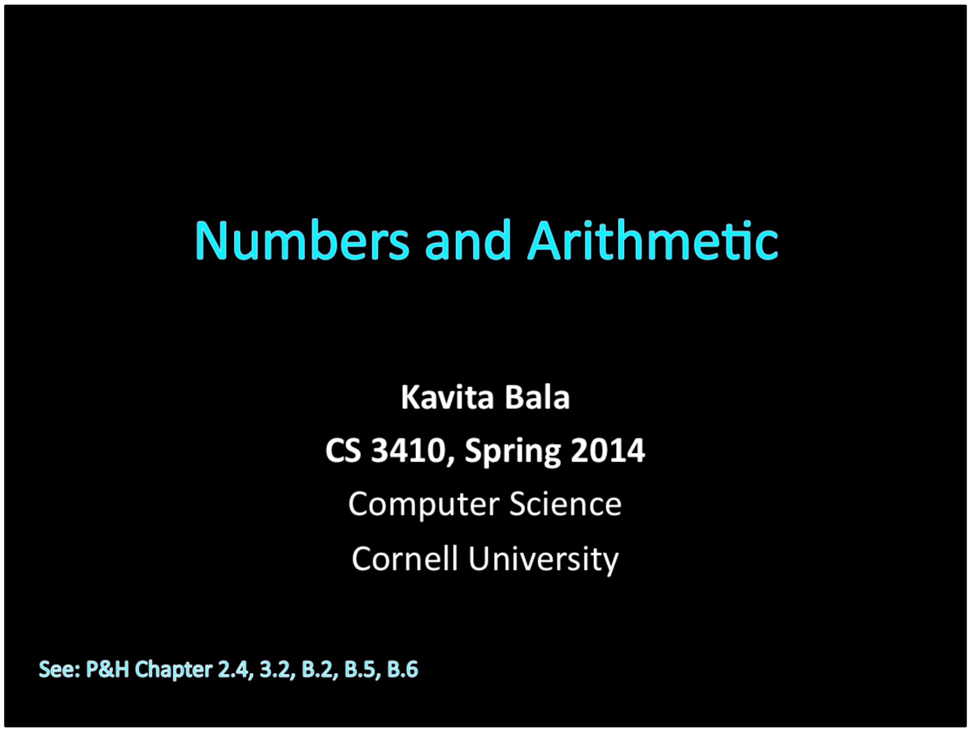


Numbers and Arithmetic

Kavita Bala

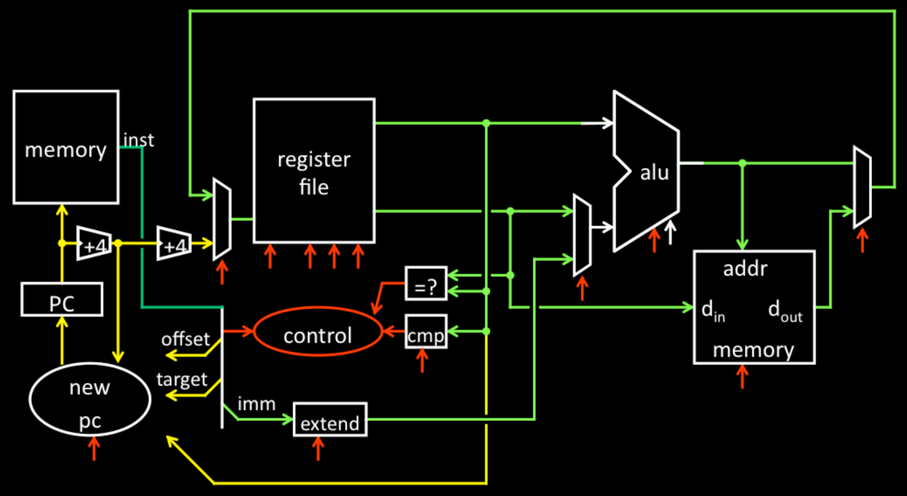
CS 3410, Spring 2014

Computer Science

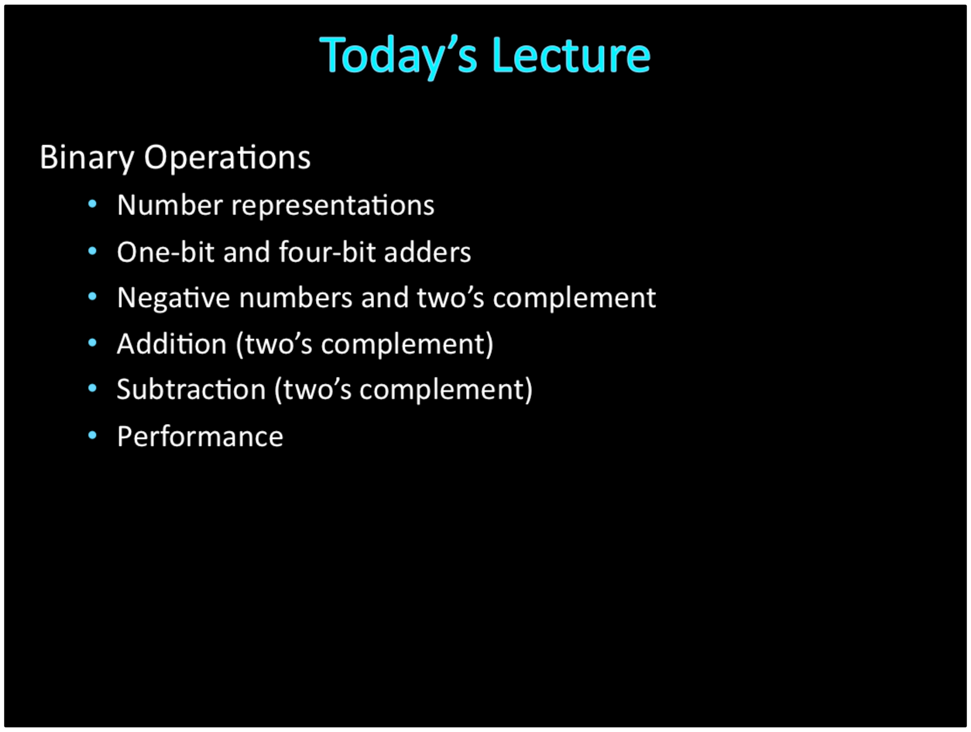
Cornell University

See: P&H Chapter 2.4, 3.2, B.2, B.5, B.6

Big Picture: Building a Processor



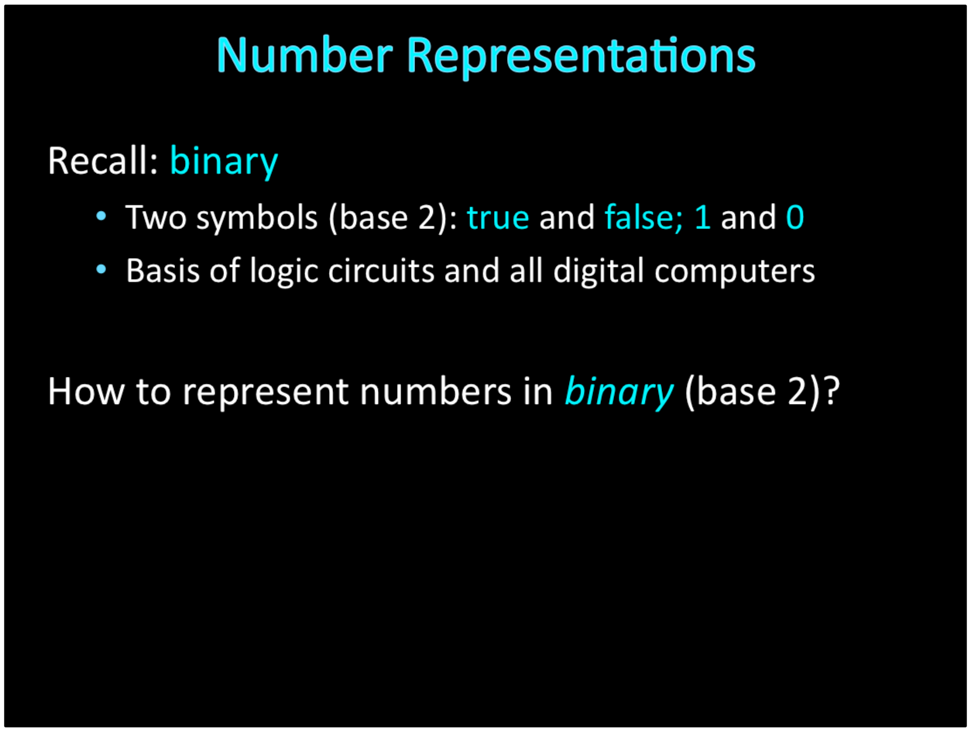
A single cycle processor



Today's Lecture

Binary Operations

- Number representations
- One-bit and four-bit adders
- Negative numbers and two's complement
- Addition (two's complement)
- Subtraction (two's complement)
- Performance

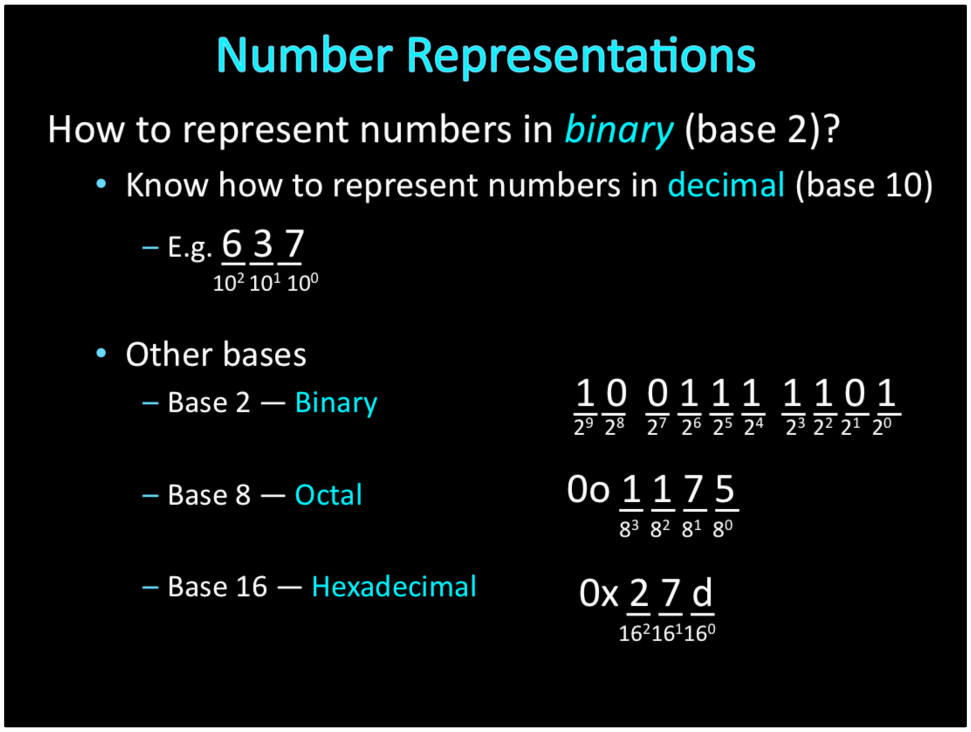


Number Representations

Recall: binary

- Two symbols (base 2): true and false; 1 and 0
- Basis of logic circuits and all digital computers

How to represent numbers in *binary* (base 2)?



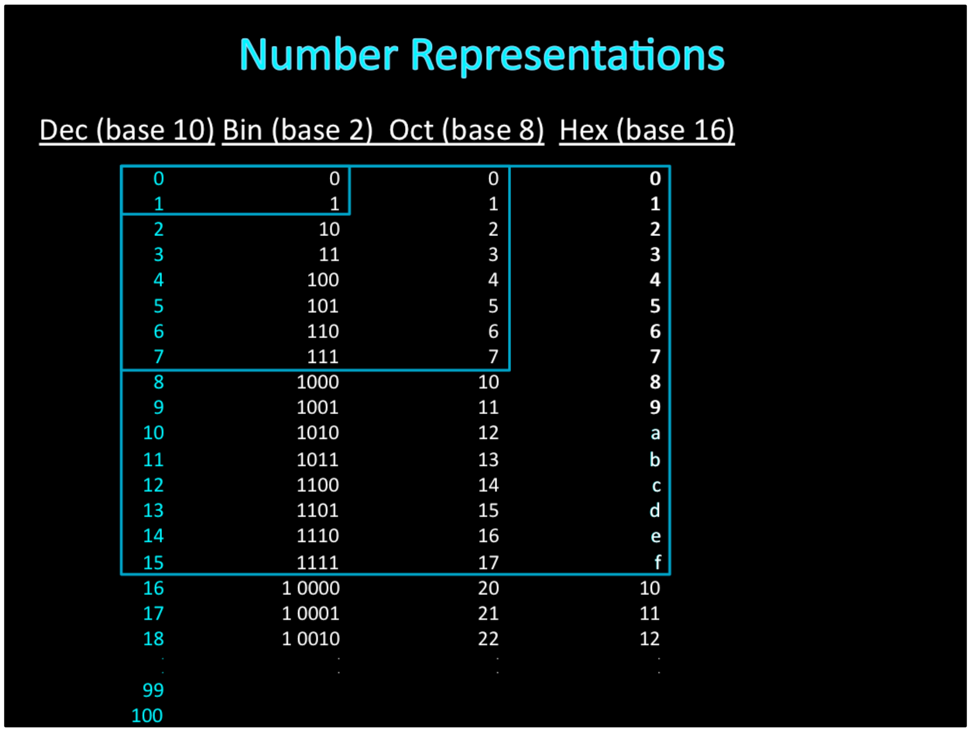
Number Representations

How to represent numbers in *binary* (base 2)?

- Know how to represent numbers in decimal (base 10)
  - E.g. $\underset{10^2}{\underline{6}}\,\underset{10^1}{\underline{3}}\,\underset{10^0}{\underline{7}}$
- Other bases
  - Base 2 — Binary $\underset{2^9}{\underline{1}}\,\underset{2^8}{\underline{0}}\;\underset{2^7}{\underline{0}}\,\underset{2^6}{\underline{1}}\,\underset{2^5}{\underline{1}}\,\underset{2^4}{\underline{1}}\;\underset{2^3}{\underline{1}}\,\underset{2^2}{\underline{1}}\,\underset{2^1}{\underline{0}}\,\underset{2^0}{\underline{1}}$
  - Base 8 — Octal $\text{0o}\,\underset{8^3}{\underline{1}}\,\underset{8^2}{\underline{1}}\,\underset{8^1}{\underline{7}}\,\underset{8^0}{\underline{5}}$
  - Base 16 — Hexadecimal $\text{0x}\,\underset{16^2}{\underline{2}}\,\underset{16^1}{\underline{7}}\,\underset{16^0}{\underline{d}}$

637 has a 1’s place , 10’s place and 100’s place
Show how to go from

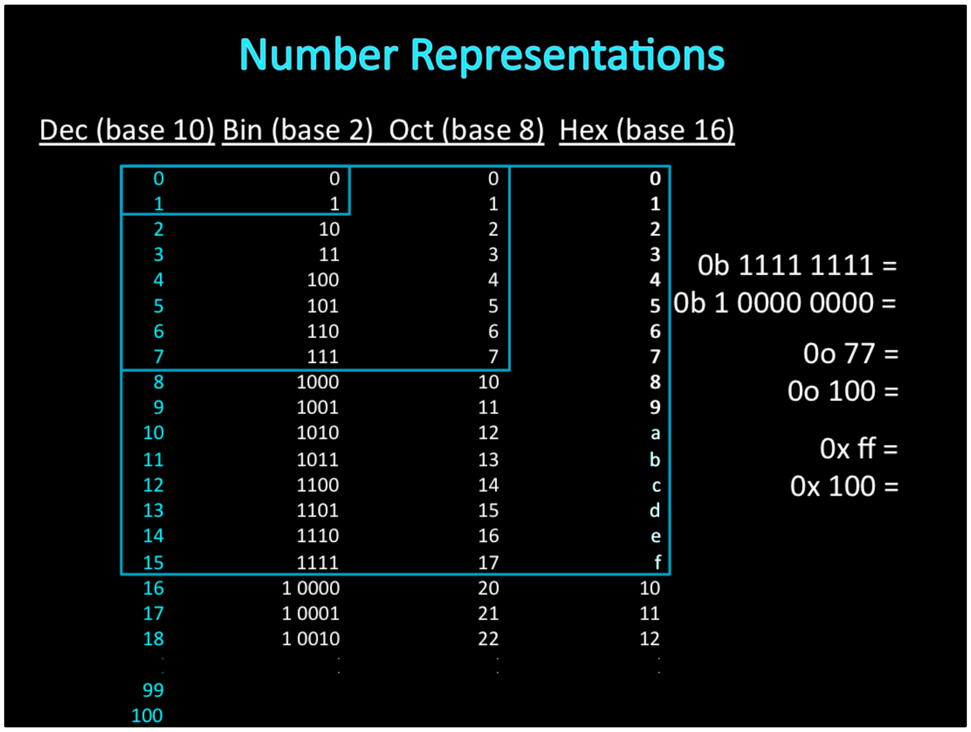
512+64+32+16+8+4+1
512+64+56+5
2*256+7*16+ 13 = 512+112+13



