Bit Representation Outline

1. Bit Representation Outline
2. How Are Integers Represented in Memory?
3. Decimal Number Representation (Base 10)
4. Decimal (Base 10) Breakdown
5. Nonal Number Representation (Base 9)
6. Nonal (Base 9) Breakdown
7. Octal Number Representation (Base 8)
8. Octal (Base 8) Breakdown
9. Trinary Number Representation (Base 3)
10. Trinary (Base 3) Breakdown
11. Binary Number Representation (Base 2)
12. Binary (Base 2) Breakdown & Conversion
13. Counting in Decimal (Base 10)
14. Counting in Nonal (Base 9)
15. Counting in Octal (Base 8)
16. Counting in Trinary (Base 3)
17. Counting in Binary (Base 2)
18. Counting in Binary (Base 2) w/Leading 0s
19. Counting in Binary Video
20. Adding Integers #1
21. Adding Integers #2
22. Binary Representation of int Values
23. Adding Bits #1
24. Adding Bits #2
25. Adding Bits #3
26. Adding Bits #4
How Are Integers Represented in Memory?

In computers, all data are represented as contiguous sequences of bits.

An integer is represented as a sequence of 8, 16, 32 or 64 bits. For example:

\[ 97 = \text{0000000000000000111100000001} \]

What does this mean???
In the *decimal* number system (base 10), we have **10 digits:**

0 1 2 3 4 5 6 7 8 9

We refer to these as the *Arabic* digits. For details, see:

http://en.wikipedia.org/wiki/Arabic_numerals
Decimal (Base 10) Breakdown

\[ 4721_{10} = \]

\[ \underline{4000_{10}} + \]
\[ \underline{700_{10}} + \]
\[ \underline{20_{10}} + \]
\[ \underline{1_{10}} = \]

\[ 4 \cdot 1000_{10} + \]
\[ 7 \cdot 100_{10} + \]
\[ 2 \cdot 10_{10} + \]
\[ 1 \cdot 1_{10} = \]

\[ 4 \cdot 10^3 + \]
\[ 7 \cdot 10^2 + \]
\[ 2 \cdot 10^1 + \]
\[ 1 \cdot 10^0 \]

**Jargon:** \(4721_{10}\) is pronounced “four seven two one base 10,” or “four seven two one one decimal.”
Nonal Number Representation (Base 9)

In the *nonal* number system (base 9), we have 9 digits:

0 1 2 3 4 5 6 7 8

**NOTE**: No one uses nonal in real life; this is just an example.
Nonal (Base 9) Breakdown

\[ 4721_9 = \]
\[ 4000_9 + \]
\[ 700_9 + \]
\[ 20_9 + \]
\[ 1_9 = \]
\[ 4 \cdot 1000_9 + \]
\[ 7 \cdot 100_9 + \]
\[ 2 \cdot 10_9 + \]
\[ 1 \cdot 1_9 = \]
\[ 4 \cdot 9^3 + \]
\[ 7 \cdot 9^2 + \]
\[ 2 \cdot 9^1 + \]
\[ 1 \cdot 9^0 = \]

So: \( 4721_9 = 3502_{10} \)

Jargon: \( 4721_9 \) is pronounced “four seven two one base 9,” or “four seven two one nonal.”
Octal Number Representation (Base 8)

In the **octal** number system (base 8), we have **8 digits**:

0 1 2 3 4 5 6 7

**NOTE**: Some computer scientists used to use octal in real life, but it has mostly fallen out of favor, because it’s been supplanted by base 16 (**hexadecimal**). Octal does show up a little bit in C character strings.
Octal (Base 8) Breakdown

\[
4721_8 = \underline{4} \cdot 512_{10} + \underline{7} \cdot 64_{10} + \underline{2} \cdot 8_{10} + \underline{1} \cdot 1_{10} = 2513_{10}
\]

So: \( 4721_8 = 2513_{10} \)

Jargon: \( 4721_8 \) is pronounced “four seven two one base 8,” or “four seven two one octal.”
Trinary Number Representation (Base 3)

In the trinary number system (base 3), we have 3 digits:

0 1 2

NOTE: No one uses trinary in real life; this is just an example.
Trinary (Base 3) Breakdown

\[2021_3 = 2000_3 + 0_3 + 20_3 + 1_3 = \]

\[
\begin{array}{c}
2 \cdot 1000_3 + \\
0 \cdot 100_3 + \\
2 \cdot 10_3 + \\
1 \cdot 1_3 =
\end{array}
\]

\[
\begin{array}{c}
2 \cdot 3^3 + \\
0 \cdot 3^2 + \\
2 \cdot 3^1 + \\
1 \cdot 3^0 =
\end{array}
\]

\[
\begin{array}{c}
2 \cdot 27_{10} + \\
0 \cdot 9_{10} + \\
2 \cdot 3_{10} + \\
1 \cdot 1_{10} = 61_{10}
\end{array}
\]

