ELD Lab 1
Design of Full Adder

Lab Objective

- Design and implement a 4-bit adder for unsigned inputs using Full Adder
- Write the suitable testbench and verify the functionality of the full adder.
- **Lab Homework:** Extend the design to 4-bit adder/subtractor circuit for signed numbers

Verilog

Verilog

- One of the two major HDLs used by hardware designers in Industry and Academia (Another is VHDL)
- C- based Syntax, easy to master and intensively used by Indian VLSI Industry
- 1983: Introduced by Gateway Design System
- Invented as simulation language. Synthesis was an afterthought
- 1987: Verilog synthesizer by Synopsis
- 1989: Cadence acquired Gateway Design System and became the language owner

Verilog

- Around the same time (1981-1988), the US Department of Defence developed VHDL (VHSIC HDL). Because it was in the public domain it began to grow in popularity.
- Afraid of losing market share, Cadence opened Verilog to the public in 1990.
- 1995: Became IEEE Standard 1364
- 2001 and 2005: New and improved version of Verilog (made life much easier)
- Latest Verilog version is “System Verilog” .
- Ongoing efforts for automating the mapping of the code written in high level language (C, System C, Python) to Verilog/VHDL

Few words of wisdom

- One of the common challenge for beginners is to think of HDL as a computer program rather than as a shorthand for describing digital hardware.
- If you do not know approximately what hardware your HDL should synthesize into, you probably won't like what you get.
- You might create far more hardware than is necessary or you might write non-synthesizable code
- **THINK** of your system in terms of blocks of combinational logic, registers and FSMs. **SKETCH** these blocks on paper and show how they are connected **BEFORE** you start writing code.
- Describing hardware with a language is similar, however, to writing a parallel program

Verilog

- Verilog looks like C, but it describes hardware
- First understand the circuit and specifications you want then figure out how to code it in Verilog.
- A large part of ELD (before mid-sem) is knowing how to write Verilog that gets you the desired circuit.
- If you do one of these activities without the other, you will not enjoy the process of algorithms to architecture mapping.
- These two activities will merge at some point for you

Verilog (Three Concepts)

- Difference between Register and Wire (Next two lectures)
- Efficient Behavioral modelling
- Difference between blocking and non-blocking assignments

Verilog:

- Verilog HDL is a case-sensitive language
- All keywords are in lowercase (assign, for, always, fork, if, else, input, output…)
- Statements are terminated by a semicolon (;)
- Two data types: Net (wire) and variable (Reg, Integer, real, time, realtime)
- Primitive Logic Gates and Switch-Level Gates, are built-in (Rarely used in ELD)
- Single line comments begin with the “// “and end with a carriage return.
  // This is one line comment
- Multi Line comments begin with the “/*” and end with the “*/”
  /* This is a multiple
  lines
  comments */

Verilog: Identifiers

- Identifiers are names given to an object, such as a register or a function or a module, so that it can be referenced from other places in a description
- Identifiers must begin with an alphabetic character or the underscore character
- Identifier cannot start with a number or dollar sign
- Identifiers may contain alphabetic characters, numeric characters, underscore, and dollar sign
- Identifiers can be up to 1024 characters long
- Identifiers examples:
  wire outAdd ; // wire is a keyword, outAdd is an identifier
  reg sum ; // reg is a keyword, sum is an identifier

Verilog: Module

- Verilog describes a digital system as a set of modules
- Each module has an **interface** and **contents** description
- Modules communicate externally with input, output and bi-directional ports (inout)
- Verilog modules consist of a list of statements declaring relationships between a module and its environment, and between signals within a module

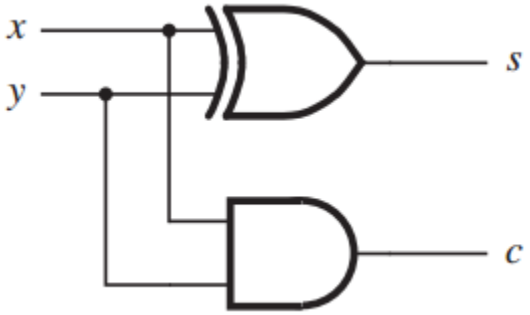
Verilog: Module (Examples)

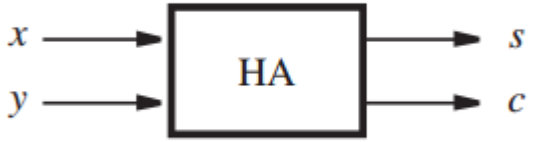

```
module AND (out, in1, in2) ; // <module name> <ports list>
    input in1, in2 ;
    output wire out ;
    assign out = in1 & in2 ;    // data flow - continuous Assignment
endmodule
```

Theory

HA and FA

x	y	Carry c	Sum s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0



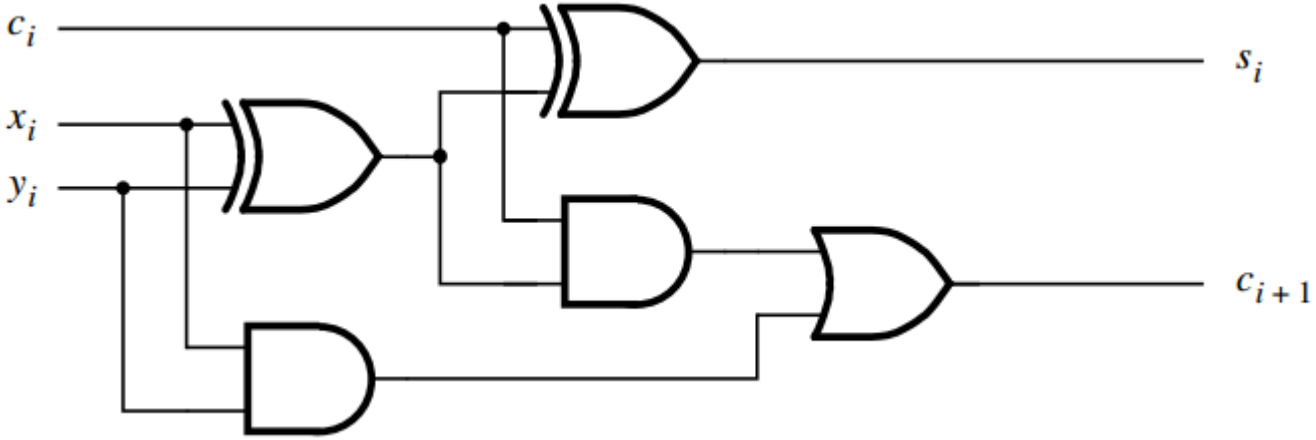


c_i	x_i	y_i	c_{i+1}	s_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$$c_{i+1} = \bar{c}_i \cdot x_i \cdot y_i + c_i \cdot \bar{x}_i \cdot y_i + c_i \cdot x_i \cdot \bar{y}_i + c_i \cdot x_i \cdot y_i$$

$$c_{i+1} = x_i \cdot y_i + c_i \cdot (\bar{x}_i \cdot y_i + x_i \cdot \bar{y}_i)$$

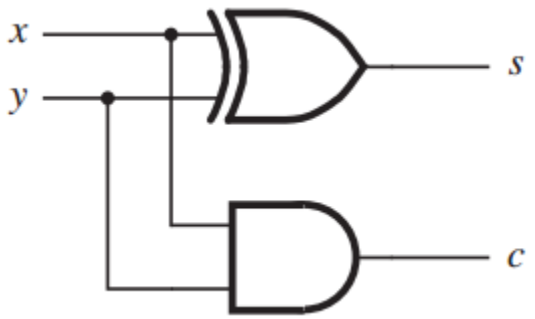
$$c_{i+1} = x_i \cdot y_i + c_i \cdot (x_i \oplus y_i)$$

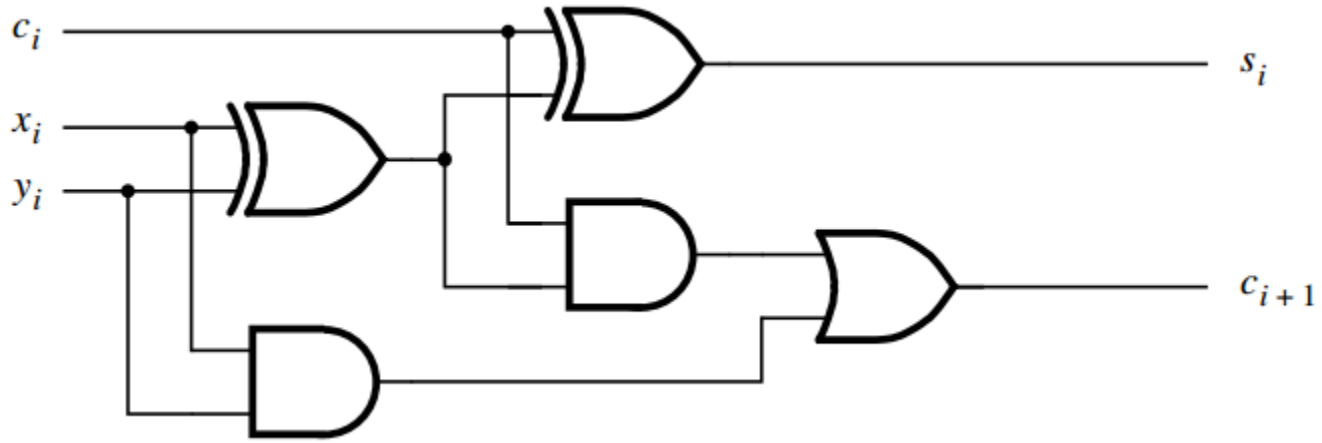


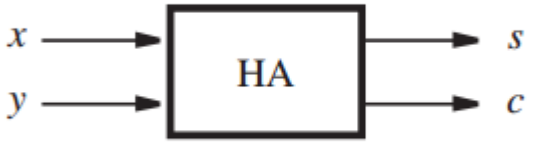
HA and FA

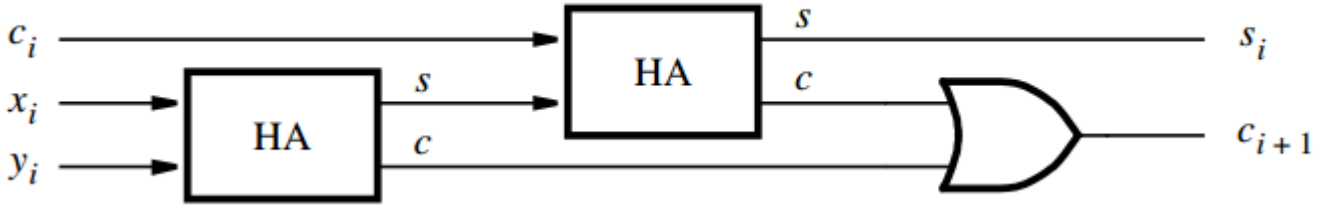
$$s_i = (x_i \oplus y_i \oplus c_i)$$

$$c_{i+1} = x_i \cdot y_i + c_i \cdot (x_i \oplus y_i)$$







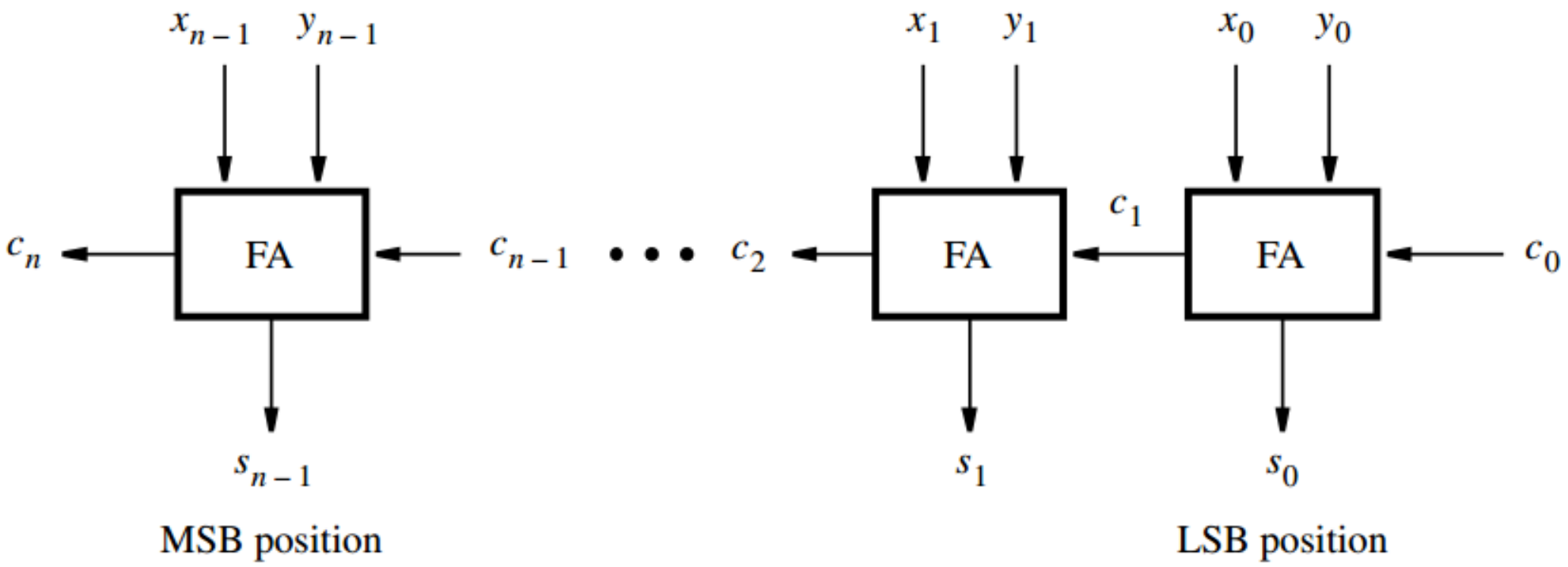


This approach minimizes the number of ICs needed to implement the circuit, and it reduces the wiring complexity substantially.

Verilog: 1 bit FA

```
module full_adder_1bit(
    input FA1_InA,
    input FA1_InB,
    input FA1_InC,
    output FA1_OutSum,
    output FA1_OutC
    );

    assign FA1_OutSum = FA1_InA^FA1_InB^FA1_InC;
    assign FA1_OutC = ((FA1_InA^FA1_InB)&FA1_InC)|(FA1_InA&FA1_InB);

endmodule
```

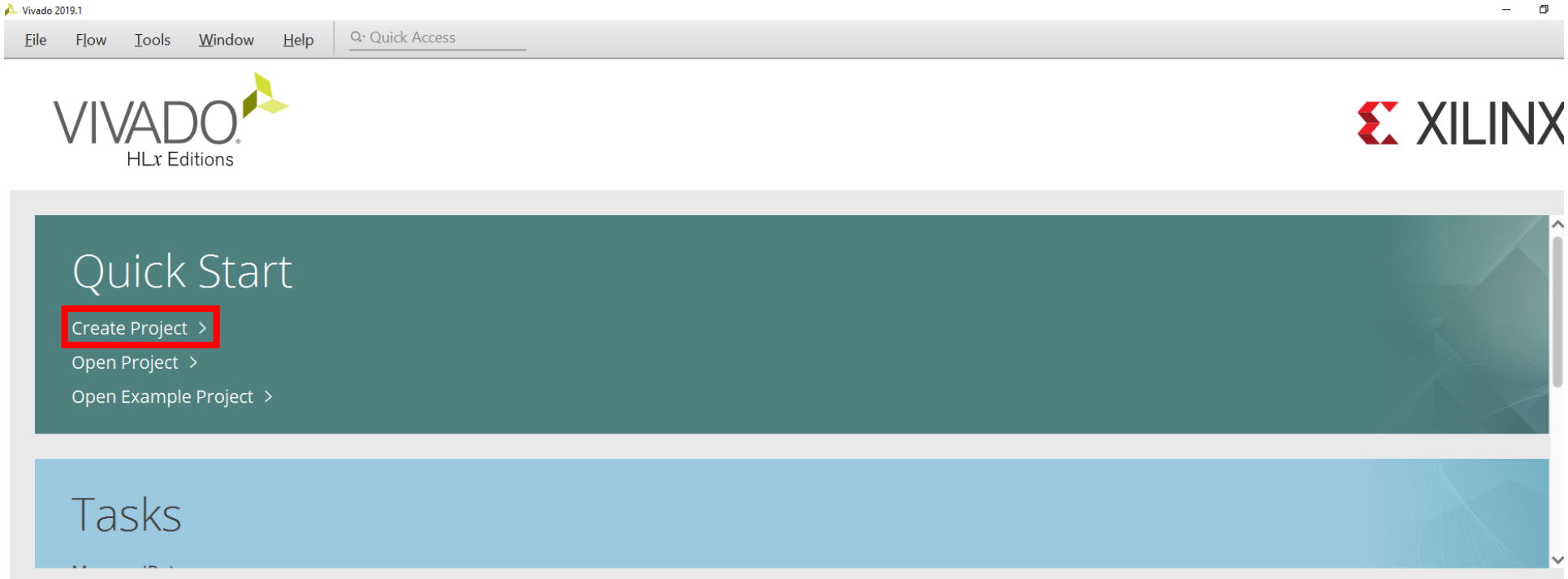
4-bit FA Block diagram



Lab

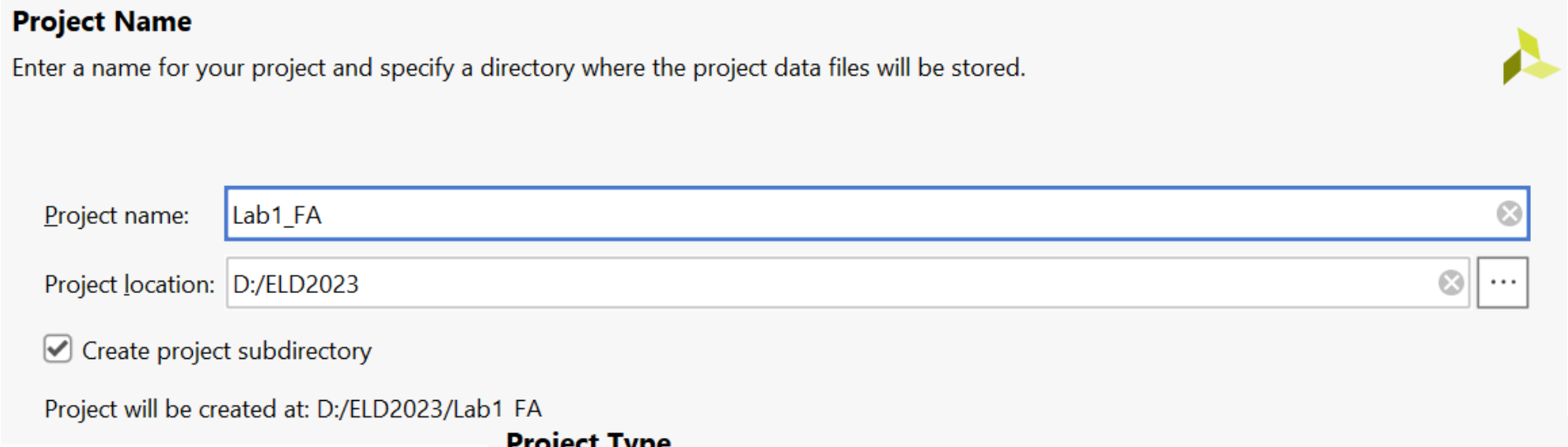
Open the Vivado

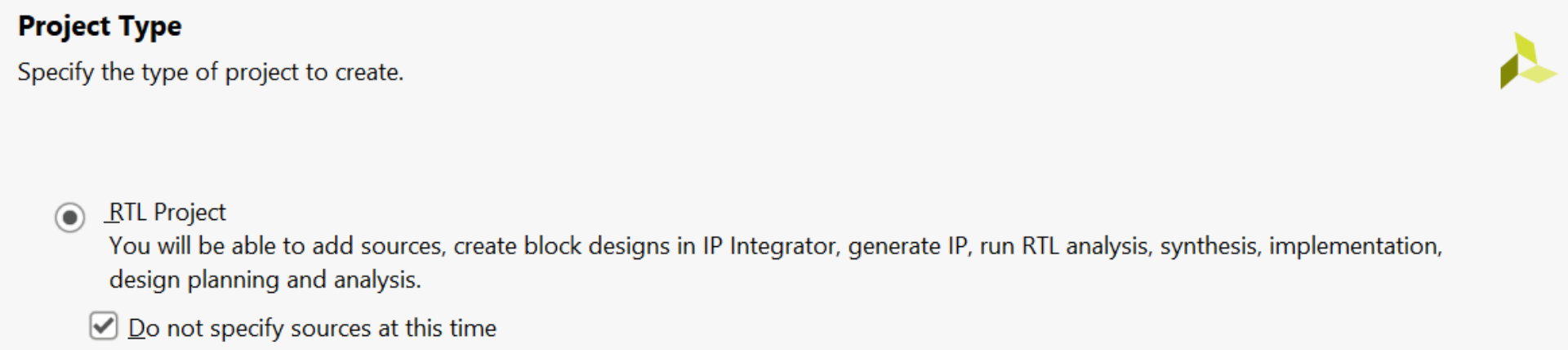
- Select Create Project and click on Next



Open the Vivado

- Select appropriate project folder.
- Avoid windows folder and space in address





Select Zedboard

Default Part

Choose a default Xilinx part or board for your project.

Parts | **Boards**

Reset All Filters

Update Board Repositories

Vendor: All

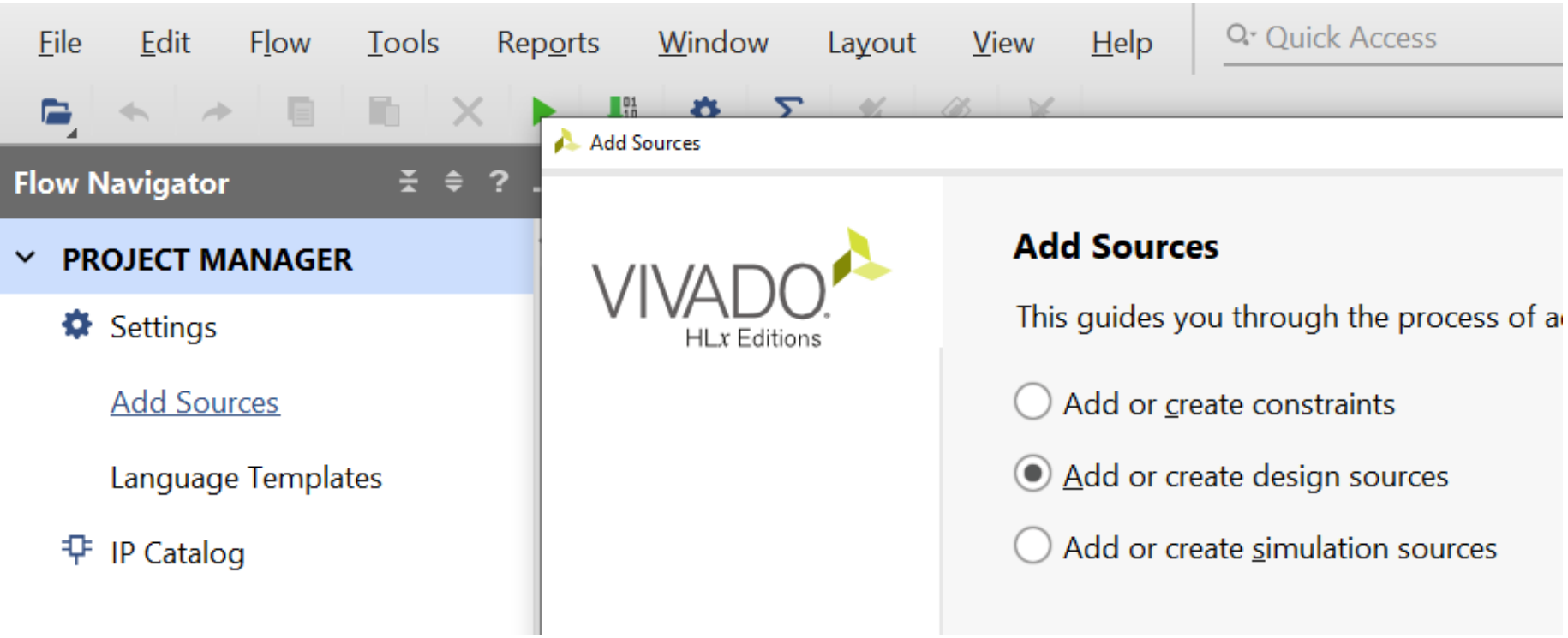
Name: All

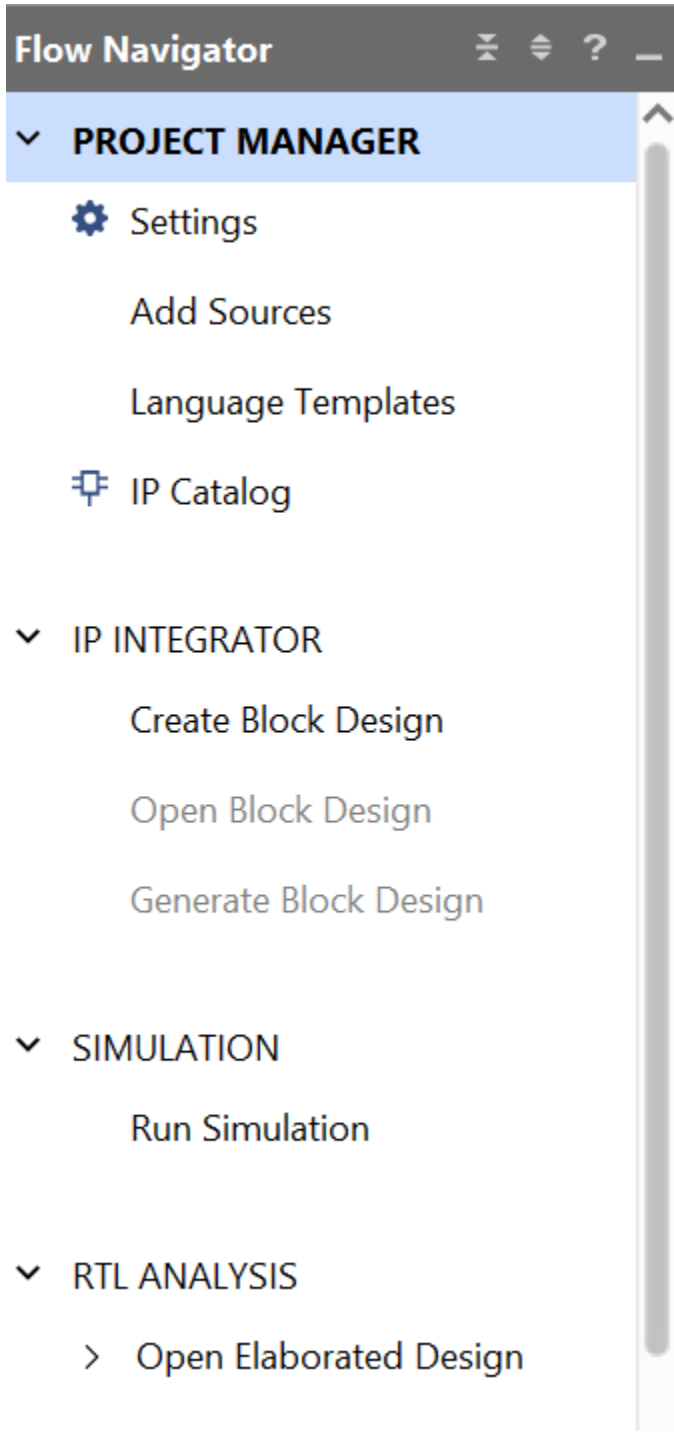
Board Rev: Latest

Search: zed (2 matches)

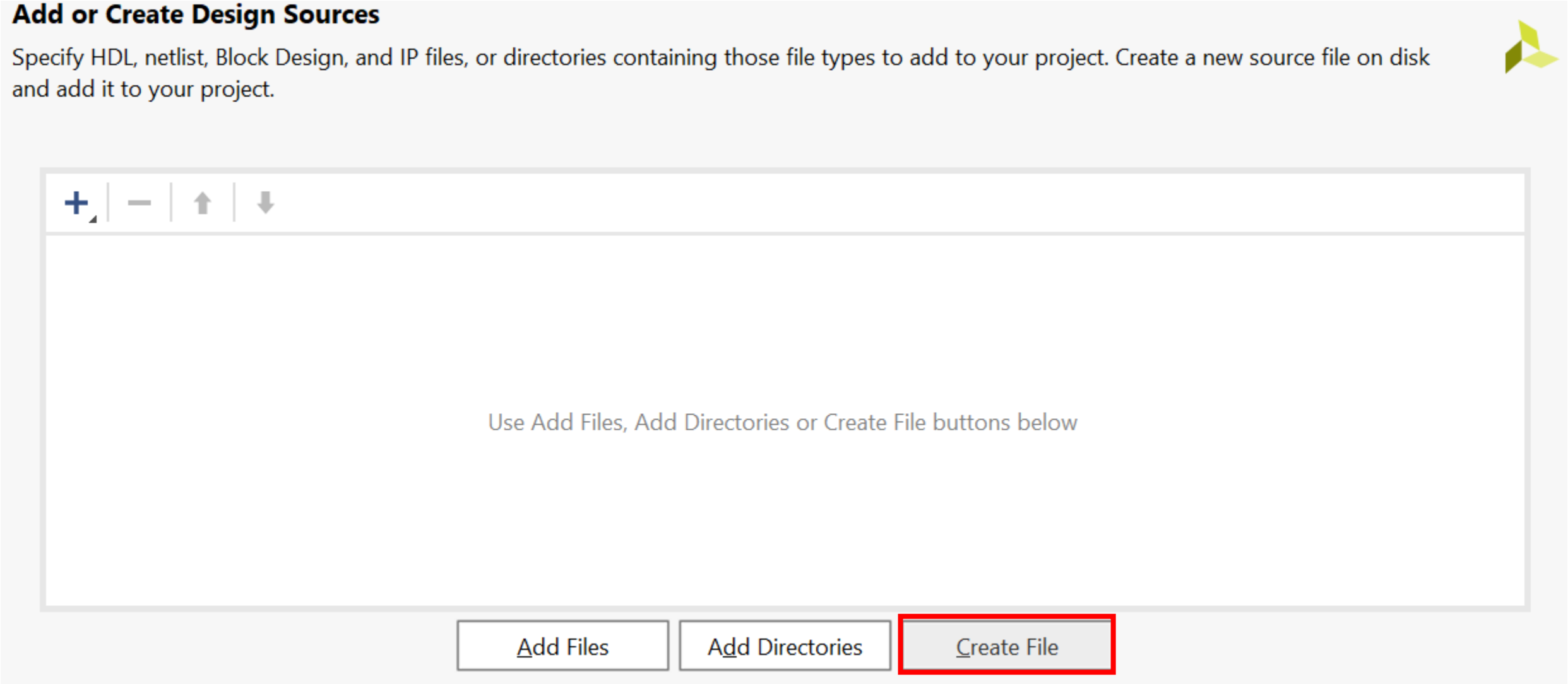
Display Name	Preview	Vendor	File Version	Part
Zedboard		digilentinc.com	1.0	xc7z020clg484-1
ZedBoard Zynq Evaluation and Development Kit Add Daughter Card Connections		em.avnet.com	1.4	xc7z020clg484-1

