Lecture 13:
Digital Logic Cont..

Announcements

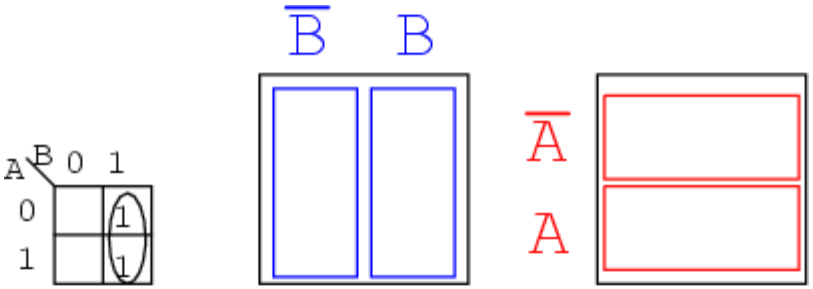
- Project 3 due on Monday
  - Due August 11th 11:55pm
  - Due day before Final, so plan accordingly
  - No Extension
- Project 2 grades will be out tonight
- Final Next Wednesday
  - 3 hour timed exam
  - Open for 24-hours
  - Few Sample Questions w/o answers by this weekend
- Lectures until final:
  - Digital Logic (Chapter 4 in book)
  - Today: Adders/ Sequential Circuits
- Rest of lecture time / recitation today:
  - Questions on todays material and programming Assignment 3

Another Optimization Approach: Karnaugh Maps

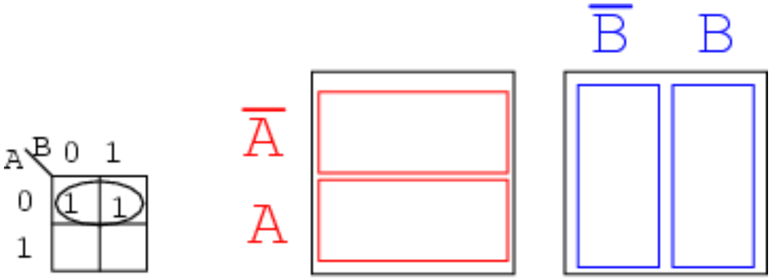
- K-maps are a graphical technique to view minterms and how they relate.
- Extremely useful to simplify boolean functions.
- The "map" is a diagram made up of squares, with each square representing a single minterm.
- Minterms resulting in a "1" are marked as "1", all others are marked "0"

Finding Commonality

- Its usefulness come from the fact that adjacent squares correspond to minterms that differ in only ONE variable.
- Combining 2^n squares eliminate n variables.

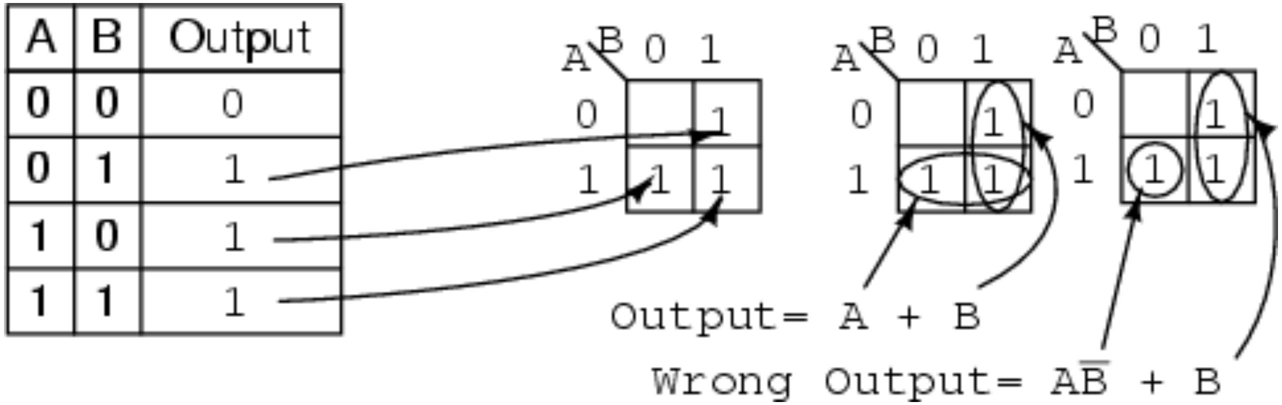


Out = B



Out = $\overline{A}$

Finding the “best” solution



A	B	Output
0	0	0
0	1	1
1	0	1
1	1	1

A\B	0	1
0		1
1	1	1

A\B	0	1
0		1
1	1	1

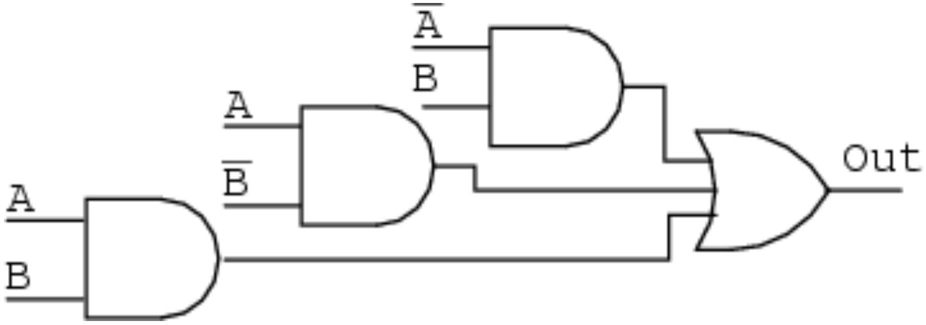
A\B	0	1
0		1
1	1	1

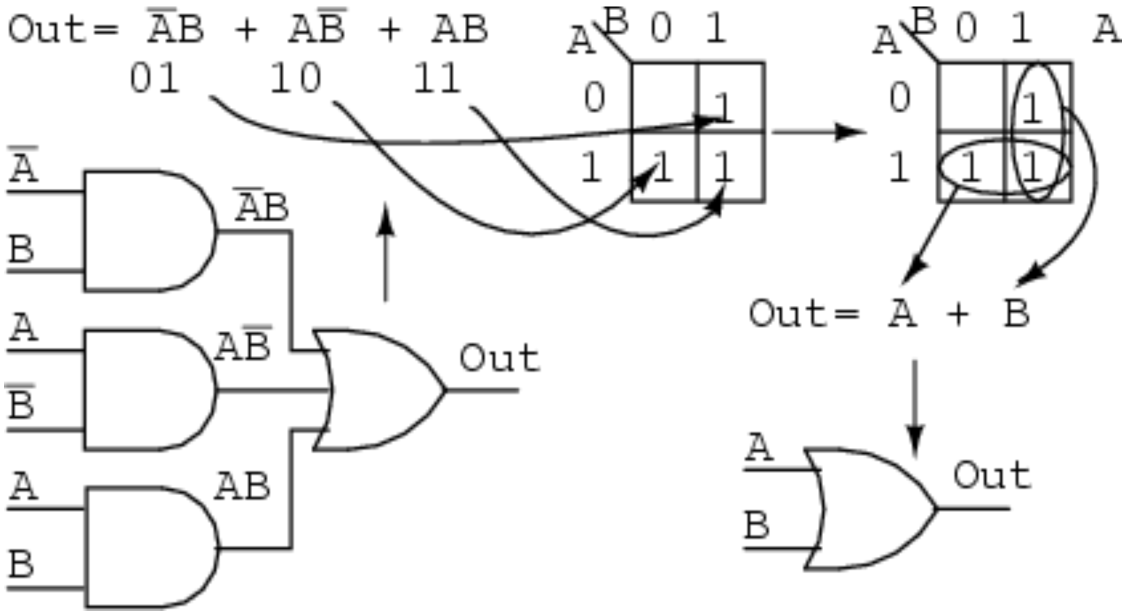
Output= A + B

Wrong Output= $A\overline{B}$ + B

- Grouping become simplified products.
- Both are “correct”. “A+B” is preferred.

Simplify Example





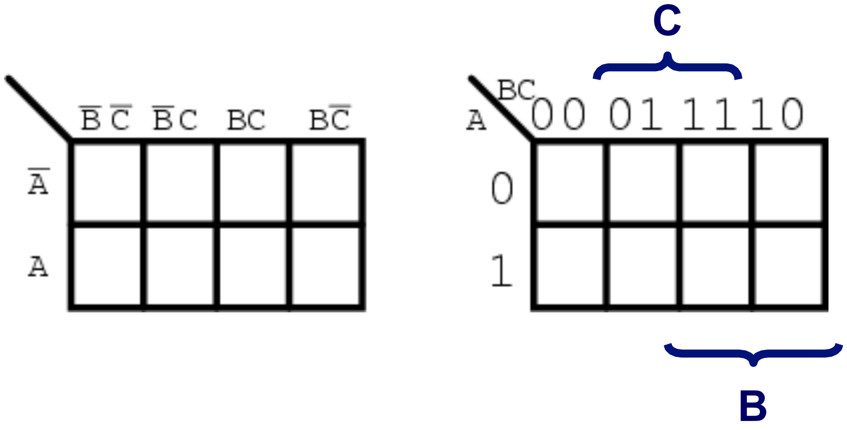
Out= $\bar{A}B + A\bar{B} + AB$

01 10 11

A \ B	0	1
0		1
1	1	1

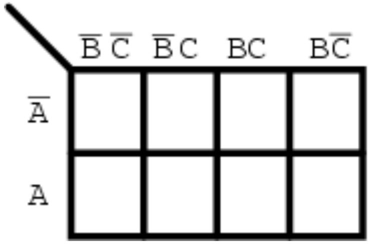
Out= A + B

3 Variable K-Maps

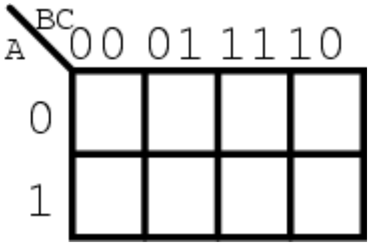


- Note in higher maps, several variables occupy a given axis
- The sequence of 1s and 0s follow a Gray Code Sequence.
- Grey code is a number system where two successive values differ only by 1-bit

3 Variable K-Maps (Example 4)

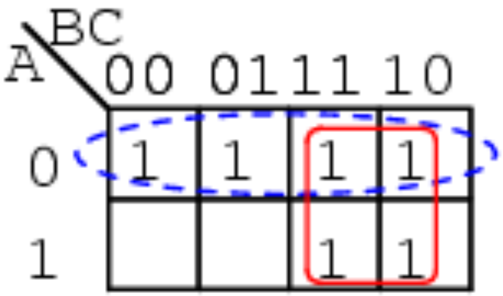


	$\bar{B}\bar{C}$	$\bar{B}C$	BC	$B\bar{C}$
$\bar{A}$				
A				



A \ BC	00	01	11	10
0				
1				

$$Out = \bar{A}\bar{B}\bar{C} + \bar{A}\bar{B}C + \bar{A}BC + \bar{A}B\bar{C} + ABC + AB\bar{C}$$

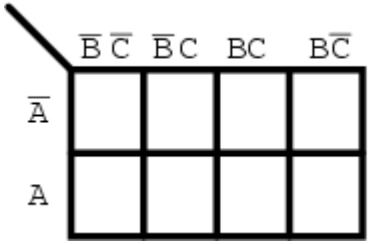


A \ BC	00	01	11	10
0	1	1	1	1
1			1	1

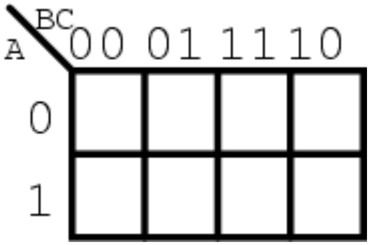
$$Out = \bar{A} + B$$

- Its usefulness come from the fact that adjacent squares correspond to minterms that differ in only ONE variable.
- **Combining 2^n squares eliminate n variables.**

3 Variable K-Maps (Example 5)

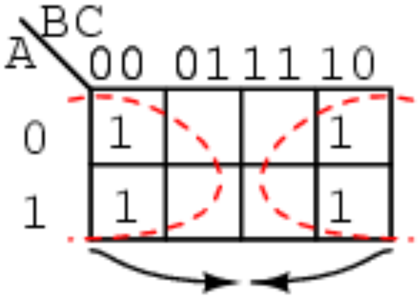


	$\overline{B}\,\overline{C}$	$\overline{B}C$	BC	$B\overline{C}$
$\overline{A}$				
A				

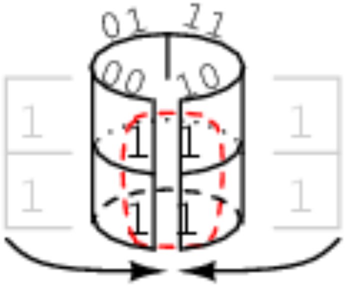


A \ BC	00	01	11	10
0				
1				

$$Out = \overline{A}\,\overline{B}\,\overline{C} + A\overline{B}\,\overline{C} + \overline{A}B\overline{C} + AB\overline{C}$$

