A multi-channel configurable sinusoidal waveform generator using CORDIC

Ashwin Rajesh (Sr. No 21519)
M.Tech Microelectronics and VLSI Design
Indian Institute of Science, Bangalore

***Abstract*— This report for the course project for the "Digital System Design for FPGA" course by prof. Kuruvilla Varghese documents a time-multiplexed sinusoidal wave generator that uses the CORDIC algorithm to generate sinusoidal waves of variable frequency, amplitude and offset. The system was tested on the Basys3 board.**

I. Introduction

CORDIC (COordinate Rotation DIgital Computer) is an algorithm for efficiently computing the rotation and inverse rotation on fixed point inputs using shift and add/subtract operations. It avoids using any multiplication or division making it very efficient in terms of resource utilization and speed.

Here, a sinusoidal waveform generator is developed that uses the CORDIC algorithm to generate a sinusoidal wave with variable frequency, gain and DC offset. The CORDIC core is time-multiplexed to enable multiple channels to share the hardware to generate independent sinusoidal waves. All key parameters of the core including phase and value register widths, number of iterations and output value width is reconfigurable.

The system is integrated and tested on a Basys3 FPGA board with a dual channel DAC.

II. Problem statement

Design a Sine waveform generator using CORDIC algorithm. You can choose the architecture you want and decide on constraints(E.g.,Min Resources/ Max frequency of waveform etc).

Considering the problem statement, **maximum bandwidth** was chosen as the optimization criteria. However, in the final integrated system, the serial communication of the DAC imposes a major bottleneck preventing maximum utilization of this bandwidth.

III. Algorithm

The CORDIC algorithm implements a 2D rotation to rotate an X and Y input by an arbitrary angle. It attempts to approximate the 2D rotation by using a combination of certain fixed "micro-rotations". These angles are specifically chosen such that they can be implemented by shifts rather than multiplications since certain elements in the matrix become inverse powers of two.

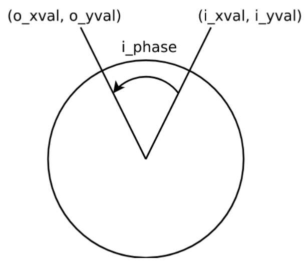


Fig. 1: CORDIC problem statement

A. CORDIC algorithm

A 2D rotation essentially comprises of multiplication of a 2D vector (with X and Y coordinates) with a 2D matrix, R(θ) as shown in Fig 2.

$$R(\theta) \;=\; \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}$$

Fig. 2: 2D rotation matrix for angle θ

The core idea in CORDIC is to choose angle such that tan(θ) is a power of two. Now, taking the cos(θ) outside and taking tan(θ) = 2^{-k}. This matrix and its equation implemented using shifts is shown in Fig. 3. This operation does a rotation by $\tan^{-1}(2^{-k})$

$$T\left(2^{-k}\right) \;=\; \begin{bmatrix} 1 & -2^{-k} \\ 2^{-k} & 1 \end{bmatrix}$$
$$\begin{aligned} x_{k+1} &= x_k - (y_k >> k) \\ y_{k+1} &= (x_k >> k) + y_k \end{aligned}$$

Fig. 3: Micro-rotaion matrix and equation

However, the cos(θ) term is not considered here. This comes up as a "scaling factor" in the rotation rotation matrix which comes up to be $\sqrt{1 + 2^{-2k}}$. This matrix can be easily converted to a rotation by -θ instead of θ by inverting the signs of the shifted terms (sine terms) in the matrix. The final positive and negative rotation matrices are shown in Fig. 4.

$$T\left(2^{-k}\right) = \sqrt{1+2^{-2k}}\begin{bmatrix}\cos(\theta_k) & -\sin(\theta_k)\\ \sin(\theta_k) & \cos(\theta_k)\end{bmatrix}$$

$$T\left(2^{-k}\right) = \sqrt{1+2^{-2k}}\begin{bmatrix}\cos(\theta_k) & -\sin(\theta_k)\\ \sin(\theta_k) & \cos(\theta_k)\end{bmatrix}$$

Fig. 3: Micro-rotaion matrix and equation

The CORDIC algorithm uses multiple such rotations in either the positive or negative direction to implement the net rotation. The direction of rotation depends on the angle left to be rotated to reach the required angle. 3 variables are required, x, y and phase. The direction is decided based on the previous value of phase. The x and y registers are updated based on the equation in Fig. 3. The angle is updated by incrementing or decrementing $\tan^{-1}(2^{-k})$ which is stored in a look-up table for each iteration, k.

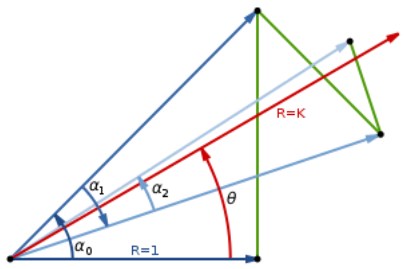


Fig. 4: CORDIC micro-rotations

The scaling factor is handled by using an initial x value of inverse of the products of all the individual scaling factors (and y = 0), so that final amplitude becomes 1.

B. *Pre-rotation*

The CORDIC algorithm converges only when the input rotation angle is between -100° and 100°. To cicumvent this limitation, we perform a pre-rotation before CORDIC. This pre-rotation brings all rotations by any angle from -180° to 180° into a rotation by an angle from -45° to 45°. This range can be achieved even without the first CORDIC rotation, and hence the iterations start from k = 2. The pre-rotation scheme is shown in Fig. 5.

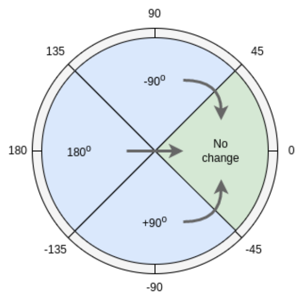


Fig. 5: CORDIC pre-rotation

Each 90 or 180 degree rotation is done by changing the signs and/or exchanging the x and y inputs.

IV. Architecture

The architecture was designed for opimized bandwidth using a pipelined architecture. Each CORDIC titration is one stage in the pipeline. The high bandwidth can be used for time-multiplexing n number of channels on the same CORDIC core with each channel getting 1/n of the original bandwidth.

A. *Number representation*

All the values in CORDIC are represented using signed fixed point representation. This is represented as Qn.m where n is the number of integer bits and m is the number of fractional bits. The number representations is shown in Fig. 6.

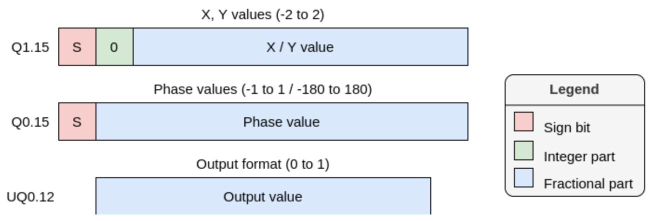


Fig. 6: Number representation

UQn.m represents unsigned fixed point format. The X and Y values have one integer bit to handle overflows because of the scaling factor. The phase is represented as -1 to 1 with -1 mapped to -180° and 1 mapped to 180°. It can also be interpreted as a UQ0.16 format from 0 to 1 with 1 representing 360° because of the way 2s complement representation works. The output is an unsigned value from 0 to 1. This can be directly given to a single-ended DAC.

B. *Wavegenerator architecture*

The wavegenerator is designed to be able to produce any sinusoidal waveform as required by using several configuration inputs. Following are the inputs to the wavegen for each channel. The widths of all inputs are parameterized, but the implemented widths are shown.

