Digital System Laboratory

Verilog HDL

T. Hui

Person In Charge

T. Hui	Maoyang	
2.515	2.515	
tthui@sutd.edu.sg	maoyang_xiang@sutd.edu.sg	

Grading Policy

Task		Percentage
Assignment	Online Homework	10%
Quiz	Online MCQ Online Lab-Test	20%
Handson	In class exercise	30%
2D	Demo Q&A Report (Github)	40%
Classroom Participation		+2%

AI?

Any kind of ***Local Installed AI*** tools are allowed in this course by proving the installation information, prompt, and other innovation.

Use of the Software

- Use only legal approved software, freeware, such as
  - Vivado, free version.
  - Matlab, school version.
  - Modelsim, free version.
- Any other software (local, online) that is used, students must include the links and declare in the report.

Submission Tips

- Github should consist of all the assignments' materials (homework, tasks, 2D etc.) that are organized properly, e.g.
  - Separate folders: Homework, Task, 2D, etc.
  - Separate subfolders: source (Verilog files), video, report
- Generally, each assignment should have:
  1. verilog files (module, testbench)
  2. video
  3. report in either word, jupyter notebook, pdf etc. that involves design document (e.g. block diagram, functionality, algorithm, references), that highlight the innovativeness, education, and other values

FPGA Software Setup - Vivado

- Register an account with Xilinx
  - You need this registration for download, and installation
- Download Vivado-design-tools
  - 2023.1
- Please refer to the below Guide for detail setup

https://pe8sutd.larksuite.com/docx/NfZid9uy5oiYvbxoEWMui94HsfY?from=from_copylink

You need to start the installation now, as it may take a long time to install.

20 minutes to kick start the simulation, and let it runs.

We will be using Vivado for simulation.

Verilog HDL

Week01 – Lesson 1 Plan

- Introduction
- Typical Design Flow
- Verilog Conventions
- A Structural Description
- Module Hierarchy
- Behavioral Modelling
- Synthesis
- Benefits of HDL Synthesis in Top-Down Design
- Implementation : FPGA

Introduction

- In the digital design field, designers felt the need for a standard language to describe digital circuits.
- Thus, Hardware Description Languages (HDLs) came into existence.
  - Verilog HDL
  - VHDL
- High Level Synthesis (HLS) is also getting popular.

a hardware description language is a specialized computer language used to describe the structure and behavior of electronic circuits

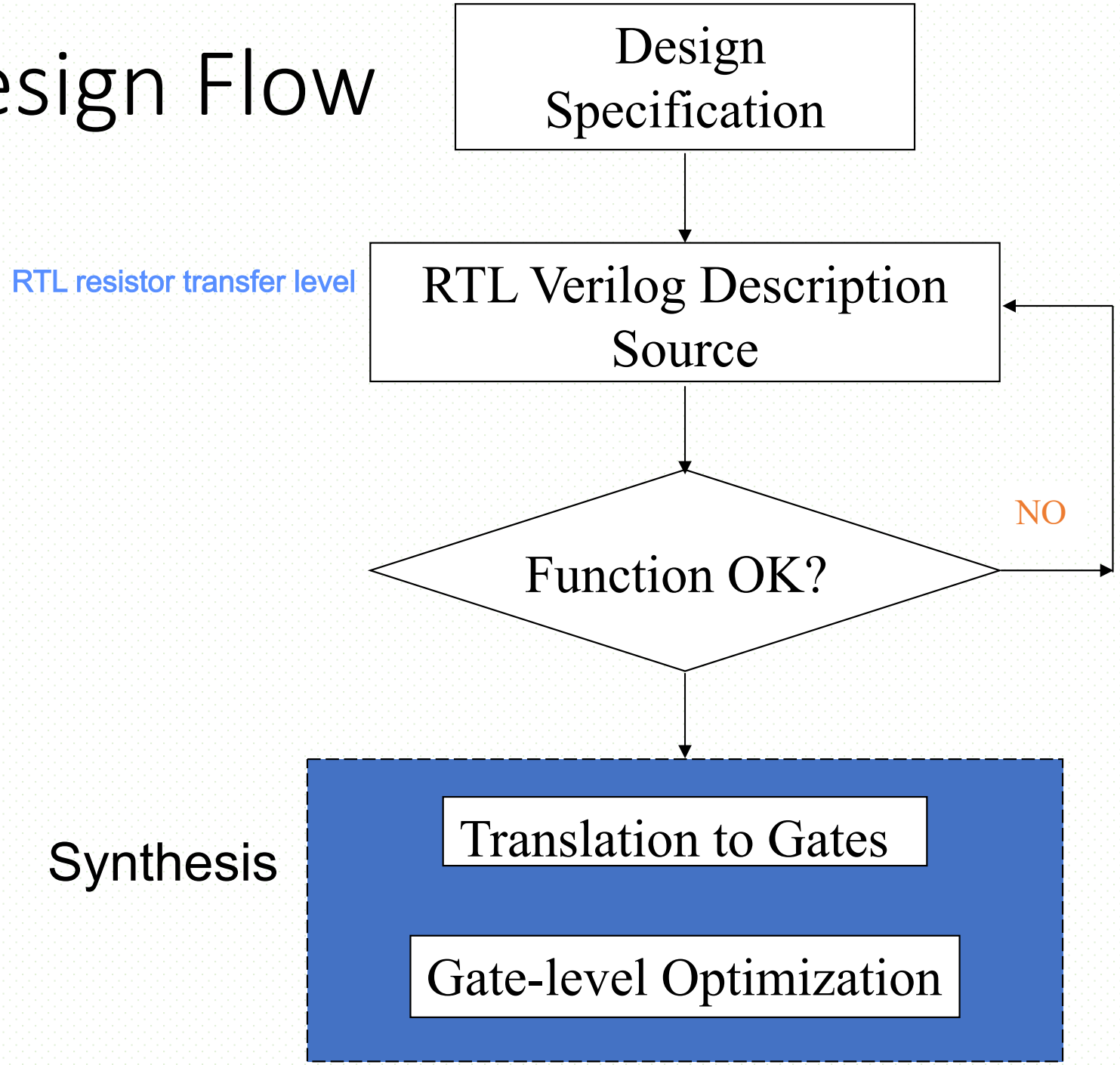
Why use HDLs?

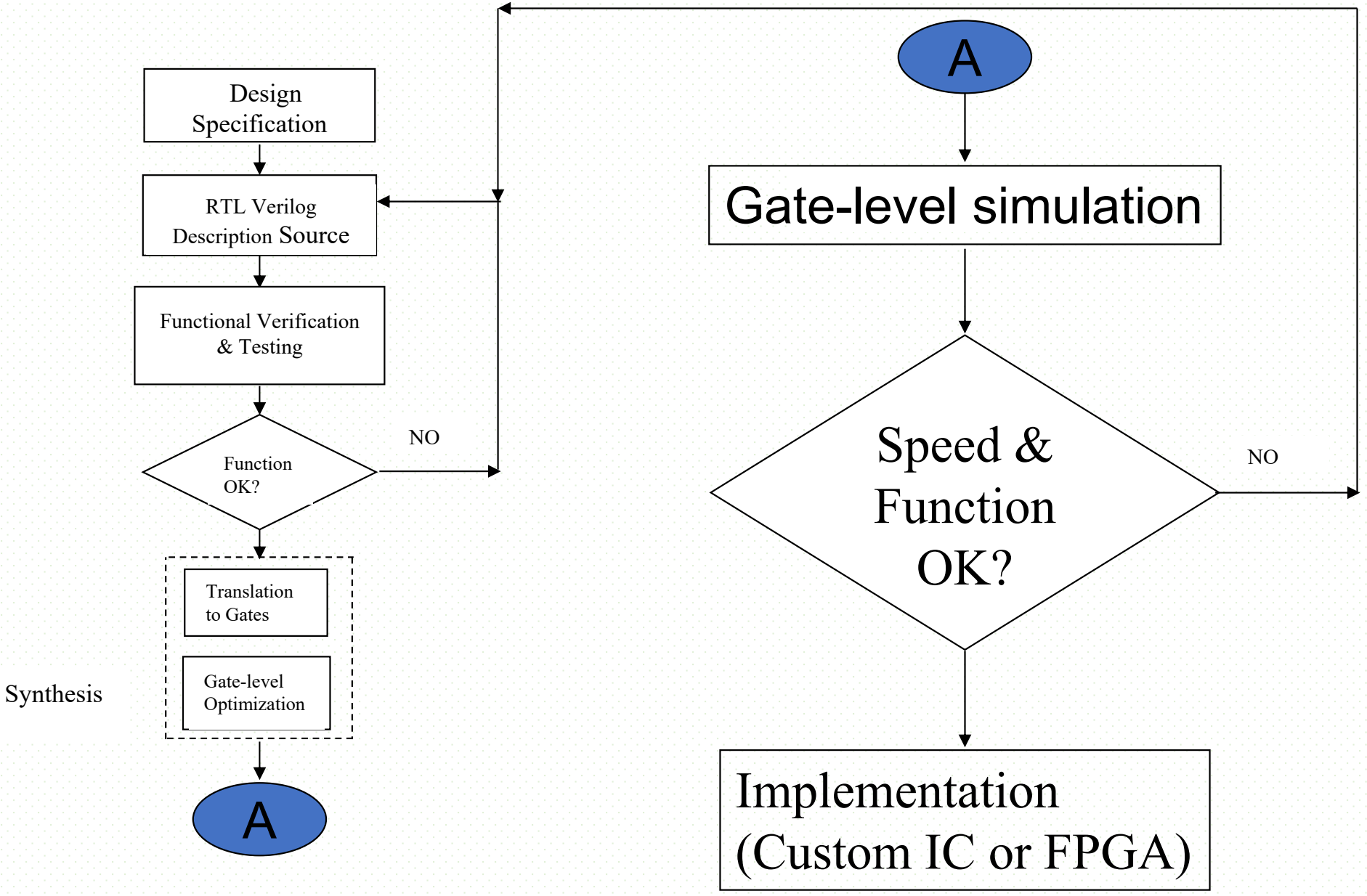
- It is becoming very difficult to design directly on hardware.
- Exploring different design options is easier and cheaper.
- Reduce time and cost versus prototyping.

Key Features of HDLs

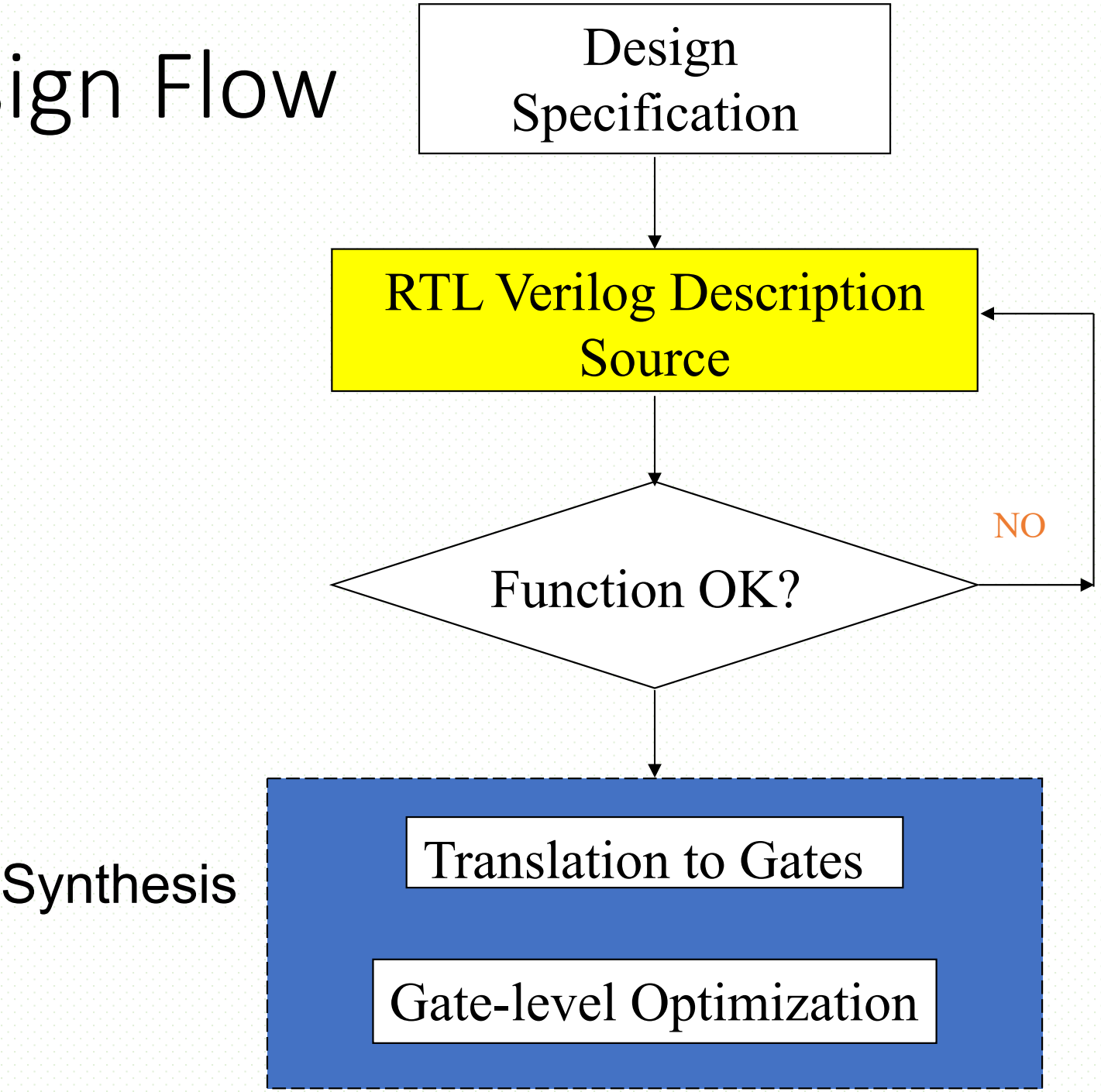
- HDLs have high-level programming constructs to describe the connectivity of the circuit.
- HDLs allow one to describe the design at various levels of abstraction.
- HDLs allow one to describe the functionality as well as the timing.
- Concurrency.
- Time.

HDL Design Flow




Design Specification
RTL Verilog Description Source
Functional Verification & Testing
Function OK?
NO
Translation to Gates
Gate-level Optimization
Synthesis
A
A
Gate-level simulation
Speed & Function OK?
NO
Implementation (Custom IC or FPGA)

HDL Design Flow

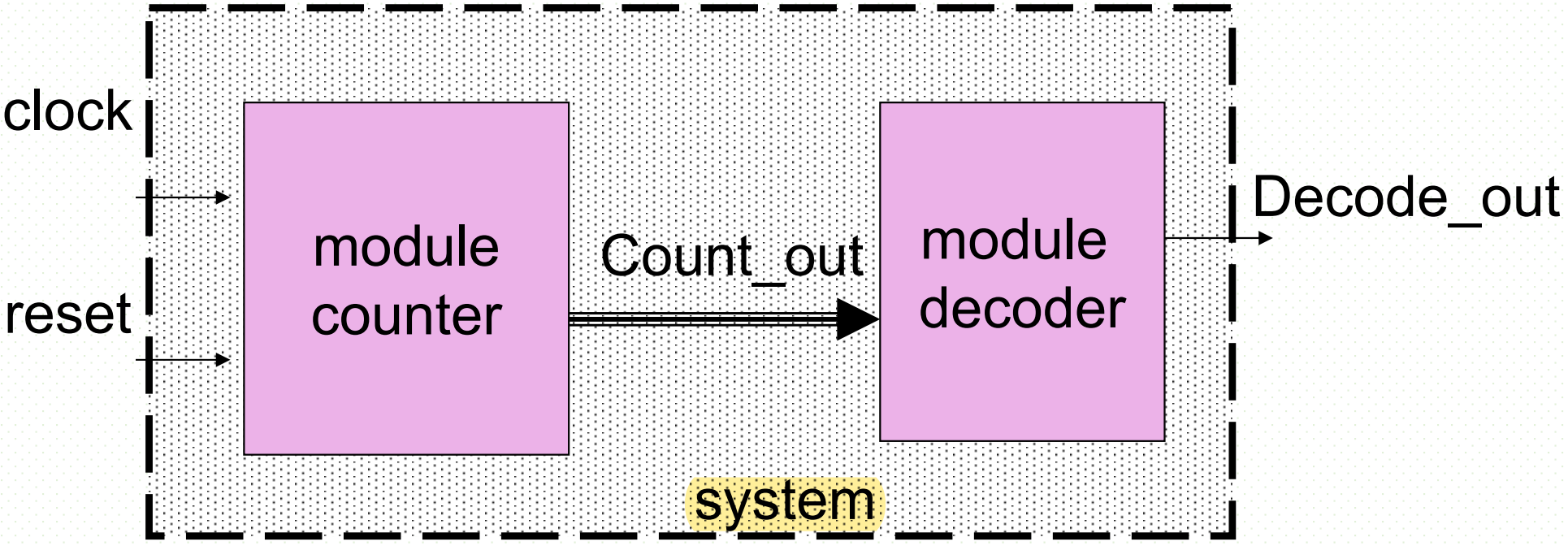


HDL History

- 1970s:
  - First HDLs
- Late 1970s: VHDL
  - VHDL = VHSIC HDL = Very High Speed Integrated Circuit HDL
  - VHDL inspired by programming languages of the day (Ada)
- 1980s:
  - Verilog first introduced
  - Verilog inspired by the C programming language
  - VHDL standardized
- 1990s:
  - Verilog standardized (Verilog-1995 standard)
- 2000s:
  - Continued evolution (Verilog-2001 standard)

Verilog HDL

- The Verilog HDL describes a digital system as a set of modules
- Each of these modules has an interface to other modules as well as a description of its contents through its port interface (inputs and outputs).



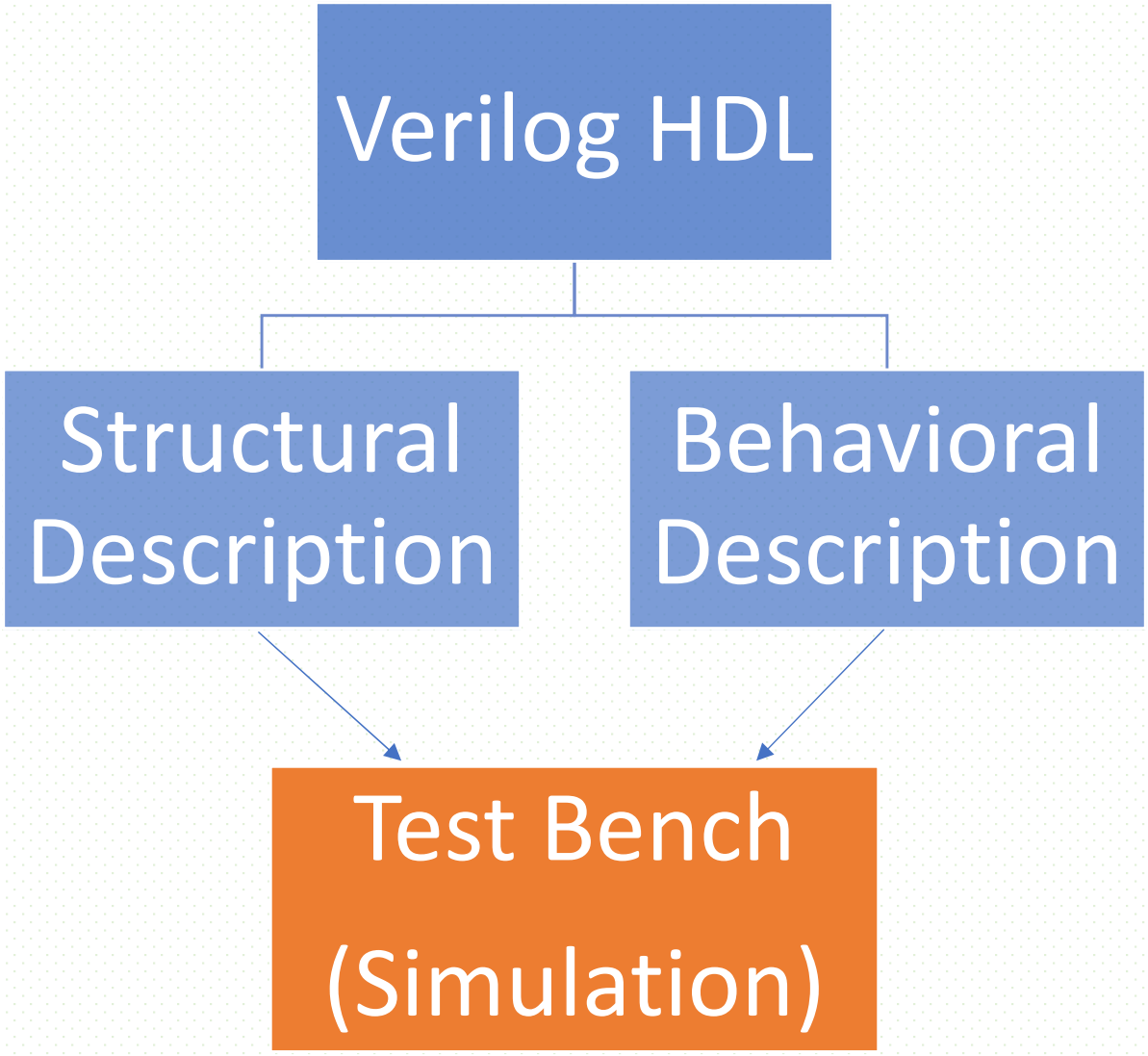
Verilog HDL

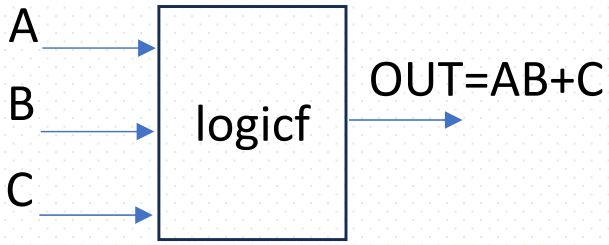
A module represents a logical unit that can be described either by :

- Structural description
  - specifying its internal logical structure
- Behavioral description
  - describing its behavior in a program-like manner

(examples as below)

Verilog HDL


Verilog HDL
Structural Description
Behavioral Description
Test Bench (Simulation)



Structural Description

```
module logicf(OUT,A,B,C);
 input A,B,C;
 output OUT;

and g1 (outand,A,B);
or g2 (OUT,outand,C);

endmodule
```

boolean algebra

- AND → multiply
- OR → add.

