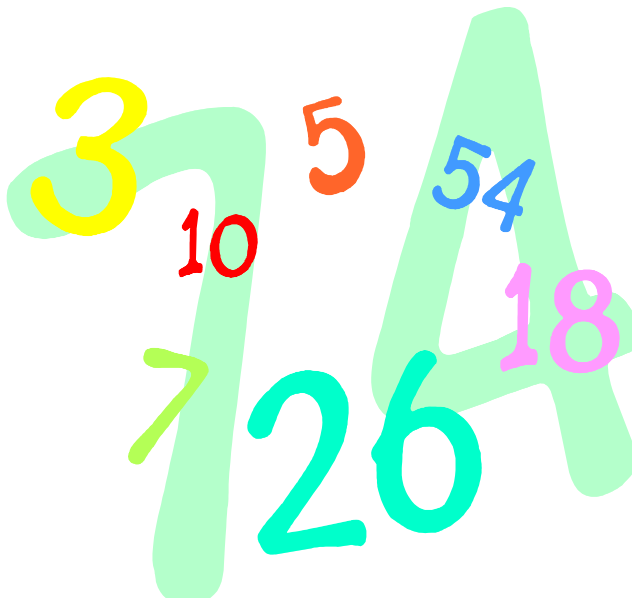
Subtraction

- The arithmetic we did so far was limited to unsigned (positive) integers.
- Today we'll consider negative numbers and subtraction.
  - The main problem is representing negative numbers in binary. We introduce three methods, and show why one of them is the best.
  - With negative numbers, we'll be able to do subtraction using the adders we made last time, because A – B = A + (–B).

Representations and algorithms

- Today we'll look at three different representations of signed numbers.
  - The best one will result in the simplest and fastest operations.
  - This is just like choosing a data structure in programming.
- We're mostly concerned with two particular operations.
  1. Negating a signed number, or finding –x from x.
  2. Adding two signed numbers, or computing x + y.



Signed magnitude representation

- Humans use the signed-magnitude system. We add + or – to the front of a number to indicate its sign.
- We can do this in binary too, by adding a sign bit in front of our numbers.
  - A 0 sign bit represents a positive number.
  - A 1 sign bit represents a negative number.

1101_2	$= 13_{10}$	(a 4-bit unsigned number)
0 1101	$= +13_{10}$	(a positive number in 5-bit signed magnitude)
1 1101	$= -13_{10}$	(a negative number in 5-bit signed magnitude)
0100_2	$= 4_{10}$	(a 4-bit unsigned number)
0 0100	$= +4_{10}$	(a positive number in 5-bit signed magnitude)
1 0100	$= -4_{10}$	(a negative number in 5-bit signed magnitude)

Signed magnitude operations

- Negating a signed-magnitude number is trivial—just change the sign bit from 0 to 1 or vice versa.
- Adding numbers is difficult. Like grade-school addition, signed magnitude addition is based on comparing the signs of the augend and addend.
  - If they have the same sign, add the magnitudes and keep that sign.
  - If they have different signs, then subtract the smaller magnitude from the larger. The result has the same sign as the operand with the larger magnitude.
- This method of subtraction would lead to a rather complex circuit.

$$
\begin{array}{ccccc}
 & + & 3 & 7 & 9 \\
+ & - & 6 & 4 & 7 \\
\hline
 & - & 2 & 6 & 8
\end{array}
\qquad \text{because} \qquad
\begin{array}{cccc}
 & 5 & 13 & 17 \\
 & 6 & 4 & 7 \\
- & 3 & 7 & 9 \\
\hline
 & 2 & 6 & 8
\end{array}
$$

Ones' complement representation

- In a different representation, ones' complement, we negate numbers by complementing each bit of the number.
- We keep the sign bits: 0 for positive numbers, and 1 for negative.
- The sign bit is complemented along with the rest of the bits.

1101_2	$= 13_{10}$	(a 4-bit unsigned number)
0 1101	$= +13_{10}$	(a positive number in 5-bit ones' complement)
1 0010	$= -13_{10}$	(a negative number in 5-bit ones' complement)

0100_2	$= 4_{10}$	(a 4-bit unsigned number)
0 0100	$= +4_{10}$	(a positive number in 5-bit ones' complement)
1 1011	$= -4_{10}$	(a negative number in 5-bit ones' complement)

Why is it called ones' complement?