Vivado: Project Manager





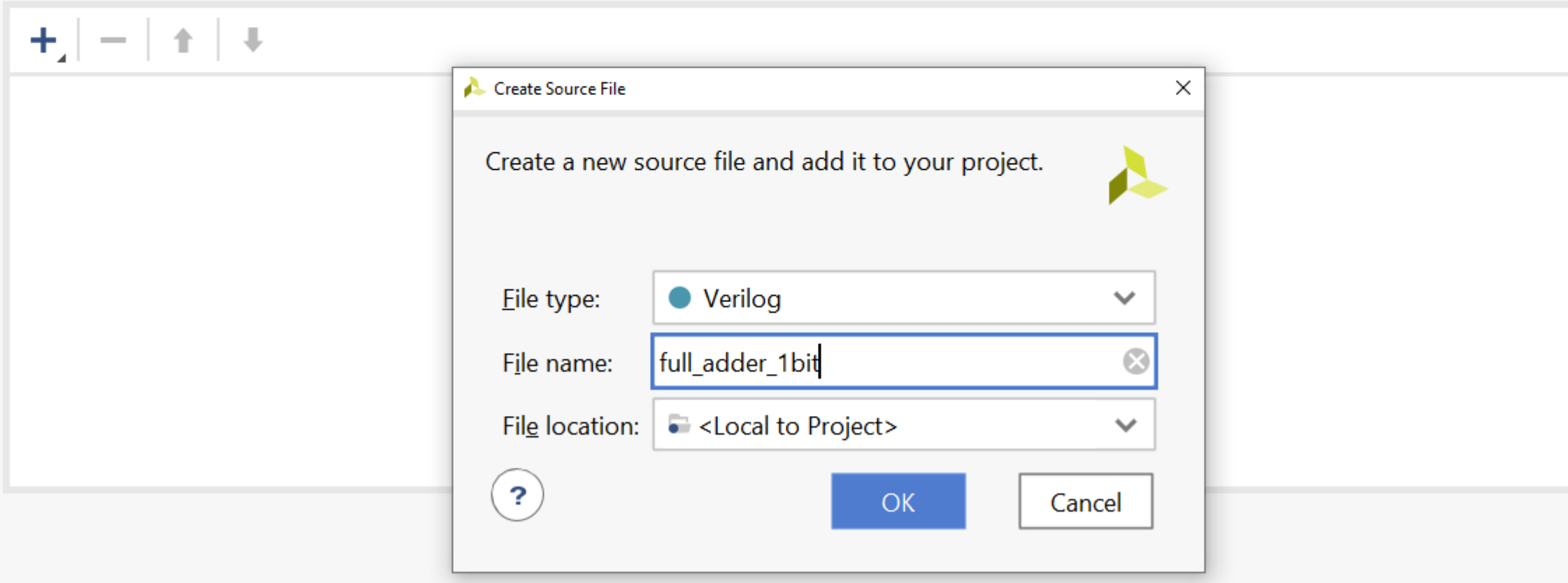
Design Sources



Design Sources

Add or Create Design Sources

Specify HDL, netlist, Block Design, and IP files, or directories containing those file types to add to your project. Create a new source file on disk and add it to your project.



Design Sources

PROJECT MANAGER - Lab1_FA

Sources

0

Design Sources (1)

full_adder_1bit (full_adder_1bit.v)

Constraints

Simulation Sources (1)

Hierarchy Libraries Compile Order

Define Module

Define a module and specify I/O Ports to add to your source file.
For each port specified:
MSB and LSB values will be ignored unless its Bus column is checked.
Ports with blank names will not be written.

Module Definition

Module name: full_adder_1bit

I/O Port Definitions

Port Name	Direction	Bus	MSB	LSB
	input	☐	0	0
FA1_InA	input	☐	0	0
FA1_InB	input	☐	0	0
FA1_InC	input	☐	0	0
FA1_OutSum	output	☐	0	0
FA1_OutC	output	☐	0	0

OK Cancel

Design Sources

File Edit Flow Tools Reports Window Layout View Help Quick Access

Flow Navigator

- **PROJECT MANAGER**
  - Settings
  - Add Sources
  - Language Templates
  - IP Catalog
- IP INTEGRATOR
  - Create Block Design
  - Open Block Design
  - Generate Block Design
- SIMULATION
  - Run Simulation

PROJECT MANAGER - Lab1_FA

Sources

- Design Sources (1)
  - **full_adder_1bit** (full_adder_1bit.v)
- Constraints
- Simulation Sources (1)
  - sim_1 (1)

Hierarchy | Libraries | Compile Order

Source File Properties

full_adder_1bit.v

Enabled

Location: D:/ELD2023/Lab1_FA/Lab1_FA.srcs/source

Project Summary | full_adder_1bit.v

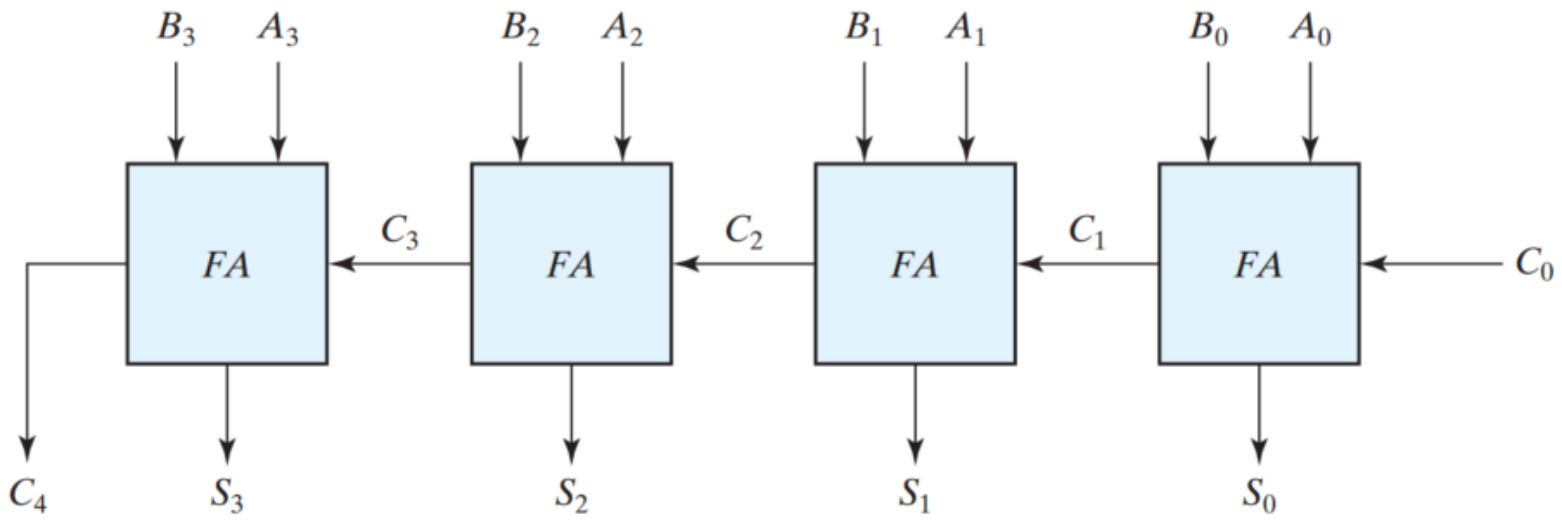
D:/ELD2023/Lab1_FA/Lab1_FA.srcs/sources_1/new/full_adder_1bit.v

```
22
23 module full_adder_1bit(
24     input FA1_InA,
25     input FA1_InB,
26     input FA1_InC,
27     output FA1_OutSum,
28     output FA1_OutC
29     );
30 endmodule
31
```

Design Sources

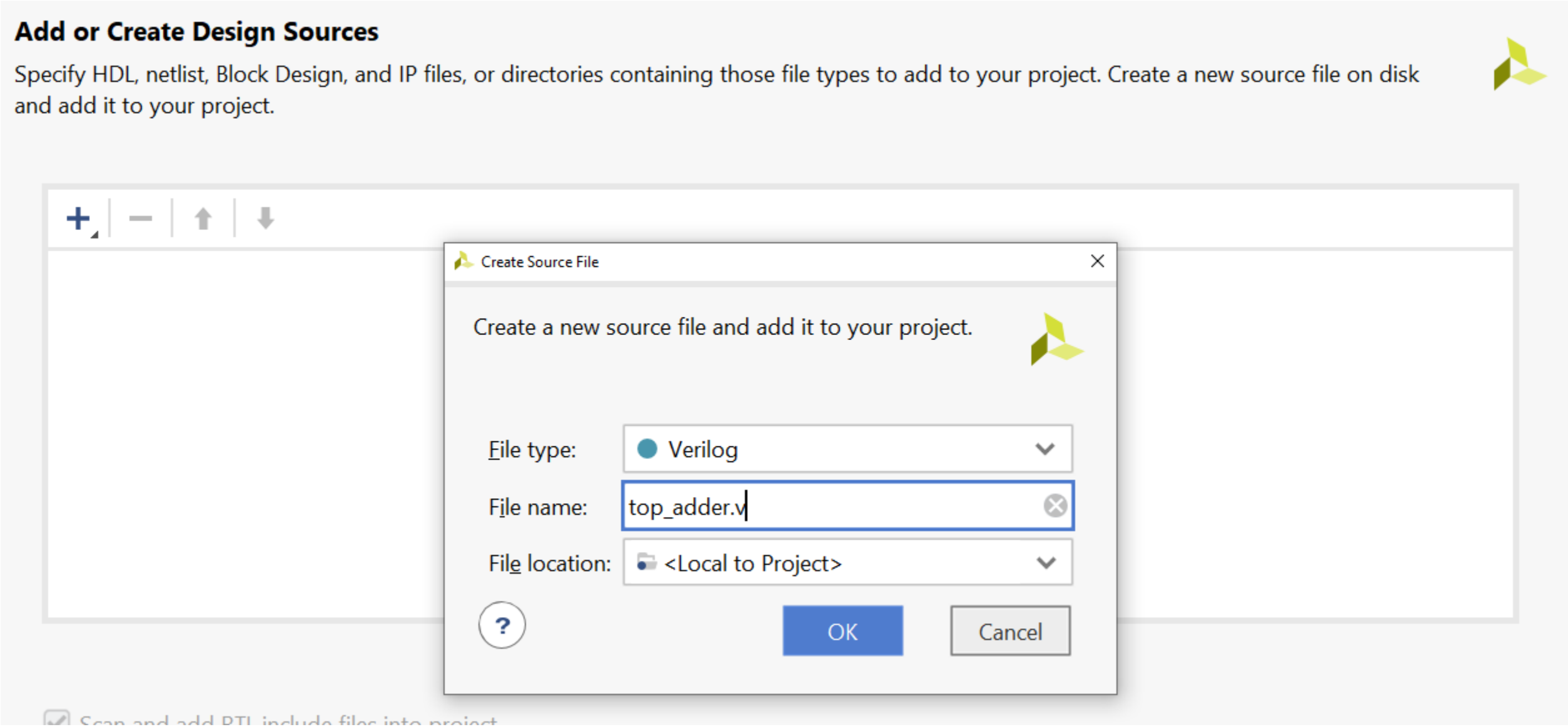
```
module full_adder_1bit(
    input FA1_InA,
    input FA1_InB,
    input FA1_InC,
    output FA1_OutSum,
    output FA1_OutC
    );
    assign FA1_OutSum = FA1_InA^FA1_InB^FA1_InC;
    assign FA1_OutC = ((FA1_InA^FA1_InB)&FA1_InC)|(FA1_InA&FA1_InB);
endmodule
```

4-bit FA

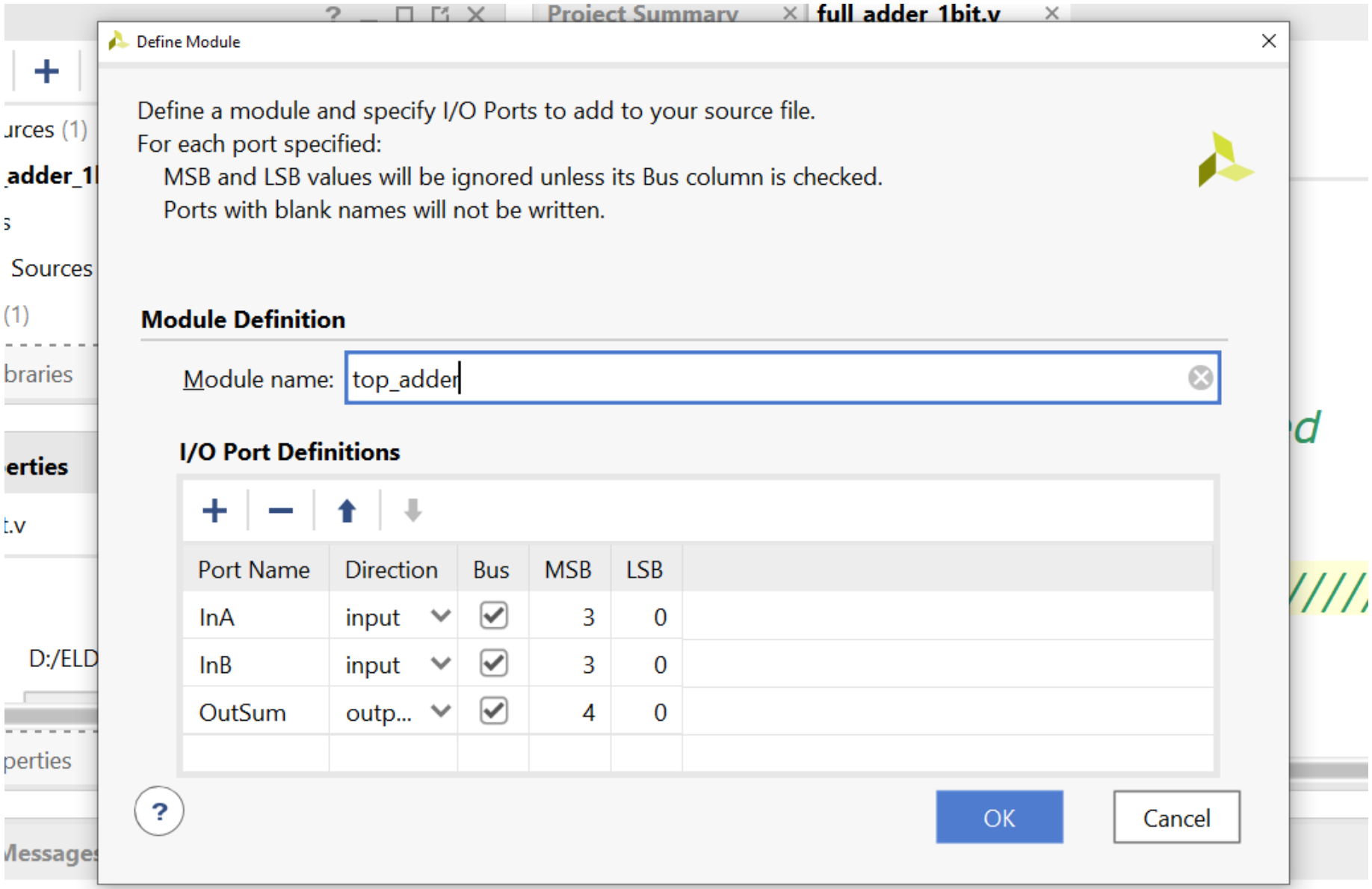


- Add new source file top_adder.v

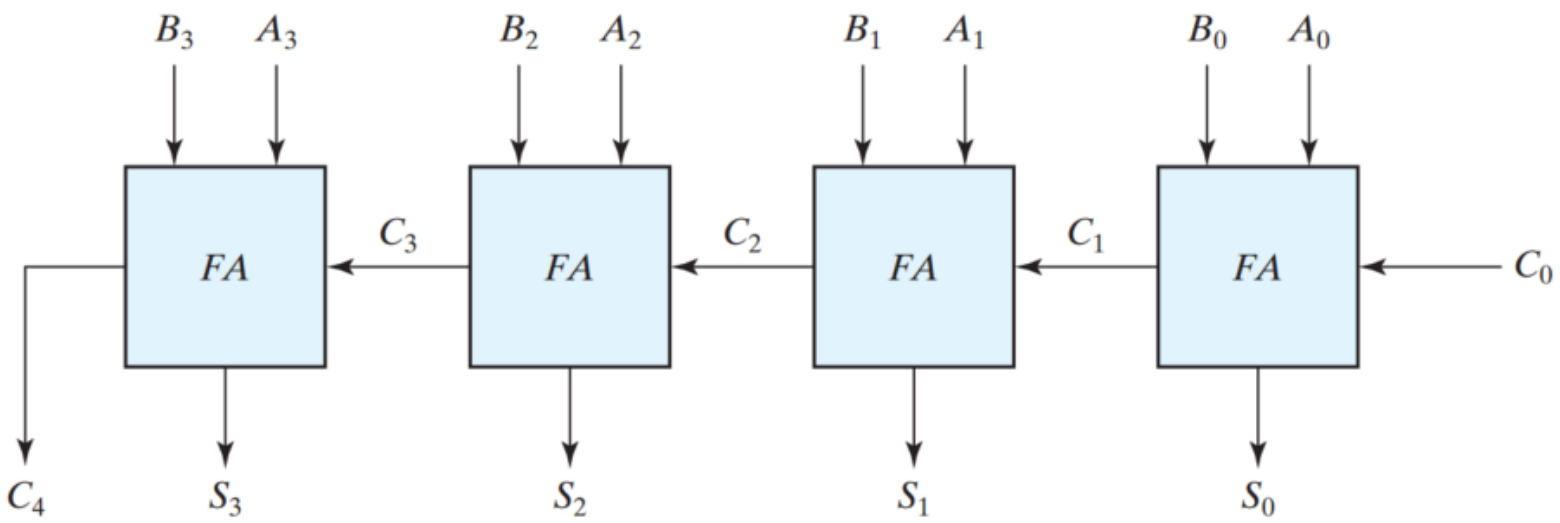
4-bit FA


Add or Create Design Sources
Specify HDL, netlist, Block Design, and IP files, or directories containing those file types to add to your project. Create a new source file on disk and add it to your project.
Create Source File
Create a new source file and add it to your project.
File type:
Verilog
File name:
top_adder.v
File location:
<Local to Project>
OK
Cancel

4-bit FA

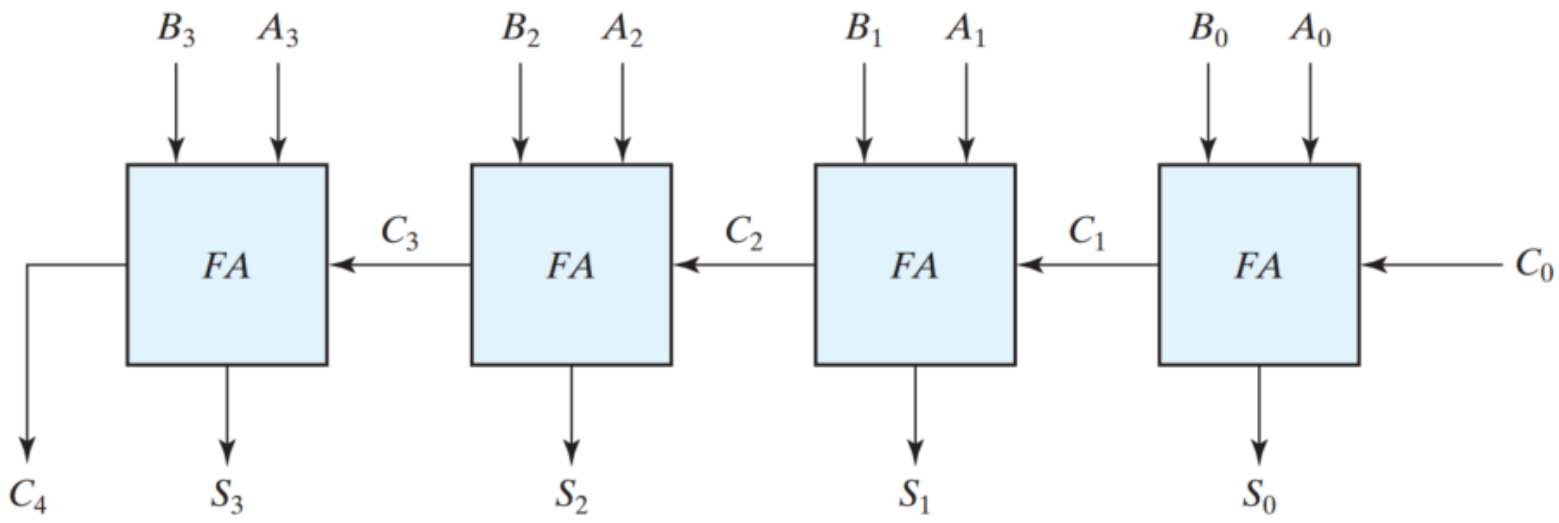

Project Summary
full_adder_1bit.v
Define Module
Define a module and specify I/O Ports to add to your source file.
For each port specified:
MSB and LSB values will be ignored unless its Bus column is checked.
Ports with blank names will not be written.
Module Definition
Module name: top_adder
I/O Port Definitions
Port Name
Direction
Bus
MSB
LSB
InA
input
3
0
InB
input
3
0
OutSum
outp...
4
0
OK
Cancel

4-bit FA

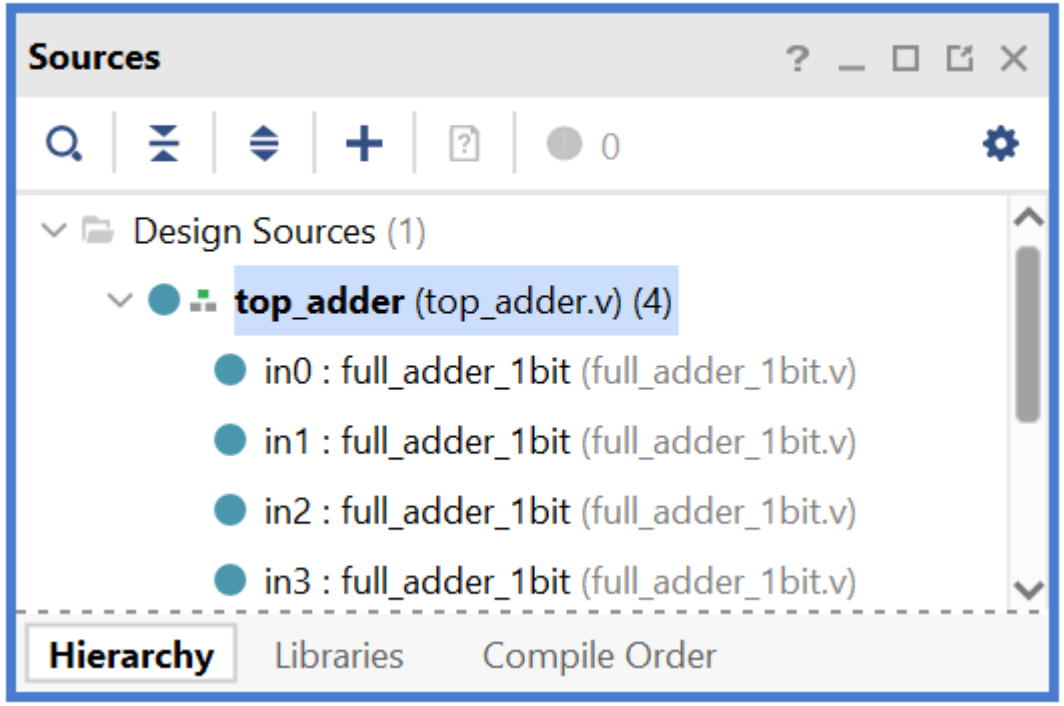



```
module top_adder(
    input [3:0] InA,
    input [3:0] InB,
    output [4:0] OutSum
    );

    wire carry1 , carry2, carry3;

    full_adder_1bit in0(.FA1_InA(InA[0]) , .FA1_InB(InB[0]) , .FA1_InC(1'b0) , .FA1_OutSum(OutSum[0]) , .FA1_OutC(carry1));
    full_adder_1bit in1(.FA1_InA(InA[1]) , .FA1_InB(InB[1]) , .FA1_InC(carry1) , .FA1_OutSum(OutSum[1]) , .FA1_OutC(carry2));
    full_adder_1bit in2(.FA1_InA(InA[2]) , .FA1_InB(InB[2]) , .FA1_InC(carry2) , .FA1_OutSum(OutSum[2]) , .FA1_OutC(carry3));
    full_adder_1bit in3(.FA1_InA(InA[3]) , .FA1_InB(InB[3]) , .FA1_InC(carry3) , .FA1_OutSum(OutSum[3]) , .FA1_OutC(OutSum[4]));

endmodule
```

