EECS 151/251A
Fall 2018
Digital Design and Integrated Circuits

Instructor:
John Wawrzynek & Nicholas Weaver

Lecture 7

Announcements...

- Midterm, in the class slot, February 15th... ;(

Procedural Assignments

The sequential semantics of the blocking assignment allows variables to be multiply assigned within a single always block. Unexpected behavior can result from mixing these assignments in a single block. Standard rules:

i. Use blocking assignments to model combinational logic within an always block (“=”).
ii. Use non-blocking assignments to implement sequential logic (“<=”).
iii. Do not mix blocking and non-blocking assignments in the same always block.
iv. Do not make assignments to the same variable from more than one always block.

Combinational logic always blocks

Make sure all signals assigned in a combinational always block are explicitly assigned values every time that the always block executes. Otherwise latches will be generated to hold the last value for the signals not assigned values.

Sel case value 2'd2 omitted.

Out is not updated when select line has 2'd2.

Latch is added by tool to hold the last value of out under this condition.

Similar problem with if-else statements.

```
module mux4to1 (out, a, b, c, d, sel);
output out;
input a, b, c, d;
input [1:0] sel;
reg out;
always @(sel or a or b or c or d)
begin
  case (sel)
    2'd0: out = a;
    2'd1: out = b;
    2'd3: out = d;
  endcase
end
endmodule
```

Combinational logic always blocks (cont.)

To avoid synthesizing a latch in this case, add the missing select line:

```
2'd2: out = c;
```

Or, in general, use the "default" case:

```
default:  out = foo;
```

If you don't care about the assignment in a case (for instance you know that it will never come up) then you can assign the value "x" to the variable. Example:

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

The x is treated as a "don't care" for synthesis and will simplify the logic.

Be careful when assigning x (don't care). If this case were to come up, then the synthesized circuit and simulation **may differ**.

Special Values: x and z...

- 1'bx == single bit of "don't care" for synthesis
  - Result can actually become anything... 0... 1... bouncing up and down, etc etc etc...
    - It gives freedom to the compiler/simulator to do whatever it wants
    - Simulator: Usually process it as an explicit X output
    - Compiler: Optimizes it to produce either a 0, 1, or Z depending on everything else in the circuit
- 1'bz == "High impedance"
  - Result is "disconnect": ***nothing*** is driving this output
  - Needed to implement connections that can be both input & output

FSM Comparison

Solution A

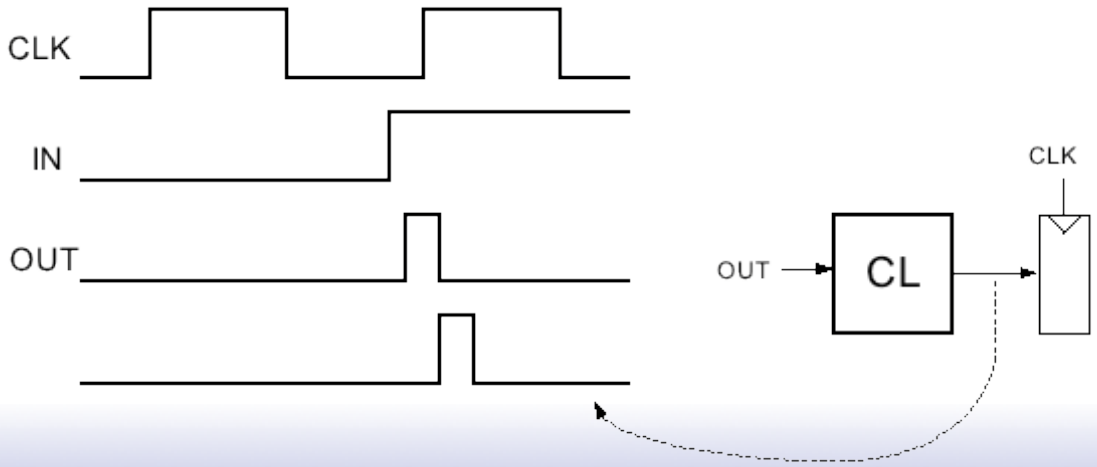
Moore Machine

- output function only of PS
- maybe more states (why?)
- synchronous outputs
  - Input glitches not send at output
  - one cycle “delay”
  - full cycle of stable output

Solution B

Mealy Machine

- *output function of both PS & input*
- *maybe fewer states*
- *asynchronous outputs*
  - *if input glitches, so does output*
  - *output immediately available*
  - *output may not be stable long enough to be useful (below):*



If output of Mealy FSM goes through combinational logic before being registered, the CL might delay the signal and it could be missed by the clock edge.

General FSM Design Process with Verilog Implementation

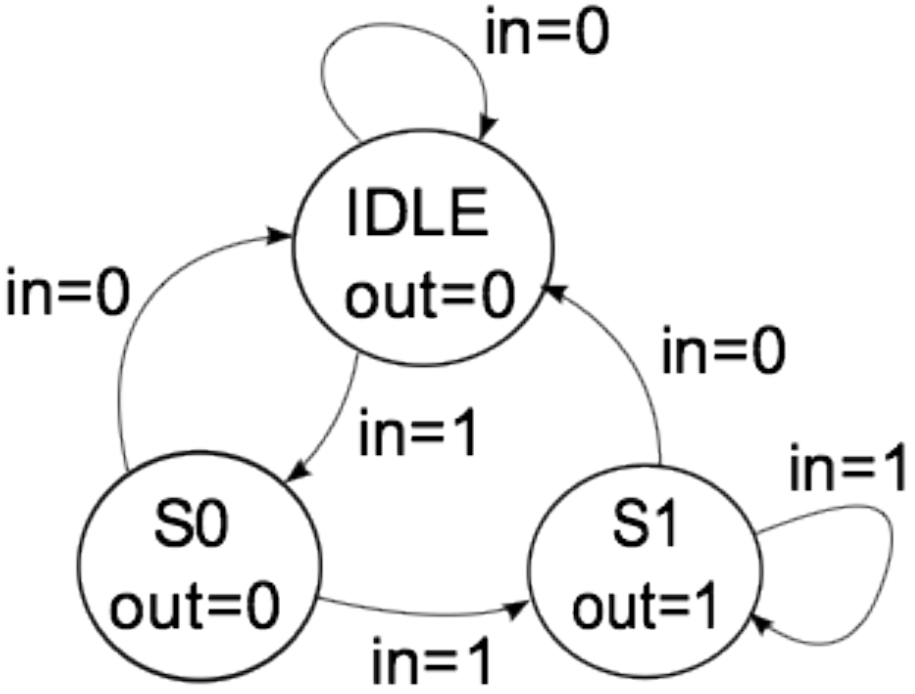
Design Steps:

1. *Specify **circuit function** (English)*
2. *Draw **state transition diagram***
3. *Write down **symbolic state transition table***
4. *Assign encodings (bit patterns) to symbolic states*
5. *Code as Verilog behavioral description.. (Or just skip from 1 to 4+5...)*

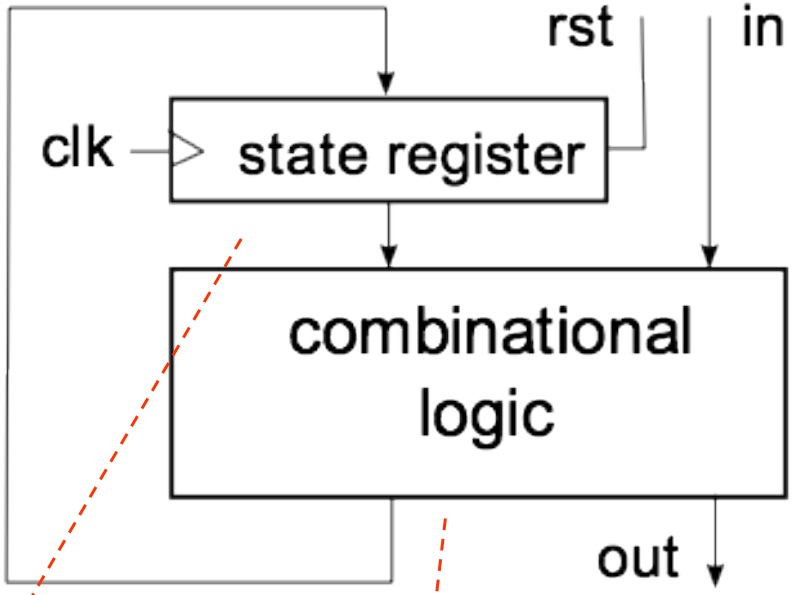
- ✓ *Use parameters to represent encoded states.*
- ✓ *Use separate always blocks for register assignment and CL logic block.*
- ✓ *Use case for CL block. Within each case section (state) assign all outputs and next state value based on inputs. Note: For Moore style machine make outputs dependent only on state not dependent on inputs.*

Finite State Machine in Verilog

State Transition Diagram



Implementation Circuit Diagram



Holds a symbol to keep track of which bubble the FSM is in.

CL functions to determine output value and next state based on input and current state.

out = f(in, current state)

next state = f(in, current state)

Finite State Machines

```
module FSM1(clk, rst, in, out);
input clk, rst;
input in;
output out;

// Defined state encoding:
parameter IDLE = 2'b00;
parameter S0 = 2'b01;
parameter S1 = 2'b10;
reg out;
reg [1:0] present_state, next_state;

// always block for state register
always @(posedge clk)
      if (rst) present_state <= IDLE;
      else present_state <= next_state;
```

Must use reset to force to initial state.

