Math 131 notes

Jason Riedy

22 September, 2008

Contents

Notes also available as PDF.

What we will cover from Chapter 4:

- Numbers and digits in different bases, with historical context
- Arithmetic, digit by digit

And additionally, I'll give a brief summary of computer arithmetic.

1 Positional Numbers

A number is a concept and not just a sequence of symbols. We will be discussing ways to express numbers.

Multiple *types* of numbers:

nominal A *nom*inal number is just an identifier (or *name*). In many ways these are just sequences of symbols.

ordinal An *ord*inal number denotes *order*: 1^{st}, 2^{nd}, ...

Adding ordinal or nominal numbers doesn't make sense. This brings up a third type:

cardinal Cardinal numbers *count*.

The name comes from the *cardinality* of sets.

Before our current form:

- Piles of rocks don't work well for merchants.
- **Marks** on sticks, then marks on papyrus.

Marking numbers is costly. A large number becomes a large number of marks. Many marks lead to many errors. Merchants don't like errors. So people started using symbols rather than plain marks.

An intermediate form, **grouping**:

- Egyptian: Different symbols for different levels of numbers: units, tens, hundreds. Grouping within the levels.
- Roman: Symbols for groups, with addition and subtraction of symbols for smaller groups.
- Greek (and Hebrew and Arabic): Similar, but using all their letters for many groups.
- Early Chinese: Denote the number of marks in the group with a number itself...

Getting better, but each system still has complex rules. The main problems are with skipping groups. We now use zero to denote an empty position, but these systems used varying amounts of space. Obviously, this could lead to trade disagreements. Once zeros were adopted, many of these systems persisted in trade for centuries.

Now into forms of positional notation, shorter and more direct:

- Babylonian:
  - Two marks, tens and units.

  - Now the marks are placed by the number of 60s.
  - Suffers from complicated rules about zeros.
  - (Using 60s persists for keeping time...)
- Mayan:
  - Again, two kinds of marks for fives and units.
  - Two positional types: by powers of 20, and by powers of 20 except for one power of 18.
  - (Note that $18 \cdot 20 = 360$, which is much closer to a year.)
  - Essentially equivalent to what we use, but subtraction in Mayan is much easier to see.
- (many other cultures adopted similar systems (*e.g.* Chinese rods)

Current: **Hindu-Arabic numeral system**

The characters differ between cultures, but the idea is the same. The characters often are similar as well. Originated in the region of India and was carried west through trade. No one knows when zero was added to the digits. The earliest firm evidence is in Arab court records regarding a visitor from India and a description of zero from around 776 AD. The first inscription found with a zero is from 876 AD in India. However, the Hindu-Arabic system was not adopted outside mathematics even in these cultures. Merchants kept to a system similar to the Greek and Hebrew systems using letters for numbers.

Leonardo Fibonacci brought the numerals to Europe in the 13^{th} century (after 1200 AD) by translating an Arabic text to Latin. By 15^{th} century, the numeral system was in wide use in Europe. During the 19^{th} century, this system supplanted the rod systems in Asia.

The final value of the number is based on the positions of the digits:

$$1234 = 1 \cdot 10^3 + 2 \cdot 10^2 + 3 \cdot 10^1 + 4 \cdot 10^0.$$

We call ten the **base**. Then numbers becomes polynomials in terms of the base b,

$$1234 = b^3 + 2 \cdot b^2 + 3 \cdot b^1 + 4.$$

Here $b = 10$.

So we moved from marks, where 1000 would require 1000 marks, to groups, where 1000 may be a single mark but 999 may require dozens of marks. Then we moved to positional schemes where the number of symbols depends on the *logarithm* of the value; $1000 = 10^3$ requires $4 = 3 + 1$ symbols.

After looking at other bases, we will look into operations (multiplication, addition, *etc.*) using the base representations.

2 Converting Between Bases

Only three bases currently are in wide use: base 10 (decimal), base 2 (binary), and base 16 (hexadecimal). Occasionally base 8 (octal) is used, but that is increasingly rare. Other conversions are useful for practice and for seeing some structure in numbers. The structure will be useful for computing.

