CEG 3156: Computer Systems Design (Winter 2024)
Prof. Rami Abielmona
Possible Solutions to Assignment #1: Arithmetic Circuits

January 30, 2024

Question I

This question deals with adder/subtractor arithmetic circuitries. Parts b and c are questions B.26 and B.27 in your textbook.

Part a

The conditions for overflow are the following:

- When adding two similarly-signed numbers, the sign of the result is the opposite of the sign of both numbers; or
- When subtracting two oppositely-signed numbers, the sign of the result is the same as the sign of the second number

Following those cases, we can develop four cases for overflow in our most-significant ALU block (Assume that the Binvert multiplexer output is called muxOut, where $muxOut = b \oplus Binvert$, and the adder output is called addOut):

$$\begin{aligned} Overflow = &\overline{Binvert} \cdot \overline{a} \cdot \overline{muxOut} \cdot addOut \ + \\ &\overline{Binvert} \cdot a \cdot muxOut \cdot \overline{addOut} \ + \\ &Binvert \cdot \overline{a} \cdot muxOut \cdot addOut \ + \\ &Binvert \cdot a \cdot \overline{muxOut} \cdot \overline{addOut} \end{aligned}$$

The first two lines represent the cases where we are adding two positive or two negative numbers and we get a result that is opposite in sign. The last two lines represent the cases where we are subtracting

two oppositely signed numbers and we get a result that has the same sign as the second number. In all of the four cases, overflow occurs and the output should be active.

Note that the carryOut input was not utilized in the overflow detection. Also note that the block could have been simply designed as the XOR operation between the carryIn and the carryOut signals, as shown in the bonus question.

The RTL VHDL model of the above circuit is shown below:

```
library ieee;
use ieee.std_logic_1164.all;

entity overflowDetector is
   port(
         i_bInvert     : in std_logic;
         i_a           : in std_logic;
         i_muxOut      : in std_logic;
         i_addOut      : in std_logic;
         o_overflow    : out std_logic
       );
end entity overflowDetector;

architecture rtl of overflowDetector is
   signal int_andOut1, int_andOut2, int_andOut3, int_andOut4    : std_logic;

begin
    int_andOut1 <= not(i_bInvert) and not(i_a) and not(i_muxOut) and i_addOut;
    int_andOut2 <= not(i_bInvert) and i_a and i_muxOut and not(i_addOut);
    int_andOut3 <= i_bInvert and not(i_a) and i_muxOut and i_addOut;
    int_andOut4 <= i_bInvert and i_a and not(i_muxOut) and not(i_addOut);


    -- Output driver
    o_overflow <= int_andOut1 or int_andOut2 or int_andOut3 or int_andOut4;
end architecture rtl;
```

Part b

Rewriting the equations using the new notation gives us four new equations as follows:

$$C1 = c4, C2 = c8, C3 = c12, C4 = c16$$

$$c4 = G_{3,0} + (P_{3,0} \cdot c0)$$

c8 is given in the exercise

$$c12 = G_{11,8} + (P_{11,8} \cdot G_{7,4}) + (P_{11,8} \cdot P_{7,4} \cdot G_{3,0}) + (P_{11,8} \cdot P_{7,4} \cdot P_{3,0} \cdot c0)$$

$$c16 = G_{15,12} + (P_{15,12} \cdot G_{11,8}) + (P_{15,12} \cdot P_{11,8} \cdot G_{7,4}) + (P_{15,12} \cdot P_{11,8} \cdot P_{7,4} \cdot G_{3,0}) + (P_{15,12} \cdot P_{11,8} \cdot P_{7,4} \cdot P_{3,0} \cdot G_{3,0})$$

Part c

The equations for c4, c8 and c12 are the same as those given in the solution to part b. Using the 16-bit adders means using another level of carry-lookahead logic to construct the 64-bit adder. The second level generate, $G0'$, and propagate, $P0'$, are

$$G0' = G_{15,0} = P_{15,12} \cdot G_{11,8} + P_{15,12} \cdot P_{11,8} \cdot G_{7,4} + P_{15,12} \cdot P_{11,8} \cdot P_{7,4} \cdot G_{3,0}$$

and

$$P0' = P_{15,0} = P_{15,12} \cdot P_{11,8} \cdot P_{7,4} \cdot P_{3,0}$$

Using $G0'$ and $P0'$ we can write c16 more compactly as

$$c16 = G_{15,0} + P_{15,0} \cdot c0$$

and

$$c32 = G_{31,16} + P_{31,16} \cdot c16$$
$$c48 = G_{47,32} + P_{47,32} \cdot c32$$
$$c64 = G_{63,48} + P_{63,48} \cdot c48$$

A 64-bit adder diagram in the style of Figure 4.24 is shown in figure 1.

