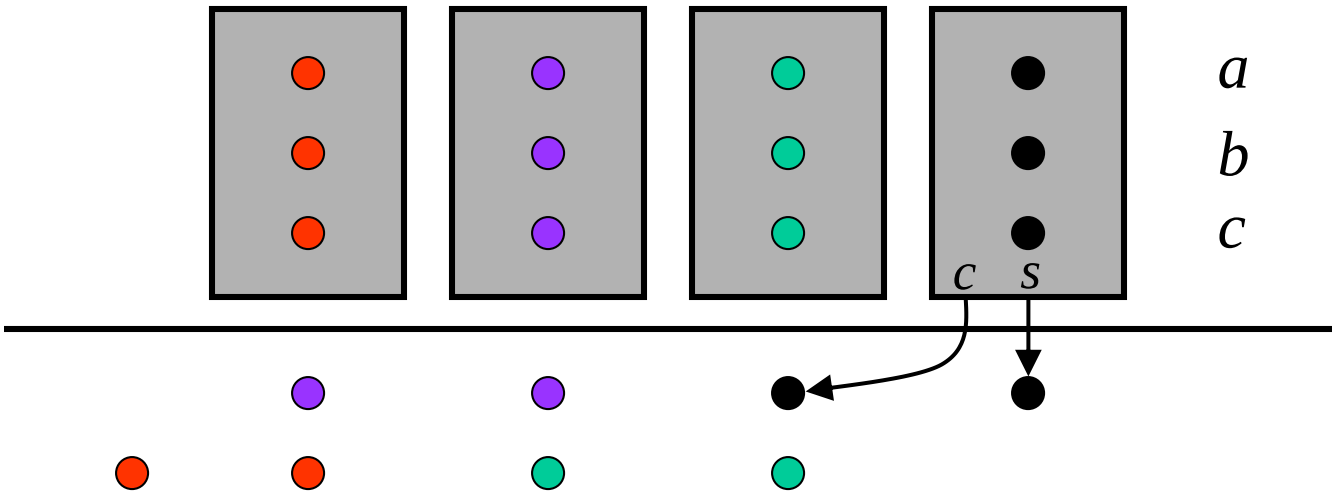
EFFICIENT MULTI-INPUT ADDITION (CARRY-SAVE)

Building Efficient Multiple-input Adders

1. Figure out: algorithm, format conversion if needed, word alignment, sign extension, rounding, etc.
2. Draw "dot diagram" of inputs (one dot per bit)
3. Cover with carry-save adder blocks as appropriate
4. Repeat step #3 until there are two output terms
5. Calculate final result with CPA (carry-propagate adder)
6. Things to check (as a reminder)
   - Output width sufficient
   - Inputs sign extended if necessary
   - Only calculate necessary output bits

