Bits, Data Types, and Operations

How do we represent data in a computer?

- At the lowest level, a computer is an electronic machine.
  - works by controlling the flow of electrons
- Easy to recognize two conditions:
  1. presence of a voltage – we'll call this state "1"
  2. absence of a voltage – we'll call this state "0"
- Could base state on *value* of voltage,
  but control and detection circuits more complex.
  - compare turning on a light switch to
    measuring or regulating voltage

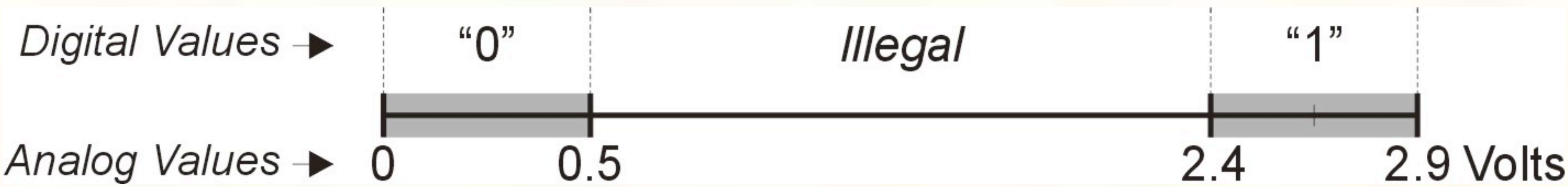
Computer is a binary digital system

Digital system:

- finite number of symbols

Binary (base two) system:

- has two states: 0 and 1



- Basic unit of information is the *binary digit*, or *bit*.
- Values with more than two states require multiple bits.
  - A collection of two bits has four possible states:
    00, 01, 10, 11
  - A collection of three bits has eight possible states:
    000, 001, 010, 011, 100, 101, 110, 111
  - *A collection of n bits has 2^n possible states.*

What kinds of data do we need to represent?

- Numbers – signed, unsigned, integers, floating point, complex, rational, irrational, …
- Text – characters, strings, …
- Images – pixels, colors, shapes, …
- Sound
- Logical – true, false
- Instructions
- …
- Data type:
  - *representation* and *operations* within the computer
- We'll start with numbers…

Unsigned Integers

- Non-positional notation
  - could represent a number ("5") with a string of ones ("11111")
  - problems?
- Weighted positional notation
  - like decimal numbers: "329"
  - "3" is worth 300, because of its position, while "9" is only worth 9

329

10^2 10^1 10^0

3x100 + 2x10 + 9x1 = 329

most significant 101 *least significant*

2^2 2^1 2^0

1x4 + 0x2 + 1x1 = 5

Unsigned Integers (cont.)

- An n-bit unsigned integer represents 2^n values:
  from 0 to 2^n-1.

2^2	2^1	2^0	
0	0	0	0
0	0	1	1
0	1	0	2
0	1	1	3
1	0	0	4
1	0	1	5
1	1	0	6
1	1	1	7

Unsigned Binary Arithmetic

- Base-2 addition – just like base-10!
  - add from right to left, propagating carry

carry

```
     10010          10010          1111
 +    1001      +    1011      +      1
     11011          11101         10000

                    10111
                +     111
```

Subtraction, multiplication, division,…

Signed Integers

- With n bits, we have 2^n distinct values.
  - assign about half to positive integers (1 through 2^{n-1}) and about half to negative (- 2^{n-1} through -1)
  - that leaves two values: one for 0, and one extra
- Positive integers
  - just like unsigned – zero in *most significant* (MS) bit
    00101 = 5
- Negative integers
  - sign-magnitude – set MS bit to show negative, other bits are the same as unsigned
    10101 = -5
  - one's complement – flip every bit to represent negative
    11010 = -5
  - in either case, MS bit indicates sign: 0=positive, 1=negative

Two's Complement

- Problems with sign-magnitude and 1's complement
  - two representations of zero (+0 and –0)
  - arithmetic circuits are complex
    - How to add two sign-magnitude numbers?
      - e.g., try 2 + (-3)
    - How to add to one's complement numbers?
      - e.g., try 4 + (-3)
- ***Two's complement*** representation developed to make circuits easy for arithmetic.
  - for each positive number (X), assign value to its negative (-X), such that X + (-X) = 0 with "normal" addition, ignoring carry out

