



SYSTEM SPECIFICATIONS USING VERILOG HDL

Dr. Mohammed M. Farag



**Faculty of Engineering
Alexandria University**



Outline

- Introduction
- Basic Concepts
- Modules and Ports
- Gate-Level Modeling
- Dataflow Modeling
- Behavioral Modeling
- Tasks and Functions

Textbook: Verilog HDL: A Guide to Digital Design and Synthesis, Second Edition By Samir Palnitkar



Introduction



Hardware Description Language

- Better be standard than be proprietary
- Can describe a design at some levels of abstraction
- Can be interpreted at many level of abstraction
 - Cross functional, statistical behavioral, multi-cycles behavioral, RTL
- Can be used to document the complete system design tasks
 - Testing, simulation, ..., related activities
- User define types, functions and packages
- Comprehensive and easy to learn



Design Methodologies

- Top-Down Design
 - Start with system specification
 - Decompose into subsystems, components, until indivisible
 - Realize the components
- Bottom-up Design
 - Start with available building blocks
 - Interconnect building blocks into subsystems, then system
 - Achieve a system with desired specification
- Meet in the middle
 - A combination of both



Importance of HDLs

- Designs can be described at very abstract levels without predefining the fabrication technology
- Functional verification of the design can be done early in the design cycle
- Designers can optimize and modify the RTL description until it meets the desired functionality
- Designing with HDLs is analogous to computer programming
- With rapidly increasing complexities of digital circuits and increasingly sophisticated EDA tools, HDLs are now the dominant method for large digital designs



Verilog History

- Gateway Design Automation
 - Phil Moorbr in 1984 and 1985
- Verilog-XL, “XL algorithm”, 1986
 - Gate-level simulation
- Verilog logic synthesizer, Synopsis, 1988
 - Top-down design methodology
- Cadence Design Systems acquired Gateway
 - December 1989
 - a proprietary HDL



Verilog History

- Open Verilog International (OVI), 1991
 - Language Reference Manual (LRM)
- The IEEE 1364 working group, 1994
- Verilog become an IEEE standard (1364-1995)
 - December, 1995
- 2001, IEEE standard 1364-2001



Popularity of Verilog HDL

- Verilog HDL 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
- Verilog HDL allows different levels of abstraction to be mixed in the same model
- Most popular logic synthesis tools support Verilog HDL
 - All fabrication vendors provide Verilog HDL 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 Verilog



Trends in HDLs

- The speed and complexity of digital circuits have increased rapidly
- Designers have responded by designing at higher levels of abstraction
- Designers have to think only in terms of functionality.
- EDA tools take care of the implementation details and optimization
- The most popular trend currently is to design in HDL at an RTL level, because logic synthesis tools can create gate-level netlists from RTL level design



Trends in HDLs (2)

- Behavioral synthesis allowed engineers to design directly in terms of algorithms and the behavior of the circuit
- However, behavioral synthesis did not gain widespread acceptance
- Today, RTL design continues to be very popular
- Designers often mix gate-level description directly into the RTL description to achieve optimum results
- Another technique that is used for system-level design is a mixed bottom-up methodology

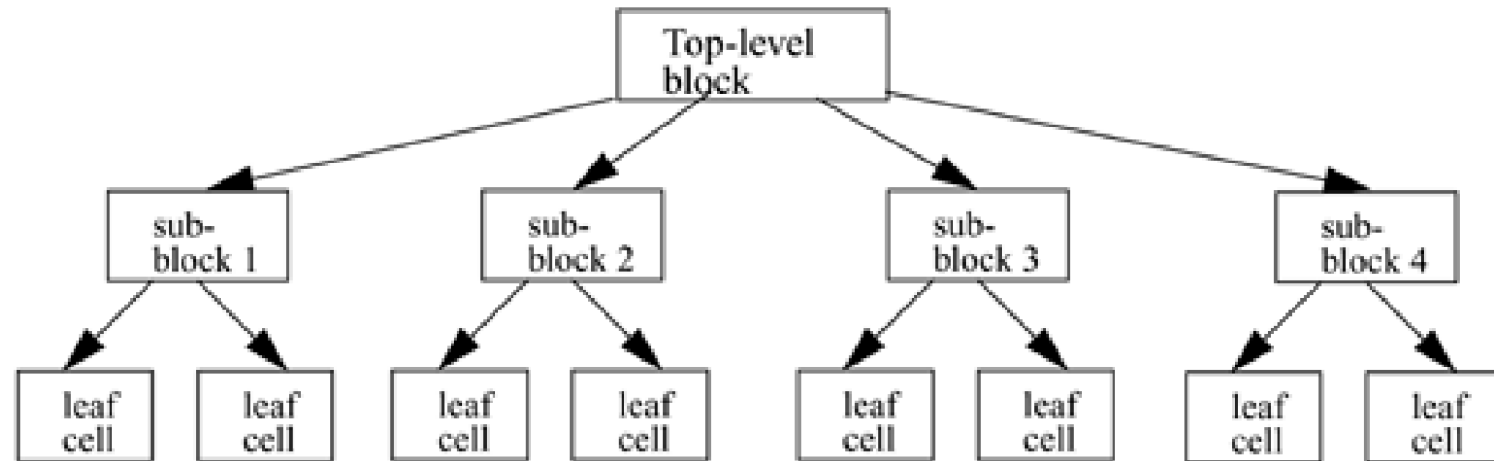


Hierarchical Modeling Concepts

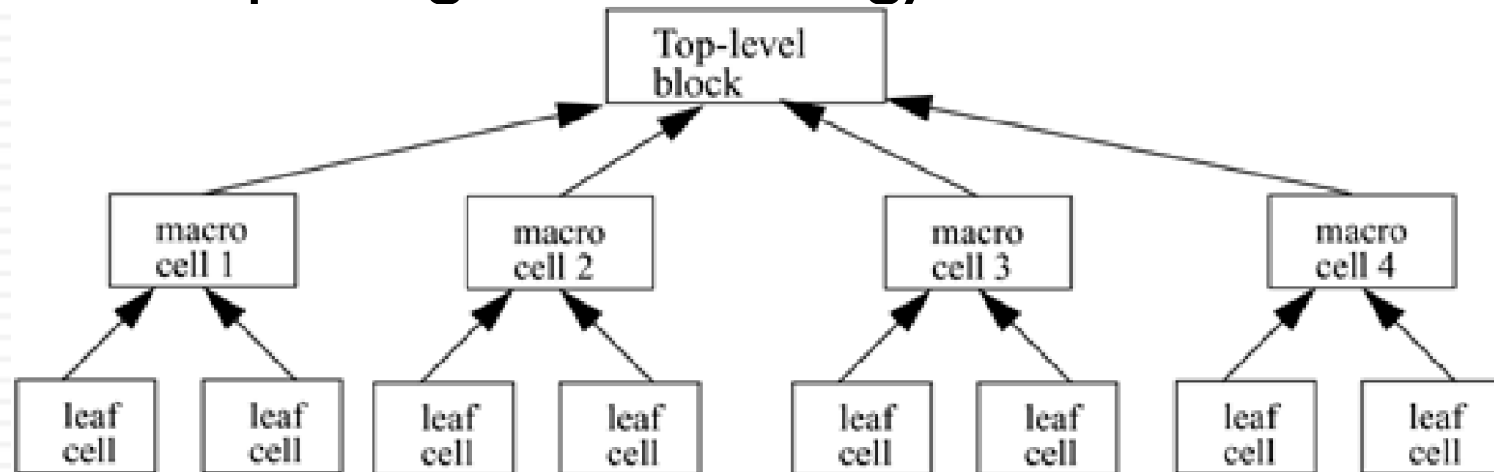


Design Methodologies

□ Top-down design methodology



□ Bottom-up design methodology



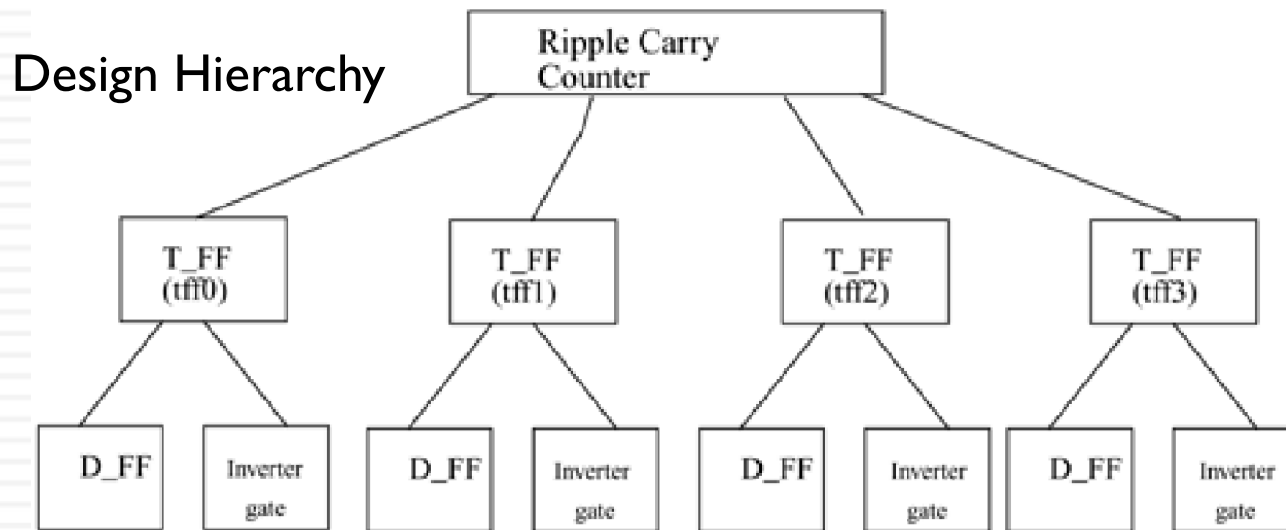
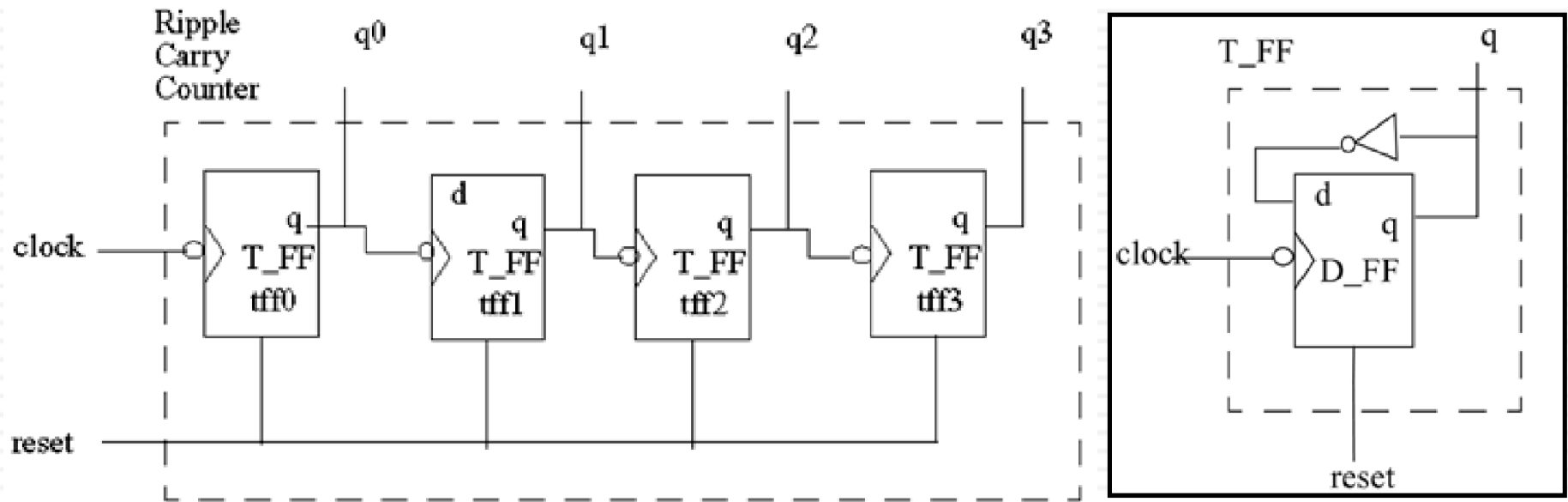


Design Methodologies (2)

- 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
- The flow meets at an intermediate point



Example: 4-bit Ripple Carry Counter





Modules

- A module is the basic building block in Verilog
- A module can be an element or a collection of lower-level design blocks
- Typically, elements are grouped into modules to provide common functionality that is used at many places in the design
- A module provides the necessary functionality to the higher-level block through its port interface, but hides the internal implementation
- This allows the designer to modify module internals without affecting the rest of the design



Module Declaration

- In Verilog, a module is declared by the keyword module
- In Verilog, a module is declared by the keyword module
- Each module must have a module_name, which is the identifier for the module, and a module_terminal_list, which describes the input and output terminals of the module

```
module <module_name> (<module_terminal_list>);
```

```
...
```

```
<module internals>
```

```
...
```

```
endmodule
```



Example

- The T-flip flop could be defined as a module as follows

```
module T_FF (q, clock, reset);
```

```
·
```

```
·
```

```
<functionality of T-flipflop>
```

```
·
```

```
·
```

```
endmodule
```



Verilog Abstraction Levels

- Verilog is both a behavioral and a structural language
- Internals of each module can be defined at four levels of abstraction, depending on the needs of the design include
 - Behavioral or algorithmic level:
 - This is the highest level of abstraction provided by Verilog HDL (similar to C programming)
 - A module can be implemented in terms of the desired design algorithm without concern for the hardware implementation details



Verilog Abstraction Levels (2)

- ❑ Dataflow level
 - The module is designed by specifying the data flow
 - The designer is aware of how data flows between hardware registers and how the data is processed in the design
- ❑ Gate level
 - The module is implemented in terms of logic gates and interconnections between these gates
 - Design at this level is similar to describing a logic design
- ❑ Switch level
 - This is the lowest level of abstraction provided by Verilog
 - A module can be implemented in terms of switches, storage nodes, and the interconnections between them



Verilog Abstraction Levels (3)

- Verilog allows the designer to mix and match all four levels of abstractions in a design
- The term register transfer level (RTL) is frequently used for a Verilog description that uses a combination of behavioral and dataflow constructs
- Normally, the higher the level of abstraction, the more flexible and technology-independent the design
- However, at this level, the designer does not have the control over low-level details which are automatically generated by the synthesis tool



Mixing Structure and Behavior

module *module_name* (*port_list*);

Declarations:

Net declarations.

Reg declarations.

Parameter declarations.

Initial statements.

Gate instantiation statements.

Module instantiation statements.

UDP instantiation statements.

Always statements.

Continuous assignment.

endmodule



Instances

- ❑ A module provides a template from which you can create actual objects
- ❑ Verilog creates a unique object from the template when a module is invoked
- ❑ The process of creating objects from a module template is called instantiation, and the objects are called instances
- ❑ In Verilog, it is illegal to nest modules
 - ❑ One module definition cannot contain another module definition within the module and endmodule statements
 - ❑ Instead, a module definition can incorporate copies of other modules by instantiating them



Example- I

```
// Define the top-level module called ripple carry  
// counter. It instantiates 4 T-flipflops. Interconnections are  
// shown in Section 2.2, 4-bit Ripple Carry Counter.
```

```
module ripple_carry_counter(q, clk, reset);  
    output [3:0] q; //I/O signals and vector declarations  
                    //will be explained later.  
    input clk, reset; //I/O signals will be explained later.  
    //Four instances of the module T_FF are created. Each has a unique  
    //name.Each instance is passed a set of signals. Notice, that  
    //each instance is a copy of the module T_FF.  
    T_FF tff0(q[0],clk, reset);  
    T_FF tff1(q[1],q[0], reset);  
    T_FF tff2(q[2],q[1], reset);  
    T_FF tff3(q[3],q[2], reset);  
endmodule
```




Example-2

```
// Define the module T_FF. It instantiates a D-flipflop. We assumed  
// that module D-flipflop is defined elsewhere in the design. Refer  
// to Figure 2-4 for interconnections.
```

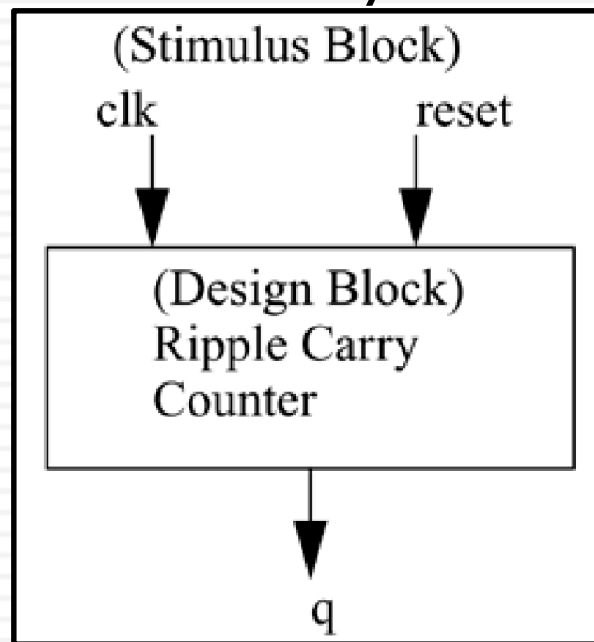
```
module T_FF(q, clk, reset);  
    //Declarations to be explained later  
    output q;  
    input clk, reset;  
    wire d;  
    D_FF dff0(q, d, clk, reset); // Instantiate D_FF. Call it dff0.  
    not n1(d, q); // not gate is a Verilog primitive. Explained  
                later.  
endmodule
```



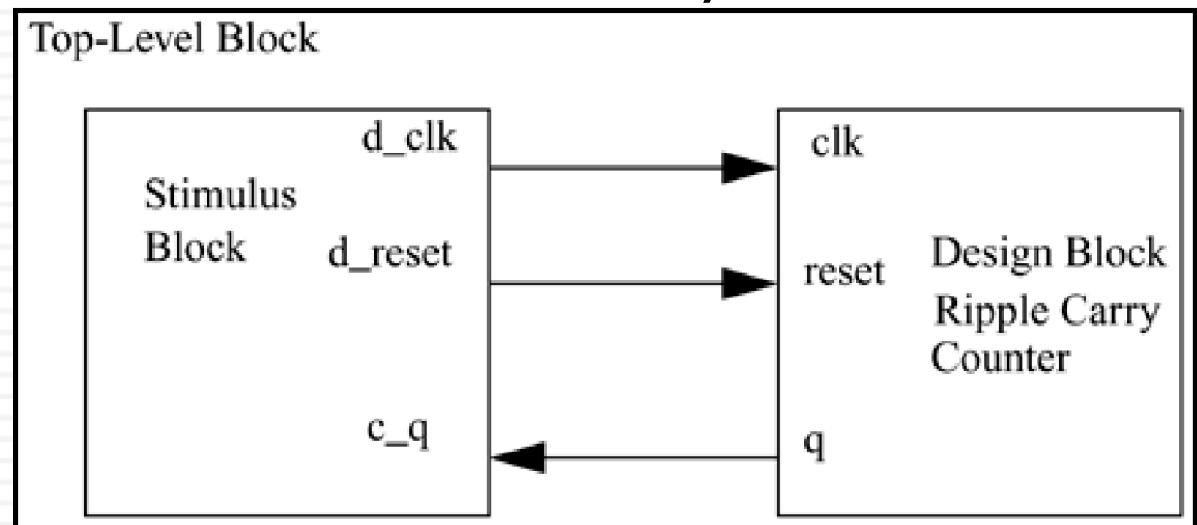
Components of Simulation

- The functionality of the design block can be tested by applying stimulus and checking results
 - We call such a block the stimulus block or a test bench
- Two styles of stimulus application are possible

First style



Second style





Example

□ Ripple Carry Counter Top Block

```
module ripple_carry_counter(q, clk,
reset);
    output [3:0] q;
    input clk, reset;
    //4 instances of the module
T_FF are created.
    T_FF tff0(q[0],clk, reset);
    T_FF tff1(q[1],q[0], reset);
    T_FF tff2(q[2],q[1], reset);
    T_FF tff3(q[3],q[2], reset);
endmodule
```

□ Flipflop T_FF

```
module T_FF(q, clk, reset);
    output q;
    input clk, reset;
    wire d;
    D_FF dff0(q, d, clk,
reset);
        not n1(d, q); /* not is a
Verilog-provided primitive.
case sensitive*/
endmodule
```



Example (2)

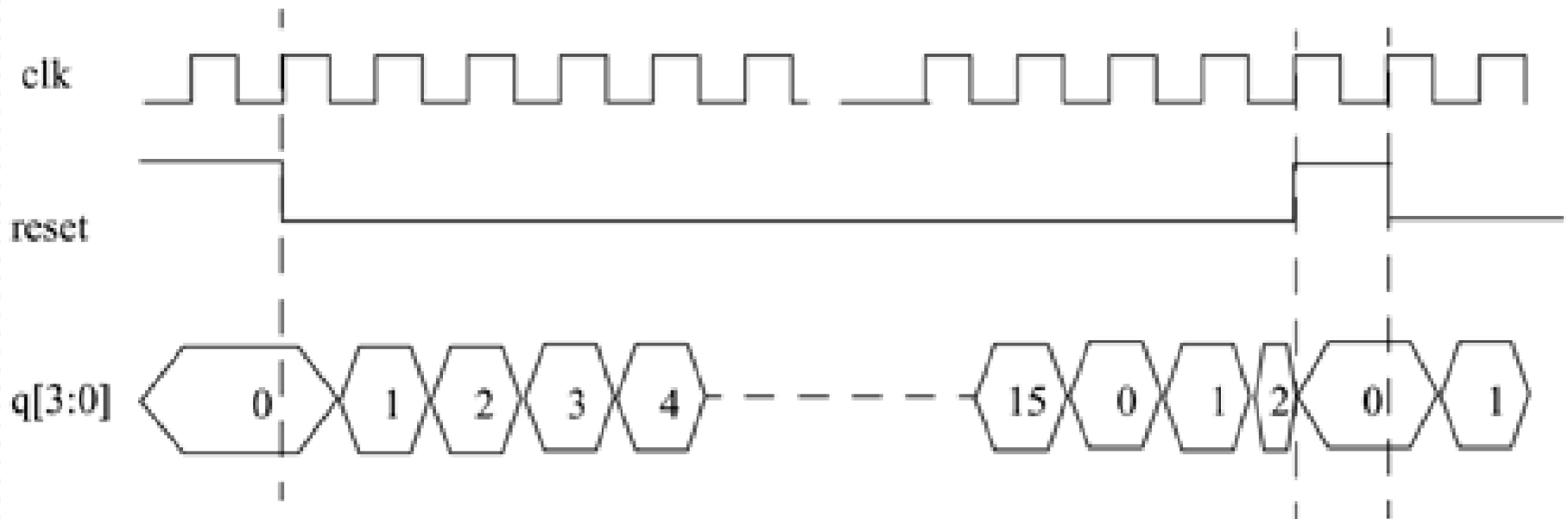
```
// module D_FF with synchronous reset
module D_FF(q, d, clk, reset);
    output q;
    input d, clk, reset;
    reg q;
// Lots of new constructs. Ignore the functionality of the
// constructs.
// Concentrate on how the design block is built in a top-down
// fashion.
    always @(posedge reset or negedge clk)
        if (reset)
            q <= 1'b0;
        else
            q <= d;
endmodule
```



Example (3)

□ Stimulus Block

- We control the signals clk and reset so that the regular function of the ripple carry counter
- Waveforms for clk, reset, and 4-bit output q are shown





Example: Stimulus Block

```
module stimulus;
reg clk;
reg reset;
wire[3:0] q;
// instantiate the design block
ripple_carry_counter r1(q, clk,
reset);
// Control the clk signal that drives
the //design block. Cycle time = 10
initial
clk = 1'b0; //set clk to 0
always
#5 clk = ~clk; //toggle clk every 5
time units
```

```
/* Control the reset signal that drives
the design block */
initial
begin
reset = 1'b1;
#15 reset = 1'b0;
#180 reset = 1'b1;
#10 reset = 1'b0;
#20 $finish; //terminate the
//simulation
end
// Monitor the outputs
initial
$monitor($time, " Output q = %d", q);
endmodule
```



Basic Concepts



Basics

- Free format
- Case sensitive
- white space (blank, tab, newline) can be used freely
- Identifiers: sequence of letters, \$ and _(underscore). First has to be a letter or an _