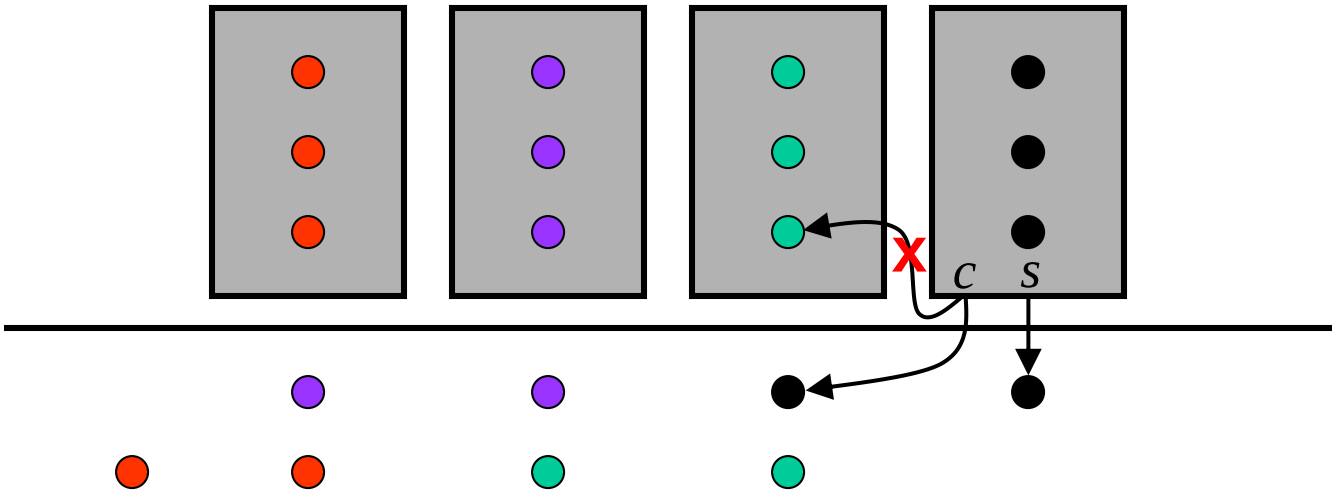
3:2 Adder Row

- Compresses 3 words (4-bit *a, b, c* in this example) to two words



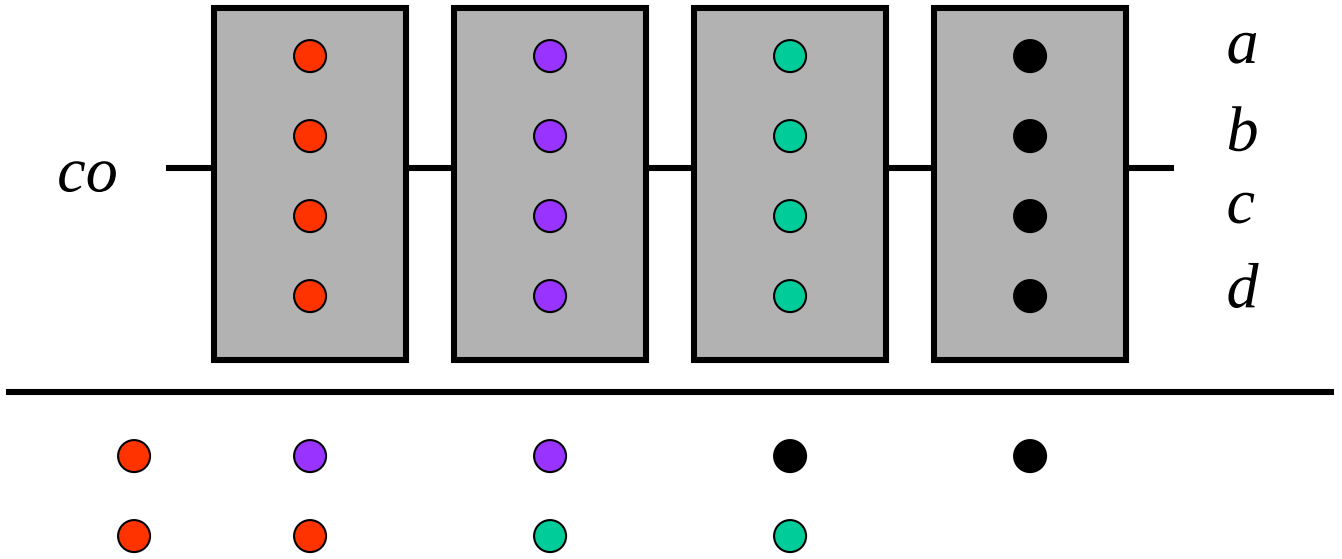
3:2 Adder Row

- No "sideways" carry signals to a neighboring 3:2 adder even though it would be correct logically (it would cause a slow ripple)
  - Having said that, it is ok if limited to a small number of ripples in special cases



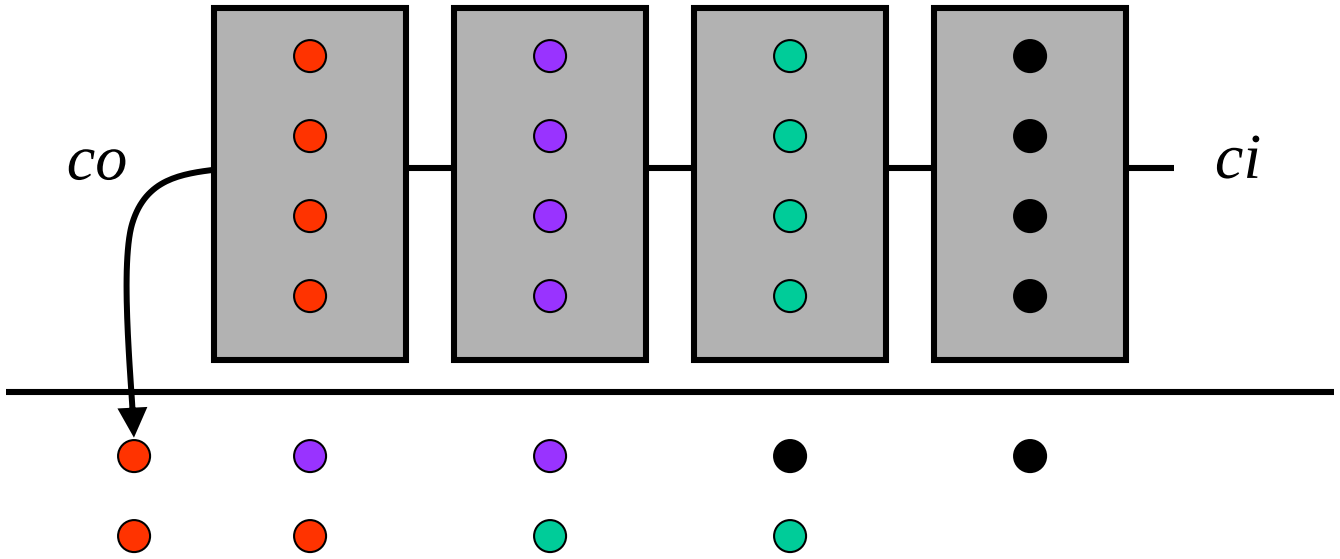
4:2 Adder Row

- Compresses 4 words (4-bit *a, b, c, d* in this example) to two words



4:2 Adder Row

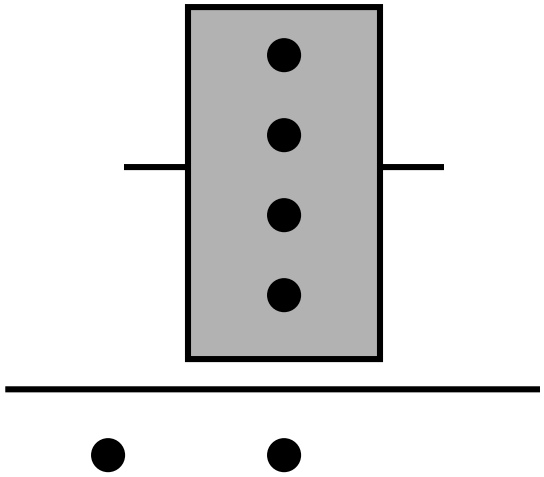
- Remember *ci* and *co* signals on the ends of the row of 4:2 adders



Carry-save Adder Building Blocks

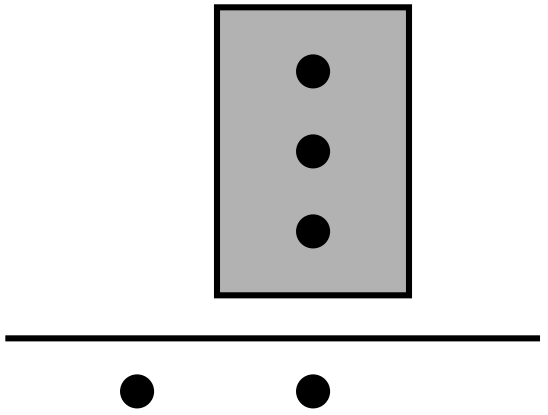
- Notice that to eliminate or "compress" one dot requires approximately 1 "Full adder of hardware"
  - A 4:2 adder can be made with two full adders

