COMP 122/L Lecture 3

Kyle Dewey

Outline

- Operations on binary values
  - Addition
  - Subtraction
- Floating point introduction

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

Now in Binary

- Arguably simpler - fewer one-bit possibilities

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

-Need an initial carry-in of zero

Adding Multiple Bits

- How might we add the numbers below?

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

-Need an initial carry-in of zero

Adding Multiple Bits

- How might we add the numbers below?

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

-Need an initial carry-in of zero

Adding Multiple Bits

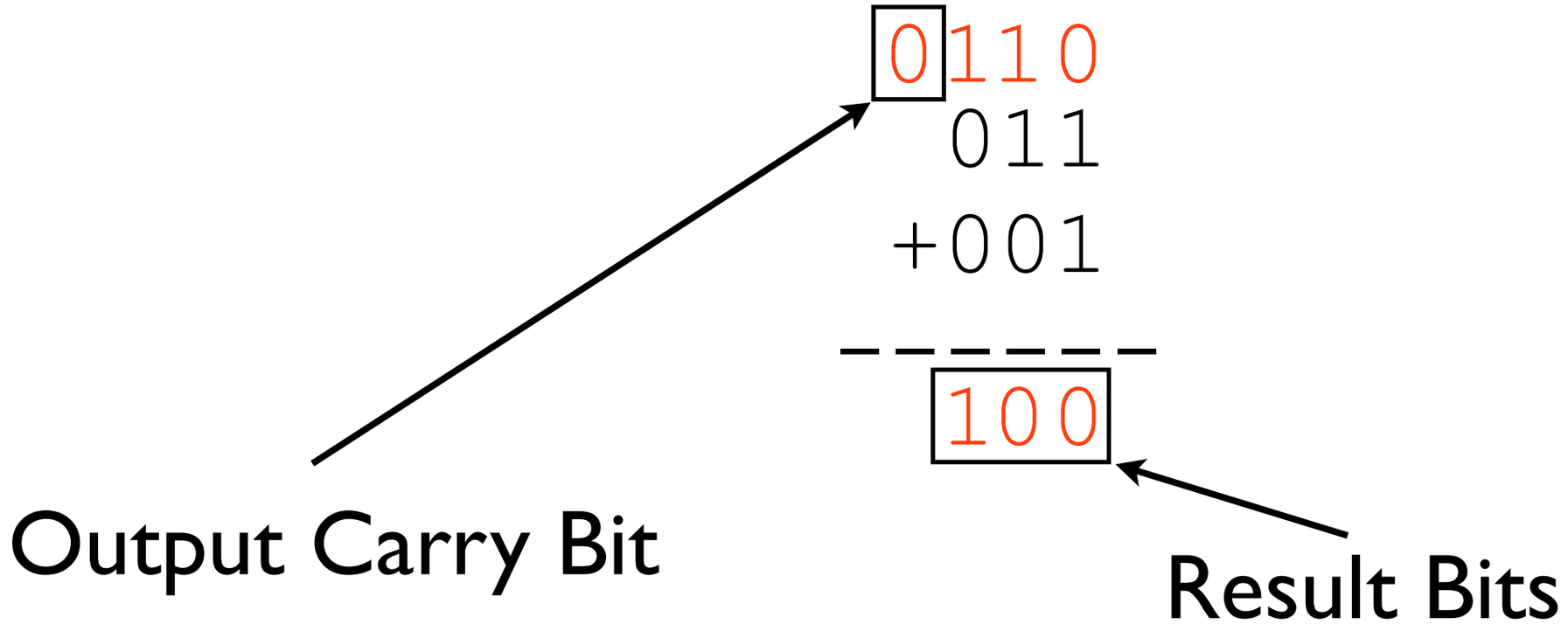
- How might we add the numbers below?

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

-Need an initial carry-in of zero

Adding Multiple Bits

- How might we add the numbers below?



