CSE 260M / ESE 260
Intro. To Digital Logic & Computer Design

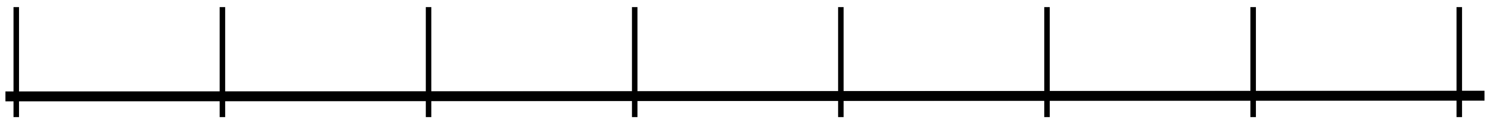
Bill Siever
&
Michael Hall

Modules 1-4A

Module 1

Binary

- Counting / bases
  - (Unsigned) Binary: 0, 1, 10, 11, …
  - One-to-one correspondence with natural numbers



Decimal:	0	1	2	3	4	5	6	7
Binary:	000	001	010	011	100	101	110	111

Place Value: Base b (to decimal)

Digits	D2	D1	D0
Place Value	b^2	b^1	b^0
Place Value In terms of Base	D2*b^2	D1*b^1	D0*b^0

Highlights

- Different bases have different strengths (for this course & computing)
  - Decimal: Human's "first learned base" (typically)
  - Binary: On/off notation is convenient for building machines
  - Hexadecimal: More human friendly than binary, but direct conversion to/from binary
  - Arithmetic in all (these) bases can be though about based number lines (Ex: Addition of positives is moving to the right on the number line)

Highlights

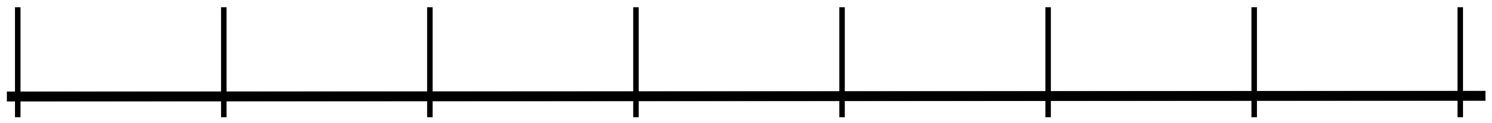
- Often used for “encoding” — using a binary number to represent some concept
  - Ex: State Encodings represent the concept of a state machine’s state.
    (Numeric value, but numeric value may not be significant)
  - Ex: ASCII (encoding English letters)

Negatives

- We’ve focused on numbers with a fixed width (i.e., n digits)
- Multiple representations
  - Sign/Magnitude: Like the notion of the “-“ in decimal representations
  - Two’s Complement: Divide the number line into non-neg and negs. *Convenient*:
    - Can use modified place-value rules to do human-friendly conversions.
    - Can use same rules for addition as with unsigned

Dividing the Line

- Split the line in half, like we've already done.
  - How can we identify if a binary number is positive or negative?
    - Ex: 010?   Or 110?



Decimal:	0	1	2	3	4	5	6	7
Binary:	000	001	010	011	100	101	110	111
2's comp behavior:					-4	—3	-2	-1

Module 2

Boolean Algebra

- Rules / processes to manipulate true/false statements and values
- Formal Algebraic Rules
  (partly from from George Boole’s “The Mathematical Analysis of Logic”)
  - Basic operations: AND, OR, Not
- Doesn’t represent notion of time or real-world behavior

Boolean Algebra

Table 2.1 **Axioms of Boolean algebra**

	Axiom		Dual	Name
A1	$B = 0$ if $B \neq 1$	A1′	$B = 1$ if $B \neq 0$	Binary field
A2	$\overline{0} = 1$	A2′	$\overline{1} = 0$	NOT
A3	$0 \bullet 0 = 0$	A3′	$1 + 1 = 1$	AND/OR
A4	$1 \bullet 1 = 1$	A4′	$0 + 0 = 0$	AND/OR
A5	$0 \bullet 1 = 1 \bullet 0 = 0$	A5′	$1 + 0 = 0 + 1 = 1$	AND/OR