4:2 (really 5:3)



- 4 dots → 2 dots (actually 5 dots → 3 dots)
- Compresses 2 dots
- "2 FA" hardware

3:2 or Full adder



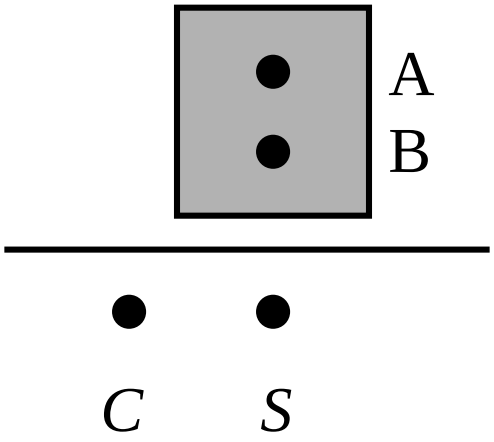
- 3 dots → 2 dots
- Compresses 1 dot
- "1 FA" hardware

Carry-save Adder Building Blocks: Half Adder

- Half adder
-

A	B	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

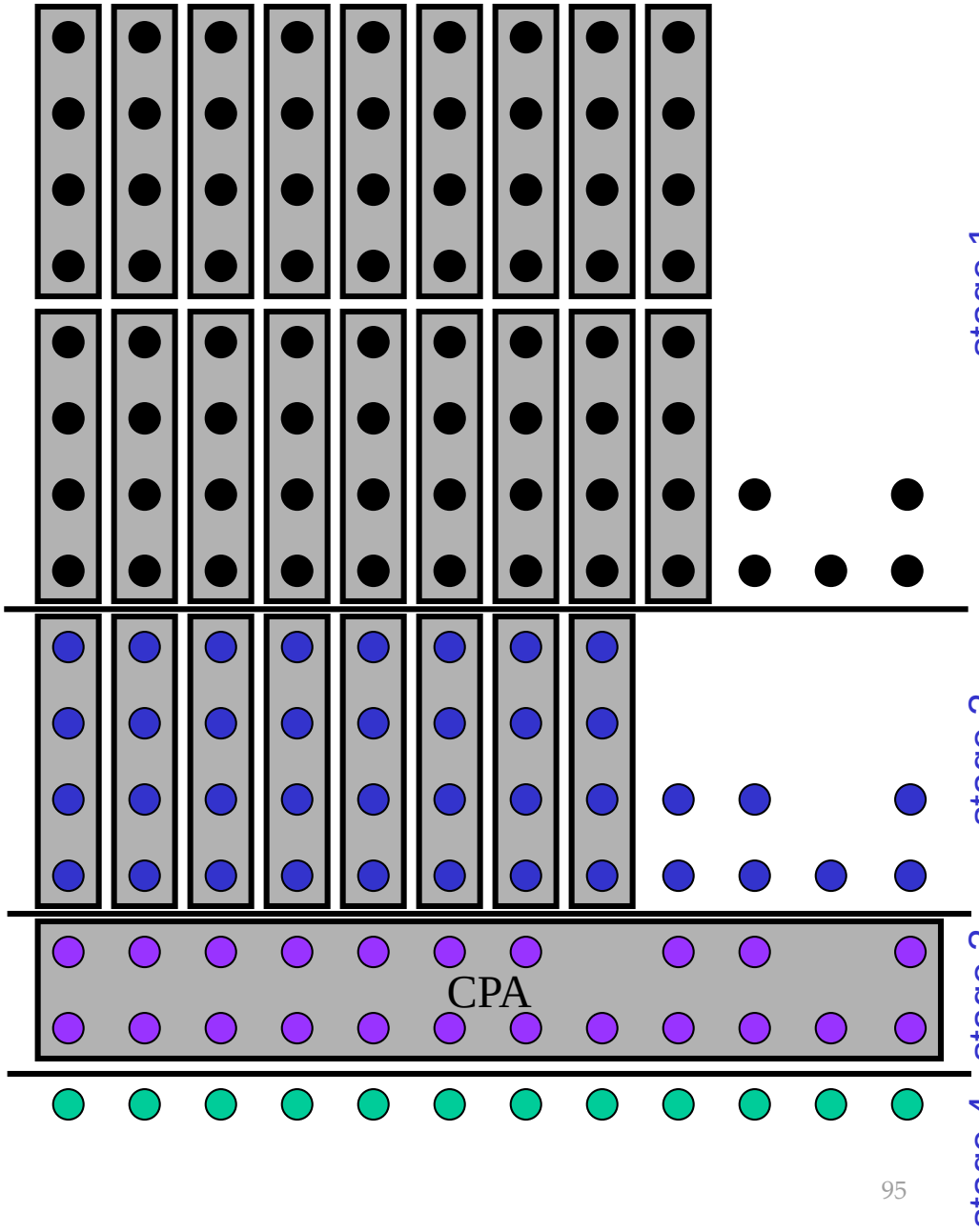
- $S = \text{XOR}(in_0, in_1)$
  $C = \text{AND}(in_0, in_1)$
- Doesn't *reduce* the number of dots but can still be very useful for *moving* dots



