NCO based CDR Documentation

Release 0.0.1

Filippo Marini

Aug 23, 2021

NCO based CDR Documentation

I IEEE RT2020 Submission **1**

1 Abstract **2**

2 Introduction **3**

3 Numerically Controlled Oscillator **4**
3.1 Phase resolution increase 5

4 Phase (Frequency) Detector **6**
4.1 Practical implementation 6

5 Conclusions **9**

II Code Documentation **10**

6 Top level **11**

7 Numerically Controlled Oscillator **14**

8 Frequency Manager **17**

9 Phase and Frequency Detector **19**
9.1 Phase Shift Filter 20
9.2 Quadrant Detector 21

10 Phase and Frequency Detector Manager **24**
10.1 Lock Manager 24

11 Phase Aligner **27**

12 CDR Frequency Library **28**
12.1 Freq_to_m 28
12.2 Freq_to_mmcm 28

13 Test Benches **29**

Part I

IEEE RT2020 Submission

CHAPTER 1

Abstract

The capability to extract timing informations out of a serial data stream to decode the incoming informations has become a very common requirement.
To sample the incoming data, the receiver usually relies on a Clock and Data Recovery (CDR) chip, which generates a clock signal at the corresponding sampling frequency, phase-aligned to the data.

Modern physics experiment have often this same requirement, where perhaps thousands of boards receive uncorrelated data and it's up to them to decode the messages. For that reason, the presence of a CDR on-board is usually mandatory.

Present readout systems in physics experiments usually rely on FPGAs to receive and transmit data at high rate to high capaicity DAQ systems; exploting FPGAs to recover timing information from streamed data is therefore beneficial for a number of reasons, including power consumption and cost reduction.

The design is based on two components: a Numerically-Controlled Oscillator (NCO), in order to create a controlled frequency clock signal, and a digital Phase Detector (PD) to match the clock frequency with the data rate.
NCOs are often coupled with a Digital to Analog Converter (DAC) to create Direct Digital Synthesizers (DDS), which are able to produce analog waveforms of any desired frequency. In the presented case, the NCO generates a digital clock signal of an arbitrary frequence, while the PD manages this frequency by intercepting any shifting on the relative phase between the clock and the data.

The paper presents the implemented CDR design, the limitations and the challenges involved, possible fields of application in actual physics experiments and, finally, some results.

CHAPTER 2

Introduction

The Clock and Data Recovery job is a relatively simple one: retrieve a clock with the frequency needed to sample each bit of the incoming data stream.
Its design, unfortunately, is not so trivial.

Usually a CDR architecture is similar to the Phase Locked Loop (PLL) model (Fig. 2.1), where the phase of a reference signal is compared to the phase of an adjustable feedback signal, generally provided by a Voltage Controlled Oscillator (VCO). The output of the Phase Detector (PD) is filtered and used to pilot the VCO frequency. When the phase comparison is in steady state, e.g. the phase and frequency of the reference signal is equal to the phase and frequency of the feeedback signal, we say that the PLL is locked.
In the case of a CDR, the steady state is reached when the VCO clock frequency match the reference signal's data rate.

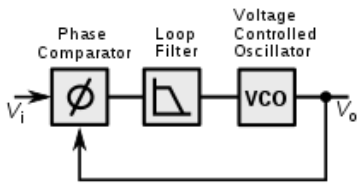


Fig. 2.1: Basic design of a PLL.

Essentially, breaking down the design, for a fully functional CDR, a controlled oscillator and a PD are needed.

This paper has the intent to show a possible implementation of a CDR. .. adopting the FPGA technology, in particular the target is a Xilinx Kintex 7 (XC7K325T–2FFG900C), which presents a good balance between performances and cost. In particular the target is a Xilinx Kintex 7 (XC7K325T-2FFG900C). The design is intended to work with a range of data rates that allows the use of the high range (HR) general purpose I/O pin of the FPGA over the dedicated tranceivers, resulting in a reduced power consumption and a more straightforward design.

CHAPTER 3

Numerically Controlled Oscillator

To generate a waveform, the VCO is substituted by a Numerically Controlled Oscillator (NCO)[1]. Its design consists of two parts:

- A phase accumulator (PA), which is basically a counter incremented by a reference clock
- A phase-to-amplitude converter, which uses the PA output as an index to a Look-Up Table (LUT)

To better understand the mechanism, we can think of a phase-wheel (Fig. 3.1). This phase-wheel is equally divided in a certain number of sections, bounded by phase-points (a.k.a. the PA output) and for each phase-point we associate the correspondant sine value (a.k.a. the LUT).

As a vector rotates around the wheel, by taking these sine values, a digital sine waveform is generated. A complete revolution around the phase-circle corresponds to a complete period of the sine wave.

Let's imagine now that the vector skips a few (fixed) points for each jump, the revolution is completed in a much shorter time: the frequency of the output waveform has increased!

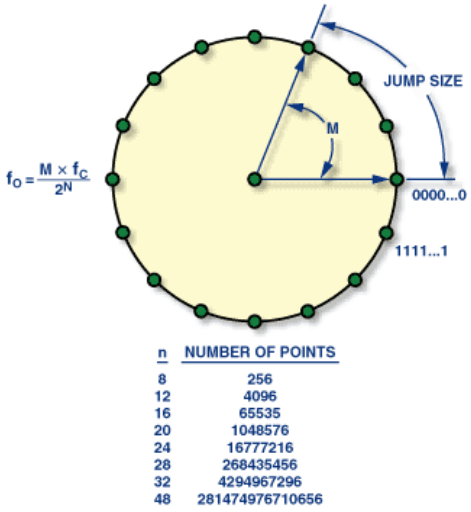
The correlation between the jump size, the reference clock and the output waveform frequency is

$f_{OUT} = \frac{M \times f_C}{2^n}$

where:

- M is the jump size
- f_{OUT} is the NCO output waveform frequency
- f_C in the reference clock frequency
- n is the length of the phase accumulator, in bits

[1] https://www.analog.com/en/analog-dialogue/articles/all-about-direct-digital-synthesis.html



n	NUMBER OF POINTS
8	256
12	4096
16	65535
20	1048576
24	16777216
28	268435456
32	4294967296
48	281474976710656

Fig. 3.1: The phase wheel

To retrieve a digital clock signal, the LUT is actually very simple: we just associate to half of the circle the digital value 0, and to the other half the digital value 1.

The design presents two main limitations:

- The first is the maximum frequency limit, which is given by Nyquist, and corresponds to half of the reference clock
- The second is the phase resolution. Since the output signal is digital, the time domain is discrete, and it corresponds to the reference clock period. This implies that the positive (and negative) fraction of the output clock signal can only be a multiple of this time domain resolution, making the output frequency only on average determined by the jump size of the accumulator.

While the first limitation is known and impossible to overcome, the second is design based, and must be resolved in order to be able to use this clock for CDR operations.

3.1 Phase resolution increase

As said, the NCO output can change its value only when the phase accumulator jumps from one phase-point to another (i.e. at the rising edge of the reference clock).
To improve the phase resolution, the parallelism capability of the FPGA is exploited.

Briefly, to reduce the NCO phase changing period, the trivial way is to increase the reference clock frequency.
To obtain the same result, without any frequency change, we can compute multiple points between one phase jump, and then serialize the results. This way, for each rising edge of the reference clock, multiple values of the output waveform are computed, increasing the resolution.

