CS212 GBA
All About Math:
Math For Systems Programming
Spring 2008

1 Numeral Systems

There exist many ways of counting. The one way we are most familiar with is ***decimal systems*** which is counting in **base 10**. Because of their electronic structure, computers today actually use a ***binary system*** to represent numbers. In order to understand system programming, we not only need to be familiar with how computers count in **base 2**, we also need to learn how ***hexadecimal numbers*** work, which are **base 16** numbers and are very useful in programming with GBA.

Types of Numeral Systems

Decimal. We are all very familiar with base 10 arithmatics, however, we want to show you explicitly how to count in base 10. Let us take a number 2759. It can be written as

$$\begin{aligned} 2975 &= 2 \cdot 1000 + 7 \cdot 100 + 5 \cdot 10 + 9 \cdot 1 \\ &= 2 \cdot 10^2 + 7 \cdot 10^1 + 5 \cdot 10^1 + 9 \cdot 10^0 \end{aligned}$$

Notice that each digit represents a power of 10. This should be easy enough. But before we proceed, let us take a look at the general form of a whole number in a base b system:

$$a_n a_{n-1} ... a_1 a_0 = a_n \cdot b^n + a_{n-1} \cdot b^{n-1} + ... + a_1 \cdot b^1 + a_0 \cdot b^0$$

Binary. Since the binary numeral system has a base 2, binary numbers only consist of digits of 0 and 1. Using the idea above, a binary number 10010111_2 can be written as

$$\begin{aligned} 10010111_2 &= 1 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 \\ &= 151_{10} \end{aligned}$$

Note that we use subscript to indicate the base.

Hexadecimal. Hexadecimal numbers are counted in base 16. As there are not as many roman numerals, we use letters $A - F$ to represent the numbers $10 - 15$. We take a hexadecimal number $4E2B$ as an example:

$$\begin{aligned} 4E2B_16 &= 4 \cdot 16^3 + 14 \cdot 16^2 + 2 \cdot 16^1 + 11 \cdot 16^0 \\ &= 20011_{10} \end{aligned}$$

Generally, we add a prefix `0x` to a number to indicate that it is a hex (i.e. `0x4E2B`). Another notation convention is to use capital `A - F` instead of lower cases.

Examples

Some examples of conversion between numeral systems.

2 Bitwise Operators

Bitwise operators operate, as they are named, on numbers bit by bit. These operators include ***AND, OR, NOT,*** and ***exclusive OR***. They are very powerful as we can directly manipulate bits in a number and become very useful if we want to program in a lower level language like **C**. Bitwise operations also have a great advantage in speed (especially in GBA) compared to the other arithmetic operations we normally use.

Types of Operators

bitwise NOT: "`~`". Bitwise NOT takes in a number and negates each bit. For example:

```
 ~10101101
= 01010010
```

bitwise AND: "`&`". Bitwise AND takes in two operands and performs logical AND operation (see **Table 1**) on each pair of corresponding bits on the operands. We usually use bitwise AND to perform ***bit masking***, which are very useful in extracting values on certain bits. We will discuss bitmask in more detail later. Here is an example of bitwise AND:

```
  10101101
 &00011110
= 00001100
```

bitwise OR: "`|`". Bitwise OR takes in two operands and performs logical OR operations (see **Table 1**) on each pair of corresponding bits on the operands. It is very useful when we want to set certain bits in a number. We will be using bitwise OR a lot when we need to set flags on registers. We will also discuss this in detail later. An example for bitwise OR:

```
  10101101
 |00011110
= 10111111
```

bitwise XOR: "`^`". Bitwise XOR takes in two operands and performs logical XOR operations (see **Table 1**) on each pair of corresponding bits on the operands. Bitwise XOR can be used to set a register to zero and it is particularly advantageous and useful in assembly languages. However, we will not be using this operator as much as the other three listed above. An example of bitwise XOR:

```
  10101101
 ^00011110
= 10110011
```

Caution: Do not confuse bitwise logical operators with other logical operators (logical NOT: `!`, logical AND: `&&`,and logical OR: `||`). The latter treats their operands as boolean values, whereas the former performs the same operations ***bit by bit***.

X	NOT **X**	A	B	**A** AND **B**	**A** OR **B**	**A** XOR **B**
1	0	0	0	0	0	0
0	1	0	1	0	1	1
		1	0	0	1	1
		1	1	1	1	0

Table 1: Truth Table for Logical Operators

3 Bit Shifting

Fact: GBA is very bad at division and multiplication. Generally, it takes GBA a very long time to do division and multiplication. They are very harmful in terms of performance and efficiency, so we want to avoid them as much as possible. Therefore, in order to make our way to have some functionalities similar to division and multiplication, we have to use the alternative: ***bit shifting***.

