CECS 225 PRECISION ARITHMETIC

Adding binary numbers or adding hex numbers is a very simple task. The process is very similar to the longhand addition of decimal numbers. As with decimal numbers, you start by adding the bits (digits) one column, or place weight, at a time, from right to left.

Unsigned Binary Addition

Unlike decimal addition, there is little to memorize in the way of rules for the addition of binary bits:

0 + 0 = 0
1 + 0 = 1
0 + 1 = 1
1 + 1 = 10
1 + 1 + 1 = 11

Just as with decimal addition, when the sum in one column is a two-bit (two-digit) number, the least significant figure is written as part of the total sum and the most significant figure is "carried" to the next left column. Consider the following examples:

```
                    11  1             11
  01001101        01001001        01000111
+ 00010010      + 00011001      + 00010110
----------      ----------      ----------
  01011111        01100010        01011101
```

The addition problem on the left did not require any bits to be carried, since the sum of bits in each column was either 1 or 0, not 10 or 11. In the other two problems, there definitely were bits to be carried, but the process of addition is still quite simple.

Unsigned Overflow

In computing there can be restrictions with respect to the size of data that can be represented. For example, in 8-bit unsigned data the largest binary quantity is 11111111 which is equivalent to decimal value 255. What happens if the result of addition produces a value larger than 255? That is when unsigned overflow occurs. Since the quantity is constrained to 8-bits a garbage result may be produced or an exception will be triggered. When doing binary addition, unsigned overflow can be identified by if there is a carry of 1 out of the MSB. Consider the following example:

```
 111100000  <--carry in bits
  11110010  <--addend -->    242
+ 10110000  <--augend -->  + 176
------------------         -------
  10100010  <--sum    -->    162
  11110000  <--carry out bits
```

Unsigned Overflow! carry out of MSB equals 1

As can be seen when the 8-bit sum is converted to decimal, it's value is incorrect. This example used 8-bits of precision but 9 bits are needed to represent the sum.

The effects of unsigned overflow in 8-bit data can be observed in the following C program.

```c
#include <stdio.h>

int main()
{
    unsigned char a, b, c;
    a = 150; //use a value between 0 and 255
    b = 150; //use a value between 0 and 255
    c = a + b;
    printf("a = %d, b = %d\n",a,b);
    printf("a + b = %d\n",c);
    if(c != a + b)
        printf("unsigned overflow!\n");

    return 0;
}
```

Signed Binary Addition

Instead of doing binary subtraction, addition with signed numbers perform operations equivalent to subtraction. Consider a decimal examples:

5 – 2 = 3 equivalently 5 + -2 = 3
60 – 91 = -31 equivalently 60 + -91 = -31

The same concept applies in the context of binary. Negative values must be 2's complemented. The previous examples will be reworked in binary:

5 = 00000101 -2 = -00000010 →2's complement → 11111110

```
 111111000 ←Carries
  00000101 ←addend→   5
+ 11111110 ←augend→  -2
----------           ---
  00000011 ←sum→      3
```

As it can be seen the correct result has been produced. There is a carry out of the MSB which is discarded, the rule as to why this is acceptable will be analyzed in the next section when Signed Overflow is discussed. And next, 60 – 91 will be performed in binary

```
 001111000  ← carry in bits
  00111100  ← addend            60
+ 10100101  ← augend           -91
-----------                    ---
  11100001  ← sum              -31
  00111100  ← carry out bits
```

Signed Overflow

Signed arithmetic done on 8-bit data is constrained to producing a result that can be accurately represented in 8-bits. For signed 8-bit quantities, values that can be accurately represented must be greater than -128 and less than 127. **When performing Signed binary addition a simple rule to check for signed overflow is that the carry into the MSB must equal the carry out of the MSB**. If this condition is violated then Signed Overflow has occurred and the sum is garbage. Observe the following example where -100 – 100 is performed:

```
 100111000  ←carry in bits
  10011100  ←addend              -100
+ 10011100  ←augend            + -100
------------------             -------
  00111000  ←sum→                  56
  10011100  <--carry out bits
```

Signed Overflow! carry out of MSB does not equal carry in to MSB
The sum should be -200 but 8-bit data size lacks the precision to represent this value. Therefore Signed overflow has occurred.

The effects of signed overflow can be observed in the following C program:

```c
#include <stdio.h>

int main()
{
    signed char a, b, c;
    a = -100; //use a value between -128 and 127
    b = -100; //use a value between -128 and 127
    c = a + b;
    printf("a = %d, b = %d\n",a,b);
    printf("a + b = %d\n",c);
    if(c != a + b)
        printf("signed overflow!\n");

    return 0;
}
```

Data Sizes

Byte = 8-bits. In many of the examples we have worked with, the data size has been 8-bits. This quantity of bits is referred to as a **Byte**. Therefore:
1 Byte → 8-bits
2 Bytes → 16-bits
4 Bytes → 32-bits
N Bytes → N*8-bits

Identification of data Bytes is important for this course. Periodically we will also be referring to other small data quantities as well. Some of those are outlined on the next page.

There are other designators:

Nibble = 4-bits
Each Hex digit can represent one Nibble of data.
In the following value 0xB7:

- B is the high order nibble
- 7 is the low order nibble

The terminology may be silly but you can think of a nibble as a little Byte

In the 16-bit data quantity 1010100110000111

- 10101001 is the high order byte
- 10000111 is the low order byte

Larger data size quantifications will be explored later.