

# Introduction to Verilog RTL

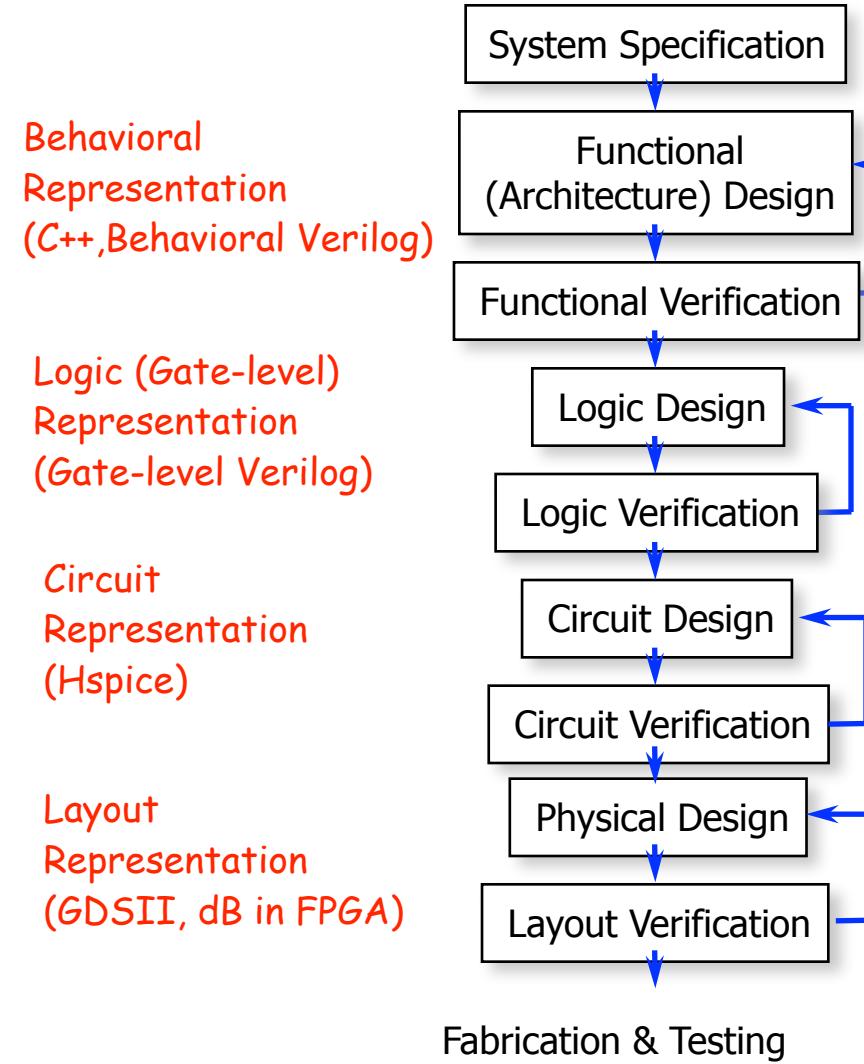
Hsi-Pin Ma

<https://eclasse.nthu.edu.tw/course/18498>

Department of Electrical Engineering  
National Tsing Hua University

# Introduction

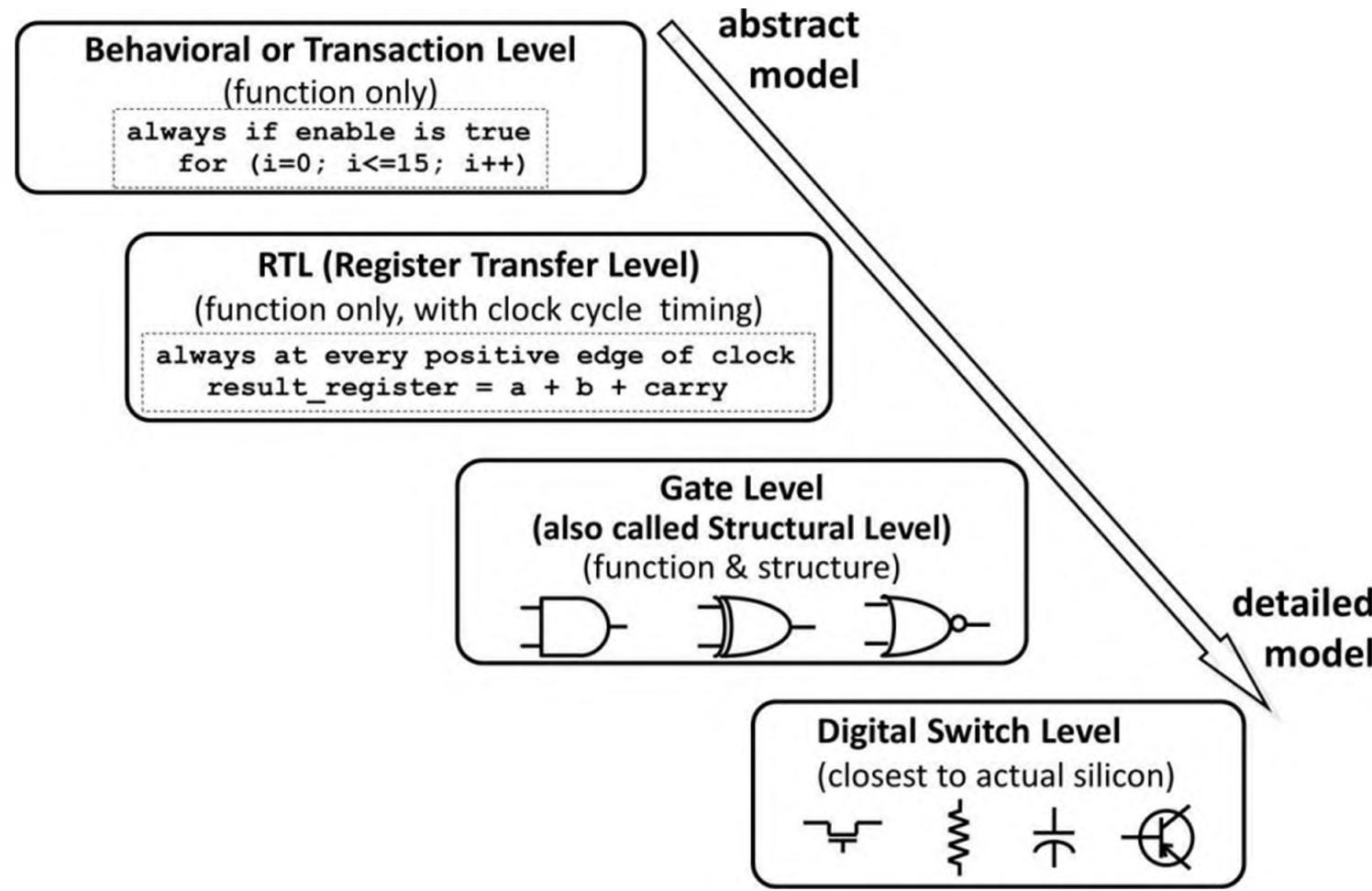
# VLSI Design Flow



# Hardware Description Language

- A high-level programming language offering special constructs for hardware **modeling** and **verification**
  - Describe the operation of a circuit at various level of abstraction
  - Describe the timing of a circuit
  - Express the concurrency of circuit operation

# Levels of Abstraction



[Stuart Sutherland, RTL Modeling with SystemVerilog for Simulation and Synthesis: Using SystemVerilog for ASIC and FPGA Design]

# IEEE Language Standard

## Verilog and SystemVerilog

SystemVerilog-2005/2009/2012/2017

verification	assertions	mailboxes	classes	dynamic arrays	2-state types
	test program blocks	semaphores	inheritance	associative arrays	shortreal type
design	clocking domains	constrained random values	strings	queues	globals
	process control	direct C function calls	references	checkers	let macros
interfaces	packed arrays	break	enum	++ -- += -= *= /=	
	nested hierarchy	array assignments	continue	typedef	>>= <<= >>>= <<<=
unrestricted ports	unique/priority case/if	return	structures	&=  = ^= %=	
	automatic port connect	void functions	do-while	unions	==? !=?
enhanced literals	function input defaults	case inside	2-state types	inside	
	function array args	aliasing	packages	streaming	
time values and units	parameterized types	const	\$unit	casting	
	specialized procedures				

Verilog-2005

uwire

'begin\_keywords

'pragma

\$clog2

Verilog-2001

ANSI C style ports

standard file I/O

(\* attributes \*)

multi dimensional arrays

generate

\$value\$plusargs

configurations

signed types

localparam

'ifndef 'elsif 'line

memory part selects

automatic

constant functions

@\*

variable part select

\*\* (power operator)

Verilog-1995 (created in 1984)

modules

\$finish \$fopen \$fclose

initial

wire reg

begin-end

+ = \* /

parameters

\$display \$write

disable

integer real

while

%

function/tasks

\$monitor

events

time

for forever

>> <<

always @

'define 'ifdef 'else

wait # @

packed arrays

if-else

assign

'include 'timescale

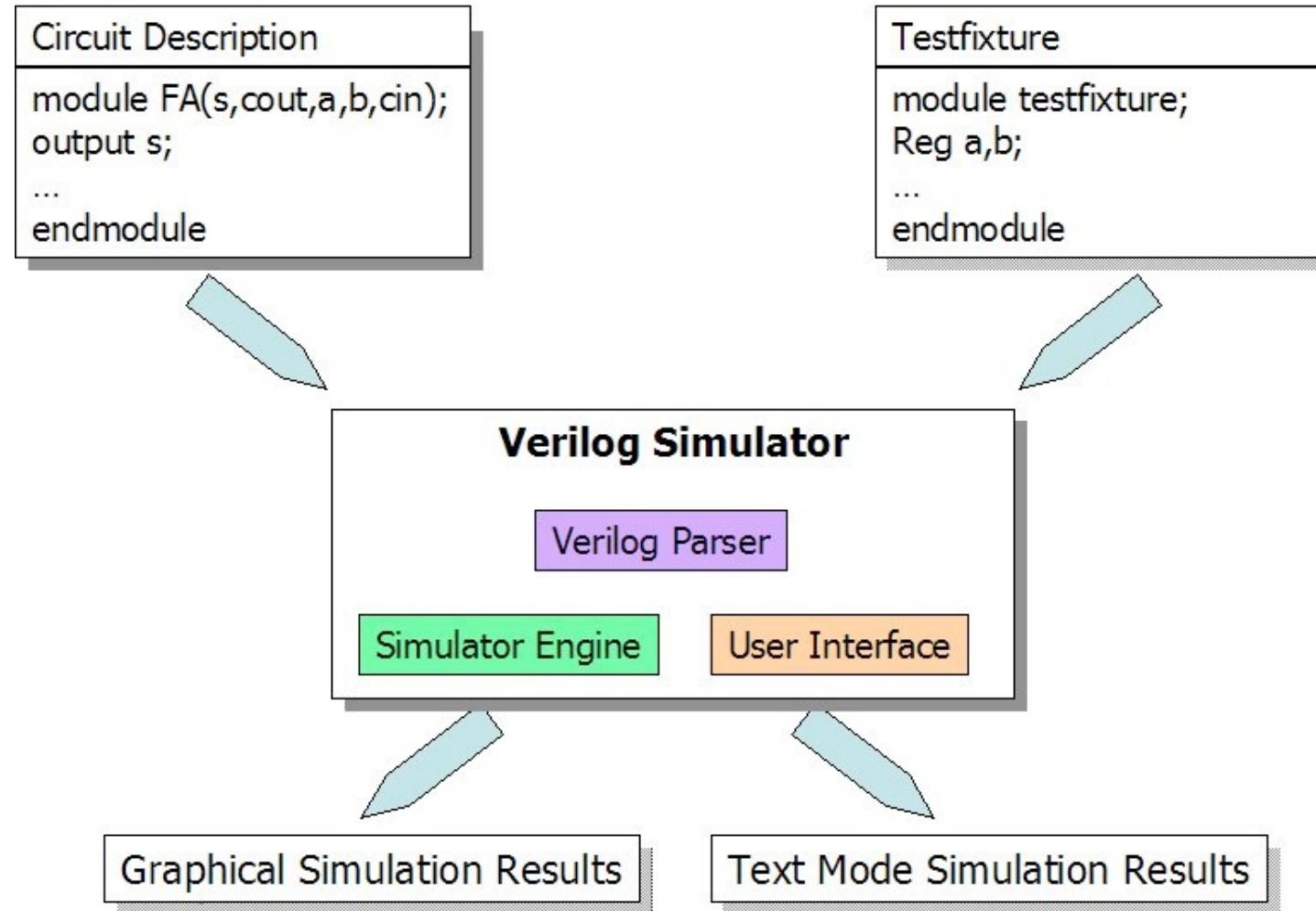
fork-join

2D memory

repeat

# Sample Design

# Verilog Simulation



# Logic Function in Verilog

- Combinational logic

- logic function:  $out = a \cdot sel + b \cdot sel'$
  - **assign**  $out = (a \& sel) | (b \& (\sim sel)) ;$

- Basic logic operator

- AND: &
  - OR: |
  - NOT: ~

# Hardware Model: Verilog Module

module name declaration

```
module smux(  
    output out,  
    input a,  
    input b,  
    input sel  
)
```

I/O definition

```
assign out = (a&sel) | (b&(~sel));
```

```
endmodule
```

module functionality

# Logic Verification: Testbench

I/O definition

```
module test_smux; ————— module name declaration
wire OUT;
reg A,B,SEL;
```

