Data Representation
Data Representation

Goal: Store numbers, characters, sets, database records in the computer.

What we got: Circuit that stores 2 voltages, one for logic 0 (0 volts) and one for logic 1 (3.3 volts).

- DRAM - uses a single capacitor to store and a transistor to select.
- SRAM - typically uses 6 transistors.

Definition: A bit is a unit of information. It is the amount of information needed to specify one of two equally likely choices.

Example: Flipping a coin has 2 possible outcomes, heads or tails. The amount of info needed to specify the outcome is 1 bit.
### Storing Information

<table>
<thead>
<tr>
<th>Value</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>H</td>
<td>0</td>
</tr>
<tr>
<td>T</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Value</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>0</td>
</tr>
<tr>
<td>True</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Value</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1e-4</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
</tr>
</tbody>
</table>

- Use more bits for more items
- Three bits can represent 8 values: 000, 001, ..., 111
- N bits can represent $2^N$ values:

<table>
<thead>
<tr>
<th>N</th>
<th>Can represent</th>
<th>Approximately</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>256</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>65,536</td>
<td>65 thousand (64K where K=1024)</td>
</tr>
<tr>
<td>32</td>
<td>4,294,967,296</td>
<td>4 billion</td>
</tr>
<tr>
<td>64</td>
<td>$1.8446 \times 10^{19}$</td>
<td>20 billion billion</td>
</tr>
</tbody>
</table>
Most computers today use:

<table>
<thead>
<tr>
<th>Type</th>
<th>bits</th>
<th>name for storage unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Character</td>
<td>8-16</td>
<td>byte (ASCII) – 16b Unicode (Java)</td>
</tr>
<tr>
<td>Integers</td>
<td>32-64</td>
<td>word (sometimes 8 or 16 bits)</td>
</tr>
<tr>
<td>Reals</td>
<td>32-64</td>
<td>word – double-word</td>
</tr>
</tbody>
</table>
Memory location for a character usually contains 8 bits:
• 00000000 to 11111111 (binary)
• 0x00 to 0xff (hexadecimal)

1. Which characters?
   • A, B, C, ..., Z, a, b, c, ..., z, 0, 1, 2, ..., 9
   • Punctuation (,:{...)
   • Special (\n \O ...)

2. Which bit patterns for which characters?
   • Want a standard!!!
   • Want a standard to help sort strings of characters.
ASCII (American Standard Code for Information Interchange)

Defines what character is represented by each sequence of bits.

Examples:

0100 0001 is 41 (hex) or 65 (decimal). It represents “A”.

0100 0010 is 42 (hex) or 66 (decimal). It represents “B”.

Different bit patterns are used for each different character that needs to be represented.
ASCII has some nice properties.

• If the bit patterns are compared, (pretending they represent integers), then
  “A” < “B”
  65  <  66

• This is good, because it helps with sorting things into alphabetical order.

• But…:
  • ‘a’ (61 hex) is different than ‘A’ (41 hex)
  • ‘8’ (38 hex) is different than the integer 8
  • ‘0’ is 30 (hex) or 48 (decimal)
  • ‘9’ is 39 (hex) or 57 (decimal)
Consider this program, what does it do?

getc \$t1        \# get a digit
add \$t2, \$t1, \$t1
putc \$t2
How to convert digits

```asm
asciibiad: .word 48  # code for '0', 49 is '1', ...
    # do a syscall to get a char, move to $t1
sub  $t2, $t1, 48    # convert char for digit to num
add  $t3, $t2, $t2
add  $t3, $t3, 48    # convert back to char
putc $t3
```

- The subtract takes the “bias” out of the char representation.
- The add puts the “bias” back in.
- This will only work right if the result is a single digit.
- Needed is an algorithm for translating character strings to integer representation.
Algorithm: Character string ➔ Integer

Example:

- For ‘3’ ‘5’ ‘4’
- Read ‘3’
  translate ‘3’ to 3
- Read ‘5’
  translate ‘5’ to 5
  integer = 3 x 10 + 5 = 35
- Read ‘4’
  translate ‘4’ to 4
  integer = 35 x 10 + 4 = 354
- Algorithm: asciibias = 48
  integer = 0
  while there are more characters
    get character
    digit ➔ character – asciibias
    integer ➔ integer x 10 + digit
Algorithm: Integer ↷ Character string