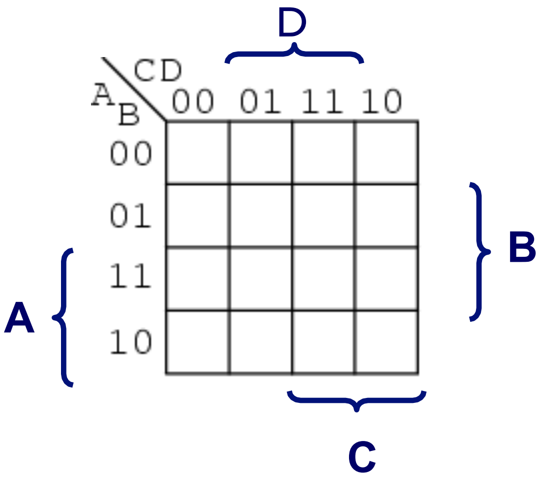


A \ BC	00	01	11	10
0	1			1
1	1			1

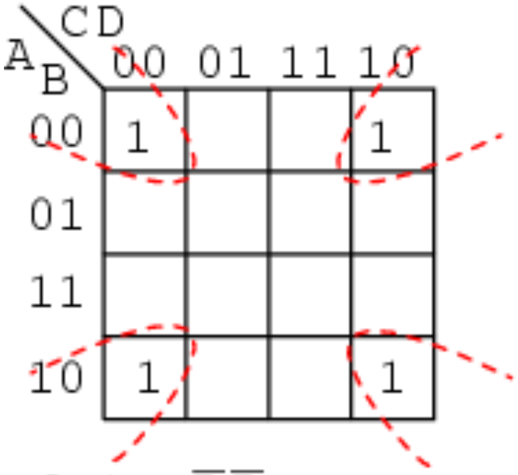


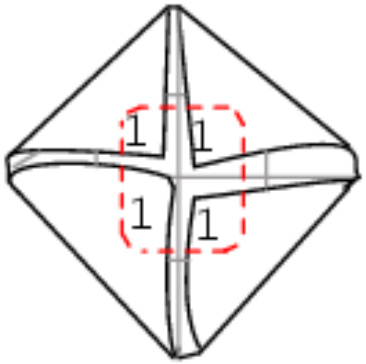
$$Out = \overline{C}$$

Up… up… and let’s keep going



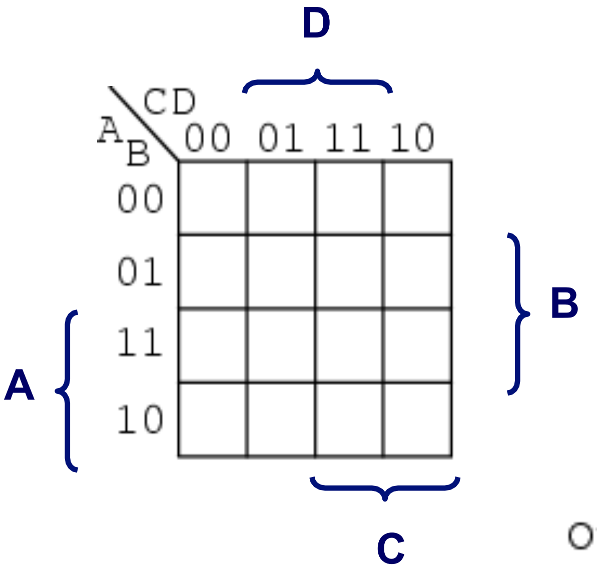
$$Out = \bar{A}\bar{B}\bar{C}\bar{D} + \bar{A}\bar{B}C\bar{D} + A\bar{B}\bar{C}\bar{D} + A\bar{B}C\bar{D}$$



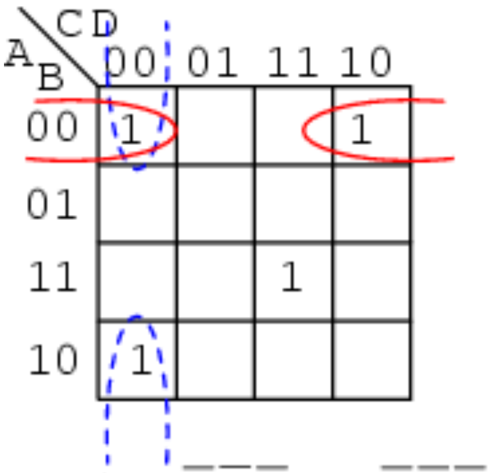


$Out = \bar{B}\bar{D}$

Few more examples



$$Out = \bar{A}\bar{B}\bar{C}\bar{D} + \bar{A}\bar{B}C\bar{D} + A\bar{B}\bar{C}\bar{D} + ABCD$$



AB \ CD	00	01	11	10
00	1			1
01				
11			1	
10	1			

$$Out = \bar{B}\bar{C}\bar{D} + \bar{A}\bar{B}\bar{D} + ABCD$$

Don’t Care Conditions

- Let F = $AB + \overline{A}\,\overline{B}$
- Suppose we know that a disallowed input combo is A=1, B=0
- Can we replace F with a simpler function G whose output matches for all inputs we do care about?
- Let H be the function with Don’t-care conditions for obsolete inputs

Inputs will not occur

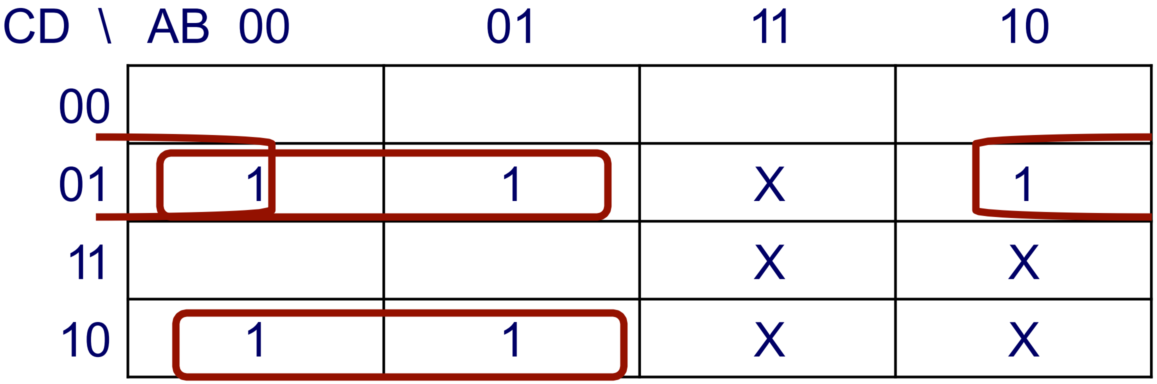
A	B	F	H	G
0	0	1	1	1
0	1	0	0	0
1	0	0	X	1
1	1	1	1	1

$$G = AB + \overline{B}$$

- Both F & G are appropriate functions for H
- G can substitute for F for valid input combinations

Don’t Cares can Greatly Simplify Circuits

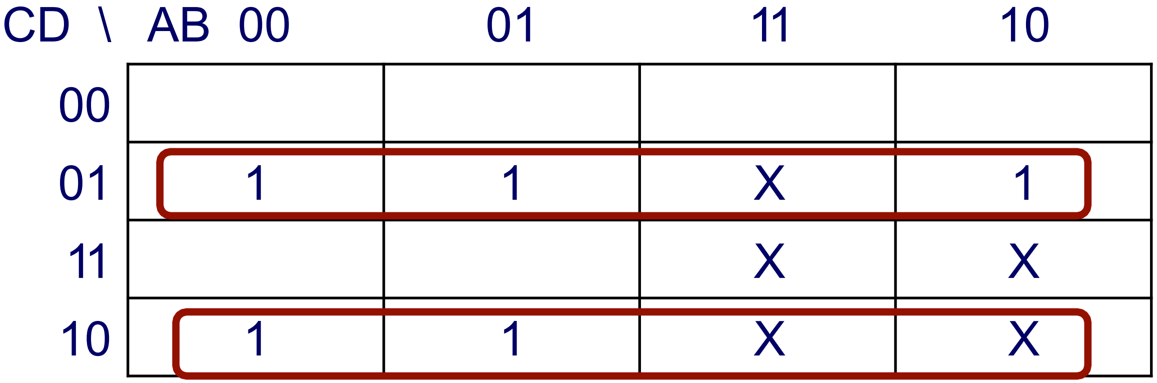
X =0



CD \ AB	00	01	11	10
00				
01	1	1	X	1
11			X	X
10	1	1	X	X

$f = \bar{A}\bar{C}D + \bar{B}\bar{C}D + \bar{A}C\bar{D}$

X=1

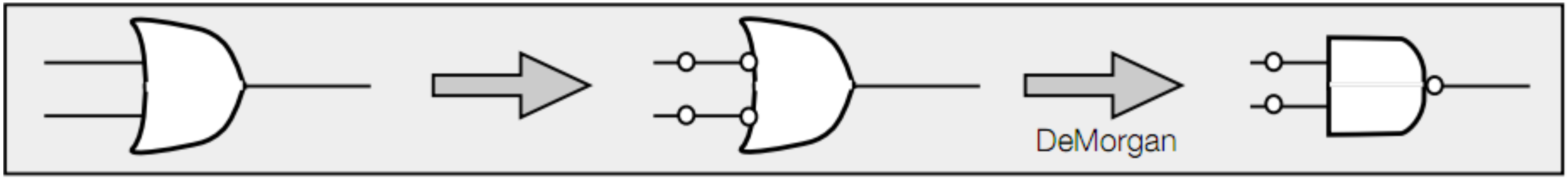


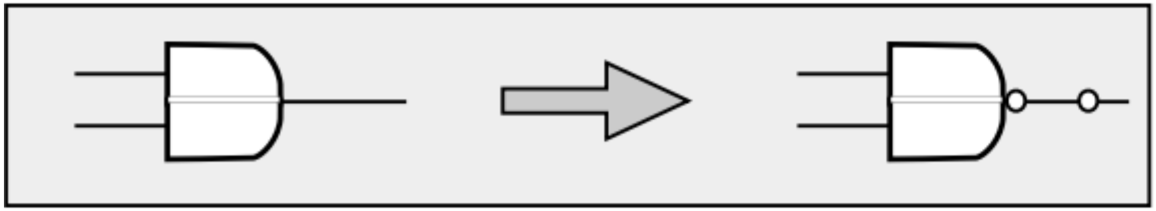
CD \ AB	00	01	11	10
00				
01	1	1	X	1
11			X	X
10	1	1	X	X

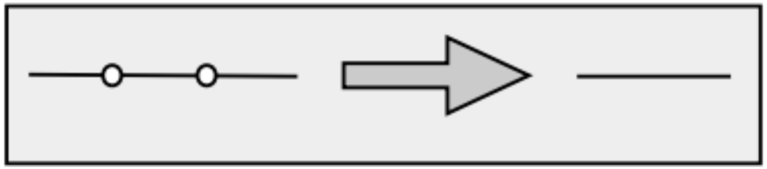
$f = \bar{C}D + C\bar{D}$

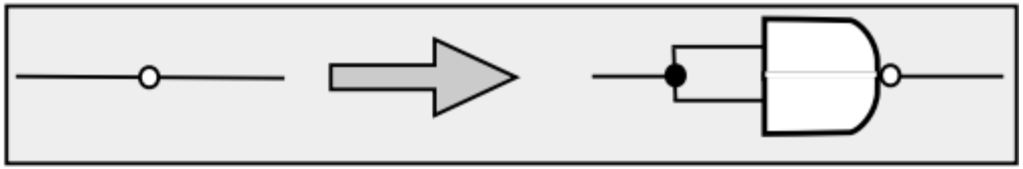
Converting Circuits to all-NAND (NOR)

- Introduce NOT gates
  - Go from left to right
  - Introduce NOT gates in pairs (does not alter logic function)
  - Isolated NOT gates can be implemented as NAND (NOR)

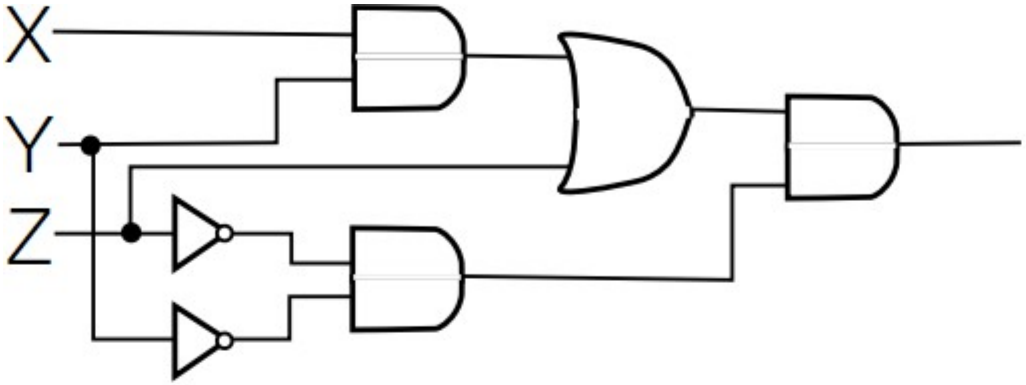


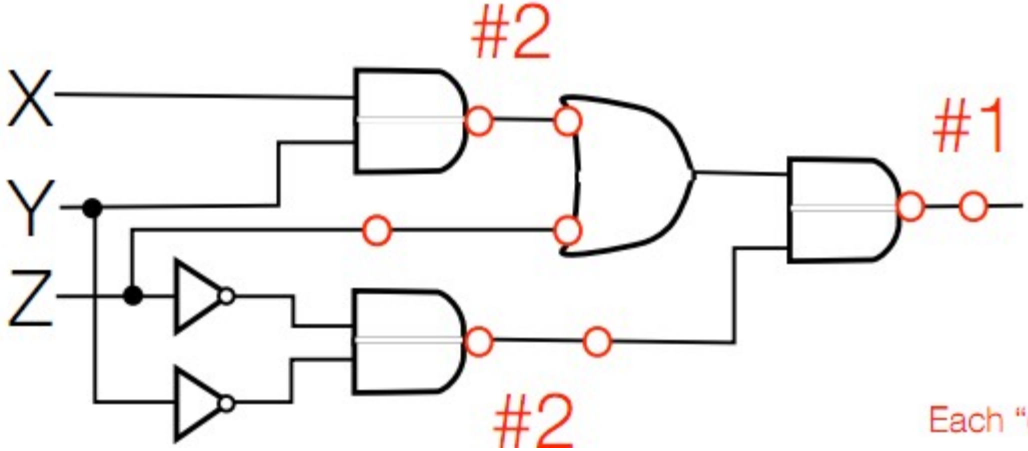






Example





Each "o" by itself represents a NOT gate

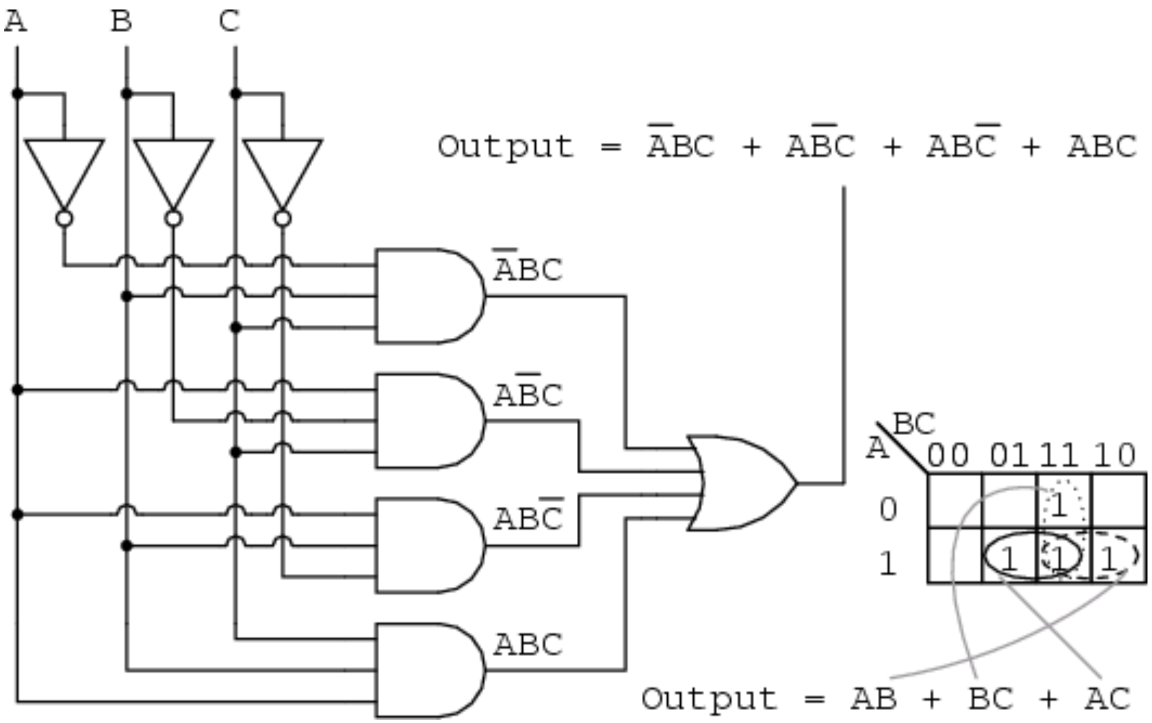
Put it all together…..