Boolean Algebra

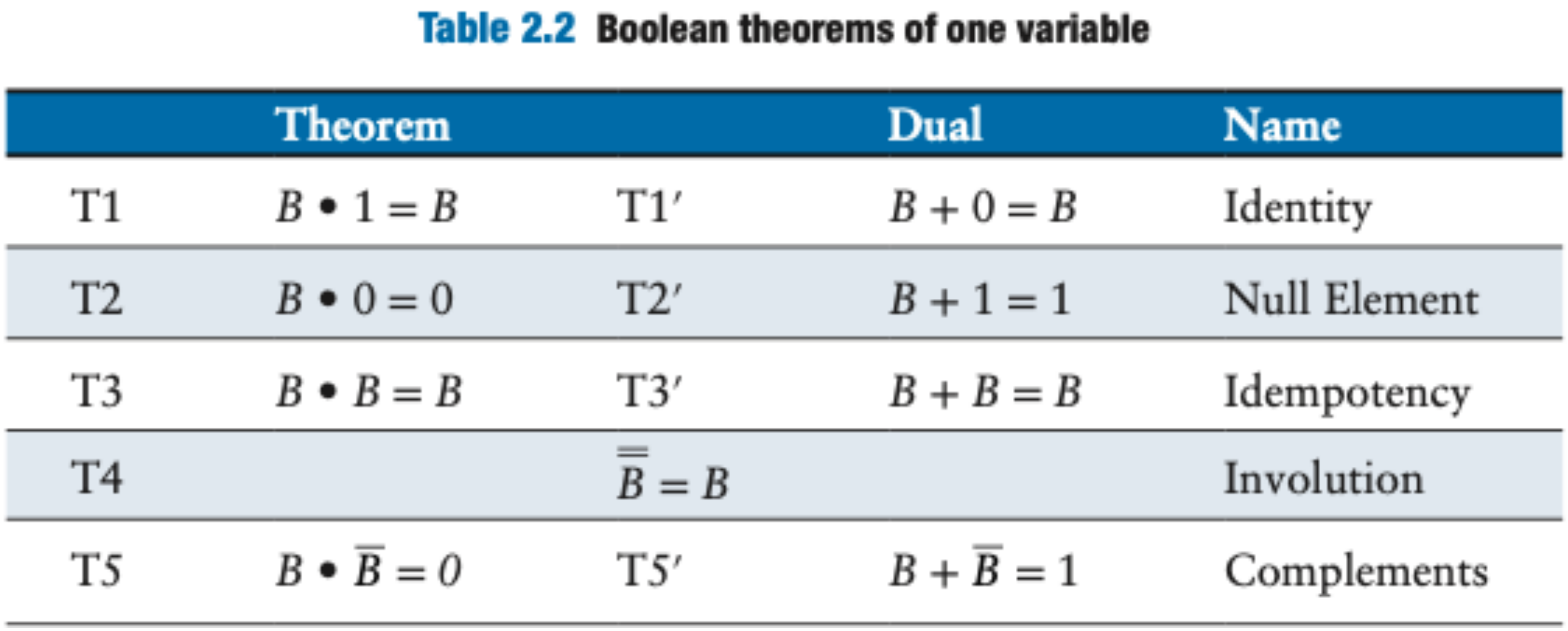


Table 2.2 **Boolean theorems of one variable**

	Theorem		Dual	Name
T1	$B \bullet 1 = B$	T1′	$B + 0 = B$	Identity
T2	$B \bullet 0 = 0$	T2′	$B + 1 = 1$	Null Element
T3	$B \bullet B = B$	T3′	$B + B = B$	Idempotency
T4		$\overline{\overline{B}} = B$		Involution
T5	$B \bullet \overline{B} = 0$	T5′	$B + \overline{B} = 1$	Complements

