# Department of Electronics & Communication Engg.

**Course : Digital System Design using Verilog.** **Sem.: 6th (2017-18)**

## Course Coordinator:

## Prof. D. M. Kumbhar

# Digital System Design Using Verilog

## Module 1

## Introduction and Methodology

# Digital Design

*Digital:* circuits that use two voltage levels to represent information

*Logic:* use truth values and logic to analyze circuits

*Design:* meeting functional requirements while satisfying constraints

History : Mechanical – electromechanical – analog

Use

Disadvantages : accuracy, speed, maintenance.

Early circuits - digital circuits.

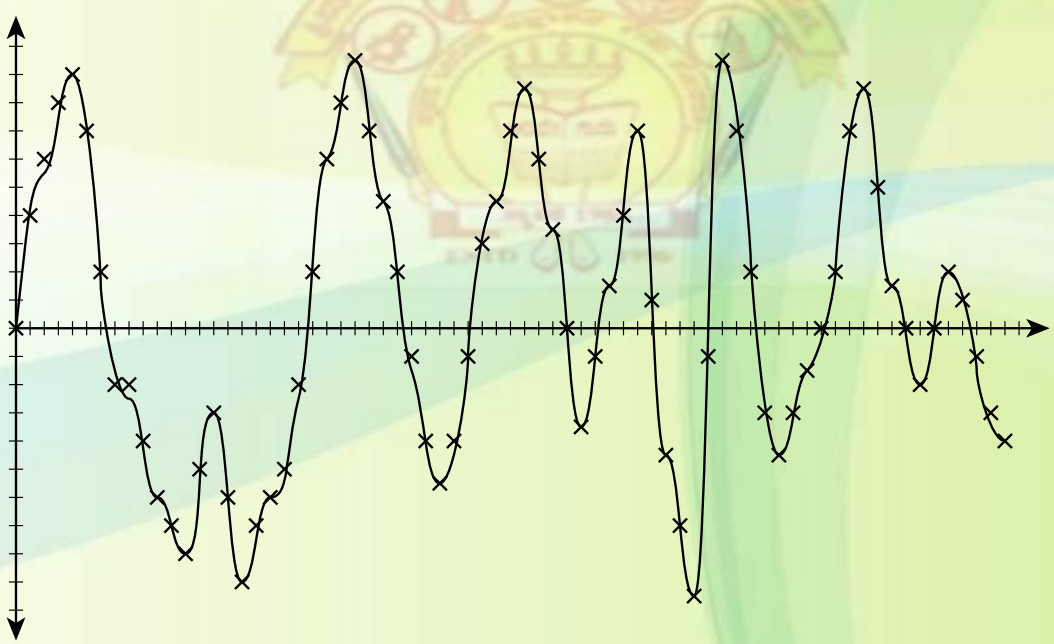
Constraints: performance, size, power, cost, etc.

# Design using Abstraction

- Circuits contain millions of transistors
  - How can we manage this complexity?
- Abstraction
  - Focus on aspects relevant aspects, ignoring other aspects
  - Don’t break assumptions that allow aspect to be ignored!
- Examples:
  - Transistors are on or off
  - Voltages are low or high

# Digital Systems

- Electronic circuits that use discrete representations of information
  - Discrete in space and time



# Embedded Systems

- Most real-world digital systems include embedded computers
  - Processor cores, memory, I/O
- Different functional requirements can be implemented
  - by the embedded software
  - by special-purpose attached circuits
- Trade-off among cost, performance, power, etc.

# Binary Representation

- Basic representation for simplest form of information, with only two states
  - a switch: open or closed
  - a light: on or off
  - a microphone: active or muted
  - a logical proposition: false or true
  - a binary (base 2) digit, or bit: 0 or 1

# Binary Representation: Example


- Signal represents the state of the switch
  - high-voltage => pressed,
    low-voltage => not pressed
- Equally, it represents state of the lamp
  - lamp_lit = switch_pressed

# Basic Gate Components

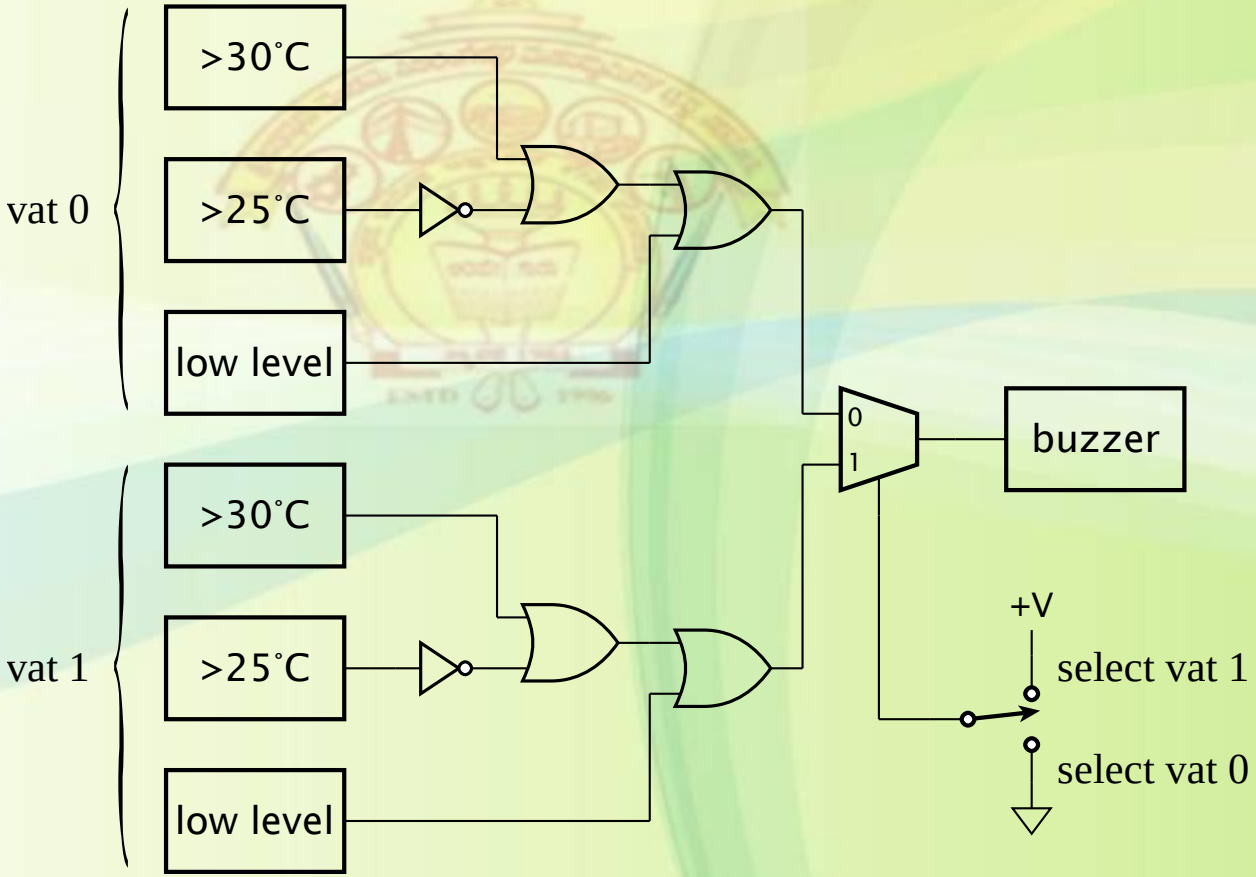
- Primitive components for logic design

AND gate OR gate


inverter multiplexer

# Combinational Circuits

- Circuit whose output values depend purely on current input values

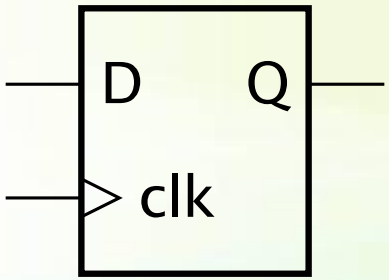


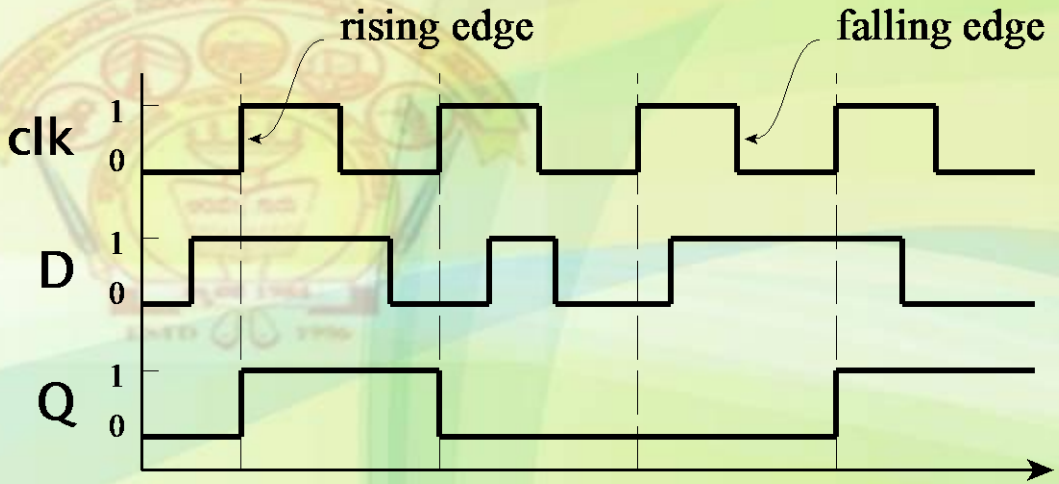
# Sequential Circuits

- Circuit whose output values depend on current *and previous* input values
  - Include some form of storage of values
- Nearly all digital systems are sequential
  - Mixture of gates and storage components
  - Combinational parts transform inputs and stored values

# Flipflops and Clocks

- Edge-triggered D-flipflop
  - stores one bit of information at a time



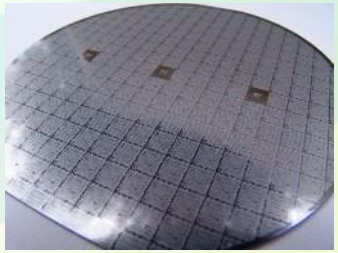

- Timing diagram
  - Graph of signal values versus time

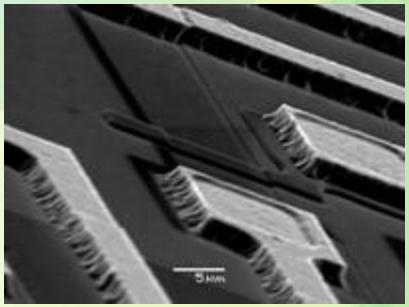
# Real-World Circuits

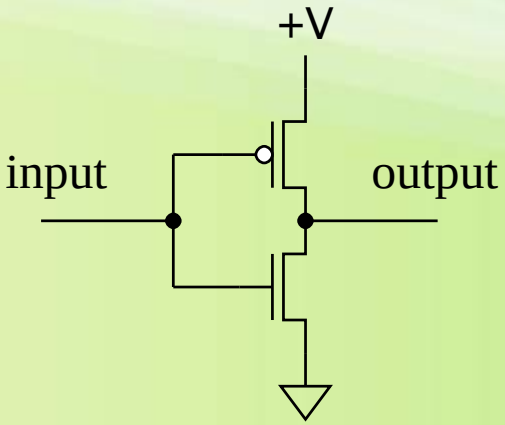
- Assumptions behind digital abstraction
  - ideal circuits, only two voltages, instantaneous transitions, no delay
- Greatly simplify functional design
- Constraints arise from real components and real-world physics
- Meeting constraints ensures circuits are “ideal enough” to support abstractions

# Integrated Circuits (ICs)

- Circuits formed on surface of silicon wafer
  - Minimum feature size reduced in each technology generation
  - Currently 90nm, 65nm
  - Moore's Law: increasing transistor count
  - CMOS: complementary MOSFET circuits

