Negative Numbers and Subtraction

- The adders we designed can add only non-negative numbers
  - If we can represent negative numbers, then subtraction is "just" the ability to add two numbers (one of which may be negative).
- We'll look at three different ways of representing signed numbers.
- How can we decide representation is better?
  - The best one should 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:
  - Negating a signed number, or converting x into -x.
  - Adding two signed numbers, or computing x + y.
  - So, we will compare the representation on how fast (and how easily) these operations can be done on them

Signed magnitude representation

- Humans use a signed-magnitude system: we add + or - in front of a magnitude to indicate the sign.
- We could do this in binary as well, by adding an extra sign bit to the front of our numbers. By convention:
  - A 0 sign bit represents a positive number.
  - A 1 sign bit represents a negative number.
- Examples:

$1101_2 = 13_{10}$ (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, though. Signed magnitude is basically what people use, so think about the grade-school approach to addition. It’s 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 one. The sign of the number with the larger magnitude is the sign of the result.
- This method of subtraction would lead to a rather complex circuit.

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

One's complement representation

- A different approach, one's complement, negates 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.
- Examples:

1101_2	$= 13_{10}$	(a 4-bit unsigned number)
0 1101	$= +13_{10}$	(a positive number in 5-bit one's complement)
1 0010	$= -13_{10}$	(a negative number in 5-bit one's complement)
0100_2	$= 4_{10}$	(a 4-bit unsigned number)
0 0100	$= +4_{10}$	(a positive number in 5-bit one's complement)
1 1011	$= -4_{10}$	(a negative number in 5-bit one's complement)

Why is it called "one's complement?"

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

  0' = 1, and 1 - 0 = 1   1' = 0, and 1 - 1 = 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^n-1 = 31_{10} = 11111_2.
  - Subtracting 01101 from 11111 yields 10010:

$$
\begin{array}{r}
1\ 1\ 1\ 1\ 1 \\
-\ 0\ 1\ 1\ 0\ 1 \\
\hline
1\ 0\ 0\ 1\ 0
\end{array}
$$

One's complement addition

- To add one's complement numbers:
  - First do unsigned addition on the numbers, *including* the sign bits.
  - Then take the carry out and add it to the sum.
- Two examples:

	0111	(+7)			0011	(+3)
+	1011	+ (-4)		+	0010	+ (+2)
1	0010			0	0101	
	0010				0101	
+	1			+	0	
	0011	(+3)			0101	(+5)

- This is simpler and more uniform than signed magnitude addition.

Two's complement

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

	$1101_2 = 13_{10}$	(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)

More about two's complement

- Two other equivalent ways to negate two's complement numbers:
  - You can subtract an n-bit two's complement number from 2^n.

$$\begin{array}{rl} 100000 & \\ -\ \ 01101 & (+13_{10}) \\ \hline 10011 & (-13_{10}) \end{array} \qquad \begin{array}{rl} 100000 & \\ -\ \ 00100 & (+4_{10}) \\ \hline 11100 & (-4_{10}) \end{array}$$

  - You can 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)

- Often, people 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.

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.
  - Ignore any carry out.
- For example, to find 0111 + 1100, or (+7) + (-4):
  - First add 0111 + 1100 as unsigned numbers:

$$\begin{array}{r} 0111 \\ +\ 1100 \\ \hline 10011 \end{array}$$

  - Discard 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:

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

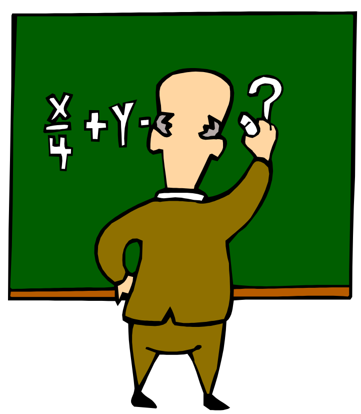
- Dropping the carry out (1) leaves us with the answer, 1011 (-5).

Why does this work?

- For n-bit numbers, the negation of B in two's complement is 2^n - B (this is one of the alternative 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 $A \geq B$, then (A - B) is a positive number, and 2^n represents a carry out of 1. Discarding this carry out is equivalent to subtracting 2^n, which leaves us with the desired result (A - B).
- If $A < B$, then (A - B) is a negative number and we have 2^n - (A - B). This corresponds to the desired 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.*
- Signed magnitude and one's complement have *two* ways of representing 0. This makes things more complicated.
- Two's complement has asymmetric ranges; there is one more negative number than positive number. Here, you 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	S.M.	1's comp.	2's comp.
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 systems on the previous page?

