



CSE140: Components and Design Techniques for Digital Systems

Adders, subtractors comparators, multipliers
and other ALU elements



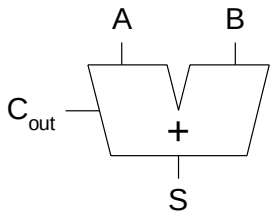
Adders

Circuit Delay

- Transistors have intrinsic resistance and capacitance
- Signals take time to propagate from the input to the output of a gate
- Sometimes delays are labeled as @<delay_value> in circuit drawings

1-Bit & Multi-bit Adders

Half Adder

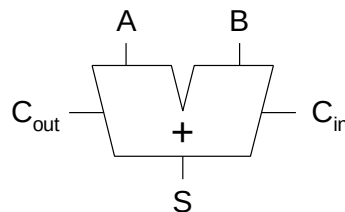


A	B	C_{out}	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$$S = A \oplus B$$

$$C_{out} = AB$$

Full Adder



C_{in}	A	B	C_{out}	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

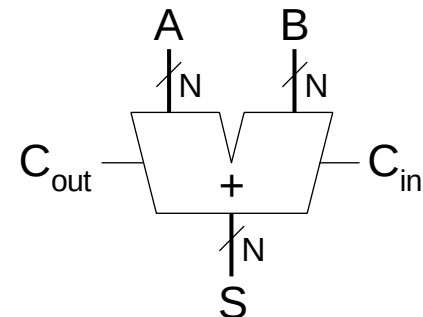
$$S = A \oplus B \oplus C_{in}$$

$$C_{out} = AB + AC_{in} + BC_{in}$$

Types of multi-bit adders

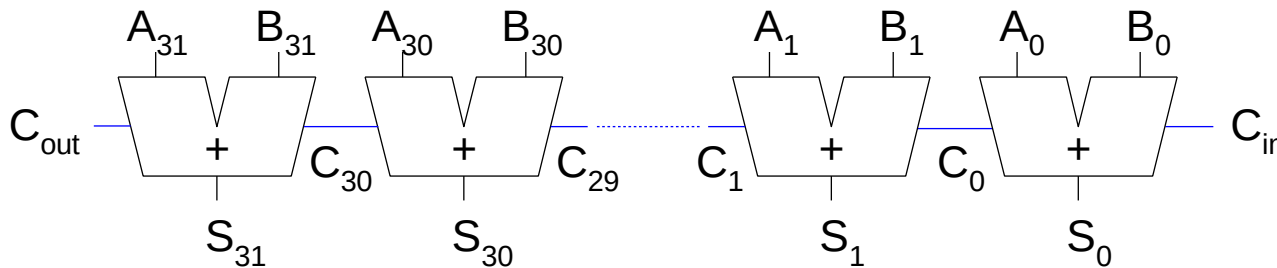
- Ripple-carry (slow)
- Carry-lookahead (faster)
- Two-level logic adder (even faster)

Symbol



Ripple-Carry Adder

- Chain 1-bit adders together
- Carry ripples through entire chain
- Disadvantage: **slow**



- Ripple-carry adder delay

$$t_{\text{ripple}} = Nt_{FA}$$

where t_{FA} is the delay of a full adder

Two-level Logic Adder

- No matter how many inputs you have,
 - look at the truth table,
 - convert to Kmap,
 - apply the algorithm for two-level logic minimization
- Very fast adder, but....
- Beyond 8 inputs, a shockingly large amount of gates!
 - Number of gates increases exponentially

Ripple carry
adder

Carry-lookahead
adder (next slide)

Two-level logic
adder



FAST



COMPLEX

Carry-lookahead adders

c4	c3	c2	c1	c0	Carries
	a3	a2	a1	a0	First operand
	b3	b2	b1	b0	Second operand
	s3	s2	s1	s0	

0	0	1	1	0
	0	0	0	1
	0	0	1	1
	0	1	0	0

From the very beginning I can “look ahead” into the value of carries

Carry-lookahead adders

- Adder with propagate (P) and generate (G) outputs:

$$C_{i+1} = \underbrace{A_i B_i}_{\text{Generate}} + C_i \underbrace{(A_i \text{ xor } B_i)}_{\text{Propagate}}$$

$$C_{i+1} = G_i + C_i P_i$$

The carry at some level is equal to 1 if either the generate signal is equal to one or if the propagate and the previous carry are both 1

Carry-lookahead adders

$$C_{i+1} = A_i B_i + C_i (A_i \oplus B_i)$$

0	0	1	1	0	Carry
	0	0	0	1	a
	0	0	1	1	b
	0	1	0	0	Sum

Carry-lookahead adders

Example: 4-bit CLA adder

$$c_1 = G_0 + P_0 c_0$$

$$c_2 = G_1 + P_1 c_1$$

$$c_3 = G_2 + P_2 c_2$$

$$c_4 = G_3 + P_3 c_3$$

$$G_i = a_i b_i \quad \text{generate}$$

$$P_i = a_i \text{ xor } b_i \quad \text{propagate}$$

All “G” and “P” are immediately available, but “c” are not (except the c_0).

So you need to make substitutions:

$$c_1 = G_0 + P_0 c_0$$

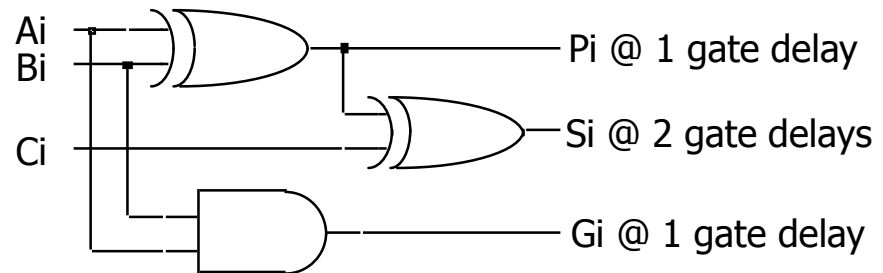
$$c_2 = G_1 + P_1 (G_0 + P_0 c_0) = G_1 + P_1 G_0 + P_1 P_0 c_0$$

$$c_3 = G_2 + P_2 c_2 = (\text{derive at home})$$

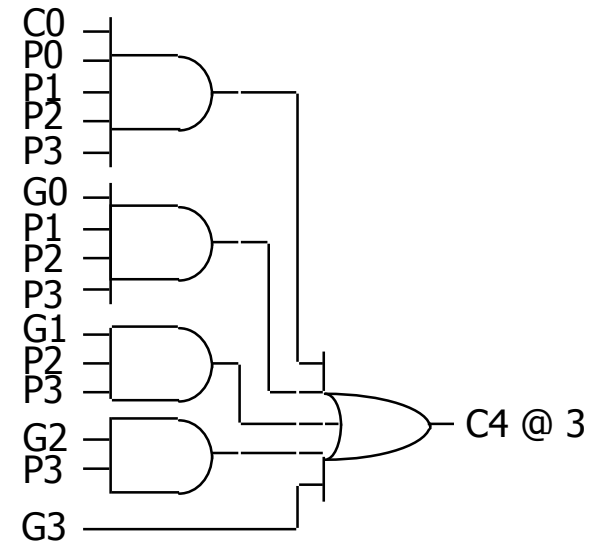
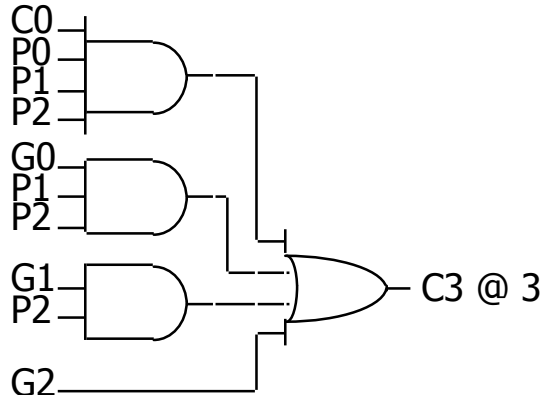
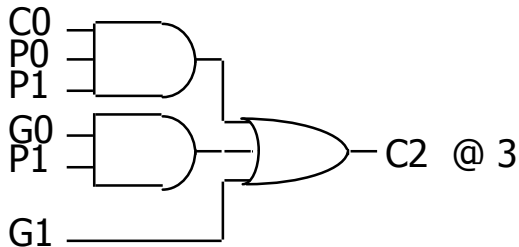
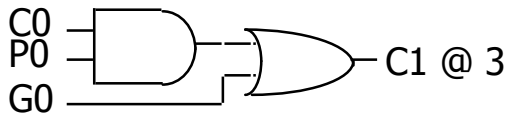
$$c_4 = G_3 + P_3 c_3 = (\text{derive at home})$$

Carry-lookahead adders

Propagate/Generate circuit
(one per each input bit)



