Introduction to Digital Logic

Lecture 13:

Demuxes

Adders

Overflow

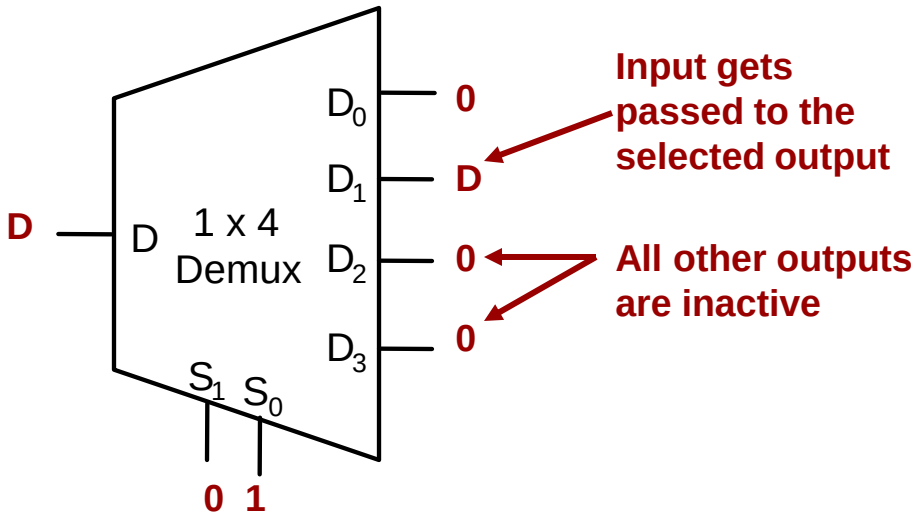
Carry-Lookahead Adders

DEMULTIPLEXERS

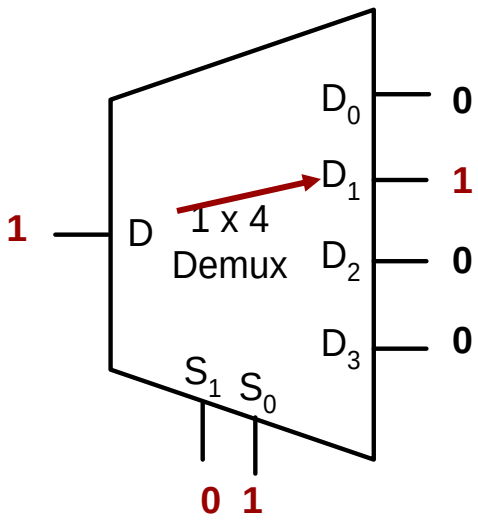
Demultiplexers

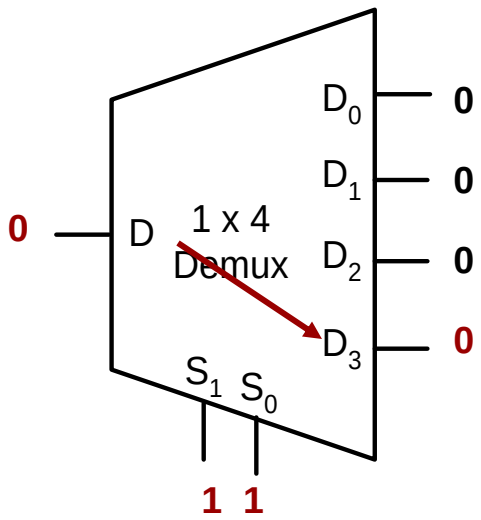
- Perform opposite function of multiplexers
- Pass the input to one selected output
- In general
  - 1 input
  - 2^n outputs
  - n select bits

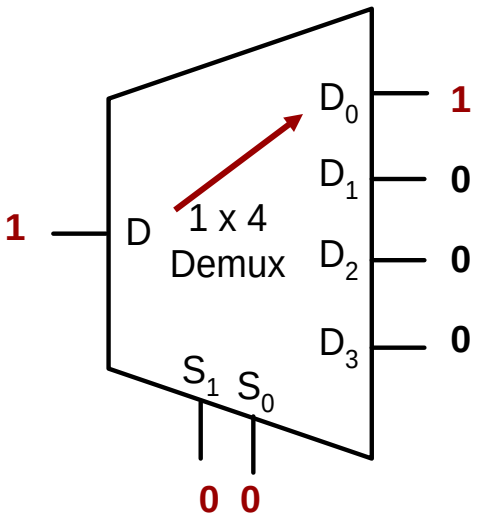
S_1	S_0	D_0	D_1	D_2	D_3
0	0	D	0	0	0
0	1	0	D	0	0
1	0	0	0	D	0
1	1	0	0	0	D