	Unsigned	Signed Magnitude	One's complement	Two's complement
Smallest Largest	0000 (0) 1111 (15)	1111 (-7) 0111 (+7)	1000 (-7) 0111 (+7)	1000 (-8) 0111 (+7)

- In general, with n-bit numbers including the sign, the ranges are:

	Unsigned	Signed Magnitude	One's complement	Two's complement
Smallest Largest	0 2^n-1	$-(2^{n-1}-1)$ $+(2^{n-1}-1)$	$-(2^{n-1}-1)$ $+(2^{n-1}-1)$	-2^{n-1} $+(2^{n-1}-1)$

Converting signed numbers to decimal

- Convert 110101 to decimal, assuming this is a number in:

(a) signed magnitude format

(b) ones' complement

(c) two's complement

Example solution

- Convert 110101 to decimal, assuming this is a number in:

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

  (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 here is that a binary number has *different* meanings depending on which representation is assumed.

Unsigned numbers overflow

- Carry-out can be used to detect overflow
- The largest number that we can represent with 4-bits using unsigned numbers is 15
- Suppose that we are adding 4-bit numbers: 9 (1001) and 10 (1010).

$$
\begin{array}{rr}
1001 & (9) \\
+\ 1010 & (10) \\
\hline
10011 & (19)
\end{array}
$$

- The value 19 cannot be represented with 4-bits
- When operating with unsigned numbers, a carry-out of 1 can be used to indicate overflow

Signed overflow

- With two's complement and a 4-bit adder, for example, the largest representable decimal number is +7, and the smallest is -8.
- What if you try to compute 4 + 5, or (-4) + (-5)?

$$\begin{array}{rr} 0100 & (+4) \\ +\ 0101 & (+5) \\ \hline 01001 & (-7) \end{array} \qquad \begin{array}{rr} 1100 & (-4) \\ +\ 1011 & (-5) \\ \hline 10111 & (+7) \end{array}$$

- We cannot just include the carry out to produce a five-digit result, as for unsigned addition. If we did, (-4) + (-5) would result in +23!
- Also, unlike the case with unsigned numbers, the carry out *cannot* be used to detect overflow.
  - In the example on the left, the carry out is 0 but there *is* overflow.
  - Conversely, there are situations where the carry out is 1 but there is *no* overflow.

Detecting signed overflow

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

```
     0100   (+4)            1100   (-4)
   + 0101   (+5)          + 1011   (-5)
   ----------            ----------
    01001   (-7)           10111   (+7)
```

- Overflow occurs only in the two situations above:
  - If you add two *positive* numbers and get a *negative* result.
  - If you add two *negative* numbers and get a *positive* result.
- Overflow cannot occur if you add a positive number to a negative number. Do you see why?

Sign extension

- In everyday life, decimal numbers are assumed to have an infinite number of 0s in front of them. This helps in "lining up" numbers.
- To subtract 231 and 3, for instance, you can imagine:

$$\begin{array}{r} 231 \\ -\ 003 \\ \hline 228 \end{array}$$

- You need to be careful in extending signed binary numbers, because the leftmost bit is the *sign* and not part of the magnitude.
- If you just add 0s in front, you might accidentally change a negative number into a positive one!
- For example, going from 4-bit to 8-bit numbers:
  - 0101 (+5) should become 0000 0101 (+5).
  - But 1100 (-4) should become 1111 1100 (-4).
- The proper way to extend a signed binary number is to replicate the sign bit, so the sign is preserved.

Signed Numbers are Sign-Extended to Infinity!

- Memorize this idea.
- It will save you every time when you are doing binary arithmetic by hand.
  - 0111 in twos complement is actually …00000111
  - 1011 in twos complement is actually …11111011
- 1011 + 1100 = 10111.
  - This is overflow. But can you see it? Look again:
- …11111011 + …11111100 = …11110111
  - The sign of the result (1) is different than bit 3 (0).
    - We cannot represent the result in 4 bits
  - The sign of the result is extended to infinity
  - Check this every time in your answers and you will always see the precision of the result (number of significant bits) very clearly.
    - Sign-extend inputs to infinity
    - Do the math on an infinite number of bits (conceptually)
    - Truncate the result to the desired output type