Symbol
symbol
R12_3\$
_R2

- Escaped identifiers: starts with a \ (backslash) and end with white space

\7400
\.*.\$
{*}
\~Q

- Keywords: Cannot be used as identifiers
E.g. **initial, assign, module**



Basics (Contd)

- Comments: Two forms
 - /* First form: can extend over many lines */
 - // Second form: ends at the end of this line
- **\$SystemTask / \$SystemFunction**
 - \$time**
 - \$monitor**
- Compiler-directive: directive remains in effect through the rest of compilation.
 - // Text substitution

 - 'define MAX_BUS_SIZE 32

 -
 - reg['MAX_BUS_SIZE-1:0] ADDRESS;



Lexical Conventions

- **Whitespace**
 - Blank spaces (`\b`) , tabs (`\t`) and newlines (`\n`) comprise the whitespace
 - Whitespace is ignored by Verilog except when it separates tokens
 - Whitespace is not ignored in strings
- **Comments**
 - Comments can be inserted in the code for readability and documentation
 - There are two ways to write comments
 - A one-line comment starts with `///
//`
 - A multiple-line comment starts with `/*` and ends with `*/`



Lexical Conventions (2)

□ Comment examples

```
a = b && c; // This is a one-line comment
```

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

```
/* This is /* an illegal */ comment */
```

```
/* This is //a legal comment */
```

□ Operators

```
a = ~ b; // ~ is a unary operator. b is the operand
```

```
a = b && c; // && is a binary operator. b and c are  
operands
```

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



Number Specification

- There are two types of number specification in Verilog: sized and unsized
 - Sized numbers `<size> '<base format> <number>`
 - `4'b1111` // This is a 4-bit binary number
 - `12'habc` // This is a 12-bit hexadecimal number
 - `16'd255` // This is a 16-bit decimal number
 - Unsized numbers: Decimal numbers with a default size
 - `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



Number Specification

□ Negative numbers

- Negative numbers can be specified by putting a minus sign before the size for a constant number

-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

□ X or Z values

- An unknown value is denoted by an x

- A high impedance value is denoted by z

12'h13x // This is a 12-bit 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



Strings

□ Strings

- A string is a sequence of characters that are enclosed by double quotes
- It cannot be on multiple lines
- Strings are treated as a sequence of one-byte ASCII values

"Hello Verilog World" // is a string

"a / b" // is a string

```
reg [8*18:1] string_value; // Declare a variable that is 18 bytes wide
```

Initial

```
string_value = "Hello Verilog World"; // String can be stored in variable
```

- An underscore character "_" is allowed anywhere in a number except the first character to improve readability



Identifiers and Keywords

- Keywords are special identifiers reserved to define the language constructs written in lowercase
- 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

`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
 - The four value levels are: 0, 1, x, z
 - strength levels are often used to resolve conflicts between drivers of different strengths in digital circuits

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

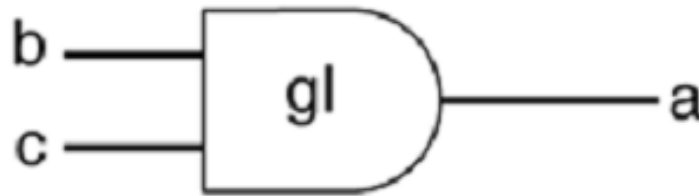
Strength Level	Type	Degree
supply	Driving	
strong	Driving	
pull	riving	
large	Storage	
weak	Driving	
medium	Storage	
small	Storage	
highz	High Impedance	



Nets

- ❑ Nets represent connections between hardware elements
- ❑ Nets are declared primarily with the keyword wire
- ❑ The terms wire and net are often used interchangeably
- ❑ Nets get the output value of their drivers
- ❑ The default value of a net is z

Example:



```
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.
```



Registers

- ❑ Registers represent data storage elements
- ❑ Registers retain value until another value is placed onto them
- ❑ Unlike a net, a register does not need a driver
- ❑ Register data types are commonly declared by the keyword `reg`
- ❑ Registers can also be declared as signed variables
- ❑ 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
```



Vectors

- Nets or reg data types can be declared as vectors (multiple bit widths)

```
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
```

- 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



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 virtual_addr
```

- Variable Vector Part Select
 - Another ability provided in Verilog HDL is to have variable part selects of a vector
 - Check the Palnitkar Verilog reference (page 48)



Integer Numbers

- An integer is a general purpose register data type
- Integers are declared by the keyword integer
- 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



Integer Numbers (2)

- Integers: Decimal, hexadecimal, octal, binary
- Simple decimal form:
 - 32 decimal 32
 - 15 decimal -15
- Signed integers
- Negative numbers are in two's complement form
- Base format form:
 - [<size>] '<base><value>
 - 'hAF (h,A, F are case insensitive) // 8-bit hex
 - 'o721 // 9-bit octal
 - 5'O37 // 5-bit octal
 - 4'D2 // 4-bit decimal
 - 4'B1x02 // 4-bit binary
 - 7'hx (x is case insensitive) // 7-bit x (x extended)
 - 4'hz // 4-bit z (z extended)



Integer Numbers (3)

- Unsigned integers
- Padding:
 - 10'b10 // padded with 0's
 - 10'bx10 // padded with x's
- ? can replace z in a number: used to enhance readability where z is a high impedance
- _(underscore) can be used anywhere to enhance readability, except as the first character
- Example:
 - 8'd-6 // illegal
 - 8'd6 // -6 held in 8 bits



Real Numbers

- Real number constants and real register data types are declared with the keyword `real`
- Real numbers cannot have a range declaration, and their default value is 0

```
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)
```




Real Numbers (2)

- Decimal notation

10.5

1.41421

0.01

- Scientific notation

235.1e2 (e is case insensitive) // 23510.0

3.6E2 // 360.0

5E-4 // 0.0005

- Must have at least one digit on either side of decimal
- Stored and manipulated in double precision (usually 64 bits)



Time Data Type

- Verilog simulation is done with respect to simulation time
- A special time register data type is used in Verilog to store simulation time
- A time variable is declared with the keyword time
- The system function \$time is invoked to get the current simulation time

```
time save_sim_time; // Define a time variable save_sim_time  
initial
```

```
    save_sim_time = $time; // Save the current simulation time
```



Strings

- “Sequence of characters”
- \n, \t, \\, \”, %%
 - \n = newline
 - \t = tab
 - \\ = backslash
 - \” = quote mark (“)
 - %% = % sign



Arrays

- Arrays are allowed in Verilog 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
- 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`

`integer matrix[4:0][0:255]; // Two dimensional array of integers`

`wire [7:0] w_array2 [5:0]; // Declare an array of 8 bit vector wire`

`wire w_array1[7:0][5:0]; // Declare an array of single bit wires`



Memories

- Memories are modeled in Verilog 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

```
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.
```



Parameters

- Verilog allows constants to be defined in a module by the keyword parameter
- Parameters cannot be used as variables
- Parameter values for each module instance can be overridden individually at compile time
- This allows the module instances to be customized
- Parameters values can be changed at module instantiation or by using the defparam statement

```
parameter port_id = 5;           // Defines a constant port_id  
parameter cache_line_width = 256; // Defines width of cache_line  
parameter signed [15:0] WIDTH; // Fixed sign and range for width
```



Module Parameter Values

- Two ways
- defparam statement:
 - Parameter value in any module instance can be changed by using hierarchical name.

```
defparam FA.n1.XOR_DELAY = 2,  
         FA.n2.AND_DELAY = 3;
```

- Module instance parameter value assignment:
 - Specify the parameter value in the module instantiation.
 - Order of assignment is the same as order of declarations within module

```
HA # (2, 3) h1 (.A(p), .B(Q), .S(S1), .C(X1));
```



System Tasks

- Verilog provides standard system tasks for certain routine operations
- All system tasks appear in the form **\$<keyword>**
- Displaying information: **\$display** is the main system task for displaying values of variables or strings or expressions

**\$display(p1, p2, p3,....., pn); // p1, p2, p3,...., pn can be quoted
//strings or variables or expressions**

/Display value of current simulation time 230

```
$display($time);
```

```
-- 230
```

//Display value of port_id 5 in binary

```
reg [4:0] port_id;
```

```
$display("ID of the port is %b", port_id);
```

```
-- ID of the port is 00101
```




System Tasks (2)

- Monitoring information: **\$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

\$monitor(p1,p2,p3,....,pn); //p1, p2, ... , pn can be variables,
//signal names, or quoted strings

//Monitor time and value of the signals clock and reset

//Clock toggles every 5 time units and reset goes down at 10 time units

initial

begin

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

end



System Tasks (3)

- Stopping and finishing in a simulation
 - The task **\$stop** is provided to stop during a simulation
 - The **\$stop** task puts the simulation in an interactive mode to enable debugging
 - The **\$finish** task terminates the simulation

// Stop at time 100 in the simulation and examine the results

// Finish the simulation at time 1000.

initial // to be explained later. time = 0

begin

clock = 0;

reset = 1;

#100 \$stop; // This will suspend the simulation at time = 100

#900 \$finish; // This will terminate the simulation at time = 1000

end



Compiler Directives

- Compiler directives are provided in Verilog
- All compiler directives are defined by using the `<keyword>` construct
- We deal with the two most useful compiler directives: `define` and `include`:
 - The `define` directive is used to define text macros in Verilog

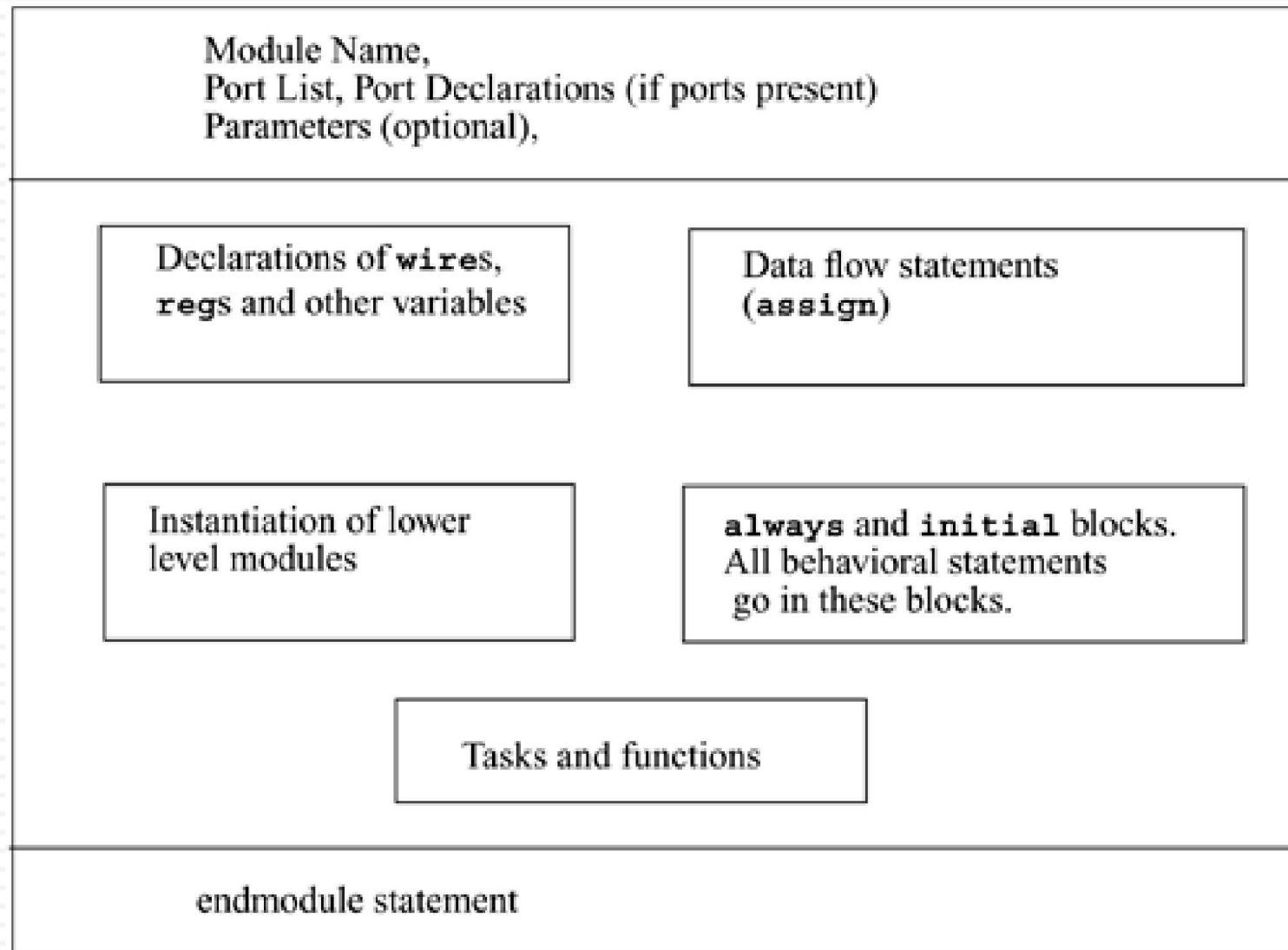
```
//define a text macro that defines default word size
//Used as 'WORD_SIZE in the code
'define WORD_SIZE 32
```
 - The `include` directive allows you to include entire contents of a Verilog source file in another Verilog file during compilation



Modules and Ports



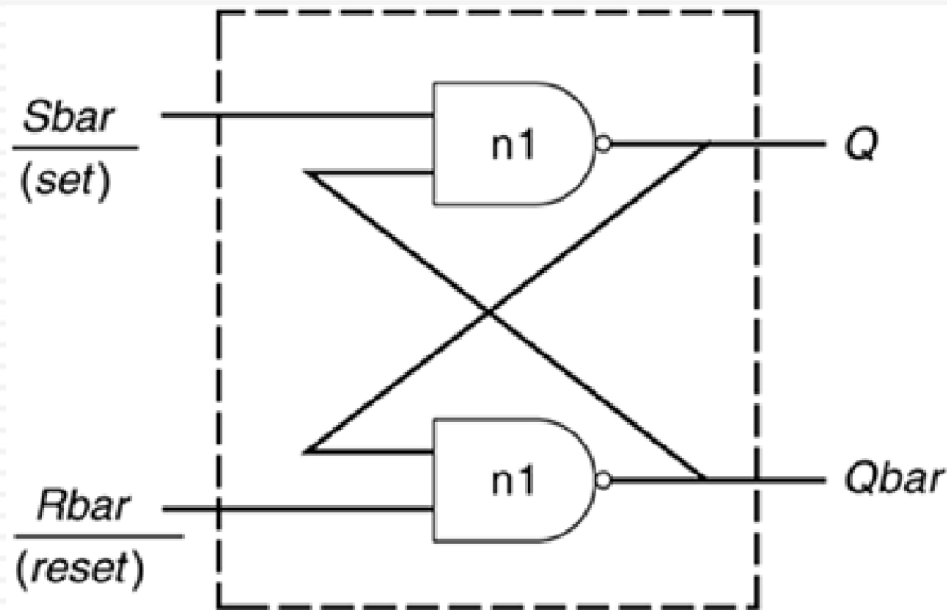
Components of a Verilog Module





Components of a Verilog Module

- To understand the components of a module shown above, check Example 4-1 (Palnitkar, P# 63)





Ports

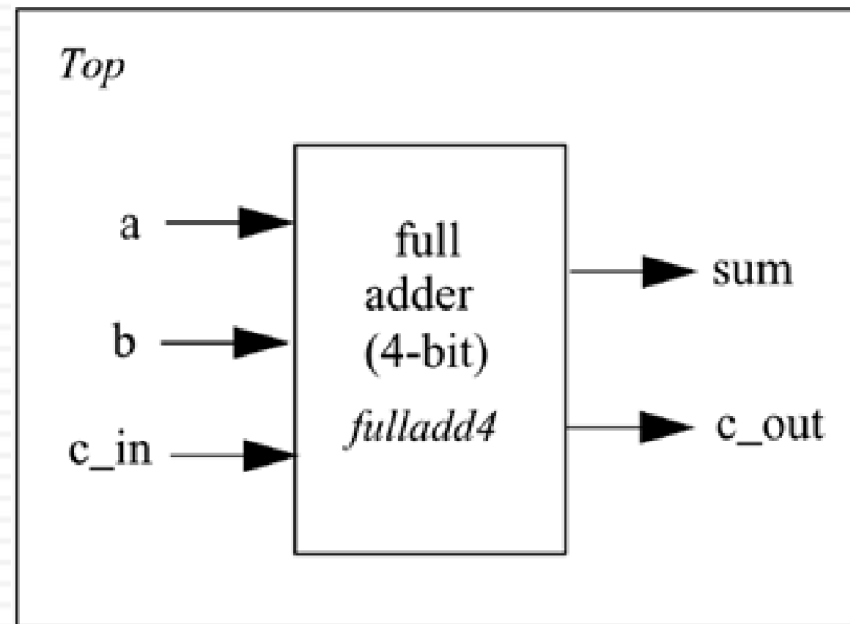
- Ports provide the interface by which a module can communicate with its environment
- The internals of the module are not visible to the environment
- This provides a very powerful flexibility to the designer
- The internals of the module can be changed without affecting the environment as long as the interface is not modified
- Ports are also referred to as terminals



List of Ports

- A module definition contains an optional list of ports
- Consider a 4-bit full adder that is instantiated inside a top-level module *Top*

Example



```
module fulladd4(sum, c_out, a, b, c_in); //Module with a list of ports
module Top; // No list of ports, top-level module in simulation
```




Port Declaration

- All ports in the list of ports must be declared in the module as follows

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

- Each port in the port list is defined as input, output, or inout, based on the direction of the port signal

```
module fulladd4(sum, c_out, a, b, c_in);
```

```
//Begin port declaration
```

```
output[3:0] sum;
```

```
output c_cout;
```

```
input [3:0] a, b;
```

```
input c_in;
```

```
//End port declaration
```

Alternative Declaration

```
module fulladd4(output reg [3:0] sum,  
output reg c_out, // output and c_out are  
//declared as reg  
input [3:0] a, b, //wire by default  
input c_in); //wire by default
```



Port Declaration (2)

- Note that all port declarations are implicitly declared as wire in Verilog
- 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 (inout cannot be declared as reg)

```
module DFF(q, d, clk, reset);
```

```
output q;
```

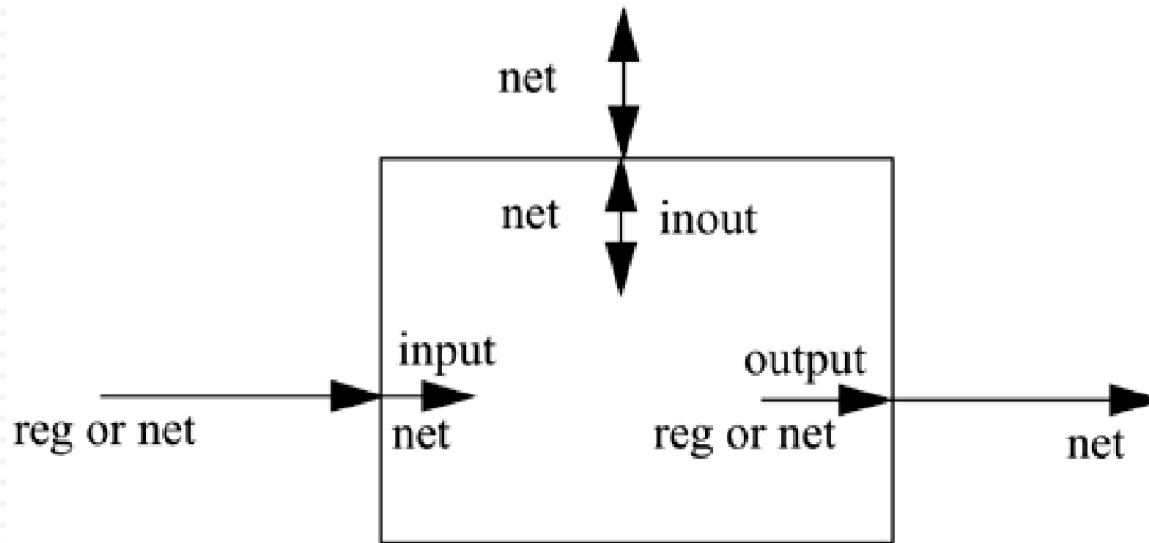
```
reg q; // Output port q holds value; therefore it is  
//declared as reg.
```

```
input d, clk, reset;
```



Port Connection Rules

- There are rules governing port connections when modules are instantiated within other modules



- Width matching
- Unconnected ports: Verilog allows ports to remain 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:** 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

```
module Top;  
//Declare connection variables  
reg [3:0]A,B;  
reg C_IN;  
wire [3:0] SUM;  
wire C_OUT;  
//Instantiate fulladd4, call it fa_ordered.  
//Signals are connected to ports in order (by position)  
fulladd4 fa_ordered(SUM, C_OUT, A, B, C_IN);
```



Connecting Ports to External Signals (2)

- ❑ **Connecting ports by name:** Verilog provides the capability to connect external signals to ports by the port names, rather than by position

```
// Instantiate module fa_byname and 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
- ❑ Unconnected ports can be dropped

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



Hierarchy

- Module definition:

```
module module_name (port_list);  
    declarations_and_statements  
endmodule
```

- Module instantiation statement:

```
module_name instance_name (port_associations);
```

- Port associations can be positional or named; cannot be mixed

```
local_net_name           // positional  
Port_Name(local_net_name) // Named
```

- Ports can be: input, output, inout
- Port can be declared as a net or a reg; must have same size as port



Hierarchy (2)

- Unconnected module inputs are driven to z state
- Unconnected module outputs are simply unused

```
DFF d1 (.Q(QS), .QBAR(), .DADA(D),  
        .PRESET(), .CLOCK(CK));      // Named
```

```
DFF d2 (QS, , D, , CK);              // Positional
```

```
// Output QBAR is not connected
```

```
// Input PRESET is open and hence set to calue z
```



Hierarchical Instantiation

```
module sub_block1 (a, z);  
input a;  
output z;  
wire a, z;  
IV U1 (.A(a), .Z(z));  
endmodule
```

```
module sub_block2 (a, z);  
input a;  
output z;  
wire a, z;  
IV U1 (.A(a), .Z(z));  
endmodule
```

```
module top (din1, din2, dout1,  
           dout2);  
input din1;  
input din2;  
output dout1;  
output dout2;
```