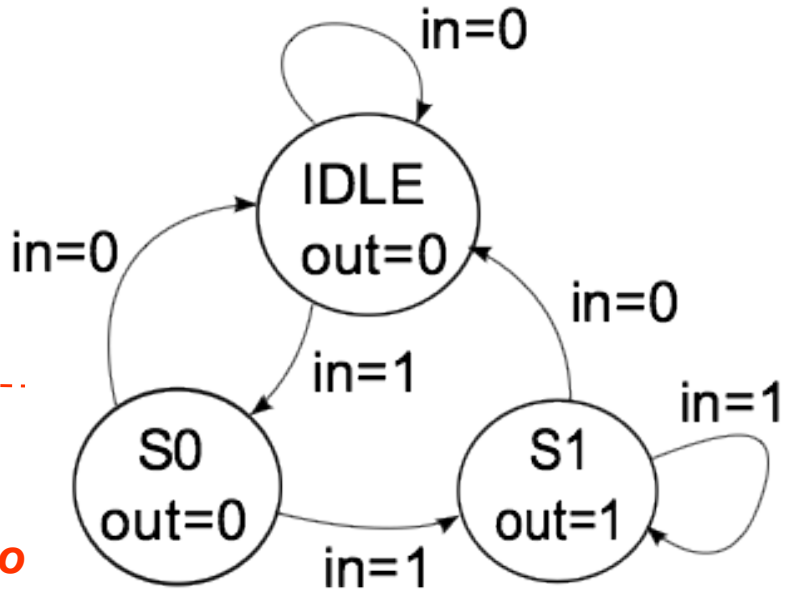
reset not always shown in STD

Constants local to this module.

out not a register, but assigned in always block

Combinational logic signals for transition.

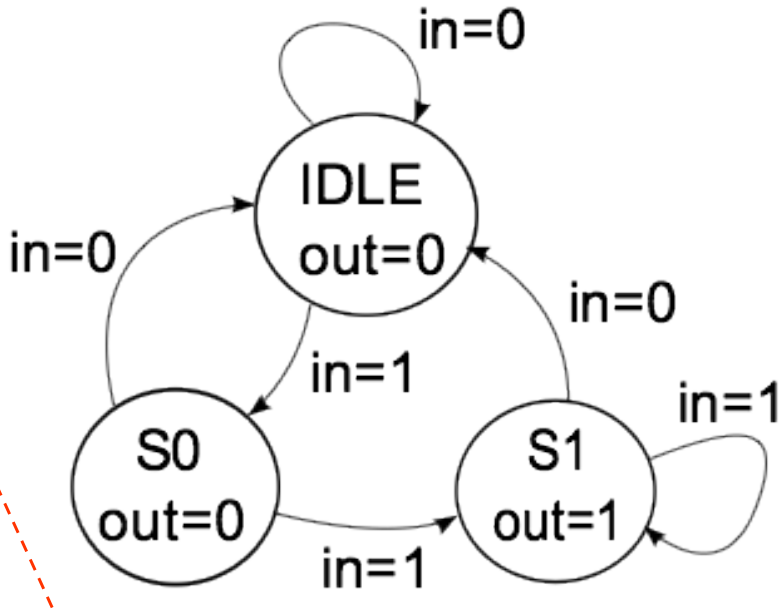
THE register to hold the "state" of the FSM.



A separate always block should be used for combination logic part of FSM. Next state and output generation. (Always blocks in a design work in parallel.)

FSMs (cont.)

```
// always block for combinational logic portion
always @(present_state or in)
case (present_state)
// For each state def output and next
  IDLE    : begin
              out = 1'b0;
              if (in == 1'b1) next_state = S0;
              else next_state = IDLE;
            end
  S0      : begin
              out = 1'b0;
              if (in == 1'b1) next_state = S1;
              else next_state = IDLE;
            end
  S1      : begin
              out = 1'b1;
              if (in == 1'b1) next_state = S1;
              else next_state = IDLE;
          end
  default: begin
              next_state = IDLE;
              out = 1'b0;
            end
endcase
endmodule
```



Each state becomes a case clause.

For each state define:

Output value(s)

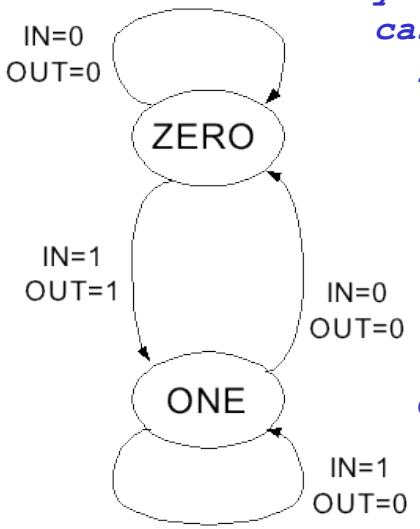
State transition

Use "default" to cover unassigned state. Usually unconditionally transition to reset state.

Mealy or Moore?

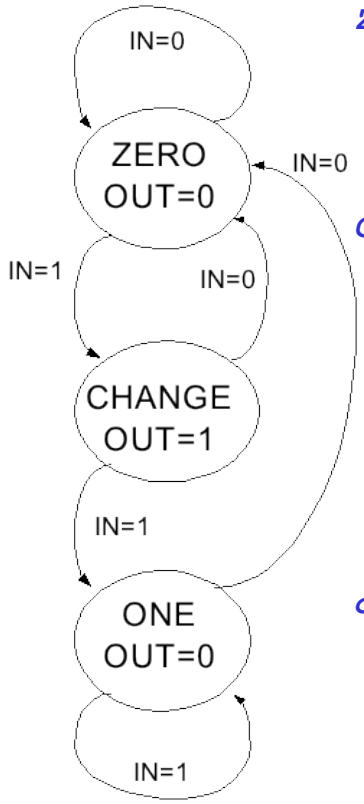
Edge Detector Example

Mealy Machine




```
always @(posedge clk)
          if (rst) ps <= ZERO;
          else ps <= ns;
always @(ps in)
     case (ps)
       ZERO: if (in) begin
              out = 1'b1;
              ns = ONE;
             end
              else begin
                out = 1'b0;
                ns = ZERO;
              end
       ONE: if (in) begin
              out = 1'b0;
              ns = ONE;
             end
             else begin
               out = 1'b0;
               ns = ZERO;
             end
       default: begin
                 out = 1'bx;
                 ns = default;
                end
```

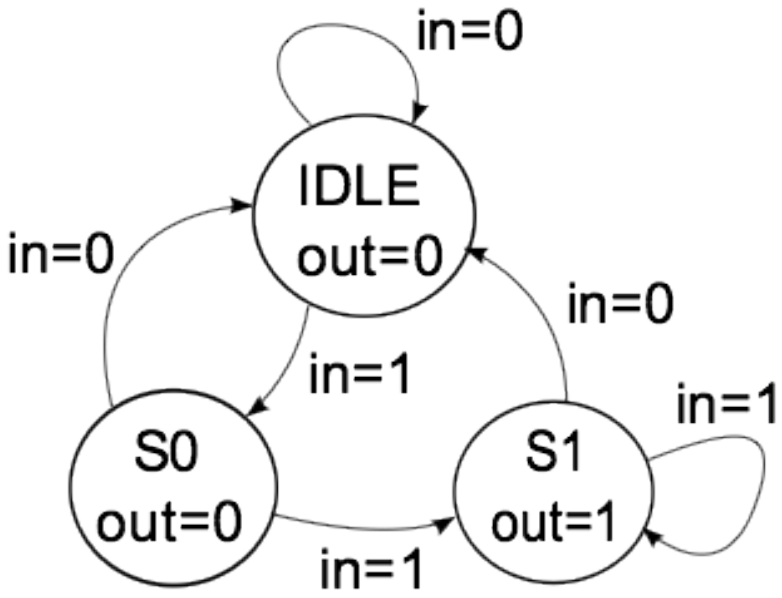
Moore Machine



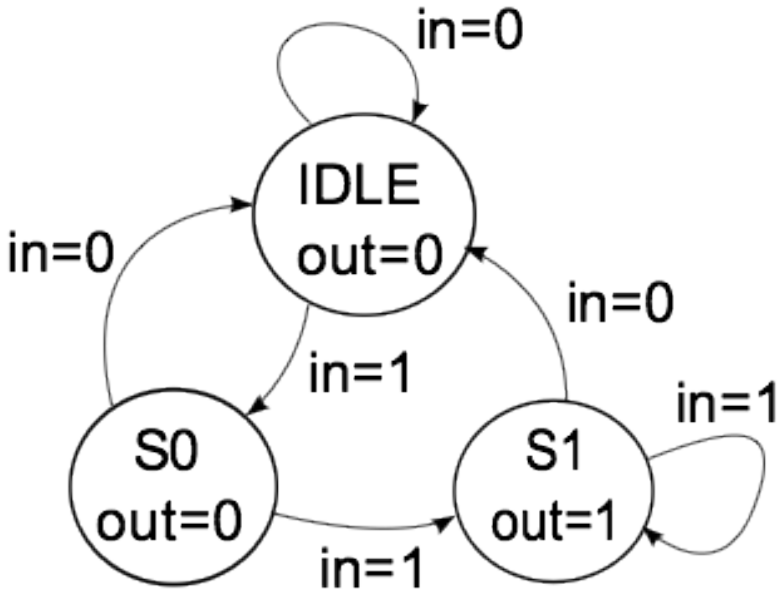

```
always @(posedge clk)
          if (rst) ps <= ZERO;
          else ps <= ns;
always @(ps in)
     case (ps)
       ZERO: begin
              out = 1'b0;
              if (in) ns = CHANGE;
               else ns = ZERO;
             end
     CHANGE: begin
              out = 1'b1;
              if (in) ns = ONE;
              else ns = ZERO;
             end
        ONE: begin
              out = 1'b0;
              if (in) ns = ONE;
              else ns = ZERO;
     default: begin
              out = 1'bx;
              ns = default;
             end
```

FSM CL block (original)

```
always @(present_state or in)
 case (present_state)
  IDLE    : begin
              out = 1'b0;
              if (in == 1'b1) next_state = S0;
              else next_state = IDLE;
            end
  S0      : begin
              out = 1'b0;
              if (in == 1'b1) next_state = S1;
              else next_state = IDLE;
            end
  S1      : begin
              out = 1'b1;
              if (in == 1'b1) next_state = S1;
              else next_state = IDLE;

             end
  default: begin
              next_state = IDLE;
              out = 1'b0;
            end
 endcase
endmodule
```



FSM CL block rewritten




```
always @*
  begin
   next_state = IDLE;
   out = 1'b0;
    case (state)
     IDLE    : if (in == 1'b1) next_state = S0;
     S0      : if (in == 1'b1) next_state = S1;
     S1      : begin
                 out = 1'b1;
                 if (in == 1'b1) next_state = S1;
               end
     default: ;
    endcase
  end
Endmodule
```

** for sensitivity list*

Normal values: used unless specified below.

