College of Computer and Information Sciences
Department of Computer Science

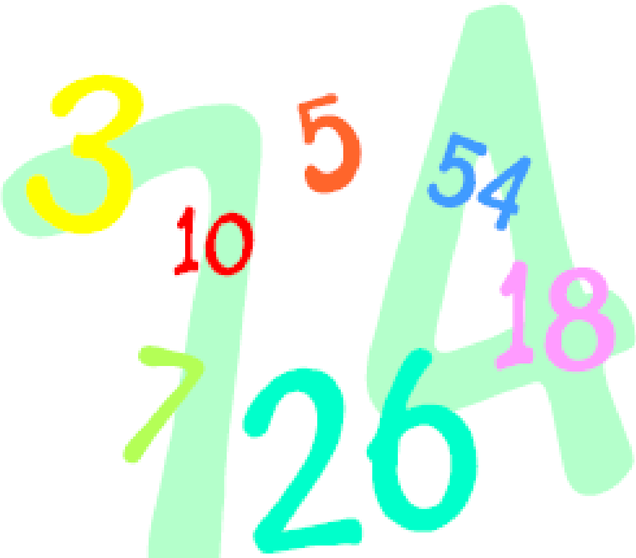
CSC 220: Computer Organization

Unit 9

Data Representation

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.



For Review

Unsigned Representation

- Represents positive integers.
- Unsigned representation of 157:

position	7	6	5	4	3	2	1	0
Bit pattern	**1**	**0**	**0**	**1**	**1**	**1**	**0**	**1**
contribution	2^7			2^4	2^3	2^2		2^0

- Addition is simple:

1 0 0 1 + 0 1 0 1 = 1 1 1 0.

Advantages and disadvantages of unsigned notation

Advantages:

One representation of zero

Simple addition

Disadvantages

Negative numbers can not be represented.

The need of different notation to represent negative numbers.

Representation of negative numbers

Is a representation of negative numbers possible?

Unfortunately:

you can not just stick a negative sign in front of a binary number. (it does not work like that)

There are three methods used to represent negative numbers.

- Signed magnitude notation
- One's complement notation
- Two's complement notation

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)

Example

- Suppose 10011101 is a signed magnitude representation.
- The sign bit is 1, then the number represented is negative

position	7	6	5	4	3	2	1	0
Bit pattern	**1**	**0**	**0**	**1**	**1**	**1**	**0**	**1**
contribution	-			2^4	2^3	2^2		2^0

- The magnitude is 0011101 with a value $2^4+2^3+2^2+2^0= 29$
- Then the number represented by 10011101 is –29.

Exercise 1

37_{10} has 0010 0101 in signed magnitude notation. Find the signed magnitude of -37_{10} ?

Using the signed magnitude notation find the 8-bit binary representation of the decimal value 24_{10} and -24_{10}.

Find the signed magnitude of –63 using 8-bit binary sequence?

Disadvantage of Signed Magnitude

Addition and subtractions are difficult.

Signs and magnitude, both have to carry out the required operation.

They are two representations of 0

00000000 = $+ 0_{10}$

10000000 = $- 0_{10}$

To test if a number is 0 or not, the CPU will need to see whether it is 00000000 or 10000000.

0 is always performed in programs.

Therefore, having two representations of 0 is inconvenient.

Signed Magnitude-Summary

In signed magnitude notation,

The most significant bit is used to represent the sign.

1 represents negative numbers

0 represents positive numbers.

The unsigned value of the remaining bits represent The magnitude.

Advantages:

Represents positive and negative numbers

Disadvantages:

two representations of zero,

Arithmetic operations are difficult.

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.

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

      0 0 1 0
+           1
---------------
      0 0 1 1        (+3)
```

```
      0 0 1 1        (+3)
+     0 0 1 0      + (+2)
---------------    --------
    0 0 1 0 1

