Problem Set 2

Justin Ely

605.204.81.FA15 Computer Organization

15 September, 2015

3.1)

$5ED4 - 07A4 = 5730$.
The subtraction here was straightforward, as there were no carries. In base 16:

```
4-4=0
D-A=3
E-7=7
5-0=5
```

3.2)

5	E	D	4
0101	1110	1101	0100
0	7	A	4
0000	0111	1010	0100

Using the left-most bit as the sign, this mathematical operation is subtracting a small positive number from a larger positive number. There will be no sign change and no underflow, and the HEX value will still end up being 5730.

3.3)

5	E	D	4
0101	1110	1101	0100

Hex is attractive because it's easier for humans to read and remember then straight binary. It is also convenient because most memory sizes are multiples of 4, which make representation by hex straightforward.

3.4)

$4365 - 3412 = 0753$.
The first two (from right) columns are straightforward to subtract, with no borrows. The third requires a borrow from the fourth column.

```
5-2=3
6-1=5
(initial) 3-4 -> (after carry from next) 11-4=7
(initial) 4-3 -> (after carry from previous) 3-3=0
```

3.5)

4	3	6	5
100	011	110	101

3	4	1	2
011	100	001	010

Using 2's complement to change to addition:

$$\begin{array}{r} 100011110101 \\ \hline +\ 100011110110 \\ \hline\hline =\ 000111101011 \end{array}$$

which evaluates to 4753 in Octal.

3.6)

Unsigned, 8-bit integers can hold values from 0-255 ($2^8 - 1$). 185 - 122 is 63, well within this range, and would cause neither overflow nor underflow.

3.9)

Signed, 8-bit, integers have a range of -128 to 127. Both 151 and 214 would exceed this maximum range, and cause overflow. However, saturating arithmetic means to simply replace the overflow value with the max of the range. Therefore:

$151 + 214 = 127 + 127 = 127$

3.10)

Similar to 3.9 above, saturation of the decimal values means they would be replaced by the maximum signed value:

$151 - 214 = 127 - 127 = 0$

3.11)

Unsigned 8-bit integers can represent values from 0-255, so this value would also saturate at the highest value.

$151 + 214 = 255$

3.32)

The first two operands have the same exponent, and they fit into the 10bit adder, so addition is straightforward without need for guard, round, or sticky bits.
$3.984375E_{10}^{-1} = 11.111111$
$3.4375E_{10}^{-1} = 11.01110$

$$\begin{array}{r} 11.111111 \\ \hline +\ 11.011100 \\ \hline\hline = 111.011011 \end{array}$$

Which evaluates to $7.421875E_{10}^{-1}$ as one would expect.

The second evaluation is more complicated. After equalizing the exponents:
$.7421875_{10} = .10111110$
$1771_{10} = 11011101011.$

$$\begin{array}{r} 11011101011.000 \\ \hline +\ 00000000000.101 \\ \hline\hline = 11011101011.101 \end{array}$$

The guard, round, and sticky bits give a value greater than .5, so the final value must be rounded up to: $11011101100 = 1772_{10}$. This arithmetic was confirmed using float16 precision in a python interpreter, but using a higher precision (32-bit) calculator yields 1771.7421875_{10} for the same operations.

3.33)

After equalizing exponents of the operands:
$1771_{10} = 11011101011.$
$.34375_{10} = .010110$

$$\begin{array}{r} 11011101011.000 \\ \hline +\ 00000000000.010 \\ \hline\hline = 11011101011.010 \end{array}$$

The guard, round, and sticky bits give a value smaller than .5, so the final value must be rounded down to: $11011101011 = 1771_{10}$. The addition was

lost due to the low precision.

The second evaluation leads to a similar result.
$1771_{10} = 11011101011.$
$.3984375_{10} = .01100110$

$$\begin{array}{r} 11011101011.000 \\ \hline + \ 00000000000.011 \\ \hline\hline = \ 11011101011.011 \end{array}$$

The guard, round, and sticky bits again give a value smaller than .5, so the final value is again set to: $11011101011 = 1771_{10}$.

This arithmetic was again verified by 16-bit arithmetic in python, while 32-bit arithmetic gives the result as 1771.7421875_{10}, which is the same as the previous order of operations given in problem 3.32.

3.34)

No, floating point arithmetic is not associative, and the order of evaluation can change the result - as was demonstrated in the previous two problems.