sensor inputs

A	B	C	Output	
0	0	0	0	
0	0	1	0	
0	1	0	0	
0	1	1	1	$\overline{A}BC = 1$
1	0	0	0	
1	0	1	1	$A\overline{B}C = 1$
1	1	0	1	$AB\overline{C} = 1$
1	1	1	1	$ABC = 1$

$\text{Output} = \overline{A}BC + A\overline{B}C + AB\overline{C} + ABC$

1. Express the truth table function in SOP (this example) or POS
2. Simplify the function using K-maps (this example) or boolean algebra

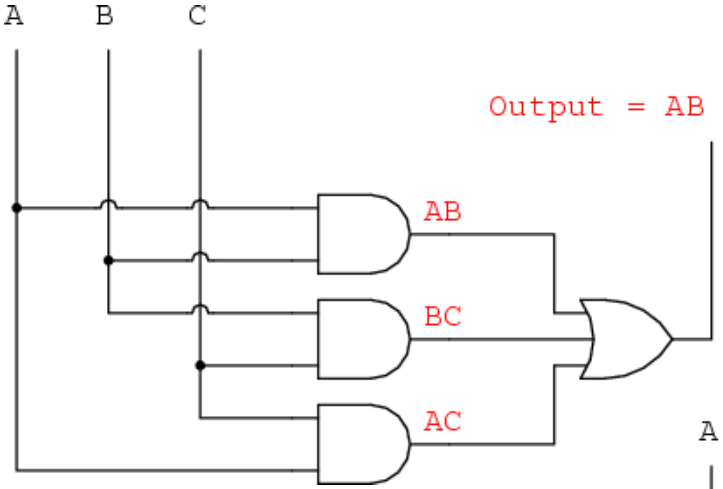


$\text{Output} = \overline{A}BC + A\overline{B}C + AB\overline{C} + ABC$

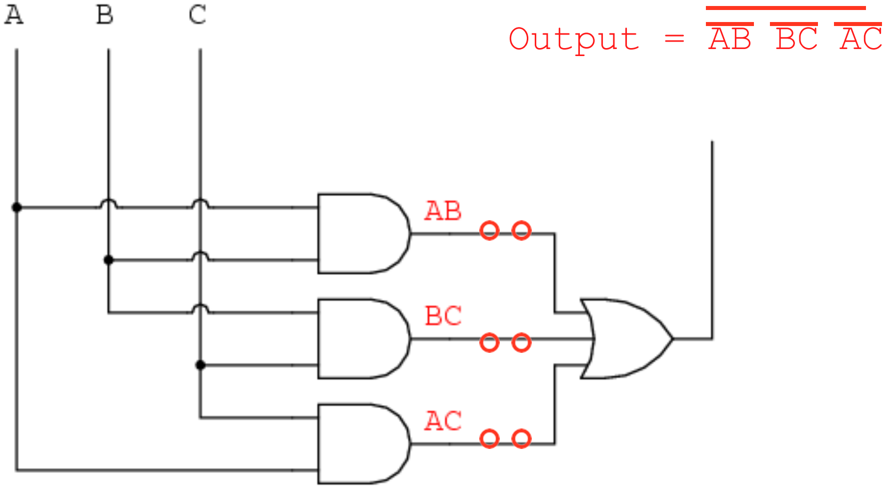
A \ BC	00	01	11	10
0			1	
1		1	1	1

$\text{Output} = AB + BC + AC$

Put it all together…..



3. Convert function to use NAND or NOR gates


A
B
C
Output = $\overline{\overline{AB}\ \overline{BC}\ \overline{AC}}$
AB
BC
AC

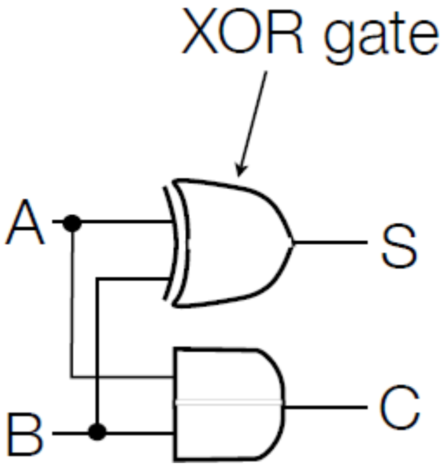
Addition: The Half Adder

Addition of 2 bits: A & B produces summand (S) and carry (C)

A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

$S = A \oplus B$

$C = AB$



But to do addition, we need 3 bits at a time (to account for carries)

$$
\begin{array}{r}
011 \\
+\ 1011 \\
\underline{1001} \\
10101
\end{array}
$$

carry bits

The Full Adder

Takes as input 2 digits (A&B) and previous carry (P)

P	A	B	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

S:

		B		
	0	1	0	1
P	1	0	1	0
			A	

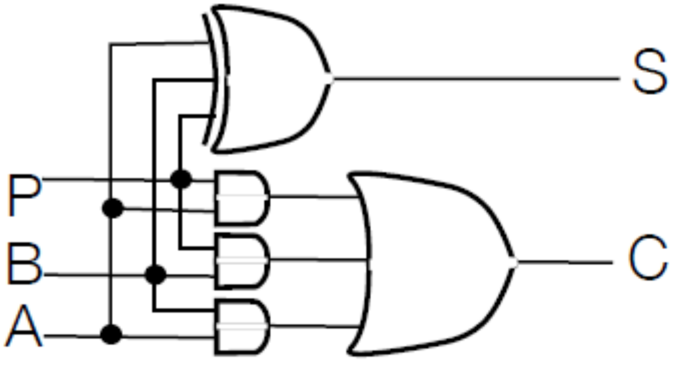
XOR:checker-board pattern

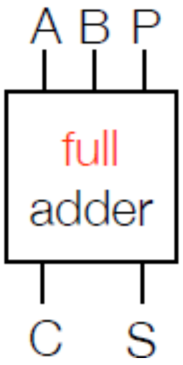
C:

		B		
	0	0	1	0
P	0	1	1	1
			A	

$$S = A \oplus B \oplus P$$

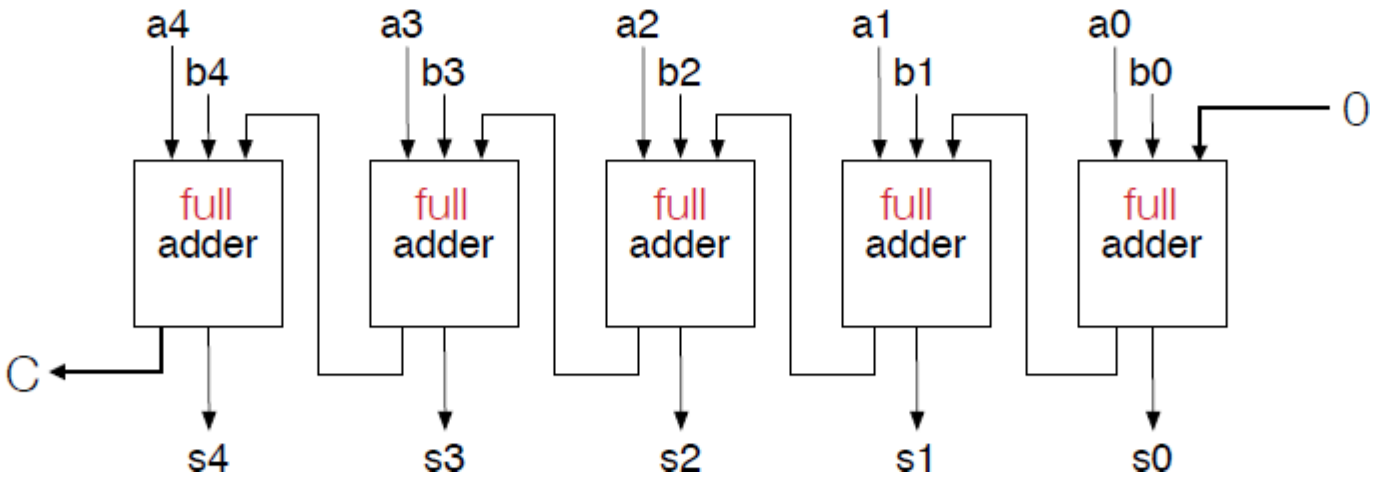
$$C = AB + AP + BP$$





5-bit Ripple Carry Adder

Computes $a_4a_3a_2a_1a_0 + b_4b_3b_2b_1b_0$



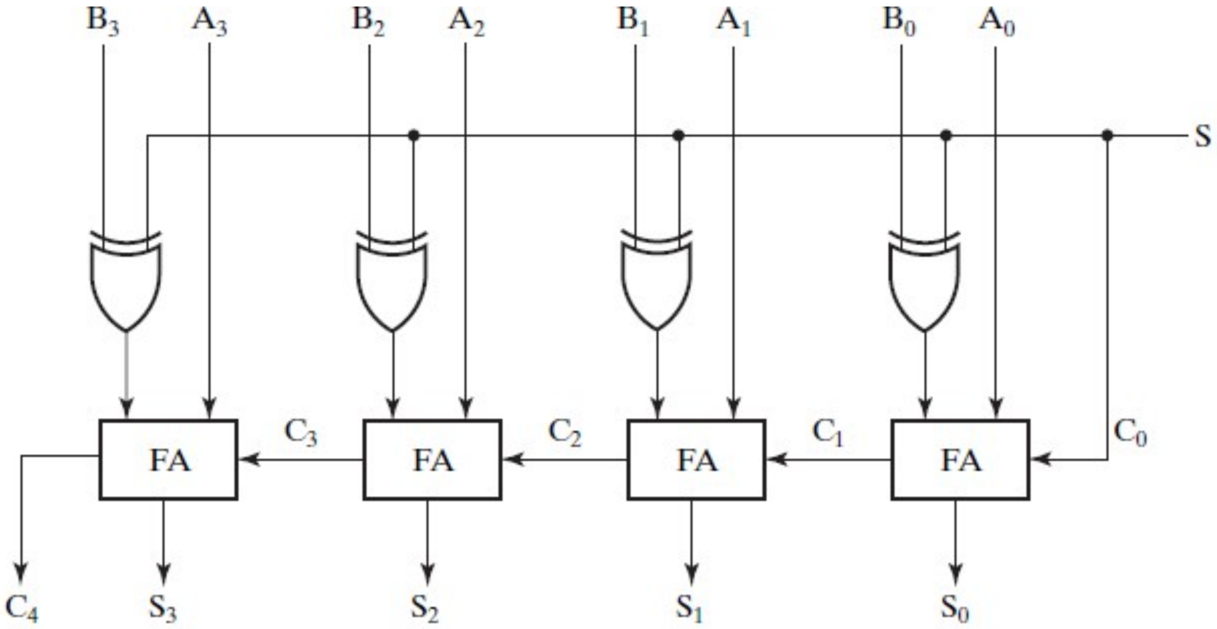
Note how computation ripples from left to right

- Each adder has depth 2 (input passes through 2 gates to reach output)
- Full adder that computes s_i cannot start its computation until previous full adder computes carry
- The longest depth in a k-bit ripple carry adder is 2k

Adder-Subtractor circuit

How to change the adder to add and subtract using same circuit?

- Add extra bit *S* to switch between addition and subtraction
- Add XOR gate at the second input to full-adder



S=0: B unchanged, C_0=0: add

S=1: B complemented, C_0=1 (bits flipped and 1 added): subtract

- Addition: A + B
- Subtraction: A – B = A + (complement of B + 1)

M. Morris Mano & Charles R. Kime
LOGIC AND COMPUTER DESIGN FUNDAMENTALS, 4e

Handling 2's Complement Overflow

Adding 2 positive numbers → negative result

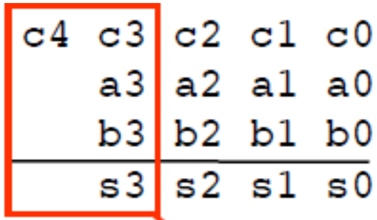
```
           1
   6       0110
 + 5       0101
 -------------
  -5       1011
```

Adding 2 negative numbers → positive result

```
          1 1
   -6      1010
 + -6      1010
 --------------
    4     ~~1~~0100
```

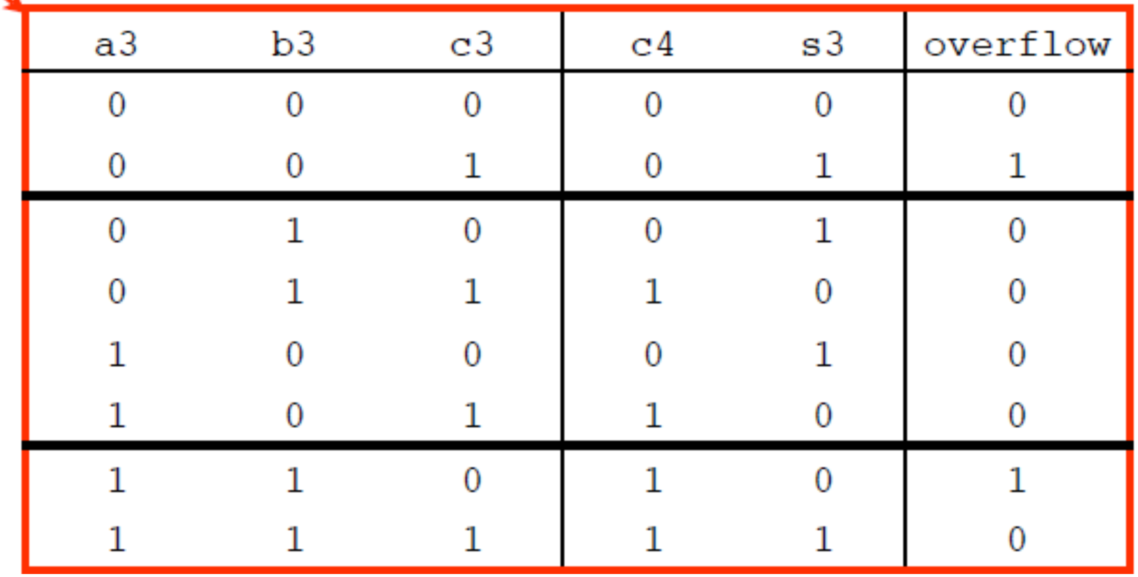
Handling 2's Complement Overflow

- Two cases (look at two most significant carries):
  1. 1 carried in (c3), and 0 carried out (c4)
     - Only if a3 = 0 and b3 = 0
  2. 0 carried in (c3), and 1 carried out (c4)
     - Only if a3 = 1 and b3 = 1



```
c4 c3 c2 c1 c0
   a3 a2 a1 a0
   b3 b2 b1 b0
--------------
   s3 s2 s1 s0
```

n bit two's comp: -2^n <---> 2^n - 1
split into pos and neg ranges and find smallest and largest possible results. show that they're in range for twos comp.



