Spiral 2-2

Arithmetic Components and Their Efficient Implementations

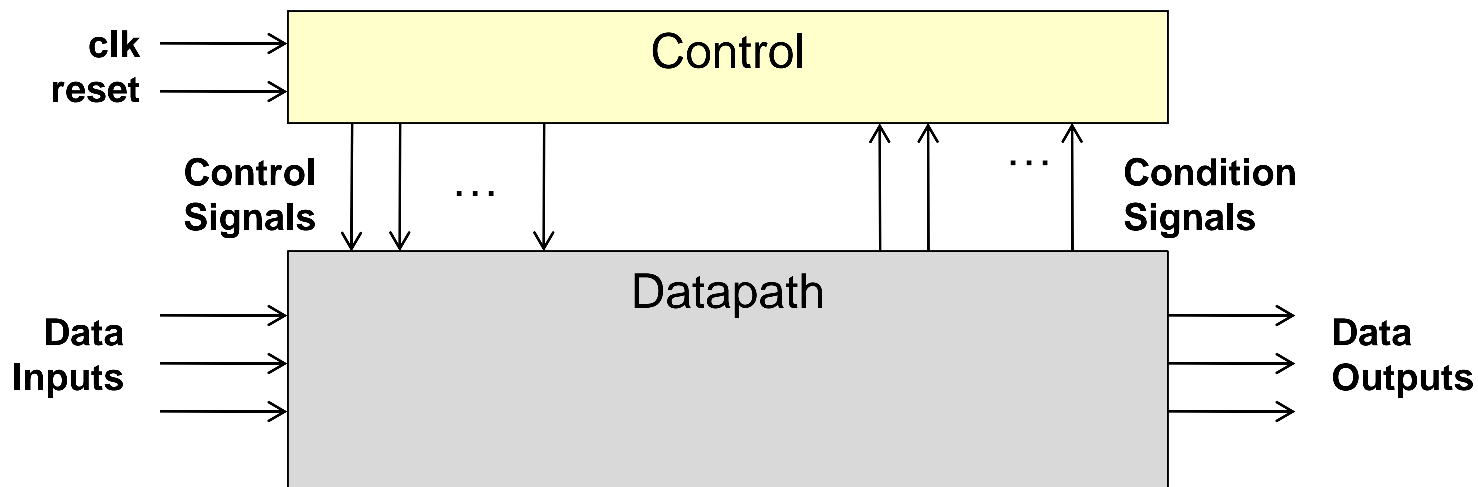
Learning Outcomes

- I know how to combine overflow and subtraction results to determine comparison results of both signed and unsigned numbers
- I understand how combination multipliers can be built
- I understand how hierarchical carry lookahead logic can be used to produce logarithmic time delay for an adder

DATAPATH COMPONENTS

Digital System Design

- Control **(CU)** and Datapath Unit **(DPU)** paradigm
  - Separate logic into datapath elements that operate on data and control elements that generate control signals for datapath elements
  - Datapath: Adders, muxes, comparators, counters, registers (shift, with enables, etc.), memories, FIFO's
  - Control Unit: State machines/sequencers



Detecting Overflow Helps Us Perform Comparison

OVERFLOW & COMPARISON

Overflow

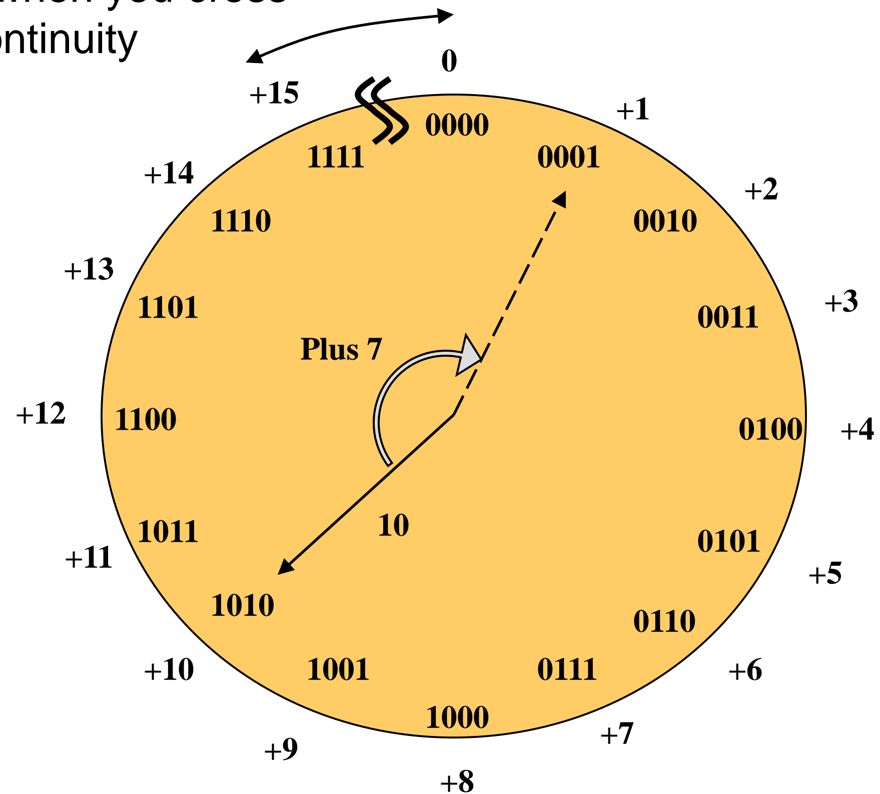
- Overflow occurs when the result of an arithmetic operation is too large to be represented with the given number of bits
  - Unsigned overflow occurs when adding or subtracting unsigned numbers
  - Signed (2's complement overflow) overflow occurs when adding or subtracting 2's complement numbers

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.

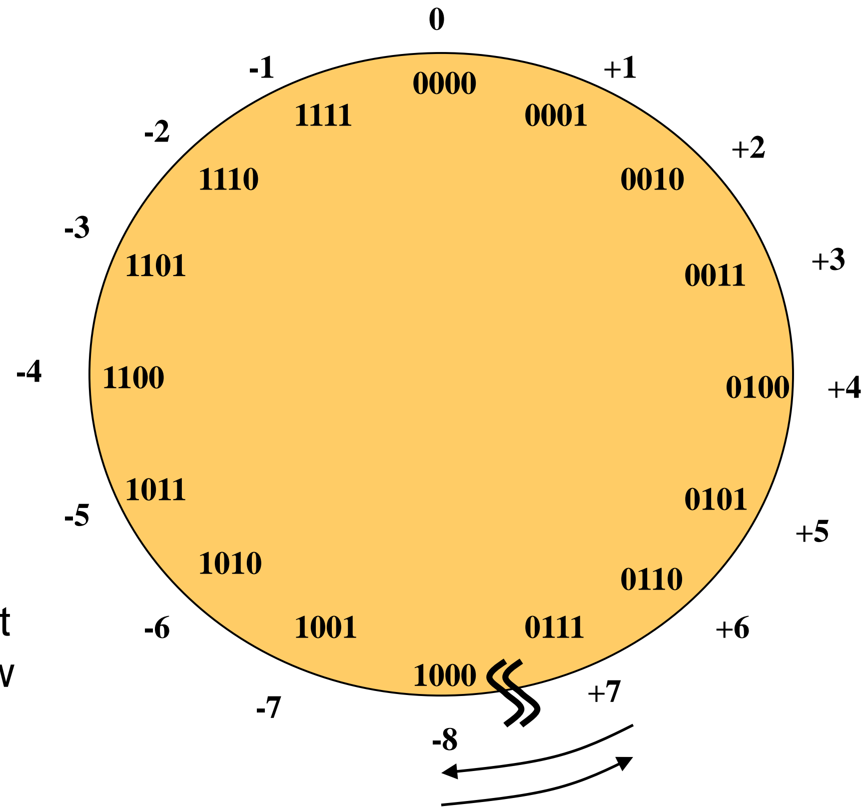


2's Complement Overflow

$5 + 7 = +12$

$-6 + -4 = -10$

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



Testing for Overflow

- Most fundamental test
  - Check if answer is wrong (i.e. Positive + Positive yields a negative)
- Unsigned overflow test [Different for add or sub]
  - Addition: If carry-out of final position equals '1'
  - Subtraction: If carry-out of final addition equals '0'
- Signed (2's complement) overflow test [Same for add or sub]
  - 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 has occurred if…
  - Unsigned Addition: If final carry-out = 1
  - Unsigned Subtraction: If final carry-out = 0

```
   1011            1011
+  0110         +  0011
-------         -------

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

Testing for Unsigned Overflow

- Unsigned Overflow has occurred if...
  - Unsigned Addition: If final carry-out = 1
  - Unsigned Subtraction: If final carry-out = 0

```
 1 1 1
  1011
+ 0110
------
  0001
```

Final carry-out = 1, thus overflow

```
 0 0 11
  1011
+ 0011
------
  1110
```

Final carry-out = 0, thus no overflow

```
  1011         1    1 1
- 0110    =>       1011
------             1001
                 +    1
                 ------
                   0101
```

Final carry-out = 1, thus no overflow

