Number Systems

Bits and Bytes

- Computers use 1s and 0s in a form called binary
- 10010011 <- binary number
- Each digit is a **B**inary dig**IT** or **BIT**
- 1s and 0s are represented by voltage levels
  - Typically: 1 is 5v, 0 = 0v
  - Zero is NOT an absence of voltage it is a connection to the circuit ground.
  - Different voltage levels can be used (3v, 12v, …)
  - There are ranges to voltage levels

    0 = 0v to .5v   1 = 4.5v to 5.5v
- 8 Bits is called a byte. Computers generally arrange things in terms of bytes.

Number Representations

- Decimal – normal base 10
- Binary – unsigned base 2
- 2's Complement – signed base 2
- Hexadecimal – base 16
- ASCII (UNICODE) – text
- Floating point – fractions or numbers with decimal points

Binary

- Convert decimal to binary.
  - Small numbers convert to groups of powers of 2

    55 = 32 + 16 + 4 + 2 + 1

32	16	8	4	2	1
1	1	0	1	1	1

  - Large numbers repeatedly divide by 2 and note remainder.

    1134

Convert decimal to binary

- Large numbers repeatedly divide by 2 and note remainder.

1134 / 2 = 567 r 0
567 / 2 = 283 r 1
283 / 2 = 141 r 1
141 / 2 = 70 r 1
70 / 2 = 35 r 0
35 / 2 = 17 r 1
17 / 2 = 8 r 1
8 / 2 = 4 r 0
4 / 2 = 2 r 0
2 / 2 = 1 r 0
1 / 2 = **0** r 1 <- **keep going until answer is zero**
Answer is remainders starting at bottom: 10001101110

Binary to Decimal

- Add up the column values

10010110

128	64	32	16	8	4	2	1
1	0	0	1	0	1	1	0

128 + 16 + 4 + 2 = 150

Representing Negative Numbers

- Signed Magnitude
  - Set sign bit for negative values
  - Easiest for humans, not so easy for computer
- One's Complement
  - Invert all bits for negative values
- Two's Complement
  - Invert all bits and add 1 for negative values
  - Easiest for computers, not so easy for humans
  - Most commonly used

Signed Magnitude

- Use the leading bit to indicate the sign

  0 means positive

  1 means negative
- We need to know how many bits
- Represent -15 as 6 bit signed magnitude

  15 = 8 + 4 + 2 + 1 = 1111 (convert as unsigned)

  001111 (write as six bits)

  101111 (1 in first position means negative)

2's Complement

- Allows both positive and negative numbers
- Works because of a fixed number of bits
- Assume you have only 5 bits to work with
- You can store 00000 to 11111
- Adding a number to its additive identity should give zero so 7 + (-7) = 0

```
  00111  (7)
 +11001  (?)
 ------
  00000  (0)
```

- 11001 must be -7

Decimal to 2's Complement

- If the number is positive
  - Treat as unsigned binary
- If the number is negative
  - Convert as unsigned
  - Invert all bits and add 1
- Convert 17 and -23 to 6 bit 2's complement

Decimal to 2's Complement Example

- Convert 17 and -23 to 6-bit 2's complement

  17 (Positive so treat as unsigned binary and done)
  17 = 16 + 1 = 010001 (make sure to write the number as six bits)

  Answer: 010001

  -23 (negative so convert as unsigned then do the following)
  23 = 16 + 4 + 2 + 1 = 010111 (six bits is important)
  101000 (invert all bits)
  101001 (add 1)

  Answer: 101001 (How can you verify this is -23?)

5-Bit Comparison

Number	Signed Magnitude	1's Complement	2's Complement
0	00000 or 10000	00000 or 11111	00000
1	00001	00001	00001
2	00010	00010	00010
3	00011	00011	00011
5	00101	00101	00101
15	01111	01111	01111
-1	10001	11110	11111
-2	10010	11101	11110
-5	10101	11010	11011
-11	11011	10100	10101
-15	11111	10000	10001
-16	-	-	10000

See Figure 2.1 in book

2's Complement Notes

- Positive numbers are exactly the same
- Half as many non-negative integers as unsigned
- -1 is always all 1s no matter how many bits

  (111111 = -1 with six bits)
- The most negative number is always leading 1 followed by zeros

  (100000 is -32 with six bits)
- There are the same number of negative and non-negative values
- The range is $(+2^{k-1}-1)$ to (-2^{k-1})
- What is the range for 8 bits?

Binary	Unsigned	2's Complement
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	-8
1001	9	-7
1010	10	-6
1011	11	-5
1100	12	-4
1101	13	-3
1110	14	-2
1111	15	-1

2's Complement Shortcut

- Invert all bits and add 1
- The following number does something special

  10**100000** Decimal

  01011111 Invert all bits.

  01**100000** Add 1
- When we invert and add 1 we get the same pattern in the bold bits above as we started with.
- Shortcut – Keep zero bits from right to left up to and including the first 1. Invert everything else.
- Note that you don't add 1 if you use this shortcut.

