EECS 151/251A SP2022 Discussion 9

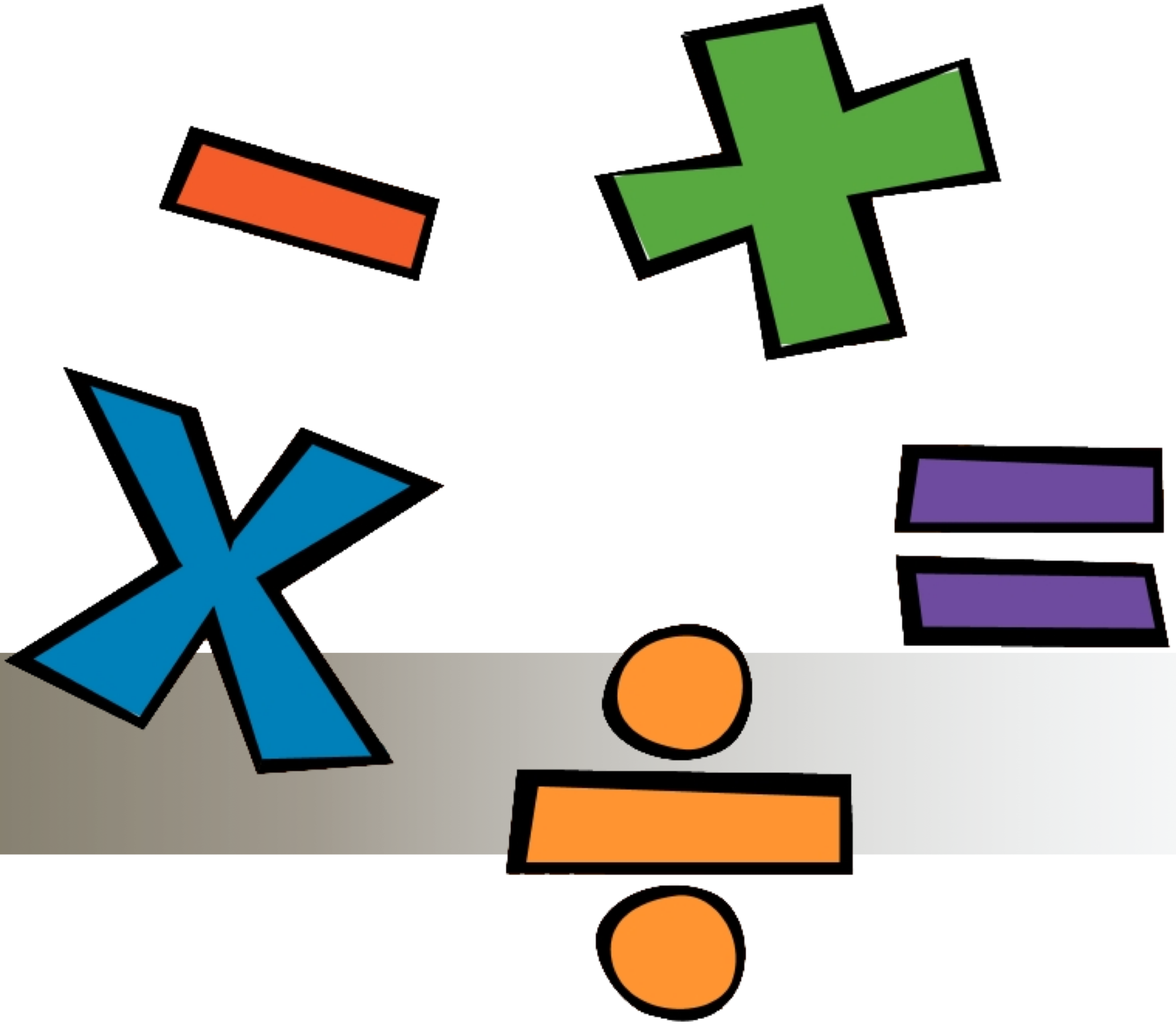
GSI: Yikuan Chen, Dima Nikiforov

Slides modified from Zhenghan Lin’s and Harrison Liew’s slides

Agenda

- Multiplier
- Latch

Multiplier



Unsigned Multiplication Example

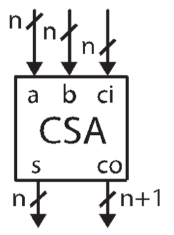
```
  4’b0011 (3)
× 4’b0110 (6)
```

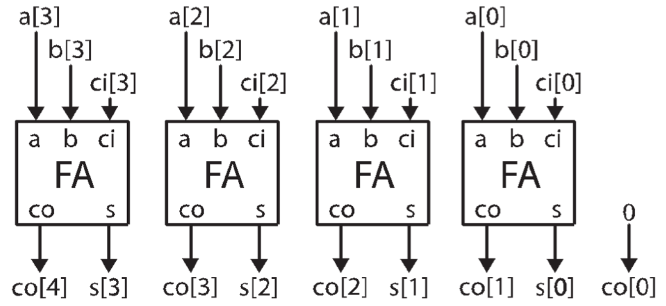
- Partial Products can be generated in parallel
- Challenge: improve the **addition** of partial products

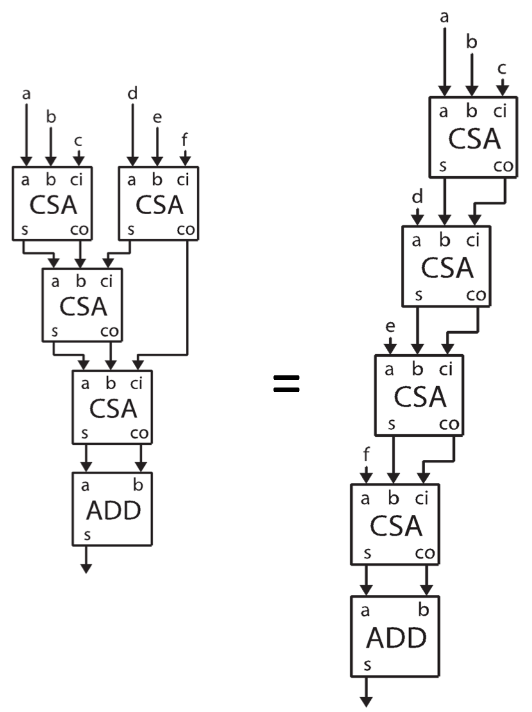
```
   4’b0011 (3)
×  4’b0110 (6)
   --------
      0000  ┐
     0011   │
    0011    ├ Partial Products
