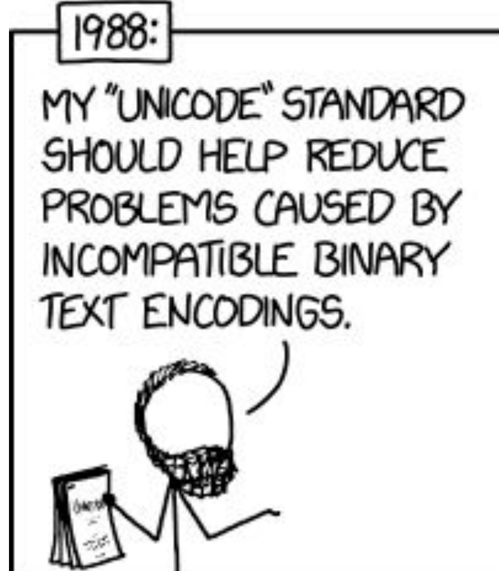
Integers II, Floating Point I

CSE 351 Summer 2021

Instructor:
Mara Kirdani-Ryan

Teaching Assistants:
Kashish Aggarwal
Nick Durand
Colton Jobes
Tim Mandzyuk



http://xkcd.com/1953/

Gentle and Loving Reminders!

- Hw4, hw5, lab1a due Monday 7/2 – 8pm
  - Submit `pointer.c` and `lab1Areflect.txt` to Gradescope for lab1a
- hw6 and hw7 due Friday 7/9 – 8pm
- Lab 1b released today, due 7/9
  - Bit manipulation problems using custom data type
  - Some helpful examples in the bonus slides today and next time
- Anything else?
  - We're here to help! You just need to ask!

Gradescope Lab Turnin

- Make sure you pass the File and Compilation Check!
- Doesn't indicate if you passed all tests, just indicates that all the correct files were found and there were no compilation or runtime errors.
- Use the testing programs we provide to check your solution for correctness (on attu or the VM)

Ohyay's more secure

- You'll need an invite to join, anyone on staff can give invites
  - This shouldn't affect anyone, let us know if it does!

Quick Aside: C Macros

- You'll use C macros in Lab1b for bit masks
- Syntax is of the form:

```
#define NAME expression
```

- Can use "NAME" instead of "expression" in code
- Useful to help with readability/factoring in code
  - Especially useful for defining constants such as bit masks!

Quick Aside: C Macros

- Are NOT exactly the same as a constant in Java
  - Does naïve copy and replace *before* compilation.
  - Everywhere the characters “NAME” appear in the code, the characters “expression” will now appear instead.
  - Lots of gotchas, be careful!
- See Lecture 4 (Integers I) slides for examples

2021

How are y'all feeling today?

We're going to talk about automation!

2021

Breakouts!

What work is being automated? Why? Is there anything that separates “automated work” from work still performed by people?

Integers

- Binary representation of integers
  - Unsigned and signed
- Shifting and **arithmetic operations** – for Lab 1a
- In C: Signed, Unsigned and Casting
- Consequences of finite width representations
  - Overflow, sign extension

Learning Objectives

By the end of this lecture, you should be able to:

- Explain integer overflow at the level of bits (ideally in plain language to someone outside this class)
- Determine when signed/unsigned overflow will occur
- Convert decimal numbers to fixed point encoding
- Explain the values embedded in the first computers, with comparisons to computing today

Reading Review

- Terminology:
  - `UMin`, `UMax`, `TMin`, `TMax`
  - Type casting: implicit vs. explicit
  - Integer extension: zero extension vs. sign extension
  - Modular arithmetic and arithmetic overflow
  - Bit shifting: left shift, logical right shift, arithmetic right shift

Review Questions

- What is the value (and encoding) of **`TMin`** for a fictional 6-bit wide integer data type?

- For `unsigned char uc = 0xA1;`, what is the result of the cast **`(short)uc`**?

- What is the result of the following expressions?
  - **`(signed char)uc >> 2`**
  - **`(unsigned char)uc >> 3`**

Two's Complement Arithmetic

- The same addition procedure works for both unsigned and two's complement integers
  - **Simplifies hardware:** only one algorithm for addition
  - **Algorithm:** simple addition, discard highest carry bit
    - Called modular addition: result is sum *modulo* 2^w
- **4-bit Examples:**

HW	TC
0100 +0011	
=	

Two's Complement Arithmetic

- The same addition procedure works for both unsigned and two's complement integers
  - **Simplifies hardware:** only one algorithm for addition
  - **Algorithm:** simple addition, discard highest carry bit
    - Called modular addition: result is sum *modulo* 2^w
- **4-bit Examples:**

HW	TC
0100	4
+0011	+3
=	

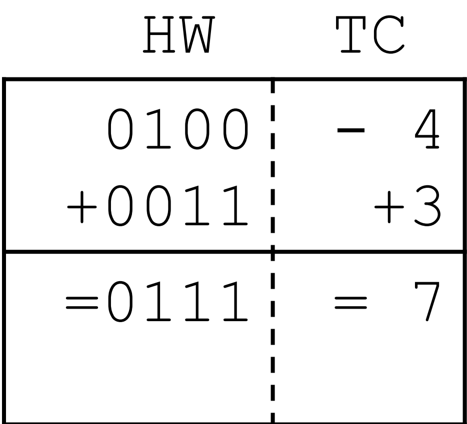
Two's Complement Arithmetic

- The same addition procedure works for both unsigned and two's complement integers
  - **Simplifies hardware:** only one algorithm for addition
  - **Algorithm:** simple addition, discard highest carry bit
    - Called modular addition: result is sum *modulo* 2^w
- **4-bit Examples:**

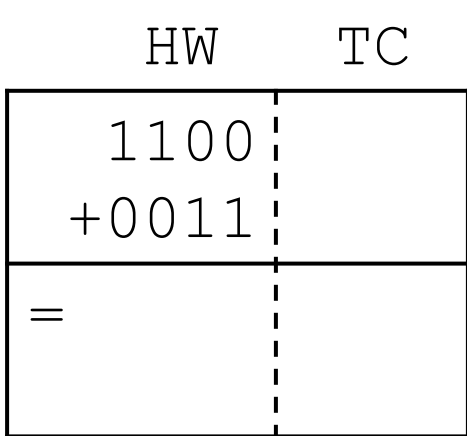
HW	TC
0100	4
+0011	+3
=0111	= 7

Two's Complement Arithmetic

- The same addition procedure works for both unsigned and two's complement integers
  - **Simplifies hardware:** only one algorithm for addition
  - **Algorithm:** simple addition, discard highest carry bit
    - Called modular addition: result is sum *modulo* 2^w
- **4-bit Examples:**



HW	TC
0100	- 4
+0011	+3
=0111	= 7

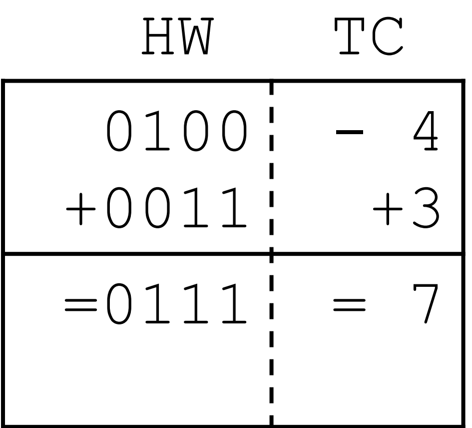


HW	TC
1100	
+0011	
=	

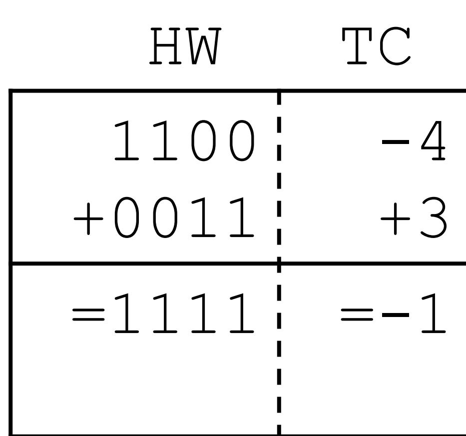
HW	TC
0100	
+1101	
=0001	

Two's Complement Arithmetic

- The same addition procedure works for both unsigned and two's complement integers
  - **Simplifies hardware:** only one algorithm for addition
  - **Algorithm:** simple addition, discard highest carry bit
    - Called modular addition: result is sum *modulo* 2^w
- **4-bit Examples:**



HW	TC
0100	- 4
+0011	+3
=0111	= 7



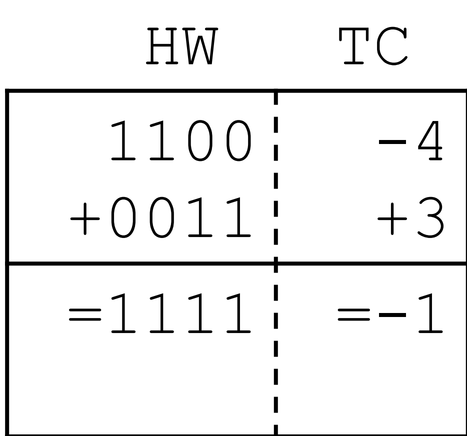
HW	TC
1100	-4
+0011	+3
=1111	=-1

HW	TC
0100	
+1101	
=	

Two's Complement Arithmetic

- The same addition procedure works for both unsigned and two's complement integers
  - **Simplifies hardware:** only one algorithm for addition
  - **Algorithm:** simple addition, discard highest carry bit
    - Called modular addition: result is sum *modulo* 2^w
- **4-bit Examples:**

HW	TC
0100	- 4
+0011	+3
=0111	= 7



HW	TC
1100	-4
+0011	+3
=1111	=-1

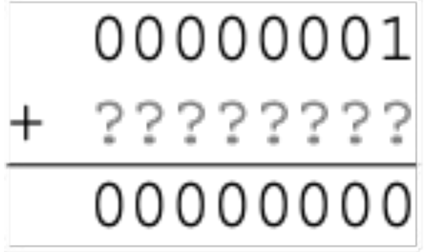
HW	TC
0100	4
+1101	-3
=0001	= 1

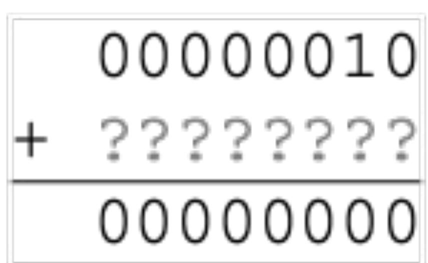
Why Does Two's Complement Work?

- For all representable positive integers x, we want:

bit representation of x
+ *bit representation of* –x (ignoring the carry-out bit)
0

  - What are the 8-bit negative encodings for the following?





11000011
+ ????????
00000000

Why Does Two's Complement Work?

- For all representable positive integers x, we want:

  bit representation of x
  *+ bit representation of –*x
  (ignoring the carry-out bit)
  0

  - What are the 8-bit negative encodings for the following?