Within case only need to specify exceptions to the normal values.

Note: The use of "blocking assignments" allow signal values to be "rewritten", simplifying the specification.

Incomplete Triggers

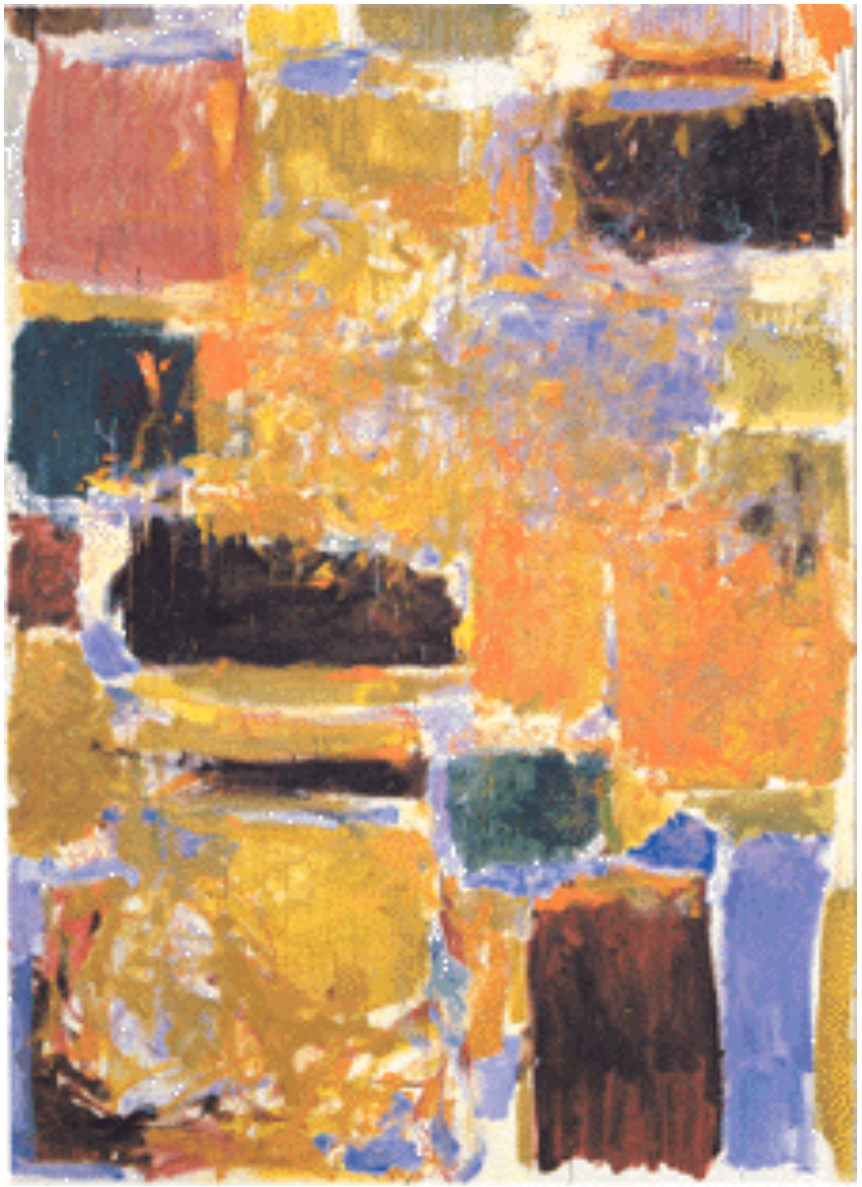
Leaving out an input trigger usually results in latch generation for the missing trigger.

```
module and_gate (out, in1, in2);
  input    in1, in2;
  output       out;
  reg          out;

  always @(in1) begin
     out = in1 & in2;
  end

endmodule
```

in2 not in always sensitivity list.

A latched version of in2 is synthesized and used as input to the and-gate, so that the and-gate output is not always sensitive to in2.

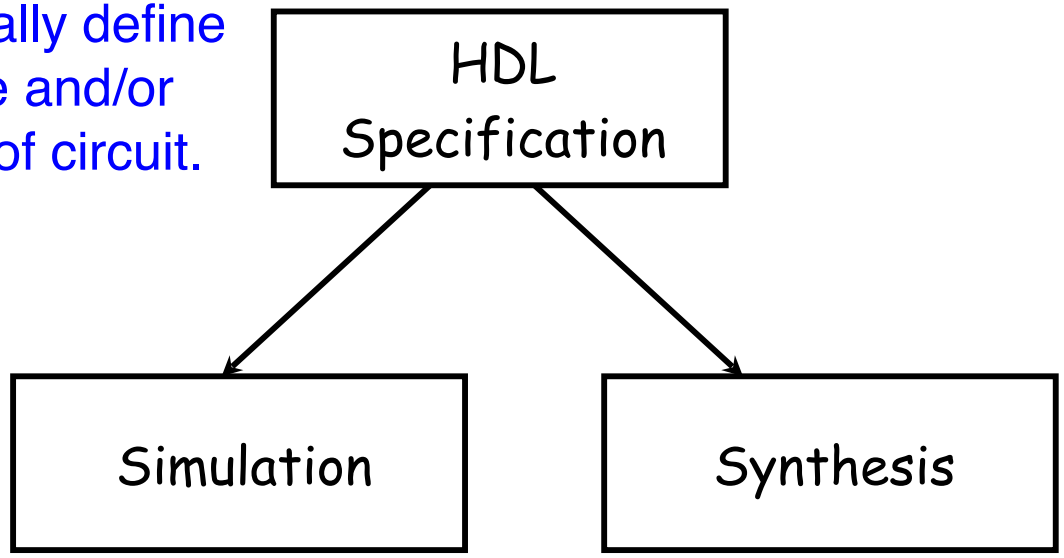
Easy way to avoid incomplete triggers for combinational logic is with: ***always @****



Intro to Logic Synthesis

EECS151/251A Design Methodology

Hierarchically define structure and/or behavior of circuit.



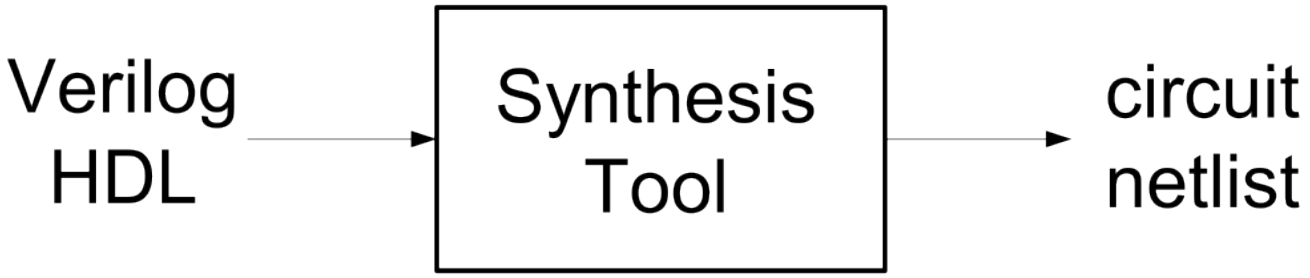
Functional verification.

Maps specification to resources of implementation platform (FPGA or ASIC).

Note: This in not the entire story. Other tools are often used analyze HDL specifications and synthesis results. More on this later.

Logic Synthesis

- Verilog and VHDL started out as simulation languages, but quickly people wrote programs to automatically convert Verilog code into low-level circuit descriptions (netlists).



- Synthesis converts Verilog (or other HDL) descriptions to implementation technology specific primitives:
  - For FPGAs: LUTs, flip-flops, and RAM blocks
  - For ASICs: standard cell gate and flip-flop libraries, and memory blocks.

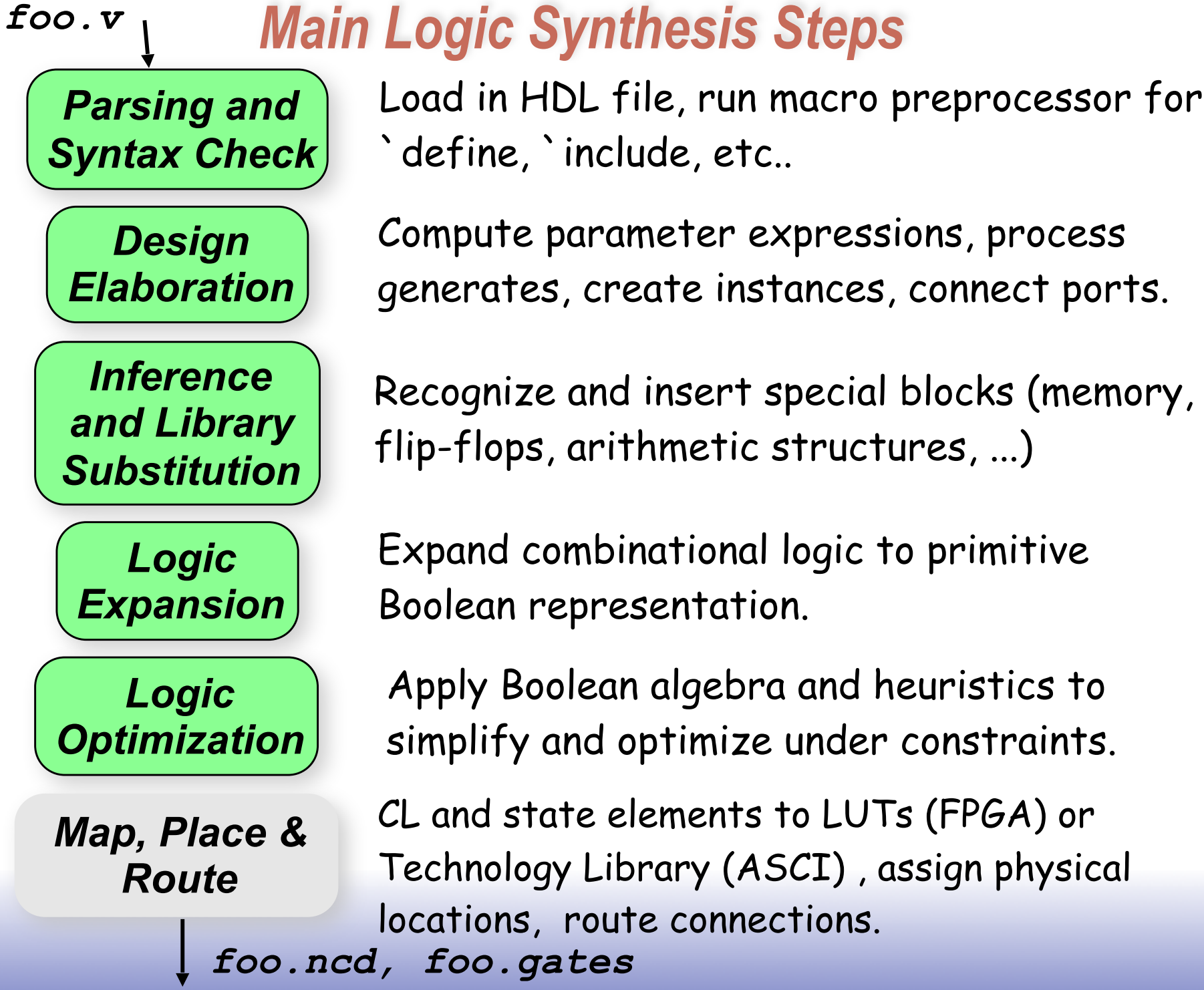
Why Logic Synthesis?

