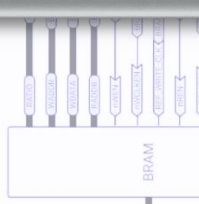
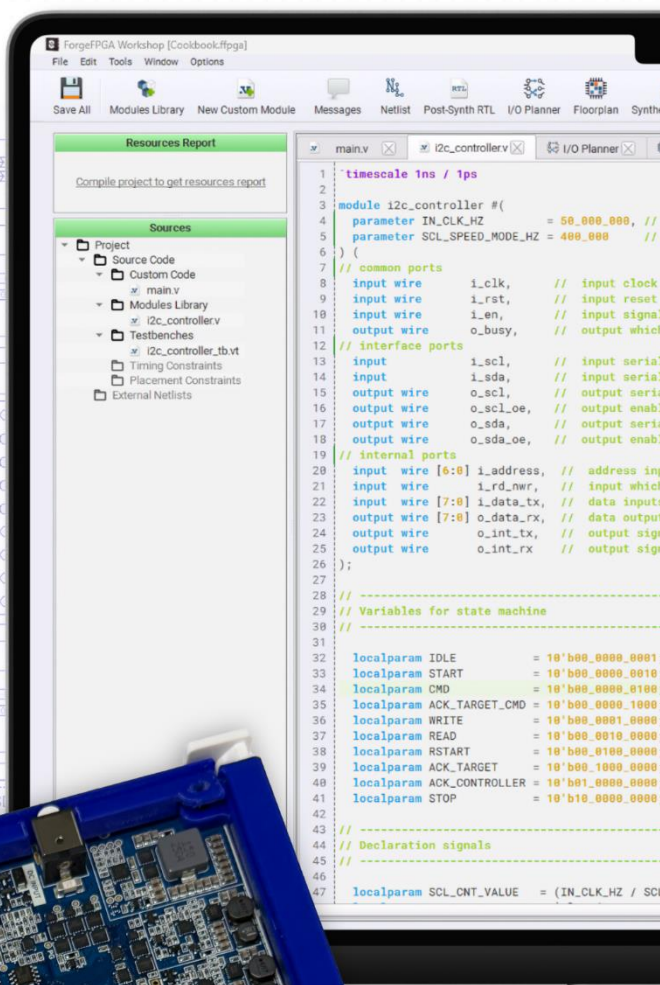
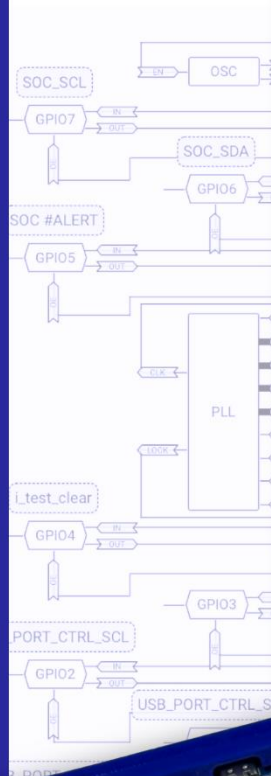
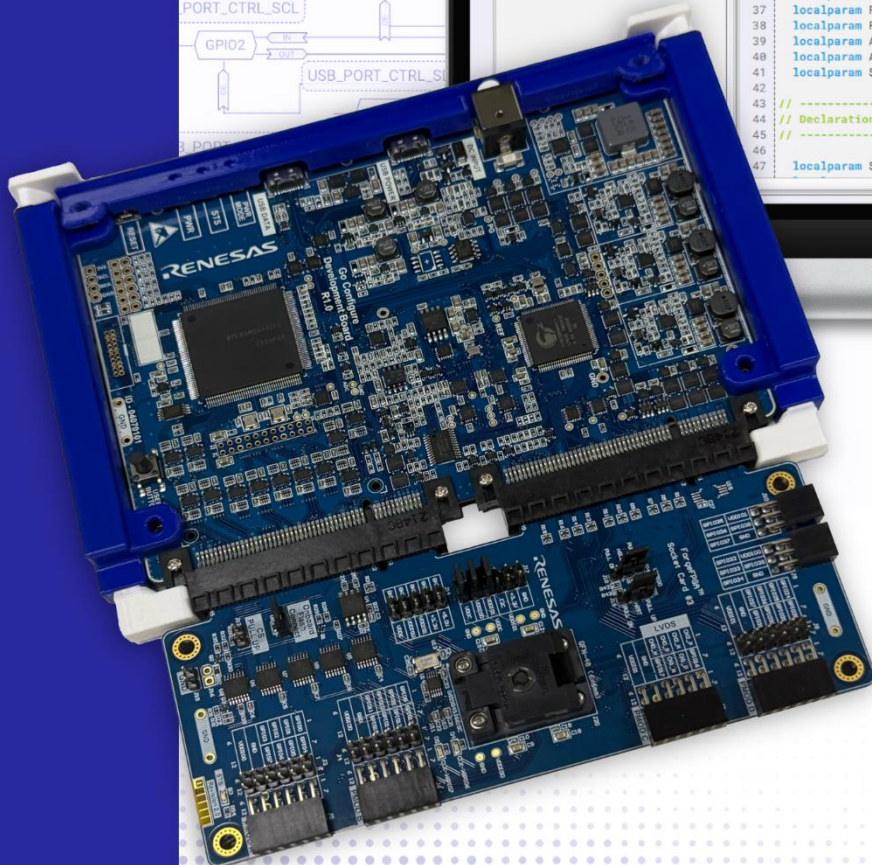


RENESAS

The ForgeFPGA™ Cookbook



Contents

Contents	1
1. Introduction	3
2. Design Techniques	3
2.1 GPIO Output Enable Usage	3
2.2 Oscillator Usage	4
2.3 PLL Usage	9
2.4 PLL Startup Time Calculation	13
2.5 Configuration from MCU	14
3. RTL Coding Guide	17
3.1 Asynchronous Reset	17
3.2 Flip-Flop Types and Control Signals	19
3.3 Verilog Coding Guidelines	21
3.4 Shift Registers (SRL)	23
3.5 Finite State Machine	26
4. Module Library	29
4.1 Communication	29
4.1.1 Simple I ² C Controller	29
4.1.2 Simple I ² C Target	32
4.1.3 I ² S Controller (Receiver)	34
4.1.4 I ² S Controller (Transmitter)	36
4.1.5 I ² S Target (Receiver)	38
4.1.6 I ² S Target (Transmitter)	40
4.1.7 Parallel to Serial Interface	42
4.1.8 SPI Controller	44
4.1.9 SPI Target	47
4.1.10 UART Receiver	50
4.1.11 UART Transmitter	53
4.1.12 UART Transceiver	56
4.2 Mathematics	59
4.2.1 Division RDA	59
4.2.2 Division NRDA	61
4.2.3 Division RSA	63
4.2.4 Serial CRC	65
4.2.5 Hamming Encoder	67
4.2.6 Hamming Decoder	68
4.3 Memory	69
4.3.1 Synchronous FIFO	69
4.4 Periphery	71
4.4.1 Keyboard Scanner	71
4.4.2 Servo Controller	73
4.4.3 7-Segment Display Controller	75

4.4.4	Smart LED Controller	77
4.4.5	Stepper Motor Controller.....	80
4.5	Signal Processing.....	82
4.5.1	Breathing Controller	82
4.5.2	Frequency Detector	84
4.5.3	PWM	86
4.5.4	Quadrature Decoder	87
4.5.5	Ripple Counter	89
5.	Applications	91
5.1	SPI to 4-channel UART Converter	91
5.2	SerDes via LVDS (SDR)	95
5.3	SerDes via LVDS (DDR)	98
6.	Revision History	101

1. Introduction

This document describes how to create optimized designs with ForgeFPGA products. The ForgeFPGA Cookbook provides a comprehensive collection of design techniques, reusable modules library, and RTL coding best practices. In addition to foundational guidance, it includes practical application examples that demonstrate how to effectively combine these techniques and components. This approach helps you develop high-quality, resource-efficient designs, accelerating time-to-market while maximizing device capabilities.

2. Design Techniques

2.1 GPIO Output Enable Usage

GPIO pins in ForgeFPGA can operate either as digital inputs or digital outputs. Each GPIO has an associated Output Enable (OE) signal that controls the pin direction.

OE Signal Behavior

The I/O direction is determined inside the FPGA fabric by the value assigned to the OE signal:

OE value	Pin mode
1'b0	Input
1'b1	Output

When the OE signal is asserted (1'b1), the GPIO output driver is enabled, and the pin operates as an output.

When the OE signal is desasserted (1'b0), the output driver is disabled, and the pin operates as an input.

Example: Configure a Pin as an Output

```
(* top *) module top (  
    output out,  
    output out_oe  
);  
  
    assign out      = user_signal;  
    assign out_oe = 1'b1;  
  
endmodule
```

In this example, out_oe is tied to logic 1, permanently configuring the pin as a digital output.

2.2 Oscillator Usage

All ForgeFPGA devices have integrated oscillators (OSC) that provide an internal clock source for FPGA designs. Using the on-chip oscillator eliminates the need for an external clock generator in many applications, reducing both system cost and board complexity.

The oscillator can be used directly as a system clock source or as a reference clock for internal PLLs.

Oscillator Features

SLG47910

The on-chip oscillator provides:

- High frequency (50 MHz) operation mode
- Oscillator enable control through the OSC_EN signal
- OSC_READY status signal indicating that the oscillator output is stable

SLG47912, SLG47920, and SLG47921

These devices provide enhanced oscillator functionality:

- Two operating modes: high frequency (50 MHz) and low frequency (3 MHz)
- Oscillator enable control through the OSC_EN signal
- OSC_READY status signal indicating that the oscillator output is stable
- Optional routing of oscillator output to dedicated GPIO
- Two independent Oscillator Post Divider outputs
- Independent divider selection for each Post Divider output
- Clock division from 1 to 128 (step with power of 2)
- POSTDIV_OUT0_READY and POSTDIV_OUT1_READY status signals indicating stable divided clock outputs
- Oscillator Post Divider outputs can be routed into the FPGA Core as data signals

Configuring OSC on SLG47910

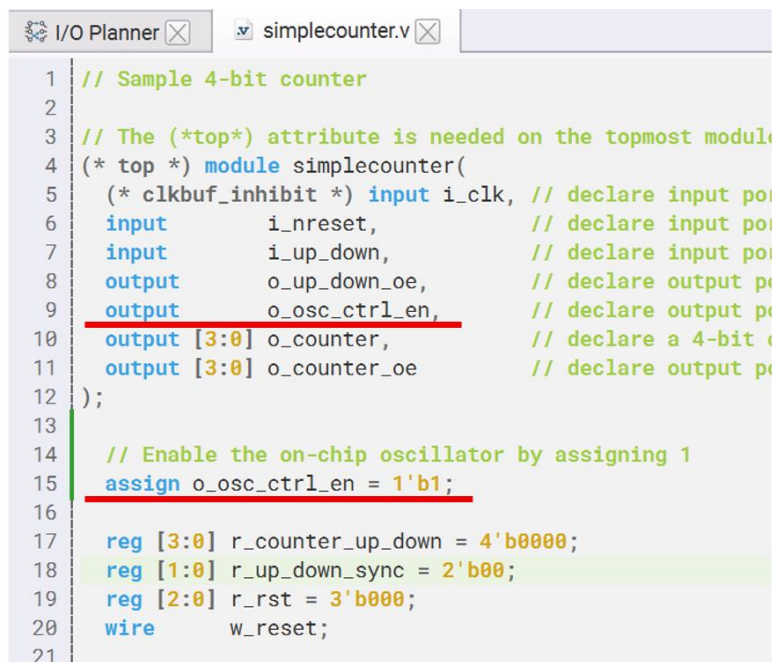
The oscillator on the SLG47910 is controlled through the OSC_EN signal.

To enable the oscillator:

1. Create an output control signal in the top-level HDL module.
2. Drive the signal HIGH (1'b1).
3. Connect the signal to the OSC_EN function in the I/O Planner.

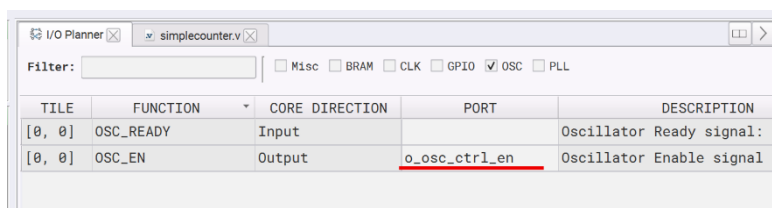
Enabling the Oscillator in HDL

```
// Enable the on-chip oscillator
assign o_osc_ctrl_en = 1'b1;
```



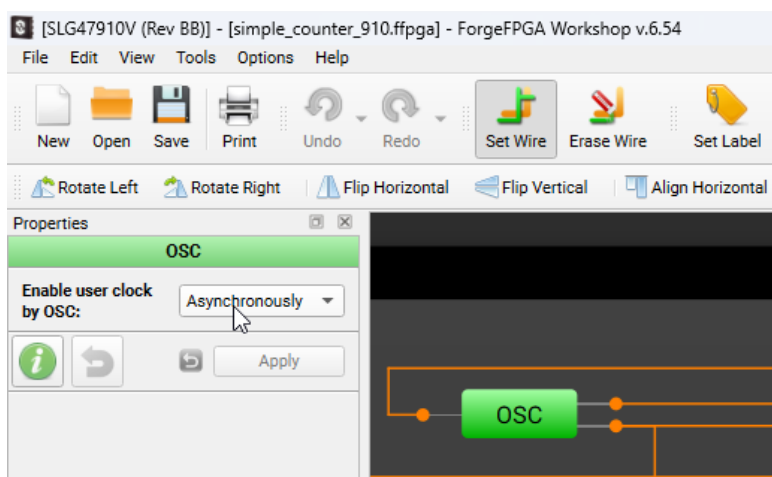
```
1 // Sample 4-bit counter
2
3 // The (*top*) attribute is needed on the topmost module
4 (* top *) module simplecounter(
5     (* clkbuf_inhibit *) input i_clk, // declare input port
6     input i_nreset, // declare input port
7     input i_up_down, // declare input port
8     output o_up_down_oe, // declare output port
9     output o_osc_ctrl_en, // declare output port
10    output [3:0] o_counter, // declare a 4-bit counter
11    output [3:0] o_counter_oe // declare output port
12 );
13
14 // Enable the on-chip oscillator by assigning 1
15 assign o_osc_ctrl_en = 1'b1;
16
17 reg [3:0] r_counter_up_down = 4'b0000;
18 reg [1:0] r_up_down_sync = 2'b00;
19 reg [2:0] r_rst = 3'b000;
20 wire w_reset;
21
```

Enabling the oscillator through the OSC_EN control signal



TILE	FUNCTION	CORE DIRECTION	PORT	DESCRIPTION
[0, 0]	OSC_READY	Input		Oscillator Ready signal
[0, 0]	OSC_EN	Output	<u>o_osc_ctrl_en</u>	Oscillator Enable signal

Connecting the HDL signal to the OSC_EN function in the I/O Planner



OSC configuration properties

The configuration property allows selection of the oscillator enable mode:

- Asynchronously – Oscillator follows the enable signal immediately.
- Synchronously – Oscillator enable is synchronized internally.

Configuring OSC and Post Divider on SLG47912, SLG47920, and SLG47921

These devices extend the oscillator functionality by adding:

- Two selectable oscillator frequencies
- Two independent Post Divider outputs
- Optional routing of divided clocks to dedicated resources
- Ready signals for oscillator and Post Divider outputs

The oscillator and Post Divider functions are controlled through user-defined HDL signals.

Example declaration:

```
output      o_osc_ctrl_en;
output      o_osc_ctrl_mode;

output      o_postdiv0_ctrl_en;
output [2:0] o_postdiv0_ctrl_sel;
```

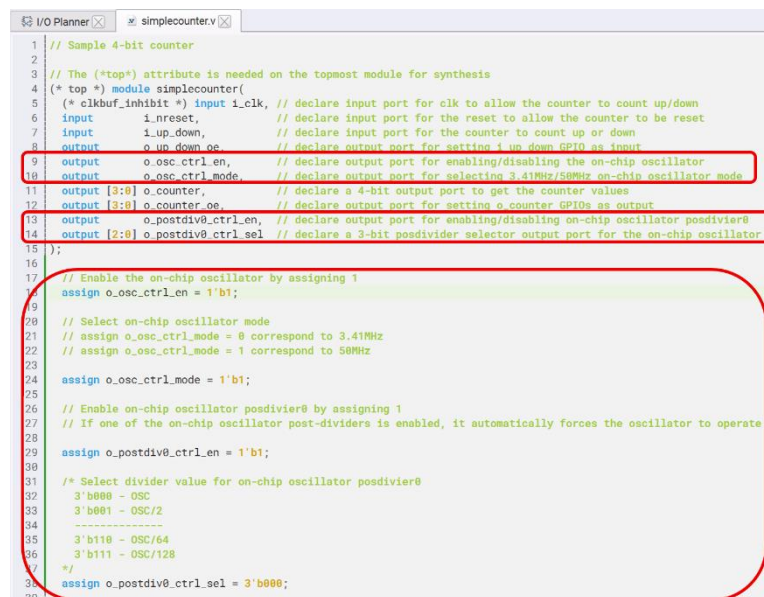
Example configuration:

```
// Enable oscillator
assign o_osc_ctrl_en = 1'b1;

// Select 50 MHz operating mode
assign o_osc_ctrl_mode = 1'b1;

// Enable Post Divider OUT0
assign o_postdiv0_ctrl_en = 1'b1;

// Divide by 1
assign o_postdiv0_ctrl_sel = 3'b000;
```



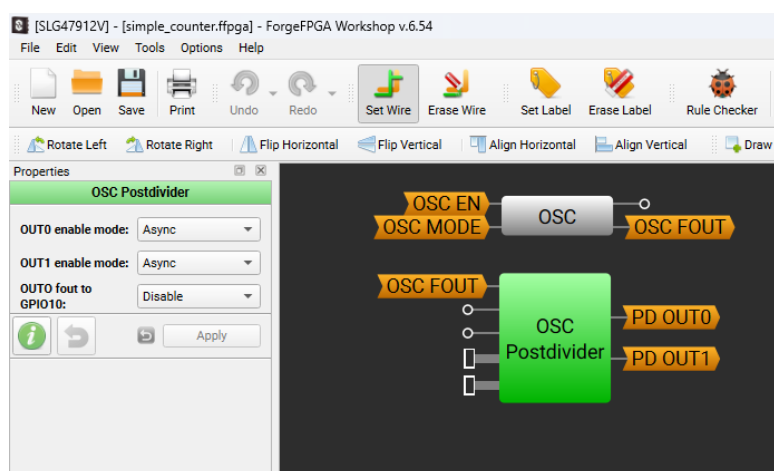
HDL signals used to control oscillator operation and Post Divider configuration

The control signals must be connected to the corresponding oscillator functions in the I/O Planner:

TILE	FUNCTION	CORE DIRECTION	PORT	DESCRIPTION
[0, 0]	OSC_EN	Output	o_osc_ctrl_en	Oscillator Enable signal
[0, 0]	OSC_MODE	Output	o_osc_ctrl_mode	Oscillator Mode Select signal
[1, 0]	OSC_POSTDIV_OUT0_AS_DATA	Input		Oscillator Postdivider Out0 input as data signal
[1, 0]	OSC_POSTDIV_OUT1_AS_DATA	Input		Oscillator Postdivider Out1 input as data signal
[0, 0]	OSC_RDY	Input		Oscillator Ready signal
[0, 0]	POSTDIV_OUT0_DIV_SEL[0]	Output	o_postdiv0_ctrl_sel[0]	Oscillator Postdivider Out0 divider select (bit 0)
[0, 0]	POSTDIV_OUT0_DIV_SEL[1]	Output	o_postdiv0_ctrl_sel[1]	Oscillator Postdivider Out0 divider select (bit 1)
[0, 0]	POSTDIV_OUT0_DIV_SEL[2]	Output	o_postdiv0_ctrl_sel[2]	Oscillator Postdivider Out0 divider select (bit 2)
[0, 0]	POSTDIV_OUT0_EN	Output	o_postdiv0_ctrl_en	Oscillator Postdivider Out0 Enable signal
[0, 0]	POSTDIV_OUT0_RDY	Input		Oscillator Postdivider Out0 Ready signal
[0, 0]	POSTDIV_OUT1_DIV_SEL[0]	Output		Oscillator Postdivider Out1 divider select (bit 0)
[0, 0]	POSTDIV_OUT1_DIV_SEL[1]	Output		Oscillator Postdivider Out1 divider select (bit 1)
[0, 0]	POSTDIV_OUT1_DIV_SEL[2]	Output		Oscillator Postdivider Out1 divider select (bit 2)
[0, 0]	POSTDIV_OUT1_EN	Output		Oscillator Postdivider Out1 Enable signal
[0, 0]	POSTDIV_OUT1_RDY	Input		Oscillator Postdivider Out1 Ready signal

Oscillator and Post Divider signal assignments in the I/O Planner

Post Divider Properties



OSC Post Divider configuration

The properties dialog allows configuration of:

- OUT0 enable mode
- OUT1 enable mode
- Optional routing of OUT0 to dedicated GPIO10

Configuration Restrictions

When using the oscillator and Post Divider circuitry, observe the following requirements.

Changing Oscillator Mode

Do not change OSC_MODE while the oscillator is enabled.

To change OSC_MODE:

1. Disable the oscillator (OSC_EN = 0).
2. Change OSC_MODE.
3. Wait at least 1 μ s.
4. Re-enable the oscillator (OSC_EN = 1).

Changing the oscillator frequency while it is running can cause output clock glitches.

Changing Post Divider Values

Do not change a Post Divider division factor while the corresponding output is enabled.

To change Post Divider values:

1. Disable the Post Divider output.
2. Change the divider value.
3. Re-enable the output.

This ensures proper divider operation and output clock stability.

Conclusion

The integrated oscillator available in the SLG47910, SLG47912, SLG47920, and SLG47921 provides a flexible and cost-effective clock source for FPGA designs. Beyond basic clock generation, the oscillator can serve as a PLL reference source, operate at multiple frequencies, and provide independently divided clock outputs through the integrated Post Divider circuitry. Combined with dedicated READY signals and configurable control interfaces, the oscillator subsystem enables reliable clock generation for a wide range of FPGA applications.

2.3 PLL Usage

The SLG47910, SLG47912, SLG47920, and SLG47921 include integrated Phase-Locked Loops (PLLs) that allow designers to generate application-specific clock frequencies without external clock-generation components. PLLs can multiply or divide a reference clock, reduce clock jitter, generate multiple operating frequencies from a single reference source.

The SLG47910 and SLG47912 contain one PLL, while the SLG47920 and SLG47921 provide two PLLs for more advanced multi-clock designs.

PLL Reference and Clock Outputs

The PLL receives a reference clock and generates one or more output clocks that can be distributed through the device's global clock network. The generated PLL clock outputs are routed through dedicated clock distribution resources for reliable low-skew clock delivery throughout the FPGA fabric. The PLL also provides a PLL_LOCK status signal to indicate when the output frequency has stabilized.

SLG47910

The device includes a single PLL with the following features:

- Internal 50 MHz oscillator clock source
- External reference clock input
- Single PLL clock output routed to the FPGA global clock network

SLG47912

The device includes a single PLL with the following features:

- Internal oscillator clock source
- External reference clock input
- LVDS differential clock source
- Two independent PLL clock outputs routable to the FPGA global clock network through CLK MUXes
- Optional routing of PLL fout0 to dedicated GPIO

SLG47920 and SLG47921

The SLG47920 and SLG47921 include two independent PLLs with the following features:

- Internal oscillator
- External clock inputs
- LVDS differential clock source
- PLL0 provides two clock outputs with individual post dividers for each clock output
- PLL1 provides one clock output
- Clock outputs routable to the FPGA global clock network through CLK MUXes
- Optional routing of PLL0 fout0 and PLL1 fout to dedicated GPIOs

PLL Frequency Calculation

The PLL first generates a high-frequency Voltage-Controlled Oscillator (VCO) clock and then divides it to produce the required output frequency.

The VCO Frequency is determined by:

$$F_{OUTVCO} = \frac{F_{REF} \times FBDIV}{REFDIV}$$

The PLL output frequency is calculated as:

$$F_{OUT} = \frac{F_{REF} \times FBDIV}{REFDIV \times POSTDIV0 \times PLL_POSTDIV1}$$

Where:

- FOUTVCO – PLL Voltage-Controlled Oscillator Output.
- FOUT – PLL clock output.
- FBDIV – Feedback Divider value (range 16 to 400).

- REFDIV – Reference divide value (1 to 63).
- POSTDIV0 – First stage of Post Divider is used to divide down the VCO Frequency before the FOUT clock output (range 1 to 7).
- POSTDIV1 – Second stage of Post Divider is used to divide down the VCO Frequency before the FOUT clock output (range 1 to 7).

PLL Configuration and Using

The PLL Configurator is available in ForgeFPGA Workshop and simplifies PLL configuration by automatically calculating valid divider settings and generating the required HDL code.

The PLL Configurator features:

- Calculates valid PLL_FBDIV, PLL_REFDIV, and PLL_POSTDIV values
- Verifies that PFD, VCO, and output frequencies are within device specifications
- Provides estimated operating parameters for the selected PLL configuration
- Generates Verilog code snippets that can be copied directly into the user design
- Automatically maps PLL configuration signals in the I/O Planner

PLL Configuration Flow

PLL Calculator tab

The PLL Calculator tab is used to create and validate PLL configurations.

1 Clock Options

Input Frequency, MHz	Required Output Frequency, MHz	Actual Output Frequency, MHz	PLL Output Clock
50.000000	85.000000	85.000000	☒ PLL0_fout0
	70.000000	70.833333	☒ PLL0_fout1

2 Allow Manual Adjustment

REFDIV [1-63]	FBDIV [16-400] *	POSTDIV0 [1-7] *	POSTDIV1 [1-7] *	PLL Output Clock
1	17	5	2	PLL0_fout0
		6	2	PLL0_fout1

Actual Output Frequency = (Input Frequency * FBDIV) / (REFDIV * POSTDIV0 * POSTDIV1)

3 PLL Properties

Enable mode	Async
PLL clock gating with LOCK	Gated
PLL fout to dedicated GPIO	Disabled
Clock source select	OSC
Bypass Mode	Disabled
Lock	Disabled
Ready Signal	Disabled

4 Clock Information

Input Period, ns	20.000000
Actual Output Period Fout0, ns	11.764706
Actual Output Period Fout1, ns	14.117647
Duty Cycle, %	[44.2:51.6]
VCO Frequency [500-1000], MHz	850.000000
PF0 Frequency [5-VCO/16], MHz	50.000000

5 Apply

PLL configurator

1. Clock requirements
2. Automatically calculated or manually changed divider parameters
3. PLL configuration properties
4. Clock operating information
5. Apply configuration

Configuration procedure:

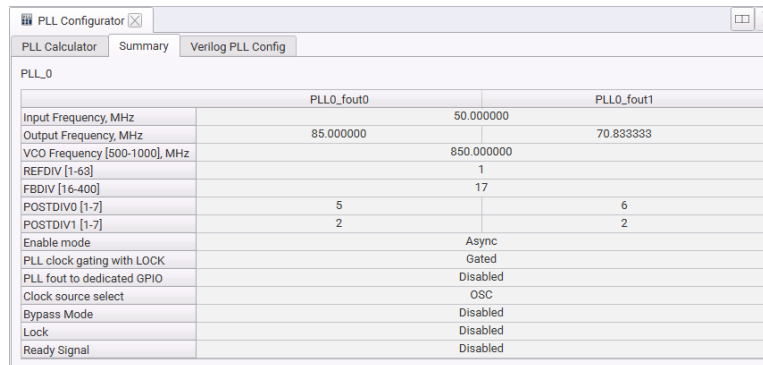
1. Configure clock requirements:
 - a. Enter the input clock frequency.
 - b. Select the PLL output clocks to enable.
 - c. Enter the required output frequencies.
2. Review calculated results:
 - Actual Output Frequency shows the achievable output frequency.
 - Divider parameters are calculated automatically or can be changed manually.
3. Configure PLL properties:
 - Enable Mode – Selects synchronous or asynchronous PLL enable pin behavior.
 - PLL Clock Gating with LOCK – Gates the output clock until the PLL achieves lock.
 - PLL FOUT to Dedicated GPIO – Routes a PLL output clock to a dedicated GPIO.
 - Clock Source Select – Selects the PLL reference clock source.
 - Bypass Mode – Bypasses the PLL and directly routes the reference clock to the output.
 - Lock – Enables PLL_LOCK signal generation.
 - Ready Signal – Enables PLL_READY signal generation.
4. Review operating information:
 - Input and output clock periods
 - Duty cycle

- VCO frequency
- PFD frequency

5. Click **Apply** to update the PLL configuration.

Summary tab

The **Summary** tab displays the applied PLL parameters and selected PLL settings in a consolidated view.

