# Sequential Logic Design

Lecture 8

# Subprograms

Subprograms are used to improve the readability and to exploit code reusability. They can be defined using:

- Tasks
- Functions

Additional Resources: http://rise.cse.iitm.ac.in/people/faculty/kama/prof/TasksnFunctions.pdf

# Tasks

- More general than functions and can be used to model both combinational and sequential logic.
- May contain timing controls, i.e. timing delays, like posedge, negedge, # delay, wait
- Can call other tasks and functions
- Can have zero, one, or more arguments.
- Values are passed to and from a task through arguments.
- Can use global variables, when no local variables are used. When local variables are used then output is assigned only at the end of task execution.
- Can have any number of inputs and outputs, the arguments can be input, output, or inout
- The variables declared within the task are local to that task. The order of declaration within the task defines how the variables passed to the task are used when called
- It must be specifically called with a statement, unlike a function which can be used within an expression
- Defined in the module in which they are used. It is possible to define a task in a separate file and use the compile directive 'include to include the task in the file which instantiates the task.

# Task

```
task task_id;
[declarations]
procedure statements
endtask
```

## Example

```
module HAS_TASK;

    parameter MAXBITS = 8;
    reg [MAXBITS – 1:0] REG_X, NEW_REG;

    task REVERSE_BITS; // task definition starts here

        input [MAXBITS – 1 : 0] DIN;
        output [MAXBITS – 1 : 0] DOUT;
        integer k;

        begin
            for (k=0; k < MAXBITS; k = k +1)
                DOUT[MAXBITS-k] = DIN[k];
        end
    endtask             // task definition ends here

    always@ (REG_X)
        REVERSE_BITS(REG_X,NEW_REG); // task being called

endmodule
```

# System Tasks

Note: system tasks are not synthesizable

`$display –` Print immediate values to standard output with an end-of-line character.

$write – Similar to the display task except it does not print an end-of-line character.

$monitor – Monitors the argument continuously. Whenever there is a change of value in an argument list, the entire argument list is displayed.

$strobe - Print the values at the end of the current timestep

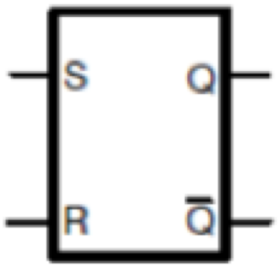
# Functions

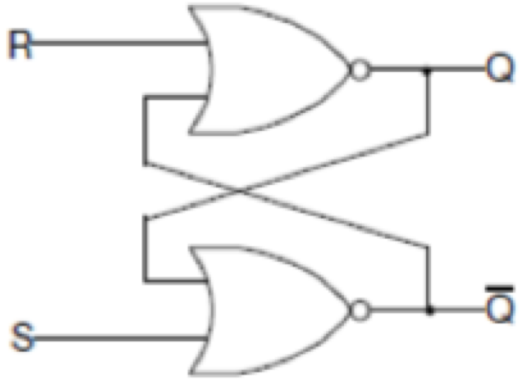
- Used to describe combinatorial logic
- Functions cannot call tasks; however, it can call other functions
- Cannot drive more than one output but can have any number of inputs
- The variables declared within the function are local and their order of declaration dictates how the variables passed to the function are used.
- Cannot use delay, timing, or event control constructs, i.e. posedge, negedge, # delay
- Defined in the module in which they are used. It is possible to define functions in separate files and use compile directive 'include to include the function in the file which instantiates the task.

# Functions

```
module HAS_FUNCTION(X_IN, REV_X);

   parameter MAXBITS = 8;
   input [MAXBITS – 1 : 0] X_IN;
   output reg [MAXBITS – 1 : 0] REV_X;

   function [MAXBITS – 1 : 0] REVERSE_BITS;//function definition starts here

        input [MAXBITS – 1 : 0] DIN;
        integer k;

      begin
          for (k=0; k < MAXBITS; k = k +1)
                REVERSE_BITS[MAXBITS-k] = DIN[k];
      end
   endfunction          //function definition ends here

   always@ (X_IN)
      REV_X = REVERSE_BITS(X_IN); // function being called
                                  // They are used in the right hand side of an assignment statement

endmodule
```

# Latches




| S | R | Q | $\overline{Q}$ |
|---|---|---|---|
| 0 | 0 | Latch | |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | Metastable | |

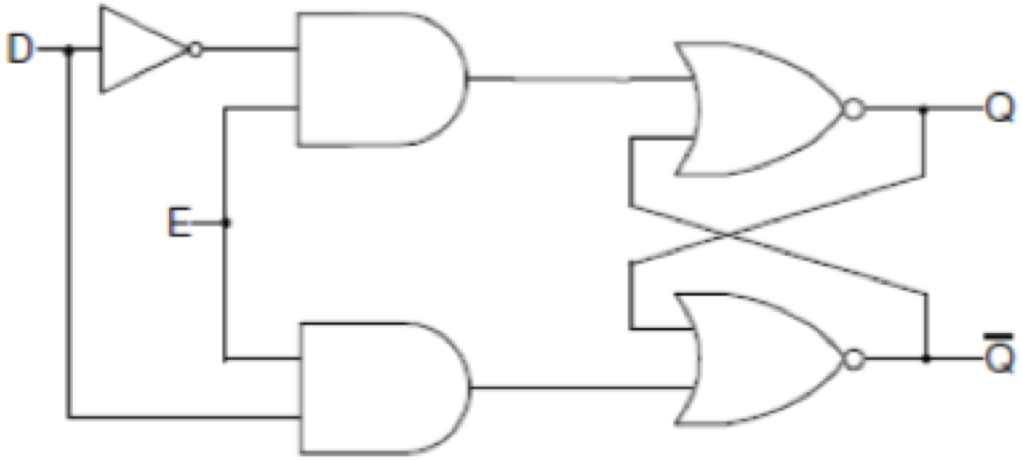
Gate-level

```
module SR_latch_gate (input R, input S, output Q, output Qbar);
nor (Q, R, Qbar);
nor (Qbar, S, Q);
endmodule
```

