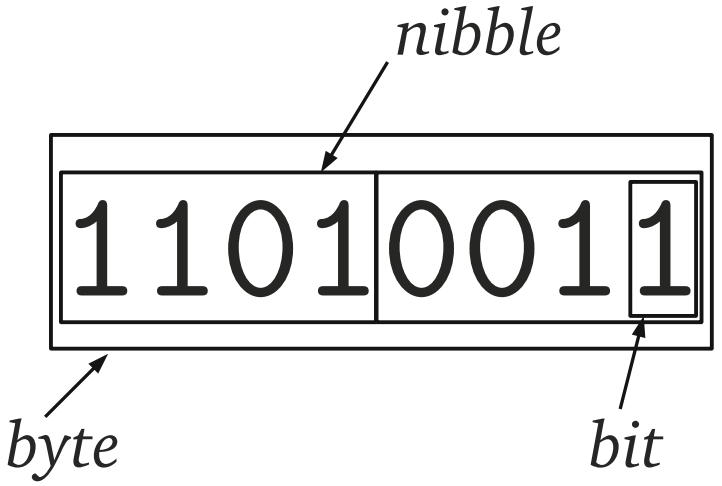

Fig. 2.1 A byte which consists of two nibbles and eight bits. The number is `211` which is made up of two nibbles, $\texttt{D}_{16}$ and $\texttt{3}_{16}$ giving $\texttt{D3}_{16}$, or eight bits $\texttt{11010011}_2$

Historically, the word size of a computer could be anywhere from four bits for early microprocessors (Intel 4004) to 60 bits for early mainframes (CDC 6600) but modern computers have settled on powers of two for their word sizes, typically 32 or 64 bits with some second generation microprocessors using 16 bit words (Intel 8086, WDC 65816). For our purposes, we can assume a 32 bit word size when describing data formats. On occasion we will need to be more careful and explicitly declare the word size we are working with.

With these preliminaries now under our belt, it is time to start working with actual data representations. We start naturally with unsigned integers.

2.2 Unsigned Integers

Unsigned integers are our first foray into the depths of the computer as far as numbers are concerned. These are the basic numbers, the positive integers, that find frequent use in representing characters, as counters, or as constants, really anything that can be mapped to the set $\{0,1,2,\ldots\}$. Of course, computers have finite memory so there is a limit to the size of an unsigned integer and we will get to that in the next section.

2.2.1 Representation

Integers are stored in memory in binary using one or more bytes depending on the range of the integer. The example in Fig. 2.1 is a one byte unsigned integer. If we are working in a typed language like C we need to declare a variable that will store one byte. Typically, this would mean using an *unsigned char* data type,

```
unsigned char myByte;
```

which can store a positive integer from $00000000_2 = 0$ to $11111111_2 = 255$. This is therefore the *range* of the `unsigned char` data type. Note how C betrays its age by referring to a byte number as a character. Unsigned integers larger than 255 require more than one byte.

Table 2.1 shows standard C types for unsigned integers and the allowed range for that type. These are fixed in the sense that numbers must fit into this many bits at all times. If not, an underflow or overflow will occur. Languages like Python abstract integers for the programmer and only support the concept of integer as opposed to a floating-point number. The unsigned integer operations discussed later in the chapter work nicely with Python but the concept of range is a little nebulous in that case since Python will move between internal representations as necessary.

Declaration	Minimum	Maximum	Number of bits
`unsigned char`	0	255	8
`unsigned short`	0	65,535	16
`unsigned int`	0	4,294,967,295	32
`unsigned long`	0	4,294,967,295	32
`unsigned long long`	0	18,446,744,073,709,551,615	64

Table 2.1 Unsigned integer declarations in C. The declaration, minimum value, maximum value and number of data bits are given

If a number like $11111111_2 = 255$ is the largest unsigned number that fits in a single byte how many bytes will be needed to store 256? If we move to the next number we see that we will need *nine* bits to store 256 which implies that we will need two bytes. However, there is a subtlety here that needs to be addressed. This is, if the number is to be stored in the computer's memory, say starting at address 1200, how should the individual bits be written to memory location 1200 and the one following (1201, since each memory location is a byte)? This question has more than one answer.

2.2.2 Storage in Memory: Endianness

Addresses. In order to talk about how we store unsigned integers in computer memory we have to first talk a bit about how memory is addressed. In this book,

we have a working assumption of a 32-bit Intel-based computer running the Linux operating system. Additionally, we assume `gcc` to be our C compiler. In this case, we have *byte-addressable* memory meaning that even though the word size is four bytes, we can refer to memory addresses using bytes. For example, this small C program,

```
 1 #include <stdio.h>
 2
 3 int main() {
 4     unsigned char *p;
 5     unsigned short n = 256;
 6
 7     p = (unsigned char *)&n;
 8     printf("address of first byte  = %p\n", &p[0]);
 9     printf("address of second byte = %p\n", &p[1]);
10
11     return 0;
12 }
```

when compiled and run produces output similar to,

```
address of first byte  = 0xbfdafe9e
address of second byte = 0xbfdafe9f
```

where the specific addresses, given in hexadecimal and starting with the `0x` prefix used by C, will vary from system to system and run to run. The key point is that the difference between the addresses is *one* byte. This is what is meant by a byte-addressable memory. If you don't know the particulars of the C language, don't worry. The program is defining a two byte number in line 5 (n) and a pointer to a single byte number in line 4 (p). The pointer stores a memory address which we set to the beginning of the memory used by n in line 7. We then ask the computer to print the numeric address of the memory location (line 8) and of the next byte (line 9) using the `&` operator and indexing for the first (`p[0]`) and second (`p[1]`) bytes. With this in mind, let's look at actually storing unsigned integers.

Bit Order. To store the number $11011101_2 = 221$ in memory we use the eight bits of the byte at a particular memory address. The question then becomes, which bits of 221 map to which bits of the memory? If we number the bits from 0 for the lowest-order bit, which is the right-most bit when writing the number in binary, to 7 for the highest-order bit, which is the left-most bit when writing in binary, we get a one-to-one mapping,

7	6	5	4	3	2	1	0
1	1	0	1	1	1	0	1

so that the highest-order bit of the number is put into the highest-order bit of the memory address. This seems sensible enough but one could imagine doing the reverse as well,

7	6	5	4	3	2	1	0
1	0	1	1	1	0	1	1

So, which is correct? The answer depends on how bits are read from the memory location. In modern computer systems, the bits are read low to high meaning the low-order bit is read first (bit 0) followed by the next higher bit (bit 1) so that the computer will use the first ordering of bits that maps 76543210 to 11011101.

Byte Order. Within a byte, bits are stored low to high. What about the bytes of a multibyte integer? We know that the number $100000000_2 = 256$ requires two bytes of memory, one storing the low set of eight bits 00000000_2 and another storing the high set of eight bits 00000001_2 where leading zeros are used to indicate that it is the first bit position set and the rest of the bits in the byte are clear (set to zero). But, what order in memory should we use,

Memory address:	1200	1201
Low first:	00000000	00000001
High first:	00000001	00000000

low first or high first? The answer is either. If we put the low order bytes of a multibyte number in memory first we are using *little-endian* storage. If we put the high order bytes first we are using *big-endian* or *network order*. The choice is the *endianness* of the number and both are in use typically as a function of the microprocessor, which has a preferred ordering. The terms little-endian and big-endian are derived from the book *Gulliver's Travels* by satirist Jonathan Swift [4]. Published in 1726 the book tells the story of Gulliver and his travels throughout a fictional world. When Gulliver comes to the land of Lilliput he encounters two religious sects who are at odds over whether to crack open their soft-boiled eggs little end first or big end first. Intel based computers use little-endian when storing multibyte integers. Motorola based computers use big-endian. Additionally, big-endian is called network order because the Internet Protocol standard [5] requires that numeric values in packet headers be stored in big-endian order.

Let's look at some examples of little and big endian numbers. To save space, we will use a 32-bit number but represent the numbers showing only the hexadecimal values of the four bytes such a number would occupy. For example, if the 32-bit number is,

$$10101110111100000101011010010110_2 = \mathtt{AE}_{16}\ \mathtt{F0}_{16}\ \mathtt{56}_{16}\ \mathtt{96}_{16}$$

we will write it as AEF05696 dropping the 16 base indicator for the time being. Putting this number into memory starting with byte address zero and using big-ending ordering gives,

Address:	0	1	2	3
Value:	AE	F0	56	96

where the first byte one would read, pulling bytes from address zero first, is the high-order byte. If we want to use little-endian we would instead put the lowest order byte first to fill in memory with,

Address:	0	1	2	3
Value:	96	56	F0	AE

which will give us the low-order byte first when reading from address zero. Note, within a particular byte we always still use the bit ordering that maps the high-order bit (bit 7) to the high-order bit of the number. Reversing the bits within a byte when using little-endian is an error.

When we are working with data on a single computer within a single program, we typically do not need to pay attention to endianness. However, when transmitting data over the network or using data files written on other machines it is a good idea to consider what the endianness of the data is. For example, when reading sensor values from a CCD camera, which are typically 16-bit numbers requiring two bytes to store, it is essential to know the endianness otherwise the image generated from the raw data will look strange because the high-order bits which contain the majority of the information about the image will be in the wrong place. From the examples above, we see that correcting for endianness differences is straightforward, simply flip the order of the bytes in memory to convert from little-endian to big-endian and vice versa.

2.3 Operations on Unsigned Integers

Binary and unary operations on unsigned integers is the heart of what computers do. In this section we go over all the common operations, bit by bit, and mention some of the key things to look out for when working with unsigned integers of fixed bit width. Examples will use byte values to keep it simple but in some cases multibyte numbers will be shown. Specific operations will be shown in C and Python. Though the Python syntax is often the same as C, the results may differ because the Python interpreter will change the internal representation as necessary.

2.3.1 Bitwise Logical Operations

Bitwise operators are binary operators, meaning they operate on two numbers (called operands), to produce a new binary number. In the previous sentence the first instance of "binary" refers to the number of operands or arguments to the operation while the second instance of "binary" refers to the base of the operands themselves. This is an unfortunate but common abuse of notation.

Digital logic circuits, which are fascinating to study but well beyond the purview of this book, are built from logic gates which implement in hardware the logical operations we are discussing here. The basic set of logic operators are given names which relate to Boolean logic: `AND`, `OR`, `XOR`, and the unary `NOT`. Negated versions of `AND` and `OR`, called `NAND` and `NOR`, are also in use but these are easily constructed by taking the output of `AND` and `OR` and passing it through a `NOT` so we will ignore them here.

Logical operations are most easily understood by looking at their *truth tables* which illustrate the function of the operator by enumerating the possible set of inputs and giving the corresponding output. They are called truth tables because of the correspondence between the `1` and `0` of a bit and the logical concept of `true` and `false` which goes back to Boolean algebra developed by George Boole in the nineteenth century [6].

A truth table shows, for each row, the explicit inputs to the operator followed by the output for those inputs. For example, the truth table for the `AND` operator is,

AND	
0 0	0
0 1	0
1 0	0
1 1	1

which says that if the two inputs to the `AND`, recall the inputs are single bits, are `0` and `0` that the output bit will also be `0`. Likewise, if the inputs are `1` and `1` the output will also be `1`. In fact, for the `AND` operation the only way to get an output of `1` is for both input bits to be set. We can think of this as meaning "only both".

The two other most commonly used binary logic operators are `OR` and `XOR`. The latter is a called exclusive-`OR` and is sometimes indicated with `EOR` instead of `XOR`. Their truth tables are,

OR	
0 0	0
0 1	1
1 0	1
1 1	1

XOR	
0 0	0
0 1	1
1 0	1
1 1	0

where we see that `OR` means "at least one" and `XOR` means "one or the other but not both". The last of the common bitwise logical operators is `NOT` and it has a very simple truth table,

NOT	
0	1
1	0

where the operation is simply turn `1` to `0` and `0` to `1`.

