Policy Reminders

- Include your CS username on your page. You will lose a few points from your score if you do not include it.

- You are allowed to work with other students on this homework, as we will not be grading it for correctness. However, each student must turn in their own copy of the homework.

- Show your work for all problems. While we won’t be grading for correctness, you will not receive full credit unless you show your work. After all, showing your work is required on the test - and homeworks are intended to help you practice for the test!

Required Problems:

1(e), 2(e), 3(c)

Problem 1 - Interpreting Bit Patterns

For each bit pattern below, what does it represent, assuming that it is

- a two’s complement integer?
- an unsigned integer?
- a single precision floating-point number?
- four ascii characters?

1(a)

1100 0011 1010 0000 0000 0000 0000 0000

1(b)

0000 0000 0000 0000 0000 0000 0000 0000

1(c)

0011 1100 0011 0000 0000 0000 0000 0000
1(d)
HINT: This number has lots of bits turned on! When converting to an integer, binary would work, but hex will be much easier. You are strongly encouraged to use hex.

1011 1110 0100 0000 0000 1010 0010 0000

1(e) - Turn in this one

0100 0001 0100 0010 0100 0011 0000 0000

Problem 2 - Encoding Floating Point

NOTE: Based on the Example in Section 3.5 (page 201)

Show the IEEE754 binary representation for the floating point number in:

- single precision
- double precision

2(a)
10.0_{ten}

2(b)
1024.5_{ten}

2(c)
-42.3125_{ten}

2(d)
123.8125_{ten}

2(e) - Turn in this one

3952_{ten}
HINT: With large even numbers, try dividing by 2 (and then again, and then again) to make the number easier to convert to binary.
Problem 3 - Cache Design

In each part below, we will give you the basic parameters of a cache (cache line size and number of cache lines). Consider three different possible cache configurations: direct mapped, 2-way set associative, and 4-way set associative. For each, calculate the following values:

- Number of sets
- Number of bits for the tag
- Total bits per cache line
- Total bits to store all cache lines
- Total capacity of the cache (not counting overhead)

In all cases, assume that addresses are 32 bits.

3(a)

\[ \text{cacheLineSize} = 32 \text{ bits} = 4 \text{ bytes} = 2^2 \text{ bytes} \]
\[ \text{cacheLineCount} = 1024 = 2^{10} \]

3(b)

\[ \text{cacheLineSize} = 1024 \text{ bits} = 128 \text{ bytes} = 2^7 \text{ bytes} \]
\[ \text{cacheLineCount} = 1024 = 2^{10} \]

3(c) - Turn in this one

\[ \text{cacheLineSize} = 256 \text{ bits} = 32 \text{ bytes} = 2^5 \text{ bytes} \]
\[ \text{cacheLineCount} = 16384 = 2^{14} \]
EXAMPLES

Example: Problem 1(a)

1100 0011 1010 0000 0000 0000 0000 0000

2’s complement integer

The high bit is 1, so the number is negative. We must invert and add one, then we’ll convert it:

\[ x = 1100 \ 0011 \ 1010 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ \text{(negated)} \]
\[ 0011 \ 1100 \ 0110 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ \text{(plus one)} \]

\[ -x = 2^{29} + 2^{28} + 2^{27} + 2^{26} + 2^{22} + 2^{21} \]
\[ -x = 536,870,912 + 268,435,456 + 134,217,728 + 67,108,864 + 4,194,304 + 2,097,152 \]
\[ -x = 1,012,924,416 \]
\[ x = -1,012,924,416 \]

Same thing, using hex:

\[ -x = 3 \cdot 16^7 + 12 \cdot 16^6 + 6 \cdot 16^5 \]
\[ -x = 805,306,368 + 201,326,592 + 6,291,456 \]
\[ -x = 1,012,924,416 \]
\[ x = -1,012,924,416 \]

unsigned integer

\[ x = 1100 \ 0011 \ 1010 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \]

\[ x = 2^{31} + 2^{30} + 2^{25} + 2^{24} + 2^{23} + 2^{21} \]
\[ x = 2,147,483,648 + 1,073,741,824 + 33,554,432 + 16,777,216 + 8,388,608 + 2,097,152 \]
\[ x = 3,282,042,880 \]