```
wire din1, din2, dout1, dout2;
```

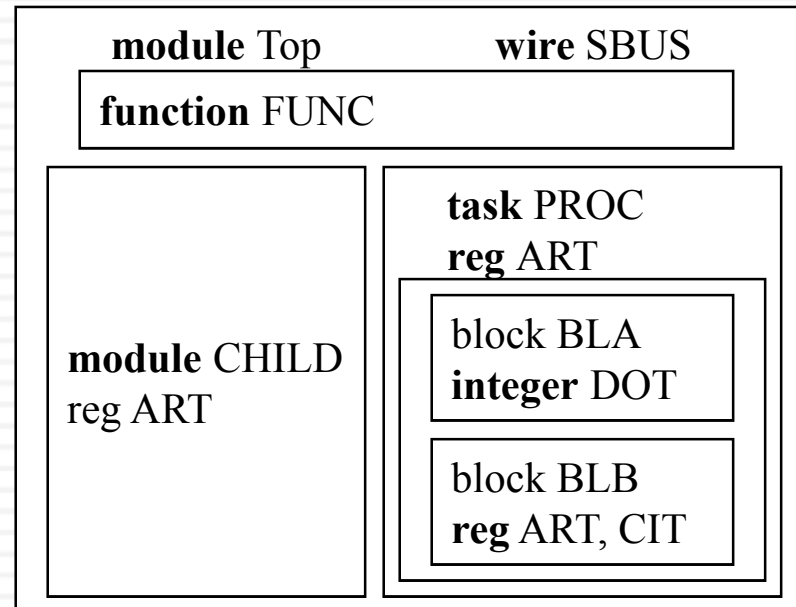
```
sub_block1 U1(.a(din1), .z(dout1));  
sub_block2 U2(.a(din2), .z(dout2));  
endmodule
```




Hierarchical Path Name

- Every identifier has a unique hierarchical path name
- Period character is the separator
- New hierarchy is defined by: module instantiation, task definition, function definition, named block

TOP.CHILD.ART
TOP.PROC.ART
TOP.PROC.BLB.CIT
TOP.PROC.BLA.DOT
TOP.SBUS





Gate-Level Modeling



Introduction

- Four levels of abstraction used to describe hardware
- At gate level, the circuit is described in terms of gates
- Actually, the lowest level of abstraction is switch-(transistor-) level modeling
- Most digital design is now done at gate level or higher levels of abstraction



Gate Types

- Verilog supports basic logic gates as predefined primitives
- These primitives are instantiated like modules except that they are predefined in Verilog and do not need a module definition
- There are two classes of basic gates: and/or gates and buf/not gates



And/Or Gates

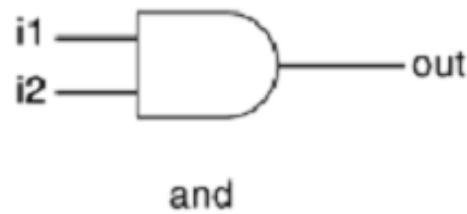
- The and/or gates available in Verilog are

and or xor

nand nor xnor

- The output terminal is denoted by out
- Input terminals are denoted by i1 and i2
- More than two inputs can be specified in a gate

instantiation



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

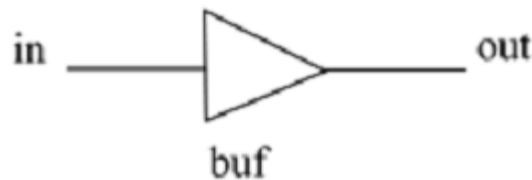
	i1			
nand	0	1	x	z
0	1	1	1	1
1	1	0	x	x
x	1	x	x	x
z	1	x	x	x



Buf/Not Gates

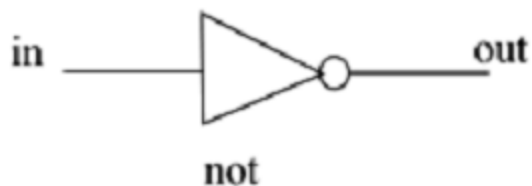
- Buf/not gates have one scalar input and one or more scalar outputs
- Two basic buf/not gate primitives are provided in Verilog

buf **not**



buf	in	out
	0	0
	1	1
	x	x
	z	x

not	in	out
	0	1
	1	0
	x	x
	z	x





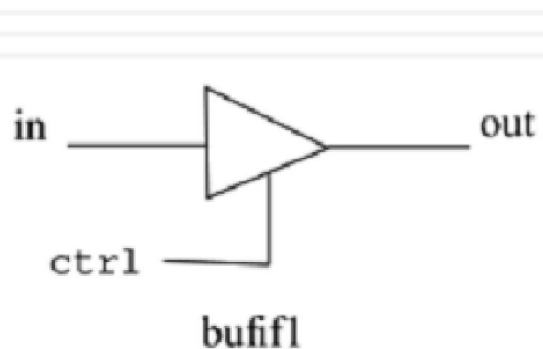
Bufif/notif

- 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



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

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



Array of Instances

- There are many situations when repetitive instances are required
- Verilog HDL allows an array of primitive instances to be defined

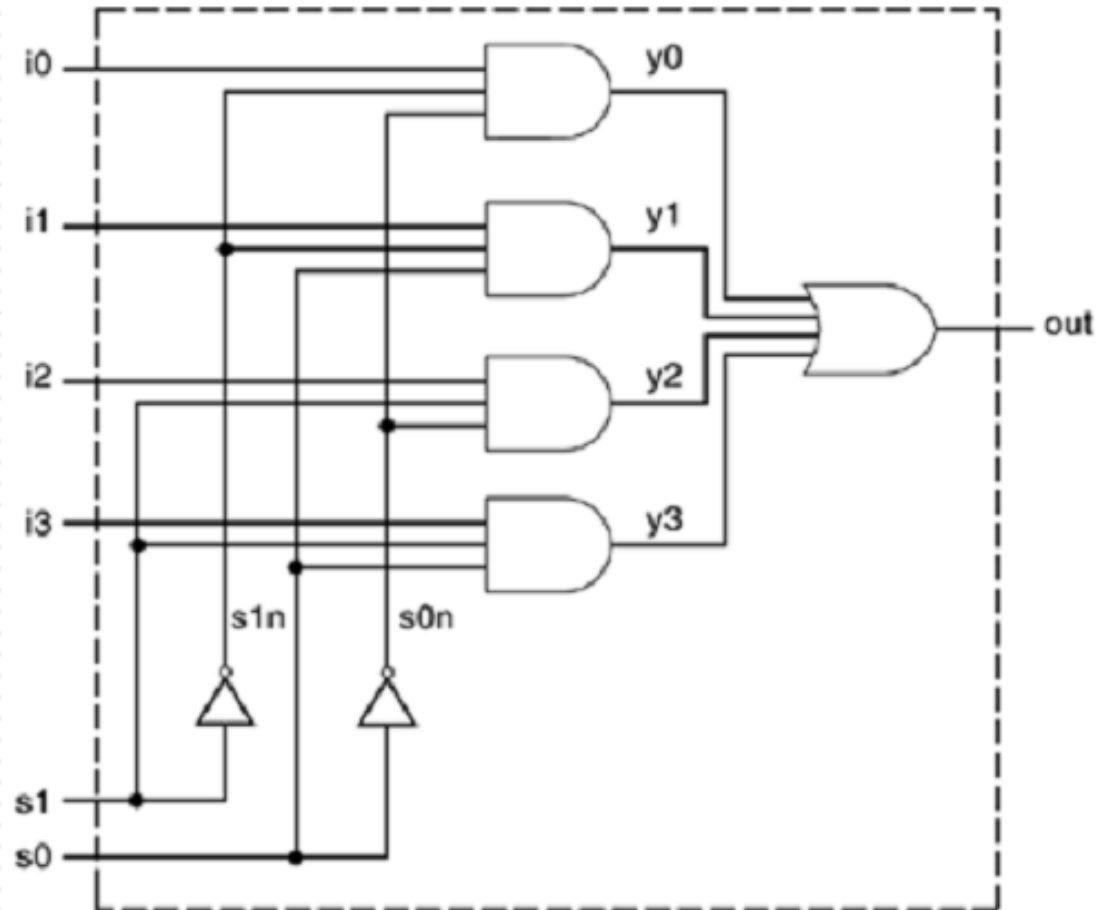
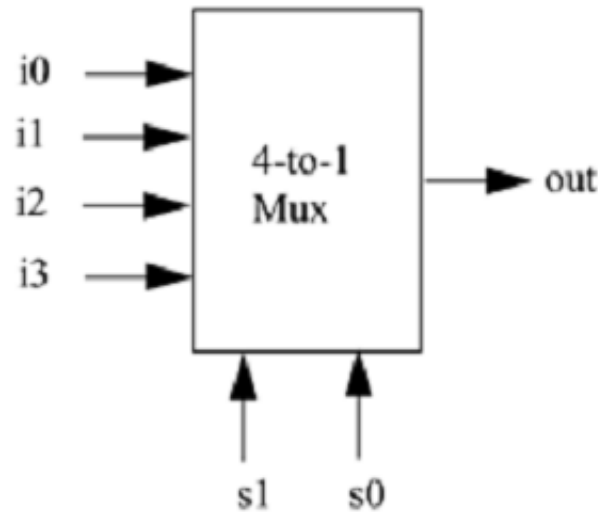
Example:

```
wire [3:0] OUT, IN1, IN2;  
// basic gate instantiations.  
nand n_gate[3:0](OUT, IN1, IN2);  
// This is equivalent to the following 4 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]);
```




Examples

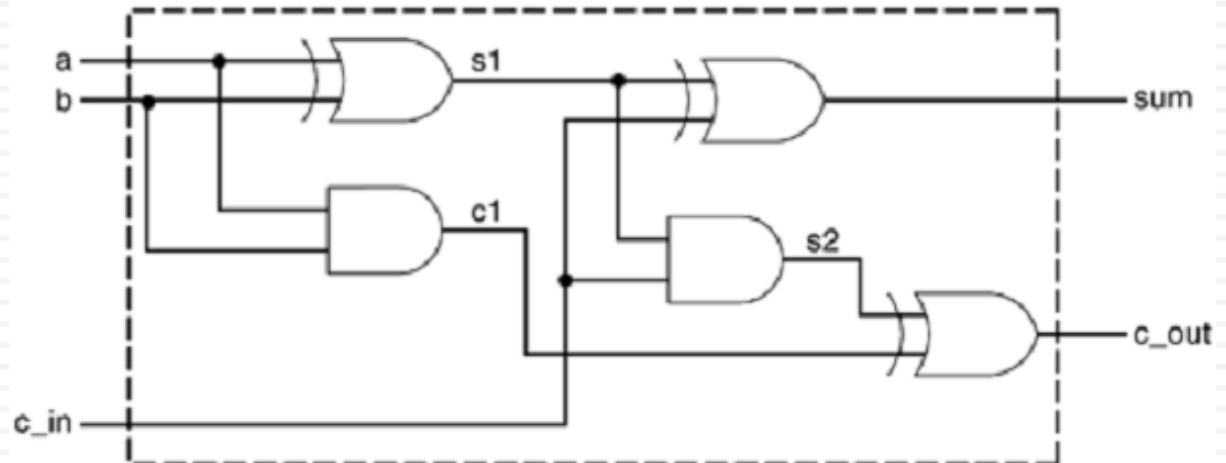
□ Gate-level multiplexer



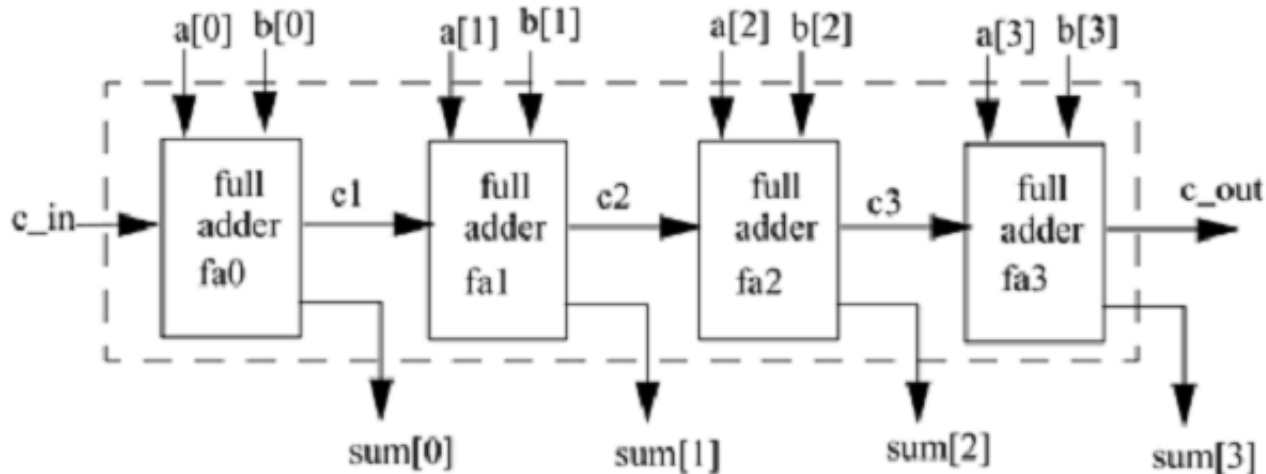


4-bit Ripple Carry Full Adder

1-bit full adder



4-bit Ripple Carry Adder





Gate Delays

- Gate delays allow the Verilog user to specify delays through the logic circuits
- Three types of delay specifications are allowed
 - Rise (0, x, z → 1)
 - Fall (1, x, z → 0)
 - Turn-off (0, 1, x → z)
- If no delays are specified, the default value is zero

// Delay of delay_time for all transitions

and #(delay_time) a1(out, i1, i2);

// Rise and Fall Delay Specification.

and #(rise_val, fall_val) a2(out, i1, i2);

// Rise, Fall, and Turn-off Delay Specification

bufif0 #(rise_val, fall_val, turnoff_val) b1 (out, in, control);



Gate Delays (2)

- Signal propagation delay from any gate input to the gate output
- Up to three values per output: rise, fall, turn-off delay

```
not N1 (XBAR, X);           // Zero delay
nand #(6) (out, in1, in2);  // All delays = 6
and #(3,5) (out, in1, in2, in3); /* rise delay = 3, fall delay = 5,
                                to_x_or_z = min(3,5) */
```
- Each delay can be written in *min:typ:max* form as well

```
nand #(2:3:4, 4:3:4) (out, in1, in2);
```
- Can also use a specify block to specify delays



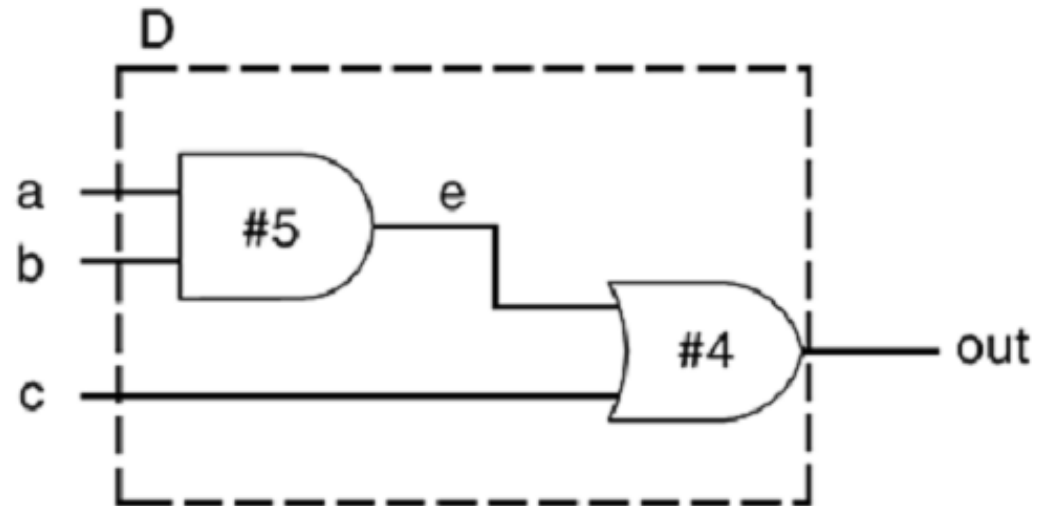
Example

```
// 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  
and #4 o1(out, e,c); //Delay of 4 on gate o1  
endmodule
```





A Clock Generator

```
module CLOCK (CLK,START);  
    output CLK;  
    Input START;  
  
    initial begin  
        START = 1;  
        #3 START = 0;  
    end  
  
        nor #5 (CLK, START, CLK);  
endmodule  
  
// Generate a clock with on-off width of 5  
// Not synthesizable  
// For waveform only
```



Time Unit and Precision

□ Compiler directive: **'timescale**

'timescale *time_unit / time_precision*;

□ 1, 10, 100 /s, ms, us, ns, ps, fs

'timescale 1ns / 100ps

module AND_FUNC (Z,A, B);

output Z;

input A, B;

and # (5.22, 6.17) A1 (Z,A, B);

endmodule

/* Delays are in ns. Delays are rounded to one-tenth of a ns (100ps).
Therefore,

5.22 becomes 5.2ns, 6.17 becomes 6.2ns and 8.59 becomes 8.6ns */

// If the following timescale directive is used:

'timescale 10ns / 1ns

// Then 5.22 becomes 52ns, 6.17 becomes 62ns, 8.59 becomes 86ns



Getting Simulation Time

- System function, **\$time**: returns the simulation time as an integer value scaled time unit specified.

```
'timescale 10ns / 1ns
```

```
module TB;
```

```
.....
```

```
initial
```

```
    $monitor ("PUT_A=%d PUT_B=%d", PUT_A, PUT_B,  
             "GET_O=%d", GET_O, "at time %t", $time);
```

```
endmodule
```

```
PUT_A=0 PUT_B=0 GET_O=0 at time 0
```

```
PUT_A=0 PUT_B=1 GET_O=0 at time 5
```

```
PUT_A=0 PUT_B=0 GET_O=0 at time 11
```

```
.....
```

```
/* $time value is scaled to the time unit and then rounded */
```




Switch-Level Modeling



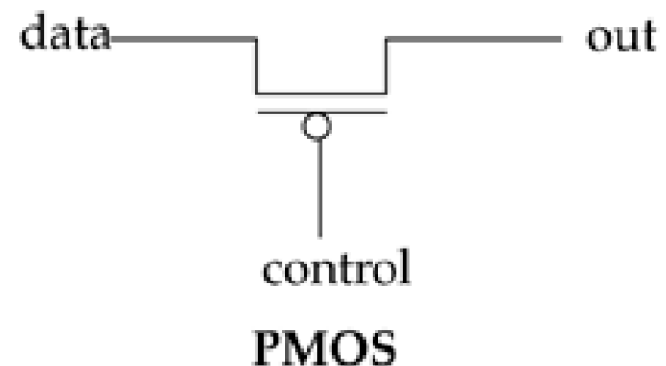
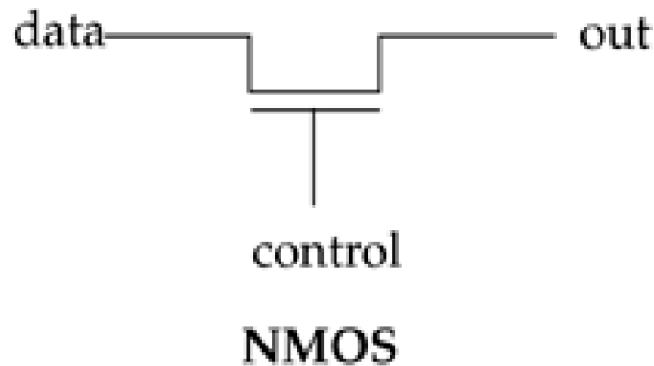
Introduction

- Verilog provides the ability to design circuits at a MOS-transistor level
- Verilog HDL currently provides only digital design capability with logic values 0, 1, x, z
- There is no analog capability
- Thus, in Verilog HDL, transistors are also known switches that either conduct or are open
- Design at this level is becoming rare with the increasing complexity of circuits



MOS Switches

- Two types of MOS switches can be defined with the keywords `nmos` and `pmos`



Example:

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



MOS Switches (2)

- The value of the out signal is determined from the values of data and control signals.
- Logic tables for out are shown in the following Table
- Some combinations of data and control cause the output to be 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

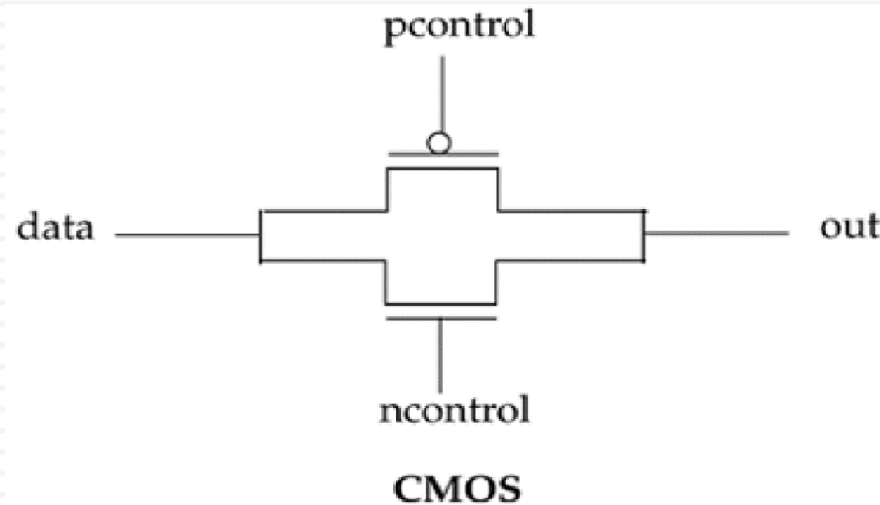
		control			
		0	1	x	z
data	0	z	0	L	L
	1	z	1	H	H
	x	z	x	x	x
	z	z	z	z	z

		control			
		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



CMOS Switches

- CMOS switches are declared with the keyword `cmos`



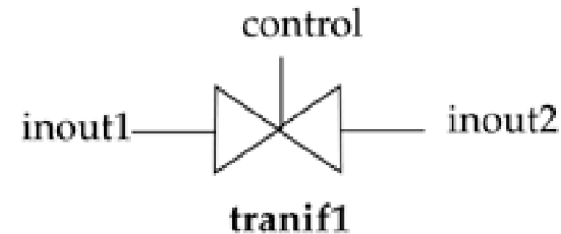
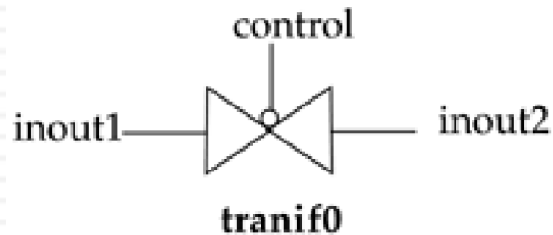
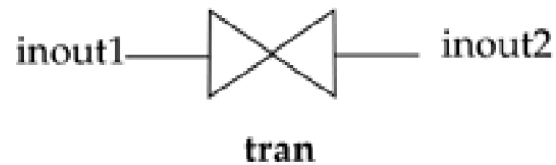
Example:

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



Bidirectional Switches

- Three keywords are used to define bidirectional switches: `tran`, `tranif0`, and `tranif1`



Example:

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

```
tranif0 (inout1, inout2, control); //instance name is not specified
```



Power and Ground

- The power (V_{dd} , logic 1) and Ground (V_{ss} , logic 0) sources are needed when transistor-level circuits are designed
- Power and ground sources are defined with keywords `supply1` and `supply0`

Example:

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



Resistive Switches

- 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

`rnmos rpmos //resistive nmos and pmos switches`

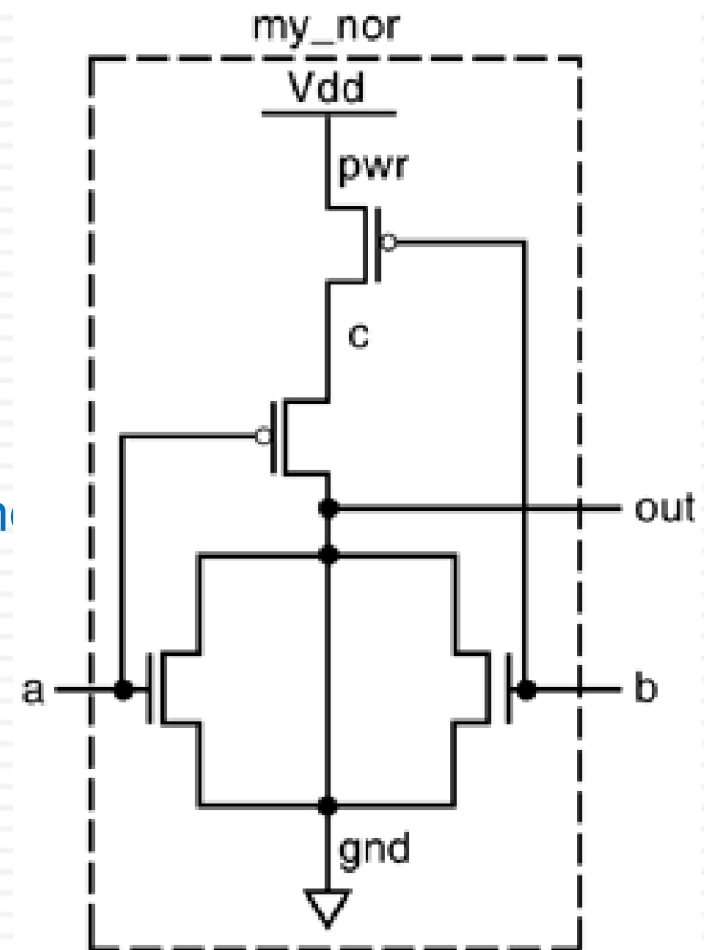
`rcmos //resistive cmos switch`

`rtran rtranif0 rtranif1 //resistive bidirectional switches`



Example: CMOS NOR Gate

```
//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
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
```





Example: 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.
```

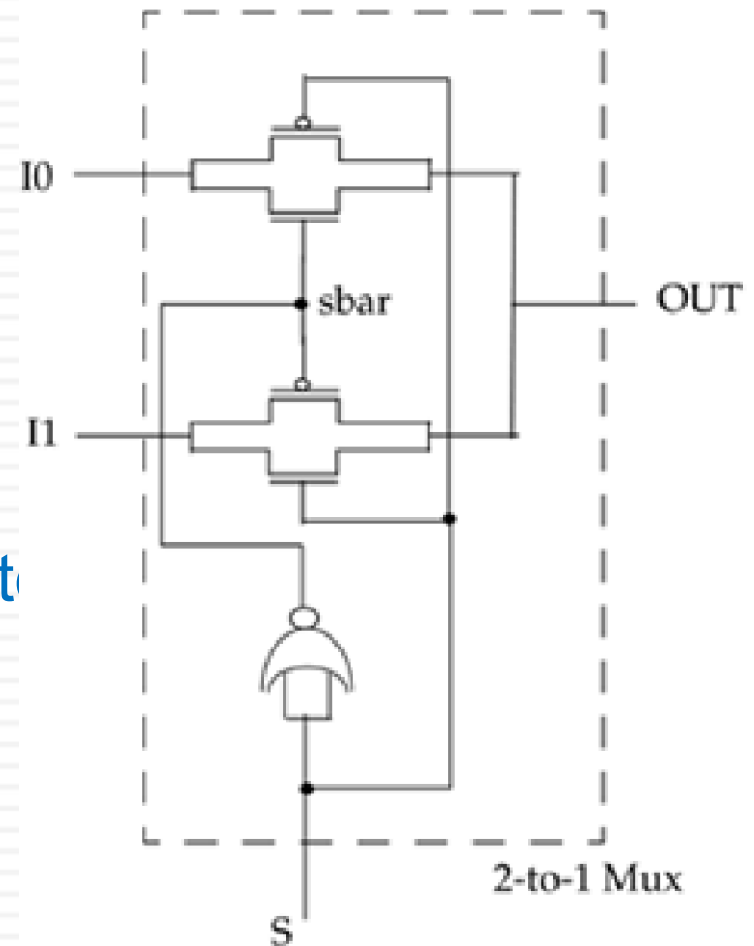
```
my_nor nt(sbar, s, s); //equivalent to
```

```
//instantiate cmos switches
```

```
cmos (out, i0, sbar, s);
```

```
cmos (out, i1, s, sbar);
```

```
endmodule
```





User-Defined Primitives



Introduction

- Verilog provides the ability to define User-Defined Primitives (UDP)
- There are two types of UDPs: combinational and sequential
 - Combinational UDPs are defined where the output is solely determined by a logical combination of the inputs
 - Sequential UDPs take the value of the current inputs and the current output to determine the value of the next output.