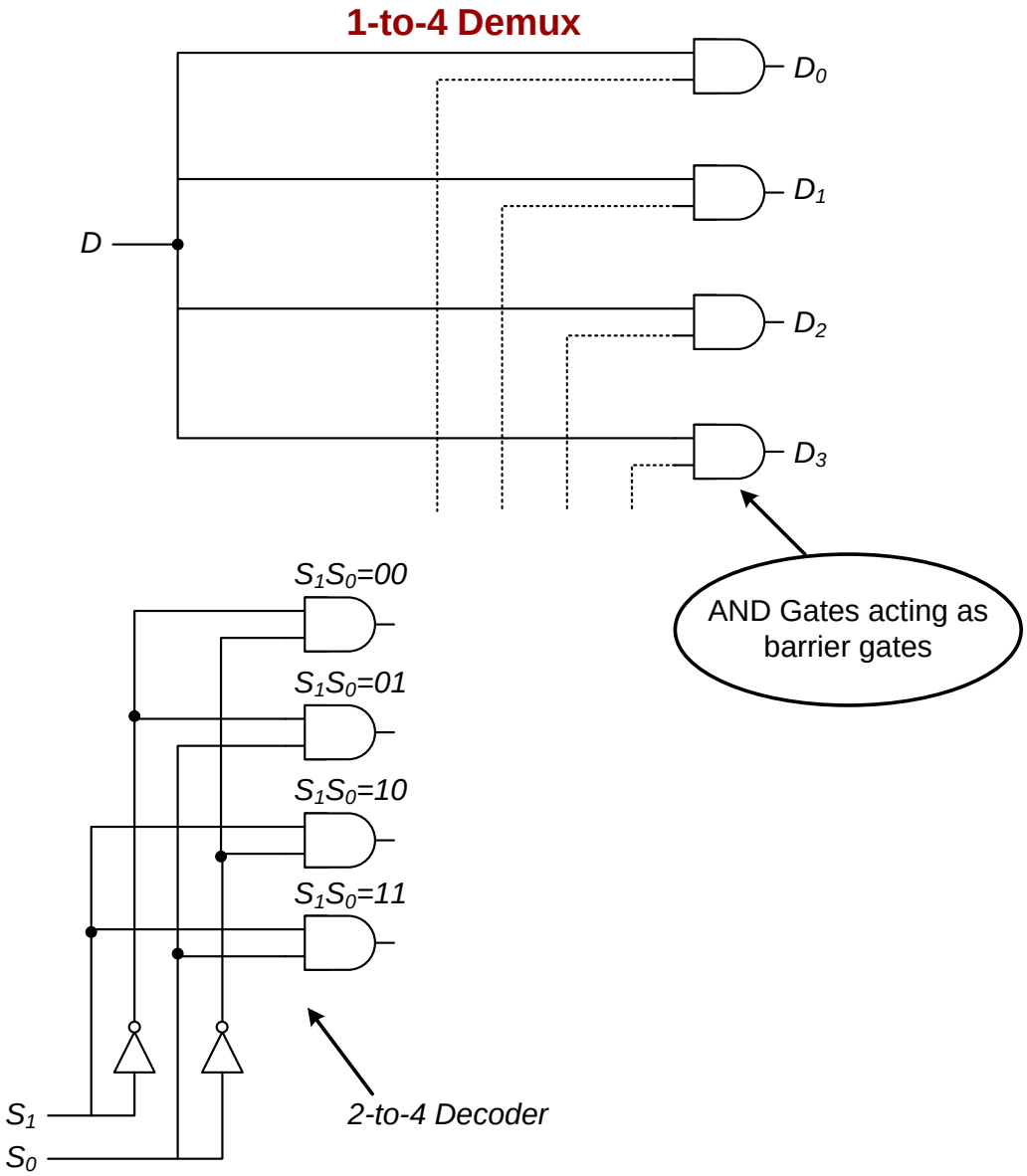
Demultiplexers



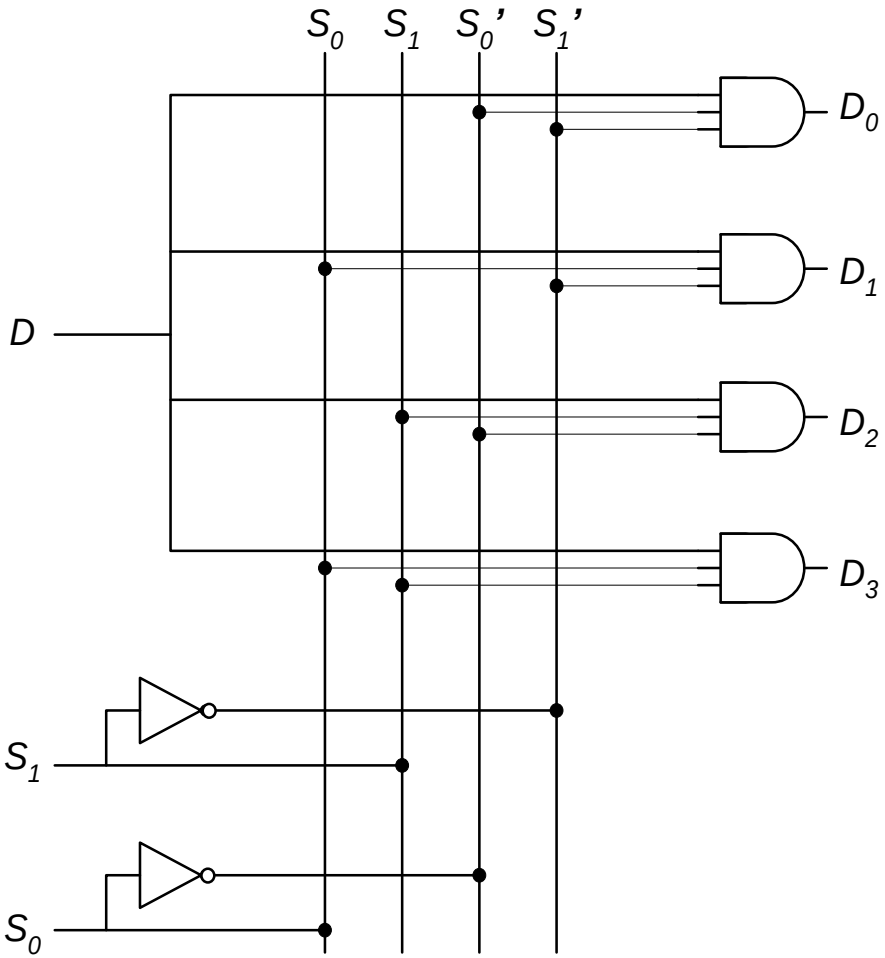




Demultiplexer Design



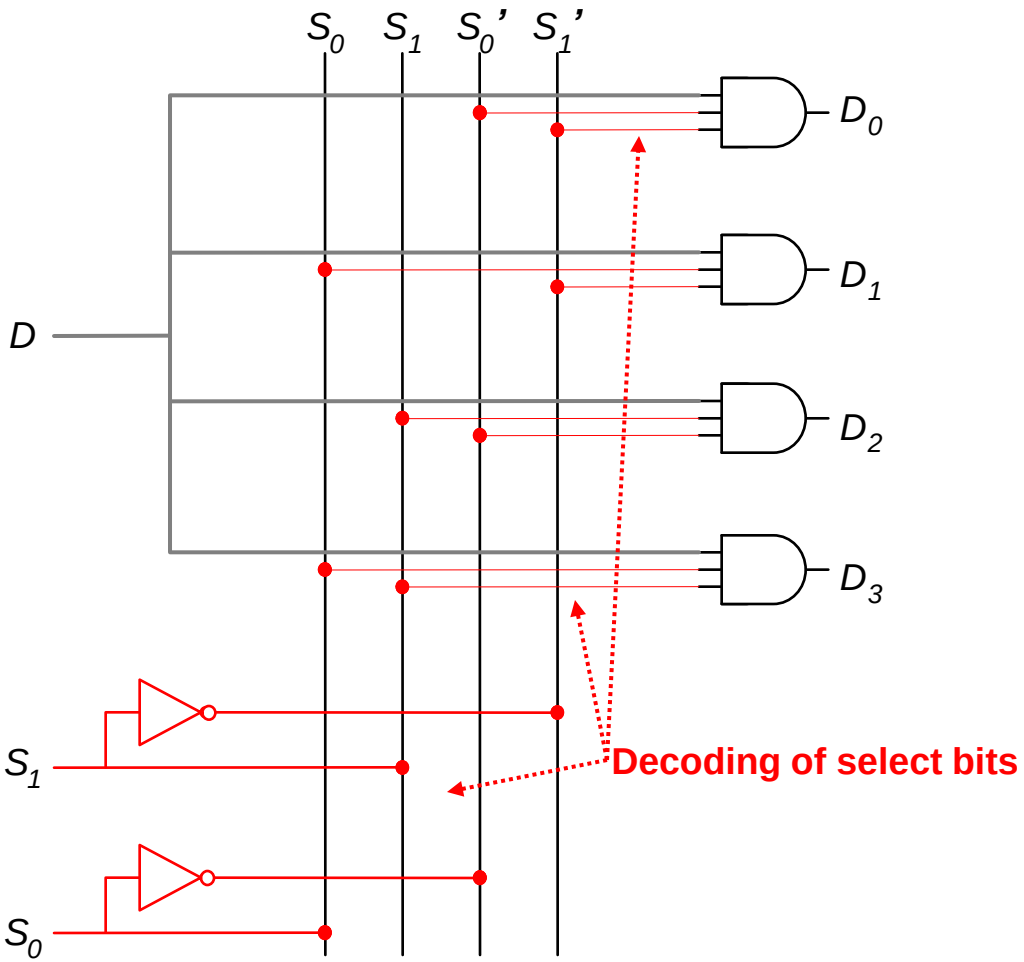
Demultiplexer Design



1-to-4 Demux

Demultiplexer Design

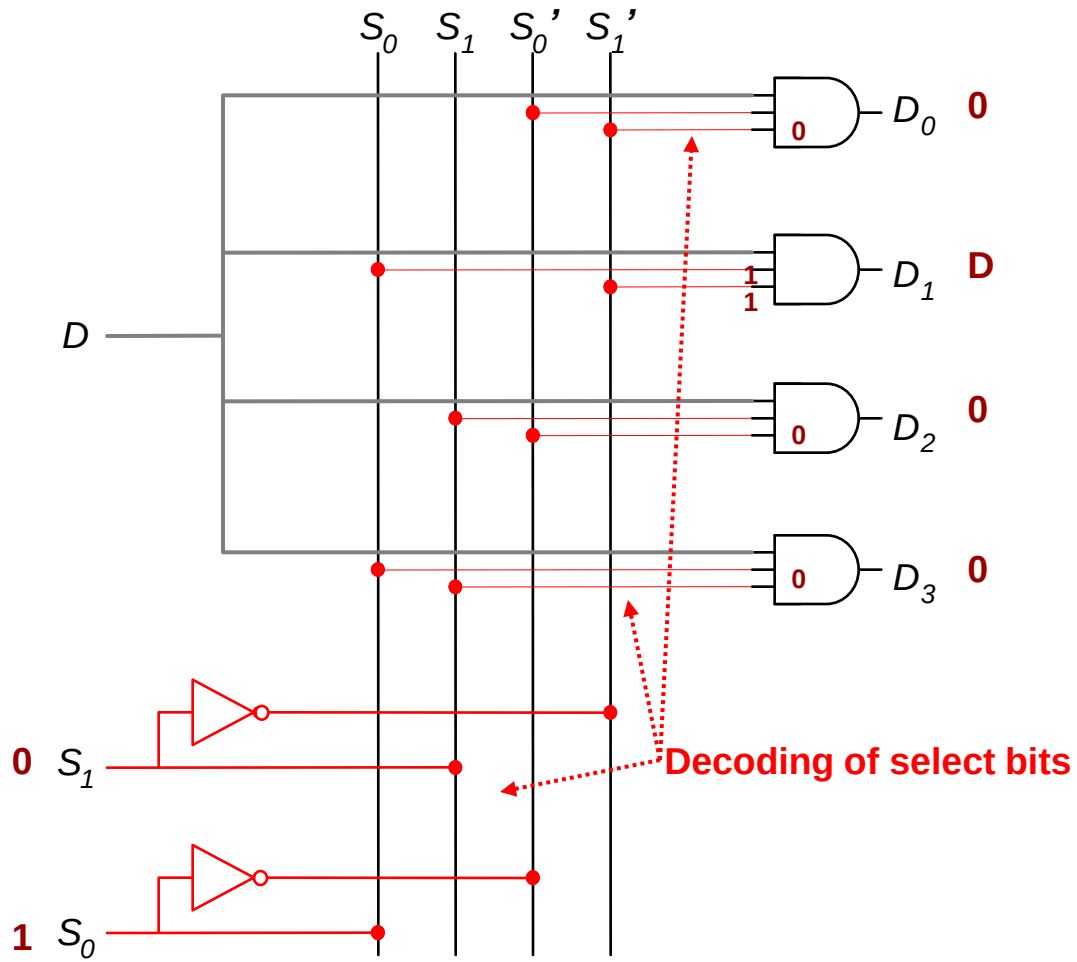
Notice D runs to all 4 output AND gates



1-to-4 Demux

Demultiplexer Design

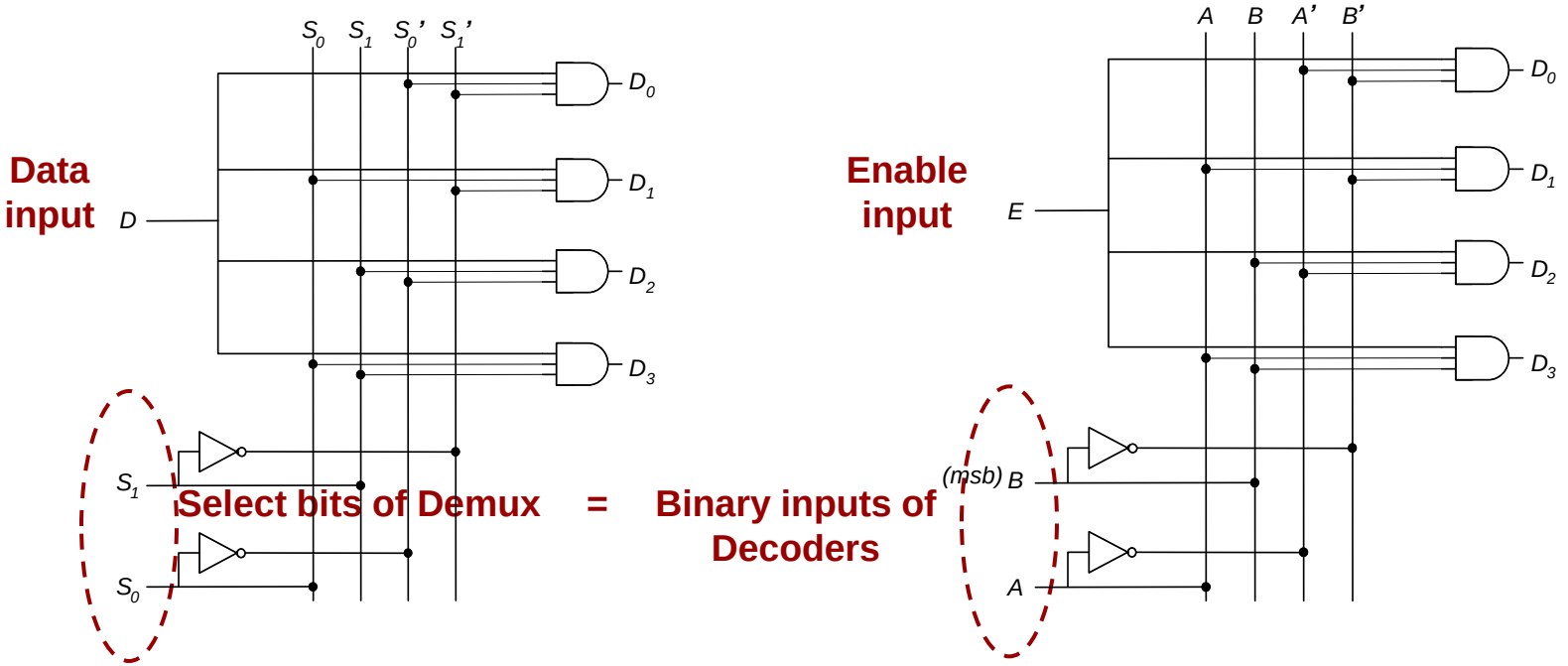
Notice D runs to all 4 output AND gates



1-to-4 Demux

Demultiplexers and Decoders

- Demultiplexers are actually just decoders w/ an enable (must have an enable)

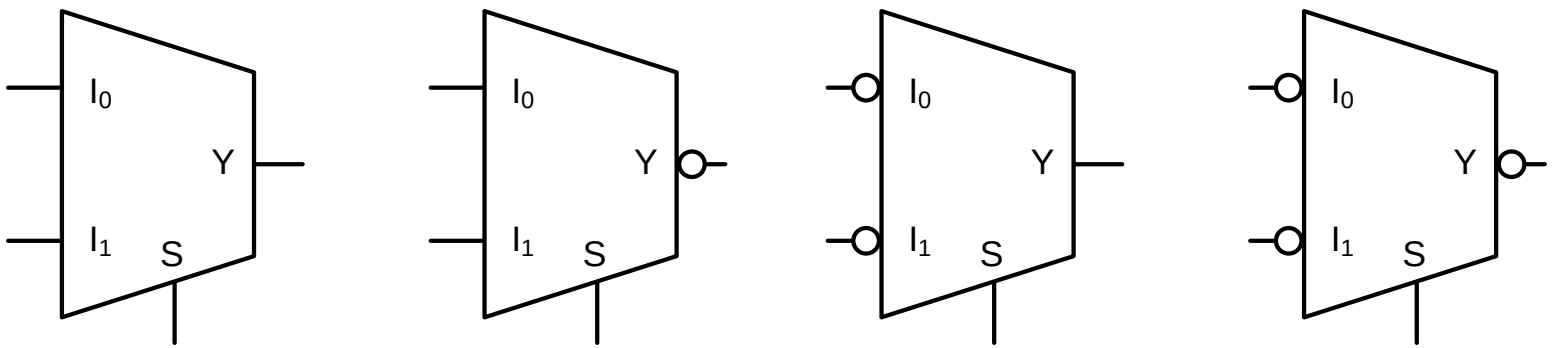


1-to-4 Demux

2-to-4 Decoder w/ Enable

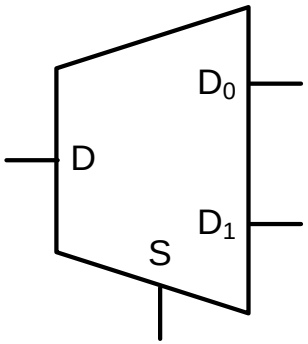
Mux Active Levels

- We don't think of mux inputs/outputs as active or inactive
- Instead we count how many inversions "a single path" hits from input to output (don't count total inversions) and list the mux as "inverting" (odd # of inversions) or non-inverting (even # of inversions)

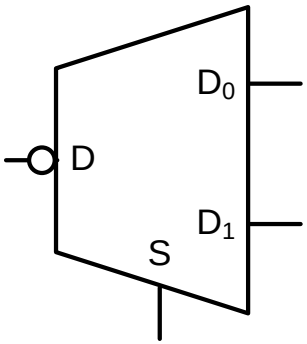


Non-Inverting Mux | **Inverting Mux** | **Inverting Mux** | **Non-Inverting Mux**

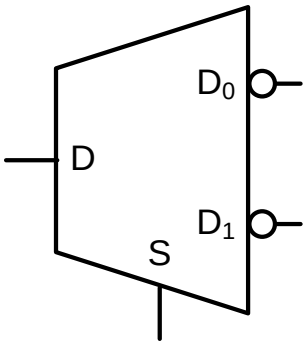
Demux Active Levels



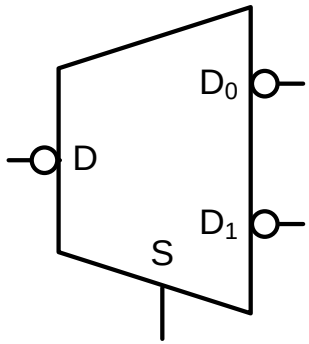
Non-Inverting Demux



Inverting Demux



Inverting Demux

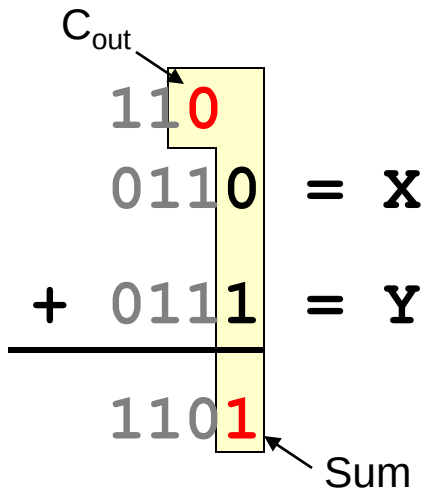


Non-Inverting Demux

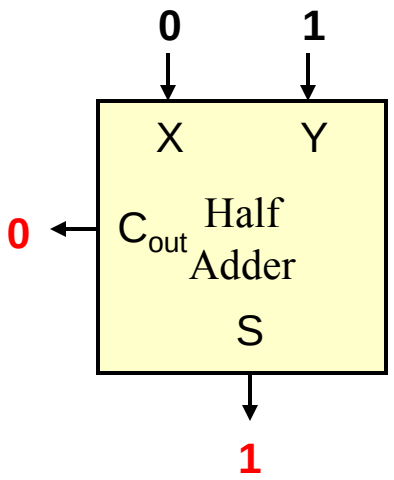
ADDERS

Addition – Half Adders

- Addition is done in columns
  - Inputs are the bit of X, Y
  - Outputs are the Sum Bit and Carry-Out (C_{out})
- Design a Half-Adder (HA) circuit that takes in X and Y and outputs S and C_{out}



X	Y	C_{out}	S
0	0	**0**	**0**
0	1	**0**	**1**
1	0	**0**	**1**
1	1	**1**	**0**



Adder Intro

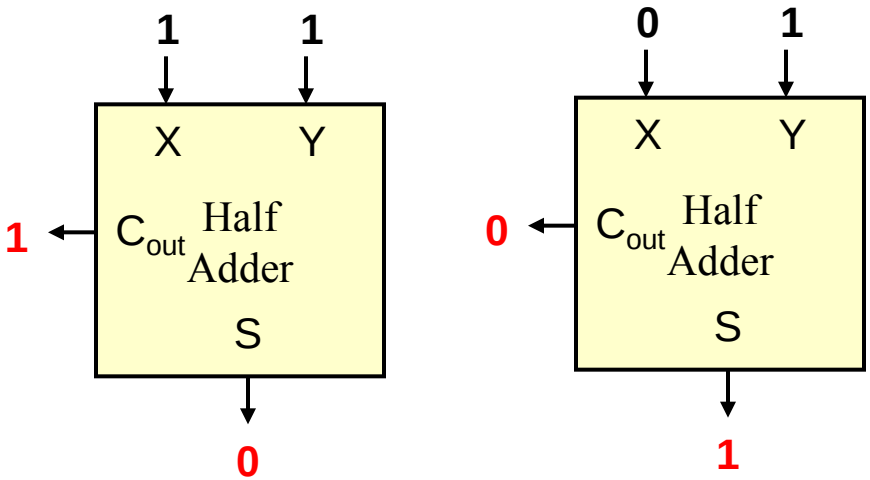
- Design a circuit to add two 4-bit numbers, X[3:0] and Y[3:0]
  - How many inputs?
  - Can we use K-Maps or Minterms or decoders + an OR gate, etc?

```
  0110 = X
+ 0111 = Y
------
  1101
```

Addition – Half Adders

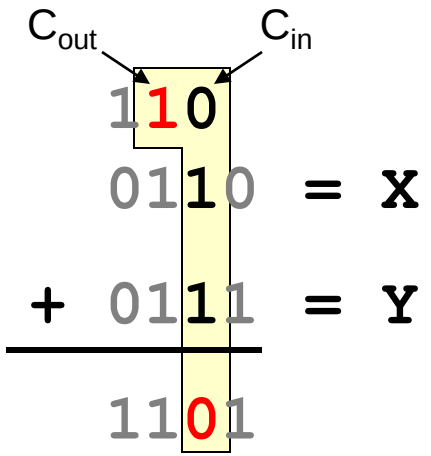
- We'd like to use one adder circuit for each column of addition
- Problem:
  - No place for Carry-out of last adder circuit
- Solution
  - Redesign adder circuit to include an input for the carry

```
  110
  0110  = X
+ 0111  = Y
-------
 1101
```

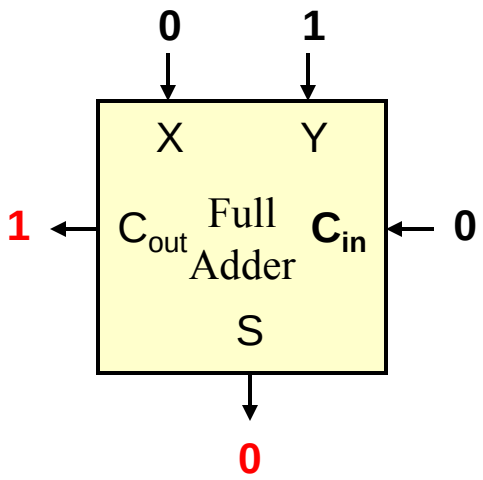


Addition – Full Adders

- Add a Carry-In input(C_{in})
- New circuit is called a Full Adder (FA)



X	Y	C_{in}	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**



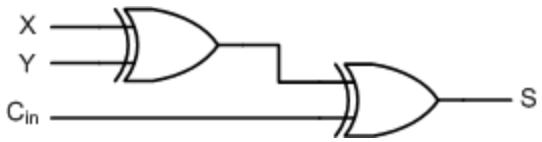
Addition – Full Adders

- Find the minimal 2-level implementations for Cout and S…

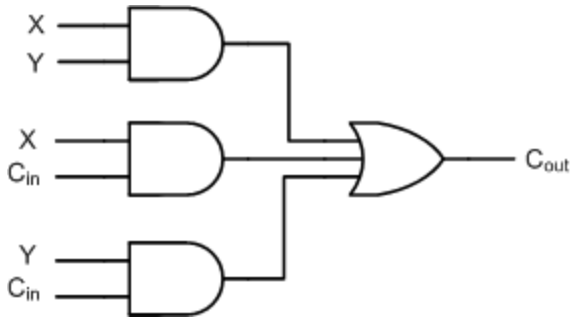
X	Y	C_{in}	C_{out}	S
0	0	0	**0**	**0**
0	0	1	**0**	**1**
0	1	0	**0**	**1**
0	1	1	**1**	**0**
1	0	0	**0**	**1**
1	0	1	**1**	**0**
1	1	0	**1**	**0**
1	1	1	**1**	**1**

Full Adder Logic

- S = X xor Y xor Cin
  - Recall: XOR is defined as true when ODD number of inputs are true…exactly when the sum bit should be 1



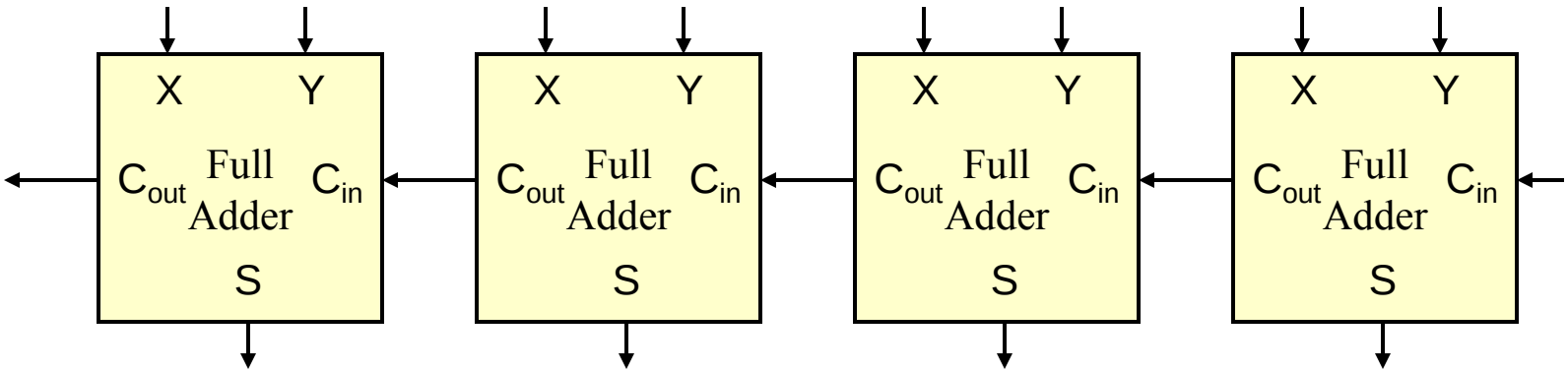
- Cout = XY + XCin + Ycin
  - Carry when sum is 2 or more (i.e. when at least 2 inputs are 1)
  - Circuit is just checking all combinations of 2 inputs



