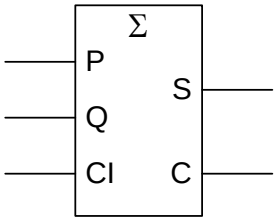
Lecture 13

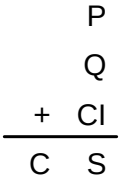
Adder Circuits

Objectives

- Understand how to add both signed and unsigned numbers
- Appreciate how the delay of an adder circuit depends on the data values that are being added together

Full Adder





Output is a 2-bit number counting how many inputs are high

P	Q	CI	C	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

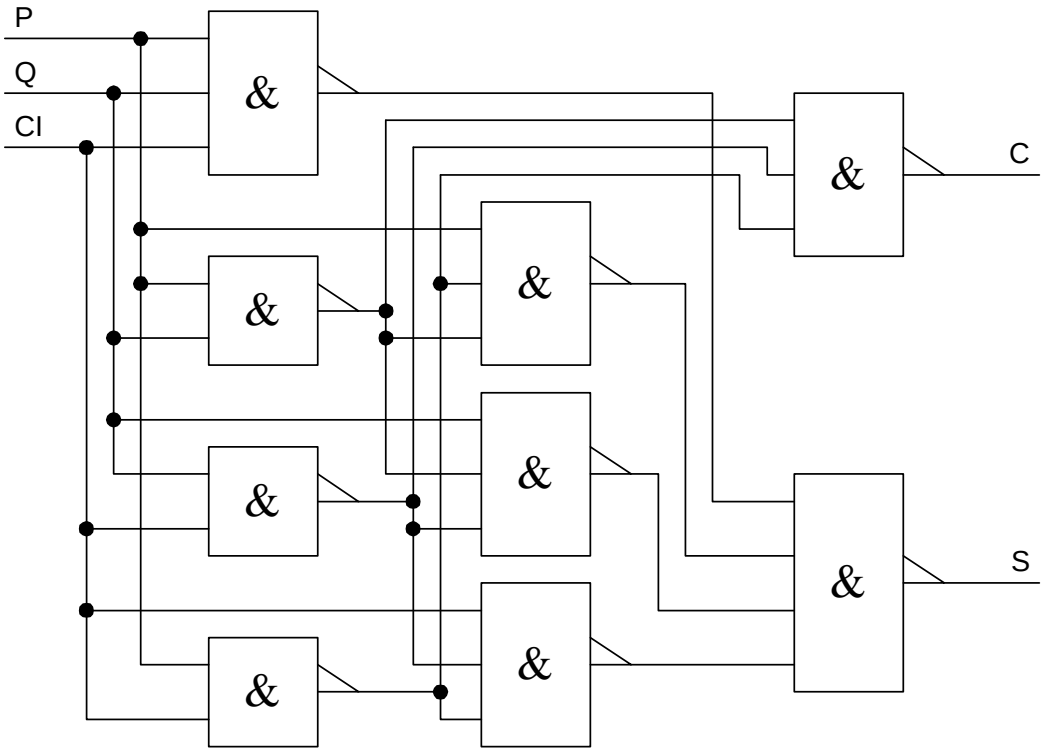
$$C = P \cdot Q + P \cdot CI + Q \cdot CI$$

$$S = P \oplus Q \oplus CI$$

- Symmetric function of the inputs
- Self-dual: Invert all inputs $\Rightarrow$ invert all outputs
  - If *k* inputs high initially then 3–*k* high when inverted
  - Inverting all bits of an *n*-bit number make $x \rightarrow 2^n - 1 - x$
- Note: $P \oplus Q \oplus CI = (P \oplus Q) \oplus CI = P \oplus (Q \oplus CI)$

Full Adder Circuit

9-gate full-adder NAND implementation (do not memorize)



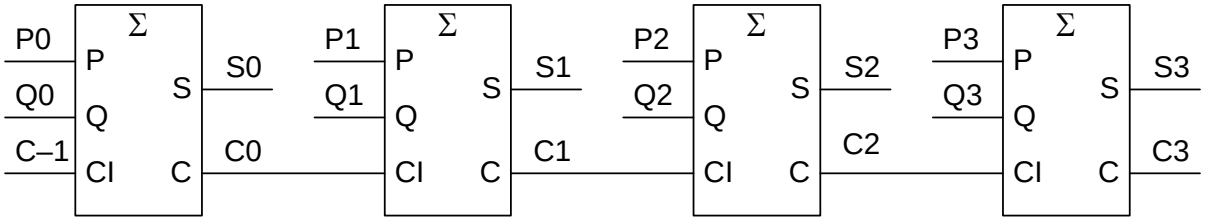
Propagation delays:

From	To	Delay
P,Q or CI	S	3
P,Q or CI	C	2

Complexity: 25 gate inputs $\Rightarrow$ 50 transistors but use of complex gates can reduce this somewhat.

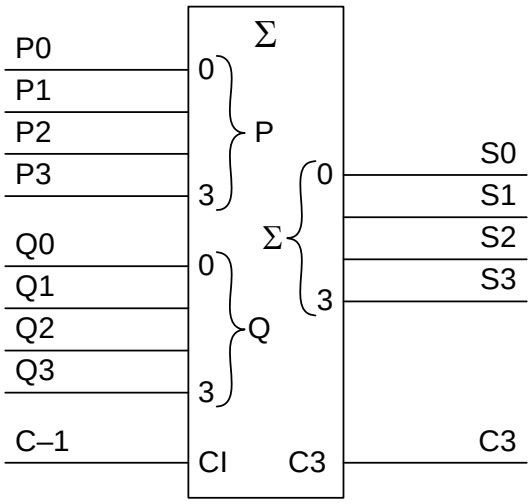
N-bit adder

We can make an adder of arbitrary size by cascading full adder sections:



The main reason for using 2's complement notation for signed numbers is that:

Signed and **unsigned** numbers can use identical circuitry



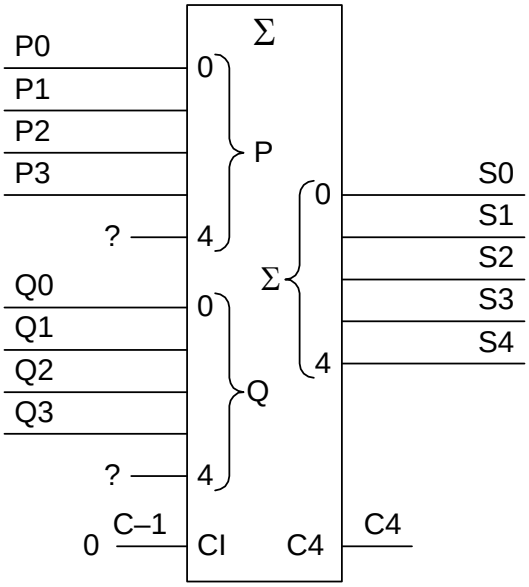
Adder Size Selection

The number of bits needed in an adder is determined by the range of values that can be taken by its **output**.

If we add two 4-bit numbers, the answer can be in the range:

- 0 to 30 for *unsigned* numbers
- -16 to +14 for *signed* numbers

In both cases we need a 5-bit adder to avoid any possibility of overflow:



We need to expand the input numbers to 5 bits. How do we do this ?