1. Automatically manages many details of the design process:
   - ⇒ Fewer bugs
   - ⇒ Improved productivity
2. Abstracts the design data (HDL description) from any particular implementation technology.
   - Designs can be re-synthesized targeting different chip technologies. Ex: first implement in FPGA then later in ASIC.
3. In some cases, leads to a more optimal design than could be achieved by manual means (ex: logic optimization)

Why Not Logic Synthesis?

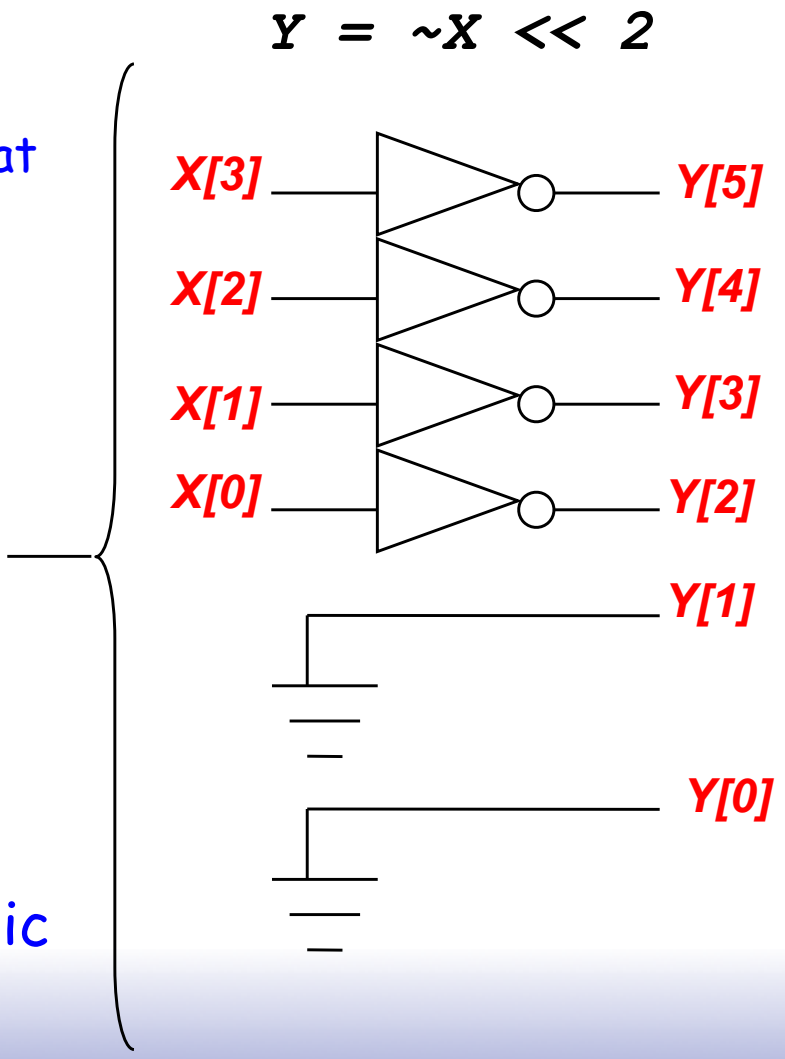
1. *May lead to non-optimal designs in some cases.*
2. *Often less transparent than desired: Good performance requires basically modeling the compiler in your head…*

Main Logic Synthesis Steps



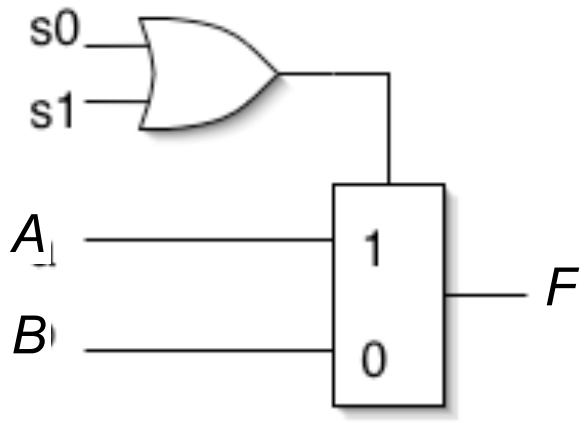
Operators and Synthesis

- Logical operators map into primitive logic gates
- Arithmetic operators map into adders, subtractors, …
  - Unsigned 2s complement
  - Model carry: target is one-bit wider that source
  - Watch out for *, %, and /
- Relational operators generate comparators
- Shifts by constant amount are just wire connections
  - No logic involved
- Variable shift amounts, a whole different story --- shifters are **expensive!**
- Conditional expression generates logic or MUX



Simple Synthesis Example

```
module foo (A, B, s0, s1, F);
  input [3:0] A;
  input [3:0] B;
  input s0,s1;
  output [3:0] F;
  reg F;
  always @ (*)
    if (!s0 && s1 || s0) F=A; else F=B;
endmodule
```



Should expand if-else into 4-bit wide multiplexor and optimize the control logic and ultimately to 4 4-LUT on an FPGA:

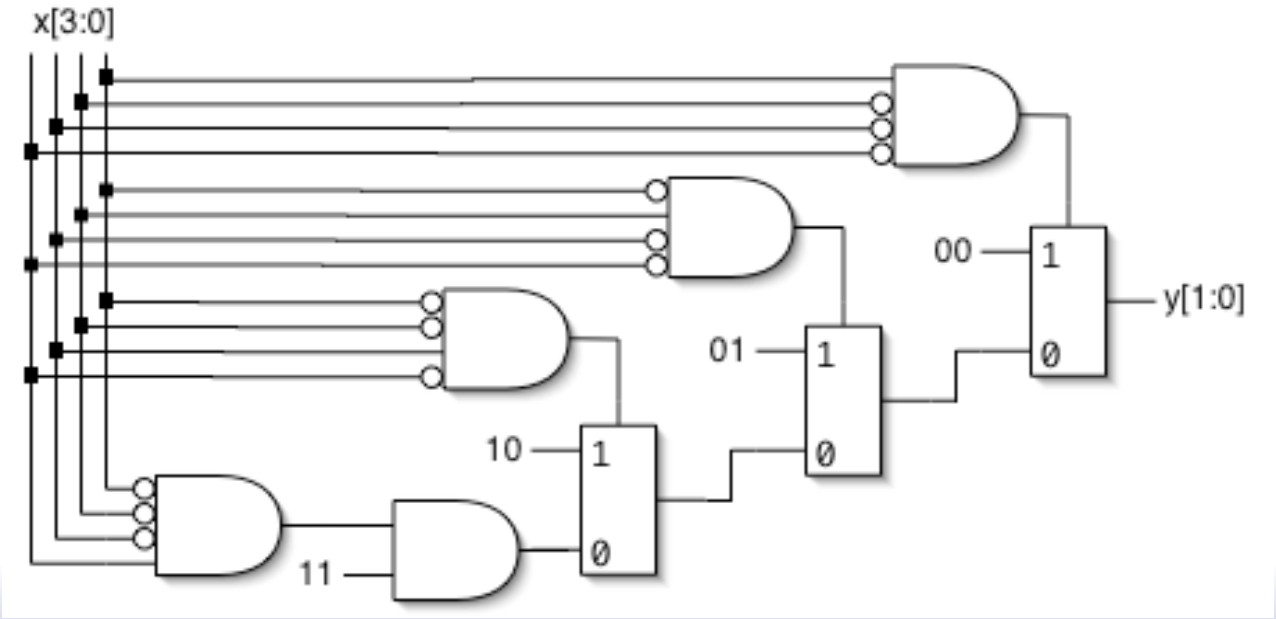
Encoder Example

Nested IF-ELSE might lead to "priority logic"

Example: 4-to-2 encoder

```
always @(x)
begin : encode
if (x == 4'b0001) y = 2'b00;
else if (x == 4'b0010) y = 2'b01;
else if (x == 4'b0100) y = 2'b10;
else if (x == 4'b1000) y = 2'b11;
else y = 2'bxx;
end
```