Part d

A full subtractor is similar to a full adder in design, but different in functionality. Let us assume we have three inputs, mainly the minuend (a_i), the subtrahend (b_i) and the previous borrow (c_i), and two outputs, mainly the difference (D_i) and the borrow (B_i). The truth table of the 1-bit full subtractor is shown in table 1. The output bit D is obtained from the subtraction, $a_i - (b_i + c_i$. The output bit B is 0 if $a_i \geq b_i$ provided $c_i = 0$. If $c_i = 1$, output bit B if 1 if and only if $a_i \leq b_i$.

The Karnaugh map-derived Boolean expressions for B and D are:

$$B = \overline{a_i} c_i + \overline{a_i} b_i + b_i c_i$$
$$D = \overline{a_i}\,\overline{b_i} c_i + \overline{a_i} b_i \overline{c_i} + a_i \overline{b_i}\,\overline{c_i} + a_i b_i c_i$$

The resulting logic implementations for the difference and the borrow outputs are shown in figure 2.

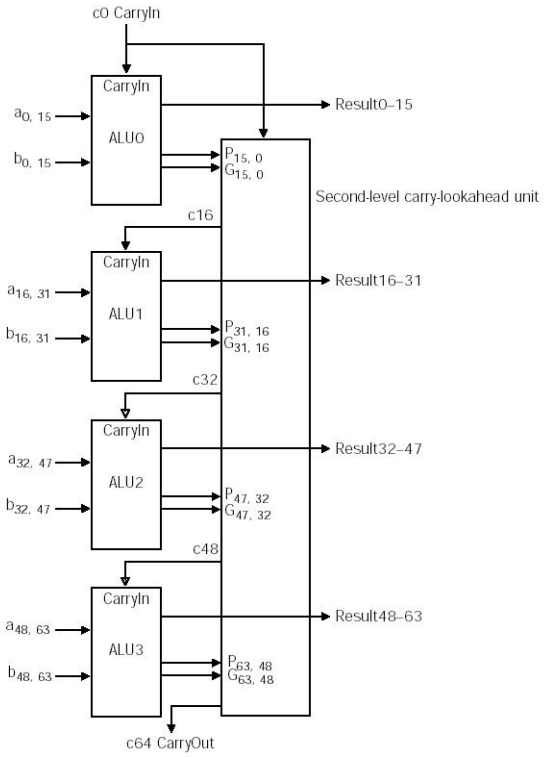

Figure 1: Possible realization of a 64-bit carry-lookahead adder

Question II

This question deals with multiplication circuitries. For part b of this question, you might want to look at the implementation of **nxn array multipliers**.

Part a

The largest value of an n-digit number, if unsigned in base B, is $B^n - 1$. Squaring this value gives us the largest possible result of $B^{2n} - 2B^n + 1$. This value is smaller than $B^{2n} - 1$, and hence can be represented using $2n$ digits.

If the n-digit numbers were taken in 2's complement representation, the largest positive n-digit number is $B^n - 1$, yielding a product of $B^{2n} - 2B^n + 1$, which is representable in 2n bits. Finally, the largest negative n-digit number is $-B^{n-1}$, yielding a product of $-B^{2n-2}$, which is representable in 2n bits.