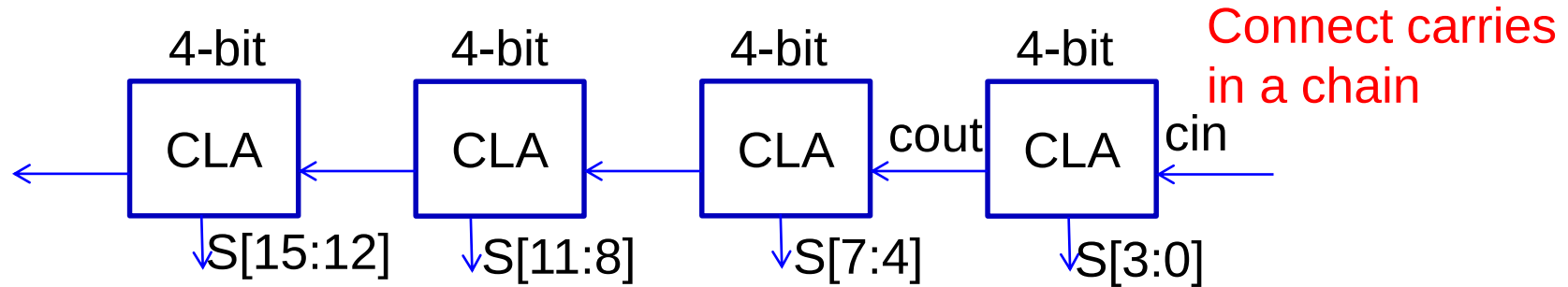
Carry circuits (implement the equations derived in the previous slide)



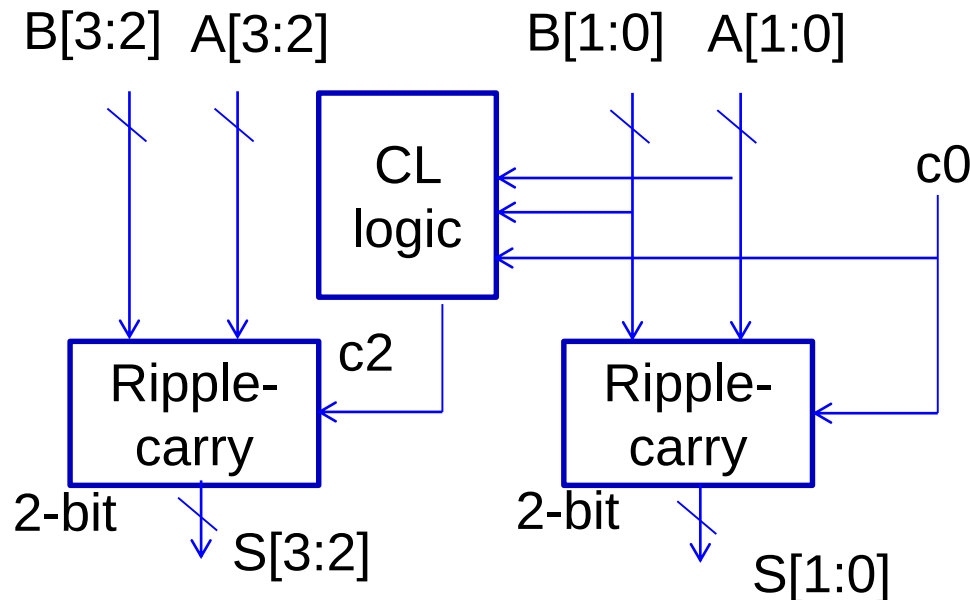
Note: this approach of “looking ahead” for building multi-bit operations is not limited to adders!

Combining Adders

Example: connect CLAs in a ripple-carry style (16-bit adder)



Example: connect ripple-carry in CLA style (4-bit adder)



$$c2 = G1 + P1G0 + P1P0c0$$

Where:

$$G0 = A0B0$$

$$G1 = A1B1$$

$$P0 = A0 \text{ xor } B0$$

$$P1 = A1 \text{ xor } B1$$



Subtractors

2s complement

- If N is a positive number, then the negative of N (its 2s complement or N^*) is bit-wise complement plus 1
- The most significant bit represent the sign: 0 for positive and 1 for negative
- N bit can represent $[0 \ 2^N - 1]$ integer positive numbers
- In 2s complement, you can represent the interval $[-(2^{N-1}) \ (2^{N-1} - 1)]$

2s Complement: Examples

A 8-bit example

0 0 0 0 1 0 1 1

(positive) = 11

1 1 1 1 0 1 0 0 (complement)

1 1 1 1 0 1 0 1 (add 1)

A 5-bit example

1 0 0 1 0

(negative) = -14

0 1 1 0 1 (complement)

0 1 1 1 0 (add 1)

Subtraction

If you are using 4 bit numbers, what is the result of the following equation in 2s complement: $y = 4 - 7$

- A. 1011
- B. 0011
- C. 1101
- D. 1100
- E. None of the above

Detecting Overflow: Method 1

- Assuming 4-bit two's complement numbers, one can detect overflow by detecting when the two numbers' sign bits are the same but are different from the result's sign bit
 - If the two numbers' sign bits are different, overflow is impossible
 - Adding a positive and negative can't exceed the largest magnitude positive or negative
- Simple circuit
 - $\text{overflow} = a_3'b_3's_3 + a_3b_3s_3'$

sign bits			
0	1	1	1
+0	0	0	1
1	0	0	0
overflow			
(a)			
1	1	1	1
+1	0	0	0
0	1	1	1
overflow			
(b)			
1	0	0	0
+0	1	1	1
1	1	1	1
no overflow			
(c)			

If the numbers' sign bits have the same value, which differs from the result's sign bit, overflow has occurred.

Detecting Overflow: Method 2

- Detect a difference between carry-in to sign bit and carry-out from it
- Yields a simpler circuit: $\text{overflow} = c_3 \text{ xor } c_4 = c_3 c_4' + c_3' c_4$

1 1 1	0 0 0	0 0 0
0 1 1 1	1 1 1 1	1 0 0 0
+ 0 0 0 1	+ 1 0 0 0	+ 0 1 1 1
0 1 0 0 0	1 0 1 1 1	0 1 1 1 1
overflow	overflow	no overflow
(a)	(b)	(c)

If the carry into the sign bit column differs from the carry out of that column, overflow has occurred.

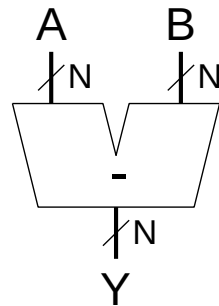
Subtractor

A subtraction between A and B is the same as the sum between the first value and the negative of the second value:

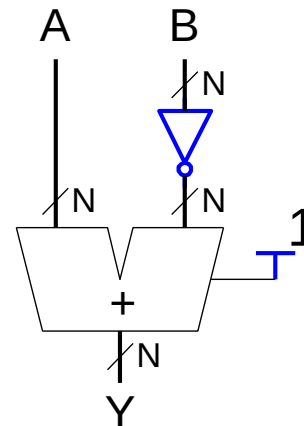
$$(A - B) = A + (-B)$$

Represent numbers in 2s complement and use a normal adder!

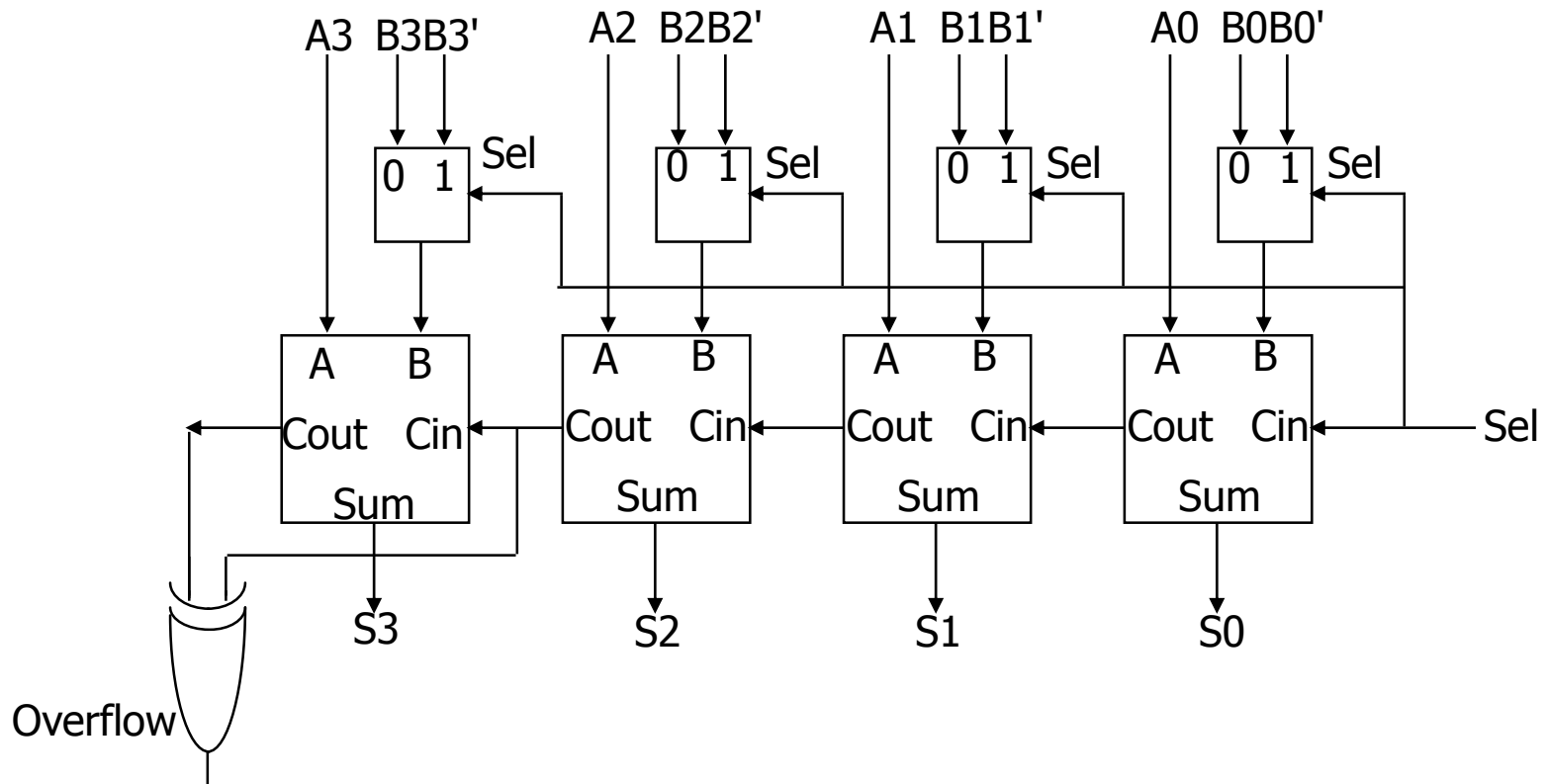
Symbol



Implementation



Adder/subtractor



In this schematic addition occurs when Sel/ signal is:

