EN605.204.82.SU17 Computer Organization

Caleb Bowers

Assignment 2

3.1

Since the numbers 5ED4 and 07A4 are unsigned hex numbers, we only have to perform simple subtraction. It follows that:

$$5\text{ED4} - 07\text{A4} = \mathbf{5730}$$

We arrive at this answer by performing simple subtraction on each column from right to left as we would in decimal format (4-4, D-A, E-7, and 5-0).

3.2

When 5ED4 and 07A4 represent signed 16 bit the easiest way to perform subtraction is to convert the hex format into binary. Here we have:

$5\text{ED4} = 0101\ 1110\ 1101\ 0100$ and $07\text{A4} = 0000\ 0111\ 1010\ 0100$.

Since the numbers are in sign-magnitude format, the MSB determines the sign of each number, so both numbers are positive numbers. This being the case, the result is the same as in problem 3.1, **5730**.

3.3

From problem 3.2 we see that 5ED4_{16} is equivalent to $0101\ 1110\ 1101\ 0100_2$, where $5_{16} = 0101_2$, $\text{E}_{16} = 1110_2$, $\text{D}_{16} = 1101_2$, and $4_{16} = 0100_2$. Hexadecimal, base 16, is an attractive numbering system for representing values in computers, because $16 = 2^4$. So one hexadecimal digit stores 4 bits, which means a byte can be represented with two hex digits. This allows

computer architects to easily represent addresses and memory specifications in an easily readable format that converts to binary in a straightforward manner.

3.4

Since the numbers 4365 and 3412 are unsigned 12-bit octal (base 8) we can perform simple subtraction on them using base 8 arithmetic:

$$4354 - 3412 = \mathbf{0753}$$

We arrive at this answer by performing simple subtraction on each digit column from right to left as we would in decimal format (4-2, 5-1, 13-4 {We have to borrow a 1 from the 4^{th} column in order to subtract}, 3-3).

3.5

When 4365 and 3412 represent signed 12-bit octal numbers in sign-magnitude format, we want to first express these numbers in binary format, so we can determine the value of the MSB: 4365 = 100011110101 and 3412 = 011100001010. From the MSB of each number we see that 4365 is a negative number, since the MSB equals 1. So, we need to convert 4365 to an unsigned number and subtract 3412 from it and then negate the sign of the solution to return to sign-magnitude format. We have 4365 in unsigned format equals 0365, which in binary is represented as (0)00011110101. Subtracting a positive number from a negative number is like adding two positive numbers together, so we can arrive at the answer fairly easy through carry addition: (0) 00011110101 – (0)11100001010 = (0)11111111111. We then invert the sign of the answer and arrive at (1)11111111111, which in octal is represented as 7777. **Therefore,**

$4365 - 3412 = 7777$ as signed 12-bit octal numbers.

3.6

Since both 185 and 122 are unsigned 8-bit decimal integers we can carry out the subtraction simply. We then have: $\mathbf{185} - \mathbf{122} = 63$ with no overflow or underflow since 64 does not exceed the capacity of an signed 8-bit integer (127 to -128).

3.7

Assuming 185 and 122 are signed 8-bit decimal integers stored in sign-magnitude and we want to know what **185 + 122** is, then we will want to first convert each value to its binary representation in order to determine the value of the MSB:

185 = 10111001 and 122 = 01111010, so 185 is a negative number whose value is (ignoring the sign bit) (1)0111001. We can then subtract (add a negative) this value from 122, which gives us: (0)1111010 – (1)0111001 = (0)1000001. The sign of our solution takes the sign of the value whose magnitude was larger, in this case 122_{10} = (0)1111010_2. In decimal (0)1000001 is 65. Since 65 is less than 127 and greater than -128, there is no overflow or underflow.

3.8

Assuming 185 and 122 are signed 8-bit decimal integers stored in sign-magnitude and we want to know what **185 - 122** is, then we will want to first convert each value to its binary representation in order to determine the value of the MSB:

185 = 10111001 and 122 = 01111010, so 185 is a negative number whose unsigned value is now (0)0111001. A negative number subtracting a negative number is the same as adding two positive numbers, so we can add these using carry addition, which give us:

(0)0111001 + (0)1111010 = (0)10110011. When we negate the sign of this answer the solution is now (1)10110011, which exceeds the 8-bit limit we have on the numbers we can represent,

so we have overflow only. In decimal this value is 435 (the correct result -179), which exceeds the limit of 127 on signed 8-bit integers. Also, from section 3.2 of the book, if the operation is A – B and A < 0 and B $\geq$ 0 and the resulting solution is $\geq$ 0 we have experienced overflow, as we did in this problem.

