inst.eecs.berkeley.edu/~eecs151

EECS151 : Introduction to Digital Design and ICs

Lecture 6 – Combinational Logic

Bora Nikolić

September 14, 2021. Apple announces A15 Bionic processor

- 6 cores
- New GPU
- New neural engine
- …

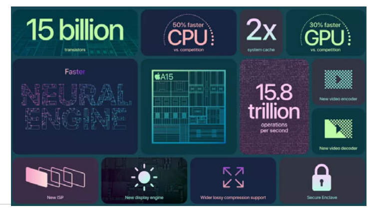


Image source: Apple

1

Review

- Sequential logic uses flip-flops and (sometimes) latches
- Flip-flops and latches are inferred in Verilog
  - Always blocks
- Practice is the best way to learn a new language…
- Blocking and non-blocking assignments

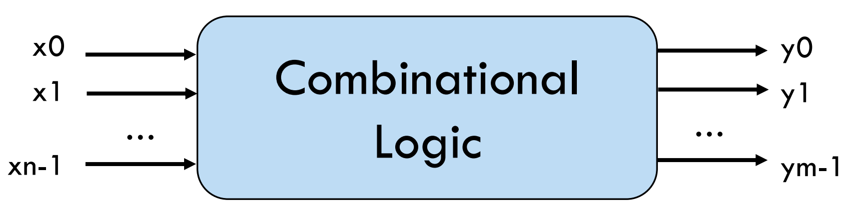
2

Combinational Logic

3

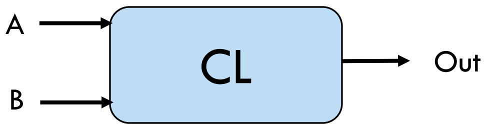
Combinational Logic

- The outputs depend *only* on the current values of the inputs.
  - Memoryless: compute the output values using the current inputs.



4

Combinational Logic Example



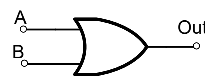
Boolean Equations:

```
Out = A OR B
  = A + B
```

Truth Table Description:

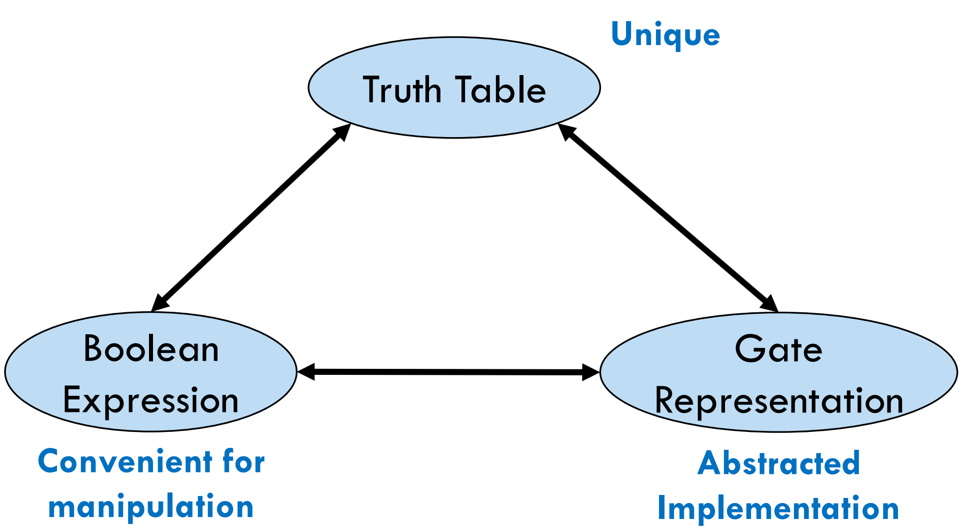
A	B	Out
0	0	0
0	1	1
1	0	1
1	1	1

Gate Representations:



5

Relationship Among Representations

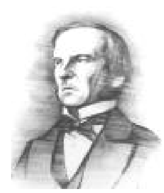


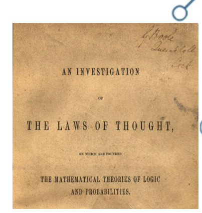
6

Boolean Algebra

Boolean Algebra Background

- Logic: The study of the principles of reasoning.
- The 19th Century Mathematician, George Boole, developed a math. system (algebra) involving logic, Boolean Algebra.
  - His variables took on TRUE, FALSE.
- Later Claude Shannon (father of information theory) showed (in his Master's thesis!) how to map Boolean Algebra to digital circuits.