If we look again at the truth tables for `AND`, `OR` and `XOR` we see that there are four rows in each table which match the four possible sets of inputs. Looking at these rows we can interpret the four bits in the output as a single four bit number. A four bit number has sixteen possible values from 0000_2 to 1111_2 which implies that there are sixteen possible truth tables for a binary bitwise operator. Three of them are `AND`, `OR` and `XOR` and another three are the tables made by negating the output of these operators. That leaves ten other possible truth tables. What do these correspond to and are they useful in computer programming? The full set of possible binary truth tables is given in Table 2.2 along with an interpretation of each operation. While all of these operators have a name and use in logic many are clearly of limited utility in

terms of computer programming. For example, always outputting `0` or `1` regardless of the input (`FALSE` and `TRUE` in the table) is of no practical use in helping the programmer. The negated versions of the three main operators, `NAND`, `NOR` and `XNOR`, are naturally built by adding a "not" word to the positive versions and are therefore useful.

p:	0	0	1	1	
q:	0	1	0	1	
FALSE	0	0	0	0	always false
AND	0	0	0	1	p and q
	0	0	1	0	if p then not q else false
	0	0	1	1	p
	0	1	0	0	if not p and q
	0	1	0	1	q
XOR	0	1	1	0	p or q but not (p and q)
OR	0	1	1	1	p, q or (p and q)
NOR	1	0	0	0	not OR
XNOR	1	0	0	1	not XOR
	1	0	1	0	not q
	1	0	1	1	if q then p else true
	1	1	0	0	not p
	1	1	0	1	if p then q else true
NAND	1	1	1	0	not AND
TRUE	1	1	1	1	always true

Table 2.2 The sixteen possible binary bitwise logical operators. If a common name exists for the operation it is given in the first column. The top two rows labeled p and q refer to the operands or inputs to the operator. A `0` is considered false while `1` is true. A description of the operator is given in the last column. In the description the appearance of p or q in an if-statement implies that the operand is true

Now that we know the basic operators and why this set is most useful to computer programmers let's look at a few examples. First the `AND` operator,

```
    10111101 = 189
AND 11011110 = 222
-------------------
    10011100 = 156
```

where the binary representation of the number is in the first column and the decimal is in the second. We see that for each bit in the two operands, `189` and `222`, the output bit is on only if the corresponding bits in the operand were on. If the operands were simply zero or one the output would be one only if the inputs were both one. Since this is exactly what we mean when we use "and" in a sentence we can use this operator to decide if two things are both true. A natural place to use this ability is in an if-statement and many programming languages, C and Python included, use the concept that zero is false and one is true. Actually, C and Python extend this concept and declare anything that is not zero to be true. It should be noted that Python also supports Boolean variables and treats the keywords `True` and `False` as true and false respectively.

We look next at the `OR` and `XOR` operators. For `OR` we have,

```
    10111101 = 189
OR  11011110 = 222
-------------------
    11111111 = 255
```

which means at in every bit position at least one of the operands had a bit set. The `XOR` operator will give us,

```
     10111101 = 189
XOR  11011110 = 222
-------------------
     01100011 =  99
```

One useful property of the `XOR` operator is that it undoes itself if applied a second time. Above, we calculated `189 XOR 222` to get `99`. If we now perform `99 XOR 222` we get,

```
     01100011 =  99
XOR  11011110 = 222
-------------------
     10111101 = 189
```

giving us back what we started with. As an aside, this is useful as a simple way to encrypt data. If Alice wishes to encrypt a stream of n bytes (M), which may represent any data at all, so that she can send it (relatively) securely to Bob, all she need do is generate a stream of n random bytes (S) and apply `XOR`, byte by byte, to the original stream M to produce a new stream, M'. Alice can then transmit M' to Bob any way she wishes knowing that if Carol intercepts the message she cannot easily decode it without the same random stream S. Of course, Bob must somehow have S as well in order to recover the original message M. If S is truly random, used only once and then discarded, this is known as a *one-time pad* encryption. The keys are that S is truly random and that it is only used once. Of course, this does not cover the possibility that Carol may get her hands on S as well, but if she cannot, there is (relatively) little chance she will be able to recover M from just M'.

Another handy use for `XOR` is that it allows us to swap two unsigned integers without using a third variable. So, in C, instead of,

```
unsigned char a,b,t;
...
t = a;
a = b;
b = t;
```

we can use,

```
unsigned char a,b;
...
a ^= b;  // a = a ^ b;
b ^= a;
a ^= b;
```

where ^ is the C operator for XOR and `a ^= b` is shorthand for `a = a ^ b`. Why does this work? If we write out an example in binary, we will see,

```
a = 01011101
b = 11011011
-------------------------------------------
a ^= b      | 01011101 ^ 11011011 → a=10000110
b ^= a      | 10000110 ^ 11011011 → b=01011101
a ^= b      | 10000110 ^ 01011101 → a=11011011
```

As mentioned above, if we do a XOR $b \rightarrow c$ and then c XOR b we will get a back. If we instead do c XOR a we will get b back. So, the trick makes use of this by first doing a XOR b and then using that result, stored temporarily in a, to get a back but this time storing it in b and likewise for getting b back and storing it in a. This, then, swaps the two values in memory. This trick works for unsigned integers of any size. Another common use of XOR is in parity calculations. The parity of an integer is the number of 1 bits in its binary representation. If the number is odd, the parity is 1, otherwise it is 0. This can be used as a simple checksum when transmitting a series of bytes. A checksum is a indicator of the integrity of the data. If the checksum calculated on the receiving end does not match then there has been an error in transmission. If the last byte of data is the running XOR of all previous bytes the parity of this byte can be used to determine if a bit was changed during transmission. In order to see how to use XOR for parity calculations we need to wait a little bit until we talk about shifts below.

The effect of NOT is straightforward to illustrate,

```
NOT  10111101 = 189
     01000010 =  66
```

it simply flips the bits from 0 to 1 and 1 to 0. We will see this operation again when we investigate signed integers.

Figure 2.2 gives a C program that implement the AND, OR, XOR and NOT logical operators. This illustrates the syntax. Note that the C example uses the `unsigned char` data type. This is an 8-bit unsigned integer like we saw in the examples above. The operators work with any size integer type so we could just as well have used `unsigned short` or `unsigned int`, etc. Figure 2.3 gives the corresponding Python code.

The standard C language does not support direct output of numbers in binary so the example in Fig. 2.2 instead outputs in decimal and hexadecimal. Recall that in Chap. 1 we learned how to easily change hexadecimal numbers into binary by replacing each hexadecimal digit with four bits representing that digit. The Python code makes use of standard string formatting language features to output in binary directly. As Python is itself written in C and the language was designed to show that heritage, the syntax for the bitwise logical operators is the same. There is one subtlety with the Python code you may have noticed. The function `pp()` uses "`z & 0xff`" in the `format()` method call where you might have expected only "`z`". If we do not AND the output value with $\texttt{FF}_{16}$ we will output a full range integer

```
1 #include <stdio.h>
2
3 void pp(unsigned char z) {
4     printf("%3d (%02x)\n", z, z);
5 }
6
7 int main() {
8     unsigned char z;
9
10    z = 189 & 222;  pp(z); // '&' is AND
11    z = 189 | 222;  pp(z); // '|' is OR
12    z = 189 ^ 222;  pp(z); // '^' is XOR
13    z = z ^ 222;    pp(z); // returns 189
14    z = ~z;         pp(z); // '~' is NOT
15
16    return 0;
17 }
```

Fig. 2.2 Bitwise logical operators in C

```
1 def pp(z):
2     print "{0:08b}".format(z & 0xff)
3
4 def main():
5     z = 189 & 222;  pp(z) # '&' is AND
6     z = 189 | 222;  pp(z) # '|' is OR
7     z = 189 ^ 222;  pp(z) # '^' is XOR
8     z = z ^ 222;    pp(z) # returns 189
9     z = ~z;         pp(z) # '~' is NOT
10
11 main()
```

Fig. 2.3 Bitwise logical operators in Python

and as we will see later in this chapter, that would be a negative number to Python. The `AND` keeps only the lowest eight bits and gives us the output we would expect matching the C code in Fig. 2.2.

2.3.2 Testing, Setting, Clearing, and Toggling Bits

Common operations on unsigned integers include the setting, clearing and testing of particular bits. Setting a bit means to turn that bit on (`1`) while clearing is to turn the bit off (`0`). Testing a bit simply returns its current state, `0` or `1`, without changing its value. In the embedded computing world this is often done to set an output line high or low or to read the state of an input line. For example, a microcontroller uses specific pins on the device as digital inputs or outputs. Typically, this requires setting bits in a control register or reading bits in a register to know whether a voltage is present or not on pins of the device.

The bitwise logical operators introduced above are the key to working with bits. As Fig. 2.3 already alluded to, the AND operator can be used to mask bits. Masking sets certain bits to zero and some examples will make this clear. First, consider a situation similar to the Python code of Fig. 2.3 that keeps the lower nibble of a byte while it clears the upper nibble. To do this, we AND the byte with $0F_{16}$,

```
      10111101 = BD
AND   00001111 = 0F
-------------------
      00001101 = 0D
```

where we now display the binary and hexadecimal equivalent. The output byte will have all zeros in the upper nibble because AND is only true when both operands are one. Since the second operand has all zeros in the upper nibble there is no situation where the output can be one. Likewise, the lower nibble is unaffected by the AND operation since every position where the first operand has a one the second operand will also have a one leading to an output of one but every position where the first operand has a zero the second operand will still have a one leading to an output of zero, which is just what the first operand has in that position. A quick look back at the truth table for AND (above) will clarify why this masking works.

This property of AND can be used to test if a bit is set. Say we wish to see if bit 3 of BD_{16} is set. All we need to do is create a mask that has a one in bit position 3 and zeros in all other bit positions (recall that bit positions count from zero, right to left),

```
      10111101 = BD
AND   00001000 = 08
-------------------
      00001000 = 08
```

if the result of the AND is not zero we know that the bit in position 3 of BD_{16} is indeed set. If the bit was clear, the output of the entire operation would be exactly zero which would indicate that the bit was clear. Since C and Python treat nonzero as true the following C code would output "bit 3 on",

```
if (0xBD & 0x08) {
    printf("bit 3 on\n");
}
```

as would this Python code,

```
if (0xBD & 0x08):
    print "bit 3 on"
```

We use AND to test bits and mask bits but we use OR to actually set bits. For example, to actually set bit 3 and leave all other bits as they were we would do something like this,

```
      10110101 = B5
OR    00001000 = 08
-------------------
      10111101 = BD
```

which works because OR returns true when either operand or both have a one in a particular bit position. Just as AND with an operand of all ones will return the other operand, so an OR with all zeros will do likewise. In order to set bits, then, we need OR, a bit mask, and assignment as this C code shows,

```
#include <stdio.h>

int main() {
    unsigned char z = 0xB5;

    z = z | 0x08;  // set bit 3
    printf("%x\n", z);
    z |= 0x40;     // set bit 6
    printf("%x\n", z);

    return 0;
}
```

which outputs $\texttt{BD}_{16}$ and $\texttt{FD}_{16}$ after setting bits 3 and 6. Since we updated z each time the set operations are cumulative.