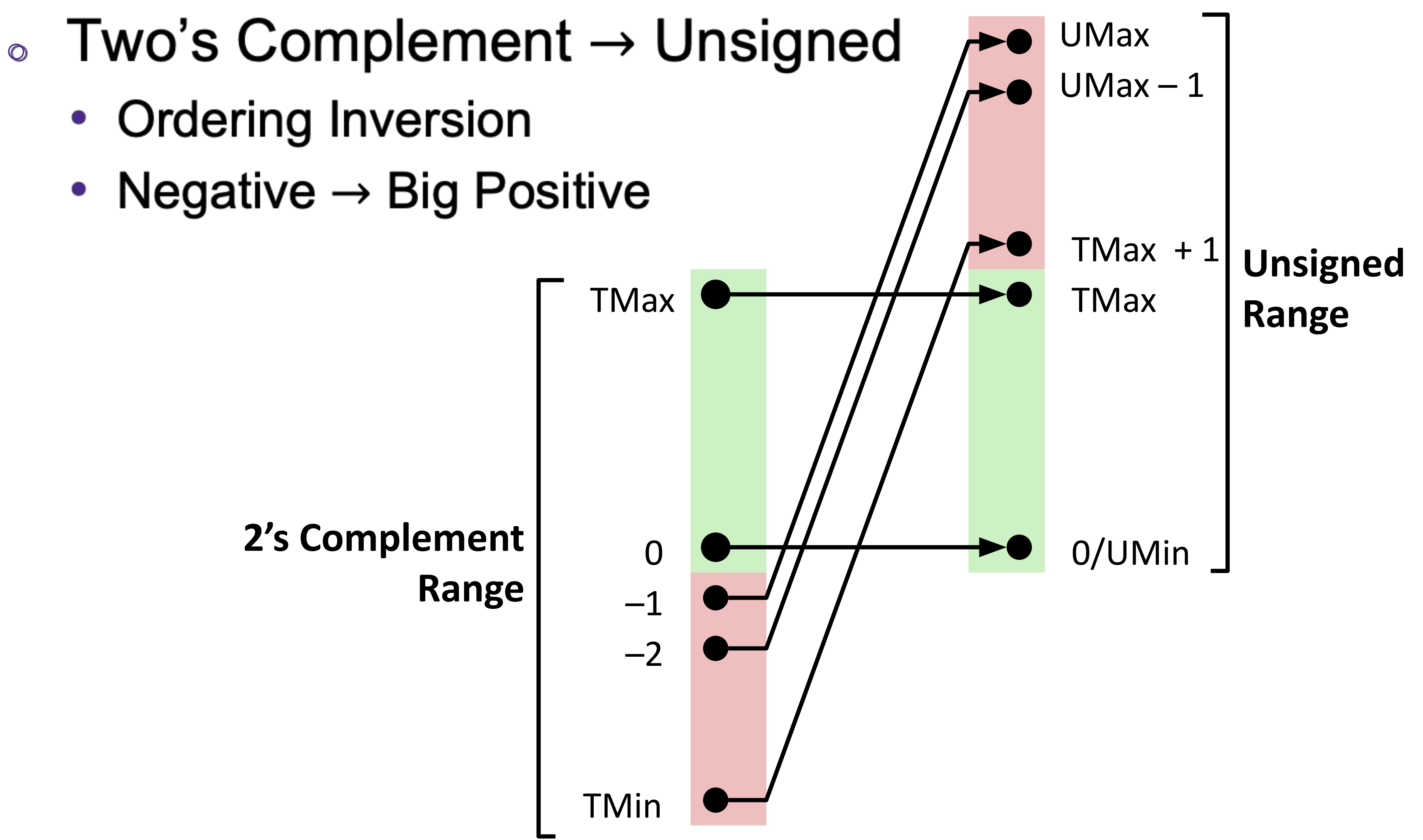
```
  00000001        00000010        11000011
+ 11111111      + 11111110      + 00111101
 100000000       100000000       100000000
```

These are the bitwise complement plus 1!

`-x == ~x + 1`

Signed/Unsigned Conversion Visualized

- Two's Complement → Unsigned
  - Ordering Inversion
  - Negative → Big Positive



Values To Remember (or lookup)

- Unsigned Values
  - UMin = `0b00…0` = 0
  - UMax = `0b11…1` = $2^w - 1$
- Two's Complement Values
  - TMin = `0b10…0` = -2^{w-1}
  - TMax = `0b01…1` = $2^{w-1} - 1$
  - -1 = `0b11…1`
- **Example:** Values for $w = 64$

	Decimal	Hex
UMax	18,446,744,073,709,551,615	FF FF FF FF FF FF FF FF
TMax	9,223,372,036,854,775,807	7F FF FF FF FF FF FF FF
TMin	-9,223,372,036,854,775,808	80 00 00 00 00 00 00 00
-1	-1	FF FF FF FF FF FF FF FF
0	0	00 00 00 00 00 00 00 00

Integers

- Binary representation of integers
  - Unsigned and signed
- Shifting and arithmetic operations – *Lab 1a*
- **In C: Signed, Unsigned and Casting**
- Consequences of finite width representations
  - Overflow, sign extension

In C: Signed vs. Unsigned

- Casting
  - Bits are unchanged, just interpreted differently!
    - **int** tx, ty;
    - **unsigned int** ux, uy;
  - *Explicit* casting
    - tx = (**int**) ux;
    - uy = (**unsigned int**) ty;
  - *Implicit* casting can occur during assignments or function calls
    - tx = ux;
    - uy = ty;

Casting Surprises

- Integer literals (constants)
  - By default, integer constants are *signed* integers
    - Hex constants already have an explicit binary representation
  - Use “U” (or “u”) suffix to explicitly force *unsigned*
    - **Examples:** 0U, 4294967259u
- Expression Evaluation
  - Mixing unsigned and signed in an expression means **signed values are implicitly cast to unsigned**
  - Including comparison operators <, >, ==, <=, >=

Casting Surprises

False

- 32-bit examples:
  - TMin = -2,147,483,648, TMax = 2,147,483,647

Left Constant	Order	Right Constant	Interpretation
0 0000 0000 0000 0000 0000 0000 0000 0000	<	**0U** 0000 0000 0000 0000 0000 0000 0000 0000	F
-1 1111 1111 1111 1111 1111 1111 1111 1111	<	**0** 0000 0000 0000 0000 0000 0000 0000 0000	T
-1 1111 1111 1111 1111 1111 1111 1111 1111	<	**0U** 0000 0000 0000 0000 0000 0000 0000 0000	
2147483647 0111 1111 1111 1111 1111 1111 1111 1111	<	**-2147483648** 1000 0000 0000 0000 0000 0000 0000 0000	
2147483647U 0111 1111 1111 1111 1111 1111 1111 1111	<	**-2147483648** 1000 0000 0000 0000 0000 0000 0000 0000	
-1 1111 1111 1111 1111 1111 1111 1111 1111	<	**-2** 1111 1111 1111 1111 1111 1111 1111 1110	
(unsigned) -1 1111 1111 1111 1111 1111 1111 1111 1111	<	**-2** 1111 1111 1111 1111 1111 1111 1111 1110	
2147483647 0111 1111 1111 1111 1111 1111 1111 1111	<	**2147483648U** 1000 0000 0000 0000 0000 0000 0000 0000	
2147483647 0111 1111 1111 1111 1111 1111 1111 1111	<	**(int) 2147483648U** 1000 0000 0000 0000 0000 0000 0000 0000	

Casting Surprises

- True False Help!

○ 32-bit examples:

- TMin = -2,147,483,648, TMax = 2,147,483,647

Left Constant	Order	Right Constant	Interpretation
0 0000 0000 0000 0000 0000 0000 0000 0000	<	**0U** 0000 0000 0000 0000 0000 0000 0000 0000	F
-1 1111 1111 1111 1111 1111 1111 1111 1111	<	**0** 0000 0000 0000 0000 0000 0000 0000 0000	T
-1 1111 1111 1111 1111 1111 1111 1111 1111	<	**0U** 0000 0000 0000 0000 0000 0000 0000 0000	F
2147483647 0111 1111 1111 1111 1111 1111 1111 1111	<	**-2147483648** 1000 0000 0000 0000 0000 0000 0000 0000	
2147483647U 0111 1111 1111 1111 1111 1111 1111 1111	<	**-2147483648** 1000 0000 0000 0000 0000 0000 0000 0000	
-1 1111 1111 1111 1111 1111 1111 1111 1111	<	**-2** 1111 1111 1111 1111 1111 1111 1111 1110	
(unsigned) -1 1111 1111 1111 1111 1111 1111 1111 1111	<	**-2** 1111 1111 1111 1111 1111 1111 1111 1110	
2147483647 0111 1111 1111 1111 1111 1111 1111 1111	<	**2147483648U** 1000 0000 0000 0000 0000 0000 0000 0000	
2147483647 0111 1111 1111 1111 1111 1111 1111 1111	<	**(int) 2147483648U** 1000 0000 0000 0000 0000 0000 0000 0000	

Casting Surprises

- 32-bit examples:
  - TMin = -2,147,483,648, TMax = 2,147,483,647

Left Constant	Order	Right Constant	Interpretation
0 0000 0000 0000 0000 0000 0000 0000 0000	<	**0U** 0000 0000 0000 0000 0000 0000 0000 0000	F
-1 1111 1111 1111 1111 1111 1111 1111 1111	<	**0** 0000 0000 0000 0000 0000 0000 0000 0000	T
-1 1111 1111 1111 1111 1111 1111 1111 1111	<	**0U** 0000 0000 0000 0000 0000 0000 0000 0000	F
2147483647 0111 1111 1111 1111 1111 1111 1111 1111	<	**-2147483648** 1000 0000 0000 0000 0000 0000 0000 0000	F
2147483647U 0111 1111 1111 1111 1111 1111 1111 1111	<	**-2147483648** 1000 0000 0000 0000 0000 0000 0000 0000	
-1 1111 1111 1111 1111 1111 1111 1111 1111	<	**-2** 1111 1111 1111 1111 1111 1111 1111 1110	
(unsigned) -1 1111 1111 1111 1111 1111 1111 1111 1111	<	**-2** 1111 1111 1111 1111 1111 1111 1111 1110	
2147483647 0111 1111 1111 1111 1111 1111 1111 1111	<	**2147483648U** 1000 0000 0000 0000 0000 0000 0000 0000	
2147483647 0111 1111 1111 1111 1111 1111 1111 1111	<	**(int) 2147483648U** 1000 0000 0000 0000 0000 0000 0000 0000	

Casting Surprises

-

False Help!

