ITU VLSI LABS

Very Large Scale Integration II - VLSI II

Verilog HDL

Serdar Duran

ITU VLSI Laboratories

Istanbul Technical University

INTRODUCTION

Introduction

- **Hardware Description Languages (HDLs)** are used to describe digital logic circuits without being tied to a specific electronic technology.

- **HDLs** uses **Register-transfer level** (**RTL**) abstraction model.
  - **Register-transfer level** (**RTL**) means the flow of digital signals (data) between hardware registers and logical operators.
  - RTL is a high-level represantation of a digital circuit.
  - RTL does not consider the physical hardware (real hardware).

- The details of gates and their interconnections **(Gate-Level Netlist)** are extracted by logic synthesis tools (like Vivado or Genus) from the RTL description.

Behavioral Algorithm → Simulate → Test Results

- Coding HDL, C - C++ or MATLAB.
- Describing the behavior of the circuit
- Textual represantation.

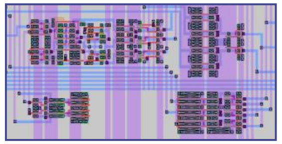
Register Transfer Level → Simulate → Test Results

- High-level represantation of the circuit.
- Describing registers and combinatorial logic.
- Schematical represantation at high-level

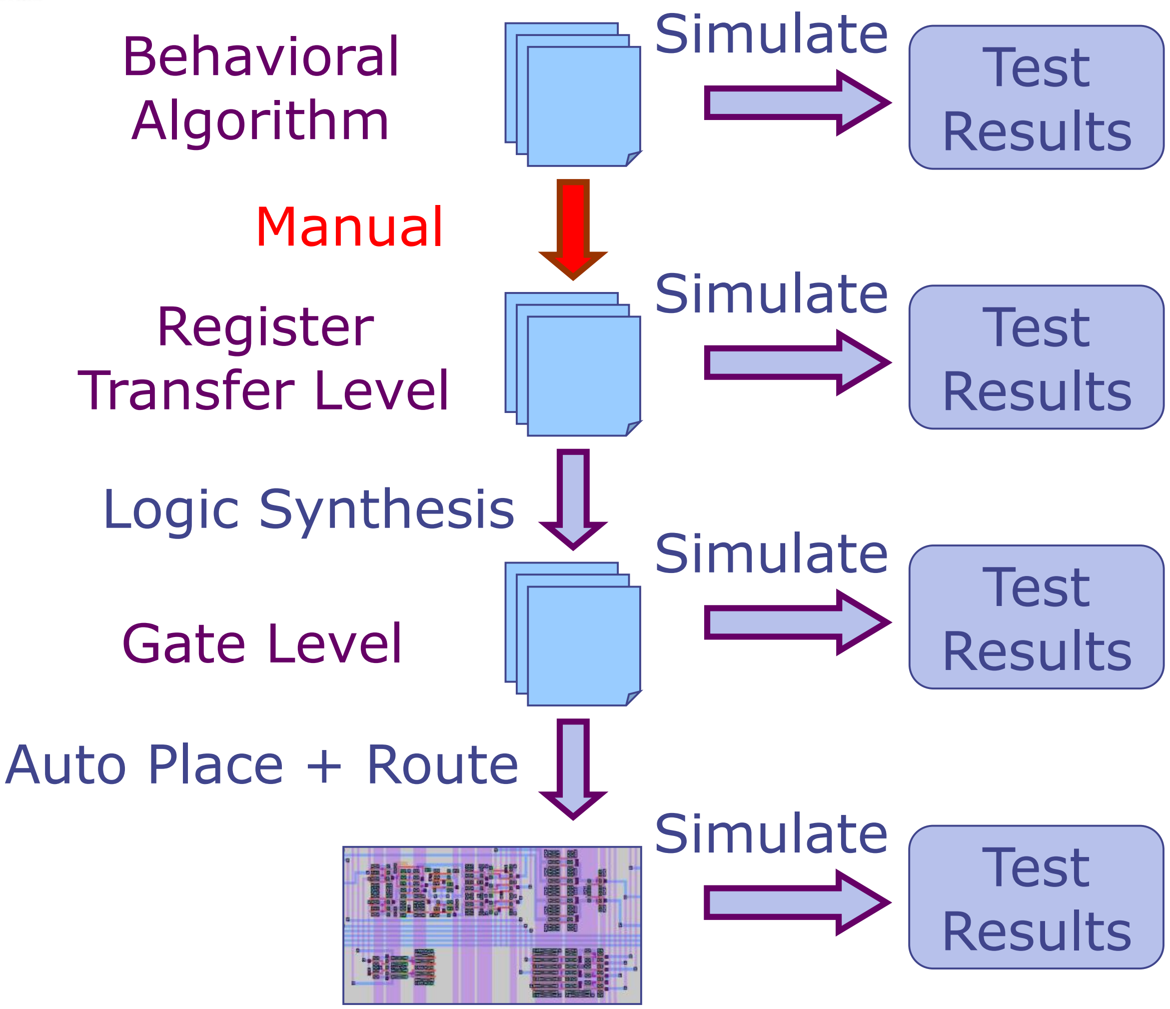
Gate Level Netlist → Simulate → Test Results

- Generating gates and their interconnections using synthesis tools.
- Schematical represantation

Place + Route



- Generate the layout (physical chip/circuit) by using automatic tools.
- Physical represantation

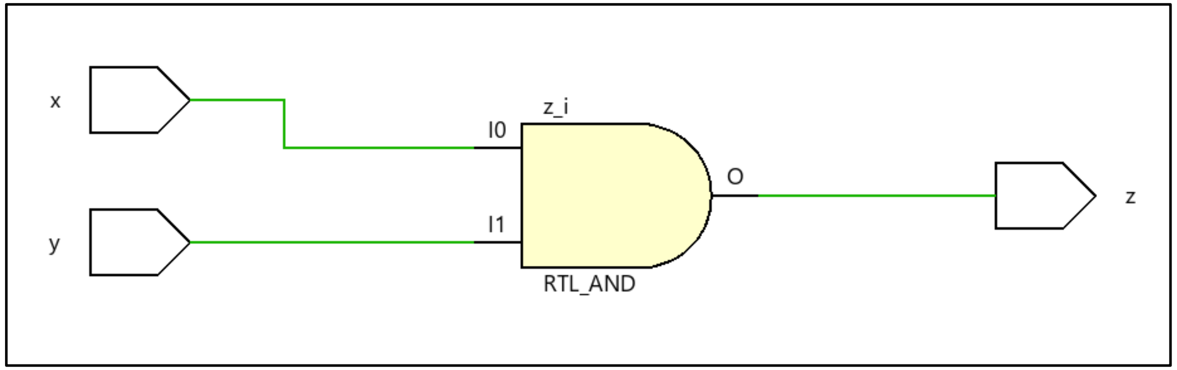


- ❖ HDL tools are used for automatic translation.
- ❖ In each step we need to simulate and verify our design.

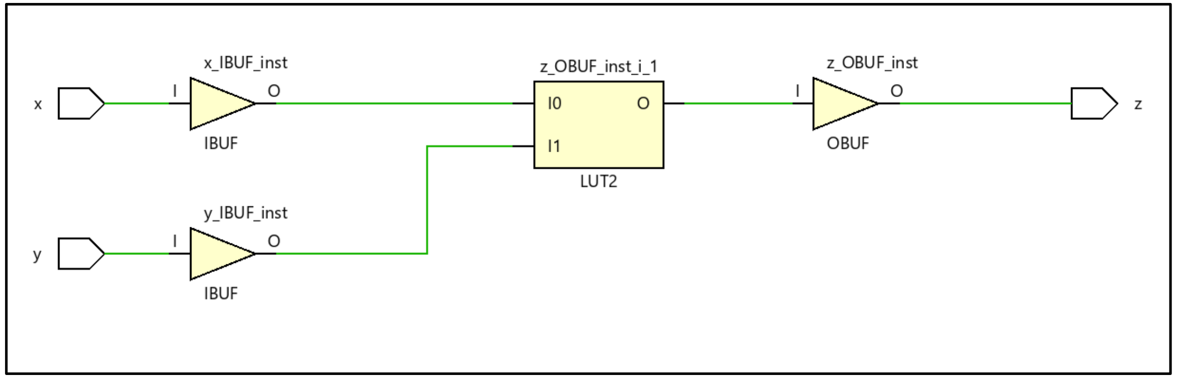
Verilog Code

```
module and2( output z, input x, input
y);

    assign z = x&y;

endmodule
```

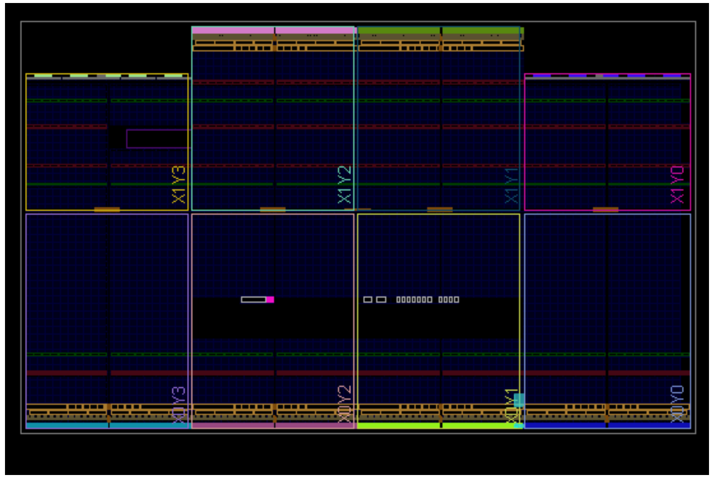
RTL Description



Gate-Level Netlist



Layout



Taken from Vivado, Xilinx Nexsys 4 DDR FPGA Chip

HDLs

- The most commonly used HDLs are **Verilog HDL** and **VHDL**.
  - **VHDL** is used more common in **FPGA Designs.**
  - **Verilog** is used more common in **ASIC Designs.**

- For simulation and Verification:
  - VHDL
  - SystemVerilog
  - Verilog

Design Tools

- **Intel Altera FPGAs:** Modelsim
- **Xilinx FPGAs:** Vivado

- **For Asic Designs:** Cadence Virtuoso
  - XCelium
  - Genus
  - Innovus

Key Points

- Verilog is **not** a programming language.
- Verilog describes the behavior of digital circuits.
- Verilog code is inherently **concurrent** contrary to regular programming languages, which are sequential (C, C++, Python).

Concurrency

- Due to the physical limitations of the transistors size, the semiconductor and microprocessor technologies is not developing fast compared to the past decades.
- Therefore, there is an increasing focus on **parallelization** and **concurrency** for the real time systems including communications, radar systems, video processing, avionic systems etc.

- **Concurrency** means performing multiple operations at the same time.

Concurrency

- Concurrency means **lower latency**.
- There are several options to deal with the latency:
  - Multi Thread Computing
  - FPGAs
  - ASICs
  - SoCs
  - Heterogeneous hardware (containing co-processors, FPGAs apart from the microprocessors and peripherals in a board)

Basics of VERILOG

Value Set

- Verilog supports four different values.

Value Level	Condition in Hardware Circuits
0	Logic zero, false condition
1	Logic one, true condition
x	Unknown logic value
z	High impedance, floating state

Modules & Ports

- A **module** is the basic building block in Verilog and it implements a certain logic behavior.
- A Verilog module has a name and a **port list**.
- **Ports** provide the interface by which a module can communicate with other modules.

