CS 64 Week 1 Lecture 1

Kyle Dewey

Overview

- Bitwise operation wrap-up
- Two's complement
- Addition
- Subtraction
- Multiplication (if time)

Bitwise Operation Wrap-up

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

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
```

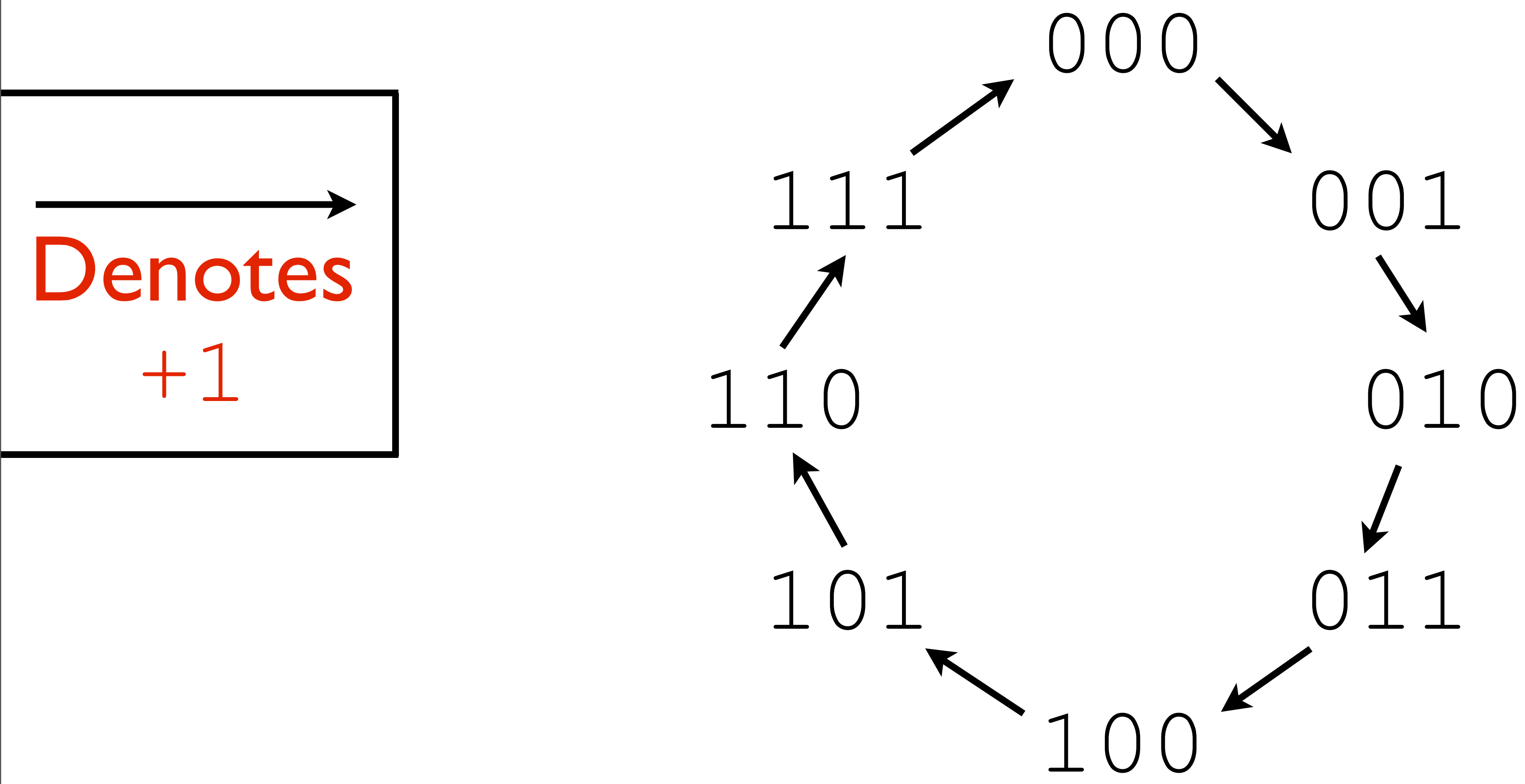
We started with `1011` - negative

