Digital System Design

With Verilog HDL

Dr. Gaurav Trivedi,

Department of EEE, IIT Guwahati, Assam, Pin 781039

Email: trivedi@iitg.ac.in

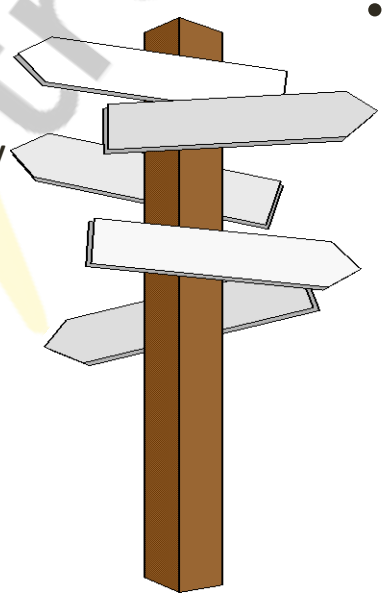
Indian Institute of Technology Guwahati

Content of Lecture

- Introduction
- History of Verilog® HDL
- Overview of Digital Design with Verilog® HDL
- Hierarchical Modeling Concepts
- Verilog Basics
- Modules Ports
- Verilog-Data Flow Modeling
- Verilog Behavioral Modeling
- Finite State Machines (FSM)
- Algorithmic State Machines (ASM)Chart
- Verilog Tasks and Functions

Major Design Challenges

- Microscopic issues
  - ultra-high speeds
  - power dissipation and supply rail drop
  - growing importance of interconnect
  - noise, crosstalk
  - reliability, manufacturability
  - clock distribution



- Macroscopic issues
  - time-to-market
  - design complexity (millions of gates)
  - high levels of abstractions
  - reuse and IP, portability
  - systems on a chip (SoC)
  - tool interoperability

Year	Tech.	Complexity	Frequency	3 Yr. Design Staff Size	Staff Costs
1997	0.35	13 M Tr.	400 MHz	210	$90 M
1998	0.25	20 M Tr.	500 MHz	270	$120 M
1999	0.18	32 M Tr.	600 MHz	360	$160 M
2002	0.13	130 M Tr.	800 MHz	800	$360 M

Technology Directions: SIA Roadmap

Year	1999	2002	2005	2008	2011	2014
Feature size (nm)	180	130	100	70	50	35
Mtrans/cm^2	7	14-26	47	115	284	701
Chip size (mm^2)	170	170-214	235	269	308	354
Signal pins/chip	768	1024	1024	1280	1408	1472
Clock rate (MHz)	600	800	1100	1400	1800	2200
Wiring levels	6-7	7-8	8-9	9	9-10	10
Power supply (V)	1.8	1.5	1.2	0.9	0.6	0.6
High-perf power (W)	90	130	160	170	174	183
Battery power (W)	1.4	2.0	2.4	2.0	2.2	2.4

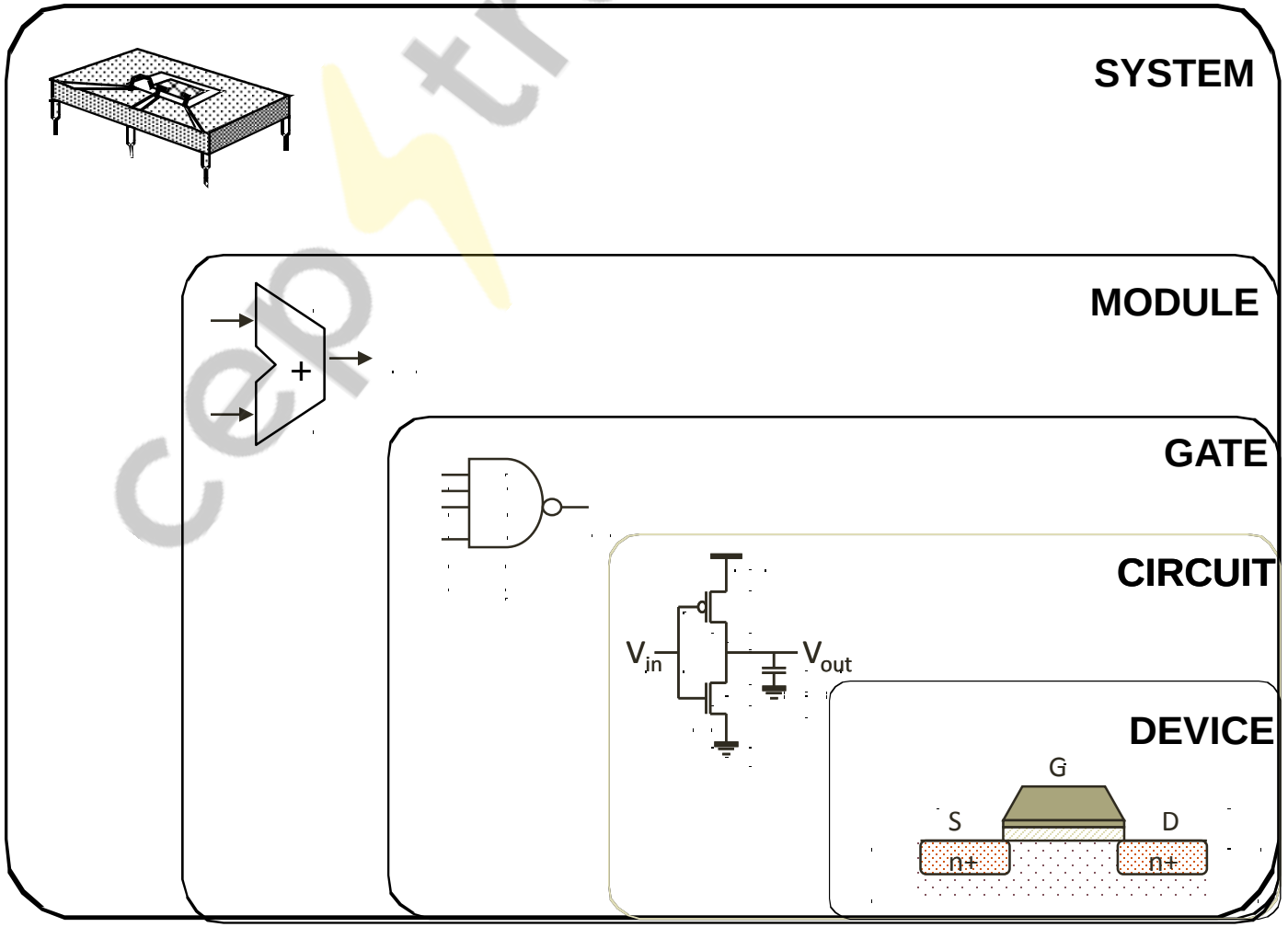
For Cost-Performance MPU (L1 on-chip SRAM cache; 32KB/1999 doubling every two years)

http://www.itrs.net/ntrs/publntrs.nsf

Why Scaling?

- Technology shrinks by ~0.7 per generation
- With every generation can integrate 2x more functions on a chip; chip cost does not increase significantly
- Cost of a function decreases by 2x
- But …
  - How to design chips with more and more functions?
  - Design engineering population does not double every two years…
- Hence, a need for more efficient design methods
  - Exploit different levels of abstraction

Design Abstraction Levels



Design Metrics

- How to evaluate performance of a digital circuit (gate, block, …)?
  - Cost
  - Reliability
  - Scalability
  - Speed (delay, operating frequency)
  - Power dissipation
  - Energy to perform a function

History of Verilog® HDL

- Beginning: 1983
  - "Gateway Design Automation" company
  - Simulation environment
    - Comprising various levels of abstraction
      - Switch (transistors), gate, register-transfer, and higher levels
- Three factors to success of Verilog
  - Programming Language Interface (PLI)
    - Extend and customize simulation environment
  - Close attention to the needs of ASIC foundries
    - "Gateway Design Automation" partnership with Motorola, National, and UTMC in 1987-89
  - Verilog-based synthesis technology
    - "Gateway Design Automation" licensed Verilog to Synopsys
    - Synopsys introduced synthesis from Verilog in 1987

History of Verilog® HDL

- VHDL
  - VHSIC (Very High Speed Integrated Circuit) Hardware Description Language
  - Developed under contract from DARPA
  - IEEE standard
  - Public domain
  - Other EDA vendors adapted VHDL
  - “Gateway” put Verilog in public domain
  - Today
    - Market divided between Verilog & VHDL
      - VHDL mostly in Europe
      - Verilog dominant in US
    - VHDL
      - More general language
      - Not all constructs are synthesizable
    - Verilog:
      - Not as general as VHDL
      - Most constructs are synthesizable

Verilog® HDL

Overview of Digital Design with Verilog® HDL

Overview of Digital Design Using Verilog

- Evolution of Computer-Aided Digital Design
- Emergence of HDLs
- Typical Design Flow
- Importance of HDLs
- Popularity of Verilog HDL
- Trends in HDLs

Evolution of Computer-Aided Digital Design

- SSI: Small scale integration
  - A few gates on a chip
- MSI: Medium scale integration
  - Hundreds of gates on a chip
- LSI: Large scale integration
  - Thousands of gates on a chip
  - CAD: Computer-Aided Design
    - CAD vs. CAE
  - Logic and circuit simulators
  - Prototyping on bread board
  - Layout by hand (on paper or a computer terminal)

Evolution of Computer-Aided Digital Design (cont'd)

- VLSI: Very Large Scale Integration
  - Hundred thousands of gates
  - Not feasible anymore:
    - Bread boarding
    - Manual layout design
  - Simulator programs
  - Automatic place-and-route
  - Bottom-Up design
    - Design small building blocks
    - Combine them to develop bigger ones
  - More and more emphasis on logic simulation

Emergence of HDLs

- The need to a standardized language for hardware description
  - Verilog® and VHDL
- Simulators emerged
  - Usage: functional verification
  - Path to implementation: manual translation into gates
- Logic synthesis technology
  - Late 1980s
  - Dramatic change in digital design
    - Design at Register-Transfer Level (RTL) using an HDL

Typical Design Flow (in 1996)

1. Design specification
2. Behavioral description
3. RTL description
4. Functional verification and testing
5. Logic synthesis
6. Gate-level netlist
7. Logical verification and testing
8. Floor planning, automatic place & route
9. Physical layout
10. Layout verification
11. Implementation

Typical Design Flow (cont'd)

- Most design activity
  - In 1996:
    - Manually optimizing the RTL design
    - CAD tools take care of generating lower-level details
    - Reducing design time to months from years
  - Today
    - Still RTL is used in many cases
    - But, synthesis from behavioral-level also possible
    - Digital design now resembles high-level computer programming

Typical Design Flow (cont'd)

- NOTE:
  - CAD tools help, but the designer still has the main role
    - GIGO (Garbage-In Garbage-Out) concept
    - To obtain an optimized design, the designer needs to know about the synthesis technology
      - Compare to software programming and compilation

Importance of HDLs

- Retargeting to a new fabrication technology
- Functional verification earlier in the design cycle
- Textual concise representation of the design
  - Similar to computer programs
  - Easier to understand

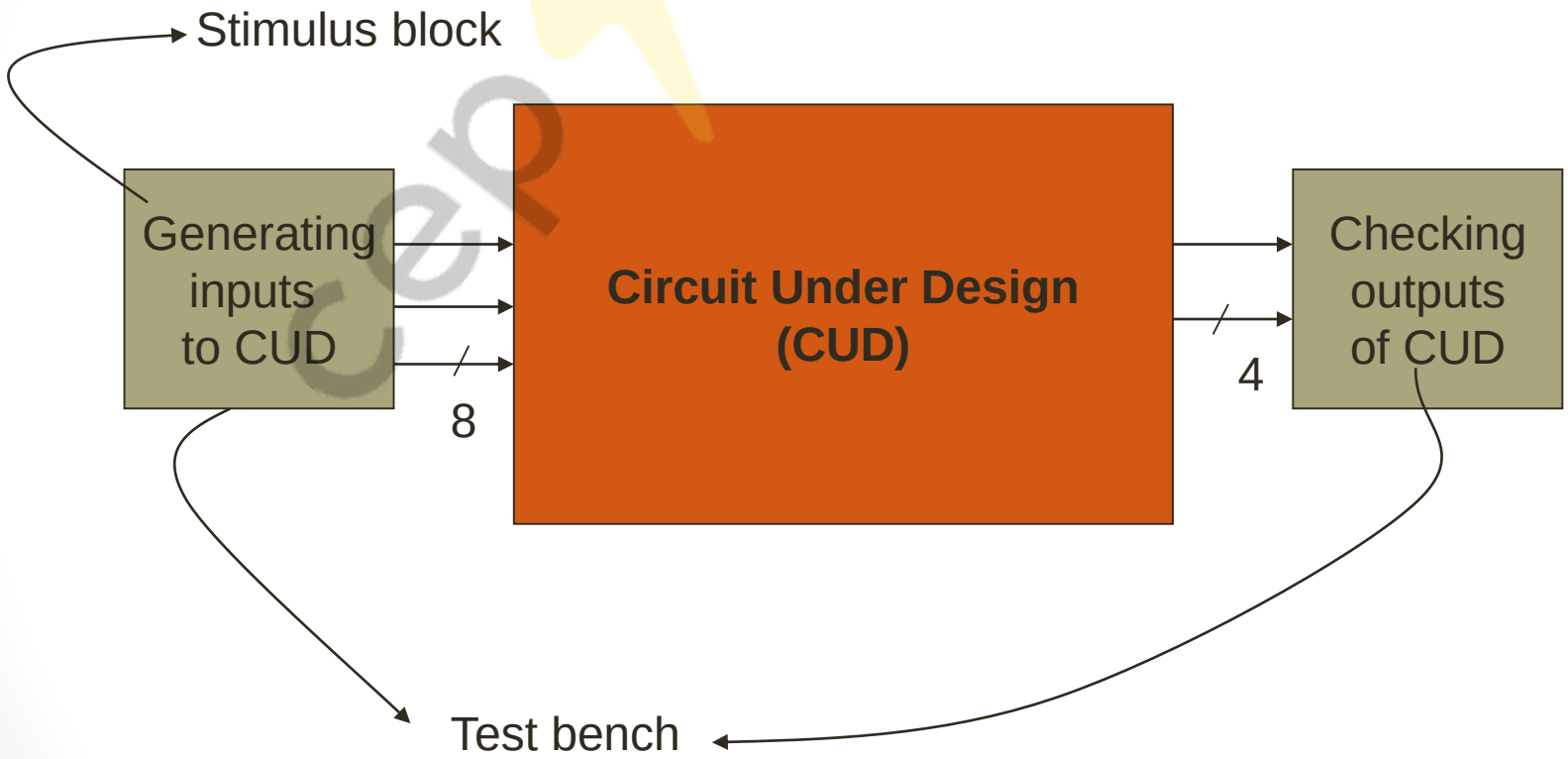
Popularity of Verilog HDL

- Verilog HDL
  - General-purpose
  - Easy to learn, easy to use
  - Similar in syntax to C
  - Allows different levels of abstraction and mixing them
  - Supported by most popular logic synthesis tools
  - Post-logic-synthesis simulation libraries by all fabrication vendors
  - PLI to customize Verilog simulators to designers' needs

Trends in HDLs

- Design at behavioral level
- Formal verification techniques
- Very high speed and time critical circuits
  - e.g. microprocessors
  - Mixed gate-level and RTL designs
- Hardware-Software Co-design
  - System-level languages: SystemC, SpecC, …

Basics of Digital Design Using HDLs



Verilog® HDL

Hierarchical Modeling Concepts

Verilog Basic Building Block

- Module

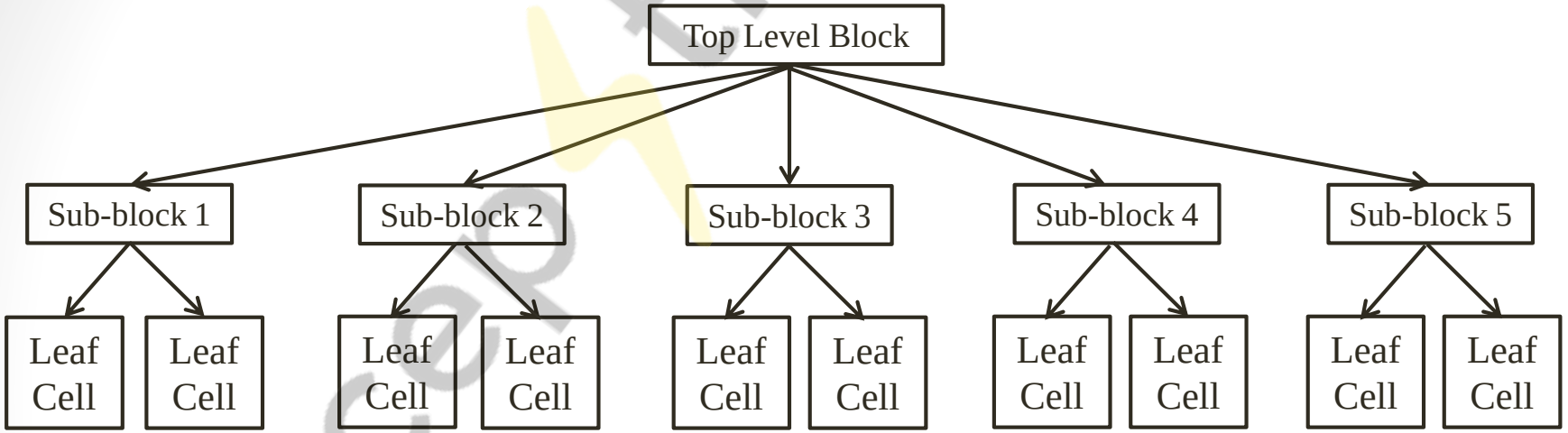
```
module not_gate(in, out); // module
  name+ports
     // comments: declaring port type
    input in;
    output out;

     // Defining circuit functionality
    assign out = ~in;

endmodule
```

useless Verilog Example

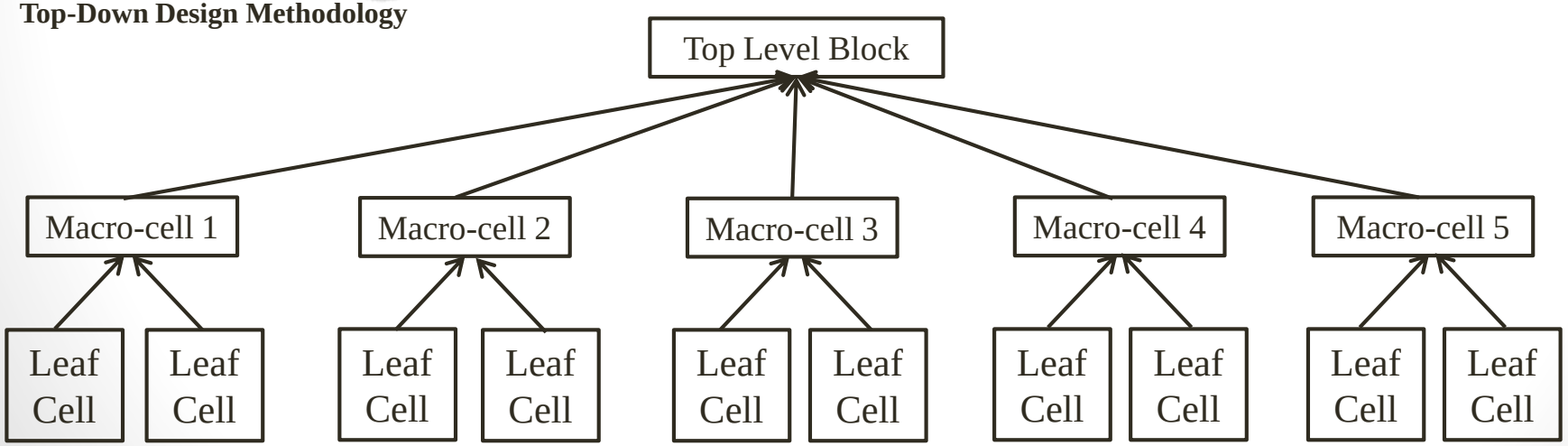
```
module useless;

    initial
        $display("Hello World!");

endmodule
```

- Note the message-display statement
  - Compare to printf() in C

Design Methodologies

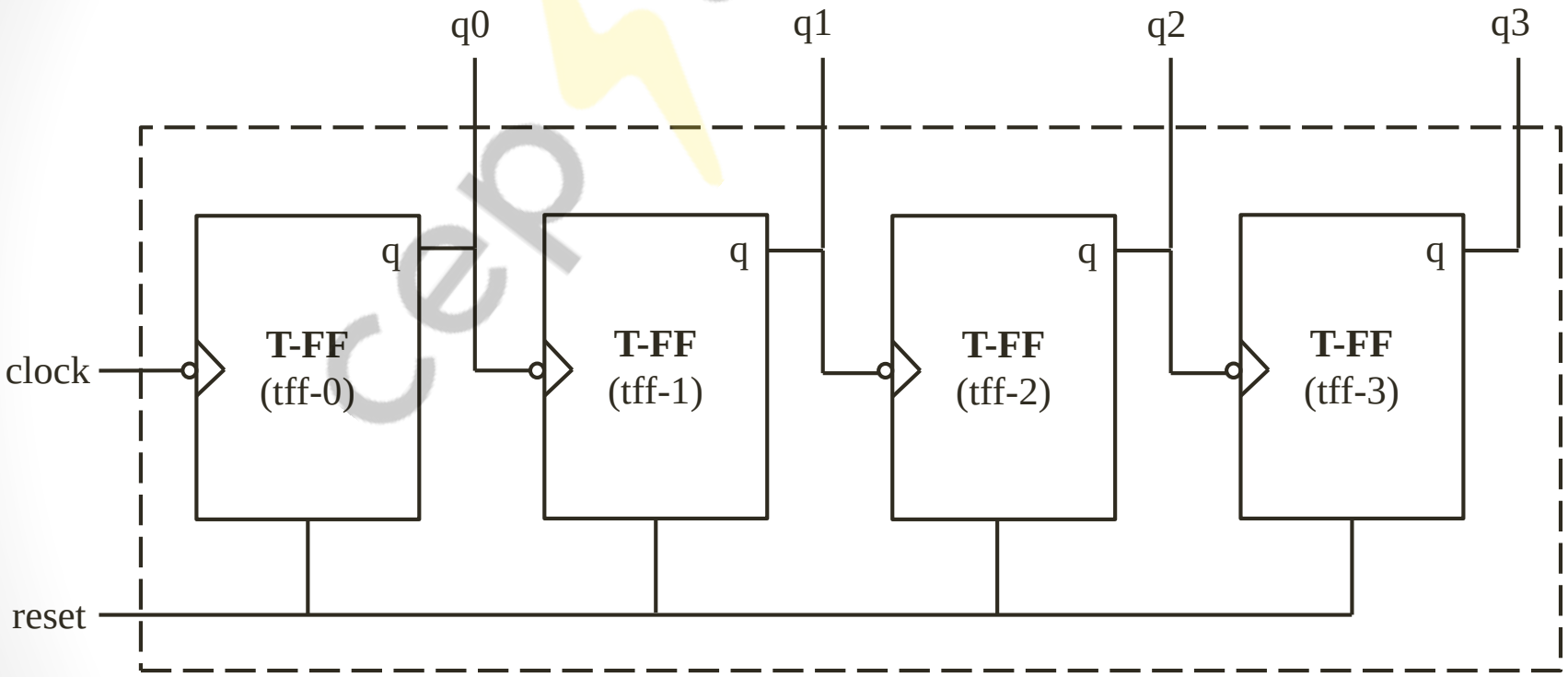


Top-Down Design Methodology



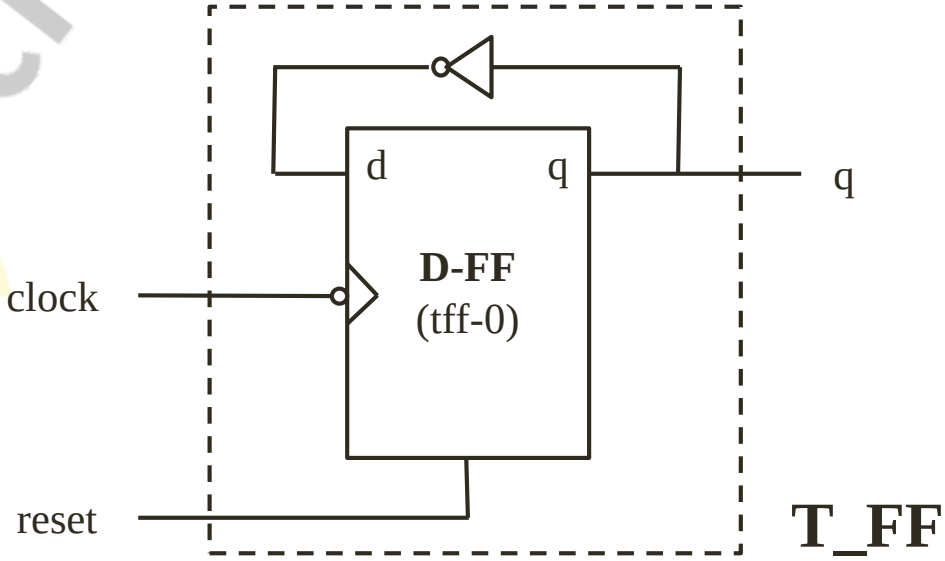
Bottom Up Design Methodology

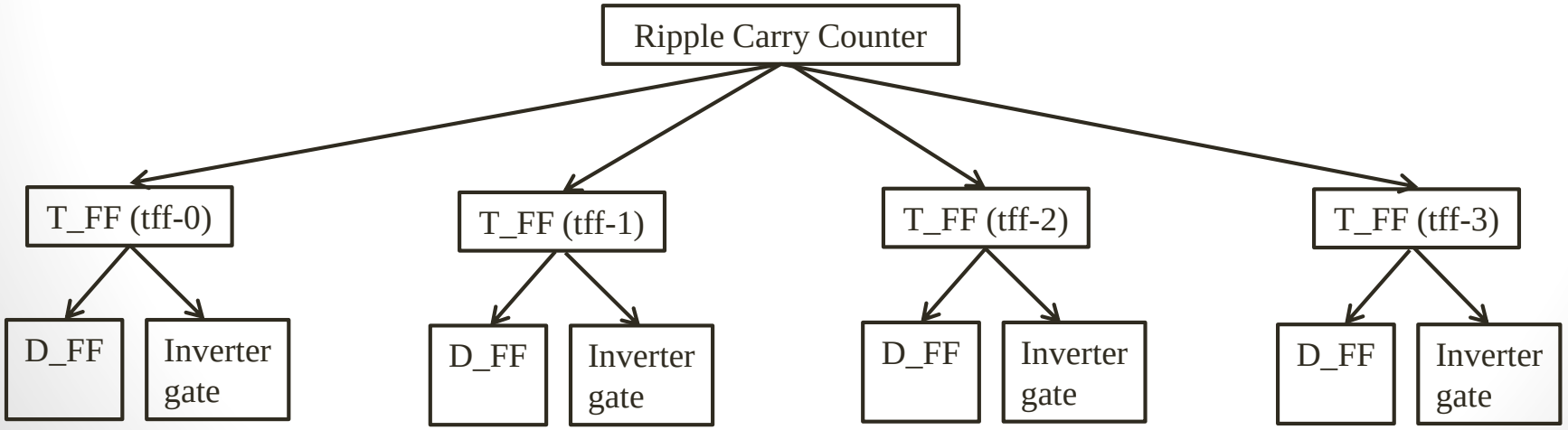
4-bit Ripple Carry Counter



T-flipflop and the Hierarchy

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





Modules

```
module
  <module_name>(<module_terminal_list>);
      ...
      <module internals>
      ...
endmodule
```