- Complementing a single bit is equivalent to subtracting it from 1.

x	x'	1 – x
0	1	1
1	0	0

- Similarly, complementing each bit of an *n*-bit number is equivalent to subtracting that number from $2^n - 1$.
- For example, we can negate the 5-bit number 01101.
  - Here $n=5$, and $2^5 - 1 = 11111_2$.
  - Subtracting 01101 from 11111 yields 10010.

$$
\begin{array}{rccccc}
 & 1 & 1 & 1 & 1 & 1 \\
- & 0 & 1 & 1 & 0 & 1 \\
\hline
 & 1 & 0 & 0 & 1 & 0
\end{array}
$$

Ones' complement addition

- There are two steps in adding ones' complement numbers.
  1. Do unsigned addition on the numbers, *including* the sign bits.
  2. Take the carry out and add it to the sum.

$\begin{array}{rrl} & 0\,1\,1\,1 & (+7) \\ + & 1\,0\,1\,1 & +\,(-4) \\ \hline 1 & 0\,0\,1\,0 & \\ & & \\ & 0\,0\,1\,0 & \\ + & 1 & \\ \hline & 0\,0\,1\,1 & (+3) \end{array}$	$\begin{array}{rrl} & 0\,0\,1\,1 & (+3) \\ + & 0\,0\,1\,0 & +\,(+2) \\ \hline 0 & 0\,1\,0\,1 & \\ & & \\ & 0\,1\,0\,1 & \\ + & 0 & \\ \hline & 0\,1\,0\,1 & (+5) \end{array}$

- This is simpler than signed magnitude addition, but still a bit tricky.

Two's complement representation

- Our final idea is two's complement. To negate a number, we complement each bit (just as for ones' complement) and then add 1.

1101_2	$= 13_{10}$	(a 4-bit unsigned number)
0 1101	$= +13_{10}$	(a positive number in 5-bit two's complement)
1 0010	$= -13_{10}$	(a negative number in 5-bit *ones*' complement)
1 0011	$= -13_{10}$	(a negative number in 5-bit two's complement)

0100_2	$= 4_{10}$	(a 4-bit unsigned number)
0 0100	$= +4_{10}$	(a positive number in 5-bit two's complement)
1 1011	$= -4_{10}$	(a negative number in 5-bit *ones*' complement)
1 1100	$= -4_{10}$	(a negative number in 5-bit two's complement)

- People often talk about "taking the two's complement" of a number. This is a confusing phrase, but it usually means to negate some number that's *already* in two's complement format.

More about two's complement

- Another way to negate an n-bit two's complement number is to subtract it from 2^n.

$$\begin{array}{rrl} & 1\,0\,0\,0\,0\,0 & \\ - & 0\,1\,1\,0\,1 & (+13_{10}) \\ \hline & 1\,0\,0\,1\,1 & (-13_{10}) \end{array} \qquad \begin{array}{rrl} & 1\,0\,0\,0\,0\,0 & \\ - & 0\,0\,1\,0\,0 & (+4_{10}) \\ \hline & 1\,1\,1\,0\,0 & (-4_{10}) \end{array}$$

- You can also complement all of the bits to the left of the rightmost 1.

01101 = $+13_{10}$ (a positive number in two's complement)
10011 = -13_{10} (a negative number in two's complement)

00100 = $+4_{10}$ (a positive number in two's complement)
11100 = -4_{10} (a negative number in two's complement)

Two’s complement addition

- Negating a two’s complement number takes a bit of work, but addition is much easier than with the other two systems.
- To find A + B, you just have to do unsigned addition on A and B (including their sign bits), and *ignore* any carry out.
- For example, we can compute 0111 + 1100, or (+7) + (–4).
  - First add 0111 + 1100 as unsigned numbers.

$$
\begin{array}{rcccc}
 & 0 & 1 & 1 & 1 \\
+ & 1 & 1 & 0 & 0 \\
\hline
1 & 0 & 0 & 1 & 1
\end{array}
$$

  - Ignore the carry out (1). The answer is 0011 (+3).

