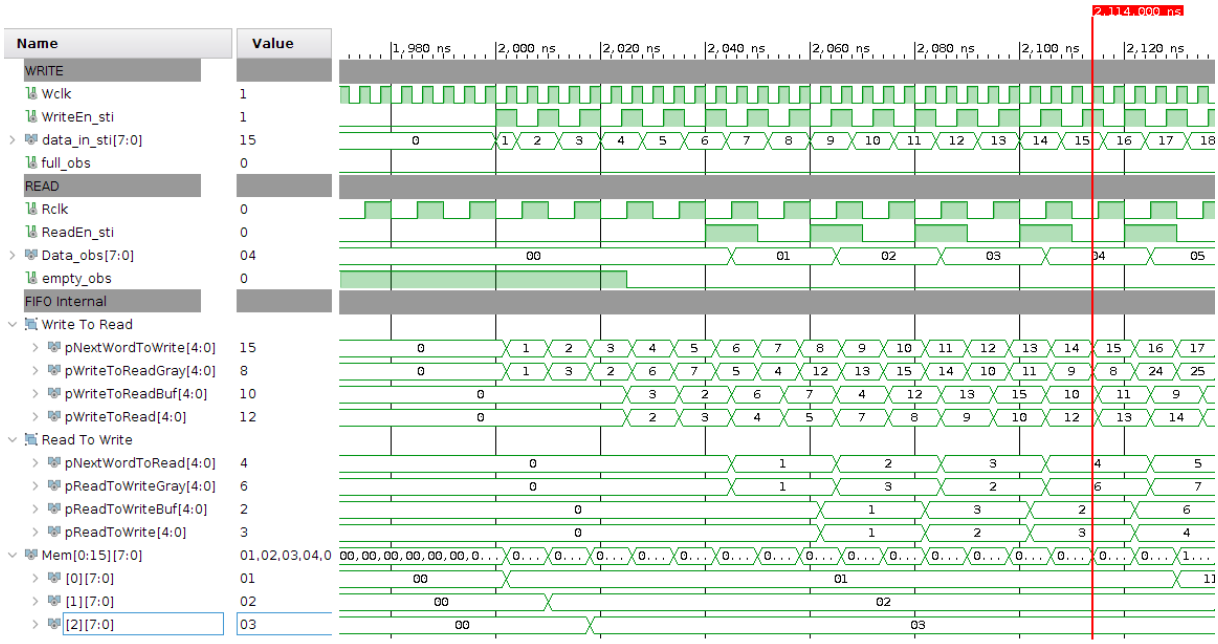


Figure 5.23: Sim : Asynchronous FIFO with clock domain and Gray Code

5.2.4.2 Limiting data width

Limiting the width of the data saved into the aFIFO greatly improves the timings, the registers are closer together and not stretch across the FPGA, their paths are therefore relatively equal. The big aFIFO containing all information and shifted to every component from the earlier design is replaced to 5 smaller asynchronous FIFOs, like shown in figure 5.24.

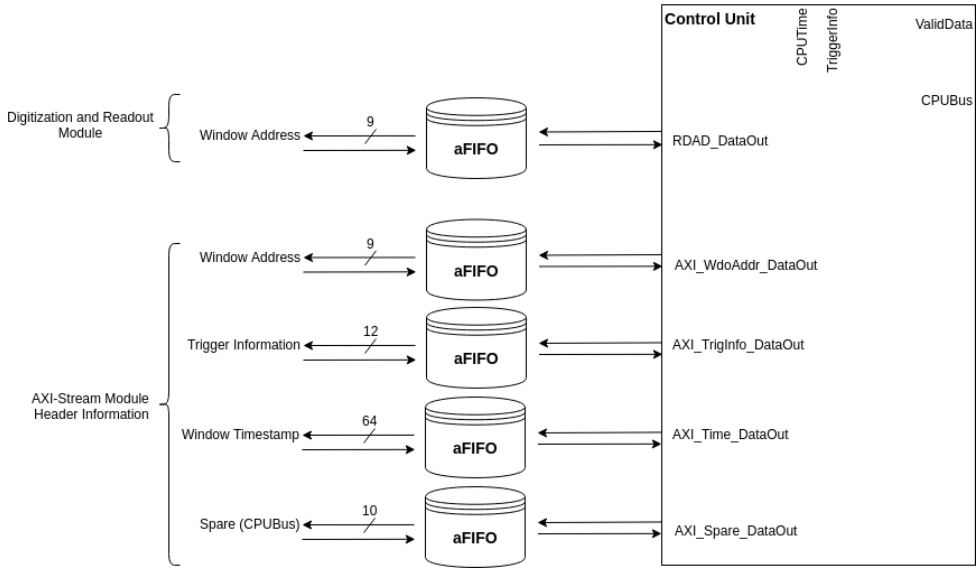


Figure 5.24: Asynchronous FIFOs for the different modules

5.2.5 Gray Time counter and binary sample count

Timing constraint were met using a binary counter for the time stamp information. The delay to store the counter in the aFIFO was too long. This counter serves to identify the sampling time of the window. This counter also serves for the construction of the SSTIN clock and the sample count. The sampling time is not relevant for the VHDL part and this time changes at the SSTIN frequency. Like for the aFIFO, to avoid metastabilities in wide data buses, gray encoding is a perfect solution. The time counter is divided into the sample count 4 LSB bits and the real time of sampling 60 MSB bits. Those last ones are gray encoded and transfered to the CPU Controller, also known as control unit for the round buffer.

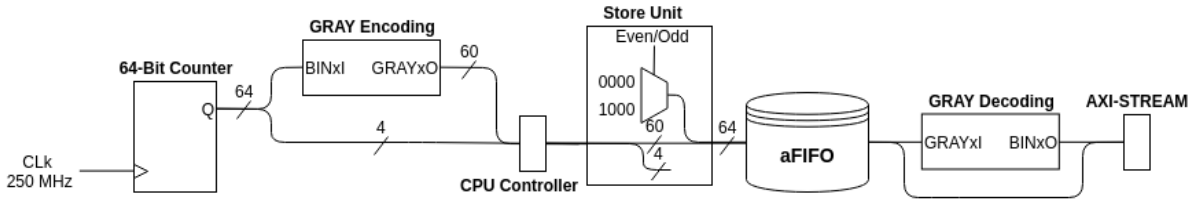


Figure 5.25: 64-Bit Gray time path across the FPGA

The store unit saves the 60-bit gray encoded timestamp concatenated with a 4-bit code. An EVEN window will have the code 0000_2 and an ODD window, the code 1000_2, both represent the least significant bits of the binary timestamp. Once a transfer is ready, the FIFO manager reads the stored 64 bits, it decodes the time into a binary value, concatenates the 4-bit code and the result value is the actual time at which the window was sampled.

