# *Appendix C*

# Computation on microprocessors

In the following, we explore a few issues associated with integer and fixed-point computation on microprocessors. We do not discuss floating point computation since floating point operations are rarely resorted to in interfacing in the context of 8-bit microprocessors.

## C.1 Representation of numbers on microprocessors

### *C.1.1 Binary numbers: unsigned integers*

Internally, the microprocessor represents all variables as binary integers, that is, base 2 integers. The integers may be unsigned (i.e., positive numbers) or may be signed (i.e., may be positive or negative). A positive decimal number, say, the four-digit number 3792, makes use of the numbers 0–9 and may be represented as

$$3792 = 3\times10^3 + 7\times10^2 + 9\times10^1 + 2\times10^0. \qquad \text{(C.1)}$$

Parallel to the decimal system, binary or base 2 integers make use of the numbers 0 and 1. An 8-bit unsigned integer, say, 10011011, may be represented as follows:

$$\begin{aligned} 10011011 &= 1\times2^7 + 0\times2^6 + 0\times2^5 + 1\times2^4 + 1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0 \\ &= 128 + 0 + 0 + 16 + 8 + 0 + 2 + 1 = 155. \end{aligned} \qquad \text{(C.2)}$$

The representation in **C.2** also shows how numbers can be converted from one system to the other. For the particular case shown here, the decimal equivalent is 155.

To convert a binary number to a decimal number, it suffices to sum up the products indicated in **(C.2)**. To convert from decimal to binary, we can proceed in two simple ways. A formal method is based on division by 2. The decimal number is divided by 2. If the division is exact, one writes the least significant bit (LSB) as a "0." If not, it is written as a "1" and the quotient is then divided again by 2 until the quotient is zero. Using again the number 3,792, we write

$$
\begin{aligned}
3792/2 &= 1896\text{---}0\\
1892/2 &= 948\text{----}0\\
948/2 &= 474\text{-----}0\\
474/2 &= 237\text{------}0\\
237/2 &= 118\text{------}1\\
118/2 &= 59\text{------}0\\
59/2 &= 29\text{---------}1\\
29/2 &= 14\text{---------}1\\
14/2 &= 7\text{----------}0\\
7/2 &= 3\text{----------}1\\
3/2 &= 1\text{----------}1\\
1/2 &= 0\text{----------}1
\end{aligned}
\qquad \text{(C.3)}
$$

The digital representation is 111011010000 and requires 12 bits.

An often-simpler method is suggested by **(C.3)**. Find the largest power of 2 that fits in the number and subtract it from the decimal number. Set the digit that corresponds to that power to "1" for the most significant bit (MSB). Find the next largest power of 2 that fits in the remainder and subtract it. Set the corresponding digit to "1" and so on until the remainder is zero. All other digits are zero. In the present example, the largest power of 2 is $2^{11} = 2048$. Digit 12 becomes "1" and the remainder is 1,744. Now the largest power that fits in 1,744 is $2^{10} = 1024$. Digit 11 is "1" and the remainder is 720. The next largest power of 2 is $2^9 = 512$. Digit 10 is "1" and the remainder is 208. The next power of 2 is $2^7 = 128$. The remainder is 80. Digit 8 is "1," but since $2^8 = 256$ was not used, digit 9 is "0." Continuing the process, we obtain the representation 111011010000, as above.

### C.1.2 Signed integers

In the decimal system, we use the minus sign to indicate negative numbers (integers or fractions), but in the digital system, there is no negative sign. In the common notation, negative integers are treated as positive integers with the MSB serving as a sign digit. If the MSB is "0," the number is considered positive, whereas if it is "1," the number is negative. Consider, for example, the signed integer 01000101. This is a positive number equivalent to 69. The signed integer 11000101 is negative, equivalent to −59. To understand the notation, we write the two integers as follows:

| 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
|---|---|---|---|---|---|---|---|
| $-2^7$ | $2^6$ | $2^5$ | $2^4$ | $2^3$ | $2^2$ | $2^1$ | $2^0$ |

| 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
|---|---|---|---|---|---|---|---|
| $-2^7$ | $2^6$ | $2^5$ | $2^4$ | $2^3$ | $2^2$ | $2^1$ | $2^0$ |

(C.4)

The first integer is

$$
\begin{aligned}
01000101 &= 0\times(-2^7)+1\times 2^6+0\times 2^5+0\times 2^4+0\times 2^3+1\times 2^2+0\times 2^1+1\times 2^0\\
&= 0+64+0+0+0+4+0+1=69.
\end{aligned}
$$