Dataflow

```
module SR_latch_dataflow (input R, input S, output Q, output Qbar);
assign #2 Q_i = Q;
assign #2 Qbar_i = Qbar;
assign #2 Q = ~ (R | Qbar);
assign #2 Qbar = ~ (S | Q);
endmodule
```

# D-latches can be modeled in behavioral modeling




| Enable | D | Q | $\overline{Q}$ |
|---|---|---|---|
| 0 | 0 | Latch | |
| 0 | 1 | Latch | |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 |

```
module D_latch_behavior (input D, input Enable, output Q, output Qbar);

always @ (D or Enable)
    if(Enable)
    begin
        Q <= D;
        Qbar <= ~D;
   end
endmodule
```

Note that since we do not specify what to do when Enable is low, the circuit “remembers” the previous state.

# Behavioral modeling for positive edge triggered D flip-flop

```
module D_ff_behavior (input D, input Clk, output reg Q);
always @ (posedge Clk)
    if(Clk)
    begin
       Q <= D;
   end
endmodule
```

## D flip-flop with Synchronous Reset

```
module D_ff_with_synch_reset_behavior(input D, input Clk, input reset, output
reg Q);
   always @(posedge Clk)
      if (reset)
      begin
         Q <= 1'b0;
      end else
      begin
         Q <= D;
      end
endmodule
```

## D flip-flop with Asynchronous Reset

```
module D_ff_with_asynch_reset_behavior(input D, input Clk, input clear,
output reg Q);
   always @(posedge Clk or posedge clear)
      if (clear)
      begin
         Q <= 1'b0;
      end else
      begin
         Q <= D;
      end
endmodule
```

# T Flip-Flop (Toggle)

Useful for constructing: binary counters, clock dividers

Example:
T flip-flop that is sensitive to a falling edge of clock and has active-low reset and active-high T control signals.

```
module T_ff_enable_behavior(input Clk, input reset_n, input T, output reg Q);
   always @(negedge Clk)
      if (!reset_n)
         Q <= 1'b0;
      else if (T)
        Q <= ~Q;
endmodule
```

# Registers

Stores bits of information in such a way that systems can write to or read out all the bits simultaneously.

Examples – Parallel Registers :
Simple register has separate data input and output pins but clocked with the same clock source.

```
module Register (input [3:0] D, input Clk, output reg [3:0] Q);
   always @(posedge Clk)
         Q <= D;
endmodule
```

Note: The information is registered every clock cycle

## Controlled register clocking with an enable pin

```
module Register_with_synch_load_behavior(input [3:0] D, input Clk, input
load, output reg [3:0] Q);
   always @(posedge Clk)
      if (load)
         Q <= D;
endmodule
```

## Controlled register clocking with a synchronous reset and enable

```
module Register_with_synch_reset_load_behavior(input [3:0] D, input Clk,
input reset, input load, output reg [3:0] Q);
   always @(posedge Clk)
      if (reset)
      begin
         Q <= 4'b0;
      end else if (load)
      begin
        Q <= D;
      end
endmodule
```

Note: Reset has higher priority over load

# Shift Register

Simple one- bit serial shift in and shift out register

```
module simple_one_bit_serial_shift_register_behavior(input Clk, input
ShiftIn, output ShiftOut);
    reg [31:0] shift_reg;

    always @(posedge Clk)
          shift_reg <= {shift_reg[30:0], ShiftIn};
    assign ShiftOut = shift_reg[31];

endmodule
```

Shifts for 32 clock cycles implementing a delay line

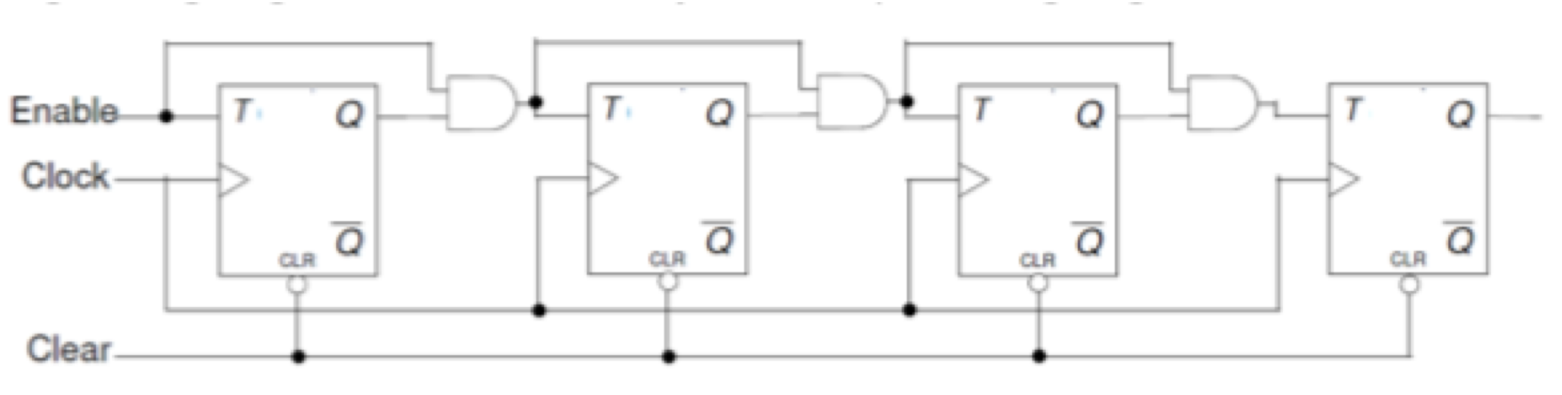
The 31 least significant bits are shifted left and ShiftIn is concatenated as the least significant bit. This is done with the concatenation operator {}.