Same thing, using hex:

\[ x = 12 \cdot 16^7 + 3 \cdot 16^6 + 10 \cdot 16^5 \]
\[ x = 3,221,225,472 + 50,331,648 + 10,485,760 \]
\[ x = 3,282,042,880 \]
single-precision floating-point number

\[
\begin{array}{cccccccccccc}
1 & 100 & 0011 & 1 & 010 & 0000 & 0000 & 0000 & 0000 & 0000 & 0000 & 0000 & 0000 \\
\end{array}
\]

\[ s \ eee \ eee \ e \ fff \ fff \ ffff \ ffff \ ffff \ ffff \]

\( s = 1, \) the number is negative.

\( \text{exponent} = 1000 \ 0111 \)

\[ e = 2^7 + 2^2 + 2^1 + 2^0 \]

\( e = 128 + 4 + 2 + 1 \)

\( e = 135 \)

subtract the bias: \( 135 - 127 = 8 \)

\( \text{fraction} = 010 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \)

\( \text{mantissa} = 1.010_{\text{bin}} = 1.25_{\text{ten}} \)

\( \text{value} = -1 \cdot 2^8 \cdot 1.25 = -320 \)

four ASCII characters

\[ 1100 \ 0011 \ - \text{high bit is 1, not an ASCII character} \]

\[ 1010 \ 0000 \ - \text{high bit is 1, not an ASCII character} \]

\[ 0000 \ 0000 \ - \text{ASCII nul} \]

\[ 0000 \ 0000 \ - \text{ASCII nul} \]
Example: Problem 1(b)

0000 0000 0000 0000 0000 0000 0000 0000

2’s complement integer
The high bit is 0, so the number is positive.
\[ x = 0 \]

unsigned integer
Since the high bit is 0, the number has the same value, whether interpreted as a signed or an unsigned value.
\[ x = 0 \]

single-precision floating-point number

\[
\begin{array}{cccccccccccccc}
0 & 000 & 0000 & 0 & 000 & 0000 & 0000 & 0000 & 0000 & 0000 & 0000 & 0000 & 0000 & 0000 & 0000 \\
\end{array}
\]

This is the special encoding for zero.

four ASCII characters

\[
\begin{array}{cccccccccccccc}
0000 & 0000 & - & ASCII & nul \\
0000 & 0000 & - & ASCII & nul \\
0000 & 0000 & - & ASCII & nul \\
0000 & 0000 & - & ASCII & nul \\
\end{array}
\]
Example: Problem 1(c)

0011 1100 0011 0000 0000 0000 0000 0000

2’s complement integer

The high bit is 0, so the number is positive.

\[ x = 2^{29} + 2^{28} + 2^{27} + 2^{26} + 2^{21} + 2^{20} \]
\[ x = 536,870,912 + 268,435,456 + 134,217,728 + 67,108,864 + 2,097,152 + 1,048,576 \]
\[ x = 1,009,778,688 \]

Same thing, using hex:

\[ x = 3 \cdot 16^7 + 12 \cdot 16^6 + 3 \cdot 16^5 \]
\[ x = 805,306,368 + 201,326,592 + 3,145,728 \]
\[ x = 1,009,778,688 \]

unsigned integer

(Since the high bit is zero, the unsigned and signed interpretations are identical.)
\[ x = 1,009,778,688 \]

single-precision floating-point number

\[
\begin{array}{cccccccccccc}
0 & 011 & 1100 & 0 & 011 & 0000 & 0000 & 0000 & 0000 & 0000 & 0000 & 0000 \\
& s & eee & eeee & e & fff & ffff & ffff & ffff & ffff & ffff & ffff
\end{array}
\]

\[ s = 0, \text{ the number is positive.} \]

exponent = 0111 1000
\[ e = 2^6 + 2^5 + 2^4 + 2^2 \]
\[ e = 64 + 32 + 16 + 8 \]
\[ e = 120 \]
subtract the bias: 120 − 127 = −7

fraction = 011 0000 0000 0000 0000 0000

mantissa = 1.011_{bin} = 1.375_{ten}

value = 2^{-7} \cdot 1.375 = .0107421875

four ASCII characters

\[
\begin{array}{cccccccccccc}
0011 & 1100 & - & 0x3c & = & '\leq' \\
0011 & 0000 & - & 0x30 & = & '\0' \\
0000 & 0000 & - & ASCII & nul \\
0000 & 0000 & - & ASCII & nul
\end{array}
\]
Example: Problem 1(d)