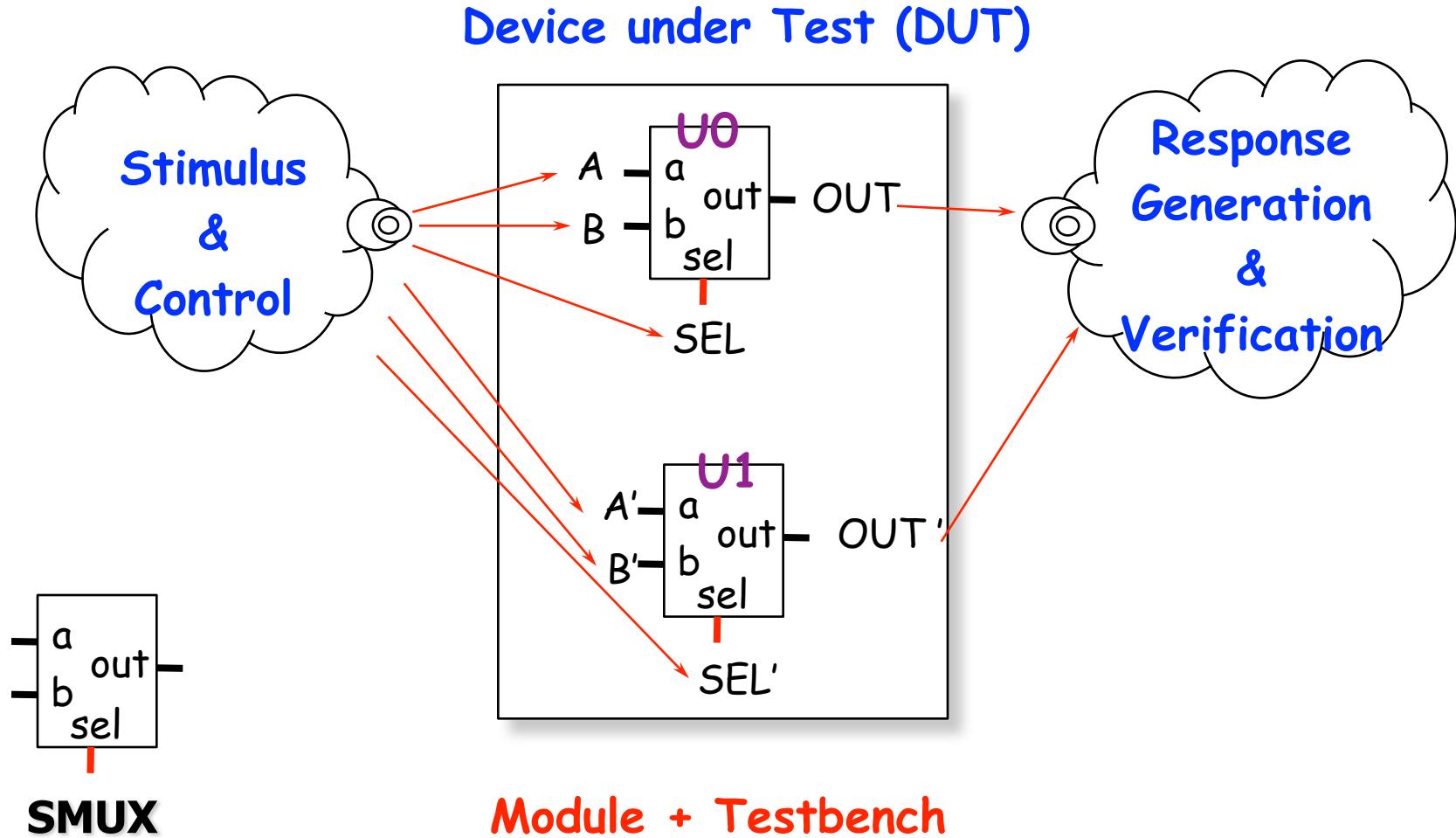
module instantiation

```
smux U0(.out(OUT),.a(A),.b(B),.sel(SEL));
```

apply stimulus

```
initial
begin
A=0;B=0;SEL=0;
#10 A=0;B=0;SEL=1;
#10 A=0;B=1;SEL=0;
#10 A=0;B=1;SEL=1;
#10 A=1;B=0;SEL=0;
#10 A=1;B=0;SEL=1;
#10 A=1;B=1;SEL=0;
#10 A=1;B=1;SEL=1;
end
endmodule
```

# Verification Scenario



# Event Simulation of a Verilog Model

- **Compilation**
  - Compilation and elaboration
- **Initialization**
  - Initialize module parameters
    - Set other storage element to unknown (X) state
      - Unknown or un-initialized
    - Set undriven nets to the high-impedance (Z) state
      - Tri-state or floating
- **Simulation**

# RTL Modeling

# Register Transfer Level (RTL) Models

- Represent digital functionality using programming statements and operators
- Functional models that do not contain details on implementation in silicon
- Two primary constructs
  - continuous assignments
    - begin with an **assign** keyword, and can represent simple combinational logics
  - **always** procedure blocks
    - A procedure block encapsulates one or more lines or programming statements, along with information about when the statements should be executed

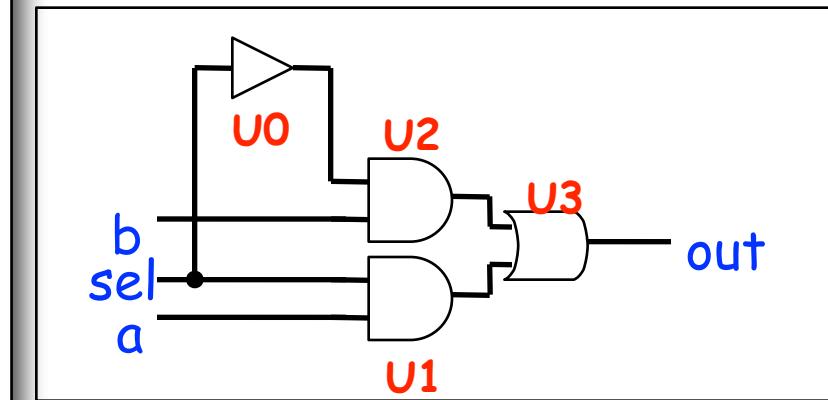
# Continuous Assignments

- **assign** continuous construct
  - combinational logics

```
module SMUX (out,a,b,sel);
output out;
input a,b,sel;

assign out = (a&sel) | (b&(~sel));

endmodule
```



This **out** has to be declared as “wire” or “output” data type.  
This expression can not be inside **always @()**.

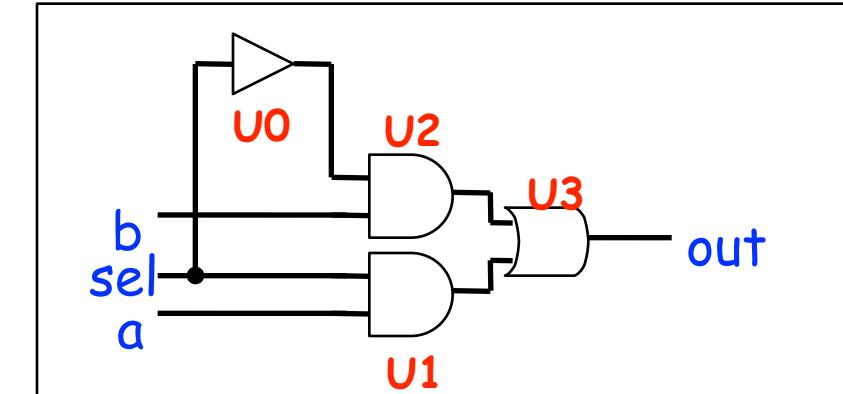
# always Procedure Block

- **always** statements

```
module SMUX (out,s,b,sel);
output out;
input a,b,sel;
reg out;

always @*
  out = (a&sel) | (b&(~sel));

endmodule
```



This **out** has to be declared as “**reg**” data type.

# Operators (1/3)

Bitwise Operators		
OP	Usage	Description
$\sim$	$\sim m$	Invert each bit of $m$
$\&$	$m \& n$	AND each bit of $m$ with each bit of $n$
$ $	$m   n$	OR each bit of $m$ with each bit of $n$
$\wedge$	$m \wedge n$	Exclusive OR each bit of $m$ with $n$
$\sim\wedge$ or $\wedge\sim$	$m \sim\wedge n$ or $m \wedge\sim n$	Exclusive NOR each bit of $m$ with $n$

Unary Reduction Operators		
OP	Usage	Description
$\&$	$\&m$	AND all bits in $m$ together (1-bit result)
$\sim\&$	$\sim\&m$	NAND all bits in $m$ together (1-bit result)
$ $	$ m$	OR all bits in $m$ together (1-bit result)
$\sim $	$\sim m$	NOR all bits in $m$ together (1-bit result)
$\wedge$	$\wedge m$	Exclusive OR all bits in $m$ (1-bit result)
$\sim\wedge$ or $\wedge\sim$	$\sim\wedge m$ or $\wedge\sim m$	Exclusive NOR all bits in $m$ (1-bit result)

# Operators (2/3)

Arithmetic Operators		
OP	Usage	Description
+	$m + n$	Add n to m
-	$m - n$	Subtract n from m
-	$-m$	Negate m (2's complement)
*	$m * n$	Multiply m by n
/	$m / n$	Divide m by n
%	$m \% n$	Modulus of m / n

Logical Operators		
OP	Usage	Description
!	$!m$	Is m not true? (1-bit True/False result)
&&	$m \&\& n$	Are both m and n true? (1-bit True/False result)
	$m    n$	Are either m or n true? (1-bit True/False result)

The divisor for divide operator may be restricted to constants and a power of 2

Equality Operators (compares logic values of 0 and 1)		
OP	Usage	Description
==	$m == n$	Is m equal to n? (1-bit True/False result)
!=	$m != n$	Is m not equal to n? (1-bit True/False result)

Identity Operators (compares logic values of 0, 1, x, and z)		
OP	Usage	Description
==	$m === n$	Is m identical to n? (1-bit True/False result)
!=	$m !== n$	Is m not identical to n? (1-bit True/False result)

Synthesis not supported

Synthesis not supported

# Operators (3/3)

Relational Operators		
OP	Usage	Description
<	$m < n$	Is m less than n? (1-bit True/False result)
>	$m > n$	Is m greater than n? (1-bit True/False result)
$\leq$	$m \leq n$	Is m less than or equal to n? (True/False result)
$\geq$	$m \geq n$	Is m greater than or equal to n? (True/False result)

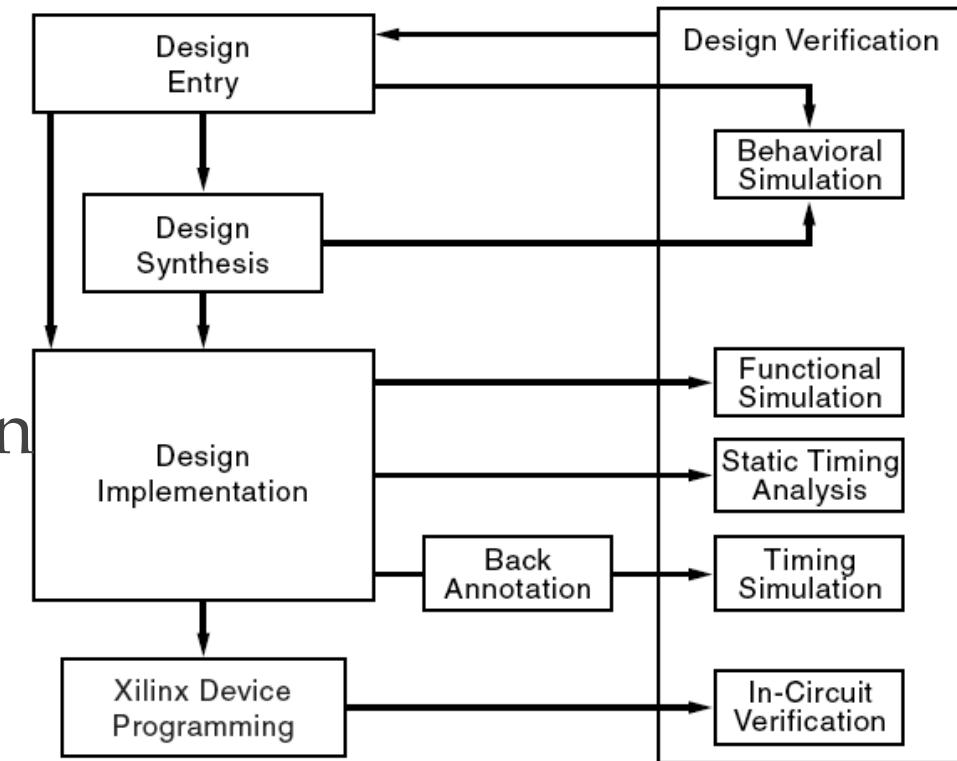
Logical Shift Operators		
OP	Usage	Description
$\ll$	$m \ll n$	Shift m left n-times
$\gg$	$m \gg n$	Shift m right n-times

Misc Operators		
OP	Usage	Description
$:$	$sel?m:n$	If sel is true, select m: else select n
{}	{m,n}	Concatenate m to n, creating larger vector
{}	{n{m}}	Replicate m n-times

# Logic Modeling and Simulation Using Xilinx Vivado

# Design Flow

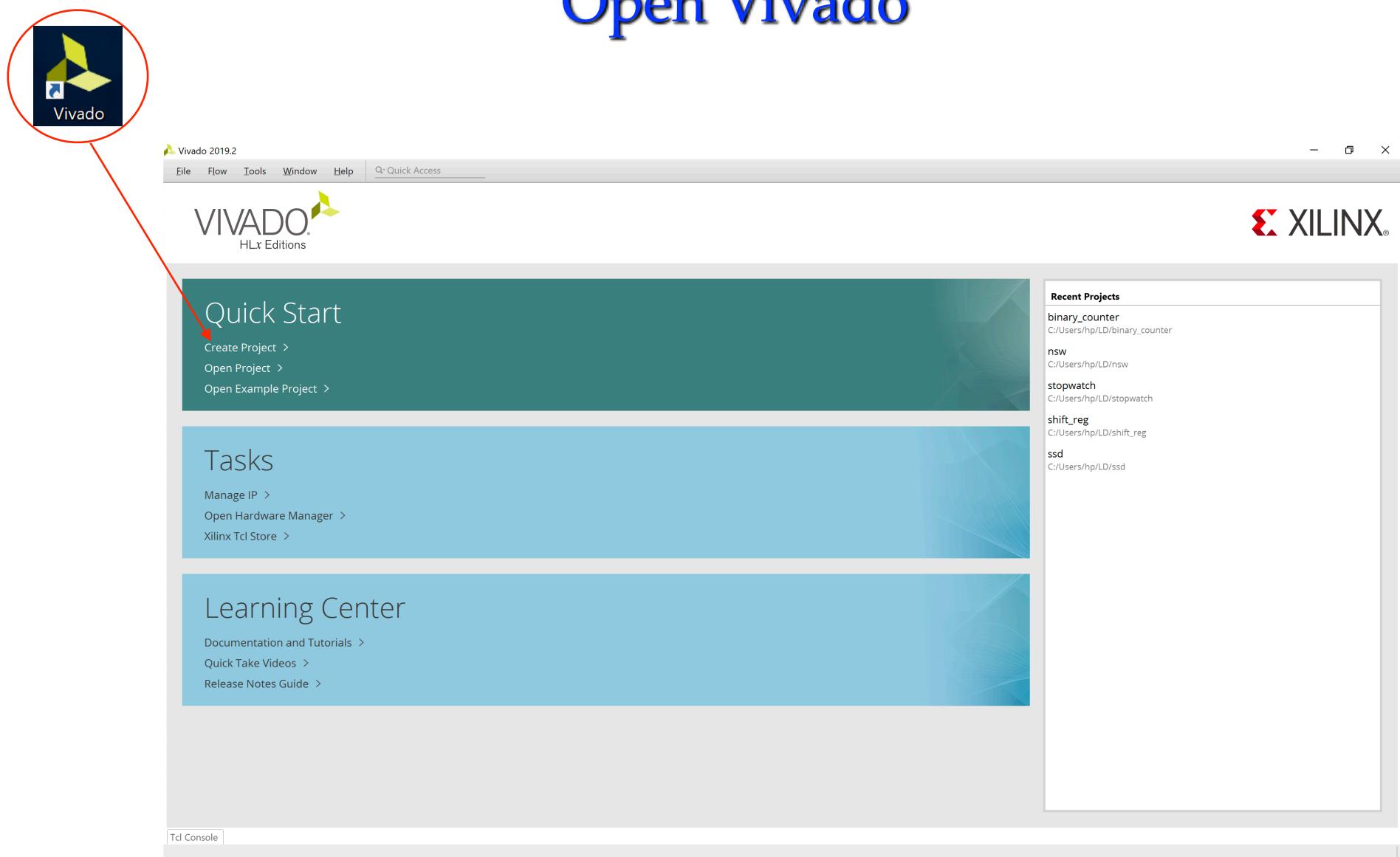
- General design flow
  - Design construction
  - Behavioral simulation
  - Design implementation
  - Timing simulation
- HDL-based design Flow



