CS/COE 0447

Binary Arithmetic

wilkie (with content borrowed from:
Jarrett Billingsley
Dr. Bruce Childers)

Binary Addition

Computers and 2 + 2 (well, 10 + 10, eh?)

Adding in Binary

- It works the same way as you learned in school
  - Except instead of carrying at 10_{10}, you carry at... 10_2!
    - 1 + 1 = $\mathbf{10_2}$ (2_{10})
    - 1 + 1 + 1 = $\mathbf{11_2}$ (3_{10})
- Let's try it. (what are these in decimal?)

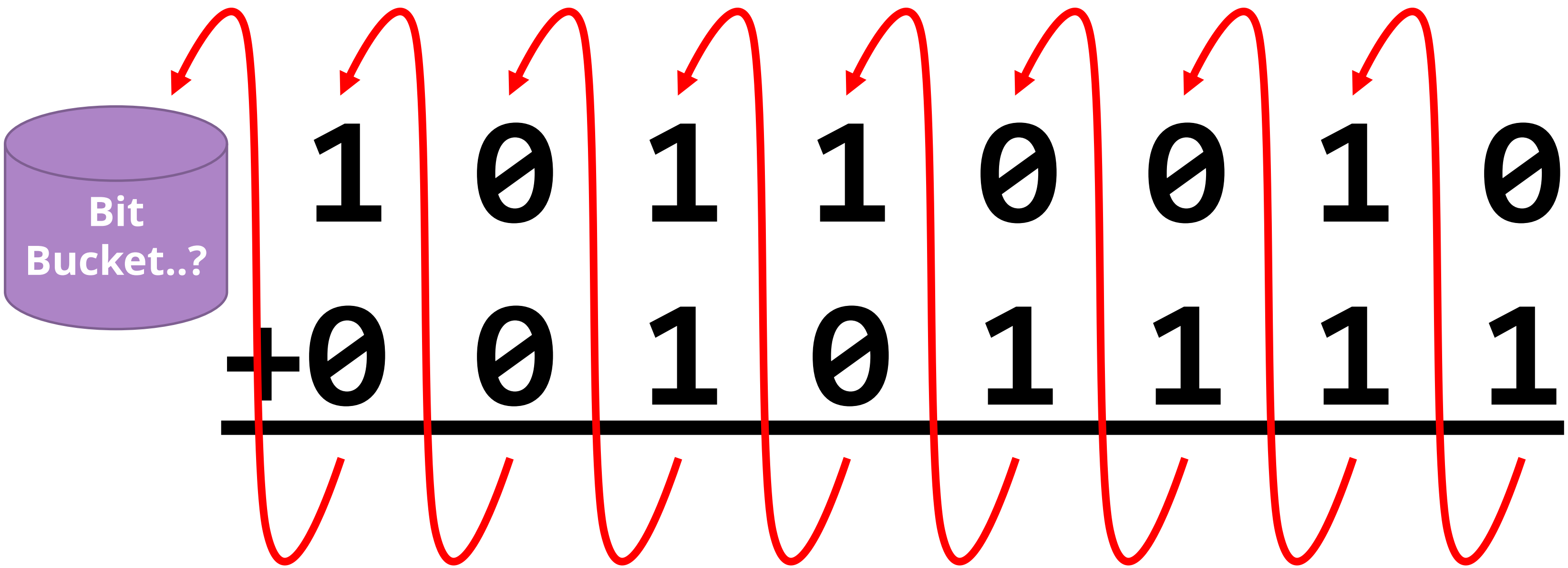
```
 1011 0010
+0010 1111
----------
```

Formalizing “Addition”

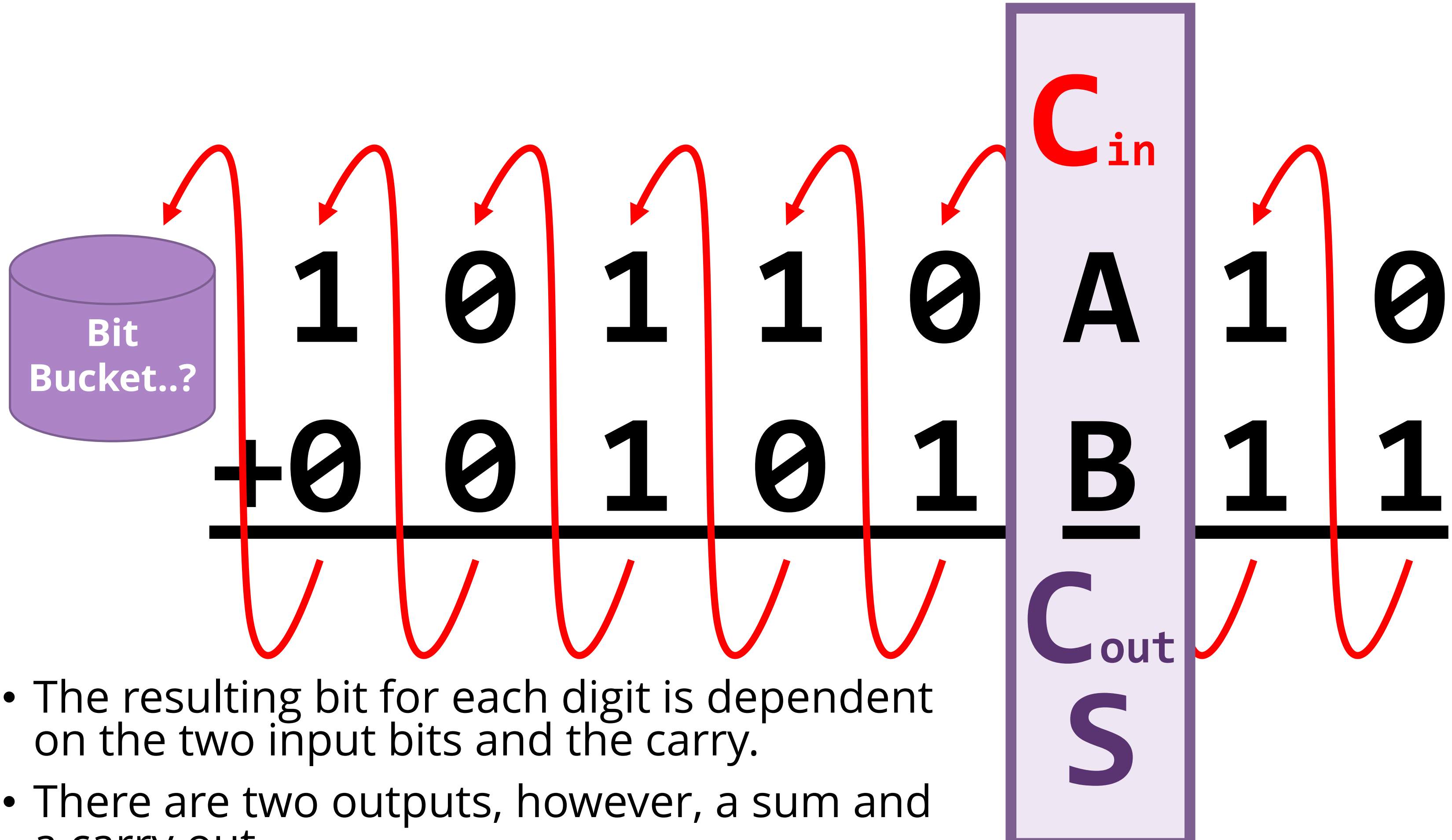
- For each pair of bits **starting at the LSB,**
  - Add the two bits and the carry
  - The **low bit** of the sum goes into the sum row
  - The **high bit** of the sum is the carry for the **next higher bit**
- This is the **grade school algorithm**
  - Cause it's how you learned to add in grade school
- **What if there's a carry out of the biggest column??**
  - That’s ovvvvvverrrrflooooow (remember that?)

Ripple Adder (The Ole Classic)

- When you add one place, you might get a **carry out**.
- That becomes the **carry in** for the next higher place.



Looking at just a bit of this...



- The resulting bit for each digit is dependent on the two input bits and the carry.
- There are two outputs, however, a sum and a carry out.

1-bit Adder

- Let's try to come up with a **truth table** for adding two bits.
- Each column will hold **1 bit**.
- We will ignore the carry input, for now.

A	B	C_o	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Half-Truth Tables

- What we just made was a **half-adder**.
- It has a carry *output* but not a carry *input*
  - (which might be useful for the lowest bit)
- To make a **full adder,** we need **3 input bits**.

C_i	A	B	C_o	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

The logic of it all

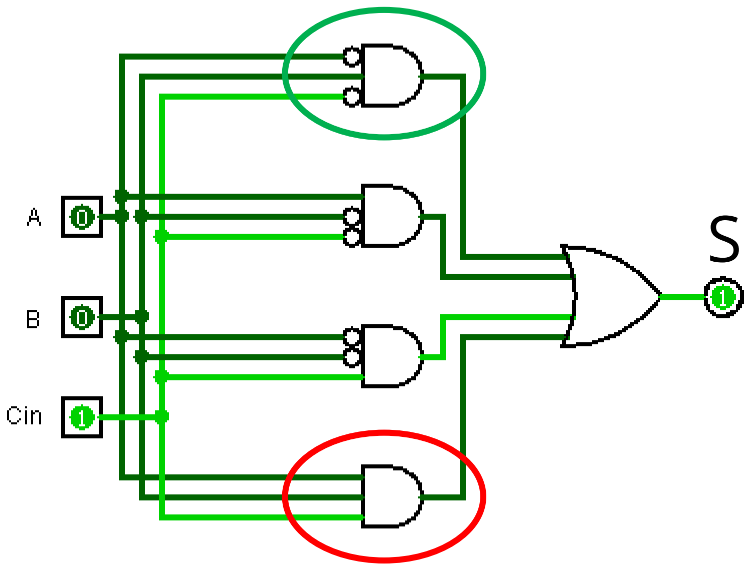
- It looks a little messy, but it kinda makes sense if you think of it like this:
  - It **counts how many input bits are "1"**
  - C_o and S are a **2-bit number!**
- If we look at the outputs in isolation:
  - S is 1 if we have an **odd number of "1s"**
  - Co is 1 if we have **2 or 3 "1s"**
- It's a little weird, but we can build this out of AND, OR, and XOR gates

C_i	A	B	C_o	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Boolean Expression

$$S = \overline{A}B\overline{C}_i + A\overline{BC_i} + \overline{AB}C_i + ABC_i$$

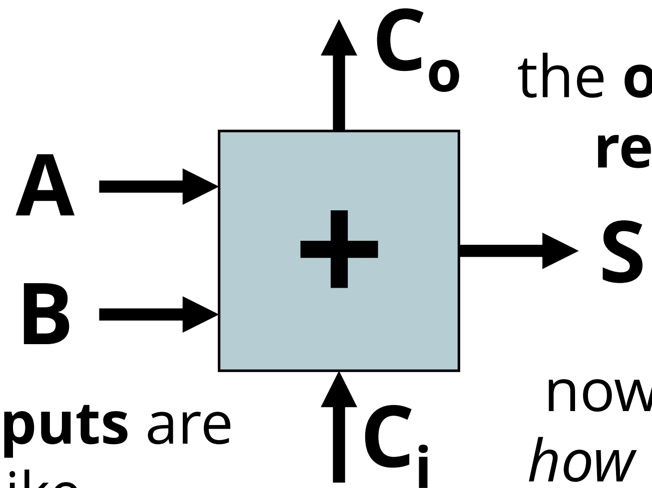
$$C_o = AB\overline{C}_i + A\overline{B}C_i + \overline{A}BC_i + ABC_i$$



C_i	A	B	C_o	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Sweeping it under the rug...

- In programming, we use **functions** to be able to reuse code.
- In hardware, we can group these 5 gates into a **component**.
- Here's the symbol for a **one-bit full adder**.



the **outputs** are like **return values**

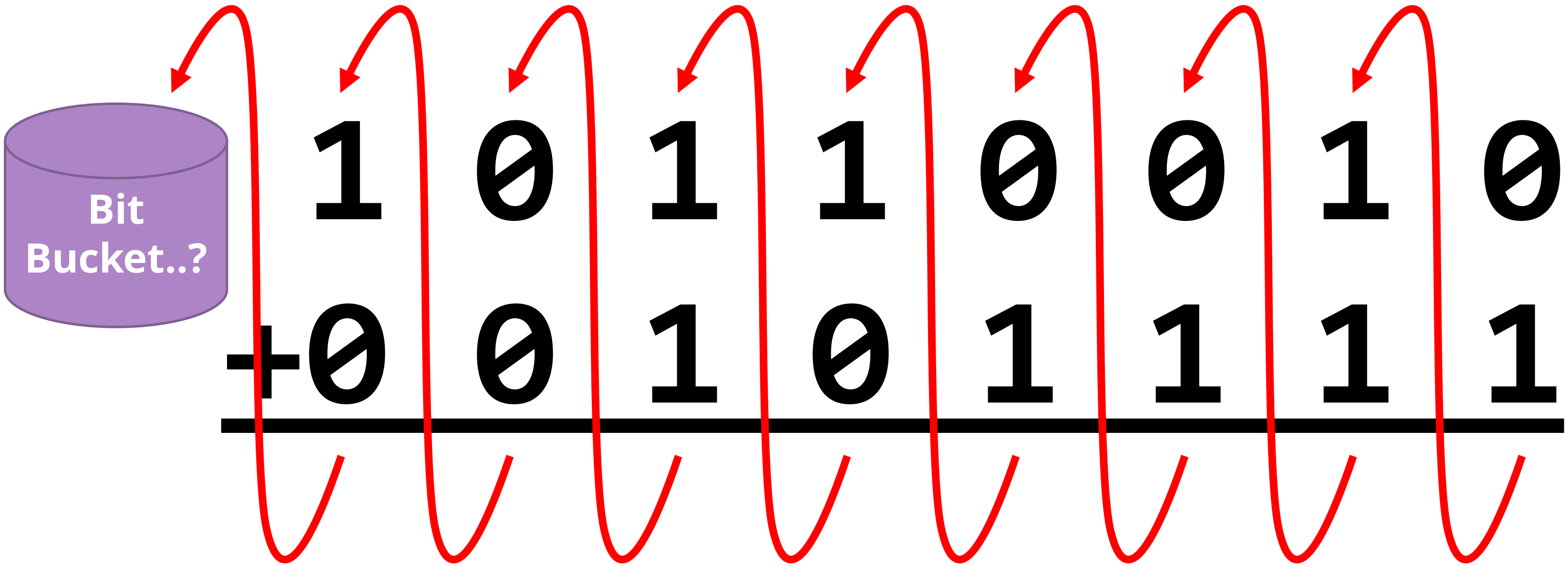
the **inputs** are like **parameters**

now we don't have to care *how* it adds, just that it *does*

Adding Longer Numbers

1-bit adders are cool and all, but, like, not very useful.

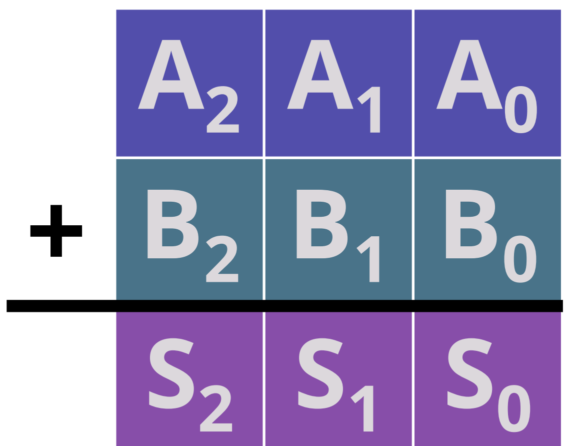
Adding Longer Numbers

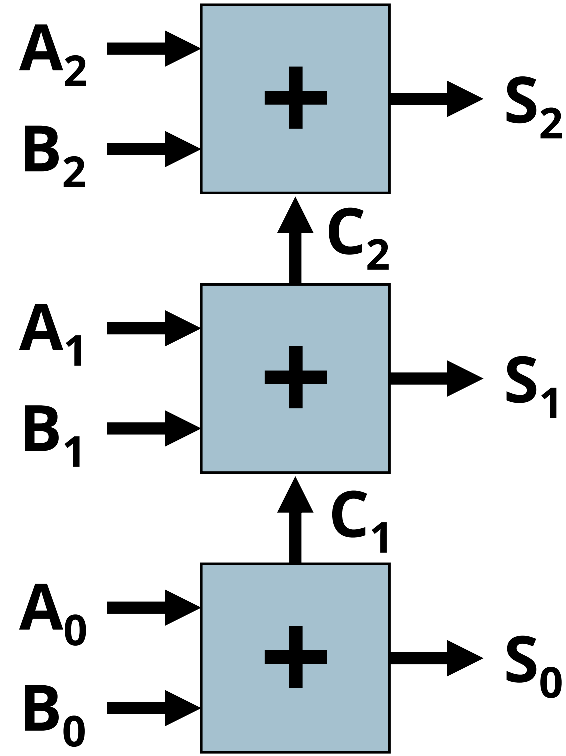


- The resulting bit for each digit is dependent on the two input bits and the carry.
- There are two outputs, however, a sum and a carry out.

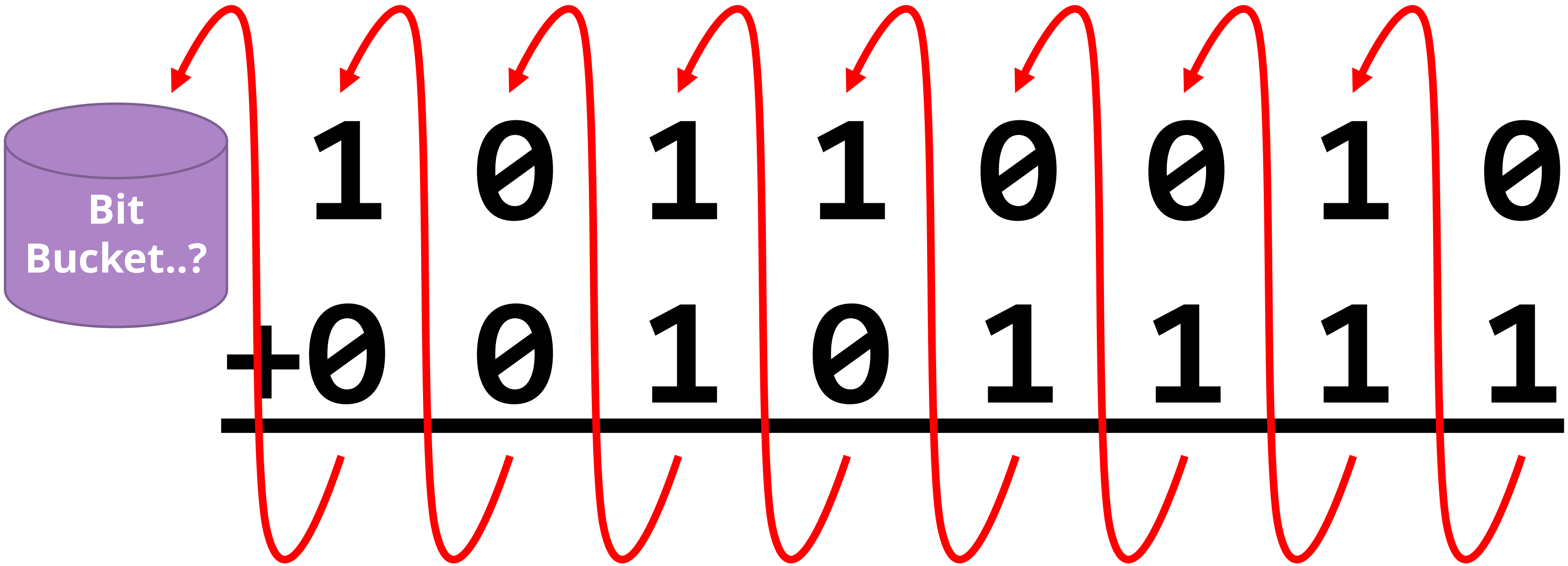
Ripple Carry

- If we want to add two **three-bit numbers,** we'll need three **one-bit** adders.
- We chain the carries from each place to the **next higher place,** like we do on paper.
- We have to split the numbers up like so:



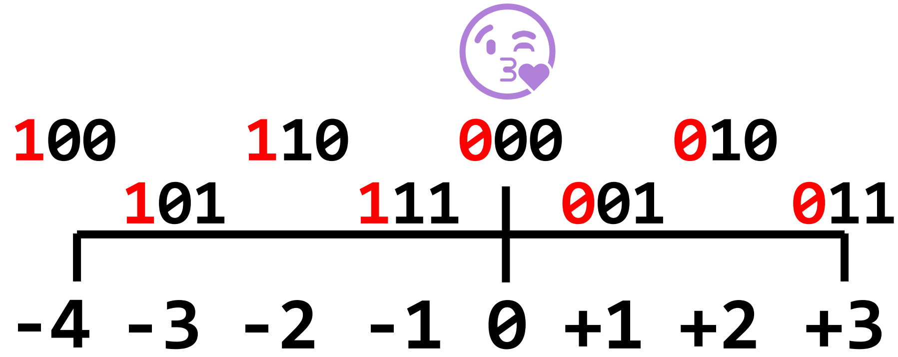


Negative Numbers



- That first number… is negative… hmm
- It works JUST FINE. It's really neat actually…

Recall 2's Complement



2's Complement

- Let's add them…

```
 1011 0010 : -78
+0010 1111 : +47
 1110 0001 : -31
```

Two’s Complement Addition

- The great thing is: you can add numbers of either sign **without having to do anything special!**

to binary?

```
       1
 3     0011
+10   +1010
 13    1101
```

bit pattern for -6... positive 6?

```
 3      0110
+-6     1001
 -3
```

flip!

+1

```
    1
  0011
 +1010
  1101
```

this is negative, so what is it? flip!

```
0010
```

+1

```
0011
```

to decimal?

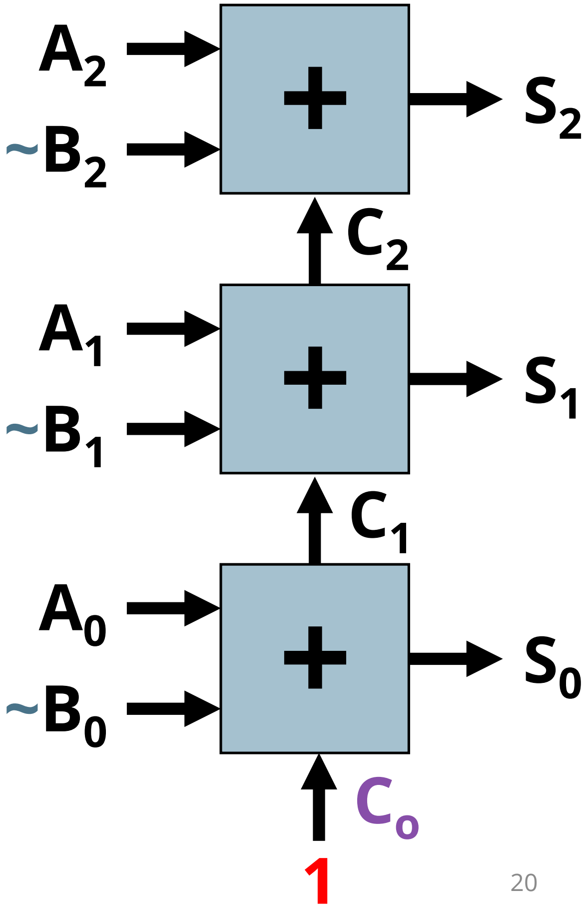
-3

the actual patterns of bits are the same.
so how does the computer "know" whether it's doing signed or unsigned addition?

IT DOESN'T

What.. Even.. Is.. Subtraction?

- We *could* come up with a separate subtraction circuit, but…
- Algebra tells us that **x - y** = **x + (-y)**
- Negation means **flip the bits** and **add 1**
- Flipping the bits uses NOT gates!
- How do we add 1 without any extra circuitry?
  - We use a *full adder for the LSB,* and when we're subtracting, set the "**carry in**" to **1**.



Overflow

Revisiting What Happens When We Run Out of Dang Space

How many bits?

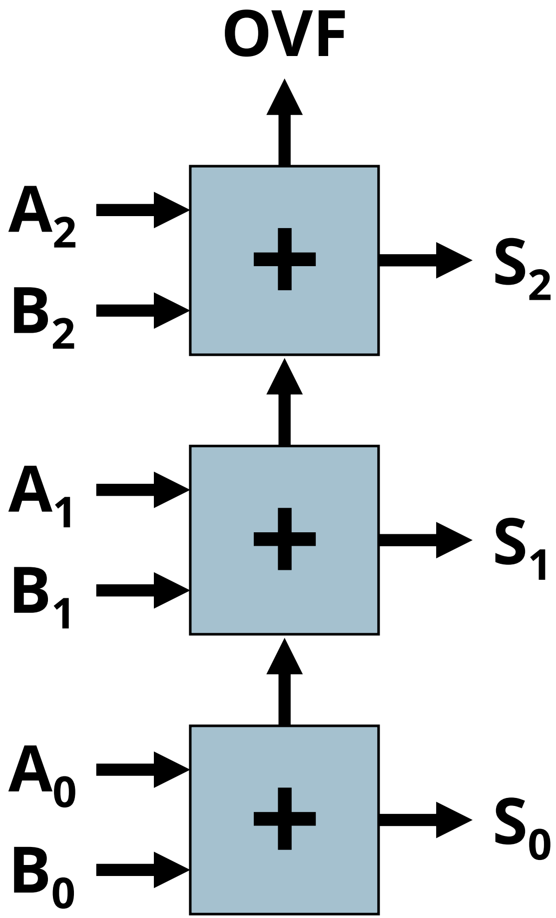
- If you add two 2-digit decimal numbers, what's the **largest number** you can get?
- What about two 4-digit decimal numbers?
- What about two 4-*bit* numbers?
- What's the pattern of the **number of digits?**
  - If you **add** two *n*-digit numbers *in any base*…
  - The result will have **at most *n* + 1 digits**
- That means if we add two **32-bit** numbers…
  - …we might get a **33-bit** result!
  - (It's the 32 S output bits, and the last carry-out bit)
  - if we have more bits than we can store in our number, that's **overflow.**

```
  99   9999
 +99  +9999
 198  19998

  1111
 +1111
 11110
```

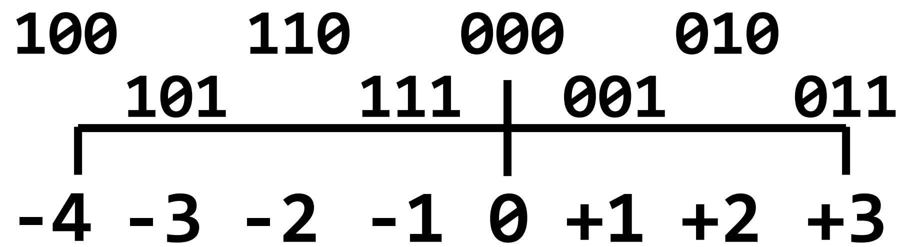
Detecting Overflow

- For *unsigned* addition, it's easy
- For an n-bit adder:
  - just look at the $\mathbf{C_o}$ of the **MSB**
  - if it's 1, **it's an overflow.**
- What about subtraction?
  - shhh



Hmm, negative numbers

- The signed number line looks like this:
- Overflow occurs if we go off **either end of the number line**
- In **unsigned** 3-bit arithmetic…
  - $111_2 + 001_2 = 1000_2$
    - that is, 7 + 1 = 8, which is **too big.**
- But in **signed** 3-bit arithmetic…
  - $111_2 + 001_2 = 0$
    - because -1 + 1 = 0!
- Same bit patterns, but **different results.**
  - Detecting signed overflow is a bit more subtle…



Detecting Signed Overflow

- If you add two numbers of **different signs** (negative and positive)…
  - Is it possible to **go off the ends** of the number line?
    - **no!**
    - That always gets you closer to 0
- If you add two numbers of **the same sign,** how do you know if you **went off the end of the number line?**
  - If you add two positive numbers and get a **negative number.**
  - If you add two negative numbers and get a **positive number.**
- How do you check the signs of the three numbers?
  - Do you think you could come up with a truth table? ;)

Handling overflow

- We could ***ignore it***
  - In MIPS: **addu, subu**
  - This is usually a **bad idea**
    - Your program is broken
  - It's also the default in most languages, thanks C
- We could fall on the **floor *- i.e. crash***
  - In MIPS **add, sub**
  - Can be handled and recovered from
  - But more complex
- We could ***store*** *that 33rd bit* ***somewhere else***

Maybe the bit bucket is a real place...

- Many *other* architectures do store this final 33rd (etc) bit.
  - MIPS does not.
- They have a "carry bit" register
  - This can be checked by the program after an add/sub
- This is very useful for **arbitrary precision arithmetic**
  - If you want to add 2048-bit numbers, chain many 32-bit additions!