Addition – Full Adders

- Use 1 Full Adder for each column of addition

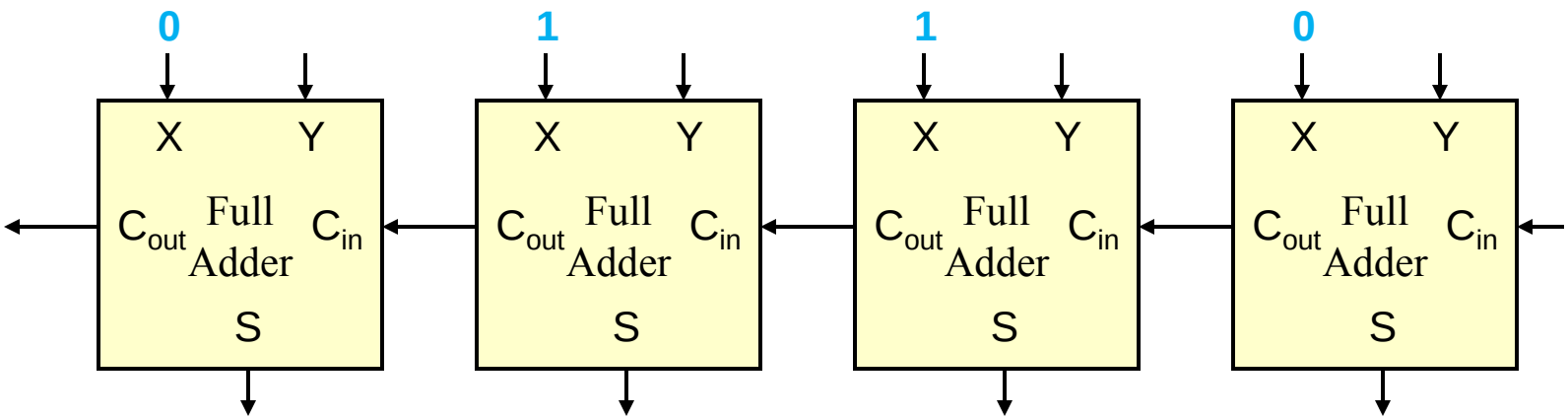
```
  0110
+ 0111
------
```



Addition – Full Adders

- Connect bits of top number to X inputs

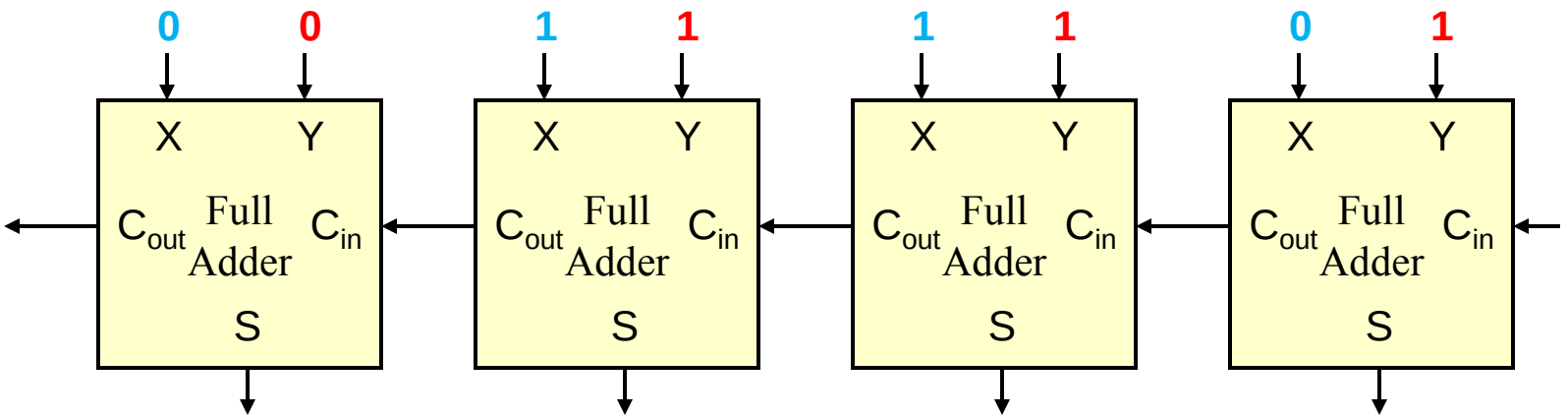
```
  0110
+ 0111
------
```



Addition – Full Adders

- Connect bits of bottom number to Y inputs

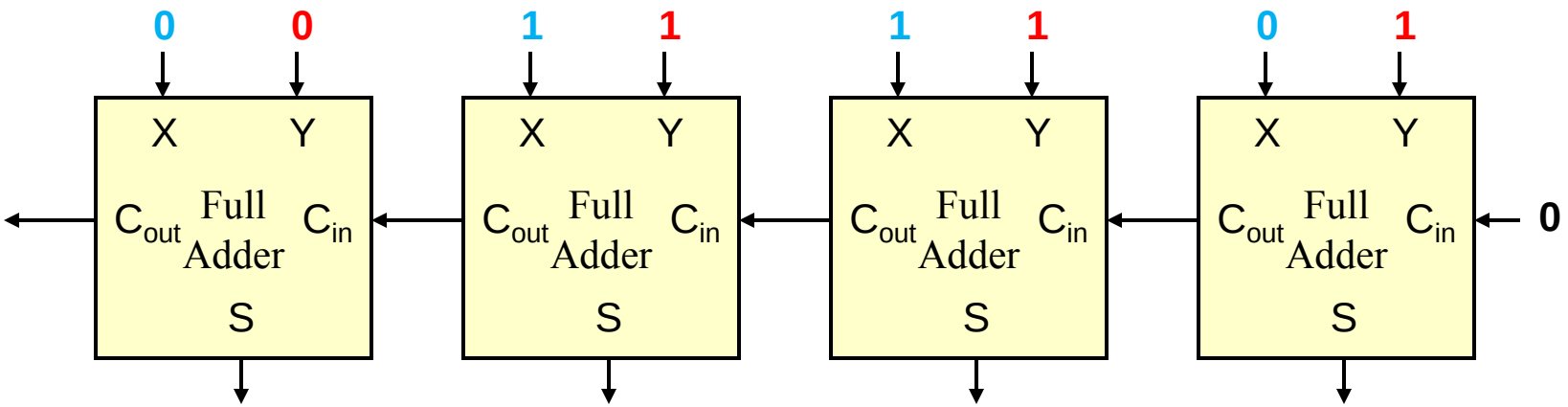
```
  0110 = X
+ 0111 = Y
```



Addition – Full Adders

- Be sure to connect first C_{in} to 0

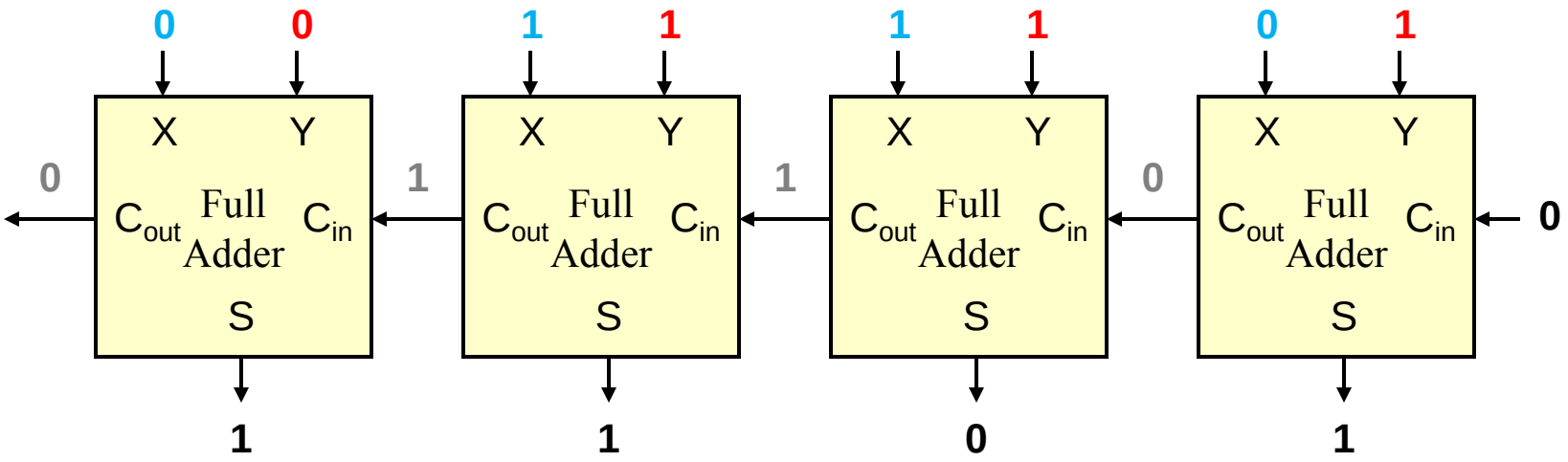
```
  0110 = X
+ 0111 = Y
```



Addition – Full Adders

- Use 1 Full Adder for each column of addition

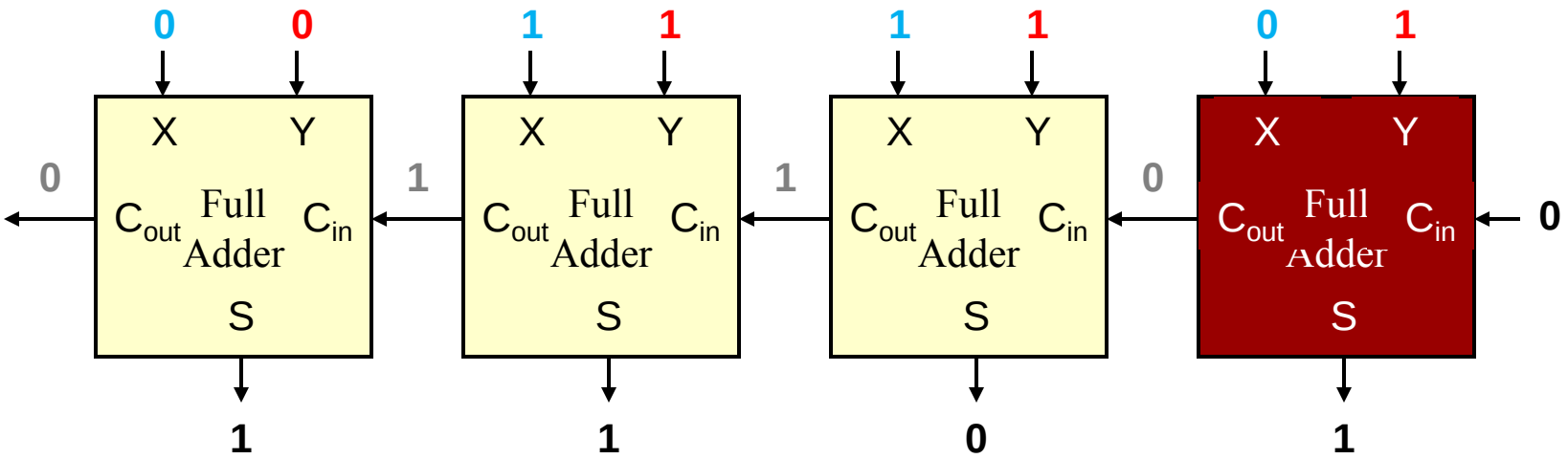
```
  01100
   0110 = X
+  0111 = Y
------
  1101
```



Addition – Full Adders

- Use 1 Full Adder for each column of addition

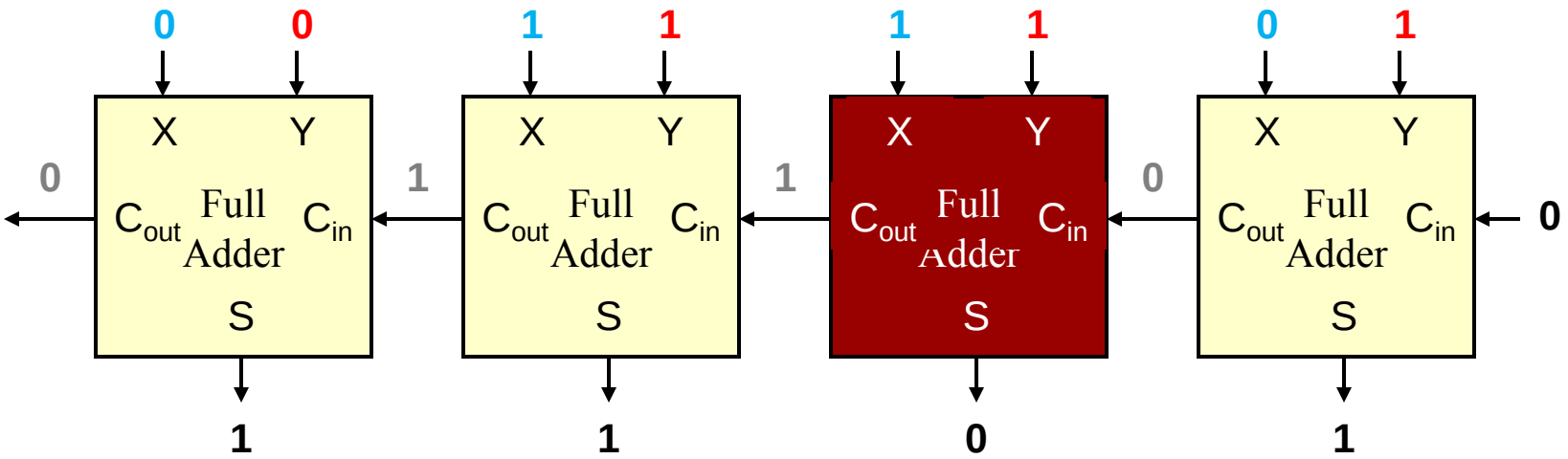
```
  01100
   0110  = X
+  0111  = Y
-------
  1101
```



Addition – Full Adders

- Use 1 Full Adder for each column of addition

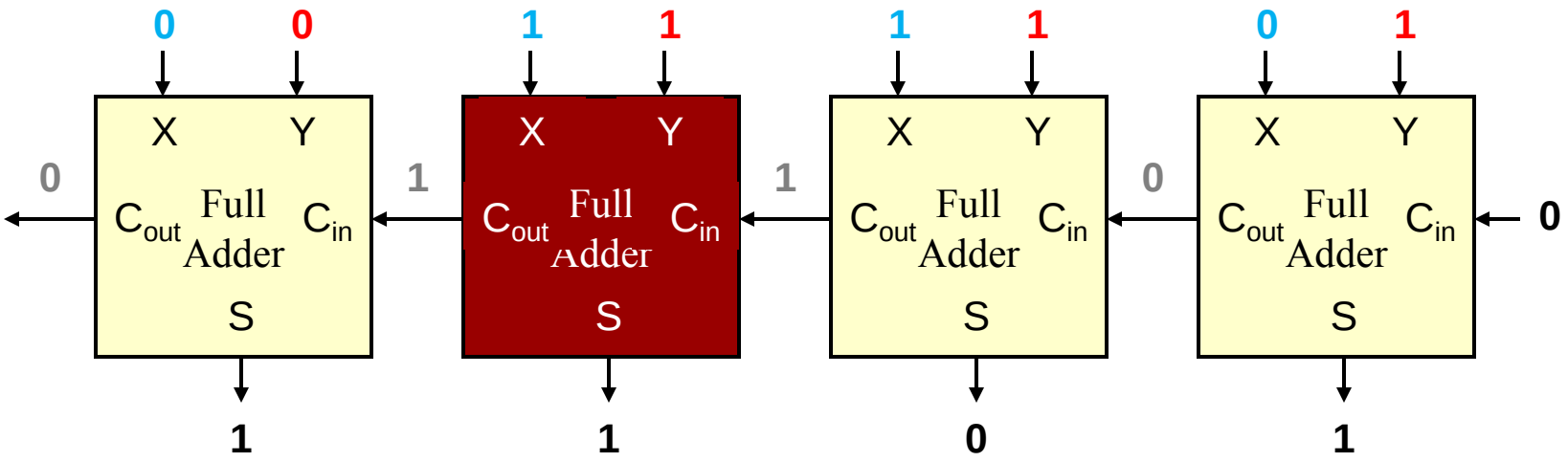
```
 01100
  0110  = X
+ 0111  = Y
------
 1101
```