- 2 dots → 2 dots
- No compression!
- Approximately "0.5 FA" hardware

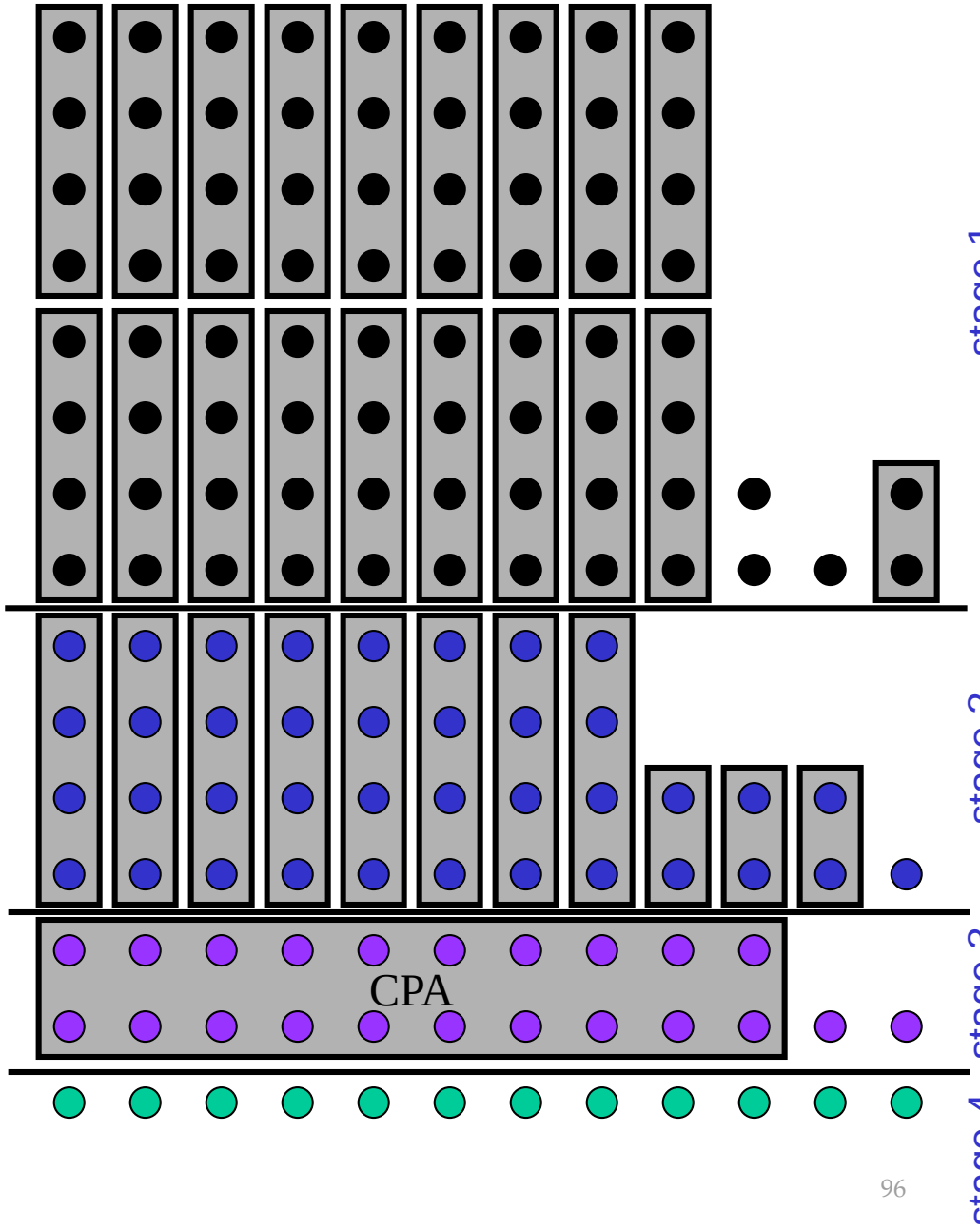
Carry-save Adder Example, Solution 1

- Example using 4:2 adders
  - Inputs: 8 words with hypothetical group of bits in the three LSBs (could be caused by rounding)
  - Output: 12-bit, single-word
- Requires 12-bit CPA for the final adder