Number Representations

Dec (base 10)	Bin (base 2)	Oct (base 8)	Hex (base 16)
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	a
11	1011	13	b
12	1100	14	c
13	1101	15	d
14	1110	16	e
15	1111	17	f
16	1 0000	20	10
17	1 0001	21	11
18	1 0010	22	12
.	.	.	.
99			
100			

The base represents the number of *unique* symbols (decimal has 10, octal has 8, hexadecimal has 16, and binary has 2 unique symbols)
Every group of four bits is called a nibble, every group of 8 bits is a byte



Number Representations

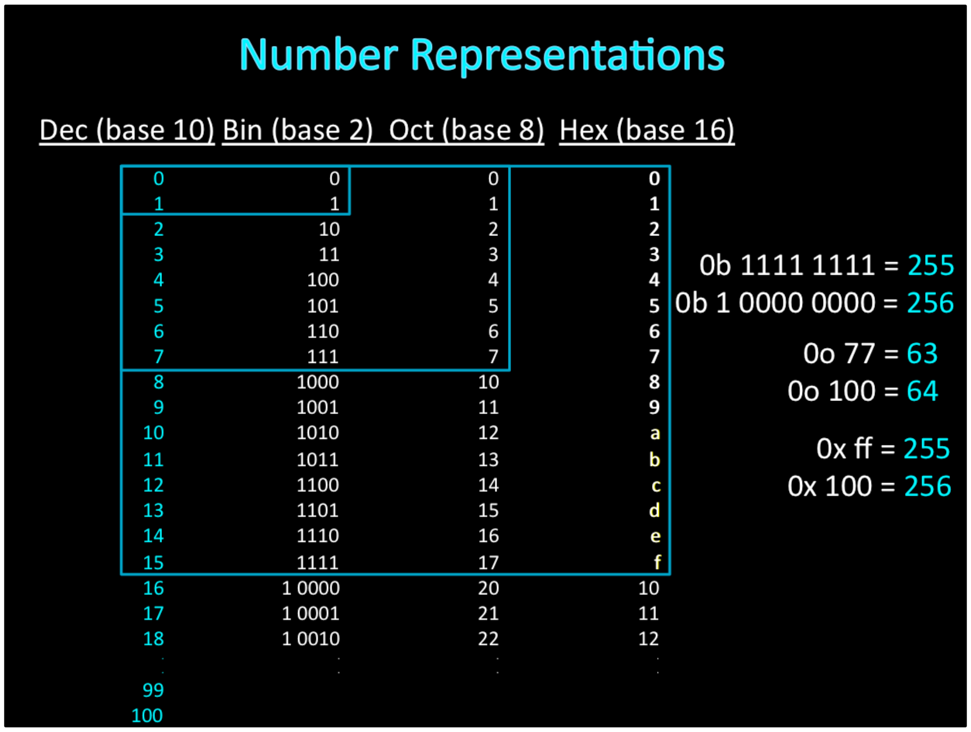
Dec (base 10)	Bin (base 2)	Oct (base 8)	Hex (base 16)
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	a
11	1011	13	b
12	1100	14	c
13	1101	15	d
14	1110	16	e
15	1111	17	f
16	1 0000	20	10
17	1 0001	21	11
18	1 0010	22	12
.	.	.	.
99			
100			

0b 1111 1111 =
0b 1 0000 0000 =

0o 77 =
0o 100 =

0x ff =
0x 100 =

The base represents the number of *unique* symbols (decimal has 10, octal has 8, hexadecimal has 16, and binary has 2 unique symbols)
Every group of four bits is called a nibble, every group of 8 bits is a byte



Number Representations

Dec (base 10)	Bin (base 2)	Oct (base 8)	Hex (base 16)
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	a
11	1011	13	b
12	1100	14	c
13	1101	15	d
14	1110	16	e
15	1111	17	f
16	1 0000	20	10
17	1 0001	21	11
18	1 0010	22	12
.	.	.	.
99			
100			

0b 1111 1111 = 255
0b 1 0000 0000 = 256

0o 77 = 63
0o 100 = 64

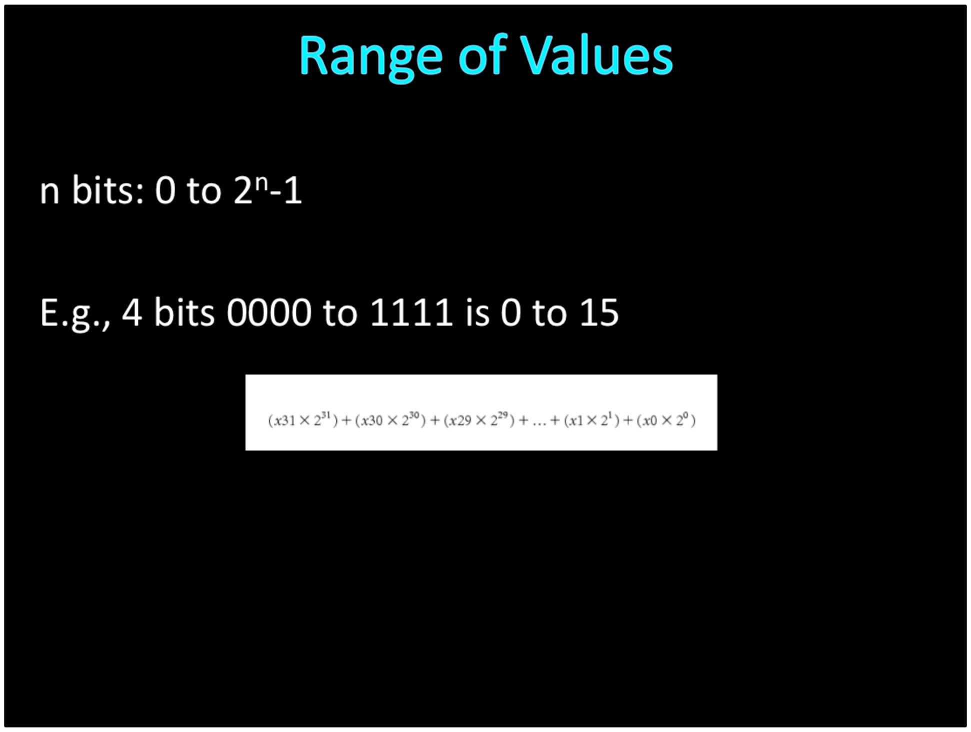
0x ff = 255
0x 100 = 256

The base represents the number of *unique* symbols (decimal has 10, octal has 8, hexadecimal has 16, and binary has 2 unique symbols)
Every group of four bits is called a nibble, every group of 8 bits is a byte

Convert to a different base instead of same base

637 = 0o1175 = 0b1001111101 = 0x27D
oct: 637 : 79 : 9 : 1 : 0
bin: 637 : 318 : 159 : 79 : 39 : 19 : 9 : 4 : 2 : 1 : 0
hex: 637 : 39 : 2 : 0

Ask students to write down binary number after we figure it out. The question is did they writ the
637 = 0o1175 = 0b10 0111 1101 = 0x27D
oct: 637 : 79 : 9 : 1 : 0
bin: 637 : 318 : 159 : 79 : 39 : 19 : 9 : 4 : 2 : 1 : 0
hex: 637 : 39 : 2 : 0



Range of Values

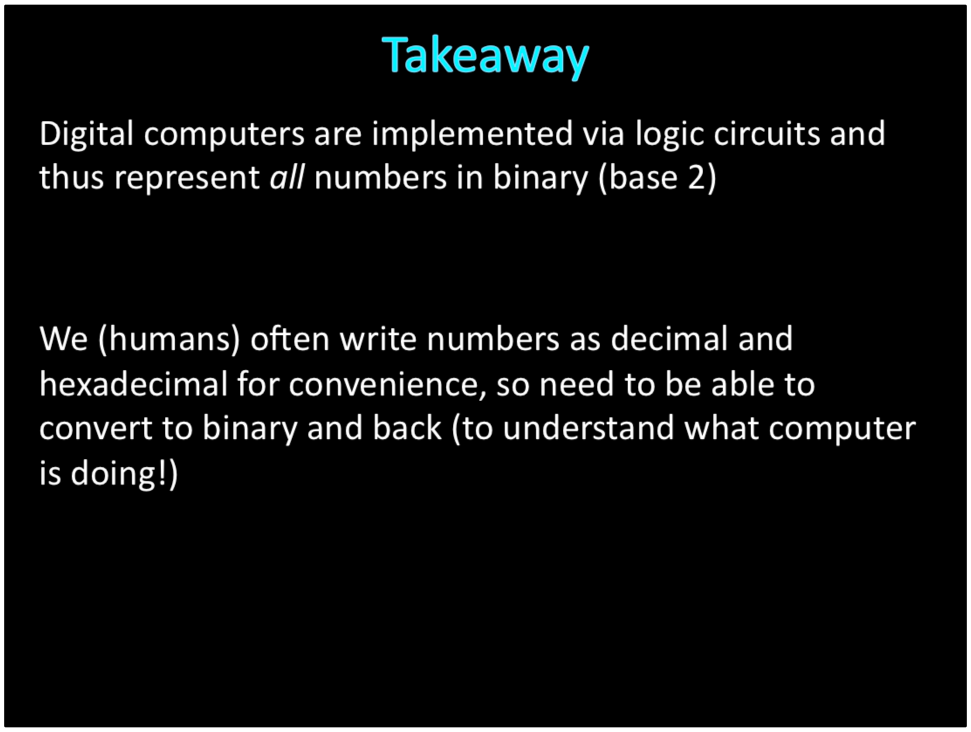
n bits: 0 to 2^n-1

E.g., 4 bits 0000 to 1111 is 0 to 15

$$(x31 \times 2^{31}) + (x30 \times 2^{30}) + (x29 \times 2^{29}) + \ldots + (x1 \times 2^{1}) + (x0 \times 2^{0})$$

2^3 + 2^2 +2^1 + 1 = 2^4-1
2^4 = 2^3 + 2^2 + 2^1 + 2^1 = 2^3 + 2^2 + 2 times 2

Ask class to convert from binary to octal
637 = 0o1175 = 0b10 0111 1101 = 0x27D
oct: 637 : 79 : 9 : 1 : 0
bin: 637 : 318 : 159 : 79 : 39 : 19 : 9 : 4 : 2 : 1 : 0
hex: 637 : 39 : 2 : 0

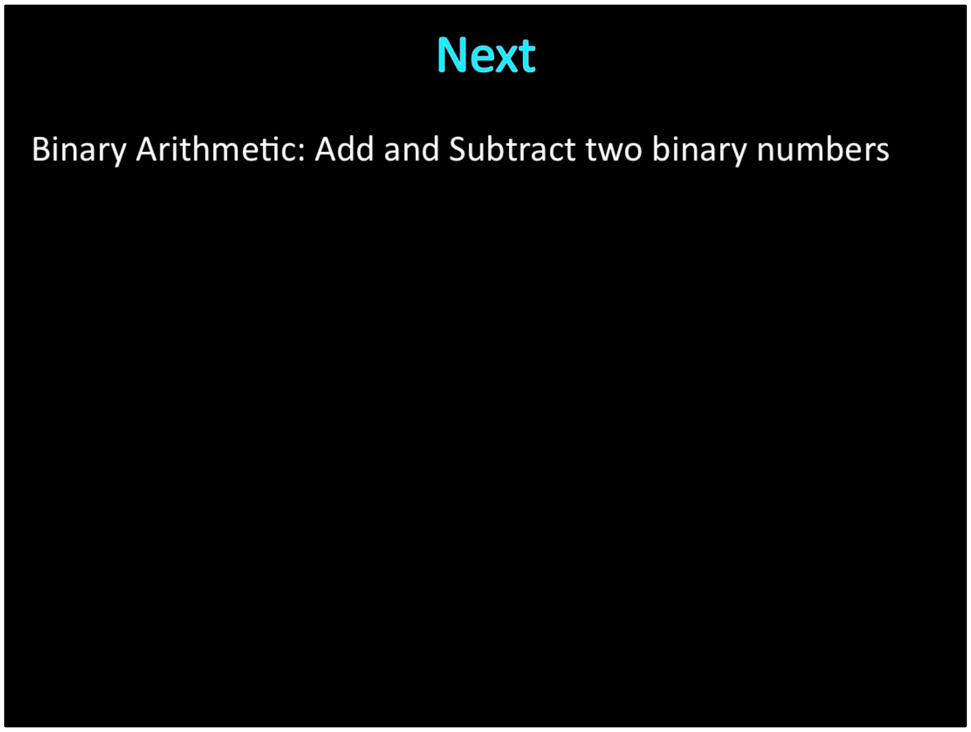


Takeaway

Digital computers are implemented via logic circuits and thus represent *all* numbers in binary (base 2)

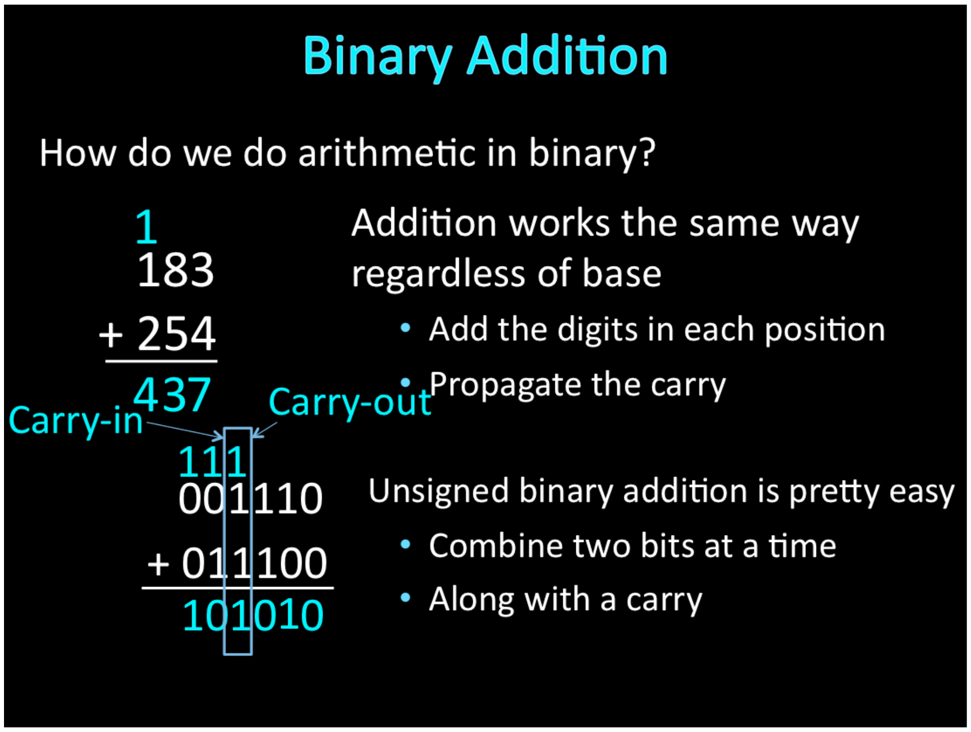
We (humans) often write numbers as decimal and hexadecimal for convenience, so need to be able to convert to binary and back (to understand what computer is doing!)

We can write any number in any base we like. The most natural base for computers is binary, which is hard to read, which is the reason we use hex and octal.