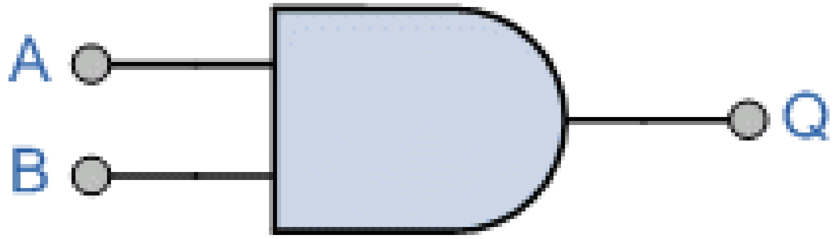
Verilog Keyword	Type of Port
input	Input port
output	Output port
inout	Bidirectional port

```
// Syntax:
module module_name ( <port_list> );
.
<logic behavior>
.
endmodule
```

Continuous Assignments

- It is used to drive a value onto a wire.
- Continuous assignments are always **active**.
- The assignment expression is evaluated as soon as the right-hand-side operands changes!

```
// Continuous assign. out is a net. i1 and i2 are nets.
assign out = i1 & i2;
```



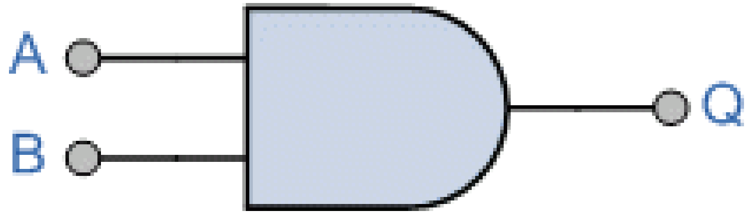
ANSI C Style PORT Declaration:

```
module AND( output Q,
            input A ,
            input B );

assign Q = A & B;


endmodule
```

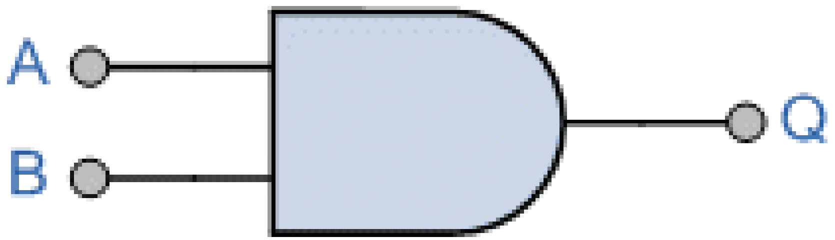
```
module AND( Q, A, B );

output Q;
input A, B;

assign Q = A & B;

endmodule
```

Wires/Nets

- Nets represent connections between hardware elements. Just as in real circuits, nets have values continuously driven on them.
- Ports are wires by default.




```
wire A;          // Declare net a for the above circuit
wire B;          // Declare two wires b,c for the above circuit
wire Q = 0;      // Net d is set to logic value 0 at declaration.
```



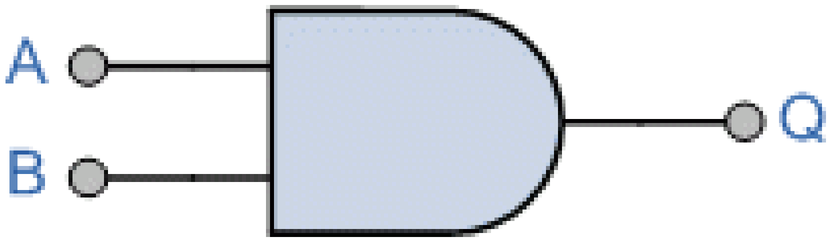
Both codes are same!

```
module AND( output wire Q,
            input wire A ,
            input wire B );

assign Q = A & B;


endmodule
```

```
module AND( output Q,
            input A ,
            input B );


assign Q = A & B;


endmodule
```

Registers

- Registers represent data storage elements. Registers retain value until another value is placed onto them.
- Registers are mostly used to describe the **sequential circuits** in Verilog.
- Do not confuse the term registers in Verilog with hardware registers in real circuits.
- The term register merely means a variable that can hold a value.




```
module AND( output reg Q,
            input A ,
            input B );
.
<functionality>
.

endmodule
```

continous assignment cannot be used for the output Q.
Because Q is not a **wire!**

To assign a value to register Q,
you need to use procedural assignment.

Structured Procedures

- Verilog is a concurrent language unlike the C programming language.
- In other words, Verilog run in parallel rather than in sequence.

```
// Runs in parallel
module test( output Q,
             output Qbar,
             input A,
             input B );

assign Q = A & B;

assign Qbar = ~( A & B);

endmodule
```

```
// Runs in sequence
int main()
{
    bool A, B, Q, Qbar;

    Q = A %% B;
    Qbar = !(A %% B);

    return 0;
}
```

Structured Procedures

- Using **always** and **initial** statement, **sequential** blocks can be created.
- The statements inside an always or initial block is executed **sequentially**!
- Multiple behavioral statements must be grouped, typically using the keywords **begin** and **end** (similar to {} in C).

```
module AND( output reg Q, input A , input B );

initial
begin
.
<initialization>
.
end

always @( ..signal_list.. )
begin
.
<functionality>
.
end
endmodule
```

Initial Block

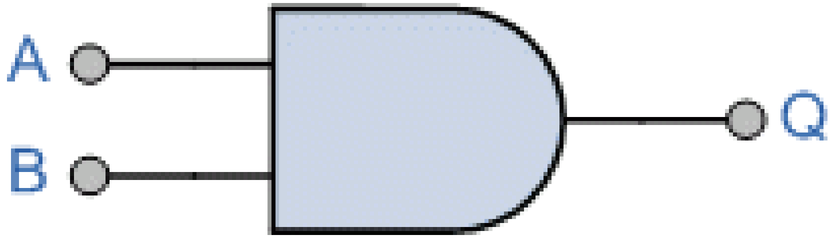
- An **initial** block executes **exactly once.**
- The initial blocks are typically used for **initialization.**
- Inizialized values are not synthesizable by logic tools!

```
module AND( output reg Q,
            input A ,
            input B );

initial
   Q = 0; // single statement no need
          // for begin and end


always @( ..signal_list.. )
begin
.
<functionality>
.
end
endmodule
```

Always Block

- An **always** block executes the statements continuously in a looping fashion.
- The @ **symbol** is used to specify an event control. Statements can be executed when the signal value is changed.

```
module AND( output reg Q,
            input A ,
            input B );
initial
   Q = 0;

// whenever the A or B values are changed
// Q value will be updated
always @(A, B)
   Q = A & B;

endmodule
```




```
module AND( output reg Q,
            input A ,
            input B );
initial
   Q = 0;

// whenever the A or B values are changed
// Q value will be updated
always @(A, B)
   Q = A & B;

endmodule
```

Behavioral Statements

- All behavioral statements should be inside an **initial** or **always** block.
- Behavioral statements are:
  - Blocking and Nonblocking assignments
  - If-Else conditional statements
  - Case statement
  - For, While loops
- If there are multiple initial or always blocks, each block starts to execute **concurrently!**

If – Else Statement

```
//Type 1 conditional statement. No else statement.
if (<expression>)
    true_statement ;

//Type 2 conditional statement. One else statement
if (<expression>)
    true_statement ;
else
    false_statement ;

//Type 3 conditional statement. Nested if-else-if.
if (<expression1>)
    true_statement1 ;
else if (<expression2>)
    true_statement2 ;
else if (<expression3>)
    true_statement3 ;
else
    default_statement ;
```

Case Statement

```
case (<expression>)

    <alternative1>: statement1;
    <alternative2>: statement2;
    <alternative3>: statement3;
    ...
    ...
    default: default_statement; // optional

endcase
```

- You must combine multiple assigments using **begin** - **end** *keywords*.

For Loop

```
for ( <initialization> ; <condition> ; <step_assignment> )
begin
    .
    statements
    .
end
```

While Loop

```
while ( <condition> )
begin
   .
   statements
   .
end
```

Initialization

- Variables (registers, integers etc.) can be initialized when they are declared.

```
module AND( output reg Q = 0,
            input A ,
            input B );

// whenever the A or B values are changed
// Q value will be updated
always @(A, B)
   Q = A & B;

endmodule
```

@* Operator

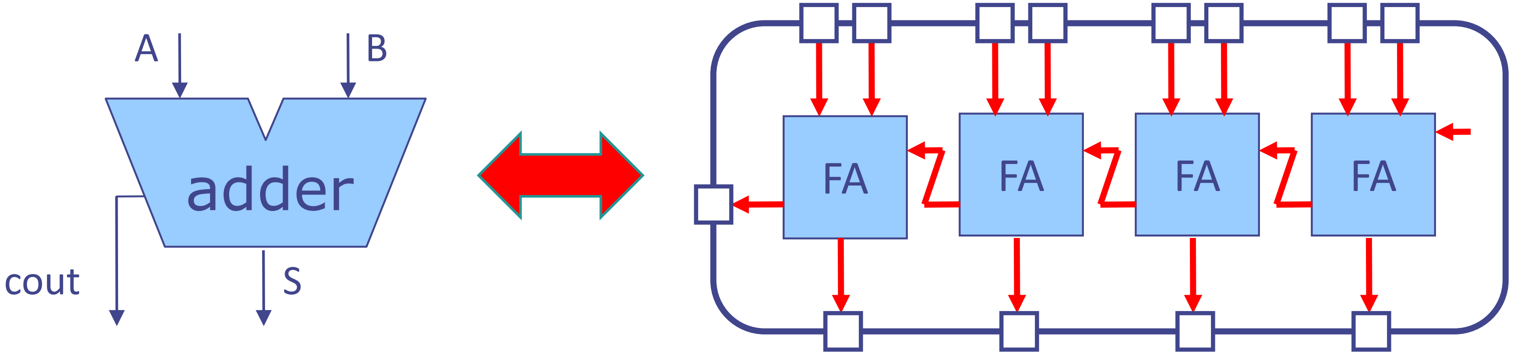
- Two special symbols: **@* and @(*)** are sensitive to a change on any signal inside the block.

```
module AND( output reg Q,
            input A ,
            input B );

// whenever the A or B values are changed
// Q value will be updated
always @(*)
   Q = A & B;

endmodule
```

Modeling & Instantiation

Modules

- The module is the basic unit of a **hierarchy** in Verilog.
- It can be a single element or a **collection** of lower level modules.