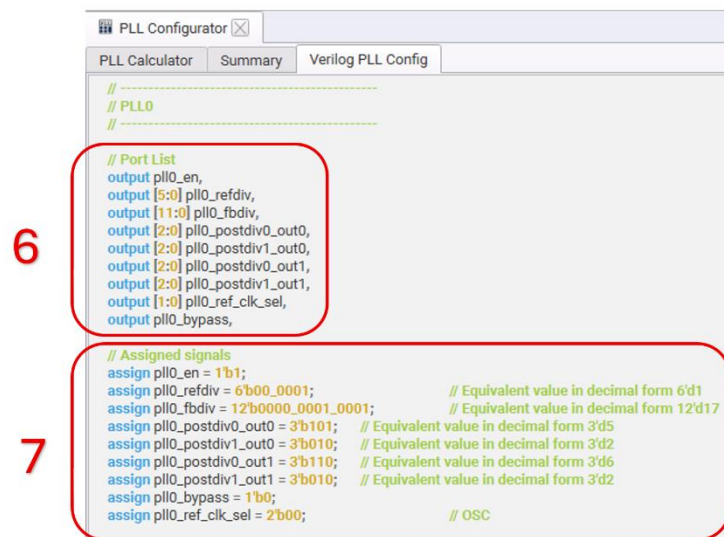


	PLL0_fout0	PLL0_fout1
Input Frequency, MHz	50.000000	
Output Frequency, MHz	85.000000	70.833333
VCO Frequency [500-1000], MHz	850.000000	
REFDIV [1-63]	1	
FBDIV [16-400]	17	
POSTDIV0 [1-7]	5	6
POSTDIV1 [1-7]	2	2
Enable mode	Async	
PLL clock gating with LOCK	Gated	
PLL fout to dedicated GPIO	Disabled	
Clock source select	OSC	
Bypass Mode	Disabled	
Lock	Disabled	
Ready Signal	Disabled	

PLL Configurator Summary tab

Verilog PLL Config tab

The **Verilog PLL Config** tab generates HDL code that can be copied directly into the design.



```
//
// PLL0
//

// Port List
output pll0_en,
output [5:0] pll0_refdiv,
output [11:0] pll0_fbddiv,
output [2:0] pll0_postdiv0_out0,
output [2:0] pll0_postdiv1_out0,
output [2:0] pll0_postdiv0_out1,
output [2:0] pll0_postdiv1_out1,
output [1:0] pll0_ref_clk_sel,
output pll0_bypass,

// Assigned signals
assign pll0_en = 1'b1;
assign pll0_refdiv = 6'b00_0001; // Equivalent value in decimal form 6'd1
assign pll0_fbddiv = 12'b0000_0001_0001; // Equivalent value in decimal form 12'd17
assign pll0_postdiv0_out0 = 3'b101; // Equivalent value in decimal form 3'd5
assign pll0_postdiv1_out0 = 3'b010; // Equivalent value in decimal form 3'd2
assign pll0_postdiv0_out1 = 3'b110; // Equivalent value in decimal form 3'd6
assign pll0_postdiv1_out1 = 3'b010; // Equivalent value in decimal form 3'd2
assign pll0_bypass = 1'b0;
assign pll0_ref_clk_sel = 2'b00; // OSC
```

Generated Verilog Code for PLL Configuration

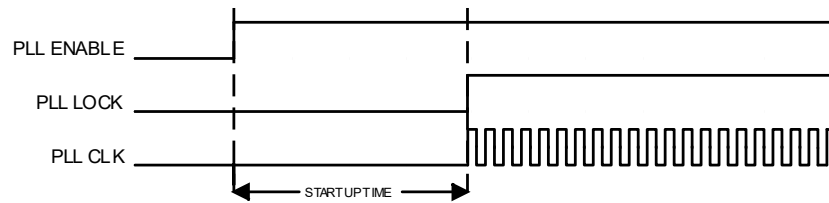
1. Copy the generated port list into the top-level module.
2. Copy the generated PLL configuration assignments into the design source code.

Conclusion

The integrated PLLs in the SLG47910, SLG47912, SLG47920, and SLG47921 provide an efficient way to generate custom clock frequencies inside the FPGA. The PLL Configurator tool in ForgeFPGA Workshop simplifies PLL setup by calculating valid divider values, verifying operating limits, and generating the HDL code required to apply the selected configuration.

2.4 PLL Startup Time Calculation

The PLL startup time is defined as the time required for the phase-locked loop to achieve stable frequency and phase lock after it is enabled or powered up.



PLL Startup Timing Diagram

PLL LOCK – Output signal which indicates that no cycle slips between the feedback clock and FPDF for 256 consecutive cycles.

Frequency of the signal at the Phase/frequency detector (PFD) input calculated by next formula:

$$F_{PFD} = \frac{F_{REF}}{REFDIV}$$

Where:

- F_{REF} – Reference clock input frequency.
- $REFDIV$ – Reference divide value (1 to 63).

Period of the signal at the PFD input calculated by next formula:

$$T_{PFD} = \frac{1}{F_{PFD}}$$

TLOCK TIME - Start UP time, between PLL EN (Enable) rising edge and PLL LOCK rising edge.

$$T_{LOCK TIME} = (600 \dots 900) T_{PFD}$$

2.5 Configuration from MCU

This example demonstrates how to configure a ForgeFPGA device from an external MCU using the SPI interface. The MCU loads a generated bitstream file (FPGA_bitstream_MCU.bin) into the FPGA configuration memory.

Supported Devices

- SLG47910
- SLG47912
- SLG47920
- SLG47921

Required Connections

FPGA signal	MCU signal
CS	GPIO
SPI_CLK	SPI SCK
SPI_MOSI	SPI MOSI
SPI_MISO	SPI MISO
nRST	GPIO
nSLEEP	GPIO

Configuration Flow

1. Assert nRST and nSLEEP.
2. Release the control pins to start the configuration sequence.
3. Select the FPGA using the CS signal.
4. Transfer the bitstream through SPI.
5. Deassert CS.
6. FPGA starts operating with the loaded configuration.

SLG47910 Configuration

The following example configures the SLG47910 device from an MCU.

Python Example

```
import machine
import utime
import binascii
# Settings
CS = machine.Pin(17, machine.Pin.OUT)
nSLEEP = machine.Pin(20, machine.Pin.OUT)
nRST = machine.Pin(21, machine.Pin.OUT)
# Initialize SPI
SPI = machine.SPI(
    0,
    baudrate=16000000,
    polarity=0,
    phase=0,
    bits=8,
    firstbit=machine.SPI.MSB,
    sck=machine.Pin(18),
    mosi=machine.Pin(19),
    miso=machine.Pin(16)
)

file_name = 'FPGA_bitstream_MCU.bin'
```

```
def send_bitstream_file(word_size=4):
    try:
        with open(file_name, 'rb') as f:
            while True:
                word = f.read(word_size)
                if not word:
                    break
                SPI.write(word)

    except OSError as e:
        print(f"Can't open the file: {e}")
    except Exception as e:
        print(f"Error: {e}")

# Main
if "__main__":
    print("Starting uploading bitstream:")
    utime.sleep(0.01)
    CS.value(0)
    nSLEEP.value(0)
    nRST.value(0)
    utime.sleep(0.1)
    nSLEEP.value(1)
    nRST.value(1)
    utime.sleep_ms(3)
    CS.value(1)
    utime.sleep_us(3)
    CS.value(0)
    send_bitstream_file()
    utime.sleep(0.1)
    CS.value(1)
```

SLG47912/SLG47920/SLG47921 Configuration

The following example configures SLG47912, SLG47920, and SLG47921 devices.

Python Example

```
import machine
import utime
import binascii

# Settings
CS = machine.Pin(17, machine.Pin.OUT)
nSLEEP = machine.Pin(20, machine.Pin.OUT)
nRST = machine.Pin(21, machine.Pin.OUT)
# Initialize SPI
SPI = machine.SPI(
    0,
    baudrate=16000000,
    polarity=0,
    phase=0,
    bits=8,
    firstbit=machine.SPI.MSB,
    sck=machine.Pin(18),
    mosi=machine.Pin(19),
    miso=machine.Pin(16)
)

file_name = 'FPGA_bitstream_MCU.bin'

def send_bitstream_file(word_size=4):
    try:
```

```
        with open(file_name, 'rb') as f:
            while True:
                word = f.read(word_size)
                if not word:
                    break
                SPI.write(word)
    except OSError as e:
        print(f"Can't open the file: {e}")
    except Exception as e:
        print(f"Error: {e}")

# Main
if "__main__":
    print("Starting uploading bitstream:")
    utime.sleep(0.01)
    CS.value(0)
    nSLEEP.value(0)
    nRST.value(0)
    utime.sleep(3)
    nSLEEP.value(1)
    nRST.value(1)
    utime.sleep(0.1)
    CS.value(0)
    utime.sleep(0.002)
    cmd = bytearray([0x02])
    SPI.write(cmd)
    send_bitstream_file()
    postamble = bytearray([0x00] * 1000)
    SPI.write(postamble)
    utime.sleep(0.1)
    CS.value(1)
    print("Finished uploading bitstream")
```

Bitstream File

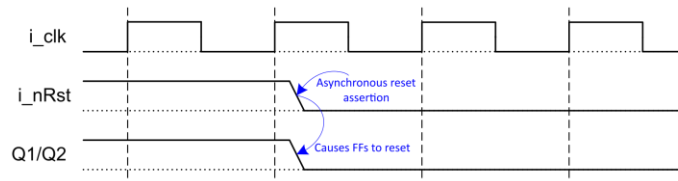
The configuration file must be generated using the ForgeFPGA toolchain and exported in MCU configuration format: `FPGA_bitstream_MCU.bin`.

Place the generated bitstream file in the same directory as the Python script before starting the configuration process.

3. RTL Coding Guide

3.1 Asynchronous Reset

An asynchronous reset allows a flip-flop or other sequential element to be reset independently of the clock signal. When the reset is asserted, the affected registers immediately transition to their reset state without waiting for a clock edge.



Asynchronous Reset Behavior

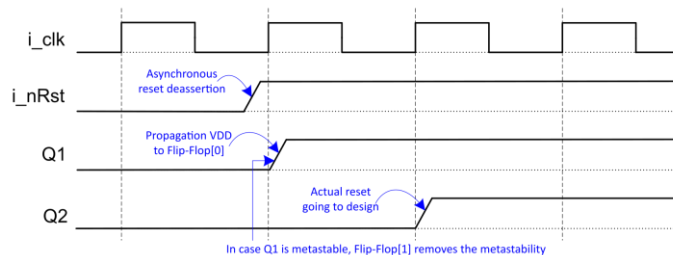
ForgeFPGA devices provide a global asynchronous chip reset connected to every flip-flop in the FPGA fabric. Therefore, it is generally unnecessary to include an additional global reset signal in user RTL.

An explicit reset should only be implemented if a register must be initialized to a value different from the device default state.

Reset Release Considerations

Although asynchronous assertion is safe, special care must be taken when de-asserting an asynchronous reset.

If the reset signal is released close to a clock edge, some flip-flops may enter a metastable state, potentially causing unpredictable behavior.



Reset Deassertion

To avoid metastability, asynchronous resets are common:

- Asserted asynchronously.
- Released synchronously to the destination clock domain.

A common implementation uses a two-stage reset synchronizer.

Reset Synchronizer Example

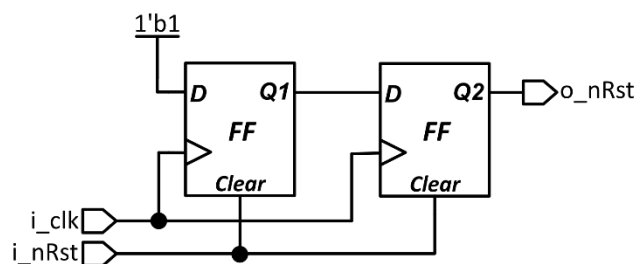
```
reg [1:0] FF;

always @(posedge i_clk or negedge
i_nRst) begin
    if (!i_nRst)
        FF <= 2'b00;
    else
        FF <= {FF[0], 1'b1};
End

assign o_nRst = FF[1];
```

In this implementation:

1. The asynchronous active-low reset (i_nRst) is applied directly to both flip-flops.
2. When reset is released, the logic value "1" propagates through the register chain.



3. The output reset (o_nRst) is released only after two clock cycles.
4. This guarantees a clean, synchronous reset release and minimizes metastability risks.

Good to Know

ForgeFPGA provides several useful status signals that can be used as reset sources depending on the application.

- OSC_READY – Can be used to hold logic in reset until the oscillator output becomes stable.
- PLL_LOCK – Recommended when logic operates in a PLL clock domain. The logic remains in reset until the PLL has achieved lock.
- FPGA_CORE_READY – Useful for one-time initialization after FPGA configuration has completed.
- FUNC_MODE – Can be used to reset internal logic when the device enters or exits sleep mode. This is useful for applications that require logic reinitialization after wake-up.

Best Practices

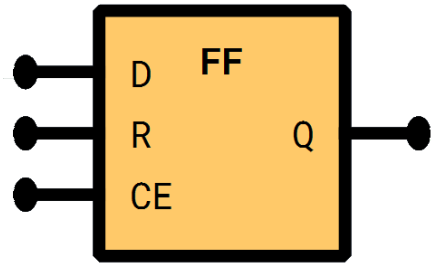
- Assert resets asynchronously.
- Release resets synchronously through a reset synchronizer.
- Use PLL_LOCK for PLL clock domains.
- Use OSC_READY for oscillator-based clock domains.
- Use FPGA_CORE_READY for power-up initialization.
- Avoid distributing asynchronous resets directly throughout large portions of the design.
- If possible, reset only the logic that requires initialization.

3.2 Flip-Flop Types and Control Signals

Flip-Flop with Clock Enable and Synchronous Reset (FDRE)

A flip-flop with clock enable and synchronous reset updates its state only on a clock edge. The synchronous reset has priority over the clock enable and forces the register into a known state in a deterministic manner, while the clock enable controls whether new data is loaded or the previous value is retained.

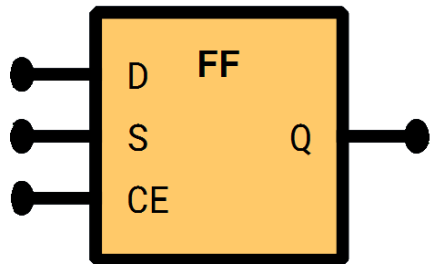
```
always @(posedge clk) begin
    if (!rst_n) begin
        q <= 1'b0;
    end else if (ce) begin
        q <= d;
    end
end
```



Flip-Flop with Clock Enable and Synchronous Set (FDSE)

A flip-flop with clock enable and synchronous set updates its state only on a clock edge. When the synchronous set is asserted, it forces the flip-flop output to a logic high value on the next active clock edge, overriding the clock enable. When the set is inactive, the clock enable controls whether new data is captured or the previous value is held, providing deterministic behavior without asynchronous timing hazards.

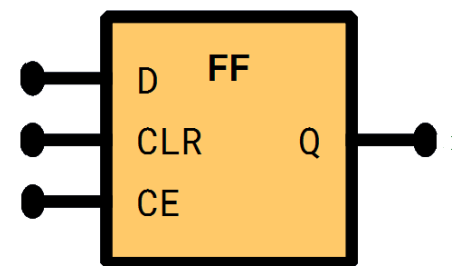
```
always @(posedge clk) begin
    if (set) begin
        q <= 1'b1;
    end else if (ce) begin
        q <= d;
    end
end
```



Flip-Flop with Clock Enable and Asynchronous Clear (FDCE)

A flip-flop with clock enable and asynchronous clear resets its output immediately when the clear signal is asserted, independently of the clock. When the asynchronous clear is inactive, the flip-flop updates its state only on the active clock edge, with the clock enable controlling whether new data is captured or the previous value is retained. While asynchronous clear allows fast and unconditional reset, its deassertion must be carefully handled to avoid metastability issues.

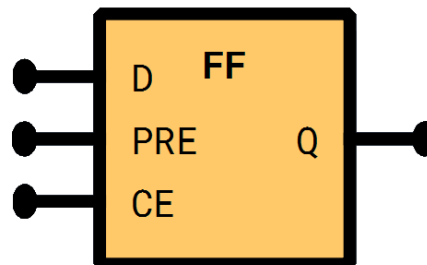
```
always @(posedge clk or negedge clr_n) begin
    if (!clr_n) begin
        q <= 1'b0;
    end else if (ce) begin
        q <= d;
    end
end
```



Flip-Flop with Clock Enable and Synchronous Set (FDPE)

A flip-flop with clock enable and synchronous set updates its state only on a clock edge. When the synchronous set is asserted, it forces the flip-flop output to logic high on the next active clock edge, overriding the clock enable. When the set is inactive, the clock enable controls whether new data is captured or the previous value is retained.

```
always @(posedge clk) begin
  if (set) begin
    q <= 1'b1;
  end else if (ce) begin
    q <= d;
  end
end
```



3.3 Verilog Coding Guidelines

This section summarizes recommended RTL coding practices for ForgeFPGA devices. Following these guidelines helps improve design robustness, timing closure, resource utilization, and compatibility with the ForgeFPGA implementation flow.

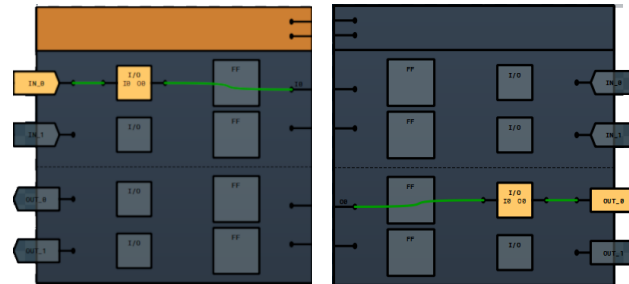
Register I/O Signals

Input and output signals should be registered whenever possible.

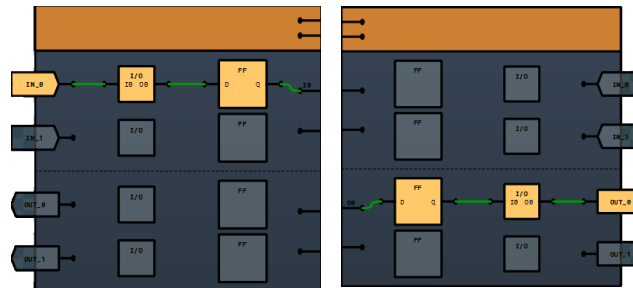
Registering I/O signals helps decouple FPGA fabric timing from external SoC timing and improves timing closure, especially when interfacing with hard IP blocks such as BRAM, LVDS, and other clock-sensitive resources.

The input and output registers should be placed in the FPGA I/O blocks. This recommendation does not apply to constant signals.

Avoid:



Recommended:



Synchronize Asynchronous Signals

All asynchronous signals entering the FPGA must be synchronized before being used by the design.

A two-stage flip-flop synchronizer is recommended to reduce metastability risk. During implementation, the Place-and-Route tool automatically places the synchronization flip-flops close together to improve reliability.

Recommended:

```
reg sync_ff1;
reg sync_ff2;

always @(posedge clk) begin
    sync_ff1 <= async_signal;
    sync_ff2 <= sync_ff1;
end
```

Flip-Flop Initialization

ForgeFPGA provides a global asynchronous chip reset connected to all flip-flops. Therefore, it is usually unnecessary to add an additional global reset signal in user RTL.

A declaration-based initialization:

```
reg r_ff = 1'b1;
```

is supported only by simulation and should not be relied upon for hardware initialization.

Recommended:

```
reg r_ff;

always @(posedge clk) begin
    if (!rst_n)
        r_ff <= 1'b1;
    else
        r_ff <= user_logic;
end
```

Use explicit reset logic whenever a register must power up to a known value other than the device default.

Avoid Latch Inference:

Latches are not supported in the ForgeFPGA core and may produce unpredictable implementation results.

In Verilog, latches are inferred when not all branches of a combinational process assign a value to the output.

Always ensure that all possible conditions are covered.

Example:

<pre>module IF_MUX (Sel, A,B,C, D, Z_out); input [1:0] Sel; input A, B, C, D; output Z_out; reg Z_out; always@ (A or B or C or D or Sel) begin if (Sel == 2'b00) Z_out = A; else if(Sel == 2'b01) Z_out = B; else if(Sel == 2'b10) Z_out = C; else Z_out = D; end endmodule</pre>	<pre>module CASE_MUX (Sel, A,B,C, D, Z_out); input [1:0] Sel; input A, B, C, D; output Z_out; reg Z_out; always@ (A or B or C or D or Sel) begin case (Sel) 2'b00: Z_out = A; 2'b01: Z_out = B; 2'b10: Z_out = C; 2'b11: Z_out = D; endcase end endmodule</pre>
---	---

Correct Latch-Free Combinational Logic Example

Unsupported Language Constructs

The ForgeFPGA core supports only binary logic values:

- 1'b0
- 1'b1

The following four-state logic values are not supported inside the FPGA fabric:

- 1'bx
- 1'bz

Avoid RTL constructs that depend on unknown or high-impedance values.

Avoid:

```
if (signal === 1'bx)
assign data = enable ? value : 1'bz;
```

Avoid Internal Tristates and `inout` Ports

Internal tristate buffers are not supported by the ForgeFPGA fabric.

Do not use:

- inout
- IOBUF
- BUFT
- internal tristate implementations

Avoid:

```
module top (
    inout  pad_io,
    input  pad_dout,
    input  pad_oe,
    output pad_din
);
    assign pad_io  = pad_oe ? pad_dout : 1'bz;
    assign pad_din = pad_io;
```

```
endmodule
```

Recommended:

Use the dedicated ForgeFPGA Output Enable (OE) mechanism as described in [2.1 GPIO Output Enable Usage](#).

This approach provides equivalent bidirectional functionality while remaining fully supported by the implementation flow.

Use Clock Enables Instead of Multiple Clocks

Avoid creating gated, divided, or derived clocks in RTL whenever possible.

Using multiple clock domains increases implementation complexity and often results in higher LUT and flip-flop utilization.

Instead of generating additional clocks, use a clock enable signal within the primary clock domain.

Recommended:

```
always @(posedge clk) begin
    if (clk_enable)
        counter <= counter + 1'b1;
end
```

Avoid:

```
always @(posedge divided_clk) begin
    counter <= counter + 1'b1;
end
```

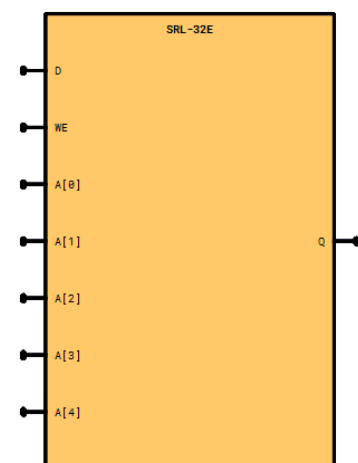
3.4 Shift Registers (SRL)

A shift register is a sequential circuit that stores and shifts data by one bit position on each clock cycle. Shift registers are widely used for data serialization, programmable delays, pattern generation, digital filters, and data synchronization.

ForgeFPGA devices support dedicated Shift Register LUT (SRL) implementations using Configurable Logic Blocks configured in Shift Register Mode (SRM). Compared to a traditional flip-flop chain, SRL implementations provide a more efficient utilization of FPGA resources.

Shift Register Resources in ForgeFPGA

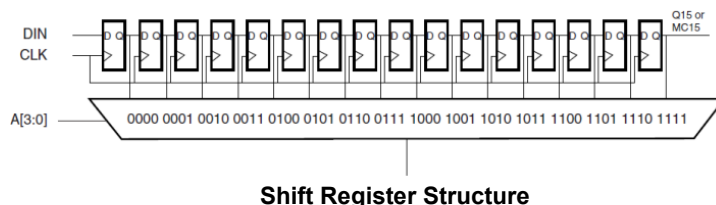
The memory-capable CLBs can be configured as shift registers.



The following configurations are supported:

Mode	Depth
SRL16E	16 bits
SRL32E	32 bits
SRL64E	64 bits

These resources are implemented using the FPGA distributed memory architecture and provide an efficient alternative to long chains of flip-flops.



Static Shift Register

A static shift register provides a fixed delay between the input and output.

Verilog Example

```

module shift_register_32 (
    input wire clk,
    input wire we,
    input wire din,
    output wire dout
);

    reg [31:0] shift_reg;

    always @(posedge clk) begin
        if (we) shift_reg <= {shift_reg[30:0], din};
    end

    assign dout = shift_reg[31];
endmodule

```

In this example, the input data is shifted through a 16-bit register chain. The output appears after a fixed delay of 16 clock cycles.

Dynamic Shift Register

A dynamic shift register allows the output delay to be selected during operation.

Verilog Example

```

module dynamic_shift_register (
    input wire clk,
    input wire we,
    input wire din,
    input wire [4:0] addr,
    output wire dout
);

    reg [31:0] shift_reg;

    always @(posedge clk) begin
        if (we) shift_reg <= {shift_reg[30:0], din};
    end

    assign dout = shift_reg[addr];
endmodule

```

endmodule

In this implementation, the output tap position is selected dynamically using the `addr` input, allowing programmable delay lengths without modifying the RTL.

Typical Applications

Shift registers are commonly used for:

- Data delay lines
- Serial-to-parallel conversion
- Parallel-to-serial conversion
- Digital filtering
- Pulse extension
- Data synchronization
- Pattern generation

Resource Optimization**Recommended:**

Use shift register coding styles that allow the synthesis tool to infer SRL resources.

```
reg [31:0] shift_reg;

always @(posedge clk) begin
    shift_reg <= {shift_reg[30:0], din};
end
```

Avoid:

Implementing long delays using individual flip-flops when SRL resources can be inferred.

```
reg ff0;
reg ff1;
reg ff2;
reg ff3;
...
```

Large flip-flop chains consume more LUT and register resources compared to an SRL implementation.

Good to Know

ForgeFPGA devices support dedicated Shift Register Mode (SRM) within memory-capable CLBs.

Depending on the selected configuration, a single memory CLB can be used as:

- Four independent 16-bit shift registers (SRL16E)
- Two independent 32-bit shift registers (SRL32E)
- One 64-bit shift register (SRL64E)

This functionality is implemented using the distributed memory resources inside the FPGA fabric.

3.5 Finite State Machine

A Finite State Machine (FSM) is a control structure that moves between a predefined set of states in response to inputs and clock events.

In FPGA designs, FSMs are commonly used to implement control and sequencing logic, such as protocol handling, operation control, data flow coordination, and peripheral management.

Moore FSM

A Moore FSM is a type of Finite State Machine where the outputs depend only on the current state, not directly on the inputs.

In FPGA designs, Moore FSMs are used for control logic that requires stable and predictable outputs, since outputs change only on state transitions synchronized to the clock.

```
module Moore_FSM (
    input          i_clk,
    input          i_data,
    input          i_nRst,
    output reg [1:0] o_data
);
    reg [3:0] state;

    parameter STATE_0 = 2'b00;
    parameter STATE_1 = 2'b01;
    parameter STATE_2 = 2'b10;
    parameter STATE_3 = 2'b11;

    always @(posedge i_clk or negedge i_nRst) begin
        if (~i_nRst) begin
            state <= STATE_0;
        end else begin
            case (state)
                STATE_0: begin
                    if (i_data) state <= STATE_0;
                    else       state <= STATE_1;
                end

                STATE_1: begin
                    if (i_data) state <= STATE_1;
                    else       state <= STATE_2;
                end

                STATE_2: begin
                    if (i_data) state <= STATE_2;
                    else       state <= STATE_3;
                end

                STATE_3: begin
                    if (i_data) state <= STATE_3;
                    else       state <= STATE_0;
                end

                default: begin
                    state <= STATE_0;
                end
            endcase
        end
    end

    always @ (*) begin
        case (state)
            STATE_0: o_data <= 2'b00;
            STATE_1: o_data <= 2'b01;
```

```
        STATE_2: o_data <= 2'b11;
        STATE_3: o_data <= 2'b11;
        default: o_data <= 2'b00;
    endcase
end
endmodule
```

Mealy FSM

A Mealy FSM is a type of Finite State Machine where the outputs depend on both the current state and the inputs.