5.3 Target C - Timings

After updating and adjusting the timings constraint, the firmware had reached its maximum in terms of optimization and enhancements. Unfortunately, addressing issues were noticed after reading out some consecutive windows, measure in figure 5.26.

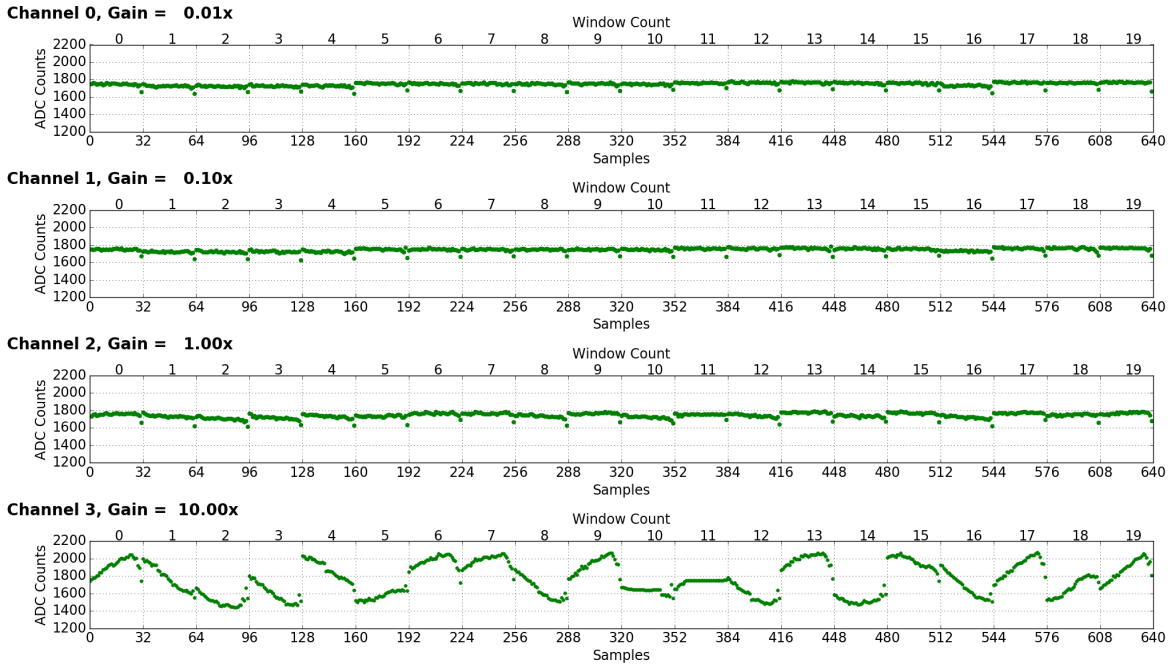


Figure 5.26: Measure of 20 consecutive windows, input voltage is 10MHz, 50mVpp and Vped = 1.25V

5.3.1 Storage Address update

The previous understanding of the storage address had been wrong. The corrected way is illustrated in the figure 2.11. The real address is the current window being sampled but is only valid a small amount of time after the sampling process due to the capacitor. Once the capacitor is stabilized the rising edge on the write address sync will latch the address on the storage memory to select the correct storage cell. The write strobe signal is level sensitive, a HIGH level will save the voltage from sampling to storage cells.

The change of address must be implemented in the VHDL. The change of address is done at the falling edge of SSTIN and no more on the rising edge. The principle of CurAddr and OldAddr is modified to fix this issue. The figure 5.28 illustrates the new architecture of the round buffer.

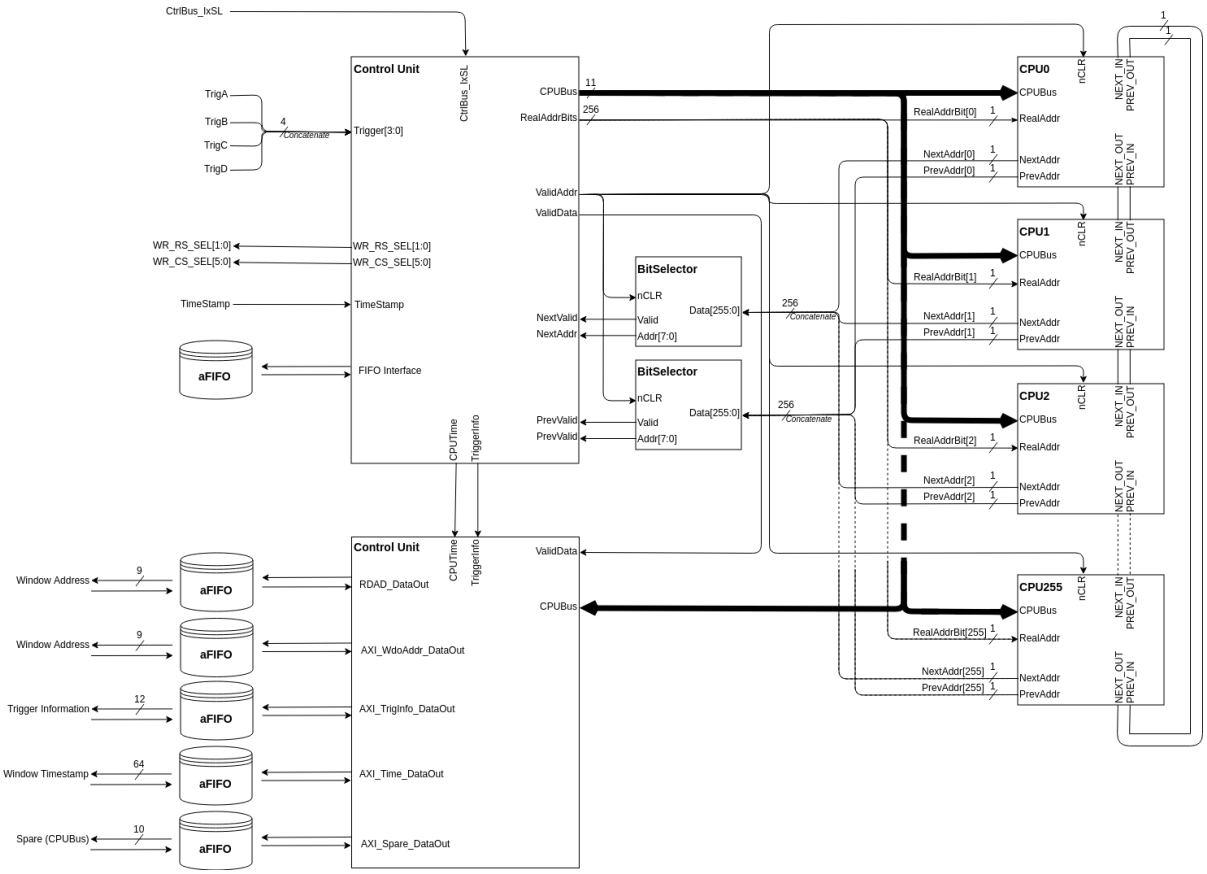


Figure 5.27: Updated version of the overall system

The simulation of the system, reacts as planned. Some signals like the old address are no longer needed outside of the CPU controller. After synthesize and implementation, the timing constraint are still good. The system reacts correctly to pedestals subtraction, FIFO Test, Test Pattern generator and consecutive windows readout. However the data is still corrupted.

