

EECS 151/251A

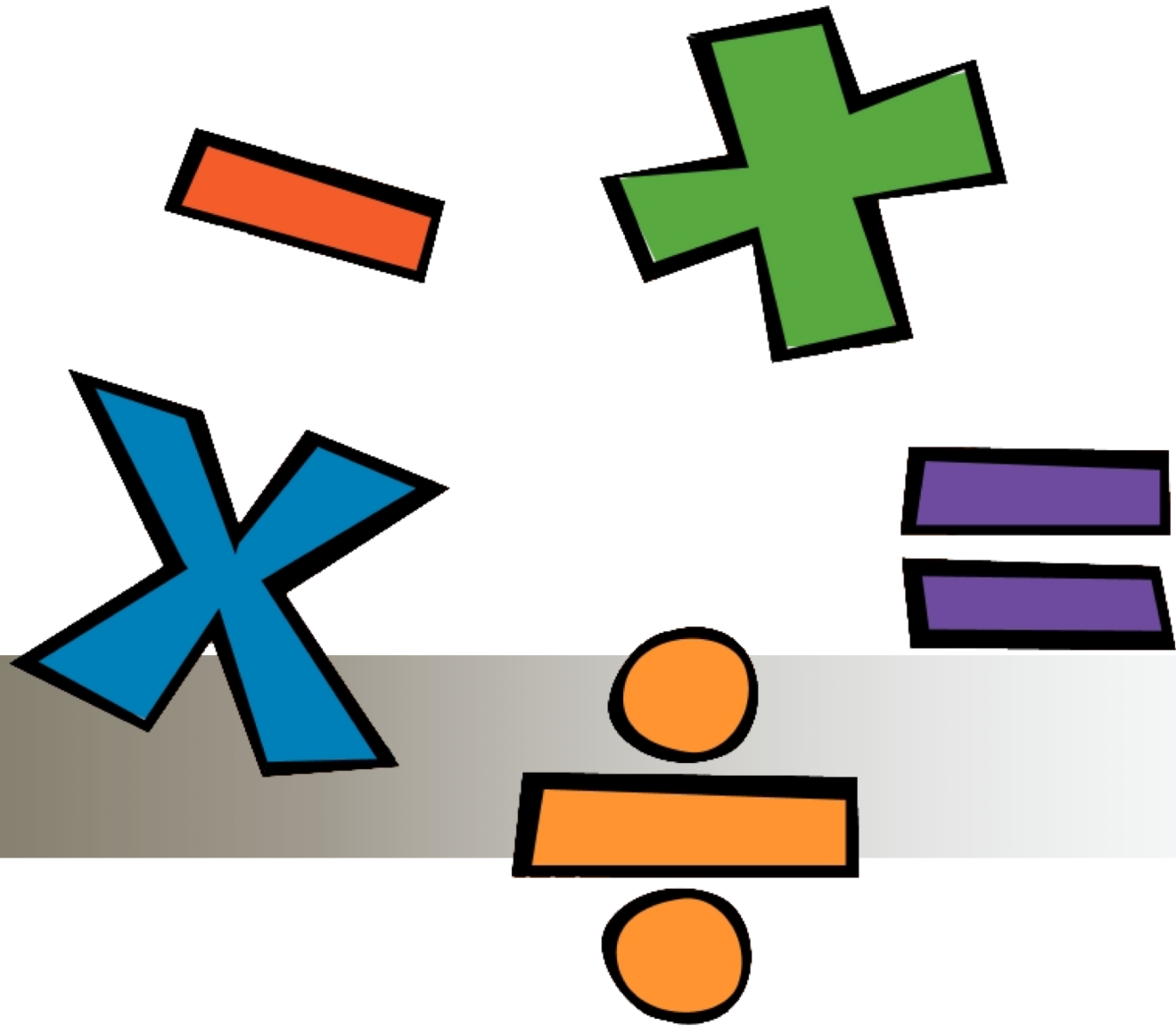
SP2022 Discussion 9

GSI: Yikuan Chen, Dima Nikiforov

Agenda

- Multiplier
- Latch

Multiplier



Unsigned Multiplication Example

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

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

The diagram illustrates the multiplication of two 4-bit numbers, 4'b0011 (3) and 4'b0110 (6). The first number is underlined in red, with a red bracket above it labeled '4'. The second number is also underlined in red, with a red bracket to its right labeled '4'. Below the second number, four red arrows point upwards to the bits 0, 1, 1, and 0, indicating the generation of partial