+  0000     ┘
----------
 00010010 (18)

    16  + 2
```

Carry-Save Addition

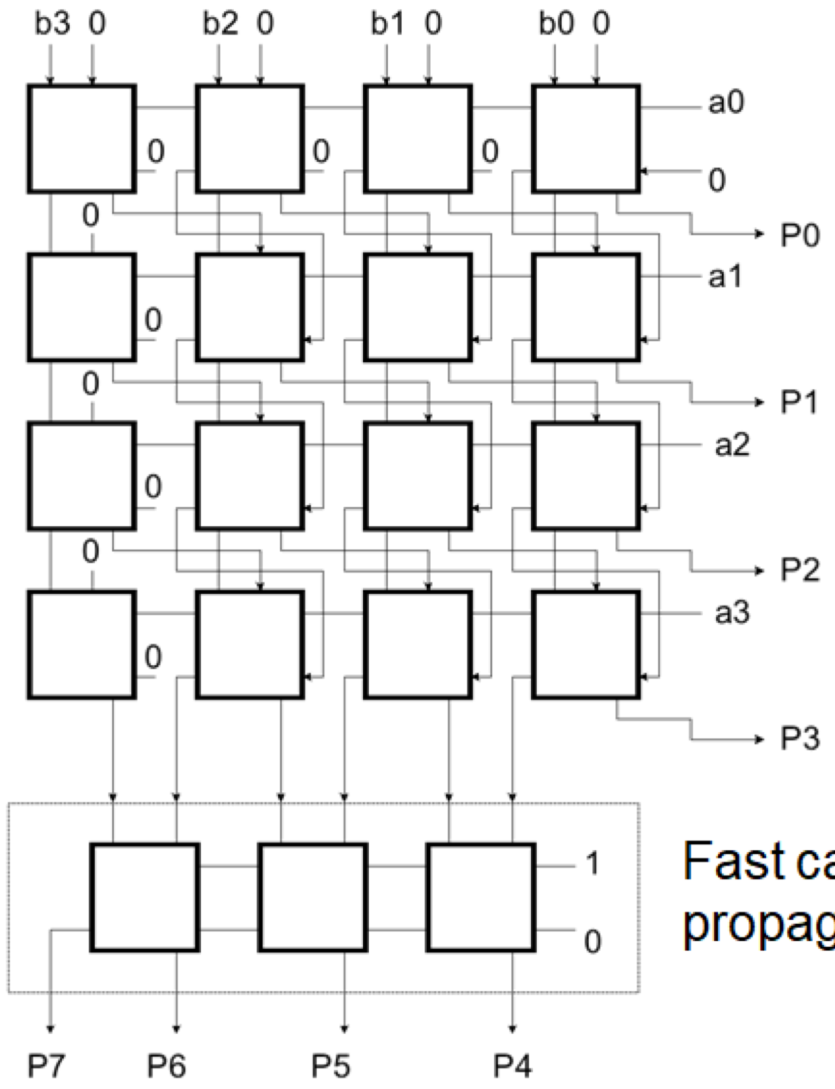
- When you want to add **more than 2 numbers**
- When we generate a carry in a given column, add it to the 2 values in the next column.
  - In turn, may generate its own new carry
- Delay adding carry bits until the end
- Basis of Unit Carry-Save Adder:
  - Takes in a, b, c_{in} (all are multi-bit)
  - Produces a sum and c_{out} (all are multi-bit)
- Benefits:
  - CSAs have no carry ripple => small & fast!
  - Only 1 standard CLA/Parallel Prefix Adder at end
  - Addition is associative => trees!