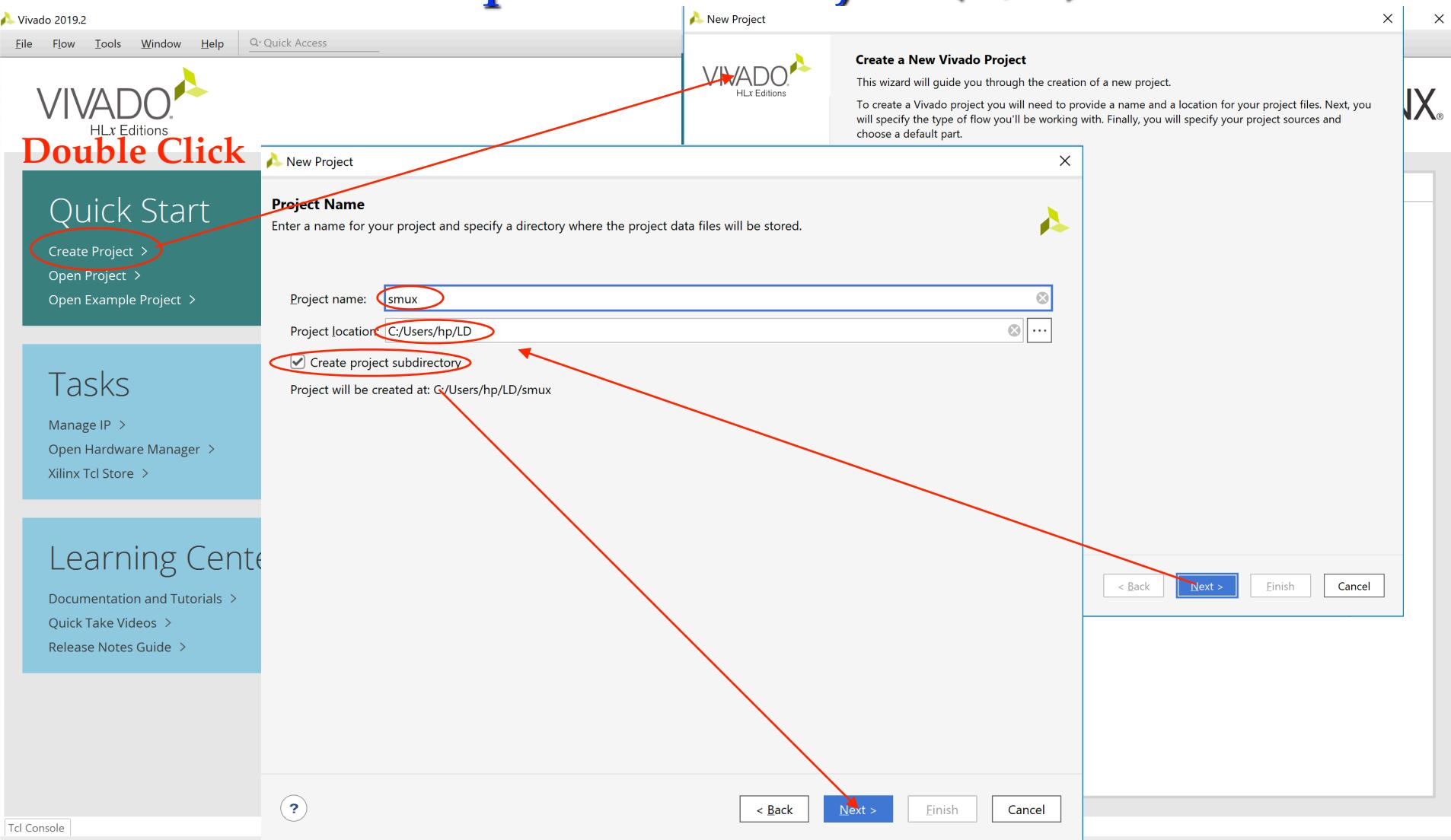
# Important Notes

- **Draw schematic first** and then construct Verilog codes.
- Verilog RTL coding philosophy is not the same as C programming
  - Every Verilog RTL construct has its own logic mapping (for synthesis)
  - You should have the logics (draw schematic) first and then the RTL codes
  - You have to write **synthesizable** RTL codes

# Open Vivado



# Open New Project (1/3)



# Open New Project (2/3)

New Project

**Project Type**  
Specify the type of project to create.

**RTL Project**  
You will be able to add sources, create block designs in IP Integrator and analysis.  
 Do not specify sources at this time

**Post-synthesis Project:** You will be able to add sources, view design reports.  
 Do not specify sources at this time

**I/O Planning Project**  
Do not specify design sources. You will be able to view part/package information.

**Imported Project**  
Create a Vivado project from a Synplify, XST or ISE Project File.

**Example Project**  
Create a new Vivado project from a predefined template.

**New Project**

**Default Part**  
Choose a default Xilinx part or board for your project. This can be changed later.

Select:  Parts  Boards

Filter

Product category: All Speed grade: -1  
Family: Artix-7 Temp grade: All Remaining  
Package: cpg236

Reset All Filters

Search: xc7a35tcp (1 match)

Part	I/O Pin Count	Block RAMs	DSPs	FlipFlops	GTPE2 Transceivers	Gb Transceivers	Available IOBs	LUT Elements
xc7a35tcpg236-1	236	50	90	41600	2	2	106	20800

**New Project Summary**

A new RTL project named 'lab1' will be created.

The default part and product family for the new project:  
Default Part: xc7a35tcpg236-1  
Product: Artix-7  
Family: Artix-7  
Package: cpg236  
Speed Grade: -1

XILINX ALI PROGRAMMABLE

To create the project, click Finish

Finish

?

< Back  Next >  Cancel

# Open New Project (3/3)

The screenshot shows the Vivado 2019.2 Project Manager interface with the 'smux' project open. A red oval highlights the 'Settings' section under 'Project Summary'.

**Project Summary**

**Settings**

- Project name: smux
- Project location: C:/Users/hp/LD/smux
- Product family: Artix-7
- Project part: xc7a35tcpg236-1
- Top module name: Not defined
- Target language: Verilog
- Simulator language: Mixed

**Synthesis**

Status: Not started  
Messages: No errors or warnings  
Part: xc7a35tcpg236-1  
Strategy: Vivado Synthesis Defaults  
Report Strategy: Vivado Synthesis Default Reports  
Incremental synthesis: None

**Implementation**

Status: Not started  
Messages: No errors or warnings  
Part: xc7a35tcpg236-1  
Strategy: Vivado Implementation Defaults  
Report Strategy: Vivado Implementation Default Reports  
Incremental implementation: None

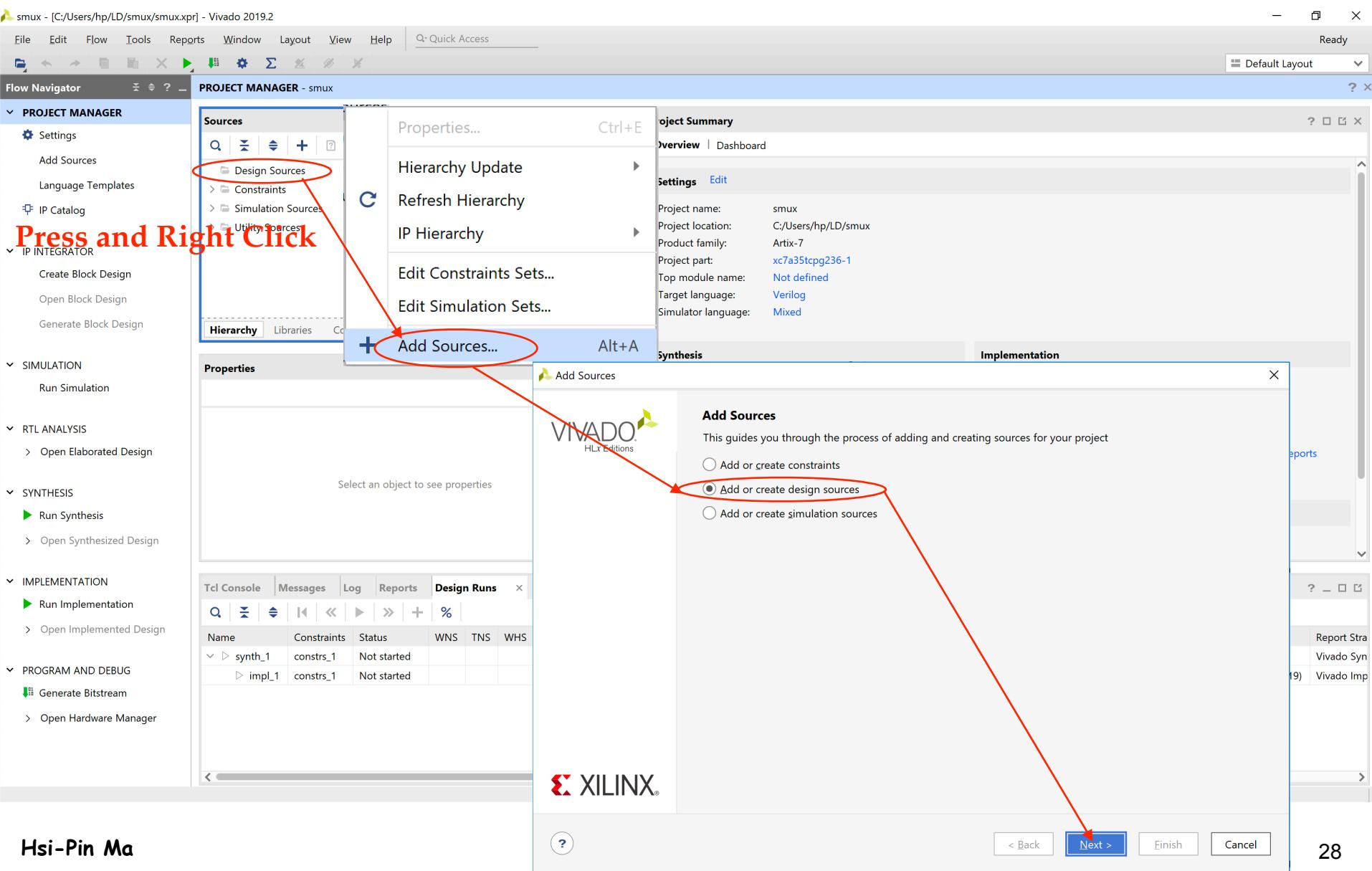
**DRC Violations**

Run Implementation to see DRC results

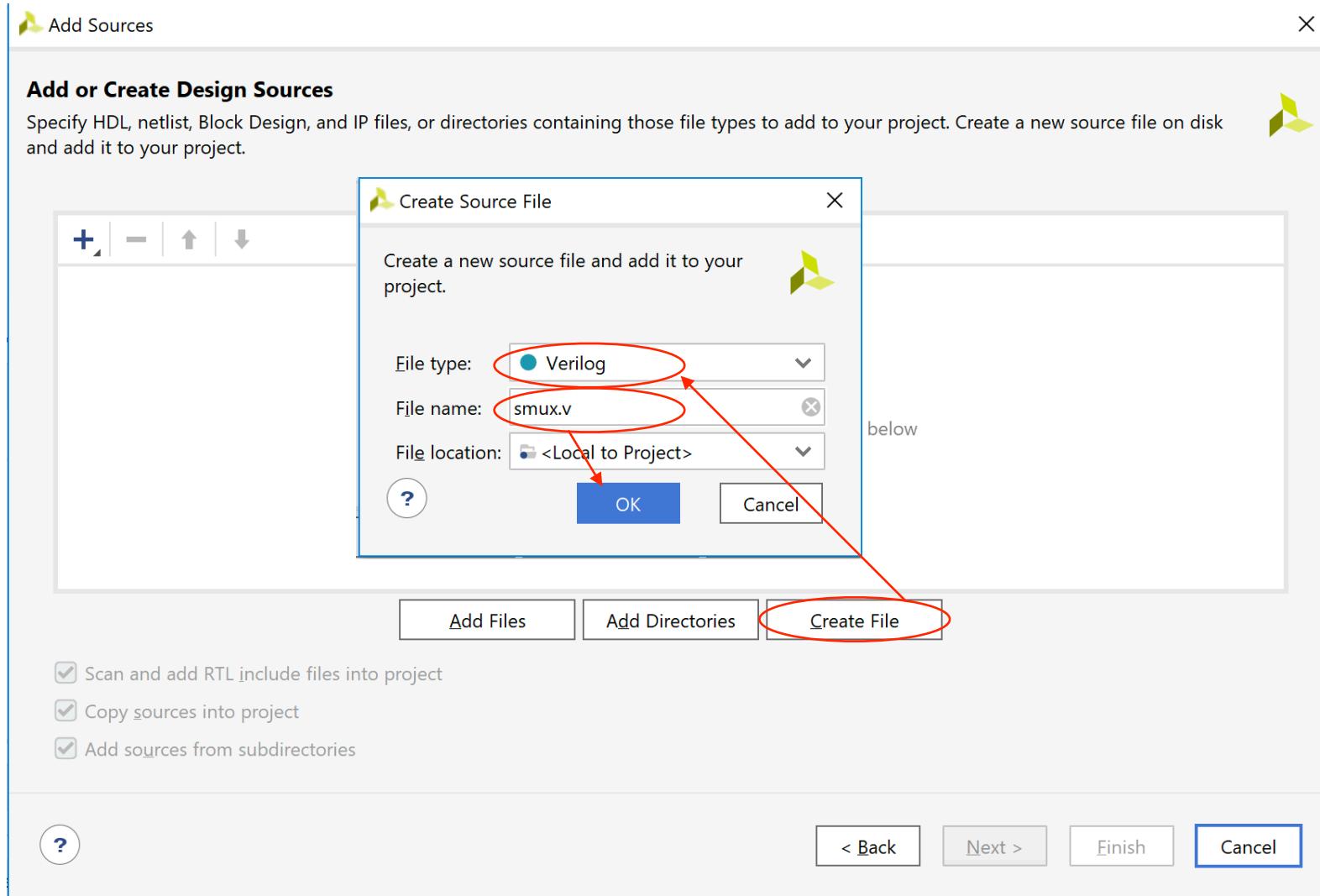
**Design Runs**

Name	Constraints	Status	WNS	TNS	WHS	THS	TPWS	Total Power	Failed Routes	LUT	FF	BRAM	URAM	DSP	Start	Elapsed	Run Strategy	Report Stra
synth_1	consts_1	Not started															Vivado Synthesis Defaults (Vivado Synthesis 2019)	Vivado Syn
impl_1	consts_1	Not started															Vivado Implementation Defaults (Vivado Implementation 2019)	Vivado Imp