1011 1110 0100 0000 0000 1010 0010 0000

2’s complement integer
The high bit is 1, so the number is negative.

\[
x = 1011 \ 1110 \ 0100 \ 0000 \ 0000 \ 1010 \ 0010 \ 0000 \ \ (\text{negated})
\]

\[
x = 0100 \ 0001 \ 1011 \ 1111 \ 1111 \ 0101 \ 1101 \ 1111 \ \ (\text{plus one})
\]

This would be very long to do in binary, I’m skipping that!

Using hex:

\[
-x = 4 \cdot 16^7 + 1 \cdot 16^6 + 11 \cdot 16^5 + 15 \cdot 16^4 + 15 \cdot 16^3 + 5 \cdot 16^2 + 14 \cdot 16^1
\]

\[
x = 1,073,741,824 + 16,777,216 + 11,534,336 + 983,040 + 61,440 + 1,280 + 224
\]

\[
x = 1,103,099,360
\]

\[
x = -1,103,099,360
\]

unsigned integer

\[
x = 11 \cdot 16^7 + 14 \cdot 16^6 + 4 \cdot 16^5 + 10 \cdot 16^2 + 2 \cdot 16^1
\]

\[
x = 2,952,790,016 + 234,881,024 + 4,194,304 + 2,560 + 32 \ x = 3,191,867,936
\]

single-precision floating-point number

\[
1 \ 011 \ 1110 \ 0 \ 100 \ 0000 \ 0000 \ 1010 \ 0010 \ 0000 \ s \ eee \ eeee \ e \ fff \ ffff \ ffff \ ffff
\]

\[
s = 1, \text{ the number is negative.}
\]

exponent = 0111 1100
\[
e = 2^6 + 2^5 + 2^4 + 2^3 + 2^2
\]

\[
e = 64 + 32 + 16 + 8 + 4
\]

\[
e = 124
\]

subtract the bias: 124 – 127 = –3

fraction = 100 0000 0000 1010 0010 0000

mantissa = 1.100 0000 0000 1010 0010 \_bin

mantissa = 1 + 2^{-1} + 2^{-12} + 2^{-14} + 2^{-18}

mantissa = 1 + .5 + 0.000244140625 + 0.00006103515625 + 0.000003814697265625

mantissa = 1.500308990478515625

value = –1 \cdot 2^{-3} \cdot 1.500308990478515625 = –0.187538623809814453125
four ASCII characters

1011 1110 - high bit is 1, not ASCII
0100 0000 - ASCII '@'
0000 1010 - ASCII '\n'
0010 0000 - ASCII space
Example: Problem 2(a) - 10.0_{ten}

The first part of the conversion is common; both single and double precision use the same values.

Number is positive, so $s = 0$.

$10_{ten} = 1010_{bin} = 1.010_{bin} \cdot 2^3$

Mantissa: $1.0100 \ 0000 \ldots$

Fraction: $0100 \ 0000 \ldots$

**Single Precision**

Exponent = 3; with bias: $3 + 127 = 130_{ten} = 1000010_{bin}$

```
s  eee  eee  e  fff  ffff  ffff  ffff  
0  100  0001 0  010  0000 0000 0000 0000
```

**Double Precision**

Exponent = 3; with bias: $3 + 1023 = 1026_{ten} = 1000000010_{bin}$

```
s  eee  eee  eee  ffff  ffff  ffff  ffff  ffff  ffff  ffff  ffff  ffff  ffff  ffff  ffff  
0  100  0000 0010 0100 0000 0000 0000 0000 0000 0000 0000 0000 \ldots
```
Example: Problem 2(b) - 1024.5\textsubscript{ten}

The first part of the conversion is common; both single and double precision use the same values.

Number is positive, so \( s = 0 \).

\[
1024.5 = 1024 + .5 = 2^{10} + 2^{-1}
\]

\[
1024.5\textsubscript{ten} = 10000000000.1\textsubscript{bin} = 1.0000000001\textsubscript{bin} \cdot 2^{10}
\]

Mantissa: 1.00 0000 0000 1...

Fraction: 0000 0000 0010 ...

**Single Precision**

Exponent = 10; with bias: 10 + 127 = 137\textsubscript{ten} = 10001001\textsubscript{bin}

```
  s eee eeee e  fff ffff ffff ffff ffff
0 100 0100 1 000 0000 0001 0000 0000
```

**Double Precision**

Exponent = 10; with bias: 10 + 1023 = 1033\textsubscript{ten} = 10000001001\textsubscript{bin}

```
  s eee eeee eee  fff ffff ffff ffff ffff ffff ffff ffff ffff ffff ffff ffff ffff
0 100 0000 1001 0000 0000 0010 0000...
```
Example: Problem 2(c) - $-42.3125_{ten}$

The first part of the conversion is common; both single and double precision use the same values.

Number is negative, so $s = 1$.

$42.3125 = 32 + 8 + 2 + .25 + .0625 = 2^5 + 2^3 + 2^1 + 2^{-2} + 2^{-4}$

$42.3125_{ten} = 101010.0101_{bin} = 1.010100101 \cdot 2^5$

Mantissa: $1.010100101\ldots$

Fraction: 0101 0010 1... 

**Single Precision**

Exponent = 5; with bias: $5 + 127 = 132_{ten} = 10000100_{bin}$

```
 s eee eee e fff ffff ffff ffff ffff ffff ffff
 1 100 0010 0 010 1001 0100 0000 0000
```

**Double Precision**

Exponent = 5; with bias: $5 + 1023 = 1028_{ten} = 10000000100_{bin}$

```
 s eee eee e eee ffff ffff ffff ffff ffff ffff ffff ffff ffff ffff ffff ffff
 1 100 0000 0100 0101 0010 0010 1000 0000 0000 0000 ... 
```
Example: Problem 2(d) - 123.8125\text{ten}

The first part of the conversion is common; both single and double precision use the same values.

Number is positive, so \( s = 0 \).

\[
123.8125 = 64 + 32 + 16 + 8 + 2 + 1 + .5 + .25 + .0625 \\
123.8125 = 2^6 + 2^5 + 2^4 + 2^3 + 2^1 + 2^0 + 2^{-1} + 2^{-2} + 2^{-4} \\
123.8125\text{ten} = 01110111.1101\text{bin} = 1.1110111101\text{bin} \cdot 2^6
\]

Mantissa: 1.1110111101...

Fraction: 1110 1111 01...

Single Precision

Exponent = 6; with bias: 6 + 127 = 133\text{ten} = 10000101\text{bin}

\[
s \ eee \ eee \ e \ fff \ ffff \ ffff \ ffff \ ffff \\
0 \ 100 \ 0010 \ 1 \ 111 \ 0111 \ 1010 \ 0000 \ 0000
\]

Double Precision

Exponent = 6; with bias: 6 + 1023 = 1029\text{ten} = 10000000101\text{bin}

\[
s \ eee \ eee \ eee \ fffff \ ffff \ ffff \ ffff \ ffff \ ffff \ ffff \ ffff \ ffff \ ffff \ ffff \ ffff \ ffff \ ffff \\
0 \ 100 \ 0000 \ 0101 \ 1110 \ 1111 \ 0100 \ 0000 \ ... 
\]
Example: Problem 3(a)

Cache line size = 32 bits = 4 bytes = \(2^2\) bytes
Cache line count = 1024 = \(2^{10}\)

**Direct-Mapped Cache**

- Number of sets = 1024 = \(2^{10}\)
  
  **Instructor’s Note:** In a direct-mapped cache, there is one line per set.

- Number of bits for the tag = 32 − 2 − 10 = 20
  
  **Instructor’s Note:** The byte offset is 2 bits, because there are 4 bytes per cache line. The set index is 10 bits, because there are \(2^{10}\) sets.