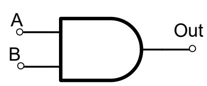


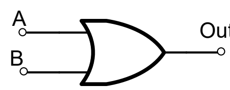
georgeboole.com
Digitized by Google

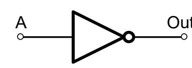
http://hdl.handle.net/1721.1/11173

Boolean Algebra Fundamentals

- Two elements {0, 1}
- Two binary operators: AND (·) OR (+)
- One unary operator: NOT (¯ , ´)







A	B	Out
0	0	0
0	1	0
1	0	0
1	1	1

A	B	Out
0	0	0
0	1	1
1	0	1
1	1	1

A	Out
0	1
1	0

9

Axioms of Boolean Algebra

Axiom	Dual	Name
B = 0 if B ≠ 1	B = 1 if B ≠ 0	Binary field
$\overline{0} = 1$	$\overline{1} = 0$	NOT
0 • 0 = 0	1 + 1 = 1	AND/OR
1 • 1 = 1	0 + 0 = 0	AND/OR
0 • 1 = 0 • 1 = 0	1 + 0 = 0 + 1 = 1	AND/OR

In mathematical logic, axioms are given
Anything else can be derived from these axioms
Each axiom has a dual

10

Boolean Operations

- Given two variables (A, B), 16 logic functions

A	B	F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_A	F_B	F_C	F_D	F_E	F_F
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

Boolean Algebra Theorems

- Null elements, identities:
  - A+0=A, A•1=A
  - A+1=1, A•0=0
- Idempotency:
  - A+A=A, A•A=A
- Complements:
  - A+A´=1, A•A´=0
- Involution:
  - (A’)’ = A
- Commutativity:
  - A+B=B+A, A•B=B•A
- Associativity:
  - (A + B) + C= A + (B + C)= A + B + C
  - (A • B) • C = A • (B • C) = A • B • C
- Distributivity:
  - A • (B+C) = (A•B) + (A•C)
  - A + (B•C) = (A+B) • (A+C)
- Covering:
  - A • (A+B) = A, A + (A•B) = B
- Consensus
  - (A • B) + (A’ • C) + (B • C) = (A • B) + (A’ • C)

Literals are variables or their complements

Proving Distributive Law

- A • (B+C) = (A•B) + (A•C)

A	B	C	(B+C)	A • (B+C)	(A•B)	(A•C)	(A•B) + (A•C)
0	0	0					
0	0	1					
0	1	0					
0	1	1					
1	0	0					
1	0	1					
1	1	0					
1	1	1					

Proving Distributive Law

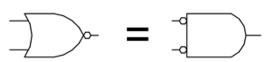
- A • (B+C) = (A•B) + (A•C)

A	B	C	(B+C)	A • (B+C)	(A•B)	(A•C)	(A•B) + (A•C)
0	0	0	0	0	0	0	0
0	0	1	1	0	0	0	0
0	1	0	1	0	0	0	0
0	1	1	1	0	0	0	0
1	0	0	0	0	0	0	0
1	0	1	1	1	0	1	1
1	1	0	1	1	1	0	1
1	1	1	1	1	1	1	1

DeMorgan's Law

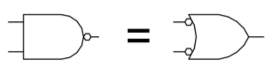
- Theorem for complementing a complex function.

(A + B)' = A' B'



A	B	A'	B'	(A + B)'	A' B'
0	0				
0	1				
1	0				
1	1				

(A B)' = A' + B'



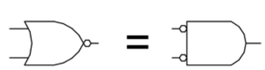
A	B	A'	A'	(A B)'	A' + B'
0	0				
0	1				
1	0				
1	1				

15

DeMorgan's Law

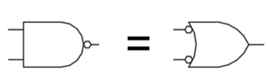
- Procedure for complementing a complex function.