# Four-bit parallel in shift left register with load and shift enable signal

```
module Parallel_in_serial_out_load_enable_behavior(input Clk, input ShiftIn,
input [3:0] ParallelIn, input load, input ShiftEn, output ShiftOut, output
[3:0] RegContent);
    reg [3:0] shift_reg;

always @(posedge Clk)
       if(load)
          shift_reg <= ParallelIn;
       else if (ShiftEn)
          shift_reg <= {shift_reg[2:0], ShiftIn};
    assign ShiftOut = shift_reg[3];
    assign RegContent = shift_reg;

endmodule
```

# Counters

- Asynchronous Counter: count the number of events using an event signal
- Synchronous Counter: use a common clock signal



- Additional Constructs for Behavioral Modeling
- Timing Constraints

# Additional Constructs for Behavioral Modeling

Primary mechanisms to model behavior of a design

- initial: mainly used for testbenches to generate inputs at a desired time
- always: mainly used to describe functionality of a circuit

Within initial and always statements , one may have procedural statements, namely:

- procedural_assignment (blocking or non-blocking)
- conditional_statement
- case_statement
- loop_statement
- wait_statement
- event_trigger
- sequential_block
- task (user or system)

Note: When multiple procedural statements are enclosed between begin ... end, they execute sequentially

# Since always statements executes continuously, they are typically controlled using either delay control or event control mechanisms

```
always
   #5 CLK = ~CLK;
```

Time delay between the statement encountered and actually executed is 5 time units.

Inter-statement delay

```
initial
begin
   #5 SIG1 = 3;
   #4 SIG1 = 7;
   #2 SIG1 = 4;
end
```

Intra-statement delay

```
DONE = #5 1'b1
```

The right hand side expression is evaluated when encountered, the result of the expression is only assigned after the stated delay.

Executes only when there is a change in value (an event) on a wire *test*

```
wire test;
always @(test)
begin
   #5 CLK = ~CLK;
end
```

Executes only when there is a rising edge change (edge-triggered events )

```
wire test;
always @(posedge test)
begin
   #5 CLK = ~CLK;
end
```

Level-sensitive event control

```
wait (SUM > 22)
   SUM = 0;

wait (DATA_READY)
   DATA = BUS;
```

# Loop Statements

forever: used when the procedural statement(s) need to be executed continuously

```
initial
begin
  CLK = 0;
  forever
    #10 CLK = ~CLK;
  end
```

repeat: used when the procedural statement(s) need to be executed for a specified number of times

```
repeat (COUNT)
   SUM = SUM +5;
```

while: procedural statement(s) are executed until certain conditions become false

```
while (COUNT < COUNT_LIMIT)
   SUM = SUM +5;
```

for: used when the procedural statement(s) need to be executed for a specified number of times. Unlike the repeat statement, an index variable is used which can be initialized to any desired value, and a condition can be given to terminate the loop statement.

```
integer K;
for (K=0; K < COUNT_LIMIT; K = K+1)
   SUM = SUM + K;
```

# Parameterized Model

Reuse the same model a number of times by passing a number of parameter values

Construct -> parameter

i.e. parameter WIDTH = 4;

Two ways to change the values:

1. During instantiation
2. Using defparam

Example

Default Value

```
module reduction #(parameter WIDTH = 4)
    (input ain [WIDTH-1:0], output result);
    assign result = | ain
endmodule


module test_reduction_tb;
    reg [3:0] ain1;
    reg [6:0] ain2;
    reg [9:0] ain3;
    wire result1, result2, result3;

    reduction U1(ain1,result1); //use default value
    reduction #(7) U2 (ain2, result2);
    reduction #(10) U3 (ain3, result3);
endmodule
```

Note:

- The parameter is defined before it's used
- In case of multiple parameter definitions, the value listed during instantiation maps to the order in which the parameters are defined
- All parameters must be defined
- The parameter definition can be used in the same file where it's used or in a separate file

  Example:

  parameter WIDTH = 4;  could be in file header.v

  ‘include header.v // or add full path to file

# Timing Constraints

Delay through combinational circuits depends on:

- number of logic levels
- number of gate inputs a net drives (fan-out)
- the capacitive loading on the output nets.