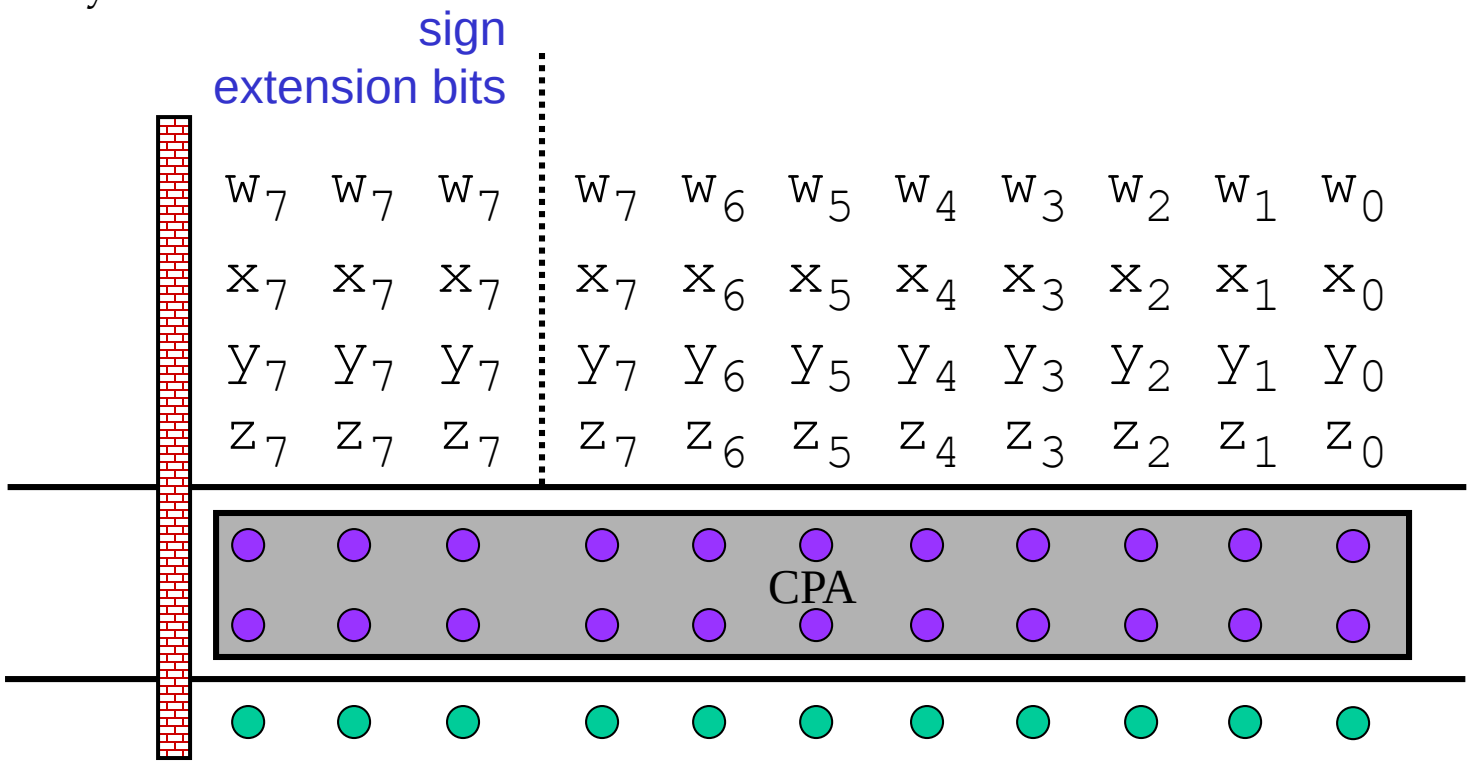
Carry-save Adder Example, Solution 2

- Uses half adders
- The final CPA is now a 10-bit adder instead of 12-bit
- Even if a smaller 10-bit CPA adder is not used, a 12-bit adder with this dot diagram will be faster than with the *Solution 1* dot diagram due to "0" inputs into LSB bits which results in a shorter longest (critical) path
  - Faster circuits are usually also smaller and lower power
  - Typically Synthesis tools will simplify circuits with constant (0 or 1) inputs



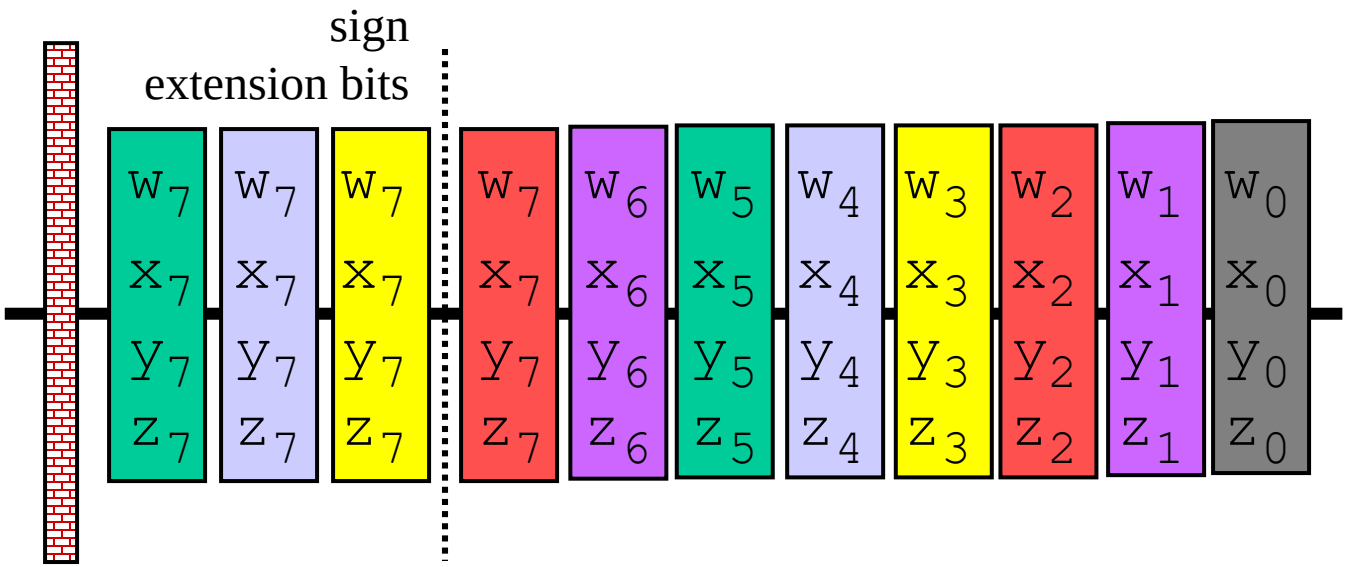
Sign Extension for Multiple-Input Addition

- Example: add four 8-bit numbers with three sign extension bits each (11-bit result). In reality, the sum of four 8-bit numbers needs to be only 10 bits, but we choose 11 here to illustrate a point.
  - In this example, we focus only on optimizing the first stage which consists of only carry-save addition