Top Module

Lower level modules inside the Top Module

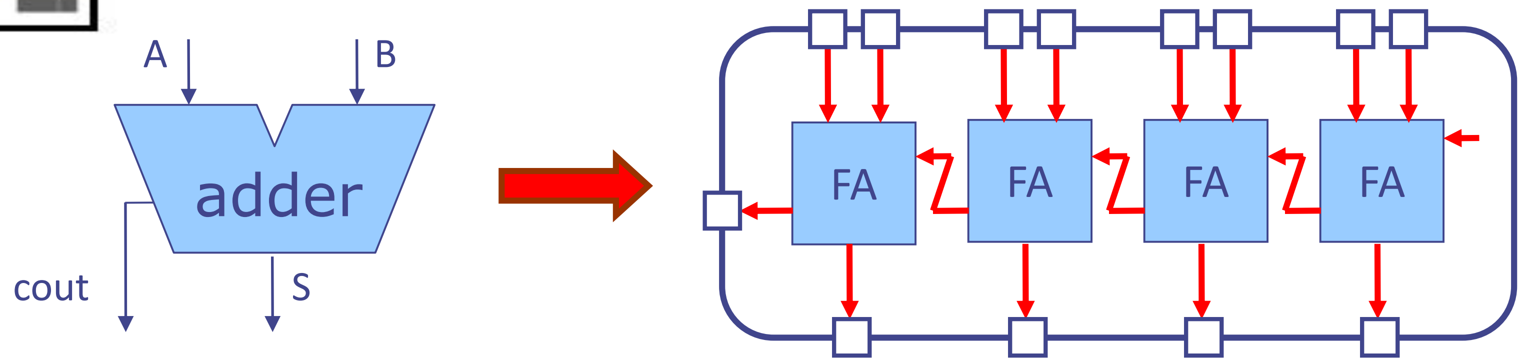
Instantiation

- Creating objects from a module is called **instantiation**, and the objects are called **instances**.

```
module <upper_module_name> (<module_terminals>)
.
.
<module_name> <instance_name> (<terminals>)
.
.
<module_internals>
.
.
endmodule
```

Instantiation

- The order of the statements (incuding instantiations) are not important.
- Remember that the code is not executed in sequence!

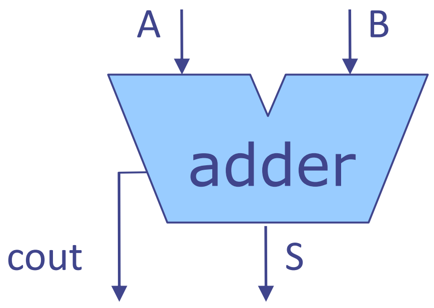



```
module adder( input [3:0] A, B,
              output cout,
              output [3:0] S );

wire c0 = 0;
wire c1, c2, c3;

FA fa0( A[0], B[0], c0, c1, S[0] );
FA fa1( A[1], B[1], c1, c2, S[1] );
FA fa2( A[2], B[2], c2, c3, S[2] );
FA fa3( A[3], B[3], c3, cout, S[3] );

endmodule
```

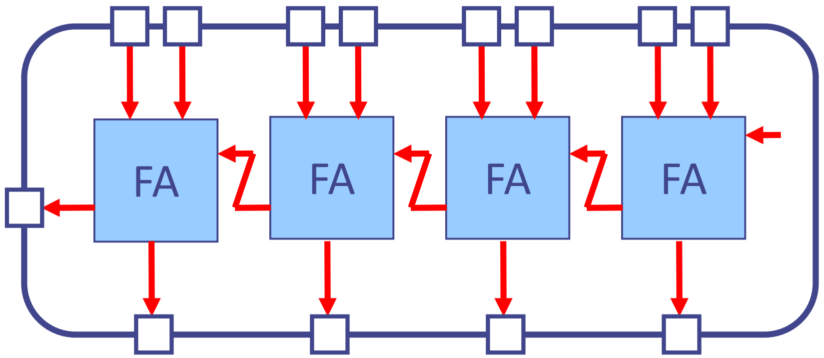
Vectors (data type)

Instantiations

- Full adder is an another module and must be defined in the **same project**.



Top Module



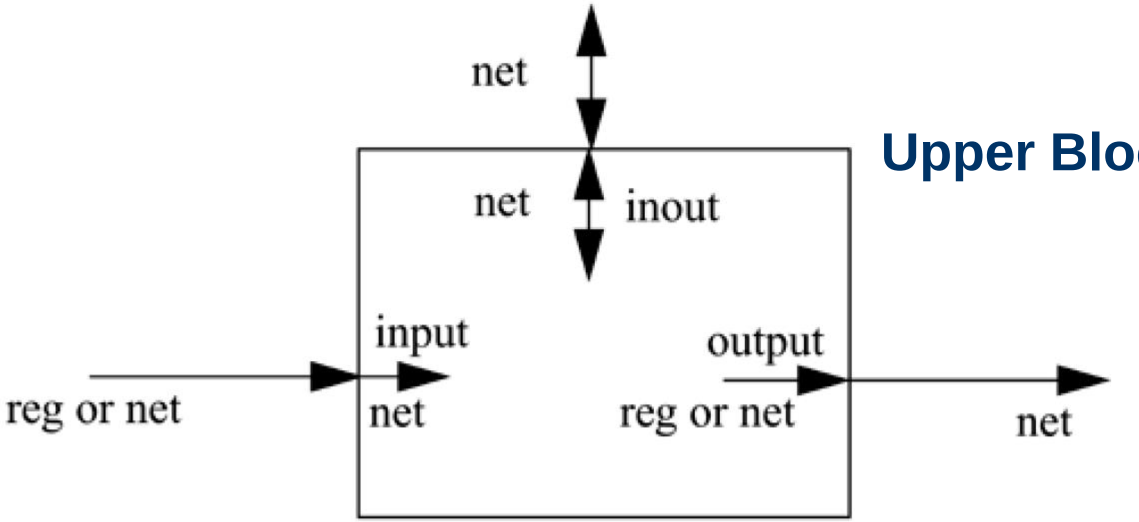
Lower Module

```
module adder( input [3:0] A, B,
              output cout,
              output [3:0] S );
wire c0 = 0;
wire c1, c2, c3;

FA fa0( A[0], B[0], c0, c1, S[0] );
FA fa1( A[1], B[1], c1, c2, S[1] );
FA fa2( A[2], B[2], c2, c3, S[2] );
FA fa3( A[3], B[3], c3, cout, S[3] );

endmodule
```

```
module FA( input a, b, c_in,
           output c_out, sum );

assign sum = a ^ b ^ c_in;
assign c_out = ( c_in & ( a ^ b ) ) | ( a & b );

endmodule;
```

ITU VLSI LABS

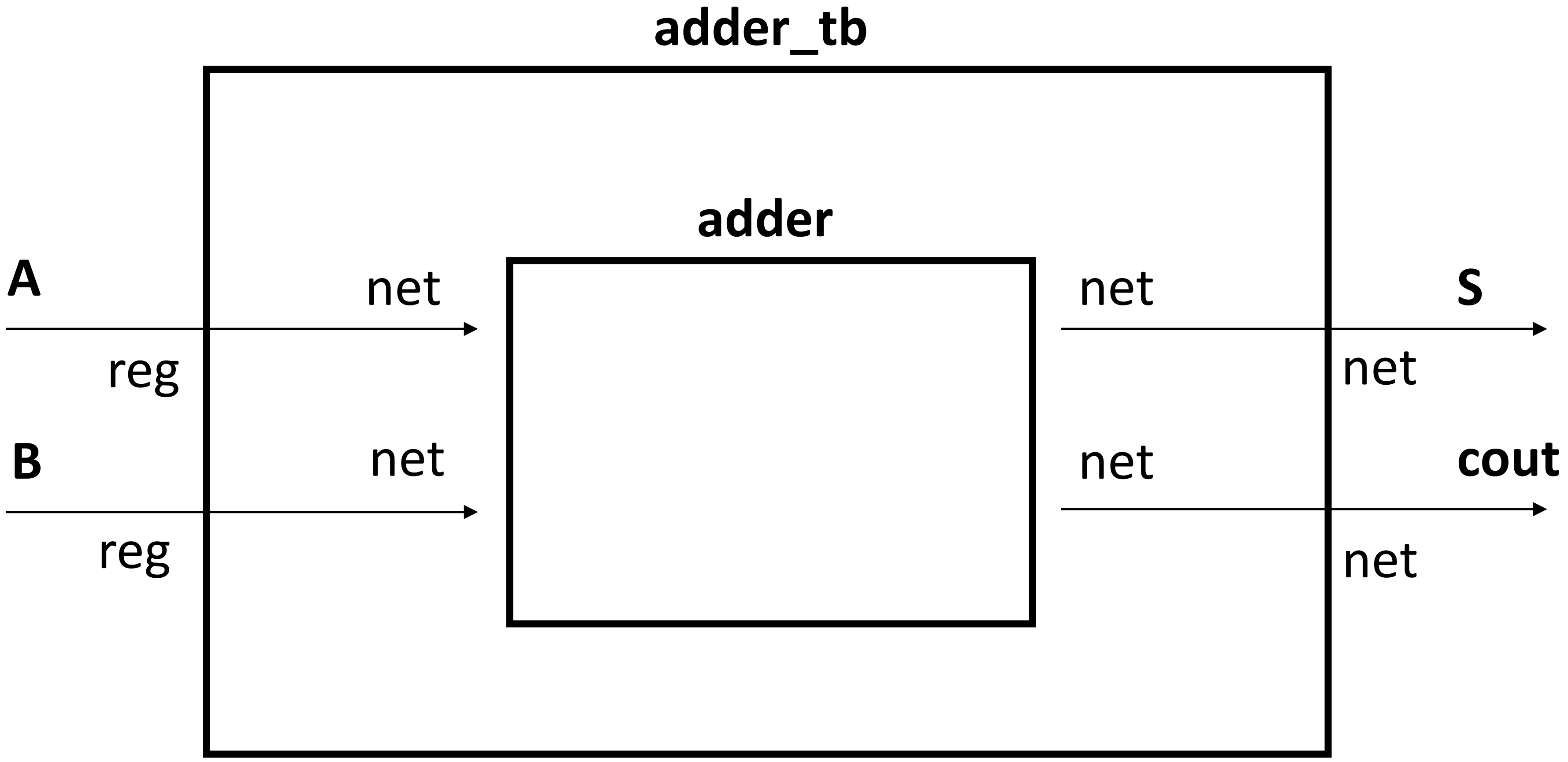
Port Connection Rules



- **Inputs:** internally must always be of type net, externally the inputs can be connected to a variable of type reg or net.
- **Outputs:** internally can be of type net or reg, externally the outputs must be connected to a variable of type net.
- **Inouts:** internally or externally must always be type net, can only be connected to a variable net type

```verilog
module adder_tb;

reg [3:0] A, B; // external inputs
wire [3:0] SUM; // external output
wire COUT;      // external output

// instance of adder
adder AD( A, B, COUT, SUM );

initial
    $monitor( $time, " A=%d  B=%d |
COUT=%d  SUM=%d ",A,B,COUT,SUM );

initial
begin
    A=0; B=0;
    #10 A=4'b0001; B=4'b1001;
    #10 A=4'b0101; B=4'b1011;
    #10 A=4'b0101; B=4'b1101;
    #10 $finish;
end
```

```verilog
module adder( input [3:0] A, B, // internal inputs
              output cout,      // internal output
              output [3:0] S ); // internal output

wire c0 = 0;
wire c1, c2, c3;

FA fa0( A[0], B[0], c0, c1, S[0] );
FA fa1( A[1], B[1], c1, c2, S[1] );
FA fa2( A[2], B[2], c2, c3, S[2] );
FA fa3( A[3], B[3], c3, cout, S[3] );

endmodule
```


adder_tb
adder
A
net
reg
B
net
reg
net
S
net
net
cout
net

Connecting by Ordered List

```
module adder( input [3:0] A, B,
              output cout,
              output [3:0] S );

wire c0 = 0;
wire c1, c2, c3;

FA fa0( A[0], B[0], c0, c1, S[0] );
FA fa1( A[1], B[1], c1, c2, S[1] );
FA fa2( A[2], B[2], c2, c3, S[2] );
FA fa3( A[3], B[3], c3, cout, S[3] );

endmodule
```

```
module FA( input a, b, c_in,
           output c_out, sum );
```