Where Is Twos Complement From?

- Intuition: try to subtract 1 from 0, in decimal
  - Involves borrowing from an invisible number on the left
  - Twos complement is based on the same idea

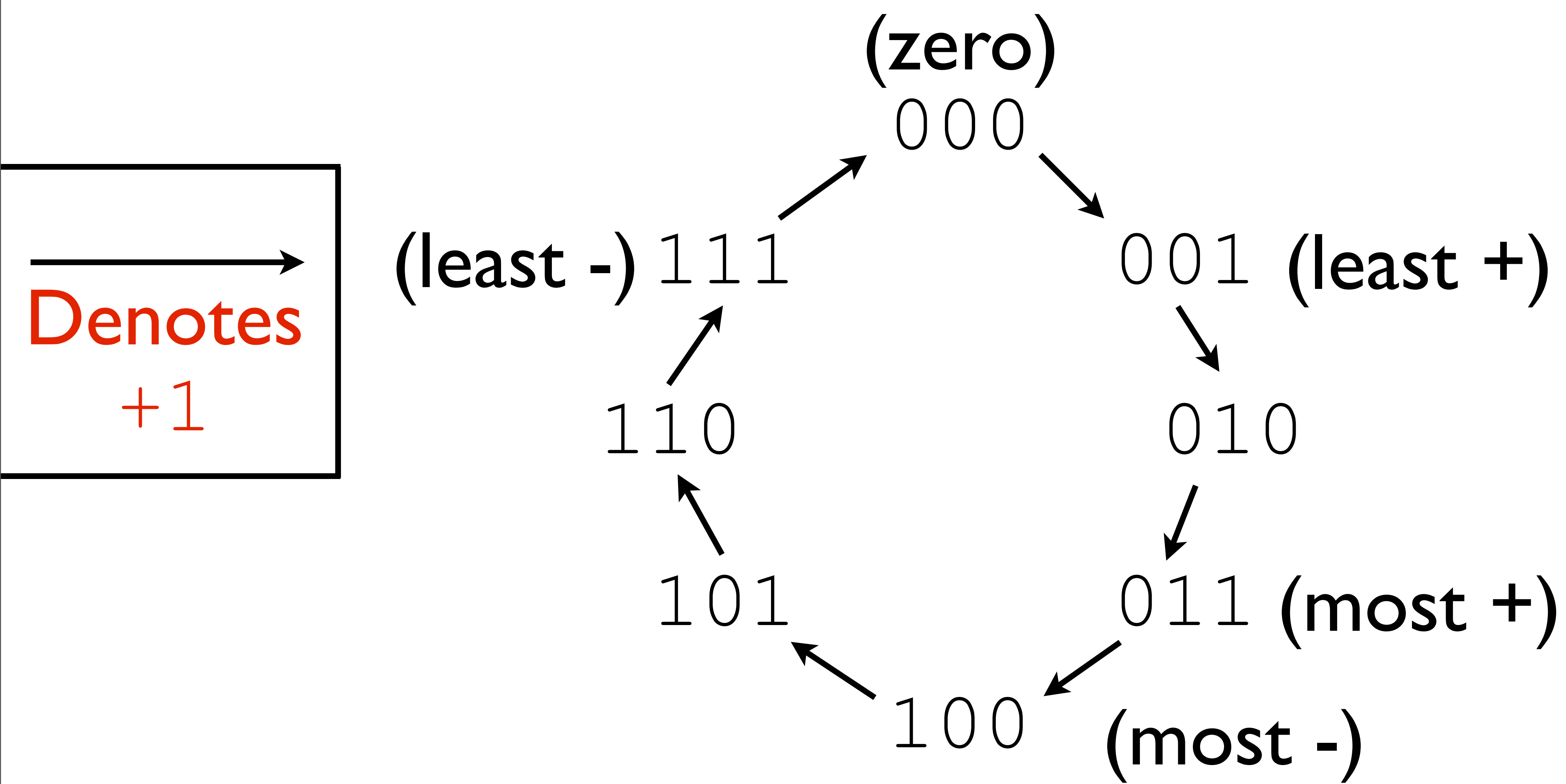
Another View

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



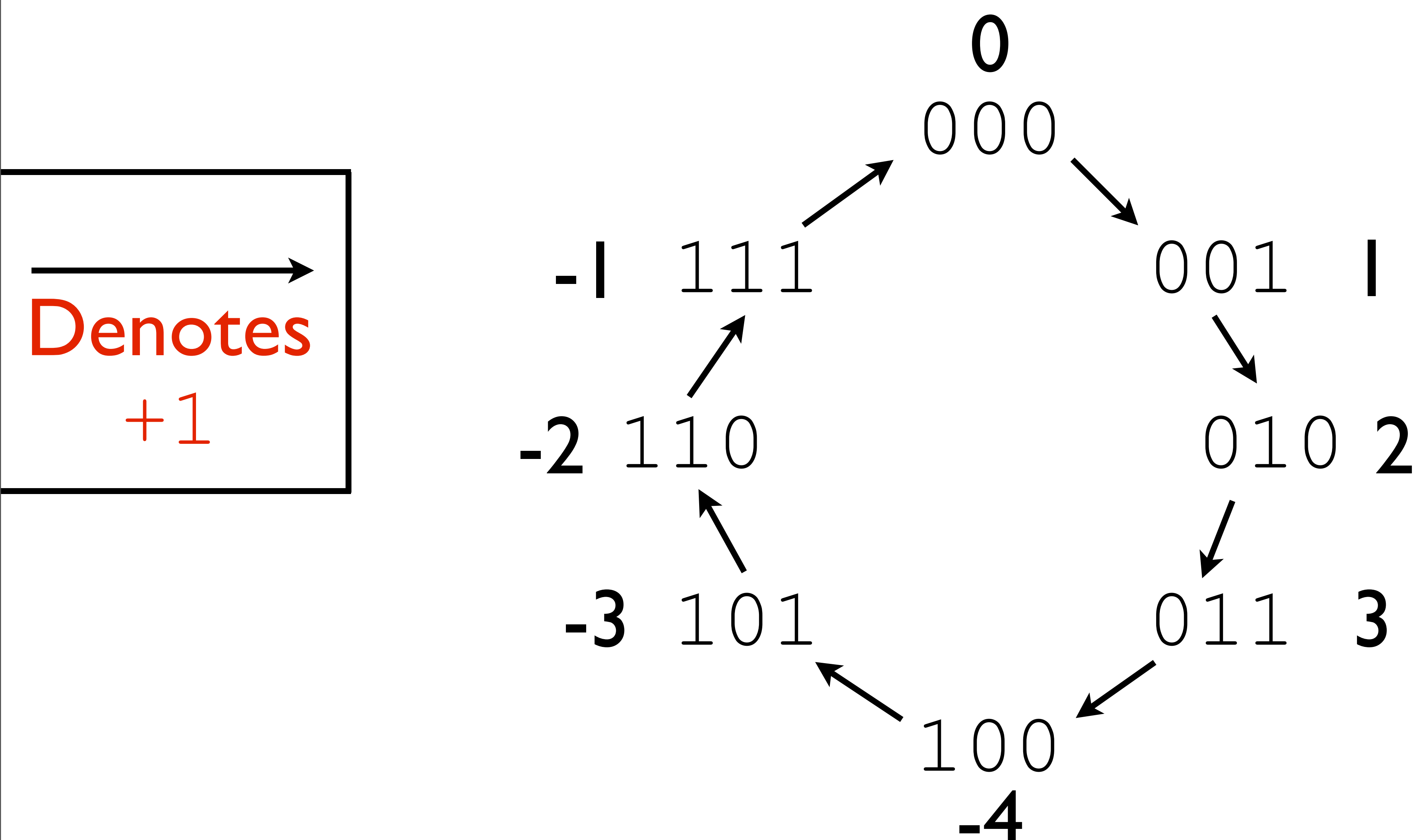
Another View