Boolean Algebra

Table 2.3 **Boolean theorems of several variables**

	Theorem		Dual	Name
T6	$B \bullet C = C \bullet B$	T6′	$B + C = C + B$	Commutativity
T7	$(B \bullet C) \bullet D = B \bullet (C \bullet D)$	T7′	$(B + C) + D = B + (C + D)$	Associativity
T8	$(B \bullet C) + (B \bullet D) = B \bullet (C + D)$	T8′	$(B + C) \bullet (B + D) = B + (C \bullet D)$	Distributivity
T9	$B \bullet (B + C) = B$	T9′	$B + (B \bullet C) = B$	Covering
T10	$(B \bullet C) + (B \bullet \bar{C}) = B$	T10′	$(B + C) \bullet (B + \bar{C}) = B$	Combining
T11	$(B \bullet C) + (\bar{B} \bullet D) + (C \bullet D)$ $= (B \bullet C) + (\bar{B} \bullet D)$	T11′	$(B + C) \bullet (\bar{B} + D) \bullet (C + D)$ $= (B + C) \bullet (\bar{B} + D)$	Consensus
T12	$\overline{B_0 \bullet B_1 \bullet B_2 \ldots}$ $= (\bar{B}_0 + \bar{B}_1 + \bar{B}_2 \ldots)$	T12′	$\overline{B_0 + B_1 + B_2 \ldots}$ $= (\bar{B}_0 \bullet \bar{B}_1 \bullet \bar{B}_2 \ldots)$	De Morgan's Theorem

Tables & Truth

- Truth tables: Provide full behavioral description of inputs/outputs
- Common Digital Logic representations:
  - AND via “multiplication”
  - OR via “addition”
- Why? Order of operations is familiar
- Result: A “product term” is an AND expression

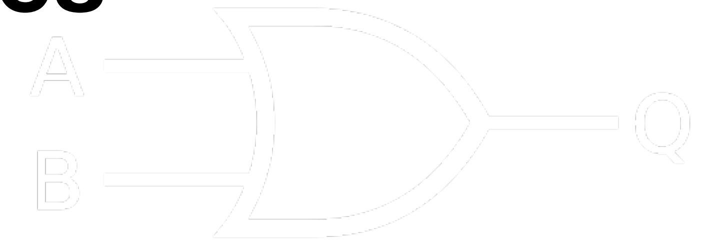
Tables & Truth

- Minterm:
  Product term that includes all input variables once (possibly with a negation)
  - Corresponds to “Selecting a row” of a truth table
- Canonical Sum-of-Products form: Sum (OR) term of all the Minterms for an output
- Look-up-table (LUT): Idea of “looking up” a set of inputs in a truth table
  to determine the ouput

Karnaugh Maps

- Convenient visual tool for a type of optimization
  - Allows easily combining terms
  - Limited to ~4 variables (typically)
  - If possible, reduces width of AND gates (smaller product terms) and number of OR terms (smaller sum)