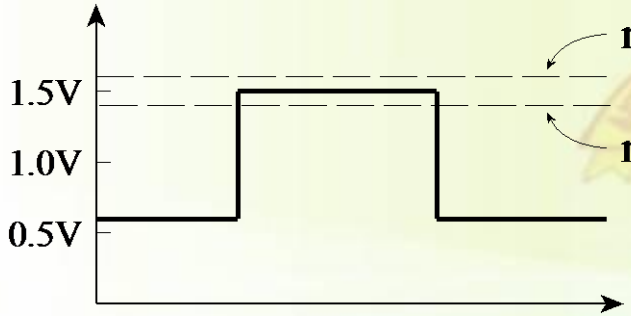


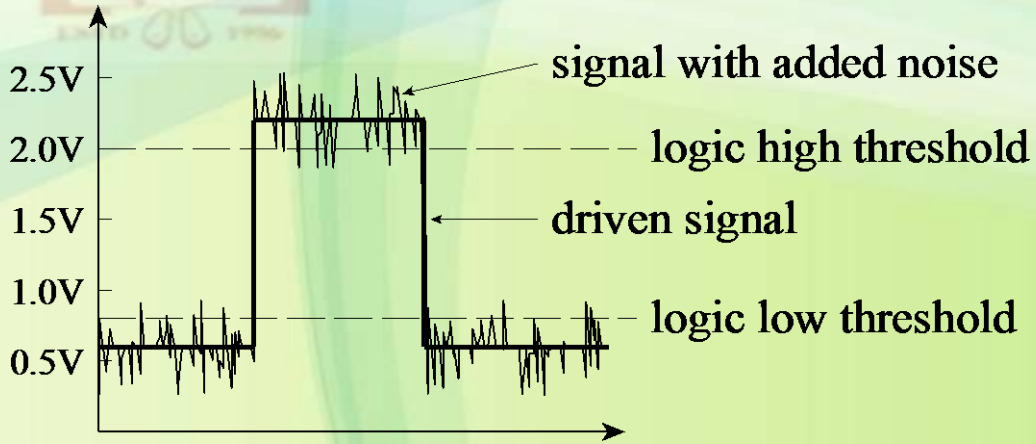




# Logic Levels

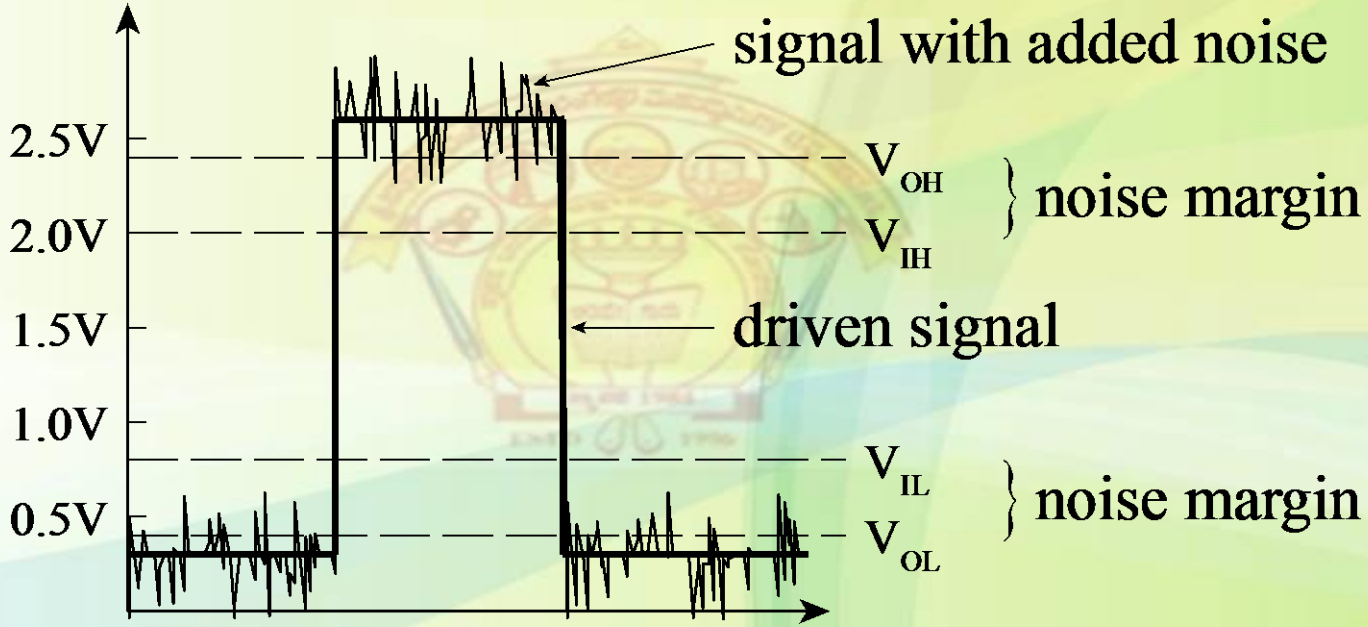
- Actual voltages for “low” and “high”
  - Example: 1.4V threshold for inputs



# Logic Levels

- TTL logic levels with noise margins




$V_{OL}$: output low voltage

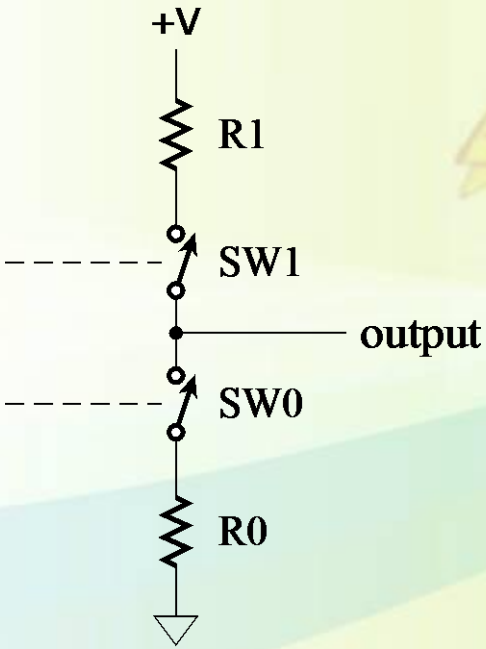
$V_{OH}$: output high voltage

$V_{IL}$: input low voltage

$V_{IH}$: input high voltage

# Static Load and Fanout

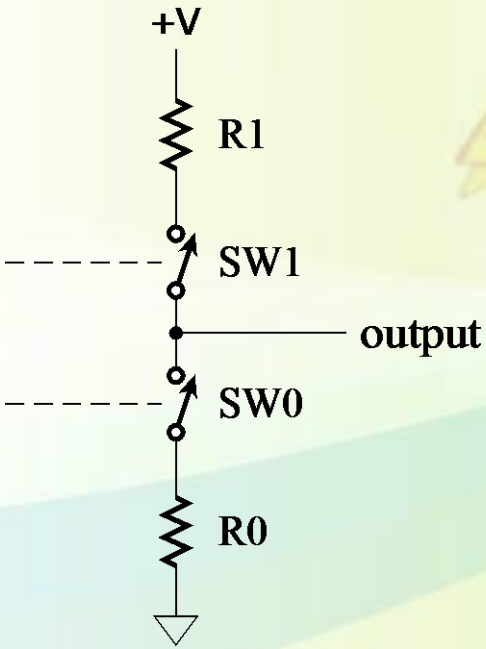
- Current flowing into or out of an output




  - High: SW1 closed, SW0 open
    - Voltage drop across R1
    - Too much current: $V_O < V_{OH}$
  - Low: SW0 closed, SW1 open
    - Voltage drop across R0
    - Too much current: $V_O > V_{OL}$
  - Fanout: number of inputs connected to an output
    - determines static load

# Static Load and Fanout

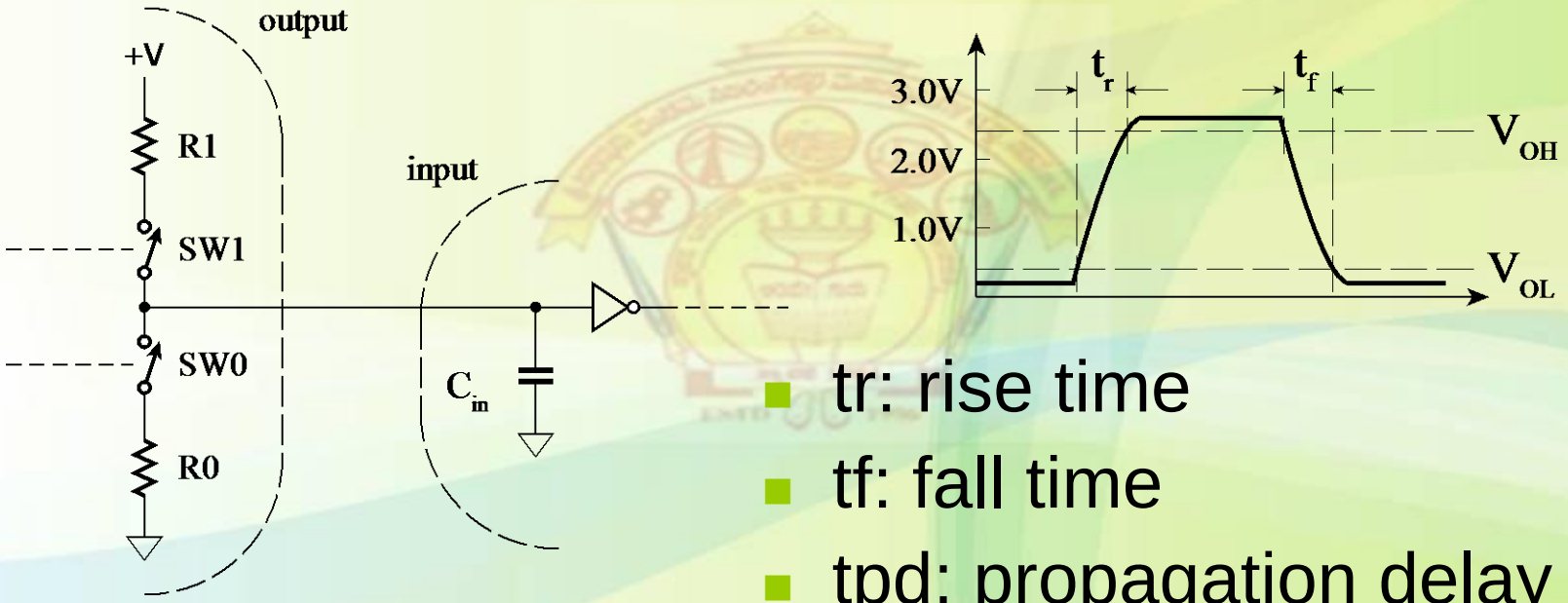
- Current flowing into or out of an output




  - High: SW1 closed, SW0 open
    - Voltage drop across R1
    - Too much current: $V_O < V_{OH}$
  - Low: SW0 closed, SW1 open
    - Voltage drop across R0
    - Too much current: $V_O > V_{OL}$
  - Fanout: number of inputs connected to an output
    - determines static load

# Capacitive Load and Prop Delay

- Inputs and wires act as capacitors




- tr: rise time
- tf: fall time
- tpd: propagation delay
  - delay from input transition to output transition
  - $t_{pd} = \max(t_{pd01}, t_{pd10})$

# Other Constraints

- Wire delay: delay for transition to traverse interconnecting wire
- Flipflop timing
  - delay from clk edge to Q output
  - D stable before and after clk edge
- Power
  - current through resistance => heat
  - must be dissipated, or circuit cooks!
  - Static & dynamic power consumption

# Area and Packaging

- Circuits implemented on silicon chips
  - Larger circuit area => greater cost
- Chips in packages with connecting wires
  - More wires => greater cost
  - Package dissipates heat
- Packages interconnected on a printed circuit board (PCB)
  - Size, shape, cooling, etc, constrained by final product

# Models

- Model: represents interested aspects omits other (abstraction of an object) Ex. House, train, plane.
- Electronic model: Prototype circuit

  Abstract expression in some modeling language
- Abstract representations of aspects of a system being designed
  - Allow us to analyze the system before building it
- Example: Ohm's Law
  - $V = I \times R$
  - Represents electrical aspects of a resistor
  - Expressed as a mathematical equation
  - Ignores thermal, mechanical, materials aspects