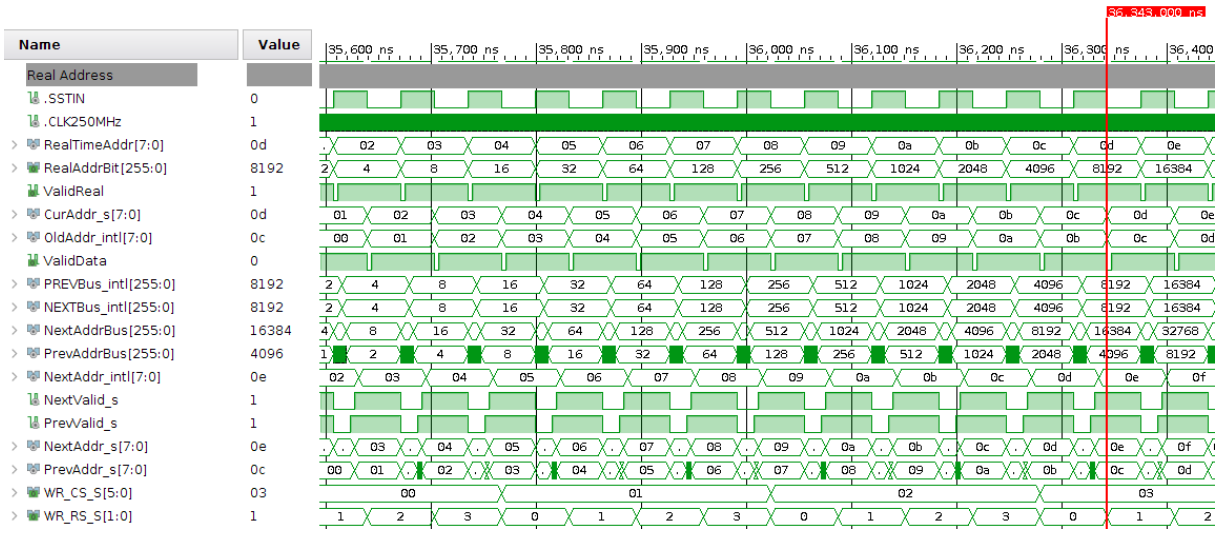


Figure 5.28: Sim : Real Address implementation

5.3.2 Target C Timing Generator

No actual paper explains how to configure the signals of the timing generator inside the ASIC, which is the heart of the system. Without it the device is not giving the right signals for sampling and storage. The previous measures were performed on a stable DC value, so shifts in sampling time or window time misalignment could not be seen during those measures.

5.3.2.1 Circuit Model

To fully understand the timing generator's functionality, it has been mobilized in a VHDL testbench by using the components on the schematics.

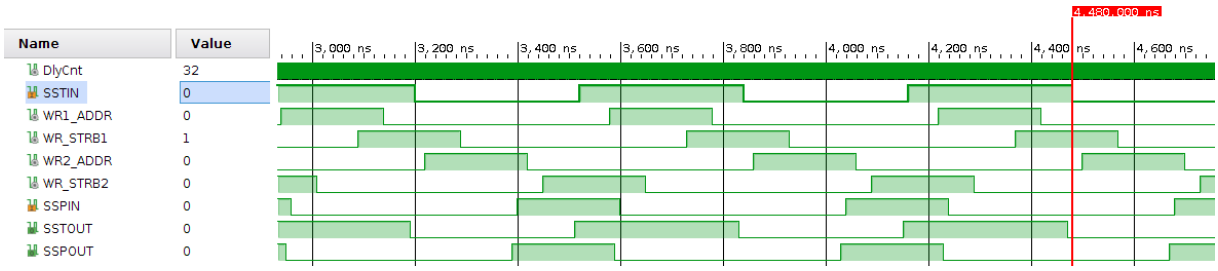


Figure 5.29: Sim : TARGETC Timing Generator (TC_TimingGen)

Unfortunately the signals are very small, in addition to that the simulation requires Vivado running using finally lots of resources just for checking our signals. A small python script was written to control the signals behavior, see figure 5.30.

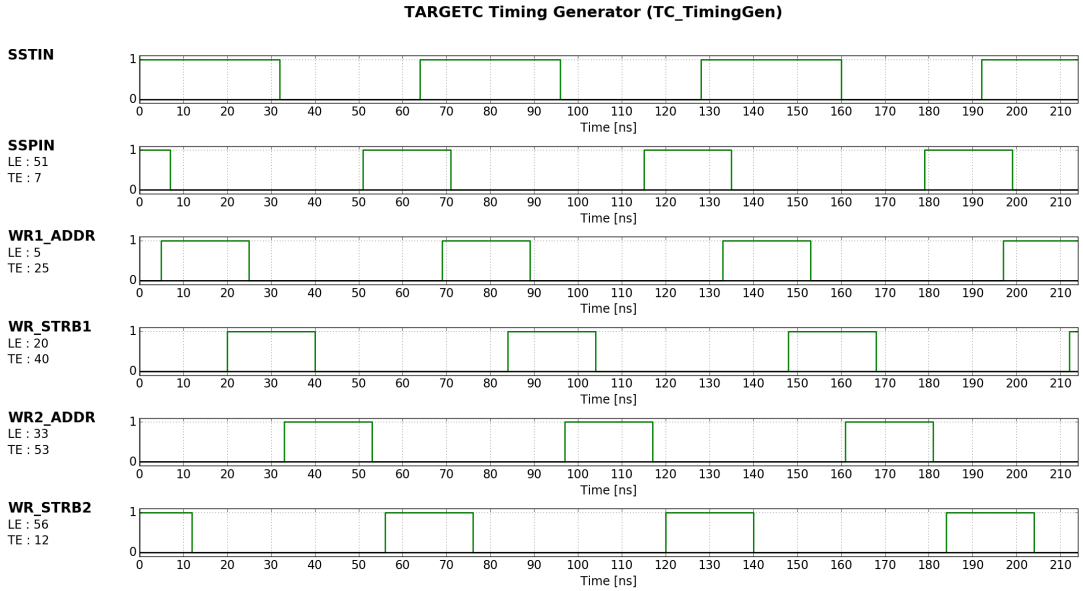


Figure 5.30: Sim : TARGETC Timing Generator (TC_TimingGen) with TARGETX timing parameters

The values tested are the ones from previous generation of the TARGETs, they should not work according to the setup of the signals discussed earlier. After a a few tests the prototyped value are entered in the simulator, the figure 5.31 shows the intended behavior of the system.