products. The partial products are shown as 0000, 0011, 0011, and 0000, each underlined in red. A blue bracket on the right groups these four lines under the label 'Partial Products'. A red bracket on the right groups the two 0011 lines. A red checkmark is present to the right of the partial products. Below a dashed line, the final product is shown as 00010010 (18). Red arrows point from the '1' at the 16's place and the '0' at the 2's place to the values '16' and '+ 2' respectively.

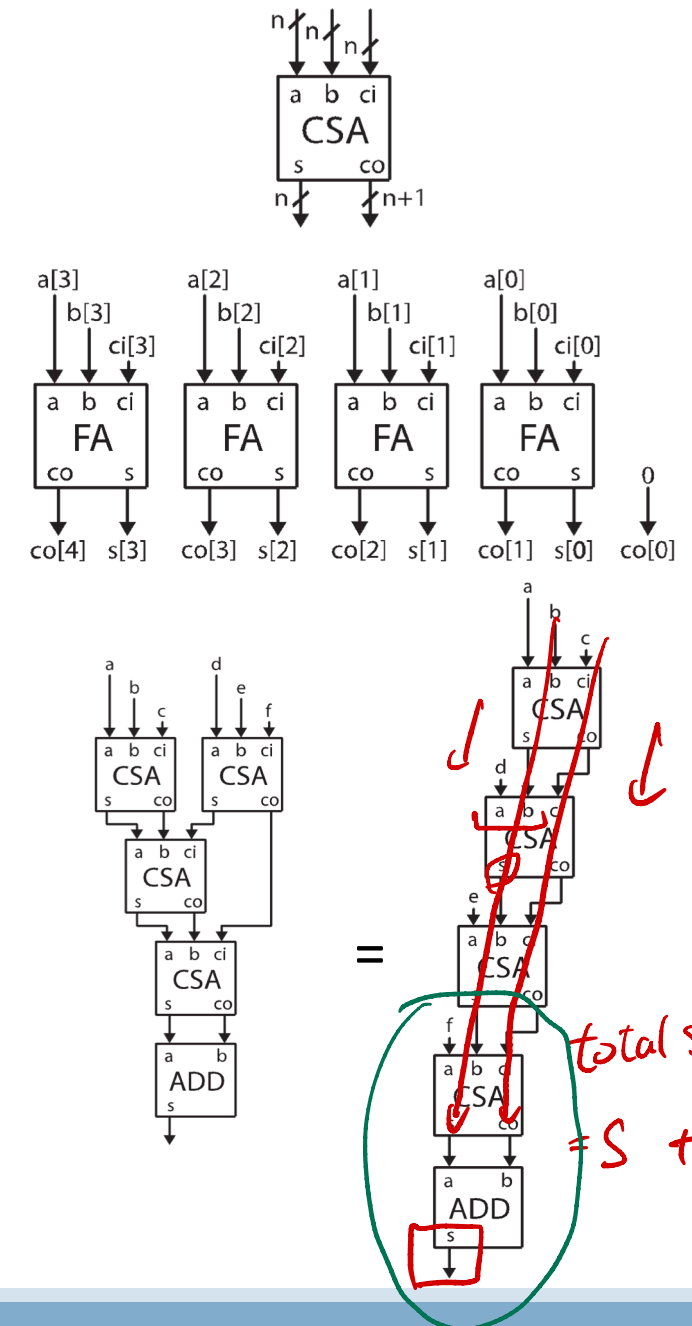
$$\begin{array}{r} 4'b0011 \quad (3) \\ \times 4'b0110 \quad (6) \\ \hline 0000 \\ 0011 \\ 0011 \\ + 0000 \\ \hline 00010010 \quad (18) \end{array}$$