TABLE I: Wavegenerator inputs

Input	Description
Enable	High to enable channel output
Step size[16]	Phase change made in every step
Prescaler[8]	A step is made every (n+1) number of clock cycles
Init[17]	Initial X value. Used to control amplitude
Offset[12]	Output offset value when CORDIC output is 0

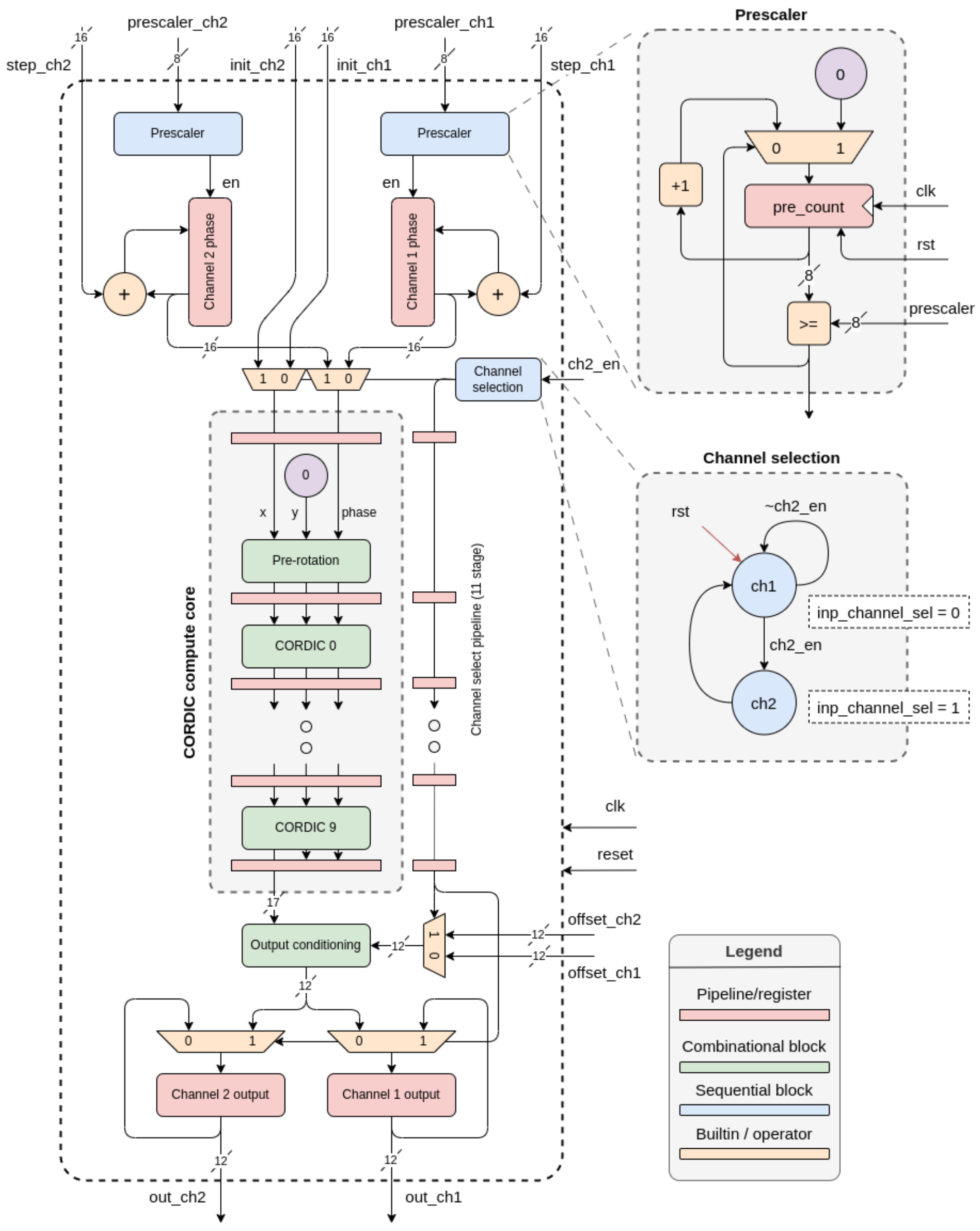


Fig. 7: Dual channel wave generator datapath and controller

Hardware testing was done on a dual channel wave generator. However, a design was also made with a parameterizable number of channels. Hereafter for simplicity, unless mentioned otherwise, we will be concerned with the design of a dual channel wave generator with 16 bit value and phase registers, 12 bit output register and an 8 bit prescaler.

The datapath of a dual channel wave generator is shown in Fig. 7. Reset and clock inputs are assumed to be fed to all registers and sequential blocks. All registers are initialized to 0. It is assumed that channel 1 is always enabled and channel 2 is disabled when channel 1 bandwidth needs to be increased. An n-channel wave generator would have a similar structure but the channel selection controller would be a n-channel round-robin scheduler.

The CORDIC compute blocks implement the equations in Fig. 3. Each of them also stores the angle rotated in that iteration to modify the phase remainder value. Theese angle values ($\tan^{-1}(2^{-k})$) are generated automatically during synthesis to enable generation of wave generators for any number of iterations and arbitrary phase resolution.

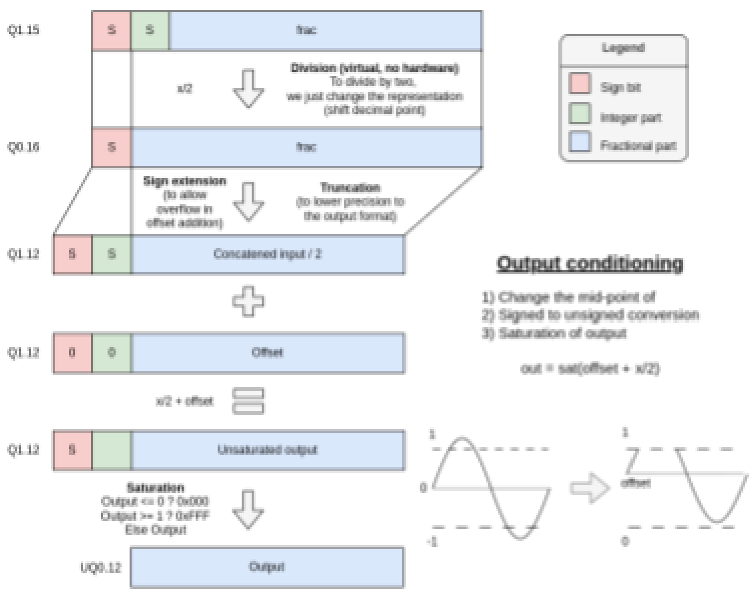


Fig. 7: Output conditioning

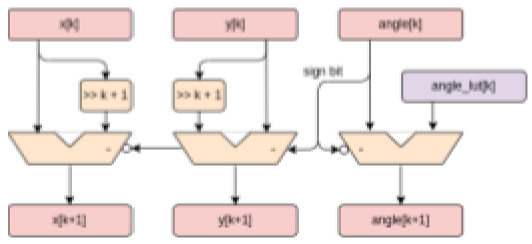


Fig. 8: CORDIC core

The output conditioning circuit implements the offset and saturation of the CORDIC output to convert it into a well behaved unsigned number that can be fed to a DAC. The steps for doing this is shown in Fig. 8.

To generate a sinusoidal waveform of a required frequency, the following equation is used.

$$Output\ frequency\ =\ \frac{Clock\ frequency\ x\ Step\ size}{2^{N}\ x\ (Prescaler + 1)}$$

Where N is the width of the phase register (16 here).

V. Design space exploration

There are multiple parameters for the CORDIC core which can be tweaked to get the required balance between accuracy, speed/latency and resource consumption. The parameters are shown in Table II.

The latency is proportional to ITER and indirectly on VW and PW since they increase the adder widths which can decrease frequency if they form part of the critical path. The resource utilization can be approximated as a product of PW, VW and ITER. OW and DW only decide the prescaler and output saturation region widths and their impact can be neglected compared to the other factors. So, to design the system for a particular application, OW and DW are fixed and the other parameters are optimized depending on the required performance, resource utilization and accuracy. PW can also be fixed since it decides the output frequency relationship and the resolution of input phase.

TABLE I: Module parameters

Param	Name	Description
PW	Phase Width	Width of phase registers (and step size)*
VW	Value width	Width of x and y registers (without sign)
DW	Divider width	Width of prescaler*
OW	Output width	Width of output value*
ITER	Iterations	Number of CORDIC iterations

*: Total width including sign bit, int and frac parts

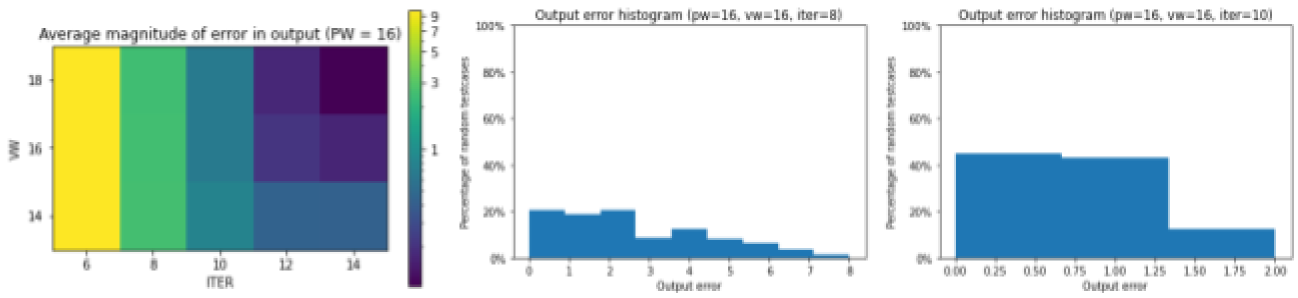


Fig. 9: Design Space Exploration data

This analysis was done on the dataset using an automated simulation environment. OW was fixed at 12 since the DAC available was 12 bit wide. DW was fixed at 8 arbitrarily (prescaler value of 255 was found to be sufficient) and PW is fixed at 16 to get sufficient resolution. For this combination, the error was measured. Error is measured as the difference in the output count (considering it to be an unsigned integer without any fractional part) from a reference output calculated using the $sin function in verilog.