Digital Machines



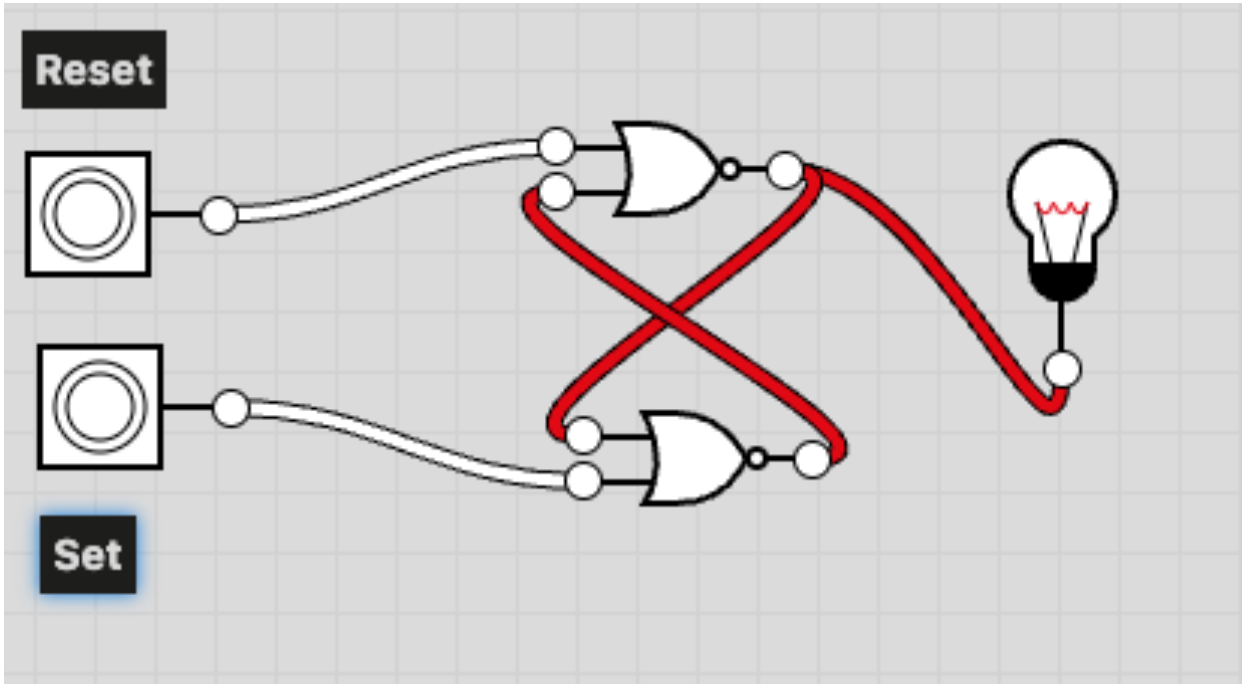
- Schematic symbols
    - “Schematic Capture” : Converting design to digital format s(like creating a circuit diagram in JLS)

Machines introduce practical concerns

- How fast/slow is it?
- How much space does it take?
- How much power does it require? / How much does it cost to operate?
- How much does it cost to construct?
- Leads to: How do different implementations compare?
  and “What is best”? (Best for what?)

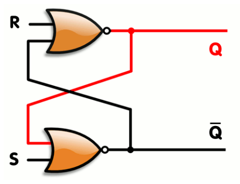
Module 3
Beyond Combinational Logic
(Beyond things that can be simple tables)

SR Latch: A way to store a value!