In FPGA designs, Mealy FSMs are used when faster response is required, since outputs can change immediately with input changes, often allowing more compact and efficient control logic.

```
module Mealy_FSM (
    input          i_clk,
    input          i_data,
    input          i_nRst,
    output reg [1:0] o_data
);

    reg [3:0] state;

    parameter STATE_0 = 2'b00;
    parameter STATE_1 = 2'b01;
    parameter STATE_2 = 2'b10;
    parameter STATE_3 = 2'b11;

    always @(posedge i_clk or negedge i_nRst) begin
        if (~i_nRst) begin
            state <= STATE_0;
        end else begin
            case (state)
                STATE_0: begin
                    state <= i_data ? STATE_0 : STATE_1;
                end

                STATE_1: begin
                    state <= i_data ? STATE_1 : STATE_2;
                end

                STATE_2: begin
                    state <= i_data ? STATE_2 : STATE_3;
                end

                STATE_3: begin
                    state <= i_data ? STATE_3 : STATE_0;
                end

                default: begin
                    state <= STATE_0;
                end
            endcase
        end
    end

    always @ (*) begin
        case (state)
            STATE_0: o_data <= i_data ? 2'b00 : 2'b01;
            STATE_1: o_data <= i_data ? 2'b01 : 2'b11;
            STATE_2: o_data <= i_data ? 2'b10 : 2'b01;
            STATE_3: o_data <= i_data ? 2'b11 : 2'b10;
```

```
        default: o_data <= 2'b00;  
    endcase  
end  
endmodule
```

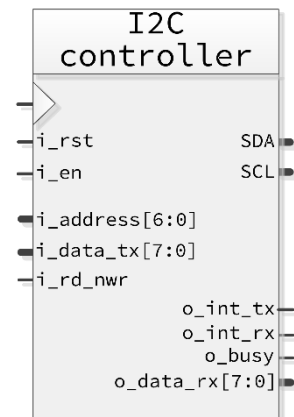
4. Module Library

4.1 Communication

4.1.1 Simple I²C Controller

I²C is used for low-speed communication between integrated circuits.

The I²C controller manages data exchange between a controller and one or more target devices using a two-wire interface, consisting of a serial data line (SDA) and a serial clock line (SCL).



I²C Controller Parameters

- **IN_CLK_HZ** – Specifies the input system clock frequency in Hz. This parameter is used to derive internal timing for I²C operations.
- **SCL_SPEED_MODE_HZ** – Defines the target output frequency of the I²C serial clock (SCL) in Hz. The parameter supports standard I²C speed modes, including Standard-mode (*100 kHz), Fast-mode (400 kHz), and Fast-mode Plus (1 MHz*).

Ports

Port name	Direction	Description
i_clk	input	System clock input
i_rst	input	Active reset input
i_en	input	Start signal that initiates an I ² C transaction
o_busy	output	Indicates that the controller is busy processing a transaction
i_scl	input	External serial clock (SCL) input
i_sda	input	External serial data (SDA) input
o_scl	output	Serial clock (SCL) output
o_scl_oe	output	Output enable for the SCL line (open-drain control)
o_sda	output	Serial data (SDA) output
o_sda_oe	output	Output enable for the SDA line (open-drain control)
i_address	input	7-bit I ² C slave address
i_rd_nwr	input	Operation select write (0) or read (1)
i_data_tx	input	Data byte to be transmitted over the I ² C bus
o_data_rx	output	Data byte received from the I ² C bus
o_int_tx	output	Asserted when the transmit operation is completed
o_int_rx	output	Asserted when new received data is available

Design steps

1. Add the **I²C controller** to the design by selecting it from the **Module Library**.
2. Instantiate the I²C controller in the top-level design file as shown in the following example.
3. Set the configuration parameters according to the application requirements.
4. Connect the controller control signals to the corresponding system logic.
5. Connect the SDA and SCL signals to the external I²C bus.

I²C Controller instantiation:

```
i2c_controller #(
    .IN_CLK_HZ          ( user_param ),
```

```

    .SCL_SPEED_MODE_HZ ( user_param )
) user_inst_name (
    .i_clk      ( user_name ),
    .i_rst      ( user_name ),
    .i_en       ( user_name ),
    .o_busy     ( user_name ),
    .i_scl      ( user_name ),
    .i_sda      ( user_name ),
    .o_scl      ( user_name ),
    .o_scl_oe   ( user_name ),
    .o_sda      ( user_name ),
    .o_sda_oe   ( user_name ),
    .i_address  ( user_name ),
    .i_rd_nwr   ( user_name ),
    .i_data_tx  ( user_name ),
    .o_data_rx  ( user_name ),
    .o_int_tx   ( user_name ),
    .o_int_rx   ( user_name )
);

```

Module operation

The simple I²C controller provides support for four basic operations: single-byte write, single-byte read, sequential write, and sequential read.

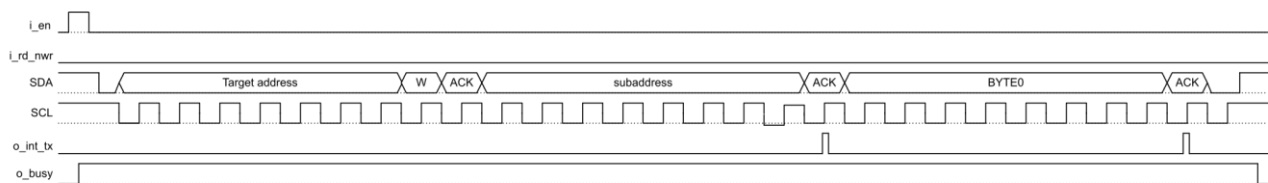
The module implements a basic I²C controller interface and is intended for straightforward communication scenarios.

Clock stretching is not supported by this controller; therefore, implement an external watchdog mechanism to handle stalled or non-responsive I²C transactions.

To perform a single-byte write operation, first provide the target I²C target address on the `i_address` bus and the register sub-address on the `i_data_tx` bus, `i_rd_nwr` should be low.

To start the write sequence, generate a single-clock-cycle pulse on the `i_en` signal. Once the operation is initiated, the controller asserts the `o_busy` signal, indicating that a transaction is in progress.

When the controller is ready to accept the data byte, the `o_int_tx` signal is asserted. This strobe indicates that you may place the data byte to be written on the `i_data_tx` bus. The controller then transmits the data to the specified register sub-address of the addressed I²C target device.



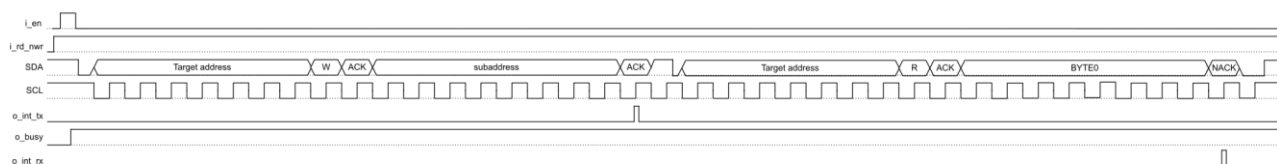
I²C Single write

For single-byte read operation, follow the same steps as for a single-byte write operation. The I²C target address and register sub-address must be provided in the same manner.

The only difference is that the `i_rd_nwr` signal must be set HIGH, indicating a read operation.

After a single-clock-cycle pulse is applied to the `i_en` signal, the controller automatically performs a single-byte write cycle to transmit the register sub-address, followed by a repeated START condition to initiate the read phase. The controller then reads one data byte from the addressed target device and updates the received data on the `o_data_rx` bus.

The completion of the read operation is indicated by the assertion of the `o_int_rx` signal, while the `o_busy` signal remains asserted for the duration of the entire transaction.

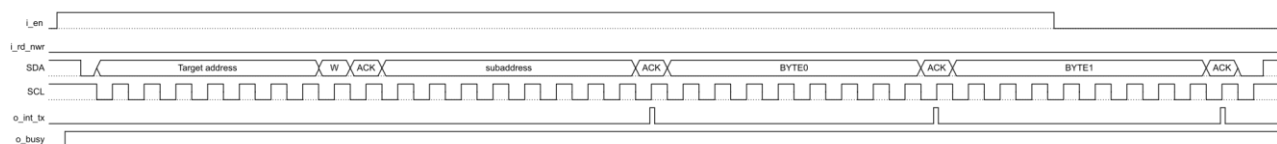


I²C Single Read

To enable sequential write mode, assert and keep the *i_en* signal HIGH for the duration of the transaction. While *i_en* remains asserted, the controller continues transmitting consecutive data bytes to the addressed slave device.

The *o_int_tx* signal is used to indicate that the controller is ready to accept the next data byte. This strobe may also be used to count the number of transmitted bytes during the sequential write operation.

To terminate the sequential write sequence, deassert the *i_en* signal by driving it LOW during the transmission of the current data byte. The controller then completes the ongoing transfer and generates the STOP condition, ending the I²C transaction.

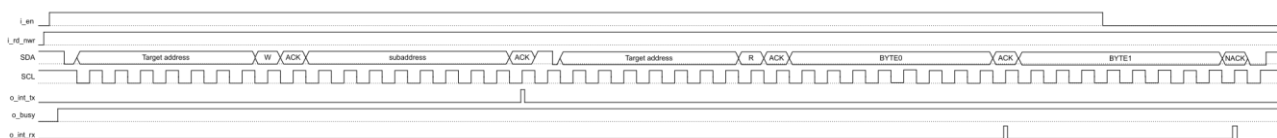


I²C Sequential Write

The sequential read operation is performed in the same manner as the sequential write operation, with the only difference being that the *i_rd_nwr* signal must be set HIGH to select read mode.

While the *i_en* signal remains asserted, the controller continuously reads consecutive data bytes from the addressed slave device. Each time a new data byte is received, the *o_int_rx* signal is asserted. This strobe indicates that valid data is available on the *o_data_rx* bus and may also be used to count the number of received bytes.

To terminate the sequential read operation, deassert the *i_en* signal by driving it LOW during the reception of the current data byte. The controller then completes the ongoing transfer and generates the STOP condition, ending the I²C transaction.

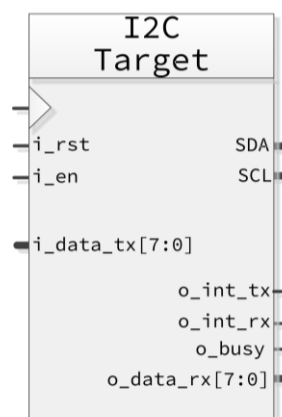


I²C Sequential Read

4.1.2 Simple I²C Target

I²C is used for low-speed communication between integrated circuits.

The I²C controller manages data exchange between a controller and one or more target devices using a two-wire interface, consisting of a serial data line (SDA) and a serial clock line (SCL).



I²C Target Parameters

- `I2C_TARGET_ADR` – Defines the address used to identify the I²C target on the bus.

Ports

Port name	Direction	Description
<code>i_clk</code>	input	System clock input
<code>i_rst</code>	input	Active reset input
<code>i_en</code>	input	Start signal that initiates an I ² C operation
<code>o_busy</code>	output	Indicates that the controller is busy processing a operation
<code>i_scl</code>	input	External serial clock (SCL) input
<code>i_sda</code>	input	External serial data (SDA) input
<code>o_scl</code>	output	Serial clock (SCL) output
<code>o_scl_oe</code>	output	Output enable for the SCL line (open-drain control)
<code>o_sda</code>	output	Serial data (SDA) output
<code>o_sda_oe</code>	output	Output enable for the SDA line (open-drain control)
<code>i_data_tx</code>	input	Data byte to be transmitted over the I ² C bus
<code>o_data_rx</code>	output	Data byte received from the I ² C bus
<code>o_int_tx</code>	output	Asserted when the transmit operation is completed
<code>o_int_rx</code>	output	Asserted when new received data is available

Design Steps

1. Add the **I²C target** to the design by selecting it from the **Module Library**.
2. Instantiate the I²C target in the top-level design file as shown in the following example.
3. Set the configuration parameters according to the application requirements.
4. Connect the controller control signals to the corresponding system logic.
5. Connect the SDA and SCL signals to the external I²C bus.

I²C Controller Instantiation

```
i2c_target #(
  .I2C_TARGET_ADR ( user_param )
) user_inst_name (
  .i_clk      ( user_name ),
  .i_rst      ( user_name ),
  .i_en       ( user_name ),
  .o_busy     ( user_name ),
  .i_scl      ( user_name ),
  .i_sda      ( user_name ),
  .o_sda      ( user_name ),
  .o_sda_oe   ( user_name ),
  .i_data_tx   ( user_name ),
  .o_data_rx   ( user_name ),
  .o_int_tx   ( user_name ),
  .o_int_rx   ( user_name )
);
```

Module Operation

The I²C Target module is enabled by an internal control signal, which must be asserted to initiate operation. This enable signal is required only to start the module at the occurrence of a START condition. If the enable signal is disserted during an ongoing transaction, the module completes the current read or write operation before stopping.

The target device address is configured through a dedicated parameter in the I²C Target module prior to design implementation.

After a START condition, the target device expects an 8-bit address word. The upper seven bits represent the device address, while the least significant bit determines the transfer direction. A logic high initiates a read operation, whereas a logic low initiates a write operation. If the address transmitted by the controller does not match the configured target address, the module transitions to an idle state.

During a write transaction, data received from the controller is stored in the o_data_rx buffer, and the o_int_rx signal is asserted for one internal clock period to indicate valid data reception. During a read transaction, the transmit data on the i_data_tx input must be valid before the START condition occurs. The o_int_tx pulse indicates completion of the data transfer, after which the transmit data may be updated.

The o_busy signal remains asserted from the detection of the START condition until the STOP condition is issued.

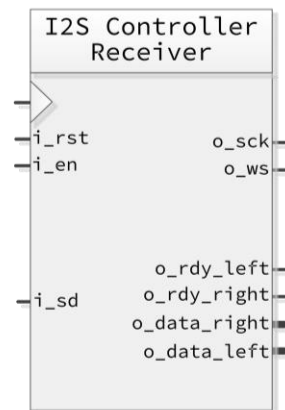


I²C Target Operation

4.1.3 I²S Controller (Receiver)

I²S is used for transferring high-quality audio data between electronic components.

The I²S receiver in the controller mode manages the incoming transfer of digital audio data from an external audio source. It operates over a three-wire serial interface consisting of the bit clock (BCLK), the word-select line (WS), and the serial data line (SD). It supports various audio sample formats, including 16-bit, 24-bit, and 32-bit.



I²S Controller (Receiver) Parameters

- **IN_CLK_HZ** – Specifies the input system clock frequency in Hz. This parameter is used to derive internal timing for I²S operations.
- **SAMPLING_RATE_HZ** – Defines the target sampling rate frequency in Hz.
- **DATA_WIDTH** – Selects the audio sample format by its bit width. Determines the port width of the output buses.

Ports

Port name	Direction	Description
i_clk	input	System clock input
i_rst	input	Active high reset input
i_en	input	Signal that enables an I ² S operation
i_sd	input	External serial data (SD) input
o_sck	output	Bit clock (BCLK) output
o_ws	output	Word select (WS) output
o_data_left	output	Left channel data output bus
o_data_right	output	Right channel data output bus
o_rdy_left	output	Interrupt signal for the left channel
o_rdy_right	output	Interrupt signal for the right channel

Design steps

1. Add the **I²S controller (receiver)** to the design by selecting it from the **Module Library**.
2. Instantiate the I²S controller (receiver) in the top-level design file as shown in the following example.
3. Set the configuration parameters according to the application requirements.
4. Connect the controller control signals to the corresponding system logic.
5. Connect the BCLK, SD, and WS signals to the external I²S bus.

I²S Controller (Receiver) instantiation:

```

i2s_controller_receiver #(
    .IN_CLK_HZ          ( user_param ),
    .SAMPLING_RATE_HZ   ( user_param ),
    .DATA_WIDTH         ( user_param )
) user_inst_name (
    .i_clk              ( user_name ),
    .i_rst              ( user_name ),
    .i_en               ( user_name ),
    .i_sd               ( user_name ),
    .o_sck              ( user_name ),

```

```

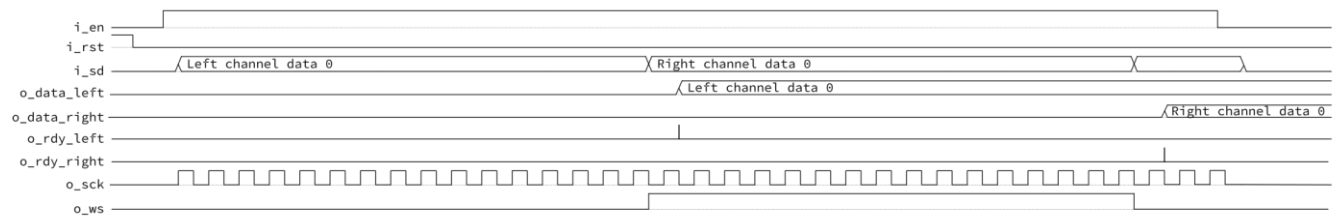
.o_ws      ( user_name ),
.o_data_left ( user_name ),
.o_data_right ( user_name ),
.o_rdy_left  ( user_name ),
.o_rdy_right ( user_name )
);

```

Module operation

The simple I²S controller receiver automatically generates the bit clock signal and word select signal. It receives data with length defined in DATA_WIDTH parameter as long as the i_en signal is held high.

To start the communication, you should assert the i_en signal. The operation continues as long as i_en remains active. The module generates the o_sck and the o_ws signals internally. After the number of clock cycles defined by the DATA_WIDTH parameter, o_ws toggles its polarity, indicating a change of the input channel. Note that, according to the protocol specification, there is an additional one-BCLK-period delay between the channel change and the WS-line toggle. The o_rdy_left port outputs a high signal for one input clock period to indicate that the data received through the left channel is available on the o_data_left bus. The data remains latched until the next ready signal. The o_ready_right port behaves the same for the o_data_right bus.

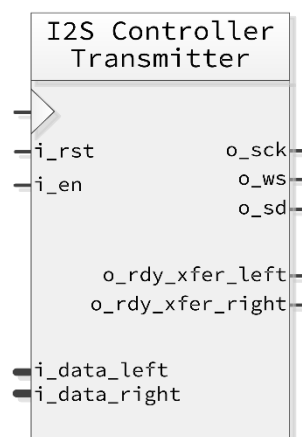


I²S controller receiver operation

4.1.4 I²S Controller (Transmitter)

I²S is used for transferring high-quality audio data between electronic components.

The I²S transmitter in the controller mode manages the transfer of digital audio data to an external audio source. It operates over a three-wire serial interface consisting of the bit clock (BCLK), the word-select line (WS), and the serial data line (SD). It supports various audio sample formats, including 16-bit, 24-bit, and 32-bit.



I²S Controller (Transmitter) Parameters

- **IN_CLK_HZ** – Specifies the input system clock frequency in Hz. This parameter is used to derive internal timing for I²S operations.
- **SAMPLING_RATE_HZ** – Defines the target sampling rate frequency in Hz.
- **DATA_WIDTH** – Selects the audio sample format by its bit width. Determines the port width of the input buses.

Ports

Port name	Direction	Description
i_clk	input	System clock input
i_rst	input	Active high reset input
i_en	input	Signal that enables an I ² S operation
o_sd	output	External serial data (SD) output
o_sck	output	Bit clock (BCLK) output
o_ws	output	Word select (WS) output
i_data_left	input	Left channel data input bus
i_data_right	input	Right channel data input bus
o_rdy_xfer_left	output	Interrupt signal for the left channel
o_rdy_xfer_right	output	Interrupt signal for the right channel

Design steps

1. Add the **I²S controller (transmitter)** to the design by selecting it from the **Module Library**.
2. Instantiate the I²S controller (transmitter) in the top-level design file as shown in the following example.
3. Set the configuration parameters according to the application requirements.
4. Connect the controller control signals to the corresponding system logic.
5. Connect the BCLK, SD, and WS signals to the external I²S bus.

I²S Controller (Transmitter) instantiation:

```
i2s_controller_transmitter #(
    .IN_CLK_HZ      ( user_param ),
    .SAMPLING_RATE_HZ ( user_param ),
    .DATA_WIDTH     ( user_param )
) user_inst_name (
    .i_clk          ( user_name ),
    .i_rst          ( user_name ),
    .i_en           ( user_name ),
    .o_sd           ( user_name ),
```

```

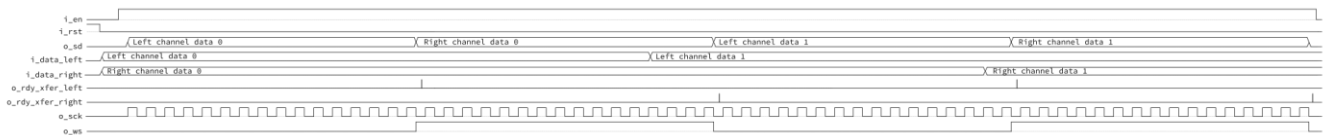
.o_sck          ( user_name ),
.o_ws          ( user_name ),
.i_data_left    ( user_name ),
.i_data_right   ( user_name ),
.o_rdy_xfer_left ( user_name ),
.o_rdy_xfer_right ( user_name )
);

```

Module operation

The simple I²S controller (transmitter) generates bit clock, word select, and serial data signals internally. It transmits data with length defined in DATA_WIDTH parameter as long as the i_en signal is held high.

Before the start of the transmission, you should set up the i_data_left and the i_data_right ports. To start the communication, you should assert the i_en signal. The operation continues as long as i_en remains active. The module generates the o_sck, the o_ws, and the o_sd signals internally. After the number of clock cycles defined by the DATA_WIDTH parameter, o_ws toggles its polarity, indicating a change of the input channel. Note that, according to the protocol specification, there is an additional one-BCLK-period delay between the channel change and the WS-line toggle. The o_rdy_xfer_left port outputs a high signal for one input clock period to indicate that the i_data_left data has been transmitted. The o_rdy_xfer_right port behaves the same for the i_data_right bus. To establish multi-word communication, you should set new data on the i_data_left and the i_data_right ports before the beginning of transmission on the corresponding channel.

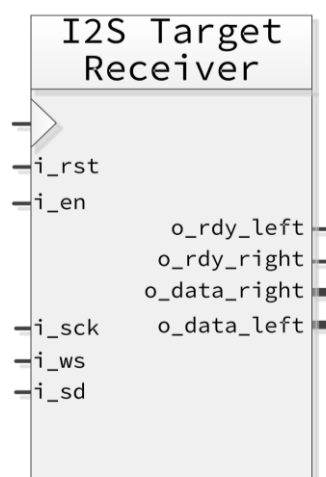


I²S controller (transmitter) operation

4.1.5 I²S Target (Receiver)

I²S is used for transferring high-quality audio data between electronic components.

The I²S receiver in the target mode handles the incoming transfer of digital audio data from an external audio source. As a target device, it relies on externally provided timing signals. It operates over a three-wire serial interface consisting of the bit clock (BCLK), the word-select line (WS), and the serial data line (SD). It supports various audio sample formats, including 16-bit, 24-bit, and 32-bit.



I²S Target (Receiver) Parameters

- **DATA_WIDTH** – Selects the audio sample format by its bit width. Determines the port width of the output buses.

Ports

Port name	Direction	Description
i_clk	input	System clock input
i_rst	input	Active high reset input
i_en	input	Signal that enables an I ² S operation
i_sd	input	External serial data (SD) input
i_sck	output	Bit clock (BCLK) input
i_ws	output	Word select (WS) input
o_data_left	output	Left channel data output bus
o_data_right	output	Right channel data output bus
o_rdy_left	output	Interrupt signal for the left channel
o_rdy_right	output	Interrupt signal for the right channel

Design steps

1. Add the **I²S target (receiver)** to the design by selecting it from the **Module Library**.
2. Instantiate the I²S target (receiver) in the top-level design file as shown in the following example.
3. Set the configuration parameters according to the application requirements.
4. Connect the target control signals to the corresponding system logic.
5. Connect the BCLK, SD, and WS signals to the external I²S bus.

I²S Target (Receiver) instantiation:

```
i2s_controller_receiver #(
    .IN_CLK_HZ      ( user_param ),
    .SAMPLING_RATE_HZ ( user_param ),
    .DATA_WIDTH     ( user_param )
) user_inst_name (
    .i_clk      ( user_name ),
    .i_rst      ( user_name ),
    .i_en       ( user_name ),
    .i_sd       ( user_name ),
    .i_sck      ( user_name ),
    .i_ws       ( user_name ),
```

```

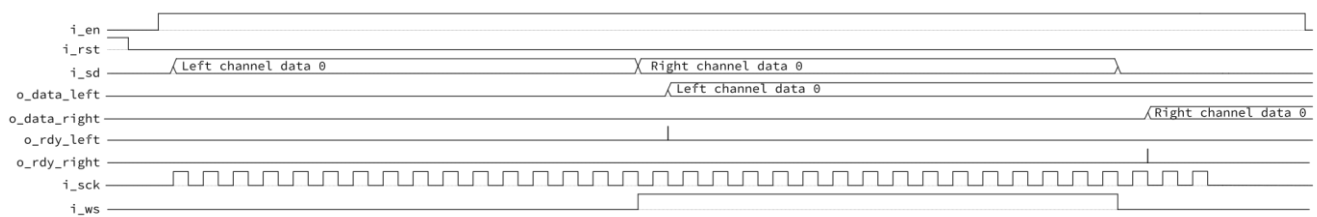
.o_data_left  ( user_name ),
.o_data_right ( user_name ),
.o_rdy_left   ( user_name ),
.o_rdy_right  ( user_name )
);

```

Module operation

The simple I²S target receiver does not generate any of the protocol signals. It receives data with length defined in DATA_WIDTH parameter as long as the i_en signal is asserted.

To enable communication, the i_en signal should be asserted. The BCLK, SD, and WS signals are generated by the controller device, receiver operates accordingly to them. The o_rdy_left port outputs a high signal for one input clock period to indicate that data received through the left channel is available on the o_data_left bus. The data remains latched until the next ready signal. The o_ready_right port behaves the same for the o_data_right bus. According to the I²S specification, there is an additional one-BCLK-period delay when switching between data channels.

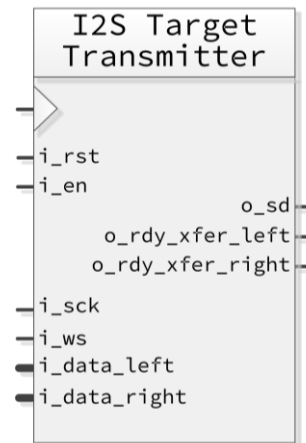


I²S target receiver operation

4.1.6 I²S Target (Transmitter)

I²S is used for transferring high-quality audio data between electronic components.

The I²S transmitter in the target mode handles the transfer of digital audio data to an external audio source. As a target device, it relies on externally provided timing signals. It operates over a three-wire serial interface consisting of the bit clock (BCLK), the word-select line (WS), and the serial data line (SD). It supports various audio sample formats, including 16-bit, 24-bit, and 32-bit.



I²S Target (Transmitter) Parameters

- **DATA_WIDTH** – Selects the audio sample format by its bit width. Determines the port width of the input buses.

Ports

Port name	Direction	Description
i_clk	input	System clock input
i_rst	input	Active high reset input
i_en	input	Signal that enables an I ² S operation
o_sd	output	External serial data (SD) output
i_sck	output	Bit clock (BCLK) input
i_ws	output	Word select (WS) input
i_data_left	input	Left channel data input bus
i_data_right	input	Right channel data input bus
o_rdy_xfer_left	output	Interrupt signal for the left channel
o_rdy_xfer_right	output	Interrupt signal for the right channel

Design steps

1. Add the **I²S target (transmitter)** to the design by selecting it from the **Module Library**.
2. Instantiate the I²S target (transmitter) in the top-level design file as shown in the following example.
3. Set the configuration parameters according to the application requirements.
4. Connect the target control signals to the corresponding system logic.
5. Connect the BCLK, SD, and WS signals to the external I²S bus.

I²S Target (Transmitter) instantiation:

```
i2s_controller_transmitter #(
    .IN_CLK_HZ      ( user_param ),
    .SAMPLING_RATE_HZ ( user_param ),
    .DATA_WIDTH     ( user_param )
) user_inst_name (
    .i_clk          ( user_name ),
    .i_rst          ( user_name ),
    .i_en           ( user_name ),
    .i_sd           ( user_name ),
    .i_sck          ( user_name ),
    .i_ws           ( user_name ),
    .i_data_left    ( user_name ),
    .i_data_right   ( user_name ),
```

```

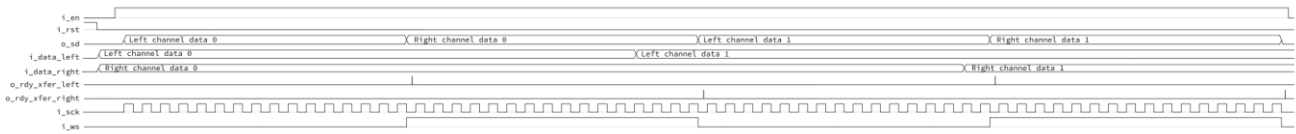
.o_rdy_xfer_left   ( user_name ),
.o_rdy_xfer_right  ( user_name )
);

```

Module operation

The simple I²S target transmitter generates SD signal based on BCLK and WS from external controller. It transmits data with length defined in DATA_WIDTH parameter as long as the i_en signal is asserted.