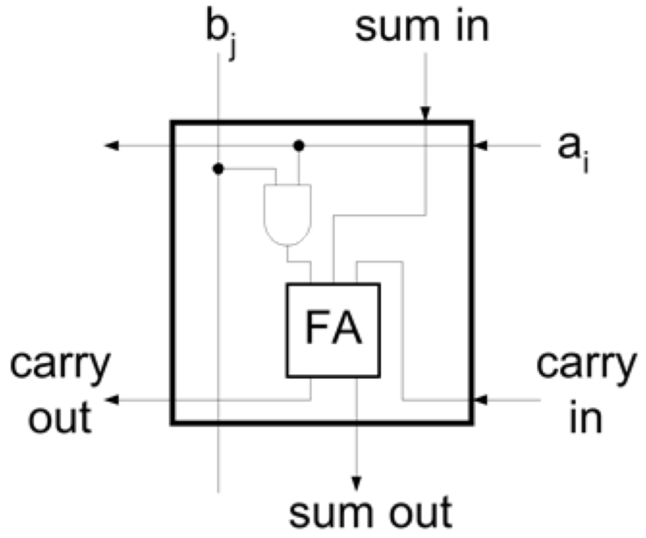




Array Multiplier w/ CSA



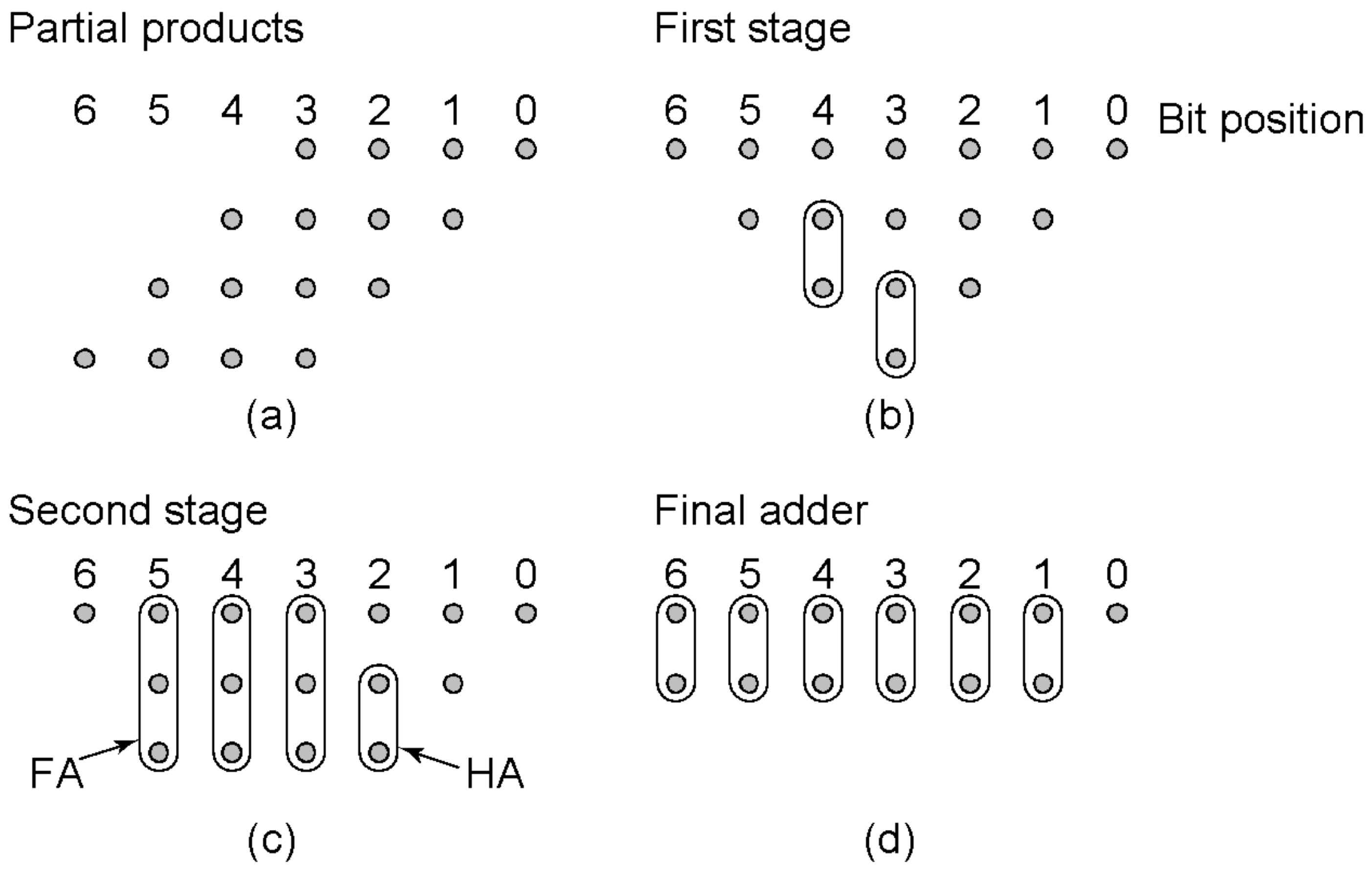
Fast carry-propagate adder



Wallace Tree Multiplier – Reduce the rows!

Method to construct Wallace Tree:

1. Draw a dot diagram where each column has as many dots as number of partial products
2. Group dots in the same column by 2 (half adder) or 3 (full adder)
3. Propagate carries and sum by adding one dot in the grouped column and one dot in the next column



Radix-4 Multiplication

- Binary multiplication -> N partial products! Can we reduce this?
  - Yes! Let’s use a larger radix (think: base)
- E.g. 2 bits at a time (radix 4) -> halve number of partial products

b Digit	Partial Product	Partial Product (Rewritten)
00	0*A	0
01	1*A	A
10	2*A	4*A - 2*A
11	3*A	4*A - A

1101 * 0110 → 1101* [01][10]

Booth Recoding –

- 4*A = A << 2
- 2*A = A << 1
- Recall: radix 4 multiplication => shift left by 2 positions for next partial product
- Therefore, any 4*A term can be handled in the next partial product!
  - Multiplier looks a 3 (rather than just 2) bits
  - Extra bit is MSB of the previous

B Digit	Partial Product	Partial Product (Rewritten)
00	0*A	0
01	1*A	A
10	2*A	4*A - 2*A
11	3*A	4*A - A

Booth Recoding

B_{i+1}	B_i	B_{i-1}	Action	Comment
0	0	0	Add 0	
0	0	1	Add A	Includes +4*A from previous radix 4 digit = +A in this position due to left shift by 2
0	1	0	Add A	
0	1	1	Add 2*A	Includes +4*A from previous round (+A in this position). *2 is implemented as a left shift by 1
1	0	0	Sub 2*A	4*A will be added in when handling next radix 4 digit. *2 is implemented as a left shift by 1
1	0	1	Sub A	4*A will be added in when handling next radix 4 digit. Includes +4*A from previous radix 4 digit (+A in this position)
1	1	0	Sub A	4*A will be added in when handling next radix 4 digit.
1	1	1	Add 0	4*A will be added in when handling next radix 4 digit. Includes +4*A from previous radix 4 digit (+A in this position)

Booth Recoding Example (Unsigned)