D-*Latch*: A *better* way to store *data*

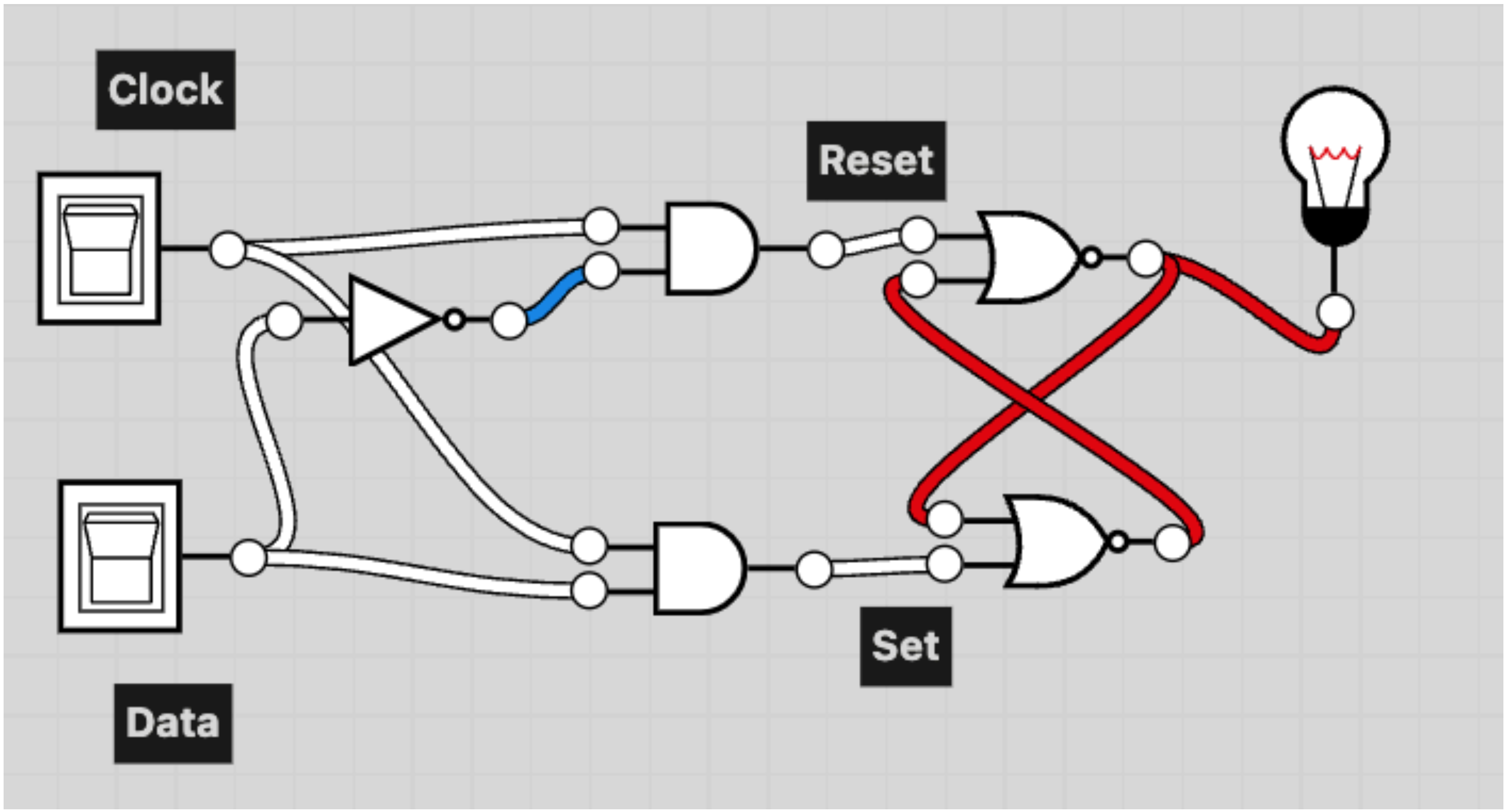
- Start with SR Latch



- Describe Desired Behavior (of output, Q)
- Just combinational logic
- Reset = Clock * /Data
  Set = Clock * Data