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

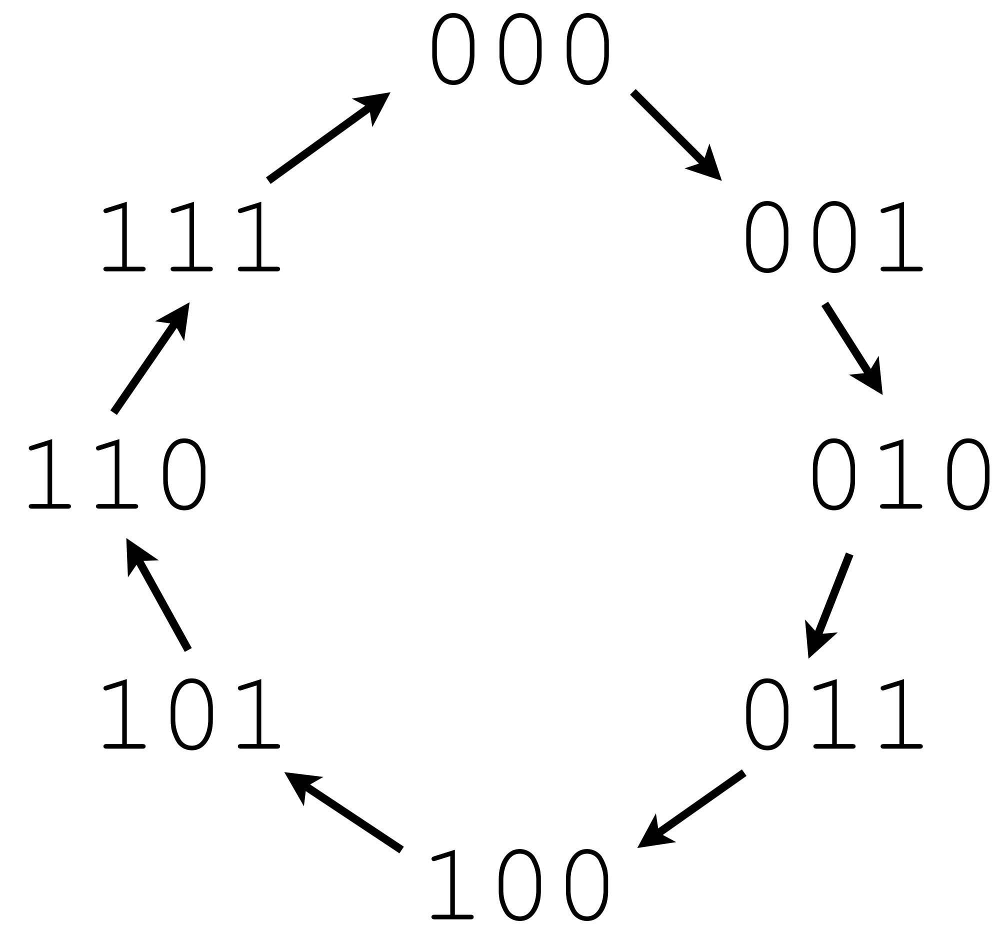


Another View

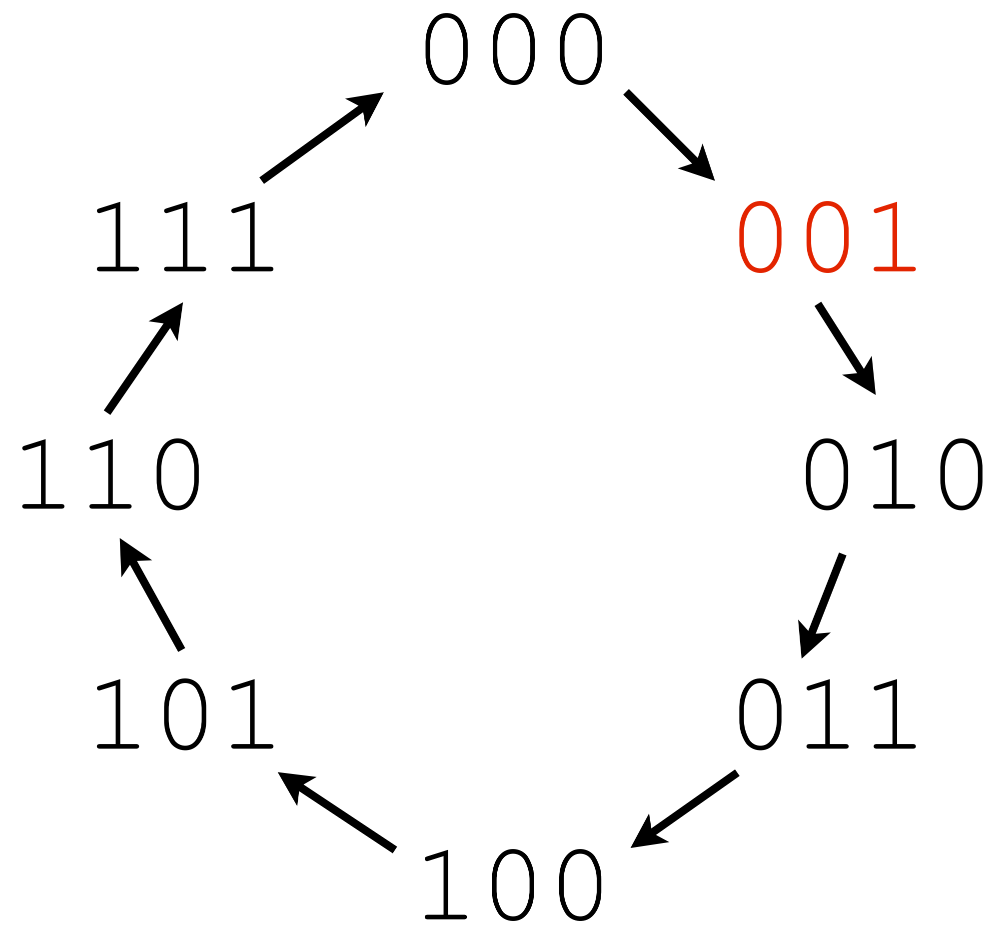
- Modular arithmetic, with the convention that a leading `1` bit means negative



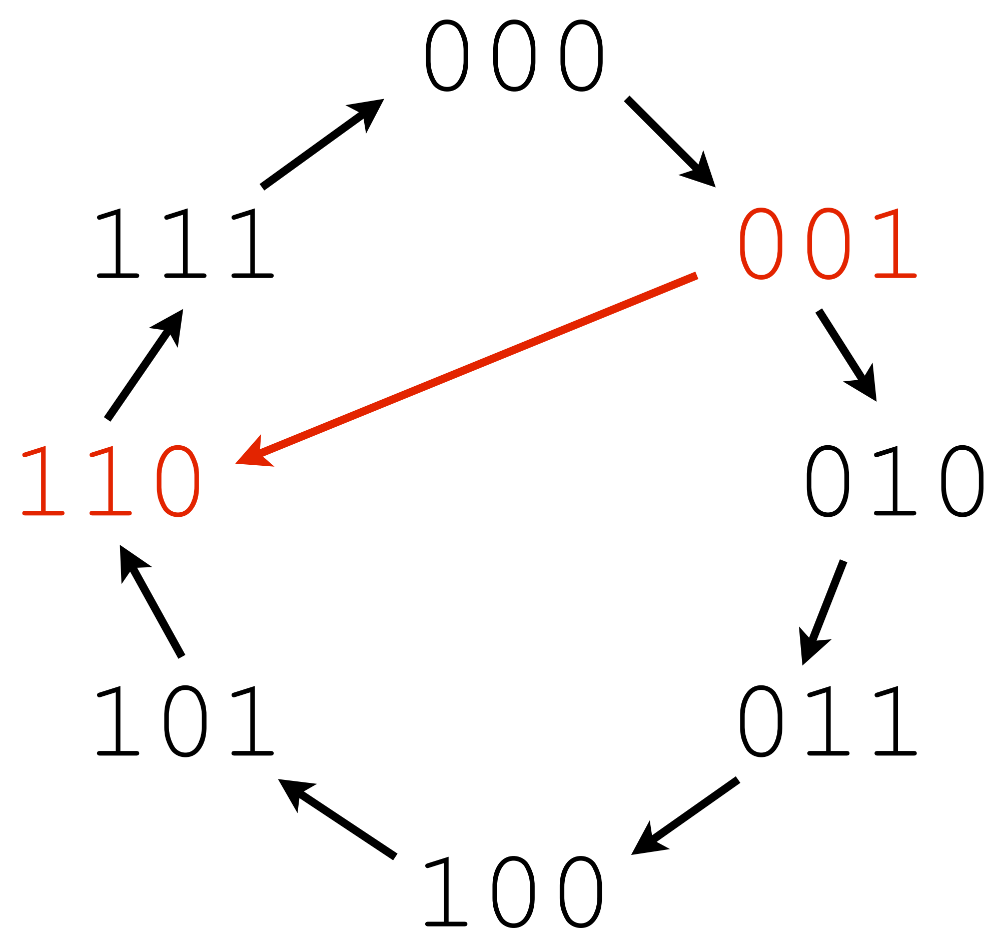
Negation of `1`



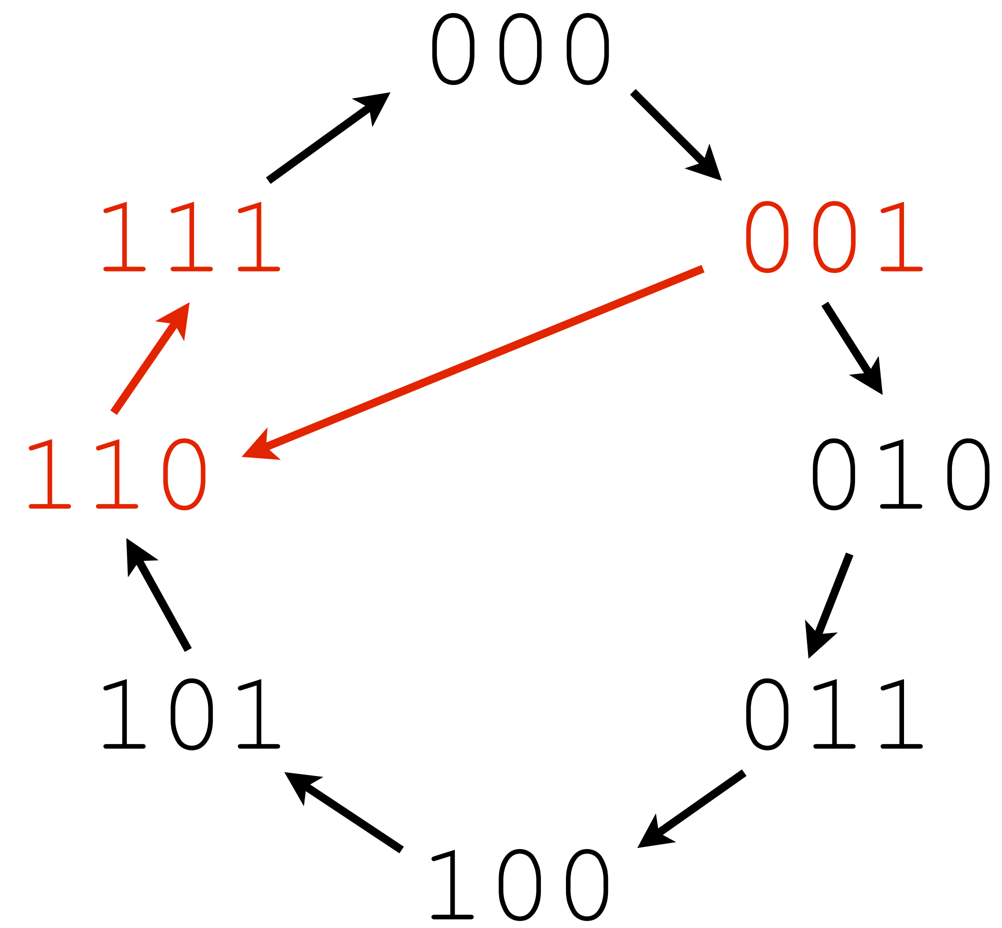
Negation of 1



Negation of `1`

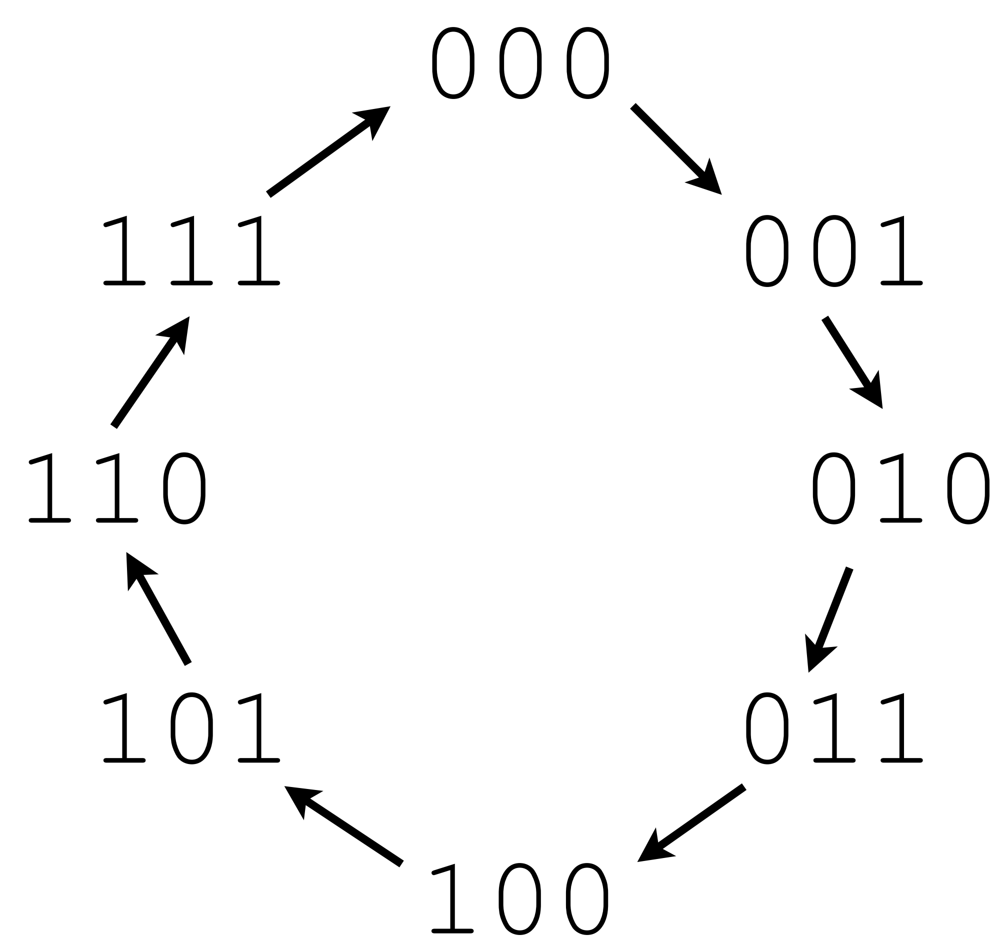