Behavioural Description

```
module logicf(OUT,A,B,C);
 input A,B,C;
 output OUT;
 reg OUT;

always @(A or B or C)
 OUT = A&B |C;

endmodule
```

Verilog Conventions: Comments

Comments can be inserted in the code for readability and documentation. There are two ways to write comments.

- A one-line comment, it starts with "//", e.g.:

```
input  a,b;  // This is a one-line comment.
```

- Multiple-line comment starts with "/*" and ends with "*/", e.g.:

```
/*  This is a multiple line
comment */
```

Numbers and Bases REFRESHER

A number base is the number of digits or combination of digits that a system of counting uses to represent numbers.
A base can be any whole number greater than 0.
The most commonly used number system is the decimal system, commonly known as base 10. Its popularity as a system of counting is most likely due to the fact that we have 10 fingers.

Binary is the most commonly used non-base 10 system. It is used for coding in computers. Binary is also known as Base 2/bit (binary digit). This means it is composed of only 0's and 1's. For example 9 in binary/base 2 is 1001. Let's see how this works.

Base 10	Base 2	2^4	2^3	2^2	2^1	2^0
1	**1**	0	0	0	0	1
9	**1001**	0	1	0	0	1
16	**10000**	1	0	0	0	0

eg.
2 | 9
2 | 4 1
2 | 2 0
1 0

$9_{10} = 1001_2$ ✓

Let's look at Base 16, also known as the hexadecimal system, another common base when coding and using computer systems. In this case, we use the digits 0–9 and the letters representing two digits A(10),B(11),C(12),D(13),E(14),F(15).

Base 10	Base 16	16^2	16^1	16^0
100	**064**	0	6	4
42	**02A**	0	2	A (10)
124	**07C**	0	7	C (12)
269	**10D**	1	0	D (13)

eg.
16 | 42 10
2

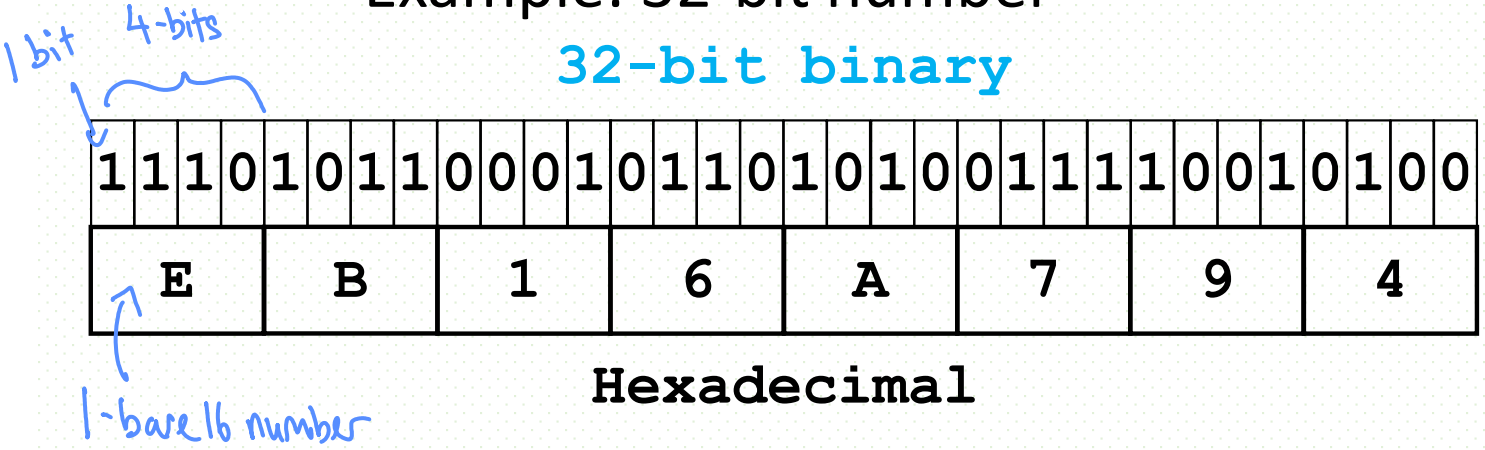
$42_{10} = 2A_{16}$ ✓

16 | 100 4
6

$100_{10} = 64_{16}$ ✓

Hexadecimal Numbers

- Decimal aka. Radix 10
- Binary aka. Radix 2
- Hexadecimal = Radix 16 aka.
  - Radix 16 = 2^4 1 base 16 number is represented by 4 bits (binary digits)
  - 16 digits: 0 to 9, A to F
  - A=10, B=11, …, F=15
  - Example: 32-bit number



Decimal Radix 10	Binary Radix 2	Hex Radix 16
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Converting Decimal to Hexadecimal

- Example: convert 422 to hexadecimal

422/16=26, /: divide 422%16=6, %: modulus

Division	Quotient	Remainder
422 / 16	26	6
26 / 16	1	A
1 / 16	0	1

least significant digit

most significant digit

stop when quotient is zero

$422_{10} = (1A6)_{16}$

Regarding memory allocation. C code allows optimization of memory, can assign # of bits to store certain data. Python allocates 32-bits for all data

Number Specification

There are two types of number specification, ie: sized and unsized.

- Sized numbers, e.g.:

4’**b**1110 // This is a 4-bit **binary** number of ‘1110’

12’**h**abc // This is a 12-bit **hexadecimal** number of ‘abc’

16’d123 // This is a 16-bit decimal number of ‘123’

- Unsized numbers, e.g.:

23456 // This is a 32-bit decimal number (by default) of ‘23456’

‘he6 // This is a 32-bit hexadecimal number of ‘e6’

Number Specification

- x or z values represent unknown and high impedance values respectively, e.g.:

```
6'hx    // This is a 6-bit hex unknown number
32'bz // This is a 32-bit high impedance number
```

- Question marks ? is an alternative for z in the context of numbers, e.g:

```
4'b10??   // This is equivalent of a 4'b10zz
```

Module

- Header

Keyword: module

enmodule

- Port declarations eg. Input output pins of IC are ports

Keyword: input

output

inout

wire

reg

- Logic and functional description

Different Way of Port Declaration

Separate port declaration

header

port declaration

commenting

footer

```
module top(y,a,b);
 output y;
 input a,b;

//logic

endmodule
```

Integrate port declaration

```
module top(
 output y,
 input a,b
 );

//logic

endmodule
```

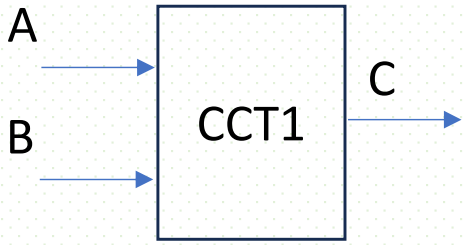
Port Connectivity

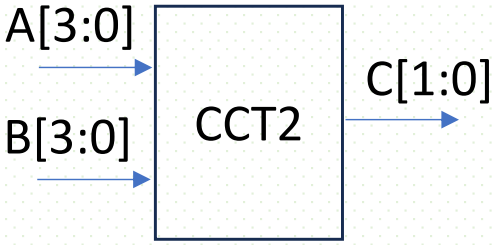
Port connection by location. Location of the ports in the same order.

```
top dut1(ylocal, alocal, blocal);
```

Port connection by name. Location of the ports can be in different order.

```
top dut1(.alocal(y), .blocal(a),
.ylocal(b));
```





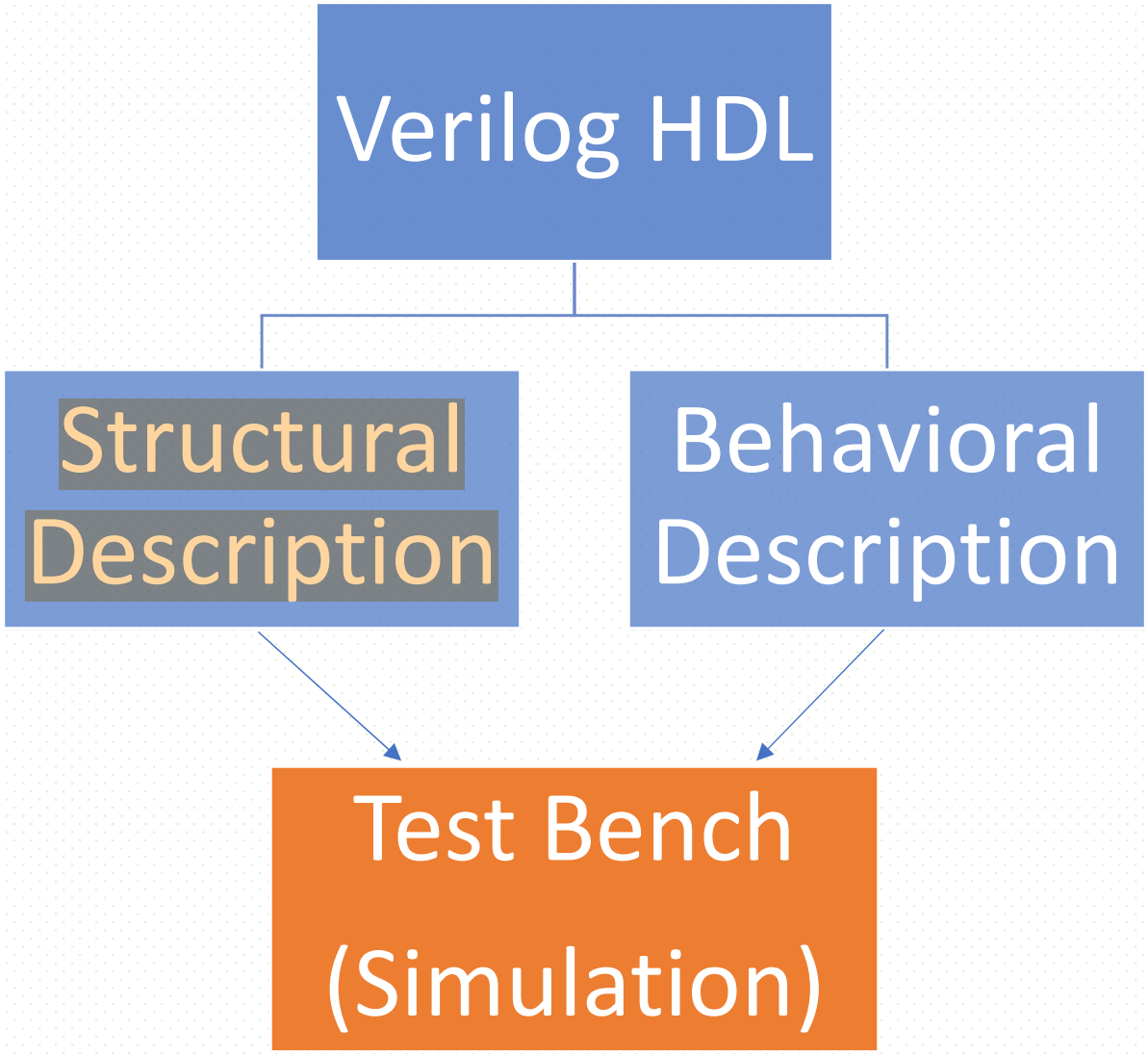
Single Bit

```
module CCT1( C, A, B );
        output C;
        input A, B ;

// Logic/functional Description
        ……..
        ……..
endmodule
```

Multiple Bit

```
module CCT2(C, A, B);
        output[1:0] C;
        input[3:0] A, B ;

// Logic/functional Description
        ……..
        ……..
endmodule
```

Verilog HDL


Verilog HDL
Structural Description
Behavioral Description
Test Bench (Simulation)

Structural Description - Examples

- NAND Latch
- NOR Latch

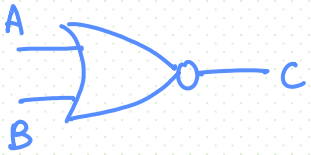
*Latches are digital circuits that store a single bit of information and hold its value until it is updated by new input signals.

NAND

truth table

A	B	C
0	0	1
0	1	1
1	0	1
1	1	0

NOR



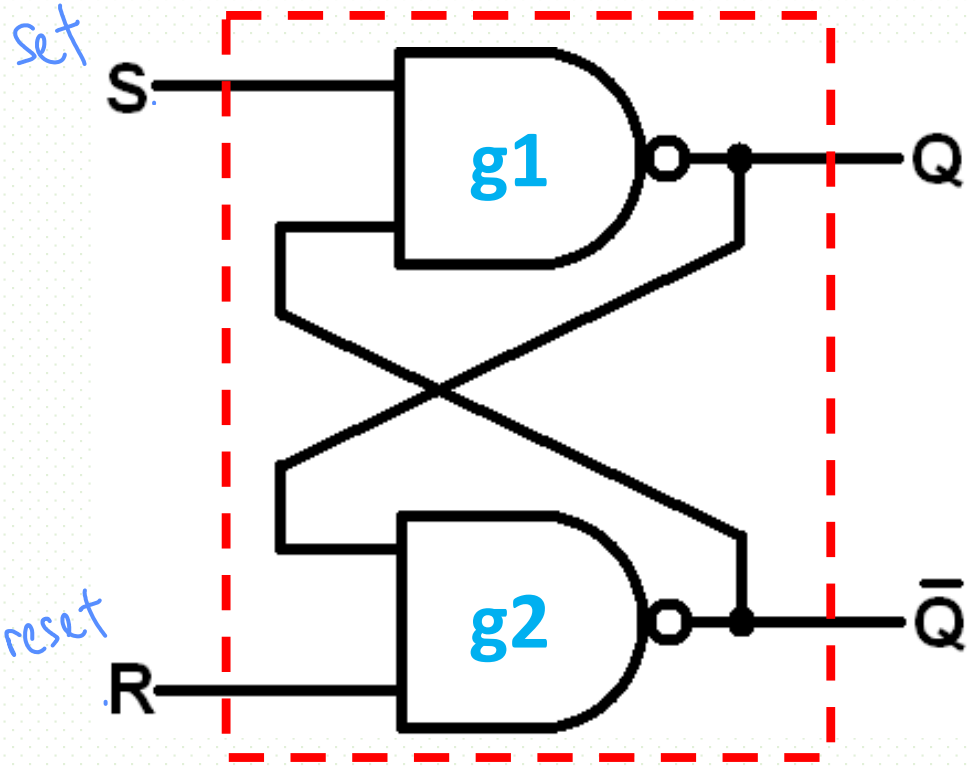
truth table

A	B	C
0	0	1
0	1	0
1	0	0
1	1	0

NAND Latch

store/memory

NAND Latch



Truth Table

t		$t+1$		
S	R	Q	$\bar{Q}$	
0	0	?	?	race
0	1	1	0	set
1	0	0	1	reset
1	1	Q	$\bar{Q}$	hold

$S = 0, Q = 1$, Active Low

set- inputs that lead to output, Q of 1

reset- inputs that lead to output, Q of 0

hold- set of inputs that do not change the output status quo

race- depending on which input reaches first

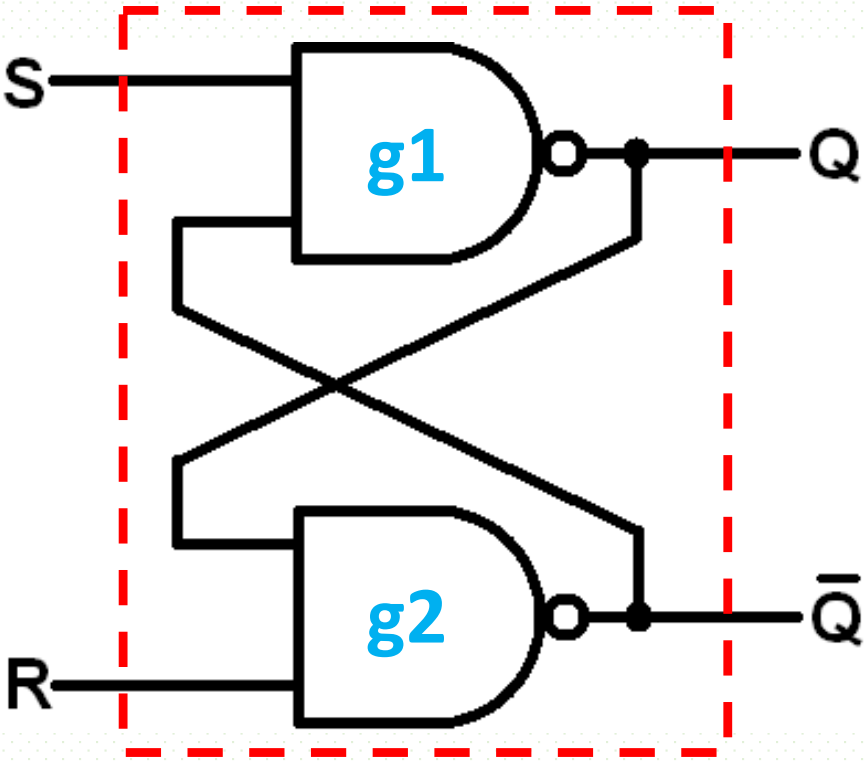
JIT

NAND Latch

For NAND, if one of the input is “0”, output must be “1”;
Thus, cannot provide complimentary output for $Q, \bar{Q}$ in NAND latch.
Causing “race” when $S = 0, R = 0$.

NAND	i1	i0
1	0	0
1	0	1
1	1	0
0	1	1

NAND Latch



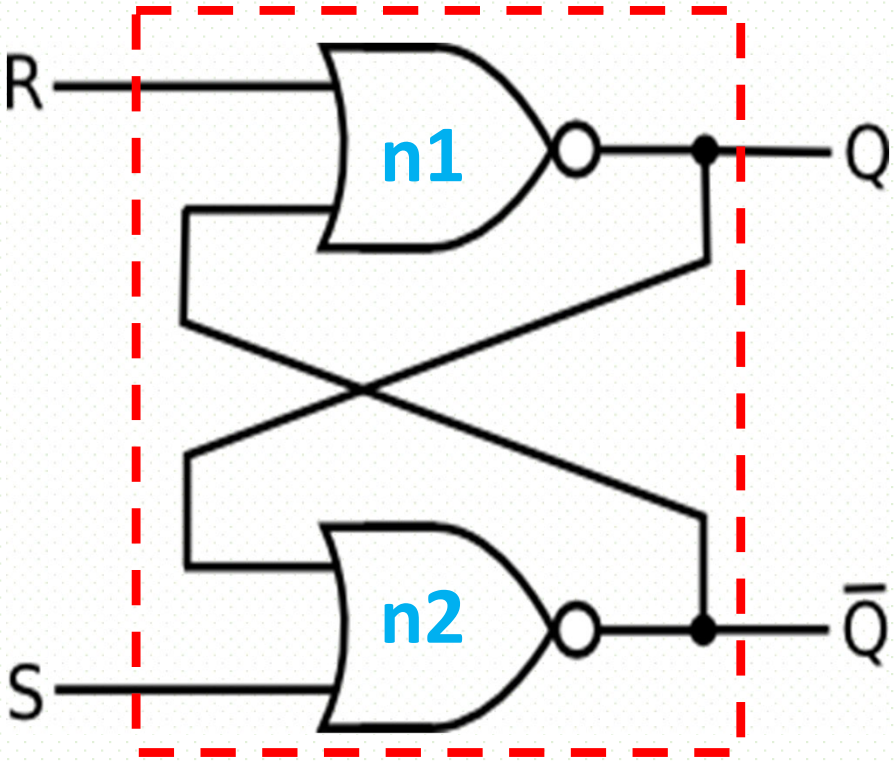
Truth Table

t		$t+1$		
S	R	Q	$\bar{Q}$	
0	0	?	?	race
0	1	1	0	set
1	0	0	1	reset
1	1	Q	$\bar{Q}$	hold

$S = 0, Q = 1$, Active Low

NOR Latch

NOR Latch



Truth Table

t		$t+1$		
S	R	Q	$\bar{Q}$	
0	0	Q	$\bar{Q}$	hold
0	1	0	1	reset
1	0	1	0	Set
1	1	?	?	race

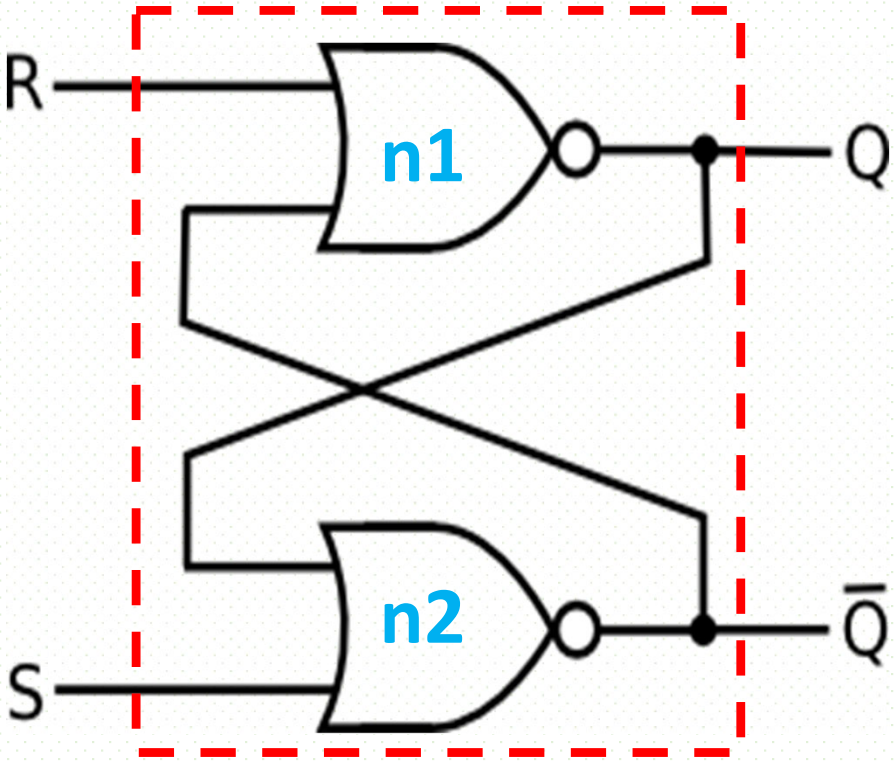
$S = 1, Q = 1$, Active High

NOR Latch

For NOR, if one of the input is “1”, output must be “0”;
Thus, cannot provide complimentary output for $Q, \bar{Q}$ in NOR latch.
Causing “race” when $S = 1, R = 1$.