\textbf{So:} \ 2021_3 = 61_{10}

\textbf{Jargon:} \ 2021_3 \text{ is pronounced} \\
\text{“two zero two one base 3,” or} \\
\text{“two zero two one trinary.”}
In the **binary** number system (base 2), we have 2 **digits**:

```
0 1
```

This is the number system that computers use internally.
Binary (Base 2) Breakdown & Conversion

\[01100001_2 = \]

\[
\begin{align*}
0 & \cdot 10000000_2 + 0 \cdot 2^7 + 0 \cdot 128_{10} + \\
1 & \cdot 10000000_2 + 1 \cdot 2^6 + 1 \cdot 64_{10} + \\
1 & \cdot 1000000_2 + 1 \cdot 2^5 + 1 \cdot 32_{10} + \\
0 & \cdot 1000000_2 + 0 \cdot 2^4 + 0 \cdot 16_{10} + \\
0 & \cdot 1000000_2 + 0 \cdot 2^3 + 0 \cdot 8_{10} + \\
0 & \cdot 1000000_2 + 0 \cdot 2^2 + 0 \cdot 4_{10} + \\
0 & \cdot 1000000_2 + 0 \cdot 2^1 + 0 \cdot 2_{10} + \\
1 & \cdot 1000000_2 = 1 \cdot 2^0 = 1 \cdot 1_{10} = \\
\end{align*}
\]

\[97_{10} = \]

\[
\begin{array}{cccccccc}
2^7 & 2^6 & 2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 \\
0 & 1 & 1 & 0 & 0 & 0 & 0 & 1
\end{array}
\]

Bit Representation Lesson
CS1313 Fall 2023
In **base 10**, we **count** like so:

0,

1, 2, 3, 4, 5, 6, 7, 8, 9, 10,

11, 12, 13, 14, 15, 16, 17, 18, 19, 20,

21, 22, 23, 24, 25, 26, 27, 28, 29, 30,

...

91, 92, 93, 94, 95, 96, 97, 98, 99, 100,

101, 102, 103, 104, 105, 106, 107, 108, 109, 110,

...

191, 192, 193, 194, 195, 196, 197, 198, 199, 200,

...

991, 992, 993, 994, 995, 996, 997, 998, 999, 1000,

...
Counting in Nonal (Base 9)

In base 9, we count like so:

0,
1, 2, 3, 4, 5, 6, 7, 8, 10,
11, 12, 13, 14, 15, 16, 17, 18, 20,
21, 22, 23, 24, 25, 26, 27, 28, 30,
...

81, 82, 83, 84, 85, 86, 87, 88, 100,
101, 102, 103, 104, 105, 106, 107, 108, 110,
...

181, 182, 183, 184, 185, 186, 187, 188, 200,
...

881, 882, 883, 884, 885, 886, 887, 888, 1000,
...
Counting in Octal (Base 8)

In **base 8**, we **count** like so:

0,
1, 2, 3, 4, 5, 6, 7, 10,
11, 12, 13, 14, 15, 16, 17, 20,
21, 22, 23, 24, 25, 26, 27, 30,
...
71, 72, 73, 74, 75, 76, 77, 100,
101, 102, 103, 104, 105, 106, 107, 110,
...
171, 172, 173, 174, 175, 176, 177, 200,
...
771, 772, 773, 774, 775, 776, 777, 1000,
...
Counting in Trinary (Base 3)

In base 3, we count like so:

0,
1, 2, 10,
11, 12, 20,
21, 22, 100,
101, 102, 110,
111, 112, 120,
121, 122, 200,
201, 202, 210,
211, 212, 220,
221, 222, 1000,
...

...
Counting in Binary (Base 2)

In base 2, we count like so:

0, 1,
10, 11,
100, 101, 110, 111,
1000, 1001, 1010, 1011, 1100, 1101, 1110, 1111
10000, ...
In base 2, we sometimes like to put in leading zeros:

00000000, 00000001,
00000010, 00000011,
00000100, 00000101, 00000110, 00000111,
00001000, 00001001, 00001010, 00001011,
00001100, 00001101, 00001110, 00001111
00010000, ...