Another two's complement example

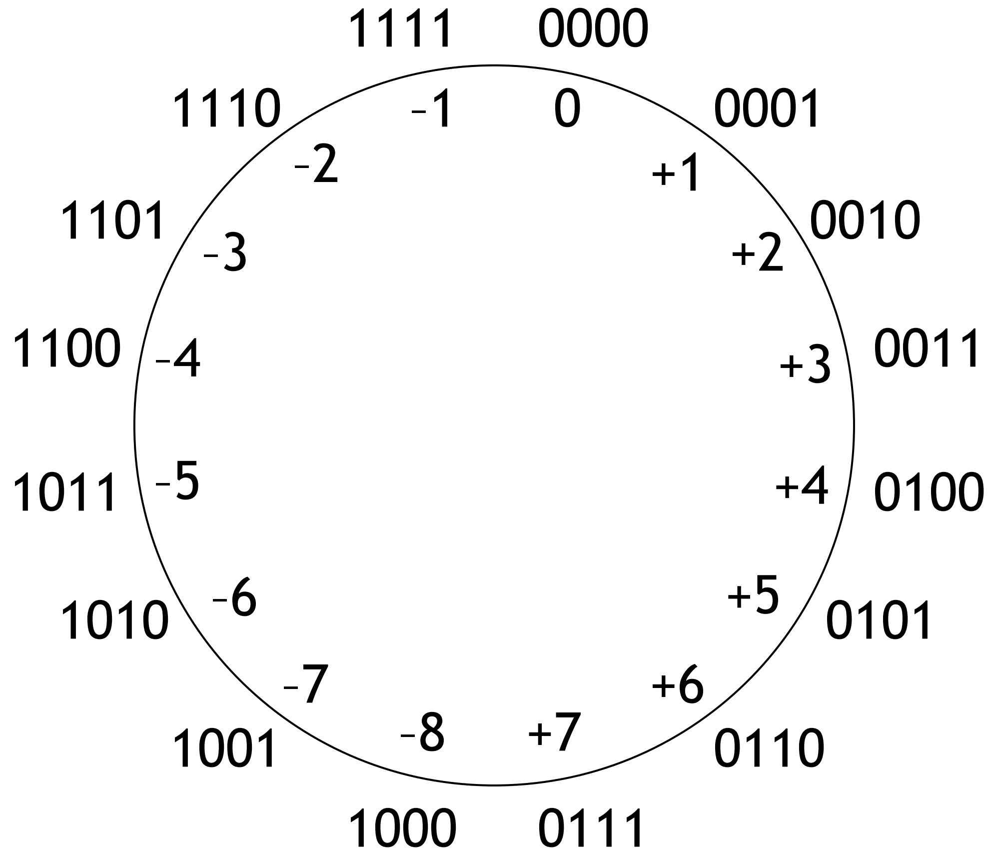
- To further convince you that this works, let's try adding two negative numbers—1101 + 1110, or (–3) + (–2) in decimal.
- Adding the numbers gives 11011.

$$
\begin{array}{r}
1\ 1\ 0\ 1 \\
+\quad 1\ 1\ 1\ 0 \\
\hline
1\ \ 1\ 0\ 1\ 1
\end{array}
$$

- Dropping the carry out (1) leaves us with the answer, 1011 (–5).