Partial Products

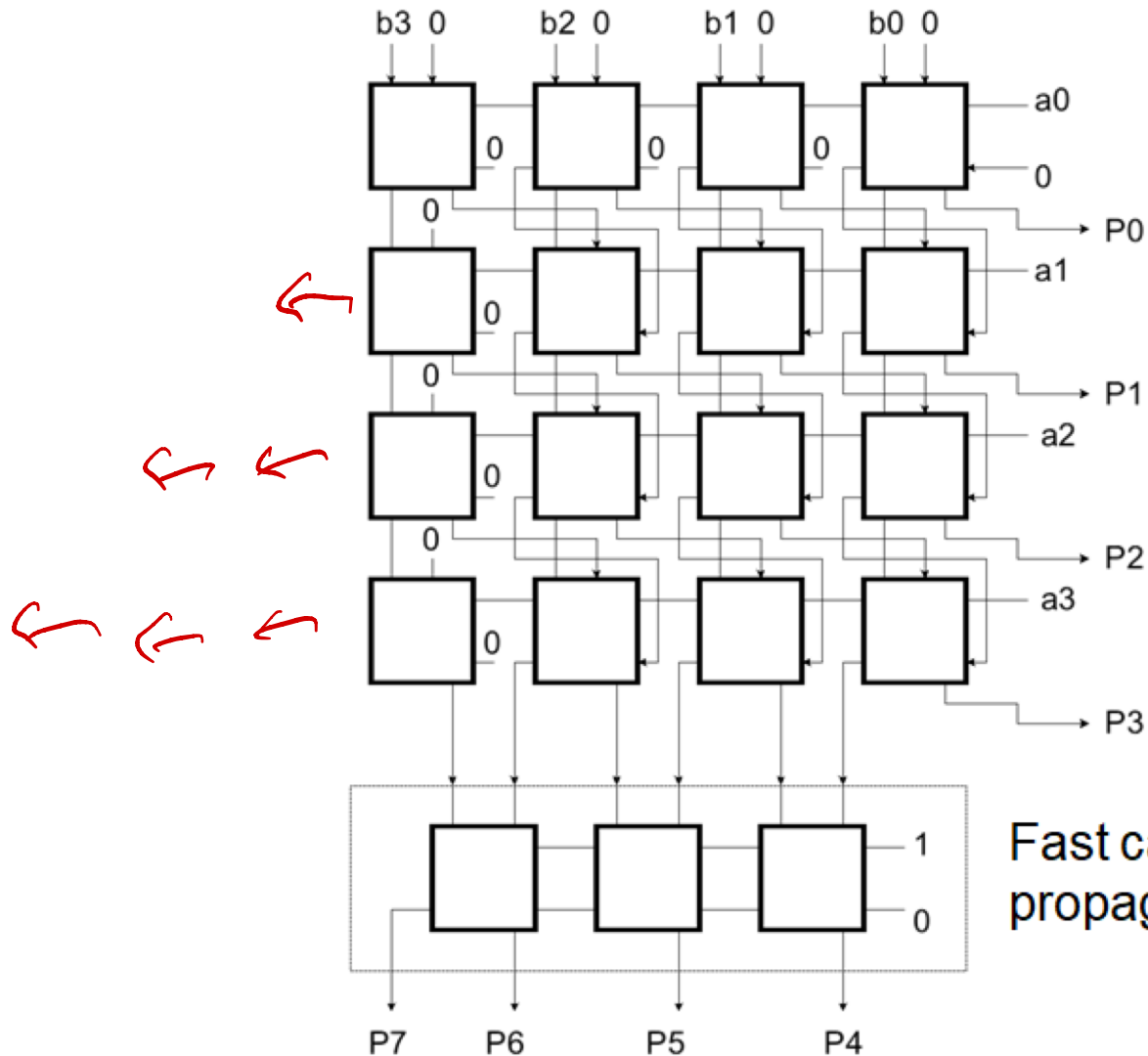
16 + 2

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



Handwritten annotations in red and green:

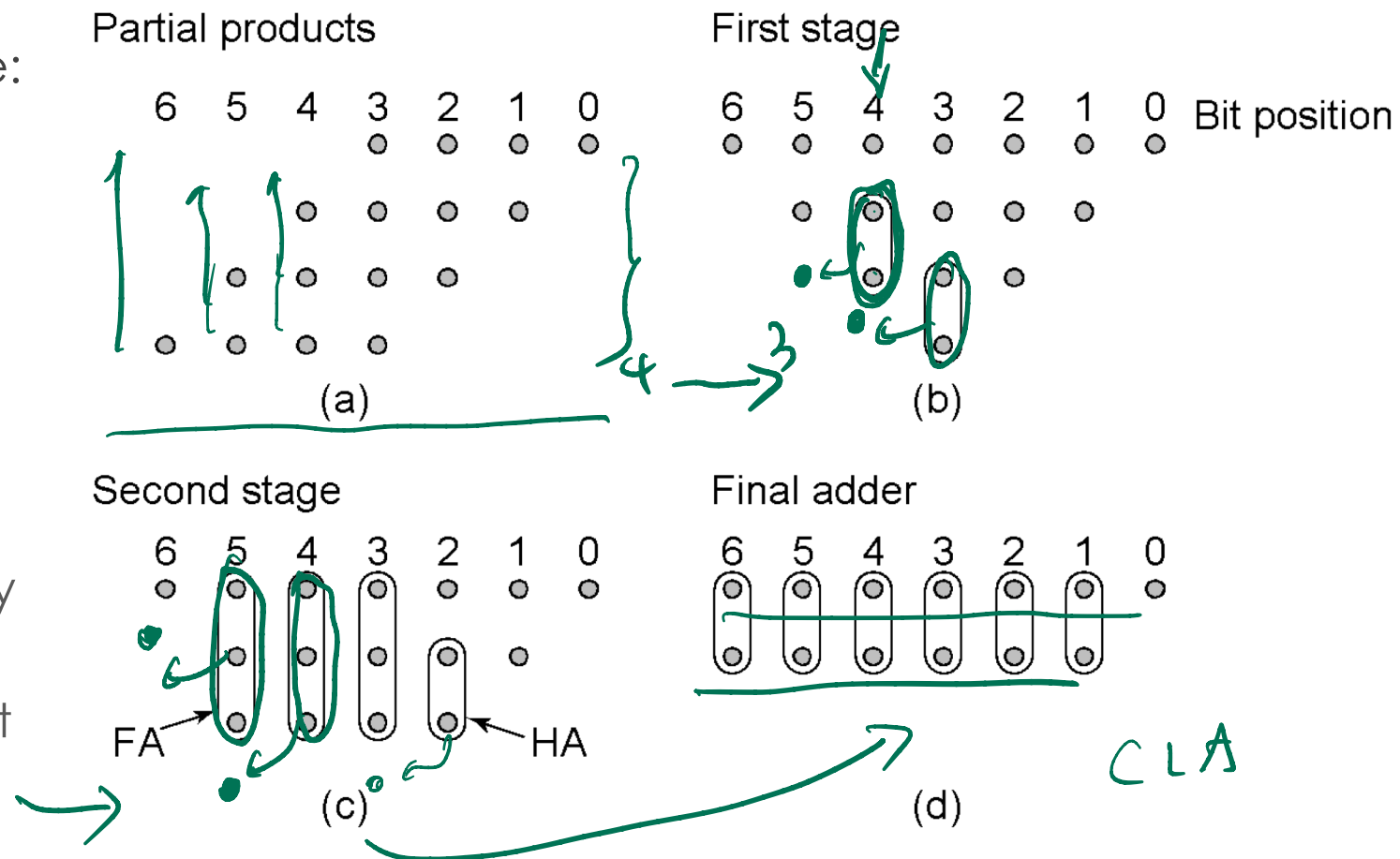
- A detailed view of a full adder cell showing inputs a_i, b_j and carry in, and outputs $sum\ out$ and $carry\ out$.
- Binary numbers: $A = 101000$ (40), $B = 011001$ (25), and $C_{in} = 010100$ (20).
- Equation: $A \oplus B \oplus C$
- Binary addition showing carry propagation:

$$\begin{array}{r}
 101000 \text{ A} \\
 + 011001 \text{ B} \\
 + 010100 \text{ C}_{in} \\
 \hline
 \text{"Sum"} \quad 100101 \quad O(1) \\
 \text{"Carry"} \quad 011000 \quad O(1) \\
 \hline
 1010101 \leftarrow O(\log N)
 \end{array}$$

