dds_cdr Documentation

Release 0.0.1

Filippo Marini

Dec 12, 2019

RT2020 ABSTRACT PAPER

CHAPTER

ONE

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

TWO

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.

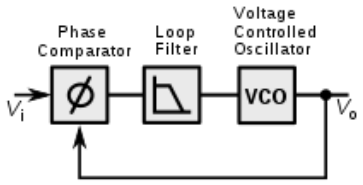


Fig. 2.1: Basic design of a PLL.

Essentially, breaking down the design, for a fully functional CDR, a controlled oscillator and a PD are needed. Needless to say, these components are not natively available in an FPGA.

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.

To generate an arbitrary frequency clock signal, a Numerically Controlled Oscillator (NCO) is designed. NCOs are digital signal generators which are able to provide discrete-time-and-values waveforms, with user-defined frequency. To control and compare the frequency of the NCO clock to the reference data stream, a few options are currently being evaluated, and will be presented in the dedicated section.

CHAPTER

THREE

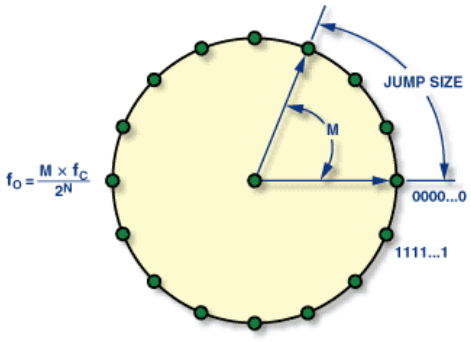
NUMERICALLY CONTROLLED OSCILLATOR

To generate a waveform, the design of a NCO[1] consists of two parts:

- A phase accumulator (PA), which is basically a counter incremented by a reference clock
- A phase-to-amplitude converter, which associates a waveform Look-Up Table (LUT) to every possible PA output value, using it as an index.

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 and for each phase-point we associate the correspondant sine value.
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!



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

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

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

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

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

For the actual implementation, the phase-point touched by the vector are defined by the PA: for each rising edge of the reference clock, the counter skips an arbitrary number of points, therefore obtaining the arbitrary frequency.
The phase-to-amplitude converter is actually very simple: since we are only interested in creating a digital clock signal, 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 offset 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

FOUR

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:

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

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

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 quadrant that detects the transition will change, in a direction compatible with the phase shifting direction.

The implementation of such a kind of phase detector is still under evaluation. One possible solution is to use two Alexander type Bang-Bang phase detector (Fig. 4.2), one working with the reference clock, the other with the $\pi/2$ phase offset, to idetify the quadrants, and adjust the NCO frequency at every quadrant transition.

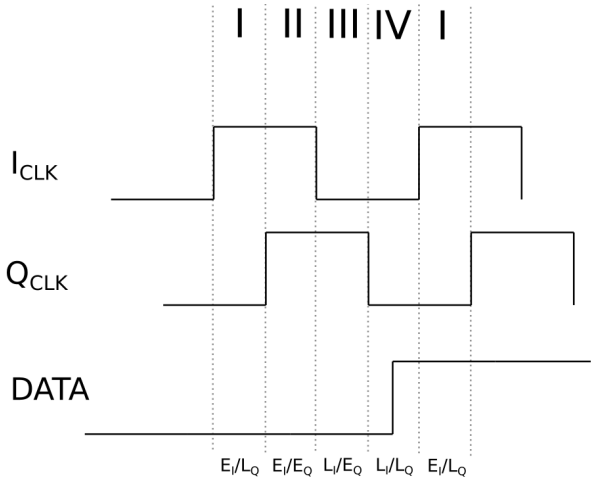


Fig. 4.1: The division of the clock period in four equal qudrants (indicated by the Roman numerals). I_{CLK} stands for In-phase Clock, which is the reference, Q_{CLK} stands for Quadrature Clock, which idetifies the $+\pi/2$ (or $-\pi/2$) phase difference clock. To idetify a quandrant, an Early (E) 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.