- A. True
- B. False



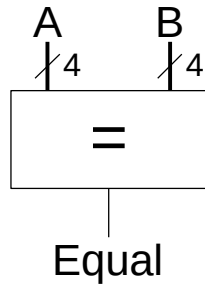
More ALU Components

Comparator: Equality

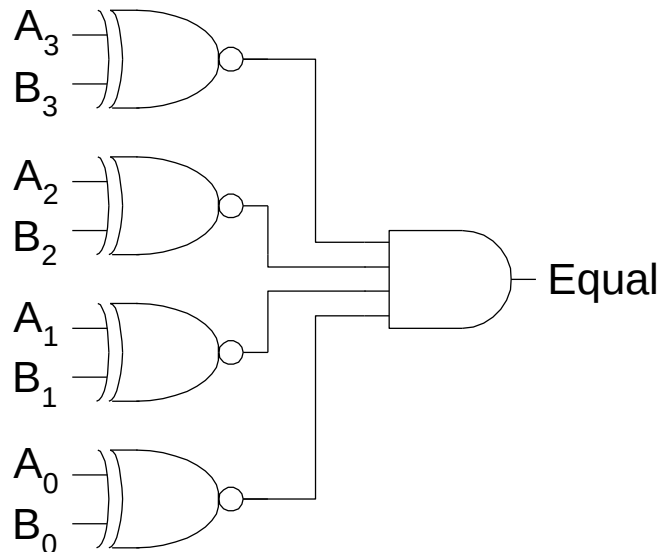
Two numbers are equal if each digit at each position is equal (this is true for any base: decimal, binary, etc).

The bit-to-bit equality can be evaluated with the XNOR gate.

Symbol

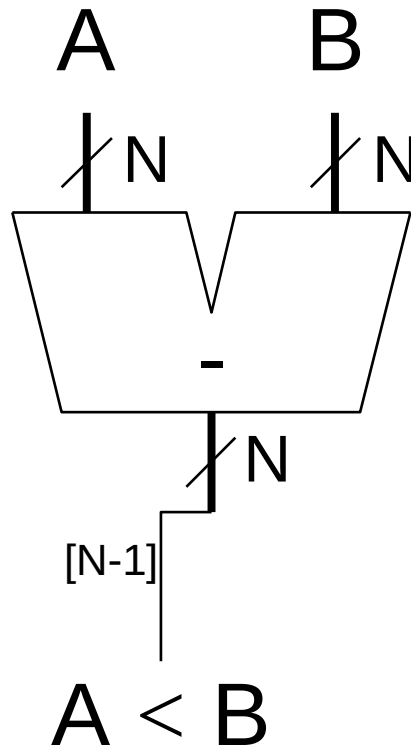


Implementation



Comparator: Less Than

- If a number A is less than B and you consider the difference $A - B$, this is:
 - negative.
- So comparing numbers is equivalent to check the sign of the difference. In 2s complement representation, the sign of the result corresponds to:
 - the most significant bit



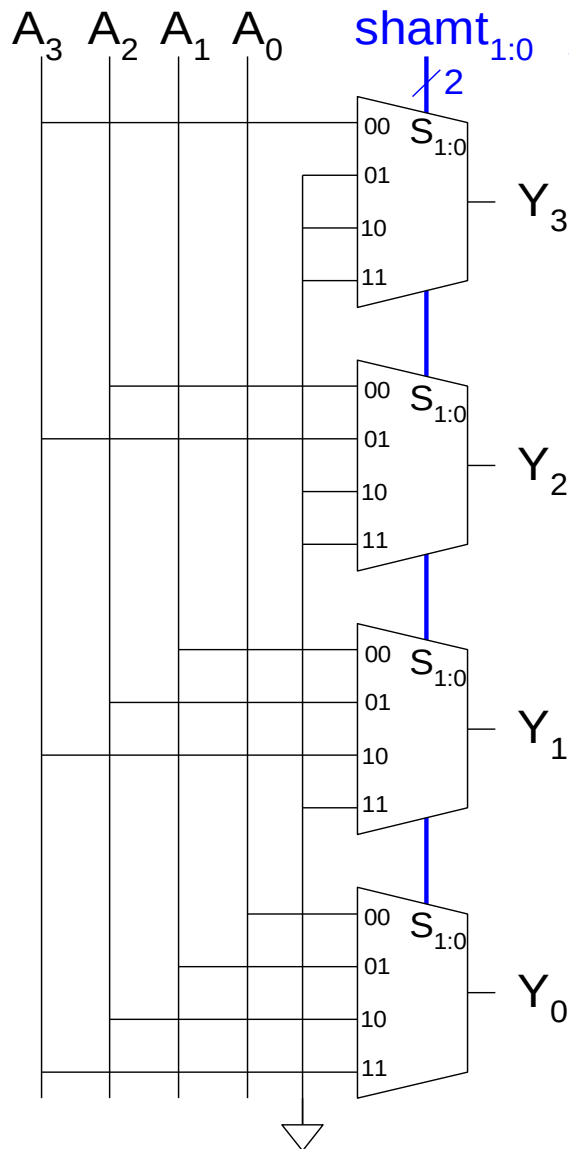
Shifters

- **Logical shifter:** shifts value to left or right and fills empty spaces with 0's
 - Ex: 11001 >> 2 = 00110
 - Ex: 11001 << 2 = 00100
- **Arithmetic shifter:** same as logical shifter, but on right shift, fills empty spaces with the old most significant bit
 - Ex: 11001 >>> 2 = 11110
 - Ex: 11001 <<< 2 = 00100

Useful for 2-complement numbers
- **Rotator:** rotates bits in a circle, such that bits shifted off one end are shifted into the other end
 - Ex: 11001 ROR 2 = 01110
 - Ex: 11001 ROL 2 = 00111

https://en.wikipedia.org/wiki/Circular_shift

General Shifter Design



Based on the value of the selection input (shamt = shift amount)

The “chain” of multiplexers determines how many bits to shift

Example: if $S = 01$ then
 $Y_3 = 0$
 $Y_2 = A_3$
 $Y_1 = A_2$
 $Y_0 = A_1$

Multiplication of positive binary numbers

- Generalized representation of multiplication by hand

Example: in decimal,

$$32 * 4 = (30+2)*4 = 30*4 + 2*4$$

Basically: sum up the partial products (pp)

The binary multiplier is based on the same idea:

				a3	a2	a1	a0	
				x b3	b2	b1	b0	

				b0a3	b0a2	b0a1	b0a0	(pp1)
			b1a3	b1a2	b1a1	b1a0	0	(pp2)
		b2a3	b2a2	b2a1	b2a0	0	0	(pp3)
+	b3a3	b3a2	b3a1	b3a0	0	0	0	(pp4)