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)
2 - 2 input add
3 - 2 input add
3. Propagate carries and sum by adding one dot in the grouped column and one dot in the next column

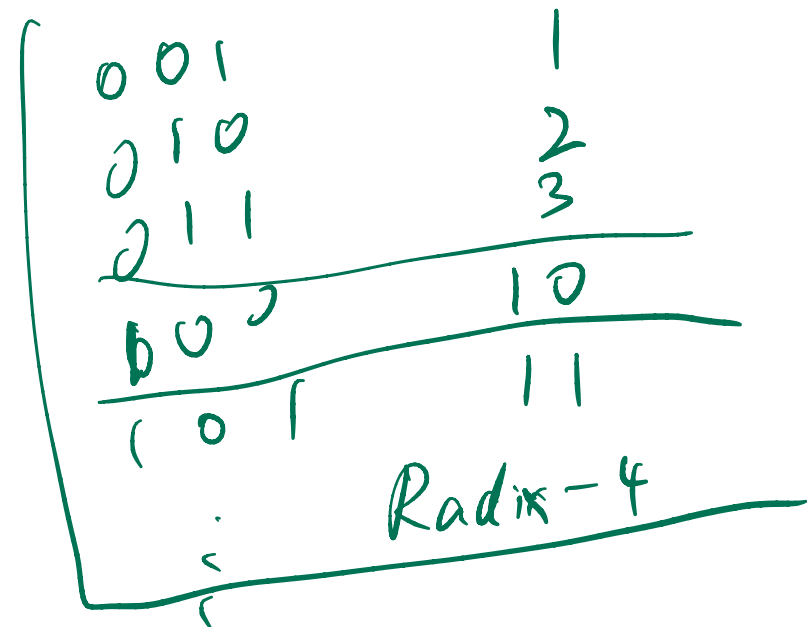


Radix-4 Multiplication

- Binary multiplication \rightarrow 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) \rightarrow halve number of partial products

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

$\underline{1101} * \underline{0110} \rightarrow 1101 * [01][10] \quad 4-2$



Booth Recoding –

- $4*A = A \ll 2$
- $2*A = A \ll 1$
- 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	$0*A$	<u>0</u> ←
01	$1*A$	<u>A</u> ←
10	$2*A$	<u>$4*A - 2*A$</u> ←
11	$3*A$	<u>$4*A - A$</u> ←

Handwritten annotations: A green circle highlights the '10' and '11' rows. A red circle highlights the 'A' in the '01' row. Red arrows point to the left in the '00', '01', '10', and '11' rows. A red arrow points to the '10' row from below.

$$B = \underbrace{01}_{4A} \underbrace{10}_{-2A}$$

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)

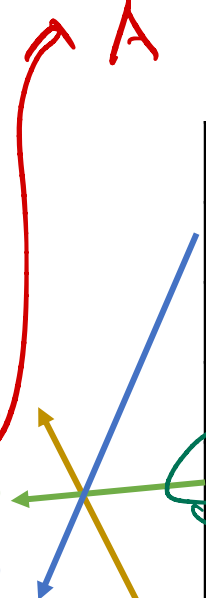
Booth Recoding Example (Unsigned)

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

$$\begin{array}{r}
 4'b0110 \quad (6) \\
 * 4'b0111 \quad (7) \\
 \hline
 - \quad 0110 \quad (\text{Sub } A) \\
 + \quad 01100 \quad (\text{Add } 2A) \\
 + \quad 0000 \quad (\text{Add } 0) \\
 \hline
 + \quad 1111010 \quad (\text{Sub } A) \\
 + \quad 01100 \quad (\text{Add } 2A) \\
 + \quad 0000 \quad (\text{Add } 0) \\
 \hline
 \text{(-1)}00101010 \quad (42)
 \end{array}$$