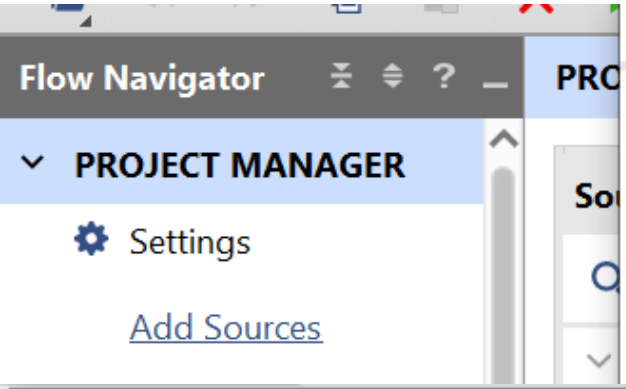
4-bit FA

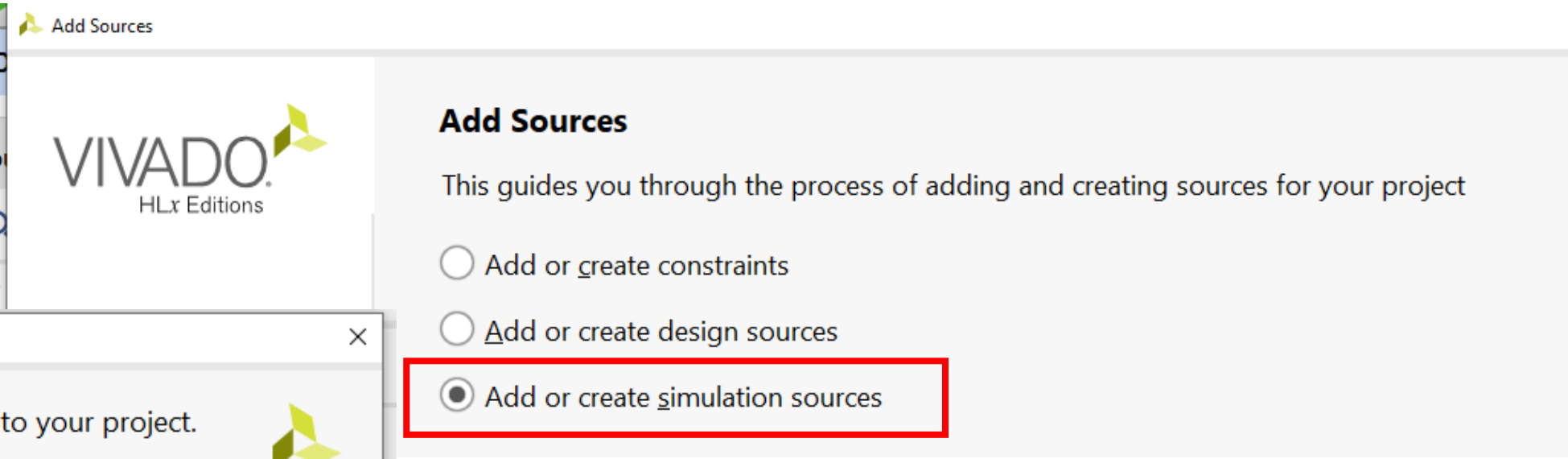


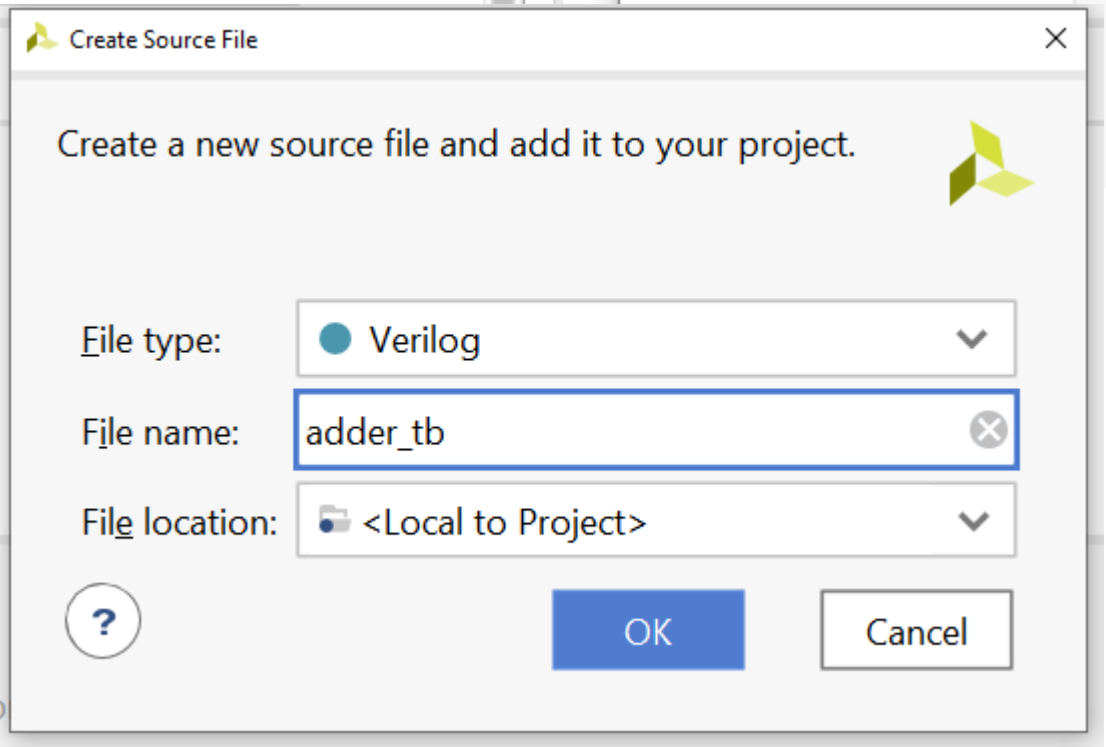
PROJECT MANAGER - Lab1_FA



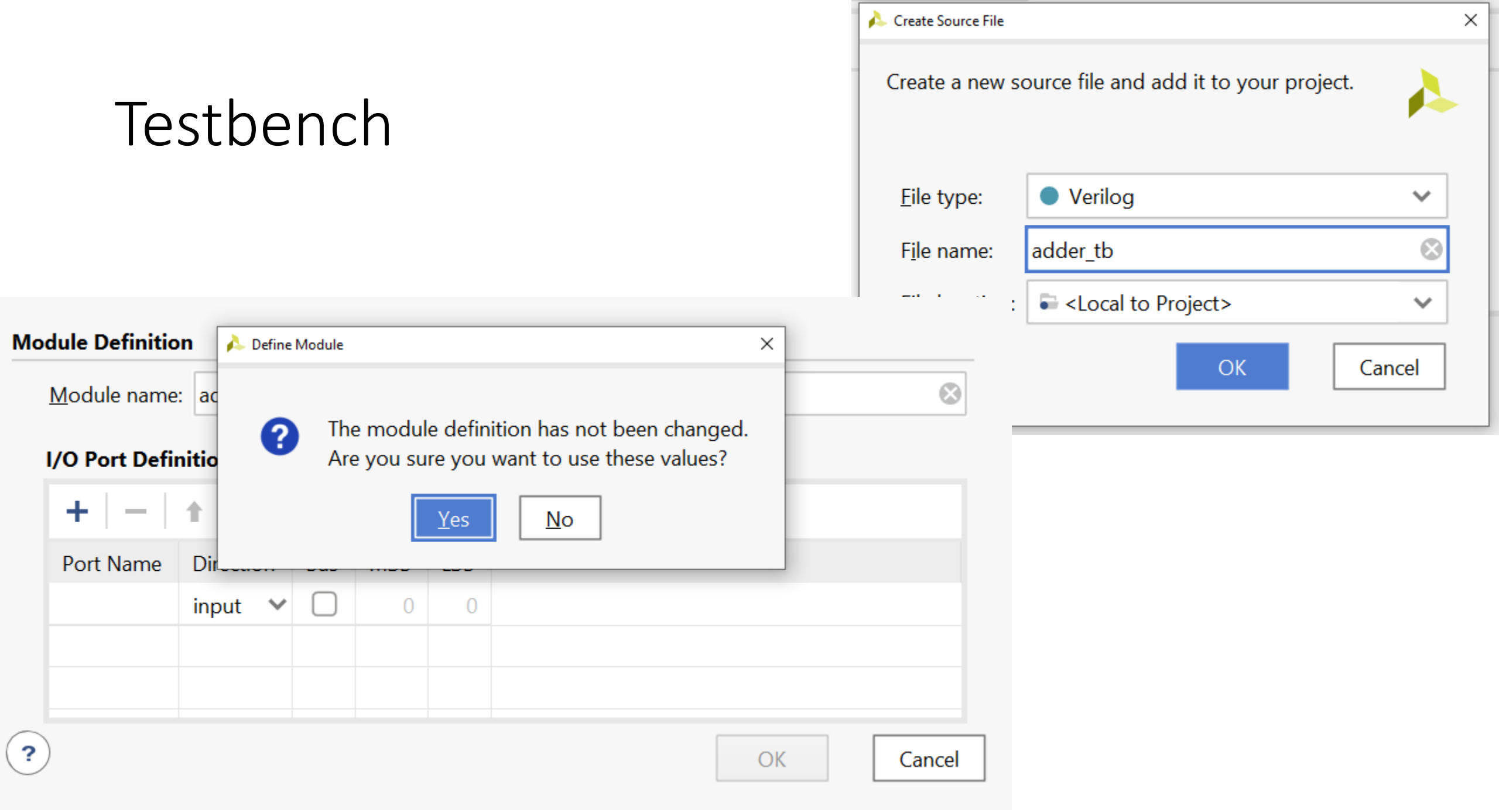
Testbench


Flow Navigator
PROJECT MANAGER
Settings
Add Sources

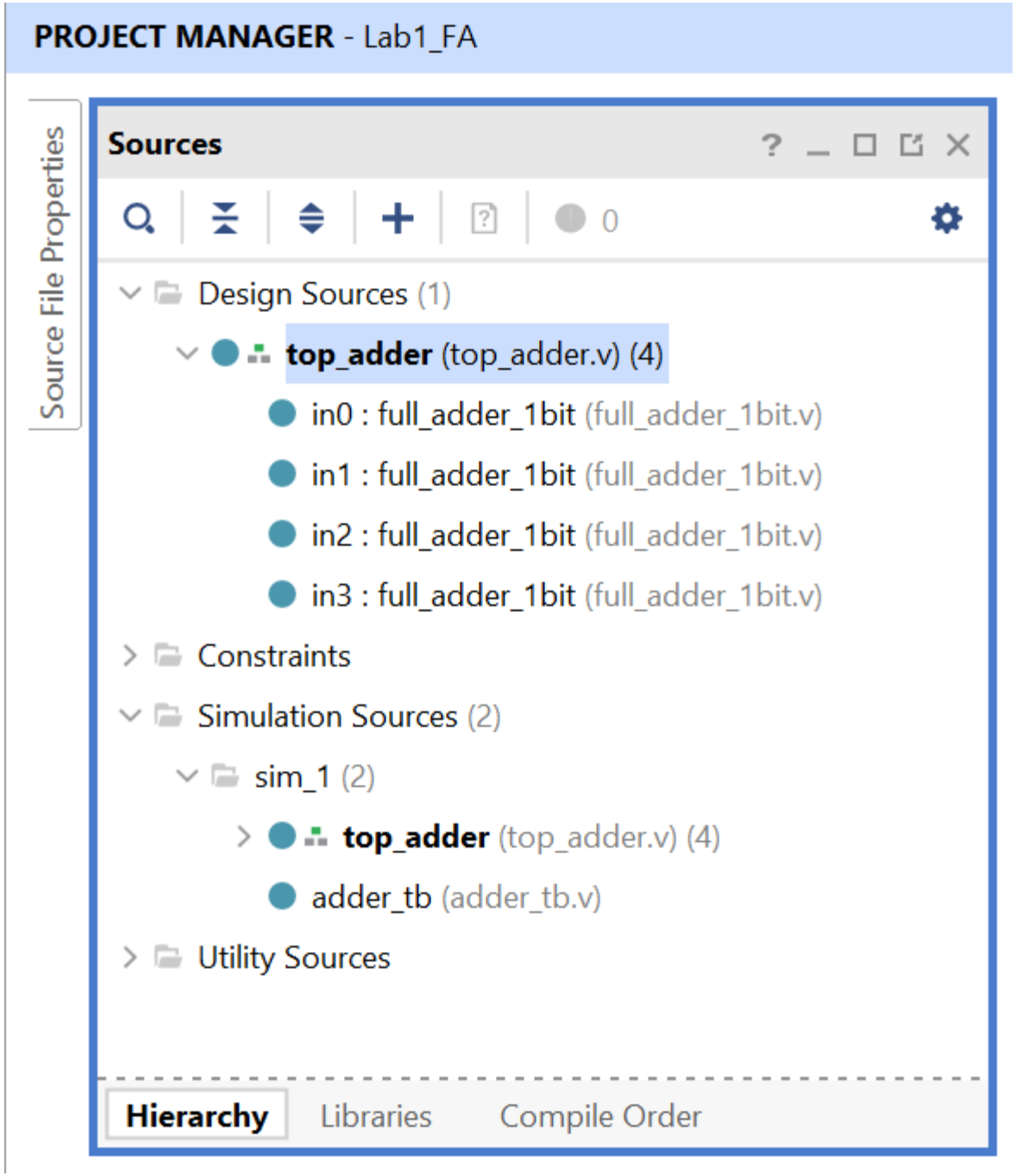

Add Sources
VIVADO
HLx Editions
Add Sources
This guides you through the process of adding and creating sources for your project
Add or create constraints
Add or create design sources
Add or create simulation sources


Create Source File
Create a new source file and add it to your project.
File type:
Verilog
File name:
adder_tb
File location:
<Local to Project>
OK
Cancel

Testbench


Create Source File
Create a new source file and add it to your project.
File type:
Verilog
File name:
adder_tb
<Local to Project>
OK
Cancel
Module Definition
Module name:
I/O Port Definitio
Define Module
The module definition has not been changed.
Are you sure you want to use these values?
Yes
No
Port Name
input
0
0
OK
Cancel

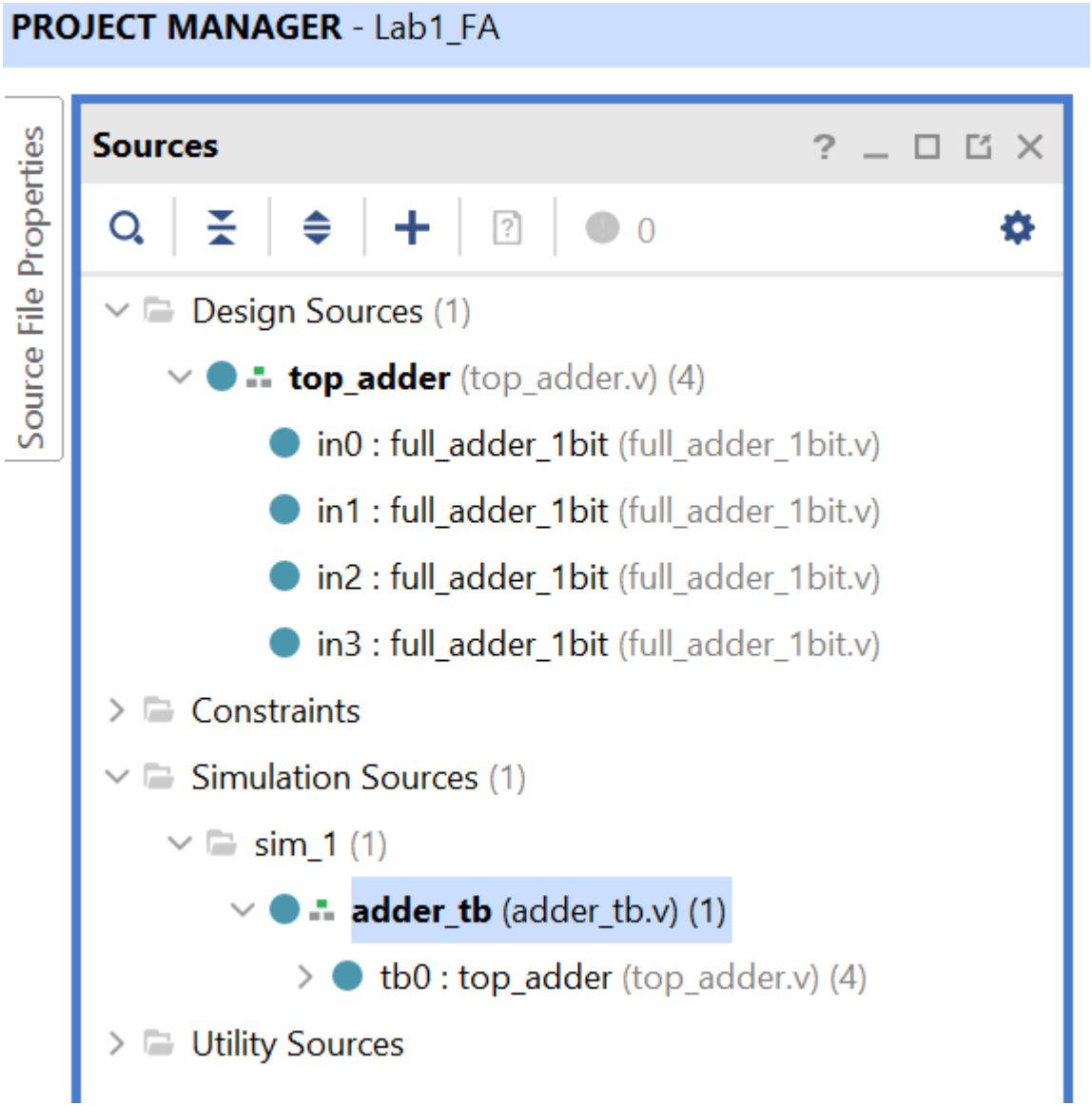
Testbench


PROJECT MANAGER - Lab1_FA
Source File Properties
Sources
0
Design Sources (1)
top_adder (top_adder.v) (4)
in0 : full_adder_1bit (full_adder_1bit.v)
in1 : full_adder_1bit (full_adder_1bit.v)
in2 : full_adder_1bit (full_adder_1bit.v)
in3 : full_adder_1bit (full_adder_1bit.v)
Constraints
Simulation Sources (2)
sim_1 (2)
top_adder (top_adder.v) (4)
adder_tb (adder_tb.v)
Utility Sources
Hierarchy
Libraries
Compile Order

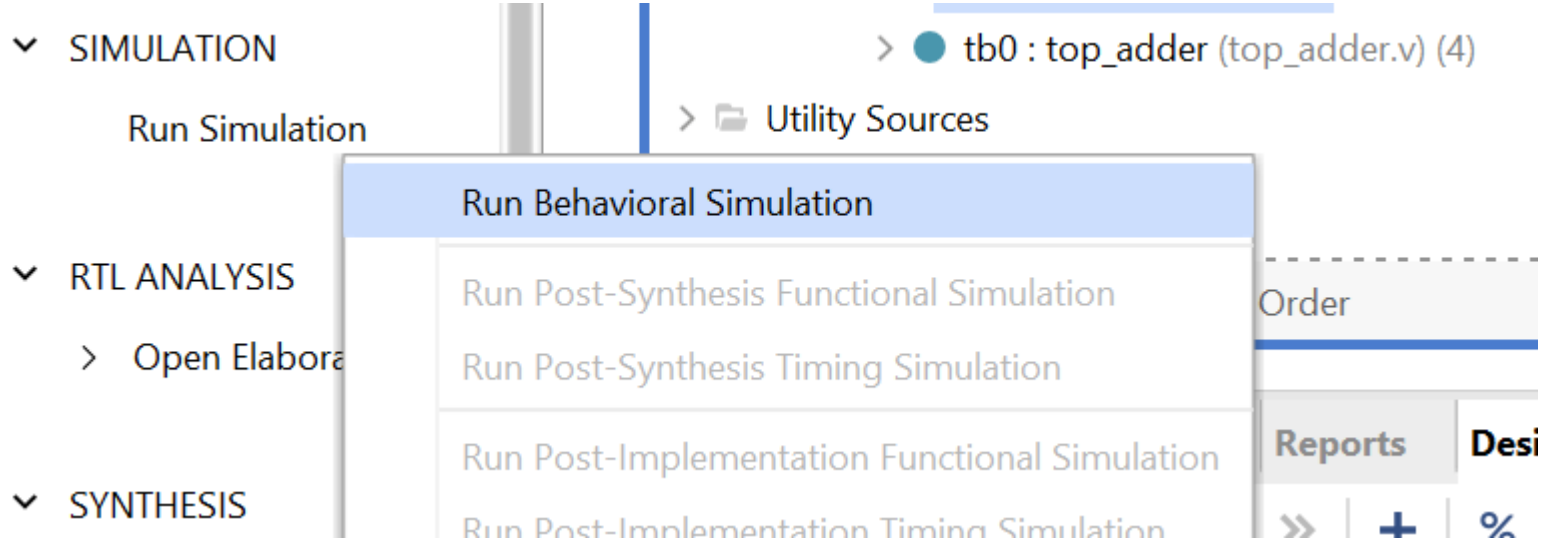
Testbench

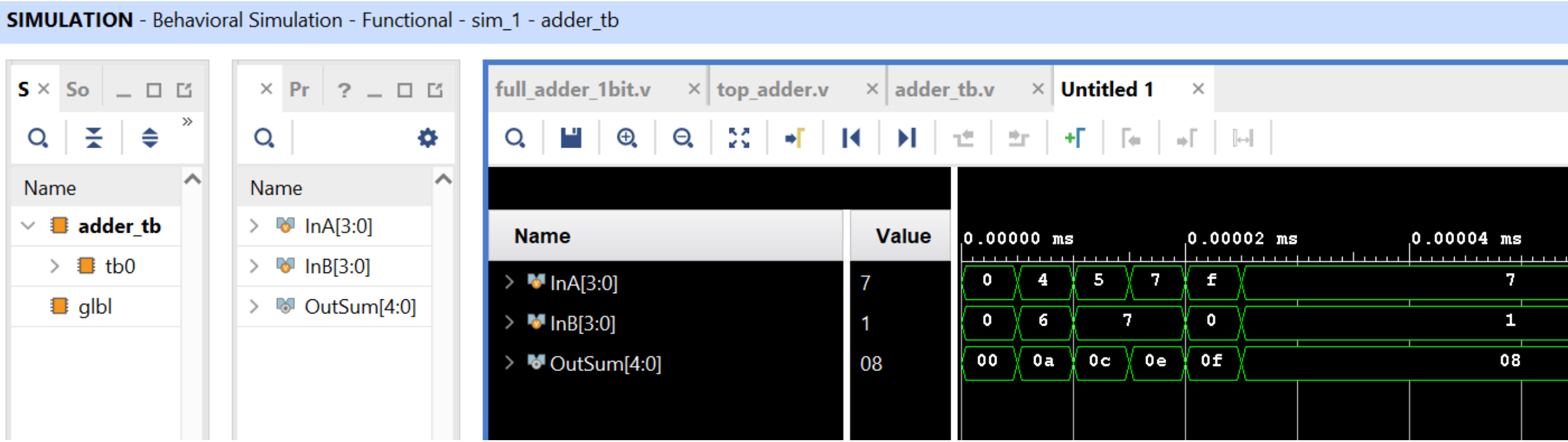
```
module adder_tb(

    );
     reg [3:0] InA ,InB;
    wire [4:0] OutSum;

    top_adder tb0(.InA(InA) , .InB(InB) , .OutSum(OutSum));


    initial begin
        InA = 4'b0000;     InB = 4'b0000;
        #5 InA = 4'b0100 ; InB = 4'b0110;
        #5 InA = 4'b0101 ; InB = 4'b0111;
        #5 InA = 4'b0111 ; InB = 4'b0111;
        #5 InA = 4'b1111 ; InB = 4'b0000;
        #5 InA = 4'b0111 ; InB = 4'b0001;
    end
endmodule
```

Testbench

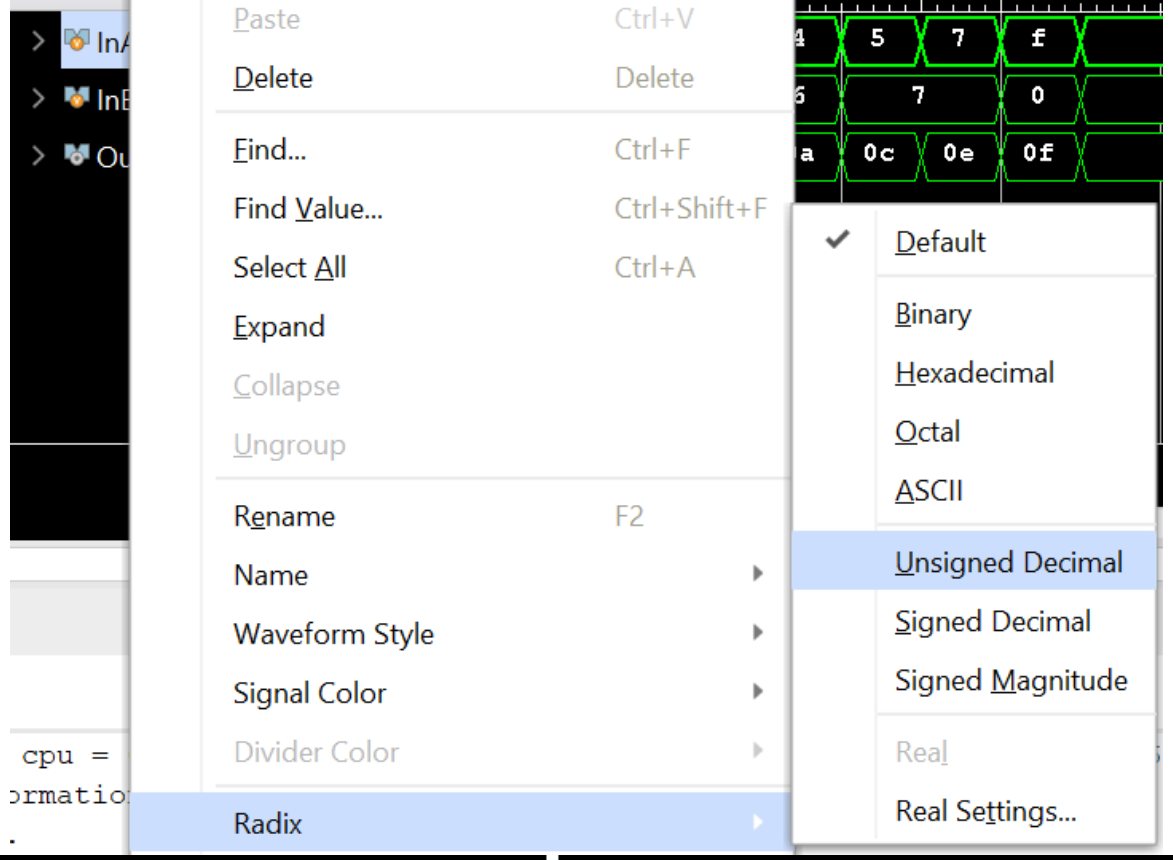

PROJECT MANAGER - Lab1_FA
Source File Properties
Sources
0
Design Sources (1)
top_adder (top_adder.v) (4)
in0 : full_adder_1bit (full_adder_1bit.v)
in1 : full_adder_1bit (full_adder_1bit.v)
in2 : full_adder_1bit (full_adder_1bit.v)
in3 : full_adder_1bit (full_adder_1bit.v)
Constraints
Simulation Sources (1)
sim_1 (1)
adder_tb (adder_tb.v) (1)
tb0 : top_adder (top_adder.v) (4)
Utility Sources

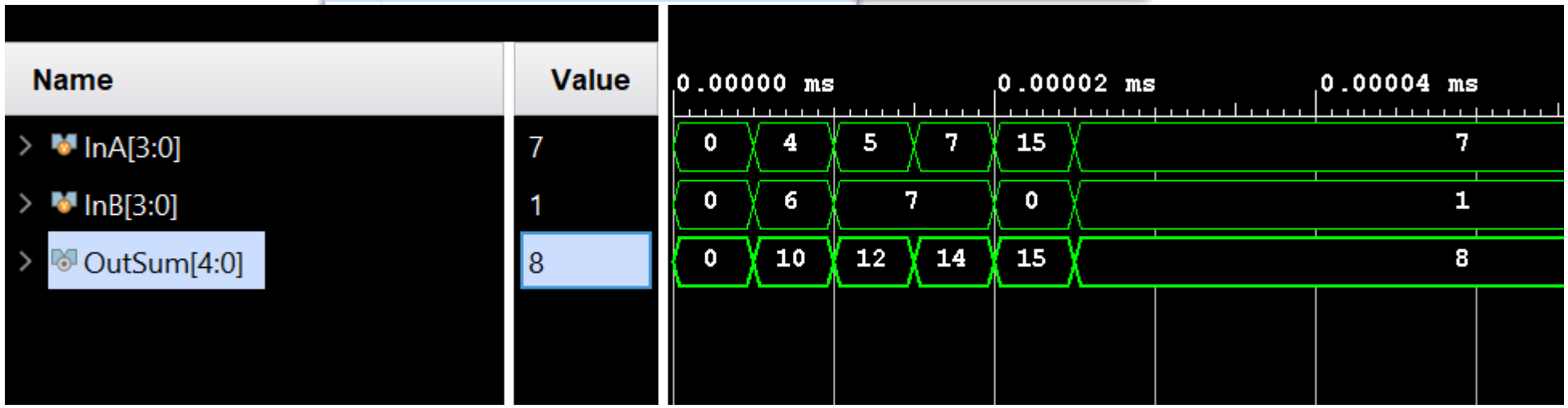
Testbench


SIMULATION
Run Simulation
RTL ANALYSIS
Open Elabora
SYNTHESIS
tb0 : top_adder (top_adder.v) (4)
Utility Sources
Run Behavioral Simulation
Run Post-Synthesis Functional Simulation
Run Post-Synthesis Timing Simulation
Run Post-Implementation Functional Simulation
Order
Reports
Desi

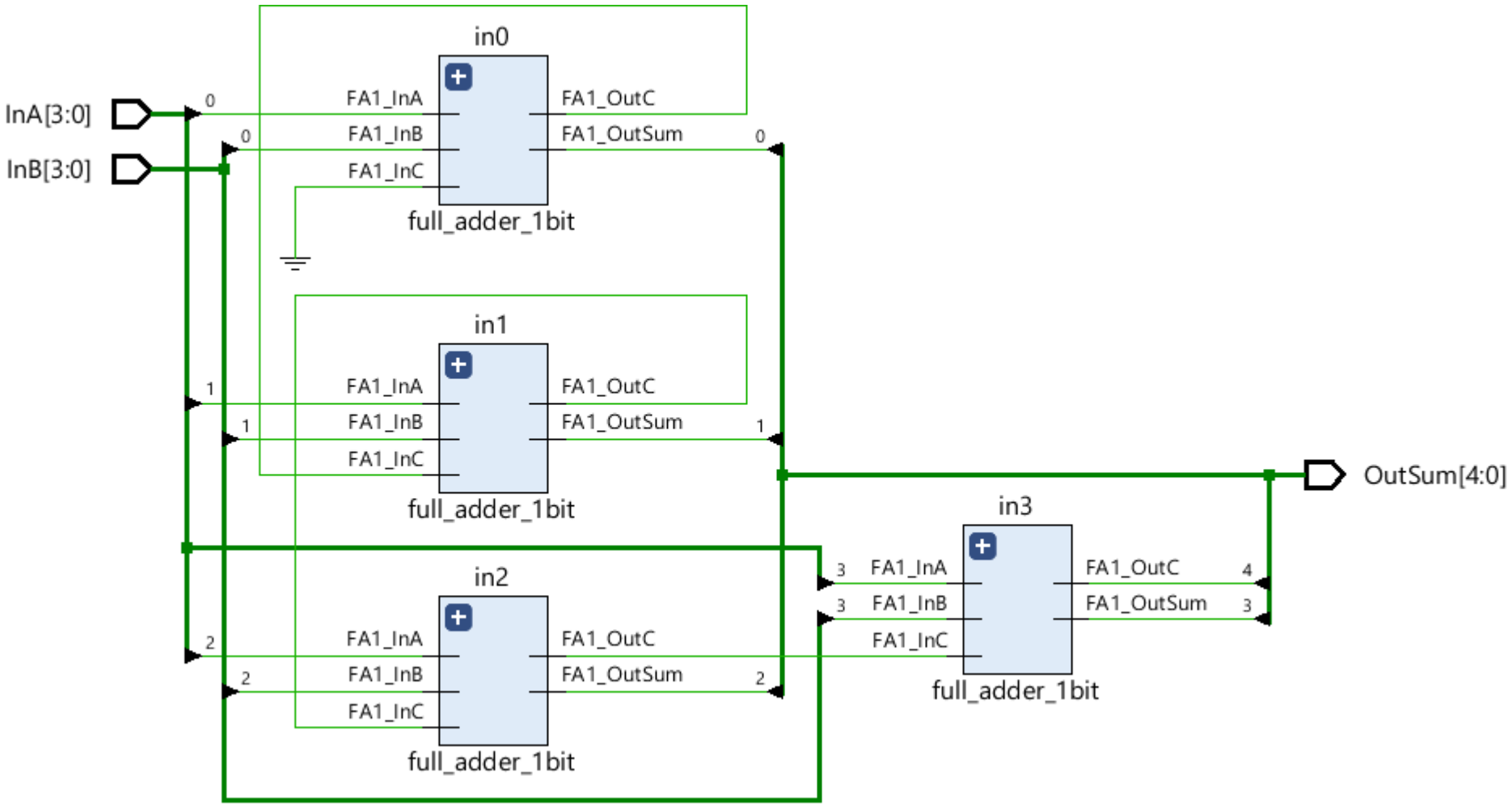

SIMULATION - Behavioral Simulation - Functional - sim_1 - adder_tb
Name
adder_tb
tb0
glbl
Name
InA[3:0]
InB[3:0]
OutSum[4:0]
full_adder_1bit.v
top_adder.v
adder_tb.v
Untitled 1
Name
Value
InA[3:0] 7
InB[3:0] 1
OutSum[4:0] 08
0.00000 ms
0.00002 ms
0.00004 ms
0 4 5 7 f 7
0 6 7 0 1
00 0a 0c 0e 0f 08

Testbench

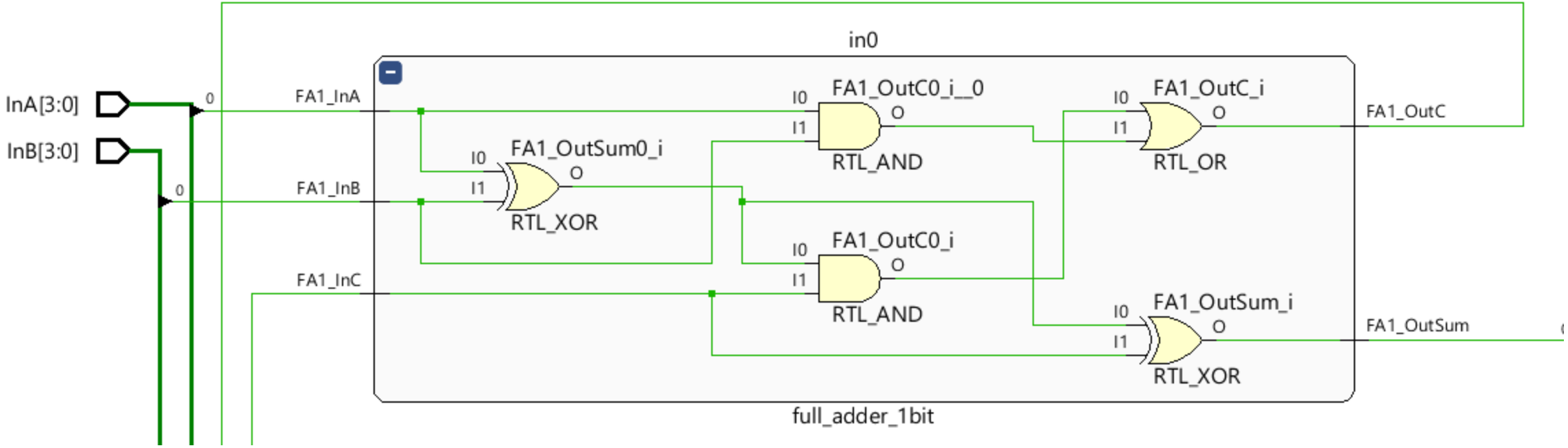

Paste Ctrl+V
Delete Delete
Find... Ctrl+F
Find Value... Ctrl+Shift+F
Select All Ctrl+A
Expand
Collapse
Ungroup
Rename F2
Name
Waveform Style
Signal Color
Divider Color
Radix
Default
Binary
Hexadecimal
Octal
ASCII
Unsigned Decimal
Signed Decimal
Signed Magnitude
Real
Real Settings...