Next

Binary Arithmetic: Add and Subtract two binary numbers



Binary Addition

How do we do arithmetic in binary?

Addition works the same way regardless of base

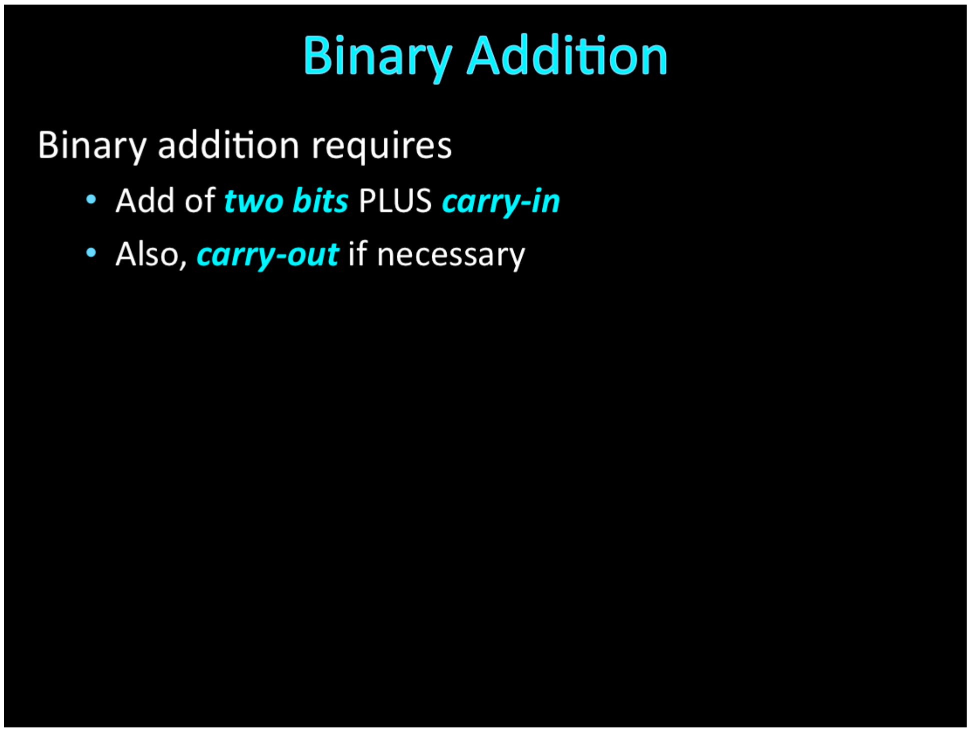
- Add the digits in each position
- Propagate the carry

Unsigned binary addition is pretty easy

- Combine two bits at a time
- Along with a carry

1
183
+ 254
437
Carry-in
Carry-out
111
001110
+ 011100
101010

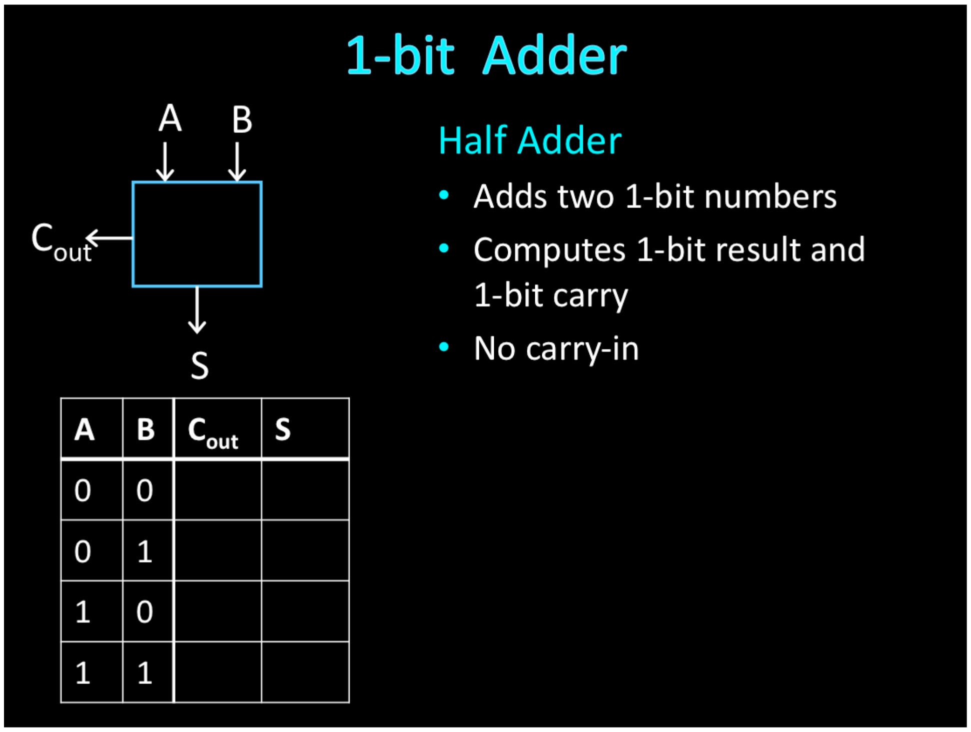
Talk about Cin (carry in) and Cout (carry out)
So we need two numbers, the sum, carry in, and carry out


Binary Addition

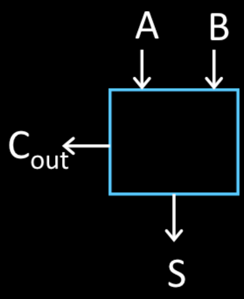
Binary addition requires

- Add of ***two bits*** PLUS ***carry-in***
- Also, ***carry-out*** if necessary

Talk about Cin (carry in) and Cout (carry out)
So we need two numbers, the sum, carry in, and carry out



1-bit Adder



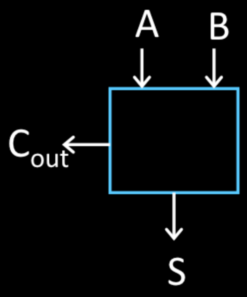
A	B	C_{out}	S
0	0		
0	1		
1	0		
1	1		

Half Adder

- Adds two 1-bit numbers
- Computes 1-bit result and 1-bit carry
- No carry-in

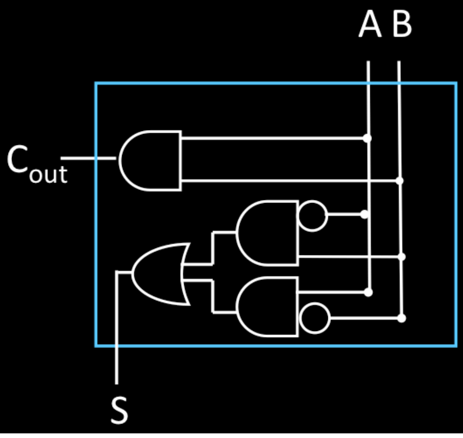
Adds two 1-bit numbers,
computes 1-bit result and carry out
Useful for the rightmost binary digit, not much else

1-bit Half Adder

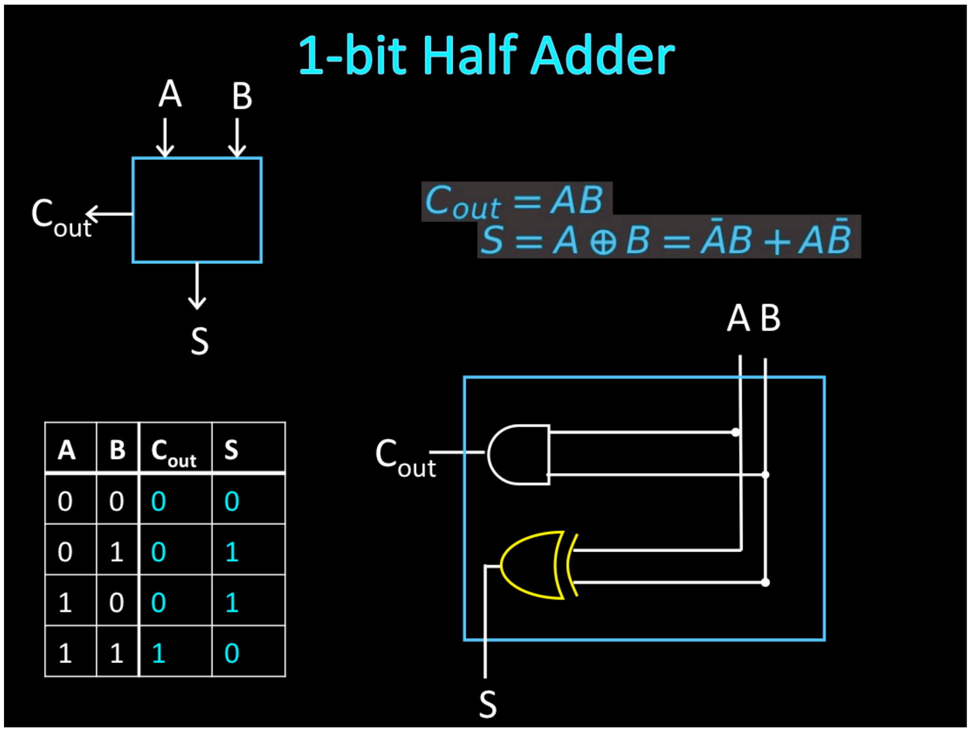


$$C_{out} = AB$$
$$S = A \oplus B = \bar{A}B + A\bar{B}$$

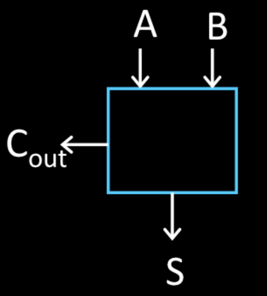
A	B	C_{out}	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0



Adds two 1-bit numbers,
computes 1-bit result and carry out
Useful for the rightmost binary digit, not much else

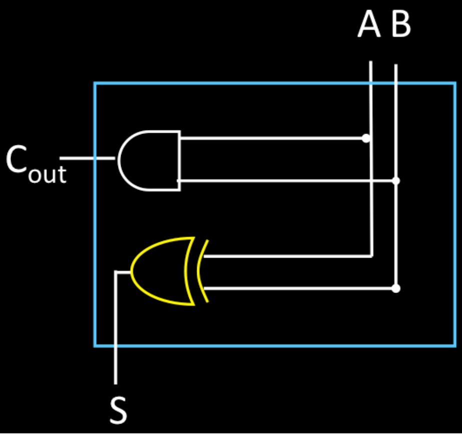


1-bit Half Adder

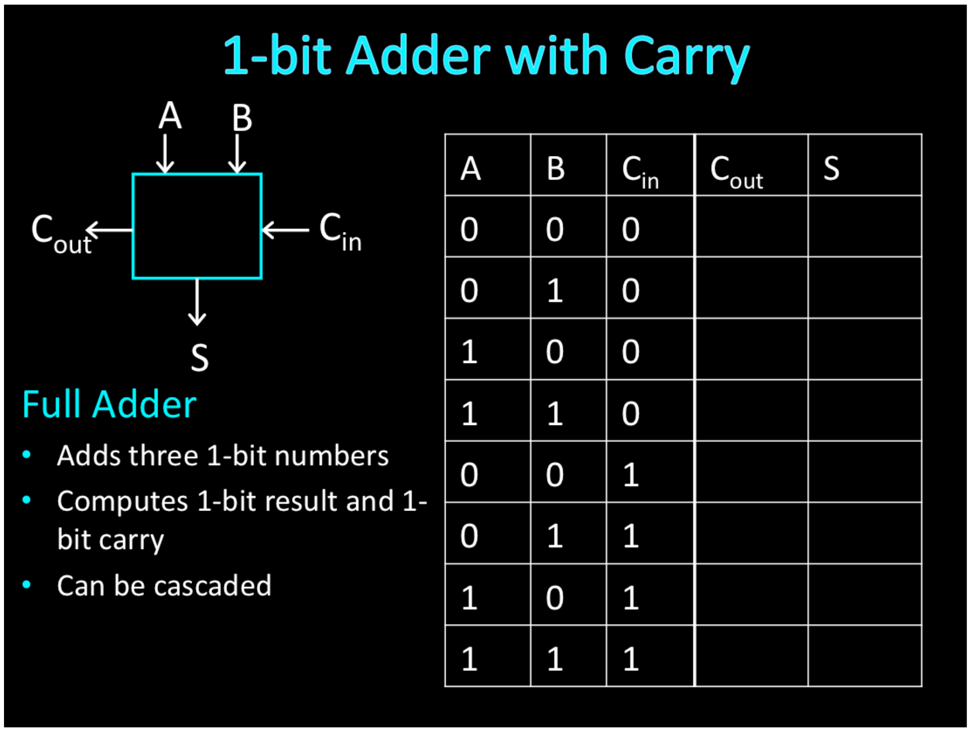


$$C_{out} = AB$$
$$S = A \oplus B = \bar{A}B + A\bar{B}$$

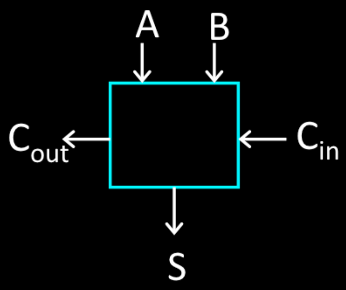
A	B	C_{out}	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0



Adds two 1-bit numbers,
computes 1-bit result and carry out
Useful for the rightmost binary digit, not much else



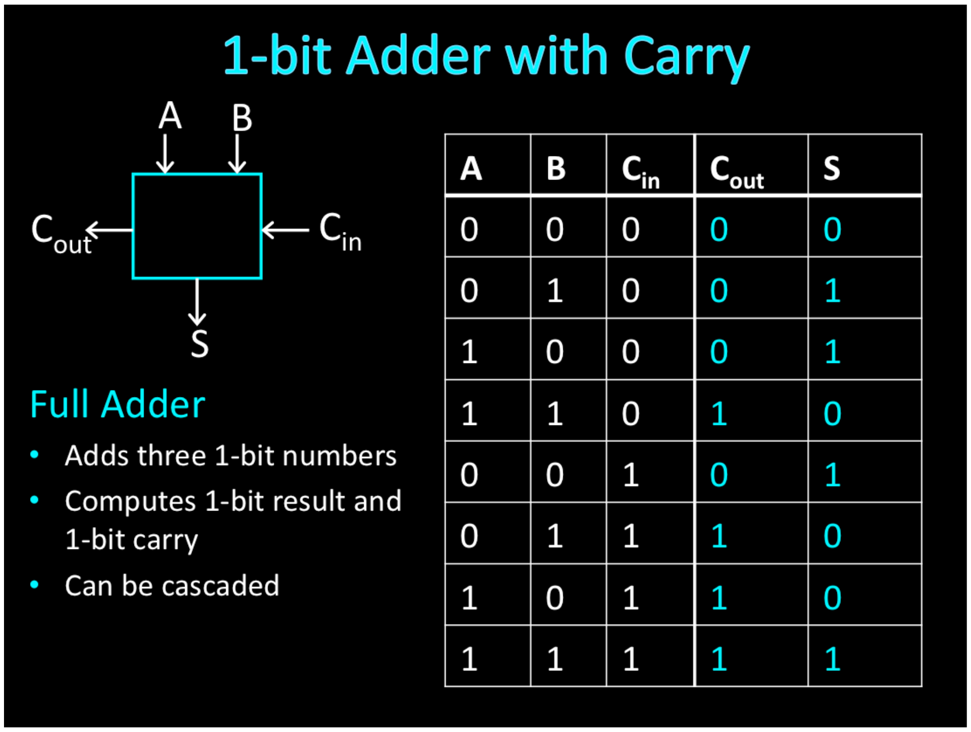
1-bit Adder with Carry



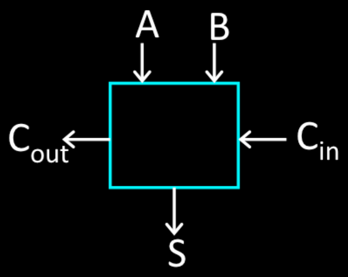
Full Adder

- Adds three 1-bit numbers
- Computes 1-bit result and 1-bit carry
- Can be cascaded

A	B	C_{in}	C_{out}	S
0	0	0		
0	1	0		
1	0	0		
1	1	0		
0	0	1		
0	1	1		
1	0	1		
1	1	1		