Order: a , b , c_in , c_out , sum

Connecting by Port's Name

```
module adder( input [3:0] A, B,
              output cout,
              output [3:0] S );

wire c0 = 0;
wire c1, c2, c3;

FA fa0( .c_out(c1), .sum(S[0]), .a(A[0]), .b(B[0]), .c_in(c0) );
FA fa1( .c_out(c2), .sum(S[1]), .a(A[1]), .b(B[1]), .c_in(c1) );
FA fa2( .c_out(c3), .sum(S[2]), .a(A[2]), .b(B[2]), .c_in(c2) );
FA fa3( .c_out(cout), .sum(S[3]), .a(A[3]), .b(B[3]), .c_in(c3) );

endmodule
```

```
module FA( input a, b, c_in,
           output c_out, sum );
```

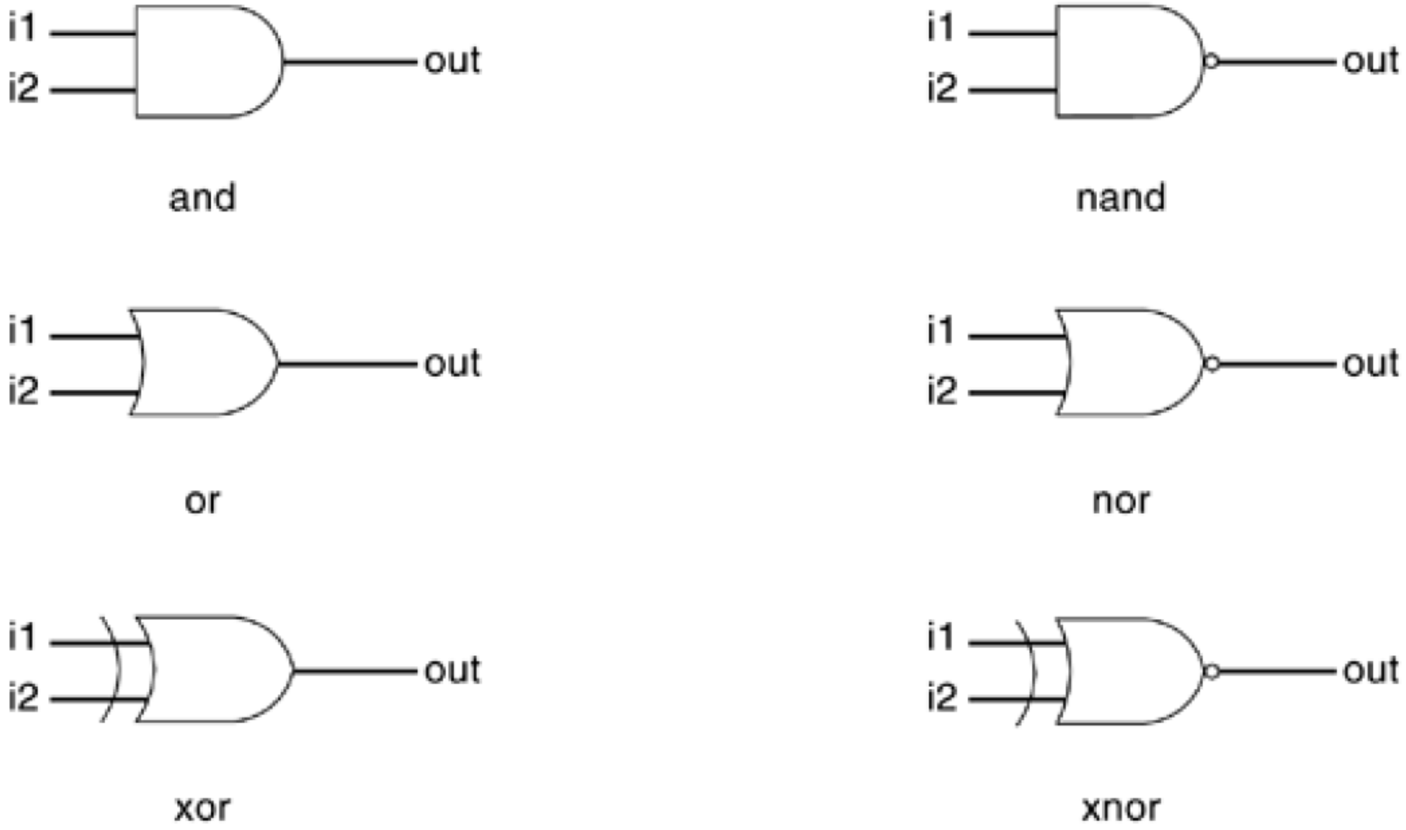
Modeling Concepts

- There are three different modelling concepts:
  - Structural Modelling
  - Dataflow Modelling
  - Behavioral Modelling

Structural Modelling

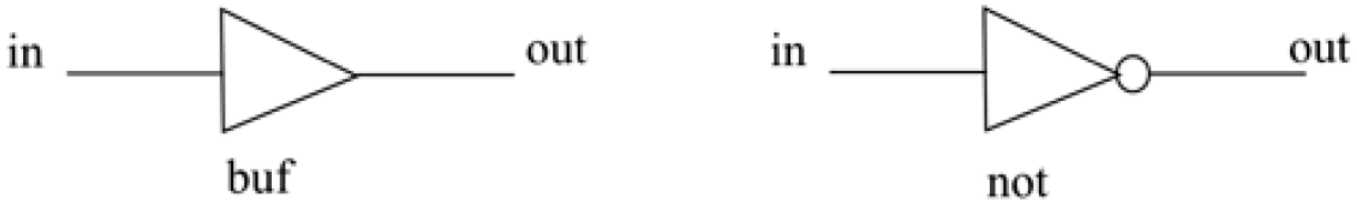
- The structural modelling is the lowest level of abstraction obtained using **logic gates**.
- It can be considered as a textual representation of logic circuit diagrams.
- **Primitive gates** are useful for structural modelling.

Primitives


i1
i2
out
and
i1
i2
out
nand
i1
i2
out
or
i1
i2
out
nor
i1
i2
out
xor
i1
i2
out
xnor

```
wire OUT, IN1, IN2;

// basic gate instantiations.
and a1(OUT, IN1, IN2);
nand na1(OUT, IN1, IN2);

or or1(OUT, IN1, IN2);
nor nor1(OUT, IN1, IN2);

xor x1(OUT, IN1, IN2);
xnor nx1(OUT, IN1, IN2);

// More than two inputs: 3 input nand gate
nand na1_3inp(OUT, IN1, IN2, IN3);

// gate instantiation without instance name
and (OUT, IN1, IN2);
```

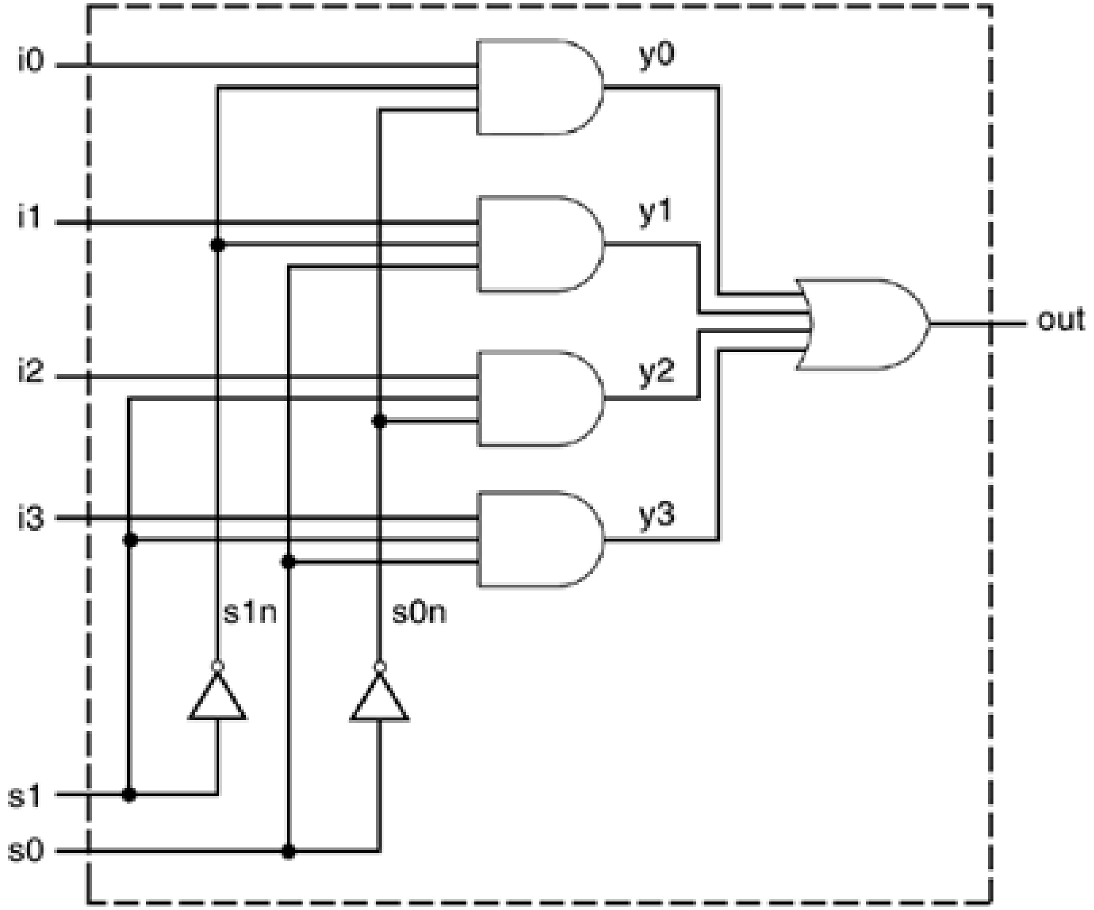
- Instance name does not need to be specified for primitives.
- More than two **inputs** can be specified for these gates.




```
 // basic gate instantiations.
buf b1(OUT1, IN);
not n1(OUT1, IN);

// More than one outputs
buf b1_2out(OUT1, OUT2, IN);

// gate instantiation without instance name
not (OUT1, IN);
```

- More than two **outputs** can be specified for these gates.