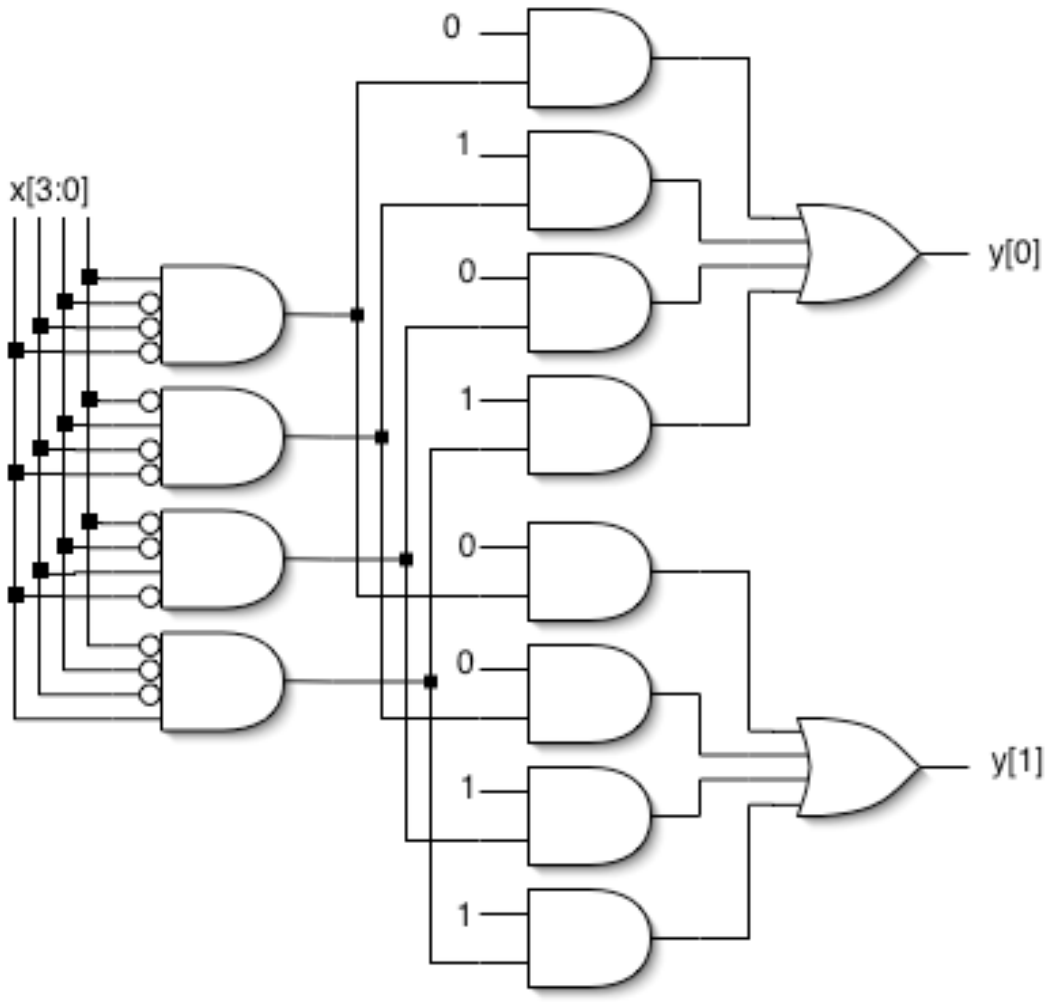
This style of cascaded logic may adversely affect the performance of the circuit.



Encoder Example (cont.)

To avoid "priority logic" use the case construct:

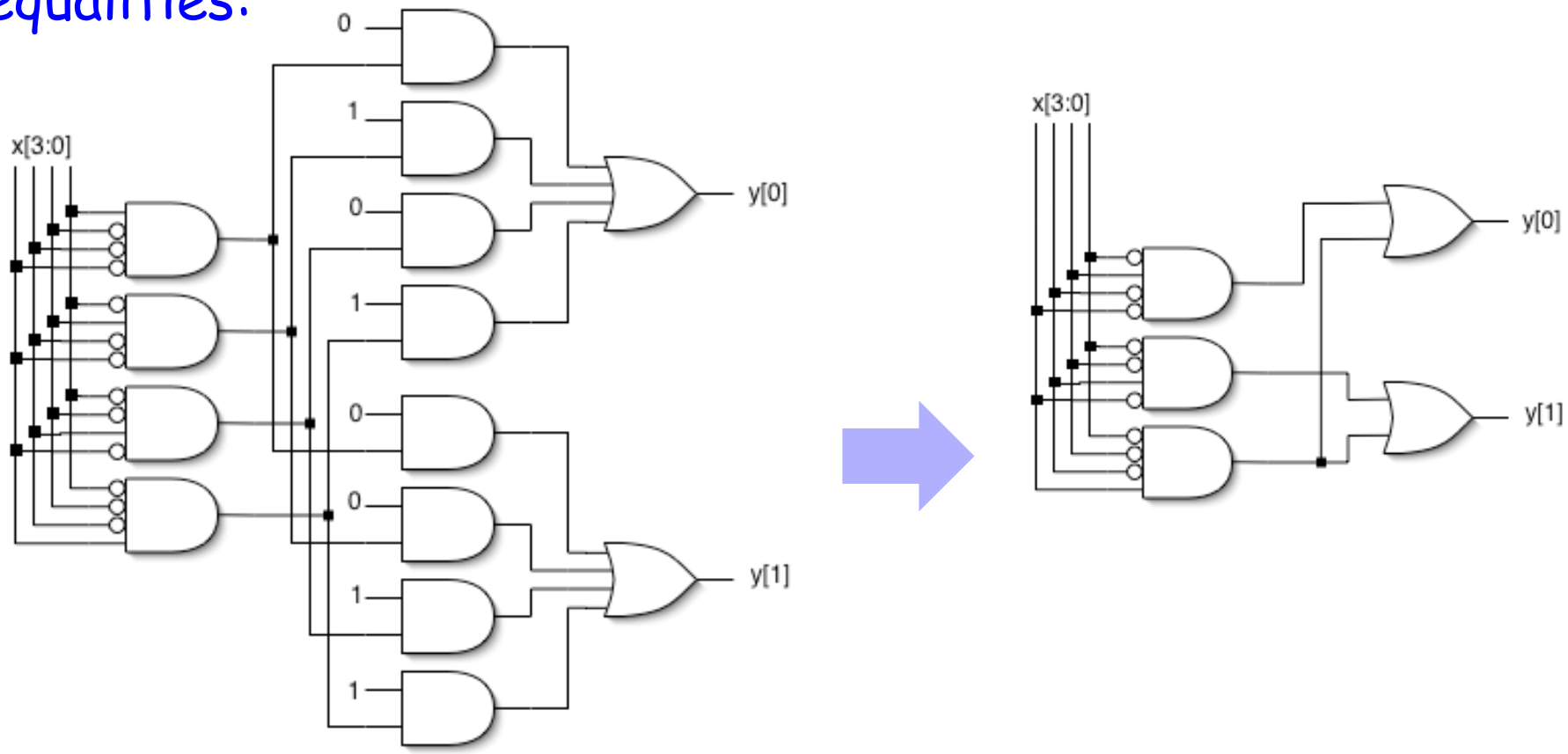
```
always @(x)
begin : encode
case (x)
4'b0001: y = 2'b00;
4'b0010: y = 2'b01;
4'b0100: y = 2'b10;
4'b1000: y = 2'b11;
default: y = 2'bxx;
endcase
end
```



All cases are matched in parallel.

Encoder Example (cont.)

This circuit would be simplified during synthesis to take advantage of constant values as follows and other Boolean equalities:



A similar simplification would be applied to the if-else version also.

More On Karnaugh Maps

- Realize I should have covered this in more detail...
  - So lets snag some older slides...

Algorithmic Two-level Logic Simplication

Key tool: The Uniting Theorem:

$$xy' + xy = x\,(y' + y) = x\,(1) = x$$

ab	f
00	0
01	0
10	1
11	1

$f = ab' + ab = a(b'+b) = a$

b values change within the on-set rows

a values don't change

b is eliminated, a remains

ab	g
00	1
01	0
10	1
11	0

$g = a'b'+ab' = (a'+a)b' = b'$

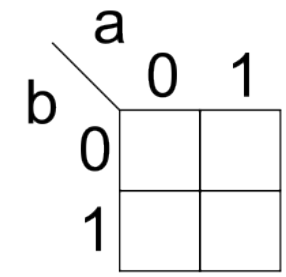
b values stay the same

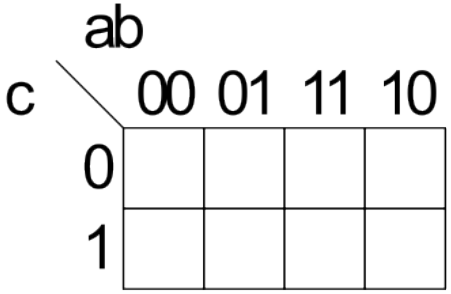
a values changes

b' remains, a is eliminated

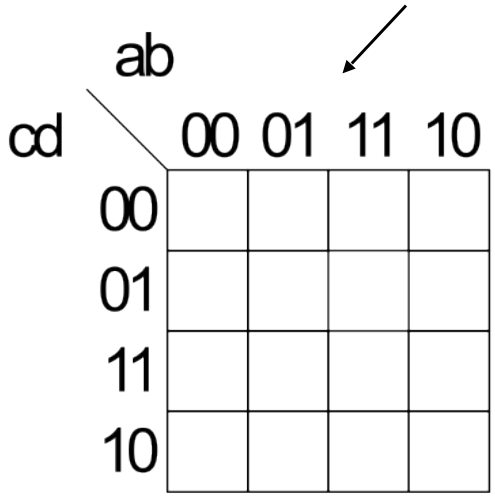
Karnaugh Map Method

- K-map is an alternative method of representing the TT and to help visual the adjacencies.





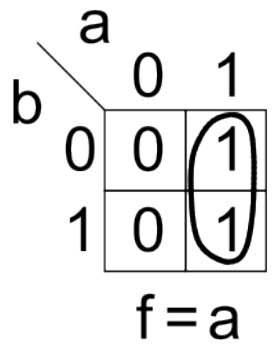
Note: "gray code" labeling.



