Overflow

The overflow flag is somewhat difficult to understand. To make things more complex, the Z80 uses the same flag bit for parity and overflow.

As the documentation explains, the overflow flag indicates and overflow condition where the result in the accumulator is greater than the maximum possible number (+127) or less than the minimum possible number (-128).

The documentation says the overflow condition is determined by examining the sign bits of the operands. If two operands are added with the same sign and the result contains a different sign, the overflow flag is set.

For subtraction, overflow happens if the two arguments have different signs and the minuend (first argument) has a different sign from the result. (That is, if you start with a positive number, subtract a negative number, and end with a negative number, overflow has occurred. Likewise, if you start with a negative number, subtract a positive number, and end with a positive number, overflow has occurred.)

The documentation also states that overflow can be observed if the carry in from the sign bit is different from the carry out of the sign bit, then overflow has occurred. This in fact is how the chip implements overflow in hardware.

Note that overflow only has meaning for signed, twos-complement numbers. For unsigned arithmetic, overflow has no meaning (the carry flag is what you should examine).

Understanding twos-complement numbers

The ones-complement of a value simply flips all the bits in the value: 0 changes to 1 and 1 changes to 0. The ones-complement of an 8-bit value N is 255-N as can be easily proved.

It turns out that the twos-complement of a value is often more useful. The twos-complement is simply the ones-complement + 1. For an 8-bit value N,

this equals 256-N.

The twos complement is useful because adding M and the twos-complement of N is the same as subtracting N from M. For example, to compute 80 - 112, simply take the twos complement of 112 (binary 10010000) and add it to 80 (binary 01010000), yielding (binary 11100000). This result is the twos complement of 32, indicating -32.

Note that 80+144 and 80-112 had exactly the same bit-level operations - only the interpretation of the bits was different. This is why twos complement numbers are so useful - the same addition circuit works with them.

To see why twos complement numbers work this way, consider M + (-N) or M - N

M - N → M - N + 256 Adding 256 doesn't change the 8-bit value.
= M + (256 - N) Simple algebra.
= M + twos complement of N Definition of twos complement.

Thus, adding the twos complement is the same as subtracting. (With the exception of the carry bit, which is affected by the extra 256. This will be discussed later)

Twos complement numbers can be used in several ways. First, subtraction is implemented in the Z80 (and most chips) by taking the twos complement and performing addition. Thus, the ALU doesn't need to explicitly implement subtraction, but can use the same adder for addition and subtraction.

Second, signed numbers can be represented by using twos-complement numbers. In this case, an 8-bit value can hold a number between -128 and 127.

Because the range of signed numbers is restricted, problems can happen. For instance, adding 80 + 80 yields 160. This value fits in 8 bits if unsigned numbers are used, but the sum is too large for an 8-bit signed number. It will get interpreted as -96, which is wrong. The Z80 indicates this problem by

setting the overflow bit.

A key thing to remember is the difference between signed and unsigned numbers is purely in their interpretation. The Z80 doesn't distinguish between signed and unsigned numbers, and they are treated exactly the same.

Note that overflow cannot occur when adding a positive number and a negative number. It can only occur when adding two positive numbers or two negative numbers.

Also note that overflow is very different from carry. A carry can occur even when there is no signed overflow. For instance 2 plus -1 is 00000010 plus 11111111 yielding 0000001 and a carry. Even though the result easily fits in one byte, a carry is generated.

Overflow from subtraction
Internally the Z80 performs subtraction by adding the complement. Thus, overflow from subtraction can be understood by taking the complement and performing an addition. In this case, overflow is computed as above.

It may be easier to understand overflow from subtraction by considering subtraction separately, though.

An example of a subtraction causing overflow is -80 minus 80, which yields -160. Since this value doesn't fit into the range -128 to 127, an overflow is indicated.

For subtraction, the carry flag is set if there is a borrow. A borrow is indicated by the lack of a carry when the complement is added. In other words, for a subtraction, the Z80 adds the complement and then stores the complement of the resulting carry into the carry flag. (This is in contrast to the 6502, which stores the complement of borrow in the carry flag. That is, the carry flag always stores the carry from the ALU.) The Z80 uses an XOR gate to flip the ALU's carry output before storing it in the carry flag for a subtraction.

How the parity / overflow flag is computed

Conceptually, the parity value is straightforward. Even parity (an even number of 1 bits) has a flag value of 1, and odd parity (an odd number of 1 bits) has a flag value of 0.

The parity computation is more complex than you'd expect. Mathematically it's just all the bits of the ALU result exclusive-or'd together and complemented. However, the circuitry doesn't work that way. One complication is that since the ALU is 4 bits, parity must be computed in two passes. But even taking that into account, the circuit is unexpectedly complex.

Parity is computed as follows. In T1 the parity flag latch is cleared. In T2, the flag latch value is exclusive-or'd with ALU result bits 0, 1, and 2 (but not 3), and stored back in the flag latch. In T3, the flag latch value is exclusive-or'd with ALU result bits 4, 5, and 6 (but not 7). In parallel, result bits 3 and 7 are exclusive-or'd together. The two results are exclusive-or'd together, finally yielding the exclusive or of all the values. This value is stored into the flag. It's unclear why the circuitry is implemented this way. Perhaps the propagation delay was too high if 4 bits were combined in sequence.

For addition and subtraction, the value stored in the flag is overflow, not parity. The overflow value is computed simply by the exclusive-or of the carry in to bit 7 (the sign bit) and the exclusive-or out of bit 7.

One unexpected case is LD A, I and LD A, R use the overflow circuit, except the value is pulled low based on IFF2. This is an unexpected hack.