- Example:

```
module T_ff(q, clock, reset);
      ...
      <functionality of T_flipflop>
      ...
endmodule
```

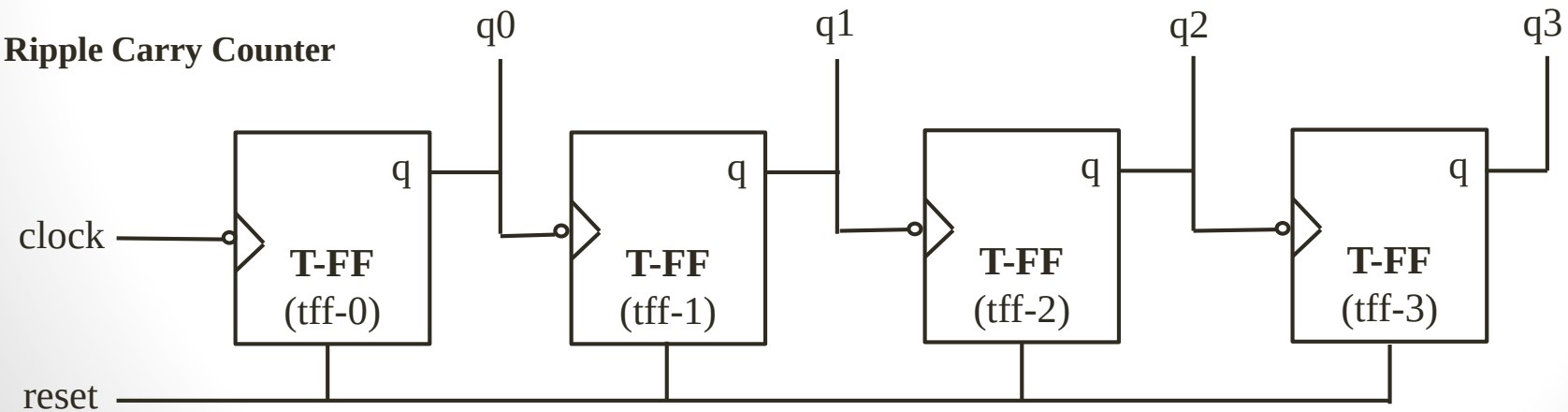
Modules (cont'd)

- Verilog supported levels of abstraction
  - Behavioral (algorithmic) level
    - Describe the algorithm used
    - Very similar to C programming
  - Dataflow level
    - Describe how data flows between registers and is processed
  - Gate level
    - Interconnect logic gates
  - Switch level
    - Interconnect transistors (MOS transistors)
- Register-Transfer Level (RTL)
  - Generally known as a combination of behavioral+dataflow that is synthesizable by EDA tools

Instances

```
module ripple_carry_counter(q, clk, reset);

 output [3:0] q;
 input clk, reset;

 TFF tff0(q[0],clk, reset);   //   Four Instances
 TFF tff1(q[1],q[0], reset);  //   of T Flip Flop
 TFF tff2(q[2],q[1], reset);  //   are instantiated.
 TFF tff3(q[3],q[2], reset);
endmodule
```

Ripple Carry Counter



Instances (cont'd)

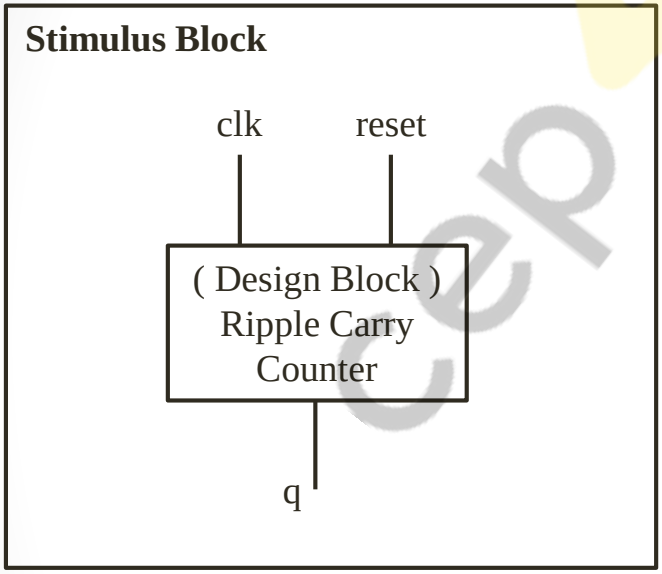
```
module TFF(q, clk, reset);
 output q;
 input clk, reset;
 wire d;

 DFF dff0(q, d, clk, reset);
 not n1(d, q);                    // not is a Verilog provided primitive.

endmodule

                                  // module DFF with asynchronous reset
module DFF(q, d, clk, reset);
 output q;
 input d, clk, reset;
 reg q;
 always @(posedge reset or negedge clk)
   if (reset)
            q = 1'b0;
   else
            q = d;
endmodule
```

Instances (cont'd)

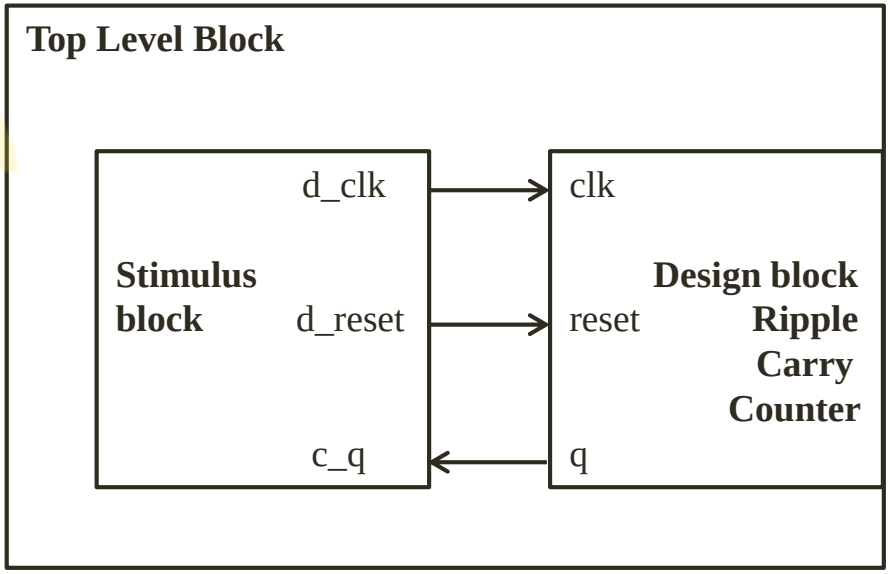
- Illegal instantiation example:
  - *Nested* module definition not allowed
  - Note the difference between *module definition* and *module instantiation*

```
// Define the top level module called ripple carry
// counter. It is illegal to define the module T_FF inside
// this module.
module ripple_carry_counter(q, clk, reset);
 output [3:0] q;
 input clk, reset;

  module T_FF(q, clock, reset);// ILLEGAL MODULE NESTING
  :
  <module T_FF internals>
  :
  endmodule // END OF ILLEGAL MODULE NESTING

endmodule
```

Simulation- Test Bench Styles



Stimulus Block Instantiates Design Block



Stimulus and Design Blocks Instantiated in
a Dummy Top-Level Module

Example

- Design block was shown before
  - ripple_carry_counter, T_FF, and D_FF modules
- Stimulus block

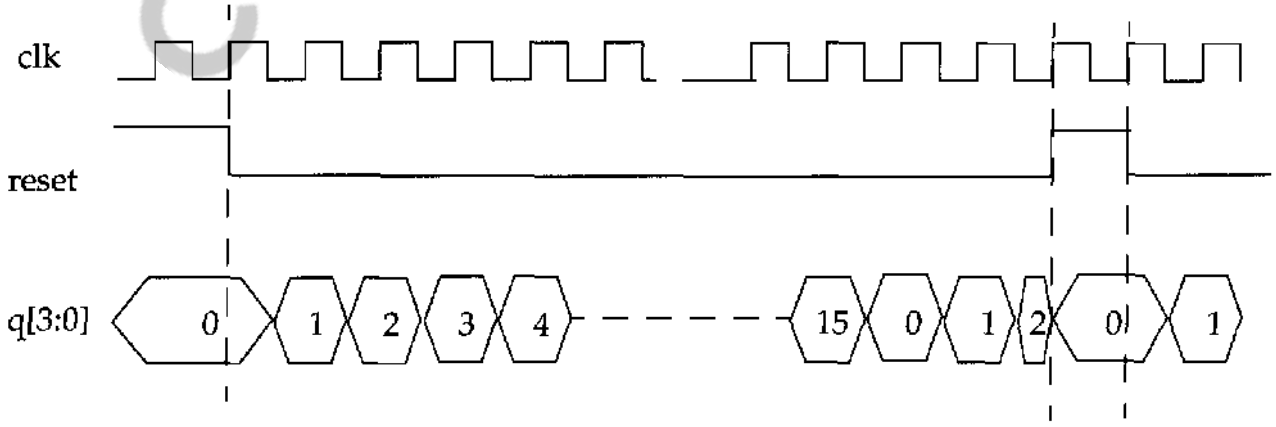


Figure 2-8 Stimulus and Output Waveforms

Example (cont'd)

```
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.
  initial clk = 1'b0;
  always #5 clk = ~clk;

  // 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 $stop;
   end

  initial // Monitor the outputs
    $monitor($time, " Output q = %d",  q);
endmodule
```

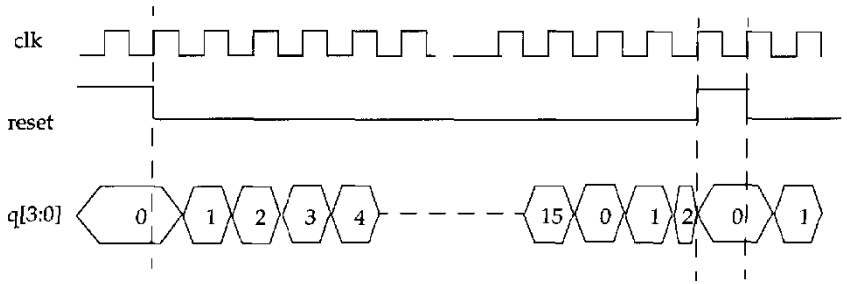


Figure 2-8 Stimulus and Output Waveforms

Verilog® HDL

Verilog Basics

Lexical Conventions

- Very similar to C
  - Verilog is case-sensitive
  - All keywords are in lowercase
  - A Verilog program is a string of tokens
    - Whitespace
    - Comments
    - Numbers
    - Strings
    - Identifiers
    - Keywords

Lexical Conventions (cont'd)

- Whitespace
  - Blank space (\b)
  - Tab (\t)
  - Newline (\n)
- Whitespace is ignored in Verilog except
  - In strings
  - When separating tokens

- Comments
  - Used for readability and documentation
  - Just like C:
    - // single line comment
    - /* multi-line comment */

```
/* Nested comments
/* like this */ may not
be acceptable (depends
on Verilog compiler) */
```

Lexical Conventions (cont'd)

- Operators
  - Unary

```
a = ~b;
```

  - Binary

```
a = b && c;
```

  - Ternary

```
a = b ? c : d; // the only ternary operator
```

Lexical Conventions (cont'd)

- Number Specification
  - Sized numbers
  - Unsized numbers
  - Unknown and high-impedance values
  - Negative numbers

Lexical Conventions (cont'd)

- Sized numbers
  - General syntax: `<size>'<base><number>`
    - `<size>` number of bits (in decimal)
    - `<number>` is the number in radix `<base>`
    - `<base>` :
      - d or D for decimal (radix 10)
      - b or B for binary (radix 2)
      - o or O for octal (radix 8)
      - h or H for hexadecimal (radix 16)
    - Examples:
      - `4'b1111`
      - `12'habc`
      - `16'd255`
- Unsized numbers
  - Default base is decimal
  - Default size is at least 32 (depends on Verilog compiler)
  - Examples
    - `23232`
    - `'habc`
    - `'o234`

Lexical Conventions (cont'd)

- X or Z values
  - Unknown value: lowercase `x`
    - 4 bits in hex, 3 bits in octal, 1 bit in binary
  - High-impedance value: lowercase `z`
    - 4 bits in hex, 3 bits in octal, 1 bit in binary
  - Examples
    - `12'h13x`
    - `6'hx`
    - `32'bz`
  - Extending the most-significant part
    - Applied when `<size>` is bigger than the specified value
      - Filled with x if the specified MSB is x
      - Filled with z if the specified MSB is z
      - Zero-extended otherwise
    - Examples:
      - `6'hx`

Lexical Conventions (cont'd)

- Negative numbers
  - Put the sign before the `<size>`
  - Examples:
    - `-6'd3`
    - `4'd-2  // illegal`
  - Two's complement is used to store the value
- Underscore character and question marks
  - Use '`_`' to improve readability
    - `12'b1111_0000_1010`
    - Not allowed as the first character
  - '`?`' is the same as '`z`' (only regarding numbers)
    - `4'b10?? // the same as 4'b10zz`

Lexical Conventions (cont'd)

- Number Specification
  - Sized numbers
  - Unsized numbers
  - Unknown and high-impedance values
  - Negative numbers

Lexical Conventions (cont'd)

- Sized numbers
  - General syntax: `<size>'<base><number>`
    - `<size>` number of bits (in decimal)
    - `<number>` is the number in radix `<base>`
    - `<base>` :
      - d or D for decimal (radix 10)
      - b or B for binary (radix 2)
      - o or O for octal (radix 8)
      - h or H for hexadecimal (radix 16)
    - Examples:
      - `4'b1111`
      - `12'habc`
      - `16'd255`
- Unsized numbers
  - Default base is decimal
  - Default size is at least 32 (depends on Verilog compiler)
  - Examples
    - `23232`
    - `'habc`
    - `'o234`

Lexical Conventions (cont'd)

- X or Z values
  - Unknown value: lowercase `x`
    - 4 bits in hex, 3 bits in octal, 1 bit in binary
  - High-impedance value: lowercase `z`
    - 4 bits in hex, 3 bits in octal, 1 bit in binary
  - Examples
    - `12'h13x`
    - `6'hx`
    - `32'bz`
  - Extending the most-significant part
    - Applied when `<size>` is bigger than the specified value
      - Filled with x if the specified MSB is x
      - Filled with z if the specified MSB is z
      - Zero-extended otherwise
    - Examples:
      - `6'hx`

Lexical Conventions (cont'd)

- Negative numbers
  - Put the sign before the `<size>`
  - Examples:
    - `-6'd3`
    - `4'd-2  // illegal`
  - Two's complement is used to store the value
- Underscore character and question marks
  - Use '`_`' to improve readability
    - `12'b1111_0000_1010`
    - Not allowed as the first character
  - '`?`' is the same as '`z`' (only regarding numbers)
    - `4'b10?? // the same as 4'b10zz`

Lexical Conventions (cont'd)

- Strings
  - As in C, use double-quotes
  - Examples:
    - `"Hello world!"`
    - `"a / b"`
    - `"text\tcolumn1\bcolumn2\n"`
- Identifiers and keywords
  - identifiers: alphanumeric characters, '_', and '$'
    - Should start with an alphabetic character or '_'
    - Only system tasks can start with '$'
  - Keywords: identifiers reserved by Verilog
  - Examples:
    - `reg value;`
    - `input clk;`

Lexical Conventions (cont'd)

- Escaped identifiers
  - Start with '\'
  - End with whitespace (space, tab, newline)
  - Can have any printable character between start and end
  - The '\' and whitespace are not part of the identifier
  - Examples:
    - `\a+b-c     // a+b-c is the identifier`
    - `\**my_name** // **my_name** is the identifier`
  - Used as name of modules

Data Types

- Value set and strengths
- Nets and Registers
- Vectors
- Integer, Real, and Time Register Data Types
- Arrays
- Memories
- Parameters
- Strings

Value Set

- Hardware modeling requirements
  - Value level
  - Value strength
    - Used to accurately model
      - Signal contention
      - MOS devices
      - Dynamic MOS
      - Other low-level details

Value Set and Strength Levels

Value level	HW Condition
0	Logic zero, false
1	Logic one, true
x	Unknown
z	High imp., floating

Strength level	Type
supply	Driving
strong	Driving
pull	Driving
large	Storage
weak	Driving
medium	Storage
small	Storage
highz	High Impedance

Registers

- Registers represent data storage elements
  - Retain value until next assignment
  - NOTE: this is not a hardware register or flipflop
  - Keyword: `reg`
  - Default value: `x`
  - Example:

```
reg reset;
initial
begin
    reset = 1'b1;
    #100 reset=1'b0;
end
```

Integer, Real, and Time Register Data Types

- Integer
  - Keyword: `integer`
  - Very similar to a vector of `reg`
    - `integer` variables are signed numbers
    - `reg` vectors are unsigned numbers, unless specified
      - reg signed [63:0] m; // 64 bit signed value
  - Bit width: implementation-dependent (at least 32-bits)
    - Designer can also specify a width:

```
integer [7:0] tmp;
```

  - Examples:

```
integer counter;
initial
   counter = -1;
```

Integer, Real, and Time Register Data Types (cont'd)

- Real
  - Keyword: `real`
  - Values:
    - Default value: 0
    - Decimal notation: `12.24`
    - Scientific notation: `3e6` (=3x10^6)
  - Cannot have range declaration
  - Example:

```
real delta;
initial
begin
   delta=4e10;
   delta=2.13;
end
integer i;
initial
   i = delta; // i gets the Value 2(rounded valueof 2.13)
```

Integer, Real, and Time Register Data Types (cont'd)

- Time
  - Used to store values of simulation time
  - Keyword: `time`
  - Bit width: implementation-dependent (at least 64)
  - `$time` system function gives current simulation time
  - Example:

```
time save_sim_time;
initial
   save_sim_time = $time;
```

Nets

- Used to represent connections between HW elements
  - Values continuously driven on nets
  - Fig. 3-1
- Keyword: `wire`
  - Default: One-bit values
    - unless declared as vectors
  - Default value: `z`
    - For `trireg`, default is `x`
  - Examples
    - `wire a;`
    - `wire b, c;`
    - `wire d=1'b0;`

Vectors

- Net and register data types can be declared as vectors (multiple bit widths)
- Syntax:
  - `wire/reg [msb_index : lsb_index] data_id;`
- Example

```
wire a;
wire [7:0] bus;
wire [31:0] busA, busB, busC;
reg clock;
reg [0:40] virtual_addr;
```

Vectors (cont'd)

- Consider

```
wire [7:0] bus;
wire [31:0] busA, busB, busC;
reg [0:40] virtual_addr;
```

- Access to bits or parts of a vector is possible:

```
busA[7]
bus[2:0] // three least-significant bits of bus
bus[0:2] // illegal
virtual_addr[0:1] /* two most-significant bits
                   * of virtual_addr
                   */
```

Vectors(cont'd)
Variable Vector Part Select

```
reg [255:0] data1; //Little endian notation
reg [0:255] data2; //Big endian notation
reg [7:0] byte;

//Using a variable part select, one can choose parts
byte = data1[31-:8]; //starting bit = 31, width =8 => data[31:24]
byte = data1[24+:8]; //starting bit = 24, width =8 => data[31:24]
byte = data2[31-:8]; //starting bit = 31, width =8 => data[24:31]
byte = data2[24+:8]; //starting bit = 24, width =8 => data[24:31]

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

//Can initialize a part of the vector
data1[(byteNum*8)+:8] = 8'b0; //If byteNum = 1, clear 8 bits [15:8]
```

Arrays

- Allowed for all data types
- Multi-dimensional
- Syntax:

```
<data_type> <var_name>[start_idx : end_idx] [start_idx :
  end_idx] ... [start_idx : end_idx];
```

- Examples:

```
integer count[0:7];
reg bool[31:0];
time chk_point[1:100]; reg
[4:0] port_id[0:7];
integer matrix[4:0][0:16];
reg [63:0] array_4d [15:0][7:0][7:0][255:0];
wire [7:0] w_array2 [5:0];
wire w_array1[7:0][5:0];
```

- Difference between vectors and arrays

Arrays (cont'd)

- Examples (cont'd)

```
integer count[0:7];
time chk_point[1:100];
reg [4:0] port_id[0:7];
integer matrix[4:0][0:16];
reg [63:0] array_4d[15:0][7:0][7:0][255:0];

count[5] = 0;
chk_point[100] = 0;
port_id[3] = 0;

matrix[1][0] = 33559;
array_4d[0][0][0][0][15:0] = 0;

port_id = 0;       // Illegal
matrix [1] = 0;    // Illegal
```

Memories

- RAM, ROM, and register-files used many times in digital systems
- Memory = array of registers in Verilog
- Word = an element of the array
  - Can be one or more bits
- Examples:

```
reg membit[0:1023];
reg [7:0] membyte[0:1023];
membyte[511]
```

- Note the difference (as in arrays):

```
reg membit[0:127];
reg [0:127] register;
```

Parameters

- Similar to `const` in C
  - But can be overridden for each module at compile-time
- Syntax:

```
parameter <const_id>=<value>;
```

- Gives flexibility
  - Allows to customize the module
- Example:

```
parameter port_id=5;
parameter cache_line_width=256;
parameter bus_width=8;
parameter signed [15:0] WIDTH;
wire [bus_width-1:0] bus;
```

Parameters (cont'd)

- `defparam` keyword
- `localparam` keyword

```
localparam state1 = 4'b0001,
           state2 = 4'b0010,
           state3 = 4'b0100,
           state4 = 4'b1000;
```

Strings

- Strings are stored in `reg` variables.
- 8-bits (1 byte) required per character
- The string is stored from the least-significant part to the most-significant part of the `reg` variable
- Example:

```
reg [8*18:1] string_value;
initial
    string_value = “Hello World!”;
```

- Escaped characters
  - \n: newline
  - %%: %
  - \”: “
  - \t: tab
  - \\: \
  - \ooo: character number in octal

System Tasks

- System Tasks: standard routine operations provided by Verilog
  - Displaying on screen, monitoring values, stopping and finishing simulation, etc.
- All start with `$`
- Instructions for the simulator

System Tasks (cont'd)

- `$display`: displays values of variables, strings, expressions.
  - Syntax: `$display(p1, p2, p3, …, pn);`
  - `p1,…, pn` can be quoted string, variable, or expression
  - Adds a new-line after displaying `pn` by default
  - Format specifiers:
    - `%d, %b, %h, %o`: display variable respectively in decimal, binary, hex, octal
    - `%c, %s`: display character, string
    - `%e, %f, %g`: display real variable in scientific, decimal, or whichever smaller notation
    - %v: display strength
    - %t: display in current time format
    - %m: display hierarchical name of this module

System Tasks (cont'd)

- `$display` examples:
  - `$display("Hello Verilog World!");`
    Output: `Hello Verilog World!`
  - `$display($time);`
    Output: `230`
  - `reg [0:40] virtual_addr;`
  - `$display("At time %d virtual address is %h", $time, virtual_addr);`
    Output: `At time 200 virtual address is 1fe000001c`

System Tasks (cont'd)

- `reg [4:0] port_id;`
- `$display("ID of the port is %b", port_id);`
  Output: `ID of the port is 00101`

- `reg [3:0] bus;`
- `$display("Bus value is %b", bus);`
  Output: `Bus value is 10xx`

- `$display("Hierarchical name of this module is %m");`
  Output: `Hierarchical name of this module is top.p1`

- `$display("A \n multiline string with a %% sign.");`
  Output: `A`
  `multiline string with a % sign.`

System Tasks (cont'd)

- `$monitor`: monitors signal(s) and displays them when their value changes
- Syntax: `$monitor(p1, p2, p3, …, pn);`
  - `p1,…, pn` can be quoted string, variable, or signal names
  - Format specifiers similar to `$display`
  - Continuously monitors the values of the specified variables or signals, and displays the entire list whenever any of them changes.
  - `$monitor` needs to be invoked only once (unlike `$display`)
    - Only one `$monitor` (the latest one) can be active at any time
    - `$monitoroff` to temporarily turn off monitoring
    - `$monitoron` to turn monitoring on again

System Tasks (cont'd)

- `$monitor` Examples:

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

  - Output:

```
0 value of signals clock=0, reset=1
5 value of signals clock=1, reset=1
10 value of signals clock=0, reset=0
```

System Tasks (cont'd)

- `$stop`: stops simulation
  - Simulation enters interactive mode
  - Most useful for debugging
- `$finish`: terminates simulation
- Examples:

```
initial
begin
   clock=0;
   reset=1;
   #100 $stop;
   #900 $finish;
end
```

Compiler Directives

- General syntax:

```
`<keyword>
```

- `` `define ``
  - similar to `#define` in C
  - `` `<macro_name> `` to use the macro defined by `` `define ``
- Examples:

```
`define WORD_SIZE 32
`define S $stop

`define WORD_REG reg [31:0]
`WORD_REG a_32_bit_reg;
```

Compiler Directives (cont'd)

- `` `include ``:
  - Similar to `#include` in C
- Example:

```
`include header.v
...
<Verilog code in file design.v>
...
```

Compiler Directives (cont'd)