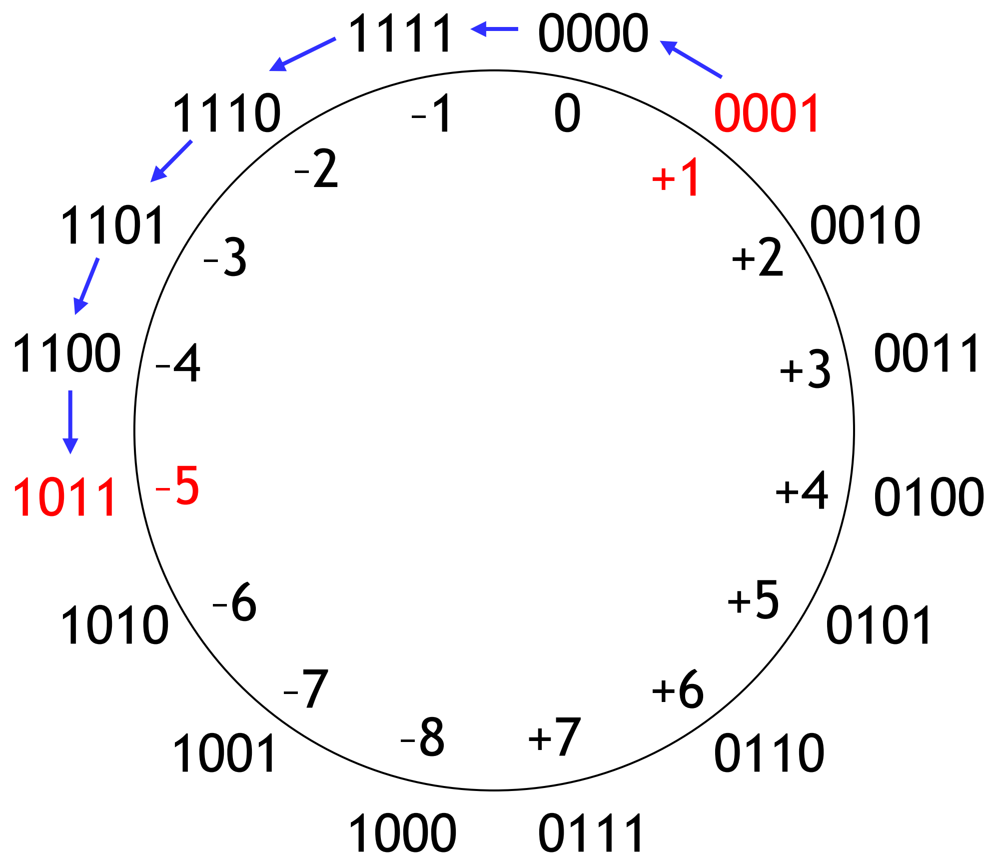
Two’s complement arithmetic is modular

- Here are the 4-bit two’s complement numbers and their decimal values.
- As with modular “clock” arithmetic, let’s think of subtraction as moving counterclockwise around the circle, and addition as moving clockwise.



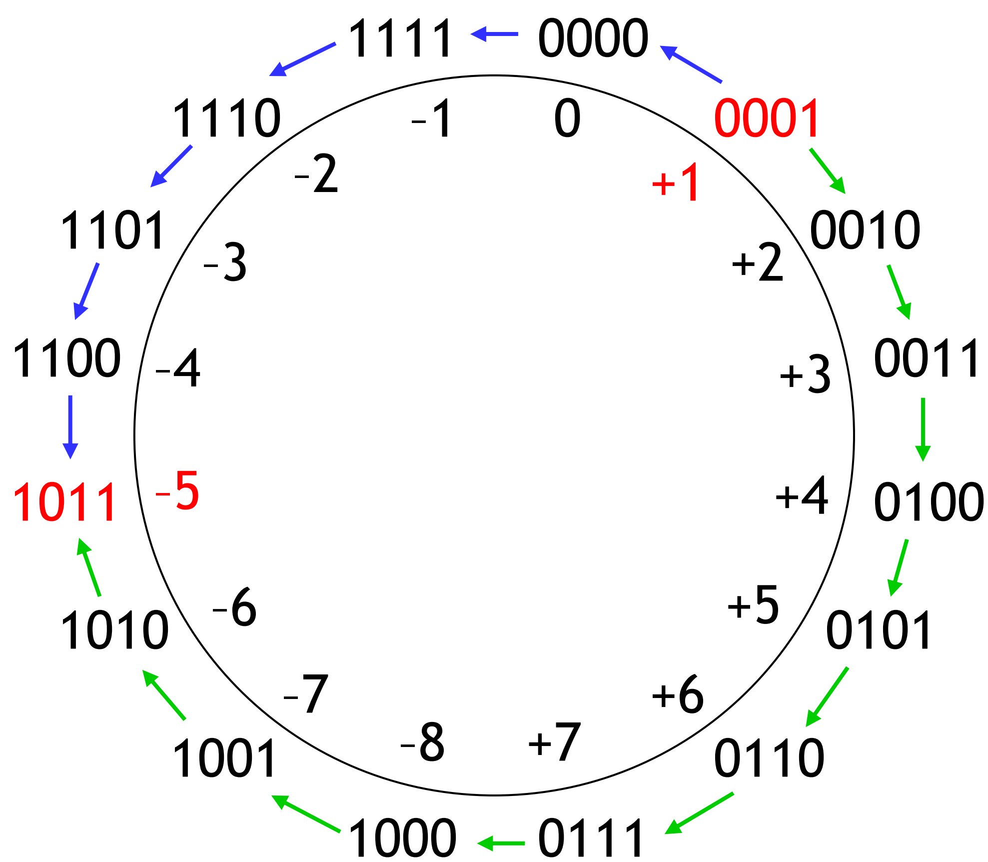
Subtracting x…

- For example, to subtract 6 from 1, go counterclockwise six positions from 1.
- You'll find the answer is –5.



…is equivalent to adding 16 - *x*

- This is the same result you would get if you *added* 10 to 1 instead!
- Subtracting 6 is the same as adding 10, which is why we represent –6 as the unsigned value 10.
- In general, we can always subtract *x* by adding 16 – *x*.