(A + B)' = A' B'



A	B	A'	B'	(A + B)'	A' B'
0	0	1	1	1	1
0	1	1	0	0	0
1	0	0	1	0	0
1	1	0	0	0	0

(A B)' = A' + B'



A	B	A'	B'	(A B)'	A' + B'
0	0	1	1	1	1
0	1	1	0	1	1
1	0	0	1	1	1
1	1	0	0	0	0

16

Canonical Forms

- Two types:
  - Sum of Products (SOP)
  - Product of Sums (POS)
- Sum of Products
  - a.k.a Disjunctive normal form, minterm expansion
  - Minterm: a product (AND) involving all inputs
  - SOP: Summing minterms for which the output is True

Minterms	a	b	c	f	f'
a'b'c'	0	0	0	0	1
a'b'c'	0	0	1	0	1
a'b c'	0	1	0	0	1
a'b c	0	1	1	1	0
a b'c'	1	0	0	1	0
a b'c	1	0	1	1	0
a b c'	1	1	0	1	0
a b c	1	1	1	1	0

One product (and) term for each 1 in f:

```
f = a'bc + ab'c' + ab'c + abc' + abc
f' = a'b'c' + a'b'c + a'bc'
```


Sum of Products (cont.)

- Canonical Forms are usually not minimal:
- Example:

```
f  = a'bc + ab'c' + ab'c + abc' + abc   (xy' + xy = x)
   = a'bc + ab' + ab
   = a'bc + a
   = a + bc

f' = a'b'c' + a'b'c + a'bc'
   = a'b' + a'bc'
   = a' ( b' + bc' )
   = a' ( b' + c' )
   = a'b' + a'c'
```

a + a'b = a + b

- Recall distributive theorem
  a+bc = (a+b)(a+c)

Canonical Forms

- Two types:
  - Sum of Products (SOP)
  - Product of Sums (POS)
- Product of Sums:
  - a.k.a. conjunctive normal form, maxterm expansion
  - Maxterm: a sum (OR) involving all inputs
  - POS: Product (AND) maxterms for which the output is FALSE
  - Can obtain POSs from applying DeMorgan's law to the SOPs of F (and vice versa)

Maxterms	a	b	c	f	f'
a+b+c	0	0	0	0	1
a+b+c'	0	0	1	0	1
a+b'+c	0	1	0	0	1
a+b'+c'	0	1	1	1	0
a'+b+c	1	0	0	1	0
a'+b+c'	1	0	1	1	0
a'+b'+c	1	1	0	1	0
a'+b'+c'	1	1	1	1	0

One sum (**or**) term for each **0** in **f**:

```
f  = (a+b+c)(a+b+c')(a+b'+c)
f' = (a+b'+c')(a'+b+c)(a'+b+c')
     (a'+b'+c)(a+b+c')
```


Quiz

- Derive the product-of-sums form of $\overline{Y}$ based on the truth table.

a) $\overline{Y} = (A+B)(A+\overline{B})$

b) $\overline{Y} = A\overline{B} + AB$

c) $\overline{Y} = \overline{A}\overline{B} + \overline{A}B$

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

Boolean Simplification

Example: Full Adder (FA) Carry out

```
co = a'bc + ab'c + abc' + abc
   = a'bc + ab'c + abc' + abc + abc
   = a'bc + abc + ab'c + abc' + abc
   = (a' + a)bc + ab'c + abc' + abc
   = (1)bc + ab'c + abc' + abc
   = bc + ab'c + abc' + abc + abc
   = bc + ab'c + abc + abc' + abc
   = bc + a(b' +b)c + abc' +abc
   = bc + a(1)c + abc' + abc
   = bc + ac + ab(c' + c)
   = bc + ac + ab(1)
   = bc + ac + ab
```

c	b	c	s	co
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Why do Boolean simplification?

- Minimize number of gates in circuit
  - Gates take area
- Minimize amount of wiring in circuit
  - Wiring takes space and is difficult to route
  - Physical gates have limited number of inputs
- Minimize number of gate levels
  - Faster is better
- How to systematically simplify Boolean logics?
  - Use tools!

