OVERFLOW

When adding numbers together using the 2's complement notation:
Add the numbers together in the usual way as if they are just normal binary numbers. When dealing with 2's complement, any bit pattern that has a sign bit of zero in other words, a ***positive*** number) is just the same as a normal binary number (you don't need to convert it back out of 2's complement in any way, just convert it straight into decimal as you would convert a normal binary number).If, on the other hand, the sign bit is 1, this means that the corresponding decimal number is ***negative***, and the bit pattern needs to be converted out of 2's complement before you can convert it from binary into decimal.Look at the 2's complement tables below; notice how all numbers from zero upwards are exactly the same as the usual binary representation for each value. Only the negative values (i.e. all those bit patterns starting with a 1) do not correspond to the usual binary representation, so they have to be converted from 2's complement notation into normal binary before we can then convert them from binary to decimal. **Two's complement using patterns of length 3 and 4 are:**

Bit pattern	Value represented
011	3
010	2
001	1
000	0
111	-1
110	-2
101	-3
100	-4

Bit pattern	Value represented
0111	7
0110	6
0101	5
0100	4
0011	3
0010	2
0001	1
0000	0
1111	-1
1110	-2
1101	-3
1100	-4
1011	-5
1010	-6
1001	-7
1000	-8

If an addition operation produces a result that exceeds the range of the number system, overflow is said to occur. In the modular counting representation shown above, overflow occurs during the addition of positive numbers when we count past +7. Addition of two numbers with different signs can never produce overflow, but addition of two numbers of like sign can, as shown by the following examples

```
   -3        1101
   -6        1010
+______   +______
   -9        10111 = +7
```

```
   +5        0101
   +6        0110
+______   +______
   +11        1011 = -5
```

```
   -8        1000
   -8        1000
+______   +______
   -16       10000 = +0
```

```
   +7        0111
   +7        0111
+______   +______
   +14        1110 = -2
```

Fortunately, there is a simple rule for detecting overflow in addition: an addition overflows if the signs of the addends are the same and the sign of the sum is different from the addend's sign. The overflow rule is sometimes state in terms of carries generated during the addition operation: an addition overflows if the carry bits c_{in} into and c_{out} of the sign position are different.

Overflow Detection for Adders

In our discussion of ripple-carry adders, we said that the same adder that adds two k-bit UB bitstrings also adds two k-bit 2C bitstrings.

That was one advantage of using 2C for signed int representation.

However, detecting overflow is different for the two representations. We look adding additional logic to the ripple carry adder, which can detect overflow for UB addition and for 2C addition.

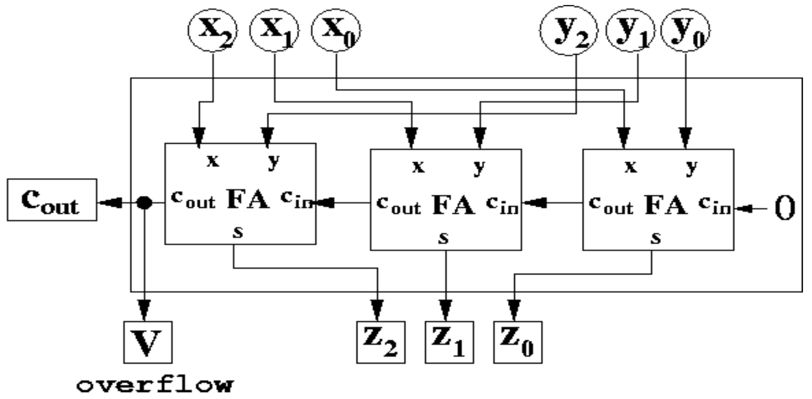
Overflow in UB addition

When adding two k-bit bitstrings using UB, you are adding two non-negative numbers. Let's call these numbers **x** and **y**. Since they are non-negative, then **x + y >= x** and **x + y >= y**. That is, adding two non-negative numbers either keeps the value the same, or increases it.

Assuming both **x** and **y** are valid 2C bitstrings, then their sum overflows only if the result is larger than the largest positive integer. For **k** bits, the maximum value is $\mathbf{2^k - 1}$.

If a sum is larger than that, it overflows. When a sum is larger than $\mathbf{2^k - 1}$, it takes **k + 1** bits to represent the result. So, you can tell overflow just by checking if the carry out of the leftmost adder (which adds the most significant bits and the carry in) is 1.

Here's how the circuit looks:



The letter "V" is often used to represent the overflow bit. Presumably "O" is not used because it looks too much like the numeral 0.

Overflow in 2C addition

The condition for overflow is different if the bitstring representation is 2C. In particular, **x + y >= x** and **x + y >= y** are no longer necessarily true.

Here are some facts about overflow in 2C.