```
  0110         0 1
- 1011    =>   0110
------         0100
             +    1
             ------
               1011
```

Final carry-out = 0, thus overflow

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)
+  0110  (6)
```

```
   1100 (-4)
+  1001 (-7)
```

```
   0011  (3)
+  0010  (2)
```

```
   1110 (-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!

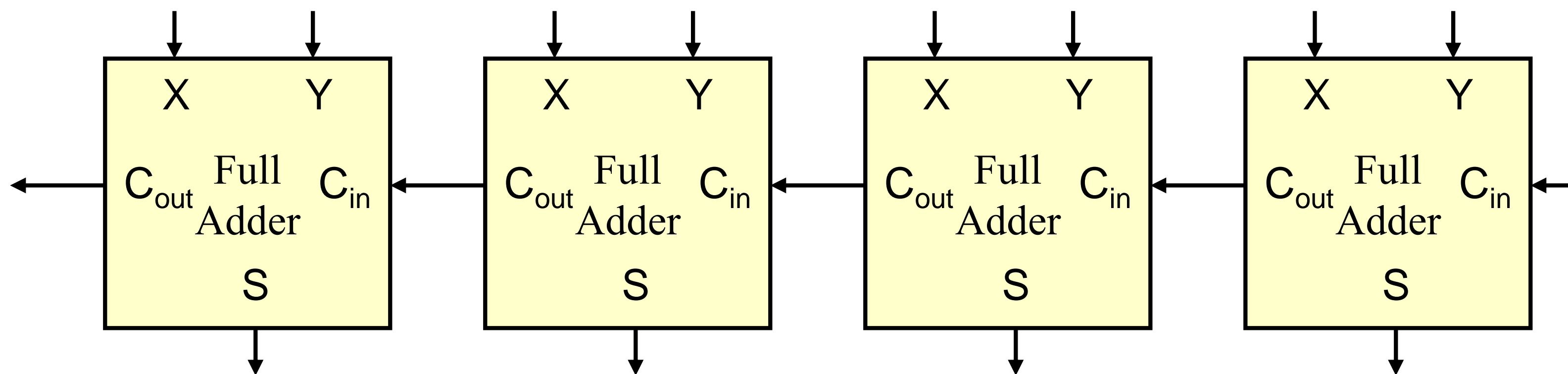
```
   0 0
   0011  (3)
 + 0010  (2)
 ------
   0101  (5)
```

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



COMPARISON

Comparison Via Subtraction

- Suppose we want to compare two numbers: A & B
- Suppose we let DIFF = A-B…what could the result tell us
  - If DIFF < 0, then A < B
  - If DIFF = 0, then A=B
  - IF DIFF > 0, then A > B
- How would we know DIFF == 0?
  - If all bits of our answer are 0…check with a NOR gate.
- How would we know DIFF < 0 (i.e. negative)?
  - Signed: Check MSB! (but what about overflow)
  - Unsigned: Huh? In unsigned there are no negative results

Computing A<B from "Negative" Result

Unsigned

- Perform A-B
- If A-B would yield a negative result, this will appear as "overflow" in an unsigned subtraction
- And we know unsigned subtraction overflow occurs if Cout = 0
- So just check if Cout=0

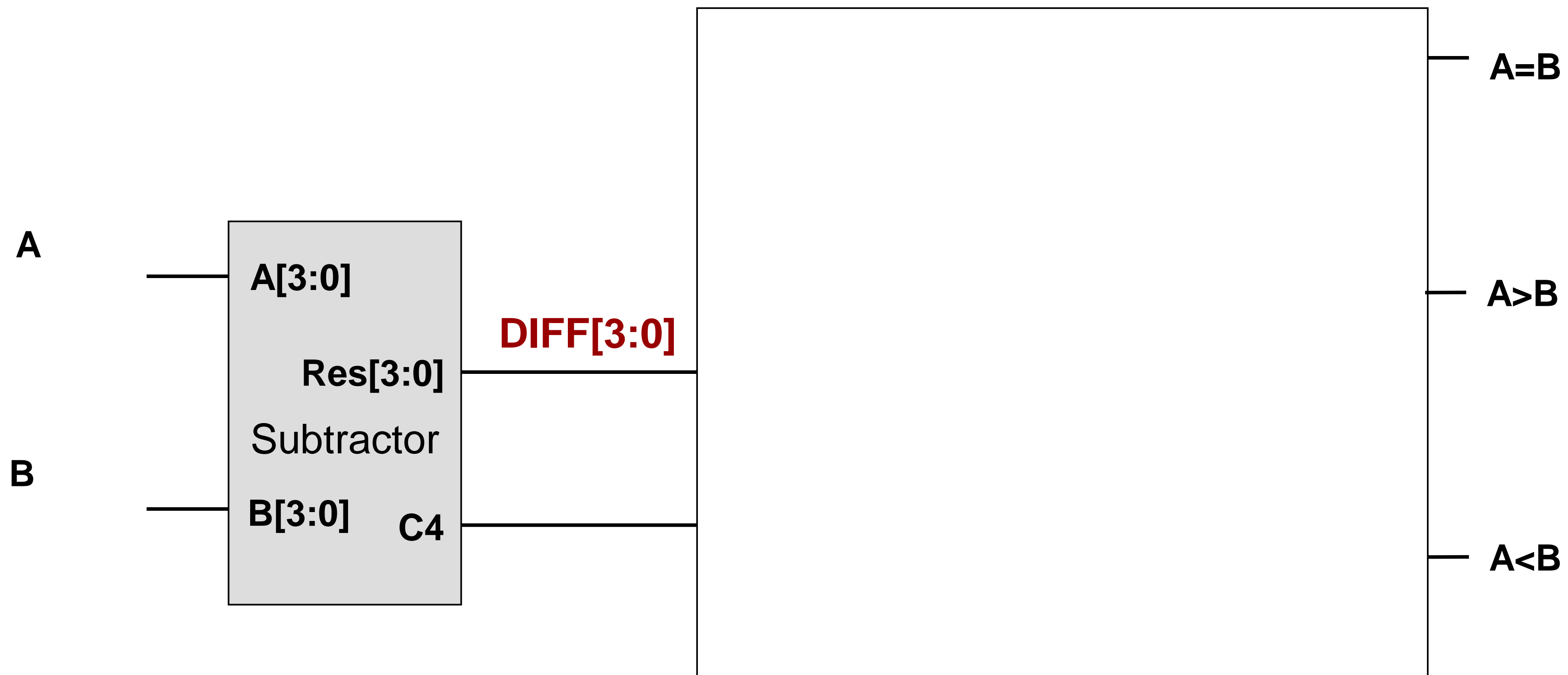
Signed

- Perform A-B
- If ***there is no overflow (V=0)***, simply check if MSB = 1
- But if there is overflow??
  - Recall overflow has the effect of flipping the sign of the result to the opposite of what it should be.
- So if ***there is overflow (V=1)*** check is MSB = 0 (i.e. positive)
- Summary: A-B is "truly' negative if V=0 & MSB=1 or V=1 & MSB=0

USC Viterbi School of Engineering

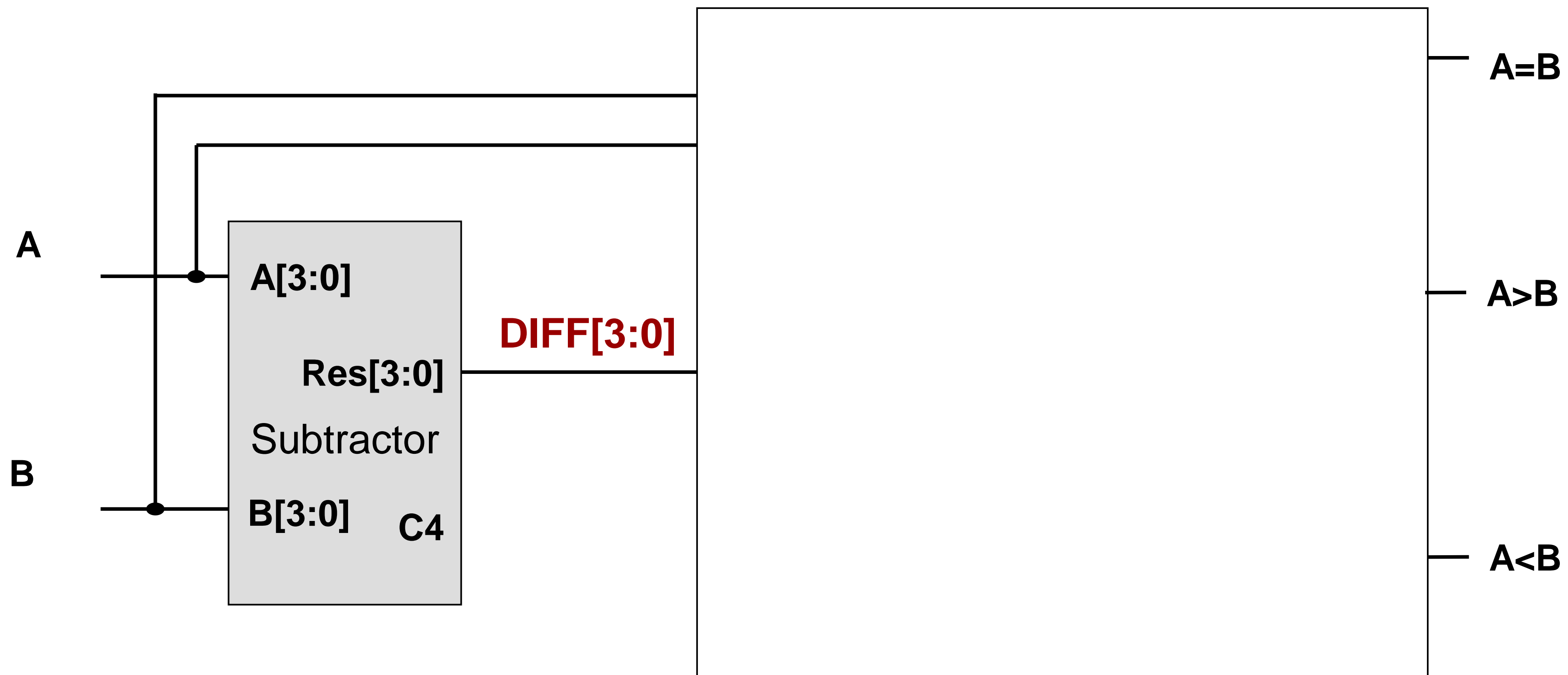
Unsigned Comparator

- A comparator can be built by using a subtractor



Signed Comparator

- A comparator can be built by using a subtractor

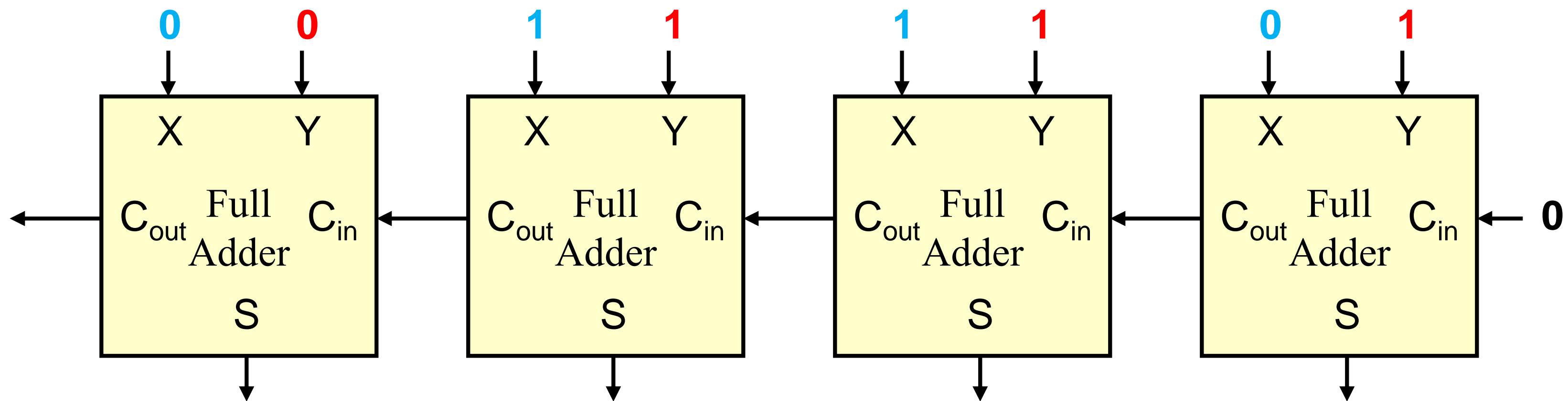


ADDER TIMING

Addition – Full Adders

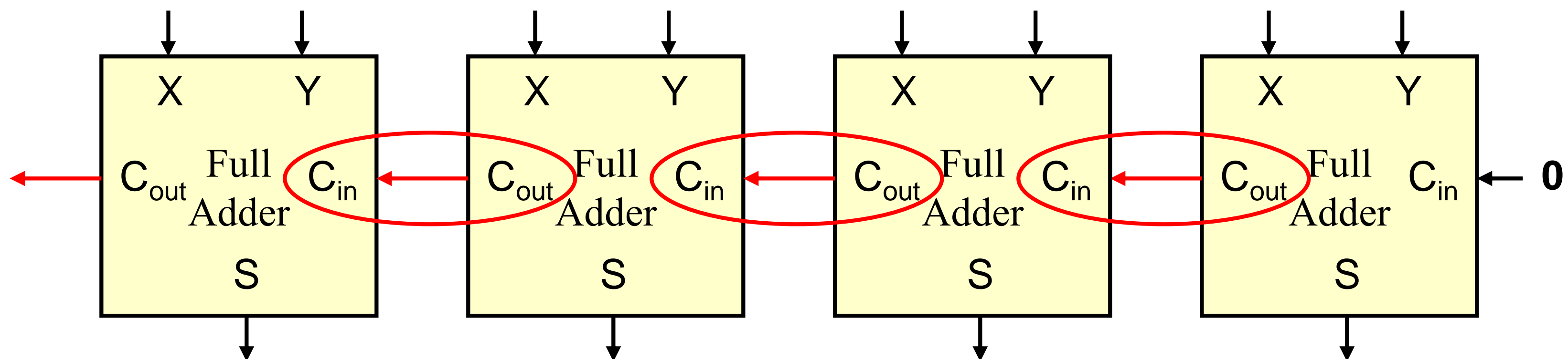
- Be sure to connect first C_{in} to 0

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



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:

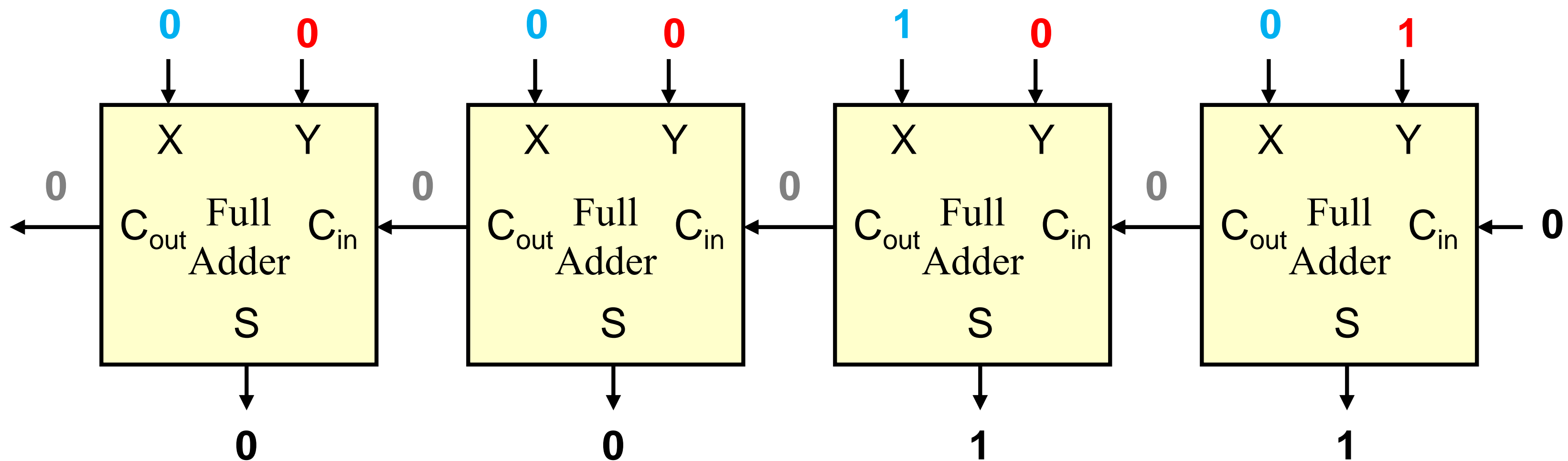
Old inputs:

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

New inputs:

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

Old inputs:



Timing

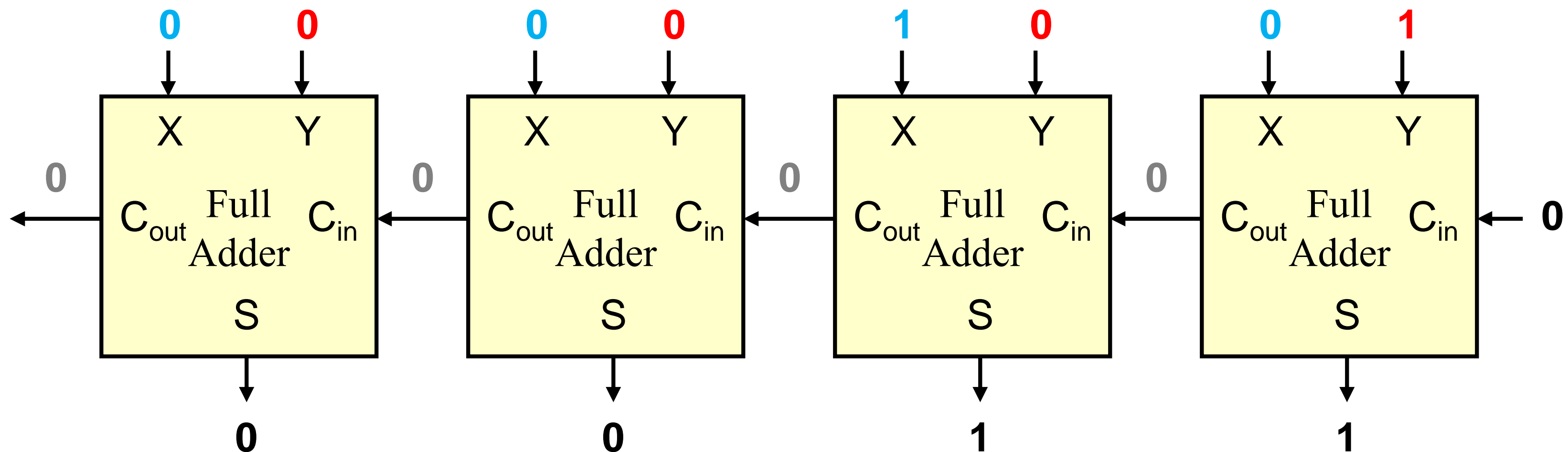
- At the time just before we enter the new input values, all carries are 0's

New inputs:

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

Time
-1

Old inputs:

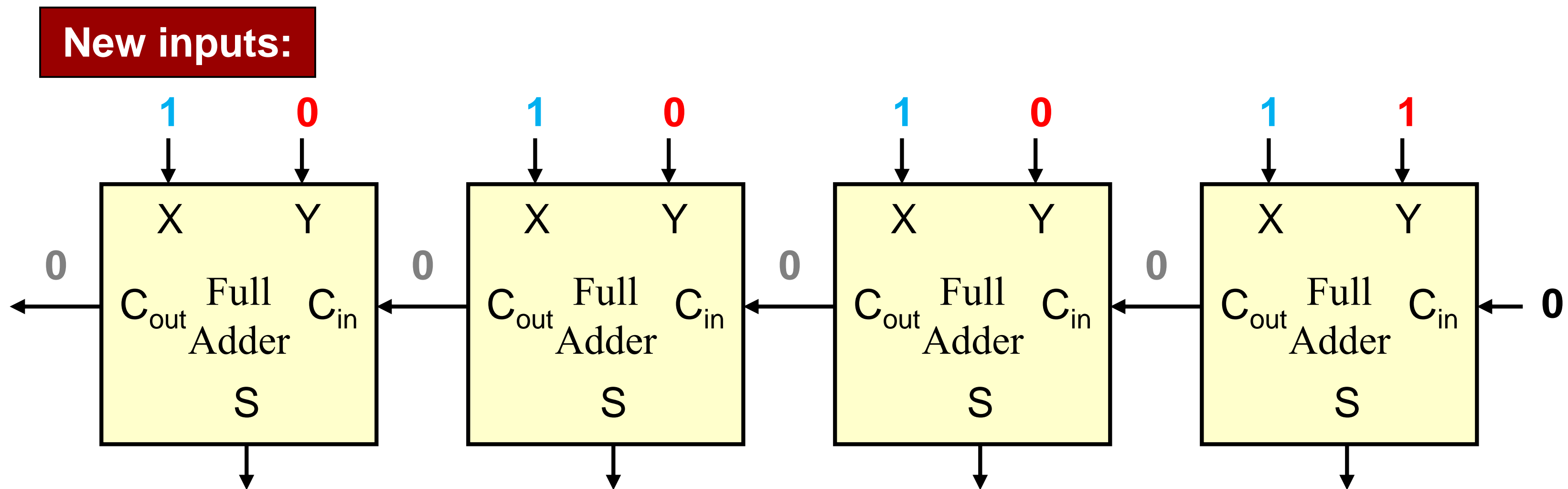


Timing

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

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

Time
0

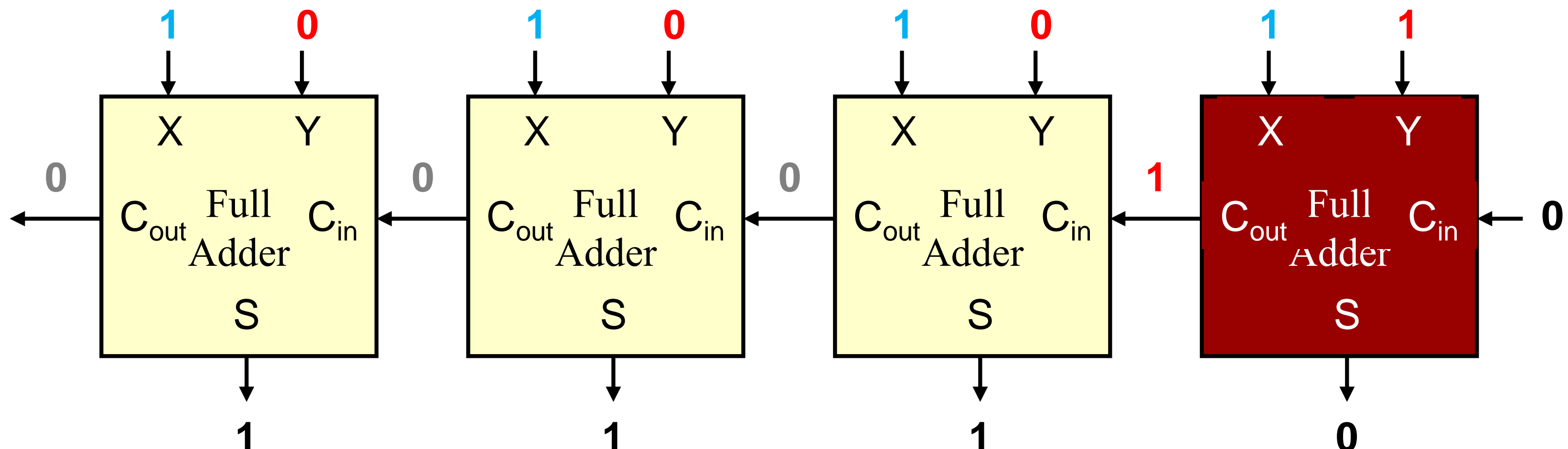


Timing

- 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



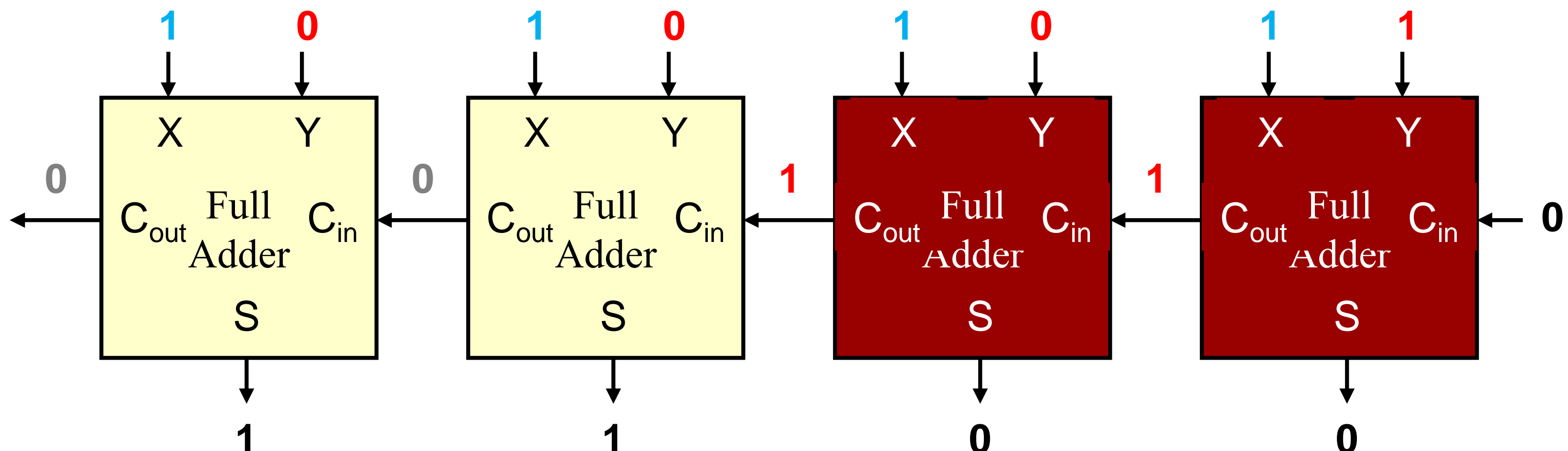
Now the carries are all based off the new inputs

Timing

- The carry is "rippling" through each adder

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

Time
2

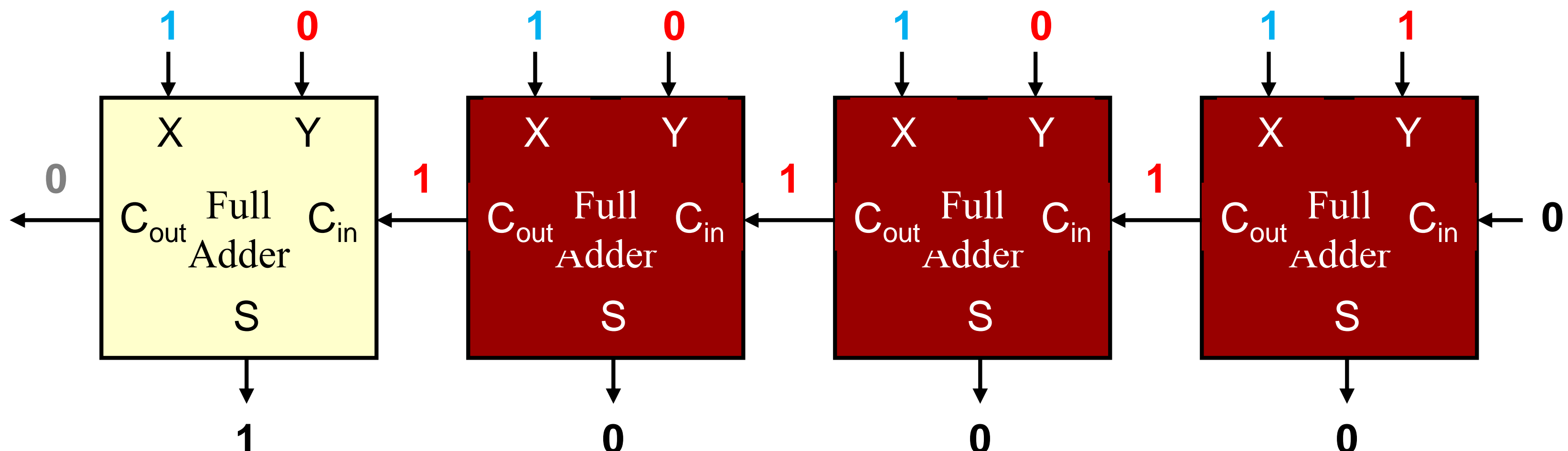


Timing

- The carry is "rippling" through each adder

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

Time
3

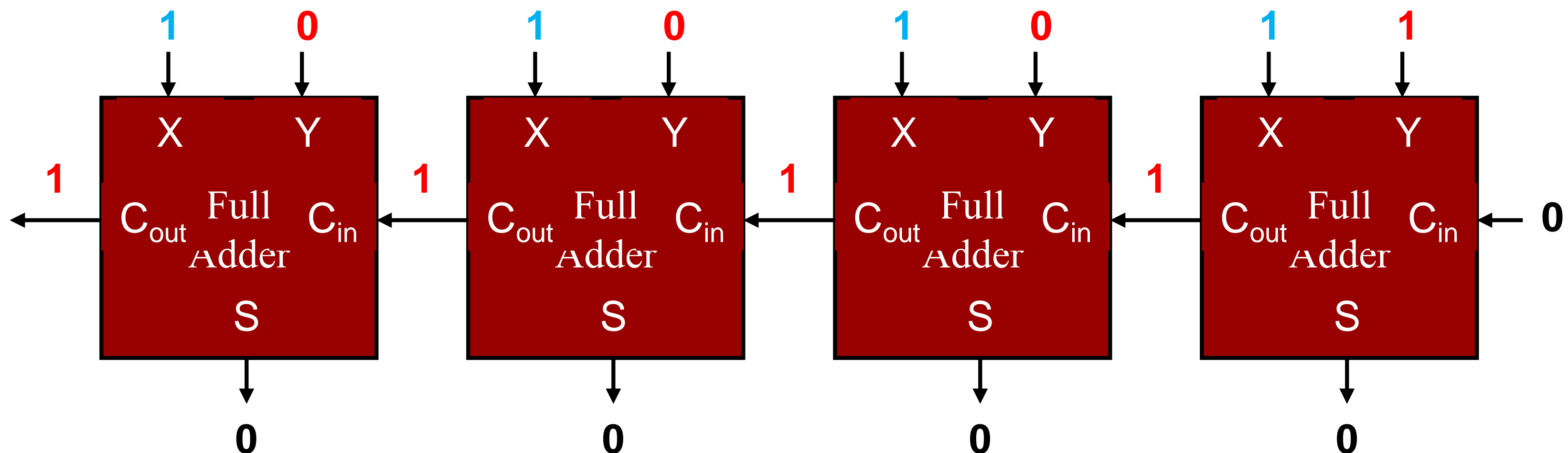


Timing

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

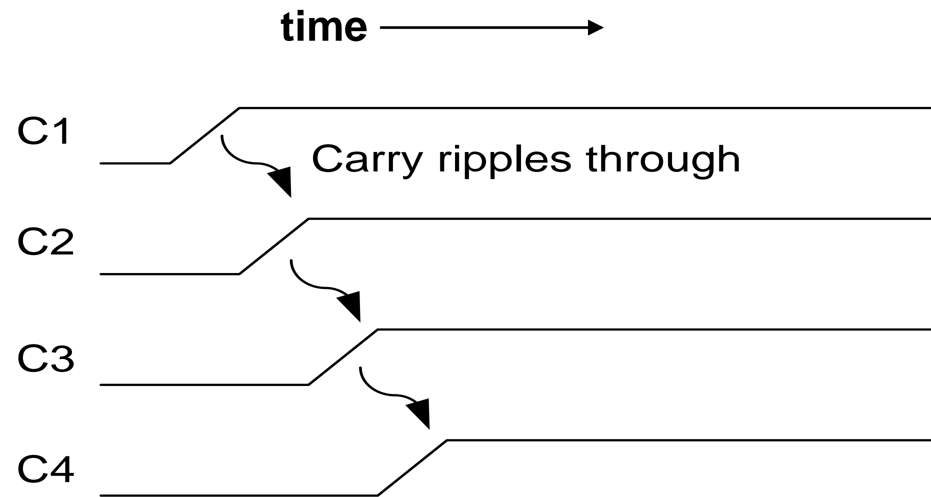
```
  1111
   1111 = X
+  0001 = Y
-------
   0000
```

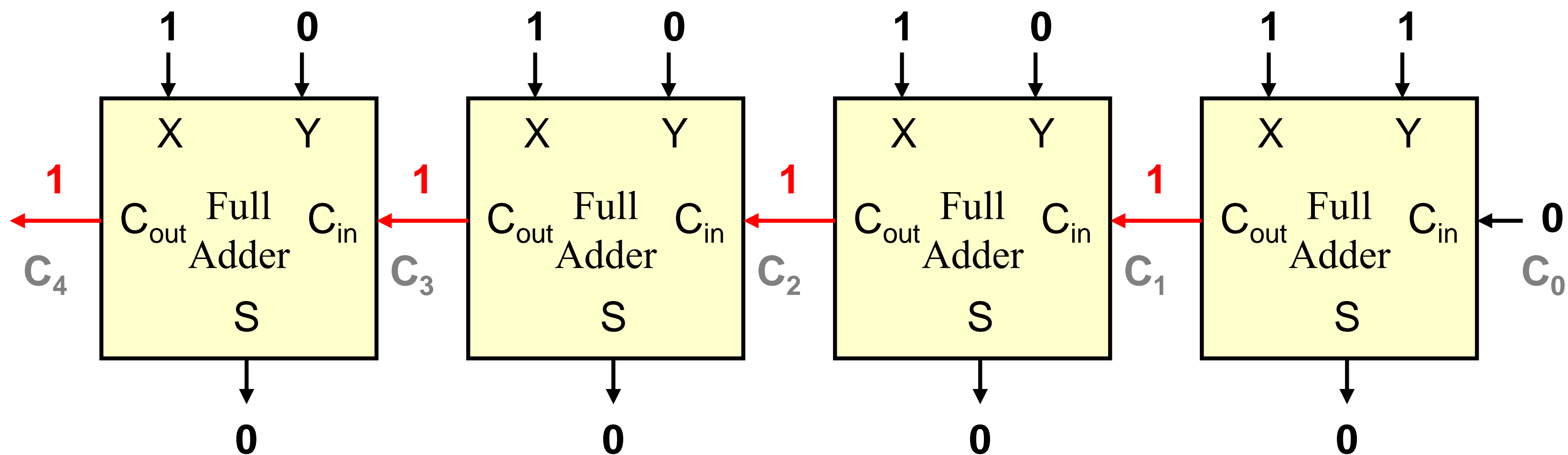
Time
4



"Ripple-Carry" Adder

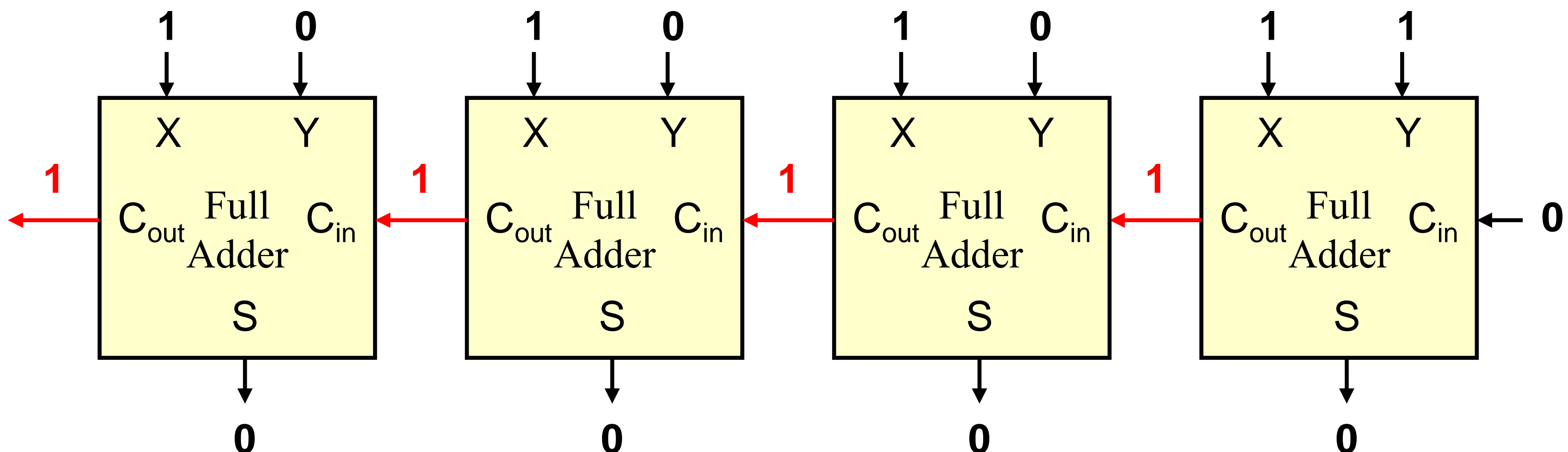
- The longest path through a chain of full adders is the carry path
- We say that the carry "ripples" through the adder





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)

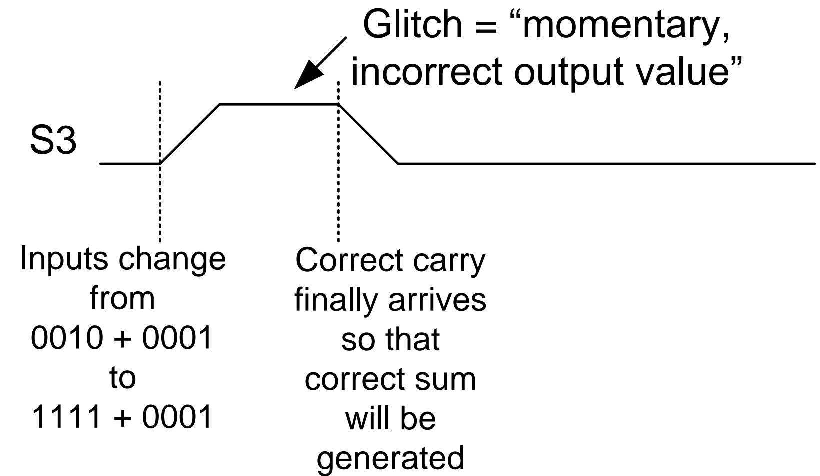


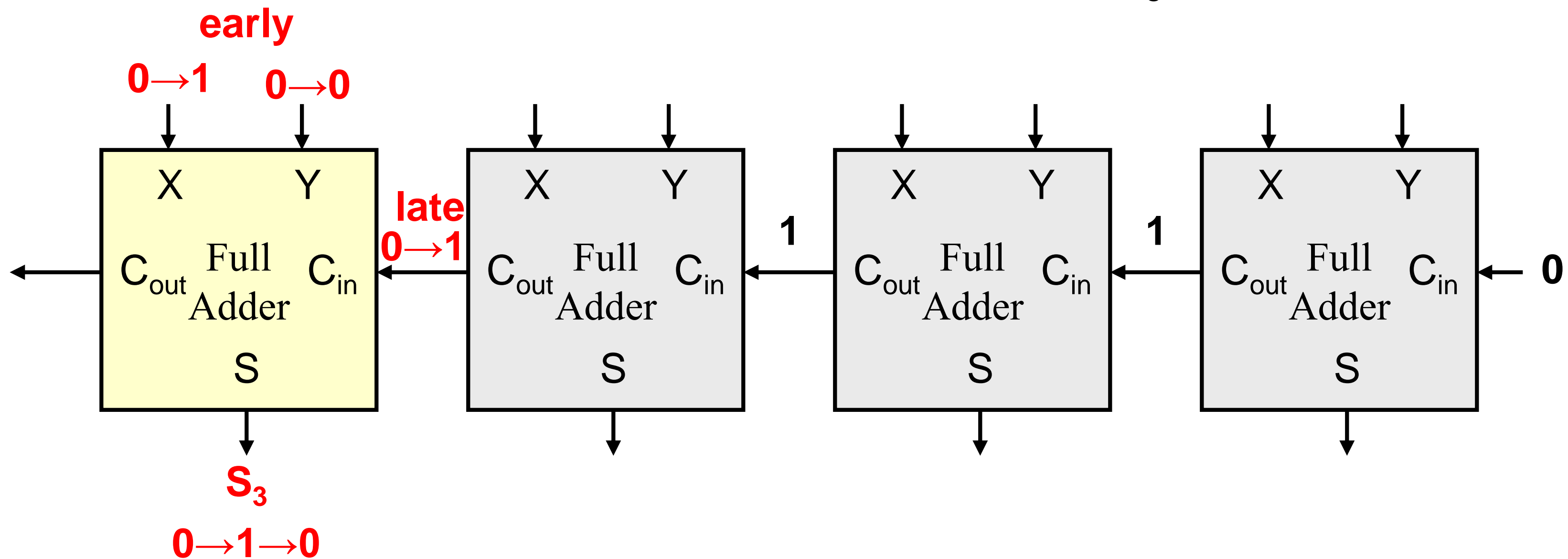
Glitches

- 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



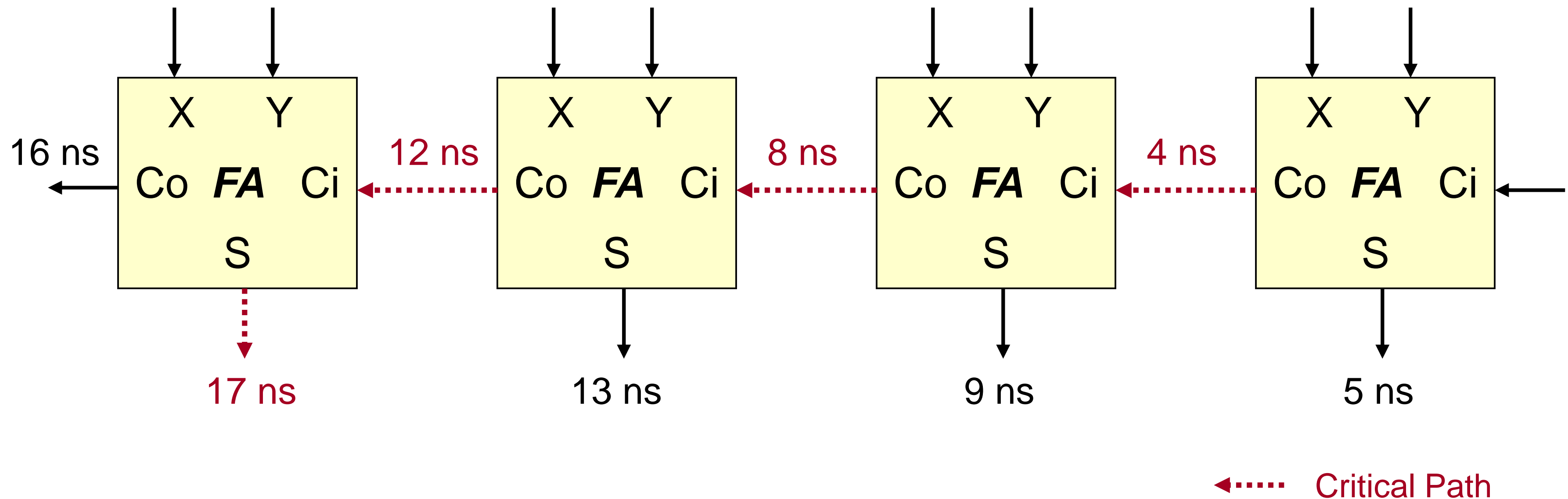


Critical Path

- Critical Path = Longest possible delay path

Assume t_{sum} = 5 ns,

t_{carry}= 4 ns



MULTIPLIERS

Unsigned Multiplication Review

- Same rules as decimal multiplication
- Multiply each bit of Q by M shifting as you go
- An m-bit * n-bit mult. produces an m+n bit result (i.e. n-bit * n-bit produces 2*n bit result)
- Notice each partial product is a shifted copy of M or 0 (zero)