NOR	i1	i0
1	0	0
0	0	1
0	1	0
0	1	1

NOR Latch



Truth Table

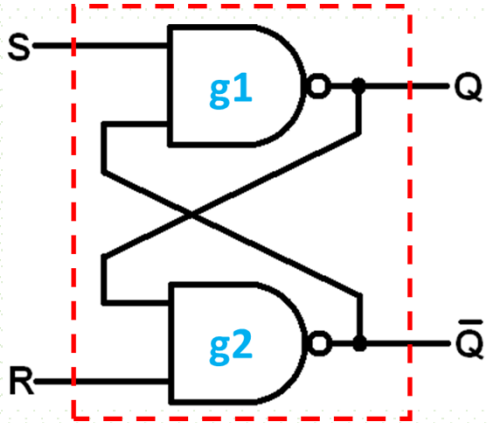
t		$t+1$		
S	R	Q	$\bar{Q}$	
0	0	Q	$\bar{Q}$	hold
0	1	0	1	reset
1	0	1	0	Set
1	1	?	?	race

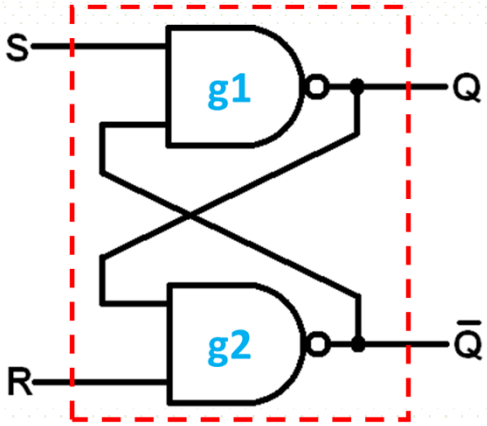
$S = 1, Q = 1$, Active High

Example

Write a Verilog HDL of a simple NAND Latch using structural description:

- has two outputs which, when legal inputs are provided, are complementary
- form using 2 primitive NAND gates (having a delay of 1)



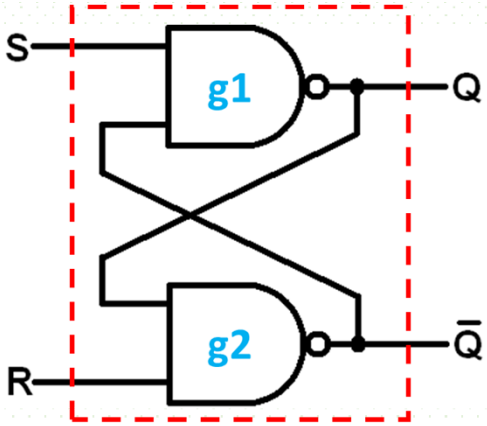


NAND Latch

```
module nand_rs(q,qb,s,r);
//port declaration
 output q,qb;
 input s,r;

//logic
nand #1 g1(q,s,qb);
nand #1 g2(qb,q,r);

endmodule
```

How to verify the functionality of the module nand_rs?

NAND Latch Testbench?



NAND Latch

```
module nand_rs(q,qb,s,r);


endmodule
```

open program

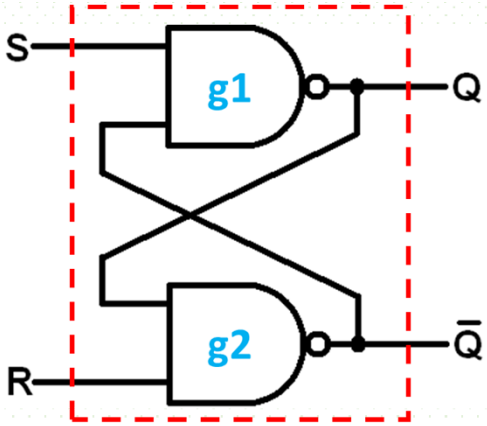
name

ports

close program

How to verify the functionality of the module nand_rs?

NAND Latch Testbench?

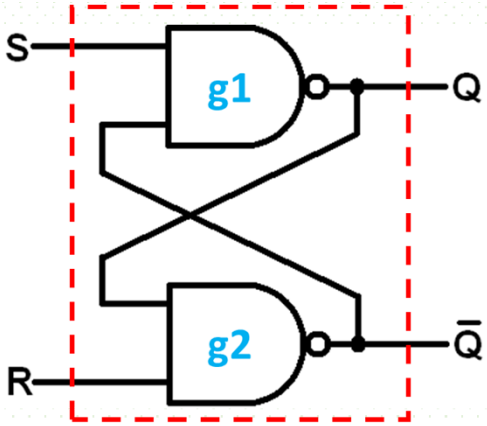


NAND Latch

```
module nand_rs(q,qb,s,r);
//port declaration
 output q,qb;




endmodule
```

How to verify the functionality of the module nand_rs?

NAND Latch Testbench?

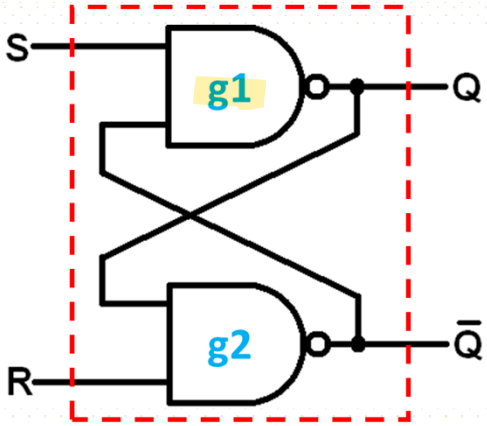


NAND Latch

```
module nand_rs(q,qb,s,r);
//port declaration
 output q,qb;
 input s,r;




endmodule
```

How to verify the functionality of the module nand_rs?

NAND Latch Testbench?



NAND Latch

```
module nand_rs(q,qb,s,r);
//port declaration
 output q,qb;
 input s,r;

//logic
nand #1 g1(q,s,qb);

endmodule
```

library

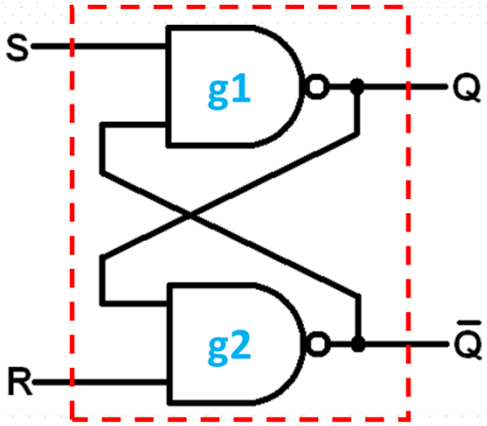
1 unit delay

output of g1

input of g1

How to verify the functionality of the module nand_rs?

NAND Latch Testbench?

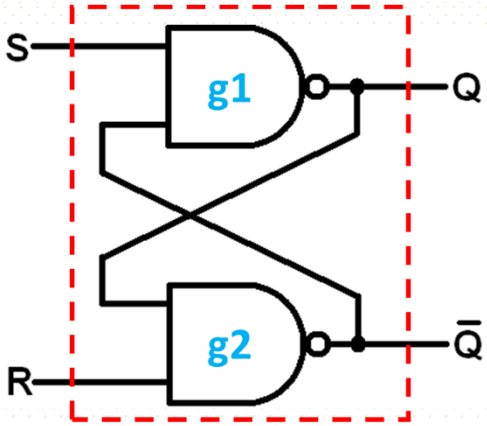


NAND Latch

```
module nand_rs(q,qb,s,r);
//port declaration
 output q,qb;
 input s,r;

//logic
nand #1 g1(q,s,qb);
nand #1 g2(qb,q,r);

endmodule
```

How to verify the functionality of the module nand_rs?

NAND Latch Testbench?



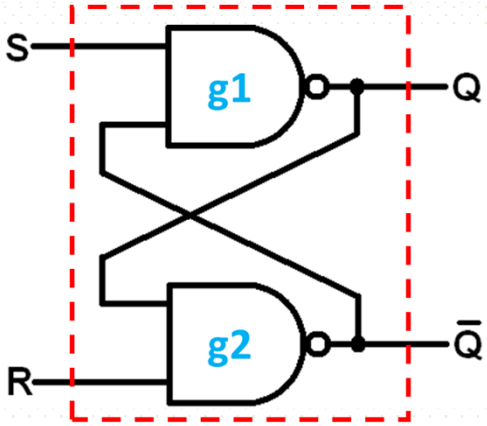
How to verify the functionality of the module nand_rs? Truth table

NAND Latch

```
module nand_rs(q,qb,s,r);
//port declaration
 output q,qb;
 input s,r;

//logic
nand #1 g1(q,s,qb);
nand #1 g2(qb,q,r);

endmodule
```

NAND Latch Testbench?

t		$t+1$		
S	R	Q	$\bar{Q}$	
0	0	?	?	race
0	1	1	0	set
1	0	0	1	reset
1	1	Q	$\bar{Q}$	hold

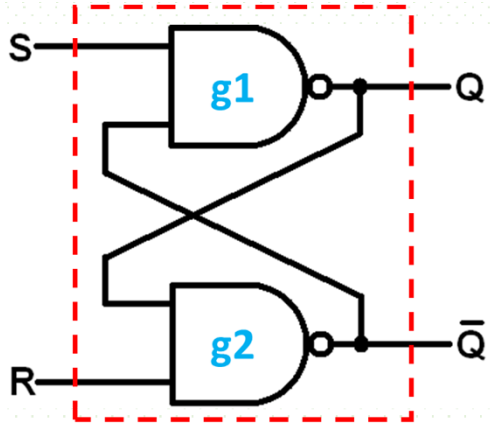


NAND Latch

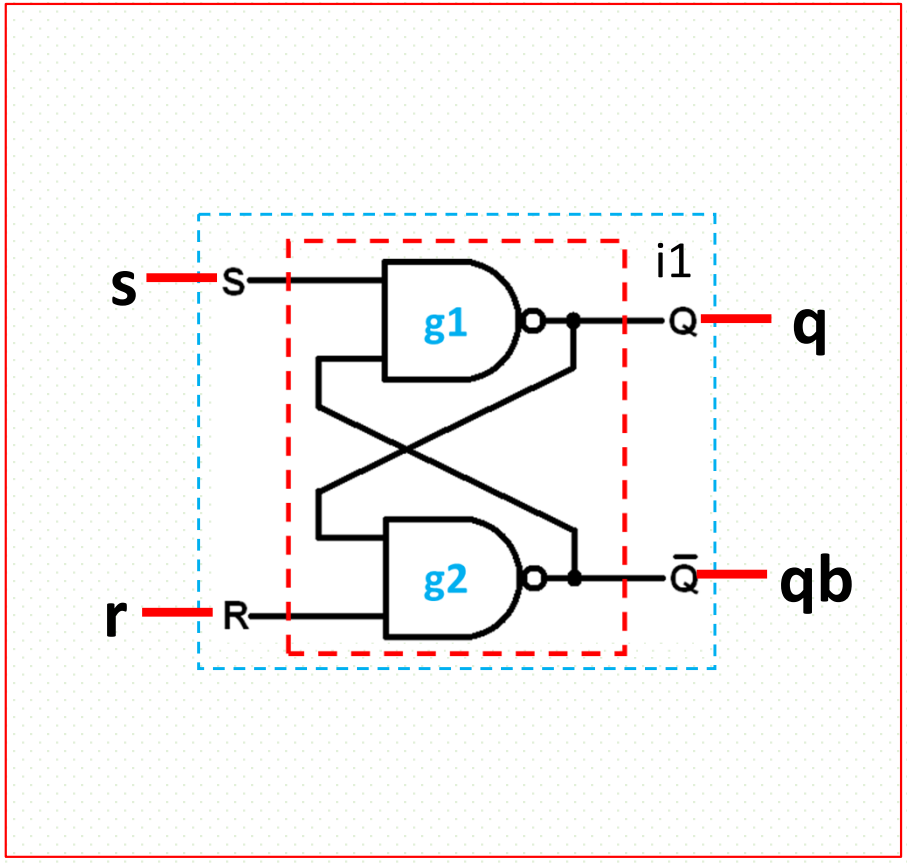
```
module nand_rs(q,qb,s,r);
//port declaration
 output q,qb;
 input s,r;

//logic
nand #1 g1(q,s,qb);
nand #1 g2(qb,q,r);

endmodule
```

How to verify the functionality of the module nand_rs?

NAND Latch Testbench?



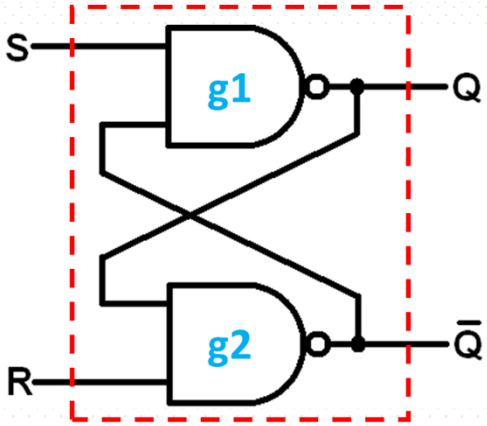
NAND Latch Testbench



NAND Latch Testbench?

```
/* module name
and NO port list */
module nand_rs_tb;
 wire q,qb;
 reg s,r;

/*instantiate the design
module */
nand_rs i1(q,qb,s,r);

initial
 begin
   s=0; r=1;
   #10 s=1; r=1;
   #10 s=1; r=0;
   #10 s=1; r=1;
 end

endmodule
```

t		$t+1$		
S	R	Q	$\bar{Q}$	
0	0	?	?	race
0	1	1	0	set
1	0	0	1	reset
1	1	Q	$\bar{Q}$	hold



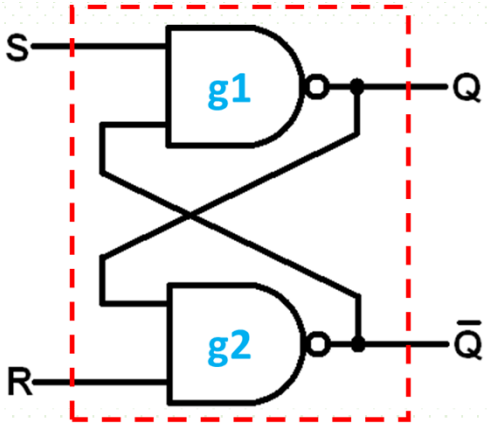
NAND Latch

```
module nand_rs(q,qb,s,r);
//port declaration
 output q,qb;
 input s,r;

//logic
nand #1 g1(q,s,qb);
nand #1 g2(qb,q,r);

endmodule
```

NAND Latch Testbench?

```
/* module name
and NO port list */
module nand_rs_tb;
 wire q,qb;
 reg s,r;

//instantiate the design
module
nand_rs i1(q,qb,s,r);

initial
 begin
   s=0; r=1;
   #10 s=1; r=1;
   #10 s=1; r=0;
   #10 s=1; r=1;
 end

endmodule
```

t		$t+1$		
S	R	Q	$\bar{Q}$	
0	0	?	?	race
0	1	1	0	set
1	0	0	1	reset
1	1	Q	$\bar{Q}$	hold

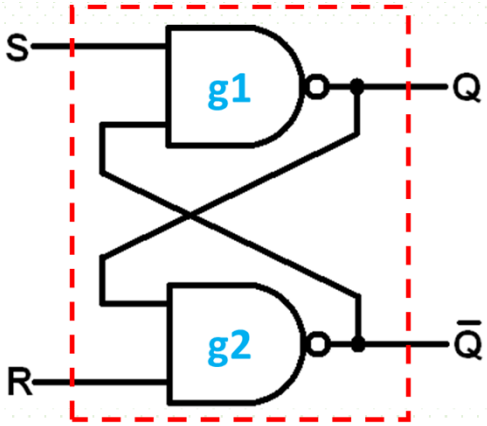


NAND Latch

```
module nand_rs(q,qb,s,r);
//port declaration
 output q,qb;
 input s,r;

//logic
nand #1 g1(q,s,qb);
nand #1 g2(qb,q,r);

endmodule
```

NAND Latch Testbench?

```
/* module name
and NO port list */
module nand_rs_tb;




endmodule
```

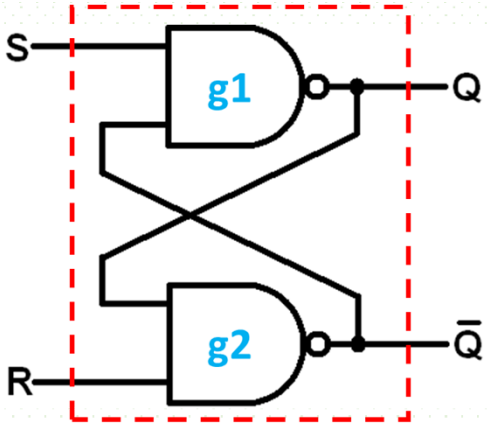


NAND Latch

```
module nand_rs(q,qb,s,r);
//port declaration
 output q,qb;
 input s,r;

//logic
nand #1 g1(q,s,qb);
nand #1 g2(qb,q,r);

endmodule
```

NAND Latch Testbench?

```
/* module name
and NO port list */
module nand_rs_tb;
 wire q,qb;

endmodule
```

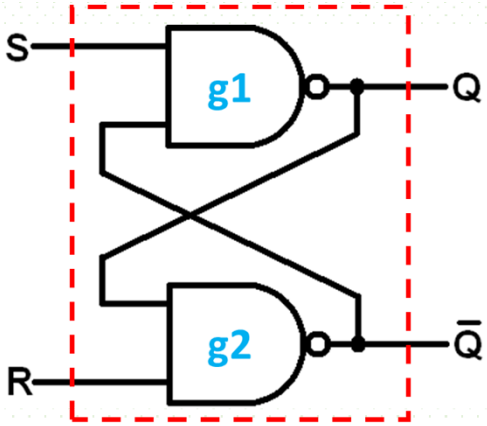


NAND Latch

```
module nand_rs(q,qb,s,r);
//port declaration
 output q,qb;
 input s,r;

//logic
nand #1 g1(q,s,qb);
nand #1 g2(qb,q,r);

endmodule
```

NAND Latch Testbench?

```
/* module name
and NO port list */
module nand_rs_tb;
 wire q,qb;
 reg s,r;




endmodule
```



NAND Latch

```
module nand_rs(q,qb,s,r);
//port declaration
 output q,qb;
 input s,r;

//logic
nand #1 g1(q,s,qb);
nand #1 g2(qb,q,r);

endmodule
```

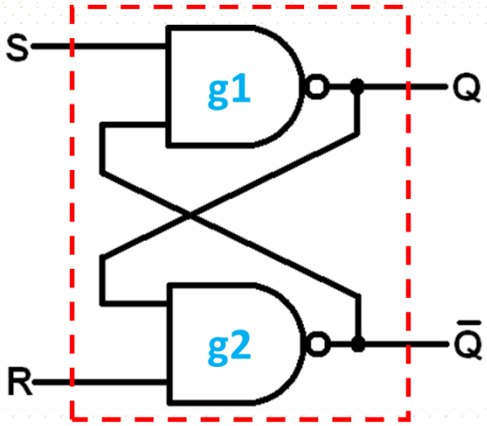
NAND Latch Testbench?

```
/* module name
and NO port list */
module nand_rs_tb;
 wire q,qb;
 reg s,r;

//instantiate the design
module (nand_rs
i1(q,qb,s,r);)
```

Correction :-

nand_rs i1 (q,qb,s,r);

```
endmodule
```



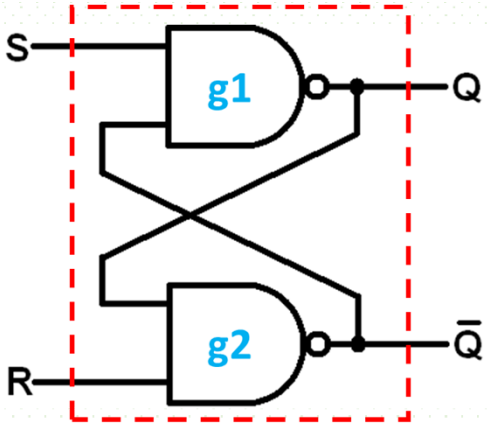
NAND Latch

```
module nand_rs(q,qb,s,r);
//port declaration
 output q,qb;
 input s,r;

//logic
nand #1 g1(q,s,qb);
nand #1 g2(qb,q,r);

endmodule
```

NAND Latch Testbench?

```
/* module name
and NO port list */
module nand_rs_tb;
 wire q,qb;
 reg s,r;

//instantiate the design
module nand_rs
i1(q,qb,s,r);

endmodule
```

t		$t+1$		
S	R	Q	$\bar{Q}$	
0	0	?	?	race
0	1	1	0	set
1	0	0	1	reset
1	1	Q	$\bar{Q}$	hold