Before conversions, we need the digits to use. In base b, numbers are expressed using digits from 0 to $b-1$. When b is past 10, we need to go beyond decimal numerals:

Value:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Digit:	0	1	2	3	4	5	6	7	8	9	**A**	**B**	**C**	**D**	**E**	**F**

Upper- and lower-case are common.

So in hexadecimal, DECAFBAD is a perfectly good number, as is DEADBEEF. If there is a question of what base is being used, the base is denoted by a subscript. So 10_{10} is a decimal ten and 10_2 is in binary.

To find values we recognize more easily, we convert to decimal. Then we will convert *from* decimal.

2.1 Converting to Decimal

Converting to decimal using decimal arithmetic is straight-forward. Remember the expansion of 1234 with base $b = 10$,

$$\begin{aligned} 1234 &= 1 \cdot 10^3 + 2 \cdot 10^2 + 3 \cdot 10^1 + 4 \cdot 10^0 \\ &= b^3 + 2 \cdot b^2 + 3 \cdot b^1 + 4. \end{aligned}$$

Each digit of DEAD has a value, and these values become the coefficients. Then we expand the polynomial with $b = 16$. In a straight-forwart way,

$$\begin{aligned} \mathsf{DEAD} &= \mathsf{D} \cdot 16^3 + \mathsf{E} \cdot 16^2 + \mathsf{A} \cdot 16^1 + \mathsf{D} \\ &= 13 \cdot 16^3 + 14 \cdot 16^2 + 10 \cdot 16 + 13 \\ &= 13 \cdot 4096 + 14 \cdot 256 + 10 \cdot 16 + 13 \\ &= 57005. \end{aligned}$$

We an use **Horner's rule** to expand the polynomial in a method that often is faster,

$$\begin{aligned} \mathsf{DEAD} &= ((13 \cdot 16 + 14) \cdot 16 + 10) \cdot 16 + 13 \\ &= (222 \cdot 16 + 10) \cdot 16 + 13 \\ &= 3562 \cdot 16 + 13 \\ &= 57005. \end{aligned}$$

Let's try a binary example. Convert 1101_2 to decimal:

$$\begin{aligned} 1101_2 &= (((1 \cdot 2 + 1) \cdot 2 + 0) \cdot 2 + 1 \\ &= (3 \cdot 2 + 0) \cdot 2 + 1 \\ &= 6 \cdot 2 + 1 \\ &= 13. \end{aligned}$$

Remember the rows of a truth table for two variables? Here,

$$\begin{aligned} 11_2 &= 2 + 1 = 3, \\ 10_2 &= 2 + 0 = 2, \\ 01_2 &= 0 + 1 = 1, \text{ and} \\ 00_2 &= 0 + 0 = 0. \end{aligned}$$

2.2 Converting from Decimal

To convert to binary from decimal, consider the previous example:

$$\begin{aligned} 13 &= 8 + \mathbf{5} \\ &= 8 + 4 + \mathbf{1} \\ &= 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 \\ &= 1101_2. \end{aligned}$$

At each step, we find the largest power of two less than the remaining number. Another example for binary:

$$\begin{aligned} 293 &= 256 + \mathbf{37} \\ &= 256 + 32 + \mathbf{5} \\ &= 256 + 32 + 4 + 1 \\ &= 1 \cdot 2^8 + 1 \cdot 2^5 + 1 \cdot 2^2 + 1 \\ &= 100100101_2. \end{aligned}$$

And in hexadecimal,

$$\begin{aligned} 293 &= 256 + \mathbf{37} \\ &= 1 \cdot 256 + 2 \cdot 16 + 5 \\ &= 125_{16}. \end{aligned}$$

You can see why some people start remembering powers of two.

If you have no idea where to start converting, remember the relations $b^{\log_b x} = x$ and $\log_b x = \log x / \log b$. Rounding $\log_b x$ up to the larger whole number gives you the number of base b digits in x.

The text has another version using remainders. We will return to that in the next chapter. And conversions to and from binary will be useful when we discuss how computers manipulate numbers.

3 Operating on Numbers

Once we split a number into digits (decimal or binary), operations can be a bit easier.

We will cover multiplication, addition, and subtraction both

- to gain familiarity with positional notation, and
- to compute results more quickly and mentally.

Properties of positional notation will help when we explore number theory.