Name
Value
InA[3:0]
7
InB[3:0]
1
OutSum[4:0]
8
0.00000 ms
0.00002 ms
0.00004 ms

Elaborated Design

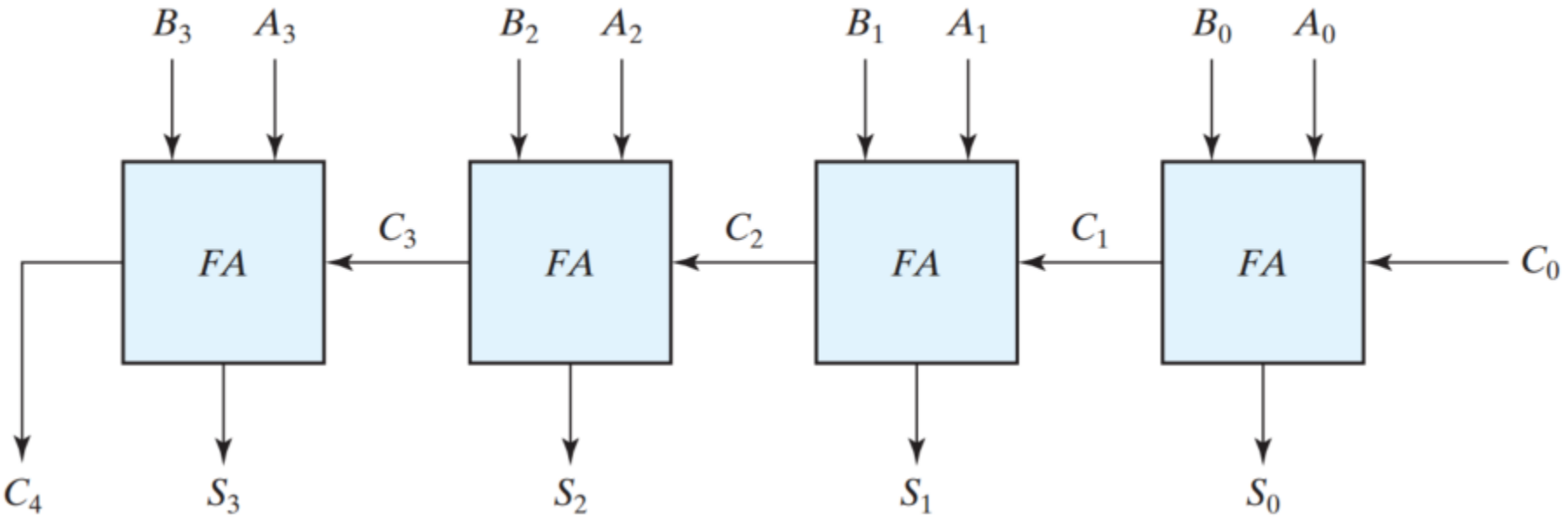


Elaborated Design



Homework

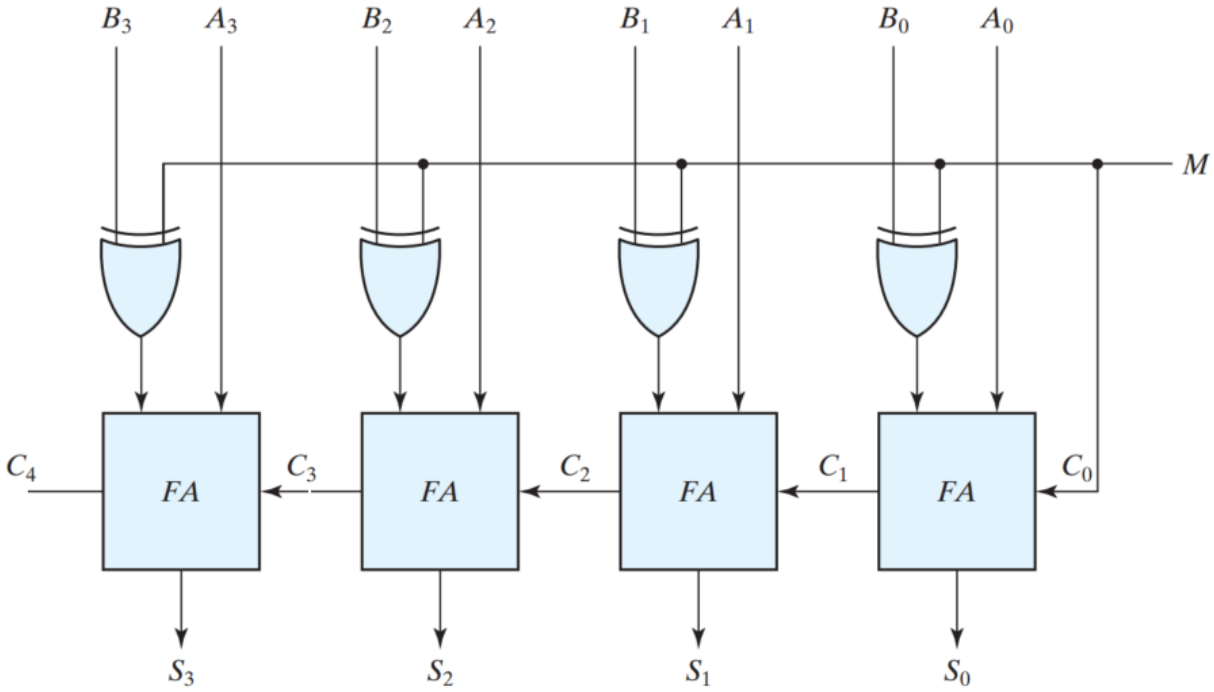
Adders



- For adder with unsigned number inputs, add output flag which goes high when overflow occurs
- When two numbers with *n* digits each are added and the sum is a number occupying *n* + 1 digits, we say that an overflow occurred.
- Overflow is a problem in digital computers because the number of bits that hold the number is finite and a result that contains *n* + 1 bits cannot be accommodated by an *n* -bit word.
- Hint: When two unsigned numbers are added, an overflow is detected from the end carry out of the most significant position.

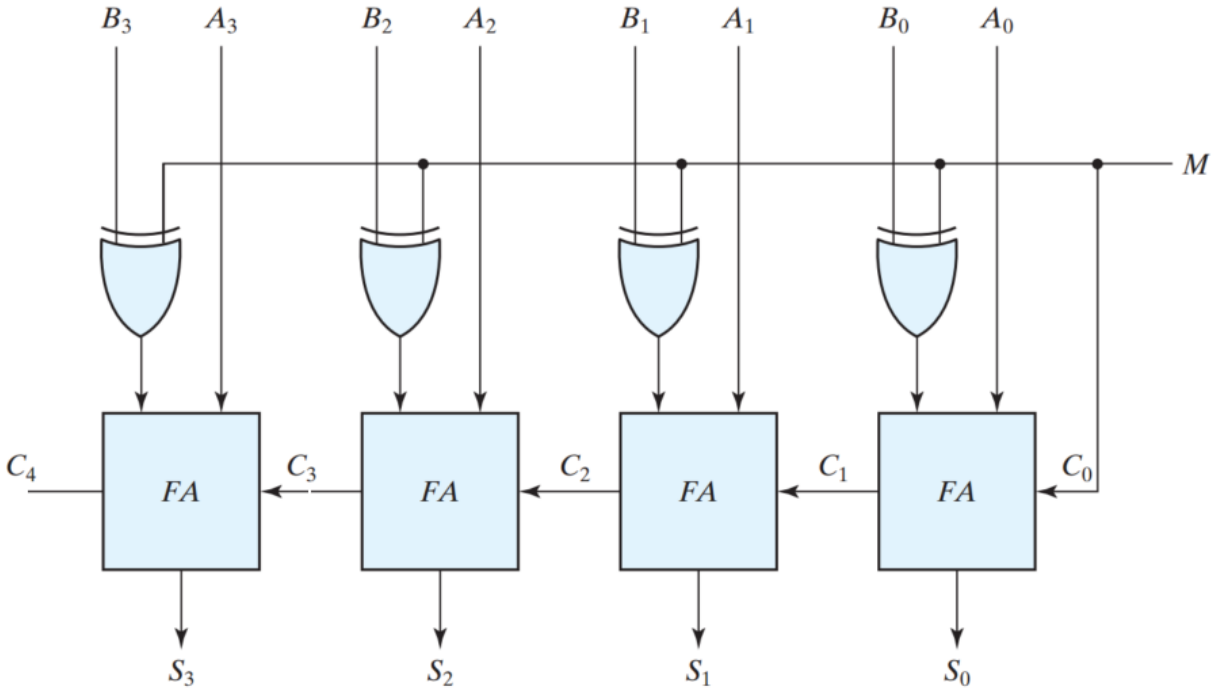
Self Study

Adder/Subtractor



- For adder/subtractor with signed number inputs, add three independent output flags 1) First output flag goes high when overflow occurs, 2) Second output flag goes high when sum is negative, and 3) Third output flag goes high when sum is zero

Overflow



$$\begin{array}{r} (+7) \\ +\ (-2) \\ \hline (+5) \end{array} \qquad \begin{array}{r} 0\,1\,1\,1 \\ +\ 1\,1\,1\,0 \\ \hline 1\,0\,1\,0\,1 \end{array}$$

$c_4 = 1$
$c_3 = 1$

$$\begin{array}{r} (-7) \\ +\ (+2) \\ \hline (-5) \end{array} \qquad \begin{array}{r} 1\,0\,0\,1 \\ +\ 0\,0\,1\,0 \\ \hline 1\,0\,1\,1 \end{array}$$

$c_4 = 0$
$c_3 = 0$

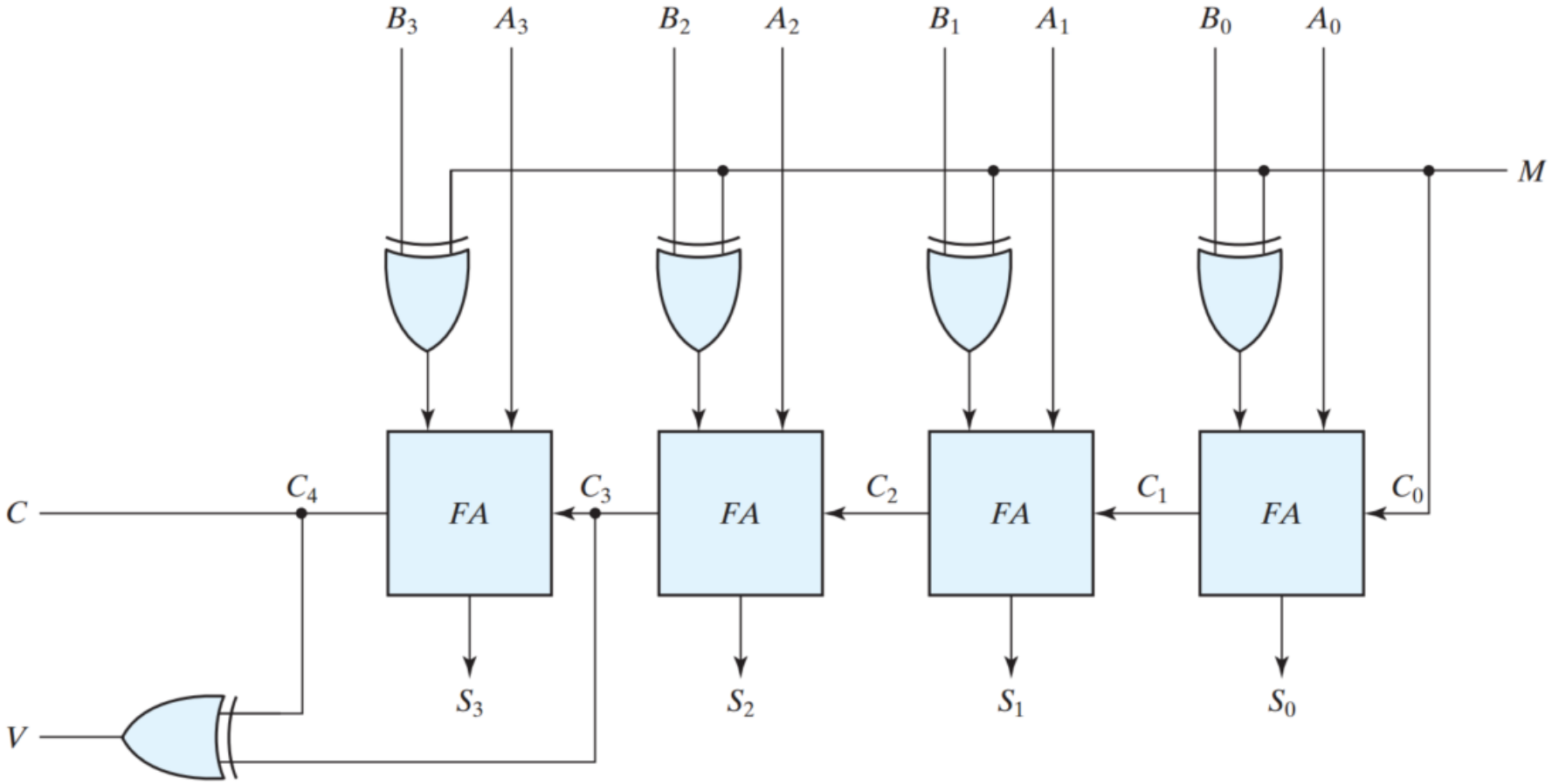
$$\begin{array}{r} (-7) \\ +\ (-2) \\ \hline (-9) \end{array} \qquad \begin{array}{r} 1\,0\,0\,1 \\ +\ 1\,1\,1\,0 \\ \hline 1\,0\,1\,1\,1 \end{array}$$

$c_4 = 1$
$c_3 = 0$

$$\begin{array}{r} (+7) \\ +\ (+2) \\ \hline (+9) \end{array} \qquad \begin{array}{r} 0\,1\,1\,1 \\ +\ 0\,0\,1\,0 \\ \hline 1\,0\,0\,1 \end{array}$$

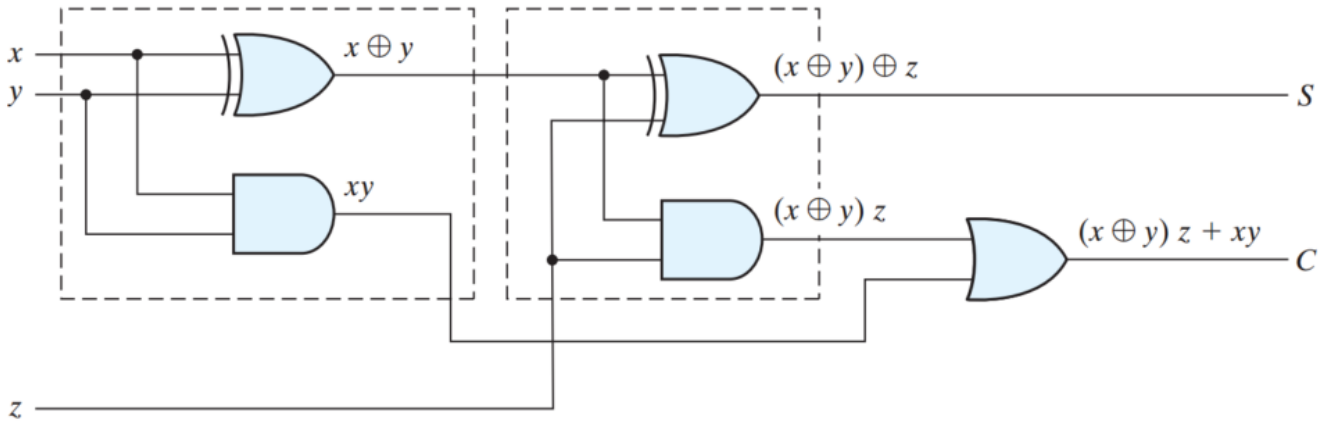
$c_4 = 0$
$c_3 = 1$

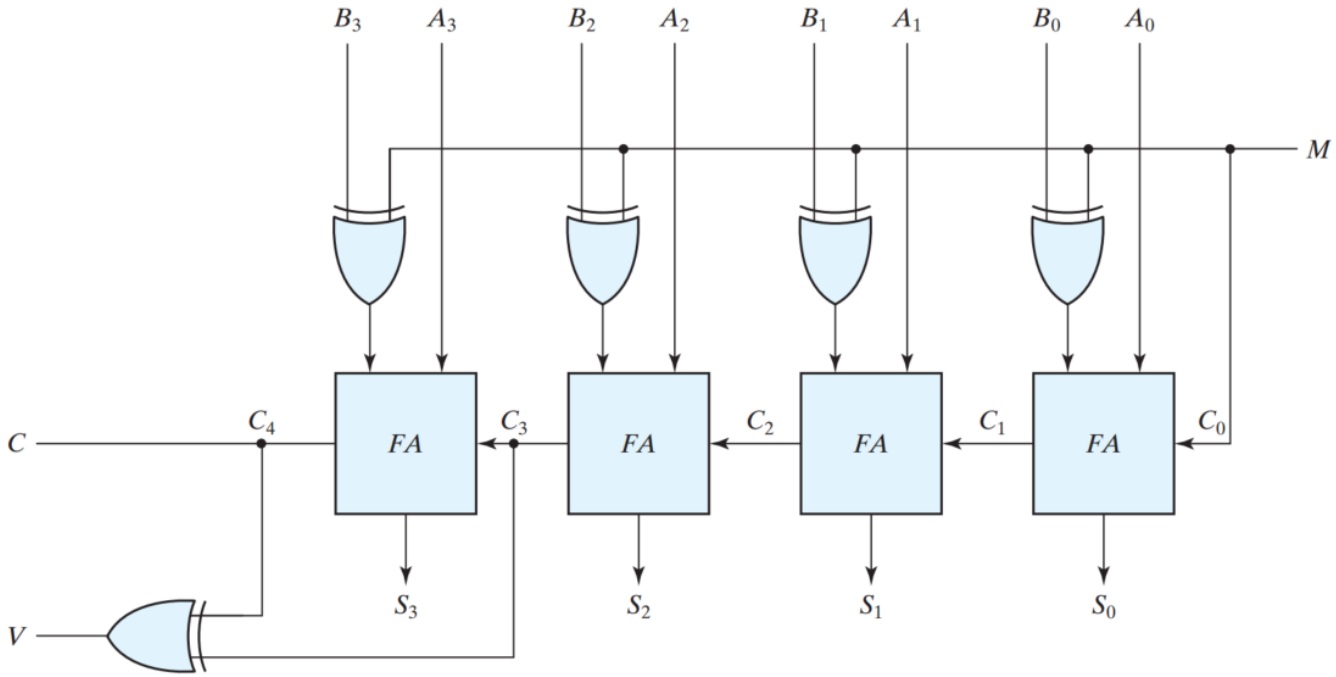
Overflow



Performance

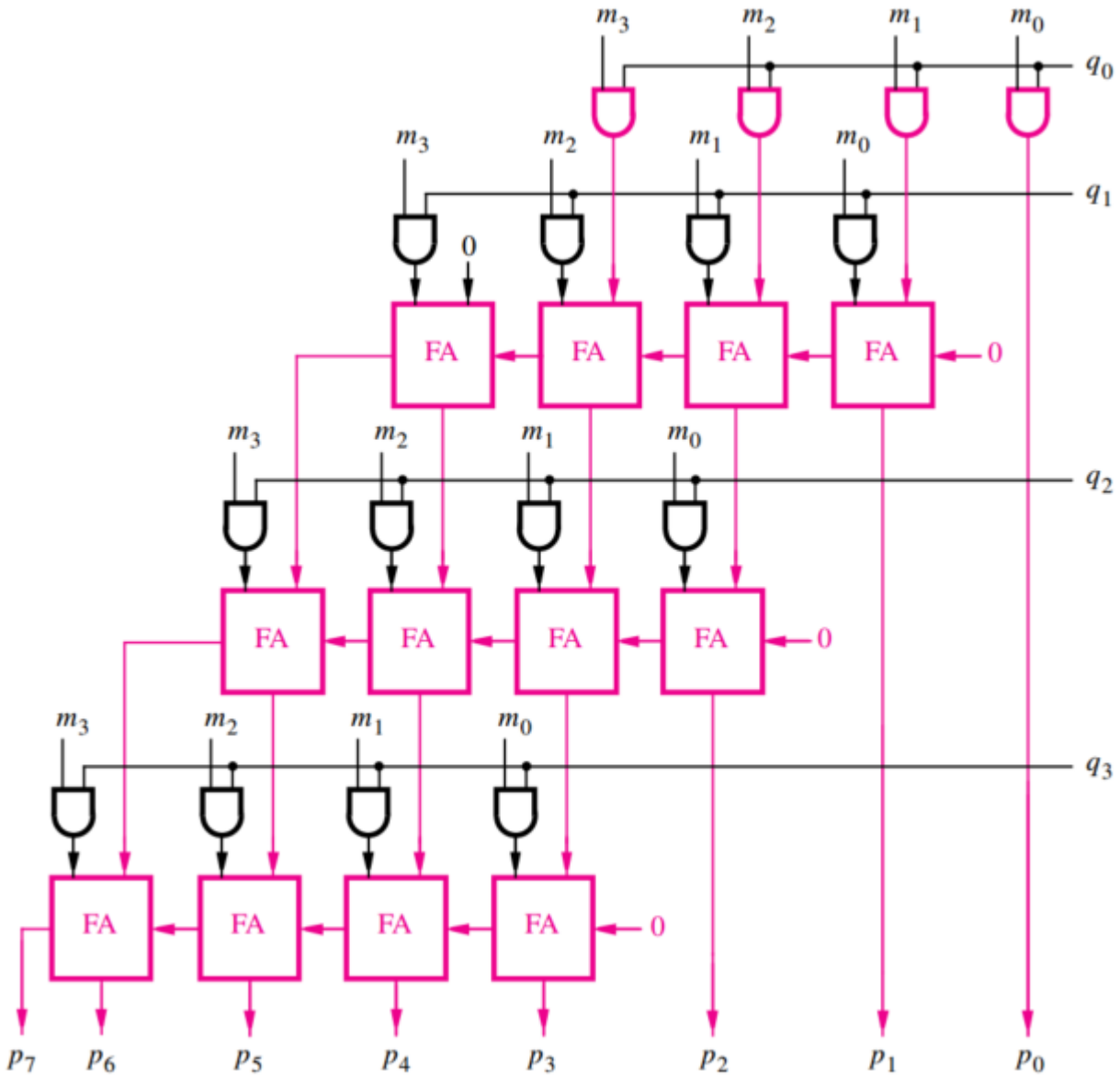
- The longest propagation delay time in an adder is the time it takes the carry to propagate through the full adders.
- Inputs *A*3 and*B*3 are available as soon as input signals are applied to the adder. However, input carry *C*3 does not settle to its final value until *C*2 is available from the previous stage. Similarly, *C*2 has to wait for *C*1 and so on down to *C*0.