• Example:
  • For 354, figure out how many characters there are (3)
  • For 354 div 100 gives 3
    translate 3 to ‘3’ and print it out
    354 mod 100 gives 54
  • 54 div 10 gives 5, translate 5 to ‘5’ and print it out, 54 mod 10 gives 4
  • 4 div 1 gives 4
    translate 4 to ‘4’ and print it out
    4 mod 1 gives 0, so you’re done
Character String / Integer Representation

Compare:

```
mystring: .asciiz "123"
mynumber: .word 123
```

"123" is '1' 0x31 0011 0001
'2' 0x32 0011 0010
'3' 0x33 0011 0011
'\0' 0x0 0000 0000

• 0011 0001 0011 0010 0011 0011 0000 0000
• A series of four ASCII characters

123 = 0x7b = 0x0000007b = 00 00 00 7b
• 0000 0000 0000 0000 0000 0000 0111 1011
• A 32-bit 2’s complement integer
Assume our representation has a fixed number of bits $n$ (e.g. 32).

1. Which 4 billion integers do we want?
   - There are an infinite number of integers less than zero and an infinite number greater than zero.

2. What bit patterns should we select to represent each integer?
   Where the representation:
   - Does not affect the result of calculation
   - Does dramatically affect the ease of calculation

3. Convert to/from human-readable representation as needed.
Usual answers: 2 types
1. Represent 0 and consecutive positive integers
   • Unsigned integers
2. Represent positive and negative integers
   • Signed magnitude
   • One’s complement
   • Two’s complement
   • Biased
Unsigned and two’s complement the most common
Unsigned Integers

- Integer represented is binary value of bits:
  - 0000 ≡ 0, 0001 ≡ 1, 0010 ≡ 2, ...
- Encodes only positive values and zero
- Range: 0 to $2^n - 1$, for n bits
- Example:
  - 4 bits, values 0 to 15
    - $n = 4$, $2^4 - 1$ is 15
    - $[0:15] = 16 = 2^4$ different numbers
  - 7 is 0111
  - 17 not represent able
  - -3 not represent able
- Example:
  - 32 bits = $[0: 4,294,967,295]$
  - $4,294,967,296 = 2^{32}$ different numbers
## Integers

<table>
<thead>
<tr>
<th>Rep</th>
<th>Unsign</th>
<th>SM</th>
<th>1SC</th>
<th>2SC</th>
<th>Bias-8</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0011</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0110</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0111</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1001</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1010</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1011</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1101</td>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1110</td>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1111</td>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Signed Magnitude Integers

• A human readable way of getting both positive and negative integers.
• Not well suited to hardware implementation.
• But used with floating point.
Representation
• Use 1 bit of integer to represent the sign of the integer
  • Sign bit is msb: 0 is +, 1 is –
• Rest of the integer is a magnitude, with same encoding as unsigned integers.
• To get the additive inverse of a number, just flip (invert, complement) the sign bit.
• Range: \(-(2^{n-1} - 1)\) to \(2^{n-1} - 1\)
Signed Magnitude - Example

• 4 bits
  • $-2^3 + 1$ to $2^3 - 1$
  • -7 to +7

Questions:
• 0101 is ?
• -3 is ?
• +12 is ?
• [-7,..., -1, 0, +1,..., +7] = 7 + 1 + 7 = 15 < 16 = 2^4
  • Why?
  • What problems does this cause?
One’s Complement Representation

- Historically important (in other words, not used today!!!)
- Early computers built by Semour Cray (while at CDC) were based on 1’s complement integers.
- Positive integers use the same representation as unsigned.
  - 0000 is 0
  - 0111 is 7, etc
- Negation is done by taking a bitwise complement of the positive representation.
  - Complement = Invert = Not = Flip = \{0 \rightarrow 1, 1 \rightarrow 0\}
  - A logical operation done on a single bit
- Top bit is sign bit
One’s Complement Representation

• To get 1’s complement of -1
  • Take +1: 0001
  • Complement each bit: 1110
  • Don’t add or take away any bits.

• Examples:
  • 1100
    • This must be a negative number. To find out which, find the additive inverse!
    • 0011 is +3
    • 1100 must be?