Parameter	Value
SSTOUFB	58
SSPIN_LE	51
SSPIN_TE	7
WR_STRB2_LE	56
WR_STRB2_TE	12
WR2_ADDR_LE	33
WR2_ADDR_TE	53
WR_STRB1_LE	20
WR_STRB1_TE	40
WR1_ADDR_LE	5
WR1_ADDR_TE	25

Table 5.2: TARGETX, timing values

Parameter	Value
SSTOUFB	58
SSPIN_LE	51
SSPIN_TE	7
WR_STRB2_LE	25
WR_STRB2_TE	35
WR2_ADDR_LE	5
WR2_ADDR_TE	15
WR_STRB1_LE	57
WR_STRB1_TE	3
WR1_ADDR_LE	37
WR1_ADDR_TE	47

Table 5.3: TARGETC, prototyped values

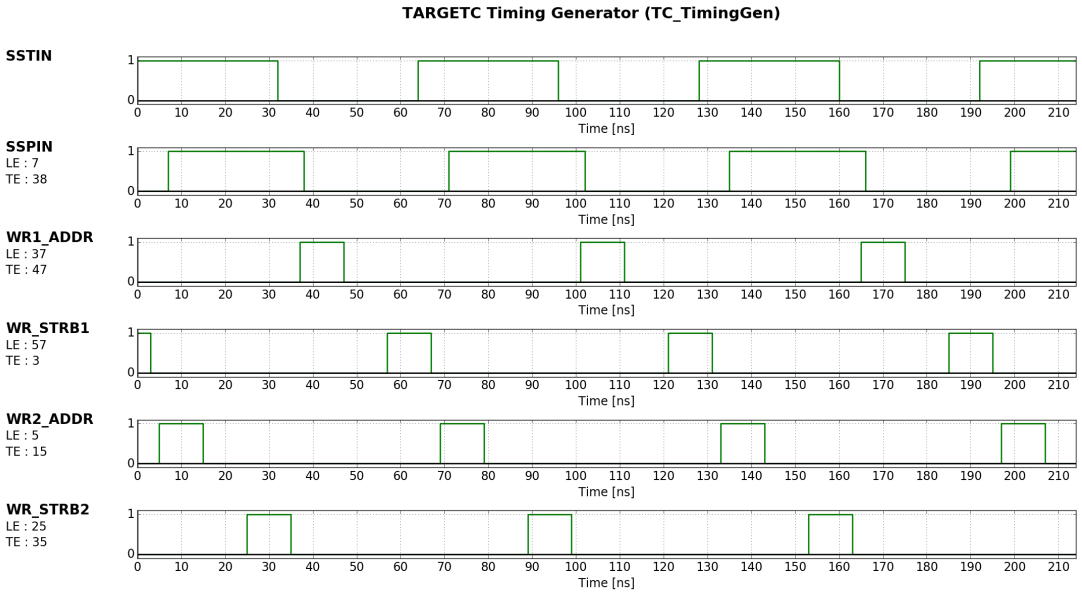


Figure 5.31: Sim : TARGETC Timing Generator (TC_TimingGen) with TARGETC prototyped values

5.3.2.2 Hardware test and verifcation

The ASIC is then tested and the internal signals are probed using the MonTiming LVDS signal. Theses pins can be configured using the register interface to output different internal signals, see table 5.4.

Bits	Function	R/W	Default
11 - 8	(unused)	-	-
7 - 4	Monitor Timing selection on MonTiming LVDS output	W	0000_2
	0000_2 : SSPout		
	0001_2 : SSTout		
	0010_2 : SSToutFB		
	0011_2 : SSPIN		
	0100_2 : WR_STRB1		
	0101_2 : WR1_ADDR_INCR		
	0110_2 : WR_STRB2		
	0111_2 : WR2_ADDR_INCR		
	$1xxx_2$: VDD (Tied to power supply)		
3	(unused)	-	-
2	Cload is for adding an additional capacitor for low pass filter	W	0
1 - 0	(unused)	-	-

Table 5.4: Reg 76 - MonTiming - 0x4C

The signals are sent to the FPGA into a LVDS buffer and come back out on a single line. Each signal is saved into a CVS file and processed using a python script to plot all signals in the following figure. The first impression is that all signals do not correspond to the values entered. However there is a coherence between the falling edges of all signal, the difference between the falling edges of WR1_ADDR and WR_STRB1 is equal to the difference of their LE parameters. These signals have a long path to the MonTiming pins, some delay is to be expected. Overall the signals are coherent with the parameter, but they are all inverted. The test to invert them is ran, but a pulse wider than 32 ns is not possible, the MonTiming signal goes directly HIGH. So the only way is to run with negative pulses, which is odd because the ASIC's logic is intended for HIGH logic.

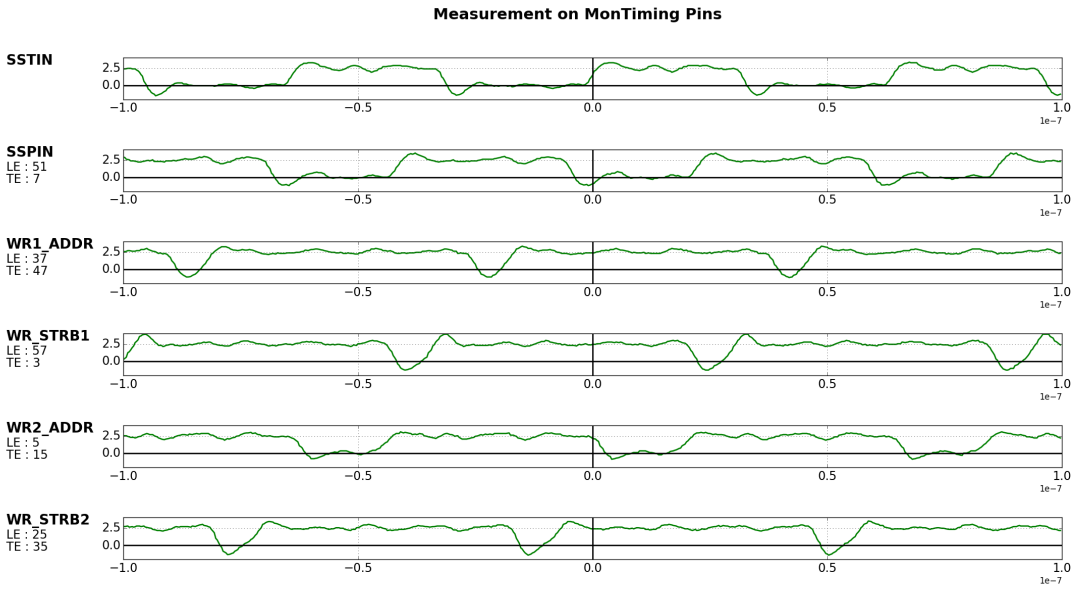


Figure 5.32: TARGETC Timing signals readout from MonTiming

Apparently there is an incoherence between schematic and layout of the ASIC, the MonTiming signal will be flipped at the LVDS pair (P is N and N is P), which will explain why all signals were inverted. Some tests are relaunched and the data acquired is still in the same as before, no real improvement is noticeable. Some windows would have some sine wave components, but then some others reflect overwrite, like the WR_STRB signals are set too late from the change in address, for some more investigation is required.