Multiplication

					m_3	m_2	m_1	m_0
				$\times$	q_3	q_2	q_1	q_0
Partial product 0					m_3q_0	m_2q_0	m_1q_0	m_0q_0
			$+$	m_3q_1	m_2q_1	m_1q_1	m_0q_1	
Partial product 1		$PP1_5$	$PP1_4$	$PP1_3$	$PP1_2$	$PP1_1$		
	$+$	m_3q_2	m_2q_2	m_1q_2	m_0q_2			
Partial product 2	$PP2_6$	$PP2_5$	$PP2_4$	$PP2_3$	$PP2_2$			
$+$	m_3q_3	m_2q_3	m_1q_3	m_0q_3				
Product P	p_7	p_6	p_5	p_4	p_3	p_2	p_1	p_0



ELD Lab 2
Design of 8-bit Counter

Objective

- Design 8-bit Up counter using behavioral modelling
- For counter to increment every second, design 1 Hz clock from input 100 MHz clock using clock divider
- ~~Verify the counter on hardware using virtual input and output (VIO).~~
- **Lab Homework:** Design up/down counter with maximum count of 85

Verilog Revision

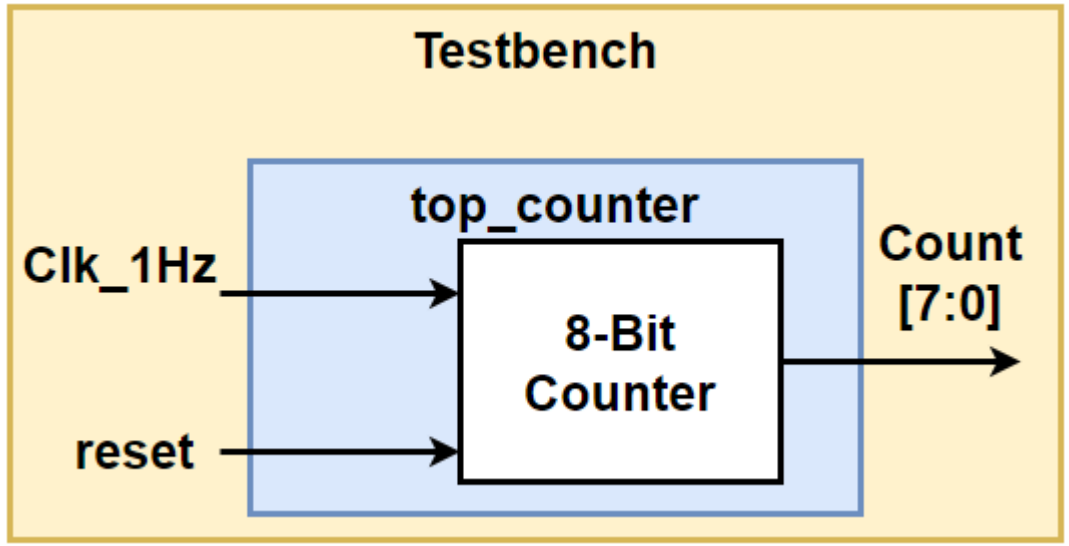
Lab

Proposed Approach

8-bit Counter

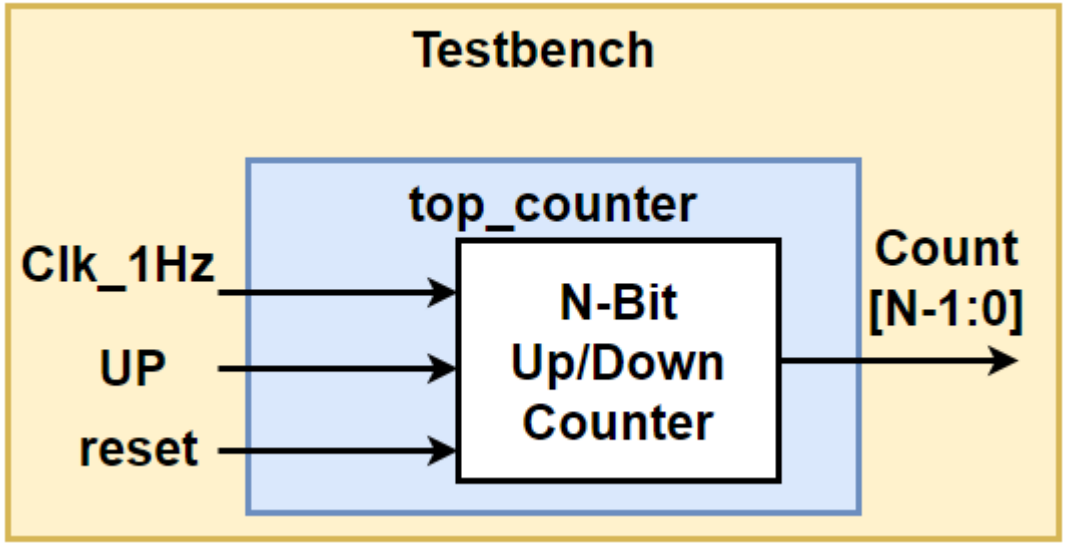
```
module Counter_8bit(
    input Clk_1Hz,
    input reset,
    output [7:0] Count
    );
    reg [7:0] Count_reg=0;
    reg [7:0] Count_next;
    always@(posedge Clk_1Hz or posedge reset)
    begin
        if(reset)
            Count_reg <= 0;
        else
            Count_reg <= Count_next;
    end
    always@(*)
    begin
        Count_next = Count_reg + 1;
    end
    assign Count = Count_reg;
endmodule
```

Testbench



Homework

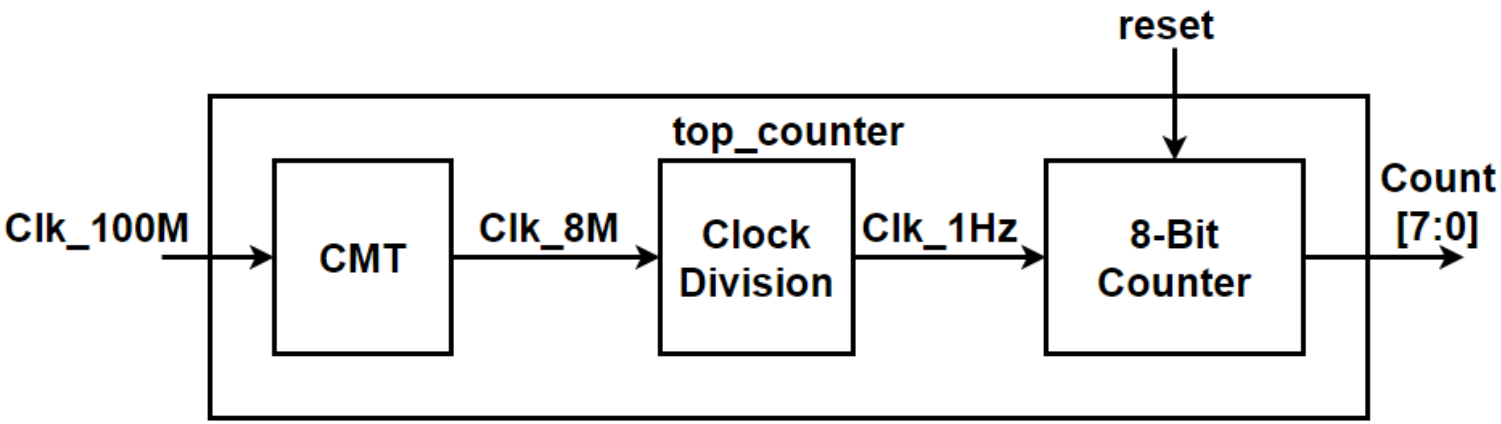
- Design up/down counter with maximum count of 85
- Write Verilog code and testbench to verify all functionalities of the counter



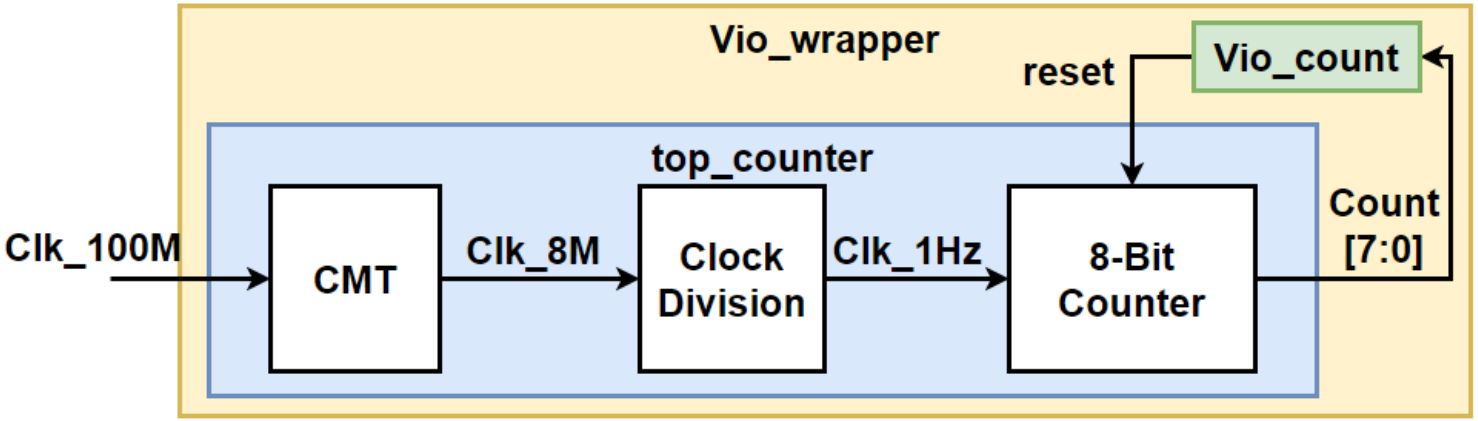
Counter on Hardware (Later)

Counter with Clock Division (Later)

```
module top_counter(
  input Clk_100M,
  input reset,
  output [7:0] Count
  );
  wire Clk_8M;
  clk_div_cmt cd
 (
  // Clock out ports
  .Clk_8M(Clk_8M),     // output Clk_8M
 // Clock in ports
  .Clk_100M(Clk_100M));

   wire Clk_1Hz;
  // This modules divide the input clock by 2^(COUNT_DIV_FACTOR+1)
  // (8x10^6)/2^(23) -> 1 Hz
  clk_div_rtl #(.COUNT_DIV_FACTOR(22)) clk_div_rtl1(.reset(reset),.clk_in(Clk_8M),.clk_out(Clk_1Hz))
  Counter_8bit Cn(.Clk_1Hz(Clk_1Hz), .reset(reset),.Count(Count));

endmodule
```



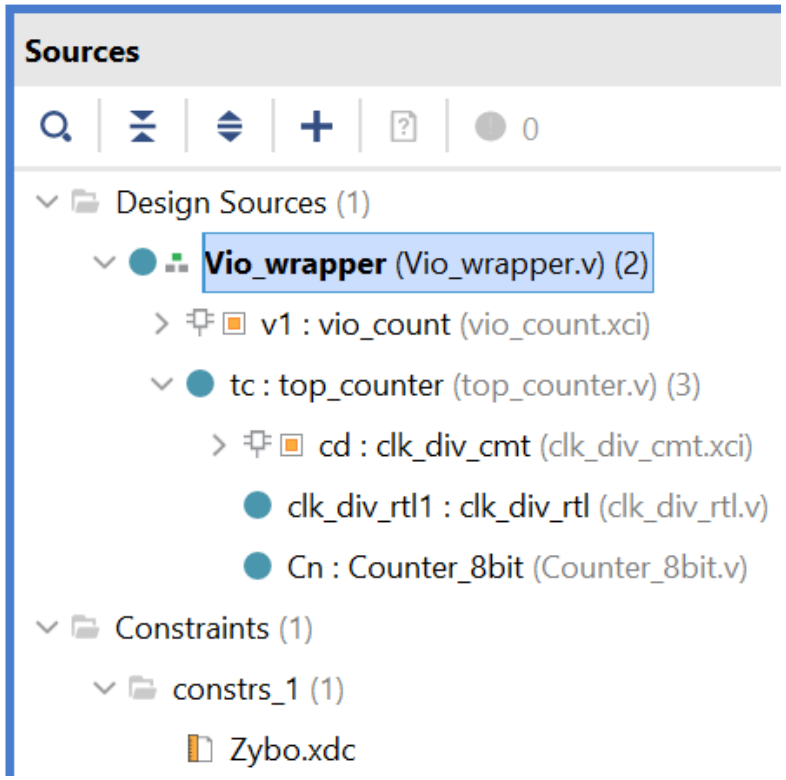
VIO Wrapper (Later)




```
module Vio_wrapper(
    input Clk_100M
    );

    wire reset;
    wire [7:0] Count;
    vio_count v1 (
  .clk(Clk_100M),                    // input wire clk
  .probe_in0(Count),     // input wire [7 : 0] probe_in0
  .probe_out0(reset)  // output wire [0 : 0] probe_out0
);


  top_counter tc(.Clk_100M(Clk_100M),.reset(reset),.Count(Count));

endmodule
```

Demo (Later)

- Add XDC file and generate bitstream
- Verify the functionality using VIO



ELD Lab 2
Design of 8-bit Counter

Academic Dishonesty

Quiz 1

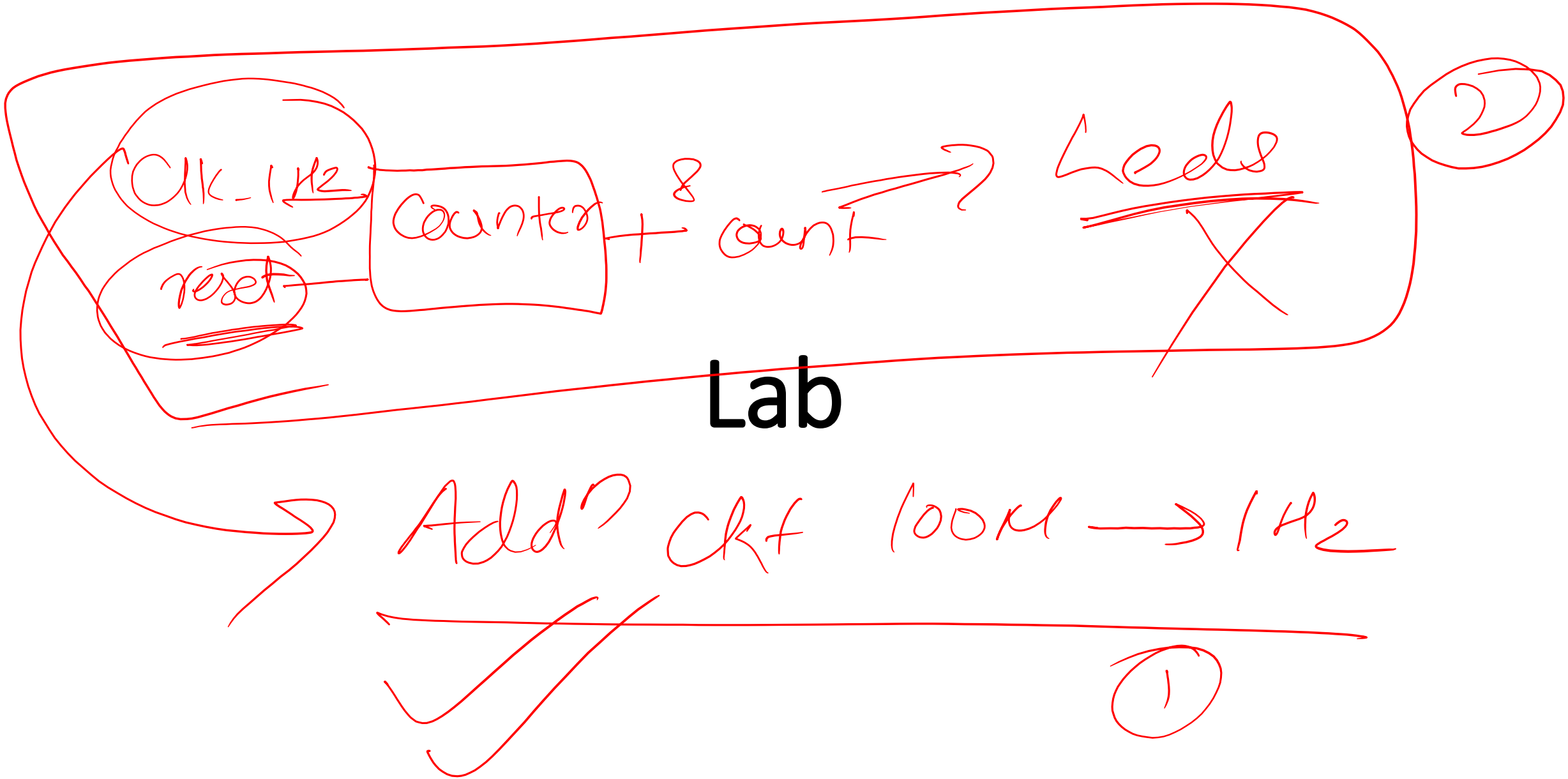
Objective

TB UP 85

- Design 8-bit Up counter using behavioral modelling
- For counter to increment every second, design 1 Hz clock from input 100 MHz clock using clock divider
- ~~Verify the counter on hardware using virtual input and output (VIO).~~
- **Lab Homework:** Design up/down counter with maximum count of 85

Hardware 10 12

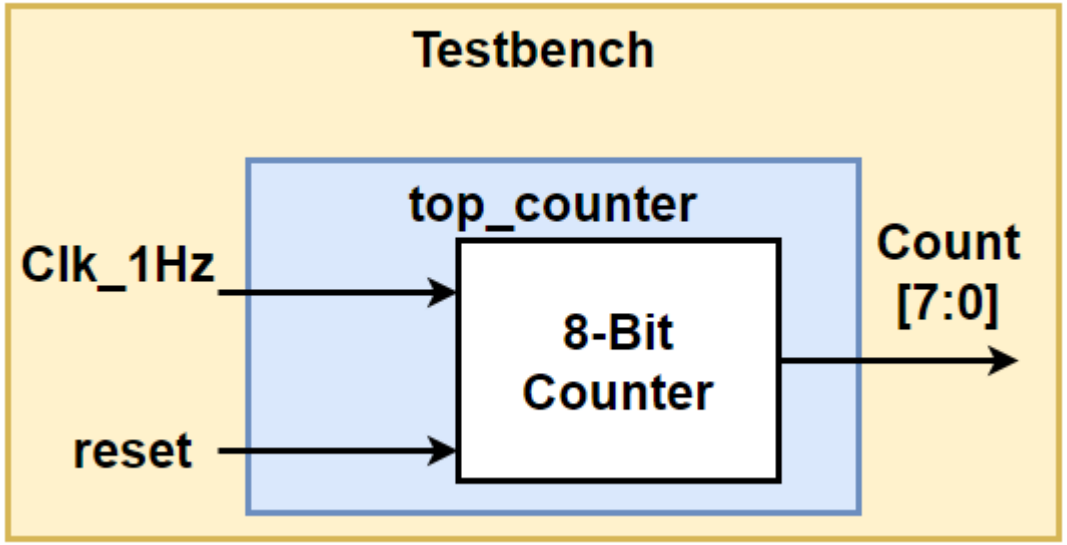
130 → 60

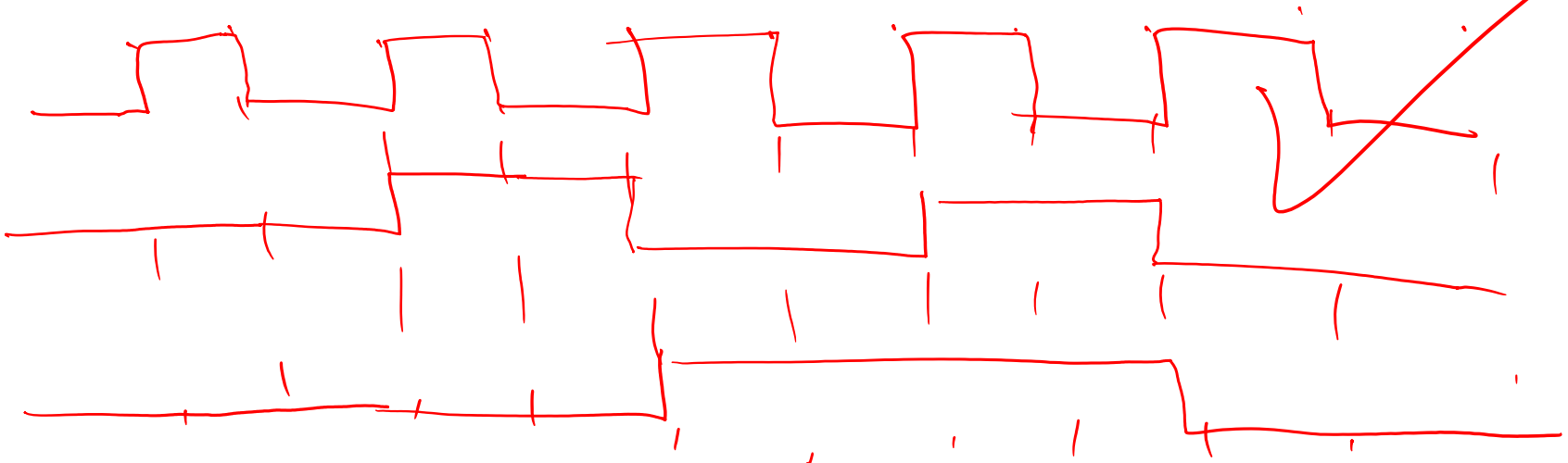


Lab

Proposed Approach

Freq. Divider





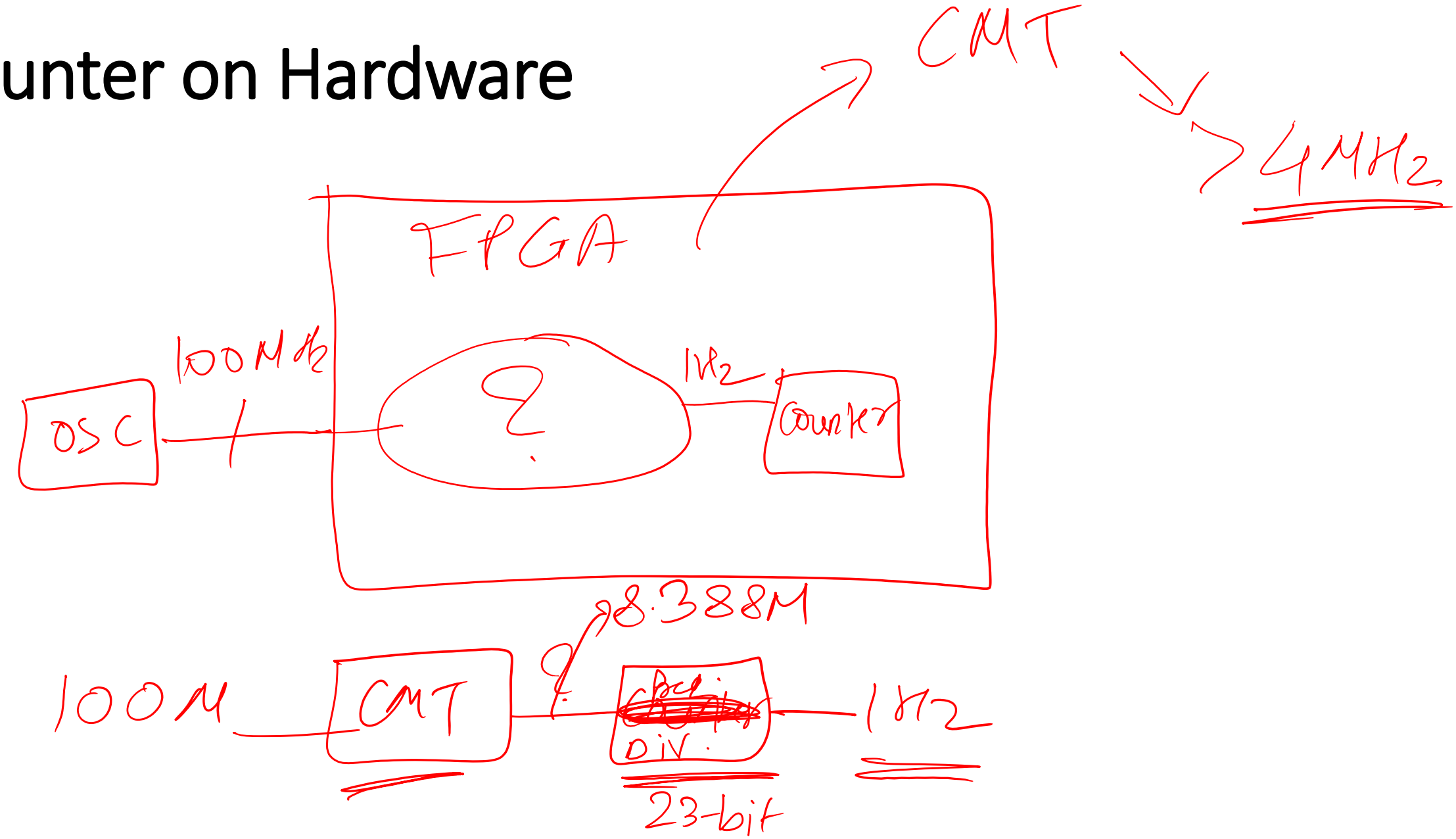
0 0 0
0 0 1
0 1 0
0 1 1
1 0 0
1 0 1
1 1 0
1 1 1

$$\frac{f}{2^x}$$

8-bit Counter

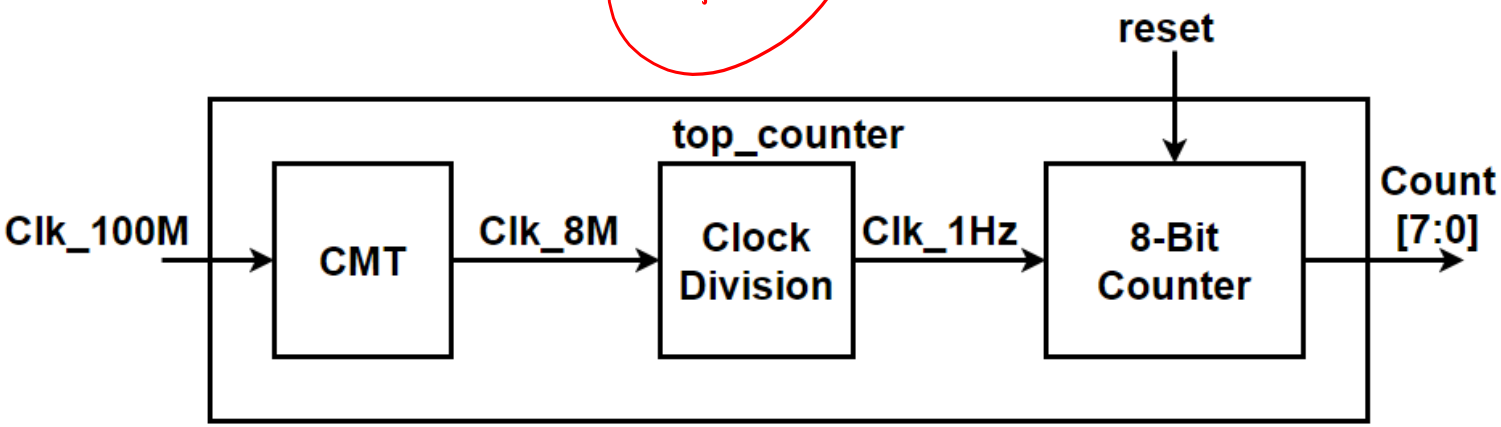
```
module Counter_8bit(
    input Clk_1Hz,
    input reset,
    output [7:0] Count
    );
    reg [7:0] Count_reg=0;
    reg [7:0] Count_next;
    always@(posedge Clk_1Hz or posedge reset)
    begin
        if(reset)
            Count_reg <= 0;
        else
            Count_reg <= Count_next;
    end
    always@(*)
    begin
        Count_next = Count_reg + 1;
    end
    assign Count = Count_reg;
endmodule
```

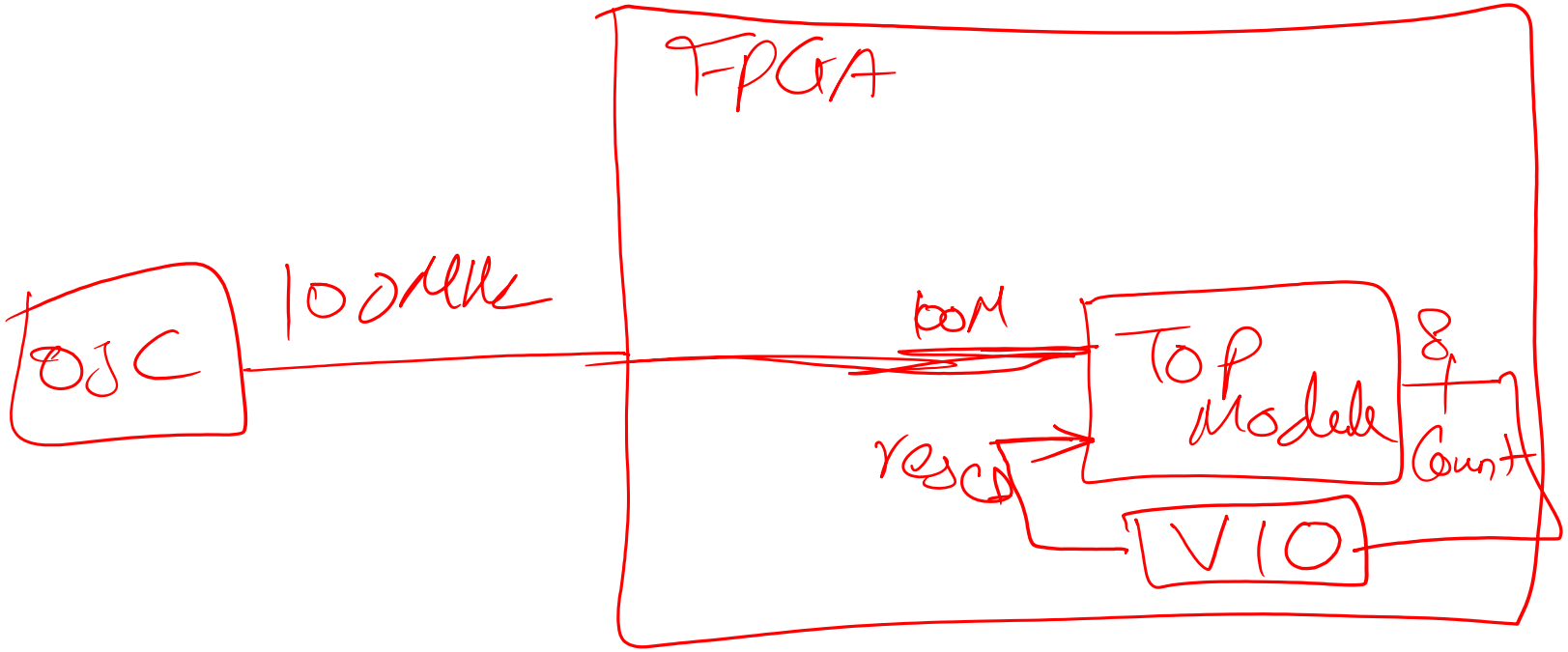
Counter on Hardware



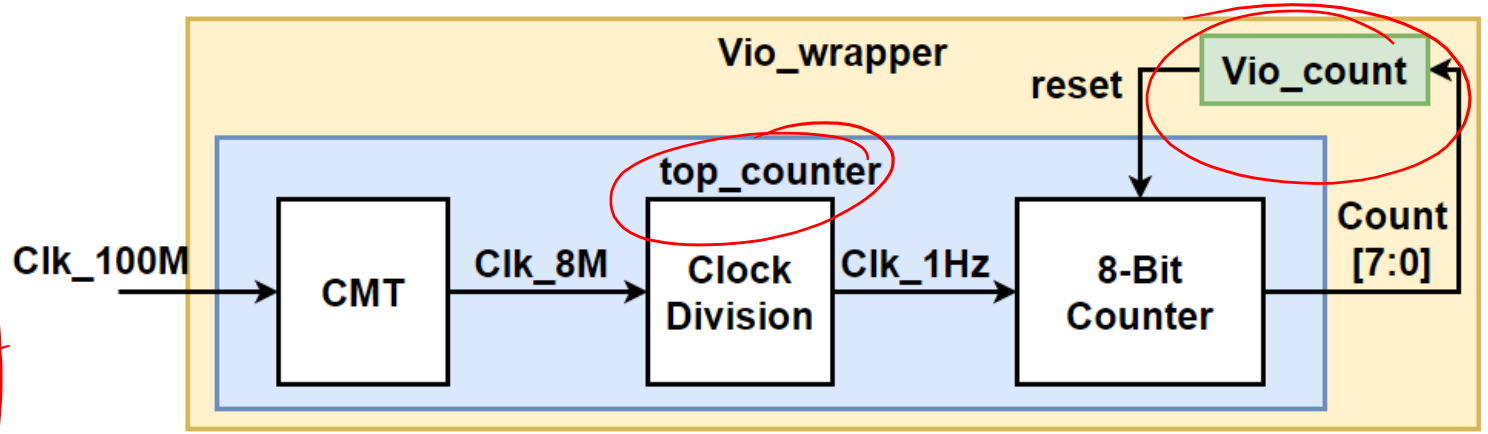
Counter with Clock Division

```
module top_counter(
    input Clk_100M,
    input reset,
    output [7:0] Count
    );
    wire Clk_8M;
    clk_div_cmt cd
   (
    // Clock out ports
    .Clk_8M(Clk_8M),     // output Clk_8M
   // Clock in ports
    .Clk_100M(Clk_100M));

     wire Clk_1Hz;
    // This modules divide the input clock by 2^(COUNT_DIV_FACTOR+1)
    // (8x10^6)/2^(23) -> 1 Hz
    clk_div_rtl #(.COUNT_DIV_FACTOR(22)) clk_div_rtl1(.reset(reset),.clk_in(Clk_8M),.clk_out(Clk_1Hz))
    Counter_8bit Cn(.Clk_1Hz(Clk_1Hz), .reset(reset),.Count(Count));

endmodule
```




FPGA
OSC
100MHz
100M
Top Module
8
Count
resc
VIO

VIO Wrapper




```
module Vio_wrapper(
    input Clk_100M
    );

    wire reset;
    wire [7:0] Count;
    vio_count v1 (
  .clk(Clk_100M),                    // input wire clk
  .probe_in0(Count),                 // input wire [7 : 0] probe_in0
  .probe_out0(reset)  // output wire [0 : 0] probe_out0
);


  top_counter tc(.Clk_100M(Clk_100M),.reset(reset),.Count(Count));

endmodule
```

Vio Wrapper

VIO

Top

How to input Clock?