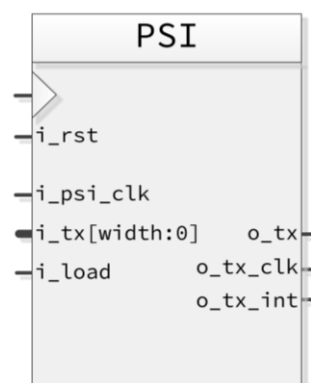
To enable transmission, the i_en signal should be asserted. The BCLK and WS signals are generated by the controller device, the transmitter operates accordingly to them and generates SD signal. The o_rdy_xfer_left port outputs a high signal for one input clock period to indicate that data has been transmitted through the left channel. The o_rdy_xfer_right port behaves the same for the right channel. According to the I²S specification, there is an additional one-BCLK-period delay when switching between data channels. This delay would be enough to change the transmitted data on the i_data_left and the i_data_right ports. The transmitted data should be set before i_en is asserted and before transmission is ongoing on the corresponding channel.



I²S target controller operation

4.1.7 Parallel to Serial Interface

PSI (Parallel to Serial) Interface is a high-speed transfer communication interface where multiple bits of data are transmitted simultaneously over multiple data lines and is received sequentially over a single wire triggered by a clock.



PSI Parameters

- **WIDTH** – The width of the input data bus (Type: Decimal; Default Value: 8, Min Value: 4, Max Value: 64).
- **LSB** – When set to 1, data starts from LSB, otherwise data starts from MSB (Type: Boolean; Default Value: 1'b0, Min Value: 1'b0, Max Value: 1'b1*).

Ports

Port name	Direction	Description
i_clk	input	System clock input
i_rst	input	Active reset input
i_tx	input	Input data bus signal
i_psi_clk	input	Input psi clock signal
i_load	input	Input load signal
o_tx	output	Output serial data signal
o_tx_oe	output	Output enable for output serial data signal
o_tx_clk	output	Output data clock signal
o_tx_clk_oe	output	Output enable for output data clock
o_tx_int	output	Output data interrupt signal
o_tx_int_oe	output	Output enable for data interrupt signal

Design Steps

1. Add the **PSI** to the design by selecting it from the **Module Library**.
2. Instantiate the PSI in the top-level design file as shown in the following example.
3. Set the configuration parameters according to the application requirements.
4. Connect the control signals to the corresponding system logic.

PSI Instantiation:

```
psi #(
    .WIDTH          ( user_param ),
    .LSB            ( user_param )
) user_inst_name (
    .i_clk          ( user_name ),
    .i_rst          ( user_name ),
    .i_load         ( user_name ),
    .i_tx           ( user_name ),
    .i_psi_clk      ( user_name ),
    .o_tx           ( user_name ),
    .o_tx_oe        ( user_name ),
    .o_tx_clk       ( user_name ),
    .o_tx_clk_oe    ( user_name ),
```

```
.o_tx_int      ( user_name ),  
.o_tx_int_oe   ( user_name ),  
);
```

Module Operation

This module is a configurable parallel-to-serial (PSI) transmitter that takes a multi-bit input word and sends out one bit at a time. It has two configurable parameters: WIDTH, which sets how many bits the input word contains, and LSB, which chooses whether the bits are transmitted starting from the least-significant bit or the most-significant bit. The module operates in the i_clk clock domain, meaning all its internal registers update on the rising edge of i_clk. It also watches two external control signals, i_load and i_psi_clk, by storing their previous values and comparing them to their current values so it can detect rising edges.

When a rising edge on i_load is detected, the module treats that as a command to begin a new transmission. It copies the parallel input bus i_tx into an internal register called r_tx_data, clears the bit counter, clears the interrupt output o_tx_int, resets the serial outputs o_tx and o_tx_clk, and enables transmission. At this point, the module is armed and waiting for timing events from i_psi_clk to shift data out.

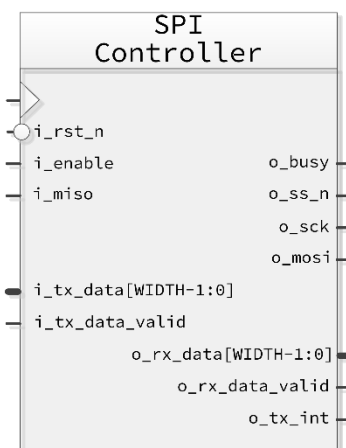
During transmission, each time a rising edge of i_psi_clk is detected, the module outputs one bit from r_tx_data onto o_tx. If LSB is 0, it sends the most-significant bit first; if LSB is 1, it sends the least-significant bit first. After outputting that bit, the module rotates the internal data register so that the next bit moves into position for the following transfer. It then increments r_clk_counter to keep track of how many bits have been sent. At the same time, it enables o_tx_clk, which is driven as the inverse of i_psi_clk while transmission is active, so an external device can use it as a serial clock reference.

Once the number of transmitted bits reaches the configured WIDTH, the module finishes the transfer. It stops driving serial data, disables the transmit clock output, clears the transmission enable flags, and raises o_tx_int to indicate that the full word has been transmitted. In simple terms, this code loads a parallel data word when told to, then shifts that word out one bit per i_psi_clk pulse, generates a matching output clock during the transfer, and finally asserts a done signal when all bits have been sent.

From the timing diagram shared, we can observe that after a rising edge is detected on the i_load signal, i_psi_clk gets ready to shift the data. Once a rising edge is detected on i_psi_clk, the module outputs one bit from i_tx (internal register r_tx_data) onto o_tx. Here the input data is 8'hA9 and the output data on a single wire can be seen as 8'b1010_1001 representing A9 in binary. After the complete data has been transmitted, the i_tx_clk disables and it raises o_tx_int to indicate that the full word has been transmitted.

4.1.8 SPI Controller

The Serial Peripheral Interface (SPI) is a synchronous, four-wire, full-duplex serial communication protocol used for short-distance, high-speed data transfer. It employs a controller-target (master-slave) architecture with a shared clock (SCK), separate data lines for transmission and reception (MOSI and MISO), and a dedicated chip-select (CS) signal for each target device. The SPI controller (master) initiates and controls all communication, including clock generation and data timing.



SPI Controller Parameters

- CPOL – Clock Polarity determines the active polarity of the clock signal from the sck output. When set to 1, clock is low in idle, otherwise clock is high.
- CPHA – Clock Phase determines the edge of data sample from the sck output. When one, sampling occurs at falling edge, otherwise at rising edge of non-inverted clock.
- WIDTH – Determines the data width of SPI to the input and output data buses (Min Value: 8, Max Value: 64).
- LSB – Determines the transfer format of output data and data order between least significant and most significant bit. When one, data starts from LSB, otherwise data starts from MSB.
- CLKS_PER_HBIT – Determines the duration of half the SPI clock period, derived from the input domain clock (Min Value: 3, Max Value: 64).
- WAIT_SS_HIGH – Determines the minimum duration for which o_ss_n remains HIGH between messages, expressed in half SPI clock periods (Min Value: 1, Max Value: 64).

Ports

Port name	Direction	Description
i_clk	input	System clock input signal
i_rst_n	input	Active-low reset input signal
i_enable	input	SPI Controller enable signal
o_busy	output	Indicates the SPI Controller is busy during a transfer
i_miso	input	Controller In Target Out input signal from the target device
o_ss_n	input	Active-low target select signal
o_sck	output	SPI serial clock output signal
o_mosi	output	Controller Out Target In output signal to the target device
i_tx_data	output	Transmit data bus input
i_tx_data_valid	output	Transmit data valid input signal
o_tx_int	input	Transmit data load interrupt output signal
o_rx_data	input	Receive data bus output
o_rx_data_valid	input	Receive data valid output signal

Design steps

1. Add the **SPI Controller** to the design by selecting it from the **Module Library**.
2. Instantiate the **SPI Controller** in the top-level design file as shown in the following example.
3. Set the configuration parameters according to the application requirements.
4. Connect the **SPI Controller** control signals to the corresponding system logic.

5. Connect the o_ss_n, the o_sck, the o_mosi, and the i_miso signals to the corresponding GPIOs using the I/O Planner.

SPI Controller instantiation:

```
spi_controller #(
    .CPOL          ( user_param ),
    .CPHA          ( user_param ),
    .WIDTH         ( user_param ),
    .LSB           ( user_param ),
    .CLKS_PER_HBIT ( user_param ),
    .WAIT_SS_HIGH  ( user_param )
) user_inst_name (
    .i_clk          ( user_name ),
    .i_rst_n        ( user_name ),
    .i_enable        ( user_name ),
    .o_busy          ( user_name ),
    .i_miso          ( user_name ),
    .o_ss_n          ( user_name ),
    .o_sck           ( user_name ),
    .o_mosi          ( user_name ),
    .i_tx_data       ( user_name ),
    .i_tx_data_valid ( user_name ),
    .o_tx_int        ( user_name ),
    .o_rx_data       ( user_name ),
    .o_rx_data_valid ( user_name )
);
```

Module operation

The SPI Controller Module supports four operating modes (Modes 0–3), which define the timing relationship between the clock (o_sck) and data (o_mosi/i_miso) using two parameters, clock polarity (CPOL) and clock phase (CPHA).

SPI Controller Modes Table

Mode	CPOL (Idle)	CPHA (Phase)	Description
0	0 (LOW)	0 (Leading)	Data sampled on rising edge, shifted on falling
1	0 (LOW)	1 (Trailing)	Data sampled on falling edge, shifted on rising
2	1 (HIGH)	0 (Leading)	Data sampled on falling edge, shifted on rising
3	1 (HIGH)	1 (Trailing)	Data sampled on rising edge, shifted on falling

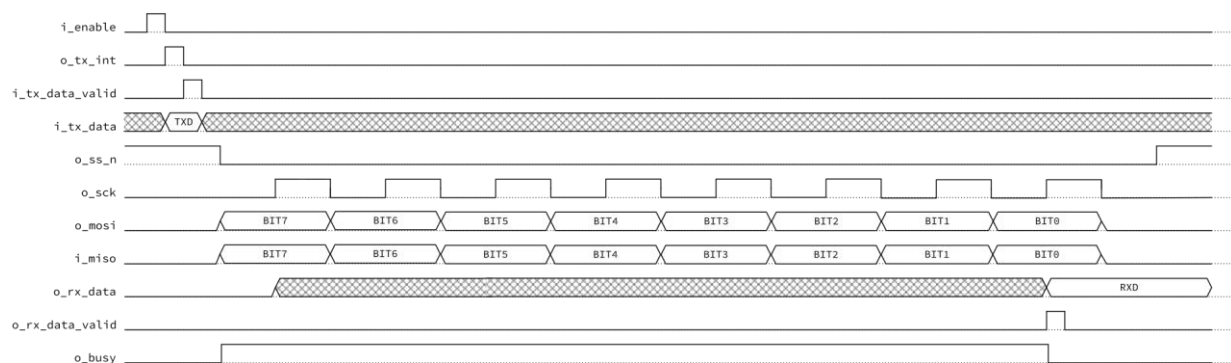
The SPI Controller supports single-frame transfers, with frame length defined by the WIDTH parameter, as well as sequential transmission and reception of multiple frames.

For single-frame operation, i_tx_data_valid must be asserted for at least one system clock cycle and deasserted before o_rx_data_valid is asserted. For sequential operation, i_tx_data_valid initiates the SPI transaction and must be deasserted before o_rx_data_valid is asserted on the final frame to terminate the transfer.

When i_enable is asserted prior to a SPI transaction, the controller generates an o_tx_int strobe to request transmit data and waits for i_tx_data_valid to begin transmission. After each frame, the controller asserts o_tx_int and o_rx_data_valid. If i_enable is deasserted before the o_rx_data_valid strobe, o_tx_int is not generated.

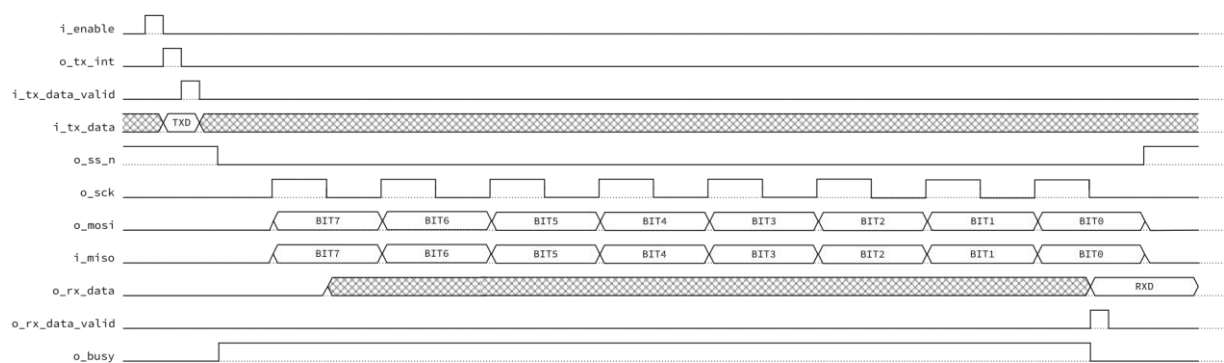
The o_busy signal remains asserted for the duration of an active SPI transaction.

CPHA = 0, CPOL = 0 (1 - when o_sck is inverted)



SPI Controller Mode 0 operation

CPHA = 0, CPOL = 0 (1 - when o_sck is inverted)

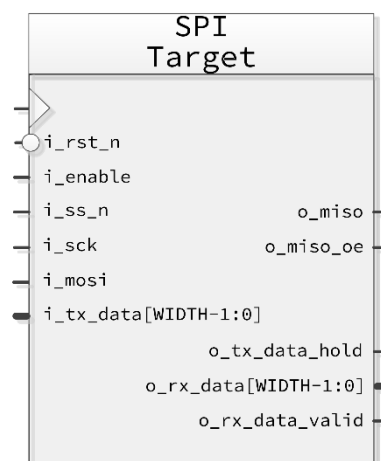


SPI Controller Mode 1 operation

4.1.9 SPI Target

The Serial Peripheral Interface (SPI) is a synchronous, four-wire, full-duplex serial communication protocol used for short-distance, high-speed data transfer. It employs a controller-target (master-slave) architecture with a shared clock (SCK), separate data lines for transmission and reception (MOSI and MISO), and a dedicated chip-select (CS) signal for each target device.

SPI Target operates as a passive SPI device that responds to controller-initiated transactions. Samples input data and shifts out response data synchronously with the SPI clock when selected by CS.



SPI Target Parameters

- CPOL – Clock Polarity determines the active polarity of the clock signal from the sck output. When one, clock is low in idle, otherwise clock is high.
- CPHA – Clock Phase determines the edge of data sample at the sck input. When one, sampling occurs at falling edge, otherwise at rising edge of non-inverted clock.
- WIDTH – Determines the data width of SPI to the input and output data buses (Min value: 8, Max value: 64).
- LSB – Determines the transfer format of output data and data order between least significant and most significant bit. When one, data starts from LSB, otherwise data starts from MSB.

Ports

Port name	Direction	Description
i_clk	input	System clock input signal
i_rst_n	input	Active-low reset input signal
i_enable	input	SPI Target enable signal
i_ss_n	input	Active-low target select signal
i_sck	input	SPI serial clock input signal
i_mosi	input	Controller Out Target In input signal from the Controller
o_miso	output	Controller In Target Out output signal to the Controller
o_miso_oe	output	o_miso output enable signal
o_rx_data	output	Receive data bus output
o_rx_data_valid	output	Receive data valid output signal
i_tx_data	input	Transmit data bus input
o_tx_data_hold	output	Transmit data hold input

Design steps

1. Add the **SPI Target** to the design by selecting it from the **Module Library**.
2. Instantiate the SPI Target in the top-level design file as shown in the following example.
3. Set the configuration parameters according to the application requirements.
4. Connect the SPI Target control signals to the corresponding system logic.
5. Connect the i_ss_n, the i_sck, the i_mosi, the o_miso, and the o_miso_oe signals to the corresponding GPIOs using the I/O Planner.

SPI Target instantiation:

```
spi_target #(
    .CPOL    ( user_param ),
    .CPHA    ( user_param ),
```

```

.WIDTH ( user_param ),
.LSB   ( user_param )
) user_inst_name (
.i_clk      ( user_name ),
.i_rst_n    ( user_name ),
.i_enable   ( user_name ),
.i_ss_n     ( user_name ),
.i_sck      ( user_name ),
.i_mosi     ( user_name ),
.o_miso     ( user_name ),
.o_miso_oe  ( user_name ),
.o_rx_data  ( user_name ),
.o_rx_data_valid ( user_name ),
.i_tx_data  ( user_name ),
.o_tx_data_hold ( user_name )
);

```

Module operation

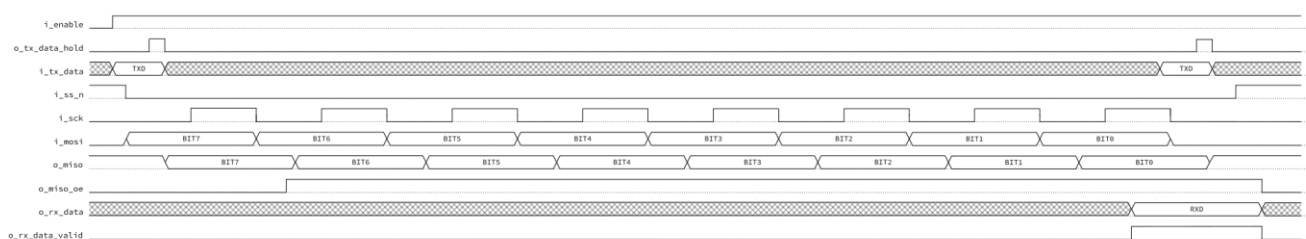
The SPI Target Module supports four operating modes (Modes 0–3), which define the timing relationship between the clock (o_sck) and data (o_mosi/i_miso) using two parameters, clock polarity (CPOL), and clock phase (CPHA).

SPI Target Modes Table

Mode	CPOL (Idle)	CPHA (Phase)	Description
0	0 (LOW)	0 (Leading)	Data sampled on rising edge, shifted on falling
1	0 (LOW)	1 (Trailing)	Data sampled on falling edge, shifted on rising
2	1 (HIGH)	0 (Leading)	Data sampled on falling edge, shifted on rising
3	1 (HIGH)	1 (Trailing)	Data sampled on rising edge, shifted on falling

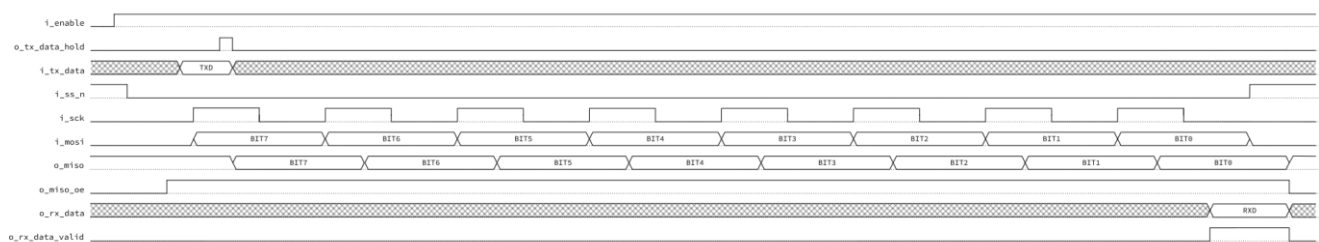
The SPI Target supports single or sequential multi frame transfers with frame length defined by WIDTH. When i_enable is asserted, the module waits for i_ss_n to be driven LOW. When i_ss_n is asserted, the SPI Target latches transmit data from i_tx_data into an internal buffer, indicated by a strobe on o_tx_data_hold, and shifts this data out to the SPI controller via o_miso. During an active SPI transaction, o_miso_oe remains HIGH, configuring the GPIO connected to o_miso as an output. When o_miso_oe is LOW, the GPIO is placed in a high impedance state. Upon completion of input and output frame transfers, the module asserts o_rx_data_valid to indicate valid data on o_rx_data and reasserts o_tx_data_hold in the case of multi frame transfers. The i_ss_n and i_sck inputs are internally synchronized to the system clock, introducing a latency on output signals. Therefore, the system clock frequency must be at least ten times higher than the SPI clock frequency.

CPHA = 0, CPOL = 0 (1 - when o_sck is inverted)



SPI Target Mode 0 operation

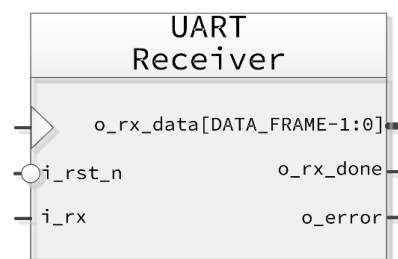
CPHA = 0, CPOL = 0 (1 - when o_sck is inverted)



SPI Target Mode 1 operation

4.1.10 UART Receiver

The UART module (Universal Asynchronous Receiver/Transmitter) is used for asynchronous serial communication with a configurable baud rate. The UART receiver converts an incoming serial binary stream into parallel data by detecting the START bit, sampling the data bits at the configured baud rate, optionally checking the PARITY bit, and validating one or more STOP bits.



UART Receiver Parameters

- **IN_CLK_HZ** – Input operating frequency in Hz (Minimum: 1_000_000, Maximum: 60_000_000).
- **DATA_FRAME** – Number of data bits per UART frame (Minimum: 5, Maximum: 9).
- **BAUD_RATE** – UART transmission baud rate (Supported values: 4_800, 9_600, 19_200, 28_800, 38_400, 57_600, 76_800, 115_200).
- **OVERSAMPLING_MODE** – Oversampling factor used for bit timing and sampling accuracy (Supported values: 8, 16).
- **PARITY_TYPE** – Parity configuration for error detection: 0 – PARITY_NONE, 1 – PARITY_EVEN, and 2 – PARITY_ODD.
- **STOP_BIT** – Stop bit length configuration 0 – 1 stop bit, 1 – 1.5 stop bits, and 2 – 2 stop bits.
- **LSB** – Determines the transfer format of output data and data order between least significant and most significant bit. When one, data starts from LSB, otherwise data starts from MSB.

Ports

Port name	Direction	Description
i_clk	input	System clock input
i_rst_n	input	Active-low reset input
i_rx	input	UART receive serial data input
o_rx_data	output	Received data output bus
o_rx_done	output	Receive done signal (asserted when a frame is received and parity error is not detected)
o_error	output	Indicates a parity mismatch in the received UART frame

Design steps

1. Add the **UART Receiver** to the design by selecting it from the **Module Library**.
2. Instantiate the **UART Receiver** in the top-level design file as shown in the following example.
3. Set the configuration parameters according to the application requirements.
4. Connect the **UART Receiver** control signals to the corresponding system logic.
5. Connect the i_rx input signal to the corresponding GPIO using the I/O Planner.

UART Receiver instantiation:

```
uart_receiver #(
    .IN_CLK_HZ          ( user_param ),
    .DATA_FRAME         ( user_param ),
    .BAUD_RATE          ( user_param ),
    .OVERSAMPLING_MODE ( user_param ),
    .PARITY_TYPE        ( user_param ),
    .STOP_BIT           ( user_param ),
    .LSB                ( user_param )
) user_inst_name (
    .i_clk      ( user_name ),
    .i_rst_n    ( user_name ),
    .i_rx       ( user_name ),
```

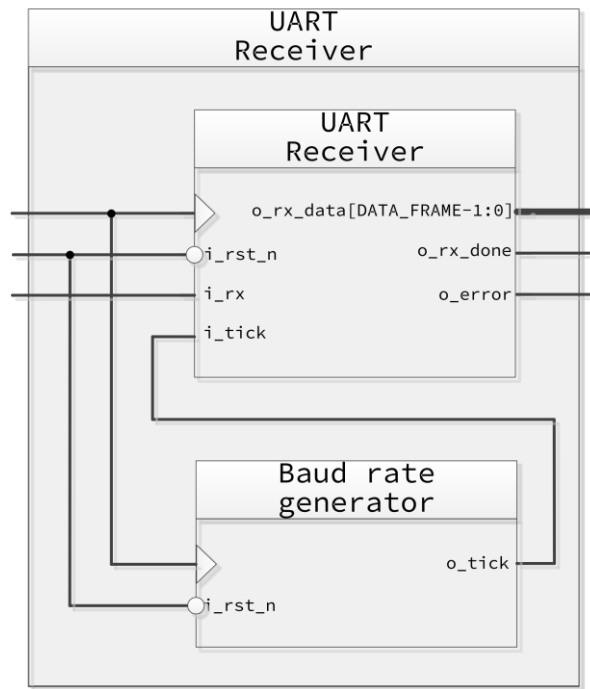
```

.o_rx_data      ( user_name ),
.o_rx_done      ( user_name ),
.o_error        ( user_name )
);

```

Module operation

The UART receiver module consists of a receiver and a baud rate generator. The baud rate generator derives receive timing from the system clock. The receiver receives the UART frame and reconstructs the parallel data word.



UART Receiver Structure

The UART Receiver module converts an incoming asynchronous serial data stream into parallel data with a configurable baud rate, data frame width, parity mode, stop-bit length, and bit reception order.

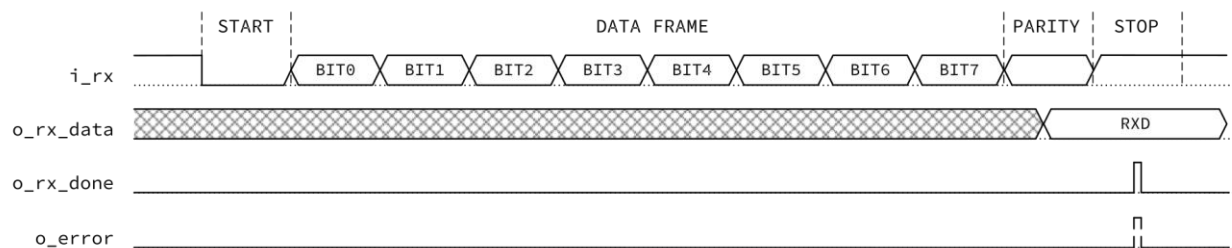
Reception begins when a valid START bit is detected on the `i_rx` input. To improve noise immunity and timing accuracy, the receive input is first synchronized to the system clock and the START bit is verified near the center of the bit period before data acquisition begins. Once a valid START bit is confirmed, the receiver samples the incoming UART frame according to the configured baud rate timing.

The received frame consists of a START bit, a configurable number of data bits, an optional PARITY bit, and one, one-and-a-half, or two STOP bits. Data bits are assembled into `o_rx_data` according to the selected transfer format. When `LSB = 1`, the least significant bit is received first. Otherwise, reception begins with the most significant bit.

Bit timing is generated by an internal baud rate generator, which derives periodic timing ticks from the system clock (`i_clk`) using the configured `IN_CLK_HZ`, `BAUD_RATE`, and `OVERSAMPLING_MODE` parameters. These timing ticks determine the sampling instants for all received bits.

If parity checking is enabled, the receiver compares the received parity bit against the parity calculated from the received data bits. In addition, the STOP-bit logic level is validated at the end of the frame. When reception completes successfully and no parity or STOP-bit errors are detected, the `o_rx_done` output generates a one-clock-cycle pulse and the received data is available on `o_rx_data`.

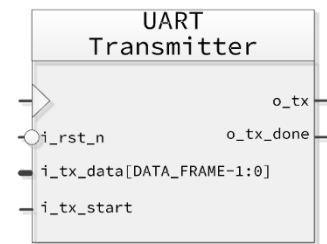
If a parity mismatch or invalid STOP bit is detected, the `o_error` output generates a one-clock-cycle pulse, indicating that the received UART frame is invalid. In this case, `o_rx_done` is not asserted.



UART Receiver Functional Waveform

4.1.11 UART Transmitter

The UART module (Universal Asynchronous Receiver/Transmitter) is used for asynchronous serial communication with a configurable baud rate. The UART transmitter converts parallel outgoing data into a serial binary stream by appending a START bit, one or more STOP bits, and an optional PARITY bit.