- 32-bit examples:
  - TMin = -2,147,483,648, TMax = 2,147,483,647

Left Constant	Order	Right Constant	Interpretation
0 0000 0000 0000 0000 0000 0000 0000 0000	<	**0U** 0000 0000 0000 0000 0000 0000 0000 0000	F
-1 1111 1111 1111 1111 1111 1111 1111 1111	<	**0** 0000 0000 0000 0000 0000 0000 0000 0000	T
-1 1111 1111 1111 1111 1111 1111 1111 1111	<	**0U** 0000 0000 0000 0000 0000 0000 0000 0000	F
2147483647 0111 1111 1111 1111 1111 1111 1111 1111	<	**-2147483648** 1000 0000 0000 0000 0000 0000 0000 0000	F
2147483647U 0111 1111 1111 1111 1111 1111 1111 1111	<	**-2147483648** 1000 0000 0000 0000 0000 0000 0000 0000	T
-1 1111 1111 1111 1111 1111 1111 1111 1111	<	**-2** 1111 1111 1111 1111 1111 1111 1111 1110	
(unsigned) -1 1111 1111 1111 1111 1111 1111 1111 1111	<	**-2** 1111 1111 1111 1111 1111 1111 1111 1110	
2147483647 0111 1111 1111 1111 1111 1111 1111 1111	<	**2147483648U** 1000 0000 0000 0000 0000 0000 0000 0000	
2147483647 0111 1111 1111 1111 1111 1111 1111 1111	<	**(int) 2147483648U** 1000 0000 0000 0000 0000 0000 0000 0000	

Casting Surprises

- 32-bit examples:
  - TMin = -2,147,483,648, TMax = 2,147,483,647

Left Constant	Order	Right Constant	Interpretation
0 0000 0000 0000 0000 0000 0000 0000 0000	<	**0U** 0000 0000 0000 0000 0000 0000 0000 0000	F
-1 1111 1111 1111 1111 1111 1111 1111 1111	<	**0** 0000 0000 0000 0000 0000 0000 0000 0000	T
-1 1111 1111 1111 1111 1111 1111 1111 1111	<	**0U** 0000 0000 0000 0000 0000 0000 0000 0000	F
2147483647 0111 1111 1111 1111 1111 1111 1111 1111	<	**-2147483648** 1000 0000 0000 0000 0000 0000 0000 0000	F
2147483647U 0111 1111 1111 1111 1111 1111 1111 1111	<	**-2147483648** 1000 0000 0000 0000 0000 0000 0000 0000	T
-1 1111 1111 1111 1111 1111 1111 1111 1111	<	**-2** 1111 1111 1111 1111 1111 1111 1111 1110	T
(unsigned) -1 1111 1111 1111 1111 1111 1111 1111 1111	<	**-2** 1111 1111 1111 1111 1111 1111 1111 1110	
2147483647 0111 1111 1111 1111 1111 1111 1111 1111	<	**2147483648U** 1000 0000 0000 0000 0000 0000 0000 0000	
2147483647 0111 1111 1111 1111 1111 1111 1111 1111	<	**(int) 2147483648U** 1000 0000 0000 0000 0000 0000 0000 0000	

Casting Surprises

- 32-bit examples:
  - TMin = -2,147,483,648, TMax = 2,147,483,647

Left Constant	Order	Right Constant	Interpretation
0 0000 0000 0000 0000 0000 0000 0000 0000	<	**0U** 0000 0000 0000 0000 0000 0000 0000 0000	F
-1 1111 1111 1111 1111 1111 1111 1111 1111	<	**0** 0000 0000 0000 0000 0000 0000 0000 0000	T
-1 1111 1111 1111 1111 1111 1111 1111 1111	<	**0U** 0000 0000 0000 0000 0000 0000 0000 0000	F
2147483647 0111 1111 1111 1111 1111 1111 1111 1111	<	**-2147483648** 1000 0000 0000 0000 0000 0000 0000 0000	F
2147483647U 0111 1111 1111 1111 1111 1111 1111 1111	<	**-2147483648** 1000 0000 0000 0000 0000 0000 0000 0000	T
-1 1111 1111 1111 1111 1111 1111 1111 1111	<	**-2** 1111 1111 1111 1111 1111 1111 1111 1110	T
(unsigned) -1 1111 1111 1111 1111 1111 1111 1111 1111	<	**-2** 1111 1111 1111 1111 1111 1111 1111 1110	F
2147483647 0111 1111 1111 1111 1111 1111 1111 1111	<	**2147483648U** 1000 0000 0000 0000 0000 0000 0000 0000	
2147483647 0111 1111 1111 1111 1111 1111 1111 1111	<	**(int) 2147483648U** 1000 0000 0000 0000 0000 0000 0000 0000	

Casting Surprises

- 32-bit examples:
  - TMin = -2,147,483,648, TMax = 2,147,483,647

Left Constant	Order	Right Constant	Interpretation
0 0000 0000 0000 0000 0000 0000 0000 0000	<	**0U** 0000 0000 0000 0000 0000 0000 0000 0000	F
-1 1111 1111 1111 1111 1111 1111 1111 1111	<	**0** 0000 0000 0000 0000 0000 0000 0000 0000	T
-1 1111 1111 1111 1111 1111 1111 1111 1111	<	**0U** 0000 0000 0000 0000 0000 0000 0000 0000	F
2147483647 0111 1111 1111 1111 1111 1111 1111 1111	<	**-2147483648** 1000 0000 0000 0000 0000 0000 0000 0000	F
2147483647U 0111 1111 1111 1111 1111 1111 1111 1111	<	**-2147483648** 1000 0000 0000 0000 0000 0000 0000 0000	T
-1 1111 1111 1111 1111 1111 1111 1111 1111	<	**-2** 1111 1111 1111 1111 1111 1111 1111 1110	T
(unsigned) -1 1111 1111 1111 1111 1111 1111 1111 1111	<	**-2** 1111 1111 1111 1111 1111 1111 1111 1110	F
2147483647 0111 1111 1111 1111 1111 1111 1111 1111	<	**2147483648U** 1000 0000 0000 0000 0000 0000 0000 0000	T
2147483647 0111 1111 1111 1111 1111 1111 1111 1111	<	**(int) 2147483648U** 1000 0000 0000 0000 0000 0000 0000 0000	

Casting Surprises

- 32-bit examples:
  - TMin = -2,147,483,648, TMax = 2,147,483,647

Left Constant	Order	Right Constant	Interpretation
0 0000 0000 0000 0000 0000 0000 0000 0000	<	**0U** 0000 0000 0000 0000 0000 0000 0000 0000	F
-1 1111 1111 1111 1111 1111 1111 1111 1111	<	**0** 0000 0000 0000 0000 0000 0000 0000 0000	T
-1 1111 1111 1111 1111 1111 1111 1111 1111	<	**0U** 0000 0000 0000 0000 0000 0000 0000 0000	F
2147483647 0111 1111 1111 1111 1111 1111 1111 1111	<	**-2147483648** 1000 0000 0000 0000 0000 0000 0000 0000	F
2147483647U 0111 1111 1111 1111 1111 1111 1111 1111	<	**-2147483648** 1000 0000 0000 0000 0000 0000 0000 0000	T
-1 1111 1111 1111 1111 1111 1111 1111 1111	<	**-2** 1111 1111 1111 1111 1111 1111 1111 1110	T
(unsigned) -1 1111 1111 1111 1111 1111 1111 1111 1111	<	**-2** 1111 1111 1111 1111 1111 1111 1111 1110	F
2147483647 0111 1111 1111 1111 1111 1111 1111 1111	<	**2147483648U** 1000 0000 0000 0000 0000 0000 0000 0000	T
2147483647 0111 1111 1111 1111 1111 1111 1111 1111	<	**(int) 2147483648U** 1000 0000 0000 0000 0000 0000 0000 0000	F

2021

How do we feel about casting?

Integers

- Binary representation of integers
  - Unsigned and signed
- Shifting and arithmetic operations – for Lab 1a
- In C: Signed, Unsigned and Casting
- **Consequences of finite width representations**
  - **Overflow, sign extension, limits**

Arithmetic Overflow

Bits	Unsigned	Signed
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	-8
1001	9	-7
1010	10	-6
1011	11	-5
1100	12	-4
1101	13	-3
1110	14	-2
1111	15	-1

- When a calculation produces a result that can't be represented in the current encoding scheme
  - Integer range limited by fixed width
  - Can occur in both the positive and negative directions
- C and Java ignore overflow exceptions
  - You end up with a bad value in your program and no warning/indication… oops!

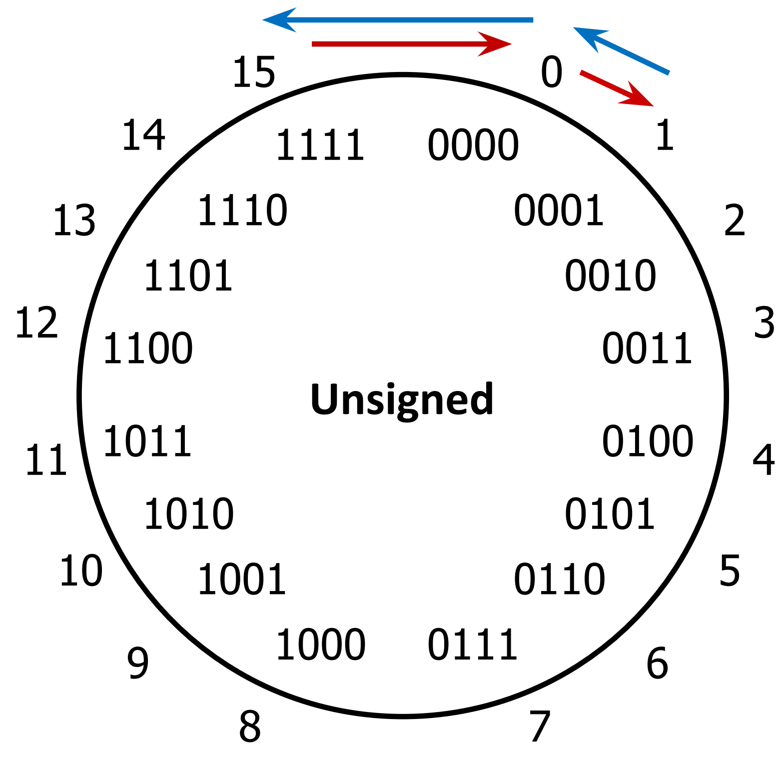
Overflow: Unsigned

- **Addition:** drop carry bit (-2^N)

```
  15        1111
+  2      + 0010
  17       10001
   1
```

- **Subtraction:** borrow ($+2^N$)

```
   1       10001
-  2      - 0010
  -1        1111
  15
```