The NCO output clock will still present differences between the average frequency value and the istantaneous frequency value (the time domain is still descrete, we just reduced its period), but this can be filtered out feeding the signal to an FPGA's MMCM/PLL, in jitter filter mode.

CHAPTER 4

Phase (Frequency) Detector

To mimic the PLL architecture for the CDR, a phase/frequency detector is needed, in order to compare the NCO output clock frequency to the data rate.

To detect a frequency difference, the transition of the data signal shall be compared with the transition of two clocks of equal frequency that have a constant phase difference.

Denoting with f_d the data frequency and with f_{VCO} the clock frequency, we have that:

$f_d = (\phi_d(t_1) - \phi_d(t_0))/(t_1 - t_0)$

$f_{VCO} = (\phi_{VCO}(t_1) - \phi_{VCO}(t_0))/(t_1 - t_0)$

where $\phi_d(t)$ and $\phi_{VCO}(t)$ represents the data and clock phase respectively at the time t.
Let's keep in mind that the time t_1 and t_0 are given by the NCO clock, as the only time based signal.

The frequency difference is then given by:

$f_d - f_{VCO} = [(\phi_d(t_1) - \phi_{VCO}(t_1)) - (\phi_d(t_0) - \phi_{VCO}(t_0))]/(t_1 - t_0)$

The two phase differences in the numerator at the righ hand side of the equation are the output of the phase detector, comparing the data transition with the NCO clock transition at the instances t_1 and t_0.
These phase differences will vary with time (in case of frequency offset), making a frequency difference detection possible.

4.1 Practical implementation

By using two clocks with 50% duty cycle and orthogonal with each-other ($\pi/2$ of phase difference), it is possible to divide the entire 360 degrees clock period into four quandrants, as shown in Fig. 4.1 .

The two phase detector (one for each clock) indicate the quandrants where the data signal transition is located, updating this information at every new data edge.

If the data phase is shifting with respect to the clock edges, than the clock quadrant that detects the data transition will increase or decrease, accordingly to the phase shifting direction.

In the implemented design, the frequency detection capability relies on the use of two clock signals, with 50% duty cycle and orthoghonal with each-other. These two signals allows the division of a clock period into four quadrants (see Fig. 4.1).

To identify the quadrant of the data edges, informations by two Alexander-type phase detectors (Fig. 4.2) are registered and processed. Further processing is needed to determine whether the data edges are drifting up or down in the clock quandrants (due to higher or lower clock frequency) to consistenly adjust the NCO frequency. These frequency change requests to the NCO are constantly monitored in order to control CDR locked flag.

Informations on the phase and frequency detection techniques whose this design is based from, can be found here[2].

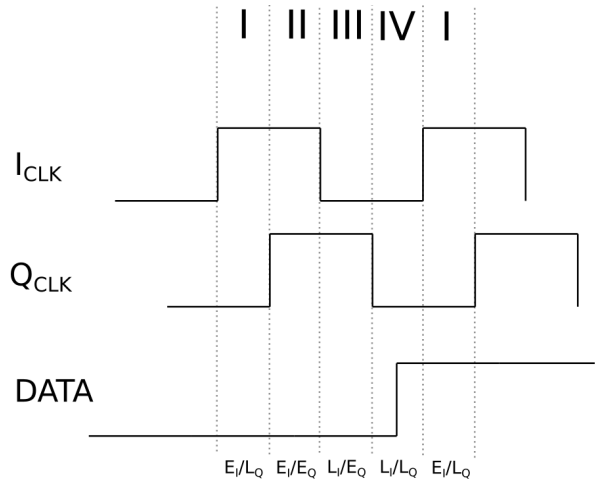


Fig. 4.1: The division of the clock period in four equal qudrants (indicated by the Roman numerals). I_{CLK} stands for In-phase Clock, which is the reference, Q_{CLK} stands for Quadrature Clock, which idetifies the $+\pi/2$ (or $-\pi/2$) phase difference clock. To idetify a quandrant, an Early (E) and Late (L) notation (Clk vs Data) is used. If a data transition is first located in quadrant III and then in quadrant II, the data phase is shifting to the left, which equals that the data transitions are based on a clock faster than the NCO clock.

[2] https://en.wikibooks.org/wiki/Clock_and_Data_Recovery

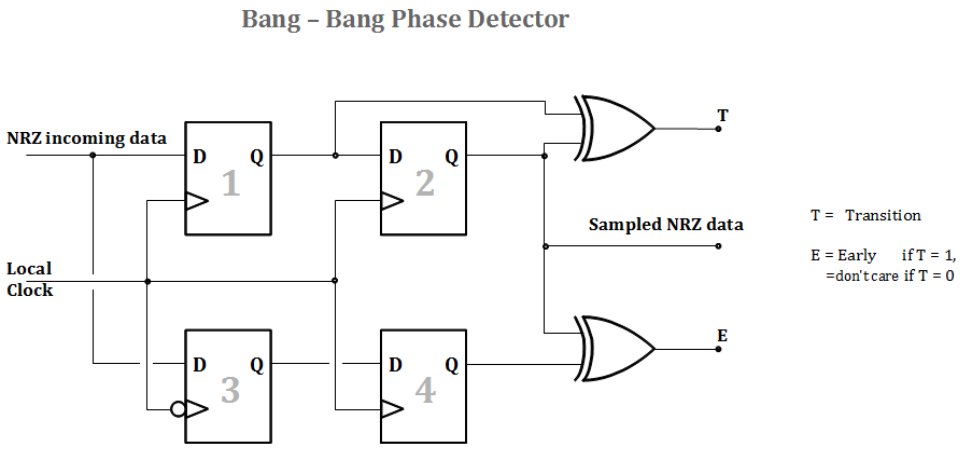


Fig. 4.2: The bang-bang PD compares the negative edge of the clock with the data transition, and the present data bit with the previous data bit. Using 4 flip flops the resulting info is contemporarily available for one entire clock period. The output T is active when a data transition is detected, the output E is active when the clock has been found early.

CHAPTER 5

Conclusions

The presented document briefly prensents the design for an FPGA implementation of a fully figital CDR.

The design is intended has proven to work with rates up to 250 Mbps. At such data rate, a possible implementation would be on the Global Control Unit (GCU) board of the JUNO experiment.

JUNO is a neutrino physics experiment, under development, where a big liquid scintillator detector will be read by about 20'000 large PMTs. Very close to the PMTs, underwater, the analogue signals are digitized, analyzed and stored in the GCU's FPGA.
Each GCU looks at three PMTs and, elaborating their data, a primitive trigger is generated. The trigger is then sent to the higher lever electronics, via a synchronous link, for a global trigger validation. In case the validation is positive, the same link is used to send back its timestamp. When received, the GCU sends the related waveform to the DAQ via Ethernet.

A CDR is needed to decode the synchronous link messages, which presents a data rate of 125 Mbps. This would be beneficial in terms of cost reduction.

Part II

Code Documentation

CHAPTER 6

Top level

file: *top_cdr_fpga.vhd*

The file *top_cdr_fpga.vhd* is the top level file for the CDR project.

For an easier code comprehension it is recommended to have the CDR documentation and code on the side.