	00101	(5)		**01001**	(9)
+	**11011**	(-5)	**+**	**10111**	(-9)
	00000	(0)		**00000**	(0)

Two's Complement Representation

- If number is positive or zero,
  - normal binary representation, zeroes in upper bit(s)
- If number is negative,
  - start with positive number
  - flip every bit (i.e., take the one's complement)
  - then add one

	00101	(5)		**01001**	(9)
	11010	(1's comp)		**10110**	(1's comp)
+	**<u>1</u>**		**+**	**<u>1</u>**	
	11011	(-5)		**10111**	(-9)

Two's Complement Shortcut

- To take the two's complement of a number:
  - copy bits from right to left until (and including) the first "1"
  - flip remaining bits to the left

```
  011010000
  100101111   (1's comp)
+         1
  100110000
```

```
011010000
(flip)   (copy)
100110000
```

Two's Complement Signed Integers

- MS bit is sign bit – it has weight -2^{n-1}.
- Range of an n-bit number: -2^{n-1} through $2^{n-1} - 1$.
  - The most negative number (-2^{n-1}) has no positive counterpart.

-2^3	2^2	2^1	2^0	
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7

-2^3	2^2	2^1	2^0	
1	0	0	0	-8
1	0	0	1	-7
1	0	1	0	-6
1	0	1	1	-5
1	1	0	0	-4
1	1	0	1	-3
1	1	1	0	-2
1	1	1	1	-1

Converting Binary (2's C) to Decimal

1. If leading bit is one, take two's complement to get a positive number.
2. Add powers of 2 that have "1" in the corresponding bit positions.
3. If original number was negative, add a minus sign.

$$X = 01101000_{two}$$
$$= 2^6+2^5+2^3 = 64+32+8$$
$$= 104_{ten}$$

n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	1024

Assuming 8-bit 2's complement numbers.

More Examples

$$X = 00100111_{two}$$
$$= 2^5+2^2+2^1+2^0 = 32+4+2+1$$
$$= 39_{ten}$$

$$X = 11100110_{two}$$
$$-X = 00011010$$
$$= 2^4+2^3+2^1 = 16+8+2$$
$$= 26_{ten}$$
$$X = -26_{ten}$$

n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	1024

Assuming 8-bit 2's complement numbers.

Converting Decimal to Binary (2's C)

- First Method: ***Division***
  1. Find magnitude of decimal number. (Always positive.)
  2. Divide by two – remainder is least significant bit.
  3. Keep dividing by two until answer is zero, writing remainders from right to left.
  4. Append a zero as the MS bit; if original number was negative, take two's complement.

$X = 104_{ten}$

104/2	=	52 r0	*bit 0*
52/2	=	26 r0	*bit 1*
26/2	=	13 r0	*bit 2*
13/2	=	6 r1	*bit 3*
6/2	=	3 r0	*bit 4*
3/2	=	1 r1	*bit 5*
1/2	=	0 r1	*bit 6*

$X = 01101000_{two}$

Converting Decimal to Binary (2's C)

- Second Method: ***Subtract Powers of Two***
  1. Find magnitude of decimal number.
  2. Subtract largest power of two less than or equal to number.
  3. Put a one in the corresponding bit position.
  4. Keep subtracting until result is zero.
  5. Append a zero as MS bit; if original was negative, take two's complement.

n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	1024

$X = 104_{ten}$

104 - 64	=	40		*bit 6*
40 - 32	=	8		*bit 5*
8 - 8	=	0		*bit 3*

$X = 01101000_{two}$

Operations: Arithmetic and Logical

- Recall:
  a data type includes *representation* and *operations*.
- We now have a good representation for signed integers, so let's look at some arithmetic operations:
  - Addition
  - Subtraction
  - Sign Extension
- We'll also look at overflow conditions for addition.
- Multiplication, division, etc., can be built from these basic operations.
- Logical operations are also useful:
  - AND
  - OR
  - NOT

Addition

- As we've discussed, 2's comp. addition is just binary addition.
  - assume all integers have the same number of bits
  - ignore carry out
  - for now, assume that sum fits in n-bit 2's comp. representation

	01101000	(104)		**11110110**	(-10)
+	**11110000**	(-16)	**+**	**11110111**	(-9)
	01011000	(98)		**11101101**	(-19)

Assuming 8-bit 2's complement numbers.