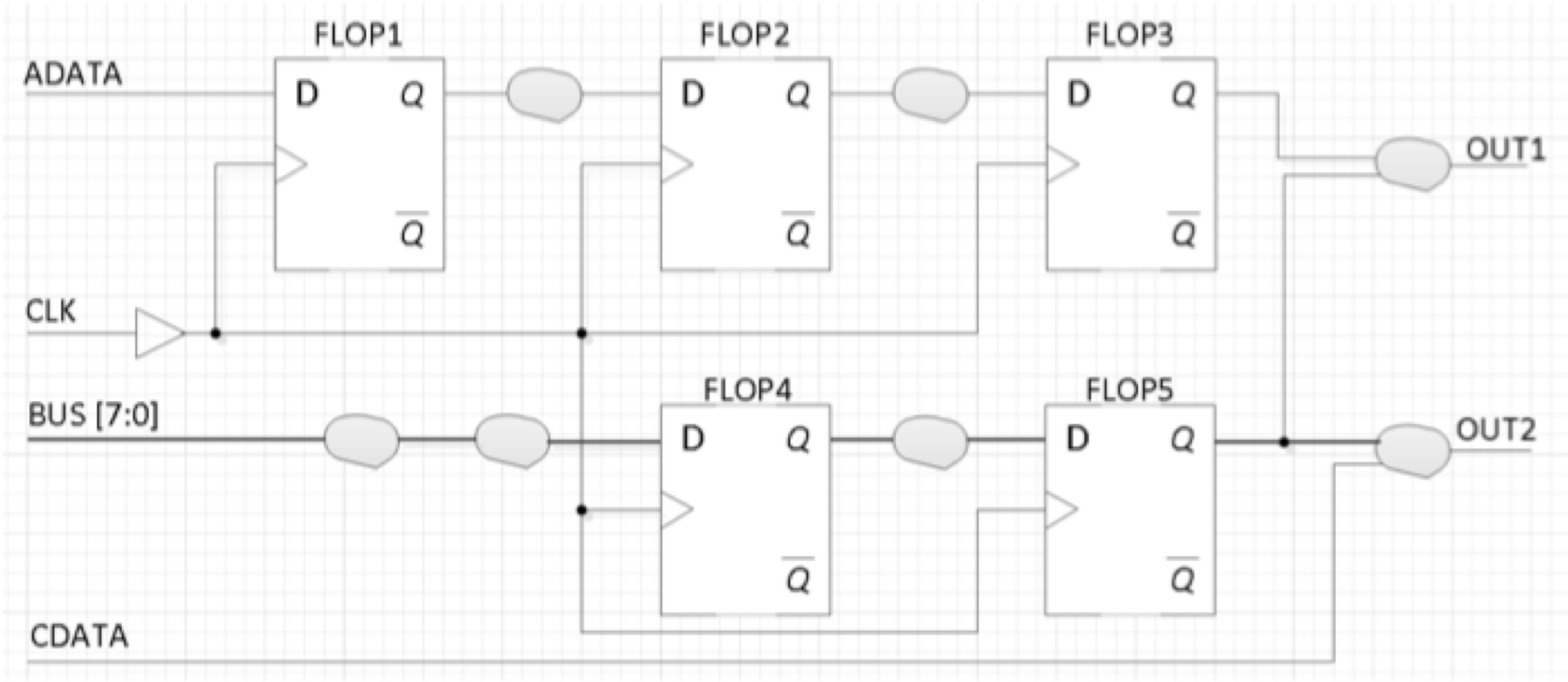
This affects the clock speeds at which sequential designs can be operated. Timing constrains allows communicating expected performance for appropriate placement of design in LUTs, flip-flops and registers when synthesized and implemented

- global timing
- path specific (have higher priority and routed first)

**Combinatorial design**: path-to-path constraint is used

**Sequential circuits**: period, input delay, and output delay constraints are used

`set_input_delay` create_clock set_output_delay set_max_delay



CLOCK_DEDICATED_ROUTE provides a means to demote a clock placement DRC (design rule checking) from an error to a warning when a clock source is placed in a sub-optimal location compared to its load clock buffer. Setting CLOCK_DEDICATED_ROUTE to False may result in sub-optimal clock delays resulting in potential timing and other issues.

The following line of code needs to be added to the XDC file to allow s Switch be used as a clock.

```
set_property CLOCK_DEDICATED_ROUTE value [get_nets net_name]
```

Where, net_name is the signal name connected to the input of a global clock buffer.

Example:

```
# Designates clk_net to have relaxed clock placement rules
set_property CLOCK_DEDICATED_ROUTE FALSE [get_nets clk_net];
```

https://www.xilinx.com/support/documentation/sw_manuals/xilinx2012_2/ug912-vivado-properties.pdf

# Synthesis Constraints

Synthesis Engine

Transforms

RTL Description

Timing-driven optimizations

Technology Mapped Netlist

Constraints are needed to guide the synthesis engine towards a solution that meets all the design requirements at the end of implementation.

# Categories of Constraints

## RTL Attributes

Directives related to

- the mapping style of certain part of the logic,
- preserving certain registers and nets, or
- controlling the design hierarchy in the final netlist.

## Timing Constraints

Passed to the synthesis engine by means of one or more XDC files.

- create_clock, set_input_delay, set_output_delay, set_max_delay, set_multicycle_path, create_generated_clock, set_clock_groups, set_false_path.

## Physical and Configuration Constraints

Ignored by the synthesis algorithms.

https://www.xilinx.com/support/documentation/sw_manuals/xilinx2012_2/ug903-vivado-using-constraints.pdf
Vivado Design Suite User Guide: Synthesis

# Fundamentals of Timing Analysis

- Timing Paths
- Clocks
- I/O Delay
- Setup and Hold Analysis
- Recovery and Removal Analysis

# Timing Paths

Defined by the connectivity between the instances of the design. In digital logic, they are formed by a pair of sequential elements controlled by the same clock, or by two different clocks:

**Input Port to Internal Sequential Cell Path**

Launched outside device by port clock, reaches device port after input delay, propagates through internal logic reaching a sequential cell clocked by the destination clock.