The heatmap fo this average error v/s different VW and ITER combinations can be seen in Fig. 9. VW=16 and ITER=10 seems to be the best trade-off since the average error is almost 1 which is negligible. Number of iterations has a much higher impact than width of value register. Decreasing iteration count to 8 increases the average error above 3, as can be seen in the histograms.

VI. System Design

The wavegenerator was tested on a Basys3 FPGA board using a serial DAC. The toplevel module and the interfaces designed with the various peripherals is shown in Fig. 10.

Since 8 inputs are to be given to the wave generator, a register file and an access system was designed. The left and right push buttons are used to select the register to be accessed and the center button enters the data from the slide switches into the required register. The LEDs display which register is currently selected and the 7 segment display shows the contents of the selected display. The up button resets the whole system. All buttons were debounced.

The x_init input is 15 bit and uses the UQ0.15 representation. It is used to control the wave amplitude. The final wave amplitude is

$$V_{ampl} = V_{cc}\ x\ init\ x\ 1.1644\ /\ 2$$

Where 1.1644 is the scaling factor for CORDIC (without 45° rotation). By default, it loads the value corresponding to 1/1.644 leading to a sine wave of Vcc / 2 amplitude and centered at Vcc / 2 (offset = 0.5). The channels output 1kHz and 2kHz waves respectively.

The DAC uses an SPI like interface with one clock pin, a chip select pin and two data pins which are latched on the negative edge of the clock. The interface is implemented using a state machine. It has 2 states and a 4-bit counter which indicates the current bit being sent. It has a start signal indicating when to start the next transmission, but this is always high to continuously send data.