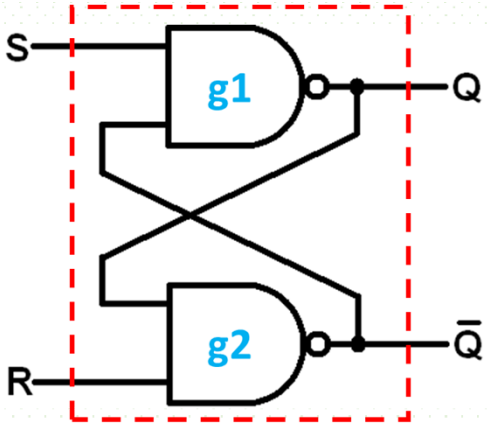
NAND Latch

```
module nand_rs(q,qb,s,r);
//port declaration
 output q,qb;
 input s,r;

//logic
nand #1 g1(q,s,qb);
nand #1 g2(qb,q,r);

endmodule
```

NAND Latch Testbench?

```
/* module name
and NO port list */
module nand_rs_tb;
 wire q,qb;
 reg s,r;

//instantiate the design
module nand_rs
i1(q,qb,s,r);

initial




endmodule
```

t		$t+1$		
S	R	Q	$\bar{Q}$	
0	0	?	?	race
0	1	1	0	set
1	0	0	1	reset
1	1	Q	$\bar{Q}$	hold



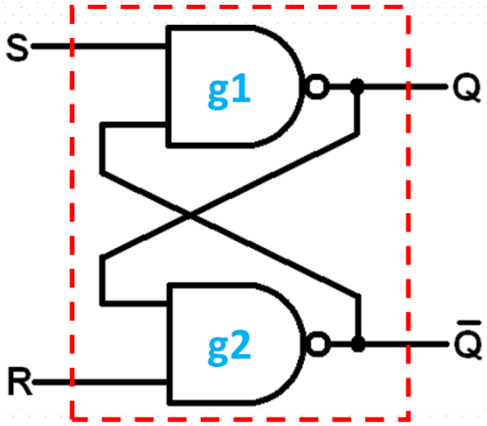
NAND Latch

```
module nand_rs(q,qb,s,r);
//port declaration
 output q,qb;
 input s,r;

//logic
nand #1 g1(q,s,qb);
nand #1 g2(qb,q,r);

endmodule
```

NAND Latch Testbench?

```
/* module name
and NO port list */
module nand_rs_tb;
 wire q,qb;
 reg s,r;

//instantiate the design
module nand_rs
i1(q,qb,s,r);

initial
 begin



 end

endmodule
```

t		$t+1$		
S	R	Q	$\bar{Q}$	
0	0	?	?	race
0	1	1	0	set
1	0	0	1	reset
1	1	Q	$\bar{Q}$	hold



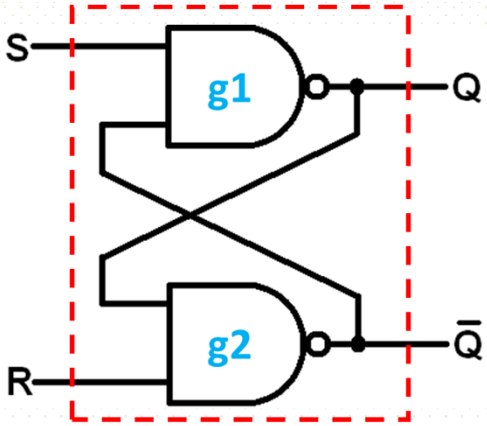
NAND Latch

```
module nand_rs(q,qb,s,r);
//port declaration
 output q,qb;
 input s,r;

//logic
nand #1 g1(q,s,qb);
nand #1 g2(qb,q,r);

endmodule
```

NAND Latch Testbench?

```
/* module name
and NO port list */
module nand_rs_tb;
 wire q,qb;
 reg s,r;

//instantiate the design
module nand_rs
i1(q,qb,s,r);

initial
 begin
   s=0; r=1;



 end

endmodule
```

t		$t+1$		
S	R	Q	$\bar{Q}$	
0	0	?	?	race
0	1	1	0	set
1	0	0	1	reset
1	1	Q	$\bar{Q}$	hold



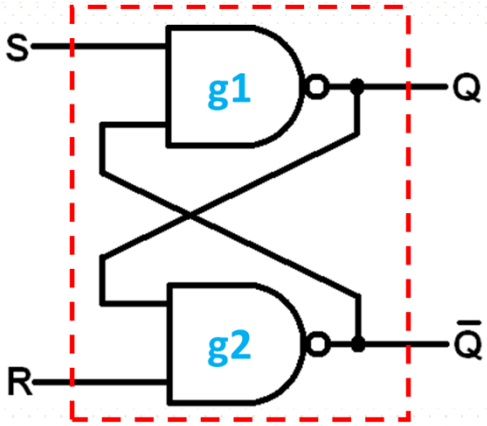
NAND Latch

```
module nand_rs(q,qb,s,r);
//port declaration
 output q,qb;
 input s,r;

//logic
nand #1 g1(q,s,qb);
nand #1 g2(qb,q,r);

endmodule
```

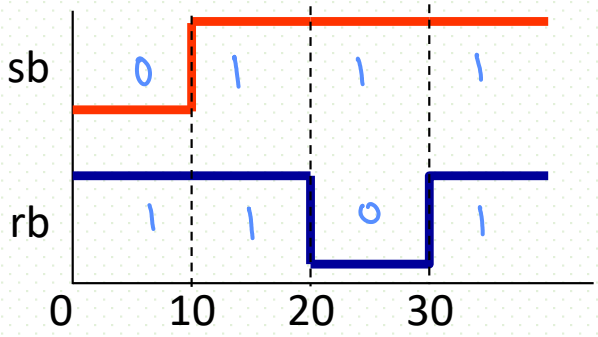
NAND Latch Testbench?

```
/* module name
and NO port list */
module nand_rs_tb;
 wire q,qb;
 reg s,r;

//instantiate the design
module nand_rs
i1(q,qb,s,r);

initial
 begin
   s=0; r=1;
   #10 s=1; r=1;
   #10 s=1; r=0;
   #10 s=1; r=1;
 end

endmodule
```

t		$t+1$		
S	R	Q	$\bar{Q}$	
0	0	?	?	race
0	1	1	0	set
1	0	0	1	reset
1	1	Q	$\bar{Q}$	hold



Test Vectors



Truth Table

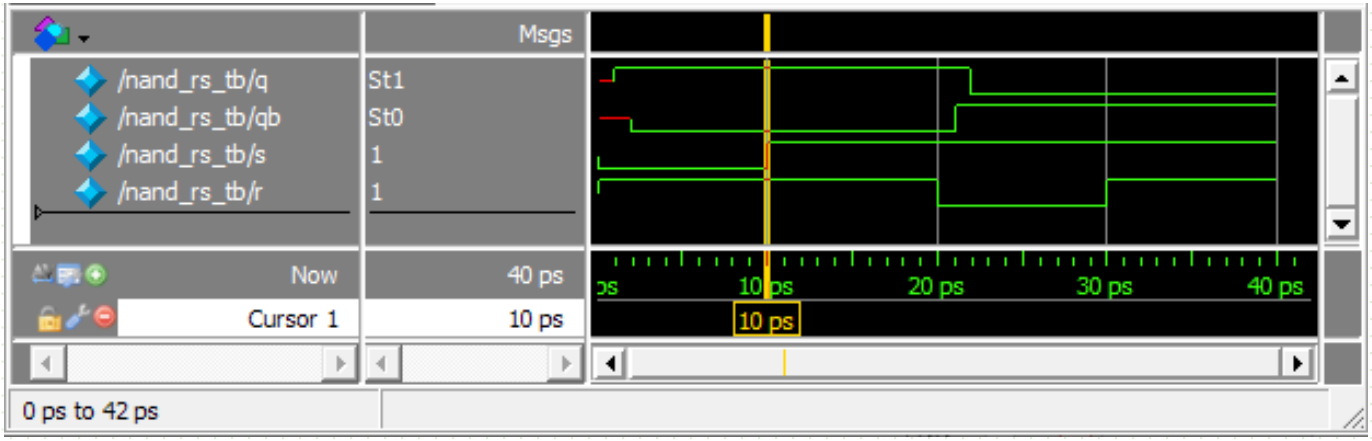
t		$t+1$		
S	R	Q	$\bar{Q}$	
0	0	?	?	race
0	1	1	0	set
1	0	0	1	reset
1	1	Q	$\bar{Q}$	hold

NAND Latch Testbench?

```
/* module name
and NO port list */
module nand_rs_tb;
 wire q,qb;
 reg s,r;

//instantiate the design
module
nand_rs i1(q,qb,s,r);

initial
 begin
   s=0; r=1;
   #10 s=1; r=1;
   #10 s=1; r=0;
   #10 s=1; r=1;
 end

endmodule
```

Display Results in Waveform

t		$t+1$		
S	R	Q	$\bar{Q}$	
0	0	?	?	race
0	1	1	0	set
1	0	0	1	reset
1	1	Q	$\bar{Q}$	hold

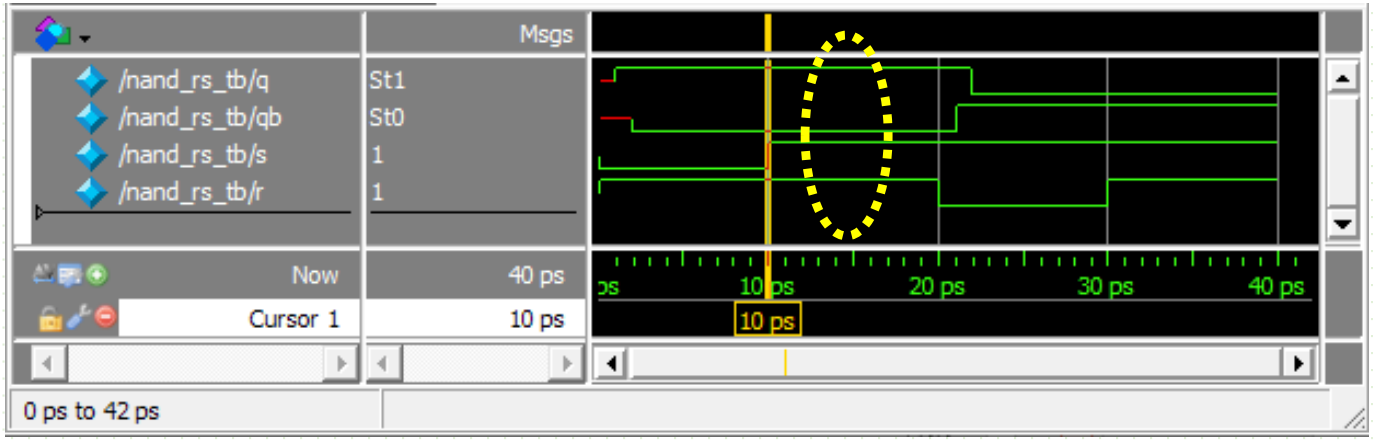


Display Results in Waveform

t		$t+1$		
S	R	Q	$\bar{Q}$	
0	0	?	?	race
0	1	1	0	set
1	0	0	1	reset
1	1	Q	$\bar{Q}$	hold

hold

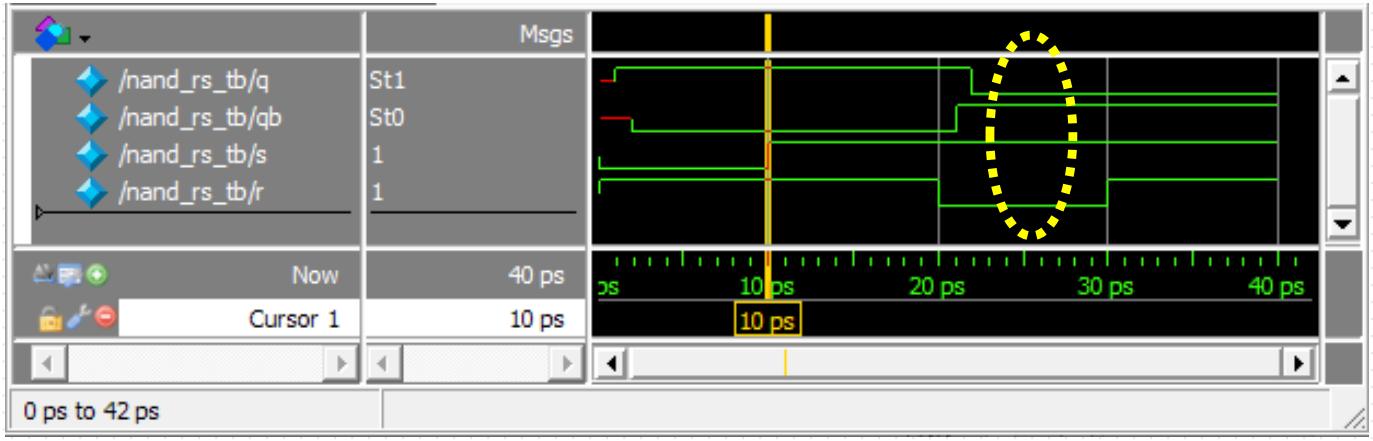
$q = 1, \bar{q} = 0, s = 0, r = 1;$ ~~set~~



Display Results in Waveform

t		$t+1$		
S	R	Q	$\bar{Q}$	
0	0	?	?	race
0	1	1	0	set
1	0	0	1	reset
1	1	Q	$\bar{Q}$	hold

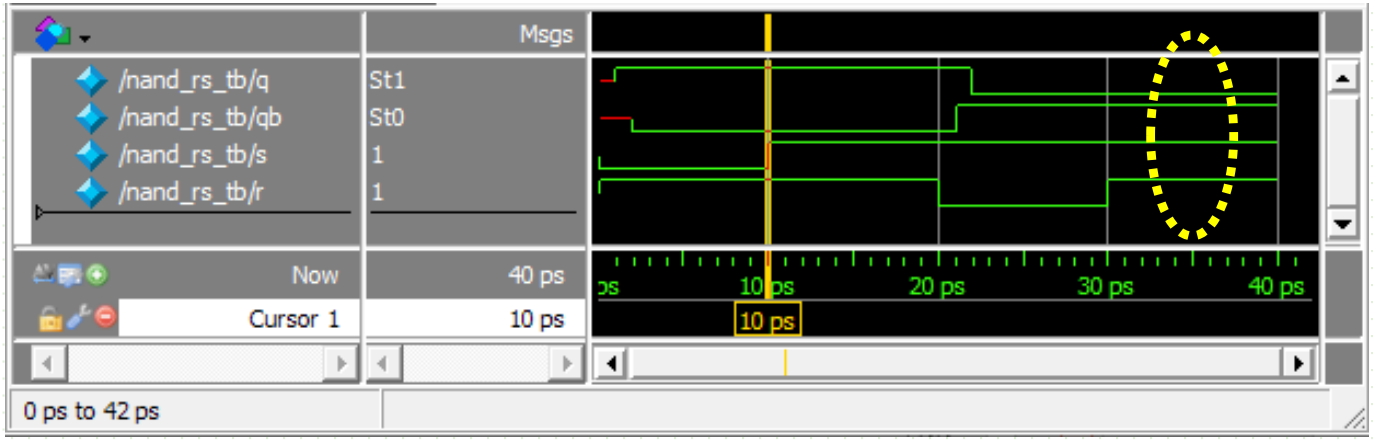
$$q = 0, \bar{q} = 1, s = 1, r = 0; reset$$



Display Results in Waveform

t		$t+1$		
S	R	Q	$\bar{Q}$	
0	0	?	?	race
0	1	1	0	set
1	0	0	1	reset
1	1	Q	$\bar{Q}$	hold

$$q = 0, \bar{q} = 1, s = 1, r = 1; \boldsymbol{hold}$$

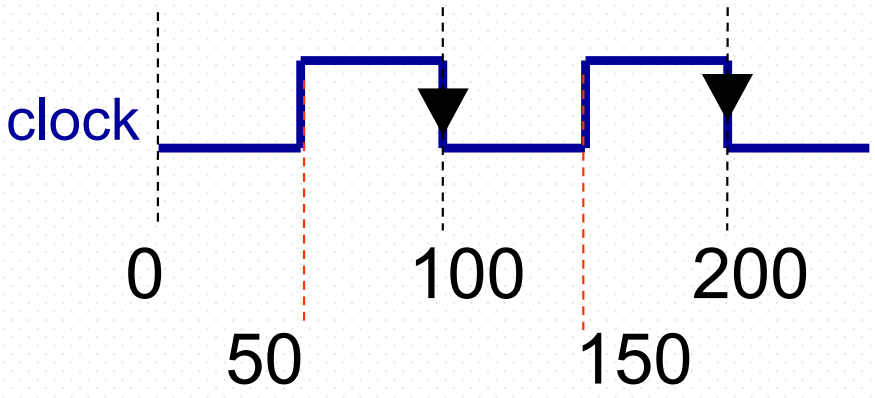


Other Verilog Syntex

Clock

How to describe clock using always statement?

```
/* initialize clock to logic 0, at
time 0 */
initial clock= 0;
always #50 clock= !clock;
```



Nets or wire

- Nets represent connections between hardware elements.
- Nets are declared with the keyword wire.
- Nets are one-bit values by default unless they are declared.
- explicitly as vectors.
- E.g.: wire a, b, c; // Declare wires a, b, c for a circuit

Registers or reg

- Registers represent data storage elements.
- Registers retain value until another value is placed onto them.
- E.g: reg reset; //Declare a variable reset that can hold its value

Scalar and Vector

- Wire or reg data type can be declared as vectors (multiple bit widths).
- If bit width is not specified, the default is scalar data type (1-bit).
- E.g.:

```
wire  a; // scalar net
wire [7:0] data; //  8-bit data, where MSB is data[7] and LSB is  data[0].
reg x,y; // scalar register x and y.
reg [0:11] addr_bus;  // vector register, 12-bit addr_bus with addr_bus[0] as MSB and addr_bus[11] as the LSB.
```

Primitive Gates

- A Logic circuit can be designed by use of logic gates.
- Verilog HDL supports basic logic gates as predefined primitives.
- E.g.:

not, and, nand, or, nor, xor and xnor

initial

- The initial block starts at time 0, executes exactly once during a simulation, and then does not execute again.
- If there are multiple initial blocks, each block starts to execute concurrently at time 0.
- Multiple statements must be grouped together using the keywords begin and end.

`timescale

- It is noted that simulation times have been described in terms of “time units”.
- The timescale compiler directive is used to attached units and a precision to these numbers.
- E.g.: `timescale 1ns/100ps
- The delay in this module are in units of **nanosecond**
- They are precise to the **hundred picosecond** digit (any time calculations would be internally rounded to the nearest one hundred picoseconds.)

`timescale

Unit/Precision	Delay Spec	Time Delayed	Notes
10ns/1ns	#7	70 ns	7x10 ns = 70 ns (1 ns precision)
10ns/1ns	#7.748	77 ns	7.748x10 ns = 77.48 = 77 ns (1 ns precision)
10ns/100ps	#7.748	77.5 ns	7.748x10 ns = 77.48 = 77.5 ns (0.1 ns precision)
10ns/10ns	#7.5	80 ns	7.5x10 ns = 75 = 80 ns (10 ns precision)

unit

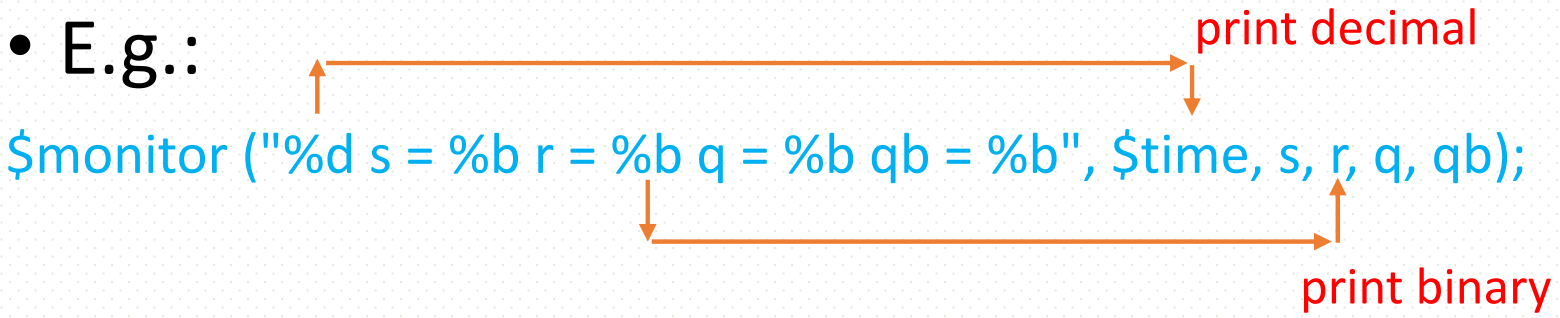
precision

0.1ns

Display Text Results

$monitor

- The quoted string is printed with the values of s, r, q and qb substituted for the %b (for binary) printing control in the string.
- E.g.:



FPGA Software Setup - Vivado

- Register an account with Xilinx
  - You need this registration for download, and installation
- Download Vivado-design-tools
  - 2023.1
- Please refer to the below Guide for detail setup

https://pe8sutd.larksuite.com/docx/NfZid9uy5oiYvbxoEWMui94HsfY?from=from_copylink

You need to start the installation now, as it may take a long time to install.

20 minutes to kick start the simulation, and let it runs.

We will be using Vivado for simulation.

Verilog HDL Simulation (Optional)

- Refer to the below document for Modelsim installation and video tutorial.

https://pe8sutd.larksuite.com/docx/Fqk7drHjXoq73bxwCq2uV5Smsee?from=from_copylink

We will be using Vivado for simulation.

You can also use Modelsim for simulation and later use Vivado for synthesis, and implementation.