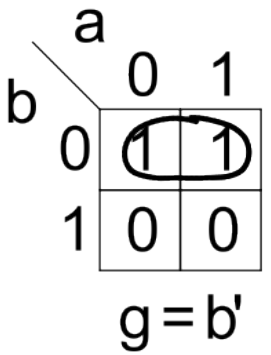
5 & 6 variable k-maps possible

Karnaugh Map Method

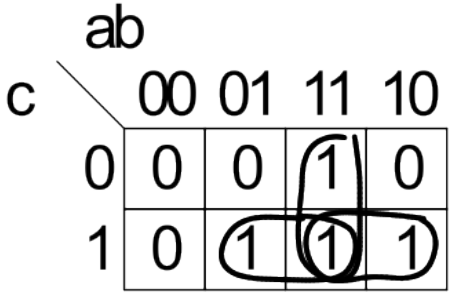
- Adjacent groups of 1's represent product terms



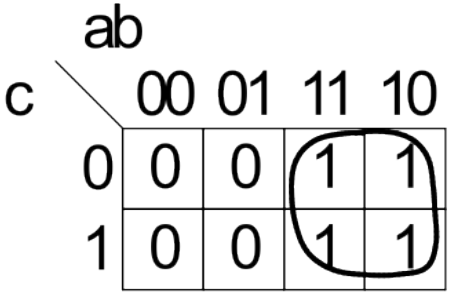
f = a



g = b'



cout = ab + bc + ac

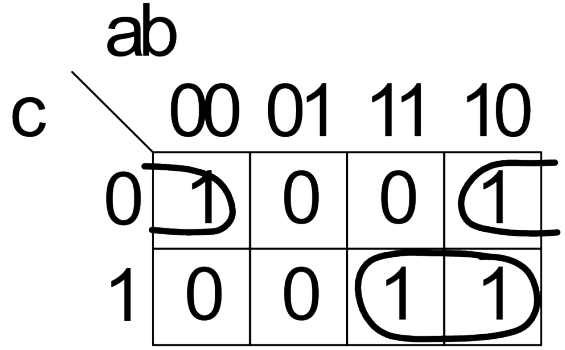


f = a

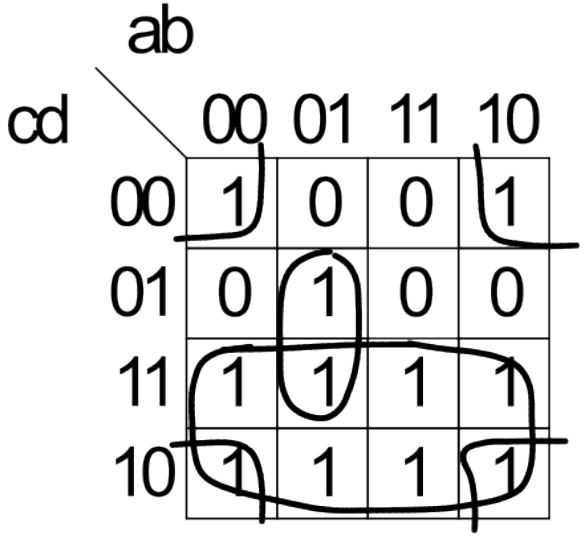
K-map Simplification

1. Draw K-map of the appropriate number of variables (between 2 and 6)
2. Fill in map with function values from truth table.
3. Form groups of 1's.
   - ✓ Dimensions of groups must be even powers of two (1x1, 1x2, 1x4, …, 2x2, 2x4, …)
   - ✓ Form as large as possible groups and as few groups as possible.
   - ✓ Groups can overlap (this helps make larger groups)
   - ✓ Remember K-map is periodical in all dimensions (groups can cross over edges of map and continue on other side)
4. For each group write a product term.
   - the term includes the "constant" variables (use the uncomplemented variable for a constant 1 and complemented variable for constant 0)
5. Form Boolean expression as sum-of-products.

K-maps (cont.)



f = b'c' + ac

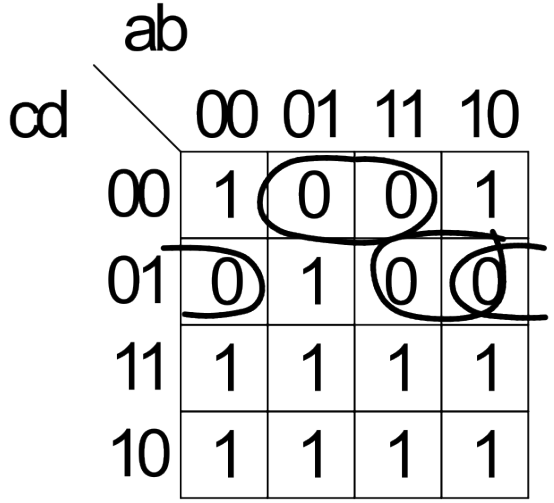


f = c + a'bd + b'd'

(bigger groups are better)

Product-of-Sums Version

1. Form groups of 0's instead of 1's.
2. For each group write a sum term.
   - the term includes the "constant" variables (use the uncomplemented variable for a constant 0 and complemented variable for constant 1)
3. Form Boolean expression as product-of-sums.

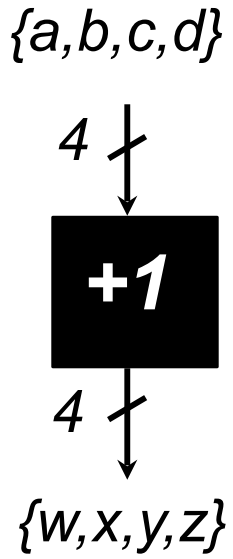


$$f = (b' + c + d)(a' + c + d')(b + c + d')$$

BCD incrementer example

Binary Coded Decimal

	a	b	c	d	w	x	y	z
0	0	0	0	0	0	0	0	1
1	0	0	0	1	0	0	1	0
2	0	0	1	0	0	0	1	1
3	0	0	1	1	0	1	0	0
4	0	1	0	0	0	1	0	1
5	0	1	0	1	0	1	1	0
6	0	1	1	0	0	1	1	1
7	0	1	1	1	1	0	0	0
8	1	0	0	0	1	0	0	1
9	1	0	0	1	0	0	0	0
	1	0	1	0	-	-	-	-
	1	0	1	1	-	-	-	-
	1	1	0	0	-	-	-	-
	1	1	0	1	-	-	-	-
	1	1	1	0	-	-	-	-
	1	1	1	1	-	-	-	-



BCD Incrementer Example

- Note one map for each output variable.
- Function includes “don’t cares” (shown as “-” in the table).
  - These correspond to places in the function where we don’t care about its value, because we don’t expect some particular input patterns.
  - We are free to assign either 0 or 1 to each don’t care in the function, as a means to increase group sizes.
- In general, you might choose to write product-of-sums or sum-of-products according to which one leads to a simpler expression.

BCD incrementer example

w

cd \ ab	00	01	11	10
00	0	0	-	1
01	0	0	-	0
11	0	1	-	-
10	0	0	-	-

x

cd \ ab	00	01	11	10
00	0	1	-	0
01	0	1	-	0
11	1	0	-	-
10	0	1	-	-

$w =$

$x =$

y

cd \ ab	00	01	11	10
00	0	0	-	0
01	1	1	-	0
11	0	0	-	-
10	1	1	-	-

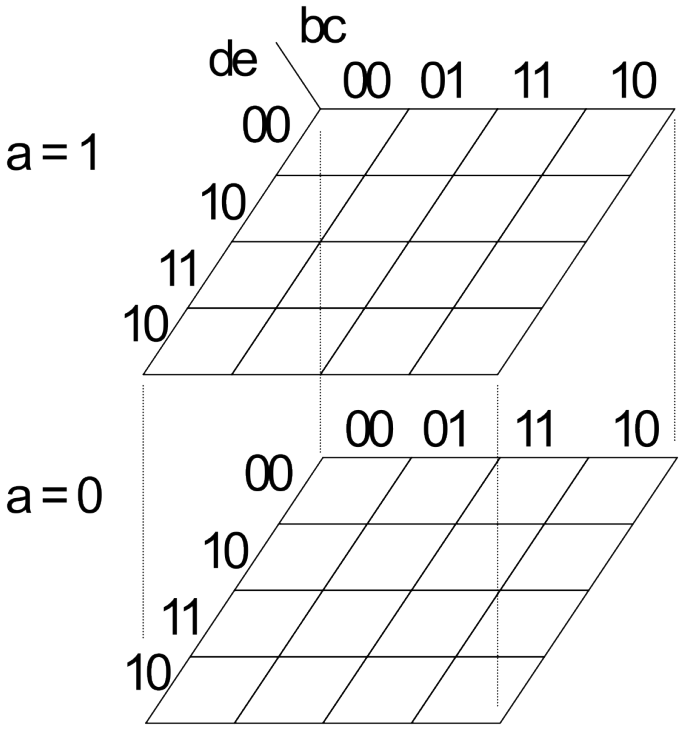
z

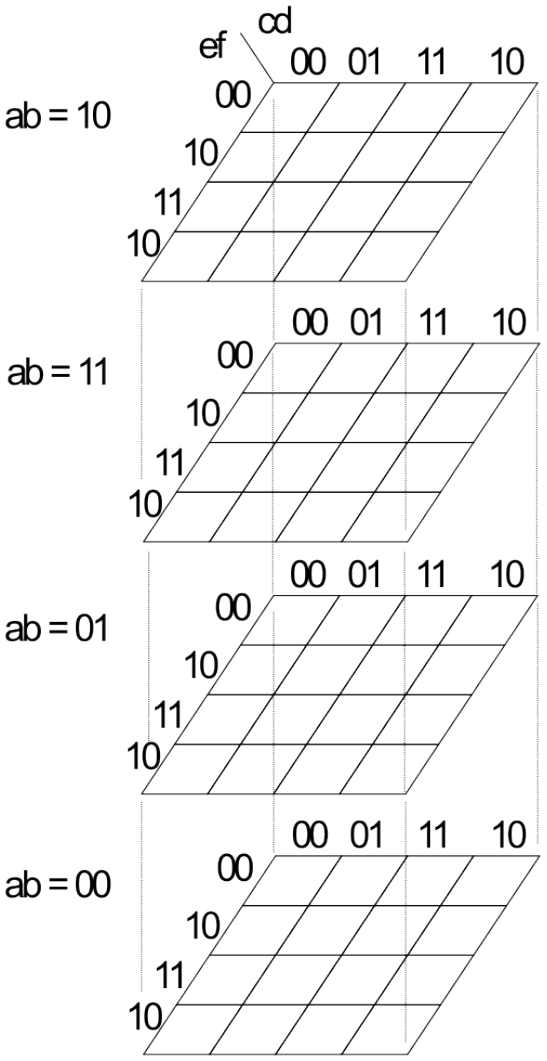
cd \ ab	00	01	11	10
00	1	1	-	1
01	0	0	-	0
11	0	0	-	-
10	1	1	-	-