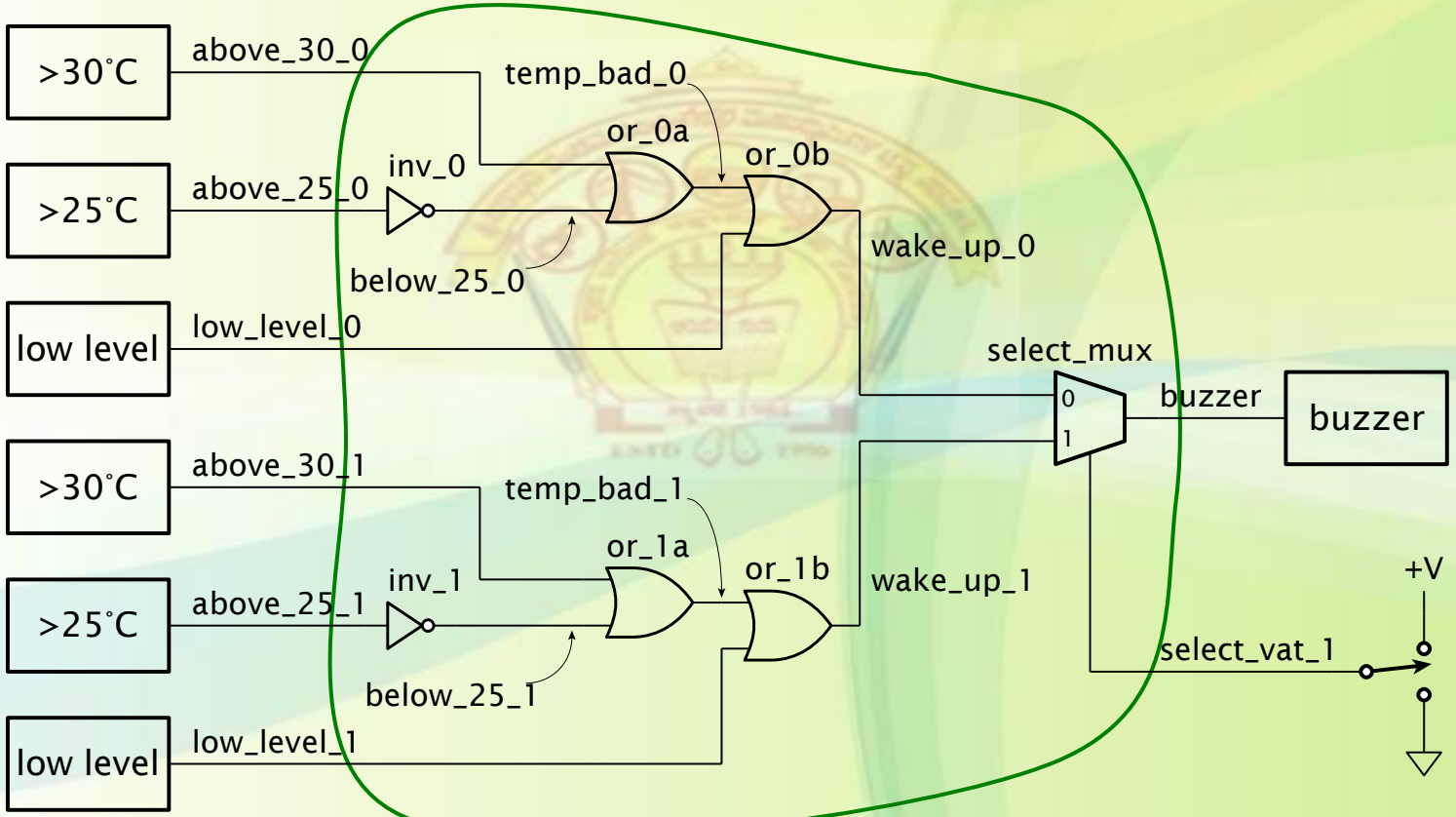
# Models

- Model: represents interested aspects omits other (abstraction of an object) Ex. House, train, plane.
- Electronic model: Prototype circuit

  Abstract expression in some modeling language
- Abstract representations of aspects of a system being designed
  - Allow us to analyze the system before building it
- Example: Ohm's Law
  - $V = I \times R$
  - Represents electrical aspects of a resistor
  - Expressed as a mathematical equation
  - Ignores thermal, mechanical, materials aspects

# Module Ports

- Describe input and outputs of a circuit



# Structural Module Definition

```
module vat_buzzer_struct
  ( output buzzer,
    input above_25_0, above_30_0, low_level_0,
    input above_25_1, above_30_1, low_level_1,
    input select_vat_1 );

  wire below_25_0, temp_bad_0, wake_up_0;
  wire below_25_1, temp_bad_1, wake_up_1;

  // components for vat 0
  not inv_0 (below_25_0, above_25_0);
  or  or_0a (temp_bad_0, above_30_0, below_25_0);
  or  or_0b (wake_up_0, temp_bad_0, low_level_0);

  // components for vat 1
  not inv_1 (below_25_1, above_25_1);
  or  or_1a (temp_bad_1, above_30_1, below_25_1);
  or  or_1b (wake_up_1, temp_bad_1, low_level_1);

  mux2 select_mux (buzzer, select_vat_1, wake_up_0, wake_up_1);

endmodule
```

# Behavioral Module Definition

```
module vat_buzzer_struct
  ( output buzzer,
    input above_25_0, above_30_0, low_level_0,
    input above_25_1, above_30_1, low_level_1,
    input select_vat_1 );

  assign buzzer =
    select_vat_1 ? low_level_1 | (above_30_1 | ~above_25_1)
                 : low_level_0 | (above_30_0 | ~above_25_0);

endmodule
```

# Design Methodology

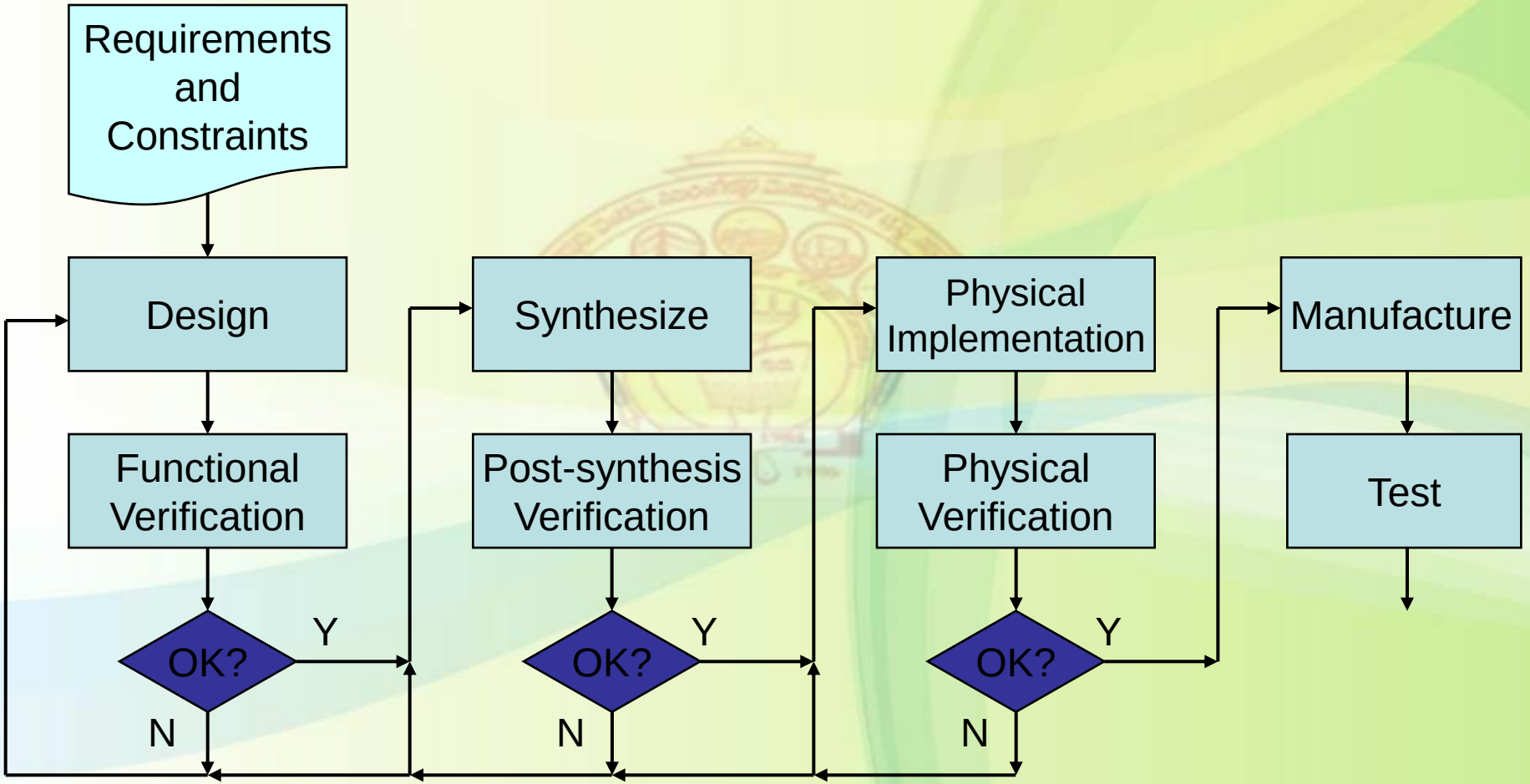
- Design: complex, large no of undertakings & requirements.

  Systematic approach of working out how to construct circuits that meets given requirements.
- Simple systems can be design by one person using *ad hoc* methods
- Real-world systems are design by teams
  - Require a systematic design methodology
  - Design methodology: systematic process of design, verification and preparation for manufacture a product.
- Specifies
  - Tasks to be undertaken
  - Information needed and produced
  - Relationships between tasks
    - dependencies, sequences
  - EDA tools used

# Design Methodology

- A mature design methodology: schedule & budget, no of errors detected and missed, data from previous projects to improve new one
- Advantages:
  - Design process more reliable and predictable
  - Reducing risk and cost
  - Reducing scale

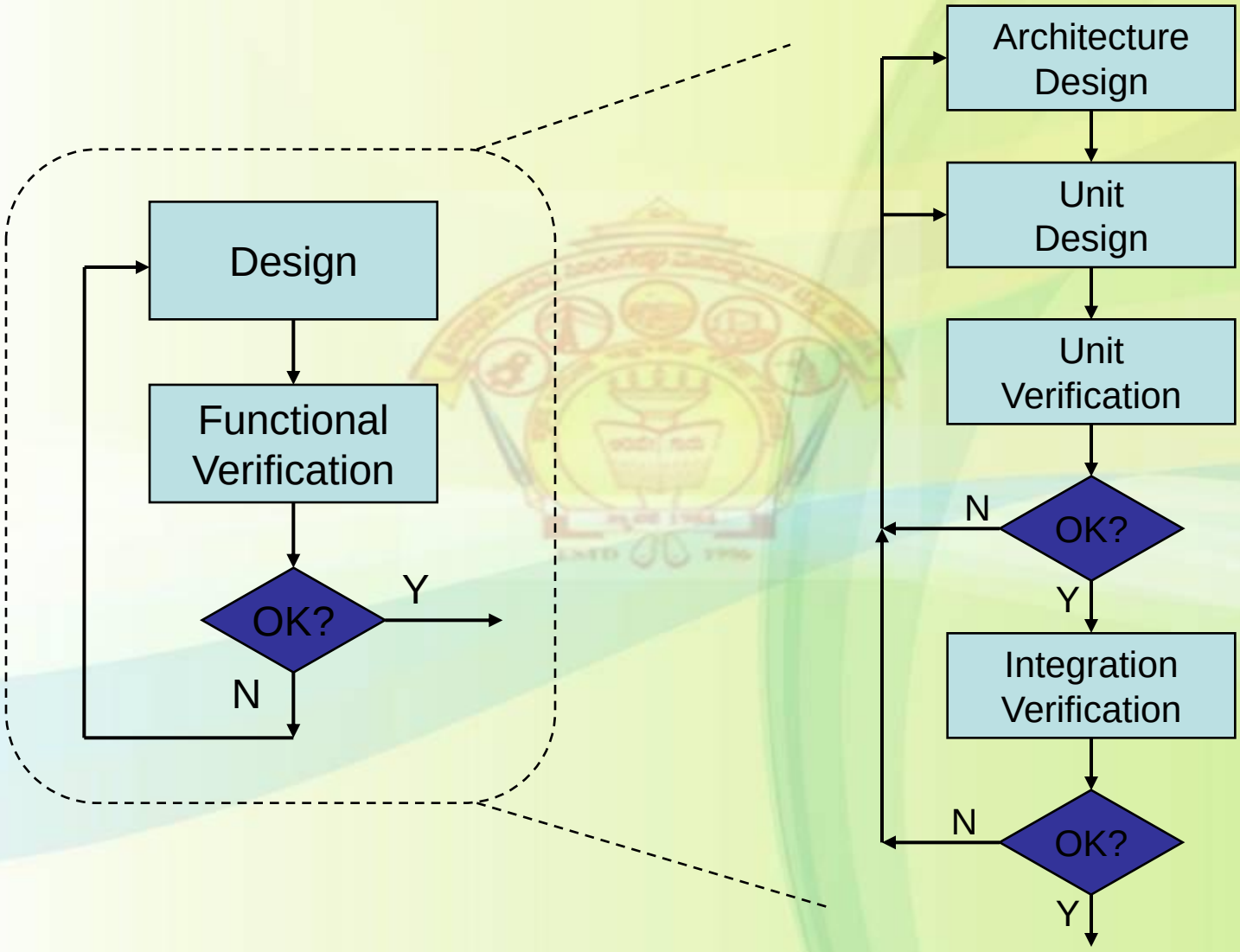
# A Simple Design Methodology



# Hierarchical Design

- Circuits are too complex for us to design all the detail at once
- Design subsystems for simple functions
- Compose subsystems to form the system
  - Treating sub circuits as “black box” components
  - Ex. Display
  - Reuse-present, previous or third party project
  - Save – design effort & cost.
  - Verify independently, then verify the composition
- Top-down/bottom-up design