2s Complement Shortcut Practice

- Convert these to decimal using the shortcut

    10100000

    10001100

    00110000

    11111101

2s Complement Shortcut Practice

- Convert these to decimal using the shortcut

  *10***100000** = *01***100000** = -96

  *10001***100** = *01110***100** = -116

  00110000 = **00110000** = +48 (Positive)

  *1111110***1** = 00000011 = -3

Bold bits are kept : Highlightedbits are inverted.

Another Shortcut

- Add up the column values for bits that are 1, just like unsigned.
- HOWEVER, if the high order bit is 1 (i.e., the number is negative) then subtract that column value.
- 6-bit 2s complement: 100110

  100110 is negative

  Add -32+4+2 = -26
- Old way, invert and add 1 to get 011010

  11010 = 16+8+2 = 26 so answer is -26

Biggest and smallest number

- What is the biggest number you can store using unsigned binary and 12 bits?

  2^n-1 (where n = number of bits)

- What is the biggest number you can store using 2's complement binary and 12 bits? Why n-1 below?

  $2^{(n-1)}-1$ (where n = number of bits)

- What is the smallest number you can store using unsigned binary and 12 bits?

  Always 0

- What is the smallest number you can store using 2's complement binary and 12 bits?

  $-2^{(n-1)}$ (where n = number of bits)

Biggest and smallest number

- What is the biggest number you can store using unsigned binary and 12 bits?

  2^{12}-1 = 4095

- What is the biggest number you can store using 2's complement binary and 12 bits?

  $2^{(12-1)}$-1 = $2^{(11)}$-1 = 2047

- What is the smallest number you can store using unsigned binary and 12 bits? Smallest will always be zero.

  0
- What is the smallest number you can store using 2's complement binary and 12 bits? Smallest is -2n.

  -$2^{(12-1)}$ = -$2^{(11)}$ = -2048

How many bits to represent a number?

- How many bits do you need to represent +111 in unsigned numbers?
- How many bits do you need to represent -111 in unsigned numbers?
- How many bits do you need to represent +111 in 2's complement numbers?
- How many bits do you need to represent -111 in 2's complement numbers?

How many bits to represent a number?

- How many bits do you need to represent +111 in unsigned numbers?

  Next higher power of 2 = 128 = $2^{\mathbf{7}}$ = **7**

- How many bits do you need to represent -111 in unsigned numbers?

  You cant.

- How many bits do you need to represent +111 in 2's complement numbers?

  Next higher power of 2 = 128 = 2^7 = 2's comp always needs 1 more = **8**

- How many bits do you need to represent -111 in 2's complement numbers?

  Next higher power of 2 = 128 = 2^7 = 2's comp always needs 1 more = **8**

  It doesn't matter if the number you want to represent is + or - , 2's comp always needs one more.

Sign Extension Signed Positive

- +5 in 4-bit 2's complement is

  0101
- +5 in 8-bit 2's complement is

  00000101
- +5 in 16-bit 2's complement is
  0000000000000101
- What about 32-Bit 2's complement?

Sign Extension Signed Negative

- -5 in 4-bit 2's complement is

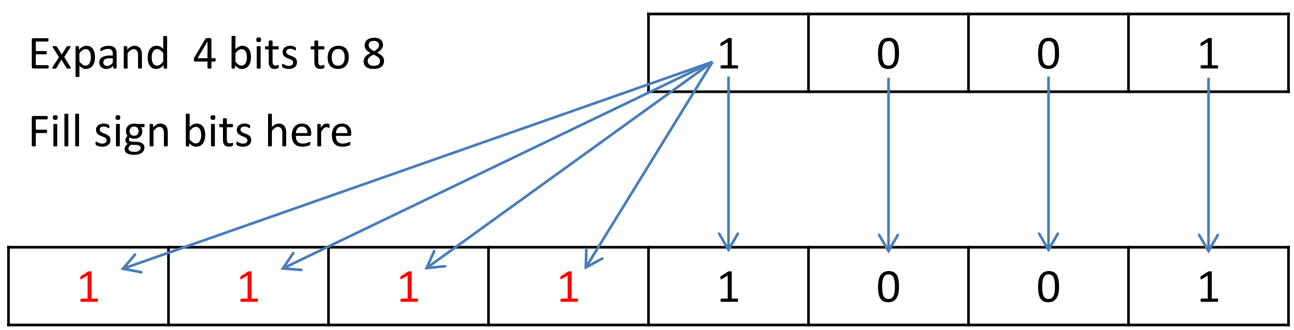
  1011
- -5 in 8-bit 2's complement is

  11111011
- -5 in 16-bit 2's complement is

  1111111111111011
- What about 32Bit 2's complement?