1-bit Adder with Carry

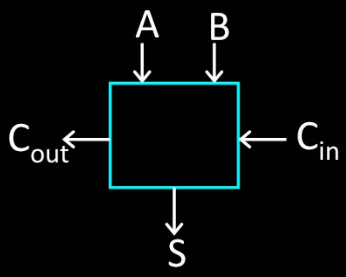


Full Adder

- Adds three 1-bit numbers
- Computes 1-bit result and 1-bit carry
- Can be cascaded

A	**B**	C_{in}	C_{out}	**S**
0	0	0	0	0
0	1	0	0	1
1	0	0	0	1
1	1	0	1	0
0	0	1	0	1
0	1	1	1	0
1	0	1	1	0
1	1	1	1	1

1-bit Adder with Carry

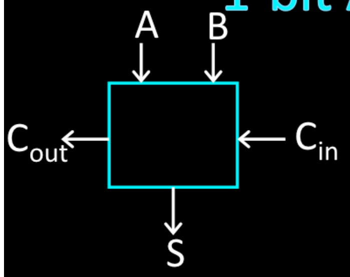


A	B	C_{in}	C_{out}	S
0	0	0	0	0
0	1	0	0	1
1	0	0	0	1
1	1	0	1	0
0	0	1	0	1
0	1	1	1	0
1	0	1	1	0
1	1	1	1	1

$$S = \overline{A}B\overline{C_{in}} + A\overline{B}\ \overline{C_{in}} + \overline{A}\ \overline{B}C_{in} + ABC_{in}$$

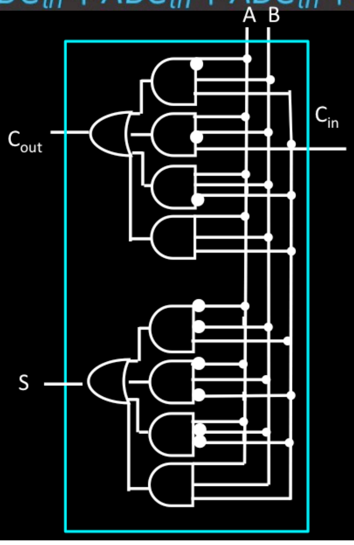
$$C_{out} = AB\overline{C_{in}} + \overline{A}BC_{in} + A\overline{B}C_{in} + ABC_{in}$$

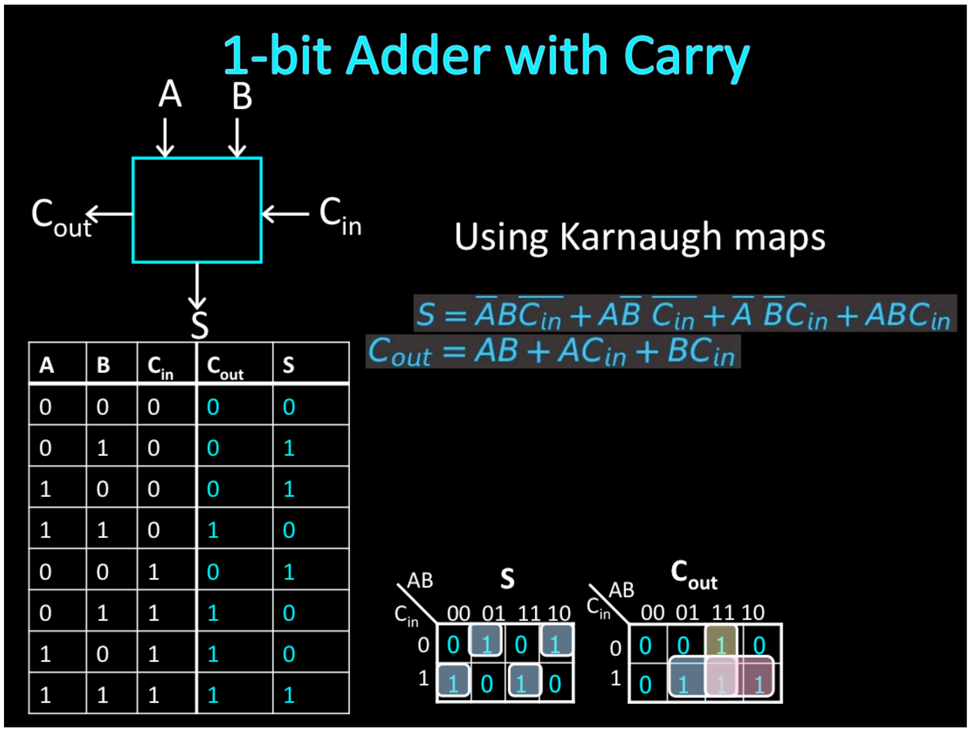
1-bit Adder with Carry



$$S = \overline{A}B\overline{C_{in}} + A\overline{B}\,\overline{C_{in}} + \overline{A}\,\overline{B}C_{in} + ABC_{in}$$
$$C_{out} = ABC_{in} + ABC_{in} + ABC_{in} + ABC_{in}$$

A	B	C_{in}	C_{out}	S
0	0	0	0	0
0	1	0	0	1
1	0	0	0	1
1	1	0	1	0
0	0	1	0	1
0	1	1	1	0
1	0	1	1	0
1	1	1	1	1




A
B
C_{out}
C_{in}
S

1-bit Adder with Carry

Using Karnaugh maps

$$S = \overline{A}B\overline{C_{in}} + A\overline{B}\,\overline{C_{in}} + \overline{A}\,\overline{B}C_{in} + ABC_{in}$$

$$C_{out} = AB + AC_{in} + BC_{in}$$

A	**B**	**C_{in}**	**C_{out}**	**S**
0	0	0	0	0
0	1	0	0	1
1	0	0	0	1
1	1	0	1	0
0	0	1	0	1
0	1	1	1	0
1	0	1	1	0
1	1	1	1	1

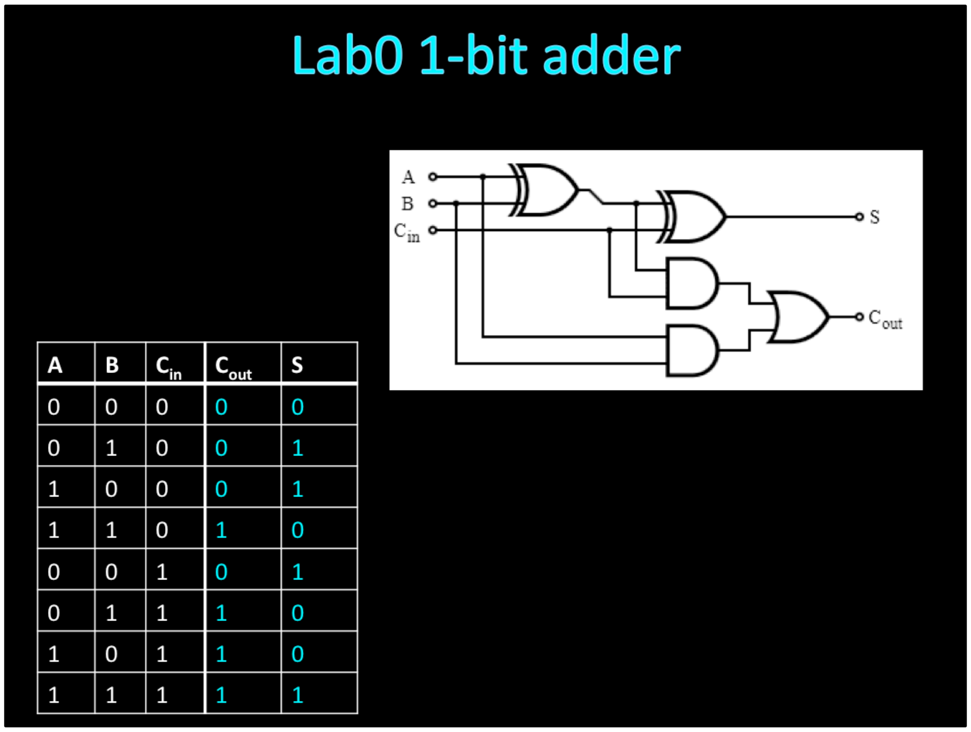
S

C_{in} \ AB	00	01	11	10
0	0	1	0	1
1	1	0	1	0

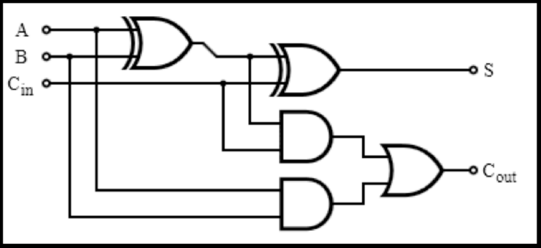
C_{out}

C_{in} \ AB	00	01	11	10
0	0	0	1	0
1	0	1	1	1

Can’t do much with S. But can optimize Cout.



Lab0 1-bit adder

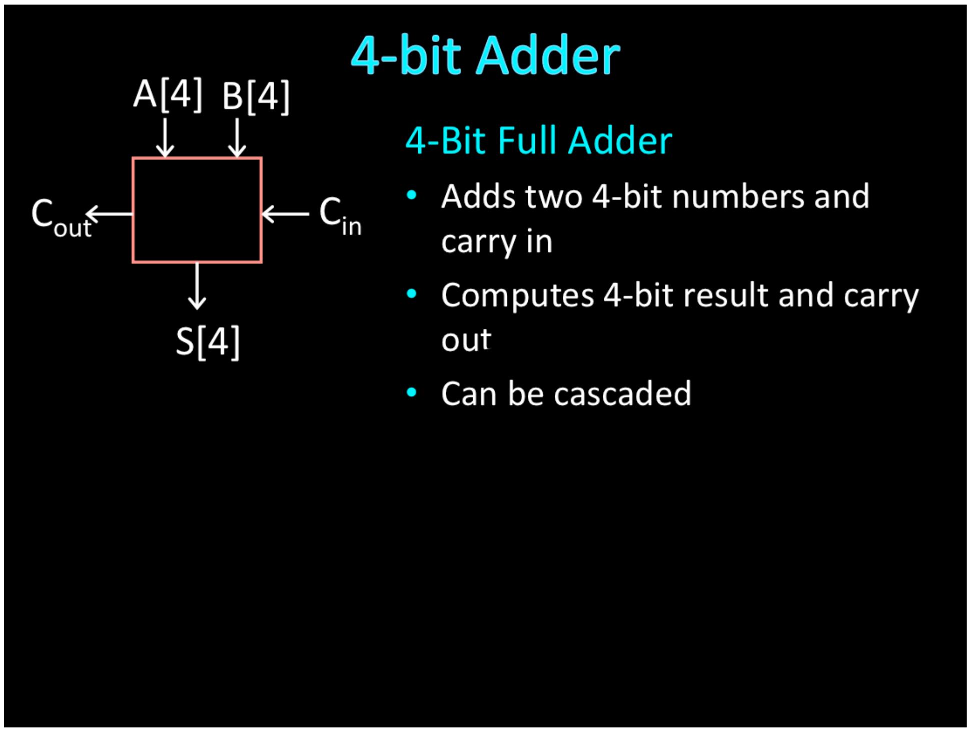


A	B	C_{in}	C_{out}	S
0	0	0	0	0
0	1	0	0	1
1	0	0	0	1
1	1	0	1	0
0	0	1	0	1
0	1	1	1	0
1	0	1	1	0
1	1	1	1	1

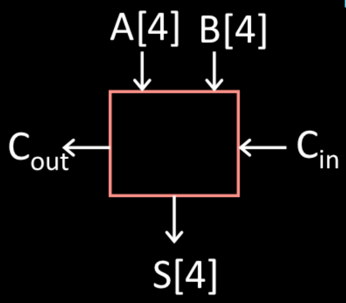
How do we get to our circuit from Lab. Using the ! notation to represent NOT here.

S = !A B !C + A !B !C + !A !B C + ABC = !A (B!C+!BC) + A (!B !C + BC) = !A (B xor C) + A !
(B xor C) = A xor (B xor C)
B xor C = !B C + B !C
! (B xor C) = !(!B C) !(B!C) = (B + !C) (!B + C) = !B!C + BC

C_out = A B !C + C (A xor B) + C A B = AB + C (A xor B)



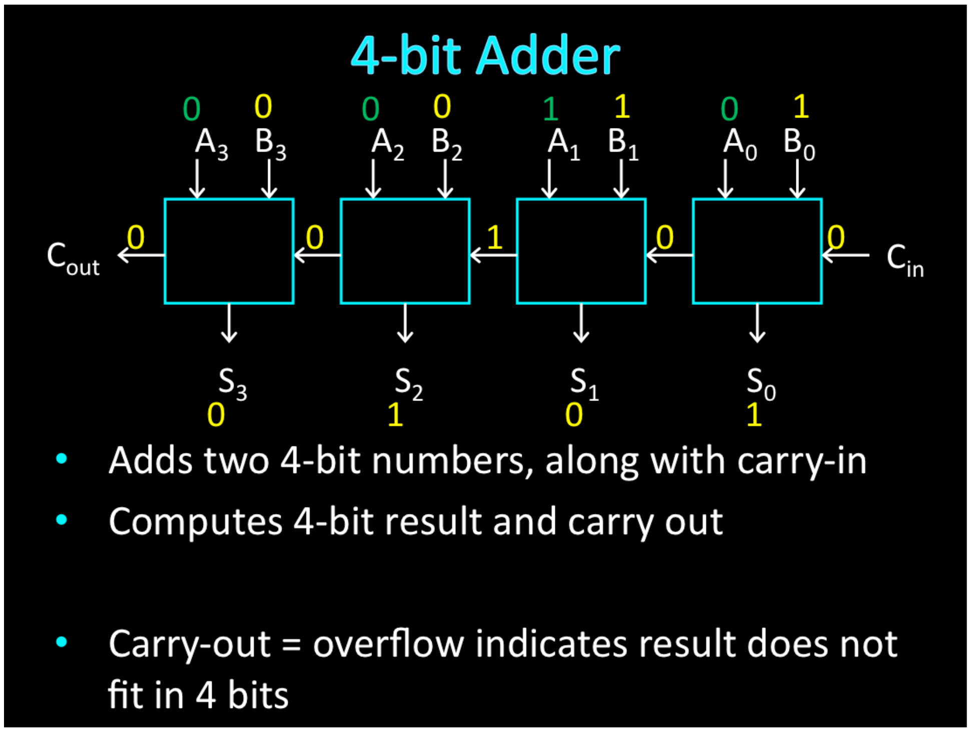
4-bit Adder



4-Bit Full Adder

- Adds two 4-bit numbers and carry in
- Computes 4-bit result and carry out
- Can be cascaded

Adds two 1-bit numbers, along with carry-in, computes 1-bit result and carry out
Can be cascaded to add N-bit numbers



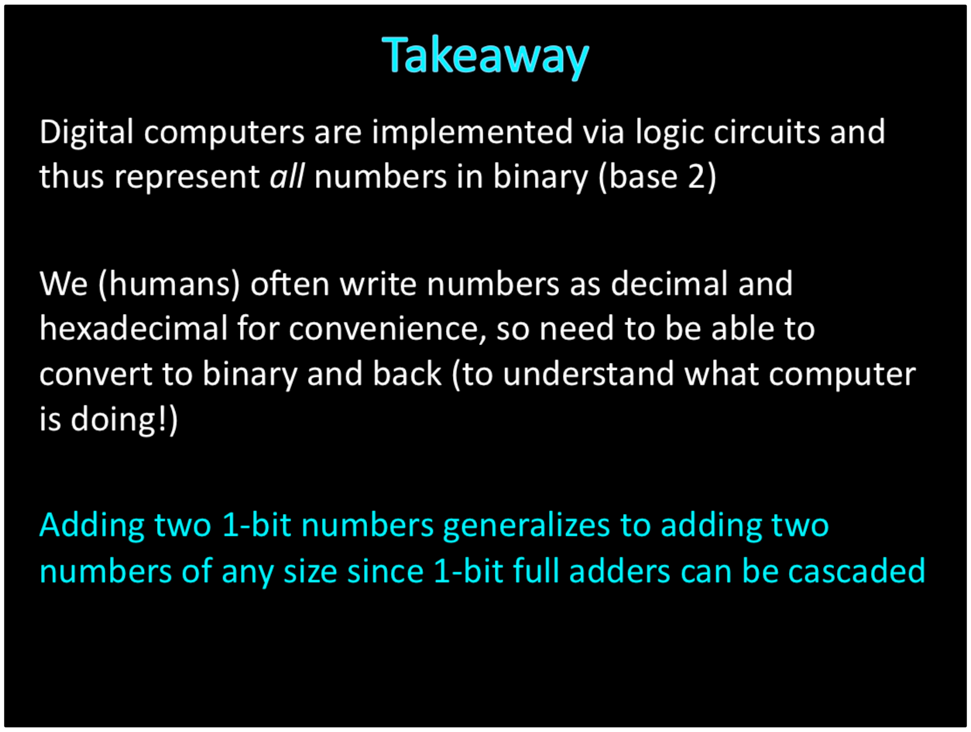
4-bit Adder

0 0 0 0 1 1 0 1
A3 B3 A2 B2 A1 B1 A0 B0
Cout 0 0 1 0 0 Cin
S3 S2 S1 S0
0 1 0 1

- Adds two 4-bit numbers, along with carry-in
- Computes 4-bit result and carry out