Thus, the multiplication of two n-digit numbers will always yield a product representable by 2n digits.

a_i	b_i	c_i	D_i	B_i
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

Table 1: Truth table for a 1-bit full subtractor

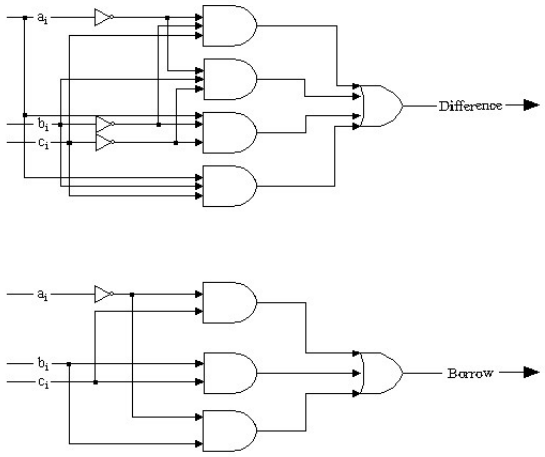

Figure 2: Difference and borrow output logic realizations

Part b

Shown in figure 3 is a possible realization of a 3,2 full adder. This module can be used, along with AND gates to form the array cell shown in the bottom of figure 4. At the top of that figure is shown the 4x4 array multiplier using the aforementioned cells. The final product is essentially the fourth partial product. It is easy to show the multiplication of 11x5 on the diagram: the multiplicand is (1011), while the multiplier is (0101), and the fourth partial product is (00110111).

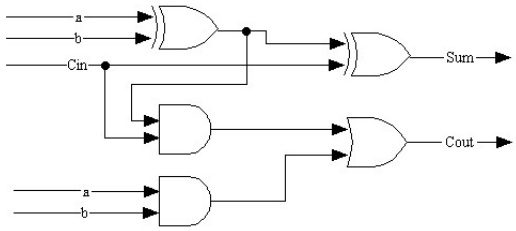

Figure 3: Possible realization of a 1-bit full adder

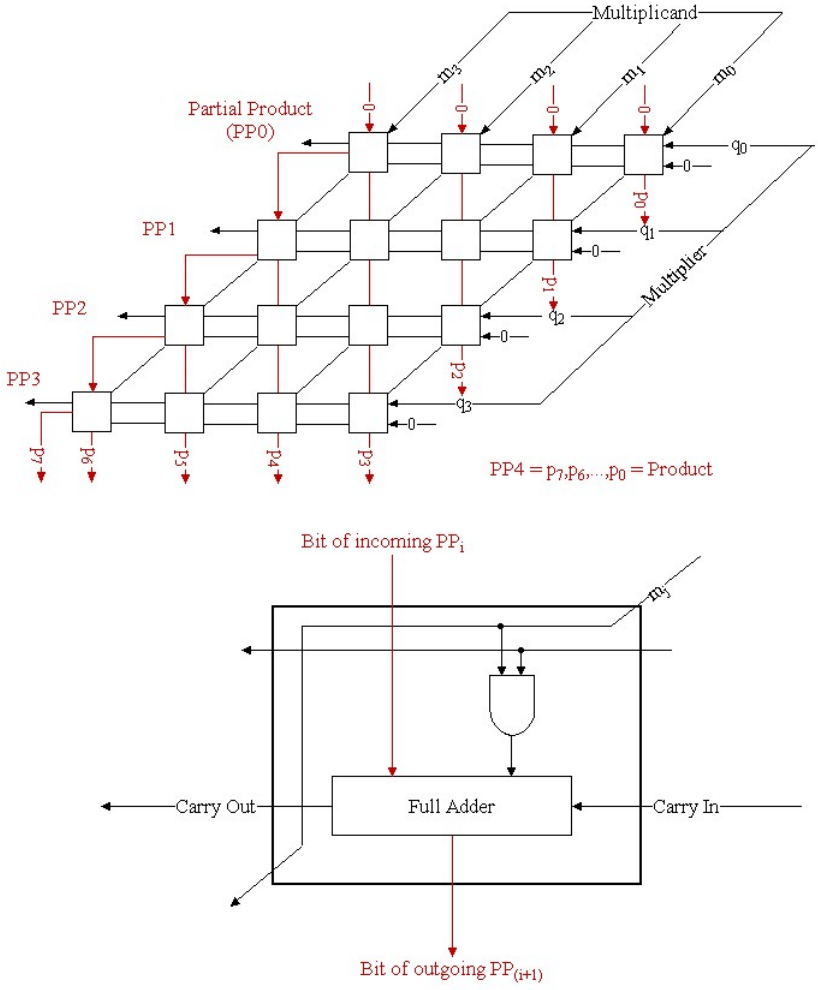

Figure 4: Possible realization of a 4x4 array multiplier

Question III

This question deals with division algorithms.