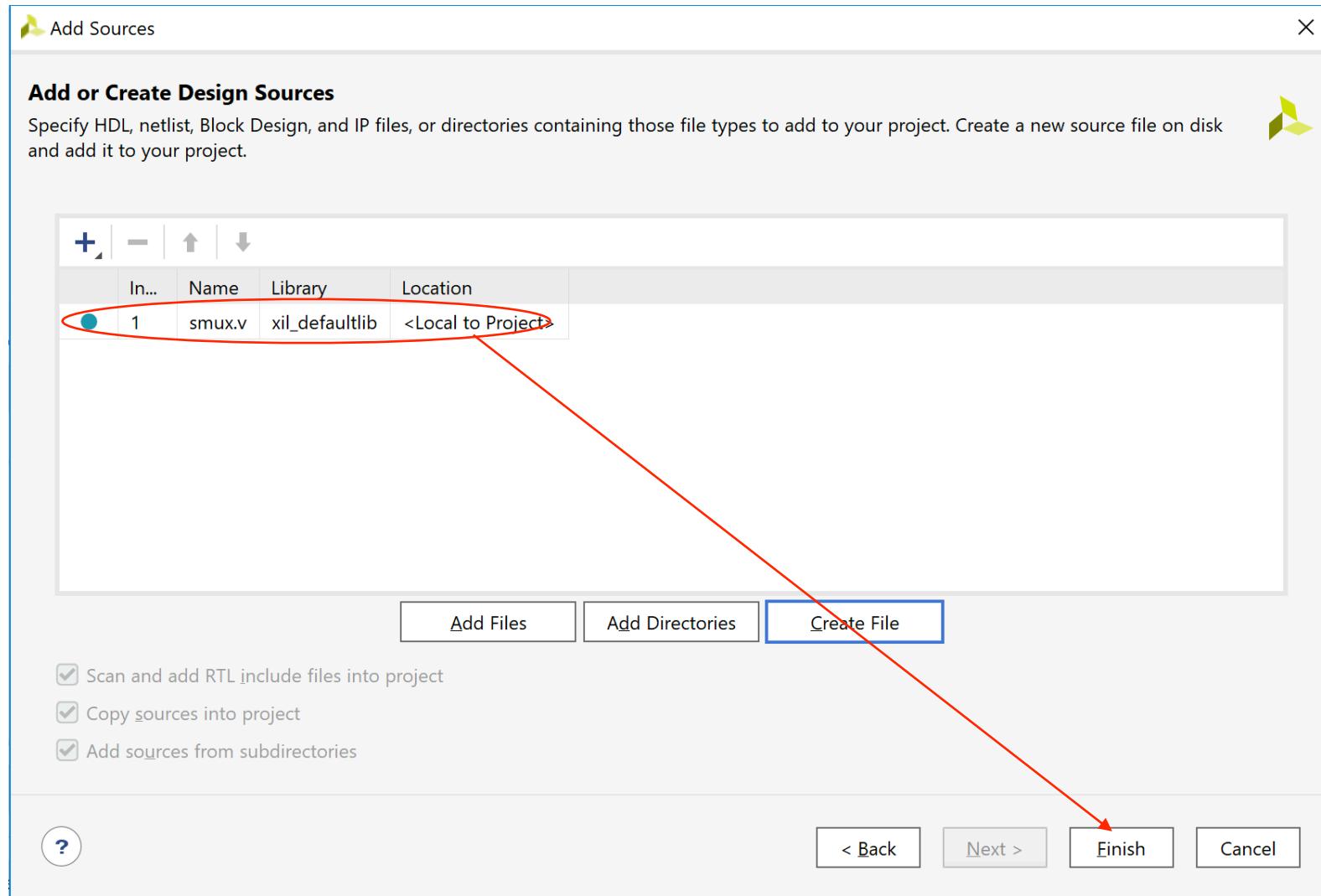
# New Source (1/5)



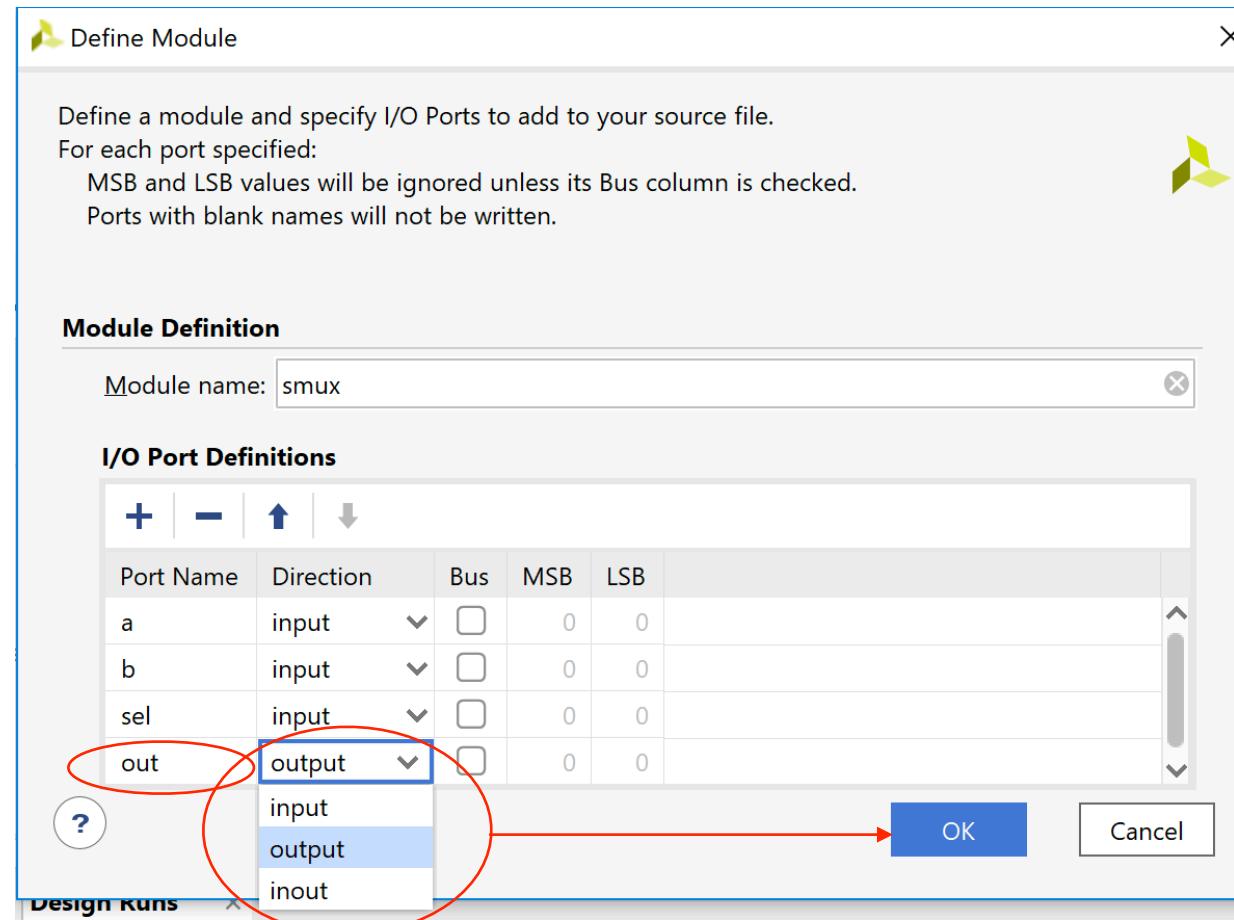
# New Source (2/5)



# New Source (3/5)



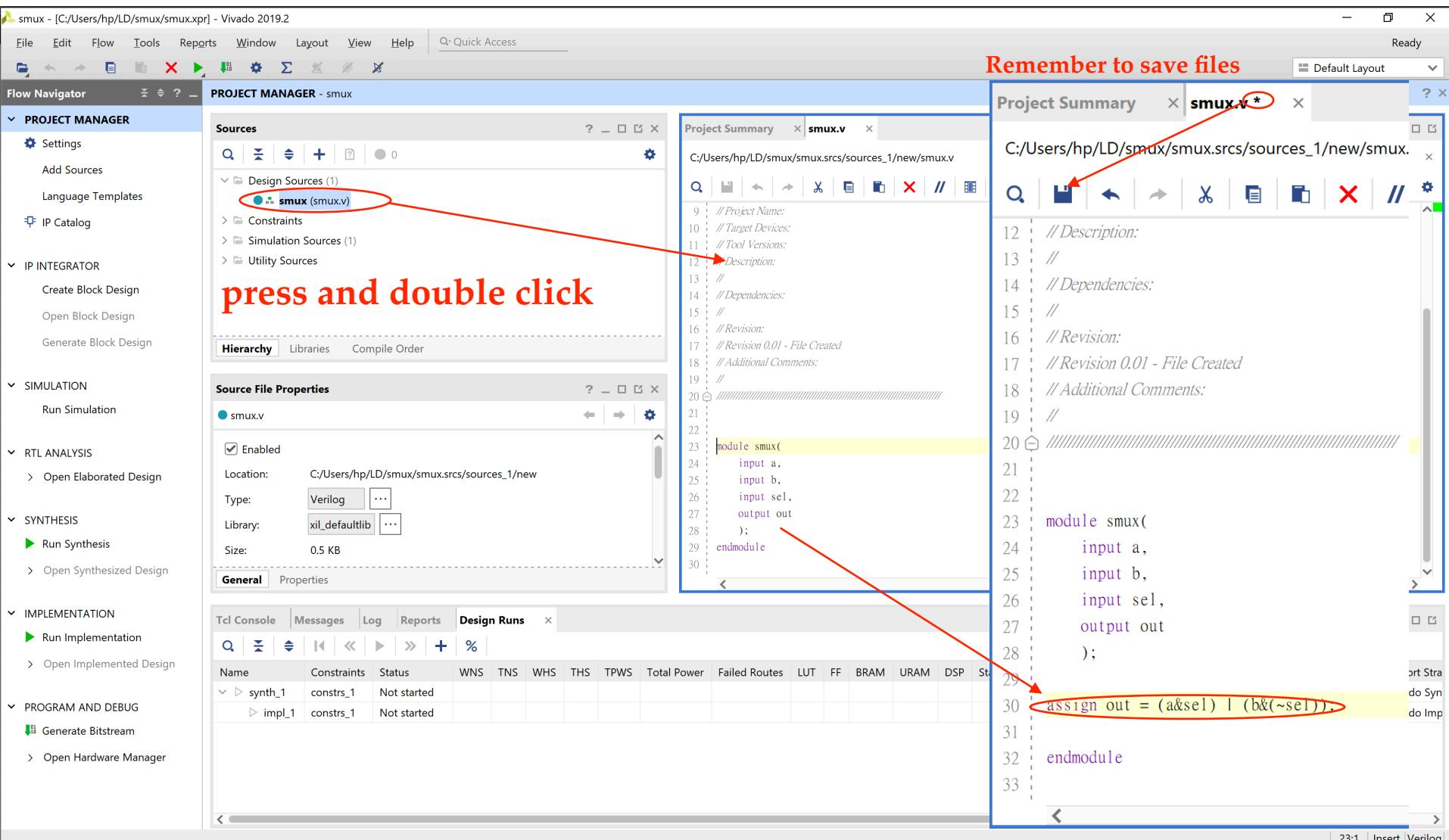
# New Source (4/5)



# New Source (5/5)

smux - [C:/Users/hp/LD/smux.smux.xpr] - Vivado 2019.2

File Edit Flow Tools Reports Window Layout View Help Q Quick Access Ready

Flow Navigator X 

PROJECT MANAGER - smux

Sources

Design Sources (1) **smux (smux.v)**

Constraints

Simulation Sources (1)

Utility Sources

Hierarchy Libraries Compile Order

Source File Properties

smux.v

Enabled

Location: C:/Users/hp/LD/smux/smux.srcs/sources\_1/new

Type: Verilog

Library: xil\_defaultlib

Size: 0.5 KB

General Properties

Tcl Console Messages Log Reports Design Runs

Name Constraints Status WNS TNS WHS THS TPWS Total Power Failed Routes LUT FF BRAM URAM DSP St

synth\_1 constrs\_1 Not started

impl\_1 constrs\_1 Not started

Project Summary x smux.v x

C:/Users/hp/LD/smux/smux.srcs/sources\_1/new/smux.v

```
// Project Name:  
// Target Devices:  
// Tool Versions:  
// Description:  
// Dependencies:  
// Revision:  
// Revision 0.01 - File Created  
// Additional Comments:  
//  
module smux(  
    input a,  
    input b,  
    input sel,  
    output out  
);  
endmodule
```

Remember to save files

Project Summary x smux.v \* x

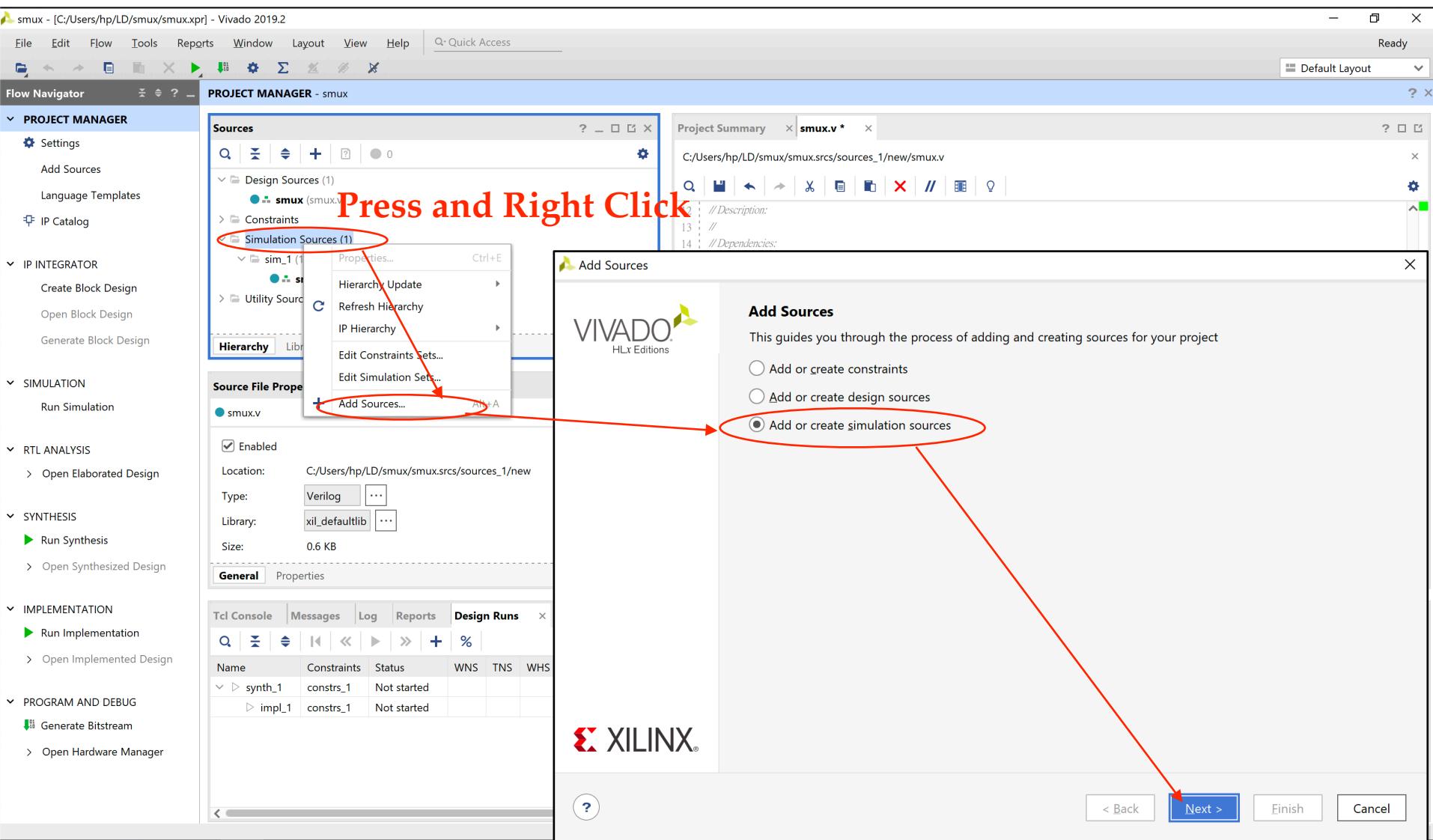
C:/Users/hp/LD/smux/smux.srcs/sources\_1/new/smux.v

```
// Description:  
//  
// Dependencies:  
//  
// Revision:  
// Revision 0.01 - File Created  
// Additional Comments:  
//  
module smux(  
    input a,  
    input b,  
    input sel,  
    output out  
);  
assign out = (a&sel) | (b&(~sel));  
endmodule
```