# Hierarchical Design



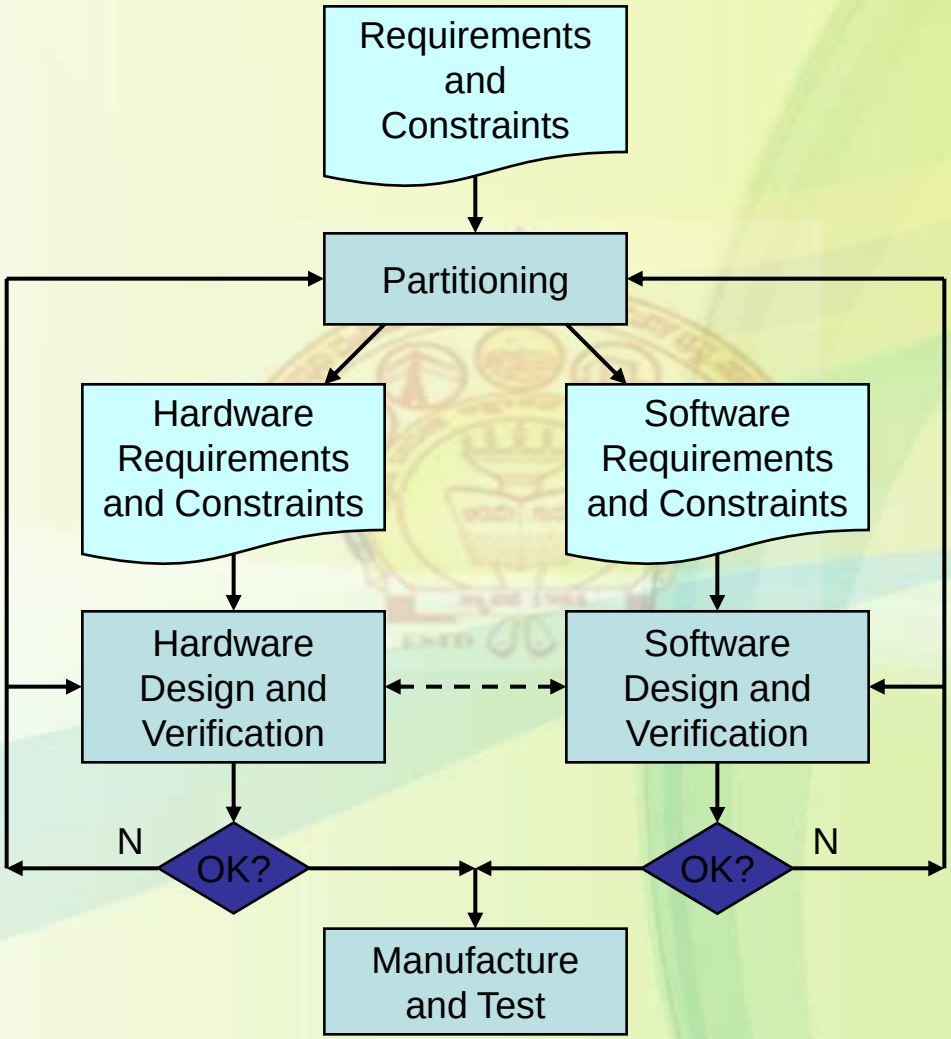
# Synthesis

- We usually design using register-transfer-level (RTL) Verilog
  - Higher level of abstraction than gates
- Synthesis tool translates to a circuit of gates that performs the same function
- Specify to the tool
  - the target implementation fabric
  - Library – properties, timing, area, power
  - constraints on timing, area, etc.
- Post-synthesis verification
  - synthesized circuit meets constraints

# Physical Implementation

- Implementation fabrics
  - Application-specific ICs (ASICs)
  - Field-programmable gate arrays (FPGAs)
- Floor-planning: arranging the subsystems
- Placement: arranging the gates within subsystems
- Routing: joining the gates with wires
- Physical verification
  - physical circuit still meets constraints
  - use better estimates of delays

# Embedded system Design Codesign Methodology



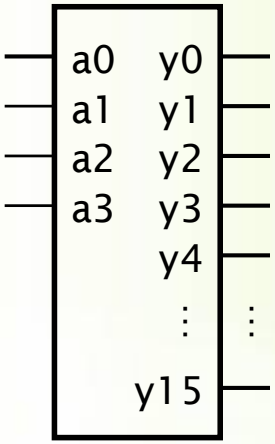
# Combinational Circuits

- Circuits whose outputs depend only on current input values
  - no storage of past input values
  - no *state*
- Can be analyzed using laws of logic
  - Boolean algebra, similar to propositional calculus

# Combinational Components

- We can build complex combination components from gates
  - Decoders, encoders
  - Multiplexers
  - …
- Use them as subcomponents of larger systems
  - Abstraction and reuse

# Decoders




- A decoder derives control signals from a binary coded signal
  - One per code word
  - Control signal is 1 when input has the corresponding code word; 0 otherwise
- For an *n*-bit code input
  - Decoder has $2^n$ outputs
- Example: ($a_3$, $a_2$, $a_1$, $a_1$)
  - Output for (1, 0, 1, 1): $y_{11} = a_3 \cdot \overline{a_2} \cdot a_1 \cdot a_0$

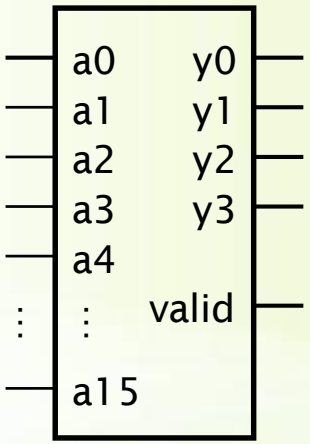
# Decoder Example

| Color | Codeword ($c_2$, $c_1$, $c_0$) |
|---|---|
| black | 0, 0, 1 |
| cyan | 0, 1, 0 |
| magenta | 0, 1, 1 |
| yellow | 1, 0, 0 |
| red | 1, 0, 1 |
| blue | 1, 1, 0 |

# Decoder Example

```
module ink_jet_decoder
  ( output black, cyan, magenta, yellow,
           light_cyan, light_magenta,
    input  color2, color1, color0 );

  assign black         = ~color2 & ~color1 &  color0;
  assign cyan          = ~color2 &  color1 & ~color0;
  assign magenta       = ~color2 &  color1 &  color0;
  assign yellow        =  color2 & ~color1 & ~color0;
  assign light_cyan    =  color2 & ~color1 &  color0;
  assign light_magenta =  color2 &  color1 & ~color0;

endmodule
```

# Encoders




- An encoder encodes which of several inputs is 1
  - Assuming (for now) at most one input is 1 at a time
- What if no input is 1?
  - Separate output to indicate this condition

# Encoder Example

- Burglar alarm: encode which zone is active

| Zone | Codeword |
|---|---|
| Zone 1 | 0, 0, 0 |
| Zone 2 | 0, 0, 1 |
| Zone 3 | 0, 1, 0 |
| Zone 4 | 0, 1, 1 |
| Zone 5 | 1, 0, 0 |
| Zone 6 | 1, 0, 1 |
| Zone 7 | 1, 1, 0 |
| Zone 8 | 1, 1, 1 |

# Encoder Example

```
module alarm_eqn ( output [2:0] intruder_zone,
                   output       valid,
                   input  [1:8] zone );

  assign intruder_zone[2] = zone[5] | zone[6] |
                            zone[7] | zone[8];
  assign intruder_zone[1] = zone[3] | zone[4] |
                            zone[7] | zone[8];
  assign intruder_zone[0] = zone[2] | zone[4] |
                            zone[6] | zone[8];

  assign valid = zone[1] | zone[2] | zone[3] | zone[4] |
                 zone[5] | zone[6] | zone[7] | zone[8];

endmodule
```

# Priority Encoders

- If more than one input can be 1
  - Encode input that is 1 with highest priority

| zone | | | | | | | | intruder_zone | | | valid |
|---|---|---|---|---|---|---|---|---|---|---|---|
| (1) | (2) | (3) | (4) | (5) | (6) | (7) | (8) | (2) | (1) | (0) | |
| 1 | – | – | – | – | – | – | – | 0 | 0 | 0 | 1 |
| 0 | 1 | – | – | – | – | – | – | 0 | 0 | 1 | 1 |
| 0 | 0 | 1 | – | – | – | – | – | 0 | 1 | 0 | 1 |
| 0 | 0 | 0 | 1 | – | – | – | – | 0 | 1 | 1 | 1 |
| 0 | 0 | 0 | 0 | 1 | – | – | – | 1 | 0 | 0 | 1 |
| 0 | 0 | 0 | 0 | 0 | 1 | – | – | 1 | 0 | 1 | 1 |
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | – | 1 | 1 | 0 | 1 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | – | – | – | 0 |

# Priority Encoder Example

```
module alarm_priority_1 ( output [2:0] intruder_zone,
                          output       valid,
                          input  [1:8] zone );

  assign intruder_zone = zone[1] ? 3'b000 :
                         zone[2] ? 3'b001 :
                         zone[3] ? 3'b010 :
                         zone[4] ? 3'b011 :
                         zone[5] ? 3'b100 :
                         zone[6] ? 3'b101 :
                         zone[7] ? 3'b110 :
                         zone[8] ? 3'b111 :
                         3'b000;

  assign valid = zone[1] | zone[2] | zone[3] | zone[4] |
                 zone[5] | zone[6] | zone[7] | zone[8];

endmodule
```

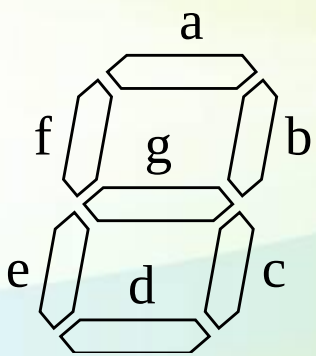
# BCD Code

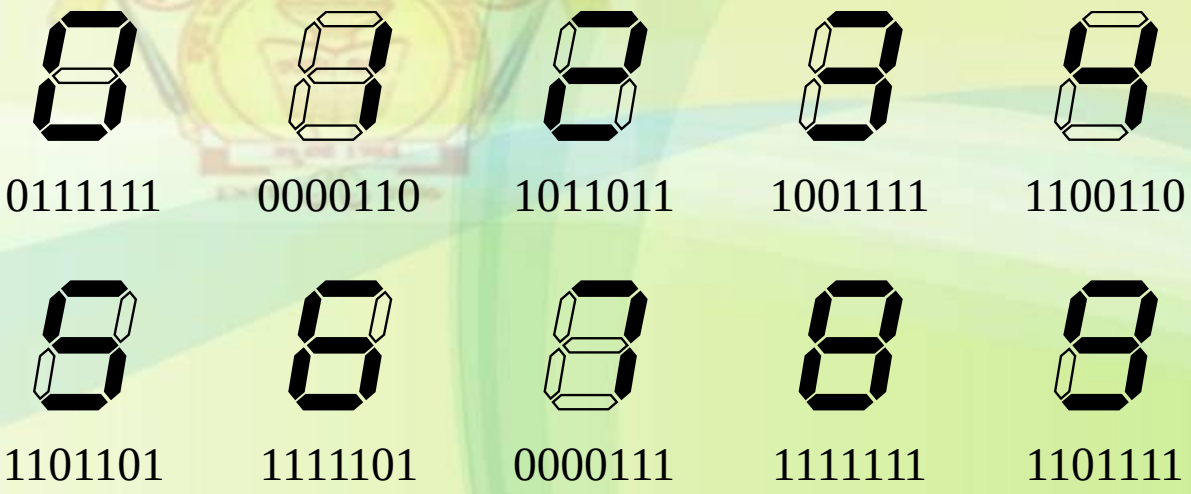
- Binary coded decimal
  - 4-bit code for decimal digits

| | | | | |
|---|---|---|---|---|
| 0: 0000 | 1: 0001 | 2: 0010 | 3: 0011 | 4: 0100 |
| 5: 0101 | 6: 0110 | 7: 0111 | 8: 1000 | 9: 1001 |