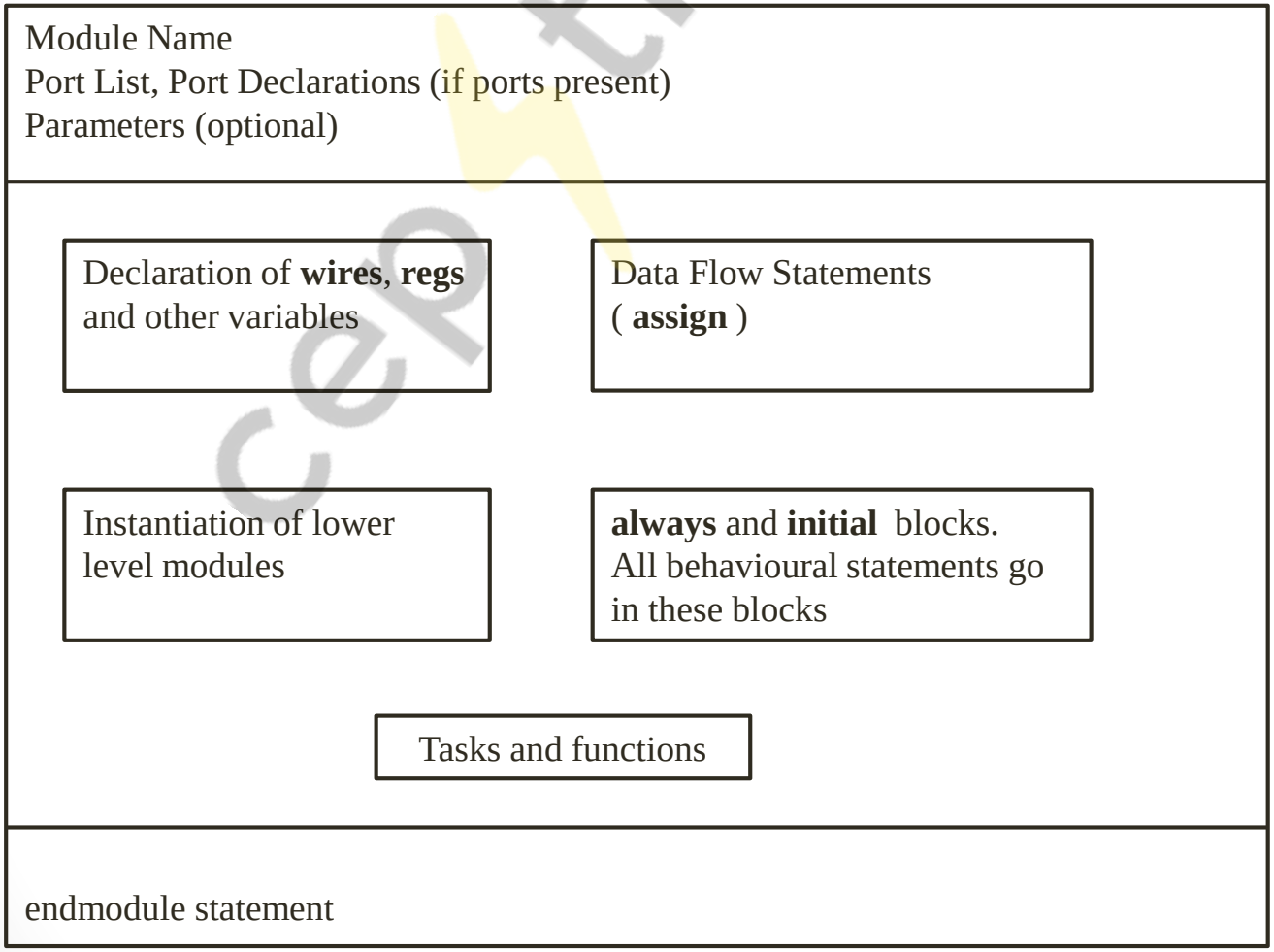
- `` `ifdef, `ifndef, `else, `elsif, and `endif. ``

```
`define TEST
'ifdef TEST //compile module test only if text macro TEST is defined
module test;
...
...
endmodule
'else //compile the module stimulus as default
module stimulus;
...
...
endmodule
'endif //completion of 'ifdef directive
```

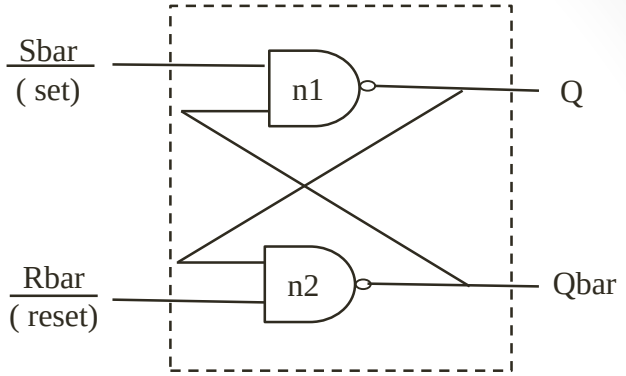
Verilog® HDL

Modules Ports

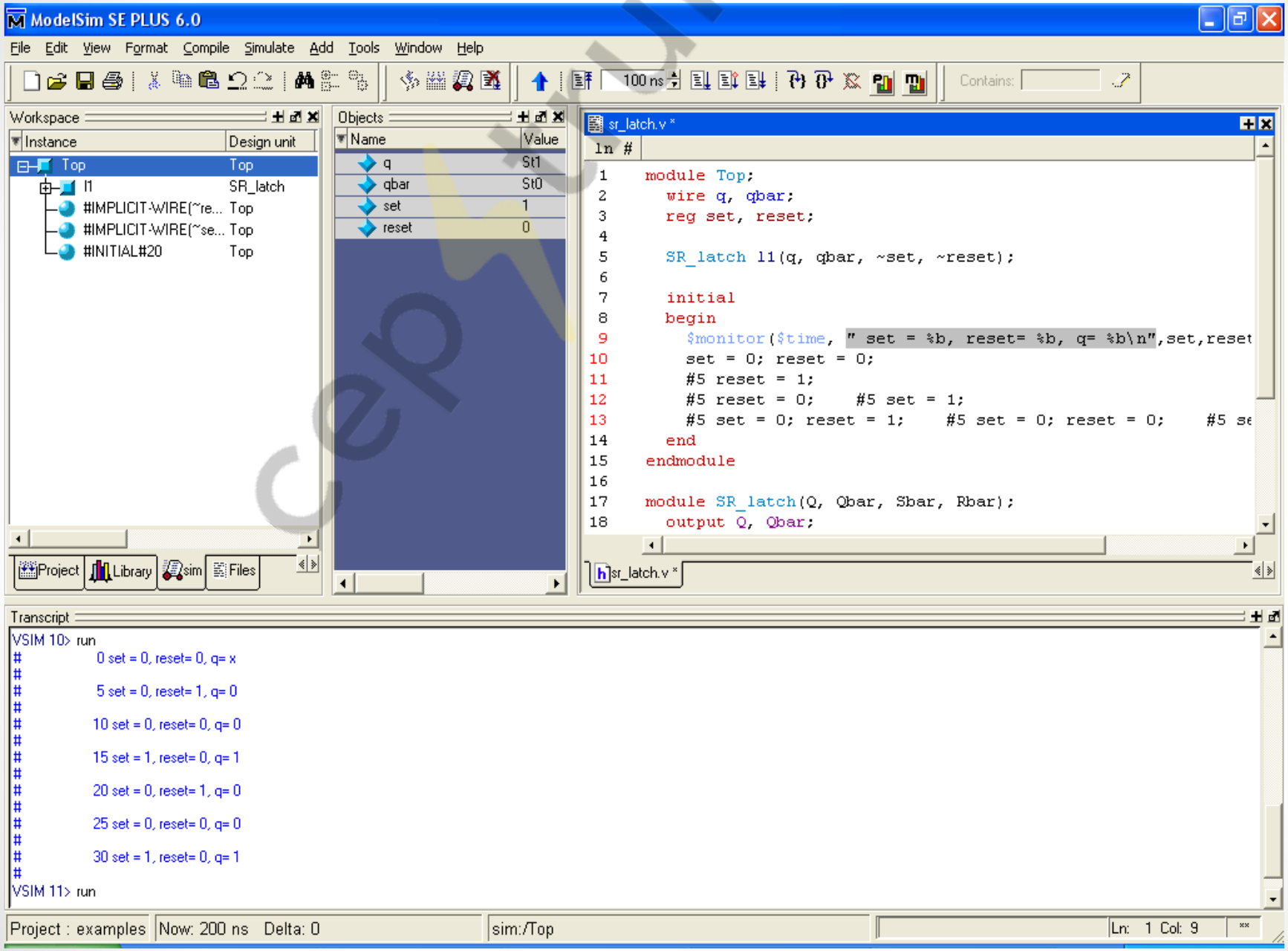
Modules

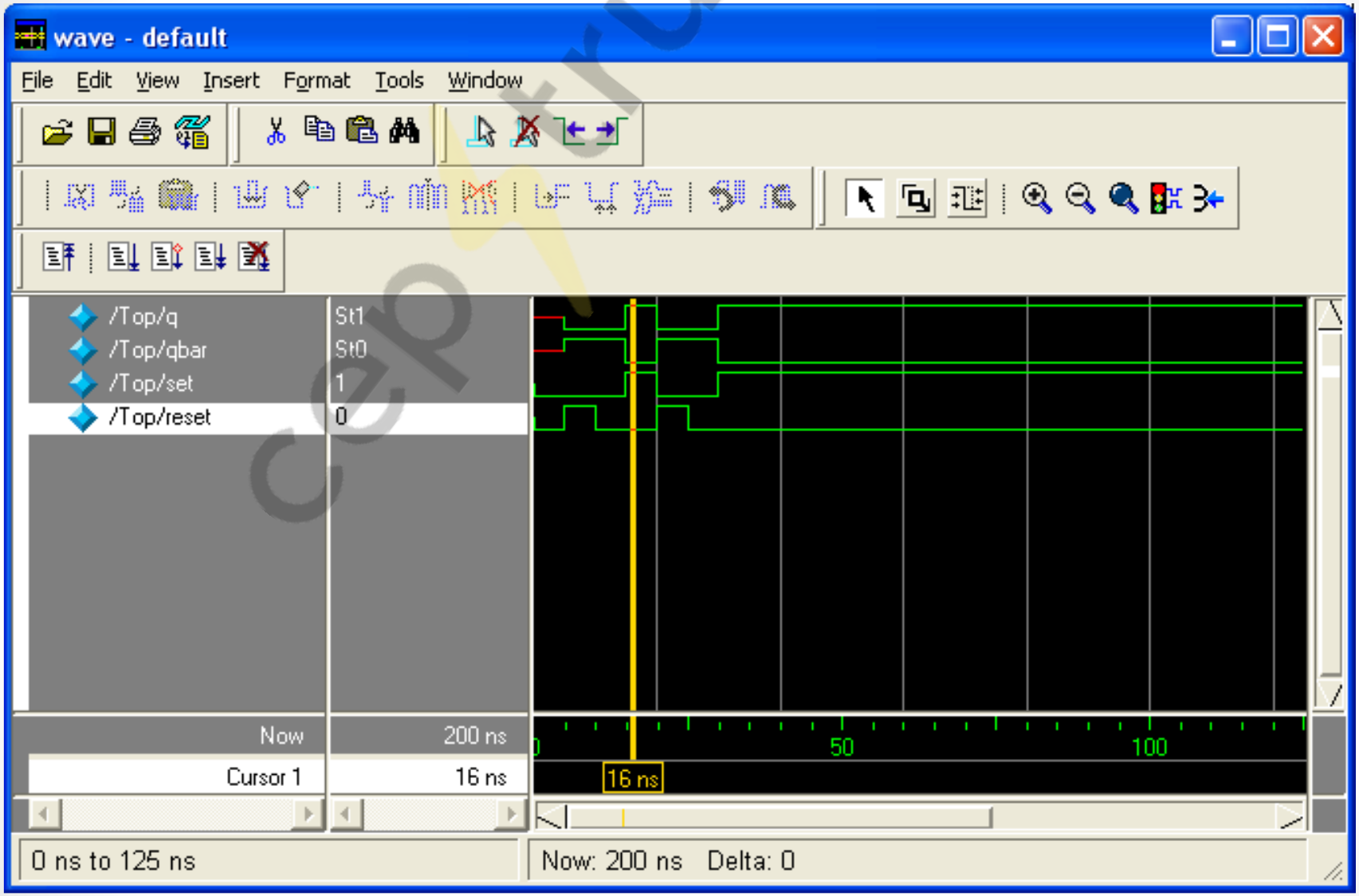

Module Name
Port List, Port Declarations (if ports present)
Parameters (optional)
Declaration of **wires**, **regs** and other variables
Data Flow Statements (**assign**)
Instantiation of lower level modules
always and **initial** blocks. All behavioural statements go in these blocks
Tasks and functions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

SR-Latch Example




```
module SR_latch(Q, Qbar, Sbar, Rbar);
  output Q, Qbar;
  input Sbar, Rbar;

  // Instantiate lower level modules
  // Note, how the wires are connected in a cross coupled fashion.
  nand n1(Q, Sbar, Qbar);
  nand n2(Qbar, Rbar, Q);
endmodule
module Top;
  wire q, qbar;
  reg set, reset;

  // Instantiate lower level modules. In this case, instantiate SR_latch
  // Feed inverted set and reset signals to the SR latch
  SR_latch l1(q, qbar, ~set, ~reset);

  // Behavioral block, initial
  initial
  begin
    $monitor($time, " set = %b, reset= %b, q= %b\n",set,reset,q);
    set = 0; reset = 0;
    #5 reset = 1;
    #5 reset = 0;    #5 set = 1;
    #5 set = 0; reset = 1;    #5 set = 0; reset = 0;    #5 set = 1; reset = 0;
  end
endmodule
```


ModelSim SE PLUS 6.0
File Edit View Format Compile Simulate Add Tools Window Help
Workspace
Instance Design unit
Top Top
I1 SR_latch
#IMPLICIT-WIRE(~re... Top
#IMPLICIT-WIRE(~se... Top
#INITIAL#20 Top
Project Library sim Files
Objects
Name Value
q St1
qbar St0
set 1
reset 0
sr_latch.v *
ln #
1 module Top;
2 wire q, qbar;
3 reg set, reset;
4
5 SR_latch l1(q, qbar, ~set, ~reset);
6
7 initial
8 begin
9 $monitor($time, " set = %b, reset= %b, q= %b\n",set,reset
10 set = 0; reset = 0;
11 #5 reset = 1;
12 #5 reset = 0; #5 set = 1;
13 #5 set = 0; reset = 1; #5 set = 0; reset = 0; #5 se
14 end
15 endmodule
16
17 module SR_latch(Q, Qbar, Sbar, Rbar);
18 output Q, Qbar;
sr_latch.v *
Transcript
VSIM 10> run
0 set = 0, reset= 0, q= x
#
5 set = 0, reset= 1, q= 0
#
10 set = 0, reset= 0, q= 0
#
15 set = 1, reset= 0, q= 1
#
20 set = 0, reset= 1, q= 0
#
25 set = 0, reset= 0, q= 0
#
30 set = 1, reset= 0, q= 1
#
VSIM 11> run
Project : examples
Now: 200 ns Delta: 0
sim:/Top
Ln: 1 Col: 9

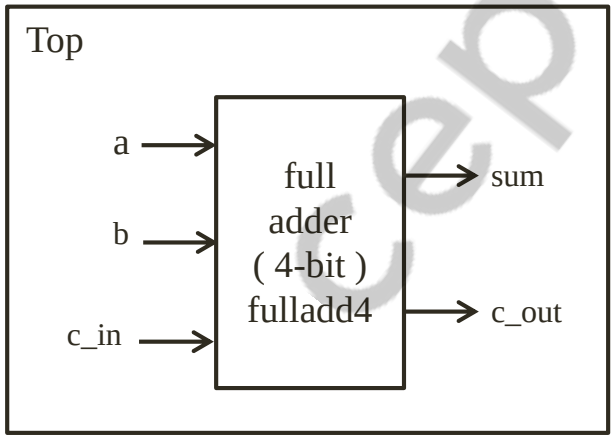

wave - default
File Edit View Insert Format Tools Window
/Top/q
/Top/qbar
/Top/set
/Top/reset
St1
St0
1
0
Now
200 ns
Cursor 1
16 ns
50
100
16 ns
0 ns to 125 ns
Now: 200 ns Delta: 0

Ports

- Interface of the module to the environment
  - Internals of the module are not visible to the environment
  - Internals can be changed as long as the interface (ports) is not changed

List of Ports

- Example: 4-bit full-adder inside a top module



I/O Ports for top and Full Adder

```
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
- Verilog keyword — Type (mode) of port

Verilog keyword	Type (mode) of port
`input`	Input port
`output`	Output port
`inout`	Bidirectional port

- Example:

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

   input [3:0] a, b;
   input c_in;
   ...
endmodule
```

Port Declaration (cont'd)

- Note: all ports are `wire` by default
  - No need to declare it again as `wire`
  - If expected to be `reg`, the port needs to be declared again (only valid for `output` ports. Why?)
  - Example: the `q` port in DFF module

```
module DFF(q, d, clk, reset);
    output q;
    reg q;  // Output port q holds value => reg
    input d, clk, reset;

    ...
endmodule
```

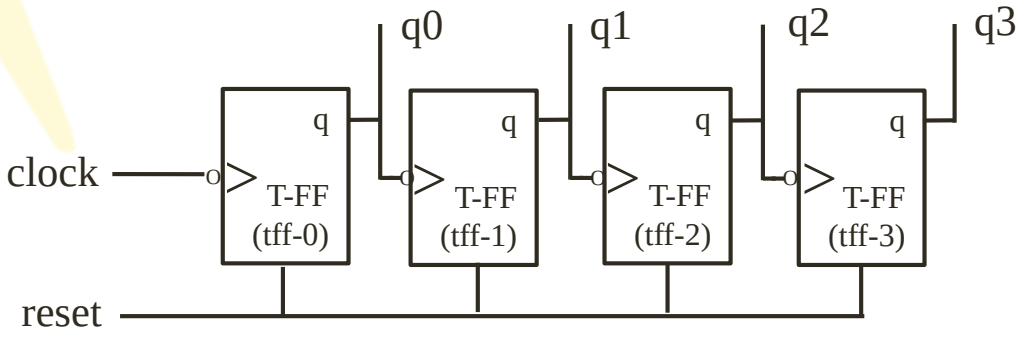
ANSI C Style Port Declaration Syntax

```
module fulladd4(output reg [3:0] sum,
                output reg c_out,
                input [3:0] a, b, //wire by default
                input c_in); //wire by default
...
<module internals>
...
endmodule
```

Module Instantiation

- Recall the Ripple-carry counter and TFF

```
module TFF(q, clk, reset);
 output q;
 input clk, reset;
 ...
endmodule
```



Ripple Carry Counter

```
module ripple_carry_counter(q, clk, reset);
 output [3:0] q;
 input clk, reset;

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

Module Instantiation (cont'd)

- General syntax

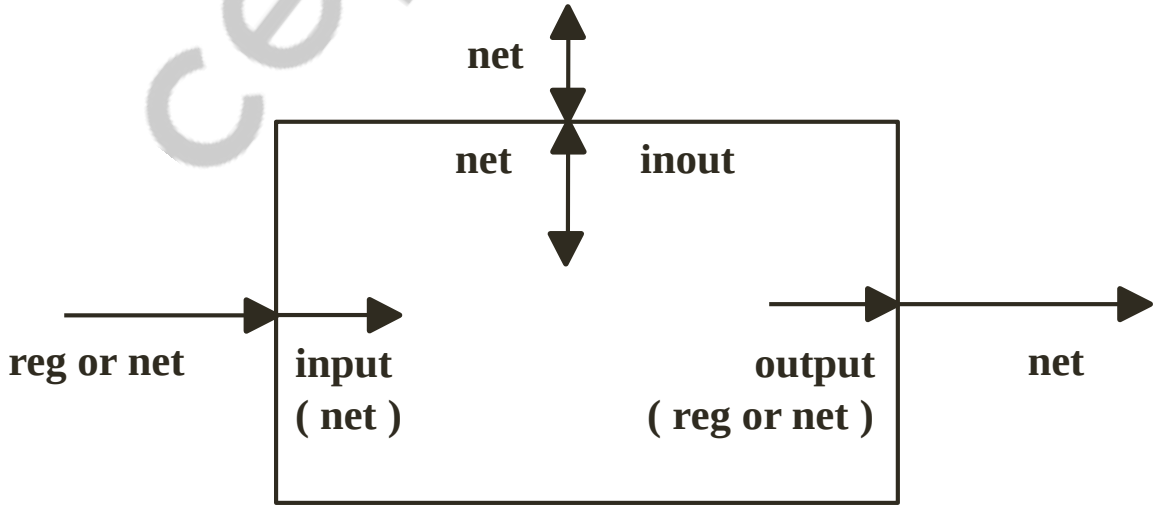
```
<module_name> <instance_name>(port connection list);
```

- Example:

```
// assuming module ripple_carry_counter(q, clk, reset);
ripple_carry_counter cntr1(wire_vec1, wire2, wire3);
```

Port Connection Rules

- Two parts of a port: internal and external to the module



Port Connection Rules (cont'd)

- Width matching
  - Legal to connect items of different sizes
    - A warning may be issued by Verilog simulator
- Unconnected ports
  - Allowed in Verilog
  - Example:

```
// module fulladd4(sum, c_out, a, b, c_in);
fulladd4 fa0(SUM, , A, B, C_IN); // output port c_out is unconnected
```

Port Connection Rules (cont'd)

- Example of illegal port connection

```
module Top;
// Declare connection variables
reg [3:0] A, B;
reg C_IN;
reg [3:0] SUM;
wire C_OUT;

// Instantiate fulladd4, call it fa0
fulladd4 fa0(SUM, C_OUT, A, B, C_IN);
// Illegal connection because output port sum is connected to reg

...

endmodule // Top
```

Connecting Ports to External Signals

- Two ways for port mapping
  - Connecting by ordered list
    - More intuitive for beginners
    - Mostly used when having few ports
  - Connecting by name
    - Used when having more ports
    - Gives independence from order of ports
      - The order of ports in the port list of a module can be changed without changing the instantiations

Connecting by Ordered List

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

module Top;
   // Declare connection variables
   reg [3:0] A, B;
   reg C_IN;
   wire [3:0] SUM;
   wire C_OUT;

   // Signals are connected to ports in order(by position)
   fulladd4 fa_ordered(SUM, C_OUT, A, B, C_IN);
  ...
endmodule
```

Connecting Ports by Name

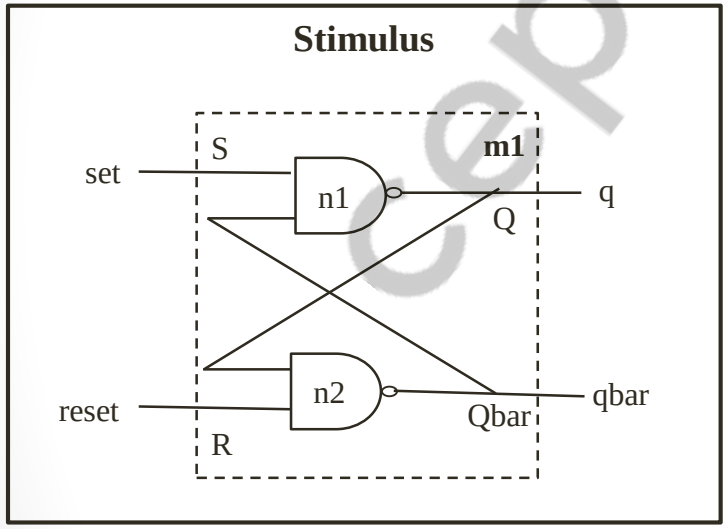
```
module fulladd4(sum, c_out, a, b, c_in);
   ...
endmodule

module Top;
   // Declare connection variables
   reg [3:0] A, B;
   reg C_IN;
   wire [3:0] SUM;
   wire C_OUT;

// Signals are connected to ports by name
fulladd4 fa_byname(.c_out(C_OUT), .sum(SUM), .b(B), .a(A));
...

   endmodule
```

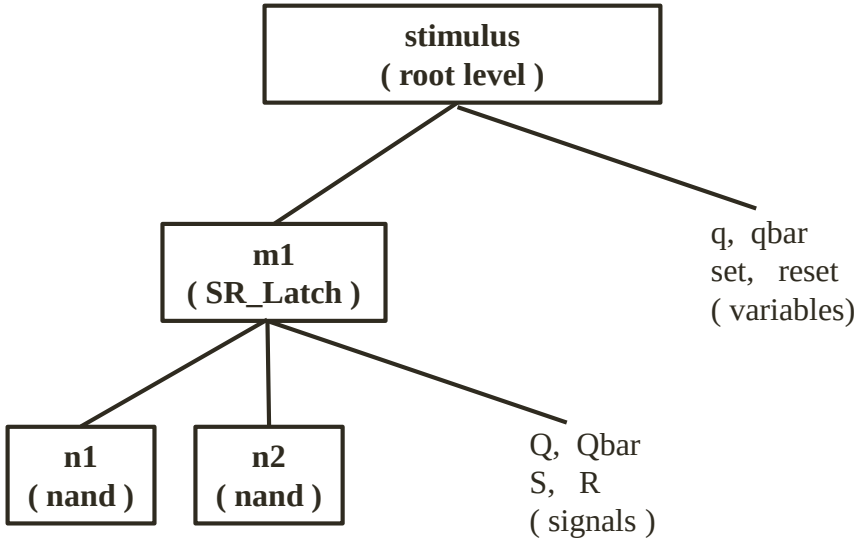
Hierarchical Names

- Hierarchical design
  - An identifier for every signal, variable, or module instance
  - The same identifier can be used in different levels of the hierarchy
  - Hierarchical name referencing
    - Unique name to every identifier in the hierarchy
    - Syntax:

```
<top-module-name>.<instance-name>.<identifier>
```

Hierarchical Names (cont'd)

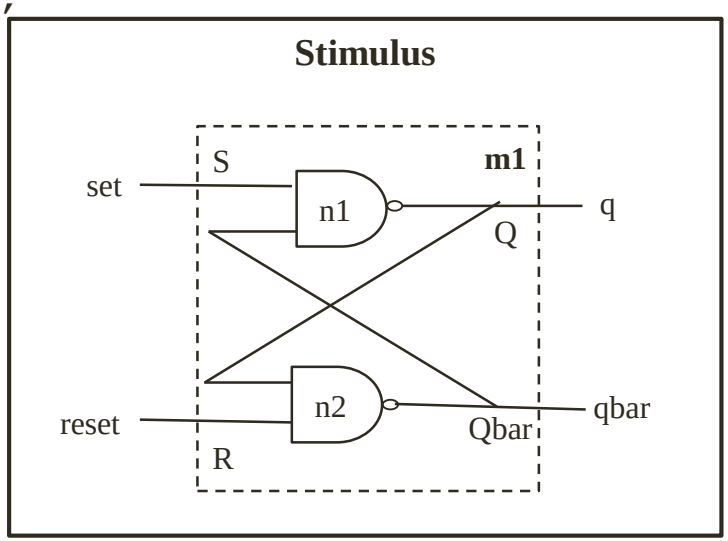




Design Hierarchy for SR Latch Simulation

Hierarchical Names (cont'd)

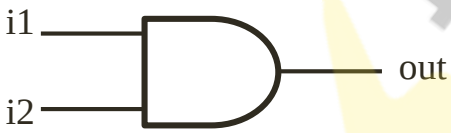
```
module stimulus;
  wire q, qbar;
  reg set, reset;
  SR_latch m1(q, qbar, ~set, ~reset);
  initial
   ...
endmodule
module SR_latch(Q, Qbar, S, R);
  output Q, Qbar;
  input S, R;
  nand n1(Q, S, Qbar);
  nand n2(Qbar, R, Q);
endmodule
```