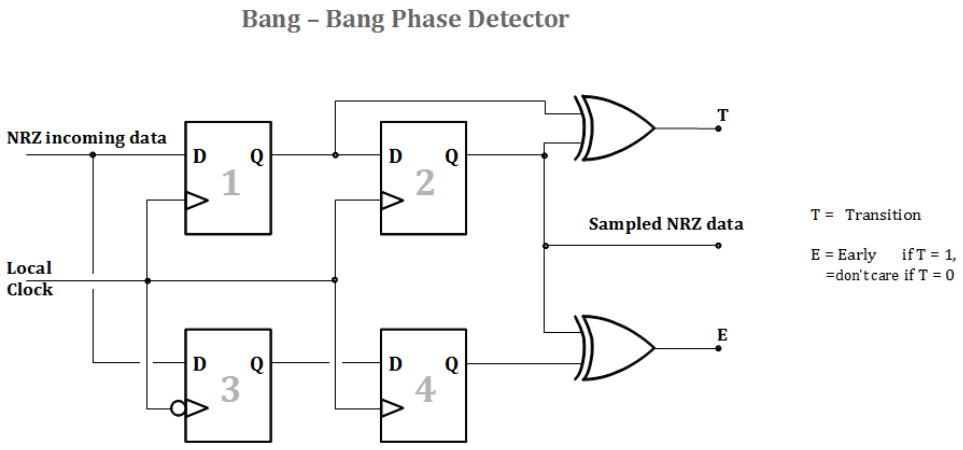


Fig. 4.2: The bang-bang PD compares the negative edge of the clock with the data transition, and the present data bit with the previous data bit. Using 4 flip flops the resulting info is contemporarily available for one entire clock period. The output T is active when a data transition is detected, the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

CHAPTER

FIVE

CONCLUSIONS

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

The design is intended to work with a data rate of 125 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.

CHAPTER

SIX

PHASE DETECTOR

After this arbitrary clock is created by the NCO, it must be confronted with the digital data entering the FPGA (for now the digital data is another clock) in order to match its frequency. This will be obtained by a digital Phase Detector (PD), in order to re-create in FPGA a Phase-Locked Loop (PLL). The PD will dynamically detect the phase shifting of data vs. clock, and will tell the NCO to increase or decrease the frequency accordingly.

Since it's impossible (or maybe just not trivial) for an FPGA to detect phase shifting with an infinite resolution (like an analog phase detector), a finite resolution on the phase shifting must be taken into account. This means that, even though is possible for the NCO to match very closely the data frequency, the clock will always "walk", up to the phase shifting resolution, before getting re-adjusted (and walking on the other direction, keeping going back and forth).

This "adjusting" operation is strongly non-linear, since

- its operation consists on the generation of fixed length pulses every time the counter meets the threshold.
- it does not collect any data about the phase-difference amount or the time between phase-shifts.

The implemented PD is based on the White Rabbit (WR) Dual Mixer Time Difference (DMTD) phase detector. For reference go to section 3.5.4 and 4.3.5 of this document.

A description of the functional blocks is following

6.1 locker_manager

Before starting to monitor the phase between the clock and the data to match the frequency, an initial condition (lock) must be asserted.

The phase information is retrieved by the n_cycle value (The reader should really give a look to section 3.5.4 of the document suggested at the beginning of the PD description) which has a well defined range, finding its maximum value when the clock is aligned with the data edge.

For the *locker_monitoring* to determine the phase shifts, the the differences of the n_cycle value over time are computed. For these differences to always make sense, we need to take into account possible overflow, which happens when n_cycle increases when on the maximum edge (max -> zero), or decreases when on the minimum edge (zero -> max). In order to have a good range of values to work with, the initial condition is fixed when the n_cycle value is at half of its range. Everything is managed by a Finite State Machine (FSM):

- st0_idle: this is the idle state of the machine. As soon as the module is enabled (DMTD_en_i = '1') then the machine proceeds to the first operative state
- st1_calculate_n_cycle_max: this is the state where the maximum value of the n_cycle range is retrieved. This is easily obtained by feeding to the n_cycle calculator the same clock signal. As a matter of fact, this is exactly what the state does by enabling the DMTD_max_en_o signal.