The second integer is

$$
\begin{aligned}
11000101 &= 1\times(-2^7)+1\times 2^6+0\times 2^5+0\times 2^4+0\times 2^3+1\times 2^2+0\times 2^1+1\times 2^0\\
&= -128+64+0+0+0+4+0+1=-59.
\end{aligned}
$$

Since the sign bit cannot be used as part of the representation of the signed integer, the range of numbers that can be represented is from −128 to +127, or in general, from $-2^{n-1}$ to $2^{n-1}-1$, whereas unsigned numbers range from 0 to 255 or from 0 to $2^n - 1$, where $n$ is the number of digits in the representation.

Negative integers are represented using the 2's complement method as follows:

1. Start with the positive value of the negative number. That is, if we need to find the representation of the number – A, start with A in binary format.

2. Calculate the 1's complement of A. The 1's complement of a binary integer is found by replacing all zeroes with ones and all ones with zeroes, including the sign bit.

3. Add "1" to the 1's complement to get the 2's complement.

For example, suppose we need to write the integer −59. We first write 59 as 00111011. The 1's complement is 11000100. Adding 1 to this gives 11000101. This is clearly a negative number. Its value is −59, as was shown earlier.

In the representation of signed integers, we used a "reserved" sign digit, but in the process, the range of values that can be represented has been seriously reduced. This is particularly problematic with 8-bit integers. To alleviate this problem, microprocessors have adopted a slightly different strategy. Both signed and unsigned numbers use 8 bits (or 16 bits in 16-bit microprocessors) so that the range of integers is the same for signed and unsigned integers. Two extra bits in a different register are used to indicate carry and borrow. When a carry bit is set, it is an indication that an add function has overflowed the register, whereas when the borrow bit is set, an underflow occurred, indicating a negative number. From the point of view of representation of negative numbers, this is the same as using a 9-bit register to represent an 8-bit signed integer.

### C.1.3 Hexadecimal numbers

Binary numbers are particularly useful for computation because the hardware used in microprocessors (and in computers) can represent two states quite easily. The main shortcoming of binary representation outside of computers is that the digital numbers are long. In programming and display, it is more convenient to use higher base numbering schemes that are related to base 2 numbers. Two distinct representations that satisfy this requirement are the octal (base 8) and hexadecimal (base 16) numbering schemes. In microprocessors, the hexadecimal representation is most commonly used. The scheme uses the numbers 0–9 plus A (=10), B (=11), C (=12), D (=13), E (=14), and F (=15). In this system, the LSB is multiplied by $16^0$, the next by $16^1$, then $16^2$, and so on. The number 3,792 can be written as

$$3792 = 14\times16^2 + 13\times16^1 + 0\times16^0.$$

The representation of 3,792 in hexadecimal format is therefore ED0. The subtraction method discussed earlier for binary numbers applies here as well, with the obvious modifications.

## C.2 Integer arithmetic

Since microprocessors were designed for digital control, they can only manipulate binary integers. That means first that integer computation is "natural" in the microprocessor and, second, that computation in any other format must be adapted to the binary integer environment.

Integer computation is exact, that is, there is no loss of accuracy due to round-off errors, as long as it can be done within the number of bits allocated. If, for example, one allocates 8 bits for unsigned integers, the largest value that can be represented is 255. As long as all numbers required for the computation as well as the result are smaller than 255, the result is exact. Allocation of, say, 16 bits allows integers from 0 to $2^{16} - 1 = 65{,}535$.

If negative numbers must be used as well, such as in performing subtraction, one must use signed integers. Since typically the MSB is used for sign indication, a 16-bit signed integer representation will use 15 digits for the integer, allowing representation of integers from $-2^{15}$ to $+2^{15} - 1$, or −32,768 to +32,767 (see, however, **Section C.1.2** on how microprocessors handle negative integers). This also shows the main shortcoming of integer computation—its dynamic range is low, that is, the range of numbers it can represent is small. For this reason, computers use floating point numbers, characterized by a mantissa and an exponent. By using a short mantissa and an exponent, one can describe any number, although numbers are, in general, truncated, hence floating-point calculation is not exact.