- Carry-out = overflow indicates result does not fit in 4 bits

Transition: to do subtraction, just add, but negate one number

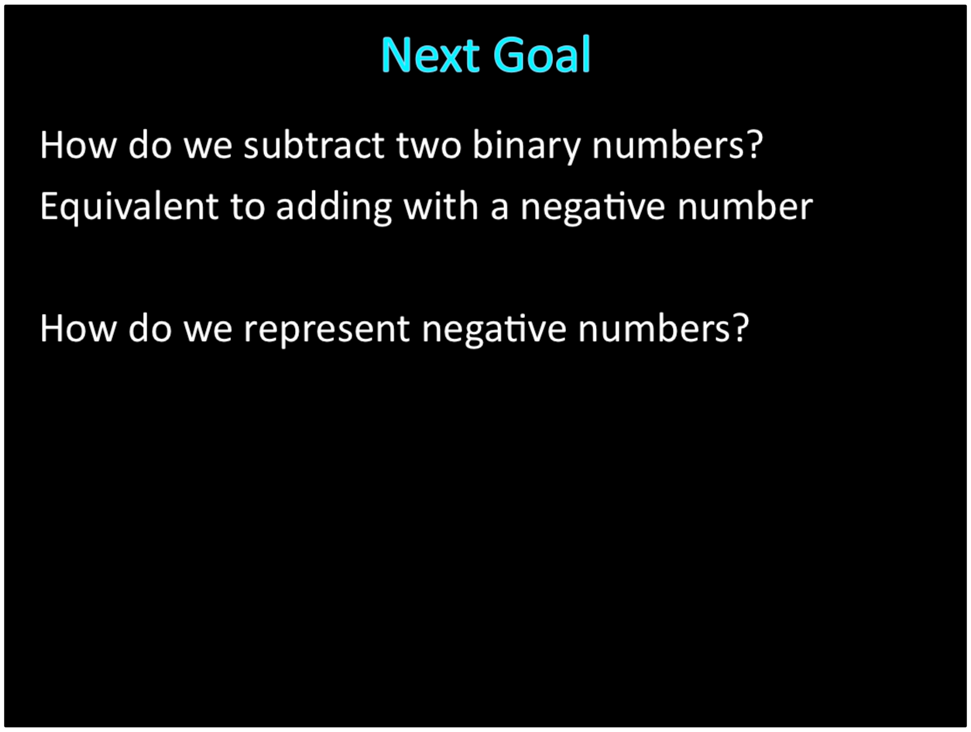


Takeaway

Digital computers are implemented via logic circuits and thus represent *all* numbers in binary (base 2)

We (humans) often write numbers as decimal and hexadecimal for convenience, so need to be able to convert to binary and back (to understand what computer is doing!)

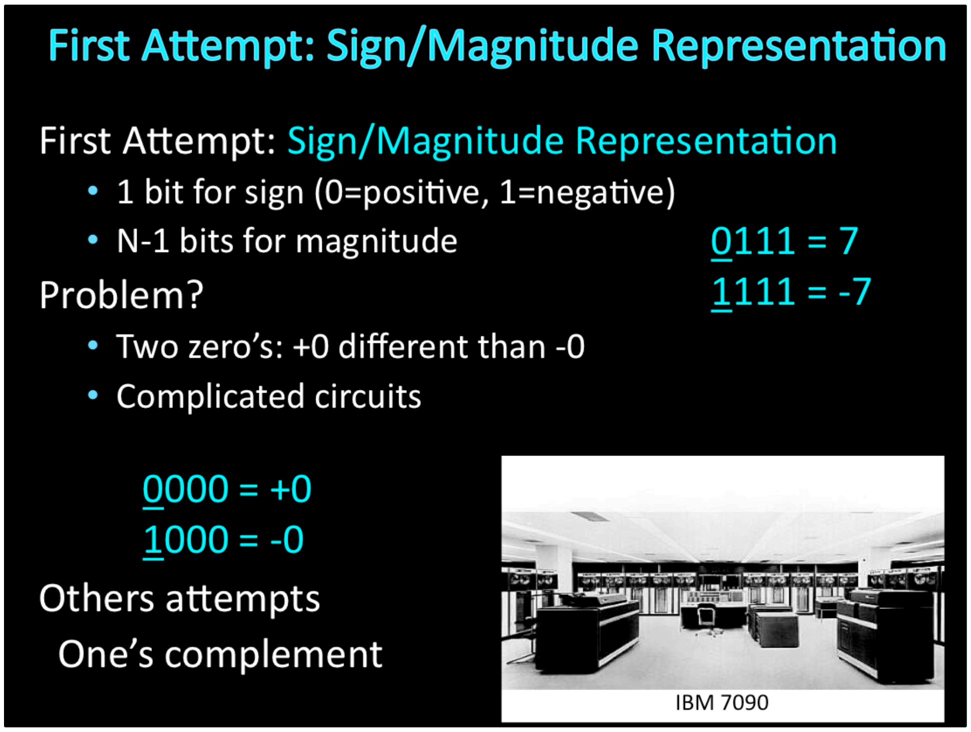
Adding two 1-bit numbers generalizes to adding two numbers of any size since 1-bit full adders can be cascaded



Next Goal

How do we subtract two binary numbers?
Equivalent to adding with a negative number

How do we represent negative numbers?



First Attempt: Sign/Magnitude Representation

First Attempt: Sign/Magnitude Representation

- 1 bit for sign (0=positive, 1=negative)
- N-1 bits for magnitude

$\underline{0}111 = 7$

$\underline{1}111 = -7$

Problem?

- Two zero's: +0 different than -0
- Complicated circuits

$\underline{0}000 = +0$

$\underline{1}000 = -0$

Others attempts

One's complement

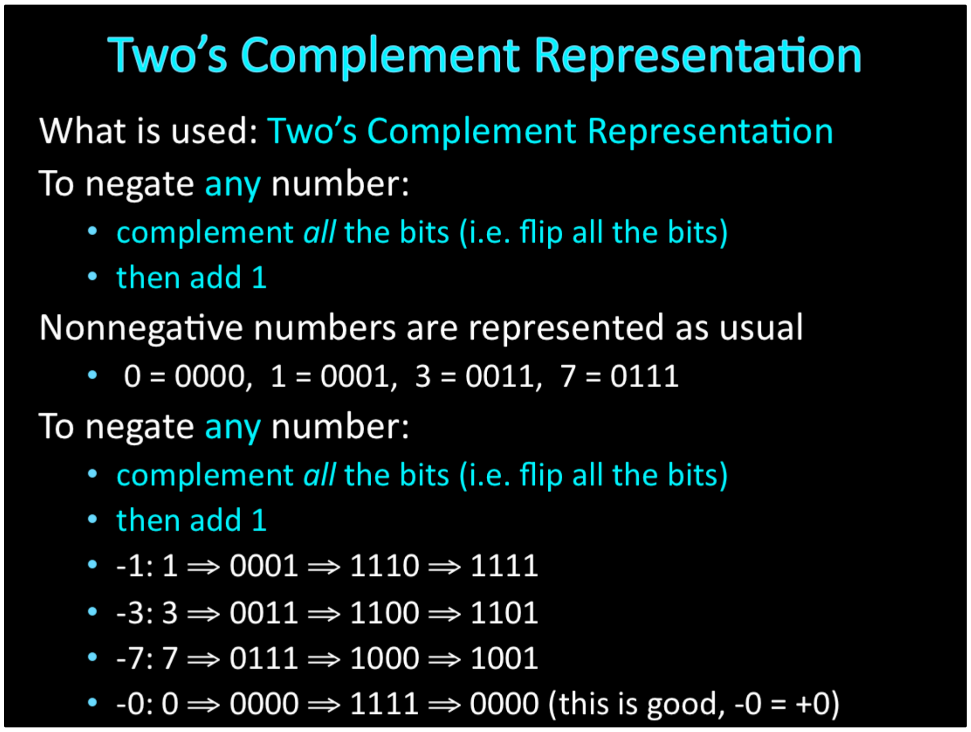
IBM 7090

problems? two zeros, circuit still complicated
e.g. the existence of two forms of the same value (-0 and +0) necessitates two rather than a single comparison when checking for equality with zero.

The CDC 6000 series and UNIVAC 1100 series computers were based on ones' complement.

Also, in addition to two zero's, there is a crazy phenomenon called "end-around carry“: If the carry extends past the end of the word it is said to have "wrapped" around, a condition called an "end-around carry". When this occurs, the bit must be added back in at the right-most bit. This phenomenon does not occur in two's complement arithmetic.
Subtraction is similar, except that borrows are propagated to the left instead of carries. If the borrow extends past the end of the word it is said to have "wrapped" around, a condition called an "end-around borrow". When this occurs, the bit must be subtracted back in at the right-most bit. This phenomenon does not occur in two's complement arithmetic.

The CDC 6000 series and UNIVAC 1100 series computers were based on ones' complement.



Two's Complement Representation

What is used: Two's Complement Representation

To negate any number:

- complement *all* the bits (i.e. flip all the bits)
- then add 1

Nonnegative numbers are represented as usual

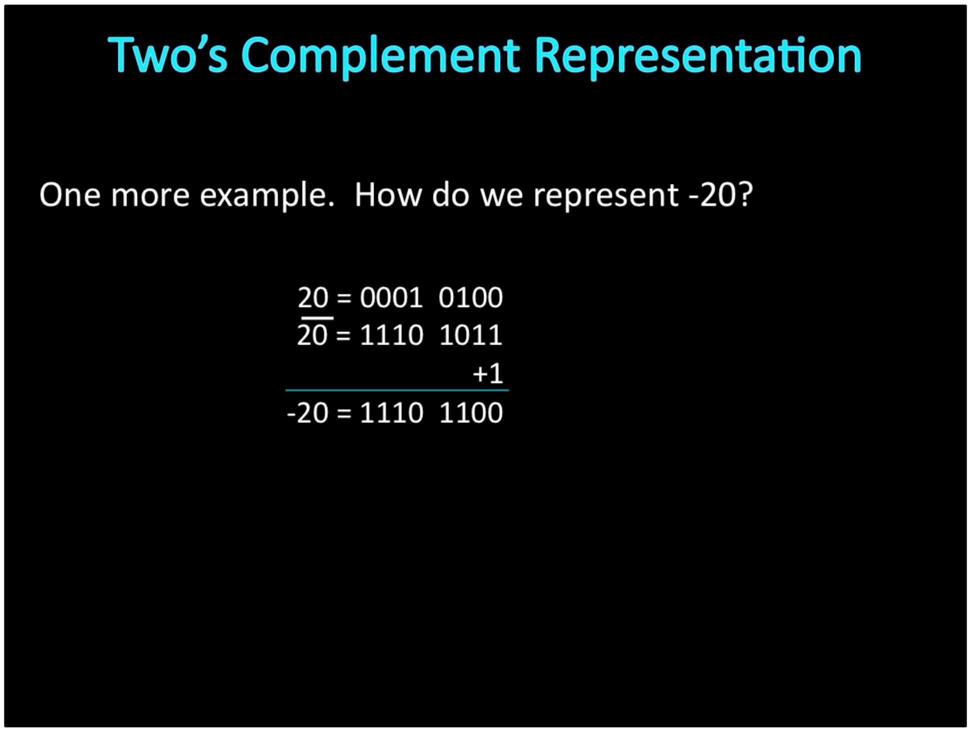
- 0 = 0000, 1 = 0001, 3 = 0011, 7 = 0111

To negate any number:

- complement *all* the bits (i.e. flip all the bits)
- then add 1
- -1: 1 $\Rightarrow$ 0001 $\Rightarrow$ 1110 $\Rightarrow$ 1111
- -3: 3 $\Rightarrow$ 0011 $\Rightarrow$ 1100 $\Rightarrow$ 1101
- -7: 7 $\Rightarrow$ 0111 $\Rightarrow$ 1000 $\Rightarrow$ 1001
- -0: 0 $\Rightarrow$ 0000 $\Rightarrow$ 1111 $\Rightarrow$ 0000 (this is good, -0 = +0)

add 1 and *discard carry*
Add a "Did you know box"
The two's complement of an N-bit number is defined as the *complement* with respect to 2^N, in other words the result of subtracting the number from 2^N. This is also equivalent to taking the *ones' complement* and then adding one, since the sum of a number and its ones' complement is all 1 bits. The two's complement of a number behaves like the negative of the original number in most arithmetic, and positive and negative numbers can coexist in a natural way.

Two's complement is the easiest to implement in hardware, which may be the ultimate reason for its widespread popularity[citation needed]. Remember that processors on the early mainframes often consisted of thousands of transistors – eliminating a significant number of transistors was a significant cost savings. The architects of the early integrated circuit based CPUs (Intel 8080, etc.) chose to use two's complement math. As IC technology advanced, virtually all adopted two's complement technology. Intel, AMD, and IBM POWER chips are all two's complement.[

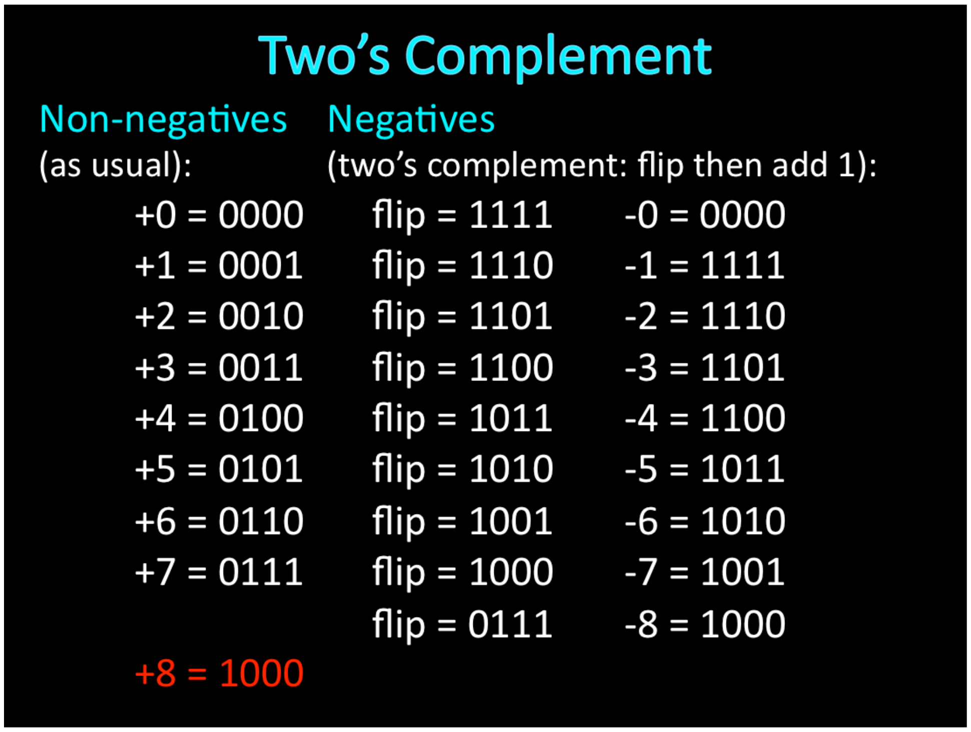


Two’s Complement Representation

One more example. How do we represent -20?