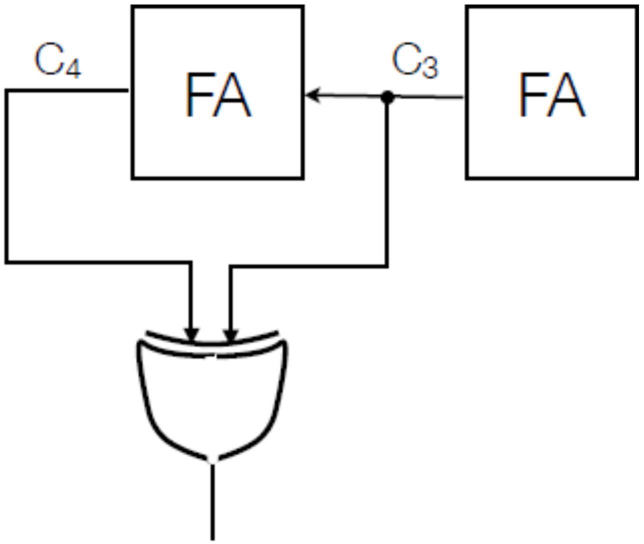
a3	b3	c3	c4	s3	overflow
0	0	0	0	0	0
0	0	1	0	1	1
0	1	0	0	1	0
0	1	1	1	0	0
1	0	0	0	1	0
1	0	1	1	0	0
1	1	0	1	0	1
1	1	1	1	1	0

sum of two pos is pos

sum of two negs is neg

Overflow Computation in Adder/Subtractor

For 2s complement, overflow if 2 most significant carries differ



= 0 then no overflow,

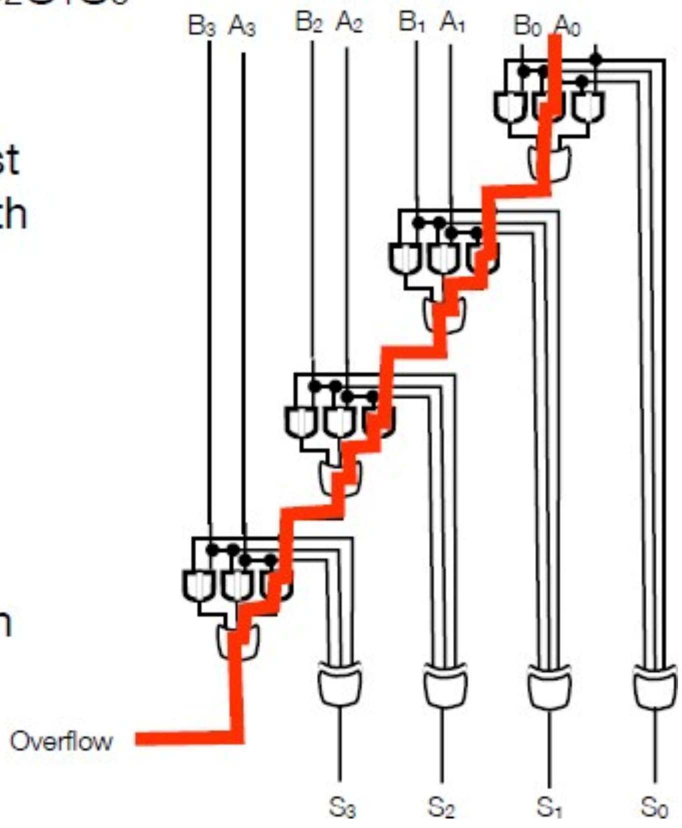
= 1 then overflow

Ripple Carry Adder Circuit Depth

Have to wait for the carry to be computed

$A_3A_2A_1A_0 + B_3B_2B_1B_0 = S_3S_2S_1S_0$

- Depth of a circuit is the longest (most gates to go through) path
- Overflow has depth 8
- S_3 has depth 7
- In general, S_i has depth 2i+1 in Ripple-Carry Adder



Carry Lookahead Adder (CLA)

Goal: produce an adder circuit of shorter depth

Mechanism: rewrite the carry function

$$c_{i+1} = a_i b_i + a_i c_i + b_i c_i$$

$$c_{i+1} = a_i b_i + c_i (a_i + b_i)$$

$$c_{i+1} = g_i + c_i (p_i)$$

Generates carry out (c_{i+1}=1)
if $a_i = b_i = 1$

Propagates carry in ($c_i = 1$)
If either $a_i = 1$ or $b_i = 1$

carry generate

$g_i = a_i b_i$

carry propagate

$p_i = a_i + b_i$

Now, look at g_i and p_i
They both use **only** a_i and b_i, neither depend on the carry. If we write the carry function in terms of g_i and p_i we might avoid waiting for the carry.

Carry Lookahead Adder (CLA)

Recursively define carries in terms of propagate and generate signals

$$
\begin{aligned}
c_1 &= g_0 + c_0p_0 \\
c_2 &= g_1 + c_1p_1 \\
&= g_1 + (g_0 + c_0p_0)p_1 \\
&= g_1 + g_0p_1 + c_0p_0p_1 \\
c_3 &= g_2 + c_2p_2 \\
&= g_2 + (g_1 + g_0p_1 + c_0p_0p_1)p_2 \\
&= g_2 + g_1p_2 + g_0p_1p_2 + c_0p_0p_1p_2
\end{aligned}
$$

We can compute carries in terms of a_i's, b_i's and c0. This bits are available right away, we don't have to wait for them.

i^{th} carry has i+1 product terms, the largest of which has i+1 literals

Carry circuit depth is 3 (SoP form)

If gates take 2 inputs, total circuit depth is $1 + \log_2(k)$ for k-bit addition

Carry Lookahead Adder (CLA)

$c_0 = 0$

$c_1 = g_0 + c_0p_0$

$c_2 = g_1 + g_0p_1 + c_0p_0p_1$

$c_3 = g_2 + g_1p_2 + g_0p_1p_2 + c_0p_0p_1p_2$

$c_4 = g_3 + g_2p_3 + g_1p_2p_3 + g_0p_1p_2p_3 + c_0p_0p_1p_2p_3$

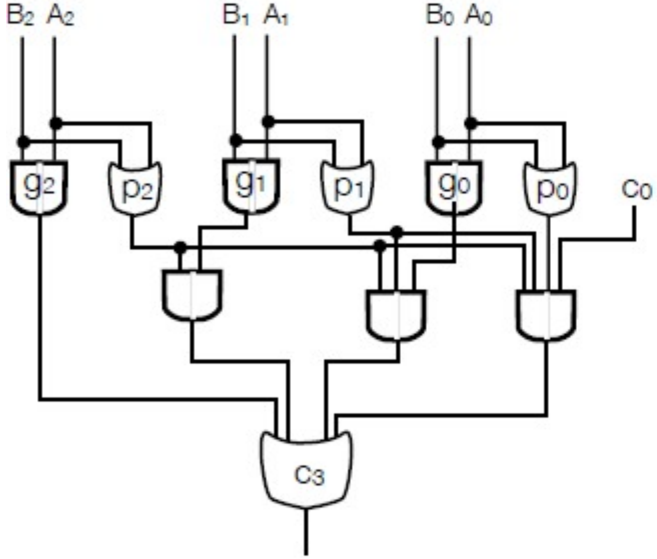
$s_0 = a_0 \oplus b_0 \oplus c_0$

$s_1 = a_1 \oplus b_1 \oplus c_1$

$s_2 = a_2 \oplus b_2 \oplus c_2$

$s_3 = a_3 \oplus b_3 \oplus c_3$

$s_4 = a_4 \oplus b_4 \oplus c_4$



Depth of 3 for all c_i

Depth of 4 for all s_i, i>0

Note: gates take only 2 inputs: depth increases by a log2 factor: still much less than linear of ripple-carry adder

Sequential circuits

Now on to Sequential Circuits

We've seen that gates are the building blocks for combinational circuits, now we'll see how latches and flip-flops are the building blocks of sequential circuits.

- Gates: built from transistors
- Latches: built from gates
- Flip-flops: built from latches

Difference between latches and flip-flops:

- Latches do not have clock signals

Latches

- Set/Reset
- D

Flip-flops

- D
- T
- Enabled
- Resettable

Registers

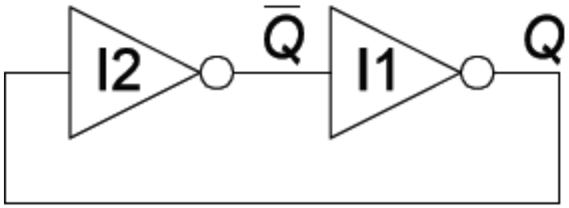
Counters

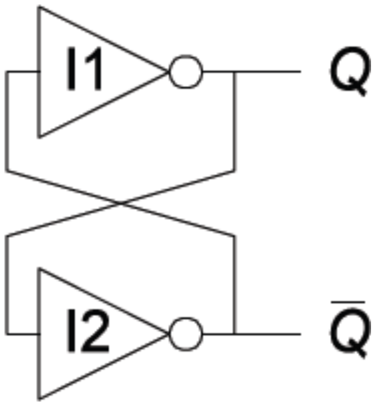
How are Sequential Circuits different from Combinational Circuits?

- How to make a circuit out of gates that is not combinational?
  - Feed-back: create loops in the diagram
  - Outputs of sequential logic depend on both current and prior values – it has memory
- Definitions:
  - State: all the information about a circuit to explain its future behavior
  - Latches and flip-flops: state elements that store one bit of state
  - Synchronous sequential elements: combinational logic followed by a bank of flip-flops

Bistable Circuits (Static latch)

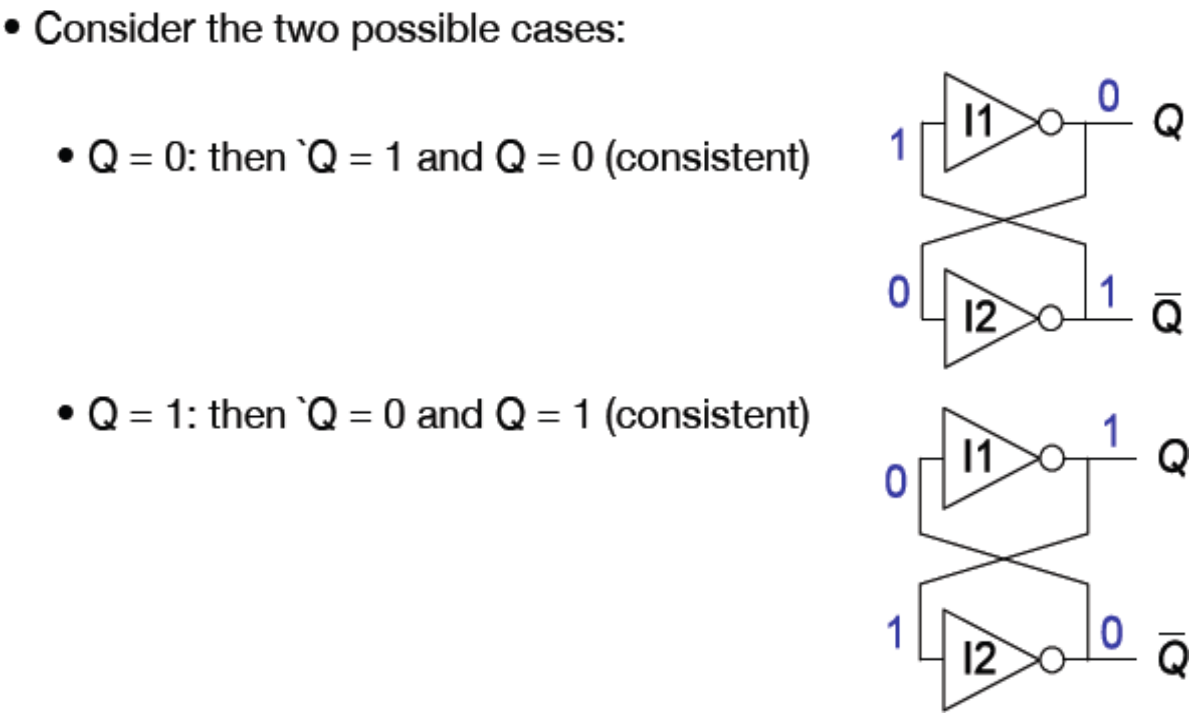
- Fundamental building blocks of other elements
  - it can remember the state of the circuit indefinitely
- No inputs
- Two outputs (Q and Q')