Expanding Binary Numbers

Unsigned numbers

Expand an unsigned number by adding the appropriate number of 0's at the MSB end:

5	0101	00000101
13	1101	00001101

Signed numbers

Expand a signed number by duplicating the MSB the appropriate number of times:

5	0101	00000101
–3	1101	11111101

This is known as *sign extension*

Shrinking Binary Numbers

Unsigned

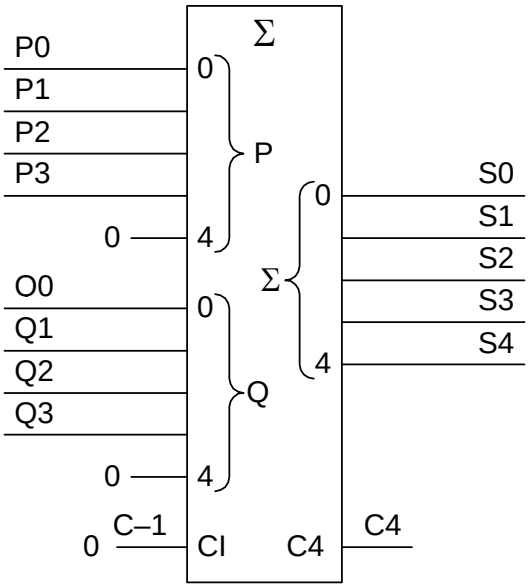
Can delete any number of bits from the MSB end so long as they are all 0's.

Signed

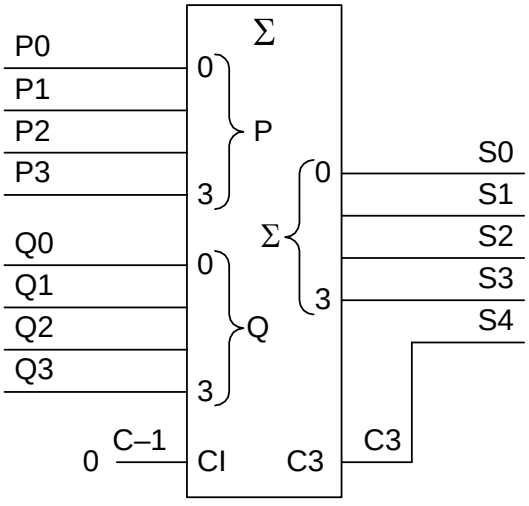
Can delete any number of bits from the MSB end so long as they are all the same as the MSB that remains.

Adding Unsigned Numbers

To avoid overflow, we use a 5-bit adder:



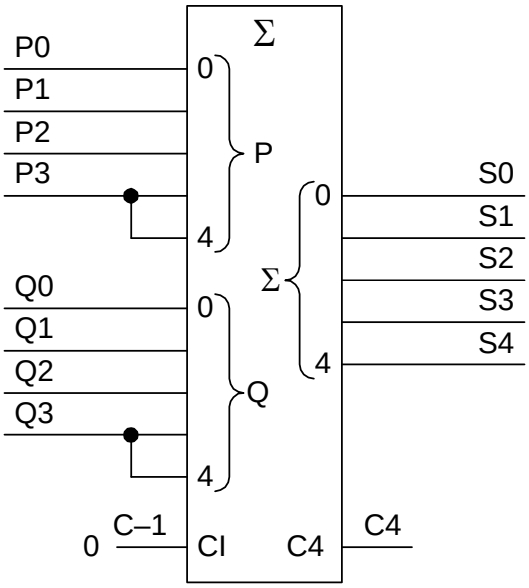
The MSB stage is performing the addition: 0 + 0 + C3. Thus S4 always equals C3 and C4 always equals 0.



We can use a 4-bit adder with C3 as an answer bit.

Adding Signed Numbers

To avoid overflow, we use a 5-bit adder:



This is different from the unsigned case because P4 and Q4 are no longer constants. We cannot simplify this circuit by removing the MSB stage.

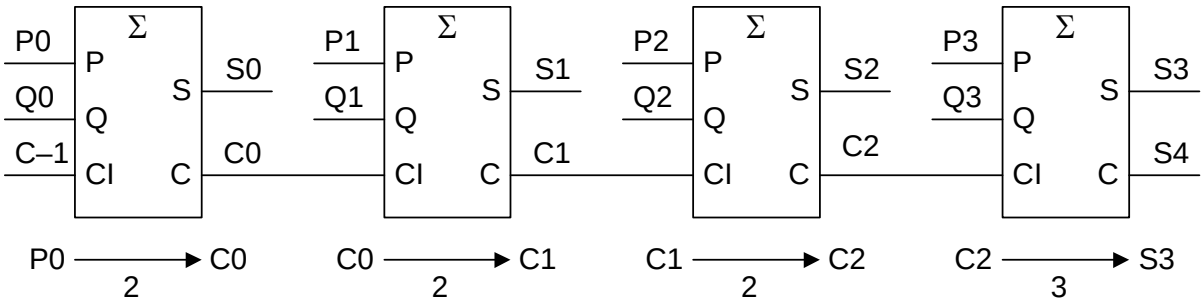
If P and Q have different signs then S4 will not equal C3.

e.g. P=0000, Q=1111
Unsigned P+Q=01111, Signed P+Q=11111

Some minor simplifications are possible:

- If the C4 output is not required, the circuitry that generates it can be removed.
- S4 can be generated directly from P3, Q3 and C3 which reduces the circuitry needed for the last stage.

Adder Propagation Delay



Delays within each stage (in gate delays):

$$P, Q, CI \rightarrow S = 3 \qquad P, Q, CI \rightarrow C = 2$$

Worst-case delay is:

$$P0 \rightarrow C0 \rightarrow C1 \rightarrow C2 \rightarrow S3 = 3\times2 + 3 = 9$$

Note: We also have $Q0 \rightarrow S3 = 9$ and $C{-}1 \rightarrow S3 = 9$

For an N-bit adder, the worst delay is $(N-1)\times2 + 3 = 2N+1$

Example of worst case delay:

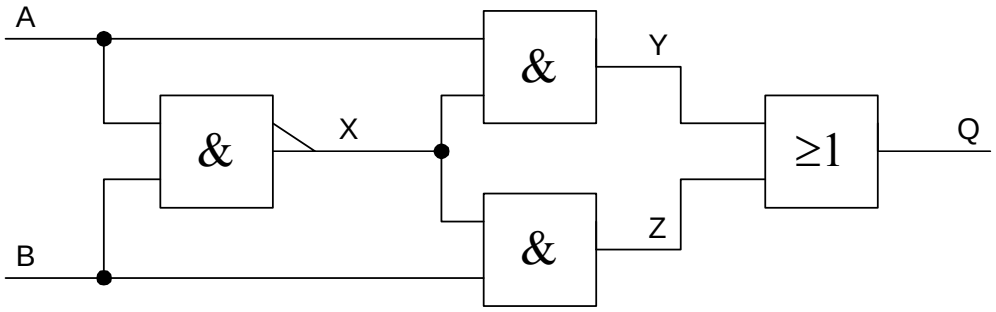
- Initially: P3:0=0000, Q3:0=1111 $\Rightarrow$ S4:0=01111
- Change to: P3:0=0001, Q3:0=1111 $\Rightarrow$ S4:0=10000

Delays are Data-Dependent

To determine the delay of a circuit, we need to specify:

1. The circuit
2. The initial value of all the inputs
3. Which of the inputs changes

Example: What is the propagation delay A→Q ?