Negation of `1`



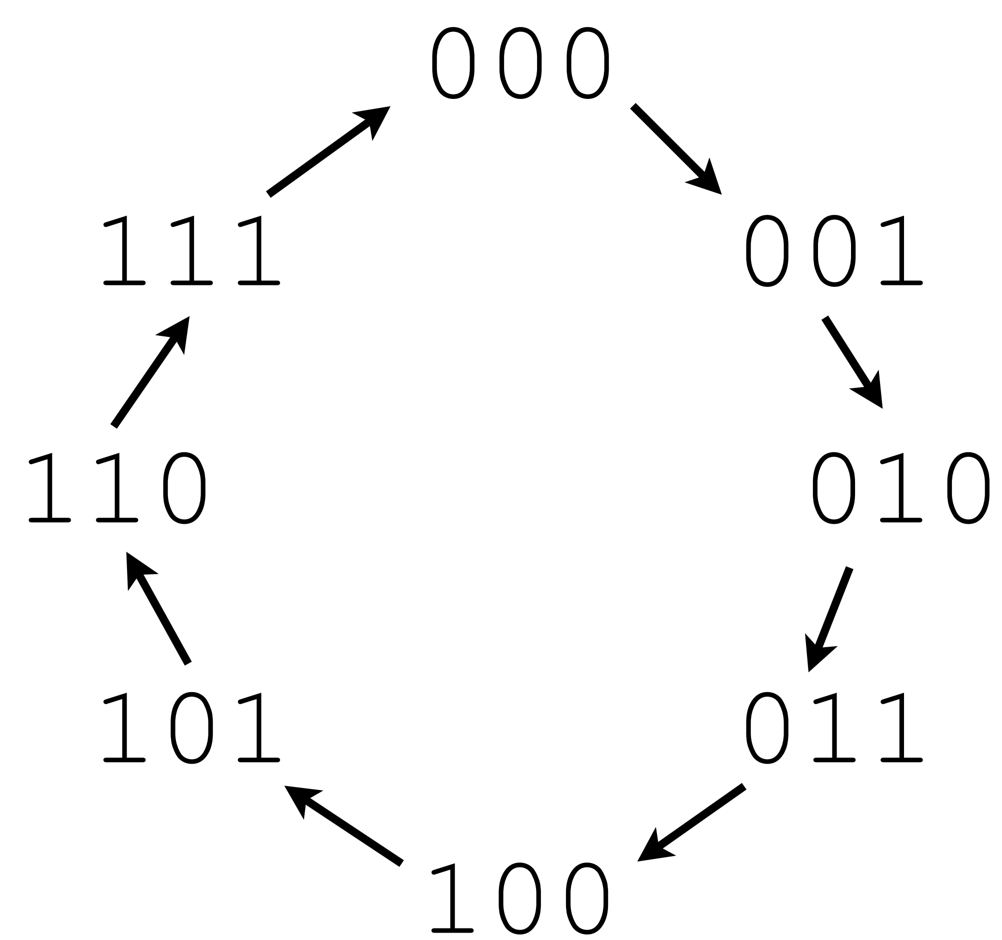
Consequences

- What is the negation of `000`?



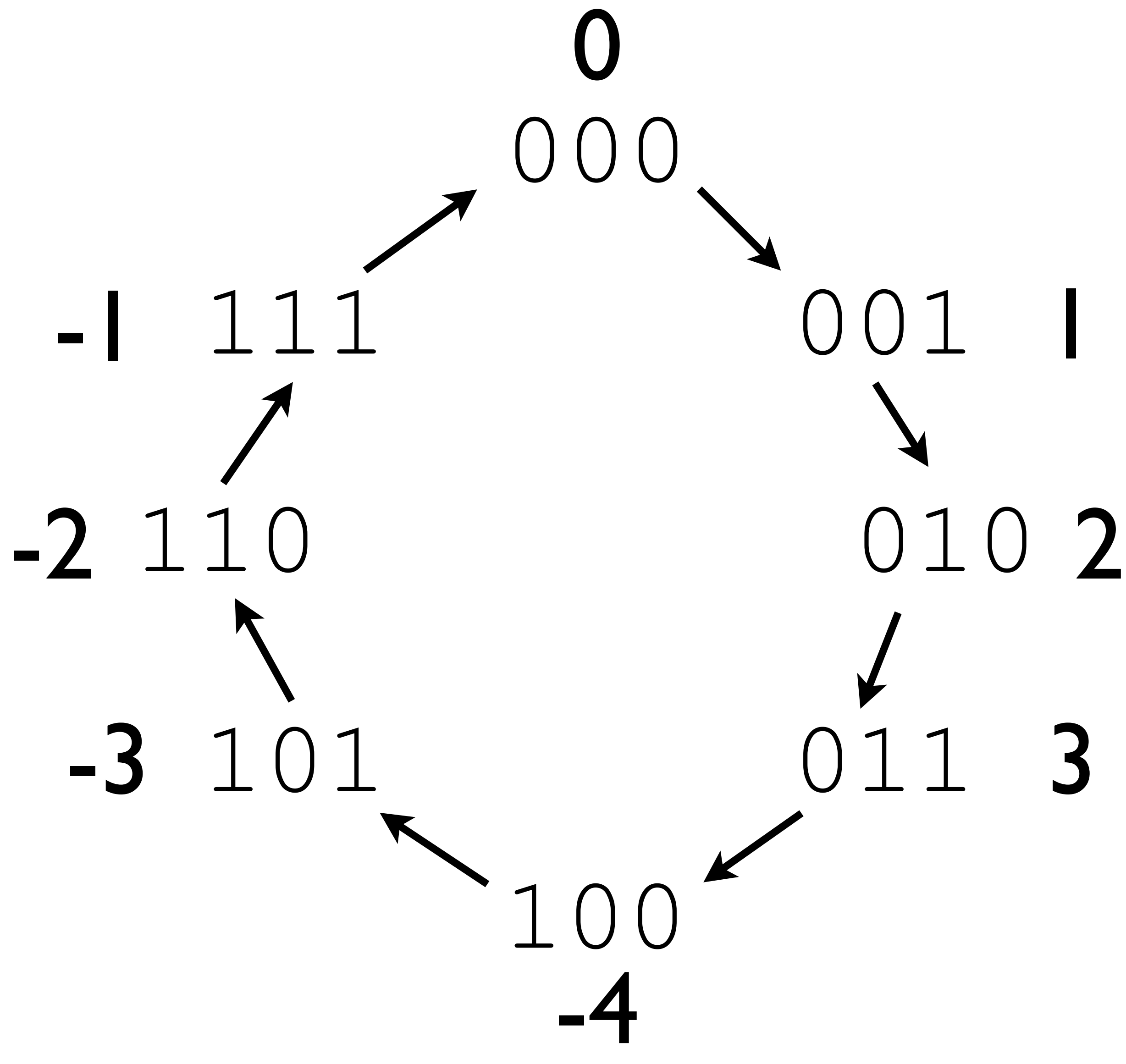
Consequences

- What is the negation of `100`?



Arithmetic Shift Right

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



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
----
    ?
```