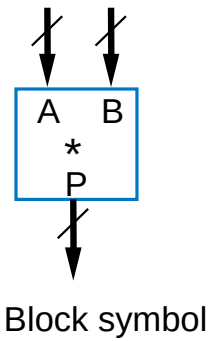
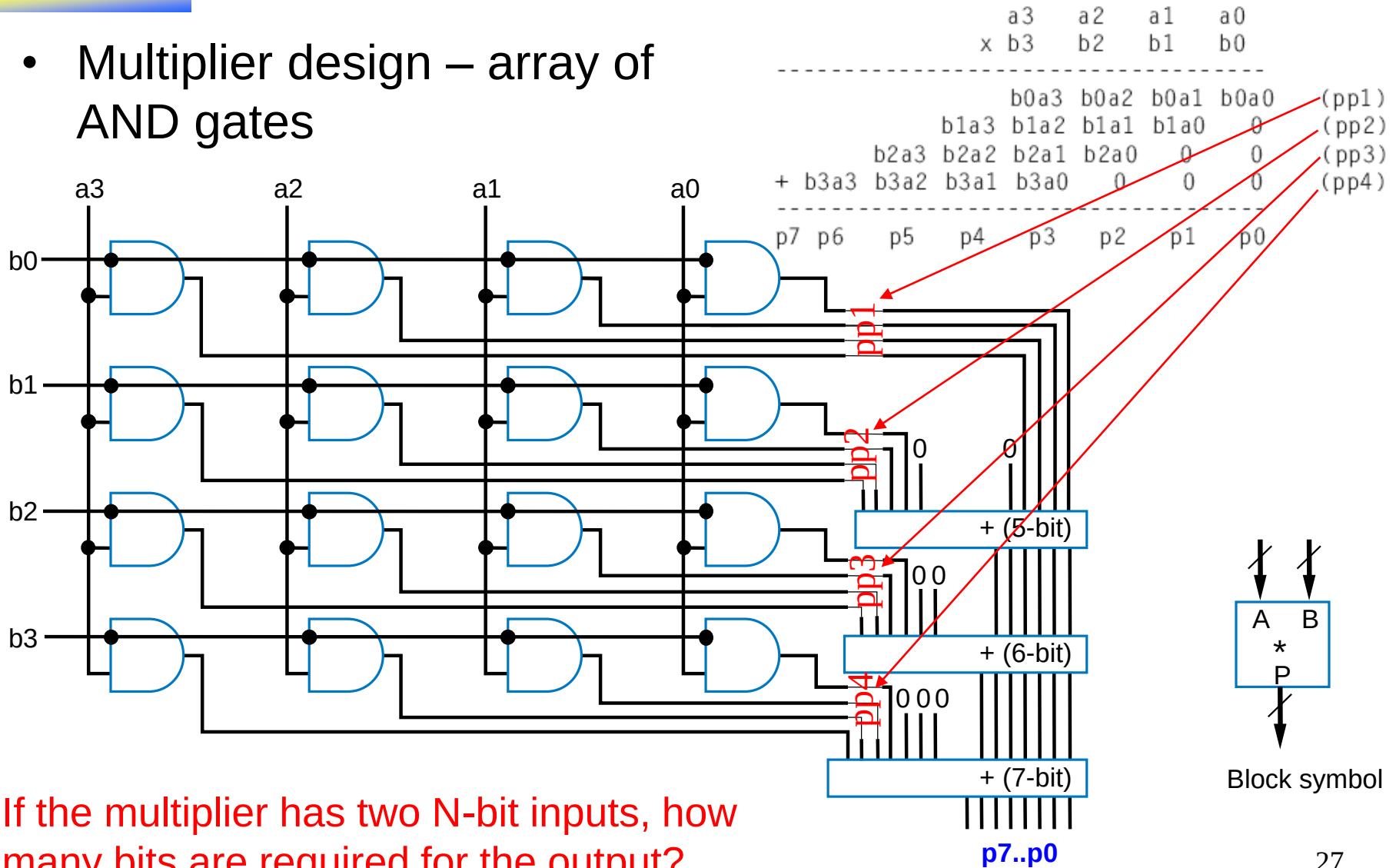
p7	p6	p5	p4	p3	p2	p1	p0	

For demo see:

<http://courses.cs.vt.edu/~cs1104/BuildingBlocks/multiply.010.html>

Multiplier – Array Style

- Multiplier design – array of AND gates



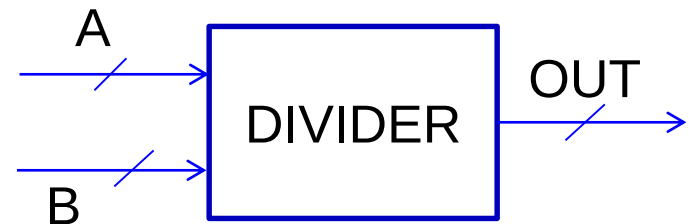
Division of positive binary numbers

- Repeated subtraction
 - Set quotient to 0
 - Repeat while dividend $\geq$ divisor
 - Subtract divisor from dividend
 - Add 1 to quotient
 - When dividend $<$ divisor:
 - Remainder = dividend
 - Quotient is correct

Example:

- Dividend: 101; Divisor: 10

Dividend		Quotient
101	-	0
10		1
11	-	1
10		1
1		10



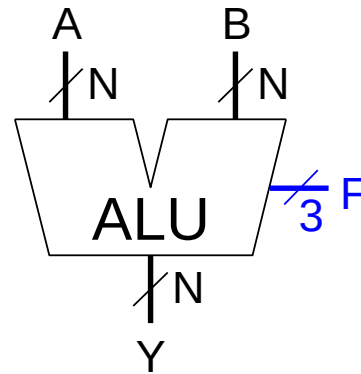
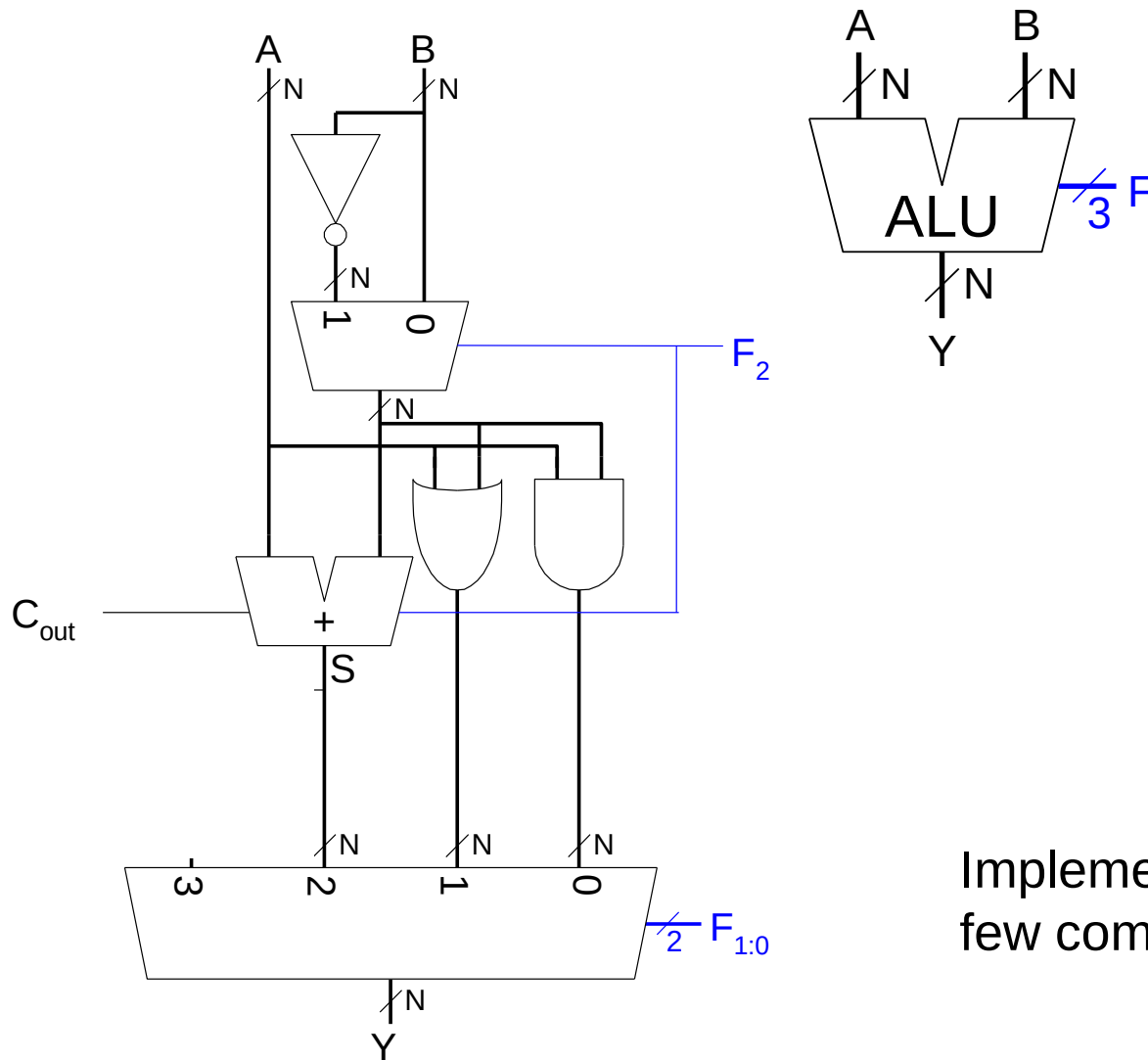
For demo see:

<http://courses.cs.vt.edu/~cs1104/BuildingBlocks/Binary.Divide.html>



ALU: Arithmetic Logic Unit

Arithmetic Logic Unit – Example



$F_{2:0}$	Function
000	$A \& B$
001	$A \mid B$
010	$A + B$
011	Not used
100	$A \& \sim B$
101	$A \mid \sim B$
110	$A - B$
111	Not used

Implement the ALU using as few components as possible

Summary of what we have seen so far

- Transistors
- Boolean algebra
- Basic gates
- Logic functions and truth tables
- Canonical forms (SOP and POS)
- Two-level logic minimization
- Kmaps
- Multiplexers (behavior and how to implement logic functions with them)
- Decoders (behavior and how to implement logic functions with them)

Today:

- Adders, subtractors, and other ALU components
- SO FAR: only COMBINATIONAL logic (i.e. no “memory” elements)