# Seven-Segment Decoder

- Decodes BCD to drive a 7-segment LED or LCD display digit
  - Segments: (g, f, e, d, c, b, a)



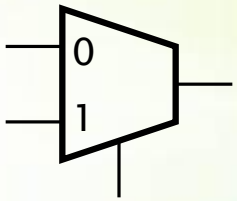
# Seven-Segment Decoder

```
module seven_seg_decoder ( output [7:1] seg,
                           input  [3:0] bcd, input blank );

  reg [7:1] seg_tmp;

  always @*
    case (bcd)
      4'b0000: seg_tmp = 7'b0111111;   // 0
      4'b0001: seg_tmp = 7'b0000110;   // 1
      4'b0010: seg_tmp = 7'b1011011;   // 2
      4'b0011: seg_tmp = 7'b1001111;   // 3
      4'b0100: seg_tmp = 7'b1100110;   // 4
      4'b0101: seg_tmp = 7'b1101101;   // 5
      4'b0110: seg_tmp = 7'b1111101;   // 6
      4'b0111: seg_tmp = 7'b0000111;   // 7
      4'b1000: seg_tmp = 7'b1111111;   // 8
      4'b1001: seg_tmp = 7'b1101111;   // 9
      default: seg_tmp = 7'b1000000;   // "-" for invalid code
    endcase

 assign seg = blank ? 7'b0000000 : seg_tmp;

endmodule
```

# Multiplexers

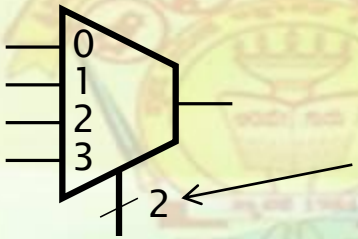
- Chooses between data inputs based on the select input

2-to-1 mux




| sel | z |
|---|---|
| 0 | $a_0$ |
| 1 | $a_1$ |

4-to-1 mux




two select bits

| sel | z |
|---|---|
| 00 | $a_0$ |
| 01 | $a_1$ |
| 10 | $a_2$ |
| 11 | $a_3$ |

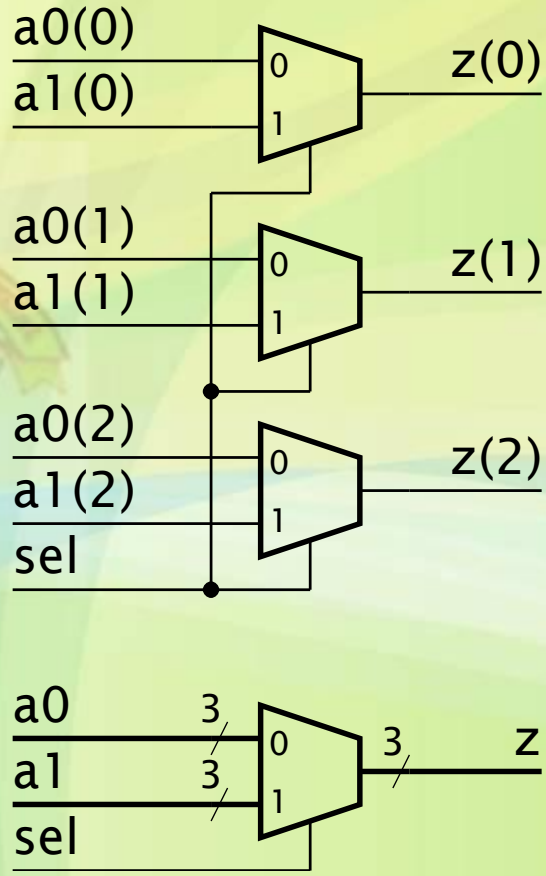
- *N*-to-1 multiplexer needs $\lceil \log_2 N \rceil$ select bits

# Multiplexer Example

```
module multiplexer_4_to_1 ( output reg       z,
                            input      [3:0] a,
                            input            sel );

  always @*
    case (sel)
      2'b00: z = a[0];
      2'b01: z = a[1];
      2'b10: z = a[2];
      2'b11: z = a[3];
    endcase

endmodule
```

# Multi-bit Multiplexers

- To select between *N* *m*-bit codeword inputs
  - Connect *m* *N*-input multiplexers in parallel
  - 3-bit 2 codewords requires 3, 2 input multiplexers
- Abstraction
  - Treat this as a component



# Multi-bit Mux Example

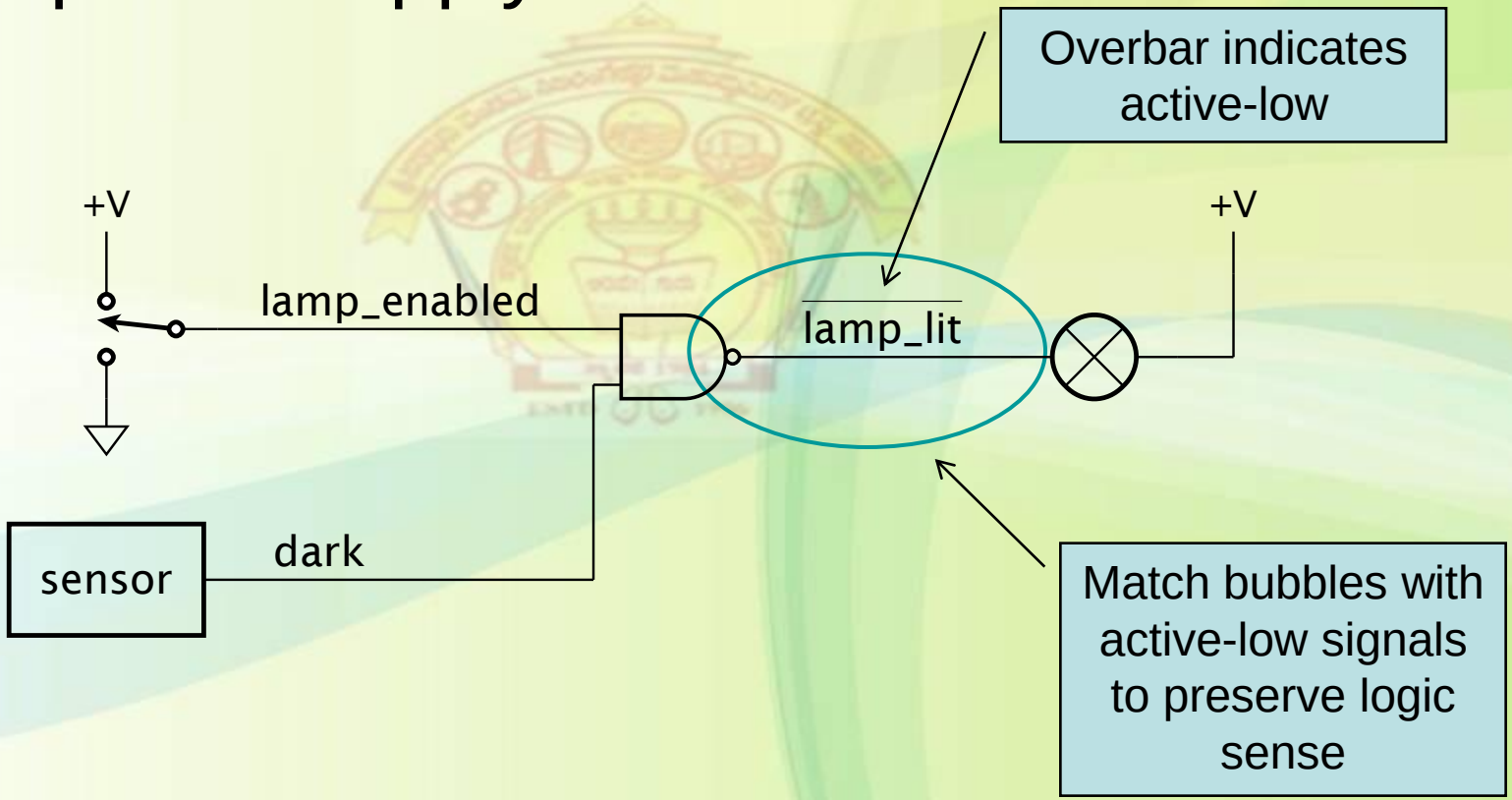
```
module multiplexer_3bit_2_to_1 ( output [2:0] z,
                                 input  [2:0] a0, a1,
                                 input        sel );

  assign z = sel ? a1 : a0;

endmodule
```

# Active-Low Logic

- We've been using active-high logic
  - 0 (low voltage): falsehood of a condition
  - 1 (high voltage): truth of a condition
- Active-low logic logic
  - 0 (low voltage): ***truth*** of a condition
  - 1 (high voltage): ***falsehood*** of a condition
  - reverses the representation, ***not*** negative voltage!
- In circuit schematics, label active-low signals with overbar notation
  - eg, $\overline{\text{lamp\_lit}}$: low when lit, high when not lit

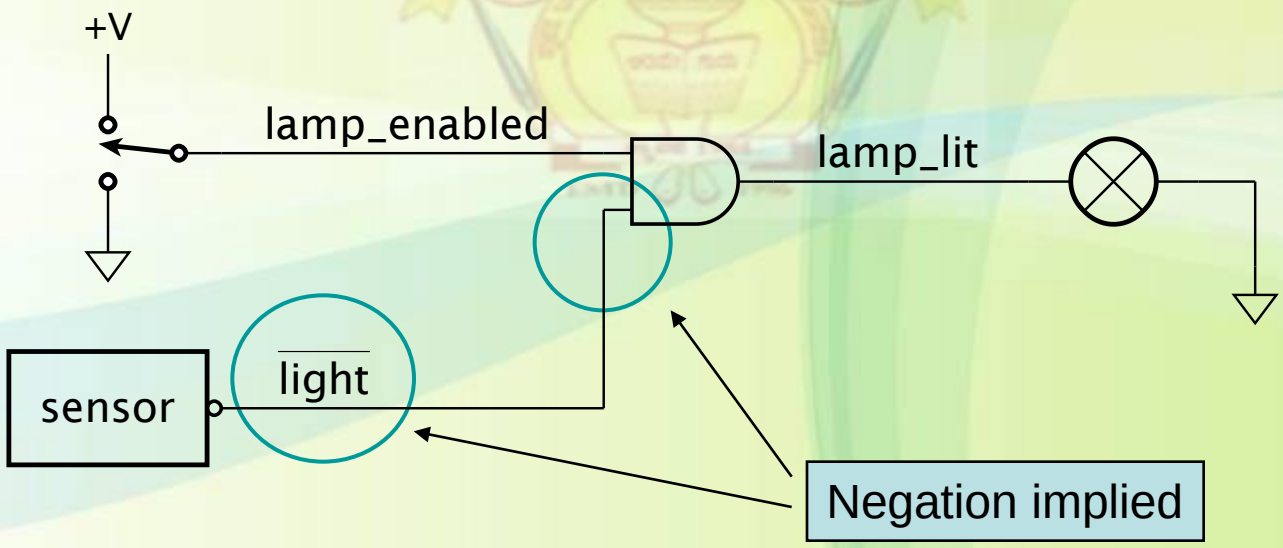
# Active-Low Example

- Night-light circuit, lamp connected to power supply



# Implied Negation

- Negation implied by connecting
  - An active-low signal to an active-high input/output
  - An active-high signal to an active-low input/output