Answer 1 (B=0):

- Initially: A=0, B=0 ⇒ X=1, Y=0, Z=0, Q=0
- Then: A↑ ⇒ Y↑ ⇒ Q↑ 2 gate delays

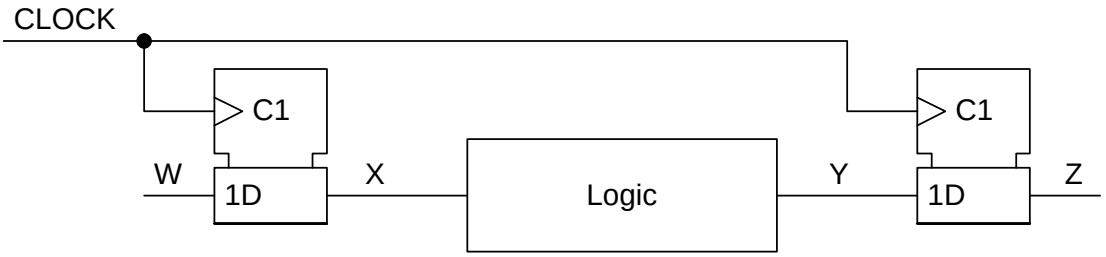
Answer 2 (B=1):

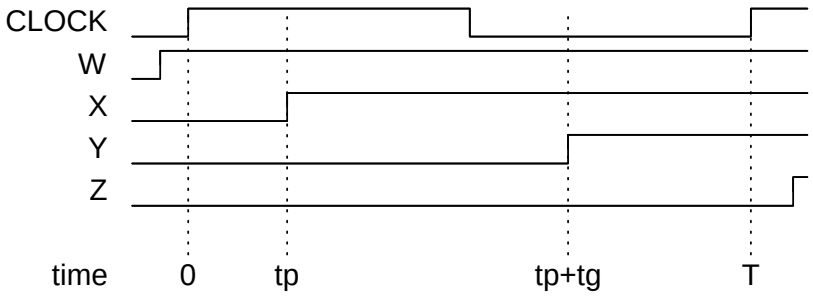
- Initially: A=0, B=1 ⇒ X=1, Y=0, Z=1, Q=1
- Then: A↑ ⇒ X↓ ⇒ Z↓ ⇒ Q↓ 3 gate delays

Worst-Case Delays

We are normally interested only in the worst-case delay from a change in any input to any of the outputs.

The worst-case delay determines the maximum clock speed in a synchronous circuit:





$$tp + tg + ts < T$$

Since the clock speed must be chosen to ensure that the circuit always works, it is only the worst-case logic delay that matters.

Quiz

1. In an full adder, why is it normally more important to reduce the delay from CI to C than to reduce the delay from P to S ?
2. How many bits are required to represent the number A+B if A and B are (a) 8-bit *unsigned* numbers or alternatively (b) 8-bit *signed* numbers.
3. How do you convert a 4-bit *signed* number into an 8-bit signed number ?
4. How do you convert a 4-bit *unsigned* number into an 8-bit signed number ?
5. How is it possible for the propagation delay of a circuit from an input to an output to depend on the value of the other inputs ?

Lecture 14

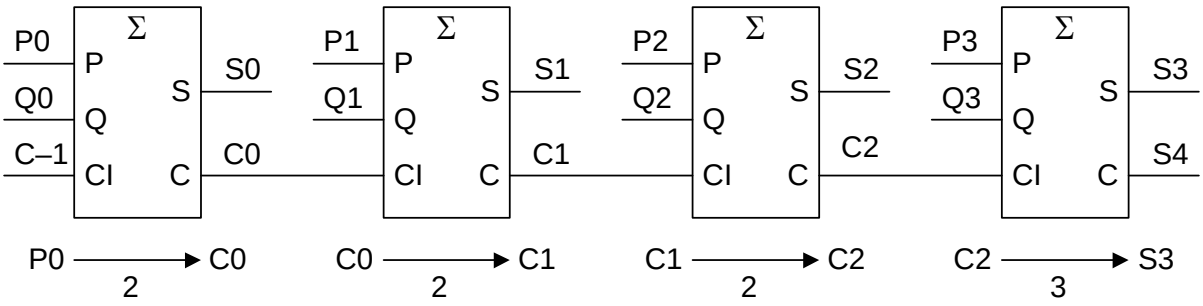
Fast Adder Circuits (1)

Objectives

- Understand how the propagation delay of an adder can be reduced by inverting alternate bits.
- Understand how the propagation delay of an adder can be reduced still further by means of carry lookahead.

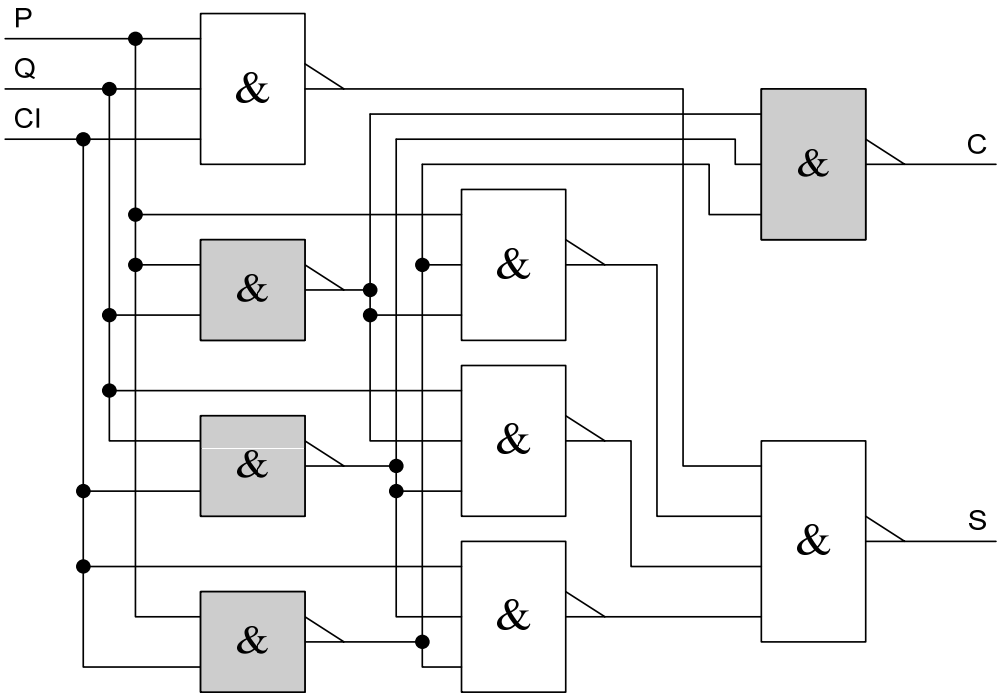
Standard N-bit Adder

Delay of standard N-bit adder = 2N+1



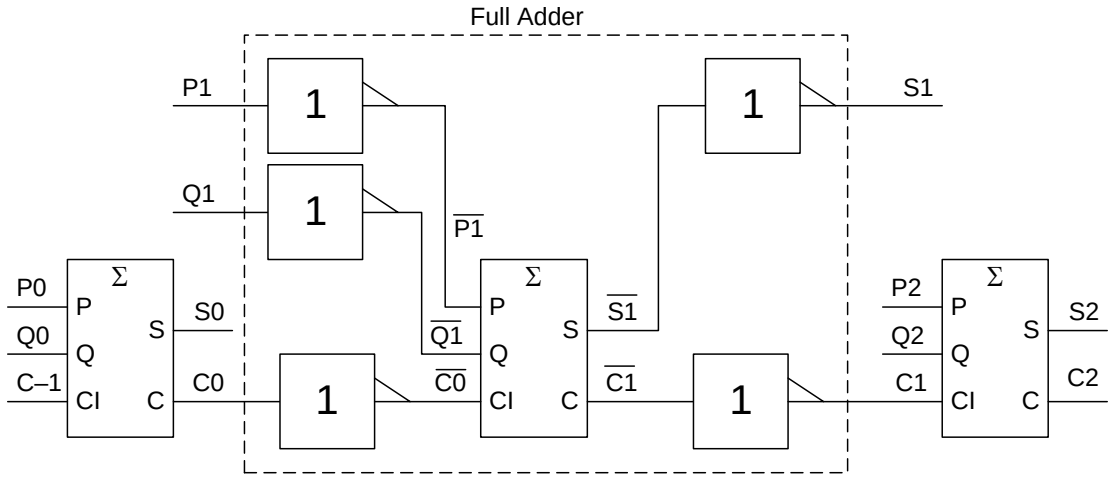
Delay of carry path within each full adder = 2