$\pm 2^N$ because of modular arithmetic

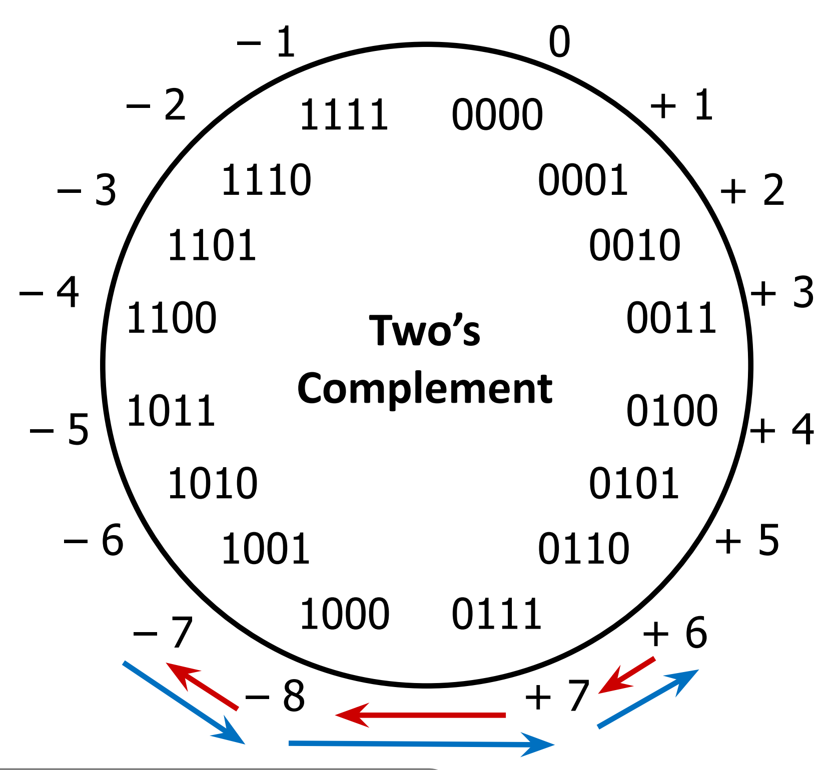
Overflow: Two's Complement

- **Addition:** (+) + (+) = (−) result?

```
   6        0110
+  3      + 0011
   9        1001
  −7
```

- **Subtraction:** (−) + (−) = (+)?

```
  −7        1001
−  3      − 0011
 −10        0110
   6
```



For signed: overflow if operands have same sign and result's sign is different

Practice Questions 2

- Assuming 8-bit integers:
  - `0x27` = 39 (signed) = 39 (unsigned)
  - `0xD9` = -39 (signed) = 217 (unsigned)
  - `0x7F` = 127 (signed) = 127 (unsigned)
  - `0x81` = -127 (signed) = 129 (unsigned)

- For the following additions, did signed and/or unsigned overflow occur?
  - 🧡 True 💚 False 🥶 Help!
  - **`0x27 + 0x81`**
  - **`0x7F + 0xD9`**

Practice Questions 2

- Assuming 8-bit integers:
  - `0x27` = 39 (signed) = 39 (unsigned)
  - `0xD9` = -39 (signed) = 217 (unsigned)
  - `0x7F` = 127 (signed) = 127 (unsigned)
  - `0x81` = -127 (signed) = 129 (unsigned)

- For the following additions, did signed and/or unsigned overflow occur?
  - 🧡 True 💚 False 🥶 Help!
  - **`0x27 + 0x81`**

Sign Extension

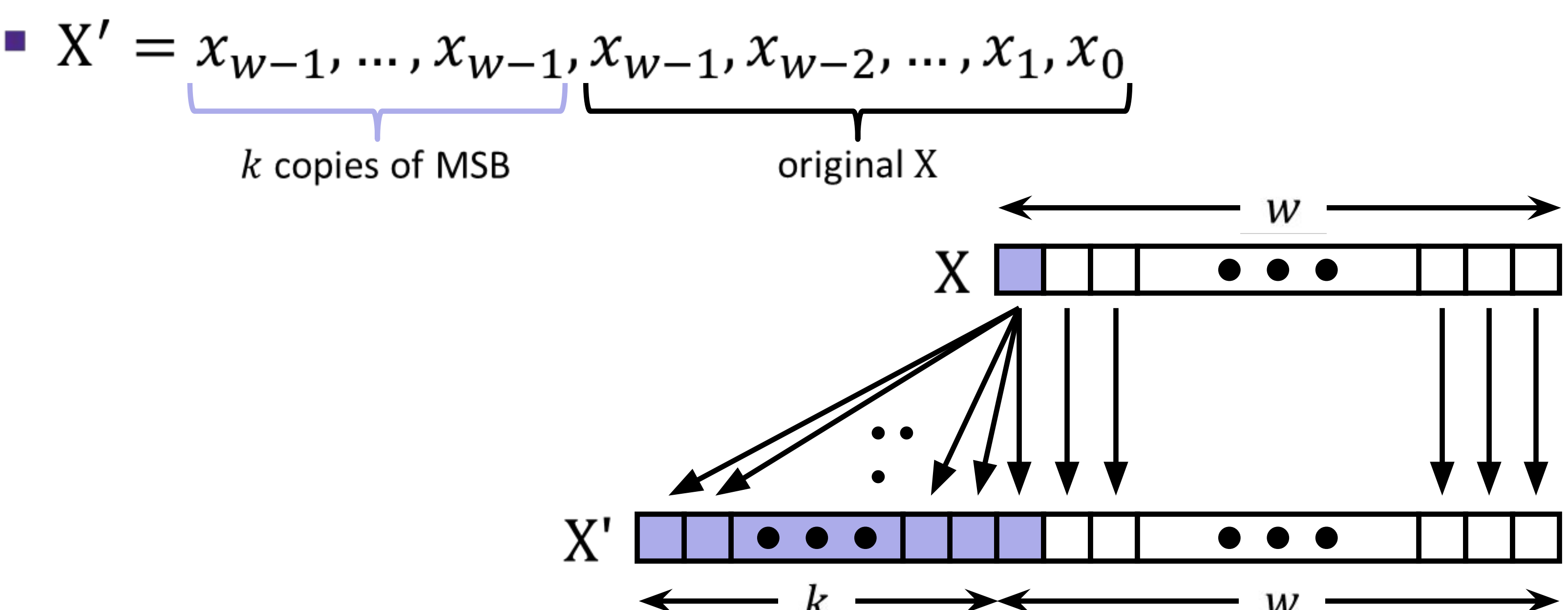
- What happens if you convert a *signed* integral data type to a larger one?
  - *e.g.* `char` → `short` → `int` → `long`
- **4-bit → 8-bit Example:**
  - Positive Case
    - ✓ Add 0's?
  - Negative Case?

4-bit: 0010 = +2

8-bit: 00000010 = +2

Sign Extension

- **Task:** Given a w-bit signed integer X, convert it to w+k-bit signed integer X′ *with the same value*
- **Rule: Add k copies of sign bit**
  - Let x_i be the i-th digit of X in binary



Sign Extension Example

- Convert from smaller to larger integral data types
- C automatically performs sign extension
  - Java too

```
short int x =   12345;
int      ix = (int) x;
short int y =  -12345;
int      iy = (int) y;
```

Var	Decimal	Hex	Binary
x	12345	30 39	00110000 00111001
ix	12345	00 00 30 39	00000000 00000000 00110000 00111001
y	-12345	CF C7	11001111 11000111
iy	-12345	FF FF CF C7	11111111 11111111 11001111 11000111

2021

How are we feeling about integers?

Aside: Unsigned Multiplication in C

Operands: w **bits** — u * v

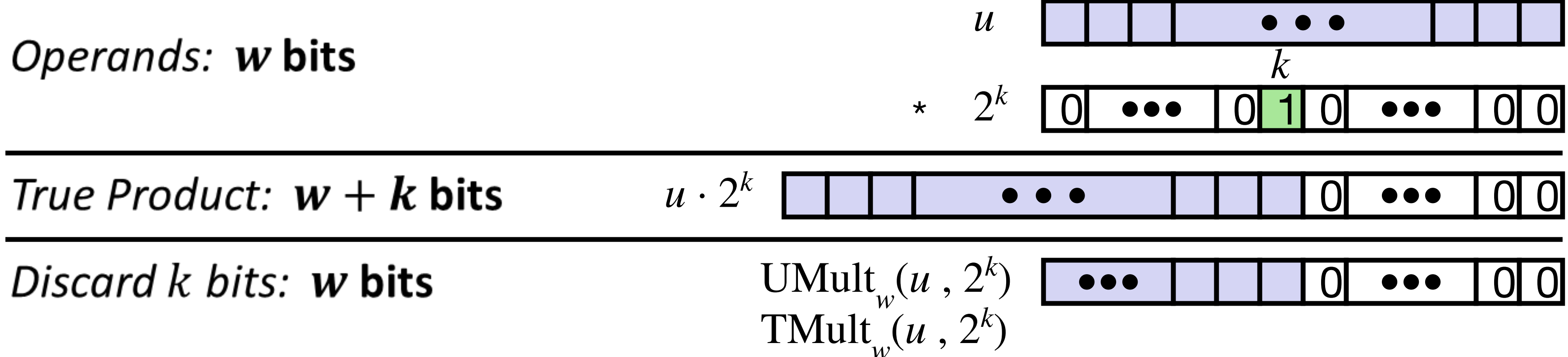
True Product: $2w$ **bits** — $u \cdot v$

Discard w bits: w **bits** — $\text{UMult}_w(u, v)$

- The same addition procedure works for both unsigned and two's complement integers
  - **Simplifies hardware:** only one algorithm for addition
  - **Algorithm:** simple addition, discard the highest carry bit
    - Called modular addition: result is sum *modulo* 2^w
- **4-bit Examples:**

Aside: Multiplication w/ shift & add

- Operation `u<<k` gives `u*`2^k
  - Both signed and unsigned



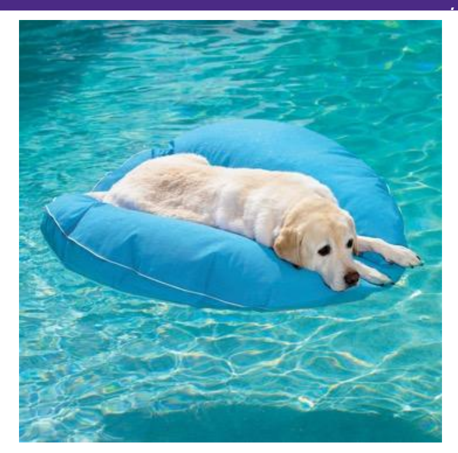
- Examples:
  - `u<<3     == u * 8`
  - `u<<5 - u<<3 == u * 24`
  - Most machines shift and add faster than multiply
    - ***Compiler generates this code automatically***

Number Representation Revisited

- What can we represent so far?
  - Signed and Unsigned Integers
  - Characters (ASCII, Unicode)
  - Addresses

- How do we encode the following:
  - Real numbers (*e.g.* 3.14159)
  - Very large numbers (*e.g.* 6.02×10^{23})
  - Very small numbers (*e.g.* 6.626×10^{-34})
  - Special numbers (*e.g.* ∞, NaN)