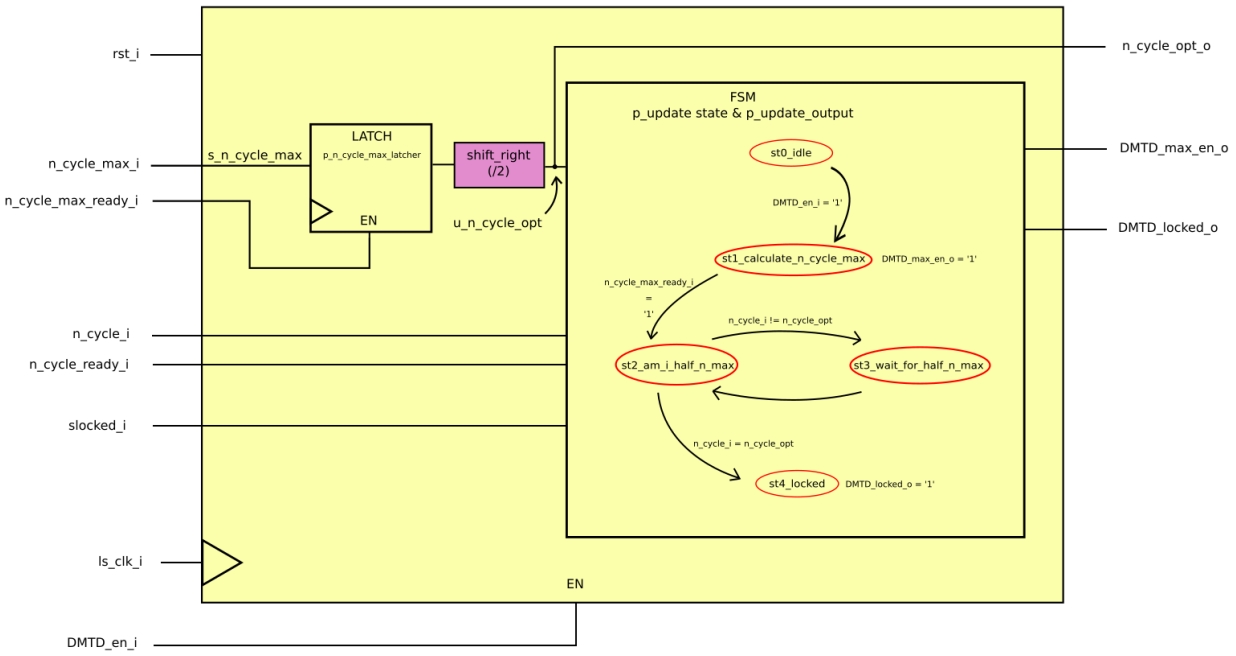


Fig. 6.1: locker_manager block diagram

- st2_am_i_half_n_max: the meaning of this state is very self-explanatory. After the n_cycle maximum value is registered (s_n_cycle_max), its half value is computed (u_n_cycle_opt), and at every rising edge of the clock the state checks whether this value equals the present n_cycle.
- st3_wait_for_half_n_max: another very-hard-to-get state (:P). If the previous state didnt find the present n_cycle to be half of the max, it goes into this state and it immediately goes back to state 2 for checking again (I just realized this state is useless, I will remove it when I'll have the chance).
- st4_locked: There it is! the locked condition is met, i.e. the present n_cycle is at half of the maximum value, and the *locker_monitoring* module have a good range af values to work with. If *locker_monitoring* determines that the lock is loss, the machine will go back to the idle state and start over.

6.2 locker_monitoring

"locker_monitoring" will detect whether the clock is faster or slower in respect to the data, by monitoring the n_cycle value. It is also responsible to check whether the lock condition is still true (or better, it checks if the loss of lock condition is satisfied). Basically "locker monitoring" will start monitoring when the rising edge of the "locked" signal is detected (as said *here* , this means that the n_cycle is exactly at half of its maximum range). When monitoring, the module will increase or decrease the NCO frequency by its output signals "change_freq_en_o" and "incr_freq_o" based on the following table:

Action	change_freq_en_o	incr_freq_en_o
Increase	1	1
Decrease	1	0

The incr/decr condition is met depending on the first derivative of n_cycle: if n_cycle is increasing over time, the the clock is too fast and a decrease pulse should be issued, vice-versa, if n_cycle is decreasing, then the pulse will be an