The generic and ports used by the CDR design are:

- g_gen_vio: boolean, when "true" the Xilinx VIO is generated, whose ports are used to make the NCO generate a fixed clock frequency (*M_i*) and to enable the phase and frequency detector (*vio_DMTD_en*)
- g_check_jc_clock: boolean, when "true" the recovered clock is forwarded out to the differential pin *cdr-clk_jc_p/n_o*
- g_check_pd: boolean, when "true" some internal signals are forwarded out from the FPGA in order to be checked (with an oscilloscope for istance). Used for debug purposes.
- g_number_of_bits: positive, this defines the number of bits used by the NCO's phase wheel. The number of bits determine the NCO's output frequency resolution
- g_multiplication_factor: positive number which is needed to have an output frequency higher than the maximum obtainable frequency of the single phase wheel (due to Nyquist law). The user only need to make sure that $g\_freq\_out/2^{g\_multiplication\_factor-1} < g\_freq\_in/2$
- g_freq_in: real, system clock frequency (i.e., the frequency of the clock that enters the *i_phase_wheel_counter_1* instance), in MHz
- g_freq_out: real, NCO nominal output frequency (i.e., the data rate), in MHz
- g_out_phase: recovered cloc - data phase relationship

- sysclk_p/n_i: clock from the board crystal
- data_to_rec_i: data from which the clock is recovered

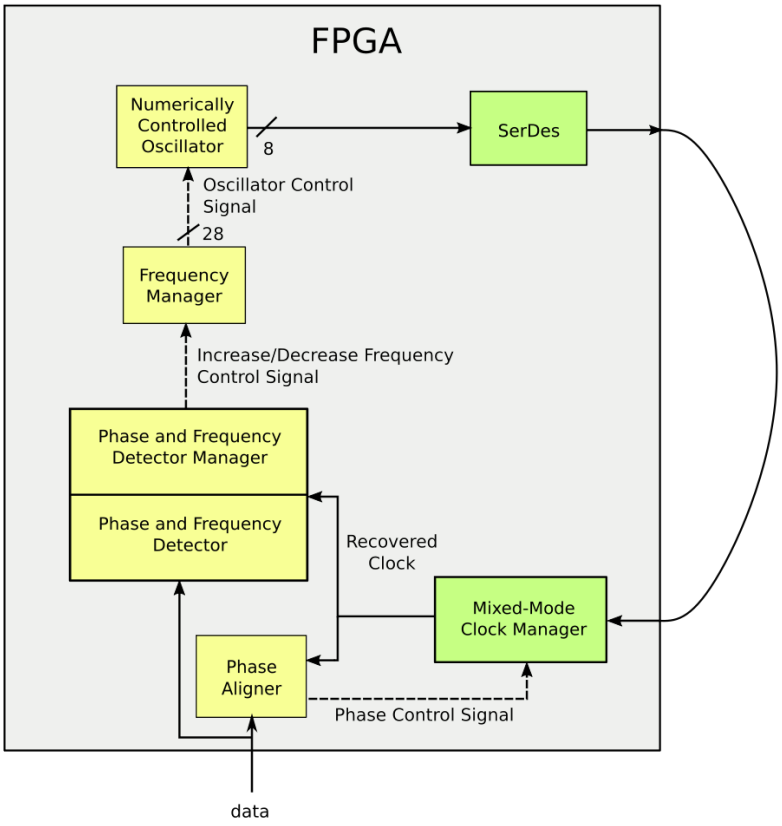


Fig. 6.1: Block level for the CDR proposed project

- cdrclk_p/n_o: NCO's generated clock which has gone through the OSERDESE2 tile and need an external loopback
- cdrclk_p/n_i: clock is going back in from the loopback
- cdrclk_jc_p/n_o: if enabled, this differential pins shows the recovered clock
- ledx_o: several LED showing whether the MMCM are locked, if data is entering the FPGA and if the NCO's clock is actually present
- shifting_o, shifting_en_o: debug ports

On the report, a block diagram of the CDR design is reported. The corresponding istances in the top level code are:

- Numerically Controlled Oscillator <=> *i_phase_wheel_counter_1*
- Frequency Manager <=> *i_frequency_manager_1*
- SerDes <=> *i_oserdese_manager_1*
- Mixed-Mode Clock Manager <=> *i_jitter_cleaner_1*, *i_i_q_cloc_gen_1*
- Phase and Frequency Detector <=> *i_pfd_1*
- Phase and Frequency Detector Manager <=> *i_pfd_manager_1*, *i_lock_manager_1*
- Phase Aligner <=> *i_phase_detector_unit_1*

Some of these istances will have its code explained here.

Other notable istances are: *i_slow_pulse_counter* which is used to show a defined LED pulse based on data rate, *PRBS_ANY_1* which is a PRBS checker, *i_prbs_counter_1* which is a counter of PRBS errors.

CHAPTER 7

Numerically Controlled Oscillator

Instance: *i_phase_wheel_counter_1*, file: *phase_wheel_counter.vhd*

The code is actually pretty simple. The phase wheel (Fig. 7.2) is actually a counter (*s_phase_wheel_counter*) which gets incremented by a fixed quantity, the jump size (*u_M*).

To improve the phase resolution, 8 phase wheels counters are generated and each one presents an offset of (jump size)/8 (refer to paper at chapter *The Serdes Technique*), so that, for example, if the jump size is 16, the offset would be 2, and if counter_1 is at 32, counter_2 would equal 34, counter_3 equals 36, counter_4 equals 38 and so on up to counter_8 with the value of 46.
The next clock clycle counter_1 will be at 48 and the others will still follow the same offset rule

The "LUT" (which is not really a LUT) which generates the clock signal from the counter is represented by the last line. Essentially you just take one bit of the counter, and these will oscillate between 0 and 1 with 50% duty cycle. Fig. 7.3 of the paper shows you an example of why 8 different clocks (2 in the figure) increase the phase resolution. For visual reason the wave in the paper is a sine wave, but the principle is the same with a digital wave.

In this module a grain and fine clock frequency selection is allowed. The grain selection is carried out during the extraction of the clock from the counter. The LSB oscillate faster than the MSB. The fine selection is performed by the jump_size (*M_i* port) which is what is used to match the NCO clock frequency with the data rate.