Floating Point Topics



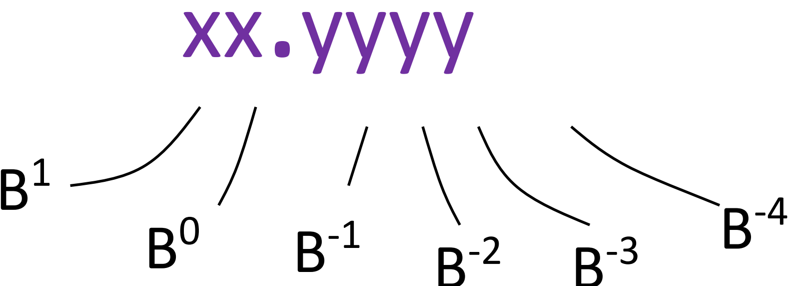
- **Fractional binary numbers**
- IEEE floating-point standard
- Floating-point operations and rounding
- Floating-point in C

- There are many more details that we won’t cover
  - It’s a 58-page “standard”…

Representation of Fractions

- Decimal Point (B=10)

Example 6-digit representation:



Representation of Fractions

- "Binary Point," like decimal point, signifies boundary between integer and fractional parts:

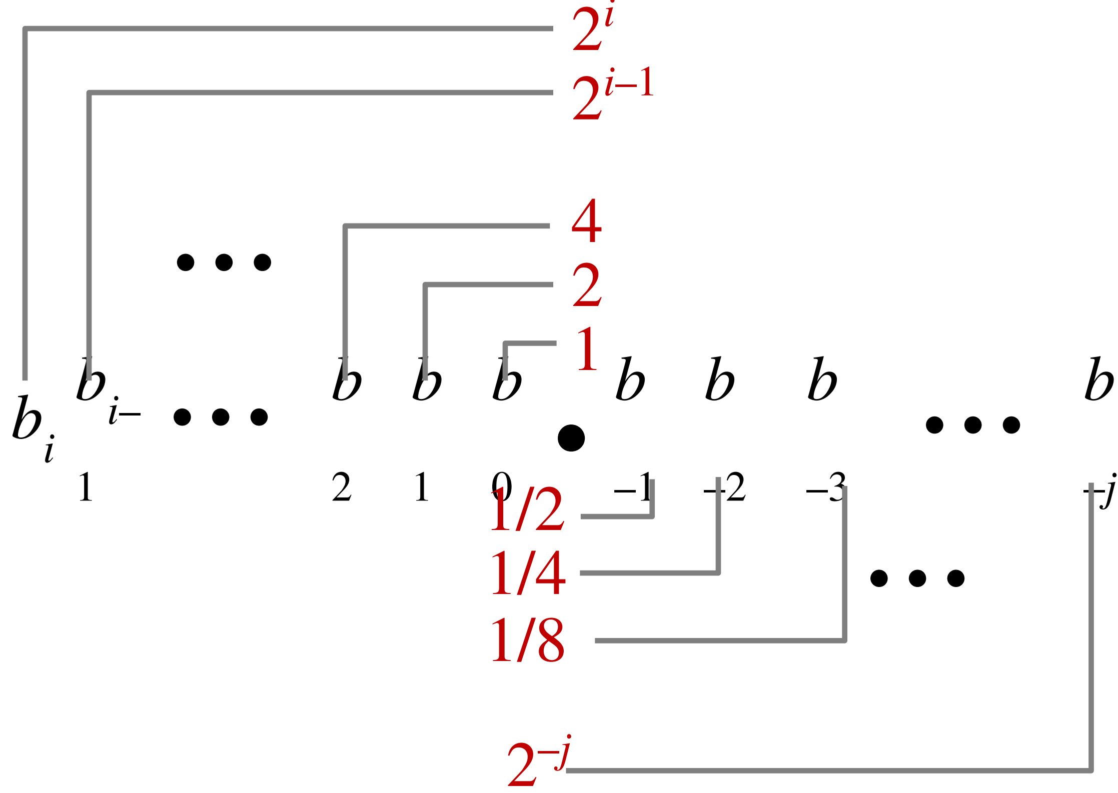
  Example 6-bit representation:

  xx.yyyy

  2^{1} 2^{0} 2^{-1} 2^{-2} 2^{-3} 2^{-4}

- Example: $10.1010_2 = 1\times2^{1} + 1\times2^{-1} + 1\times2^{-3} = 2.625_{10}$

Fractional Binary Numbers



- Representation
  - Bits to right of "binary point" == fractional powers of 2
  - Represents rational number: $\sum_{k=-j}^{i} b_k \cdot 2^k$

Fractional Binary Numbers

- Value Representation
  - 5 and 3/4 $\quad$ `101.11`$_2$
  - 2 and 7/8 $\quad$ `10.111`$_2$
  - 47/64 $\quad$ `0.101111`$_2$
- Observations
  - Shift left = multiply by power of 2
  - Shift right = divide by power of 2
  - Numbers of the form `0.111111`…$_2$ are just below 1.0
    - $1/2 + 1/4 + 1/8 + \ldots + 1/2^i + \ldots \rightarrow 1.0$
    - Use notation $1.0 - \varepsilon$

Limits of Representation

- Limitations:
  - Even given an arbitrary number of bits, can only **exactly** represent numbers of the form $x * 2^y$ (y can be negative)
  - Other rational numbers have repeating bit representations

Value: Binary Representation:

- $1/3 = 0.333333\ldots_{10} = 0.01010101[01]\ldots_2$
- $1/5 = 0.2 = 0.001100110011[0011]\ldots_2$
- $1/10 = 0.1 = 0.0001100110011[0011]\ldots_2$

Fixed Point Representation

- Implied binary point. Two example schemes:

  #1: the binary point is between bits 2 and 3

  $b_7\ b_6\ b_5\ b_4\ b_3\ [.]\ b_2\ b_1\ b_0$

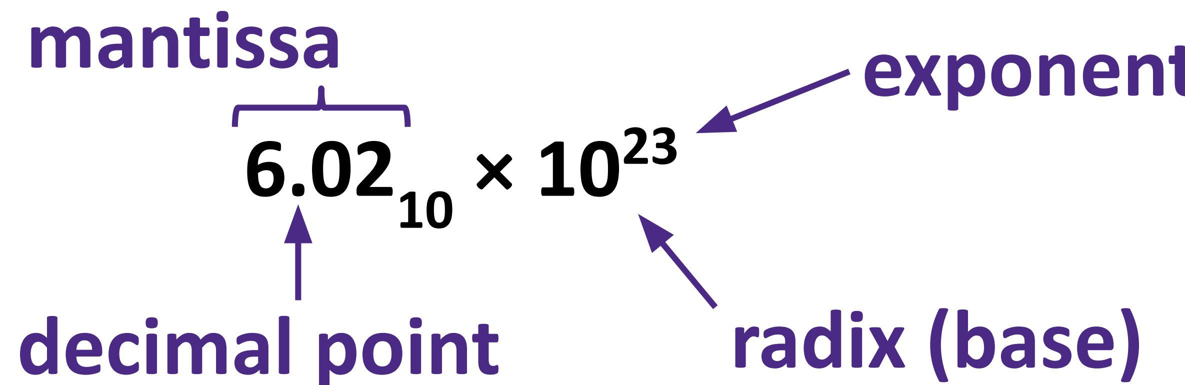
  #2: the binary point is between bits 4 and 5

  $b_7\ b_6\ b_5\ [.]\ b_4\ b_3\ b_2\ b_1\ b_0$
- Wherever we put the binary point, with fixed point representations there is a trade off between the amount of range and precision we have
- Fixed point = fixed *range* and fixed *precision*
  - range: difference between largest and smallest numbers possible
  - precision: smallest possible difference between any two numbers
- Hard to pick how much you need of each!

Floating Point Representation

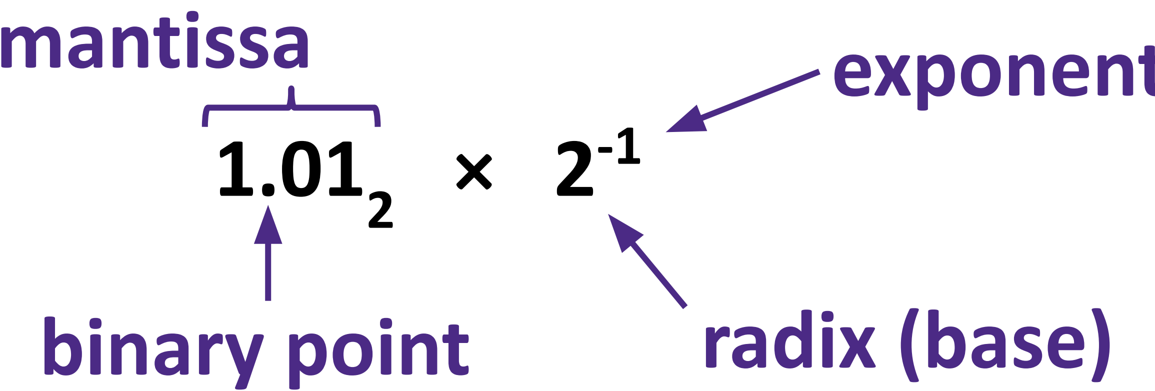
- Analogous to scientific notation
  - In Decimal:
    - Not 12000000, but 1.2×10^{7} In C: 1.2e7
    - Not 0.0000012, but 1.2×10^{-6} In C: 1.2e-6
  - In Binary:
    - Not 11000.000, but 1.1×2^{4}
    - Not 0.000101, but 1.01×2^{-4}
- We have to divvy up the bits we have among:
  - the sign (1 bit)
  - the mantissa (significand)
  - the exponent

Scientific Notation (Decimal)



- *Normalized form*: exactly one digit (non-zero) to left of decimal point
- Alternatives to representing 1/1,000,000,000
  - Normalized: 1.0×10^{-9}
  - Not normalized: $0.1 \times 10^{-8}, 10.0 \times 10^{-10}$

Scientific Notation (Binary)



- Computer arithmetic that supports this called floating point due to the "floating" of the binary point
  - Declare such variable in C as `float` (or `double`)

Scientific Notation Translation

- Convert from scientific notation to binary point
  - Perform the multiplication by shifting the decimal until the exponent disappears
    - Example: $1.011_2 \times 2^4 = 10110_2 = 22_{10}$
    - Example: $1.011_2 \times 2^{-2} = 0.01011_2 = 0.34375_{10}$
- Convert from binary point to *normalized* scientific notation
  - Distribute out exponents until binary point is to the right of a single digit
    - Example: $1101.001_2 = 1.101001_2 \times 2^3$
- **Practice:** Convert 11.375_{10} to normalized binary scientific notation

  1.011011×2^3

2021

How are we feeling about fixed point?