We've seen how to test and set bits now let's look at clearing bits without affecting other bits. We know that if we AND with a mask of all ones we will get back our first operand. So, if we AND with a mask of all ones that has a zero in the bit position we want to clear we will turn off that bit and leave all other bits as they were. For example, to turn off bit 3 we do something like this,

$$\begin{array}{rr} & \texttt{10111101 = BD} \\ \texttt{AND} & \texttt{11110111 = F7} \\ \hline & \texttt{10110101 = B5} \end{array}$$

where the mask value for AND is $\texttt{F7}_{16}$. But, this begs the question of how we get the magic $\texttt{F7}_{16}$ in the first place? This is where NOT comes into play. We know that to set bit 3 we need a mask with only bit 3 on and all others set to zero. This mask value is easy to calculate because the bit positions are simply powers of two so that the third position is $2^3 = 8$ implying that the mask is just $\texttt{8} = \texttt{00001000}_2$. If we apply NOT to this mask we will invert all the bits giving us the mask we need: $\texttt{11110111}_2$. This is readily accomplished in C,

```
unsigned char z = 0xBD;
z = z & (~0x08);
printf("%x\n", z);
```

which outputs $\texttt{B5}_{16} = \texttt{10110101}_2$ with bit 3 clear. The same syntax will also work in Python.

Our last example toggles bits. When a bit is toggled its value is changed from off to on or on to off. We can use NOT to quickly toggle all the bits, since this is what NOT does, but how to toggle only one bit and leave the rest unchanged? The answer lies in using XOR. The XOR operator returns true only when the operands are different. If both are zero or one, the output is zero. This is what we need. If we XOR our value with a mask that has a one in the bit position we desire to toggle we will get something like this for bit 1,

$$\begin{array}{rr} & 10111101 = \texttt{BD} \\ \texttt{XOR} & 00000010 = 02 \\ \hline & 10111111 = \texttt{BF} \end{array}$$

where we have turned bit 1 on when previously it was off. If bit 1 was already on we would have $1 \texttt{ XOR } 1 = 0$ which would turn it off in the output. Clearly, we can toggle multiple bits by setting the positions we want to toggle in the mask and then apply the XOR. In C we have,

```
unsigned char z = 0xBD;
z = z ^ 0x02;
printf("%x\n", z);
```

which will output $\texttt{BF}_{16} = 10111111_2$ as expected.

2.3.3 Shifts and Rotates

So far we have looked at operations that manipulate bits more or less independently of other bits. Now we take a look at sliding bits from one position to another within the same value. These manipulations are accomplished through the shift and rotate operators. A *shift* is as straightforward as it sounds, just move bits from lower positions in the value to higher, if shifting to the left, or from higher positions to lower if shifting to the right. When shifting, bits simply "fall off" the left or right if they hit the end of the integer. This implies something, namely, that we impose a specific number of bits on the integer. For our examples we will stick with 8-bit unsigned integers though all of these operations work equally well on integers of any size. Let's look at what happens when we shift a value to the left one bit position using binary notation,

$$10101111 \leftarrow 1 = 01011110$$

where we use the $\leftarrow$ symbol to mean shift to the left and the 1 is the number of bit positions. The leading 1 drops off the left end and a zero moves in from the right end to fill in the empty space. All other bits move up one bit position. Now, what happens when only a single bit is set and we shift one position to the left,

$$00000010 \leftarrow 1 = 00000100$$

we see that we started with a value of $2^1 = 2$ and we ended with a value of $2^2 = 4$. Therefore, a single position shift to the left will move each bit to the next highest bit position which is the same as multiplying it by two. Since this will happen to all bits, the net effect is to multiply the number by two. Of course, if bits that were set fall off the left end we will lose precision but the remaining bits will be multiplied by two. For example,

$$00101110 \leftarrow 1 = 01011100$$

which takes $\texttt{00101110}_2 = 46$ to $\texttt{01011100}_2 = 92$ which is 46×2. Shifting to the left by more than one bit position is equivalent to repeated shifts by one position so shifting by two positions will multiply the number by $2 \times 2 = 4$,

$$\texttt{00101110} \leftarrow 2 = \texttt{10111000}$$

giving $\texttt{10111000}_2 = 184$ as expected.

It is natural to think that if a left shift multiplies by two a right shift would divide by two and this is indeed the case,

$$\texttt{00101110} \rightarrow 1 = \texttt{00010111}$$

gives $00010111_2 = 23$ which is $46 \div 2$. Just as bits falling off the left end of the number will result in a loss of precision so will bits falling off the right end with one significant difference. If a bit falls off the right end of the number it is lost from the ones position. If the ones position is set, that means the number is odd and not evenly divisible by two. If the bit is lost the result is still the number divided by two but the division is *integer division* which ignores any remainder. Information is lost since a right shift followed by a left shift results in a number that is one less than what we started with if the original number was odd. We can see this by shifting two positions to the right,

$$\texttt{00101110} \rightarrow 2 = \texttt{00001011}$$

to get $00001011_2 = 11$ which is $46 \div 4$ ignoring the remainder of 2.

Both C and Python support shifts using the `<<` and `>>` operators for left and right shifts respectively. The `<<` operator is frequently used with an argument of `1` in order to quickly build bit masks,

$$\texttt{1 << 3} = \texttt{00001000}_2$$
$$\texttt{1 << 5} = \texttt{00100000}_2$$

which is a pretty handy way to move bits into position.

Before we leave shifts, let's return to the parity calculation mentioned above. Recall that the parity of a number is determined by the number of `1` bits in its binary representation. If odd, the parity is `1`, otherwise it is `0`. The key observation here is that `XOR` preserves the parity of its arguments. For example, if, in binary, `a = 1101` and `b = 0111` then the parity of the two together is zero since there are a total of six on bits. If we apply `XOR` we get `a XOR b = 1010` which also has an even number of on bits and therefore has the same parity. This suggests the trick. If we `XOR` a number with itself but first shift the number half its bit width the resulting bits of the lower half of the output of `XOR` will have the same parity as the original number. We can see this if we look at `a = 01011101` and `XOR` it with itself after shifting down by four bits, which is half the width of the number,

```
   01011101
XOR 00000101
------------
   xxxx1000
```

where we are ignoring the upper four bits. The original number has 5 on bits therefore the parity is `1`. If we look at the lower four bits after the `XOR` we see it also has a parity of `1`. If we repeat the process but using the new value and this time shifting by half its effective width, which is 4 bits, we will end up with a number that has the same parity as we started with and the same parity as the original number. If we repeat this all the way the final result will be a one bit number that is the parity of the original number. Therefore, for `01011101`,

original	`01011101`
right shift 4	`00000101`
`XOR`	`01011000`
right shift 2	`00010110`
`XOR`	`01001110`
right shift 1	`00100111`
`XOR`	`01101001`
parity bit	`xxxxxxx1`

For an 8-bit number we can define a parity function in C as,

```
unsigned char parity(unsigned char x) {
    x = (x >> 4) ^ x;
    x = (x >> 2) ^ x;
    x = (x >> 1) ^ x;

    return x & 1;
}
```

where the `return x & 1;` line returns only the lowest order bit since all other bits are meaningless at this point. The extension of this function to wider integers is straightforward.

Shifting moves bits left or right and drops them at the ends. This is not the only option. Instead of dropping the bits we might want to move the bits that would have fallen off to the opposite end. This is a *rotation* which like a shift can be to the left or right. Unfortunately, while many microprocessors contain rotate instructions as primitive machine language operations neither C nor Python support rotations directly. However, they can be simulated easily enough in code and an intelligent compiler will even turn the simulation into a single machine instruction (e.g., `gcc`).

Again, to keep things simple, we will work with 8-bit numbers. For an 8-bit number, say $\texttt{AB}_{16} = \texttt{10101011}_2$, a rotation to the right of one bit looks like this,

$$\texttt{10101011} \Rightarrow \texttt{1} = \texttt{11010101}$$

where we introduce the $\Rightarrow$ symbol to mean rotation to the right instead of simple shifting ($\rightarrow$). Rotation to the left bit position gives,

$$\texttt{10101011} \Leftarrow \texttt{1} = \texttt{01010111}$$

where we see that the bit that would have fallen off at the left end has now moved around to the right side.

To simulate rotation operators in C and Python we use a combination of << and >> with an OR operator. We need to pay attention to the number of bits in the data type in this case. For example, in C, we can define rotations to the right by any number of bits using,

```
unsigned char rotr(unsigned char x, int n) {
    return (x >> n) | (x << 8 - n);
}
```

with a similar definition for rotations to the left,

```
unsigned char rotl(unsigned char x, int n) {
    return (x << n) | (x >> 8 - n);
}
```

where we have changed << to >> and vice versa. One thing to note is the 8 in the second line of these functions. This number represents the number of bits used to store the data value. Since we have declared the argument x to be of type unsigned char we know it uses eight bits. If we wanted to modify the functions for 32-bit integers we would replace the 8 with 32 or, in general, use sizeof(x) * 8 to convert the byte size to bits.

The Python versions of the C rotation functions are similar.

```
def rotr(x,s):
    return ((x >> s) | (x << 8 - s)) & 0xff
```

and,

```
def rotl(x,s):
    return ((x << s) | (x >> 8 - s)) & 0xff
```

where the only change is that the return value is AND'ed with $\texttt{FF}_{16}$ which as we have seen will keep the lowest eight bits and set all the others to zero. This is again because Python uses 32-bit integers internally and we are interested in keeping the output in the 8-bit range. If we wanted these functions to work with 16-bit integers, we would replace the 8 with 16, as in the C versions, but we would also need to make the mask keep the lowest 16 bits by replacing 0xff with 0xffff.

The rotation functions are helpful, but why do they work? Let's take a look by breaking the operations up individually and seeing how they combine to produce the final result. We start with the original input, $\texttt{AB}_{16} = 10101011_2$, and show, row by row, the first shift to the left, then the second shift to the right, and finally the OR to combine them. The values between the vertical lines are those that fit within the 8-bit range of the number, the other bits are those that are lost in the shift operations,

```
        |10101011|      AB16
        |01010101|1     AB16 >> 1
1010101 |10000000|      AB16 << 8-1
        |11010101|      OR
```

where we see that the bit lost when shifting to the right one position has been added back at the front of the number by the shift to the left and the OR operation. Since the

shifts always introduce a zero bit the `OR` will always set the output bit to the proper value because `1 OR 0 = 1` and `0 OR 0 = 0`. This works for any number of bits to rotate,

```
         |10101011|       AB16
         |00101010|11     AB16 >> 2
1010101  |11000000|       AB16 << 8-2
         |11101010|       OR
```

2.3.4 Comparisons

Magnitude comparison operators take two unsigned integers and return a truth value about whether or not the relationship implied by the operator holds for the operands. Here we look at the basic three, equality ($\texttt{A} = \texttt{B}$), greater than ($\texttt{A} > \texttt{B}$), and less than ($\texttt{A} < \texttt{B}$). There is a second set which can be created easily from the first: not equal ($\texttt{A} \neq \texttt{B}$), greater than or equal ($\texttt{A} \geq \texttt{B}$), and less than or equal ($\texttt{A} \leq \texttt{B}$).

At its core, a computer uses digital logic circuits to compare bits. Figure 2.4 illustrates the layout of a 1-bit comparator. Comparators for larger numbers of bits are repeated instances of this basic pattern. As this not a book on digital logic we will go no further down this avenue but will instead talk about comparing unsigned integers from a more abstract point of view.

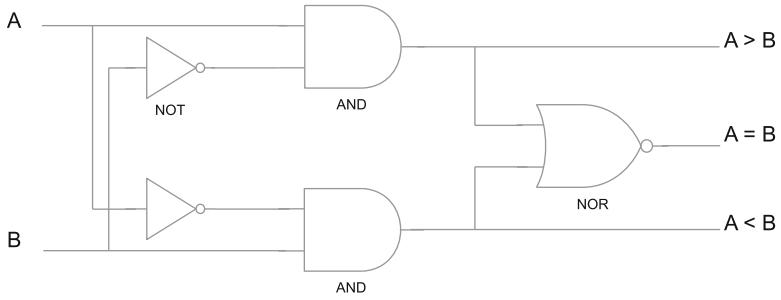

Fig. 2.4 A 1-bit digital comparator. The three output values represent the three possible relationships between the input values `A` and `B`. Recall that input values are `0` or `1`. The output is `1` for the relationship that is true and `0` for those that are not meet. Cascades of this basic form can be used to create multi-bit comparators

Most microprocessors have primitive instructions for comparison of integers as this operation is so fundamental to computing. In addition to direct comparison, many instructions affect processor flags based on privileged numbers like zero. For example, to keep things simple, the 8-bit 6502 microprocessor, which has a single accumulator, `A`, for arithmetic, performs comparisons with the `CMP` instruction but also sets processor status flags whenever a value is loaded from memory using the `LDA` instruction. There are other registers and instructions, of course, but we focus

on the accumulator to keep the example simple. The 6502 uses branch instructions like BEQ and BNE to branch on equal or not equal respectively. This also applies to the implied comparison with the special value 0 which happens when the LDA instruction loads the accumulator.

Armed with this brief review of an old 8-bit microprocessor we can see that the following set of instructions would indeed perform an 8-bit comparison between 22_{16} already in the accumulator via the first LDA instruction and $1A_{16}$ stored in memory location 2035_{16} and branch if they are not equal. Additionally, we will also perform an implicit comparison of the value of memory location $20FE_{16}$ with zero and branch if it is,

```
LDA #$22       ; A = $22
CMP $2035      ; compare to location $2035
BNE noteql     ; branch to "noteql" if not equal

LDA #$20FE     ; A = contents of $20FE
BEQ iszero     ; branch to "iszero" if zero
```

where we use the classic notation of $\$22 = 22_{16}$.

Why bring up this example? In part to show that comparison is very fundamental to computers and is done as efficiently as possible in hardware and to set the stage for our less efficient alternatives to digital logic. The comparisons we are implementing in code are pure hardware even in the simplest of microprocessors.

Since for any two integers A and B exactly one of the following is true: $A = B$, $A < B$, or $A > B$, it follows that if we know how to test for any two the last condition is simply when neither of the two we can test for applies. In our case, we look at the situation where we know how to test for equality ($A = B$) and greater than ($A > B$). We do this with two predicate functions that simply return 1 if the relationship holds for the arguments and 0 otherwise. Let's call these predicates `isZero(A)` and `isGreater(A,B)` and see how we might implement them directly in C for 8-bit values using the `unsigned char` data type.

You may be wondering why we chose to use `isZero(A)` instead of the perhaps more obvious `isEqual(A,B)`. If so, good, you are paying attention. Given our experience with the XOR operator we now know that,

$$a \text{ XOR } a \rightarrow 0$$

so we can immediately see that,

```
isEqual(A,B) = isZero(A XOR B)
```

but how do we implement `isZero(A)`? One approach in code would be to shift the bits and test the lowest order one. If we find one that is not zero then the number is not zero. The test is via an OR which is only zero when both operands are zero. We use AND to do the bit comparison and then shift the result down so that the compared bit is in the lowest position. Then, the OR of all these tests will be 1 if any bits are set and 0 if not. This is the exact opposite of what we want so we add a NOT to reverse the sense of the logic and a final AND with 1 to mask out all other bits

and return the state of the lowest bit only. This, then, is the full predicate function `isZero(x)`,

```
unsigned char isZero(unsigned char x) {
    unsigned char ans;

    return ~(((x & (1<<7)) >> 7) |  // test bit 7
             ((x & (1<<6)) >> 6) |
             ((x & (1<<5)) >> 5) |
             ((x & (1<<4)) >> 4) |
             ((x & (1<<3)) >> 3) |
             ((x & (1<<2)) >> 2) |
             ((x & (1<<1)) >> 1) |
              (x & 1)) & 1;         // test bit 1
}
```

Notice that there are no actual comparison operators in this function, only logical bitwise operators. With this we can quickly implement `isEqual(x,y)`,

```
unsigned char isEqual(unsigned char x,
                      unsigned char y) {
    return isZero(x ^ y);
}
```

We now need `isGreater(A,B)` which is implemented with bit operators, shifts, and a call to `isZero(x)`. This is why we started with `isZero(x)` instead of `isEqual(x,y)`. The C code for our function is given first followed by the explanation of why it works,

```
 1 unsigned char isGreater(unsigned char a,
 2                         unsigned char b) {
 3     unsigned char x,y;
 4
 5     x = ~a & b;
 6     y = a & ~b;
 7
 8     x = x | (x >> 1);
 9     x = x | (x >> 2);
10     x = x | (x >> 4);
11
12     return ~isZero(~x & y) & 1;
12 }
```

In order to tell if `a` is greater than `b` we need to know the first place where their respective bits do not agree. Once we know this bit position we know that `a` is greater than `b` if at that bit position `a` has a `1` while `b` has a `0`. So, we need to find the locations of where the bits differ. To make the example concrete, we let `a` be $\texttt{00011101}_2 = 29$ and `b` be $\texttt{00011010}_2 = 26$. If we look at line 5 we see,

```
x = ~a & b;
```

which sets `x` to the `AND` of the `NOT` of `a` and `b`. This leaves `x` with a `1` in all the positions where the bit in `a` is less than the same bit in `b`. For our example this

sets x to 00000010_2 which tells us that the only bit position in a that is less than the corresponding bit position in b is bit 1. Likewise, line 6 asks where are the bit positions where a is greater than b? In this case, we set y to 00000101_2 to indicate that in bit 0 and bit 2 the value of a is greater than b. In order to see if a is greater than b we need to find the highest bit position where the two differ and see if that bit is set. We can do this if we take the value in x, which tells us where a bits are less than b bits, and build a mask which is 1 for all bit positions at or below the highest bit position where a is first less than b. We do this with lines 8 through 10. This operation which ORs the value with shifted versions of itself duplicates the highest 1 bit among all the lower bits. In this case,

```
x                  → 00000010
x = x | (x >> 1) → 00000011
x = x | (x >> 2) → 00000011
x = x | (x >> 4) → 00000011
```

where the last two steps add nothing new since `x >> 2` and `x >> 4` both result in zero which will set no new bits. We now have a mask in x that tells us all the bit positions below the first place where the bit in a is less than the bit in b. If we NOT this mask, $\neg$ 00000011 $\rightarrow$ 11111100, we can use the new mask to tell us all the bit positions where a was not less than b. Lastly, with this mask and the value in y which tells us where the bits in a were greater than the bits in b, we can perform one final AND, `~x & y`, which will result in zero if $a \leq b$ since no bits will be set in y in the region where the bits of a were greater than those of b, or a nonzero value since at least one bit will be set in y in that region. Line 12, then, asks if this result is zero by passing the output to `isZero`. It then applies NOT to change the output of `isZero` since the result is zero when $a \leq b$ and not zero when $a > b$. The final AND with 1 gives us only the final bit since the NOT will change all the bits of the result of `isZero`.

We are nearly finished with our comparison operators. We have equality (`isEqual`) and greater than (`isGreater`). With these we see that `isLess` would be,

```
unsigned char isLess(unsigned char x,
                     unsigned char y) {
    return (!isEqual(x,y)) && (!isGreater(x,y));
}
```

which is reasonable since for any two unsigned integers A and B, if $A \not\geq B$ then $A < B$ must be true. Testing for not equal is even simpler,

```
unsigned char isNotEqual(unsigned char x,
                         unsigned char y) {
    return !isEqual(x,y);
}
```

since the only way for A to *not* not equal B is if the two are indeed equal. Less than or equal and greater than or equal follow directly from the functions already defined,

```
unsigned char isLessOrEqual(unsigned char x,
                            unsigned char y) {
    return isEqual(x,y) || isLess(x,y);
}
```

and

```
unsigned char isGreaterOrEqual(unsigned char x,
                               unsigned char y) {
    return isEqual(x,y) || isGreater(x,y);
}
```

which completes our implementation of comparison operators using only bitwise operations.

2.3.5 Arithmetic

Just as comparison is a fundamental microprocessor operation, so is arithmetic. In this section we look at arithmetic with unsigned binary integers, first from the point of view of doing it "by hand" and then from the point of view of a simple 8-bit microprocessor. These two approaches should illustrate the mechanism behind the operations. We will not, however, attempt to implement these operations in C as we did above for the comparison operators though we will use C to demonstrate overflow and underflow conditions.

The addition facts in binary are,

$$\begin{aligned} 0 + 0 &= 0 \\ 0 + 1 &= 1 \\ 1 + 0 &= 1 \\ 1 + 1 &= 0\ \texttt{carry}\ 1 \end{aligned}$$

from which we see that one of them produces a carry since it results in a two digit number. Just as in decimal, the carry is applied to the next digit over to the left. So, to add two unsigned 8-bit binary numbers we move right to left, bit by bit, adding and moving any carry to the next digit to the left,

$$\begin{array}{rl} \texttt{11111} & \leftarrow \textit{carry} \\ \texttt{01101110} & \leftarrow \textit{first operand} \\ +\ \texttt{00110101} & \leftarrow \textit{second operand} \\ \hline \texttt{10100011} & \leftarrow \textit{answer} \end{array}$$

where the carry from the second to leftmost bit does not cause difficulty since the highest bits of each number are zero. However, what would happen if there was a carry from the leftmost bit? In mathematics, nothing special would happen, there would simply be a new leftmost bit, but in computers this is not the case. Recall that we are working with 8-bit unsigned numbers which means that all numbers fit in eight bits in memory. If we use eight bits for numbers we have no place in which to put any final carry bit. This results in an *overflow* condition. The computer simply discards this new highest value bit and retains the lowest eight bits which fit in memory,

$$\begin{array}{rrr} & 11111 & \\ & 11101110 & \texttt{EE}_{16} \\ + & 00110101 & 35_{16} \\ \hline 1 & 00100011 & 123_{16} \end{array}$$

which is stored as 23_{16} discarding the carry on the leftmost bit. This is precisely what we see with the following C code,

```
#include <stdio.h>

int main() {
    unsigned char x, y, z;

    x = 0xEE;
    y = 0x35;
    z = x + y;

    printf("%x\n", z);
}
```

Let's take a look at how a simple 8-bit microprocessor would implement an addition. In this case, we look at an unsigned 16-bit addition which requires two addition operations. Working again with the 6502 processor mentioned above we see that an 8-bit addition between a value in memory locations 23_{16} and 25_{16} will involve a load into the accumulator (`LDA`), a clearing of the carry flag which catches any overflow bit (`CLC`) and an addition with memory (`ADC`). Specifically, we assume memory location 23_{16} contains $\texttt{EE}_{16}$, memory location 25_{16} contains 35_{16}. We then load the accumulator, clear the carry and add,

`LDA $23`	$A \leftarrow \text{EE}$
`CLC`	$C \leftarrow 0$
`ADC $25`	$A \leftarrow A + 35 + C$
	$A \leftarrow 23, C \leftarrow 1$

with the accumulator set to 23_{16}, the lowest eight bits of the sum of $\texttt{EE}_{16}$ and 35_{16}, and the carry flag set to `1` to indicate an overflow happened. This setting of the carry flag is the key to implementing multibyte addition. In C, we would simply declare the variables to be of data type `unsigned short` which is 16-bits and add as before. For example, in C we have,

```
#include <stdio.h>

int main() {
    unsigned short x, y, z;

    x = 0xEE;
    y = 0x35;
    z = x + y;

    printf("%x\n", z);
}
```

which gives us a 16-bit answer of 123_{16}. In memory, using little-endian representation for multibyte numbers, we have,

```
memory location $24 : 23
                $25 : 01
```

since we store the lowest byte first.

In the simpler world of the 8-bit microprocessor we store the lowest part of the sum, the low byte, and add the high bytes without clearing the carry. Assuming memory is set to

```
memory location $23 : EE
                $24 : 00
                $25 : 35
                $26 : 00
```

we clear the carry flag, add the two low bytes, store the partial sum, add the high bytes with any carry, and store the final part of the sum like this,

```
LDA $23  | A ← EE
CLC      | C ← 0
ADC $25  | A ← A + 35 + C,   A=23
STA $27  | $27 ← 23, C ← 1
LDA $24  | A ← 0
ADC $26  | A ← A + 0 + C,   A=1
STA $28  | $28 ← 1
```

where we have introduced a new instruction, `STA`, which stores the accumulator in memory. When this sequence of instructions is complete we have the final answer in memory locations \$27 and \$28 as 23_{16} and 1_{16} respectively as we expect for a little-endian number.

The addition above is equivalent to this single 16-bit addition

```
          1 11111
  00000000 11101110  EE16
+ 00000000 00110101  3516
-------------------------
  00000001 00100011 12316
```

where we have separated the upper eight bits of the high byte from the lower eight bits of the low byte.

Addition of unsigned binary numbers is by now straightforward. We add, left to right, bit by bit with carry bit when necessary. If the result is too large, we overflow and retain the lowest n bits where n is the width of the number in bits. We now move on to subtraction of unsigned binary numbers.

The subtraction facts in binary are,

$$
\begin{aligned}
0 - 0 &= 0 \\
0 - 1 &= 1, \textit{underflow} \\
1 - 0 &= 1 \\
1 - 1 &= 0
\end{aligned}
$$

where the *underflow* condition will require a borrow from the next higher bit position. Like overflow, *underflow* is the situation where we cannot properly represent the number using the number of bits we have to work with. In this case, the underflow happens when we attempt to subtract a larger number from a smaller and we have no way to represent the resulting negative number. We'll address this issue below.

To continue with the example we used for addition, we now evaluate $\texttt{EE}_{16} - \texttt{35}_{16}$ to get $\texttt{B9}_{16}$. In binary, using the subtraction facts, we have,

$$\begin{array}{rr} & \texttt{01} \\ & \texttt{01}\ \ \texttt{01} \\ & \texttt{11101110}\ \texttt{EE}_{16} \\ - & \texttt{00110101}\ \texttt{35}_{16} \\ \hline & \texttt{10111001}\ \texttt{B9}_{16} \end{array}$$

with each borrow written above the bit and the next bit set to one less than it was. If a second borrow is necessary for a bit position, we write it above again. Let's look at the subtraction again, bit by bit, right to left, we are computing,

bit 0	10 - 1 = 1, *borrow*
bit 1	0 - 0 = 0
bit 2	1 - 1 = 0
bit 3	1 - 0 = 1
bit 4	10 - 1 = 1, *borrow*
bit 5	10 - 1 = 1, *borrow*
bit 6	0 - 0 = 0
bit 7	1 - 0 = 1

which, reading from bottom to top, gives $\texttt{10111001}_2 = \texttt{B9}_{16}$ as expected.

What happens if we need to borrow across more than one bit position? For example, in base 10 a problem like,

$$\begin{array}{r} \texttt{7003} \\ \texttt{-\ \ 972} \\ \hline \texttt{6031} \end{array}$$

involves borrowing across two digits in order to subtract the `7` of `972` which we can write as,

$$\begin{array}{r} \texttt{69}^{1}\texttt{03} \\ \texttt{-\ \ 9\ 72} \\ \hline \texttt{6031} \end{array}$$

where we change the `700` into $\texttt{69}^{1}\texttt{0}$ to subtract `9 7` giving the partial result `603`. We subtracted one from the next two digits and added it in as a ten to the digit we were working with. The same thing happens in binary. Consider this subtraction problem,

$$\begin{array}{rr} & \texttt{01} \\ & \texttt{1010}\cancel{\texttt{10}}^{\texttt{1}}\texttt{01}\ \ \texttt{A9}_{16} \\ - & \texttt{101001 10}\ \ \texttt{A6}_{16} \\ \hline & \texttt{00000011}\ \ \texttt{03}_{16} \end{array}$$

where we attempt to subtract `1` from `0` in the second bit position (bit 1, since we always count from zero and from right to left). We need to borrow from bit 2, but since this is also zero, we instead borrow from bit 3 and change `100` into $\texttt{01}^{\texttt{1}}\texttt{0}$ in order to do the subtraction.

As we are working with 8-bit unsigned integers one will eventually be tempted to ask what happens if we try to subtract `1` from `0` since we cannot represent `-1`. What happens is we get the result we would get if we had an extra bit at left-most position and borrowed from it like so,

$$\begin{array}{rr} & \texttt{0 1111111} \\ & \cancel{\texttt{1}}\ \cancel{\texttt{0000000}}^{\texttt{1}}\texttt{0} \\ - & \texttt{0000000 1} \\ \hline & \texttt{11111111} \end{array}$$

meaning that subtracting one from the smallest number we can represent, namely zero, results in the largest number we can represent which for an 8-bit unsigned integer is every bit set, $2^8 - 1 = 255$. Another way to think about it is that the numbers form a loop with `00000000` and `11111111` set next to each other. If we move one position down below zero we wrap around to the top and get `255`. Likewise, if we move one position up from `255` we will wrap around to the bottom and get zero. Moving down below zero is an underflow condition while moving up above `255` is an overflow.

Before we move on to multiplication and division of unsigned integers, let's look at one more subtraction example that would result in a negative number. We'll use our running example but swap the operands,

$$\begin{array}{rr} & \texttt{1 00110101}\ \ \texttt{35}_{16} \\ - & \texttt{11101110}\ \ \texttt{EE}_{16} \\ \hline & \texttt{01000111}\ \ \texttt{47}_{16} \end{array}$$

where we have indicated the implied `1` bit from which we can borrow. This implied `1` bit is in the bit position for $2^8 = 256$ which suggests another way to think about the answer we would expect if we go negative in a subtraction of unsigned numbers. For our example, $\texttt{35}_{16} - \texttt{EE}_{16} = -185$, but if we add in 256 which is essentially what we are doing in thinking there is an implied extra high bit, we get $-185 + 256 = 71 = \texttt{47}_{16}$ which is the answer we found previously. We have been working with 8-bit wide unsigned integers. If we are using 16-bit integers the implied bit 16 (recall, we count bits from zero) is $2^{16} = 65536$ which means we would add 65536 to any negative value to get the result we would expect from unsigned subtraction.

The following C example demonstrates that what we have been discussing is indeed the case,

```
#include <stdio.h>

int main() {
    unsigned char x=0xEE, y=0x35, z;

    z = x - y;
    printf("%X\n", z);

    z = y - x;
    printf("%X\n", z);

    z = 0 - 1;
    printf("%X\n", z);
}
```

The output of this program is,

$$\begin{aligned} \texttt{B9}_{16} &= \texttt{10111001}_2 \\ \texttt{47}_{16} &= \texttt{01000111}_2 \\ \texttt{FF}_{16} &= \texttt{11111111}_2 \end{aligned}$$

which is exactly what we saw in the examples above.

Now that we know how to subtract in binary we can examine a useful trick involving AND. If one bit in a number is set this implies that the number is a power of two since every position in a binary number is, by definition, a power of two. If we know which bit we want to test for, which power of two, it is straightforward to use a mask and check that bit. But, what if we wanted to know if the number in question was any power of two? We can use AND here along with an observation about bits that are on in a number that is a power of two. For example, if the number we want to test is $\texttt{00100000}_2 = 32$ we see that it is a power of two and only one bit is on. Now, subtract one from this number. In this case, we will get $\texttt{00011111}_2 = 31$. What has happened is that the single bit that was on is now off and some of the bits that were off are now on. Finally, what happens if we AND these two values together? We get,

$$\begin{array}{rr} & \texttt{00100000} = 32 \\ \texttt{AND} & \texttt{00011111} = 31 \\ \hline & \texttt{00000000} = 00 \end{array}$$

which is exactly zero. From this we see that we will only get exactly zero when one of two conditions are met: either the number is itself zero or it is a power of two which had only one bit set. This is nicely captured in a simple C function,

```
unsigned char is_power_of_two(unsigned char n) {
    return (n == 0) ? 0
                    : (n & (n-1)) == 0;
}
```

which returns `1` if the argument is a power of two and `0` otherwise. The function checks if the argument is zero, if so, return `0`. If not, then check whether `n & (n-1)` is exactly `0`. If it is, the expression is true and the function returns `1` to indicate a power of two, otherwise it returns `0`. While written for `unsigned char` data type the function will work for any unsigned integer type.

We've looked in detail at addition and subtraction, now we turn our attention to multiplication and division. Modern microprocessors perform multiplication and division as operations in hardware. This is a very good thing but makes it difficult for us in a sense so we will, as before, look at more primitive approaches which might be used in small 8-bit microcontrollers that lack hardware instructions for multiplication and division. To keep things simple we will illustrate the algorithms in C even though this would never be done in practice.

Since multiplication is really repeated addition one approach to finding $\mathtt{n} \times \mathtt{m}$, where `n` and `m` are unsigned integers, would be to add `n` to itself `m` times or vice versa. Naturally, it would make sense to run the loop as few times as possible so we would loop over the smaller of `n` or `m` adding the other number. In C we have,

```
unsigned short mult1(unsigned char n, unsigned char m) {
    unsigned char i;
    unsigned short ans = 0;

    if (n < m) {
        for(i=0; i < n; i++)
            ans += m;
    } else {
        for(i=0; i < m; i++)
            ans += n;
    }

    return ans;
}
```

which leaves the product of the 8-bit numbers in `n` and `m` in the now possibly 16-bit value `p`. Why is the product possibly 16-bits? Because the largest possible number we can get by multiplying two 8-bit numbers requires 16-bits to store it in memory since $255 \times 255 = 65025$ which is above $2^8 - 1 = 255$ meaning it needs more than 8-bits to store but is below $2^{16} - 1 = 65535$ which is the maximum for a 16-bit unsigned integer.

Is this really a good way to multiply numbers, however? Probably not. The loop needs to be repeated for the smaller of `n` or `m` which may be up to `255` times. Given we must do 16-bit addition inside the loop, recalling the example above, we see that the simple multiplication may turn into many thousands of individual machine instructions. Surely we can do better than this? To answer this question, let's look a little more closely at multiplication in binary as we might do it by hand,

```
          00010100  14₁₆ =  20
×         00001110  0E₁₆ =  14
-----------------------------
          00000000
         00010100
        00010100
+       00010100
-----------------------------
        00100011000 118₁₆ = 280
```

in which we see that if the binary digit in the multiplier is `0` we simply copy down all zeros and if it is a `1` we copy the multiplicand lining it up beneath the multiplier bit as we would do in decimal. Then, again as in decimal multiplication, we add all the partial products to arrive at the final answer. For simplicity we did not write the leading zeros which would be present if showing all 16-bits of the result.

This method suggests a possible improvement over our existing multiplication function, `mult1`. Rather than repeatedly add the multiplier or multiplicand, we can copy the process just shown by shifting the multiplicand into position and adding it to the partial product if the multiplier bit is `1` otherwise ignore those that are `0`. This leads to a second multiplication function in C,

```
unsigned short mult2(unsigned char n, unsigned char m) {
    unsigned char i;
    unsigned short ans = 0;

    for(i=0; i < 8; i++) {
        if (m & 1) {
            ans += n << i;
        }
        m >>= 1;
    }

    return ans;
}
```

which, when compared to `mult1` and run ten million times proves to be about 1.6x faster. Let's look at what `mult2` is actually doing.

We are multiplying two 8-bit numbers so we need to look at each bit in the multiplier, `m`. This is the source of the `for` loop. The `if` statement `AND`'s the multiplier with `1` to extract the lowest bit. If this bit is set, we want to add the multiplicand, `n`, to the partial product stored in `ans`. Note, though, that before we add the multiplicand, we need to shift it up to the proper bit position. Note also that this works because the result of the operation is a 16-bit value which will not lose any of the bits of `n` when we do the shift. Regardless of whether we add anything to the partial product we need to shift the multiplier down one bit so that in the next pass through the loop the `if` will be looking at the next highest bit of the original `m`. Lastly, we see that there are no side-effects to this function because C passes all arguments by value meaning `n` and `m` are copies local to the function..

The speed improvement between `mult1` and `mult2` becomes much more dramatic when we move from multiplying two 8-bit numbers to multiplying two 16-bit numbers. To do this, we take the source for `mult1` and `mult2` and replace

all instances of `unsigned char` by `unsigned short` and all instances of `unsigned short` by `unsigned int`. Lastly, we change the loop limit in `mult2` from `8` to `16` since we are multiplying two 16-bit numbers. When this done we see that `mult2` is nearly 3500x faster than `mult1` for the same arguments (assuming both to be near the limit of `65535`).

What about division? We cover two operations with division since the algorithm returns the quotient and any remainder. The operations are integer division (/) which returns the quotient and modulo (%) which returns the remainder. For example, we need an algorithm that produces these answers,

```
123 / 4 = 30
123 % 4 =  3
```

since $123/4 = 30$ with a remainder of 3.

We could implement division by repeated subtraction. If we count the number of times we can subtract the divisor from the dividend before we get a partial result that is less than the divisor we will have the quotient and the remainder. We might code this in C as,

```
unsigned char div1(unsigned char n,
                   unsigned char m,
                   unsigned char *r) {
    unsigned char q=0;

    *r = n;

    while (*r > m) {
        q++;
        *r -= m;
    }

    return q;
}
```

and test it with,

```
int main() {
    unsigned char n=123, m=7;
    unsigned char q,r;

    q = div1(n, m, &r);
    printf("quotient=%d, remainder=%d\n", q,r);
}
```

which prints `quotient=30, remainder=3` which is the answer we are expecting.

This function requires three arguments since we want to return the quotient as the function value and the remainder as a side-effect value. This is why we pass the remainder as a third argument using a pointer. Inside of `div1` we set the remainder (`r`) to our dividend and continually subtract the divisor (`m`) until we get a result less than the divisor. While doing this we keep count of the number of times we subtract in `q` which we return as the quotient.

Like our `mult1` example above, `div1` is an inefficient way to implement division. What happens if the dividend is large and the divisor is small? We must loop many times in that case before we are done. The problem is even worse if we use integers larger than 8-bits. What to do, then?

Just as we did for multiplication, let's look at binary division by hand. Unlike decimal long division, binary division is rather simple, either the divisor is less than or equal to the dividend in which case the quotient bit is `1`, otherwise, the quotient bit is `0`; there are no trial multiplications. Dividing $123 = \texttt{01111011}_2$ by $4 = \texttt{100}_2$ in this way gives,

$$
\begin{array}{r}
\texttt{00011110} \\
\texttt{100}\,\overline{)\,\texttt{01111011}} \\
\texttt{0} \\
\hline
\texttt{01} \\
\texttt{0} \\
\hline
\texttt{011} \\
\texttt{0} \\
\hline
\texttt{0111} \\
\texttt{100} \\
\hline
\texttt{0111} \\
\texttt{100} \\
\hline
\texttt{0110} \\
\texttt{100} \\
\hline
\texttt{0101} \\
\texttt{100} \\
\hline
\texttt{11} \\
\texttt{0} \\
\hline
\texttt{11}
\end{array}
$$

with $\texttt{00011110}_2 = 30$ and a remainder of $\texttt{11}_2 = 3$ as expected.

As we have seen several times already, modern microprocessors implement such a fundamental operation as division in hardware, but we can look at unsigned division in the way it might be implemented in a more primitive microprocessor or microcontroller. For simplicity, we again implement the algorithm in C. With all of this in mind, we get Fig. 2.5 which will require a bit of explanation.

The key to understanding what Fig. 2.5 is doing it to observe that binary division by hand is really a matter of testing whether or not we can subtract the divisor from the partial dividend. If so, we set a one in that bit of the quotient, otherwise we set a zero. The algorithm of Fig. 2.5 is setup for 8-bit division using 8-bit dividends and divisors, and by implication quotients. Therefore, we need to examine all eight bits of the dividend starting with the highest bit. To do this, and to save space as this algorithm is often implemented in hardware, we take advantage of the fact that C passes arguments by value and use `n` to be both dividend and quotient. When we have examined all eight bits the value in `n` will be the quotient. We can do this because as we look at each bit of the dividend we shift it out to the left while shifting in the new bit of the quotient from the right.

```
 1  unsigned char div2(unsigned char n,
 2                     unsigned char m,
 3                     unsigned char *r) {
 4      unsigned char i;
 5
 6      *r = 0;
 7
 8      for(i=0; i<8; i++) {
 9          *r = (*r << 1) + ((n & 0x80) != 0);
10          n <<= 1;
11
12          if ((*r-m) >= 0) {
13              n |= 1;
14              *r -= m;
15          }
16      }
17
18      return n;
19  }
```

Fig. 2.5 Shift, test, and restore unsigned integer division

We store the remainder in `r` and pass it back out of the function by using a pointer. To start the division process we set `r` to zero and the quotient to the dividend in `n`. Since `n` already has the dividend there is no explicit operation to do this, we get it for free. If we call `div2(123, 4, &r)` to continue with our running division example, the state of affairs in binary after the execution of line 6 in Fig. 2.5 is,

i	r	n	m
undefined	00000000	01111011	100

where the dividend is in `n` and the remainder is zero. We next hit the loop starting in line 8. This loop executes eight times, once for each bit of the dividend. Lines 9 and 10 perform a double left shift. This is the equivalent of treating `r` and `n` as a single 16-bit variable with `r` the high order bits. First we shift `r` one bit to the left (`*r << 1`) and then comes the rather cryptic expression,

```
((n & 0x80) != 0)
```

which tests whether the highest bit in `n`, our dividend and quotient is set. Recall our discussion of `AND` above. If it is set, we add it into `r`. This is because we are about to left shift `n` one bit and if the bit we are shifting out of `n` is set, we need to move it to `r` to complete the virtual 16-bit shift of `r` and `n`. We then shift `n` in line 10.

Line 12 checks whether or not we can subtract the divisor in `m` from the partial dividend which is being built, bit by bit, in `r`. If we can, we set the first bit of `n`, our quotient, in line 13 and then update the partial dividend by subtracting the divisor in line 14. Recall, we are examining the dividend one bit at a time by moving it into `r`. We are simultaneously storing the quotient in `n` one bit at a time by putting it in on the right side. Since we already shifted to the left the first bit in `n` is always zero,

we only update it if the subtraction succeeds. After this first pass through the loop we have,

i	r	n	m
0	00000000	11110110	100

with the first bit of the quotient, a zero, in the first bit of `n`. If we continue through the loop we will get the following sequence of values,

i	r	n	m
1	00000001	11101100	100
2	00000011	11011000	100
3	00000011	10110001	100
4	00000011	01100011	100
5	00000010	11000111	100
6	00000001	10001111	100
7	00000011	00011110	100

where we end with a quotient of `30` in `n` which is the return value of the function and a remainder in `r` of `3`. Notice how `n` changes as we move through the loop. Each binary digit is shifted into `r` from the right as the new quotient bits are assigned from the left until all bits are examined. This algorithm, unlike the `div1` example, operates in constant time. There are a fixed number of operations needed regardless of the input values.

2.3.6 Square Roots

We briefly consider here a simple integer square root algorithm which makes use of an interesting mathematical fact. This algorithm works by counting the number of times an ever increasing odd number can be subtracted before reaching or going below zero. The algorithm itself is easy to implement in C,

```
unsigned char sqr(unsigned char n) {
    unsigned char c=0, p=1;

    while (n >= p) {
        n -= p;
        p += 2;
        c++;
    }

    return c;
}
```

where we again make use of the fact that C passes arguments by value which allows us to modify `n` in the function without changing it outside of the function. Our count, which will be the square root of `n`, is initialized to zero in `c`. We start our odd number in `p` at `1` and then move to `3`, `5`, `7`, and so on. The `while` loop is checking to see if our `n` value is still larger or the same as `p` and if so, we subtract `p` and count

one more subtraction in `c`. When `n` is less than `p` we are done counting and return `c` as the square root. Naturally, this algorithm is only approximate by underestimating when `n` is not actually a perfect square.

If we call `sqr` with `64` as the argument, we get the following sequence of values in the `while` loop,

n	p	c
63	3	1
60	5	2
55	7	3
48	9	4
39	11	5
28	13	6
15	15	7
0	17	8

where the final value of `n` is zero since `64` is a perfect square and `c` is `8`, which is the square root of `64`. We see that the algorithm works, but why?

The trick is the observation that the sum of the sequence of odd numbers is always a perfect square. For example,

$$\begin{aligned} 1 &= 1 \\ 1+3 &= 4 \\ 1+3+5 &= 9 \\ 1+3+5+7 &= 16 \\ 1+3+5+7+9 &= 25 \end{aligned}$$

or more compactly,

$$\sum_{i=1}^{n} 2i-1 = n^2$$

where n^2 is the argument to `sqr` and n is the square root.

2.4 What About Negative Integers?

In the previous section we took a thorough look at unsigned integers and the sorts of operations computers typically perform on them. Without a doubt, unsigned integers are the mainstay of computers, but often it is necessary to represent quantities that are less than zero. What do we do about that? In this section we examine three options, two in detail, for tracking the sign of an integer and performing operations with signed integers. We will naturally build on what we have learned about unsigned integers and bear in mind that, as before, while we may show examples using 8-bit numbers for simplicity, everything immediately translates to numbers with more bits, be they 16, 32, or larger.

The most common techniques for handling signs are sign-magnitude, one's complement, and two's complement.

2.4.1 Sign-Magnitude

Perhaps the most natural way to represent the sign of an integer is to reserve one bit of its representation for the sign and this is precisely what early computers did. If we decide that we will keep the highest-order bit of the number for the sign we can use the remainder of the bits to represent the magnitude as an unsigned integer and this is the *sign-magnitude* form,

```
01111111 =  127
...
00000010 =    2
00000001 =    1
00000000 =    0
10000001 =   -1
10000010 =   -2
...
11111111 = -127
```

This seems to be a perfectly reasonable way to store a signed integer but notice a few things,

- We lose range in terms of magnitude. An unsigned 8-bit number can take values from `0` to `255` while a sign-magnitude number is restricted to `-127` to `+127`. As we will see, keeping track of the sign always results in a loss of magnitude range.
- There are two ways to represent zero: `+0` = `00000000` and `-0` = `10000000`. This seems unnecessary and wasteful of a bit pattern.
- Arithmetic becomes more tedious since we need to bear the sign of the number in mind at all times. It would be nice to be able to do some arithmetic without requiring separate logic for the sign.

For the reasons given above, especially the additional hardware logic, the sign-magnitude representation for integers has been abandoned by modern computer systems. Let us now turn our attention to possible replacements.

2.4.2 One's Complement

Our first candidate for a suitable replacement to the sign-magnitude form is called *one's complement*. In this notation we represent negative numbers by taking the positive form and calculating the one's complement. The one's complement is simple to do, just negate (logical `NOT`) every bit in the positive form of the number. So, we have,

```
01111111 =  127
...
00000010 =    2
00000001 =    1
00000000 =    0
11111110 =   -1
11111101 =   -2
...
10000000 = -127
```

which again seems good in that we can look at the highest order bit to see if the number is negative or not and as we will shortly see we can use this notation for arithmetic without too much trouble, but we do still have two ways to represent zero since `00000000` → `11111111`.

2.4.3 Two's Complement

The one's complement of a positive number is the bit pattern we get when we change all the zero bits to one and all the one bits to zero. The *two's complement* of a positive number is the bit pattern we get when we take the one's complement and then add one to it. This is the notation that has been accepted as the way to represent negative integers and the advantages will be come clear when we look at operations on signed integers. As with one's complement integers, a positive two's complement integer is represented in just the same way as an unsigned integer. With two's complement we have,

```
01111111 =  127
...
00000010 =    2
00000001 =    1
00000000 =    0
11111111 =   -1
11111110 =   -2
...
10000000 = -128
```

where we now have only one way to represent zero since,

`00000000` → `11111111` → `00000000`
positive *one's complement* *two's complement*

since adding one to `11111111` maps back around to `00000000` with the overflow bit ignored. Additionally, we have increased our range by one since we can represent numbers from `-128` to `+127` instead of `-127` to `127` as with one's complement or sign-magnitude.

2.5 Operations on Signed Integers

We would like to be able to perform operations on signed integers. The bit level operations like AND, OR and NOT work the same way with signed integers as with unsigned integers. To these operators, the bits are just bits, the "fact" of a negative integer is just a convention forced on certain bit patterns. Since this is the case, we need only look at how to compare negative integers, how to perform arithmetic on negative integers, and, as a special operation, how to deal with the sign of a two's complement integer when changing the number of bits used to represent the number. Let us first start with comparing two signed integers.

2.5.1 Comparison

Comparison of two signed integers, *A* and *B*, implies determining which relational operator, $<$, $>$, or $=$, should be put between them. When we compared unsigned integers we looked at the bits from highest to lowest. We still to that for signed integers but we need to first consider the signs of the two numbers. If the signs differ we know very quickly the relationship between the numbers without considering all the bits. If the signs match, either both positive or both negative, we need to look at the magnitude bits to see where they might be different. This will tell us the relationship between the two numbers. Naturally, if the bits are all the same, position for position, then the two numbers are equal and use the exact same bit pattern.

We can use a C function like the one in Fig. 2.6, with included helper function to determine whether or not a particular bit position is on, to compare two signed numbers in two's complement notation. Note that we are now working with variables of type `signed char` which are 8-bit signed integers.

The helper function (`bset`, lines 1–4) returns `1` if the *n*-th bit of *v* is on, otherwise it returns a `0`. It uses the shift and AND mask trick we saw above to test a bit position value. The main function, `scomp`, looks first at the signs of the arguments (lines 10–13) and returns the relationship if they differ. If the sign bit of `a` is zero, `a` is positive. If the sign bit of `b` is one, then `b` is negative and `a` must be greater than `b` so return `1` to indicate `a` $>$ `b`. If the signs are reversed, `a` is less than `b` so return `-1` to indicate `a` $<$ `b`.

If the signs of `a` and `b` match, either positive or negative, we then look at the remaining bits from highest to lowest to see where there are any differences. This is the loop of lines 15 through 20 in Fig. 2.6. If we find a bit position where `a` has a zero and `b` has a one we know that `a` $<$ `b` must be true so we return `-1`. Likewise, if we find that `a` is one and `b` is zero at that bit position we know that `a` $>$ `b` so we return `1`. Finally, if we make it through all the bits and find no differences the two integers are equal so we return `0`. With a comparison function like `scomp` it is easy to create predicate functions checking for equal, less than and greater than. Recall that a predicate function is a function that returns true or false. For example,

```
signed char bset(signed char v,
                 signed char n) {
    return (v & (1 << n)) != 0;
}

signed char scomp(signed char a,
                  signed char b) {
    unsigned char i;

    if ((bset(a,7) == 0) && (bset(b,7) == 1))
        return 1;
    if ((bset(a,7) == 1) && (bset(b,7) == 0))
        return -1;

    for(i=0; i<7; i++) {
        if ((bset(a,6-i) == 0) && (bset(b,6-i) == 1))
            return -1;
        if ((bset(a,6-i) == 1) && (bset(b,6-i) == 0))
            return 1;
    }

    return 0;
}
```

Fig. 2.6 Comparison of two signed integers `a` and `b`. The function returns `0` if `a` $==$ `b`, `1` if `a` $>$ `b` and `-1` if `a` $<$ `b`

```
unsigned char isEqual(signed char a,
                      signed char b) {
    return scomp(a,b) == 0;
}

unsigned char isLessThan(signed char a,
                         signed char b) {
    return scomp(a,b) == -1;
}

unsigned char isGreaterThan(signed char a,
                            signed char b) {
    return scomp(a,b) == 1;
}
```

define functions which return true (`1`) when $a = b$, $a < b$, and $a > b$, respectively.

2.5.2 Arithmetic

Let's take a look at the basic arithmetic operations ($+$,$-$,$\times$,$\div$, and %) as they apply to signed numbers. For addition and subtraction we will consider both one's and two's complement negative numbers to show why two's complement is

often preferable. We focus on operations involving negative numbers as operations involving positive numbers follow the techniques described earlier in the chapter for unsigned integers.

Addition and Subtraction. Addition in one's complement notation is nearly identical to unsigned addition with one extra operation should there be a final carry. To see this, let's take a look at adding two negative integers represented in one's complement,

```
    11000000   -63
+   11000010   -61
-------------------   (one's complement)
  1 10000010  -125
    10000011  -124
```

where the carry at the end, shown by the extra `1` on the left, is added back into the result to get the correct answer of `-124`. This adding in of any carry is called the *end-around carry* and is the extra twist necessary when adding one's complement numbers.

The same addition in two's complement notation produces a carry which we ignore,

```
    11000001   -63
+   11000011   -61   (two's complement)
-------------------
  1 10000100  -124
```

since we see that `10000100` is `-124` by making it positive,

```
    01111011
+ 00000001
------------
    01111100  124
```

Addition of a positive and negative number works in the same way for both one's and two's complement numbers. For example, in one's complement we have,

```
+   01111100  124
    11000010  -61   (one's complement)
-------------------
  1 00111110   62
    00111111   63
```

where we have again made use of the end-around carry to give us the correct answer. The two's complement version is similar,

```
+   01111100  124
    11000011  -61   (two's complement)
-------------------
  1 00111111   63
```

where we again ignore the carry and keep only the lower eight bits. Recall, we are giving all examples as signed or unsigned 8-bit numbers. If we were working with 16-bit or 32-bit numbers we would keep that many bits in the answer.

Computers implement subtraction by negation and addition. This allows for only one set of hardware circuits to be used for both operations. With that in mind, subtraction becomes particularly simple. If we want to calculate $124 - 61 = 63$ we actually calculate $124 + (-61) = 63$ which is exactly the example calculated above. For calculation by hand it is helpful to think of subtraction as an actual operation but, as we see here, when done with the appropriate notation for negative numbers, subtraction is really an "illusion" and is nothing more than addition with a negative number.

While addition and subtraction are straightforward, especially with two's complement notation, we have to consider one question: what happens if the result of the operation does not fit in the number of bits we are working with? For our examples, this means that the result does not fit in eight bits taking the sign into account. Let's consider only two's complement numbers. We already saw in the examples above that we could ignore the carry to the 9th bit and saw that the lower eight bits were correct. Is this always the case?

A signed 8-bit two's complement number has a range from `-128` to `127`. If we attempt an operation that falls outside of this range we will not be able to properly represent the answer. We call this condition an *overflow*. How can we detect this? By following two simple rules,

1. *If the sum of two positive numbers is negative, overflow has happened.*
2. *If the sum of two negative numbers is positive, overflow has happened.*

We need not worry about the sum of a positive and negative number because both the positive and negative number are already in the allowed range and it is impossible, because of the difference in sign, for the sum to be outside the allowed range. This is why we ignored the last carry bit when adding -61 to 124. Let's look at cases that prove the rules. If we try to calculate $124 + 124 = 248$ we know we will have trouble because 248 is greater than 127 which is the largest 8-bit positive two's complement number. We get,

```
+ 01111100  124
  01111100  124     (two's complement)
-----------------
  11111000   -8
```

which is clearly a wrong answer. According to our rule for addition of two positive numbers we know overflow has happened because the sign bit, bit 7, is one, indicating a negative answer. Similarly, two large negative numbers added will prove our second rule,

```
+ 10000100  -124
  10000100  -124    (two's complement)
-----------------
  00001000     8
```

where we have ignored the carry to the 9th bit. We see that the result is positive since bit 7 is zero. This proves our second rule and we know that overflow has happened.

Multiplication. We now consider multiplication of signed integers. One approach to signed multiplication would be to make use of the rules for products to track the sign of the result. If we do this, we can make any negative number positive, do unsigned integer multiplication as described above in Sect. 2.3.5, and negate the result if necessary to make it negative. This approach will work for both one's and two's complement numbers. As we recall from school, when multiplying two numbers there are four possible scenarios related to the signs,

1. *positive* × *positive* = *positive*
2. *positive* × *negative* = *negative*
3. *negative* × *positive* = *negative*
4. *negative* × *negative* = *positive*

with this in mind it is simple to extend our `mult2` example from Sect. 2.3.5 to check the signs of the inputs and negate the negative numbers to make them positive before multiplying. Then, the result can be negated to make it negative if the result should be negative. In C this gives us Fig. 2.7

```
 1 signed short signed_mult2(signed char n, signed char m) {
 2     unsigned char i, s=0;
 3     signed short ans=0;
 4
 5     if ((n > 0) && (m < 0)) {
 6         s = 1;
 7         m = -m;
 8     }
 9     if ((n < 0) && (m > 0)) {
10         s = 1;
11         n = -n;
12     }
13     if ((n < 0) && (m < 0)) {
14         n = -n;
15         m = -m;
16     }
17
18     for(i=0; i < 8; i++) {
19         if (m & 1) ans += n << i;
20         m >>= 1;
21     }
22
23     if (s) ans = -ans;
24     return ans;
25 }
```

Fig. 2.7 Unsigned integer multiplication modified for signed numbers

where the main loop in lines 18 through 21 has not changed but before we run it we check the signs of the inputs to see if we need to negate any negative numbers to make them positive. The variable `s` holds the flag to tell us that the answer needs to

be negative. We initially set it to `0` with the assumption that the inputs, `n` and `m`, will both be positive. In lines 5 through 16 we check this assumption. If `n` is positive and `m` is negative, we set `s` in line 6 and make `m` positive. Likewise, in line 9 we check to see if `n` is negative and `m` is positive and make `n` positive if this is the case. We also set the flag in `s` since we know the answer needs to be negative. Lastly, if both `n` and `m` are negative we make them both positive and leave the flag in `s` unset. Then we multiply as before. In line 23 we check to see if the negative flag is set, if so, we make the answer negative before returning it.

In this example we have taken advantage of the fact that the C compiler will properly negate a value as in line 7 regardless of the underlying notation used for negative numbers. We know, however, in practice that this will typically be two's complement. Can we multiply numbers directly in two's complement? Yes, in fact, there are several existing algorithms which to exactly that. Let's consider one of the more popular of them, the Booth algorithm [1] which was developed by Andrew Booth in 1950. A C implementation of this algorithm for multiplying two signed 8-bit integers is given in Fig. 2.8. Let's take a look at what it is doing.

```
 1 signed short multb8(signed char m, signed char r) {
 2     signed int A, S, P;
 3     unsigned char i;
 4
 5     A = m << 9;
 6     S = (-m) << 9;
 7     P = (r & 0xff) << 1;
 8
 9     for(i=0; i < 8; i++) {
10         switch (P & 3) {
11             case 1:    // 01
12                 P += A;
13                 break;
14             case 2:    // 10
15                 P += S;
16                 break;
17             default:   // 11 or 00
18                 break;
19         }
20         P >>= 1;
21     }
22
23     return P>>1;
24 }
```

Fig. 2.8 The Booth algorithm for multiplication of two's complement 8-bit integers

Booth's essential insight was that when we multiply two binary numbers a string of ones can be replaced by a positive and negative sum in the same way that $16 \times 6 = 16 \times (8-2)$ but since we are in binary we can always write any string of ones as the next higher bit minus one. So, we have,

$$00011100 = 00100000 + 000000\text{-}10$$

where we have written a `-1` for a specific bit position to indicate subtraction. This means, if we scan across the multiplicand and see that at bit position i and $i-1$ there is a `0` and `1`, respectively, we can add the multiplier. Similarly, when we see a `1` and `0`, respectively, we can subtract the multiplier. At other times, we neither add nor subtract the multiplier. After each pair of bits, we shift to the right.

In Fig. 2.8 we initialize the algorithm by setting `A` to the multiplier, `m`, shifted nine places to the left, and similarly set `S` to the negative of the multiplier (two's complement form), also shifted nine positions to the left. The product, `P`, is initialized to the multiplicand in `r` but shifted one position to the left. This is done in lines 5 through 7. Why all the shifting? We are multiplying two eight bit signed numbers so the result may have as many as 16 bits, hence using `signed int` for `A`, `S` and `P`. This is the origin of eight of the nine bit positions. The extra bit position for `A` and `S`, and the single extra bit for `P` (line 7), is so that we can look at the last bit position and the one that would come after which we always set to zero. This means that the last bit position, bit 0, and the next, bit -1, could be `1` and `0` to signal the end of a string of ones. We mask the multiplicand, `r`, with `0xFF` to ensure that the sign is not extended when we move from the `signed char` to `signed int` data type. See the next section for a description of sign extension.

The loop in lines 9 through 21 examines the first two bits of `P`, which are the two we are currently considering, and decides what to do based on their values. Our options are,

bit i	bit $i-1$	operation
`0`	`0`	*do nothing*
`0`	`1`	add multiplier to product
`1`	`0`	subtract multiplier from product
`1`	`1`	*do nothing*

which is reflected in the `switch` statement of line 10. The phrase `P & 3` masks off the first two bits of `P`, which is what we want to examine. After the operation, we shift the product (`P`) to the right to examine the next pair of bits. When the loop finishes, we shift `P` once more to the right to remove the extra bit we added at the beginning in line 7. This completes the algorithm and we have the product in `P`, which we return. This algorithm substitutes a starting add and ending subtraction for what might be a long string of additions for each `1` bit in a string of `1` bits. Also, when not adding or subtracting, we simply shift bits. This makes the algorithm particularly efficient.

Sign Extension and Signed Division. Just as we did for multiplication above, we can modify the unsigned integer algorithm for division in Fig. 2.5 to work with signed integers by determining the proper sign of the output, then making all arguments positive and dividing, negating the answer if necessary. However, before we do that, let's take a quick look at sign extension.

Sign Extension. What happens if we take an 8-bit number and make it a 16-bit number? If the number is positive, we simply set the upper eight bits of the new 16-bit number to zero and the lower eight bits to our original number like so,

```
00010110 → 0000000000010110
```

which is pretty straightforward. Now, if we have a negative 8-bit signed integer in two's complement notation we know that the leading bit will be a `1`. If we simply add leading zeros we will get,

```
11011101 → 0000000011011101
```

which is no longer a negative number because the leading bit is now a `0`. To avoid this problem and preserve the value we extend the sign when we form the 16-bit number by making all the new higher bits `1` instead of `0`,

```
11011101 → 1111111111011101
```

which we know is the same value numerically and we can check it by looking at the magnitude of the number. Recall, we convert between positive and negative two's complement by flipping the bits and adding one. So, the 8-bit version becomes,

$$\texttt{11011101} \rightarrow \texttt{00100010 + 1} \rightarrow \texttt{00100011} = 35_{10}$$

and the 16-bit version becomes,

$$\texttt{1111111111011101} \rightarrow \texttt{0000000000100010 + 1} \rightarrow \texttt{0000000000100011} = 35_{10}$$

which means that both bit patterns represent -35 as desired. We intentionally frustrated sign extension in Fig. 2.8 by masking `r` with `0xFF` before assigning it to `P` which was a 32-bit integer.

Signed Division. Figure 2.5 implements unsigned division. If we track the signs properly we can modify it to work with signed integers. Division actually returns two results. The first is the quotient and the second is any remainder. The sign we should use for the quotient is straightforward enough,

Dividend	*Divisor*	*Quotient*
$+$	$+$	$+$
$+$	$-$	$-$
$-$	$+$	$-$
$-$	$-$	$+$

ambiguity arises when we think about what sign to apply to the remainder. It turns out that different programming languages have adopted different conventions. For example, C chooses to make the remainder have the same sign as the dividend while Python gives the remainder the sign of the divisor. Unfortunately, the situation is more complicated still. When dividing negative numbers we are often returning an approximate quotient (unless the remainder is zero) and that approximate quotient has to be rounded in a particular direction. All division algorithms in programming languages satisfy $d = nq + r$ which means that the quotient, q, times the divisor, n, plus the remainder, r, equals the dividend, d. However, we have choices regarding how to set the signs and values of q and r. There are three options,

1. *Round towards zero.* In this case, we select q to be the integer closest to zero that when multiplied by n is less than or equal to d. In this case, if d is negative, r will also be negative. For example, $-33/5 = -6$ with a remainder of -3 so that $-33 = 5(-6) + (-3)$. This is the option used by C.
2. *Round towards negative infinity.* Here we round the quotient down and end up with a remainder that has the same sign as the divisor. In this case, $-33/5 = -7$ with a remainder of 2 giving $-33 = 5(-7) + 2$. This is the option used by Python.
3. *Euclidean definition.* This definition makes the remainder always positive. If $n > 0$ then $q = floor(d/n)$ and if $n < 0$ then $q = ceil(d/n)$. In either case, r is always positive, $0 \leq r < |n|$.

Let's make these definitions more concrete. The table below shows the quotient and remainder for several examples in both C and Python. These will give us an intuitive feel for how these operations work.

Dividend	*Divisor*	*Quotient*	*Remainder*	
33	7	4	5	*C*
		4	5	*Python*
-33	7	-4	-5	*C*
		-5	2	*Python*
33	-7	-4	5	*C*
		-5	-2	*Python*
-33	-7	4	-5	*C*
		4	-5	*Python*

We see that differences only arise when the signs of the dividend and divisor are opposite. It is here that the C choice of rounding towards zero and the Python choice of rounding towards negative infinity come into play. The C choice seems more consistent at first because it always results in quotients and remainders with the same magnitude, only the signs change, but from a mathematical point of view it is less desirable because certain expected operations do not give valid results. For example, to test whether or not an integer is even it is common to check if the remainder is zero or one when dividing by two. If the remainder is zero, the number is even, if one, it is odd. This works in Python for negative integers since `-43 % 2 = 1` as expected, but in C this fails since we get `-43 % 2 = -1` because of the convention to give the remainder the sign of the dividend.

With all of the above in mind, we can now update our `div2` algorithm in Fig. 2.5 to handle signed integers. We show the updated algorithm, now called `signed_div2`, in Fig. 2.9. Let's look at what has changed.

The actual division algorithm in lines 20 through 30 is the same as in Fig. 2.5 since we are still performing unsigned division. Lines 6 through 18 check on which of the arguments, the dividend (`n`) or divisor (`m`) or both, are negative. We use two auxiliary variables, `sign` and `rsign`, to track how we deal with the sign of the answer. If the dividend is negative but the divisor is positive the quotient should be negative so we set `sign` to `1`. If the dividend is positive but the divisor is negative, we also need to make the quotient negative. If both the dividend and divisor are

```
signed char signed_div2(signed char n,
                        signed char m,
                        signed char *r) {
    unsigned char i, sign=0, rsign=0;

    if ((n < 0) && (m > 0)) {
        sign = rsign = 1;
        n = -n;
    }
    if ((n > 0) && (m < 0)) {
        sign = 1;
        m = -m;
    }
    if ((n < 0) && (m < 0)) {
        rsign = 1;
        n = -n;
        m = -m;
    }

    *r = 0;

    for(i=0; i<8; i++) {
        *r = (*r << 1) + ((n & 0x80) != 0);
        n <<= 1;

        if ((*r-m) >= 0) {
            n |= 1;
            *r -= m;
        }
    }

    if (sign) n = -n;
    if (rsign) *r = -*r;

    return n;
}
```

Fig. 2.9 Shift, test, and restore unsigned integer division updated to handle signed integers

negative, the quotient is positive. In all cases we make the dividend and divisor positive after we know how to account for the sign of the quotient. For the remainder, we use the variable `rsign` to decide when to make it negative. The division algorithm itself will make the remainder, in `*r`, positive but in order for our answer to be sound we must sometimes make `*r` negative. When to do this? A sound answer will always satisfy,

$$n = m \times q + r$$

so if the dividend in `n` was negative, we will require the remainder to be negative as well. In this case we follow the C convention.

If we run `signed_div2` on `n = 123` and `m = 4` with all combinations of signs, we get the following output,

n	m	q	r	$m \times q + r$
123	4	30	3	4(30)+3
-123	4	-30	-3	4(-30)-3
123	-4	-30	3	-4(-30)+3
-123	-4	30	-3	-4(30)-3

where the column on the right shows that our choice of sign for the remainder is correct.

In this section we have reviewed implementations of signed arithmetic on integers. In some cases we were able to build directly on existing algorithms for unsigned arithmetic while in some we worked directly in two's complement format. There is no need to talk about signed integer square root since the square root of a negative number is imaginary and we are not yet ready to work with complex numbers.

2.6 Binary-Coded Decimal

Binary-Coded Decimal (BCD) numbers make use of specific bit patterns corresponding to the digits $0 \ldots 9$ in order to store one or two decimal digits in each byte. Storing numbers in this format allows for decimal operations in place of binary and, indeed, some early microprocessors such as the Western Digital 6502 had BCD modes. In this section we describe how to encode numbers in BCD and how to do simple arithmetic with those numbers.

2.6.1 Introduction

As we saw earlier in this chapter, working with numbers expressed in binary can be cumbersome. Binary is the natural base for a computer to use given its construction but humans, with ten fingers and ten toes, generally prefer to work in decimal. One possibility is to encode decimal digits in binary and let the computer work with data in this format. Binary-coded decimal does just this. For our purposes we will work with what is generally called *packed BCD* where we use each nibble of a byte to represent exactly one decimal digit. Historically, other sizes were used, one digit per byte (unpacked) for example, or even other values, typically with early computers. In addition, while any set of distinct bit patterns can be used to represent decimal digits we will use the obvious choice, $0 = 0000, 1 = 0001, \ldots, 9 = 1001$, so there is a direct conversion between a decimal digit and its BCD bit pattern. The remaining six bit patterns, $1010 \ldots 1111$ are not used or allowed in properly formatted BCD numbers. In the previous sections we ignored the difficulties in converting human