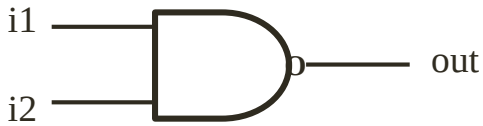
- Hierarchical names

```
stimulus          stimulus.q
stimulus.qbar     stimulus.set
stimulus.reset    stimulus.m1
stimulus.m1.Q     stimulus.m1.Qbar
stimulus.m1.S     stimulus.m1.R
stimulus.n1       stimulus.n2
```

Primitive Gates in Verilog

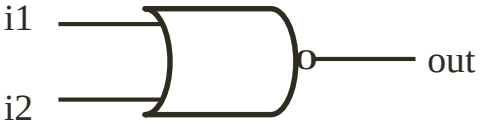


and

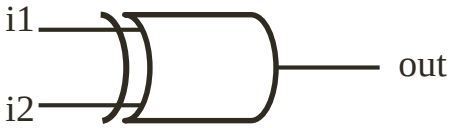


nand

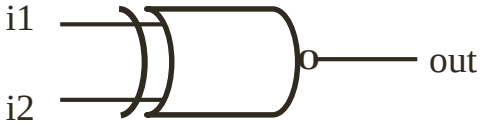
or



nor



xor



xnor

Primitive Gates in Verilog

- Syntax
  - Similar to module instantiations
- Examples

```
wire out, in1, in2, in3;

and a1(out, in1, in2);
nand na1(out, in1, in2);
nand na1_3inp(out, in1, in2, in3);
```

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

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

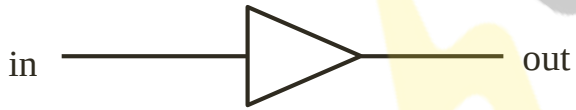
or	i1 = 0	i1 = 1	i1 = x	i1 = z
i2 = 0	0	1	x	x
i2 = 1	1	1	1	1
i2 = x	x	1	x	x
i2 = z	x	1	x	x

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

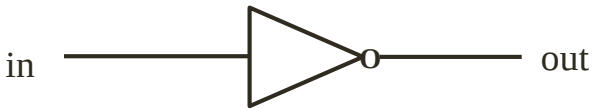
xor	i1 = 0	i1 = 1	i1 = x	i1 = z
i2 = 0	0	1	x	x
i2 = 1	1	0	x	x
i2 = x	x	x	x	x
i2 = z	x	x	x	x

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

Buf/Not Gates



buf

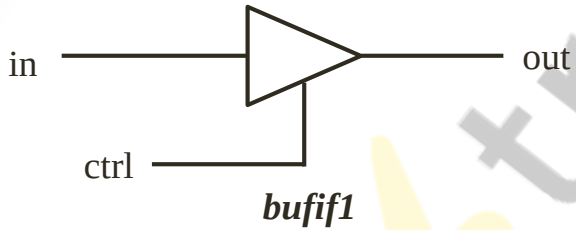


not

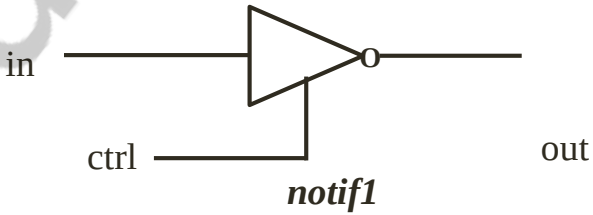
buf	in	out
	0	0
	1	1
	x	x
	z	x

not	in	out
	0	1
	1	0
	x	x
	z	x

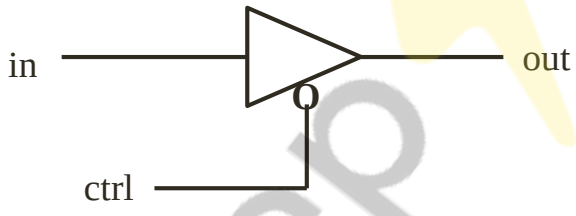
Bufif/Notif Gates



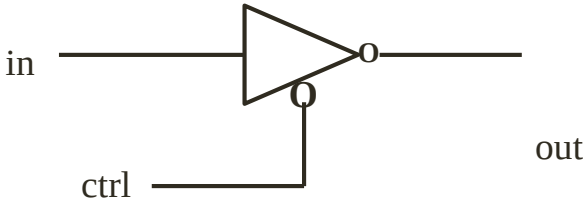
bufif1



notif1



bufif0



notif0

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

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

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

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

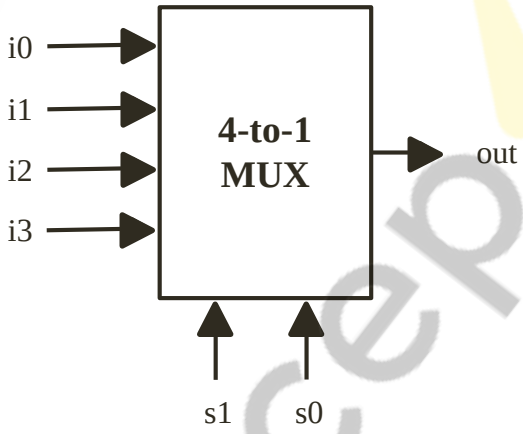
Verilog® HDL

Verilog-Data Flow Modeling

Arrays of Gate Instances

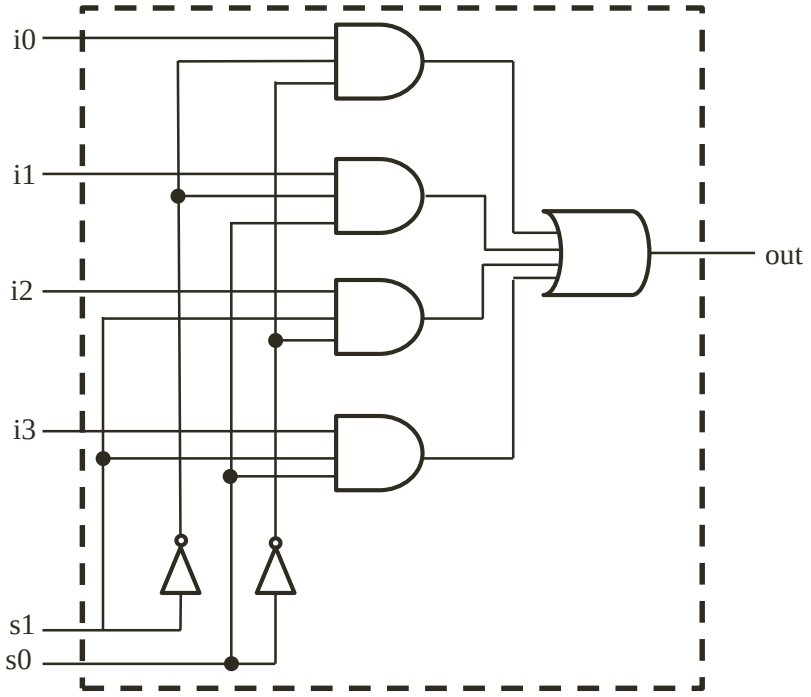
```
wire [7:0] OUT, IN1, IN2;

nand n_gate0(OUT[0], IN1[0],IN2[0]);
nand n_gate1(OUT[1], IN1[1],IN2[1]);
nand n_gate2(OUT[2], IN1[2],IN2[2]);
nand n_gate3(OUT[3], IN1[3],IN2[3]);
nand n_gate4(OUT[4], IN1[4],IN2[4]);
nand n_gate5(OUT[5], IN1[5],IN2[5]);
nand n_gate6(OUT[6], IN1[6],IN2[6]);
nand n_gate7(OUT[7], IN1[7],IN2[7]);

// The above code can be written in just two lines as shown below

wire [7:0] OUT, IN1, IN2;
nand n_gate[7:0](OUT, IN1, IN2);
```

Examples: 4-to-1 Multiplexer



s1	s0	out
0	0	i0
0	1	i1
1	0	i2
1	1	i3

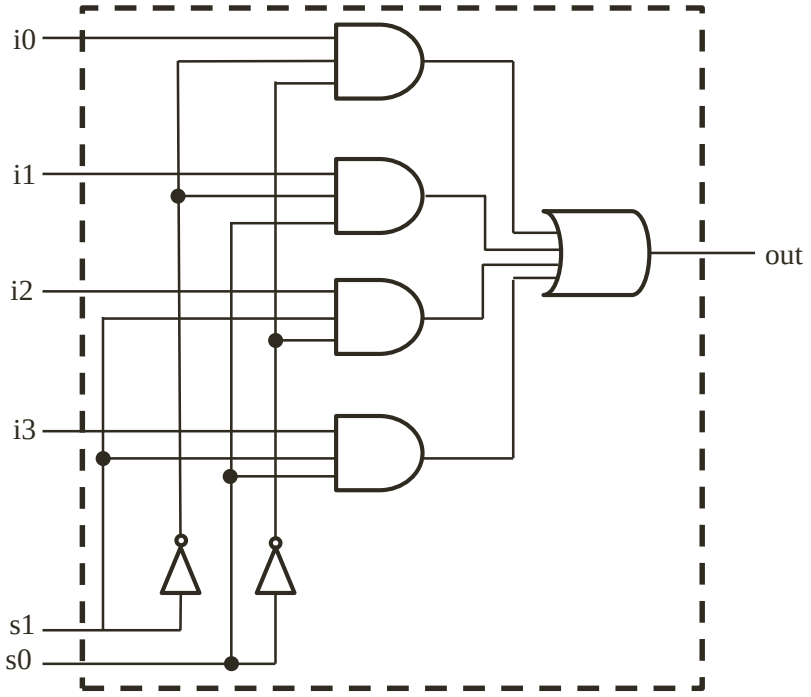
MUX Truth Table



Logic Diagram for Multiplexer

Examples: 4-to-1 Multiplexer (cont'd)

```
// Module 4-to-1 multiplexer. Port list is taken directly from
// the I/O diagram.
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);

// Port declarations from the I/O diagram
output out;
input i0, i1, i2, i3;
input s1, s0;

// Internal wire declarations
wire s1n, s0n;
wire y0, y1, y2, y3;

// Gate Instantiations

// Create s1n and s0n signals.
not (s1n, s1);
not (s0n, s0);

// 3-input and gates instantiated
and(y0, i0, s1n, s0n);
and(y1, i1, s1n, s0);
and(y2, i2, s1, s0n);
and(y3, i3, s1, s0);

// 4-input or gate instantiated
or(out, y0, y1, y2, y3);
endmodule
```

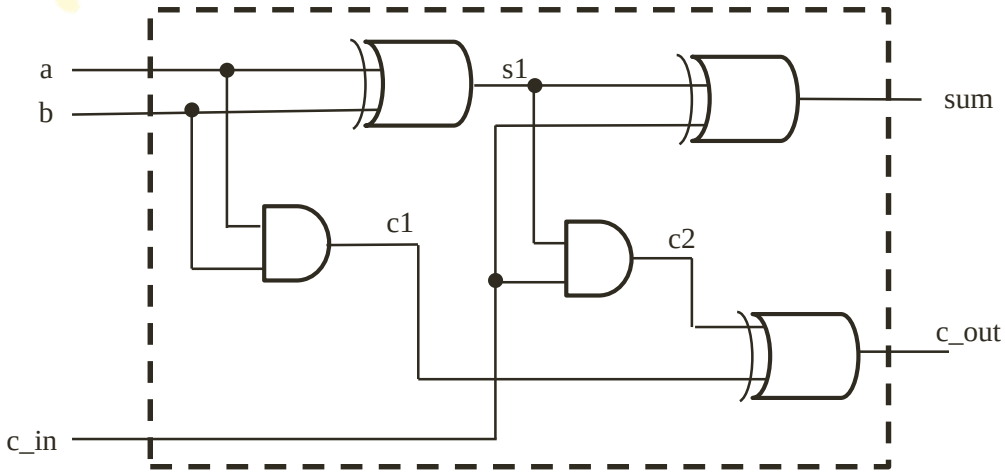


Logic Diagram for Multiplexer

```
module stimulus;

// Declare variables to be connected to inputs
reg  IN0,  IN1,  IN2,  IN3;
reg  S1,  S0;

// Declare output wire
wire  OUTPUT;

// Instantiate the multiplexer
mux4_to_1  mymux(OUTPUT,  I0,  I1,  I2,  I3,  S1,  S0);

// Stimulate the inputs
// Define the stimulus module ( no ports )
initial
begin
    // set input lines
    IN0 = 1;  IN1 = 0;  IN2 = 1;  IN3 = 0;
    #1 $display("IN0 = %b,  IN1 = %b,  IN2 = %b,  IN3 = %b\n",  IN0,  IN1,  IN2,  IN3);
    // choose IN0
    S1 = 0;  S0 = 0;
    #1 $display("S1 = %b,  S0 = %b, OUTPUT = %b\n", S1,  S0,  OUTPUT);
    // choose IN1
    S1 = 0;  S0 = 1;
    #1 $display("S1 = %b,  S0 = %b, OUTPUT = %b\n", S1,  S0,  OUTPUT);
    // choose IN2
    S1 = 1;  S0 = 0;
    #1 $display("S1 = %b,  S0 = %b, OUTPUT = %b\n", S1,  S0,  OUTPUT);
    // choose IN3
    S1 = 1;  S0 = 1;
    #1 $display("S1 = %b,  S0 = %b, OUTPUT = %b\n", S1,  S0,  OUTPUT);
end

endmodule
```

Examples: 4-bit ripple-carry full adder

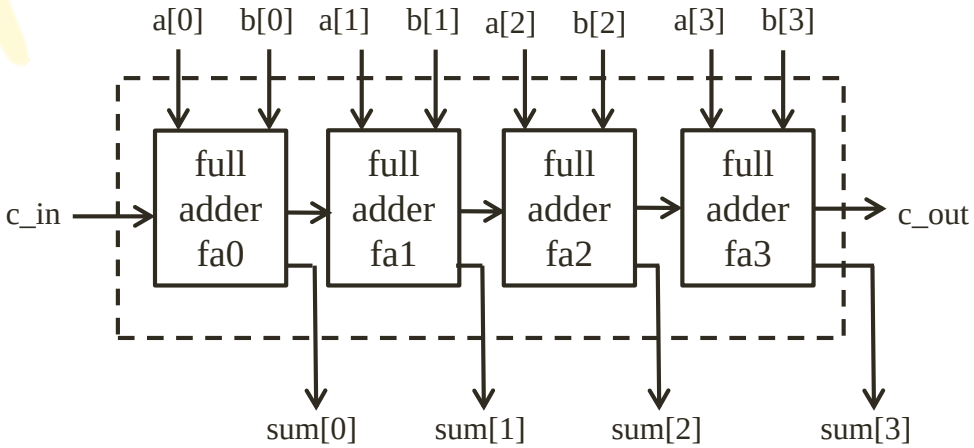
```
// Define a 1-bit Full Adder
module fulladd ( sum,  c_out,  a,  b,  c_in );

  // I/O Port Declarations
  output  sum,  c_out;
  input  a,  b,  c_in;

  // Internal Nets
  wire s1,  c1,  c2;

  // Instantiate logic gate primitives
  xor (s1,  a,  b);
  and (c1,  a,  b);

  xor (sum,  s1,  c_in);
  and (c2,  s1,  c_in);

   xor (c_out,  c1,  c2);

endmodule
```



1-bit Full Adder

Examples: 4-bit ripple-carry full adder (cont'd)

```
// Define a 4-bit Full Adder
module fulladd4 ( sum,  c_out,  a,  b,  c_in );

  // I/O Port Declarations
  output  [3:0] sum;
  output  c_out;
  input  [3:0] a,  [3:0] b;
  input  c_in;

  // Internal Nets
  wire c1,  c2,  c3;

  // Instantiate four 1-bit full adders
  fulladd4 fa0 ( sum[0],  c1,  a[0],  b[0],  c_in );
  fulladd4 fa1 ( sum[1],  c2,  a[1],  b[1],  c1 );
  fulladd4 fa2 ( sum[2],  c3  a[2],  b[2],  c2 );
  fulladd4 fa3 ( sum[3],  c_out,  a[3],  b[3],  c3 );

endmodule
```



4-bit Ripple Carry Full Adder

Stimulus Block

```
module stimulus;
// Set up variables
reg [3:0] A, [3:0] B;
reg C_IN;
wire [3:0] SUM;
wire C_OUT;
// Instantiate the 4-bit full adder. Call it FA1_4
fulladd4 FA1_4(SUM, C_OUT, A, B, C_IN);
// Set up the monitoring for the signal values
initial
begin
    $monitor($time," A = %b, B = %b, C_IN = %b, C_OUT = %b, SUM = %b\n", A, B, C_IN, C_OUT, SUM);
end

// Simulate Inputs
initial
begin
    A = 4'd0; B = 4'd0; C_IN = 1'b0;
    #5  A = 4'd3; B = 4'd4;
    #5  A = 4'd2; B = 4'd5;
    #5  A = 4'd9; B = 4'd9;
    #5  A = 4'd10; B = 4'd15;
    #5  A = 4'd10; B = 4'd5; C_IN = 1'b0;
end

endmodule
```

- Logic Synthesis
- Continuous assignment
  - The `assign` keyword

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

`assign` statement

- LHS data type: net (`wire`)
- RHS data type: any type (register, net)

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

// Continuous assign for vector nets. addr is a 16-bit vector net
// addr1 and addr2 are 16-bit vector registers.
assign  addr [15:0] = addr1_bits[15:0]^addr2_bits[15:0];

// Concatenation. Left-hand side is a concatenation of a scalar net and a vector net
assign { c_out, sum[3:0] } = a[3:0] +b[3:0] +c_in;
```

Implicit continuous assignment

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

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

Implicit net declaration

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

Gate Level vs. Dataflow Modeling

Gate Level Model

```
// Module 4-to1 Multiplexer. Port list is taken exactly from the I/O diagram.
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);

// Port declarations from the I/O diagram
output out;
input i0,  i1,  i2,  i3;
input s1,  s0;

// Internal wire declarations
wire s1n,  s0n;
wire y0,  y1,  y2,  y3;

// Gate instantiations
// Create  s1n  and  s0n  signals
not ( s1n, s1);
not (s0n, s0);

// 3-input and gates instantiated
and ( y0,  i0,  s1n,  s0n );
and ( y1,  i1,  s1n,  s0 );
and ( y2,  i2,  s1,  s0n );
and ( y3,  i3,  s1,  s0 );

// 4-input or gate instantiated
or (out,  y0,  y1,  y2,  y3 );

endmodule
```

Gate Level vs. Dataflow Modeling

Dataflow Model

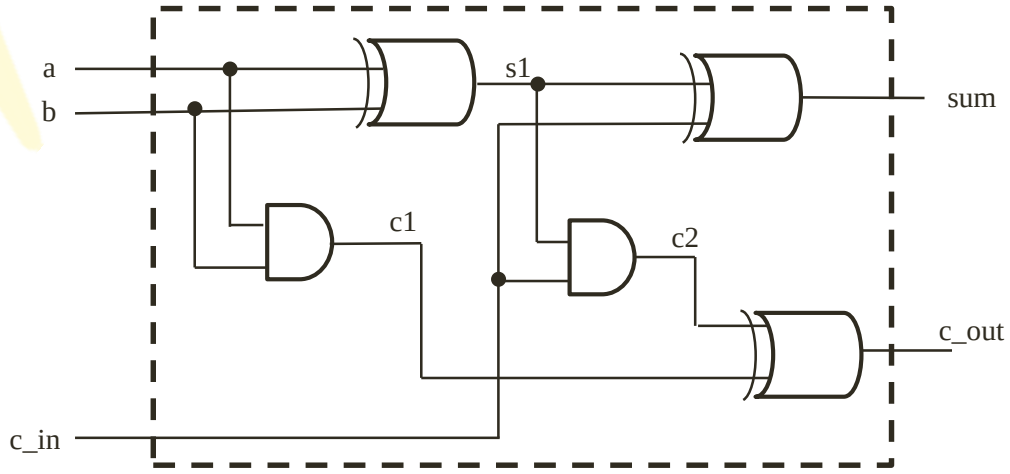
```
// Module 4-to1 Multiplexer. Port list is taken exactly from the I/O diagram.
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);

// Port declarations from the I/O diagram
output out;
input i0,  i1,  i2,  i3;
input s1,  s0;

// Logic equation for out
assign out = (~s1 & ~s0 & i0) | (~s1 & s0 & i1) | (s1 & ~s0 & i2) | (s1 & s0 & i3);

endmodule
```

Mux 4-to-1 alternative method

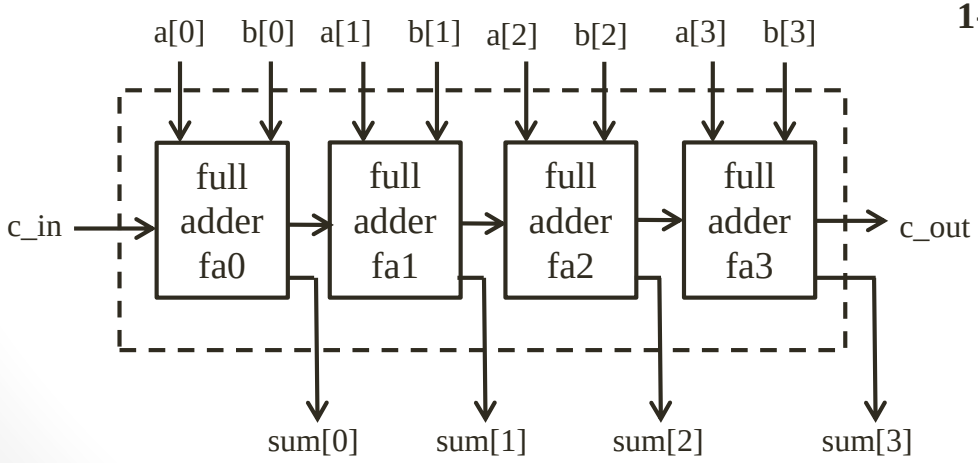
```
// Module 4-to1 Multiplexer. Port list is taken exactly from the I/O diagram.
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);

// Port declarations from the I/O diagram
output out;
input i0,  i1,  i2,  i3;
input s1,  s0;

// Logic equation for out
assign out = (~s1 & ~s0 & i0) | (~s1 & s0 & i1) | (s1 & ~s0 & i2) | (s1 & s0 & i3);
endmodule

// Alternative method – Using conditional operator
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;

// Use nested conditional operator
assign out = s1 ? ( s0 ?  i3 : i2 ) : ( s0 ?  i1 : i0 )  ;
endmodule
```

4-bit Full-adder Example

```
// Define a 4-bit full adder by using dataflow statements.
module fulladd4( sum, c_out, a, b, c_in );

// I/O port declarations
output [3:0] sum;
output c_out;
input [3:0] a, [3:0] b;
input c_in;

// Specify the function of a full adder
assign { c_out,  sum } = a + b + c_in;
endmodule
```



1-bit Full Adder



4-bit Ripple Carry Full Adder

Continuous assignment statement

- Expression
- Operand
- Operator

```
integer count, final_count;
final_count = count + 1;
```

```
real a, b, c;
c = a - b;
```

```
reg [15:0] reg1, reg2;
reg [3:0] reg_out;
reg_out = reg1[3:0] ^ reg2[3:0];
```

```
reg ret_value;
ret_value = calculate_parity(A, B);
```

Operators in Verilog

- Arithmetic
  - * / + - % **
- Logical
  - ! && ||
- Relational
  - > < <= >=
- Equality
  - == != === !===
- Bitwise
  - ~ & | ^ ^~ ~^
- Reduction
  - & ~& | ~| ^ ~^ ^~
- Shift
  - >> << >>> <<<
- Concatenation
  - { }
- Replication
  - { { } }
- Conditional
  - ? :

Arithmetic Operators

```
A = 4'b0011;
B = 4'b0100;
D = 6; E = 4; F=2;

A * B
D / E
A + B
B - A
F = E ** F;

13 % 3
16 % 4
-7 % 2
7 % -2
```

```
+5
-4

-10/5
-'d10/5  // Do NOT use
```

- 4-valued logic issue:
  `x` and `z` values

```
in1 = 4'b101x;
in2 = 4'b1010;
sum = in1 + in2;
```

Logical Operators

```
A = 3;
B = 0;

A && B
A || B
!A
!B


(A == 3) && (B == 2)
```

```
A = 2'b0;
B = 2'b10;
A && B
```

Relational Operators

```
// A = 4, B = 3

A <= B
A > B

// X = 4'b1010,
// Y = 4'b1101, Z = 4'b1xxx

Y >= X
Y < Z
```

Equality Operators

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

```
// A = 4, B = 3
// X = 4'b1010, Y = 4'b1101

A == B
X != Y
```

```
// Z = 4'b1xxz, M = 4'b1xxz
// N = 4'b1xxx

X == Z
Z === M
Z === N
M !== N
```

Bitwise Operators

```
// X = 4'b1010
// Y = 4'b1101
// Z = 4'b10x1

~X
X & Y
X | Y
X ^ Y
X ^~ Y
X & Z
```

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

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

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

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

bitwise negation	result
0	1
1	0
x	x

- Bitwise vs. Logical operators
- Bit-width mismatch issue

Reduction Operators

```
// X = 4'b1010

&X
|X
^X
```

Shift Operators

```
// X = 4'b1100

Y = X >> 1;
Y = X << 1;
Y = X <<< 2;

integer a, b, c;
a = 0;
b = -10;
c = a + (b >>> 3);
```

Concatenation Operator

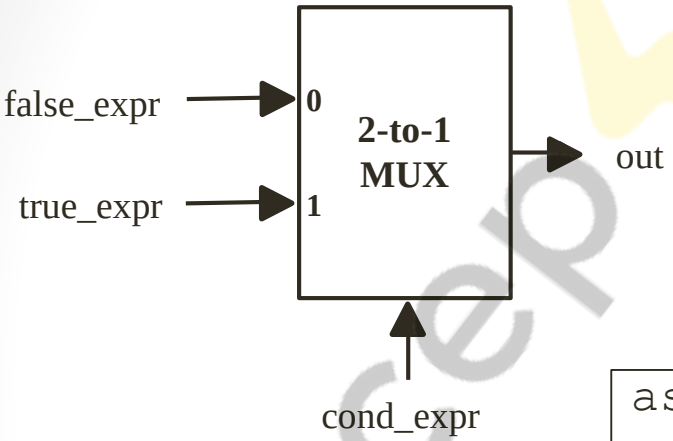
```
// A = 1'b1, B = 2'b00,
// C = 2'b10, D = 3'b110


