Chapter 6 -- integer arithmetic

```
all about integer arithmetic.
-------------------------------------


operations we'll get to know (and love):
    addition
    subtraction
    multiplication
    division
    logical operations (not, and, or, nand, nor, xor, xnor)
    shifting


the rules for doing the arithmetic operations vary depending
on what representation is implied.

A LITTLE BIT ON ADDING
----------------------
  an overview.


        carry in  a  b | sum  carry out
        ---------------+----------------
           0      0  0 |  0     0
           0      0  1 |  1     0
           0      1  0 |  1     0
           0      1  1 |  0     1
           1      0  0 |  1     0
           1      0  1 |  0     1
           1      1  0 |  0     1
           1      1  1 |  1     1
                       |

          a       0011
         +b      +0001
         --      -----
        sum       0100


  its really just like we do for decimal!
    0 + 0 = 0
    1 + 0 = 1
    1 + 1 = 2  which is 10 in binary, sum is 0 and carry the 1.
    1 + 1 + 1 = 3  sum is 1, and carry a 1.




ADDITION
--------

unsigned:
  just like the simple addition given.

  examples:
```

```
      100001           00001010 (10)
     +011101          +00001110 (14)
     -------          ---------
      111110           00011000 (24)

   ignore (throw away) carry out of the msb.
   Why?  Because computers ALWAYS work with a fixed precision.
```

```
 sign magnitude:

   rules:
     - add magnitudes only (do not carry into the sign bit)
     - throw away any carry out of the msb of the magnitude
       (Due, again, to the fixed precision constraints.)
     - add only integers of like sign ( + to +    or    - to -)
     - sign of result is same as sign of the addends

   examples:

     0  0101 (5)        1  1010 (-10)
   + 0  0011 (3)      + 1  0011 (-3)
   ---------          ---------
     0  1000 (8)        1  1101 (-13)


     0  01011 (11)
   + 1  01110 (-14)
   ----------------
   don't add! must do subtraction!
```

```
 one's complement:

    by example


     00111 (7)         111110 (-1)           11110 (-1)
   + 00101 (5)       + 000010 (2)          + 11100 (-3)
   -----------       ------------          ------------
     01100 (12)      1 000000 (0) wrong!   1 11010 (-5) wrong!
                         +  1                  +  1
                       ----------            ----------
                       000001 (1) right!     11011 (-4) right!


    so it seems that if there is a carry out (of 1) from the msb, then
    the result will be off by 1, so add 1 again to get the correct
    result.  (Implementation in HW called an "end around carrry.")
```

```
 two's complement:

    rules:
```

```
     - just add all the bits
     - throw away any carry out of the msb
     - (same as for unsigned!)

   examples


    00011 (3)         101000               111111 (-1)
  + 11100 (-4)      + 010000             + 001000 (8)
  ------------      --------             --------
    11111 (-1)        111000             1 000111 (7)


 after seeing examples for all these representations, you may see
 why 2's complement addition requires simpler hardware than
 sign mag. or one's complement addition.



 SUBTRACTION
 -----------
   general rules:
     1 - 1 = 0
     0 - 0 = 0
     1 - 0 = 1
    10 - 1 = 1
     0 - 1 = borrow!




 unsigned:

   -  it only makes sense to subtract a smaller number from a larger one

   examples


      1011 (11)   must borrow
   -  0111 (7)
   ------------
      0100 (4)



 sign magnitude:

   - if the signs are the same, then do subtraction
   - if signs are different, then change the problem to addition
   - compare magnitudes, then subtract smaller from larger
   - if the order is switched, then switch the sign too.

   - when the integers are of the opposite sign, then do
         a - b    becomes   a + (-b)
         a + b    becomes   a - (-b)


     examples

    0 00111 (7)             1 11000 (-24)
  - 0 11000 (24)          - 1 00010 (-2)
```

```
 --------------          --------------
                           1 10110 (-22)
do 0 11000 (24)
 - 0 00111 (7)
 --------------
   1 10001 (-17)
   (switch sign since the order of the subtraction was reversed)
```

one's complement:

     figure it out on your own

two's complement:

  - change the problem to addition!

       a - b becomes   a + (-b)

  - so, get the additive inverse of b, and do addition.

    examples

```
   10110 (-10)
 - 00011 (3)    -->       00011
 ------------               |
                           \|/
                          11100
                        +     1
                        -------
                          11101 (-3)
 so do

     10110 (-10)
   + 11101 (-3)
   ------------
  1  10011  (-13)
  (throw away carry out)
```

