Discussions

- Overview of Digital Design with VerilogHDL
- Hierarchical Modeling Concepts
- Basic Concepts
- Modules and Ports
- Gate-Level Modeling
- Data Flow Modeling
- Behavioral Modeling
- Switch-Level Modeling

Lecture – 1
Overview of Digital Design with VerilogHDL

Evolution of Computer-Aided Digital Design

Emergence of HDLs

Typical Design Flow

Importance of HDLs

Popularity of VerilogHDL

Evolution of Computer-Aided Digital Design

- SSI (Small Scale Integration) chips where the gate count was very small.
- MSI (Medium Scale Integration) chips with hundreds of gates on a chip.
- LSI (Large Scale Integration), with thousands of gates on a single chip.
- Design processes started getting very complicated, and designers felt the need to automate these processes.
- Electronic Design Automation (EDA) techniques began to evolve.

Evolution of Computer-Aided Digital Design

- Circuit and logic simulation techniques to verify the functionality of building blocks of the order of about 1 million transistors.

- Computer-Aided Design (CAD) tools refer to back- end tools that perform functions related to place and route, and layout of the chip.

- Computer-Aided Engineering (CAE) tools refer to tools that are used for front-end processes such HDL simulation, logic synthesis, and timing analysis.

- Today, the term Electronic Design Automation is used for both CAD and CAE. So, all design tools are referred as EDA tools.

Emergence of HDLs

- Programming languages such as FORTRAN, Pascal, and C were being used to describe computer programs that were sequential in nature.

- Similarly, 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.

- HDLs allowed the designers to model the concurrency of processes found in hardware elements.
- Hardware description languages such as Verilog HDL and VHDL became popular.

- Verilog HDL originated in 1983 at Gateway Design Automation.

- Later, VHDL was developed under contract from DARPA. Both Verilog and VHDL simulators to simulate large digital circuits quickly gained acceptance from designers.

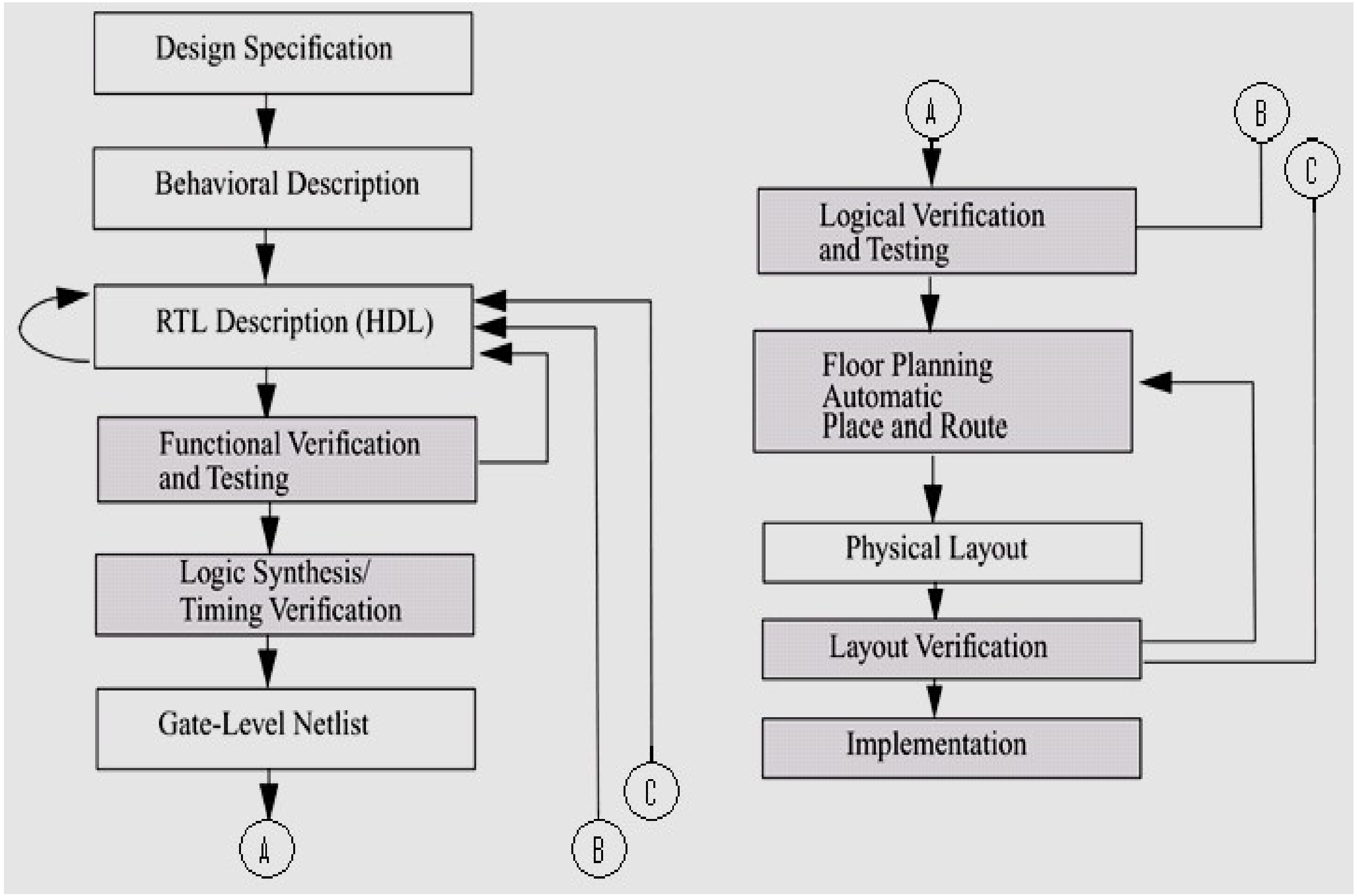
Emergence of HDLs

- Even though HDLs were popular for logic verification, designers had to manually translate the HDL-based design into a schematic circuit with interconnections between gates.
- The advent of logic synthesis in the late 1980s changed the design methodology radically.
- Digital circuits could be described at a register transfer level (RTL) by use of an HDL.
- Thus, the designer had to specify how the data flows between registers and how the design processes the data.
- The details of gates and their interconnections to implement the circuit were automatically extracted by logic synthesis tools from the RTL description.
- Thus, logic synthesis pushed the HDLs into the forefront of digital design.

Emergence of HDLs

- Designers no longer had to manually place gates to build digital circuits. They could describe complex circuits at an abstract level in terms of functionality and data flow by designing those circuits in HDLs.
- Logic synthesis tools would implement the specified functionality in terms of gates and gate interconnections.
- HDLs also began to be used for system-level design. HDLs were used for simulation of system boards, interconnect buses, FPGAs (Field Programmable Gate Arrays), and PALs (Programmable Array Logic).
- A common approach is to design each IC chip, using an HDL, and then verify system functionality via simulation.

Typical Design Flow


Design Specification
Behavioral Description
RTL Description (HDL)
Functional Verification and Testing
Logic Synthesis/ Timing Verification
Gate-Level Netlist
A
B
C
A
B
C
Logical Verification and Testing
Floor Planning Automatic Place and Route
Physical Layout
Layout Verification
Implementation

Typical Design Flow

- The design flow shown in above is typically used by designers who use HDLs.
- In any design, specifications are written first.

- Specifications describe abstractly the functionality, interface, and overall architecture of the digital circuit to be designed.

- At this point, the architects do not need to think about how they will implement this circuit.

- A behavioral description is then created to analyze the design in terms of functionality, performance, compliance to standards, and other high-level issues.

Typical Design Flow

- Behavioral descriptions are often written with HDLs.
- The behavioral description is manually converted to an RTL description in an HDL.
- The designer has to describe the data flow that will implement the desired digital circuit.
- From this point onward, the design process is done with the assistance of EDA tools.
- Logic synthesis tools convert the RTL description to a gate-level netlist.
- A gate-level netlist is a description of the circuit in terms of gates and connections between them.
- Logic synthesis tools ensure that the gate- level netlist meets timing, area, and power specifications.

Typical Design Flow

- The gate-level netlist is input to an Automatic Place and Route tool, which creates a layout.
- The layout is verified and then fabricated on a chip.

- Thus, most digital design activity is concentrated on manually optimizing the RTL description of the circuit.
- After the RTL description is frozen, EDA tools are available to assist the designer in further processes.

- Designing at the RTL level has shrunk the design cycle times from years to a few months.
- It is also possible to do many design iterations in a short period of time.

Typical Design Flow

- Behavioral synthesis tools have begun to emerge recently.
- These tools can create RTL descriptions from a behavioral or algorithmic description of the circuit.
- As these tools mature, digital circuit design will become similar to high-level computer programming.
- Designers will simply implement the algorithm in an HDL at a very abstract level.
- EDA tools will help the designer convert the behavioral description to a final IC chip.

Typical Design Flow

- Although EDA tools are available to automate the processes and cut design cycle times, the designer is still the person who controls how the tool will perform.
- If used improperly, EDA tools will lead to inefficient designs. Thus, the designer still needs to understand the nuances of design methodologies, using EDA tools to obtain an optimized design.

Importance of HDLs

- HDLs have many advantages compared to traditional schematic-based design
- Designs can be described at a very abstract level by use of HDLs.
  - Designers can write their RTL description without choosing a specific fabrication technology.
  - Logic synthesis tools can automatically convert the design to any fabrication technology.
  - If a new technology emerges, designers do not need to redesign their circuit. They simply input the RTL description to the logic synthesis tool and create a new gate-level netlist, using the new fabrication technology.
  - The logic synthesis tool will optimize the circuit in area and timing for the new technology.

Importance of HDLs

- By describing designs in HDLs, functional verification of the design can be done early in the design cycle.
  - Since designers work at the RTL level, they can optimize and modify the RTL description until it meets the desired functionality.
  - Most design bugs are eliminated at this point. This cuts down design cycle time significantly because the probability of hitting a functional bug at a later time in the gate-level netlist or physical layout is minimized.

Importance of HDLs

- Designing with HDLs is analogous to computer programming.
  - A textual description with comments is an easier way to develop and debug circuits.
  - This also provides a concise representation of the design, compared to gate-level schematics. Gate-level schematics are almost incomprehensible for very complex designs.
- HDL-based design is here to stay.
  - With rapidly increasing complexities of digital circuits and increasingly sophisticated EDA tools, HDLs are now the dominant method for large digital designs.
  - No digital circuit designer can afford to ignore HDL-based design.

Popularity of VerilogHDL

- VerilogHDL has evolved as a standard hardware description language. VerilogHDL offers many useful features
  - VerilogHDL is a general-purpose hardware description language that is easy to learn and easy to use. It is similar in syntax to the C programming language.
  - VerilogHDL allows different levels of abstraction to be mixed in the same model. Thus, a designer can define a hardware model in terms of switches, gates, RTL, or behavioral code.
  - Most popular logic synthesis tools support VerilogHDL. This makes it the language of choice for designers.
  - All fabrication vendors provide VerilogHDL libraries for post- logic synthesis simulation.
  - The Programming Language Interface (PLI) is a powerful feature that allows the user to write custom C code to interact with the internal data structures of VerilogHDL. Designers can customize a VerilogHDL simulator to their needs with the PLI.

Hierarchical Modeling Concepts

Design Methodologies

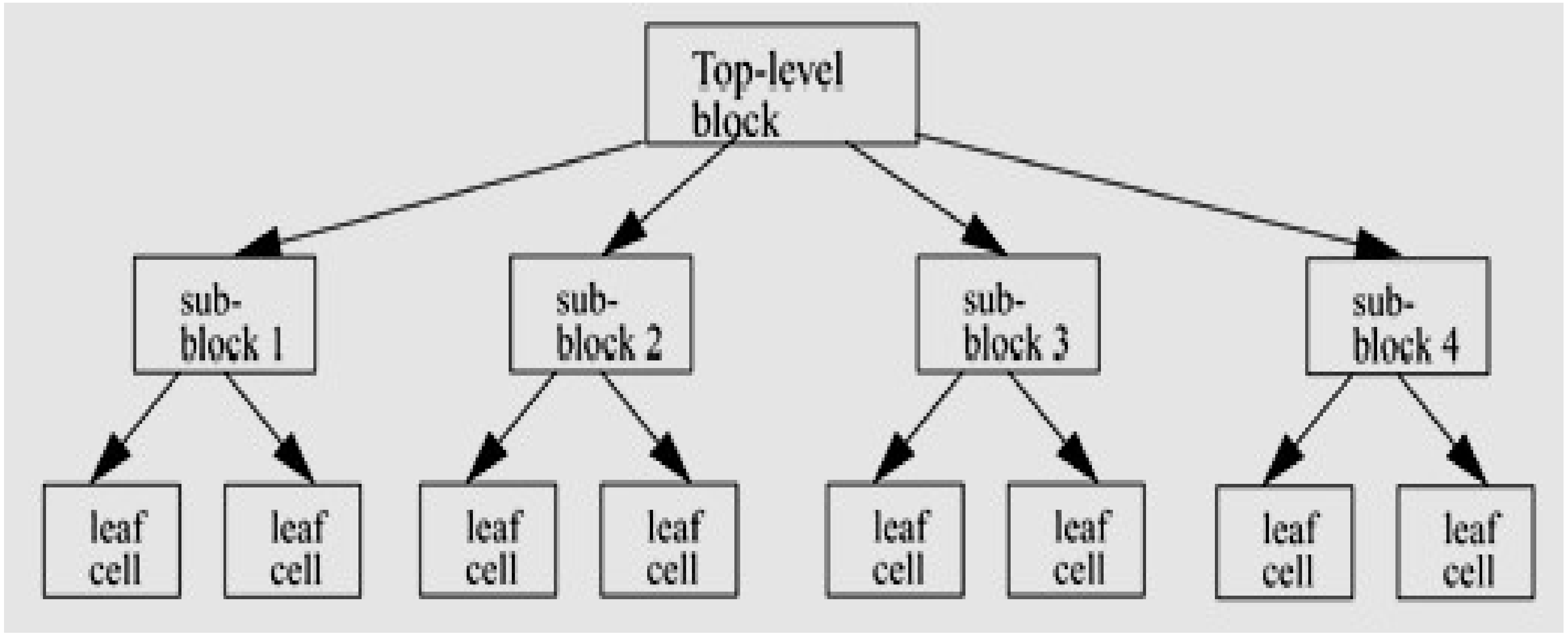
Modules

Instances

Components of a Simulation

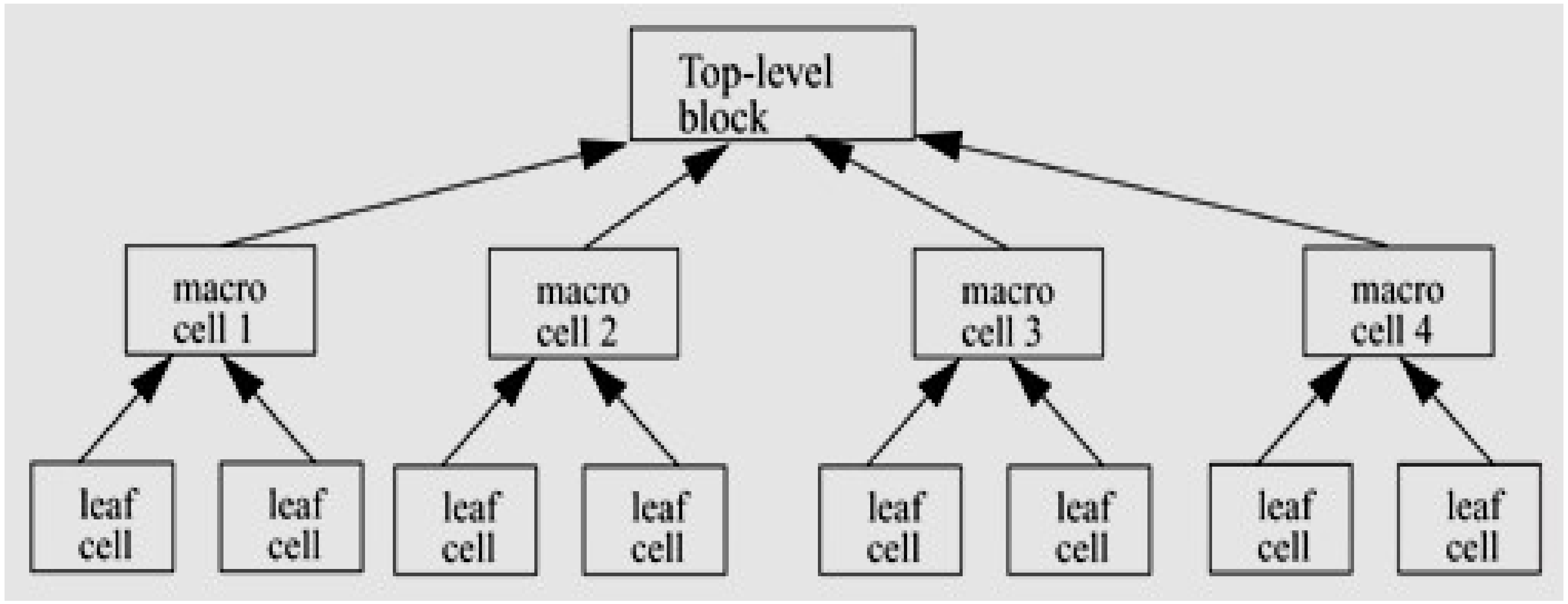
Design Methodologies

- Top-down Design Methodology
  - Define the top-level block and identify the sub-blocks necessary to build the top-level block.
  - Further subdivide the sub-blocks until the leaf cells, which are the cells that cannot further be divided, are achieved.



Design Methodologies

- Bottom-up Design Methodology
  - Identify the building blocks that are available.
  - Build bigger cells, using these building blocks. These cells are then used for higher-level blocks until the top-level block in the design is built.

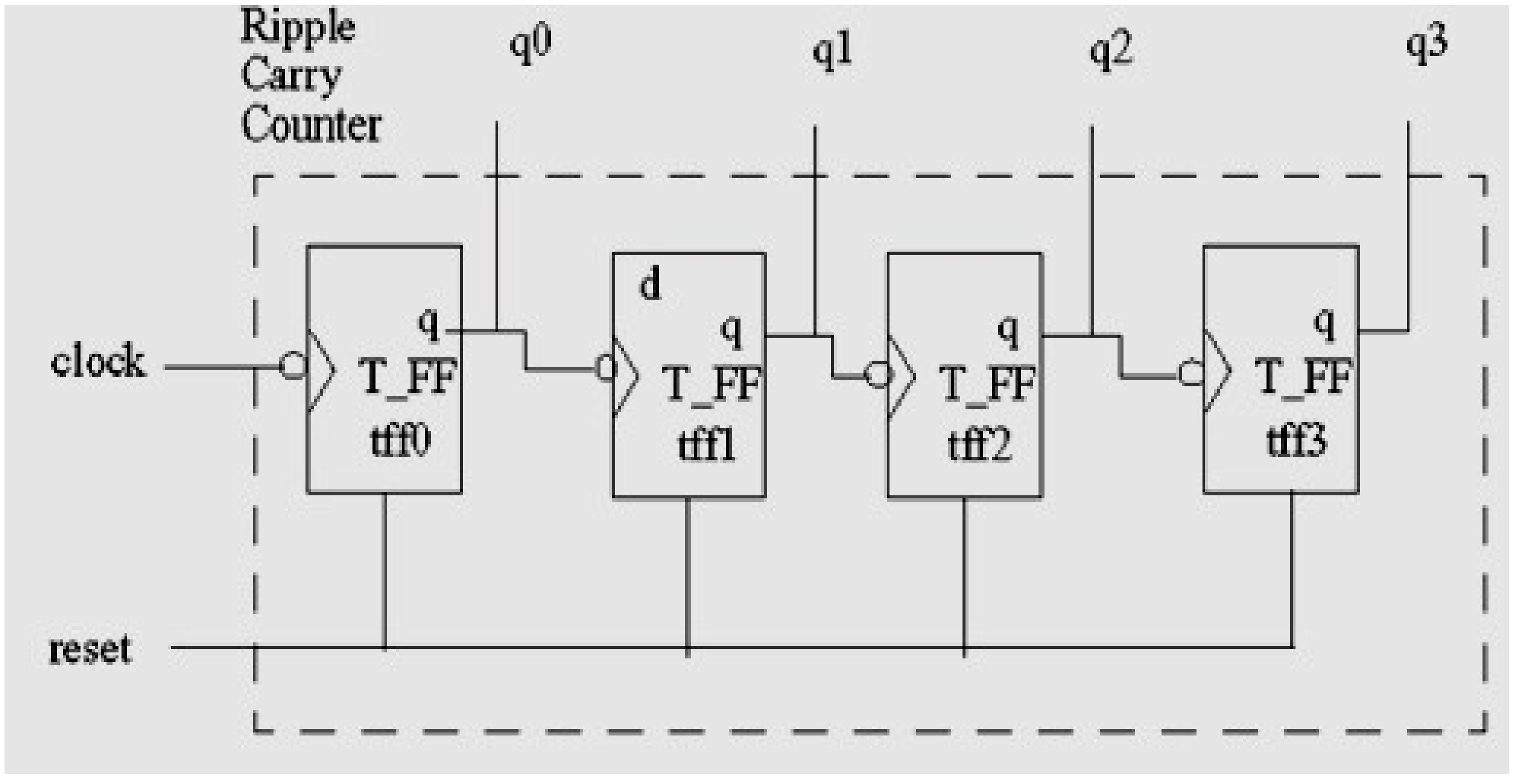


Design Methodologies

- Typically, a combination of top-down and bottom-up flows is used.
- Design architects define the specifications of the top-level block.
- Logic designers decide how the design should be structured by breaking up the functionality into blocks and sub-blocks.
- At the same time, circuit designers are designing optimized circuits for leaf-level cells. They build higher-level cells by using these leaf cells.
- The flow meets at an intermediate point where the switch-level circuit designers have created a library of leaf cells by using switches, and the logic level designers have designed from top-down until all modules are defined in terms of leaf cells.

4-bit Ripple Carry Counter

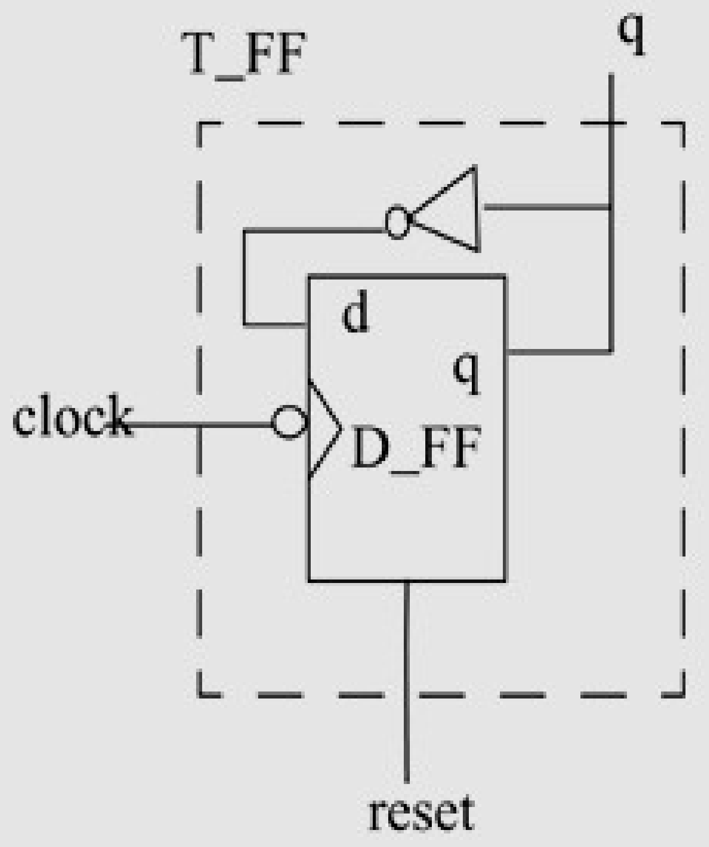
- The ripple carry counter as shown in figure is made up of negative edge-triggered toggle flipflops (T-FF).



4-bit Ripple Carry Counter

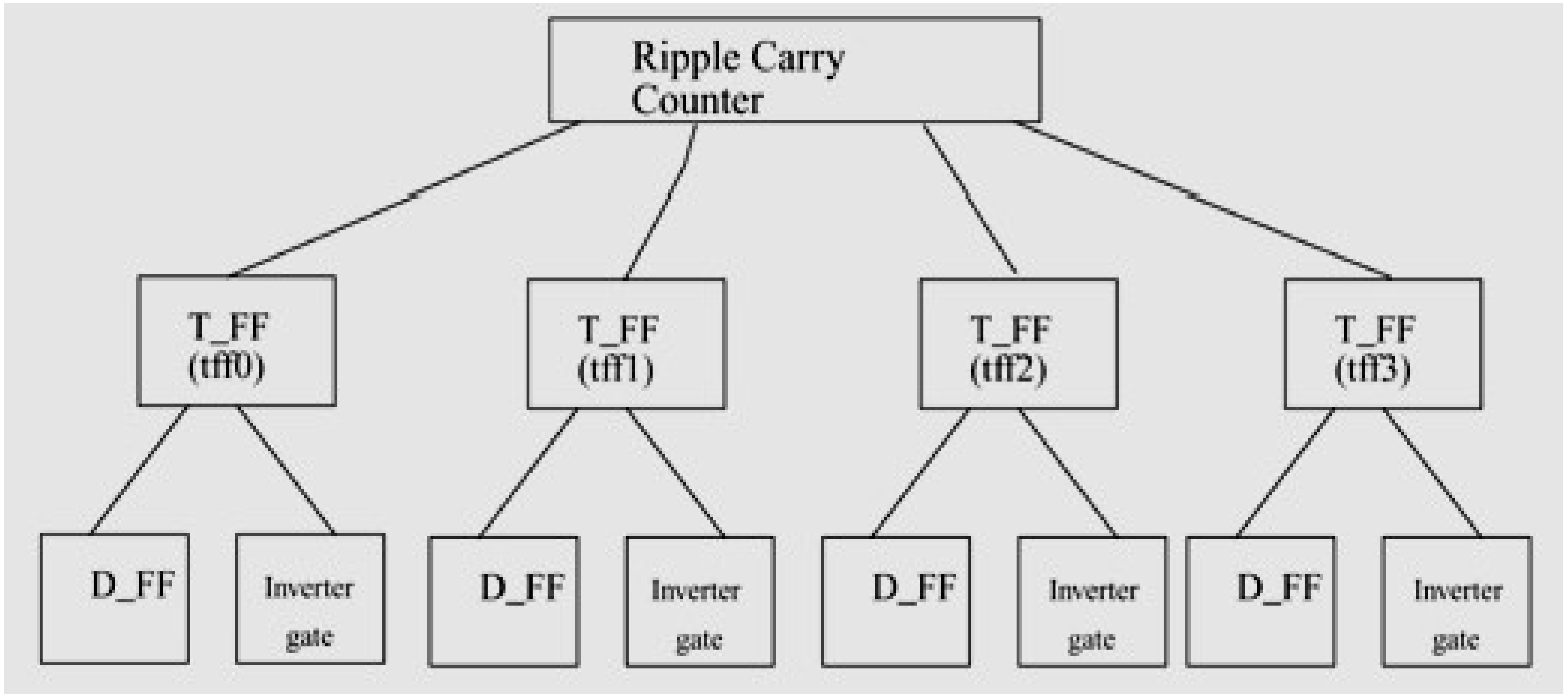
- Each of the T_FFs can be made up from negative edge-triggered D-flipflops (D_FF) and inverters (assuming q_bar output is not available on the D_FF), as shown in figure.

reset	q_n	q_{n+1}
1	1	0
1	0	0
0	0	1
0	1	0
0	0	0



4-bit Ripple Carry Counter

- Thus, the ripple carry counter is built in a hierarchical fashion by using building blocks. The diagram for the design hierarchy is shown in figure.



4-bit Ripple Carry Counter

- In a top-down design methodology,
  - First, specify the functionality of the ripple carry counter, which is the top-level block.
  - Then, implement the counter with T_FFs.
  - Build the T_FFs from the D_FF and an additional inverter gate.
  - Thus, break bigger blocks into smaller building sub-blocks until the blocks cannot be broken up any further.
- A bottom-up methodology flows in the opposite direction.
  - Combine small building blocks and build bigger blocks
  - Build D_FF from and and or gates, or build a custom D_FF from transistors.
- Thus, the bottom-up flow meets the top-down flow at the level of the D_FF.

Basic Concepts

Lexical Conventions

Data Types

System Tasks and Compiler Directives

Lexical Conventions

- The basic lexical conventions used by VerilogHDL are similar to those in the C programming language.
- VerilogHDL contains a stream of tokens.
- Tokens can be comments, delimiters, numbers, strings, identifiers, and keywords.
- All keywords are in lowercase.

Lexical Conventions

Whitespace

- Blank spaces (\b) , tabs (\t) and new lines (\n) comprise the whitespace.
- Whitespace is ignored by VerilogHDL except when it separates tokens.
- Whitespace is not ignored in strings.

Lexical Conventions

Comments

- There are two ways to write comments.
  - A one-line comment starts with "//". Verilog skips from that point to the end of line.

    a = b && c; // This is a one-line comment
  - A multiple-line comment starts with "/*" and ends with "*/".
  - Multiple-line comments cannot be nested.
  - However, one-line comments can be embedded in multiple-line comments.

```
/* This is a multiple line
                       comment */
/* This is /* an illegal */ comment */
/* This is //a legal comment */
```

Lexical Conventions

Operators

- Operators are of three types: unary, binary, and ternary.
  - Unary operators precede the operand.

    ```
    a = ~ b; // ~ is a unary operator. b is the  operand
    ```
  - Binary operators appear between two operands.

    ```
    a = b && c; // && is a binary operator.
                // b and c are  operands
    ```
  - Ternary operators have two separate operators that separate three operands.

    ```
    a = b ? c : d; // ?: is a  ternary operator.
                   // b, c and d are  operands
    ```

Lexical Conventions

Number Specification

- Two types of number specification: sized and unsized.
- Sized numbers
  - Sized numbers are represented as

    <size> '<base format> <number>.
  - <size> is written only in decimal and specifies the number of bits in the number.
  - Legal base formats are decimal ('d or 'D), hexadecimal ('h or 'H), binary ('b or 'B) and octal ('o or 'O).
  - The number is specified as consecutive digits from 0 – 9, a – f.
  - Only a subset of these digits is legal for a particular base.
  - Uppercase letters are legal for number specification.

```
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.
```

Lexical Conventions

<u>Number Specification</u>

- Unsized numbers
  - Numbers that are specified without a <base format> specification are decimal numbers by default.
  - 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).

```
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
```

Lexical Conventions

Number Specification

- X or Z values
  - VerilogHDL 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.
  - These values are very important for modeling real circuits.

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

Lexical Conventions

Number Specification

- X or Z values
  - An x or z sets four bits for a number in the hexadecimal base, three bits for a number in the octal base, and one bit for a number in the binary base.
  - 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.
  - This makes it easy to assign x or z to whole vector.
  - If the most significant digit is 1, then it is also zero extended.

Lexical Conventions

Number Specification

- **Negative numbers**
  - Negative numbers can be specified by putting a minus sign before the size for a constant number.
  - Size constants are always positive.
  - It is illegal to have a minus sign between <base format> and <number>.
  - An optional signed specifier can be added for signed arithmetic.

```
-6'd3    // 8-bit negative number stored as 2's complement of 3
-6'sd3  // Used for performing signed integer math
4'd-2   // Illegal specification
```

Lexical Conventions

Number Specification

- Underscore characters
  - 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
```

Lexical Conventions

Number Specification

- Question marks
  - A question mark "?" is the Verilog HDL alternative for z in the context of numbers.
  - The “?” is used to enhance readability in the casex and casez statements, where the high impedance value is a don't care condition.
  - The “?” has a different meaning in the context of user-defined primitives.

```
4'b10??      // Equivalent of a 4'b10zz
```

Lexical Conventions

Strings

- A string is a sequence of characters that are enclosed by double quotes.
- The restriction on a string is that it must be contained on a single line, that is, without a carriage return. It cannot be on multiple lines.
- Strings are treated as a sequence of one-byte ASCII values.

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

Lexical Conventions

Identifiers and Keywords

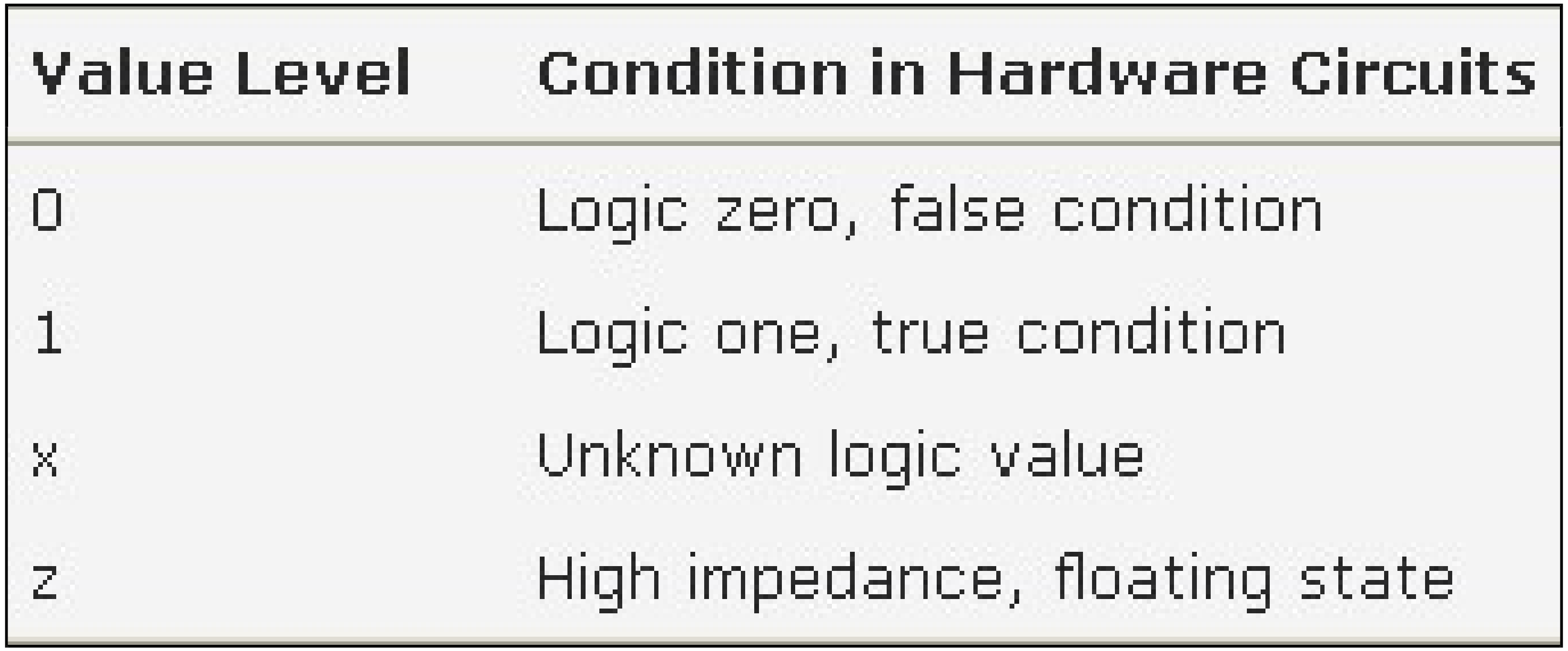
- Identifiers
  - Identifiers are names given to objects so that they can be referenced in the design.
  - Identifiers are made up of alphanumeric characters, the underscore (_), or the dollar sign ($).
  - Identifiers are case sensitive.
  - Identifiers start with an alphabetic character or an underscore.
  - They cannot start with a digit or a $ sign
- Keywords
  - Keywords are special identifiers reserved to define the language constructs.
  - Keywords are in lowercase.

reg value; // reg is a keyword; value is an identifier

input clk; // input is a keyword, clk is an identifier

Data Types (Value Set)

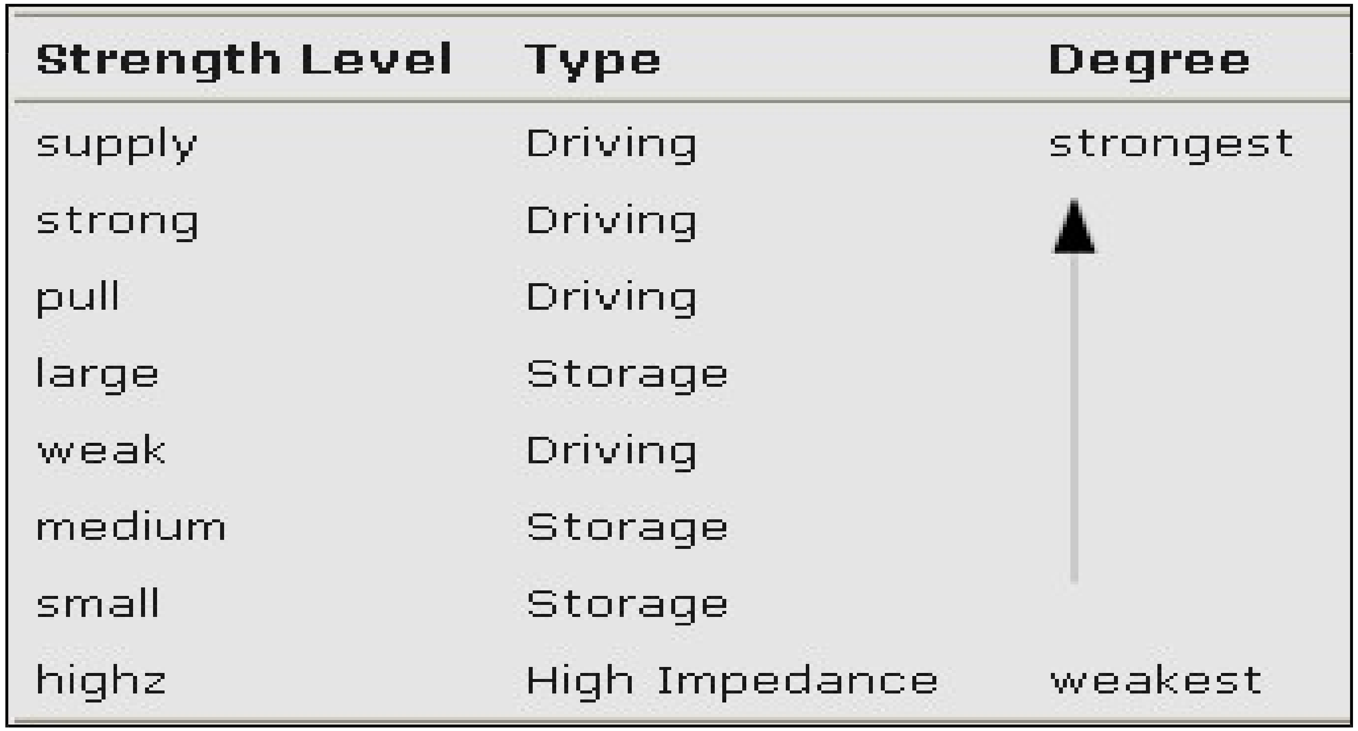
- **Verilog supports four values and eight strengths to model the functionality of real hardware.**
- **Value Levels**



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

Data Types (Value Set)

- **Strength Levels**
  - **In addition to logic values, strength levels are often used to resolve conflicts between drivers of different strengths in digital circuits. Value levels 0 and 1 can have the strength levels listed as below.**



Strength Level	Type	Degree
supply	Driving	strongest
strong	Driving	▲
pull	Driving	
large	Storage	
weak	Driving	
medium	Storage	
small	Storage	
highz	High Impedance	weakest

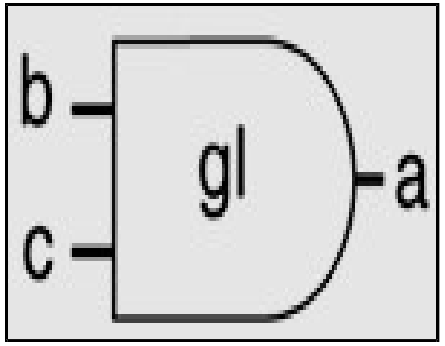
Data Types (Value Set)

- Strength Levels
  - If two signals of unequal strengths are driven on a wire, the stronger signal prevails.
    - If two signals of strength strong1 and weak0 contend, the result is resolved as a strong1.
    - If two signals of equal strengths are driven on a wire, the result is unknown.
    - If two signals of strength strong1 and strong0 conflict, the result is an x.
  - Strength levels are particularly useful for accurate modeling of signal contention, MOS devices, dynamic MOS, and other low-level devices.

Data Types (Nets)

- **Nets represent connections between hardware elements.**
- **Just as in real circuits, nets have values continuously driven on them by the outputs of devices that they are connected to.**

net a is connected to the output of and gate g1. Net a will continuously assume the value computed at the output of gate g1, which is b & c.



Data Types (Nets)

- Nets are declared primarily with the keyword wire.
- Nets are one-bit values by default unless they are declared explicitly as vectors.
- The terms wire and net are often used interchangeably.
- The default value of a net is z (except the trireg net, which defaults to x).
- Nets get the output value of their drivers. If a net has no driver, it gets the value z.
- Net is not a keyword but represents a class of data types such as wire, wand, wor, tri, triand, trior, trireg, etc.

```
wire a;          // Declare net a for the above circuit.
wire b,c;        // Declare two wires b,c for
                 // the above circuit.
wire d = 1'b0;   // Net d is fixed to logic value 0 at
                 //declaration.
```

Data Types (Registers)

- Registers represent data storage elements.
- Registers retain value until another value is placed onto them.
- Do not confuse the term registers in VerilogHDL with hardware registers built from edge-triggered flipflops in real circuits.
- In VerilogHDL, the term register merely means a variable that can hold a value.
- Unlike a net, a register does not need a driver.
- VerilogHDL registers do not need a clock as hardware registers do.
- Values of registers can be changed anytime in a simulation by assigning a new value to the register.

Data Types (Registers)

- Register data types are commonly declared by the keyword reg.
- The default value for a reg data type is x.

```
reg reset; //declare a variable reset that can hold its value
initial // this construct will be discussed later
begin
reset = 1'b1; //initialize reset to 1 to reset the digital circuit.
#100 reset = 1'b0; //after 100 time units reset is deasserted.
end
```

- Registers can also be declared as signed variables. Such registers can be used for signed arithmetic.

```
reg signed [63:0] m; //64 bit signed value
integer i; //32 bit signed value
```

Data Types (Vectors)

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

```
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
                         // Virtual address 41 bits wide
```

- Vectors can be declared at [high# : low#] or [low# : high#], but the left number in the squared brackets is always the most significant bit of the vector.
- In the example shown above, bit 0 is the most significant bit of vector virtual_addr.

Data Types (Vectors)

- **Vector Part Select - For the vector declarations shown above, it is possible to address bits or parts of vectors**

  busA[7] // bit # 7 of vector busA

  bus[2:0] // Three least significant bits of vector bus,

  // using bus[0:2] is illegal because the significant bit

  // should always be on the left of a range specification

  virtual_addr[0:1] // Two most significant bits of vector

- **Another ability provided in Verilog HDI is to have variable part selects of a vector. This allows part selects to be put in for loops to select various parts of the vector. There are two special part-select operators:**

  [<starting_bit>+:width] part-select increments from starting bit

  [<starting_bit>-:width] part-select decrements from starting bit

Data Types (Vectors)

- **The starting bit of the part select can be varied, but the width has to be constant.**

```
reg [255:0] data1; //Little endian notation
reg [0:255] data2; //Big endian notation
reg [7:0] byte;
//Using a variable part select, one can choose parts
byte = data1[31-:8];
    //starting bit = 31, width =8 => data[31:24]
byte = data1[24+:8];
    //starting bit = 24, width =8 => data[31:24]
byte = data2[31-:8];
    //starting bit = 31, width =8 => data[24:31]
byte = data2[24+:8];
    //starting bit = 24, width =8 => data[24:31]
```

Data Types (Vectors)

```
// The starting bit can also be a variable.
// The width has to be constant.
// Therefore, one can use the variable part select
// in a loop to select all bytes of the vector.
      for (j=0; j<=31; j=j+1)
            byte = data1[(j*8)+:8];
//Sequence is [7:0], [15:8]... [255:248]
/Can initialize a part of the vector
      data1[(byteNum*8)+:8] = 8'b0;
//If byteNum = 1, clear 8 bits [15:8]
```

Data Types (Integers)

- An integer is a general purpose register data type used for manipulating quantities.
- Integers are declared by the keyword integer. Although it is possible to use reg as a general-purpose variable, it is more convenient to declare an integer variable for purposes such as counting.
- The default width for an integer is the host-machine word size, which is implementation-specific but is at least 32 bits.
- Registers declared as data type reg 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
```

Data Types (Real)

- Real number constants and real register data types are declared with the keyword real.
- They can be specified in decimal notation (e.g., 3.14) or in scientific notation (e.g., 3e6, which is 3 x 10^6).
- Real numbers cannot have a range declaration, and their default value is 0.
- When a real value is assigned to an integer, the real number is rounded off to the nearest integer.

```
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)
```

Data Types (Arrays)

- Arrays are allowed in VerilogHDL for reg, integer, time, real, realtime and vector register data types.
- Multi-dimensional arrays can also be declared with any number of dimensions.
- Arrays of nets can also be used to connect ports of generated instances.
- Each element of the array can be used in the same fashion as a scalar or vector net.
- Arrays are accessed by

  <array_name>[<subscript>]

```
integer count[0:7];  // An array of 8 count variables reg
bool[31:0];
    // Array of 32 one-bit boolean register variables
time chk_point[1:100];
    // Array of 100 time checkpoint variables
```

Data Types (Arrays)

- For multi-dimensional arrays, indexes need to be provided for each dimension.

```
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
```

- It is important not to confuse arrays with net or register 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.
- The assignments to elements of arrays are as follows

```
count[5] = 0;
// Reset 5th element of array of count variables
matrix[1][0] = 33559;
// Set value of element indexed by [1][0] to 33559
```

Data Types (Memories)

- Memories are modeled in VerilogHDL simply as a one-dimensional array of registers.
  - Each element of the array is known as an element or word and is addressed by a single array index.
  - Each word can be one or more bits.
  - t is important to differentiate between n 1-bit registers and one n-bit register.
  - A particular word in memory is obtained by using the address as a memory array subscript.

```
reg mem1bit[0:1023];
    // Memory mem1bit with 1K 1-bit words reg [7:0]
membyte[0:1023];
    // Memory membyte with 1K 8-bit words(bytes)
membyte[511]
    // Fetches 1 byte word whose address is 511.
```

System Tasks

- VerilogHDL provides standard system tasks for certain routine operations.
- All system tasks appear in the form $<keyword>.
- Operations
  - Displaying on the screen
  - Monitoring values of nets
  - Stopping, and finishing in a simulation.

Displaying information

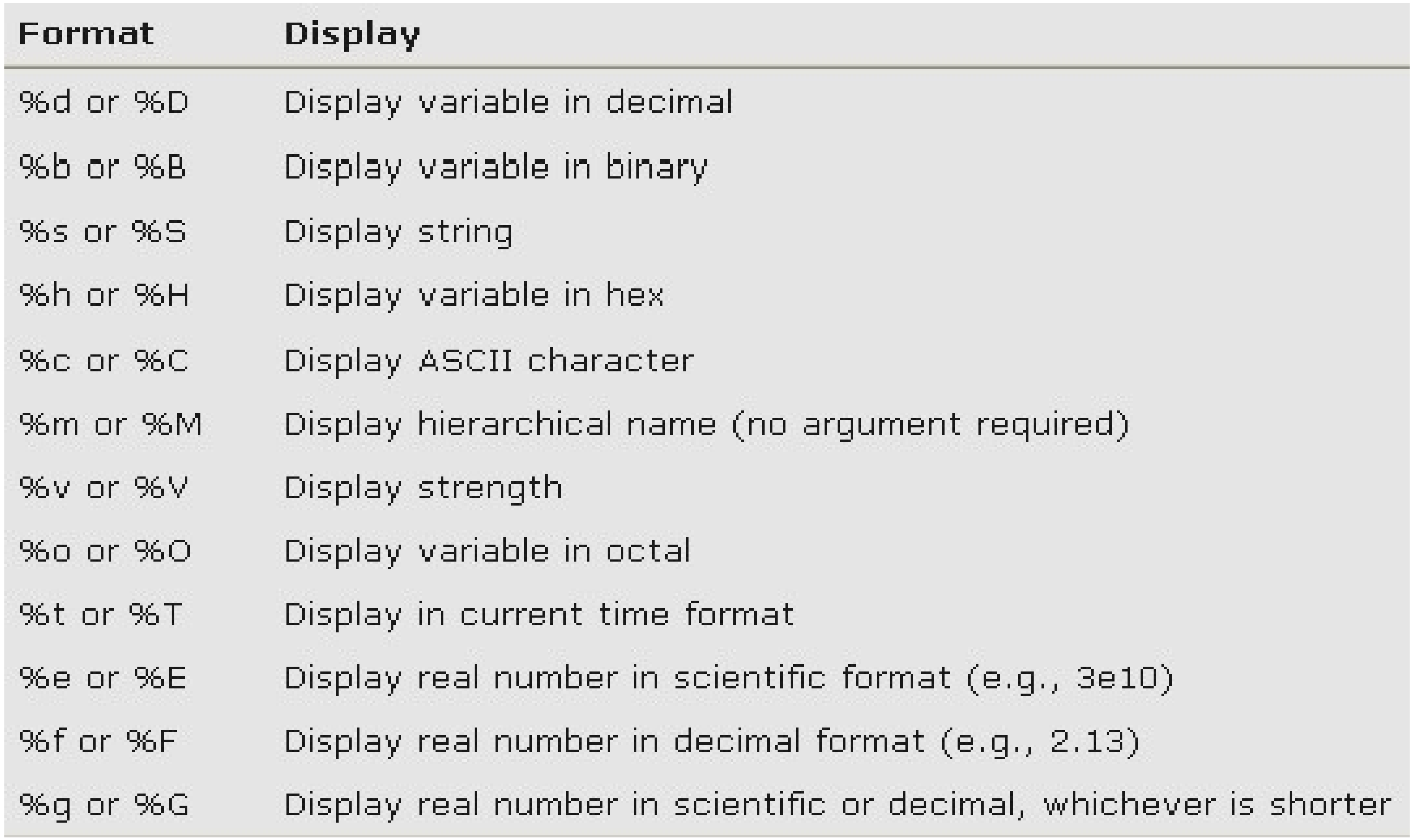
- $display is the main system task for displaying values of variables or strings or expressions.
- This is one of the most useful tasks in VerilogHDL.

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

- p1, p2, p3,..., pn can be quoted strings or variables or expressions.
- The format of $display is very similar to printf in C.
- A $display inserts a newline at the end of the string by default.
- A $display without any arguments produces a newline.

System Tasks

- **Strings can be formatted using the specifications as listed in the table.**



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

Monitoring information

- VerilogHDL provides a mechanism to monitor a signal when its value changes by using the task $monitor.

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

- The parameters p1, p2, ... , pn can be variables, signal names, or quoted strings.
- A format similar to the $display task is used in the $monitor task.
- $monitor continuously monitors the values of the variables or signals specified in the parameter list and displays all parameters in the list whenever the value of any one variable or signal changes.
- Unlike $display, $monitor needs to be invoked only once.

Monitoring information

- Only one monitoring list can be active at a time. If there is more than one $monitor statement in your simulation, the last $monitor statement will be the active statement. The earlier $monitor statements will be overridden.
- Two tasks are used to switch monitoring on and off.

```
$monitoron;      $monitoroff;
```

  - The $monitoron tasks enables monitoring.
  - The $monitoroff task disables monitoring during a simulation.
- Monitoring is turned on by default at the beginning of the simulation and can be controlled during the simulation with the $monitoron and $monitoroff tasks.

Stopping and finishing in a simulation

- The task $stop is provided to stop during a simulation.

```
$stop;
```

  - The $stop task puts the simulation in an interactive mode.
  - The designer can then debug the design from the interactive mode.
  - The $stop task is used whenever the designer wants to suspend the simulation and examine the values of signals in the design.
- The $finish task terminates the simulation.

```
$finish;
```

Compiler Directives

- Compiler directives are provided in Verilog. All compiler directives are defined by using the `<keyword> construct.
- `define - The `define directive is used to define text macros in VerilogHDL.
  - The VerilogHDL compiler substitutes the text of the macro wherever it encounters a `<macro_name>.
  - This is similar to the #define construct in C.
  - The defined constants or text macros are used in the VerilogHDL code by preceding them with a ` (back tick).
- `include - The `include directive allows you to include entire contents of a VerilogHDL source file in another VerilogHDL file during compilation.
  - This works similarly to the #include in C.
  - This directive is typically used to include header files, which typically contain global or commonly used definitions.

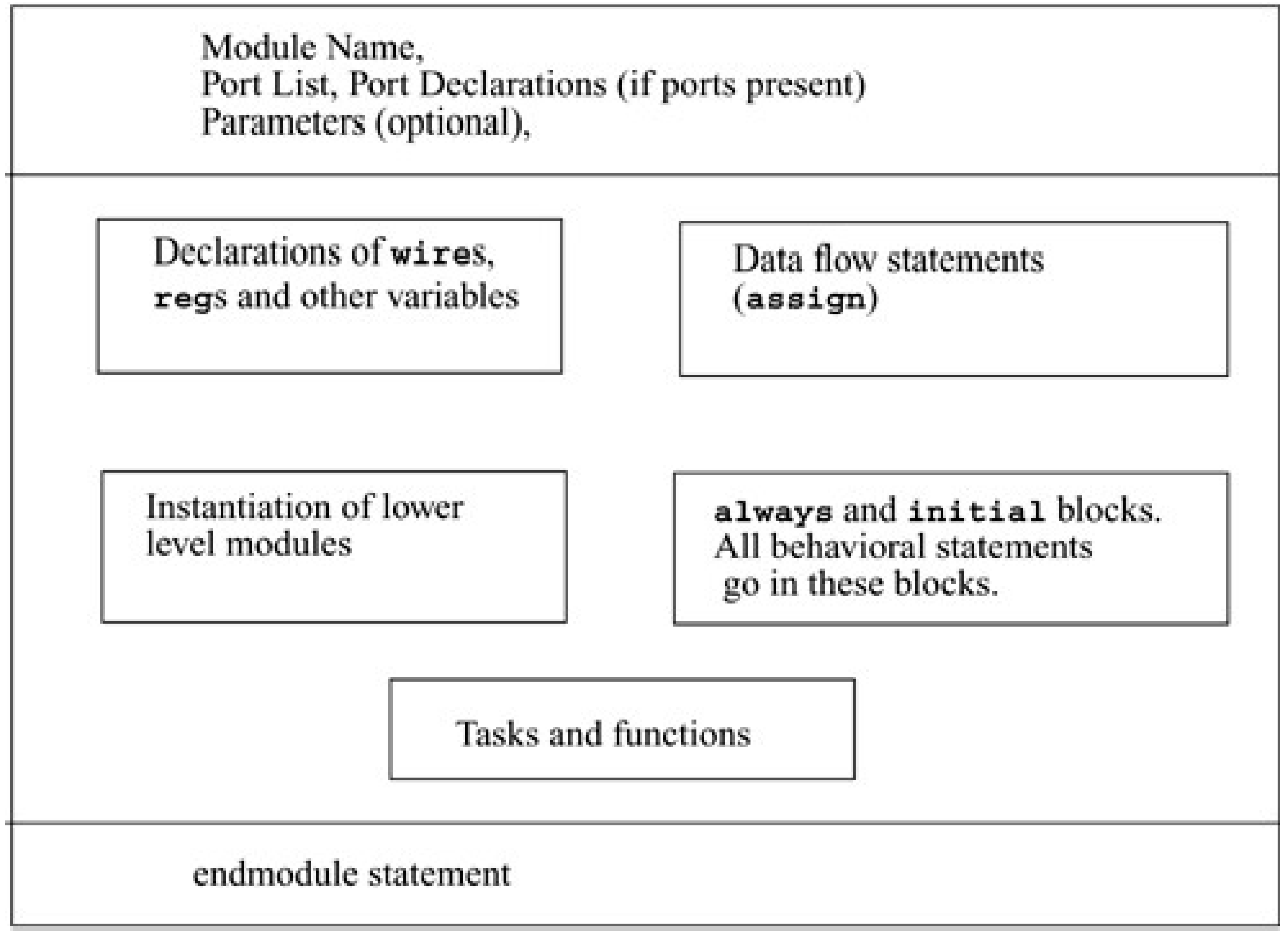
Modules and Ports

Modules

Ports

Modules

- **A module in VerilogHDL consists of distinct parts, as shown in figure.**


Module Name,
Port List, Port Declarations (if ports present)
Parameters (optional),
Declarations of wires, regs and other variables
Data flow statements (assign)
Instantiation of lower level modules
always and initial blocks. All behavioral statements go in these blocks.
Tasks and functions
endmodule statement

Modules

- A module definition always begins with the keyword module.
- The module name, port list, port declarations, and optional parameters must come first in a module definition.
- Port list and port declarations are present only if the module has any ports to interact with the external environment.
- The five components within a module are: variable declarations, dataflow statements, instantiation of lower modules, behavioral blocks, and tasks or functions.

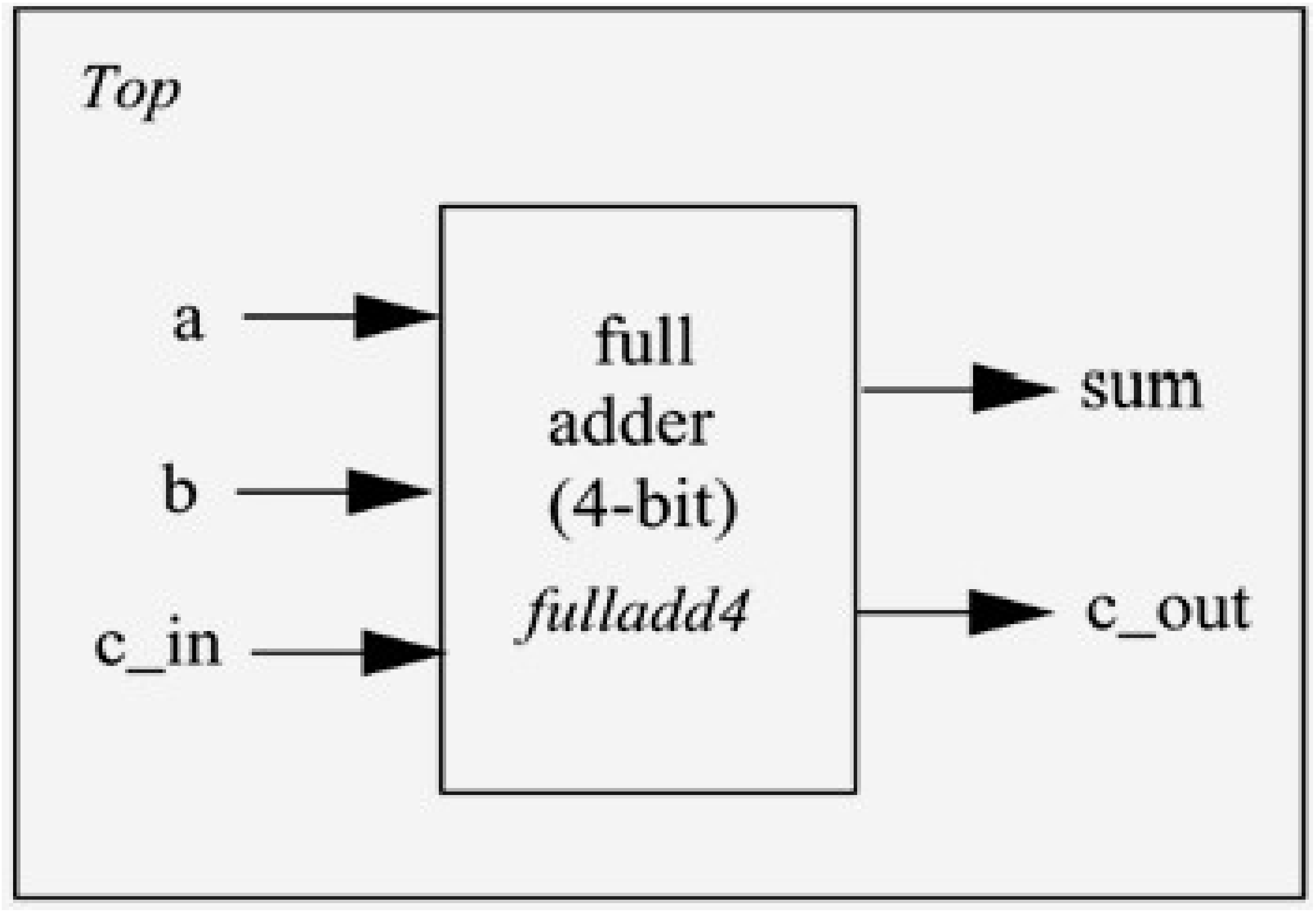
Modules

- These components can be in any order and at any place in the module definition.
- The endmodule statement must always come last in a module definition.
- All components except module, module name, and endmodule are optional and can be mixed and matched as per design needs.
- VerilogHDL allows multiple modules to be defined in a single file. The modules can be defined in any order in the file.

Ports

- Ports provide the interface by which a module can communicate with its environment.
  - For example, the input/output pins of an IC chip are its ports.
- The environment can interact with the module only through its ports.
  - The internals of the module are not visible to the environment. This provides a very powerful flexibility to the designer.
  - The internals of the module can be changed without affecting the environment as long as the interface is not modified.
- Ports are also referred to as terminals.
- A module definition contains an optional list of ports. If the module does not exchange any signals with the environment, there are no ports in the list.

I/O Ports for Top & Full Adder



I/O Ports for Top & Full Adder

- The module Top is a top-level module.
  - The module Top is a top-level module in the simulation and does not need to pass signals to or receive signals from the environment. Thus, it does not have a list of ports.

```
module Top; // No list of ports
```

- The module fulladd4 is instantiated below Top.
  - The module fulladd4 takes input on ports a, b, and c_in and produces an output on ports sum and c_out. Thus, module fulladd4 performs an addition for its environment.

```
module fulladd4(sum, c_out, a, b, c_in);
//Module with a list of ports
```

Port Declaration

- All ports in the list of ports must be declared in the module. Ports can be declared as shown in the table.
- Each port in the port list is defined as input, output, or inout, based on the direction of the port signal.

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

Port Declaration – Full Adder

```
module fulladd4(sum, c_out, a, b, c_in);
//Begin port declarations section
output[3:0] sum;
output c_cout;
input [3:0] a, b;
input c_in;
//End port declarations section
                ...
<module internals>
                 ...
 endmodule
```

Ports – Rules

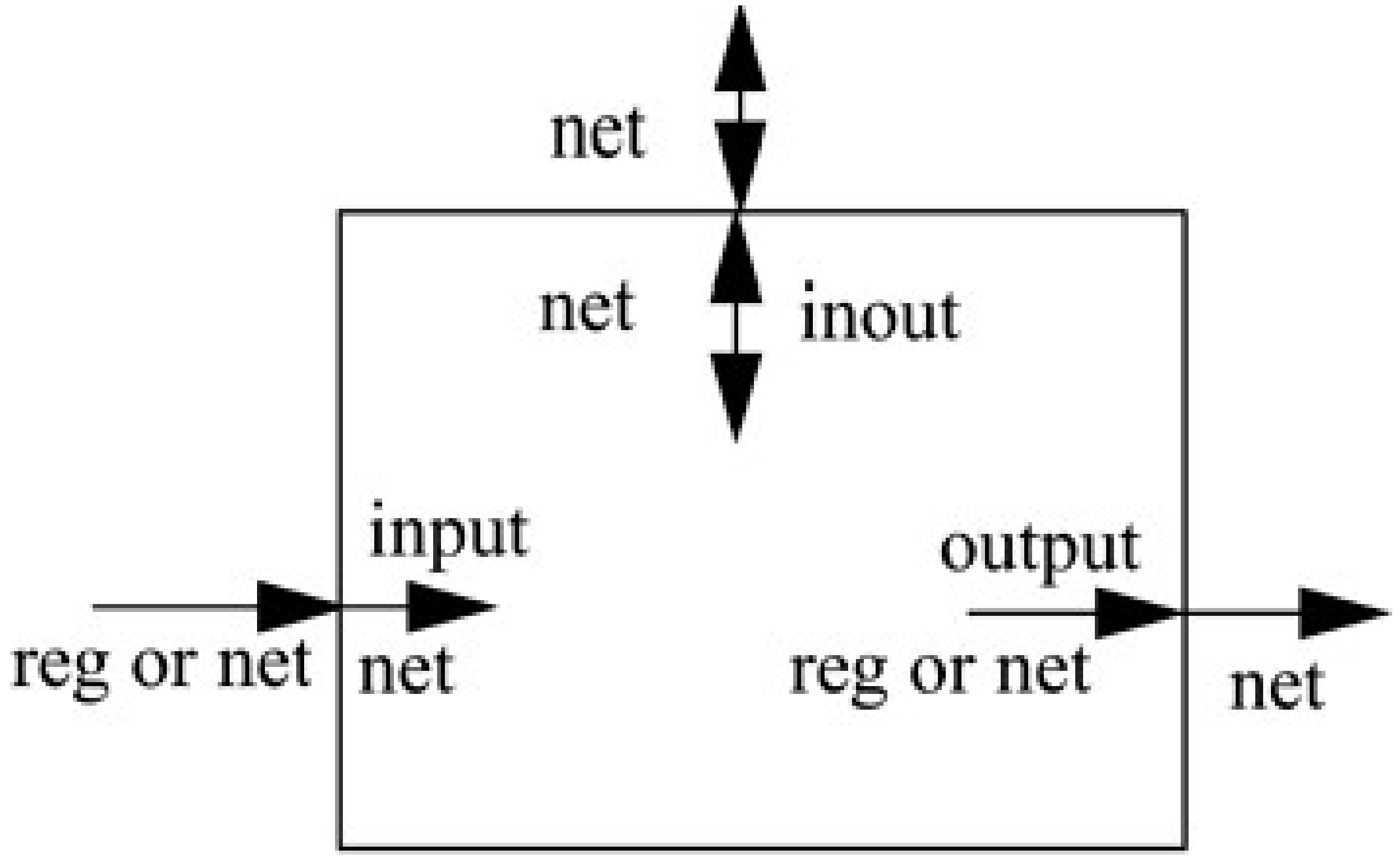
- All port declarations are implicitly declared as wire in VerilogHDL.
- Thus, if a port is intended to be a wire, it is sufficient to declare it as output, input, or inout.
- Input or inout ports are normally declared as wires.
- However, if output ports hold their value, they must be declared as reg.
- Ports of the type input and inout cannot be declared as reg because reg variables store values and input ports should not store values but simply reflect the changes in the external signals they are connected to.

Port Connection Rules

- A port is visualized as consisting of two units, one unit that is internal to the module and another that is external to the module.
- The internal and external units are connected.
- There are rules governing port connections when modules are instantiated within other modules.
- The VerilogHDL simulator complains if any port connection rules are violated.
- These rules are summarized in the table

Port Connection Rules

- These rules are summarized in the figure.



Port Connection Rules

- Inputs
  - Internally, input ports must always be of the type net.
  - Externally, the inputs can be connected to a variable which is a reg or a net.
- Outputs
  - Internally, outputs ports can be of the type reg or net.
  - Externally, outputs must always be connected to a net.
  - They cannot be connected to a reg.
- Inouts
  - Internally, inout ports must always be of the type net.
  - Externally, inout ports must always be connected to a net.

Port Connection Rules

- Width matching
  - It is legal to connect internal and external items of different sizes when making inter-module port connections. However, a warning is typically issued that the widths do not match.
- Unconnected ports
  - VerilogHDL allows ports to remain unconnected. For example, certain output ports might be simply for debugging, and you might not be interested in connecting them to the external signals.

```
fulladd4 fa0(SUM, , A, B, C_IN);
// Output port c_out is unconnected
```

Connecting Ports to External Signals

- There are two methods of making connections between signals specified in the module instantiation and the ports in a module definition.
  - Connecting by ordered list
  - Connecting ports by name
- These two methods cannot be mixed.

Connection by Ordered List

- Connecting by ordered list is the most intuitive method for most beginners.
- The signals to be connected must appear in the module instantiation in the same order as the ports in the port list in the module definition.
- Once again, consider the module fulladd4.
- To connect signals in module Top by ordered list, the Verilog code is shown in figure.
- Notice that the external signals SUM, C_OUT, A, B, and C_IN appear in exactly the same order as the ports sum, c_out, a, b, and c_in in module definition of fulladd4.

Connection by Ordered List

```
module fulladd4(sum, c_out, a, b, c_in);
output[3:0] sum;
output  c_cout;
input [3:0] a, b;
input c_in;

                ...
  <module internals>
                ...
endmodule
```

Connection by Names

- For large designs, remembering the order of the ports in the module definition is impractical and error-prone.
- VerilogHDL provides the capability to connect external signals to ports by the port names, rather than by position.
- The ports are connected by name as follows.
- Specify the port connections in any order as long as the port name in the module definition correctly matches the external signal.

```
// Instantiate module fa_byname
// Connect signals to ports by name fulladd4
      fa_byname(.c_out(C_OUT), .sum(SUM),
                          .b(B), .c_in(C_IN), .a(A),);
```

- Note that only those ports that are to be connected to external signals must be specified in port connection by name.

Connection by Names

Unconnected ports can be dropped.

- For example, if the port c_out were to be kept unconnected, the instantiation of fulladd4 would look as follows.
- The port c_out is simply dropped from the port list.

```
// Instantiate module fa_byname
// Connect signals to ports by name
fulladd4 fa_byname(.sum(SUM), .b(B), .c_in(C_IN), .a(A),);
```

- Another advantage of connecting ports by name is that as long as the port name is not changed, the order of ports in the port list of a module can be rearranged without changing the port connections in module instantiations.

Gate – Level Modeling

Gate Types

Gate Delays

Examples

Gate Types

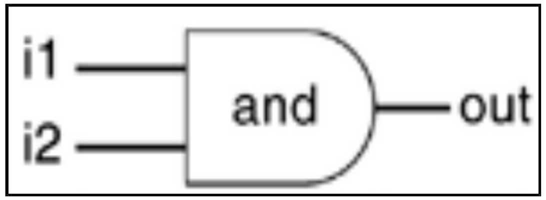
- A logic circuit can be designed by use of logic gates.
- VerilogHDL supports basic logic gates as predefined primitives.
- These primitives are instantiated like modules except that they are predefined in VerilogHDL and do not need a module definition.
- All logic circuits can be designed by using basic gates.
- There are two classes of basic gates
  - and/or gates
  - buf/not gates.