4-bit MULTIPLEXER



Structural Design

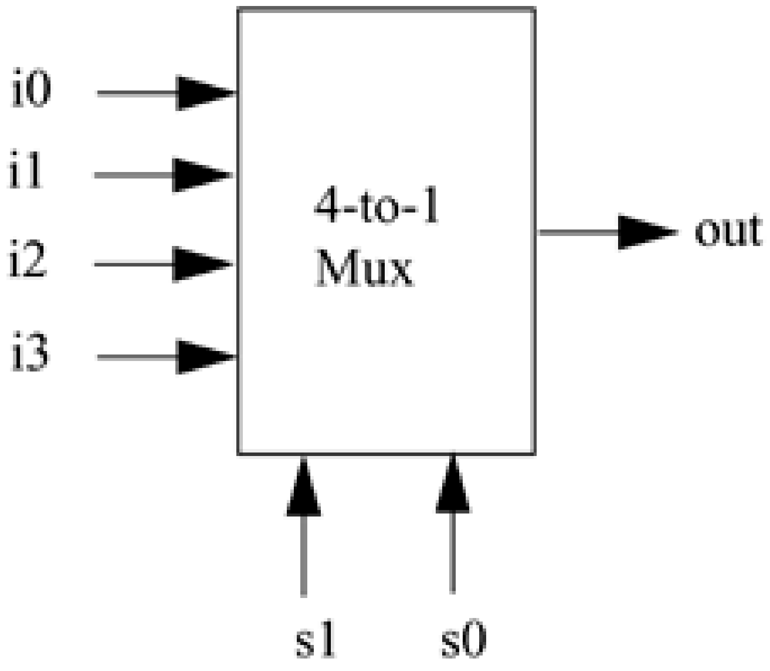
```
module mux4( out, i0, i1, i2, i3, s1,
s0);

output out;
input i0, i1, i2, i3;
input s0, s1;
wire s1n, s0n;
wire y0, y1, y2, y3;

not ( s1n, s1 );
not ( s0n, s0 );
and (y0, i0, s1n, s0n);
and (y1, i1, s1n, s0);
and (y2, i2, s1, s0n);
and (y3, i3, s1, s0);
or (out, y0, y1, y2, y3);

endmodule
```

Dataflow Modelling

- Dataflow modeling style is mainly used to describe combinational circuits.
- The basic mechanism is **the continuous assignment (*keyword* assign)**.
- **Reduction operators** are used to implement the combinational circuits.

Reduction	&	reduction and
	~&	reduction nand
	|	reduction or
	~|	reduction nor
	^	reduction xor
	^~ or ~^	reduction xnor

4-bit MULTIPLEXER



Dataflow Design

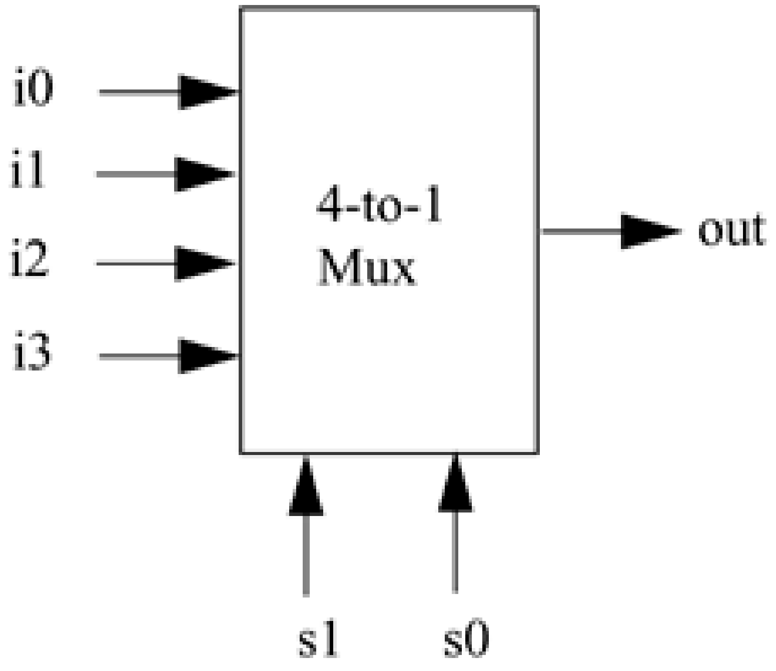
```
module mux4_to_1( out, i0, i1, i2, i3, s1, s0 );

output out;
input i0, i1, i2, i3;
input s1, s0;

assign out = s1 ? ( s0 ? i3 : i2 ) : ( s0 ? i1 : i0 );

endmodule
```

```
module mux4_to_1( out, i0, i1, i2, i3, s1, s0 );

output out;
input i0, i1, i2, i3;
input s1, s0;

assign out = ( ~s1 & ~s0 & i0 ) | ( ~s1 & s0 & i1 )
           | ( s1 & ~s0 & i2 ) | ( s1 & s0 & i3 );

endmodule
```

Behavioral Modelling

- Behavioral modeling is used to describe complex circuits.
- It is primarily used to model sequential circuits, but can also be used to model pure combinatorial circuits.
- The mechanisms (statements) for modeling the behavior of a design are:
  - **initial** statements
  - **always** statements
- A module may contain an arbitrary number of **initial** or **always** statements

4-bit MULTIPLEXER



Behavioral Design

```
module mux4( out, i0, i1, i2, i3, s1, s0 );

output out;
input i0, i1, i2, i3;
input s0, s1;
reg out;


always @( s1, s0, i0, i1, i2, i3 )

case ( {s1, s0} )
    2'd0  :  out = i0;
    2'd1  :  out = i1;
    2'd2  :  out = i2;
    2'd3  :  out = i3;
    default : $display("Invalid control signals");
endcase
endmodule
```

Data Types & Numbers

Number Representations

```
<size> '<base format> <number>
```

Number

Base format
(d, b, o, h)

Decimal number
representing size in bits

ITU VLSI LABS

Sized Numbers

```
<size> '<base format> <number>
```

- <size> is written only in decimal and specifies the number of bits in the number.
- Base formats are decimal ('d or 'D), hexadecimal ('h or 'H), binary ('b or 'B) and octal ('o or 'O).

```
4'b1111   // This is a 4-bit binary number
12'habc   // This is a 12-bit hexadecimal number
16'd255   // This is a 16-bit decimal number.
```

Unsized Numbers

- Numbers that are written without a <size> specification have a default number of bits that is simulator- and machine-specific (must be at least 32).
- Numbers that are specified without a <base format> specification are decimal numbers by default.

```
23456 // This is a 32-bit decimal number by default
'hc3  // This is a 32-bit hexadecimal number
'o21  // This is a 32-bit octal number
```

Negative Numbers

- Negative numbers can be specified by putting a minus sign before the size for a constant number.
- Size constants are always positive. They are always 2’s complement of corresponding number.

```
-6'd3   // 8-bit negative number stored as 2's complement of 3
```

X and Z Values

- Verilog has two symbols for **unknown** and **high impedance** values. An unknown value is denoted by an **X**. A high impedance value is denoted by **Z**.
- If the most significant bit of a number is 0, x, or z, the number is automatically extended to fill the most significant bits, respectively, with 0, x, or z.

```
12'h13x   // This is a 12-bit hex number; 4 least significant bits unknown
6'hx      // This is a 6-bit unknown hex number
32'bz     // This is a 32-bit high impedance number
```

Underscore Character

- An underscore character "_" is allowed anywhere in a number except the first character.
- Underscore characters are allowed only to improve readability of numbers and are ignored by Verilog.

```
12'b1111_0000_1010  // Use of underline characters for readability
```

Strings

- A string is a sequence of characters that are enclosed by double quotes.

```
"Hello Verilog World"     // is a string
"a / b"                   // is a string
```

- Blank spaces (\b) , tabs (\t) and newlines (\n) are whitespace characters.

Operators

Operator Type	Operator Symbol	Operation Performed	Number of Operands
Arithmetic	*	multiply	two
	/	divide	two
	+	add	two
	-	subtract	two
	%	modulus	two
	**	power (exponent)	two
Logical	!	logical negation	one
	&&	logical and	two
	||	logical or	two
Relational	>	greater than	two
	<	less than	two
	>=	greater than or equal	two
	<=	less than or equal	two
Equality	==	equality	two
	!=	inequality	two
	===	case equality	two
	!==	case inequality	two
Bitwise	~	bitwise negation	one
	&	bitwise and	two
	|	bitwise or	two
	^	bitwise xor	two
	^~ or ~^	bitwise xnor	two
Reduction	&	reduction and	one
	~&	reduction nand	one
	|	reduction or	one
	~|	reduction nor	one
	^	reduction xor	one
	^~ or ~^	reduction xnor	one
Shift	>>	Right shift	Two
	<<	Left shift	Two
	>>>	Arithmetic right shift	Two
	<<<	Arithmetic left shift	Two
Concatenation	{ }	Concatenation	Any number
Replication	{ { } }	Replication	Any number
Conditional	?:	Conditional	Three

Concatenation Operator

- Concatenation operator appends multiple operands.

```
// A = 1'b1, B = 2'b00, C = 2'b10, D = 3'b110
Y = {B , C}                          // Result Y is 4'b0010
Y = {A , B , C , D , 3'b001} // Result Y is 11'b10010110001
Y = {A , B[0], C[1]}                 // Result Y is 3'b101
```

Replication Operator

- Repetitive concatenation of the same number.

```
// A = 1'b1; B = 2'b00; C = 2'b10; D = 3'b110;
Y = { 4{A} } // Result Y is 4'b1111
Y = { 4{A} , 2{B} } // Result Y is 8'b11110000
Y = { 4{A} , 2{B} , C } // Result Y is 8'b1111000010
```

Conditional Operator

- It is the same as in the C language.

```
//model functionality of a 2-to-1 mux
assign out = control ? in1 : in0;
```

DATA TYPES

- **Wires (Nets)**
- **Registers**
  - Integers
  - Real Numbers

- Vectors
- Arrays
- Strings

Integers

- An integer is a general purpose data type used for manipulating quantities.
- Registers declared as data type store values as unsigned quantities, whereas integers store values as **signed** quantities.

```
integer counter;  // general purpose variable used as a counter.
initial
    counter = -1; // A negative one is stored in the counter
```

Real Numbers (Floating Point)

- They can be specified in decimal notation (e.g., 3.14) or in scientific notation (e.g., 3e6, which is 3×10^6).
- When a real value is assigned to an integer, the real number is rounded off to the nearest integer.

```
// real
real delta; // Define a real variable called delta
initial
begin
    delta = 4e10; // delta is assigned in scientific
                  // notation
    delta = 2.13; // delta is assigned a value 2.13
end

integer i; // Define an integer i
initial
    i = delta; // i gets the value 2 (rounded value
of 2.13)
end
```

Vectors

- **Wires** or **reg** data types can be declared as vectors (multiple bit widths).
- If bit width is not specified, the default is a scalar (1-bit).

```
// DECLARATION:
wire a;                        // scalar net variable, default
wire [7:0] bus;                // 8-bit bus
wire [31:0] busA,busB,busC;    // 3 buses of 32-bit width.

reg clock;                     // scalar register, default
reg [0:40] virtual_addr;       // Vector register, 41 bits wide

// ACCESSING:
busA[7]      // bit # 7 of vector busA
bus[2:0]     // Three least significant bits of vector bus,
virtual_addr[0:1]// Two most significant bits of vectorvirtual_addr
```

Arrays

- Multi-dimensional arrays can also be declared with any number of dimensions.
- Don’t confuse arrays with vectors!
- A vector is a single element that is n-bits wide. On the other hand, arrays are multiple elements that are 1-bit or n-bits wide.

```
// DECLARATION:
integer count[0:7];        // An array of 8 count variables
reg bool[31:0];            // Array of 32 one-bit boolean register variables

reg [4:0] port_id[0:7];          // Array of 8 port_ids; each port_id is 5 bits wide
integer matrix[4:0][0:255];      // Two dimensional array of integers
reg [63:0] array_4d [15:0][7:0][7:0][255:0];      //Four dimensional array
wire [7:0] w_array2 [5:0];       // Declare an array of 8 bit vector wire
wire w_array1[7:0][5:0];         // Declare an array of single bit wires
```

```
// ACCESSING:
count[5] = 0;             // Reset 5th element of array of count variables
chk_point[100] = 0;       // Reset 100th time check point value
port_id[3] = 0;           // Reset 3rd element (a 5-bit value) of port_id array.
matrix[1][0] = 33559;     // Set value of element indexed by [1][0] to

array_4d[0][0][0][0][15:0] = 0;     // Clear bits 15:0 of the register
                                    // Accessed by indices [0][0][0][0]
port_id = 0;          // Illegal syntax - Attempt to write the entire array
matrix [1] = 0;       // Illegal syntax - Attempt to write [1][0]..[1][255]
```