OVERFLOW DETECTION IN ADDITION

  unsigned -- when there is a carry out of the msb

```
           1000 (8)
          +1001 (9)
          -----
         1 0001 (1)
```

  sign magnitude -- when there is a carry out of the msb of the magnitude

```
         1 1000 (-8)
       + 1 1001 (-9)
          -----
         1 0001 (-1)  (carry out of msb of magnitude)
```

```
   2's complement -- when the signs of the addends are the same, and the
                     sign of the result is different


            0011 (3)
          + 0110 (6)
          ----------
            1001 (-7)   (note that a correct answer would be 9, but
                         9 cannot be represented in 4 bit 2's complement)

   a detail -- you will never get overflow when adding 2 numbers of
               opposite signs


 OVERFLOW DETECTION IN SUBTRACTION

    unsigned -- never
    sign magnitude -- never happen when doing subtraction
    2's complement -- we never do subtraction, so use the addition rule
       on the addition operation done.




 MULTIPLICATION of integers

     0 x 0 = 0
     0 x 1 = 0
     1 x 0 = 0
     1 x 1 = 1

     -- longhand, it looks just like decimal

     -- the result can require 2x as many bits as the larger multiplicand

     -- in 2's complement, to always get the right answer without
        thinking about the problem, sign extend both multiplicands to
        2x as many bits (as the larger).  Then take the correct number
        of result bits from the least significant portion of the result.


    2's complement example:


              1111 1111        -1
            x 1111 1001     x  -7
         ----------------    ------
               11111111         7
              00000000
             00000000
            11111111
           11111111
          11111111
         11111111
    +   11111111
        ----------------
         1  00000000111
               --------  (correct answer underlined)
```

```
        0011 (3)              0000 0011 (3)
      x 1011 (-5)           x 1111 1011 (-5)
      ------                -----------
        0011                   00000011
       0011                   00000011
      0000                   00000000
   + 0011                   00000011
   ---------               00000011
     0100001              00000011
  not -15 in any         00000011
   representation!     + 00000011
                      ------------------
                             1011110001

                         take the least significant 8 bits 11110001 = -15



DIVISION of integers
     unsigned only!

     example of 15 / 3         1111 / 011

     To do this longhand, use the same algorithm as for decimal integers.



LOGICAL OPERATIONS
   done bitwise

                    X =  0011
                    Y =  1010

       X  AND  Y is       0010
       X   OR  Y is       1011
       X  NOR  Y is       0100
       X  XOR  Y is       1001
                etc.


SHIFT OPERATIONS
  a way of moving bits around within a word

  3 types:     logical, arithmetic and rotate
               (each type can go either left or right)

  logical left - move bits to the left, same order
               - throw away the bit that pops off the msb
               - introduce a 0 into the lsb


               00110101

               01101010 (logically left shifted by 1 bit)


  logical right - move bits to the right, same order
               - throw away the bit that pops off the lsb
               - introduce a 0 into the msb

               00110101
```

```
              00011010  (logically right shifted by 1 bit)


  arithmetic left - move bits to the left, same order
              - throw away the bit that pops off the msb
              - introduce a 0 into the lsb
              - SAME AS LOGICAL LEFT SHIFT!



  arithmetic right - move bits to the right, same order
              - throw away the bit that pops off the lsb
              - reproduce the original msb into the new msb
              - another way of thinking about it:  shift the
                bits, and then do sign extension

              00110101 ->  00011010 (arithmetically right shifted by 1 bit)

              1100 -> 1110   (arithmetically right shifted by 1 bit)



  rotate left - move bits to the left, same order
             - put the bit that pops off the msb into the lsb,
               so no bits are thrown away or lost.

              00110101 -> 01101010 (rotated left by 1 place)
              1100 -> 1001 (rotated left by 1 place)


  rotate right - move bits to the right, same order
             - put the bit that pops off the lsb into the msb,
               so no bits are thrown away or lost.

              00110101 -> 10011010 (rotated right by 1 place)
              1100 ->  0110 (rotated right by 1 place)
```

SASM INSTRUCTIONS FOR LOGICAL AND SHIFT OPERATIONS

SASM has instructions that do bitwise logical operations and shifting operations.

```
     lnot   x          x <-  NOT (x)
     land   x, y       x <-  (x) AND (y)
     lor    x, y       x <-  (x) OR (y)
     lxor   x, y       x <-  (x) XOR (y)

     llsh  x     x <- (x), logically left shifted by 1 bit
     rlsh  x     x <- (x), logically right shifted by 1 bit
     rash  x     x <- (x), arithmetically right shifted by 1 bit
     rror  x     x <- (x), rotated right by 1 bit
```