Y = {B , C}
Y = {A , B , C , D , 3'b001}
Y = {A , B[0], C[1]}
```

- Unsized operands not allowed

Replication Operator

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

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

Y = { 4{A} }

Y = { 4{A} , 2{B} }

Y = { 4{A} , 2{B} , C }
```

Conditional Operator




```
assign addr_bus = drive_enable ?
                        addr_out : 36'bz;


assign out = control ? in1 : in0;



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

Operators Precedence

- Unary
- * / %
- + -
- >> <<
- < <= => >
- == != === !==
- Reduction
- Logical
- Conditional

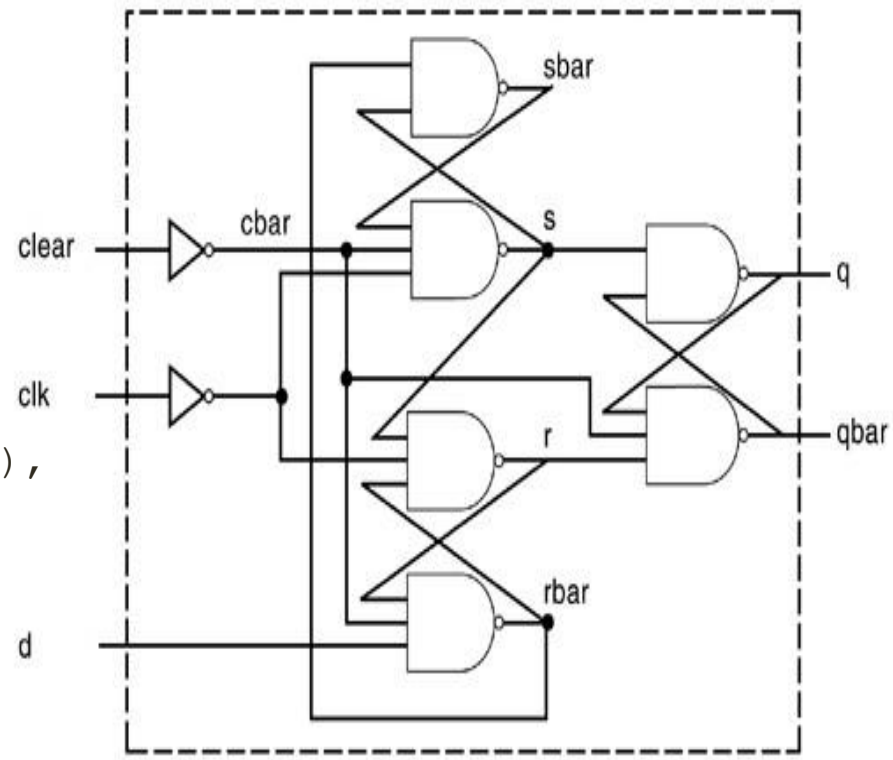
Example: Carry Look ahead Adder

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


output [3:0] sum;
output c_out;
input [3:0] a,b;
input c_in;

wire p0,g0, p1,g1, p2,g2,
p3,g3, c4, c3, c2, c1;

assign p0 = a[0] ^ b[0],
            p1 = a[1] ^ b[1],
            p2 = a[2] ^ b[2],
            p3 = a[3] ^ b[3];

assign g0 = a[0] & b[0],
            g1 = a[1] & b[1],
            g2 = a[2] & b[2],
            g3 = a[3] & b[3];
```

```
assign c1 = g0 | (p0 & c_in),
       c2 = g1 | (p1 & g0) | (p1 & p0 & c_in),
       c3 = g2 | (p2 & g1) | (p2 & p1 & g0) | (p2 & p1 & p0 & c_in),
       c4 = g3 | (p3 & g2) | (p3 & p2 & g1) | (p3 & p2 & p1 & g0) | (p3 & p2 & p1 & p0 & c_in);

assign sum[0] = p0 ^ c_in,

       sum[1] = p1 ^ c1,

       sum[2] = p2 ^ c2,

       sum[3] = p3 ^ c3;

assign c_out = c4;

endmodule
```

Negative Edge Triggered D-FF

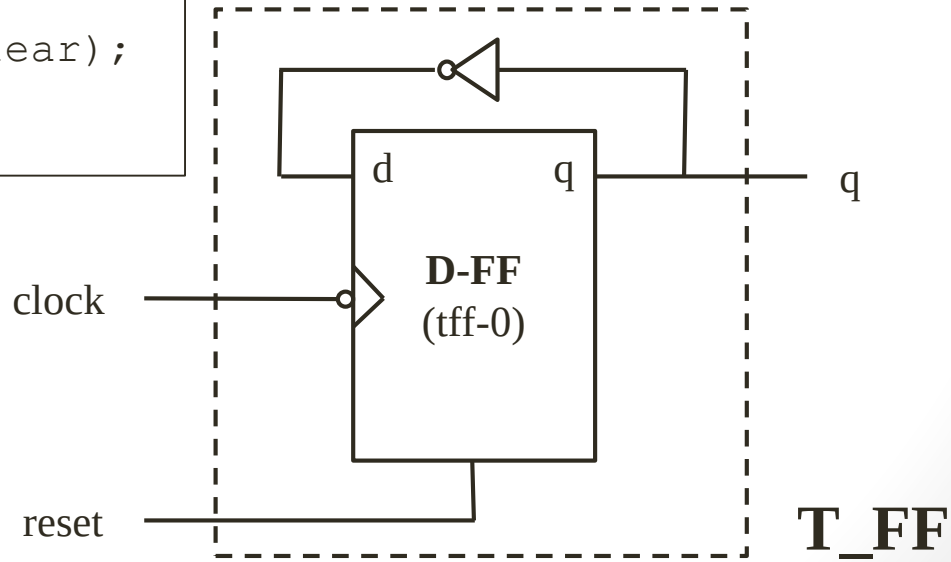
```
module edge_dff(q, qbar, d, clk,
clear);

output q,qbar;
input d, clk, clear;

wire s, sbar, r, rbar,cbar;

assign cbar = ~clear;

assign  sbar = ~(rbar & s),
        s = ~(sbar & cbar & ~clk),
        r = ~(rbar & ~clk & s),
        rbar = ~(r & cbar & d);

assign  q = ~(s & qbar),
        qbar = ~(q & r & cbar);

endmodule
```



Edge-Triggered T-FF

```
module edge_dff(q, qbar, d, clk,
clear);

. . .

endmodule
```

```
module T_FF(q, clk, clear);

output q;
input clk, clear;

edge_dff ff1(q, ,~q, clk, clear);

endmodule
```



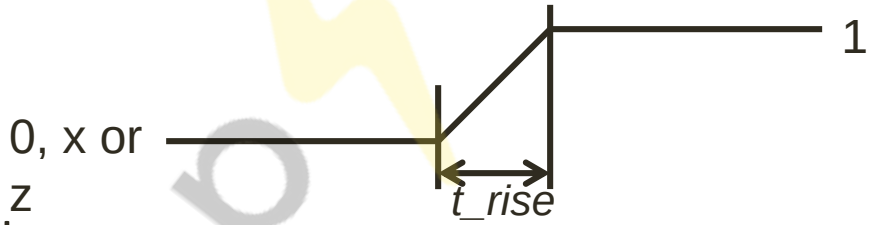
Verilog® HDL

Different delays in Verilog

Gate Delays

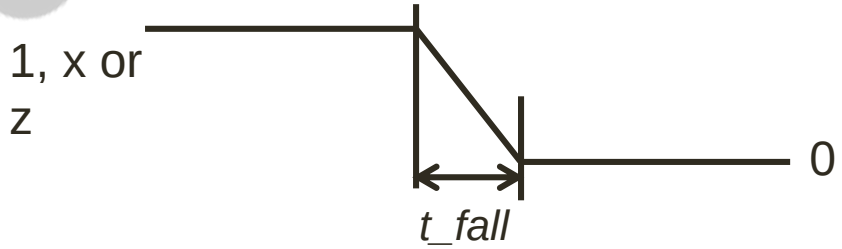
- Rise Delay

Rise delay is associated with a gate output transition to a 1 from another value



- Fall Delay

The fall delay is associated with a gate output transition to a 0 from another value



- Turn-off Delay

The turn-off delay is associated with a gate output transition to the high impedance value (z) from another value. If the value changes to x, the minimum of the three delays is considered.

Types of delay specification

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

// Rise and Fall delay specification
and #(rise_val, fall_val, turnoff_val) a2(out,i1,i2);

// Rise, fall, and turn-off delay specification
bufif0 #(rise_val, fall_val, turnoff_val) b1(out,in,control);
```

Examples of delay specification

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

and #(4,6) a2(out,i1,i2); //rise=4, fall=6

bufif0 #(3,4,5) b1(out,in,control); //rise=3, fall=4, turn-off=5
```

Minimum, Typical, Maximum values

Verilog provides an additional level of control for each type of delay discussed in previous slides (rise, fall and turn-off)

- Min value

The minimum delay value that the designer expects the gate to have

- Typ value

The typical delay value that the Verilog designer expects the gate to have

- Max value

The max value is the maximum delay value that the designer expects the gate to have

Min, typ or max values can be chosen at Verilog runtime. Method for choosing a min/max/typ value may vary for different simulators or operating systems. For Verilog-XL, the values are chosen by specifying options **`+maxdelays`**, **`+typdelays`**, and **`+mindelays`** at run time. If no option is specified the typical delay value is the default.

Min, Max and Typical delay values

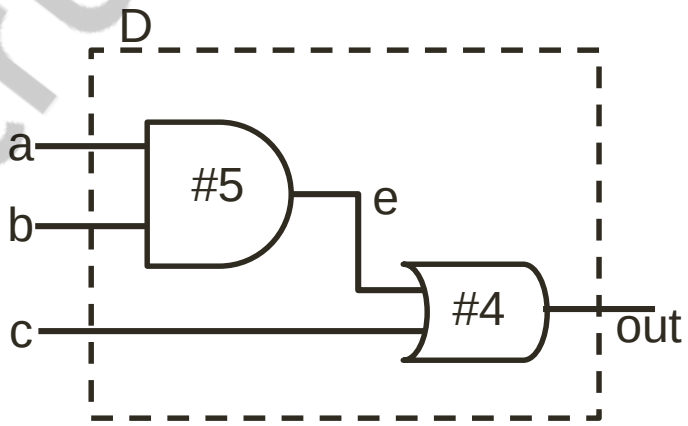
```
//one delay
//if +mindelays, delay=4
//if +typdelays, delay=5
//if +maxdelays, delay=6
and #(4:5:6) a1(out,i1,i2);

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

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

Invoking Verilog-XL simulator with command line

```
//invoke simulator with maximum delay
➢ verilog test.v +maxdelays

//invoke simulation with minimum delay
➢ verilog test.v +mindelays

//invoke simulation with typical delay
➢ verilog test.v +typdelays
```

Delay example

out= (a.b) +c




```
//define a simple combination module called D
module D (out, a, b, c);

//I/O port declarations
output out;
input a,b,c;

//Internal nets
wire e;

//Instantiate primitive gates to build the circuit
and #(5) a1(e, a, b); //Delay of 5 on gate a1
or #(4) o1(out, e, c); //Delay of 4 on gate o1

endmodule
```

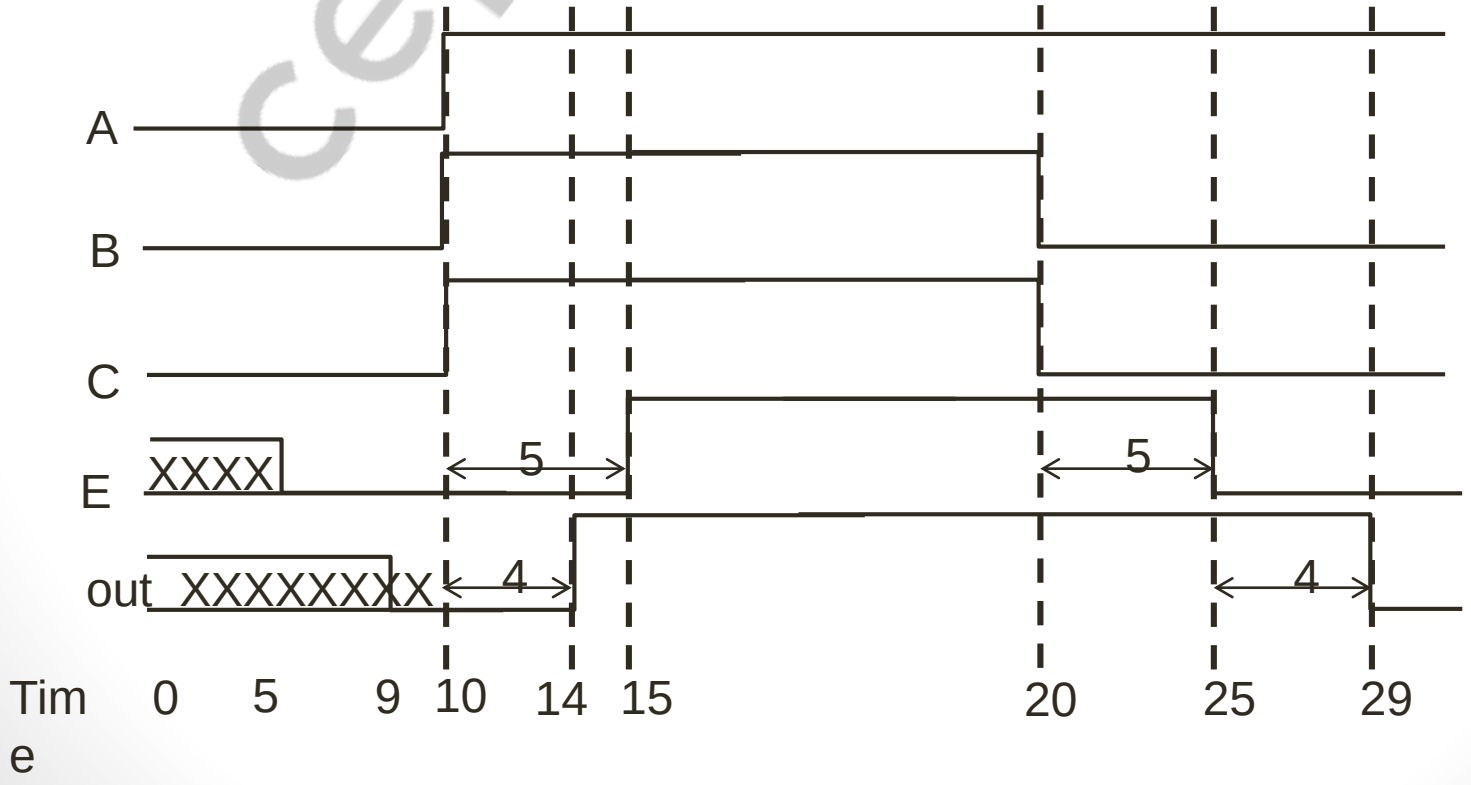
Delay example cont.

```
//Stimulus (top-level module)
module stimulus;

//Declare variables
reg A, B, C;
wire OUT;

//Instantiate the module D
D d1 (OUT, A, B, C);

//Stimulate the inputs. Finish the simulation at 40 time
//units.
initial
begin
      A= 1'b0; B= 1'b0; C= 1'b1;

      #10 A= 1'b0; B= 1'b0; C= 1'b1;

      #10 A= 1'b0; B= 1'b0; C= 1'b1;

      #20 $finish;
end

endmodule
```

Delay example cont.

Waveform of the stimulus block applied on Module D and corresponds to the fate delays shown below

1. *The outputs E and OUT are initially unknown*
2. *At time 10, after A,B and C all transition to 1, OUT transitions to 1 after a delay of time units and E changes value to 1 after 5 time units*
3. *At time 20, B and C transition to 0. E changes value to 0 after 5 time units, and OUT transitions to 0, 4 time units after E changes.*



Important notes on delays

- Basic types of gates are and, or, xor, buf & not each have Verilog primitive which are instantiated like modules except that they are predefined in Verilog. Output of a gate is evaluated as soon as one of its inputs changes.

- Three types of delays are associated with gates, rise, fall and turn off. Verilog allows specification of one, two or three delays for each gates. Values of rise fall and turn-off delays are computed by Verilog, based on the one, two or three delays specified

- For each type of delay minimum, typical and maximum value can be specified. The user can choose which value to apply at simulation time. This provides the flexibility to experiment with three delay values without changing the Verilog code.

- The effect of propagation delay on waveform was explained by simple, two-gate logic example. For each gate with a delay of t, the output changes t time units after any f the inputs changes.

Verilog® HDL

Verilog Behavioral Modeling

Introduction

- The move toward higher abstractions
  - Gate-level modeling
    - Netlist of gates
  - Dataflow modeling
    - Boolean function assigned to a net
  - Now, behavioral modeling
    - A sequential algorithm (quite similar to software) that determines the value(s) of signal(s)

Structured Procedures

- Two basic structured procedure statements

  `always`

  `initial`

  - All behavioral statements can appear only inside these blocks
  - Each always or initial block has a separate activity flow (concurrency)
  - Start from simulation time 0
  - No nesting

Structured Procedures: `initial` statement

- Starts at time 0
- Executes only once during a simulation
- Multiple `initial` blocks, execute in parallel
  - All start at time 0
  - Each finishes independently
- Syntax:

```
initial
begin
  // behavioral statements
end
```

Structured Procedures: `initial` statement (cont'd)

Example

```
module stimulus;
  reg x, y, a, b, m;
  initial
    m = 1'b0;
  initial
    begin
      #5 a = 1'b1;
      #25 b = 1'b0;
    end
  initial
    begin
      #10 x = 1'b0;
      #25 y = 1'b1;
    end
  initial
    #50 $finish;
endmodule
```

time	statement executed
0	m = 1'b0;
5	a = 1'b1;
10	x = 1'b0;
30	b = 1'b0;
35	y = 1'b1;
50	$finish;

Simpler Syntax

- Combined declaration and initialization

```
reg clock;
initial clock = 0;

reg clock = 0;

module adder (output reg [7:0] sum = 0,
              output reg co = 0,
              input [7:0] a, [7:0] b,
              input ci);
   _ _ _ _
endmodule
```

Structured Procedures: `always` statement

- Start at time 0
- Execute the statements in a looping fashion
- Example

```
module clock_gen;
  reg clock;

  // Initialize clock at time zero
  initial
    clock = 1'b0;

  // Toggle clock every half-cycle (time period =20)
  always
    #10 clock = ~clock;

  initial
    #1000 $finish;
endmodule
```

Can we move this to the *always* block ?

What happens if such a $finish is not included ?

Procedural Assignments

- Assignments inside `initial` and `always`
- To update values of “register” data types
  - The value remains unchanged until another procedural assignment updates it
  - Compare to continuous assignment (Dataflow Modeling, previous chapter)

Procedural Assignments (cont'd)

- Syntax
  - `<lvalue> = <expression>`
  - `<lvalue>` can be
    - `reg`, `integer`, `real`, `time`
    - A bit-select of the above (e.g., `addr[0]`)
    - A part-select of the above (e.g., `addr[31:16]`)
    - A concatenation of any of the above
  - `<expression>` is the same as introduced in modeling dataflow
  - What happens if the widths do not match?
    - LHS wider than RHS => RHS is zero-extended
    - RHS wider than LHS => RHS is truncated
      (Least significant part is kept)

Behavioral Modeling Statements: Conditional Statements

- Just the same as `if-else` in C
- Syntax:

```
if (<expression>)
true_statement;
if (<expression>)
true_statement;
  else  false_statement;
if (<expression>)
true_statement1;
  else if (<expression>)
  else true_statement2;
  else if (<expression>)
       true_statement3;
       default_statement;
```

- True is `1` or non-zero
- False is `0` or ambiguous (`x` or `z`)
- More than one statement: `begin end`

Conditional Statements (cont'd)

- Examples:

```
if (!lock) buffer = data;

if (enable) out = in;

if (number_queued < MAX_Q_DEPTH)
begin
  data_queue = data;
  number_queued = number_queued +1;
end
else $display("Queue full! Try again.");

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.");
```

Behavioral Modeling Statements: Multiway Branching

- Similar to `switch-case` statement in C
- Syntax:

```
case (<expression>)
   alternative1: statement1;
   alternative2: statement2;
      ...
   default: default_statement; // optional
endcase
```

- Notes:
  - `<expression>` is compared to the `alternatives` in the order specified.
  - Default statement is optional

Multiway Branching (cont'd)

- Examples:

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

- Now, you write a 4-to-1 multiplexer.

Multiway Branching (cont'd)

- Example 2:

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

  always @(s1 or s0 or i0 or i1 or i2 or i3)
    case ({s1,s0})
      2'd0: out = i0;
      2'd1: out = i1;
      2'd2: out = i2;
      2'd3: out = i3;
    endcase
endmodule
```

Multiway Branching (cont'd)

- The `case` statements compare <expression> and alternatives bit-for-bit
  - `x` and `z` values should match

```
module demultiplexer1_to_4(out0, out1, out2, out3, in, s1, s0);
    output out0, out1, out2, out3;
    input in, s1, s0;
    always @(s1 or s0 or in)
      case( {s1, s0} )
        2'b00: begin ... end
        2'b01: begin ... end
        2'b10: begin ... end
        2'b11: begin ... end
        2'bx0, 2'bx1, 2'bxz, 2'bxx, 2'b0x, 2'b1x, 2'bzx:
               begin ... end
        2'bz0, 2'bz1, 2'bzz, 2'b0z, 2'b1z:
               begin ... end
        default: $display("Unspecified control signals");
      endcase
  endmodule
```

Multiway Branching (cont'd)

- `casex` and `casez` keywords
  - `casez` treats all `z` values as "don't care"
  - `casex` treats all `x` and `z` values as "don't care"
- Example:

```
reg [3:0]
integer state;
casex(encoding)
   4'b1xxx: next_state=3;
   4'bx1xx: next_state=2;
   4'bxx1x: next_state=1;
   4'bxxx1: next_state=0;
   default: next_state=0;
endcase
```

Behavioral Modeling Statements: Loops

- Loops in Verilog
  - `while, for, repeat, forever`
- The `while` loop syntax:

```
while (<expression>)
    statement;
```

`while` Loop Example

```
integer count;

initial
begin
        count = 0;

        while (count < 128)
        begin
                $display("Count = %d", count);
               count = count + 1;
        end
end
```

```
'define TRUE 1'b1';
'define FALSE 1'b0;
reg [15:0] flag;
integer i; //integer to keep count
reg continue;

initial
begin
  flag = 16'b 0010_0000_0000_0000;
  i = 0;
  continue = 'TRUE;

  while((i < 16) && continue )
  begin
    if (flag[i])
    begin
      $display("Encountered a TRUE bit at element number %d", i);
      continue = 'FALSE;
    end
    i = i + 1;
  end
end
```

Loops (cont'd)

- The `for` loop
  - Similar to C
  - Syntax:

```
for( init_expr; cond_expr; change_expr)
   statement;
```

  - Example:

```
integer count;

initial
    for ( count=0; count < 128; count = count + 1)
          $display("Count = %d", count);
```

Loops (cont'd)

- The `repeat` loop
  - Syntax:

```
repeat( number_of_iterations )
    statement;
```

  - The number is evaluated only when the loop is first encountered

```
integer count;
initial
begin
   count = 0;
   repeat(128)   begin
        $display("Count = %d", count);
        count = count + 1;
   end
end
```

```
module data_buffer(data_start, data, clock);
parameter cycles = 8;
input data_start;
input [15:0] data;
input clock;
reg [15:0] buffer [0:7];
integer i;

always @(posedge clock)
begin
  if(data_start) //data start signal is true
  begin
    i = 0;
    repeat(cycles)
    begin
      @(posedge clock) buffer[i] = data;
        i = i + 1;
    end
  end
end

endmodule
```

Loops (cont'd)

- The `forever` loop
  - Syntax:

```
forever
    statement;
```

  - Equivalent to `while(1)`

`forever` Loop Example

```
reg clock;

initial
begin
        clock = 1'b0;
        forever #10 clock = ~clock;
end

reg clock;
reg x, y;

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

Procedural Assignments (cont'd)

- The two types of procedural assignments
  - Blocking assignments
  - Non-blocking assignments
- Blocking assignments
  - are executed in order (sequentially)
  - Example:

```
reg x, y, z;
reg [15:0] reg_a, reg_b;
integer count;
initial begin
x=0; y=1; z=1;
count=0; reg_a= 16'b0; reg_b = reg_a;
#15 reg_a[2] = 1'b1;
 #10 reg_b[15:13] = {x, y, z};
 count = count + 1;
end
```

All executed at time 0

executed at time 15

All executed at time 25

Procedural Assignments (cont'd)

- Non-blocking assignments
  - The next statements are not blocked for this one
  - Syntax:

```
<lvalue> <= <expression>
```

  - Example:

```
reg x, y, z;
reg [15:0] reg_a, reg_b;
integer count;
initial begin
  x=0; y=1; z=1;
  count=0;
  reg_a= 16'b0; reg_b = reg_a;
  reg_a[2] <= #15 1'b1;
  reg_b[15:13] <= #10 {x, y, z};
  count <= count + 1;
end
```

All executed at time 0

Scheduled to run at time 15

Scheduled to run at time 10

Procedural Assignments (cont'd)

- Application of non-blocking assignments
  - Used to model concurrent data transfers
  - Example: Write behavioral statements to swap values of two variables

```
always @(posedge clock)
begin
   reg1 <= #1 in1;
   reg2 <= @(negedge clock) in2 ^ in3;
   reg3 <= #1 reg1;
end
```

The old value of reg1 is used

Procedural Assignments (cont'd)

- Race condition
  - When the final result of simulating two (or more) concurrent processes depends on their order of execution
- Example:

```
always @(posedge clock)
   b = a;
always @(posedge clock)
   a = b;
```

- Solution:

```
always @(posedge clock)
   b <= a;
always @(posedge clock)
   a <= b;
```

```
always @(posedge clock)
begin
  temp_b = b;
  temp_a = a;
  b = temp_a;
  a = temp_b;
end
```

Procedural Assignments (cont'd)

- Recommendation
  - Concurrent data transfers => race condition
  - Use non-blocking assignments for concurrent data transfers
  - Example: pipeline modelling
  - Disadvantage:
    - Lower simulation performance
    - Higher memory usage in the simulator

Timing

- No timing controls $\Rightarrow$ No advance in simulation time
- Three methods of timing control
  - delay-based
  - event-based
  - level-sensitive

Delay-based Timing Controls

- **Delay** ≡ Duration between encountering and executing a statement
- Delay symbol: #
- Delay specification syntax:

<delay>

::= #<NUMBER>

||= #<identifier>

||= #<mintypmax_exp <,<mintypmax_exp>>*
>)

Delay-based Timing Controls (cont'd)

- Types of delay-based timing controls
  1. Regular delay control
  2. Intra-assignment delay control
  3. Zero-delay control

Regular Delay Control

- Symbol: non-zero delay before a procedural assignment