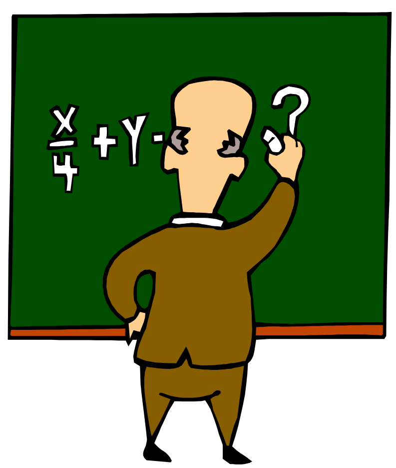


An algebraic explanation

- For n-bit numbers, the negation of B in two's complement is 2^n – B. (This was one of the alternate ways of negating a two's complement number.)

$$\begin{aligned} A - B &= A + (-B) \\ &= A + (2^n - B) \\ &= (A - B) + 2^n \end{aligned}$$

- If A ≥ B, then (A – B) has to be positive, and the 2^n represents a carry out of 1. Discarding this carry out leaves us with the desired result, (A – B).
- If A < B, then (A – B) must be negative, and 2^n – (A – B) corresponds to the correct result –(A – B) in two's complement form.



Comparing the signed number systems

- Here are all the 4-bit numbers in the different systems.
- Positive numbers are the same in all three representations.
- There are *two* ways to represent 0 in signed magnitude and ones' complement. This makes things more complicated.
- In two's complement, there is one more negative number than positive number. Here, we can represent -8 but not +8.
- However, two's complement is preferred because it has only one 0, and its addition algorithm is the simplest.

Decimal	SM	1C	2C
7	0111	0111	0111
6	0110	0110	0110
5	0101	0101	0101
4	0100	0100	0100
3	0011	0011	0011
2	0010	0010	0010
1	0001	0001	0001
0	0000	0000	0000
-0	1000	1111	—
-1	1001	1110	1111
-2	1010	1101	1110
-3	1011	1100	1101
-4	1100	1011	1100
-5	1101	1010	1011
-6	1110	1001	1010
-7	1111	1000	1001
-8	—	—	1000

Ranges of the signed number systems

- How many negative and positive numbers can be represented in each of the different four-bit systems on the previous page?

	Unsigned	SM	1C	2C
Smallest	0000 (0)	1111 (–7)	1000 (–7)	1000 (–8)
Largest	1111 (15)	0111 (+7)	0111 (+7)	0111 (+7)

- The ranges for general *n*-bit numbers (including the sign bit) are below.

	Unsigned	SM	1C	2C
Smallest	0	$-(2^{n-1}-1)$	$-(2^{n-1}-1)$	-2^{n-1}
Largest	2^n-1	$+(2^{n-1}-1)$	$+(2^{n-1}-1)$	$+(2^{n-1}-1)$

Representation example

- Convert 110101 to decimal, assuming several different representations.

  Since the sign bit is 1, this is a negative number. The easiest way to find the magnitude is to negate it.

  (a) signed magnitude format

  Negating the original number, 110101, gives 010101, which is +21 in decimal. So 110101 must represent –21.

  (b) ones' complement

  Negating 110101 in ones' complement yields 001010 = $+10_{10}$*, so the original number must have been* -10_{10}*.*

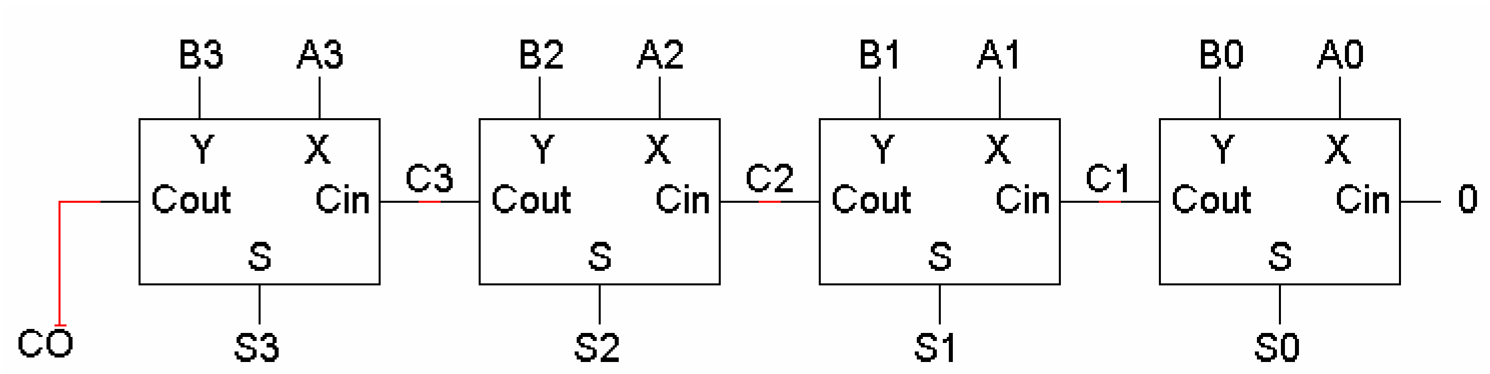
  (c) two's complement

  Negating 110101 in two's complement gives 001011 = 11_{10}*, which means 110101 =* -11_{10}*.*

- The most important point is that a binary value has *different* meanings depending on which number representation is assumed.