Bistable Circuit Analysis

Consider all the cases

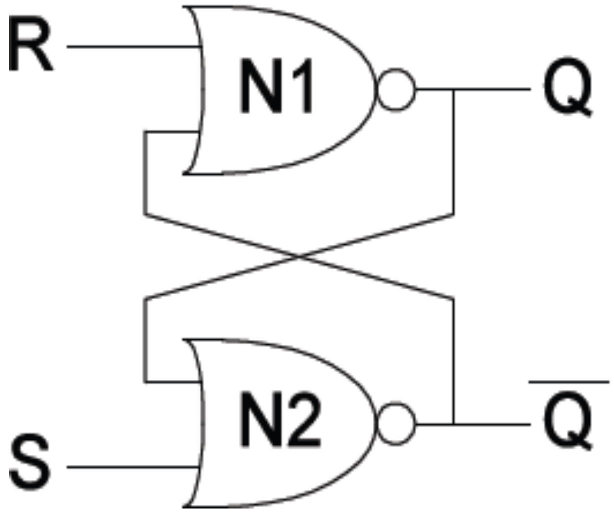


- Consider the two possible cases:
  - Q = 0: then `Q = 1 and Q = 0 (consistent)
  - Q = 1: then `Q = 0 and Q = 1 (consistent)

- Bistable circuit stores 1 bit of state (Q, or Q')
- But there are no inputs to control state

Set/Reset Latch

A latch with NOR gates

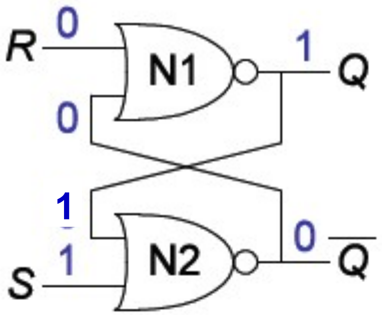


- S = 1, R = 0
- S = 0, R = 1
- S = 0, R = 0
- S = 1, R = 1

S/R Latch Analysis

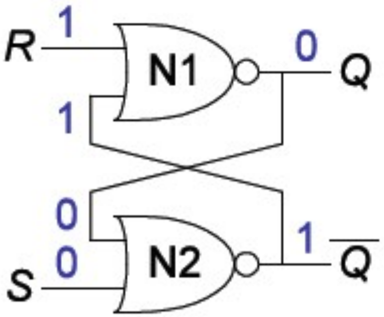
A	B	$\overline{A+B}$
0	0	1
0	1	0
1	0	0
1	1	0

- S = 1, R = 0: then Q = 1



set

- S = 0, R = 1: then Q = 0



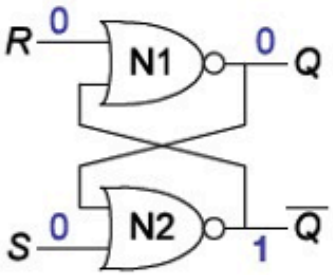
reset

S/R Latch Analysis

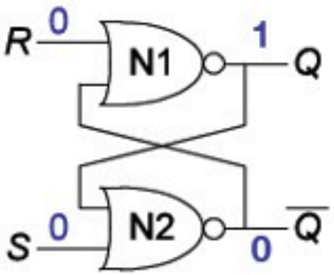
A	B	$\overline{A+B}$
0	0	1
0	1	0
1	0	0
1	1	0

- S = 0, R = 0: then Q = Q_{prev}

$Q_{prev} = 0$

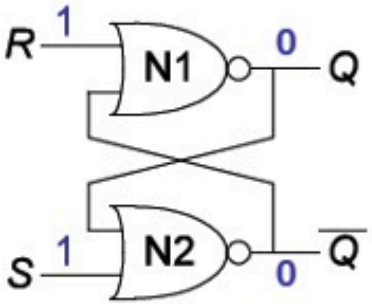


$Q_{prev} = 1$



memory!

- S = 1, R = 1: then Q = 0 and `Q = 0



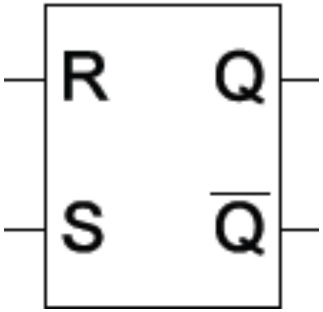
Q=`Q
Invalid state

S/R Latch Symbol

- Set operation
  - makes output 1 (S = 1, R = 0, Q = 1)
- Reset operation
  - makes output 0 (S = 0, R = 1, Q = 0)
- What about invalid state? (S = 1, R = 1)

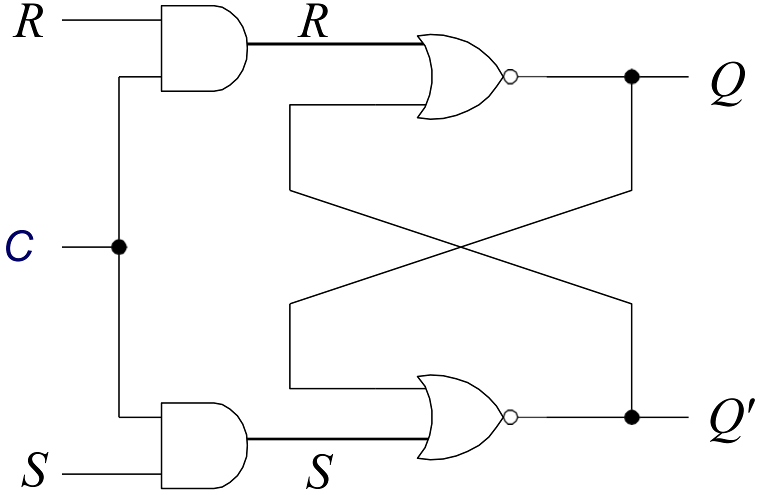
  makes Q = Q = 0

SR Latch Symbol



S/R Latch with Enable

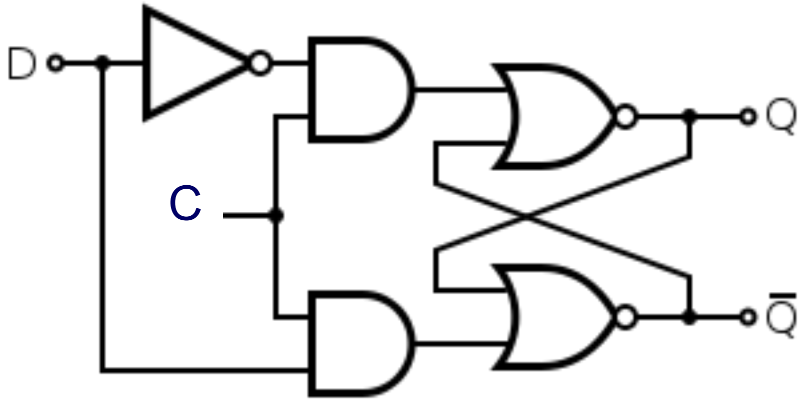
- The SR latch is sensitive to its inputs all the time. It is sometimes useful to be able to disable the inputs.
- The *SR latch with enable* accomplishes this by adding an enable input, *C*, to the original implementation of the latch that allows the latch to be enabled or disabled.



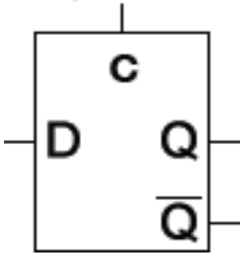
- The SR latch with enable is still sensitive to S = R = 1

D Latch

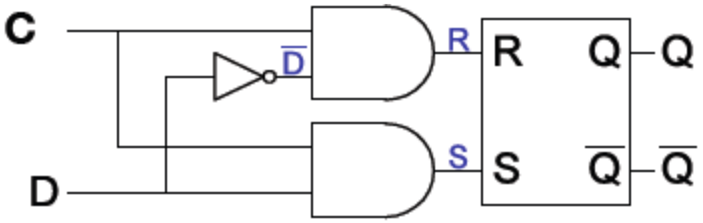
- Prevents S = R = 1 by adding a inverter
- The D latch has two inputs (C and D)
  - C (control input): controls when the output changes
  - D (data input): controls what the output changes to
- When C = 1, D passes through to Q (transparent latch)
- When C = 0, Q holds previous value (opaque latch)



D Latch Symbol



D Latch Internal Circuit



C	D	`D	S	R	Q	`Q	
0	X	X	0	0	Prev	Prev	**No change**
1	0	1	0	1	0	1	**Reset**
1	1	0	1	0	1	0	**Set**

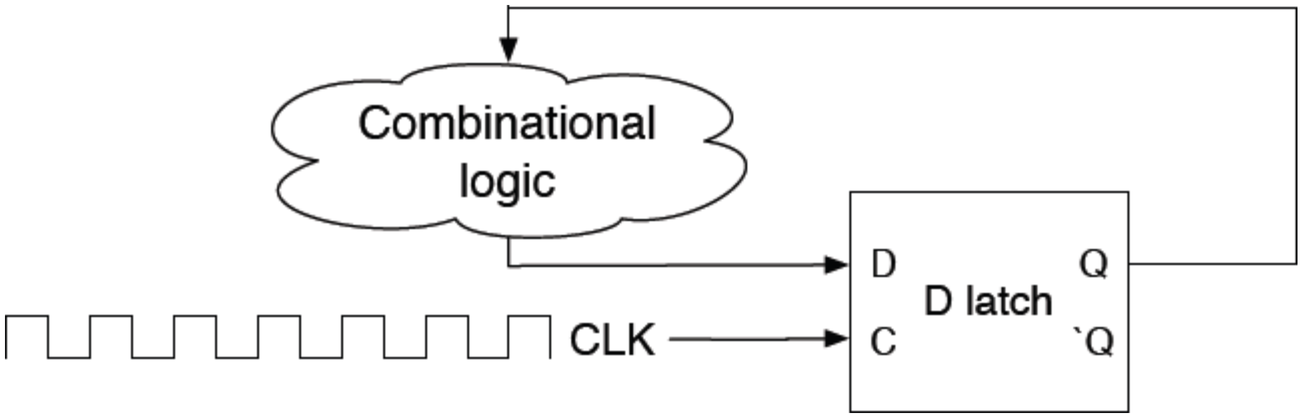
- Level-sensitive: output follows the input as long as it is enabled (C=1)
- If D changes while C=1 the output will reflect D (transparent)

How to Coordinate Multiple Components of a Circuit?

- To do computations, we need more than just combinational circuits We need to use past circuit state to influence future output (latches)
- But how do we coordinate computations and the changing of state values across lots of different parts of a circuit?
  - How to synchronize latches to change values at the same time?
- We use CLOCKING (eg. 1GHz clock on Intel processors)
- **On each clock pulse, combinational computations are performed, and results stored in latches**
- How to introduce clocks into latches?

Flip-flops: Latches on a Clock

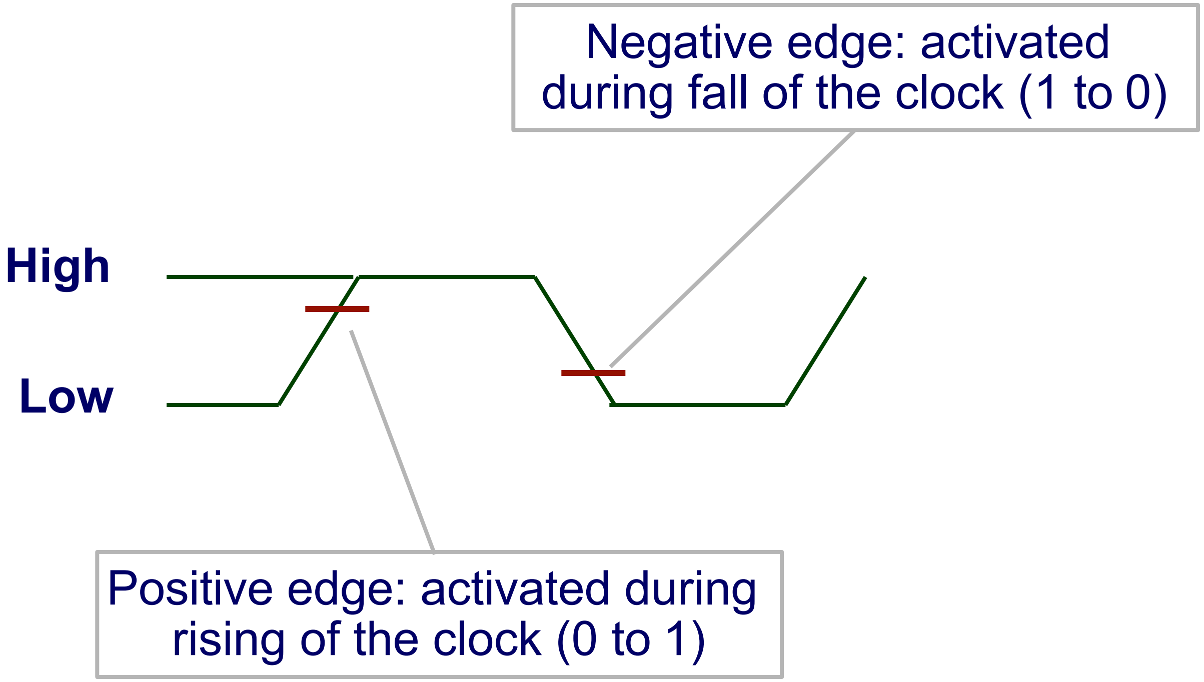
- A straightforward latch is not safely synchronous (or predictably synchronous)
  - Flip-flops can have its state changed only at a single, known instant of time.