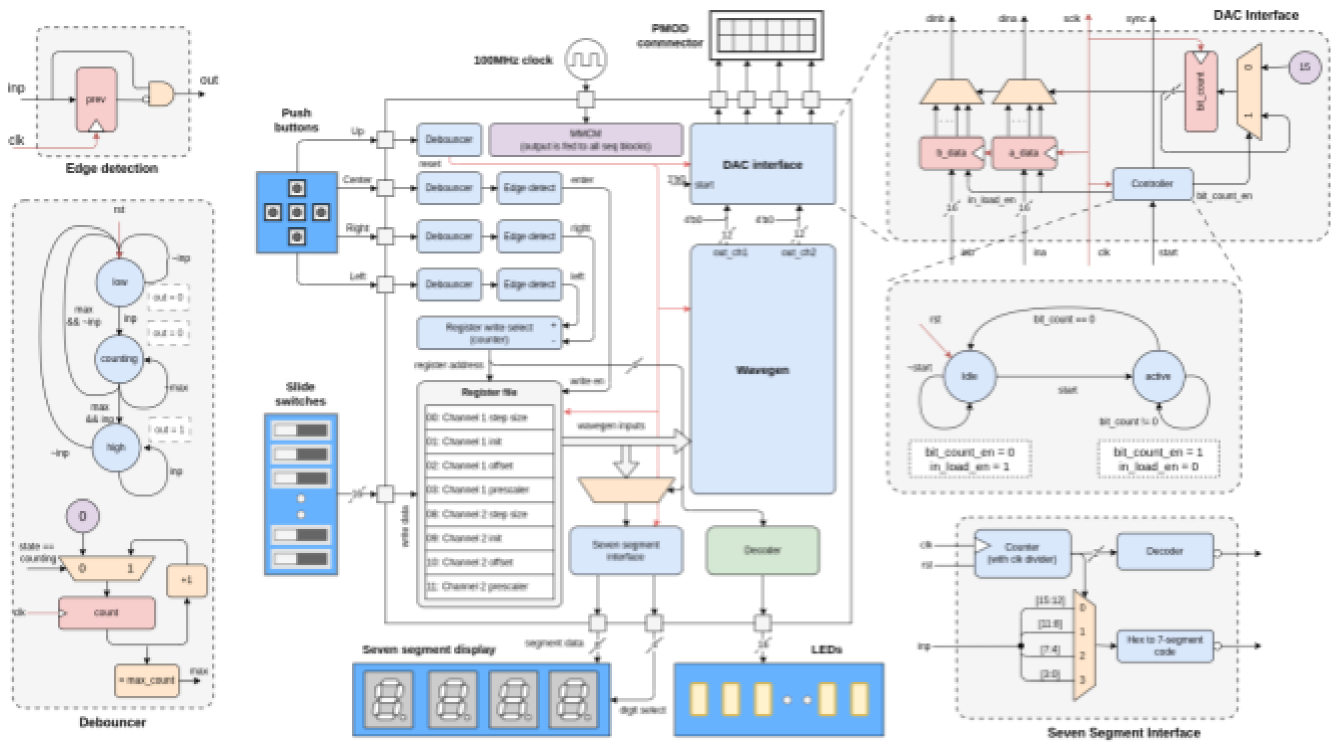


Fig. 10: Top level system design and peripheral interfaces

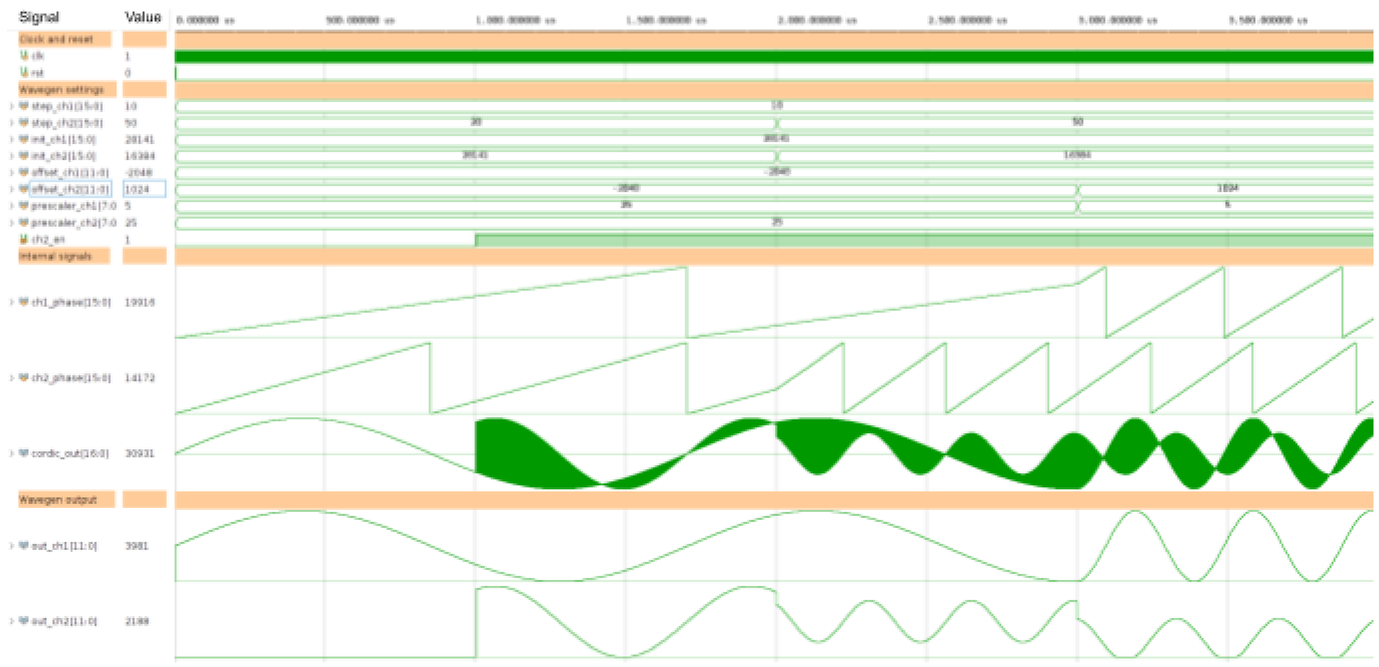


Fig. 11: Dual-channel wave generator behavioral simulation results

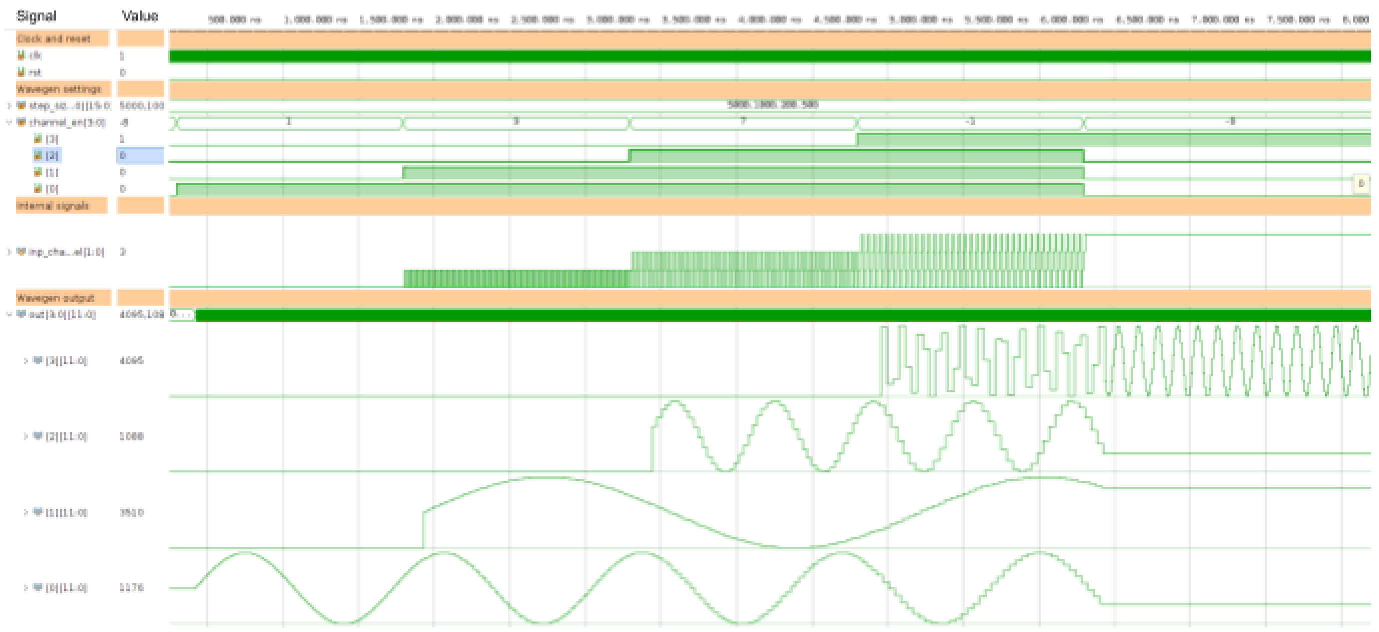


Fig. 12: Quad-channel wave generator behavioral simulation results

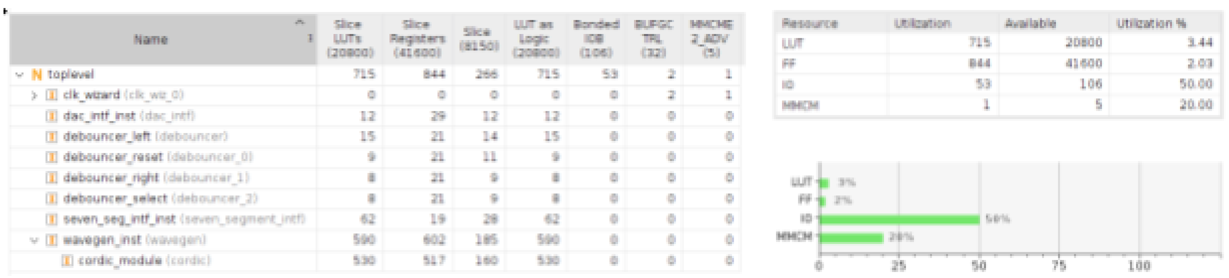
The DAC sends one sample every 17 cycles (16 data bits and a SYNC cycle). However, the wavegen generates a sample every cycle. This mismatch can be an issue at high frequencies. The DAC practically samples the output of the wavegenerator again, which can cause some aliasing effects.

DAC Update Rate = Clock frequency / 17

VII. Simulation results

Simulations were performed to verify the component at different stages. The results are shown in Fig. 11.

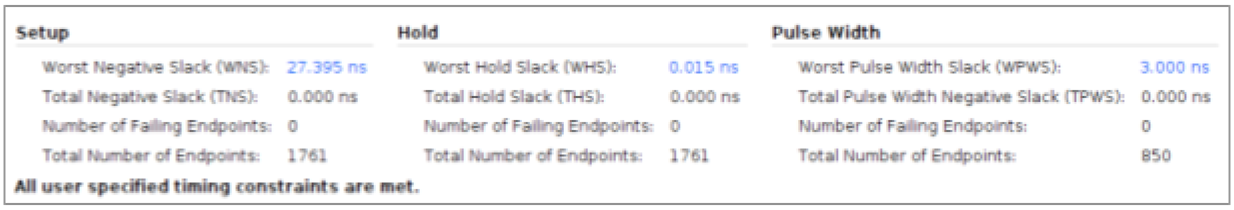
Different settings like step size, prescaler ratio, amplitude and offset were changed in the course of the test, as can be seen from the figure. The CORDIC output is seen as a filled waveform between the channel 1 and channel 2 outputs, when channel 2 is enabled. This is because the CORDIC output is time-multiplexed, and switches between the two channels in every cycle.



Name	Slice LUTs (20800)	Slice Registers (41600)	Slice (8150)	LUT as Logic (20800)	Bonded IOB (106)	BUFGCTRL (32)	MMCME2_ADV (5)
toplevel	715	844	266	715	53	2	1
clk_wizard (clk_wiz_0)	0	0	0	0	0	2	1
dac_intf_inst (dac_intf)	12	29	12	12	0	0	0
debouncer_left (debouncer)	15	21	14	15	0	0	0
debouncer_reset (debouncer_0)	9	21	11	9	0	0	0
debouncer_right (debouncer_1)	8	21	9	8	0	0	0
debouncer_select (debouncer_2)	8	21	9	8	0	0	0
seven_seg_intf_inst (seven_segment_intf)	62	19	28	62	0	0	0
wavegen_inst (wavegen)	590	602	185	590	0	0	0
cordic_module (cordic)	530	517	160	530	0	0	0

Resource	Utilization	Available	Utilization %
LUT	715	20800	3.44
FF	844	41600	2.03
IO	53	106	50.00
MMCM	1	5	20.00

Fig. 13: Hierarchical resource utilization (left) and summary (right)



Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	27.395 ns	Worst Hold Slack (WHS):	0.015 ns	Worst Pulse Width Slack (WPWS):	3.000 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	1761	Total Number of Endpoints:	1761	Total Number of Endpoints:	850

All user specified timing constraints are met.

Fig. 15: Timing analysis

The n-channel wavegenerator was also tested for N=4. The switching behavior between the channels can be clearly observed. It can also be observed that on increasing the number of channels the waveform starts to become less smooth because the output sampling frequency has become ¼ times the maximum frequency.

VIII. Implementation results

The design was implemented on the basys3 FPGA board. The PMOD DA2 DAC used can only operate at a maximum frequency of 30MHz. So, our design was synthesized for 30MHz. The clock wizard IP was used to generate the 30MHz clock from the 100MHz clock available on the board.

A. *Resource Utilization*

The resource utilization is shown in Fig. 13. It can be observed that the cordic module takes up about 75% of the resources. The additional components in the wave generator consume about 8%. By multiplexing the CORDIC module, we are able to utilize the same hardware without adding significant overhead.

B. *Timing and maximum frequency*

The worst negative slack (WNS) is reported as 27.4ns as can be seen from Fig. 14. Our clock frequency is 30Mhz. Using this, we can find the maximum frequency at which the circuit can operate.

$$T_{min} = T_{clk} - WNS = 33.33 - 27.40 = 5.93ns$$

$$f_{max} = 1/T_{min} = 168.63\,MHz$$

However, the throughput of the whole system is limited by the DAC which can only send one sample every 17 cycles. The DAC update rate is

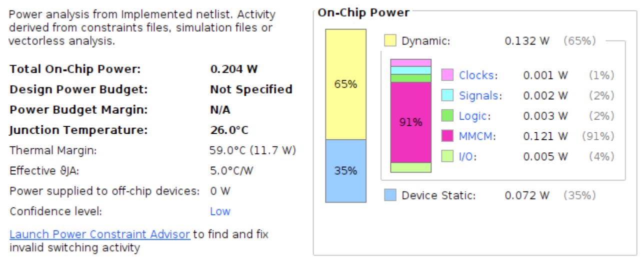
$$f_{DAC} = 30\,/\,17 = 1.77MHz$$

The frequency of the output waveform is limited by this update rate. The maximum output frequency depends on the number of samples required per cycle. If this is two along the lines of the nyquist sampling theorem, the maximum frequency is 1.77 / 2 = 882.85 kHz. However, a minimum of 10 samples is recommended to get a smooth waveform bringing the frequency down to 176.4 kHz.

The input setup and hold time and detailed timing analysis are given in the end.

C. *Power*

The net power consumption is 0.204W. About 65% of the power dissipation is dynamic with the MMCM consuming 91% of this. Logic only accounts of 4% of the dynamic delay.



Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

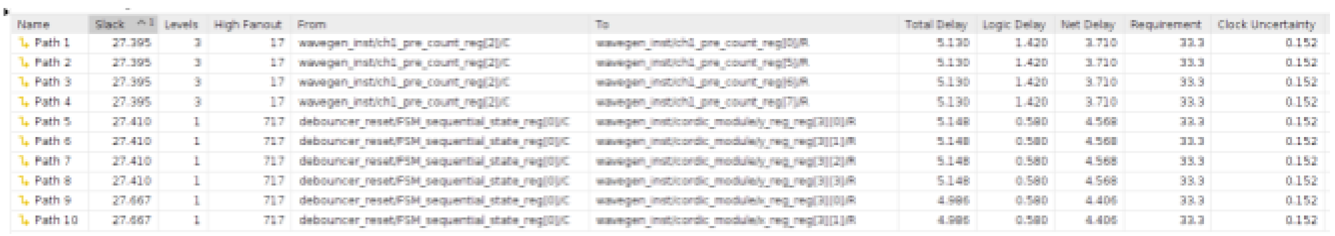
Total On-Chip Power:	**0.204 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**26.0°C**
Thermal Margin:	59.0°C (11.7 W)
Effective ϑJA:	5.0°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

Launch Power Constraint Advisor to find and fix invalid switching activity

On-Chip Power

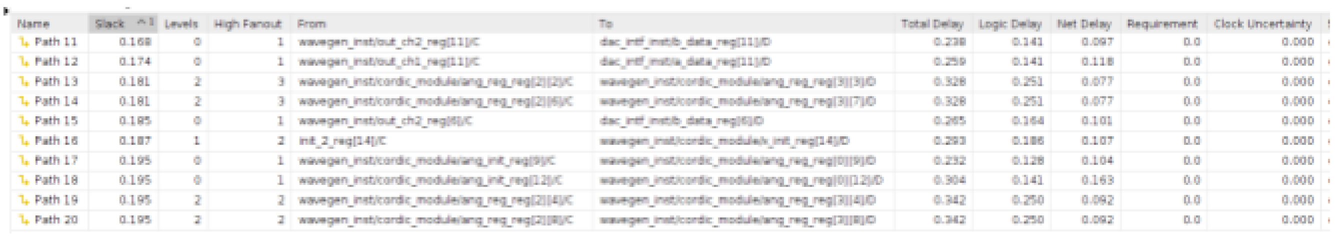
🟨	Dynamic:	0.132 W	(65%)
🟪	Clocks:	0.001 W	(1%)
🟦	Signals:	0.002 W	(2%)
🟩	Logic:	0.003 W	(2%)
🟪	MMCM:	0.121 W	(91%)
🟩	I/O:	0.005 W	(4%)
🟦	Device Static:	0.072 W	(35%)

Fig. 16: Power analysis summary



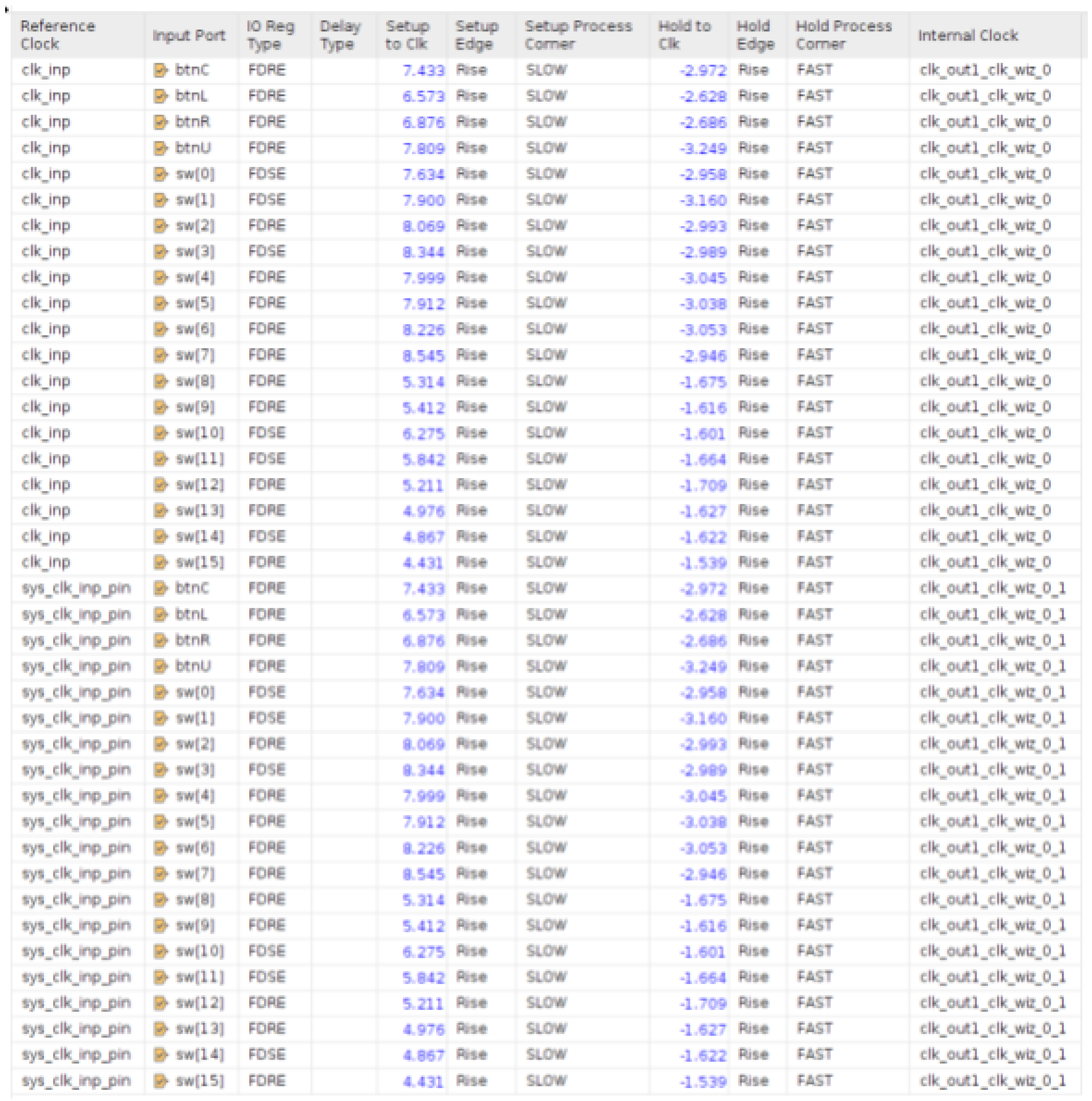
Name	Slack	Levels	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	Clock Uncertainty
Path 1	27.395	3	17	wavegen_inst/ch1_pre_count_reg[2]/C	wavegen_inst/ch1_pre_count_reg[0]/R	5.130	1.420	3.710	33.3	0.152
Path 2	27.395	3	17	wavegen_inst/ch1_pre_count_reg[2]/C	wavegen_inst/ch1_pre_count_reg[5]/R	5.130	1.420	3.710	33.3	0.152
Path 3	27.395	3	17	wavegen_inst/ch1_pre_count_reg[2]/C	wavegen_inst/ch1_pre_count_reg[6]/R	5.130	1.420	3.710	33.3	0.152
Path 4	27.395	3	17	wavegen_inst/ch1_pre_count_reg[2]/C	wavegen_inst/ch1_pre_count_reg[7]/R	5.130	1.420	3.710	33.3	0.152
Path 5	27.410	1	717	debouncer_reset/FSM_sequential_state_reg[0]/C	wavegen_inst/cordic_module/y_reg_reg[3][0]/R	5.148	0.580	4.568	33.3	0.152
Path 6	27.410	1	717	debouncer_reset/FSM_sequential_state_reg[0]/C	wavegen_inst/cordic_module/y_reg_reg[3][1]/R	5.148	0.580	4.568	33.3	0.152
Path 7	27.410	1	717	debouncer_reset/FSM_sequential_state_reg[0]/C	wavegen_inst/cordic_module/y_reg_reg[3][2]/R	5.148	0.580	4.568	33.3	0.152
Path 8	27.410	1	717	debouncer_reset/FSM_sequential_state_reg[0]/C	wavegen_inst/cordic_module/y_reg_reg[3][3]/R	5.148	0.580	4.568	33.3	0.152
Path 9	27.667	1	717	debouncer_reset/FSM_sequential_state_reg[0]/C	wavegen_inst/cordic_module/x_reg_reg[3][0]/R	4.986	0.580	4.406	33.3	0.152
Path 10	27.667	1	717	debouncer_reset/FSM_sequential_state_reg[0]/C	wavegen_inst/cordic_module/x_reg_reg[3][1]/R	4.986	0.580	4.406	33.3	0.152

Setup analysis



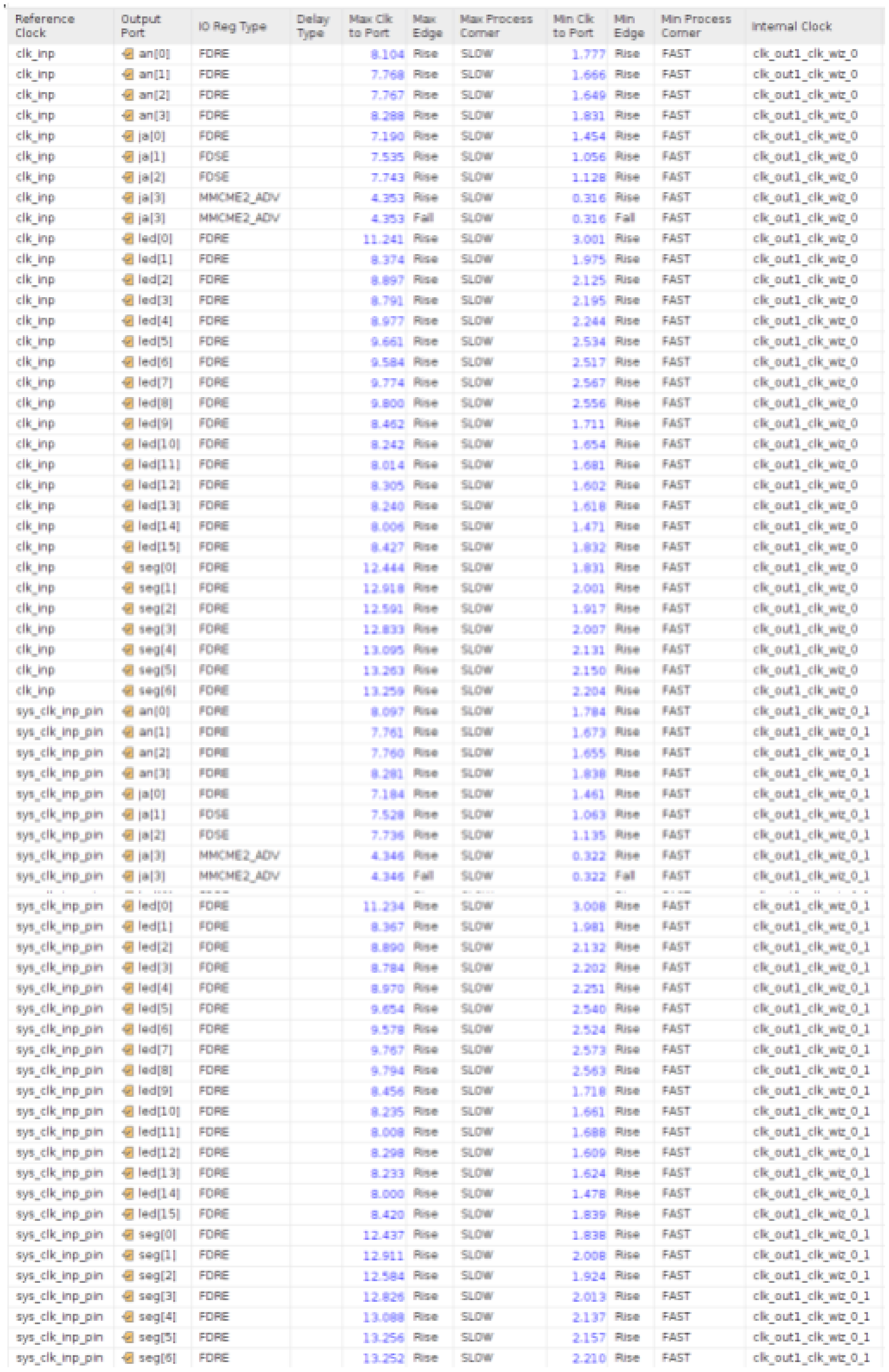
Name	Slack	Levels	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	Clock Uncertainty
Path 11	0.168	0	1	wavegen_inst/out_ch2_reg[11]/C	dac_intf_inst/b_data_reg[11]/D	0.238	0.141	0.097	0.0	0.000
Path 12	0.174	0	1	wavegen_inst/out_ch1_reg[11]/C	dac_intf_inst/a_data_reg[11]/D	0.259	0.141	0.118	0.0	0.000
Path 13	0.181	2	3	wavegen_inst/cordic_module/ang_reg_reg[2][2]/C	wavegen_inst/cordic_module/ang_reg_reg[3][3]/D	0.328	0.251	0.077	0.0	0.000
Path 14	0.181	2	3	wavegen_inst/cordic_module/ang_reg_reg[2][6]/C	wavegen_inst/cordic_module/ang_reg_reg[3][7]/D	0.328	0.251	0.077	0.0	0.000
Path 15	0.185	0	1	wavegen_inst/out_ch2_reg[6]/C	dac_intf_inst/b_data_reg[6]/D	0.265	0.164	0.101	0.0	0.000
Path 16	0.187	1	2	init_2_reg[14]/C	wavegen_inst/cordic_module/x_init_reg[14]/D	0.293	0.186	0.107	0.0	0.000
Path 17	0.195	0	1	wavegen_inst/cordic_module/ang_init_reg[9]/C	wavegen_inst/cordic_module/ang_reg_reg[0][9]/D	0.232	0.128	0.104	0.0	0.000
Path 18	0.195	0	1	wavegen_inst/cordic_module/ang_init_reg[12]/C	wavegen_inst/cordic_module/ang_reg_reg[0][12]/D	0.304	0.141	0.163	0.0	0.000
Path 19	0.195	2	2	wavegen_inst/cordic_module/ang_reg_reg[2][4]/C	wavegen_inst/cordic_module/ang_reg_reg[3][4]/D	0.342	0.250	0.092	0.0	0.000
Path 20	0.195	2	2	wavegen_inst/cordic_module/ang_reg_reg[2][8]/C	wavegen_inst/cordic_module/ang_reg_reg[3][8]/D	0.342	0.250	0.092	0.0	0.000