Addition – Full Adders

- Use 1 Full Adder for each column of addition

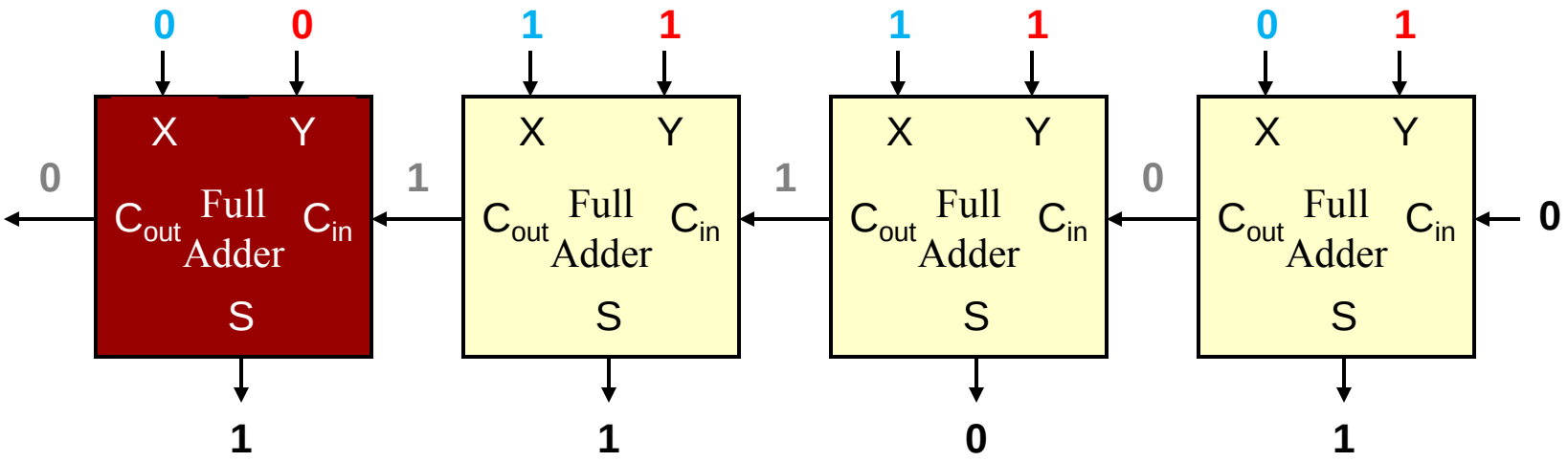
```
 01100
  0110  = X
+ 0111  = Y
-------
 1101
```



Addition – Full Adders

- Use 1 Full Adder for each column of addition

```
 01100
  0110  = X
+ 0111  = Y
------
 1101
```

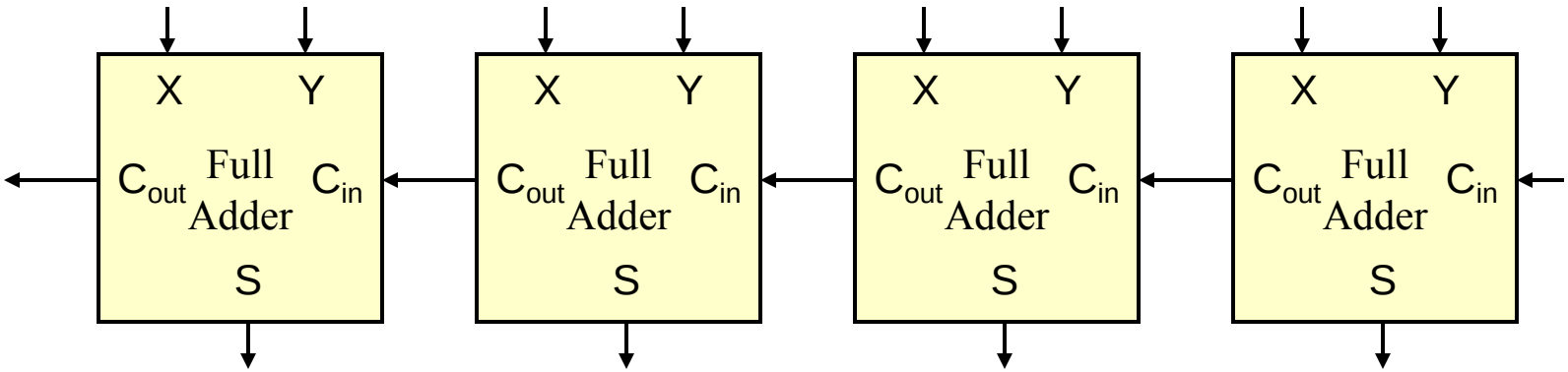


Performing Subtraction w/ Adders

- To subtract
  - Flip bits of Y
  - Add 1

```
  0101 = X
- 0011 = Y
------
  0010
```

```
  0101
+ 1100
     1
------
  0010
```

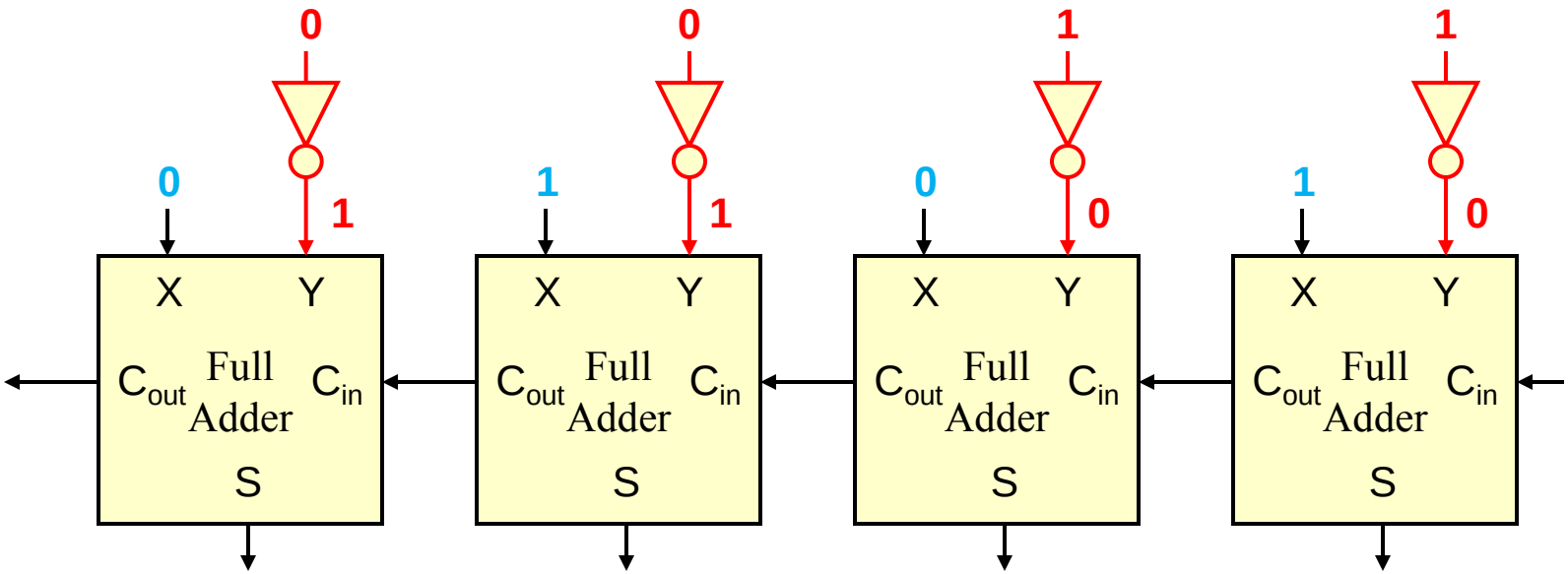


Performing Subtraction w/ Adders

- To subtract
  - Flip bits of Y
  - Add 1

```
  0101 = X
- 0011 = Y
--------
  0010
```

```
  0101
+ 1100
     1
--------
  0010
```

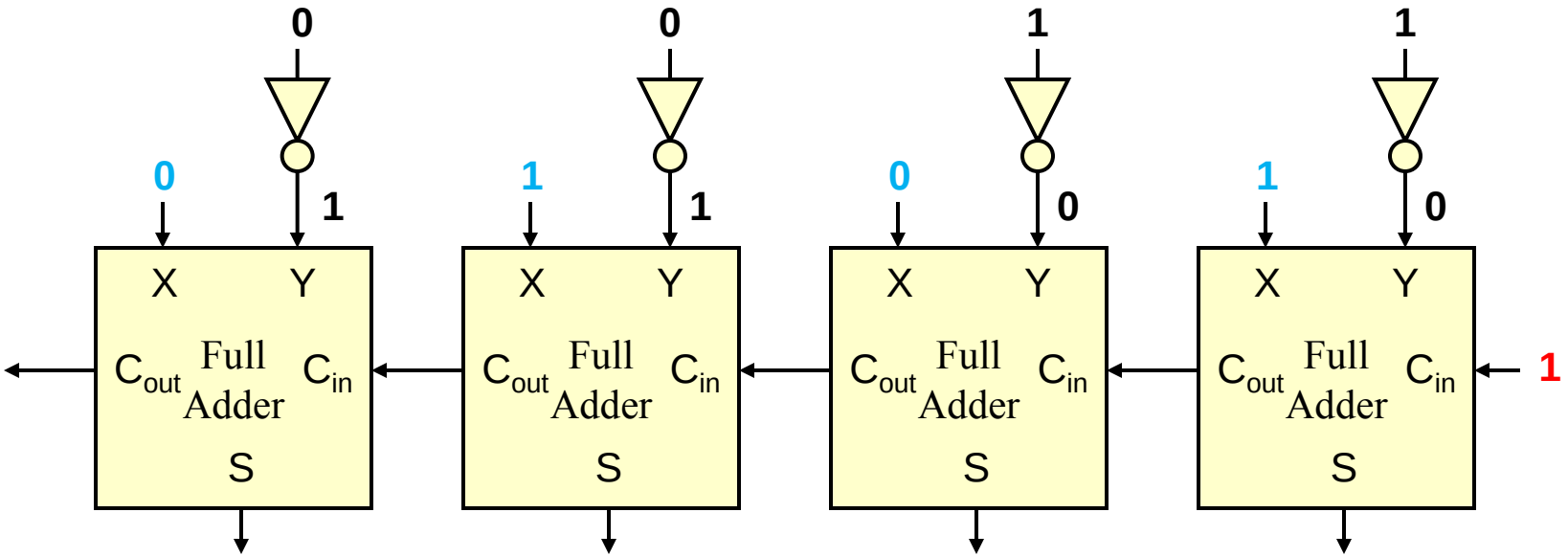


Performing Subtraction w/ Adders

- To subtract
  - Flip bits of Y
  - Add 1

```
  0101 = X
- 0011 = Y
--------
  0010
```

```
  0101
+ 1100
     1
--------
  0010
```

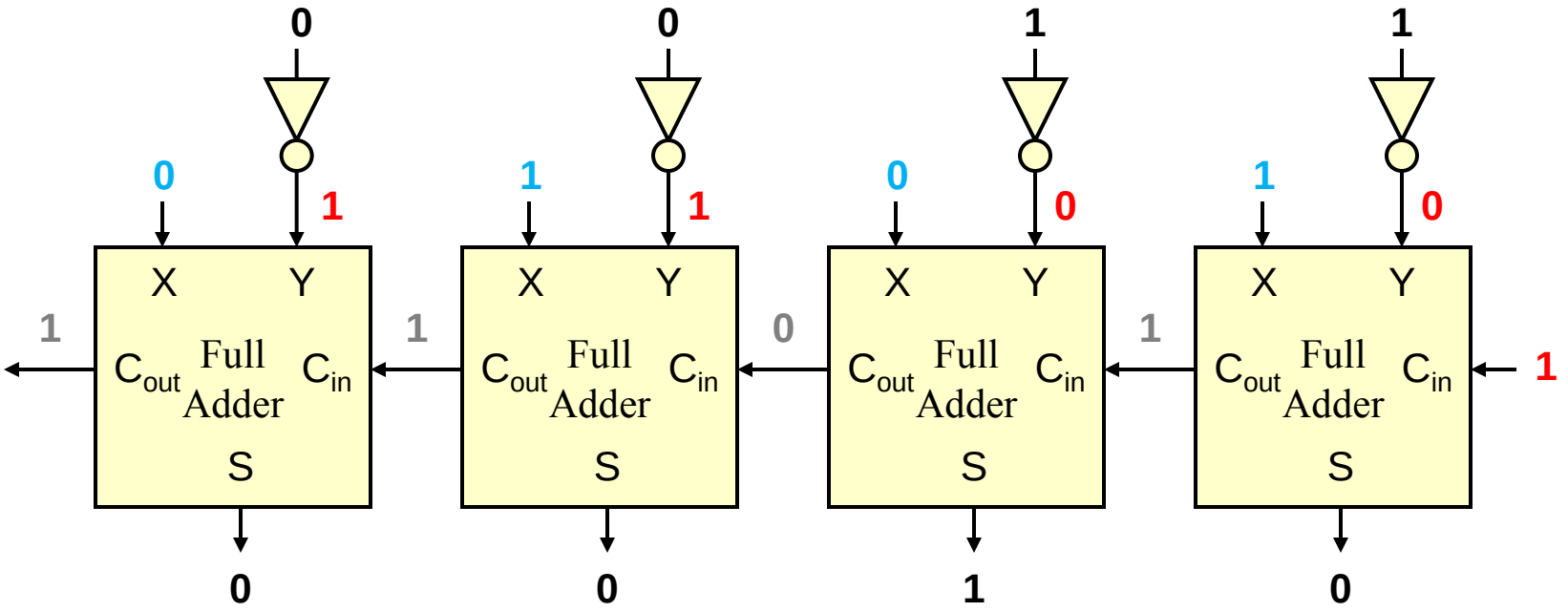


Performing Subtraction w/ Adders

- To subtract
  - Flip bits of Y
  - Add 1

```
  0101 = X
- 0011 = Y
--------
  0010
```

```
  0101
+ 1100
     1
--------
  0010
```



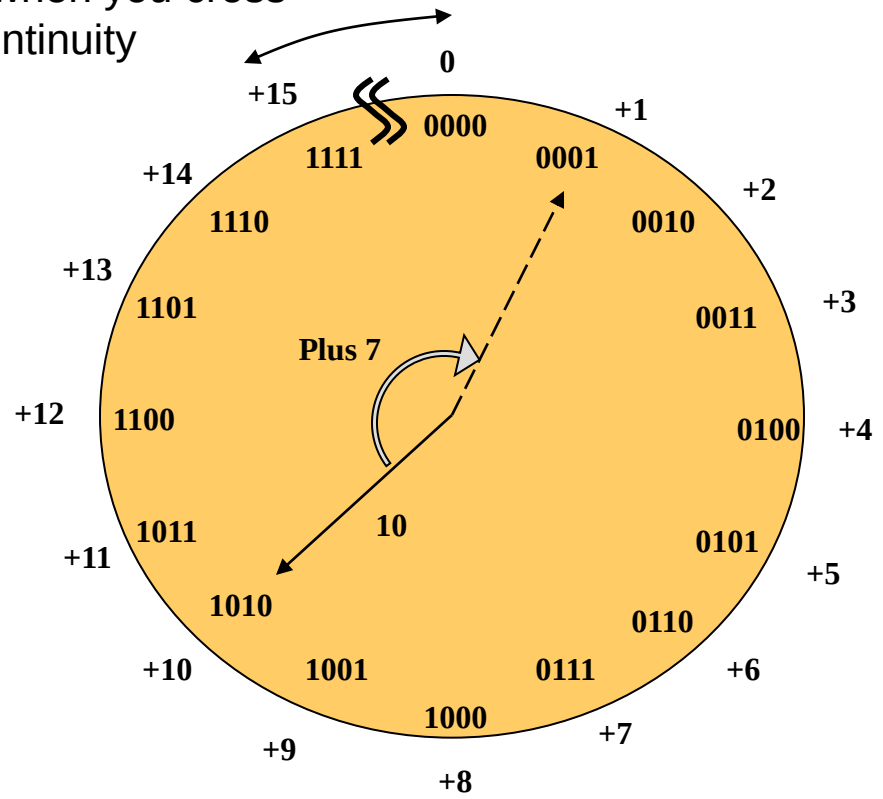
OVERFLOW

Overflow

- Overflow occurs when the result of an arithmetic operation is too large to be represented with the given number of bits
  - Unsigned overflow occurs when adding or subtracting unsigned numbers
  - Signed (2's complement overflow) overflow occurs when adding or subtracting 2's complement numbers