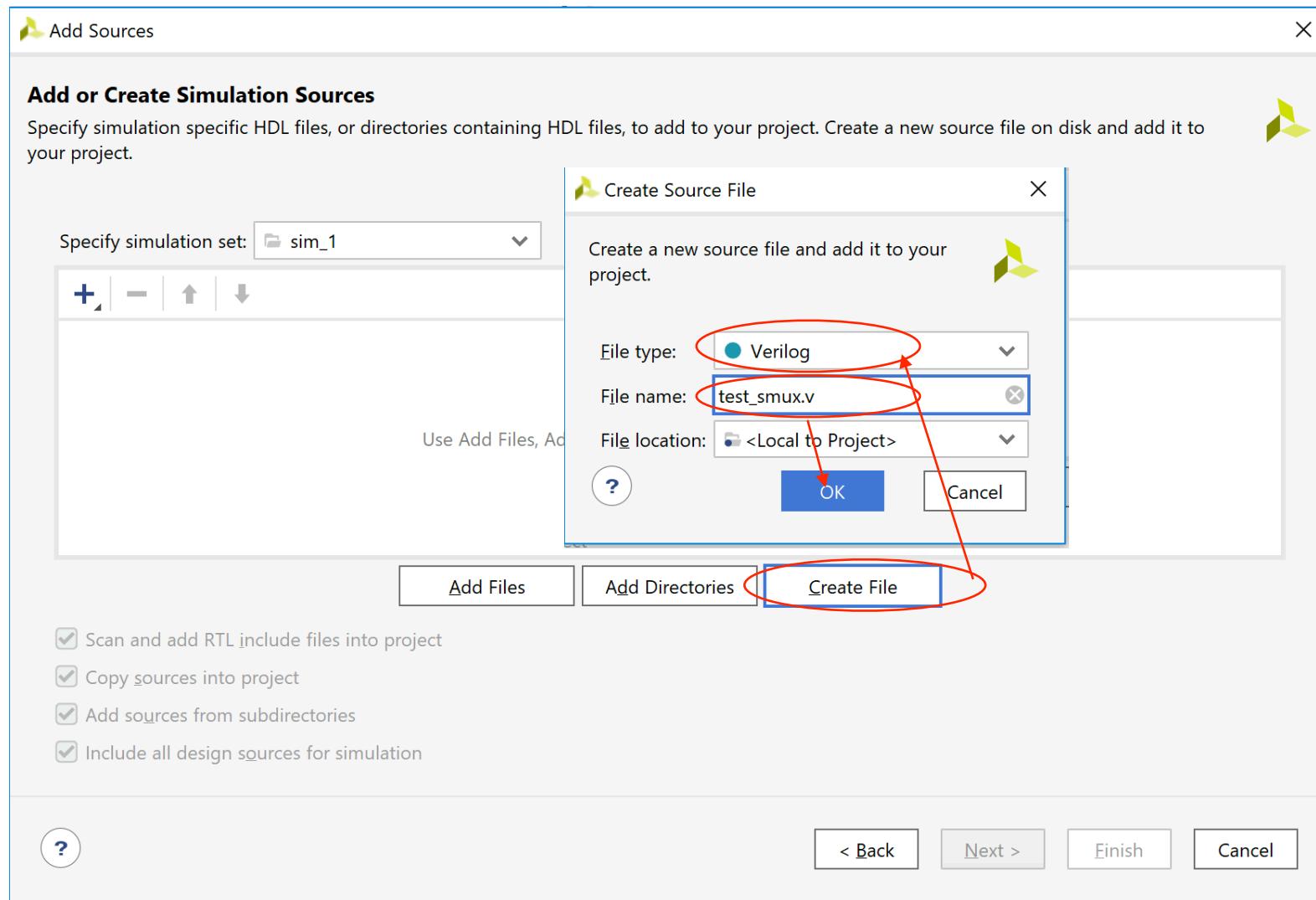
Default Layout ?

press and double click

# Add Testbench (1/6)



# Add Testbench (2/6)



# Add Testbench (3/6)

The screenshot shows the Xilinx Vivado interface. On the left, the 'Add Sources' window is open, displaying the 'Add or Create Simulation Sources' section. It shows a simulation set named 'sim\_1' and a table with one row: 'test\_smux.v' (Name), 'xil\_defaultlib' (Library), and '<Local to Project>' (Location). A red oval highlights the first row. Below the table are buttons for 'Add Files', 'Add Directories', and 'Create File' (which is highlighted with a blue border). At the bottom are checkboxes for project settings: 'Scan and add RTL include files into project', 'Copy sources into project', 'Add sources from subdirectories', and 'Include all design sources for simulation'. On the right, the 'Define Module' dialog box is open. It contains instructions for defining a module and specifying I/O ports. The 'Module Definition' section shows 'Module name: test\_smux'. The 'I/O Port Definitions' section has a single row: 'Port Name: input', 'Direction: input', 'Bus: 0', 'MSB: 0', and 'LSB: 0'. The 'OK' button at the bottom right of the dialog box is circled in red. Red arrows point from the circled 'OK' button to the 'Create File' button in the main window and to the 'Finish' button at the bottom right of the main window.

Add Sources

Add or Create Simulation Sources

Specify simulation set: sim\_1

In...	Name	Library	Location
1	test_smux.v	xil_defaultlib	<Local to Project>

Add Files Add Directories Create File

Scan and add RTL include files into project

Copy sources into project

Add sources from subdirectories

Include all design sources for simulation

?

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: test\_smux

I/O Port Definitions

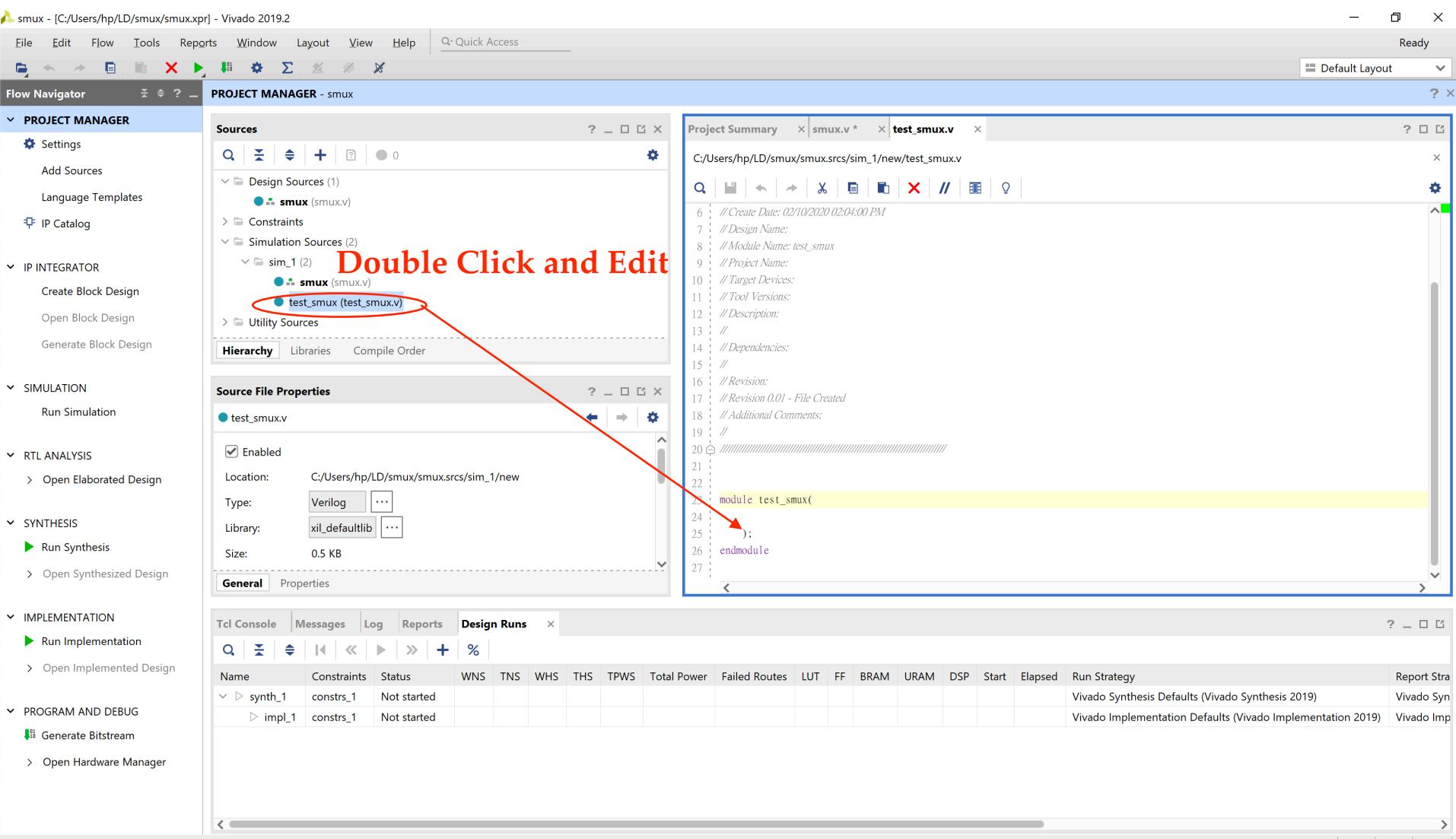
Port Name	Direction	Bus	MSB	LSB
input	input	0	0	0

?

OK Cancel

< Back Next > Finish Cancel

# Add Testbench (4/6)



# Add Testbench (5/6)

The screenshot shows the Vivado 2019.2 interface with the project 'smux' open. The 'PROJECT MANAGER' pane on the left lists various design phases: PROJECT MANAGER, IP Catalog, IP INTEGRATOR, SIMULATION, RTL ANALYSIS, SYNTHESIS, IMPLEMENTATION, and PROGRAM AND DEBUG. In the 'SIMULATION' section, 'Run Simulation' is selected. The 'Sources' tab in the project manager shows 'Design Sources (1)' containing 'smux (smux.v)'. Under 'Simulation Sources (2)', there is a folder 'sim\_1 (2)' containing 'smux (smux.v)' and 'test\_smux (test\_smux.v)'. The 'Source File Properties' for 'test\_smux.v' are displayed, showing it is enabled, located at 'C:/Users/hp/LD/smux/smux.srcts/sim\_1/new', is a Verilog file, and belongs to the 'xil\_defaultlib' library. The 'Design Runs' tab shows two runs: 'synth\_1' and 'impl\_1', both of which have not started. The main workspace displays the Verilog code for 'test\_smux.v':

```
21
22
23 module test_smux();
24     wire OUT;
25     reg A, B, SEL;
26
27     smux U0(.a(A), .b(B), .sel(SEL), .out(OUT));
28
29 initial
30 begin
31
32     A=0;B=0;SEL=0;
33     #10 A=0;B=0;SEL=1;
34     #10 A=0;B=1;SEL=0;
35     #10 A=0;B=1;SEL=1;
36     #10 A=1;B=0;SEL=0;
37     #10 A=1;B=0;SEL=1;
38     #10 A=1;B=1;SEL=0;
39     #10 A=1;B=1;SEL=1;
40     #10 A=0;B=0;SEL=0;
41 end
42
```

The code from line 34 to line 39 is highlighted with a red rectangle, indicating the portion being modified or added.

# Add Testbench (6/6)

The screenshot shows the Vivado 2021.2 Project Manager interface for a project named "smux".

**PROJECT MANAGER - smux**

- Sources**: Contains "smux (smux.v)".
  - Constraints**
  - Simulation Sources (1)**: Contains "sim\_1 (1)" which includes "test\_smux (test\_smux.v)" (circled in red).
  - Utility Sources**
- Hierarchy** and **Libraries** tabs.
- Compile Order** tab.

**Project Summary** for smux.v

Project name:	smux
Project location:	C:/Users/hp/LD/smux
Product family:	Artix-7
Project part:	xc7a35tcpg236-1
Top module name:	smux
Target language:	Verilog
Simulator language:	Mixed

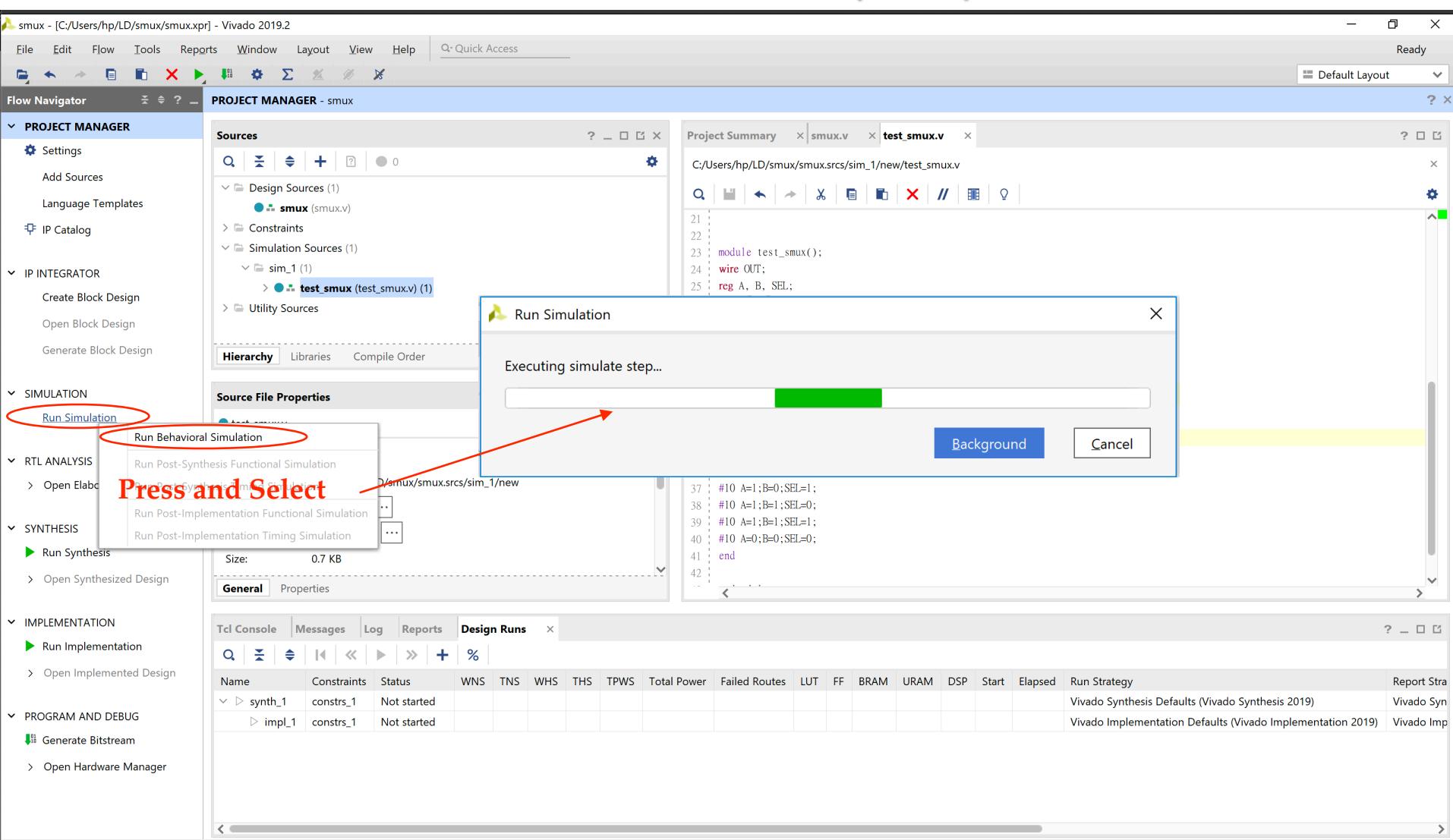
**Synthesis** and **Implementation** sections show initial status.