      0 1 0 1
+           0
---------------
      0 1 0 1        (+5)
```

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

Two's Complement Notation

The most used representation for integers.

All positive numbers begin with 0.

All negative numbers begin with 1.

One representation of zero

i.e. 0 is represented as 0000 using 4-bit binary sequence.

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)

More about two's complement

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

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

- 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.

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

  - 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.

		1	1	0	1
+		1	1	1	0
	1	1	0	1	1

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

Advantages of Two's Complement Notation

It is easy to add two numbers.

0 0 0 1 +1	1 0 0 0 -8
+	+
0 1 0 1 +5	0 1 0 1 +5
0 1 1 0 +6	**1 1 0 1 -3**

Subtraction can be easily performed.

Multiplication is just a repeated addition.

Division is just a repeated subtraction

Two's complement is widely used in ***ALU***

Example- 10101 in Two's Complement

The most significant bit is 1, hence it is a negative number.

Corresponding + number is 01011 = 8 + 2+1 = 11

the result is then –11.

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 $A \geq B$, 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}.

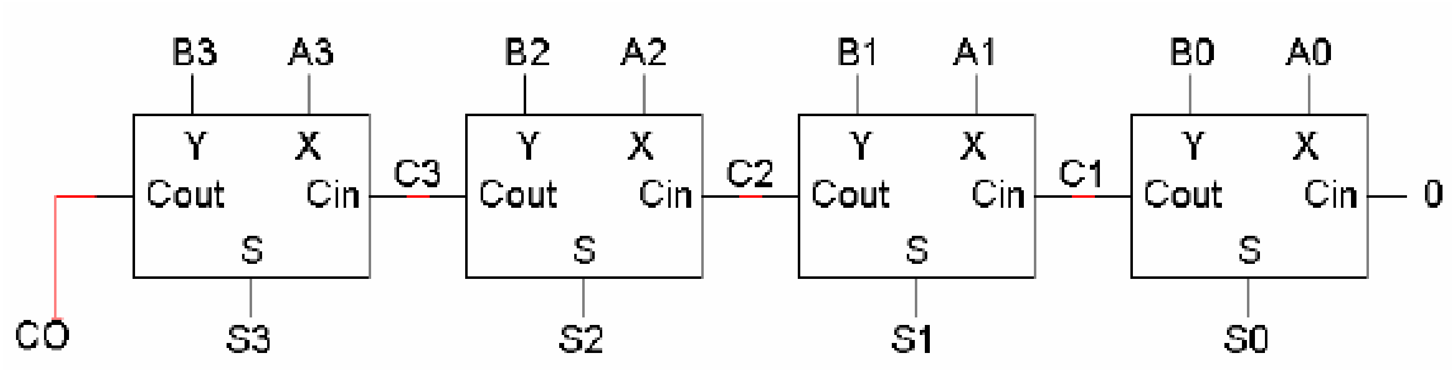
  (c) two's complement

  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

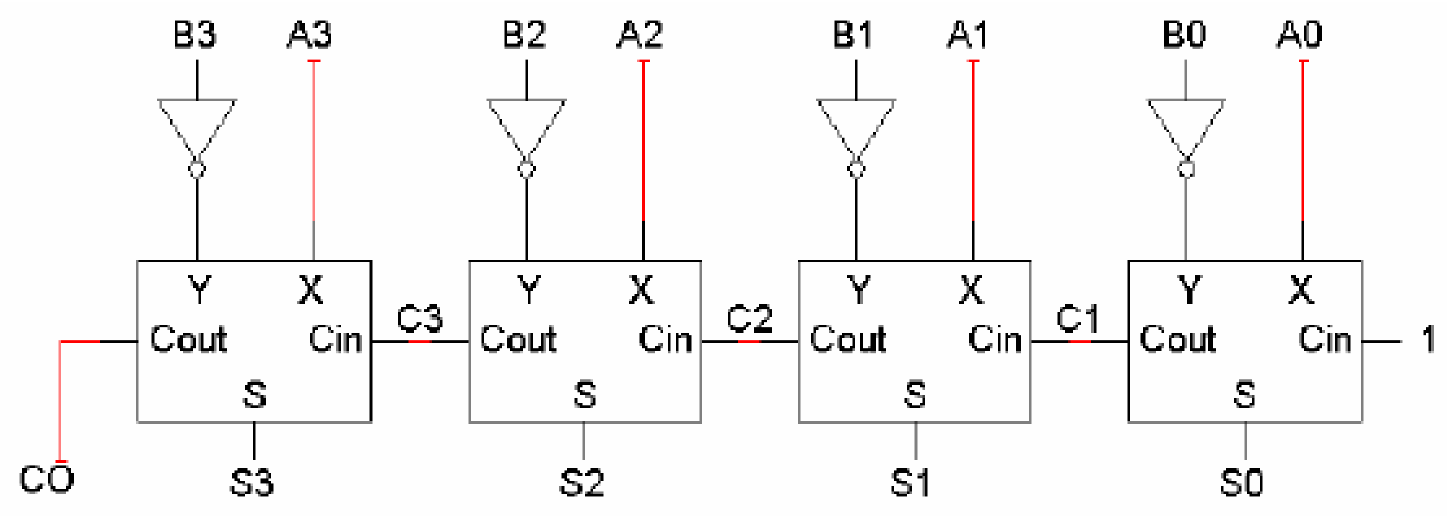


- We could build a subtraction circuit like this too.
- An alternative solution is to re-use this unsigned adder by converting subtraction operations into addition.
- To subtract B from A, we can *add* the negation of B to A.

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

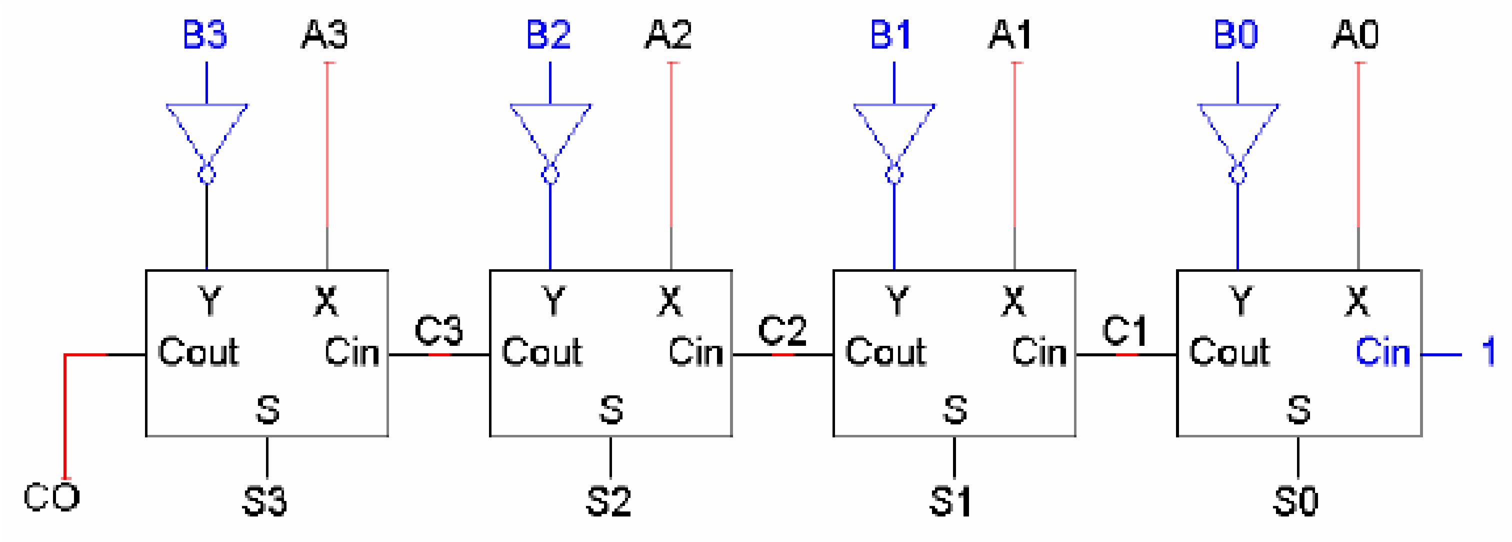
A two's complement subtraction circuit