Practical methods for Boolean simplification

- Still based on Boolean algebra, but more systematic
- 2-level simplification -> multilevel
- Key tool: The Uniting Theorem

$$ab' + ab = a\ (b' + b) = a\ (1) = a$$

ab	f
00	0
01	0
10	1
11	1

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

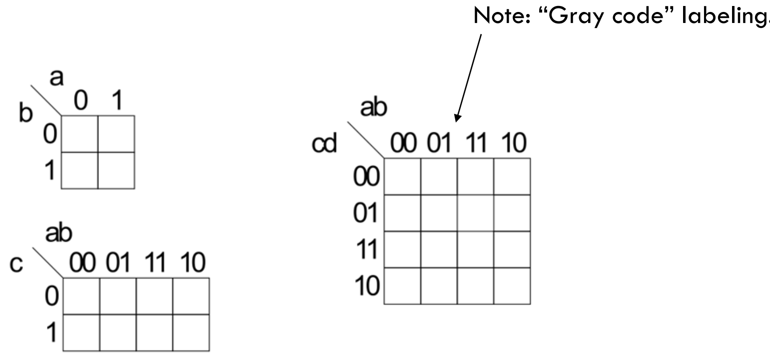
b values change within rows

a values don't change

b is eliminated, **a** remains

Karnaugh Map Method

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



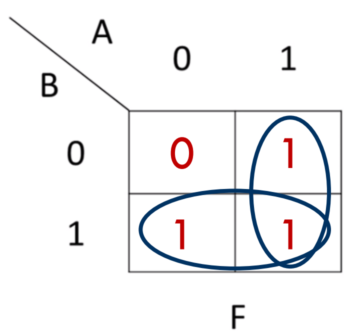
Karnaugh Map Method

- Adjacent groups of 1’s represent product terms

OR

A	B	F
0	0	0
0	1	1
1	0	1
1	1	1

Karnaugh Map



F = A + B

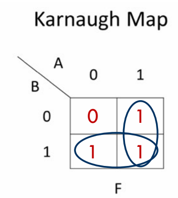
Karnaugh Map Method

1. Draw K-map of the appropriate number of variables.
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, …)
   - ✓ <u>Form as large as possible groups and as few groups as possible.</u>
   - ✓ 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.

OR

A	B	F
0	0	0
0	1	1
1	0	1
1	1	1

Karnaugh Map



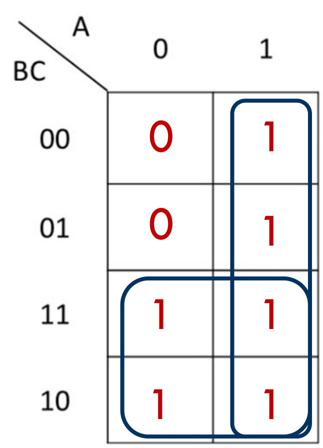
F = A + B

27

Karnaugh Map Method

A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

F = A + B



28

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.

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

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

Karnaugh Maps with Don't Cares

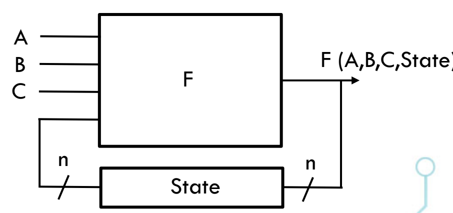
- Don't cares (x's) in the truth table can be either 0's or 1's

CD \ AB	00	01	11	10
00	1	1	X	1
01	1	0	X	1
11	1	1	X	X
10	1	0	X	X

Finite-State Machines

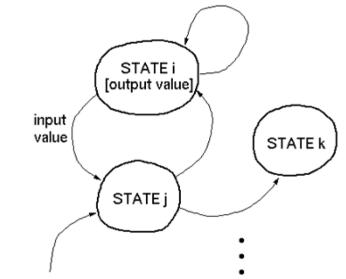
Sequential logic

- Combinational logic:
  - Memoryless: the outputs only dependent on the current inputs.
- Sequential logic:
  - Memory: the outputs depend on both current and previous values of the inputs.
    - Distill the prior inputs into a smaller amount of information, i.e., states.
  - State: the information about a circuit
    - Influences the circuit's future behavior
    - Stored in Flip-flops and Latches
  - Finite State Machines:
    - Useful representation for designing sequential circuits
    - As with all sequential circuits: output depends on present and past inputs
    - We will first learn how to design by hand then how to implement in Verilog.