# Active-Low Signals and Gates

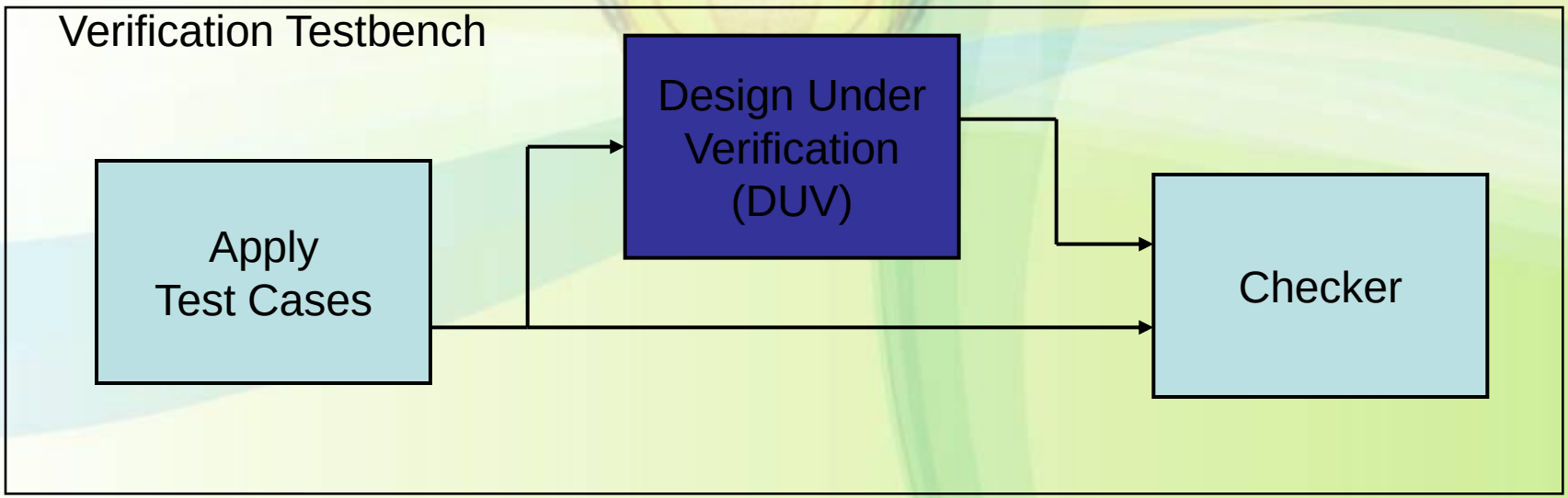
- DeMorgan's laws suggest alternate views for gates
  - They're the same electrical circuit!
  - Use the view that best represents the logical function intended
  - Match the bubbles, unless implied negation is intended

# Active-Low Logic in Verilog

- Can’t draw an overbar in Verilog
  - Use _N suffix on signal or port name
- 1'b0 and 1'b1 in Verilog mean low and high
- For active-low logic
  - 1'b0 means the condition is true
  - 1'b1 means the condition is false
- Example
  - `assign lamp_lit_N = 1'b0;`
  - turns the lamp on

# Combinational Verification

- Design Methodology – requirements & constraints
- Combination circuits: outputs are a function of inputs
  - Functional verification: making sure it's the right function!
  - Testbench model
  - DUV /DUT



# Verification Example

- Verify operation of traffic-light controller
- Property to check
  - enable $\Rightarrow$ lights_out == lights_in
  - !enable $\Rightarrow$ all lights are inactive
- Represent this as an assertion in the checker

# Testbench Module

```
`timescale 1ms/1ms
module light_testbench;
  wire [1:3] lights_out;
  reg  [1:3] lights_in;
  reg        enable;
  light_controller_and_enable duv ( .lights_out(lights_out),
                                    .lights_in(lights_in),
                                    .enable(enable) );
```

# Applying Test Cases

```
initial begin
        enable = 0; lights_in = 3'b000;
  #1000 enable = 0; lights_in = 3'b001;
  #1000 enable = 0; lights_in = 3'b010;
  #1000 enable = 0; lights_in = 3'b100;
  #1000 enable = 1; lights_in = 3'b001;
  #1000 enable = 1; lights_in = 3'b010;
  #1000 enable = 1; lights_in = 3'b100;
  #1000 enable = 1; lights_in = 3'b000;
  #1000 enable = 1; lights_in = 3'b111;
  #1000 $finish;
end
```

# Checking Assertions

```
  always @(enable or lights_in) begin
    #10
    if (!( ( enable && lights_out == lights_in) ||
           (!enable && lights_out == 3'b000) ))
      $display("Error in light controller output");
  end

endmodule
```

# Functional Coverage

- Did we test all possible input cases?
- For large designs, exhaustive testing is not tractable
  - *N* inputs: number of cases = $2^N$
- Functional coverage
  - Proportion of test cases covered by a testbench
  - It can be hard to decide how much testing is enough

# Sequential Basics

- Sequential circuits
  - Outputs depend on current inputs and previous inputs
  - Store *state*: an abstraction of the history of inputs
- Usually governed by a periodic clock signal
- Flip flop, registers, counters

# Datapaths and Control

- Digital systems perform sequences of operations on encoded data
- *Datapath*
  - Combinational circuits for operations
  - Registers for storing intermediate results
- *Control section*: control sequencing
  - Generates *control signals*
    - Selecting operands
    - Selecting operations to perform
    - Enabling registers at the right times
    - Activate signal at right order & right time
  - Uses *status signals* from datapath
- *Challenging task:* requirements & constraints
  - Functional requirements – alternatives for implementation
  - Tradeoff – area, performance.

# Example: Complex Multiplier

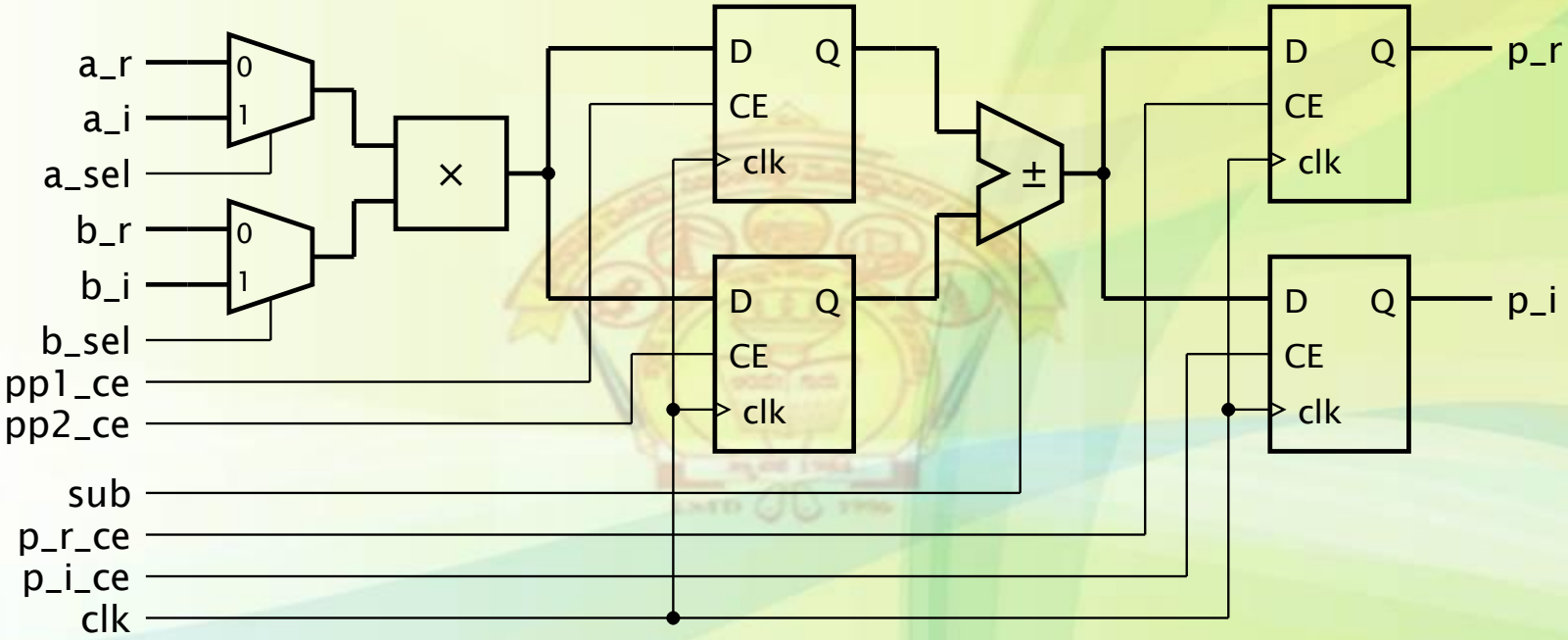
- Cartesian form, fixed-point
  - operands: 4 pre-, 12 post-binary-point bits
  - result: 8 pre-, 24 post-binary-point bits
- Subject to tight area constraints

$$a = a_r + ja_i \qquad b = b_r + jb_i$$

$$p = ab = p_r + jp_i = (a_r b_r - a_i b_i) + j(a_r b_i + a_i b_r)$$

- 4 multiplies, 1 add, 1 subtract
  - Perform sequentially using 1 multiplier, 1 adder/subtracter

# Complex Multiplier Datapath



# Complex Multiplier in Verilog

```
module multiplier
  ( output reg signed [7:-24] p_r, p_i,
    input      signed [3:-12] a_r, a_i, b_r, b_i,
    input                     clk, reset, input_rdy );
  reg a_sel, b_sel, pp1_ce, pp2_ce, sub, p_r_ce, p_i_ce;
  wire signed [3:-12] a_operand, b_operand;
  wire signed [7:-24] pp, sum
  reg  signed [7:-24] pp1, pp2;
  ...
```

# Complex Multiplier in Verilog

```
  assign a_operand = ~a_sel ? a_r : a_i;
  assign b_operand = ~b_sel ? b_r : b_i;
  assign pp = {{4{a_operand[3]}}, a_operand, 12'b0} *
              {{4{b_operand[3]}}, b_operand, 12'b0};
  always @(posedge clk)  // Partial product 1 register
    if (pp1_ce) pp1 <= pp;
  always @(posedge clk)  // Partial product 2 register
    if (pp2_ce) pp2 <= pp;
  assign sum = ~sub ? pp1 + pp2 : pp1 - pp2;
  always @(posedge clk)  // Product real-part register
    if (p_r_ce) p_r <= sum;
  always @(posedge clk)  // Product imaginary-part register
    if (p_i_ce) p_i <= sum;
  ...
endmodule
```

# Multiplier Control Sequence

- Avoid resource conflict
- First attempt
  1. a_r * b_r → pp1_reg
  2. a_i * b_i → pp2_reg
  3. pp1 – pp2 → p_r_reg
  4. a_r * b_i → pp1_reg
  5. a_i * b_r → pp2_reg
  6. pp1 + pp2 → p_i_reg
- Takes 6 clock cycles

# Multiplier Control Sequence

- Merge steps where no resource conflict
- Revised attempt
  1. a_r * b_r → pp1_reg
  2. a_i * b_i → pp2_reg
  3. pp1 – pp2 → p_r_reg
     a_r * b_i → pp1_reg
  4. a_i * b_r → pp2_reg
  5. pp1 + pp2 → p_i_reg
- Takes 5 clock cycles

