<html><head></head><body><pre style="word-wrap: break-word; white-space: pre-wrap;">===

The CARRY flag and OVERFLOW flag in binary arithmetic

===

- Ian! D. Allen - idallen@idallen.ca - www.idallen.com

Do not confuse the "carry" flag with the "overflow" flag in integer arithmetic. Each flag can occur on its own, or both together. The CPU's ALU doesn't care or know whether you are doing signed or unsigned mathematics; the ALU always sets both flags appropriately when doing any integer math. The ALU doesn't know about signed/unsigned; the ALU just does the binary math and sets the flags appropriately. It's up to you, the programmer, to know which flag to check after the math is done.

If your program treats the bits in a word as unsigned numbers, you must watch to see if your arithmetic sets the carry flag on, indicating the result is wrong. You don't care about the overflow flag when doing unsigned math. (The overflow flag is only relevant to signed numbers, not unsigned.)

If your program treats the bits in a word as two's complement signed values, you must watch to see if your arithmetic sets the overflow flag on, indicating the result is wrong. You don't care about the carry flag when doing signed, two's complement math. (The carry flag is only relevant to unsigned numbers, not signed.)

In unsigned arithmetic, watch the carry flag to detect errors.
In unsigned arithmetic, the overflow flag tells you nothing interesting.

In signed arithmetic, watch the overflow flag to detect errors.
In signed arithmetic, the carry flag tells you nothing interesting.

English

Do not confuse the English verb "to overflow" with the "overflow flag" in the ALU. The verb "to overflow" is used casually to indicate that some math result doesn't fit in the number of bits available; it could be integer math, or floating-point math, or whatever. The "overflow flag" is set specifically by the ALU as described below, and it isn't the same as the casual English verb "to overflow".

In English, we may say "the binary/integer math overflowed the number of bits available for the result, causing the carry flag to come on". Note how this English usage of the verb "to overflow" is *not* the same as saying "the overflow flag is on". A math result can overflow (the verb) the number of bits available without turning on the ALU "overflow" flag.

Carry Flag

The rules for turning on the carry flag in binary/integer math are two:

1. The carry flag is set if the addition of two numbers causes a carry out of the most significant (leftmost) bits added.

   ```
   1111 + 0001 = 0000 (carry flag is turned on)
   ```

2. The carry (borrow) flag is also set if the subtraction of two numbers requires a borrow into the most significant (leftmost) bits subtracted.

   ```
   0000 - 0001 = 1111 (carry flag is turned on)
   ```

Otherwise, the carry flag is turned off (zero).

* 0111 + 0001 = 1000 (carry flag is turned off [zero])
* 1000 - 0001 = 0111 (carry flag is turned off [zero])

In unsigned arithmetic, watch the carry flag to detect errors.
In signed arithmetic, the carry flag tells you nothing interesting.

Overflow Flag

The rules for turning on the overflow flag in binary/integer math are two:

1. If the sum of two numbers with the sign bits off yields a result number
   with the sign bit on, the "overflow" flag is turned on.

   0100 + 0100 = 1000 (overflow flag is turned on)

2. If the sum of two numbers with the sign bits on yields a result number
   with the sign bit off, the "overflow" flag is turned on.

   1000 + 1000 = 0000 (overflow flag is turned on)

Otherwise, the overflow flag is turned off.
* 0100 + 0001 = 0101 (overflow flag is turned off)
* 0110 + 1001 = 1111 (overflow flag is turned off)
* 1000 + 0001 = 1001 (overflow flag is turned off)
* 1100 + 1100 = 1000 (overflow flag is turned off)

Note that you only need to look at the sign bits (leftmost) of the three
numbers to decide if the overflow flag is turned on or off.

If you are doing two's complement (signed) arithmetic, overflow flag on
means the answer is wrong - you added two positive numbers and got a
negative, or you added two negative numbers and got a positive.

If you are doing unsigned arithmetic, the overflow flag means nothing
and should be ignored.

The rules for two's complement detect errors by examining the sign of
the result. A negative and positive added together cannot be wrong,
because the sum is between the addends. Since both of the addends fit
within the allowable range of numbers, and their sum is between them, it
must fit as well. Mixed-sign addition never turns on the overflow flag.

In signed arithmetic, watch the overflow flag to detect errors.
In unsigned arithmetic, the overflow flag tells you nothing interesting.

How the ALU calculates the Overflow Flag

This material is optional reading.

There are several automated ways of detecting overflow errors in two's
complement binary arithmetic (for those of you who don't like the manual
inspection method). Here are two:

Calculating Overflow Flag: Method 1

Overflow can only happen when adding two numbers of the same sign and
getting a different sign. So, to detect overflow we don't care about
any bits except the sign bits. Ignore the other bits.

With two operands and one result, we have three sign bits (each 1 or
0) to consider, so we have exactly 2**3=8 possible combinations of the
three bits. Only two of those 8 possible cases are considered overflow.

Below are just the sign bits of the two addition operands and result:

```
       ADDITION SIGN BITS
    num1sign num2sign sumsign
   ---------------------------
        0 0 0
 *OVER* 0 0 1 (adding two positives should be positive)
        0 1 0
        0 1 1
        1 0 0
        1 0 1
 *OVER* 1 1 0 (adding two negatives should be negative)
        1 1 1
```

We can repeat the same table for subtraction. Note that subtracting
a positive number is the same as adding a negative, so the conditions that
trigger the overflow flag are:

```
      SUBTRACTION SIGN BITS
    num1sign num2sign sumsign
   ---------------------------
        0 0 0
        0 0 1
        0 1 0
 *OVER* 0 1 1 (subtracting a negative is the same as adding a positive)
 *OVER* 1 0 0 (subtracting a positive is the same as adding a negative)
        1 0 1
        1 1 0
        1 1 1
```

A computer might contain a small logic gate array that sets the overflow
flag to "1" iff any one of the above four OV conditions is met.

A human need only remember that, when doing signed math, adding
two numbers of the same sign must produce a result of the same sign,
otherwise overflow happened.

Calculating Overflow Flag: Method 2

When adding two binary values, consider the binary carry coming into
the leftmost place (into the sign bit) and the binary carry going out
of that leftmost place. (Carry going out of the leftmost [sign] bit
becomes the CARRY flag in the ALU.)

Overflow in two's complement may occur, not when a bit is carried out
out of the left column, but when one is carried into it and no matching
carry out occurs. That is, overflow happens when there is a carry into
the sign bit but no carry out of the sign bit.

The OVERFLOW flag is the XOR of the carry coming into the sign bit (if
any) with the carry going out of the sign bit (if any). Overflow happens
if the carry in does not equal the carry out.

Examples (2-bit signed 2's complement binary numbers):

```
    11
   +01
   ===
    00

   - carry in is 1
   - carry out is 1
   - 1 XOR 1 = NO OVERFLOW
```

```
     01
    +01
    ===
     10

    - carry in is 1
    - carry out is 0
    - 1 XOR 0 = OVERFLOW!


     11
    +10
    ===
     01

    - carry in is 0
    - carry out is 1
    - 0 XOR 1 = OVERFLOW!


     10
    +01
    ===
     11

    - carry in is 0
    - carry out is 0
    - 0 XOR 0 = NO OVERFLOW
```

Note that this XOR method only works with the *binary* carry that goes
into the sign *bit*. If you are working with hexadecimal numbers, or
decimal numbers, or octal numbers, you also have carry; but, the carry
doesn't go into the sign *bit* and you can't XOR that non-binary carry
with the outgoing carry.

Hexadecimal addition example (showing that XOR doesn't work for hex carry):

```
     8Ah
    +8Ah
    ====
     14h
```

The hexadecimal carry of 1 resulting from A+A does not affect the
sign bit. If you do the math in binary, you'll see that there is
no carry *into* the sign bit; but, there is carry out of the sign
bit. Therefore, the above example sets OVERFLOW on. (The example
adds two negative numbers and gets a positive number.)

```
--
| Ian! D. Allen  -  idallen@idallen.ca  -  Ottawa, Ontario, Canada
| Home Page: http://idallen.com/   Contact Improv: http://contactimprov.ca/
| College professor (Free/Libre GNU+Linux) at: http://teaching.idallen.com/
| Defend digital freedom:  http://eff.org/  and have fun:  http://fools.ca/
```

</pre></body></html>