$$
\begin{array}{rl}
20 = & 0001\ \ 0100 \\
\overline{20} = & 1110\ \ 1011 \\
 & +1 \\
\hline
-20 = & 1110\ \ 1100
\end{array}
$$

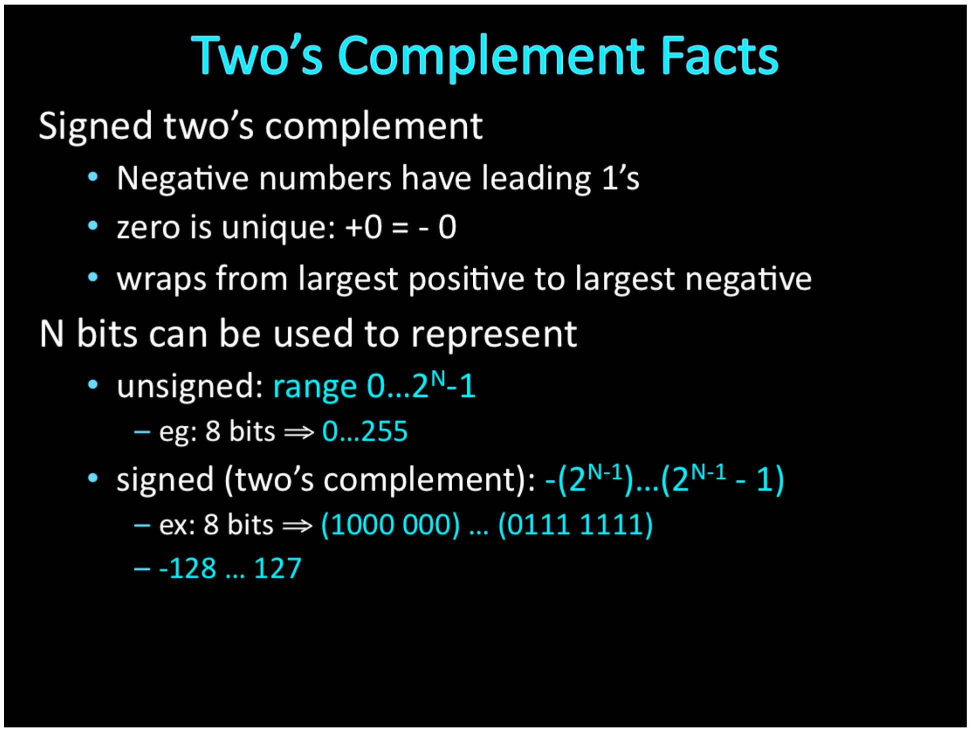
Add and subtract the same number to show that two’s complement works



Two’s Complement

Non-negatives (as usual):	Negatives (two’s complement: flip then add 1):	
+0 = 0000	flip = 1111	-0 = 0000
+1 = 0001	flip = 1110	-1 = 1111
+2 = 0010	flip = 1101	-2 = 1110
+3 = 0011	flip = 1100	-3 = 1101
+4 = 0100	flip = 1011	-4 = 1100
+5 = 0101	flip = 1010	-5 = 1011
+6 = 0110	flip = 1001	-6 = 1010
+7 = 0111	flip = 1000	-7 = 1001
	flip = 0111	-8 = 1000
+8 = 1000		

choose -8 so we have a sign bit
+0 = -0
wraps from +7 to -8
asymmetric: no +8
Range of values with n bits goes from unsigned: 0 to 2^n – 1
For signed: 2^(n-1)-1 to -2^n



Two's Complement Facts

Signed two's complement

- Negative numbers have leading 1's
- zero is unique: +0 = - 0
- wraps from largest positive to largest negative

N bits can be used to represent

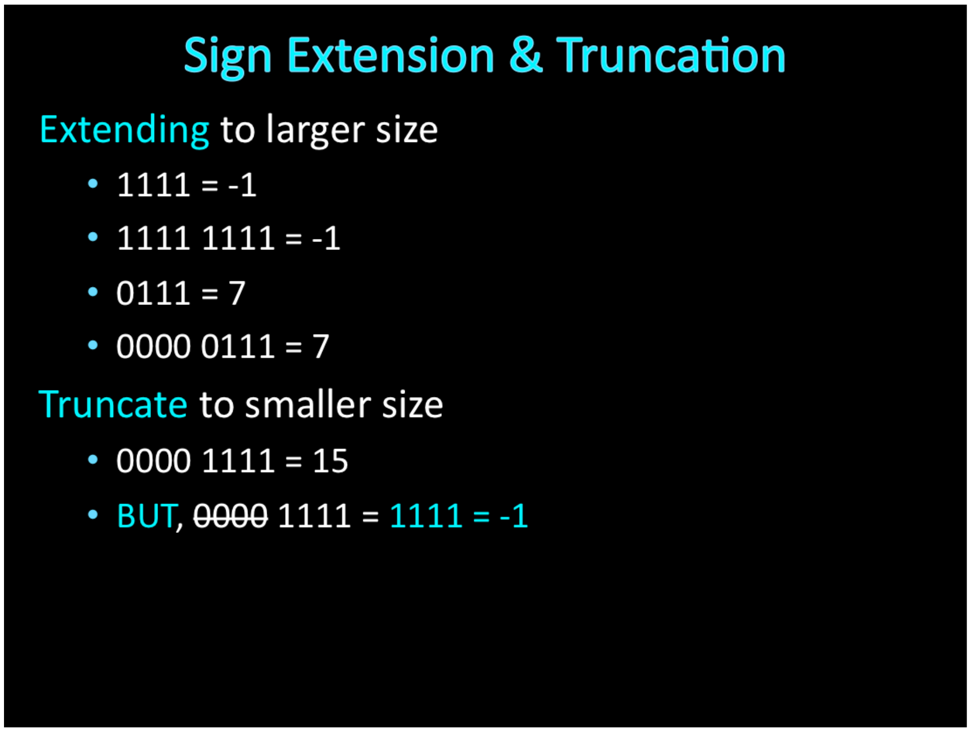
- unsigned: range 0…2^N-1
  - eg: 8 bits ⇒ 0…255
- signed (two's complement): -(2^{N-1})…(2^{N-1} - 1)
  - ex: 8 bits ⇒ (1000 000) … (0111 1111)
  - -128 … 127

Why two's complement works:
Given a set of all possible N-bit values, we can assign the lower (by binary value) half to be the integers from 0 to (2^[N−1]−1) inclusive and the upper half to be −2^[N−1] to −1 inclusive. The upper half can be used to represent negative integers from −2^[N −1] to −1 because, under addition modulo 2^N they behave the same way as those negative integers. That is to say that because i + j mod 2^N = i + (j + 2^N) mod 2^N any value in the set { j + k2^N | k is an integer } can be used in place of j.

For example, with eight bits, the unsigned bytes are 0 to 255. Subtracting 256 from the top half (128 to 255) yields the signed bytes −128 to −1.

The relationship to two's complement is realized by noting that 256 = 255 + 1, and (255 − x) is the ones' complement of x.

http://en.wikipedia.org/wiki/Two%27s_complement



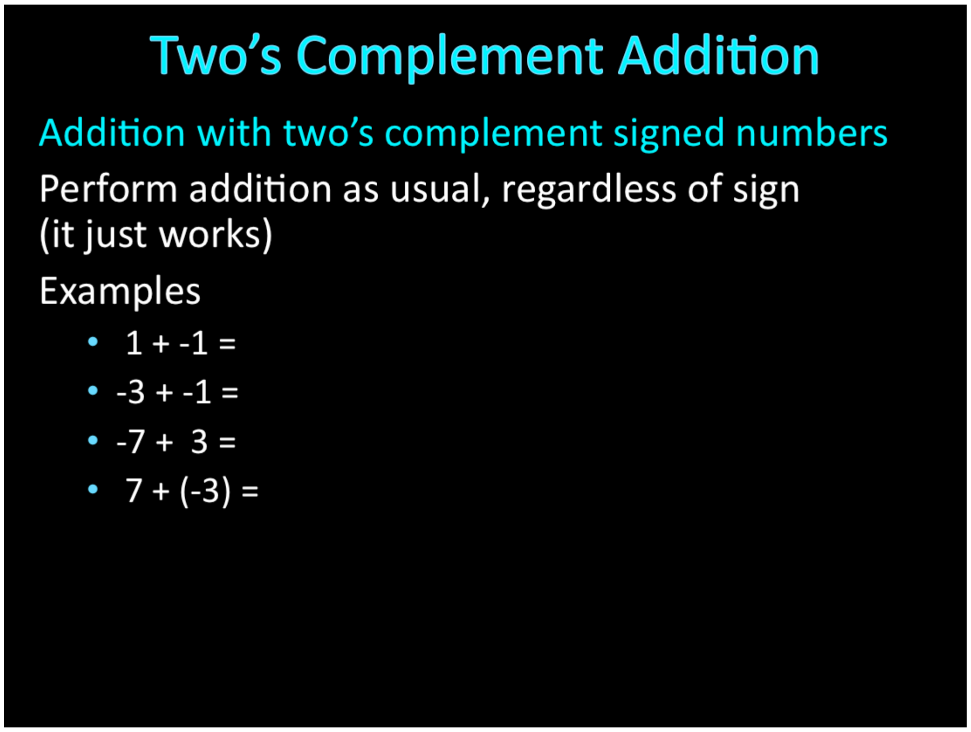
Sign Extension & Truncation

Extending to larger size

- 1111 = -1
- 1111 1111 = -1
- 0111 = 7
- 0000 0111 = 7

Truncate to smaller size

- 0000 1111 = 15
- BUT, ~~0000~~ 1111 = 1111 = -1



Two's Complement Addition

Addition with two's complement signed numbers

Perform addition as usual, regardless of sign (it just works)

Examples

- 1 + -1 =
- -3 + -1 =
- -7 + 3 =
- 7 + (-3) =

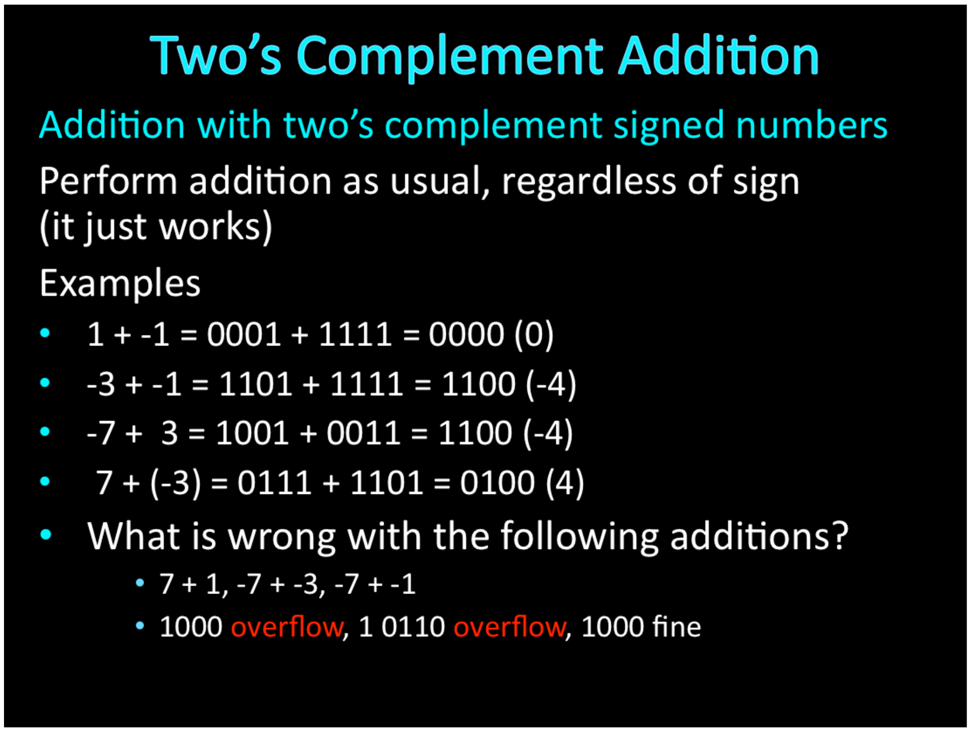
- 1 + -1 = 0001 + 1111 = 0000 (0)
- -3 + -1 = 1101 + 1111 = 1100 (-4)
- -7 + 3 = 1001 + 0011 = 1100 (-4)
- 7 + (-3) = 0111 + 1101 = 0100 (4)

7 + 1 = 0111+0001 = 1000 = -8 (OVERFLOW)! (Had a carry in to the MSB!) Sign of out != sign of in

7 + (-3) 0111+ 1101 = 1100 = -4

-7+-3 = 1001 + 1101 = 0110 (cout = 1) (Did not have a carry in to the MSB)

-7+-1 = 1001 + 1111 = 1000 (cout = 1)



Two's Complement Addition

Addition with two's complement signed numbers

Perform addition as usual, regardless of sign
(it just works)

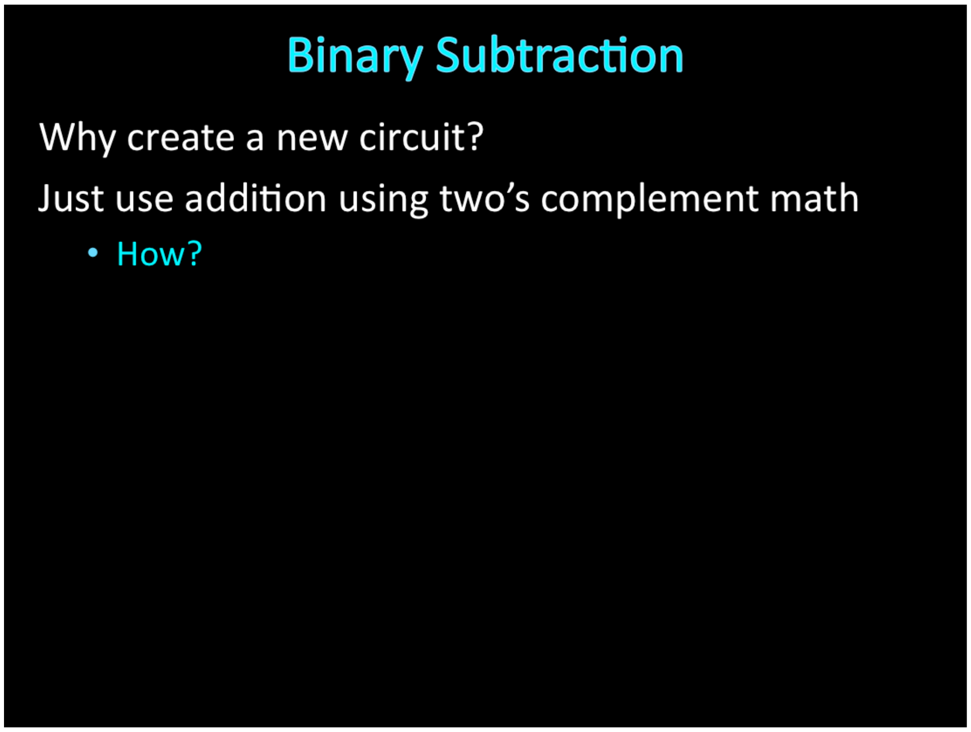
Examples

- 1 + -1 = 0001 + 1111 = 0000 (0)
- -3 + -1 = 1101 + 1111 = 1100 (-4)
- -7 + 3 = 1001 + 0011 = 1100 (-4)
- 7 + (-3) = 0111 + 1101 = 0100 (4)
- What is wrong with the following additions?
  - 7 + 1, -7 + -3, -7 + -1
  - 1000 overflow, 1 0110 overflow, 1000 fine

1 + (-1)
(-3) + (-1) = 1101 + 1111 = 1100 = (-4)
(-7) + 3 = 1001 + 0011 = 1100 (-4)
7 + (-3) = 0111 + 1101 = 0100 (4)

7 + 1 : overflow
(-7) + (-3) : overflow
(-7) + (-1)

7 + 1 = 0111+0001 = 1000 = -8 (OVERFLOW)! (Had a carry in to the MSB!) Sign of out != sign of in
-7+-3 = 1001 + 1101 =   0110 (cout = 1) (Did not have a carry in to the MSB)
-7+-1 = 1001 + 1111 =   1000 (cout = 1)



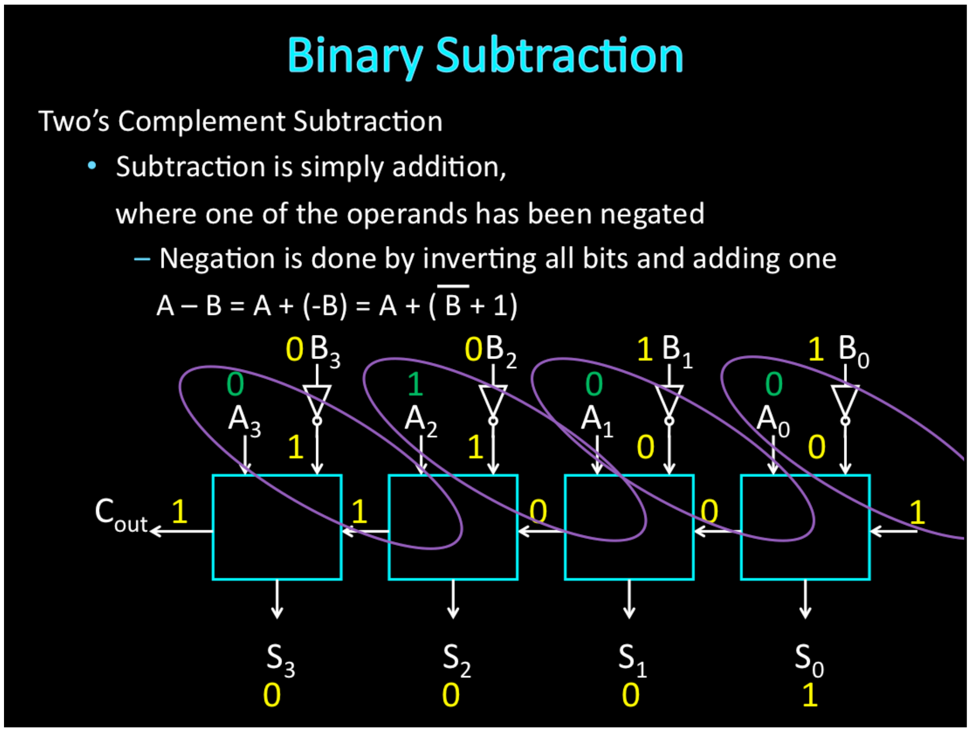
Binary Subtraction

Why create a new circuit?