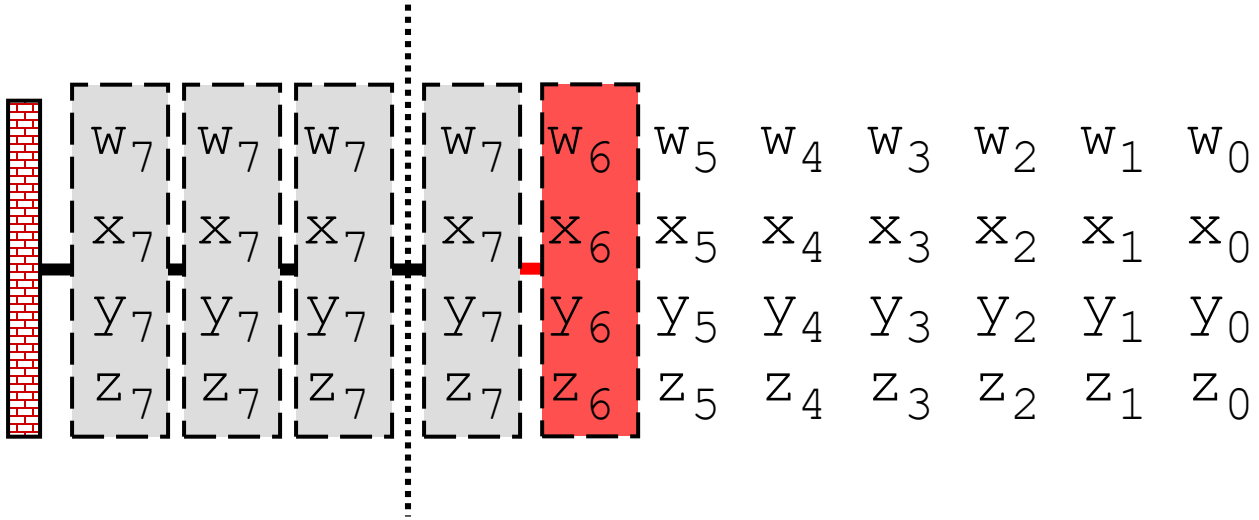
Sign Extension for Multiple-Input Addition

- Method #0: Straightforward solution with a row of 4:2 carry-save adders
  - *c0* → *cin* "sideways" connections are made between all 4:2s



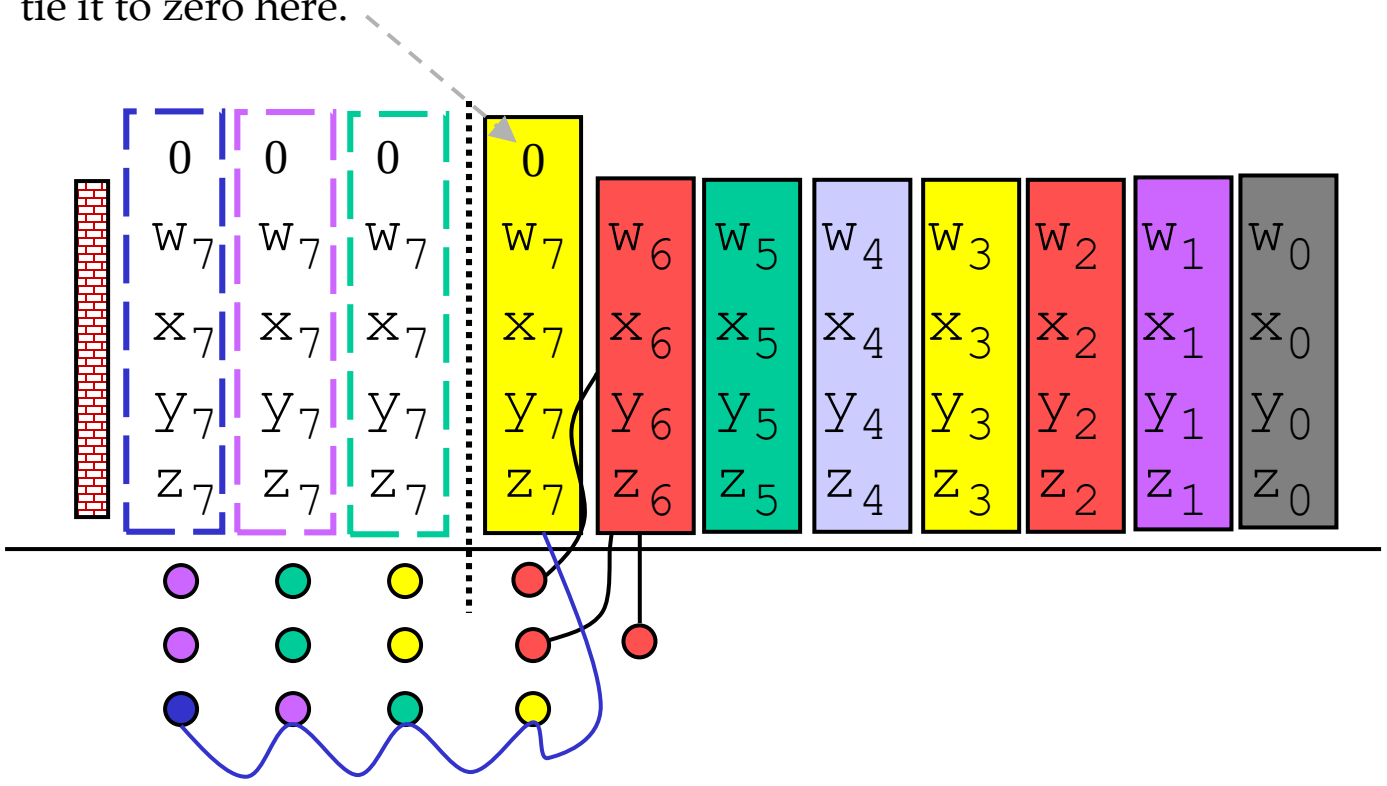
Sign Extension for Multiple-Input Addition

- To be more efficient than the straightforward solution, notice that the input bits in the 4 MSB columns are the same
  - It would be inefficient to calculate the same output multiple times
  - Although the 4 inputs are the same, the *ci* bit is not the same.....for *one* of the 4:2s!
  - Recall that the "side" carry-out is a function of only the four 4:2 inputs shown in the diagram below as *w, x, y, z*. It is by design not a function of the *ci* bit.