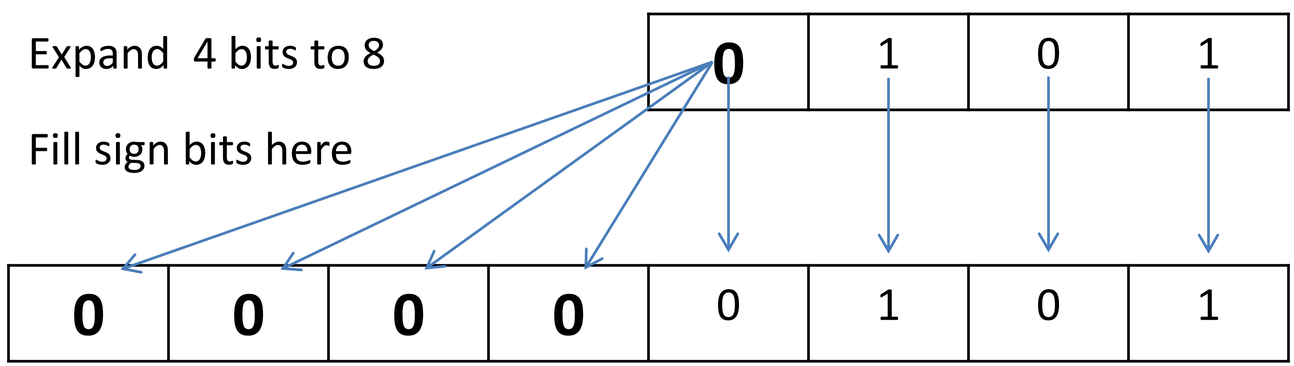
Sign Extension Negative Example

- When expanding a signed value into a larger storage size, the sign must be extended.
- Whatever the old sign bit was must fill all of the new empty bits in the bigger number



Sign Extension Positive Example

- When expanding a signed value into a larger storage size, the sign must be extended.
- Whatever the old sign bit was must fill all of the new empty bits in the bigger number



Zero Extension

- Zero extension is placing zeros in the new expanded bits regardless of sign.
- Unsigned numbers are always zero extended.

Expand 4 bits to 8

1	1	0	1

Zero fill here

↓ ↓ ↓ ↓

0	**0**	**0**	**0**	1	1	0	1

Unsigned Overflow

- Adding 4-bit unsigned numbers

Can't store carry out->	***0***	1	0	0		<-Carry
No Overflow		0	1	0	1	(5)
+		0	1	1	0	(6)
		1	0	1	1	(11)

Can't store carry out->	***1***	1	0	0		<-Carry
Overflow		1	1	0	0	(12)
+		0	1	1	1	(7)
		0	0	1	1	(3)

For unsigned number (NOT FOR SIGNED) you can detect overflow by looking at the carry out bit.

Signed Overflow Positive

- Adding 4-bit signed numbers

Can't store carry out->	***0***	0	0	0		<-Carry
3+7 = 7 (No Overflow)		0	0	1	1	(3)
+		0	1	0	0	(4)
		0	1	1	1	(7)

Can't store carry out->	***0***	1	0	0		<-Carry
5+6 $\neq$-5 (Overflow)		0	1	0	1	(5)
+		0	1	1	0	(6)
		1	0	1	1	(-5)

NOTE that there is no carry out in the second example, yet overflow occurred. For signed numbers YOU CANNOT DETECT OVERFLOW USING THE CARRY OUT.

Signed Overflow Negative

- Adding 4-bit signed numbers

Can't store carry out->	***1***	1	0	0		<-Carry
-3 + -2 = -5 (No Overflow)		1	1	0	1	(-3)
+		1	1	1	0	(-2)
		1	0	1	1	(-5)

Can't store carry out->	***1***	0	1	0		<-Carry
-5 + -6 ≠ +5 (Overflow)		1	0	1	1	(-5)
+		1	0	1	0	(-6)
		0	1	0	1	(+5)

NOTE there IS A CARRY OUT first example even though overflow did not occur.
For signed numbers YOU CANNOT DETECT OVERFLOW USING THE CARRY OUT.

Subtraction Example No Overflow

- Many processors don't have a subtraction operation. They simply add a negative.

	1	0	0		<-Carry
	0	1	0	1	(5)
-	0	0	1	1	(3) (subtracting +3 is same as adding -3)
	?	?	?	?	(2)

- Subtracting +3 is the same as adding -3

	1	0	1		<-Carry
	0	1	0	1	(5)
+	1	1	0	1	(-3) (subtracting +3 is same as adding -3)
	0	0	1	0	(2)

Subtraction Example Overflow

- Many processors don't have a subtraction operation. They simply add a negative.