CSE140: Components and Design Techniques for Digital Systems

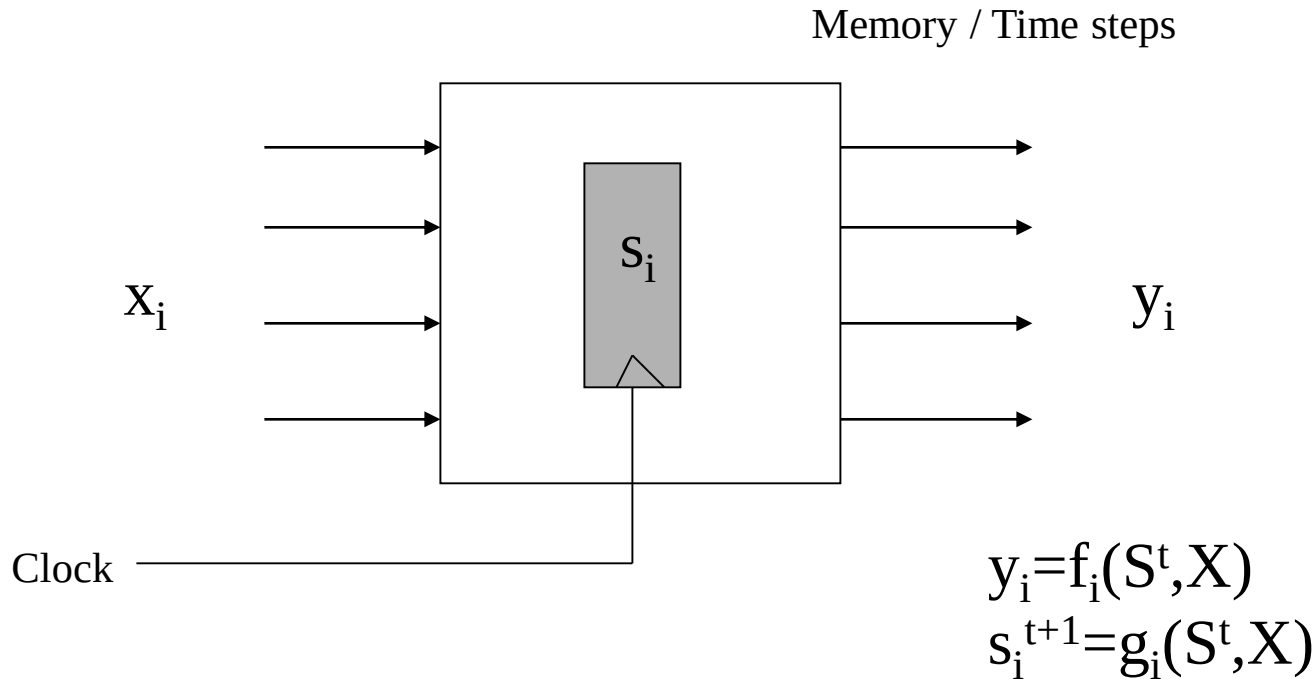
Sequential Circuit Introduction Latches and Flip-Flops

Tajana Simunic Rosing

What is a sequential circuit?

A circuit whose output depends on current inputs and past outputs

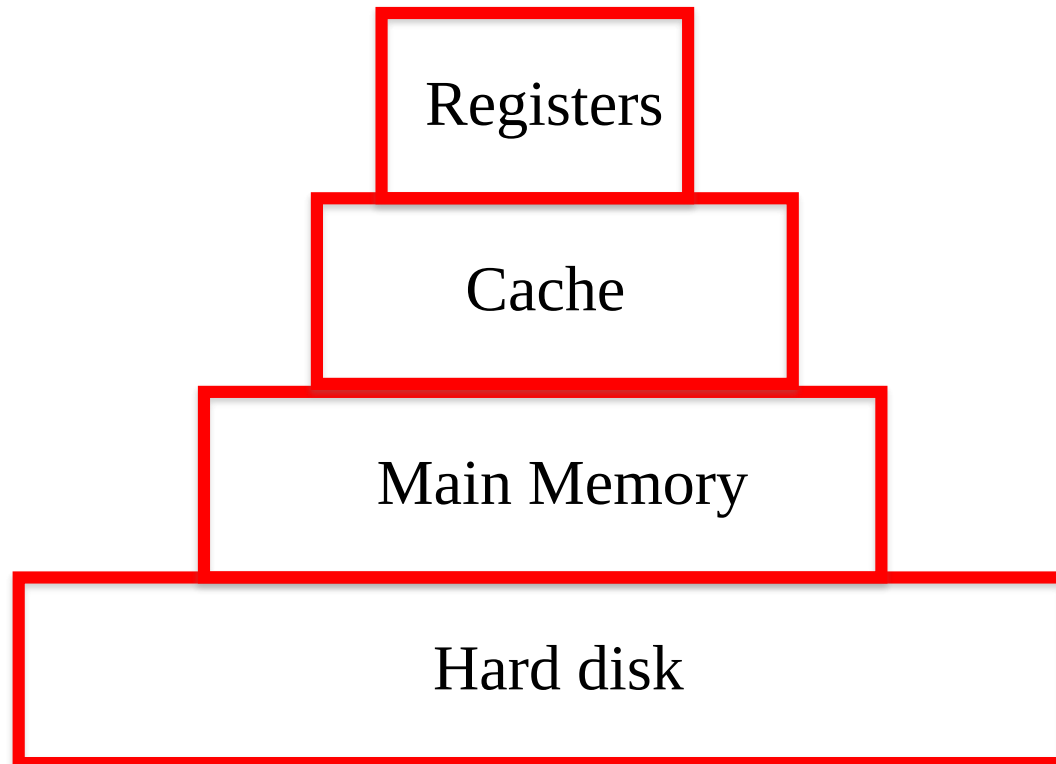
A circuit with **memory**



Why do we need circuits with 'memory'?

- Circuits with memory can be used to store data
- Systems have circuits that run a sequence of tasks

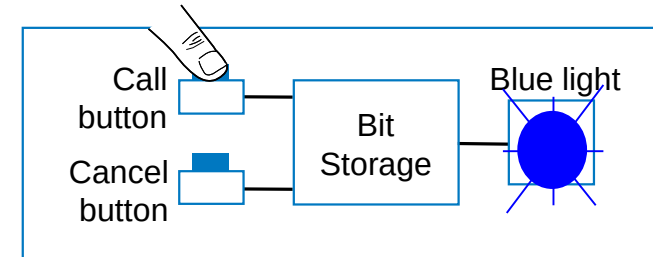
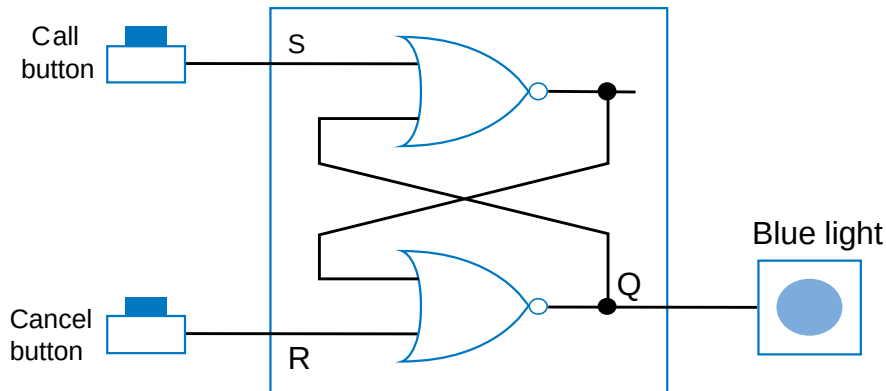
Memory Hierarchy



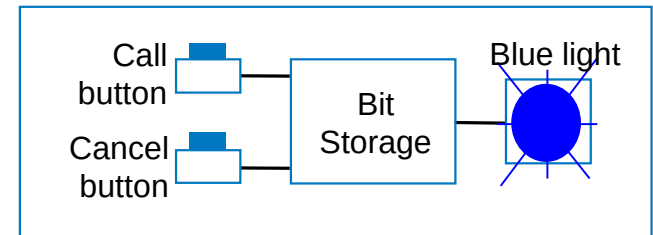
Flight attendant call button

- Flight attendant call button
 - Press call: light turns on
 - Stays on** after button released
 - Press cancel: light turns off
 - Logic circuit to implement this?

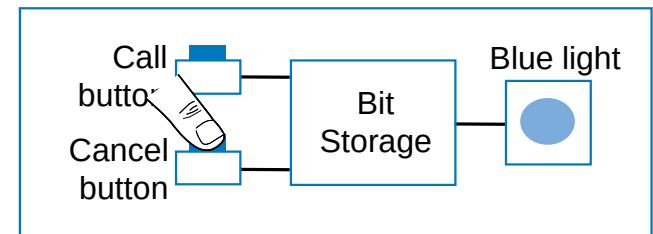
- SR latch implementation
 - Call=1 : sets Q to 1 and keeps it at 1
 - Cancel=1 : resets Q to 0