Unsigned Overflow

Overflow occurs when you cross this discontinuity



10 + 7 = 17

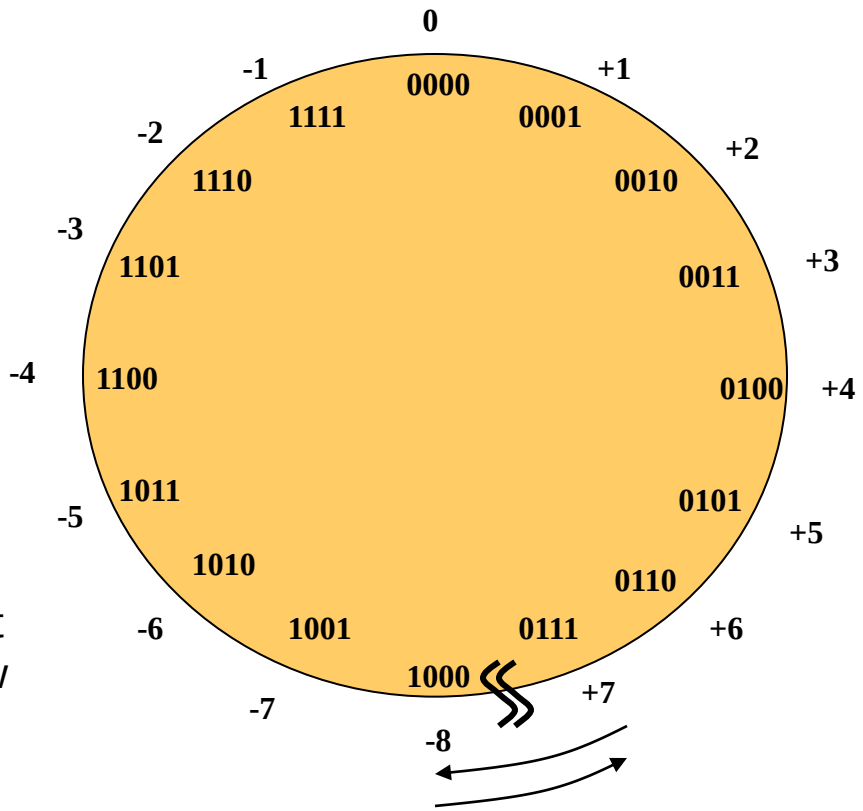
With 4-bit *unsigned* numbers we can only represent 0 – 15. Thus, we say overflow has occurred.

2's Complement Overflow

5 + 7 = +12

-6 + -4 = -10

With 4-bit *2's complement* numbers we can only represent -8 to +7. Thus, we say overflow has occurred.



Overflow occurs when you cross this discontinuity

Testing for Overflow

- Most fundamental test
  - Check if answer is wrong (i.e. Positive + Positive yields a negative)
- Unsigned overflow test
  - If carry-out of final position equals ‘1’
- Signed (2’s complement) overflow test
  - Only occurs if two positives are added and result is negative or two negatives are added and result is positive
  - Alternate test: if carry-in and carry-out of final position are different

Testing for Unsigned Overflow

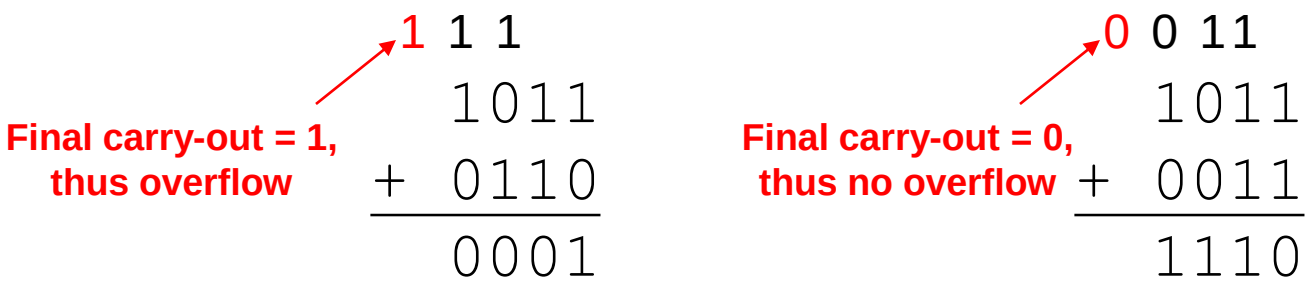
- Unsigned Overflow test
  - Unsigned Addition: If final carry-out = 1

```
  1011
+ 0110
------
```

```
  1011
+ 0011
------
```

Testing for Unsigned Overflow

- Unsigned Overflow test
  - Unsigned Addition: If final carry-out = 1



Testing for 2's Comp. Overflow

- 2's Complement Overflow Occurs If…
  - Test 1: If pos. + pos. = neg. or neg. + neg. = pos.
  - Test 2: If carry-in to MSB position and carry-out of MSB position are different

```
  0101  (5)          1100 (-4)
+ 0110  (6)        + 1001 (-7)

  0011  (3)          1110 (-2)
+ 0010  (2)        + 1010 (-6)
```

Testing for 2's Comp. Overflow

- 2's Complement Overflow Occurs If…
  - Test 1: If pos. + pos. = neg. or neg. + neg. = pos.
  - Test 2: If carry-in to MSB position and carry-out of MSB position are different

Carry-in to MSB and carry-out of MSB position are *different*…Overflow!

```
 0 1
  0101  (5)
+ 0110  (6)
  1011 (-5)
```

Carry-in to MSB and carry-out of MSB position are *different*…Overflow!

```
 1 0
  1100 (-4)
+ 1001 (-7)
  0101 (+5)
```

Carry-in to MSB and carry-out of MSB position are *same*…No Overflow!

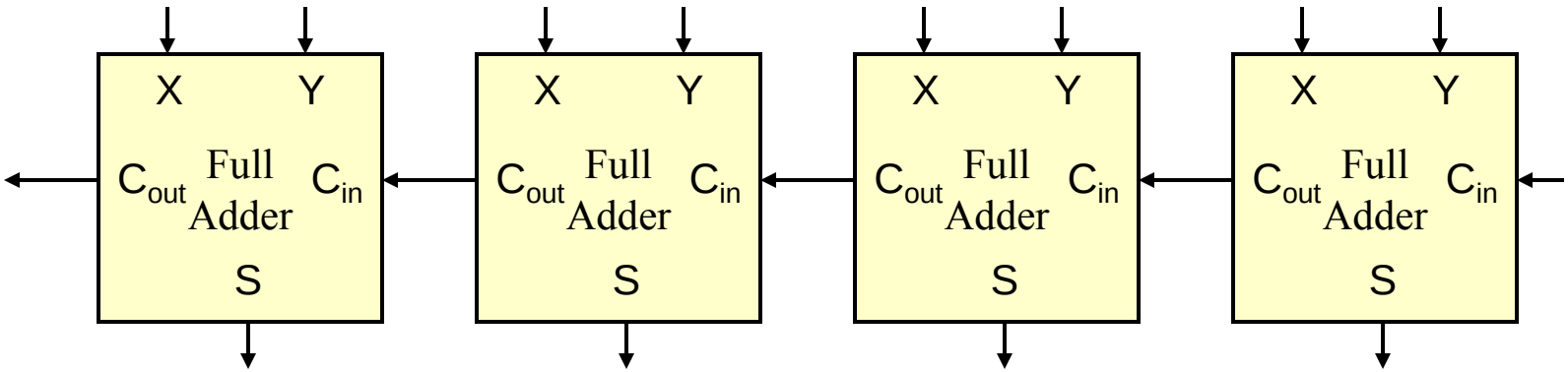
```
 0 0
  0011  (3)
+ 0010  (2)
  0101  (5)
```

Carry-in to MSB and carry-out of MSB position are *same*…No Overflow!

```
 1 1
  1110 (-2)
+ 1010 (-6)
  1000 (-8)
```

Checking for Overflow

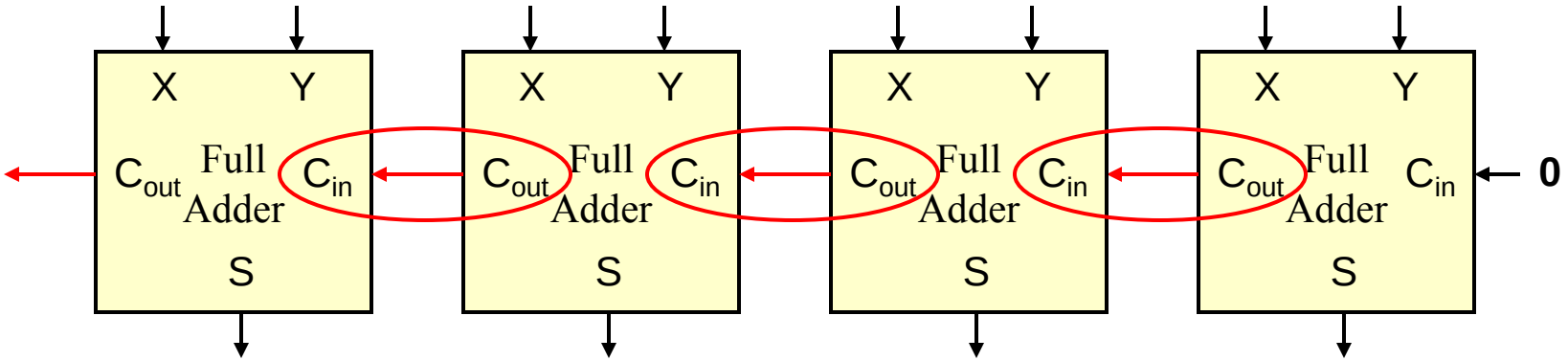
- Produce additional outputs to indicate if unsigned (UOV) or signed (SOV) overflow has occurred



ADDER DELAY

Timing

- A chain of full adders presents an interesting timing analysis problem
- To correctly compute its own Sum and Carry-out, each full adder requires the carry-out bit from the previous full adder
- Because hardware works in parallel, the full adders further down the chain may momentarily produce the wrong outputs because the carry has not had time to propagate to them



Timing Example

- Assume that we were adding one set of inputs and then change to a new set of inputs:

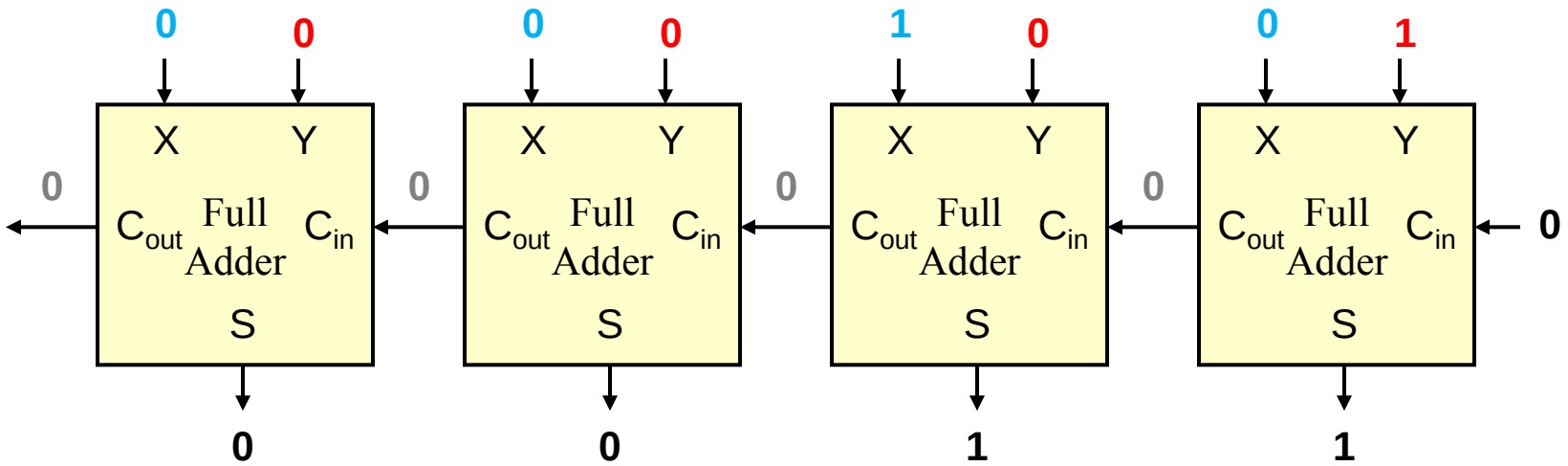
Old inputs:

```
  0000
  0010   = X
+ 0001   = Y
------
  0011
```

New inputs:

```
  1111
  1111   = X
+ 0001   = Y
------
  0000
```

Old inputs:



Timing

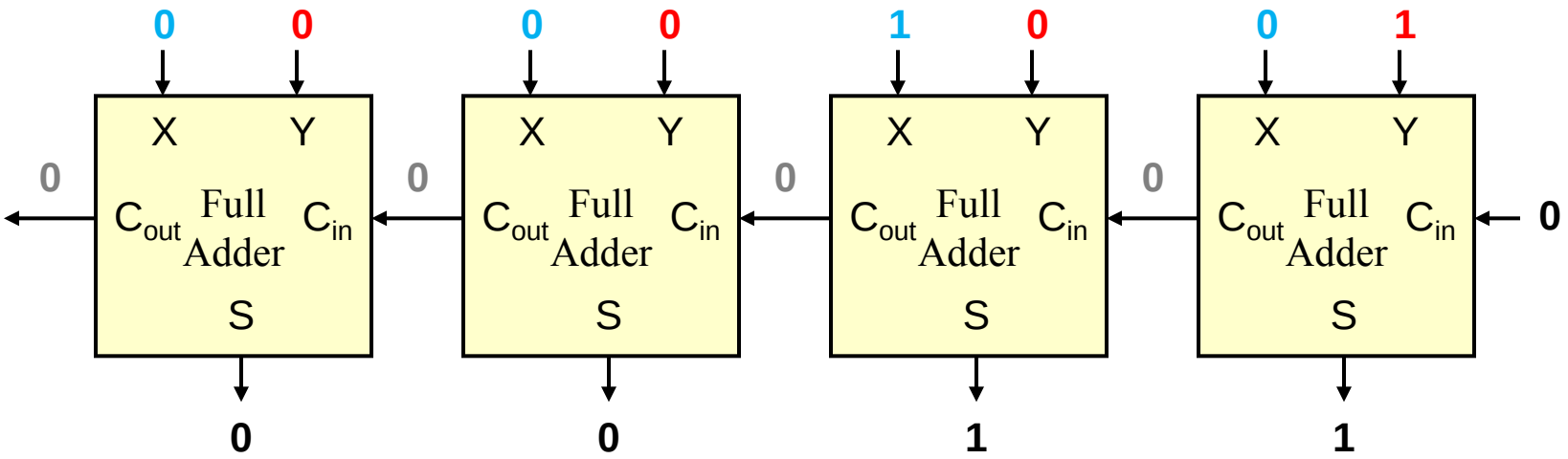
- At the time just before we enter the new input values, all carries are 0's

New inputs:

```
  0000
  0010   = X
+ 0001   = Y
------
  0011
```

Time
-1

Old inputs:

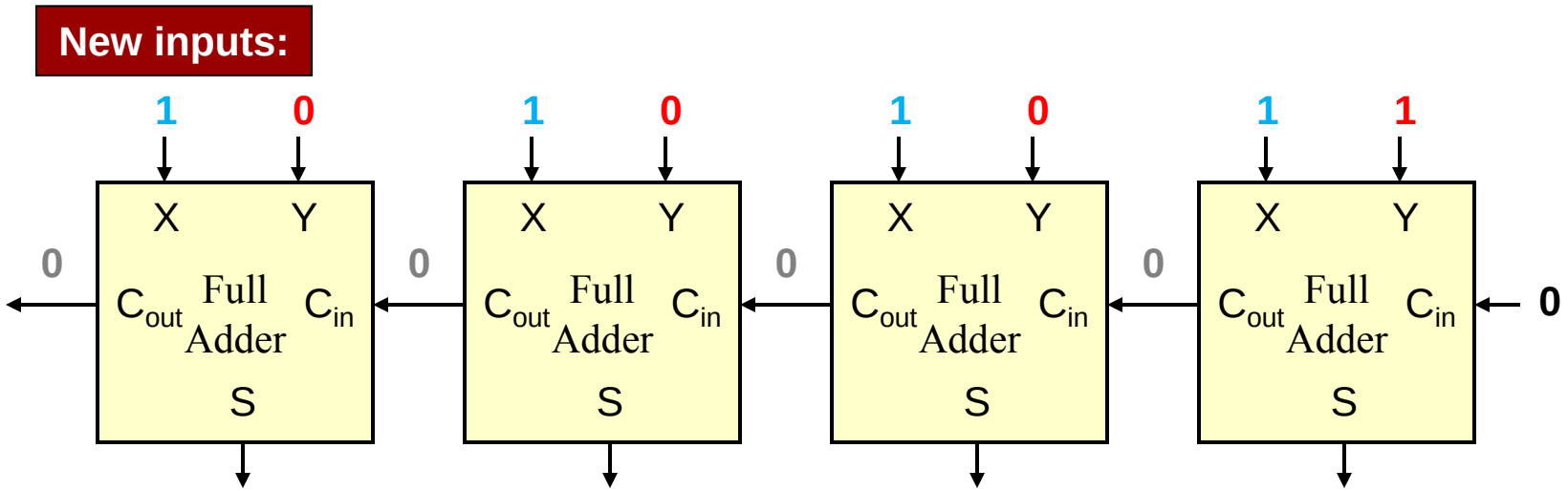


Timing

- Now we enter the new inputs and all the FA's starting adding their respective inputs

```
  1111
   1111 = X
+  0001 = Y
------
  0000
```

Time
0



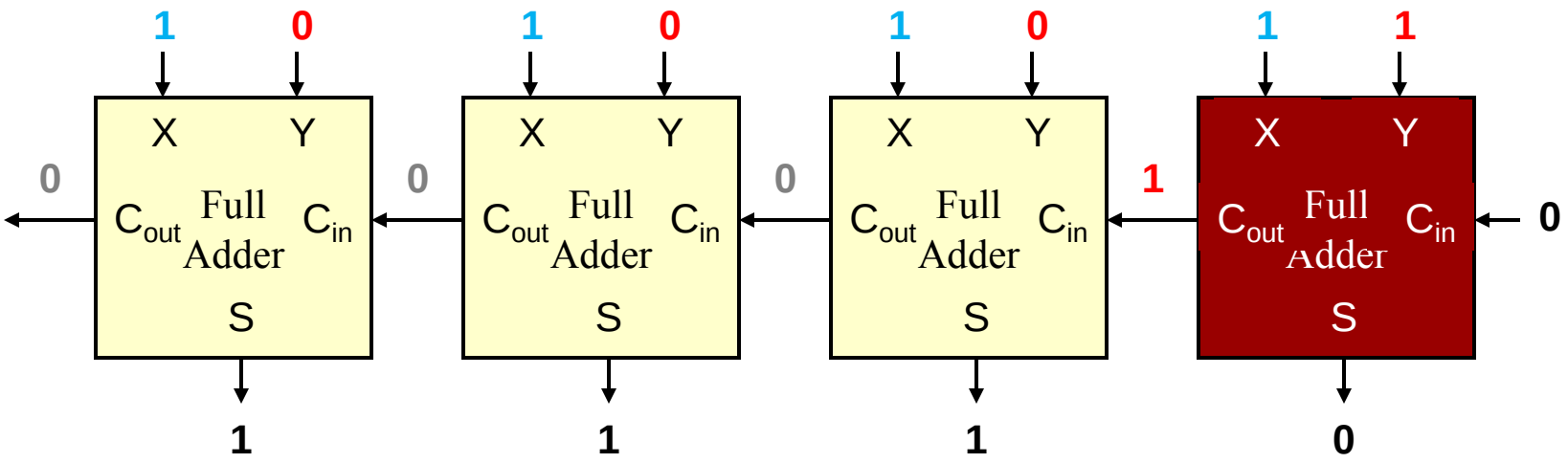
Due to propagation delay, the carries are still from the old inputs

Timing

- Each adder computes from the current inputs (notice the sum of 1110 is incorrect at this point)

```
  1111
  1111 = X
+ 0001 = Y
------
  0000
```

Time
1



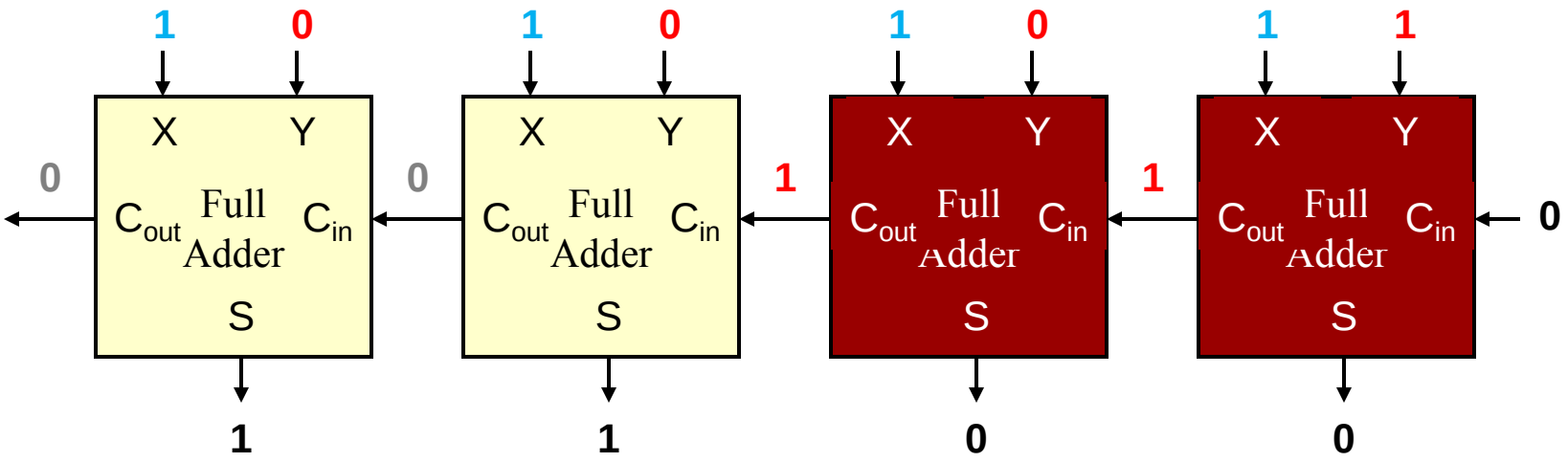
Now the carries are all based off the new inputs

Timing

- The carry is "rippling" through each adder

```
  1111
   1111 = X
+  0001 = Y
------
  0000
```

Time
2

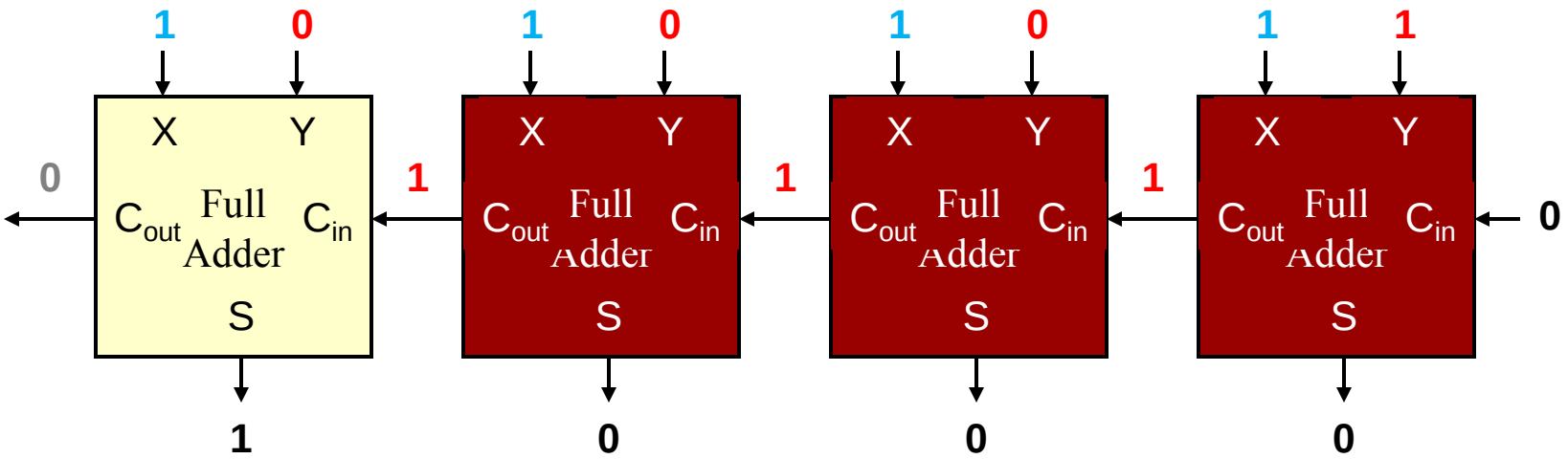


Timing

- The carry is "rippling" through each adder

```
  1111
   1111 = X
+  0001 = Y
------
  0000
```

Time
3

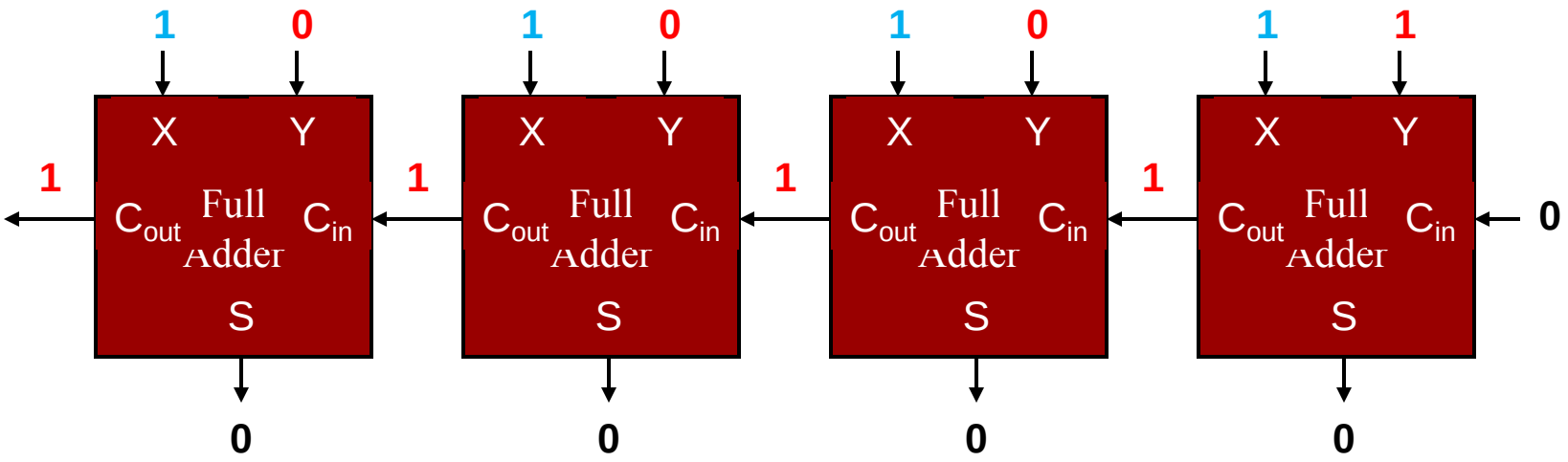


Timing

- Only after the carry propagates through all the adders is the sum valid and correct

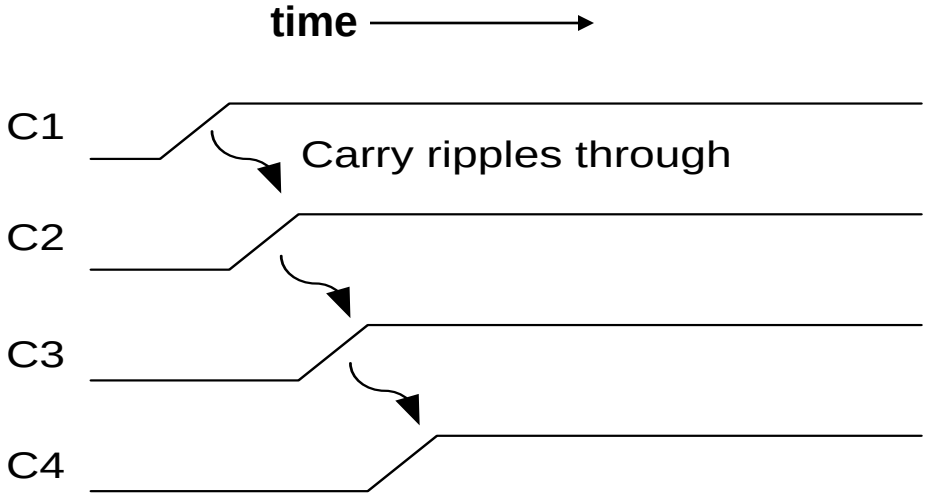
```
  1111
   1111 = X
+  0001 = Y
-------
  0000
```

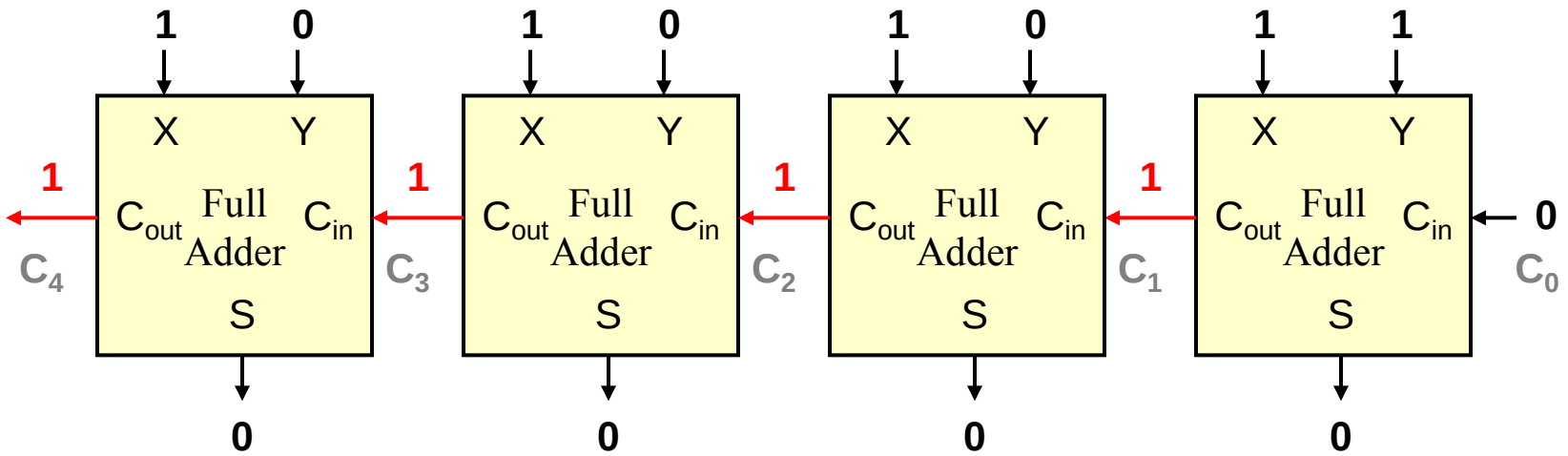
Time
4



"Ripple-Carry" Adder

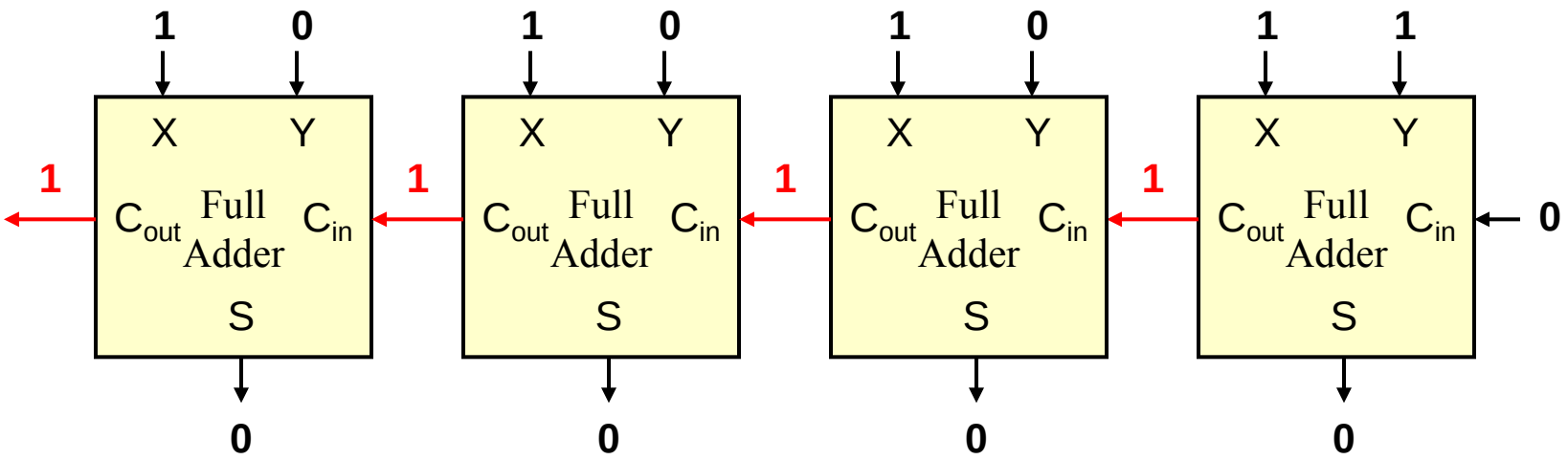
- The longest path through a chain of full adders is the carry path
- We say that the carry "ripples" through the adder