Blocking & Nonblocking Assignments

Blocking Assignment

- Blocking assignment statements are executed in the order they are specified in a sequential block.

```
reg x, y, z;

initial
begin
    x = 0; y = 0; z = 0;      // x = 0, y = 0, z = 0 are executed at time 0

    #15 x = 1;      // x = 1 at time = 15
    #10 y = 1;      // y = 1 at time = 25
    z = 1;          // z = 1 at time = 25 but after the statement above
end
```

Non-Blocking Assignment

- Nonblocking assignments allow **<u>scheduling</u>** of assignments without blocking execution of the statements that follow in a sequential block.
- In nonblocking assignments read and write operations are **separated**.
- Nonblocking assignments is used for concurrent data transfers in a sequential block.
- A “ **<=** “ operator is used to specify nonblocking assignments.

```
reg x, y, z;

initial
begin
    x = 0; y = 0; z = 0;

    // x = 1 is scheduled to execute after 15 units:  at time = 15
    #15 x <= 1;

    // y = 1 is scheduled after 10 time units:  at time = 10
    #10 y <= 1;

    // z = 1 is scheduled without any delay:   at time = 0
    z <= 1;
end
```

- Nonblocking Assignments:

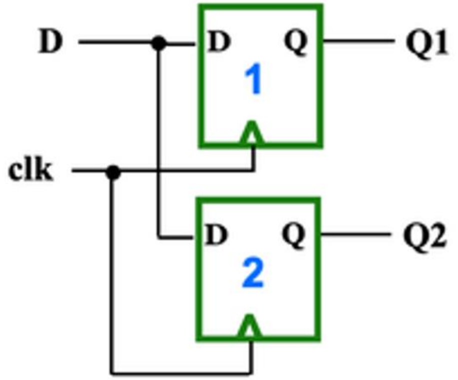
```
always @( posedge clock )
begin
    // Read the data a and b, after that
    // write them to A and B
    A <= a;
    B <= b;
end
```

- Implementing Nonblocking Assignments using Blocking Assignments:

```
always @( posedge clock )
begin
    // Read Operation
    temp_a = a;
    temp_b = b;

    // Write Operation
    A = temp_a;
    B = temp_b;
end
```

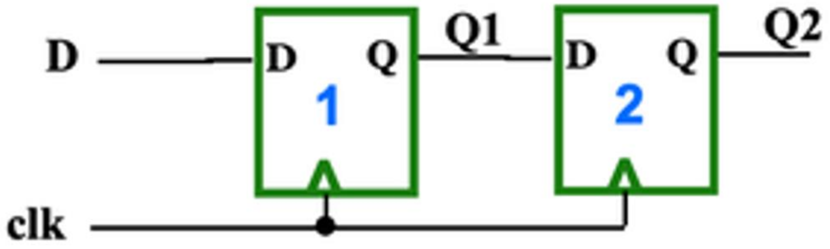
Blocking

```
module block(Q1, Q2, D, clk);
    output reg Q1,Q2;
    input D, clk;

    always@(posedge clk)
    begin
        Q1 = D;
        Q2 = Q1;
    end
endmodule
```



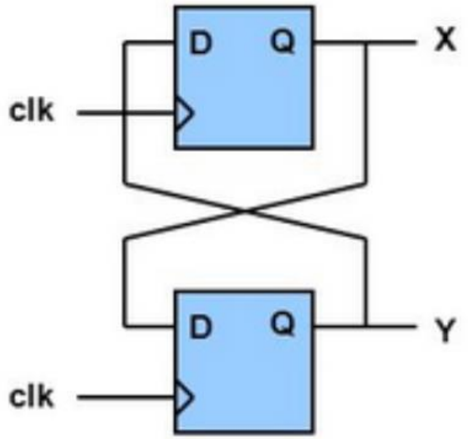
Non-Blocking

```
module non_block(Q1, Q2, D, clk);
    output reg Q1,Q2;
    input D, clk;

    always@(posedge clk)
    begin
        Q1 <= D;
        Q2 <= Q1;
    end
endmodule
```



- Use of nonblocking assignments in place of blocking assignments is highly recommended in places where concurrent data transfers take place after a common event.
- For the example below, there is a race condition.
- Either a = b would be executed before b = a, or vice versa, depending on the simulator implementation.

```
// Blocking Assignments
always @( posedge clock )
    x = y;

always @( posedge clock )
    y = x;
// Nonblocking Assignments to Eliminate Race Conditions
always @(posedge clock)
    x <= y;

always @(posedge clock)
    y <= x;
```



Parameters & Generate Blocks

Parameters

- Verilog allows **constants** to be defined in a module by the keyword **parameter**.
- Parameters cannot be used as variables. They are constants.
- Parameters values can be changed at **module instantiation**.

```
parameter port_id = 5;                    // Defines a constant port_id
parameter cache_line_width = 256;   // Constant defines width of cache line
```

Declarations

```
module hello_world #(parameter id_num = 0)();

initial
    $display("Displaying hello_world
              id number = %d", id_num);

endmodule
```

ANSI C Style Declaration

```
module hello_word();

parameter id_num = 0;

initial
    $display("Displaying hello_word
              id number = %d", id_num );
endmodule
```

Overriding

- Parameter values can be overridden when a module is instantiated.

```
module hello_world #(parameter id_num = 0);

initial
    $display("Displaying hello_world
              id number = %d", id_num);

endmodule
```

```
module top;
// Parameter value assignment by ordered list
hello_world #(1) w1;

//Parameter value assignment by name
hello_world #(.id_num(2)) w2;

endmodule
```

Defparam

- **Defparam** statement and the hierarchical name of the instance can be used to **override parameter** values.

```
module hello_word;

parameter id_num = 0;

initial
    $display("Displaying hello_word
             id number = %d", id_num );
endmodule
```

```
module top;

//change parameter values in the instantiated modules
defparam w1.id_num = 1, w2.id_num = 2;

hello_word w1();
hello_word w2();

endmodule

/*
Output:
Displaying hello_world id number = 1
Displaying hello_world id number = 2
*/
```

Generate Loop

- Generate statements are convenient when the same module instance is repeated.

```
module bitwise_xor( out, i0, i1 );

parameter N = 32;

output [N-1:0] out;
input [N-1:0] i0, i1;

genvar j;    // temp loop variable, used only
             // in the evaluation of the generate blocks

generate
for( j=0; j<N; j=j+1 )
    begin : xor_loop
        xor g1( out[j], i0[j], i1[j] );
    end
endgenerate
```

❖ Creating 32 XOR primitives using generate block

Generate Conditional

- A generate conditional is used for conditionally instantiation.

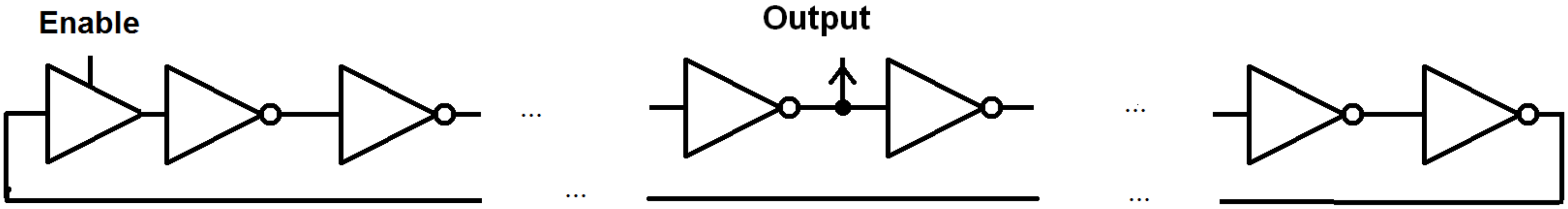
```
module multiplier( product, a0, a1 );

parameter a0_width = 8;
parameter a1_width = 8;
parameter product_width = a0_width + a1_width;

output [product_width-1:0] product;
input [a0_width-1:0] a0;
input [a1_width-1:0] a1;

// Instantiate the type of multiplier conditionally.
generate
    if( a0_width < 8) || (a1_width < 8) )
        cla_multiplier #(a0_width, a1_width) m0 (product, a0, a1);
    else
        tree_multiplier #(a0_width, a1_width) m0 (product, a0, a1);
endgenerate

endmodule
```

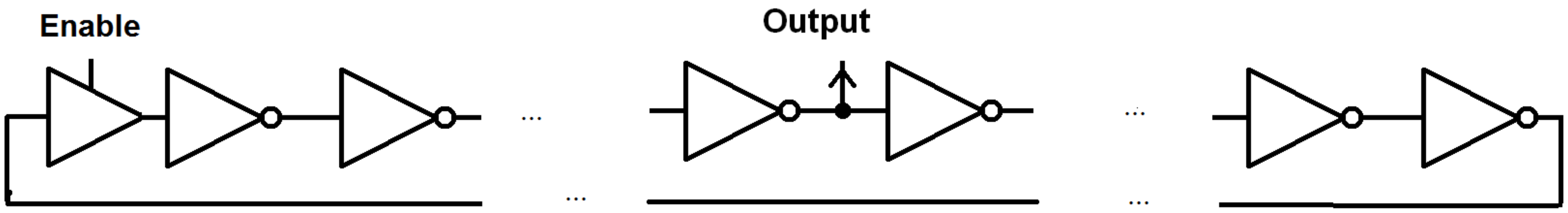
Generate Case

```
module adder( co, sum, a0, a1, ci );
parameter N = 4;

output co;  output [N-1:0] sum;
input [N-1:0] a0, a1; input ci;

// Instantiate the appropriate adder based on the width of the bus.
// This is based on parameter N that can be redefined at
// instantiation time.
generate
    case(N)
        1: adder_1bit adder1( c0, sum, a0, a1, ci );
        2: adder_2bit adder2( c0, sum, a0, a1, ci );
        default: adder_cla #(N) adder3(c0, sum, a0, a1, ci);
    endcase
endgenerate

endmodule
```

Ring Oscillator

- A **ring oscillator** is composed of an odd number of NOT gates. Its output oscillates between two voltage levels.






```
module ring_oscillator #( parameter size = 100) ( input E, output O );

(* dont_touch="true" *) wire [size-1:0]w;

genvar i;

generate
   for(i=0; i<size-1; i=i+1)
   begin
      NOT notk(.I(w[i]), .O(w[i+1]));
   end
endgenerate

TRI tr1(.I(w[size-1]), .E(E), .O(w[0]));
assign O = w[size/2 -1];

endmodule
```

Simulation & System Tasks

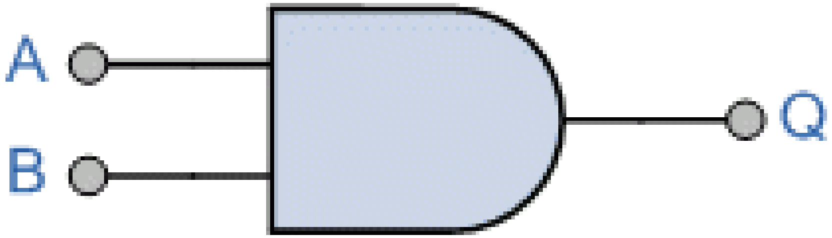
Simulation

- When using Verilog to **design** digital circuits, **testbench codes** are also created to **simulate** the design code and ensure that it functions as expected.
- A **testbench** is simply a Verilog module. But it is different from the design modules.
- A **testbench** is not implemented as a circuit, it is just used for the simulation of a design code. Therefore, design modules must be synthesizable, whereas a testbench module need not be synthesizable.