Just use addition using two's complement math

- How?

a – b, where b is so large that –B overflows
B must be -8, so that –B = -8
last bit will change
if a >= 0, will correctly signal overflow
if a < 0, will correctly subtract and not signal overflow

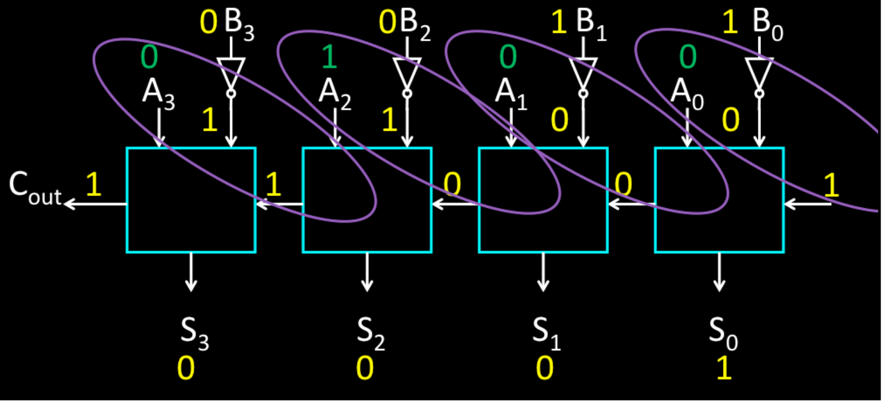


Binary Subtraction

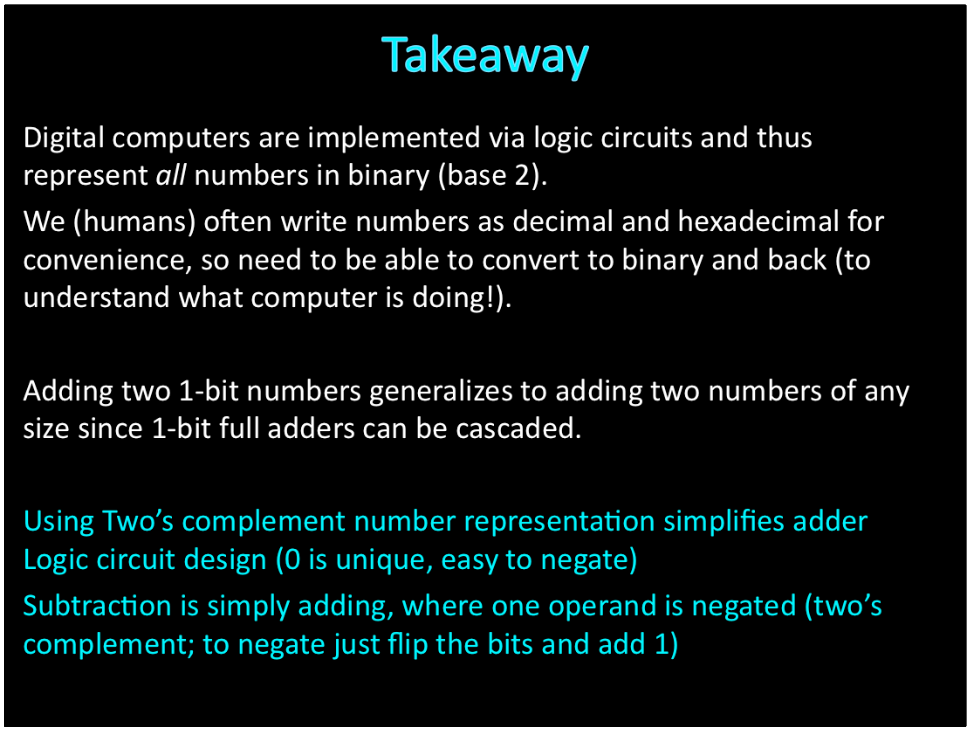
Two's Complement Subtraction

- Subtraction is simply addition, where one of the operands has been negated
  - Negation is done by inverting all bits and adding one

    $A - B = A + (-B) = A + (\overline{B} + 1)$



a – b, where b is so large that –B overflows
B must be -8, so that –B = 8
last bit will change
if a >= 0, will correctly signal overflow
if a < 0, will correctly subtract and not signal overflow



Takeaway

Digital computers are implemented via logic circuits and thus represent *all* numbers in binary (base 2).

We (humans) often write numbers as decimal and hexadecimal for convenience, so need to be able to convert to binary and back (to understand what computer is doing!).

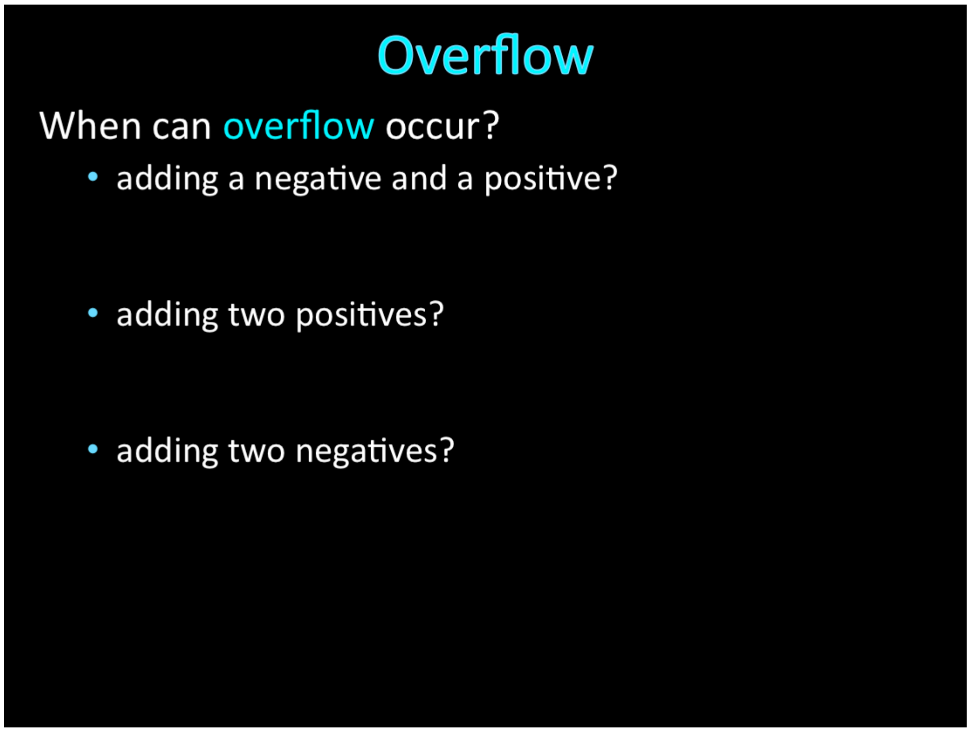
Adding two 1-bit numbers generalizes to adding two numbers of any size since 1-bit full adders can be cascaded.

Using Two's complement number representation simplifies adder Logic circuit design (0 is unique, easy to negate)

Subtraction is simply adding, where one operand is negated (two's complement; to negate just flip the bits and add 1)

Next Goal

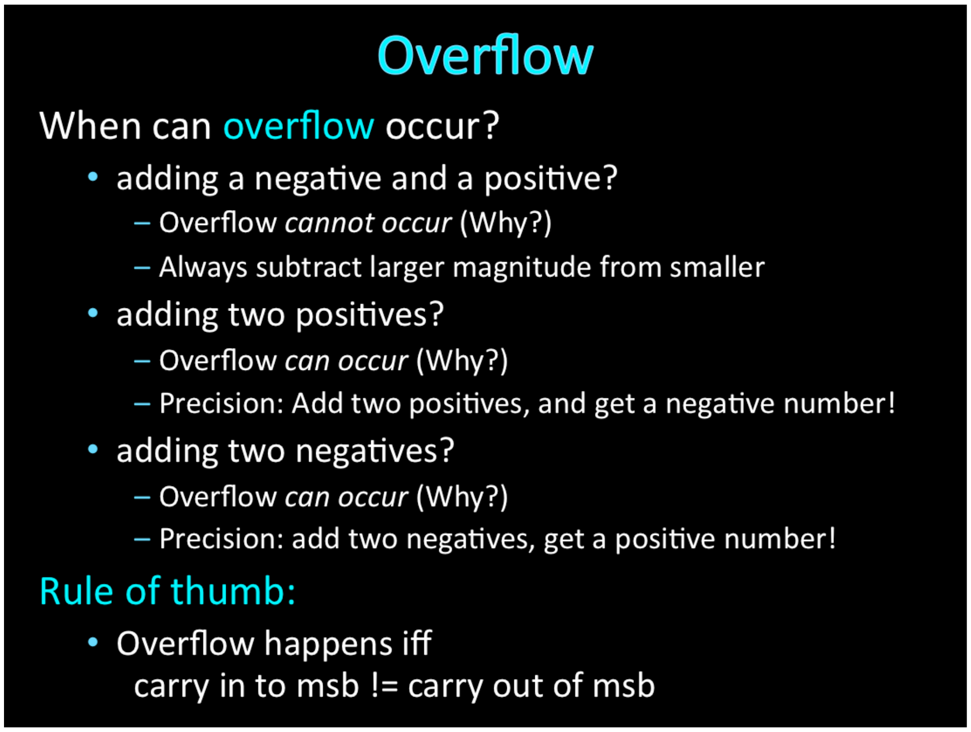
In general, how do we detect and handle overflow?



Overflow

When can overflow occur?

- adding a negative and a positive?
- adding two positives?
- adding two negatives?



Overflow

When can overflow occur?

- adding a negative and a positive?
  - Overflow *cannot occur* (Why?)
  - Always subtract larger magnitude from smaller
- adding two positives?
  - Overflow *can occur* (Why?)
  - Precision: Add two positives, and get a negative number!
- adding two negatives?
  - Overflow *can occur* (Why?)
  - Precision: add two negatives, get a positive number!

Rule of thumb:

- Overflow happens iff carry in to msb != carry out of msb

•7 + 1, -7 + -3
•1000 overflow, 1 0110 overflow,

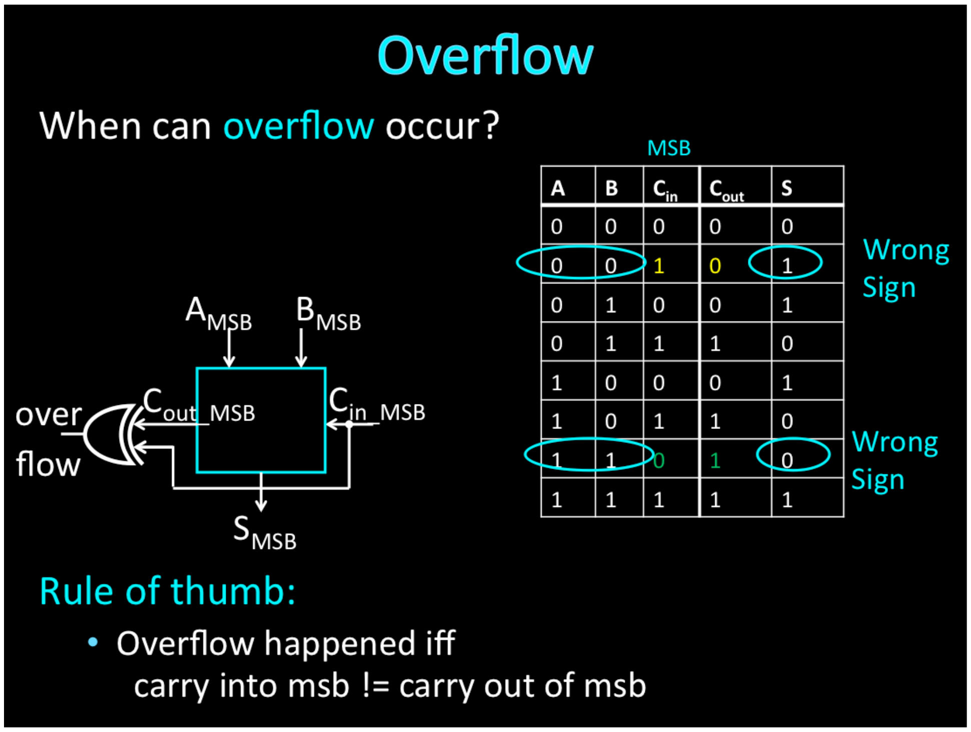
•0 1 1 1

```
0 1 1 1
0 0 0 1
1 0 0 0
c_in = 1, c_out = 0

 1001
 1101
10110
c_in = 0, c_out = 1
```

Overflow occurs because there are not enough bits to represent the precision/ magnitude of the result of the add

```
 MSB
           0
           0
Cout = 0   0    Cin = 0
           0
           0
Cout = 0   1    Cin = 1 (Overflow!)
```



Overflow

When can overflow occur?