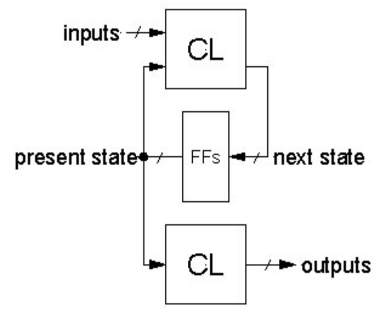


Finite State Machines

- A sequential circuit which has
  - External inputs
  - Externally visible outputs
  - Internal states
- Consists of:
  - State register
    - Stores current state
    - Loads previously calculated next state
    - # of states <= 2^(# of FFs)
  - Combinational logic
    - Computes the next state
    - Computes the outputs



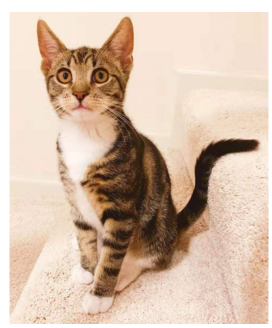
State Transition Diagram

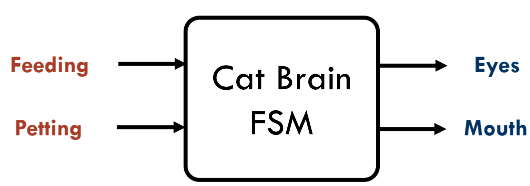


33 Berkeley

FSM Example

- Cat Brain (Simplified…)
  - Inputs:
    - Feeding
    - Petting
  - Outputs:
    - Eyes: open or close
    - Mouth: open or close
  - States:
    - Eating
    - Sleeping
    - Annoyed…

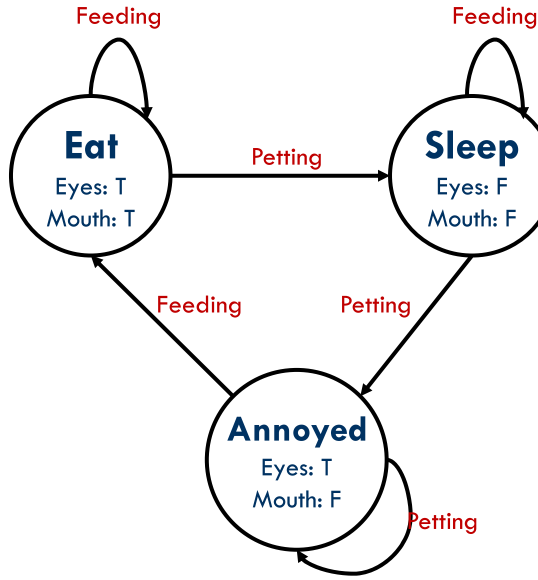




34 Berkeley

FSM State Transition Diagram

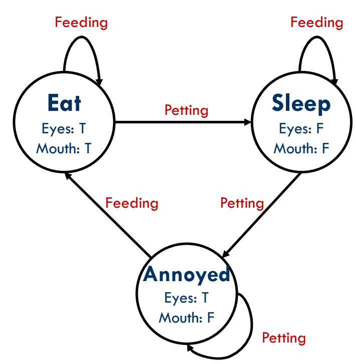
- States:
  - Circles
- Outputs:
  - Labeled in each state
  - Arcs
- Inputs:
  - Arcs



35

FSM Symbolic State Transition Table

Current State	Inputs	Next State
Eat	Feeding	Eat
Eat	Petting	Sleep
Sleep	Feeding	Sleep
Sleep	Petting	Annoyed
Annoyed	Feeding	Eat
Annoyed	Petting	Annoyed

