120-final-studyGuide

An encoding chooses which values to represent objects. If we have strings of n bits to work with, we can encode 2^n different objects: each of the n-length bit strings from 00...0 to 11...1 can be used to represent a unique object.

Unsigned binary representation of non-negative integers indicates which powers of two to include:

```
100     says include 4; but not 2 or 1.
100100  says include 32 and 4; but not 16, 8, 2, or 1.
```

Signed representation can be done by extending unsigned binary representation:

```
a leading 0  indicates a positive sign
a leading 1  indicates a negative sign
```

Sign-magnitude binary representation uses a leading sign bit, followed by an unsigned binary representation of a non-negative integer:

```
0100   says  +100  or +4
1100   says  -100   or  -4
```

For negative values, n-bit Twos-Complement (2s-comp) representation inverts the bits of the magnitude portion of sign-magnitude representation, with an adjustment of +1:

```
01   is +1  in sign-magnitude
10   is -0   in sign-magnitude
11   is -1   in 2s-comp
```

In general, to convert a 2s-comp representation of a number x to the 2s-comp representation of -x:

```
invert bits
add 1
```

e.g., suppose x = -6. Then in 2s-comp:

```
x ===> 1010     -x  ===> 0101+1  =  0110
```

e.g., suppose x = +9. Then in 2s-comp:

```
x ===> 01001     -x  ===> 10110+1  =  10111
```

The largest signed value representable in n-bit 2s-comp is represented by the n-bit string 011...1 and has the value,
+(2^(n-1) - 1).

The largest magnitude number representable has the n-bit represetation 100...0 and represents the value,
-2^(n-1).

It is a feature of 2s-comp representation that converting the n-bit 2s-comp representation to (n+1)-bit 2s-comp simply duplicates the leftmost bit. This is called "sign-extension" from n bits to n+1 bits.

E.g. 111 is -1 in 3-bit 2s-comp, and 1111 is -1 in 4-bit 2s-comp.
E.g. 001 is +1 in 3-bit 2s-comp, and 0001 is +1 in 4-bit 2s-comp.

Because we use n-bit registers in our computer hardware, we are limited to n-bit representation. (We can use representations with more than n bits, but we must then play some tricks to keep track of the additional bits.) For n-bit operations, if there is a carry from the n-th bit position, we cannot represent the value correctly. Likewise, we cannot represent a value if there is a borrow from the n-th position.

These error conditions are called "overflow". (There is another error condition called "underflow" that occurs in floating-point arithmetic.)

Unsigned arithmetic overflow will occur if the sum of two n-bit numbers is greater than (2^n - 1), or 11...1 in n-bits. If so, there will be a carry out from the n-th bit position.

For unsigned subtraction, if the result is less than 0 (or 00...0 in n-bits), there will be a borrow into the n-th bit position.

The logic for a 1-bit ADD(A, B, C*in*) is:

S == XOR(A, B, C*in*)

and

C*out* == MAJ(A, B, C*in*)   (MAJ produces a 1 if at least two of its inputs are 1)

We can chain n ADDs together to form an n-bit ADD by connecting C*out* from bit position i to C*in* at bit position (i+1). This is called a "ripple-carry" adder. The C*out* from bit position n can be used as an indicator that there was unsigned overflow. An unsigned SUB is similar.

2s-comp arithmetic in n bits can be done using an n-bit ADD. It is a feature that adding 2s-comp representations as if they were unsigned results in the 2s-comp representation of the sum. In this case, a carry out from the n-th position does not indicate an error. However, if adding two positive values results in a negative value, or adding two negative values results in a positive value, an overflow error has occurred.

Subtraction is also done using an n-bit ADD: Because (A - B) is the same as (A + (-B)), we can simply take the 2s-comp of B and then ADD. Taking the 2s-comp of B is simple because we can invert B's bits, then add 1. That can be accomplished by adding an inverter to each of B's input wires, and setting C*in* == 1 in bit position 0.

The logic delay through an n-bit ripple-carry ADD is the time it takes for C*out* at bit position n to settle to its correct value. Because the carrys are chained together, a change in bit position 0 could cause C*out* to change at bit position n. The time for this is n times the delay for C*out* from a 1-bit ADD. Assuming that delay is d, the total delay of an n-bit ADD is (n X d).

Multiplying unsigned A X B can be done using shifts and ADDs. The k-th partial sum is A left-shifted k times. If the k-th bit of B is 1, the partial sum is added to the accumulated sum; otherwise, it is ignored. An iterated MULT(A, B) left-shifts A one position at a time, and checks the appropriate bit of B to see if an ADD should occur. As a MULT produces 2-n bits from two n-bit inputs, the ADDs are 2-n bits. The delay is then (n X (2n X d)). An array MULT can do the same operation in parallel, using only one 2n-bit ADD, for a delay of a little more than (2n X d).

DIV(A, B) can be done by shifting and unsigned subtraction. If A is greater or equal to B X 2^k, then B goes into A at least 2^k times, and the k-th quotient bit is 1. In that case, the remainder (A - (B X 2^k)) is non-negative. Starting with B left-shifted n times, we subtract from A. If the result is positive, we store the remainder as a partial sum, store the quotient's n-th bit as 1. We then repeat, shifting B right once, subtracting from the partial sum, and storing the (n-1)-th bit of the quotient. We continue to iterate this way for a total of n iterations. The subtractions are 2n-bit, and we do n iterations: delay is (n X (2n X d)).

If we are using signed values, we must convert them to unsigned, do the MULT or DIV, then convert back to signed.