Hold analysis



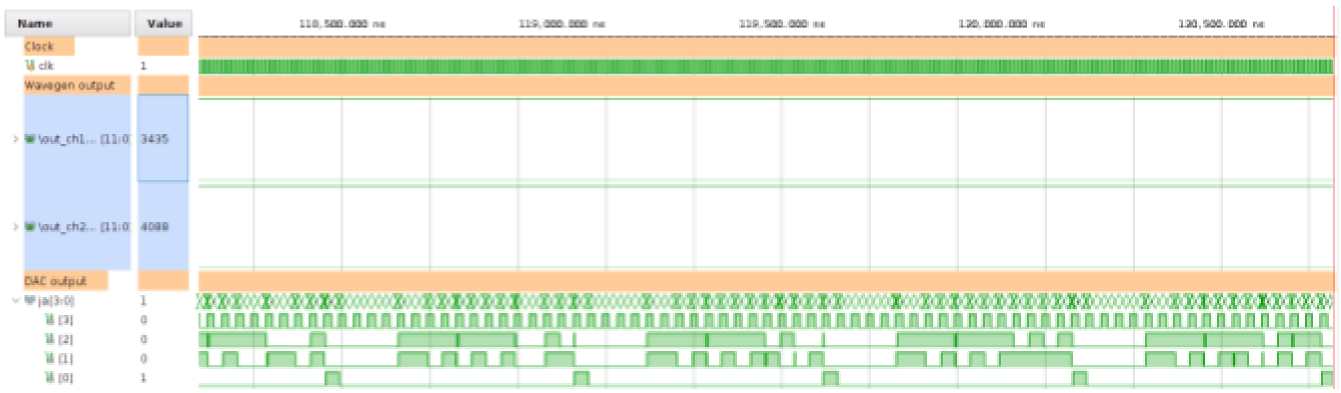
Reference Clock	Input Port	IO Reg Type	Delay Type	Setup to Clk	Setup Edge	Setup Process Corner	Hold to Clk	Hold Edge	Hold Process Corner	Internal Clock
clk_inp	btnC	FDRE		7.433	Rise	SLOW	-2.972	Rise	FAST	clk_out1_clk_wiz_0
clk_inp	btnL	FDRE		6.573	Rise	SLOW	-2.628	Rise	FAST	clk_out1_clk_wiz_0
clk_inp	btnR	FDRE		6.876	Rise	SLOW	-2.686	Rise	FAST	clk_out1_clk_wiz_0
clk_inp	btnU	FDRE		7.809	Rise	SLOW	-3.249	Rise	FAST	clk_out1_clk_wiz_0
clk_inp	sw[0]	FDSE		7.634	Rise	SLOW	-2.958	Rise	FAST	clk_out1_clk_wiz_0
clk_inp	sw[1]	FDSE		7.900	Rise	SLOW	-3.160	Rise	FAST	clk_out1_clk_wiz_0
clk_inp	sw[2]	FDRE		8.069	Rise	SLOW	-2.993	Rise	FAST	clk_out1_clk_wiz_0
clk_inp	sw[3]	FDSE		8.344	Rise	SLOW	-2.989	Rise	FAST	clk_out1_clk_wiz_0
clk_inp	sw[4]	FDRE		7.999	Rise	SLOW	-3.045	Rise	FAST	clk_out1_clk_wiz_0
clk_inp	sw[5]	FDRE		7.912	Rise	SLOW	-3.038	Rise	FAST	clk_out1_clk_wiz_0
clk_inp	sw[6]	FDRE		8.226	Rise	SLOW	-3.053	Rise	FAST	clk_out1_clk_wiz_0
clk_inp	sw[7]	FDRE		8.545	Rise	SLOW	-2.946	Rise	FAST	clk_out1_clk_wiz_0
clk_inp	sw[8]	FDRE		5.314	Rise	SLOW	-1.675	Rise	FAST	clk_out1_clk_wiz_0
clk_inp	sw[9]	FDRE		5.412	Rise	SLOW	-1.616	Rise	FAST	clk_out1_clk_wiz_0
clk_inp	sw[10]	FDSE		6.275	Rise	SLOW	-1.601	Rise	FAST	clk_out1_clk_wiz_0
clk_inp	sw[11]	FDSE		5.842	Rise	SLOW	-1.664	Rise	FAST	clk_out1_clk_wiz_0
clk_inp	sw[12]	FDRE		5.211	Rise	SLOW	-1.709	Rise	FAST	clk_out1_clk_wiz_0
clk_inp	sw[13]	FDRE		4.976	Rise	SLOW	-1.627	Rise	FAST	clk_out1_clk_wiz_0
clk_inp	sw[14]	FDSE		4.867	Rise	SLOW	-1.622	Rise	FAST	clk_out1_clk_wiz_0
clk_inp	sw[15]	FDRE		4.431	Rise	SLOW	-1.539	Rise	FAST	clk_out1_clk_wiz_0
sys_clk_inp_pin	btnC	FDRE		7.433	Rise	SLOW	-2.972	Rise	FAST	clk_out1_clk_wiz_0_1
sys_clk_inp_pin	btnL	FDRE		6.573	Rise	SLOW	-2.628	Rise	FAST	clk_out1_clk_wiz_0_1
sys_clk_inp_pin	btnR	FDRE		6.876	Rise	SLOW	-2.686	Rise	FAST	clk_out1_clk_wiz_0_1
sys_clk_inp_pin	btnU	FDRE		7.809	Rise	SLOW	-3.249	Rise	FAST	clk_out1_clk_wiz_0_1
sys_clk_inp_pin	sw[0]	FDSE		7.634	Rise	SLOW	-2.958	Rise	FAST	clk_out1_clk_wiz_0_1
sys_clk_inp_pin	sw[1]	FDSE		7.900	Rise	SLOW	-3.160	Rise	FAST	clk_out1_clk_wiz_0_1
sys_clk_inp_pin	sw[2]	FDRE		8.069	Rise	SLOW	-2.993	Rise	FAST	clk_out1_clk_wiz_0_1
sys_clk_inp_pin	sw[3]	FDSE		8.344	Rise	SLOW	-2.989	Rise	FAST	clk_out1_clk_wiz_0_1
sys_clk_inp_pin	sw[4]	FDRE		7.999	Rise	SLOW	-3.045	Rise	FAST	clk_out1_clk_wiz_0_1
sys_clk_inp_pin	sw[5]	FDRE		7.912	Rise	SLOW	-3.038	Rise	FAST	clk_out1_clk_wiz_0_1
sys_clk_inp_pin	sw[6]	FDRE		8.226	Rise	SLOW	-3.053	Rise	FAST	clk_out1_clk_wiz_0_1
sys_clk_inp_pin	sw[7]	FDRE		8.545	Rise	SLOW	-2.946	Rise	FAST	clk_out1_clk_wiz_0_1
sys_clk_inp_pin	sw[8]	FDRE		5.314	Rise	SLOW	-1.675	Rise	FAST	clk_out1_clk_wiz_0_1
sys_clk_inp_pin	sw[9]	FDRE		5.412	Rise	SLOW	-1.616	Rise	FAST	clk_out1_clk_wiz_0_1
sys_clk_inp_pin	sw[10]	FDSE		6.275	Rise	SLOW	-1.601	Rise	FAST	clk_out1_clk_wiz_0_1
sys_clk_inp_pin	sw[11]	FDSE		5.842	Rise	SLOW	-1.664	Rise	FAST	clk_out1_clk_wiz_0_1
sys_clk_inp_pin	sw[12]	FDRE		5.211	Rise	SLOW	-1.709	Rise	FAST	clk_out1_clk_wiz_0_1
sys_clk_inp_pin	sw[13]	FDRE		4.976	Rise	SLOW	-1.627	Rise	FAST	clk_out1_clk_wiz_0_1
sys_clk_inp_pin	sw[14]	FDSE		4.867	Rise	SLOW	-1.622	Rise	FAST	clk_out1_clk_wiz_0_1
sys_clk_inp_pin	sw[15]	FDRE		4.431	Rise	SLOW	-1.539	Rise	FAST	clk_out1_clk_wiz_0_1