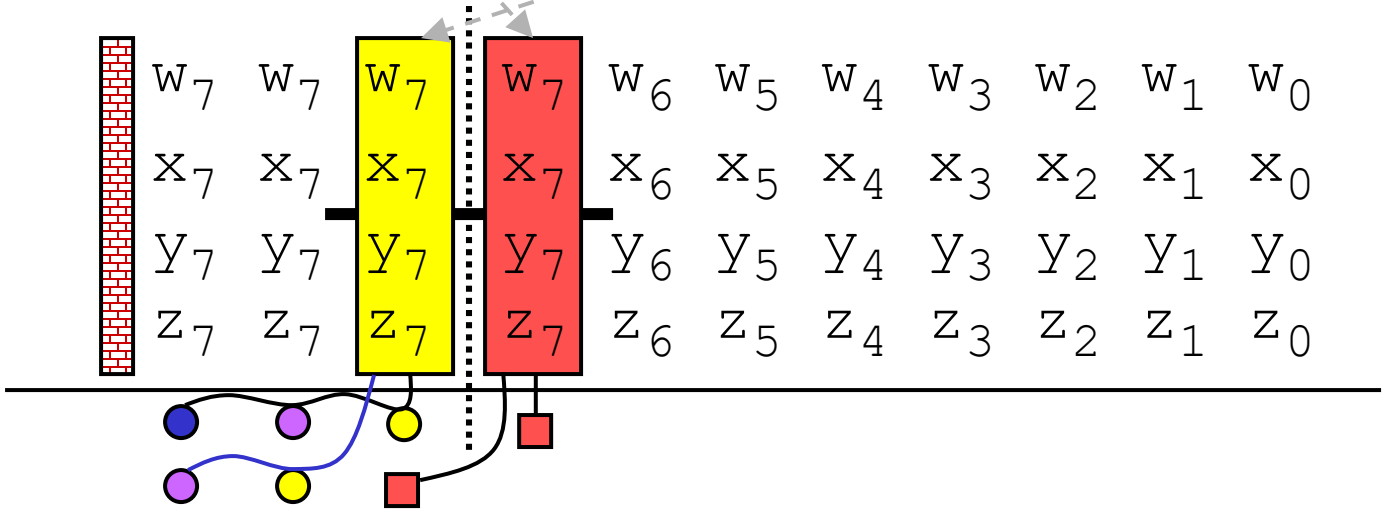
Sign Extension for Multiple-Input Addition

- Method #1a: Use one 4:2 and replicate its output; but *ci* can not be used—so all four columns will be the same
  - There is a huge drawback—the output is in three terms, not two, so this method requires a second stage of carry-save adders
  - Is is hard to imagine when the *ci* input could be useful in this example. We tie it to zero here.



Sign Extension for Multiple-Input Addition

- Method #1b: Use two 4:2s and replicate output of left 4:2
  - All four MSB 4:2s have the same side *co* (≠ f(*ci*))
  - Must keep right 4:2 since its *sum* and *c1* (■ below) depend on a different *ci* than the other three 4:2s' *ci* inputs



Sign Extension for Multiple-Input Addition

- Simplification method #2
  - Look at one sign-extended word
  - There are two cases for the sign-extension bits: 0 or 1
- We want
  - 0 0 0 0 when the MSB = 0
    1 1 1 1 when the MSB = 1
- Note
  - 0 0 0 0 = 1 1 1 1
    + 1
  - 1 1 1 1 = 1 1 1 1
    + 0

Sign Extension for Multiple-Input Addition

Two cases:

w_7=0:	0	0	0	0	w_6	w_5	w_4	w_3	w_2	w_1	w_0
w_7=1:	1	1	1	1	w_6	w_5	w_4	w_3	w_2	w_1	w_0
Substitute:	1	1	1	1							
				$\overline{w_7}$	w_6	w_5	w_4	w_3	w_2	w_1	w_0

- Looks like we made it worse: 1 row → 2 rows; but it gets better...

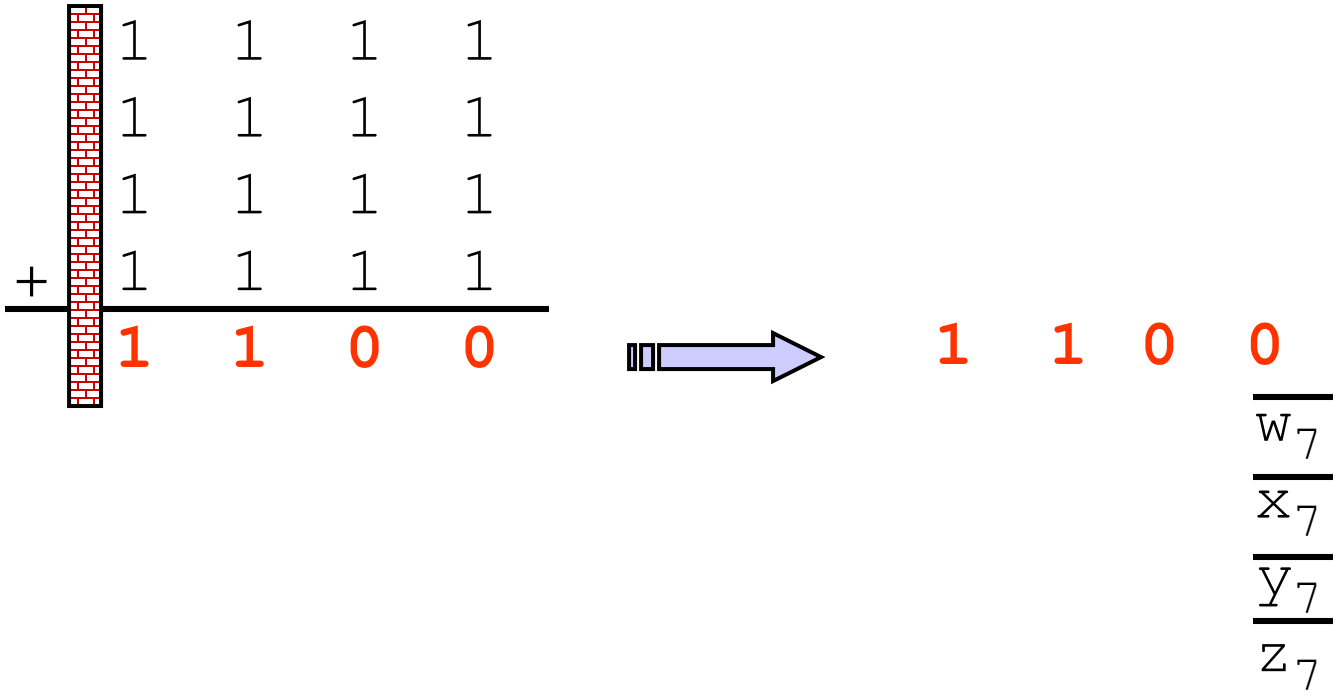
Sign Extension for Multiple-Input Addition