Making a subtraction circuit

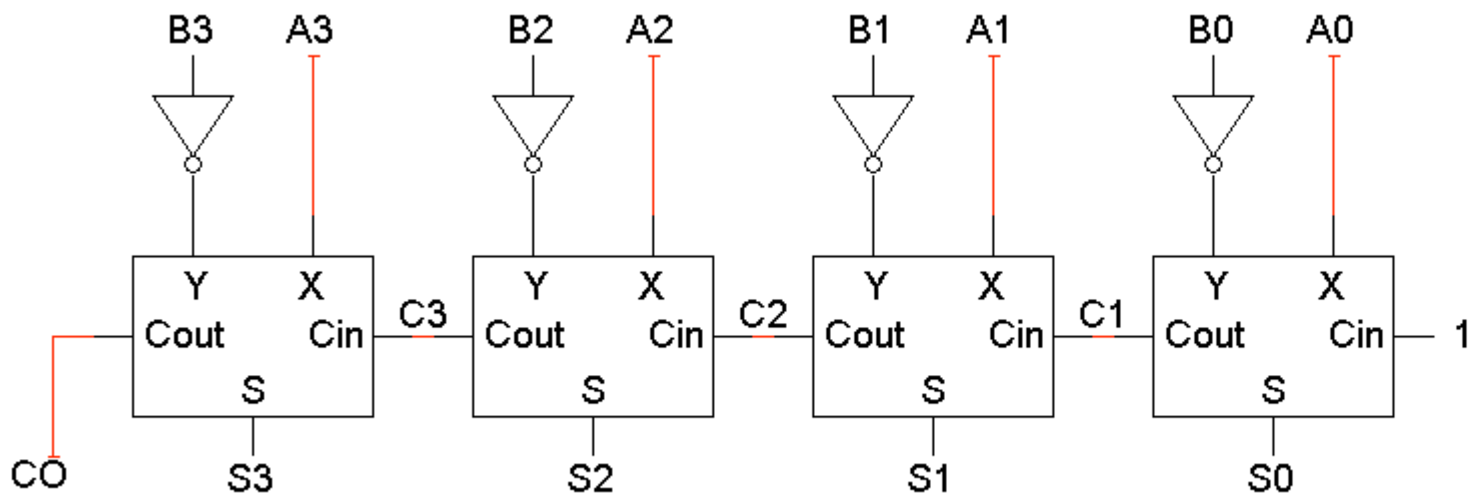
- We could build a subtraction circuit directly, similar to the way we made unsigned adders.
- However, by using two's complement we can convert any subtraction problem into an addition problem. Algebraically,

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

- So to subtract B from A, we can instead *add* the negation of B to A.
- This way we can re-use the unsigned adder hardware from last time.

A two's complement subtraction circuit

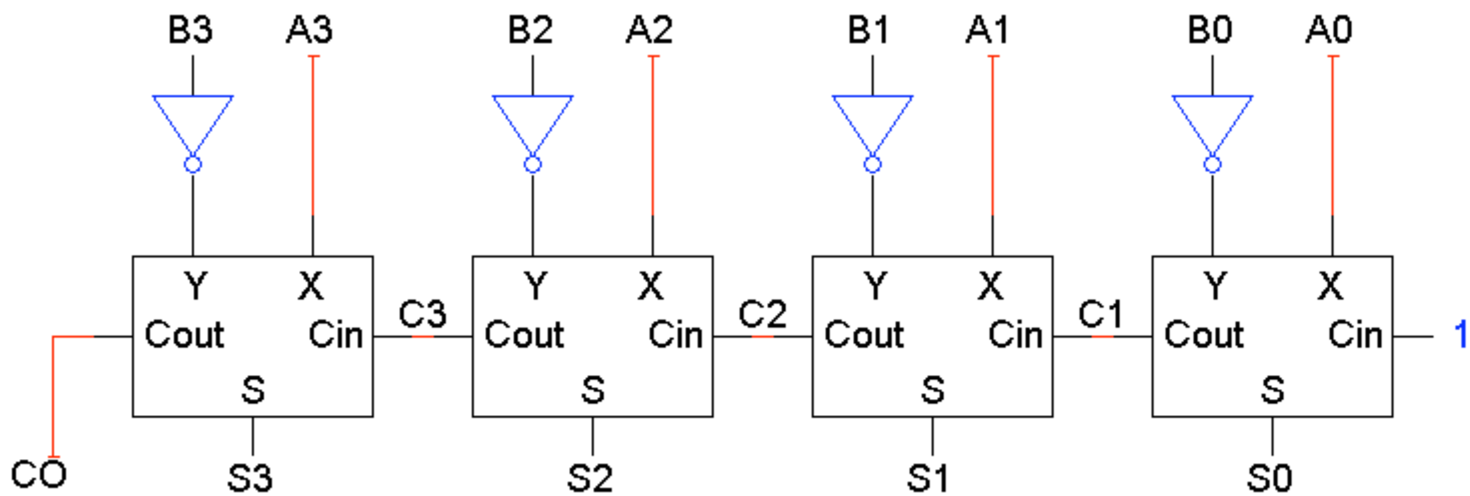
- To find A - B with an adder, we'll need to:
  - Complement each bit of B.
  - Set the adder's carry in to 1.
- The net result is A + B' + 1, where B' + 1 is the two's complement negation of B.



