Binary Arithmetic

Daniel Sanchez
Computer Science & Artificial Intelligence Lab
M.I.T.
Reminder: Encoding Positive Integers

- Bit \( i \) in a binary representation (in right-to-left order) is assigned weight \( 2^i \)

\[
\begin{array}{cccccccccccc}
2^{11} & 2^{10} & 2^9 & 2^8 & 2^7 & 2^6 & 2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 \\
0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0
\end{array}
\]
Reminder: Encoding Positive Integers

- Bit $i$ in a binary representation (in right-to-left order) is assigned weight $2^i$

\[
\begin{array}{cccccccccccc}
2^{11} & 2^{10} & 2^9 & 2^8 & 2^7 & 2^6 & 2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 \\
\hline
0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0
\end{array}
\]

- Value of an N-bit number is given by the formula
Reminder: Encoding Positive Integers

- Bit $i$ in a binary representation (in right-to-left order) is assigned weight $2^i$

  \[
  \begin{array}{cccccccc}
  \hline
  2^{11} & 2^{10} & 2^9 & 2^8 & 2^7 & 2^6 & 2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 \\
  \hline
  0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\
  \hline
  \end{array}
  \]

- Value of an $N$-bit number is given by the formula

  \[\nu = \sum_{i=0}^{N-1} 2^i b_i\]
Reminder: Encoding Positive Integers

- Bit $i$ in a binary representation (in right-to-left order) is assigned weight $2^i$

<table>
<thead>
<tr>
<th>$2^{11}$</th>
<th>$2^{10}$</th>
<th>$2^9$</th>
<th>$2^8$</th>
<th>$2^7$</th>
<th>$2^6$</th>
<th>$2^5$</th>
<th>$2^4$</th>
<th>$2^3$</th>
<th>$2^2$</th>
<th>$2^1$</th>
<th>$2^0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- Value of an $N$-bit number is given by the formula

$$v = \sum_{i=0}^{N-1} 2^i b_i$$

- What value does 011111010000 encode?
Reminder: Encoding Positive Integers

- Bit $i$ in a binary representation (in right-to-left order) is assigned weight $2^i$

<table>
<thead>
<tr>
<th>$2^0$</th>
<th>$2^1$</th>
<th>$2^2$</th>
<th>$2^3$</th>
<th>$2^4$</th>
<th>$2^5$</th>
<th>$2^6$</th>
<th>$2^7$</th>
<th>$2^8$</th>
<th>$2^9$</th>
<th>$2^{10}$</th>
<th>$2^{11}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- Value of an $N$-bit number is given by the formula

$$V = \sum_{i=0}^{N-1} 2^i b_i$$

- What value does 011111010000 encode?

$$V = 0 \times 2^{11} + 1 \times 2^{10} + 1 \times 2^9 + ...$$
Reminder: Encoding Positive Integers

- Bit $i$ in a binary representation (in right-to-left order) is assigned weight $2^i$

<table>
<thead>
<tr>
<th>$2^{11}$</th>
<th>$2^{10}$</th>
<th>$2^9$</th>
<th>$2^8$</th>
<th>$2^7$</th>
<th>$2^6$</th>
<th>$2^5$</th>
<th>$2^4$</th>
<th>$2^3$</th>
<th>$2^2$</th>
<th>$2^1$</th>
<th>$2^0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- Value of an N-bit number is given by the formula $v = \sum_{i=0}^{N-1} 2^i b_i$

- What value does 011111010000 encode?

$$V = 0 \times 2^{11} + 1 \times 2^{10} + 1 \times 2^9 + \ldots$$
$$= 1024 + 512 + 256 + 128 + 64 + 16 = 2000$$
Reminder: Encoding Positive Integers

- Bit $i$ in a binary representation (in right-to-left order) is assigned weight $2^i$

<table>
<thead>
<tr>
<th>2^11</th>
<th>2^10</th>
<th>2^9</th>
<th>2^8</th>
<th>2^7</th>
<th>2^6</th>
<th>2^5</th>
<th>2^4</th>
<th>2^3</th>
<th>2^2</th>
<th>2^1</th>
<th>2^0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- Value of an N-bit number is given by the formula

$$v = \sum_{i=0}^{N-1} 2^i b_i$$

- What value does 011111010000 encode?

$$V = 0*2^{11} + 1*2^{10} + 1*2^9 + ...$$

$$= 1024 + 512 + 256 + 128 + 64 + 16 = 2000$$

Smallest number? Largest number?
Reminder: Encoding Positive Integers

- Bit $i$ in a binary representation (in right-to-left order) is assigned weight $2^i$

<table>
<thead>
<tr>
<th>$2^{11}$</th>
<th>$2^{10}$</th>
<th>$2^9$</th>
<th>$2^8$</th>
<th>$2^7$</th>
<th>$2^6$</th>
<th>$2^5$</th>
<th>$2^4$</th>
<th>$2^3$</th>
<th>$2^2$</th>
<th>$2^1$</th>
<th>$2^0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

- Value of an $N$-bit number is given by the formula

$$v = \sum_{i=0}^{N-1} 2^i b_i$$

- What value does 011111010000 encode?

$$V = 0 \times 2^{11} + 1 \times 2^{10} + 1 \times 2^9 + ...$$

$$= 1024 + 512 + 256 + 128 + 64 + 16 = 2000$$

Smallest number? 0

Largest number?
Reminder: Encoding Positive Integers

- Bit $i$ in a binary representation (in right-to-left order) is assigned weight $2^i$

<table>
<thead>
<tr>
<th></th>
<th>$2^{11}$</th>
<th>$2^{10}$</th>
<th>$2^9$</th>
<th>$2^8$</th>
<th>$2^7$</th>
<th>$2^6$</th>
<th>$2^5$</th>
<th>$2^4$</th>
<th>$2^3$</th>
<th>$2^2$</th>
<th>$2^1$</th>
<th>$2^0$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- Value of an N-bit number is given by the formula

\[ v = \sum_{i=0}^{N-1} 2^i b_i \]

- What value does 011111010000 encode?

\[ V = 0 \times 2^{11} + 1 \times 2^{10} + 1 \times 2^9 + \ldots \]

\[ = 1024 + 512 + 256 + 128 + 64 + 16 = 2000 \]

Smallest number? 0  
Largest number? $2^{N-1}$
Hexadecimal Notation

Long strings of bits are tedious and error-prone to transcribe, so we often use a higher-radix notation, choosing the radix so that it is simple to recover the original bit string.

A popular choice is to use base-16, called hexadecimal. Each group of 4 adjacent bits is encoded as a single hexadecimal digit.

\[
\begin{array}{cccccccccc}
2^{11} & 2^{10} & 2^9 & 2^8 & 2^7 & 2^6 & 2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 \\
0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0
\end{array}
\]
Hexadecimal Notation

Long strings of bits are tedious and error-prone to transcribe, so we often use a higher-radix notation, choosing the radix so that it is simple to recover the original bit string.

A popular choice is to use base-16, called hexadecimal. Each group of 4 adjacent bits is encoded as a single hexadecimal digit.

<table>
<thead>
<tr>
<th>Hexadecimal - base 16</th>
<th>2^11 2^10 2^9 2^8 2^7 2^6 2^5 2^4 2^3 2^2 2^1 2^0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 - 0 1000 - 8</td>
<td>0 1 1 1 1 1 1 0 1 0 0 0 0</td>
</tr>
<tr>
<td>0001 - 1 1001 - 9</td>
<td></td>
</tr>
<tr>
<td>0010 - 2 1010 - A</td>
<td></td>
</tr>
<tr>
<td>0011 - 3 1011 - B</td>
<td></td>
</tr>
<tr>
<td>0100 - 4 1100 - C</td>
<td></td>
</tr>
<tr>
<td>0101 - 5 1101 - D</td>
<td></td>
</tr>
<tr>
<td>0110 - 6 1110 - E</td>
<td></td>
</tr>
<tr>
<td>0111 - 7 1111 - F</td>
<td></td>
</tr>
</tbody>
</table>
Hexadecimal Notation

Long strings of bits are tedious and error-prone to transcribe, so we often use a higher-radix notation, choosing the radix so that it is simple to recover the original bit string.

A popular choice is to use base-16, called hexadecimal. Each group of 4 adjacent bits is encoded as a single hexadecimal digit.

Hexadecimal - base 16

<table>
<thead>
<tr>
<th>Hexadecimal</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 - 0</td>
<td>0000</td>
</tr>
<tr>
<td>0001 - 1</td>
<td>0001</td>
</tr>
<tr>
<td>0010 - 2</td>
<td>0010</td>
</tr>
<tr>
<td>0011 - 3</td>
<td>0011</td>
</tr>
<tr>
<td>0100 - 4</td>
<td>0100</td>
</tr>
<tr>
<td>0101 - 5</td>
<td>0101</td>
</tr>
<tr>
<td>0110 - 6</td>
<td>0110</td>
</tr>
<tr>
<td>0111 - 7</td>
<td>0111</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2(^0)</th>
<th>2(^1)</th>
<th>2(^2)</th>
<th>2(^3)</th>
<th>2(^4)</th>
<th>2(^5)</th>
<th>2(^6)</th>
<th>2(^7)</th>
<th>2(^8)</th>
<th>2(^9)</th>
<th>2(^10)</th>
<th>2(^11)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Hexadecimal Notation

Long strings of bits are tedious and error-prone to transcribe, so we often use a higher-radix notation, choosing the radix so that it is simple to recover the original bit string.

A popular choice is to use base-16, called hexadecimal. Each group of 4 adjacent bits is encoded as a single hexadecimal digit.

Hexadecimal - base 16

<table>
<thead>
<tr>
<th>Hexadecimal</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>0001</td>
<td>0001</td>
</tr>
<tr>
<td>0010</td>
<td>0010</td>
</tr>
<tr>
<td>0011</td>
<td>0011</td>
</tr>
<tr>
<td>0100</td>
<td>0100</td>
</tr>
<tr>
<td>0101</td>
<td>0101</td>
</tr>
<tr>
<td>0110</td>
<td>0110</td>
</tr>
<tr>
<td>0111</td>
<td>0111</td>
</tr>
</tbody>
</table>

2^{11} 2^{10} 2^9 2^8 2^7 2^6 2^5 2^4 2^3 2^2 2^1 2^0

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

D 0
Hexadecimal Notation

Long strings of bits are tedious and error-prone to transcribe, so we often use a higher-radix notation, choosing the radix so that it is simple to recover the original bit string.

A popular choice is to use base-16, called hexadecimal. Each group of 4 adjacent bits is encoded as a single hexadecimal digit.

Hexadecimal - base 16

<table>
<thead>
<tr>
<th>Hexadecimal</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 - 0</td>
<td>0</td>
</tr>
<tr>
<td>0001 - 1</td>
<td>1</td>
</tr>
<tr>
<td>0010 - 2</td>
<td>2</td>
</tr>
<tr>
<td>0011 - 3</td>
<td>3</td>
</tr>
<tr>
<td>0100 - 4</td>
<td>4</td>
</tr>
<tr>
<td>0101 - 5</td>
<td>5</td>
</tr>
<tr>
<td>0110 - 6</td>
<td>6</td>
</tr>
<tr>
<td>0111 - 7</td>
<td>7</td>
</tr>
<tr>
<td>1000 - 8</td>
<td>8</td>
</tr>
<tr>
<td>1001 - 9</td>
<td>9</td>
</tr>
<tr>
<td>1010 - A</td>
<td>10</td>
</tr>
<tr>
<td>1011 - B</td>
<td>11</td>
</tr>
<tr>
<td>1100 - C</td>
<td>12</td>
</tr>
<tr>
<td>1101 - D</td>
<td>13</td>
</tr>
<tr>
<td>1110 - E</td>
<td>14</td>
</tr>
<tr>
<td>1111 - F</td>
<td>15</td>
</tr>
</tbody>
</table>
Hexadecimal Notation

Long strings of bits are tedious and error-prone to transcribe, so we often use a higher-radix notation, choosing the radix so that it is simple to recover the original bit string.

A popular choice is to use base-16, called **hexadecimal**. Each group of 4 adjacent bits is encoded as a single hexadecimal digit.

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

\[\text{0b}011111010000 = \text{0x7D0}\]
Binary Addition and Subtraction

- Addition and subtraction in base 2 are performed just like in base 10
Binary Addition and Subtraction

- Addition and subtraction in base 2 are performed just like in base 10

  Base 10

  14
  + 7
  ---
  21
Binary Addition and Subtraction

- Addition and subtraction in base 2 are performed just like in base 10

Base 10

\[
\begin{array}{c}
14 \\
+ 7 \\
\hline
1 \\
\end{array}
\]
Binary Addition and Subtraction

- Addition and subtraction in base 2 are performed just like in base 10

Base 10

\[
\begin{array}{c}
14 \\
+ 7 \\
\hline
1 \\
\end{array}
\]

carry
Binary Addition and Subtraction

- Addition and subtraction in base 2 are performed just like in base 10

Base 10

```
  14
+  7
---
  21
```
Binary Addition and Subtraction

- Addition and subtraction in base 2 are performed just like in base 10

<table>
<thead>
<tr>
<th>Base 10</th>
<th>Base 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>1110</td>
</tr>
<tr>
<td>+ 7</td>
<td>+ 111</td>
</tr>
<tr>
<td>21</td>
<td></td>
</tr>
</tbody>
</table>
Binary Addition and Subtraction

- Addition and subtraction in base 2 are performed just like in base 10

<table>
<thead>
<tr>
<th>Base 10</th>
<th>Base 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>1110</td>
</tr>
<tr>
<td>+ 7</td>
<td>+ 111</td>
</tr>
<tr>
<td>21</td>
<td>1</td>
</tr>
</tbody>
</table>

*carry*
Binary Addition and Subtraction

- Addition and subtraction in base 2 are performed just like in base 10

<table>
<thead>
<tr>
<th>Base 10</th>
<th>Base 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>1110</td>
</tr>
<tr>
<td>+ 7</td>
<td>+ 111</td>
</tr>
<tr>
<td>21</td>
<td>01</td>
</tr>
</tbody>
</table>

*carry*
Binary Addition and Subtraction

- Addition and subtraction in base 2 are performed just like in base 10

Base 10       Base 2

14 + 7 = 21   1110 + 111 = 101
## Binary Addition and Subtraction

- Addition and subtraction in base 2 are performed just like in base 10

<table>
<thead>
<tr>
<th>Base 10</th>
<th>Base 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>1110</td>
</tr>
<tr>
<td>+ 7</td>
<td>+ 111</td>
</tr>
<tr>
<td>21</td>
<td>0101</td>
</tr>
</tbody>
</table>

*carry*
Binary Addition and Subtraction

- Addition and subtraction in base 2 are performed just like in base 10

<table>
<thead>
<tr>
<th>Base 10</th>
<th>Base 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>1110</td>
</tr>
<tr>
<td>+ 7</td>
<td>+ 111</td>
</tr>
<tr>
<td>21</td>
<td>10101</td>
</tr>
</tbody>
</table>

**carry**

Base 10: 14 + 7 = 21
Base 2: 1110 + 111 = 10101
Binary Addition and Subtraction

- Addition and subtraction in base 2 are performed just like in base 10

<table>
<thead>
<tr>
<th>Base 10</th>
<th>Base 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>111</td>
</tr>
<tr>
<td>+ 7</td>
<td>1110</td>
</tr>
<tr>
<td>21</td>
<td>10101</td>
</tr>
</tbody>
</table>

14
- 7
---

1110
+ 111
---
Binary Addition and Subtraction

- Addition and subtraction in base 2 are performed just like in base 10

Base 10

\[
\begin{array}{c}
14 \\
+ 7 \\
\hline
21
\end{array}
\]

Base 2

\[
\begin{array}{c}
1110 \\
+ 111 \\
\hline
10101
\end{array}
\]

\[
\begin{array}{c}
14 \\
- 7 \\
\hline
7
\end{array}
\]

\[\text{carry}\]
Binary Addition and Subtraction

- Addition and subtraction in base 2 are performed just like in base 10

Base 10

\[
\begin{align*}
14 & \quad + \\ 7 & \quad = \\ 21 & \quad \text{carry}
\end{align*}
\]

Base 2

\[
\begin{align*}
111 & \quad + \\ 1110 & \quad = \\ 10101 & \quad \text{borrow}
\end{align*}
\]
Binary Addition and Subtraction

- Addition and subtraction in base 2 are performed just like in base 10

Base 10 | Base 2
---|---
14 + 7 | 111 + 111
---|---
21 | 10101

-1 | borrow
---|---
14 - 7 | 1110 - 111
---|---
07 | 10101
Binary Addition and Subtraction

- Addition and subtraction in base 2 are performed just like in base 10

### Base 10

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>1110</td>
</tr>
<tr>
<td>+ 7</td>
<td>111</td>
</tr>
<tr>
<td></td>
<td>10101</td>
</tr>
<tr>
<td>21</td>
<td></td>
</tr>
</tbody>
</table>

- **carry**

### Base 2

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>111</td>
<td>1110</td>
</tr>
<tr>
<td>+ 111</td>
<td>10101</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **borrow**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>1110</td>
</tr>
<tr>
<td>− 7</td>
<td>111</td>
</tr>
<tr>
<td>07</td>
<td></td>
</tr>
</tbody>
</table>

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>111</td>
<td>111</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

September 18, 2018
### Binary Addition and Subtraction

- Addition and subtraction in base 2 are performed just like in base 10

<table>
<thead>
<tr>
<th>Base 10</th>
<th>Base 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>1110</td>
</tr>
<tr>
<td>+ 7</td>
<td>+ 111</td>
</tr>
<tr>
<td><strong>21</strong></td>
<td><strong>10101</strong></td>
</tr>
<tr>
<td><strong>1</strong> → carry</td>
<td><strong>111</strong> → borrow</td>
</tr>
</tbody>
</table>

| 14      | 1110 |
| - 7     | - 111 |
| **07**  | **1** |
| **-1** → borrow | **-1** → carry |
Binary Addition and Subtraction

- Addition and subtraction in base 2 are performed just like in base 10

<table>
<thead>
<tr>
<th>Base 10</th>
<th>Base 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 + 14 + 7</td>
<td>111 + 111</td>
</tr>
<tr>
<td>21</td>
<td>10101</td>
</tr>
</tbody>
</table>

Carry

<table>
<thead>
<tr>
<th>Base 10</th>
<th>Base 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1 + 14 - 7</td>
<td>-1-1 + 111</td>
</tr>
<tr>
<td>07</td>
<td>11</td>
</tr>
</tbody>
</table>

Borrow
Binary Addition and Subtraction

- Addition and subtraction in base 2 are performed just like in base 10

<table>
<thead>
<tr>
<th>Base 10</th>
<th>Base 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>14 + 7</td>
<td>1110 + 111</td>
</tr>
<tr>
<td>21</td>
<td>10101</td>
</tr>
</tbody>
</table>

1 (carry)

-1 (borrow)

<table>
<thead>
<tr>
<th>Base 10</th>
<th>Base 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>14 - 7</td>
<td>1110 - 111</td>
</tr>
<tr>
<td>07</td>
<td>111</td>
</tr>
</tbody>
</table>
Binary Addition and Subtraction

- Addition and subtraction in base 2 are performed just like in base 10

<table>
<thead>
<tr>
<th>Base 10</th>
<th>Base 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>14 + 7</td>
<td>1110 + 111</td>
</tr>
<tr>
<td>21</td>
<td>10101</td>
</tr>
</tbody>
</table>

-1 borrow

<table>
<thead>
<tr>
<th>Base 10</th>
<th>Base 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>14 - 7</td>
<td>1110 - 111</td>
</tr>
<tr>
<td>07</td>
<td>0111</td>
</tr>
</tbody>
</table>
Binary Addition and Subtraction

- Addition and subtraction in base 2 are performed just like in base 10

<table>
<thead>
<tr>
<th>Base 10</th>
<th>Base 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>14 + 7</td>
<td>1110 + 111</td>
</tr>
<tr>
<td>21</td>
<td>10101</td>
</tr>
</tbody>
</table>

-1 borrowing

<table>
<thead>
<tr>
<th>Base 10</th>
<th>Base 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>14 - 7</td>
<td>1110 - 111</td>
</tr>
<tr>
<td>07</td>
<td>0111</td>
</tr>
</tbody>
</table>

- 101
Binary Addition and Subtraction

- Addition and subtraction in base 2 are performed just like in base 10

<table>
<thead>
<tr>
<th>Base 10</th>
<th>Base 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>1110</td>
</tr>
<tr>
<td>+ 7</td>
<td>+ 111</td>
</tr>
<tr>
<td>21</td>
<td>10101</td>
</tr>
</tbody>
</table>

-1— borrow

<table>
<thead>
<tr>
<th>Base 10</th>
<th>Base 2</th>
<th>2's complement</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>1110</td>
<td>0111</td>
</tr>
<tr>
<td>-7</td>
<td>-111</td>
<td>-101</td>
</tr>
<tr>
<td>07</td>
<td>0111</td>
<td>0</td>
</tr>
</tbody>
</table>
### Binary Addition and Subtraction

- Addition and subtraction in base 2 are performed just like in base 10

<table>
<thead>
<tr>
<th>Base 10</th>
<th>Base 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>1110</td>
</tr>
<tr>
<td>+ 7</td>
<td>+ 111</td>
</tr>
<tr>
<td>21</td>
<td>10101</td>
</tr>
</tbody>
</table>

- **carry**

<table>
<thead>
<tr>
<th>-1</th>
<th>-1-1-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>1110</td>
</tr>
<tr>
<td>- 7</td>
<td>- 111</td>
</tr>
<tr>
<td>07</td>
<td>0111</td>
</tr>
</tbody>
</table>

- **borrow**

<table>
<thead>
<tr>
<th>-101</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>-101</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>011</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>10</th>
</tr>
</thead>
</table>

September 18, 2018
Binary Addition and Subtraction

- Addition and subtraction in base 2 are performed just like in base 10

<table>
<thead>
<tr>
<th>Base 10</th>
<th>Base 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>1110</td>
</tr>
<tr>
<td>+ 7</td>
<td>+ 111</td>
</tr>
<tr>
<td>21</td>
<td>10101</td>
</tr>
</tbody>
</table>

-1 carry

<table>
<thead>
<tr>
<th>Base 10</th>
<th>Base 2</th>
<th>Check</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>1110</td>
<td>0111</td>
</tr>
<tr>
<td>- 7</td>
<td>- 111</td>
<td>- 101</td>
</tr>
<tr>
<td>07</td>
<td>0111</td>
<td>110</td>
</tr>
</tbody>
</table>
Binary Addition and Subtraction

- Addition and subtraction in base 2 are performed just like in base 10

<table>
<thead>
<tr>
<th>Base 10</th>
<th>Base 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>1110</td>
</tr>
<tr>
<td>+ 7</td>
<td>+ 111</td>
</tr>
<tr>
<td>21</td>
<td>10101</td>
</tr>
</tbody>
</table>

**carry**

-1 | borrow |

<table>
<thead>
<tr>
<th>Base 10</th>
<th>Base 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>1110</td>
</tr>
<tr>
<td>- 7</td>
<td>- 111</td>
</tr>
<tr>
<td>07</td>
<td>0111</td>
</tr>
</tbody>
</table>

**borrow**
Binary Addition and Subtraction

- Addition and subtraction in base 2 are performed just like in base 10

<table>
<thead>
<tr>
<th>Base 10</th>
<th>Base 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>14 + 7</td>
<td>1110 + 111</td>
</tr>
<tr>
<td>21</td>
<td>10101</td>
</tr>
</tbody>
</table>

-1 borrow

<table>
<thead>
<tr>
<th>Base 10</th>
<th>Base 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>14 - 7</td>
<td>1110 - 111</td>
</tr>
<tr>
<td>07</td>
<td>0111</td>
</tr>
</tbody>
</table>

-1

<table>
<thead>
<tr>
<th>Base 10</th>
<th>Base 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>14 - 7</td>
<td>1110 - 101</td>
</tr>
<tr>
<td>07</td>
<td>0111 ??? 110</td>
</tr>
</tbody>
</table>
Binary Addition and Subtraction

- Addition and subtraction in base 2 are performed just like in base 10

<table>
<thead>
<tr>
<th>Base 10</th>
<th>Base 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>1110</td>
</tr>
<tr>
<td>+ 7</td>
<td>+ 111</td>
</tr>
<tr>
<td>21</td>
<td>10101</td>
</tr>
</tbody>
</table>

-1 (carry)

<table>
<thead>
<tr>
<th>Base 10</th>
<th>Base 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>1110</td>
</tr>
<tr>
<td>- 7</td>
<td>- 111</td>
</tr>
<tr>
<td>07</td>
<td>0111</td>
</tr>
</tbody>
</table>

-1 (borrow)

<table>
<thead>
<tr>
<th>Base 10</th>
<th>Base 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>1110</td>
</tr>
<tr>
<td>- 7</td>
<td>- 101</td>
</tr>
<tr>
<td>07</td>
<td>0111</td>
</tr>
</tbody>
</table>

What does this mean?
Binary Addition and Subtraction

- Addition and subtraction in base 2 are performed just like in base 10

<table>
<thead>
<tr>
<th>Base 10</th>
<th>Base 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>14 + 7</td>
<td>1110 + 111</td>
</tr>
<tr>
<td>21</td>
<td>10101</td>
</tr>
<tr>
<td></td>
<td>What does this mean? -2^3 + 0b110</td>
</tr>
</tbody>
</table>

-1 borrow

<table>
<thead>
<tr>
<th>Base 10</th>
<th>Base 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>14 - 7</td>
<td>1110 - 111</td>
</tr>
<tr>
<td>07</td>
<td>0111 - 101</td>
</tr>
<tr>
<td></td>
<td>??? 110</td>
</tr>
</tbody>
</table>
### Binary Addition and Subtraction

- Addition and subtraction in base 2 are performed just like in base 10

<table>
<thead>
<tr>
<th>Base 10</th>
<th>Base 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>1110</td>
</tr>
<tr>
<td>+ 7</td>
<td>+ 111</td>
</tr>
<tr>
<td>21</td>
<td>10101</td>
</tr>
</tbody>
</table>

**What does this mean?**

- $-2^3 + 0b110$
- We need a way to represent negative numbers!
Binary Modular Arithmetic

- If we use a fixed number of bits, addition and other operations may produce results outside the range that the output can represent (up to 1 extra bit for addition)
  - This is known as an overflow
Binary Modular Arithmetic

- If we use a fixed number of bits, addition and other operations may produce results outside the range that the output can represent (up to 1 extra bit for addition)
  - This is known as an overflow
- Common approach: Ignore the extra bit
  - Gives rise to modular arithmetic: With N-bit numbers, equivalent to following all operations with mod $2^N$
If we use a fixed number of bits, addition and other operations may produce results outside the range that the output can represent (up to 1 extra bit for addition)

- This is known as an overflow

Common approach: Ignore the extra bit

- Gives rise to modular arithmetic: With N-bit numbers, equivalent to following all operations with \( \text{mod } 2^N \)
- Visually, numbers “wrap around”:
Binary Modular Arithmetic

- If we use a fixed number of bits, addition and other operations may produce results outside the range that the output can represent (up to 1 extra bit for addition)
  - This is known as an overflow
- Common approach: Ignore the extra bit
  - Gives rise to modular arithmetic: With N-bit numbers, equivalent to following all operations with mod $2^N$
  - Visually, numbers “wrap around”:
Binary Modular Arithmetic

- If we use a fixed number of bits, addition and other operations may produce results outside the range that the output can represent (up to 1 extra bit for addition)
  - This is known as an overflow
- Common approach: Ignore the extra bit
  - Gives rise to modular arithmetic: With N-bit numbers, equivalent to following all operations with \( \text{mod } 2^N \)
  - Visually, numbers “wrap around”:

Example: \((3 - 5) \text{ mod } 2^3\)?
Binary Modular Arithmetic

- If we use a fixed number of bits, addition and other operations may produce results outside the range that the output can represent (up to 1 extra bit for addition)
  - This is known as an overflow
- Common approach: Ignore the extra bit
  - Gives rise to modular arithmetic: With N-bit numbers, equivalent to following all operations with $\mod 2^N$
  - Visually, numbers “wrap around”:

  \[
  \text{Example: } (3 - 5) \mod 2^3
  \]
Binary Modular Arithmetic

- If we use a fixed number of bits, addition and other operations may produce results outside the range that the output can represent (up to 1 extra bit for addition)
  - This is known as an **overflow**
- Common approach: Ignore the extra bit
  - Gives rise to **modular arithmetic**: With N-bit numbers, equivalent to following all operations with \( \text{mod } 2^N \)
  - Visually, numbers “wrap around”:

**Example**: \((3 - 5) \mod 2^3 \)?
Encoding Negative Integers

We use sign-magnitude representation for decimal numbers, encoding the number’s sign (using “+” and “-“) separately from its magnitude (using decimal digits).

Attempt #1: Use the same approach for binary numbers:

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

=-2000
Encoding Negative Integers

We use sign-magnitude representation for decimal numbers, encoding the number’s sign (using “+” and “−”) separately from its magnitude (using decimal digits).

Attempt #1: Use the same approach for binary numbers:

```
1 1 1 1 1 1 1 0 1 0 0 0 0 0
```

“0” for “+”
“1” for “-”

= -2000
Encoding Negative Integers

We use **sign-magnitude representation** for decimal numbers, encoding the number’s sign (using “+” and “-”) separately from its magnitude (using decimal digits).

Attempt #1: Use the same approach for binary numbers:

```
1 1 1 1 1 1 1 0 1 0 0 0 0 0
```

“0” for “+”
“1” for “-”

What issues might this encoding have?
Encoding Negative Integers

We use sign-magnitude representation for decimal numbers, encoding the number’s sign (using “+” and “-”) separately from its magnitude (using decimal digits).

Attempt #1: Use the same approach for binary numbers:

```
1 1 1 1 1 1 1 0 1 0 0 0 0 0
```

“0” for “+”
“1” for “-”

What issues might this encoding have?

Two representations for 0 (+0, -0)
Encoding Negative Integers

We use sign-magnitude representation for decimal numbers, encoding the number’s sign (using “+” and “-”) separately from its magnitude (using decimal digits).

Attempt #1: Use the same approach for binary numbers:

```
 1 1 1 1 1 1 1 0 1 0 0 0 0 0
```

“0” for “+”  “1” for “-”

What issues might this encoding have?

- Two representations for 0 (+0, -0)
- Circuits for addition and subtraction are different and more complex than with unsigned numbers
Can you simply relabel some of the digits to represent negative numbers while retaining the nice properties of modular arithmetic?
Can you simply relabel some of the digits to represent negative numbers while retaining the nice properties of modular arithmetic? Yes!
Can you simply relabel some of the digits to represent negative numbers while retaining the nice properties of modular arithmetic? Yes!
Deriving a Better Encoding

Can you simply relabel some of the digits to represent negative numbers while retaining the nice properties of modular arithmetic? Yes!
Can you simply relabel some of the digits to represent negative numbers while retaining the nice properties of modular arithmetic? Yes!
Can you simply relabel some of the digits to represent negative numbers while retaining the nice properties of modular arithmetic? Yes!
Can you simply relabel some of the digits to represent negative numbers while retaining the nice properties of modular arithmetic? Yes!

This is called two’s complement encoding.
Two’s Complement Encoding

In two’s complement encoding, the high-order bit of the N-bit representation has negative weight:

\[ v = -2^{N-1}b_{N-1} + \sum_{i=0}^{N-2} 2^i b_i \]

- Negative numbers have “1” in the high-order bit
- **Most negative number?** 10...0000 \(-2^{N-1}\)
- **Most positive number?** 01...1111 \(+2^{N-1} - 1\)
- **If all bits are 1?** 11...1111 \(-1\)
Two’s Complement and Arithmetic

- To negate a number (i.e., compute $-A$ given $A$), we invert all the bits and add one
Two’s Complement and Arithmetic

- To negate a number (i.e., compute $-A$ given $A$), we invert all the bits and add one

*Why does this work?*
Two’s Complement and Arithmetic

- To negate a number (i.e., compute \(-A\) given \(A\)), we invert all the bits and add one

Why does this work?

\[-A + A = 0 = -1 + 1\]
Two’s Complement and Arithmetic

- To negate a number (i.e., compute \(-A\) given \(A\)), we invert all the bits and add one.

*Why does this work?*

\[
-A + A = 0 = -1 + 1
\]
\[
-A = (-1 - A) + 1
\]
Two’s Complement and Arithmetic

- To negate a number (i.e., compute \(-A\) given \(A\)), we invert all the bits and add one

*Why does this work?*

\[-A + A = 0 = -1 + 1\]

\[-A = (-1 - A) + 1\]

\[
\begin{array}{c}
1 \ldots 1 1 \\
\hline
-A_{n-1} \ldots A_1 A_0
\end{array}
\]
Two’s Complement and Arithmetic

- To negate a number (i.e., compute \(-A\) given \(A\)), we invert all the bits and add one

\[ -A + A = 0 = -1 + 1 \]

Why does this work?

\[ -A = (-1 \cdot A) + 1 \]

\[
\begin{array}{c}
1 \ldots 1 \\
\hline
A_{n-1} \ldots A_1 A_0 \\
\hline
\overline{A_{n-1}} \ldots \overline{A_1} \overline{A_0}
\end{array}
\]
Two’s Complement and Arithmetic

- To negate a number (i.e., compute \(-A\) given \(A\)), we invert all the bits and add one

  \[A + A = 0 = -1 + 1\]

  \[-A = (-1 - A) + 1\]

  \[\begin{array}{c}
  1 \ldots 1 \\
  \hline
  -A_{n-1} \ldots A_1 A_0 \\
  \hline
  \overline{A}_{n-1} \ldots \overline{A}_1 \overline{A}_0
  \end{array}\]

- To compute \(A - B\), we can simply use addition and compute \(A + (-B)\)
  - Same circuit can add and subtract!
Two’s Complement Example

Compute $3 - 6$ using 4-bit 2’s complement addition
Two’s Complement Example

Compute $3 - 6$ using 4-bit 2’s complement addition

- $3: 0011$
Two’s Complement Example

Compute $3 - 6$ using 4-bit 2’s complement addition

- $3: 0011$
- $6: 0110$
Two’s Complement Example

Compute $3 - 6$ using 4-bit 2’s complement addition

- $3: 0011$
- $6: 0110$
- $-6: 1010$
Two’s Complement Example

Compute 3 − 6 using 4-bit 2’s complement addition

- 3: 0011
- 6: 0110
- -6: 1010


0011

+ 1010
Two’s Complement Example

*Compute 3 – 6 using 4-bit 2’s complement addition*

- 3: 0011
- 6: 0110
- -6: 1010

\[ \begin{array}{c}
\text{0011} \\
\text{+ 1010} \\
\hline
\text{1}\end{array} \]
Two’s Complement Example

Compute $3 - 6$ using 4-bit 2’s complement addition

- $3: 0011$
- $6: 0110$
- $-6: 1010$

\[
\begin{array}{c}
1 \\
0011 \\
+ 1010 \\
01
\end{array}
\]
Two’s Complement Example

Compute $3 - 6$ using 4-bit 2’s complement addition

- $3$: 0011
- $6$: 0110
- $-6$: 1010

\[
\begin{array}{c}
\text{1} \\
0011 \\
+ \\
1010 \\
\hline
101
\end{array}
\]
Two’s Complement Example

*Compute 3 – 6 using 4-bit 2’s complement addition*

- 3: 0011
- 6: 0110
- -6: 1010

\[
\begin{array}{c}
3: 0011 \\
+ 1010 \\
\hline 
1101
\end{array}
\]

\[1 + 0011 \rightarrow 1010 \rightarrow 1101\]
Two’s Complement Example

Compute $3 - 6$ using 4-bit 2’s complement addition

- $3: 0011$
- $6: 0110$
- $-6: 1010$

\[
\begin{array}{c}
3: 0011 \\
+ 1010 \\
\hline
1101
\end{array}
\]

Compute $3 - 2$ using 3-bit 2’s complement addition
Two’s Complement Example

Compute 3 – 6 using 4-bit 2’s complement addition

- 3: 0011
- 6: 0110
- -6: 1010

\[
\begin{array}{c}
3 \quad \text{0011} \\
1 \\
\text{0011} \\
\hline
\text{1101}
\end{array}
\]

Compute 3 – 2 using 3-bit 2’s complement addition

- 3: 011
Two’s Complement Example

Compute 3 – 6 using 4-bit 2’s complement addition

- 3: 0011
- 6: 0110
- -6: 1010

\[ \begin{array}{c}
1 \\
0011 \\
+ \\
1010 \\
\hline
1101 \\
\end{array} \]

Compute 3 – 2 using 3-bit 2’s complement addition

- 3: 011
- 2: 010
Two’s Complement Example

Compute $3 - 6$ using 4-bit 2’s complement addition

- $3: 0011$
- $6: 0110$
- $-6: 1010$

$$
\begin{array}{c}
\text{1} \\
\text{0011} \\
+ \text{1010} \\
\hline
\text{1101}
\end{array}
$$

Compute $3 - 2$ using 3-bit 2’s complement addition

- $3: 011$
- $2: 010$
- $-2: 110$
Two’s Complement Example

Compute $3 - 6$ using 4-bit 2’s complement addition

- $3$: 0011
- $6$: 0110
- $-6$: 1010

$$
\begin{array}{c}
\text{1} \\
0011 \\
+ \\
1010 \\
\hline
1101 \\
\end{array}
$$

Compute $3 - 2$ using 3-bit 2’s complement addition

- $3$: 011
- $2$: 010
- $-2$: 110

$$
\begin{array}{c}
011 \\
+ \\
110 \\
\hline
111 \\
\end{array}
$$
Two’s Complement Example

**Compute 3 \(- 6\) using 4-bit 2’s complement addition**

- 3: 0011
- 6: 0110
- \(-6\): 1010

\[
\begin{array}{c}
\text{0011} \\
+ \text{1010} \\
\hline
\text{1101}
\end{array}
\]

**Compute 3 \(- 2\) using 3-bit 2’s complement addition**

- 3: 011
- 2: 010
- \(-2\): 110

\[
\begin{array}{c}
\text{011} \\
+ \text{110} \\
\hline
\text{111}
\end{array}
\]

\[
\text{1}
\]
Two’s Complement Example

Compute 3 – 6 using 4-bit 2’s complement addition
- 3: 0011
- 6: 0110
- -6: 1010

```
  0011
+ 1010
1101
```

Compute 3 – 2 using 3-bit 2’s complement addition
- 3: 011
- 2: 010
- -2: 110

```
  011
+ 110
  01
```
Two’s Complement Example

Compute 3 – 6 using 4-bit 2’s complement addition

- 3: 0011
- 6: 0110
- -6: 1010

\[
\begin{array}{c}
0011 \\
+ 1010 \\
\hline
1101
\end{array}
\]

Compute 3 – 2 using 3-bit 2’s complement addition

- 3: 011
- 2: 010
- -2: 110

\[
\begin{array}{c}
11 \\
011 \\
+ 110 \\
\hline
001
\end{array}
\]
Two’s Complement Example

Compute 3 – 6 using 4-bit 2’s complement addition

- 3: 0011
- 6: 0110
- -6: 1010

\[
\begin{array}{c}
1 \\
0011 \\
+ 1010 \\
\hline
1101
\end{array}
\]

Compute 3 – 2 using 3-bit 2’s complement addition

- 3: 011
- 2: 010
- -2: 110

\[
\begin{array}{c}
11 \\
011 \\
+ 110 \\
\hline
1001
\end{array}
\]
Two’s Complement Example

Compute 3 – 6 using 4-bit 2’s complement addition

- 3: 0011
- 6: 0110
- -6: 1010

\[0011 + 1010 = 1101\]

Compute 3 – 2 using 3-bit 2’s complement addition

- 3: 011
- 2: 010
- -2: 110

\[011 + 110 = 1001\]

Keep only last 3 bits
Two’s Complement Example

Compute $3 - 6$ using 4-bit 2’s complement addition

- $3: 0011$
- $6: 0110$
- $-6: 1010$

$$
\begin{array}{c}
  0011 \\
  + 1010 \\
  \hline
  1101 \\
\end{array}
$$

Compute $3 - 2$ using 3-bit 2’s complement addition

- $3: 011$
- $2: 010$
- $-2: 110$

$$
\begin{array}{c}
  011 \\
  + 110 \\
  \hline
  1001 \\
\end{array}
$$

What does this 1 mean?

Keep only last 3 bits
Two’s Complement Example

Compute $3 - 6$ using 4-bit 2’s complement addition

- $3: 0011$
- $6: 0110$
- $-6: 1010$

$$\begin{array}{c}
0011 \\
+ 1010 \\
\hline
1101
\end{array}$$

Compute $3 - 2$ using 3-bit 2’s complement addition

- $3: 011$
- $2: 010$
- $-2: 110$

$$\begin{array}{c}
011 \\
+ 110 \\
\hline
1001
\end{array}$$

What does this $1$ mean?

Keep only last 3 bits
Two’s Complement Example

Compute \(3 - 6\) using 4-bit 2’s complement addition

- 3: 0011
- 6: 0110
- \(-6\): 1010

\[
\begin{array}{c}
3: 0011 \\
+ 6: 0110 \\
\hline
1101
\end{array}
\]

Compute \(3 - 2\) using 3-bit 2’s complement addition

- 3: 011
- 2: 010
- \(-2\): 110

\[
\begin{array}{c}
3: 011 \\
+ 2: 010 \\
\hline
1001
\end{array}
\]

Keep only last 3 bits

What does this 1 mean?
Two’s Complement Example

Compute 3 – 6 using 4-bit 2’s complement addition

- 3: 0011
- 6: 0110
- -6: 1010

\[
\begin{array}{c}
3: 0011 \\
+ 1010 \\
\hline
1101
\end{array}
\]

Compute 3 – 2 using 3-bit 2’s complement addition

- 3: 011
- 2: 010
- -2: 110

\[
\begin{array}{c}
3: 011 \\
+ 110 \\
\hline
1001
\end{array}
\]

What does this 1 mean?

Keep only last 3 bits
Two’s Complement Example

Compute 3 – 6 using 4-bit 2’s complement addition

- 3: 0011
- 6: 0110
- -6: 1010

\[\begin{align*}
3 & = 0011 \\
6 & = 0110 \\
-6 & = 1010 \\
\end{align*}\]

\[\begin{align*}
0011 + 1010 & = 1101 \\
\end{align*}\]

Compute 3 – 2 using 3-bit 2’s complement addition

- 3: 011
- 2: 010
- -2: 110

\[\begin{align*}
3 & = 011 \\
2 & = 010 \\
-2 & = 110 \\
\end{align*}\]

\[\begin{align*}
011 + 110 & = 1001 \\
\end{align*}\]

Keep only last 3 bits

What does this 1 mean?

Zero crossing
Binary Multiplication

- Multiplication is simply repeated addition

Multiplicand  1101 (13)
Multiplier    *  1011 (11)
Binary Multiplication

- Multiplication is simply repeated addition

Multiplicand  1101 (13)  
Multiplier * 1011 (11)

At each step we add either 1101 (the multiplicand) or 0 to the result depending on the corresponding bit in the multiplier.
Multiplication is simply repeated addition

Multiplicand \( \times \) Multiplier

\[
\begin{array}{c}
1101 (13) \\
1011 (11)
\end{array}
\]

\[
\begin{array}{c}
0000 \\
+ 1101
\end{array}
\]

At each step we add either 1101 (the multiplicand) or 0 to the result depending on the corresponding bit in the multiplier.
Binary Multiplication

- Multiplication is simply repeated addition

<table>
<thead>
<tr>
<th>Multiplicand</th>
<th>1101 (13)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiplier</td>
<td>1011 (11)</td>
</tr>
<tr>
<td></td>
<td>0000</td>
</tr>
<tr>
<td></td>
<td>+ 1101</td>
</tr>
<tr>
<td></td>
<td>01101</td>
</tr>
</tbody>
</table>

At each step we add either 1101 (the multiplicand) or 0 to the result depending on the corresponding bit in the multiplier.
Binary Multiplication

- Multiplication is simply repeated addition

<table>
<thead>
<tr>
<th>Multiplicand</th>
<th>1101 (13)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiplier</td>
<td>1011 (11)</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------</td>
</tr>
</tbody>
</table>

At each step we add either 1101 (the multiplicand) or 0 to the result depending on the corresponding bit in the multiplier.
Binary Multiplication

- Multiplication is simply repeated addition

<table>
<thead>
<tr>
<th>Multiplicand</th>
<th>1101 (13)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiplier</td>
<td>1011 (11)</td>
</tr>
</tbody>
</table>

At each step we add either 1101 (the multiplicand) or 0 to the result depending on the corresponding bit in the multiplier.

We also shift the result by one position at every step.
Binary Multiplication

- Multiplication is simply repeated addition

<table>
<thead>
<tr>
<th>Multiplicand</th>
<th>1101 (13)</th>
<th>Multiplier</th>
<th>1011 (11)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>0000</td>
<td>1101</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>1101</td>
<td>01101</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>1101</td>
<td>100111</td>
<td></td>
</tr>
</tbody>
</table>

At each step we add either 1101 (the multiplicand) or 0 to the result depending on the corresponding bit in the multiplier.

We also shift the result by one position at every step.
Binary Multiplication

- Multiplication is simply repeated addition

<table>
<thead>
<tr>
<th>Multiplicand</th>
<th>1101 (13)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiplier</td>
<td>1011 (11)</td>
</tr>
</tbody>
</table>

At each step we add either 1101 (the multiplicand) or 0 to the result depending on the corresponding bit in the multiplier.

We also shift the result by one position at every step.
### Binary Multiplication

- Multiplication is simply repeated addition

<table>
<thead>
<tr>
<th>Multiplicand</th>
<th>1101 (13)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiplier</td>
<td>1011 (11)</td>
</tr>
<tr>
<td><strong>0000</strong></td>
<td><strong>01101</strong></td>
</tr>
<tr>
<td>+ 1101</td>
<td><strong>01101</strong></td>
</tr>
<tr>
<td>+ 1101</td>
<td><strong>100111</strong></td>
</tr>
<tr>
<td>+ 0000</td>
<td><strong>0100111</strong></td>
</tr>
</tbody>
</table>

At each step we add either 1101 (the multiplicand) or 0 to the result depending on the corresponding bit in the multiplier.

We also shift the result by one position at every step.
Binary Multiplication

- Multiplication is simply repeated addition

<table>
<thead>
<tr>
<th>Multiplicand</th>
<th>1101 (13)</th>
<th>Multiplier</th>
<th>1011 (11)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0000</td>
<td></td>
<td>01101</td>
</tr>
<tr>
<td>+</td>
<td>1101</td>
<td>+ 1101</td>
<td>100111</td>
</tr>
<tr>
<td></td>
<td>0000</td>
<td>+ 0000</td>
<td>0100111</td>
</tr>
<tr>
<td></td>
<td>+ 1101</td>
<td>+ 1101</td>
<td>0100111</td>
</tr>
</tbody>
</table>

At each step we add either 1101 (the multiplicand) or 0 to the result depending on the corresponding bit in the multiplier.

We also shift the result by one position at every step.
### Binary Multiplication

- Multiplication is simply repeated addition

<table>
<thead>
<tr>
<th>Multiplicand</th>
<th></th>
<th>1101 (13)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiplier</td>
<td></td>
<td>1011 (11)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>0000</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td></td>
<td>1101</td>
</tr>
<tr>
<td></td>
<td></td>
<td>01101</td>
</tr>
<tr>
<td>+</td>
<td></td>
<td>1101</td>
</tr>
<tr>
<td></td>
<td></td>
<td>100111</td>
</tr>
<tr>
<td>+</td>
<td></td>
<td>0000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0100111</td>
</tr>
<tr>
<td>+</td>
<td></td>
<td>1101</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10001111</td>
</tr>
</tbody>
</table>

At each step we add either 1101 (the multiplicand) or 0 to the result depending on the corresponding bit in the multiplier.

We also shift the result by one position at every step.

= 10001111 (143)
Design Tradeoffs in Arithmetic Circuits
Algorithmic Tradeoffs in Hardware Design

- Each function allows many implementations with widely different delay, area, and power tradeoffs
Algorithmic Tradeoffs in Hardware Design

- Each function allows many implementations with widely different delay, area, and power tradeoffs

Problem

Hardware designer

High-level circuit description

Synthesis tool

Optimized circuit implementation
Algorithmic Tradeoffs in Hardware Design

- Each function allows many implementations with widely different delay, area, and power tradeoffs.

- Choosing the right **algorithms** is key to optimizing your design.
  - Tools cannot compensate for an inefficient algorithm (in most cases).

Problem

Hardware designer

High-level circuit description

Synthesis tool

Optimized circuit implementation
Algorithmic Tradeoffs in Hardware Design

- Each function allows many implementations with widely different delay, area, and power tradeoffs

- Choosing the right algorithms is key to optimizing your design
  - Tools cannot compensate for an inefficient algorithm (in most cases)
  - Just like programming software
Algorithmic Tradeoffs in Hardware Design

- Each function allows many implementations with widely different delay, area, and power tradeoffs

- Choosing the right **algorithms** is key to optimizing your design
  - Tools cannot compensate for an inefficient algorithm (in most cases)
  - Just like programming software

- Case study: Building a better adder
Ripple-Carry Adder: Simple but Slow

c_{out} \quad a_{n-1} \quad b_{n-1} \quad A \quad B \quad CO \quad FA \quad CI \quad S_{n-1}

A \quad B \quad CO \quad FA \quad CI \quad S_{n-2}

... \quad a_2 \quad b_2 \quad A \quad B \quad CO \quad FA \quad CI \quad S_2

A \quad B \quad CO \quad FA \quad CI \quad S_1

A \quad B \quad CO \quad FA \quad CI \quad S_0

c_{in}
Ripple-Carry Adder: Simple but Slow

- Worst-case path: Carry propagation from LSB to MSB, e.g., when adding 11...111 to 00...001
Ripple-Carry Adder: Simple but Slow

- Worst-case path: Carry propagation from LSB to MSB, e.g., when adding 11...111 to 00...001
Ripple-Carry Adder: Simple but Slow

- Worst-case path: Carry propagation from LSB to MSB, e.g., when adding 11...111 to 00...001

\[ t_{PD} = (n-1)*t_{PD, CI \rightarrow CO} + t_{PD, CI \rightarrow S} \]
Ripple-Carry Adder: Simple but Slow

- Worst-case path: Carry propagation from LSB to MSB, e.g., when adding 11...111 to 00...001

\[ t_{PD} = (n-1) \times t_{PD, CI \rightarrow CO} + t_{PD, CI \rightarrow S} \approx \Theta(n) \]
Ripple-Carry Adder: Simple but Slow

- Worst-case path: Carry propagation from LSB to MSB, e.g., when adding 11...111 to 00...001

\[ t_{PD} = (n-1) \times t_{PD,CI \rightarrow CO} + t_{PD,CI \rightarrow S} \approx \Theta(n) \]

- \( \Theta(n) \) is read “order n” and tells us that the latency of our adder grows linearly with the number of bits of the operands.
Asymptotic Analysis

- Formally, $g(n) = \Theta(f(n))$ iff there exist $C_2 \geq C_1 > 0$ such that for all but finitely many integers $n \geq 0$,

$$C_2 \cdot f(n) \geq g(n) \geq C_1 \cdot f(n)$$
Asymptotic Analysis

- Formally, \( g(n) = \Theta(f(n)) \) iff there exist \( C_2 \geq C_1 > 0 \) such that for all but finitely many integers \( n \geq 0 \),

\[
C_2 \cdot f(n) \geq g(n) \geq C_1 \cdot f(n)
\]

\( g(n) = O(f(n)) \) \( \Theta(...) \) implies both inequalities; 
\( O(...) \) implies only the first.
Asymptotic Analysis

- Formally, \( g(n) = \Theta(f(n)) \) iff there exist \( C_2 \geq C_1 > 0 \) such that for all but finitely many integers \( n \geq 0 \),

\[
C_2 \cdot f(n) \geq g(n) \geq C_1 \cdot f(n)
\]

- Example: \( n^2 + 2n + 3 = \Theta(n^2) \) (read “is of order \( n^2 \)”)
Asymptotic Analysis

- Formally, \( g(n) = \Theta(f(n)) \) iff there exist \( C_2 \geq C_1 > 0 \) such that for all but finitely many integers \( n \geq 0 \),

\[
C_2 \cdot f(n) \geq g(n) \geq C_1 \cdot f(n)
\]

- Example: \( n^2 + 2n + 3 = \Theta(n^2) \) (read “is of order \( n^2 \)”)
  since \( 2n^2 > n^2 + 2n + 3 > n^2 \) except for a few small integers.
Carry-Select Adder Trades Area for Speed

a[31:16]  b[31:16]  \downarrow  \downarrow
16-bit Adder

\downarrow  \downarrow
16-bit Adder

\downarrow  \downarrow
16-bit Adder

s[31:16]

a[15:0]  b[15:0]  \downarrow  \downarrow
16-bit Adder

\downarrow  \downarrow
16-bit Adder

\downarrow  \downarrow
16-bit Adder

s[15:0]
Two copies of the high half of the adder: one assumes carry-in of “0”, the other carry-in of “1”
Two copies of the high half of the adder: one assumes carry-in of “0”, the other carry-in of “1”.

The carry-out of the low half selects the correct version of the high-half addition.
Carry-Select Adder Trades Area for Speed

Two copies of the high half of the adder: one assumes carry-in of “0”, the other carry-in of “1”

- Propagation delay: $t_{PD,32} = t_{PD,16} + t_{PD,MUX}$

The carry-out of the low half selects the correct version of the high-half addition.
Carry-Select Adder Trades Area for Speed

Two copies of the high half of the adder: one assumes carry-in of “0”, the other carry-in of “1”.

- **Propagation delay:** \( t_{PD,32} = t_{PD,16} + t_{PD,MUX} \)
  - If we used 16-bit ripple-carry adders, this would roughly halve delay over a 32-bit ripple-carry adder
Carry-Select Adder Trades Area for Speed

- **Propagation delay:** \( t_{PD,32} = t_{PD,16} + t_{PD,MUX} \)
  - If we used 16-bit ripple-carry adders, this would roughly halve delay over a 32-bit ripple-carry adder
  - If we apply the same strategy recursively (build each 16-bit adder from 8-bit carry-select adders, etc.), \( t_{PD,n} = \Theta(\log n) \)

Two copies of the high half of the adder: one assumes carry-in of “0”, the other carry-in of “1”

The carry-out of the low half selects the correct version of the high-half addition.
Carry-Select Adder Trades Area for Speed

- Propagation delay: $t_{PD,32} = t_{PD,16} + t_{PD,MUX}$
  - If we used 16-bit ripple-carry adders, this would roughly halve delay over a 32-bit ripple-carry adder
  - If we apply the same strategy recursively (build each 16-bit adder from 8-bit carry-select adders, etc.), $t_{PD,n} = \Theta(\log n)$

**Drawbacks?**
Carry-Select Adder Trades Area for Speed

- Propagation delay: $t_{PD,32} = t_{PD,16} + t_{PD,MUX}$
  - If we used 16-bit ripple-carry adders, this would roughly halve delay over a 32-bit ripple-carry adder
  - If we apply the same strategy recursively (build each 16-bit adder from 8-bit carry-select adders, etc.), $t_{PD,n}=\Theta(\log n)$

Drawbacks? Consumes much more area than ripple-carry adder
Carry-Select Adder Trades Area for Speed

Two copies of the high half of the adder: one assumes carry-in of “0”, the other carry-in of “1”

- **Propagation delay:** \( t_{PD,32} = t_{PD,16} + t_{PD,MUX} \)
  - If we used 16-bit ripple-carry adders, this would roughly halve delay over a 32-bit ripple-carry adder
  - If we apply the same strategy recursively (build each 16-bit adder from 8-bit carry-select adders, etc.), \( t_{PD,n} = \Theta(\log n) \)

**Drawbacks?** Consumes much more area than ripple-carry adder
Wide mux adds significant delay (lab 2)
Carry-Lookahead Adders (CLAs)

- CLAs compute all carry bits in $\Theta(\log n)$ delay
Carry-Lookahead Adders (CLAs)

- CLAs compute all carry bits in $\Theta(\log n)$ delay
Carry-Lookahead Adders (CLAs)

- CLAs compute all carry bits in $\Theta(\log n)$ delay

```
<table>
<thead>
<tr>
<th>an-1</th>
<th>bn-1</th>
<th>an-2</th>
<th>bn-2</th>
<th>a2</th>
<th>b2</th>
<th>a1</th>
<th>b1</th>
<th>a0</th>
<th>b0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Carry Generation Logic

- $c_{out}$
- $s_{n-1}$
- $c_{n-1}$
- $a_{n-1}$
- $b_{n-1}$
- $c_{n-2}$
- $a_{n-2}$
- $b_{n-2}$
- $c_{n-3}$
- $a_2$
- $b_2$
- $c_1$
- $a_1$
- $b_1$
- $c_0$
- $a_0$
- $b_0$

`
Carry-Lookahead Adders (CLAs)

- CLAs compute all carry bits in $\Theta(\log n)$ delay

- Key idea: Transform chain of carry computations into a tree
Carry-Lookahead Adders (CLAs)

- CLAs compute all carry bits in $\Theta(\log n)$ delay

- Key idea: Transform chain of carry computations into a tree
  - Transforming a chain of associative operations (e.g., AND, OR, XOR) into a tree is easy
  - But how to do this with carries?
We can rewrite $c_{out} = ab + (a+b)c_{in}$.
We can rewrite \( c_{out} = ab + (a+b)c_{in} \)
as \( c_{out} = g + pc_{in} \)
with \( g = ab \) (generate)
and \( p = a+b \) (propagate)
Carry Generation and Propagation

- We can rewrite $c_{out} = ab + (a+b)c_{in}$ as $c_{out} = g + pc_{in}$ with $g = ab$ (generate) and $p = a+b$ (propagate)
  - $g=1$ $\rightarrow$ $c_{out} = 1$ (FA generates a carry)
  - $p=1$ (and $g=0$) $\rightarrow$ $c_{out} = c_{in}$ (FA propagates carry)
Carry Generation and Propagation

- We can rewrite \( c_{\text{out}} = ab + (a+b)c_{\text{in}} \) as \( c_{\text{out}} = g + pc_{\text{in}} \) with \( g = ab \) (generate) and \( p = a+b \) (propagate)
  - \( g=1 \) \( \rightarrow \) \( c_{\text{out}} = 1 \) (FA generates a carry)
  - \( p=1 \) (and \( g=0 \)) \( \rightarrow \) \( c_{\text{out}} = c_{\text{in}} \) (FA propagates carry)

- Generate and propagate compose hierarchically
Carry Generation and Propagation

- We can rewrite $c_{out} = ab + (a+b)c_{in}$ as $c_{out} = g + pc_{in}$ with $g = ab$ (generate) and $p = a+b$ (propagate).
  - $g=1$ $\rightarrow$ $c_{out} = 1$ (FA generates a carry)
  - $p=1$ (and $g=0$) $\rightarrow$ $c_{out} = c_{in}$ (FA propagates carry)

- Generate and propagate compose hierarchically
Carry Generation and Propagation

- We can rewrite $c_{\text{out}} = ab + (a+b)c_{\text{in}}$ as $c_{\text{out}} = g + pc_{\text{in}}$
  with $g = ab$ (generate)
  and $p = a+b$ (propagate)

- $g=1 \Rightarrow c_{\text{out}} = 1$ (FA generates a carry)
- $p=1$ (and $g=0) \Rightarrow c_{\text{out}} = c_{\text{in}}$ (FA propagates carry)

- Generate and propagate compose hierarchically

Can you derive $c_{1} = g_{10} + p_{10}c_{\text{in}}$ where $g_{10}, p_{10}$ use the $g$ and $p$ signals of each FA?
Carry Generation and Propagation

- We can rewrite \( c_{out} = ab + (a+b)c_{in} \) as \( c_{out} = g + pc_{in} \) with \( g = ab \) (generate) and \( p = a+b \) (propagate).
  - \( g=1 \) \( \Rightarrow \) \( c_{out} = 1 \) (FA generates a carry)
  - \( p=1 \) (and \( g=0 \)) \( \Rightarrow \) \( c_{out} = c_{in} \) (FA propagates carry)

- Generate and propagate compose hierarchically

Can you derive \( c_1 = g_{10} + p_{10}c_{in} \) where \( g_{10}, p_{10} \) use the \( g \) and \( p \) signals of each FA?

\[
c_1 = g_1 + p_1c_0
\]
Carry Generation and Propagation

- We can rewrite $c_{out} = ab + (a+b)c_{in}$ as $c_{out} = g + pc_{in}$
  with $g = ab$ (generate)
  and $p = a+b$ (propagate)
  - $g=1 \rightarrow c_{out} = 1$ (FA generates a carry)
  - $p=1$ (and $g=0) \rightarrow c_{out} = c_{in}$ (FA propagates carry)

- Generate and propagate compose hierarchically

Can you derive $c_1 = g_{10} + p_{10}c_{in}$ where $g_{10}, p_{10}$ use the $g$ and $p$ signals of each FA?

$c_1 = g_1 + p_1c_0 = g_1 + p_1(g_0 + p_0c_{in})$
Carry Generation and Propagation

- We can rewrite $c_{out} = ab + (a+b)c_{in}$ as $c_{out} = g + pc_{in}$ with $g = ab$ (generate) and $p = a+b$ (propagate).
  - $g=1 \rightarrow c_{out} = 1$ (FA generates a carry)
  - $p=1$ (and $g=0) \rightarrow c_{out} = c_{in}$ (FA propagates carry)

- Generate and propagate compose hierarchically

Can you derive $c_1 = g_{10} + p_{10}c_{in}$ where $g_{10}, p_{10}$ use the $g$ and $p$ signals of each FA?

$$c_1 = g_1 + p_1c_0 = g_1 + p_1(g_0 + p_0c_{in}) = g_1 + p_1g_0 + p_1p_0c_{in}$$
Carry Generation and Propagation

- We can rewrite $c_{\text{out}} = ab + (a+b)c_{\text{in}}$ as $c_{\text{out}} = g + pc_{\text{in}}$
  with $g = ab$ (generate)
  and $p = a+b$ (propagate)
- $g=1$ \rightarrow $c_{\text{out}} = 1$ (FA generates a carry)
- $p=1$ (and $g=0$) \rightarrow $c_{\text{out}} = c_{\text{in}}$ (FA propagates carry)

- Generate and propagate compose hierarchically

Can you derive $c_1 = g_{10} + p_{10}c_{\text{in}}$
where $g_{10}, p_{10}$ use the $g$ and $p$
signals of each FA?

$c_1 = g_1 + p_1c_0 = g_1 + p_1(g_0 + p_0c_{\text{in}}) = g_1 + p_1g_0 + p_1p_0c_{\text{in}}$
Carry Generation and Propagation

- We can rewrite \( c_{out} = ab + (a+b)c_{in} \) as \( c_{out} = g + pc_{in} \) with \( g = ab \) (generate) and \( p = a+b \) (propagate)
  - \( g=1 \) \( \rightarrow \) \( c_{out} = 1 \) (FA generates a carry)
  - \( p=1 \) (and \( g=0 \)) \( \rightarrow \) \( c_{out} = c_{in} \) (FA propagates carry)

- Generate and propagate compose hierarchically

Can you derive \( c_1 = g_{10} + p_{10}c_{in} \) where \( g_{10}, p_{10} \) use the \( g \) and \( p \) signals of each FA?

\[
c_1 = g_1 + p_1c_0 = g_1 + p_1(g_0 + p_0c_{in}) = g_1 + p_1g_0 + p_1p_0c_{in}
\]
CLA Building Blocks

- Step 1: Generate individual g & p signals

\[
g = ab \\
p = a + b
\]

\[gp = \{g, p\}\]
CLA Building Blocks

- **Step 1:** Generate individual g & p signals
  
  \[ g = ab \]
  \[ p = a+b \]

  \[ gp = \{ g, p \} \]

- **Step 2:** Combine adjacent g & p signals
  
  \[ g_{ik} = g_{ij} + p_{ij}g_{(j-1)k} \]
  \[ p_{ik} = p_{ij}p_{(j-1)k} \quad (i \geq j > k) \]
CLA Building Blocks

- **Step 1: Generate individual g & p signals**
  
  \[ \text{gp} = \{g, p\} \quad \text{where} \quad g = ab, \quad p = a + b \]

- **Step 2: Combine adjacent g & p signals**
  
  \[
  \begin{align*}
  \text{gp}_{ij} & \quad \text{gp}_{(j-1)k} \\
  g_{ik} & = g_{ij} + p_{ij}g_{(j-1)k} \\
  p_{ik} & = p_{ij}p_{(j-1)k} \quad \text{(i ≥ j > k)}
  \end{align*}
  \]

- **Step 3: Generate individual carries**

  \[
  \begin{align*}
  \text{gp}_{ij} & \quad \text{c}_{j-1} \\
  c_i & = g_{ij} + p_{ij}c_{j-1}
  \end{align*}
  \]
CLA Building Blocks

- **Step 1:** Generate individual \( g \) & \( p \) signals
  
  \[
  a \quad b
  \]
  
  \[
  g = ab \\
  p = a+b
  \]
  
  \( gp = \{g, p\} \)

- **Step 2:** Combine adjacent \( g \) & \( p \) signals
  
  \[
  \begin{align*}
  gp_{ij} & \quad gp_{(j-1)k} \\
  g_{ik} & = g_{ij} + p_{ij}g_{(j-1)k} \\
  p_{ik} & = p_{ij}p_{(j-1)k} & (i \geq j > k)
  \end{align*}
  \]
  
  \[
  \text{GP}
  \]

- **Step 3:** Generate individual carries
  
  \[
  \begin{align*}
  gp_{ij} & \quad c_{j-1} \\
  c_{i} & = g_{ij} + p_{ij}c_{j-1}
  \end{align*}
  \]
  
  \[
  \text{C}
  \]

There are many CLA variants. Let’s derive the Brent-Kung CLA.
Generating and Combining gp’s
Generating and Combining gp’s

Diagram showing a network of GP operations with gp nodes and variables a and b.
Generating and Combining gp’s

\[
\begin{align*}
  \text{GP} & \quad \text{GP} & \quad \text{GP} & \quad \text{GP} \\
  a_7 & \quad b_7 & \quad a_6 & \quad b_6 & \quad a_5 & \quad b_5 & \quad a_4 & \quad b_4 & \quad a_3 & \quad b_3 & \quad a_2 & \quad b_2 & \quad a_1 & \quad b_1 & \quad a_0 & \quad b_0 \\
  \text{gp}_7 & \quad \text{gp}_6 & \quad \text{gp}_5 & \quad \text{gp}_4 & \quad \text{gp}_3 & \quad \text{gp}_2 & \quad \text{gp}_1 & \quad \text{gp}_0 \\
  \text{gp}_{76} & \quad \text{gp}_{54} & \quad \text{gp}_{32} & \quad \text{gp}_{30} \\
  \text{gp}_{74} & \quad \text{gp}_{54} & \quad \text{gp}_{32} & \quad \text{gp}_{30} \\
\end{align*}
\]
Generating and Combining gp’s