### C.2.1 Addition and subtraction of binary integers

The basic arithmetic operation in a microprocessor is the addition of two binary integers. Almost all other mathematical operations must rely on addition, together with logical operations. Notable exceptions are multiplication and division by powers of 2, since that can be done as logic shifting to the left (shifting the bits one location to the left multiplies the number by 2) or to the right (each shift divides the number by 2).

Addition is performed as with decimal numbers except that 1 + 1 produces a carry 1, that is, 1 + 1 = 10 and 1+ 1 + 1 = 11 (i.e., 1 + 1 + carry = 11, whereas 1 + carry = 10). For two 8-bit signed integer digital numbers, A = 00110101 and B = 00111011, we have

$$00110101 + 00111011 = 01110000 \quad (53 + 59 = 112).$$

In this case, all three values are less than 255 and no carry is generated. But suppose that B = 01111011 (123):

$$00110101 + 01111011 = 10110000 \quad (53 + 123 = 176).$$

The result has a "1" where the sign bit should be, but because both integers are positive, the result cannot be negative, hence the resulting integer is properly interpreted as 176.

Suppose now we wish to subtract B = 01111011 (123) from A = 00110101 (53). First we must write the value − B, so we can perform the operation A + (−B), We write

10000100, the 1's complement of B

10000100 + 1 = 10000101, the 2's complement of B.

This equals −123 (i.e., −128 + 4 + 1= −123).

Now we add the 2's complement of B and A:

$$00110101 + 10000101 = 10111010 \quad (53 - 123 = -70).$$

This is clearly a negative number, equal to (−128 + 32 + 16 + 8 + 2 = 70), as can be verified using the decimal numbers. Note that since we perform subtraction, the sign bit cannot be interpreted as a carry, it can only be considered as a sign.

*C.2.2 Multiplication and division*

The principle of multiplication and division of binary numbers follows the same principles as the multiplication of decimal numbers. The simple long multiplication and long division, as used in hand calculation, can be adapted to binary numbers, and in fact it is much easier to do than in decimal arithmetic. However, the term "long" has a particularly clear meaning in binary multiplication and division. Because binary numbers are very long (in comparison with equivalent decimal numbers), the process of long multiplication and division requires a significant number of steps. For this reason, microprocessors and computers perform these operations either in special hardware or using optimized algorithms that are much more efficient than long multiplication or division. However, to see the principles involved, we will look here at the process by emulating hand multiplication/division.

### C.2.2.1 Binary integer multiplication

Multiplication is implemented by a sequence of shifts and adds, operations that do not require any special hardware. Consider the multiplication of two 8-bit unsigned integers A = 10110011 (179) and B = 11010001 (209). The result is 37,411, or 1001001000100011.

```
        10110011 ×
        11010001
        10110011 +
   10110011
   101111100011 +
 10110011
 111000101000011 +
10110011
1001001000100011 .
```

This calculation differs from the common long multiplication in that intermediate sums are computed and instead of multiplying by zero, the multiplicand is shifted one position to the left. The multiplication process requires that the multiplicand be placed in a variable twice as long as itself (16 bits in this case, to allow shifting). The intermediate results are also placed in a 16-bit integer. The multiplier remains unchanged (8 bits). Multiplication is simply a matter of shifting the multiplicand to the left and adding it to the previously calculated intermediate result.

The algorithm is as follows:

1. Place the multiplicand in a 16-bit register (M register).
2. Zero the intermediate result register (I register).

3. If MSB is “1,” add the M and I registers. Place result in the I register.

4. If MSB is “0,” shift the M register 1 bit to the left.

5. Set STEP = 2 and scan the multiplier register starting from bit STEP.

6. If bit # = STEP is “0,” shift the M register 1 bit to the left.

7. If bit # = STEP is “1,” shift the M register 1 bit to the left and add it to the I register.

8. Increment STEP.

9. If STEP = 9, go to STOP. If not, got to (6).

10. STOP. The result is in the I register.

It should be noted that the result is twice as long as the original integers. The algorithm itself requires eight shifts and eight adds and longer word lengths but requires no other type of operation. It is clearly a much slower algorithm than either addition or subtraction. As indicated earlier, this is a “worst case” algorithm and serves to illustrate the process. Actual multiplication algorithms are much more sophisticated than this and require considerably fewer operations.

We have assumed here that the integers are unsigned. If they are, the multiplication is done by first changing any negative integers into their positive equivalent, multiplying the integers without the sign bits, and then calculating the sign of the product from the signs of the two multiplicands. If the product is negative, it is converted to its negative representation as discussed in **Section C.1.2**.

