3666 hw 5

Casey Provitera

March 2024

1

Find the normalized (binary) representation of the following single-precision floating point numbers. Then, write the number in a) in decimal, and write the significand in b) in decimal.
Format for normalized single-precision floating point numbers:
First bit is sign bit, 0 for positive, 1 for negative.
Next 8 bits are exponents which are unsigned you then subtract 127 from the encoded exponent to get the actual exponent.
Next 23 bits are the number
There should only be 1 non zero value to the left of the decimal point

1. 0x45652000
   To binary:
   0b 0100 0101 0110 0101 0010 0000 0000 0000
   sign bit = 0
   exponent = 100 0101 0
   = 1000 1010 = 128+8+2 = 138 = encoded exponent
   actual exponent = 138-127 = 11
   numerical value = 1.110 0101 001 $*2^{11}$
   We can say the missing 1 is in binary and simply add it before the decimal on the line above since 1_2 and 1_{10} are the same
   = 0b 1110 0101 0010 = 2+16+64+512+1024+2408
   = 3666

2. 0x00070000
   To binary:
   0b 0000 0000 0000 0111 0000 0000 0000
   sign bit = 0
   exponent = 000 0000 0
   denormalized number so instead of 1 on the right side of the decimal we have 0, and exponent is 126.
   numerical value = 0.000 0111$*2^{-126}$
   move the bits over 5 places 1.11 $*2^{-5} * 2^{-126}$

$= (1.5+.25)*2^{-131}$
$= 1.75*2^{-131}$

2

Find the single-precision representation of the following values/numbers.

1. $-15.82 * 2^{-10}$
   15 in binary = 0b1111
   .82 in binary =
   .82*2 = 1.64
   .64*2 = 1.28
   .28*2 = .56
   .56*2 = 1.12
   .12*2 = .24
   .24*2 = .48
   .48*2 = .96
   .96*2 = 1.92
   .92*2 = 1.84
   .84*2 = 1.68
   .68*2 = 1.36
   .36*2 = .72
   .72*2 = 1.44
   .44*2 = .88
   .88*2 = 1.76
   .76*2 = 1.52
   .52*2 = 1.04
   .04*2 = .08
   .08*2 = .16
   .16*2 = .32
   .32*2 = .64
   .64*2 = REPEATING TERM
   Since there is a repeating term we know the decimal will continue forever so we stop here.
   binary representation = 0b1111.11010001111010111000
   move decimal points to get 0b1.11111010001111010111000 * 2^3
   need to include the sign bit and add the powers of 2 to find the exponent
   sign bit is 1 since the number is negative then $2^3 * 2^{-10} = 2^{-7}$
   Actual exponent = -7 = encoded exponent -127, encoded exponent = 120
   120 to binary = 0b1111000 this is a 7 bit number but we have 8 bits for the exponent so we pad with a 0 to get 0b01111000.
   Now put all the part together to get
   0b 1011 1100 0111 1101 0001 1110 1011 1000
   to hex = 0xBC7D1EB8

2. -831.9

831 in binary = 0b1100111111
.9 in binary = .9*2 = 1.8
.8*2 = 1.6
.6*2 = 1.2
.2*2 = .4
.4*2 = .8
.8*2 = REPEATING TERM
Since there is a repeating term we know the decimal will continue forever so we stop here.
binary representation = 0b1100111111.11100110011001
need to round since it repeats forever = 0b1100111111.11100110011010
move decimal point to get 0b1.10011111111100110011010 * 2^9
Actual exponent = 9 = encoded exponent - 127, encoded exponent = 136
136 to binary = 0b10001000
sign bit = 1
Put all the part together to get
= 0b 1100 0100 0100 1111 1111 1001 1001 1010
to hex = 0xC44FF99A

We may need to do rounding, which is another complicated issue. In this question, we use the rule "Round to nearest, ties away from zero". For example, if keep only two bits after the binary points, 0.1110 is rounded to 1.00. -0.1010 is rounded to -0.11
Write the normalized (binary) representation of each number first. Then find out the bits in each field.
Example:
2.32 = 0b10.0101_0001_1110_1011_1000_0101...
= 0b1.00101_0001_1110_1011_1000_01 ... $\times 2^1$

Number	2.32	$-15.82 \text{ x } 2^{-10}$	-831.9
S	0	1	1
Exponent	1000 0000	0111 1000	1000 1000
Fraction	00101000111101011100001	11111010001111010111000	10011111111100110011010
Single-precision	0x40147AE1	0xBC7D1EB8	0xC44FF99A

3

Find the largest odd integer that can be represented in single-precision format. Write the number in decimal and 8 hexadecimal digits for its single-precision floating-point representation. Justify your answer.
First we will start by writing out the number in binary.
Since we are looking for the largest number it should be positive so the sign bit is 0.
Secondly we know that all the bits in the factional portion of the number should

be 1 to not loose out on any potential value so the fractional portion looks like this:
0b.11111111111111111111111
Since we need the number to be the highest ODD number we need the final bit before the decimal point to be 1. That means we can not multiply the fractional part by more than the number of digits in it. That would make the actual exponent be 23 since if we multiplied by 2 one more time we would have an even number, which makes the encoded exponent 127+23= 150
149 in binary = 0b10010110
Now putting all the part together the final answer in binary is as follows:
0 sign bit, 10010110 exponent, 11111111111111111111111 fraction
= 0b 0100 1011 0111 1111 1111 1111 1111 1111
in hex = 0x4B7FFFFF
To convert binary to decimal easily we can say that the final binary is just
111 1111 1111 1111 1111 1111 which is the same as 2^{23}-1 = 8388607

4

Implement the following C code with RISC-V assembly code. Assume F-extension is available. Follow RISC-V calling conventions and use symbolic register names (e.g., ft0 and fa0) in your code. Skeleton code is provided. Only include this function in the submission.

1. float dot_product(float x[ ], float y[ ], int n) {
2. float sum = 0.0;
3. for (int i = 0; i < n; i += 1)
4. sum += $x[i] * y[i]$;
5. return sum; }

1. dot_product:
   # f0 floating point 0
2. fcvt.s.w f0, x0
3. fadd.s f1, f0, f0, dyn # f1 is representing sum
   # convert s0 to floating point number
4. add t0, x0, x0 # used as counter
5. blt t0, a2, loop # if counter less than n
6. beq x0, x0, end

7. loop:
   # get floats from memory
8. flw f2, 0(a0) # represents x[i]
9. flw f3, 0(a1) # represents y[i]
   # move index of array of floats
   # using 4 because all values in array are 4 digits long
10. addi a1, a1, 4 # y[ ]
11. addi a0, a0, 4 # x[ ]
12. fmadd.s f1, f2, f2, f1 # equivalent to f1 = f2*f3+f1
13. addi t0, t0, 1 # increment counter
14. blt t0, a2, loop # if counter less than n rerun loop
15. end:
    # convert to integer so we can return from function
16. fcvt.w.s a0, f1
17. jalr ra, x0, 0

5

Consider two processors P1 and P2 that have the same ISA but different implementations. The ISA has four classes of instructions: class A, B, C, and D. The clock rate of two processors and the number of clock cycles required for each class on the processors are listed in the following table.

Processor	Clock Rate	Class A	Class B	Class C	Class D
P1	2 GHz	1	2	3	3
P2	3 GHz	1	1	4	5

Suppose the breakdown of instructions executed in a program is as follows:
10% class A, 20% class B, 50% class C, and 20% class D.
Round answers to the nearest hundredth if necessary. For example, 1/2 to 0.5, 2/3 to 0.67.

1. What is the overall CPI of the program on P1?
   Sum of (number of clock cycles times respective percentage of instructions for each class of instruction)
   = 1*10%+2*20%+3*50%+3*20%
   = 2.6

2. What is the overall CPI of the program on P2?
   Sum of (number of clock cycles times respective percentage of instructions for each class of instruction)
   = 1*10%+1*20%+4*50%+5*20%
   = 3.3

3. How many times faster is the program on P2 than on P1? Note that the clock rate is different on P1 and P2.
   P2 is *n* times faster than P1
   n = performance of P2 / performance of P1
   performance = 1 / CPU time (execution time)
   CPU time (execution time) = clock cycles / clock rate
   = execution time of P1 / execution time of P2
   execution time of P1 = 2.6 clock cycles / 2 GHz clock rate = 1.3
   execution time of P2 = 3.3 clock cycles / 3 GHz clock rate = 1.1
   n = 1.3/1.1 = 1.18

4. Suppose a compiler can optimize the program, replacing all class D instructions with class A instructions. Each class D instruction requires two class A instructions. What is the average CPI of the program on P2 after the optimization?
   Multiply the percent of D by 2 and add to percent of A since D = 2A and we are replacing D with A.
   2*.2+.1 = .5 for percent of A in decimal form
   CPI = (.5*1+1*.2+4*.5)/.5+.2+.5
   = 2.7/1.2 = 2.25

5. What is the speedup the compiler in d) can achieve on processor P2?
   Let P2O be P2 optimized P2O is n times faster than P2
   n = CPU time of P2 / CPU time of P2O
   CPU time of P2 = 3.3 (from part 2)
   CPU time of P2O = 2.25 * 1.2 = 2.7
   n = 3.3/2.7 = 1.22

6

Suppose you have two different methods to accelerate a program. Method 1 can accelerate 20% of the program 100 times. Method 2 can accelerate 20% of the program 10 times and 15% of the program 6 times. The part of code enhanced by each method does not overlap.
Round answers to the nearest hundredth if necessary. For example, 1/2 to 0.5, 2/3 to 0.67.

1. What is the speedup Method 1 can achieve on the entire application?
   The best speedup method 1 can achieve can be described by Amdhal's law which is $1/((1-p)+p/x)$ where x is the number of times the program is to be accelerated and p is the percentage of the program that can be

accelerated.
when we substitute into the equation we get the following:
1/((1-20%)+20%/100) = 1/.802 = 1.25

2. What is the speedup Method 2 can achieve on the entire application?
This is not much different from the method in part 1 but now we sipmly subtract both percentages from 1 and add the fractions of percent to number of times accelerated in a single fraction.
max speed up = 1/(1-20%-15%+(20%/10)+(15%/6))
= 1/(.65+.02+.25) = 1.44

3. What is the speedup if both methods are applied?
We can apply a very similar method for this part as we did for part 2.
max speed up = 1/(1-20%-15%-20%+(20%/100)+(20%/10)+(15%/6))
= 1/(.45+.02+.025+.002) = 1/.497 = 2.012

4. After both methods are applied, what is the best speedup (or the upper bound) one can achieve if they continue to optimize the code that is already enhanced by the two methods? Note that the reference is the code after both methods are applied.
After both methods had been applied the execution time for unoptimized code was 1-20%-20%-15% = .45
and the execution time for the optimized code was 20/100+20/10+15/6 = .047
To find how much more we could optimize the code we calculate the sum of the execution times of the unoptimized and the execution time of the optimized all divided by the ounptimized.
= (.45+.047)/.45 = 1.104