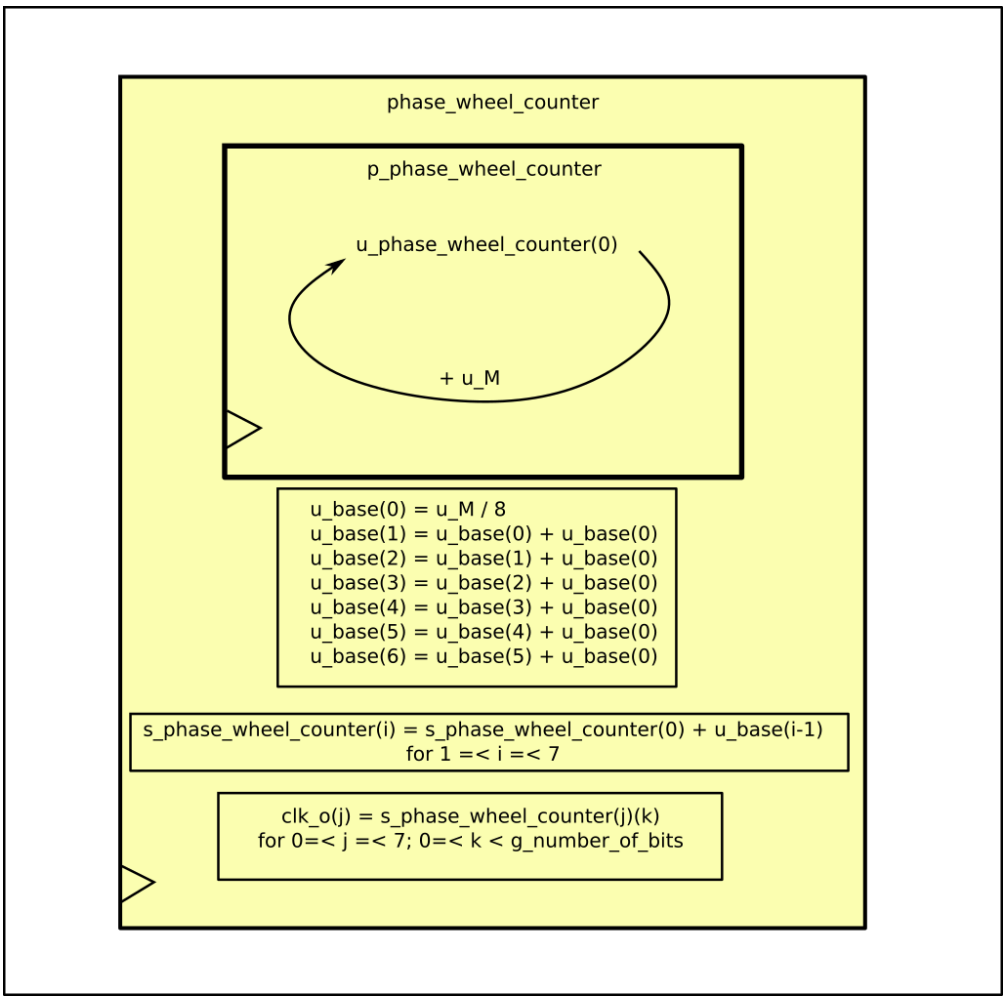


Fig. 7.1: Block diagram for the phase_wheel_counter component

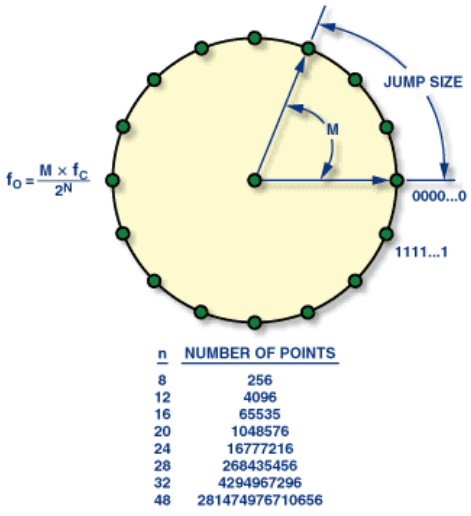


Fig. 7.2: The phase wheel. The equation on the left shows how to retrieve the out frequency starting from M, the jump size, f_C, the system clock and N, the number of bits used by the vector *s_phase_wheel_counter* (total - bit chosen for the clock)

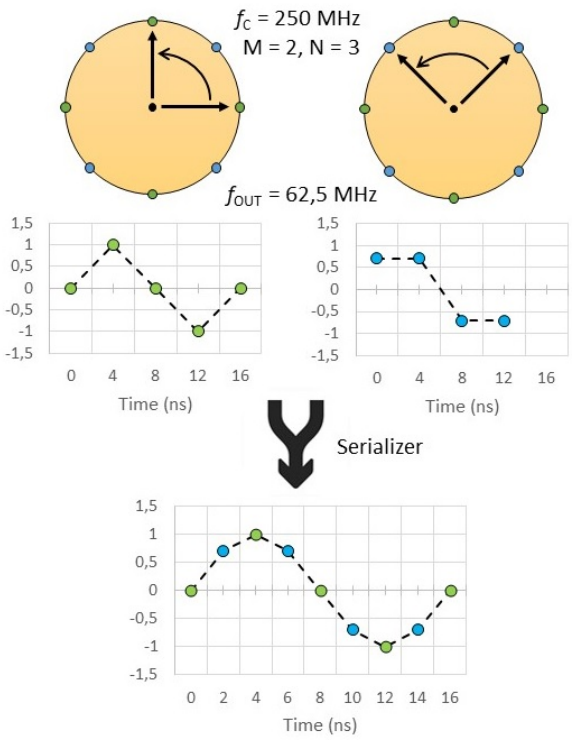


Fig. 7.3: example of phase resolution improvement by serialing different phase-wheel

CHAPTER 8

Frequency Manager

Instance: *i_frequency_manager_1*, file: *frequency_manager.vhd*

This module takes the frequency change requests in input and change the NCO jump size accordingly.

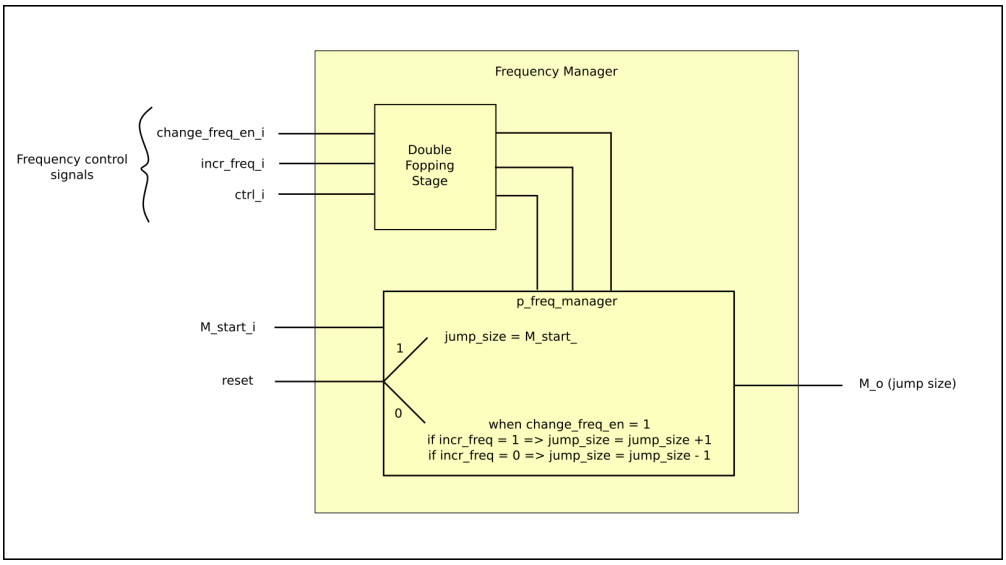


Fig. 8.1: Block diagram for the frequency_manager component

To take into account the clock domain crossing of the change frequency requests (source is the cdr clock while the destination is the system clock), to the enable and increase freuqency signals (which are the signals actually used to know whether to increase or decrease the NCO frequency), a third signal has been added, the control. As can be seen by Fig. 8.2, a single requests lasts for a few clock cycles, to make sure they (especially the control signal) stay up for more than two destion clock periods.

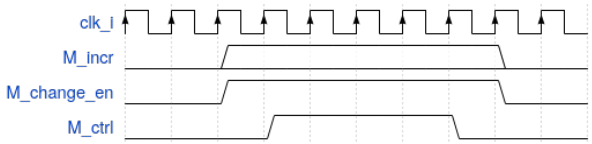


Fig. 8.2: Frequency chage requests timing diagram.

CHAPTER 9

Phase and Frequency Detector

Instance: *i_pfd_1*, file: *pfd.vhd*

In this section we are going to analyze the vhd files used to compare the NCO clock frequency with the data rate.

As explained in the documentation, the frequency matching is basen on dividing the whole clock period in 4 quadrants, and monitoring in which quandrant the data present its edges. If the quadrant of the data edges changes onver time, the clock frequency does not match the data rate. In particular if the data edges quadrant is shifting up, the NCO clock is faster, while if the data edges are costantly moving towards a lower quandrants, the NCO clock frequency is slower.

The quadrant detection capability relies on the use of two Alexander type bang bang phase detector (Fig. 9.1), one working with a so-called "in-phase clk" (clk_i_i) and the other with a "quadrature clock" (clk_q_i), featuring a pi/2 phase difference.

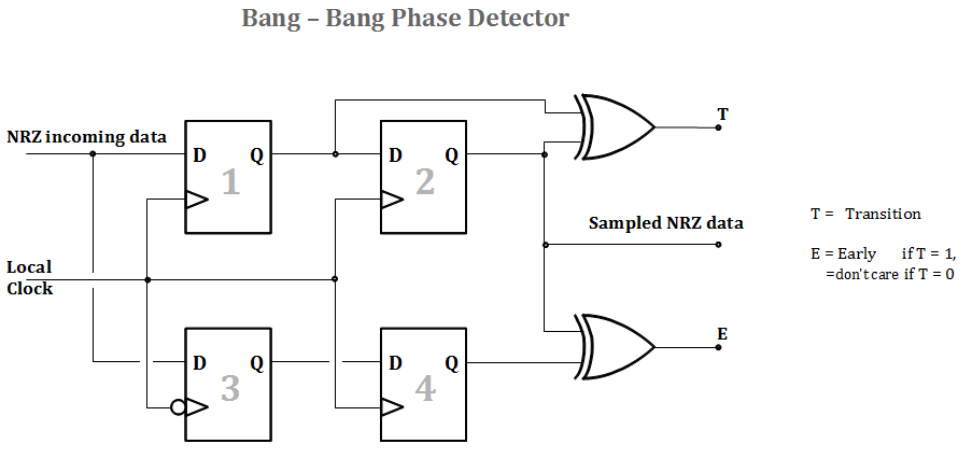


Fig. 9.1: Alexander-type Bang-Bang phase detector

The Early/Late signals of the phase detectors are filtered by the *phase_shift_filter* Master/Slave couple modules. The filtering is explained in the dedicated section.

The filtered Early/Late signals are monitoried by the *quadrant_detector* module which dinamically determines the current quadrant of the data edges. The shifting of quadrants is given by the *quadrant_shifting_detector* module.

9.1 Phase Shift Filter

Instance: *i_phase_shift_filter_slave_1, i_phase_shift_filter_slave_2*, file: *phase_shift_filter_slave.vhd* Instance: *i_phase_shift_filter_master_1*, file: *phase_shift_filter_master.vhd*

The *phase_shift_filter_master/slave* are components used to filter the raw up/down data-to-clock phase by the phase detectors in order to get rid of possible errors caused by jitter and bad sampling due to flip-flop setup/hold violations.

The mechanism of the filtering is very trivial: the master gives the slaves a user-defined time window in which the slaves counts the raw phase up or down flags. When the master window goes to 0, the slaves look at their counter and, based on a defined threshold, decide whether te data-to-clock phase is actually up or down.
Regarding the master, the length of the filtering window is $2^{g\_num\_trans}$. Concerning the slaves, the minimum data transition in order to take a decision is $2^{g\_num\_trans\_min}$, while the threshold for the counter to take a decision is half of the registered number of data transitions.

The phase_up/down output is stretched for a configured number of steps (usually 3) for Clock Domain Crossing (CDC) reasons.

In order fo the slaves to take a decision, a minimum of data edges must be present (data must be AC balanced).

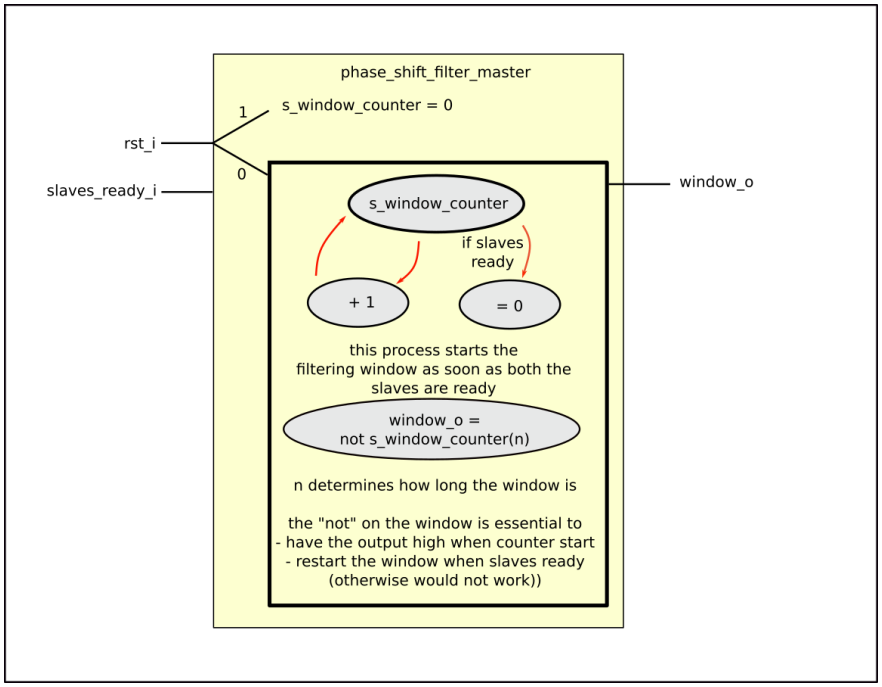


Fig. 9.2: Block diagram for the phase_shift_filter_master

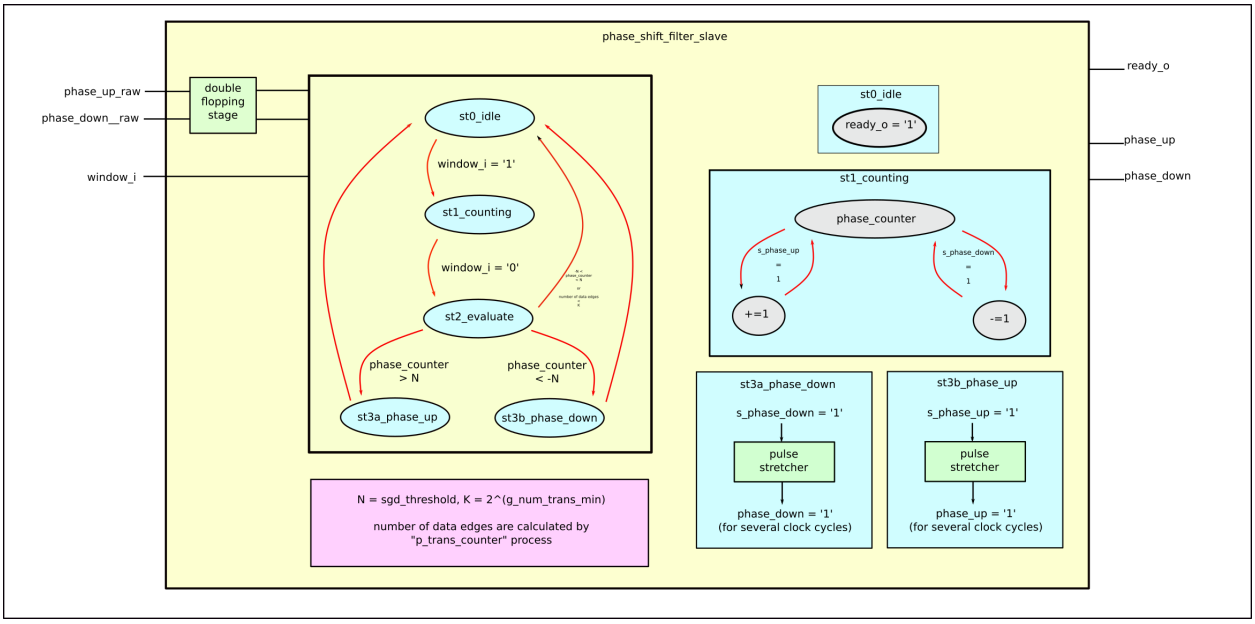


Fig. 9.3: Block diagram for the phase_shift_filter_slave

9.2 Quadrant Detector

Instance: *i_quadrant_detector_1*, file: *quadrant_detector.vhd* Instance: *i_quadrant_shifter_detector_1*, file: *quadrant_shifting_detector.vhd*

The *quadrant_detector* module detects in which clock quadrant the data has its edges. To do so, it processes the informations passed on by the *phase_shift_filter_slave* modules.

The quadrant information is ten used by the *quadrant_shifting_detector* module in order to monitor the shifting of the data edges quadrant to dictate whether the clock frequency is faster or slower than the data rate.

To understand how the quadrants are identified, please refer to Fig. 9.4

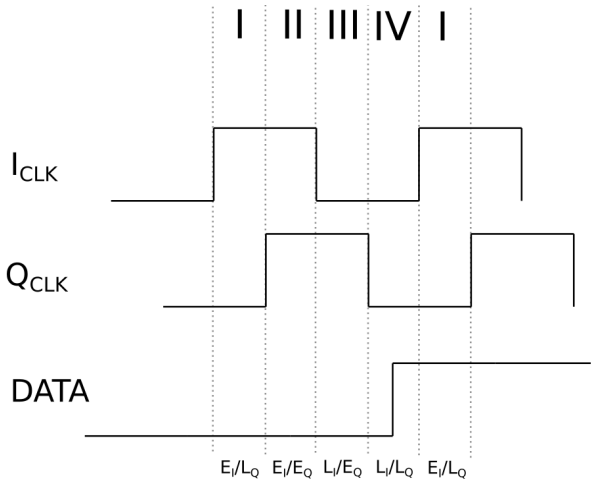


Fig. 9.4: Quadrants definitions based on early/late data-to-clock phase.

The concept behind how the modules work is not really difficult. Please look at the source VHDL code and look at

the following figures for an easier comprehension.

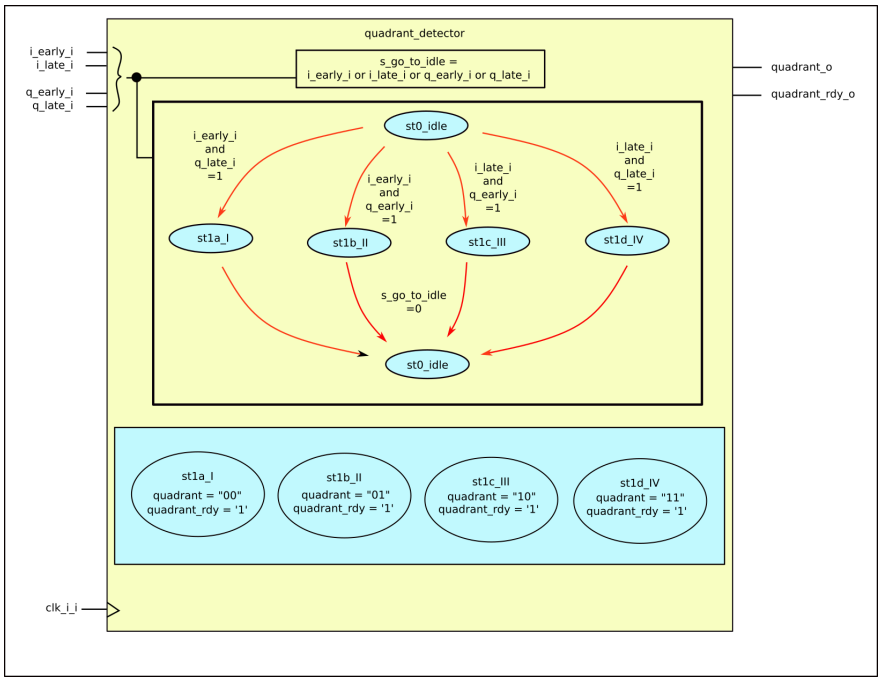


Fig. 9.5: Block diagram for the quadrant_detector

To avoid any mis-shifting-detection going from the idle state to the next states, the *quadrant_shifting_detector* module presents a set-reset flip-flop which enables the shifting identification only when at least one quadrant was already identified.

The *locked_o* port of the *quadrant_shifter_detector* module can be though as a primordial CDR lock flag, but in the code this is actually not used and the locked flag comes from the *lock_manager* module.

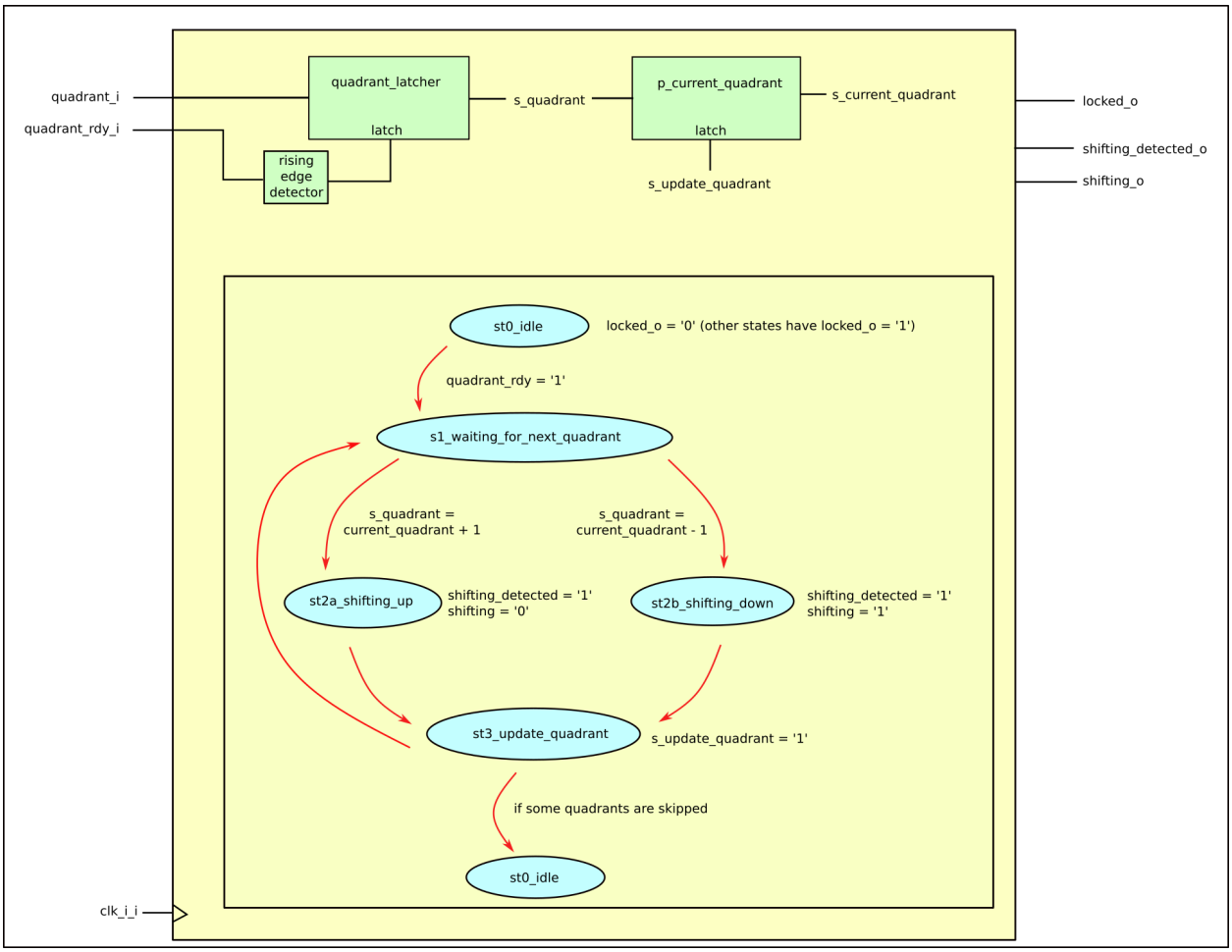


Fig. 9.6: Block diagram for the quadrant_shifting_detector

CHAPTER 10

Phase and Frequency Detector Manager

Instance: *i_pfd_manager_1*, file: *pfd_manager.vhd*

The frequency manager module's job is to make sure the NCO clock freuqency is as close as possible to the data rate. Since its impossible for the two to be an exact match, due to the finite resolution of the clock frequency and real-world conditions (i.e., jitter, setup/hold time violation ...), the Frequency Manager exploits a counter filtering method (similar to the *phase_shift_filter* module) with several different threshold to get to the closest wanted frequency. Also, when this condition is met, the *locked_o* flag is asserted high and will be deasserted if the input data stops or the data rate mismatches the NCO clock frequency.

As said, the counter mechanism (+1 when frequency increase request, -1 when frequency decrease request) employs different threshold in order to detect whether the CDR is locked:

- *lock threshold* (around 10% of the maximum possible value): if counter ends up inside this range, the CDR is locked
- *activate threshold* (around 50%) when CDR is locked, outside this range a frequency change request is forwarded to te NCO
- *unlock threshold* (around 90%) if exceeded, the CDR locked is deasserted

A Set/Reset Flip-Flop manages the lock and unlock flags.

Together with the M-change requests, a control signal is sent out, to comply with the CDC that will happen when passing this signal to the NCO.

10.1 Lock Manager

Instance: *i_lock_manager_1*, file: *lock_manager.vhd*

The *lock_manager* module monitors the *locked_o* signal from the Frequency Manager to decide whether the CDR is locked to the data or not.

Basically, if the *locked_o* stays up for a certain number of periods, than the CDR is locked. On the other hand, if *locked_o* stays low for the same certain number of period, than the CDR is not locked.

Watch out for aliases!!