**Internal Path from Sequential Cell to Sequential Cell**

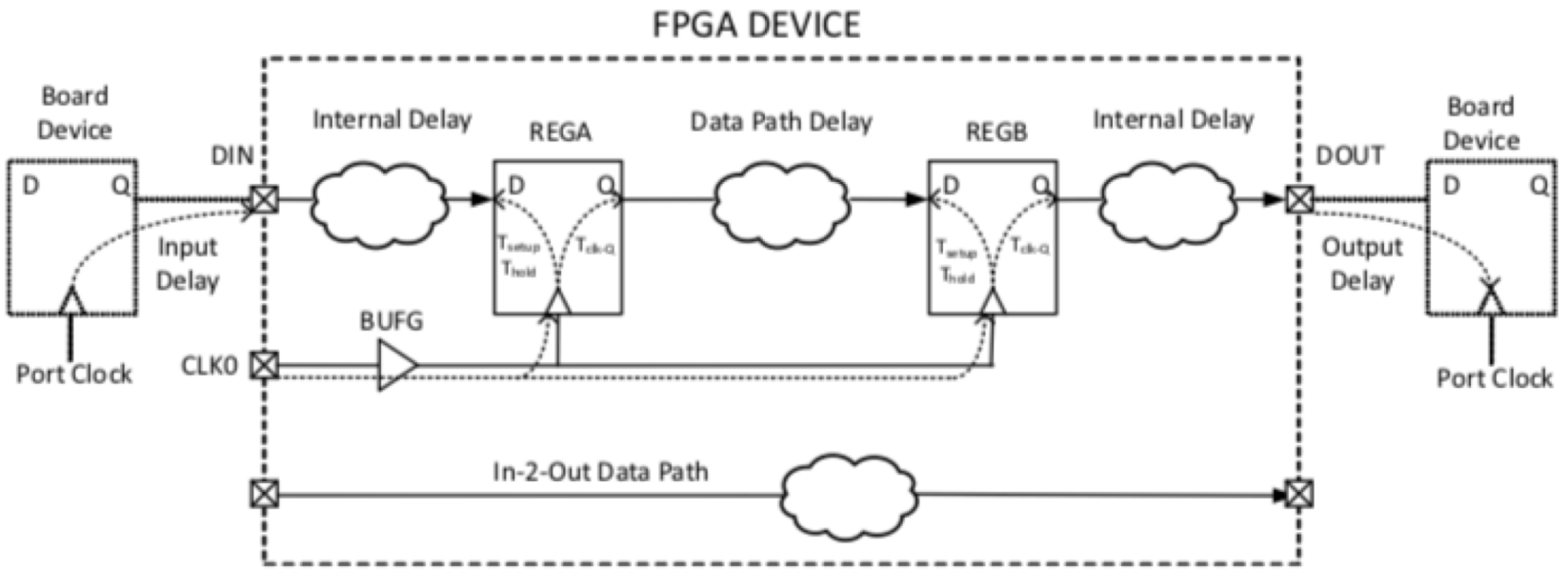
Launched by sequential cell clocked by source clock, propagates through internal logic reaching a sequential cell clocked by destination clock.

**Internal Sequential Cell to Output Port Path**

Launched inside the device by a sequential cell clocked by the source clock, propagates through internal logic reaching the output port, captured by a port clock after an additional delay called output delay.

**Input Port to Output Port Path**

Propagates directly from an input port to an output port without being latched inside the device.

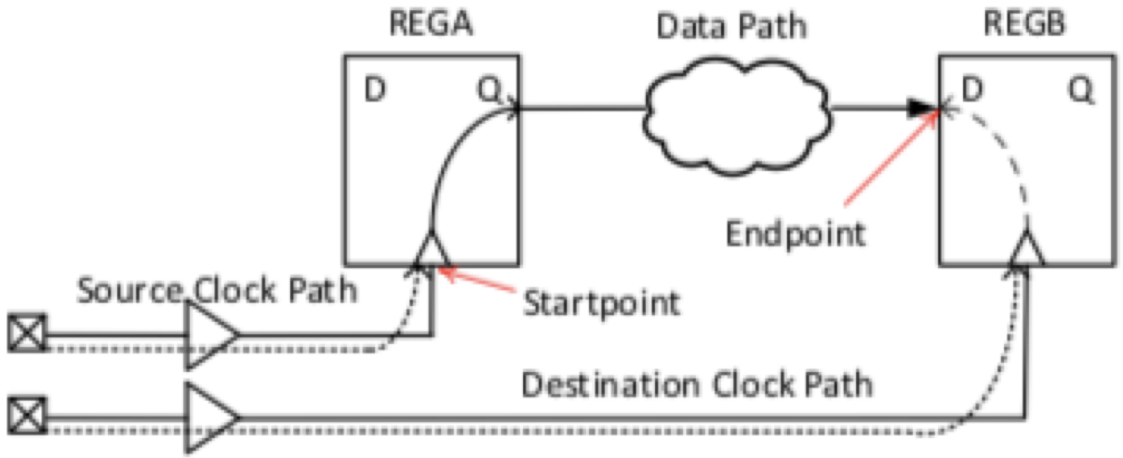


# Timing Path Sections

- Source Clock Path: the path followed by the source clock from its source point to the clock pin of the launching sequential cell

- Data Path: the data path is the path between the launching and capturing sequential cells.

- Destination Clock Path: the path followed by the destination clock from its source point, to the clock pin of the capturing sequential cell.



# Primary Clocks

Defined by the create_clock command.