	1	0	0		<-Carry
	1	0	1	1	(-5)
-	0	1	0	0	(4) (subtracting 4 is same as adding -4)
	?	?	?	/	(2)

- Determine overflow by converting subtraction to addition.

	0	0	0		<-Carry
	1	0	1	1	(-5)
+	1	1	0	0	(-4) (subtracting 4 is same as adding -4)
	0	1	1	1	Negative + Negative = Positive = Overflow

Detecting overflow

- Unsigned: A carry out of 1 is always an overflow.
- Signed – Computer: If the carry in and the carry out of the most significant bit are different, overflow has occurred.
- Signed – Human: If you add two positive numbers and get a negative result or you add two negative numbers and you get a positive result, overflow has occurred.
- For subtraction, think of x – y as x + (-y) and use the above rules accordingly.
- Note: You can never have overflow when adding a negative and a positive number.

Bit Twiddling

- Sometimes you want to set specific bits inside of a binary number.
- Sometimes you want to examine specific bits inside of a binary number.
- Bitwise operators allow manipulation and examination of specific bits.
- Operators are: and, or, xor, not, shift left, shift right.

Operators

Operator	**Symbol**	**Example**
And	&	And each bit: 1010 & 1100 = 1000
Or	|	Or each bit: 1010 | 1100 = 1110
Xor	^	Xor each bit: 1011 ^ 1001 = 0010
Right Shift (Sign Extends for signed numbers. Zero extends unsigned numbers)	>>	Shift all bits to the right a given number of bits 0100 >> 1 = 0010 0100 >> 2 = 0001 0110 >> 2 = 0001 (bits fall off end) 1100 >> 2 = 1111 (signed numbers) 1100 >> 2 = 0011 (unsigned numbers)
Left Shift	<<	Shift all bits to the left a given number of bits 0011 << 1 = 0110 0011 << 2 = 1100 1011 << 2 = 1100 (bits fall off end)
Not	~	Invert all bits ~1010 = 0101

Truth Tables

A	B	AND
0	0	0
0	1	0
1	0	0
1	1	1

A	B	OR
0	0	0
0	1	1
1	0	1
1	1	1

A	B	XOR
0	0	0
0	1	1
1	0	1
1	1	0

A	NOT
0	1
1	0

Clearing bits with AND

- 0110100***1010***01000
- Clear the indicated bits to all zeros.
- Boolean laws

  $x \& 0 = 0$

  $x \& 1 = x$

A	B	AND
0	0	0
0	1	0
1	0	0
1	1	1

```
  0110100 1010 11000
& 1111111 0000 11111   <- Known as a mask
  ------------------
  0110100 0000 11000
```

To clear bits (clear means make the bit zero) use bitwise and (&).
Put 1s in the mask where you want the bit to remain unchanged.
Put 0s in the mask where you want to make bits zero.

Clearing Bits Practice

- Show how to clear the indicated bits.

0110100***101***011000

0110100101011000

01***10***1001010***11***000

A	B	AND
0	0	0
0	1	0
1	0	0
1	1	1

Setting bits with OR

- 0110100***1010***01000
- Set the indicated bits to all ones.
- Boolean laws

  $x \mid 0 = x$

  $x \mid 1 = 1$

A	B	OR
0	0	0
0	1	1
1	0	1
1	1	1

```
  0110100101011000
| 0000000111100000
  ----------------
  0110100111111000
```

To set bits (set means make the bit one) use bitwise or (|).
Put 1s in the mask where you want to make the bit one.
Put 0s in the mask where you want the bit to remain unchanged.

Setting Bits Practice

- Show how to set the indicated bits.

0110100***101***011000

0110100101011000

01***10***100101***01***1000

A	B	OR
0	0	0
0	1	1
1	0	1
1	1	1

Looking at bits

- Sometimes you just want to look at a subset of bits.
- If I want to look only at the bits below, I can create a mask and use the AND operation

```
 0110100*1010*11000(Number)
&0000000111100000(Mask)
 0000000*1010*00000(Result. Is this 10?)
```

- To interpret the value above as 10, I also need to shift. (Result >> 5. Why 5?)

```
 000000000000*1010*(Shifted Result)
```

- Note the shifted Result is now 10.

Bit Vector Example

- Each bit represents the presence or absence of an item.
- A = { b, d, m, z}
- Represent A as a bit vector where the domain is all letters.
- Assume each letter is positional a=1, b=2, c=3, etc.
- How many bits will we need?
- How many bytes?
- What data type to use in Java?

Bit Vector Solution

- Each bit represents the presence or absence of an item.
- A = { *b, d, m, z*}
- Represent A as a bit vector where the domain is all letters.
- Assume each letter is positional *a=1, b=2, c=3*, etc.
- How many bits will we need? 26
- How many bytes? 4
- What data type to use in Java? int

```
abcdefghijklmnopqrstuvwxyz------
01010000000010000000000001000000
```

Bit Vector Practice

- Assuming the letters bit vector what set would the following represent?