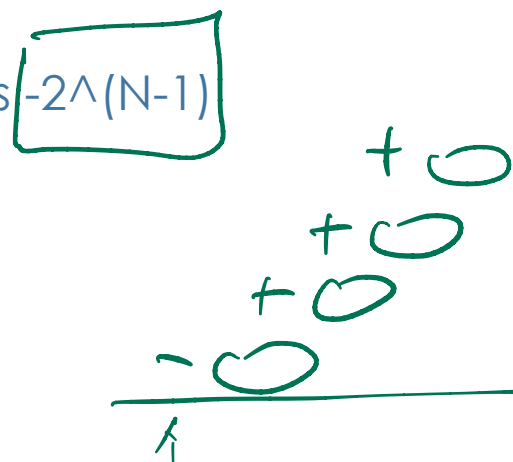
A for 01
A for previous stage's 4A

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:
 - Subtract last partial product
 - Sign-extend the rest of the partial products
- Recall 2's complement negation: subtract $A = \text{add } (\sim(A) + 1)$
 - Result: basically same hardware as unsigned mult.
 - Implementation: invert some bits, insert a 1 left of the first & last partial products



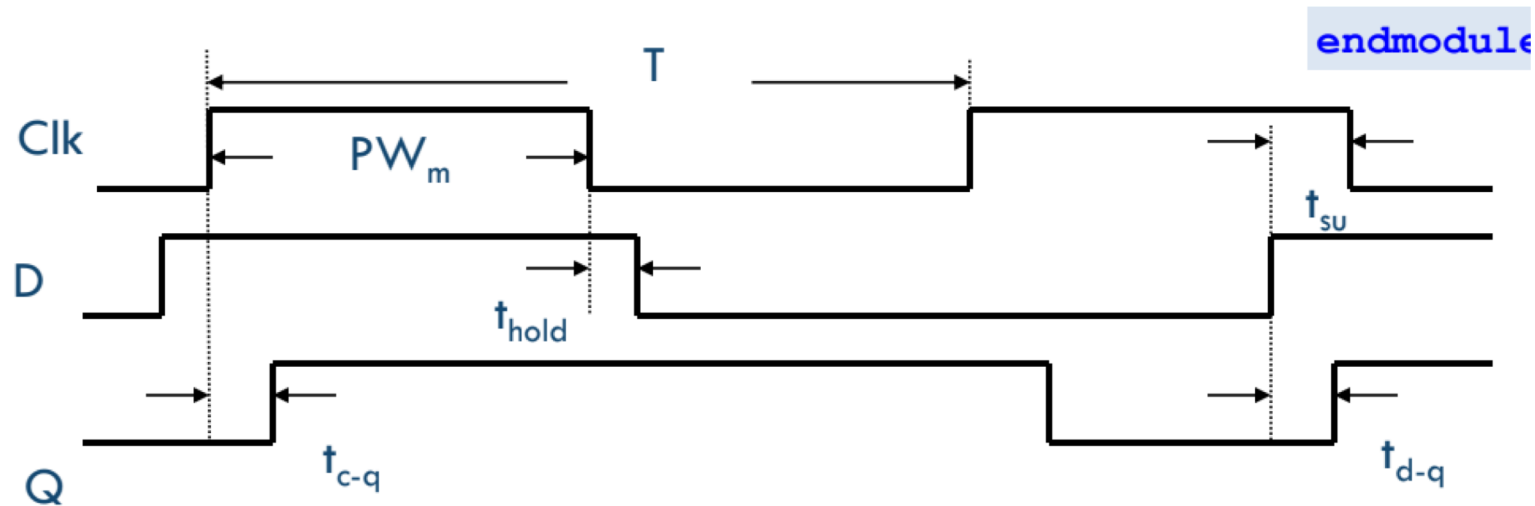
			1	p0[3]	p0[2]	p0[1]	p0[0]	
+			~p1[3]	p1[2]	p1[1]	p1[0]	0	
+		~p2[3]	p2[2]	p2[1]	p2[0]	0	0	
+	1	~p3[3]	~p3[2]	~p3[1]	~p3[0]	0	0	0
	P[7]	P[6]	P[5]	P[4]	P[3]	P[2]	P[1]	P[0]