Subtraction

- Negate subtrahend (2nd no.) and add.
  - assume all integers have the same number of bits
  - ignore carry out
  - for now, assume that difference fits in n-bit 2’s comp. representation

	01101000	(104)		**11110110**	(-10)
-	**<u>00010000</u>**	(16)	-	**<u>11110111</u>**	(-9)
	01101000	(104)		**11110110**	(-10)
+	**<u>11110000</u>**	(-16)	+	**<u>00001001</u>**	(9)
	01011000	(88)		**11111111**	(-1)

Assuming 8-bit 2’s complement numbers.

Sign Extension

- To add two numbers, we must represent them with the same number of bits.
- If we just pad with zeroes on the left:

4-bit		**8-bit**	
0100	(4)	**00000100**	(still 4)
1100	(-4)	**00001100**	(12, not -4)

- Instead, replicate the MS bit -- the sign bit:

4-bit		**8-bit**	
0100	(4)	**00000100**	(still 4)
1100	(-4)	**11111100**	(still -4)

Overflow

- If operands are too big, then sum cannot be represented as an *n*-bit 2's comp number.

	01000	(8)		**11000**	(-8)
+	**01001**	(9)	**+**	**10111**	(-9)
	10001	(-15)		**01111**	(+15)

- We have overflow if:
  - signs of both operands are the same, and
  - sign of sum is different.
- Another test -- easy for hardware:
  - carry into MS bit does not equal carry out

Logical Operations

- Operations on logical TRUE or FALSE
  - two states -- takes one bit to represent: TRUE=1, FALSE=0

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

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

A	NOT A
0	1
1	0

- View *n*-bit number as a collection of *n* logical values
  - operation applied to each bit independently

Examples of Logical Operations

- AND
  - useful for clearing bits
    - AND with zero = 0
    - AND with one = no change

	11000101
AND	<u>00001111</u>
	00000101

- OR
  - useful for setting bits
    - OR with zero = no change
    - OR with one = 1

	11000101
OR	<u>00001111</u>
	11001111

- NOT
  - unary operation -- one argument
  - flips every bit

NOT	<u>11000101</u>
	00111010

Hexadecimal Notation

- It is often convenient to write binary (base-2) numbers as hexadecimal (base-16) numbers instead.
  - fewer digits -- four bits per hex digit
  - less error prone -- easy to corrupt long string of 1’s and 0’s

Binary	Hex	Decima	Binary	Hex	Decima
0000	0	0	1000	8	8
0001	1	1	1001	9	9
0010	2	2	1010	A	10
0011	3	3	1011	B	11
0100	4	4	1100	C	12
0101	5	5	1101	D	13
0110	6	6	1110	E	14
0111	7	7	1111	F	15

Converting from Binary to Hexadecimal

- Every four bits is a hex digit.
  - start grouping from right-hand side

011	1010	1000	1111	0100	1101	0111
3	A	8	F	4	D	7

011101010001111010011010111

This is not a new machine representation, just a convenient way to write the number.

Fractions: Fixed-Point

- How can we represent fractions?
  - Use a "binary point" to separate positive from negative powers of two -- just like "decimal point."
  - 2's comp addition and subtraction still work.
    - if binary points are aligned

$2^{-1} = 0.5$

$2^{-2} = 0.25$

$2^{-3} = 0.125$

00101000.101 (40.625)

\+ **<u>11111110.110</u>** (-1.25)

00100111.011 (39.375)

No new operations -- same as integer arithmetic.