- 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



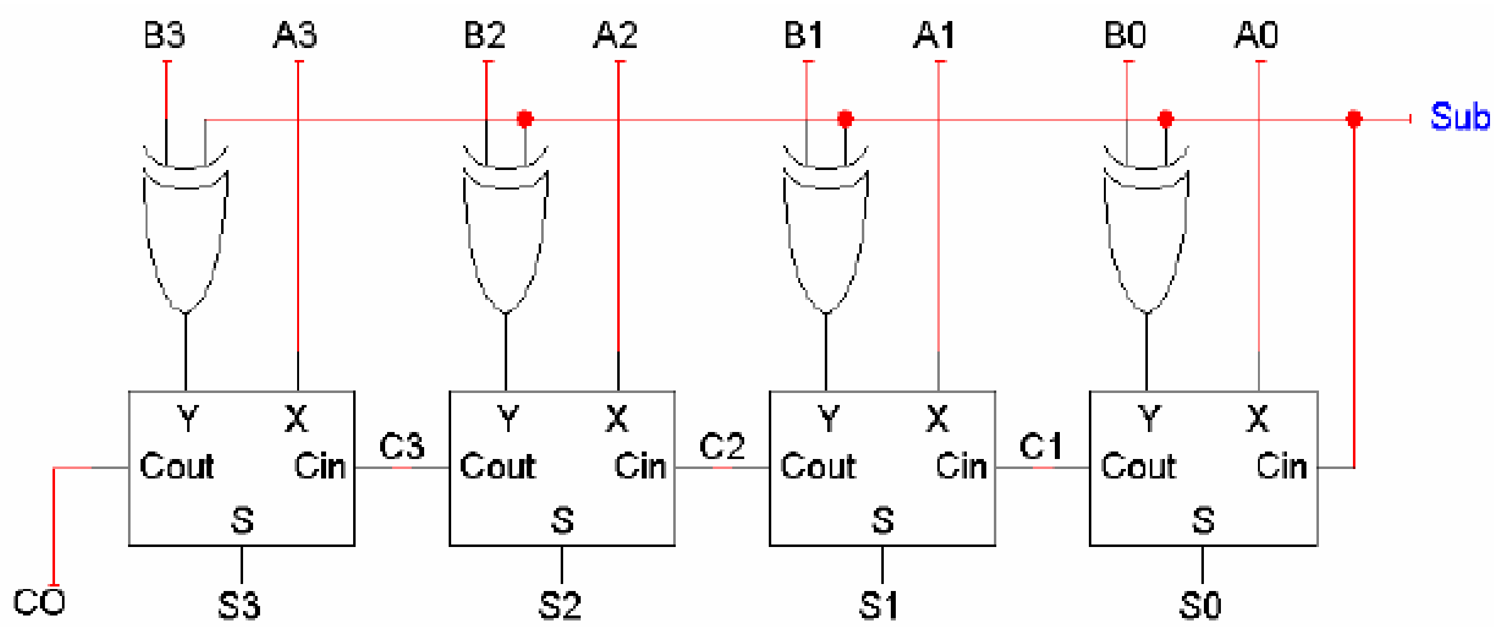
- There are only two differences between an adder and subtractor circuit.
  - The subtractor has to negate B3 B2 B1 B0.
  - The subtractor sets the initial carry in to 1, instead of 0.
- It's not hard to make one circuit that does *both* addition and subtraction.

An adder-subtractor circuit

- XOR gates let us selectively complement the B input.

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

- 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)?

	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)

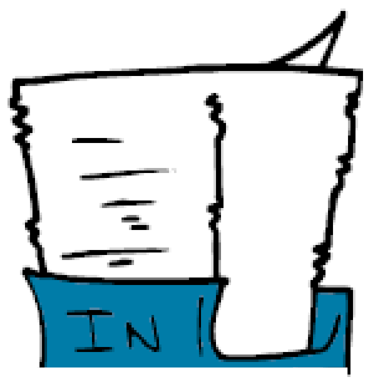
- Signed overflow is very different from unsigned overflow.
  – The carry out is not enough to detect overflow. In the example on the left, the carry out is 0 but there *is* overflow.

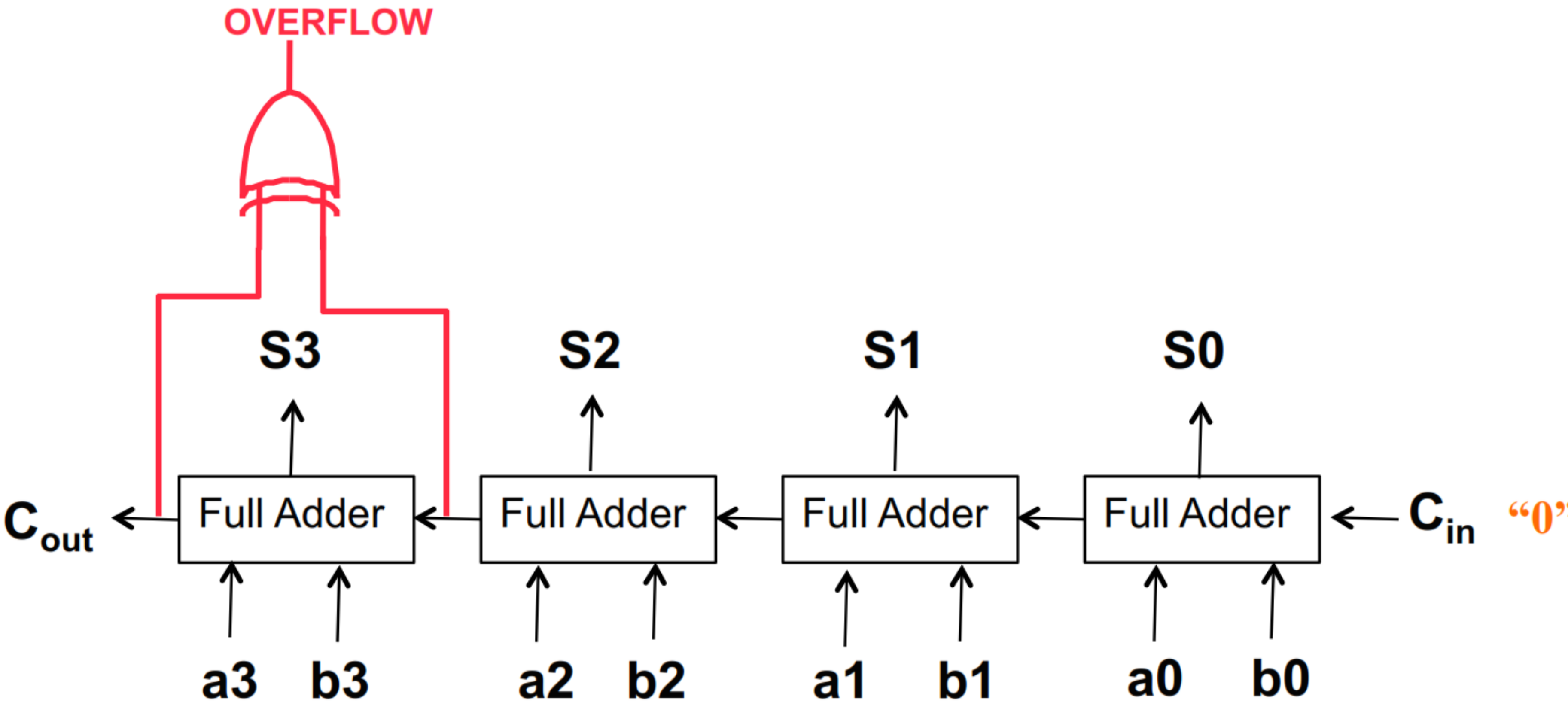
Detecting signed overflow

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

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