```
    1010  M (Multiplicand)
*   1011  Q (Multiplier)
--------
```

Unsigned Multiplication Review

- Same rules as decimal multiplication
- Multiply each bit of Q by M shifting as you go
- An m-bit * n-bit mult. produces an m+n bit result (i.e. n-bit * n-bit produces 2*n bit result)
- Notice each partial product is a shifted copy of M or 0 (zero)

```
         1010   M  (Multiplicand)
     *   1011   Q  (Multiplier)
     --------
         1010
        1010_   PP(Partial
       0000__      Products)
  +   1010___
  -----------
     01101110   P  (Product)
```

Signed Multiplication Techniques

- When adding signed (2's comp.) numbers, some new issues arise
- Must sign extend partial products (out to 2n bits)

Without Sign Extension…
Wrong Answer!

```
      1001   = -7
   *  0110   = +6
   --------
      0000
     1001_
    1001__
 +  0000___
 ----------
  00110110   = +54
```

With Sign Extension…
Correct Answer!

```
      1001   = -7
   *  0110   = +6
   --------
  00000000
  1111001_
  111001__
+ 00000___
----------
  11010110   = -42
```

Signed Multiplication Techniques

- Also, must worry about negative multiplier
  - MSB of multiplier has negative weight
  - If MSB=1, multiply by -1 (i.e. take 2's comp. of multiplicand)

With Sign Extension but w/o consideration of MSB… Wrong Answer!

```
      1100   = -4
  *   1010   = -6
  --------
  00000000
  1111100_
  000000__
+ 11100___
  --------
  11011000   = -40
```

With Sign Extension and w/ consideration of MSB… Correct Answer!

Place Value: -8
Multiply by -1

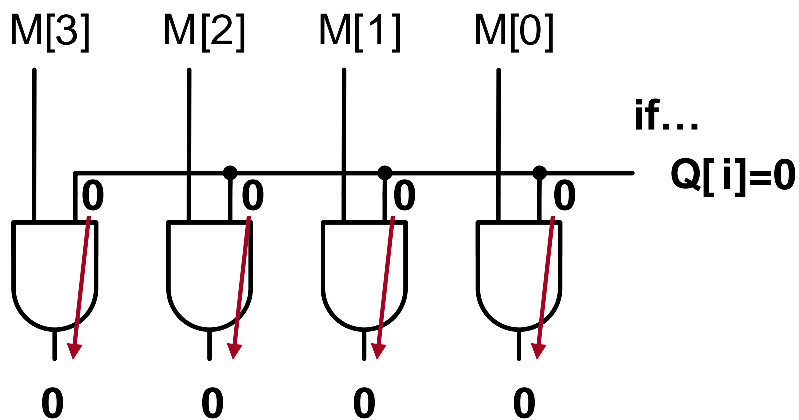
```
      1100   = -4
  *   1010   = -6
  --------
  00000000
  1111100_
  000000__
+ 00100___
  --------
  00011000   = +24
```

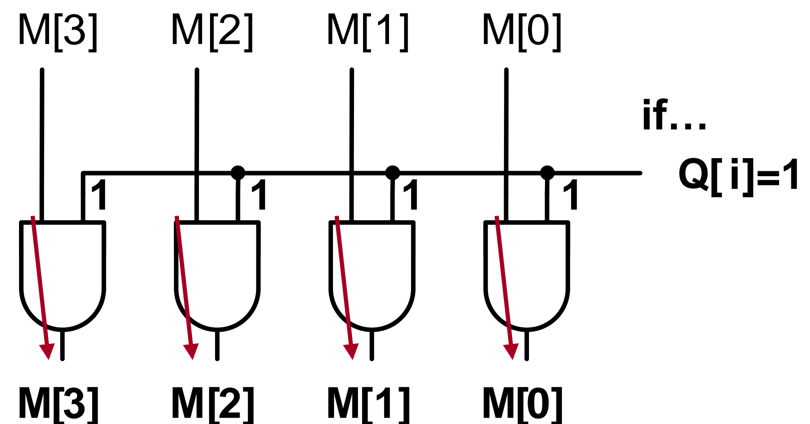
Combinational Multiplier

- Partial Product (PP_i) Generation
  - Multiply Q[i] * M
    - if Q[i]=0 => PP_i = 0
    - if Q[i]=1 => PP_i = M

Combinational Multiplier

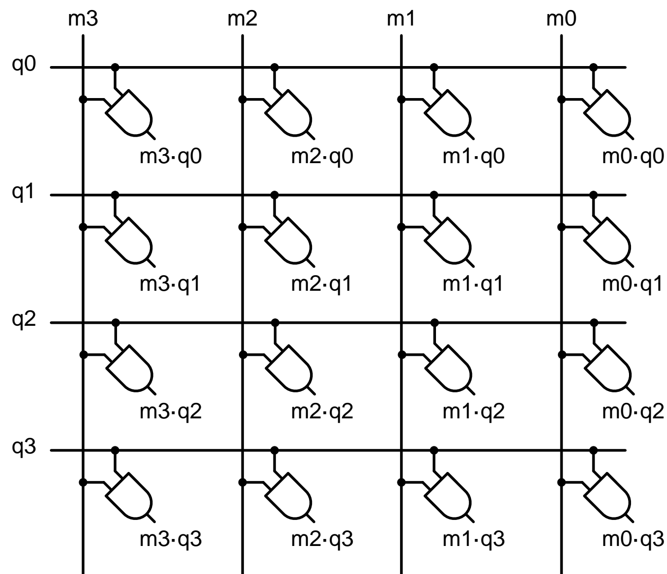
- Partial Product (PP_i) Generation
  - Multiply Q[i] * M
    - if Q[i]=0 => PP_i = 0
    - if Q[i]=1 => PP_i = M
  - AND gates can be used to generate each partial product





Multiplication Overview

- Multiplication approaches:
  - Sequential: Shift-and-Add produces one product bit per clock cycle time (usually slow)
  - Combinational: Array multiplier uses an array of adders
    - Can be as simple as N-1 ripple-carry adders for an NxN multiplication



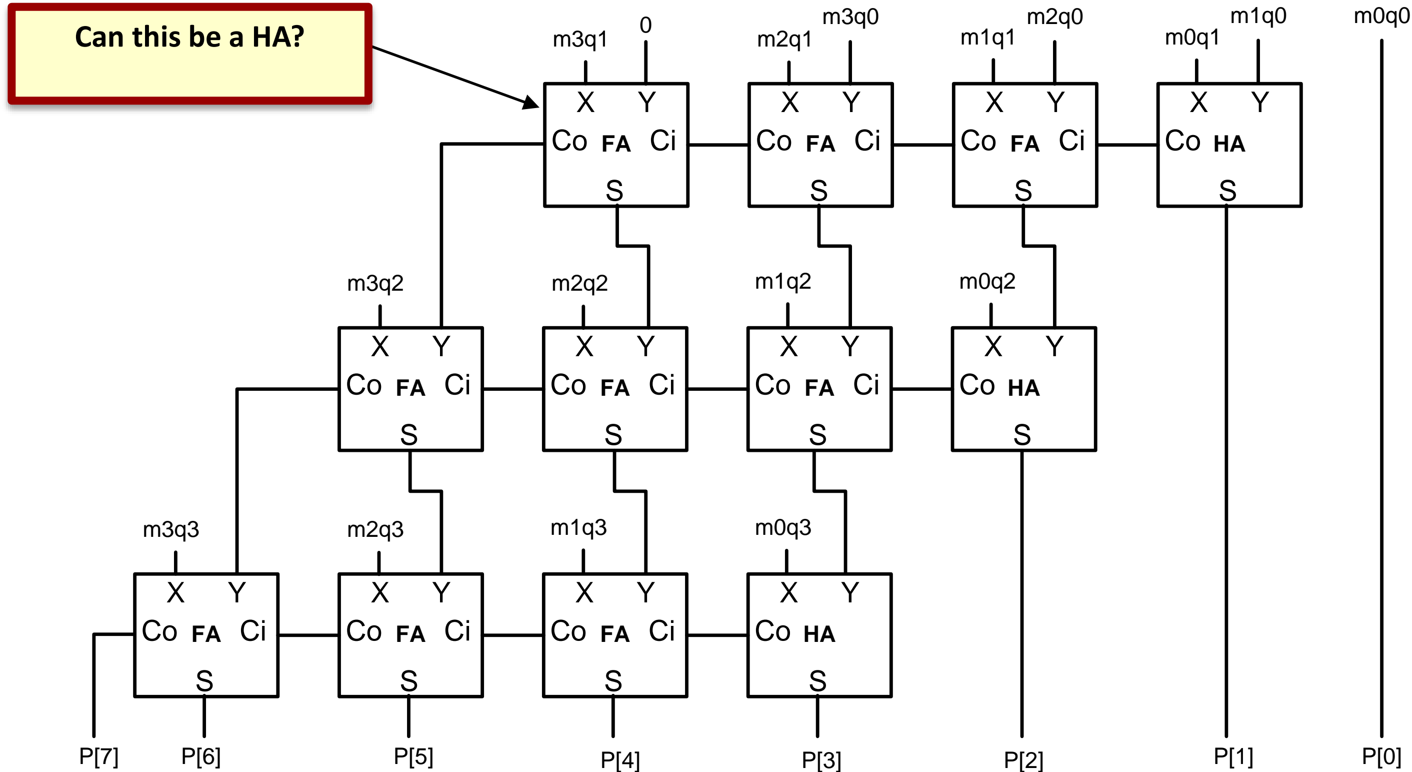
AND Gate Array produces partial product terms

				m3	m2	m1	m0
	x			q3	q2	q1	q0
				m3q0	m2q0	m1q0	m0q0
			m3q1	m2q1	m1q1	m0q1	
		m3q2	m2q2	m1q2	m0q2		
	+ m3q3	m2q3	m1q3	m0q3			
p7	p6	p5	p4	p3	p2	p1	p0

Combinational Multiplier

- Partial Products must be added together
- Combinational multipliers require long propagation delay through the adders
  - propagation delay is proportional to the number of partial products (i.e. number of bits of input) and the width of each adder

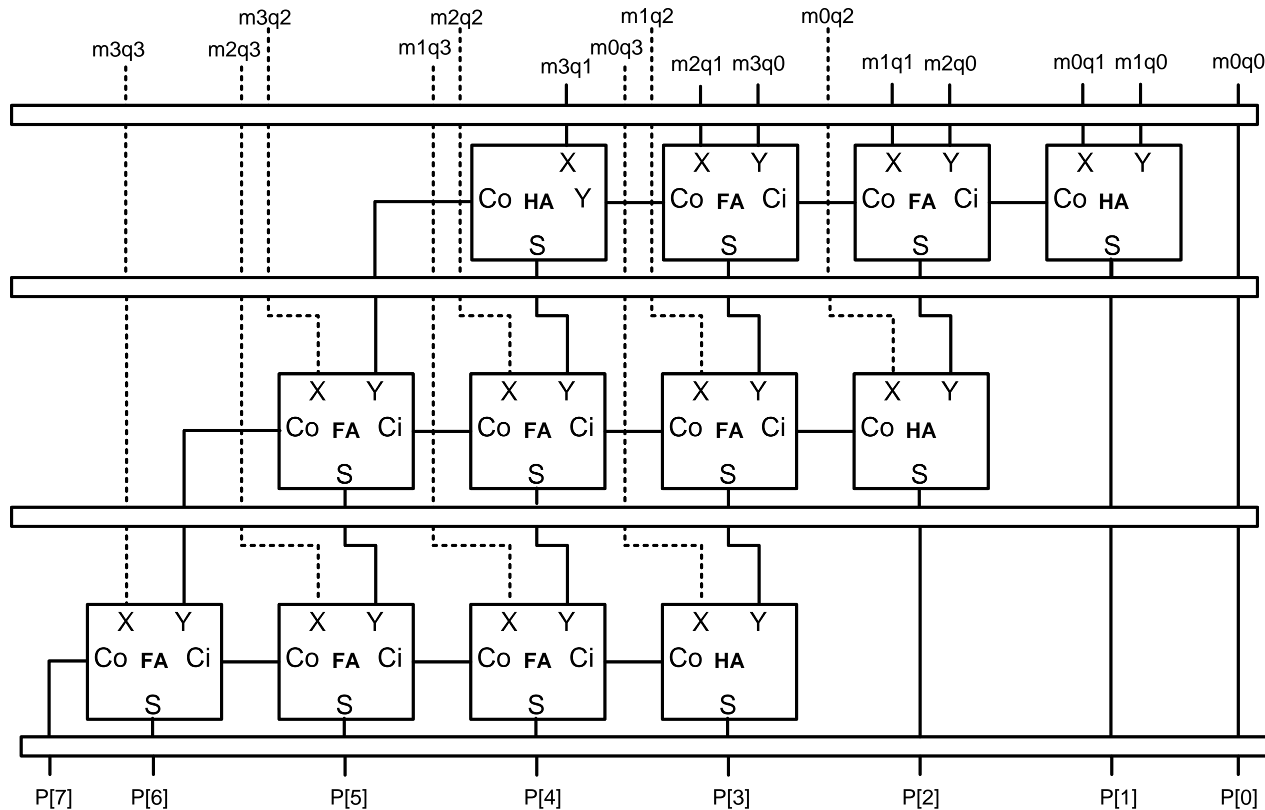
Array Multiplier



- Maximum delay = ?
  - Do you look for the longest path or the shortest path between any input and output?
  - Compare with the delay of a shift-and-add method

Pipelined Multiplier

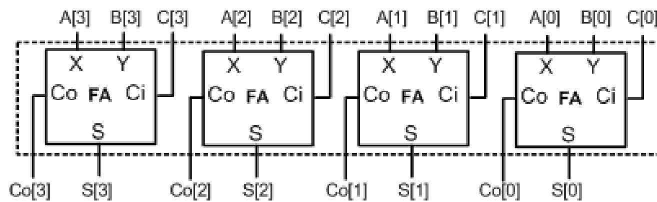
- Now try to pipeline the previous design

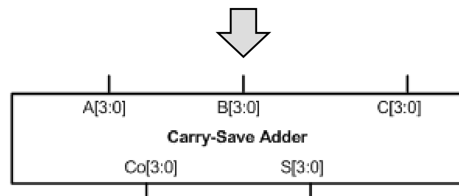


Determine the maximum stage delay to decide the pipeline clock rate. Assume zero-delay for stage latches. How does the latency of the pipeline compare with the simple combinational array of the previous stage?

Carry-Save Multiplier

- Instead of propagating the carries to the left in the same row, carries are now sent down to the next stage to reduce stage delay and facilitate pipelining





The upper three stages are 3-bit Carry Save Adders (CSA's) each with 2-gate delays.

The last stage is a Ripple Carry Adder (RCA) which requires longer delay. It can be replaced by a CLA for larger multipliers.

CSA's

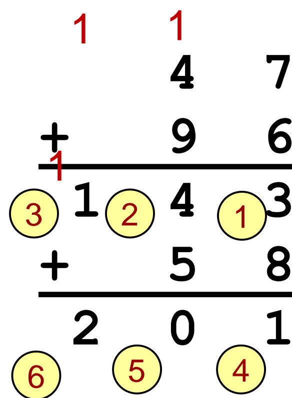
RCA

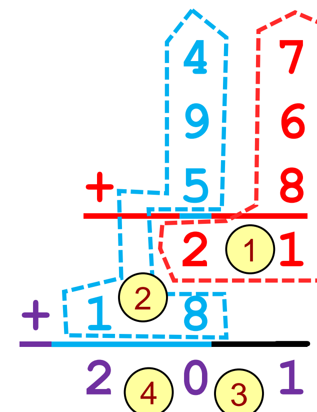
Carry Save Adders

- Consider the decimal addition of

  47 + 96 + 58 = 201

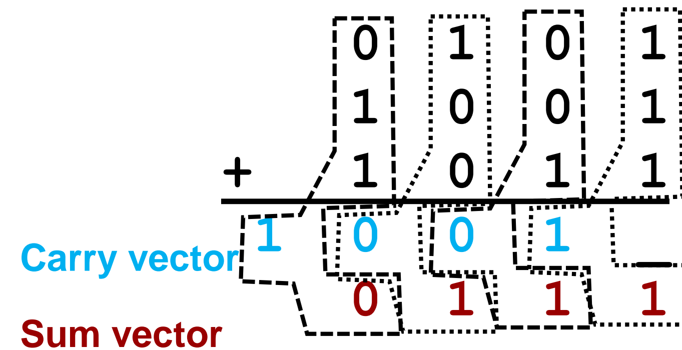
- One way is to add 47 to 96 to get 143 and then add 58
- Here the ten's column cannot be added until the carry is produced
- In the carry-save style, we add the one's column and ten's column simultaneous

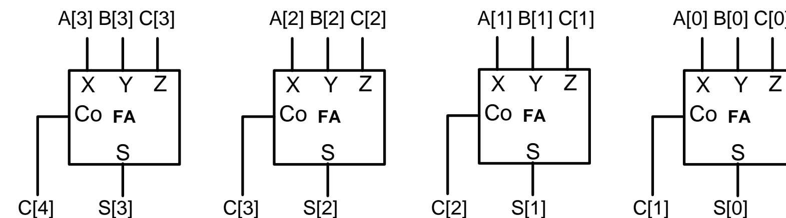




Carry-Save (3,2) Adders

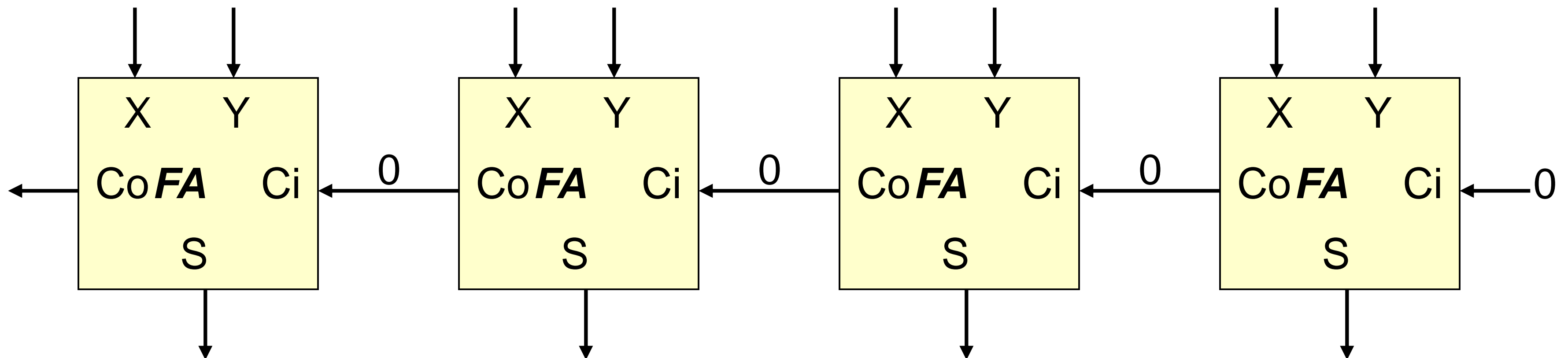
- A carry save adder is also called a (3,2) adder or a (3,2) counter (refer to Computer Arithmetic Algorithms by Israel Koren) as it takes three vectors, adds them up, and reduces them to two vectors, namely a sum vector and a carry vector
- CSA's are based on the principle that carries do not have to be added as soon as possible, but can be combined in a later step
- An n-bit CSA consist of n disjoint full adders





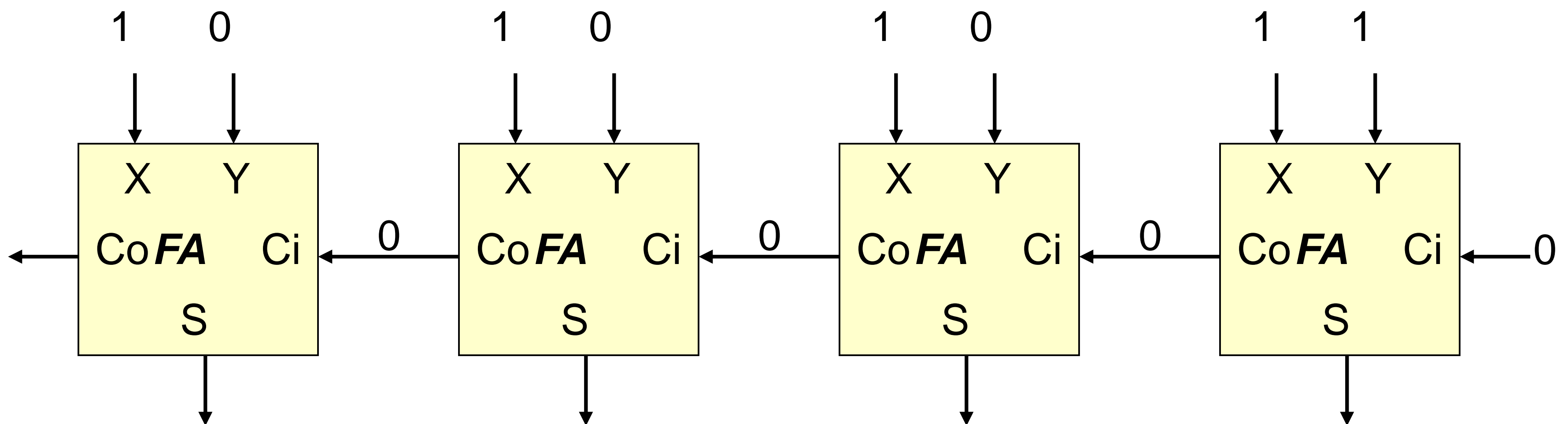
Adder Propagation Delay

