COMP 122/L Lecture 2

Kyle Dewey

Outline

- Operations on binary values
  - AND, OR, XOR, NOT
  - Bit shifting (left, two forms of right)
  - Addition
  - Subtraction
- Twos complement

Bitwise Operations

Bitwise AND

- Similar to logical AND (`&&`), except it works on a bit-by-bit manner
- Denoted by a single ampersand: `&`

```
(1001 &
 0101)=
 0001
```

Bitwise OR

- Similar to logical OR (`||`), except it works on a bit-by-bit manner
- Denoted by a single pipe character: `|`

```
(1001 |
 0101)=
 1101
```

Bitwise XOR

- Exclusive OR, denoted by a carat: `^`
- Similar to bitwise OR, except that if both inputs are `1` then the result is `0`

```
(1001 ^
 0101)=
 1100
```

Bitwise NOT

- Similar to logical NOT (`!`), except it works on a bit-by-bit manner
- Denoted by a tilde character: `~`

```
~1001 =
 0110
```

Shift Left

- Move all the bits `N` positions to the left, subbing in `N` `0`s on the right

Shift Left

- Move all the bits `N` positions to the left, subbing in `N` `0`s on the right

```
1001
```

Shift Left

- Move all the bits `N` positions to the left, subbing in `N` `0`s on the right

```
1001 << 2 =
100100
```

Shift Left

- Useful as a restricted form of multiplication
- Question: how?

```
1001 << 2 =
100100
```

Shift Left as Multiplication

- Equivalent decimal operation:

```
234
```

Shift Left as Multiplication

- Equivalent decimal operation:

```
234 << 1 =
2340
```

Shift Left as Multiplication

- Equivalent decimal operation:

```
234 << 1 =
2340

234 << 2 =
23400
```

Multiplication

- Shifting left `N` positions multiplies by `(base)`^{N}
- Multiplying by 2 or 4 is often necessary (shift left 1 or 2 positions, respectively)
- Often a whooole lot faster than telling the processor to multiply
- Compilers try hard to do this

```
234 << 2 =
23400
```

Shift Right

- Move all the bits `N` positions to the right, subbing in **either** `N` `0`s or `N` `1`s on the left
  - Two different forms

Shift Right

- Move all the bits `N` positions to the right, subbing in **either** `N` `0`s or `N` (whatever the leftmost bit is)s on the left
  - Two different forms

```
1001 >> 2 =
either 0010 or 1110
```

Shift Right Trick

- Question: If shifting left multiplies, what does shift right do?

Shift Right Trick

- Question: If shifting left multiplies, what does shift right do?
    - Answer: divides in a similar way, but truncates result

Shift Right Trick

- Question: If shifting left multiplies, what does shift right do?
  - Answer: divides in a similar way, but truncates result

```
234
```

Shift Right Trick

- Question: If shifting left multiplies, what does shift right do?
  - Answer: divides in a similar way, but truncates result

```
234 >> 1 =
23
```

Two Forms of Shift Right

- Subbing in `0`s makes sense
- What about subbing in the leftmost bit?
    - And why is this called “arithmetic” shift right?

```
1100 (arithmetic)>> 1 =
1110
```

Answer...Sort of

- Arithmetic form is intended for numbers in *twos complement*, whereas the non-arithmetic form is intended for *unsigned* numbers

Twos Complement

Problem

- Binary representation so far makes it easy to represent positive numbers and zero
- Question: What about representing negative numbers?

Twos Complement

- Way to represent positive integers, negative integers, and zero
- If `1` is in the *most significant bit* (generally leftmost bit in this class), then it is negative

Decimal to Twos Complement

- Example: -5 decimal to binary (twos complement)

Decimal to Twos Complement

- Example: -5 decimal to binary (twos complement)
- First, convert the magnitude to an unsigned representation

Decimal to Twos Complement

- Example: -5 decimal to binary (twos complement)
- First, convert the magnitude to an unsigned representation

`5` (decimal) = `0101` (binary)

Decimal to Twos Complement

- Then, take the bits, and negate them

Decimal to Twos Complement

- Then, take the bits, and negate them

```
0101
```

Decimal to Twos Complement

- Then, take the bits, and negate them

```
~0101 =
 1010
```

Decimal to Twos Complement

- Finally, add one:

Decimal to Twos Complement

- Finally, add one:

```
1010
```

Decimal to Twos Complement

- Finally, add one:

```
1010 + 1 =
1011
```

Twos Complement to Decimal

- Same operation: negate the bits, and add one

Twos Complement to Decimal

- Same operation: negate the bits, and add one

```
1011
```

Twos Complement to Decimal

- Same operation: negate the bits, and add one

```
~1011 =
 0100
```

Twos Complement to Decimal

- Same operation: negate the bits, and add one

```
0100
```

Twos Complement to Decimal

- Same operation: negate the bits, and add one

```
0100 + 1 =
0101
```

Twos Complement to Decimal

- Same operation: negate the bits, and add one

```
0100 + 1 =
0101 =
-5
```

We started with `1011` - negative

Intuition

- Modular arithmetic, with the convention that a leading `1` bit means negative

$\longrightarrow$

Denotes
+1

$$\begin{tikzcd}
 & 000 \arrow[dr] & \\
111 \arrow[ur] & & 001 \arrow[d] \\
110 \arrow[u] & & 010 \arrow[d] \\
101 \arrow[u] & & 011 \arrow[dl] \\
 & 100 \arrow[ul] &
\end{tikzcd}$$

Intuition

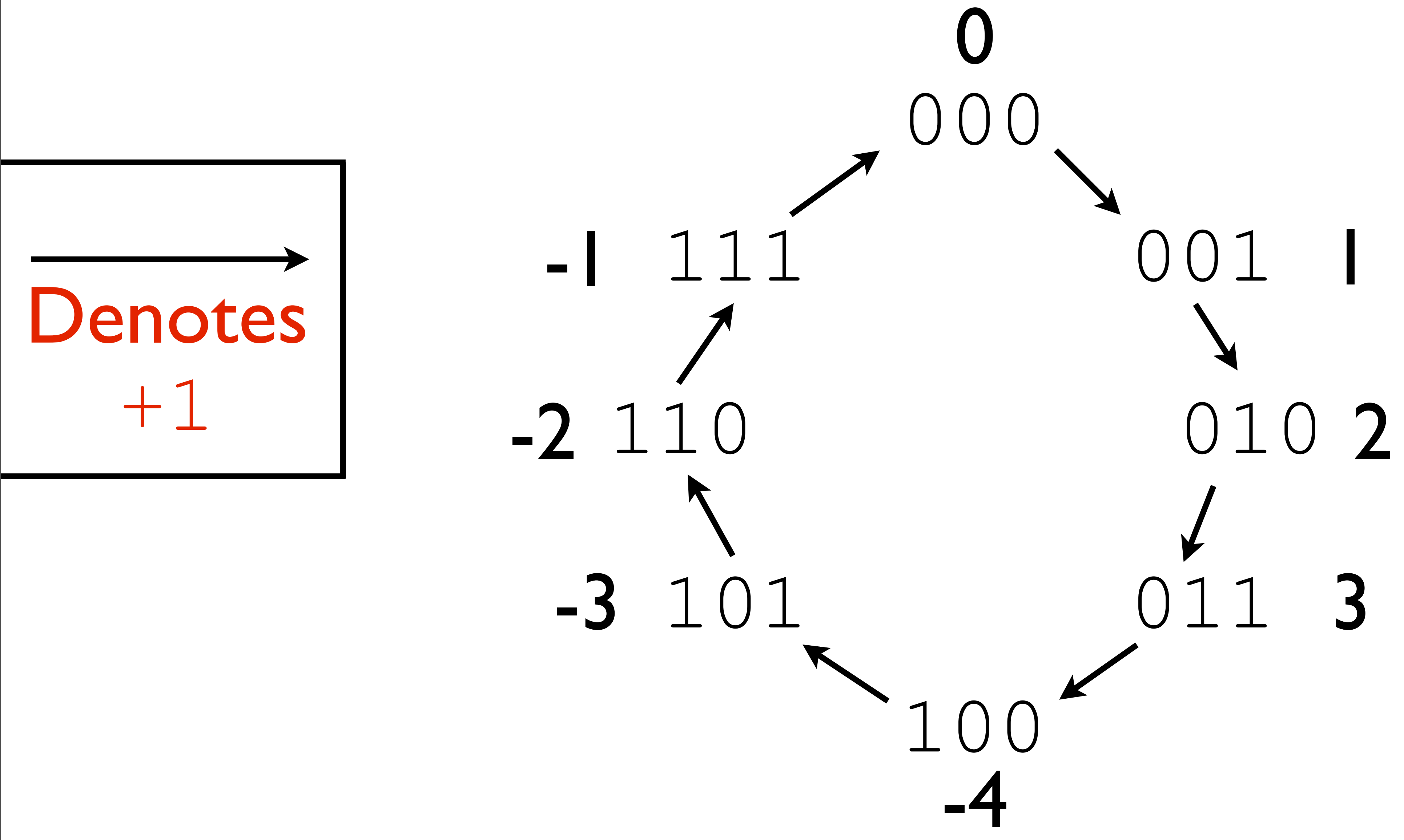
- Modular arithmetic, with the convention that a leading `1` bit means negative

$\longrightarrow$
Denotes
+1

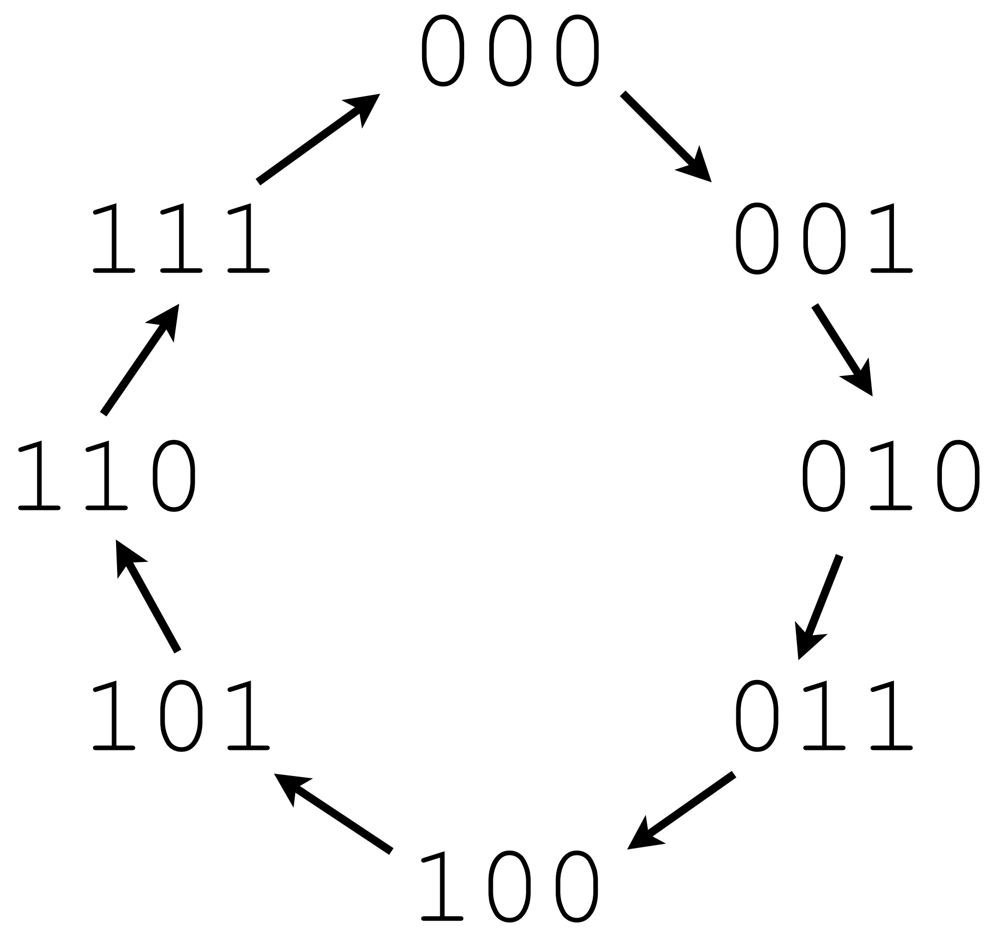
$$\begin{tikzcd}
 & \overset{\text{(zero)}}{000} \arrow[dr] & \\
\text{(least -) } 111 \arrow[ur] & & 001 \text{ (least +)} \arrow[d] \\
110 \arrow[u] & & 010 \arrow[d] \\
101 \arrow[u] & & 011 \text{ (most +)} \arrow[dl] \\
 & 100 \text{ (most -)} \arrow[ul] &
\end{tikzcd}$$

Intuition

- Modular arithmetic, with the convention that a leading `1` bit means negative



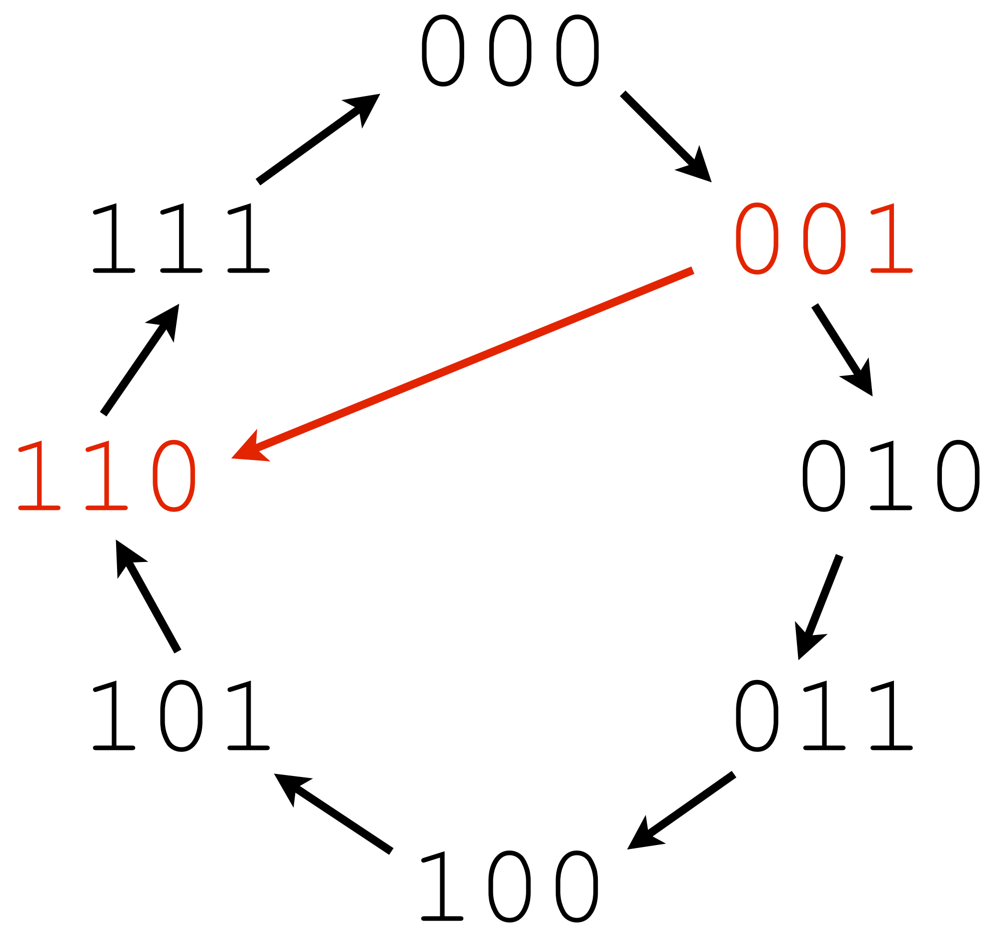
Negation of `1`



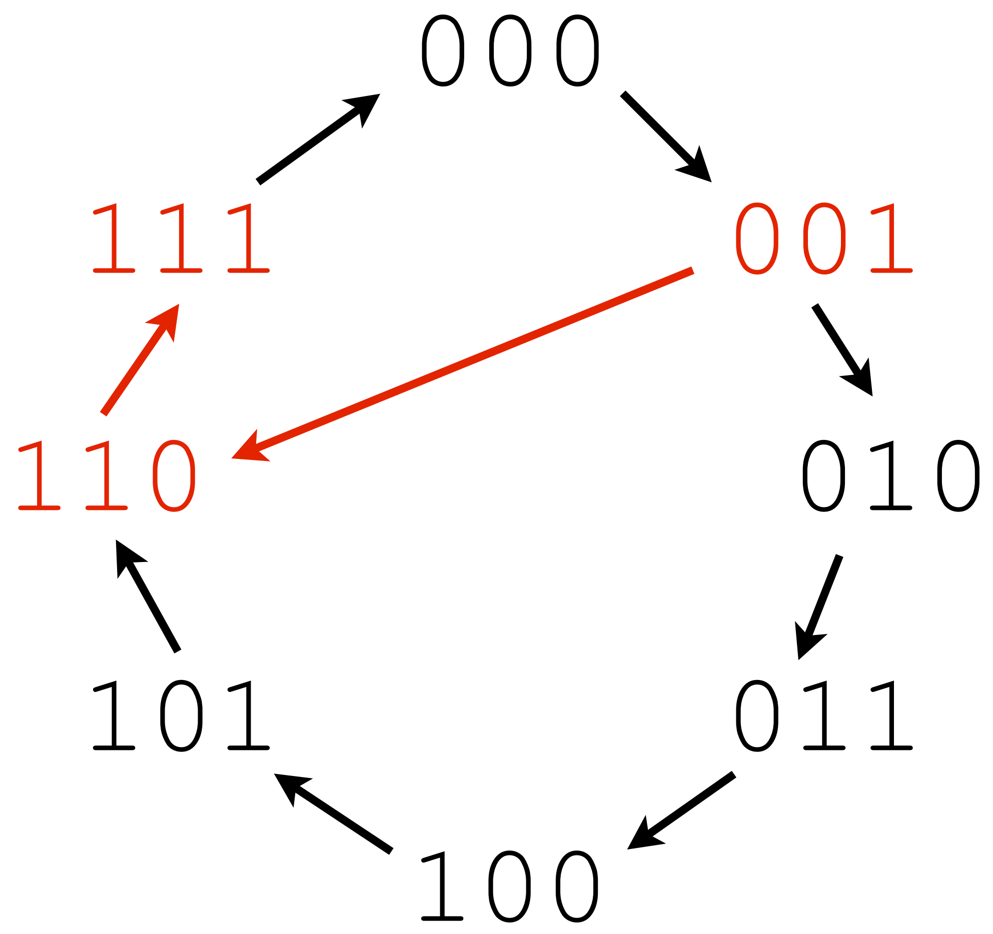
Negation of `1`

$$\begin{tikzcd}
 & & 000 \arrow[dr] & & \\
 & 111 \arrow[ur] & & 001 \arrow[dr] & \\
110 \arrow[ur] & & & & 010 \arrow[dl] \\
 & 101 \arrow[ul] & & 011 \arrow[dl] & \\
 & & 100 \arrow[ul] & &
\end{tikzcd}$$

Negation of `1`



Negation of `1`



Consequences

- What is the negation of `000`?

$$\begin{tikzcd}
 & 000 \arrow[dr] & \\
111 \arrow[ur] & & 001 \arrow[d] \\
110 \arrow[u] & & 010 \arrow[d] \\
101 \arrow[u] & & 011 \arrow[dl] \\
 & 100 \arrow[ul] &
\end{tikzcd}$$

Consequences

- What is the negation of `100`?

$$\begin{tikzcd}
 & & 000 \arrow[dr] & & \\
 & 111 \arrow[ur] & & 001 \arrow[dr] & \\
110 \arrow[ur] & & & & 010 \arrow[dl] \\
 & 101 \arrow[ul] & & 011 \arrow[dl] & \\
 & & 100 \arrow[ul] & &
\end{tikzcd}$$

Arithmetic Shift Right

- **Not exactly** division by a power of two
- Consider -3 / 2

$$\begin{tikzcd}
 & \overset{0}{000} \arrow[dr] & \\
-1\quad 111 \arrow[ur] & & 001\quad 1 \arrow[d] \\
-2\quad 110 \arrow[u] & & 010\quad 2 \arrow[d] \\
-3\quad 101 \arrow[u] & & 011\quad 3 \arrow[dl] \\
 & \underset{-4}{100} \arrow[ul] &
\end{tikzcd}$$

Addition

Building Up Addition

- Question: how might we add the following, in decimal?

```
  986
 +123
 ----
    ?
```

Building Up Addition

- Question: how might we add the following, in decimal?

```
 986
+123
----
   ?
```

			` 6`
			`+3`
			`--`
			` ?`

Building Up Addition

- Question: how might we add the following, in decimal?

```
 986
+123
----
   ?
```

		```
		 8
		+2
		--
		 ?
		```
			```
			 6
			+3
			--
			 9
			```

Building Up Addition

- Question: how might we add the following, in decimal?

```
  986
+123
----
    ?
```

	Carry: 1	8 +2 -- 0	6 +3 -- 9

Building Up Addition

- Question: how might we add the following, in decimal?

```
 986
+123
----
   ?
```

	1	8	6
	9	+2	+3
	+1	--	--
	--	0	9
	?		

Building Up Addition

- Question: how might we add the following, in decimal?

```
  986
+123
----
    ?
```

Carry: 1	1 9 +1 -- 1	8 +2 -- 0	6 +3 -- 9

Building Up Addition

- Question: how might we add the following, in decimal?

```
  986
 +123
 ----
    ?
```

```
 1
+0
--
 1
```
	```
	 1
	 9
	+1
	--
	 1
	```
		```
		 8
		+2
		--
		 0
		```
			```
			 6
			+3
			--
			 9
			```

Core Concepts

- We have a “primitive” notion of adding single digits, along with an idea of *carrying* digits
- We can build on this notion to add numbers together that are more than one digit long

Now in Binary

- Arguably simpler - fewer one-bit possibilities

0	0	1	1
+0	+1	+0	+1
--	--	--	--
?	?	?	?

Now in Binary

- Arguably simpler - fewer one-bit possibilities

```
  0 |   0 |   1 |   1
 +0 |  +1 |  +0 |  +1
 -- |  -- |  -- |  --
  0 |   1 |   1 |   0
```

Carry: 1

Chaining the Carry

- Also need to account for any input carry

```
  0              0                0              0
  0              0                1              1
 +0             +1               +0             +1
 --             --               --             --
  0              1                1              0 Carry: 1

  1              1                1              1
  0              0                1              1
 +0             +1               +0             +1
 --             --               --             --
  1              0    Carry: 1    0   Carry: 1   1 Carry: 1
```

Adding Multiple Bits

- How might we add the numbers below?

```
  011
 +001
------
```

Adding Multiple Bits

- How might we add the numbers below?

```
    0
  011
 +001
------
```

Adding Multiple Bits

- How might we add the numbers below?

```
  10
 011
+001
------
   0
```

Adding Multiple Bits

- How might we add the numbers below?

```
  110
  011
 +001
------
   00
```

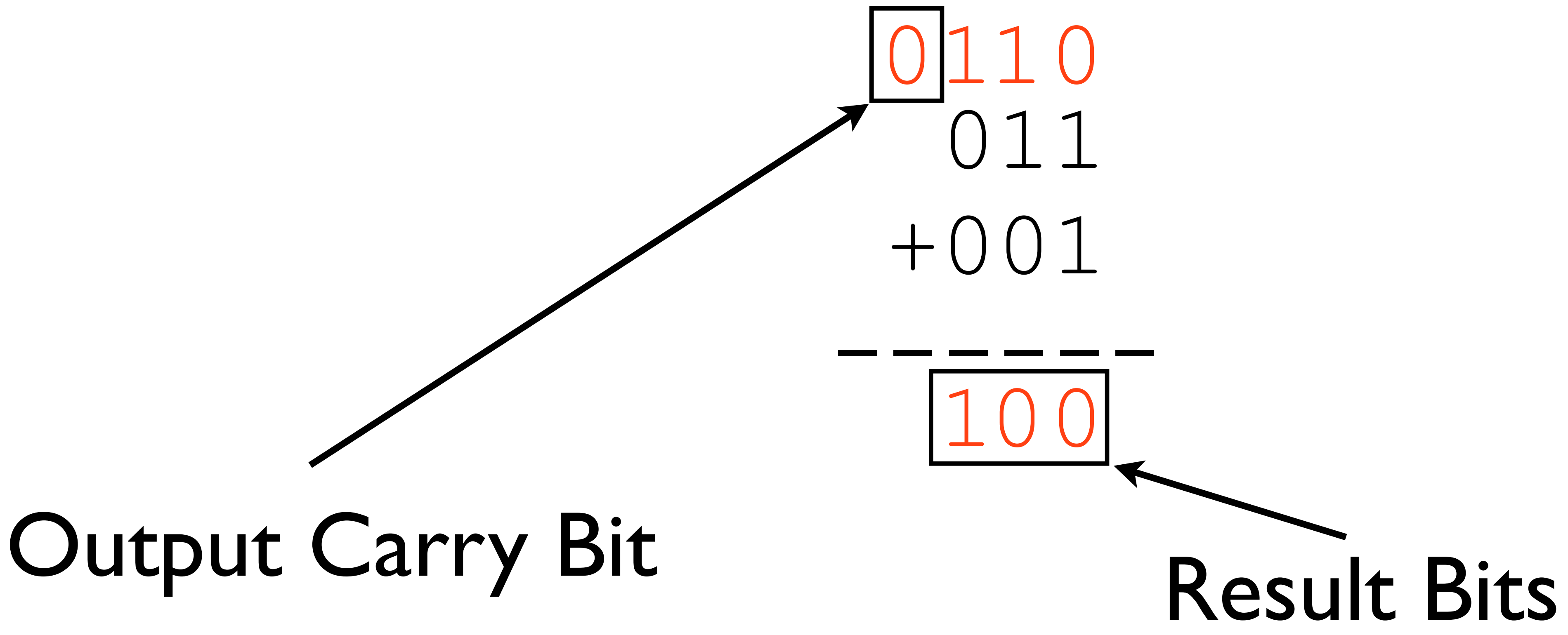
Adding Multiple Bits

- How might we add the numbers below?

```
0110
 011
+001
------
 100
```

Adding Multiple Bits

- How might we add the numbers below?



Another Example

```
  111
 +001
------
```

Another Example

```
    0
  111
 +001
------
```

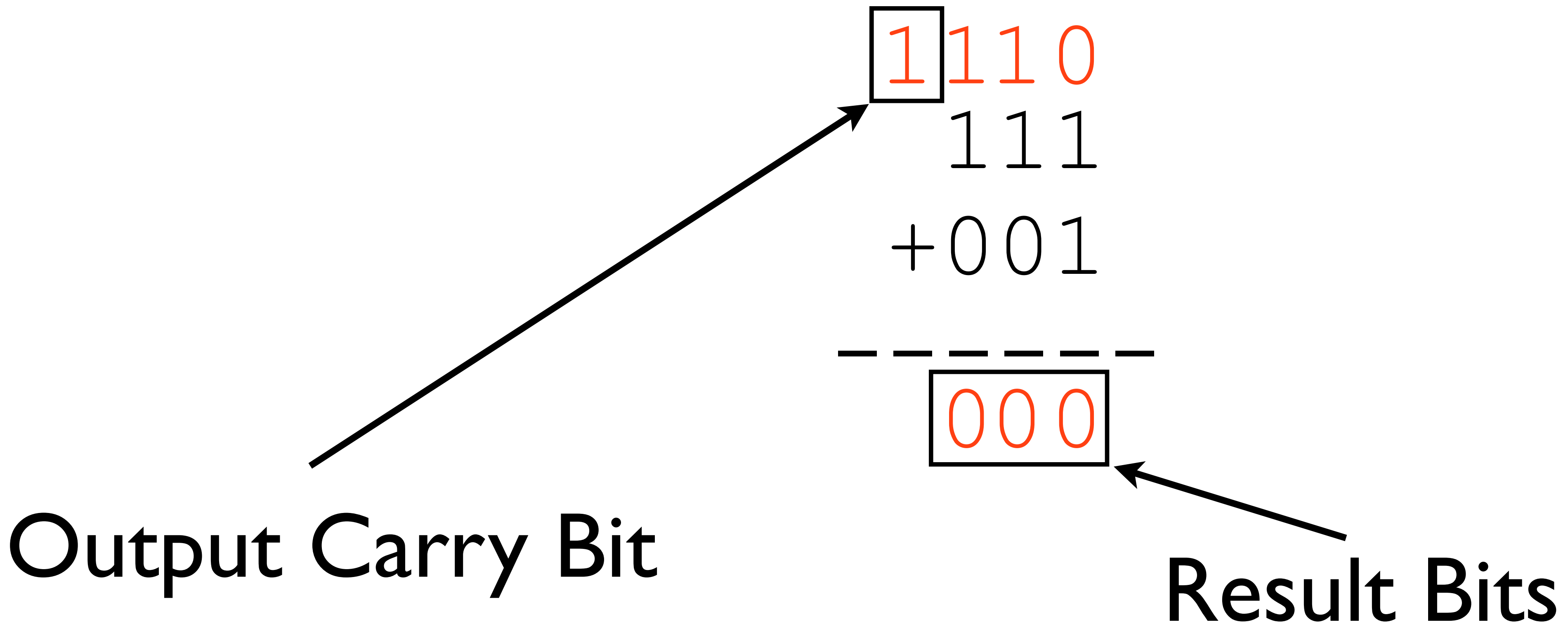
Another Example

```
  10
 111
+001
------
   0
```

Another Example

```
  110
  111
 +001
------
   00
```

Another Example



Output Carry Bit Significance

- For unsigned numbers, it indicates if the result did not fit all the way into the number of bits allotted
- May be an error condition for software

Signed Addition

- Question: what is the result of the following operation?

```
 011
+011
----
   ?
```

Signed Addition

- Question: what is the result of the following operation?

```
 011
+011
----
0110
```

Overflow

- In this situation, *overflow* occurred: this means that both the operands had the same sign, and the result's sign differed

```
 011
+011
----
 110
```

- Possibly a software error

Overflow vs. Carry

- These are **different ideas**
  - Carry is relevant to **unsigned** values
  - Overflow is relevant to **signed** values

111	011	111	001
+001	+011	+100	+001
----	----	----	----
000	110	011	010
No Overflow; Carry	Overflow; No Carry	Overflow; Carry	No Overflow; No Carry

Subtraction

Subtraction

- Have been saying to invert bits and add one to second operand
- Could do it this way in hardware, but there is a trick

```
 001
-001
----
   ?
```

Hint: these two questions are equivalent

```
 001
+111
----
   ?
```

Subtraction Trick

- Assume we can cheaply invert bits, but we want to avoid adding twice (once to add 1 and once to add the other result)
- How can we do this easily?

Subtraction Trick

- Assume we can cheaply invert bits, but we want to avoid adding twice (once to add 1 and once to add the other result)
- How can we do this easily?
  - Set the initial carry to `1` instead of `0`

Subtraction Example

```
 0101
-0011
 ----
```

Subtraction Example

```
 0101
-0011
 ----
```

$$\xrightarrow{\textbf{Invert } \texttt{0011}}$$

Subtraction Example

```
 0101
-0011
 ----
```

$\xrightarrow{\textbf{Invert } \texttt{0011}} \texttt{1100}$

Subtraction Example

```
 0101
-0011
 ----
```

$\xrightarrow{\text{Invert } \texttt{0011}}$ `1100` $\xrightarrow{\text{Equivalent to}}$

Subtraction Example

$$\begin{array}{r} \texttt{0101} \\ \texttt{-0011} \\ \texttt{----} \end{array} \xrightarrow{\text{Invert } \texttt{0011}} \texttt{1100} \xrightarrow{\text{Equivalent to}} \begin{array}{r} \texttt{1} \\ \texttt{0101} \\ \texttt{+1100} \\ \texttt{----} \end{array}$$

Subtraction Example

$$\begin{array}{r} 0101 \\ -0011 \\ \text{----} \end{array} \xrightarrow{\text{Invert } 0011} 1100 \xrightarrow{\text{Equivalent to}} \begin{array}{r} \boxed{1}1011 \\ 0101 \\ +1100 \\ \text{----} \\ 0010 \end{array}$$