3.1 What is 5ED4 − 07A4 when these values represent unsigned 16-bit hexadecimal numbers? The result should be written in hexadecimal. Show your work.
Convert both numbers to decimals:
5ED4 (hex) = 24244 (decimal)
$(5ED4)_{16} = (5 \times 16^3) + (14 \times 16^2) + (13 \times 16^1) + (4 \times 16^0) = (24276)_{10}$
07A4 (hex) = 1956 (decimal).
$(07A4\ )_{16} = (0 \times 16^4) + (7 \times 16^3) + (10 \times 16^2) + (4 \times 16^1) + (-16 \times 16^0) = (1956)_{10}$
Subtract the second number from the first.
24244 (decimal) - 1956 (decimal) = 22288 (decimal)
Convert the decimal result back to hexadecimal:
22288 (decimal) = 57C0 (hexadecimal)

(22288)/16 = 1393	R0	place0
(1393)/16 = 87	R1	place1
(87)/16 = 5	R7	place2
(5)/16 = 0	R5	place3

3.2 What is 5ED4 − 07A4 when these values represent signed 16-bit hexadecimal numbers stored in sign-magnitude format? The result should be written in hexadecimal. Show your work.
5ED4 (hex) = 24276 (decimal)
For 07A4 (hex), the leftmost bit is also 0, so the value is positive:
07A4 (hex) = 1956 (decimal)
To subtract 07A4 from 5ED4, we can subtract the decimal values as we did in part 3.1:
24244 - 1956 = 22288
5ED4 - 07A4 = 57C0 in sign-magnitude format.

3.3 Convert 5ED4 into a binary number. What makes base 16 (hexadecimal) an attractive numbering system for representing computer values?
5 in hexadecimal is 0101 in binary.
E in hexadecimal is 1110 in binary.
D in hexadecimal is 1101 in binary.
4 in hexadecimal is 0100 in binary.
It's easy to split up into 4 different sets of 4 bits
5ED4 in hexadecimal is equivalent to 0101111011010100 in binary.

3.7 Assume 185 and 122 are signed 8-bit decimal integers stored in sign-magnitude format. Calculate 185 + 122. Is there overflow, underflow, or neither?
185 = 0 01111010
Since both numbers have the same sign bit (0), we can perform addition as usual:
10111001 + 01111010 = 1 00110011
The remaining bits represent the magnitude of the number, which is 00110011 in binary or 51 in decimal. Therefore, the sum of 185 and 122 in sign-magnitude format is -51. In this case, there is no overflow or underflow since the result is within the range of representable integers for 8-bit signed integers in sign-magnitude format.

3.17 As discussed in the text, one possible performance enhancement is to do a shift and add instead of an actual multiplication. Since 9 × 6, for example, can be written (2 × 2 × 2 + 1) × 6, we can calculate 9 × 6 by shifting 6 to the left three times and then adding 6 to that result. Show the best way to calculate 0 × 33 × 0 × 55 using shifts and adds/subtracts. Assume both inputs are 8-bit unsigned integers.