```
    1111
+   0001
```

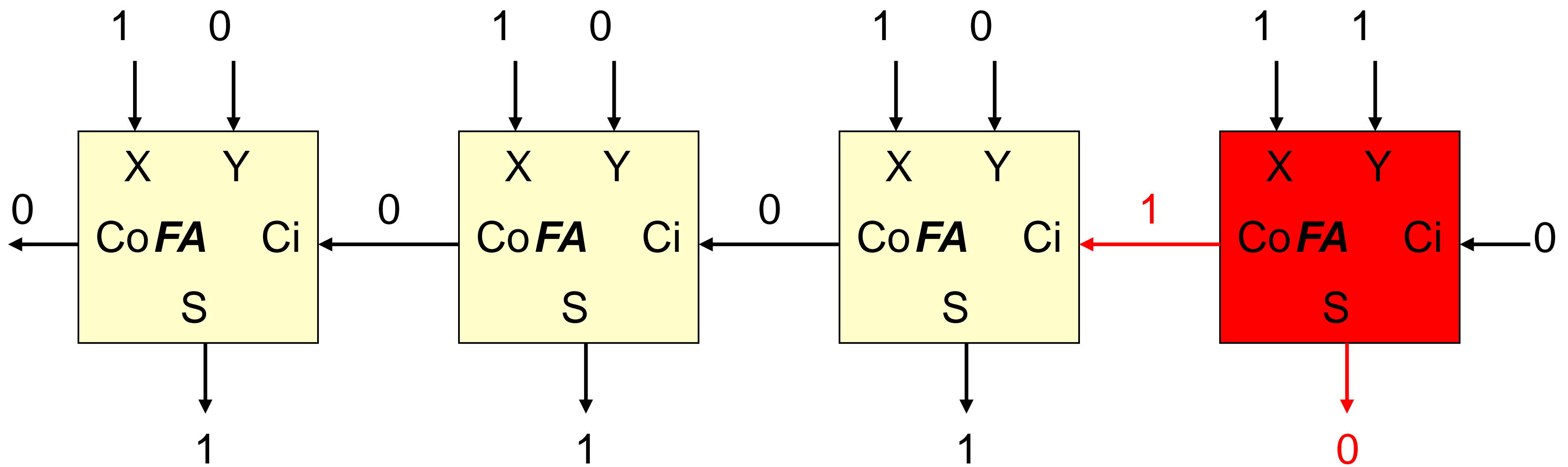


Adder Propagation Delay

```
   1111
+  0001
```

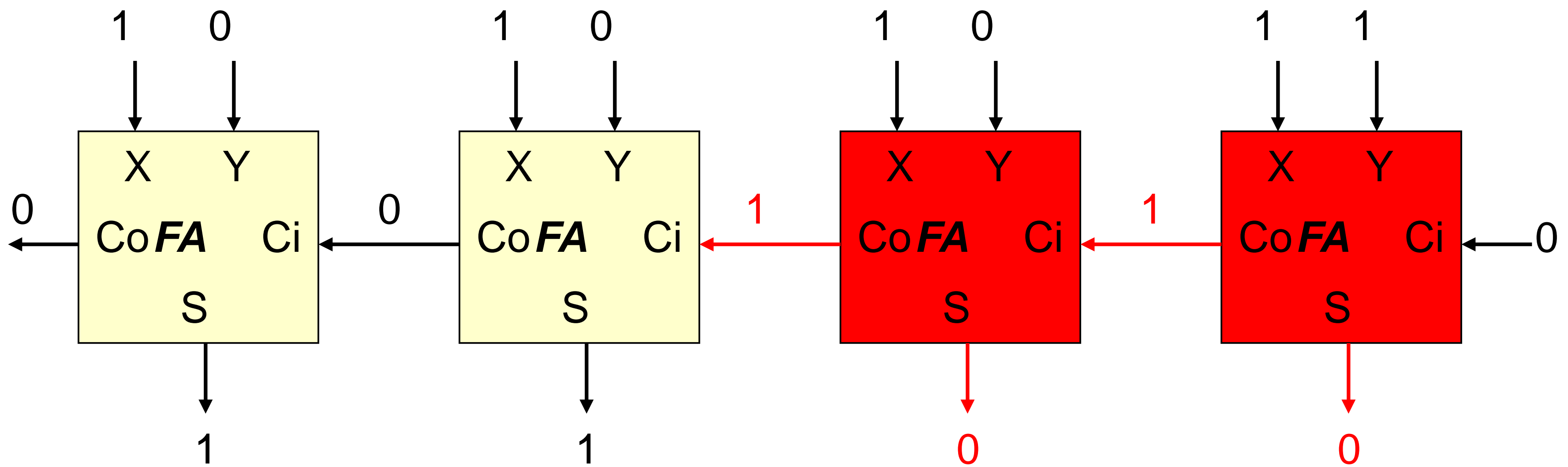


Adder Propagation Delay



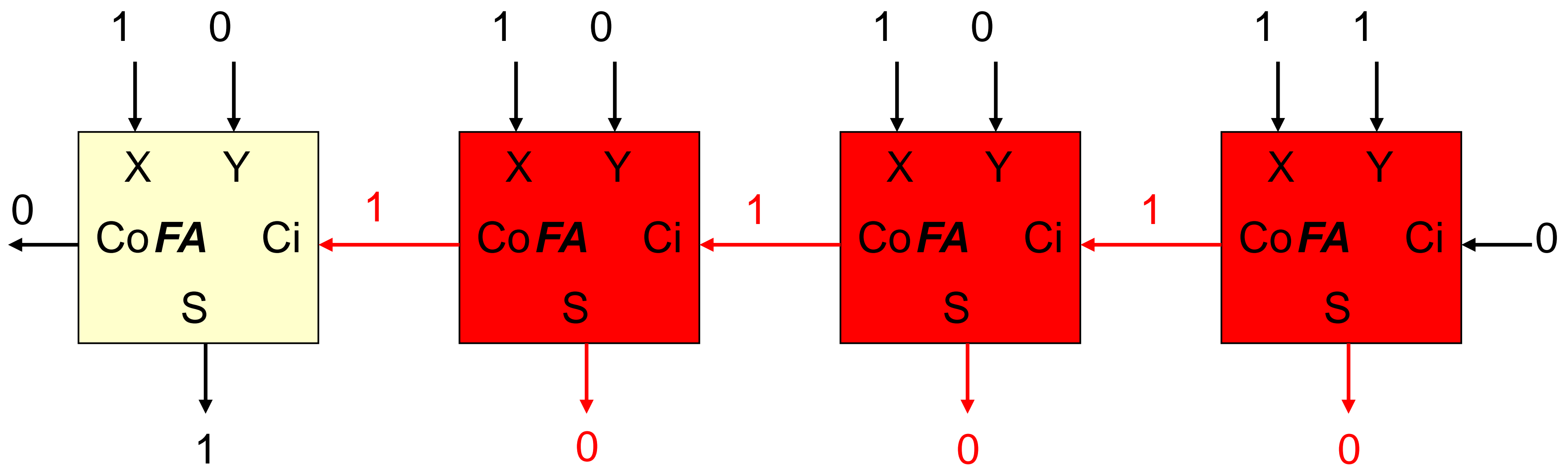
Adder Propagation Delay

```
   1111
+  0001
```



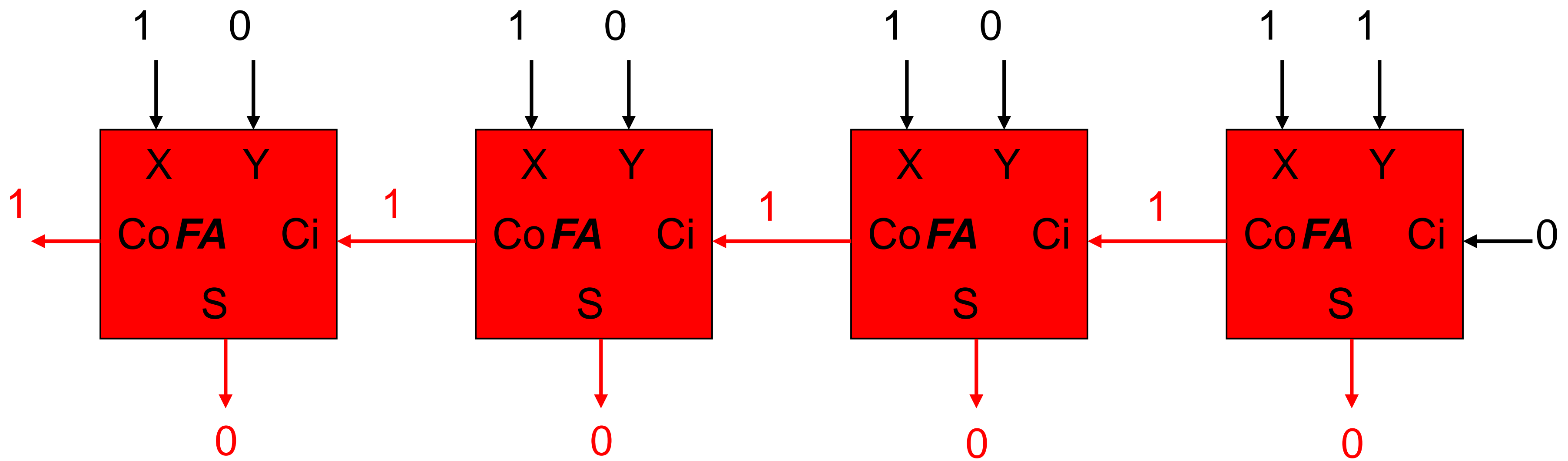
Adder Propagation Delay