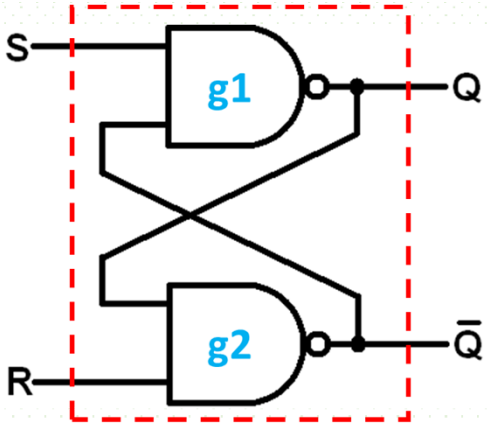
Handson - E1-NORLatch

- Refer to the below document for this Verilog Coding exercise.

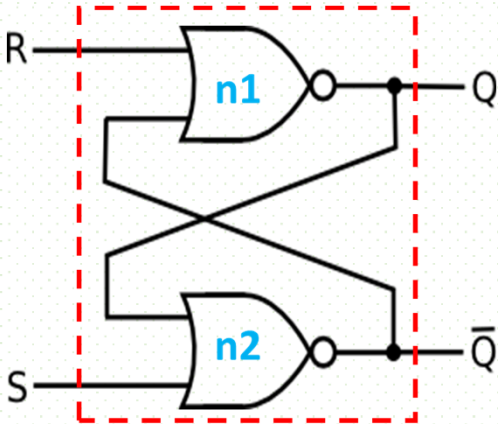
https://pe8sutd.larksuite.com/docx/Z6yRdD0yPoPnJqx5vQ0u5j67sfg?from=from_copylink

Handson Verilog – NOR Latch

- Refer to the NAND latch, perform coding of a NOR latch module, and a testbench.
- Perform the simulation in Modelsim or Vivado, to verify the functionality of the NOR Latch.

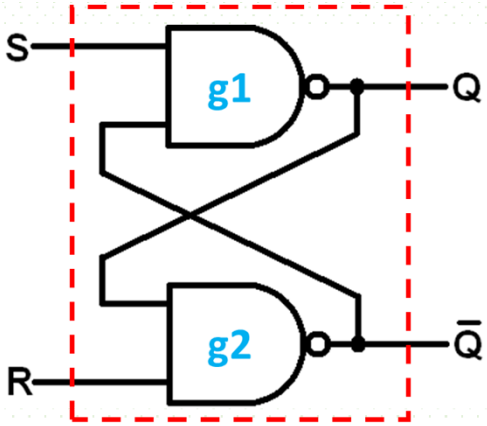


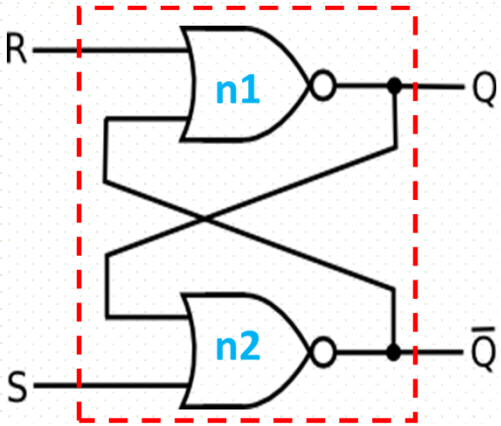
NAND Latch



NOR Latch?

```
module nand_rs(q,qb,s,r);
//port declaration
 output q,qb;
 input s,r;

//logic
nand #1 g1(q,s,qb);
nand #1 g2(qb,q,r);

endmodule
```



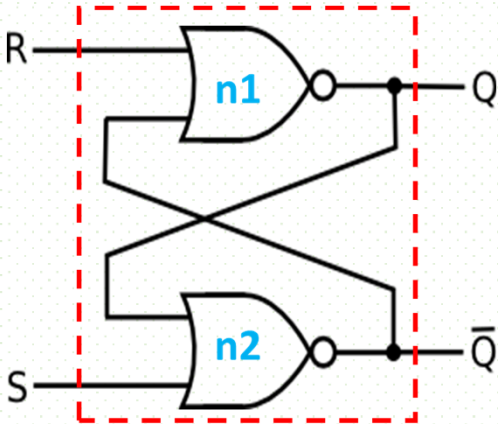


NAND Latch

```
module nand_rs(q,qb,s,r);
//port declaration
 output q,qb;
 input s,r;

//logic
nand #1 g1(q,s,qb);
nand #1 g2(qb,q,r);

endmodule
```

NOR Latch?

```
module nor_rs(q,qb,r,s);
//port declaration
 output q,qb;
 input r,s;

// logic
nor #1 n1(q,r,qb);
nor #2 n2(qb,q,s);

endmodule
```



NOR Latch Truth Table

t		$t+1$		
S	R	Q	$\bar{Q}$	
0	0	Q	$\bar{Q}$	hold
0	1	0	1	reset
1	0	1	0	Set
1	1	?	?	race

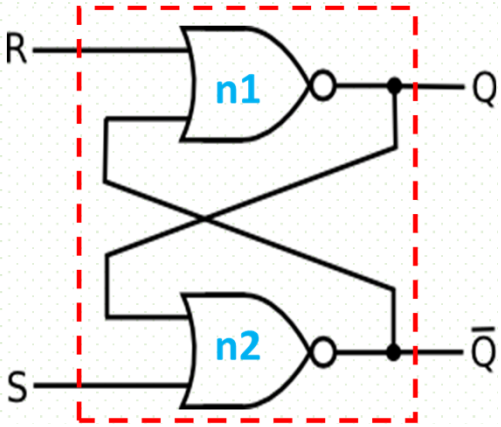
NOR Latch?

```
module nor_rs(q,qb,r,s);
//port declaration
 output q,qb;
 input r,s;

// logic
nor #1 n1(q,r,qb);
nor #2 n2(qb,q,s);

endmodule
```

NOR Latch Testbench

```
module nor_rs_tb;
 wire q,qb;
 reg r,s;
/* instantiate the design
module */
nor_rs i1(q,qb,r,s);

 initial
  begin
   r=1;s=0;
   #10 r=0;s=0;
   #10 r=0;s=1;
   #10 r=0;s=0;
  end

endmodule
```

t		$t+1$		
S	R	Q	$\bar{Q}$	
0	0	Q	$\bar{Q}$	hold
0	1	0	1	reset
1	0	1	0	Set
1	1	?	?	race

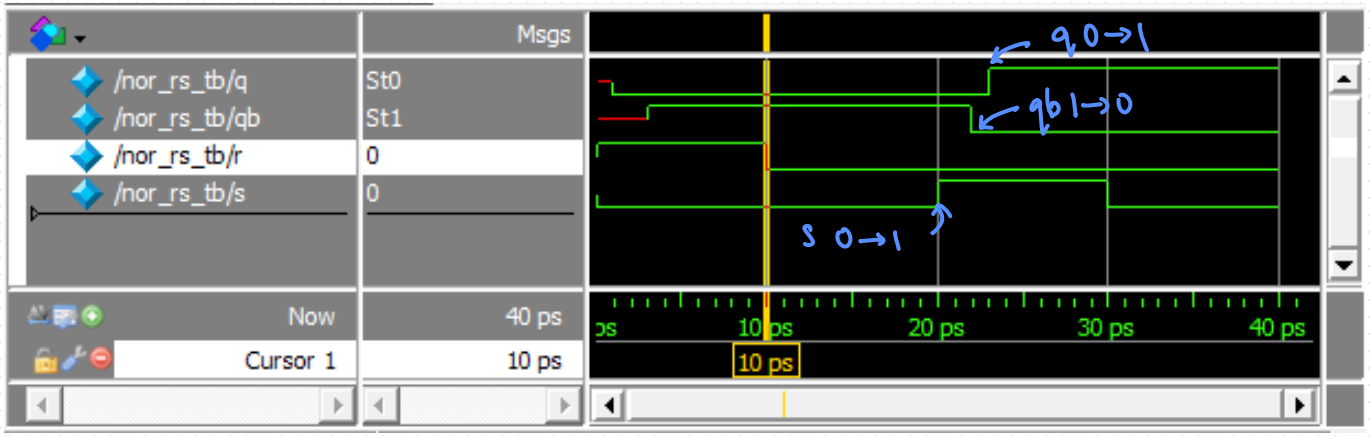


NOR Latch

```
module nor_rs(q,qb,r,s);
//port declaration
 output q,qb;
 input r,s;

// logic
nor #1 n1(q,r,qb);
nor #2 n2(qb,q,s);

endmodule
```

Display Results in Text (NOR Latch)

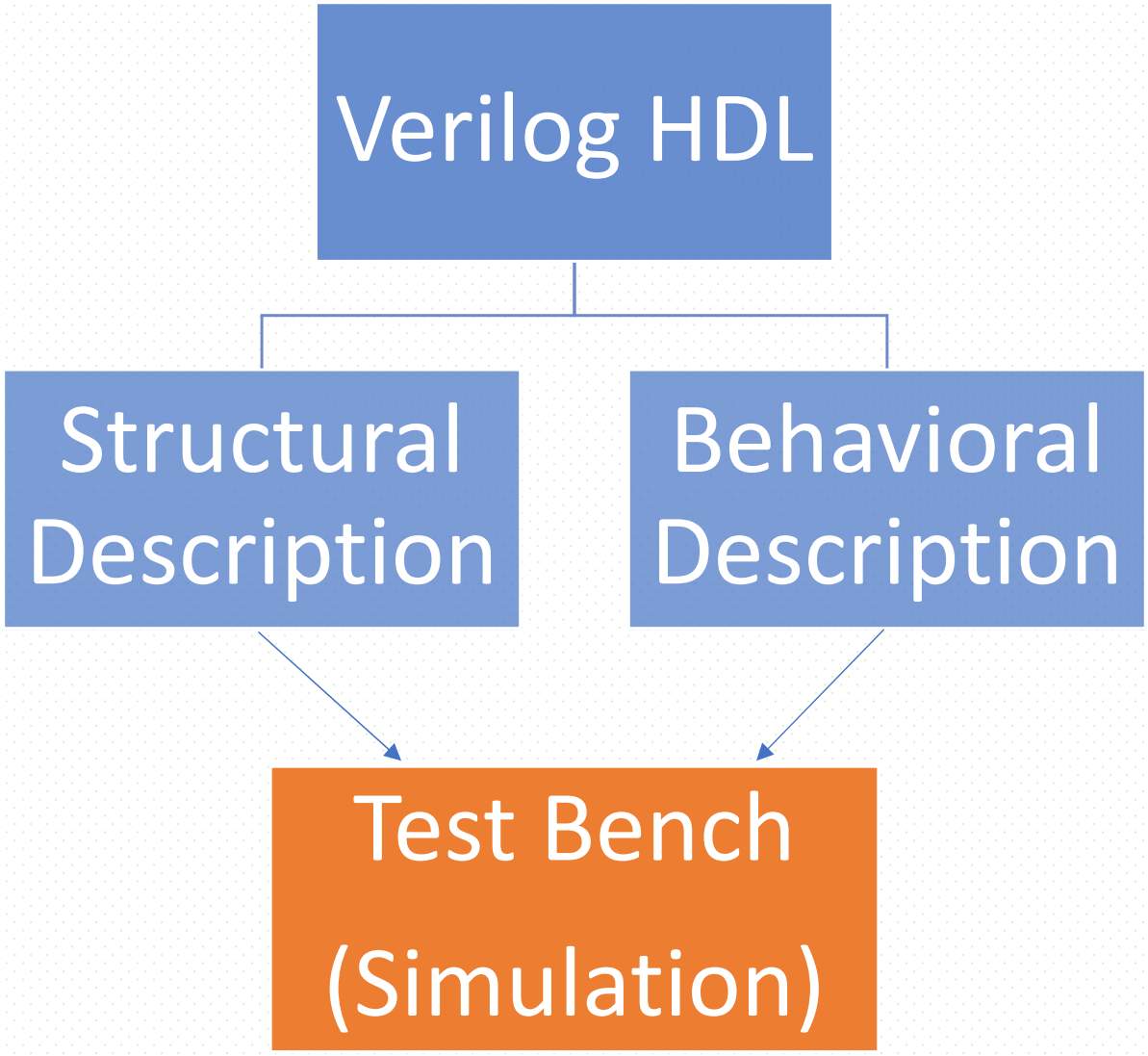
t		$t+1$		
S	R	Q	$\bar{Q}$	
0	0	Q	$\bar{Q}$	hold
0	1	0	1	reset
1	0	1	0	Set
1	1	?	?	race



Week05 – Lesson 1 Plan

- Behavioral Modeling

Verilog HDL


Verilog HDL
Structural Description
Behavioral Description
Test Bench (Simulation)

Behavioral Modelling

- Behavioral Verilog provides designers the ability to describe the behaviour of the circuit.
- Thus, behavioral modelling represents the circuit at a very high level of abstraction.

Conditional Statements

- ***if-else-if***
- Conditional statements such as if statement and its variations are used in a sequential behavior description to alter the flow of control.
- The relational operators typically used in conditional expressions are:
    - `!` (logical negation/not)
    - `>` (greater than)
    - `>=` (greater than or equal)
    - `==` (equal)
    - `!=` (not equal)
    - `<` (less than)
    - `<=` (less than or equal)

Conditional Statements

```
if (divisor==1)
        begin
           // ... statements 1
        end
else
        begin
           // ... statements 2
        end
```

- If ***divisor is TRUE***, statements 1 will be executed.
- If ***divisor is FALSE***, statements 2 will be executed.

Loop Statement

for (initial condition; end of loop statement; loop update statement)

```
for (i=16; i<18 ; i=i-1)
        begin
            ... // statements
        end
```

Loop Statement

```
initial condition;
while (end of loop statement)
        begin
            loop update statement
        end
```

```
i=16;
while (i>0)
        begin
            .../// statements
            i=i-1;
        end
```

case Statement

```
case (controlling expression)
        case expression : statement;
endcase
```

```
case (select)
        2'b00:  begin   … end
        2'b01:  begin   … end
        2'b10:  begin   … end
        2'b11:  begin   … end
endcase
```

Concatenation operator

The concatenation operator (***{}***) provides a mechanism to append multiple operands, to form a single word.

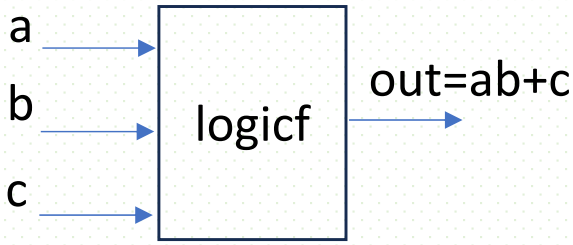
A = 1’b1; B = 2’b00; C = 2’b10; D = 3’b110;

Y = {B , C} // 4’b0010

Y = {A, B, C, D, 3’b001} // 11’b10010110001

always Statement

- The ***always*** continuously repeats its statement, never exiting or stopping.
- A behavioral model may contain one or more ***always*** statements.
- In terms of a design process using synthesis tools, all of the functionality of the module should be specified within the ***always*** statement.
- The ***always*** statement, essentially a "while (TRUE)" statement, includes one or more procedural statements that are repeatedly executed.

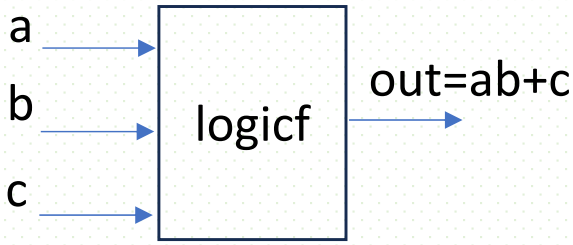


Structural Description

```
module logicfs(out,a,b,c);
 output out;
 input a,b,c;

and g1 (ab,a,b);
or g2 (out,ab,c);

endmodule
```

Behavioural Description

```
module logicfb(out,a,b,c);
 output out;
 reg out;
 input a,b,c;

always@(a or b or c)
 begin
  out=a&b|c;
 end

endmodule
```



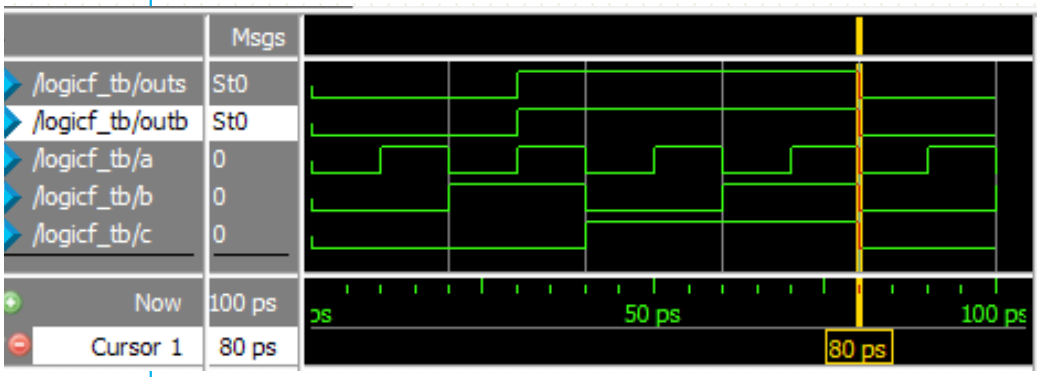
always Statement

Output, ***outb*** will change if any of inputs ***a***, ***b***, or ***c*** changes.

Notes:

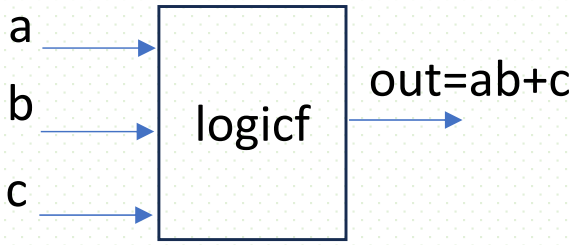
outb in testbench for logicfb.

outs in testbench for logicfs



Behavioural Description

```
module logicfb(out,a,b,c);
 output out;
 reg out;
 input a,b,c;

always@(a or b or c)
 begin
  out=a&b|c;
 end

endmodule
```



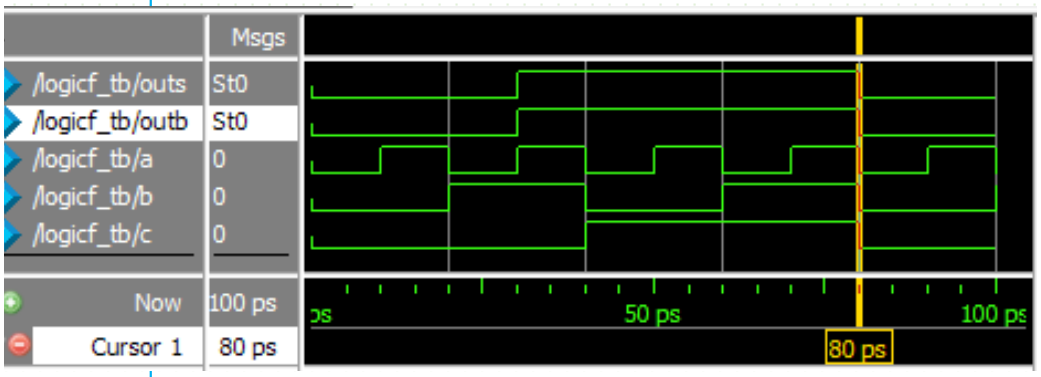
always Statement

Output, ***outb*** will change if any of inputs changes.

Notes:

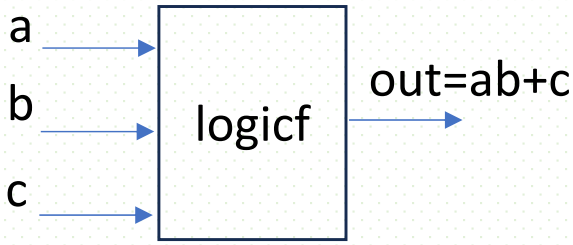
outb in testbench for logicfb.

outs in testbench for logicfs



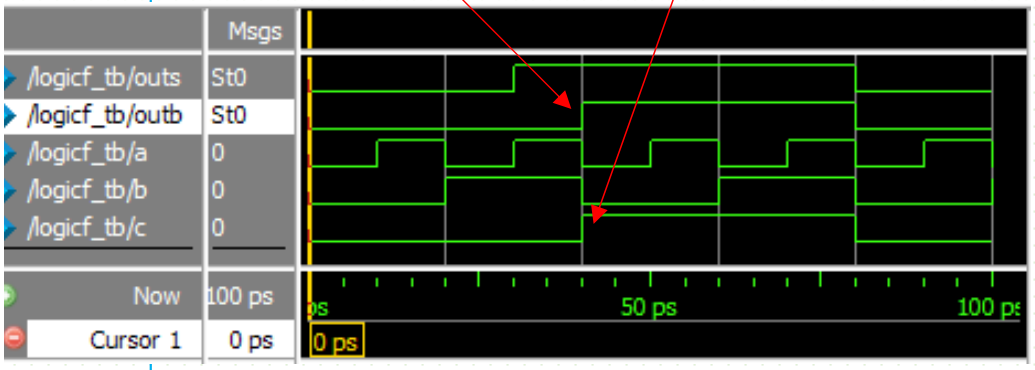
Behavioural Description

```
module logicfb(out,a,b,c);
 output out;
 reg out;
 input a,b,c;

always@*
 begin
  out=a&b|c;
 end

endmodule
```



always Statement

Output, ***outb*** will change if ***c*** changes.