- Simplify all input words similarly

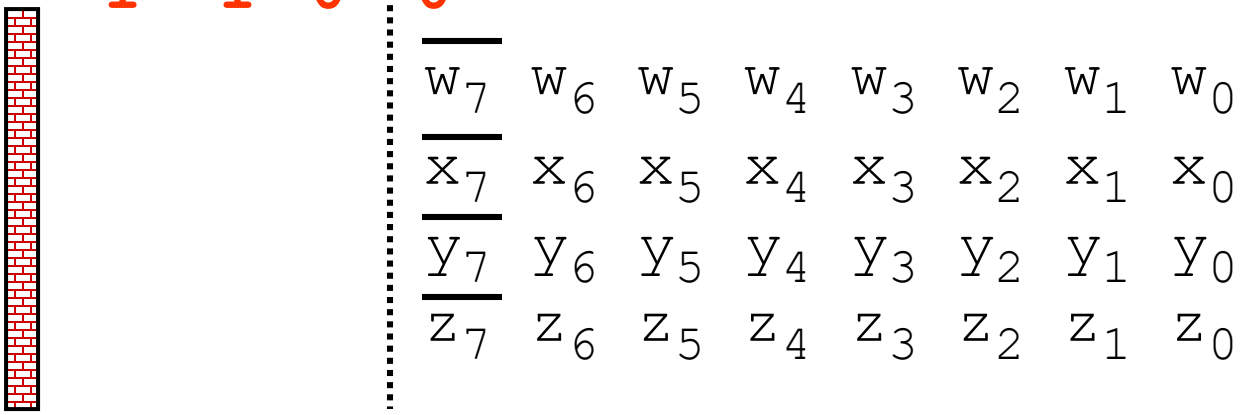
$$
\begin{array}{rcccc}
 & 1 & 1 & 1 & 1 \\
 & 1 & 1 & 1 & 1 \\
 & 1 & 1 & 1 & 1 \\
+ & 1 & 1 & 1 & 1 \\
\hline
 & & & w_7 & \ldots \\
\hline
 & & & x_7 & \ldots \\
\hline
 & & & y_7 & \ldots \\
\hline
 & & & z_7 & \ldots
\end{array}
$$

Sign Extension for Multiple-Input Addition

- Rule: Never use hardware to add constants together
- The four constants can be pre-added in this case

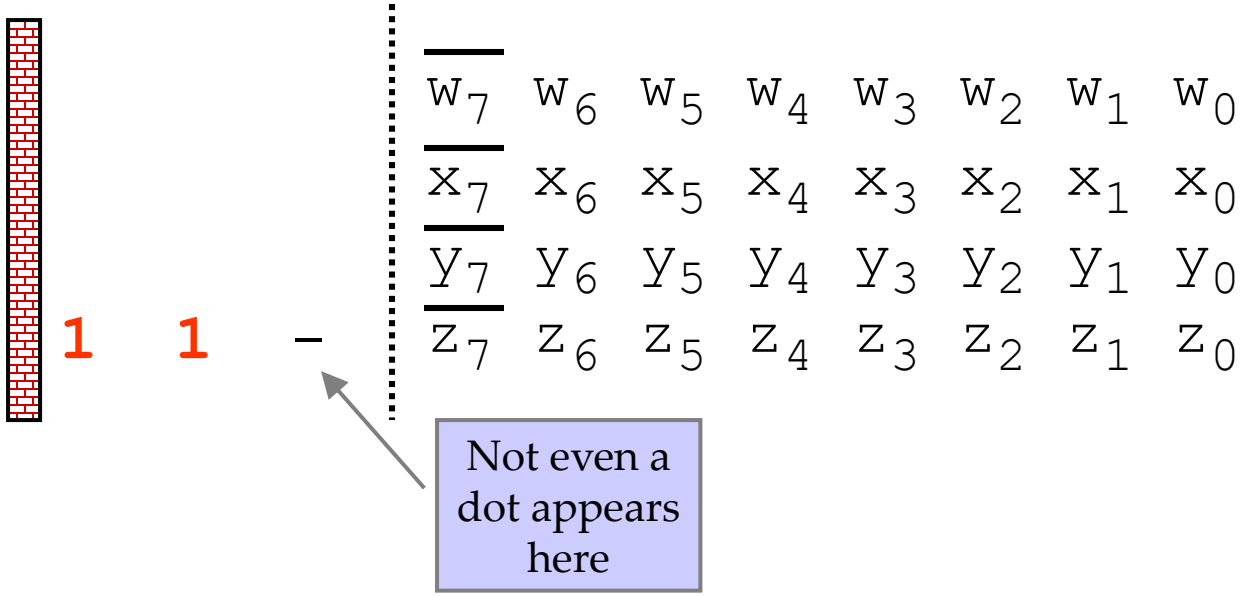


Sign Extension for Multiple-Input Addition



Sign Extension for Multiple-Input Addition

Finally,



- Rule: Never use hardware to add a zero to anything

Sign Extension for Multiple-Input Addition

The dot diagram with no dot in column 8 (LSB = [0], MSB = [10]),