0x33 = 3*161 + 3*160 = 51 = 25 + 24 + 2 + 1 = 0000000000110011

and, 0x55 = 5*161 + 5*160 = 85 = 26 + 24 + 22 + 1 = 0000000001010101

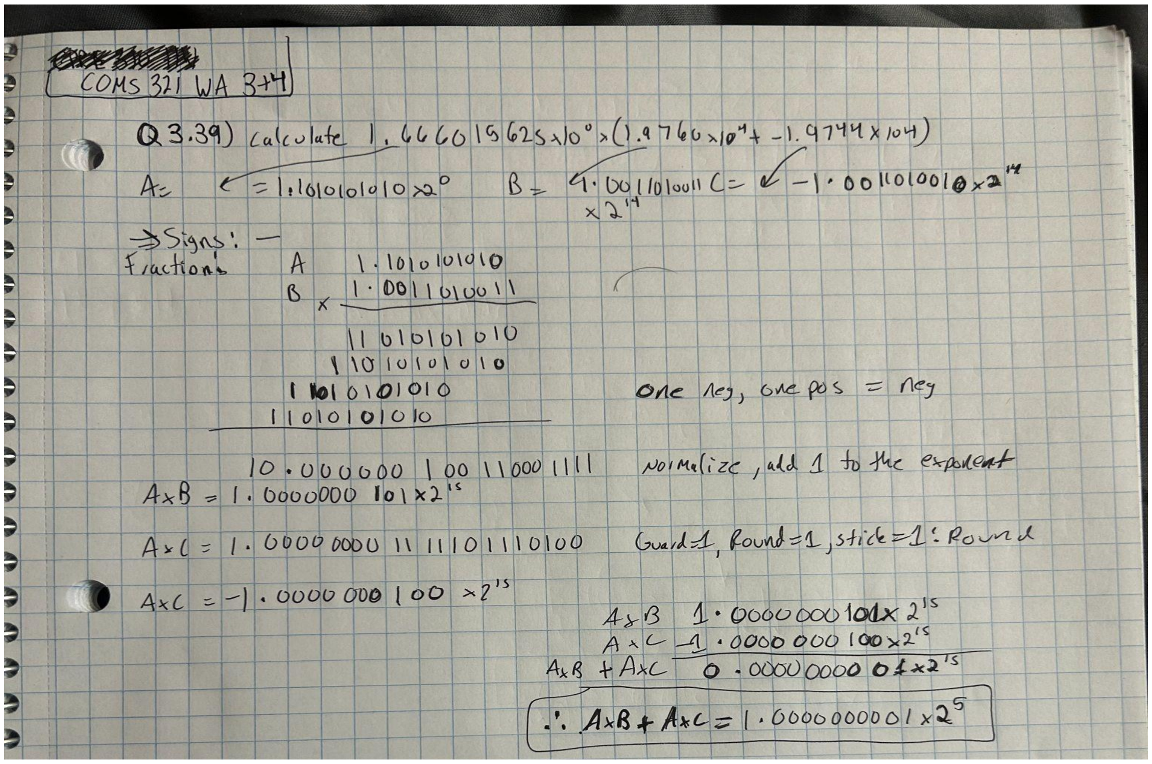
Now, we can write the equation as,

0x33 * 0x55 = (2^5 + 2^4 + 2 + 1) * 0x55

By using shifts and adds, we assume the result is of 32-bit and initialize it as zero and, add 0x55 into the result by shifting it 5 times, then 4 times, then 1 time and then 0 time in left.

The value of the result is: 00000000000000000001000011101111 =10EF(in hexadecimal) = 4335(in decimal)

3.39 Calculate (1.666015625 × 100 × 1.9760 × 104) + (1.666015625 × 100 × −1.9744 × 104) by hand, assuming each of the values is stored in the 16-bit half precision format described in Exercise 3.27 (and also described in the text). Assume 1 guard, 1 round bit, and 1 sticky bit, and round to the nearest even. Show all the steps, and write your answer in both the 16-bit floating point format and in decimal.



~~[illegible]~~

COMS 321 WA 3+4

Q 3.39) calculate $1.666015625\times10^0\times(1.9760\times10^4 + -1.9744\times104)$

A = 1.1010101010×2^0 B = 1.0011010011×2^{14} C = $-1.0011010010\times2^{14}$

→ Signs: −

Fractions

```
  A       1.1010101010
  B  ×    1.0011010011
          ------------
           11010101010
          110101010100
        1101010101010
       11010101010
    ----------------
      10.0000000100111000111
```

One neg, one pos = neg

Normalize, add 1 to the exponent

$A\times B = 1.0000000101\times2^{15}$

$A\times C = 1.0000000011111011110100$ Guard=1, Round=1, stick=1: Round

$A\times C = -1.0000000100\times2^{15}$

```
       A×B    1.0000000101×2^15
       A×C   −1.0000000100×2^15
   A×B + A×C  0.0000000001×2^15
```

$\boxed{\therefore A\times B + A\times C = 1.0000000001\times2^5}$

4.1 Consider the following instruction: Instruction: AND Rd, Rn, Rm Interpretation: Reg[Rd] = Reg[Rn] AND Reg[Rm]

4.1.1 What are the values of control signals generated by the control in Figure 4.10 for this instruction?
The control signals intended for this AND instruction be,
ffrite =1
memRead =0
MemWrite = 0
ALU Mux =1
ALU Op =AND
RegMux =1, ALU Branch = 0

4.1.2 Which resources (blocks) perform a useful function for this instruction?
The blocks, ALU, Registers, PC, and instruction memory are useful, except the block data memory is not useful.

4.1.3 Which resources (blocks) produce no output for this instruction? Which resources produce output that is not used?
Block Data Memory does not produce any output for this AND-instruction. The block BranchAdd produces output that is not utilized for this AND instruction. The production of BranchAdd is useful simply for Brach instruction.

4.2 Explain each of the “don’t cares” in Figure 4.18.
There are 3 "don't care" values in the table denoted by "X".
These "don't care" values represent settings for unnecessary control signals or not required for a particular instruction. The designer can select any value in these "don't care" fields by understanding the system's behavior.

4.9 Consider adding a multiplier to the CPU shown in Figure 4.23. This addition will add 300ps to the latency of the ALU but will reduce the number of instructions by 5% (because there will no longer be a need to emulate the multiply instruction).

4.9.1 What is the clock cycle time with and without this improvement?
Clock cycle
= 1- men + Mux + ALU + MUI + MUX + D men + Regs.
= 400 + 200 + 30 + 120 - 350 + 30 + 200
= 1330 ps
Without multiplier:
Clock cycle = Regs + MUX + 1 - Men + ALU + MUX + Regs + D- Men
= 400+200+30+420-350+200
= 1630 ps

4.9.2 What is the speedup achieved by adding this improvement?
Speed without improvement:
1330ps/1630ps = about 0.815
Speed up with improvement:

1330ps/ (1630 ps + 300 ps) = about 0.732

4.9.3 What is the slowest the new ALU can be and still result in improved performance?
To have improved performance, the cost/performance ratio should decrease.
The slowest ALU latency for improved performance is 700 ps, which achieves better cost/performance ratio (2.76) compared to the original latency of 100 ps while still maintaining performance improvement.

4.18 Assume that X1 is initialized to 11 and X2 is initialized to 22. Suppose you executed the code below on a version of the pipeline from Section 4.5 that does not handle data hazards (i.e., the programmer is responsible for addressing data hazards by inserting NOP instructions where necessary). What would the final values of registers X3 and X4 be? ADDI X1, X2, #5 ADD X3, X1, X2 ADDI X4, X1, #15
The final values of registers X3 and X4 would be 33 and 26 respectively.
The value of register X4 is not affected by the data hazard because it does not use the value of register X1. The final value of register X4 would be 26 (11 + 15).

4.19 Assume that X1 is initialized to 11 and X2 is initialized to 22. Suppose you executed the code below on a version of the pipeline from Section 4.5 that does not handle data hazards (i.e., the programmer is responsible for addressing data hazards by inserting NOP instructions where necessary). What would the final values of register X5 be? Assume the register file is written at the beginning of the cycle and read at the end of a cycle. Therefore, an ID stage will return the results of a WB state occurring during the same cycle. See Section 4.7 and Figure 4.51 for details. ADDI X1, X2, #5 ADD X3, X1, X2 ADDI X4, X1, #15 ADD X5, X1, X1
ADDI X1, X2, #5 adds the value of register X2 (which is 22) to the immediate value 5, resulting in 27. The result is stored in register X1, so X1 now contains 27.
ADD X3, X1, X2 adds the value of register X1 (which is 27) to the value of register X2 (which is 22), resulting in 49. The result is stored in register X3, so X3 now contains 49.
ADDI X4, X1, #15 adds the value of register X1 (which is 27) to the immediate value 15, resulting in 42. The result is stored in register X4, so X4 now contains 42.
ADD X5, X1, X1 adds the value of register X1 (which is 27) to itself, resulting in 54. The result is stored in register X5, so X5 now contains 54.
Therefore, the final value of register X5 is 54.

4.20 Add NOP instructions to the code below so that it will run correctly on a pipeline that does not handle data hazards. ADDI X1, X2, #5 ADD X3, X1, X2 ADDI X4, X1, #15 ADD X5, X3, X2

```
ADDI X1, X2, #5
NOP
NOP
ADD X3, X1, X2
ADDI X4, X1, #15
NOP
ADD X5, X3, X2
```