Very Large and Very Small: Floating-Point

- Large values: 6.023×10^{23} -- requires 79 bits
- Small values: 6.626×10^{-34} -- requires >110 bits
- Use equivalent of "scientific notation": $F \times 2^E$
- Need to represent F (*fraction*), E (*exponent*), and sign.
- IEEE 754 Floating-Point Standard (32-bits):

1b	*8b*	*23b*
S	Exponent	Fraction

$$N = (-1)^S \times 1.\text{fraction} \times 2^{\text{exponent}-127},\ 1 \le \text{exponent} \le 254$$

$$N = (-1)^S \times 0.\text{fraction} \times 2^{-126},\ \text{exponent} = 0$$

Floating Point Example

- Single-precision IEEE floating point number:

  1 | 01111110 | 10000000000000000000000

  sign *exponent* *fraction*

  - Sign is 1 – number is negative.
  - Exponent field is 01111110 = 126 (decimal).
  - Fraction is 0.100000000000… = 0.5 (decimal).

- Value = $-1.5 \times 2^{(126-127)} = -1.5 \times 2^{-1} = -0.75$.

Floating-Point Addition

- Will regular 2's complement arithmetic work for Floating Point numbers?
- (*Hint*: In decimal, how do we compute $3.07 \times 10^{12} + 9.11 \times 10^{8}$?)
  - Step 1: match the exponents. We'll prefer to do this by making the small exponent match the large one since that means shifting the mantissa to the right, and with finite precision representations any lost significant digits will be at the low order end of the number
  - Step 2: Add the mantissas.
  - Step 3: Normalize the result if necessary.

$$3.75 \times 10^{12} + 9.125 \times 10^{8}$$

$$= 3.75 \times 10^{12} + .0009125 \times 10^{12}$$

$$= 3.7509125 \times 10^{12}$$

Floating-Point Addition

- Will regular 2's complement arithmetic work for Floating Point numbers?
- Same algorithm in binary
  - Let's do it in IEEE 754

$$3.75 \times 10^{12} + 9.125 \times 10^{8}$$
$$= 3.75 \times 10^{12} + .0009125 \times 10^{12}$$
$$= 3.7509125 \times 10^{12}$$

$$3.75 \times 10^{12} \approx 11.11 \times 2^{40} = 1.111 \times 2^{41}$$
$$= 0 \mid 10101000 \mid 11100000000000000000000 = \text{x}54700000$$

$$9.125 \times 10^{8} \approx 1001.001 \times 2^{27} = 1.001001 \times 2^{30}$$
$$= 0 \mid 10000010 \mid 00100100000000000000000 = \text{x}41200000$$

$$1.111 \times 2^{41} + 1.001001 \times 2^{30} = 1.111 \times 2^{41} + 0.00000000001001001 \times 2^{41}$$
$$= 1.11100000001001001 \times 2^{41}$$
$$= 0 \mid 10101000 \mid 11100000001001001000000 = \text{x}54701240$$

(note: binary and decimal not quite equal due to inexact conversion of decimal exponents to binary)

Floating-Point Multiplication

- In decimal
  - Step 1: add exponents
  - Step 2: multiply mantissas
  - Step 3: normalize result
- Same algorithm in binary
  - Let's do it in IEEE 754

$$3.75 \times 10^{12} \times 9.125 \times 10^{8}$$
$$= 3.75 \times 9.125 \times 10^{20}$$
$$= 34.21875 \times 10^{20}$$
$$= 3.421875 \times 10^{21}$$

$$1.111 \times 2^{41} \times 1.001001 \times 2^{30}$$
$$= 1.111 \times 1.001001 \times 2^{71}$$
$$= 10.001000111 \times 2^{71}$$
$$= 1.0001000111 \times 2^{72}$$
$$= 0 \mid 11000111 \mid 00010001110000000000000 = \text{x6388E000}$$

(note: binary and decimal not quite equal due to inexact conversion of decimal exponents to binary)

Text: ASCII Characters

- ASCII: Maps 128 characters to 7-bit code.
  - both printable and non-printable (ESC, DEL, …) characters

<table>
<tr><td>00</td><td>nul</td><td>10</td><td>dle</td><td>20</td><td>sp</td><td>30</td><td>0</td><td>40</td><td>@</td><td>50</td><td>P</td><td>60</td><td>`</td><td>70</td><td>p</td></tr>
<tr><td>01</td><td>soh</td><td>11</td><td>dc1</td><td>21</td><td>!</td><td>31</td><td>1</td><td>41</td><td>A</td><td>51</td><td>Q</td><td>61</td><td>a</td><td>71</td><td>q</td></tr>
<tr><td>02</td><td>stx</td><td>12</td><td>dc2</td><td>22</td><td>"</td><td>32</td><td>2</td><td>42</td><td>B</td><td>52</td><td>R</td><td>62</td><td>b</td><td>72</td><td>r</td></tr>
<tr><td>03</td><td>etx</td><td>13</td><td>dc3</td><td>23</td><td>#</td><td>33</td><td>3</td><td>43</td><td>C</td><td>53</td><td>S</td><td>63</td><td>c</td><td>73</td><td>s</td></tr>
<tr><td>04</td><td>eot</td><td>14</td><td>dc4</td><td>24</td><td>$</td><td>34</td><td>4</td><td>44</td><td>D</td><td>54</td><td>T</td><td>64</td><td>d</td><td>74</td><td>t</td></tr>
<tr><td>05</td><td>enq</td><td>15</td><td>nak</td><td>25</td><td>%</td><td>35</td><td>5</td><td>45</td><td>E</td><td>55</td><td>U</td><td>65</td><td>e</td><td>75</td><td>u</td></tr>
<tr><td>06</td><td>ack</td><td>16</td><td>syn</td><td>26</td><td>&</td><td>36</td><td>6</td><td>46</td><td>F</td><td>56</td><td>V</td><td>66</td><td>f</td><td>76</td><td>v</td></tr>
<tr><td>07</td><td>bel</td><td>17</td><td>etb</td><td>27</td><td>'</td><td>37</td><td>7</td><td>47</td><td>G</td><td>57</td><td>W</td><td>67</td><td>g</td><td>77</td><td>w</td></tr>
<tr><td>08</td><td>bs</td><td>18</td><td>can</td><td>28</td><td>(</td><td>38</td><td>8</td><td>48</td><td>H</td><td>58</td><td>X</td><td>68</td><td>h</td><td>78</td><td>x</td></tr>
<tr><td>09</td><td>ht</td><td>19</td><td>em</td><td>29</td><td>)</td><td>39</td><td>9</td><td>49</td><td>I</td><td>59</td><td>Y</td><td>69</td><td>i</td><td>79</td><td>y</td></tr>
<tr><td>0a</td><td>nl</td><td>1a</td><td>sub</td><td>2a</td><td>*</td><td>3a</td><td>:</td><td>4a</td><td>J</td><td>5a</td><td>Z</td><td>6a</td><td>j</td><td>7a</td><td>z</td></tr>
<tr><td>0b</td><td>vt</td><td>1b</td><td>esc</td><td>2b</td><td>+</td><td>3b</td><td>;</td><td>4b</td><td>K</td><td>5b</td><td>[</td><td>6b</td><td>k</td><td>7b</td><td>{</td></tr>
<tr><td>0c</td><td>np</td><td>1c</td><td>fs</td><td>2c</td><td>,</td><td>3c</td><td><</td><td>4c</td><td>L</td><td>5c</td><td>\</td><td>6c</td><td>l</td><td>7c</td><td>|</td></tr>
<tr><td>0d</td><td>cr</td><td>1d</td><td>gs</td><td>2d</td><td>-</td><td>3d</td><td>=</td><td>4d</td><td>M</td><td>5d</td><td>]</td><td>6d</td><td>m</td><td>7d</td><td>}</td></tr>
<tr><td>0e</td><td>so</td><td>1e</td><td>rs</td><td>2e</td><td>.</td><td>3e</td><td>></td><td>4e</td><td>N</td><td>5e</td><td>^</td><td>6e</td><td>n</td><td>7e</td><td>~</td></tr>
<tr><td>0f</td><td>si</td><td>1f</td><td>us</td><td>2f</td><td>/</td><td>3f</td><td>?</td><td>4f</td><td>O</td><td>5f</td><td>_</td><td>6f</td><td>o</td><td>7f</td><td>del</td></tr>
</table>

Interesting Properties of ASCII Code

- What is relationship between a decimal digit ('0', '1', …) and its ASCII code?
- What is the difference between an upper-case letter ('A', 'B', …) and its lower-case equivalent ('a', 'b', …)?
- Given two ASCII characters, how do we tell which comes first in alphabetical order?
- Are 128 characters enough? (http://www.unicode.org/)

No new operations -- integer arithmetic and logic.