Setup and times for inputs

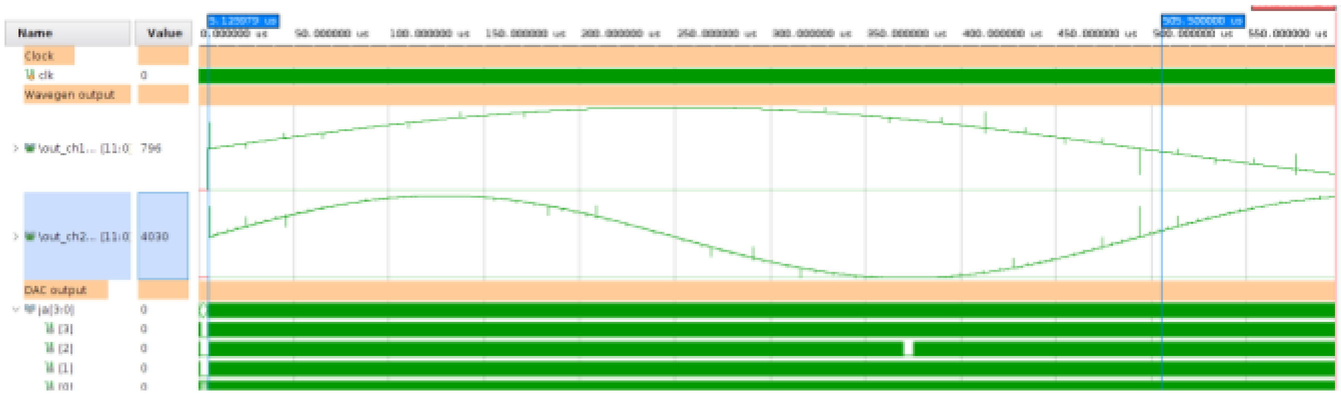


Reference Clock	Output Port	IO Reg Type	Delay Type	Max Clk to Port	Max Edge	Max Process Corner	Min Clk to Port	Min Edge	Min Process Corner	Internal Clock
clk_inp	an[0]	FDRE		8.104	Rise	SLOW	1.777	Rise	FAST	clk_out1_clk_wiz_0
clk_inp	an[1]	FDRE		7.768	Rise	SLOW	1.666	Rise	FAST	clk_out1_clk_wiz_0
clk_inp	an[2]	FDRE		7.767	Rise	SLOW	1.649	Rise	FAST	clk_out1_clk_wiz_0
clk_inp	an[3]	FDRE		8.288	Rise	SLOW	1.831	Rise	FAST	clk_out1_clk_wiz_0
clk_inp	ja[0]	FDRE		7.190	Rise	SLOW	1.454	Rise	FAST	clk_out1_clk_wiz_0
clk_inp	ja[1]	FDSE		7.535	Rise	SLOW	1.056	Rise	FAST	clk_out1_clk_wiz_0
clk_inp	ja[2]	FDSE		7.743	Rise	SLOW	1.128	Rise	FAST	clk_out1_clk_wiz_0
clk_inp	ja[3]	MMCME2_ADV		4.353	Rise	SLOW	0.316	Rise	FAST	clk_out1_clk_wiz_0
clk_inp	ja[3]	MMCME2_ADV		4.353	Fall	SLOW	0.316	Fall	FAST	clk_out1_clk_wiz_0
clk_inp	led[0]	FDRE		11.241	Rise	SLOW	3.001	Rise	FAST	clk_out1_clk_wiz_0
clk_inp	led[1]	FDRE		8.374	Rise	SLOW	1.975	Rise	FAST	clk_out1_clk_wiz_0
clk_inp	led[2]	FDRE		8.897	Rise	SLOW	2.125	Rise	FAST	clk_out1_clk_wiz_0
clk_inp	led[3]	FDRE		8.791	Rise	SLOW	2.195	Rise	FAST	clk_out1_clk_wiz_0
clk_inp	led[4]	FDRE		8.977	Rise	SLOW	2.244	Rise	FAST	clk_out1_clk_wiz_0
clk_inp	led[5]	FDRE		9.661	Rise	SLOW	2.534	Rise	FAST	clk_out1_clk_wiz_0
clk_inp	led[6]	FDRE		9.584	Rise	SLOW	2.517	Rise	FAST	clk_out1_clk_wiz_0
clk_inp	led[7]	FDRE		9.774	Rise	SLOW	2.567	Rise	FAST	clk_out1_clk_wiz_0
clk_inp	led[8]	FDRE		9.800	Rise	SLOW	2.556	Rise	FAST	clk_out1_clk_wiz_0
clk_inp	led[9]	FDRE		8.462	Rise	SLOW	1.711	Rise	FAST	clk_out1_clk_wiz_0
clk_inp	led[10]	FDRE		8.242	Rise	SLOW	1.654	Rise	FAST	clk_out1_clk_wiz_0
clk_inp	led[11]	FDRE		8.014	Rise	SLOW	1.681	Rise	FAST	clk_out1_clk_wiz_0
clk_inp	led[12]	FDRE		8.305	Rise	SLOW	1.602	Rise	FAST	clk_out1_clk_wiz_0
clk_inp	led[13]	FDRE		8.240	Rise	SLOW	1.618	Rise	FAST	clk_out1_clk_wiz_0
clk_inp	led[14]	FDRE		8.006	Rise	SLOW	1.471	Rise	FAST	clk_out1_clk_wiz_0
clk_inp	led[15]	FDRE		8.427	Rise	SLOW	1.832	Rise	FAST	clk_out1_clk_wiz_0
clk_inp	seg[0]	FDRE		12.444	Rise	SLOW	1.831	Rise	FAST	clk_out1_clk_wiz_0
clk_inp	seg[1]	FDRE		12.918	Rise	SLOW	2.001	Rise	FAST	clk_out1_clk_wiz_0
clk_inp	seg[2]	FDRE		12.591	Rise	SLOW	1.917	Rise	FAST	clk_out1_clk_wiz_0
clk_inp	seg[3]	FDRE		12.833	Rise	SLOW	2.007	Rise	FAST	clk_out1_clk_wiz_0
clk_inp	seg[4]	FDRE		13.095	Rise	SLOW	2.131	Rise	FAST	clk_out1_clk_wiz_0
clk_inp	seg[5]	FDRE		13.263	Rise	SLOW	2.150	Rise	FAST	clk_out1_clk_wiz_0
clk_inp	seg[6]	FDRE		13.259	Rise	SLOW	2.204	Rise	FAST	clk_out1_clk_wiz_0
sys_clk_inp_pin	an[0]	FDRE		8.097	Rise	SLOW	1.784	Rise	FAST	clk_out1_clk_wiz_0_1
sys_clk_inp_pin	an[1]	FDRE		7.761	Rise	SLOW	1.673	Rise	FAST	clk_out1_clk_wiz_0_1
sys_clk_inp_pin	an[2]	FDRE		7.760	Rise	SLOW	1.655	Rise	FAST	clk_out1_clk_wiz_0_1
sys_clk_inp_pin	an[3]	FDRE		8.281	Rise	SLOW	1.838	Rise	FAST	clk_out1_clk_wiz_0_1
sys_clk_inp_pin	ja[0]	FDRE		7.184	Rise	SLOW	1.461	Rise	FAST	clk_out1_clk_wiz_0_1
sys_clk_inp_pin	ja[1]	FDSE		7.528	Rise	SLOW	1.063	Rise	FAST	clk_out1_clk_wiz_0_1
sys_clk_inp_pin	ja[2]	FDSE		7.736	Rise	SLOW	1.135	Rise	FAST	clk_out1_clk_wiz_0_1
sys_clk_inp_pin	ja[3]	MMCME2_ADV		4.346	Rise	SLOW	0.322	Rise	FAST	clk_out1_clk_wiz_0_1
sys_clk_inp_pin	ja[3]	MMCME2_ADV		4.346	Fall	SLOW	0.322	Fall	FAST	clk_out1_clk_wiz_0_1
sys_clk_inp_pin	led[0]	FDRE		11.234	Rise	SLOW	3.008	Rise	FAST	clk_out1_clk_wiz_0_1
sys_clk_inp_pin	led[1]	FDRE		8.367	Rise	SLOW	1.981	Rise	FAST	clk_out1_clk_wiz_0_1
sys_clk_inp_pin	led[2]	FDRE		8.890	Rise	SLOW	2.132	Rise	FAST	clk_out1_clk_wiz_0_1
sys_clk_inp_pin	led[3]	FDRE		8.784	Rise	SLOW	2.202	Rise	FAST	clk_out1_clk_wiz_0_1
sys_clk_inp_pin	led[4]	FDRE		8.970	Rise	SLOW	2.251	Rise	FAST	clk_out1_clk_wiz_0_1
sys_clk_inp_pin	led[5]	FDRE		9.654	Rise	SLOW	2.540	Rise	FAST	clk_out1_clk_wiz_0_1
sys_clk_inp_pin	led[6]	FDRE		9.578	Rise	SLOW	2.524	Rise	FAST	clk_out1_clk_wiz_0_1
sys_clk_inp_pin	led[7]	FDRE		9.767	Rise	SLOW	2.573	Rise	FAST	clk_out1_clk_wiz_0_1
sys_clk_inp_pin	led[8]	FDRE		9.794	Rise	SLOW	2.563	Rise	FAST	clk_out1_clk_wiz_0_1
sys_clk_inp_pin	led[9]	FDRE		8.456	Rise	SLOW	1.718	Rise	FAST	clk_out1_clk_wiz_0_1
sys_clk_inp_pin	led[10]	FDRE		8.235	Rise	SLOW	1.661	Rise	FAST	clk_out1_clk_wiz_0_1
sys_clk_inp_pin	led[11]	FDRE		8.008	Rise	SLOW	1.688	Rise	FAST	clk_out1_clk_wiz_0_1
sys_clk_inp_pin	led[12]	FDRE		8.298	Rise	SLOW	1.609	Rise	FAST	clk_out1_clk_wiz_0_1
sys_clk_inp_pin	led[13]	FDRE		8.233	Rise	SLOW	1.624	Rise	FAST	clk_out1_clk_wiz_0_1
sys_clk_inp_pin	led[14]	FDRE		8.000	Rise	SLOW	1.478	Rise	FAST	clk_out1_clk_wiz_0_1
sys_clk_inp_pin	led[15]	FDRE		8.420	Rise	SLOW	1.839	Rise	FAST	clk_out1_clk_wiz_0_1
sys_clk_inp_pin	seg[0]	FDRE		12.437	Rise	SLOW	1.838	Rise	FAST	clk_out1_clk_wiz_0_1
sys_clk_inp_pin	seg[1]	FDRE		12.911	Rise	SLOW	2.008	Rise	FAST	clk_out1_clk_wiz_0_1
sys_clk_inp_pin	seg[2]	FDRE		12.584	Rise	SLOW	1.924	Rise	FAST	clk_out1_clk_wiz_0_1
sys_clk_inp_pin	seg[3]	FDRE		12.826	Rise	SLOW	2.013	Rise	FAST	clk_out1_clk_wiz_0_1
sys_clk_inp_pin	seg[4]	FDRE		13.088	Rise	SLOW	2.137	Rise	FAST	clk_out1_clk_wiz_0_1
sys_clk_inp_pin	seg[5]	FDRE		13.256	Rise	SLOW	2.157	Rise	FAST	clk_out1_clk_wiz_0_1
sys_clk_inp_pin	seg[6]	FDRE		13.252	Rise	SLOW	2.210	Rise	FAST	clk_out1_clk_wiz_0_1

Output clock-to-out times



Post-implementation timing simulation results (zoomed in for DAC output)



Post-implementation timing simulation results