Making a subtraction circuit

- Here is the four-bit unsigned addition circuit from last Wednesday.

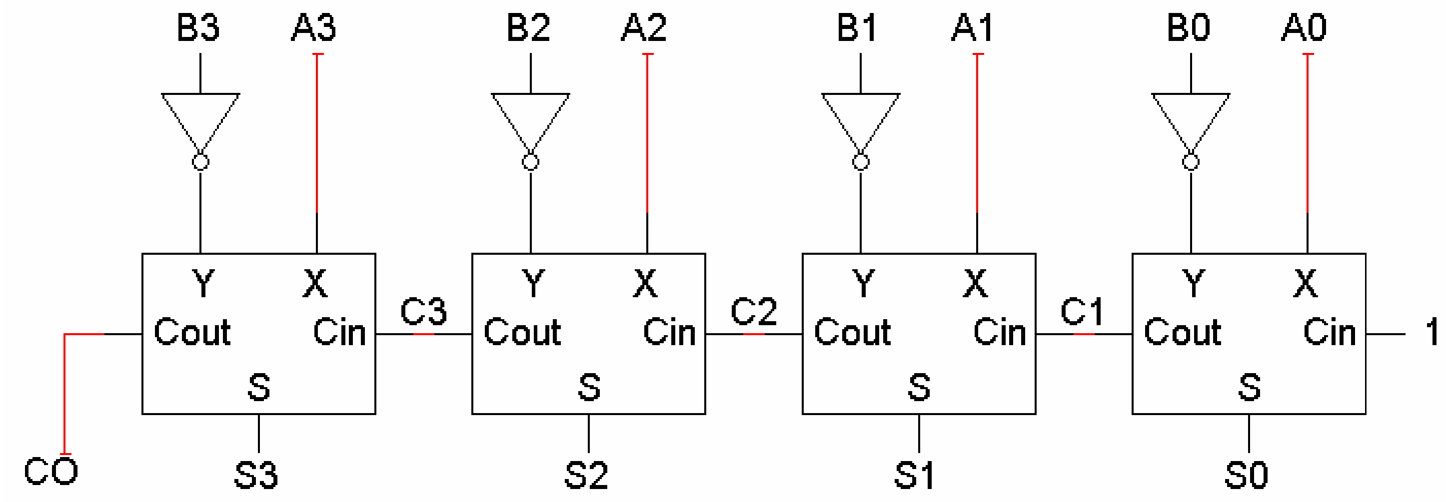


- We could build a subtraction circuit like this too.
- An alternative solution is to re-use this unsigned adder by converting subtraction operations into addition.
- To subtract B from A, we can *add* the negation of B to A.

