4CS015 – Workshop #1

Introduction

Nice, easy workshop to get us started. This week we covered the rules of number systems and the representation of positive *and* negative numbers in binary. This workshop will reinforce these concepts and push you just a little bit further, to cover addition in binary.

Once you've got your head around this, you'll be in a better position to understand how we manipulate the signals within the components of a computer.

Onwards!

Stuff to Remember

As we're going to be playing around with a few different number systems this semester, we will use a 'subscript'(this is a subscript) to identify which base we are working in. For example 25_{10} is 25 in decimal, whereas 25_8 is a value in octal (which is 21_{10}. Obviously).

Binary values should be assumed to be simple magnitudes (neither positive or negative), unless otherwise stated.

Converting *From* Decimal *To* Binary

There are two methods for converting from decimal to binary. The correct way (correct in that, as with the other rules covered in the first lecture, this method can be applied to all number systems) is easiest to understand when shown with an example:

Let us convert 118_{10} into binary:

$\frac{118}{2} = 59\ remainder\ 0$ (This will become the LEAST SIGNIFICANT BIT)

$\frac{59}{2} = 29\ remainder\ 1$

$\frac{29}{2} = 14\ remainder\ 1$

$\frac{14}{2} = 7\ remainder\ 0$

$\frac{7}{2} = 3\ remainder\ 1$

$\frac{3}{2} = 1\ remainder\ 1$

$\frac{1}{2} = 0\ remainder\ 1$ (This will become the MOST SIGNIFICANT BIT)

118_{10} = 1110110_2 (Note that I have used the blue and red to highlight which is MSB and which is LSB).

Breaking that example down.

1. Start by dividing the decimal value by 2 (the base we want to convert ***to***).
2. The *remainder* (which will always be a '1' or '0', is used to build up our binary version).
3. Divide the ***result*** of the previous division by 2.
4. Go to step 2 until the result is 0.

Some find it easier to convert from decimal to binary simply by inspection. You can simply 'fill in' the columns. You are essentially doing the same thing, but you may find it easier to look at it like this.

Same example, 118_{10} into binary. We know the values of each column, they follow the same rules as any number system:

x^7	x^6	x^5	x^4	x^3	x^2	x^1	x^0
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	8	4	2	1

We can see that 118 is smaller than **128**, so a '0' goes in the 128 column.

128	64	32	16	8	4	2	1
0	*x*	*x*	*x*	*x*	*x*	*x*	*x*

118 is larger than **64**, so we stick a '1' in that column and subtract **64** from 118.

128	64	32	16	8	4	2	1
0	**1**	*x*	*x*	*x*	*x*	*x*	*x*

118 – 64 = 54. 54 is larger than **32**, so we stick a '1' in that column and subtract **32** from 54.

128	64	32	16	8	4	2	1
0	**1**	**1**	*x*	*x*	*x*	*x*	*x*

54 – 32 = 22. 22 is larger than **16**, so we stick a '1' in that column and subtract **16** from 22.

128	64	32	16	8	4	2	1
0	**1**	**1**	**1**	*x*	*x*	*x*	*x*

22 – 16 = 6. 6 is smaller than **8**, so we stick a '0' in that column and **DON'T** subtract **8** from 6.

128	64	32	16	8	4	2	1
0	**1**	**1**	**1**	**0**	*x*	*x*	*x*

6 is larger than **4**, so we stick a '1' in that column and subtract **4** from 6.

128	64	32	16	8	4	2	1
0	**1**	**1**	**1**	**0**	**1**	*x*	*x*

6 – 4 = 2. 2 is the same as **2**, so we stick a '1' in that column and subtract **2** from 2.

128	64	32	16	8	4	2	1
0	**1**	**1**	**1**	**0**	**1**	**1**	*x*

2 – 2 = 0. 0 is smaller than **1**, so we stick a '0' in that column and stop because we're finished.

128	64	32	16	8	4	2	1
0	**1**	**1**	**1**	**0**	**1**	**1**	**0**

Remember that we can always check that our conversion is correct by going the other way, *FROM* binary *TO* decimal. This is simply multiplying the value in each column by the 'weight' of that column and adding them up:

0*128 + 1*64 + 1*32 + 1*16 + 0*8 + 1*4 + 1*2 + 0*1 = 118! Hurray!

Exercise 1

Convert the following decimal values to binary.

a. 64

b. 129

c. 255

d. 256

e. 99

Signed Magnitude

As covered in the lecture, when working with decimal, we introduce an 11th symbol, '-', to denote a negative number. This is hard for us to translate into something that we can use in hardware, so we employ a couple of different methods (after all, as long as we all know the same set of rules, we can do whatever we want!). The first and simpler method is signed magnitude. We simply associate the Most Significant Bit (the bit on the left hand side) with the symbols '+' and '-'. If the MSB is a '0', we treat the number as a positive and if the number is a '1', we treat the number as a negative. The rest of the columns make up the magnitude of the number. As an example:

00000001 = +1, because

SIGN	64	32	16	8	4	2	1
0	**0**	**0**	**0**	**0**	**0**	**0**	**1**

While 10000010 = -2, because

SIGN	64	32	16	8	4	2	1
1	**0**	**0**	**0**	**0**	**0**	**1**	**0**

You can employ any of the methods discussed above to convert between the bases, but note that we stick to an 8-bit word.