UDP Syntax

- UDP definition in pseudo syntax form is as follows:

```
//UDP name and terminal list  
primitive <udp_name> (  
<output_terminal_name>(only  
one allowed)  
<input_terminal_names> );  
  
//Terminal declarations  
output  
<output_terminal_name>;  
input <input_terminal_names>;  
reg  
<output_terminal_name>;(optio  
nal; only for sequential UDP)
```

```
// UDP initialization (optional;  
only for sequential UDP  
initial <output_terminal_name>  
= <value>;  
  
//UDP state table  
table  
    <table entries>  
endtable  
  
//End of UDP definition  
endprimitive
```



UDP Rules

- ❑ UDPs can take only scalar input terminals (1 bit)
- ❑ UDPs can have only one scalar output terminal (1 bit) always appearing first in the terminal list
- ❑ In the declarations section, the output terminal is declared with the keyword output
- ❑ The output terminal of sequential UDP is declared as a reg
- ❑ The inputs are declared with the keyword input
- ❑ The state in a sequential UDP can be initialized with an initial statement
- ❑ The state table entries can contain values 0, 1, or x
- ❑ UDPs are defined at the same level as modules
- ❑ UDPs do not support inout ports.



State Table Entries

- Each entry in the state table in a combinational UDP has the following pseudosyntax

`<input1> <input2> <inputN> : <output>;`

- The `<input#>` values in a state table entry must appear in the same order as they appear in the input terminal list
- Inputs and output are separated by a ":"
- A state table entry ends with a ";"
- All possible combinations of inputs, where the output produces a known value, must be explicitly specified. Otherwise, if a certain combination occurs and the corresponding entry is not in the table, the output is x.




Combinational UDP

```
primitive MUX1BIT (Z, A, B, SEL);  
  output Z;  
  input A, B, SEL;
```

```
  table
```

```
    // A  B  SEL  :  Z  
    0  ?  1  :  0 ;  
    1  ?  1  :  1 ;  
    ?  0  0  :  0 ;  
    ?  1  0  :  1 ;  
    0  0  x  :  0 ;  
    1  1  x  :  1 ;
```

Don't care



```
  endtable  
endprimitive
```

- Any combination that is not specified is an x.
- Output port must be the first port.
- “?” represents iteration over “0”, “1”, or “x” logic values



Sequential UDPs

- The output of a sequential UDP is always declared as a reg
- An initial statement can be used to initialize output of sequential UDPs
- The format of a state table entry is slightly different
$$\langle \text{input1} \rangle \langle \text{input2} \rangle \dots \langle \text{inputN} \rangle : \langle \text{current_state} \rangle : \langle \text{next_state} \rangle ;$$
- There are three sections in a state table entry: inputs, current state, and next state separated by a colon (:) symbol
- The input specification of state table entries can be in terms of input levels or edge transitions
- The current state is the current value of the output register
- The next state is computed based on inputs and the current state
- All possible combinations of inputs must be specified to avoid unknown output values



Sequential UDP (Level)

- Level-sensitive sequential UDP example:

```
primitive LATCH (Q, CLK, D);  
  output Q;  
  reg Q;  
  input CLK, D;
```

```
table
```

```
  //CLK Data State Output(next state)  
  1  1  :  ?  :  1  ;  
  1  0  :  ?  :  0  ;  
  0  ?  :  ?  :  -  ;
```

```
endtable
```

```
Endprimitive
```

- “?” means don’t-care
- “-” means no change in output



Sequential UDP (Edge)

- Edge-sensitive sequential UDP example:

```
primitive D_EDGE_FF (Q, CLK, D);
  output Q;
  reg Q;
  input CLK, D;

  table
    // CLK Data State Output(next state)
    (01) 0 : ? : 0 ;
    (01) 1 : ? : 1 ;
    (0x) 1 : 1 : 1 ;
    (0x) 0 : 0 : 0 ;
    // Ignore negative edge of clock;
    (?0) ? : ? : - ;
    // Ignore data change on steady clock;
    ? (??) : ? : - ;
  endtable
endprimitive
```



Dataflow Modeling



Introduction

- For small circuits, the gate-level modeling approach works very well
- 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
- In logic synthesis, automated tools create a gate-level circuit from a dataflow description



Basics

- Models behavior of combinational logic
- Dataflow style using continuous assignment
- Assign a value to a net
- Examples:

```
wire [3:0] Z, PRESET, CLEAR;  
assign Z = PRESET & CLEAR;
```

```
wire COUT, CIN;
```

```
wire [3:0] SUM, A, B;
```

```
assign {COUT, SUM} = A + B + CIN;
```

```
assign MUX = (S == 0) ? A: 'bz,
```

```
        MUX = (S == 1) ? B: 'bz,
```

```
        MUX = (S == 2) ? C: 'bz,
```

```
        MUX = (S == 3) ? D: 'bz;
```

```
assign Z = ~(A | B) & (C | D);
```

- Expression on right-hand side is evaluated whenever any operand changes



Net Declaration

- Can have an assignment in the net declaration

```
wire [3:0] A = 4'b0;  
assign PRESET = 'b1;
```

```
wire #10 A_GT_B = A > B;
```

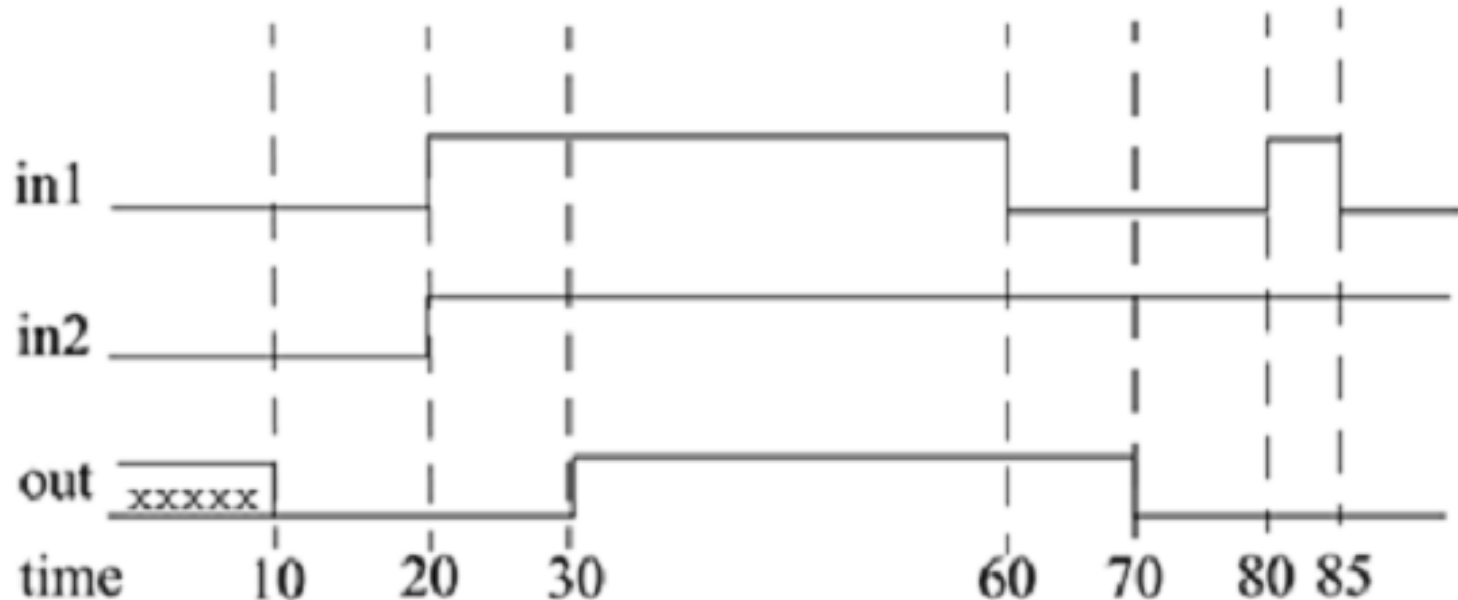
- Only one assignment to a net using net declaration
- Multiple assignments to a net is done using continuous assignments
- Continuous assignments have the following characteristics:
 - The left hand side of an assignment must be a net (cannot be a reg)
 - Continuous assignments are always active
 - The operands on the right-hand side can be registers or nets or function calls
 - Delay values can be specified for assignments in terms of time units



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
- Regular Assignment Delay

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





Delays (2)

- Delay between assignment of right-hand side to left-hand side

```
wire #10 A = B && C;    // Continuous delay
```

- Net delay

```
wire #10 A;
```

```
// Any change to A is delayed 10 time units before it takes effect
```

- If delay is in a net declaration assignment, then delay is not net delay

```
wire #10 A = B + C;
```

```
// 10 time units id it part of the continuous assignment and not net delay
```

- If value changes before it has a chance to propagate, latest value change will be applied

- Inertial delay



Expressions, Operators, and Operands

- Dataflow modeling describes the design in terms of expressions instead of primitive gates
- Expressions are constructs that combine operators and operands to produce a result

// Examples of expressions. Combines operands and operators

$a \wedge b$

`addr1[20:17] + addr2[20:17]`

`in1 | in2`

- Operands can be any one of the data types defined previously

`integer count, final_count;`

`final_count = count + 1; //count is an integer operand`



Operands

1. Numbered operands

2. Functional call operands

- A functional call can be used as an operand within an expression

```
wire [7:0] A;
```

```
// PARITY is a function described elsewhere
```

```
assign PAR_OUT = PARITY(A);
```

3. Bit selects

```
input [3:0] A, B, C;
```

```
output [3:0] SUM;
```

```
assign SUM[0] = (A[0] ^ B[0] ^ C[0]);
```

4. Part selects

5. Memory addressing

```
reg [7:0] RegFile [0:10];           // 11 b-bit registers
```

```
reg [7:0] A;
```

```
RegFile[3] = A;                     // A assigned to 3rd register in RegFile
```



Operators

- Operators act on the operands to produce desired results

$d1 \ \&\& \ d2 \ // \ \&\&$ is an operator on operands $d1$ and $d2$

Arithmetic	$*$ $/$ $+$ $-$ $\%$ $**$
Logical	$!$ $\&\&$ $\ \ \$
Relational	$>$ $<$ $>=$ $<=$
Equality	$==$ $!=$ $===$ $!==$
Bitwise	\sim $\&$ $ $ \wedge $\wedge\sim$ or $\sim\wedge$
Reduction	$\&$ $\sim\&$ $ $ $\sim $ \wedge $\wedge\sim$ or $\sim\wedge$
Shift	$>>$ $<<$ $>>>$ $<<<$
Concatenation	$\{\}$
Replication	$\{\{\}\}$
Conditional	$?:$



Arithmetic Operators

+	(plus)
-	(minus)
*	(multiply)
/	(divide)
%	(modulus)

- Integer division will truncate
- % gives the remainder with the sign of the first operand
- If any bit of operand is **x** or **z**, the result is **x**
- *reg* data type holds an unsigned value, while *integer* data type holds a signed value

```
reg [0:7] A;  
integer B;  
A = -4'd6;           // reg A has value unsigned 10  
B = -4'd6;          // integer B has value signed -6  
A-2                 // result is 8  
B-2                 // result is -8
```



Relational Operators

>	(greater than)
<	(less than)
>=	(greater than or equal to)
<=	(less than or equal to)

- If there are x or z in operand, the result is x
- If unequal bit lengths, smaller operand is zero-filled on most significant side (i.e., on left)



Equality Operators

<code>==</code>	(logical equality, result may be unknown)
<code>!=</code>	(logical inequality, result may be unknown)
<code>===</code>	(case equality, x and z also compared)
<code>!==</code>	(case inequality, x and z also compared)

```
A = 'b11x0;
```

```
B = 'b11x0;
```

```
A == B is known.
```

```
A === B is true
```

- Unknown is same as false in synthesis
- Compare bit by bit, zero-filling on most significant side



Logical Operators

`&&` (logical and)

`||` (logical or)

```
A = 'b0110; // non-zero
```

```
B = 'b0100; // non-zero
```

`A || B` is 1.

`A && B` is also 1

- Non-zero value is treated as 1
- If result is ambiguous, set it to x



Bit-wise Operators

~	(unary negation)
&	(binary and)
	(binary or)
^	(binary exclusive-or)
~^, ^~	(binary exclusive-nor)

A = 'b0110;

B = 'b0100;

A | B is 0110

A & B is 0100

- If operand sizes are unequal, smaller is zero-filled in the most significant bit side



Reduction Operators

&	(unary and)
~&	(unary nand)
	(unary or)
~	(unary nor)
^	(unary xor)
~^	(unary xnor)

```
A = 'b0110;
```

```
B = 'b0100;
```

```
| B is 1
```

```
& B is 0
```

- Bitwise operation on a single operand to produce 1-bit result



Shift Operators

<< (left shift)
>> (right shift)

```
reg [0:7] D;
```

```
D = 4'b0111;
```

```
D >> 2 has the value 0001
```

- Logic shift, fill vacant bits with 0
- If right operand is an x or a z, result is x
- Right operand is always an unsigned number



Conditional Operators

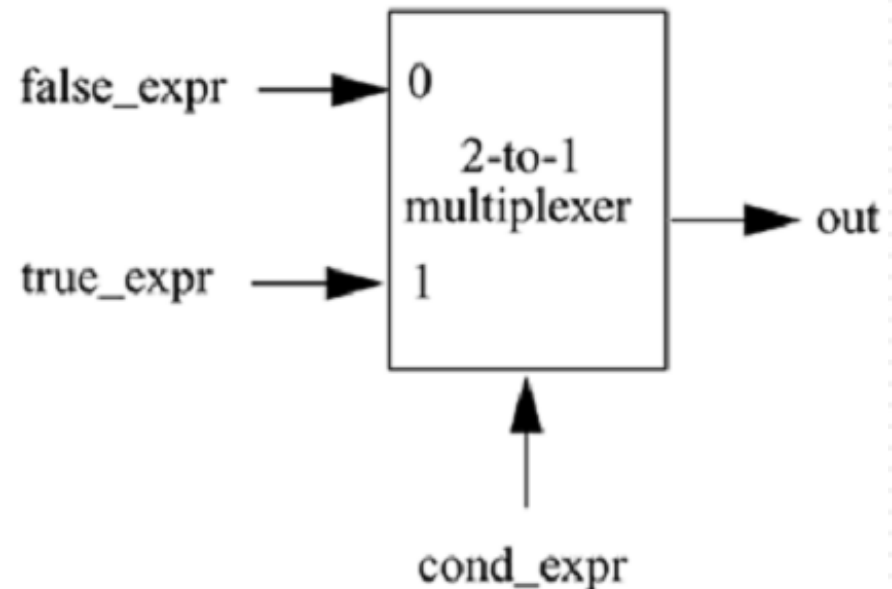
expr1 ? expr2:expr3

`wire [0:2] GRADE = SCORE > 60 ? PASS:FAIL;`

- If `expr1` is an `x` or a `z`, `expr2` and `expr3` are combined bit by bit (all `x`'s except `0` with `0 = 0`, `1` with `1 = 1`)

Example:

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





Concatenation and Replication

```
wire [7:0] DBUS;
```

```
wire [11:0] ABUS;
```

```
assign ABUS[7:4] = {DBUS[0], DBUS[1], DBUS[2], DBUS[3]};
```

```
assign ABUS = {DBUS[3:0], DBUS[7:4]};
```

```
assign DBUS[7:4] = {2{4'B1011}}; // 1011_1011
```

```
assign ABUS[7:4] = {{4{DBUS[7]}, DBUS}}; // sign extension
```



Examples

Write the Verilog description and test benches of the following circuits using dataflow modeling:

- 4-to-1 Multiplexer
 - Logic equation
 - Conditional operator
- 4-bit Full Adder
 - Dataflow operators
 - Full adder with carry lookahead
- Ripple Counter



Behavioral Modeling



Introduction

- Designers need to be able to evaluate the trade-offs of various architectures and algorithms in the earlier design stages
- Architectural evaluation takes place at an algorithmic level
- Verilog provides designers the ability to describe design functionality in an algorithmic manner
- In other words, the designer describes the behavior of the circuit
- Behavioral modeling represents the circuit at a very high level of abstraction



Procedure Blocks

- Use procedural blocks to describe the operation of the circuit
- Two procedural blocks:
 - *always* block: executes repetitively
 - *initial* block: executes once
- Concurrent procedural blocks
 - All execute concurrently
 - All activated at time 0



The initial Block

- An initial block starts at time 0, executes exactly once during a simulation
- Ports and variables can be initialized in declaration
- The initial block is always used in testbenches

Example:

```
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; //multiple  
    //statements; need to be grouped
```

```
    #25 b = 1'b0;
```

```
end
```

```
initial
```

```
begin
```

```
    #10 x = 1'b0;
```

```
    #25 y = 1'b1;
```

```
end
```

```
initial
```

```
    #50 $finish;
```

```
endmodule
```



The *always* Block

Can model:

- combinational logic
- sequential logic

Syntax:

```
// Single statement  
always @ (event expression)  
statement  
  
// Sequential statements  
always @ (event expression)  
begin  
    sequential statements  
end
```



The *always* Block (2)

- “event expression” specified a set of events based on which statements within the always block are executed sequentially
- The type of logic synthesized is based on what is specified in the “event expression”
- Four forms of event expressions are supported
 - An OR of several identifiers (comb/seq logic)
 - The rising edge clock (register inference)
 - The falling edge clock (register inference)
 - Asynchronous reset (register inference)



Event Expressions

I. An OR of several identifiers

- Combinational or synchronous logic may be represented by a set of sequential statements

```
always @ (id1 or id2 or id3 or ... or idn)  
begin  
    sequential_statements  
end
```

- A synchronous block may appear inside an always block (representing synchronous logic) in two forms:

```
always @ (posedge clock_name)  
begin  
    sequential_statements  
end
```

```
always @ (negedge clock_name)  
begin  
    sequential_statements  
end
```

- Sequential statements not within a sequential block represents combinational logic



Example

```
module Comb (A, B, C, Y);  
    input A, B, C;  
    output Y;  
    reg Y;  
  
    always @ (A or B or C)  
        begin  
            Y = A ^ B ^ C;  
        end  
endmodule
```



Event Expressions

2. The rising edge clock (register inference)

- ❑ The event expression denotes the rising edge of a clock
- ❑ The behavior within block represents synchronous logic triggered on the rising edge of clock

```
always @ (posedge CLK)
```

```
begin
```

```
    Q = D;
```

```
end
```

3. The falling edge clock (register inference)

- ❑ The event expression denotes the falling edge of a clock
- ❑ The behavior within block represents synchronous logic triggered on the rising edge of clock

```
always @ (negedge CLK)
```

```
begin
```

```
    Next_state = Current_state;
```

```
end
```



Register Inference

```
module SEQ (CLK, A, B, Y);  
  input CLK, A, B;  
  output Y;  
  reg Y;
```

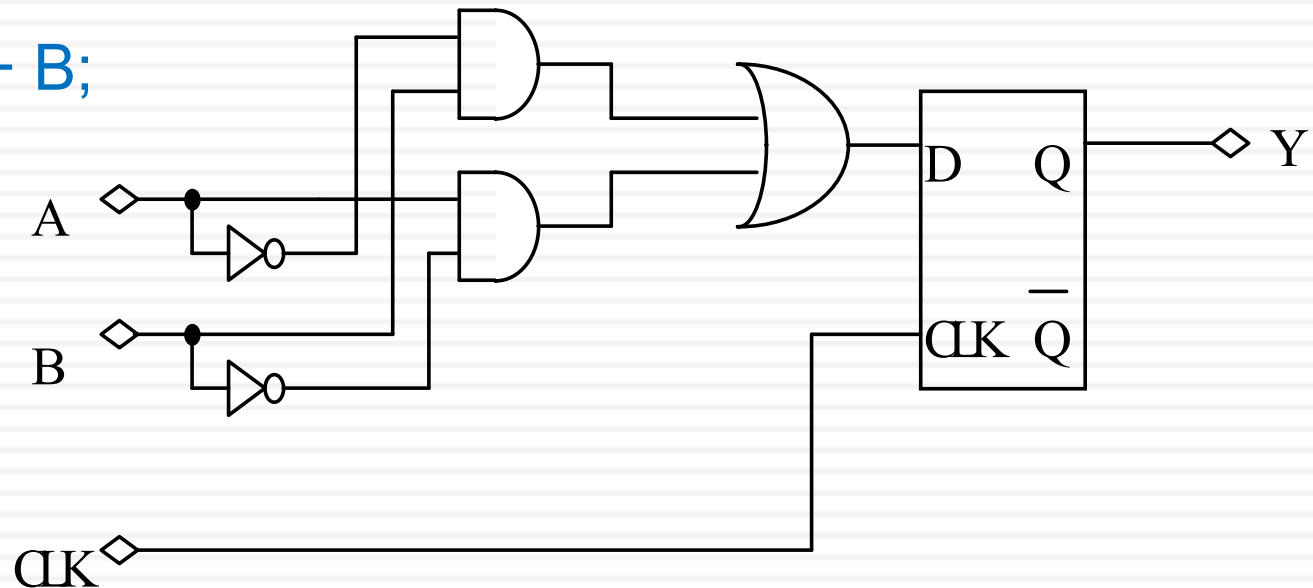
```
  always @ (posedge CLK)
```

```
  begin
```

```
    Y = A + B;
```

```
  end
```

```
endmodule
```





Event Expressions

4. Asynchronous reset (register inference)

- Asynchronous resets in addition to register inferences (2 & 3 above)

```
always @ (negedge reset1 or posedge CLK or posedge reset2)
begin
    if (!reset1)
        begin
            /* sequential_statements
            asynchronous input triggered by the false condition of reset1 to the registers */
            end
        else
            begin
                /* sequential_statement
                Optional for sequential statements. Could well have "else-if" clauses*/
                if (!reset2)
                    begin
                        /* sequential_statements: Asynchronous inputs triggered by reset2 */
                        end
                    else
                        begin
                            // sequential_statements: register inference statements.
                        end
            end
        end
end
```



Register Inference

- The language constructs “registers” is synthesized as a hardware register (flip-flop) if the register is assigned a value in:
 - A sequential block
 - An “always” block that has an event expression denoting a rising or falling clock edge
- It is illegal to assign a register value on both rising and falling edges of a clock