Carry path consists of three 2-input + one 3-input NANDs

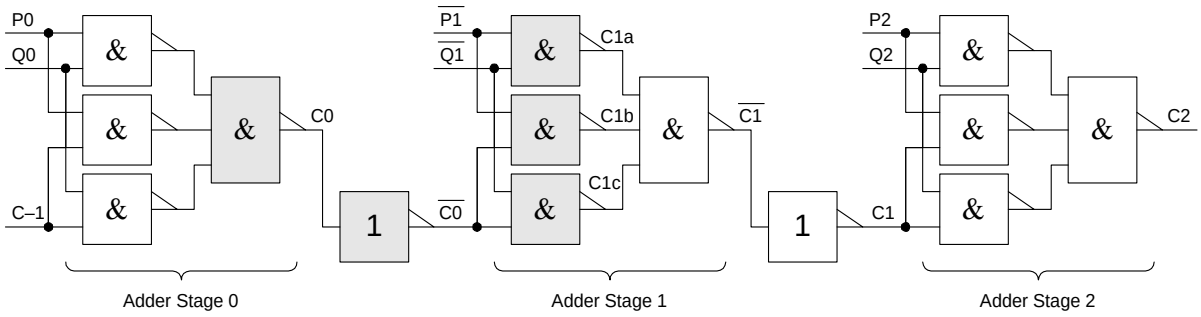


Faster Adder Circuits: 1

Because a full-adder is *self-dual*, it will still work if for alternate stages we invert both the inputs and the outputs:

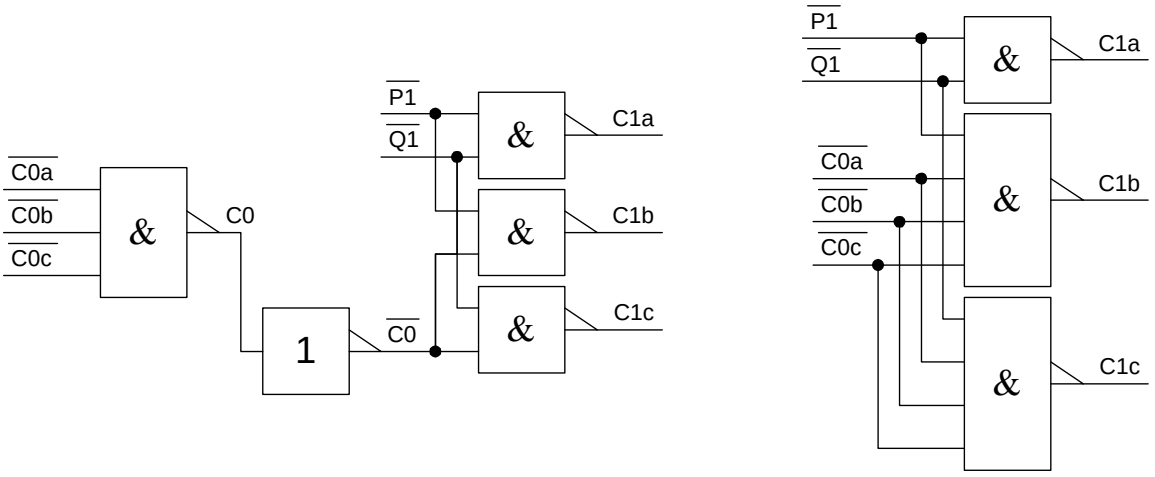


Now consider only the Carry signals:

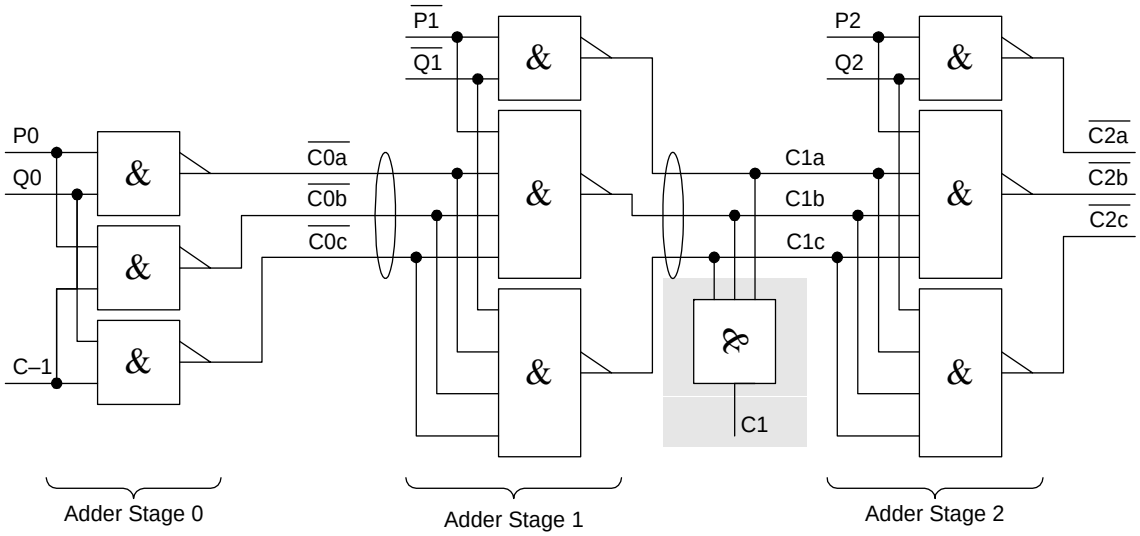


By merging the shaded gates we can reduce the delay to one gate per adder stage.

Fast Adder Circuits: 1 (part 2)



We can merge the 3-NAND and inverter into the final column of gates as shown; this gives one delay per stage:



The signals C1a, C1b, C1c form an *AND-bundle*: C1 is true only if all of them are high. We don't need the signal C1 directly so the shaded gate can be omitted.

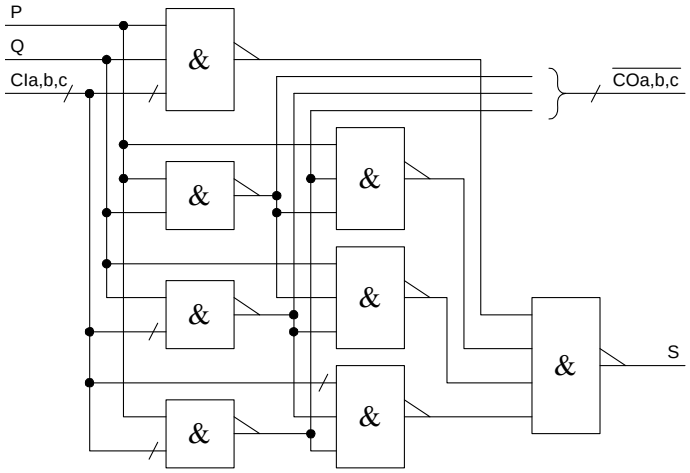
Fast Adder Circuits: 1 (part 3)

Even stages:

Delays:

P,Q,CI $\rightarrow$ S 3

P,Q,CI $\rightarrow$ C 1



30 gate inputs $\Rightarrow$ 60 transistors

Odd stages:

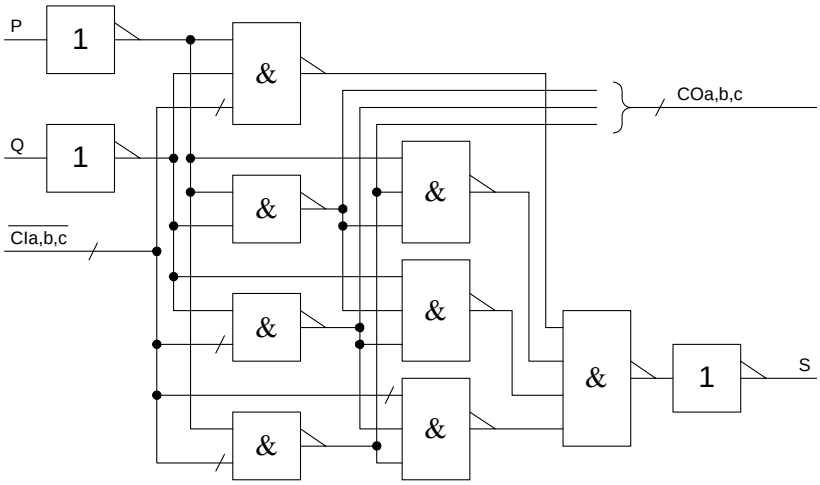
Delays:

P,Q $\rightarrow$ S 5

P,Q $\rightarrow$ C 2

CI $\rightarrow$ S 4

CI $\rightarrow$ C 1



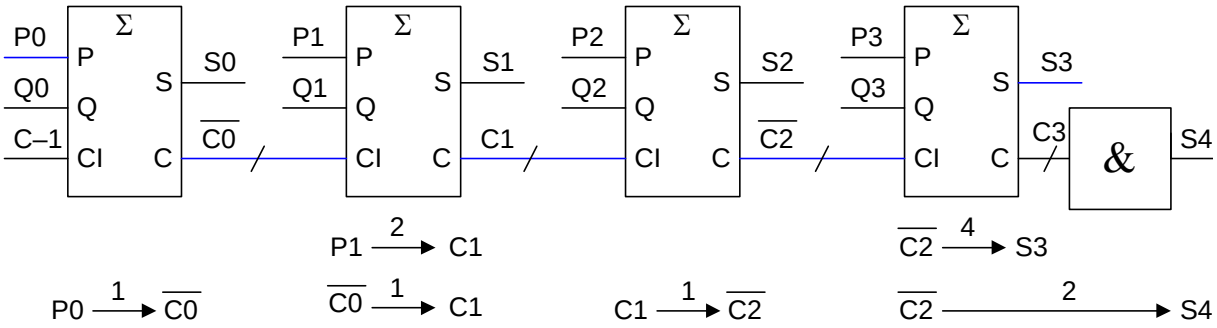
33 gate inputs $\Rightarrow$ 66 transistors

Bundles are denoted by a single wire with a / through it.

22% more transistors but twice as fast.

Fast Adder Circuits: 1 (part 4)

For an N-bit adder we alternate the two modules (with a normalish first stage):



Worst case delay is:

P0 → !C0 → C1 → !C2 → S3 = 7 gate delays

Note that:

- Delay to S4 is shorter than delay to S3
- Delay from P1 is the same as delay from P0
- Worst-case example:

  Initially: P3:0=0000, Q3:0=1111, then P0↑

Delay for N-bit adder (N even) is N+3

(compare with 2N+1 for original circuit)

Carry Lookahead (1)

For each bit of an N-bit adder we get a carry out (CO=1) if two or more of P,Q,CI are equal to 1.

There are three possibilities:

- P,Q=00: C=0 always *Carry Inhibit*
- P,Q=01 or 10: C=CI *Carry Propagate*
- P,Q=11: C=1 always *Carry Generate*

We define three signals:

- CG = P • Q Carry Generate
- CP = P $\oplus$ Q Carry Propagate
- CGP = P + Q Carry Generate or Propagate

We get a carry out from a bit position either if that bit generates a carry (CG=1) or else if it propagates the carry and there is a carry in from the previous bit (CP•CI = 1):

$$C = CG + CP \bullet CI$$

Since CGP = CG + CP, an alternate expression is:

$$C = CG + CGP \bullet CI$$

The second expression is usually used since P + Q is easier and faster to generate than P $\oplus$ Q.

Carry Lookahead (2)

Consider all the ways in which we get a carry out of bit position 3:

1) Bit 3 generates a carry:

```
  1???
+ 1???
```

2) Bit 2 generates a carry and bit 3 propagates it.

```
  11??
+ 01??
```

3) Bit 1 generates a carry and bit 2 propagates it and bit 3 propagates it.

```
  101?
+ 011?
```

4) Bit 0 generates a carry and bit 1 propagates it and bit 2 propagates it and bit 3 propagates it.

```
  1011
+ 0101
```

5) The C–1 input is high and bits 0,1,2 and 3 all propagate the carry.

```
  1011
+ 0100  +1
```

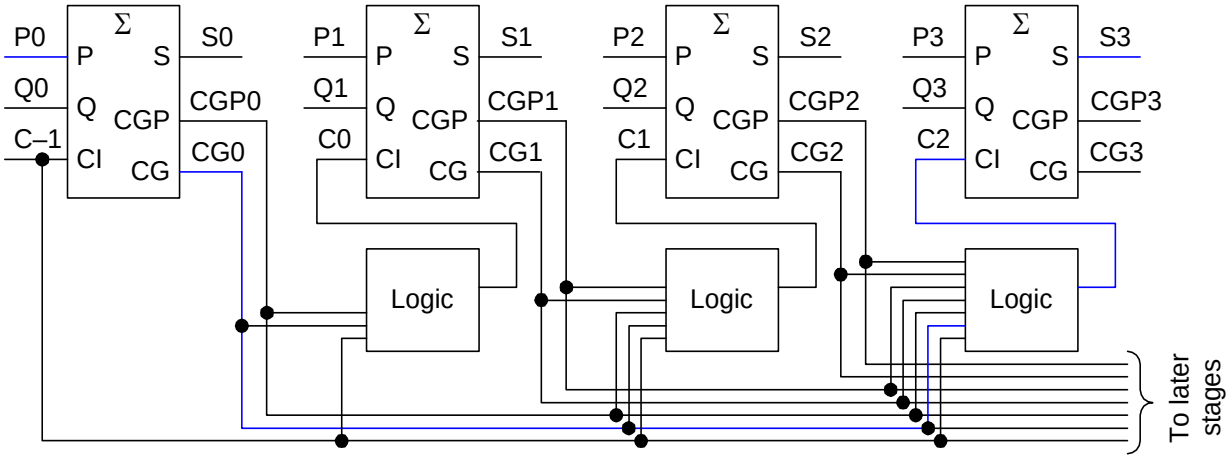
Thus

C3 = CG3 + CP3•CG2 + CP3•CP2•CG1 +
CP3•CP2•CP1•CG0+CP3•CP2•CP1•CP0•C–1

As before, we can use CGPn in place of CPn.

Carry Lookahead (3)

Each stage must now generate CP and CGP instead of C:



$C_0 = CG_0 + CGP_0 \bullet C_{-1}$

$C_1 = CG_1 + CGP_1 \bullet CG_0 + CGP_1 \bullet CGP_0 \bullet C_{-1}$

$C_2 = CG_2 + CGP_2 \bullet CG_1 + CGP_2 \bullet CGP_1 \bullet CG_0 + CGP_2 \bullet CGP_1 \bullet CGP_0 \bullet C_{-1}$

Worst-case propagation delay:

P0 $\rightarrow$ CG0 = 1 gate delay (CG0 = P0•Q0)
CG0 $\rightarrow$ C2 = 2 gate delays (see above expression)
C2 $\rightarrow$ S3 = 3 gate delays (from full adder circuit)

Total = 6 gate delays (independent of adder length)

Carry Lookahead (4)

Carry lookahead circuit complexity for N-bit adder:

- Expression for Cn involves n+2 product terms each containing an average of ½(n+3) input signals.
- Direct implementation of equations for all N carry signals involves approx $N^3/3$ transistors.

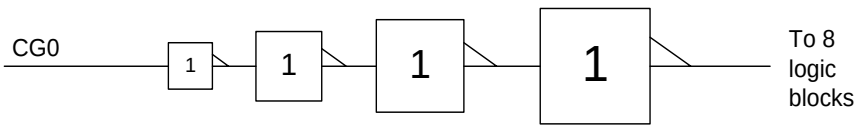
$$N = 64 \Rightarrow N^3/3 = 90{,}000$$

- By using a complex CMOS gate, we can actually generate Cn using only 4n+6 transistors so all N signals require approx $2N^2$ transistors.

$$N = 64 \Rightarrow 2N^2 = 8{,}000$$

Actual gain is not as great as this because for large n, the expression for Cn is too big to use a single gate.