- Total bits per cache line = valid + tag + data = 1 + 20 + 32 = 53

- Total bits to store all cache lines = bits per line * lines = 53 * 1024 = 53 Kb

  **Instructor’s Note:** The abbreviation KB means “kilobytes.” Kb means “kilobits.” 53 Kb is roughly 6.6 KB.

- Total capacity of the cache, in bits = bits per data * lines = 32 * 1024 = 32 Kb

  **Instructor’s Note:** Note that the capacity is less than the total amount of actual hardware, because of overhead. With small cache lines, the overhead is very significant.

**2-way Set Associative Cache**

- Number of sets = \(\frac{1024}{2} = 2^9\)
  
  **Instructor’s Note:** 2 lines per set.

- Number of bits for the tag = 32 − 2 − 9 = 21

  **Instructor’s Note:** The byte offset is 2 bits, because there are 4 bytes per cache line. The set index is 9 bits, because there are \(2^9\) sets.

- Total bits per cache line = valid + tag + data = 1 + 21 + 32 = 54

- Total bits to store all cache lines = bits per line * lines = 54 * 1024 = 54 Kb

- Total capacity of the cache, in bits = bits per data * lines = 32 * 1024 = 32 Kb

**4-way Set Associative Cache**

- Number of sets = \(\frac{1024}{4} = 2^8\)

- Number of bits for the tag = 32 − 2 − 8 = 22

- Total bits per cache line = valid + tag + data = 1 + 22 + 32 = 55

- Total bits to store all cache lines = bits per line * lines = 55 * 1024 = 55 Kb

- Total capacity of the cache, in bits = bits per data * lines = 32 * 1024 = 32 Kb

  **Instructor’s Note:** Changing from a direct-mapped cache to a 4-way set associative adds to the overhead a tiny bit (because it makes the tag larger). But it is usually worthwhile, because set-associativity hugely reduces cache line contention!
Example: Problem 3(b)

Cache line size = 1024 bits = 128 bytes = 2^7 bytes
Cache line count = 1024 = 2\textsuperscript{10}

Direct-Mapped Cache

• Number of sets = 1024 = 2\textsuperscript{10}
• Number of bits for the tag = 32 − 7 − 10 = 15

\textbf{Instructor's Note:} The byte offset is 7 bits, because there are 128 bytes per cache line. The set index is 10 bits, because there are 2\textsuperscript{10} sets.

• Total bits per cache line = valid + tag + data = 1 + 15 + 1024 = 1040
• Total bits to store all cache lines = bits per line * lines = 1040 * 1024 = 1040 \text{ Kb}
• Total capacity of the cache, in bits = bits per data * lines = 1024 * 1024 = 1024 \text{ Kb}

\textbf{Instructor's Note:} When the cache line is large enough, the cost of the overhead is almost not worth talking about. However, large cache lines are often wasteful, storing data that the program actually won’t be using.

2-way Set Associative Cache

• Number of sets = \frac{1024}{2} = 2^9
• Number of bits for the tag = 32 − 7 − 9 = 16

\textbf{Instructor's Note:} The byte offset is 7 bits, because there are 128 bytes per cache line. The set index is 9 bits, because there are 2\textsuperscript{9} sets.

• Total bits per cache line = valid + tag + data = 1 + 16 + 1024 = 1041
• Total bits to store all cache lines = bits per line * lines = 1041 * 1024 = 1041 \text{ Kb}
• Total capacity of the cache, in bits = bits per data * lines = 1024 * 1024 = 1024 \text{ Kb}

4-way Set Associative Cache

• Number of sets = \frac{1024}{4} = 2^8
• Number of bits for the tag = 32 − 7 − 8 = 17

• Total bits per cache line = valid + tag + data = 1 + 17 + 1024 = 1042
• Total bits to store all cache lines = bits per line * lines = 1042 * 1024 = 1042 \text{ Kb}
• Total capacity of the cache, in bits = bits per data * lines = 1024 * 1024 = 1024 \text{ Kb}