A=11010001000000000000000000000000

B=10000101001000000000000000000000

- How could we calculate C if C= A $\cup$ B
- How could we remove element *a* from B?
- How could we add *y* and *z* to B?

Clearing all bits with XOR

What is the result?

```
 00010010000100010011100010100100
^00010010000100010011100010100100
```

int a = 25;

a = a ^ a;

A	B	XOR
0	0	0
0	1	1
1	0	1
1	1	0

Checking equality with XOR

What is the result?

```
a = 00000000000000000000000000000100
b = 00000000000000000000000000000111
a ^ b = ?
    lda a
    xor b
    bz  addr
```

A	B	XOR
0	0	0
0	1	1
1	0	1
1	1	0

Sign Extension Example

This is an example of an LC3 instruction.
Bits 0 through 8 represent a 2's complement number.
What is that 2's complement number?

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Opcode				DR or SR			2's comp value added to PC								
1	0	1	0	0	1	0	1	1	1	1	1	0	1	0	0

Sign Extension Example

The number is -12.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Opcode				DR or SR			2's comp value added to PC								
1	0	1	0	0	1	0	1	1	1	1	1	0	1	0	0

Which of these will give the correct value?

offset_value = instruction & 0x01FF

offset_value = instruction | 0xFE00

Sign Extension Example

The number is +12.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Opcode				DR or SR			2's comp value added to PC								
1	0	1	0	0	1	0	0	0	0	0	0	1	1	0	0

Which of these will give the correct value?

offset_value = instruction & 0x01FF

offset_value = instruction | 0xFE00

Sign Extension Example

The number is +12.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Opcode				DR or SR			2's comp value added to PC								
1	0	1	0	0	1	0	0	0	0	0	0	1	1	0	0

To be able to sign extend you must check the high order bit.

```
if (instruction & 0x0100 == 0)
        offset_value = instruction & 0x01FF
else
        offset_value = instruction | 0xFE00
```

Octal – Base 8

- Convert decimal to octal
  - Repeatedly divide by 8.
- Convert octal to decimal
  - Sum of column digits x column values.
- Convert octal to binary
  - Write each digit as 3 bit binary
- Convert binary to octal
  - Convert each group of three bits, starting from least significant, to a single octal digit.

Hexadecimal – base 16

- Sometimes called simply Hex
- Convert decimal to hexadecimal
  - Repeatedly divide by 16.
- Convert hexadecimal to decimal
  - Sum of column digits x column values.
- Convert hexadecimal to binary
  - Write each digit as 4 bit binary
- Convert binary to hexadecimal
  - Convert each group of four bits, starting from least significant, to a single hex digit.

Convert the following

- 456_{10} to octal.
- 274_8 to decimal.
- 101010001010 to octal.
- 274_8 to binary.

- 5126_{10} to hexadecimal.
- $AB4_{16}$ to decimal.
- 101010001010 to hexadecimal.
- $AB4_{16}$ to binary.

Using Hexadecimal

- In many programming languages, hex values are preceded by 0x

  0x123 or 0X123

- Your book simply uses an x

  x123 or X123

- Hexadecimal is a convenient way of representing binary information, not just binary numbers.

- For example, the following binary could represent pixel data, part of an integer, a 16-bit floating point number, or the word Hi in ascii.

  0100100001101001

- Regardless of its meaning, the value can still be stored for convenience as hexadecimal as the value

  x4869

Powers of 2

- $2^0 = 1$
- $2^1 = 2$
- $2^2 = 4$
- $2^3 = 8$
- $2^4 = 16$
- $2^5 = 32$
- $2^6 = 64$
- $2^7 = 128$
- $2^8 = 256$
- $2^9 = 512$
- $2^{10} = 1024$
- $2^{11} = 2048$
- $2^{12} = 4096$
- $2^{13} = 8192$

Metric Prefixes

- When talking about computer memory, we use the following generalizations.

- 2^{10} = 1K (~One thousand)
- 2^{20} = 1M (~One Million)
- 2^{30} = 1G (~One Billion)
- 2^{40} = 1T (~One Trillion)

Metric Prefixes

Prefix	Computers	Engineering
Kilo (K)	2^{10} = 1024	10^{3} = 1,000
Mega (M)	2^{20} = 1,048,576	10^{6} = 1,000,000
Giga (G)	2^{30} = 1,073,741,824	10^{9} = 1,000,000,000
Tera (T)	2^{40} = 1,099,511,627,776	10^{12} = 1,000,000,000,000
Peta (P)	2^{50} = 1,125,899,906,842,624	10^{15} = 1,000,000,000,000,000
Exa (E)	2^{60} = billion billion	10^{18} = 1 followed by 18 zeros
Zetta(Z)	2^{70} = 1024 billion billion	10^{21} = 1 followed by 21 zeros
Yotta (Y)	2^{80} = million billion billion	10^{24} = 1 followed by 24 zeros