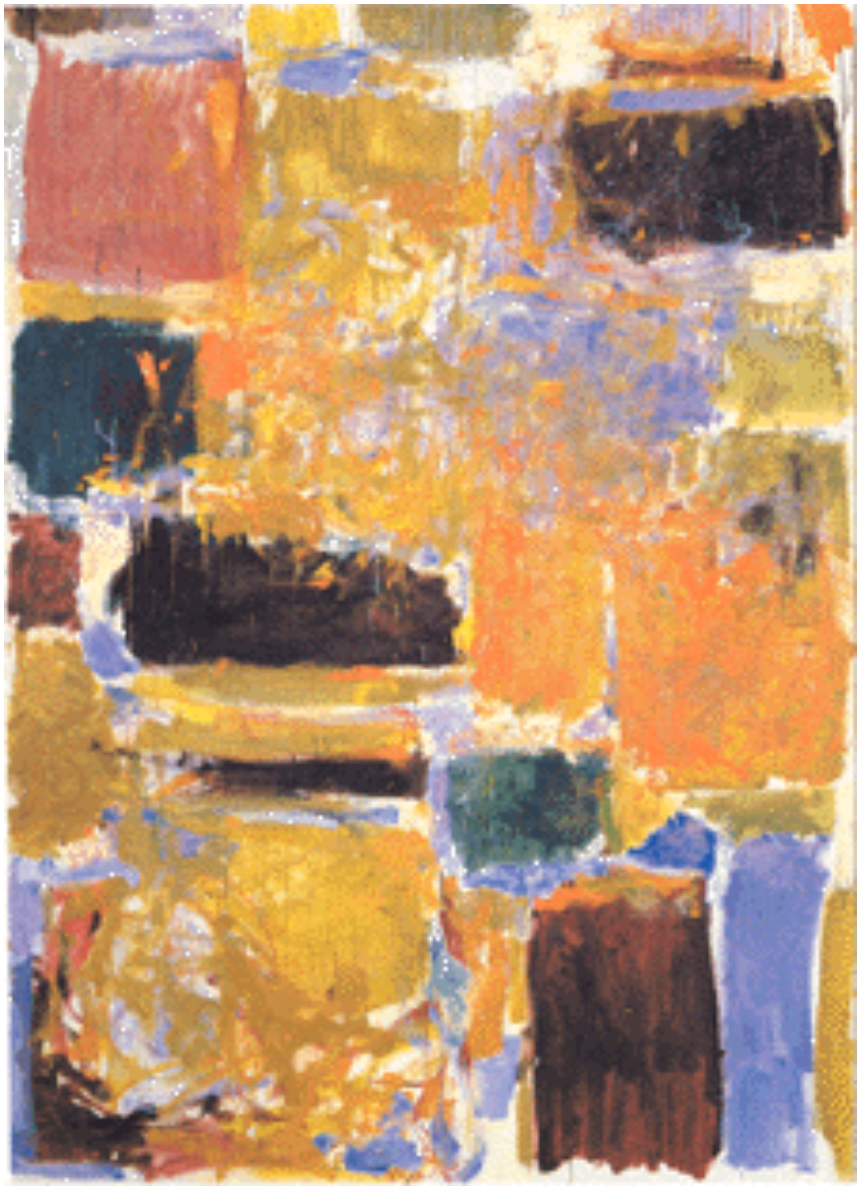
$y =$

$z =$

Higher Dimensional K-maps







Digital abstraction

Bridging the digital and the analog worlds

- How to represent 0's and 1's in a world that is analog?

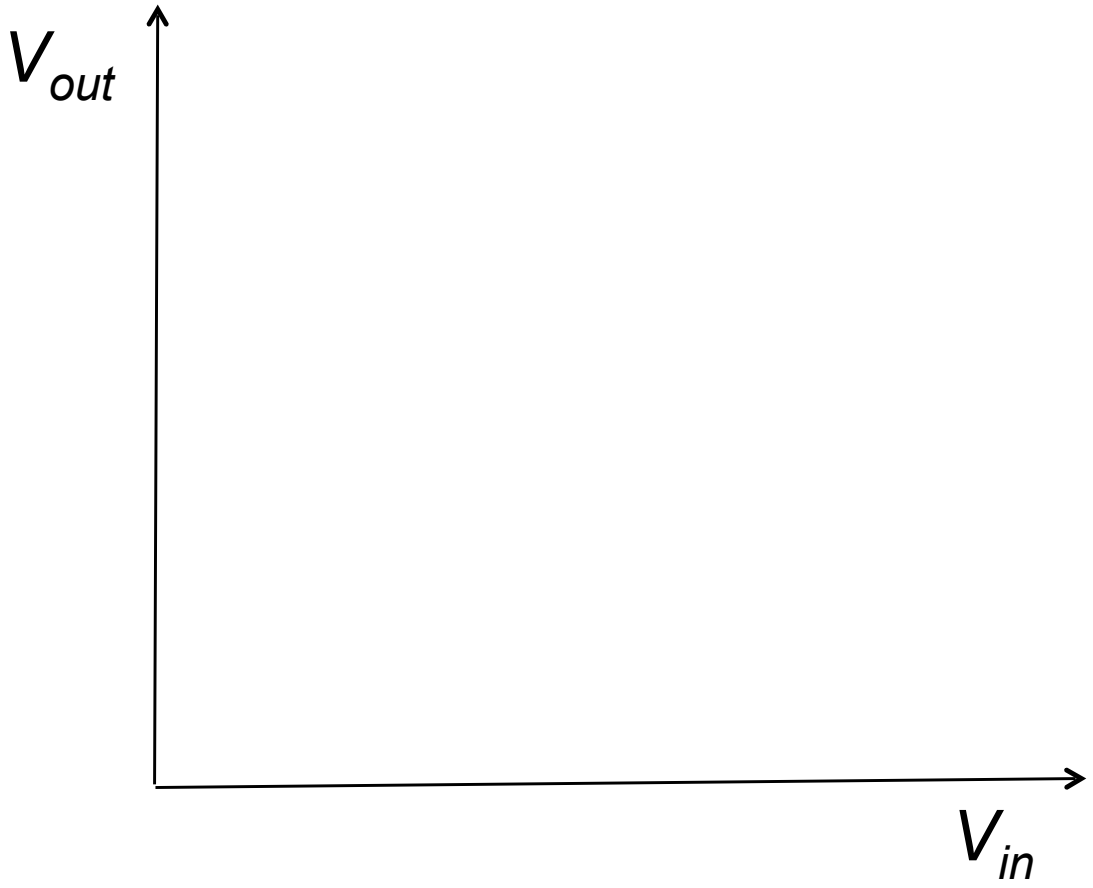
Noise and Digital Systems

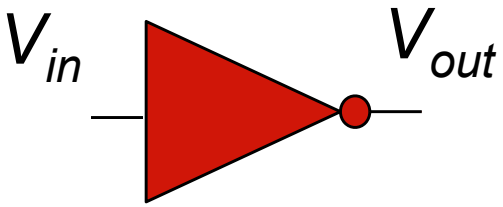
- Circuit needs to works despite "analog" noise
  - Digital gates can and must reject noise
  - This is actually how digital systems are defined

- Digital system is one where:
  - Discrete values mapped to analog levels and back
  - All the elements (gates) can reject noise
    - For "small" amounts of noise, output noise is less than input noise
  - Thus, for sufficiently "small" noise, the system acts as if it was noiseless
  - This is called regeneration

Noise?

