Rodrigo Becerril Ferreyra

Dr. Gevik Sardarbegians

CECS 440 Section 02

02 November 2021

Exercise 3

Note: in this document, I will prefix hexadecimal numbers with 0x (for example, 0xFF) and octal numbers with 0 (for example, 057). I will suffix signed decimal numbers with d (for example, 96d) and unsigned decimal numbers with ud (for example, -5ud). Binary numbers will bear no prefix or suffix.

3.1

```
0x5ED4 - 0x07A4 =

              0 10 10 10     0 10
 0  1  0  1  ~1  ~1  ~1  ~0  1  ~1  ~0  1  0  1  0  0
-0  0  0  0  0  1  1  1  1  0  1  0  0  1  0  0
 ---------------------------------------------
 0  1  0  1  0  1  1  1  0  0  1  1  0  0  0  0 = 0x5730
```

The answer is 0x5730.

3.2

```
0x5ED4 - 0x07A4 =

              0 10 10 10     0 10
 0  1  0  1  ~1  ~1  ~1  ~0  1  ~1  ~0  1  0  1  0  0
-0  0  0  0  0  1  1  1  1  0  1  0  0  1  0  0
 ---------------------------------------------
 0  1  0  1  0  1  1  1  0  0  1  1  0  0  0  0 = 0x5730
```

The answer is 0x5730. There is no difference from the unsigned version because both numbers are positive numbers.

3.6

```
185ud - 122ud =

 0  1 10 10 10  1 10
 1  0  1  1  1  0  0  1
-0  1  1  1  1  0  1  0
 ---------------------
 0  0  1  1  1  1  1  1 = 0x3F = 63ud
```

The answer is 63ud. There is no overflow nor underflow.

3.7

```
185d - 122d =

 1  1 10 10 10  1 10
 0  0  1  1  1  0  0  1
-1  1  1  1  1  0  1  0
 ---------------------
 0  0  1  1  1  1  1  1 = 0x3F = 63d
```

Note that the number 185 cannot be represented with 8 bits in sign-magnitude format. The highest number you can represent in this format is 127. The number 185 has overflowed into the number 57. There is no overflow nor underflow in the math, but the number 185 overflowed into 57. I also ran out of ones to borrow from. It is interesting to note that the answer is the same as before, even though the number 185 itself overflowed, and I ran out of bits to borrow from, meaning that the bottom number was greater than the top number.

3.9

The range of values that 8 signed bits can represent is -128d to 127d. This means that the numbers 151 and 214 cannot be represented with 8 signed bits in two's compliment representation. I decided to add the two numbers as is, which are actually -105d and -41d, respectively. Here is the calculation:

```
151d + 214d =

1 0 0 1 0 1 1 0 0 <- carry
  1 0 0 1 0 1 1 1
+ 1 1 0 1 0 1 1 0
  ---------------
1 0 1 1 0 1 1 0 1 = 0x6D = 109d
```

3.10

```
151d - 214d = 151d + (~214d) + 1 =

0 0  1  1  1  1  1  1    <- carry
  1  0  0  1  0  1  1  1
+ 0  0  1  0  1  0  0  1
+ 0  0  0  0  0  0  0  1
 ----------------------
  1  1  0  0  0  0  0  1 = 0xC1 = -63d
```

The answer here is actually correct. 151d - 214d = -63d. This is entirely a coincidence, however. Again, 8 signed bits can represent numbers from -128d to 127d. This means that neither the number 151d nor -214d can be represented in 8 bits (i.e. both values will overflow). The first number overflows to -105d, and the second number overflows to -42d. If I attempt to negate -42d and add 1 to generate the two's compliment, then I get 00101010, which is 42d.

3.11

```
151ud + 214ud =

1 0 0 1 0 1 1 0 <- carry
  1 0 0 1 0 1 1 1
+ 1 1 0 1 0 1 1 0
  ---------------
1 0 1 1 0 1 1 0 1 = 0x6D = 109ud
```

In this problem, since the 8 bits are unsigned, all numbers can be represented. 8 unsigned bits can represent the values 0d through 255d. However, in this problem, there is an overflow,

which is signaled by the last carry bit in the addition. The result is too big to be represented with 8 unsigned bits; 9 are needed.