Sequential Statements and the Always Block

- Only “registers” and “integers” may be assigned values in sequential statements
- If an output port of the module is to be assigned a value in an always block it generally must be declared to be of a register type as well
- Possible sequential statements within an always block are
 - procedure assignment
 - synchronous block
 - if statement
 - case statement
 - for-loop statement
 - repeat loop statement
 - block statement
 - task enabling



Sequential Statements and the Always Block (2)

- An “always” block is concurrent and so may appear in any order within a module body with other continuous assignments (module instantiations or other always block)
- Data is passed out of an always block using register variables



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
- The syntax of procedural assignment
`assignment ::= variable_lvalue = [delay_or_event_control]
expression`
- The left-hand side of a procedural assignment <lvalue> can be
 - 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
- There are two types of procedural assignment statements: blocking and nonblocking



Blocking versus Non-Blocking

- ❑ Blocking procedural assignment:
 - ❑ Assignment is executed before any of the following ones are executed.
 - ❑ Only applies to its own sequential block.
`reg_a = 10;`
- ❑ Non-blocking procedural assignment.
 - ❑ The procedural flow is not blocked
`# 2`
`reg_a <= LOAD;`
`reg_b <= STORE;`
 - ❑ Evaluate right-hand side and schedules the assignment.
 - ❑ At the end of the current loop, assignment to the left-hand side is made



Blocking versus Non-Blocking (2)

□ Blocking (=)

- Assignment are blocked, i.e., they must be executed before subsequent statements are executed
always @ (posedge clock)

```
begin
    B = A;
    C = B;
    D = C;
end
```

- 1 flip-flop (Data in is A, data out is D)

□ Non-blocking (<=)

- Assignment are not blocked, i.e., can be scheduled to occur without blocking the procedural flow
always @ (posedge clock)

```
begin
    B <= A;
    C <= B;
    D <= C;
end
```

- 3 pipelined flip-flops (A to B to C to D)



Application of Non-blocking Assignments

- It is recommended that blocking and non-blocking assignments not be mixed in the same always block
- Using non-blocking assignments in place of blocking assignments is highly recommended in places where concurrent data transfers take place after a common event
- blocking assignments can potentially cause race conditions because the final result depends on the order in which the assignments are evaluated
- Typical applications of non-blocking assignments include pipeline modeling and modeling of several mutually exclusive data transfers
- On the downside, non-blocking assignments can potentially cause a degradation in the simulator performance and increase in memory usage



Example

- Blocking assignment

initial

begin

```
CLR = #5 0;
```

```
CLR = #4 1;
```

```
CLR = #10 1;
```

delay between *RHS* and *LHS*

end

// CLR is assigned at time 5, and then at 9 and then at 19

- Non-blocking assignment

initial

begin

```
CLR <= #5 1;
```

```
CLR <= #4 0;
```

```
CLR <= #10 1;
```

end

// CLR is assigned 0 at time 4, 1 at time 5 and 1 at time 10

- Value is indetermined if multiple values are assigned at the same time



Procedural Continuous Assignment

- Allow expression to be driven continuously into integers or nets
- *assign* and *deassign* procedural statements: for integers
- *force* and *release* procedural statements: for nets



Assign and Deassign

- An *assign* procedural statement overrides all procedural assignments to a register
- The *deassign* procedural statement ends the continuous assignment to a register
- Value remains until assigned again
- If *assign* applied to an already assigned register, it is *deassigned* first before making the new procedural continuous assignment



Example

```
module DFF (D, CLR, CLK, PRESET, Q);  
  input D, CLR, CLK, PRESET;  
  output Q;  
  reg Q;  
  always @ (CLR or PRESET)  
    if (CLR)  
      assign Q = 0; // D has no effect on Q  
    else if (PRESET)  
      assign Q = 1; // D has no effect on Q  
    else  
      deassign Q;  
  always @ (posedge CLK)  
    Q = D;  
endmodule;
```



Force and Release

- Similar to assign-deassign, except that it can be applied to nets as well as registers
- *force* procedural statement on a net overrides all drivers of the net, until a *release* is executed on the net

.....

or #1 (PRT, STD, DXZ);

initial

begin

force PRT = DXZ & STD;

#5 // Wait for 5 time units.

release PRT;

\$finish;

end



High-Level Constructs

- if statement
- loop statement (*forever, repeat, while, for*)
- *case* statement



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



Conditional Statements (2)

- Usage of conditional statements is shown below

//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 ;
```




Examples

```
//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");
```

```
//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");
```



Multi-way Branching (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`



case Statement

- 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
- The case statement compares 0, 1, x, and z values in the expression and the alternative bit for bit.



Example

```
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
```



casex, casez Keywords

- There are two variations of the case statement. They are denoted by keywords, casex and casez
 - casez treats all z values in the case alternatives or the case expression as don't cares
 - All bit positions with z can also be represented by ? in that position.
 - casex treats all x and z values in the case item or the case expression as don't cares
- The use of casex and casez allows comparison of only non-x or -z positions in the case expression and the case alternatives



Sequential Statements in a Block

- “case” statement
 - Specifies a multi-way branch based on the value of an expression
 - Is sequential and therefore may be nested to any level.
 - \$parallel ensures not priority encoded (less logic)

```
module EX_CASE (A, B, F, D, H, N);
  input A, B;
  input [3:0] F;
  inout H, N;
  output [1:0] D;
  reg H, N;
  reg [1:0] D;
  always @(A or B or F or H or N)
  begin
    case (F) // $parallel
      0, 4, 8, 9: D = {H, A};
      5: N = N & B;
      7: H = B;
      default: begin
        D = {H, A}; N = N & A; H = B;
      end
    endcase
  end
endmodule
```



case Statement (Full and Parallel)

```
module FULL (c1, c2, c3, c4, a, y);
  input c1, c2, c3, c4;
  output y;
  input [1:0] a;
  output [6:0] y;
  reg [6:0] y;
  always @ (posedge Pclk)
  begin
    if (Preset)
      begin
        y <= #2 {6'b0, 1'b0};
      end
    else casex ({c1, c2, c3, c4}) // $full $parallel
      4'b1x0x:    y[0] <= #2 a[0];
      4'b1x1x:    y[1:0] <= #2 a;
      4'b01x0:    y <= #2 y + 7'b1;
      4'b0101:    y <= #2 {y[6:1]+6'b1, 1'b0};
      4'b1111:    y <= #2 {y[6:2]+5'b1, 2'b0};
    endcase
  end
end
```



Loops

- There are four types of looping statements in Verilog: while, for, repeat, and forever
- All looping statements can appear only inside an initial or always block
- Loops may contain delay expressions
- Nested loops are allowed



while 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

Example:

```
//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
```



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

Example:

```
integer count;
```

```
initial
```

```
for ( count=0; count < 128;  
count = count + 1)
```

```
$display("Count = %d", count);
```

- for loops are generally used when there is a fixed beginning and end to the loop
- If the loop is simply looping on a certain condition, it is better to use the while loop



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 repeat construct must contain a number, which can be a constant, a variable or a signal value

Example:

```
//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
```



forever loop

- The keyword `forever` is used to express this loop
- The loop does not contain any expression and executes forever until the `$finish` task is encountered
- The loop is equivalent to a while loop with an expression that always evaluates to true
- A forever loop is typically used in conjunction with timing control constructs

Example:

//Example 1: Clock generation

```
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 +ve clock edge

```
reg clock;
```

```
reg x, y;
```

```
initial
```

```
    forever @(posedge clock) x = y;
```



Sequential and Parallel Blocks

- Block statements are used to group multiple statements to act together as one
- There are two types of blocks:
 - Sequential blocks
 - Parallel blocks
- Blocks can be named optionally
 - Registers can be declared locally
 - Blocks can be referenced (*disable* statement)
 - Can uniquely identify registers



Sequential blocks

- The keywords begin and end are used to group statements into sequential blocks
- Sequential blocks have the following characteristics
 - The statements in a sequential block are processed in the order they are specified
 - A statement is executed only after its preceding statement completes execution (except for nonblocking assignments with intra-assignment timing control)
 - If delay or event control is specified, it is relative to the simulation time when the previous statement in the block completed execution



Sequential Blocks (2)

begin

[: *block_id* { *declarations* }]

statements

end

// Waveform generation

begin

#2 stream = 1;

#7 stream = 0;

#10 stream = 1;

#14 stream = 0;

#16 stream = 1;

#21 stream = 0;

end



Parallel Blocks

- ❑ Parallel blocks, specified by keywords fork and join, provide interesting simulation features
- ❑ Parallel blocks have the following characteristics
 - ❑ Statements in a parallel block are executed concurrently
 - ❑ Ordering of statements is controlled by the delay or event control assigned to each statement
 - ❑ If delay or event control is specified, it is relative to the time the block was entered
- ❑ The order in which the statements are written in the block is not important
- ❑ Parallel blocks might cause implicit race conditions if two statements that affect the same variable complete at the same time



Parallel Blocks (2)

- Control passes out of block after all statements finish

fork

```
[ : block_id { declarations } ]
```

```
statements
```

join

```
// Waveform generation
```

fork

```
#2 stream = 1;
```

```
#5 stream = 0;
```

```
#3 stream = 1;
```

```
#4 stream = 0;
```

```
#2 stream = 1;
```

```
#5 stream = 0;
```

join



Examples

```
//Example 1: Parallel blocks  
with delay.
```

```
reg x, y;
```

```
reg [1:0] z, w;
```

```
initial
```

```
fork
```

```
    x = 1'b0; //completes at  
//simulation time 0
```

```
    #5 y = 1'b1; //completes at  
//simulation time 5
```

```
    #10 z = {x, y}; //completes at  
//simulation time 10
```

```
    #20 w = {y, x}; //completes  
//at simulation time 20
```

```
join
```

```
//Parallel blocks with deliberate  
//race condition
```

```
reg x, y;
```

```
reg [1:0] z, w;
```

```
initial
```

```
fork
```

```
    x = 1'b0;
```

```
    y = 1'b1;
```

```
    z = {x, y};
```

```
    w = {y, x};
```

```
join
```



Named Blocks

- Blocks can be given names.
 - Local variables can be declared for the named block
 - Named blocks are a part of the design hierarchy. Variables in a named block can be accessed by using hierarchical name referencing.
 - Named blocks can be disabled, i.e., their execution can be stopped

Example:

```
//Named blocks
module top;
initial
begin: block1 //sequential block named
block1
integer i; //integer i is static and local to
//block1
// can be accessed by hierarchical name,
//top.block1.i
...
...
End
```



Generate Blocks

- Generate statements allow Verilog code to be generated dynamically at elaboration time before the simulation begins
- This facilitates the creation of parameterized models.
- Generate statements are particularly convenient when the same operation or module instance is repeated for multiple bits of a vector, or when certain Verilog code is conditionally included based on parameter definitions
- Generate statements allow control over the declaration of variables, functions, and tasks, as well as control over instantiations
- All generate instantiations are coded with a module scope and require the keywords generate - endgenerate



Generate Blocks (2)

- Generated instantiations can be one or more of the following types
 - Modules
 - User defined primitives
 - Verilog gate primitives
 - Continuous assignments
 - initial and always blocks
 - Generated declarations and instantiations can be conditionally instantiated into a design
- Generated variable declarations and instantiations can be multiply instantiated into a design
 - Generated instances have unique identifier names and can be referenced hierarchically
 - Generate statements permit the following Verilog data types to be declared
 - net, reg
 - integer, real, time, realtime
 - event



Generate Blocks (3)

- There are three methods to create generate statements
 - Generate loop
 - Generate conditional
 - Generate case



Generate Loop

- A generate loop permits one or more of the following to be instantiated multiple times using a for loop:
 - Variable declarations
 - Modules
 - User defined primitives, Gate primitives
 - Continuous assignments
 - initial and always blocks



Example

```
// This module generates a bit-wise xor of two N-bit buses
module bitwise_xor (out, i0, i1);
// Parameter Declaration. This can be redefined
parameter N = 32; // 32-bit bus by default
// Port declarations
output [N-1:0] out;
input [N-1:0] i0, i1;
// Declare a temporary loop variable
genvar j;

//Generate the bit-wise Xor with a single loop
generate for (j=0; j<N; j=j+1) begin: xor_loop
xor g1 (out[j], i0[j], i1[j]);
end //end of the for loop inside the generate block
endgenerate //end of the generate block
endmodule
```




Generate Conditional

- A generate conditional is like an if-else-if generate construct that permits the following
- Verilog constructs to be conditionally instantiated into another module based on an expression that is deterministic at the time the design is elaborated
 - Modules
 - User defined primitives, Gate primitives
 - Continuous assignments
 - initial and always blocks



Example

```
// This is a parametrized multiplier
module multiplier (product, a0, a1);
// Parameter Declaration.
parameter a0_width = 8;
parameter a1_width = 8;
// Local Parameter declaration.
localparam product_width =
a0_width + a1_width;

// Port declarations
output [product_width -1:0]
product;
input [a0_width-1:0] a0;
input [a1_width-1:0] a1;
```

```
// Instantiate the type of multiplier
//conditionally depending on the
//value of the a0_width and a1_width

if (a0_width <8) || (a1_width < 8)
cla_multiplier #(a0_width, a1_width)
m0 (product, a0, a1);
else
tree_multiplier #(a0_width, a1_width)
m0 (product, a0, a1);
endgenerate //end of the generate
block

endmodule
```



Generate Case

- A generate case permits the following Verilog constructs to be conditionally instantiated into another module based on a select-one-of-many case construct that is deterministic at the time the design is elaborated
 - Modules
 - User defined primitives, Gate primitives
 - Continuous assignments
 - initial and always blocks



Example

```
// This module generates an N-bit
//adder
module adder(co, sum, a0, a1, ci);
// Parameter Declaration. This can
//be redefined
parameter N = 4; // 4-bit bus by
//default
// Port declarations
output [N-1:0] sum;
output co;
input [N-1:0] a0, a1;
input ci;
// Instantiate the appropriate adder
//based on the width of the bus.
// This is based on parameter N that
//can be redefined at instantiation
//time
```

```
generate
case (N)
//Special cases for 1 and 2 bit adders
1: adder_1bit adder1(c0, sum, a0, a1,
    ci); //1-bit implementation
2: adder_2bit adder2(c0, sum, a0, a1,
    ci); //2-bit implementation
// Default is N-bit adder
default: adder_cla #(N) adder3(c0,
    sum, a0, a1, ci);
endcase
endgenerate //end of the generate

endmodule
```



Timing Controls

- Timing control over when procedural statements can occur
- Delay control
 - # *delay*
 - Timing duration from time initially encountered the statement to the time it executes
- Event control
 - Statement execution is delayed until the occurrence of some simulation event
 - @ *symbol*
 - Edge-triggered control
 - Level-sensitive control



Delay Control

- The procedural statement execution is delayed by the specified delay
- If delay expression is x or z, it is 0
- If delay expression is negative, use two's complement unsigned integer

```
#2 TX = RX - 5;
```

```
# STROBE COMPARE = TX ^ MASK;
```

```
 #(PERIOD/2) CLOCK = ~CLOCK;
```



Edge-Triggered Event Control

```
@ (posedge CLOCK)    CURR_STATE = NEXT_STATE;
```

```
@ (posedge RESET)   COUNT = 0;
```

```
@ (CTRL_A or CTRL_B)      DBUS = 'bz;
```

```
@ CLA ZOO = FOO;
```

```
// Assign on any change of value in register CLA.
```

```
@ (posedge CLEAR or negedge RESET)      Q = 0;
```

- Negative edge: (1->x, z, or 0), (x or z -> 0)
- Positive edge: (0->x, z, or 0), (x or z -> 1)
- Events can be OR'ed as well to indicate “if any one of the events occur”



Level-Sensitive Event Control

- Execution of a procedural statement is delayed until a condition becomes true
- wait statement:
wait (*condition*)
statement
- If condition is already true, the next statement is evaluated immediately.
wait (SUM > 22) SUM = 0;



Intra-assignment Timing Control

- Timing control within an assignment
- Delay assigning right-hand side to left-hand side
- Right-hand side expression is evaluated before the delay

```
DONE = #5 A;
```

```
// is the same as
```

```
begin
```

```
    temp = A;
```

```
    #5 DONE = temp;
```

```
end
```

```
Q = @(posedge CLK) D;
```

```
// is the same as
```

```
begin
```

```
    temp = D;
```

```
    @(posedge CLK) Q = temp;
```

```
end
```

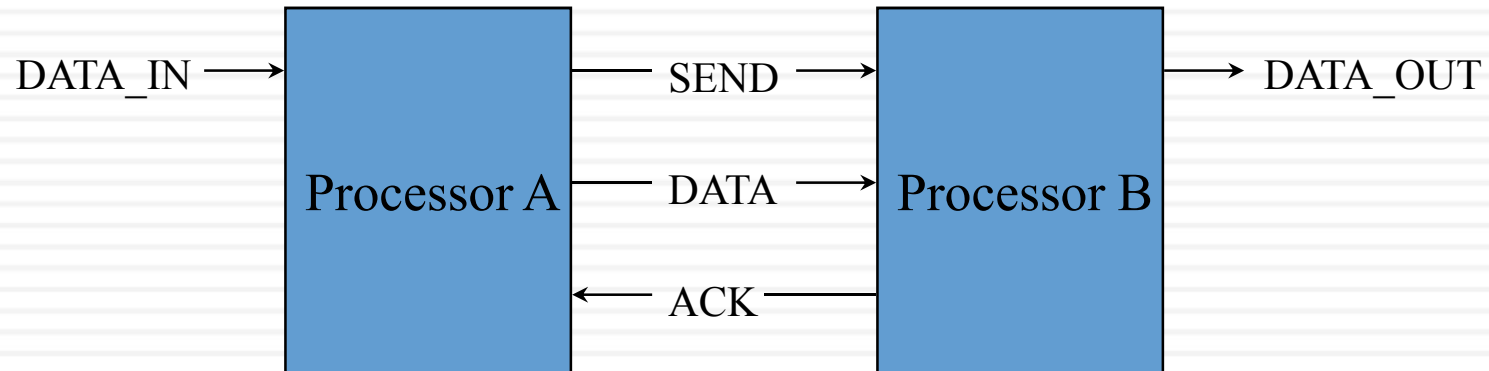


Example: D flip-flop with asynchronous reset

```
module DFF (CLK, D, PRESET, Q, QBAR);
    input CLK, D, PRESET;
    output Q, QBAR;
    reg Q, QBAR;
    always wait (PRESET == 1)
    begin
        #3 Q = 1;
        #2 QBAR = 0;
        wait (PRESET == 0);
    end
    always @ (negedge CLK)
    begin
        if (PRESET != 1)
        begin
            #5 Q = D;
            #1 QBAR = ~Q;
        end
    end
end
endmodule
```



Handshake Example



```
'timescale 1 ns/100ps
module HAND_SHAKE (DATA_IN, DATA_OUT);
    input [0:31] DATA_IN;
    output [0:31] DATA_OUT;
    reg SEND, ACK;
    reg [0:31] DATA;
    initial {ACK, SEND} = 0;
```



Handshake Example (2)

```
always
begin
    SEND = 1;
    DATA = DATA_IN;
    wait (ACK = 1);
    SEND = 0;
    #50;           // Wait for 50 time units.
end

always
begin
    #25;           // Wait for 25 time units.
    DATA_OUT = DATA;
    ACK = 1;
    #25 ACK = 0;
    wait (SEND == 1)
end
endmodule
```



Tasks and Functions



Introduction

- A designer is frequently required to implement the same functionality at many places in a behavioral design
- This means that the commonly used parts should be abstracted into routines and the routines must be invoked instead of repeating the code
- Most programming languages provide procedures or subroutines to accomplish this
- Verilog provides tasks and functions to break up large behavioral designs into smaller pieces
- Tasks and functions allow the designer to abstract Verilog code that is used at many places in the design



Differences between Tasks and Functions

Functions

- ❑ A function can enable another function but not another task
- ❑ Functions always execute in 0 simulation time
- ❑ Functions must not contain any delay, event, or timing control statements
- ❑ Functions must have at least one input argument. They can have more than one input
- ❑ Functions always return a single value. They cannot have output or inout arguments

Tasks

- ❑ A task can enable other tasks and functions
- ❑ Tasks may execute in non-zero simulation time
- ❑ Tasks may contain delay, event, or timing control statements
- ❑ Tasks may have zero or more arguments of type input, output, or inout
- ❑ Tasks do not return with a value, but can pass multiple values through output and inout arguments



Tasks and Functions

- Provides the ability to call common procedures from different places within a description
- Enables large procedures to be broken into smaller ones making reading and debugging easier
- The I/O passed in a task enabling statement must match that of the I/O in the task declaration

Example: (Task)

```
task PROC
  input A, B;
  inout D;
begin
  D = A + B
end
endtask
```

Example: (Task enabling for the left task)

```
always @ (in1 or in2)
  if (in1)
    PROC (in1, Y, result);
  else if (in2)
    PROC (in2, Y, result);
```




Tasks and Functions (2)

- Both tasks and functions must be defined in a module and are local to the module
- Tasks are used for common Verilog code that contains delays, timing, event constructs, or multiple output arguments
- Functions are used when common Verilog code is purely combinational, executes in zero simulation time, and provides exactly one output
- Functions are typically used for conversions and commonly used calculations



Tasks and Functions (3)

- ❑ Tasks can have input, output, and inout arguments; functions can have input arguments
- ❑ In addition, they can have local variables, registers, time variables, integers, real, or events
- ❑ Tasks or functions cannot have wires. Tasks and functions contain behavioral statements only.
- ❑ Tasks and functions do not contain always or initial statements but are called from always blocks, initial blocks, or other tasks and functions



Tasks

- ❑ Tasks are declared with the keywords `task` and `endtask`
- ❑ Tasks must be used if any one of the following conditions is true for the procedure:
 - ❑ There are delay, timing, or event control constructs in the procedure.
 - ❑ The procedure has zero or more than one output arguments.
 - ❑ The procedure has no input arguments



Tasks (2)

- ❑ Tasks are normally static in nature. All declared items are statically allocated and they are shared across all uses of the task executing concurrently
- ❑ Therefore, if a task is called concurrently from two places in the code, these task calls will operate on the same task variables
- ❑ It is highly likely that the results of such an operation will be incorrect
- ❑ To avoid this problem, a keyword `automatic` is added in front of the task keyword to make the tasks re-entrant. Such tasks are called automatic tasks
- ❑ All items declared inside automatic tasks are allocated dynamically for each invocation



Tasks (3)

- Procedure
- Can contain timing control
- Can call other functions and tasks
- 0 or more arguments
- Output or input arguments can be updated
- Task definition:

```
task task_id;  
    [ declarations ]  
    statements  
endtask
```

- Cannot declare a new type within a task



Task Example

```
parameter MAXBITS = 8;
```

```
task REVERSE_BITS;
```

```
    input [MAXBITS-1 : 0] DIN;
```

```
    output [MAXBITS-1 : 0] DOUT;
```

```
    integer K;
```

```
    begin
```

```
        for (K=0;K<MAXBITS;K=K+1)
```

```
            DOUT [MAXBITS-K] = DIN[K];
```

```
    end
```

```
endtask
```



Task Calling

- Task calling:

```
task_id [ (expr1, expr2, ..., exprN) ];
```

- List of arguments must match the order of arguments in task definition
- Arguments are passed by value
- Task can be called more than once concurrently
- Local variables are static; if concurrently called, same local variables are shared
- Calling statement for task REVERSE_BITS:

```
// Declarations:
```

```
reg [MAXBITS-1 : 0] REG_X, NEW_REG_X;
```

```
REVERSE_BITS (REG_X, NEW_REG_X); // Calling task.
```



Example

```
...
parameter delay = 10;
reg [15:0] A, B;
reg [15:0] AB_AND,
AB_OR, AB_XOR;
always @(A or B)
begin
//invoke the task
//bitwise_oper.
bitwise_oper(AB_AND,
AB_OR, AB_XOR, A, B);
end
...
```

```
//define task bitwise_oper
task automatic
bitwise_oper (output [15:0]
ab_and, ab_or, ab_xor,
input [15:0] a, b);
begin
#delay ab_and = a & b;
ab_or = a | b;
ab_xor = a ^ b;
end
endtask
...
```




Functions

- ❑ Functions are declared with the keywords function and endfunction
- ❑ Functions are used if all of the following conditions are true for the procedure
 - ❑ There are no delay, timing, or event control constructs in the procedure
 - ❑ The procedure returns a single value
 - ❑ There is at least one input argument
 - ❑ There are no output or inout arguments
 - ❑ There are no nonblocking assignments



Functions

- Executes in one simulation time unit.
- No delays.
- Cannot call another task.
- Must have at least one input.
- Returns a value that can be used in an expression.
- Function definition:

function [range] function_id;

input_declarations

other_declarations

statements

endfunction

- If no range is specified, 1 bit is assumed.