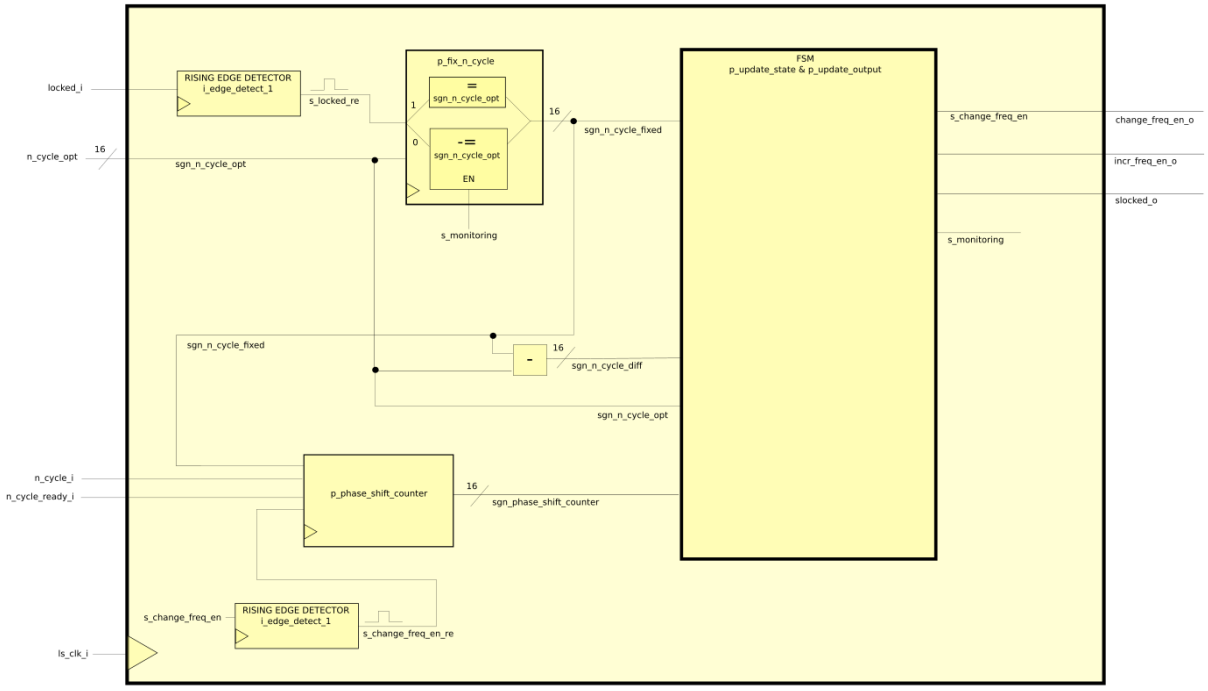


Fig. 6.2: locker_monitoring block diagram

icrease one. Of course, the n_cycle must go under a low-pass filter in order to avoid mis-pulses (due to sampling and clock jitter). Take a look at *p_phase_shift_counter* to have an idea how the first derivative sign (the algorithm is only sensible to the increasing or decreasing of the phase, it does not gather any information on the "speed" of the shifting) is determined.

All the module presented in the block biagram below will now be explained in details.

6.2.1 p_fix_n_cycle

This process will determine which is the present n_cycle value (sgn_n_cycle_fixed).

If a rising edge of the locked condition is detected, the n_cycle is resetted to the optimal value given by the locker_manager (sgn_n_cycle_opt), which is half of the maximum n_cycle range.
Otherwise, if the nco frequency is changed, i.e. the change_freq_en_o goes to '1', than sgn_n_cycle_fixed increases (or decreases) its present value by sgn_phase_shift, which represents the current n_cycle offset (more details on *p_phase_shift_counter*).

The difference between the current n_cycle (sgn_n_cycle_fixed) and the starting n_cycle (n_cycle_opt) is always given by the sgn_n_cycle_diff signal.

6.2.2 p_phase_shift_counter

The process is in charge of the phase shifting determination, that means it decides whether the phase has actually moved (forward or backward) one step (phase detector's sensibility) from the previous position (sgn_n_cycle_fixed).