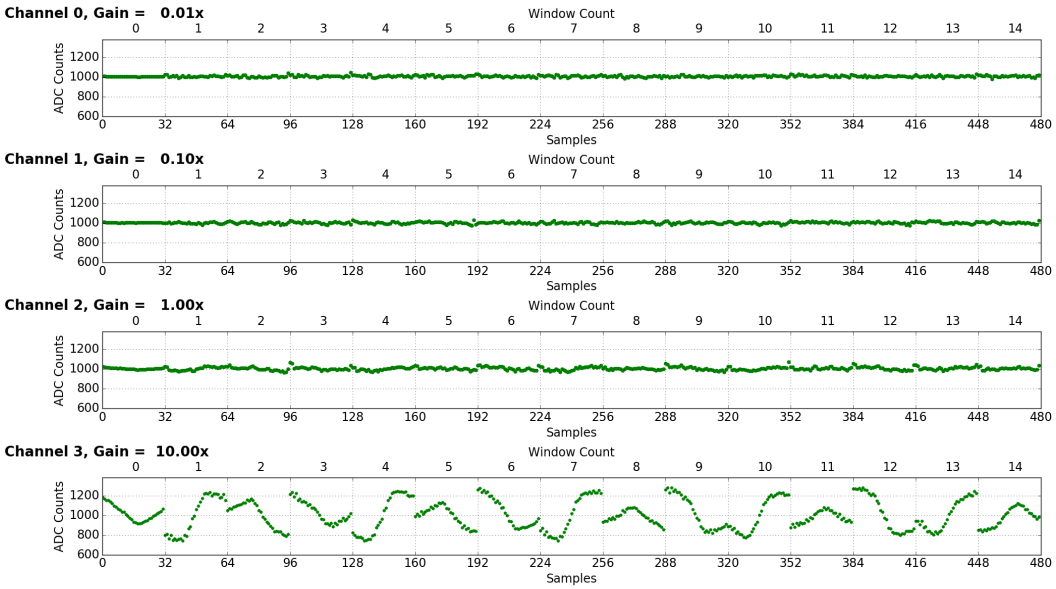


Figure 5.33: Measure of 15 consecutive windows, input voltage is 10MHz, 50mVpp and Vped = 1.25V

6 Conclusion and recommendations

6.1 Project

Most of the time, the small details which leads the correct or incorrect data readout are written in small characters in the product specifications. Nonetheless this information is written somewhere, we struggle to find it at first but we are glad to that it explains all the 99% of the issue. In this project, the documentation is clearly lacking. The TARGETC has no technical specification or manual on how to interface it. Previous designs do not share the same register map and the internal architecture is unknown. In addition the simulation files are inexistent, which would have been useful to analyze the internal behavior to input response. To sum up, any documentation is good documentation. Generally within this project, a lot of time was wrongly spent looking for information rather than building the interface between the ASIC and the FPGA.

Despite the above mentioned drawbacks regarding the project preparation, the new defined goals were met, namely , the full acquisition chain is working, the round buffer is capable of dealing with multiple window readout. The trigger system under simulated parameters, the whole process shows good responses. The late issue with storage address (SSTIN Update) was not fixed for the trigger system. Even though the data acquired is not consecutive yet, the overall design is implemented and working. It is a young design, with probably many unnecessary loops, states or signals, which could be improved in the future. From optimization perspectives, it is close to the optimal performance. Due to lack of time, the performance have not been tested yet, this is to say the fine tuning in frequency speed for the Wilkinson conversion mapped with fast charge to reduce the conversion time by 4.

Future perspectives are the investigation of the timing signals. Also between the schematic and layout there is a swap in signals, it is possible the storage address or internal write signals are experiencing the same behavior. From the firmware point of view, all the whole WATCHMAN TARGETC project was based on acquiring data from the ASIC, upon arrival to HAWAII no firmware was done. The TARGETC had never been touched, in order to get as fast as possible some data acquisition, the firmware was built up really quickly. Since it was prepared in a rush, the firmware is very messy and some components could be re-build/re-thought to be more efficient, ie the register design of the AXI-Lite interfacing for updating the Target C could be greatly improved. An over-watch module is also a nice feature, it could determine if the PL side is reacting correctly and raises an error if this is not the case. The internal control buses used here are very basic, some more advanced methods could be used. The cache coherence is maybe a problem as it seems, the switch to ACP port is a possible solution. The trigger system needs some few changes.

Overall, the system is in good status for a prototype version, the interfaces to send and read data from TARGETC are correct and working as can be seen in the different simulation (TPG, consecutive window readout, etc).

6.2 Objectives

Taking a second to look at my objectives, my primary ones have all been validated from the work presented in this report.

- ☑ Development of the data interface between the PS and PL (AXI-Lite and AXI-Stream).
- ☑ Data readout from TARGET C
- ☑ Elaborate the trigger system
- ☑ Round Buffer algorithm to deal with new events and previous event still being digitized

The secondary objectives were not all fulfilled, as the data readout is not consistent with our expectations. This issue is a technical and component based problem not related to the current status of firmware.

- ☑ Setting up an AXI-Lite Register Module
- ☑ VHDL Test component for AXI-Stream
- ☑ VHDL Project for AXI-Stream DMA with PL-PS
- ☐ Rough understanding of Target C ASIC
- ☐ Elaborate Datasheet for Target C ASIC
- ☑ Register Communication with Target C
- ☐ Sampling and storage of data with TARGETC
- ☑ Wilkinson digitization process
- ☐ Readout Data from Target C
- ☑ Round Buffer elaboration in VHDL
- ☑ Trigger system based on trigger information
- ☑ Analyze the received data

6.3 Personal

As personal conclusion, I was thrilled to build firmware again, getting hands on the Zynq with an unknown device was a real challenge. Most of all the round buffer concept was like having a new toy to play with. I really enjoyed putting the hard work into developing the new features unseen till now. Personally, one of my objectives was to use System Verilog to verify my system and use formal verification, due to the rush this personal objective was not meet. Overall, I am pleased and wished there was more time to investigate the last few problems.