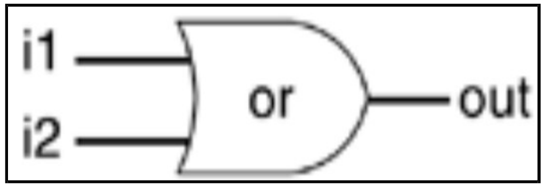
And/Or Gates

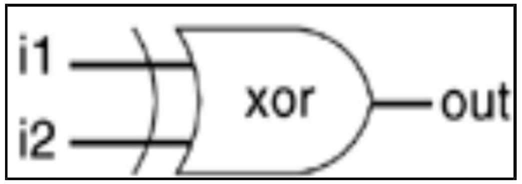
- And/or gates have one scalar output and multiple scalar inputs.
- The first terminal in the list of gate terminals is an output and the other terminals are inputs.
- The output of a gate is evaluated as soon as one of the inputs changes.
- The and/or gates available in VerilogHDL are shown below.

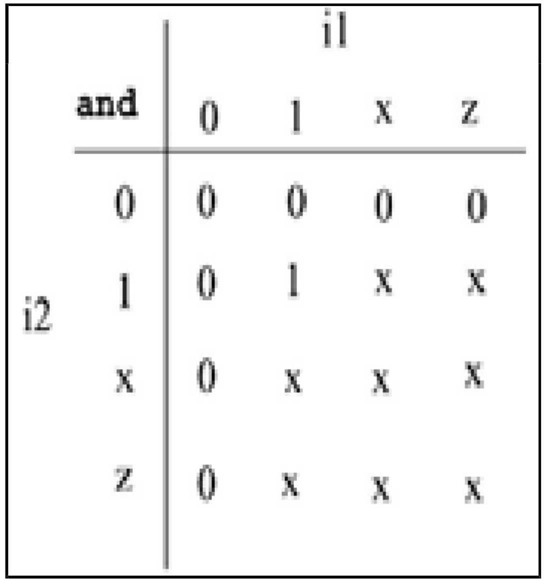
and or xor nand nor xnor

Truth Tables for And/Or Gates

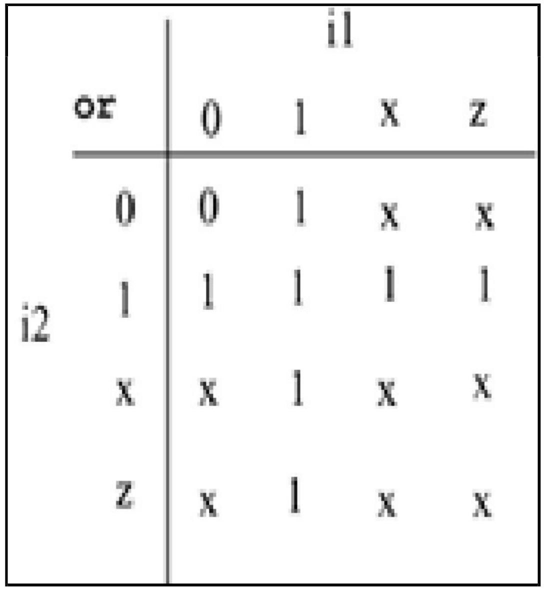




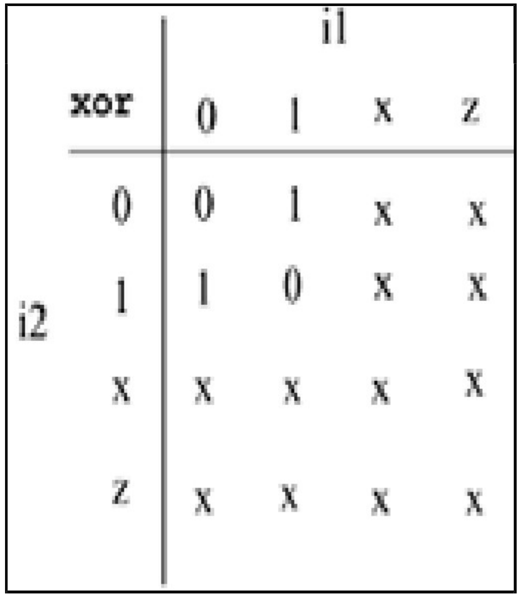




		i1			
	and	0	1	x	z
	0	0	0	0	0
i2	1	0	1	x	x
	x	0	x	x	x
	z	0	x	x	x

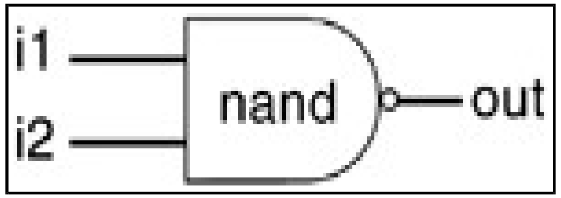


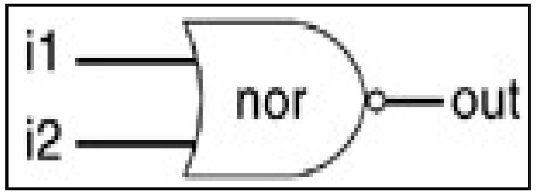
		i1			
	or	0	1	x	z
	0	0	1	x	x
i2	1	1	1	1	1
	x	x	1	x	x
	z	x	1	x	x

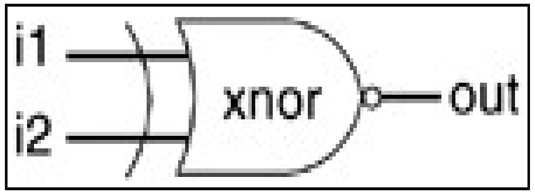


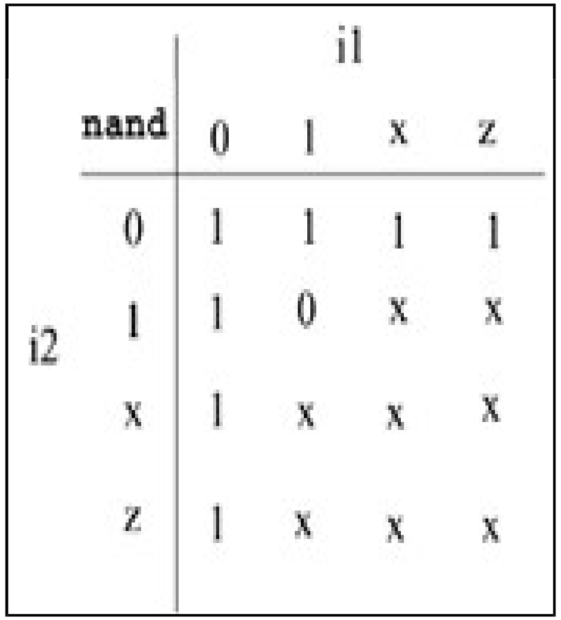
		i1			
	xor	0	1	x	z
	0	0	1	x	x
i2	1	1	0	x	x
	x	x	x	x	x
	z	x	x	x	x

Truth Tables for And/Or Gates

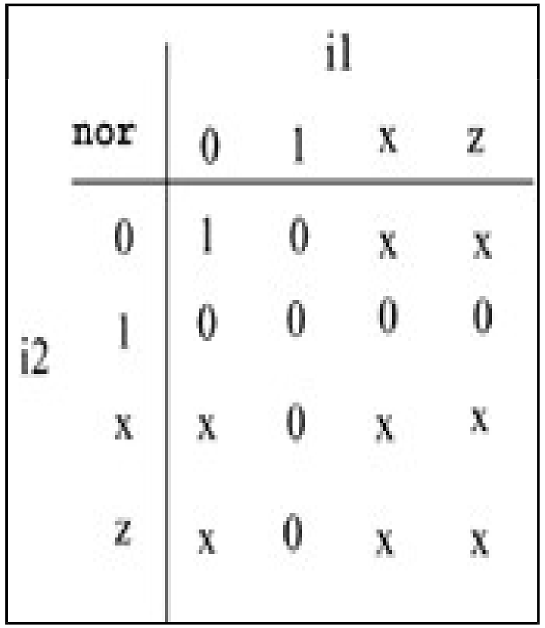




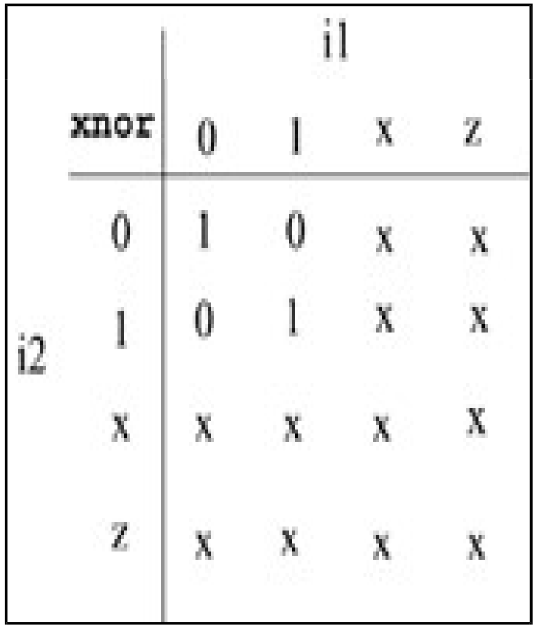




		i1			
	nand	0	1	x	z
	0	1	1	1	1
i2	1	1	0	x	x
	x	1	x	x	x
	z	1	x	x	x



		i1			
	nor	0	1	x	z
	0	1	0	x	x
i2	1	0	0	0	0
	x	x	0	x	x
	z	x	0	x	x



		i1			
	xnor	0	1	x	z
	0	1	0	x	x
i2	1	0	1	x	x
	x	x	x	x	x
	z	x	x	x	x

Gate Instantiation of And/Or

```
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); // legal gate instantiation
```

Buf/Not Gates

- Buf/not gates have one scalar input and one or more scalar outputs.
- The last terminal in the port list is connected to the input.
- Other terminals are connected to the outputs.
- These gates have one input and one output.
- Two basic buf/not gate primitives are provided in VerilogHDL.

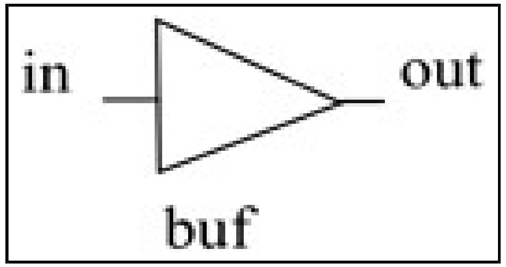
buf not

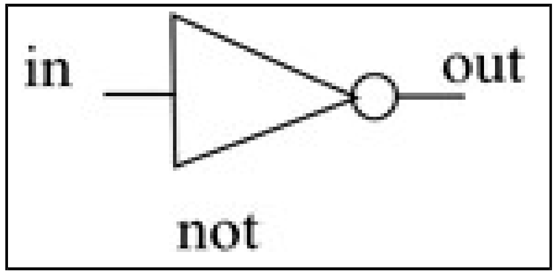
Buf/Not Gates

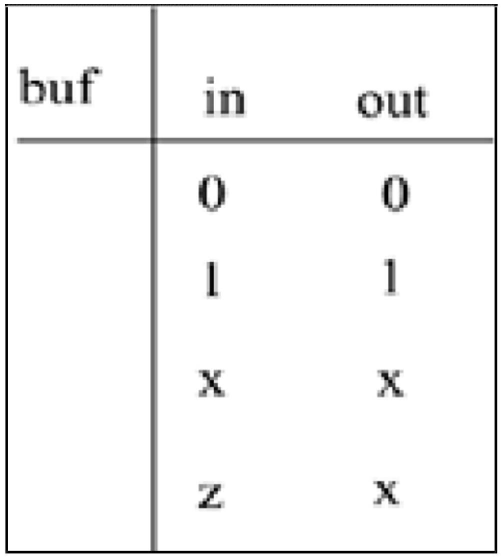
- Buf/not gates have one scalar input and one or more scalar outputs.
- The last terminal in the port list is connected to the input.
- Other terminals are connected to the outputs.
- These gates have multiple outputs but exactly one input, which is the last terminal in the port list.
- Two basic buf/not gate primitives are provided in VerilogHDL.

buf not

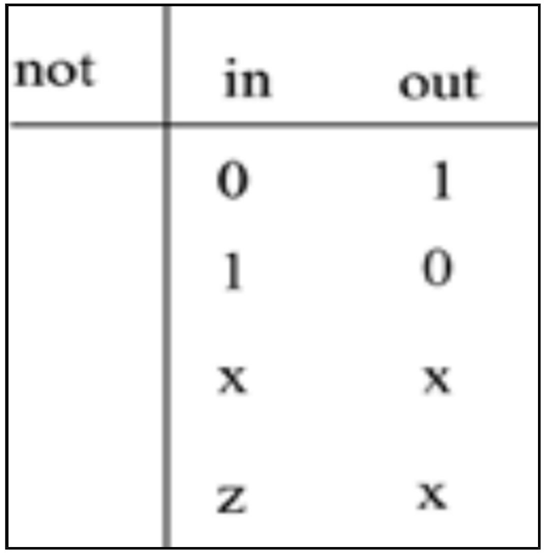
Truth Table - Buf/Not Gates







buf	in	out
	0	0
	1	1
	x	x
	z	x



not	in	out
	0	1
	1	0
	x	x
	z	x

Gate Instantiation of Buf/Not

```
// basic gate instantiations.
buf b1(OUT1, IN);
not n1(OUT1, IN);
// More than two outputs
buf b1_2out(OUT1, OUT2, IN);
// gate instantiation without instance name
not (OUT1, IN); // legal gate instantiation
```

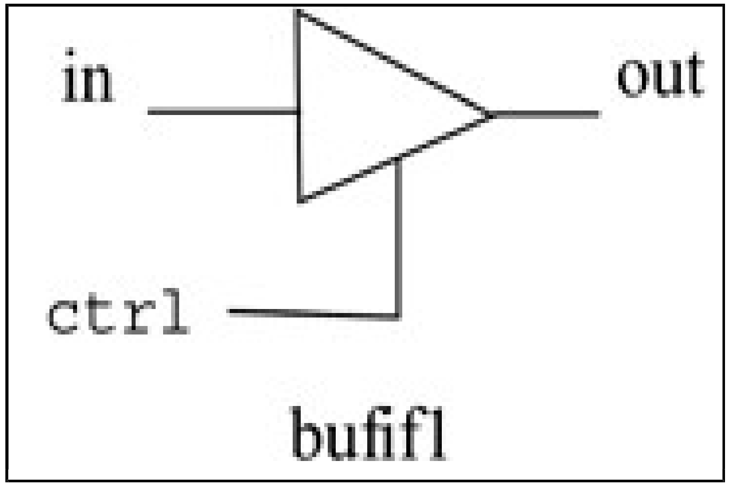
Bufif/notif Gates

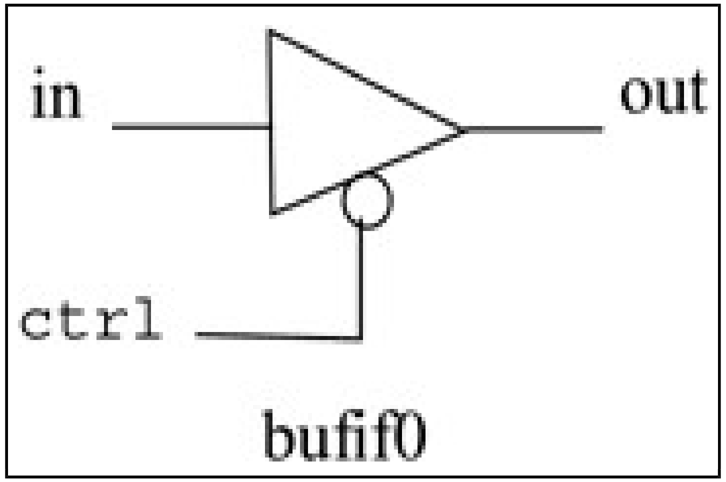
- Gates with an additional control signal on buf and not gates are also available.

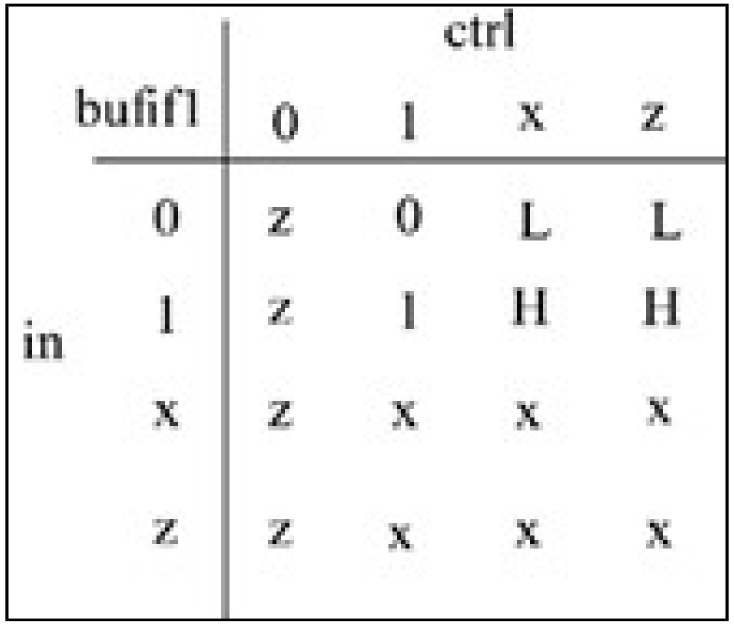
  bufif1 notif1 bufif0 notif0

- These gates propagate only if their control signal is asserted. They propagate z if their control signal is deasserted.
- These gates are used when a signal is to be driven only when the control signal is asserted. Such a situation is applicable when multiple drivers drive the signal.
- These drivers are designed to drive the signal on mutually exclusive control signals.

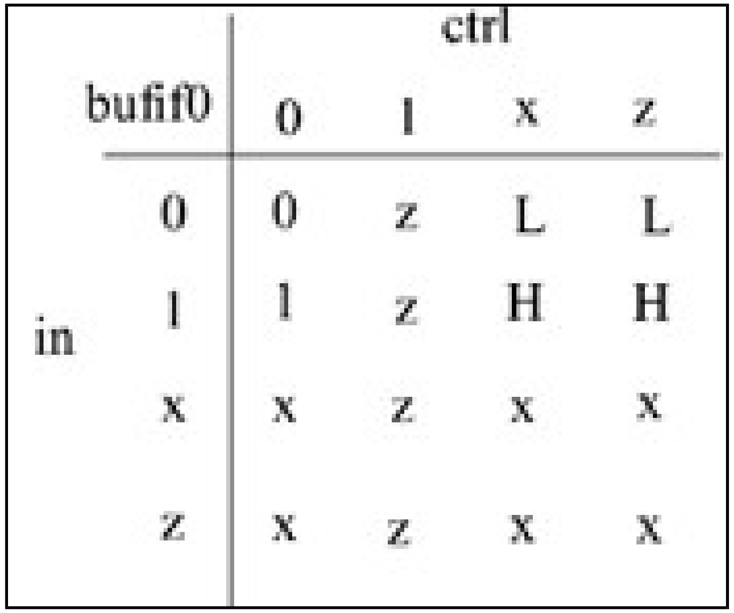
Truth table – Bufif Gates





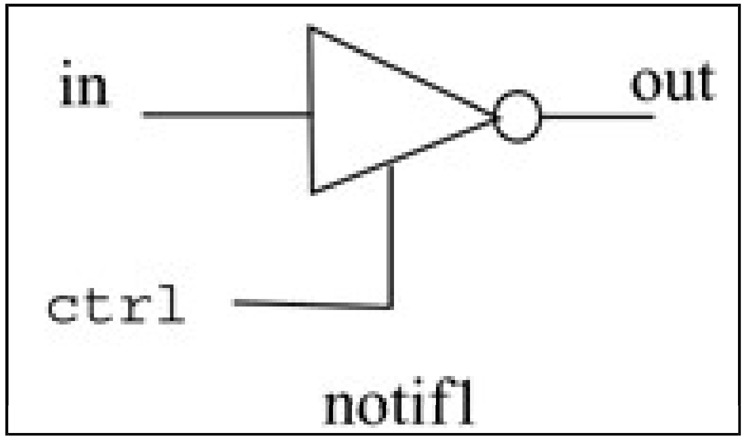


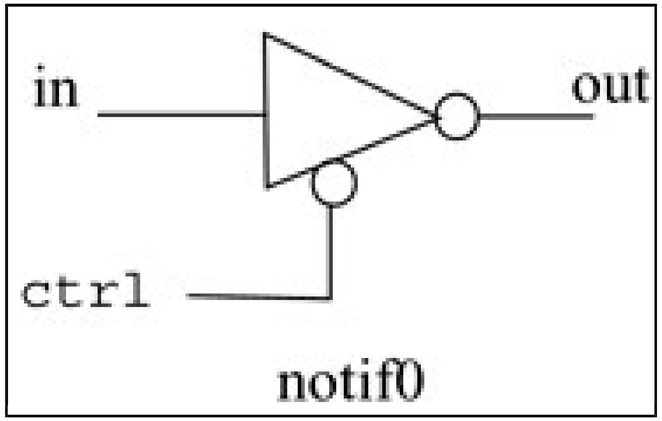
		ctrl			
	bufif1	0	1	x	z
in	0	z	0	L	L
	1	z	1	H	H
	x	z	x	x	x
	z	z	x	x	x

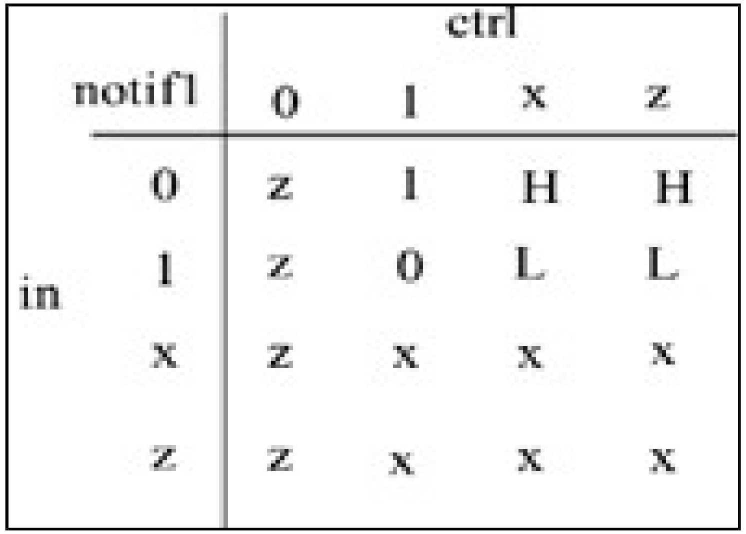


		ctrl			
	bufif0	0	1	x	z
in	0	0	z	L	L
	1	1	z	H	H
	x	x	z	x	x
	z	x	z	x	x

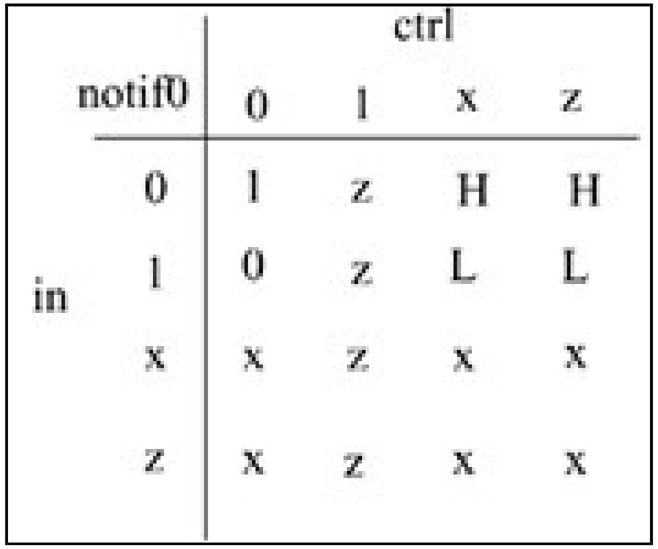
Truth table - notif Gates







	notif1	ctrl 0	1	x	z
in	0	z	1	H	H
	1	z	0	L	L
	x	z	x	x	x
	z	z	x	x	x



	notif0	ctrl 0	1	x	z
in	0	1	z	H	H
	1	0	z	L	L
	x	x	z	x	x
	z	x	z	x	x

Gate Instantiation of Bufif/Notif

```
//Instantiation of bufif gates.
   bufif1 b1 (out, in, ctrl);
   bufif0 b0 (out, in, ctrl);
//Instantiation of notif gates
   notif1 n1 (out, in, ctrl);
   notif0 n0 (out, in, ctrl);
```

Array of Instances

- There are many situations when repetitive instances are required.
- These instances differ from each other only by the index of the vector to which they are connected.
- To simplify specification of such instances, VerilogHDL allows an array of primitive instances to be defined

Array of Instances

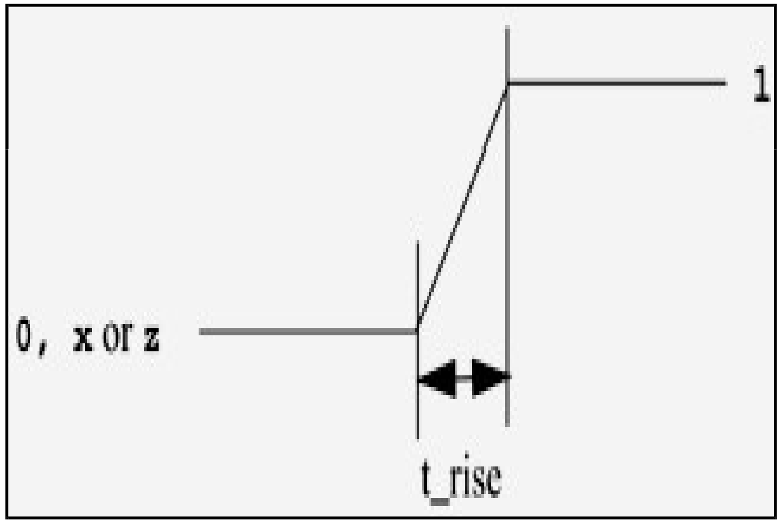
```
    wire [7:0] OUT, IN1, IN2;
// basic gate instantiations.
    nand n_gate[7:0](OUT, IN1, IN2);
 // This is equivalent to the following 8 instantiations
    nand n_gate0(OUT[0], IN1[0], IN2[0]);
    nand n_gate1(OUT[1], IN1[1], IN2[1]);
    nand n_gate2(OUT[2], IN1[2], IN2[2]);
    nand n_gate3(OUT[3], IN1[3], IN2[3]);
    nand n_gate4(OUT[4], IN1[4], IN2[4]);
    nand n_gate5(OUT[5], IN1[5], IN2[5]);
    nand n_gate6(OUT[6], IN1[6], IN2[6]);
    nand n_gate7(OUT[7], IN1[7], IN2[7]);
```

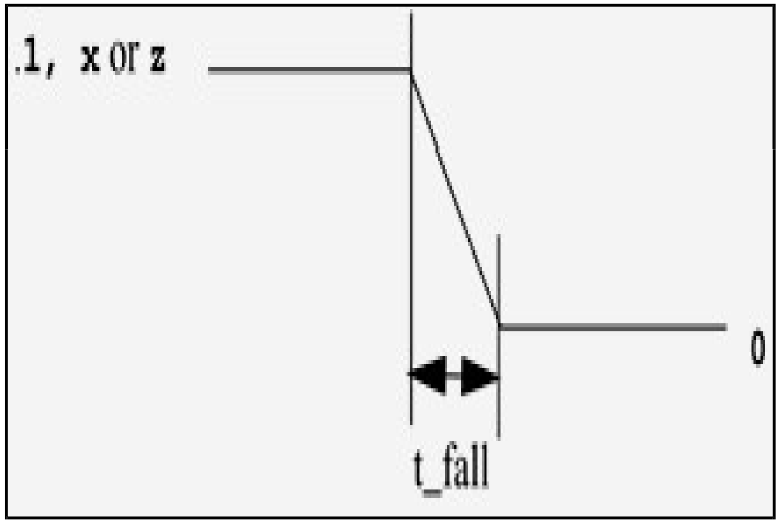
Gate Delays

- In real circuits, logic gates have delays associated with them.
- Gate delays allow the VerilogHDL user to specify delays through the logic circuits.
- Pin-to-pin delays can also be specified in VerilogHDL.
- There are three types of delays from the inputs to the output of a primitive gate.
  - Rise Delay
  - Fall Delay
  - Turnoff Delay

Gate Delays

- **Rise Delay - Delay associated with a gate output transition to a 1 from another value.**
- **Fall Delay - Delay associated with a gate output transition to a 0 from another value.**





- **Turn Off Delay - Delay associated with a gate output transition to the high impedance value (z) from another value.**

Gate Delays

- If the value changes to x, the minimum of the three delays is considered.
- Three types of delay specifications are allowed.
- If only one delay is specified, this value is used for all transitions.

```
// Delay of delay_time for all transitions
      and #(delay_time) a1(out, i1, i2);
// Delay of 5 for all transitions
       and #(5) a1(out, i1, i2);
```

Gate Delays

- If two delays are specified, they refer to the rise and fall delay values. The turn-off delay is the minimum of the two delays.

```
// Rise and Fall Delay Specification.
        and #(rise_val, fall_val) a2(out, i1, i2);
// Rise = 4, Fall = 6
         and #(4,6) a2(out, i1, i2);
```

- If all three delays are specified, they refer to rise, fall, and turn-off delay values.

```
// Rise, Fall, and Turn-off Delay Specification
        bufif0 #(rise_val, fall_val, turnoff_val) b1 (out, in, control);
// Rise = 3, Fall = 4, Turn-off = 5
         bufif0 #(3,4,5) b1 (out, in, control);
```

- If no delays are specified, the default value is zero.

Min/Typ/Max Values

- VerilogHDL provides an additional level of control for each type of delay mentioned above.
- For each type of delay—rise, fall, and turn-off—three values, min, typ, and max, can be specified.
- Any one value can be chosen at the start of the simulation.
- Min/typ/max values are used to model devices whose delays vary within a minimum and maximum range because of the IC fabrication process variations.

Min/Typ/Max Values

- Min value
  - The min value is the minimum delay value that the designer expects the gate to have.
- Typ val
  - The typ value is the typical delay value that the designer expects the gate to have.
- Max value
  - The max value is the maximum delay value that the designer expects the gate to have.
- Min, typ, or max values can be chosen at Verilog run time.

Min/Typ/Max Values

- Method of choosing a min/typ/max value may vary for different simulators or operating systems.
- The values are chosen by specifying options +maxdelays, +typdelays, and +mindelays at run time.
- If no option is specified, the typical delay value is the default).
- This allows the designers the flexibility of building three delay values for each transition into their design.
- The designer can experiment with delay values without modifying the design.

Min/Typ/Max Values

```
// One delay
// if +mindelays, delay= 4
// if +typdelays, delay= 5 /
/ if +maxdelays, delay= 6
      and #(4:5:6) a1(out, i1, i2);
// Two delays
// if +mindelays, rise= 3, fall= 5, turn-off = min(3,5)
// if +typdelays, rise= 4, fall= 6, turn-off = min(4,6)
// if +maxdelays, rise= 5, fall= 7, turn-off = min(5,7)        and
#(3:4:5, 5:6:7) a2(out, i1, i2);
// Three delays
// if +mindelays, rise= 2 fall= 3 turn-off = 4
// if +typdelays, rise= 3 fall= 4 turn-off = 5
// if +maxdelays, rise= 4 fall= 5 turn-off = 6
      and #(2:3:4, 3:4:5, 4:5:6) a3(out, i1,i2);
```

Min/Typ/Max Values

- **Examples of invoking the Verilog-XL simulator with the command-line options are shown below.**
- **Assume that the module with delays is declared in the file test.v.**

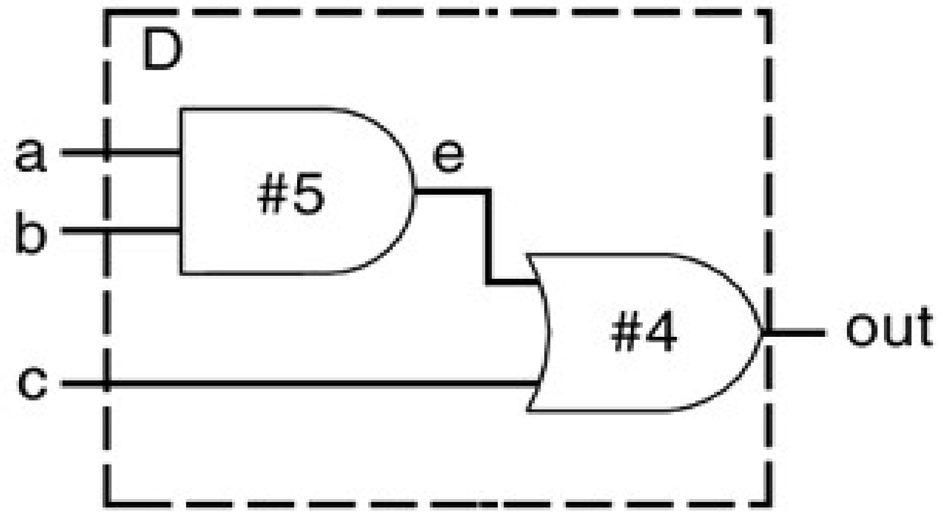
```
//invoke simulation with maximum delay
> verilog test.v +maxdelays
//invoke simulation with minimum delay
> verilog test.v +mindelays
//invoke simulation with typical delay
> verilog test.v +typdelays
```

Delay Example

- Let us consider a simple example to illustrate the use of gate delays to model timing in the logic circuits.
- A simple module called D implements the following logic equations:

  out = (a b) + c

- The gate-level implementation is shown in Module D.