```
    1111
+   0001
```



Adder Propagation Delay

```
    1111
+   0001
```

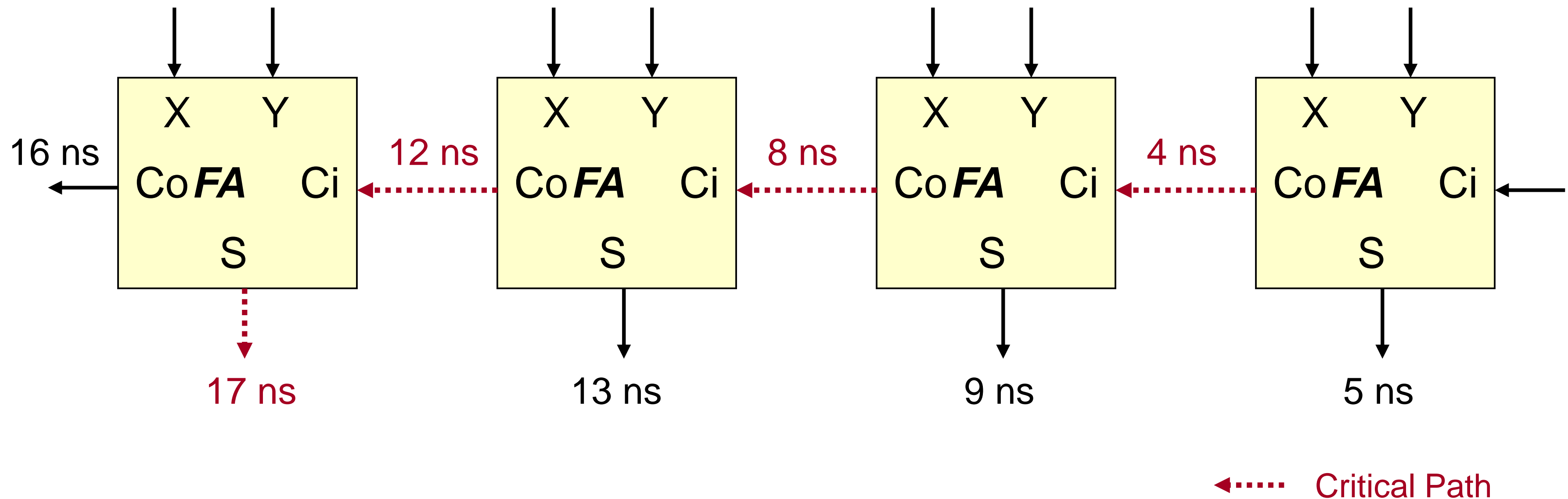


Critical Path

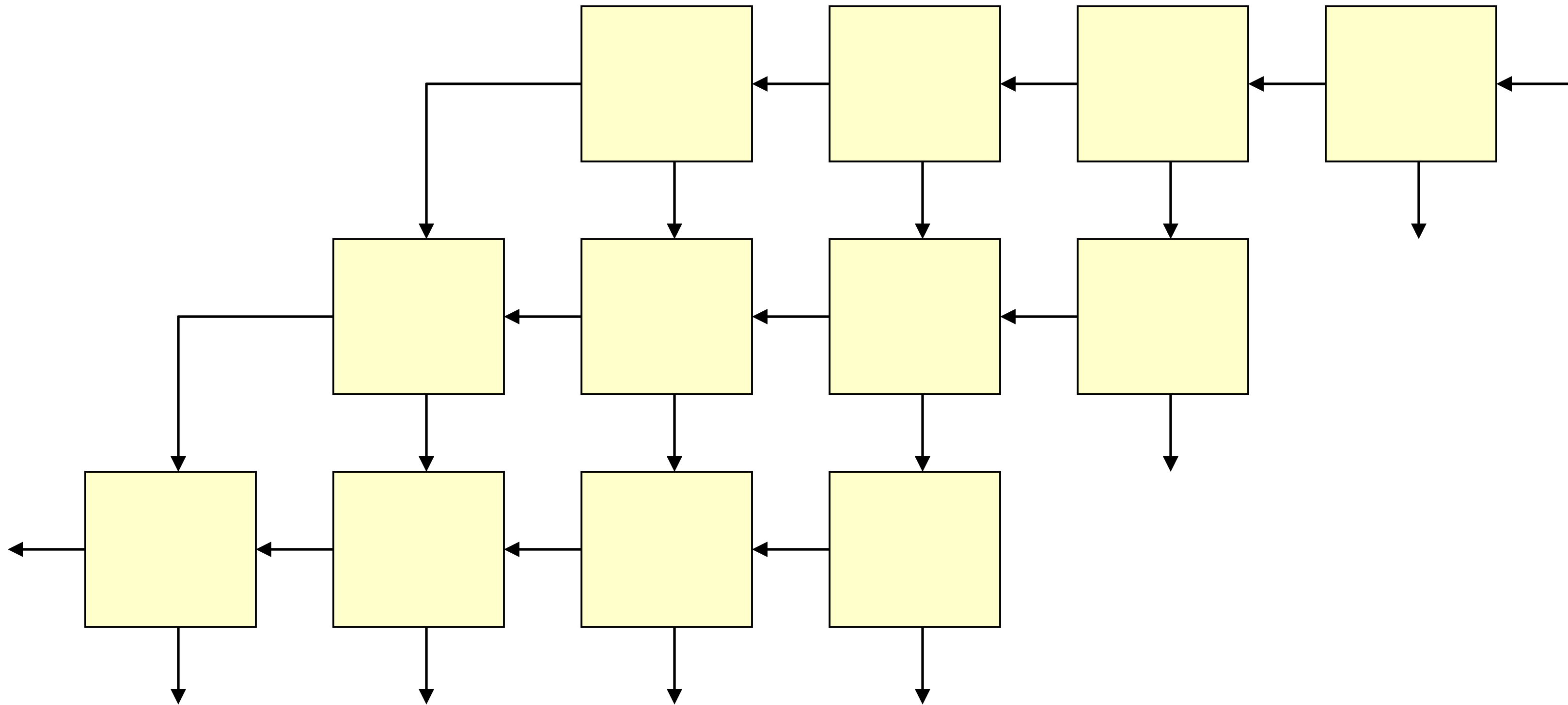
- Critical Path = Longest possible delay path

Assume t_{sum} = 5 ns,

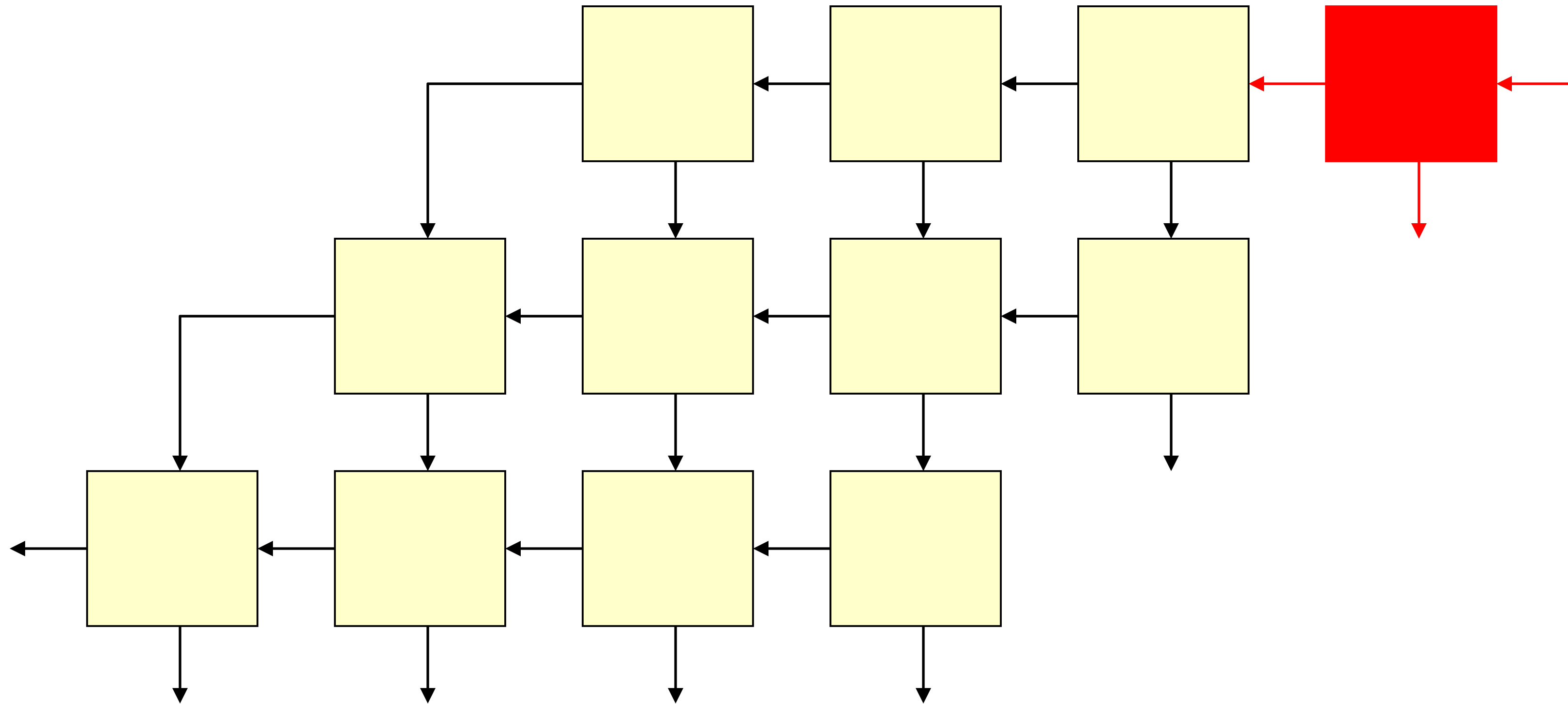
t_{carry}= 4 ns



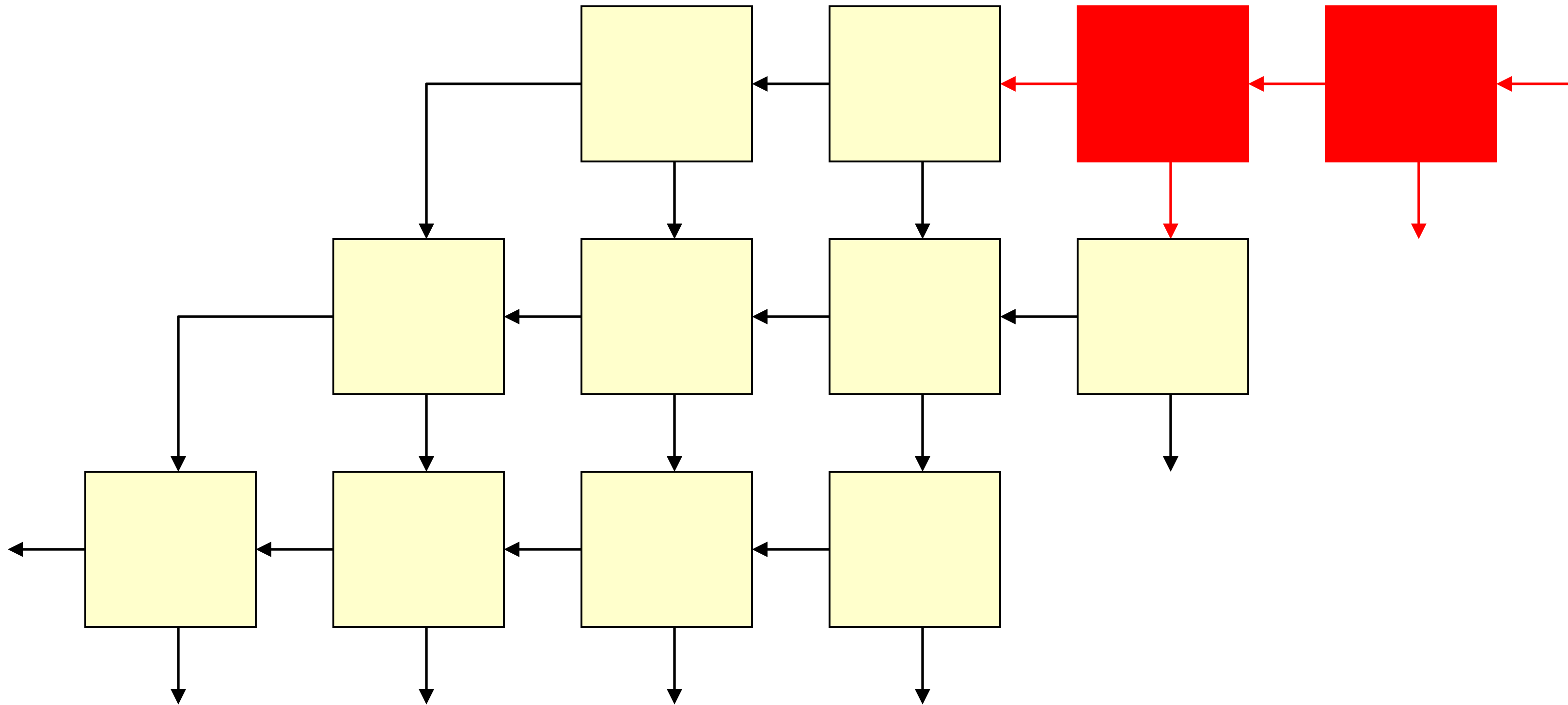
Combinational Multiplier



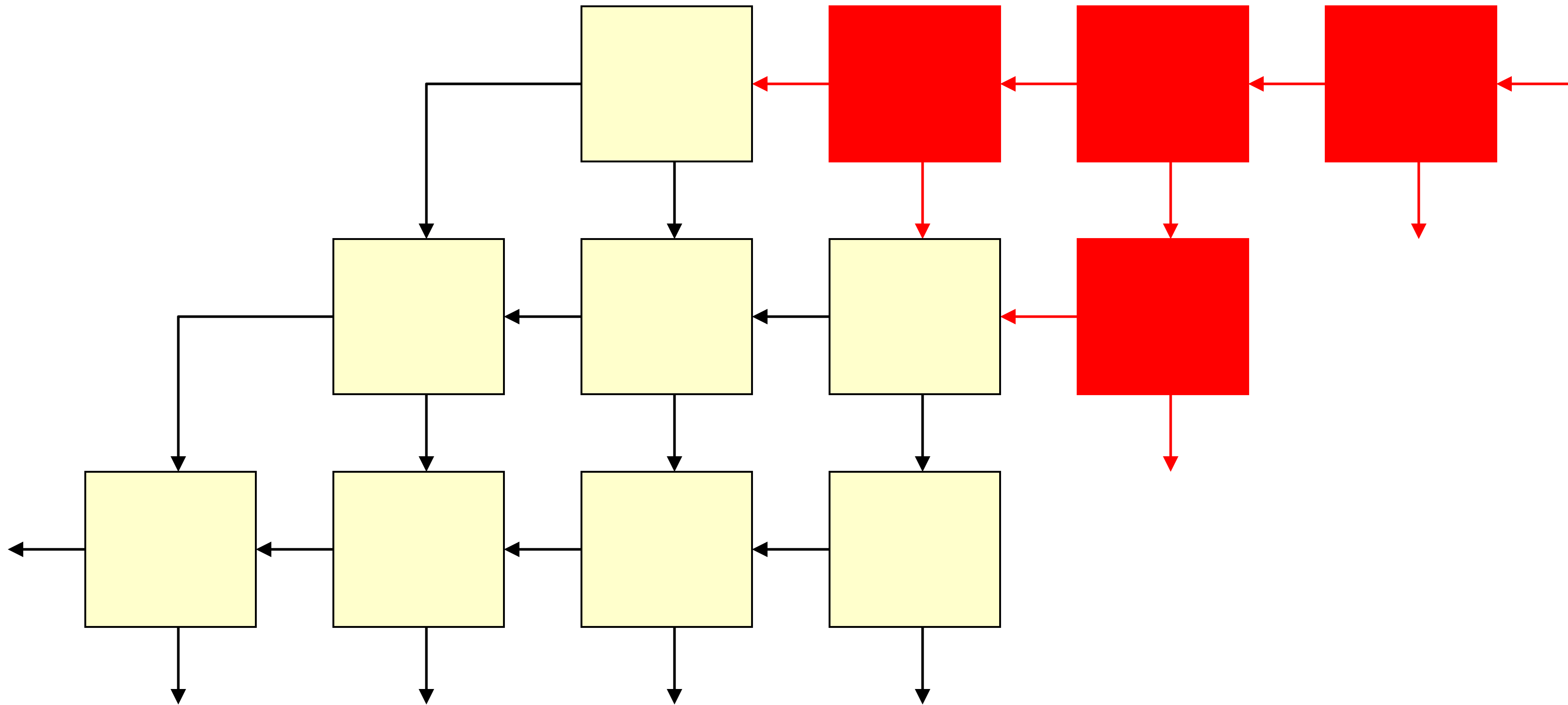
Combinational Multiplier



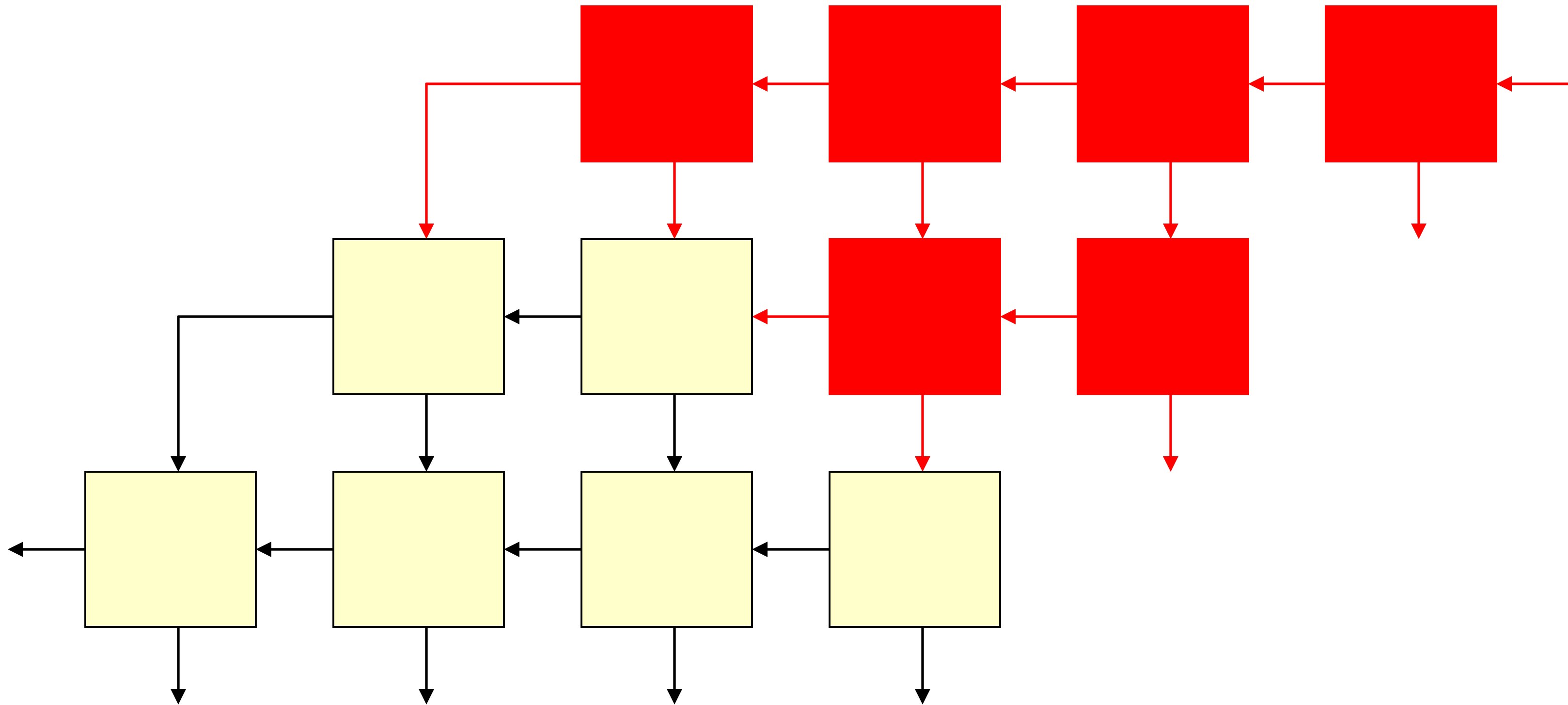
Combinational Multiplier



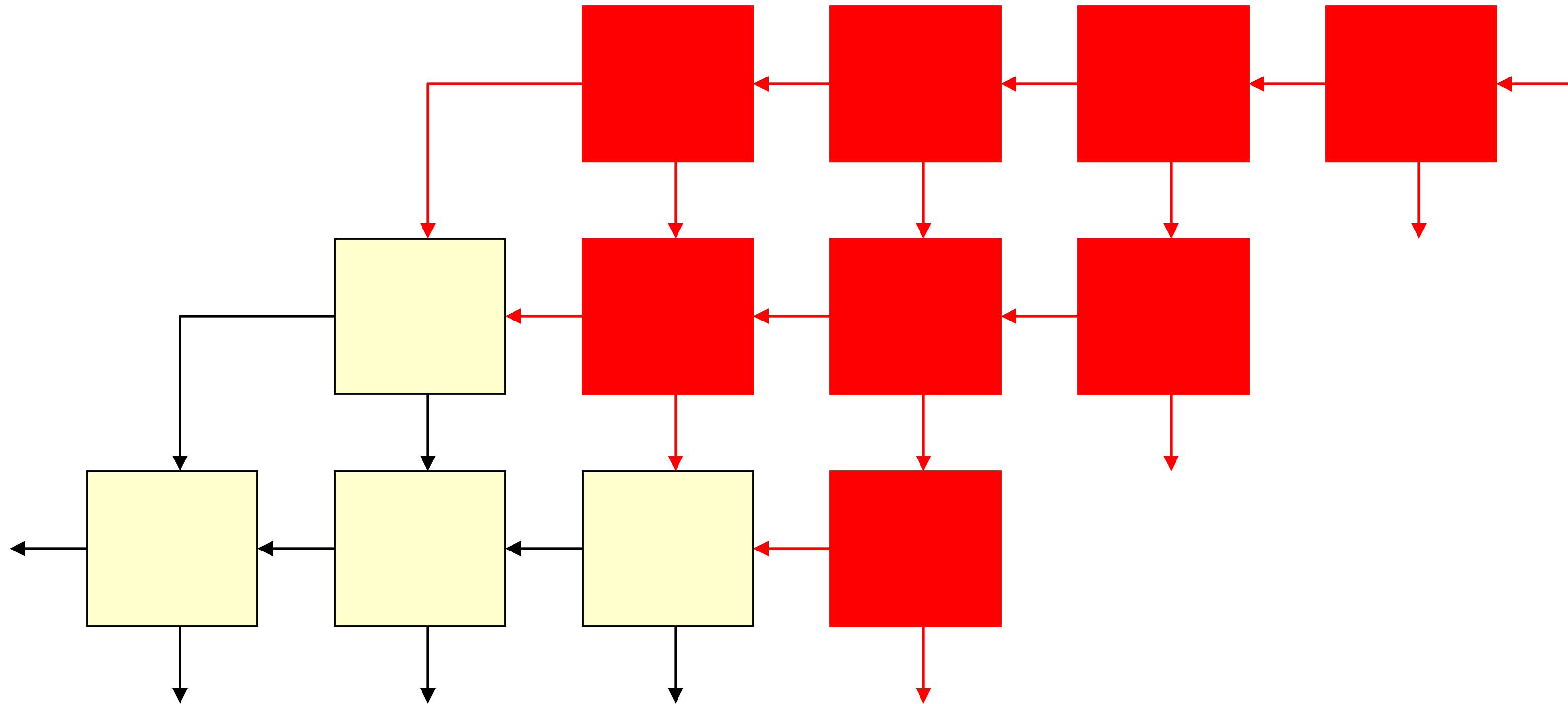
Combinational Multiplier



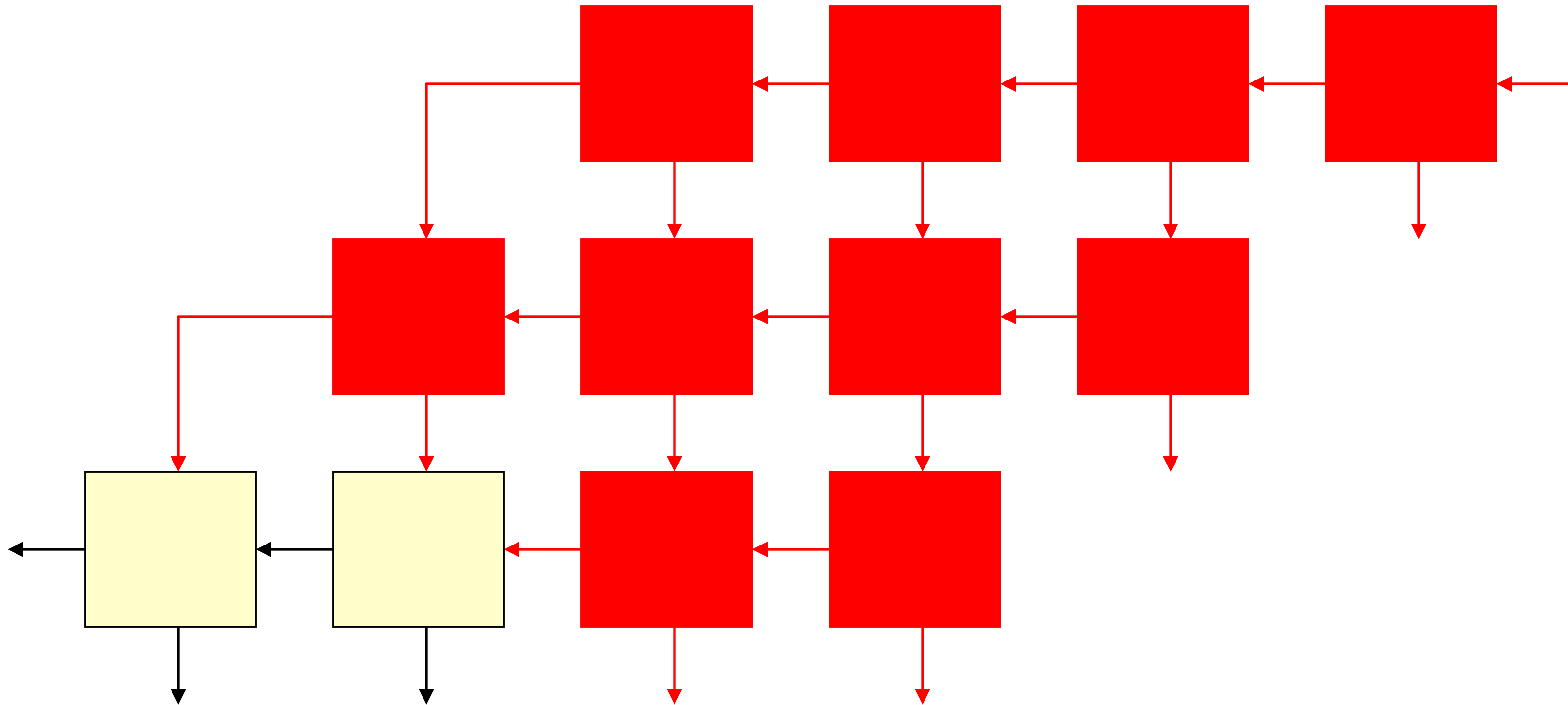
Combinational Multiplier



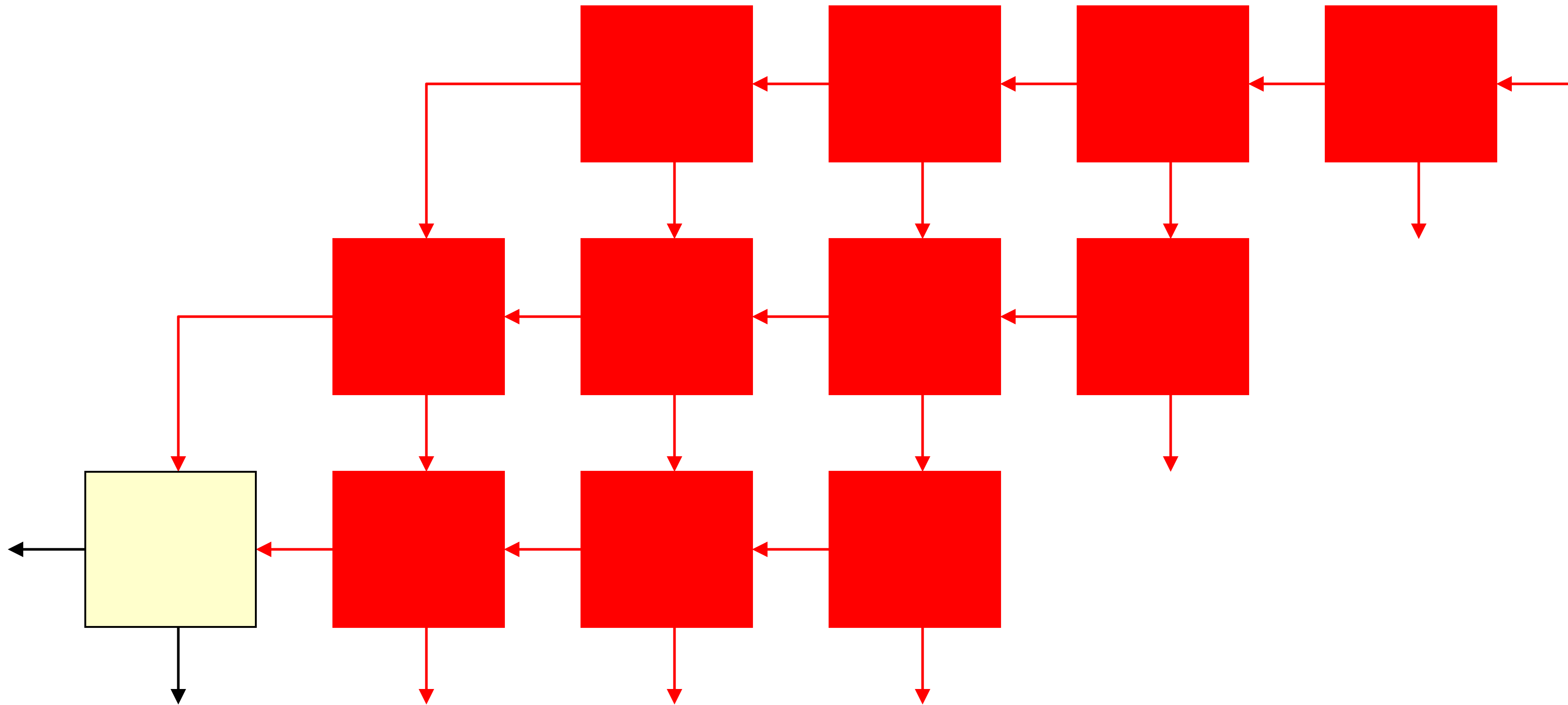
Combinational Multiplier



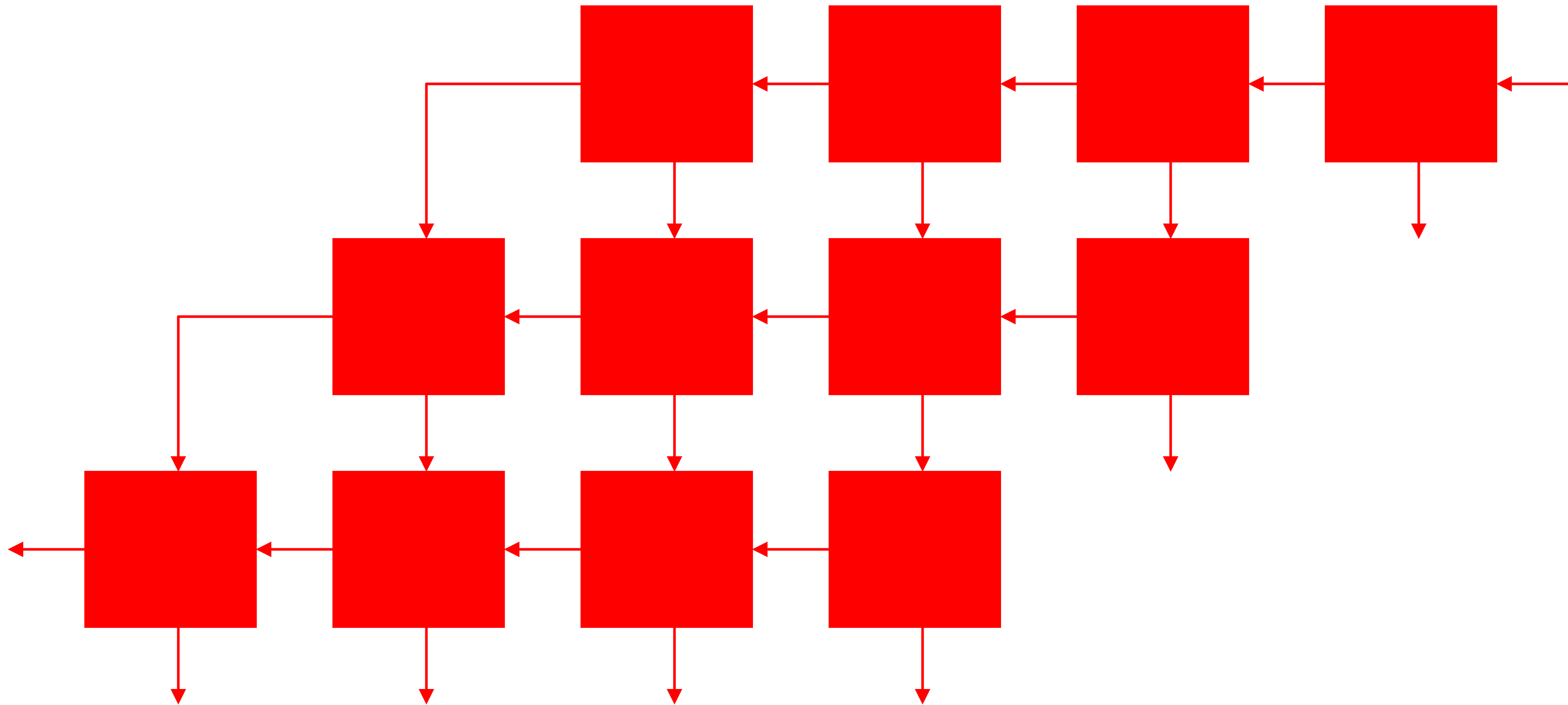
Combinational Multiplier



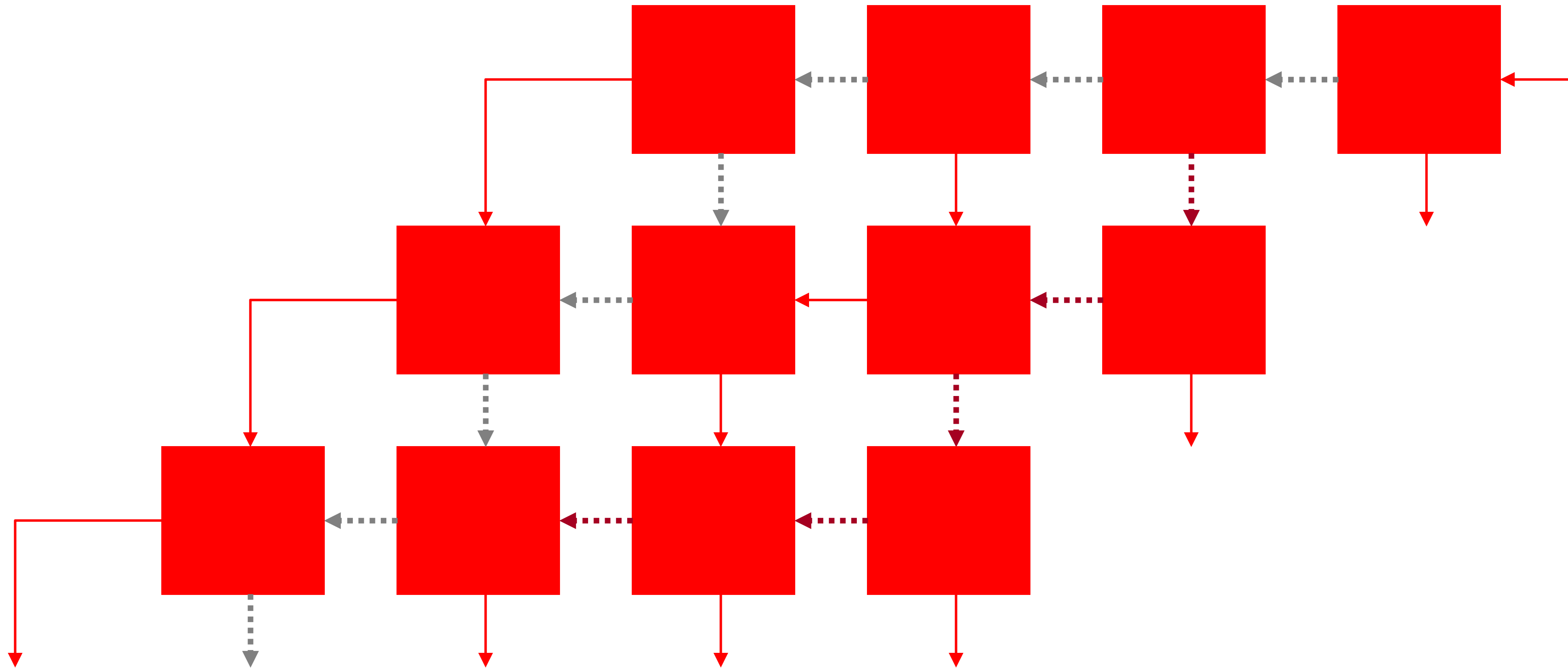
Combinational Multiplier



Combinational Multiplier



Critical Paths



Combinational Multiplier Analysis

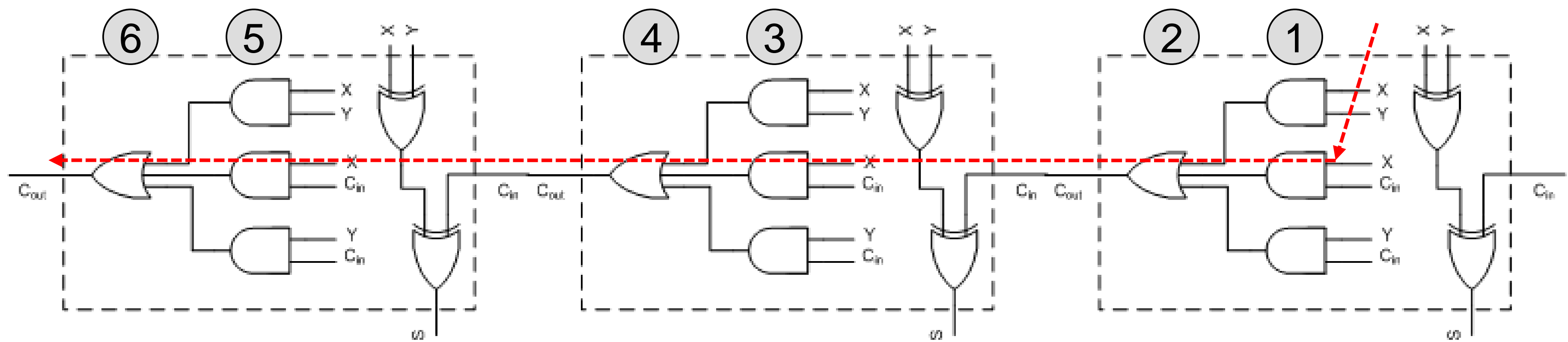
- Large Area due to (n-1) m-bit adders
  - n-1 because the first adder adds the first two partial products and then each adder afterwards adds one more partial product