- It must be attached to a netlist object. This netlist object represents the point in the design from which all the clock edges originate and propagate downstream on the clock tree. In other words, the source point of a primary clock defines the time zero used by the Vivado IDE when computing the clock latency and uncertainty used in the slack equation.

# Examples

The board clock enters the device through the port sysclk, then propagates through an input buffer and a clock buffer before reaching the path registers.

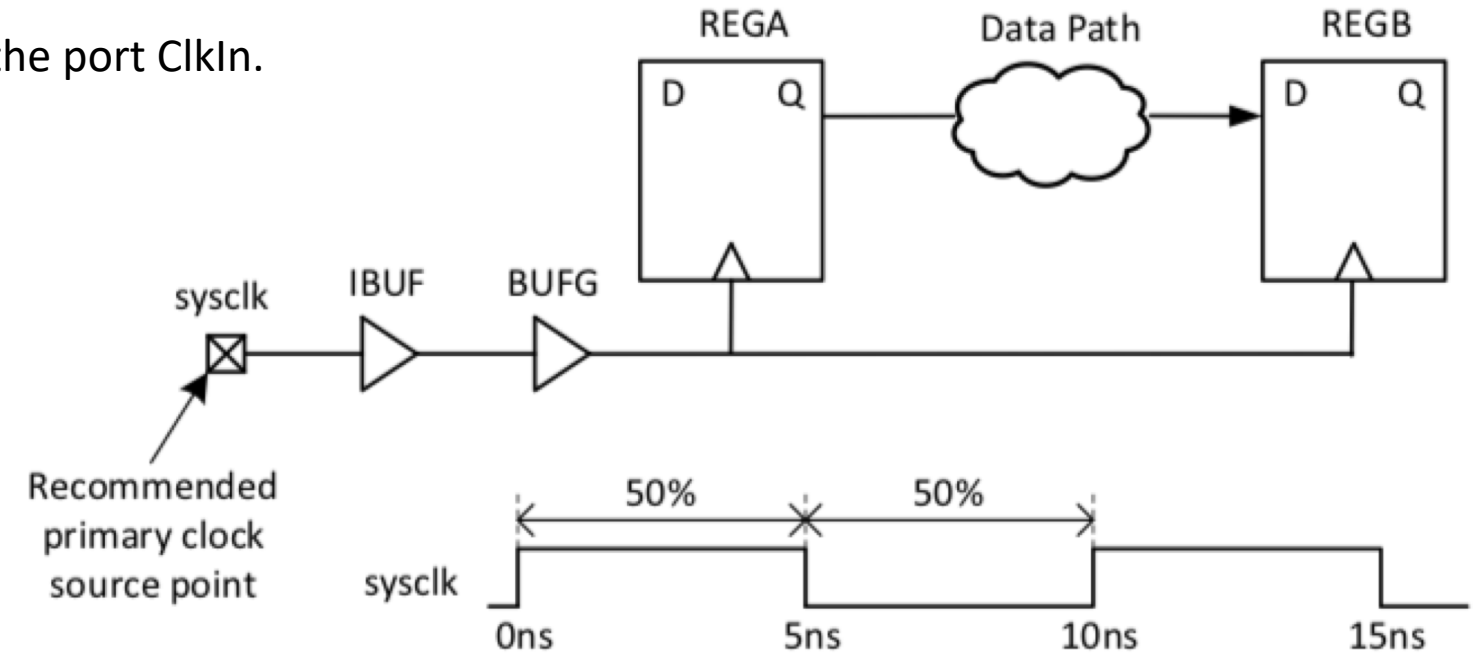
```
create_clock -period 10 [get_ports sysclk]
```

- Its period is 10ns.
- Its duty cycle is 50%
- Its phase is not shifted.

```
create_clock -name devclk -period 10 -waveform {2.5 5} [get_ports ClkIn]
```

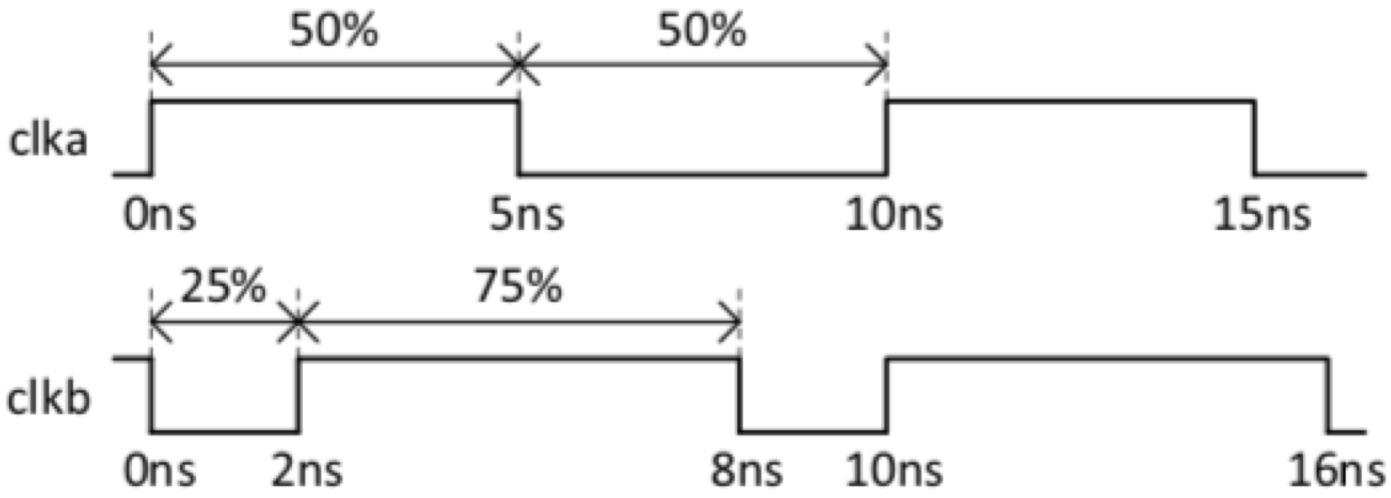
a board clock devclk enters the device through the port ClkIn.