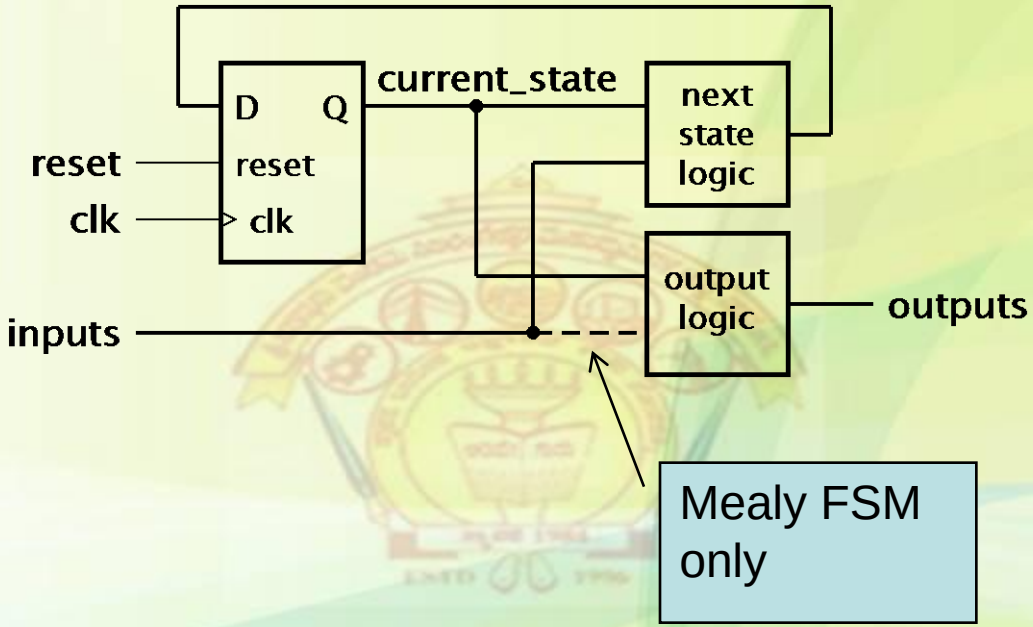
# Multiplier Control Signals

| Step | a_sel | b_sel | pp1_ce | pp2_ce | sub | p_r_ce | p_i_ce |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 1 | 0 | – | 0 | 0 |
| 2 | 1 | 1 | 0 | 1 | – | 0 | 0 |
| 3 | 0 | 1 | 1 | 0 | 1 | 1 | 0 |
| 4 | 1 | 0 | 0 | 1 | – | 0 | 0 |
| 5 | – | – | 0 | 0 | 0 | 0 | 1 |

# Finite-State Machines

- Used the implement control sequencing
  - Based on mathematical automaton theory
- A FSM is defined by
  - set of inputs: $\Sigma$
  - set of outputs: $\Gamma$
  - set of states: S
  - initial state: $s_0 \in S$
  - transition function: $\delta: S \times \Sigma \rightarrow S$
  - output function: $\omega: S \times \Sigma \rightarrow \Gamma$ or $\omega: S \rightarrow \Gamma$

# FSM in Hardware




- Mealy FSM: $\omega: S \times \Sigma \rightarrow \Gamma$
- Moore FSM: $\omega: S \rightarrow \Gamma$

# FSM Example: Multiplier Control

- One state per step
- Separate idle state?
  - Wait for input_rdy = 1
  - Then proceed to steps 1, 2, ...
  - But this wastes a cycle!
- Use step 1 as idle state
  - Repeat step 1 if input_rdy ≠ 1
  - Proceed to step 2 otherwise
- Output function
  - Defined by table on slide 43
  - Moore or Mealy?

Transition function

| current_ state | input_ rdy | next_ state |
|---|---|---|
| step1 | 0 | step1 |
| step1 | 1 | step2 |
| step2 | – | step3 |
| step3 | – | step4 |
| step4 | – | step5 |
| step5 | – | step1 |

# State Encoding

- Encoded in binary
  - $N$ states: use at least $\lceil \log_2 N \rceil$ bits
- Encoded value used in circuits for transition and output function
  - encoding affects circuit complexity
- Optimal encoding is hard to find
  - CAD tools can do this well
- One-hot works well in FPGAs
- Often use 000...0 for idle state
  - reset state register to idle

# FSMs in Verilog

- Use parameters for state values
  - Synthesis tool can choose an alternative encoding

```
parameter [2:0] step1 = 3'b000, step2 = 3'b001,
                step3 = 3'b010, step4 = 3'b011,
                step5 = 3'b100;
reg [2:0] current_state, next_state ;
...
```

# Multiplier Control in Verilog

```
always @(posedge clk or posedge reset)  // State register
  if (reset) current_state <= step1;
  else       current_state <= next_state;
always @*  // Next-state logic
  case (current_state)
    step1: if (!input_rdy) next_state = step1;
           else            next_state = step2;
    step2:                 next_state = step3;
    step3:                 next_state = step4;
    step4:                 next_state = step5;
    step5:                 next_state = step1;
  endcase
```

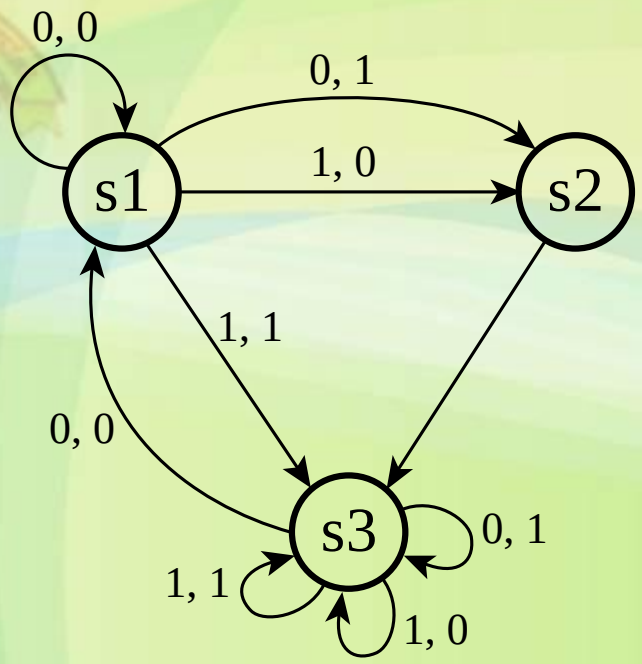
# Multiplier Control in Verilog

```
always @* begin  // Output_logic
  a_sel = 1'b0; b_sel = 1'b0;  pp1_ce = 1'b0; pp2_ce = 1'b0;
  sub = 1'b0;   p_r_ce = 1'b0; p_i_ce = 1'b0;
  case (current_state)
    step1: begin
             pp1_ce = 1'b1;
           end
    step2: begin
             a_sel = 1'b1; b_sel = 1'b1; pp2_ce = 1'b1;
           end
    step3: begin
             b_sel = 1'b1; pp1_ce = 1'b1;
             sub = 1'b1;   p_r_ce = 1'b1;
           end
    step4: begin
             a_sel = 1'b1; pp2_ce = 1'b1;
           end
    step5: begin
             p_i_ce = 1'b1;
           end
  endcase
end
```

# State Transition Diagrams

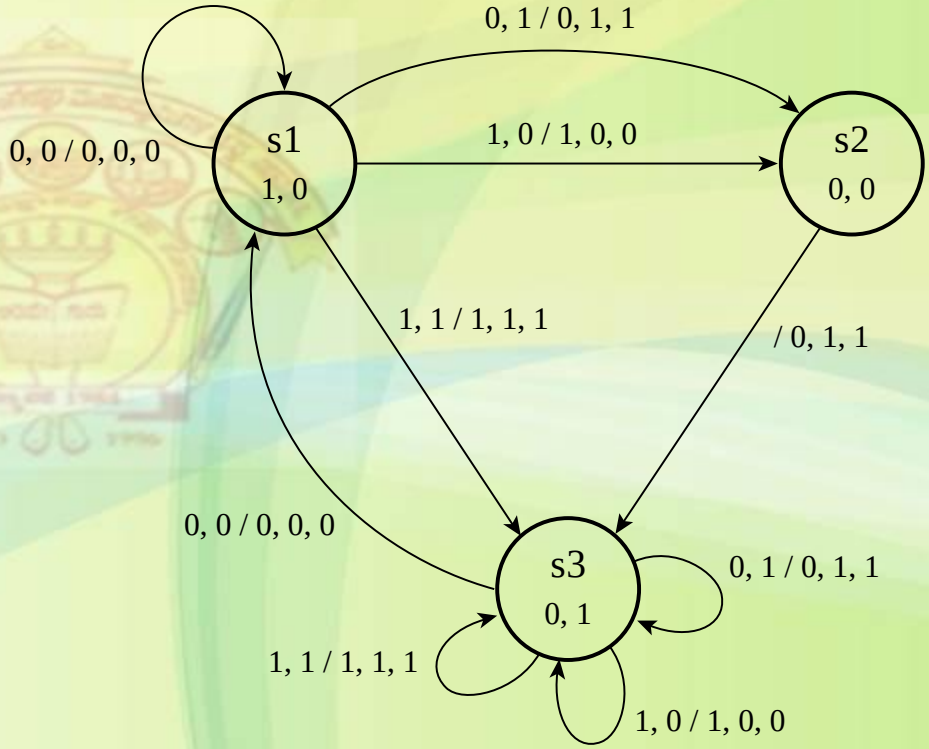
- Bubbles to represent states
- Arcs to represent transitions

- Example
  - S = {s1, s2, s3}
  - Inputs (a1, a2):
    Σ = {(0,0), (0,1), (1,0), (1,1)}
  - δ defined by diagram



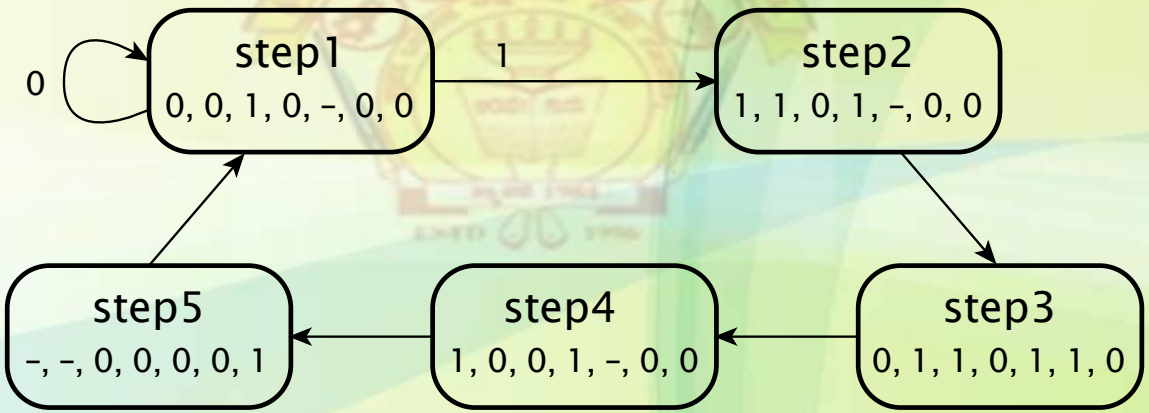
# State Transition Diagrams

- Annotate diagram to define output function
  - Annotate states for Moore-style outputs
  - Annotate arcs for Mealy-style outputs
- Example
  - $x_1$, $x_2$: Moore-style
  - $y_1$, $y_2$, $y_3$: Mealy-style



# Multiplier Control Diagram

- Input: input_rdy
- Outputs
  - a_sel, b_sel, pp1_ce, pp2_ce, sub, p_r_ce, p_i_ce

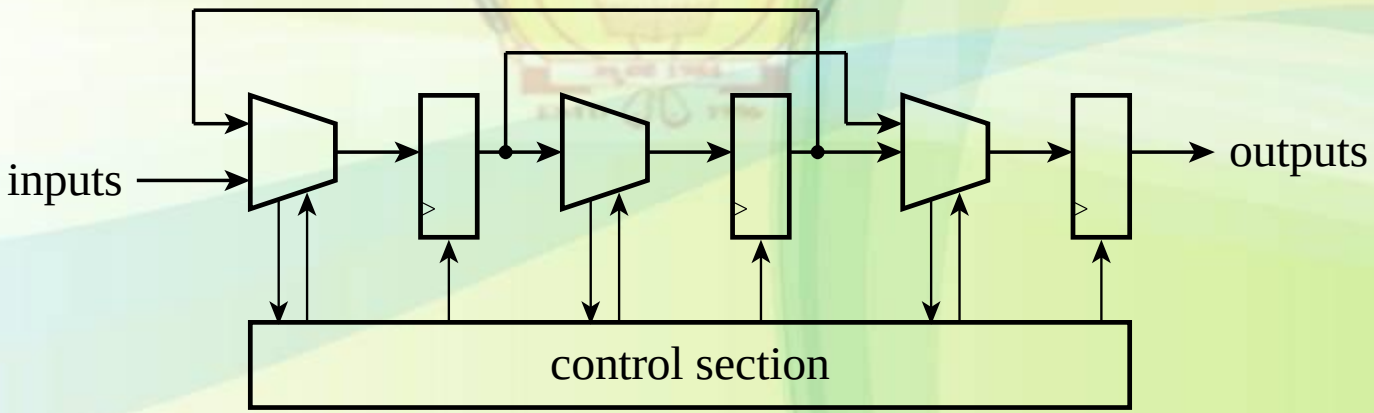


# Bubble Diagrams or Verilog?

- Many CAD tools provide editors for bubble diagrams
  - Automatically generate Verilog for simulation and synthesis
- Diagrams are visually appealing
  - but can become unwieldy for complex FSMs
- Your choice...
  - or your manager's!