Metric Prefixes

- The prefixes we will use

  2^{10} = K (thousand)

  2^{20} = M (million)

  2^{30}=G (billion)

  2^{40}=T (trillion)
- Converting bits to prefix

  35 bits = $2^{35} = 2^{5}x2^{30}$ = 32G

  22 bits = $2^{22} = 2^{2}x2^{20}$ = 4M
- Converting prefix to bits

  8G = $2^{3} \text{ x } 2^{30} = 2^{33}$ = 33 bits

  32K = $2^{5} \text{ x } 2^{10} = 2^{15}$ = 15 bits

Metric Prefixes and Biggest/Smallest

- Biggest values unsigned: 2^n - 1

  With 35 bits = $2^{35} – 1 = 2^5 \times 2^{30} - 1$= 32G-1

  With 22 bits = $2^{22} – 1 = 2^2 \times 2^{20} - 1$= 4M-1

- Smallest values unsigned: always zero

  With 35 bits = 0

  With 22 bits = 0

Metric Prefixes

- $2^{15} = 2^5 * 2^{10} = 32K$ (since $2^5=32$ and $2^{10}= 1K$)
- $2^8 = 256$
- $2^{26} = 2^6 * 2^{20} = 64M$
- $2^{31} = 2^1 * 2^{30} = 2G$
- $2^{48} = 2^8 * 2^{40} = 256T$
- Anytime I ask for an answer using metric prefixes, you must use the above form if the value is > 8192

Metric Prefixes

- Try some

2^{16}	2^{37}	2^{29}	2^{45}
2^{22}	2^{13}	2^{5}	2^{44}

Powers of 2

- $2^0 = 1$
- $2^1 = 2$
- $2^2 = 4$
- $2^3 = 8$
- $2^4 = 16$
- $2^5 = 32$
- $2^6 = 64$
- $2^7 = 128$
- $2^8 = 256$
- $2^9 = 512$

- $2^{10} = 1K$
- $2^{11} = 2K$
- $2^{12} = 4K$
- $2^{13} = 8K$
- $2^{14} = 16K$
- $2^{15} = 32K$
- $2^{16} = 64K$
- $2^{17} = 128K$
- $2^{18} = 256K$
- $2^{19} = 512K$

- $2^{20} = 1M$
- $2^{21} = 2M$
- $2^{22} = 4M$
- $2^{23} = 8M$
- $2^{24} = 16M$
- $2^{25} = 32M$
- $2^{26} = 64M$
- $2^{27} = 128M$
- $2^{28} = 256M$
- $2^{29} = 512M$
- $2^{30} = 1G$

Binary Fractions

- What does .3 mean in a number like 1.3?
- Why?

Binary Fractions

- What does .3 mean in a number like 1.3?

100s	10s	1s	.	1/10	1/100	1/1000
		1	.	3		

- It means 3/10
- So what would 11.11 mean in binary?

Binary Fractions

- What does 11.11 mean in binary?

4s	2s	1s	.	1/2	1/4	1/8
	1	1	.	1	1	

- It means 2 + 1 + ½ + ¼ = 3¾ or 3.75

Fractional Binary to Decimal

- The same as non-fractional numbers.
- Add up the column values where a 1 is the digit.

Convert 1011.1101

$8 + 2 + 1 + \frac{1}{2} + \frac{1}{4} + 1/16 = 11$ & $13/16$

Fractional Binary to Decimal (Easy)

- Whole part same as non-fractional numbers.
- Numerator for fraction will be the binary number to the right of radix.
- Denominator for fraction will be the least column value.

Convert 1011.1101 = Decimal part is 13

Convert 1011.1101 = column value is 1/16

Answer is 11 & 13/16

Decimal to Fractional Binary

- Convert whole part as learned earlier.
  - That is repeatedly divide by 2.
- Convert fractional part by repeatedly multiplying by 2.

Convert 12.3 to fractional binary	
Convert whole part (12)	Convert fraction (0.3)
12 / 2 = 6r0	0.3 * 2 = 0.6
6 / 2 = 3r0	0.6 * 2 = 1.2
3 / 2 = 1r1	0.2 * 2 = 0.4
1 / 2 = 0r1	0.4 * 2 = 0.8
1100 (reverse order)	0.8 * 2 = 1.6
	0.6 * 2 = 1.2
	0.2 * 2 = 0.4
	… When do we quit?

Decimal to Fractional Binary

- Done with the fractional part when:
  - Result becomes zero
  - Result repeats
  - We have expressed a given number of bits
- Which should we use for the example to the right?
- $1100.0\overline{1001}$

Convert 12.3 to fractional binary	
Convert whole part (12)	Convert fraction (0.3)
12 / 2 = 6r0	0.3 * 2 = 0.6
6 / 2 = 3r0	0.6 * 2 = 1.2
3 / 2 = 1r1	0.2 * 2 = 0.4
1 / 2 = 0r1	0.4 * 2 = 0.8
1100 (reverse order)	0.8 * 2 = 1.6
	0.6 * 2 = 1.2
	0.2 * 2 = 0.4
	.0100110… (in order)
	… When do we quit?

IEEE Floating Point Storage

- Floating point numbers are stored as binary, normalized scientific notation.
- Store Sign, Exponent and Mantissa in a single unit.

Size	Sign bits	Exponent Bits	Mantissa bits
16	1	5	10
32	1	8	23
64	1	11	51
128	1	16	113

- We will learn 16 and 32 bit floating point numbers.

Scientific notation

- Decimal

  1.234×10^{2}

- Binary

  1.10101×2^{3}

- The red above is the mantissa.
- The green above is the exponent.
- Normalized – Only one non-zero digit to the left of the radix.

Writing Normalized SN

- Normalize the following.

  111.11101

  0.0000110

  1001.111×2^{2}

  0.0111×2^{-4}

Writing Normalized SN Answers

- Normalize the following.

$111.11101 = 1.1111101 \times 2^{2}$

$0.0000110 = 1.10 \times 2^{-5}$

$1001.111 \times 2^{2} = 1.001111 \times 2^{5}$

$0.0111 \times 2^{-4} = 1.11 \times 2^{-6}$

IEEE 16 Bit Floating Point

- Sign 1 bit
  - 0 for positive, 1 for negative
- Exponent 5 bits
  - Stored as Excess-15
- Mantissa 10 bits
  - Normalized – Remove leading 1

S	E	E	E	E	E	M	M	M	M	M	M	M	M	M	M
1	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0

Excess-15

- Add excess before storing.
  - When converting and storing in IEEE form.
  - The exponent + excess is known as the **Characteristic**
  - You store the characteristic.

- Subtract 15 after removing.
  - When removing from and interpreting IEEE form.
  - The exponent is the characteristic – 15.

Convert to Floating Point

- Write your number in normalized binary scientific notation if not already done for you. The mantissa of this number MUST have enough bits to fill the mantissa portion of the floating-point number. (10 or 24)
- Calculate the characteristic by adding the excess to the exponent.
- Store the sign: 0 = Positive, 1 = Negative
- Store the characteristic as calculated above.
- Remove the leading 1 from the mantissa and store.
- For most quizzes and homework write the value as a hexadecimal value.

16 Bit Example

- Store 1.11100×2^4 int 16 bit IEEE form.
  - What is the sign?
  - What is the exponent?
  - What is the mantissa?

Convert to Floating Point

- Write your number in normalized binary scientific notation if not already done for you. The mantissa of this number MUST have enough bits to fill the mantissa portion of the floating-point number.

  $+1.11100 \times 2^{4}$

- Calculate the characteristic by adding the excess to the exponent.

  Characteristic = exponent + excess = 4 + 15 = **19**

- Store the sign: 0 = Positive, 1 = Negative

  0

- Convert the characteristic to binary and store.

  010011

- Remove the leading 1 from the mantissa and store. Make sure the entire area for mantissa bits is filled.

  0100111110000000

- For most quizzes and homework write the value as a hexadecimal value.

  0100 1111 1000 0000 = 0x4F80

Convert to decimal from Floating Point

- If the number is in hexadecimal, convert to a binary pattern.
- Calculate the characteristic by converting the exponent bits to decimal.
- Calculate the exponent by subtracting the excess from the characteristic.
- Write the mantissa bits and add 1. to the beginning.
- Write x 2^{exponent} using the exponent calculated above.
- Put in the sign as determined by bit 1.
- Convert from SN to normal fractional binary by moving the radix the required number of bits in the specified direction.
- Convert the fractional binary to decimal as described previously.

16 Bit Example

- Convert 0xC640 from 16-bit IEEE to decimal
  - What is the sign?
  - What is the exponent?
  - What is the mantissa?

Convert to decimal from Floating Point

- 0xC640
- If the number is in hexadecimal, convert to a binary pattern.

  1100011001000000
- Calculate the characteristic by converting the exponent bits to decimal.

  10001 = 17
- Calculate the exponent by subtracting the excess from the characteristic.

  17 – 15 = 2
- Write the mantissa bits and add 1. to the beginning.

  1.1001
- Write x $2^{exponent}$ using the exponent calculated above.

  1.1001 x 2^2
- Put in the sign according to bit 1.
- Convert from SN to normal fractional binary by moving the radix the required number of bits in the specified direction.
- Convert the fractional binary to decimal as described previously.