Function Example

```
parameter MAXBITS = 8;
```

```
function [MAXBITS-1:0] REVERSE_BITS;
```

```
input [MAXBITS-1 : 0] DIN;
```

```
reg K;
```

```
begin
```

```
    for (K=0;K<MAXBITS;K=K+1)
```

```
        REVERSE_BITS [MAXBITS-K] = DIN[K];
```

```
    end
```

```
endfunction
```

- Implicit declaration of a reg, same as function name
- Must include assignment to function name



Function Call

- Can be used in any expression

- Function call:

```
function_id [ (expr1, expr2, ..., exprN) ];
```

- Calling function REVERSE_BITS:

```
// Declarations:
```

```
reg [MAXBITS-1 : 0] REG_X, NEW_REG_X;
```

```
NEW_REG_X = REVERSE_BITS (REG_X);
```



Function Call

- Return a value (unlike task)
- Must have at least one input argument
- Can not enable task, but task may enable other tasks and functions

Example: (function)

```
function [3:0] DEPTH
    input D, T;
    reg [3:0] R, p;
begin
    if (D)
        p = p*2;
    else if (~T)
        R = *p;
    DEPTH = R*p;
end
endfunction
```

Example: (calling function)

```
// RESULT is a 4-bit wire
assign RESULT = DEPTH (in1, in2);
```



Example

```
//Define a module that contains the
//function calc_parity
module parity;
...
reg [31:0] addr;
reg parity;
//Compute new parity whenever
//address value changes
always @(addr)
begin
parity = calc_parity(addr); //First
invocation of calc_parity
$display("Parity calculated = %b",
calc_parity(addr) );
//Second invocation of calc_parity
end
```

```
//define the parity calculation
function
function calc_parity (input [31:0]
address);
begin
//set the output value
appropriately. Use the implicit
//internal register calc_parity.
calc_parity = ^address; //Return
the xor of all address bits.
end
endfunction

...
endmodule
```



Automatic (Recursive) Functions

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



Other Topics



System Tasks

□ Display System Tasks

- **\$display** and **\$monitor** system tasks.
- **\$write** task: Write the specified argument values to output, but does not add a newline character (as opposed to **\$display** which does)

\$write | **\$writeh** | **\$writeb** | **\$writeo** (*arg1, arg2, ..., argN*)

■ Different default bases

\$write (“simulation time is ”);

\$write (“%t\n”, **\$time**);

- **\$strobe** task: Display run data at selected time but at end of current simulation time.

always @ (posedge RST)

\$strobe (“the flip-flop value is %b at time %t”, Q, **\$time**);

/ After end of simulation time when RST has a positive edge, the \$strobe task prints the values of Q and current simulation time. */*

- %h, %d, %o, %b, %c, %m, %s



System Tasks (2)

□ File I/O system tasks

- **\$fopen, \$fclose**
- **\$fdisplay, \$fwrite, \$fstrobe, \$fmonitor**
- **\$readmemb, \$readmemh**: Loads memory data from a file

```
reg [0:3] MEM_A [0:63];
```

```
$readmemb ("ones_and_zeros.vec", MEM_A);
```

□ Simulation control system tasks

- **\$finish;** // make the simulation exit
- **\$stop;** // Causes simulation to suspend

□ Simulation time system functions

- **\$time** // 64-bit time value is returned
- **\$stime** // 32-bit unsigned integer time value is returned
- **\$realtime** // return real number time



Disable Statement

- Can be used to terminate a task or a block before it finishes executing all its statements
- Used to model hardware interrupts and global resets

```
disable task_id;
```

```
disable block_id;
```

- Execution continues with the next statement

```
begin BLK_A
```

```
    // statement 1
```

```
    disable BLK_A;
```

```
    // statement 2
```

```
end
```

```
    // statement 3
```

```
    // Statement 2 is never executed. After disable statement is executed,  
    // statement 5 is executed
```



Value Change Dump File

- Contain information about value changes on specified variables in design

```
$dumpfile ("uart.dump");
```

```
$dumpvars (level_num, CLK, CLR);
```

```
$dumpvars; // Dumps all variables.
```

```
$dumpoff ;
```

```
$dumpon;
```

```
$dumpall;
```

```
$dumplimit (file_size);
```

```
$dumpflush;
```



Compiler Directives

- Always begins with a character ' '
- Definition holds accross multiple files for one compilation

```
'default_nettype wand // specifies net type for implicit net
//
'define WORD 16 // Create a macro for text substitution
'undef WORD // Undefines a previously defined text macro.
'ifdef SUN
...
['else
...
'endif ] // Conditional compilation
'include ".././rx.h" // Inserted the contents of the specified file.
'resetall // All compiler directives are reset to default.
'timescale 1 ns / 10 ps // 1ns is the time unit and delays are rounded to
10ps
```



Nested Macros

```
module NESTED_MACROS (A, B, C, D, Y);  
input A, B, C, D;  
output Y;  
  
reg Y;  
  
'define ONE 1'b1  
'define ZERO 1'b0  
'define TEST (A == 'ONE)  
  
always @ (A or B or C or D)  
if ('TEST && (C != D))  
Y = 1;  
else  
Y = 0;  
  
endmodule
```



Specify Block

- Used to assign delay to these paths
- Used to describe paths between a source and a destination
- Used to perform timing checks.

specify

```
spec_param_declarations           // Declares parameters
```

```
// for use only within the specify block.
```

```
path_declarations
```

```
system_timing_checks
```

endspecify



Specify Block (2)

```
specify
```

```
    specparam tCLK_Q = (4:5:6);
```

```
// Path delays:
```

```
(CLOCK*> Q) = tCLK_Q;
```

```
(DATA*> Q) = 12;
```

```
(CLEAR, PRESET*> Q) = (4:5:3);
```

```
/* Can specify pulse width to be rejected and a range  
   for which to generate an x */
```

```
    specparam PATHPULSE$ = (1,2);
```

```
// Reject limit = 1, error limit = 2.
```

```
    specparam PATHPULSE$DATA$Q = 6;
```

```
endspecify
```




Modeling for Synthesis



Modeling Structures

- Netlist
 - structural description
- Primitives
- Continuous assignment
 - Data flow specification
 - Verilog operators
- Procedural blocks
 - always and initial blocks
 - Allow timing control and concurrency
 - C-like procedural statements
- Task and function



Continuous Assignments

- Model combinational logic
- Operands + operators
- Drive values to a net
 - `assign out = a & b;`
 - `assign eq = (a==b);`
 - `wire #10 inv = ~in;`
 - `wire [7:0] c = a + b;`
- Avoid logic loops
 - `assign a = a + b;`
 - asynchronous design



Operator Precedence

[] bit-select or part select

() parenthesis

!, ~ logic and bit-wise
negation

&, |, ~&, ~|, ^, ~^, ^~
reduction operators

+, - unary arithmetic

{ } concatenation

*, /, % arithmetic

+, - arithmetic

>>, << shift

>, >+, <, <=

relational

==, != logical equality

& bit-wise AND

^, ^~, ~^ bit-wise XOR
and XNOR

| bit-wise OR

&& logical AND

|| logical or

?: conditional



RTL Model

- Describe the system at a higher level of abstraction
- Specify a set of concurrently active procedural blocks
- Procedural blocks
 - *initial* blocks
 - For test-fixtures to generate test vectors
 - *always* blocks
 - Can be combinational circuits
 - Can infer latches or flip-flops
 - Procedural blocks have the following components
 - Procedural assignment statements
 - timing controls
 - high-level programming language constructs



RTL Statements

- Procedural and RTL assignments
 - reg and integer
 - `out = a+ b;`
- begin ... end block statements
 - group statements
- if ... else statements
- case statements
- for loops
- while loops
- forever loops
- disabled statements
 - Disable a named block



Combinational Always Blocks

- A complete sensitivity list (inputs)

```
always @ (a or b or c)
```

```
f = a&~c | b&c;
```

- Simulation results

```
always @ (a or b) // conditions are ignored by synthesizer
```

```
f = a&~c | b&c;
```

- Parentheses

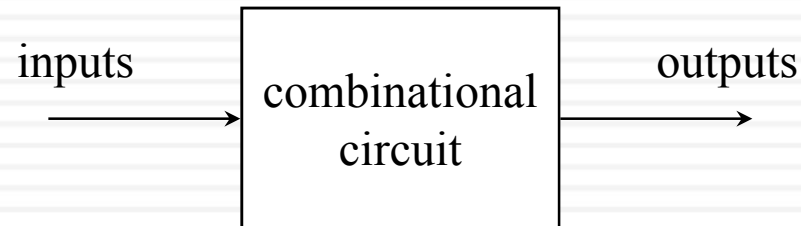
```
always @ (a or b or c or d)
```

```
z = a+b+c+d; // z = (a+b) + (c+d);
```



Combinational Circuit Design

- Outputs are functions of inputs



- Examples
 - Mux
 - Decoder
 - Priority encoder
 - Adder



Multiplexer

□ Net-list (gate-level)

```
module mux2_1 (out, a, b, sel);
    output out;
    input a, b, sel;
    not (sel_, sel);
    and (a1, a, sel_);
    and (b1, b, sel);
    or (out, a1, b1);
endmodule
```

□ Continuous assignment

```
module mux2_1 (out, a, b, sel);
    output out;
    input a, b, sel;
    assign out = (a&~sel) | (b&sel);
endmodule
```

□ RTL modeling

```
module mux2_1 (out, a, b, sel);
    output out;
    input a, b, sel;
    always @ (a or b or sel)
        if (sel)
            out = b;
        else
            out = a;
endmodule
```



Multiplexer (2)

□ 4-to-1 multiplexor

```
module mux4_1 (out, in0, in1,
in2, in3, sel);
output out;
input in0, in1, in2, in3;
input [1:0] sel;
```

```
assign out = (sel == 2'b00) ?
in0 :
```

```
(sel == 2'b01) ? in1 :
```

```
(sel == 2'b10) ? in2 :
```

```
(sel == 2'b11) ? in3 :
```

```
1'bx;
```

```
endmodule
```

□ 4-to-1 multiplexor

```
module mux4_1 (out, in, sel);
output out;
input [3:0] in;
input [1:0] sel;
reg out;
```

```
always @ (sel or in) begin
case (sel)
```

```
2'd0: out = in[0];
```

```
2'd1: out = in[1];
```

```
2'd2: out = in[2];
```

```
2'd3: out = in[3];
```

```
default: out = 1'bx;
```

```
endcase
```

```
end
```

```
endmodule
```



Decoder (3-8 decoder with an enable control)

```
module decoder (o, enb_, sel);
    output [7:0] o;
    input enb_;
    input [2:0] sel;
    reg [7:0] o;
    always @ (enb_ or sel)
        if (enb_)
            o = 8'b1111_1111;
        else
            case (sel)
                3'b000: o = 8'b1111_1110;
                3'b001: o = 8'b1111_1101;
                3'b010: o = 8'b1111_1011;
                3'b011: o = 8'b1111_0111;
                3'b100: o = 8'b1110_1111;
                3'b101: o = 8'b1101_1111;
                3'b110: o = 8'b1011_1111;
                3'b111: o = 8'b0111_1111;
                default: o = 8'bx;
            endcase
    endmodule
```



Priority Encoder

always @ (d0 or d1 or d2 or d3)

if (d3 == 1)

{x, y, v} = 3'b111;

else if (d2 == 1)

{x, y, v} = 3'b101;

else if (d1 == 1)

{x, y, v} = 3'b011;

else if (d0 == 1)

{x, y, v} = 3'b001;

else

{x, y, v} = 3'bxx0;

inputs				outputs		
d0	d1	d2	d3	x	y	v
0	0	0	0	x	x	0
1	0	0	0	0	0	1
x	1	0	0	0	1	1
x	x	1	0	1	0	1
x	x	x	1	1	1	1



Parity Checker

```
module parity_checker (data, parity);
    input [0:7] data;
    output parity;
    reg parity;
    always @ (data)
        begin: check_parity
            reg partial;
            integer n;
            partial = data[0];
            for (n=1; n<=7; n=n+1)
                begin
                    partial = partial^data[n];
                end
            parity <= partial;
        end
    endmodule
```



Adder

- RTL modeling

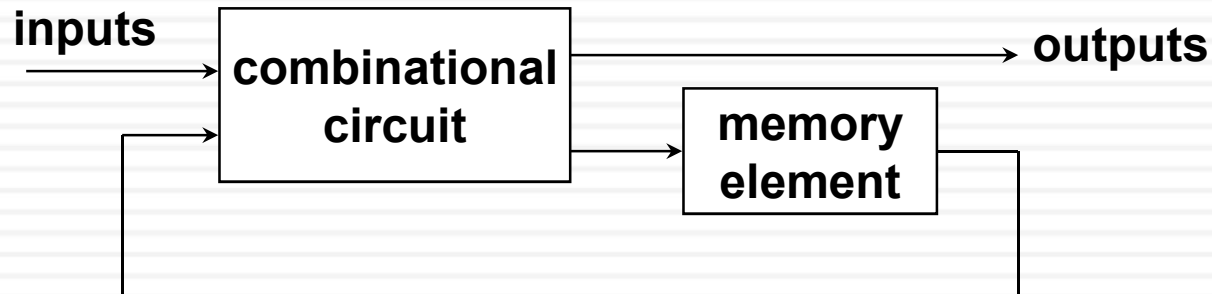
```
module adder (c, s, a, b);  
    output c;  
    output [7:0] s;  
    input [7:0] a, b;  
  
    assign {c, s} = a + b;  
endmodule
```

- Logic synthesis

- Carry Look-Ahead (CLA) adder for speed optimization
- Ripple adder for area optimization



Sequential Circuit Design



- A feedback path
- The state of the sequential circuits
- The state transition
 - Synchronous circuits
 - Asynchronous circuits



Register Inference

- Inference of Positive Edge Triggered Flip-flops

```
module ff1 (data, clk, q);  
    input [3:0] data;  
    input clk;  
    output [3:0] q;  
    reg [3:0] q;  
    always (posedge clk)  
        q = data;  
endmodule
```

- Inference of Positive Edge Triggered Flip-flops with active low reset

```
module inf_ff (clk, reset, a,  
    b, q);  
    input clk, reset;  
    input [3:0] a, b;  
    output [3:0] q;  
    reg [3:0] q;  
    always (posedge  
    clk or negedge reset)  
        if (!reset) q = 0;  
    //asynchronous input first  
        else q = a&b;  
endmodule
```




Variable Within Always

- Redundant register variables can cause extra logics

```
module BAD_DESIGN (D, CLK, RN, Q, QBAR);
```

```
input D, CLK, RN;
```

```
output Q, QBAR;
```

```
reg Q, QBAR;
```

```
always (negedge RN or posedge CLK)
```

```
begin
```

```
if (! RN) begin
```

```
Q = 1'b0;
```

```
QBAR = 1'b1;
```

```
end
```

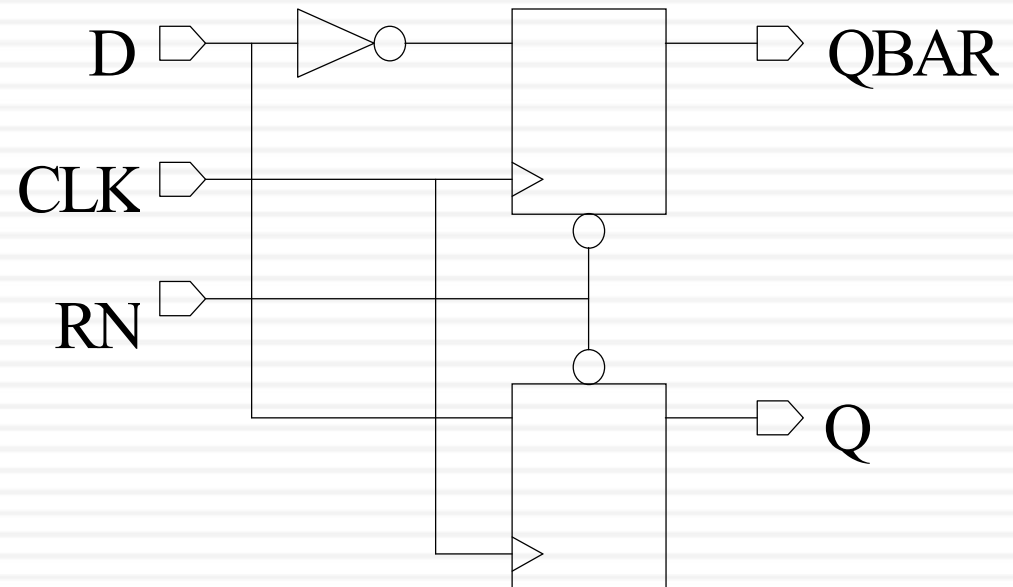
```
else begin
```

```
Q = D;
```

```
QBAR = ~D;
```

```
end
```

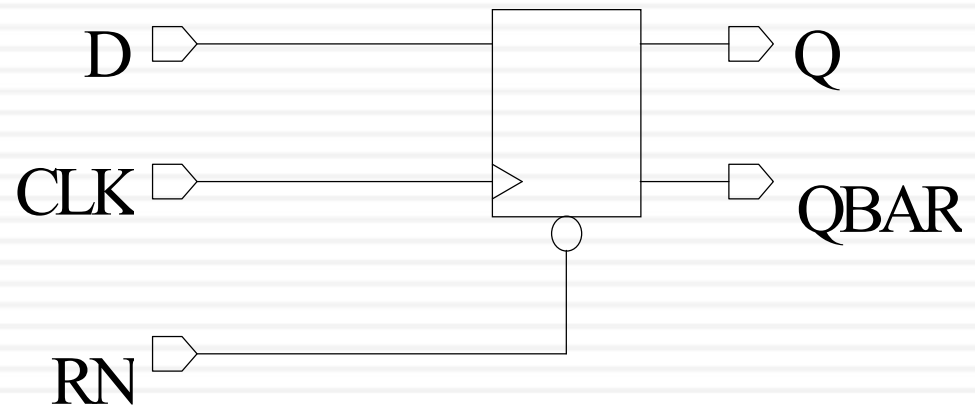
```
endmodule
```





Variable Within Always (2)

```
module GOOD_DESIGN (D, CLK, RN, Q, QBAR);  
  input D, CLK, RN;  
  output Q, QBAR;  
  reg Q, QBAR;  
  
  always (negedge RN or posedge CLK)  
  begin  
    if (! RN)  
      Q = 1'b0;  
    else  
      Q = D;  
    end  
    assign ABAR = ~Q;  
  
endmodule
```





D Latches

□ D latch

always @ (enable or data)

if (enable)

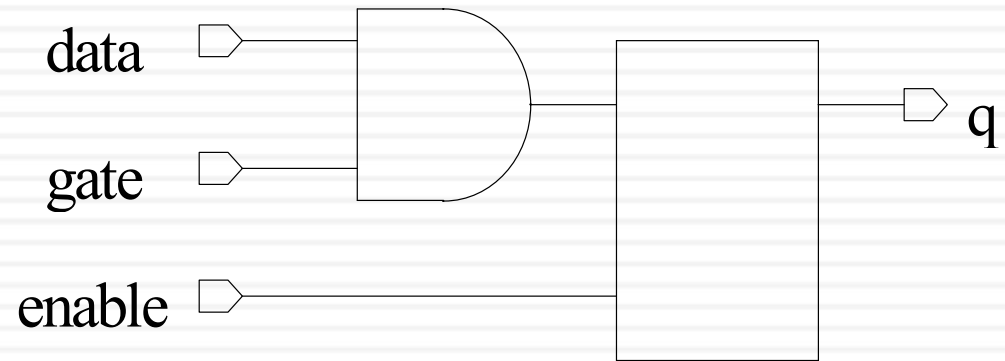
q = data;

□ D latch with gated asynchronous data

always @ (enable or data or gate)

if (enable)

q = data&gate;





Latches

□ D latch with gated “enable”

always @ (enable or data or gate)

if (enable & gate)

q = data;

□ D latch with asynchronous reset

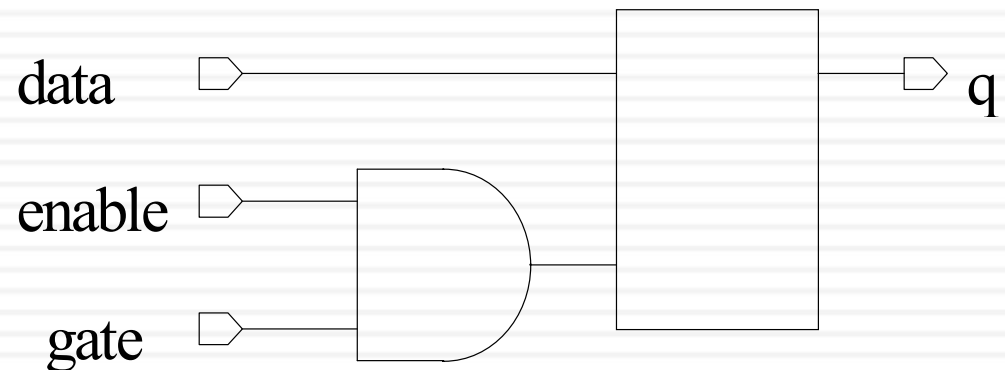
always @ (reset or data)

if (reset)

q = 1'b0;

else if (enable)

q = data;





Latch Inference (2)

- What if g is false?

```
module latch4 (d, en, g, k);  
  input [3:0] d;  
  input en, g;  
  output [3:0] k;  
  reg [3:0] k;  
  always (d or en or g)  
    if (g)  
      k = d & {en, en, en, en};  
endmodule
```

- y and z are not fully specified

```
always @ (control or a or b or c)  
begin  
  if (control > 2)  
    begin  
      x = a;  
      y = b;  
      z = c;  
    end  
  else x = b;  
end
```



Latch Inference (3)

- What if bcd is 5?

```
wire [3:0] bcd;
```

```
...
```

```
case (bcd)
```

```
  4'd0: begin zero = 1'b1; {one, two} = 2'b0; end
```

```
  4'd1: begin {zero, two} = 2'b0; one = 1'b1; end
```

```
  4'd2: begin {zero, one} = 2'b0; two = 1'b1; end
```

```
endcase
```

- Using default to prevent latch inference

```
wire [3:0] bcd;
```

```
...
```

```
case (bcd)
```

```
  4'd0: begin zero = 1'b1; {one, two} = 2'b0; end
```

```
  4'd1: begin {zero, two} = 2'b0; one = 1'b1; end
```

```
  4'd1: begin {zero, one} = 2'b0; two = 1'b1; end
```

```
  default: {zero, one, two} = 3'bxxx;
```

```
endcase
```



Latch Inference (4)

- Why latches are inferred?

```
wire [3:0] bcd;
```

```
...
```

```
case (bcd)
```

```
  4'd0: zero = 1'b1;
```

```
  4'd1: one = 1'b1;
```

```
  4'd1: two = 1'b1;
```

```
  default: {zero, one, two} = 3'xxx;
```

```
endcase
```

- Using full_case directive to prevent latch inference

```
wire [3:0] bcd;
```

```
...
```

```
case (bcd) //synopsys full_case
```

```
  4'd0: begin zero = 1'b1; {one, two} = 2'b0; end
```

```
  4'd1: begin {zero, two} = 2'b0; one = 1'b1; end
```

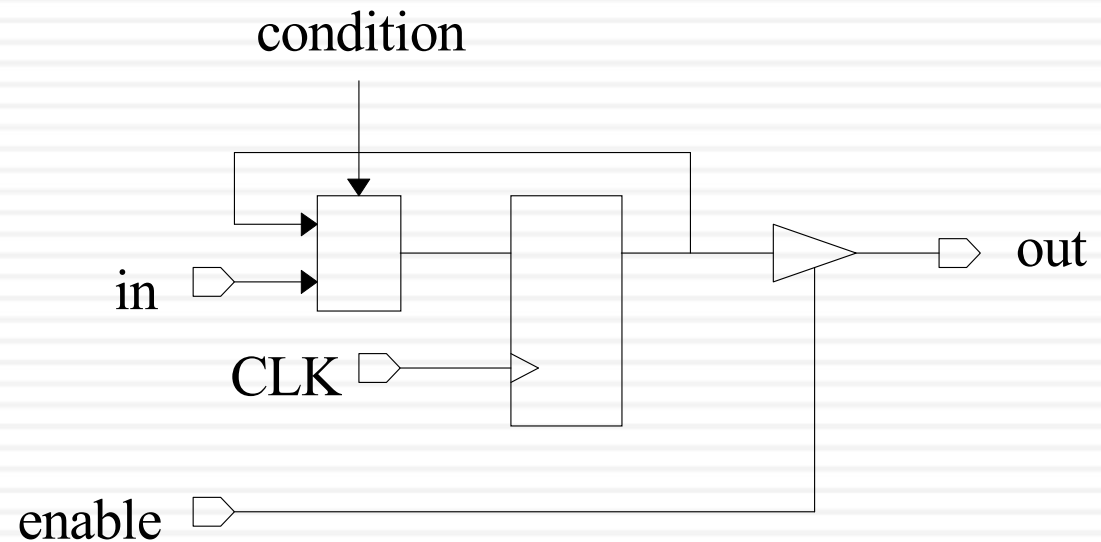
```
  4'd1: begin {zero, one} = 2'b0; two = 1'b1; end
```

```
endcase
```