Other Data Types

- Text strings
  - sequence of characters, terminated with NULL (0)
  - typically, no hardware support
- Image
  - array of pixels
    - monochrome: one bit (1/0 = black/white)
    - color: red, green, blue (RGB) components (e.g., 8 bits each)
    - other properties: transparency
  - hardware support:
    - typically none, in general-purpose processors
    - MMX -- multiple 8-bit operations on 32-bit word
- Sound
  - sequence of fixed-point numbers

LC-3 Data Types

- Some data types are supported directly by the instruction set architecture.

- For LC-3, there is only one hardware-supported data type:
  - 16-bit 2's complement signed integer
  - Operations: ADD, AND, NOT

- Other data types are supported by interpreting 16-bit values as logical, text, fixed-point, etc., in the software that we write.

Chapter 2

Bits, Data Types & Operations

- *Integer Representation*
- *Floating-point Representation*
- *Logic Operations*

Data types

- Our first requirement is to find a way to represent information (data) in a form that is mutually comprehensible by human and machine.
  - Ultimately, we will have to develop schemes for representing all conceivable types of information - language, images, actions, etc.
  - We will start by examining different ways of representing *integers*, and look for a form that suits the computer.
  - Specifically, the devices that make up a computer are switches that can be on or off, i.e. at high or low voltage. Thus they naturally provide us with two symbols to work with: we can call them *on* & *off*, or (more usefully) *0* and *1*.

Decimal Numbers

- "decimal" means that we have ten digits to use in our representation (the symbols 0 through 9)

- What is 3,546?
  - it is *three* thousands plus *five* hundreds plus *four* tens plus *six* ones.
  - i.e. $3{,}546 = 3.10^3 + 5.10^2 + 4.10^1 + 6.10^0$

- How about negative numbers?
  - we use two more symbols to distinguish positive and negative:

    + and -

Unsigned Binary Integers

$Y = \text{“abc”} = a.2^2 + b.2^1 + c.2^0$

(where the digits a, b, c can each take on the values of 0 or 1 only)

N = number of bits

Range is:
$0 \leq i < 2^N - 1$

	3-bits	5-bits	8-bits
0	000	00000	0000000
1	001	00001	0000000
2	010	00010	0000001
3	011	00011	0000001
4	100	00100	0000010
			0

Problem:
- How do we represent *negative* numbers?

Signed Magnitude

- Leading bit is the sign bit

$Y = \text{"abc"} = (-1)^a (b.2^1 + c.2^0)$

Range is:
$-2^{N-1} + 1 < i < 2^{N-1} - 1$

Problems:
- How do we do addition/subtraction?
- We have two numbers for zero (+/-)

-4	10100
-3	10011
-2	10010
-1	10001
-0	10000
+0	00000
+1	00001
+2	00010
+3	00011
+4	00100

One's Complement

- Invert all bits

If msb (most significant bit) is 1 then the number is negative (same as signed magnitude)

Range is:
$-2^{N-1} + 1 < i < 2^{N-1} - 1$

Problems:

•Same as for signed magnitude

-4	11011
-3	11100
-2	11101
-1	11110
-0	11111
+0	00000
+1	00001
+2	00010
+3	00011
+4	00100

Two's Complement

- Transformation
  - To transform a into -a, invert all bits in a and add 1 to the result

Range is:
$-2^{N-1} < i < 2^{N-1} - 1$

-16	10000
…	…
-3	11101
-2	11110
-1	11111
0	00000
+1	00001
+2	00010
+3	00011
…	…
+15	01111

Manipulating Binary numbers - 1

- Binary to Decimal conversion & vice-versa
  - A 4 bit binary number $A = a_3a_2a_1a_0$ corresponds to:

    $a_3.2^3 + a_2.2^2 + a_1.2^1 + a_0.2^0 = a_3.8 + a_2.4 + a_1.2 + a_0.1$

    (where a_i = 0 or 1 only)

  - A decimal number can be broken down by iterative division by 2, assigning bits to the columns that result in an odd number:

    e.g. $(13)_{10}$ => ((((13 - 1)/2 - 0)/2 - 1)/2 - 1) = 0 => $(01101)_2$

  - In the 2's complement representation, leading zeros <u>do not</u> affect the value of a positive binary number, and leading ones <u>do not</u> affect the value of a negative number. So:

    01101 = 00001101 = 13   and   11011 = 11111011 = -5

Manipulating Binary numbers - 2

- Binary addition simply consists of applying, to each column in the sum, the rules:

  $\underline{0 + 0 = 0}$ $\underline{1 + 0 = 0 + 1 = 1}$ $\underline{1 + 1 = 10}$

  - With 2's complement representation, this works for both positive and negative integers *so long as both numbers being added are represented with the same number of bits.*

    e.g. to add the number 13 => 00001101 (8 bits) to -5 => 1011 (4 bits):

    we have to ***sign-extend*** (SEXT) the representation of -5 to 8 bits:

    00001101
    11111011
    00001000 => 8 (as expected!)

Manipulating Binary numbers - 3

- Overflow
  - If we add the two (2's complement) 4 bit numbers representing 7 and 5 we get :

    0111 => +7

    <u>0101</u> => <u>+5</u>

    1100 => -4 (in 4 bit 2's comp.)

    - We get -4, not +12 as we would expect !!
    - We have *overflowed* the range of 4 bit 2's comp. (-8 to +7), so the result is invalid.
    - Note that if we add 16 to this result we get back 16 - 4 = 12
      - this is like "stepping up" to 5 bit 2's complement representation
  - In general, if the sum of two positive numbers produces a negative result, or vice versa, an overflow has occurred, and the result is invalid in that representation.

Limitations of fixed-point

- Fixed point numbers are not limited to representing only integers
  - But there are other considerations:
- Range:
  - The magnitude of the numbers we can represent is determined by how many bits we use:
    - e.g. with 32 bits the largest number we can represent is about +/- 2 billion, far too small for many purposes.
- Precision:
  - The exactness with which we can specify a number:
    - e.g. a 32 bit number gives us 31 bits of precision, or roughly 9 figure precision in decimal repesentation.
- We need another data type!

Real numbers

- Our decimal system handles non-integer *real* numbers by adding yet another symbol - the decimal point (.) to make a *fixed point* notation:
  - e.g. $3,456.78 = 3.10^3 + 5.10^2 + 4.10^1 + 6.10^0 + 7.10^{-1} + 8.10^{-2}$
- The *floating point*, or scientific, notation allows us to represent very large and very small numbers (integer or real), with as much or as little precision as needed:
  - Unit of electric charge $e = 1.602\ 176\ 462 \times 10^{-19}$ Coul.
  - Volume of universe = 1×10^{85} cm^3
    - the two components of these numbers are called the mantissa and the exponent

Floating point numbers in binary

- We mimic the decimal floating point notation to create a "hybrid" binary floating point number:
  - We first use a "binary point" to separate whole numbers from fractional numbers to make a fixed point notation:
    - e.g. 00011001.110 = $1.2^4 + 1.10^3 + 1.10^1 + 1.2^{-1} + 1.2^{-2}$ => 25.75

      ($2^{-1} = 0.5$ and $2^{-2} = 0.25$, etc.)
  - We then "float" the binary point:
    - 00011001.110 => 1.1001110 x 2^4

      mantissa = 1.1001110, exponent = 4
  - Now we have to express this without the extra symbols (x, 2, .)
    - by convention, we divide the available bits into three fields:

      sign, mantissa, exponent
  - These are still fixed-precision, only approximate real numbers

IEEE-754 fp numbers - 1

1	8 bits	23 bits
s	biased exp.	fraction

$$\mathbf{N} = (-1)^s \times 1.\mathbf{fraction} \times 2^{(\mathbf{biased\ exp.} - 127)}$$