- Flip-flops designed so that outputs will NOT change within a single clock pulse

Flip-Flop

- Sensitive to the edge (transition) of the clock
  - rising or falling of the clock



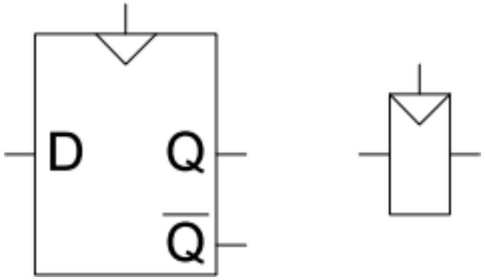
Negative edge: activated during fall of the clock (1 to 0)

Positive edge: activated during rising of the clock (0 to 1)

Positive Edge-Triggered D Flip-Flop

- Two inputs:
  - Clk, D
- Function
  - The flip-flop samples D on rising edge of the Clk
    - When Clk goes from 0 to 1, D passes through Q
  - Otherwise, Q holds its value
  - Q only changes on rising edge of the Clk
- A flip-flop is called "edge-triggered" because it is activated only on the clock edge

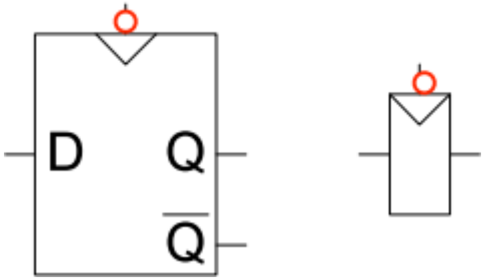
D Flip-Flop Symbols



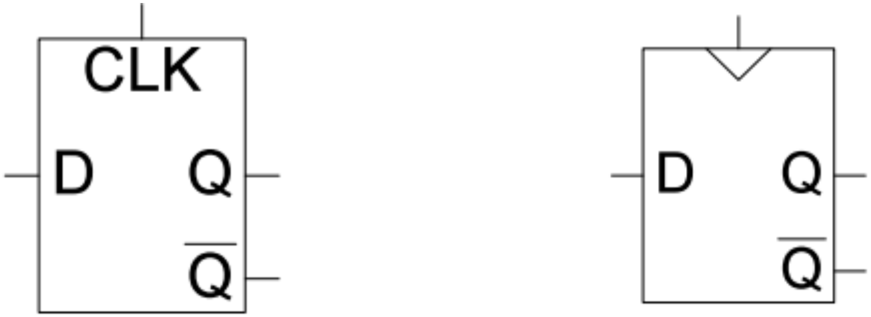
Negative Edge-Triggered D Flip-Flop

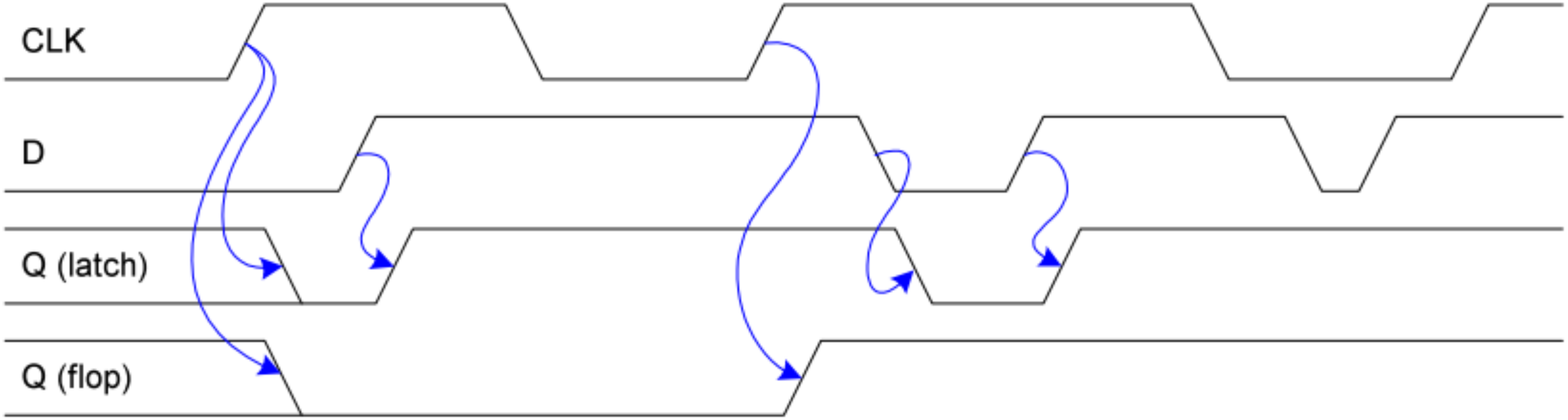
- Two inputs:
  - Clk, D
- Function
  - The flip-flop samples D on the falling edge of Clk
    - When Clk falls from 1 to 0, D passes through to Q
  - Otherwise, Q holds its previous values
  - Q changes only on the falling edge of Clk
- A flip-flop is called an edge-triggered device because it is activated on a clock cycle

D Flip-Flop
Symbols



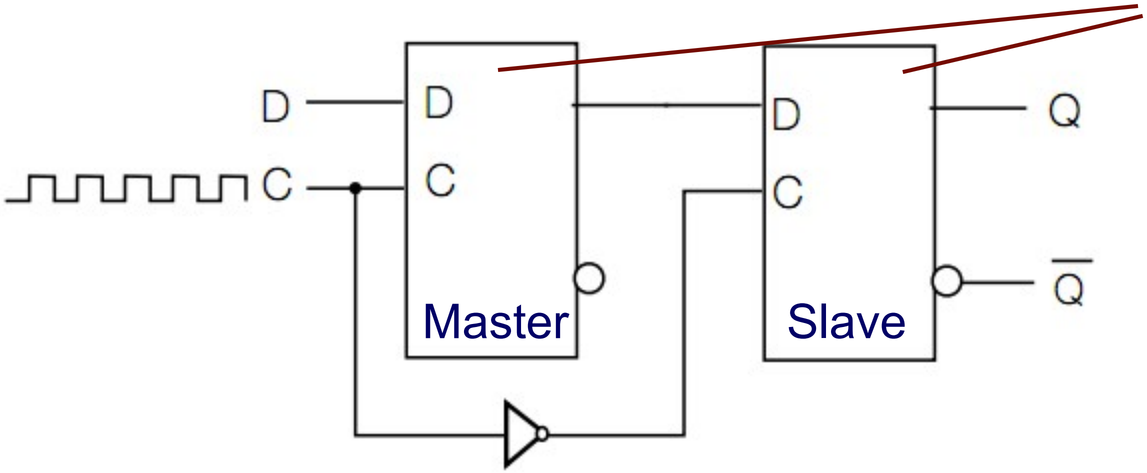
Flip-Flop versus Latch





Latch outputs change at any time, flip-flops only during clock transitions

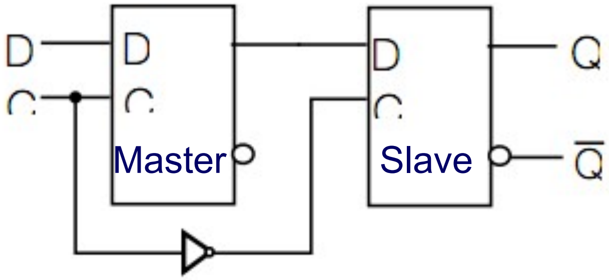
Master Slave D Flip-Flop

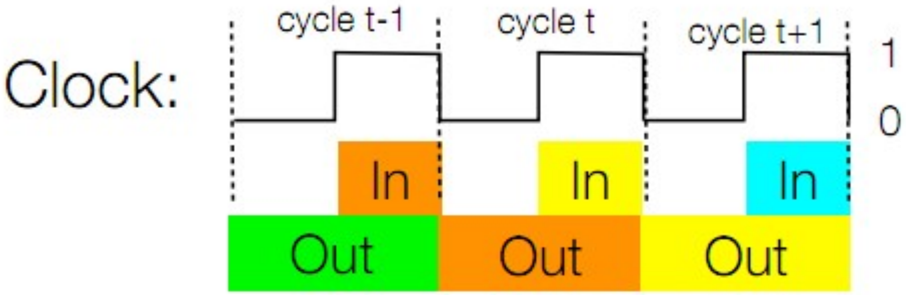


D Latch
C=1, transparent latch
C=0, opaque latch

- C (Control) is fed a clock pulse (alternates between 0 and 1 with fixed period)
  - C=1: Master latch "on", Slave latch "off"
    - New D input read into master
    - Previous Q values still emitted (not affected by new D inputs)
  - C=0: Master latch "off", Slave latch "on"
    - Changing D inputs has no effect on Master (or Slave) latch
    - D inputs from last time C=1 stored safely in Master and transferred into Slave and reflected on output Q

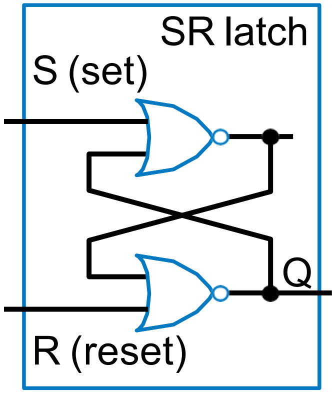
Master Slave D Flip-Flop Activation Time



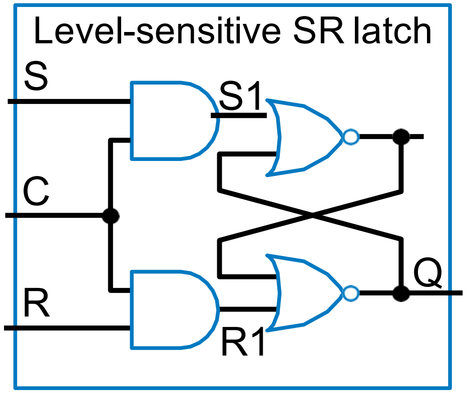


- Q(t): value output by Flip-Flop during the tth clock cycle (clock =0, then 1 during a full cycle)
- Depends on input during end of t-1st cycle

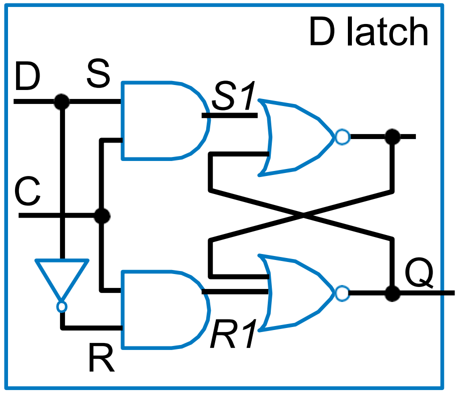
Bit Storage Summary



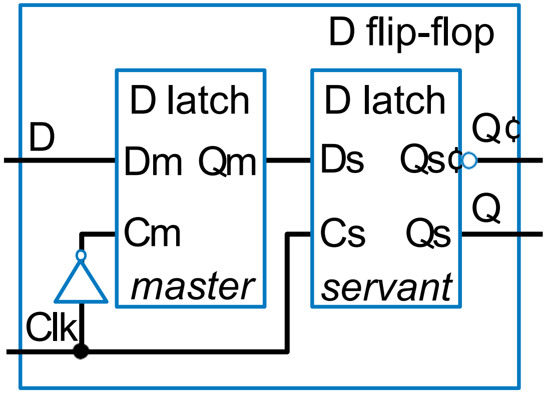
Feature: S=1 sets Q to 1, R=1 resets Q to 0.
Problem: SR=11 yields undefined Q, other glitches may set/reset inadvertently.



Feature: S and R only have effect when C=1. An external circuit can prevent SR=11 when C=1.
Problem: avoiding SR=11 can be a burden.



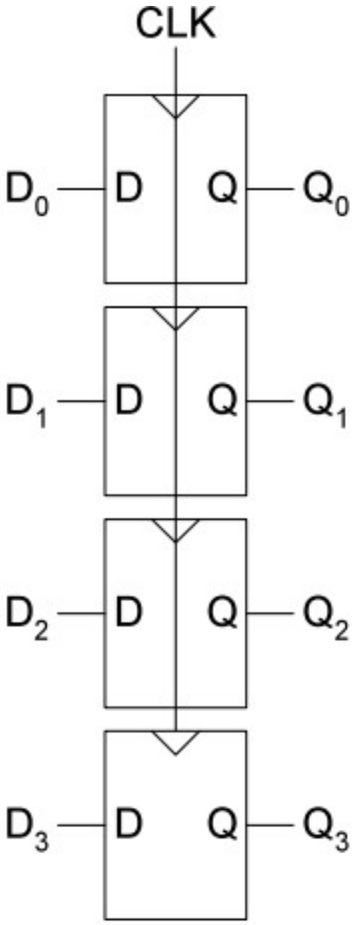
Feature: SR can't be 11.
Problem: C=1 for too long will propagate new values through too many latches; for too short may not result in the bit being stored.

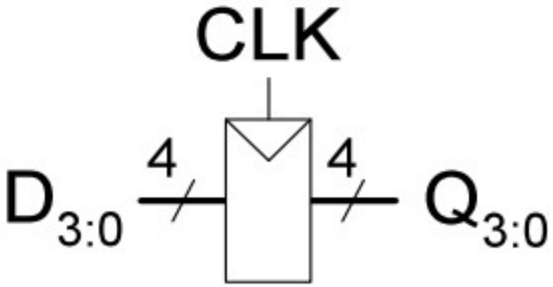


Feature:Only loads D value present at rising clock edge, so values can't propagate to other flip-flops during same clock cycle.
Tradeoff: uses more gates internally, and requires more external gates than SR– but transistors today are more plentiful and cheaper.

We considered increasingly better bit storage until we arrived at the robust D flip-flop bit storage

Registers





Synchronous Sequential Logic Design

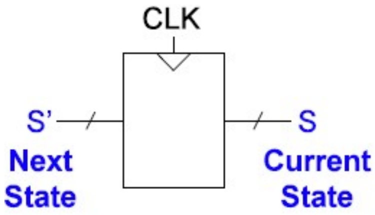
- Structures that can process and store information
  - Registers contain the state of the system
  - Combinational circuits process information
- State changes at the clock edge, so the system is synchronized to the clock
- Rules of synchronous sequential circuit composition:
  - Every circuit is either a register or a combinational circuit
  - At least one circuit element is a register
  - All registers receive the same clock signal
  - Every cyclic path contains one register
- Two common synchronous sequential circuits
  - Finite state machines (FSMs)
  - Pipelines

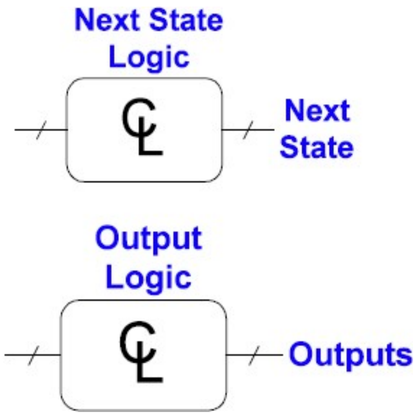
Finite State Machines

- A mathematical model of computation used to design:
  - Computer programs
  - Sequential logic circuits
- FSM = State register + combinational logic

Stores the next state and loads the next state at clock edge

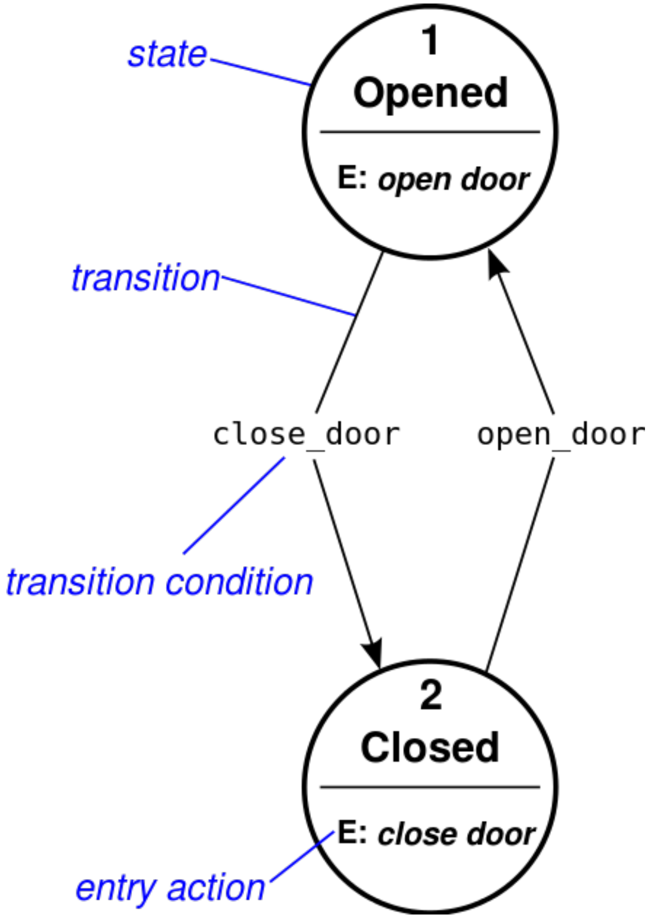
Computes the next state and computes the outputs





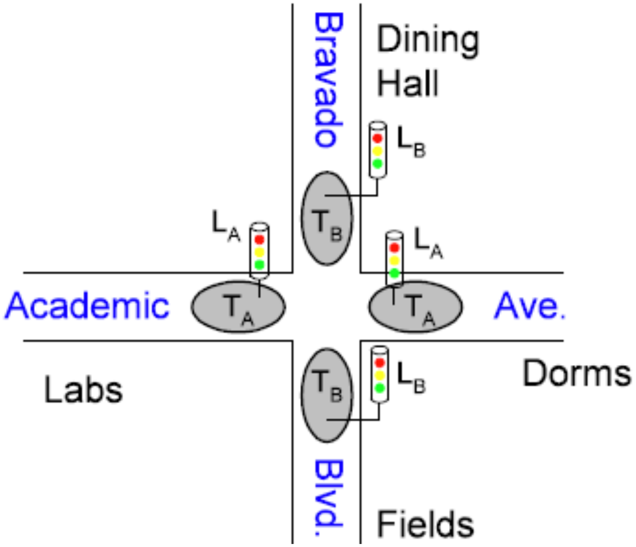
Finite State Machines

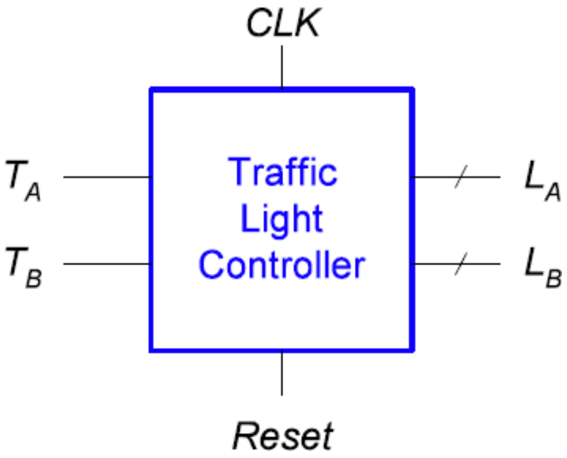
- Can be represented using a state diagram
  - Finite number of states
  - Transitions



FSM: Traffic Light Controller Example

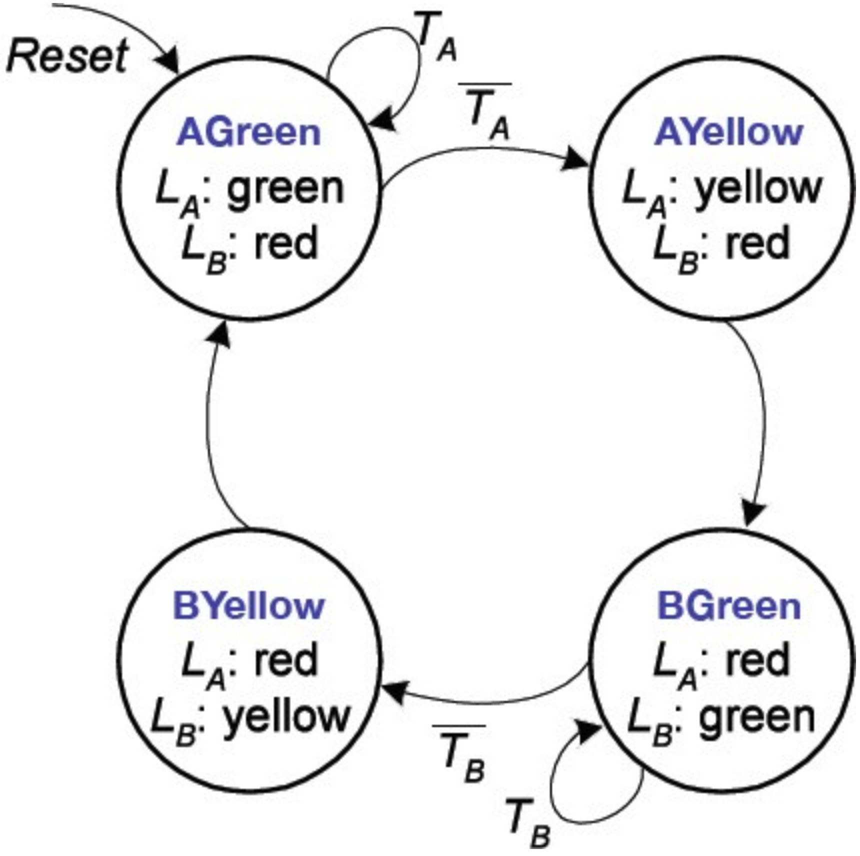
- Traffic sensors: TA, TB (TRUE when there is traffic)
- Lights: LA, LB





FSM State Transition Diagram

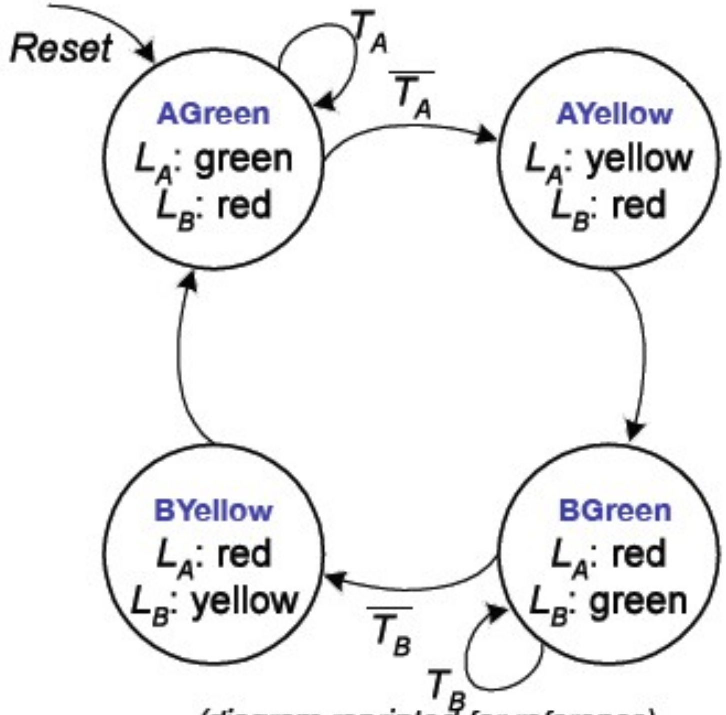
- States: Circles
- Transitions: Arcs



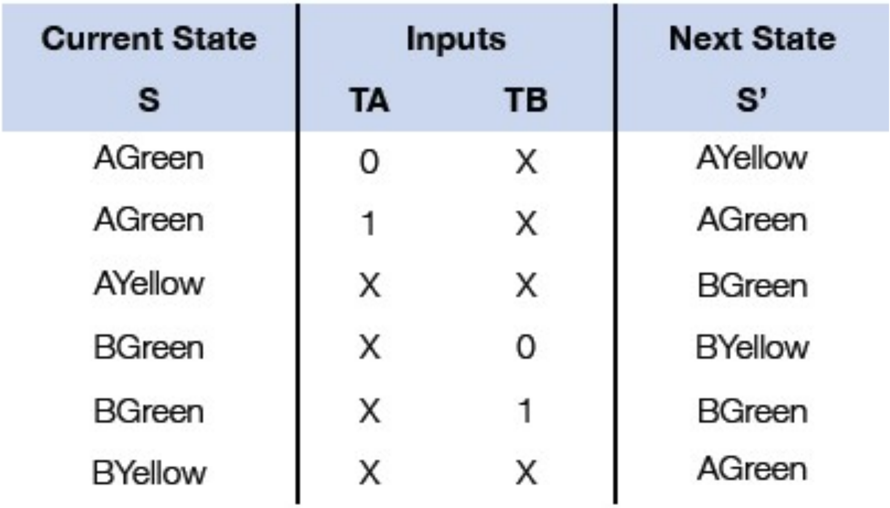
FSM State Transition Table

- State transitions from diagram can be rewritten in a state transition table

(S = current state, S' = next state)



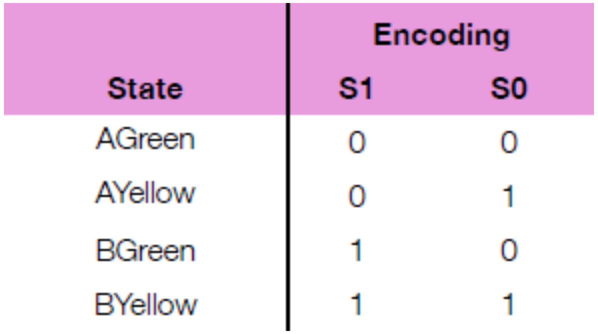
(diagram reprinted for reference)



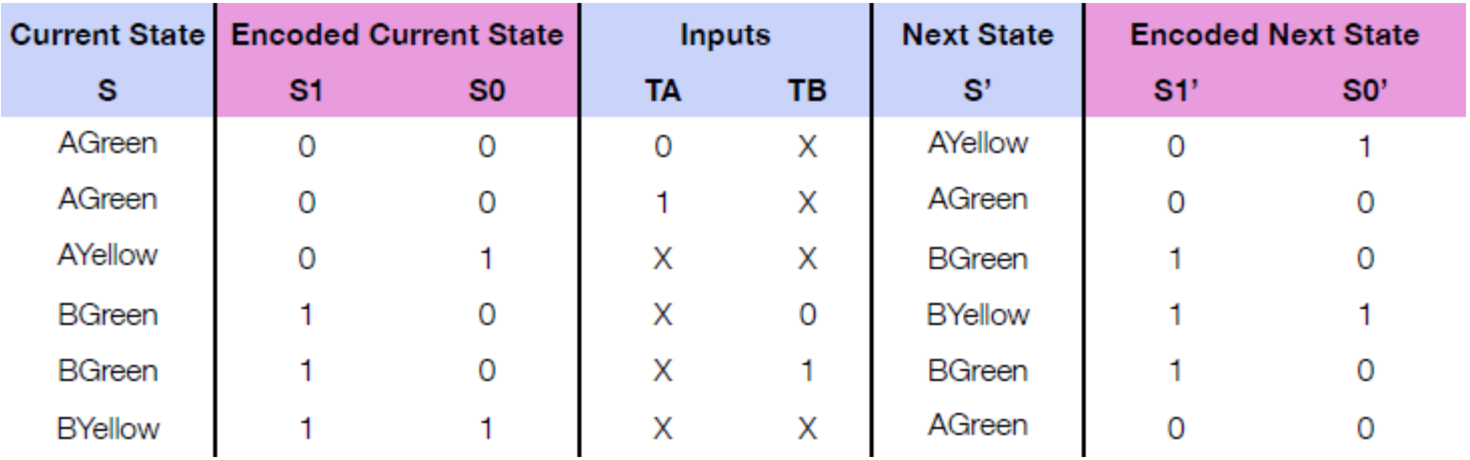
Current State	Inputs		Next State
S	TA	TB	S'
AGreen	0	X	AYellow
AGreen	1	X	AGreen
AYellow	X	X	BGreen
BGreen	X	0	BYellow
BGreen	X	1	BGreen
BYellow	X	X	AGreen

Encoded State Transition Table

- After selecting a state encoding, the symbolic states in the transition table can be realized with current state/next state bits

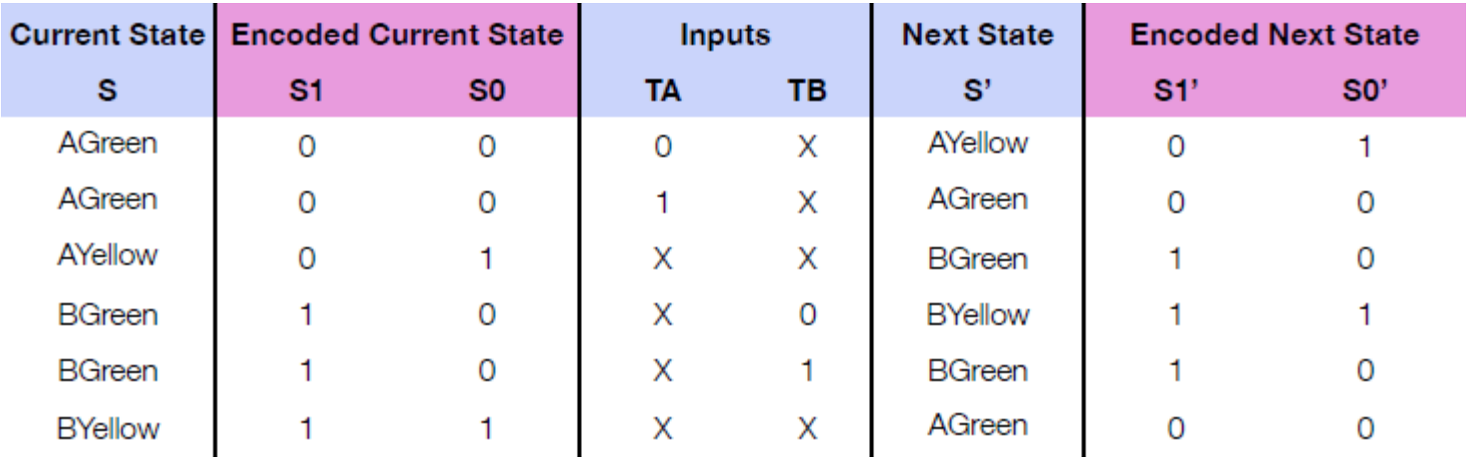


	Encoding	
State	S1	S0
AGreen	0	0
AYellow	0	1
BGreen	1	0
BYellow	1	1



Current State	Encoded Current State		Inputs		Next State	Encoded Next State	
S	S1	S0	TA	TB	S’	S1’	S0’
AGreen	0	0	0	X	AYellow	0	1
AGreen	0	0	1	X	AGreen	0	0
AYellow	0	1	X	X	BGreen	1	0
BGreen	1	0	X	0	BYellow	1	1
BGreen	1	0	X	1	BGreen	1	0
BYellow	1	1	X	X	AGreen	0	0