**Tcl Console**, **Messages**, **Log**, **Reports**, and **Design Runs** tabs.

**Design Runs** table:

Name	Constraints	Status	WNS	TNS	WHS	THS	WBSS	TPWS	Total Power	Failed Routes	LUT	FF	BRAM	URAM	DSP	Start	Elapsed	Run Strategy	Report Str
synth_1	constrs_1	Not started																Vivado Synthesis Defaults (Vivado Synthesis 2021)	Vivado Syn
impl_1	constrs_1	Not started																Vivado Implementation Defaults (Vivado Implementation 2021)	Vivado Imp

# Simulation (1/4)



# Simulation (2/4)

smux - [C:/Users/hp/LD/smux/smux.xpr] - Vivado 2019.2

File Edit Flow Tools Reports Window Layout View Run Help Q Quick Access

Flow Navigator SIMULATION - Behavioral Simulation - Functional - sim\_1 - test\_smux

PROJECT MANAGER

- Settings
- Add Sources
- Language Templates
- IP Catalog

IP INTEGRATOR

- Create Block Design
- Open Block Design
- Generate Block Design

SIMULATION

- Run Simulation

RTL ANALYSIS

- Open Elaborated Design

SYNTHESIS

- Run Synthesis
- Open Synthesized Design

IMPLEMENTATION

- Run Implementation
- Open Implemented Design

PROGRAM AND DEBUG

- Generate Bitstream
- Open Hardware Manager

Scope Sources Objects Protocol Instances

smux.v test\_smux.v Untitled 1

Press to maximize

Default Layout

Sim Time: 1 us

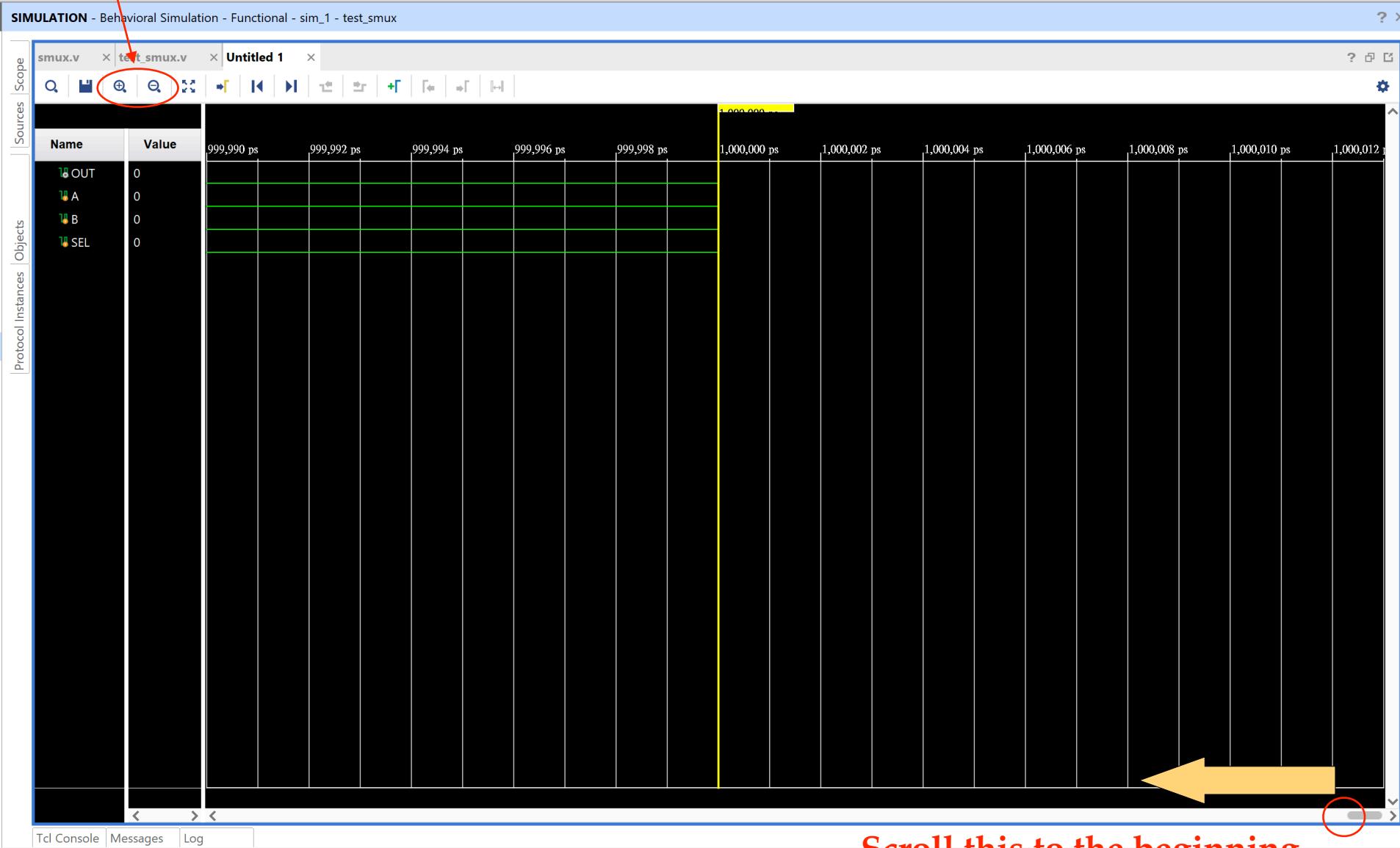
Tcl Console Messages Log

Type a Tcl command here

The screenshot shows the Vivado 2019.2 interface with the 'SIMULATION' tab selected. In the center, there's a waveform viewer titled 'Untitled 1' showing the behavior of a 'smux' component. The waveform displays four logic signals over time: OUT, A, B, and SEL. All signals start at 0. The OUT signal remains at 0 until approximately 999,990 ps, then transitions to 1. The SEL signal is high (1) from 999,990 ps to 999,998 ps, and low (0) otherwise. Signals A and B are both 0 throughout the entire simulation period. At the bottom, the 'Tcl Console' window shows the command 'run 1000ns' and the output of the XSim simulation, indicating it completed successfully. The top right corner of the waveform viewer has a red box and a red arrow pointing to the 'Press to maximize' button.

Use to zoom in or zoom out

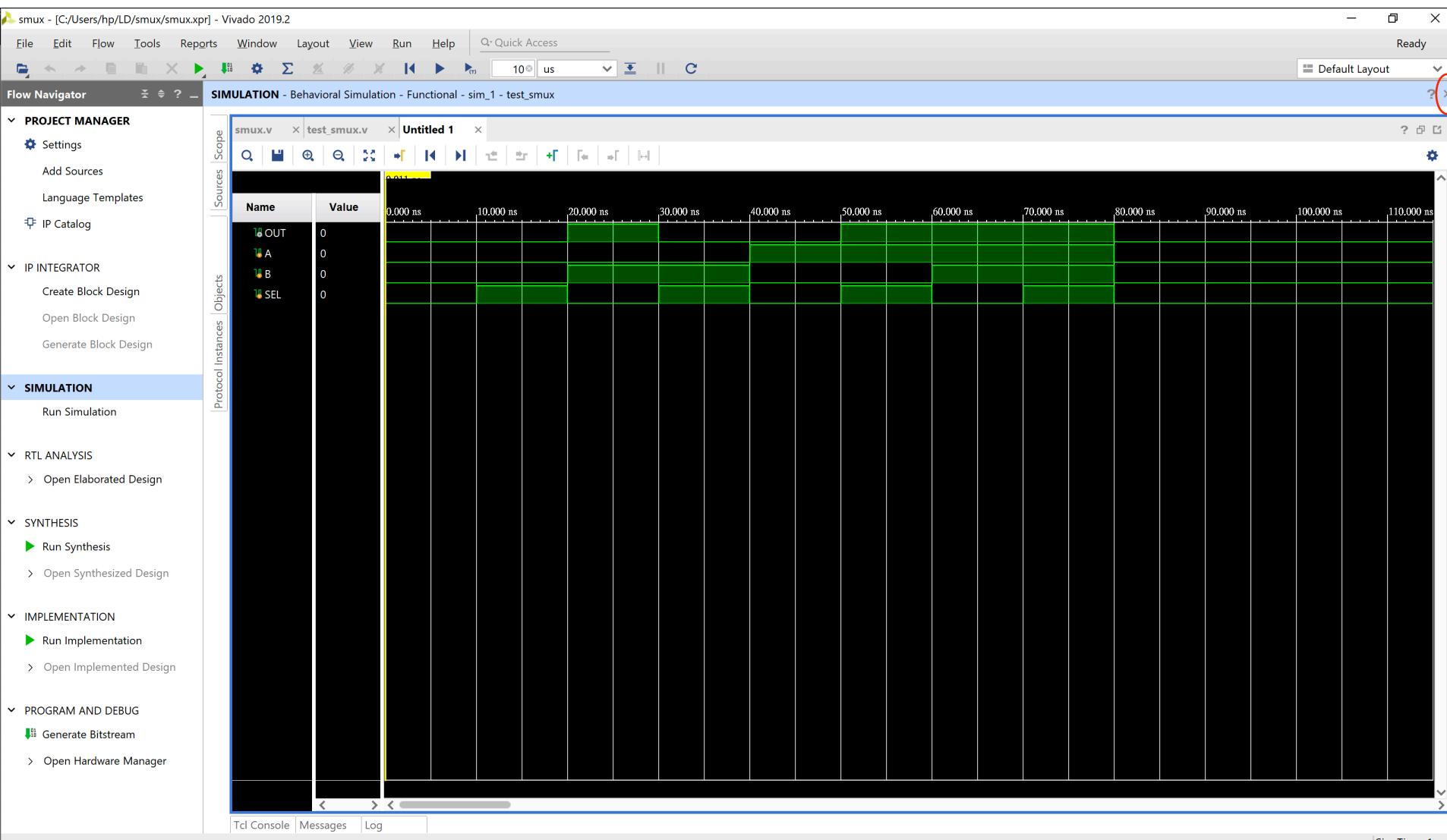
# Simulation (3/4)



Scroll this to the beginning

# Simulation (4/4)

Press to close simulation



# Types of Verilog Construction

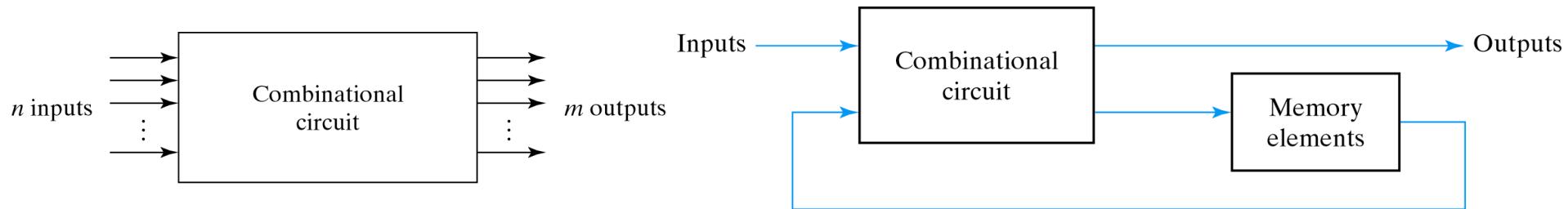
# Logic Circuits for the Digital System

- Combinational circuits

- Logic circuits whose outputs at any time are determined *directly* and *only* from the present *input combination*.

- Sequential circuits

- Circuits that employ memory elements + (combinational) logic gates
- Outputs are determined from the present input combination as well as the state of the memory cells.



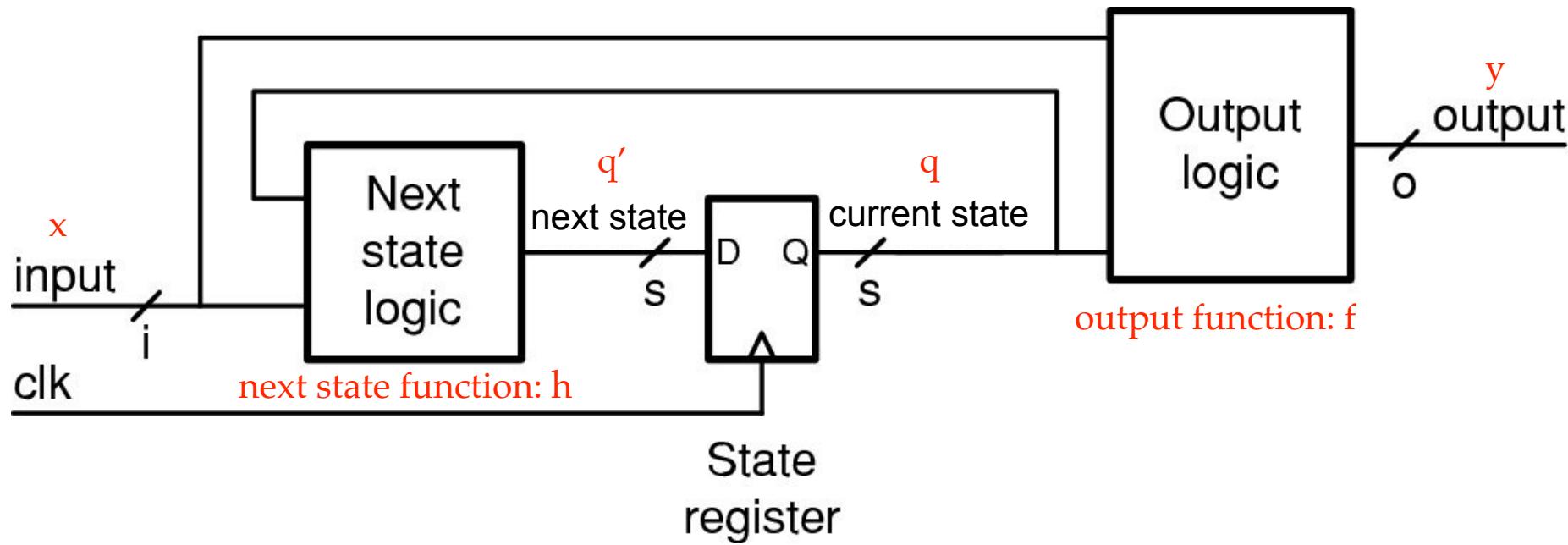
# Synchronous Sequential Circuits

- A synchronous sequential circuit can be modeled by a *finite state machine* (FSM).

$$M = (x, y, q, f, h)$$

- where  $f : x \times q \rightarrow y$  (Mealy machine) or  $f : q \rightarrow y$  (Moore machine) is the output function, and  $h : x \times q \rightarrow q'$  is the next-state function
- Note  $q'(t) = q(t + 1)$