3.9

151 and 214 are signed 8-bit integers in decimal representation stored in two's complement. In order to find **151 + 214** using saturating arithmetic, we first want to determine the binary two's complement representation of these values. In binary $151_{10} = 01101001_2$ and $214_{10} = 00101010_2$. Saturating arithmetic is limited to the range of the memory allocated, so a value over or under will be represented as the maximum or minimum, respectfully, of that value range. In the case of 8-bit signed integers, that range is -128 to 127 inclusive. When we add 151 to 214 using saturating arithmetic we have 01101001 + 00101010 = 10010011, which is greater than $127_{10} = 01111111_2$. **Therefore, our answer to 151 + 214 is 127.**

3.10

Referring to problem 3.9, our setup of this differs slightly in that we wish to find **151 – 214** using saturating arithmetic. Again, $151_{10} = 01101001_2$ and $214_{10} = 00101010_2$. Therefore, 151 – 214 in two's complement produces 01101001 – 00101010 = 00111111, **which in base ten is 63**.

3.11

Again, using 151 and 214 as we did in problems 3.9 and 3.10, but in this case they are represented as unsigned 8-bit integers in decimal format. Therefore, their binary representation is as follows: $151_{10} = 10010111_2$ and $214_{10} = 11010110_2$. Since these are unsigned 8-bit integers the maximum limit using saturated arithmetic is 255. Adding our numbers together we have:

$$10010111 + 11010110 = 101101101.$$

This is outside the range for our unsigned 8-bit integers. **Our answer is then $11111111_2 = 255_{10}$.**

3.32

Our numbers for this exercise are as follows:

$A = 3.984375 * 10^{-1}$, $B = 3.4375 * 10^{-1}$, and $C = 1.771 * 10^{3}$.

We wish to calculate (A + B) + C with each of these values stored in 16-bit half precision format assuming 1 guard, 1 round bit, and 1 sticky bit rounding to the nearest even. First, we need to convert our numbers into binary:

$$A = 1.10011 * 2^{-2}$$

$$B = 1.011 * 2^{-2}$$

$$C = 1.1011101011 * 2^{10}$$

Then, we must calculate A + B. Since, these exponents are already aligned we simply add them together: $1.10011 * 2^{-2} + 1.011 * 2^{-2} = 1.011111 * 2^{-1}$. We now take this result and add it to value C. In order to do this, we need to place their exponents in the same position. We now have:

$$(A + B) = .0000000000101111 * 2^{10}$$

Adding this value to C, we have (Guard, Round, Sticky):

$$(A + B) + C = 1.1011101011\mathbf{1}\mathbf{0}\mathbf{1}1111 * 2^{10}$$

Since the guard-round-sticky is greater than half, we round up and this leaves us with the solution equaling $\boldsymbol{(A + B) + C = 1.1011101100 * 2^{10} = 1.772 * 10^{3}}$ **in decimal**. In 16-bit floating point format it will have a sign bit of 0, an exponent of 25 (bias is 15, so 10 = 25 – 15), and a fraction of 1011101100 (with an assumed 1):

S	E					F									
0	1	1	0	0	1	1	0	1	1	1	0	1	1	0	0

3.33

Using the same problem values for A, B, and C from problem 3.32, we now wish to find $A + (B + C)$. First, we have (B + C); in order to calculate this value, we need to move the exponent of B into that of C's. This give us:

$$B = .000000000001011 * 2^{10}$$

adding this to C, we get (Guard, Round, Sticky):

$(B + C) = 1.10111010110\mathbf{1}\mathbf{0}11 * 2^{10}$. Using the rounding rules guard-round-sticky is less than half, so we truncate and have $(B + C) = 1.1011101011 * 2^{10}$. We need to then move the exponent of A to match the quantity (B + C):

$$A = .00000000000110011 * 2^{10}$$

adding this to the quantity (B + C), we get (Guard, Round, Sticky):

$$A + (B + C) = 1.10111010110\mathbf{1}\mathbf{1}0011 * 2^{10}$$

Since the guard-round-sticky is less than half, we truncate and arrive at our final answer being: $\boldsymbol{A + (B + C) = 1.1011101011 * 2^{10} = 1.771 * 10^{3}}$ **in decimal.** In 16-bit floating point format it will have a sign of 0, an exponent of 25 (bias is 15, so 10 = 25 – 15), and a fraction of 1011101011 (with an assumed 1):

S	E					F									
0	1	1	0	0	1	1	0	1	1	1	0	1	0	1	1

3.34

Based on my answers from 3.32 and 3.33, changing the order of operations has an effect on the solution outcome. In 3.32, we want to perform (A + B) + C and our answer is $1.772 * 10^{3}$. In 3.33, we are trying to find A + (B + C) and our answer is $1.771 * 10^{3}$. Changing the order of

operations changes the answer by a value of ten, so no we cannot rely on floating point arithmetic to ensure the associative property of math.