CLOCK	DATA	Q
0	0	
0	1	
1	0	RESET
1	1	SET

D-Latch

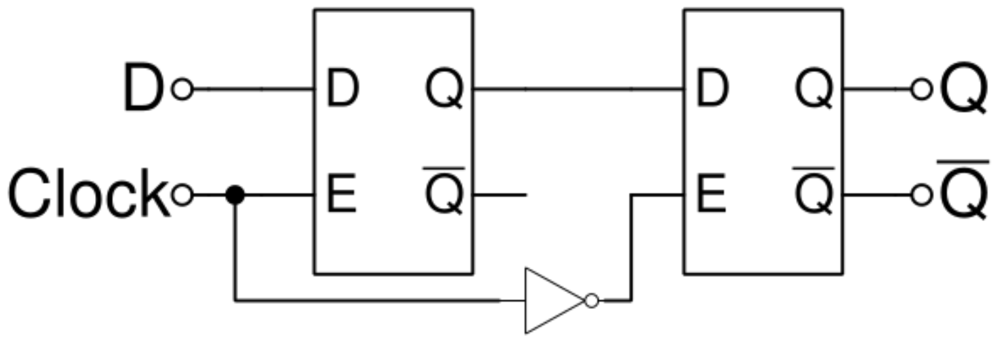


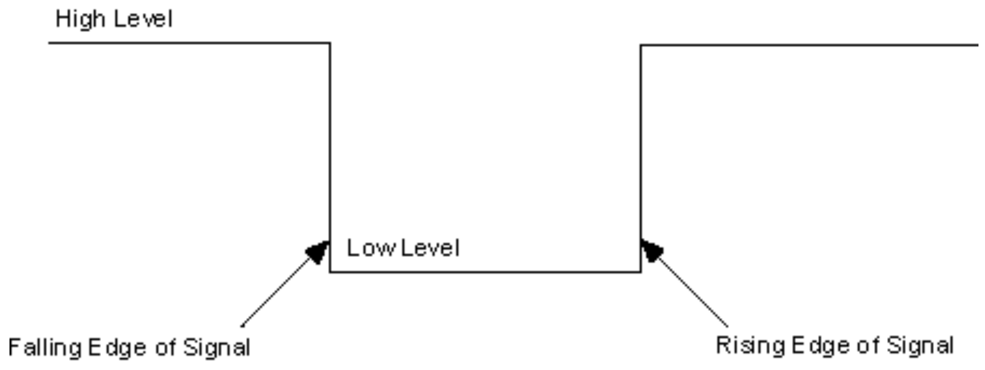
D-Latch

- “Latches on” to last data value when clock goes low
    - Is sensitive to the *level* of the clock
    - Is “transparent” when the clock is high
- A bit inconvenient

D *Flip-Flop*: More precision

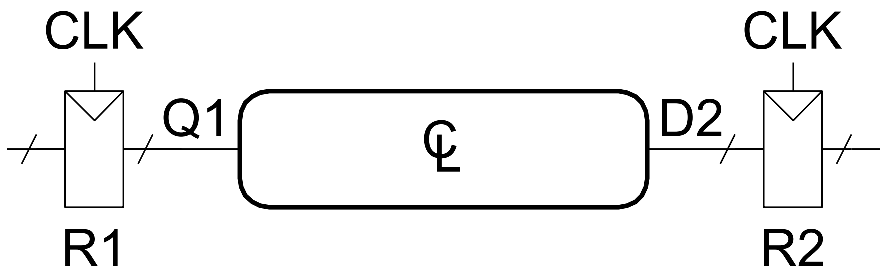
- Combines two D-latches using opposite levels
- Results in behavior that is *edge sensitive*: Very precise





Synchronous Sequential Circuits

- *Synchronized* by a clock
  - Utilize D Flop Flops and combinational logic
  - Clock can ensures proper behavior