Simulation

- In testbenches, **delay units** are necessary to test possible inputs.
- The # character followed by a number are used to model delays.
- Time unit is determined by ‘`timescale` command.

- There are also some useful inbuilt tasks and functions to use in a testbench (e.g. $display, $monitor, $finish).



Design Code

```
module AND( output Q,
            input A ,
            input B );

assign Q = A & B;

endmodule
```

Testbench Code

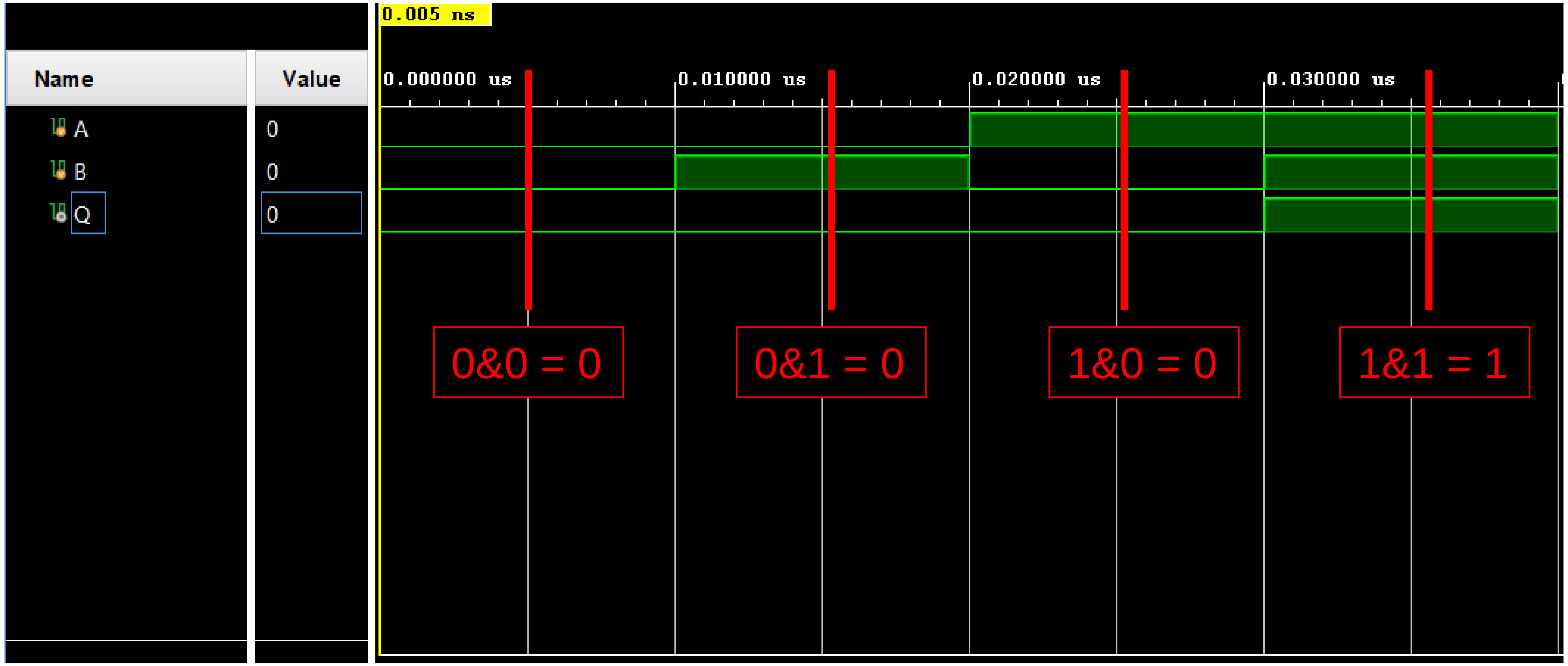
```
`timescale 1ns / 1ps // Timescale of the simulation
                     // 1 time unit= 1ns
module AND_tb();  // Testbench module
wire Q;
reg A,B;
and2 A1(Q, A, B); // Intantiate top Module of the design

initial
begin
    // monitor and show the values of A,B,Q in the console
    $monitor(" A=%b B=%b  | Q = %b",A,B,Q);

    A = 0; B = 0;       // initial values of A,B
    #10 A = 0; B = 1;   // change the values A,B after 10 time unit
    #10 A = 1; B = 0;   // change the values A,B after 10 time unit
    #10 A = 1; B = 1;   // change the values A,B after 10 time unit
    #10 $finish;        // finish the simulation after 10 time unit
end
endmodule
```

Output in the console

```
# run 1000ns
 A=0 B=0  | Q = 0
 A=0 B=1  | Q = 0
 A=1 B=0  | Q = 0
 A=1 B=1  | Q = 1
```

Simulation

- Using a **simulation tool** which allows for **waveforms** to be viewed directly is very useful to verify your design.



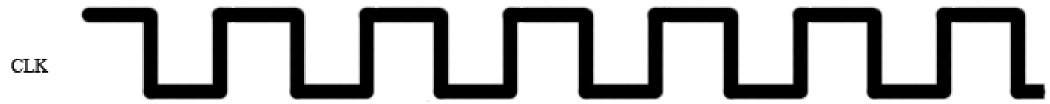
And Gate

Waveform is taken from Vivado

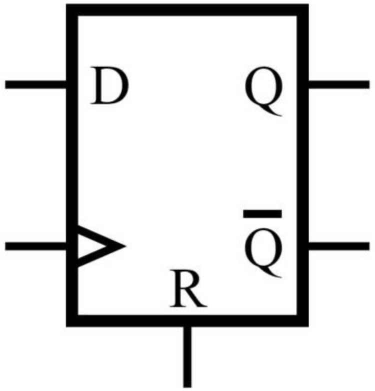
Simulation

- For sequential circuits, the clock signals are essential for its functioning. Hence, a virtual clock is necessary in the testbench to simulate the sequential circuits.
- **`posedge`** and **`negedge`** keywords are used to refer rising edge and falling edge of the corresponding signal respectively.

```
// Virtual Clock:
always
    #10 CLK = ~CLK;
```



Design Code



D-FF

```
module dff( clk, rst, d, q, qbar );

input clk,rst,d;
output reg q, qbar;

// "posedge: rising edge", "negedge: falling edge"
// of the corresponding signal
always@(posedge clk)
begin
    if(rst == 1)
    begin
        q <= 0;
        qbar <= 1;
    end
    else
    begin
        q <= d;
        qbar <= ~d;
    end
end
endmodule
```

ITU VLSI LABS

Testbench Code

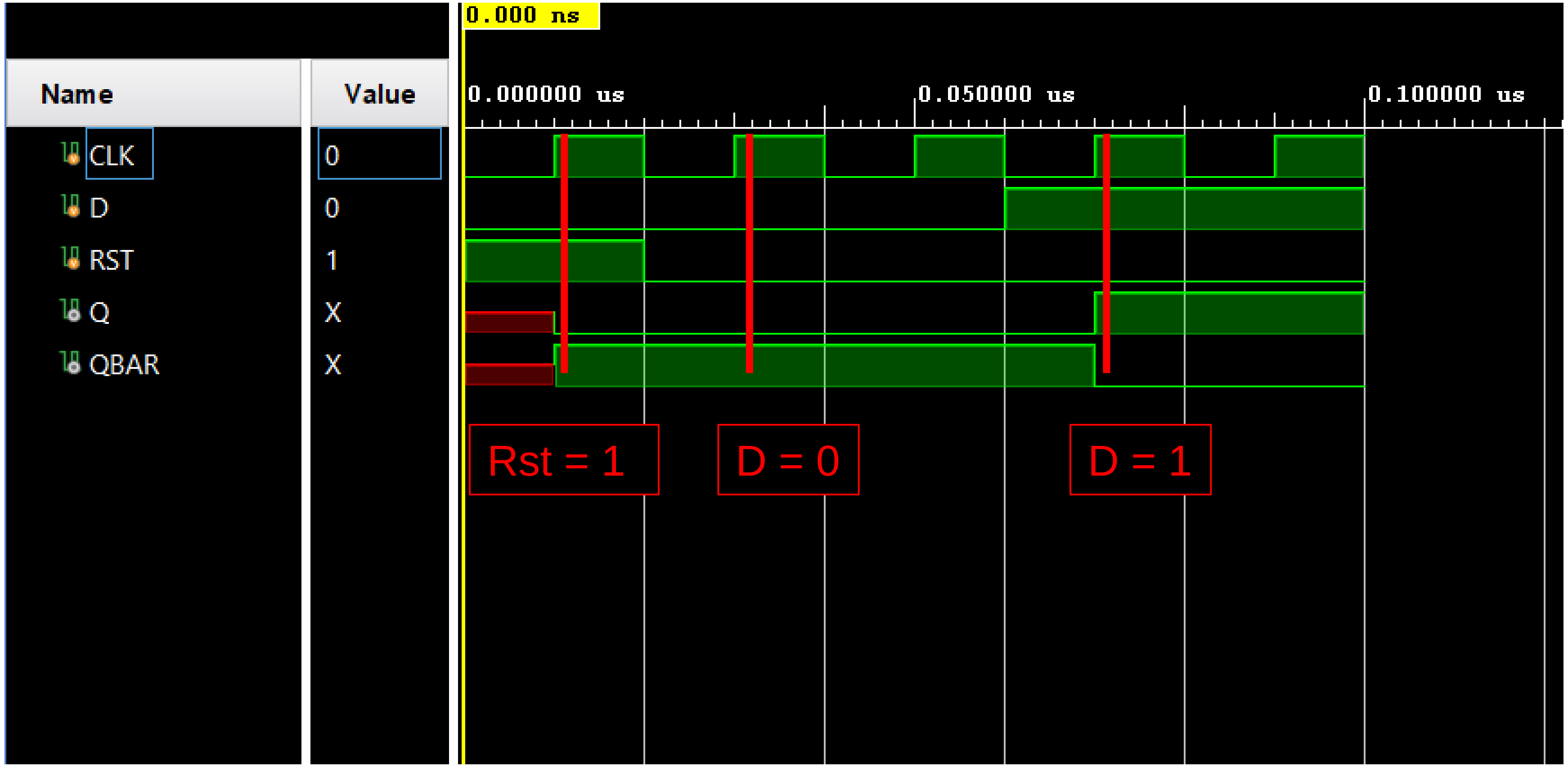
```
`timescale 1ns / 1ps

module dff_tb();

reg CLK = 0;
reg D,RST;
wire Q,QBAR;

dff DFF(.clk(CLK), .rst(RST), .d(D), .q(Q), .qbar(QBAR));

always // Virtual Clock:
    #10 CLK = ~CLK;

initial
begin
    $monitor("simetime = %g, CLK = %b, RST =%b, D = %b, Q
=%b, QBAR =%b", $time, CLK,RST,D,Q,QBAR);
    D=0; RST = 1;
    #20 RST = 0;
    #20 D = 0;
    #20 D = 1;
    #40 $finish;
end
endmodule
```

Output in the console

```
simetime = 0, CLK = 0, RST =1, D = 0, Q =x, QBAR =x
simetime = 10, CLK = 1, RST =1, D = 0, Q =0, QBAR =1
simetime = 20, CLK = 0, RST =0, D = 0, Q =0, QBAR =1
simetime = 30, CLK = 1, RST =0, D = 0, Q =0, QBAR =1
simetime = 40, CLK = 0, RST =0, D = 0, Q =0, QBAR =1
simetime = 50, CLK = 1, RST =0, D = 0, Q =0, QBAR =1
simetime = 60, CLK = 0, RST =0, D = 1, Q =0, QBAR =1
simetime = 70, CLK = 1, RST =0, D = 1, Q =1, QBAR =0
simetime = 80, CLK = 0, RST =0, D = 1, Q =1, QBAR =0
simetime = 90, CLK = 1, RST =0, D = 1, Q =1, QBAR =0
```

ITU VLSI LABS

Waveform



Timing Control / Delays

- Delay values control the time between the change in a right-hand-side operand and when the new value is assigned to the left-hand side.
- Delays are **not synthesizable**! Timing control is used for simulations and verification.
- Timing control can be made for both continuous assignments and procedural assignments.