- C–1, CG0 and CGP0 must drive N–1 logic blocks. For large N we must use a chain of buffers to reduce delay:



The circuit delay is thus not quite independent of N.

Quiz

1. What does it mean to say that a full-adder is *self-dual* ?
2. How does placing an inverter between each stage of a multi-bit adder allow the merging of gates in consecutive stages ?
3. In a 4-bit adder, give an example of a propagation delay that *increases* when alternate bits are inverted.
4. Why is a carry-lookahead adder generally implemented using CGP rather than CP outputs ?

Lecture 15

Fast Adder Circuits (2)

Objectives

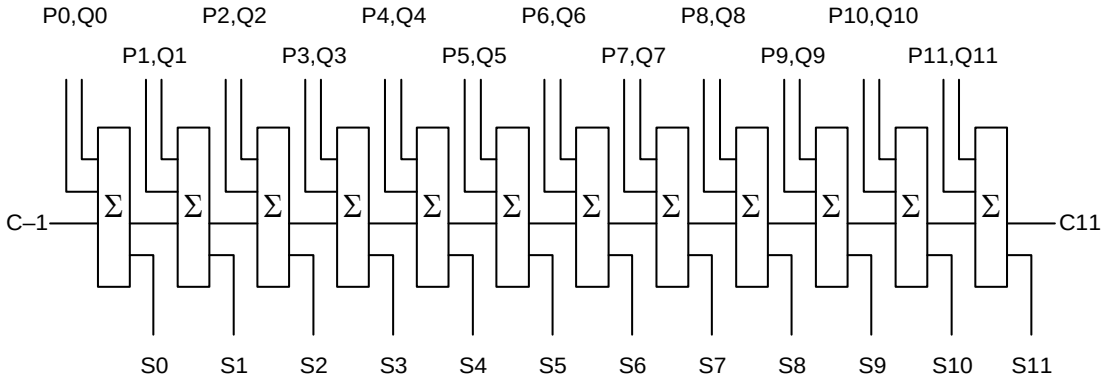
- Understand the *carry skip* technique for reducing the propagation delay of an adder circuit.
- Understand how the *carry save* technique can be used when adding together several numbers.

Summary So Far:

- Cascading full adders:
  2N+1 gate delays, 50N transistors
- Use self-duality to invert odd-numbered stages:
  N+3 gate delays, 61N transistors
- Carry lookahead:
  6 gate delays, between $2N^2$ and $0.3N^3$ transistors

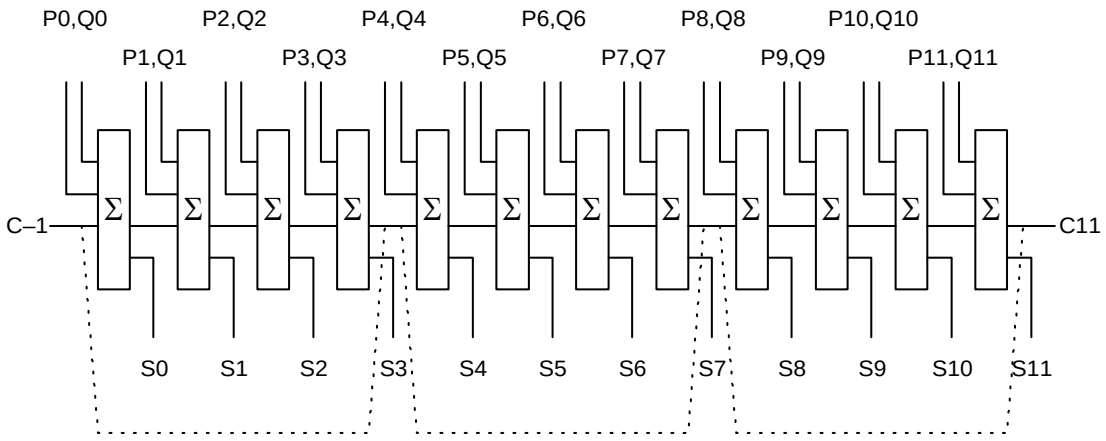
Carry Skip (1)

Consider a 12-bit adder:



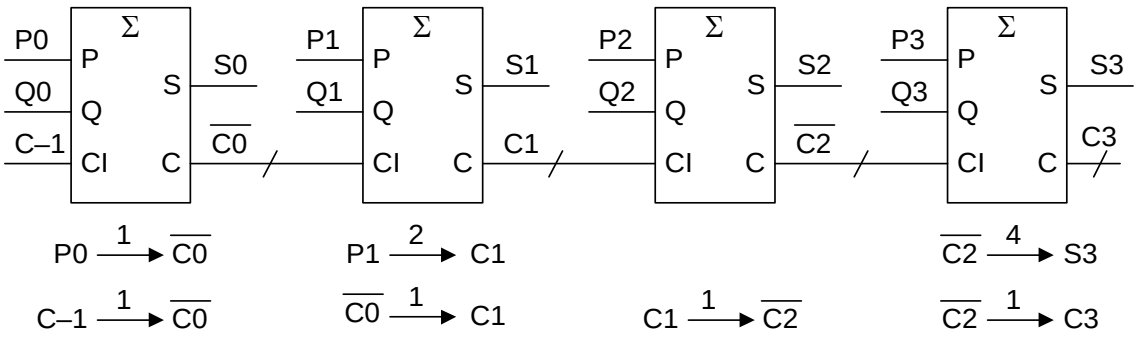
The worst-case delay path is from C–1 to S11.

In *carry skip*, we speed up this path by allowing the carry signal to skip over several adder stages at a time:



Carry Skip (2)

Consider our fast adder circuit without carry lookahead (but using alternate-bit inversion):



There are two possible sorts of addition sum:

- *All bits propagate the carry* $\Rightarrow C_3 = C_{-1}$:

0101	0101
1010	1010
0	1
01111	10000

$C_{-1}\uparrow \rightarrow C_3\uparrow$ = 4 gate delays

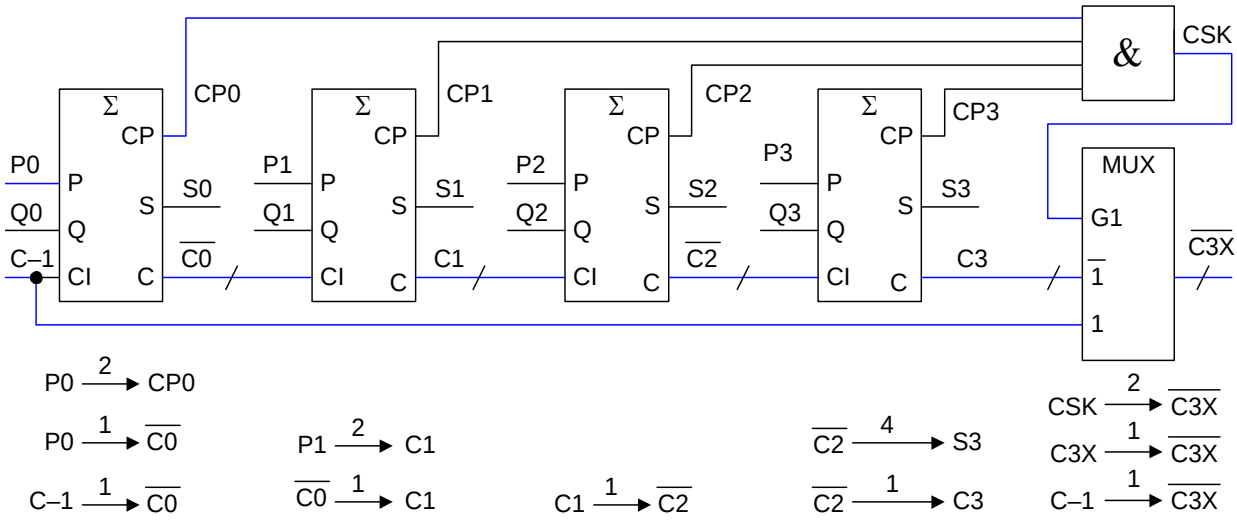
- *At least one bit doesn't propagate the carry*
  $\Rightarrow C_3$ is <u>completely independent</u> of C_{-1}:

0101	0101
1110	1110
0	1
10011	10100

$C_{-1}\uparrow \rightarrow C_3$ = 0 gate delays

Carry Skip (3)

We speed up C–1 → C3 by detecting when all bits propagate the carry and using a multiplexer to allow C–1 to skip all the way to C3:



Calculate Carry Propagate (CP = P ⊕ Q) for each bit. Call this 2 gate delays since XOR gates are slow. CSK=1 if all bits propagate the carry.