Add Sources

VIVADO HLx Editions

Add Sources

This guides you through the process of adding and creating sources for your project

- (●) Add or create constraints
- ( ) Add or create design sources
- ( ) Add or create simulation sources

Specify constraint set: constrs_1 (active)

Constraint File	Location
Zed_cons.xdc	<Local to Project>

XDC File

- Zedboard

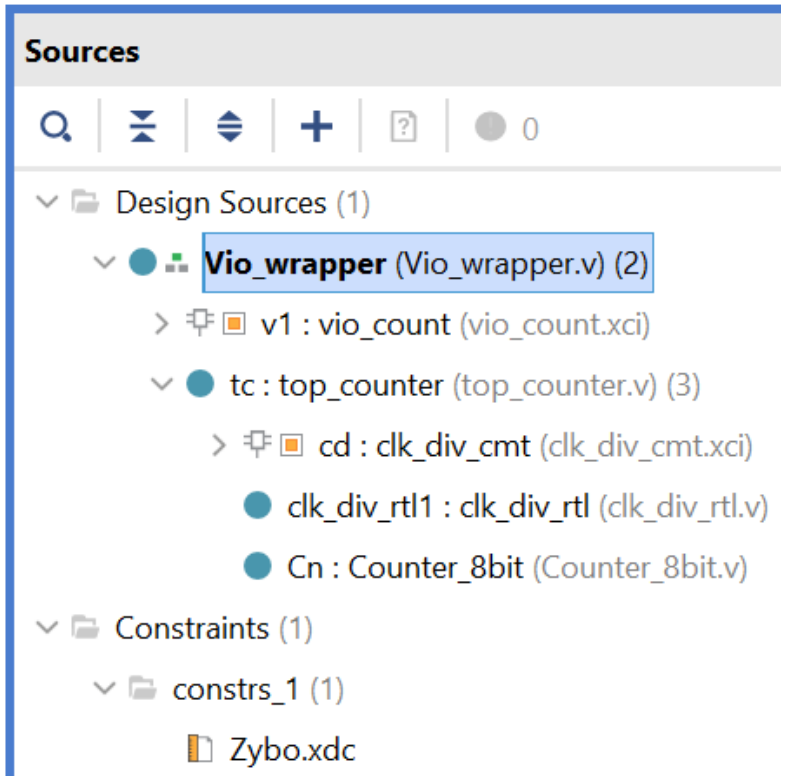
```
# ---------------------------------------------------------------------------
# Clock Source - Bank 13
# ---------------------------------------------------------------------------
set_property PACKAGE_PIN Y9 [get_ports {Clock}];  # "GCLK"
```

- Zybo

```
##Clock signal
set_property -dict { PACKAGE_PIN K17   IOSTANDARD LVCMOS33 } [get_ports { Clock }];
create_clock -add -name sys_clk_pin -period 8.00 -waveform {0 4} [get_ports { Clock }];
```

Demo

- Add XDC file and generate bitstream
- Verify the functionality using VIO



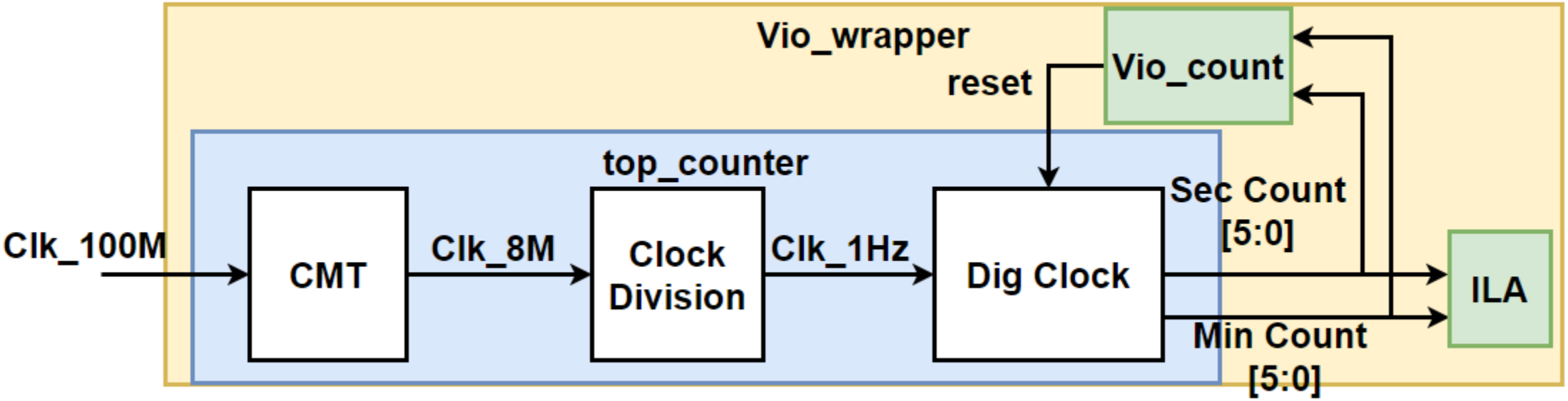
ELD Lab 4
Design of Digital Clock

Objective

- Design digital clock with Second and Minute Display using behavioral modelling
- Verify the circuit using virtual input and output (VIO) and Integrated Logic Analyzer.
- **Lab Homework:** Modify the Digital Clock by using CMT output of 16.777 MHz

Lab

Proposed Approach (Extension of Lab 3)



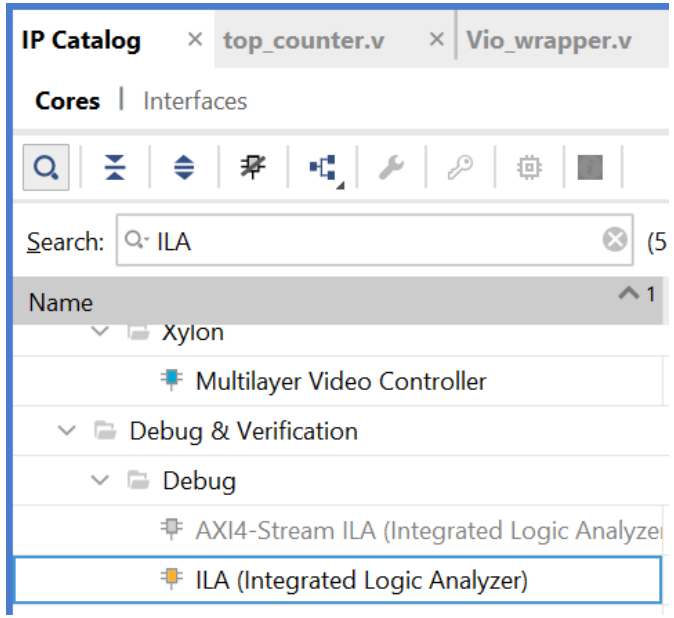
Digital Clock

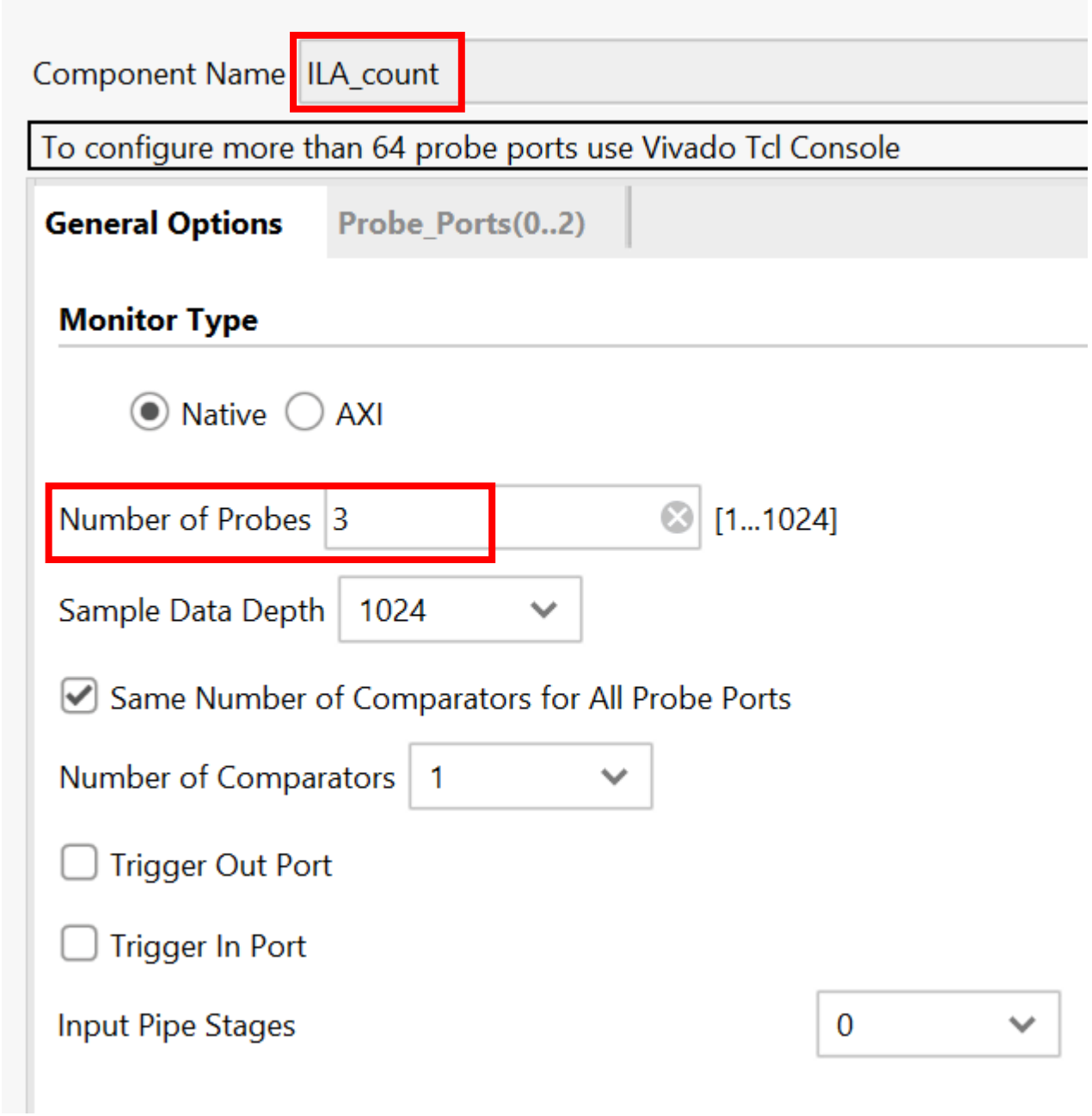
```
module Dig_clock(
    input Clk_1Hz,
    input reset,
    output [5:0] Sec_Count,
    output [5:0] Min_Count
    );
```

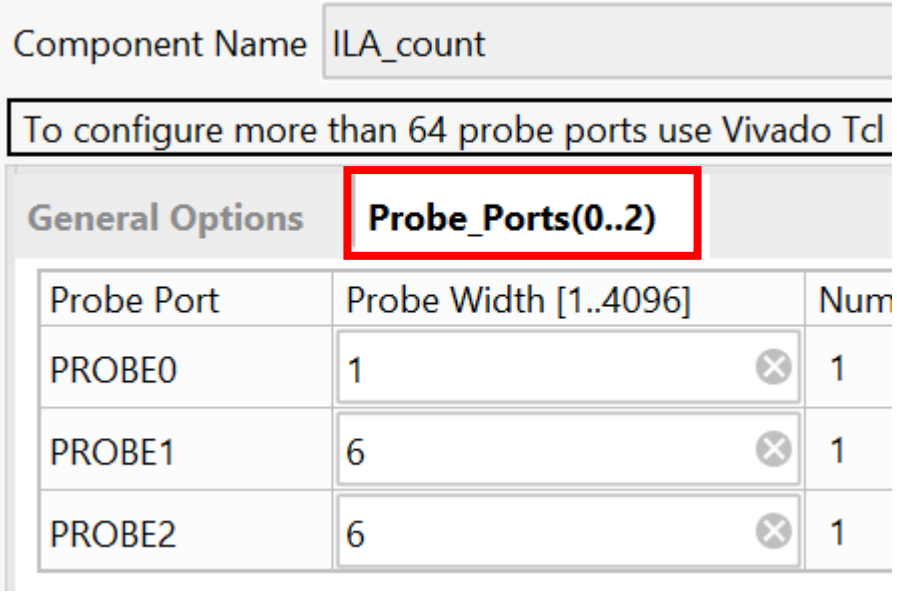
```
reg [5:0] Sec_Count_next;
reg [5:0] Sec_Count_reg=0;
always@(posedge Clk_1Hz or posedge reset)
begin
    if(reset)
        Sec_Count_reg <= 0;
    else
        Sec_Count_reg <= Sec_Count_next;
end
always@(*)
begin
    if(Sec_Count_reg == 59)
        Sec_Count_next = 0;
    else
        Sec_Count_next = Sec_Count_reg + 1;
end
```

```
reg [5:0] Min_Count_next;
reg [5:0] Min_Count_reg=0;
always@(posedge Clk_1Hz or posedge reset)
begin
    if(reset)
        Min_Count_reg <= 0;
    else
        Min_Count_reg <= Min_Count_next;
end
always@(*)
begin
    if(Sec_Count_reg == 59)
        if(Min_Count_reg == 59)
            Min_Count_next = 0;
        else
            Min_Count_next = Min_Count_reg + 1;
    else
        Min_Count_next = Min_Count_reg;
end
assign Sec_Count = Sec_Count_reg;
assign Min_Count = Min_Count_reg;
```

ILA


IP Catalog
top_counter.v
Vio_wrapper.v
Cores
Interfaces
Search: ILA
(5
Name
1
Xylon
Multilayer Video Controller
Debug & Verification
Debug
AXI4-Stream ILA (Integrated Logic Analyzer
ILA (Integrated Logic Analyzer)

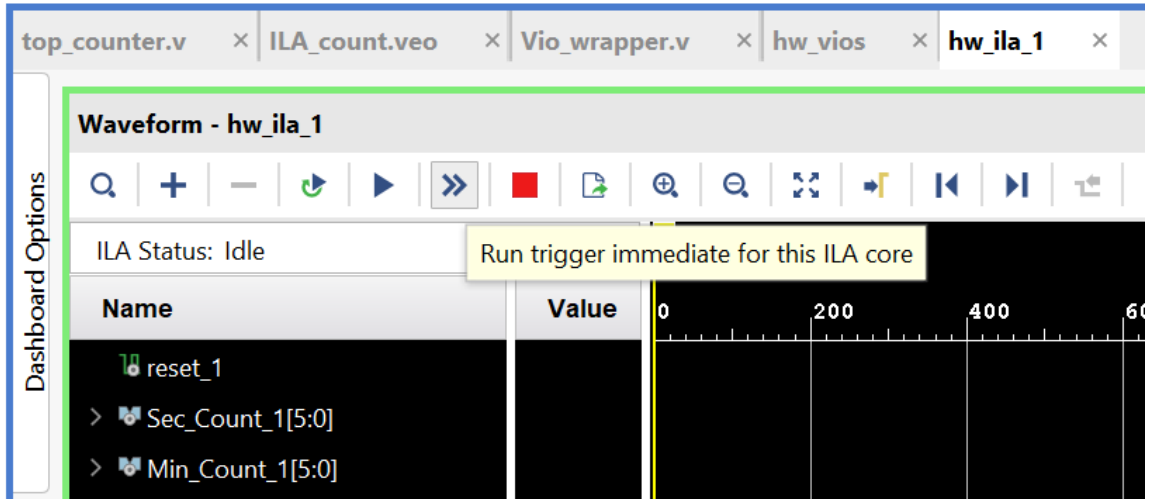

Component Name
ILA_count
To configure more than 64 probe ports use Vivado Tcl Console
General Options
Probe_Ports(0..2)
Monitor Type
Native
AXI
Number of Probes
3
[1...1024]
Sample Data Depth
1024
Same Number of Comparators for All Probe Ports
Number of Comparators
1
Trigger Out Port
Trigger In Port
Input Pipe Stages
0

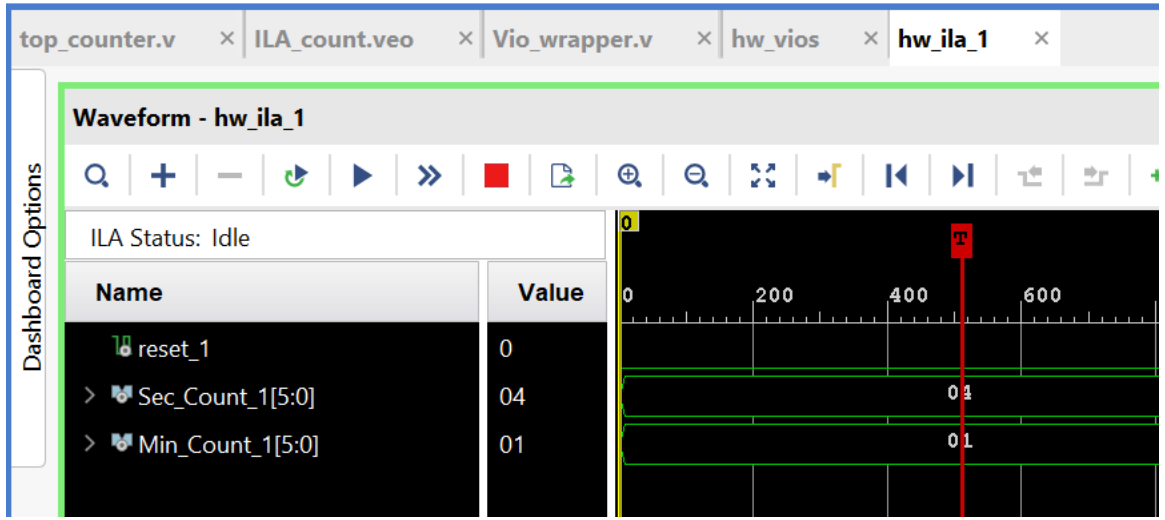

Component Name
ILA_count
To configure more than 64 probe ports use Vivado Tcl
General Options
Probe_Ports(0..2)
Probe Port
Probe Width [1..4096]
Num
PROBE0
1
1
PROBE1
6
1
PROBE2
6
1

Demo

hw_vio_1

Name	Value	Activity	Direction	VIO
reset	[B] 0		Output	hw_vio_1
Min_Count[5:0]	[U] 0		Input	hw_vio_1
Sec_Count[5:0]	[U] 3		Input	hw_vio_1



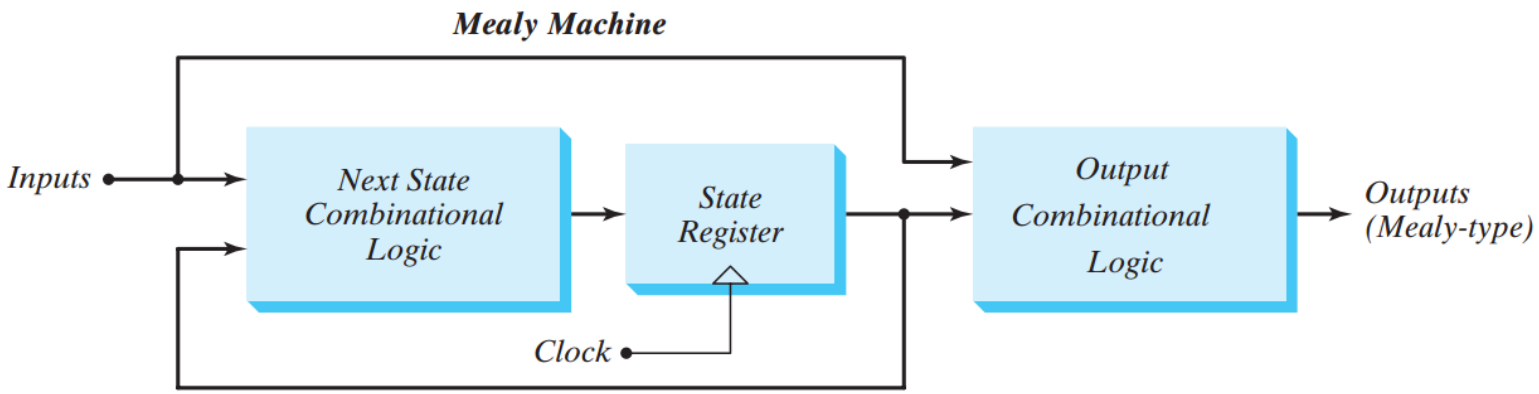


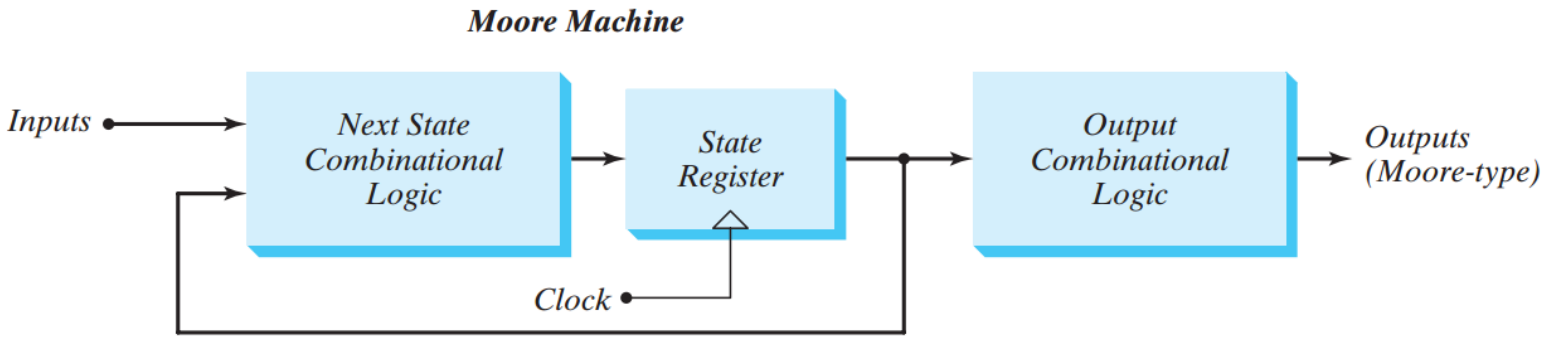
ELD Lab 5
Design of Sequence Detector

Objective

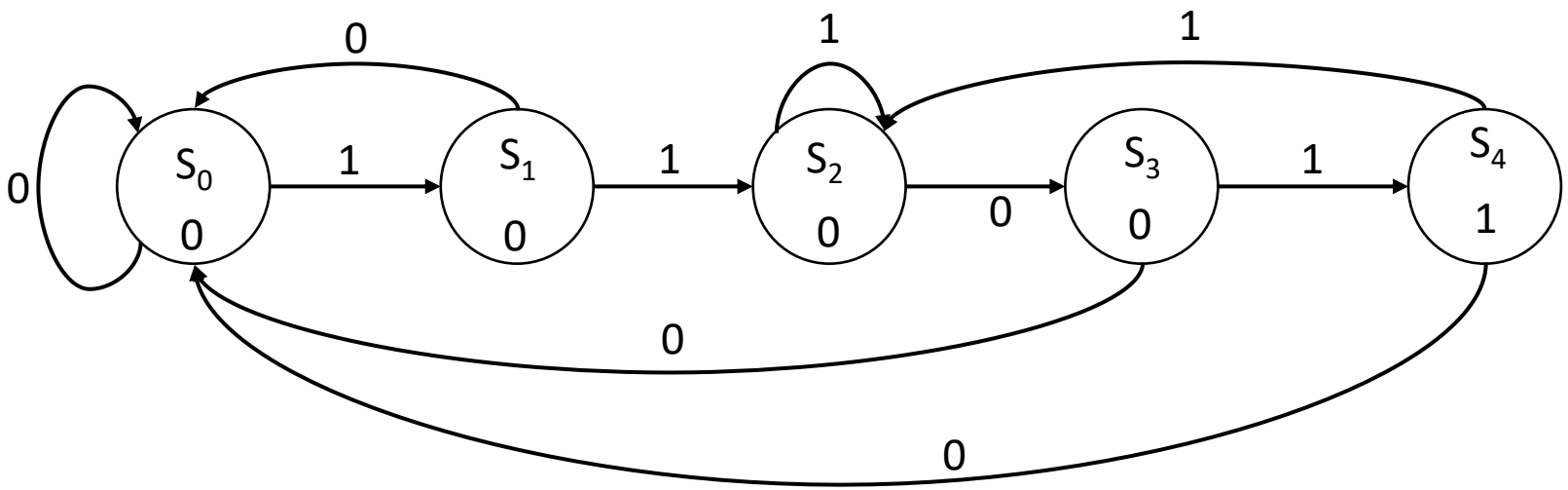
- Design 1011 sequence detector using behavioral modelling
- Verify the circuit using virtual input and output (VIO).
- **Lab Homework:** Change the sequence to 1111

Finite State Machines

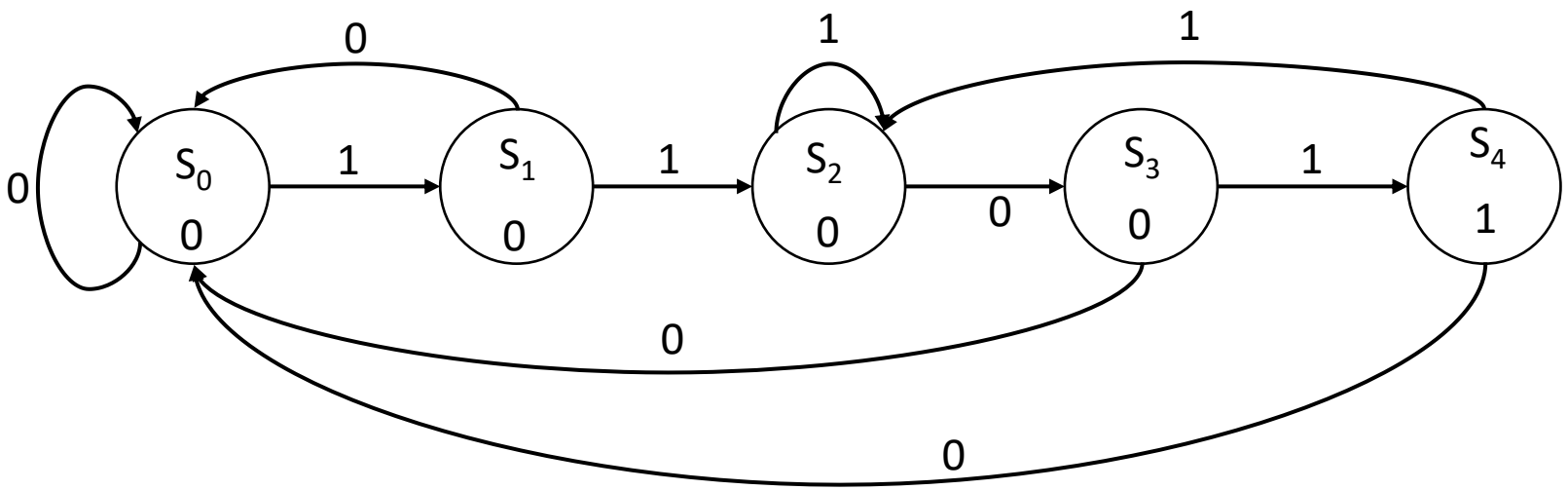




FSM: Sequence Detector 1101 (Moore)