We will use two properties frequently:

- Both multiplication and addition **commute** ($a + b = b + a$) and re-**associate** $(a + b) + c = a + (b + c)$.
- Multiplication **distributes** over addition, so $a(b + c) = ab + ac$.
- Multiplying powers of a common base adds exponents, so $b^a \cdot b^c = b^{a+c}$.

3.1 Multiplication

Consider multiplication. I once had to learn multiplication tables for 10, 11, and 12, but these are completely pointless.

Any decimal number multiplied by 10 is simply shifted over by one digit,

$$\begin{aligned} 123 \cdot 10 &= (1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0) \cdot 10 \\ &= 1 \cdot 10^3 + 2 \cdot 10^2 + 3 \cdot 10^1 \\ &= 1230. \end{aligned}$$

Multiplying by $11 = 1 \cdot 10 + 1$ is best accomplished by adding the other number to itself shifted,

$$123 \cdot 11 = 123 \cdot (10 + 1) = 1230 + 123 = 1353.$$

And for $12 = 1 \cdot 10 + 2$, you double the number,

$$123 \cdot 12 = 123 \cdot (10 + 2) = 1230 + 246 = 1476.$$

Multiplying longer numbers quickly follows the same pattern of shifting and adding. We can expand $123 \cdot 123 = 123 \cdot (1 \cdot 10^2 + 2 \cdot 10 + 3)$ to

$$\begin{array}{r} 123 \\ \times\ 123 \\ \hline 369 \\ 2460 \\ 12300 \\ \hline 15129 \end{array}$$

Another method expands the product of numbers as a product of polynomials, working one term at a time. This is essentially the same but not in tabular form:

$$\begin{aligned} 123 \cdot 123 &= (1 \cdot 10^2 + 2 \cdot 10 + 3) \cdot (1 \cdot 10^2 + 2 \cdot 10 + 3) \\ &= (1 \cdot 10^2 + 2 \cdot 10 + 3) \cdot (1 \cdot 10^2 + 2 \cdot 10) + (1 \cdot 10^2 + 2 \cdot 10 + 3) \cdot 3 \\ &= (1 \cdot 10^2 + 2 \cdot 10 + 3) \cdot (1 \cdot 10^2 + 2 \cdot 10) + (3 \cdot 10^2 + 6 \cdot 10 + 9) \\ &= \ldots \end{aligned}$$

This form splits the sums apart as well; we will cover that next.

Bear in mind that short-term memory is limited to seven to eight pieces of information. Structure mental arithmetic to keep as few pieces in flight as possible. One method is to break multiplication into stages. In long form, you can group the additions. For example, expanding $123 \cdot 123 = 123 \cdot (1 \cdot 10^2) + (123 \cdot 23) = 123 \cdot (1 \cdot 10^2) + (123 \cdot 2 \cdot 10 + 123 \cdot 3)$,

$$\begin{array}{r} 123 \\ \times\ 123 \\ \hline 369 \\ 2460 \\ \hline 2829 \\ 12300 \\ \hline 15129 \end{array}$$

Assuming a small number uses only one slot in your short-term memory, need track only where you are in the multiplier, the current sum, the current product, and the next sum. That leaves three to four pieces of information to use while adding.

One handy trick for 15% tips: divide by ten, divide that amount by two, and add the pieces. We can use positional notation to demonstrate how that works,

$$\begin{aligned} x \cdot 15\% &= (x \cdot 15)/100 \\ &= ((x \cdot (10 + 5))/100 \\ &= ((x \cdot 10) + (x \cdot (10/2)))/100 \\ &= x/10 + (x/10)/2 \end{aligned}$$

3.2 Addition

Digit-by-digit addition uses the commutative and associative properties:

$$\begin{aligned} 123 + 456 &= (1 \cdot 10^2 + 2 \cdot 10 + 3) + (4 \cdot 10^2 + 5 \cdot 10 + 6) \\ &= (1 + 4) \cdot 10^2 + (2 + 5) \cdot 10 + (3 + 6) \\ &= 579. \end{aligned}$$