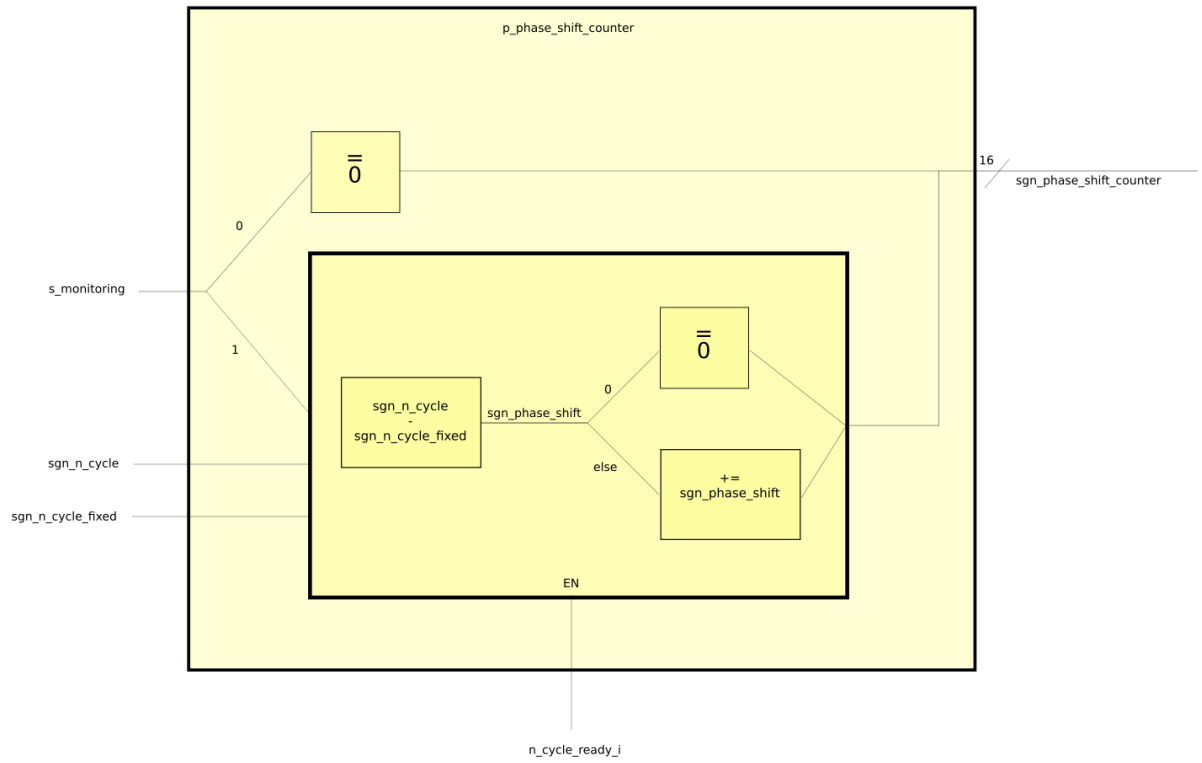


Fig. 6.3: p_phase_shift_counter process block diagram

The idea is to mimic the low-pass filter of a PLL with a simple counter and threshold.

First of all the process only works when the signal s_monitoring from the *FSM* = '1', otherwise the counter is resetted.
When active, the istantaneous phase shift is dynamically monitored (sgn_phase_shift), subtracting the present n_cycle value (sgn_n_cycle_fixed) to the istantanous n_cycle (sgn_n_cycle). Of course this presents an enable signal (sgn_n_cycle_ready) in order to sample the correct n_cycle value.

The n_cycle counter, when entering a new value, keeps jumping from the old to the new value (and vice-versa) until a semi-stable condition is reached (for example, is n_cycle is increasing, sgn_phase_shift would be something like this ...000010010001010111111..., if decreasing just substitute '1' with '-1').
The counter should only reach the threshold when the stable condition is met, therefore if sgn_phase_shift = 0 the counter is resetted. This way, only consecutive ones or minus ones are taken into cosideration.

The output of the process is the phase shift counter (sgn_phase_shift_counter) which will be a crucial input to the *FSM*.

6.2.3 FSM

The FSM (p_update_state and p_update_output processes) manages the frequency of the NCO and the loss of lock control.

As soon as the locker_manager module is locked, the FSM enters the st1_monitoring state. Here the phase shift counter is continuosly monitored to check whether it goes above threshold or beyond (- threshold). Depending on which of these two conditions are satisfied, the next state will be st2a_incr or st2b_decr, which will change the NCO frequency.