```
// define parameters
parameter latency = 20;
parameter delta = 2;

// define register variables
reg  x,  y,  z,  p,  q;

initial
begin
      x = 0;                        // no delay control.
      #10 y = 1;                    // delay control with a number.  Delay execution of
                                    //  y = 1 by 10 units.
      #latency z = 0                // Delay control with identifier.  Delay of 20 units.
      #(latency+delta) p = 1;       // Delay control with expression.
      #y x = x + 1;                 // Delay control with identifier. Take value of y.
      #(4:5:6) q = 0;               // Minimum,  typical and maximum delay values.
end
```

Intra-assignment Delay Control

- Symbol: non-zero delay to the right of the assignment operator
- Operation sequence:
  1. Compute the RHS expression at current time.
  2. Defer the assignment of the above computed value to the LHS by the specified delay.

Intra-Assignment Delay Examples

```
// define register variables
reg x,  y,  z;

// intra assignment delays
initial
begin
      x = 0;  z = 0;
      y = #5 x + z;      // Take value of x and z at the time=0, evaluate x + z and
                         // wait 5 time units to assign value to y.
end
// Equivalent method with temporary variables and regular delay control
initial
begin
      x = 0;  z = 0;
      temp_xz = x + z;
      #5 y = temp_xz;    // Take value of x + z at the current time and store it in a
                         // temporary variable. Even though x and z might change
                         // between time 0 and 5, the value assigned to y at time 5 is
                         // unaffected.
end
```

Zero-Delay Control

- Symbol: `#0`
- Different initial/alwaysblocks in the same simulation time
  - Execution order non-deterministic
- Ensures execution after all other statements
  - Eliminates race conditions
- Multiple zero-delay statements
  - Non-deterministic execution order

Zero-delay Control Examples

```
initial
begin
     x = 0;
     y = 0;
end

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

Event-based Timing Control

- Event
  - Change in the value of a register or net
  - Used to trigger execution of a statement or block (reactive behavior/reactivity)
- Types of Event-based timing control
  1. Regular event control
  2. Named event control
  3. Event OR control
  4. Level-sensitive timing control (next section)

Regular Event Control

- Symbol: `@(<event>)`
- Events to specify:
  - `posedge sig`
    - Change of `sig` from any value to 1 or from 0 to any value
  - `negedge sig`
    - Change of `sig` from any value to 0 or from 1 to any value
  - `sig`
    - Any change in `sig` value

Regular Event Control Examples

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

Named Event Control

- You can declare (name) an event, and then trigger and recognize it.
- Keyword: **`event`**

```
event calc_finished;
```

- Verilog symbol for triggering: **`->`**

```
->calc_finished
```

- Verilog symbol for recognizing: **`@()`**

```
@(calc_finished)
```

Named Event Control Examples

```
//This is an example of a data buffer storing data after the
//last packet of data has arrived.

event received_data; //Define an event called received_data

always @(posedge clock) //check at each positive clock edge
begin
        if(last_data_packet) //If this is the last data packet
                ->received_data; //trigger the event received_data
end

always @(received_data) //Await triggering of event received_data
                        //When event is triggered, store all four
                        //packets of received data in data buffer
                      //use concatenation operator { }
     data_buf = {data_pkt[0], data_pkt[1], data_pkt[2], data_pkt[3]};
```

Event OR control

- Used when need to trigger a block upon occurrence of any of a set of events.
- The list of the events: sensitivity list
- Keyword: **or**

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

Event OR control

- Simpler syntax
  - `always @( reset, clock, d)`

  - `@*` and `@(*)`

```
always @(a or b or c or d or e or f or g or h or
  p or m)

begin
out1 = a ? b+c : d+e;
out2 = f ? g+h : p+m;
end
```

Level-sensitive Timing Control

- Level-sensitive vs. event-based
  - event-based: wait for triggering of an event (change in signal value)
  - level-sensitive: wait for a certain condition (on values/levels of signals)
- Keyword: `wait()`

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

Sequential and Parallel Blocks

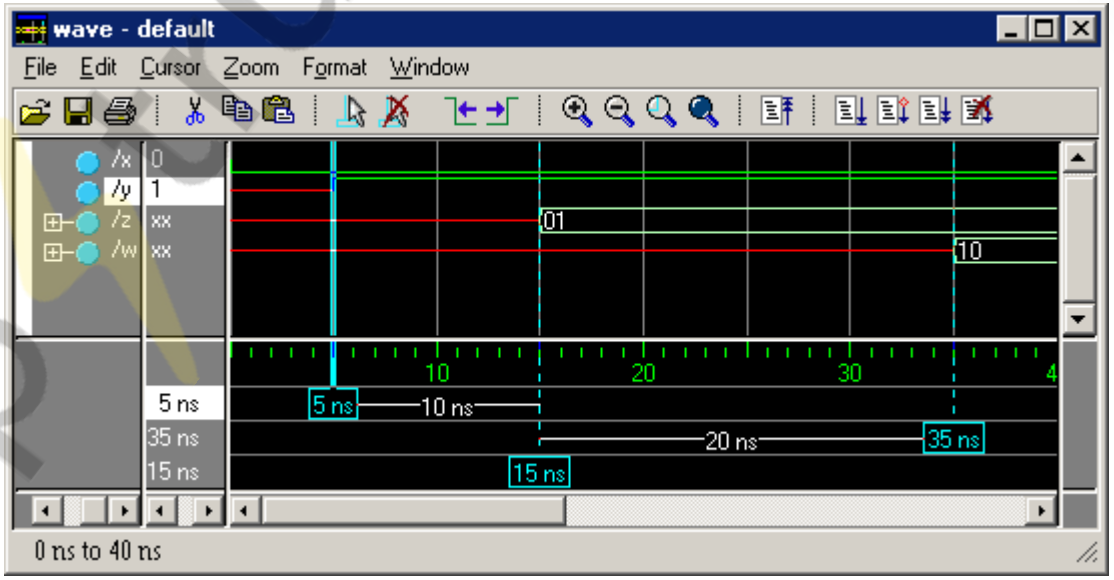
- Blocks: used to group multiple statements
- Sequential blocks
  - Keywords: `begin end`
  - Statements are processed in order.
  - A statement is executed only after its preceding one completes.
    - Exception: non-blocking assignments with intra-assignment delays
  - A delay or event is relative to the simulation time when the previous statement completed execution

Sequential and Parallel Blocks (cont'd)

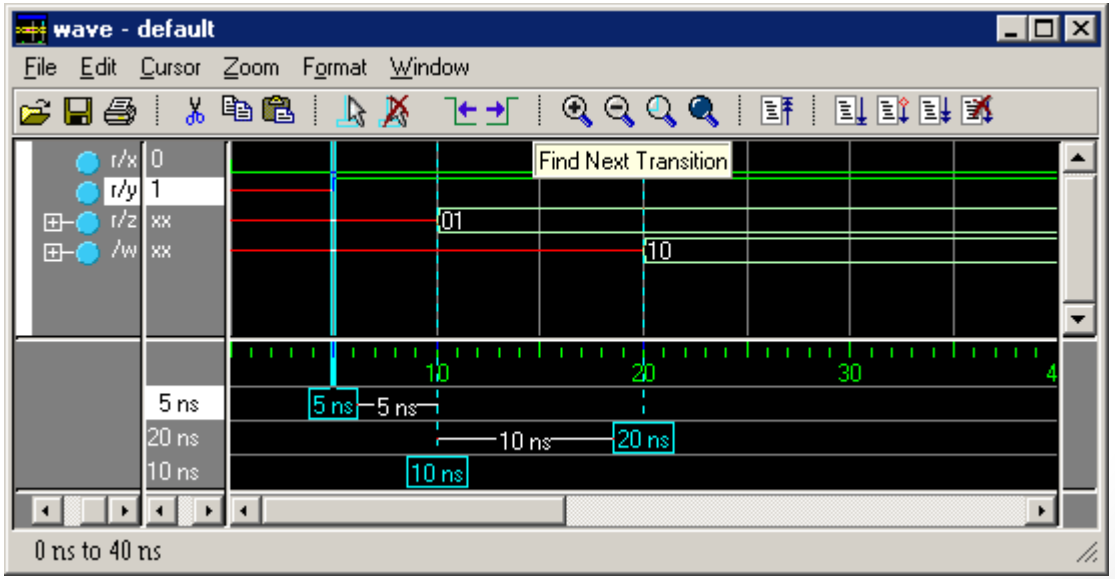
- Parallel Blocks
  - Keywords: `fork, join`
  - Statements in the blocks are executed concurrently
  - Timing controls specify the order of execution of the statements
  - All delays are relative to the time the block was entered
    - The written order of statements is not important
    - The `join` is done when all the parallel statements are finished

Sequential and Parallel Blocks (cont'd)

```
initial
begin
        x = 1'b0;
   #5   y = 1'b1;
   #10  z = {x,y};
   #20  w = {y,x};
end
```




```
initial
Fork
        x=1'b0;
   #5   y=1'b1;
   #10  z={x,y};
   #20  w={y,x};
join
```



Sequential and Parallel Blocks (cont'd)

- Parallel execution $\Rightarrow$ Race conditions may arise

```
initial
fork
        x=1'b0;
        y=1'b1;
        z={x,y};
        w={y,x};
join
```

- `z, w` can take either `2'b01, 2'b10` or `2'bxx, 2'bxx` depending on simulator

Special Features of Blocks

- Contents
  - Nested blocks
  - Named blocks
  - Disabling named blocks

Special Features of Blocks (cont'd)

- Nested blocks
  - Sequential and parallel blocks can be mixed

```
initial
begin
  x=1'b0;
  fork
       #5 y=1'b1;
       #10 z={x,y};
  join
  #20  w={y,x};
end
```

Special Features of Blocks (cont'd)

- Named blocks
  - Syntax:

```
begin: <the_name>          fork: <the_name>
  …                              …
end                        join
```

  - Advantages:
    - Can have local variables
    - Are part of the design hierarchy.
    - Their local variables can be accessed using hierarchical names
    - Can be disabled

Special Features of Blocks (cont'd)

```
module top;
   initial
   begin : block1
     integer i;   //hiera  name: top.block1.
   …                .                 i
   end

   initial
   fork : block2
   reg i;    //hierarchica name: top.block2.
   …         l                       i
   join
endmodule
```

Special Features of Blocks (cont'd)

- Disabling named blocks
  - Keyword: `disable`
  - Action:
    - Similar to `break` in C/C++, but can disable any named block not just the inner-most block.

Special Features of Blocks (cont'd)

```
module find_true_bit;

reg [15:0] flag;
integer i;

initial
begin
   flag = 16'b
    0010_0000_0000_0000;
   i = 0;
   begin: block1
   while(i < 16)
   begin
      if (flag[i])
      begin
       $display("Encountered a
            TRUE bit at element
            number %d", i);
       disable block1;
      end // if
       i = i + 1;
       end // while
   end // block1
   end //initial
   endmodule
```

4-to-1 Multiplexer

```
// 4-to-1 multiplexer. Port list is taken exactly from
// the I/O diagram.
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);

  // Port declarations from the I/O diagram
  output out;
  input i0, i1, i2, i3;
  input s1, s0;
  reg out; //output declared as register

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

4-bit Counter

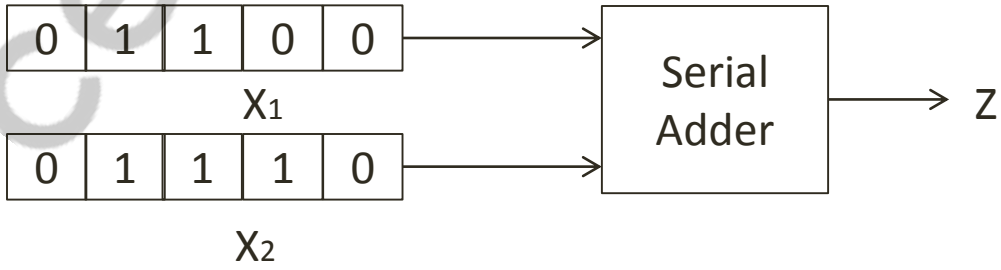
```
//Binary  counter
module counter(Q , clock, clear);

  // I/O ports
  output [3:0] Q;
  input clock, clear;
  //output defined as register
  reg [3:0] Q;

  always @( posedge clear  or negedge clock)
  begin
    if (clear)
      Q = 4'd0;
    else
      //Q = (Q + 1) % 16;
      Q = (Q + 1) ;
  end

endmodule
```

Verilog® HDL

Finite State Machines (FSM)

Sequential circuits

- A finite state machine is an abstract model describing the synchronous sequential machine and is spatial counterpart, the iterative network. FSM is the basis for understanding and development of the various computational structures both time dependent and state transition dependent.



Above circuit is the serial binary adder which is a synchronous circuit with two inputs, X1 and X2, carrying the two binary number to be added and one output Z.

State table

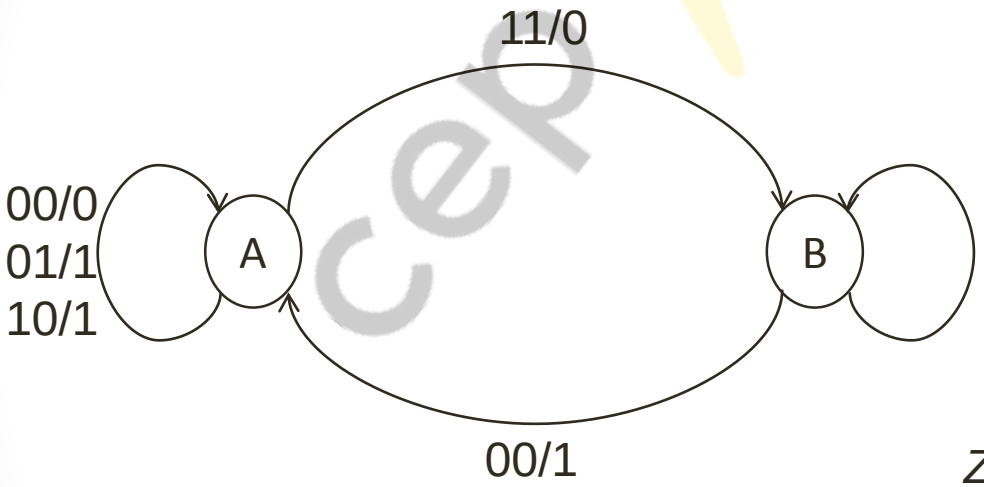
- Serial binary adder shown in the figure
- It is a synchronous circuit
- X1 and X2 carrying the two binary numbers to be added and one output z
- Addition table of X1 and X2
- State table of the previous circuit NS is next state and PS is previous state

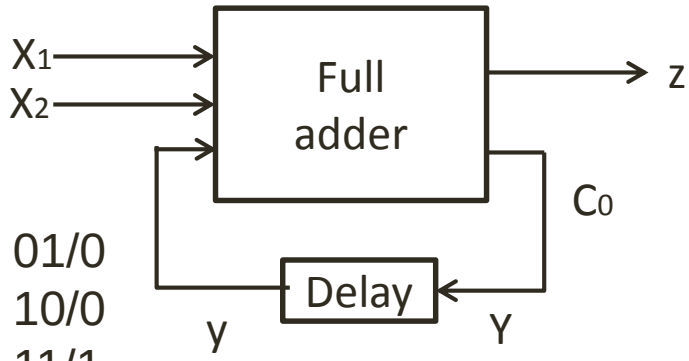
	t_5	t_4	t_3	t_2	t_1		
	0	1	1	0	0	=	X_1
+	0	1	1	1	0	=	X_2
	1	1	0	1	0	=	Z

	NS,z			
PS	X_1X_2=00	01	11	10
A	A,0	A,1	B,0	A,1
B	A,1	B,0	B,1	B,0

State assignment

- Delay element using D flip flop
- Length of delay is usually equal to the interval between two successive clock pluses
- Y(t)=y(t+1)





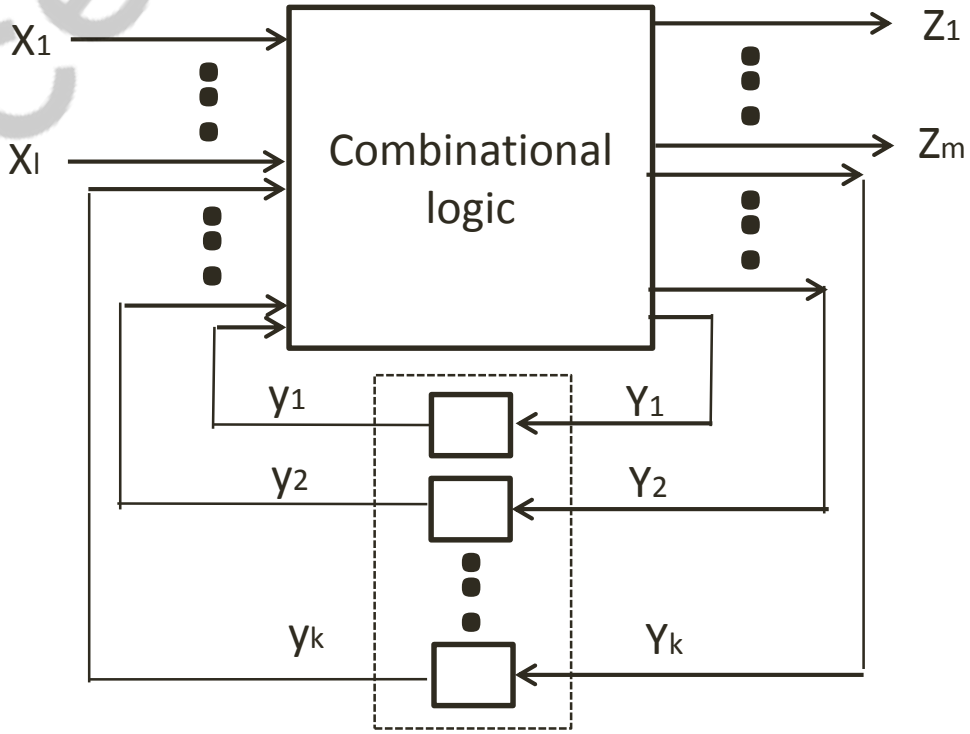
$$Y=x_1x_2+x_1y+x_2y$$

$$Z=x_1'x_2'y+x_1'x_2y'+x_1x_2'y'+x_1x_2y$$

	Next state Y				Output z			
	x_1x_2				x_1x_2			
y	00	01	11	10	00	01	11	10
0	0	0	1	0	0	1	0	1
1	0	1	1	1	1	0	1	0

The Finite State machine- basic definition

The behavior of a finite-sate machine is described as a sequence f events that occur at discrete instants designated t=1,2,3 etc. suppose that a machine M has been receiving input signals and has been responding by producing output signals. If now at time t, we were to apply an input signal x(t) to M then its response z(t) would depend on x(t) as well as on the past input signals to M. also since a given machine M might have an infinite variety of possible histories, it would need an infinite capacity for storing them.



Synchronous Sequential Machine

Synchronous sequential machine is represented schematic by the circuit in the previous slide.

- The circuit has a finite number l of input terminals.
- The signals entering the circuit via these terminals continue a set $(x_1,x_2,\ldots,x_l\}$ of input variables, where each x_j, for all j, may take on the 2 possible values 0 or 1.
- An ordered l-tuple of 0's and 1's is an input configuration (alternatively, input symbols, pattern, or vector).
- The set of p=2^l distinct input patterns is called the input alphabet, and each configuration is referred t as a symbol of the alphabet. Thus the input alphabet is given by

  $I= \{I_1, I_2,\ldots., I_p\}$.
- An ordered m-tuple output configuration set of q=2^m ordered m-tuple is called output alphabet and given by

  $O= \{O_1, O_2, \ldots, O_q\}$
- The signal value at the output of each memory element is referred to as the state variable, and $\{y_1,y_2,\ldots,y_k\}$ constitutes the set of state variables. The set S of n=2^k k-tuples constitutes the entire set od state of the machine, where

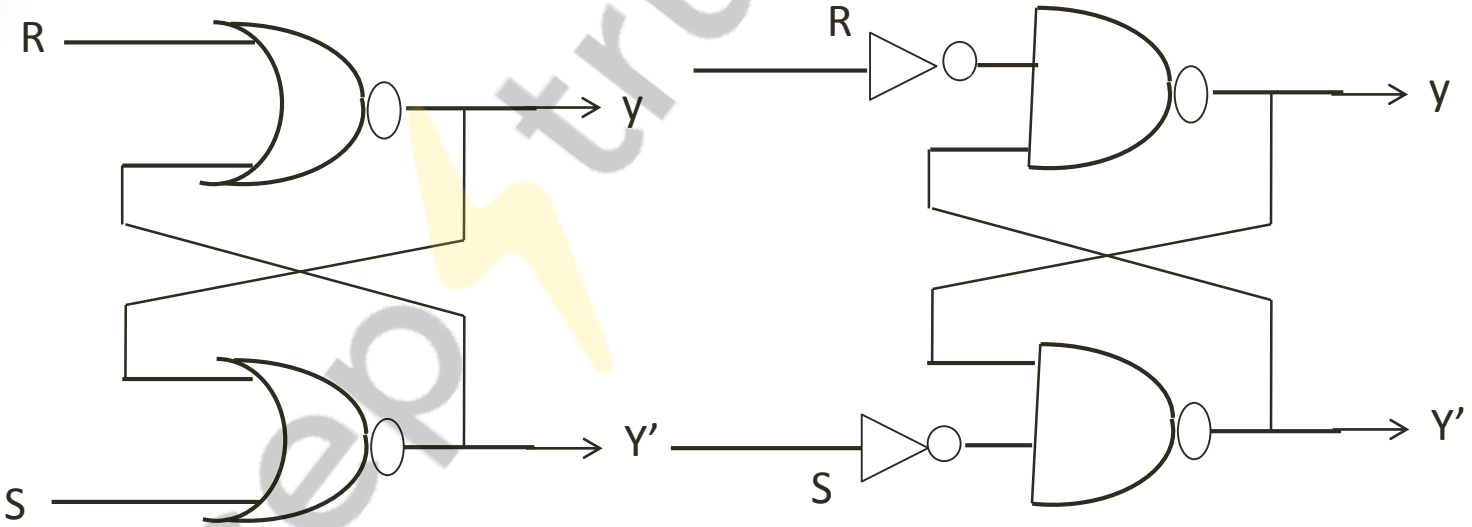
  $S= \{S_1,S_2,\ldots.,S_n\}$
- Synchronization is achieved by means of clock pluses feeding the memory devices

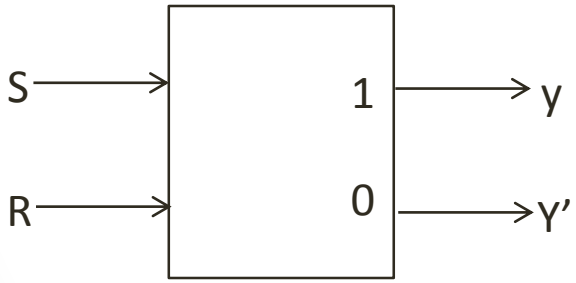
Memory elements and excitations

In order to design a circuit that operates according to the specifications of a given table, it is necessary first to select a number of memory elements, each of which is device with two distinct states and is capable of storing a binary bit.

- SR latch
- T latch
- D latch
- JK latch
- Clock timing and master-slave flip flop
- Edge triggered flip flop

S-R latch



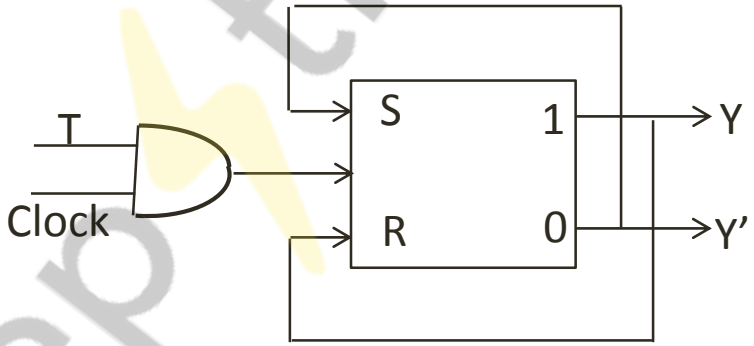


Y(t+1)= R'y(t) + S

Excitation table of S-R Flip-flop

Change in y		Required Value	
From	To	S	R
0	0	0	-
0	1	1	0
1	0	0	1
1	1	-	0

T latch

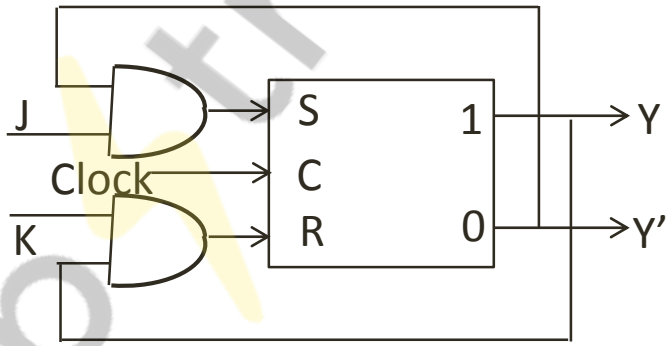


Excitation table of T Flip-flop

Change in y		Required Value
From	To	Y
0	0	0
0	1	1
1	0	1
1	1	0

$$Y(t+1)= T \oplus y(t)$$

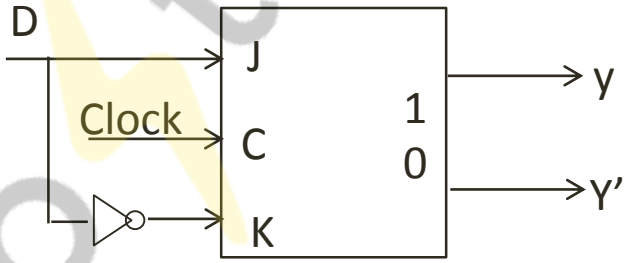
J-K latch



Excitation table of J-K Flip-flop

Change in y		Required Value	
From	To	J	K
0	0	0	-
0	1	1	-
1	0	-	1
1	1	-	0

D latch



Excitation table of D Flip-flop

Change in y		Required Value
From	To	Y
0	0	0
0	1	1
1	0	0
1	1	0

Y(t+1)= D(t)

Combinational Logic Modeling

- Gate level design

```
and a0(out, in1, in0);
```

- Dataflow design

```
assign out = in1 & in0;
```

- Behavioral level design

```
always @(in1 or in0)
  out = in1 & in0;
```

Sequential Logic Modeling

- Gate level design
  - Connect gates and flip-flops
- Dataflow design
  - Use `assign` for combinational components
- Behavioral design
  - Alternative 1: Use combinational logic style
  - Alternative 2:

```
always @(posedge clk)
  q <= d;
```

FSM: Finite State Machines

- Definitions
  - State bits
  - Output logic
  - Next-state logic
- Types
  - Moore FSM
  - Mealy FSM

Main steps for designing sequential circuits areas below:

1. From a word description of the problem, from a state table (or a state diagram) that specifies the circuit behavior.
2. Check this table to determine whether it contains any redundant states.
3. Select a state assignment and determine the type of memory elements to be used
4. Derive the transition and output tables.
5. Derive an excitation table and obtain the excitation and output functions from their respective tables.
6. Draw a circuit diagram

In effect, in step 5 we are converting a less familiar problem that of sequential circuit synthesis, into a more familiar problem, that of combinational circuit synthesis, since the construction of the excitation table is actually equivalent to the construction of a set of maps, from which the derivation of the excitation functions is straight forward.

General Moore FSM

```
module moore_fsm (ins, outs, clk);
  input clk;
  input ins;
  output reg outs;
  reg […] state = 0, next = 0;
  always @(posedge/negedge clk)
     // change the state
  always @ (state or ins)
     // evaluate next state
  always @ (state)
     // evaluate output
endmodule
```

General Mealy FSM

```
module mealy_fsm (ins, outs, clk);
   input clk;
   input ins;
   output reg outs;
   reg […] state = 0, next = 0;
   always @(posedge/negedge clk)
      // change the state
   always @ (state or ins)
      // evaluate next state
   always @ (state or ins)
      // evaluate output
endmodule
```

Example: 011 Detector

- 1 input: i
- 1 output: f

- Mealy: 3 states
- Moore: 4 states

Moore 011 Detector

```
module moore_011 (i, f, clk);
    input i, clk; output f;
    reg [2:1] state = 0, next_st=0; reg f;
    always @(posedge clk)
        state <= next_st;
    always @(state or i) begin
        // change the state and output->next slide
    end
    always @(state)
        if (state == 2'b11) f <= 1; else f <= 0;
endmodule
```

Moore 011 Detector (cont'd)

```
if (state == 0) next_st = i ? 0 : 1;
 else if (state == 1) next_st = i ? 2 : 1;
 else if (state == 2) next_st = i ? 3 : 1;
 else if (state == 3) next_st = i ? 0 : 1;
```

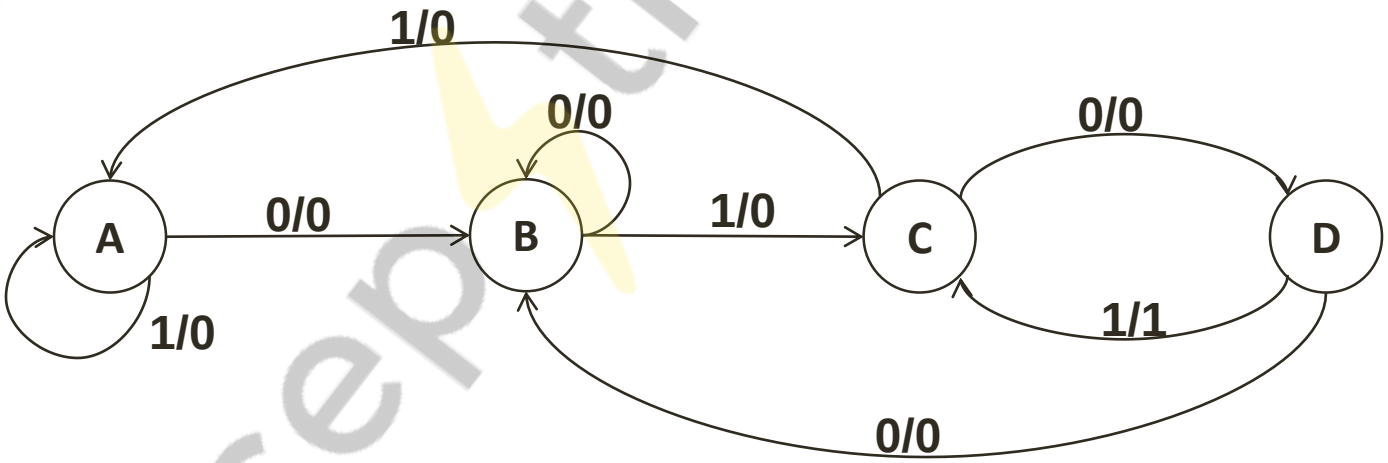
Mealy 011 Detector

```
module mealy_011 (i, f, clk);
   input i, clk; output reg f;
   reg [2:1] state = 0, next_st = 0;
   always @(posedge clk)
        state <= next_st;
   always @ (state or i)
        if (state == 3 && i) f <= 1;
        else f <= 0;
```

Mealy 011 Detector (cont.)

```
always @(state or i)
   case (state)
        2'b00 : next_st <= i ? 0 : 1;
        2'b01 : next_st <= i ? 2 : 1;
        2'b10 : next_st <= i ? 0 : 1;
        default: next_st <= 0;
   endcase
endmodule
```

The Sequence detector



	NS,z	
PS	X=0	X=1
A	B,0	A,0
B	B,0	C,0
C	D,0	A,0
D	B,0	C,1

The Sequence detector cont.

		Y1Y2		z	
	y1y2	X=0	X=1	X=0	X=1
A	00	01	00	0	0
B	01	01	11	0	0
C	11	10	00	0	0
D	10	01	11	0	1

y1y2 \ X	0	1
00		
01		
11		
10		1

y1y2 \ X	0	1
00		
01		1
11	1	
10		1

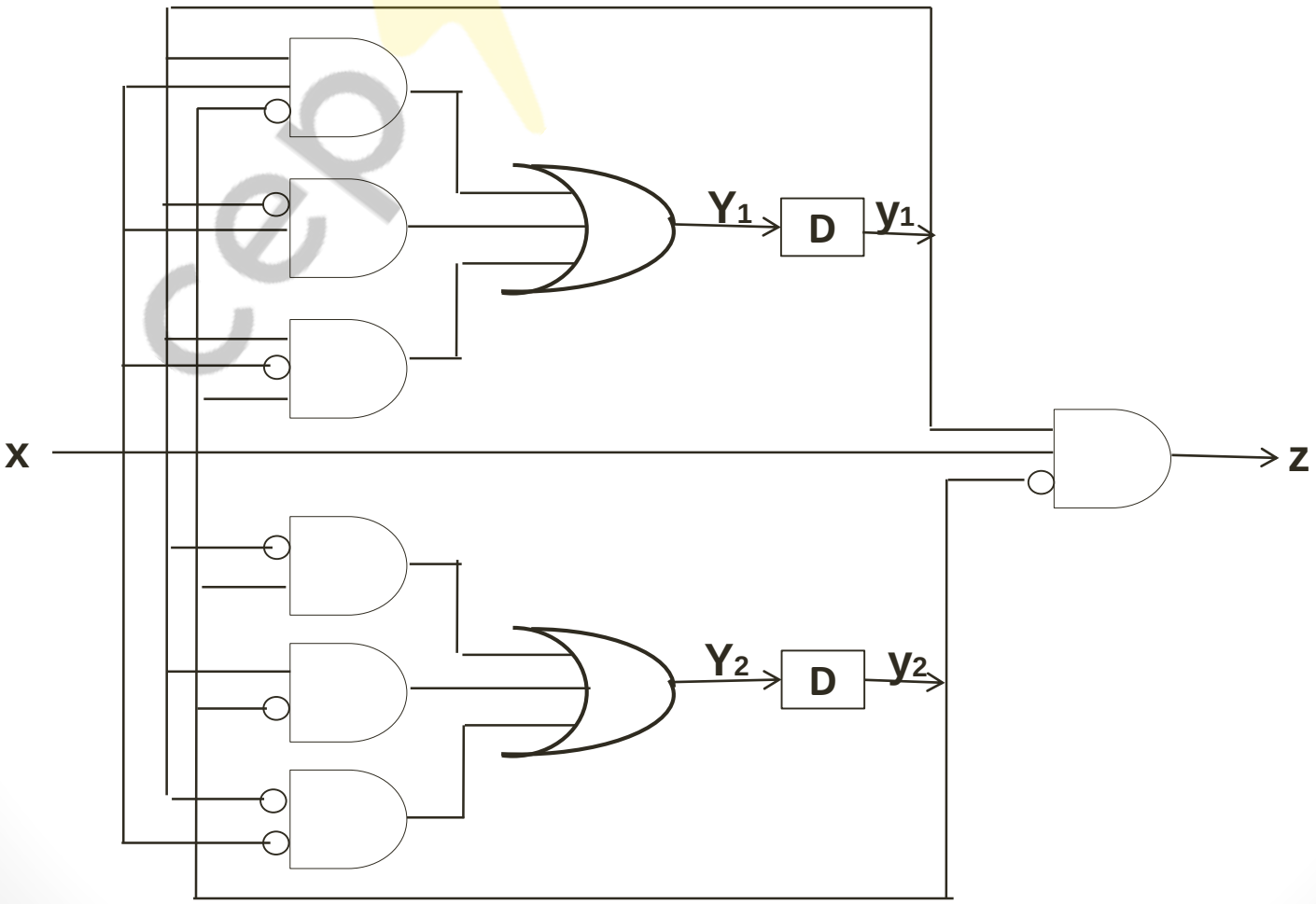
y1y2 \ X	0	1
00	1	
01	1	1
11		
10	1	1

The Sequence detector cont.

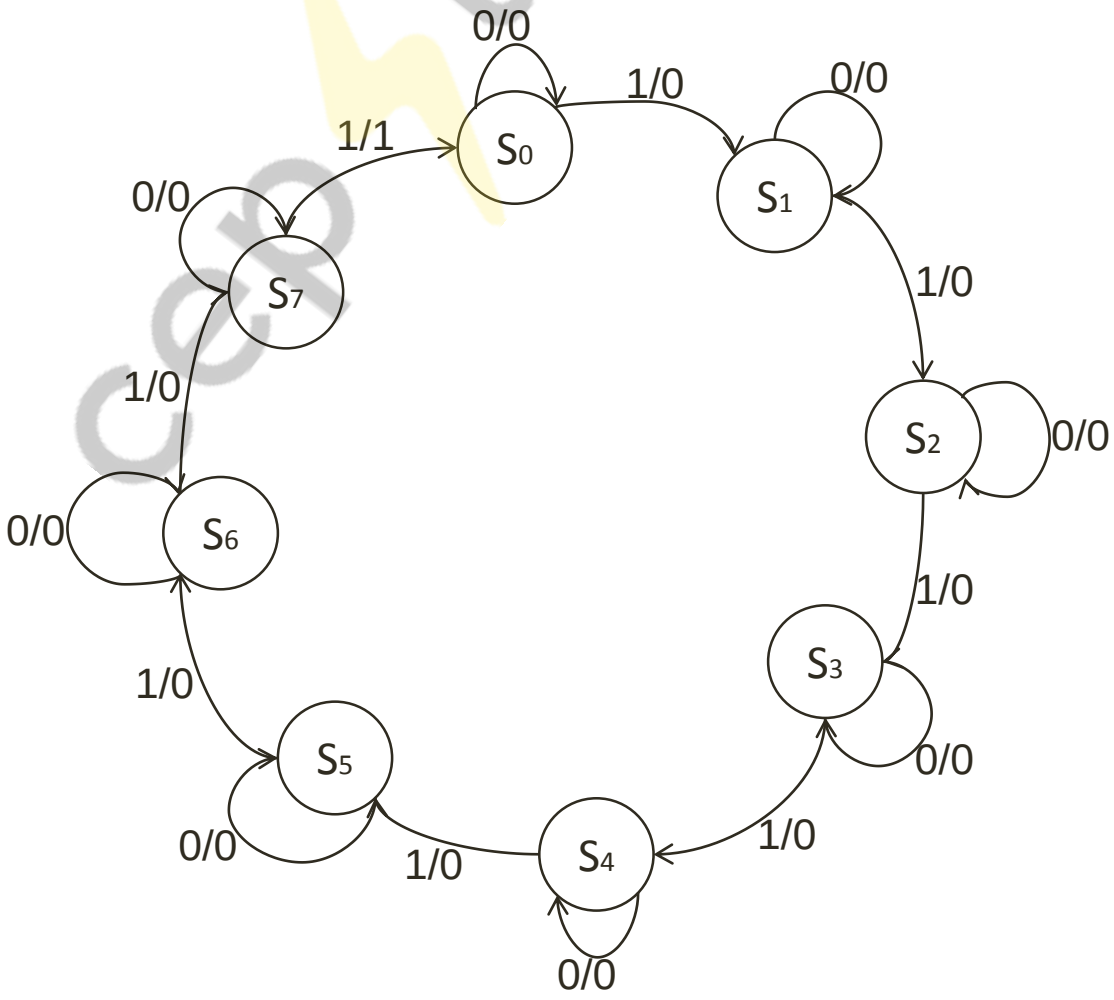
$Z = xy_1y_2'$;

$Y_1 = x'y_1y_2 + xy_1'y_2 + xy_1y_2'$;

$Y_2 = y_1y_2' + x'y_1' + y_1'y_2$;



The Binary Counter



The Binary Counter cont.

PS	NS X=0	NS X=1	Output X=0	Output X=1
S_0	S_0	S_1	0	0
S_1	S_1	S_2	0	0
S_2	S_2	S_3	0	0
S_3	S_3	S_4	0	0
S_4	S_4	S_5	0	0
S_5	S_5	S_6	0	0
S_6	S_6	S_7	0	0
S_7	S_7	S_0	0	1

State table of counter

PS $Y_3Y_2Y_1$	NS X=0	NS X=1	Output X=0	Output X=1
000	000	001	0	0
001	001	010	0	0
010	010	011	0	0
011	011	100	0	0
100	100	101	0	0
101	101	110	0	0
110	110	111	0	0
111	111	000	0	1

Transition and output table of counter

The Binary Counter cont.

	$T_1T_2T_3$	
$Y_3Y_2Y_1$	X=0	X=1
000	000	001
001	001	011
010	010	001
011	011	111
100	100	001
101	101	011
110	110	001
111	111	111

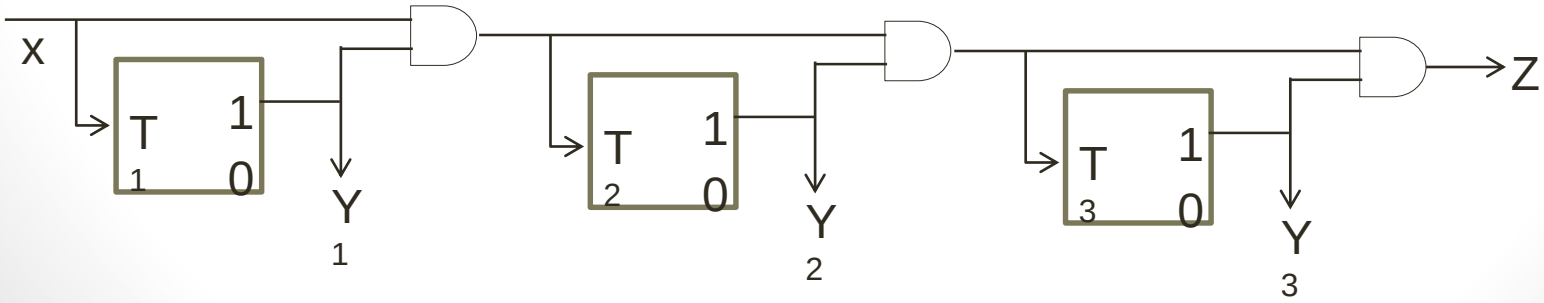
Using T flip-flop

$T_1=X;$

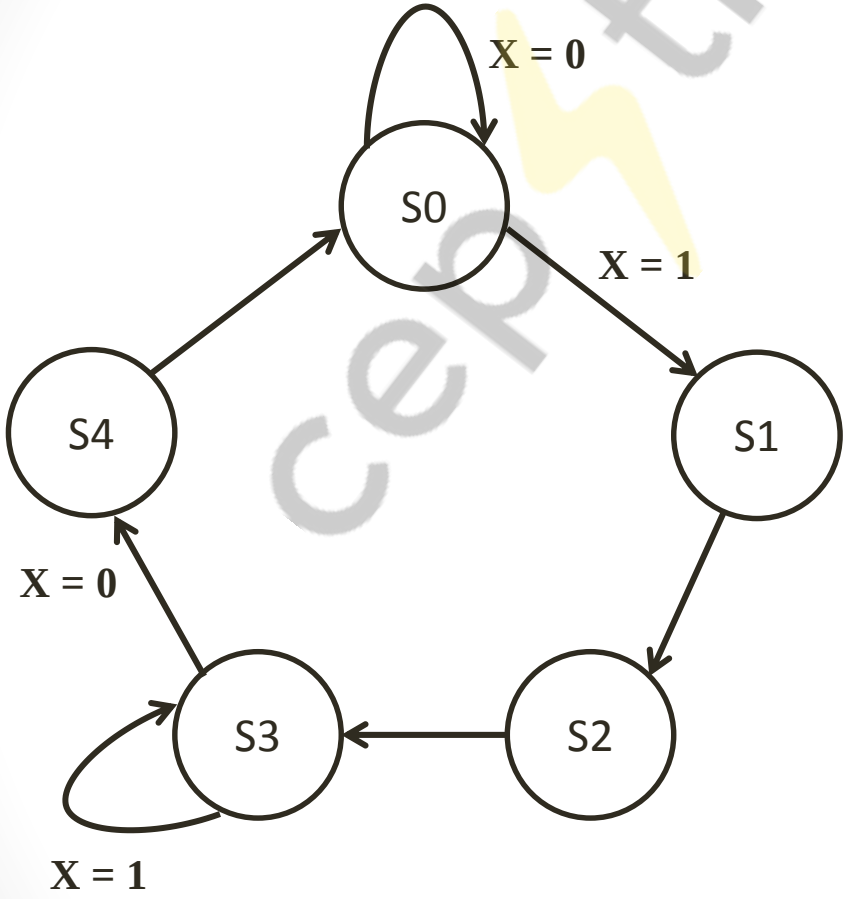
$T_2=XY_1;$

$T_3=XY_1Y_2;$

$Z=XY_1Y_2Y_3;$

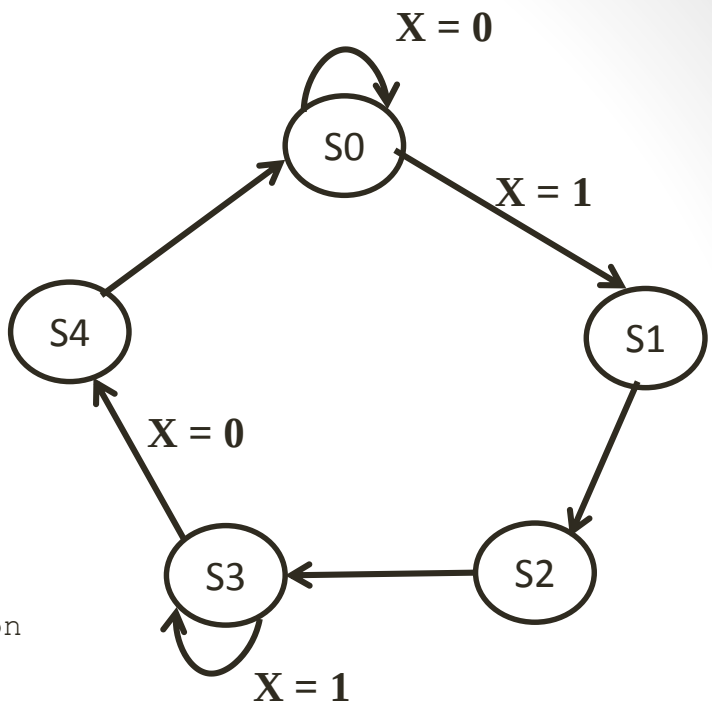


Traffic Signal Controller



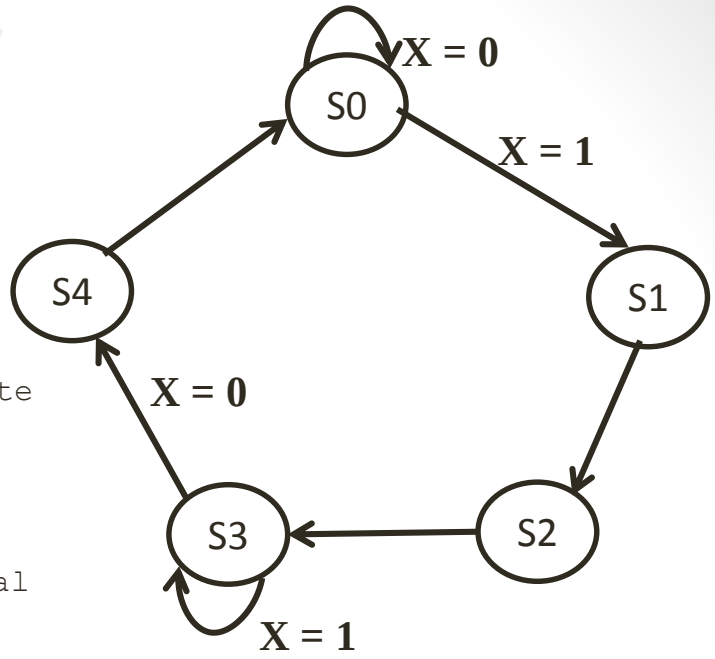
State	Signals
S0	Hwy = G, Cntry = R
S1	Hwy = Y, Cntry = R
S2	Hwy = R, Cntry = R
S3	Hwy = R, Cntry = G
S4	Hwy = R, Cntry = Y

```
`define TRUE  1'b1
`define FALSE 1'b0

`define Y2RDELAY  3 //Yellow to red delay
`define R2GDELAY  2 //Red to green delay

module sig_control(hwy, cntry, X, clock, clear);

output [1:0] hwy, cntry;
      //2-bit output for 3 states of signal
      //GREEN, YELLOW, RED;
reg [1:0] hwy, cntry;
      //declared output signals are registers

input X; //if TRUE, indicates that there is car on
         //the country road, otherwise FALSE

input clock, clear;

parameter RED = 2'd0,
          YELLOW = 2'd1,
          GREEN = 2'd2;

//State definition     HWY          CNTRY
parameter S0 = 3'd0, //GREEN         RED
          S1 = 3'd1, //YELLOW        RED
          S2 = 3'd2, //RED           RED
          S3 = 3'd3, //RED           GREEN
          S4 = 3'd4; //RED           YELLOW
```



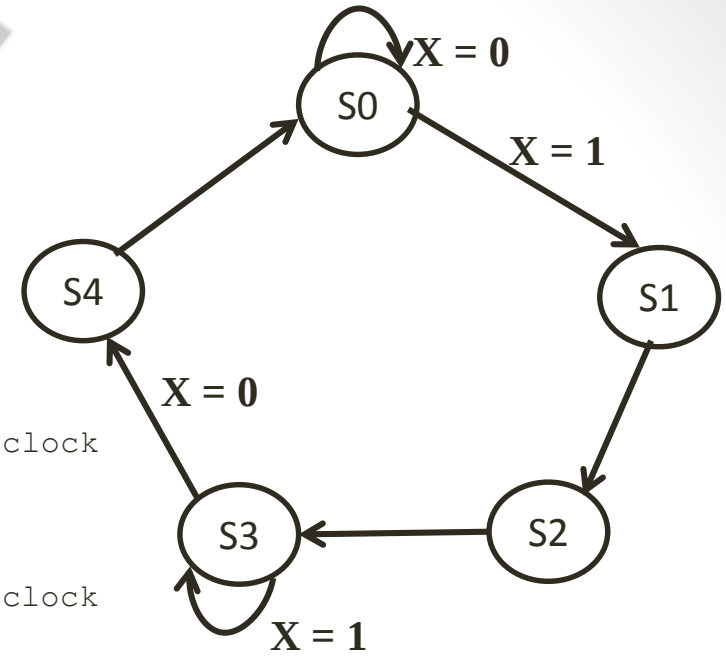
State	Signals
S0	Hwy = G, Cntry = R
S1	Hwy = Y, Cntry = R
S2	Hwy = R, Cntry = R
S3	Hwy = R, Cntry = G
S4	Hwy = R, Cntry = Y

```
//Internal state variables
reg [2:0] state;
reg [2:0] next_state;

//state changes only at positive edge of clock
always @(posedge clock)
  if (clear)
      state <= S0; //Controller starts in S0 state
  else
      state <= next_state; //State change


//Compute values of main signal and country signal
always @(state)
begin
  hwy = GREEN; //Default Light Assignment for Highway light
  cntry = RED; //Default Light Assignment for Country light
  case(state)
     S0: ; // No change, use default
     S1: hwy = YELLOW;
     S2: hwy = RED;
     S3:  begin
            hwy = RED;
            cntry = GREEN;
          end
     S4:  begin
            hwy = RED;
            cntry = `YELLOW;
          end
  endcase
end
```



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