Registered Three-State

```
always (posedge CLK)
begin
  if (enable)
    out = (~condition) ? in : out;
  else
    out = 1'bz;
end
```





Inefficient Description

```
module count ( clock, reset, and_bits, or_bits, xor_bits);  
input clock, reset;  
output and_bits, or_bits, xor_bits;  
reg and_bits, or_bits, xor_bits;  
reg [2:0] count;  
    always @ (posedge clock) begin  
        if (reset)  
            count = 0;  
        else  
            begin  
                count = count + 1;  
                and_bits = &count;  
                or_bits = |count;  
                xor_bits = ^count;  
            end  
        end  
    end  
Endmodule    //Six inferred registers
```



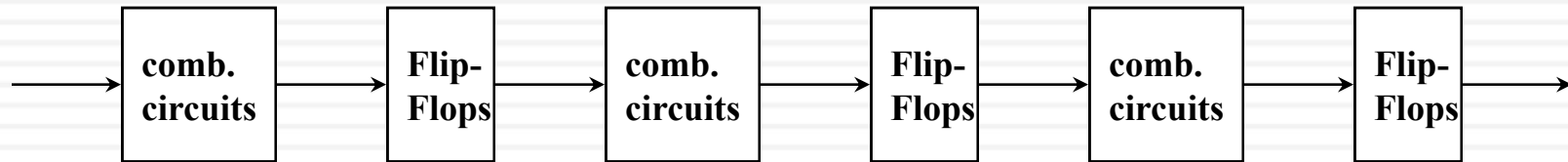
Efficient Description

- Separate combinational and sequential circuits

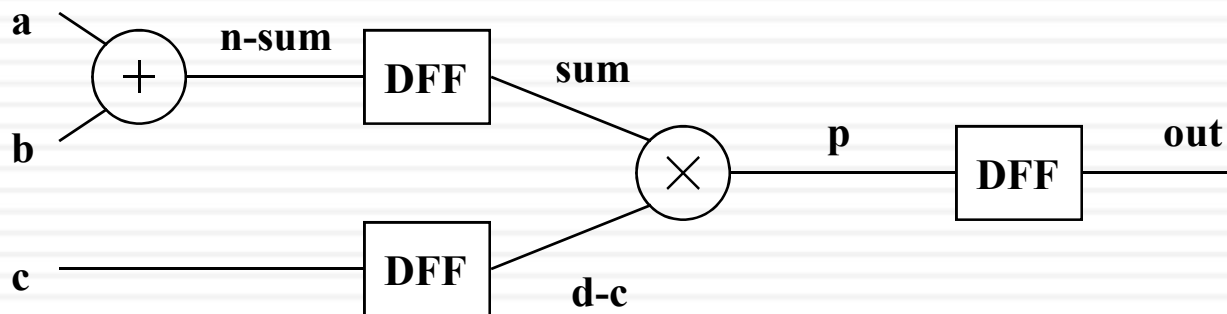
```
module count ( clock, reset, and_bits, or_bits, xor_bits);
input clock, reset;
output and_bits, or_bits, xor_bits;
reg and_bits, or_bits, xor_bits;
reg [2:0] count;
    always @ (posedge clock) begin
        if (reset)
            count = 0;
        else
            count = count + 1;
    end
    always @ (count) begin           // combinational circuits
        and_bits = &count;
        or_bits = |count;
        xor_bits = ^count;
    end
endmodule  \\Three registers are inferred
```



Pipelines



□ An example



```
assign n_sum = a + b;
```

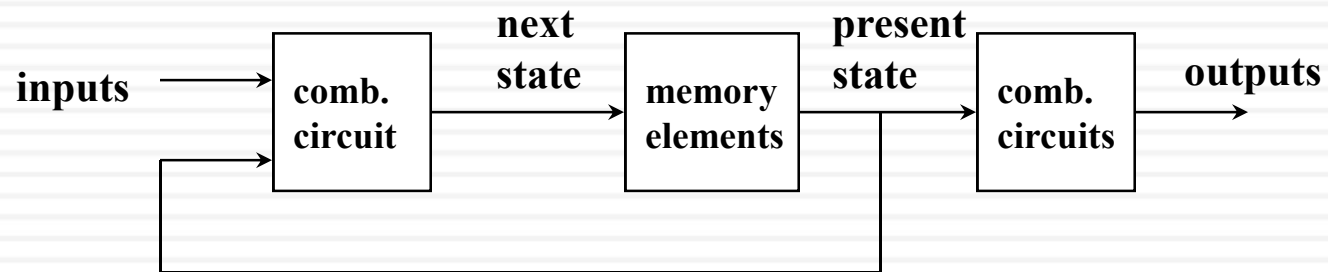
```
assign p = sum*d_c;
```

```
// plus D flip-flops
```

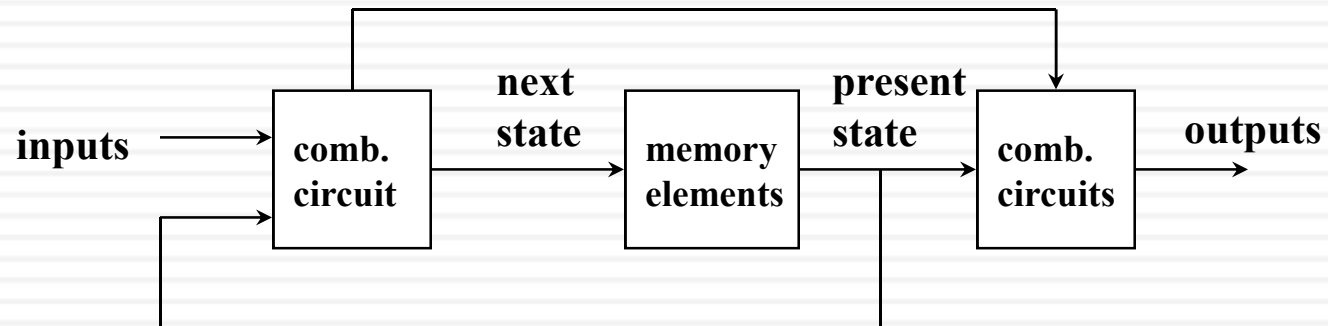


Finite State Machine

□ Moore model



□ Mealy model





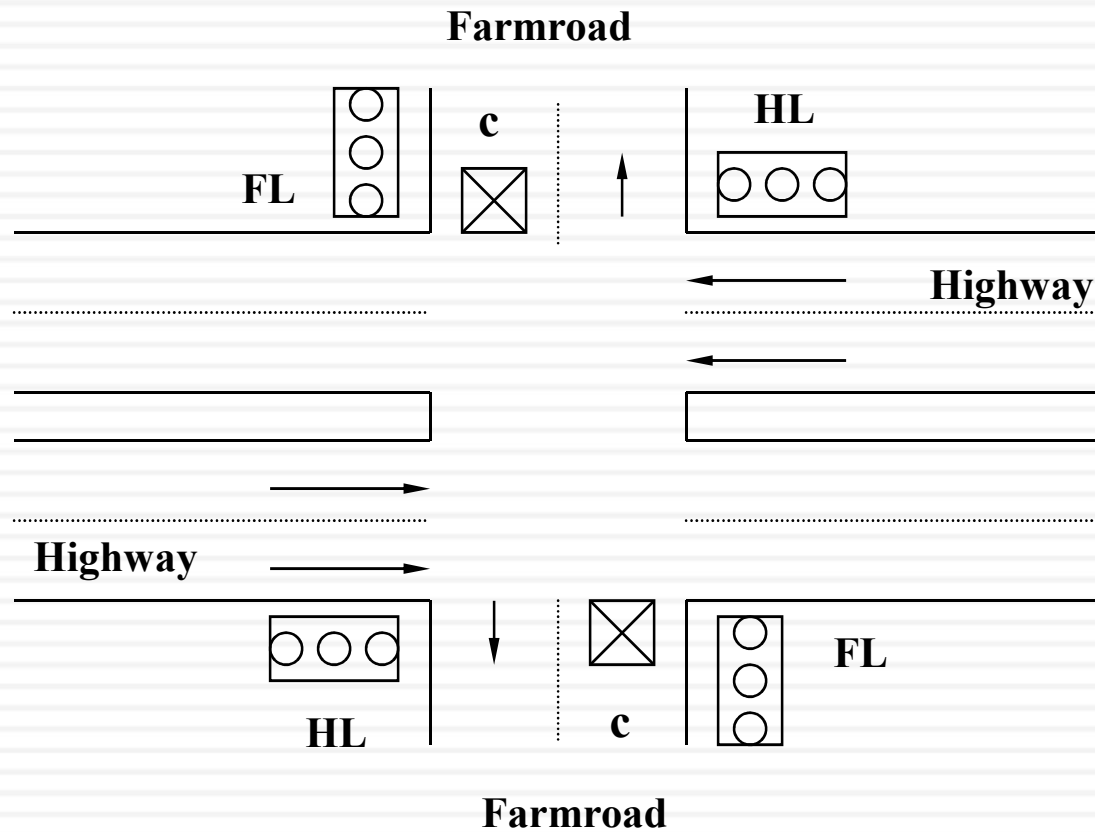
Mealy Machine Example

```
module mealy (in1, in2, clk, reset, out);
input in1, in2, clk, reset;
output out;
reg present_state, next_state, out;
    // state flip-flops
    always @ (posedge clk or negedge reset)
        if (!reset)
            present_state = 0;
        else
            present_state = next_state;
    // combinational circuits
    always @ (in1 or in2 or present_state)
        case (present_state)
            0: begin
                next_state = 1;
                out = 1'b0;
            end
            1: begin
                next_state = 0;
                out = 1'b1;
            end
        endcase
endmodule
```



A FSM Example

□ Traffic Light Controller





Specification

□ Input Signal

Reset

place control in initial state

C

detects vehicle on farmroad in either direction

TS

short timer interval has expired

TL

long timer interval has expired

□ Output Signal

HG, HY, HR

assert green, yellow, red highway lights

FG, FY, FR

assert green, yellow, red farmroad lights

ST

commence timing a long or short interval

□ State

S0

highway green (farmroad red)

S1

highway yellow (farmroad red)

S2

farmroad green (highway red)

S3

farmroad yellow (highway red)

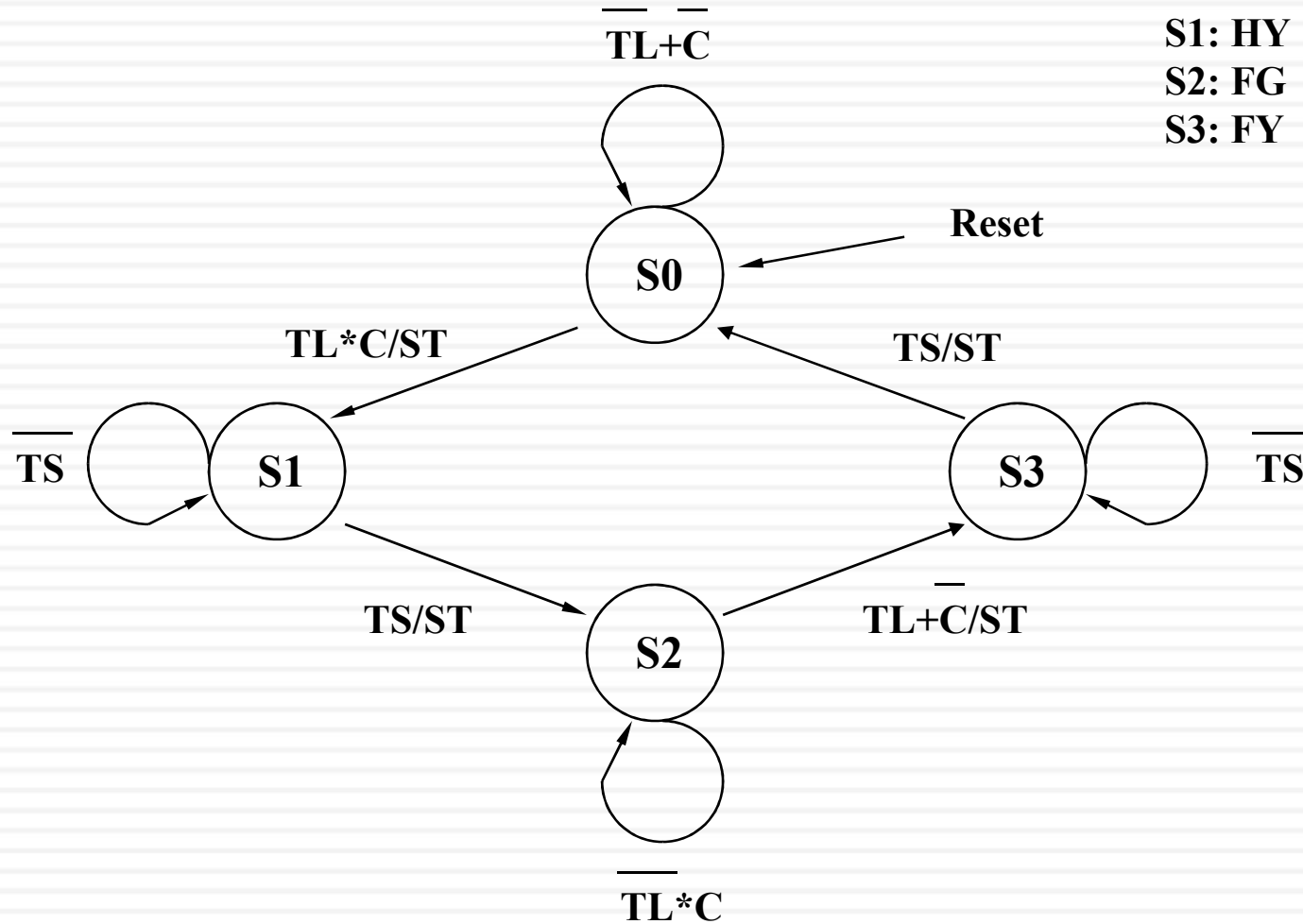
Description

Description

Description



State Transition Diagram





Verilog Description

```
module traffic_light (HG, HY, HR, FG, FY, FR, ST_o, tl, ts, clk, reset, c);  
output HG, HY, HR, FG, FY, FR, ST_o;  
input tl, ts, clk, reset, c;  
reg ST_o, ST;  
reg [0:1] state, next_state;  
parameter EVEN=0, ODD=1;  
parameter S0=2b'00, S1=2b'01, S2=2b'10, S3=2b'11;  
assign HG = (state==S0);  
assign HY = (state==S1);  
assign HR = (state==S2) || (state==S3);  
assign FG = (state==S2);  
assign FY = (state==S3);  
assign FR = (state==S0) || (state==S1);
```



```
//flip-flops
always @(posedge clk or posedge reset)
    if (reset) // an asynchronous reset
        begin
            state = S0;
            ST_o = 0;
        end
    else
        begin
            state = next_state;
            ST_o = ST;
        end
end
```



always @(state or c or tl or ts)

case (state) // state

transition

S0:

if (tl & c)

begin

next_state = S1;

ST = 1;

end

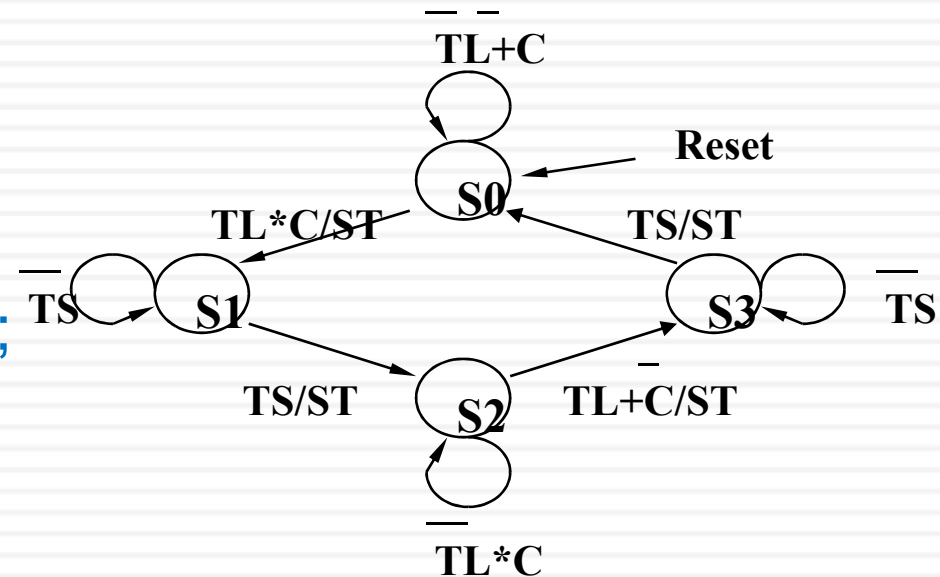
else

begin

next_state = S0;

ST = 0;

end



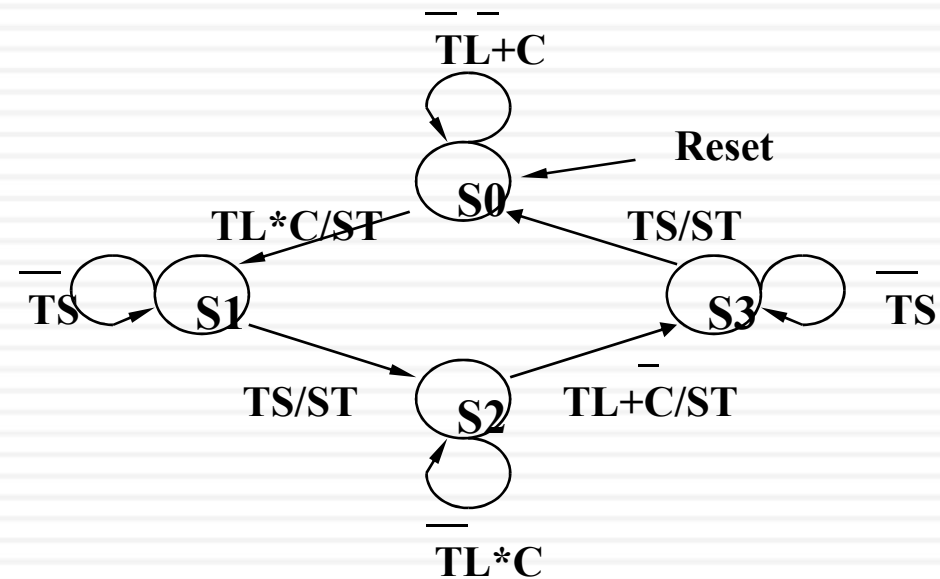


S1:

```
if (ts) begin
    next_state = S2;
    ST = 1;
end
else begin
    next_state = S1;
    ST = 0;
end
```

S2:

```
if (tl | !c) begin
    next_state = S3;
    ST = 1;
end
else begin
    next_state = S2;
    ST = 0;
end
```





S3:

if (ts)

begin

next_state = S0;

ST = 1;

end

else

begin

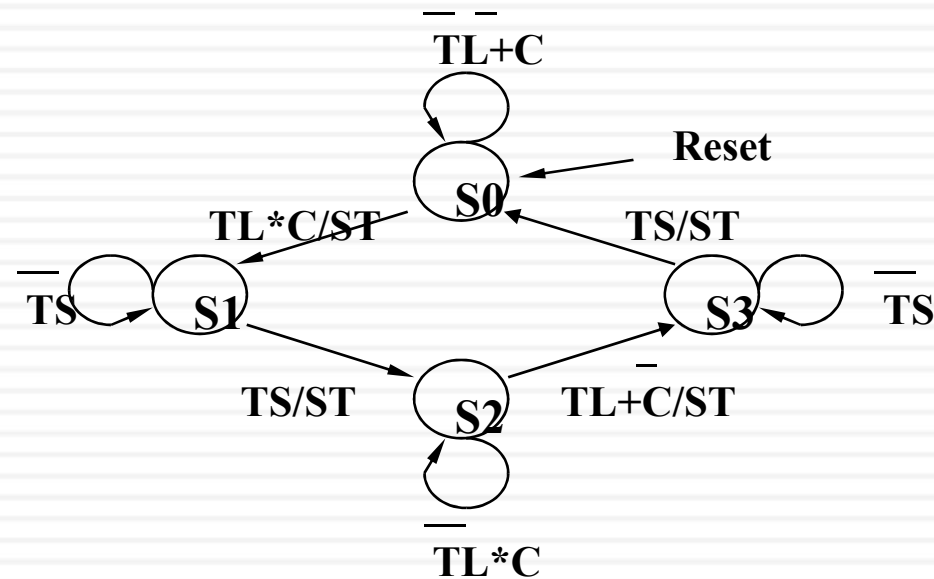
next_state = S3;

ST = 0;

end

endcase

endmodule





FSM I

```
module FSM1(CLK, OP1, OP2);  
  input CLK;  
  output OP1, OP2;  
  reg OP1, OP2;  
  parameter S0=0, S1=1, S2=2;  
  reg [1:0] STATE;  
  always @(posedge CLK)  
  begin  
    case (STATE)  
      S0: begin  
        OP1 = 1;  
        OP2 = 1;  
        STATE=S1;  
      end  
    end
```

```
      S1: begin  
        OP1 = 0;  
        STATE=S2;  
      end  
      S2: begin  
        OP2 = 0;  
        STATE=S0;  
      end  
      default:  
        STATE=S0;  
    endcase  
  end  
endmodule
```



FSM 2

```
module FSM2(CONTROL, CLK,  
    RESET, Y);  
    input control, CLK, RESET;  
    output [0:2] Y;  
    reg [0:2] Y;  
    parameter S0=0, S1=1, S2=2,  
        S3=3;  
    reg [1:0] STATE;  
    always @(posedge CLK)  
    begin  
        if (RESET) begin  
            Y=0;  
            STATE=S0;  
        end  
        else  
            case (STATE)  
                S0: begin  
                    Y = 1;  
                    STATE=S1;  
                end  
            end
```

```
        S1: begin  
            Y = 2;  
            if (CONTROL == 1)  
                STATE=S2;  
            else  
                STATE=S3;  
            end  
        S2: begin  
            Y=3;  
            STATE=S3;  
        end  
        S3: begin  
            Y=4;  
            STATE=S0;  
        default:  
            STATE=S0;  
        endcase  
    end  
endmodule
```



FSM 3

```
module FSM3(CONTROL, CLK, RESET, Y);
    input control, CLK, RESET;
    output [0:2] Y;
    reg [0:2] Y;
    parameter S0=0, S1=1, S2=2, S3=3;
    reg [1:0] STATE;
    always @(posedge RESET or posedge CLK)
    begin
        if (RESET)
            STATE=S0;
        else
            case (STATE)
                S0: STATE=S1;
                S1: if (CONTROL == 1)
                    STATE=S2;
                    else
                        STATE=S3;
                S2: STATE=S3;
                S3: STATE=S0;
                default: STATE=S0;
            endcase
        end
    always @(STATE)
    case (STATE)
        S0: Y=1;
        S1: Y=2;
        S2: Y=3;
        S3: Y=4;
        default: Y=0;
    endcase
    end
endmodule
```




FSM 4

```
module FSM4(CONTROL, CLK, RESET, Y);
  input control, CLK, RESET;
  output [0:2] Y;
  reg [0:2] Y;
  parameter S0=0, S1=1, S2=2, S3=3;
  reg [1:0] PRESENT_STATE, NEXT_STATE;
  always @(PRESENT_STATE or CONTROL)
  begin
    Y=0;
    NEXT_STATE=S0;
    case (PRESENT_STATE)
      S0: begin
          Y=1;
          NEXT_STATE=S1;
        end
      S1: begin
          if (CONTROL == 1)
            NEXT_STATE=S2;
          else
            NEXT_STATE=S3;
        end
    endcase
  end
endmodule
```

```
      S2: begin
          Y=3;
          NEXT_STATE=S3;
        end
      S3: begin
          Y=4;
          STATE=S0;
        endcase
    end
  always @(posedge RESET or
  posedge CLK)
  begin
    if (RESET)
      PRESENT_STATE=S0;
    else
      PRESENT_STATE =
        NEXT_STATE;
    end
  endmodule
```



Z Handling

- To compare with “z” (High Impedance) use only equality operators
- Using “==” z is always evaluated FALSE
- Using “!=” z is always evaluated TRUE

Example: (z handling using “==” operator)

```
if (A == 'bz)
```

```
    B = 0;
```

```
else
```

```
    B = 1;
```

```
// B is always 1 (comparison evaluated FALSE)
```

Example: (z handling using “!=” operator)

```
if (A != 4'hz)
```

```
    B = 4'h0;
```

```
else
```

```
    B = 4'h1;
```

```
// B is always 0 in 4-bit hex (comparison evaluated TRUE)
```



Z Handling (2)

□ Tri-state inference

- Tri-state buffer may be inferred from the high impedance value z.
- When the high impedance value z is assigned to a variable, the output of the tri-state buffer is disabled.