			<pre> 6 +3 -- ?</pre>

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

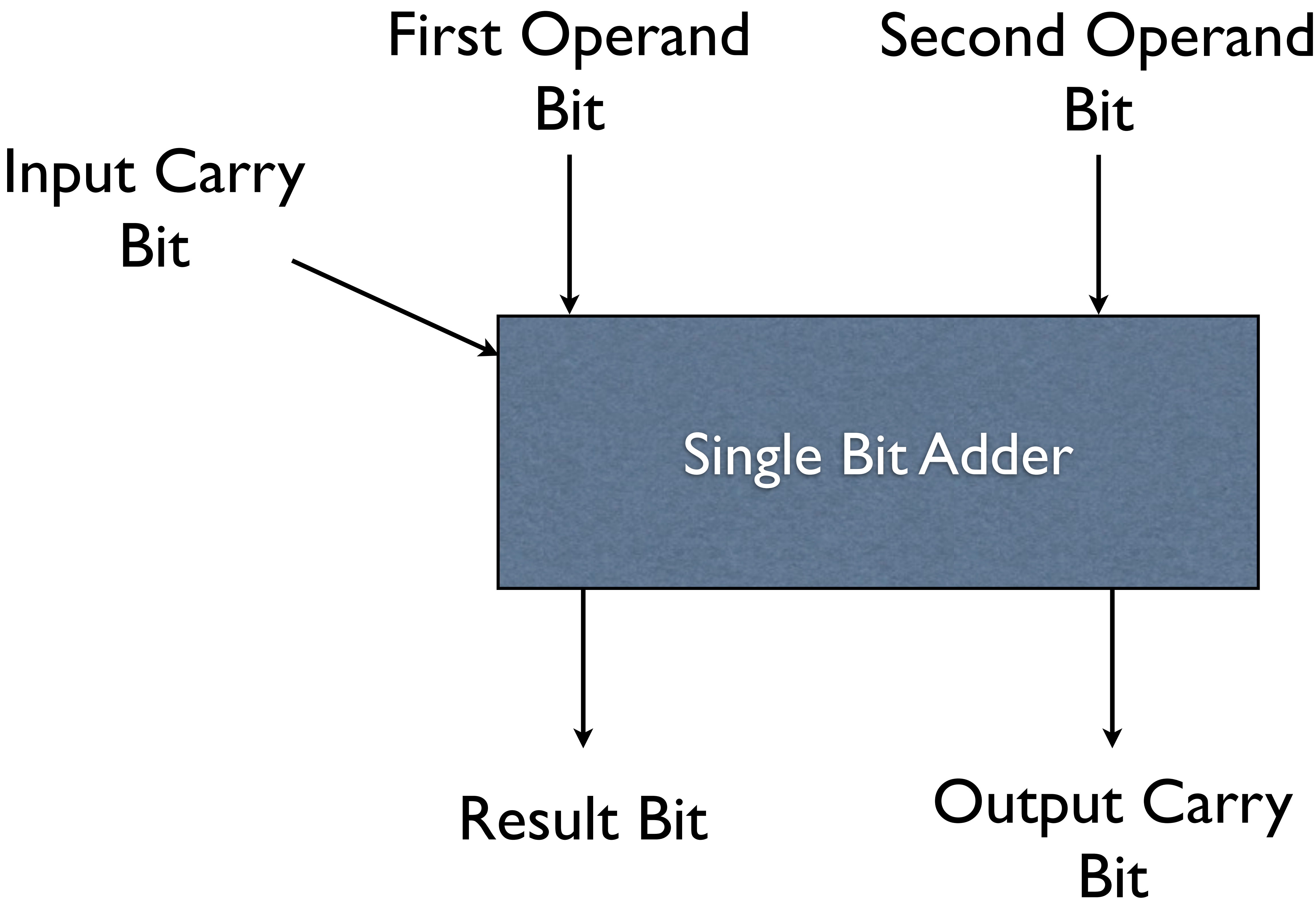
0 +0 -- 0	0 +1 -- 1	1 +0 -- 1	1 +1 -- 0 Carry: 1

Chaining the Carry

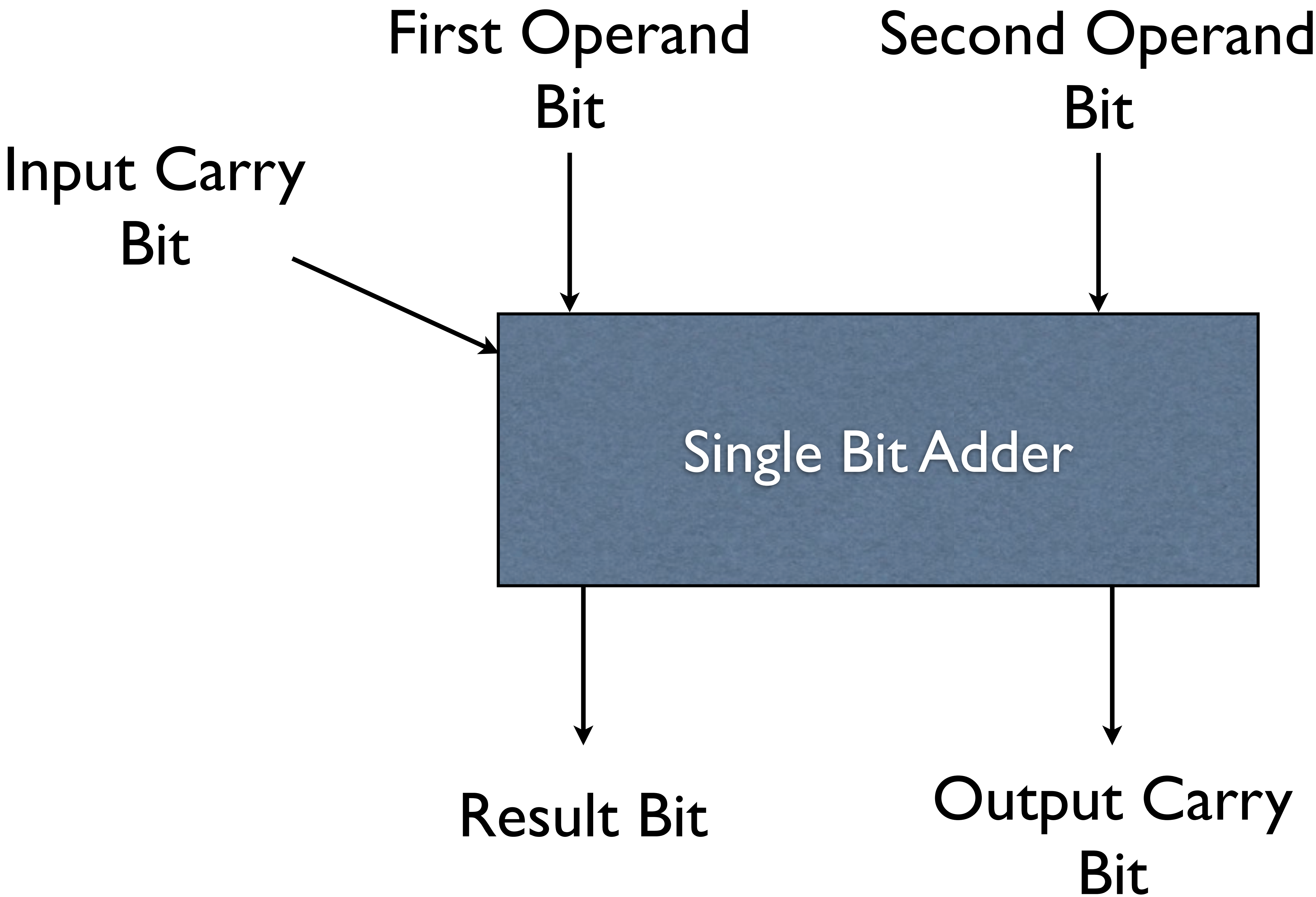
- Also need to account for any input carry