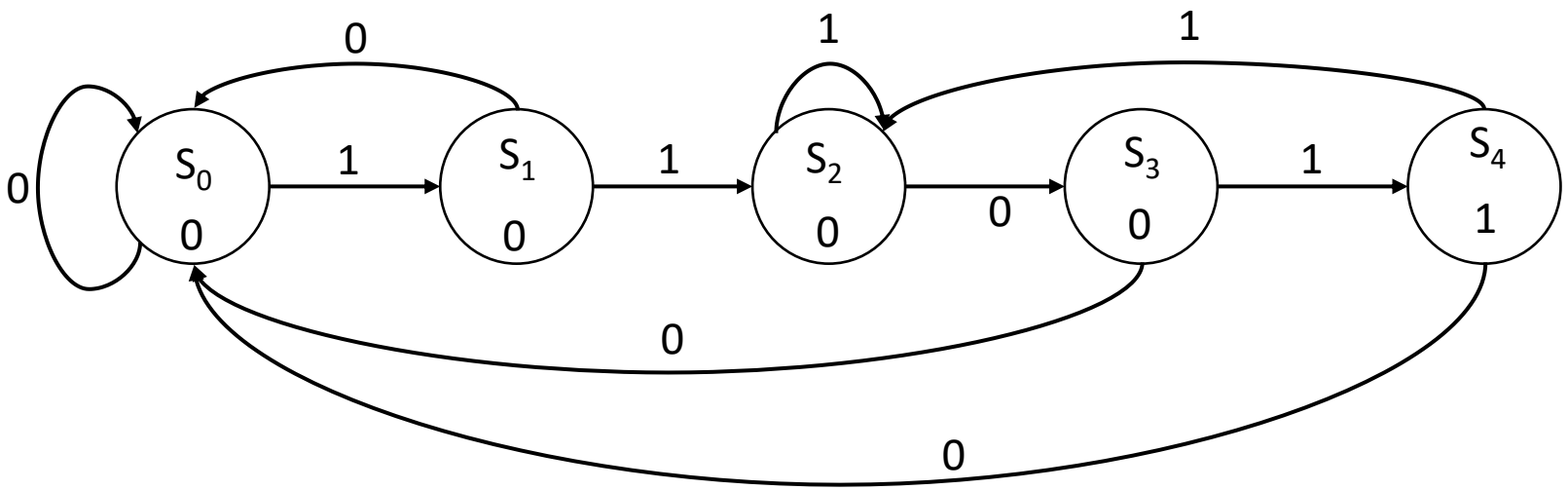
FSM: Sequence Detector 1101 (Moore)



1
```
module FSM_moore_1101 (
input wire clk ,
input wire clr ,
input wire din ,
output reg dout
);
```

2
```
reg[2:0] present_state, next_state;
parameter  S0 = 3'b000, S1 =3'b001, S2 = 3'b010,  // states
           S3 = 3'b011, S4 = 3'b100;
```

FSM: Sequence Detector 1101 (Moore)



4

```
always @(*)
  begin
    case(present_state)
      S0: if(din == 1)
            next_state = S1;
          else
            next_state = S0;
      S1: if(din == 1)
            next_state = S2;
          else
            next_state = S0;
      S2: if(din == 0)
            next_state = S3;
          else
            next_state = S2;
      S3: if(din == 1)
            next_state = S4;
          else
            next_state = S0;
      S4: if(din == 0)
            next_state = S0;
          else
            next_state = S2;
    default next_state = S0;
    endcase
  end
```

3

```
// State registers
always @(posedge clk or posedge clr)
  begin
    if (clr == 1)
          present_state <= S0;
    else
        present_state <= next_state;
  end
```

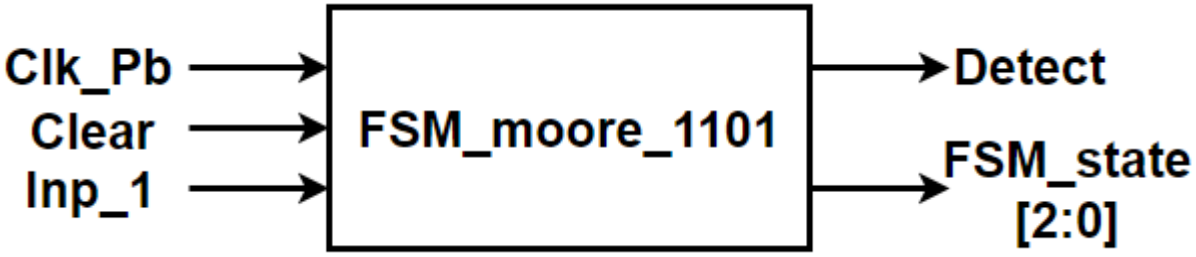
5

```
always @(*)
  begin
    if(present_state == S4)
        dout = 1;
    else
        dout = 0;
  end
```

Lab

Proposed Approach

FSM



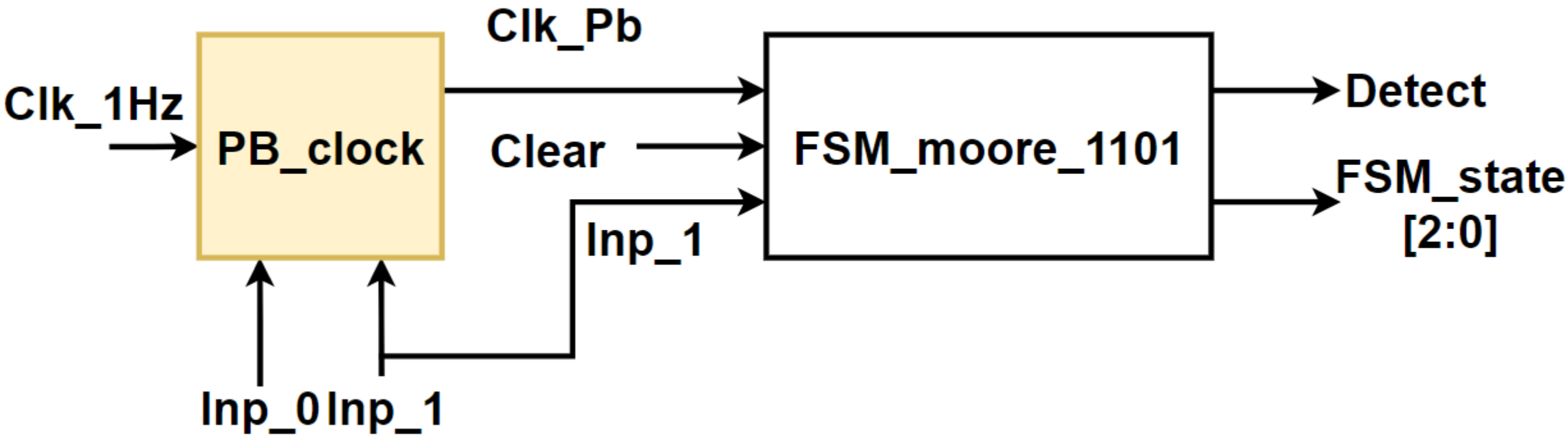

```
module FSM_moore_1101(
   input Clk_pb,
   input Clear,
   input Inp_1,
   output reg Detect,
   output [2:0] FSM_state
   );

   parameter S0 = 3'b000, S1 = 3'b001, S2 = 3'b010, S3 = 3'b011, S4 = 3'b100;
   reg [2:0] present_state = S0;
   reg [2:0] next_state;
```

```
always@(posedge Clk_pb or posedge Clear)
begin
    if (Clear)
        present_state <= S0;
    else
        present_state <= next_state;
end
```

```
always@(*)
begin
    if(present_state == S4)
        Detect = 1;
    else
        Detect = 0;
end
assign FSM_state = present_state;
```

```
always@(*)
begin
   case(present_state)
      S0: if(Inp_1 == 1)
            next_state = S1;
          else
            next_state = S0;
      S1: if(Inp_1 == 1)
            next_state = S2;
          else
            next_state = S0;
      S2: if(Inp_1 == 0)
            next_state = S3;
          else
            next_state = S2;
      S3: if(Inp_1 == 1)
            next_state = S4;
          else
            next_state = S0;
      S4: if(Inp_1 == 0)
            next_state = S0;
          else
            next_state = S2;
      default next_state = S0;
   endcase
end
```

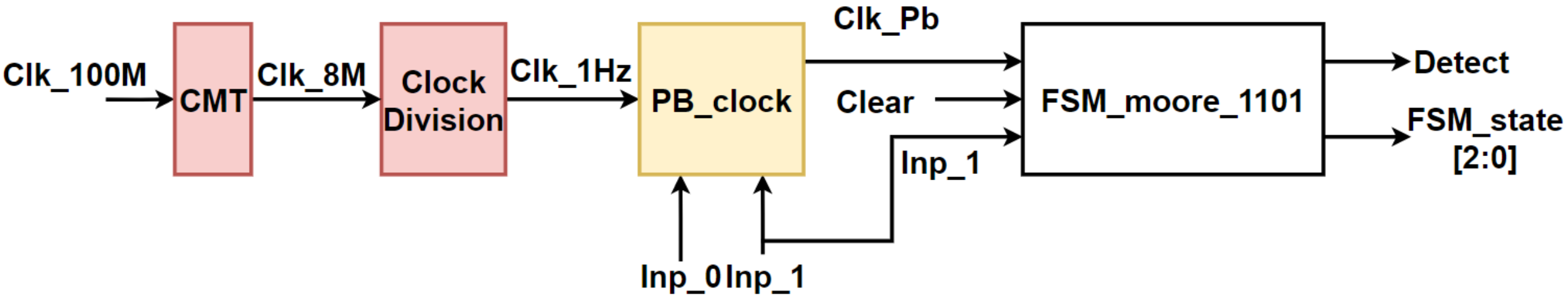
Detection of Push Button Press



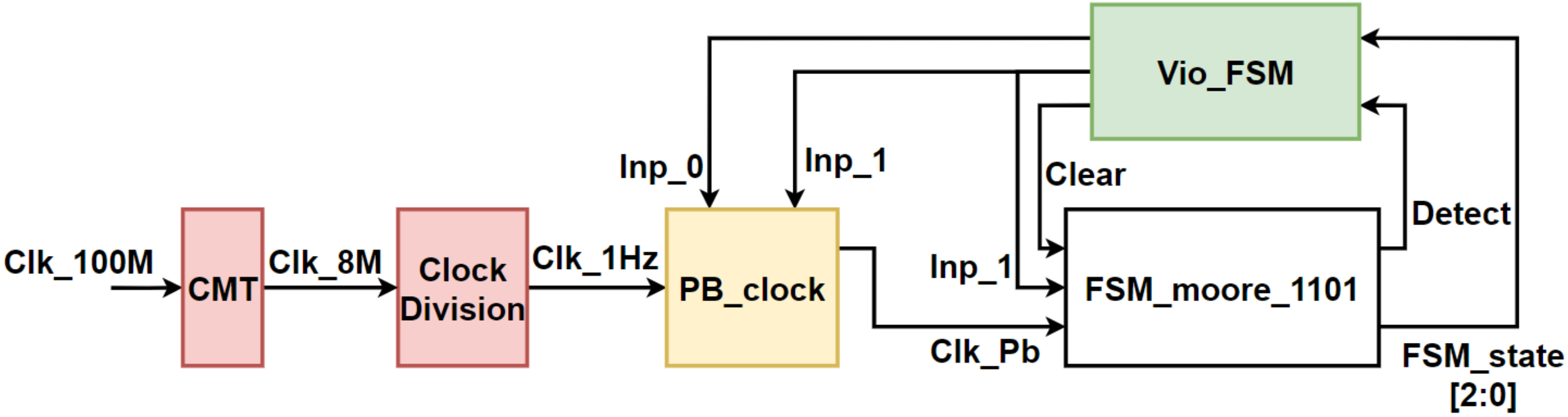
Detection of Push Button Press

```
module PB_clock(
    input Clk_1Hz,
    input Inp_0,
    input Inp_1,
    output reg Clk_pb
    );
    wire Inp_pulse;
    assign Inp_pulse = Inp_0 | Inp_1;

    always@(posedge Clk_1Hz)
        Clk_pb <= Inp_pulse;

endmodule
```

FSM with Clock Division

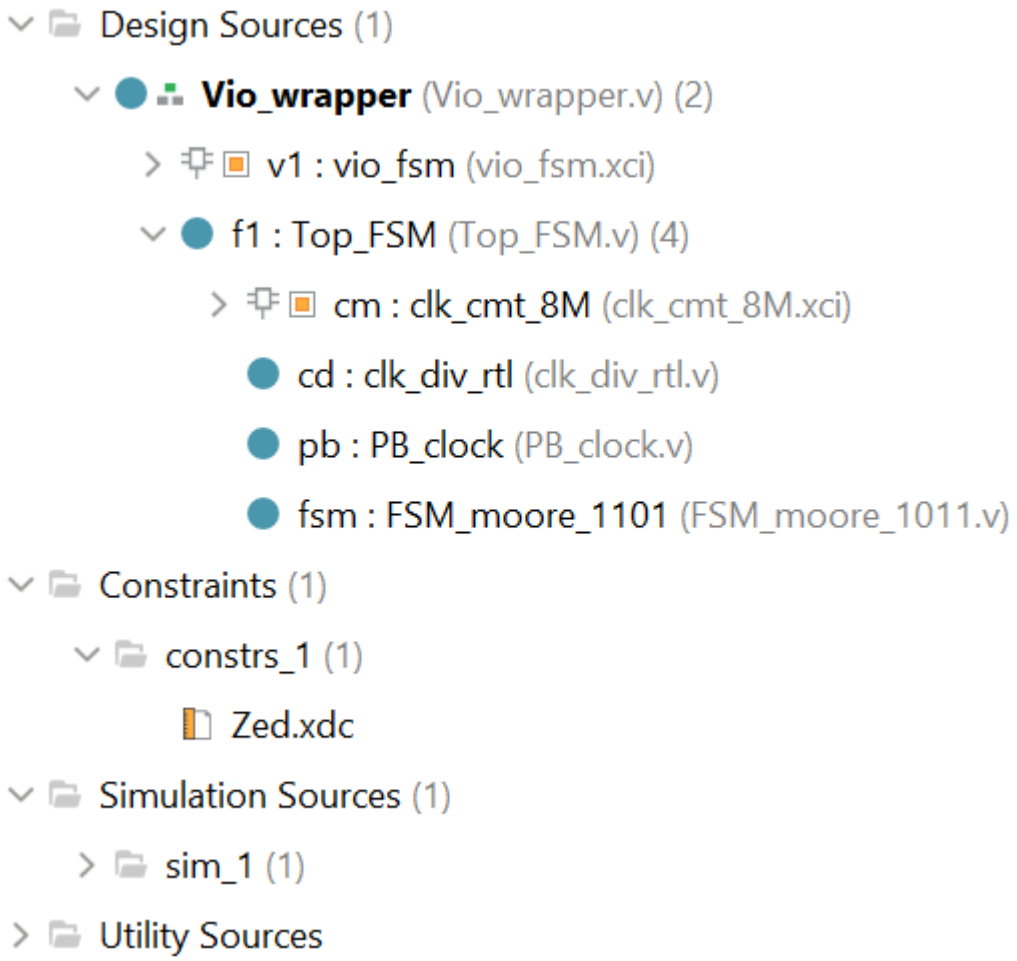


VIO Wrapper


Vio_FSM
Inp_0
Inp_1
Clear
Detect
Clk_100M
CMT
Clk_8M
Clock Division
Clk_1Hz
PB_clock
Inp_1
FSM_moore_1101
Clk_Pb
FSM_state [2:0]

Demo

- Add XDC file and generate bitstream
- Verify the functionality using VIO
- Use toggle button option in VIO



Possible Extensions

- Moore/Mealy FSMs for any sequence
- Asynchronous/Synchronous active high/low Clear
- Change the FSM input clock to Clk_xHz where x can be any positive integer
- Design a counter (2->4->8->2->4->8…..) using FSM
- FSM with two outputs: Detect and Error. Detect is set to 1 when sequence is correct and Error is set to 1 when sequence is wrong.

ELD Lab 6
Exploring AXI Interface

Objective

- Understand the basics of AXI Interface
- Design floating point arithmetic using logarithmic and square root IPs available in Vivado

$$y = \frac{1}{\ln(x)}$$

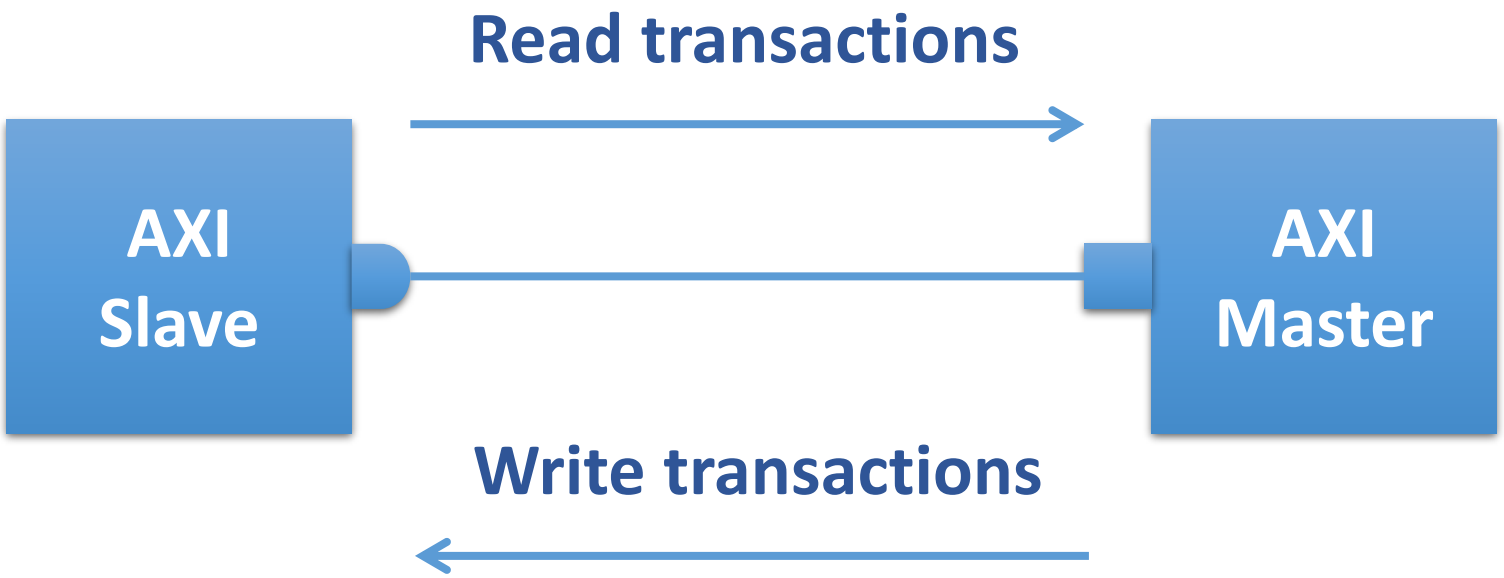
- **Lab Homework:** Design the floating point arithmetic circuit to implement following equation

$$z = \sqrt{x} + \frac{1}{\ln y} + 1.5$$

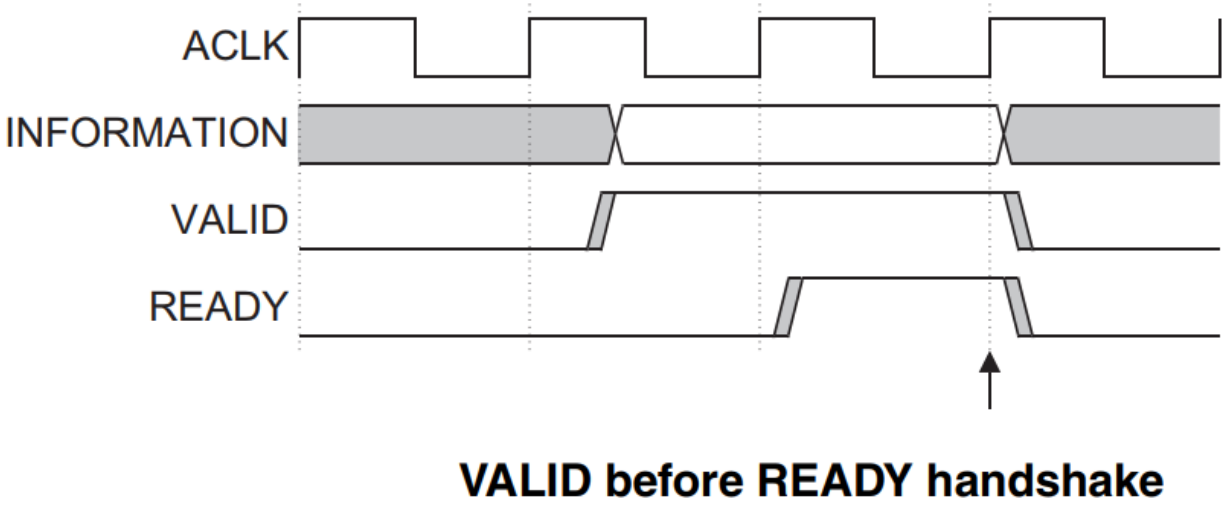
Theory

AXI

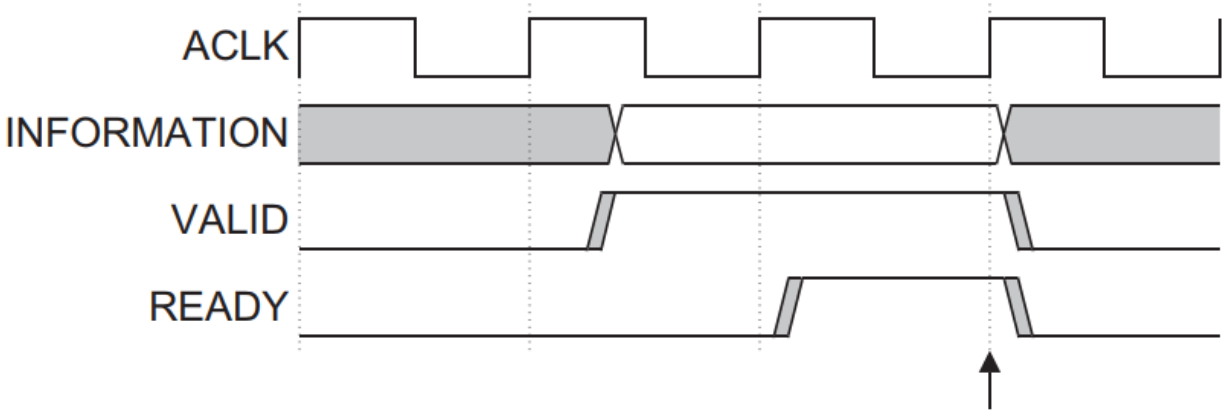
- Every AXI link contains two part: AXI master and AXI slave.
- AXI master initializes the transactions such as read and write. AXI slave is the one who responds to AXI master transactions.
- **Transaction:** Transfer of data from one point to another point



AXI Memory Mapped



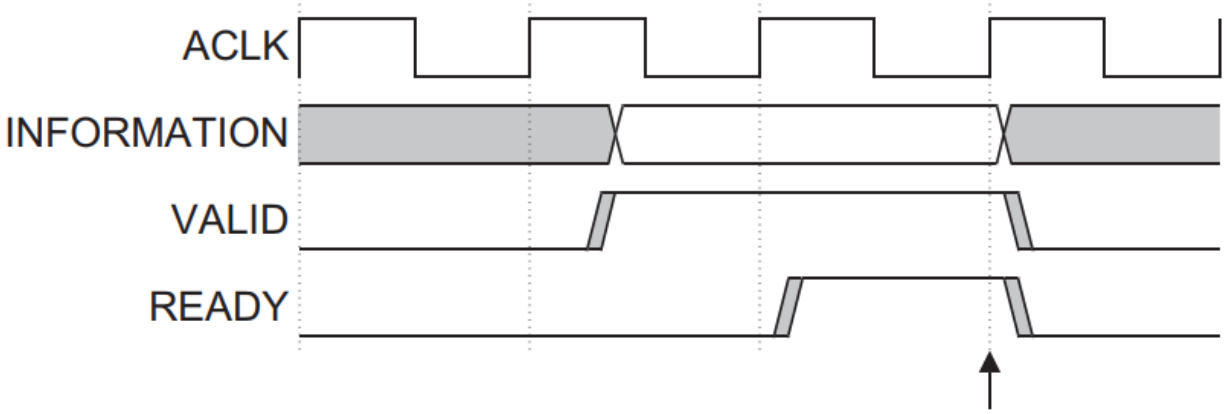
AXI Memory Mapped



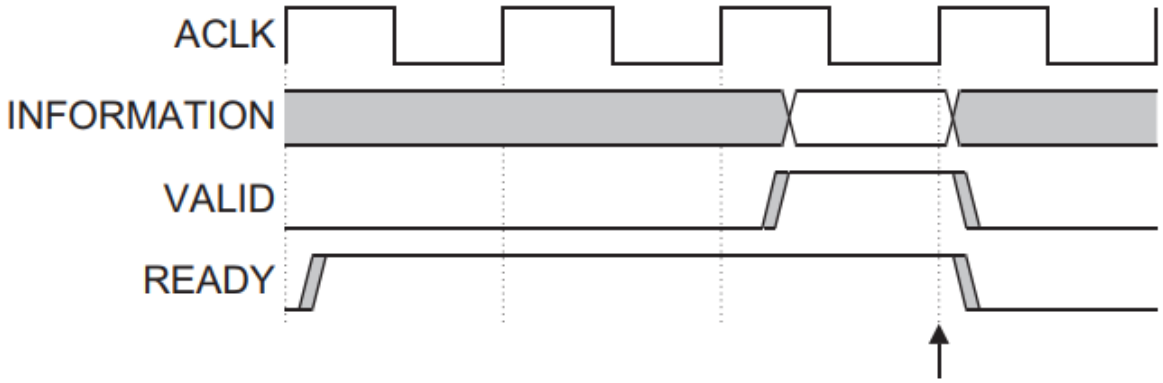
VALID before READY handshake

- The *source* generates the **VALID** signal to indicate when the address, data or control information is available.
- The *destination* generates the **READY** signal to indicate that it can accept the information.
- Transfer occurs only when *both* the **VALID** and **READY** signals are HIGH