### C.2.2.2 Binary integer division

Long division of unsigned integers is similar to long division of decimal numbers and results in a quotient and remainder. As with multiplication, it is done by a series of right shifts and subtractions (which themselves are done through additions). Consider the division of the dividend A = 11101110 (238) by the divisor B = 00001001 (9). The result expected is 26 (the quotient) with a remainder of 4 since we discuss here integer division ($26 \times 9 + 4 = 238$). It is instructive first to look at the division in **Figure C.1(a)**, as it is done by hand. First, the divisor is aligned with the leftmost bit of the dividend. If the divisor is smaller than the number made of the digits above it, we subtract the divisor from the (four) digit number above it to obtain a remainder, in this case equal to 101. The MSB of the quotient is set to 1. If this is not to be the case, subtraction is not performed and the MSB stays 0. Next, we drop the next digit (fifth digit from the left) of the dividend to the right of the remainder to produce 1011. Now we subtract the divisor from this, producing a remainder equal to 10 and the next digit in the quotient becomes 1 (the quotient now is 11). The sixth digit from the dividend is dropped to the remainder to change it to 101. This is smaller than the divisor. As a consequence, the next digit in the quotient is 0 and no subtraction takes place. The seventh digit is dropped and the remainder becomes 1011. The divisor is subtracted to produce a remainder equal to 10 and the quotient becomes 1101. The last digit is dropped down to the remainder, making it 100. Since this is smaller than the divisor, the quotient becomes 11010 (26) and the remainder is 100 (4), as expected. Now consider **Figure C.1(b)**, which produces the same

result using 8-bit integers. The following algorithm may be used to accomplish this.

1. The dividend (DD register) and the divisor (DR register) are each in an 8-bit register. Create two additional registers: a quotient (Q register) and a remainder (R register).

2. Zero the Q and R registers.

3. Shift the DR register to the left until all leading zeroes clear out and the MSB is "1." Save the number *n* as the number of shifts that was necessary.

4. Subtract the first *n* digits of the DR register from the first *n* digits of the DD register. If the result is negative, set the *n*th digit of the Q register to "0" and disregard the subtraction.

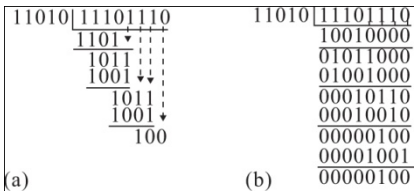



***FIGURE C.1*** *(a) Hand process for long division. (b) Equivalent long division on a microprocessor using 8-bit integers*

5. If the subtraction in (4) results in a positive value, place that in the R register and set the *n*th digit in the Q register to "1."

6. Shift the DR register one position to the right and increment *n* ($n = n + 1$).

7. Set the *n*th digit in the R register equal to the *n*th digit in the DD register.

8. Subtract the DR register from the R register. If the value is negative, set the *n*th bit in the Q register to 0. Disregard the subtraction. If $n = 8$, go to (10). Otherwise, go to (6).

9. If R – DR > 0. set the *n*th bit in the Q register to 1. If $n = 8$, go to (10). Otherwise, go to (6).

10. The quotient is in the Q register, and the remainder is in the R register.

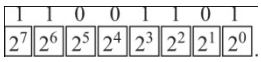
This is a long algorithm that can be implemented in software, but it is an inefficient way of performing division. It does exemplify, however, the fact that division can be performed using only shifts and additions, as indicated previously.

## C.3 Fixed point arithmetic

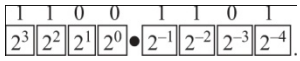
In some instances, one must use fractional numbers. For example, one may need to find a ratio of two integers or scale an input voltage to a microprocessor by a noninteger value or add a fixed, offset value to a result. In microprocessors, this is usually done using fixed point arithmetic, as it requires fewer resources than floating point arithmetic using established routines, available from various sources.