Delay Example

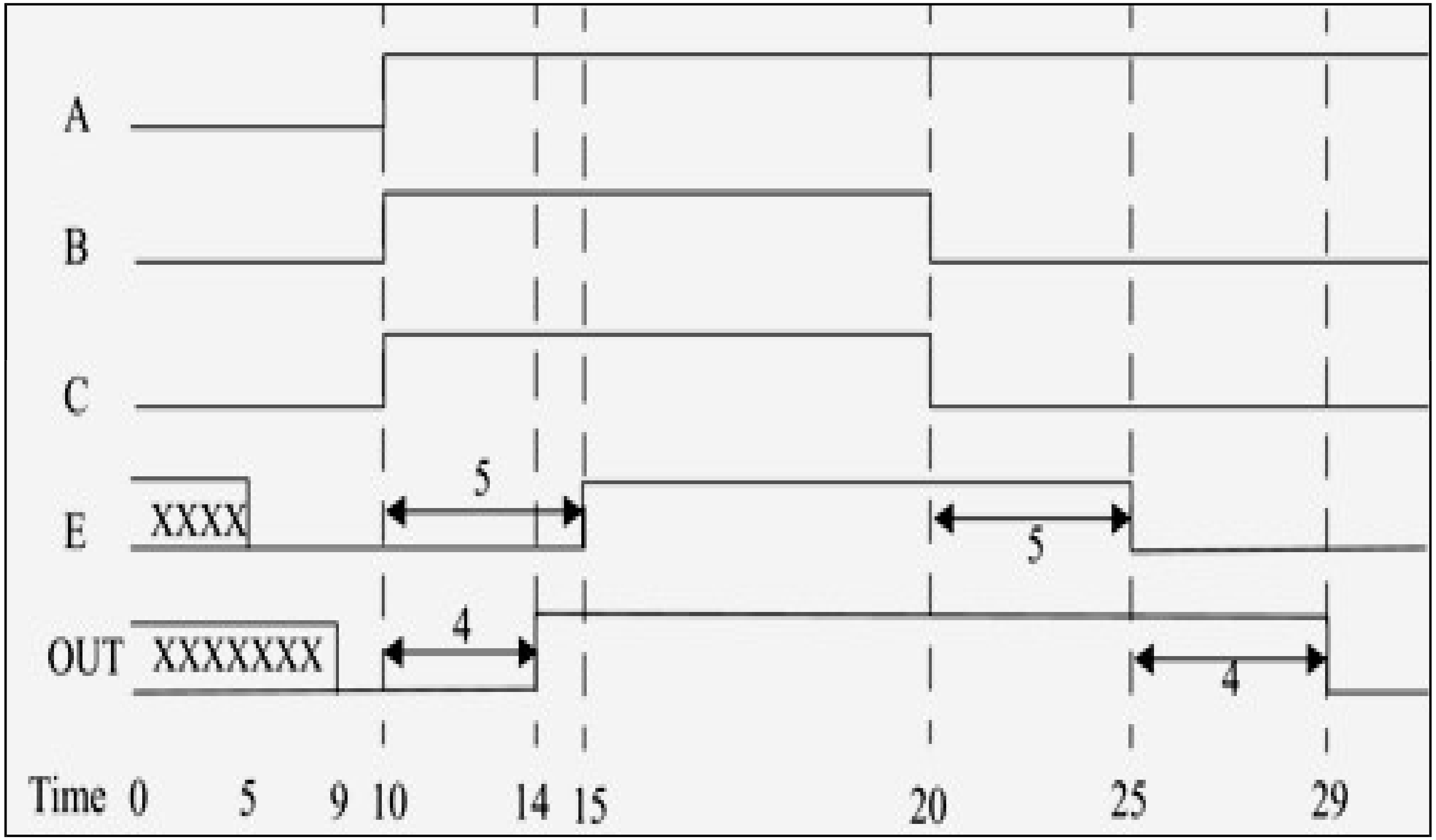
- The module contains two gates with delays of 5 and 4 time units.The module D is defined in VerilogHDL as follows.

```
// Define a simple combination module called D      module D
(out, a, b, c);
// I/O port declarations
        output out;
        input a,b,c;
// Internal nets
        wire e;
// Instantiate primitive gates to build the circuit
        and #(5) a1(e, a, b);
//Delay of 5 on gate a1
        or #(4) o1(out, e,c);
//Delay of 4 on gate o1 endmodule
```

Stimulus for Module D with Delay

```
        module stimulus; // Stimulus (top-level module)
        reg A, B, C; wire OUT; // Declare variables
        D d1( OUT, A, B, C); // Instantiate the module D
  // Stimulate the inputs.
  // Finish the simulation at 40 time units.
        initial
            begin
                    A= 1'b0; B= 1'b0; C= 1'b0;
                #10 A= 1'b1; B= 1'b1; C= 1'b1;
                #10 A= 1'b1; B= 1'b0; C= 1'b0;
                #20 $finish;
            end
        endmodule
```

Waveforms for Delay Simulation

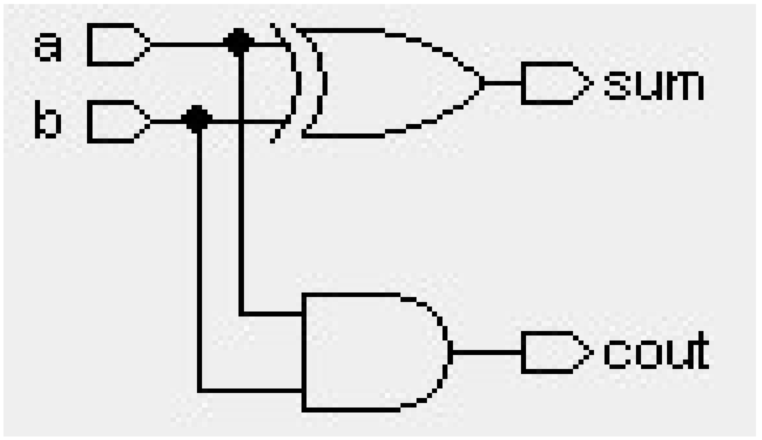


Waveforms for Delay Simulation

- The waveforms from the simulation are shown in figure.
  - The outputs E and OUT are initially unknown.
  - At time 10, after A, B, and C all transition to 1, OUT transitions to 1 after a delay of 4 time units and E changes value to 1 after 5 time units.
  - At time 20, B and C transition to 0. E changes value to 0 after 5 time units, and OUT transitions to 0, 4 time units after E changes.

Half Adder

```
module halfadder(cout,sum,a,b);
   input a,b;
   output cout,sum;
   xor (sum,a,b);
   and (cout,a,b);
endmodule
```



Stimulus File for HA

```
Module stimulus;
   reg a,b;
   wire cout,sum;
   halfadder h1(cout, sum,a,b);
   initial
       begin
         $monitor($time,"a=%b, b=%b,
        sum=%b, cout=%b", a, b, sum, cout);
       end
```

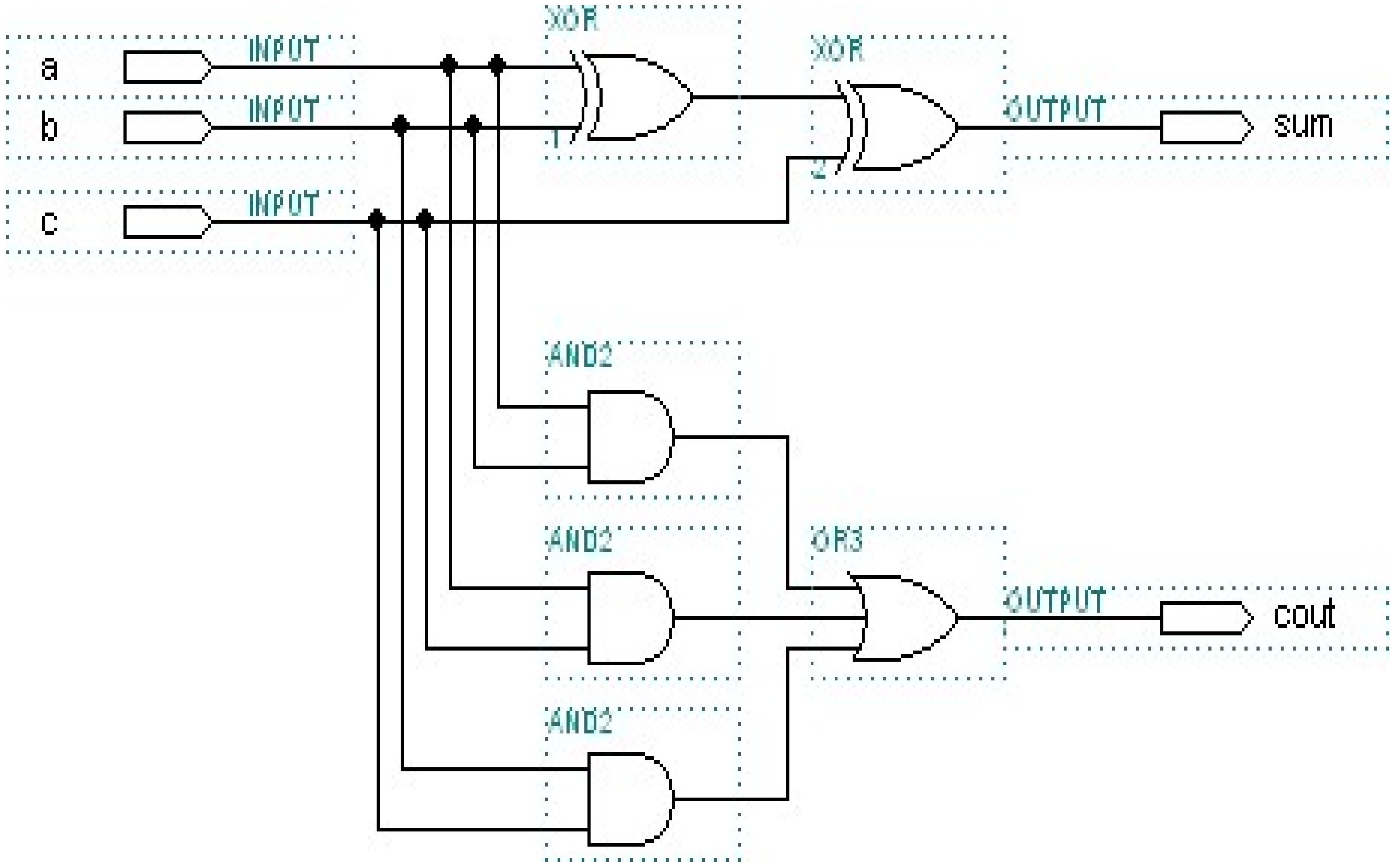
Stimulus File for HA

```
initial
   begin
     a=1'b0;b=1'b0;
          #5 a=1'b0;b=1'b1;
          #5 a=1'b1;b=1'b0;
          #5 a=1'b1;b=1'b1;
    end
endmodule
```

Simulation Results for HA

```
#0      a=0, b=0, sum=0, cout=0
# 5     a=0, b=1, sum=1, cout=0
# 10    a=1, b=0, sum=1, cout=0
# 15    a=1, b=1, sum=0, cout=1
```

Full Adder


a
b
c
INPUT
INPUT
INPUT
XOR
1
XOR
2
OUTPUT
sum
AND2
AND2
AND2
OR3
OUTPUT
cout

Full Adder

```
// Define a 1-bit full adder
    module fulladd(sum, cout, a, b, c);
// I/O port declarations
          output sum, cout;
          input a, b, c;
// Internal nets
          wire s1, c1, c2, c3;
// Instantiate logic gate primitives
          xor x1(s1, a, b);
          xor x2(sum, s1, c);
          and a1(c1, a, b);
          and a2(c2, a, c);
          and a3(c3, b, c);
          or (cout, c1, c2, c3);
    endmodule
```

Stimulus File for FA

```
module fulladdstim;
    reg a,b,c;
    wire sum,cout;
    fulladder fa(sum,cout,a,b,c);
    initial
    begin
        $monitor($time, "a=%b, b=%b, c=%b,sum=%b, cout=%b",a, b, c,
sum, cout);
    end
```

Stimulus File for FA

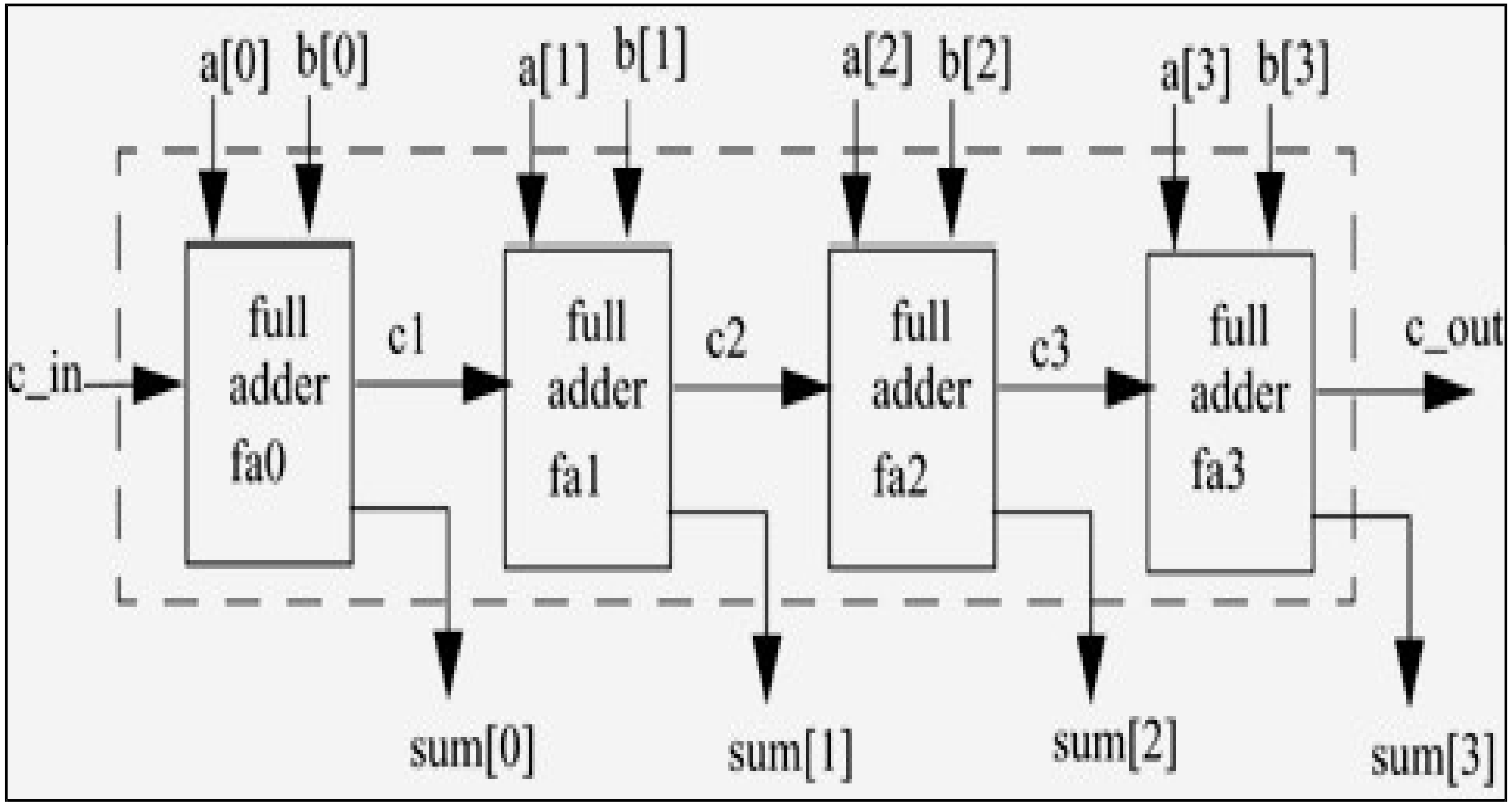
```
    initial
      begin
             a=1'b0;b=1'b0;c=1'b0;
          #5 a=1'b0;b=1'b0;c=1'b1;
          #5 a=1'b0;b=1'b1;c=1'b0;
          #5 a=1'b0;b=1'b1;c=1'b1;
          #5 a=1'b1;b=1'b0;c=1'b0;
          #5 a=1'b1;b=1'b0;c=1'b1;
          #5 a=1'b1;b=1'b1;c=1'b0;
          #5 a=1'b1;b=1'b1;c=1'b1;
    end
endmodule
```

Simulation Results for FA

```
#0      a=0, b=0, c=0, sum=0, cout=0
#5      a=0, b=0, c=1, sum=1, cout=0
#10     a=0, b=1, c=0, sum=1, cout=0
#15     a=0, b=1, c=1, sum=0, cout=1
#20     a=1, b=0, c=0, sum=1, cout=0
#25     a=1, b=0, c=1, sum=0, cout=1
#30     a=1, b=1, c=0, sum=0, cout=1
#35     a=1, b=1, c=1, sum=1, cout=1
```

Ripple Carry Adder(4-bit)

- **A 4-bit ripple carry full adder can be constructed from four 1-bit full adders, as shown in figure.**



Ripple Carry Adder(4-bit)

```
// Define a 4-bit full adder
    module ripple4(sum, c_out, a, b, c_in);
// I/O port declarations
         output [3:0] sum;
         output c_out;
         input [3:0] a, b;
         input c_in;
// Internal nets
         wire c1, c2, c3;
// Instantiate four 1-bit full adders.
         fulladd fa0(sum[0], c1, a[0], b[0], c_in);
         fulladd fa1(sum[1], c2, a[1], b[1], c1);
         fulladd fa2(sum[2], c3, a[2], b[2], c2);
         fulladd fa3(sum[3], c_out, a[3], b[3], c3);
endmodule
```

Stimulus for RCA

```
// Define the stimulus (top level module)
    module stimulus;
// Set up variables
        reg [3:0] A, B;
        reg C_IN;
        wire [3:0] SUM;
        wire C_OUT;
// Instantiate the 4-bit full adder. call it FA1_4
        ripple4 FA1_4(SUM, C_OUT, A, B, C_IN);
// Set up the monitoring for the signal values
        initial
            begin
            $monitor($time," A= %b, B=%b, C_IN= %b, C_OUT= %b, SUM=
    %b\n“, A, B, C_IN, C_OUT, SUM);
            end
```

Stimulus for RCA

```
// Stimulate inputs
    initial
      begin
              A = 4'd0; B = 4'd0; C_IN = 1'b0;
        #5  A = 4'd3; B = 4'd4;
        #5  A = 4'd2; B = 4'd5;
        #5  A = 4'd9; B = 4'd9;
        #5  A = 4'd10; B = 4'd15;
        #5  A = 4'd10; B = 4'd5; C_IN = 1'b1;
      end
  endmodule
```

Stimulus for RCA

The output of the simulation is shown below.

```
 0   A= 0000, B=0000, C_IN= 0, --- C_OUT= 0, SUM= 0000
 5   A= 0011, B=0100, C_IN= 0, --- C_OUT= 0, SUM= 0111
10  A= 0010, B=0101, C_IN= 0, --- C_OUT= 0, SUM= 0111
15  A= 1001, B=1001, C_IN= 0, --- C_OUT= 1, SUM= 0010
20  A= 1010, B=1111, C_IN= 0, --- C_OUT= 1, SUM= 1001
25  A= 1010, B=0101, C_IN= 1, --- C_OUT= 1, SUM= 0000
```

Ripple Carry Adder(8-bit)

```
    module ripple8(sum, c_out, a, b, c_in);
// I/O port declarations
        output [7:0] sum;
        output c_out;
        input [7:0] a, b;
        input c_in;
// Internal nets
        wire c1, c2, c3, c4, c5, c6, c7;
```

Ripple Carry Adder(8-bit)

```
// Instantiate eight 1-bit full adders.
        fulladd fa0(sum[0], c1, a[0], b[0], c_in);
        fulladd fa1(sum[1], c2, a[1], b[1], c1);
        fulladd fa2(sum[2], c3, a[2], b[2], c2);
        fulladd fa3(sum[3], c4, a[3], b[3], c3);
        fulladd fa4(sum[4], c5, a[4], b[4], c4);
        fulladd fa5(sum[5], c6, a[5], b[5], c5);
        fulladd fa6(sum[6], c7, a[6], b[6], c6);
        fulladd fa7(sum[7], c_out, a[7], b[7], c7);
 endmodule
```

Half Subtractor

```
module halfsubtractor(d, b, x, y);
    input x,y;
    output d,b;
    wire i1;
    xor (d,x,y);
    not (i1,x);
    and (b,i1,y);
endmodule
```

Stimulus File for HS

```
module halfsubstim;
    reg x,y;
    wire d,b;
    halfsubtractor h1(d, b,x,y);
    initial
      begin
            $monitor($time,"a=%b, b=%b, diff=%b,bor=%b", x, y, d, b);
      end
```

Stimulus File for HS

```
initial
   begin
     x=1'b0;y=1'b0;
          #5 x=1'b0;y=1'b1;
          #5 x=1'b1;y=1'b0;
          #5 x=1'b1;y=1'b1;
    end
endmodule
```

Simulation Results for HS

```
#0      x=0, y=0, diff=0, bor=0
# 5     x=0, y=1, diff=1, bor=1
# 10    x=1, y=0, diff=1, bor=0
# 15    x=1, y=1, diff=0, bor=0
```

Full Subtractor

```
module fullsub(d,b,x,y,z);
    output d,b;
    input x,y,z;
    wire x1,a1,i1,o1,a2;
    xor (x1,x,y);
    xor (d,x1,z);
    and (a1,y,z);
    not (i1,x);
    or (o1,y,z);
    and (a2,i1,o1);
    or (b,a2,a1);
endmodule
```

Stimulus File for FS

```
module fullsubstim;
    reg x,y,z;
    wire d,b;
    fullsub fa(d,b,x,y,z);
    initial
    begin
        $monitor($time, "x=%b, y=%b, z=%b,d=%b, b=%b",x, y, z, d, b);
    end
```

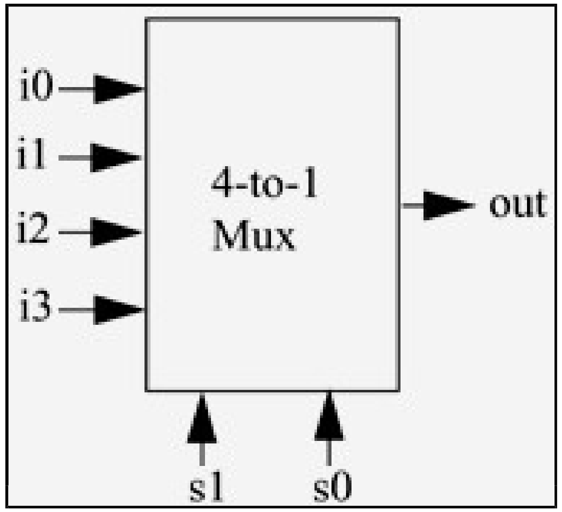
Stimulus File for FS

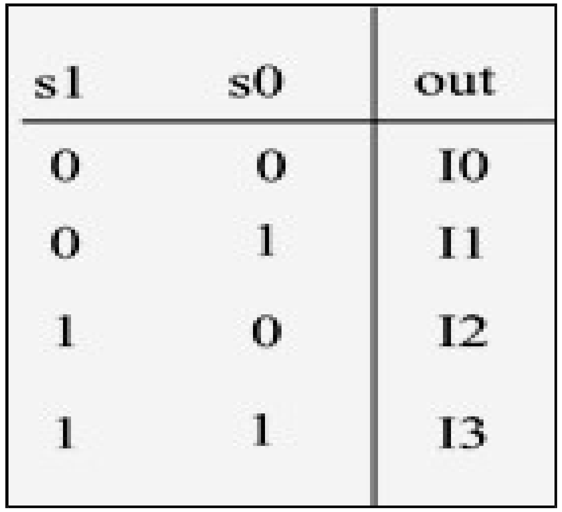
```
initial
  begin
                x=1'b0;y=1'b0;z=1'b0;
     #5         x=1'b0;y=1'b0;z=1'b1;
     #5         x=1'b0;y=1'b1;z=1'b0;
     #5         x=1'b0;y=1'b1;z=1'b1;
     #5         x=1'b1;y=1'b0;z=1'b0;
     #5         x=1'b1;y=1'b0;z=1'b1;
     #5         x=1'b1;y=1'b1;z=1'b0;
     #5         x=1'b1;y=1'b1;z=1'b1;
  end
endmodule
```

Simulation Results for FS

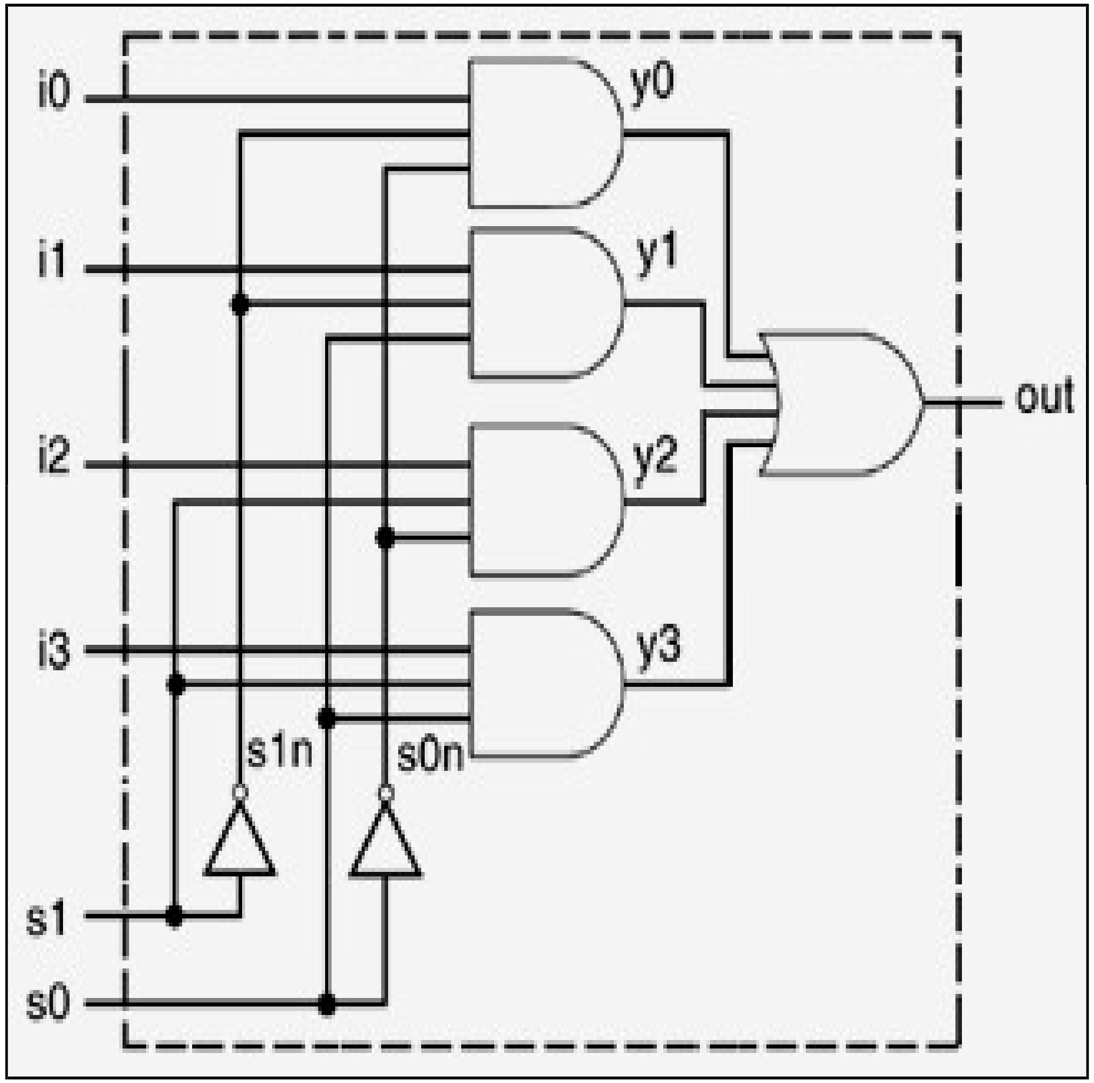
```
# 0     x=0, y=0, z=0, d=0, b=0
# 5     x=0, y=0, z=1, d=1, b=1
# 10    x=0, y=1, z=0, d=1, b=1
# 15    x=0, y=1, z=1, d=0, b=1
# 20    x=1, y=0, z=0, d=1, b=0
# 25    x=1, y=0, z=1, d=0, b=0
# 30    x=1, y=1, z=0, d=0, b=0
# 35    x=1, y=1, z=1, d=1, b=1
```

4-to-1 Multiplexer





s1	s0	out
0	0	I0
0	1	I1
1	0	I2
1	1	I3



4-to-1 Multiplexer

- The logic diagram has a one-to-one correspondence with the VerilogHDL description.
- Two intermediate nets, s0n and s1n, are created. They are complements of input signals s1 and s0.
- Internal nets y0, y1, y2, y3 are also required.
- The instance names are not specified for primitive gates, not, and, and or.
- Instance names are optional for VerilogHDL primitives but are mandatory for instances of user-defined modules.

4-to-1 MUX – Source Code

```
// Module 4-to-1 multiplexer.
// Port list is taken exactly from the I/O diagram.
    module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);
// Port declarations from the I/O diagram
    output out;
    input i0, i1, i2, i3;
    input s1, s0;
// Internal wire declarations
    wire s1n, s0n;
    wire y0, y1, y2, y3;
```

4-to-1 MUX – Source Code

```
// Gate instantiations
// Create s1n and s0n signals.
    not (s1n, s1);
    not (s0n, s0);
// 3-input and gates instantiated
    and (y0, i0, s1n, s0n);
    and (y1, i1, s1n, s0);
    and (y2, i2, s1, s0n);
    and (y3, i3, s1, s0);
// 4-input or gate instantiated
    or (out, y0, y1, y2, y3);
    endmodule
```

4-to-1 MUX – Stimulus

```
// Define the stimulus module (no ports)
 module stimulus;
// Declare variables to be connected to inputs
 reg IN0, IN1, IN2, IN3;
 reg S1, S0;
// Declare output wire
 wire OUTPUT;
// Instantiate the multiplexer
 mux4_to_1  mymux(OUTPUT, IN0, IN1, IN2, IN3, S1, S0);
// Stimulate the inputs
// Define the stimulus module (no ports)
 initial
          begin
```

4-to-1 MUX – Stimulus

```
// set input lines
          IN0 = 1; IN1 = 0; IN2 = 1; IN3 = 0;
          #1 $display("IN0= %b, IN1= %b, IN2= %b,
                                 IN3= %b\n",IN0,IN1,IN2,IN3);
// choose IN0
          S1 = 0; S0 = 0;
          #1 $display("S1 = %b, S0 = %b, OUTPUT = %b
                                                \n", S1, S0, OUTPUT);
// choose IN1
          S1 = 0; S0 = 1;
          #1 $display("S1 = %b, S0 = %b, OUTPUT = %b
                                      \n", S1, S0, OUTPUT);
// choose IN2
          S1 = 1; S0 = 0;
          #1 $display("S1 = %b, S0 = %b, OUTPUT = %b
                                                \n", S1, S0, OUTPUT);
```

4-to-1 MUX – Stimulus

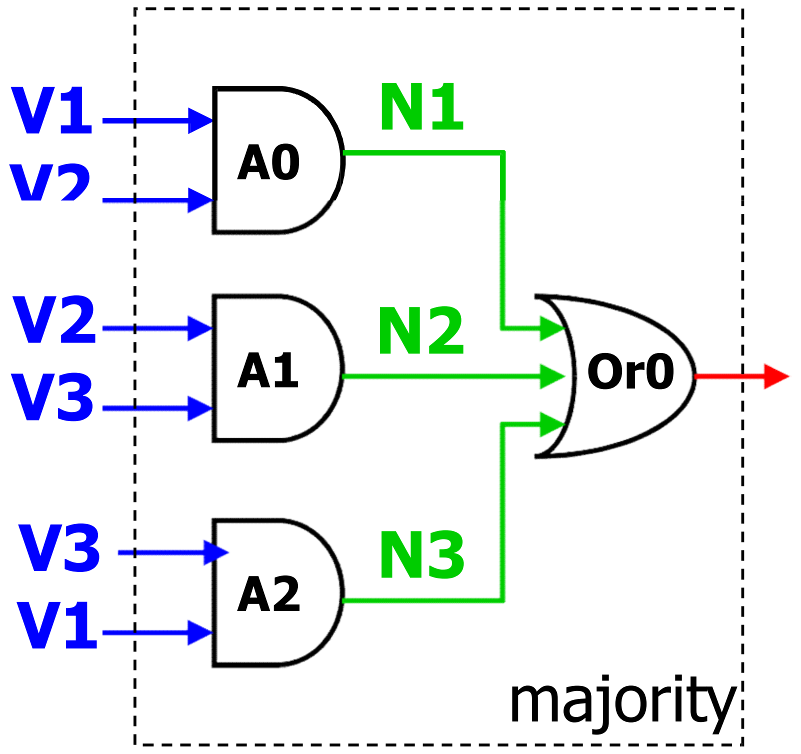
```
// choose IN3
        S1 = 1; S0 = 1;
        #1 $display("S1 = %b, S0 = %b, OUTPUT = %b
                                      \n", S1, S0, OUTPUT);
        end
        endmodule
```

- The output of the simulation is shown below. Each combination of the select signals is tested.

```
IN0= 1, IN1= 0, IN2= 1, IN3= 0
S1 = 0, S0 = 0, OUTPUT = 1
S1 = 0, S0 = 1, OUTPUT = 0
S1 = 1, S0 = 0, OUTPUT = 1
S1 = 1, S0 = 1, OUTPUT = 0
```

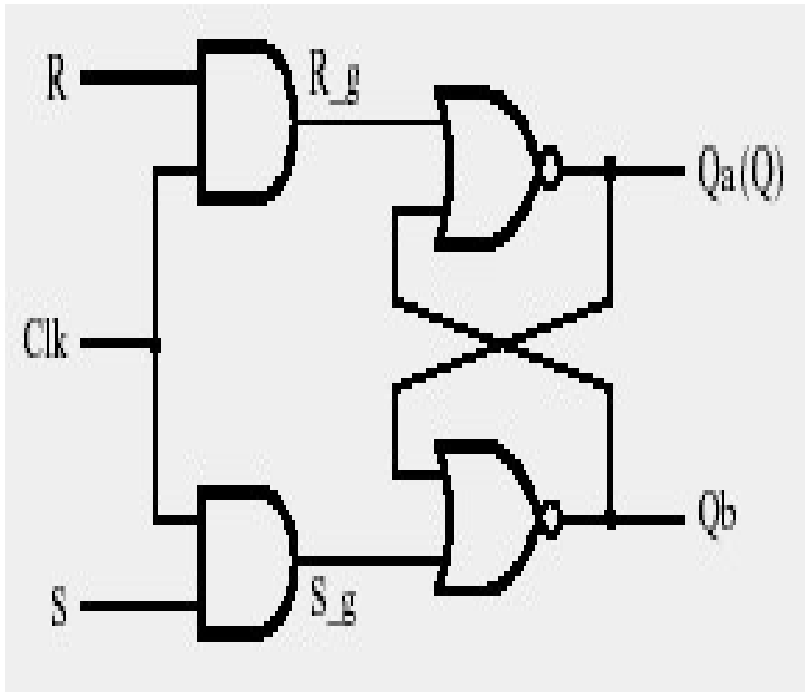
Majority Voter

```
module majority (major, V1, V2, V3) ;
    output major ;
    input V1, V2, V3 ;
    wire N1, N2, N3;
    and A0 (N1, V1, V2),
        A1 (N2, V2, V3),
        A2 (N3, V3, V1);
    or  Or0 (major, N1, N2, N3);
endmodule
```



Gated RS-Latch

```
module part1 (Clk, R, S, Q);
    input Clk, R, S;
    output Q;
    wire R_g, S_g, Qa, Qb;
    and (R_g, R, Clk);
    and (S_g, S, Clk);
    nor (Qa, R_g, Qb);
    nor (Qb, S_g, Qa);
    assign Q = Qa;
endmodule
```



Lecture – 6
Data Flow Modeling

Continuous Assignments

Delays

Expressions, Operators, and Operands

Operator Types

Examples

(4-to-1 Multiplexer, 4-bit Full Adder

Ripple Counter)

Data Flow Modeling

- For small circuits, the gate-level modeling approach works very well because the number of gates is limited and the designer can instantiate and connect every gate individually.
- Also, gate-level modeling is very intuitive to a designer with a basic knowledge of digital logic design.
- However, in complex designs the number of gates is very large.
- Thus, designers can design more effectively if they concentrate on implementing the function at a level of abstraction higher than gate level.
- Dataflow modeling provides a powerful way to implement a design.