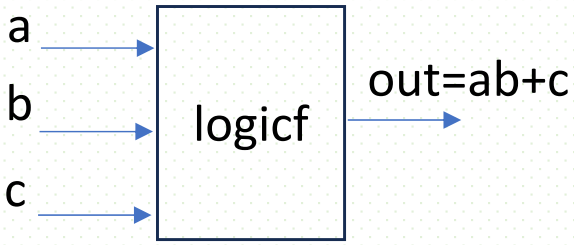


Behavioural Description

```
module logicfb(out,a,b,c);
 output out;
 reg out;
 input a,b,c;

always@(c)
 begin
  out=a&b|c;
 end

endmodule
```

Testbench

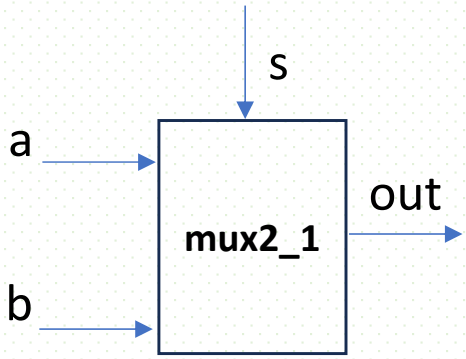
```
module logicf_tb;
 wire outs,outb;
 reg a,b,c;

logicfs i1 (outs,a,b,c);
logicfb i2 (outb,a,b,c);

initial a=0; always #10 a=!a;
initial b=0; always #20 b=!b;
initial c=0; always #40 c=!c;

endmodule
```



Behavioural Description

```
module logicfb(out,a,b,c);
 output out;
 reg out;
 input a,b,c;

always@(c)
 begin
  out=a&b|c;
 end

endmodule
```

Multiplexor

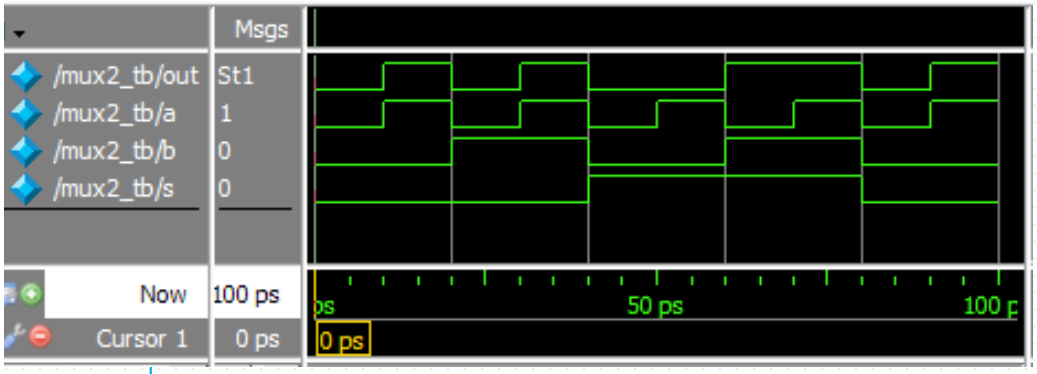


Multiplexor

$$out = as + b\bar{s}$$

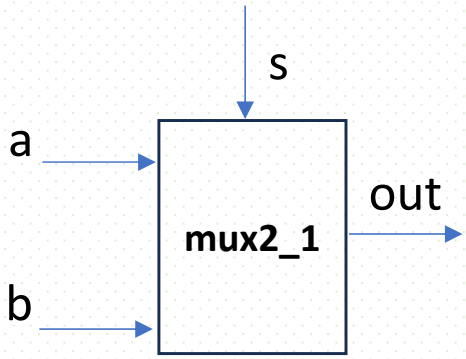
When $s = 1$, a is selected

When $s = 0$, b is selected



Behavioural Description for multiplexor 2 to 1

```
module mux2(out,a,b,s);
 output out;
 input a,b,s;
 reg out;

always @(a or b or s)
 begin
   if (s==0) out=a;
   else out=b;
 end

endmodule
```



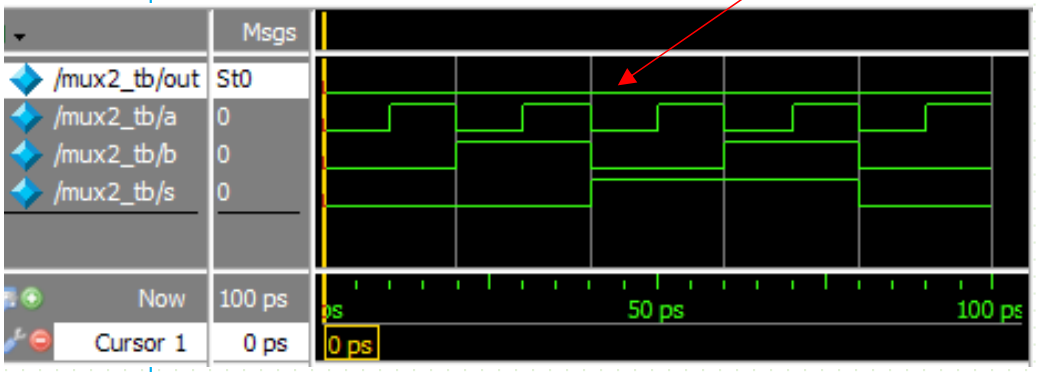
Multiplexor

$$out = as + b\bar{s}$$

When $s = 1$, a is selected

When $s = 0$, b is selected

out has no update



Behavioural Description for multiplexor 2 to 1

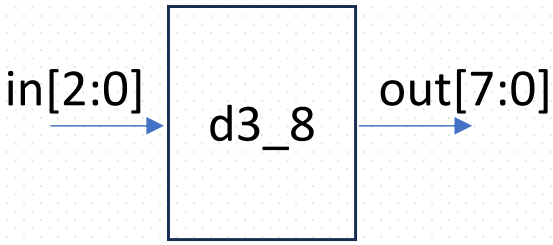
```
module mux2(out,a,b,s);
 output out;
 input a,b,s;
 reg out;

always @(s)
 begin
   if (s==0) out=a;
   else out=b;
 end

endmodule
```

What happen if you did not include ***a*** or ***b*** in the sensitivity list ?

Decoder

Behavioural Description for decoder 3 to 8

```
module d3_8(out,in);
 output [7:0] out;
 input [2:0] in;
 reg [7:0] out;

//always @(in[2] or in[1] or in[0])
always @(in)
 begin
//  case({in[2],in[1],in[0]})
  case(in)
   3'b000: out=8'h01;
   3'b001: out=8'h02;
   3'b010: out=8'h04;
   3'b011: out=8'h08;
   3'b100: out=8'h10;
   3'b101: out=8'h20;
   3'b110: out=8'h40;
   3'b111: out=8'h80;
  endcase
 end

endmodule
```

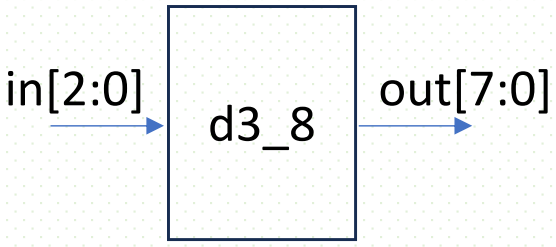


Testbench

```
module d3_8_tb;
 wire [7:0] out;
 reg [2:0] in;

d3_8 i1 (out,in);

initial in[2]=0; always #40 in[2]=!in[2];
initial in[1]=0; always #20 in[1]=!in[1];
initial in[0]=0; always #10 in[0]=!in[0];

initial
 begin
  #100 $stop;
  #110 $finish;
 end

endmodule
```

Behavioural Description for decoder 3 to 8

```
module d3_8(out,in);
 output [7:0] out;
 input [2:0] in;
 reg [7:0] out;

//always @(in[2] or in[1] or in[0])
always @(in)
 begin
//  case({in[2],in[1],in[0]})
  case(in)
   3'b000: out=8'h01;
   3'b001: out=8'h02;
   3'b010: out=8'h04;
   3'b011: out=8'h08;
   3'b100: out=8'h10;
   3'b101: out=8'h20;
   3'b110: out=8'h40;
   3'b111: out=8'h80;
  endcase
 end

endmodule
```

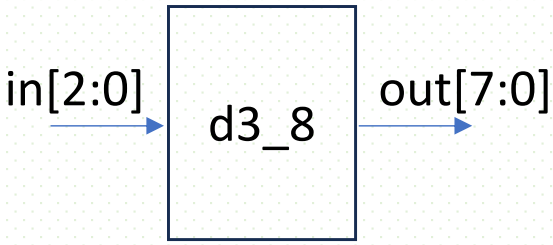


Truth Table

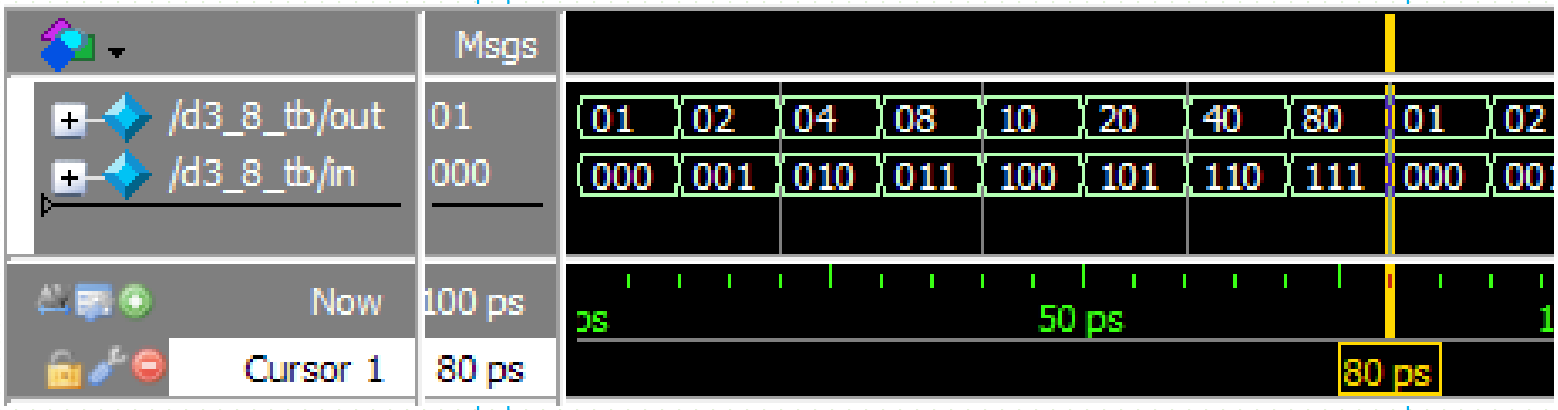
in[2:0]	out[7:0]	HEX
000	00000001	01
001	00000010	02
010	00000100	04
011	00001000	08
100	00010000	10
101	00100000	20
110	01000000	40
111	10000000	80

Behavioural Description for decoder 3 to 8

```
module d3_8(out,in);
 output [7:0] out;
 input [2:0] in;
 reg [7:0] out;

//always @(in[2] or in[1] or in[0])
always @(in)
 begin
//  case({in[2],in[1],in[0]})
  case(in)
   3'b000: out=8'h01;
   3'b001: out=8'h02;
   3'b010: out=8'h04;
   3'b011: out=8'h08;
   3'b100: out=8'h10;
   3'b101: out=8'h20;
   3'b110: out=8'h40;
   3'b111: out=8'h80;
  endcase
 end

endmodule
```



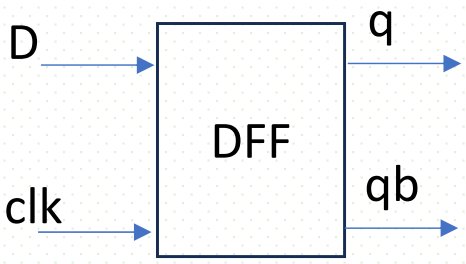
Testbench



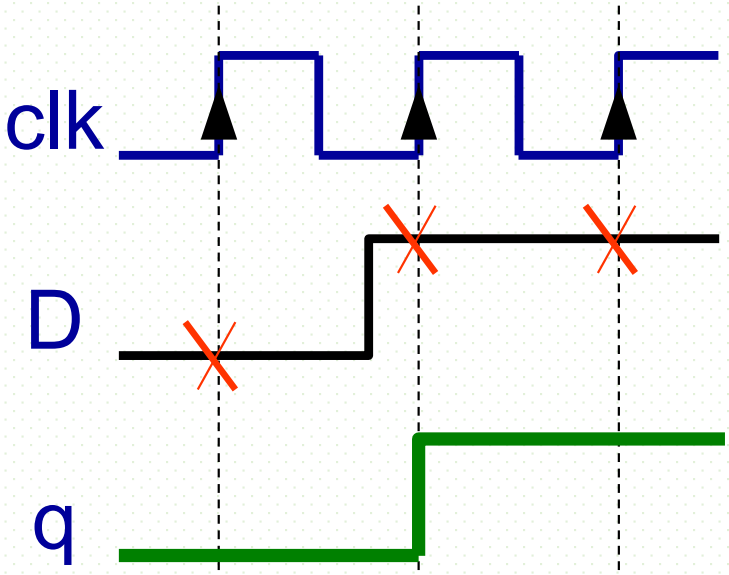
Week06 – Lesson 1 Plan

Synchronous

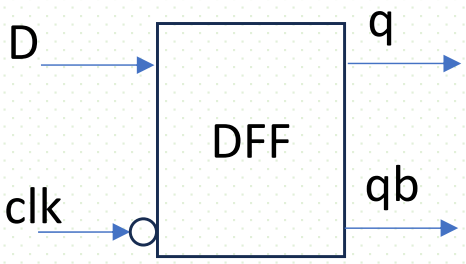
- Positive edge triggered clk
- Negative edge triggered clk
- Synchronous signal
- Asynchronous signal
- Active high/low signal



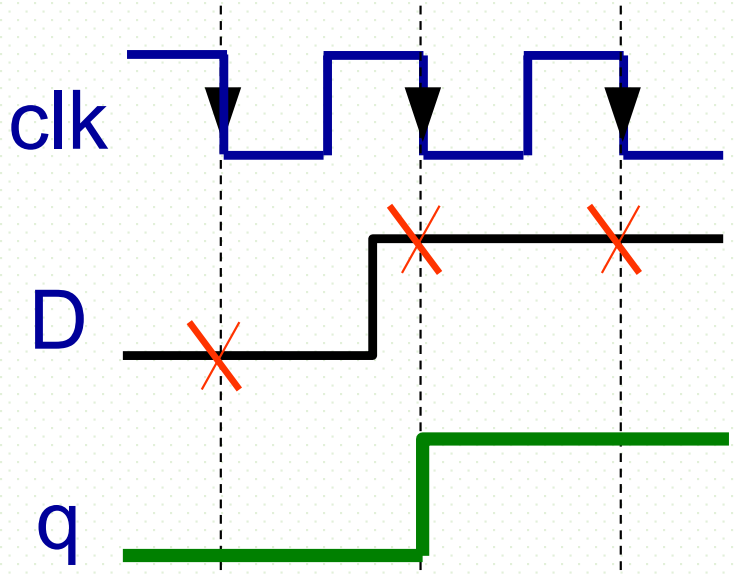
Positive edge triggered clk



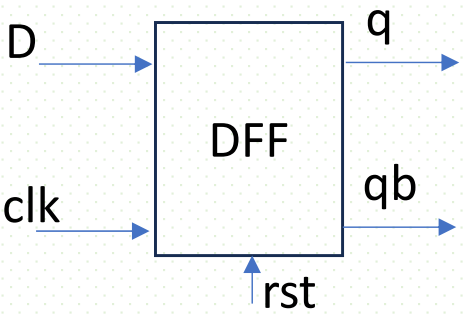

```
always @(posedge clk)
begin
  q = D;
  qb = !q;
end
```



Negative edge triggered clk

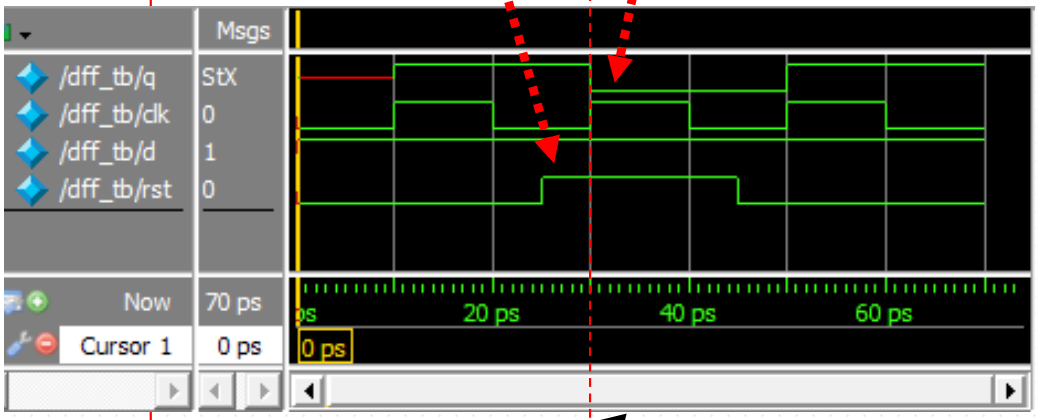



```
always @(negedge clk)
begin
  q = D;
  qb = !q;
end
```



Positive edge triggered clock and active high synchronous reset description

- when rst=1, q=0

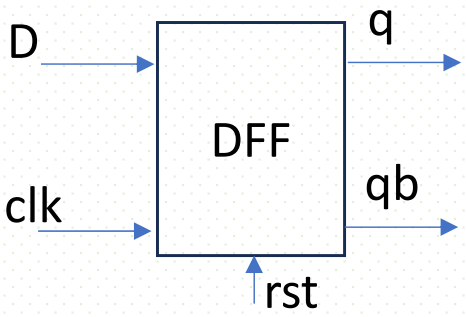


Reset will only activate with +ve edge clk

DFF (+ve edge, syn rst)

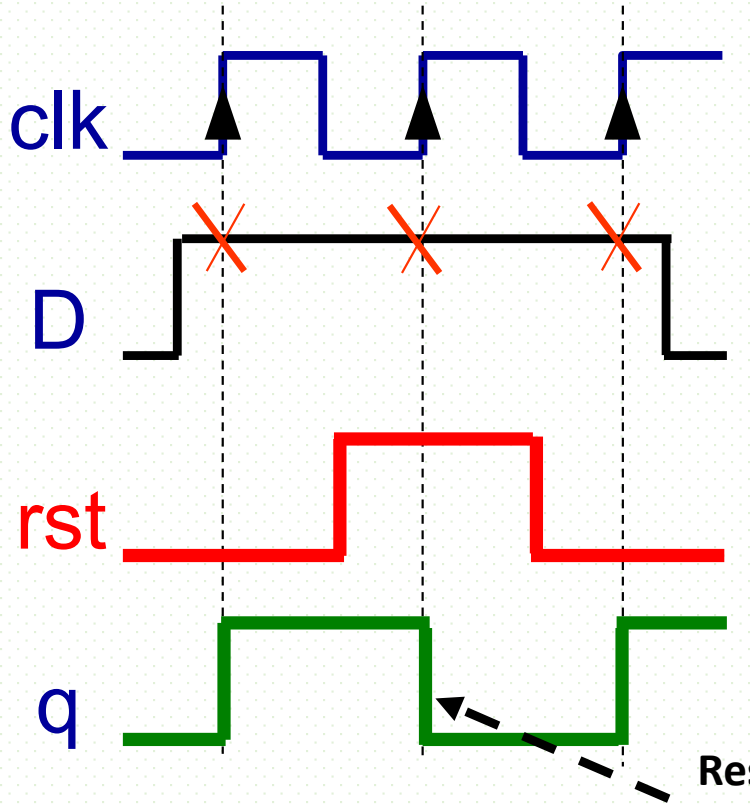
```
module dff(q,qb,clk,d,rst);
 output q,qb;
 input clk,d,rst;
 reg q,qb;

always @(posedge clk)
 begin
  if(rst) begin q=0;qb=!q; end
  else begin q=d;qb=!q; end
 end

endmodule
```

Positive edge triggered clock and active high synchronous reset description



DFF (+ve edge, syn rst)

- when rst=1, q=0

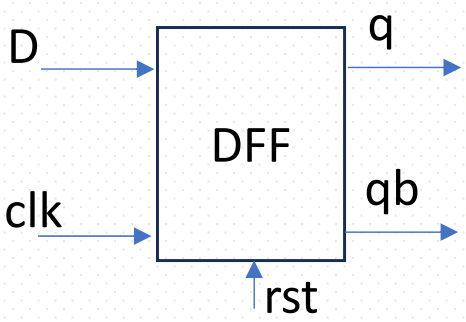


Reset will only activate with +ve edge clk

```
module dff(q,qb,clk,d,rst);
 output q,qb;
 input clk,d,rst;
 reg q,qb;

always @(posedge clk)
 begin
  if(rst) begin q=0;qb=!q; end
  else begin q=d;qb=!q; end
 end

endmodule
```

Testbench

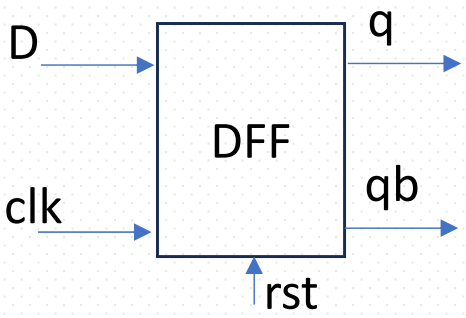
```
module dff_tb;
 wire q,qb;
 reg clk,d,rst;

dff i1 (q,qb,clk,d,rst);

initial clk=0; always #10 clk=!clk;

initial
 begin
  rst=0;d=1;
  #25 rst=1;
  #20 rst=0;
  #25 $stop;
  #25 $finish;
 end

endmodule
```



DFF (+ve edge, syn rst)

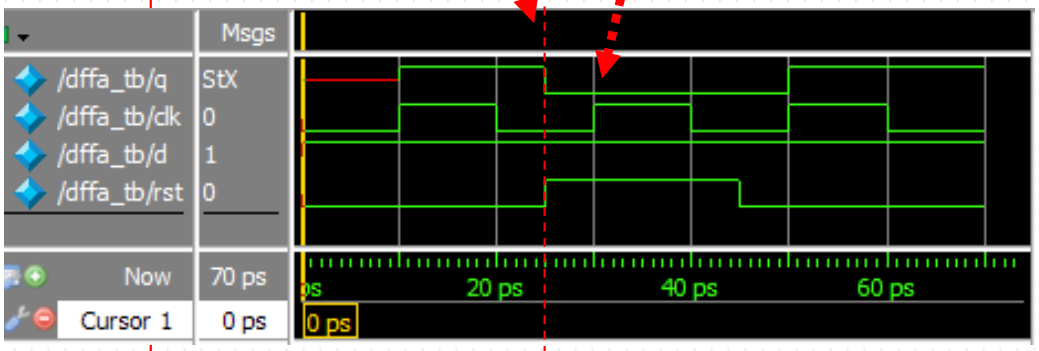
```
module dff(q,qb,clk,d,rst);
 output q,qb;
 input clk,d,rst;
 reg q,qb;

always @(posedge clk)
 begin
  if(rst) begin q=0;qb=!q; end
  else begin q=d;qb=!q; end
 end

endmodule
```