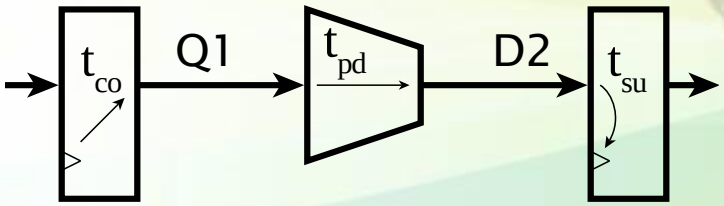
# Register Transfer Level

- RTL — a level of abstraction
  - data stored in registers
  - transferred via circuits that operate on data

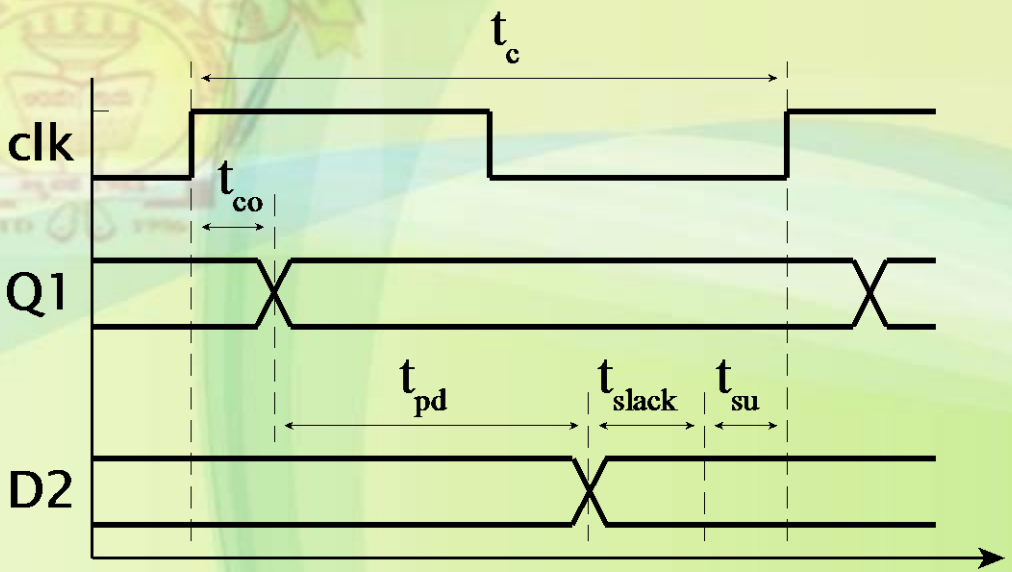


# Clocked Synchronous Timing

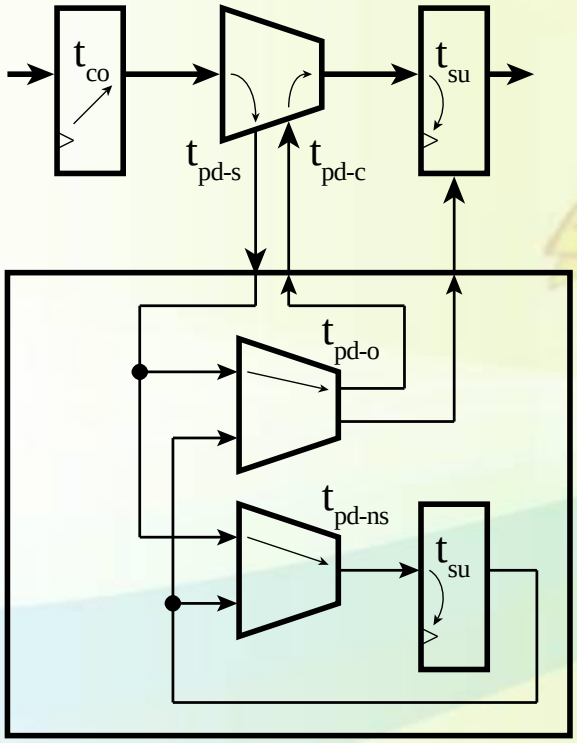
- Registers driven by a common clock
  - Combinational circuits operate during clock cycles (between rising clock edges)




$$t_{co} + t_{pd} + t_{su} < t_c$$



# Control Path Timing




$$t_{co} + t_{pd-s} + t_{pd-o} + t_{pd-c} + t_{su} < t_c$$

$$t_{co} + t_{pd-s} + t_{pd-ns} + t_{su} < t_c$$

Ignore $t_{pd-s}$ for a Moore FSM

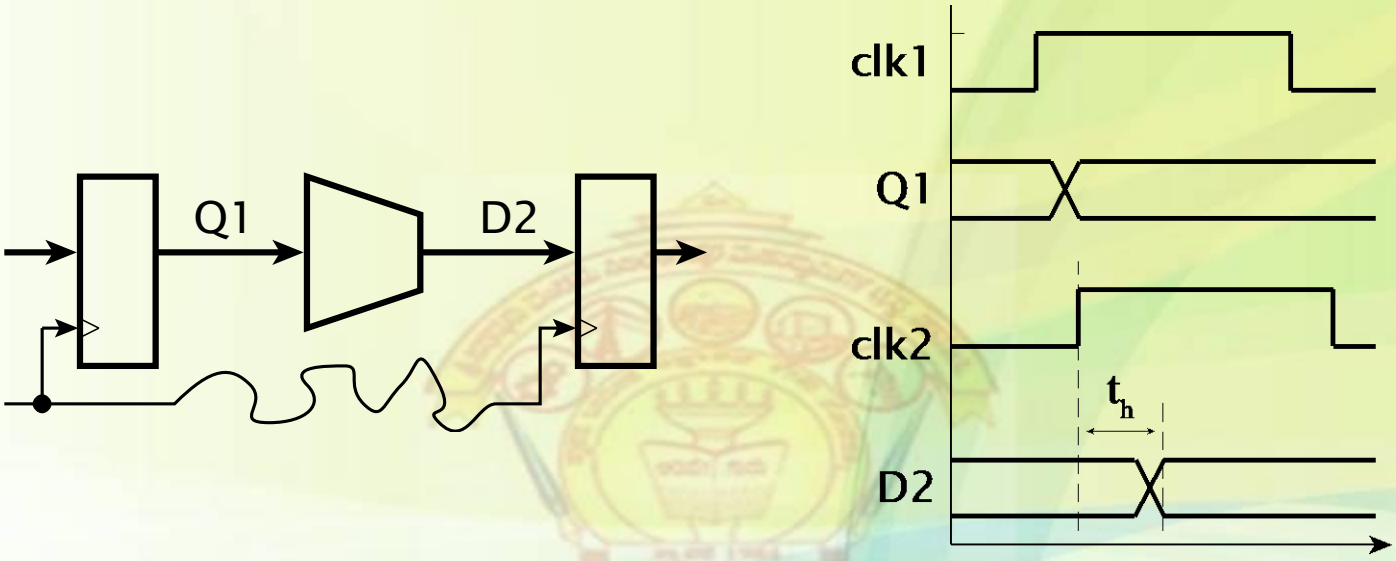
# Timing Constraints

- Inequalities must hold for all paths
- If $t_{co}$ and $t_{su}$ the same for all paths
  - Combinational delays make the difference
- *Critical path*
  - The combinational path between registers with the longest delay
  - Determines minimum clock period for the entire system
- Focus on it to improve performance
  - Reducing delay may make another path critical

# Interpretation of Constraints

1. Clock period depends on delays
   - System can operate at any frequency up to a maximum
   - OK for systems where high performance is not the main requirement
2. Delays must fit within a target clock period
   - Optimize critical paths to reduce delays if necessary
   - May require revising RTL organization

# Clock Skew




- Need to ensure clock edges arrive at all registers at the same time
  - Use CAD tools to insert clock buffers and route clock signal paths

# Off-Chip Connections

- Delays going off-chip and inter-chip
  - Input and output pad delays, wire delays
- Same timing rules apply
  - Use input and output registers to avoid adding external delay to critical path

# Asynchronous Inputs

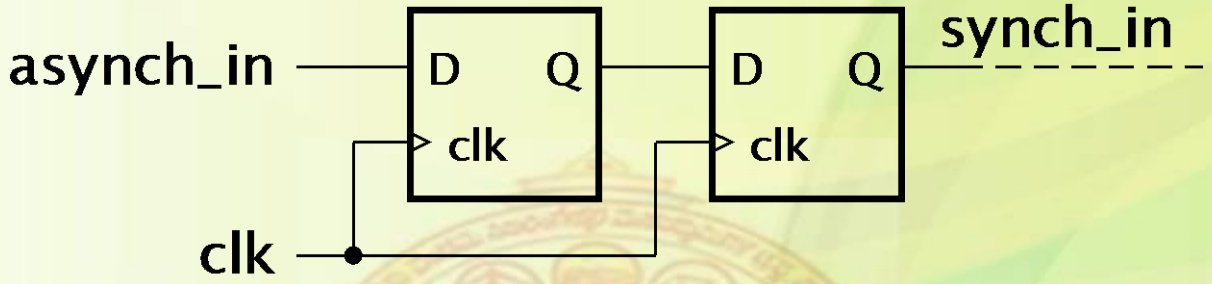
- External inputs can change at any time
  - Might violate setup/hold time constraints
- Can induce *metastable state* in a flipflop


- Unbounded time to recover
  - May violate setup/hold time of subsequent flipflop

$$MTBF = \frac{e^{k_2 t}}{k_1 f_f f_2}$$

$$k_2 >> 0$$

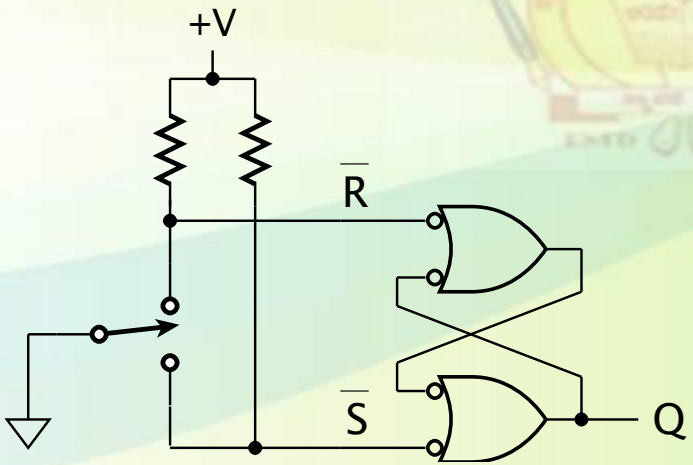
# Synchronizers




- If input changes outside setup/hold window
  - Change is simply delayed by one cycle
- If input changes during setup/hold window
  - First flipflop has a whole cycle to resolve metastability
- See data sheets for metastability parameters

# Switch Inputs and Debouncing

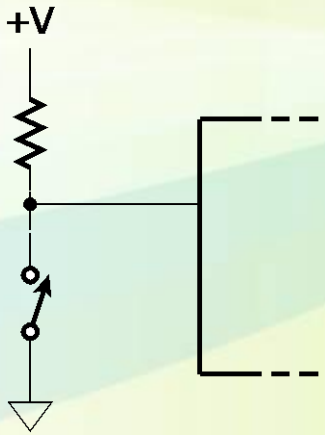
- Switches and push-buttons suffer from contact bounce
  - Takes up to 10ms to settle
- Need to *debounce* to avoid false triggering




- Requires two inputs and two resistors
- Must use a break-before-make double-throw switch

# Switch Inputs and Debouncing

- Alternative
  - Use a single-throw switch
  - Sample input at intervals longer than bounce time
  - Look for two successive samples with the same value




- Assumption
  - Extra circuitry inside the chip is cheaper than extra components and connections outside

# Queries ...?