Data Flow Modeling

- VerilogHDL allows a circuit to be designed in terms of the data flow between registers and how a design processes data rather than instantiation of individual gates.
- With gate densities on chips increasing rapidly, dataflow modeling has assumed great importance.
- No longer can companies devote engineering resources to handcrafting entire designs with gates.
- Currently, automated tools are used to create a gate-level circuit from a dataflow design description. This process is called logic synthesis.
- Dataflow modeling has become a popular design approach as logic synthesis tools have become sophisticated.

Data Flow Modeling

- This approach allows the designer to concentrate on optimizing the circuit in terms of data flow.
- For maximum flexibility in the design process, designers typically use a VerilogHDL description style that combines the concepts of gate-level, data flow, and behavioral design.
- In the digital design community, the term RTL (Register Transfer Level) design is commonly used for a combination of dataflow modeling and behavioral modeling.

Continuous Assignments

- A continuous assignment is the most basic statement in dataflow modeling, used to drive a value onto a net.
- This assignment replaces gates in the description of the circuit and describes the circuit at a higher level of abstraction.
- The assignment statement starts with the keyword assign. Syntax is given below.

```
continuous_assign ::= assign [ drive_strength ] [delay3 ]
                                        list_of_net_assignments ;
list_of_net_assignments ::= net_assignment
                                        {, net_assignment}
net_assignment ::= net_lvalue = expression
```

Continuous Assignments

- Notice that drive strength is optional and can be specified in terms of strength levels. The default value for drive strength is strong1 and strong0.
- The delay value is also optional and can be used to specify delay on the assign statement. This is like specifying delays for gates.
- Continuous assignments have the following characteristics:
  - The left hand side of an assignment must always be a scalar or vector net or a concatenation of scalar and vector nets. It cannot be a scalar or vector register

Continuous Assignments

- Continuous assignments have the following characteristics:
  - Continuous assignments are always active. The assignment expression is evaluated as soon as one of the right-hand-side operands changes and the value is assigned to the left-hand-side net.
  - The operands on the right-hand side can be registers or nets or function calls. Registers or nets can be scalars or vectors.
  - Delay values can be specified for assignments in terms of time units. Delay values are used to control the time when a net is assigned the evaluated value. This feature is similar to specifying delays for gates. It is very useful in modeling timing behavior in real circuits.

Continuous Assignments

- Examples of Continuous Assignment

```
// Continuous assign. out is a net. i1 and i2 are nets.
        assign out = i1 & i2;
// Continuous assign for vector nets. addr is a 16-bit
// vector net. addr1 and addr2 are 16-bit vector
   registers.
   assign addr[15:0] = addr1_bits[15:0] ^
   addr2_bits[15:0];
// Concatenation. Left-hand side is a concatenation of a
//scalar net and a vector net.
   assign {c_out, sum[3:0]} = a[3:0] + b[3:0] + c_in;
```

Implicit Continuous Assignment

- Instead of declaring a net and then writing a continuous assignment on the net, VerilogHDL provides a shortcut by which a continuous assignment can be placed on a net when it is declared.
- There can be only one implicit declaration assignment per net because a net is declared only once.

```
// Regular continuous assignment wire out;
   assign out = in1 & in2;
// Same effect is achieved by an implicit continuous
// assignment
wire out = in1 & in2;
```

Implicit Net Declaration

- If a signal name is used to the left of the continuous assignment, an implicit net declaration will be inferred for that signal name.
- If the net is connected to a module port, the width of the inferred net is equal to the width of the module port.

```
// Continuous assign. out is a net.
    wire i1, i2;
    assign out = i1 & i2;
// Note that out was not declared as a wire but an
// implicit wire declaration for out //is done by the
// simulator
```

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.
- Three ways of specifying delays in continuous assignment statements are
  - Regular assignment delay
  - Implicit continuous assignment delay
  - Net declaration delay.

Regular Assignment Delay

- To assign a delay value in a continuous assignment statement.
- The delay value is specified after the keyword assign.

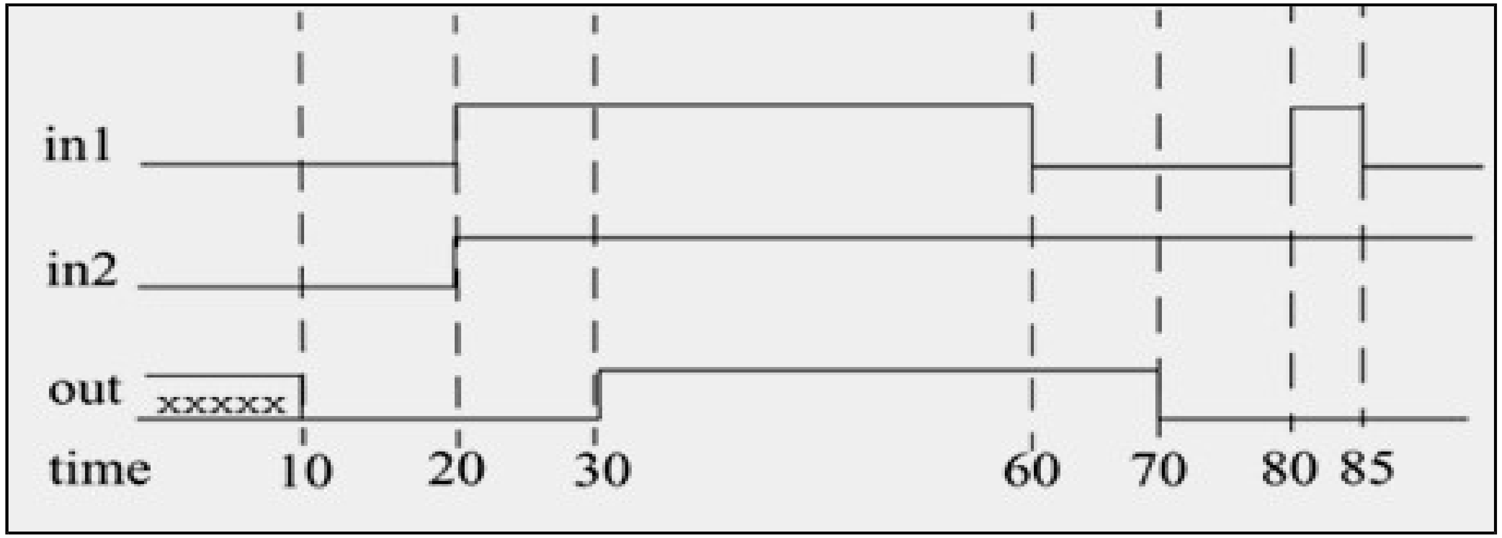
```
assign #10 out = in1 & in2;
// Delay in a continuous assign
```

  - Any change in values of in1 or in2 will result in a delay of 10 time units before recomputation of the expression in1 & in2, and the result will be assigned to out.
  - If in1 or in2 changes value again before 10 time units when the result propagates to out, the values of in1 and in2 at the time of recomputation are considered. This property is called inertial delay.
  - An input pulse that is shorter than the delay of the assignment statement does not propagate to the output.

Regular Assignment Delay

assign #10 out = in1 & in2; // Delay in a continuous assign

- When signals in1 and in2 go high at time 20, out goes to a high 10 time units later (time = 30).
- When in1 goes low at 60, out changes to low at 70.
- However, in1 changes to high at 80, but it goes down to low before 10 time units have elapsed.
- Hence, at the time of recomputation, 10 units after time 80, in1 is 0. Thus, out gets the value 0. A pulse of width less than the specified assignment delay is not propagated to the output.



Implicit Continuous Assignment Delay

- An equivalent method is to use an implicit continuous assignment to specify both a delay and an assignment on the net.

```
//implicit continuous assignment delay
wire #10 out = in1 & in2;
//same as
wire out;
assign #10 out = in1 & in2;
```

- The declaration above has the same effect as defining a wire out and declaring a continuous assignment on out.

Net Declaration Delay

- A delay can be specified on a net when it is declared without putting a continuous assignment on the net.
- If a delay is specified on a net out, then any value change applied to the net out is delayed accordingly.
- Net declaration delays can also be used in gate-level modeling.

```
//Net Delays
wire # 10 out;
assign out = in1 & in2;
//The above statement has the same effect as follows.
wire out;
assign #10 out = in1 & in2;
```

Expressions

- Expressions are constructs that combine operators and operands to produce a result.

```
//Examples of expressions.
//Combines operands and operators
a ^ b
addr1[20:17] + addr2[20:17]
in1 | in2
```

Operands

- Operands can be any one of the data types.
- Some constructs will take only certain types of operands.
- Operands can be constants, integers, real numbers, nets, registers, times, bit-select (one bit of vector net or a vector register), part-select (selected bits of the vector net or register vector), and memories or function calls.

```
integer count, final_count;
final_count = count + 1; //count is an integer operand
real a, b, c; c = a - b;
//a and b are real operands
reg [15:0] reg1, reg2;
reg [3:0] reg_out;
reg_out = reg1[3:0] ^ reg2[3:0];
//reg1[3:0] and reg2[3:0] are part-select register operands
```

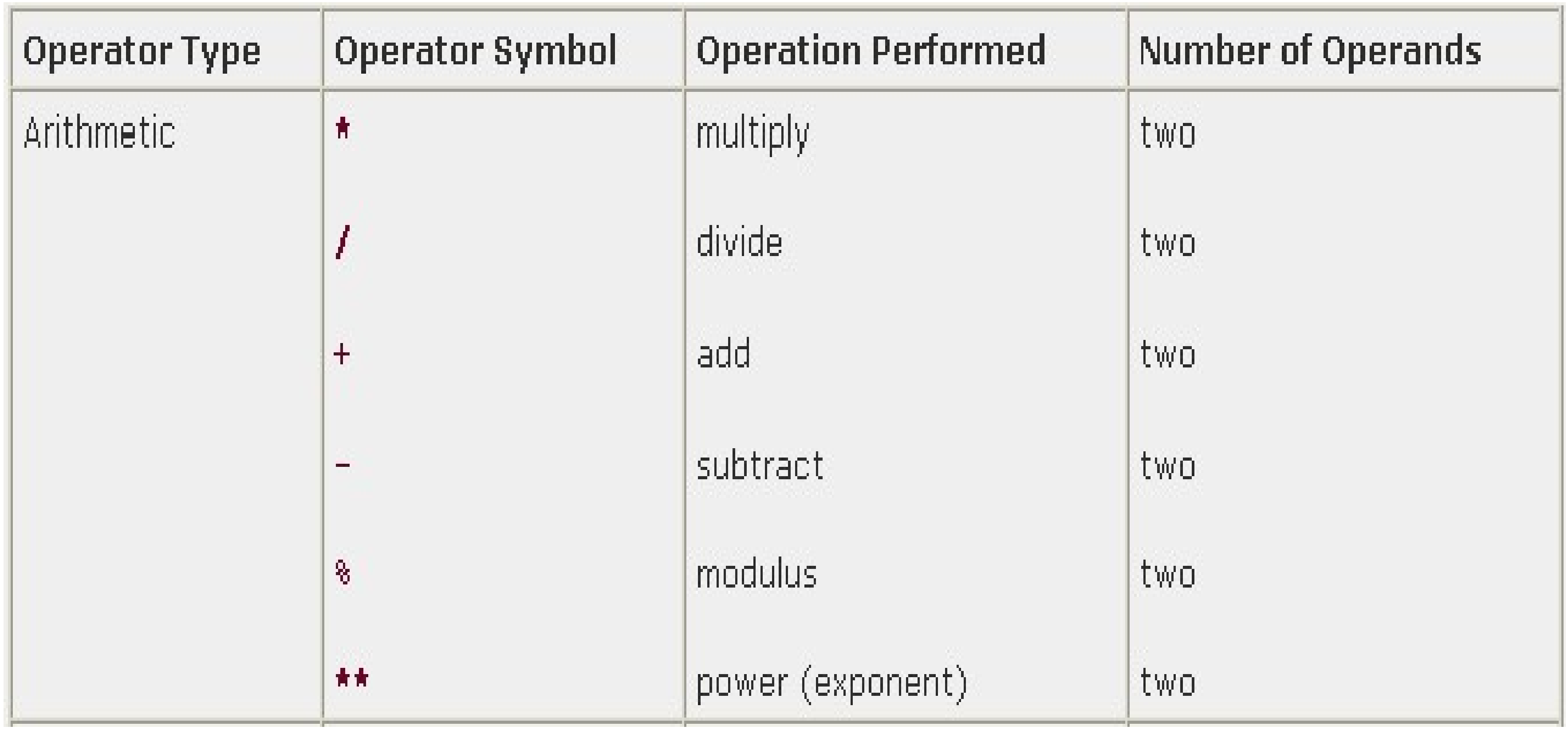
Operators

- Operators act on the operands to produce desired results.
- VerilogHDL provides various types of operators.
  - Arithmetic Operators, Logical Operators
  - Relational Operators, Equality Operators
  - Bitwise Operators, Reduction Operators
  - Shift Operators, Concatenation Operator
  - Replication Operator, Conditional Operator

d1 && d2 // && is an operator on operands d1 and d2 !a[0] // ! is an operator on operand a[0]

B >>1 // >> is an operator on operands B and 1

Arithmetic Operators



Operator Type	Operator Symbol	Operation Performed	Number of Operands
Arithmetic	*	multiply	two
	/	divide	two
	+	add	two
	-	subtract	two
	%	modulus	two
	**	power (exponent)	two

- There are two types of arithmetic operators: binary and unary.

Binary Operators

- **Binary arithmetic operators are multiply (*), divide (/), add (+), subtract (-), power (**), and modulus (%). Binary operators take two operands.**

```
A = 4'b0011; B = 4'b0100; // A and B are register vectors
D = 6; E = 4; F=2// D and E are integers
A * B // Multiply A and B. Evaluates to 4'b1100
D / E // Divide D by E. Evaluates to 1. Truncates any
        //fractional part.
A + B // Add A and B. Evaluates to 4'b0111
B - A // Subtract A from B. Evaluates to 4'b0001
F = E ** F; //E to the power F, yields 16
```

Binary Operators

- If any operand bit has a value x, then the result of the entire expression is x. This seems intuitive because if an operand value is not known precisely, the result should be an unknown.

```
in1 = 4'b101x;
in2 = 4'b1010;
sum = in1 + in2; // sum will be evaluated to the value 4'bx
```

- Modulus operators produce the remainder from the division of two numbers. They operate similarly to the modulus operator in the C programming language.

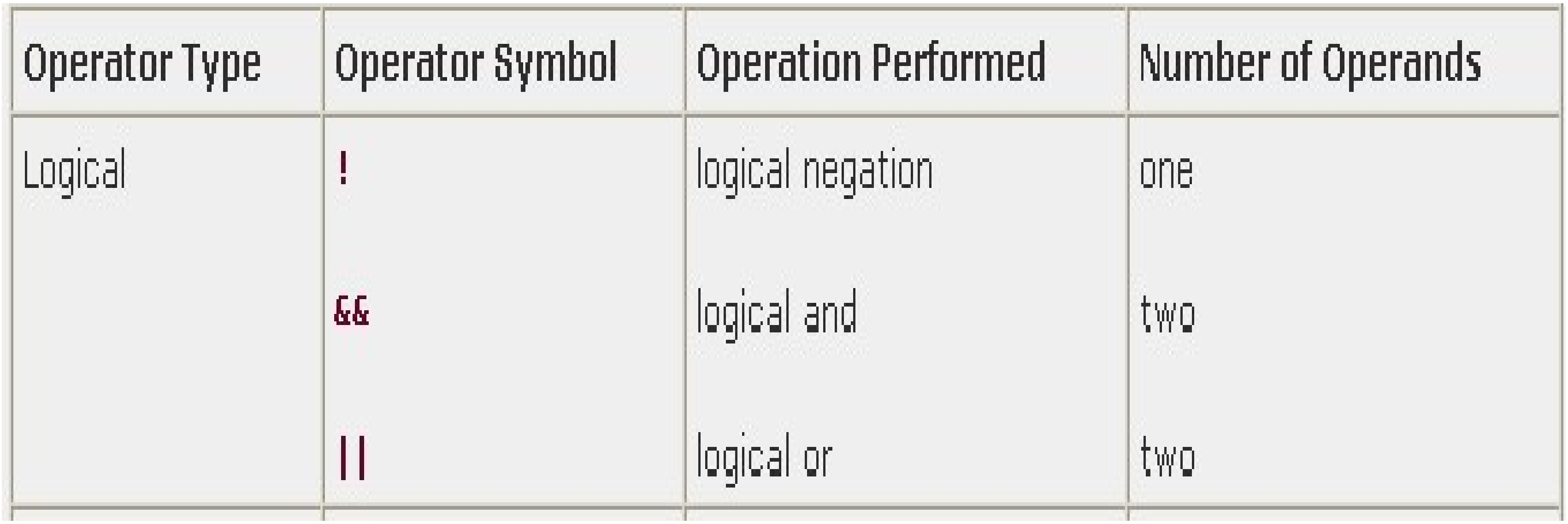
```
13 % 3 // Evaluates to 1
16 % 4 // Evaluates to 0
-7 % 2 // Evaluates to -1, takes sign of the first operand
7 % -2 // Evaluates to +1, takes sign of the first  operand
```

Unary Operators

- The operators + and - can also work as unary operators.
- They are used to specify the positive or negative sign of the operand.
- Unary + or – operators have higher precedence than the binary + or – operators.

  -4 // Negative 4 +5 // Positive 5

Logical Operators



Operator Type	Operator Symbol	Operation Performed	Number of Operands
Logical	!	logical negation	one
	&&	logical and	two
	||	logical or	two

- **Logical operators are logical-and (&&), logical-or (||) and logical-not (!). Operators && and || are binary operators.**

Logical Operators

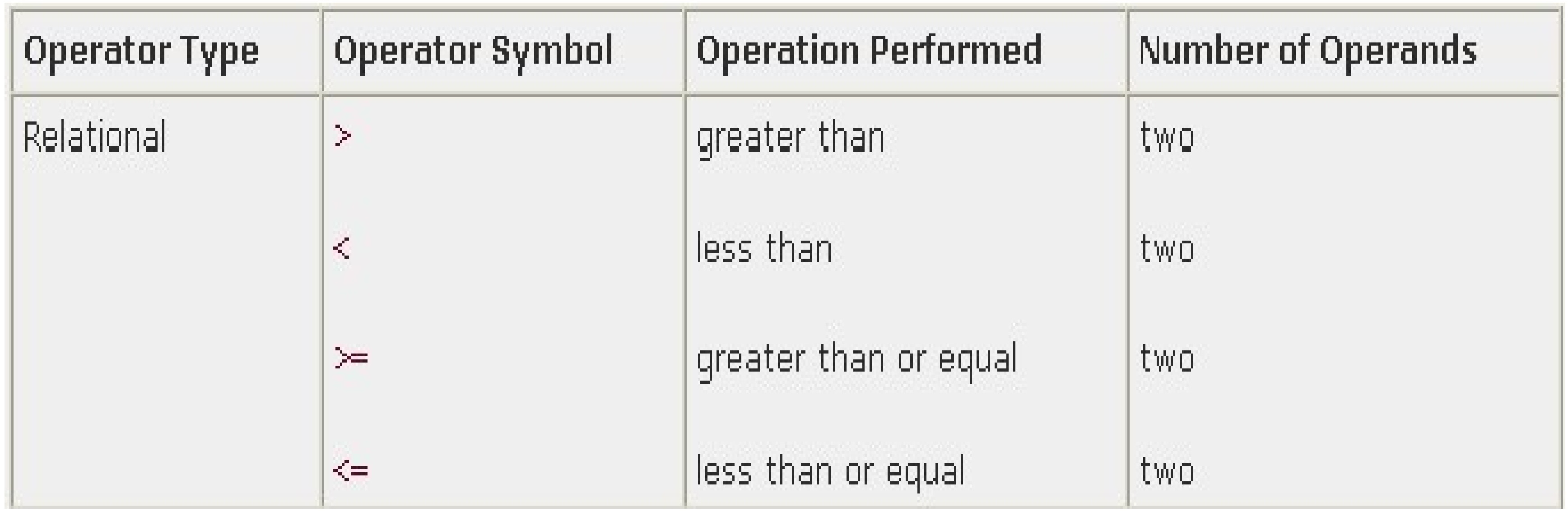
- Operator ! is a unary operator. Logical operators follow these conditions:
  - Logical operators always evaluate to a 1-bit value, `0` (false), `1` (true), or `x` (ambiguous).
  - If an operand is not equal to zero, it is equivalent to a logical `1` (true condition). If it is 01equal to zero, it is equivalent to a logical `0` (false condition). If any operand bit is `x` or `z`, it is equivalent to `x` (ambiguous condition) and is normally treated by simulators as a false condition.
  - Logical operators take variables or expressions as operands.

Logical Operators

- **Use of parentheses to group logical operations is highly recommended to improve readability. Also, the user does not have to remember the precedence of operators.**

```
// Logical operations
A= 3; B=  0;
A && B //Evaluates to 0.
A || B // Evaluates to  1.
!A// Evaluates to 0.
!B// Evaluates to 1.
// Unknowns
A = 2'b0x; B = 2'b10; A && B // Evaluates to x.
// Expressions
(a == 2) && (b == 3) // Evaluates to 1
                                   //If both a == 2 and b == 3 are  true.
```

Relational Operators



Operator Type	Operator Symbol	Operation Performed	Number of Operands
Relational	>	greater than	two
	<	less than	two
	>=	greater than or equal	two
	<=	less than or equal	two

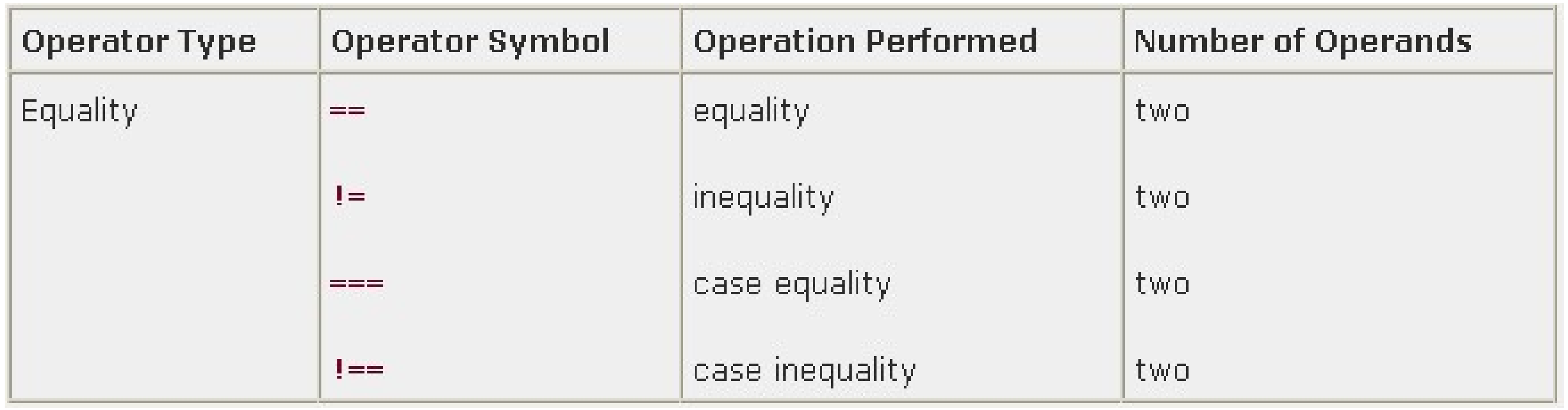
- Relational operators are greater-than (>), less-than (<), greater-than-or-equal-to (>=), and less-than-or-equal-to (<=).

Relational Operators

- If relational operators are used in an expression, the expression returns a logical value of 1 if the expression is true and 0 if the expression is false.
- If there are any unknown or z bits in the operands, the expression takes a value x.
- These operators function exactly as the corresponding operators in the C programming language.

```
// A = 4, B = 3
// X = 4'b1010, Y = 4'b1101, Z = 4'b1xxx
A <= B // Evaluates to a logical 0
A > B // Evaluates to a logical 1
Y >= X // Evaluates to a logical 1
Y < Z // Evaluates to an x
```

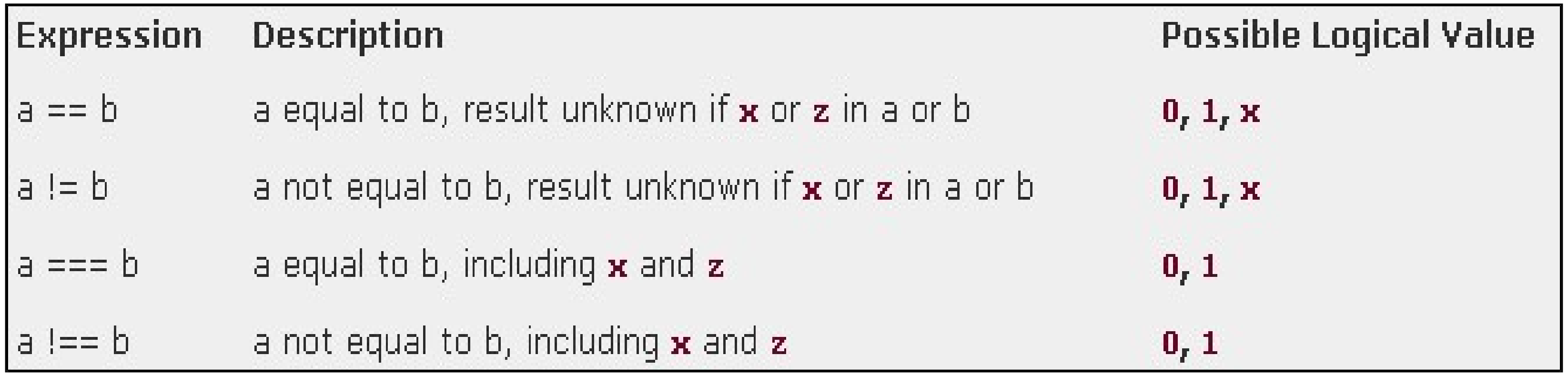
Equality Operators



Operator Type	Operator Symbol	Operation Performed	Number of Operands
Equality	==	equality	two
	!=	inequality	two
	===	case equality	two
	!==	case inequality	two

- **Equality operators are logical equality (==), logical inequality (!=), case equality (===), and case inequality (!==).**
- **When used in an expression, equality operators return logical value 1 if true, 0 if false.**
- **These operators compare the two operands bit by bit, with zero filling if the operands are of unequal length.**

Equality Operators



Expression	Description	Possible Logical Value
a == b	a equal to b, result unknown if **x** or **z** in a or b	**0, 1, x**
a != b	a not equal to b, result unknown if **x** or **z** in a or b	**0, 1, x**
a === b	a equal to b, including **x** and **z**	**0, 1**
a !== b	a not equal to b, including **x** and **z**	**0, 1**

- **It is important to note the difference between the logical equality operators (==, !=) and case equality operators (===, !==).**
- **The logical equality operators (==, !=) will yield an x if either operand has x or z in its bits.**
- **However, the case equality operators (===, !==) compare both operands bit by bit and compare all bits, including x and z.**
- **The result is 1 if the operands match exactly, including x and z bits.**
- **The result is 0 if the operands do not match exactly. Case equality operators never result in an x.**

Equality Operators

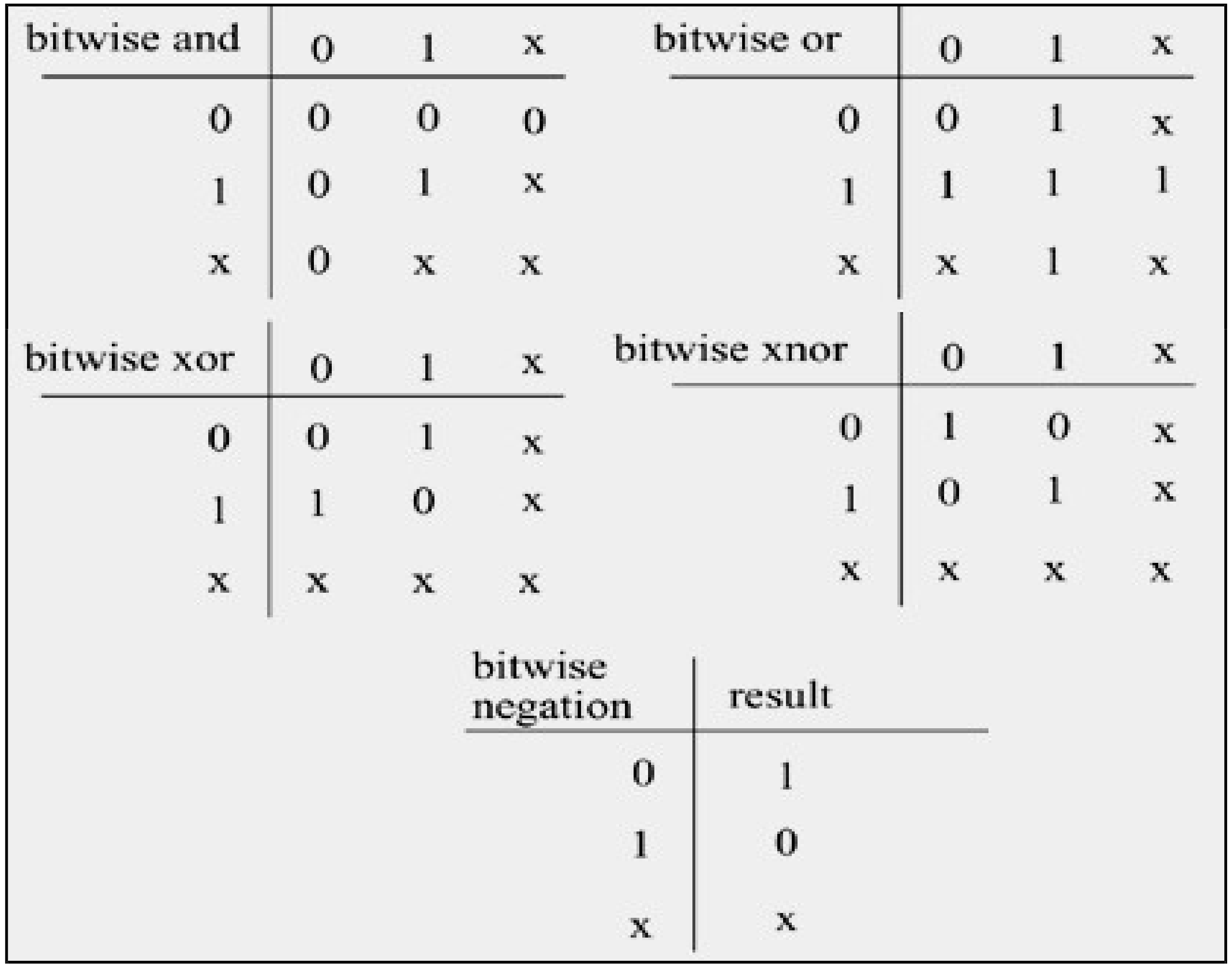
```
// A = 4, B = 3
// X = 4'b1010, Y = 4'b1101
// Z = 4'b1xxz, M = 4'b1xxz, N = 4'b1xxx
A == B // Results in logical 0
X != Y // Results in logical 1
X == Z // Results in x
Z === M // Results in logical 1 (all bits match,
              //including x and z)
Z === N // Results in logical 0 (least significant bit
             //does not match)
M !== N // Results in logical 1
```

Bit-Wise Operators

- Bitwise operators are negation (~), and(&), or (|), xor (^), xnor (^~, ~^).
- Bitwise operators perform a bit-by-bit operation on two operands.
- They take each bit in one operand and perform the operation with the corresponding bit in the other operand.
- If one operand is shorter than the other, it will be bit-extended with zeros to match the length of the longer operand.
- A z is treated as an x in a bitwise operation.
- The exception is the unary negation operator (~), which takes only one operand and operates on the bits of the single operand.

Bit-Wise Operators

- **Logic tables for the bit-by-bit computation are shown in the table.**



bitwise and	0	1	x
0	0	0	0
1	0	1	x
x	0	x	x

bitwise or	0	1	x
0	0	1	x
1	1	1	1
x	x	1	x

bitwise xor	0	1	x
0	0	1	x
1	1	0	x
x	x	x	x

bitwise xnor	0	1	x
0	1	0	x
1	0	1	x
x	x	x	x

bitwise negation	result
0	1
1	0
x	x

Bit-Wise Operators

- **Examples of bitwise operators are shown below.**

```
// X = 4'b1010, Y = 4'b1101
 // Z = 4'b10x1
~X // Negation. Result is 4'b0101
X & Y // Bitwise and. Result is 4'b1000
X | Y // Bitwise or. Result is 4'b1111
X ^Y // Bitwise xor. Result is 4'b0111
X ^~ Y // Bitwise xnor. Result is 4'b1000
X & Z // Result is 4'b10x0
```

Bit-Wise Operators

- It is important to distinguish bit-wise operators ~, &, and | from logical operators !, &&, ||.
- Logical operators always yield a logical value 0, 1, x, whereas bitwise operators yield a bit-by-bit value.
- Logical operators perform a logical operation, not a bit-by-bit operation.

```
 // X = 4'b1010, Y = 4'b0000
X | Y // bitwise operation. Result is 4'b1010
X || Y // logical operation.
        //Equivalent to 1 || 0. Result is 1.
```

Reduction Operators

- Reduction operators are and (&), nand (~&), or (|), nor (~|), xor (^), and xnor (~^, ^~). Reduction operators take only one operand.
- Reduction operators perform a bit-wise operation on a single vector operand and yield a 1-bit result.
- The difference is that bit-wise operations are on bits from two different operands, whereas reduction operations are on the bits of the same operand.
- Reduction operators work bit by bit from right to left.
- Reduction nand, reduction nor, and reduction xnor are computed by inverting the result of the reduction and, reduction or, and reduction xor, respectively.

Reduction Operators

// X = 4'b1010 &X

//Equivalent to 1 & 0 & 1 & 0. Results in 1'b0 |X

//Equivalent to 1 | 0 | 1 | 0. Results in 1'b1 ^X

//Equivalent to 1 ^0 ^1 ^0. Results in 1'b0

//A reduction xor or xnor can be used for even or odd //parity generation of a vector.

- The use of a similar set of symbols for logical (!, &&, ||), bitwise (~, &, |, ^), and reduction operators (&, |, ^) is somewhat confusing initially.
- The difference lies in the number of operands each operator takes and also the value of results computed.

Shift Operators

- Shift operators are right shift (>>), left shift (<<), arithmetic right shift (>>>), and arithmetic left shift (<<<).
- Regular shift operators shift a vector operand to the right or the left by a specified number of bits.
- The operands are the vector and the number of bits to shift.
- When the bits are shifted, the vacant bit positions are filled with zeros.
- Shift operations do not wrap around.
- Arithmetic shift operators use the context of the expression to determine the value with which to fill the vacated bits.