Positive edge triggered clock and active high asynchronous reset description



DFF (+ve edge, asyn rst)

- when rst=1, q=0

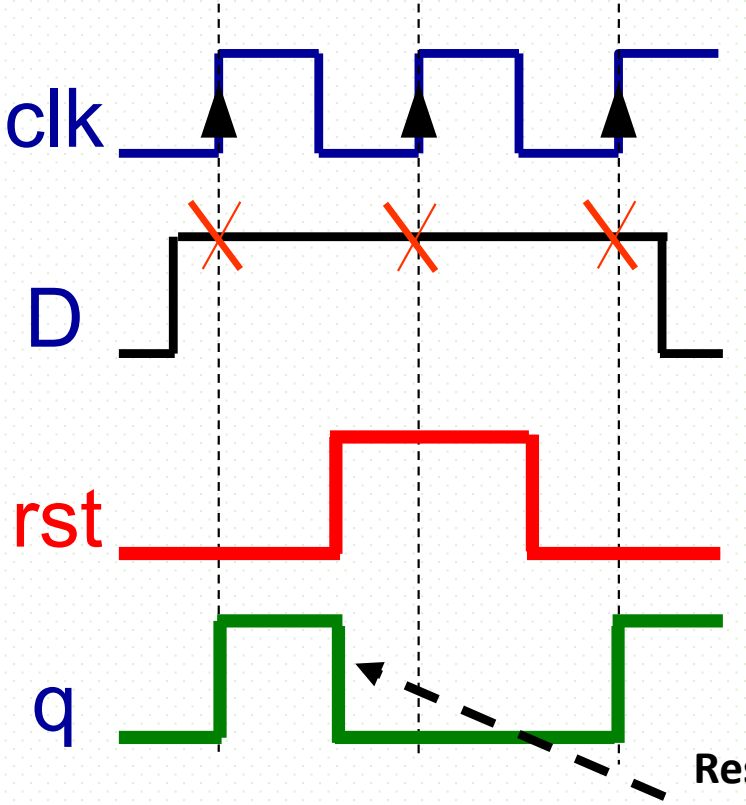


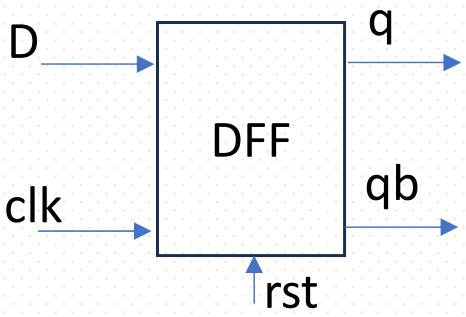
Reset will activate at its +ve edge

```
module dffa(q,qb,clk,d,rst);
 output q,qb;
 input clk,d,rst;
 reg q,qb;

always @(posedge clk or posedge rst)
 begin
  if(rst) begin q=0;qb=!q; end
  else begin q=d;qb=!q; end
 end

endmodule
```

Positive edge triggered clock and active high asynchronous reset description

- when rst=1, q=0





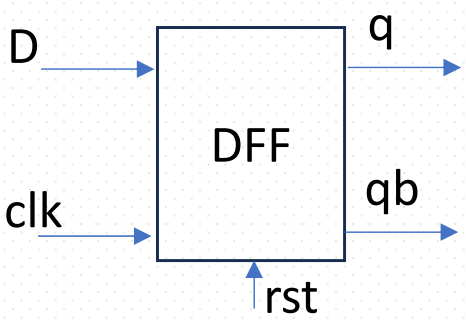
DFF (+ve edge, asyn rst)

```
module dffa(q,qb,clk,d,rst);
 output q,qb;
 input clk,d,rst;
 reg q,qb;

always @(posedge clk or posedge rst)
 begin
  if(rst) begin q=0;qb=!q; end
  else begin q=d;qb=!q; end
 end

endmodule
```

Reset will activate at its +ve edge

Testbench

```
module dff_tb;
 wire q,qb;
 reg clk,d,rst;

dff i1 (q,qb,clk,d,rst);

initial clk=0; always #10 clk=!clk;

initial
 begin
  rst=0;d=1;
  #25 rst=1;
  #20 rst=0;
  #25 $stop;
  #25 $finish;
 end

endmodule
```

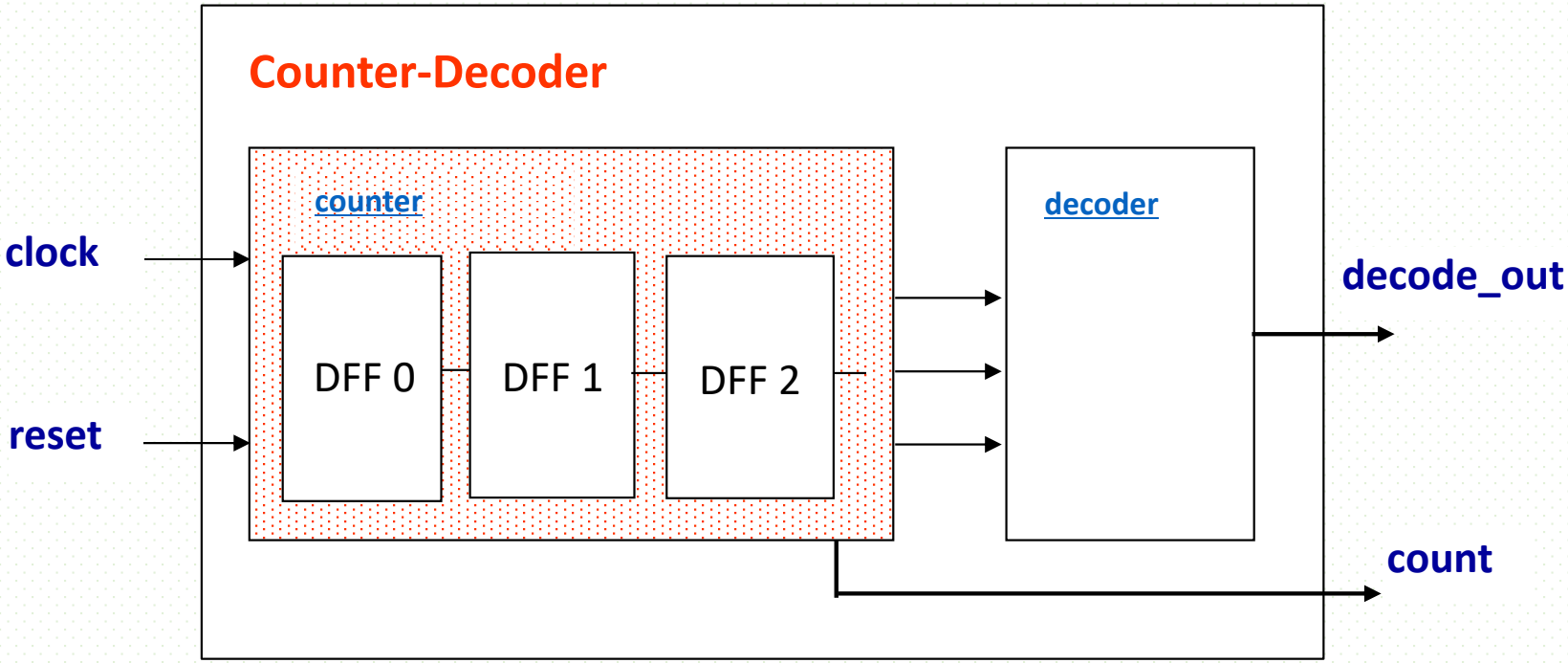


DFF (+ve edge, asyn rst)

```
module dffa(q,qb,clk,d,rst);
 output q,qb;
 input clk,d,rst;
 reg q,qb;

always @(posedge clk or posedge rst)
 begin
  if(rst) begin q=0;qb=!q; end
  else begin q=d;qb=!q; end
 end

endmodule
```

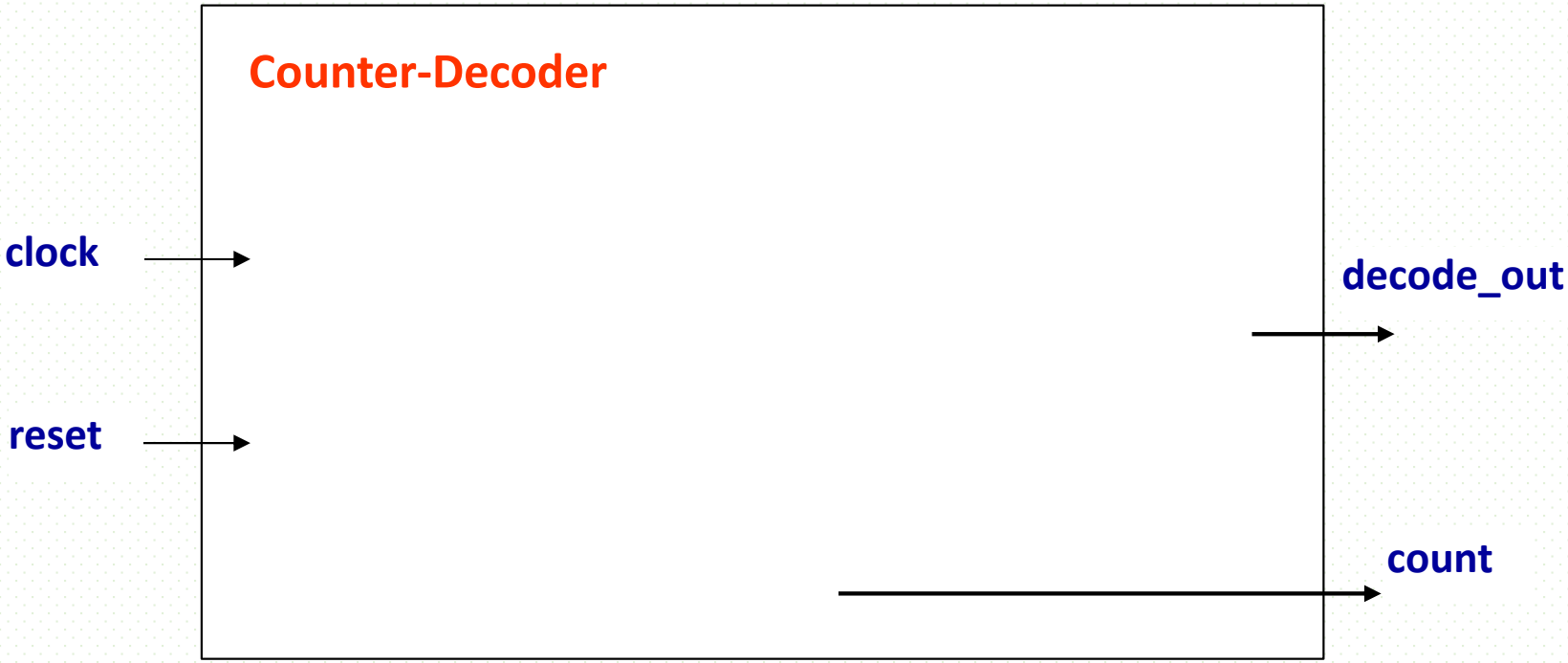
Module Hierarchy

- Figure illustrates pictorially a Counter-Decoder design.



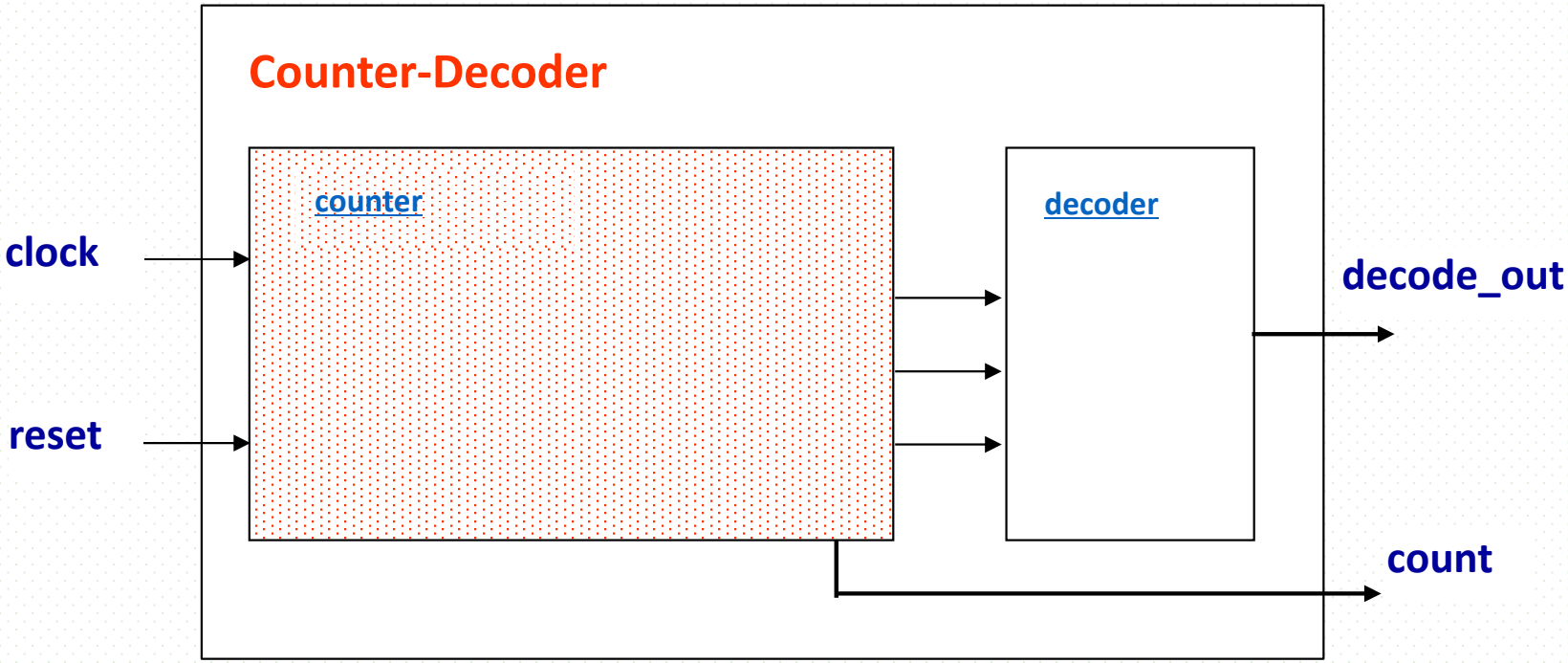
Module Hierarchy

- Figure illustrates pictorially a Counter-Decoder design. Top hierarchy.



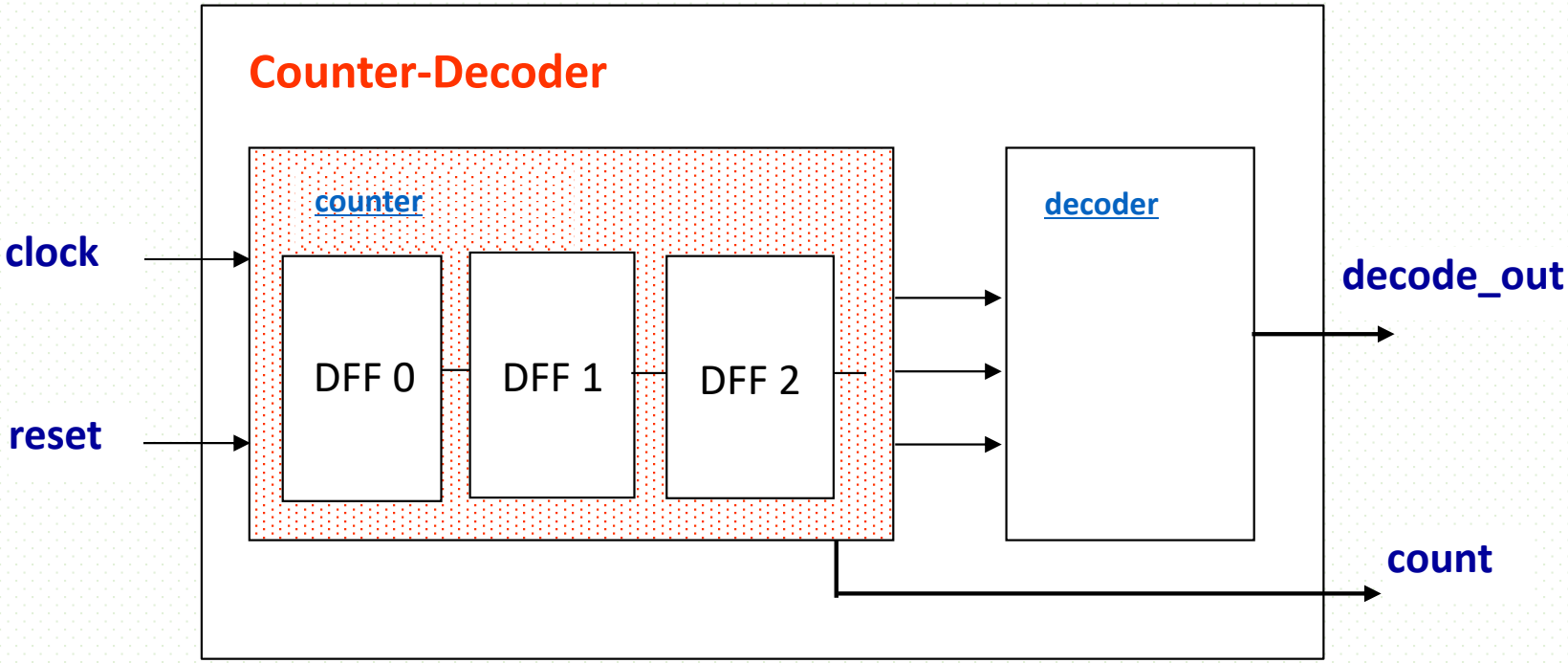
Module Hierarchy

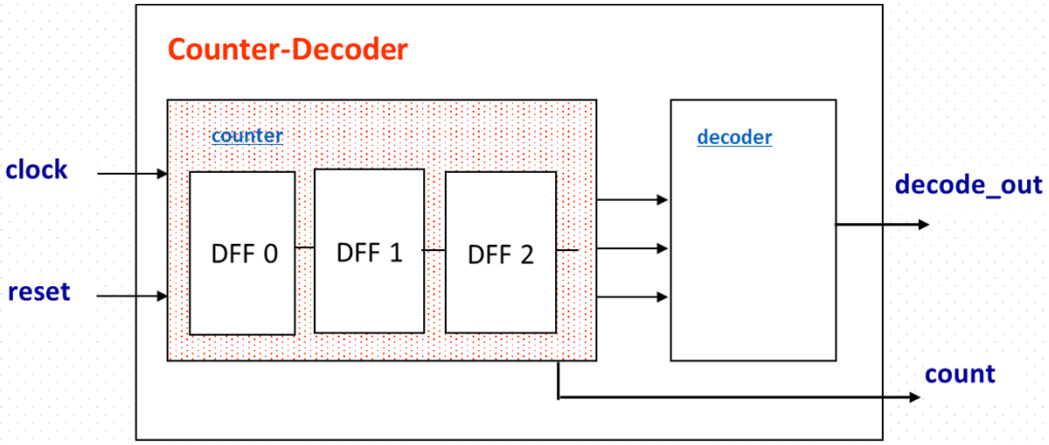
- Figure illustrates pictorially a Counter-Decoder design. Lower hierarchy.



Module Hierarchy

- Figure illustrates pictorially a Counter-Decoder design. Lower hierarchy.