- Propagation delay is in two dimensions
  - proportional to m+n

Carry-Lookahead Adders

FAST ADDERS

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

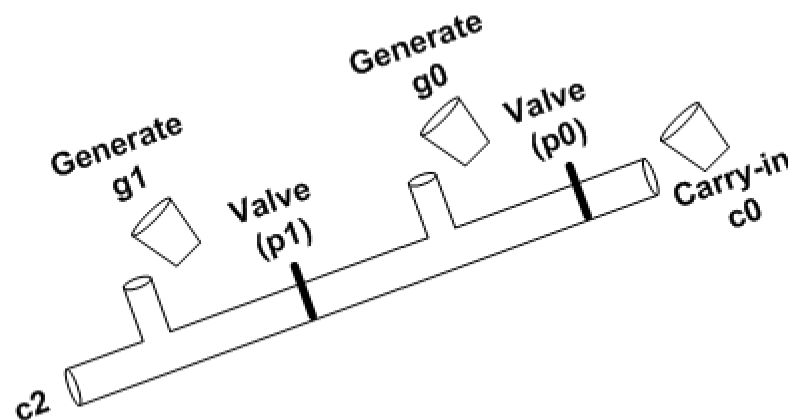
- Define each carry in terms of p_i, g_i and the initial carry-in (c_0) and not in terms of carry chain (intermediate carries: c1,c2,c3,…)
- c1 =
- c2 =
- c3 =
- c4 =

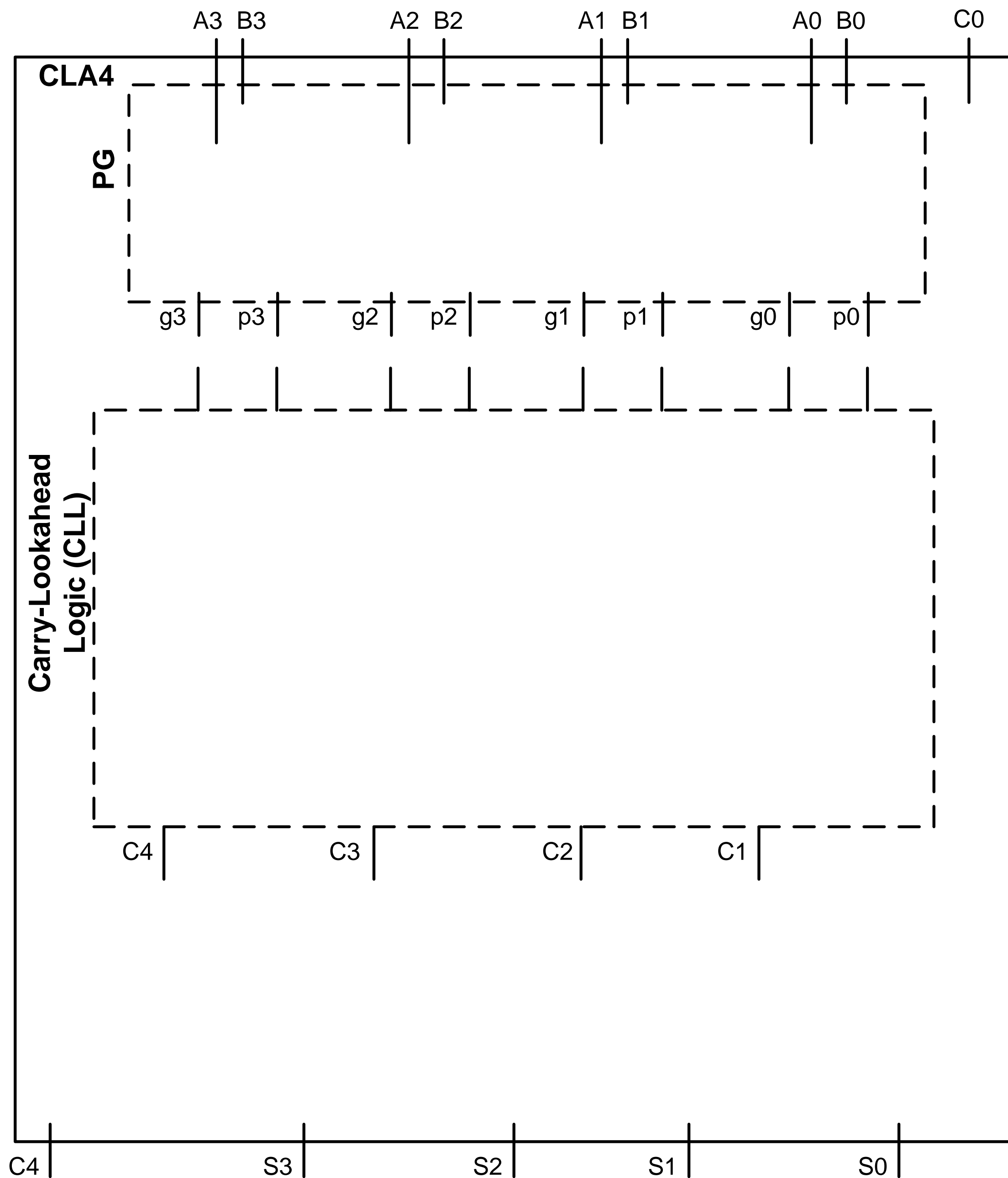
Carry Lookahead Logic

- 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

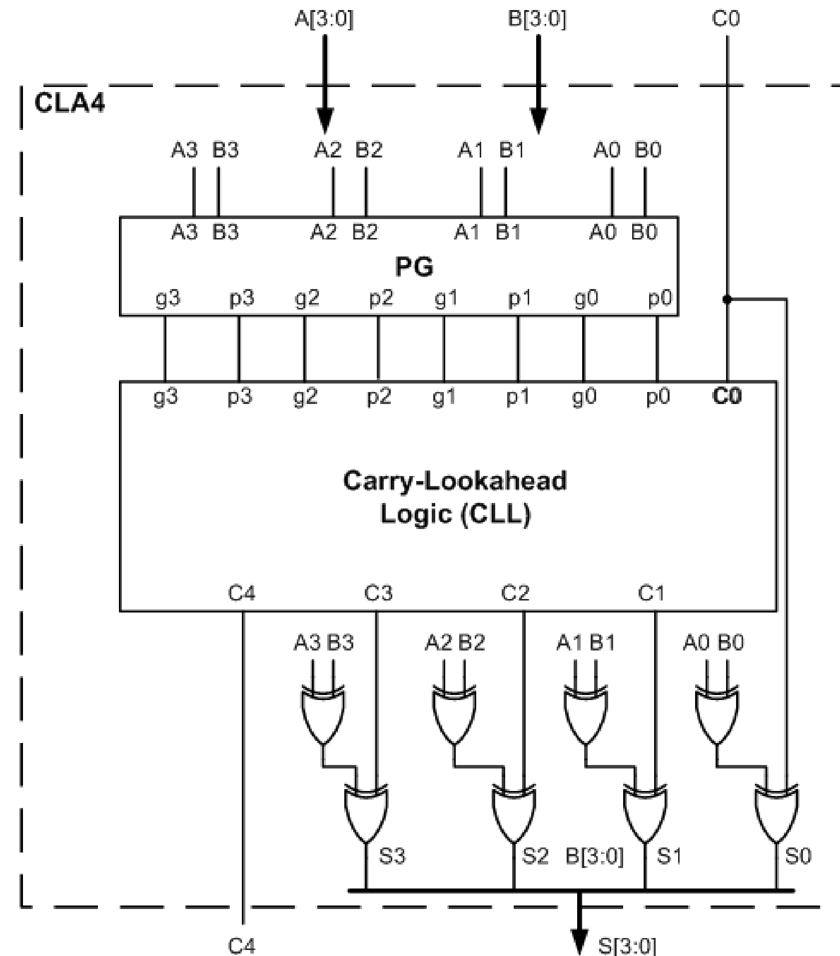
- Consider the carry-chain like a long tube broken into segments. Each segment is controlled by a valve (propagate signal) and can insert a fluid into that segment (generate signal)
- The carry-out of the diagram below will be true if g1 is true or p1 is true and g0 is true, or p1, p0 and c1 is true




A3 B3
A2 B2
A1 B1
A0 B0
C0
CLA4
PG
g3 p3
g2 p2
g1 p1
g0 p0
Carry-Lookahead Logic (CLL)
C4
C3
C2
C1
C4
S3
S2
S1
S0

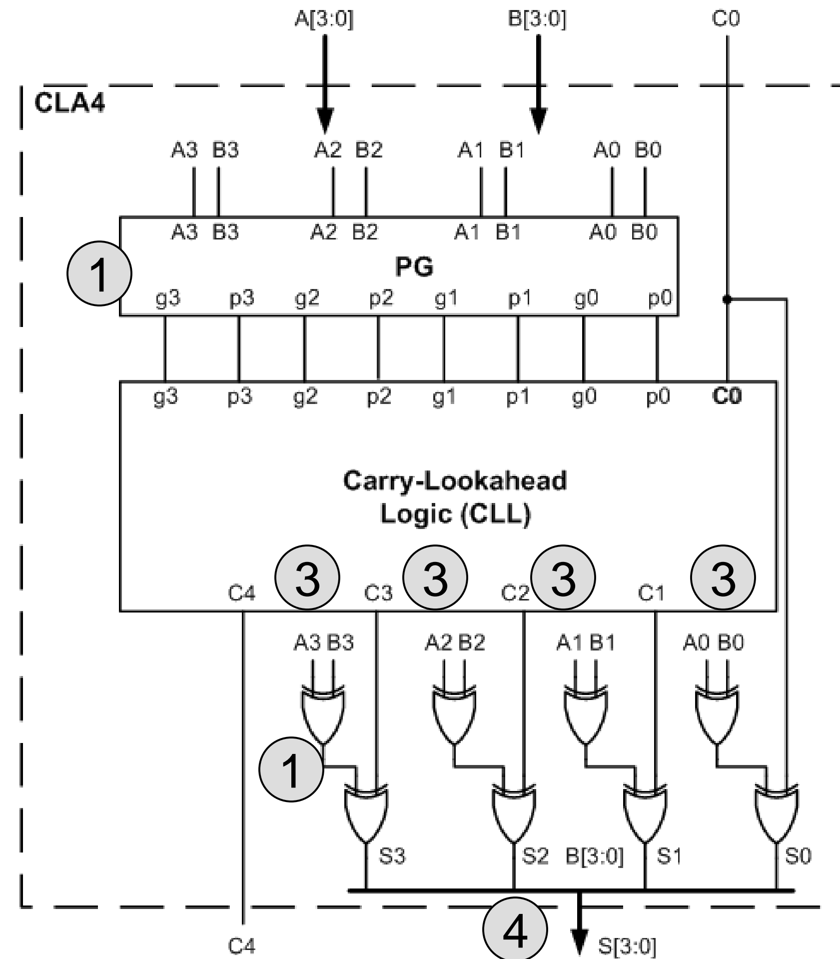
Carry Lookahead Adder

- 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