Shift Operators

- Shift operators are useful because they allow the designer to model shift operations, shift-and-add algorithms for multiplication, and other useful operations.

```
// X = 4'b1100
Y = X >> 1; //Y is 4'b0110. Shift right 1 bit. 0 filled in MSB position.
Y = X << 1; //Y is 4'b1000. Shift left 1 bit. 0 filled in LSB position.
Y = X << 2; //Y is 4'b0000. Shift left 2 bits.
integer a, b, c; //Signed data types
a = 0;
b = -10; // 00111...10110 binary
c = a + (b >>> 3); //Results in -2 decimal, due to arithmetic shift
```

Concatenation Operator

- The concatenation operator ({, }) provides a mechanism to append multiple operands.
- The operands must be sized.
- Unsized operands are not allowed because the size of each operand must be known for computation of the size of the result.
- Concatenations are expressed as operands within braces, with commas separating the operands.
- Operands can be scalar nets or registers, vector nets or registers, bit-select, part-select, or sized constants.

```
// 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
```

Concatenation Operator

```
module  adder4b (sum, c_out, a, b, c_in);
    input       [3:0]  a,   b;
    input              c_in;
    output      [3:0]  sum;
    output             c_out;

    assign {c_out, sum} = a + b + c_in;

endmodule
```

Replication Operator

- Repetitive concatenation of the same number can be expressed by using a replication constant.
- A replication constant specifies how many times to replicate the number inside the brackets ({ }).

```
reg A;
reg [1:0] B, C;
reg [2:0] D;
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

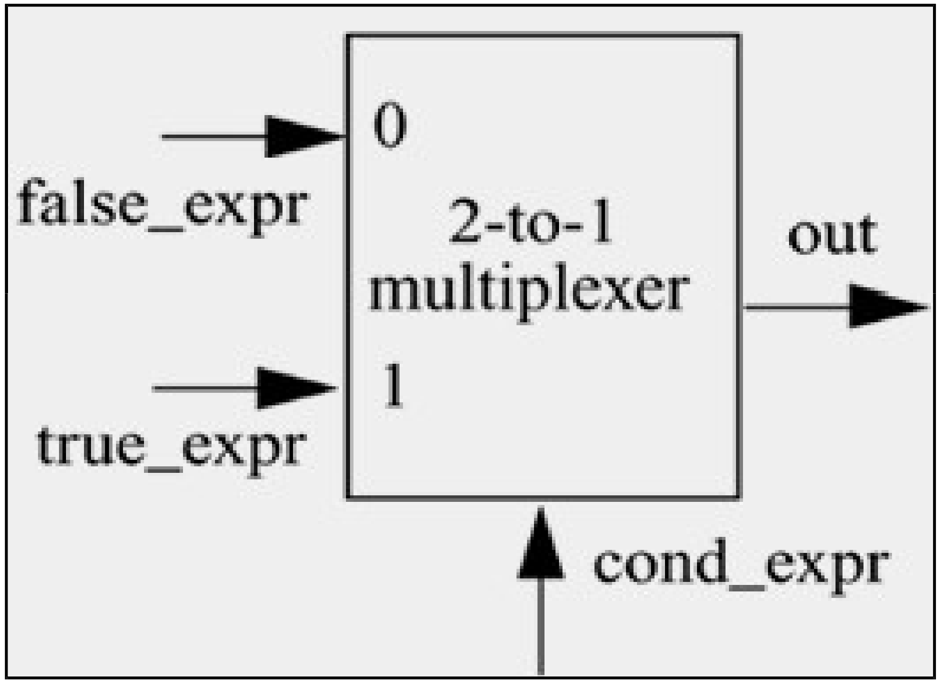
- The conditional operator(?:) takes three operands

  Usage: condition_expr ? true_expr : false_expr ;

  - The condition expression (condition_expr) is first evaluated.
  - If the result is true (logical 1), then the true_expr is evaluated.
  - If the result is false (logical 0), then the false_expr is evaluated.
  - If the result is x (ambiguous), then both true_expr and false_expr are evaluated and their results are compared, bit by bit, to return for each bit position an x if the bits are different and the value of the bits if they are the same.

Conditional Operator

- The action of a conditional operator is similar to a multiplexer.



- Alternately, it can be compared to the if-else expression.

Conditional Operator

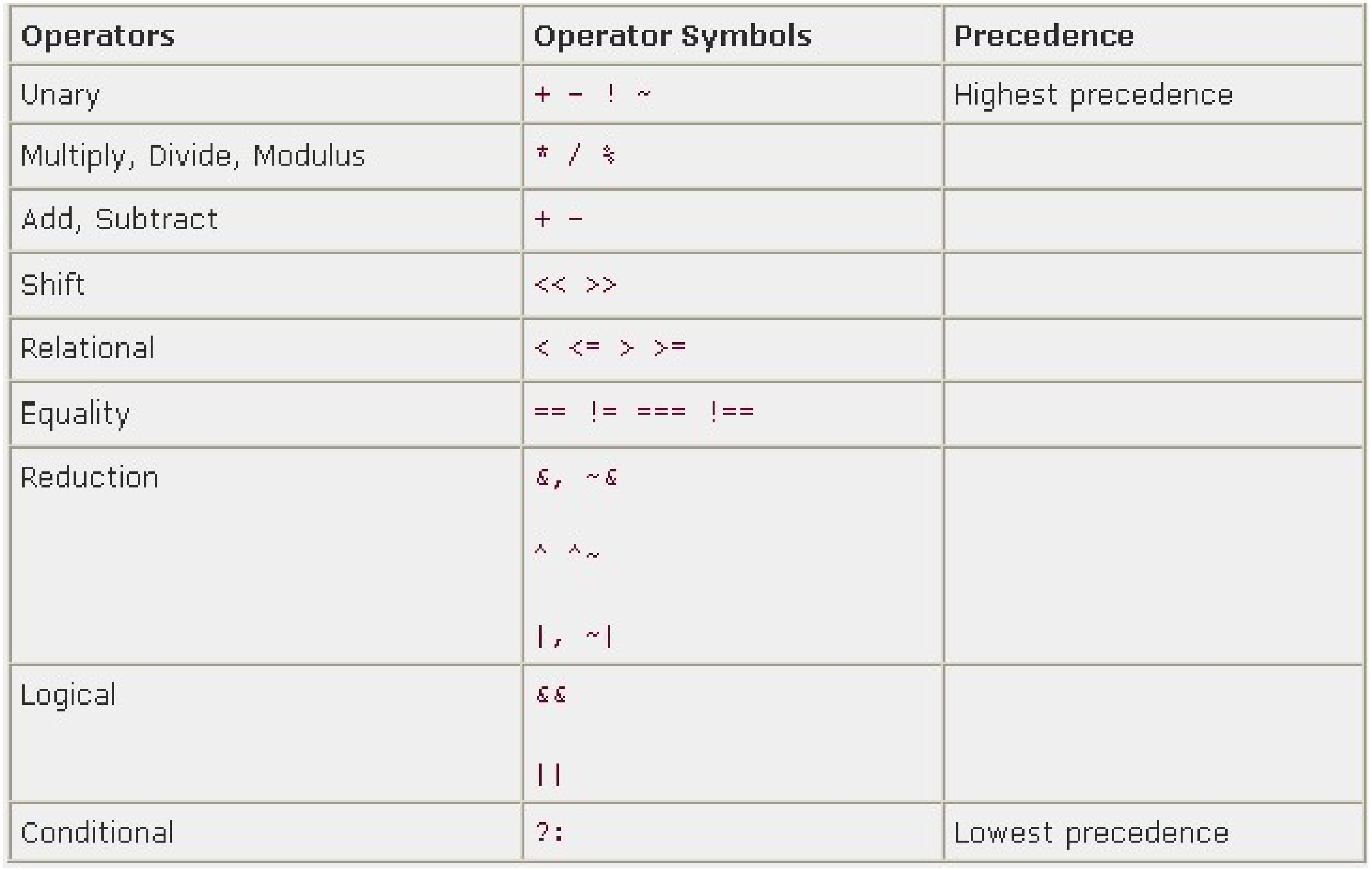
- Conditional operators are frequently used in dataflow modeling to model conditional assignments.
- The conditional expression acts as a switching control.

```
//model functionality of a tristate buffer
assign addr_bus = drive_enable ? addr_out : 36'bz;
//model functionality of a 2-to-1 mux
assign out = control ? in1 : in0;
```

- Conditional operations can be nested. Each true_expr or false_expr can itself be a conditional operation.
- In the given example, the (A==3) and control are the two select signals of 4-to-1 multiplexer with n, m, y, x as the inputs and out as the output signal.

```
assign out = (A == 3) ? ( control ? x : y ): ( control ? m : n) ;
```

Operator Precedence



Operators	Operator Symbols	Precedence
Unary	+ - ! ~	Highest precedence
Multiply, Divide, Modulus	* / %	
Add, Subtract	+ -	
Shift	<< >>	
Relational	< <= > >=	
Equality	== != === !==	
Reduction	&, ~& ^ ^~ |, ~|	
Logical	&& ||	
Conditional	?:	Lowest precedence

Full Adder

```
module fa_rtl (A, B, CI, S, CO) ;
   input A, B, CI ;
   output S, CO ;
// use continuous assignments
   assign S = A ^  B ^ CI;
   assign C0 = (A & B) | (A & CI) | (B & CI);
endmodule
```

4-to-1 Multiplexer

- **Using Logic Equations**

```
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);
// Port declarations from the I/O diagram
output out;
input i0, i1, i2, i3;
input s1, s0;
//Logic equation for out
assign out = (~s1 & ~s0 & i0)|
             (~s1 & s0 & i1) |
             (s1 & ~s0 & i2) |
             (s1 & s0 & i3) ;
 endmodule
```

4-to-1 Multiplexer

- **Using Conditional Operators**

```
module multiplexer4_to_1 (out, i0, i1, i2, i3, s1, s0);
// Port declarations from the I/O diagram output out;
input i0, i1, i2, i3;
input s1, s0;
// Use nested conditional operator
assign out = s1 ? ( s0 ? i3 : i2) : (s0 ? i1 : i0) ;
endmodule
```

4-bit Full Adder

- **Using Dataflow Operators**

```
module fulladd4(sum, c_out, a, b, c_in);
// I/O port declarations
output [3:0] sum;
output c_out;
input[3:0] a, b;
input c_in; // Specify the function of a full adder
assign {c_out, sum} = a + b + c_in;
endmodule
```

4-bit Full adder
(with carry look-ahead)

- In ripple carry adders, the carry must propagate through the gate levels before the sum is available at the output terminals.
- An n-bit ripple carry adder will have 2n gate levels.
- The propagation time can be a limiting factor on the speed of the circuit.
- One of the most popular methods to reduce delay is to use a carry lookahead mechanism.
- Logic equations for implementing the carry lookahead mechanism can be found in any logic design book.
- The propagation delay is reduced to four gate levels, irrespective of the number of bits in the adder.

4-bit Full adder
(with carry look-ahead)

```
module fulladd4(sum, c_out, a, b, c_in);
// Inputs and outputs
    output [3:0] sum;
    output c_out;
    input [3:0] a,b;
    input c_in;
// Internal wires
    wire p0,g0, p1,g1, p2,g2, p3,g3;
    wire c4, c3, c2, c1;
```

4-bit Full adder

(with carry look-ahead)

```
// compute the p for each stage
      assign p0 = a[0] ^ b[0],
                    p1 = a[1] ^ b[1],
                    p2 = a[2] ^ b[2],
                    p3 = a[3] ^ b[3];
// compute the carry for each stage
// c_in is equivalent c0 in the arithmetic
equation for carry lookahead computation
      assign c1 = g0 | (p0 & c_in),
                 c2 = g1 | (p1 & g0) | (p1 & p0 & c_in),
                 c3 = g2 | (p2 & g1) | (p2 & p1 & g0)
                          | (p2 & p1 & p0 & c_in),
```

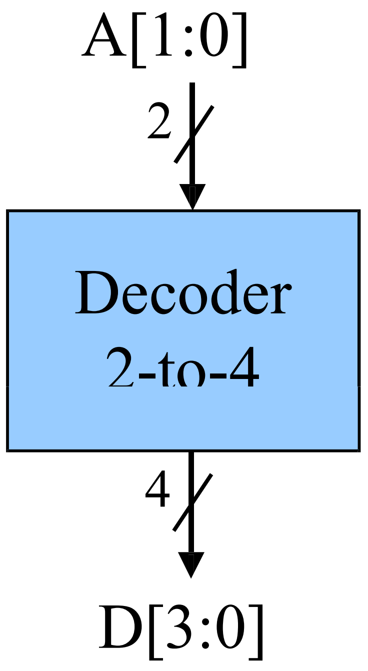
4-bit Full adder

(with carry look-ahead)

```
    c4 = g3 | (p3 & g2) | (p3 & p2 & g1)
                 | (p3 & p2 & p1 & g0)
                 | (p3 & p2 & p1 & p0 & c_in);
// Compute Sum assign
    sum[0] = p0 ^ c_in,
    sum[1] = p1 ^ c1,
    sum[2] = p2 ^ c2,
    sum[3] = p3 ^ c3;
// Assign carry output
    assign c_out = c4;
endmodule
```

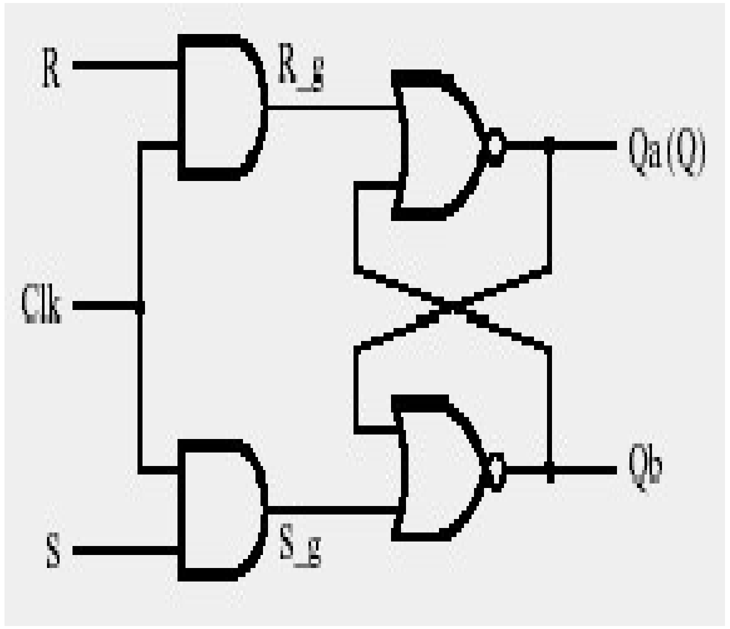
Decoder (2-TO-4)

```
module decoder_2_to_4 (A, D) ;
      input [1:0] A ;
      output [3:0] D ;
      assign D = (A == 2'b00) ? 4'b0001 :
                          (A == 2'b01) ? 4'b0010 :
                          (A == 2'b10) ? 4'b0100 :
                          (A == 2'b11) ? 4'b1000 ;
endmodule
```



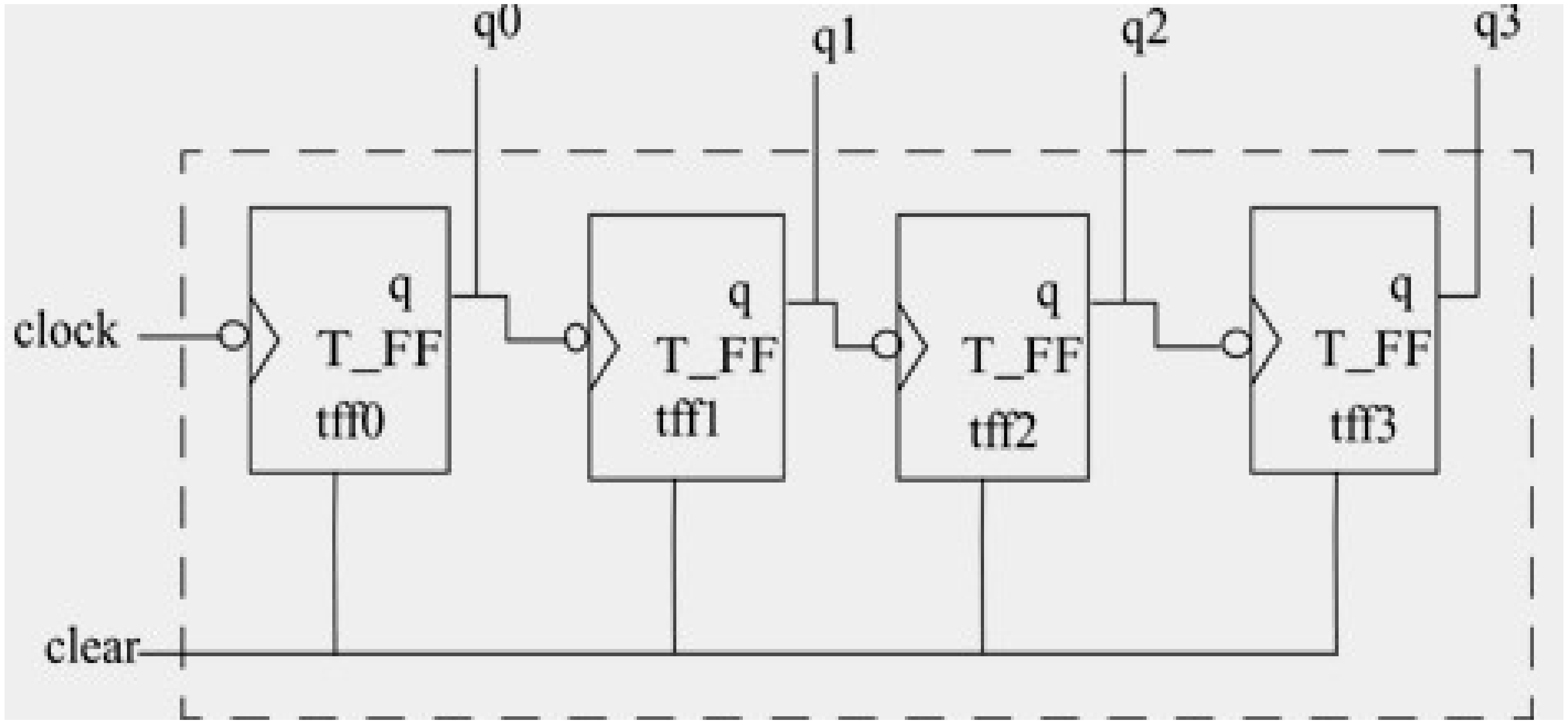
Gated RS-Latch

```
module part1 (Clk, R, S, Q);
    input Clk, R, S;
    output Q;
    wire R_g, S_g, Qa, Qb;
    assign R_g = R & Clk;
    assign S_g = S & Clk;
    assign Qa = ~ (R_g | Qb);
    assign Qb = ~ (S_g | Qa);
    assign Q = Qa;
endmodule
```



Ripple Counter

- The diagrams for the 4-bit ripple carry counter modules are shown below.
- The counter is built with four T-flipflops.

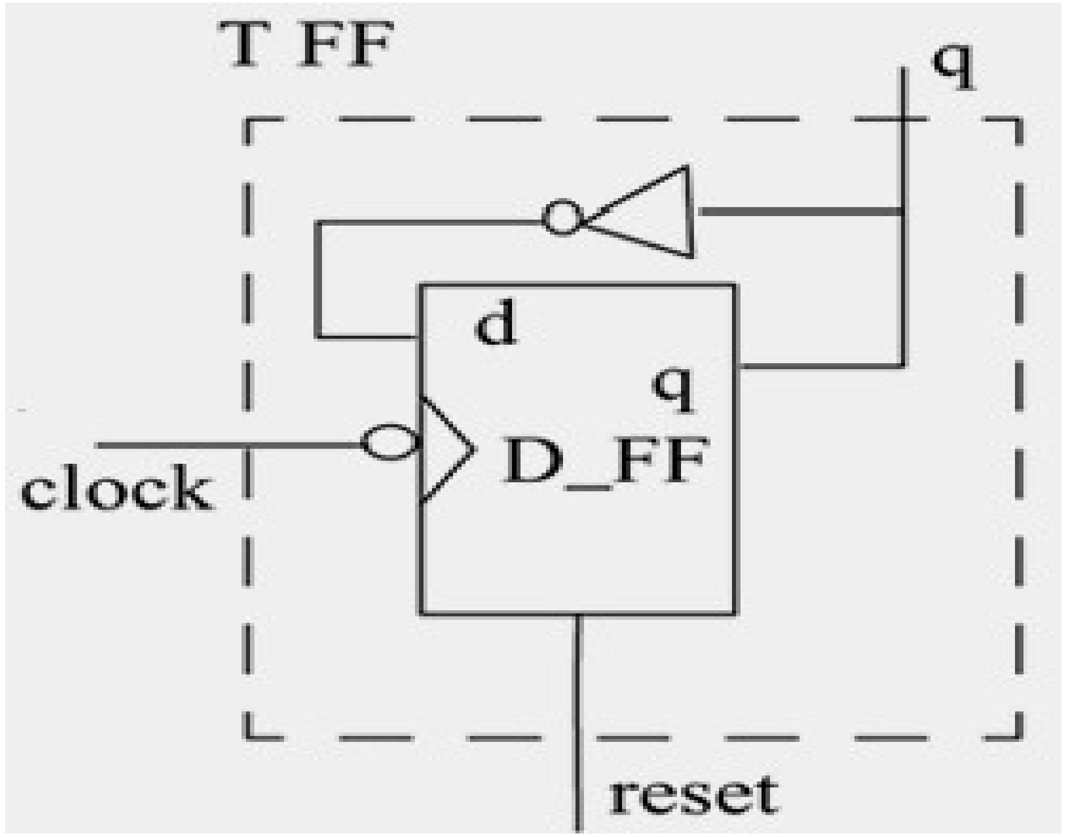


Ripple Counter

```
module counter(Q , clock, clear);
// I/O ports
    output [3:0] Q;
    input clock, clear;
// Instantiate the T flipflops
    T_FF tff0(Q[0], clock, clear);
    T_FF tff1(Q[1], Q[0], clear);
    T_FF tff2(Q[2], Q[1], clear);
    T_FF tff3(Q[3], Q[2], clear);
endmodule
```

T Flip-Flop

- The T-flipflop is built with one D-flipflop and an inverter gate.

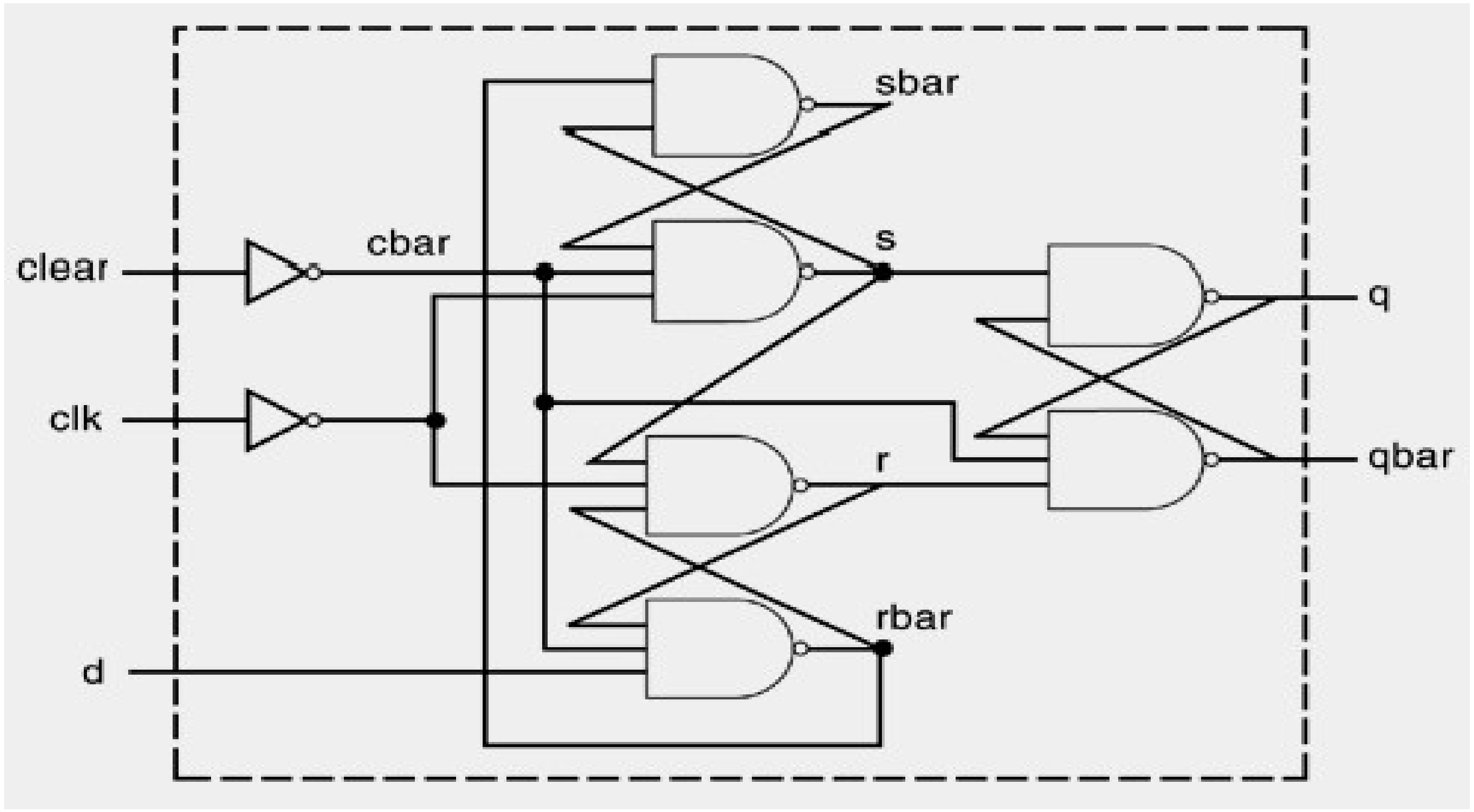


T Flip-Flop

```
//Toggles every clockcycle.
module T_FF(q, clk, clear);
// I/O ports
    output q;
    input clk, clear;
// Instantiate the edge-triggered DFF
// Complement of output q is fed back.
// Notice qbar not needed. Unconnected port.
    edge_dff ff1(q, ,~q, clk, clear);
endmodule
```

D Flip-Flop

- The D-flipflop constructed from basic logic gates.



D Flip-Flop

```
// Edge-triggered D flipflop
module edge_dff(q, qbar, d, clk, clear);
// Inputs and outputs
      output q,qbar;
      input d, clk, clear;
// Internal variables
      wire s, sbar, r, rbar,cbar;
// dataflow statements
//Create a complement of signal clear
      assign cbar = ~clear;
```

D Flip-Flop

```
// Input latches; A latch is level sensitive.
// An edge-sensitive flip-flop is implemented
// by using 3 SR latches.
      assign sbar = ~(rbar & s),
                  s = ~(sbar & cbar & ~clk),
                  r = ~(rbar & ~clk & s),
              rbar = ~(r & cbar & d);
// Output latch
assign q = ~(s & qbar),
       qbar = ~(q & r & cbar);
endmodule
```

Lecture – 7
Behavioral Modeling

- Structured Procedures
- Procedural Assignments
- Timing Controls
- Conditional Statements
- Multiway Branching
- Loops
- Examples

Structured Procedures

- There are two structured procedure statements in VerilogHDL
  - always and initial.
- These statements are the two most basic statements in behavioral modeling.
- All other behavioral statements can appear only inside these structured procedure statements.
- VerilogHDL is a concurrent programming language unlike the C programming language, which is sequential in nature.
- Activity flows in VerilogHDL run in parallel rather than in sequence.
- Each always and initial statement represents a separate activity flow in VerilogHDL.
- Each activity flow starts at simulation time 0.
- The statements always and initial cannot be nested.

initial

- All statements inside an initial statement constitute an initial block.
- An 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.
- Each block finishes execution independently of other blocks.
- Multiple behavioral statements must be grouped, typically using the keywords begin and end.
- If there is only one behavioral statement, grouping is not necessary. This is similar to the begin-end blocks in Pascal programming language or the { } grouping in the C programming language.

initial

```
module stimulus;
        reg x,y, a,b, m;
        initial m = 1'b0;
//single statement does not need to be grouped
        initial
        begin
                #5 a = 1'b1;
                #25 b = 1'b0;
        end
 //multiple statements need to be grouped
        initial
        begin
                #10 x = 1'b0;
                #25 y = 1'b1;
        end
        initial
                #50 $finish;
endmodule
```

initial

- The initial blocks are typically used for initialization, monitoring, waveforms and other processes that must be executed only once during the entire simulation run.
- The values can be initialized using alternate shorthand syntax. The use of such shorthand syntax has the same effect as an initial block combined with a variable declaration.
- Combined Variable Declaration and Initialization

```
//The clock variable is defined first
        reg clock;
//The value of clock is set to 0
        initial clock = 0;
//Instead of the above method, clock variable can be
//initialized at the time of declaration. This is allowed
//only for variables declared //at module level.
        reg clock = 0;
```

initial

- Combined Port/Data Declaration and Initialization
  - The combined port/data declaration can also be combined with an initialization

```
module adder (sum, co, a, b, ci);
//Initialize 8 bit output sum output
      output reg [7:0] sum = 0;
//Initialize 1 bit output
      reg co = 0;
      co input [7:0] a, b;
      input ci;
      --
      --
endmodule
```

initial

- Combined ANSI C Style Port Declaration and Initialization
  - ANSI C style port declaration can also be combined with an initialization

```
module adder
      (output reg [7:0] sum = 0, //Initialize 8 bit output
      output reg co = 0, //Initialize 1 bit output co
      input [7:0] a, b,
      input ci );
      --
      --
endmodule
```

always

- All behavioral statements inside an always statement constitute an always block.
- The always statement starts at time 0 and executes the statements in the always block continuously in a looping fashion.
- This statement is used to model a block of activity that is repeated continuously in a digital circuit.
- An example is a clock generator module that toggles the clock signal every half cycle.
- In real circuits, the clock generator is active from time 0 to as long as the circuit is powered on.

always

```
module clock_gen (output reg clock);
//Initialize clock at time zero
initial
    clock = 1'b0;
//Toggle clock every half-cycle (time period = 20)
always
    #10 clock = ~clock;
initial
    #1000 $finish;
endmodule
```

- **The always statement starts at time 0 and executes the statement clock = ~clock every 10 time units.**

Procedural Assignments

- Procedural assignments update values of reg, integer, real, or time variables.
- The value placed on a variable will remain unchanged until another procedural assignment updates the variable with a different value.
- These are unlike continuous assignments, where one assignment statement can cause the value of the right-hand-side expression to be continuously placed onto the left-hand-side net.
- The syntax for the simplest form of procedural assignment is shown below.

```
assignment ::= variable_lvalue =
                    [delay_or_event_control ] expression
```

Procedural Assignments

- The left-hand side of a procedural assignment <lvalue> can be one of the following:
  - A reg, integer, real, or time register variable or a memory element
  - A bit select of these variables (e.g., addr[0])
  - A part select of these variables (e.g., addr[31:16])
  - A concatenation of any of the above
- The right-hand side can be any expression that evaluates to a value.
- In behavioral modeling, all operators discussed in previous lectures can be used in behavioral expressions.
- There are two types of procedural assignment statements:
  - blocking and nonblocking.

Blocking Assignments

- Blocking assignment statements are executed in the order they are specified in a sequential block.
- A blocking assignment will not block execution of statements that follow in a parallel block.
- The = operator is used to specify blocking assignments.

Blocking Assignments

```
reg x, y, z;
reg [15:0] reg_a, reg_b;
integer count;
//All behavioral statements must be inside an initial or always
block
initial
  begin
     x = 0; y = 1; z = 1; //Scalar assignments
     count = 0; //Assignment to integer variables
     reg_a = 16'b0; reg_b = reg_a; //initialize vectors
     #15 reg_a[2] = 1'b1; //Bit select assignment with delay
     #10 reg_b[15:13] = {x, y, z}
        //Assign result of concatenation to part select of a vector
     count = count + 1;
        //Assignment to an integer (increment) end
```

Non Blocking Assignments

- Nonblocking assignments allow scheduling of assignments without blocking execution of the statements that follow in a sequential block.
- A <= operator is used to specify nonblocking assignments.
- Note that this operator has the same symbol as a relational operator, less_than_equal_to.
- The operator <= is interpreted as a relational operator in an expression and as an assignment operator in the context of a nonblocking assignment.
- To illustrate the behavior of nonblocking statements and its difference from blocking statements, let us consider the example, where we convert some blocking assignments to nonblocking assignments, and observe the behavior.

Non Blocking Assignments

```
reg x, y, z;
reg [15:0] reg_a, reg_b;
integer count;
//All behavioral statements must be inside an initial or always block
    initial
      begin
          x = 0; y = 1; z = 1;
    //Scalar assignments
          count = 0; //Assignment to integer variables
          reg_a = 16'b0; reg_b = reg_a; //Initialize vectors
          reg_a[2] <= #15 1'b1; //Bit select assignment with delay
          reg_b[15:13] <= #10 {x, y, z};
            //Assign result of concatenation //to part select of a vector
          count <= count + 1;
            //Assignment to an integer (increment)
      end
```

Timing Controls

- Various behavioral timing control constructs are available in Verilog.
- In Verilog, if there are no timing control statements, the simulation time does not advance.
- Timing controls provide a way to specify the simulation time at which procedural statements will execute.
- There are three methods of timing control:
  - delay-based timing control,
  - event-based timing control, and
  - level-sensitive timing control.

Delay-based timing control

- **Syntax for the delay-based timing control statement is shown below.**

```
delay3 ::= # delay_value | # ( delay_value [ ,
                    delay_value [ , delay_value ] ] )
delay2 ::= # delay_value | # ( delay_value [ ,
                                        delay_value ] )
delay_value ::=
      unsigned_number |
      parameter_identifier |
      specparam_identifier |
      mintypmax_expression
```

Delay-based timing control

- Delay-based timing control in an expression specifies the time duration between when the statement is encountered and when it is executed.
- Delays are specified by the symbol #.
- Delay-based timing control can be specified by a number, identifier, or a min/typ/max_expression.
- There are three types of delay control for procedural assignments:
  - Regular delay control
  - Intra-assignment delay control
  - Zero delay control.

Delay-based timing control

- Regular delay control
  - Regular delay control is used when a non-zero delay is specified to the left of a procedural assignment.

```
//define parameters
    parameter latency = 20;
    parameter delta = 2;
//define register variables
    reg x, y, z, p, q;
initial
    begin
      x = 0; // no delay control
      #10 y = 1; // delay control with a number.
      #latency z = 0; // Delay control with identifier.
      #(latency + delta) p = 1; // Delay control with expression
    end
```

Delay-based timing control

- Intra-assignment delay control
  - Instead of specifying delay control to the left of the assignment, it is possible to assign a delay to the right of the assignment operator.
  - Such delay specification alters the flow of activity in a different manner.

Delay-based timing control

- **Intra-assignment delay control**

```
//define register variables
   reg x, y, z;
//intra assignment delays
   initial
     begin
          x = 0; z =  0;
           y = #5 x +  z;
//Take value of x and z at the time=0, evaluate x +  z
//and then wait 5 time units to assign value to y.
   end
```

Delay-based timing control

- Intra-assignment delay control
  - Note the difference between intra-assignment delays and regular delays.
  - Regular delays defer the execution of the entire assignment.
  - Intra-assignment delays compute the right-hand-side expression at the current time and defer the assignment of the computed value to the left-hand-side variable.
  - Intra-assignment delays are like using regular delays with a temporary variable to store the current value of a right-hand-side expression.

Delay-based timing control

- Zero delay control
  - Procedural statements in different always-initial blocks may be evaluated at the same simulation time.
  - The order of execution of these statements in different always-initial blocks is non-deterministic.
  - Zero delay control is a method to ensure that a statement is executed last, after all other statements in that simulation time are executed.
  - This is used to eliminate race conditions.
  - However, if there are multiple zero delay statements, the order between them is non-deterministic.

Delay-based timing control

- Zero delay control

```
initial
   begin
         x = 0; y = 0;
   end
initial
   begin
         #0 x = 1; //zero delay control
         #0 y = 1;
   end
```

Event-Based Timing Control

- An event is the change in the value on a register or a net.
- Events can be utilized to trigger execution of a statement or a block of statements.
- There are four types of event-based timing control:
  - Regular event control
  - Named event control
  - Event OR control
  - Level-sensitive timing control.

Event-Based Timing Control

- Regular event control
  - The @ symbol is used to specify an event control.
  - Statements can be executed on changes in signal value or at a positive or negative transition of the signal value.
  - The keyword posedge is used for a positive transition.
  - The keyword negedge is used for a positive transition.

Event-Based Timing Control

- **Regular event control**

```
        @(clock) q = d;
//q = d is executed whenever signal clock changes value
        @(posedge clock) q = d;
 //q = d is executed whenever signal clock does a positive
 //transition ( 0 to 1,x or z, // x to 1, z to 1 )
      @(negedge clock) q = d;
 //q = d is executed whenever signal clock does a
 //negative transition ( 1 to 0,x or z, //x to 0, z to 0)
      q = @(posedge clock) d;
 //d is evaluated immediately and assigned to q at the
 //positive edge of clock.
```

Event-Based Timing Control

- Named event control
  - VerilogHDL provides the capability to declare an event and then trigger and recognize the occurrence of that event.
  - The event does not hold any data.
  - A named event is declared by the keyword event.
  - An event is triggered by the symbol ->.
  - The triggering of the event is recognized by the symbol @.

Event-Based Timing Control

- Named event control

```
//This is an example of a data buffer storing data after  the
//last packet of data has arrived.
      event received_data; //Define an event called received_data
      always @(posedge clock) //check at each positive clock edge
                begin
                if(last_data_packet) //If this is the last data packet -
                >received_data; //trigger the event received_data end
                always @(received_data)
//Await triggering of event  received_data.
//When event is triggered, store all four packets of received data in data
//buffer use concatenation operator { }
                data_buf = {data_pkt[0], data_pkt[1], data_pkt[2], data_pkt[3]};
```

Event-Based Timing Control

- Event OR control
  - Sometimes a transition on any one of multiple signals or events can trigger the execution of a statement or a block of statements.
  - This is expressed as an OR of events or signals.
  - The list of events or signals expressed as an OR is also known as a sensitivity list.
  - The keyword or is used to specify multiple triggers,

Event-Based Timing Control

- Event OR control

```
//A level-sensitive latch with asynchronous reset
        always @( reset or clock or d)
//Wait for reset or clock or d to change
            begin
                if (reset)
//if reset signal is high, set q to 0.
                    q = 1'b0;
                else if(clock)
//if clock is high, latch input
                    q = d;
            end
```

Event-Based Timing Control

- Event OR control
  - Sensitivity lists can also be specified using the "," (comma) operator instead of the or operator.

```
//A level-sensitive latch with asynchronous reset
    always @( reset, clock, d)
//Wait for reset or clock or d to change
    begin
        if (reset) //if reset signal is high, set q to 0.
            q = 1'b0;
        else if(clock) //if clock is high, latch input
            q = d;
    end
```

Event-Based Timing Control

- Event OR control
  - Comma operators can also be applied to sensitivity lists that have edge-sensitive triggers.

```
//A positive edge triggered D flipflop with asynchronous
//falling reset can be modeled as shown below
   always @(posedge clk, negedge reset)
//Note use of comma operator
     if(!reset)
            q <=0;
   else
            q <=d;
```

Event-Based Timing Control

- Event OR control (Use of @* Operator)
  - When the number of input variables to a combination logic block are very large, sensitivity lists can become very cumbersome to write.
  - Moreover, if an input variable is missed from the sensitivity list, the block will not behave like a combinational logic block.
  - To solve this problem, VerilogHDL contains two special symbols: @* and @(*).
  - Both symbols exhibit identical behavior.
  - These special symbols are sensitive to a change on any signal that may be read by the statement group that follows this symbol

Event-Based Timing Control

- **Event OR control (Use of @* Operator)**

```
//Combination logic block using the or operator
//Cumbersome to write and it is easy to miss one input to the block
    always @(a or b or c or d or e or f or g or h or p or m)
            begin
                out1 = a ? b+c : d+e; out2 = f ? g+h : p+m;
            end
//Instead of the above method, use @(*) symbol. Alternately, the
//@* symbol can be used. All input variables are automatically
//included in the sensitivity list.
    always @(*)
            begin
                out1 = a ? b+c : d+e; out2 = f ? g+h : p+m;
            end
```

Event-Based Timing Control

- **Level-sensitive timing control.**
  - Event control discussed earlier waited for the change of a signal value or the triggering of an event.
  - The symbol @ provided edge-sensitive control.
  - VerilogHDL also allows level-sensitive timing control, that is, the ability to wait for a certain condition to be true before a statement or a block of statements is executed.
  - The keyword wait is used for level-sensitive constructs.

```
always wait (count_enable) #20 count = count +
1;
```

Conditional Statements

- Conditional statements are used for making decisions based upon certain conditions.
- These conditions are used to decide whether or not a statement should be executed.
- Keywords if and else are used for conditional statements.
- There are three types of conditional statements.
- Usage of conditional statements is shown below.

Conditional Statements

```
//Type 1 conditional statement. No else statement.
//Statement executes or does not execute.
        if (<expression>) true_statement ;
//Type 2 conditional statement. One else statement
//Either true_statement or false_statement is evaluated
        if (<expression>) true_statement ;
        else false_statement ;
//Type 3 conditional statement. Nested if-else-if.
//Choice of multiple statements. Only one is executed.
        if (<expression1>) true_statement1 ;
        else if (<expression2>) true_statement2 ;
        else if (<expression3>) true_statement3 ;
        else default_statement ;
```

Conditional Statements

- The <expression> is evaluated.
- If it is true (1 or a non-zero value), the true_statement is executed.
- However, if it is false (zero) or ambiguous (x), the false_statement is executed.
- The <expression> can contain any operators.
- Each true_statement or false_statement can be a single statement or a block of multiple statements.
- A block must be grouped, typically by using keywords begin and end.
- A single statement need not be grouped.

Conditional Statements

```
//Type 1 statements
    if(!lock) buffer = data;
    if(enable) out = in;
//Type 2 statements
    if (number_queued < MAX_Q_DEPTH)
        begin
                data_queue = data;
                number_queued = number_queued + 1;
        end
    else
        $display("Queue Full. Try again");
```

Conditional Statements

```
//Type 3 statements
//Execute statements based on ALU control signal.
   if (alu_control == 0)
        y = x + z;
   else if(alu_control == 1)
        y = x - z;
   else if(alu_control == 2)
        y = x * z;
   else
        $display("Invalid ALU control signal");
```

Multiway Branching

- The nested if-else-if can become unwieldy if there are too many alternatives. A shortcut to achieve the same result is to use the case statement.
- The keywords case, endcase, and default are used in the case statement.

```
case (expression)
    alternative1: statement1;
    alternative2: statement2;
    alternative3: statement3;
            ...
            ...
    default: default_statement;
endcase
```

Multiway Branching

- Each of statement1, statement2 …, default_statement can be a single statement or a block of multiple statements.
- A block of multiple statements must be grouped by keywords begin and end.
- The expression is compared to the alternatives in the order they are written.
- For the first alternative that matches, the corresponding statement or block is executed.
- If none of the alternatives matches, the default_statement is executed.
- The default_statement is optional.
- Placing of multiple default statements in one case statement is not allowed.
- The case statements can be nested.

Multiway Branching

```
//Execute statements based on the ALU control signal
  reg [1:0] alu_control;
      ...
      ...
  case (alu_control)
      2'd0 : y = x + z;
      2'd1 : y = x - z;
      2'd2 : y = x *  z;
      default : $display("Invalid ALU control signal");
  endcase
```

4-to-1 Multiplexer (case)

```
    module mux4_to_1 (out, i0, i1, i2, i3, s1,  s0);
// Port declarations from the I/O diagram
    output out;
    input i0, i1, i2, i3;
    input s1, s0;
    reg out;
    always @(s1  or s0 or i0 or i1 or i2 or i3)
        case ({s1, s0}) //Switch based on concatenation of control
    signals
                2'd0 : out = i0;
                2'd1 : out = i1;
                2'd2 : out = i2;
                2'd3 : out =  i3;
        default: $display("Invalid control signals");
        endcase
    endmodule
```

Loops

- There are four types of looping statements in Verilog:
  - while, for, repeat, and forever.
- The syntax of these loops is very similar to the syntax of loops in the C programming language.
- All looping statements can appear only inside an initial or always block.
- Loops may contain delay expressions.

while loop

- The keyword while is used to specify this loop.
- The while loop executes until the while-expression is not true.
- If the loop is entered when the while-expression is not true, the loop is not executed at all.
- Each expression can contain the operators discussed already.
- Any logical expression can be specified with these operators.
- If multiple statements are to be executed in the loop, they must be grouped typically using keywords begin and end.

while loop

```
//Illustration 1: Increment count from 0 to 127.
//Exit at count 128.
//Display the count variable.
    integer count;
    initial
      begin
          count = 0;
              while (count < 128)
//Execute loop till count is 127. Exit at count 128
              begin
                    $display("Count = %d", count);
                    count = count + 1;
              end
       end
```

while loop

```
//Illustration 2: Find the first bit with a value 1 in flag (vector variable)
    'define TRUE 1'b1';
    'define FALSE 1'b0;
    reg [15:0] flag;
    integer i; //integer to keep count
    reg continue;
    initial
       begin
          flag = 16'b 0010_0000_0000_0000;
          i = 0;
          continue = 'TRUE;
          while((i < 16) && continue ) //Multiple conditions using operators.
          begin
            if (flag[i])
            begin
              $display("Encountered a TRUE bit at element number %d", i);
             continue = 'FALSE;
            end
            i = i + 1;
          end
      end
```

for loop

- The keyword for is used to specify this loop. The for loop contains three parts:
  - An initial condition
  - A check to see if the terminating condition is true
  - A procedural assignment to change value of the control variable
- The initialization condition and the incrementing procedural assignment are included in the for loop and do not need to be specified separately.
- Thus, the for loop provides a more compact loop structure than the while loop.
  - The while loop is more general-purpose than the for loop.
  - The for loop cannot be used in place of the while loop in all situations.

for loop

```
//Initialize array elements
    'define MAX_STATES 32
    integer state [0: 'MAX_STATES-1];
//Integer array state with elements 0:31
    integer i;
    initial
      begin
         for(i = 0; i < 32; i = i + 2)
//initialize all even locations with 0
              state[i] = 0;
         for(i = 1; i < 32; i = i + 2)
//initialize all odd locations with 1
              state[i] = 1;
       end
```

repeat loop

- The keyword repeat is used for this loop.
- The repeat construct executes the loop a fixed number of times.
- A repeat construct cannot be used to loop on a general logical expression.
- A while loop is used for that purpose. A repeat construct must contain a number, which can be a constant, a variable or a signal value.
- However, if the number is a variable or signal value, it is evaluated only when the loop starts and not during the loop execution.

repeat loop

```
//Illustration 1 : increment and display  count
//from 0 to 127 integer count;
        initial
             begin
                count = 0;
                repeat(128)
                begin
                    $display("Count = %d", count);
                    count = count + 1;
                end
            end
```

repeat loop

//Illustration 2 : Data buffer module example

//After it receives a data_start signal.

//Reads data for next 8 cycles.

module data_buffer(data_start, data, clock); parameter cycles = 8; input data_start; input [15:0] data; input clock; reg [15:0] buffer [0:7]; integer i; always @(posedge clock) begin if(data_start) //data start signal is true begin i = 0; repeat(cycles) //Store data at the posedge of next 8 clock //cycles begin @(posedge clock) buffer[i] = data; //waits till next // posedge to latch data i = i + 1; end end end endmodule

forever loop

- The keyword repeat is used for this loop.
- The repeat construct executes the loop a fixed number of times.
- A repeat construct cannot be used to loop on a general logical expression.
- A while loop is used for that purpose. A repeat construct must contain a number, which can be a constant, a variable or a signal value.
- However, if the number is a variable or signal value, it is evaluated only when the loop starts and not during the loop execution.

forever loop

```
//Example 1: Clock generation
//Use forever loop instead of always block
    reg clock;
    initial
          begin
                clock = 1'b0;
                forever #10 clock = ~clock;
//Clock with period of 20 units end
//Example 2: Synchronize two register values at every
//positive edge of clock
    reg clock;
    reg x, y;
    initial
          forever @(posedge clock) x = y;
```

Full Adder

```
module fa_bhv (A, B, CI, S, CO) ;

input A, B, CI;
output S, CO;
reg S, CO;
// use procedural assignments
always@(A or B or CI)
   begin
       S = A ^ B ^ CI;
       CO = (A & B) | (A & CI) | (B & CI);
    end
endmodule
```

4-to-1 Multiplexer

```
// 4-to-1 multiplexer.
//Port list is taken exactly from the I/O diagram.
    module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);
//Port declarations from the I/O diagram
    output out;
    input i0, i1, i2, i3;
    input s1, s0;
//output declared as register
    reg out;
//recompute the signal out if any input signal changes.
```

4-to-1 Multiplexer

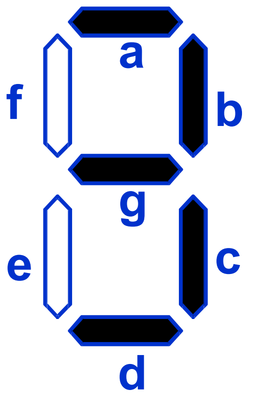
```
//All input signals that cause a recomputation of out to
//occur must go into the always @(...) sensitivity list.
    always @(s1  or s0 or i0 or i1 or i2 or i3)
    begin
        case ({s1, s0})
            2'b00: out = i0;
            2'b01: out = i1;
            2'b10: out = i2;
            2'b11: out = i3;
           default: out = 1'bx;
        endcase
    end
    endmodule
```

8-to-3 Encoder

```
module encoder (output reg [2:0] Code, input [7:0] Data);
always @  (Data)
    begin
      // encode the data
      if (Data == 8'b00000001) Code = 3’d0;
      else if (Data == 8'b00000010) Code = 3’d1;
    else if (Data == 8'b00000100) Code = 3’d2;
    else if (Data == 8'b00001000) Code = 3’d3;
    else if (Data == 8'b00010000) Code = 3’d4;
    else if (Data == 8'b00100000) Code = 3’d5;
    else if (Data == 8'b01000000) Code = 3’d6;
    else if (Data == 8'b10000000) Code = 3’d7;
    else Code = 3'bxxx; // invalid, so don’t care
    end
endmodule
```

Seven Segment Display

```
module Seven_Seg_Display (Display, BCD, Blanking);
  output         reg [6: 0]      Display;          // abc_defg
  input  [3: 0]                 BCD;
  input                         Blanking;
  parameter      BLANK = 7'b111_1111;              // active low
  parameter      ZERO  =  7'b000_0001;             // h01
  parameter      ONE    =   7'b100_1111;           // h4f
  parameter      TWO =7'b001_0010;                 // h12
  parameter      THREE = 7'b000_0110;              // h06
  parameter      FOUR   =  7'b100_1100;            // h4c
  parameter      FIVE    =   7'b010_0100;          // h24
  parameter      SIX  = 7'b010_0000;               // h20
  parameter      SEVEN = 7'b000_1111;              // h0f
  parameter      EIGHT  = 7'b000_0000;             // h00
  parameter      NINE     = 7'b000_0100;           // h04
```



Defined constants – can make code more understandable!

Seven Segment Display

```
always @ (BCD or Blanking)
    if (Blanking) Display = BLANK;
    else
      case (BCD)
          4’d0:               Display = ZERO;
          4’d1:               Display = ONE;
          4’d2:               Display = TWO;
          4’d3:               Display = THREE;
          4’d4:               Display = FOUR;
          4’d5:               Display = FIVE;
          4’d6:               Display = SIX;
          4’d7:               Display = SEVEN;
          4’d8:               Display = EIGHT;
          4’d9:               Display = NINE;
          default:            Display = BLANK;
       endcase
endmodule
```

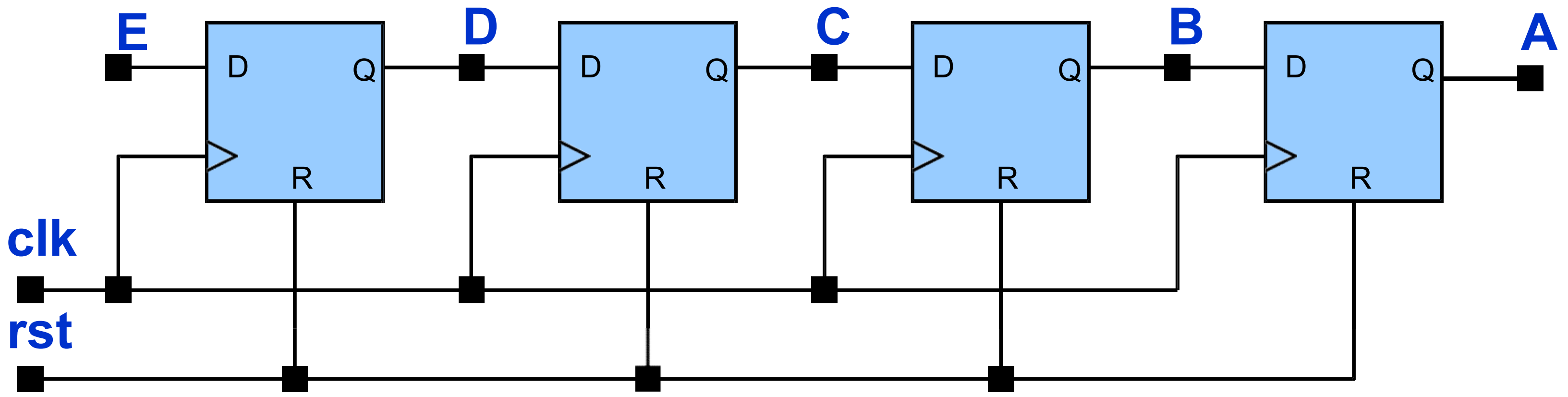
D - FF

```
   module dff(q, d, clk);
   output q;
   reg q,
   input d, clk;
     always @(posedge clk)
     begin
          q <= d;
     end
endmodule
```

D - FF

```
module dff(q, d, clk);
output q;
reg q,
input d, clk;
 always @(posedge clk, posedge reset)
 begin
    if (reset)
            q <= d;
       else
            q <= 0;
 end
endmodule
```

4-bit Shift Register



4-bit Shift Register

```
module shiftreg(E, A, clk, rst);
 output      A;
 input       E;
 input       clk, rst;
 reg A, B, C, D;
 always @ (posedge clk or posedge rst)
 begin
  if (rst)
     begin
       A <= 0; B <= 0; C <= 0; D <=0;
     end
  else
     begin
       D = E; C = D; B = C; A =B;
     end
endmodule
```

4-bit Counter

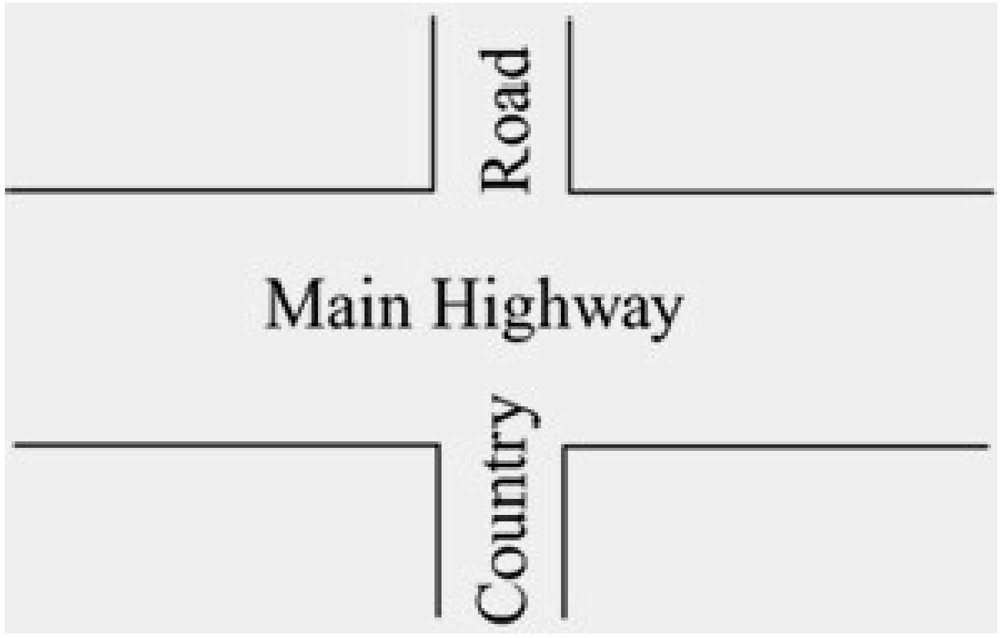
```
//4-bit Binary counter
    module counter(Q , clock, clear);
// I/O ports
    output [3:0] Q;
    input clock, clear;
//output defined as register
    reg [3:0] Q;
    always @( posedge clear or negedge clock)
    begin
    if (clear)
                Q <= 4'd0;
```

4-bit Counter

```
//Nonblocking assignments are recommended
//for creating sequential logic such as flipflops
          else
                Q <= Q + 1;
// Modulo 16 is not necessary because Q is a
//4-bit value and wraps around.
    end
    endmodule
```

Traffic Signal Controller

- Design a traffic signal controller, using a finite state machine approach.
- Consider a controller for traffic at the intersection of a main highway and a country road.



Traffic Signal Controller

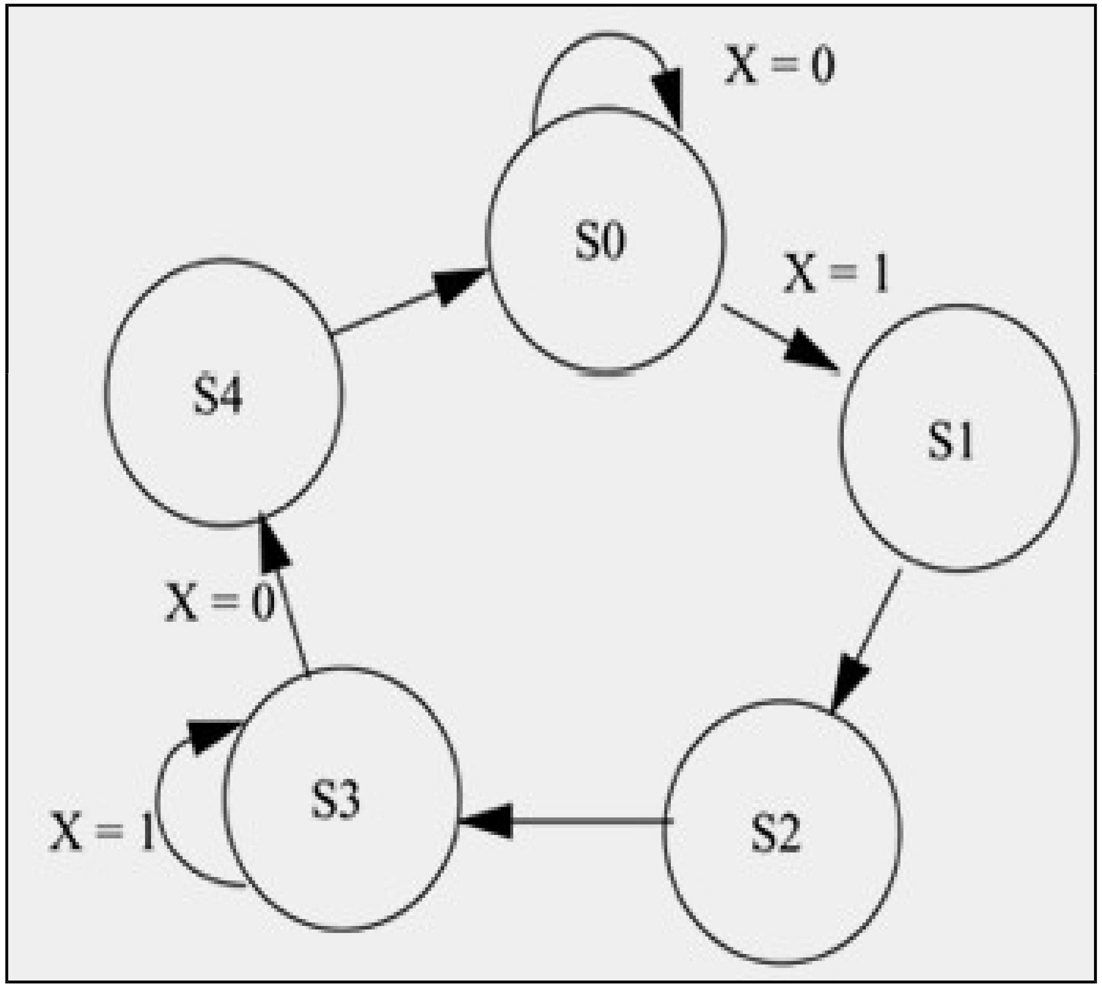
- The following specifications must be considered:
  - The traffic signal for the main highway gets highest priority because cars are continuously present on the main highway. Thus, the main highway signal remains green by default.
  - Occasionally, cars from the country road arrive at the traffic signal. The traffic signal for the country road must turn green only long enough to let the cars on the country road go.
  - As soon as there are no cars on the country road, the country road traffic signal turns yellow and then red and the traffic signal on the main highway turns green again.
  - There is a sensor to detect cars waiting on the country road. The sensor sends a signal X as input to the controller. X = 1 if there are cars on the country road; otherwise, X= 0.
  - There are delays on transitions from S1 to S2, from S2 to S3, and from S4 to S0. The delays must be controllable.

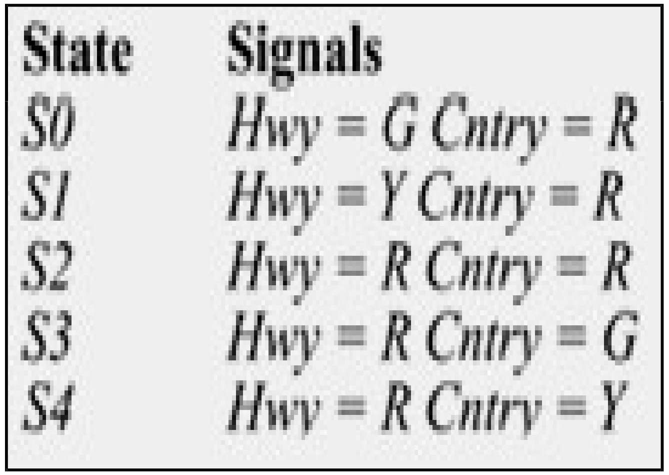
Traffic Signal Controller

- Specifications
  - There is a sensor to detect cars waiting on the country road. The sensor sends a signal X as input to the controller. X = 1 if there are cars on the country road; otherwise, X= 0.
  - There are delays on transitions from S1 to S2, from S2 to S3, and from S4 to S0. The delays must be controllable.

Traffic Signal Controller

FSM for Traffic Signal Controller





State	Signals
S0	*Hwy = G Cntry = R*
S1	*Hwy = Y Cntry = R*
S2	*Hwy = R Cntry = R*
S3	*Hwy = R Cntry = G*
S4	*Hwy = R Cntry = Y*

Traffic Signal Controller

```
    `define TRUE 1'b1
    `define FALSE 1'b0
//Delays
    `define Y2RDELAY 3
//Yellow to red delay
    `define R2GDELAY 2 /
/Red to green delay
    module sig_control (hwy, cntry, X, clock, clear);
//I/O ports
    output [1:0] hwy, cntry;
//2-bit output for 3 states of signal
//GREEN, YELLOW, RED;
    reg [1:0] hwy, cntry;
```

```
//declared output signals are registers
    input X;
//if TRUE, indicates that there is car on the  country
//road, otherwise FALSE
    input clock, clear;
    parameter RED = 2'd0,
                  YELLOW = 2'd1,
                  GREEN = 2'd2;
//State definition                HWY          CNTRY
    parameter  S0 = 3'd0,  //GREEN         RED
               S1 = 3'd1,  //YELLOW        RED
               S2 = 3'd2,  //RED           RED
               S3 = 3'd3,  //RED           GREEN
               S4 = 3'd4;  //RED           YELLOW
```

```
//Internal state variables
    reg [2:0] state;
    reg [2:0] next_state;
//state changes only at positive edge of clock
    always @(posedge clock)
          if (clear)
               state <= S0; //Controller starts in S0 state
          else
               state <= next_state; //State change
//Compute values of main signal and country signal
    always @(state)
    begin
//Default Light Assignment for Highway light
          hwy = GREEN;
```

```
//Default Light Assignment for Country light
        cntry = RED;
        case(state)
                S0: ; // No change, use default
                S1: hwy = YELLOW;
                S2: hwy = RED;
                S3: begin
                        hwy = RED;
                        cntry = GREEN;
                     end
                S4: begin
                        hwy = RED;
                        cntry = `YELLOW;
             end
        endcase
   end
```

```
//State machine using case statements
    always @(state or X)
    begin
          case (state)
             S0: if(X)
                      next_state = S1;
                    else
                      next_state = S0;
             S1: begin //delay some positive edges of clock
                       repeat(`Y2RDELAY) @(posedge clock) ;
                       next_state = S2;
                     end
             S2: begin //delay some positive edges of clock
                        repeat(`R2GDELAY) @(posedge clock);
                        next_state = S3;
                     end
```

```
  S3:  if(X)
          next_state = S3;
       else
          next_state = S4;
  S4: begin //delay some positive edges of clock
         repeat(`Y2RDELAY) @(posedge clock) ;
         next_state = S0;
       end
  default: next_state = S0;
endcase
end
endmodule
```

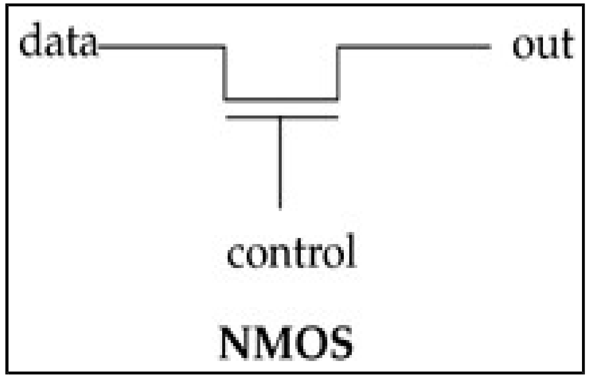
Lecture – 8
Switch-Level Modeling

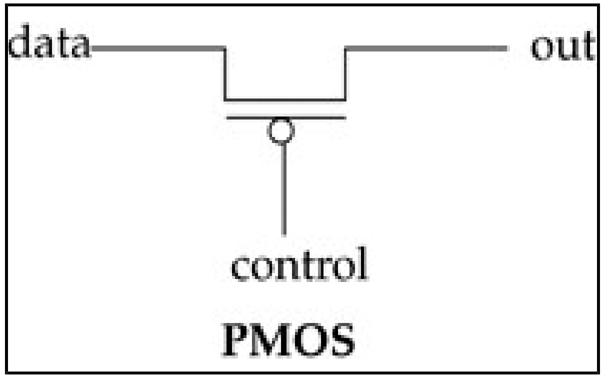
- Switch-Modeling Elements
- Examples
  - CMOS NOR Gate
  - 2-to-1 Multiplexer
  - CMOS Inverter
  - CMOS Flipflop

Switch-Modeling Elements

MOS Switches

- Verilog provides various constructs to model switch-level circuits. Digital circuits at MOS-transistor level are described using these elements
- MOS Switches
  - Two types of MOS switches can be defined with the keywords nmos and pmos





Switch-Modeling Elements

Instantiation of NMOS and PMOS Switches

- In VerilogHDL, nmos and pmos switches are instantiated as follows

  ```
  nmos n1(out, data, control); //instantiate a nmos switch
  pmos p1(out, data, control); //instantiate a pmos switch
  ```

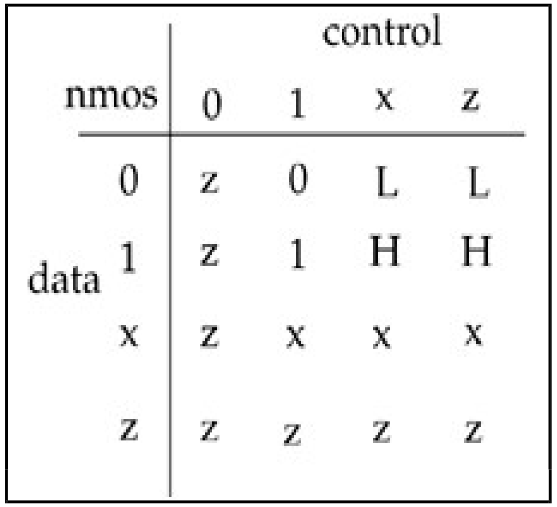
- Since switches are Verilog primitives, like logic gates, the name of the instance is optional. Therefore, it is acceptable to instantiate a switch without assigning an instance name.

  ```
  nmos (out, data, control); //no instance name
  pmos (out, data, control); //no instance name
  ```

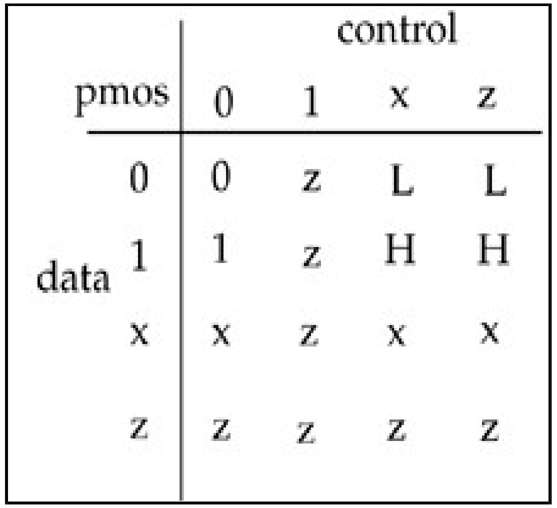
Switch-Modeling Elements

Logic Tables for NMOS and PMOS

- **The value of the out signal is determined from the values of data and control signals.**
  - **The nmos switch conducts when its control signal is 1. If the control signal is 0, the output assumes a high impedance value.**
  - **Similarly, a pmos switch conducts if the control signal is 0.**
- **Logic tables for out are shown in table. Some combinations of data and control signals cause the gates to output to either a 1 or 0, or to an z value without a preference for either value. The symbol L stands for 0 or z; H stands for 1 or z.**



		control			
	nmos	0	1	x	z
data	0	z	0	L	L
	1	z	1	H	H
	x	z	x	x	x
	z	z	z	z	z

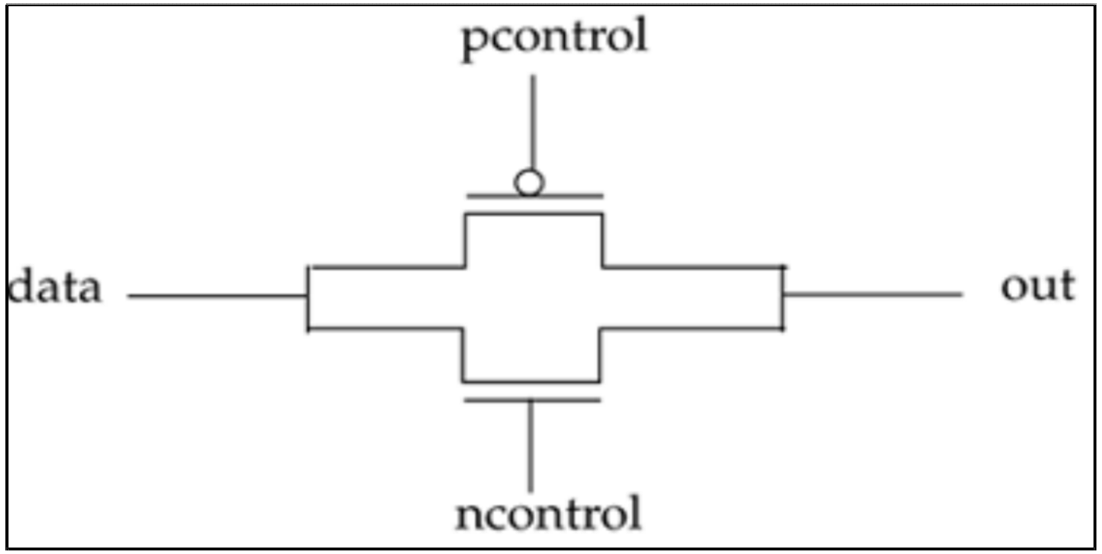


		control			
	pmos	0	1	x	z
data	0	0	z	L	L
	1	1	z	H	H
	x	x	z	x	x
	z	z	z	z	z

Switch-Modeling Elements

CMOS Switches

- **CMOS switches are declared with the keyword cmos**
- **A cmos device can be modeled with a nmos and a pmos device. The symbol for a cmos switch is shown in figure.**



Switch-Modeling Elements

Instantiation of CMOS Switches

- A cmos switch is instantiated as shown below.

  cmos c1(out, data, ncontrol, pcontrol);//instantiate cmos gate.

  cmos (out, data, ncontrol, pcontrol); //no instance name given.
- The ncontrol and pcontrol are normally complements of each other.
  - When the ncontrol signal is 1 and pcontrol signal is 0, the switch conducts.
  - If ncontrol signal is 0 and pcontrol is 1, the output of the switch is high impedance value.
- The cmos gate is essentially a combination of two gates: one nmos and one pmos. Thus the cmos instantiation shown above is equivalent to the following:

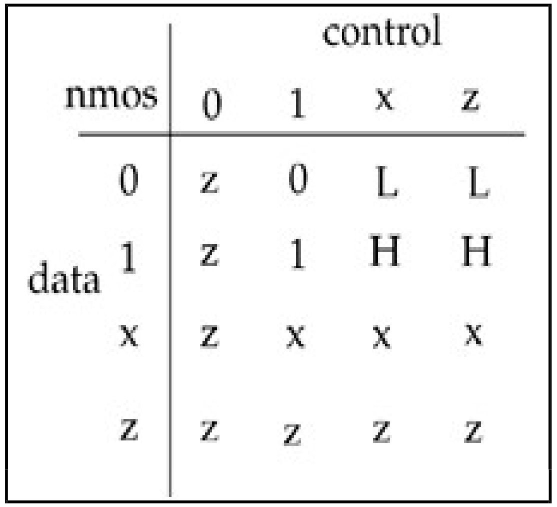
  nmos (out, data, ncontrol); //instantiate a nmos switch

  pmos (out, data, pcontrol); //instantiate a pmos switch

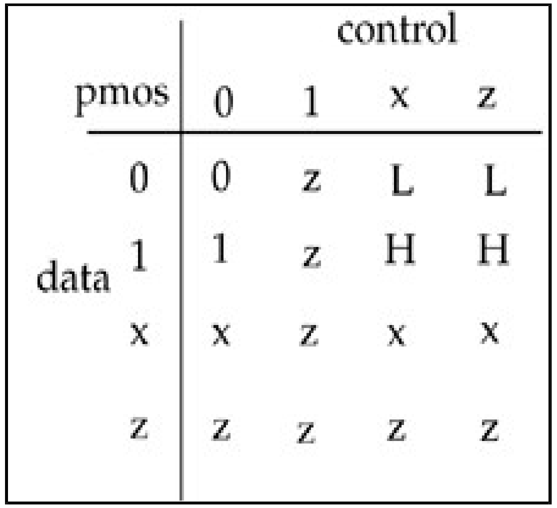
Switch-Modeling Elements

Logic Tables for CMOS

- **Since a cmos switch is derived from nmos and pmos switches, it is possible to derive the output value from table given values of data, ncontrol, and pcontrol signals.**
- **Some combinations of data and control signals cause the gates to output to either a 1 or 0, or to an z value without a preference for either value. The symbol L stands for 0 or z; H stands for 1 or z.**



		control			
	nmos	0	1	x	z
data	0	z	0	L	L
	1	z	1	H	H
	x	z	x	x	x
	z	z	z	z	z

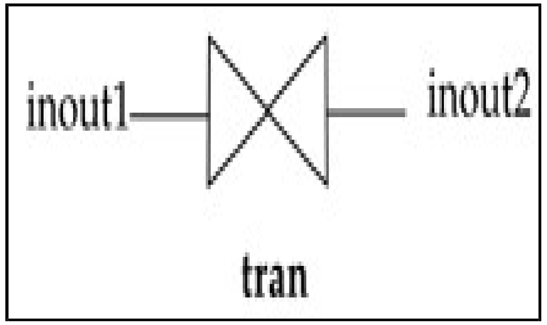


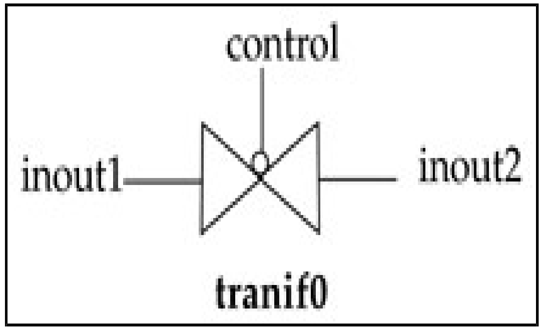
		control			
	pmos	0	1	x	z
data	0	0	z	L	L
	1	1	z	H	H
	x	x	z	x	x
	z	z	z	z	z

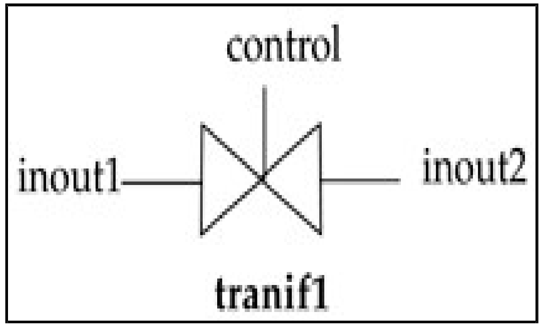
Switch-Modeling Elements

Bidirectional Switches

- NMOS, PMOS and CMOS gates conduct from drain to source.
- It is important to have devices that conduct in both directions. In such cases, signals on either side of the device can be the driver signal. Bidirectional switches are provided for this purpose.
- Three keywords are used to define bidirectional switches: tran, tranif0, and tranif1.
- Symbols for these switches are shown in figure.







Switch-Modeling Elements

Instantiation of Bidirectional Switches

- The tran switch acts as a buffer between the two signals inout1 and inout2. Either inout1 or inout2 can be the driver signal.

  tran t1(inout1, inout2); //instance name t1 is optional

- The tranif0 switch connects the two signals inout1 and inout2 only if the control signal is logical 0. If the control signal is a logical 1, the nondriver signal gets a high impedance value z. The driver signal retains value from its driver.

  tranif0 (inout1, inout2, control

- The tranif1 switch conducts if the control signal is a logical 1.

  tranif1 (inout1, inout2, control);

- Bidirectional switches are typically used to provide isolation between buses or signals

Switch-Modeling Elements

Power and Ground

- The power (Vdd, logic 1) and Ground (Vss, logic 0) sources are defined with keywords supply1 and supply0.
  - Sources of type supply1 are equivalent to Vdd in circuits and place a logical 1 on a net.
  - Sources of the type supply0 are equivalent to ground or Vss and place a logical 0 on a net.
  - Both supply1 and supply0 place logical 1 and 0 continuously on nets throughout the simulation.
- Sources supply1 and supply0 are shown below.

```
supply1 vdd;
supply0 gnd;
assign a = vdd; //Connect a to vdd
assign b = gnd; //Connect b to gnd
```

Switch-Modeling Elements

Resistive Switches

- MOS, CMOS, and bidirectional switches can be modeled as corresponding resistive devices.
- Resistive switches have higher source-to-drain impedance than regular switches and reduce the strength of signals passing through them.
- Resistive switches are declared with keywords that have an "r" prefixed to the corresponding keyword for the regular switch.
- Resistive switches have the same syntax as regular switches.

```
rnmos   rpmos    //resistive nmos and pmos switches
rcmos            //resistive cmos switch
rtran  rtranif0   rtranif1 //resistive bidirectional switches.
```

Switch-Modeling Elements

Resistive Switches

- There are two main differences between regular switches and resistive switches
  - Their source-to-drain impedances and the way they pass signal strengths.
- Resistive devices have a high source-to-drain impedance. Regular switches have a low source-to-drain impedance.
- Resistive switches reduce signal strengths when signals pass through them. Regular switches retain strength levels of signals from input to output. The exception is that if the input is of strength supply, the output is of strong strength.

Switch-Modeling Elements

Resistive Switches

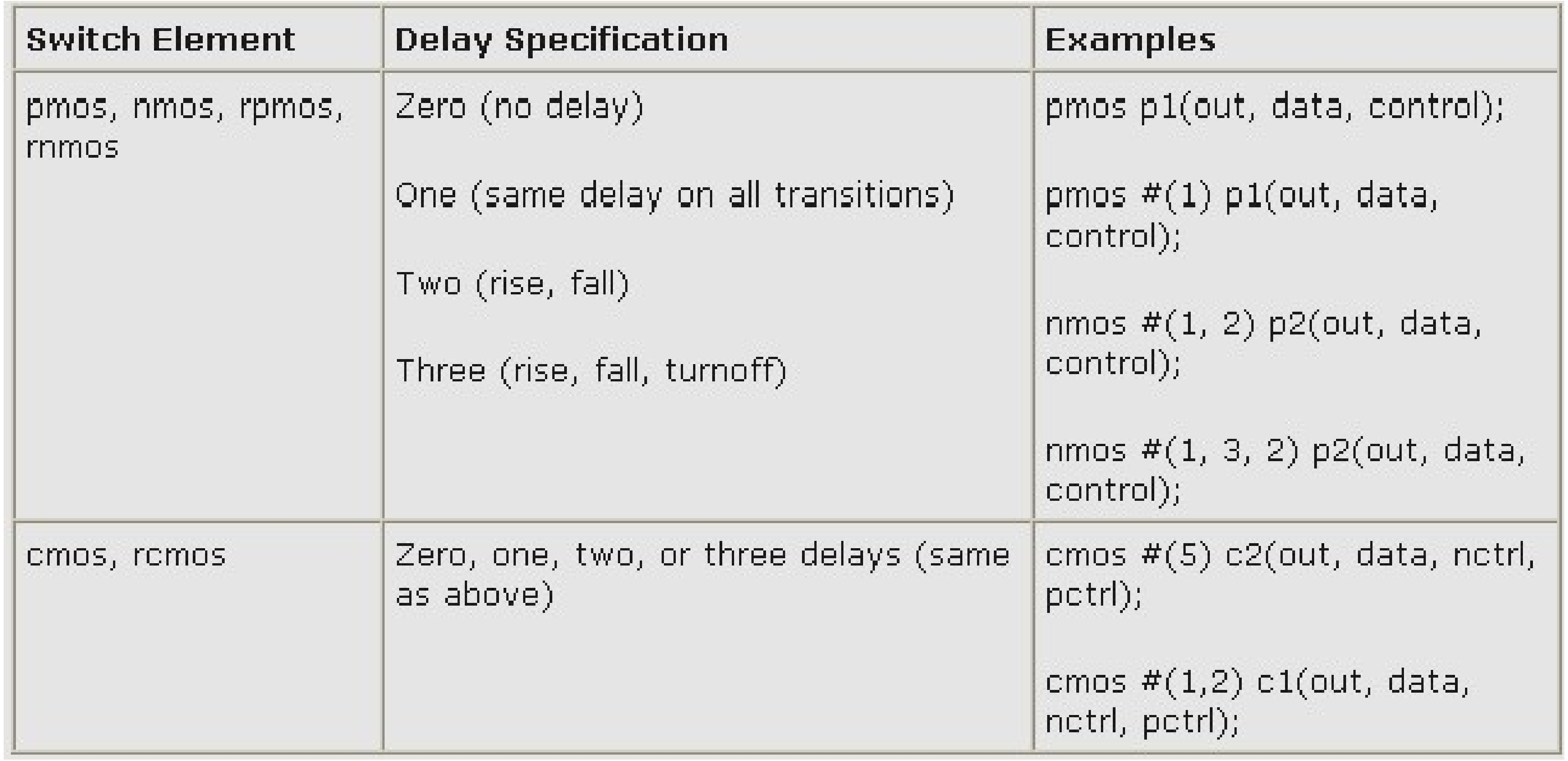
- Table shows the strength reduction due to resistive switches.

Input Strength	Output Strength
supply	pull
strong	pull
pull	weak
weak	medium
large	medium
medium	small
small	small
high	high

Switch-Modeling Elements

Delay Specification on MOS and CMOS Switches

- **Delays can be specified for signals that pass through these switch-level elements. Delays are optional and appear immediately after the keyword for the switch.**



Switch Element	Delay Specification	Examples

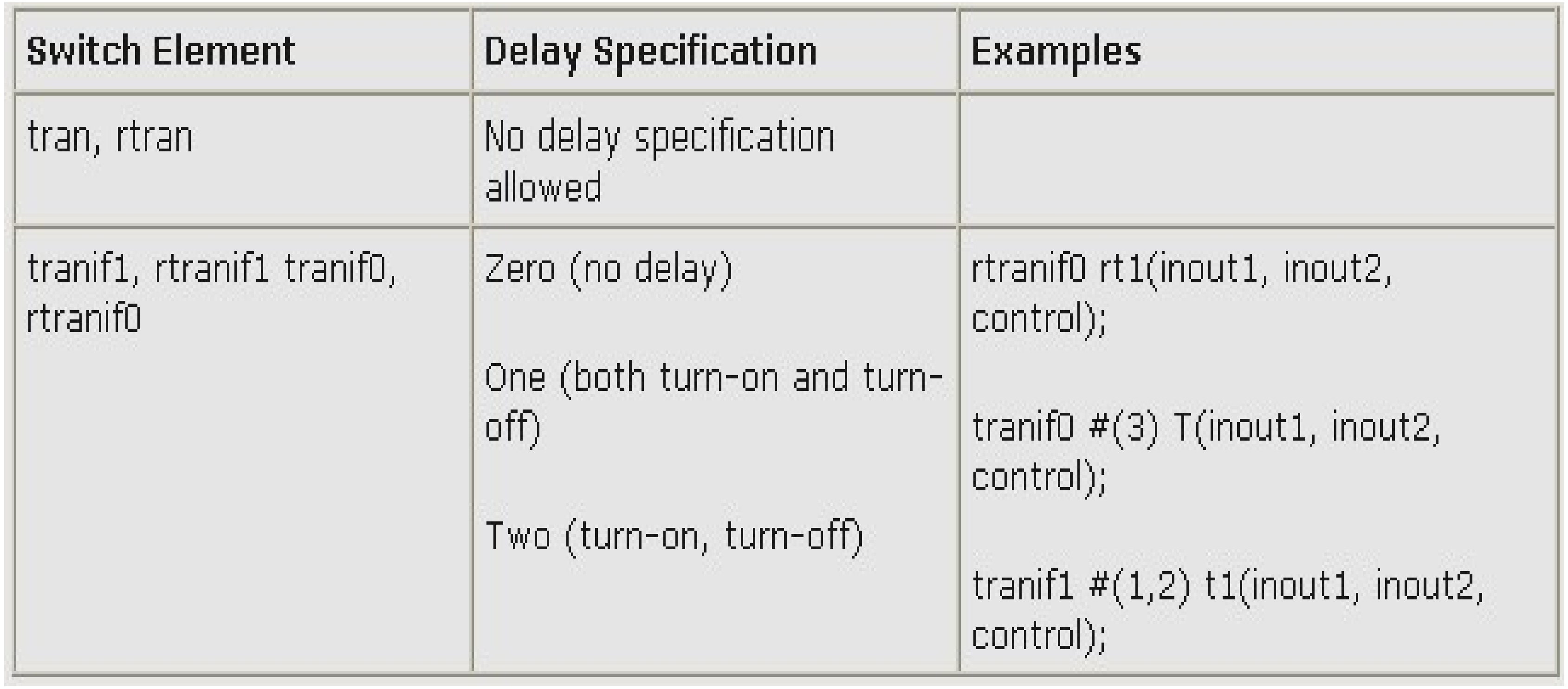
pmos, nmos, rpmos, rnmos	Zero (no delay)
	
	One (same delay on all transitions)
	
	Two (rise, fall)
	
	Three (rise, fall, turnoff)
		pmos p1(out, data, control);
		
		pmos #(1) p1(out, data, control);
		
		nmos #(1, 2) p2(out, data, control);
		
		nmos #(1, 3, 2) p2(out, data, control);

cmos, rcmos	Zero, one, two, or three delays (same as above)	cmos #(5) c2(out, data, nctrl, pctrl);
		
		cmos #(1,2) c1(out, data, nctrl, pctrl);

Switch-Modeling Elements

Delay Specification for Bidirectional Switches

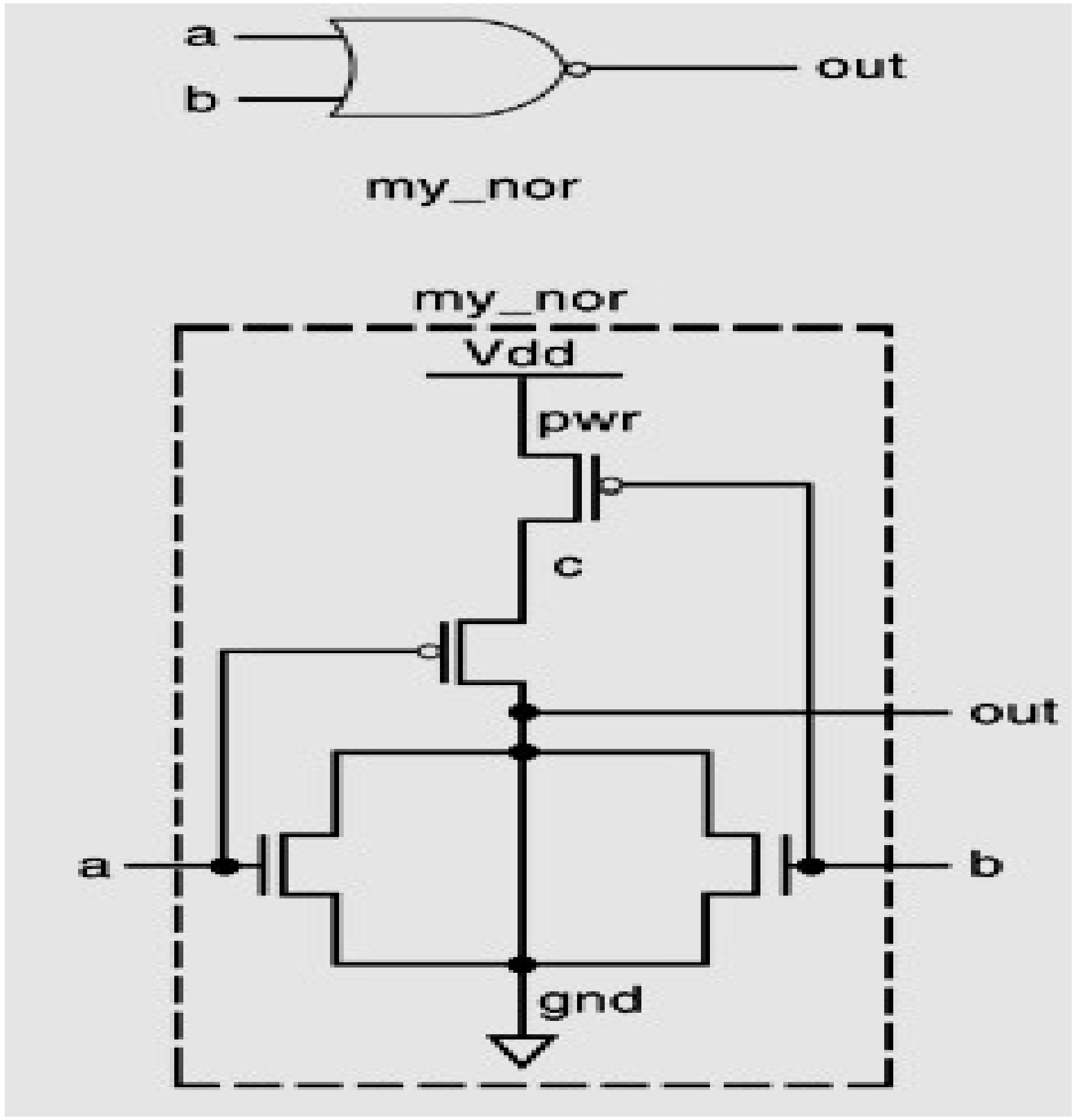
- **Delay specification is interpreted slightly differently for bidirectional pass switches. These switches do not delay signals passing through them. Instead, they have turn-on and turn-off delays while switching.**



Switch Element	Delay Specification	Examples
tran, rtran	No delay specification allowed	

tranif1, rtranif1 tranif0, rtranif0	Zero (no delay)
	
	One (both turn-on and turn-off)
	
	Two (turn-on, turn-off)
		rtranif0 rt1(inout1, inout2, control);
		
		tranif0 #(3) T(inout1, inout2, control);
		
		tranif1 #(1,2) t1(inout1, inout2, control);

CMOS NOR Gate



CMOS NOR Gate – Source Code

```
//Define our own nor gate, my_nor
module my_nor(out, a, b);
    output out;
    input a, b;
    //internal wires
    wire c;
    //set up power and ground lines
    supply1 pwr; //pwr is connected to Vdd (power supply)
    supply0 gnd; //gnd is connected to Vss(ground)
    //instantiate pmos switches
    pmos (c, pwr, b);
    pmos (out, c, a);
    //instantiate nmos switches
    nmos (out, gnd, a);
    nmos (out, gnd, b);
endmodule
```

CMOS NOR Gate – Stimulus File

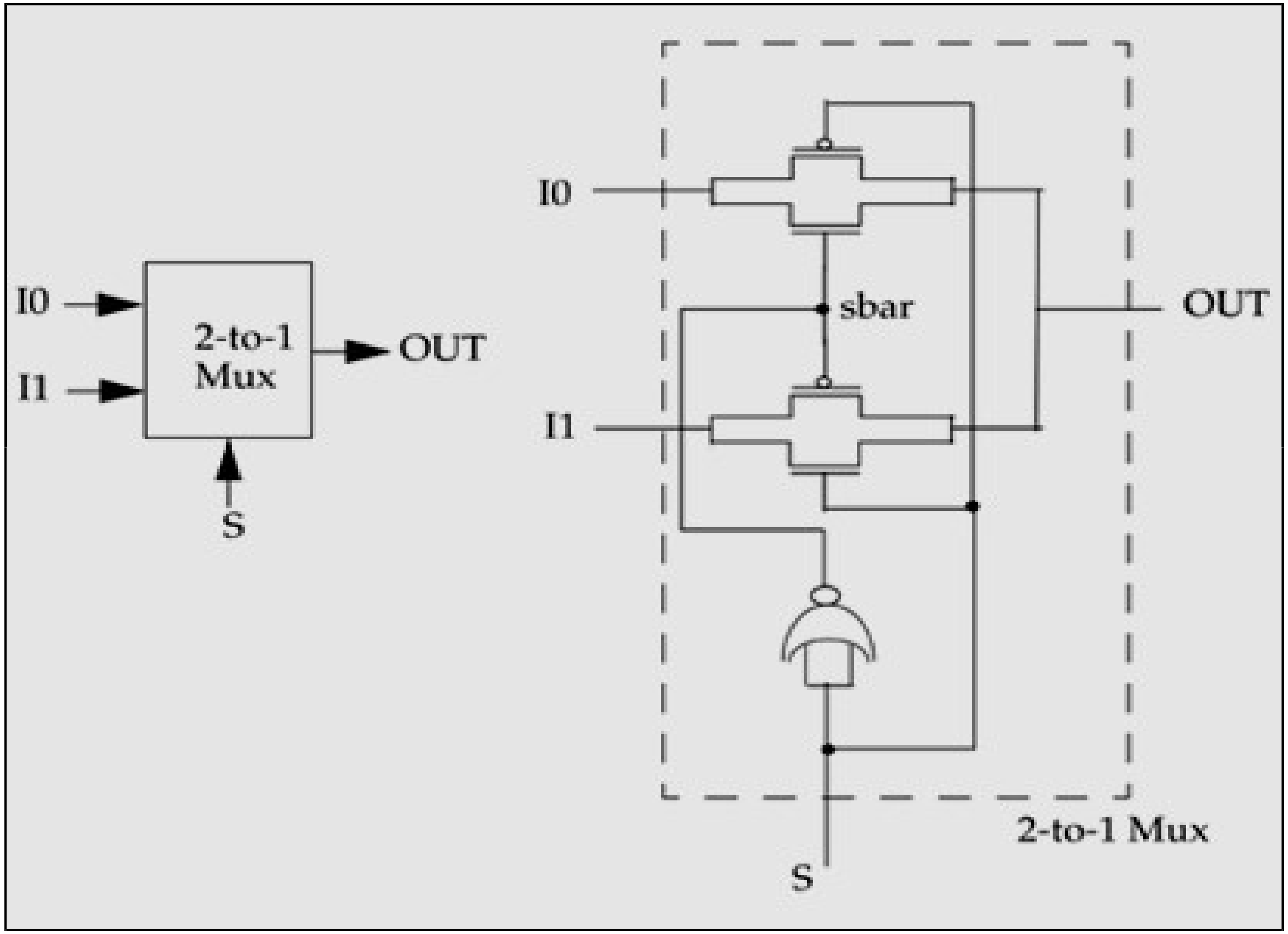
```
//stimulus to test the gate
module stimulus;
        reg A, B;
        wire OUT;
        //instantiate the my_nor module
        my_nor n1(OUT, A, B);
        //Apply stimulus
        initial
        begin
        //test all possible combinations
        A = 1'b0; B = 1'b0;
        #5 A = 1'b0; B = 1'b1;
        #5 A = 1'b1; B = 1'b0;
        #5 A = 1'b1; B =1'b1;
end
```

```
    //check results
    initial
    $monitor($time, " OUT = %b, A = %b, B = %b", OUT, A, B);
endmodule
```

The output of the simulation is shown below.

```
0       OUT = 1, A = 0, B =0
5       OUT = 0, A = 0, B =1
10      OUT = 0, A = 1, B =0
15      OUT = 0, A = 1, B =1
```

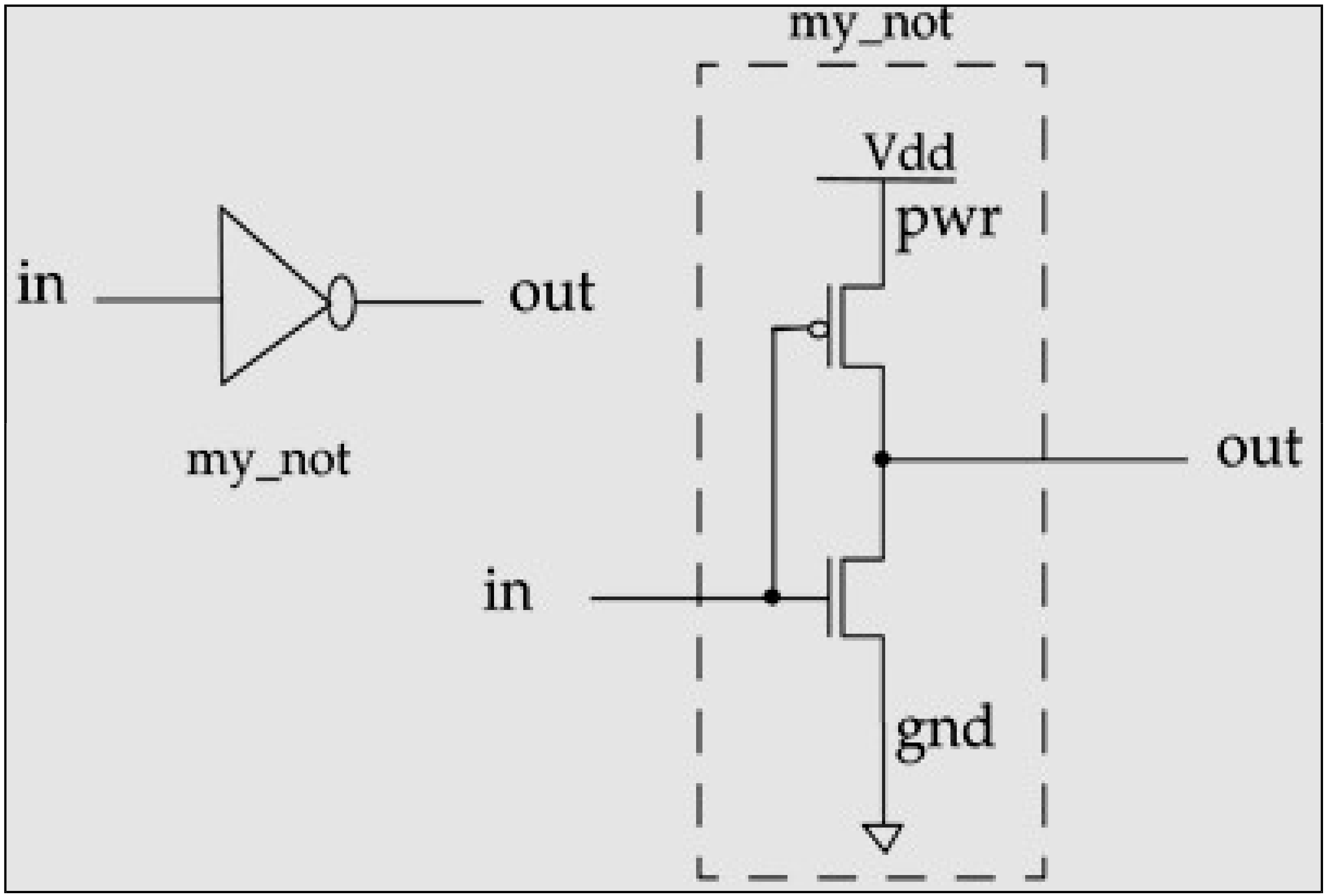
2-to-1 Multiplexer



2-to-1 Multiplexer

```
//Define a 2-to-1 multiplexer using switches
module my_mux (out, s, i0, i1);
    output out;
    input s, i0, i1;
    //internal wire
    wire sbar; //complement of s
    //create the complement of s;
    //use my_nor defined   previously.
    my_nor nt (sbar, s, s); //equivalent to a not gate
    //instantiate cmos switches
    cmos (out, i0, sbar, s);
    cmos (out, i1, s, sbar);
endmodule
```

CMOS Inverter



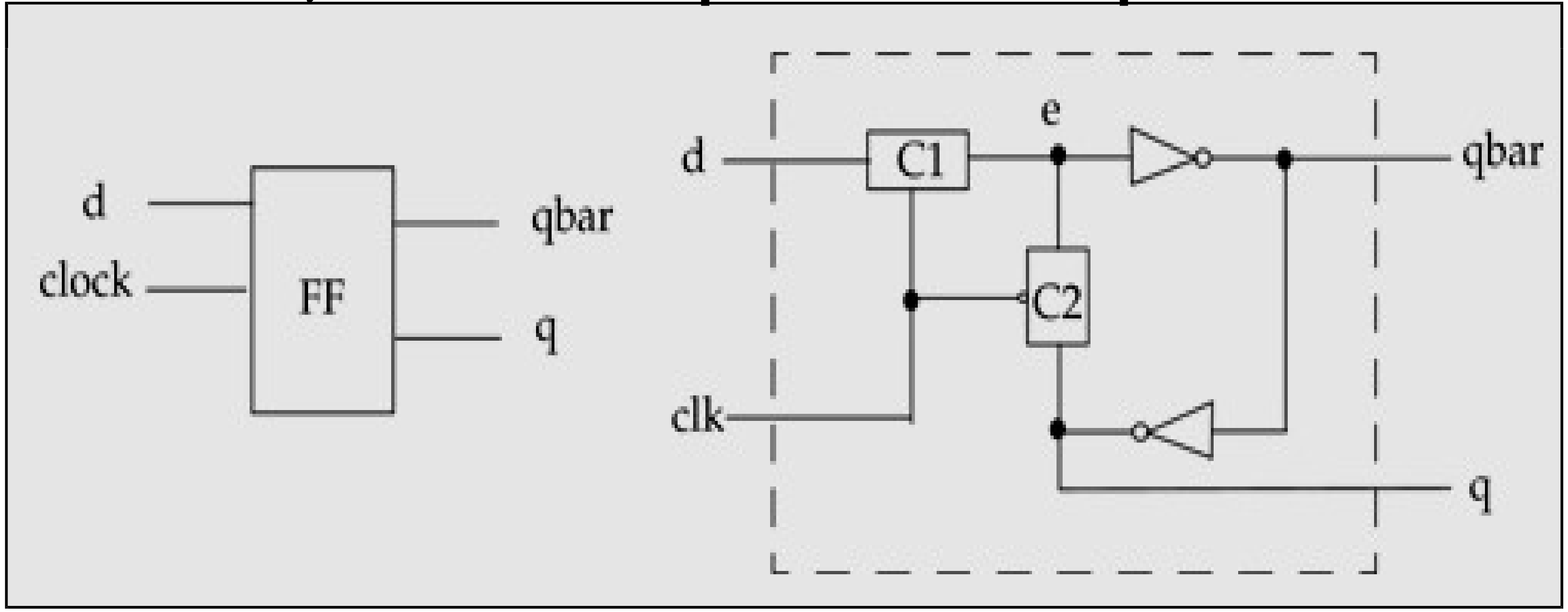
CMOS Inverter

- //Define an inverter using MOS switches

```
module my_not(out, in);
    output out;
    input in;
    //declare power and ground
    supply1 pwr;
    supply0 gnd;
    //instantiate nmos and pmos switches
    pmos (out, pwr, in);
    nmos (out, gnd, in);
endmodule
```

CMOS Flipflop

- The switches C1 and C2 are CMOS switches.
  - Switch C1 is closed if clk = 1, and switch C2 is closed if clk = 0.
- Complement of the clk is fed to the ncontrol input of C2.
- The CMOS inverters can be defined by using MOS switches, as shown in previous example.



CMOS Flipflop

```
//Define a CMOS flipflop
module cff ( q, qbar, d, clk);
    output q, qbar;
    input d, clk;
    //internal nets
    wire e;
    wire nclk; //complement of clock
    //instantiate the inverter
    my_not nt(nclk, clk);
    //instantiate CMOS switches
    cmos (e, d, clk, nclk); //switch C1 closed i.e. e = d, when clk = 1.
    cmos (e, q, nclk, clk); //switch C2 closed i.e. e = q, when clk = 0.
    //instantiate the inverters
    my_not nt1(qbar, e);
    my_not nt2(q, qbar);
endmodule
```