Naturally, when a digit threatens to roll over ten, it **carries** to the next digit. Expanding the positional notation,

$$\begin{aligned} 123 + 987 &= (1 \cdot 10^2 + 2 \cdot 10 + 3) + (9 \cdot 10^2 + 8 \cdot 10 + 7) \\ &= (1 + 9) \cdot 10^2 + (2 + 8) \cdot 10 + (3 + 7) \\ &= 10 \cdot 10^2 + 10 \cdot 10 + 10. \end{aligned}$$

Because the coefficients are greater than $b - 1 = 9$, we expand those coefficients. Commuting and reassociating,

$$\begin{aligned} 123 + 987 &= 10 \cdot 10^2 + 10 \cdot 10 + 10 \\ &= (1 \cdot 10 + 0) \cdot 10^2 + (1 \cdot 10 + 0) \cdot 10 + (1 \cdot 10 + 0) \\ &= 1 \cdot 10^3 + 1 \cdot 10^2 + 1 \cdot 10 + 0 \\ &= 1110. \end{aligned}$$

However, when working quickly, or when the addition will be used in another operation, you do not need to expand the carries immediately. This is called a **redundant representation** because numbers now have multiple representations. You can represent 13 as $1 \cdot 10 + 3$ or simply as 13.

If you work that way mentally, you need to keep the intermediate results in memory. So during multiplying, you only need to work out the carries every three to four digits...

3.3 Subtraction

In systems with signed numbers, we know that subtracting a number is the same as adding its negation: $a - b = a + (-b)$. So we expect the digit-by-digit method to work with each digit subtracted, and it does. Because $-a = -1 \cdot a$, we can distribute the sign over the digits:

$$\begin{aligned} 456 - 123 &= (4 \cdot 10^2 + 5 \cdot 10 + 6) - (1 \cdot 10^2 + 2 \cdot 10 + 3) \\ &= (4 \cdot 10^2 + 5 \cdot 10 + 6) + (-(1 \cdot 10^2 + 2 \cdot 10 + 3)) \\ &= (4 \cdot 10^2 + 5 \cdot 10 + 6) + (-1 \cdot 10^2 + -2 \cdot 10 + -3) \\ &= (4 - 1) \cdot 10^2 + (5 - 2) \cdot 10 + (6 - 3) \\ &= 333. \end{aligned}$$

As with carrying, **borrowing** occurs when a digit goes negative:

$$\begin{aligned} 30 - 11 &= (3 \cdot 10^1 + 0) - (1 \cdot 10^1 + 1) \\ &= (3 - 1) \cdot 10^1 + (0 - 1) \\ &= 2 \cdot 10^1 + -1 \\ &= 1 \cdot 10^1 + (10 - 1) \\ &= 1 \cdot 10^1 + 9 \\ &= 19. \end{aligned}$$

Again, you can use a redundant intermediate representation of $2 \cdot 10^1 - 1$ if you're continuing to other operations. And if **all** the digits are negative, you can factor out -1,

$$\begin{aligned} 123 - 456 &= (1 \cdot 10^2 + 2 \cdot 10 + 3) - (4 \cdot 10^2 + 5 \cdot 10 + 6) \\ &= (1 - 4) \cdot 10^2 + (2 - 5) \cdot 10 + (3 - 6) \\ &= (-3) \cdot 10^2 + (-3) \cdot 10 + (-3) \\ &= -(3 \cdot 10^2 + 3 \cdot 10 + 3) \\ &= -333. \end{aligned}$$

3.4 Division and Square Root: Later

We will cover these later with number theory.

4 Computing with Circuits

No one can argue that computing devices (computers, calculators, medical monitors, *etc.*) are irrelevant to everyday life. Here we lay the groundwork for how computers compute.

Essentially, computers perform arithmetic on binary numbers. But different methods of combining the arithmetic operations produce character strings, sounds, graphics, ...

While those are courses in themselves, we at least can explain the very lowest levels of computer arithmetic. Automated computing is in its relative infancy. People have been building roads, bridges, and vehicles for thousands of years. Even motors are hundreds of years old. But modern computing is less than a hundred years old and became wide-spread only 30 years ago. Before the 1970s, desktop *calculators* were rare. And before the 1980s, calculators were virtually unaffordable.