- 6 * 7
- $B_{-1} = 0$

```
        4’b0110   (6)
     *  4’b0111   (7)
        --------
-          0110 ( Sub A)
+       01100   (Add 2A)
+      0000     ( Add 0)
     ----------
+      11111010 ( Sub A)
+       01100   (Add 2A)
+      0000     ( Add 0)
     ----------
~~(1)~~00101010 (42)
```

B_{i+1}	B_i	B_{i-1}	Action
0	0	0	Add 0
0	0	1	Add A
0	1	0	Add A
0	1	1	Add 2*A
1	0	0	Sub 2*A
1	0	1	Sub A
1	1	0	Sub A
1	1	1	Add 0

Signed Multiplication: Baugh-Wooley

- Recall: 2's complement MSB has negative weight, meaning:

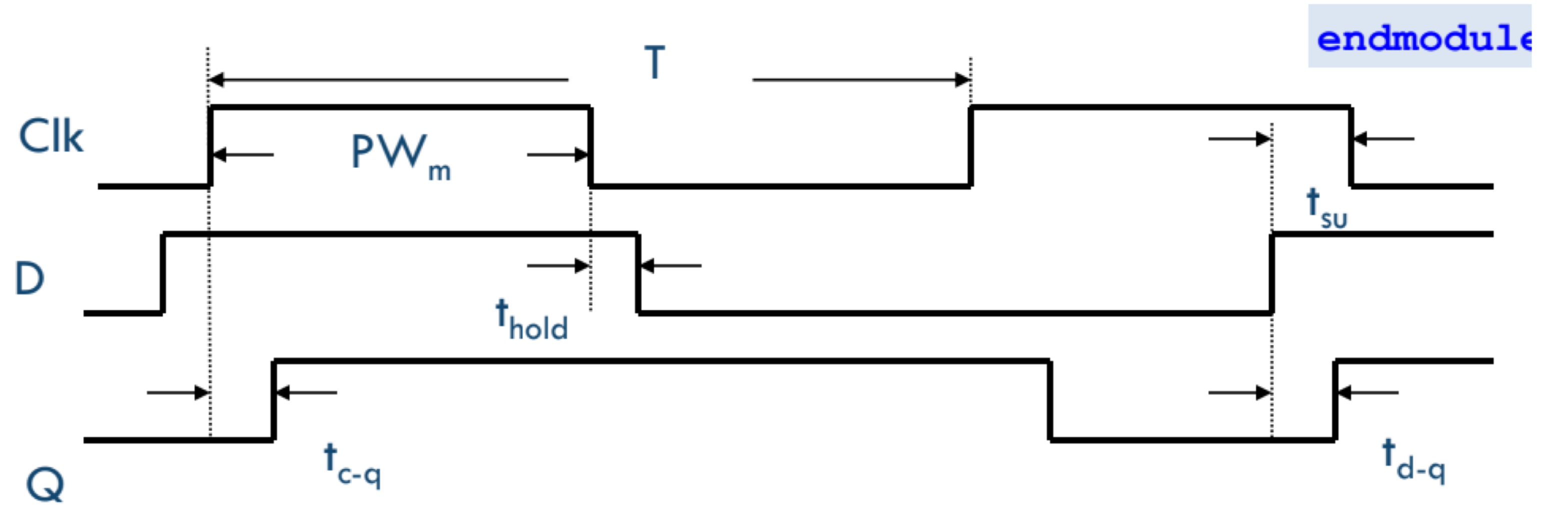
  If a[N-1:0] has its MSB=1, then rather than meaning 2^(N-1), it means -2^(N-1)
- Nominally:
  1. Subtract last partial product
  2. Sign-extend the rest of the partial products
- Recall 2's complement negation: subtract A = add (~(A) + 1)
  - Result: basically same hardware as unsigned mult.
  - Implementation: invert some bits, insert a 1 left of the first & last partial products

```
                              1    p0[3]  p0[2]  p0[1]  p0[0]
+                        ~p1[3]    p1[2]  p1[1]  p1[0]    0
+                ~p2[3]   p2[2]    p2[1]  p2[0]    0      0
+   1    ~p3[3]  ~p3[2]  ~p3[1]   ~p3[0]    0      0      0
--------------------------------------------------------------
P[7]     P[6]    P[5]     P[4]     P[3]   P[2]   P[1]   P[0]
```