1. Call button pressed – light turns on



2. Call button released – light **stays on**



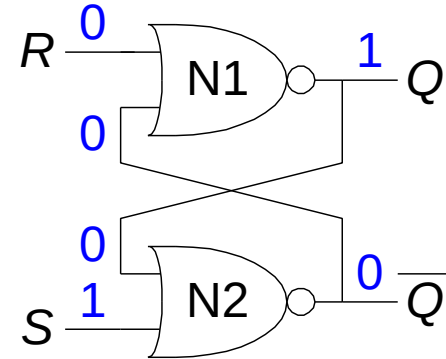
3. Cancel button pressed – light turns off

a

SR Latch Analysis

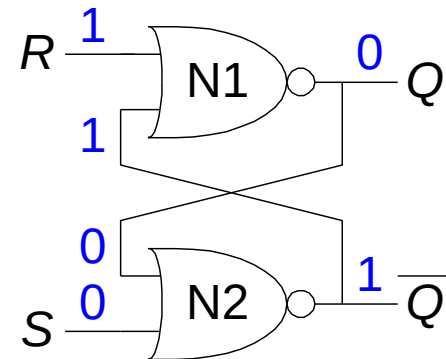
– **$S = 1, R = 0$:**

then $Q = 1$ and $\overline{Q} = 0$



– **$S = 0, R = 1$:**

then $\overline{Q} = 1$ and $Q = 0$

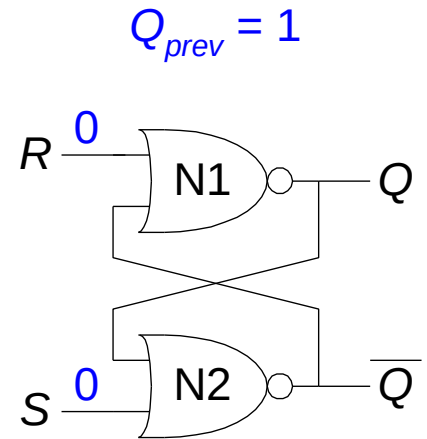
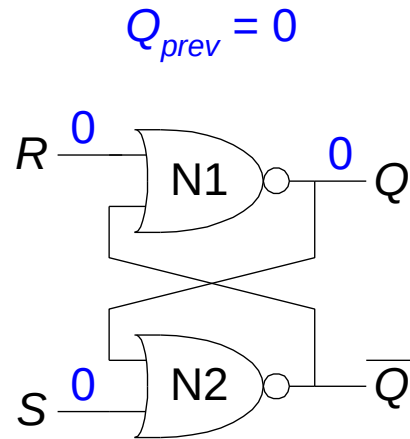


SR Latch Analysis

– $S = 0, R = 0$:

then $Q = Q_{prev}$

– **Memory!**

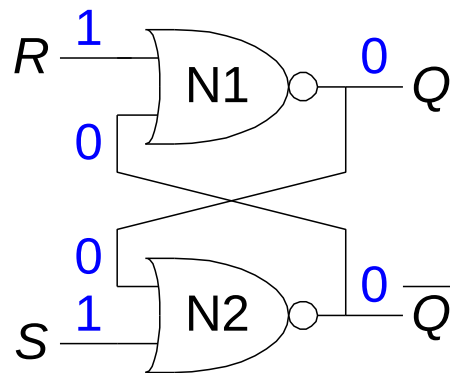


– $S = 1, R = 1$:

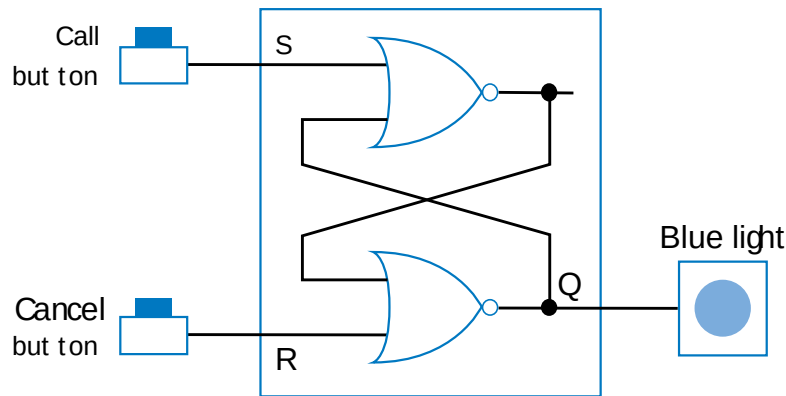
then $Q = 0, \bar{Q} = 0$

– **Invalid State**

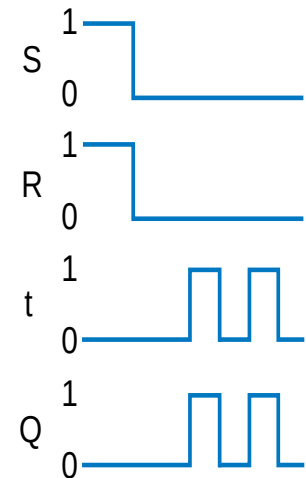
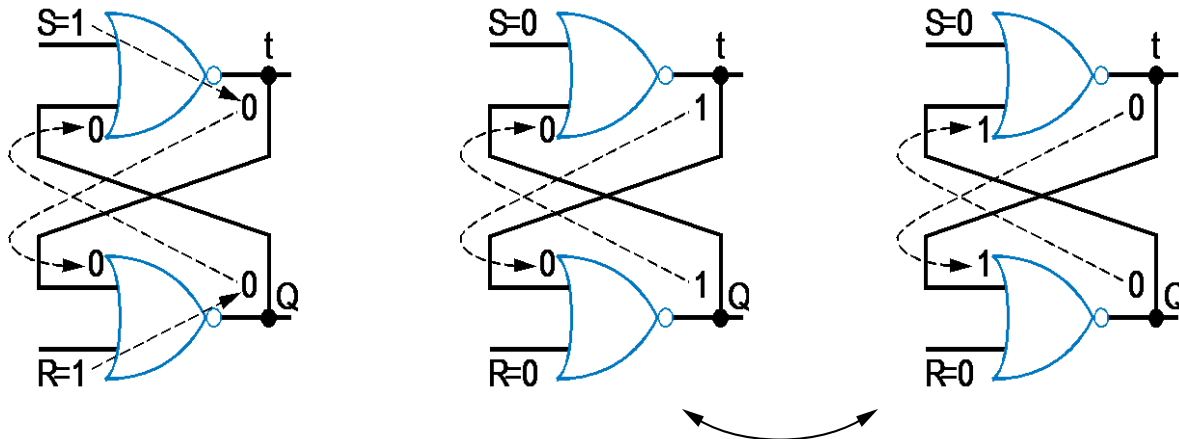
$\bar{Q} \neq \text{NOT } Q$



What if a kid presses both call and cancel & then releases them?



S	R	Q
0	0	hold
0	1	0
1	0	1
1	1	not allowed

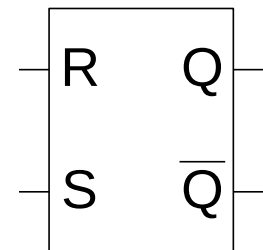


- If $S=1$ and $R=1$ at the same time and then released, $Q=?$
 - Can also occur also due to different delays of different paths
 - Q may oscillate and eventually settle to 1 or 0 due to diff. path delay

SR Latch Symbol

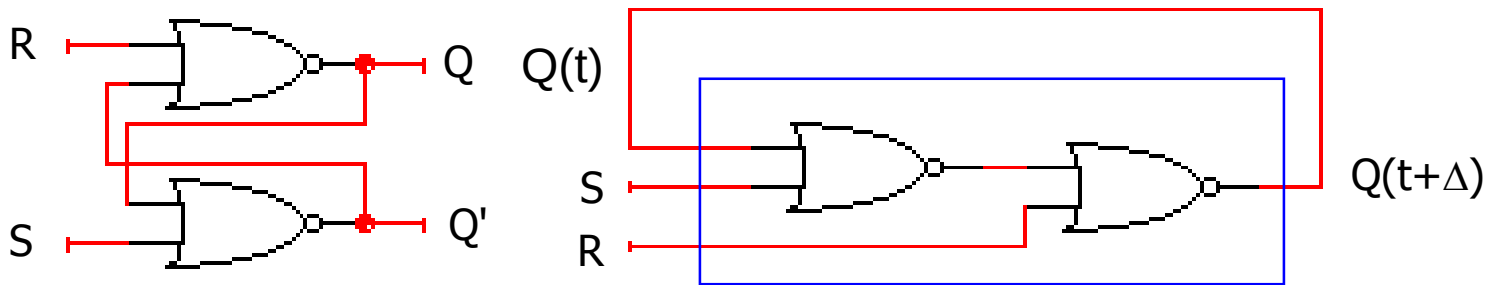
- SR stands for Set/Reset Latch
 - Stores one bit of state (Q)
- Control what value is being stored with S , R inputs
 - **Set:** Make the output 1
($S = 1, R = 0, Q = 1$)
 - **Reset:** Make the output 0
($S = 0, R = 1, Q = 0$)
 - **Hold:** Keep data stored
($S = 0, R = 0, Q = Q_{previous}$)

SR Latch
Symbol

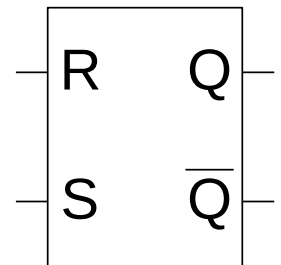


SR Latch Characteristic Equation

To analyze, break the feedback path



SR Latch
Symbol

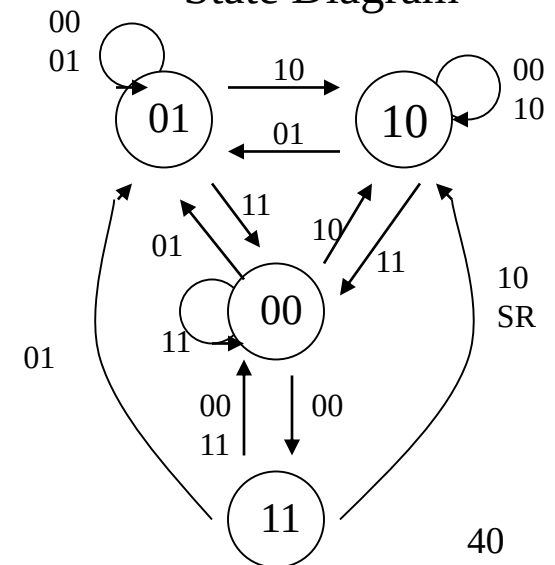


S	R	Q(t)	Q(t+Δ)	
0	0	0	0	hold
0	0	1	1	
0	1	0	0	reset
0	1	1	0	
1	0	0	1	set
1	0	1	1	
1	1	0	X	not allowed
1	1	1	X	