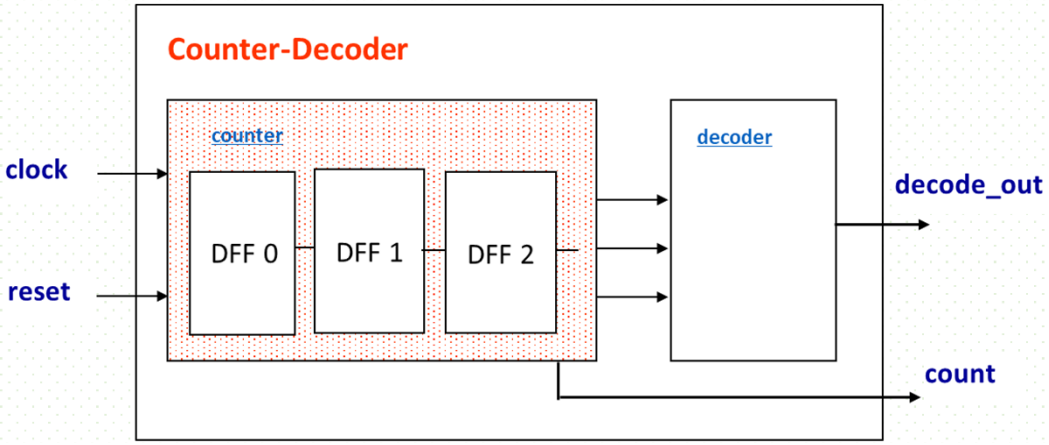


Hierarchy

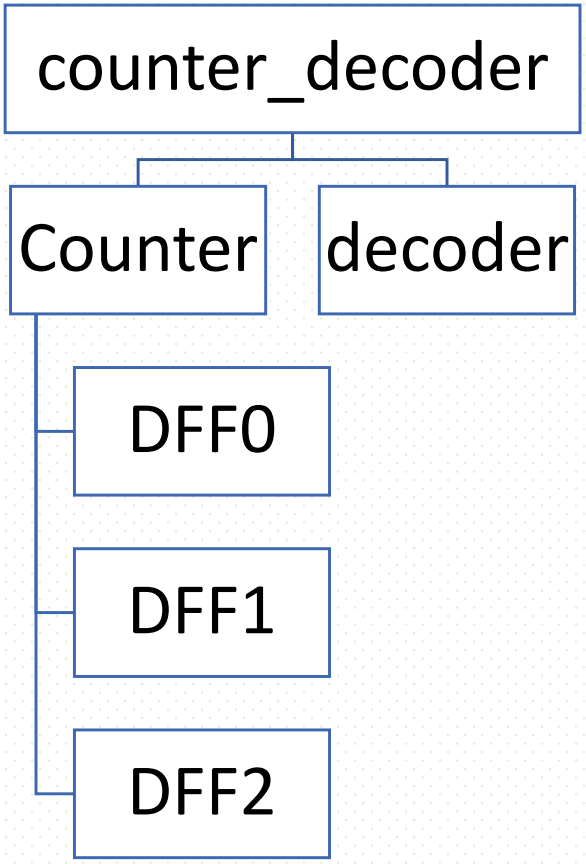
- counter_decoder
  - counter
    - DFF
  - decoder

Verilog HDL

```
module counter_decoder (clock, reset,
decode_out, count);
 input reset, clock;
 output [7:0] decode_out;
 output [2:0] count;

counter c1(q0,q1,q2,clock,count,reset);

decoder d1(q2,q1,q0,decode_out);

endmodule
```



Hierarchy



Verilog HDL

```
module counter_decoder (clock, reset,
decode_out, count);
 input reset, clock;
 output [7:0] decode_out;
 output [2:0] count;

counter c1(q0,q1,q2,clock,count,reset);

decoder d1(q2,q1,q0,decode_out);

endmodule
```

Synthesis and Optimization

- Logic synthesis is a process of converting a high-level description of the design into an optimised gate-level representation, given a standard cell library and certain design constraints.
- A standard cell library can have simple cells, such as basic logic gates like and, or and nor or macro cells, such as adders, muxes and flip-flops. A standard cell library is also known as the technology library.

Synthesis Process Flow

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

always @ (a or b or s)
        case (s)
                1'b0 : out = a ¦ b;
                1'b1 : out = a & b;
        endcase
endmodule
```

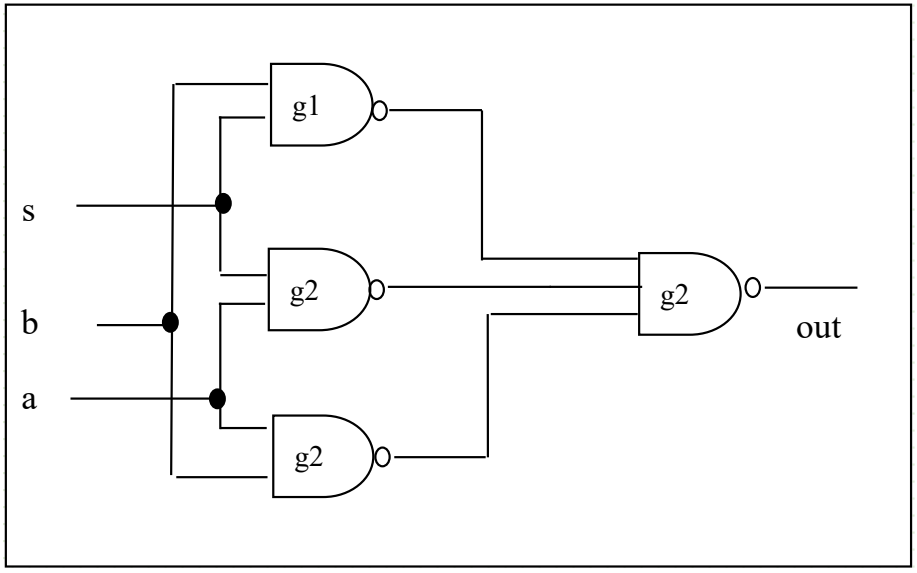
HDL Synthesis

$$out = \bar{s}(a+b) + sab$$

Technology–Independent Optimization

$$out = \bar{s}a + \bar{s}b + ab$$

Technology-specific Optimization & Mapping



Benefits of HDL Synthesis in Top-Down Design

- Designer productivity.
- Design at a higher level of abstraction.
- Improved design quality.
- Reduction of the silicon literacy requirement.
- Technology-independent design.
- Facilitates design re-use.
- One language for simulation and synthesis.

Implementation: ASIC, FPGA

- ASICs have dominated the chip market because of their large logic-gate capacity and high performance.
- FPGAs usually had fewer gates available and could not run at very high clock frequencies.

Behavioural 4 bits Adder

assign

```
// Behavioral - assign
module add4_b(co,s,ci,b,a);
 output co;
 output[3:0] s;
 input ci;
 input[3:0] b,a;

 assign {co,s} = ci + b + a;

endmodule
```

always

```
// Behavioral - always
module add4_ba(co,s,ci,b,a);
 output reg co;
 output reg [3:0] s;
 input ci;
 input[3:0] b,a;

 always @(ci or b or a)
  {co,s} = ci + b + a;

endmodule
```

Structure / RTL 4-bits Adder

1-bit Full Adder

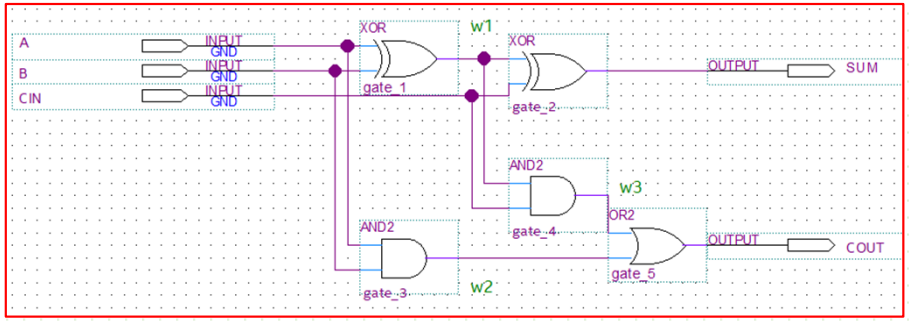
```
module fa(co,s,ci,b,a);
 output co,s;
 input ci,b,a;
 wire w3,w2,w1;

 xor g1 (w1,a,b);
 xor g2 (s,w1,ci);
 and g3 (w2,a,b);
 and g4 (w3,w1,ci);
 or  g5 (co,w2,w3);

endmodule
```

4-bits Adder

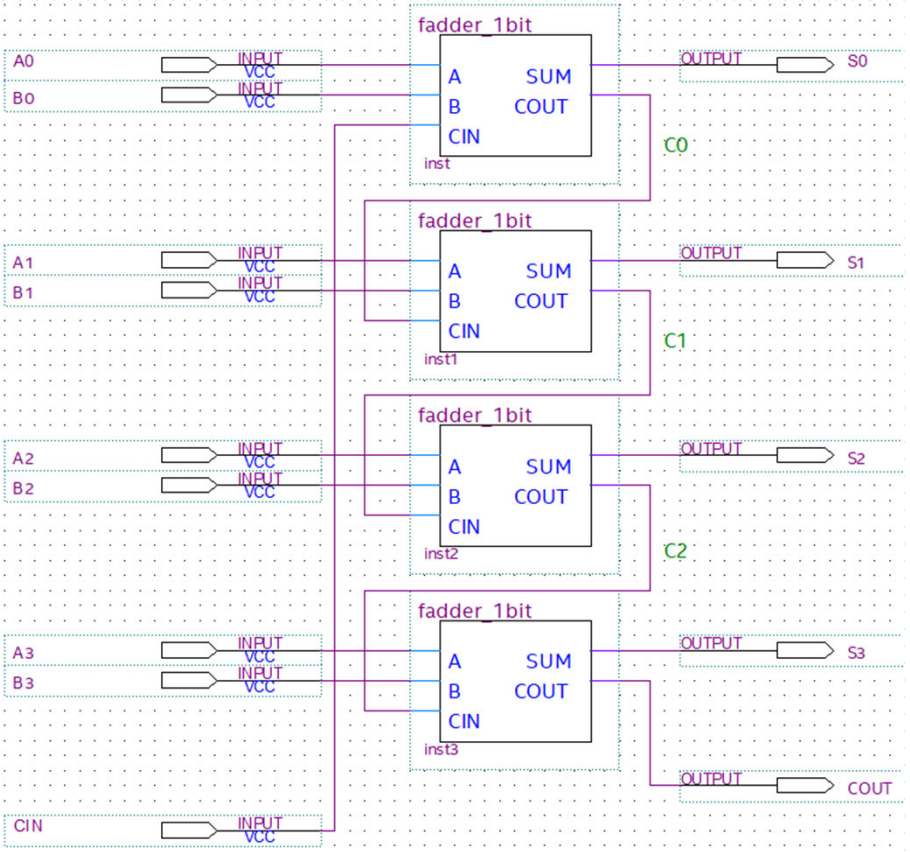
```
module add4_s(co,s,ci,b,a);
 output co;
 output [3:0] s;
 input ci;
 input [3:0] b,a;
 wire [2:0] c;

 fa i0 (c[0],s[0],ci,  b[0],a[0]);
 fa i1 (c[1],s[1],c[0],b[1],a[1]);
 fa i2 (c[2],s[2],c[1],b[2],a[2]);
 fa i3 (co,  s[3],c[2],b[3],a[3]);

endmodule
```

Structure / RTL 4-bits Adder

1-bit Full Adder

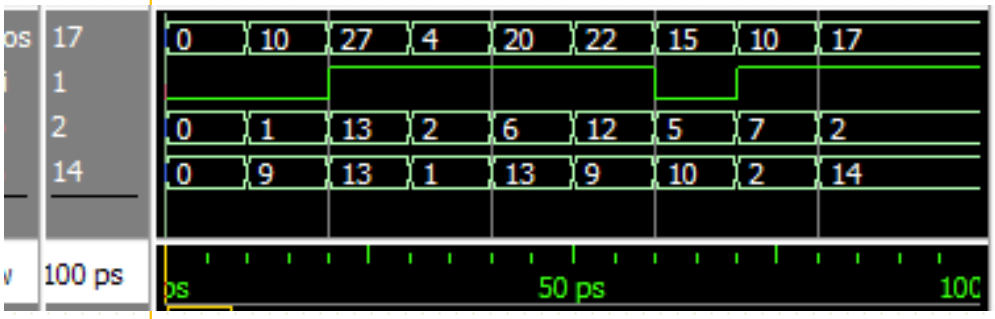


4-bits Adder



Testbench

Waveform



Same Testbench

```
module add4_b_tb;
 wire co;
 wire[3:0] s;
 reg ci;
 reg[3:0] b,a;
 integer i;

 //add4_b  i1(co,s,ci,b,a);
 //add4_ba i1(co,s,ci,b,a);
 add4_s  i1(co,s,ci,b,a);

 initial
  begin
   ci=0;b=0;a=0;
   for (i=0;i<8;i=i+1)
    begin #10 ci=$random; b=$random; a=$random; end
  end

endmodule
```

Common FAQ

How to avoid creating Latches

```
module bad_latch_2 (i_select, o_latch);
 input [1:0]  i_select;
 output [3:0] o_latch;
 reg [3:0]    o_latch;

always @ (i_select)
 begin
   if (i_select == 2'b00)
     o_latch <= 4'b0101;
   else if (i_select == 2'b01)
     o_latch <= 4'b0111;
   else if (i_select == 2'b10)
     o_latch <= 4'b1111;
   // Missing one last ELSE statement!
 end

endmodule
```

- This code above will generate a latch, because the output **o_latch** is not defined when the input **i_select** is equal to **2'b11**.
- To avoid this
  - All conditions, e.g. all combination of 2 bits 2'b(00,01,10,11)
  - Use ***default*** assignment
- This is only a problem when generating **combinational** logic. When you have registered logic (in a **sequential *always*** block in Verilog) you will never generate a latch.

Blocking - Nonblocking

- The main reason to use either Blocking or Nonblocking assignments is to generate either combinational or sequential logic.
- In ***C*** code, all assignments work one at a time.

```
1 LED_on = 0;
2 count  = count + 1;
3 LED_on = 1;
```

- The second line is only allowed to be executed once the first line is complete. This is an example of a ***blocking*** assignment.
- In Verilog HDL, there are logics that can execute concurrently and one-line-at-a-time:
  - Nonblocking – concurrently, <=
  - Blocking – serially, =

Which is correct for shift register?

Blocking

```
module s2b_b(q1,q0,d0,clk);
 output[1:0] q1,q0;
 input[1:0] d0;
 input clk;
 reg[1:0] q1,q0;

always@(posedge clk)
begin
 q0= #1 d0;
 q1= #2 q0;
end

endmodule
```

Non-Blocking

```
module s2b_n(q1,q0,d0,clk);
 output[1:0] q1,q0;
 input[1:0] d0;
 input clk;
 reg[1:0] q1,q0;

always@(posedge clk)
begin
 q0<= #1 d0;
 q1<= #2 q0;
end

endmodule
```

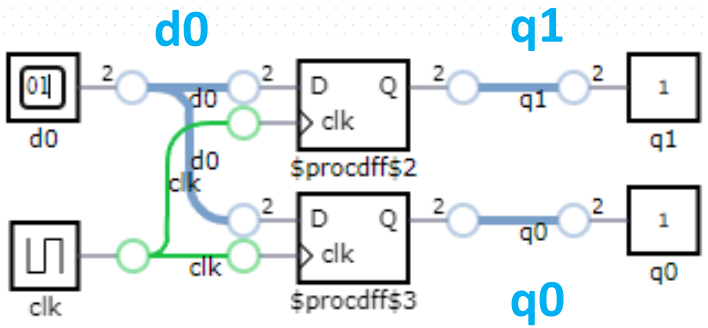
Example: Testbench

```
module s2b_tb;
 wire[1:0] q1,q0;
 reg [1:0] d0;
 reg clk;

//s2b_b i1(q1,q0,d0,clk); //blocking
s2b_n i1(q1,q0,d0,clk); //non-blocking

initial clk=0; always #10 clk=!clk;

initial
 begin
  d0=2'b00;
  #25 d0=2'b01;
  #10 d0=2'b10;
  #30 d0=2'b01;
  #10 d0=2'b10;
 end

endmodule
```

Which is correct for shift register?

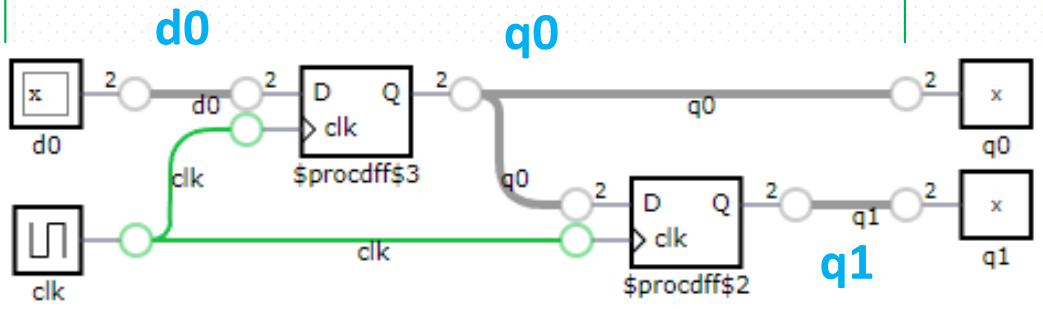
Blocking (Wrong)

```
always @(posedge clk)
begin
 q0= #1 d0;
 q1= #2 q0;
end
```



Non-Blocking

```
always @(posedge clk)
begin
 q0<= #1 d0;
 q1<= #2 q0;
end
```



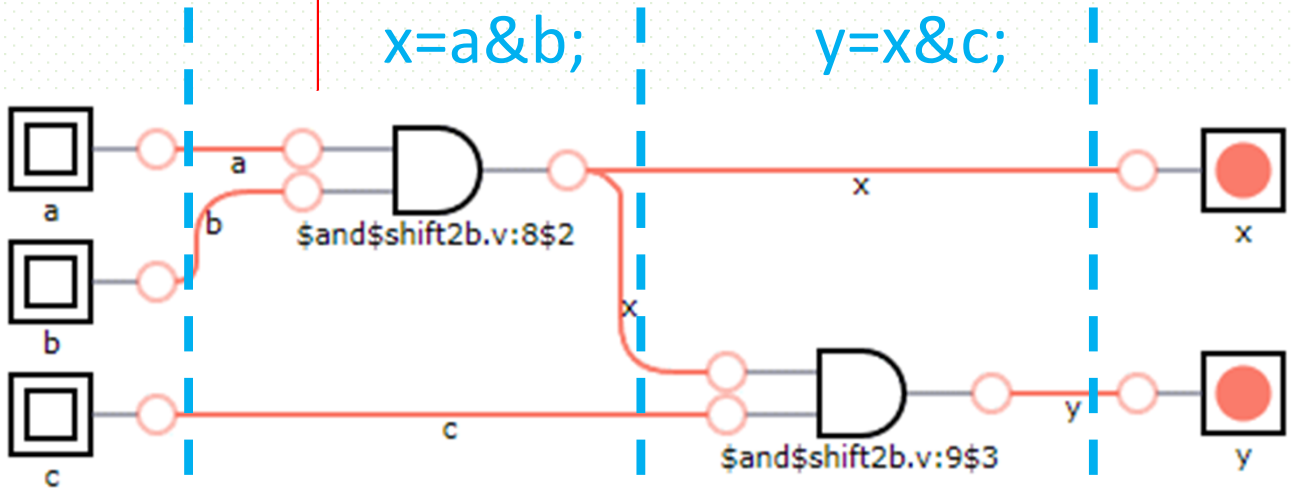
Blocking - Nonblocking

- In Verilog, if you want to create **sequential logic** use a clocked ***always*** block with ***Nonblocking*** assignments. If you want to create **combinational logic** use an ***always*** block with ***Blocking*** assignments. Try not to mix the two in the same always block.
- When talking about Blocking and Nonblocking Assignments we are referring to Assignments that are exclusively used in **Procedures** (***always***, ***initial***, task, function). You are only allowed to assign the ***reg*** data type in ***procedures***. This is different from a **Continuous** Assignment. Continuous Assignments are everything that's not a Procedure, and only allow for updating the ***wire*** data type.

Blocking – Combination Logic

```
module abc_n(x,y,a,b,c);
 output x,y;
 input a,b,c;
 reg x,y;

always @(a or b or c)
 begin
  x=a&b;
  y=x&c;
 end

endmodule
```

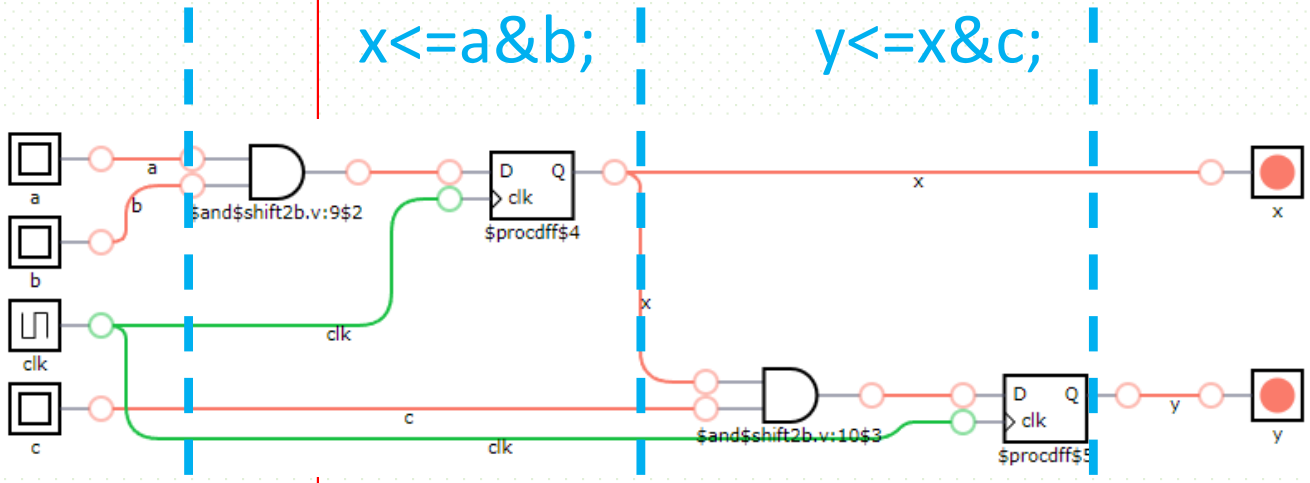
Blocking statement's order execution mimics the inherent logic flow of combinational logic.



Non-Blocking – Sequential Logic

```
module abc_n(x,y,a,b,c,clk);
 output x,y;
 input a,b,c;
 input clk;
 reg x,y;

always @(posedge clk)
 begin
  x<=a&b;
  y<=x&c;
 end

endmodule
```

Nonblocking statement's order execution mimics the inherent logic flow of sequential logic, register / DFF.



Continuous Assignments

- Continuously assigns right side of expression to left side.
- Limited to basic Boolean and ? operators. E.g. a 2:1 mux:

```
assign D=(A==1)?B:C; //if A then D=B else D=C;

assign D=(B&A)|(C&~A); //if A then D=B else D=C;
```

Procedure Assignments

- Executes a procedure allowing for more detail functions such as ***if-then-else, case*** , ***always*** statement. E.g. a 2:1 mux:

```
if (A) then D=B else D=C;


case(A)
 1'b1:D=B;
 1'b0:D=C;
endcase
```

Procedural Assignments

always

```
module mux2_1(out,a,b,sel);
 output out;
 input a,b,sel;
 reg out;

always@(a or b or sel)
begin
 if (sel) out=a;
 else out=b;
end

endmodule
```

All data types in always blocks must be declared as a ***reg*** type.

This is required even if the data type is for combinational logic.

Blocking statement is used here for combination logic.