- 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?)




OVERFLOW
S3
S2
S1
S0
C_{out}
Full Adder
Full Adder
Full Adder
Full Adder
C_{in}
“0”
a3
b3
a2
b2
a1
b1
a0
b0

Overflow

Example1:

$$\begin{array}{rl} 0100000 & \\ 0110101_2 & (=\ 53_{10}) \\ \underline{+0101010_2} & (=\ 42_{10}) \\ 1011111_2 & (=-33_{10}) \end{array}$$

Example2:

$$\begin{array}{rl} 1000000 & \\ 1010101_2 & (=-43_{10}) \\ \underline{+1001010_2} & (=-54_{10}) \\ 0011111_2 & (=\ 31_{10}) \end{array}$$

Example3:

$$\begin{array}{rl} 1100000 & \\ 0110101_2 & (=\ 53_{10}) \\ \underline{+1101010_2} & (=-22_{10}) \\ 0011111_2 & (=\ 31_{10}) \end{array}$$

Example4:

$$\begin{array}{rl} 0000000 & \\ 0010101_2 & (=\ 21_{10}) \\ \underline{+0101010_2} & (=\ 42_{10}) \\ 0111111_2 & (=\ 63_{10}) \end{array}$$

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	$\longrightarrow$	0000 0101	(+5)
(-4)	1100	$\longrightarrow$	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.