The basic idea of fixed point arithmetic is to treat the integer and fraction parts of a single integer with the radix point placed implicitly at a fixed location within

the integer. By doing so, one has the ability to represent fractional numbers while still operating only on integers. Consider the 8-bit unsigned number shown in the following with the weights associated with each digit:



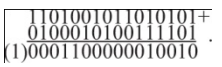

The digital number represents the decimal value 205. We could just as easily say that it represents the value 205.0, but by doing so, we placed a decimal point (in general we will call this a radix point) at the end of the number. Suppose now we move the decimal point one position to the left. We obtain the value 20.5. In decimal format, this can be written as $2 \times 10^1 + 0 \times 10^0 + 5 \times 10^{-1}$. In effect, the number has been divided by 10, or more appropriately, scaled by $10^{-1}$. Similarly, if we place a radix point, say, between the fourth and fifth digits in the binary number 11001101, we obtain 1100.1101. Following the example for the decimal number, the representation becomes




The digital number represents the value 12.8125 and is in fact $205/2^4 = 205 \times 2^{-4} = 12.8125$. For this reason, fixed point numbers are often called scaled integers. Note that the digital number itself has not changed but the weights of the digits have been scaled by $10^{-4}$.

The problem with this representation is that the integer can only vary between 0000 and 1111 (or 0 and 15), whereas the fraction can vary between 0000 and 1111 (or 0 and 0.9375 with a resolution of 1/16). If we wish to represent signed numbers as fixed point numbers, the integer in the previous example is reduced by 1 bit and will only be able to represent numbers between 0 and 7 (000 and 111). Clearly for the method to be useful, a larger number of bits are needed. In an 8-bit microprocessor, the natural choice is to use 8 bits for the integer and 8 bits for the fraction, allowing unsigned numbers between 0 and 255.99609575 and signed numbers from −127.99609575 and 127.99609575. Of course, one can use, say, 12 bits for the fraction and 4 bits for the integer or any combination that fits the application at hand. However, most routines supplied by manufacturers of microprocessors use 8 (or 16) bits for each part. Any other combination might need to be written by the user, usually as a subroutine to be called when needed.

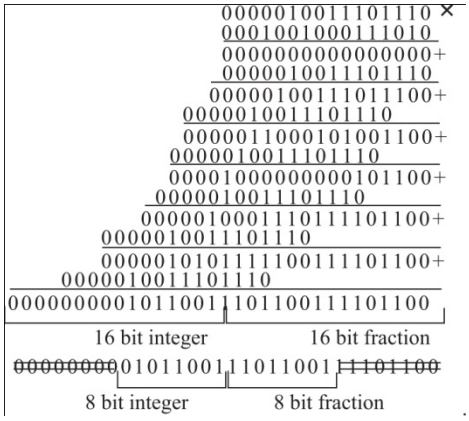
All operations on fixed point numbers are done as with any integer numbers, with the obvious allowance for the radix point. Addition and subtraction are identical to that of signed or unsigned integers. For example, the sum of the two unsigned fixed point numbers A = 11010010.11010101 (210.83203125) and B = 01000101.00111101 (69.23828125) is




The result is 00011000.00010010 with a carry 1. In decimal representation, the result is 24.0703125 and a carry equivalent to 256. The decimal result is 280.073125 = 24.0703125 + 256. Obviously, the result requires nine digits for the integer rather than eight, but the summation is correct. The main point to notice is that the two numbers are treated as any two integers and that the radix point does not affect the operations performed to obtain the result. Signed fixed point numbers are handled the same as signed integers.

Multiplication and division of fixed-point numbers also follow the same process as for integers, but the radix point must be adjusted. For example,

multiplication of two 16-bit fixed point numbers, each with 8 bits for the integer and 8 bits for the fraction, results in a 16-bit integer and a 16-bit fraction that must be truncated to a total of 16 bits. Assuming the integer can fit in 8 bits, there will be no loss in accuracy, or one would have to use a larger number of bits. As an example, suppose the product of A = 00000100.11101110 (4.9296875) and B = 00010010.00111010 (18.2265625) is required. The multiplication follows the same rules as decimal multiplication:




The result is

$$c = 0000000001011001 \cdot 1101100111101100.$$

The multiplication results in 16 bits for the fraction and 16 bits for the integer. The eight LSB digits must be removed from the fraction and the eight MSB digits removed from the integer since only 8 bits are available for each representation even though the internal multiplication is handled with 16 bits for each part. Therefore, the result is

$$c = 01011001 \cdot 11011001 = 89.84765625.$$

Because of the truncation of the fraction this is not exact, the correct result should be 89.8512573242 (the error is about 0.8%). Note that truncation of the fraction results in a loss of accuracy but the truncation of the integer does not, as long as the product can fit in 8 bits. If it does not, then the result is wrong, since the MSB will be truncated.