- Sign: 1 bit
- Mantissa: 23 bits
  - We “normalize” the mantissa by dropping the leading 1 and recording only its fractional part (why?)
- Exponent: 8 bits
  - In order to handle both +ve and -ve exponents, we add 127 to the actual exponent to create a “biased exponent”:
    - 2^{-127} => biased exponent = 0000 0000 (= 0)
    - 2^{0} => biased exponent = 0111 1111 (= 127)
    - 2^{+127} => biased exponent = 1111 1110 (= 254)

IEEE-754 fp numbers - 2

- Example:
  - 25.75 => 00011001.110 => 1.1001110 x 2^4
  - sign bit = 0 (+ve)
  - normalized mantissa (fraction) = 100 1110 0000 0000 0000 0000
  - biased exponent = 4 + 127 = 131 => 1000 0011
  - so 25.75 => 0 1000 0011 100 1110 0000 0000 0000 0000 => x41CE0000
- Values represented by convention:
  - Infinity (+ and -): exponent = 255 (1111 1111) and fraction = 0
  - NaN (not a number): exponent = 255 and fraction ≠ 0
  - Zero (0): exponent = 0 and fraction = 0
    - note: exponent = 0 => fraction is *de-normalized,* i.e no hidden 1

IEEE-754 fp numbers - 3

- Double precision (64 bit) floating point

1	11 bits	52 bits
s	biased exp.	fraction

$$\mathbf{N} = (-1)^{\mathbf{s}} \times 1.\mathbf{fraction} \times 2^{(\mathbf{biased\ exp}. - 1023)}$$

- Range & Precision:
  - 32 bit:
    - mantissa of 23 bits + 1 => approx. 7 digits decimal
    - $2^{+/-127}$ => approx. $10^{+/-38}$
  - 64 bit:
    - mantissa of 52 bits + 1 => approx. 15 digits decimal
    - $2^{+/-1023}$ => approx. $10^{+/-306}$

Other Data Types

- Other numeric data types
  - e.g. BCD
- Bit vectors & masks
  - sometimes we want to deal with the individual bits themselves
- Text representations
  - ASCII: uses 8 bits to represent main Western alphabetic characters & symbols, plus several "control codes",
  - Unicode: 16 bit superset of ASCII providing representation of many different alphabets and specialized symbol sets.
  - EBCDIC: IBM's mainframe representation.

Hexadecimal Representation

- Base 16 (hexadecimal)
  - More a convenience for us humans than a true data type
  - 0 to 9 represented as such
  - 10, 11, 12, 13, 14, 15 represented by A, B, C, D, E, F
  - $16 = 2^4$: i.e. every hexadecimal digit can be represented by a 4-bit binary (unsigned) and vice-versa.

- Example

$$(16AB)_{16} = x16AB$$

$$= 1.16^3 + 6.16^2 + 10.16^1 + 11.16^0$$

$$= (5803)_{10} = \#5803$$

$$= b0001\ 0110\ 1010\ 1011$$

Another use for bits: Logic

- Beyond numbers
  - *logical variables* can be *true* or *false*, *on* or *off*, etc., and so are readily represented by the binary system.
  - A logical variable A can take the values *false = 0* or *true = 1* only.
  - The manipulation of logical variables is known as Boolean Algebra, and has its own set of operations - which are not to be confused with the arithmetical operations of the previous section.
  - Some basic operations: NOT, AND, OR, XOR

Basic Logic Operations

NOT

A	A'
0	1
1	0

AND

A	B	A.B
0	0	0
0	1	0
1	0	0
1	1	1

OR

A	B	A+B
0	0	0
0	1	1
1	0	1
1	1	1

- Equivalent Notations
  - not A = A’ = $\overline{A}$
  - A and B = A.B = A∧B = A intersection B
  - A or B = A+B = A∨B = A union B

More Logic Operations

XOR

A	B	A⊕B
0	0	0
0	1	1
1	0	1
1	1	0

XNOR

A	B	(A⊕B)’
0	0	1
0	1	0
1	0	0
1	1	1

- Exclusive OR (XOR): either A or B is 1, not both
- $A \oplus B = A.B' + A'.B$