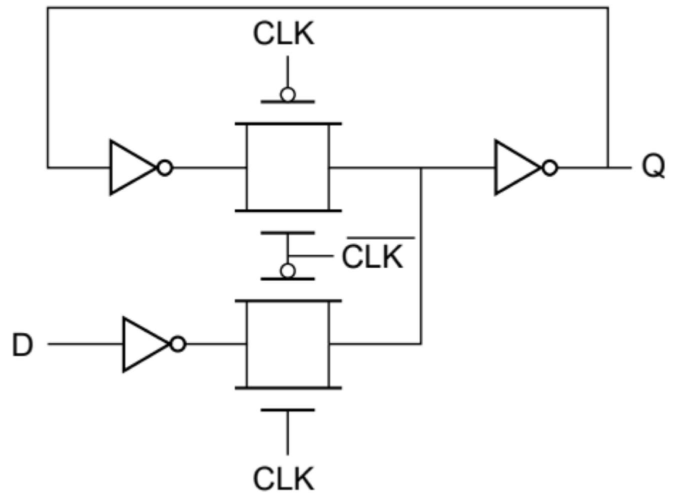
Latch

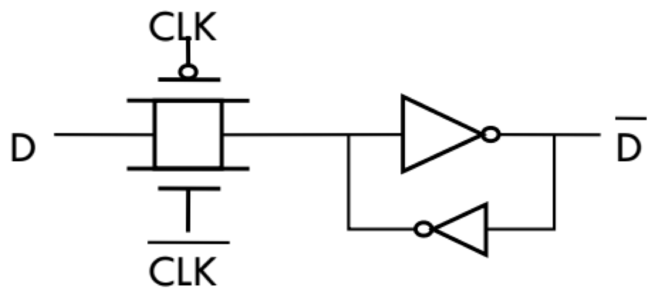
Latch Timing

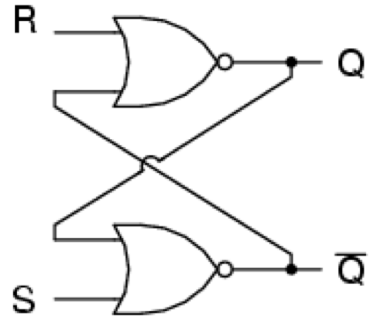
- A positive latch is **transparent** (q = d) when the **clock is high** and **opaque** (q = d, sampled at negedge clock) when the **clock is low**
- $t_{d\text{->}q}$: delay from d to q when the latch is transparent
- $t_{clk\text{->}q}$: delay from the rising clock edge to d propagating to q



Latch Circuits







S	R	Q	$\overline{Q}$
0	0	latch	latch
0	1	0	1
1	0	1	0
1	1	0	0

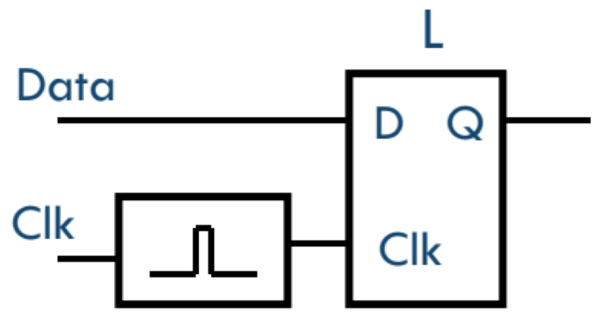
‘Feedback-breaking’ latch
Transparent high

‘State-forcing’ latch
Transparent low

SR latch
Common interview question

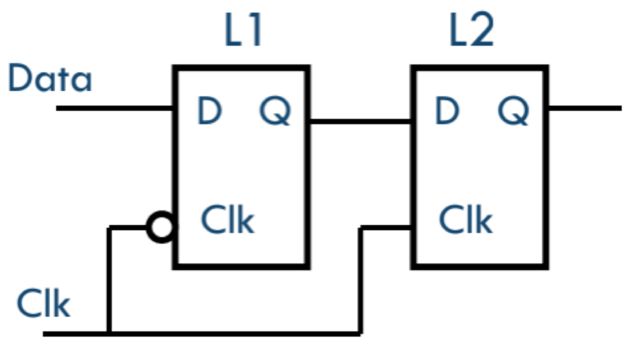
Building a Flip-Flop from Latches

- Clock pulsed latch
  - Latch becomes transparent for the pulse duration only, then holds data
  - Not common anymore, sometimes used in high performance circuits
  - Positive hold time



Pair of latches – edge triggered (posedge clk!!!)

- Commonly used technique
- L2 holds output data stable when clock is high.
- Negative hold time



Questions?