Ripple Carry Adder Delay

- An n-bit ripple carry adder has a worst case delay proportional to n (i.e. n-bits => n columns of addition => n-full adders)

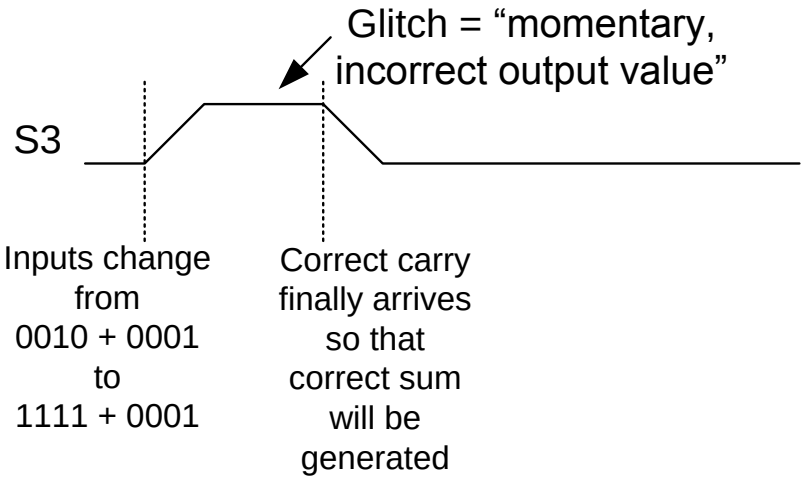


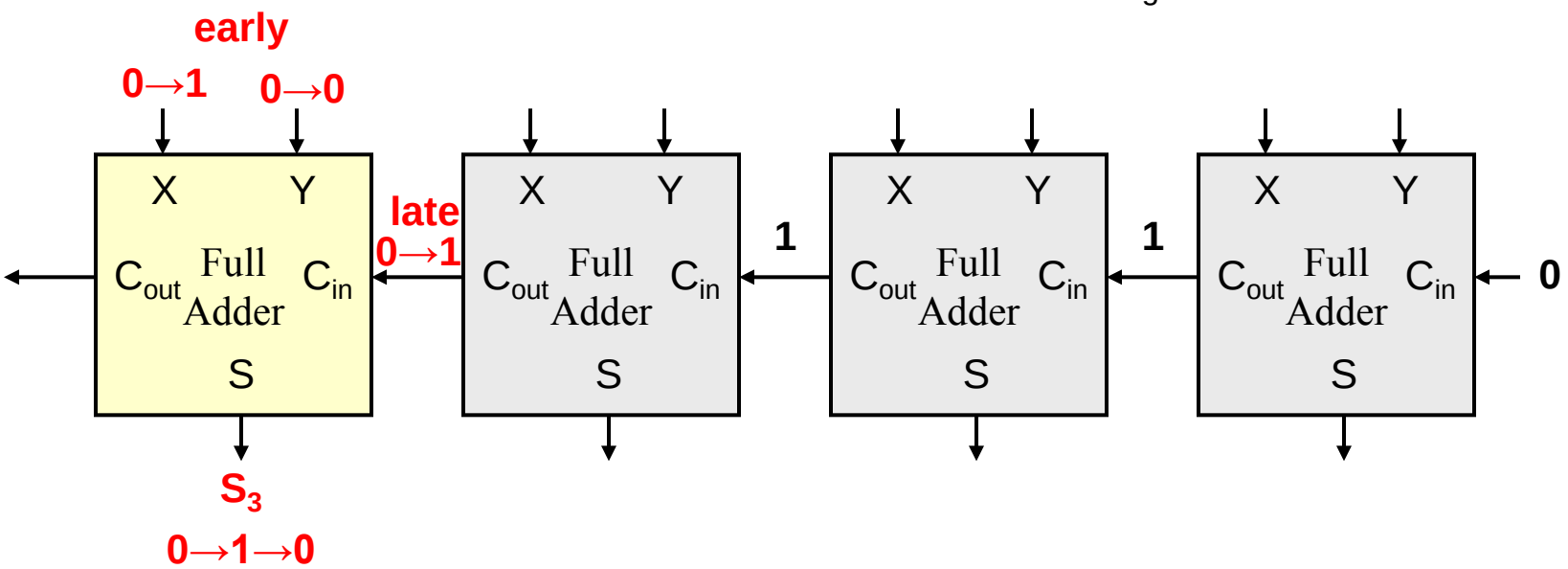
Glitches

- Transient, incorrect output values due to differing arrival times of gate inputs

Output Glitches

- Delay of the carry causes glitches on the sum bits
- Glitch = momentarily, incorrect output value



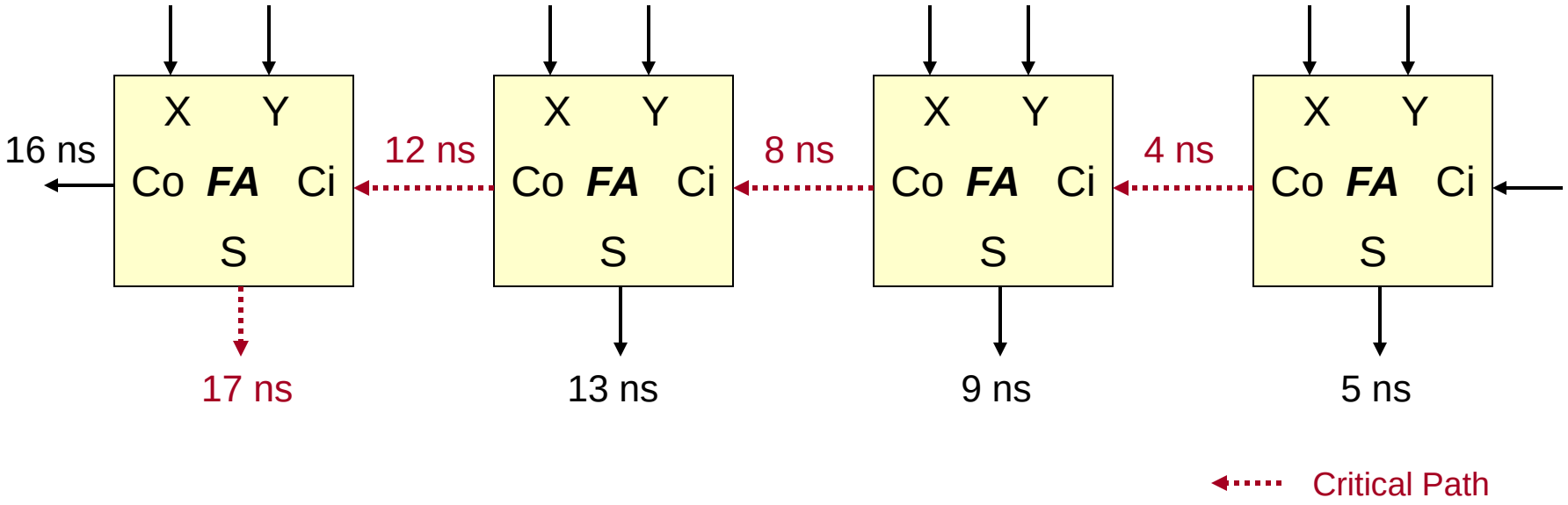


Critical Path

- Critical Path = Longest possible delay path

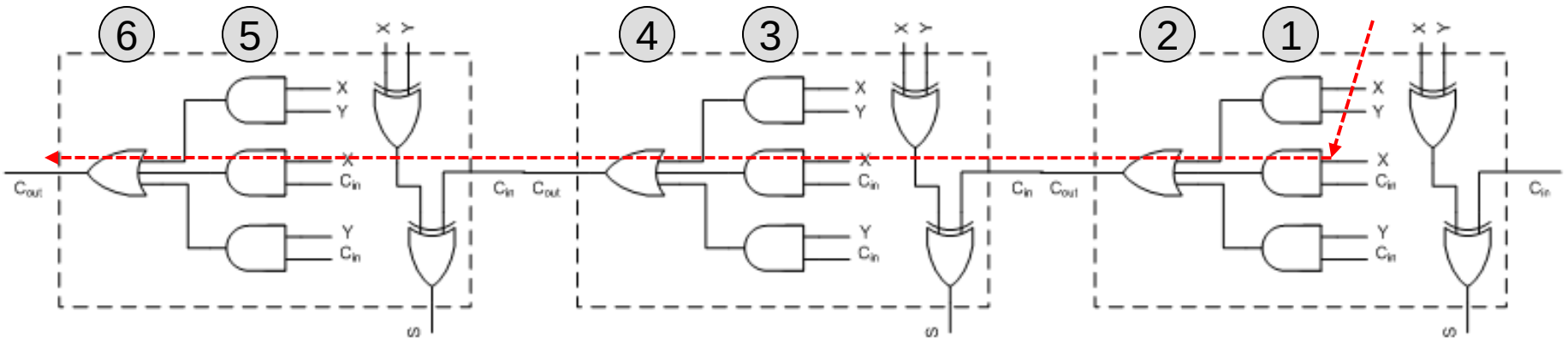
Assume t_{sum} = 5 ns,

t_{carry}= 4 ns



Ripple Carry Adders

- Ripple-carry adders (RCA) are slow due to carry propagation
  - At least 2 levels of logic per full adder



Fast Adders

- Rather than calculating one carry at a time and passing it down the chain, can we compute a group of carries at the same time
- To do this, let us define some new signals for each column of addition:
  - p_i = Propagate: This column will propagate a carry-in (if there is one) to the carry-out.
    p_i is true when A_i or B_i is 1 => $p_i = A_i + B_i$
  - g_i = Generate: This column will generate a carry-out whether or not the carry-in is '1'
    g_i is true when A_i and B_i is 1 => $g_i = A_i \bullet B_i$
- Using these signals, we can define the carry-out (c_{i+1}) as:

$$c_{i+1} = g_i + p_i c_i$$

Carry Lookahead Logic

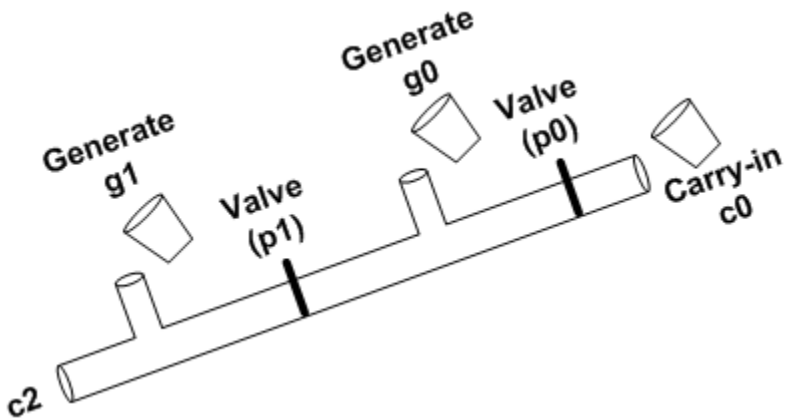
- Define each carry in terms of p_i, g_i and the initial carry-in (c_0) and not in terms of carry chain (intermediate carries: c1,c2,c3,…)
- c1 =
- c2 =
- c3 =
- c4 =

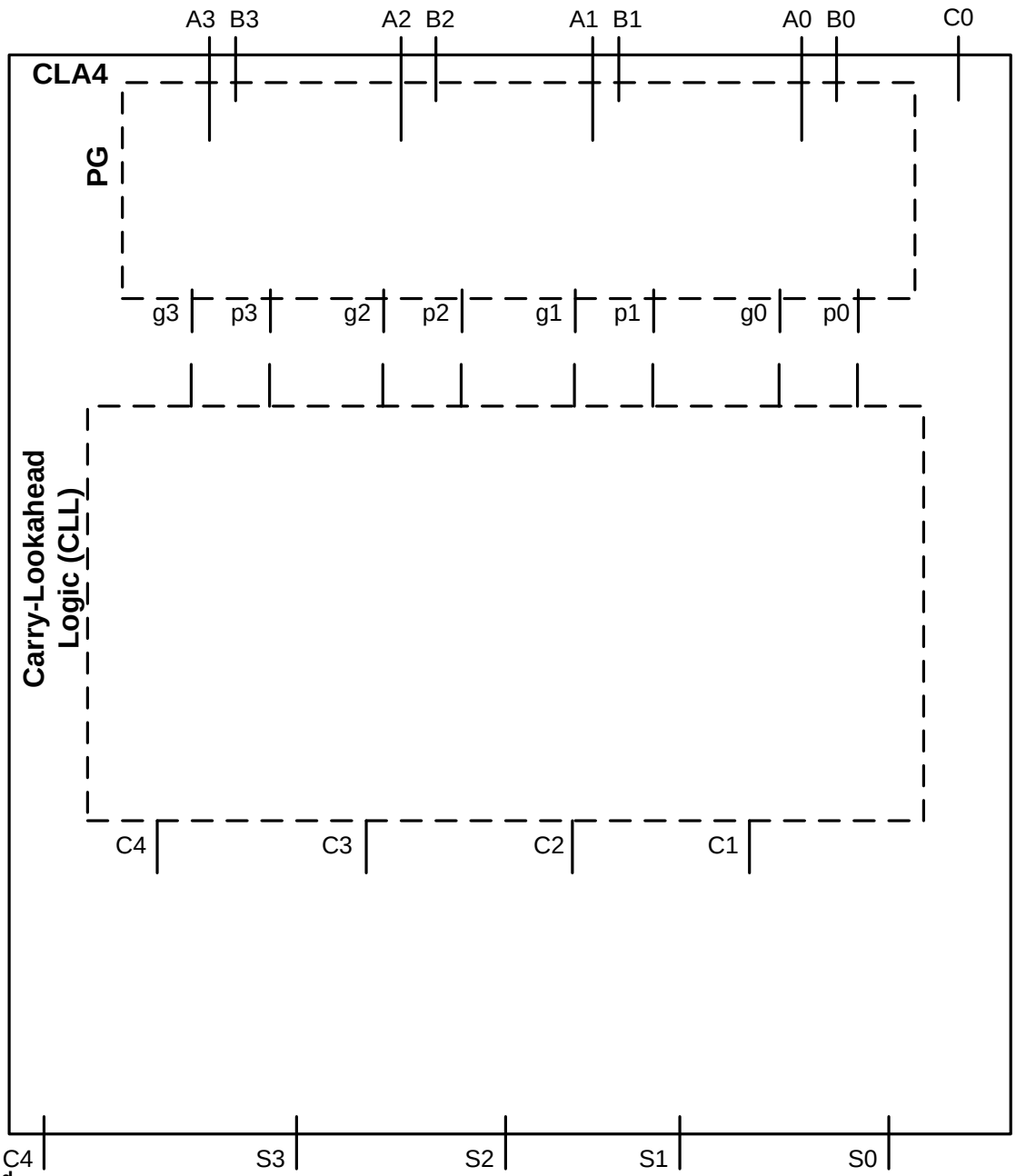
Carry Lookahead Logic

- Define each carry in terms of p_i, g_i and the initial carry-in (c_0) and not in terms of carry chain (intermediate carries: c1,c2,c3,…)
- $c1 = g_0 + p_0c_0$
- $c2 = g_1 + p_1c_1 = g_1 + p_1g_0 + p_1p_0c_0$
- c3 = …
- c4 = …