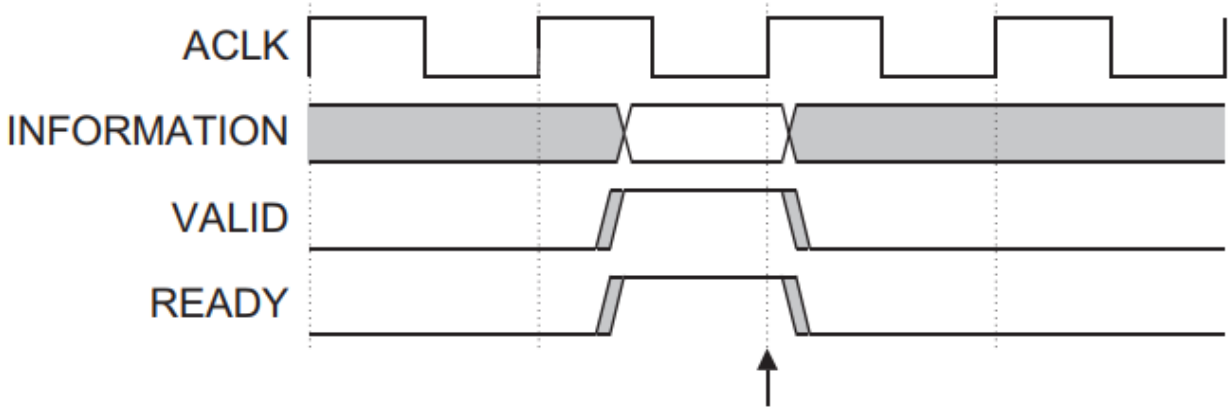
AXI Memory Mapped



VALID before READY handshake

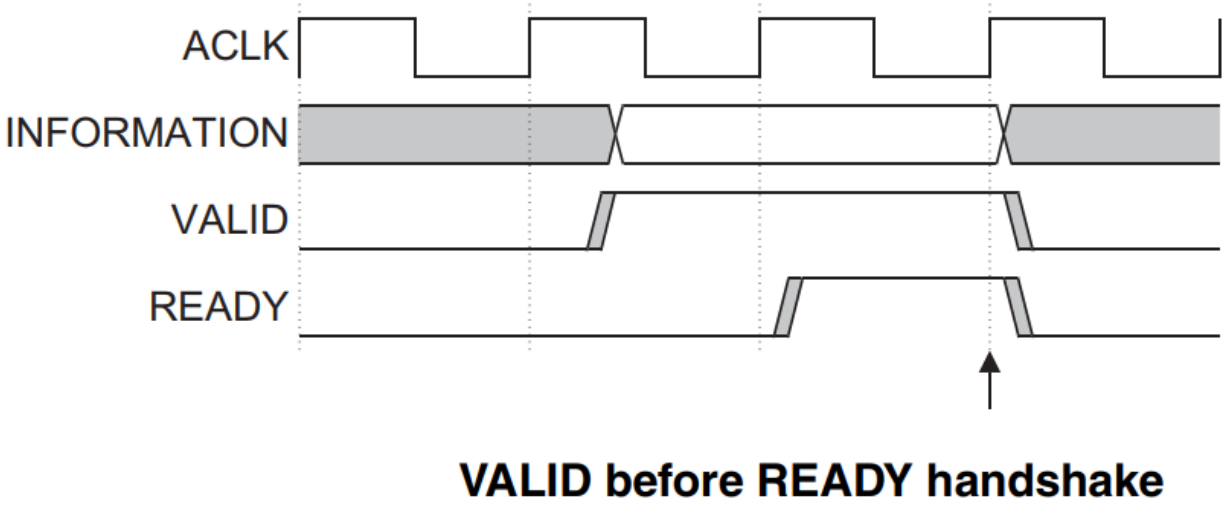


READY before VALID handshake



VALID with READY handshake

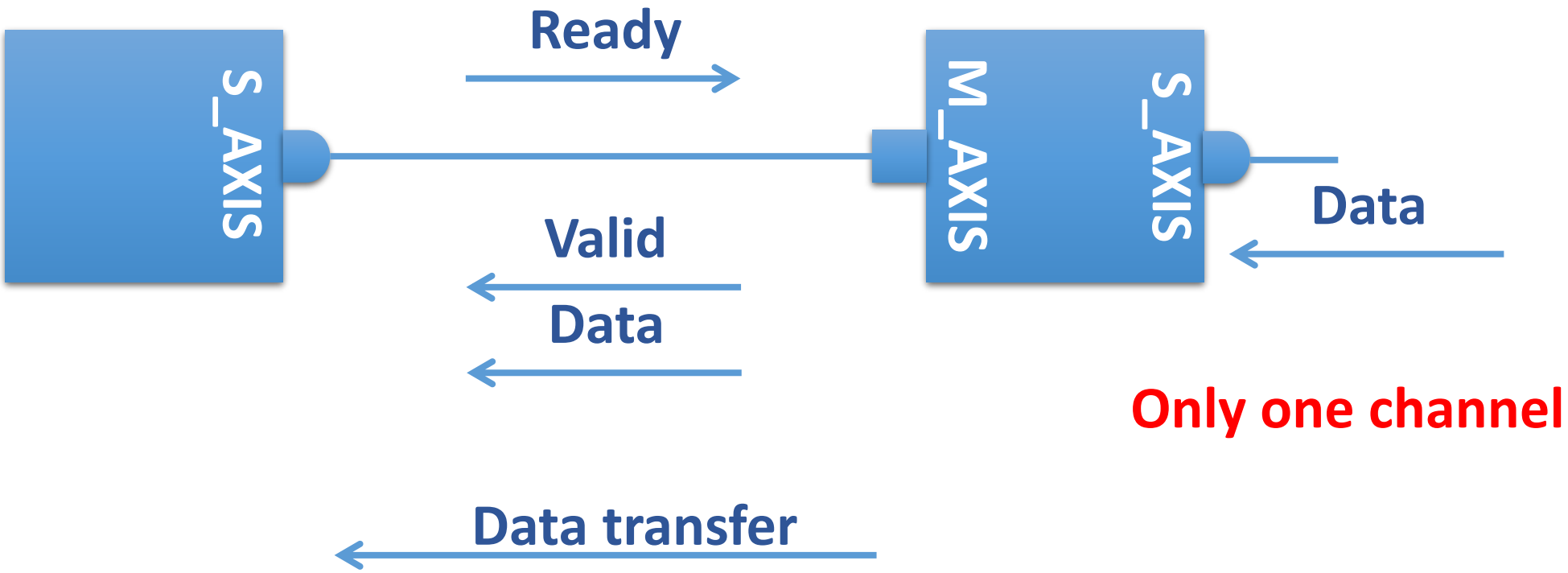
AXI Memory Mapped



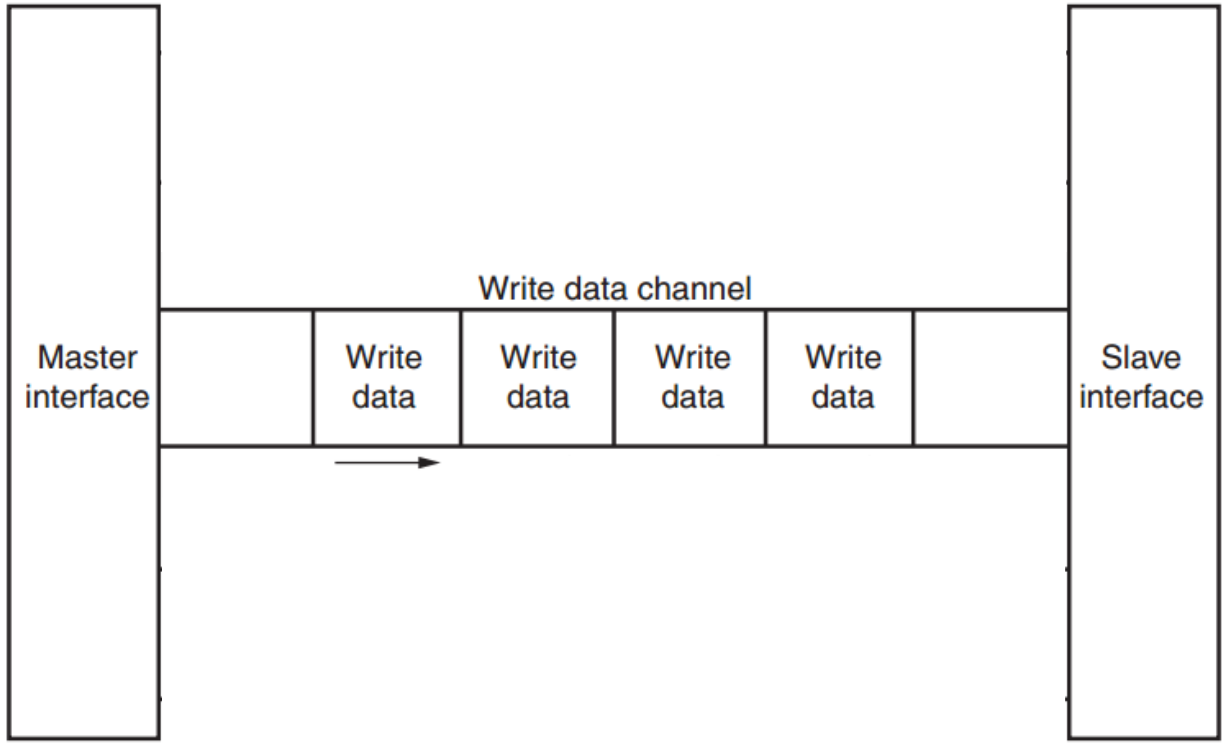
- A source is NOT permitted to wait until READY is asserted before asserting VALID
- Once VALID is asserted it must remain asserted until the handshake occurs, at a rising clock edge at which VALID and READY are both asserted.
- A destination is permitted to wait for **VALID** to be asserted before asserting the corresponding **READY**.
- If **READY** is asserted, it is permitted to deassert **READY** before **VALID** is asserted.

AXI Stream

- The AXI4-Stream protocol defines a single channel for transmission of streaming data (unlimited burst).



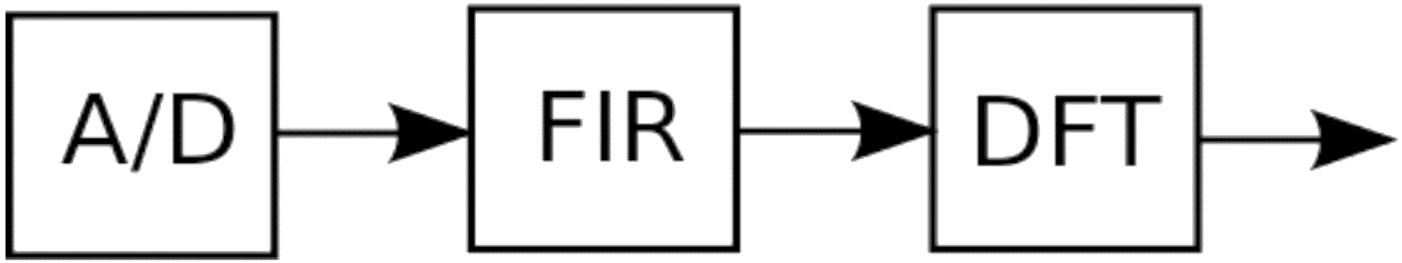
AXI Stream



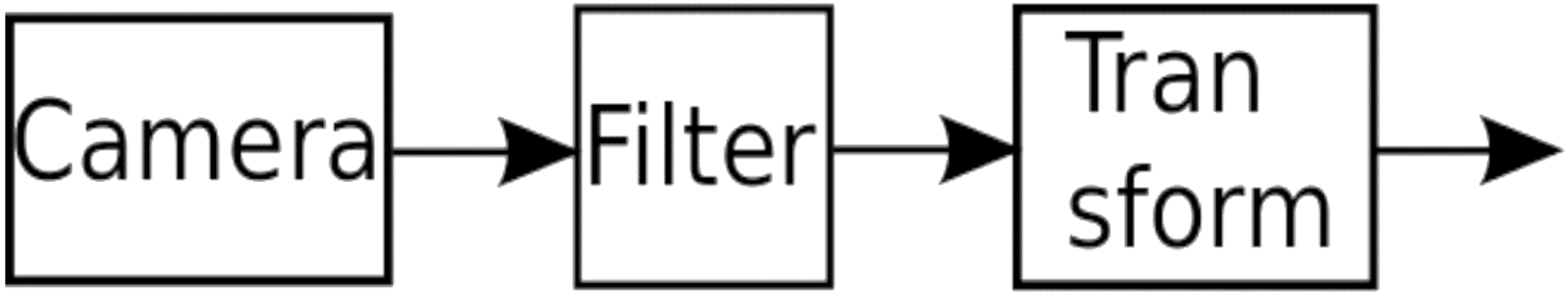
- The AXI4-Stream channel is modeled after the Write Data channel of the AXI4.
- Unlike AXI4, AXI4-Stream interfaces can burst an **unlimited** amount of data.

AXI Stream

Signal Processing

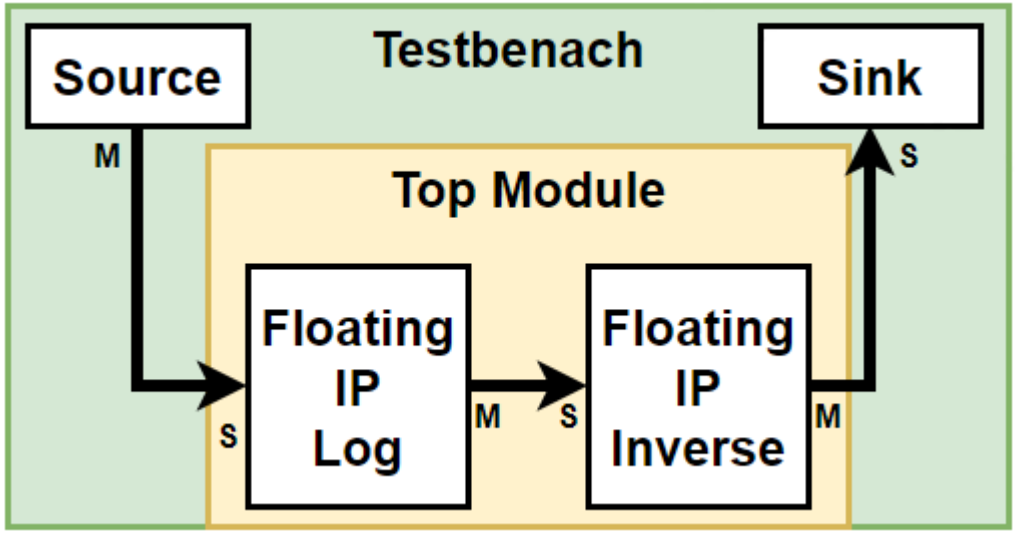


Video Processing

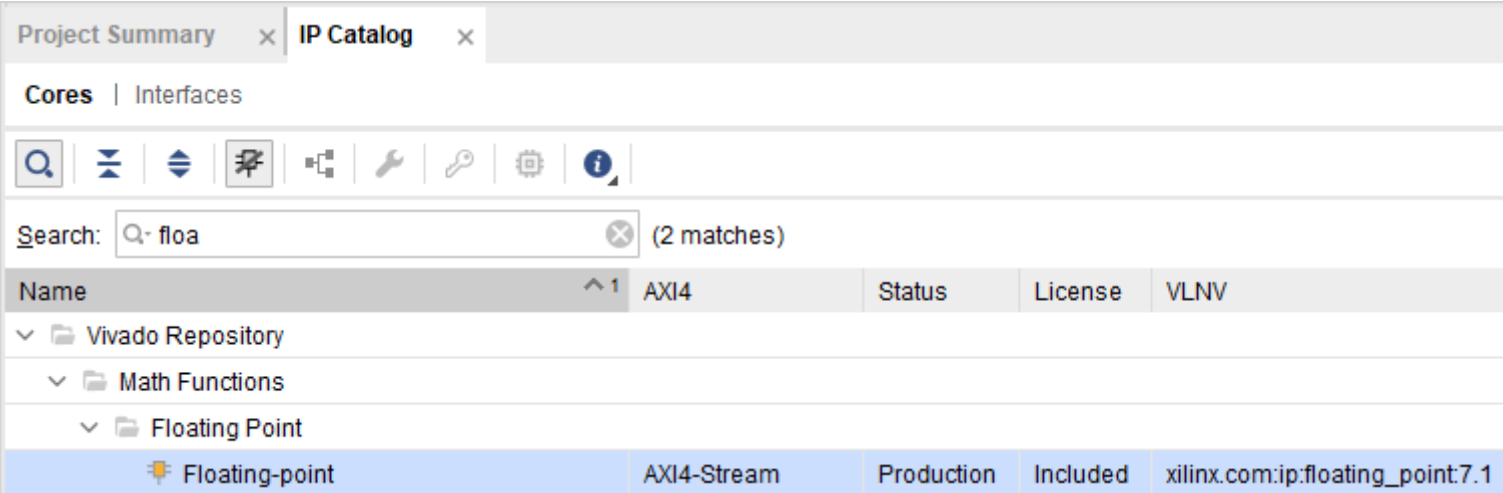


Lab

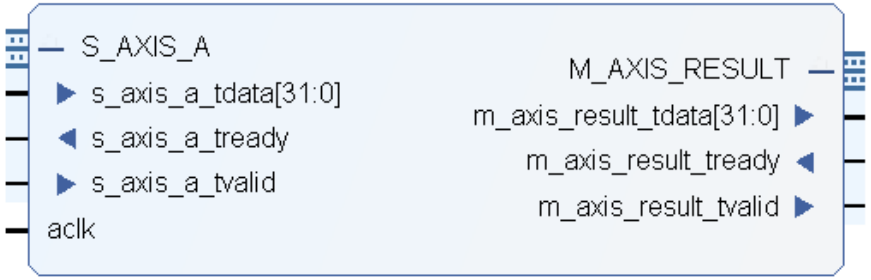
Proposed Approach



Locate the IP in Vivado



Log Operation



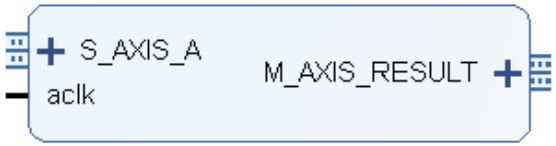
Customize IP

Floating-point (7.1)

Documentation | IP Location | Switch to Defaults

IP Symbol | Implementation Details

Show disabled ports



Component Name FP_log

Operation Selection | Precision of Inputs | Optimizations | Interface Optic

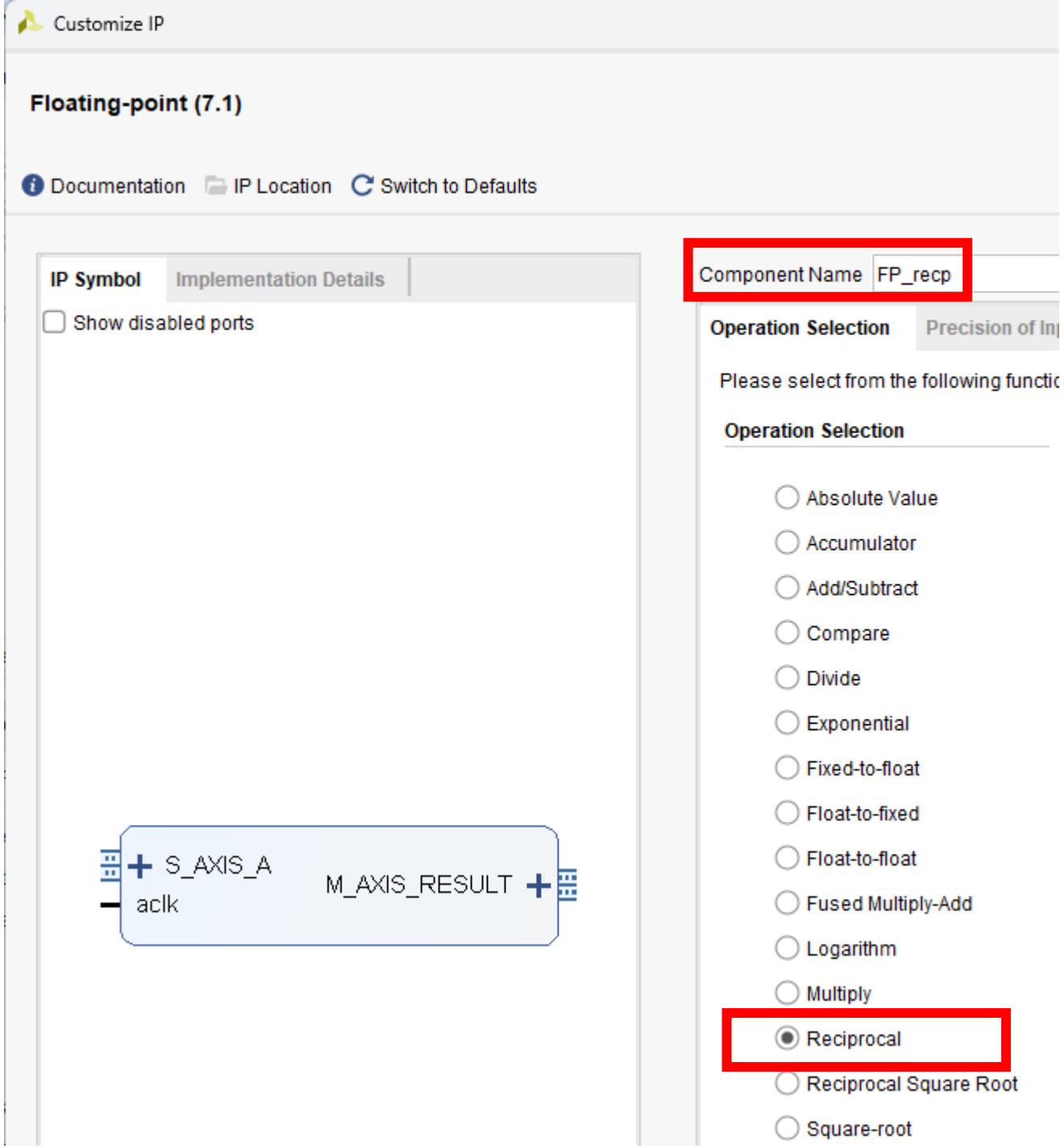
Please select from the following functions:

Operation Selection

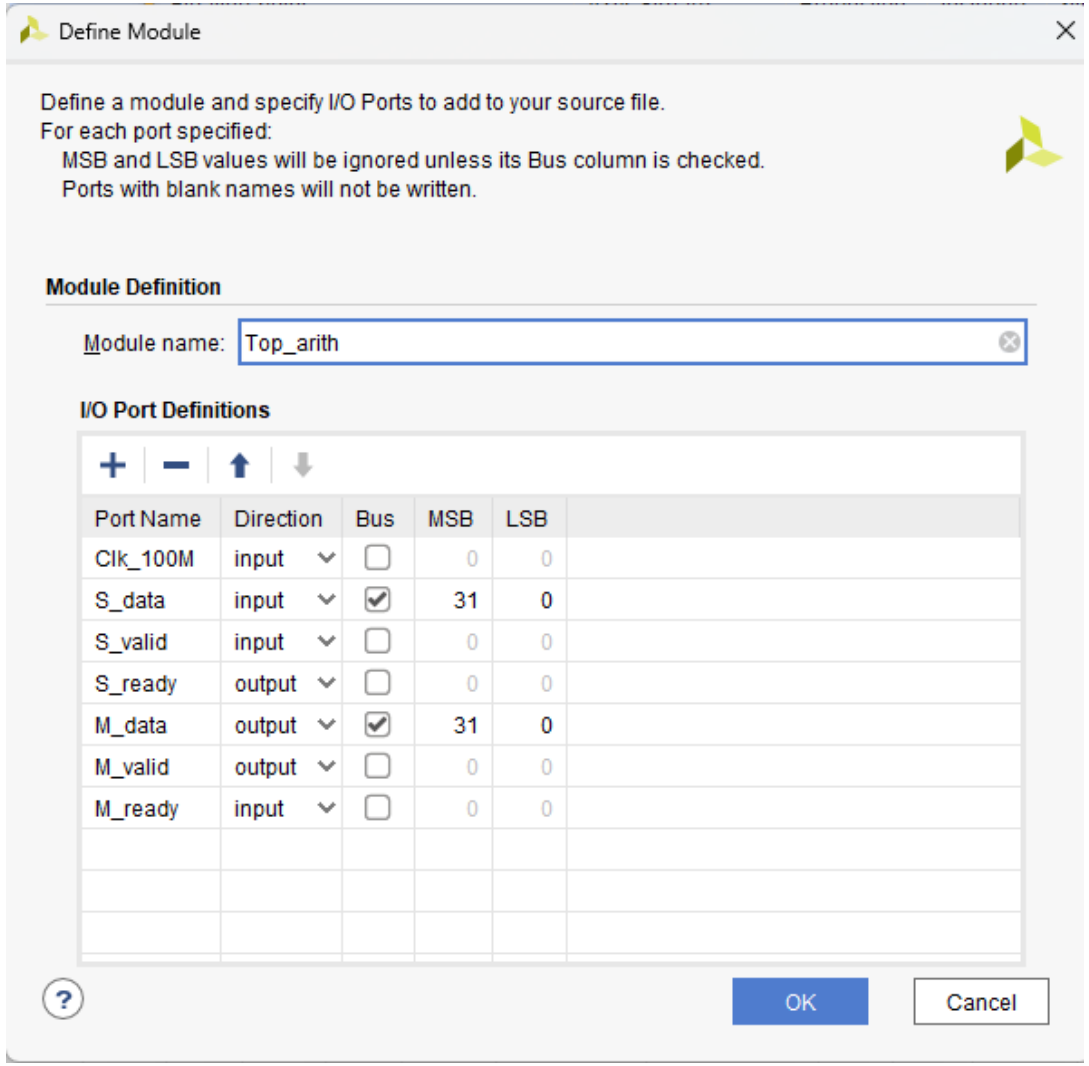
- Absolute Value
- Accumulator
- Add/Subtract
- Compare
- Divide
- Exponential
- Fixed-to-float
- Float-to-fixed
- Float-to-float
- Fused Multiply-Add
- Logarithm
- Multiply
- Reciprocal
- Reciprocal Square Root
- Square-root

Logarithm operation selected. **RESULT = ln(A)**

Reciprocal operation

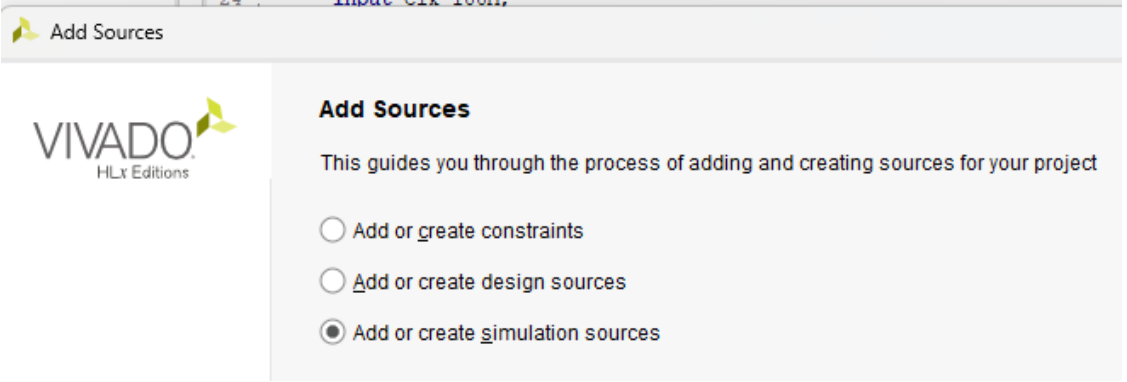


Top Module




```
module Top_arith(
    input Clk_100M,
    input [31:0] S_data,
    input S_valid,
    output S_ready,
    output [31:0] M_data,
    output M_valid,
    input M_ready
    );

wire int_valid, int_ready;
wire [31:0] int_data;
FP_log l1 (
  .aclk(Clk_100M),                                  // input wire aclk
  .s_axis_a_tvalid(S_valid),              // input wire s_axis_a_tvalid
  .s_axis_a_tready(S_ready),              // output wire s_axis_a_tready
  .s_axis_a_tdata(S_data),                // input wire [31 : 0] s_axis_a_tdata
  .m_axis_result_tvalid(int_valid),  // output wire m_axis_result_tvalid
  .m_axis_result_tready(int_ready),  // input wire m_axis_result_tready
  .m_axis_result_tdata(int_data)    // output wire [31 : 0] m_axis_result_tdata
);

FP_recp r1 (
  .aclk(Clk_100M),                                  // input wire aclk
  .s_axis_a_tvalid(int_valid),              // input wire s_axis_a_tvalid
  .s_axis_a_tready(int_ready),              // output wire s_axis_a_tready
  .s_axis_a_tdata(int_data),                // input wire [31 : 0] s_axis_a_tdata
  .m_axis_result_tvalid(M_valid),  // output wire m_axis_result_tvalid
  .m_axis_result_tready(M_ready),  // input wire m_axis_result_tready
  .m_axis_result_tdata(M_data)    // output wire [31 : 0] m_axis_result_tdata
);

endmodule
```

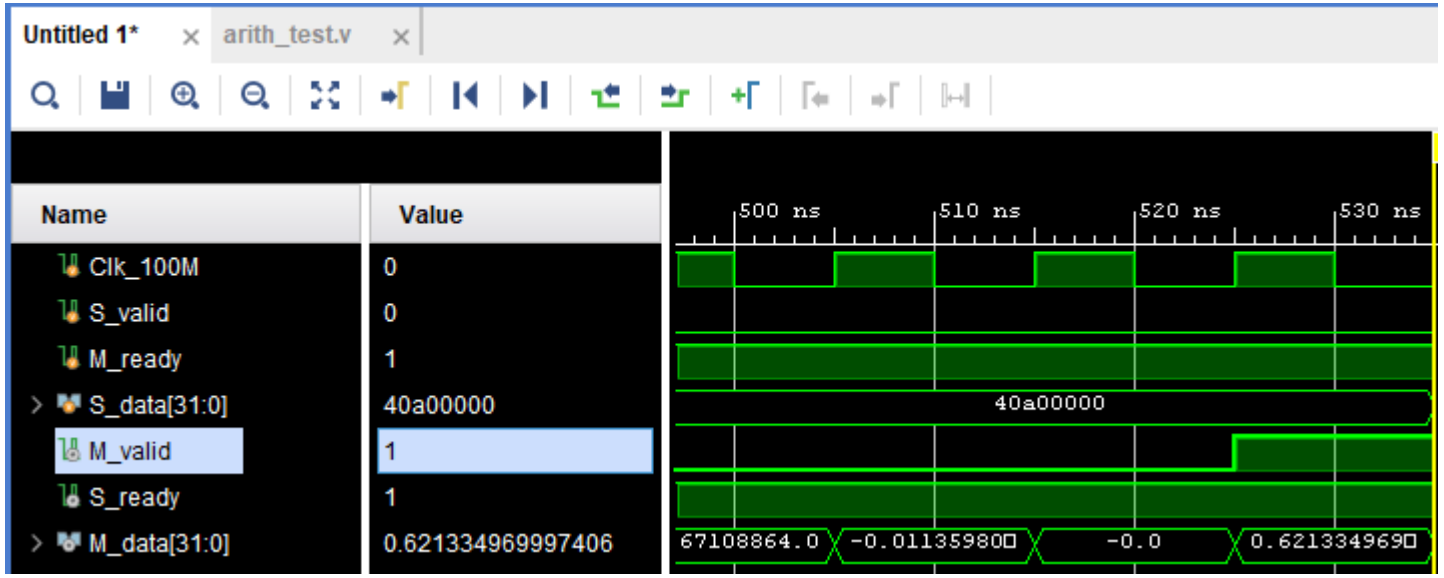
Testbench



- Floating point converter: https://www.h-schmidt.net/FloatConverter/IEEE754.html

```
module arith_test(

    );

    reg Clk_100M, S_valid, M_ready;
    reg [31:0] S_data;
    wire M_valid, S_ready;
    wire [31:0] M_data;
    Top_arith t1 (.Clk_100M(Clk_100M),.S_data(S_data),.S_valid(S_valid),.S_ready(S_ready),.M_data(M_data),.M_valid(M_valid),.M_ready(M_ready));

    initial begin
        Clk_100M = 0;
        S_valid = 0;
        S_data = 0;
        M_ready = 1;
    end

    always
        #5 Clk_100M = ~Clk_100M;

    initial begin
        S_data = 32'b01000000101000000000000000000000;
        S_valid = 1;
        while (S_ready == 0)
            S_valid = 1;
        #5 S_valid = 0;
        @(posedge M_valid);
        #10 $stop;
    end
```

Simulations



Simulations

```
initial begin
    S_data = 32'b01000000101000000000000000000000;
    S_valid = 1;
    M_ready = 1;
    while (S_ready == 0)
        S_valid = 1;
    #5 S_valid = 0;
    @(posedge M_valid);
    #5 M_ready = 0;

    #50 S_data = 32'b01000001110010000000000000000000;
    S_valid = 1;
    M_ready = 1;
    while (S_ready == 0)
        S_valid = 1;
    #5 S_valid = 0;
    @(posedge M_valid);

    #10 $stop;
```