$$A - B = A + (-B)$$

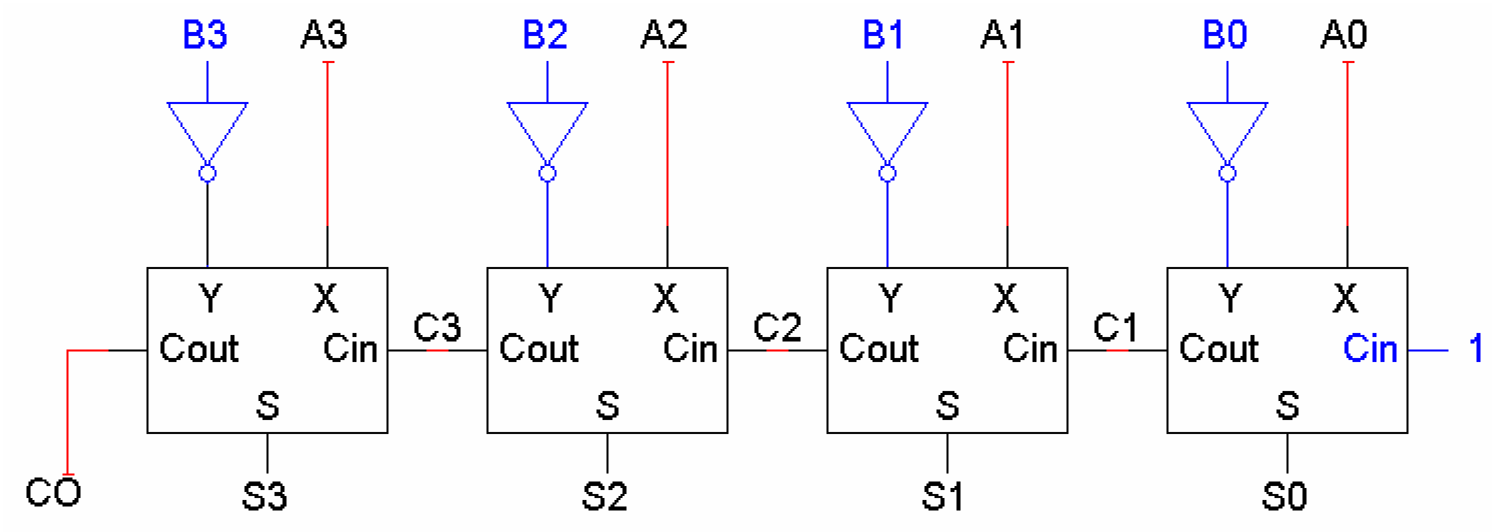
A two’s complement subtraction circuit

- Our circuit has to add A to the two’s complement negation of B.
  - We can complement B by inverting the input bits B3 B2 B1 B0.
  - We can add by setting the carry in to 1 instead of 0.



- The sum is A + (B’ + 1), which is the two’s complement subtraction A – B.
- Remember that A3, B3 and S3 here are actually sign bits.

Small differences



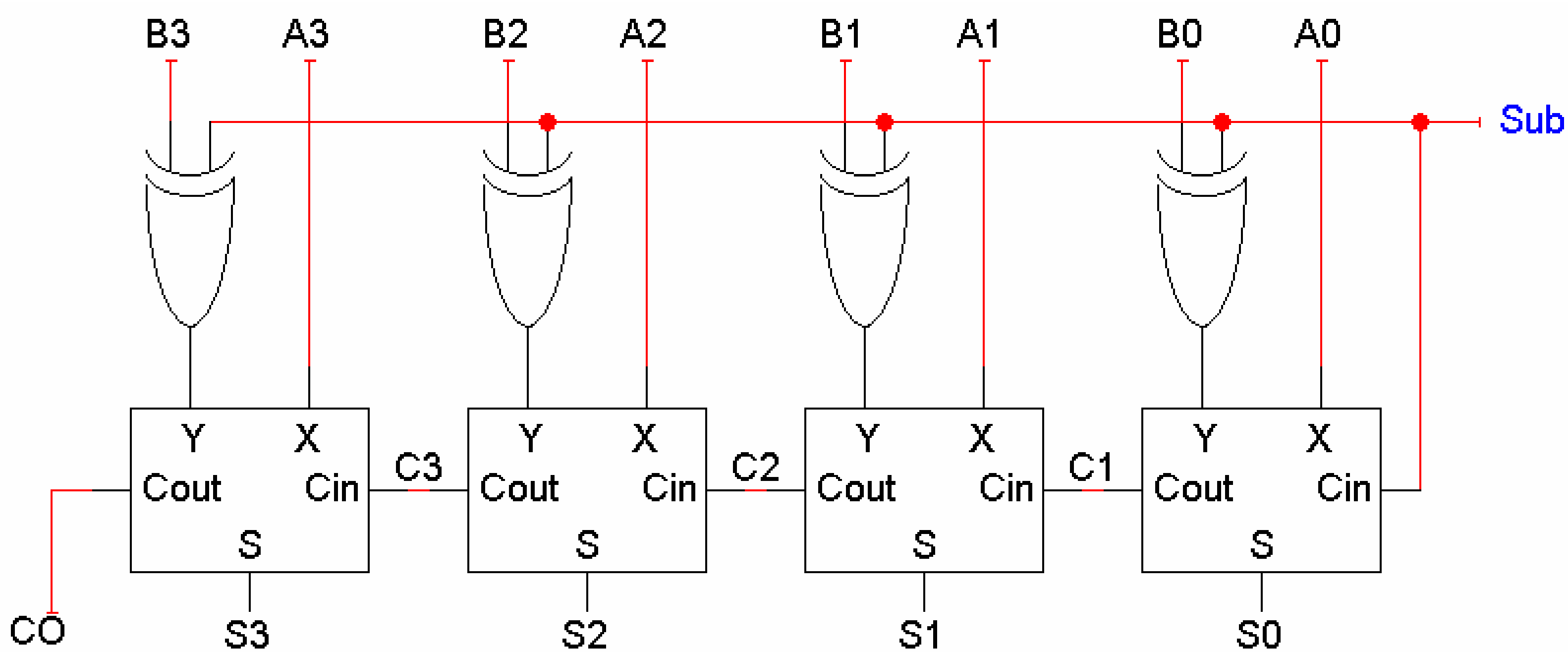
- There are only two differences between an adder and subtractor circuit.
  - The subtractor has to negate B3 B2 B1 B0.
  - The subtractor sets the initial carry in to 1, instead of 0.
- It's not hard to make one circuit that does *both* addition and subtraction.