- Its period is 10ns.
- Its duty cycle is 25%.
- It is phase shifted by 90 degrees.



# Waveform

```
Clk0: period = 10, waveform = {0 5}
Clk1: period = 8, waveform = {2 8}
```




The clock Clk0 has a 10ns period, a 50% duty cycle and 0ns phase.

The clock Clk1 has 8ns period, 75% duty cycle and a 2ns phase shift.

# I/O Delay

Model external timing to specify delay values that exist external to the FPGA

## set_input_delay

specifies the input path delay on an input port relative to a clock edge at the interface of the design representing the phase difference between:

- The data propagating from an external chip through the board to an input package pin of the FPGA device,
- The relative reference board clock.

set_input_delay –clock CLK –max 3.0 [all_inputs]

set_output_delay

# set_input_delay

specifies the input path delay on an input port relative to a clock edge at the interface of the design representing the phase difference between:

- The data propagating from an external chip through the board to an input package pin of the FPGA device,
- The relative reference board clock.

set_input_delay -clock CLK -max 3.0 [all_inputs]

relative clock  Input Delay

# Input Delay Options

The **-min** and **-max** options specify different values for:

- Min delay analysis (hold/removal)
- Max delay analysis (setup/recovery).
- If neither is used, the input delay value applies to both min and max.

**-clock_fall** specifies that the input delay constraint also applies to timing paths launched by the falling clock edge of the relative clock

The **-add_delay** must be used if two max or min delay constrains exist, i.e. DDR interfaces

Example:

```
➢ create_clock –name clk_ddr –period 6 [get_ports DDR_CLK_IN]
➢ set_input_delay –clock clk_ddr –max 2.1 [get_ports DDR_IN]
➢ set_input_delay –clock clk_ddr –max 1.9 [get_ports DDR_IN] –clock_fall –add_delay
➢ set_input_delay –clock clk_ddr –min 0.9 [get_ports DDR_IN]
➢ set_input_delay –clock clk_ddr –min 1.1 [get_ports DDR_IN] –clock_fall –add_delay
```

# set_output_delay

Specifies the output path delay of an output port relative to a clock edge at the interface of the design representing the phase difference between:

- The data propagating from the output package pin of the FPGA device, through the board to another device, and
- The relative reference board clock.

## Example

```
set_output_delay -clock CLK 2.0 [all_outputs]
```

# Output Delay Options

**-min** and **–max**

**-clock_fall**

**-add_delay**

## Example:

```
create_clock –name clk_ddr –period 6 [get_ports DDR_CLK_IN]
set_output_delay –clock clk_ddr –max 2.1 [get_ports DDR_OUT]
set_output_delay –clock clk_ddr –max 1.9 [get_ports DDR_OUT] –clock_fall –add_delay
set_output_delay –clock clk_ddr –min 0.9 [get_ports DDR_OUT]
set_output_delay –clock clk_ddr –min 1.1 [get_ports DDR_OUT] –clock_fall –add_delay
```

# Timing Expectations

It is needed when the logic behaves in a way that is not timed correctly by default. You must use a timing exception command any time you want the timing handled differently (for example, for logic that only has the result captured every other clock cycle by design).

| | |
|---|---|
| set_max_delay<br>set_min_delay | Sets the minimum and maximum path delay value. This overrides the default setup and hold constraints with user specified maximum and minimum delay values. |

- Constrain special paths, such as `in-to-out` I/O paths
- Override the default path requirement on selected paths usually defined by the clocks.

Example:

```
set_max_delay 5.0 –from [get_ports CDATA] –to [get_ports OUT2]
```

# Using the on-board CLK (ZedBoard)

The Zynq-7000 AP SoC's PS subsystem uses a dedicated 33.3333 MHz clock source

An on-board 100 MHz oscillator, supplies the PL subsystem clock input on bank 13, pin Y9.

.XDC File Constraints, using GCLK

```
set_property IOSTANDARD LVCMOS33 [get_ports clk]
set_property PACKAGE_PIN Y9 [get_ports clk]
create_clock -period 20 [get_ports clk]
```

**Hardware Specs:** http://zedboard.org/sites/default/files/documentations/ZedBoard_HW_UG_v1_9.pdf

**Clocking Wizard (change the clock frequency):** https://www.xilinx.com/support/documentation/ip_documentation/clk_wiz/v5_1/pg065-clk-wiz.pdf

CLOCK_DEDICATED_ROUTE provides a means to demote a clock placement DRC (design rule checking) from an error to a warning when a clock source is placed in a sub-optimal location compared to its load clock buffer. Setting CLOCK_DEDICATED_ROUTE to False may result in sub-optimal clock delays resulting in potential timing and other issues.

The following line of code needs to be added to the XDC file to allow s Switch be used as a clock.

set_property CLOCK_DEDICATED_ROUTE value [get_nets net_name]

Where, net_name is the signal name connected to the input of a global clock buffer.