Counting in Binary (Base 2) with Leading 0s
Counting in Binary Video

https://img-9gag-fun.9cache.com/photo/aq7Q4AZ_460svvp9.webm
## Adding Integers #1

<table>
<thead>
<tr>
<th></th>
<th>128</th>
<th>64</th>
<th>32</th>
<th>16</th>
<th>8</th>
<th>4</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^7$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$2^6$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$2^5$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$2^4$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$2^3$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$2^2$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$2^1$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$2^0$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**97\textsubscript{10} =**

\[
\begin{array}{cccccccc}
0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\
\end{array}
\]

**+ 15\textsubscript{10} =**

\[
\begin{array}{cccccccc}
0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
\end{array}
\]

\[
\begin{array}{cccccccc}
0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
\end{array}
\]
Adding Integers #2

<table>
<thead>
<tr>
<th>128</th>
<th>64</th>
<th>32</th>
<th>16</th>
<th>8</th>
<th>4</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^7$</td>
<td>$2^6$</td>
<td>$2^5$</td>
<td>$2^4$</td>
<td>$2^3$</td>
<td>$2^2$</td>
<td>$2^1$</td>
<td>$2^0$</td>
</tr>
</tbody>
</table>

97\(_{10}\) = 

| 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 |

+ 06\(_{10}\) = 

| 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |

103\(_{10}\) = 

| 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 |
Binary Representation of int Values

```c
#include <stdio.h>

int main ()
{ /* main */
  int x;
  x = 97;
  printf("%d\n", x);
  x = x + 6;
  printf("%d\n", x);
  return 0;
} /* main */
```

% cat xadd.c
#include <stdio.h>

int main ()
{ /* main */
  int x;
  x = 97;
  printf("%d\n", x);
  x = x + 6;
  printf("%d\n", x);
  return 0;
} /* main */

% gcc -o xadd xadd.c
% xadd
97
103
Adding Bits #1

How does a binary bitwise adder actually work?
The following is an example solution, but not how it’s actually done.
Consider adding a bit to a bit.
You’ll get a sum bit and a carry bit.

<table>
<thead>
<tr>
<th>Sum</th>
<th>Carry</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 + 0 = 0</td>
<td>carry the 0</td>
</tr>
<tr>
<td>0 + 1 = 1</td>
<td>carry the 0</td>
</tr>
<tr>
<td>1 + 0 = 1</td>
<td>carry the 0</td>
</tr>
<tr>
<td>1 + 1 = 0</td>
<td>carry the 1</td>
</tr>
</tbody>
</table>

sum = addend XOR augend =

(\text{addend OR augend}) \AND \text{NOT(addend AND augend)}

carry = \text{addend AND augend}
Adding Bits #2

After you add a bit to a bit, you get the sum bit and the carry bit. That’s what will happen for the rightmost bit of a byte or a word. But what about the next-to-rightmost bit?

The next-to-rightmost bit (and all other bits to the left of the rightmost bit) will be the sum of the previous (to the right) carry bit plus the addend bit plus the augend bit in that bit’s place:

previous_carry + addend + augend:

\[
\begin{align*}
0 + 0 + 0 &= 0 \text{ carry 0} & 1 + 0 + 0 &= 1 \text{ carry 0} \\
0 + 0 + 1 &= 1 \text{ carry 0} & 1 + 0 + 1 &= 0 \text{ carry 1} \\
0 + 1 + 0 &= 1 \text{ carry 0} & 1 + 1 + 0 &= 0 \text{ carry 1} \\
0 + 1 + 1 &= 0 \text{ carry 1} & 1 + 1 + 1 &= 1 \text{ carry 1}
\end{align*}
\]
Adding Bits #3

previous_carry + addend + augend:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0 + 0</td>
<td>0 + 1</td>
<td>1 + 1</td>
</tr>
<tr>
<td></td>
<td>0 + 1</td>
<td>1 + 1</td>
</tr>
<tr>
<td></td>
<td>0 + 1</td>
<td>1 + 1</td>
</tr>
<tr>
<td>0 + 0</td>
<td>0 + 1</td>
<td>1 + 1</td>
</tr>
</tbody>
</table>

sum =

((NOT previous_carry) AND (NOT addend) AND augend) OR
((NOT previous_carry) AND addend AND (NOT augend)) OR
(previous_carry AND (NOT addend) AND (NOT augend)) OR
(previous_carry AND addend AND augend)

carry =

(previous_carry AND addend) OR (previous_carry AND augend)
OR (addend AND augend)
Adding Bits #4

You can add a pair of binary numbers of whatever number of bits (for example, 8 bits, 16 bits, 32 bits, 64 bits) by having a series of bitwise adders, each of which takes as its input the associated bits from the addend and augend, plus the carry bit from the previous bit to its right.

(The exception is that the rightmost bit’s carry bit is always zero.) So a binary adder is very cheap to build!