# Model of Synchronous Sequential Circuits



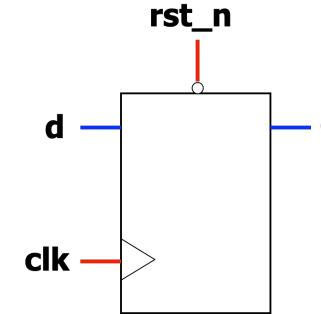
# Verilog Module Construction (1/2)

- Separate flip-flops with other logics (two types)
  - flip-flops (edge-triggered with clock, reset)
  - combinational logics (level sensitive)
- Combinational logics
  - simple logics (AND, OR, NOT)
  - coder/decoder (mapping, addressing)
  - comparison (conditional/equality test)
  - selection (select correct results, MUX)
  - arithmetic functions and superposition (+,-,\* ,binary shift)
- Synchronous sequential circuits / Finite state machine (FSM)

# Verilog Module Construction (2/2)

- Separate flip-flops with other logics
  - For a positive-edge-triggered D-type flip-flop with asynchronous reset

```
always @(posedge clk or negedge rst_n)
  if (~rst_n)
    q<=0;
  else
    q<=d;
```



# Some Combinational Logic Examples

$$F(x, y, z) = \sum(1, 4, 5, 6, 7) = f$$

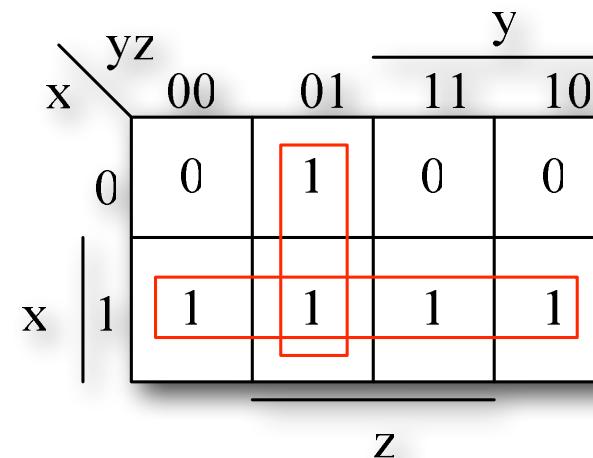
1

input: x,y,z

output: f

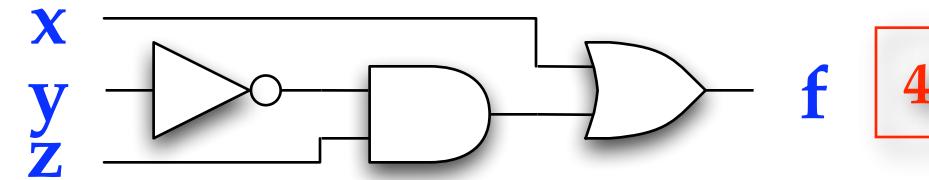
2

x	y	z	f
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1



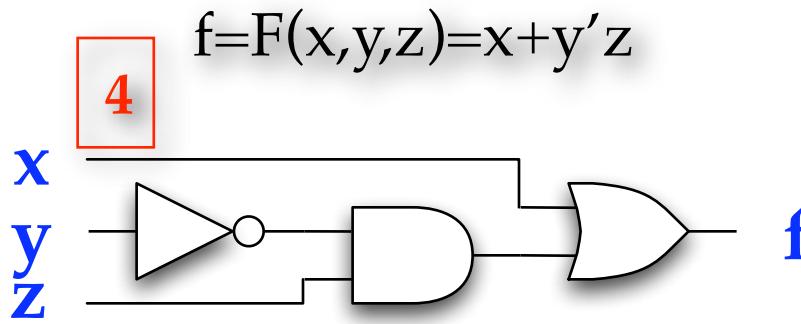
3

$$f = F(x, y, z) = x + y'z$$



4

$$F(x, y, z) = \sum(1, 4, 5, 6, 7) = f$$



5

```
module ex(f,x,y,z);
output f;
input x,y,z;

assign f = x | ((~y)&z);

endmodule
```

6

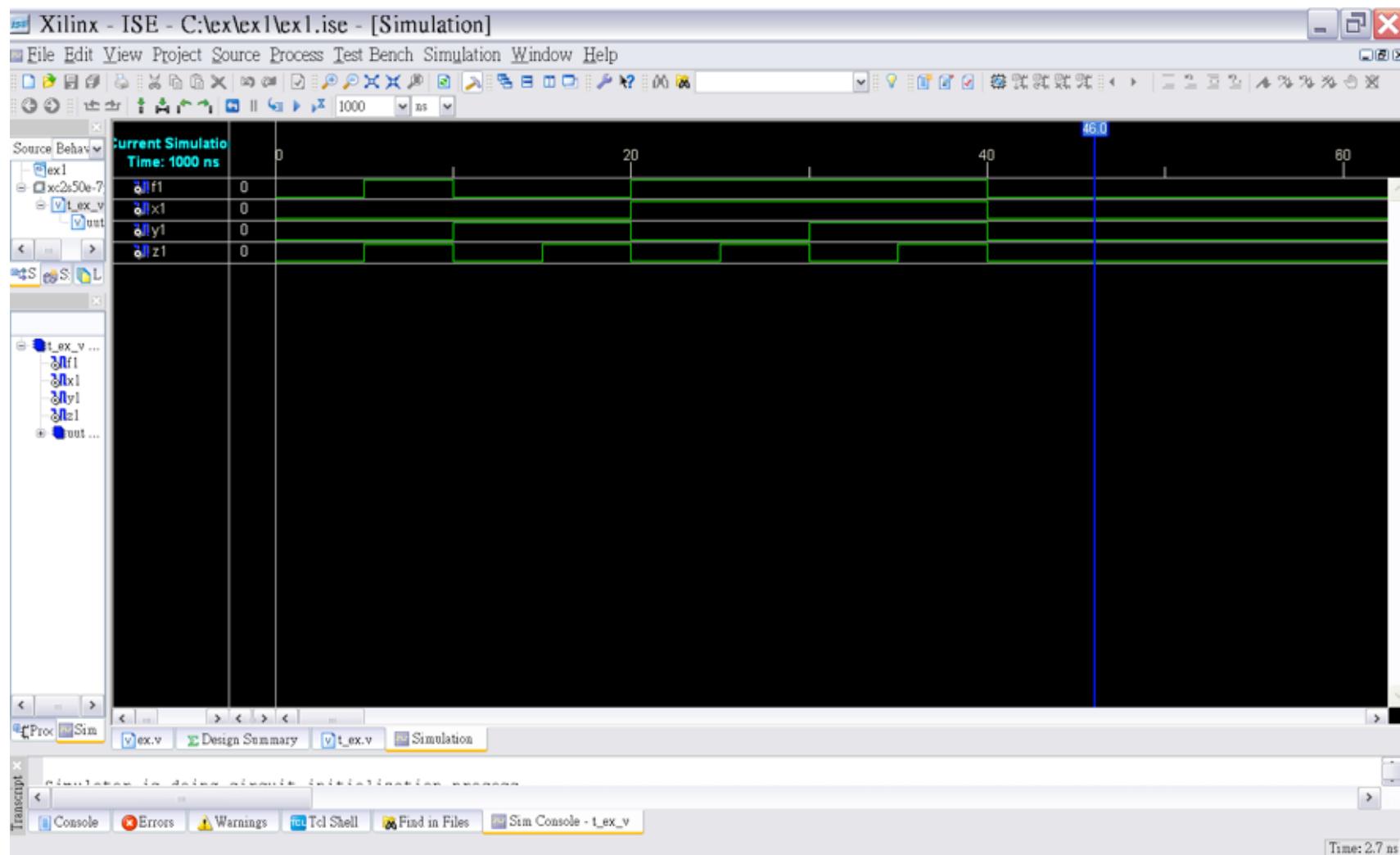
```
module t_ex;
wire f1;
reg x1,y1,z1;

ex U0(.f(f1),.x(x1),.y(y1),.z(z1));

initial
begin
x1=0;y1=0;z1=0;
#5 x1=0;y1=0;z1=1;
#5 x1=0;y1=1;z1=0;
#5 x1=0;y1=1;z1=1;
#5 x1=1;y1=0;z1=0;
#5 x1=1;y1=0;z1=1;
#5 x1=1;y1=1;z1=0;
#5 x1=1;y1=1;z1=1;
#5 x1=0;y1=0;z1=0;
end

endmodule
```

$$F(x, y, z) = \sum(1, 4, 5, 6, 7) = f$$



# Example 2 (1/3)

$$\begin{aligned}f_1 &= x'y'z + xy'z' + xyz \\&= m_1 + m_4 + m_7 = \sum (1, 4, 7)\end{aligned}$$

- Module definition

```
module exp1(f1, x, y, z);
output f1;
input x,y,z;

assign f1 = ((~x)&(~y)&z) | (x&(~y)&(~z) | x&y&z);

endmodule
```

x	y	z	f <sub>1</sub>
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

# Example 2 (2/3)

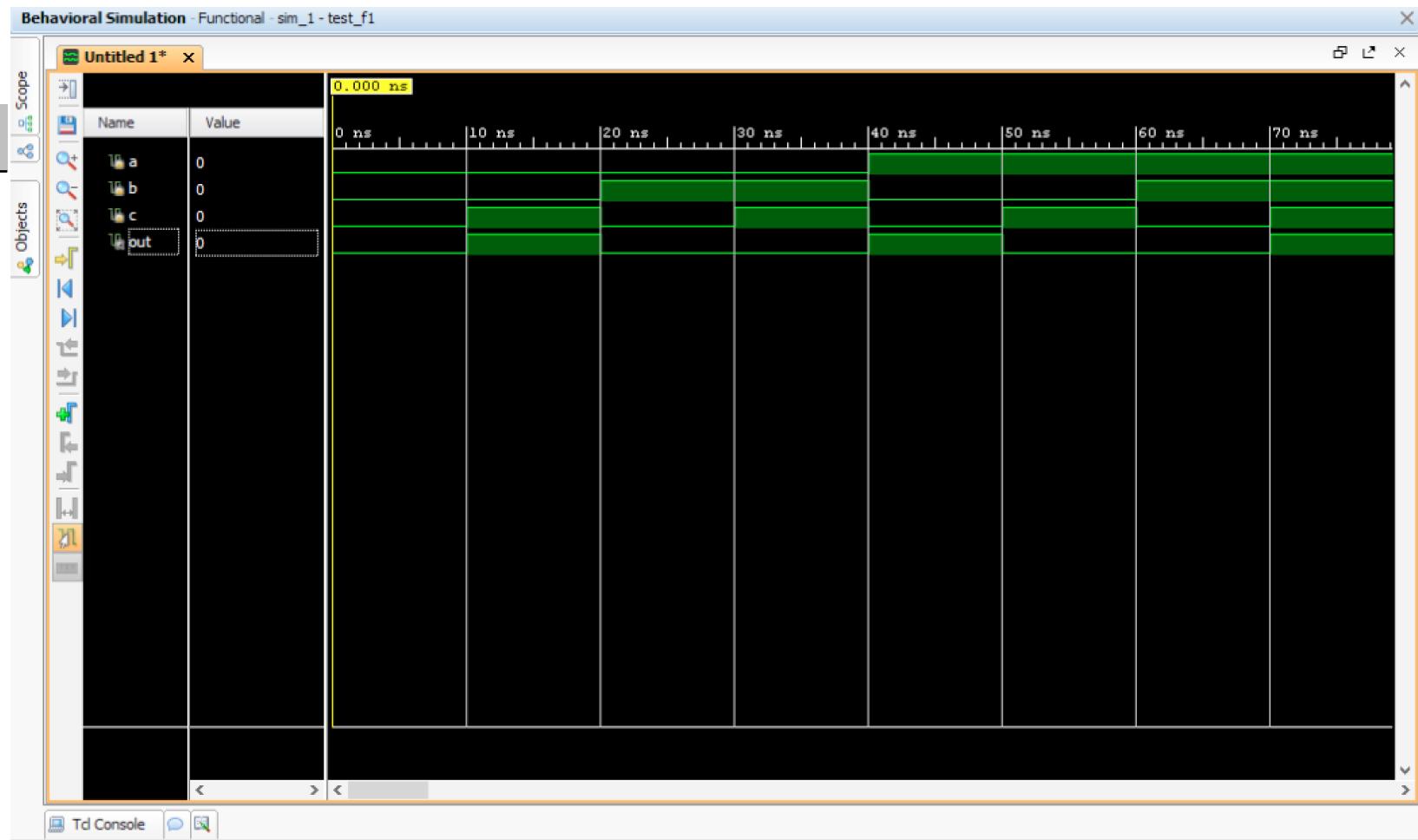
- Testbench

```
module test_f1;  
wire out;  
reg a,b,c;  
  
exp1 U0(.f1(out),.x(a),.y(b),.z(c));  
  
initial  
begin  
a=0;b=0;c=0;  
#10 a=0;b=0;c=1;  
#10 a=0;b=1;c=0;  
#10 a=0;b=1;c=1;  
#10 a=1;b=0;c=0;  
#10 a=1;b=0;c=1;  
#10 a=1;b=1;c=0;  
#10 a=1;b=1;c=1;  
end  
  
endmodule
```

x	y	z	f1
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

# Example 2 (3/3)

- Simulation Results



# Example 3 (1/3)

$$\begin{aligned}f_2 &= x'yz + xy'z + xyz' + xyz \\&= m_3 + m_5 + m_6 + m_7 = \sum (3, 5, 6, 7)\end{aligned}$$