Honolulu, 8th of February 2019
Jonathan Hendriks

List of Figures

Bibliography

[1] UC Davis Neutrino Group, *WATCHMAN*
UC DAVIS NEUTRINO GROUP. DEPARTMENT OF PHYSICS
Link : `http://svoboda.ucdavis.edu/experiments/watchman/`

[2] AXI DMA v7.1, *LogiCORE IP Product Guide*
Vivado Design Suite PG021, April 4, 2018

[3] XILINX axidma, *Xilinx SDK Drivers API Documentation*
Xilinx GitHub
Link : `https://xilinx.github.io/embeddedsw.github.io/axidma/doc/html/api/index.html`

[4] 7 Series FPGAs Clocking Resources, *User Guide*
UG472 (v1.14), July 30, 2018
Link: `https://www.xilinx.com/support/documentation/user_guides/ug472_7Series_Clocking.pdf`

[5] FPGAdevloper, *Using the AXI DMA in Vivado*
Jeff Johnson, Aug 6 ,2014
Link : `http://www.fpgadeveloper.com/2014/08/using-the-axi-dma-in-vivado.html`

[6] Link : `https://www.eetimes.com/document.asp?doc_id=1276114`

[7] Link : `https://zipcpu.com/blog/2018/07/06/afifo.html`

[8] Xilinx AXI , *Reference Guide*
UG761 (v13.1) March 7, 2011

MSE

Hes·so
Haute Ecole Spécialisée de Suisse occidentale
Fachhochschule Westschweiz
University of Applied Sciences and Arts Western Switzerland

Master of Science HES-SO in Engineering

Orientation : Technologies industrielles (TIN)

Appendices

UNIVERSITY of HAWAI'I®
MĀNOA

WATCHMAN

at the University of Hawaii

Honolulu, University of Hawai'i, February 2019

Contents

A Preliminary Work

This chapter covers the basic firmware and software for communicating between the PS and PL of the Zynq. This includes the AXI4-Lite interface and the AXI4-Stream with the AXI DMA Core and the first project on Vivado for the development of a dummy test component.

A.1 AXI Bus Type

AXI4-Lite

AXI4-Lite is a light-weight, single transaction memory mapped interface. It has a small logic footprint and is a simple interface to work with both in design and usage.[**?**]

AXI4-Stream

AXI4-Stream allows unlimited data burst size write sequence. AXI4-Stream interfaces are very light because it is a Master to Slave connection.[**?**]

A.2 Register Handler through AXI4-Lite

Xilinx tool Vivado©Design Suite proposes an IP Core configurable for Slave or Master mode with the choice between different AXI4 interfaces. By default the Vivado tool does not write the registers as an array but multiple signals for each register.

Code A.1: Default register generation

```
1  signal slv_reg0 :std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
2  signal slv_reg1 :std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
3  signal slv_reg2 :std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
4  signal slv_reg3 :std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
5  signal slv_reg4 :std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
6  signal slv_reg5 :std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
7  signal slv_reg6 :std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
8  signal slv_reg7 :std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
9  signal slv_reg8 :std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
10 signal slv_reg9 :std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
```

Code A.2: Array optimization

```
1  type slv_array is array (integer range <>) of std_logic_vector(C_S_AXI_DATA_WIDTH-1 ↩
       downto 0);
2  signal reg: slv_array(0 to 9);
```

In Vivado, an address is associated to the device which is the base address used by any device connected on the AXI bus to interact with this new component.

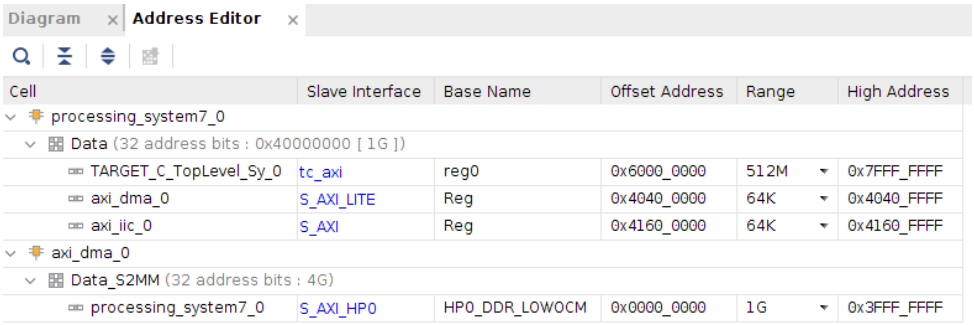


Figure A.1: Address editor window in Vivado

From the hardware definition, SDK will generate a Board support package (BSP), with a file called xparameters.h. This file contains the exact same address for the PS side to interact with device.

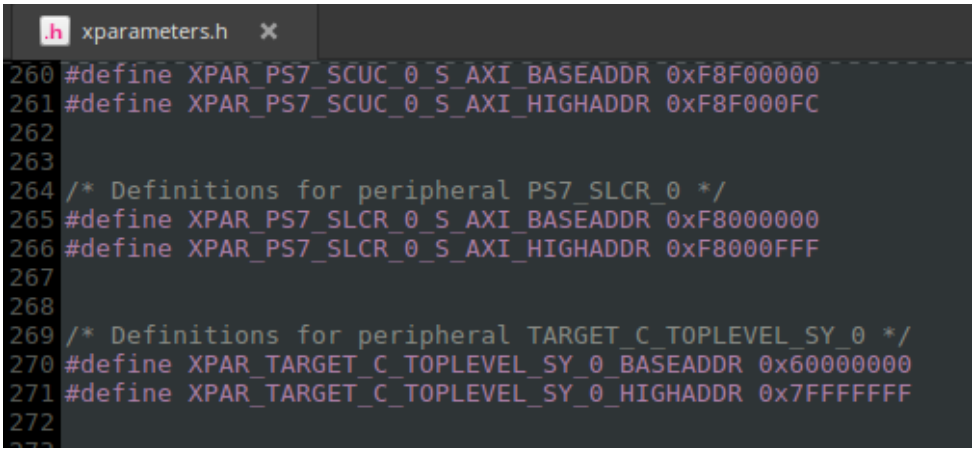


Figure A.2: xparameters.h file for SDK linking AXI Base addresses of components to the hardware

A.3 Data Transfer using AXI4-Stream

In order to acquire the data from the TARGETC, a fast type of data transfer had to be implemented. The idea is to use the AXI-DMA Core from Xilinx to transfer the data directly into the DDR at a specified memory location. Only then, is the software notified that a transfer is completed. It can readout the memory at this point.

A test component had to be written to transfer some dummy data to the IP core. This small RTL component will enable to verify the bandwidth and transfer capabilities of the AXI-Stream bus, to see how well this bus reacts to the changes of number of data to be transmitted.