Maybe someday we will be able to take safe computing for granted just like we take safe bridges for granted, but not yet. It's important at least to have heard how computing works so you can gain a sense of where limitations are. Consider an issue like the largest range of numbers you can represent exactly in a calculator, spreadsheet, or other program. Each may have different limitations that appear random but certainly are not. Having some sense of how computers compute lets you explain or (hopefully) anticipate limitations and work around them.

4.1 Representing Signed Binary Integers

Converting non-negative numbers to binary is straight-forward. Computer representations work with a limitied number of **b**inary dig**it**s, or **bit**s. With 32 bits, any non-negative *integer* less than $2^{33} = 8589934592 \approx 10^{9.9}$ can be represented exactly. With n bits, all non-negative integers less than 2^{n+1} can be represented exactly. For example, the largest two bit number is $11_2 = 3 < 2^2 = 4$.

Representing both positive and *negative* numbers, however, presents some design choices. One can use one of the bits (often the leading bit) as a sign bit. The number then becomes $-1^{\text{sign bit}} \cdot$ the rest of the bits. This reduces the representable range of n bits to $(-(2^n), 2^n)$ and requires treating one bit specially during operations. (The notation (a, b) is an **open range**, one that does not include its endpoints.) We need separate operations for $a + b$ and $a - b$. Also, we need to cope with $+0$ and -0.

We can eliminate the need for separate operations and also eliminate the signed zero.

A representation named **one's complement** plays a little trick with arithmetic to absorb the sign into the number. This allows using addition for subtraction...

We start by **negating** a number if it is negative:

#	Bits
3	011
2	010
1	001
0	000
-0	111
-1	110
-2	101
-3	100

Adding two n-digit numbers may produce an $n + 1$-digit result. For example, $11_2 + 11_2 = 110_2$ in binary or $5 + 6 = 11$ in decimal. Consider three bit addition:

$$\begin{array}{rr} & 110 \\ + & 10 \\ \hline & \mathbf{1}000 \end{array}$$

If we capture the carry bit **1** and feed it back around, then $110_2 + 10_2 \mapsto 000_2 + 1_2 = 1_2$. In one's complement, this is $-1 + 2 = 1$ as expected.

So to add two numbers, *positive or negative*, we just add the one's complement representation. To subtract $a - b$, we negate b and add it to a. **We only need one operation, addition, for addition and subtraction.**

But we still have given an entire bit over to the sign. We can do slightly better with **two's complement.** More importantly, we can reduce the system to having only a single, unsigned zero. Having an unsigned zero is much easier to handle with multiplication and division.

To represent a negative number in two's complement, we negate it and add one:

#	Bits
3	011
2	010
1	001
0	000
-1	111
-2	110
-3	101
-4	111

By not including -0, we have room for one more number. By the two's complement method, it happens to fall on the end of the negative scale. Here, n bits represent all integers in $[-2^n, 2^n)$. (The notation $[a, b]$ is a **closed range** including a and b. Notations using square brackets on one side but not the other are half-open and include the end-point against the square bracket.)

There are other representations:

- A **biased** representation adds 2^{n-1} or $2^{n-1} - 1$ to every number and then represents the result. This shifts all the negative numbers to be non-negative. This representation has an explicit sign bit but only a single zero.
- A **base -2** rather than base 2 representation is bizarre, but it works. These most often are used for redundant representations inside other arithmetic operations. There are twice as many negative numbers as positive numbers, no sign bit, and only a single zero.
- Larger bases can be used by grouping bits. This also allows for more redundant representations. One representation using 1, 0, and -1 for digits is particularly interesting, but we won't cover it here.

4.2 Adding in Binary with Logic

Above we have reduced addition and subtraction of signed numbers into simple addition. Here we implement addition in logic and construct the **half adder** and **full adder** circuits.

Consider a truth table for $a \wedge b$ and $a \oplus b$ (*exclusive or*):

a	b	$a \wedge b$	$a \oplus b$	$a + b$
1	1	1	0	10_2
1	0	0	1	01_2
0	1	0	1	01_2
0	0	0	0	00_2

If we append a column representing the sum of a and b in binary, we see that the first digit is $a \wedge b$ and the second is $a \oplus b$!

This is a **half adder**. The half adder takes two bits as input and produces a sum bit $s = a \oplus b$ and a carry bit $c = a \wedge b$.