Technical Summary

- Sign and unsigned variables in C
  - Bit pattern remains the same, just *interpreted* differently
  - Strange things can happen with our arithmetic when we convert/cast between sign and unsigned numbers
    - Type of variables affects behavior of operators (shifting, comparison)
- We can only represent so much in w bits
  - When we exceed the limits, *arithmetic overflow* occurs
  - *Sign extension* tries to preserve value when expanding
- Floating point approximates real numbers
  - More details on Friday!

2021

What values were embedded in the first computing machine?

(Modern) Hardware: Historic View

- **Computer**: one who computes

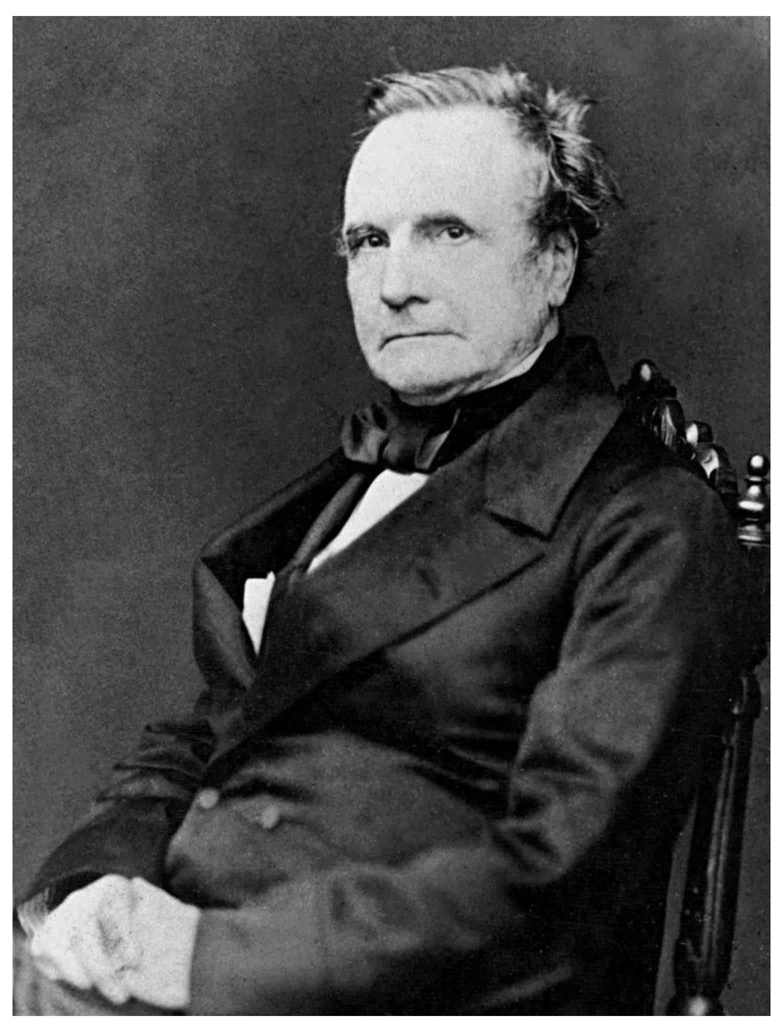
The women of Bletchley Park, Credit: BBC

  - Mostly white cis-women
  - “Boring, repetitive work”, doing math quickly

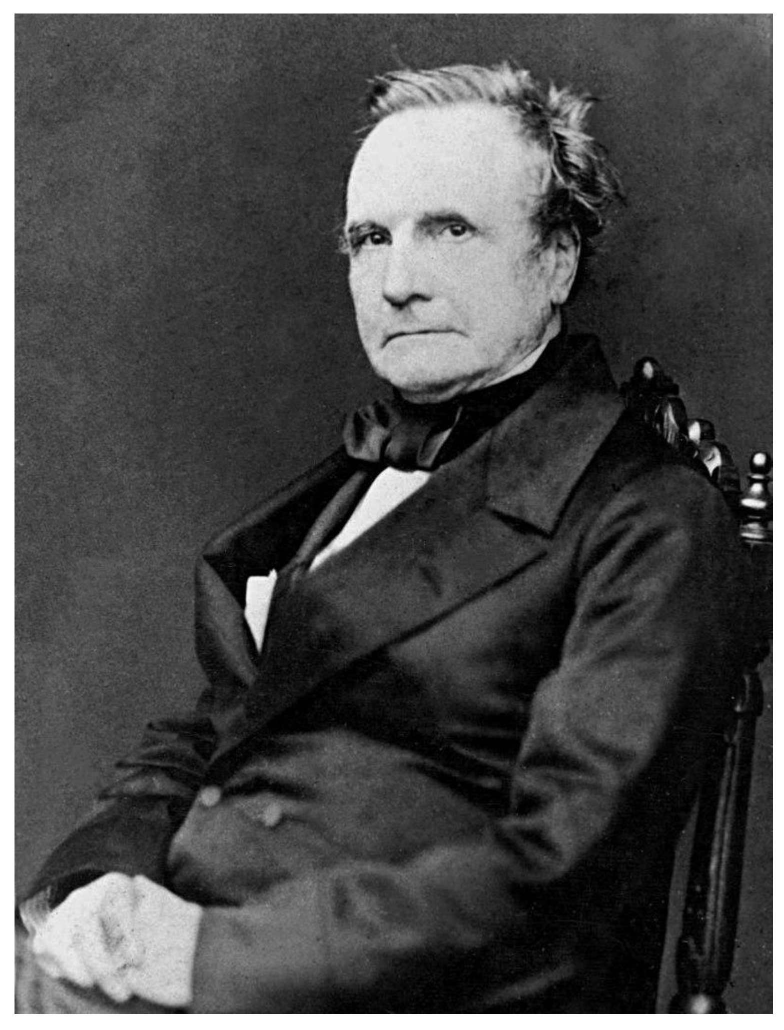
2021

But, let’s go further back!

The "Father" of Computing (~1820)



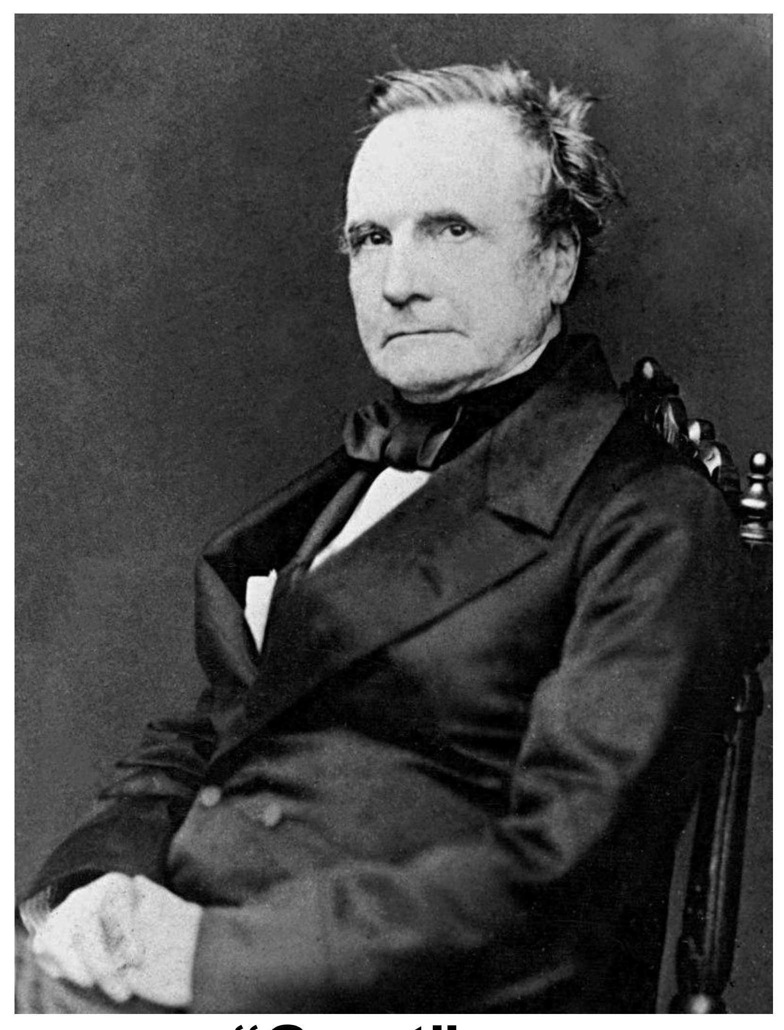
The "Father" of Computing (~1820)



Charles Babbage

Ada Lovelace

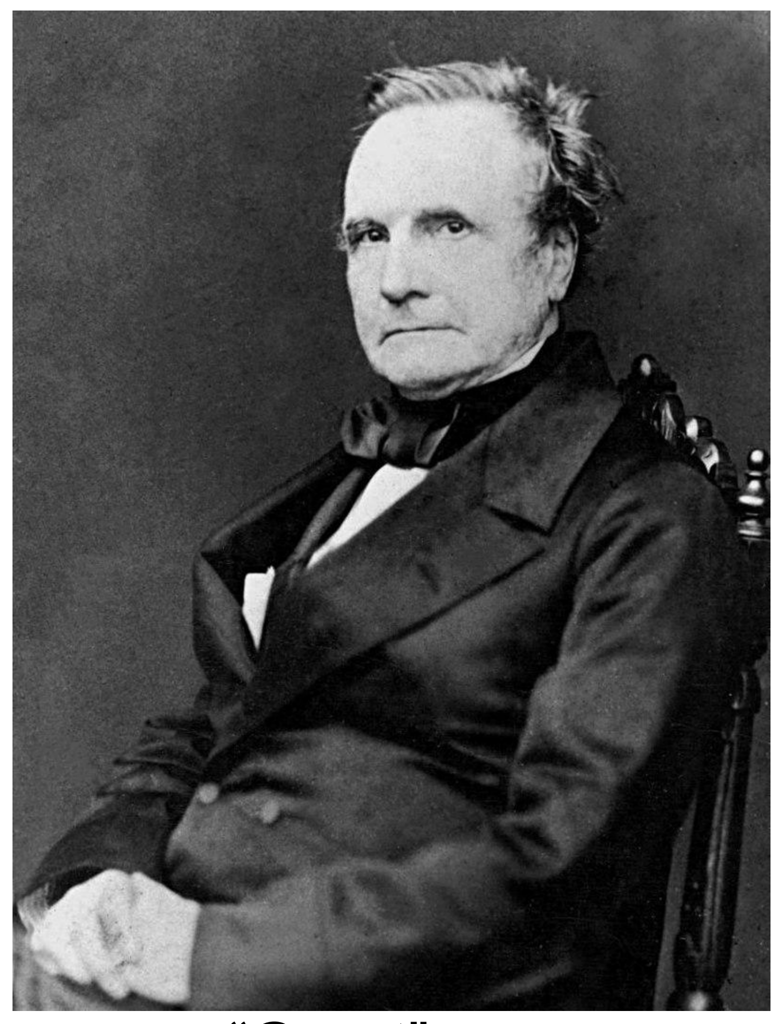
The “Father” of Computing (~1820)