Latch

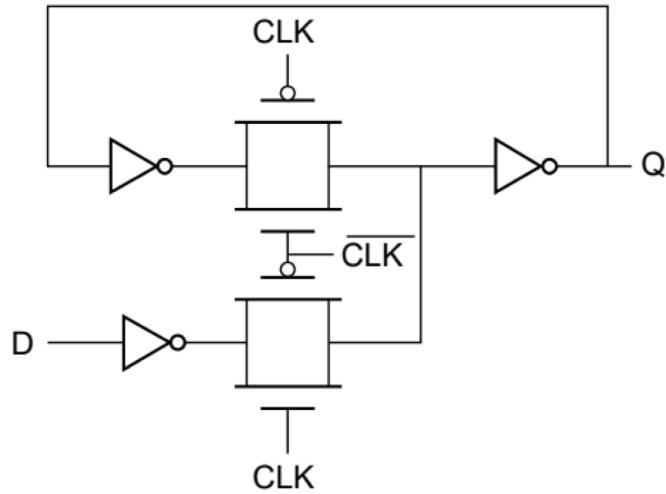


Latch Timing

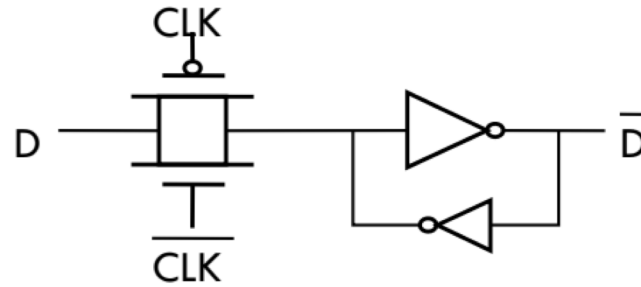
- A positive latch is **transparent** ($q = d$) when the **clock is high** and **opaque** ($q = d$, sampled at negedge clock) when the **clock is low**
- $t_{d \rightarrow q}$: delay from d to q when the latch is transparent
- $t_{clk \rightarrow q}$: delay from the rising clock edge to d propagating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



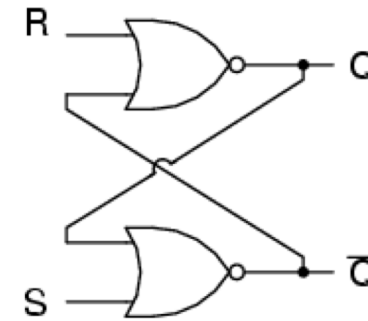
Latch Circuits



'Feedback-breaking' latch
Transparent high



'State-forcing' latch
Transparent low

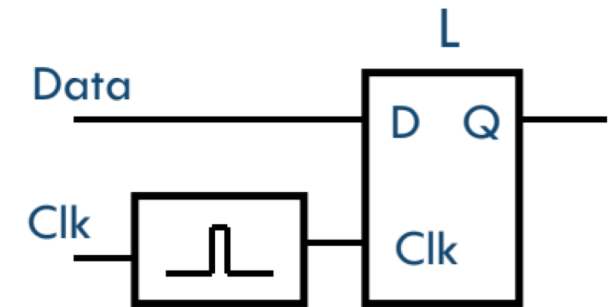


SR latch
Common interview question

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

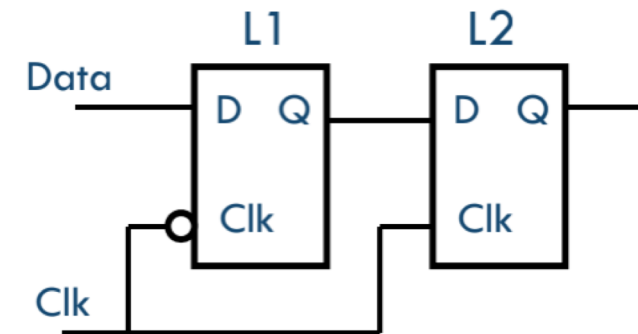
Building a Flip-Flop from Latches

- 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?