An adder-subtractor circuit

- XOR gates let us selectively complement the B input.

$$X \oplus 0 = X \qquad\qquad X \oplus 1 = X'$$

- When Sub = 0, the XOR gates output B3 B2 B1 B0 and the carry in is 0. The adder output will be A + B + 0, or just A + B.
- When Sub = 1, the XOR gates output B3’ B2’ B1’ B0’ and the carry in is 1. Thus, the adder output will be a two’s complement subtraction, A – B.



Signed overflow

- With 4-bit two's complement numbers, the largest representable decimal value is +7, and the smallest is –8.
- What if you try to compute 4 + 5, or (–4) + (–5)?

$$
\begin{array}{rr}
 & 0\ 1\ 0\ 0 \\
+ & 0\ 1\ 0\ 1 \\
\hline
 & 0\ 1\ 0\ 0\ 1
\end{array}
\qquad
\begin{array}{rr}
 & (+4) \\
+ & (+5) \\
\hline
 & (-7)
\end{array}
\qquad\qquad
\begin{array}{rr}
 & 1\ 1\ 0\ 0 \\
+ & 1\ 0\ 1\ 1 \\
\hline
 & 1\ 0\ 1\ 1\ 1
\end{array}
\qquad
\begin{array}{rr}
 & (-4) \\
+ & (-5) \\
\hline
 & (+7)
\end{array}
$$

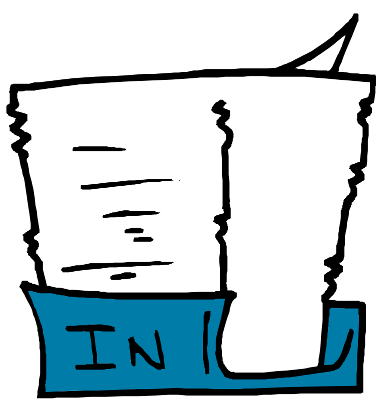
- Signed overflow is very different from unsigned overflow.
  - The carry out is not enough to detect overflow. In the example on the left, the carry out is 0 but there *is* overflow.
  - Also, we cannot include the carry out to produce a five-digit result. In the example on the right, (–4) + (–5) should *not* result in +23!

Detecting signed overflow

- The easiest way to detect signed overflow is to look at all the sign bits.

	0 1 0 0	(+4)			1 1 0 0	(-4)
+	0 1 0 1	+ (+5)		+	1 0 1 1	+ (-5)
	0 1 0 0 1	(-7)			1 0 1 1 1	(+7)

- Overflow occurs only in the two situations above.
  1. If you add two *positive* numbers and get a *negative* result.
  2. If you add two *negative* numbers and get a *positive* result.
- Overflow can never occur when you add a positive number to a negative number. (Do you see why?)



Sign extension

- Decimal numbers are assumed to have an infinite number of 0s in front of them, which helps in "lining up" values for arithmetic operations.

$$\begin{array}{r} 2\,2\,5 \\ +\ 0\,0\,6 \\ \hline 2\,3\,1 \end{array}$$

- You need to be careful in extending signed binary numbers, because the leftmost bit is the *sign* and not part of the magnitude.
- To extend a signed binary number, you have to replicate the sign bit. If you just add 0s in front, you might accidentally change a negative number into a positive one!
- For example, consider going from 4-bit to 8-bit numbers.

(+5)	0101	→	0000 0101	(+5)
(-4)	1100	→	1111 1100	(-4)

Summary

- Data representations are all-important!
  - A good representation for negative numbers can make subtraction hardware much simpler to design.
  - Using two's complement, it's easy to build a single circuit for both addition and subtraction.
- Working with signed numbers involves several issues.
  - Signed overflow is very different from the unsigned overflow we talked about last week.
  - Sign extension is needed to properly "lengthen" negative numbers.