A.3.1 AXI-DMA IP Core

DMA is short for Direct Memory Access, this IP enables the user to transfer data from memory to PL and from PL to memory. Various tutorials help to setup correctly the AXI-DMA, this

particular one explains the connections interactions with the different IPs (see [5]). The AXI-DMA has two modes of operation the simple mode and scatter and gather. For the purpose of this project, the simple mode is more than suitable. The address is sent through AXI-Lite interface to the AXI-DMA along with the number of data to be sent.

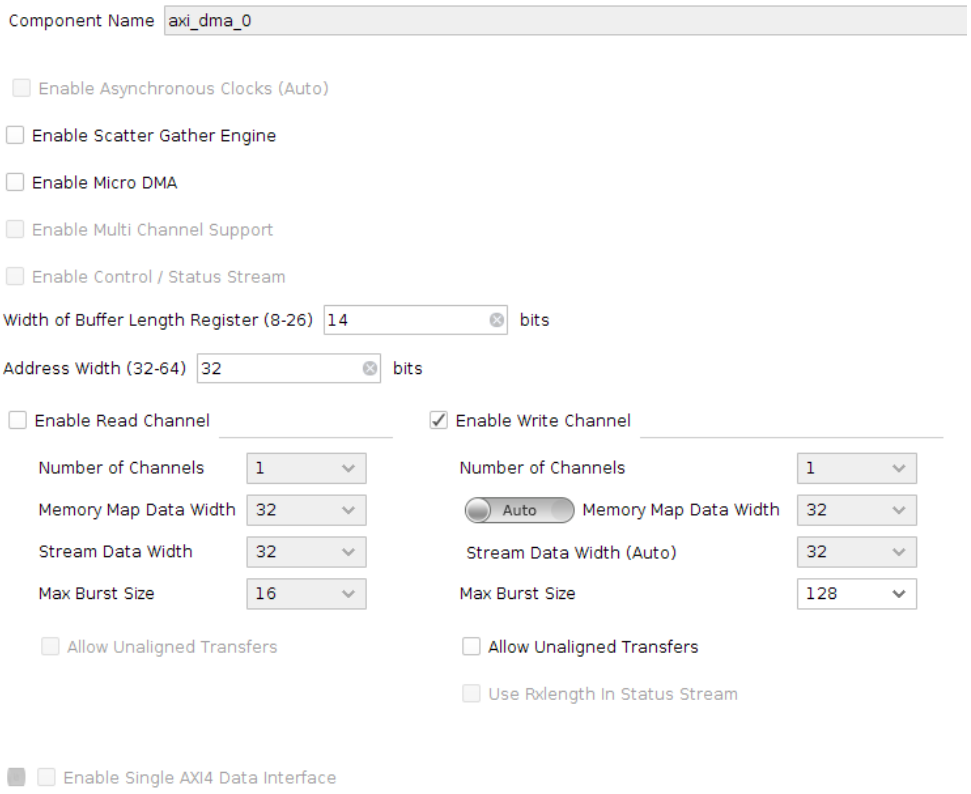


Figure A.3: AXI-DMA Vivado configuration

A.3.2 Test Component

The test component was for testing the AXI-Stream interface and provide different signals that could be read from the PS.

```
DMA Test - Multiple Packets
Reset XAxiDma Done
Packet N1 :     Nbr:10  Content:100     Mode:1
>> RxDone Interrupt Caught
        Verifying...SUCCESS
Output took 298 clock cycles.
Output took 0.45 us.
Bandwidth:715.17 Mbit/s
Packet :        Nbr:10  Content:100     Mode:1
118AD8  PtrData[0]      = 0      (= 0)
118ADC  PtrData[1]      = 1      (= 1)
118AE0  PtrData[2]      = 2      (= 2)
118AE4  PtrData[3]      = 3      (= 3)
118AE8  PtrData[4]      = 4      (= 4)
118AEC  PtrData[5]      = 0      (= 0)
118AF0  PtrData[6]      = 1      (= 1)
118AF4  PtrData[7]      = 2      (= 2)
118AF8  PtrData[8]      = 3      (= 3)
118AFC  PtrData[9]      = 4      (= 4)

Packet N2 :     Nbr:20  Content:200     Mode:2
...
>> RxDone Interrupt Caught
        Verifying...SUCCESS
Output took 784 clock cycles.
Output took 1.18 us.
Bandwidth:543.67 Mbit/s
Packet :        Nbr:20  Content:200     Mode:2
118FD8  PtrData[0]      = 0      (= 0)
118FDC  PtrData[1]      = 1      (= 1)
118FE0  PtrData[2]      = 0      (= 0)
118FE4  PtrData[3]      = 1      (= 1)
118FE8  PtrData[4]      = 0      (= 0)
118FEC  PtrData[5]      = 1      (= 1)
118FF0  PtrData[6]      = 0      (= 0)
118FF4  PtrData[7]      = 1      (= 1)
118FF8  PtrData[8]      = 0      (= 0)
118FFC  PtrData[9]      = 1      (= 1)
119000  PtrData[10]     = 0      (= 0)
119004  PtrData[11]     = 1      (= 1)
119008  PtrData[12]     = 0      (= 0)
11900C  PtrData[13]     = 1      (= 1)
119010  PtrData[14]     = 0      (= 0)
119014  PtrData[15]     = 1      (= 1)
119018  PtrData[16]     = 0      (= 0)
11901C  PtrData[17]     = 1      (= 1)
119020  PtrData[18]     = 0      (= 0)
119024  PtrData[19]     = 1      (= 1)

Packet N3 :     Nbr:30  Content:300     Mode:0
...
>> RxDone Interrupt Caught
        Verifying...SUCCESS
Output took 1086 clock cycles.
Output took 1.63 us.
Bandwidth:588.73 Mbit/s
Packet :        Nbr:30  Content:300     Mode:0
118FD8  PtrData[0]      = 300    (= 300)
118FDC  PtrData[1]      = 301    (= 301)
118FE0  PtrData[2]      = 302    (= 302)
118FE4  PtrData[3]      = 303    (= 303)
118FE8  PtrData[4]      = 304    (= 304)
118FEC  PtrData[5]      = 305    (= 305)
118FF0  PtrData[6]      = 306    (= 306)
118FF4  PtrData[7]      = 307    (= 307)
118FF8  PtrData[8]      = 308    (= 308)
118FFC  PtrData[9]      = 309    (= 309)
```