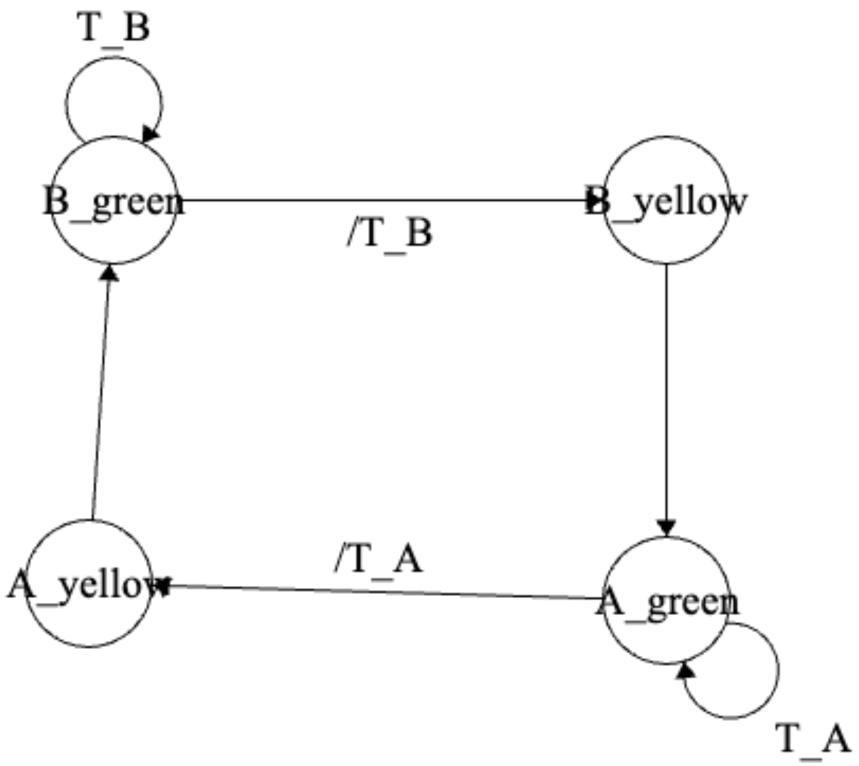


(Deterministic) Finite State Machines

- Concept that can be used for many practical problems
  - States: The current “condition” of the machine
    (Requiring some concept of current location)
  - Arc: Describe why/when to change states (based on inputs)
  - Outputs: Based on state and input (latter in Mealy machines)
  - Our implementations: Clock controls timing of when states may change

State Machines

- Just combine prior ideas
  - Binary encoding / State table: Uses binary encoding to represent state
  - State tables: Truth table that captures the “arcs”
  - Output tables: Truth table that captures how outputs behave
- All those are simple combinational logic concepts:
  - Can be represented with equations
  - Can be built using gates
  - Specific design can be depicted with gate-level schematic