Exercise 2

a. What is the biggest negative number that can be represented by an 8-bit word? Give your answer in both signed magnitude AND decimal.

b. What is the biggest positive number that can be represented? Give your answer in both signed magnitude AND decimal.

c. Convert -68_{10} into signed magnitude.

d. Convert the signed magnitude value 01110101 into decimal.

e. Convert 0_{10} into signed magnitude.

f. Convert the signed magnitude value 10000000 into decimal.

Questions 2e and 2f above prove what we've said before; that signed magnitude wastes, nay, *squanders* a valuable combination of bits. There are 256 different combinations of 8 zeros and ones, so there should be 256 unique numbers represented. However, 00000000_2 and 10000000_2 both mean the same thing! A better solution to the problem of representing negative numbers in binary lies in 2's Complement. But before we head there, let's consider addition of binary numbers.

Addition in Binary

Addition in any number system is the same. But you've probably forgotten the rules as it has become second nature. Taking decimal as an example, we write one of the numbers to be added under the other one, lining up the column weights (The ones column, the tens column, the hundreds column etc). Starting from the Least Significant Position, we then add the numbers in a column. If the total is larger than the maximum allowed value in that column, we 'carry' a portion over to the next column.

```
 1 5
+1 6
 ---
   1
 ---
 1
```

In the example above, the total in the first column (11) is too large to fit in the 'ones' column (the largest value possible would be 9), so the 'tens' part of the total is 'carried' over to the next ('tens') column. The process continues up the chain (if the total in the 'tens' column was more than 9, the 'hundreds' portion would be 'carried' over to the 'hundreds' column for the next operation and so on).

I know that this all seems very obvious, but if you keep the process in your mind for decimal, you'll have no problems in binary, or octal, or hexadecimal for that matter.

So, addition in binary.

```
 0 0 0 1        0 0 0 1        0 0 0 1        0 0 0 1
+0 0 1 1   →    0 0 1 1   →    0 0 1 1   →    0 0 1 1
 -------        -------        -------        -------
   0              0 0            1 0 0          0 1 0 0
 -------        -------        -------        -------
   1              1
```

It's easy for us to check too. The top number is 1_{10} and the bottom number is 3_{10}, which would make a total of 4_{10}, and 0100_2 is 4_{10}!

Exercise 3, Binary Addition

Perform the following additions in binary. Show your working.

a. $110_{10} + 19_{10}$

b. $63_{10} + 78_{10}$

c. $50_{10} + 83_{10}$

2's Complement

Right. Now that we've got that out of the way, let's return to the second method of representing negative numbers in binary. There are multiple reasons for choosing 2's complement over simple signed magnitude. For now, we will concentrate on three of them. First, 2's complement exhibits the same quality as signed magnitude; the MSB defines whether the number is positive or negative. Second, we end up with 256 unique numbers (there is no positive and negative zero issue). Third, we can use addition to perform subtraction (i.e. We can *add* a negative number, rather than *subtract* a positive number), meaning we can use the same hardware design for both operations within our computer. Something we will look at in more detail in the coming weeks.

The rules of 2's complement are quite simple:

1. If the number is positive, convert it to binary and do nothing
2. If the number is negative:
    a. Convert the magnitude into binary
    b. Invert (make all the zeros into ones and *vice versa*).
    c. Add one to the inverted binary number

That's it! Going the other way, converting 2's complement binary to decimal:

3. If the number has a '0' as the MSB, it is positive and can be converted directly to decimal
4. If the MSB is '1', the number is a negative:
    a. Invert
    b. Add one to the inverted value
    c. Convert the result to decimal
    d. Put a '-' sign in front of the converted value

See? Not hard!

A few examples:

Example 1, +7.

1. Number is positive, so convert to binary and leave it.

    $7_{10} = 00000111_2$

Example 2, -6.

2. Number is negative
    a. Convert the '6' to binary 00000110
    b. Invert 11111001

c. Add 1

```
1 1 1 1 1 0 0 1
0 0 0 0 0 0 0 1
1 1 1 1 1 0 1 0
          1
```

d. $-6_{10} = 11111010_2$

Example 3, 00000111_2

3. MSB is a '0', so convert as normal.

   $00000111_2 = 7_{10}$

Example 4, 11111010_2

4. MSB is a '1', so number is negative

   a. Invert 00000101

   b. Add 1

```
0 0 0 0 0 1 0 1
0 0 0 0 0 0 0 1
0 0 0 0 0 1 1 0
          1
```

   c. Convert to decimal: $00000110_2 = 6_{10}$

   d. Put a '-' in front: -6

The only other thing to remember is that any 'carry' that goes past the 8th bit (i.e. would be carried in to the 9th column) is ignored.

Exercise 4

All binary values are 2's Complement

1. Convert 11111111_2 to decimal

2. Convert 01111111_2 to decimal

3. Convert -63_{10} to 2's complement

4. Convert 111_{10} to 2's complement

5. Convert 10101010_2 to decimal

Binary Subtraction

As stated earlier, one of the biggest advantages of 2's complement is that we can *add* a negative number to a positive number. Given that 63 – 63 = 63 + (-63). So, given a subtraction operation, like 10 – 5, we do the following:

Convert 5 to 2's complement -5:

5	00000101	
Invert	11111010	
Add 1	11111011	

Add 2's complement -5 to 10

10	00001010	
-5	11111011	
Add together	00000101	*remember that any 'carry' that continues past bit 8 is ignored. CHECK THIS RESULT. IS IT CORRECT?*

Exercise 5

Perform the following subtractions using 2's complement arithmetic

a. 100 – 35

b. 18 – 6

c. 15 – 23

d. 255 – 11

e. -10 – 63