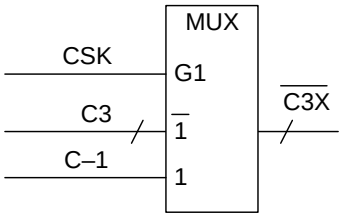
- Case 1: All bits propagate the carry
  C–1 → !C3X = 1 gate delay (via multiplexer)

- Case 2: At least one bit inhibits or propagates the carry
  ⇒ **C–1 does not affect C3**

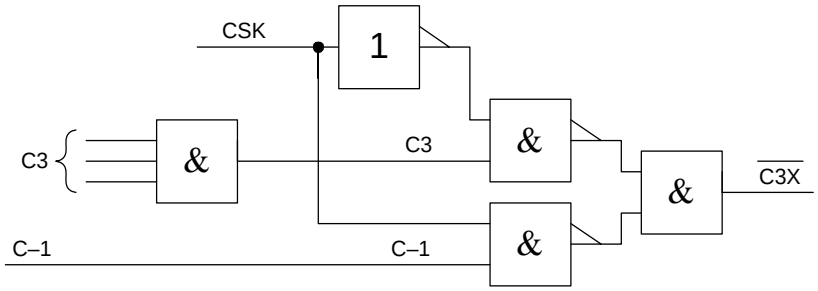
  Longest delays to !C3X and S3:
  - P0 →!C3X = 5 (via either !C0 or CSK)
  - P0 →S3 = 7

Carry Skip (4)

Multiplexer Details

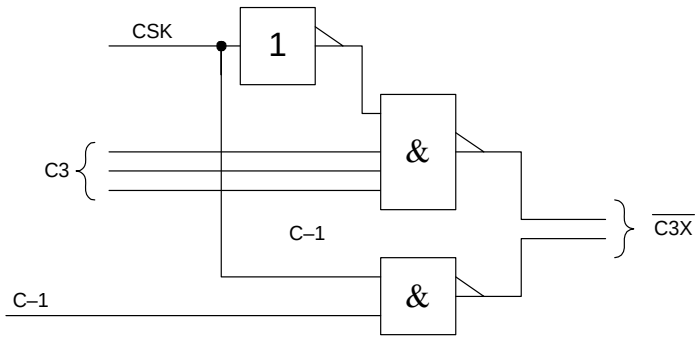


CSK	!C3X
0	!C3
1	!C–1



We merge both AND gates:

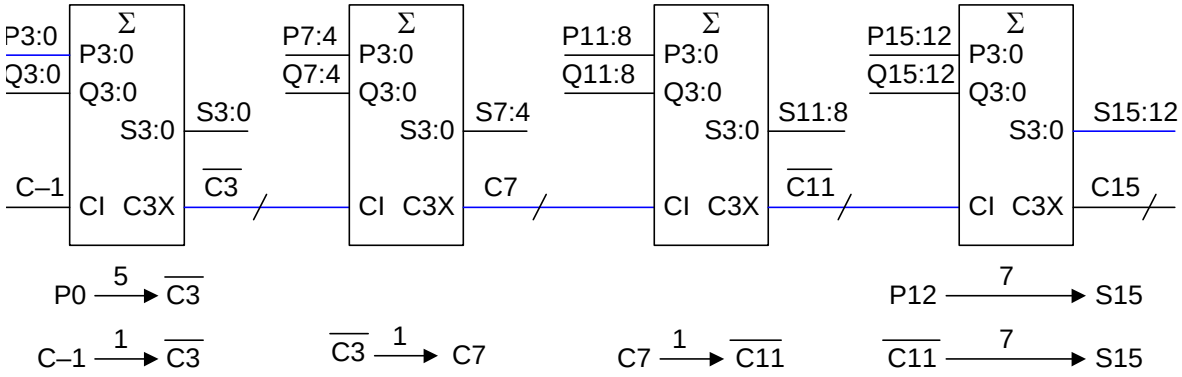
- the 3-AND gate merges into the following NAND
- the 2-AND gate merges into the next adder stage



C–1 → !C3X now equals 1 gate delay.

Carry Skip (5)

Combine 4 blocks to make a 16-bit adder:



Worst-case delay is:

$$P0 \rightarrow !C3 \rightarrow C7 \rightarrow !C11 \rightarrow S15 = 14 \text{ gate delays}$$

Each additional block of 4 bits gives a delay of only 1 gate delay: this corresponds to ¼ gate delay per bit.

For an N-bit adder we have a delay of ¼N+10. We can reduce this still further by having larger super-blocks.

<u>Carry circuit delays</u>:

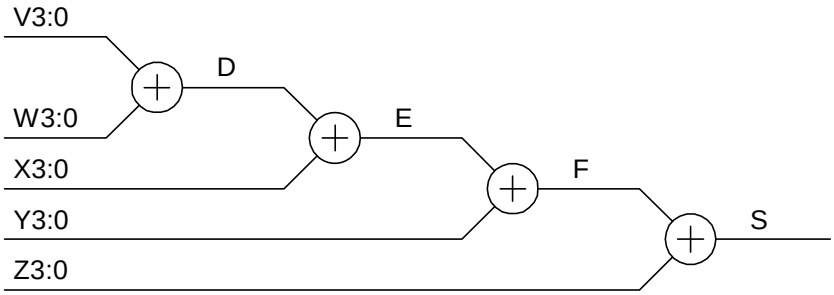
Simple	2N+1
Bit-inversion	N+3
Carry Skip	¼N+10
Carry Lookahead	6

- but lots of circuitry and high gate fanout ⇒ more delay

Adding lots of numbers

In multiplication circuits and digital filters we need to add lots of numbers together.

Suppose we want to add together five four-bit unsigned numbers: V, W, X, Y and Z.



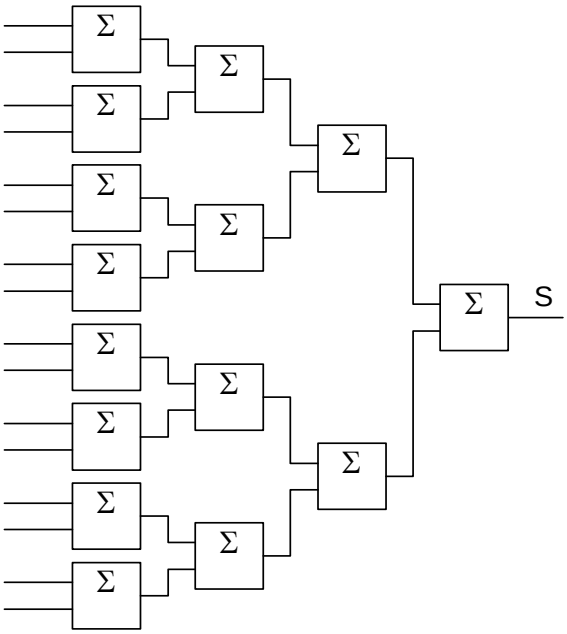
If we use carry-lookahead adders, each stage will have 6 gate delays.