MSB

A	B	C_{in}	C_{out}	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

Wrong Sign

Wrong Sign

Rule of thumb:

- Overflow happened iff carry into msb != carry out of msb

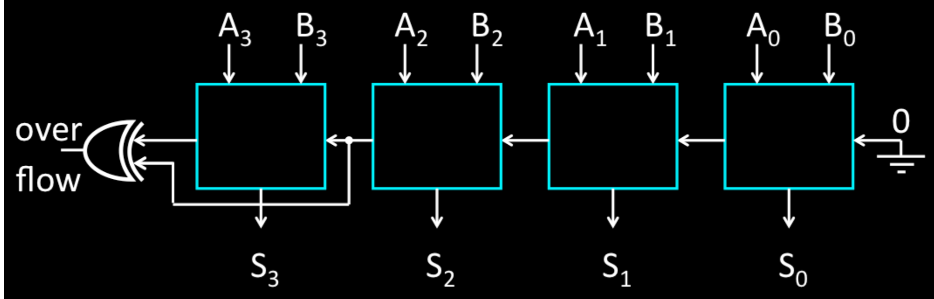
Outgoing sign of result (S) does match in coming sign of operands, A and B. This only happens when Cin != Cout
MSB!!

```
              0
              0
Cout = 0   0     Cin = 0
              0
              0
Cout = 0   1     Cin = 1 (Overflow!)

              1
              1
Cout = 1   0     Cin = 0 (Overflow!)
              1
              1
Cout = 1   1     Cin = 1
```

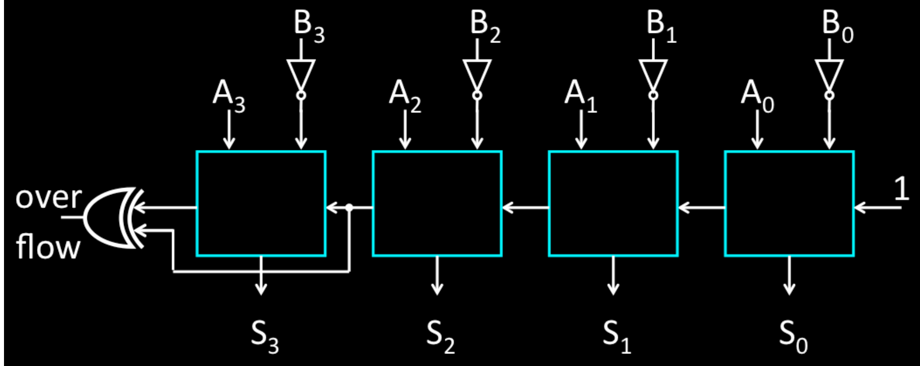
Two's Complement Adder

Two's Complement Adder with overflow detection



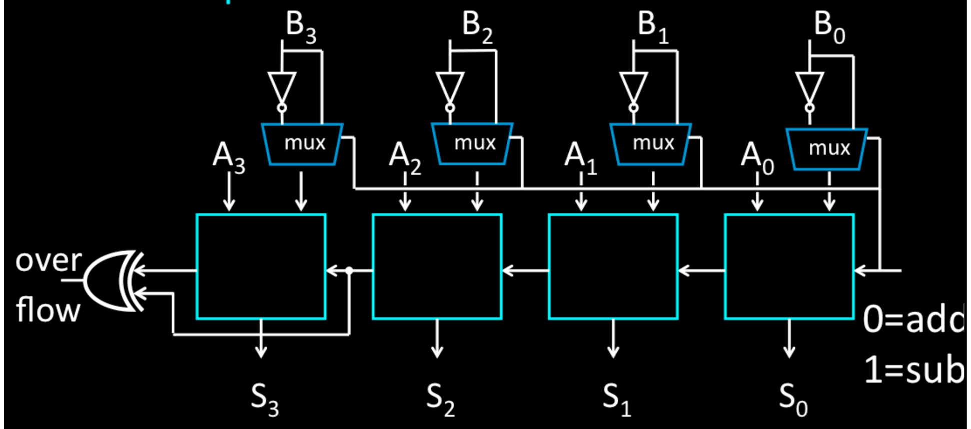
Two's Complement Adder

Two's Complement Subtraction with overflow detection



Two’s Complement Adder

Two’s Complement Adder with overflow detection



Takeaway

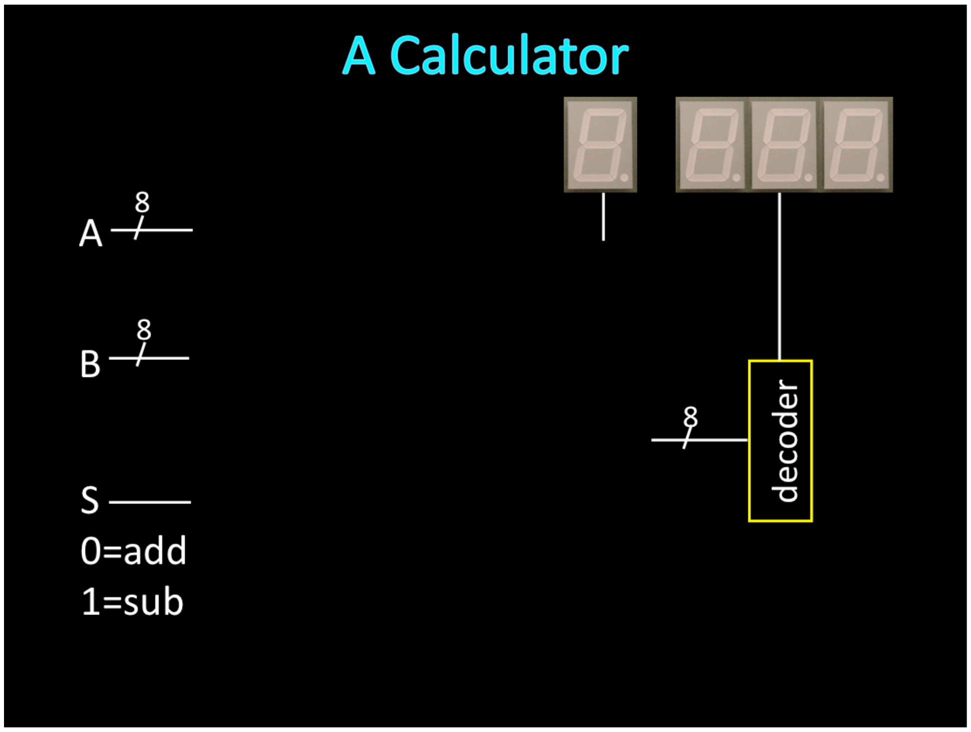
Digital computers are implemented via logic circuits and thus represent *all* numbers in binary (base 2)

We (humans) often write numbers as decimal and hexadecimal for convenience, so need to be able to convert to binary and back (to understand what computer is doing!)

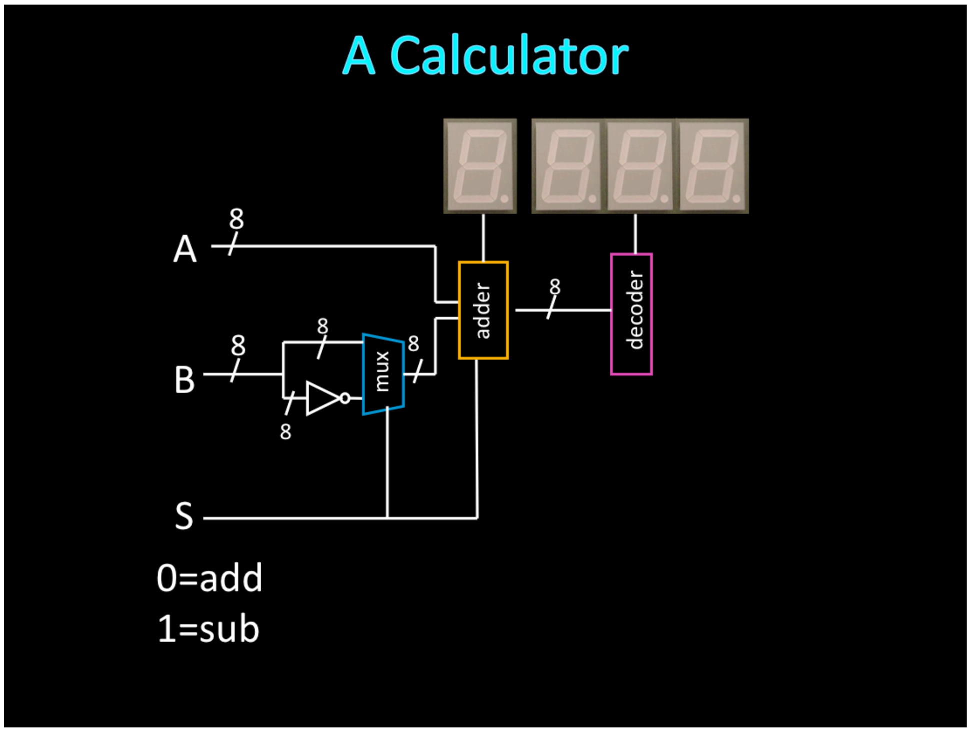
Adding two 1-bit numbers generalizes to adding two numbers of any size since 1-bit full adders can be cascaded

Using two's complement number representation simplifies adder Logic circuit design (0 is unique, easy to negate). Subtraction is simply adding, where one operand is negated (two's complement; to negate just flip the bits and add 1)

Overflow if sign of operands A and B != sign of result S. Can detect overflow by testing C_{in} != C_{out} of the most significant bit (msb), which only occurs when previous statement is true



User enters the numbers to be added or subtracted using toggle switches
User selects ADD or SUBTRACT
Muxes feed A and B,
or A and –B, to the 8-bit adder
The 8-bit decoder for the hex display is straightforward (but not shown in detail)

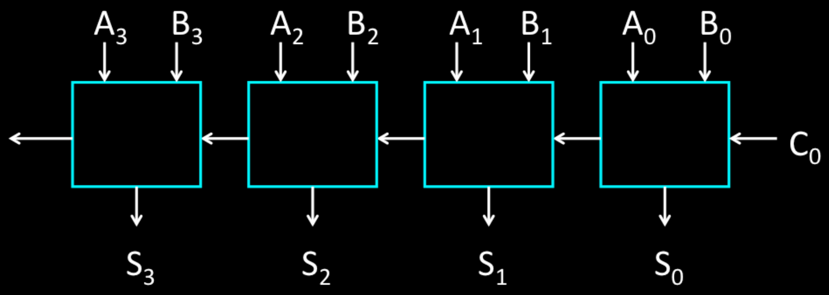


A Calculator

red herring? constants into mux

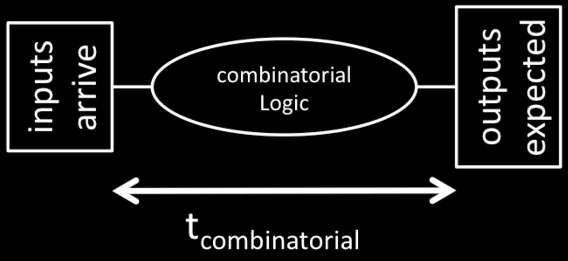
Efficiency and Generality

- Is this design fast enough?
- Can we generalize to 32 bits? 64? more?

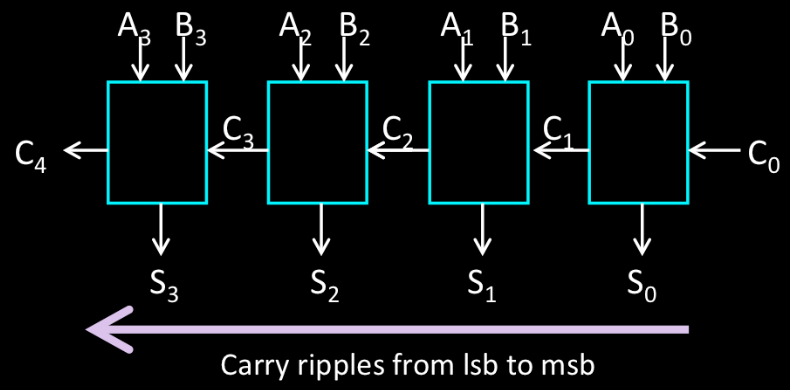


Performance

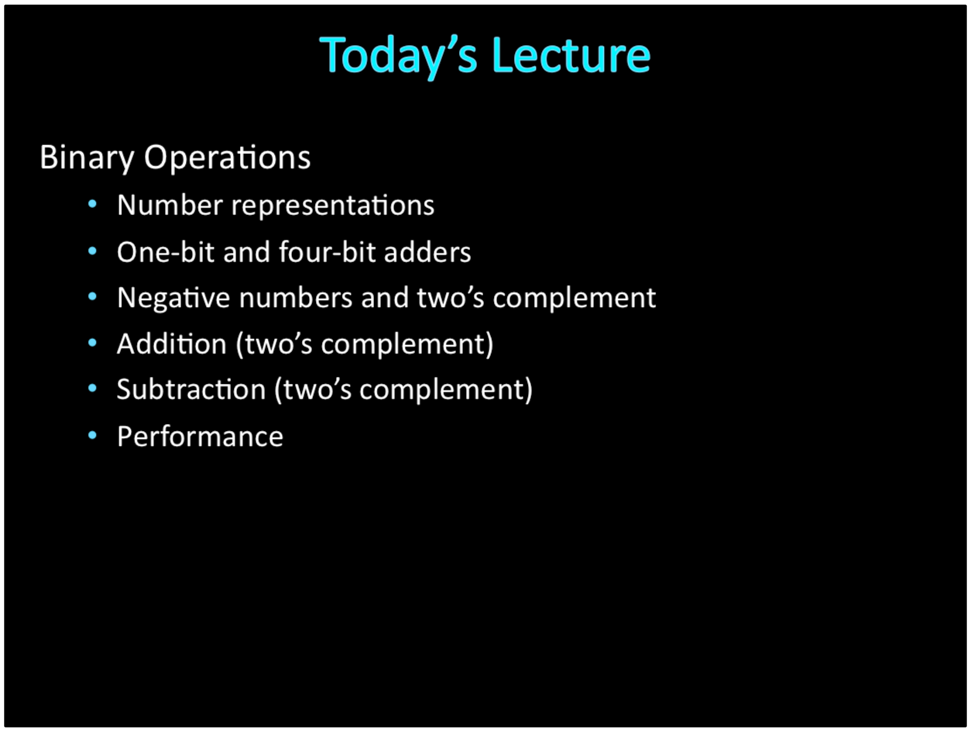
Speed of a circuit is affected by the number of gates in series (on the *critical path* or the *deepest level of logic*)



4-bit Ripple Carry Adder



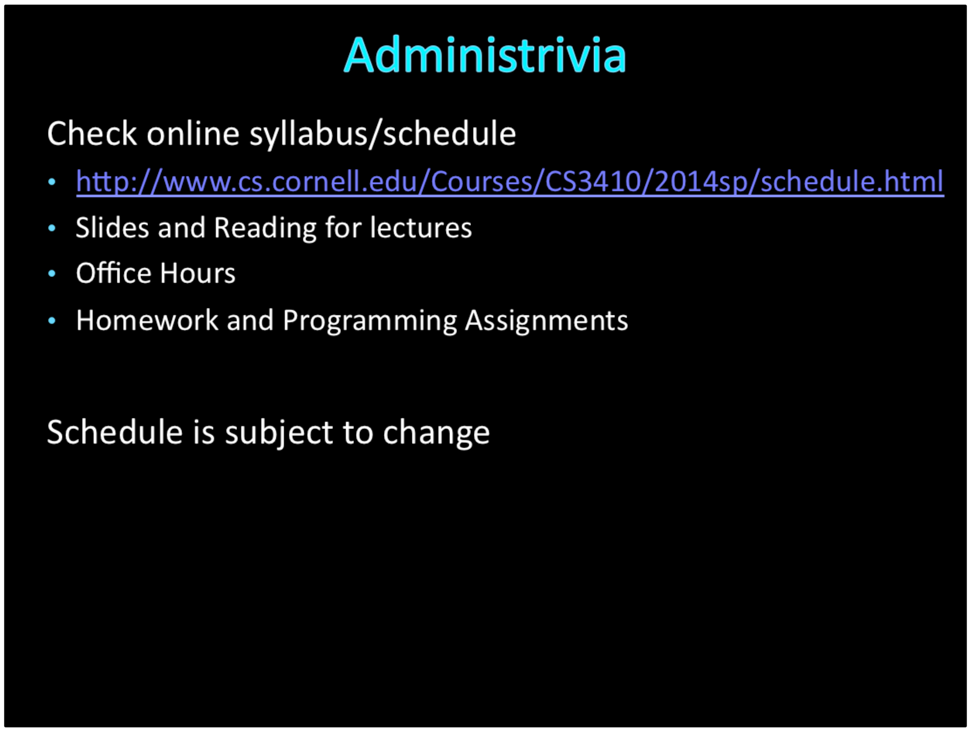
- First full adder, 2 gate delay
- Second full adder, 2 gate delay
- …



Today's Lecture

Binary Operations

- Number representations
- One-bit and four-bit adders
- Negative numbers and two's complement
- Addition (two's complement)
- Subtraction (two's complement)
- Performance



Administrivia

Check online syllabus/schedule

- http://www.cs.cornell.edu/Courses/CS3410/2014sp/schedule.html
- Slides and Reading for lectures
- Office Hours
- Homework and Programming Assignments

Schedule is subject to change