The other major numeric representation is floating-point, which is based on scientific notation such as,

```
  + 6.022 X 10^23 (Avogadro's number)
```

There are four parts: a sign, an exponent, a digit, and a "mantissa" (which is the part after the decimal point). In observations, one very often has to approximate a measurement. For instance, in reading a ruler, we try to see which line is closest to the thing we are measuring. We write down as many digits of observation as we are able to discern. The last digit is assumed to be uncertain: the actual value might be +-1 in the last digit. This is our observational precision. If we are measuring in milimeters, for instance, we might record 367 mm. Translated to meters, this becomes 0.367 meter. We could also write 3.67 10^(-1) meter to put it in "normalized" form.

Notice that if we add two numbers that have errors of 10%,

```
   (70 +- 7) + (60 +- 6)  == 130 +- 13
```

the error in the result is about 10%. For multiplication or division, the % errors add.

Floating point representation is based on the binary version of scientific notation, e.g.,

```
   + 1.011010  X  2^(0101)
```

where the mantissa and exponent are binary. Note that the digit is always 1. We need to decide the tradeoff between the number of bits we have, how much range we want in the exponent, and what % error we want. A balance has been established by the IEEE 754 standard. For 32-bit Floating Point (FP), the format is,

```
   [ S EEEEEEEE mmmmmmmmmmmmmmmmmmmmmmm ]
```

which represents

```
   Sign X 1.mmmmmmmmmmmmmmmmmmmmmmm  X  2^(EEEEEEEE)
```

where Sign is (+1) if S == 0, or (-1) if S == 1; mm...m are the 23 mantissa bits; and EEEEEEEE are the eight exponent bits. The minimum error is +- 2^(-23), which is one part in 2^23 parts,
(or 2^(3) X 2^20, which is one part in about 8 million. Here's a number encoded in 32-bit FP format,

```
   [ 0 00000110 10100000000000000000000 ]
```

which represents

```
   +  2^(+6) X 1.10100000000000000000000    ==  +64 X 1.101    ==  64 + 64( 1/2 + 1/8 )
                                                              ==  64 + 32 + 8
                                                              ==  104
```

There is one complicating twist made to the above format to accommodate rapid sorting: the exponent is represented as the actual exponent plus +127 (or 01111111). That is, the exponent is some signed integer represented in 2s-comp form, which has then had 127 added to it. In the example above,
the "excess-127" encoded exponent would be,

```
        00000110
      + 01111111
==      10000101
```

and the actual representation would be [0 10000101 10100000000000000000000]

FP addition involves 6 steps.
Suppose have A and B represented in FP,

```
[ 0 AAAAAAAA aaaaaaaaaaaaaaaaaaaaaaa ]
[ 0 BBBBBBBB bbbbbbbbbbbbbbbbbbbbbbb ]
```

where capital letters are excess-127 encoded exponent bits, and lower-case letters are mantissa bits.

1) subtract 127 from the exponents of A and B:

```
          AAAAAAAA            BBBBBBBB
         -01111111           -01111111
```

2) shift the digit and mantissa of the number with the smallest power to the right, adding 1 to its exponent each shift, until the exponents are equal.

3) add the digits and mantissas as shifted.

4) shift the result until normalized, adjusting the exponent each shift.

5) round the mantissa to 23 bits (re-normalize, if needed)

6) add 127 to the exponent

An example:

```
  [ 0 10000100 00101100000000000100100 ]   (A)
  [ 0 01111011 11000001100000110000000 ]   (B)

1)    10000100             01111011
     -01111111            -01111111
   == 00000101  (+5)     == 11111100  (-4)

2)(B)  1.11000001100000110000000  ===> 0.000000001110000011000001100000000

3)   1.00101100000000000100100
   + 0.000000001110000011000001100000000
 ==  1.001011001110000100001001100000000

4)   1.001011001110000100001001100000000    (no shifting needed)
5)   1.00101100111000010000101              (round up)
6)      00000101
      + 01111111
    ==  10000100

    [0 100000101 00101100111000010000101 ] (encoded result)
```

So, what difference in exponents would cause A + B == A?
Are there enough bits in the exponent field for that to happen?

The FP add involved 3 8-bit adds, shifting not more than 26 times (w/ guard, round, and sticky bits), one 46-bit add (or less), and rounding.

FP divide and multiply are similar. Here are the steps for multiply:

1) decode exponents, add exponents

2) 24-bit integer multiply of digits and mantissas

3) adjust exponent

4) normalize (shift and adjust exponent)

5) round (re-normalize, if needed)

6) encode exponent

The biggest cost is the integer multiply (divide). But this uses fewer bits than a full integer multiply (divide), and so is faster. We left out one important detail: zero detection. Zero must be detected before any of the steps above occur. The result can be computed immediately in one step. Zero is coded as,

```
[0 00000000 00000000000000000000000 ]  or
[1 00000000 00000000000000000000000 ]
```

There are also denormalized numbers (exponent = 00000000, and mantissa is non-zero), and NaN (exponent is 11111111).

There are many other coding schemes for various kinds of data. Each requires its own processing methods, often implemented in special-purpose hardware. For example,

--- 8-bit ASCII codes for characters and primitive graphics elements

--- 24-bit per-pixel color coding for graphics mode displays

--- 8-bit sound amplitude samples at 8,000 samples per second (PCM digital telephony)

Beyond this are many "lossy" coding schemes that reduce the number of bits needed overall or per second by carefully throwing away some information.