```
//State machine using case statements
always @(state or  X)
begin
    case (state)
       S0: if(X)
            next_state = S1;
          else
            next_state = S0;
       S1: begin //delay some positive edges of clock
            repeat(`Y2RDELAY) @(posedge clock) ;
            next_state = S2;
          end
       S2: begin //delay some positive edges of clock
            repeat(`R2GDELAY) @(posedge clock);
            next_state = S3;
          end
       S3: if(X)
            next_state = S3;
          else
            next_state = S4;
       S4: begin //delay some positive edges of clock
            repeat(`Y2RDELAY) @(posedge clock) ;
            next_state = S0;
          end
      default: next_state = S0;
    endcase
end


endmodule
```



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

Verilog® HDL

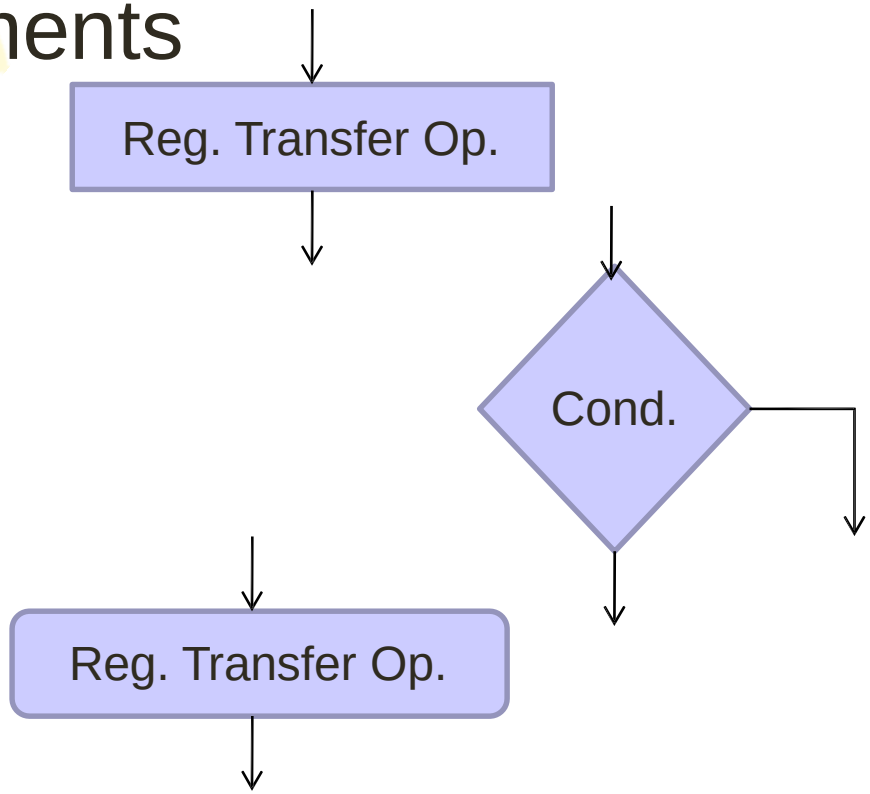
Algorithmic State Machines (ASM)Chart

Digital System Components

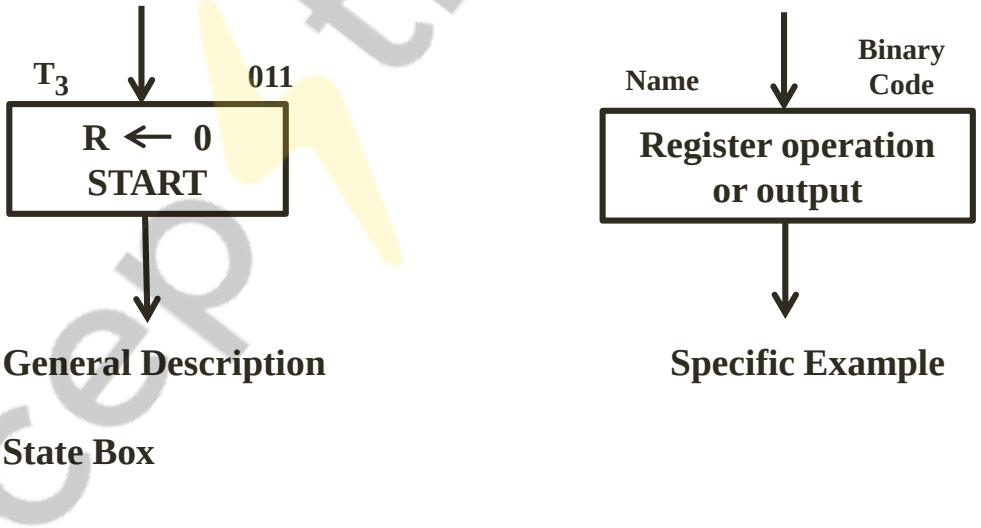
- Datapath (Data Processor)
  - Performs computations
- Control part
  - Controls the sequence of operations
- FSM-D model
  - Finite-State Machine + Datapath
  - Reference
    - M. Mano, Digital Design, 3rd Edition, Chapter 8: Algorithmic state machines

ASM Chart

- One systematic way to design complex digital systems
- ASM Chart Elements
  - State box
  - Decision box
  - Conditional box



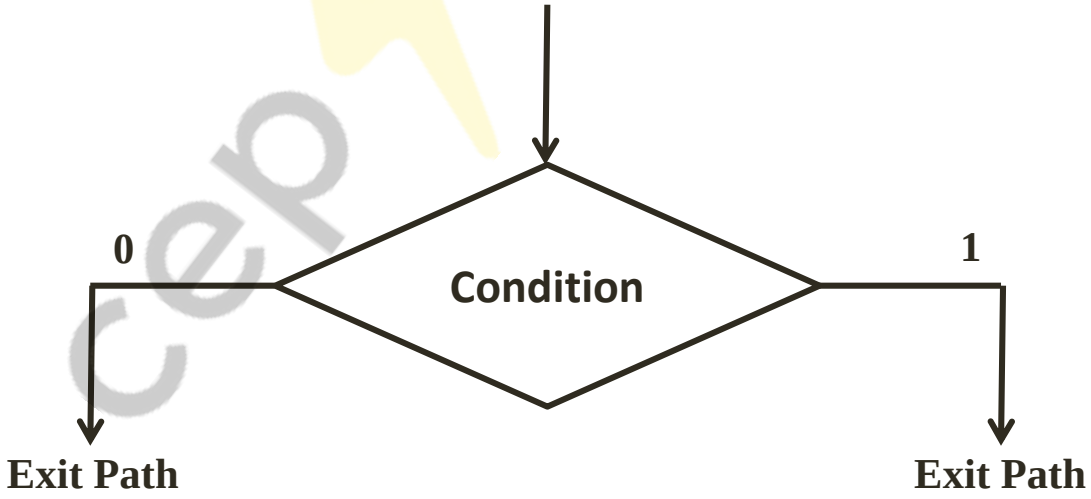
State Box



Symbolic Notation for Register Operations

Symbolic Notation	Description
A ← B	Transfer contents of register B into register A.
R ← 0	Clear register R.
F ← 1	Set flip-flop F to 1.
A ← A + 1	Increment register A by 1 (count-up).
A ← A – 1	Decrement register A by 1 (count-down).
A ← A + B	Add contents of register B to register A.

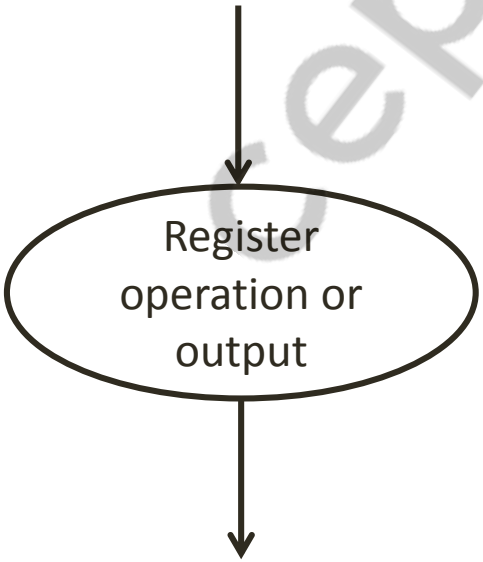
Decision Box



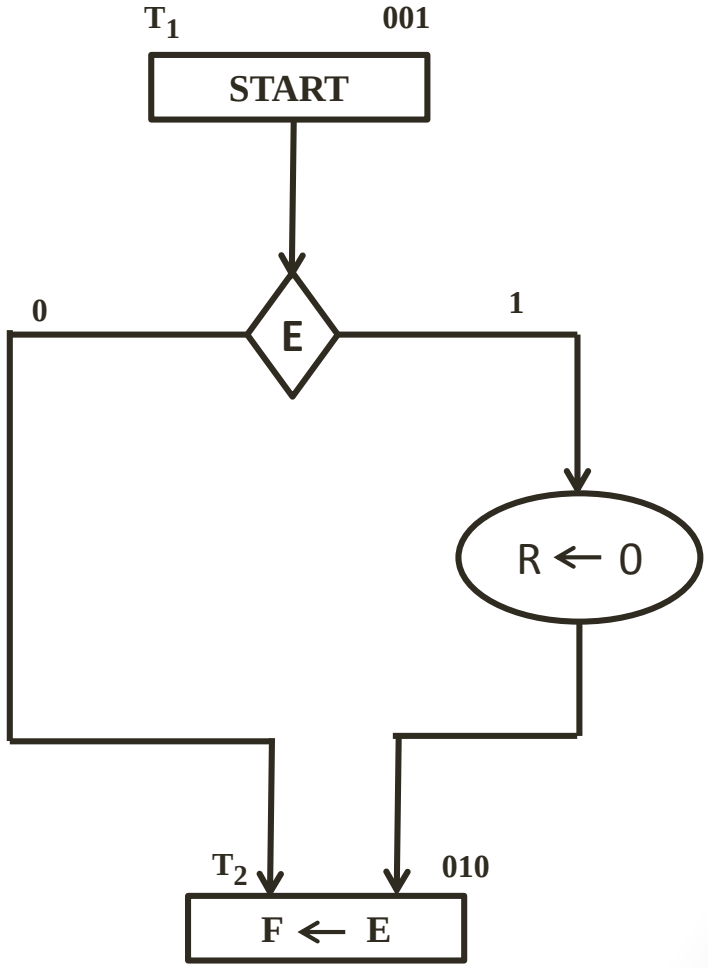
Decision Box

Conditional Box

From exit path of decision box

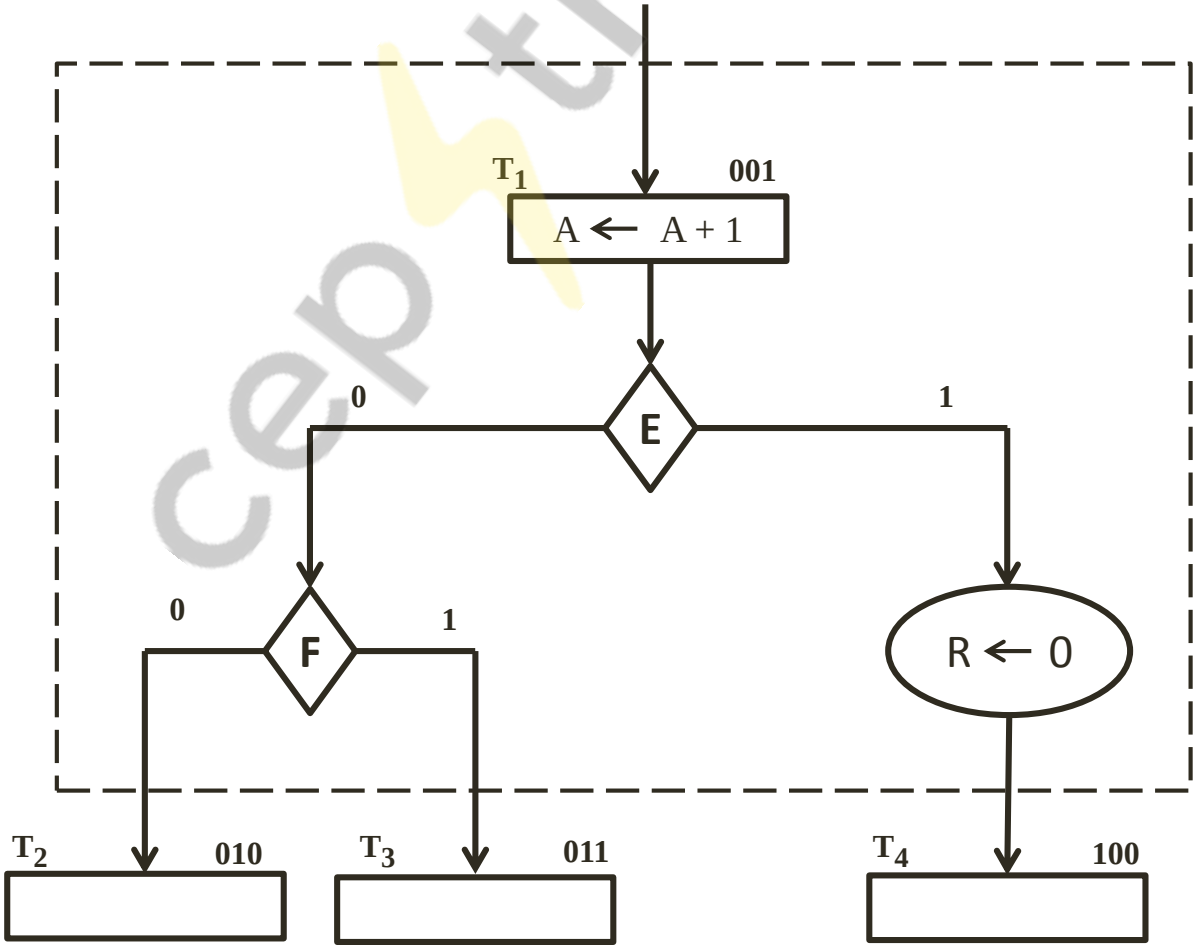


Conditional Box



Example with Conditional Box

ASM Block



ASM Block

Timing Consideration

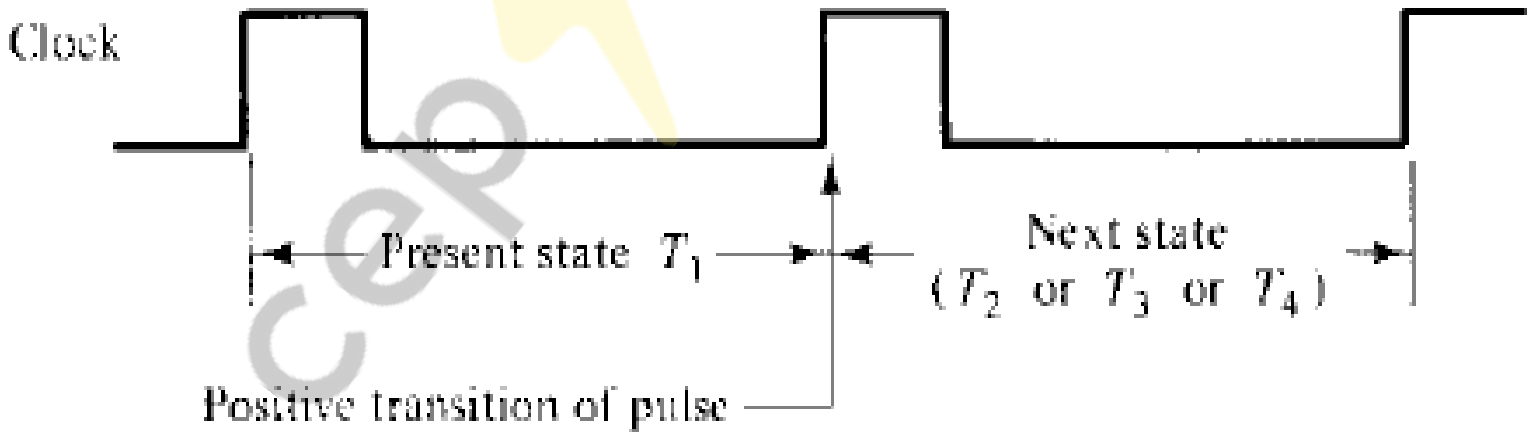
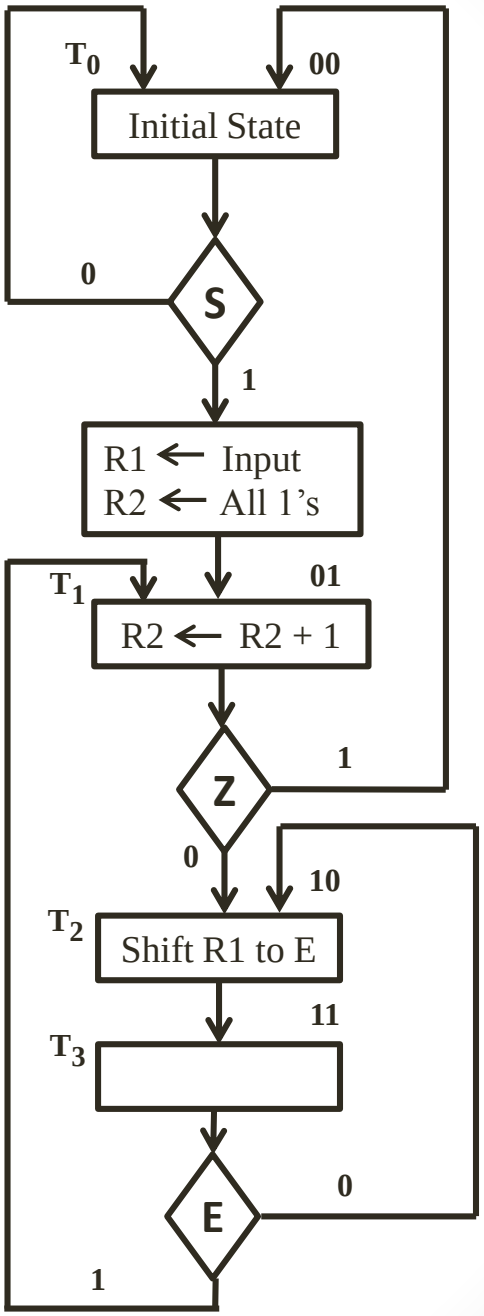


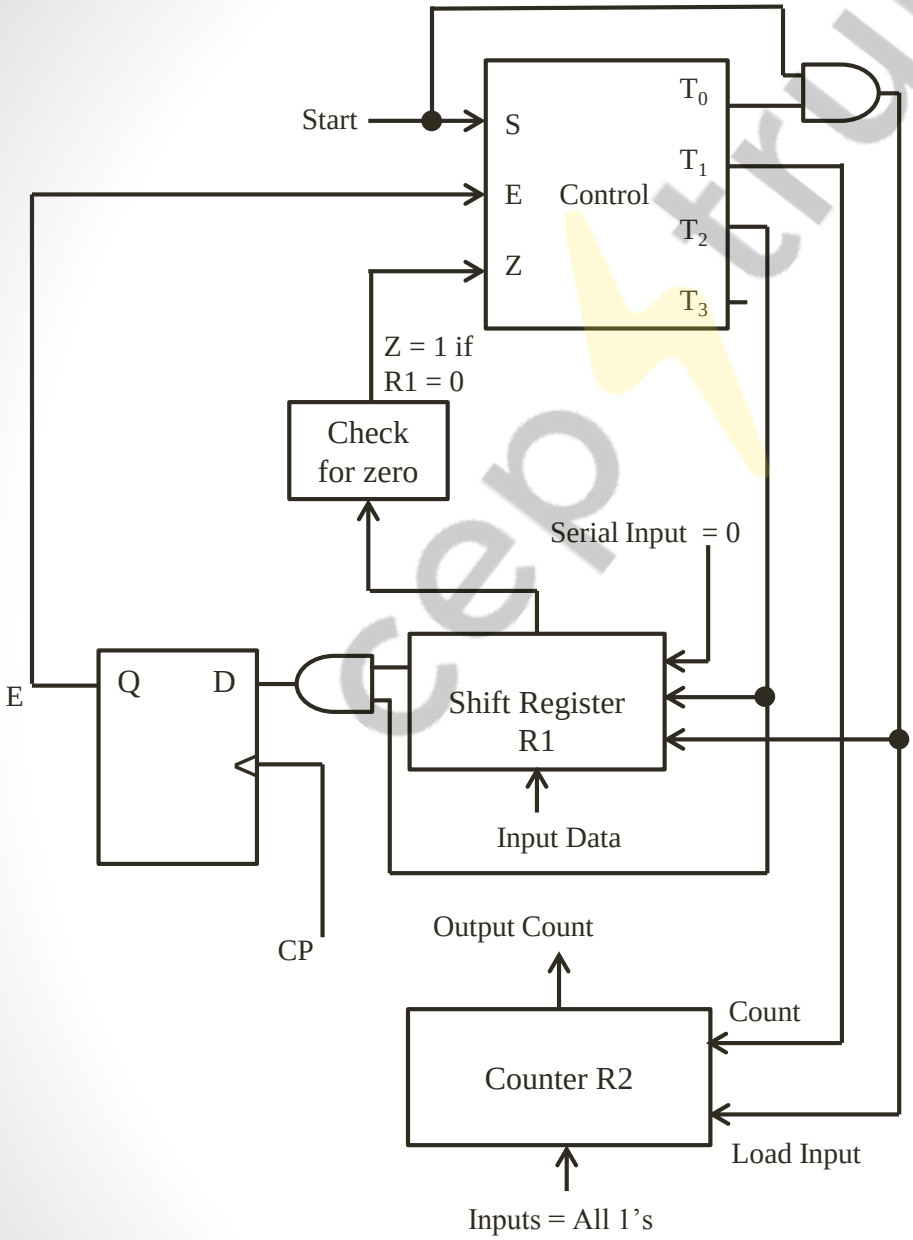
FIGURE 8-8

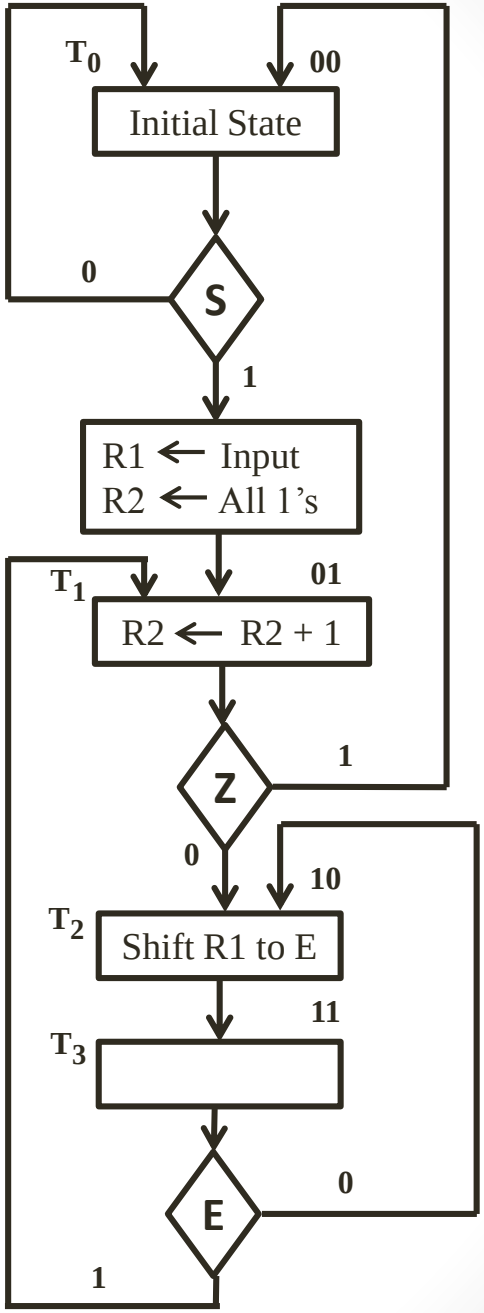
Transition between states

Design Example

- One's counter
  - input in R1: **1010110**
    => output in R2: **4**







Data processor subsystem for Design Example

Verilog® HDL

Verilog Tasks and Functions

Introduction

- Procedures/Subroutines/Functions in SW programming languages
  - The same functionality, in different places
- Verilog equivalence:
  - Tasks and Functions
  - Used in behavioral modeling
  - Part of design hierarchy $\Rightarrow$ Hierarchical name

Functions

- Keyword: `function, endfunction`
- Can be used if the procedure
  - does not have any timing control constructs
  - returns exactly a single value
  - has at least one input argument

Functions (cont'd)

- Function Declaration and Invocation
  - Declaration syntax:

```
function <range_or_type> <func_name>;
  <input declaration(s)>
  <variable_declaration(s)>
  begin // if more than one statement needed
    <statements>
  end // if begin used
endfunction
```

Functions (cont'd)

- Function Declaration and Invocation
  - Invocation syntax:

```
<func_name> (<argument(s)>);
```

Functions (cont'd)

- Semantics
  - much like `function` in Pascal
  - An internal implicit `reg` is declared inside the function with the same name
  - The return value is specified by setting that implicit `reg`
  - `<range_or_type>` defines width and type of the implicit `reg`
    - `<type>` can be `integer` or `real`
    - default bit width is 1

Function Examples
Parity Generator

```
module parity;
reg [31:0] addr;
reg parity;

initial begin
   …
end

always @(addr)
begin
   parity = calc_parity(addr);
   $display("Parity calculated = %b",
        calc_parity(addr) );
end
```

```
function calc_parity;
input [31:0] address;
begin
   calc_parity = ^address;
end
endfunction

endmodule
```

Function Examples
Controllable Shifter

```
module shifter;
`define LEFT_SHIFT      1'b0
`define RIGHT_SHIFT     1'b1
reg [31:0] addr, left_addr,
   right_addr;
reg control;

initial
begin
   …
end

always @(addr)
begin
  left_addr  =shift(addr, `LEFT_SHIFT);
  right_addr =shift(addr,`RIGHT_SHIFT);
end
```

```
function [31:0] shift;
input [31:0] address;
input control;
begin
  shift = (control==`LEFT_SHIFT)
     ?(address<<1) :(address>>1);
 end
 endfunction

    endmodule
```

Tasks

- Keywords: `task, endtask`
- Must be used if the procedure has
  - any timing control constructs
  - zero or more than one output arguments
  - no input arguments

Tasks (cont'd)

- Task declaration and invocation
  - Declaration syntax

```
task <task_name>;
    <I/O declarations>
    <variable and event declarations>
    begin // if more than one statement needed
      <statement(s)>
    end   // if begin used!
endtask
```

Tasks (cont'd)

- Task declaration and invocation
  - Task invocation syntax

```
<task_name>;
<task_name> (<arguments>);
```

  - `input` and `inout` arguments are passed into the task
  - `output` and `inout` arguments are passed back to the invoking statement when task is completed

Tasks (cont'd)

- I/O declaration in modules vs. tasks
  - Both used keywords: `input, output, inout`
  - In modules, represent ports
    - connect to external signals
  - In tasks, represent arguments
    - pass values to and from the task

Task Examples
Use of input and output arguments

```
module operation;
parameter delay = 10;
reg [15:0] A, B;
reg [15:0] AB_AND, AB_OR, AB_XOR;

initial
   $monitor( …);

initial
begin
   …
end

always @(A or B)
begin
   bitwise_oper(AB_AND, AB_OR,
        AB_XOR, A, B);
end
```

```
task 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


endmodule
```

Task Examples
Use of module local variables

```
module sequence;
reg clock;

initial
begin
   …
end

initial
   init_sequence;

always
   asymmetric_sequence;
```

```
task init_sequence;
        clock = 1'b0;
endtask

task asymmetric_sequence;
begin
        #12 clock = 1'b0;
        #5  clock = 1'b1;
        #3  clock = 1'b0;
        #10 clock = 1'b1;
end
endtask

endmodule
```

Automatic Tasks and Functions

- Used for re-entrant code
- Keyword
  - `automatic`
- Example

```
function automatic integer factorial;
task automatic bitwise_xor;
```

Differences between...

- Functions
  - Can enable (call) just another function (not task)
  - Execute in 0 simulation time
  - No timing control statements allowed
  - At lease one input
  - Return only a single value
- Tasks
  - Can enable other tasks and functions
  - May execute in non-zero simulation time
  - May contain any timing control statements
  - May have arbitrary `input`, `output`, or `inout`
  - Do not return any value

Similarities between Tasks & Functions

- Both
  - are defined in a `module`
  - are local to the `module`
  - can have local variables (registers, but not nets) and events
  - contain only behavioral statements
  - do not contain `initial` or `always` statements
  - are called from `initial` or `always` statements or other tasks or functions

Differences between… (cont'd)

- Tasks can be used for common Verilog code
- Function are used when the common code
  - is purely combinational
  - executes in 0 simulation time
  - provides exactly one output
- Functions are typically used for conversions and commonly used calculations