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

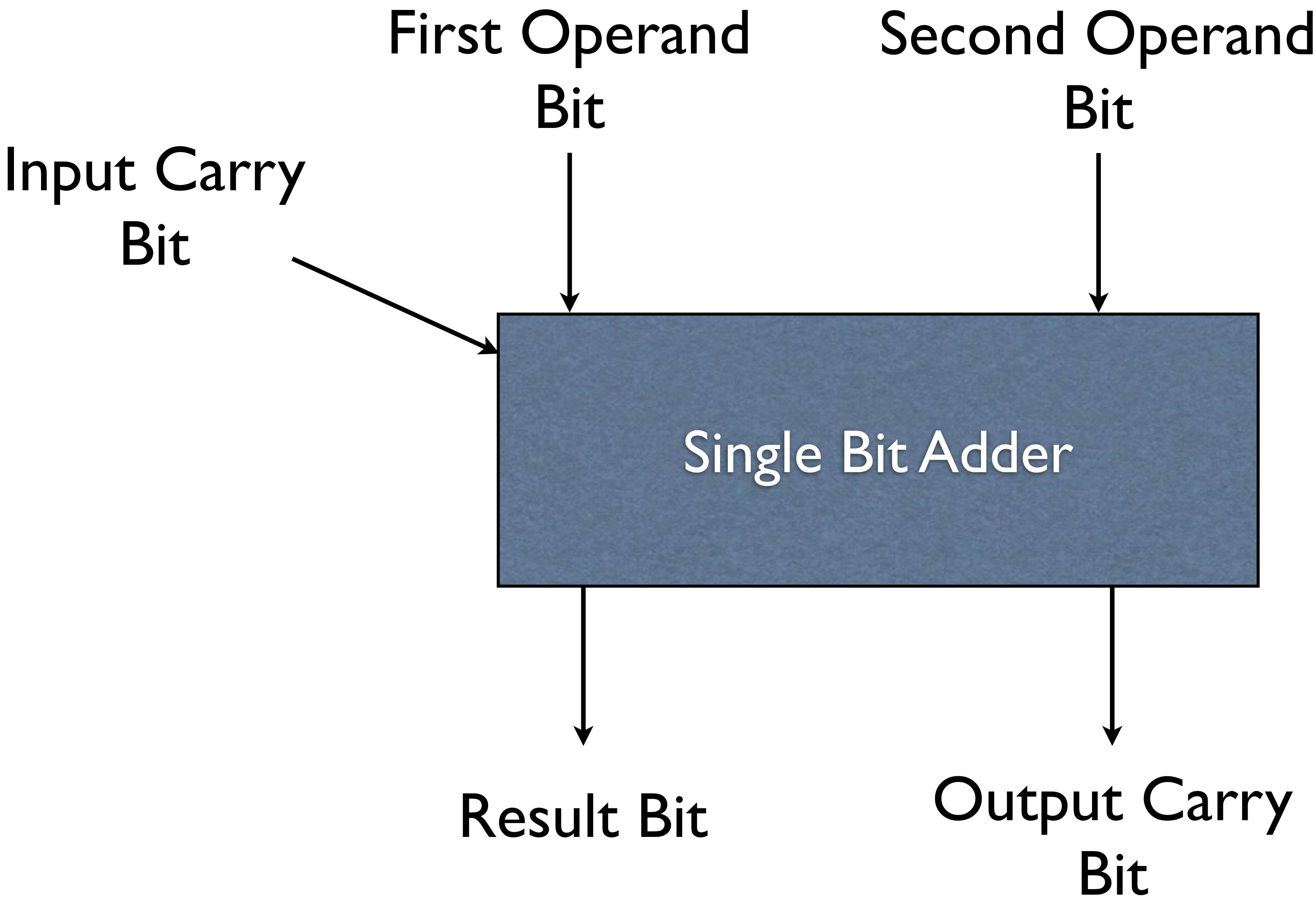
In Summary


First Operand Bit
Second Operand Bit
Input Carry Bit
Single Bit Adder
Result Bit
Output Carry Bit

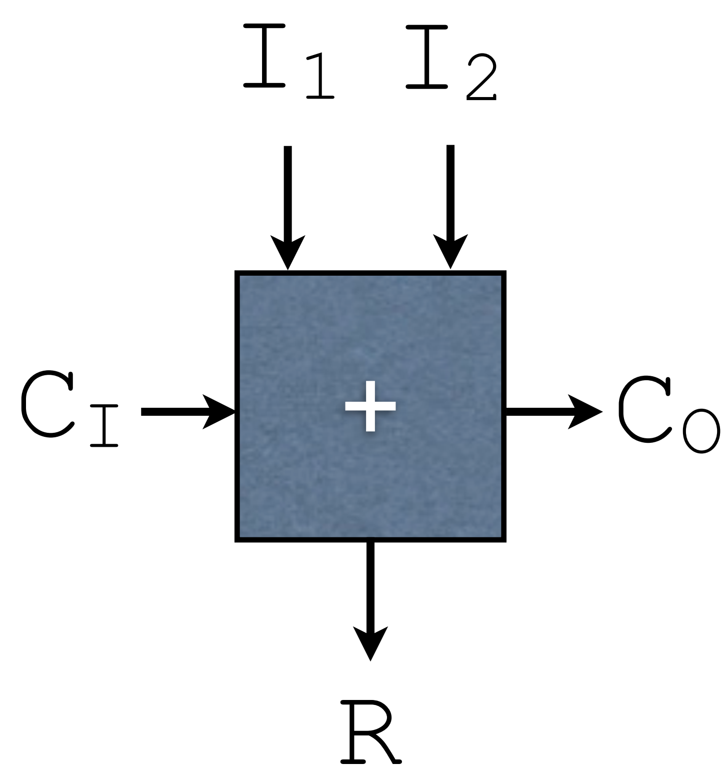
- How can we adapt this to add multi-digit binary numbers together?