UART Transmitter Parameters

- **IN_CLK_HZ** – Input operating frequency in Hz (Minimum: 1_000_000, Maximum: 60_000_000).
- **DATA_FRAME** – Number of data bits per UART frame (Minimum: 5, Maximum: 9).
- **BAUD_RATE** – UART transmission baud rate (Supported values: 4_800, 9_600, 19_200, 28_800, 38_400, 57_600, 76_800, 115_200).
- **OVERSAMPLING_MODE** – Oversampling factor used for bit timing and sampling accuracy (Supported values: 8, 16).
- **PARITY_TYPE** – Parity configuration for error detection: 0 – PARITY_NONE, 1 – PARITY_EVEN, and 2 – PARITY_ODD.
- **STOP_BIT** – Stop bit length configuration 0 – 1 stop bit, 1 – 1.5 stop bits, and 2 – 2 stop bits.
- **LSB** – Determines the transfer format of output data and data order between least significant and most significant bit. When one, data starts from LSB, otherwise data starts from MSB.

Ports

Port name	Direction	Description
i_clk	input	System clock input
i_rst_n	input	Active-low reset input
o_tx	output	UART transmit serial data output
i_tx_data	input	Transmit data input bus
i_tx_start	input	Transmit start enable signal
o_tx_done	output	Transmission complete indication signal

Design steps

1. Add the **UART Transmitter** to the design by selecting it from the **Module Library**.
2. Instantiate the **UART Transmitter** in the top-level design file as shown in the following example.
3. Set the configuration parameters according to the application requirements.
4. Connect the **UART Transmitter** control signals to the corresponding system logic.
5. Connect the o_tx output signal to the corresponding GPIO using the I/O Planner.

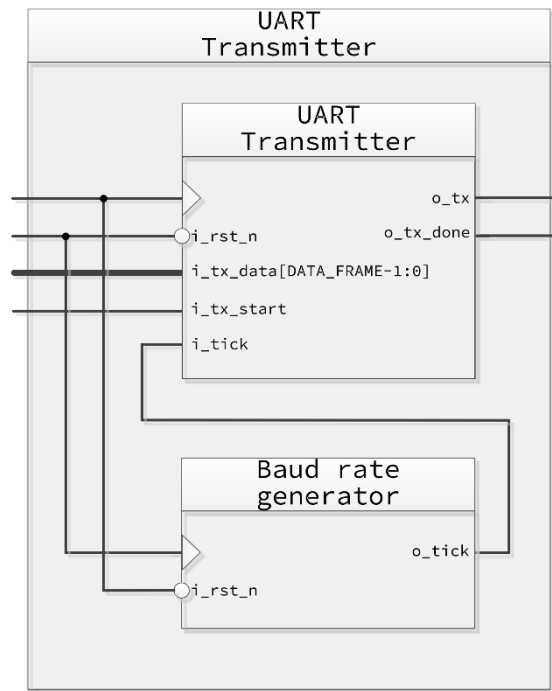
UART Transmitter instantiation:

```
uart_transmitter #(
    .IN_CLK_HZ          ( user_param ),
    .DATA_FRAME         ( user_param ),
    .BAUD_RATE          ( user_param ),
    .OVERSAMPLING_MODE ( user_param ),
    .PARITY_TYPE         ( user_param ),
    .STOP_BIT           ( user_param ),
    .LSB                ( user_param )
) user_inst_name (
    .i_clk      ( user_name ),
    .i_rst_n    ( user_name ),
    .o_tx       ( user_name ),
    .i_tx_data  ( user_name ),
    .i_tx_start ( user_name ),
```

```
.o_tx_done ( user_name )
);
```

Module operation

The UART transmitter module consists of a transmitter and a baud rate generator. The baud rate generator derives transmit timing from the system clock. The transmitter formats parallel input data into a UART frame and serializes it onto the transmit output.



UART Transmitter Structure

The UART Transmitter module supports transmission of parallel data over an asynchronous serial interface with a configurable baud rate, data frame format, parity mode, and number of stop bits.

A transmission is initiated when `i_tx_start` is asserted for at least one system clock cycle. Upon assertion, the transmitter captures `i_tx_data`, formats it into a UART frame by generating a START bit, an optional PARITY bit, and the configured STOP bit(s), and serializes the frame onto the `o_tx` output according to the configured baud rate.

The baud rate generator derives the transmit timing from the system clock (`i_clk`) based on the configured `BAUD_RATE`, `IN_CLK_HZ`, and `OVERSAMPLING_MODE` parameters and operates continuously.

The `o_tx_done` signal is asserted HIGH during generation of the STOP bit(s), indicating that the current UART frame transmission is completing and that the transmitter is ready to accept new data. While a transmission is in progress, additional assertions of `i_tx_start` are ignored.

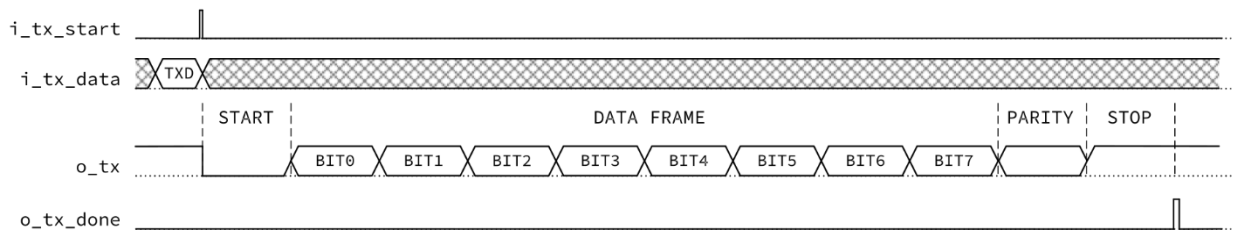
The UART Transmitter module converts parallel input data into an asynchronous serial data stream with a configurable baud rate, data frame width, parity mode, stop-bit length, and bit transmission order.

A transmission is initiated by asserting `i_tx_start` while the transmitter is idle. Upon reception of the start request, the module captures `i_tx_data`, calculates the optional parity bit, and begins transmitting a UART frame on the `o_tx` output. The transmitted frame consists of a START bit (0), the data bits, an optional PARITY bit, and one, one-and-a-half, or two STOP bits (1), depending on the configuration.

The order in which data bits are transmitted is controlled by the `LSB` parameter. When `LSB = 1`, the least significant bit is transmitted first. Otherwise, transmission starts with the most significant bit.

Bit timing is controlled by an internal baud rate generator, which derives periodic timing ticks from the system clock (*i_clk*) according to the configured *IN_CLK_HZ*, *BAUD_RATE*, and *OVERSAMPLING_MODE* parameters. These timing ticks define the duration of each transmitted bit.

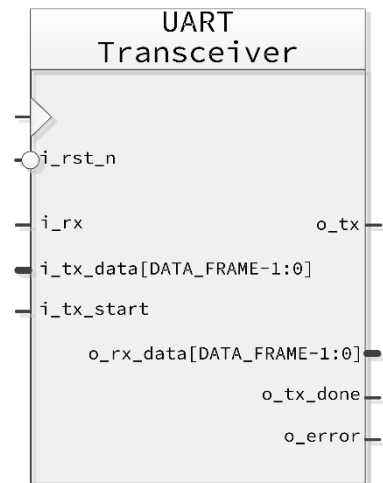
While a UART frame is being transmitted, the transmitter remains busy and ignores new data-loading requests. Upon completion of the configured STOP-bit period, the *o_tx_done* output generates a one-clock-cycle pulse indicating that the current frame transmission has finished and the transmitter is ready to accept a new transmission request. In the idle state, the *o_tx* output remains HIGH.



UART Transmitter Functional Waveform

4.1.12 UART Transceiver

The UART module (Universal Asynchronous Receiver/Transmitter) provides asynchronous serial communication with a configurable baud rate. It consists of a transmitter and a receiver. The transmitter converts parallel data into a serial data stream by appending a START bit, an optional PARITY bit, and one or more STOP bits. The receiver performs the reverse operation by detecting the START bit, sampling incoming data at the configured baud rate, optionally checking the PARITY bit, and validating the STOP bit(s).



UART Transceiver Parameters

- IN_CLK_HZ – Input operating frequency in Hz (Minimum: 1_000_000, Maximum: 60_000_000).
- DATA_FRAME – Number of data bits per UART frame (Minimum: 5, Maximum: 9).
- BAUD_RATE – UART transmission baud rate (Supported values: 4_800, 9_600, 19_200, 28_800, 38_400, 57_600, 76_800, 115_200).
- OVERSAMPLING_MODE – Oversampling factor used for bit timing and sampling accuracy (Supported values: 8, 16).
- PARITY_TYP – Parity configuration for error detection: 0 – PARITY_NONE, 1 – PARITY_EVEN, and 2 – PARITY_ODD.
- STOP_BIT – Stop bit length configuration 0 – 1 stop bit, 1 – 1.5 stop bits, and 2 – 2 stop bits.
- LSB – Determines the transfer format of output data and data order between least significant and most significant bit. When one, data starts from LSB, otherwise data starts from MSB.

Ports

Port name	Direction	Description
i_clk	input	System clock input
i_rst_n	input	Active-low reset input
i_rx	input	UART receive serial data input
o_tx	output	UART transmit serial data output
i_tx_data	input	Transmit data input bus
i_tx_start	input	Transmit start enable signal
o_tx_done	output	Transmission complete indication signal
o_rx_data	output	Received data output bus
o_rx_done	output	Receive done signal
o_error	output	Indicates a parity mismatch in the received UART frame

Design steps

1. Add the **UART Transceiver** to the design by selecting it from the **Module Library**.
2. Instantiate the **UART Transceiver** in the top-level design file as shown in the following example.
3. Set the configuration parameters according to the application requirements.
4. Connect the **UART Transceiver** control signals to the corresponding system logic.
5. Connect the i_rx input and the o_tx output signals to the corresponding GPIO using the I/O Planner.

UART Transceiver Instantiation:

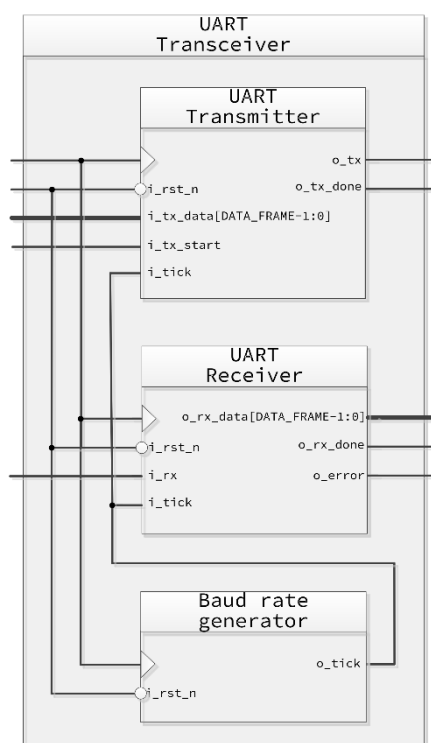
```

uart_transceiver #(
  .IN_CLK_HZ          ( user_param ),
  .DATA_FRAME        ( user_param ),
  .BAUD_RATE          ( user_param ),
  .OVERSAMPLING_MODE ( user_param ),
  .PARITY_TYPE        ( user_param ),
  .STOP_BIT           ( user_param ),
  .LSB                ( user_param )
) user_inst_name (
  .i_clk              ( user_name ),
  .i_rst_n            ( user_name ),
  .i_tx_data          ( user_name ),
  .i_tx_start         ( user_name ),
  .o_tx               ( user_name ),
  .o_tx_done          ( user_name ),
  .i_rx               ( user_name ),
  .o_rx_data          ( user_name ),
  .o_rx_done          ( user_name ),
  .o_error             ( user_name )
);

```

Module operation

The UART Transmitter module consists of a transmitter, a receiver and a baud rate generator. The baud rate generator derives transmit timing from the system clock. The transmitter formats parallel input data into a UART frame and serializes it onto the transmit output. The receiver receives the UART frame and reconstructs the parallel data word.

**UART Transceiver Structure**

The UART Transceiver module provides full-duplex asynchronous serial communication with configurable baud rate, data frame width, parity mode, stop-bit length, and bit transfer order. The transmitter and receiver operate independently and share a common baud-rate generator.

Transmit Path

A transmission is initiated by asserting `i_tx_start` while the transmitter is idle. The module captures `i_tx_data`, generates the optional parity bit, and transmits a UART frame consisting of a START bit, data bits, an optional PARITY bit, and the configured STOP bit(s). Data can be transmitted either LSB-first or MSB-first, depending on the `LSB` parameter.

The `o_tx_done` output generates a one-clock-cycle pulse when frame transmission is complete. Additional transmit requests are ignored while a frame is being transmitted.

Receive Path

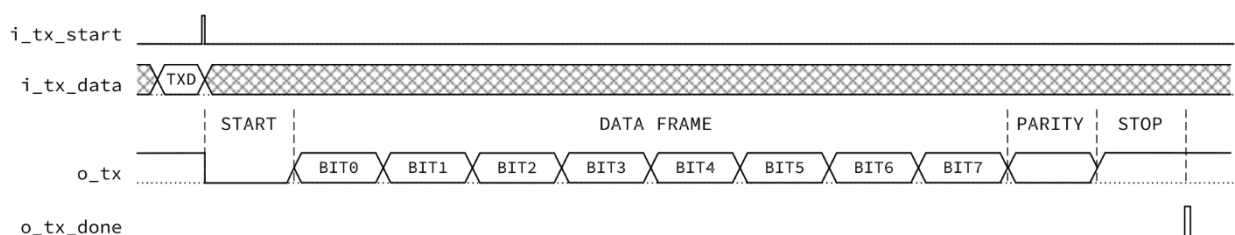
The receiver continuously monitors `i_rx` for a valid START bit. After START-bit verification, the incoming frame is sampled according to the configured baud rate and reconstructed into `o_rx_data`. Data can be received either LSB-first or MSB-first, depending on the `LSB` parameter.

If parity checking is enabled, the received parity bit is verified. The STOP bit(s) are also validated. When a frame is received successfully, `o_rx_done` generates a one-clock-cycle pulse. If a parity or STOP-bit error is detected, `o_error` generates a one-clock-cycle pulse, and the received frame is reported as invalid.

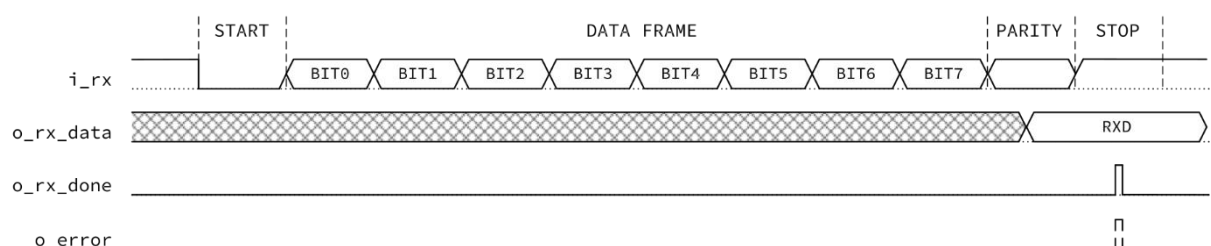
Timing

Bit timing is derived from `i_clk` using the configured `IN_CLK_HZ`, `BAUD_RATE`, and `OVERSAMPLING_MODE` parameters.

UART Transceiver functional timing waveforms



UART Transmitter Operation



UART Receiver Operation

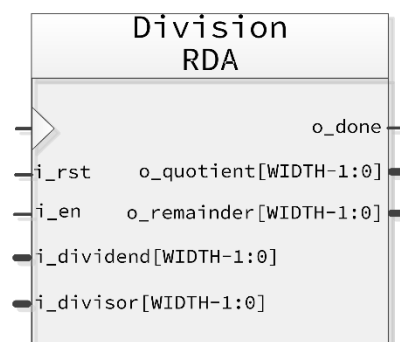
4.2 Mathematics

4.2.1 Division RDA

Restoring Division Algorithm (RDA) is a basic binary division technique used in digital systems to compute the quotient and remainder through repeated shifting and subtraction. If a subtraction produces a negative result, the previous remainder is restored.

Division RDA parameter:

- WIDTH – The operand bit width (Minimum: 1, Maximum: 32).



Ports

Port name	Direction	Description
i_clk	input	System clock input
i_rst	input	Active-high synchronous reset input
i_en	input	Enable signal to start the division operation
i_dividend	input	Dividend input value
i_divisor	input	Divisor input value
o_quotient	output	Computed quotient value
o_remainder	output	Computed remainder value
o_done	output	Division completion indicator

Design steps

1. Add the **Division RDA** module to the design by selecting it from the **Module Library**.
2. Instantiate the **Division RDA** in the design as shown in the following example.
3. Set the configuration parameter according to the application requirements.
4. Connect the **Division RDA** input/output signals to the corresponding system logic.

Division RDA module instantiation:

```
division_rda #(
    .WIDTH ( user_param )
) user_inst_name (
    .i_clk      ( user_name ),
    .i_rst      ( user_name ),
    .i_en       ( user_name ),
    .i_dividend ( user_name ),
    .i_divisor  ( user_name ),
    .o_quotient ( user_name ),
    .o_remainder ( user_name ),
    .o_done     ( user_name )
);
```

Module operation

The Division RDA module performs unsigned binary division using the Restoring Division Algorithm (RDA). The algorithm computes the quotient and remainder through iterative shift-and-subtract operations controlled by a finite state machine (FSM) and a fixed iteration counter.

Initialization (IDLE)

When *i_en* is asserted and *i_divisor* is not zero while the module is in the IDLE state, the internal combined remainder-and-quotient register is initialized as:

```
w_number = {WIDTH * {1'b0}, i_dividend}
```

The upper half of the register (partial remainder) is cleared to zero, while the lower half is loaded with the dividend. At the same time, the divisor is stored in an internal register for use during the division process. If `i_divisor` is zero, the division operation is not started, and the internal register remains cleared.

Iterative division (CALC)

In the CALC state, one restoring-division iteration is executed per clock cycle. First, the combined remainder-and-quotient register is shifted left by one bit. The divisor is then subtracted from the shifted partial remainder. The sign of the subtraction result determines the next operation:

- If the result is non-negative, the subtraction is accepted, the updated remainder is stored, and a 1 is inserted into the least significant quotient bit.
- If the result is negative, the remainder is restored to its previous value and a 0 is inserted into the least significant quotient bit.

This process is repeated for exactly `WIDTH` iterations. The iteration count is maintained by an internal counter that increments once per cycle while the module remains in the CALC state.

Completion (DONE)

When the iteration counter reaches its final value (`WIDTH - 1`), the module asserts `o_done` for one clock cycle, indicating that the division operation has completed. The division result is stored in the internal combined register `r_number`. The upper half of the register contains the remainder, and the lower half contains the quotient. After the final iteration `o_quotient` contains the calculated quotient and `o_remainder` contains the calculated remainder. The FSM then returns to the IDLE state and becomes ready to accept a new division request.

Reset behavior

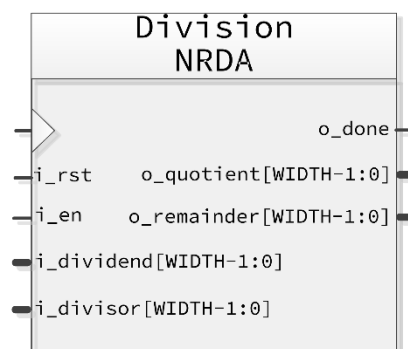
When `i_rst` is asserted, all internal registers, counters, and outputs are cleared, and the module returns to the IDLE state.

4.2.2 Division NRDA

Division NRDA (Non-Restoring Division Algorithm) is a binary division method that computes the quotient and remainder by shifting and conditionally adding or subtracting the divisor, without restoring the partial remainder after negative results.

Division NRDA parameter:

- **WIDTH** – The operand bit width (Minimum: 1, Maximum: 32).



Ports

Port name	Direction	Description
i_clk	input	System clock input
i_rst	input	Active-high synchronous reset input
i_en	input	Enable signal to start the division operation
i_dividend	input	Dividend input value
i_divisor	input	Divisor input value
o_quotient	output	Computed quotient value
o_remainder	output	Computed remainder value
o_done	output	Division completion indicator

Design steps

1. Add the **Division NRDA** module to the design by selecting it from the **Module Library**.
2. Instantiate the Division NRDA in the design as shown in the following example.
3. Set the configuration parameter according to the application requirements.
4. Connect the Division NRDA input/output signals to the corresponding system logic.

Division NRDA module instantiation:

```
division_nrda #(
    .WIDTH ( user_param )
) user_inst_name (
    .i_clk      ( user_name ),
    .i_rst      ( user_name ),
    .i_en       ( user_name ),
    .i_dividend ( user_name ),
    .i_divisor  ( user_name ),
    .o_quotient ( user_name ),
    .o_remainder ( user_name ),
    .o_done     ( user_name )
);
```

Module operation

The Division NRDA module performs unsigned binary division using an iterative non-restoring division algorithm. The operation is controlled by a two-state finite state machine (FSM) and a fixed-length iteration counter.

Initialization (IDLE)

When i_en is asserted and i_divisor is non-zero, the internal combined remainder-and-quotient register is initialized as:

```
w_number = {WIDTH * {1'b0}, i_dividend}
```

The upper half of the register (partial remainder) is cleared to zero, while the lower half is loaded with the dividend. The divisor is stored in an internal register for use during the division process.

Iterative Division (CALC)

In the CALC state, one division iteration is executed per clock cycle. The combined remainder-and-quotient register is shifted left by one bit, and the partial remainder is conditionally modified by adding or subtracting the divisor. Based on the resulting partial remainder, either a 1 or 0 is inserted into the least significant quotient bit.

The iteration counter increments once per cycle and the process continues for WIDTH iterations.

Completion

When the iteration counter reaches its final value (WIDTH - 1), the module asserts o_done for one clock cycle. The quotient and remainder are available on o_quotient and o_remainder.

After completion, the FSM returns directly to the IDLE state and becomes ready for a new division operation.

Reset Behavior

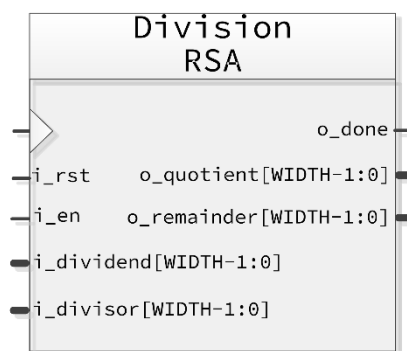
When i_rst is asserted, all internal registers, counters, and outputs are cleared, and the FSM returns to the IDLE state.

4.2.3 Division RSA

Division repeated subtraction algorithm is one of the methods of division that the same number is subtracted repeatedly from another larger number until the remainder is zero, or smaller than the number being subtracted.

Division RSA parameter:

- **WIDTH** – The operand bit width (Minimum: 1, Maximum: 32).



Ports

Port name	Direction	Description
i_clk	input	System clock input
i_rst	input	Active-high synchronous reset input
i_en	input	Enable signal to start the division operation
i_dividend	input	Dividend input value
i_divisor	input	Divisor input value
o_quotient	output	Computed quotient value
o_remainder	output	Computed remainder value
o_done	output	Division completion indicator

Design steps

1. Add the **Division RSA** module to the design by selecting it from the **Module Library**.
2. Instantiate the Division RSA in the design as shown in the following example.
3. Set the configuration parameter according to the application requirements.
4. Connect the Division RSA input/output signals to the corresponding system logic.

Division RSA module instantiation:

```
division_rsa #(
    .WIDTH ( user_param )
) user_inst_name (
    .i_clk      ( user_name ),
    .i_rst      ( user_name ),
    .i_en       ( user_name ),
    .i_dividend ( user_name ),
    .i_divisor  ( user_name ),
    .o_quotient ( user_name ),
    .o_remainder ( user_name ),
    .o_done     ( user_name )
);
```

Module operation

The Division RSA module performs unsigned integer division using the Repeated Subtraction Algorithm (RSA). The division process is controlled by a three-state finite state machine (FSM) consisting of IDLE, CALC, and DONE states and operates synchronously with the input clock.

When i_en is asserted, the input dividend and divisor are stored in internal registers. If the divisor is non-zero, the FSM transitions from IDLE to CALC. Otherwise, it proceeds directly to DONE.

In the CALC state, the divisor is repeatedly subtracted from the internal dividend register on each clock cycle. After each subtraction, the quotient counter is incremented by one. This process continues while the current dividend value is greater than or equal to the divisor.

When the remaining dividend becomes smaller than the divisor, the FSM transitions to the DONE state. The remaining dividend value is captured as the remainder, o_done is asserted for one clock cycle, and the valid quotient and remainder values are available on o_quotient and o_remainder, respectively.

After completion, the FSM automatically returns to the IDLE state, where the outputs are cleared and the module becomes ready for the next division operation.

When i_rst is asserted, all internal registers, outputs, and control signals are synchronously cleared, and the FSM returns to the IDLE state.

4.2.4 Serial CRC

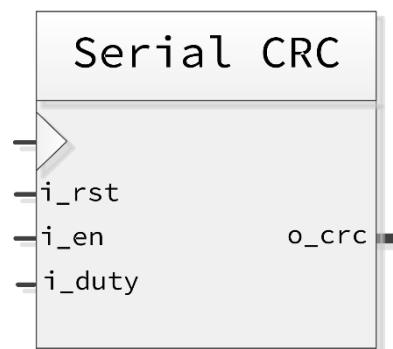
Cyclic Redundancy Check is used to detect errors in digital data. This module calculates CRC value in sequential mode, allowing y to specify polynomial and data width through parameters.

Serial CRC Parameters

- WIDTH – Width of the CRC word.
- POLYNOMIAL – CRC polynomial for result calculation.
- XOR_OUT – XOR output data
- INITIAL – Initial value before the start of CRC calculation.
- REF_INOUT – CRC data reverse mode (LSB first if 1'b1).

Ports

Port name	Direction	Description
i_clk	input	System clock input
i_rst	input	Active high reset input
i_en	input	Signal that enables an CRC operation
i_data	input	Serial data input signal
o_crc	output	Calculated CRC value



Design steps

1. Add the **Serial CRC** to the design by selecting it from the **Module Library**.
2. Instantiate the Serial CRC in the top-level design file as shown in the following example.
3. Set the configuration parameters according to the application requirements.
4. Connect the control signals to the corresponding system logic.

Serial CRC instantiation:

```
serial_crc #(
    .WIDTH      ( user_param ),
    .POLYNOMIAL ( user_param ),
    .XOR_OUT     ( user_param ),
    .INITIAL     ( user_param ),
    .REF_INOUT   ( user_param )
) user_inst_name (
    .i_clk  ( user_name ),
    .i_rst  ( user_name ),
    .i_en   ( user_name ),
    .i_data ( user_name ),
    .o_crc  ( user_name )
);
```

Module operation

The simple Serial CDC module calculates CRC value as long as the i_en signal is asserted.

To enable operation, the i_en signal should be asserted, and the i_rst signal should be deasserted. The module calculates CRC value every clock cycle based on previously entered data on the i_data line. Calculated data appears on the o_crc bus the next i_clk rising edge after the i_data is consumed. When the new chunk of data should be calculated, it is required to reset the module for correct calculations. After the reset, the base value defined in the INITIAL parameter is set. The XOR_OUT parameter is used to specify the value which is xored with the calculated data before output. If the data should be consumed and outputted LSB first, then the REF_INOUT parameter should be 1'b1.

Serial CRC Operation

4.2.5 Hamming Encoder

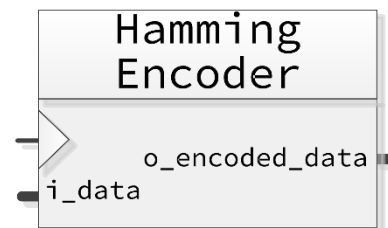
Hamming Codes are error-correcting codes (ECC) capable of detecting and correcting one-bit errors in data. The encoder calculates and inserts additional parity bits into the data word to encode the data before transmitting.

Hamming Encoder Parameters

- DATA_WIDTH – Original data bus width.

Ports

Port name	Direction	Description
i_clk	input	System clock input
i_data	input	Input data bus
o_encoded_data	output	Output encoded data bus



Design steps

1. Add the **Hamming encoder** to the design by selecting it from the **Module Library**.
2. Instantiate the encoder in the top-level design file as shown in the following example.
3. Set the configuration parameters according to the application requirements.
4. Connect the control signals to the corresponding system logic.
5. Connect the output bus to the corresponding system logic.

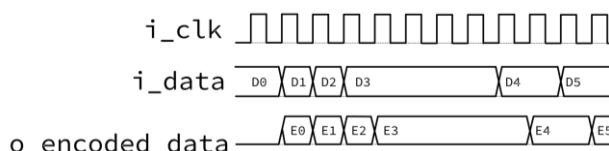
Hamming Encoder instantiation:

```
hamming_encoder #(
    .DATA_WIDTH ( user_param )
) user_inst_name (
    .i_clk      ( user_name ),
    .i_data     ( user_name ),
    .o_encoded_data ( user_name )
);
```

Module operation

The module encodes incoming data on the i_data bus and outputs the result on the o_encoded_data bus.

The module does not require enable signal for operation, encoding occurs every clock cycle. The module needs one clock cycle to consume input data, the output appears on the subsequent rising edge of i_clk. This simple encoder supports various data length defined by the DATA_WIDTH parameter. The encoder supports the classic Hamming(7,4), Hamming(15,11), and Hamming(31,26) codes. It also supports Hamming(12,8), Hamming(17,12), Hamming(21,16), Hamming(25,20), and Hamming(29,24) codes. For example, to select Hamming(12,8), the DATA_WIDTH parameter should be set to 8.



Hamming Encoder Operation

4.2.6 Hamming Decoder

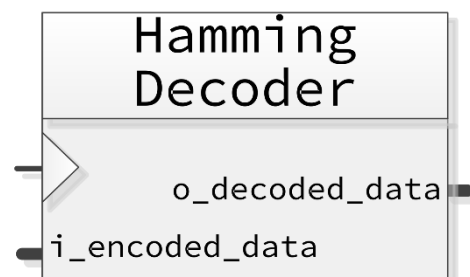
Hamming Codes are error-correcting codes (ECC) capable of detecting and correcting one-bit errors in data. The decoder detects error in the encoded message and automatically flips corrupted bit.

Hamming Decoder Parameters

- DATA_BITS – Amount of data bits in the encoded word.

Ports

Port name	Direction	Description
i_clk	input	System clock input
i_encoded_data	input	Input encoded data bus
o_decoded_data	output	Output decoded data bus



Design steps

1. Add the **Hamming decoder** to the design by selecting it from the **Module Library**.
2. Instantiate the decoder in the top-level design file as shown in the following example.
3. Set the configuration parameters according to the application requirements.
4. Connect the control signals to the corresponding system logic.
5. Connect the output bus to the corresponding system logic.

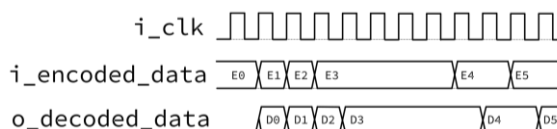
Hamming Decoder instantiation:

```
hamming_decoder #(
    .DATA_BITS ( user_param )
) user_inst_name (
    .i_clk      ( user_name ),
    .i_encoded_data ( user_name ),
    .o_decoded_data ( user_name )
);
```

Module operation

The module decodes incoming data on the i_encoded_data bus and outputs the result on the o_decoded_data bus.

The module does not require enable signal for operation, decoding occurs every clock cycle. The module needs one clock cycle to consume input data, the output appears on the subsequent rising edge of i_clk. This simple decoder supports various data length defined by the DATA_BITS parameter. The decoder supports the classic Hamming(7,4), Hamming(15,11), and Hamming(31,26) codes. It also supports Hamming(12,8), Hamming(17,12), Hamming(21,16), Hamming(25,20), and Hamming(29,24) codes. For example, to select Hamming(12,8), the DATA_BITS parameter should be set to 8.



Hamming Decoder Operation

4.3 Memory

4.3.1 Synchronous FIFO

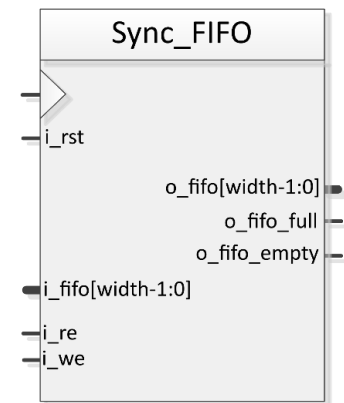
The synchronous FIFO module is a type of memory which deals with first-in, first-out buffer. This memory is used for in-order data storage and retrieval.

FIFO Parameters

- **WIDTH** – The width of bus or number of wires dedicated (Type: Decimal; Default Value: 8, Min value: 4, Max Value: 63).
- **DEPTH** – The depth of the FIFO (Type: Decimal; Default Value: 8, Min Value: 1, Max Value: 63*).

Ports

Port name	Direction	Description
i_clk	input	System clock input
i_rst	input	Active reset input
i_fifo	input	Input data bus signal
i_we	input	Input write enable
i_re	input	Input read enable
o_fifo	output	Output data signal
o_fifo_empty	output	Signal that indicates when FIFO is empty
o_fifo_full	output	Signal that indicates when FIFO is full



Design Steps

1. Add the sync_fifo to the design by selecting it from the Module Library.
2. Instantiate the sync_fifo in the top-level design file as shown in the following example.
3. Set the configuration parameters according to the application requirements.
4. Connect the control signals to the corresponding system logic.

FIFO Instantiation:

```
sync_fifo #(
    .WIDTH          ( user_param ),
    .DEPTH          ( user_param )
) user_inst_name (
    .i_clk          ( user_name ),
    .i_rst          ( user_name ),
    .i_fifo         ( user_name ),
    .i_we           ( user_name ),
    .i_re           ( user_name ),
    .o_fifo         ( user_name ),
    .o_fifo_full    ( user_name ),
    .o_fifo_empty   ( user_name )
);
```

Module Operation

This Verilog implements a synchronous FIFO: a small first-in, first-out buffer where both reading and writing happen on the same clock, i_clk. The FIFO stores WIDTH-bit words and can hold up to DEPTH entries. Internally, r_fifo is the storage array, r_write_pointer marks where the next write will go, r_read_pointer marks where the next read will come from, and r_count tracks how many valid items are currently stored. On reset, the pointers and count are cleared, which effectively makes the FIFO empty.

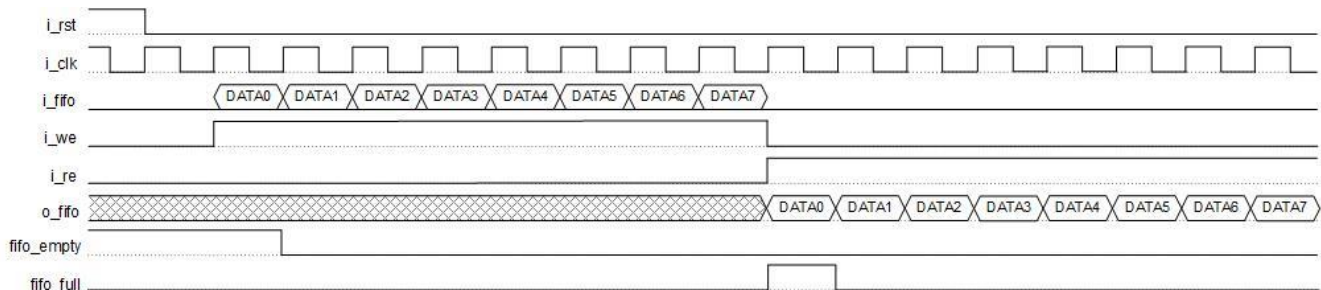
On each rising clock edge, the design checks i_we and i_re.

If i_we is asserted, it writes i_fifo into the current write location and increments the write pointer.

If `i_re` is asserted, it reads the current read location into `o_fifo` and increments the read pointer.

Separately, the count is updated: write-only increments it, read-only decrements it, and simultaneous read/write leaves it unchanged because one item enters while one leaves. The outputs `o_fifo_full` and `o_fifo_empty` are just comparisons on `r_count`, indicating when the FIFO has reached DEPTH items or zero items.

The respective signals can be observed in the timing diagram. It `o_fifo` that reads out the data written in. And subsequently, the `o_fifo_empty` flag is turned on after all the data has been read out from the stored data.



Synchronous FIFO operation

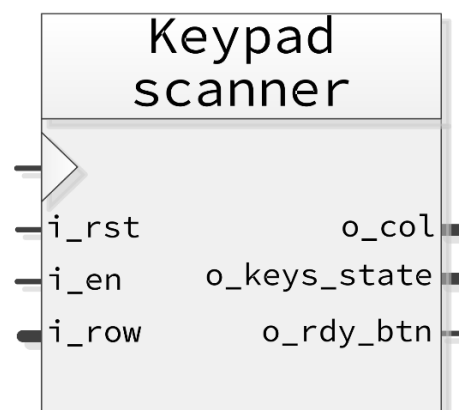
4.4 Periphery

4.4.1 Keyboard Scanner

Keyboard Scanner is used to scan matrix keypad and detect which key is being pressed.

Keyboard Scanner Parameters

- IN_CLK_HZ – Input clock frequency.
- DEBOUNCE_VALUE_MS – Input debounces time value in ms.
- SCAN_VALUE_US – Scan time for one row in μ s.
- COL_NUM – Number of columns.
- ROW_NUM – Number of rows.



Ports

Port name	Direction	Description
i_clk	input	System clock input
i_rst	input	Active high reset input
i_row	input	Input bus for rows
o_col	output	Output bus for columns
o_keys_state	output	Output state of keys
o_rdy_btn	output	Output ready signal

Design steps

1. Add the **Keyboard Scanner** to the design by selecting it from the **Module Library**.
2. Instantiate the scanner in the top-level design file as shown in the following example.
3. Set the configuration parameters according to the application requirements.
4. Connect the control signals to the corresponding system logic.
5. Connect the external ports to the matrix keyboard.

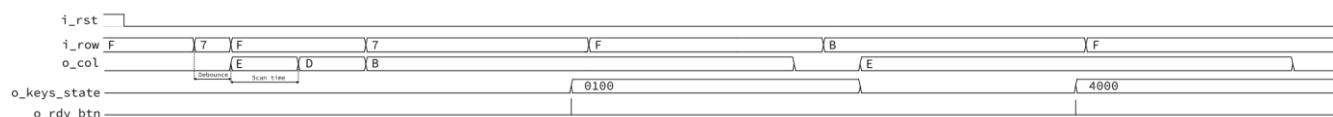
Keyboard Scanner instantiation:

```
keypad_scanner #(
    .IN_CLK_HZ          ( user_param ),
    .DEBOUNCE_VALUE_MS ( user_param ),
    .SCAN_VALUE_US      ( user_param ),
    .COL_NUM            ( user_param ),
    .ROW_NUM            ( user_param )
) user_inst_name (
    .i_clk      ( user_name ),
    .i_rst      ( user_name ),
    .i_row      ( user_name ),
    .o_col      ( user_name ),
    .o_keys_state ( user_name ),
    .o_rdy_btn  ( user_name )
);
```

Module operation

The module scans each matrix column to detect a pressed key.

To enable operation, the `i_rst` should be deasserted. When no buttons are pressed, all row pins are pulled high by pull-up resistors, so the `i_row` value is all ones. Scanning does not run continuously. It starts only when a button press is detected. To detect a key press, the `o_col` value in the idle state is kept low, this enables monitoring of all columns. When a key press is detected, the module waits for the debounce time specified in the `DEBOUNCE_VALUE_MS` parameter and if value did not change during this time, it checks every line until it does not identify a pressed key. Each column is selected by pulling one of the pins low, while other remain high. Each column is checked for time specified by the `SCAN_VALUE_US` parameter, and if the scanner does not detect any zero on the `i_row` bus after timeout it moves to the next column. After key identification and key release, the `o_rdy_button` signal generates one-clock-cycle strobe, and the `o_keys_state` bus outputs currently pressed button. This value is latched until the next scanning begins. During scanning, the `o_keys_state` is reset and equals to all zero. Note that this module does not support multiple key detection. The number of rows can be parametrized by the `ROW_NUM` parameter, and the number of columns is specified by the `COL_NUM` parameter. To ensure proper timings, the input design clock frequency should be set using the `IN_CLK_HZ` parameter.



Keypad scanner operation

4.4.2 Servo Controller

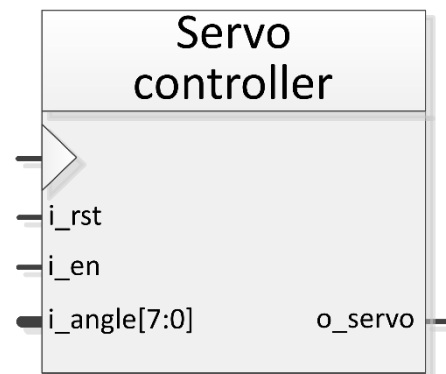
The Servo Controller is a module responsible for controlling a servo motor. Servo control is achieved by sending the servo a PWM (Pulse Width Modulation) signal with a specific duty cycle (pulse width). This signal determines the target position of the servo.

Servo Controller Parameters

- **IN_CLK_HZ** – Specifies the input system clock frequency in Hz. This parameter is used to derive internal timing for PWM generation.
- **WIDTH_MIN_US** – Specifies the minimum pulse width value, defined in μ s.
- **WIDTH_MAX_US** – Specifies the maximum pulse width value, defined in μ s.

Ports

Port name	Direction	Description
i_clk	input	System clock input
i_rst	input	Active reset input
i_en	input	Start signal that initiates PWM generation
i_angle	input	Data byte that defines the pulse width
o_servo	output	Servo control signal (PWM)



Design steps

1. Add the **Servo Controller** to the design by selecting it from the **Module Library**.
2. Instantiate the Servo Controller in the top-level design file as shown in the following example.
3. Set the configuration parameters according to the application requirements.
4. Connect the controller control signals to the corresponding system logic.
5. Connect the PWM signal to the Servo control input.

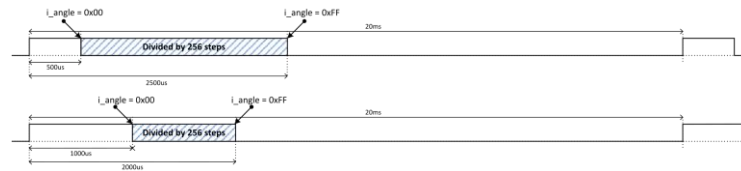
Servo Controller instantiation:

```
servo_ctrl #(
    .IN_CLK_HZ      (user_param),
    .WIDTH_MIN_US   (user_param),
    .WIDTH_MAX_US   (user_param)
) user_inst_name (
    .i_clk      (user_port_name),
    .i_rst      (user_port_name),
    .i_en       (user_port_name),
    .i_angle    (user_port_name),
    .o_servo    (user_port_name)
);
```

Module operation

The servo_ctrl module generates a servo-compatible PWM control signal based on an input angle value. It converts an 8-bit digital angle command into a pulse width within a configurable minimum and maximum range, following standard RC servo timing conventions.

The module produces a 50 Hz PWM signal (20 ms period), commonly required by RC servos. The high-time of the PWM signal determines the servo position. Pulse width is linearly scaled between programmable minimum and maximum values using the i_angle input.



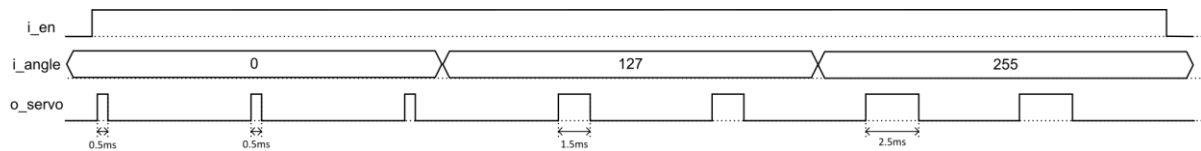
Servo Controller Pulse Width

The input clock frequency is defined by the IN_CLK_HZ parameter. An internal counter (r_cnt_pwm) is used to divide this clock down to the required 50 Hz PWM period. CNT represents the number of input clock cycles per PWM period. CNT_WIDTH sets the required counter bit width. When the enable signal i_en is HIGH, the counter increments on each rising edge of i_clk. When the counter reaches CNT, it resets to zero, starting a new PWM cycle. This ensures accurate and stable PWM timing derived from the system clock.

Servo pulse width is calculated in clock cycles using the parameters:

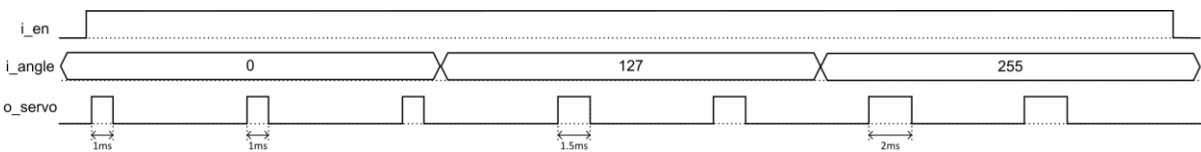
- WIDTH_MIN_US – Minimum pulse width in μ s.
- WIDTH_MAX_US – Maximum pulse width in μ s.

Timing diagram for servo control, at WIDTH_MIN_US = 500 and WIDTH_MAX_US = 2500:



Servo Controller Operation at WIDTH_MIN_US = 500 and WIDTH_MAX_US = 2500

Timing diagram for servo control, at WIDTH_MIN_US = 1000 and WIDTH_MAX_US = 2000:



Servo Controller Operation at WIDTH_MIN_US = 1000 and WIDTH_MAX_US = 2000

i_rst a synchronous reset initializes the PWM counter and forces the servo output LOW. When the enable signal i_en is LOW, the counter halts and the output remain LOW. This allows the servo control signal to be safely disabled.

The o_servo output is generated by comparing the PWM counter value with the calculated pulse width:

- At the start of each PWM period, o_servo is driven HIGH.
- While r_cnt_pwm is less than the calculated pulse width, the output remains HIGH.
- Once the counter exceeds the pulse width, o_servo is driven LOW for the remainder of the period.

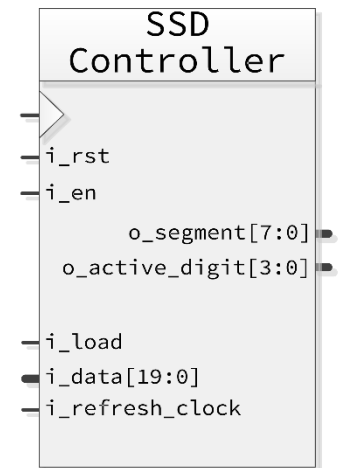
This results in a fixed-frequency PWM signal with a variable duty cycle.

4.4.3 7-Segment Display Controller

A 7-segment display controller is a digital logic module responsible for converting numerical or symbolic data into the corresponding segment control signals required to drive a seven-segment display.

7-segment Display Controller Parameters

- SEL_CA – controls the segment output polarity based on the selected display type (0 – common anode, 1 – common cathode).



Ports

Port name	Direction	Description
i_clk	input	System clock input. Used to synchronize all internal logic of the display controller.
i_rst	input	Active-high reset signal. Resets the module to its initial state.
i_en	input	Display enable signal, the display output is disabled or refresh is halted.
i_load	input	Load strobe signal used to latch new data into an external counter or display register.
i_refresh_clock	input	Refresh clock generated for the external counter. Defines the dynamic display frequency.
i_data	input	Data bus carrying digit or segment data to the external display logic.
o_active_digit	output	Digit select signals for the seven-segment display.
o_segment	output	Seven-segment LED control outputs: "[a, b, c, d, e, f, g, dp]".

Design steps

1. Add the **7-segment display controller** to the design by selecting it from the **Module Library**.
2. Instantiate the controller in the design file as shown in the following example.
3. Set the configuration parameters according to the application requirements.
4. Connect the control signals to the corresponding system logic.
5. Connect the external ports to the 7-segment display controller.

Seven Segment Display Controller Instantiations:

```

seven_seg_disp_ctrl_1d #(
    .SEL_CA          ( user_param )
) user_inst_name (
    .i_clk           ( user_name ),
    .i_rst           ( user_name ),
    .i_en            ( user_name ),
    .i_load          ( user_name ),
    .i_refresh_clock ( user_name ),
    .i_data          ( user_name ),
    .o_segment       ( user_name )
);

seven_seg_disp_ctrl_2d #(
    .SEL_CA          ( user_param )
) user_inst_name (
    .i_clk           ( user_name ),
    .i_rst           ( user_name ),
    .i_en            ( user_name ),
    .i_load          ( user_name ),
    .i_refresh_clock ( user_name ),
    .i_data          ( user_name ),
    .o_active_digit  ( user_name ),

```

Module Operation

The Seven-Segment Display Controller modules provide a configurable interface for driving 1-digit, 2-digit, or 4-digit seven-segment displays. The modules support both common-anode and common-cathode display types through the SEL_CA parameter and can display hexadecimal values (0–F) with optional decimal-point control.

Input data is supplied through the i_data bus and loaded into an internal register when i_load is asserted. Each displayed digit consists of a 4-bit hexadecimal value and an associated decimal-point control bit. Once loaded, the displayed value remains unchanged until new data is written.

Input	a	b	c	d	e	f	g	display
0000	0	0	0	0	0	0	1	0
0001	1	0	0	1	1	1	1	1
0010	0	0	1	0	0	1	0	2
0011	0	0	0	0	1	1	0	3
0100	1	0	0	1	1	0	0	4
0101	0	1	0	0	1	0	0	5
0110	0	1	0	0	0	0	0	6
0111	0	0	0	1	1	1	1	7
1000	0	0	0	0	0	0	0	8
1001	0	0	0	0	1	0	0	9
1010	0	0	0	1	0	0	0	A
1011	1	1	0	0	0	0	0	b
1100	0	1	1	0	0	0	1	C
1101	1	0	0	0	0	1	0	d
1110	0	1	1	0	0	0	0	E
1111	0	1	1	1	0	0	0	F

For the 2-digit and 4-digit versions, the display is driven using dynamic multiplexing. An external refresh signal (*i_refresh_clock*) sequentially activates each display digit while selecting the corresponding data nibble and decimal-point bit from the stored input data. The active digit rotates continuously, allowing all digits to appear illuminated simultaneously due to persistence of vision. The refresh rate is determined by the source generating *i_refresh_clock*.

The selected hexadecimal digit is passed through an internal decoder that converts the 4-bit value into the corresponding seven-segment pattern. The decoder supports hexadecimal characters 0 through F. Segment patterns are automatically generated with the correct polarity for the selected display type (common-anode or common-cathode).

When *i_en* is asserted, the decoded segment data and decimal-point control are driven onto the display outputs. When *i_en* is deasserted, all segments are disabled, blanking the display.

The 1-digit version operates similarly but does not require multiplexing because only a single digit is present. The loaded digit value is decoded directly and continuously driven to the segment outputs while the decimal point is controlled by the most significant bit of the input data.

The *o_segment* output provides the segment control signals [*a*, *b*, *c*, *d*, *e*, *f*, *g*, *dp*]. For multiplexed implementations, *o_active_digit* indicates the currently active display digit and should be connected to the display common terminals according to the selected display type.

4.4.4 Smart LED Controller

Smart LED Controller is a configurable module for driving addressable LED strings such as WS2812 and SK6812, supporting precise single-wire timing and GRB/GRBW data formats. It enables reliable control of chained smart LEDs with programmable timing, LED count, and colour width.

UART Transmitter Parameters

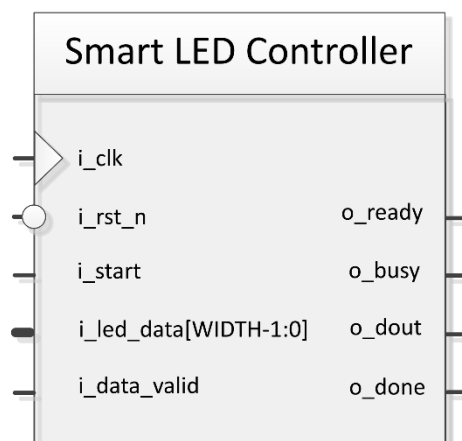
- IN_CLK_HZ – Input operating frequency in Hz (Minimum: 10_000_000, Maximum: 50_000_000).
- WIDTH – Pixel data width per LED. Supported values are 24 bits (GRB) for RGB LEDs and 32 bits (GRBW) for RGBW LEDs (Supported values: 24, 32).
- LED_NUM – Number of LEDs connected in the daisy-chained LED string (Minimum: 1, Maximum: 1024).
- T0H – High-level duration, in ns, representing a logical 0 bit (Minimum: 200, Maximum: 500).
- T1H – High-level duration, in ns, representing a logical 1 bit (Minimum: 600, Maximum: 1000).
- T0L – Low-level duration, in ns, following a logical 0 bit (Minimum: 200, Maximum: 500).
- T1L – Low-level duration, in ns, following a logical 1 bit (Minimum: 600, Maximum: 1000).
- RES – Reset (latch) time in μ s. Defines the minimum low-level period required to latch transmitted data and reset the LED internal state (Minimum: 50, Maximum: 100).

Ports

Port name	Direction	Description
i_clk	input	System clock input
i_rst_n	input	Active-low reset input
i_start	input	Starts LED data transmission when asserted
i_led_data	input	Pixel data input bus in GRB or GRBW format
i_data_valid	input	Indicates that i_led_data contains valid pixel data
o_ready	output	Indicates that the controller is ready to accept new pixel data
o_busy	output	Asserted while LED data is being transmitted
o_dout	output	Single-wire serial data output to the LED chain
o_done	output	Indicates completion of the full LED chain transmission and reset period

Design steps

1. Add the **Smart LED Controller** to the design by selecting it from the **Module Library**.
2. Instantiate the Smart LED Controller in the design as shown in the following example.
3. Set the configuration parameters according to the application requirements.
4. Connect the Smart LED Controller signals to the corresponding system logic.
5. Connect the o_dout output signal to the corresponding GPIO using the I/O Planner.



Smart LED Controller instantiation:

```
smart_led_controller #(
    .IN_CLK_HZ      ( user_param ),
    .WIDTH          ( user_param ),
    .LED_NUM        ( user_param ),
    .T0H            ( user_param ),
    .T1H            ( user_param ),
    .T0L            ( user_param ),
    .T1L            ( user_param ),
    .RES            ( user_param )
) user_inst_name (
    .i_clk          ( user_name ),
    .i_rst_n        ( user_name ),
    .i_start        ( user_name ),
    .i_led_data     ( user_name ),
    .i_data_valid   ( user_name ),
    .o_ready        ( user_name ),
    .o_busy         ( user_name ),
    .o_dout         ( user_name ),
    .o_done         ( user_name )
);
```

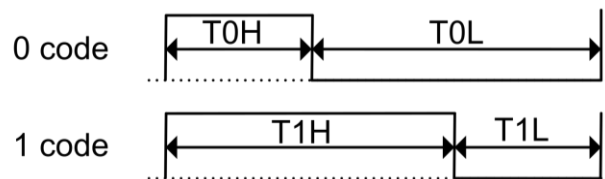
Module operation

The Smart LED Controller module implements a timing-accurate single-wire transmitter for addressable smart LEDs such as WS2812 and SK6812. It converts parallel pixel data into a serialized waveform that complies with smart LED protocol timing requirements. All timing is derived from the system clock and programmable timing parameters.

The module operates synchronously with the system clock (*i_clk*) and uses configurable parameters to generate the required HIGH and LOW pulse widths for logical 0 and 1 bits, as well as the reset (latch) period. Pixel data is handled in GRB or GRBW format depending on the configured data width.

Transmission is initiated when *i_start* is asserted. Once started, the controller enters an active state and asserts *o_busy* and *o_ready* outputs. The *o_busy* is HIGH during LEDS data transmission and reset period. The *o_ready* indicates that the controller is ready to accept new pixel data. Pixel data for each LED is provided sequentially on *i_led_data* and accepted when *i_data_valid* is asserted while *o_ready* is HIGH. For each bit of the pixel data word, the controller generates a corresponding waveform on *o_dout*:

- A logical 0 is transmitted using the configured T0H and T0L timing.
- A logical 1 is transmitted using the configured T1H and T1L timing.



Smart LED Controller bit timings

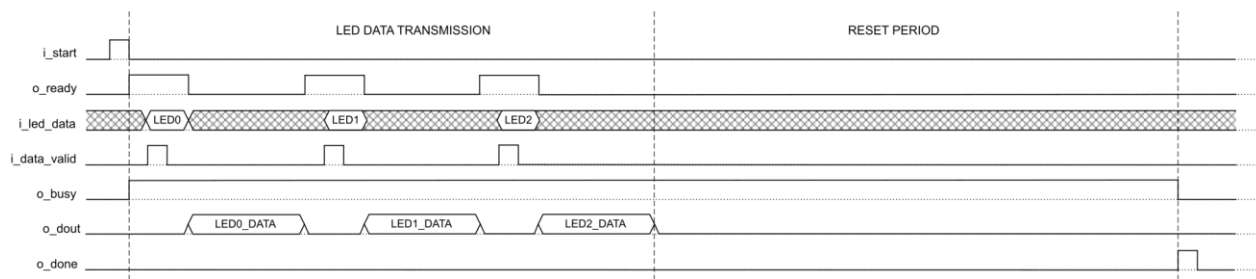
Bits are transmitted MSB-first. After all bits of a pixel are sent, the controller advances to the next LED in the chain until LED_NUM pixels have been transmitted.

After transmission of all LED data, the controller holds the output low for the configured reset time (RES) to latch the data into the LED string. This reset interval ensures proper update of all connected LEDs.

When the full transmission sequence and reset period complete:

- *o_busy* is deasserted
- *o_done* is asserted for one clock cycle
- The controller returns to the idle state, ready for the next transmission

The active-low reset input (*i_rst_n*) resets the controller to a known idle state, clears internal counters, and drives the output low.



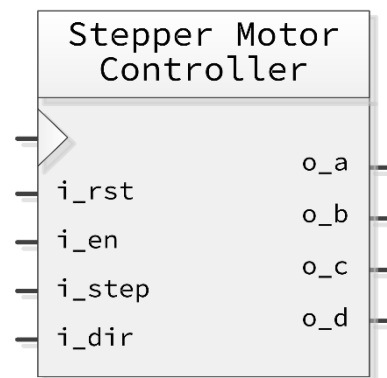
Smart LED Controller Operation

4.4.5 Stepper Motor Controller

Stepper motor controllers manage the position, speed and torque of a stepper motor by directing pulses to a driver, allowing for precise angular movement. This simple module controls unipolar motor allowing for wave and full control modes.

Stepper Motor Controller Parameters

- **TWO_PHASES_ACTIVE** – Defines whether motor is single phase controlled or dual phase controlled.
- **MICROSTEPS** – Defines number of microsteps per step.
- **IN_CLK_HZ** – Defines system clock frequency for correct PWM calculations.
- **PWM_FREQ_HZ** – Defines output PWM frequency.



Ports

Port name	Direction	Description
i_clk	input	System clock input
i_en	input	Enable signal
i_rst	input	Active high reset signal
i_step	input	One step signal
i_dir	input	Stepping direction signal
o_a	output	Control signal A
o_b	output	Control signal B
o_c	output	Control signal C
o_d	output	Control signal D

Design steps

1. Add the stepper motor controller to the design by selecting it from the **Module Library**.
2. Instantiate the controller in the top-level design file as shown in the following example.
3. Set the configuration parameters according to the application requirements.
4. Connect the control signals to the corresponding system logic.
5. Connect the output control signals to the desired GPIOs.

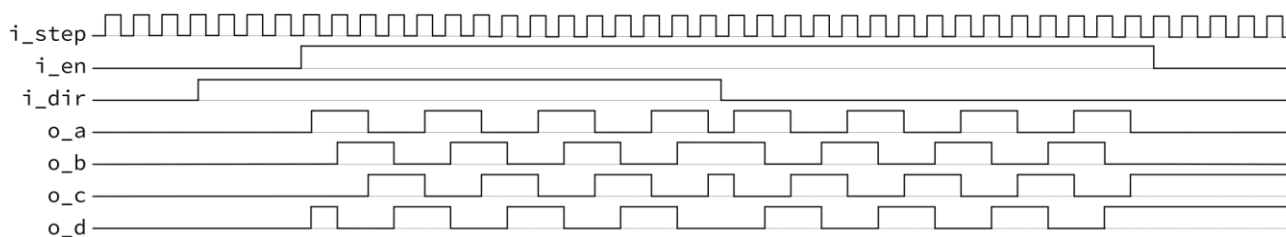
Stepper Motor Controller instantiation:

```
stepper_ctrl_multimode #(
    .TWO_PHASES_ACTIVE ( user_param )
    .MICROSTEPS         ( user_param )
    .IN_CLK_HZ          ( user_param )
    .PWM_FREQ_HZ        ( user_param )
) user_inst_name (
    .i_clk  ( user_name ),
    .i_rst  ( user_name ),
    .i_en   ( user_name ),
    .i_step ( user_name ),
    .i_dir  ( user_name ),
    .o_a    ( user_name ),
    .o_b    ( user_name ),
    .o_c    ( user_name ),
    .o_d    ( user_name )
);
```

Module operation

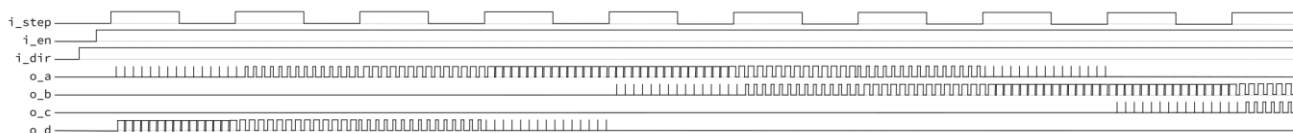
The module can control motors in two modes with various number of microsteps per step. Those modes are selected by the **TWO_PHASES_ACTIVE** parameter. When zero, the module operates in Wave mode activating

only single phase, otherwise Full mode is enabled with both phases activated. Number of microsteps is defined by the MICROSTEPS parameter. Each `i_step` rising edge causes indexer to advance one step in direction set by the `i_dir` port. To enable operation, reset signal should be deasserted and the `i_en` port should be high. The following figure shows module operation for full mode with one microstep.



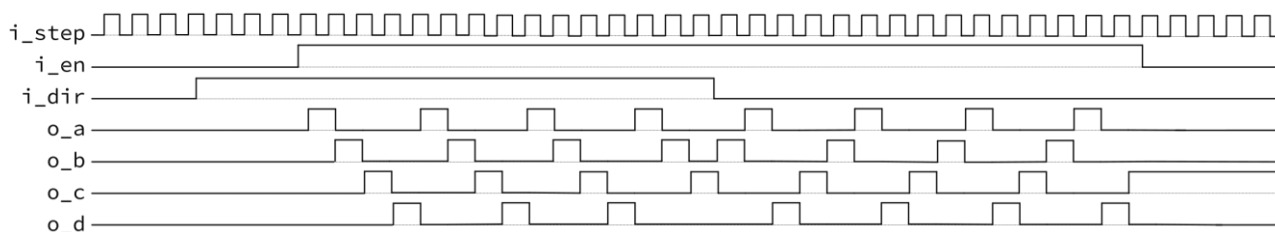
Operation in full mode with one microstep

If the number of microsteps is 4 or higher, the module outputs PWM signal with frequency defined in the PWM_FREQ_HZ parameter. Each `i_step` rising edge causes duty cycle of the outputs to change. The following figure shows module operation for full mode with four microsteps.



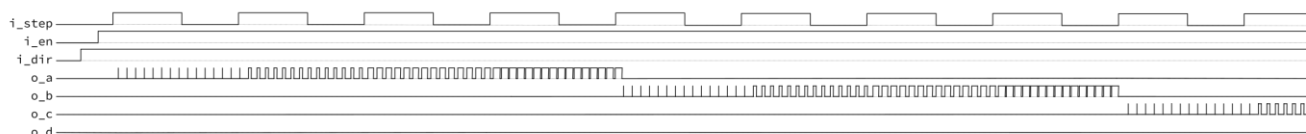
Operation in full mode with four microsteps

If the wave mode selected, only one phase can be active at the time. The following figure shows the module operation in wave mode with one microstep.



Operation in wave mode with one microstep

The following figure shows the module operation in wave mode with 4 microsteps.



Operation in wave mode with four microsteps

4.5 Signal Processing

4.5.1 Breathing Controller

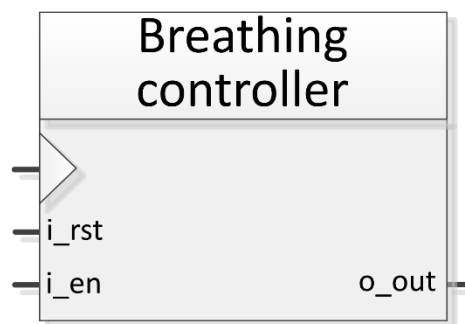
The breathing controller is a module designed to generate a PWM (Pulse Width Modulation) signal with a smooth "breathing" effect. This effect gradually increases and decreases the duty cycle, making it ideal for LED dimming applications where a fade-in/fade-out behavior is required.

Breathing Controller Parameters

- **IN_CLK_HZ** – Specifies the input system clock frequency in Hz. Used to derive internal timing for PWM generation.
- **DEPTH** – Defines the bit width of the PWM counter. This determines the resolution of the duty cycle.
- **PWM_FREQ_HZ** – Specifies the target PWM output frequency in Hz.
- **RAMP_MULT** – Controls the speed of the brightness transition. Higher values result in slower breathing (smoother transitions).

Ports

Port name	Direction	Description
i_clk	input	System clock input
i_rst	input	Active reset input
i_en	input	Enable signal for PWM generation
o_out	output	PWM output signal with breathing effect



Design steps

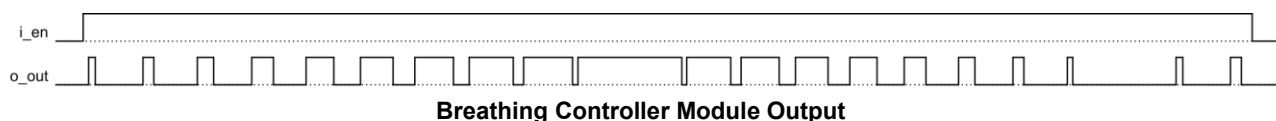
1. Add the **Breathing Controller** to the design by selecting it from the **Module Library**.
2. Instantiate the Breathing Controller in the top-level design file as shown in the following example.
3. Set the configuration parameters according to the application requirements.
4. Connect the controller control signals to the corresponding system logic.
5. Connect the PWM output signal to the LED or controlled device.

Breathing Controller instantiation:

```
breathing_ctrl #(
    .IN_CLK_HZ    (user_param),
    .DEPTH        (user_param),
    .PWM_FREQ_HZ  (user_param),
    .RAMP_MULT    (user_param)
) user_inst_name (
    .i_clk        (user_port_name),
    .i_rst        (user_port_name),
    .i_en         (user_port_name),
    .o_out        (user_port_name)
);
```

Module operation

The breathing_ctrl module generates a PWM signal with a dynamically changing duty cycle to create a smooth breathing effect.



The PWM timing is derived from the system clock (IN_CLK_HZ). An internal divider reduces the clock frequency to achieve the desired PWM frequency (PWM_FREQ_HZ). This is done using a slow counter (r_cnt_slow) that generates an enable tick (r_clk_en) for PWM updates.

A PWM counter (`r_cnt_pwm`) runs at the scaled frequency and defines the current position within the PWM period. The output signal is generated by comparing this counter with a duty cycle value (`r_cnt_duty`).

The breathing effect is implemented using:

- A ramp counter (`r_cnt_ramp`) to slow down duty cycle updates.
- A duty counter (`r_cnt_duty`) to gradually change brightness.
- A direction flag (`r_dir`) to switch between increasing and decreasing brightness.

Operation flow:

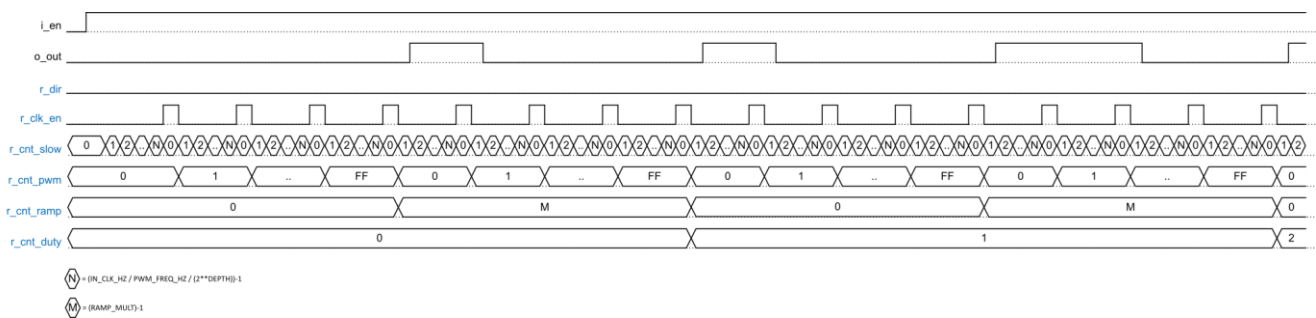
1. The PWM counter continuously cycles through its range.
2. The duty cycle increases gradually from minimum to maximum.
3. Once the maximum is reached, the direction reverses.
4. The duty cycle then decreases back to minimum.
5. This cycle repeats continuously, producing a smooth fade-in/fade-out effect.

The output signal (o_out) is generated as follows:

- When $r_cnt_pwm < r_cnt_duty$, the signal is active.
- The polarity is controlled by the direction signal (r_dir) to ensure symmetrical transitions.

Reset (i_rst) initializes all counters and forces the output LOW.

When enable (i_en) is LOW, the module stops operation and the output remains LOW.



Breathing Controller Module Operation

4.5.2 Frequency Detector

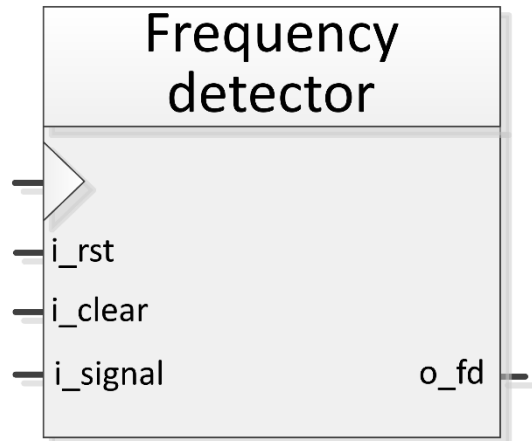
The Frequency Detector is a module designed to measure the frequency of an incoming digital signal and determine whether it falls within a specified range. It produces a binary output indicating whether the detected frequency is valid based on configured minimum and maximum limits.

Frequency Detector Parameters

- **IN_CLK_HZ** – Specifies the input system clock frequency in Hz. Used as a reference for frequency measurement.
- **FREQ_MAX** – Defines the upper bound of the acceptable input signal frequency range.
- **FREQ_MIN** – Defines the lower bound of the acceptable input signal frequency range.
- **OUT_POL** – Controls output polarity. 1'b0 = normal output; 1'b1 = inverted output.

Ports

Port name	Direction	Description
i_clk	input	System clock input
i_rst	input	Active reset input
i_clear	input	Clear signal to reset measurement logic
i_signal	input	Input signal whose frequency is measured
o_fd	output	Frequency detection output signal



Design steps

1. Add the Frequency Detector to the design by selecting it from the Module Library.
2. Instantiate the Frequency Detector in the top-level design file as shown in the following example.
3. Set the configuration parameters according to the application requirements.
4. Connect system signals (i_clk, i_rst, and i_clear) appropriately.
5. Connect the input signal (i_signal) and use the output signal (o_fd) as the detection result.

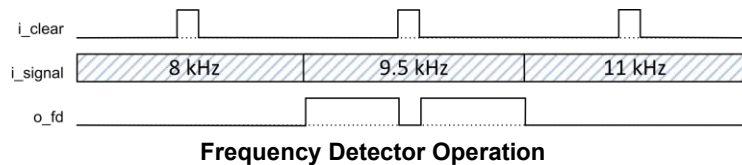
Frequency Detector instantiation:

```
breathing_ctrl #(
    .IN_CLK_HZ      (user_param),
    .DEPTH          (user_param),
    .PWM_FREQ_HZ    (user_param),
    .RAMP_MULT      (user_param)
) user_inst_name (
    .i_clk          (user_name),
    .i_rst          (user_name),
    .i_en           (user_name),
    .o_out          (user_name)
);
```

Module operation

The freq_detector module determines whether the frequency of the input signal i_signal lies within a defined range FREQ_MIN to FREQ_MAX. It achieves this by measuring the number of system clock cycles between consecutive rising edges of the input signal.

Timing diagram for Frequency Detector, at FREQ_MIN = 9000, FREQ_MAX = 10000 and OUT_POL = 0.



Frequency Detector Operation

The acceptable range is converted into clock cycle limits:

- $\text{MIN_VAL} = \text{IN_CLK_HZ} / \text{FREQ_MAX}$: Minimum allowed period.
- $\text{MAX_VAL} = \text{IN_CLK_HZ} / \text{FREQ_MIN}$: Maximum allowed period.

Reset and clear signals:

- *i_rst* – Fully resets internal registers and output
- *i_clear* – Clears current measurement and restarts detection

The input signal *i_signal* is first synchronized to the system clock using a two-stage register (*r_signal_in_state*). This prevents metastability and allows reliable edge detection. The first detected edge initializes the measurement (*r_start*). The module begins valid frequency checking only after this initial synchronization.

A counter (*r_count_data*) increments on each clock cycle after the first detected edge. When a rising edge of the input signal is detected, the counter value represents the signal period and the measured period is compared against the allowed range.

- If $\text{r_count_data} < \text{MIN_VAL}$, frequency is too high, output is inactive.
- If $\text{r_count_data} \geq \text{MIN_VAL}$, frequency is within range, output is active.
- If no edge is detected within MAX_VAL cycles, the signal is considered too slow or missing. Measurement resets and output returns to inactive state.

Output signal *o_fd* reflects frequency validity. Polarity is controlled by *OUT_POL*.

- Normal: active = valid frequency
- Inverted: active = invalid frequency

4.5.3 PWM

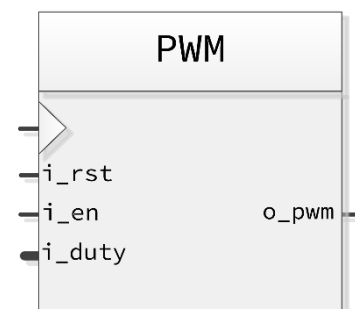
PWM (Pulse-width modulation) is a method of controlling average power or representing an analog value using a digital signal by rapidly switching it on and off and varying the duty cycle of the resulting rectangular waveform. PWM is widely used in applications such as motor speed control, LED brightness adjustment, power supplies, and audio signal generation.

PWM Parameters

- WIDTH – Width of PWM counter.

Ports

Port name	Direction	Description
i_clk	input	System clock input
i_rst	input	Active high reset input
i_duty	input	Input bus for duty cycle selection
o_pwm	output	Output bus for PWM signal



Design steps

1. Add the **PWM** to the design by selecting it from the **Module Library**.
2. Instantiate PWM in the top-level design file as shown in the following example.
3. Set the configuration parameters according to the application requirements.
4. Connect the control signals to the corresponding system logic.
5. Connect the external port to the corresponding system logic or external blocks.

PWM instantiation:

```
pwm #(
    .WIDTH ( user_param ),
) user_inst_name (
    .i_clk      ( user_name ),
    .i_rst      ( user_name ),
    .i_duty     ( user_name ),
    .o_pwm      ( user_name )
);
```

Module operation

To enable operation, i_rst should be deasserted. When i_en is set high, the module operation starts immediately, and the PWM signal appears on the o_pwm port.

The i_duty bus defines the duty cycle for the counter. This port has a width specified by the WIDTH parameter, which also determines the PWM bit depth. The value of i_duty can vary from 0 to $2^{\text{WIDTH}} - 1$. For example, to represent a 50% PWM duty cycle, i_duty should be set to $(2^{\text{WIDTH}} - 1)/2$.

The duty cycle value can be changed during operation; however, the first cycle after the change may be invalid.



PWM Output

4.5.4 Quadrature Decoder

The Quadrature Decoder module converts two phase-shifted input signals (A and B) into position and direction information.

Quadrature signals are digital signals with a 90° phase difference, typically generated by rotary or linear encoders. By analyzing the phase relationship between these signals, the module determines both movement direction and displacement.

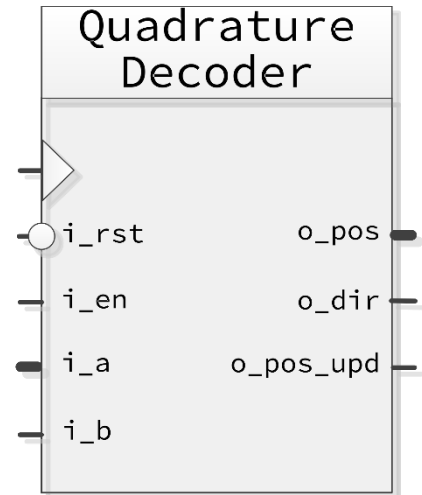
The module supports configurable decoding resolution (x1, x2, and x4) and provides position, direction, and position update indication outputs.

Quadrature Decoder Parameters

- **WIDTH** – Position counter bit width (Minimum: 2, Maximum: 32).
- **DECODE_MODE** – Quadrature decoding resolution: 1 – x1 (rising edge of A only), 2 – x2 (both edges of A), 4 – x4 (all valid A/B transitions).

Ports

Port name	Direction	Description
i_clk	input	System clock input
i_rst	input	Active-high synchronous reset input
i_en	input	Enable signal (gates counting)
i_a	input	Quadrature channel A
i_b	input	Quadrature channel B
o_pos	output	Position counter
o_dir	output	Direction (1 = forward, 0 = reverse)
o_pos_upd	output	Position update strobe (1-cycle pulse)



Design steps

1. Add the **Quadrature Decoder** module to the design by selecting it from the **Module Library**.
2. Instantiate the Quadrature Decoder in the design as shown in the following example.
3. Set the configuration parameter according to the application requirements.
4. Connect the Quadrature Decoder input/output signals to the corresponding system logic.

Quadrature Decoder module instantiation:

```
quadrature_decoder #(
    .WIDTH ( user_param )
) user_inst_name (
    .i_clk      ( user_name ),
    .i_rst      ( user_name ),
    .i_en       ( user_name ),
    .i_a        ( user_name ),
    .i_b        ( user_name ),
    .o_pos      ( user_name ),
    .o_dir      ( user_name ),
    .o_pos_upd  ( user_name )
);
```

Module operation

The Quadrature Decoder module tracks position and direction based on transitions of input signals i_a and i_b. The decoding resolution is selected via the DECODE_MODE parameter. Input signals are sampled using a multi-stage shift register to ensure stable edge detection and reduce metastability effects. The module generates an internal counting enable signal (w_count_en) depending on the selected resolution:

- x1 mode – Counts only on rising edge of i_a with valid i_b phase.

- x2 mode – Counts on both edges of i_a with valid i_b phase.
- x4 mode – Counts on every valid transition of i_a or i_b.
- Direction is determined from the relative phase of signals i_a and i_b:
- 1 – Forward rotation
- 0 – Reverse rotation

The direction is updated only when a valid transition occurs.

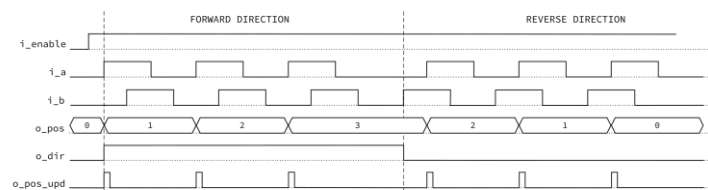
When i_en is asserted and a valid transition is detected:

- The position counter o_pos is incremented or decremented based on the direction.
- The o_dir output is updated.
- The o_pos_upd signal produces a single-cycle pulse.

Position counter operates continuously while enabled. On overflow/underflow, natural wrap-around occurs (based on WIDTH bit size)

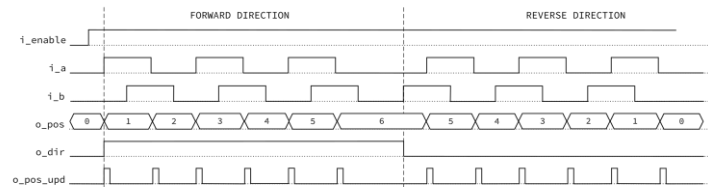
Quadrature Decoder timing diagrams:

x1 decoding mode:



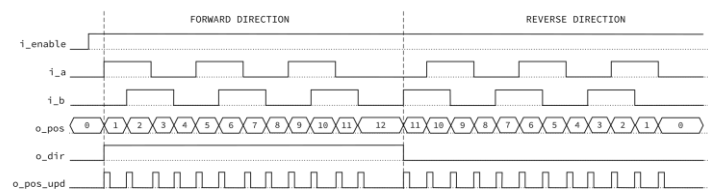
Quadrature Decoder x1 decoding mode

x2 decoding mode:



Quadrature Decoder x2 decoding mode

x4 decoding mode:



Quadrature Decoder x4 decoding mode

Notes:

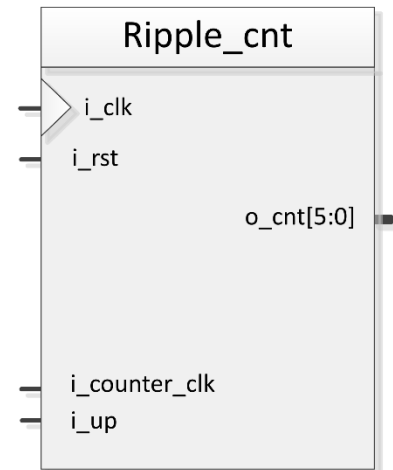
- External debouncing of i_a and i_b is recommended for mechanical encoders.
- DECODE_MODE = 4 provides the highest resolution but is most sensitive to input noise.

4.5.5 Ripple Counter

The ripple counter is created by cascading flip-flops, where the output of one flip-flop drives the clock of the next.

Ripple Counter Parameters

- **SET_VAL** – Set ripple counter value (Type: Decimal; Default Value: *0, Min value: 0, Max Value = 30*).
- **END_VAL** – End ripple counter value (Type: Decimal; Default Value: *31, Min Value: 1, Max Value: 31*).
- **FULL_RANGE** – When zero, count range is between SET_VAL and END_VAL values, otherwise full count range (Type: Boolean; Default Value: *1'b0, Min Value: 1'b0, Max Value: 1'b1*).



Ports

Port name	Direction	Description
i_clk	input	Input clock domain signal
i_rst	input	Input reset input
i_counter_clk	input	Input counter clock signal
i_up	input	Input direction count signal
o_cnt	output	Output ripple counter signal
o_cnt_oe	output	Output enable for ripple counter signal

Design Steps

1. Add the **ripple_cnt** to the design by selecting it from the **Module Library**.
2. Instantiate the ripple_cnt in the top-level design file as shown in the following example.
3. Set the configuration parameters according to the application requirements.
4. Connect the control signals to the corresponding system logic.

Ripple Counter Instantiation:

```
ripple_cnt #(
    .SET_VAL          ( user_param ),
    .END_VAL          ( user_param ),
    .FULL_RANGE       ( user_param )
user_inst_name (
    .i_clk            ( user_name ),
    .i_rst            ( user_name ),
    .i_counter_clk    ( user_name ),
    .i_up             ( user_name ),
    .o_cnt             ( user_name ),
    .o_cnt_oe         ( user_name )
);
```

Module Operation

This design is a parameterized 5-bit counter that operates in the *i_clk* domain and changes its count value only when a rising edge is detected on the input signal *i_counter_clk*. The key signals in the module are *i_clk*, which is the main clock for all sequential logic; *i_rst*, a synchronous reset that initializes the counter; *i_counter_clk*, which acts as the event signal that tells the counter when to update; *i_up*, which controls the direction of counting; and *o_cnt*, the 5-bit counter output.

The behavior of the counter is controlled by three parameters: **SET_VAL**, which defines the initial/reset value and, in limited-range mode, the lower bound of the counter; **END_VAL**, which defines the upper limit of the counter; and **FULL_RANGE**, which selects whether the counter wraps between **SET_VAL** and **END_VAL** or between 0 and **END_VAL**. Because *o_cnt* is 5 bits wide, the valid counting range is up to 31.

Functionally, the design first detects edges on `i_counter_clk` rather than using it directly as a clock. To do this, the register `r_counter_clk_old` stores the previous sampled value of `i_counter_clk` on every rising edge of `i_clk`. That previous value is then concatenated with the current `i_counter_clk` value to form `w_counter_clk_state`. When `w_counter_clk_state` becomes `2'b01`, the module knows that `i_counter_clk` has made a low-to-high transition, so this is treated as a count event. In the main counter logic, if reset is asserted, `o_cnt` is loaded with `SET_VAL`. Otherwise, on each detected rising edge of `i_counter_clk`, the counter either increments or decrements depending on `i_up`.

If `FULL_RANGE` is 0, the counter wraps within the user-defined range: when counting up, reaching `END_VAL` causes it to jump back to `SET_VAL`, and when counting down, reaching `SET_VAL` causes it to wrap back to `END_VAL`. If `FULL_RANGE` is 1, the counter instead wraps between 0 and `END_VAL`: counting up rolls over from `END_VAL` to 0 and counting down rolls under from 0 back to `END_VAL`. Overall, this design provides a flexible, programmable up/down counter with selectable wrap limits and an internal edge-detection scheme for the count trigger input.

5. Applications

[Design file](#)

5.1 SPI to 4-channel UART Converter

Overview

This application demonstrates an SPI-controlled multi-channel UART bridge implemented in ForgeFPGA. The design enables a host MCU or processor to communicate with up to four independent UART interfaces through a single SPI connection.

The solution is suitable for systems requiring UART expansion with minimal pin usage, centralized control, or protocol bridging between SPI-based masters and multiple serial peripherals.

Target Devices

- SLG47920
- SLG47921

Key Features

- SPI slave interface for host communication with up to 25 MHz clock speed.
- Four independent UART channels (TX/RX per channel).
- Configurable baud rate for each UART channel. Supported baud rate range: from 4800 to 115200 bits/s.
- Full-duplex communication support.
- FIFO-based buffering for reliable data transfer. Each UART transmitter and receiver has its own synchronous FIFO (8-sync. FIFOs in total) with size 64 x 8 bit. Supported FIFO's depth range: 4 - 64 bytes.

Design Overview

Architecture

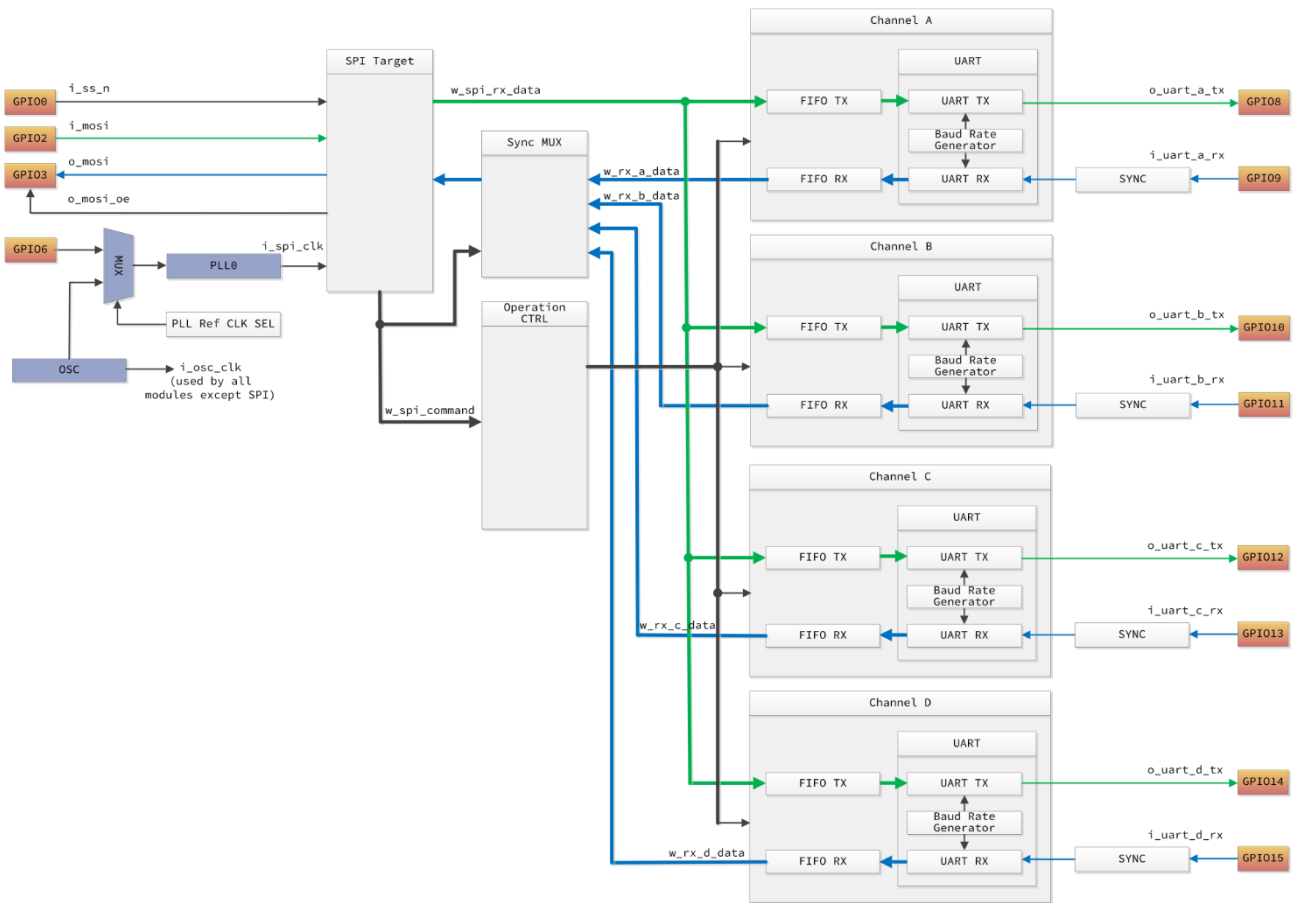
The SPI to 4-channel UART converter is implemented using a combination of FPGA soft logic and dedicated hardware resources available in the SLG47921 device. The design acts as a bridge between a single SPI interface and four independent UART channels, allowing a host controller to communicate with multiple UART devices through a unified interface.

The design architecture consists of:

- SPI target interface for communication with the external controller
- Control logic for command decoding and channel selection
- Four independent UART channels (A–D)
- FIFO-based buffering between SPI and UART domains
- Clock generation and synchronization logic

The design structure diagram does not illustrate all internal blocks. Just necessary blocks to describe the operation flow.

Design Structure



SPI to 4-channel UART converter structure

The implementation uses both Verilog-based soft modules and integrated hard blocks such as oscillator, PLL, power-on reset, and BRAM for FIFO storage.

Soft-code Blocks

Implemented in Verilog using FPGA core resources. Include: SPI logic, UART modules, FIFOs, and control logic.

Hard Blocks

Integrated physical resources connected to FPGA core. Used to support clocking, reset, and memory.

Key hardware resources used:

- POR (Power-On Reset) – Global reset for most logic.
- OSC (Oscillator) – Provides 50 MHz system clock.
- PLL – Generates SPI clock domain signal (bypass mode).
- BRAM (0...7) – Used to implement FIFO buffers for data storage.

UART Channel Structure

The design implements four independent UART channels:

Each channel contains:

- UART transmitter
- UART receiver
- TX FIFO

▪ RX FIFO

Each FIFO is implemented using embedded BRAM. Data buffering ensures reliable communication between SPI and UART interfaces with different data rates.

SPI Interface and Control Logic

The SPI module operates as a target and is responsible for:

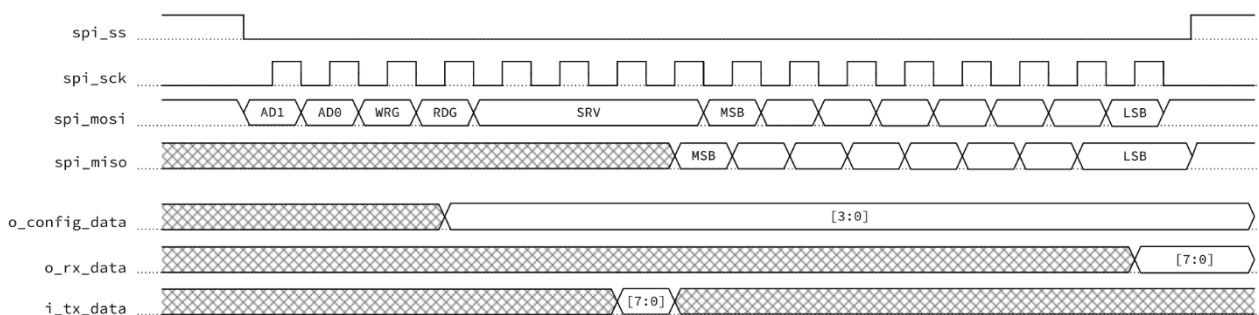
- Receiving command and data bytes from the controller
- Returning data from selected UART channels
- Decoding command fields and routing transactions

The SPI protocol configuration:

- Uses CPOL = 0 and CPHA = 0
- Transfers data MSB-first
- Starts transaction when chip select is asserted
- Supports up to 25 MHz clock frequency

A command byte defines:

- Target UART channel (A–D)
- Direction of data transfer (write, read, or full duplex)
- Synchronization/control bits for internal operation



SPI Timing Diagram

This diagram illustrates the SPI transaction format, including command byte fields (AD1, AD0, WRG, RDG, and SRV) and the data transfer sequence between the controller and converter.

The command byte includes:

- AD1, AD0 – 2-bit UART address: "00" – Channel A, "01" – Channel B, "10" – Channel C and "11" – Channel D.
- WRG, RDG – 2-bit data direction: "10" – Only transmit data from SPI to UART, "01" – Only receive data from UART to SPI, "11" – Full duplex communication, and "00" – Reserved combination.
- SRV – Synchronization bits for internal operation

Data Path and Multiplexing

- SPI → UART: Data is written directly to the selected channel TX FIFO. A 1-to-4 demultiplexer inside Operation CTRL module is used to generate channel-select control signals (write enables).
- UART → SPI: RX FIFO outputs from all channels are routed through a 4-to-1 multiplexer, which selects the active channel data for SPI readback.

Channel selection: Controlled by address bits (AD1, AD0) from the SPI command byte, which are updated during each SPI transaction

Clock Domains and Synchronization

The design operates with two clock domains:

- SPI clock domain. Driven by external controller via SPI clock.
- System clock domain. Generated internally by the 50 MHz oscillator.

Synchronization between domains is achieved using dedicated synchronization modules to ensure safe data transfer.

Operation Flow

1. The SPI controller issues a command selecting a UART channel and transfer type.
2. Data is written to or read from the corresponding FIFO.
3. UART modules independently transmit or receive serial data.
4. Buffered data is exchanged between domains without loss.
5. This architecture enables full-duplex or directional communication between SPI host and multiple UART devices.

Summary

This SPI-to-4-channel UART converter provides a compact solution for bridging SPI Controllers with multiple UART devices. It is particularly useful in embedded and industrial applications where pin count, flexibility, and protocol interoperability are critical.

5.2 SerDes via LVDS (SDR)

Overview

This application demonstrates a point-to-point communication link between two SLG47921 devices using an LVDS interface operating in SDR (Single Data Rate) mode. One device functions as a serializer, converting parallel data into a high-speed LVDS serial stream, while the second device functions as a deserializer, recovering the transmitted data and reconstructing the original parallel word. The design enables reliable communication while minimizing the number of required interconnect signals. It can be used for board-to-board communication, sensor interfaces, custom serial links, and FPGA expansion applications.

Target Devices

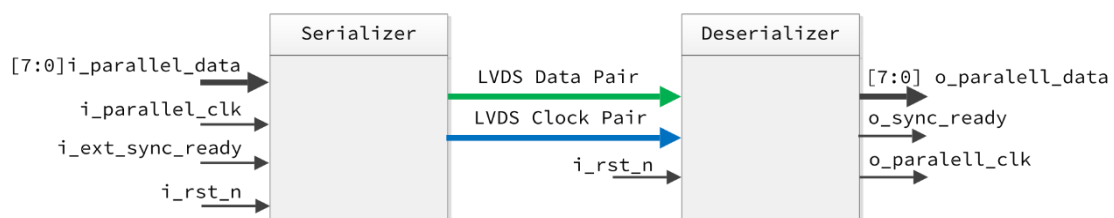
- SLG47912
- SLG47920
- SLG47921

Key Features

- LVDS differential interface requiring only two LVDS pairs
- SDR data transmission with a 50 Mbps data rate
- Parallel-to-serial and serial-to-parallel conversion with an 8:1 serialization factor
- Source-synchronous clock forwarding
- Automatic word synchronization using a synchronization frame word
- Optional serializer-to-deserializer synchronization handshake

Designs Overview

Block Diagram

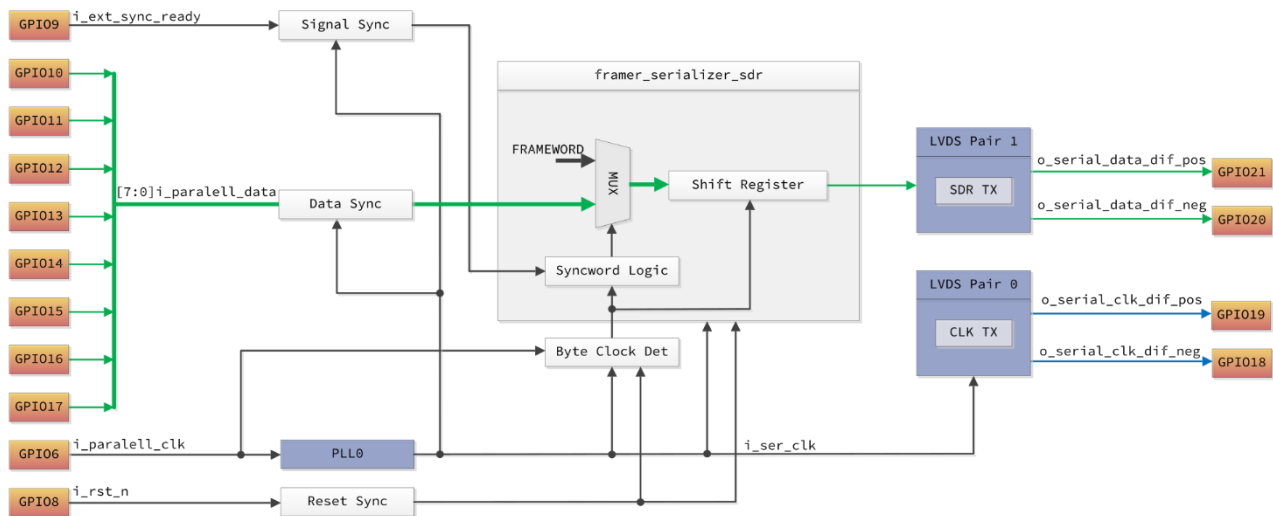


SerDes via LVDS (SDR) block diagram

The system consists of two SLG47921 devices connected through two LVDS differential pairs. One pair transmits SDR data, while the second pair forward the serializer clock to the deserializer.

The serializer converts 8-bit parallel data into a SDR serial stream and transmits it over the LVDS data pair. The deserializer captures the incoming serial data using the forwarded clock, reconstructs the original parallel data, and generates a recovered parallel clock.

Serializer design structure



Serializer Internal Structure

Design Operation

The serializer accepts 8-bit parallel data and converts it into a serial stream for transmission over the LVDS interface. Parallel data is loaded at the byte-clock rate, while serialization is performed using a higher-frequency serializer clock.

A byte-clock edge detector generates a load pulse on the rising edge of `i_parallel_clk`, allowing new parallel data to be loaded into the serializer. The input byte clock operates at 6.25 MHz.

PLL0 uses the input byte clock as its reference clock and generates the 50 MHz serializer clock (`i_ser_clk`). This clock drives all serializer logic and is forwarded to the deserializer through LVDS Pair 0.

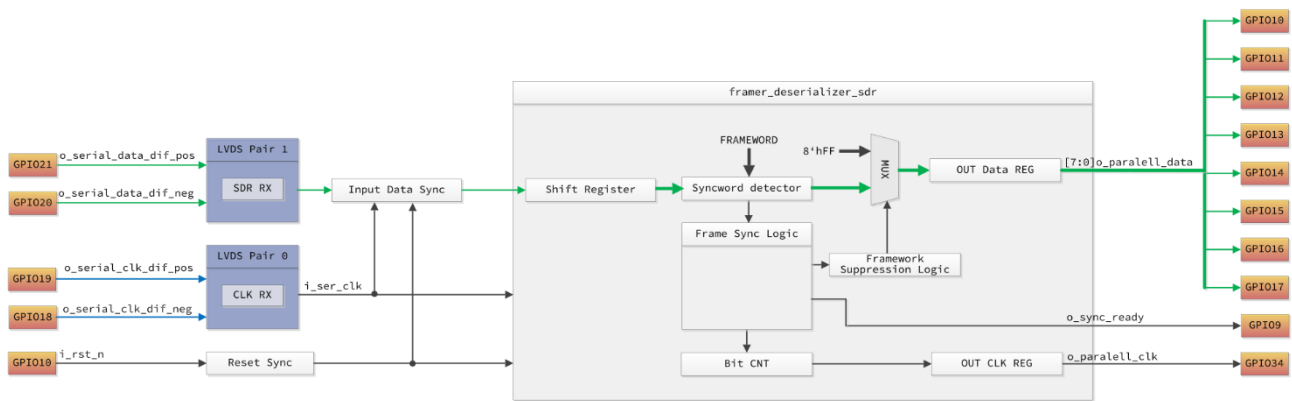
PLL0 requires approximately 600–900 input clock cycles to achieve lock and generate a stable 50 MHz clock. Therefore, the byte clock must be present before communication begins.

The `i_rst_n` input is used to enable the serializer. After reset is released, the serializer initially transmits a predefined synchronization frame word (0x7E). A synchronization timer controls the number of frame words transmitted before normal user data is selected. Once the timer reaches the programmed threshold, the serializer begins transmitting the incoming parallel data stream. Optionally, the `i_ext_sync_ready` signal can be used to delay user-data transmission until the deserializer has successfully synchronized with the serializer. This feature is enabled by setting `WAIT_SYNC_READY = 1`.

The selected byte is loaded into an 8-bit shift register. During serialization, the shift registers shift once per serializer clock cycle. The least-significant bit is presented on `o_serial_sdr_data` and transmitted through LVDS Pair 1.

This architecture provides a simple and efficient parallel-to-serial conversion mechanism for reliable communication between SLG47921 devices.

Deserializer design structure



Deserializer Internal Structure

Design Operation

The deserializer receives a serial stream from the LVDS receiver and reconstructs the original 8-bit parallel data transmitted by the serializer. Incoming serial data is synchronized to the local serial clock and shifted into an internal 8-bit shift register.

The 50 MHz clock received through LVDS Pair 0 is used as the internal clock for the deserializer logic. The serialized data stream is received through LVDS Pair 1.

The `i_rst_n` input is used to enable the deserializer. Following reset, the receiver continuously monitors the incoming serial stream for the predefined synchronization frame word (0x7E). A synchronization counter tracks consecutive frame-word detections. Synchronization is considered valid when the number of detected frame words reaches the `FRAME_SYNC_THRES` threshold. Once synchronization is established, the `o_sync_ready` signal is asserted.

The received serial data is accumulated in the shift register and reconstructed into an 8-bit parallel word. An internal bit counter tracks the byte boundary and controls generation of the recovered parallel clock.

During startup and synchronization acquisition the `o_parallel_data` is forced to 8'hFF and the recovered clock remains inactive.

Optionally, leading synchronization frame words can be suppressed after synchronization to prevent them from appearing on the parallel interface. This feature is enabled by setting `ESCAPE_LEAD_FRAMEWORDS = 1`. Once the first non-frame word byte is received (a value different from 0x7E), normal operation begins and valid data appears on `o_parallel_data` together with the recovered byte clock (`o_parallel_clk`).

This architecture provides automatic byte alignment and synchronization recovery using only the received serial data stream and forwarded clock.

Summary

This article demonstrates a Serializer/Deserializer (SerDes) communication link implemented using two SLG47921 ForgeFPGA devices connected through an LVDS SDR interface. The serializer converts 8-bit parallel data into a high-speed SDR serial stream, while the deserializer reconstructs the original data and automatically establishes byte alignment using a synchronization frame word.

The design provides a compact, low-pin-count solution for reliable high-speed communication between ForgeFPGA devices.

5.3 SerDes via LVDS (DDR)

[Design file](#)

Overview

This application demonstrates a high-speed point-to-point communication link between two SLG47921 devices using an LVDS interface operating in DDR (Double Data Rate) mode. One device functions as a serializer (SerDes TX), converting parallel data into a high-speed LVDS serial stream, while the second device functions as a deserializer (SerDes RX), recovering the transmitted data and reconstructing the original parallel word.

The design enables efficient high-throughput communication while minimizing the number of required interconnect signals. It can be used for board-to-board communication, video and sensor interfaces, custom high-speed links, and FPGA expansion applications.

Target Devices

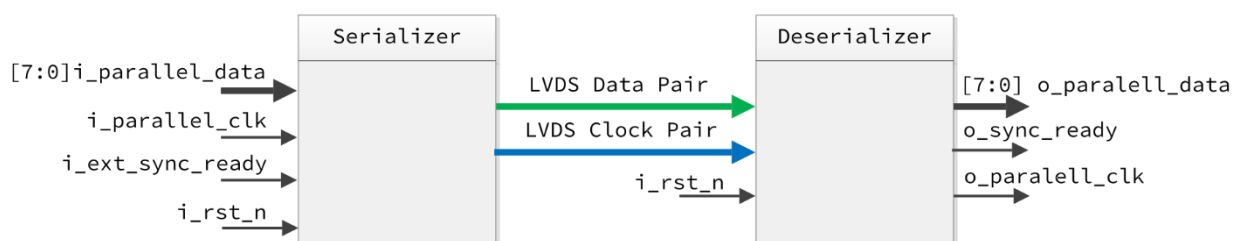
- SLG47912
- SLG47920
- SLG47921

Key Features

- LVDS differential interface requiring only two LVDS pairs
- DDR data transmission with a 100 Mbps data rate
- Parallel-to-serial and serial-to-parallel conversion with an 8:1 serialization factor
- Source-synchronous clock forwarding
- Automatic word synchronization using a synchronization frame word
- Optional serializer-to-deserializer synchronization handshake

Designs Overview

Block Diagram



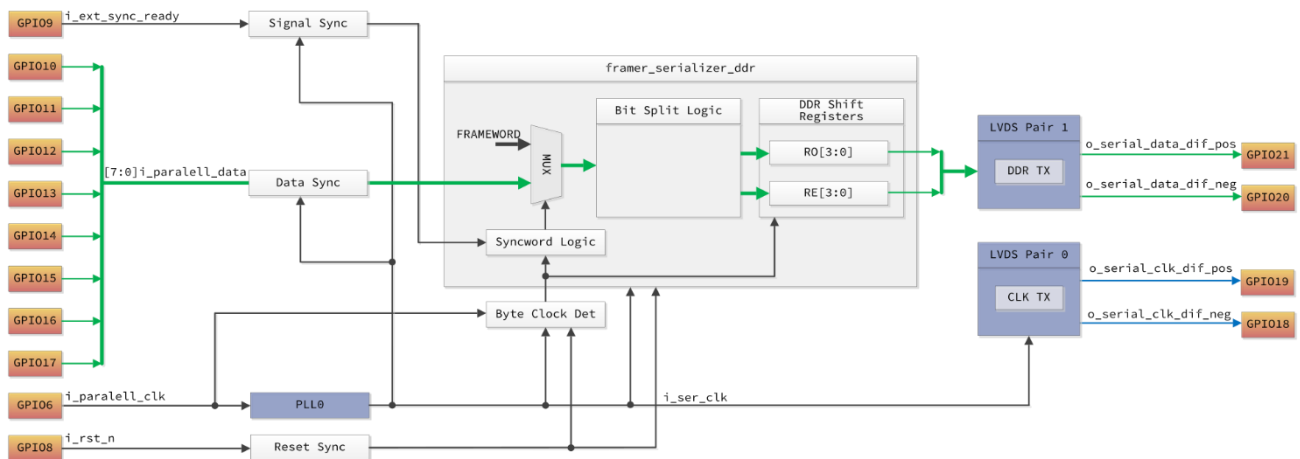
SerDes via LVDS (DDR) block diagram

The system consists of two SLG47921 devices connected through two LVDS differential pairs. One pair transmits DDR data, while the second pair forwards the serializer clock to the deserializer.

The serializer converts 8-bit parallel data into a DDR serial stream and transmits it over the LVDS data pair. The deserializer captures the incoming serial data using the forwarded clock, reconstructs the original parallel data, and generates a recovered parallel clock.

Serializer

Design structure



Serializer Internal Structure

Design Operation

The serializer accepts 8-bit parallel data and converts it into a two-bit DDR serial stream for transmission over the LVDS interface. Parallel data is loaded at the byte-clock rate, while serialization is performed using a higher-frequency serializer clock.

A byte-clock edge detector generates a load pulse on the rising edge of `i_parallel_clk`, allowing new parallel data to be loaded into the serializer.

PLL0 uses the 12.5 MHz byte clock as its reference clock and generates the 50 MHz serializer clock (`i_ser_clk`). This clock drives all serializer logic and is forwarded to the deserializer through LVDS Pair 0.

PLL0 requires approximately 600–900 input clock cycles to achieve lock and generate a stable 50 MHz clock. Therefore, the byte clock must be present before communication begins.

The `i_rst_n` input is used to enable the serializer. After reset is released, the serializer initially transmits a predefined synchronization frame word (0x7E). A synchronization timer controls how many frame words are transmitted before normal user data is selected.

Optionally, the `i_ext_sync_ready` signal can be used to delay user-data transmission until the deserializer has successfully synchronized with the serializer. This feature is enabled by setting `WAIT_SYNC_READY = 1`.

The selected byte is divided into odd and even bit groups and loaded into two independent shift registers:

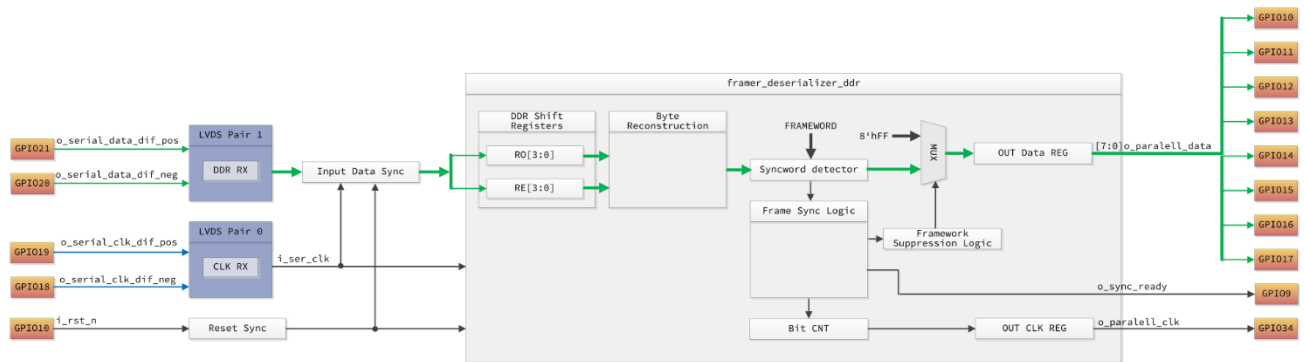
- Odd bits: [7], [5], [3], [1]
- Even bits: [6], [4], [2], [0]

During serialization, both shift registers shift once per serializer clock cycle. The least significant bits of the odd and even shift registers are presented on `o_serial_dds_data`, allowing the LVDS transmitter to send one bit on each DDR phase. The serialized data stream is transmitted through LVDS Pair 1. This arrangement reconstructs the original byte sequence at the receiver while doubling the effective data rate compared to SDR transmission.

The serializer can operate either with timer-based synchronization or with optional synchronization feedback from the deserializer. This architecture provides a simple and efficient parallel-to-serial conversion mechanism for reliable high-speed communication between SLG47921 devices.

Deserializer

Design structure



Deserializer Internal Structure

Design Operation

The deserializer receives a two-bit DDR serial stream from the LVDS receiver and reconstructs the original 8-bit parallel data transmitted by the serializer. Incoming DDR data is synchronized to the local serial clock and captured by two independent shift registers corresponding to the odd and even data paths.

The 50 MHz clock received through LVDS Pair 0 is used as the internal clock for the deserializer logic. The serialized data stream is received through LVDS Pair 1.

The `i_rst_n` input is used to enable the deserializer.

To establish byte alignment, the deserializer continuously monitors the reconstructed data stream for the predefined synchronization frame word (0x7E). A synchronization counter tracks consecutive frame-word detections. Synchronization is considered valid when the number of detected frame words reaches the `FRAME_SYNC_THRES` threshold. Once synchronization is achieved, the `o_sync_ready` signal is asserted.

The received DDR data is reassembled by interleaving bits from the odd and even shift registers, recreating the original 8-bit parallel word. An internal bit counter tracks the byte boundary and generates the recovered parallel clock.

During startup and synchronization acquisition the `o_parallel_data` is forced to 8'hFF and the recovered clock remains inactive.

Optionally, leading synchronization frame words can be suppressed after synchronization to prevent them from appearing on the parallel interface. This feature is enabled by setting `ESCAPE_LEAD_FRAMEWORDS = 1`. Once the first non-frame word byte is received (a value different from 0x7E), normal operation begins and valid data appears on `o_parallel_data` together with the recovered byte clock (`o_parallel_clk`).

This architecture provides automatic byte alignment and synchronization recovery, enabling reliable reconstruction of the transmitted parallel data stream using only the received DDR serial data and clock.

Summary

This article demonstrates a Serializer/Deserializer (SerDes) communication link implemented using two SLG47921 ForgeFPGA devices connected through an LVDS DDR interface. The serializer converts 8-bit parallel data into a high-speed DDR serial stream, while the deserializer reconstructs the original data and automatically establishes byte alignment using a synchronization frame word.

The design provides a compact, low-pin-count solution for reliable high-speed communication between ForgeFPGA devices.

6. Revision History

Revision	Date	Description
1.00	Oct 6, 2026	Initial release.