(a) Different Test Modes for the AXI Stream Test Component

```
Reset XAxiDma Done
Packet N1 :     Nbr:100 Content100
...
>> RxDone Interrupt Caught
        Verifying...SUCCESS
Output took 2374 clock cycles.
Output took 3.56 us.
Bandwidth:897.73 Mbit/s

Packet N2 :     Nbr:200 Content200
...
>> RxDone Interrupt Caught
        Verifying...SUCCESS
Output took 4810 clock cycles.
Output took 7.22 us.
Bandwidth:886.15 Mbit/s

Packet N3 :     Nbr:300 Content300
...
>> RxDone Interrupt Caught
        Verifying...SUCCESS
Output took 6156 clock cycles.
Output took 9.24 us.
Bandwidth:1038.60 Mbit/s

Packet N4 :     Nbr:400 Content400
...
>> RxDone Interrupt Caught
        Verifying...SUCCESS
Output took 7470 clock cycles.
Output took 11.22 us.
Bandwidth:1141.20 Mbit/s

*** END PROGRAM ***

DMA Test - Multiple Packets
Reset XAxiDma Done
Packet N1 :     Nbr:100 Content100
...
>> RxDone Interrupt Caught
        Verifying...SUCCESS
Output took 2376 clock cycles.
Output took 3.57 us.
Bandwidth:896.97 Mbit/s

Packet N2 :     Nbr:200 Content200
...
>> RxDone Interrupt Caught
        Verifying...SUCCESS
Output took 4810 clock cycles.
Output took 7.22 us.
Bandwidth:886.15 Mbit/s

Packet N3 :     Nbr:300 Content300
...
>> RxDone Interrupt Caught
        Verifying...SUCCESS
Output took 6156 clock cycles.
Output took 9.24 us.
Bandwidth:1038.60 Mbit/s

Packet N4 :     Nbr:400 Content400
...
>> RxDone Interrupt Caught
        Verifying...SUCCESS
Output took 7480 clock cycles.
Output took 11.23 us.
Bandwidth:1139.68 Mbit/s
```

(b) Multiple packets read and bandwidth

Figure A.4: Test component simulation outputs

A.4 Project Collaboration

A.4.1 GiHub

To share projects between collaborators a github repository was created. All the codes are on here.

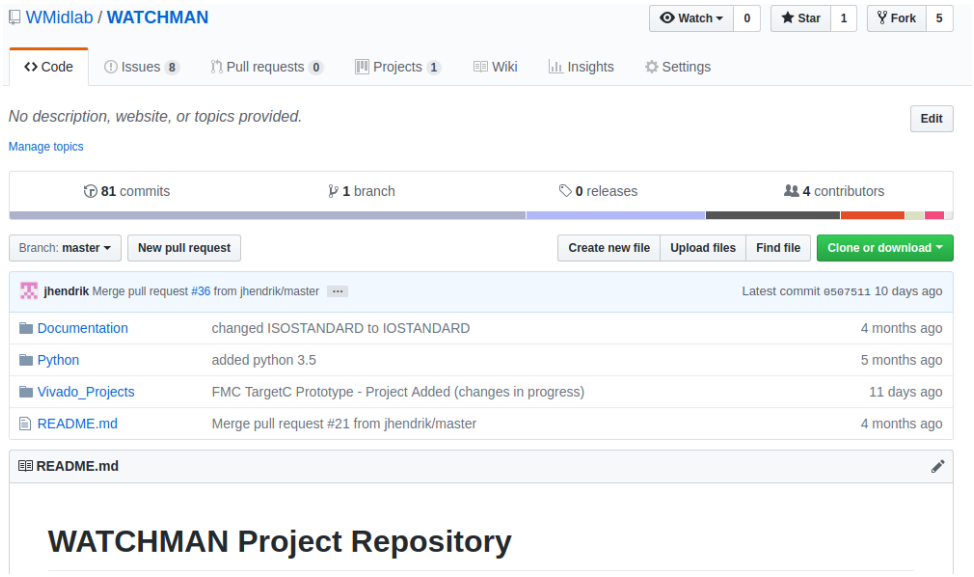


Figure A.5: GitHub Repository

A.4.2 TCL Script

Tool Command Language Script is used to share the Vivado Projects between different collaborators working on this project. Indeed sharing projects is very difficult in terms of computer, PATHs and operating systems. In addition to that a lot of temporary files are created using the Vivado Project Mode. Hence a new way had to be setup to share easily the the projects. Vivado GUI uses TCL commands for every action, finding the different commands to re-generate the project folder was step one. Step two is adding the different files (C code, VHDL and constraints) to the project and have the correct path. A folder structure is mandatory (see figure A.6).

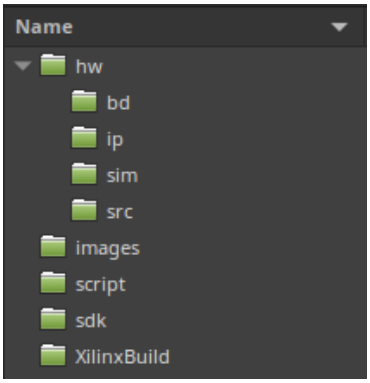


Figure A.6: Folder structure for Vivado Projects for GitHub

hw

The hardware (hw) folder contains all what Vivado needs to create a project, this is to say the block design (bd), the IPs used in the project, the simulation files (sim) like testbenches and waveform configuration and finally the source code (src) description of circuit in VHDL, Verilog or even System Verilog.

images

This folder contains the built images, this to say the bit file for programming the FPGA (logic side) and the hardware definition need for SDK to build C projects.

script

Various TCL script which help to build or save data including the one to create the new project on Windows and Linux platforms.

sdk

SDK creates a project of of its own and does not use TCL script. Also different problems linked to opening SDK without Vivado made problems. Hence the SDK projects are keep untouched and just linked to the new sdk project created by Vivado.

XilinxBuild

The build folder of Xilinx project, this folder is in the GitIgnore file and all files inside this folder are ignored, limiting the unnecessary files on GitHub.

B

Source Code - VHDL

Active source code

All codes are found on the gitHub repository `https://github.com/WMidlab/WATCHMAN`

AXI_Lite_pkg.vhd,
AddressDecoder.vhdl,
BitSelector.vhd,
WindowCPU_pkg.vhd,
TARGETC_pkg.vhd,
BlockDelay.vhd,
CNT_EN.vhdl,
CPU_CONTROLLERV3.vhd,
DFF.vhd,
DataDecoder.vhdl,
GrayDecoder.vhd,
GrayEncoder.vhd,
HammingDecoderV2.vhd,
LookupTable_LE.vhd,
RDAD_WL_SMPL.vhd,
RoundBufferV6.vhd,
SingleTrigger.vhd,
TARGETC_ClockManagementV3.vhd,
TARGETC_Control.vhd,
TARGETX_DAC_CONTROL.vhd,
TRIGGER_CONTROLLER.vhd,
WindowStoreV4.vhd,
Window_BrainV2.vhd,
afifoV2.vhdl,
clockcrossing_Buffer.vhd,
fifo.vhdl,
fifoManager_V4.vhdl,
TARGETC_TOPLEVEL.vhd,
base_zynq,
base_zynq_wrapper.vhd