(drawing)

A **full adder** takes input bits and a previous carry bit to produce an output sum and carry. We can add $a + b$ and then $(a + b) + c_{\text{in}}$. Note that only one of those sums can generate a carry, so *or*-ing the carry outputs generates the final output. $1 + 1 + 1 = 3 = 11_2 < 100_2$, so the sum's output cannot require more than two bits.

So a full adder can be constructed with two half-adders and one extra or-gate for the carry:

a	b	c_{in}	$(a \oplus b) \oplus c_{\text{in}}$	$(a \wedge b) \vee (c_{\text{in}} \wedge (a \oplus b))$
1	1	1	1	1
1	1	0	1	0
1	0	1	1	0
1	0	0	0	1
0	1	1	1	0
0	1	0	0	1
0	0	1	0	1
0	0	0	0	0

To add two n bit numbers, you start by adding the low-order bits (coefficient in front of 2^0) with a half-adder. The sum is output and the carry follows into a full adder for adding the coefficients of 2^1. The process continues resulting in an n-bit sum and a single carry bit.

The carry bit often is ignored, leading to **overflow** and **wrap-around**. At a low level, adding two positive integers each greater than 2^{n-1} produces a *negative number*! This is terribly handy for some algorithms and detrimental to others.

All architectures make the carry bit available for diagnosing overflow, but not all programming environments let users access that information.

Adding two n-bit numbers requires a minimum of one half-adder and $n-1$ full adders, or $2n-1$ half-adders and $n-1$ *or* gates, or $2n-1$ *exclusive-or* gates, $2n-1$ *and* gates, and $n-1$ *or* gates. Because of the dependence on the previous bit sum's carry output, it appears that each bit must be computed one at a time, or **serially**. There are tricks using redundant representations that allow computing the result in larger chunks, exposing more **parallelism** within the logic gates.

4.3 Building from Adders

Given addition, we could implement multiplication as *repeated addition*. Remember the Egyptian algorithm from the text?

The binary representation of the multiplier serves as a **mask**. Consider multiplying the 3-bit numbers $011_2 = 3$ and $101_2 = 5$:

011	·1
011	·0
011	·1
01111	

At each step, the bits of 101_2 determine whether or not a shifted copy of 011_2 is added into the result. We can implement this by shifting and adding serially, or we can construct a **multiplier array** out of adders.

Again, there are optimizations related to redundant representations, but ultimately most processors dedicate a large amount of their physical size (and "power budget") to multiplier arrays.

The problem of overflow becomes very important for multiplication. Because $2^n \cdot 2^n = 2^{2n}$, the product of two n-bit numbers may require $2n$ bits. Most architectures deliver the result in two n-bit **registers** (the limited number of variables a processor has to work with).

4.4 Decimal Arithmetic from Binary Adders

Ok, so we can add, subtract, and multiply numbers in binary. What about decimal? Alas, we lack the nifty two's complement tricks in decimal, so all decimal units need to cope with signs differently. Most use explicit signs and always convert -0 to 0.

For integers, conversion back and forth can occur exactly as in class. With 32 bits, there are at most $\lceil 32 \cdot \log_{10} 2 \rceil = 10$ decimal digits. (The notation $\lceil x \rceil$

rounds x to the closest integer $k > x$.) So software can lop off digits one at a time, often using the text's algorithm with remainders.

There are times when you want to work directly with decimal numbers, however. Some of these are dictated by legal or engineering considerations. For example, the "cpu" of a hand-held calculator is does not really run software or store many intermediate results. There, every result is calculated in decimal often using a representation called **BCD** for **b**inary **c**oded **d**ecimal.

A decimal number is represented digit-by-digit in binary. So $29 = 2 \cdot 10 + 9 = (10_2) \cdot 10 + (1001_2)$. This is relatively inefficient. The largest two digits 8 and 9 both require four bits, but the rest require only three. So six binary strings are not used and cannot represent digits. For example, $1010_2 = 10 > 9$, so 1010_2 will never appear in a correct BCD digit encoding. In BCD, results mostly are computed digit-by-digit in binary and then manipulated into a correct BCD encoding.