- Module definition

```
module exp2(f2, x, y, z);
output f2;
input x,y,z;

assign f2 = ((~x)&y&z) | (x&(~y)&z) | ( x&y&(~z))| (x&y&z);

endmodule
```

x	y	z	f <sub>2</sub>
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

# Example 3 (2/3)

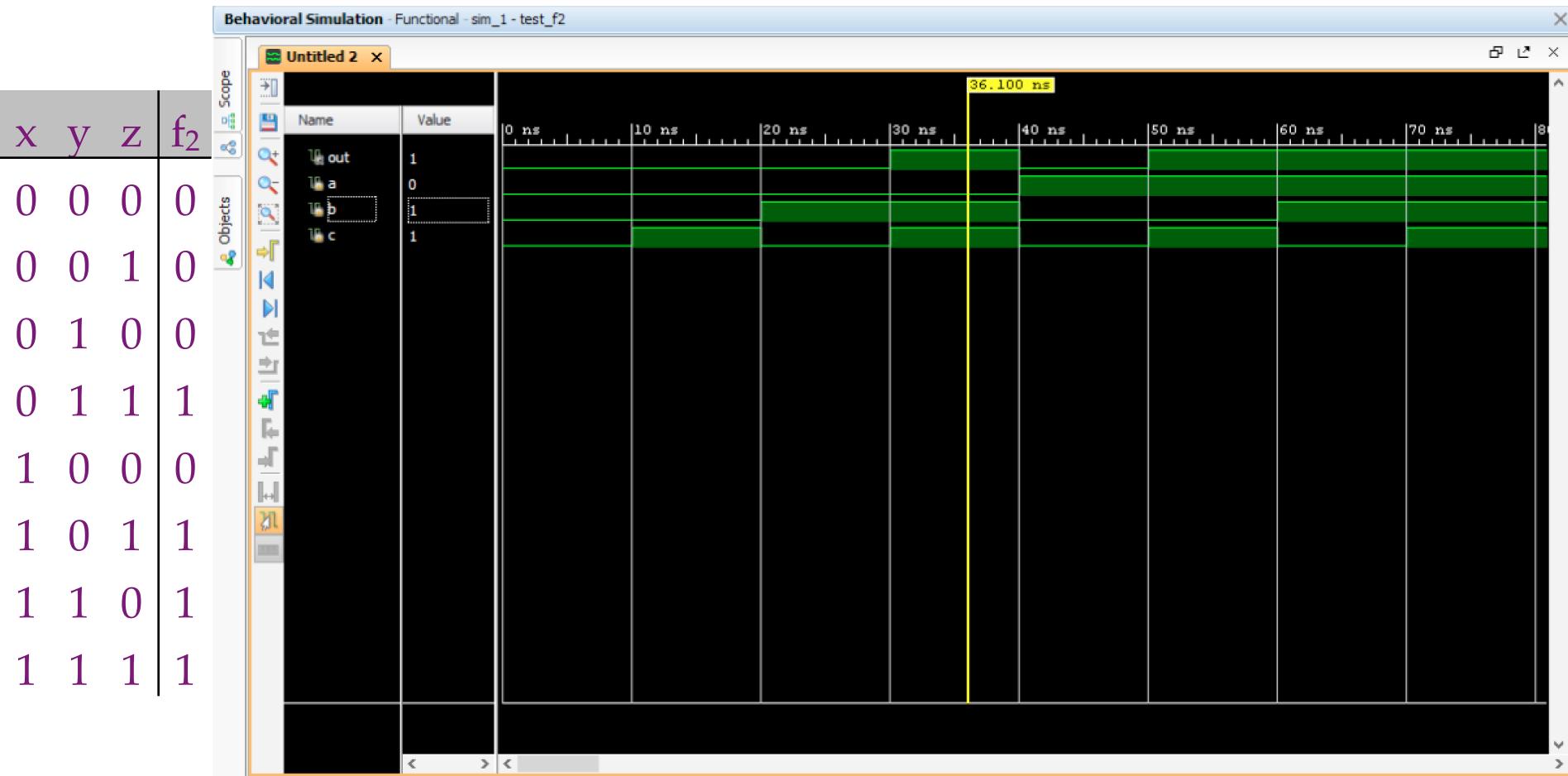
- Testbench

```
module test_f2;  
wire out;  
reg a,b,c;  
  
exp2 U0(.f2(out),.x(a),.y(b),.z(c));  
  
initial  
begin  
a=0;b=0;c=0;  
#10 a=0;b=0;c=1;  
#10 a=0;b=1;c=0;  
#10 a=0;b=1;c=1;  
#10 a=1;b=0;c=0;  
#10 a=1;b=0;c=1;  
#10 a=1;b=1;c=0;  
#10 a=1;b=1;c=1;  
end  
  
endmodule
```

x	y	z	f <sub>2</sub>
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

# Example 3 (3/3)

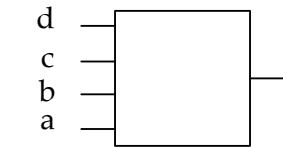
- Simulation Results



# Example 4: Prime Detector (1/2)

- Spec 1

- A circuit outputs a 1 when its four-bit input represents a prime number in binary
- $f(d,c,b,a) = 1$  if  $dcba_2$  is a prime



- Truth table 2

No	dcba	f
0	0000	0
1	0001	1
2	0010	1
3	0011	1
4	0100	0
5	0101	1
6	0110	0
7	0111	1
8	1000	0
9	1001	0
10	1010	0
11	1011	1
12	1100	0
13	1101	1
14	1110	0
15	1111	0

abbreviated truth table

No	dcba	f
1	0001	1
2	0010	1
3	0011	1
5	0101	1
7	0111	1
11	1011	1
13	1101	1
Otherwise		0

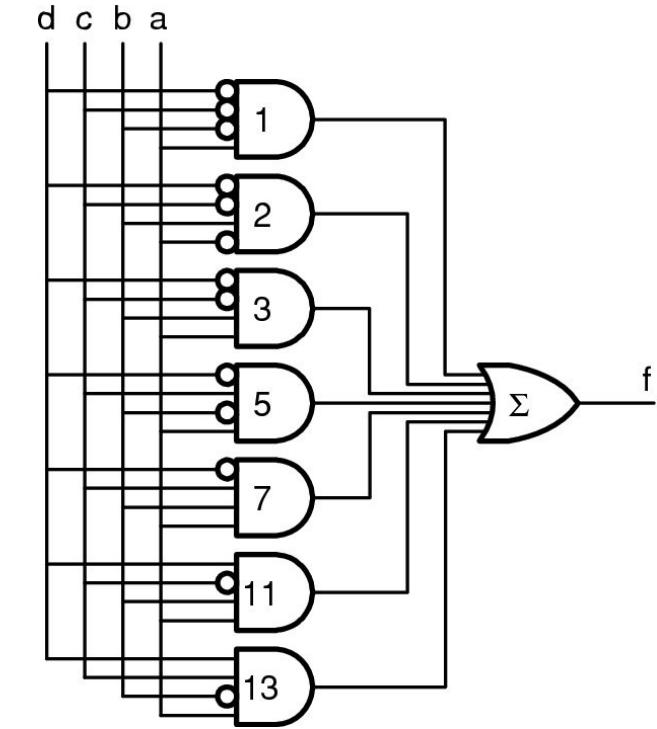
# Example 4: Prime Detector (2/2)

- Normal form 2

$$\neg f(d, c, b, a) = d'c'b'a + d'c'b'a' + d'c'ba + d'cb'a + d'cba + dc'ba + dc'b'a$$

- Sum of minterms

$$\neg f(d, c, b, a) = \sum m(1, 2, 3, 5, 7, 11, 13)$$



# Verilog Module v1 (assign)

$$f(d, c, b, a) = d'a + d'c'b + cb'a + c'ba$$

```
module prime(isprime, a, b, c, d);
output isprime;
input a,b,c,d;

assign isprime= ((~d)&~a) | ((~d)&(~c)&b) | (c&(~b)&~a) | ((~c)&b&a);

endmodule
```

# Verilog Module v2 (always block)

$$f(d, c, b, a) = d'a + d'c'b + cb'a + c'ba$$

```
module prime(isprime, a, b, c, d);
output isprime;
reg isprime;
input a,b,c,d;

always @*
isprime= ((~d)&a) | ((~d)&(~c)&b) | (c&(~b)&a) | ((~c)&b&a);

endmodule
```

# Verilog Module v3 (case)

let in[3:0]=dcba (bus)

- Use “case”

Table lookup!

```
module prime(isprime, in);
    output isprime; // true if input is prime
    input [3:0] in; // 4-bit input
    reg isprime; // for signal to be assigned in always block
```

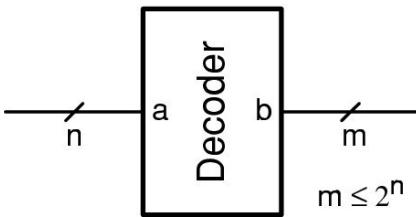
```
always @*
    case (in)
        1,2,3,5,7,11,13: isprime = 1'b1;
        default: isprime = 1'b0;
    endcase

endmodule
```

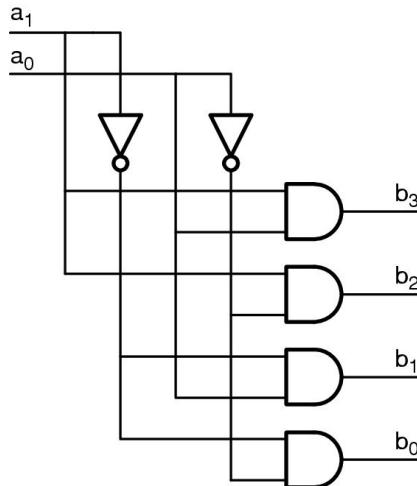
# Testbench

```
module test_prime;  
  
wire isprime;  
reg [3:0] in;  
  
// instantiate module to test  
prime U0(.isprime(isprime),.in(in));  
  
initial  
begin  
    in=0; // set to initial value  
    repeat (16) // loop 16 times  
    begin  
        #100 // delay for 100 time unit  
        $display("in = %2d isprime = %1b", in, isprime);  
        in = in +1; // increment input  
    end  
end  
endmodule
```

# A 2-to-4 Line Decoder



$a_1$	$a_0$	$b_3$	$b_2$	$b_1$	$b_0$
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0



**module** Dec24(a, b);

input [1:0] a; // binary input (2 bits wide)

output [3:0] b; // one-hot output (4 bits wide)

assign b[3]=a[1]&a[0];

assign b[2]=a[1]&(~a[0]);

assign b[1]=(~a[1])&a[0];

assign b[0]=(~a[1])&(~a[0]);

**endmodule**

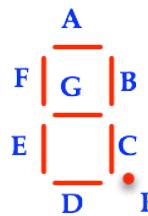
$$b_3 = a_1 a_0$$

$$b_2 = a_1 a'_0$$

$$b_1 = a'_1 a_0$$

$$b_0 = a'_1 a'_0$$

# A BCD to Seven-Segment Display Decoder



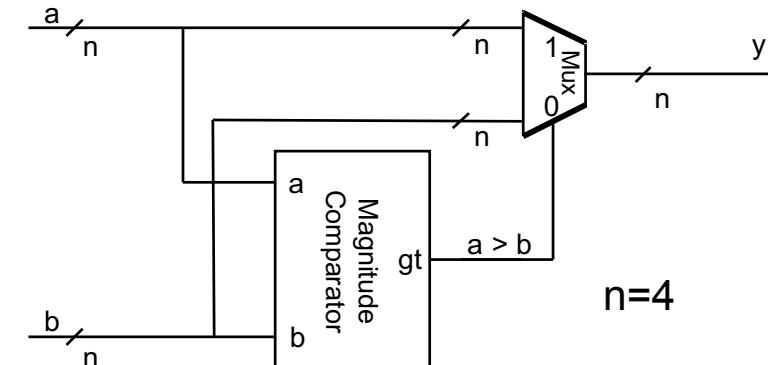
```
// define segment codes
`define SS_0 7'b1111110
`define SS_1 7'b0110000
`define SS_2 7'b1101101
`define SS_3 7'b1111001
`define SS_4 7'b0110011
`define SS_5 7'b1011011
`define SS_6 7'b1011111
`define SS_7 7'b1110000
`define SS_8 7'b1111111
`define SS_9 7'b1111011
```

```
module sseg(segs, bin);
output [6:0] segs;
input [3:0] bin;
reg [6:0] segs;

always @*
  case (bin)
    4'd0: segs = `SS_0;
    4'd1: segs = `SS_1;
    4'd2: segs = `SS_2;
    4'd3: segs = `SS_3;
    4'd4: segs = `SS_4;
    4'd5: segs = `SS_5;
    4'd6: segs = `SS_6;
    4'd7: segs = `SS_7;
    4'd8: segs = `SS_8;
    4'd9: segs = `SS_9;
    default: segs = 7'b0000000;
  endcase
endmodule
```

# Maximum Unit

$$y = \max\{a, b\}$$



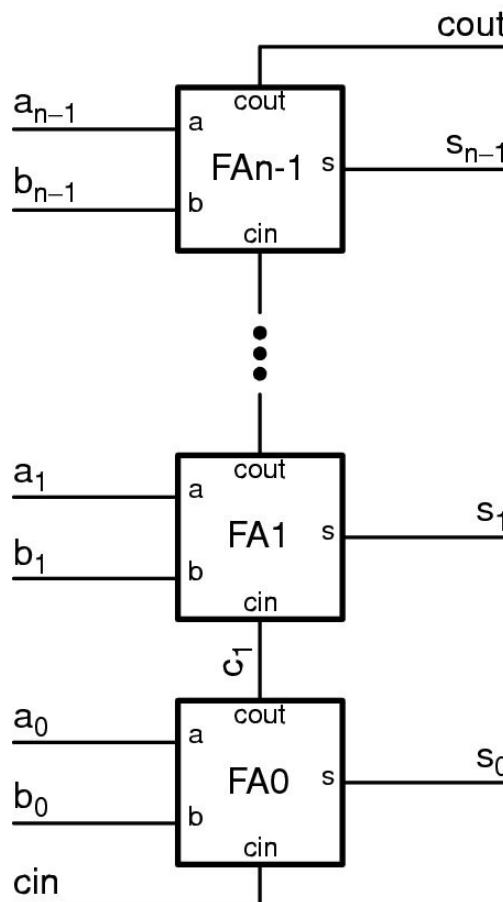
```
module Max(a, b, y);
input [3:0] a, b; // two input variables (4 bits wide)
output reg [3:0] y; // maximum output (4 bits wide)
```

```
assign gt=(a>b) ? 1 : 0;
always @*
if (gt)
    y = a;
else
    y = b;
```

```
endmodule
```

```
always @*
if (a > b)
    y = a;
else
    y = b;
```

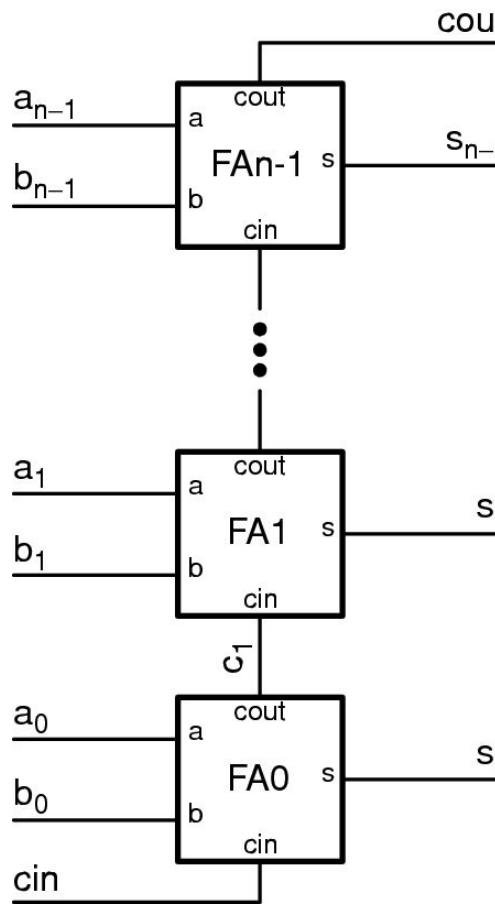
# Ripple-Carry Adder (1/2)



```
module adder(a, b, cin, cout, s);  
parameter n=4;  
input [n-1:0] a, b;  
input cin;  
output reg [n-1:0] s;  
output cout;  
reg [n:0] c;  
integer i;  
  
assign {c[1],s[0]} = a[0] + b[0] + c[0];  
assign {c[2],s[1]} = a[1] + b[1] + c[1];  
assign {c[3],s[2]} = a[2] + b[2] + c[2];  
assign {c[4],s[3]} = a[3] + b[3] + c[3];  
  
assign cout = c[n];  
always @*  
  c[0] = cin;  
always @*  
  for (i=0;i<n;i=i+1)  
    {c[i+1],s[i]} = a[i] + b[i] + c[i];  
  
endmodule
```



# Multi-bit Binary Adder (2/2)



```
module Adder1(a, b, cin, cout, s);  
parameter n=8;  
input [n-1:0] a, b;  
input cin;  
output [n-1:0] s;  
output cout;  
  
assign {cout, s} = a + b + cin;  
  
endmodule
```