		S	
	0	0	1
	X	X	1
Q(t)	1	0	1
	R		

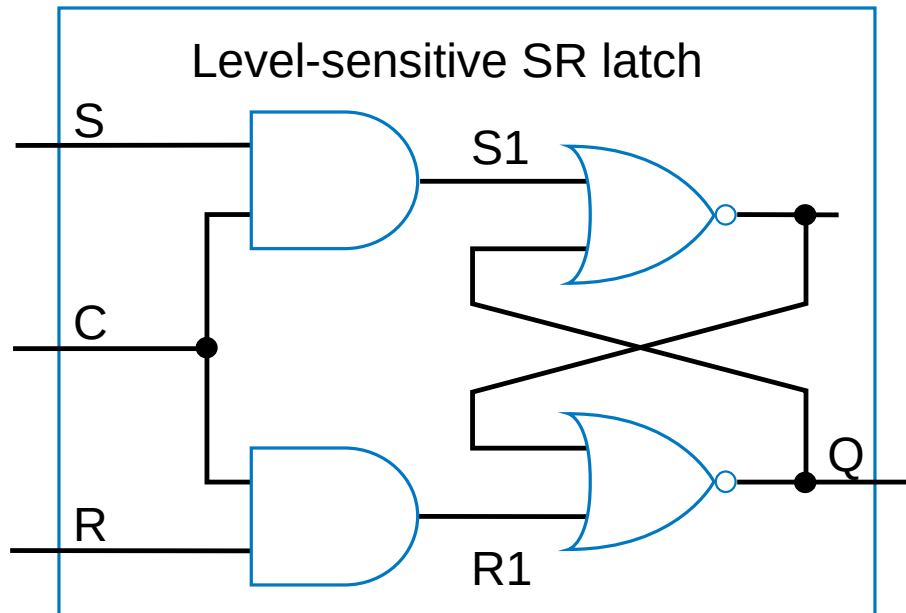
characteristic equation
 $Q(t+\Delta) = S + R' Q(t)$

State Diagram



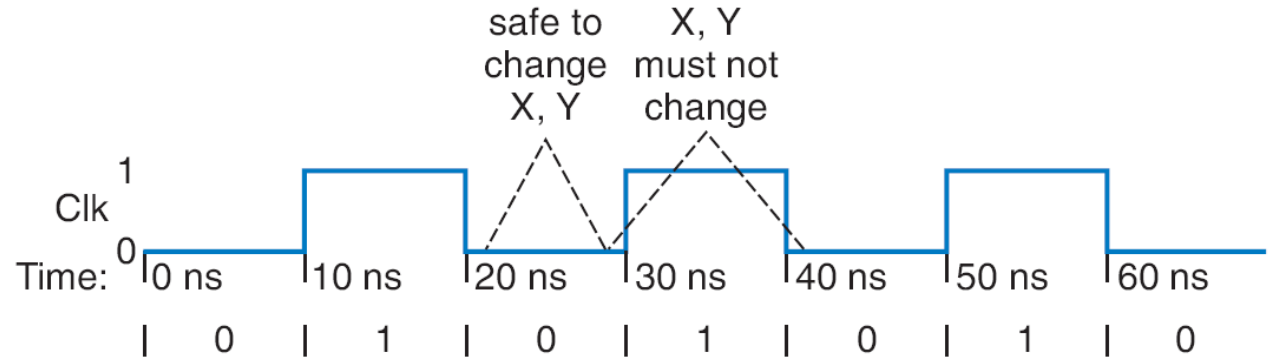
Avoiding S=R=1 Part 1: Level-Sensitive SR Latch

- Add input “C”
 - Change C to 1 only after S and R are stable
 - C is usually a clock (CLK)



Clocks

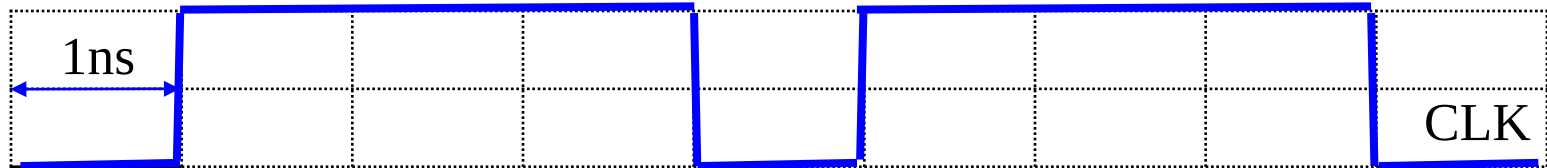
Freq	Period
100 GHz	0.01 ns
10 GHz	0.1 ns
1 GHz	1 ns
100 MHz	10 ns
10 MHz	100 ns



- **Clock** -- Pulsing signal for enabling latches; ticks like a clock
- **Synchronous** circuit: sequential circuit with a clock
- **Clock period**: time between pulse starts
 - Above signal: period = 20 ns
- **Clock cycle**: one such time interval
 - Above signal shows 3.5 clock cycles
- **Clock duty cycle**: time clock is high
 - 50% in this case
- **Clock frequency**: $1/\text{period}$
 - Above : $\text{freq} = 1 / 20\text{ns} = 50\text{MHz}$;

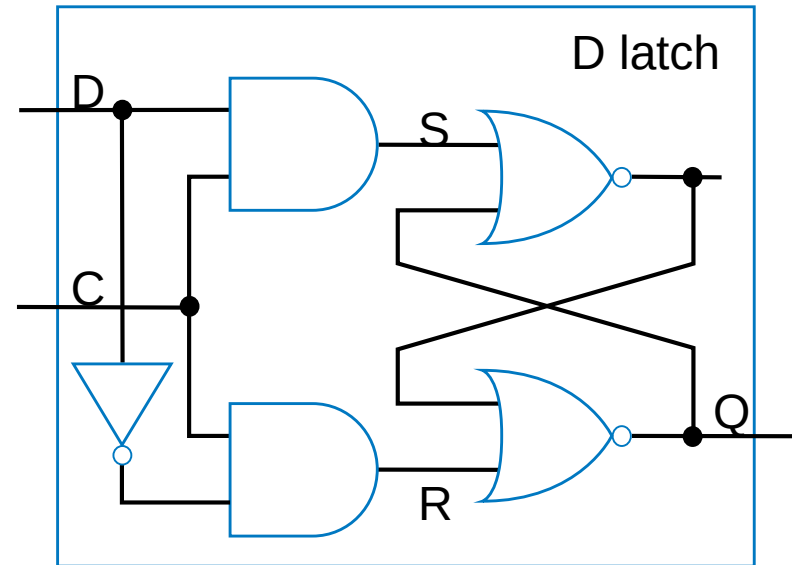
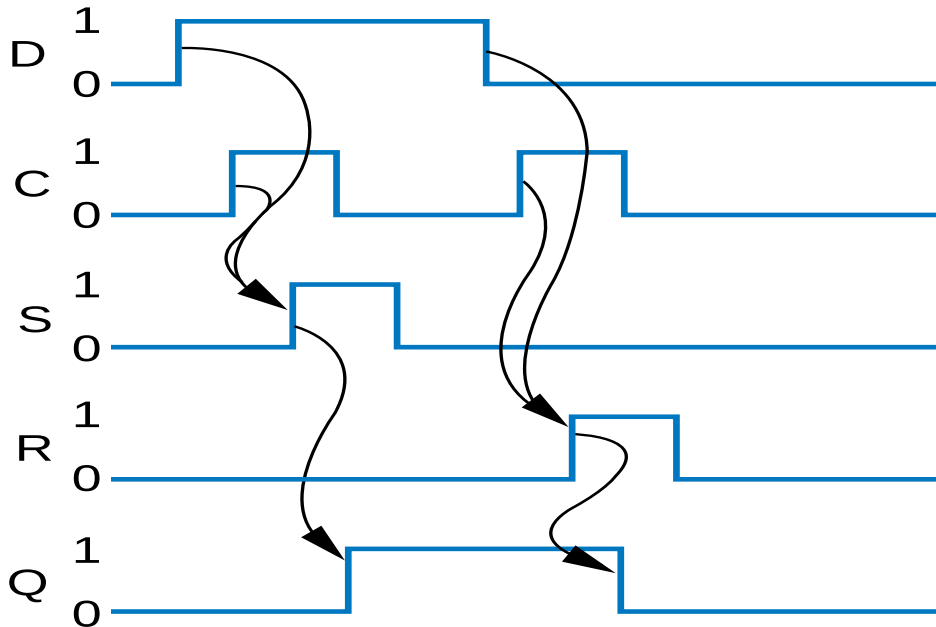
Clock question

The clock shown in the waveform below has:



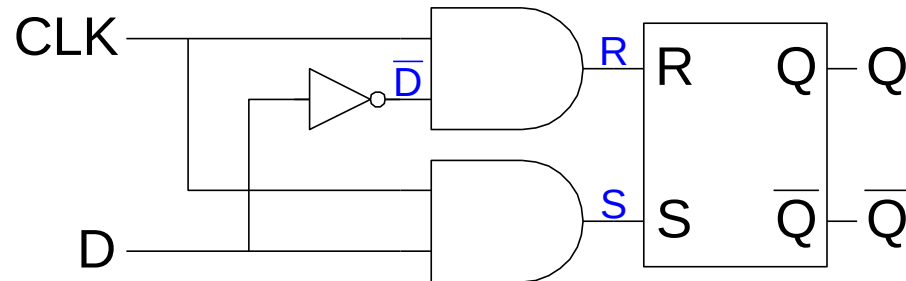
- A. Clock period of 4ns with 250MHz frequency
- B. Clock duty cycle 75%
- C. Clock period of 1ns with 1GHz frequency
- D. A. & B.
- E. None of the above

Avoiding $S=R=1$ Part 2: Level-Sensitive D Latch



- SR latch requires careful design so $SR=11$ never occurs
- D latch helps by inserting the inverter between S & R inputs
 - Inserted inverter ensures R is always the opposite of S when $C=1$

D Latch Truth Table

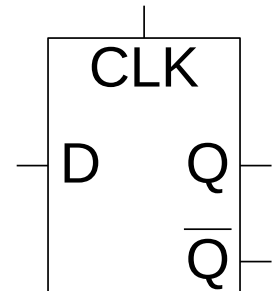


CLK	D	$\overline{D}$	S	R	Q	$\overline{Q}$
0	X	$\overline{X}$	0	0	Q_{prev}	$\overline{Q}_{prev}$
1	0	1	0	1	0	1
1	1	0	1	0	1	0

D Latch Summary

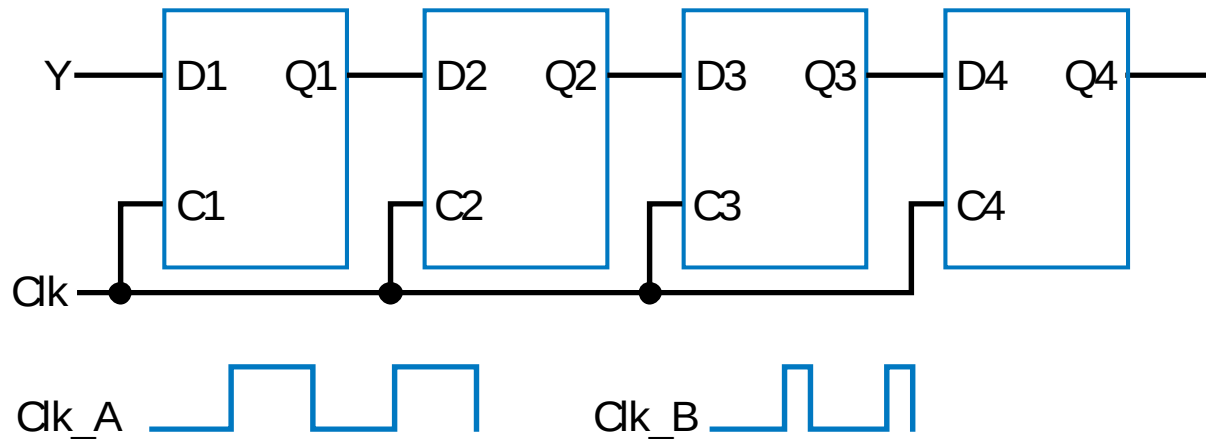
- Two inputs: CLK , D
 - CLK : controls *when* the output changes
 - D (the data input): controls *what* the output changes to
- Function
 - When $CLK = 1$,
 D passes through to Q (*transparent*)
 - When $CLK = 0$,
 Q holds its previous value (*opaque*)
- (Mostly) avoids invalid case $Q = Q'$

D Latch
Symbol



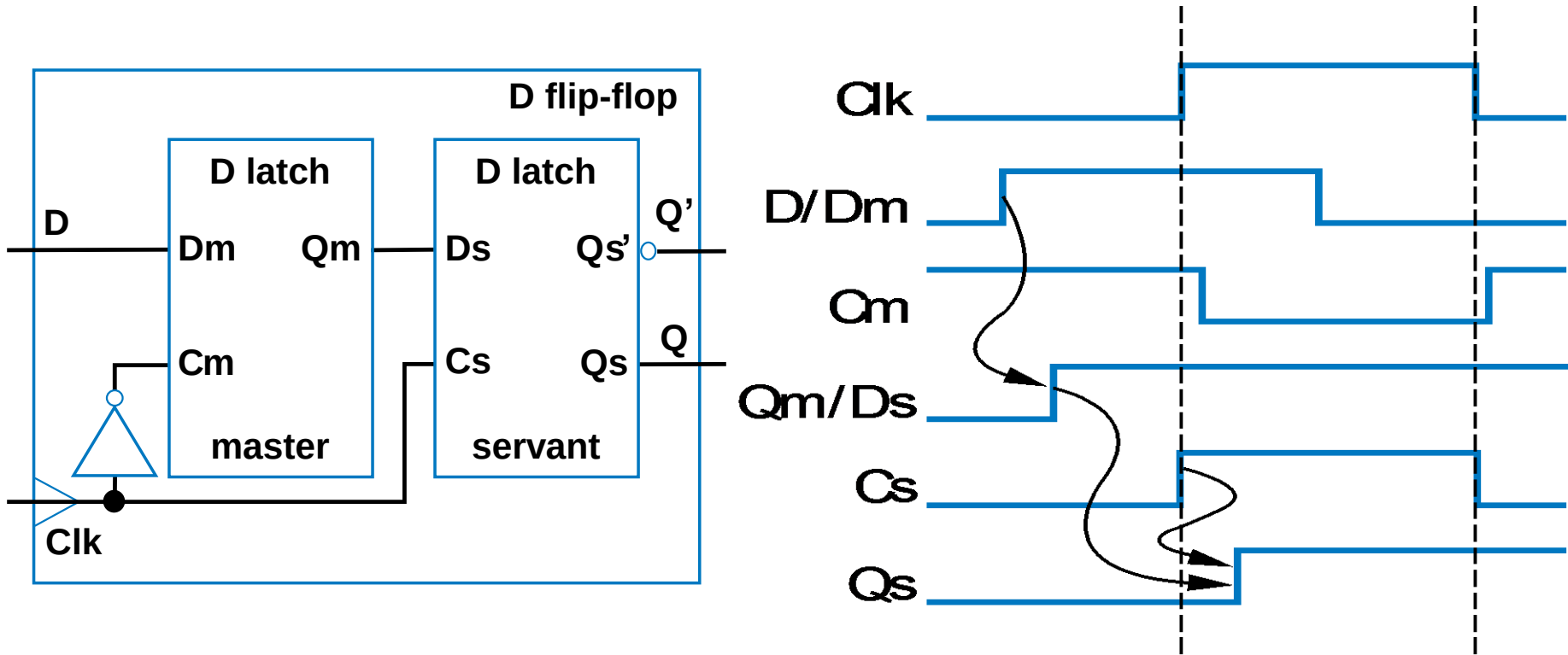
Level-Sensitive D Latches

Assume that data in all latches is initially 0. Input $Y=1$ and Clk transitions from 0- $\rightarrow$ 1. When $\text{Clk}=0$ again, the stored values in latches are:



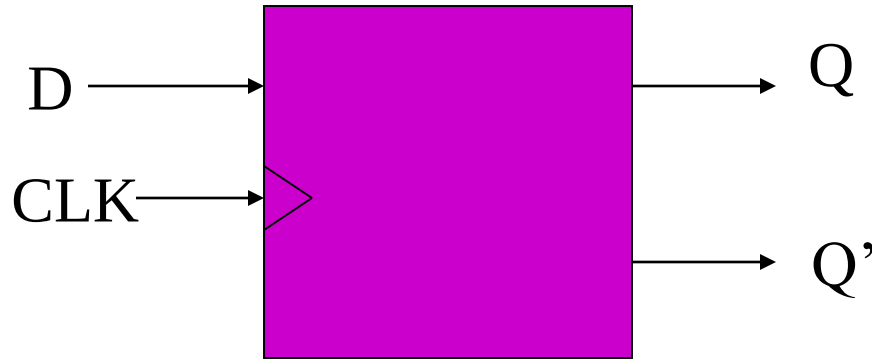
- A. $Q1=1, Q2=0, Q3=0, Q4=0$ for both clock A & B
- B. $Q1=1, Q2=1, Q3=1, Q4=1$ for clock A
 $Q1=1, Q2=0, Q3=0, Q4=0$ for clock B
- C. $Q1=1, Q2=1, Q3=1, Q4=1$ for both clocks
- D. More information is needed to determine the answer
- E. None of the above

D Flip-Flop Design & Timing Diagram



- **Flip-flop**: Bit storage that stores on the clock edge, not level
- Master-slave design: master loads when Clk=0, then slave when Clk=1

D Flip-Flop: Characteristic Equation



Id	D Q(t)		Q(t+1)
0	0	0	0
1	0	1	0
2	1	0	1
3	1	1	1

Characteristic Equation

$$Q(t+1) = D(t)$$