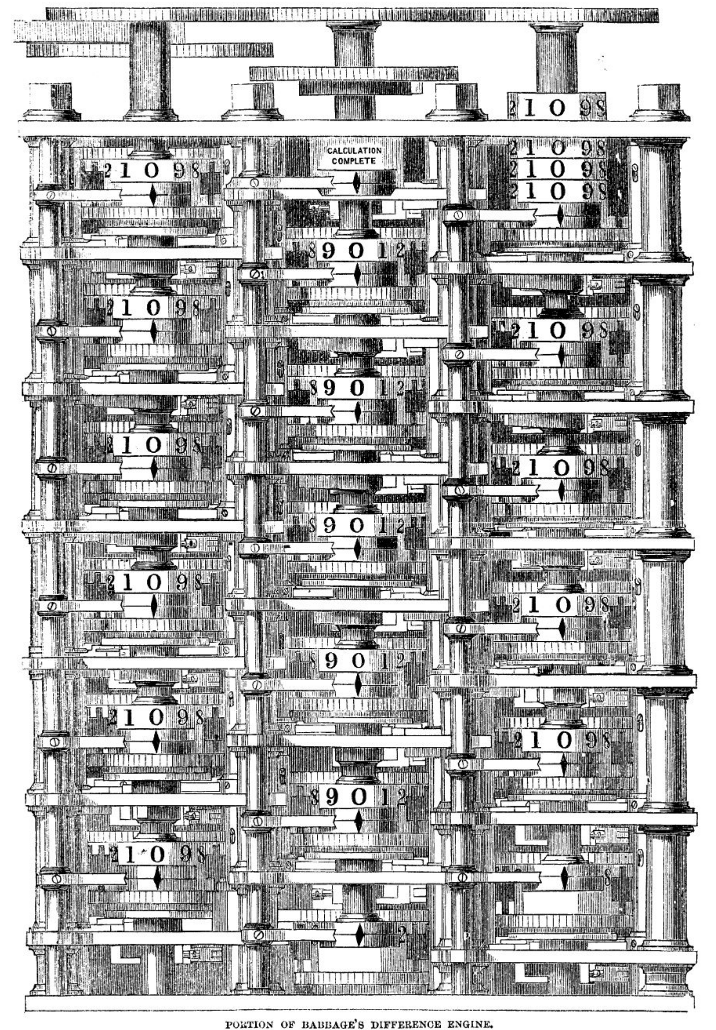
“Great” man

First Programmer!

The "Father" of Computing (~1820)



"Great" man



"Great" machine

2021

Why make an analytical engine?

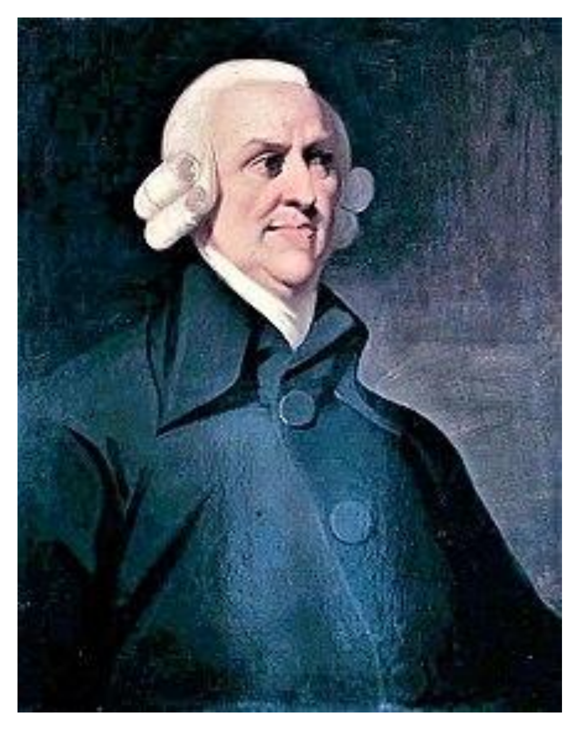
2021

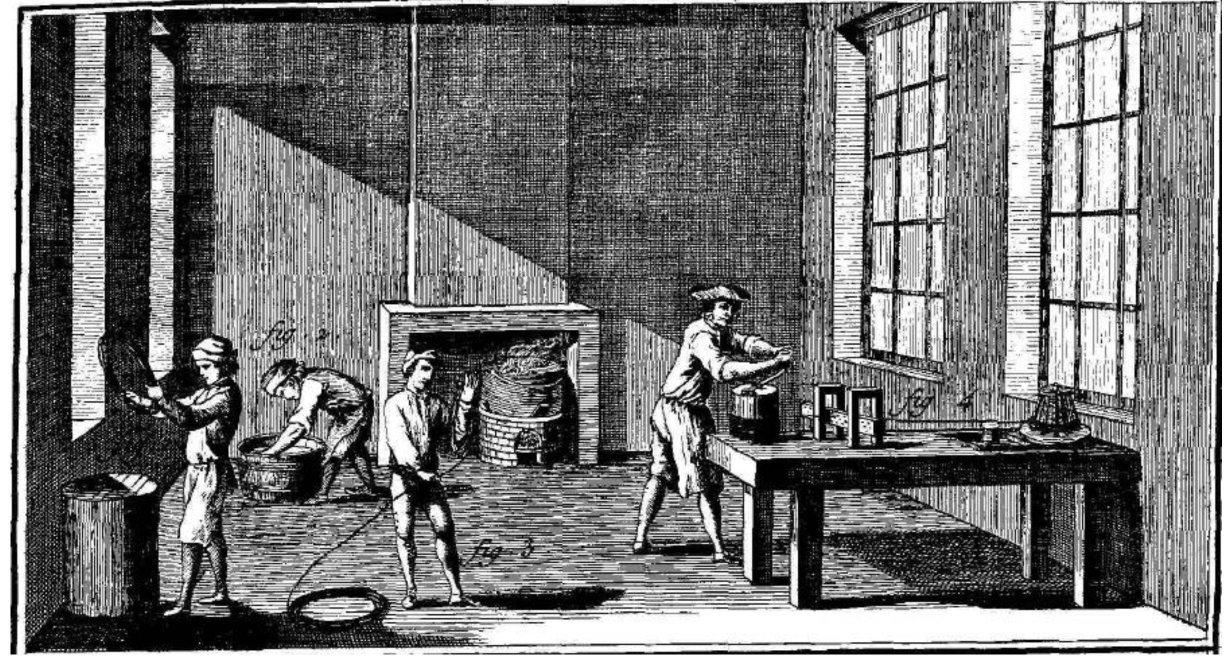
Why make an analytical engine?

Let's go further back!

The Division of Labor

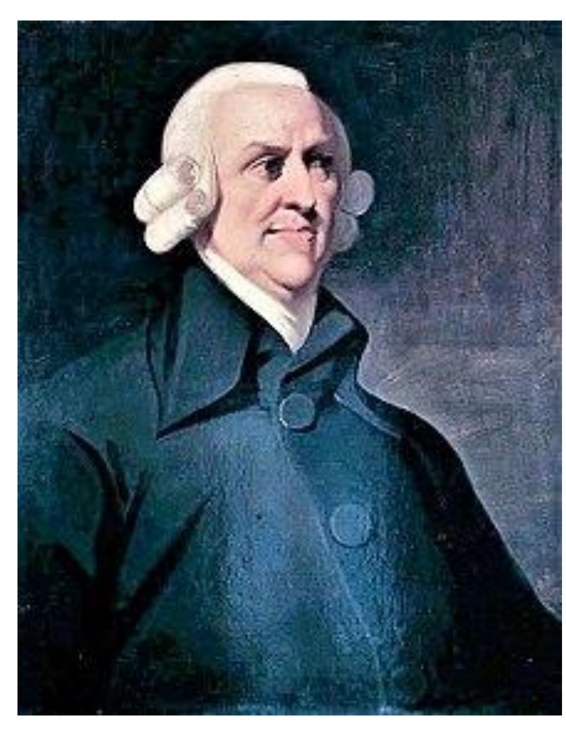
"The Wealth of Nations", 1776

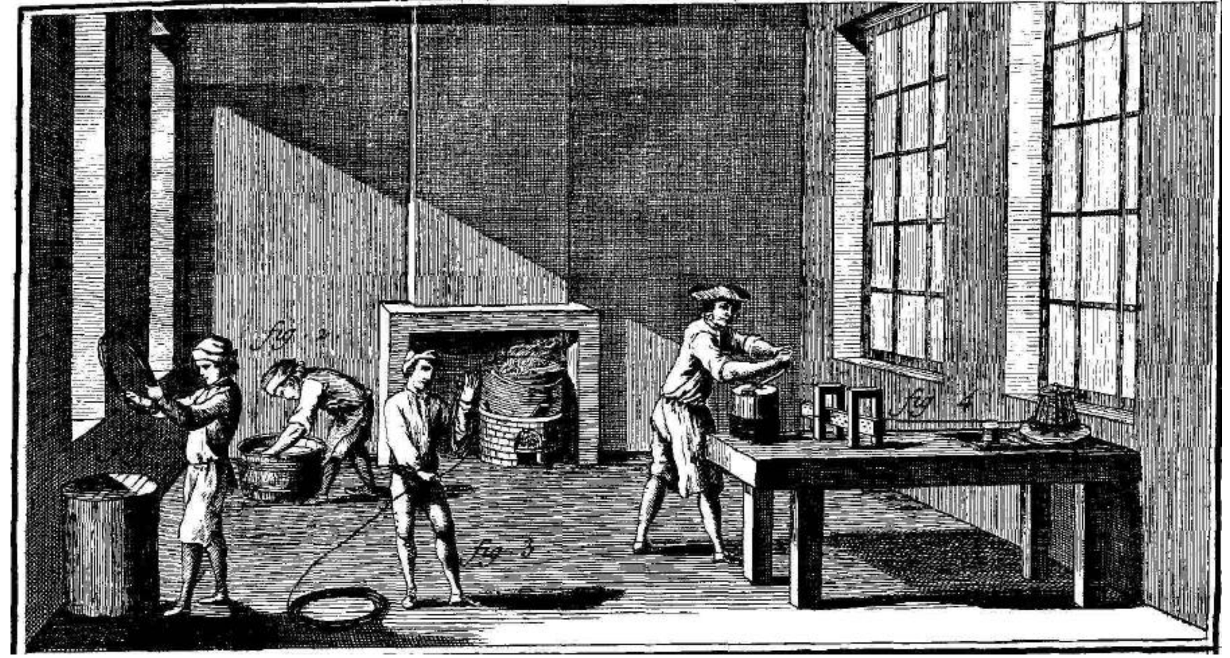




The Division of Labor (Adam Smith)