Example: (Tri-state inference using z handling)

```
module tri_state(a, b);  
    parameter N = 15;  
    input [N:0] a;  
    input enable;  
    output [N:0] b;  
  
    assign b = (enable) ? a : 16'bz;  
endmodule;
```



X Handling

X (unknown) handling

- Similar to value “z”
- Using “==” x is always evaluated FALSE
- Using “!=” x is always evaluated TRUE

Example: (x handling using “==” operator)

```
if (A == 'bx')
    B = 0;
else
    B = 1;
// B is always 1 (comparison evaluated FALSE)
```

Example: (x handling using “!=” operator)

```
if (A != 4'hx)
    B = 4'h0;
else
    B = 4'h1;
// B is always 0 in 4-bit hex (comparison evaluated TRUE)
```



Wired-AND

- Resolves the shorting of an output wire receiving multiple assignments to an AND gate
 - Declaring “wand”
- Example

```
module mywand (a, b, c);  
    input a, b;  
    output c;  
  
    wand c;  
  
    assign c = a;  
    assign c = b;  
endmodule;
```



Wired-OR

- Resolves the shorting of an output wire receiving multiple assignments to an OR gate
 - Declaring “wor”
- Example

```
module mywor (a, b, c);  
    input a, b;  
    output c;  
  
    wor c;  
  
    assign c = a;  
    assign c = b;  
endmodule;
```



Tri-State

- Shorting of an output wire receiving multiple assignments
 - Declaring “tri”

- Example

```
module mytri (a, b, c);  
    input a, b, control;  
    output c;
```

```
    tri c;
```

```
    assign c = (control ? a : 'bz);  
    assign c = (control ? 'bz : b);
```

```
endmodule;
```



Resolution Functions

□ Example

```
module T (A, B, C, D, E, Z);  
    input A, B, C, D, E;  
    output Z;  
  
    // wor BAT;  
    // wand BAT;  
    // tri BAT;  
    wire BAT;  
  
    assign BAT = A & B;  
    assign BAT = C | D;  
    assign Z = BAT | E;  
endmodule;
```




Resolution Functions (2)

- Case 1: If BAT is a wire
 - Warning: Unable to determine wired-logic type for multiple-driver net 'BAT'
 - Information: Assuming multiple-driver net 'BAT' is a wired-AND
 - Connects the drivers of BAT using an AND gate
- Case 2: If BAT is a tri
 - Warning: In design 'T', there is 1 three-state bus with non-three state drivers.
 - Connects the drivers of BAT using an AND gate
- Case 3: If BAT is a wand
 - Information: In design 'T', there is 1 wired-AND net
 - Connects the drivers of BAT using an AND gate
- Case 4: If BAT is a word
 - Information: In design 'T', there is 1 wired-OR net
 - Connects the drivers of BAT using an OR gate



Resolution Functions (3)

□ Example

```
module T (A, B, C, D, E, F, G, Z);  
    input A, B, C, D, E, F, G;  
    output Z;  
  
    // wor BAT;  
    // wand BAT;  
    tri BAT;  
    // wire BAT;  
  
    assign BAT = (F ? A&B : 'bz);  
    assign BAT = (G ? C | D : 'bz);  
    assign Z = BAT | E;  
endmodule;
```



Resolution Functions (4)

- Case 5: If BAT is a tri
 - BAT is shorted after tri-state gates
- Case 6: If BAT is a wire/wand/wor
 - BAT is shorted after tri-state gates



Resolution Functions (5)

□ Example

```
module T (A, B, C, D, E, F, G, Z);  
    input A, B, C, D, E, F, G;  
    output Z;  
    // wor BAT;  
    wand BAT;  
    // tri BAT;  
    // wire BAT;  
    assign BAT = (F ? A&B : 'bz);  
    assign BAT = C | D;  
    assign Z = BAT | E;  
endmodule;
```

□ Case 7: BAT is a wand/wor/tri/wire

- Warning: In design 'T', there is 1 three-state but with non three-state drivers
- BAT drivers are shorted



Resolution Functions (6)

□ Example

```
module T (A, B, C, D, E, F, G, H, J, Z);  
    input A, B, C, D, E, F, G, H, J;  
    output Z;  
    wor BAT;  
    // wand BAT;  
    // tri BAT;  
    // wire BAT;  
  
    assign BAT = (F ? A&B : 'bz);  
    assign BAT = C | D;  
    assign BAT = H ^ J;  
    assign Z = BAT | E;  
endmodule;
```

□ Case 8: BAT is a wand/wor/tri/wire

- 2nd and 3rd drivers are AND'ed. Could be a bug

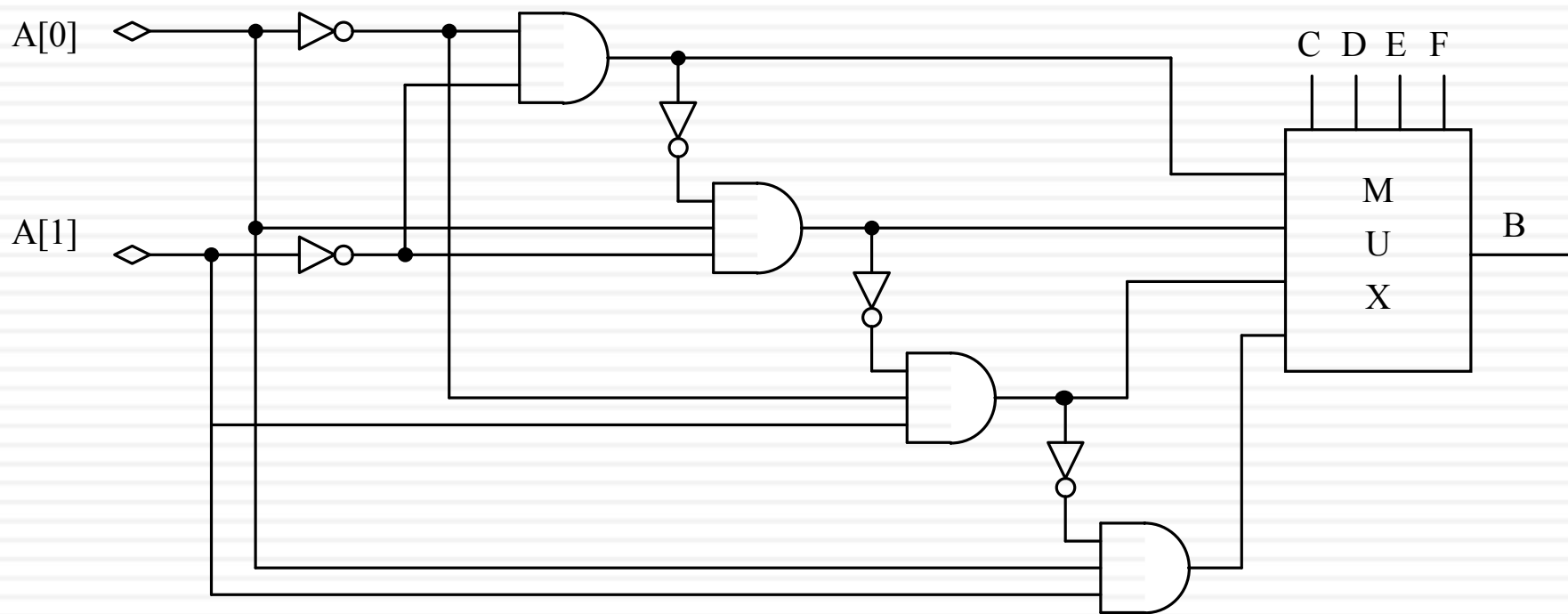


If-Else Statement

```
if (A[0] == 0 && A[1] == 0)
    B = C;
else if (A[0] == 1 && A[1] == 0)
    B = D;
else if (A[0] == 0 && A[1] == 1)
    B = E;
else
    B = F;
```



If-Else Hardware Implementation



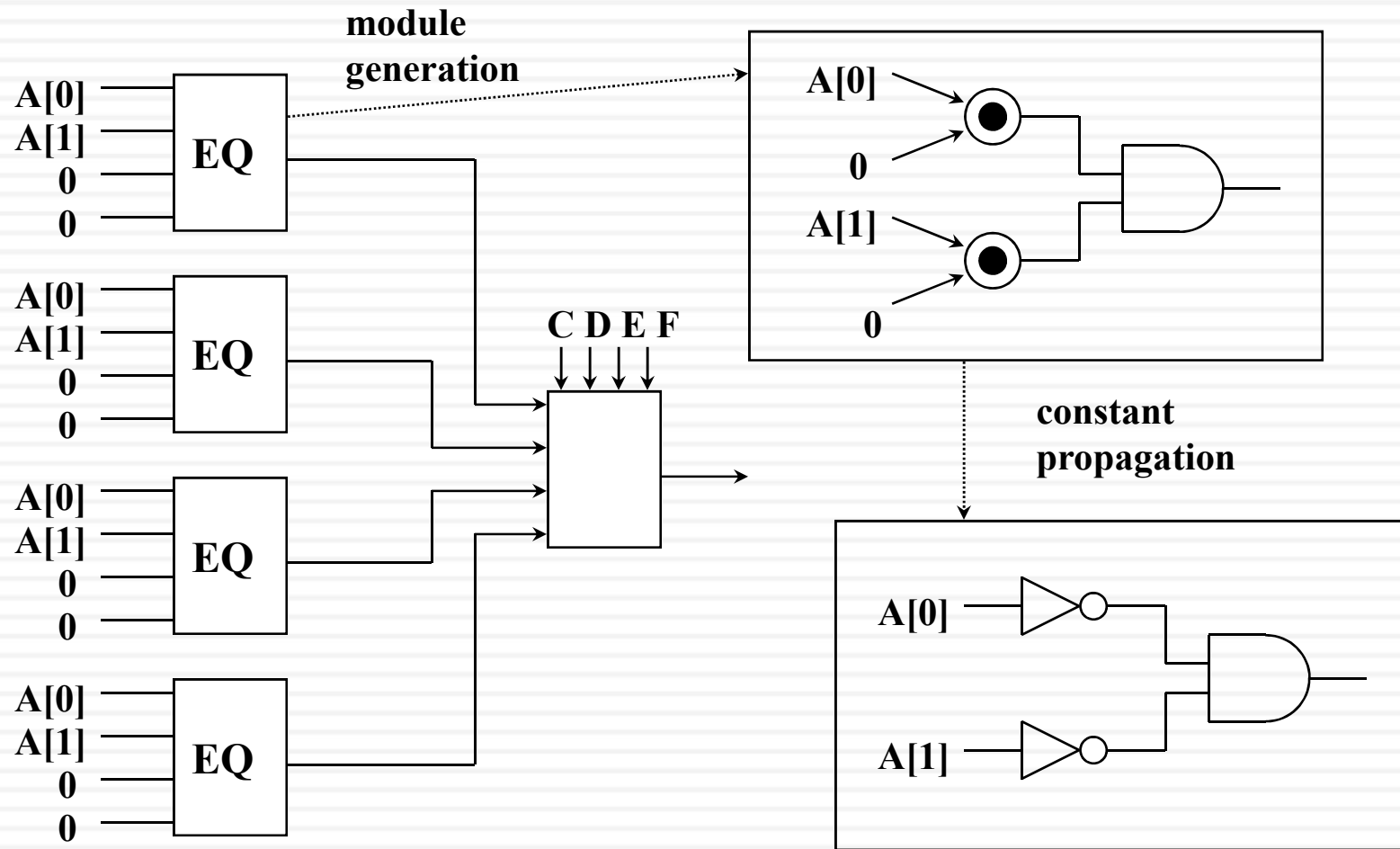


Case Statement

```
case (A)
  2'b00: B = C;
  2'b01: B = D;
  2'b10: B = E;
  2'b11: B = F;
endcase
```




Case Hardware Implementation





Modeling for Optimization

- Sharing common subexpressions
 - Data path
 - Common block of logic
 - Sharing between synchronous and combinational sections
- Adding Structure
 - Sharing resource explicitly
 - Using parentheses
 - Detailing the logic structure
- Using design knowledge
 - Bit-width calculation
 - Constant propagation
- Understanding hardware implication
 - Hardware implication over software efficiency



Common Subexpression

- Don't repeatedly calculate the same operation
 - Use temporary assignment
- Original

$$A = B + C + D;$$
$$D = F + C + B;$$

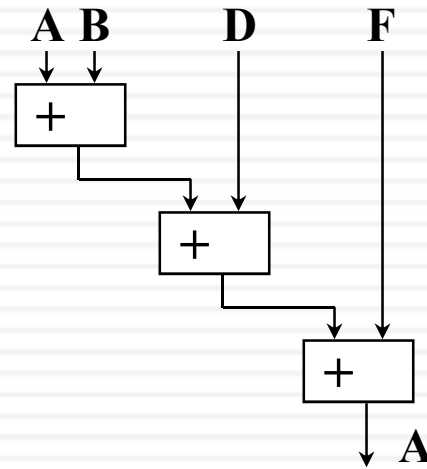
- Modified

$$E = B + C;$$
$$A = E + D;$$
$$D = F + E;$$

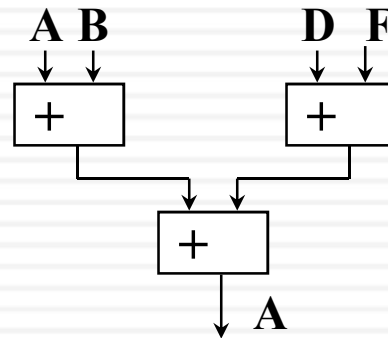


Using Parenthesis

$$A = B + C + D + F;$$



$$A = (B + C) + (D + F);$$





Constant Propagation

□ Original style

```
if (RST == 'b0) begin
    A = 11 - 2;
    B = A + 1
    D = 12 - B;
    C = 2 * D;
end
else
    C = B;
```

□ Better style

```
if (RST == 'b0)
    C = 4;
else
    C = 10;
```



Control of Logic Network

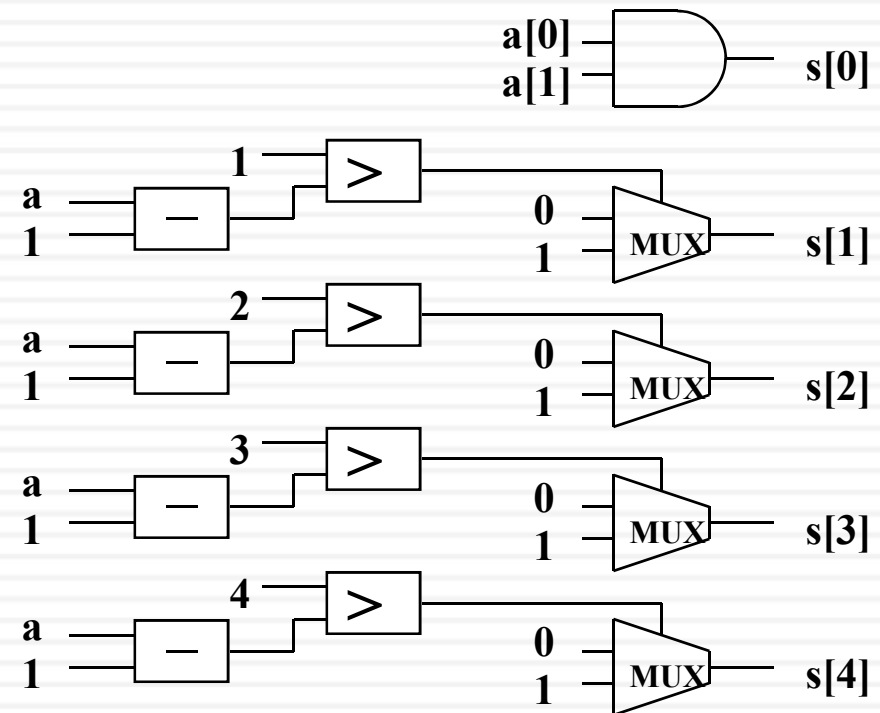
- The most straightforward coding is not always the best choice for actual hardware implementation

/ Verilog Version */*

```

module worst_ver (a, s);
  input [4:0] a;
  output [4:0] s;
  reg [4:0] s; integer i;
  always @ (a) begin
    if (a == 0) s = 5'b11110;
    else begin
      s[0] = a[0] && a[1];
      for (i=1; i <= 4; i=i+1)
        if (i > a-1'b1)
          s[i] = 1'b1;
        else
          s[i] = 1'b0;
    end
  end
end
endmodule

```



Generic logic schematic



Control of Logic Network

- Adding structure information by introduction of temporary variables

```
/* Verilog Version */
```

```
module worse_ver (a, s);
```

```
  input [4:0] a;
```

```
  output [4:0] s;
```

```
  reg [4:0] s, tmp;
```

```
  integer 1;
```

```
  always @ (a) begin
```

```
    if (a == 0) s = 5'b11110;
```

```
    else begin
```

```
      s[0] = a[0] && a[1];
```

```
      tmp = a-1;
```

```
      for (l=1; l <= 4; l=l+1)
```

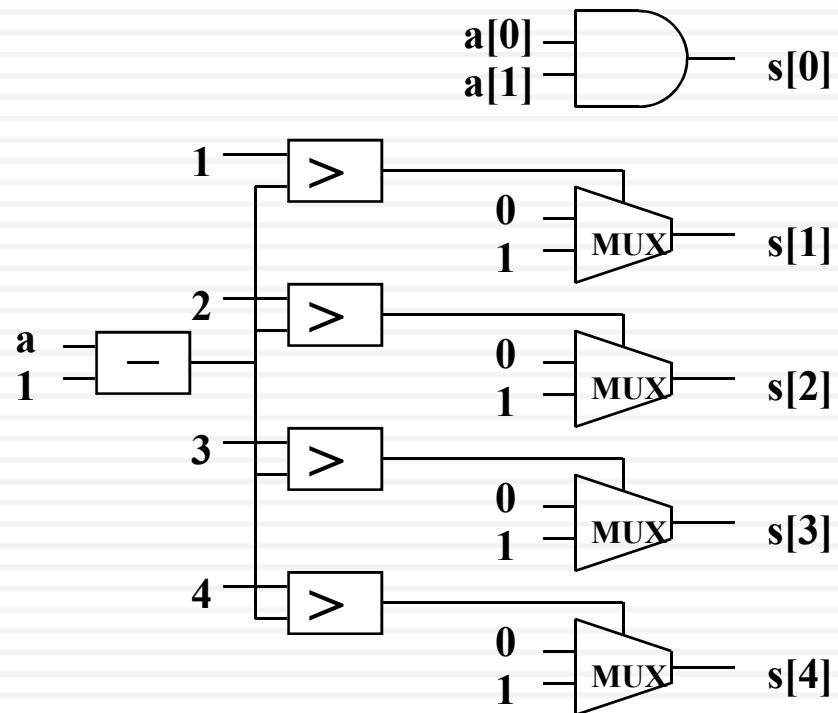
```
        if (l > tmp) s[l] = 1'b1;
```

```
        else s[l] = 1'b0;
```

```
    end
```

```
  end
```

```
endmodule
```





Control of Logic Network

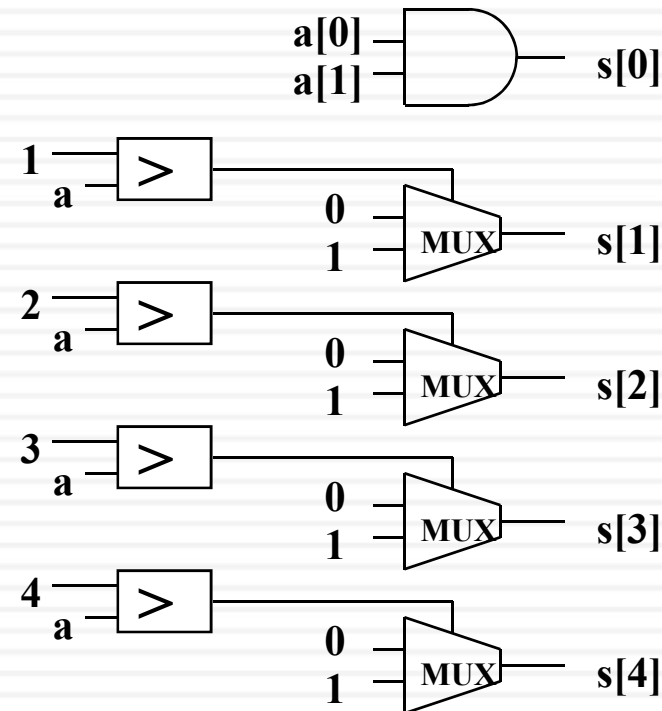
□ Move constants to one side

/ Verilog Version */*

```

module good_ver (a, s);
  input [4:0] a;
  output [4:0] s;
  reg [4:0] s;
  integer 1;
  always @ (a) begin
    if (a == 0) s = 5'b11110;
    else begin
      s[0] = a[0] && a[1];
      for (l=1; l <= 4; l=l+1)
        if (l+1 > a)
          s[l] = 1'b1;
        else
          s[l] = 1'b0;
    end
  end
end
endmodule

```





Control of Logic Network

- Think more as a logic specification than a function specification

```
/* Verilog Version */
```

```
module better_ver (a, s);
```

```
  input [4:0] a;
```

```
  output [4:0] s;
```

```
  reg [4:0] s;
```

```
  integer 1;
```

```
  always @ (a) begin
```

```
    case (a)
```

```
      5'b00000: s = 5'b11110;
```

```
      5'b00001: s = 5'b11110;
```

```
      5'b00010: s = 5'b11100;
```

```
      5'b00011: s = 5'b11001;
```

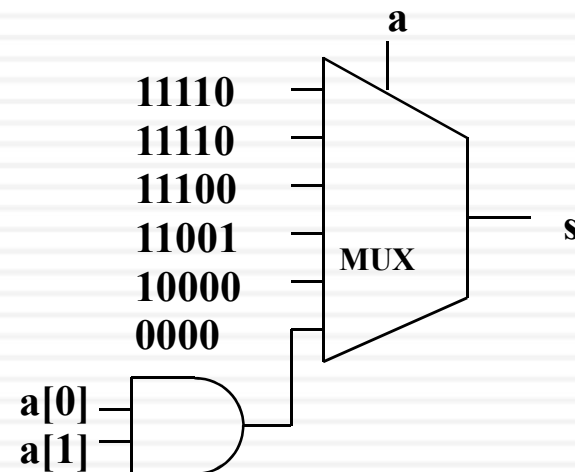
```
      5'b00100: s = 5'b10000;
```

```
      default: s={4'b0000,a[0]&&a[1]};
```

```
    endcase
```

```
  end
```

```
endmodule
```



Generic logic schematic



Hardware Implication

- Smart comparison
 - if (CTRL >= 3'b100)
 - if (CTRL[2] == 1'b1)

- Multiplication vs. shifting
 - A * 4
 - A << 2

- Unnecessary operation
 - if ((cnt+1) == 120)
 - if (cnt == 119)



Complex Operation

- Multiplication, division and remainder by variables or constants (not power of 2)
- Addition and subtraction by variables
- Shifting by non-computable amount
- Non-computable array indexing
- Comparison with variables
- Preferred actions:
 - Avoiding
 - Simplifying
 - Sharing