```
<table>
<thead>
<tr>
<th>a7  b7</th>
<th>a6  b6</th>
<th>a5  b5</th>
<th>a4  b4</th>
<th>a3  b3</th>
<th>a2  b2</th>
<th>a1  b1</th>
<th>a0  b0</th>
</tr>
</thead>
<tbody>
<tr>
<td>gp7</td>
<td>gp6</td>
<td>gp5</td>
<td>gp4</td>
<td>gp3</td>
<td>gp2</td>
<td>gp1</td>
<td>gp0</td>
</tr>
<tr>
<td>GP</td>
<td>GP</td>
<td>GP</td>
<td>GP</td>
<td>GP</td>
<td>GP</td>
<td>GP</td>
<td>GP</td>
</tr>
<tr>
<td>gp76</td>
<td></td>
<td>gp54</td>
<td></td>
<td>gp32</td>
<td></td>
<td>gp10</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>gp30</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>gp70</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Generating and Combining gp’s

How does delay grow with number of bits?
Generating and Combining gp’s

How does delay grow with number of bits? $\Theta(\log n)$
Generating and Combining gp’s

How does delay grow with number of bits? \( \Theta(\log n) \)
Generating the Carries
Generating the Carries
Generating the Carries

September 18, 2018

MIT 6.004 Fall 2018
Generating the Carries
Generating the Carries

\[ a_7 b_7 \quad a_6 b_6 \quad a_5 b_5 \quad a_4 b_4 \quad a_3 b_3 \quad a_2 b_2 \quad a_1 b_1 \quad a_0 b_0 \]

\[ \text{gp}_7 \quad \text{gp}_6 \quad \text{gp}_5 \quad \text{gp}_4 \quad \text{gp}_3 \quad \text{gp}_2 \quad \text{gp}_1 \quad \text{gp}_0 \]

\[ \text{GP} \quad \text{GP} \quad \text{GP} \quad \text{GP} \quad \text{GP} \quad \text{GP} \quad \text{GP} \quad \text{GP} \]

\[ \text{gp}_{76} \quad \text{gp}_{54} \quad \text{gp}_{32} \quad \text{gp}_{30} \quad \text{gp}_{10} \]

\[ \text{gp}_{74} \quad \text{gp}_{70} \]

\[ c_{in} \]

\[ c_7 \quad c_6 \quad c_5 \quad c_4 \quad c_3 \quad c_2 \quad c_1 \quad c_0 \]
Carry-Lookahead Adder Takeaways

- There are many CLA designs
  - We’ve seen a Brent-Kung CLA
  - Several other types (e.g., Kogge-Stone)
There are many CLA designs
- We’ve seen a Brent-Kung CLA
- Several other types (e.g., Kogge-Stone)
- Different variants for each type, e.g., using higher-radix trees to reduce depth
There are many CLA designs
- We’ve seen a Brent-Kung CLA
- Several other types (e.g., Kogge-Stone)
- Different variants for each type, e.g., using higher-radix trees to reduce depth

This technique is useful beyond adders: computes any one-dimensional binary recurrence in $\Theta(\log n)$ delay
- e.g., comparators, priority encoders, etc.
Summary

- We can encode unsigned integers using strings of bits. Addition and subtraction done as in decimal.
- Two’s complement encodes negative integers while preserving the simplicity of unsigned arithmetic.
- Binary multiplication is simply repeated addition.
Summary

- We can encode unsigned integers using strings of bits. Addition and subtraction done as in decimal.
- Two’s complement encodes negative integers while preserving the simplicity of unsigned arithmetic.
- Binary multiplication is simply repeated addition.
- Choosing the right algorithms is crucial to design good digital circuits—tools can only do so much!
- Carry-lookahead adders perform $\Theta(\log n)$ addition with modest area cost. This technique can be used to optimize a broad class of circuits.
Thank you!

Next lecture: Implementing Complex Combinational Circuits in Bluespec