Example:

# Designates clk_net to have relaxed clock placement rules

```
set_property PACKAGE_PIN H17 [get_ports {clk}];  # "SW6"
set_property CLOCK_DEDICATED_ROUTE FALSE [get_nets {clk}];
```

https://www.xilinx.com/support/documentation/sw_manuals/xilinx2012_2/ug912-vivado-properties.pdf

# Testbench for the shift Register Example

```
`timescale 1ns / 1ps


module sim;

    reg clk;
    reg clr;
    reg load;
    reg [3:0] D;

    wire [3:0] Q;

    integer i;

    ssr dut( .clk(clk), .clr(clr), .load(load), .D(D), .Q(Q));

    always #1 clk = !clk;

    initial
        begin
            clk = 0;
            clr = 0;
            load = 1;
            D = 0;
            for(i=0;i<16;i=i+1)
               #5 D = i;
            #10;
            clk = 0;
            clr = 1;
            load = 1;
            D = 0;
            for(i=0;i<16;i=i+1)
                #5 D = i;
            #10;
            clk = 0;
            clr = 1;
            load = 0;
            D = 0;
            for(i=0;i<16;i=i+1)
                #5 D = i;
            #10;
           end
endmodule
```

# Lab 4

**1. Model a 4-bit register with synchronous reset and load. Develop a testbench and simulate the design. Assign Clk, D input, reset, load, and output Q. Verify the design in hardware.**

a. Develop a testbench and simulate the design for **300ns**. Analyze the output.

b. Create the XDC file reflecting the LEDs and Switches of your choice for implementation. Assign the clock (clk) to one of the switches and remember to include the following

```
set_property CLOCK_DEDICATED_ROUTE FALSE [get_nets { clk }];
```

c. Synthesize, implement the design (Take a look at and report the project summary and utilization), generate the bitstream and download it into the ZedBoard to verify the functionality.

**2. Model a 4-bit parallel in left shift register. Develop a testbench and simulate the design. Assign Clk, ParallelIn, load, ShiftEn, ShiftIn, RegContent, and ShiftOut. Verify the design in hardware.**

a. Develop a testbench and simulate the design for **400ns**

b. Create the XDC file

c. Synthesize, implement the design (Take a look at and report the project summary and utilization), generate the bitstream and download it into the ZedBoard to verify the functionality.

**3. Design a 8-bit counter using T flip-flops. Usw a T flip-flop in behavioral modeling and rest either in dataflow or gate-level modeling. Develop a testbench and validate the design. Assign Clock input, Clear_n, Enable, and Q. Implement the design and verify the functionality in hardware.**

a. Develop a testbench and validate the design for 500 ns.

b. Create the XDC file.

c. Synthesize the design and view the schematic. Indicate what kind and how many resources are used

d. implement the design and generate the bitstream and download it into the ZedBoard to verify the functionality.

**4. Modify the counter designed in part 3 using a D flip-flops**

**5. Write a model of a counter which counts in the following sequence 000, 001, 011, 101, 111, 010, (repeat 000). The counter should use behavioral modeling and a case statement.** The counter should have an enable signal, a reset signal, and a clock signal. Display the output on any 3 LEDs.

a. Develop a testbench to test it displaying the counter output in the simulator console output.

b. Simulate using the clock period of 10 units for 200 ns.

**6. Implement a 3-bit Multiplier**

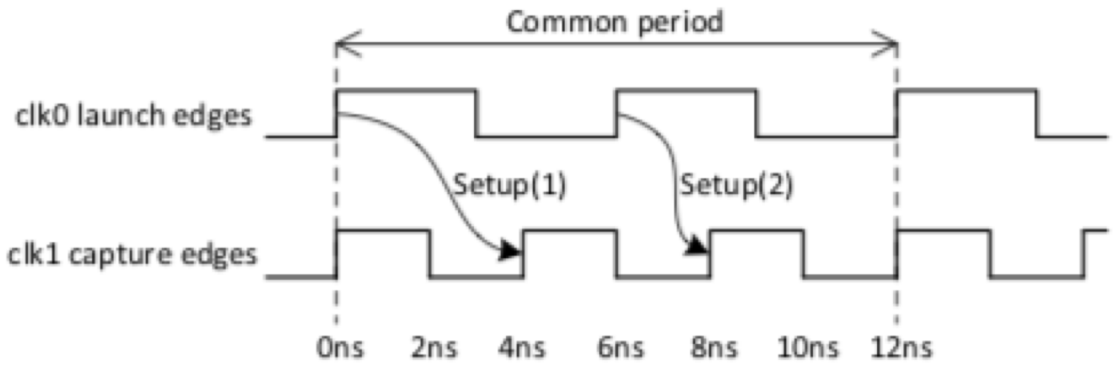
# Setup and Hold Analysis

Analyzes and reports slack at the timing path endpoints. That is, the difference between the data required time and the data arrival timing at the path endpoint.

Slack > 0 => functional path

## Setup Check Example

Consider a path between two registers which are sensitive to the rising edge of their respective clock. The active clock edges of this path are the rising edges only. The clocks are defined as follows:




There are two unique source and destination clock edges that qualify for setup analysis: setup(1) and setup(2). The smallest positive delta from clk0 to clk1 is 2ns, which corresponds to setup(2):

```
Source Clock Launch Edge Time: 0ns + 1 * T(clk0) = 6ns
Destination Clock Capture Edge Time: 0ns + 2 * T(clk1) = 8ns
Setup Path Requirement = capture edge time - launch edge time = 2ns
```