*Overflow Flag in PixieVM Addition*

**Description**

Overflow is a condition in signed binary arithmetic where the results of an operation cannot be represented in the resultant signed value. We restrict the discussion to 8-bit addition.

An 8-bit signed number can represent any integer in the range [-128, +127].

For example, if a and b are both 8-bit signed integers with a=-128 and b=-1, then the result:

R = a + b will result in an overflow condition, because the result R = -129 cannot be stored in an 8-bit signed integer. The CPU uses an overflow flag (V) for this condition.

In the above example, the value 127 is stored in R, and the CPU overflow flag (V) is set high.

This document hopes to explain (to me) how the overflow flag is computed during an addition operation.

Example: **adc al, bl**

Pseudo C-style implementation code:

|  |
| --- |
| unsigned int t = al + bl + GET\_CARRY(); |
| V = ((t ^ al) & NEG\_FLAG) && !((al ^ bl) & NEG\_FLAG); |

We note that GET\_CARRY() returns 0 or 1 based on whether the carry flag is set in the CPU FLAGS register. NEG\_FLAG is the sign flag in the CPU FLAGS register = 0x80, which is also the sign bit of an 8-bit signed value. Also note, we use the C-operators, ^ for exclusive-or, & for bitwise-AND, && for logical-AND, and ! for logical-NOT.

NOTES:

* If **((t ^ al) & NEG\_FLAG)** is *true*, it implies that one and only one of **t** and **al** is negative.
* **!((al ^ bl) & NEG\_FLAG)** implies that **al** and **bl** must be both positive or both negative for the statement to be true.

Based on the above notes, for the overflow flag (V) to be set, the following conditions must hold:

* **t** and **al** have opposing signs.
* **al** and **bl** have the same sign.

**Observations**

* First we notice that if **al**, and **bl** have opposing signs, then the result cannot overflow.

For example (-128) + 127 = -1.

* If **al**, **bl** have opposing signs, but **t = al + bl + C** also has a sign opposite of **al**, then this implies that the overflow was due to a carry. For example, consider a=-1, b=1:

If , **t = al + bl + C = 1**, then both **al** and **bl** have opposing signs, and both **al** and **t** have opposing signs. Clearly, C = 1.

In this special case, it seems we do not want to set V=1, if the overflow was the result of a carry. Why? **Because carry is ignored with signed addition**.

**Conclusion**

Based on the analysis of what I have seen done, the above code computes overflow based on the two following conditions:

* The 16-bit result **t = al + bl + C** has a sign opposite that of **al**.
* **!((al ^ bl) & NEG\_FLAG)** ensures the above overflow was not the result of carry. This condition explicitly guards against setting the overflow flag in this case.