Part a

A *nonrestoring division* algorithm in the style of Figure 4.40 is shown in figure 5.

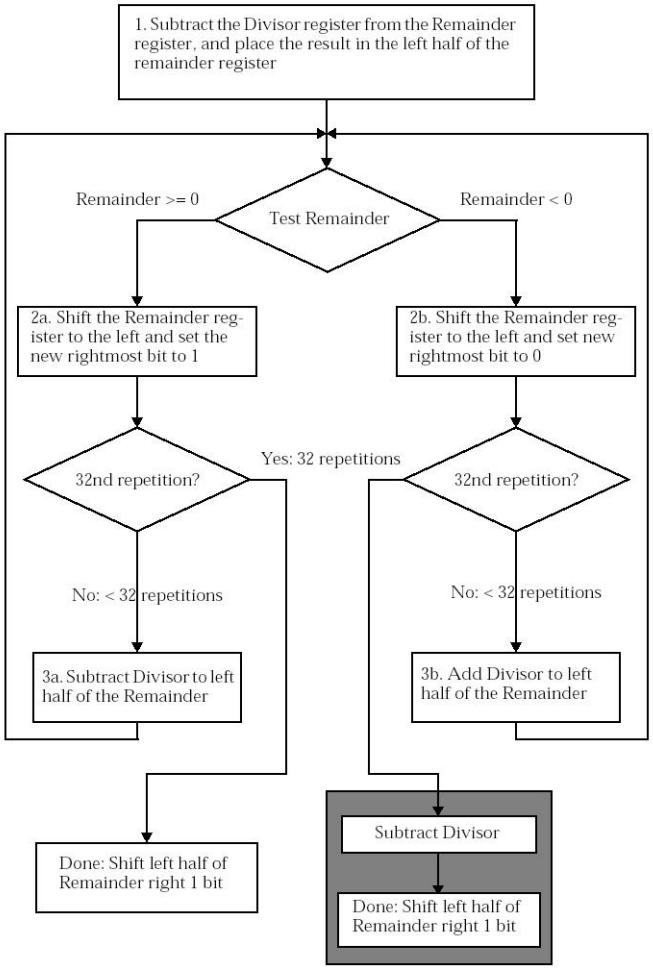

Figure 5: Possible realization of nonrestoring division algorithm

Note that the shaded boxes are necessary when Remainder < 0 to correct the redundant addition in step 3b should the last step end up in this

branch of the flow chart. As shown in figure 6, 7 divided by 2 is 3 with a remainder of 1.

Iteration	Step	Divisor	Remainder
0	Initial Values	0010	0000 1110
1	1: Rem – Rem – Div	0010	1110 1110
	2b: Rem < 0 ⇒ sll R, R0 = 0	0010	1101 1100
	3b: Rem = Rem + Div	0010	1111 1100
2	2b: Rem < 0 ⇒ sll R, R0 = 0	0010	1111 1000
	3b: Rem = Rem + Div	0010	0001 1000
3	2a: Rem > 0 ⇒ sll R, R0 = 1	0010	0011 0001
	3a: Rem = Rem – Div	0010	0001 0001
4	2a: Rem > 0 ⇒ sll R, R0 = 1	0010	0010 0011
Done	Shift Rem right 1	0010	0001 0011

Figure 6: Nonrestoring division example

Part b

The steps for deriving the floating-point division algorithm involve:

Step 1 Unlike multiplication, we calculate the exponent of the product by just subtracting the exponents of the operands:

New exponent = 10 - (-5) = 15

Step 2 Next comes the division of the significands, shown in figure 7

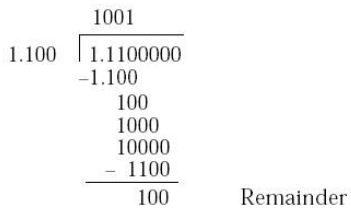

Figure 7: Division of the significands

1.001 with a remainder of 0.100/1.100. Hence the quotient is $1.001 * 10^{15}$. Of course we must check for division by zero.

Step 3 This product is normalized, so we need nothing more to do.

Step 4 Check to see if the exponents overflow or underflow.

Step 5 The sign of the quotient depends on the signs of the original operands. If they are both the same, the sign is positive, otherwise it's negative. Hence the quotient is $+1.001 * 10^{15}$.

A floating-point division algorithm in the style of Figure 4.46 is shown in figure 8.

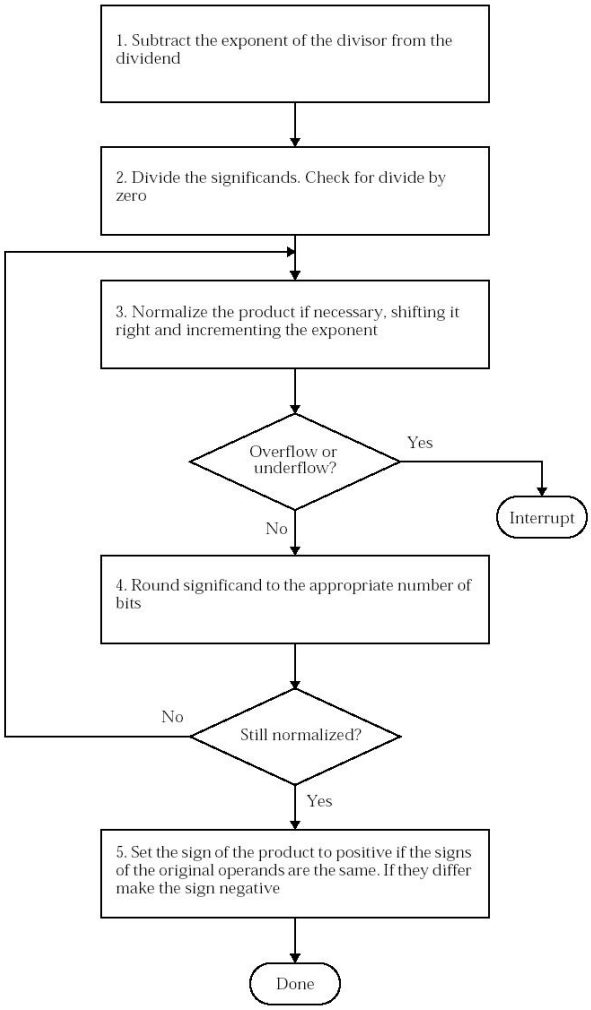

Figure 8: Possible realization of floating-point division algorithm

Question IV

This question deals with floating-point arithmetic. Assume that for our floating-point representation here, we have a ceg3156-precision as shown in figure 9.

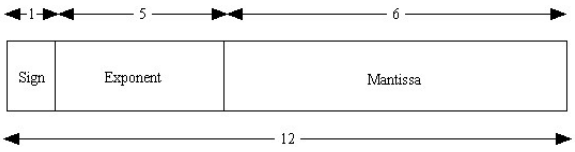

Figure 9: Ceg3156-precision floating-point number representation

Here are a couple of notes to keep in mind while working with ceg3156-precision:

Sign 1-bit wide, and used to denote the sign of the number (0 signifies + and 1 signifies -);

Exponent 5-bit signed exponent in excess-15 representation;

Mantissa 6-bit fractional component.

Part a

`Let us derive the IEEE-754 binary representation of the number using ceg3156-precision.`

$$\begin{aligned}(25.125)_{10} &= (201 * 2^{-3})_{10} \\ &= (1.100100)_2 * 2^4 \\ &= (-1)^0 * (1 + 0.100100) * 2^{19-154}\end{aligned}$$

`Hence, we have the following values for`

Sign $(0)_2$`, indicating it is a positive number;`

Exponent $(10011)_2$`, representing` $(19)_{10}$`;`

Mantissa $(100100)_2$`, representing the rounded value of the 6-bit field.`

`and the final 16-bit number is` $(010011100100)_2$`.`

Part b

`Let us derive the IEEE-754 binary representation of the number using ceg3156-precision.`

$$\begin{aligned}(-1023.666..)_{10} &= -(1023 + 0.666...)_{10} \\ &= (1.111111111110101010...)_2 * 2^9 \\ &= (-1)^1 * (1 + 0.111111) * 2^{24-15}\end{aligned}$$

`Hence, we have the following values for`

Sign $(1)_2$`, indicating it is a negative number;`

Exponent $(11000)_2$, representing $(24)_{10}$;

Mantissa $(111111)_2$, representing the rounded value of the 6-bit field.

and the final 16-bit number is $(111000111111)_2$.

Part c

Let us derive the IEEE-754 binary representation of the largest number using ceg3156-precision.
Here, assuming no special cases (zero, denormalized numbers, infinity and not a number), we have the following values for

Sign $(0)_2$, indicating it is a positive number;

Exponent $(11111)_2$, representing $(31)_{10}$;

Mantissa $(111111)_2$, representing the largest value of the 6-bit field.

Now, let us derive the decimal value of $(011111111111)_2$:

$$\begin{aligned} &= (-1)^0 * (1 + 0.111111) * 2^{31-15} \\ &= (1.111111)_2 * 2^{16} \\ ((1 + \tfrac{1}{2} + \tfrac{1}{4} + \tfrac{1}{8} + \tfrac{1}{16} + \tfrac{1}{32} + \tfrac{1}{64}) * 2^{16})_{10} &= (1.984375 * 2^{16})_{10} \end{aligned}$$

Thus, the largest number is $(130048)_{10}$.

Let us now derive the IEEE-754 binary representation of the smallest number using ceg3156-precision.
Again, assuming no special cases (zero, denormalized numbers, infinity and not a number), we have the following values for

Sign $(1)_2$, indicating it is a negative number;

Exponent $(11111)_2$, representing $(31)_{10}$;

Mantissa $(111111)_2$, representing the largest value of the 6-bit field.

Now, let us derive the decimal value of $(111111111111)_2$:

$$\begin{aligned} &= (-1)^1 * (1 + 0.111111) * 2^{31-15} \\ &= -(1.111111)_2 * 2^{16} \\ ((1 + \tfrac{1}{2} + \tfrac{1}{4} + \tfrac{1}{8} + \tfrac{1}{16} + \tfrac{1}{32} + \tfrac{1}{64}) * 2^{16})_{10} &= (-1.984375 * 2^{16})_{10} \end{aligned}$$

Thus, the smallest number is $(-130048)_{10}$.

Also accepted as a solution is the derivation of the smallest number in magnitude, which goes as follows:

Sign $(1)_2$, indicating it is a negative number;

Exponent $(00000)_2$, representing $(0)_{10}$;

Mantissa $(000000)_2$, representing the smallest value of the 6-bit field.

Now, let us derive the decimal value of $(1000000000000)_2$:

$$\begin{aligned} &= (-1)^1 * (1 + 0.000000) * 2^{0-15} \\ &= -(1.000000)_2 * 2^{-15} \\ (-1.0) * 2^{-15})_{10} &= (-1.0 * 2^{-15})_{10} \end{aligned}$$

Thus, the smallest number in magnitude is $(0.0000305175)_{10}$.

Note that in the IEEE-754, single- or double-precision, there are special definitions for both positive and negative representations of infinity and 0. If the exponent field is all 1's, the mantissa field is all 0's and the sign bit is 0, then +Infinity is indicated, however if the exponent field is all 1's, the mantissa field is all 0's and the sign bit is 1, then -Infinity is indicated. Similarly, if both the exponent and mantissa fields are all 0's and the sign bits is 0, then +0 is indicated, however if both the exponent and mantissa fields are all 0's and the sign bit is 1, then -0 is indicated. Denormalized numbers are represented by varying the mantissa field, while keeping the exponent field at all 0's, and not a numbers (NaNs) are represented by varying the mantissa field, while keeping the exponent field at all 1's.

Part d

We are performing the addition, subtraction, multiplication and division of A $(100010011001)_2$ and B $(001110111000)_2$, two numbers written in ceg3156-precision floating-point representation.

Firstly, the addition of the two numbers is performed. Since E_{diff} in this case is greater than the number of bits in the fractional part (12 > 6), we clear the fractional part of the smaller exponent number, in this case A. Also, since the exponent of B is greater that of A (14 > 2), the exponent of the result will be that of B. Hence, we have

- $RF_Z = RF_A + RF_B$
- $RF_Z = RF_B$
- $RE_Z = RE_B$
- if $RF_Z \geq 1$ then $RE_Z = REZ + 1;\ RF_Z = RF_Z << 1;$

Thus, $RF_Z = (111000)_2$ and $REZ = (01110)_2$, and the result is already normalized. Hence, the final sum is $Z = (001110111000)_2$ or $Z = (+0.9375)_{10}$.

Secondly, the subtraction of the two numbers is performed. Since we are subtracting a large number from a very small number, which does

not fit into our fractional field (6-bits), we negate the previous result to get a result of $Z = (101110111000)_2$ or $Z = (-0.9375)_{10}$.

Thirdly, the multiplication of the two numbers is performed. We perform the following multiplication procedure:

$$[(-1^1 * (1 + 0.011001) * 2^{2-15}] * [(-1)^0 * (1 + 0.111000) * 2^{14-15}] \quad (1)$$
$$= -(1.011001 * 1.111000)_2 * 2^{-14} \quad (2)$$
$$= -(10.100110111000)_2 * 2^{-14} \quad (3)$$
$$= -(1.0100110111000)_2 * 2^{-13} \quad (4)$$
$$= (-1)^1 * (1 + 0.010011) * 2^{2-15} \quad (5)$$
$$= (-1.296875 * 2^{-13})_{10} \quad (6)$$

As can be seen, the product of the mantissas was performed in step 2 above, and normalized in step 4. Henceforth, our final result is $Z = (100010010011)_2$ or $Z = (-1.296875 * 2^{-13})_{10}$.

Finally, the division of the two numbers is performed. We perform the following division procedure:

$$[(-1^1 * (1 + 0.011001) * 2^{2-15}]/[(-1)^0 * (1 + 0.111000) * 2^{14-15}] \quad (7)$$
$$= -(1.011001/1.111000)_2 * 2^{-12} \quad (8)$$
$$= -(0.101111)_2 * 2^{-12} \quad (9)$$
$$= -(1.011110)_2 * 2^{-13} \quad (10)$$
$$= (-1)^1 * (1 + 0.011110) * 2^{2-15} \quad (11)$$
$$= (-1.46875 * 2^{-13})_{10} \quad (12)$$

As can be seen, the quotient of the mantissas was performed in step 8 above, and normalized in step 10. Henceforth, our final result is $Z = (100010011110)_2$ or $Z = (-1.46875 * 2^{-13})_{10}$.

As a note, let us explore the decimal differences between double-precision and ceg3156-precision results. Listed in table 2 are those results, both shown to eight significant digits. We can see the minor differences due to the gain in precision when going to the 64-bit double-precision format defined in the IEEE-754 floating-point representation standard.

Bonus Question

An overflow occurs when two 2's complement numbers are added, if the carry-in bit into the sign bit is different from the carry-out bit from the sign bit. The sign bit, in this case, is the most significant

Operation	*Double-precision*	*ceg3156-precision*
$A + B$	0.93733024	0.9375000
$A - B$	-0.93766975	-0.9375000
$A * B$	-0.00015914	-0.00015830
A/B	-0.00018107	-0.00017929

Table 2: Difference in precision

bit. Hence, the exclusive-or of the carry-in and carry-out signals of the most significant bit, produces the overflow bit. Refer to table 3 for a clearer explanation of the cases involved.

Sign A	*Sign B*	*Carry In*	*Carry Out*	*Sign result*	*Correct sign of result*	*Overflow ?*	*Carry In XOR Carry Out*	*Notes*
0	0	0	0	0	0	No	0	
0	0	1	0	1	0	Yes	1	Carries differ
0	1	0	0	1	1	No	0	$\|A\| < \|B\|$
0	1	1	1	0	0	No	0	$\|A\| > \|B\|$
1	0	0	0	1	1	No	0	$\|A\| > \|B\|$
1	0	1	1	0	0	No	0	$\|A\| < \|B\|$
1	1	0	1	0	1	Yes	1	Carries differ
1	1	1	1	1	1	No	0	

Table 3: Table explaining overflow detection

Acknowledgements

Answers and figures related to textbook questions are drawn from the accompanying professor's manual of *Computer Organization and Design*, fifth edition, by Patterson and Hennessy.