- Remember that A3, B3 and S3 here are actually sign bits.

Small differences

- The only differences between the adder and subtractor circuits are:
  - The subtractor has to negate B3 B2 B1 B0.
  - The subtractor sets the initial carry in to 1, instead of 0.



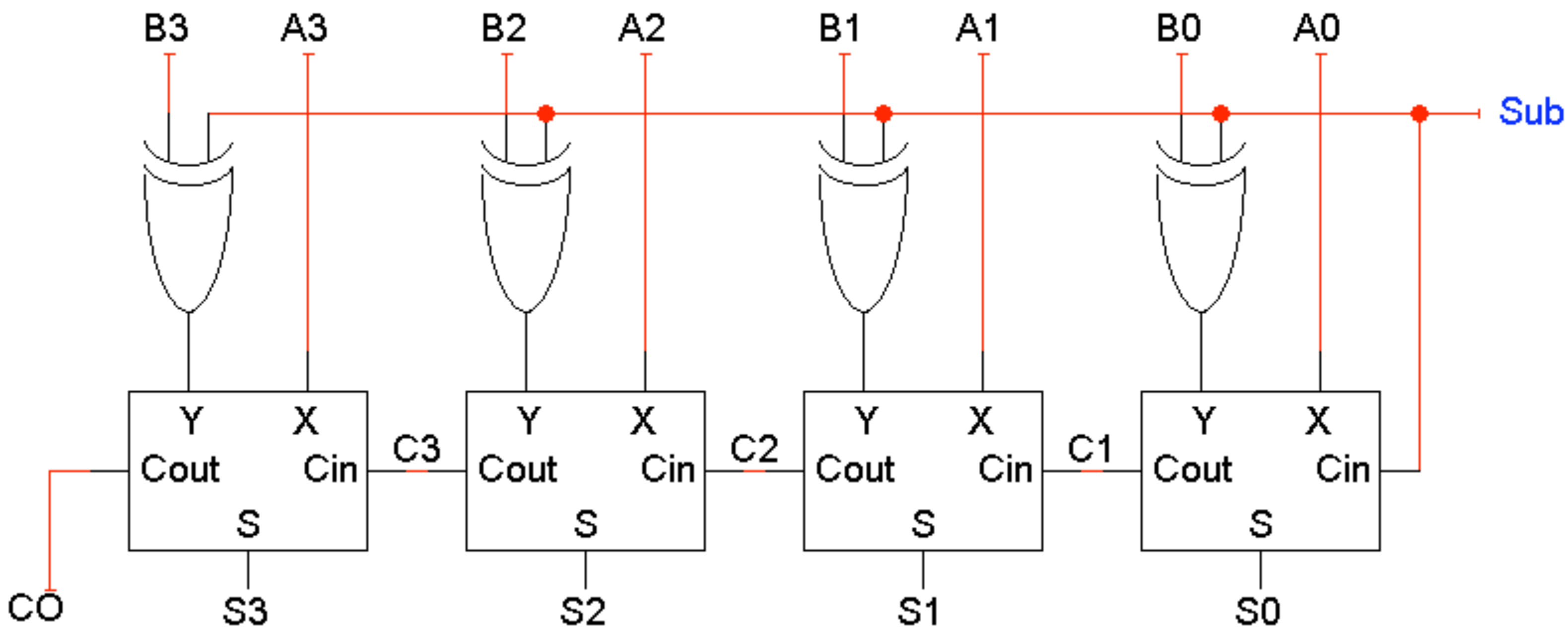
- It's not too hard to make one circuit that does *both* addition and subtraction.

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.



Subtraction summary

- A good representation for negative numbers makes subtraction hardware much easier to design.
  - Two's complement is used most often (although signed magnitude shows up sometimes, such as in floating-point systems, which we'll discuss later).
  - Using two's complement, we can build a subtractor with minor changes to the adder from last time.
  - We can also make a single circuit which can both add and subtract.
- Overflow is still a problem, but signed overflow is very different from the unsigned overflow we mentioned last time.
- Sign extension is needed to properly "lengthen" negative numbers.

- After the midterm we'll use most of the ideas we've seen so far to build an ALU – an important part of a processor.