T_B
B_green
/T_B
B_yellow
/T_A
A_yellow
A_green
T_A

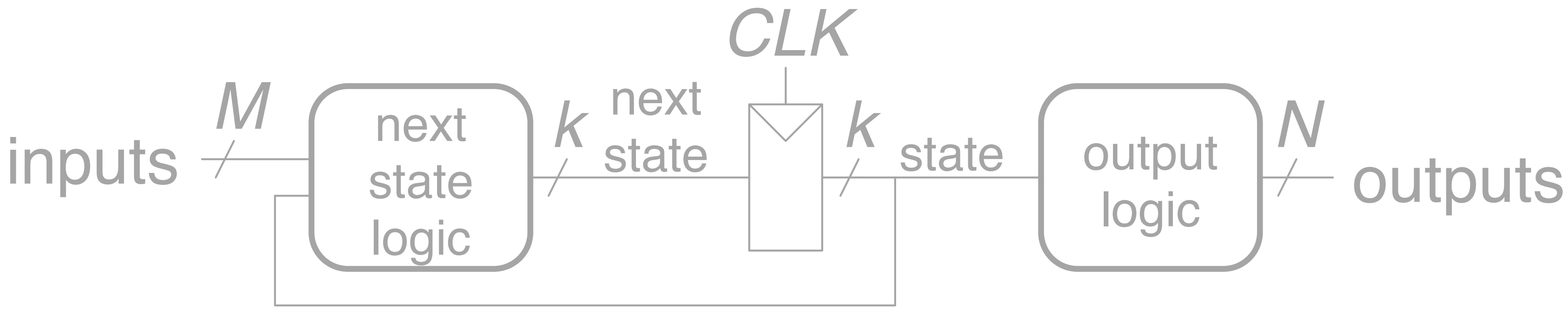
FSM: Moore Machine Structure

$t_{pd} = 10$

$t_{cd} = 8$

$t_{pd} = 9$

$t_{cd} = 7$



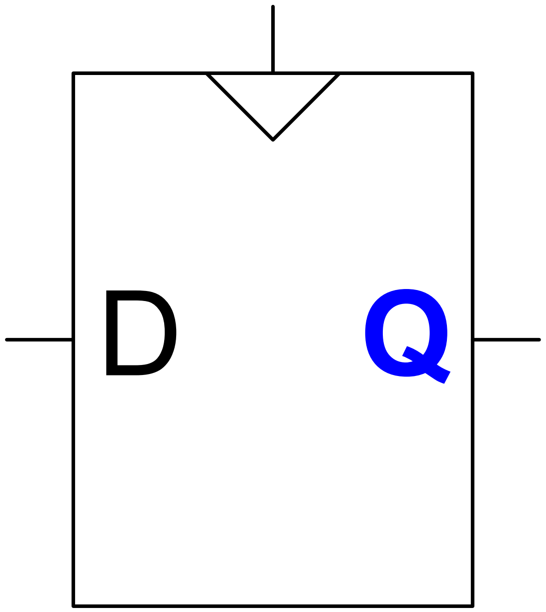
t_{pcq}: 6 t_{ccq}: 2 t_{setup}:6

t_{hold}: What's needed?

What's the max clock?

Dff Time Parameters

- t_{pcq}: Propagation delay from Clock to Q (pcq)
- t_{ccq}: Contamination delay from C to Q (ccq)
- t_{setup}: Setup time (for d before clock)
- t_{hold}: Hold time (for d after clock)



Module 4

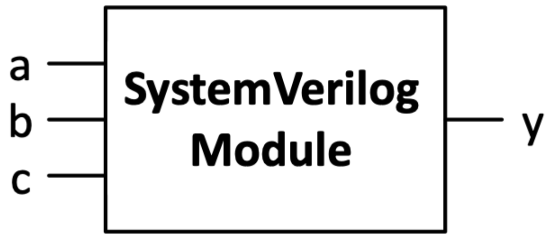
Hardware Description Languages (HDLs)

- Specifies logic function only
- Computer-aided design (CAD) tool produces or synthesizes the optimized gates
- Most commercial designs built using HDLs

HDL

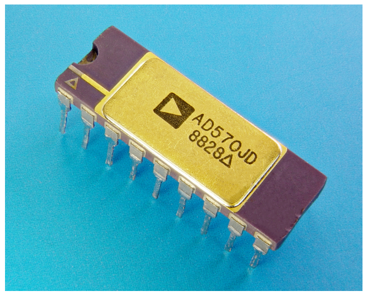
- A HDL is ***NOT*** a computer program!

(System) Verilog Module Example

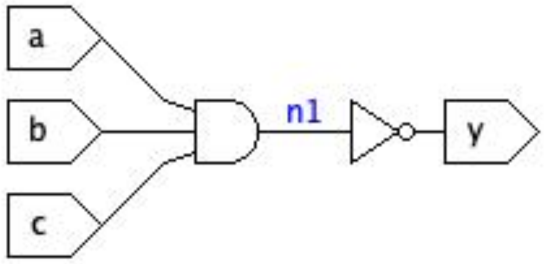


```
module example(input  logic a, b, c,
               output logic y);
  // module body goes here
endmodule
```

Input & Output are like the Pins On chips or in JLS



HDL: *Structural* (Verilog)



```
module nand3(input  logic a, b, c
             output logic y);
  logic n1;                      // internal signal

  and  my_and(n1, a, b, c);      // instance of and3
  not  my_inverter(y,n1);        // instance of inv
endmodule
```

HDL: *Behavioral* (Verilog)

```
module nand3(input  logic a, b, c
             output logic y);
   assign y = ~(a & b & c);
endmodule
```