Using four bits per digit has one major advantage; each decimal digit is a hexadecimal digit. So the hex number 1594_{16} is interpreted as the decimal 1594. This also allows a nifty trick for adding two BCD-encoded numbers.

Say we want to add $a = 1103$ and $b = 328$. In decimal, $a + b = 1431$. If we were to add these directly in hexadecimal, $a + b = 142\mathsf{B}_{16}$. There needs to be some mechanism for carrying. We can use the six missing code points to force a carry into the next digit, and then we can compare with the *exclusive-or* to detect where carries actually happened.

The procedure starts by adding 6 to each BCD digit as if they were hexadecimal. So we shift each digit of a to the top of its hex range and use $a + 6666_{16} = 7769_{16}$. Now we compute a sum $s_1 = (a + 6666_{16}) + b = 7\mathsf{A}91_{16}$. This isn't the final sum; if we subtract 6 from every digit, $7\mathsf{A}91_{16} - 6666_{16} = 142\mathsf{B}_{16}$, we do not obtain a BCD-encoded number.

We need to subtract 6 only from those digits that did not generate a carry, $7\mathsf{A}91_{16} - 6660_{16} = 1431_{16}$. This is a correct BCD number and the correct result. The carries can be detected by comparing $(a + 6666_{16}) + b$ with $(a + 6666_{16}) \oplus b$, the bitwise *exclusive or*. If the two results differ in the lowest bit per hex digit / BCD digit, we know there was a carry and we know where to subtract 6_{16}.

Alas, there are no particularly nice tricks for multiplication. But if most uses include adding a list of prices and applying a tax once, it's not so bad.

Another form that wastes far less space is called **millennial encoding**. Because $2^{10} = 1024 > 10^3$, ten bits can represent all three decimal digit numbers. This wastes only 25 encoding points per three decimal digits, as opposed to wasting six points every for every single decimal digit. Arithmetic operates in binary on the chunks of ten bits and then manipulates the results.

And there are more encodings, including Tien Chi Chen and Dr. Irving T. Ho's **Chen-Ho** encoding (1975), Mike Cowlishaw's **DPD** encoding (**d**ensely **p**acked

decimal, 2002), and Intel's **BID** encoding (**b**inary **i**nteger **d**ecimal). These require more complicated coding techniques to explain, but the latter two (DPD and BID) are now (as of August, 2008) international standards.

5 Homework

Practice is absolutely critical in this class.

Groups are fine, turn in your own work. Homework is due in or before class on Mondays.

- Section 4.1:
  - Problems 35, 36 (the algorithm is in the text, see Section 4.1, Example 4)
- Section 4.2:
  - Problems 2, 3, 5, 6, 11, 12
- Section 4.3:
  - Problem 2, 7, 8
  - Problems 19-22 (the "calculator shortcut" is Horner's rule)
  - Problems 37-40
  - Problem 57 (he played at the festival)
- Expressing numbers in positional form:
  - Take a familiar incomplete integer, _679_, and express it as a sum of the digits times powers of ten using variables x_0 and x_4 for the digits in the blanks. Simplify to the form of $x_4 \cdot 10^4 + x_0 \cdot 10^0 + z$, where z is a single number in positional form (a sequence of digits). Does 72 divide z? Does 8 divide z? Does 9 divide z? Remember that $72 = 8 \cdot 9$. We will use this example again in the next chapter.
- Operations;
  - Multiply 47 by each of 3, 13, and 23. Show your work, and work digit-by-digit. Use either the expanded form (expanding $(4{\cdot}10{+}7){\cdot}(2{\cdot}10{+}3)$ or the tabular form collapsing the sum every two steps.
  - Add 47 to each of 52, 53, and 54. Show your work, and work digit-by-digit. Show an intermediate redundant representation if there is one.
  - Subtract 19 from each of 7, 19 (not a typo), 20, and 29. Show your work, and work digit-by-digit. Show an intermediate redundant representation if there is one.

Note that you *may* email homework. However, I don't use Microsoft™ products (*e.g.* Word), and software packages are notoriously finicky about translating mathematics.

If you're typing it (which I advise just for practice in whatever tools you use), you likely want to turn in a printout. If you do want to email your submission, please produce a PDF or PostScript document.