Total delay to add together K values will be (K–1) × 6.

Thus K=16 gives a delay of 90 gate delays.

Addition Tree

In practice we use a tree arrangement of adders:



Number of values, K	16	8	4	2	1
$\log_2(K)$	4	3	2	1	0

Each column of adders adds a delay of 6 and halves the number of values needing to be added together.

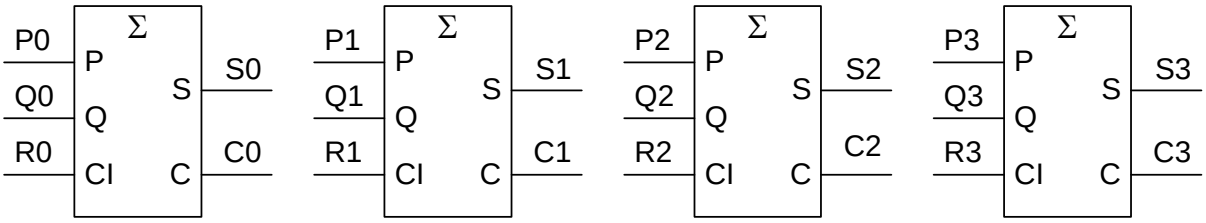
Equivalently, each column of adders reduces $\log_2 K$ by one.

Hence the total delay is is $\log_2 K \times 6$ giving a delay of 24 to add together 16 values.

The total number of adders required is still K–1 as before.

Carry-Save Adder

Take a normal 4-bit adder but *don't connect up the carrys*:

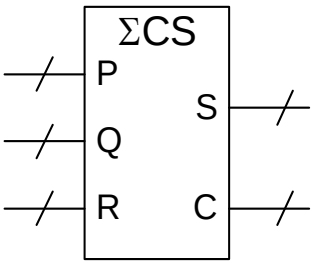


We have P+Q+R = 2C + S

E.g. P=9, Q=12, R=13
gives C=13, S=8

P:	1001
Q:	1100
R:	1101
S:	1000
C:	1101_

We call this a *carry-save* adder: it reduces the addition of 3 numbers to the addition of 2 numbers.

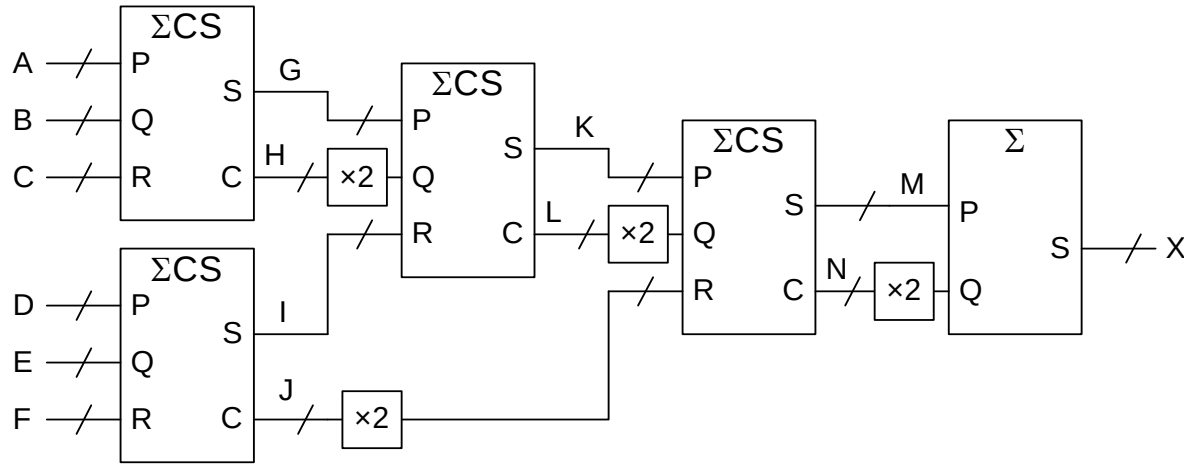


The propagation delay is 3 gates regardless of the number of bits. The amount of circuitry is much less than a carry-lookahead adder.

The circuit reduces $\log_2 K$ by 0.585 (from 1.585 to 1.0) for a delay of 3. The overall delay we can expect is therefore $\log_2 K \times 3/0.585 = \log_2 K \times 5.13$. This is *better* than carry lookahead for *less circuitry*.

Carry Save Example

We will calculate: 13+10+5+11+12+1 = 52

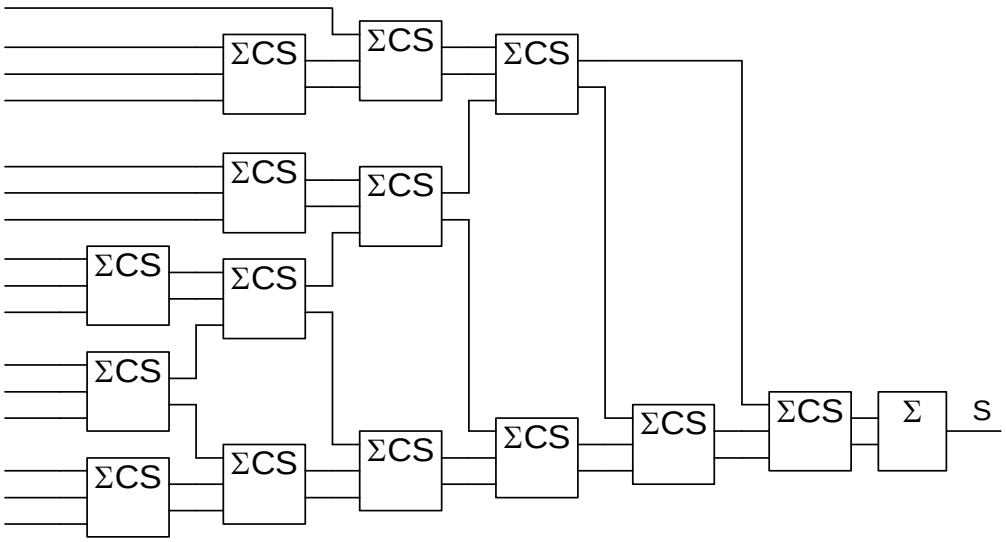



```
A:   1101        G:  _0010
B:   1010       2H:  1101_
C:   0101        I:  _0110
G:   0010        K:  11110
H:  1101_        L: _0010_

D:   1011        K:  11110
E:   1100       2L:  0010_
F:   0001       2J:  1001_
I:   0110        M:  01000
J:  1001_        N: 1011__

M:   01000
2N:  1011__
X:  0110100
```

Notes:
1. ×2 requires no logic: just connect wires appropriately
2. No logic required for adder columns with only 1 input
3. All adders are actually only 4 bits wide
4. Final addition M+2N requires a proper adder

Carry-Save Tree

We can construct a tree to add sixteen values together:



Number of values, K	16		13		9		6		4		3		2		1
$\log_2(K)$	4		3.7		3.17		2.58		2		1.58		1		0
Delay	0		3		6		9		12		15		18		24
$\Delta Delay/\Delta \log_2(K)$		10		5.65		5.13		5.13		7.23		5.13		6	

- The final stage must be a normal adder because we need to obtain a single output.
- The delay is the same as for a conventional lookahead-adder tree but uses much less circuitry.
- The irregularity of the tree causes a reduction in efficiency but this is relatively small (and becomes even smaller for large K).
- Inverting alternate stages will speed up both tree circuits still further but requires more circuitry.

Merry Christmas

The End

Quiz

1. In a 4-bit adder, how can you tell from P0:3 and Q0:3 whether or not C3 is dependent on C–1 ?
2. A multiplexer normally has 2 gate delays from its data inputs to its output. How is this reduced to 1 gate delay in the carry skip circuit ?
3. If five 4-bit numbers are added together, how many bits are needed to represent the result ?