Carry Lookahead Analogy

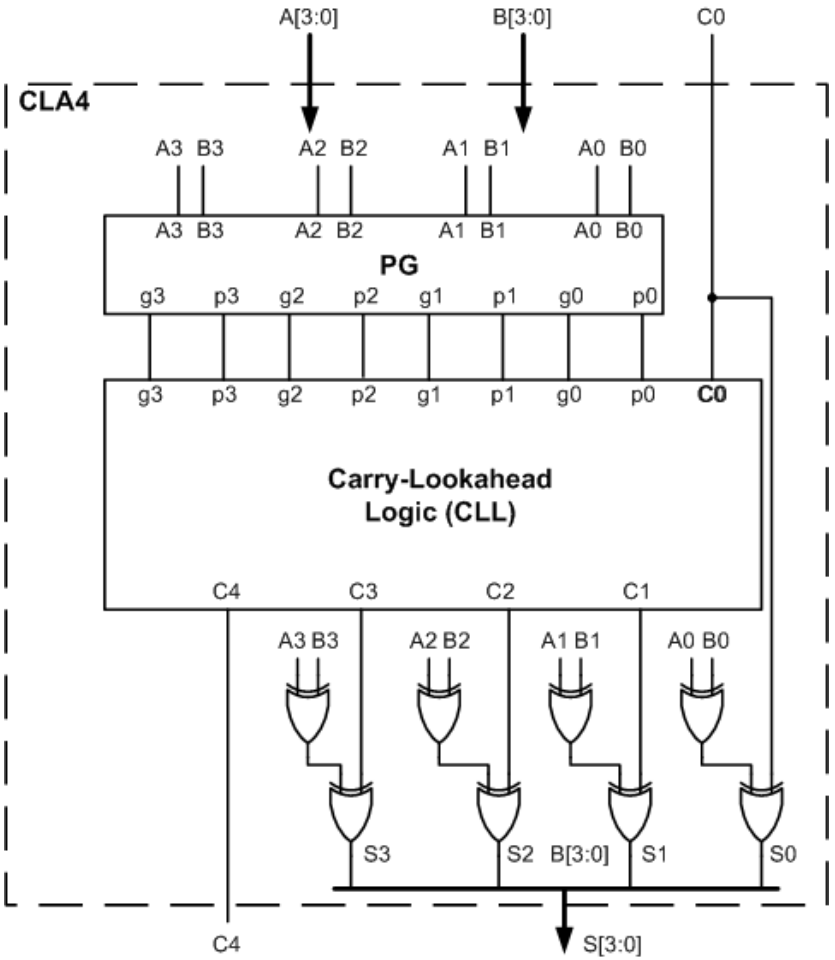
- Consider the carry-chain like a long tube broken into segments. Each segment is controlled by a valve (propagate signal) and can insert a fluid into that segment (generate signal)
- The carry-out of the diagram below will be true if g1 is true or p1 is true and g0 is true, or p1, p0 and c1 is true





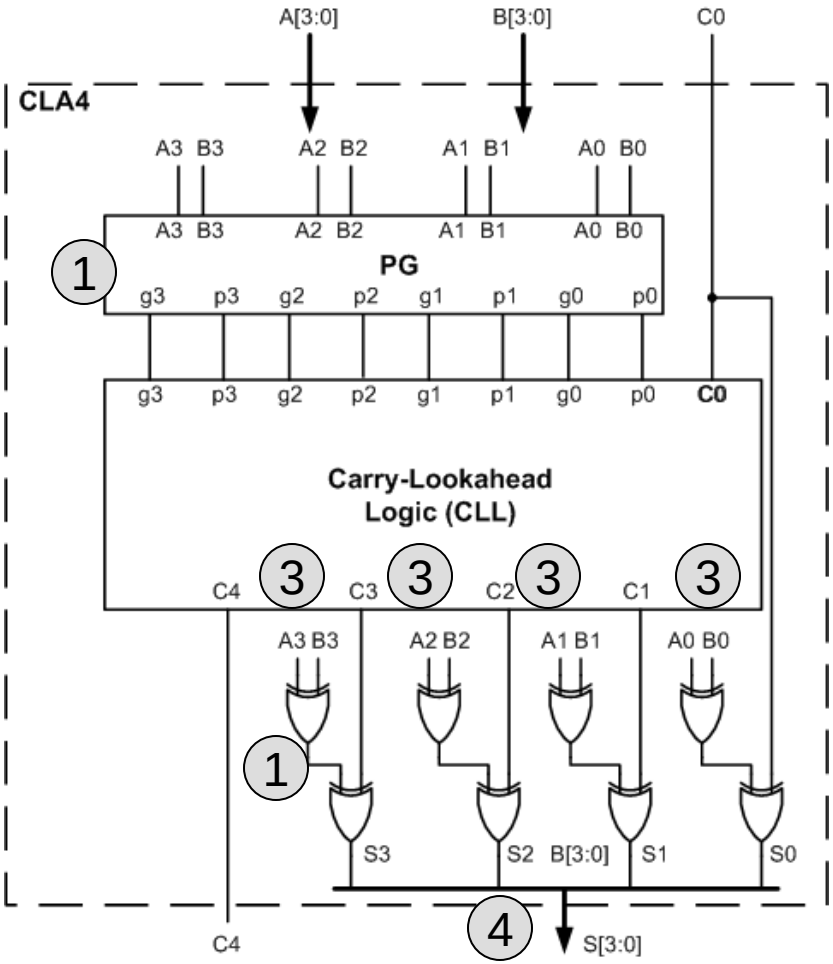
Carry Lookahead Adder

- Use carry-lookahead logic to generate all the carries in one shot and then create the sum
- Example 4-bit CLA shown below



Carry Lookahead Adder

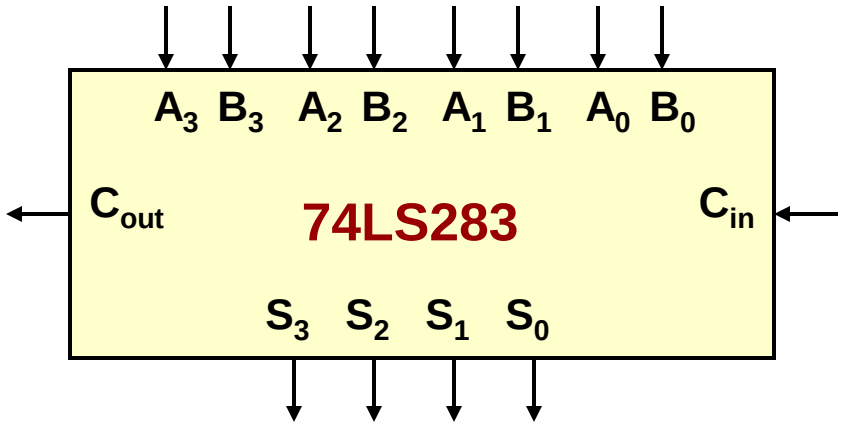
- Use carry-lookahead logic to generate all the carries in one shot and then create the sum
- Example 4-bit CLA shown below



4-bit Adders

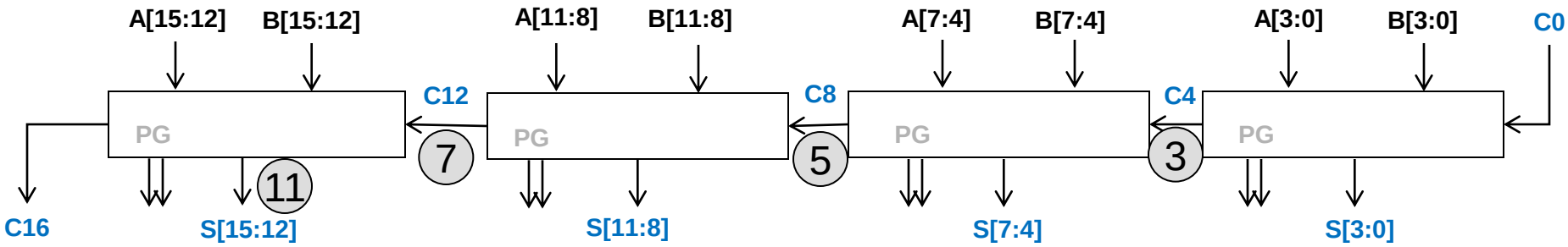
- 74LS283 chip implements a 4-bit adder using CLA methodology

$$\begin{array}{rl} A_3A_2A_1A_0 & = A \\ + \; B_3B_2B_1B_0 & = B \\ \hline S_4S_3S_2S_1S_0 & = S \end{array}$$



16-Bit CLA

- At this point we should probably stop as we have a 5-input gate in our equation



16-bit RCA Delay = 16*2 = 32 gate delays

Delay of the above adder design = 3+2+2+4 = 11 gates

Let us improve by looking ahead at a higher level to produce C16, C12, C8, C4 in parallel

What's the difference between the equation for G here and C4 on the previous slides

Define P and G as the overall Propagate and Generate signals for a set of 4 bits

$P = p3 \bullet p2 \bullet p1 \bullet p0$

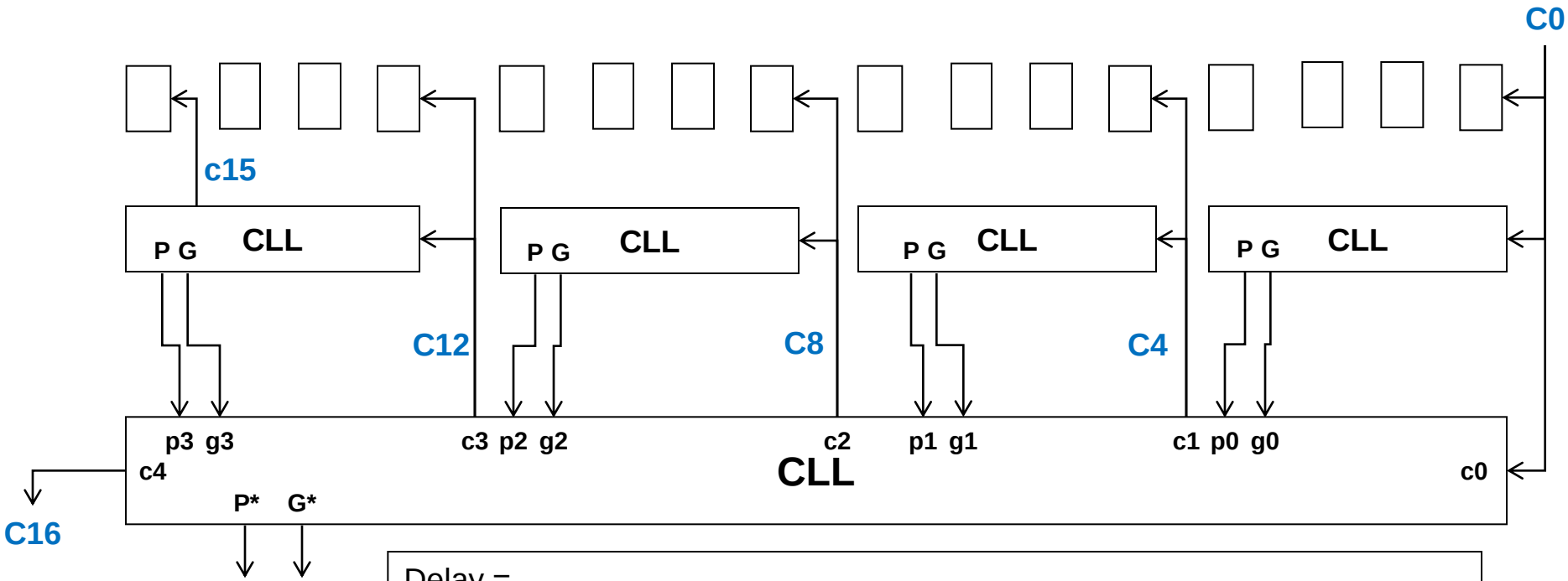
$G = g3 + p3 \bullet g2 + p3 \bullet p2 \bullet g1 + p3 \bullet p2 \bullet p1 \bullet g0$

16-bit CLA Closer Look

- Each 4-bit CLA only propagates its overall carry-in if each of the 4 columns propagates:
  - P0 = p3• p2 •p1 •p0
  - P1 = p7• p6 •p5 •p4
  - P2 = p11• p10 •p9 •p8
  - P3 = p15• p14 •p13 •p12
- Each 4-bit CLA generates a carry if any column generates and the more significant columns propagate
  - G0 = g3 + (p3 •g2) + (p3 •p2 •g1)+(p3 •p2 •p1 •g0)
  - …
  - G3 = g15 + (p15 •g14) + (p15 •p14 •g13)+(p15 •p14 •p13 •g12)
- The higher order CLL logic (producing C4,C8,C12,C16) then is realized as:
  - (C4) =>C1 = G0 + (P0 •c0)
  - …
  - (C16) => C4 = G3 + (P3 •G2) + (P3 •P2 •G1) +(P3 • P2 • P1 • G0)+ (P3 •P2 •P1 •P0 •c0)
- These equations are exactly the same CLL logic we derived earlier

16-Bit CLA

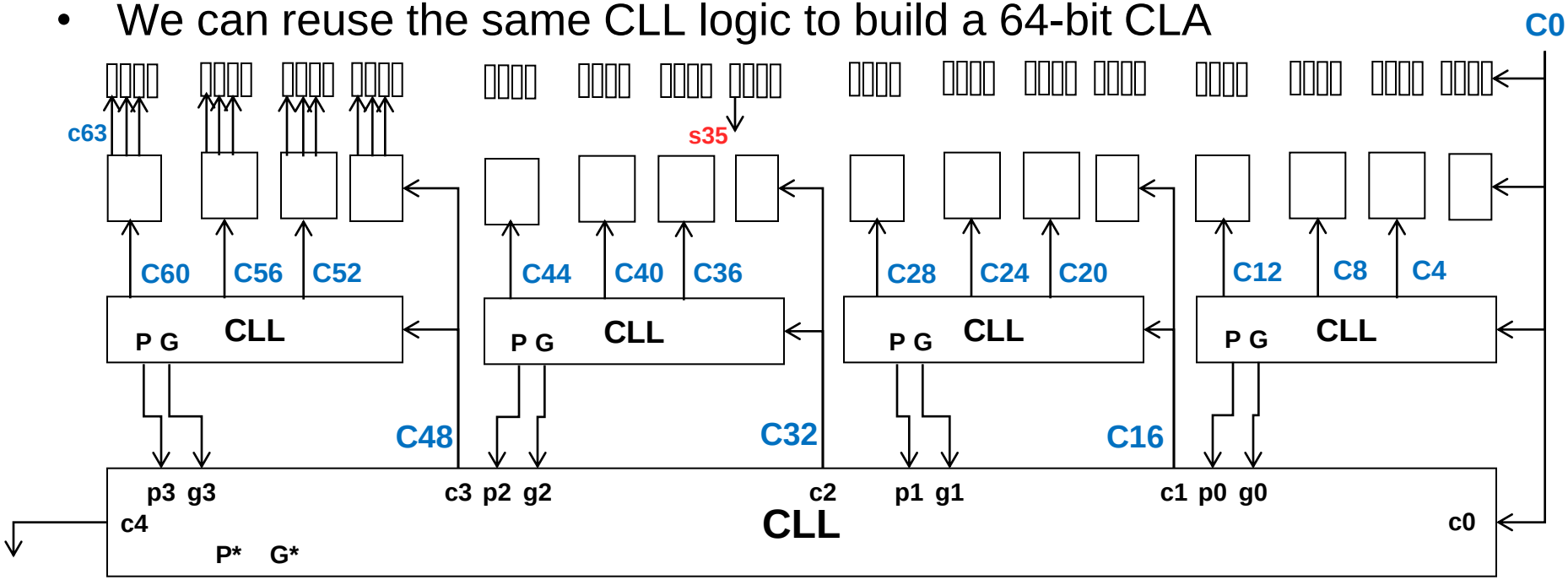
- Understanding 16-bit CLA hierarchy…



Delay =
= ___ = Delay in producing pi,gi
= ___ = Delay in producing Pi*,Gi*
= ___ = Delay in producing C4,C8,C12,C16
= ___ = Delay in producing c15
= ___ = Delay in producing S15

64-Bit CLA

- We can reuse the same CLL logic to build a 64-bit CLA



= ___ = Delay in producing S63

Is the delay in producing s63 the same as in s35?

= ___ = Delay in producing S2

= ___ = Delay in producing S0

= ___ = Delay in producing pi*,gi*

= ___ = Delay in producing Pj**,Gj**

= ___ = Delay in producing C48

= ___ = Delay in producing C60

= ___ = Delay in producing C63

= ___ = Delay in producing S63

= _____ Total Delay