-Need an initial carry-in of zero

Another Example

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

Another Example

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

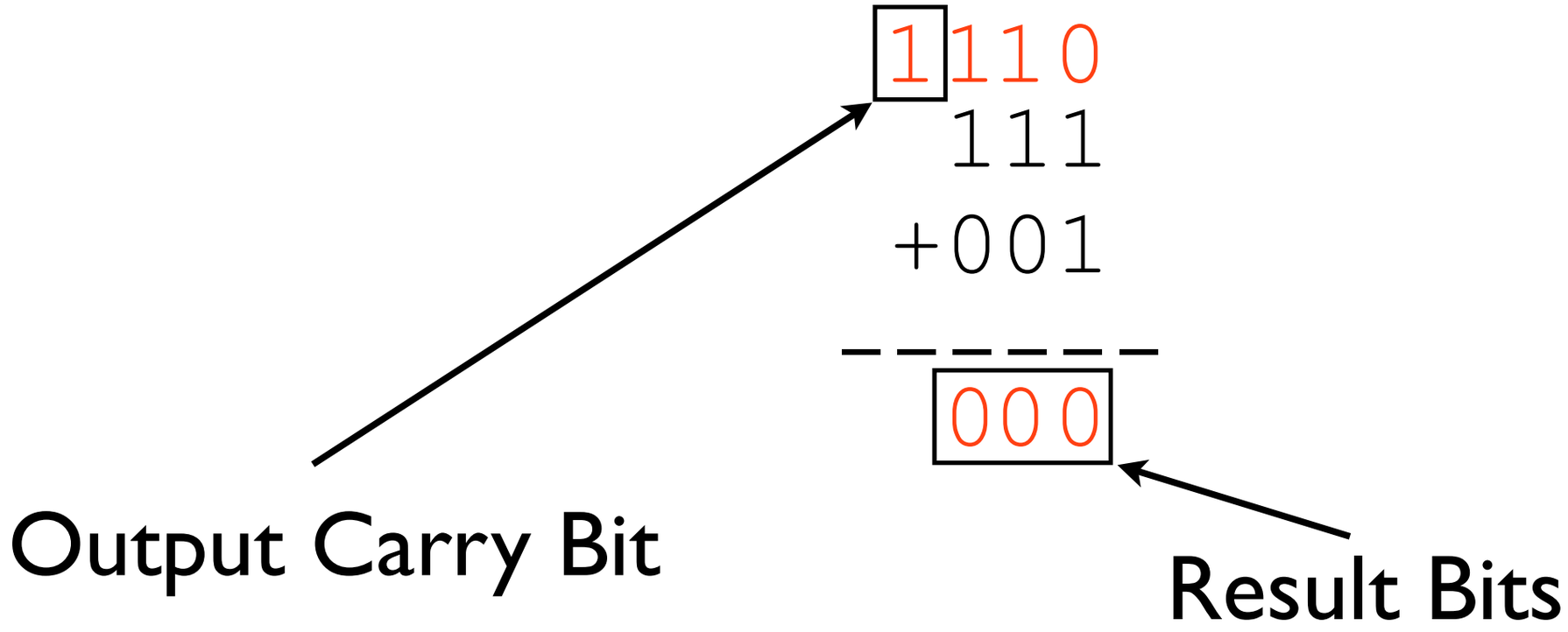
Another Example

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

Another Example

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

Another Example



-Now we have an output carry bit of 1. What does this mean?

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
```

-If these are treated as signed numbers in two's complement, then we need a leading 0 to indicate that this is a positive number
-Truncated to three bits, the result is a negative number!

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

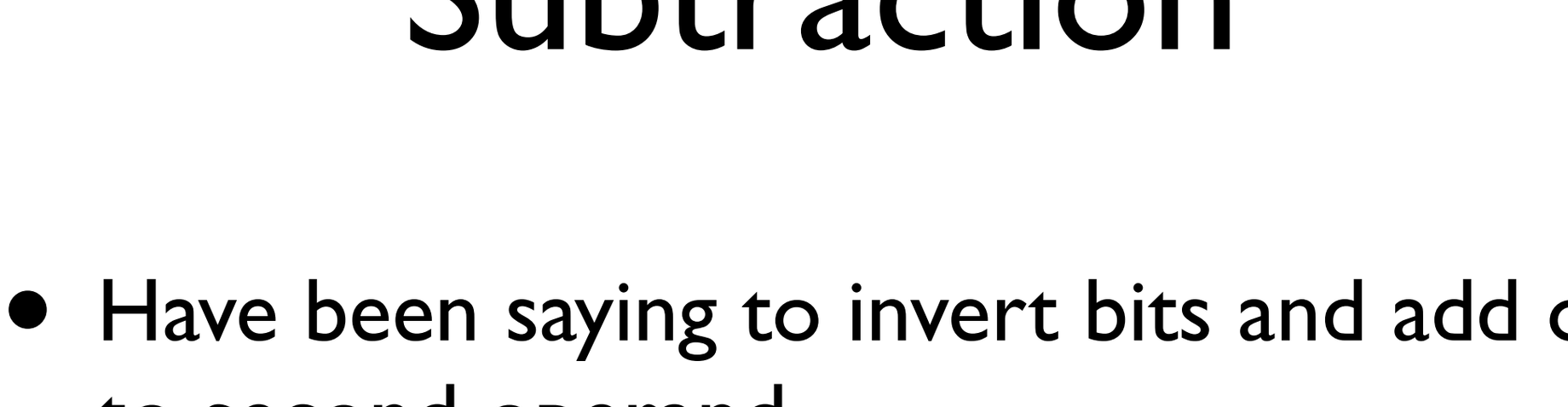
111 +001 ---- 000	011 +011 ---- 110	111 +100 ---- 011	001 +001 ---- 010
No Overflow; Carry	Overflow; No Carry	Overflow; Carry	No Overflow; No Carry

-As to when is it a problem, this all depends on exactly what it is you're doing

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



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
-----
```

Invert `0011`

Subtraction Example

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

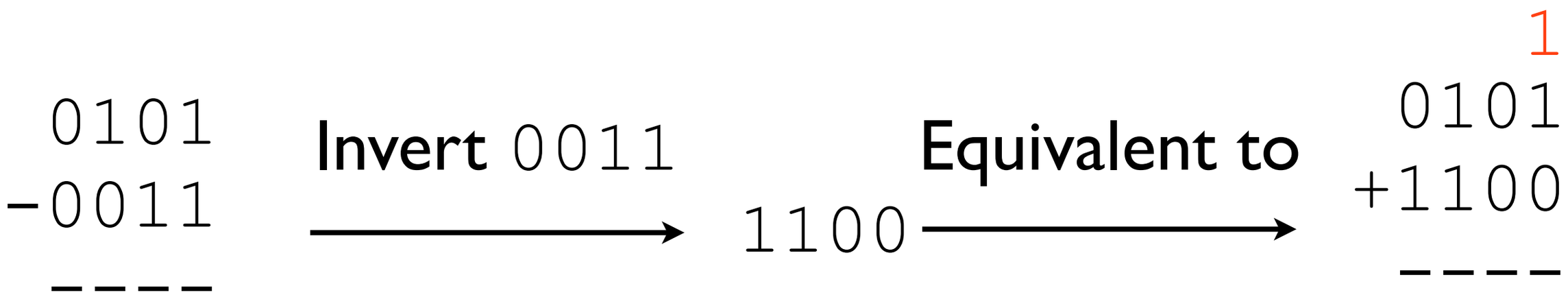
Invert 0011 → 1100

Subtraction Example

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

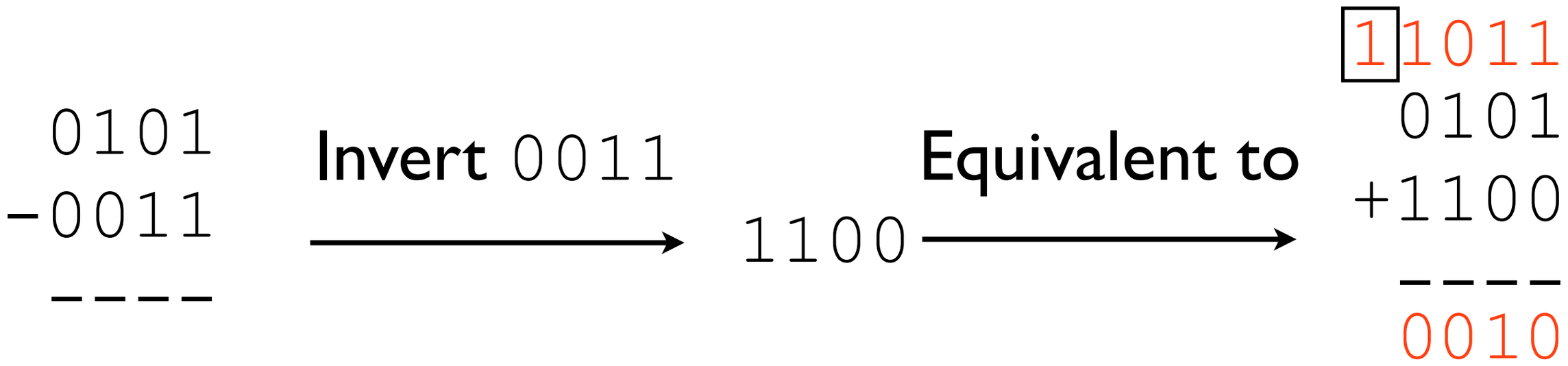
Invert `0011` → `1100` → Equivalent to

Subtraction Example



-An initial carry-in of 1 is equivalent to adding 1 and then adding the other operand

Subtraction Example



-An initial carry-in of 1 is equivalent to adding 1 and then adding the other operand

Floating Point Introduction

Question

How might we represent floating point numbers?

```
1.25

47.9

0.82
```

-A lot of different ways possible
-A whole lot of problems related to precision arise. Just about any representation devisable will be complex.

Enter IEEE-754

- Standardized floating point representation and operations
- Modern systems all use this
- Complex and *weird*

Enter IEEE-754

- Standardized floating point representation and operations
- Modern systems all use this
- Complex and *weird*

```
min(X, Y) =? min(Y, X)
```

Enter IEEE-754

- Standardized floating point representation and operations
- Modern systems all use this
- Complex and *weird*

`min(X, Y) =? min(Y, X)`

May or may not be true...

-Standard doesn't enforce that this is true in general. Implementations are permitted to make it so this isn't true in all cases.

Basis

Based on the idea of *scientific notation*

Basis

Based on the idea of *scientific notation*

$4.23 * 10^7$

Basis

Based on the idea of *scientific notation*

$4.23 * 10^{7}$

Save these

Basis

Based on the idea of *scientific notation*

$4.23 * 10^{7}$

Save these

Caveat: this is in binary

$1.1 * 2^{-1}$

Components

$1.1 * 2^{-1}$

- Sign bit (+/-)
- Exponent
- Fraction / mantissa

-We'll get more into representation next class; this is the birds-eye view of how this works for now.