There are two bit shifting operators: ***left shift*** (`<<`), and ***right shift*** (`>>`). Bit shifting operators do exactly as they sound; they shift bits of a number to left or right. For example, the expression `00001101 << 3` shifts the binary number `00001101` leftward by 3 bits. The result of this expression is `01101000`. Notice that the left most 3 bits are discarded after the shifting and the right most 3 bits are filled in with `0`. Similarly, the expression `10101101 >> 3` shifts the binary number `10101101` rightward by 3 bits. The result is `00010101`. Notice that when we shift a number `X` leftward by `n` bits, we are actually doing a multiplication $X \cdot 2^n$, and when we shift it to the right by `n` bits, we are performing a division $\frac{X}{2^n}$, or $X \cdot 2^{-n}$ with decimals truncated. e.g.

$$\begin{aligned} 00001101_2 &= 13 \\ 01101000_2 &= 104 = 13 \cdot 8 = 13 \cdot 2^3 \\ 10101101_2 &= 173 \\ 00010101_2 &= 21 \qquad (173 \div 8 = 21.625) \end{aligned}$$

Because numbers in computers have a fixed number of available bits, if we shifted a number by `n` bits in either direction and `n` is greater than or equal to the number of available bits, we would just get `0` as the result. So we have to be careful with ***overflow*** (bits getting discarded when shifted to the left) and ***underflow*** (bits getting truncated when shifted to the right) if we want to bit shift with a large number.

4 Bit Fields and Bit Masking

A bit field is structure used to store flags in certain bits of an integer. Using this data structure, we can compactly store multiple information in one register. For example, if we have two boolean values `A` and `B` and a number `C` that can only take on values `0,1,2,3`, we can store them to a 4-bit integer `X` by allocating the bits. We can specify the bit field of this register `X` as the following:

bits	3	2	1	0
data	A	B	C	

We can use bitwise operators to set, reset, and test the bits in this paricular register.

Setting the bits. Say we want to set this register with `A = 1, B = 0, C = 2`. Using bitwise OR and bit shifting operators, we can easily achieve this in an expression of one line:

```
X = 0x1 << 3 | 0x0 << 2 | 0x2
```

`X` now has the binary value of `1010`.

Bit masking: reset, extract, and test. As we mentioned before, bitwise AND can be used to perform bit masking. The logical AND operation has the property that it returns true if and only there both operands are true, and returns false otherwise. Using this property, we can easily reset and extract information from a resister. Following the previous example, we can reset `C` (bits 0-1) with the expression `X = X & 0xC` . The bitwise AND operation involved is:

```
  1010
 &1100
= 1000
```

Notice that bits 0 and 1 are reset to 0, because `Y&0 = 0` (where `Y` can be anything); and the values in 2 and 3 are preserved because `Y&1 = Y` (where `Y` can be anything).
Similarly, we can extract the value of `C` (bits 0-1) and store it to `Z` with the following expression:
`Z = X & 0x3` . The bitwise AND operation involved is:

```
  1010
 &0011
= 0010
```

We can also use bitwise AND to test whether a particular bit holds a certain value. For example, we can test if `A` has the value `1` with the expression `A & 0x8` . This expression is **false** only if the resulting value is `0` and is **true** otherwise. The bitwise operation involved is:

```
  1010
 &1000
= 1000
```

Since `1000` is not zero, we know the bit that contains the value of `A` must have been set.

5 One's and Two's Complement

People may be wondering how computer represent negative numbers in binary system. In fact, there are many ways to do so, and two of the common ones are ***one's complement*** and ***two's complement***. The latter is the most widely used of all.

One's Complement

We usually denote a negative number with a negative sign in the front of the number. However, binary numbers only consist of 0's and 1's. So we add a ***sign bit*** in the front of the number to indicated the sign. In one's complement, we use `0` to indicate positive numbers and `1` for negative numbers. For example, a 4-bit signed integer `0010` represents `2` and `1010` represents `-2` (bit 3 is the sign bit) However, problems arise when it comes to the number zero. For example, for a 4-bit signed integer, `0000` and `1000` both represent zero. It turns out that in this system, arithematic operations are not as simple as we might expect. Some of the older computers use one's complement as their arithmetic representation, however it is not widely used anymore.

Two's Complement

Two's complement has turned out to be much nicer than one's complement in binary arithmetics. Similar to one's complement, two's complement uses the **most significant bit** as the sign bit, but there is only one representation of zero in this system, which is all zero (e.g. `0000` for 4-bit integers). The negative numbers start from all one's (-1) to one followed by zeros (the smallest integer). (e.g. `1111` represents -1 and `1000` is -7.) To negate a number, we simply invert all its bits and add 1 to the result. For example, to negate `0110` (6), we first invert its bits(`1001`) and then add 1 to get `1010`. The following table shows the two's complement intepretation of all possible values of a 4-bit binary number. Notice that the range is $[-8,7]$ instead of $[-7,7]$

decimal value	binary value
7	0111
6	0110
5	0101
4	0100
3	0011
2	0010
1	0001
-1	1111
-2	1110
-3	1101
-4	1100
-5	1011
-6	1010
-7	1001
-8	1000

6 Additional Information

Number Systems course module at Virginia Tech
`http://courses.cs.vt.edu/ csonline/NumberSystems/Lessons/BinaryNumbers/index.html`
Bitwise Operations and Bit Shifting at Wikipedia.org

http://en.wikipedia.org/wiki/Bitwise_operations

Bit Fields at Wikipedia.org

http://en.wikipedia.org/wiki/Bit-field

Signed Number Representations at Wikipedia.org

http://en.wikipedia.org/wiki/Signed_number_representations