• Properties of 1’s complement:
  • Any negative number will have a 1 in the MSB
  • There are 2 representations for 0, 0000 and 1111
Two’s Complement

- Variation on 1’s complement that does not have 2 representations for 0.
- This makes the hardware that does arithmetic simpler and faster than the other representations.
- The negative values are all “slid” by one, eliminating the −0 case.
- How to get 2’s complement representation:
  - Positive: just as if unsigned binary
  - Negative:
    - Take the positive value
    - Take the 1’s complement of it
    - Add 1
Two’s Complement

• Example:
  
  \[
  \begin{array}{c}
  0101 \quad (+5) \\
  \hline
  1010 \quad (-5 \text{ in 1’s complement}) \\
  +1 \\
  \hline
  1011 \quad (-5 \text{ in 2’s complement})
  \end{array}
  \]

• To get the additive inverse of a 2’s complement integer,
  1. Take the 1’s complement
  2. Add 1

• Number of integers represented:
  • With 4 bits:
    \([-8,\ldots,-1,0,+1,\ldots,+7] = 8+1+7=16=2^4 \text{ numbers}\]
  • With 32 bits:
    \([-2^{31},\ldots,-1,0,+1,\ldots,(2^{31}-1)] = 2^{31}+1+(2^{31}-1)=2^{32}
    \]
    \([-2^{1474836448},\ldots,-1,0,+1,\ldots,2^{1474836447}] \sim \pm 2^{32}\]
A Little Bit on Adding

Simple way of adding 1:
• Start at LSB, for each bit (working right to left)
  • While the bit is a 1, change it to a 0.
  • When a 0 is encountered, change it to a 1 and stop.
  • Can combine with bit inversion to form 2’s complement.
• More generally, it’s just like decimal!!
  • 0 + 0 = 0
  • 1 + 0 = 1
  • 1 + 1 = 2, which is 10 in binary, sum is 0, carry is 1.
  • 1 + 1 + 1 = 3, sum is 1, carry is 1.

\[
\begin{array}{c}
  x & 0011 \\
  +y & 0001 \\
  \hline
  \text{sum} & 0100
\end{array}
\]
A Little Bit on Adding

<table>
<thead>
<tr>
<th>Carry in</th>
<th>A</th>
<th>B</th>
<th>Sum</th>
<th>Carry out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Biased Representation

- An integer representation that skews the bit patterns so as to look just like unsigned but actually represent negative numbers.
- Example: 4-bit, with BIAS of \(2^3\) (8) (Excess 8)
  - True value to be represented: 3
  - Add in the bias: +8
  - Unsigned value: 11
- The bit pattern of 3 in biased-8 representation will be 1011
- Suppose we were given a biased-8 representation as 0110
  - Unsigned 0110 represents: 6
  - Subtract out the bias: -8
  - True value represented: -2
- Operations on the biased numbers can be unsigned arithmetic but represent both positive and negative values
- How do you add two biased numbers? Subtract?
Biased Representation

• $25_{10}$ in excess 100:
• $52_{10,\text{excess}127}$ is:
• $101101_2,\text{excess}31$ is:
• $001101_2,\text{excess}31$ is:
• Where is the sign “bit” in excess notation?
• Used in floating-point exponents
• n-bit biased notation must specify the bias
• Choosing a bias:
  • To get a ~ equal distribution of values above and below 0, the bias is usually $2^{n-1}$ or $2^{n-1} - 1$. 
Sign Extension

• How to change a number with a smaller number of bits into the same number (same representation) with a larger number of bits?
• This must be done frequently by arithmetic units
• Unsigned: xxxxx \( \rightarrow \) 000xxxxx
  • Copy the original integer into the LSBs, and put 0’s elsewhere
• Sign/magnitude: sxxxxx \( \rightarrow \) s00xxxxx
  • Copy the original integer’s magnitude into the LSBs
  • Put the original sign into the MSB, put 0’s elsewhere
Sign Extension

• 1’s and 2’s complement: Sign Extension
  \[ \text{xxxxxx} \oplus \text{ssssxxxxx} \]
  • Copy the original integer into the LSBs
  • Take the MSB of the original and copy it elsewhere
  • 0010101 \( \oplus \) 00000000 0010101
  • 11110000 \( \oplus \) 11111111 11110000
• In 2’s complement, the MSB (sign bit) is the \(-2^{n-1}\) place.
• It says “subtract \(2^{n-1}\) from \(b_{n-2}...b_0\).
• Sign extending one bit
  • Adds a \(-2^n\) place
  • Changes the old sign bit to a \(+2^{n-1}\) place
  • \(-2^n + 2^{n-1} = -2^{n-1}\), so the number stays the same
Sign Extension

-12 in 8-bit 2's complement:

It's additive inverse is: