

# Discussion 12

Multipliers, Timing, FF/Latch Design, SRAMs

Sp19 Discussion 12 (multipliers): <http://inst.eecs.berkeley.edu/~eecs151/sp19/files/discussion12.pdf>

# Unsigned Multiplication Example

$$\begin{array}{r} 4'b0011 \quad (3) \\ * 4'b0110 \quad (6) \end{array}$$

- Partial Products can be generated in parallel
- Let's try to improve the addition of the partial products

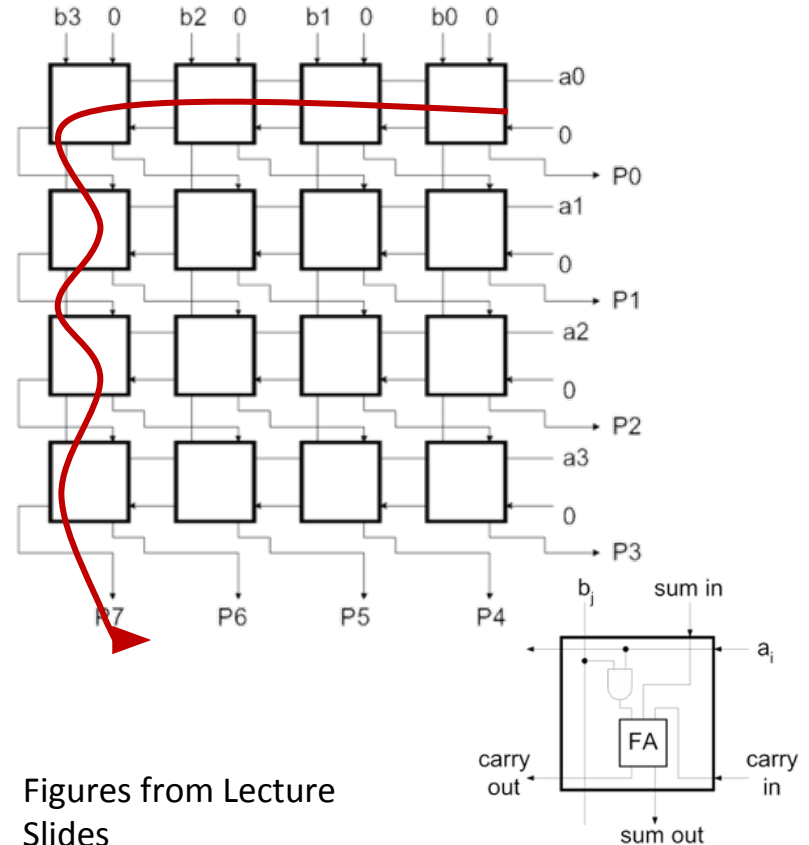
$$\begin{array}{r} 4'b0011 \quad (3) \\ * 4'b0110 \quad (6) \\ \hline \quad \quad 0000 \\ \quad \quad 0011 \\ \quad \quad 0011 \\ + \quad 0000 \\ \hline \quad \quad 00010010 \quad (18) \end{array} \quad \left. \begin{array}{l} \\ \\ \\ \end{array} \right\} \text{Partial Products}$$

# Multipliers

- Remember, the mechanics of multiplication in binary are generally the same as decimal multiplication (signed multiply requires a slight tweak).
- 2 Steps to Multiplication:
  - Generation of partial products
  - Adding partial products
- Making faster multipliers mostly involves changing how we deal with generating and adding the partial products

# Accelerating the Addition of Partial Products

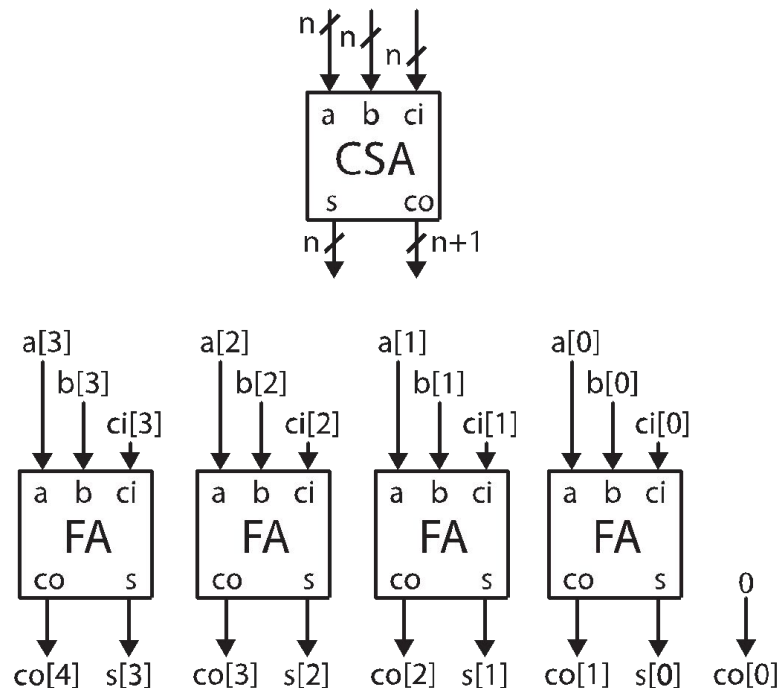
- Let's look at an (unsigned) array multiplier
- The products can be computed in parallel but the carry chain when adding partial products is limiting the speed
- How do we improve performance without having a large increase in hardware?
  - We could implement each adder as a parallel prefix or a carry-lookahead adder
  - However, remember that these adders require more logic than a simple carry ripple adder



Figures from Lecture Slides

# Carry Save Addition

- When we generate a carry in a given column of an addition, we add it to the 2 values in the next column.
  - This addition may in turn generate its own carry
- If adding carries is just like another addition, can we delay adding the carry bits until later?
  - Yes, so long as we remember what the carry bits need to be added
- This is the basis of the carry save adder:
  - Takes in a, b, and carry\_in (multi-bit)
  - Produces a sum and carry\_out (multi-bit)



# Carry Save Addition

Example: sum three numbers,

C

$$3_{10} = 0011, 2_{10} = 0010, 3_{10} = 0011$$

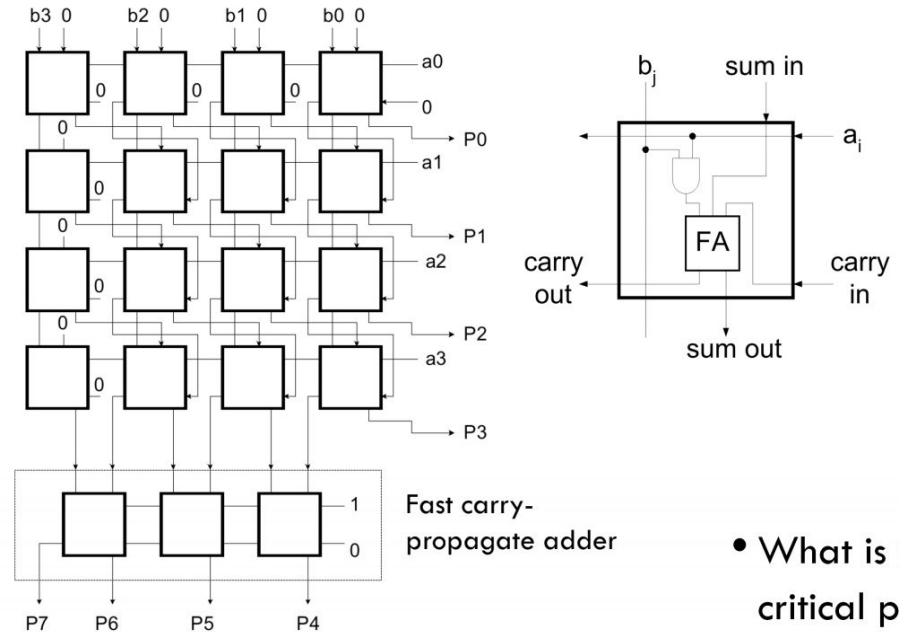
$$\left. \begin{array}{r} 3_{10} \ 0011 \\ + 2_{10} \ \underline{0010} \\ \hline c \ 0100 = 4_{10} \\ s \ 0001 = 1_{10} \end{array} \right\} \text{carry-save add}$$
  
$$\left. \begin{array}{r} + 3_{10} \ \underline{0011} \\ c \ 0010 = 2_{10} \\ s \ \underline{0110} = 6_{10} \\ 1000 = 8_{10} \end{array} \right\}$$

# Using Carry Save Addition

- Using Carry Save Addition Allows us to create a multi-input adder that is:
  - Relatively fast: Carry Save Adders do not have a carry ripple
  - Relatively small: do not need the logic to handle the carry logic to create a fast adder
- However, still need a standard adder at the end to add the final carry-out and sum.
  - This is one of the fast adders such as the Carry Lookahead or Parallel Prefix Adders
  - Good news! We only need one of them.

# Using Carry Save Addition in Multipliers

- Carry now propagates down each column.
  - Carry ripple across rows is eliminated in the array
- Still need to handle carries at the end with a fast adder
- Critical path now down a column + the carry-propagate adder delay

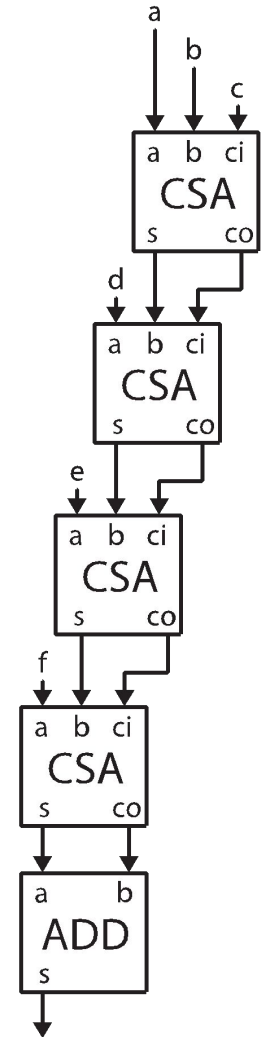
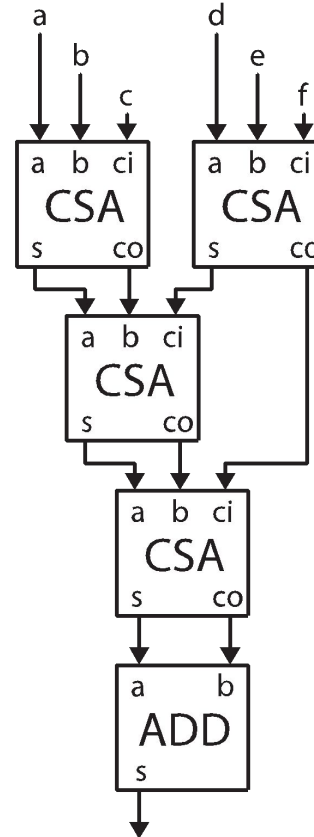


- What is  $t_{critical}$  for this circuit?



# Using Carry Save Addition

- Because addition is associative, it actually does not matter what order the carry bits are added back into the sum
  - Can use a tree structure

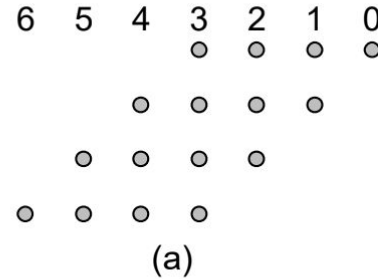


# Wallace Tree Multiplier

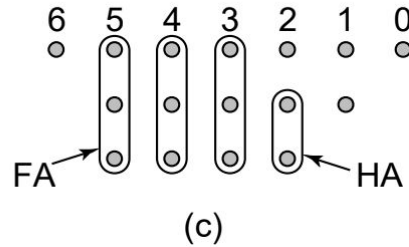
Method to construct a Wallace Tree:

1. Draw a dot diagram where each column has as many dots as the 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

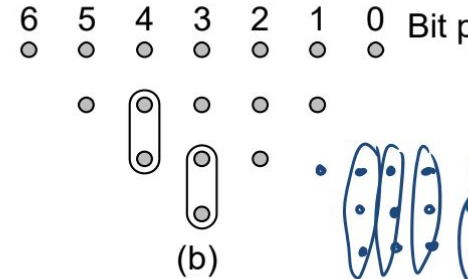
Partial products



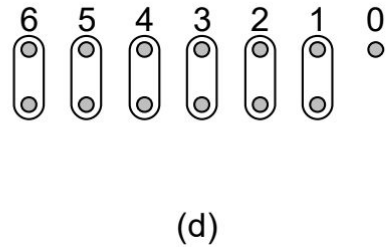
Second stage



First stage



Final adder



# Radix and Multiplication

- Binary arithmetic has some advantages
  - Partial product generation is just a series of AND gates (including sign extension)
- However, there are also disadvantages
  - There is a partial product for each bit of the multiplier
  - That leads to a lot of partial products (a lot of additions)
- Ex.  $3 \times 4$ 
  - single partial product in base 10
  - 4 partial products in base 2.
- Why don't we consider a larger radix?

# Radix 4 Multiplication

- Let's consider 2 bits at a time
  - Halve the number of partial products we generate
- Radix 4 multiplication  $A * B$ 
  - Partial Product Shift By 2 bits each time

B Digit	Partial Product	Partial Product (Rewritten)
0	$0 * A$	0
1	$1 * A$	A
2	$2 * A$	$4 * A - 2 * A$
3	$3 * A$	$4 * A - A$

- Recall: Multiplications by powers of 2 are left shifts
- Can we use this property?

# Booth Recoding

- Uses radix 4 arithmetic
- Modification: Partial Products for  $B=2$  and  $B=3$  can be separated into  $4*A - \{2, 1\}A$
- $4*A$  can be implemented as a shift to the left by 2
- $2*A$  can be implemented as a shift to the left by 1
- Recall that we are doing radix 4 multiplication, we shift left by 2 positions for the next partial product
- Therefore, any  $4*A$  term can be handled in the next partial product!
  - To do this, the multiplier needs to look at 3 (rather than just 2) bits. The extra bit is the MSB of the previous

B Digit	Partial Product	Partial Product (Rewritten)
0	$0*A$	0
1	$1*A$	A
2	$2*A$	$4*A - 2*A$
3	$3*A$	$4*A - A$

# Booth Recoding

$B_{i+1}$	$B_i$	$B_{i-1}$	Action	Comment
1				
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)

B Digit	Partial Product	Partial Product (Rewritten)
0	$0*A$	0
1	$1*A$	A
2	$2*A$	$4*A - 2*A$
3	$3*A$	$4*A - A$

# Booth Recoding Example (Unsigned)

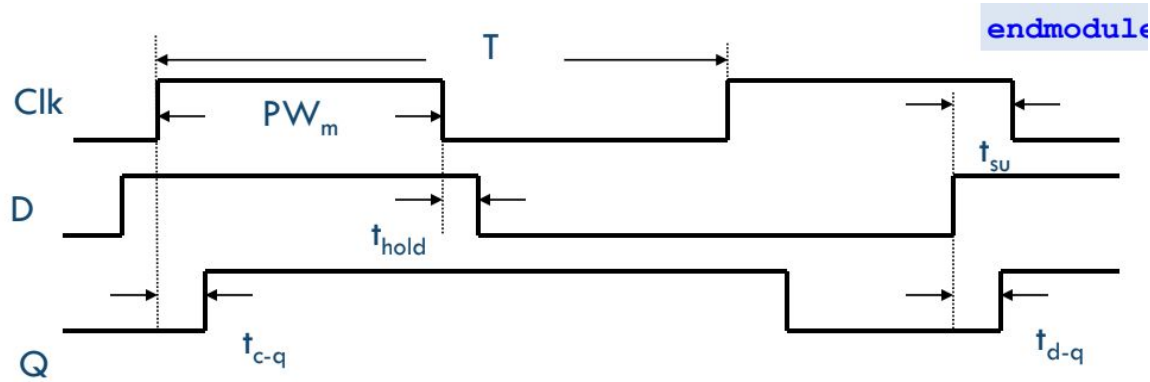
- Example:  $6 \times 4$
- $B_{-1} = 0$

```

      4'b0110 (6)
    * 4'b0111 (7)
    -----
-      0110 (Sub A)
+   01100 (Add 2A)
+   0000 (Add 0)
    -----
+  1111010 (Sub A)
+   01100 (Add 2A)
+   0000 (Add 0)
    -----
(1)00101010 (42)
  
```

$B_{i+1}$	$B_i$	$B_{i-1}$	Action
1			
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

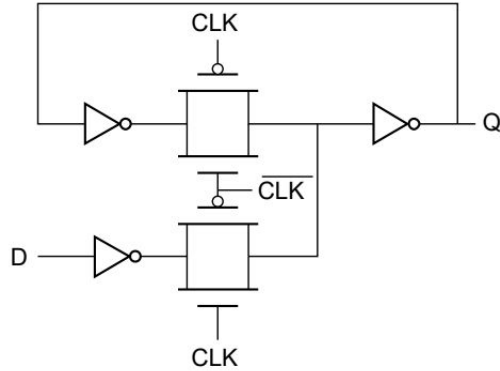
# Latch Timing



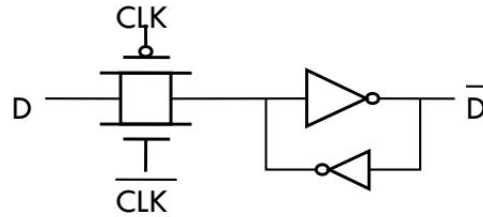
- A positive latch is transparent ( $q = d$ ) when the clock is high and opaque ( $q = d$ , during negedge clock) when the clock is low
- $t_{d \rightarrow q}$  is the delay from d to q when the latch is transparent
- $t_{clk \rightarrow q}$  is the delay from the rising clock edge to the new value of d propagating to q



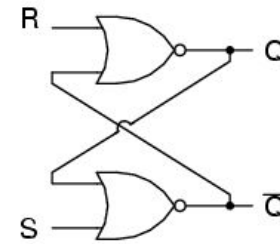
# Latch Circuits



‘Feedback-breaking’  
mux latch  
Transparent high



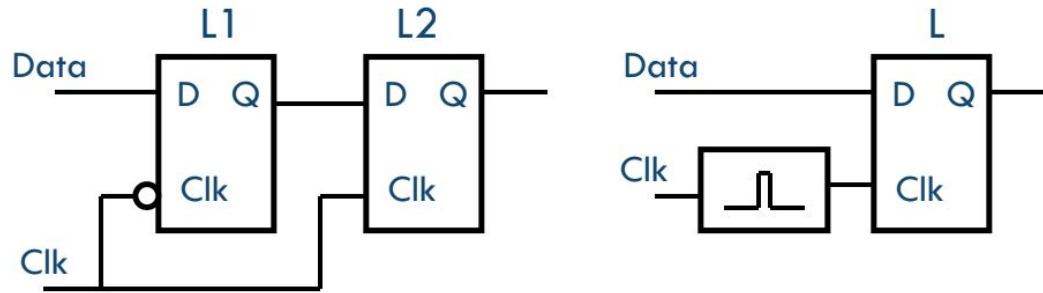
‘State-forcing’ latch  
Transparent low



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

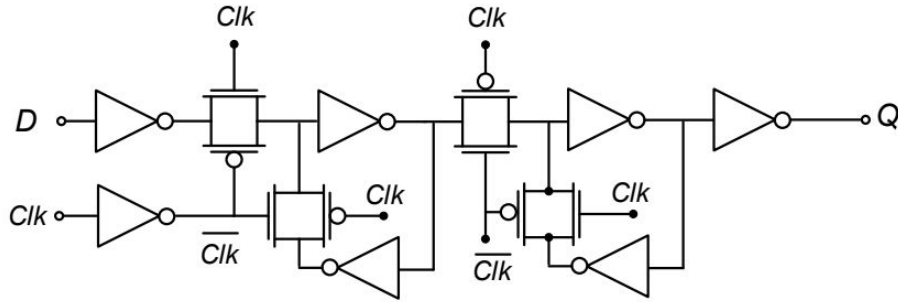
SR latch  
Common interview  
question

# Building a Flip-Flop from Latches



- Clock pulsed latch
  - Latch becomes transparent for a short time and holds the value it received on the pulse
  - Not common anymore, sometimes used in high performance circuits
- Master-slave latches
  - Commonly used technique, go over timing diagram on board

# Flip-Flop Hold/Setup/clk->q Time



- This is a negative edge flip-flop as drawn
- We'll consider the positive edge case

- Hold time = the amount of time after a clock edge that the data input needs to be stable for
- Setup time = the amount of time before a clock edge that the data input needs to be stable to be properly latched internally
- Clk-q time = delay from a clock edge to q being updated with the new value

# Path Timing Constraints (Hold + Setup)

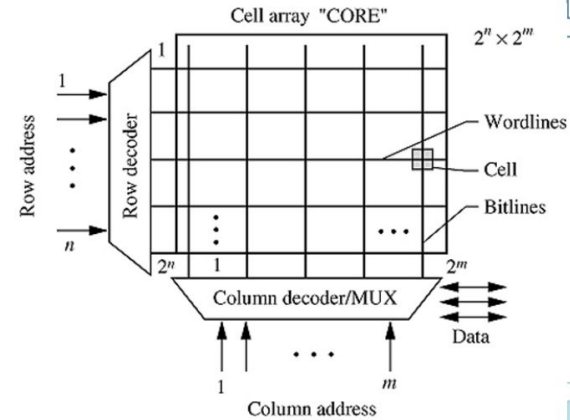
Setup constraint:  $T_{\text{clk}} > t_{\text{clk} \rightarrow \text{q}} + t_{\text{logic,max}} + t_{\text{setup}}$

Hold constraint:  $t_{\text{hold}} < t_{\text{clk} \rightarrow \text{q}} + t_{\text{logic,min}}$

- Skew is the deterministic clock arrival time difference between 2 flops
- Positive skew = receiving edge arrives later than nominal
- Negative skew = receiving edge arrives earlier than nominal
- Jitter is non-deterministic clock arrival differences
  - Can be treated like skew in timing calculations, assuming worst case jitter
- New timing equations:
  - $T_{\text{clk}} > t_{\text{clk} \rightarrow \text{q}} + t_{\text{logic,max}} + t_{\text{setup}} - t_{\text{skew}}$ 
    - Note positive skew can improve clock frequency
  - $t_{\text{hold}} + t_{\text{skew}} < t_{\text{clk} \rightarrow \text{q}} + t_{\text{logic,min}}$ 
    - Note positive skew hurts hold margin

# SRAM Architecture

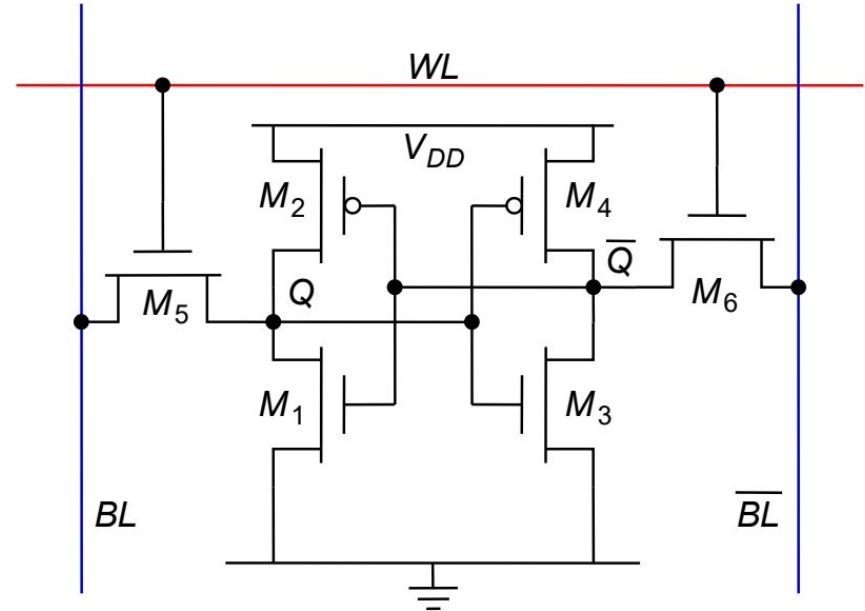
- CORE
  - Wordlines to access rows
  - Bitlines to access columns
  - Data multiplexed onto columns
- Decoders
  - Addresses are binary
  - Row/column MUXes are 'one-hot' - only one is active at a time



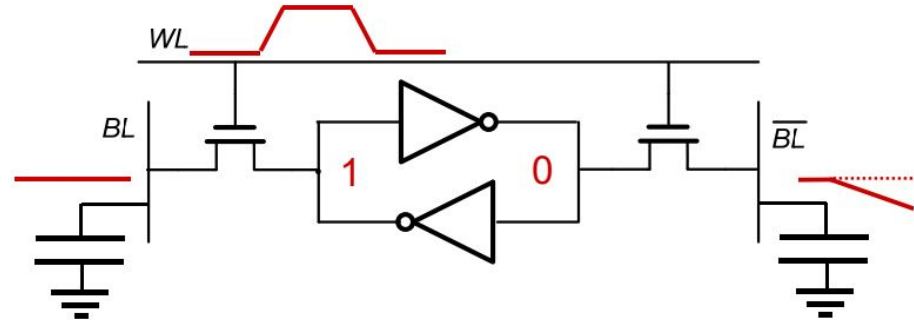
- SRAM cells arranged in a grid
- Bitlines are shared across cells in a column, they are often long wires with a large capacitive load (connect to drains of access transistors)
- Wordlines are shared across cells in a row, they connect to the gates of access transistors
- Peripheral circuitry (bitline drivers, sense amp, decoders)

# 6T SRAM Cell

- Inverters in positive feedback form the memory element
- M5/M6 are the access transistors; they allow the bitlines to access the memory nodes (Q, Qbar) when  $WL = 1$
- Only 1 WL in an SRAM array is active at a time and it addresses an entire row of SRAM cells
- Bitlines are controlled differently for read and write



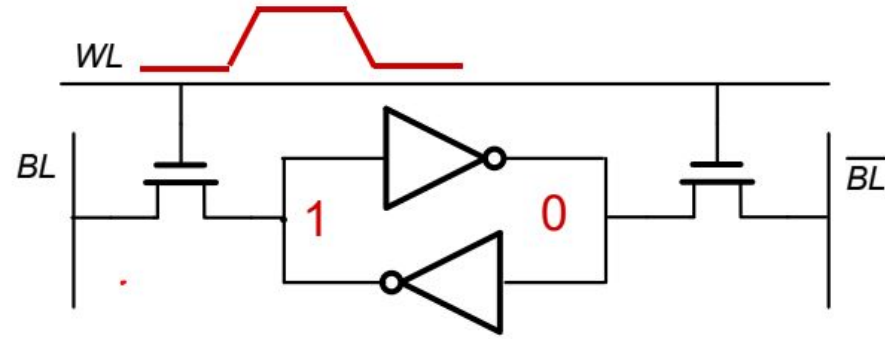
# SRAM Read



- 1) precharge BL and BLbar to VDD, 2) raise WL, 3) sense dip on one bitline with sense amp, 4) lower WL, 5) discharge bitlines
- Read stability = reading doesn't corrupt the value stored in Q and Qbar
  - The pass transistor shouldn't overpower the node storing a '0' and flip its state (consider voltage divider from bitline to Q)
- We choose to make the NMOSes in the inverters stronger than the pass transistor = ( $W_n > W_{pass}$ ) to prevent read corruption

# SRAM Write

Write



- 1) drive BL and BLbar with new values, 2) raise WL, 3) wait some time (write time), 4) lower WL, 5) discharge bitlines
- Write-ability = the cell's memory value can be changed
  - requires the pass transistor overpower one of the data nodes
  - If we assume the cell is read stable, the inverter NMOS is stronger than the pass transistor. This means the node with '0' can't be overpowered => so we must overpower PMOS.
- Pass transistor strength > PMOS pullup strength = ( $W_{pass} > W_p$ )
  - Voltage divider on '1' node must be strong enough to cause inverters to switch