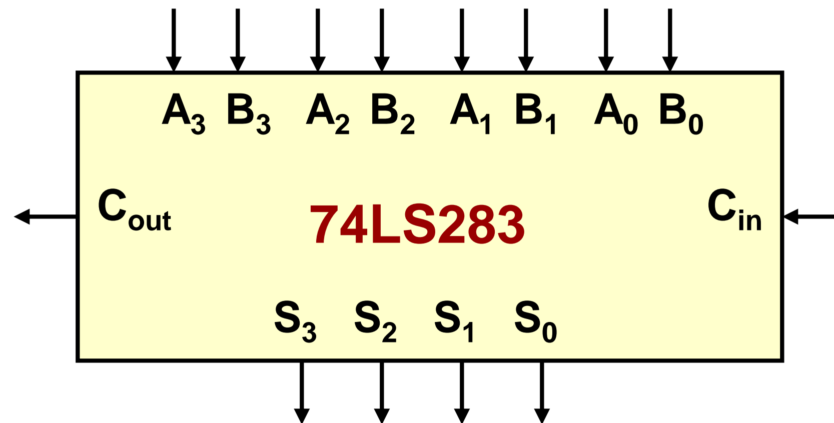
- Use carry-lookahead logic to generate all the carries in one shot and then create the sum
- Example 4-bit CLA shown below



4-bit Adders

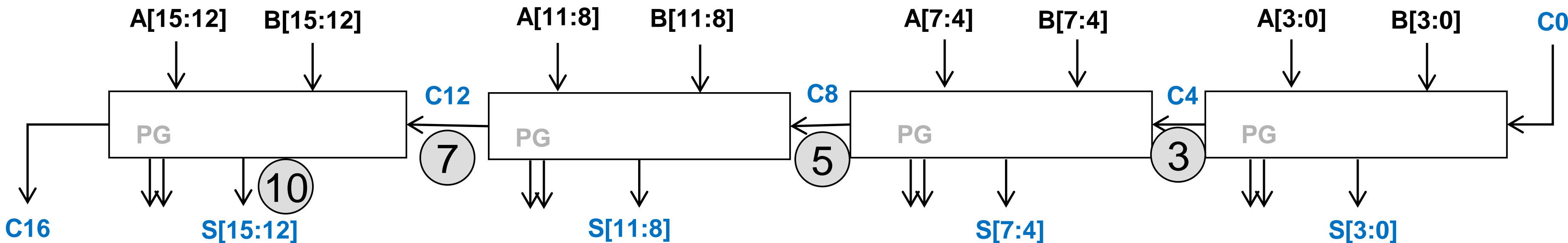
- 74LS283 chip implements a 4-bit adder using CLA methodology

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



16-Bit CLA

- But how would we make a 16-bit adder?
- Should we really just chain these fast 4-bit adders together?
  - Or can we do better?



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

Delay of the above adder design = 3+2+2+3 = 10 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

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

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

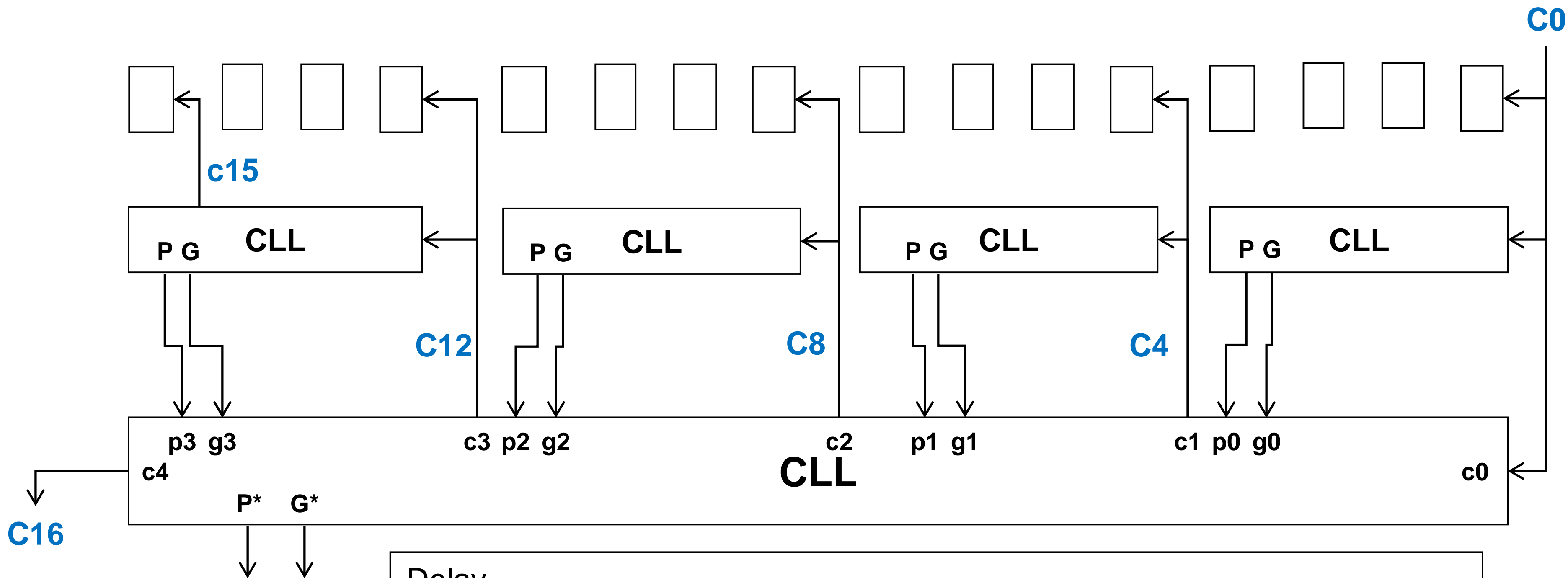
REVIEW ON YOUR OWN FOR CLA LAB

16-bit CLA Closer Look

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

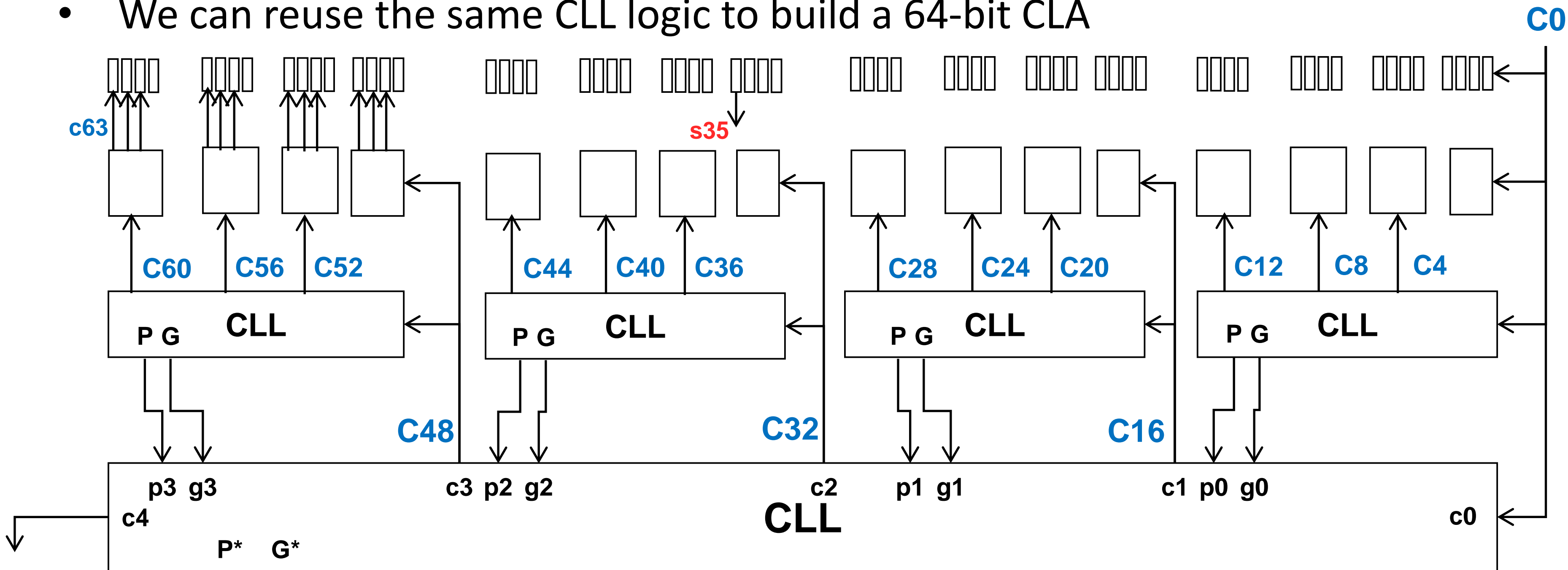
= ____ = Delay in producing C4,C8,C12,C16

= ____ = 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

Summary

- You should now be able to build:
  - Fast Adders
  - Comparators