When the FSM find itself in the incr/decr state, an "if" condition monitors the loss of lock: if the current n_cycle (n_cycle_fixed) is very close to the edge of the n_cycle range, than the lock is lost (the reader should remember that the lock condition starts at the middle of the range, so to get to the edge means that the NCO frequency is really not that close to the data frequency).
Also, if the istantaneous n_cycle (sgn_n_cycle) differs from the current n_cycle (again, sgn_n_cycle_fixed) of user-decided units (something like 3), than there is loss of lock. This condition is needed to avoid funny behaviour when the clock frequency is very different and n_cycle changes several time while the counter is reaching the threshold.

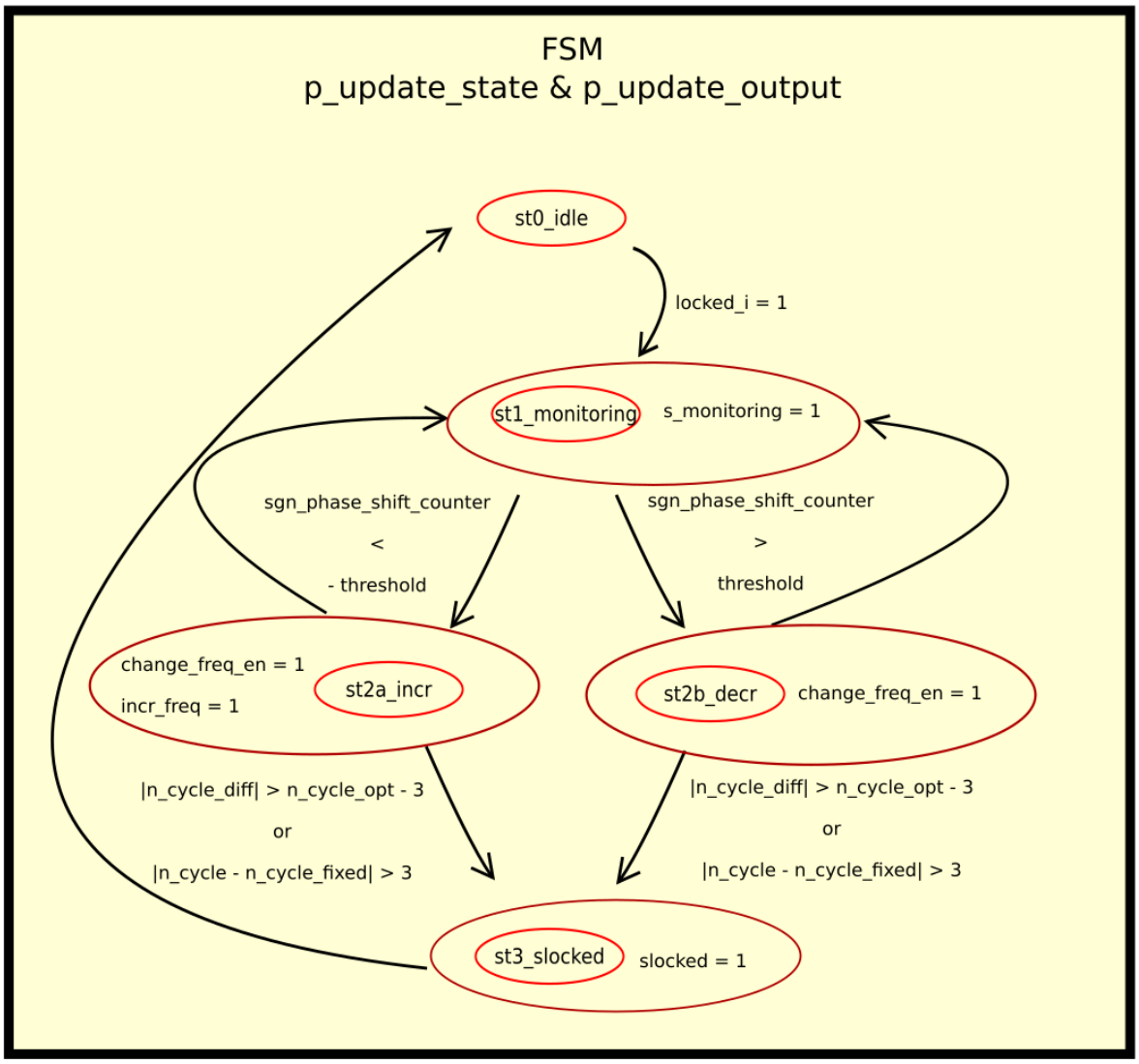


Fig. 6.4: FSM block diagram