- If **x** and **y** have opposite signs (one is negative, the other is non-negative), then the sum will never overflow. Just try it out. The result will either be **x** or **y** or somewhere in between.
- Thus, overflow can only occur when **x** and **y** have the same sign.
- One way to detect overflow is to check the sign bit of the sum. If the sign bit of the sum does not match the sign bit of **x** and **y**, then there's overflow. This only makes sense.
- Suppose **x** and **y** both have sign bits with value 1. That means, both representations represent negative numbers. If the sum has sign bit 0, then the result of adding two negative numbers has resulted in a non-negative result, which is clearly wrong. Overflow has occurred.
- Suppose **x** and **y** both have sign bits with value 0. That means, both representations represent non-negative numbers. If the sum has sign bit 1, then the result of adding two non-negative numbers has resulted in a negative result, which is clearly wrong. Overflow has occurred.

  So that would suggest that one way to detect overflow is to look at the sign bits of the two most signicant bits and compare it to the sum.

Let's derive a formula for this. Let's say we want to add two **k** bit numbers: $\mathbf{x_{k-1}...x_0}$ and $\mathbf{y_{k-1}...y_0}$. The sum is $\mathbf{s_{k-1}...s_0}$.

Thus, one formula for detecting overflow is:

$V = x_{k-1}y_{k-1}\backslash s_{k-1} + \backslash x_{k-1}\backslash y_{k-1}s_{k-1}$

This formula basically says that overflow has occurred if the sign bit **x** and **y** are 1, and the sign bit of the sum is 0, or if the sign bit of **x** and **y** are 0, and the sign bit of the sum is 1.

A Simpler Formula for Overflow

However, there is an easier formula, though one that is more obscure. Let the carry out of the full adder adding the least significant bit be called $\mathbf{c_0}$. Then, the carry out of the full adder adding the next least significant bit is $\mathbf{c_1}$. Thus, the carry out of the full adder adding the most significant bits is $\mathbf{c_{k-1}}$. This assumes that we are adding two **k** bit numbers.

We can write the formula as:

$V = c_{k-1} \text{ XOR } c_{k-2}$

This is effectively XORing the carry-in and the carry-out of the leftmost full adder.

Why does this work? The XOR of the carry-in and carry-out differ if there's either a 1 being carried in, and a 0 being carried out, or if there's a 0 being carried in, and a 1 being carried out.

When does that happen? Let's look at each case:

Case 1: 0 carried in, and 1 carried out

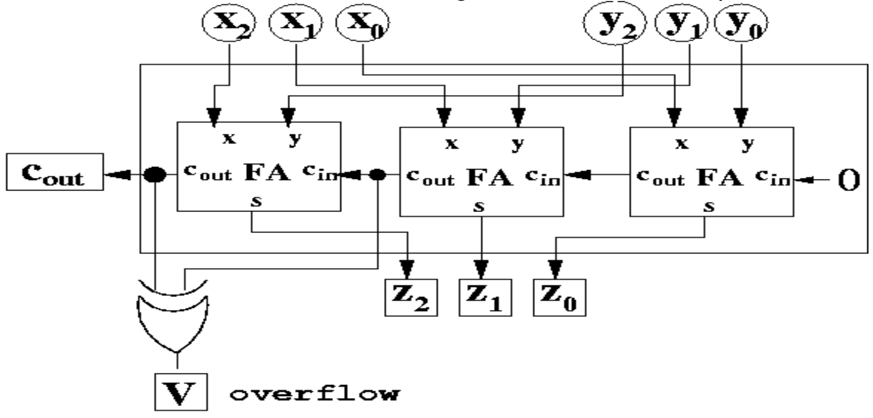
If a 0 is carried, then the only way that 1 can be carried out is if $\mathbf{x_{k-1} = 1}$ and $\mathbf{y_{k-1} = 1}$. That way, the sum is 0, and the carry out is 1. This is the case when you add two negative numbers, but the result is non-negative.

Case 1: 1 carried in, and 0 carried out

The only way 0 can be carried out if there's a 1 carried in is if $\mathbf{x_{k-1} = 0}$ and $\mathbf{y_{k-1} = 0}$. In that case, 0 is carried out, and the sum is 1. This is the case when you add two non-negative numbers and get a negative result.

The Circuit

Here's the circuit that detects overflow when adding two k-bit numbers represented in 2C.



Notice the XOR gate at the bottom, which is exclusive-ORing the carry-in of the full adder of most significant bits to the carry-out.

Misconceptions about Overflow

Overflow occurs when you do some operation to two valid representations, and the result can not be represented in the representation because the value is too large or too smal.

Overflow detection is detecting overflow for a specific representation. Too often people mistake overflow detection for overflow. Thus, students say "overflow is when the carry out is a 1". Specific detection of overflow requires knowing the operation *and* the representatin.

Thus, "overflow is when the carry out is a 1" is only correct when the representation is UB and the operation is addition. In this case, I claim that this definition is how to detect overflow for UB addition.

Overflow detection for 2C addition is different. One way to detect it is to XOR the carry-in and the carry-out. We also discussed another Boolean expression for detecting overflow.