Delays

- For continuous assignments, timing control can be made by:

```
// Any change in values of in1 or in2 will result in a delay of 10 time units
assign #10 out = in1 & in2;


// An equivalent method
wire #10 out = in1 & in2;

//same as
wire out;
assign #10 out = in1 & in2;
```

Delays

- For procedural assignments timing control can be made by:

```
initial
begin
    x = 0; z = 0;

    #5 y = x + z;  // wait 5 time units, take value of x and z at the time=5,
                   // evaluate x + z and then assign value to y
end
```

Display

- $display is used for displaying values, strings or expressions. Like printf in C.
- Syntax:

```
$display(p1, p2, p3,....., pn);
```

Format	Display
%d or %D	Display variable in decimal
%b or %B	Display variable in binary
%s or %S	Display string
%h or %H	Display variable in hex

%c or %C	Display ASCII character
%m or %M	Display hierarchical name (no argument required)
%v or %V	Display strength
%o or %O	Display variable in octal
%t or %T	Display in current time format
%e or %E	Display real number in scientific format (e.g., 3e10)
%f or %F	Display real number in decimal format (e.g., 2.13)
%g or %G	Display real number in scientific or decimal, whichever is shorter

```
//Display the string in quotes
$display("Hello Verilog World");
-- Hello Verilog World

//Display value of current simulation time 230
$display($time);
-- 230

 //Display value of 41-bit virtual address 1fe0000001c at time 200
reg [0:40] virtual_addr;
$display("At time %d virtual address is %h", $time, virtual_addr);
-- At time 200 virtual address is 1fe0000001c

//Display value of port_id 5 in binary
reg [4:0] port_id;
$display("ID of the port is %b", port_id);
-- ID of the port is 00101
```

Hierarchical Name of Instances

- Verilog allows the displaying values of lower level instances.

```
module Z;
reg [1:0] c=2;
Y y1(); //instance
endmodule


module Y;
reg b=1;
X x1(); //instance
endmodule


module X;
reg a=0;
endmodule
```

```
module tb;

Z z1();

initial
begin
$display("value of a in instance x1 is %d", z1.y1.x1.a );
$display("value of b in instance y1 is %d", z1.y1.b );
$display("value of c in instance z1 is %d", z1.c );

end
endmodule
```

Monitor

- $monitor **continuously** monitors the values of the variables or signals whereas $display displays the values exactly **once**.

```
//Monitor time and value of the signals clock and
reset
module tb();

wire CNT;
reg CLK = 0; reg RST = 1;
counter C1( CNT, CLK, RST );

initial
   $monitor($time, " Value of signals clock = %b
reset = %b", CLK,RST);

always #5 CLK = ~CLK;

initial #10 RST = ~RST;
endmodule
```

```
Log:
-- 0  Value of signals CLK = 0 RST = 1
-- 5  Value of signals CLK = 1 RST = 1
-- 10 Value of signals CLK = 0 RST = 0
```

ITU VLSI LABS

Stop and Finish

- $stop suspends the simulation.
- $finish terminates the simulation.

```
// Stop at time 100 in the simulation and examine the results
// Finish the simulation at time 1000.
initial
begin
clock = 0;
reset = 1;
#100 $stop;   // This will suspend the simulation at time = 100
#900 $finish; // This will terminate the simulation at time = 1000
end
```

Timescales

- Verilog simulation depends on how time is defined because the simulator needs to know what a #1.
- Syntax:

```
`timescale <reference_time_unit> / <time_precision>
```

- The time precision specifies how delay values are rounded off. Delays in the circuit are rounded according to the precision value.

Timescales

- Example:

```
'timescale 1ns/1ps
'timescale 10us/100ns
'timescale 10ns/1ns
```

Character	Unit
s	seconds
ms	milliseconds
us	microseconds
ns	nanoseconds
ps	picoseconds
fs	femtoseconds

Time & Realtime

- **`$time`** and **`$realtime`** system functions return the current time of the simulation.
- **`$time`** round offs the time to nearby integer whereas **`$realtime`** does not. So **`$realtime`** uses real valued delays and **`$time`** integer valued delays.

```
'timescale 1ns/1ns
module tb;
reg val;

initial
begin
val = 0;
#1 $display("T=%t at time #1", $realtime);

val = 1;
#0.49 $display("T=%t at time #0.49", $realtime);  // rounded to the 0ns (precision)

val = 0;
#0.5 $display("T=%t at time #0.50", $realtime); // rounded to the 1ns (precision)

val = 1;
#0.51 $display("T=%t at time #0.51", $realtime); // rounded to the 1ns (precision)

#5 $finish;
end
endmodule
```

```
Log:

T=1 at time #1
T=1 at time #0.49
T=2 at time #0.50
T=3 at time #0.51
```

```verilog
'timescale 10ns/1ns
module tb;
reg val;

initial
begin
val = 0;
#1 $display("T=%t at time #1", $realtime);

val = 1;
#0.49 $display("T=%t at time #0.49", $realtime);  // rounded to the 5ns (precision)

val = 0;
#0.5 $display("T=%t at time #0.50", $realtime);

val = 1;
#0.51 $display("T=%t at time #0.51", $realtime); // rounded to the 5ns (precision)

#5 $finish;
end
endmodule
```

Log:

```
T=10 at time #1
T=15 at time #0.49
T=20 at time #0.50
T=25 at time #0.51
```

Directives & Functions

Define Directive - Macros

- The **`define** directive is used to define text macros.

```
// Define a size
'define WORD_SIZE 32

// Define a data type
'define WORD_REG reg [31:0]

// Define a function
'define add(A,B) A+B

// define an alias for a system task.
// $stop will be substituted with 'S
'define S $stop;
```

Define Directive

```
'define val 10
'define add(A,B) A+B

module example();

integer var_a, var_b;

var_a = 'val + 45;              // val_a = 55
var_b = 'add(var_a, 45);        // var_b = 100

endmodule
```

ITU VLSI LABS

Define Directive

- Multiline macros:

```
'define CALC(VAL1, VAL2, RESULT, EXPR)   \
         RESULT = VAL1 EXPR VAL2;        \
         $display("Result is %d, RESULT);

module example();
int a=15, b=7;
int c;

initial
begin
    'CALC( a,b,c,+ ); // c = a + b
end
endmodule
```

Conditional Compilation

- A particular portion of a testbench code can be compiled by using compiler directives:
  - **`` `ifdef, 'ifndef, ``**
  - **`` `else, `elsif,`endif ``**

- Conditional compilation can be useful to conditionally output the debug messages on the terminal or an output file.

ITU VLSI LABS

Conditional Compilation

```
module tb;

initial
begin
    'ifndef MACRO1
        $display("This is for MACRO1");
    'elseif MACRO2
        $display("This is MACRO2");
    'endif
end
endmodule
```

Conditional Compilation-Instantiation

```
module top;

bus_master b1(); //instantiate module unconditionally

// b2 is instantiated conditionally if text macro ADD_B2 is defined
'ifdef ADD_B2
    bus_master b2();

// b3 is instantiated conditionally if text macro ADD_B3 is defined
'elsif ADD_B3
    bus_master b3();
//b4 is instantiate by default
'else
    bus_master b4();
'endif

endmodule
```

Functions

- There can be repetitive pieces of code exist inside a design. In such cases, functions can be used in order to reduce the amount of code.
- Functions in Verilog are very similar to functions in C.

```
// Function Definition
function calc_parity;

input [31:0] address;
begin
    //internal register calc_parity.
    calc_parity = ^address;  // return
end
endfunction
```

```
// ANSI C Style:
function calc_parity (input [31:0] address);

begin
    //internal register calc_parity.
    calc_parity = ^address;
end

endfunction
```

Functions

- At least one input argument must be defined for a function.
- There are no output arguments for functions because the implicit register *function_identifer* contains the output value.
- We can define an optional range or type specifies the width of the internal register. The default bit width is 1.

```
// ANSI C Style:
function calc_parity (input [31:0] address);

begin
    //internal register calc_parity.
    calc_parity = ^address;
end

endfunction
```

```
module Parity_check;

reg [31:0] addr;
reg parity;

always @(addr)
Begin
    // function call, 1 bit output
    parity = calc_parity(addr);
end
```

Automatic (Recursive) Functions

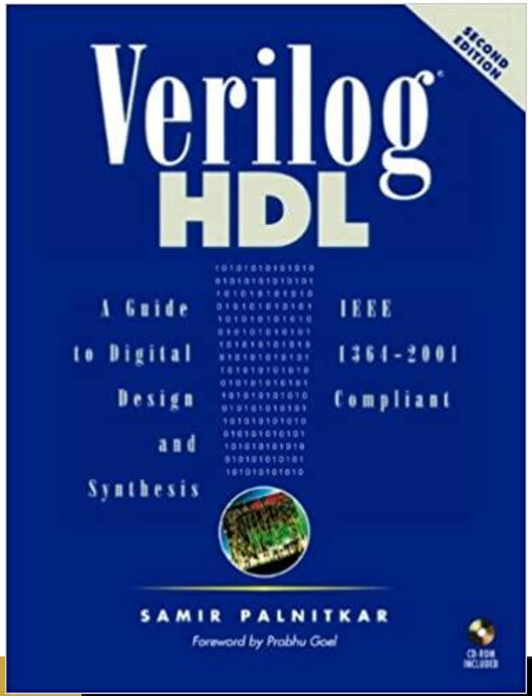
- If a function is called concurrently from two locations, the results are non-deterministic because both calls operate on the same variable space!
- The keyword automatic can be used to declare a recursive (automatic) function where all function declarations are allocated dynamically for each recursive calls.
- Each call to an automatic function operates in an independent variable space.

Automatic (Recursive) Functions

```
function automatic integer factorial; // output is integer type

input [31:0] oper;
integer i;

begin
    if ( oper >= 2 )
        // recursive call
        factorial = factorial( oper - 1 ) * oper;
    else
        factorial = 1;
end
endfunction
```

Reference

- For further information, the following textbook is a good option:
  - *Verilog HDL: A Guide to Digital Design and Synthesis, Second Edition, Samir Palnitkar*



- Useful online resources about Verilog:
  - https://www.chipverify.com
  - https://reference.digilentinc.com/start
  - https://www.xilinx.com/support/university.html

Thank you for listening!