3.13

Iteration	Step	Multiplier	Multiplicand	Product
0	Initial values	00001010	00110010	00000000_00000000
1	Add registers if 1	00001010	00110010	00000000_00000000
1	Shift multiplicand left	00001010	01100100	00000000_00000000
1	Shift multiplier right	00000101	01100100	00000000_00000000
2	Add registers if 1	00000101	01100100	00000000_01100100
2	Shift multiplicand left	00000101	11001000	00000000_01100100
2	Shift multiplier right	00000010	11001000	00000000_01100100
3	Add registers if 1	00000010	11001000	00000000_01100100
3	Shift multiplicand left	00000010	10010000	00000000_01100100
3	Shift multiplier right	00000001	10010000	00000000_01100100
4	Add registers if 1	00000001	10010000	00000000_11110100
4	Shift multiplicand left	00000001	00100000	00000000_11110100
4	Shift multiplier right	00000000	00100000	00000000_11110100

The final answer to 062×012 is 0364, or in unsigned decimal, 50ud×10ud is 244ud. This is obviously not correct. The problem is that we are missing one bit: the 9th bit should be set. The correct answer should be 500ud, or in octal, 0764. This error likely comes from the fact that the multiplier is only 8 bits wide. If it were 16 bits wide, the product would be correct.

3.15

Each adder takes τ amount of time. Each bit of the multiplier has its own adder. This means that the total amount of time taken will be 8 bits × 4 time units = **32 time units** total.

3.16

The time taken using this fast multiplication configuration takes $\tau \log_2(b)$ time, where τ is the amount of time that it takes for one adder to output its result, and b is the amount of bits

of the numbers that one wishes to multiply. In this case, a $b = 8$ bit multiplier that uses adders that take 4 time units to complete will take a total of $4 \times \log_2(8) = 4 \times 3 =$ **12 time units** to output the product.

3.23

63.25d can be converted to binary scientific notation: 63.25d → 111111.01 → 1.1111101×2^5. Since this number is positive, the MSB is 0. The exponent is 5d + 127d = 132d → 10000100. Therefore, the IEEE 32-bit floating point single-precision number is 01000010011111010000000000000000 (or, in hexadecimal, 0x427D0000).

3.24

The same values for 63.25d can be used for IEEE double precision, except for the exponent, which has an offset value of 1023d. 63.25d → 1.1111101×2^5. The exponent is now 5d + 1023d = 1028d → 10000000100. Therefore, the IEEE 64-bit floating point double-precision number is 0100000001001111101000000000000000000000000000000000000000000000 (or 0x404FA00000000000 for short).

3.27

The number to convert to IEEE 16-bit half-precision floating point is 1.5626×10^{-1}d. In fixed point, this is -0.15625d which is -5d/32d. This can be turned into binary as -0.000101, or equivalently, -1.01×2^{-4}. Because the exponent offset is 15d, the exponent is -4d + 15d = 11d → 01011. Therefore, the 16-bit floating point number is 1010110100000000, or 0xAD00. It is important to note that there is still much space for more precise significands (the least significant 8 bits are not used in this example). The 16-bit format is easier to work with by hand and takes up less space, but it is much less precise than a 32-bit floating-point number, and can hold a

smaller range of numbers. It is incomparable to the precision and range of the 64-bit floating-point number.

3.29

```
2.6125d * 10^1 + 4.150390625d * 10^(-1) =
26.125d + 0.4150390625d =
11010.001 + 0.0110101001 = // convert numbers to binary
1.1010001 * 2^4 + 1.10101001 * 2^(-2) = // normalize binary sci notation
1.1010001 * 2^4 + 0.000001101010 * 2^4 = // shift smallest to the right
// 0.000001101010 has 10 bits of precision from the IEEE standard and 2 bits
// from the guard and round bits.
// We lost the least significant 2 bits (01).

 0 0 0 0 0 1 1 0 0 0 0 0 0 <- carry
 1.1 0 1 0 0 0 1 0 0 0 0 0 * 2^4
+0.0 0 0 0 0 1 1 0 1 0 1 0 * 2^4
 -------------------------
 1.1 0 1 0 1 0 0 0 1 0 1 0 * 2^4 = 0100111010100011 (IEEE) = 0x4EA3
```

The answer to the addition is (in IEEE 16-bit floating-point format) 0100111010100011 or 0x4EA3. The 16-bit format uses 1 bit for the sign and 5 bits for the exponent, leaving 10 bits for the significand. This means that, with the inclusion of the guard and round bits, the arithmetic was made with 18 bits after the binary point. The significand after the addition is 101010001010. Since the bit after the 10th bit on the right (the 11th bit from the right) is a 1, we round the 10th bit up to a 1.