Putting it Together

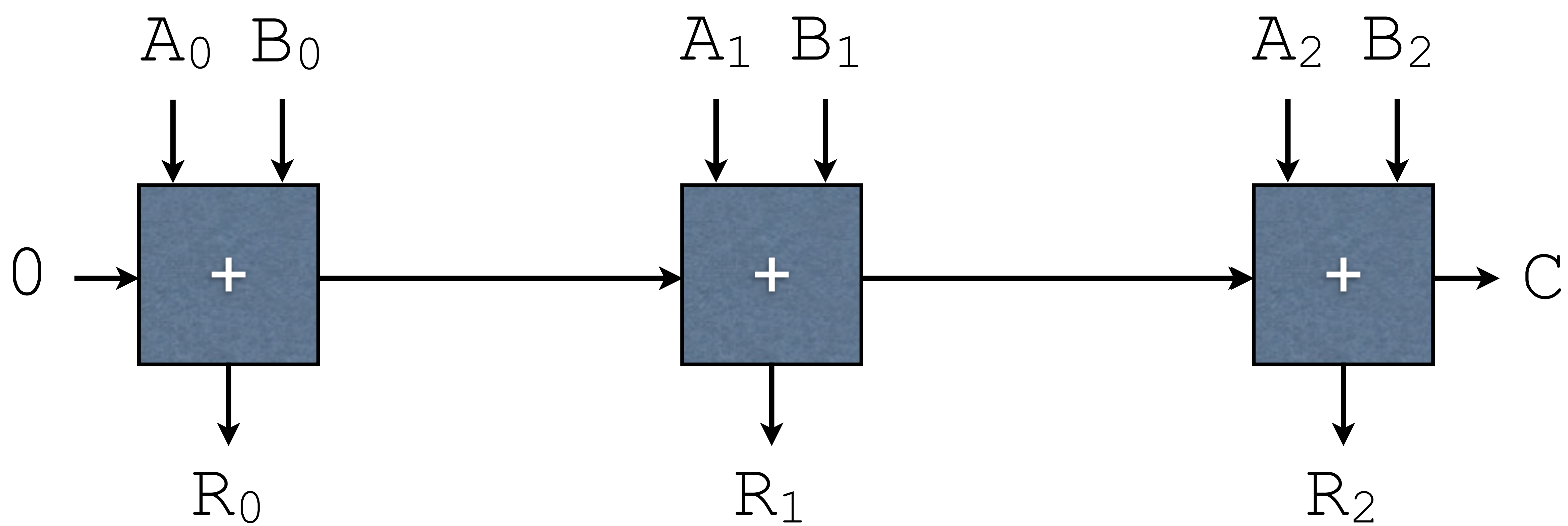


Putting it Together



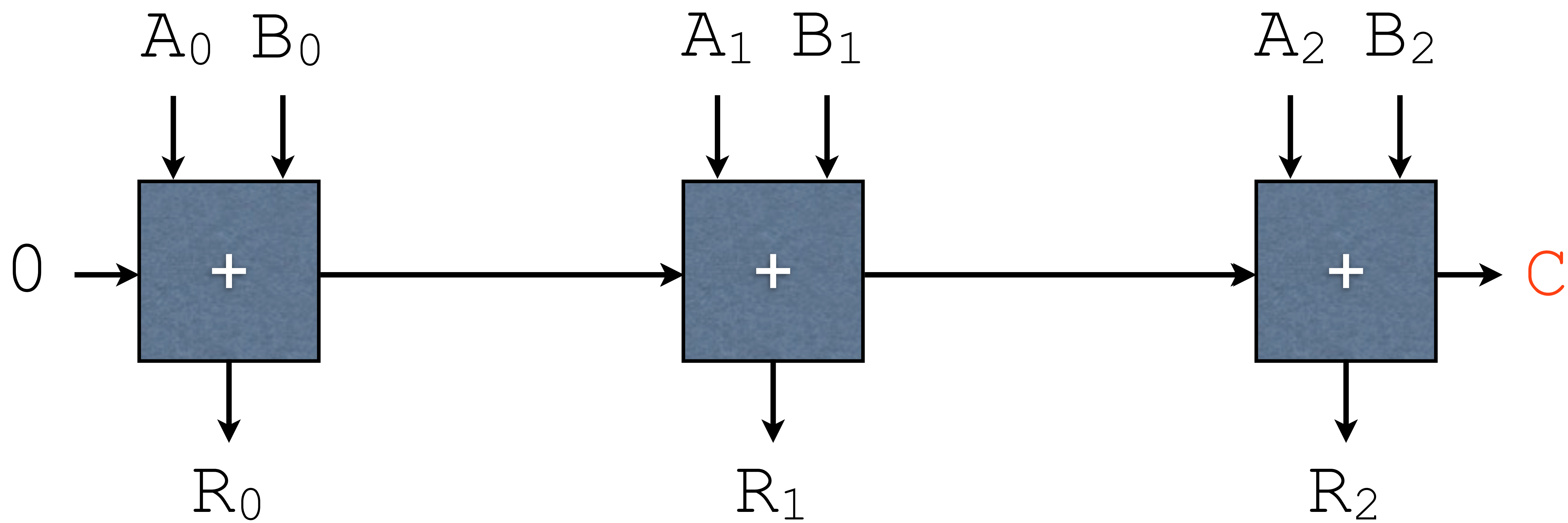
Putting it Together

For two three-bit numbers, A and B, resulting in a three-bit result R



Output Carry

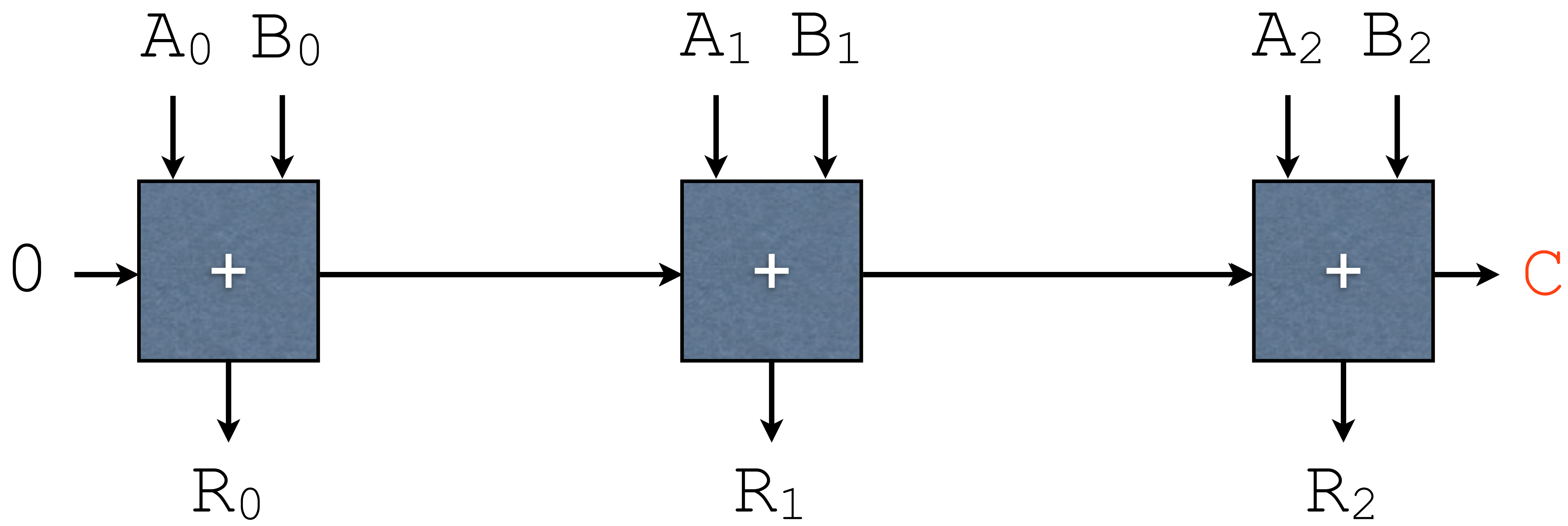
What about the output carry bit?



Output Carry

What about the output carry bit?

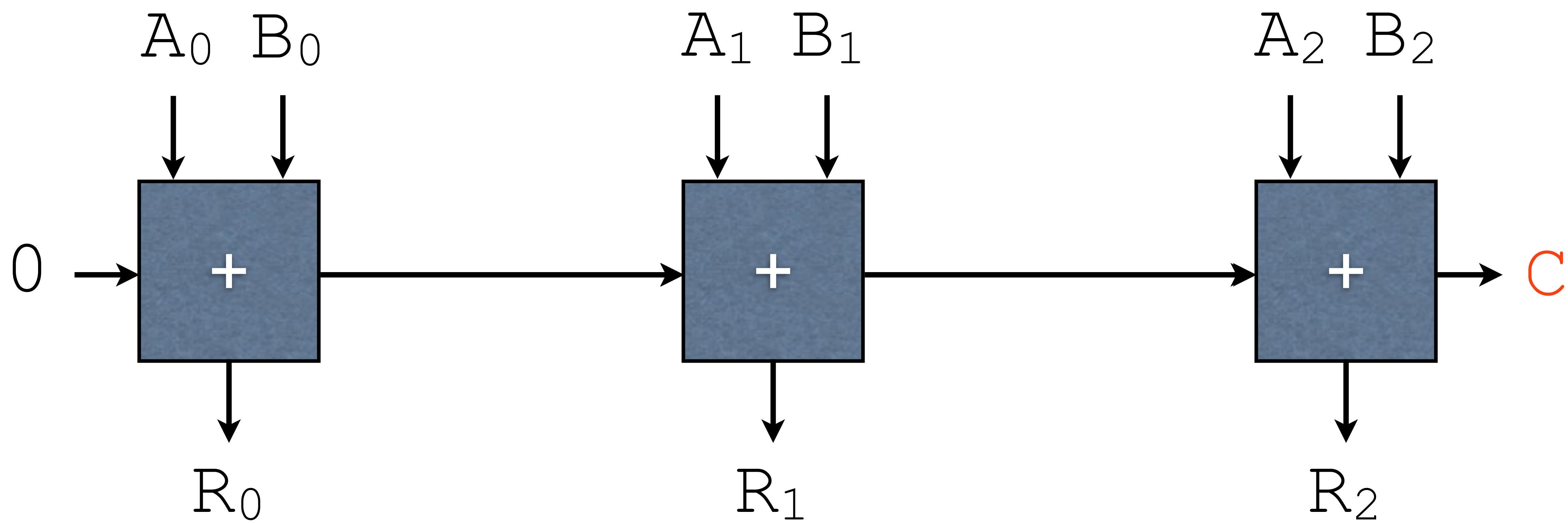
```
A: 111
B: 001
R: ?, C: ?
```



Output Carry

What about the output carry bit?

```
A: 111
B: 001
R: 000, C: 1
```



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
----
0111
```

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`
	`----`
	`111`
		`111`
		`+100`
		`----`
		`011`
			`001`
			`+001`
			`----`
			`010`

No Overflow; Carry	Overflow; No Carry	Overflow; Carry	No Overflow; No Carry

Subtraction

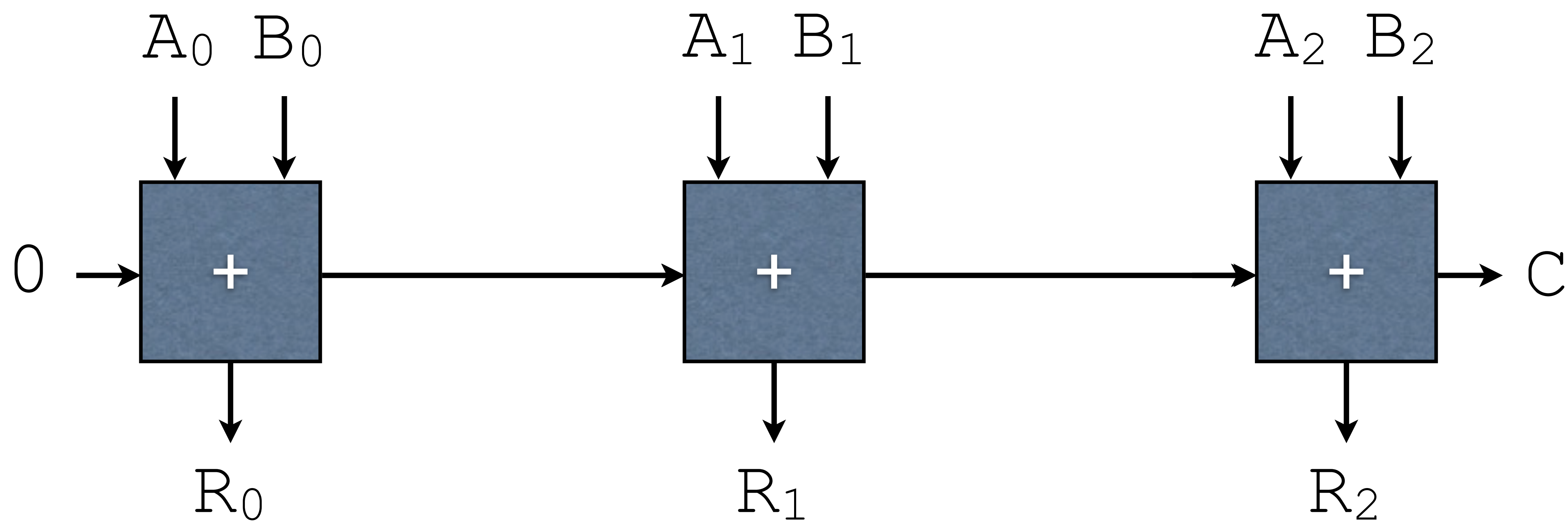
Subtraction

- 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
-001                             +111
----     (equivalent to)         ----
   ?                                ?
```

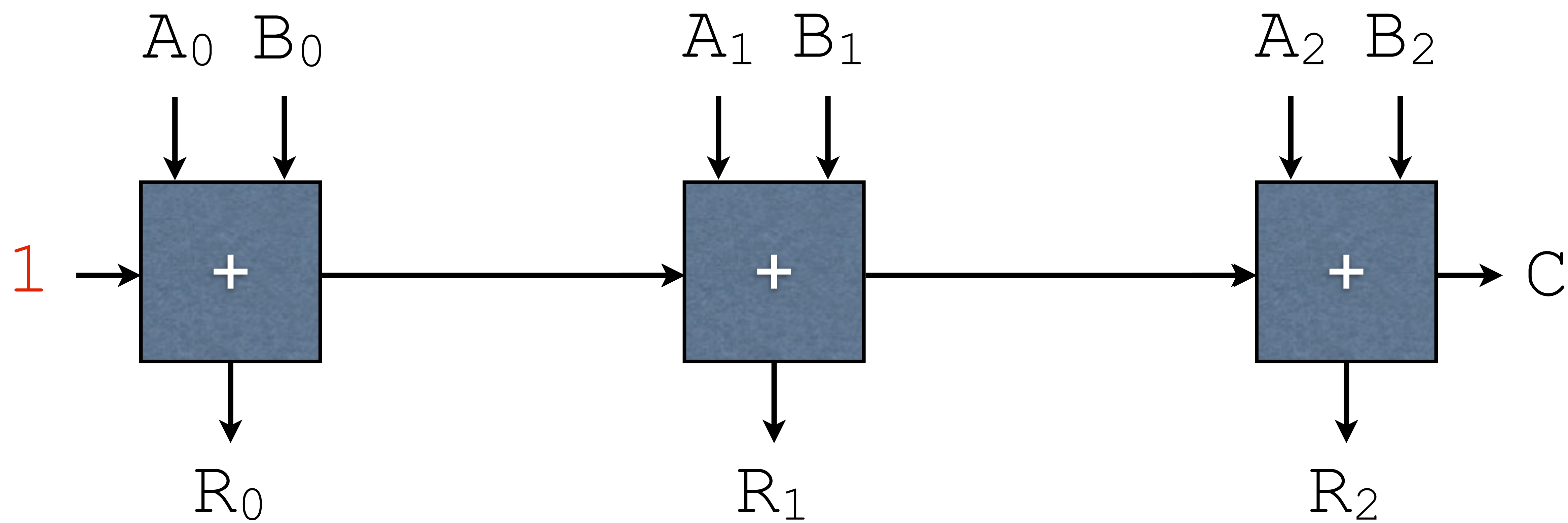
Subtraction Trick

- Assume we can invert bits, but we cannot add one in a separate step
- How might we make this work given only our three-bit adder from before?



Subtraction Trick

- Put in an initial carry of `1`: this indicates to add `1` anyways



Multiplication (if time)

Multiplication

- For simplicity, we will only consider positive values here
- A number of different algorithms exist; we will only look at one of them

Central Idea

- Accumulate a *partial product*: the result of the multiplication as we go on
  - Computed via a series of additions
- When we are finished, the partial product becomes the final product (the result)
- Build off of addition and multiplication of a single digit (much like with addition)

Decimal Algorithm

- Let `P` be the partial product, `M` be the multiplicand, and `N` be the multiplier
- Initially, `P` is 0
- If `N` is `0`, then `P` = the result
- If not, then `P  +=` (the rightmost digit of `N`) times `M`
- Shift `N` right once, and `M` left once
- Repeat

Example

- Performing 803 * 151

Example

- Performing 803 * 151

P	M	N

Example

- Performing 803 * 151

P	M	N
0	803	151

Initially `P = 0`,
`N` = multiplicand
`M` = multiplier

Example

- Performing 803 * 151

P	M	N
0	803	151

N is not 0

Example

- Performing 803 * 151

P	M	N
0	803	151
803		

`P += ` (the rightmost digit of `N`) times `M`

Example

- Performing 803 * 151

P	M	N
0	803	151
803	8030	15

Shift N right once, and M left once

Example

- Performing 803 * 151

P	M	N
0	803	151
803	8030	15

N **is not** 0

Example

- Performing 803 * 151

P	M	N
0	803	151
803	8030	15
40953		

`P += ` (the rightmost digit of `N`) times `M`

Example

- Performing 803 * 151

P	M	N
0	803	151
803	8030	15
40953	80300	1

Shift N right once, and M left once

Example

- Performing 803 * 151

P	M	N
0	803	151
803	8030	15
40953	80300	1

N is not 0

Example

- Performing 803 * 151

P	M	N
0	803	151
803	8030	15
40953	80300	1
121253		

`P` += (the rightmost digit of `N`) times `M`

Example

- Performing 803 * 151

P	M	N
0	803	151
803	8030	15
40953	80300	1
121253	803000	0

Shift N right once, and M left once

Example

- Performing 803 * 151

P	M	N
0	803	151
803	8030	15
40953	80300	1
121253	803000	0

N is 0; done

Intuition

- Only looking at rightmost digit of `N`: getting partial product of that digit with the rest
- Shifting `M` left: for each digit of `N` observed, we look one digit deeper in `M` (and result gets correspondingly larger)
- Similar to traditional pencil-and-paper algorithm (which shifts partial products instead)

Why this Algorithm?

- Looks complex...ish
- On binary, things get simpler. Why?

- Initially, `P` is 0
- If `N` is `0`, then `P` = the result
- If not, then `P += `(the rightmost digit of `N`) times `M`
- Shift `N` right once, and `M` left once
- Repeat

Why this Algorithm?

- Looks complex...ish
- On binary, things get simpler. Why?

- Initially, `P` is 0
- If `N` is `0`, then `P` = the result
- If not, then `P += ` (the rightmost digit of `N`) times `M`
- Shift `N` right once, and `M` left once
- Repeat

Simplified Binary Algorithm

- Initially, `P` is 0
- If `N` is `0`, then `P` = the result
- If not, then `P += `(the rightmost digit of `N`) times `M`
- Shift `N` right once, and `M` left once
- Repeat

Simplified Binary Algorithm

- Initially, `P` is 0
- If `N` is `0`, then `P` = the result
- If the rightmost digit if `N` is `1`:
  - `P += M`
- Shift `N` right once, and `M` left once
- Repeat

Dealing with Negative Numbers

- Can still be done, but we need extra logic
  - Negative times negative is a positive, positive and a negative is a positive...
- Not fundamentally harder, and showing this extra detail just complicates things in an uninteresting way