IEEE 32 Bit Floating Point

- Sign 1 bit
  - 0 for positive, 1 for negative
- Exponent 8 bits
  - Stored as Excess-___?
- Mantissa 23 bits
  - Normalized – Remove leading 1

S	E	E	E	E	E	E	E	E	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M
1	1	0	0	0	0	0	0	1	1	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Excess-127

- Add excess (127) before storing.
  - When converting and storing in IEEE form.
  - The exponent + excess is known as the **Characteristic**
  - You store the characteristic.

- Subtract excess (127) after removing.
  - When removing from and interpreting IEEE form.
  - The exponent is the characteristic – 127.

Floating point conversions 16 Bit

Convert from decimal to 16-bit floating point:

10.09375

2.71

Convert from 16-bit floating point:

xCB40

x5112

Floating point conversion 32 bit

Convert from decimal to 32-bit floating point:

10.09375

2.71

Convert from 32-bit floating point:

C18C0000

BF980000

Special Numbers

- IEEE Defines certain special numbers
  - Zero (+ and -)
  - Denormalized
  - Infinity (+ and -)
  - Not a Number (NaN)
- Special numbers use patterns of zeros in the characteristic and mantissa field
- For normal floating-point representation
  - You cannot use all zeros in the characteristic.
  - You cannot use all zeros in the mantissa
- For this course the only special number you need to know is how to store zero.

Zero and Denormalized

- Zero
  - We cannot represent zero in the normal floating-point representation due to the assumption of a leading 1.
  - Zero is a special value denoted with all zeros in the characteristic and all zeros in the mantissa.
  - Note that -0 and +0 are distinct values, though they both compare as equal.

+0 = 0x00000000	-0 = 0x80000000	(32 bit)
+0 = 0x0000	-0 = 0x8000	(16 bit)

- Denormalized
  - Denormalized means the scientific notation has a zero to the left of the radix instead of a 1.

    0.1×2^3 instead of 1.0×2^2

  - Denormalized is represented as all zeros in the characteristic and anything except zero in the manitssa.
  - From this you can interpret zero as a special type of denormalized number.

Infinity

- Infinity
  - The values +infinity and -infinity are denoted with all ones in the characteristic and all zeros in the mantissa.
  - The sign bit distinguishes between negative infinity and positive infinity.
  - Being able to denote infinity as a specific value is useful because it allows operations to continue past overflow situations. *Operations with infinite values are well defined in IEEE floating point.*
  - *Hex values*

+∞ = 0x7C00	*-∞ = 0xFC00*	*(16 bit)*
+∞ = 0x7F800000	*-∞ = 0xFF800000*	*(32 bit)*

Not A Number

- Not A Number (Nan) is used to represent a value that does not represent a real number.
- NaN's are represented by a bit pattern with a characteristic of all ones and a non-zero mantissa.
- There are two categories of NaN: QNaN (*Quiet NaN*) and SNaN (*Signalling NaN*).
  - A QNaN is a NaN with the most significant fraction bit set. QNaN's propagate freely through most arithmetic operations. These values pop out of an operation when the result is not mathematically defined.
  - An SNaN is a NaN with the most significant fraction bit clear. It is used to signal an exception when used in operations. SNaN's can be handy to assign to uninitialized variables to trap premature usage.
- Semantically, QNaN's denote *indeterminate* operations, while SNaN's denote *invalid* operations.

Sign	Exponent (e)	Fraction (f)	Value
0	00⋯00	00⋯00	+0
0	00⋯00	00⋯01 ⋮ 11⋯11	Positive Denormalized Real $0.f \times 2^{(-b+1)}$
0	00⋯01 ⋮ 11⋯10	XX⋯XX	Positive Normalized Real $1.f \times 2^{(e-b)}$
0	11⋯11	00⋯00	+Infinity
0	11⋯11	00⋯01 ⋮ 01⋯11	SNaN
0	11⋯11	10⋯00 ⋮ 11⋯11	QNaN
1	00⋯00	00⋯00	-0
1	00⋯00	00⋯01 ⋮ 11⋯11	Negative Denormalized Real $-0.f \times 2^{(-b+1)}$
1	00⋯01 ⋮ 11⋯10	XX⋯XX	Negative Normalized Real $-1.f \times 2^{(e-b)}$
1	11⋯11	00⋯00	-Infinity
1	11⋯11	00⋯01 ⋮ 01⋯11	SNaN
1	11⋯11	10⋯00 ⋮ 11.11	QNaN

Operations and Results

Operation	Result
n ÷ ±Infinity	0
±Infinity × ±Infinity	±Infinity
±nonzero ÷ 0	±Infinity
Infinity + Infinity	Infinity
±0 ÷ ±0	*NaN*
Infinity - Infinity	*NaN*
±Infinity ÷ ±Infinity	*NaN*
±Infinity × 0	*NaN*