Computing Next State Logic

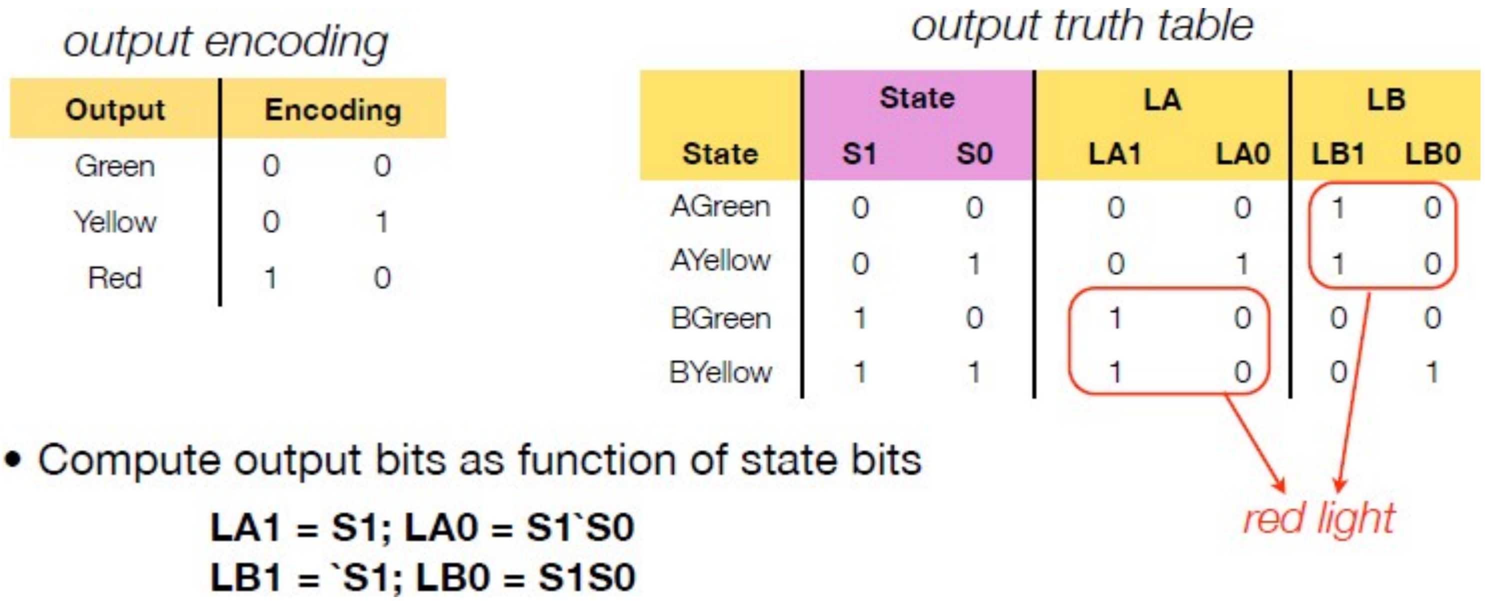


Current State	Encoded Current State		Inputs		Next State	Encoded Next State	
S	S1	S0	TA	TB	S'	S1'	S0'
AGreen	0	0	0	X	AYellow	0	1
AGreen	0	0	1	X	AGreen	0	0
AYellow	0	1	X	X	BGreen	1	0
BGreen	1	0	X	0	BYellow	1	1
BGreen	1	0	X	1	BGreen	1	0
BYellow	1	1	X	X	AGreen	0	0

- From K-maps, figure out expressions for the next state:
  - $S1(t+1) = S1(t) \text{ xor } S0(t)$
  - $S0(t+1) = \overline{S1}(t)\ \overline{S0}(t)\ \overline{TA} + S1(t)\ \overline{S0}(t)\ \overline{TB}$
- Another way of writing the same thing (just a change of notation):
  - S1' = S1 XOR S0
  - S0' = S1^S0^TA^ + S1S0^TB^

FSM Output Table

- FSM output logic is computed in similar manner as next state logic
- In this system, output is a function of current state (Moore machine)
- Alternative – Mealy machine (output function of both current state and inputs, though we won't cover this in class)



output encoding

Output	Encoding	
Green	0	0
Yellow	0	1
Red	1	0

output truth table

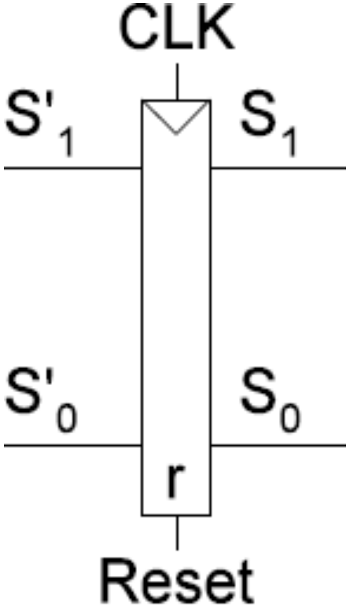
	State		LA		LB	
State	S1	S0	LA1	LA0	LB1	LB0
AGreen	0	0	0	0	1	0
AYellow	0	1	0	1	1	0
BGreen	1	0	1	0	0	0
BYellow	1	1	1	0	0	1

red light

- Compute output bits as function of state bits

LA1 = S1; LA0 = S1`S0
LB1 = `S1; LB0 = S1S0

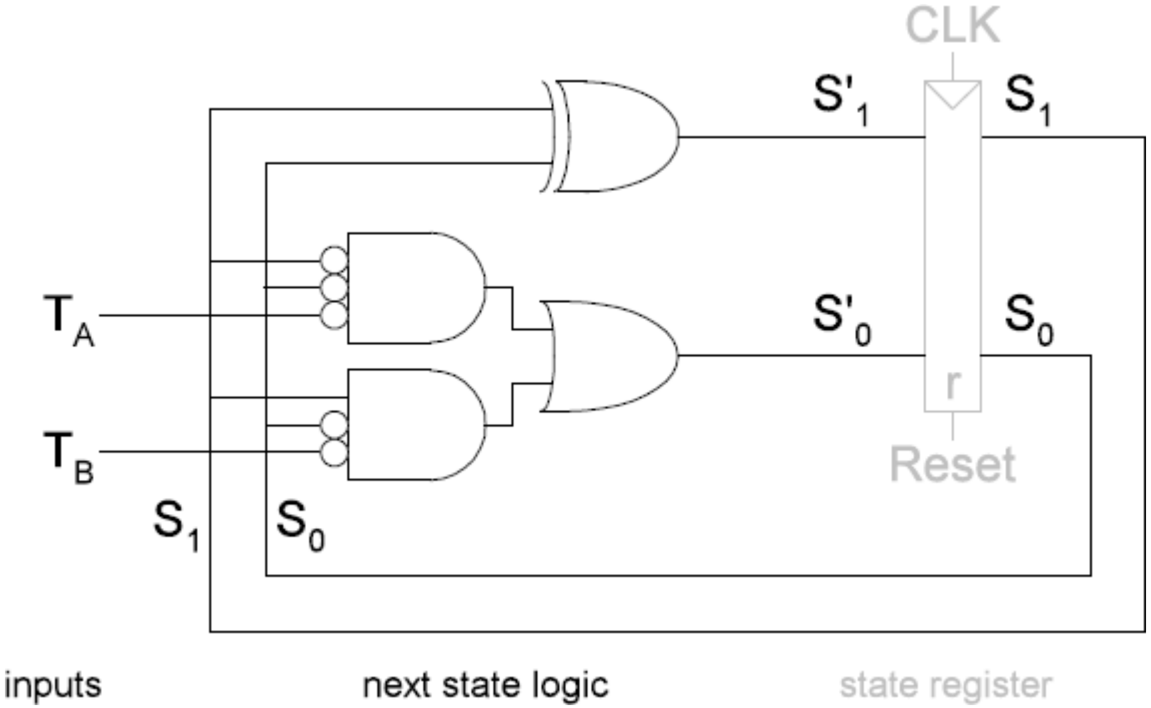
State Register: Assume D Flip-Flop



state register

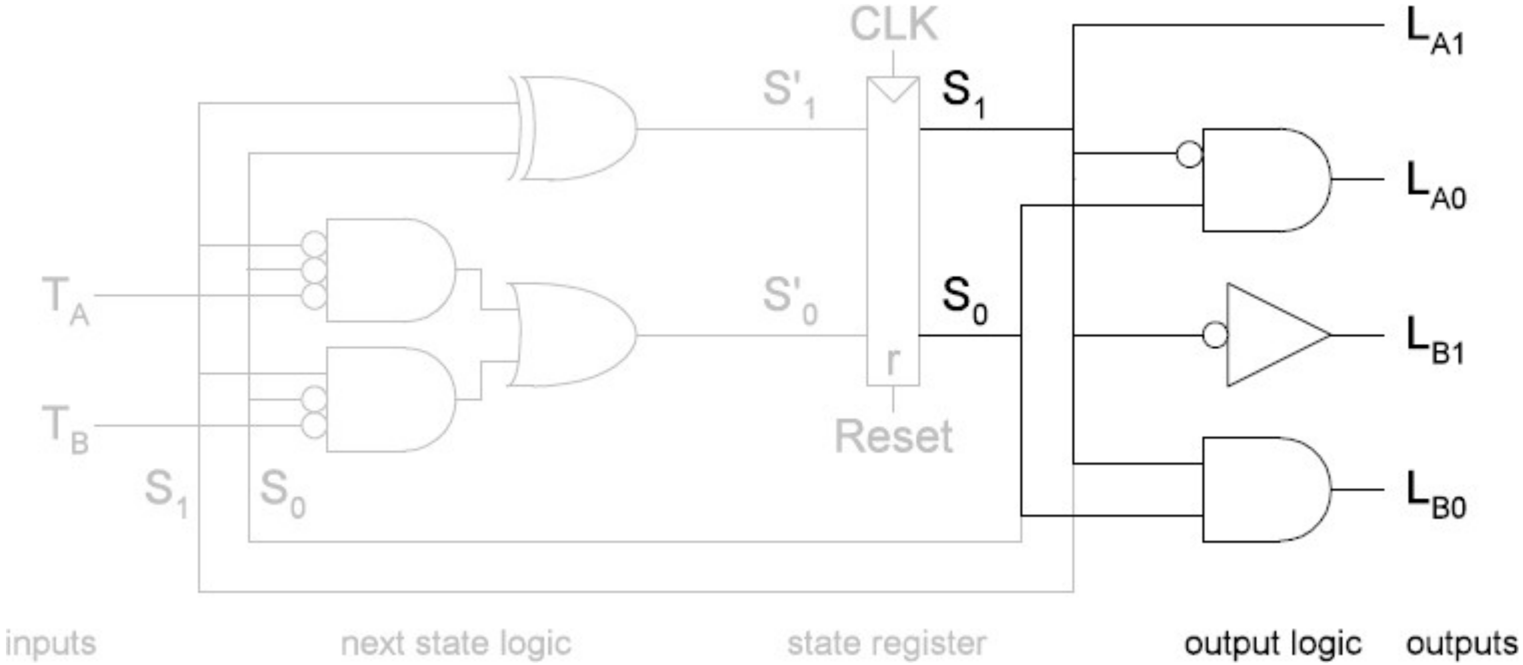
FSM: Figure out Next State Logic

- S1’ = S1 XOR S0
- S0’ = S1^S0^TA^ + S1S0^TB^



FSM: Figure out Output Logic

LA1 = S1; LA0 = S1`S0
LB1 = `S1; LB0 = S1S0



FSM: divisible by 3

- Construct a "divisible by 3" FSM that accepts a binary number entered 1 bit at a time, MSB first, and indicate with a light if the number entered so far is divisible by 3.

1) N mod 3

    0 -> N = 3p + 0
    1 -> N = 3p + 1
    2 -> N = 3p + 2

2) If the number so far is N, after a digit b is entered, the new number is N’ = 2N + b

N mod 3

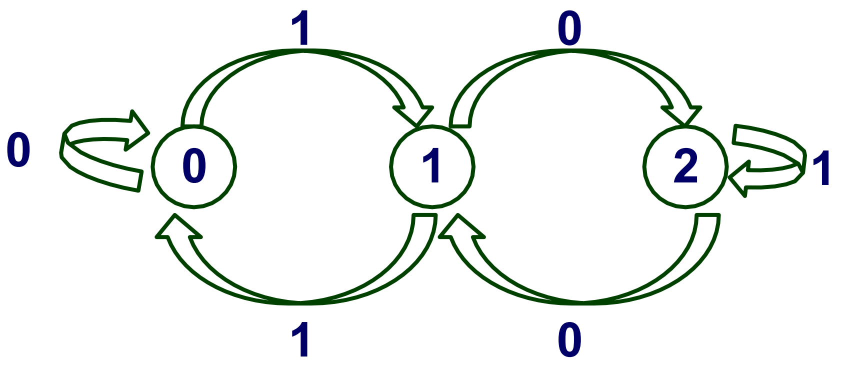
1 -> N = 3p + 0, after digit b is entered, N’ = 6p + b. N’ = b mod 3

2-> N = 3p + 1, after digit b is entered, N’ = 6p + 2 + b. N’ = b+2 mod 3

2 -> N = 3p + 2, after digit b is entered, N’ = 6p + 4 + b. N’ = b+1 mod 3

FSM: divisible by 3

1. State Transition Diagram



2. State Encoding

State	S1	S0
0	0	0
1	0	1
2	1	0

2. State Transition Table

S1	S0	b	S1’	S0’	light
0	0	0	0	0	1
0	0	1	0	1	0
0	1	0	1	0	0
0	1	1	0	0	1
1	0	0	0	1	0
1	0	1	1	0	0

3. Next State Logic

$$S1' = \overline{S1}\ S0\ \overline{b} + S1\ \overline{S0}\ b$$

$$S0' = \overline{S1}\ \overline{S0}\ b + S1\ \overline{S0}\ \overline{b}$$

4. Output Logic

$$\text{light} = \overline{S1}\ \overline{S0}$$