SUBSTRACTION AND MULTIPLICATION

SUBTRACTION OF SIGNED NUMBERS

Subtract the following signed numbers

a)15-5 b)15-35

11110 000 (borrow)
0000 1111 (15)
− 1111 1011 (−5)
==========
0001 0100 (20)

Overflow is detected the same way as for addition, by examining the two leftmost (most significant) bits of the borrows; overflow has occurred if they are different.

b) 15 − 35 = −20:

11100 0000 (borrow)
0000 1111 (15)
− 0010 0011 (35)
==========
1110 1100 (−20)

As for addition, overflow in subtraction may be avoided (or detected after the operation) by first sign-extending both inputs by an extra bit.

Adding two's-complement numbers requires no special processing if the operands have opposite signs: the sign of the result is determined automatically. For example, adding 15 and -5:

```
 11111 111  (carry)
  0000 1111 (15)
+ 1111 1011 (-5)
==================
  0000 1010 (10)
```

This process depends upon restricting to 8 bits of precision; a carry to the (nonexistent) 9th most significant bit is ignored, resulting in the arithmetically correct result of 10_{10}.

The last two bits of the carry row (reading right-to-left) contain vital information: whether the calculation resulted in an arithmetic overflow, a number too large for the binary system to represent (in this case greater than 8 bits). An overflow condition exists when a carry (an extra 1) is generated out of the far left bit (the MSB), but not into the MSB. As mentioned above, the sign of the number is encoded in the MSB of the result.

In other terms, if the left two carry bits (the ones on the far left of the top row in these examples) are both 1's or both 0's, the result is valid; if the left two carry bits are "1 0" or "0 1", a sign overflow has occurred. Conveniently, an XOR operation on these two bits can quickly determine if an overflow condition exists. As an example, consider the 4-bit addition of 7 and 3:

```
  0111   (carry)
```

```
  0111 (7)
+ 0011 (3)
=============
  1010 (−6) invalid!
```

In this case, the far left two (MSB) carry bits are "01", which means there was a two's-complement addition overflow. That is, $1010_2 = 10_{10}$ is outside the permitted range of −8 to 7.

In general, any two *n*-bit numbers may be added *without* overflow, by first sign-extending both of them to *n*+ 1 bit, and then adding as above. The *n*+1 bit result is large enough to represent any possible sum (*e.g.*, 5 bits can represent values in the range −16 to 15) so overflow will never occur. It is then possible, if desired; to 'truncate' the result back to *n* bits while preserving the value if and only if the discarded bit is a proper sign extension of the retained result bits. This provides another method of detecting overflow—which is equivalent to the method of comparing the carry bits—but which may be easier to implement in some situations, because it does not require access to the internals of the addition.

MULTIPLICATION

The product of two *n*-bit numbers requires 2*n* bits to contain all possible values. If the precision of the two two's-complement operands is doubled before the multiplication, direct multiplication (discarding any excess bits beyond that precision) will provide the correct result. For example, take 6 × −5 = −30. First, the precision is extended from 4 bits to 8. Then the numbers are multiplied, discarding the bits beyond 8 (shown by 'x'):

```
  00000110 (6)
× 11111011 (-5)
==========
   110
  110
  0
```

110

110

110

x10

xx0

==========

xx11100010 (-30)