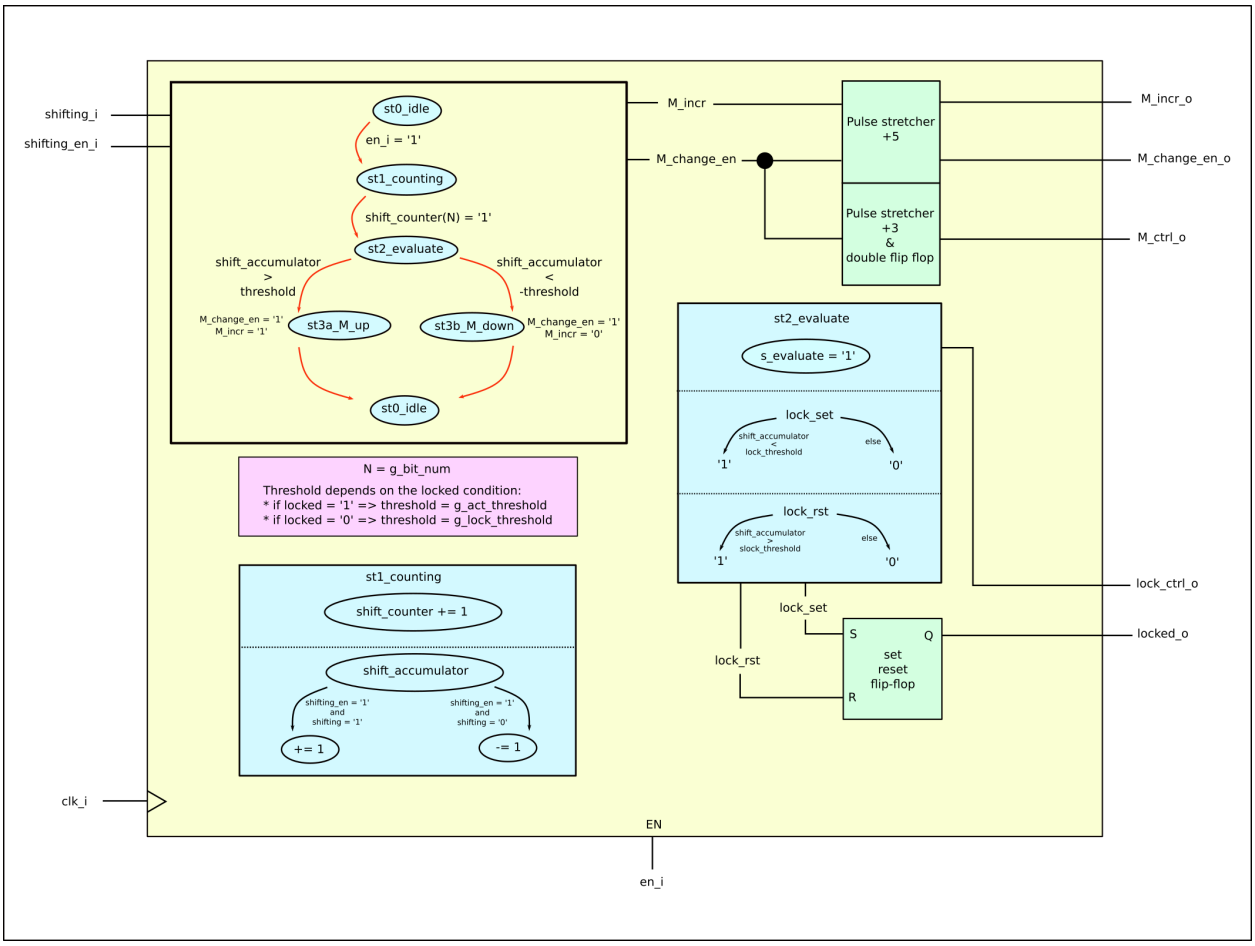


Fig. 10.1: Block diagram for the pfd_manager module

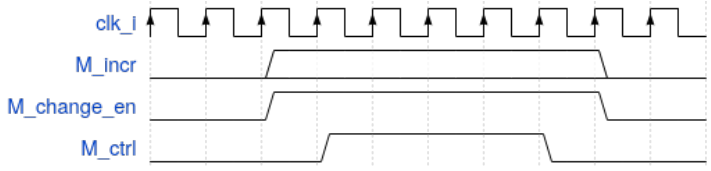


Fig. 10.2: Timing diagram for the M-change request to be passed to the NCO

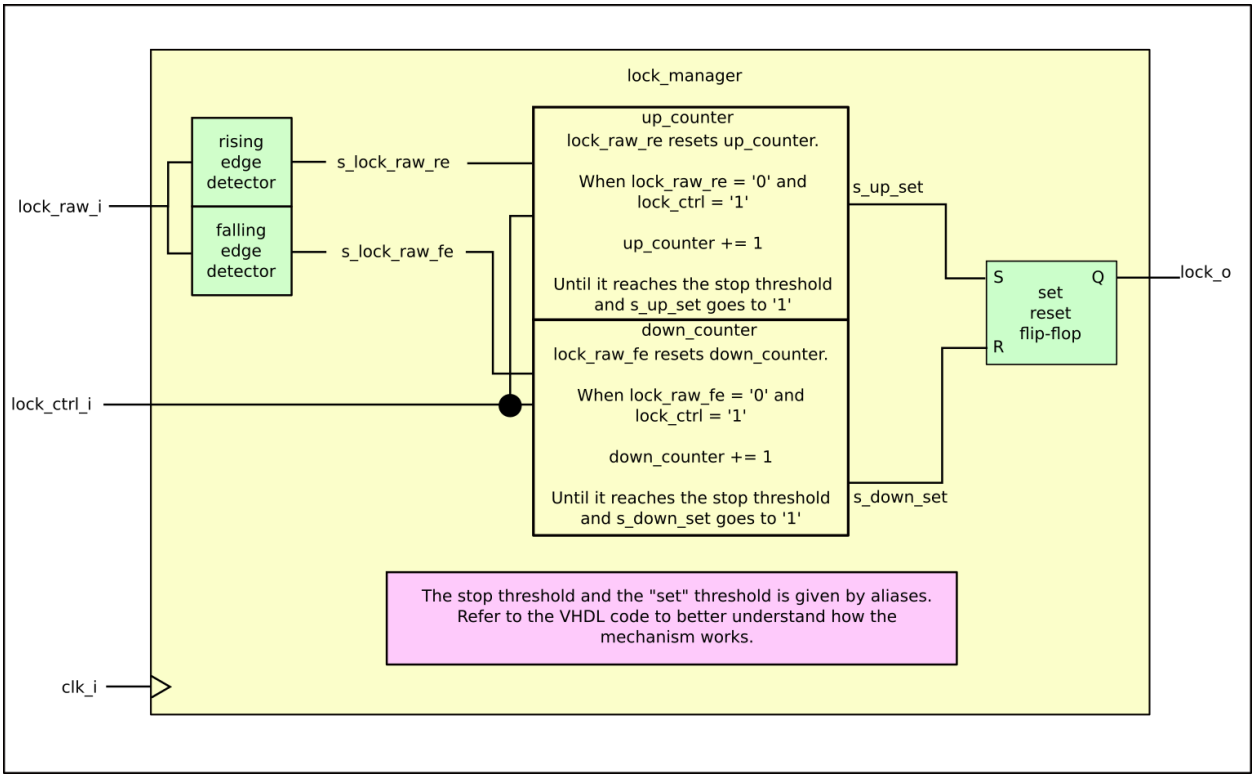


Fig. 10.3: Block diagram for the lock manager module

CHAPTER 11

Phase Aligner

Instance: *i_phase_detector_unit_1*, file: *phase_detector_unit.vhd*

The *phase_detector_unit* instance is needed to have a very well defined phase relationship between the recovered clock and the incoming data stream. Moreover, since the frequency detector is not able to perfectly match the NCO clock with the data rate, the recovered clock will never stop drifting. This dynamic phase adjustment fixes this issue.

The *phase_detector_unit* module consists of three parts. Since these are very similar to the "Phase Shift Filter" components, no block diagrams are shown. Please refer to that chapter to easily understand the VHDL code. The differences will be listed in the following.

- *i_phase_detector_1* is an Alexander-type Bang-Bang phase dector
- *i_phase_shift_filter* is very similar to the *phase_shift_filter_slave* component. The difference is that here there is no master, as we don't need to compare two different phase detection streams. The filtering window is therefore generated directly inside the module, using the *bit_num_trans_time* generic for lenght definition.
- *i_ps_controller_1* is an MMCM dynamic phase adjustment signal controller. Since the "Phase Detector" has to communicate with the MMCME_2_ADV tile, the *phase_up* and *phase_down* flags generated by the *i_phase_shift_filter* intance must comply with the MMCM phase adjustment protocol. This is what the ps_controller achieves.

CHAPTER 12

CDR Frequency Library

The *freq_utils* package include two functions that are used by the top level project file.

12.1 Freq_to_m

The *freq_to_m* function is used in order to transform the *g_freq_out* top level generic into a jump size value for the NCO.

The function's inputs are:

- The system clock frequency, given by the *g_freq_in* top level generic, real
- The NCO expected nominal frequency, given by the *g_freq_out* top level generic, real
- The multiplication factor, given by the *g_multiplication_factor* top level generic, positive
- The NCO number of bits, given by the *g_number_of_bits* top level generic, positive

The function's declaration is *freq_to_m(g_freq_in, g_freq_out, g_multiplication_factor, g_number_of_bits)* and returns a real

12.2 Freq_to_mmcm

The *freq_to_mmcm* function is used by the MMCM to generate the *clkfbout_mult_f* and *clkin1_period* generics in order to keep the VCO frequency at 1 GHz

The function's input is:

- The NCO expected nominal frequency, given by the *g_freq_out* top level generic, real

The function's declaration is *freq_to_mmmc(g_freq_out)* and returns a real

CHAPTER 13

Test Benches

Included in the "src" folder, several test benches are available to test different modules of the project. The test files are distinguishable from the syntesizable files as they are contained in folders ending with the "_tb" suffix.

The test are ment to be run with GHDL software (Tested with GHDL 0.37-dev, llvm version).

In order to generate the test bench executionable file, run the "Makefile" with the "make" command.
Remember to dump the wave file, as there is no automatic test or assertion.

For the GHDL user guide on how to run a test bench, dump the wave file and define its time lenght, refer to[1].

[1] https://ghdl.readthedocs.io/en/latest/