Noise Rejection and the Voltage Transfer Characteristic

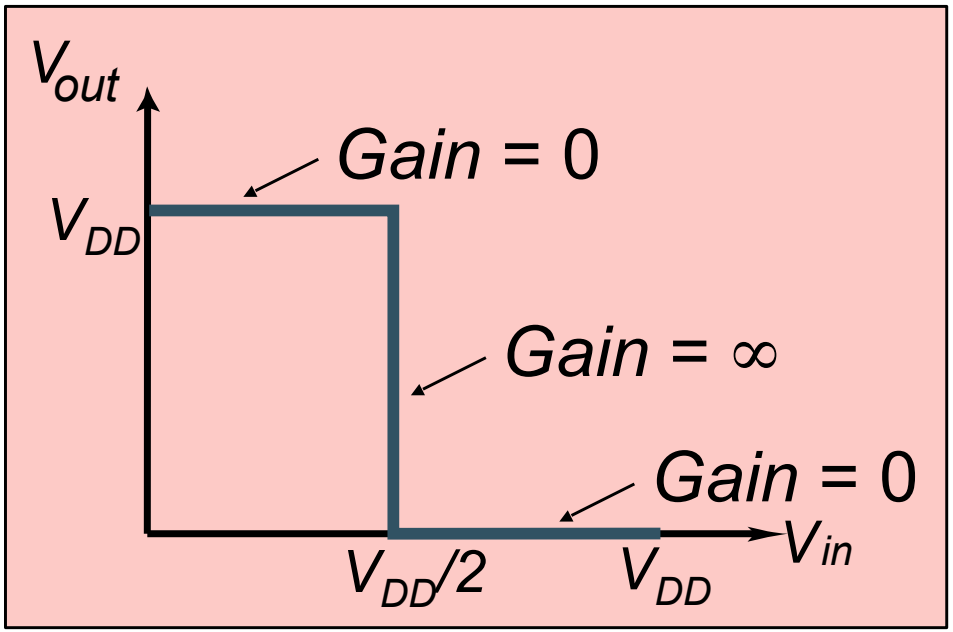




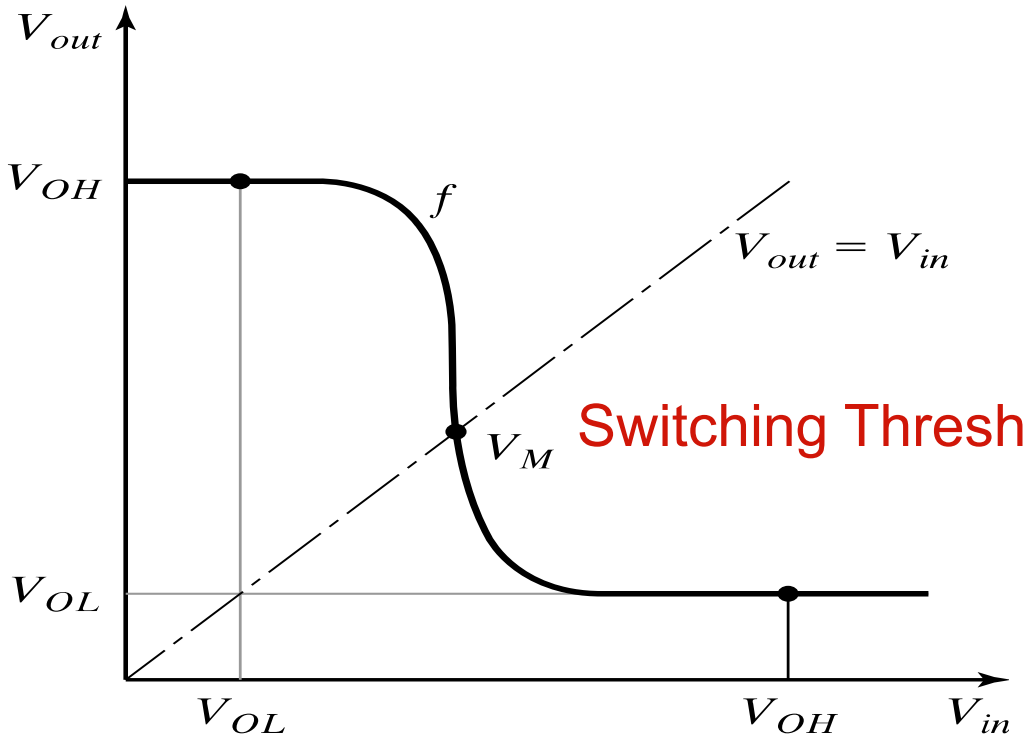
Noise Rejection

- To see if a gate rejects noise
  - Look at its DC voltage transfer characteristic (VTC)
  - See what happens when input is not exactly 1 or 0

- Ideal digital gate:
  - Noise needs to be larger than $V_{DD}/2$ to have any effect on gate output



Realistic Voltage Transfer Characteristic Definitions

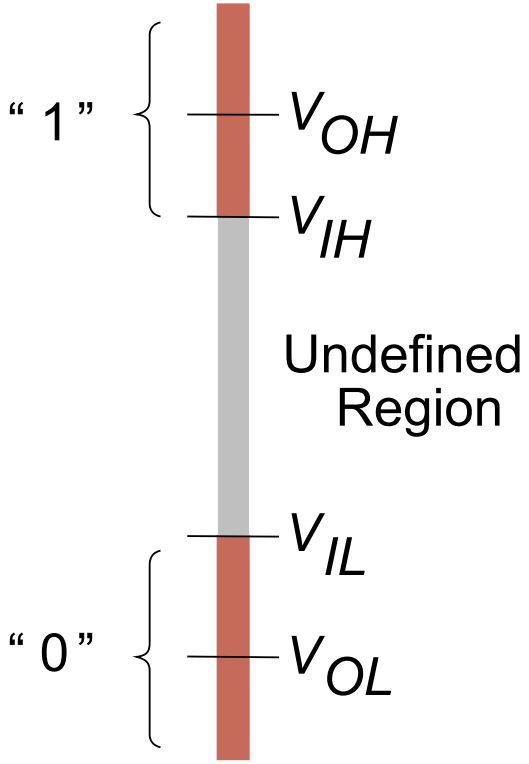


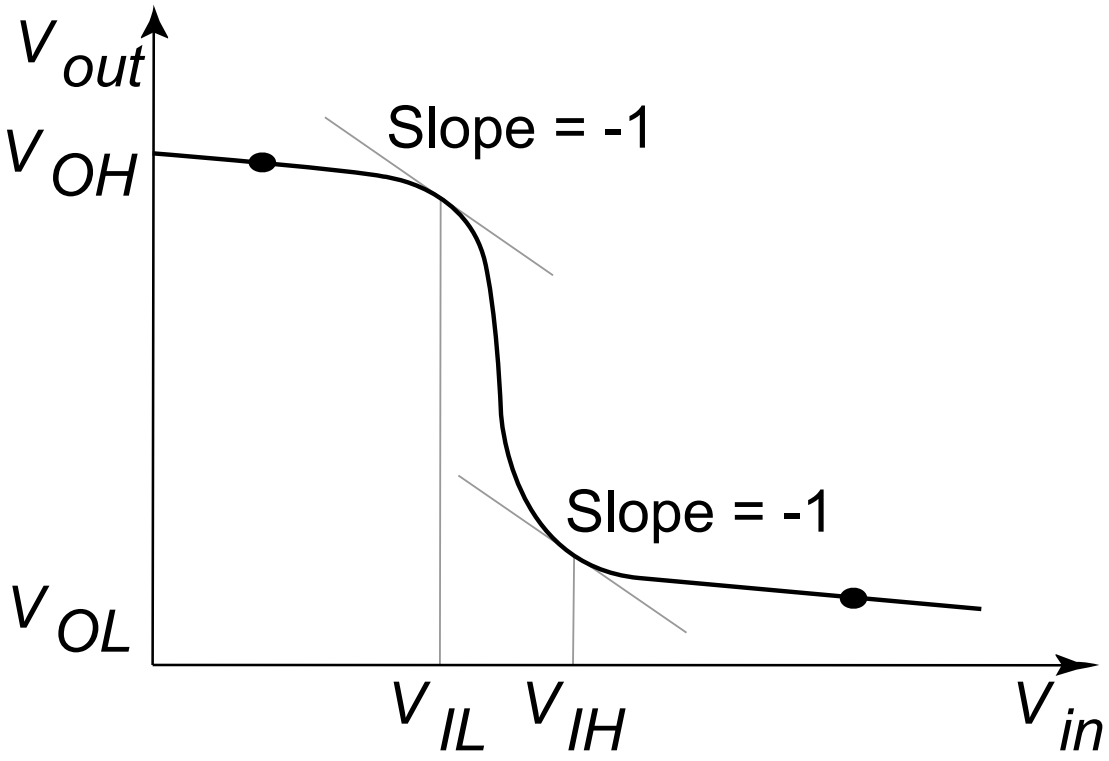
Switching Threshold

VOH = *f*(VOL)
VOL = *f*(VOH)
VM = *f*(VM)

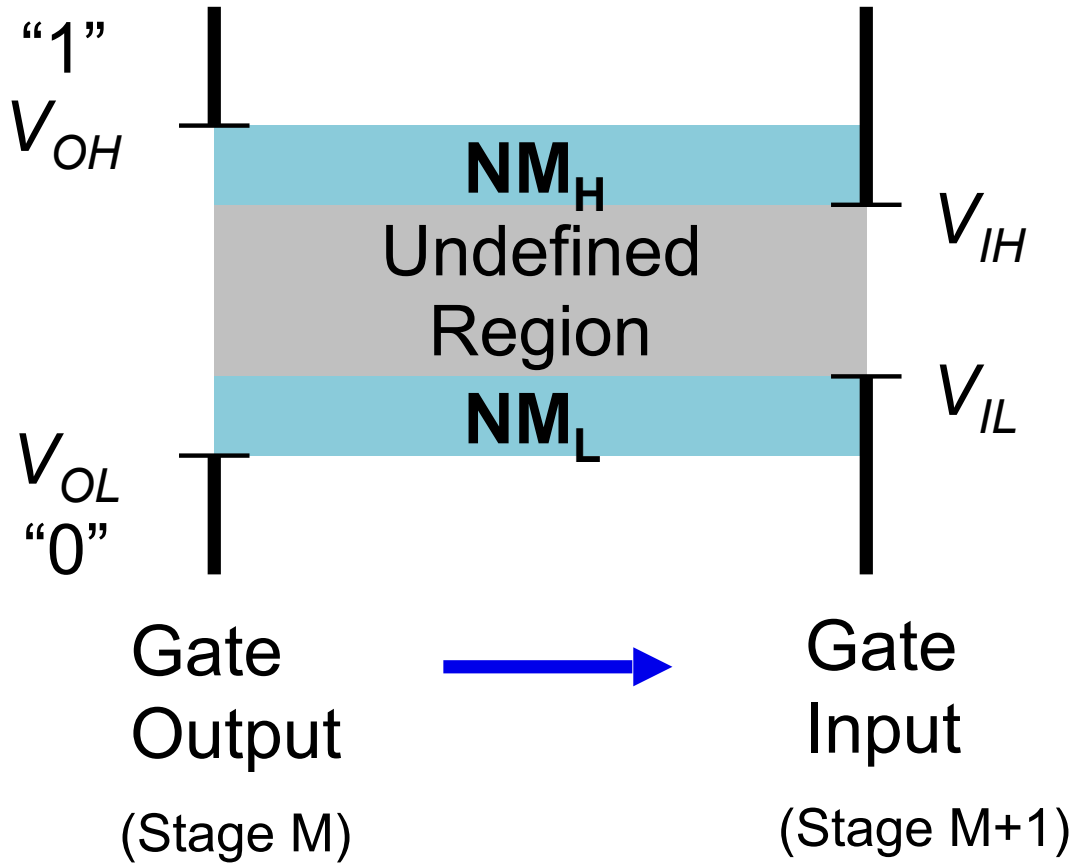
Nominal Voltage Levels

Voltage Mapping





Definition of Noise Margins



Noise margin high:
$NM_H = V_{OH} - V_{IH}$

Noise margin low:
$NM_L = V_{IL} - V_{OL}$