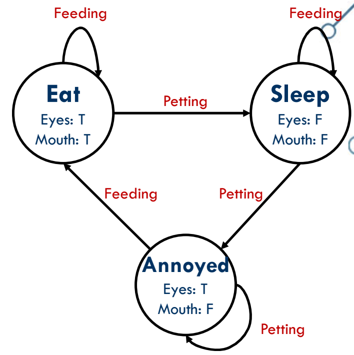


36

FSM Encoded State Transition Table

State	Encoding
Eat	00
Sleep	01
Annoyed	10

Current State		Input	Next State	
S1	S0	X	S1'	S0'
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	0
1	0	1	1	0



Current State	Inputs	Next State
Eat	Feeding	Eat
Eat	Petting	Sleep
Sleep	Feeding	Sleep
Sleep	Petting	Annoyed
Annoyed	Feeding	Eat
Annoyed	Petting	Annoyed

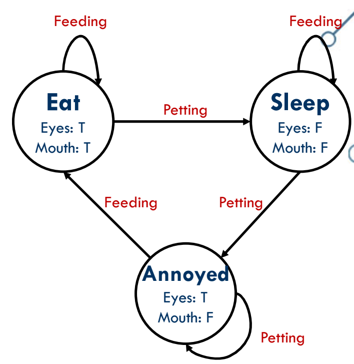
$$S0' = \overline{S1}\,\overline{S0}X + \overline{S1}S0\overline{X} = \overline{S1}\left(\overline{S0}X + S0\overline{X}\right) = \overline{S1}(S0 \oplus X)$$
$$S1' = \overline{S1}S0X + S1\overline{S0}X = (S1 \oplus S0)X$$

 Berkeley

FSM Output Table

State	Encoding
Eat	00
Sleep	01
Annoyed	10

Current State		Outputs	
S1	S0	E	M
0	0	1	1
0	1	0	0
1	0	1	0

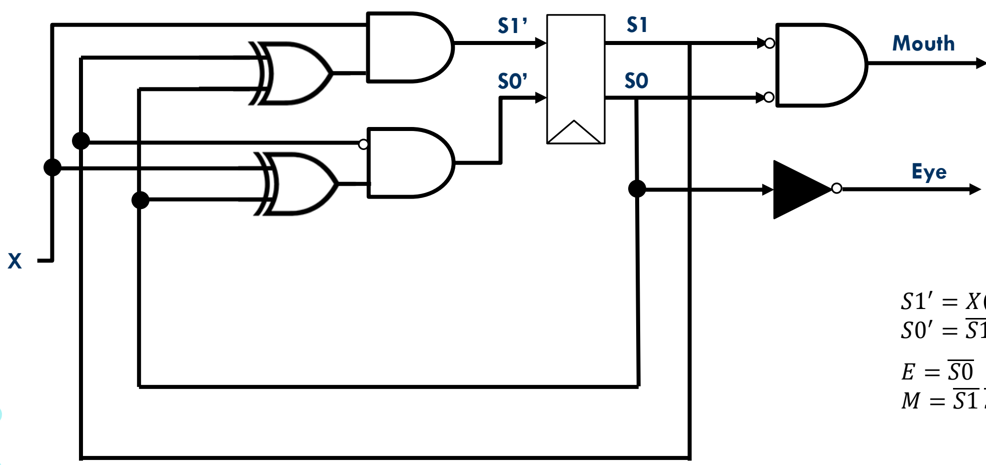


Outputs		Encoding
Eyes	Mouth	
Open	Open	11
Close	Close	00
Open	Close	10

$$E = \overline{S1}\,\overline{S0} + S1\overline{S0} = \overline{S0}$$
$$M = \overline{S1}\,\overline{S0}$$

 Berkeley

FSM Gate Representation



$$S1' = X(S0 \oplus S1)$$
$$S0' = \overline{S1}(S0 \oplus X)$$
$$E = \overline{S0}$$
$$M = \overline{S1}\,\overline{S0}$$

Summary

- Combinational logic:
  - The outputs only depend on the current values of the inputs (memoryless)
  - The functional specification of a combinational circuit can be expressed as:
    - A truth table
    - A Boolean equation
- Boolean algebra
  - Deal with variables that are either True or False
  - Map naturally to hardware logic gates
  - Use theorems of Boolean algebra and Karnaugh maps to simplify equations
- Finite state machines: Common example of sequential logic
- Common job interview questions ☺