"The Wealth of Nations", 1776





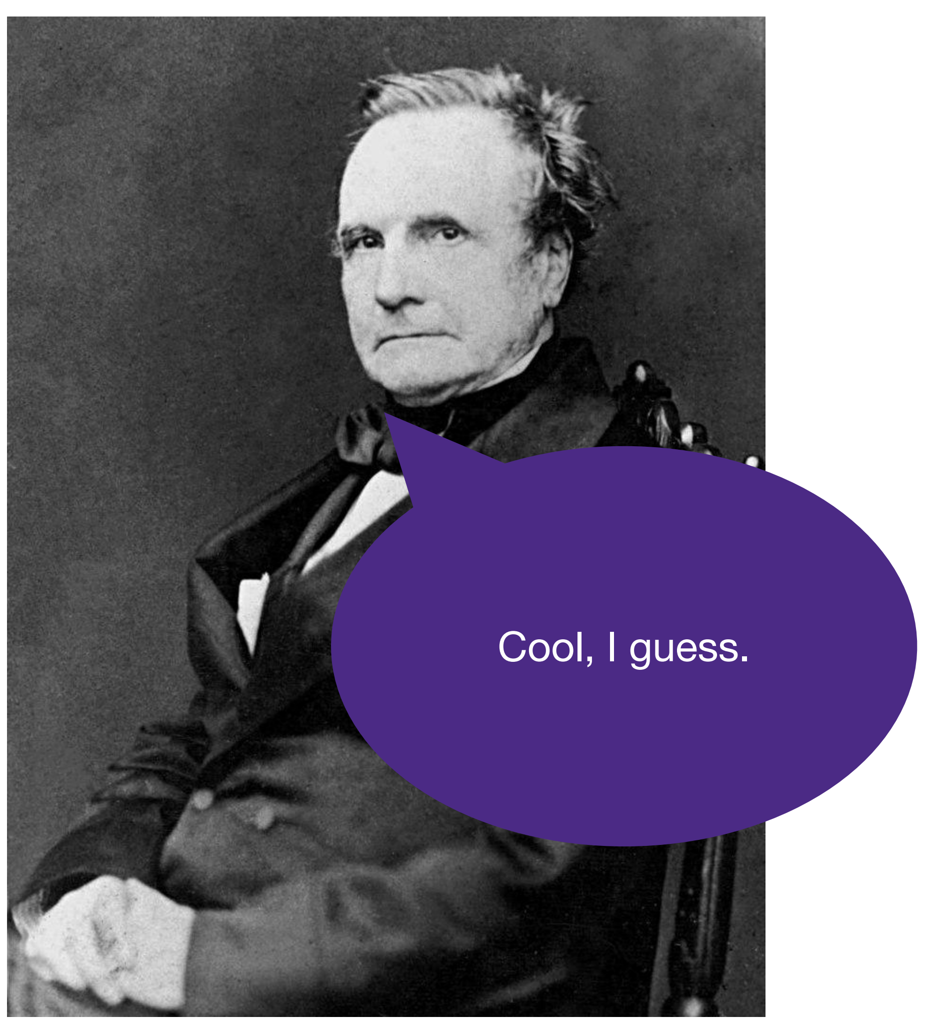
Making Butter

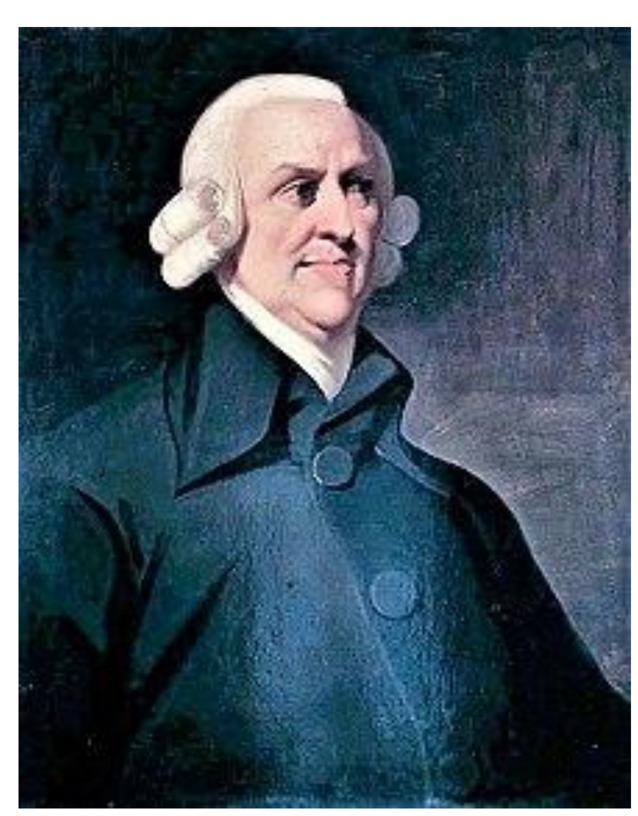
1. Milk cows
2. Turn milk into cream
3. Ferment cream
4. Churn cream
5. Strain milk
6. Get butter!

- Historically, all one person
- Adam Smith: it'll go faster if we specialize
- "Division of Labor"

Basically, specialization improves efficiency

Inspiring Babbage





Babbage, inspo by….

Babbage, inspo by Gaspard De Prony

- Applied division of labor to produce logarithms!

2021

Who wants butter, when you can have logarithms?

TABLE OF COMMON LOGARITHMS

N	0	1	2	3	4	5	6	7	8	9
10	0000	0043	0086	0128	0170	0212	0253	0294	0334	0374
11	0414	0453	0492	0531	0569	0607	0645	0682	0719	0755
12	0792	0828	0864	0899	0934	0969	1004	1038	1072	1106
13	1139	1173	1206	1239	1271	1303	1335	1367	1399	1430
14	1461	1492	1523	1553	1584	1614	1644	1673	1703	1732
15	1761	1790	1818	1847	1875	1903	1931	1959	1987	2014
16	2041	2068	2095	2122	2148	2175	2201	2227	2253	2279
17	2304	2330	2355	2380	2405	2430	2455	2480	2504	2529
18	2553	2577	2601	2625	2648	2672	2695	2718	2742	2765
19	2788	2810	2833	2856	2878	2900	2923	2945	2967	2989
20	3010	3032	3054	3075	3096	3118	3139	3160	3181	3201
21	3222	3243	3263	3284	3304	3324	3345	3365	3385	3404
22	3424	3444	3464	3483	3502	3522	3541	3560	3579	3598
23	3617	3636	3655	3674	3692	3711	3729	3747	3766	3784
24	3802	3820	3838	3856	3874	3892	3909	3927	3945	3962
25	3979	3997	4014	4031	4048	4065	4082	4099	4116	4133
26	4150	4166	4183	4200	4216	4232	4249	4265	4281	4298
27	4314	4330	4346	4362	4378	4393	4409	4425	4440	4456
28	4472	4487	4502	4518	4533	4548	4564	4579	4594	4609
29	4624	4639	4654	4669	4683	4698	4713	4728	4742	4757
30	4771	4786	4800	4814	4829	4843	4857	4871	4886	4900
31	4914	4928	4942	4955	4969	4983	4997	5011	5024	5038
32	5051	5065	5079	5092	5105	5119	5132	5145	5159	5172
33	5185	5198	5211	5224	5237	5250	5263	5276	5289	5302
34	5315	5328	5340	5353	5366	5378	5391	5403	5416	5428
35	5441	5453	5465	5478	5490	5502	5514	5527	5539	5551
36	5563	5575	5587	5599	5611	5623	5635	5647	5658	5670
37	5682	5694	5705	5717	5729	5740	5752	5763	5775	5786
38	5798	5809	5821	5832	5843	5855	5866	5877	5888	5899
39	5911	5922	5933	5944	5955	5966	5977	5988	5999	6010
40	6021	6031	6042	6053	6064	6075	6085	6096	6107	6117
41	6128	6138	6149	6160	6170	6180	6191	6201	6212	6222
42	6232	6243	6253	6263	6274	6284	6294	6304	6314	6325
43	6335	6345	6355	6365	6375	6385	6395	6405	6415	6425
44	6435	6444	6454	6464	6474	6484	6493	6503	6513	6522
45	6532	6542	6551	6561	6571	6580	6590	6599	6609	6618
46	6628	6637	6646	6656	6665	6675	6684	6693	6702	6712
47	6721	6730	6739	6749	6758	6767	6776	6785	6794	6803
48	6812	6821	6830	6839	6848	6857	6866	6875	6884	6893
49	6902	6911	6920	6928	6937	6946	6955	6964	6972	6981
50	6990	6998	7007	7016	7024	7033	7042	7050	7059	7067
51	7076	7084	7093	7101	7110	7118	7126	7135	7143	7152
52	7160	7168	7177	7185	7193	7202	7210	7218	7226	7235
53	7243	7251	7259	7267	7275	7284	7292	7300	7308	7316
54	7324	7332	7340	7348	7356	7364	7372	7380	7388	7396
N	0	1	2	3	4	5	6	7	8	9

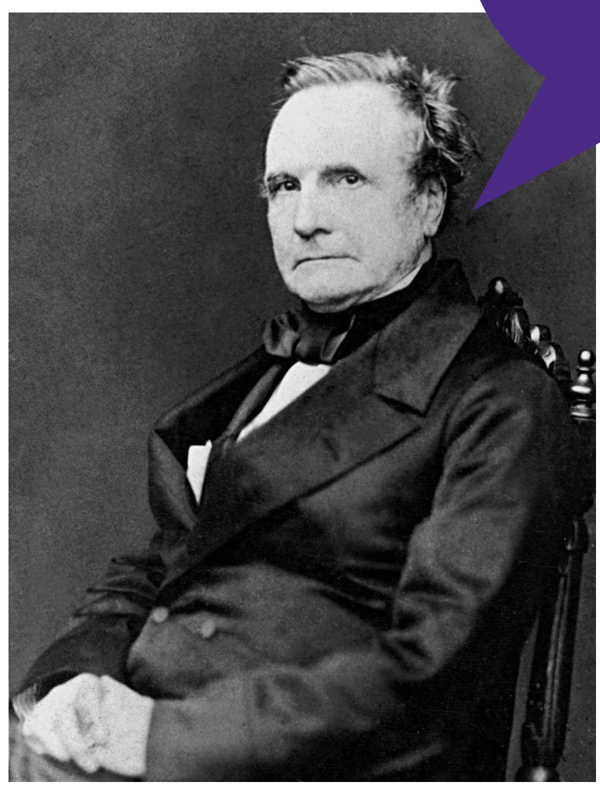
TABLE OF COMMON LOGARITHMS

N	0	1	2	3	4	5	6	7	8	9
55	7404	7412	7419	7427	7435	7443	7451	7459	7466	7474
56	7482	7490	7497	7505	7513	7520	7528	7536	7543	7551
57	7559	7566	7574	7582	7589	7597	7604	7612	7619	7627
58	7634	7642	7649	7657	7664	7672	7679	7686	7694	7701
59	7709	7716	7723	7731	7738	7745	7752	7760	7767	7774
60	7782	7789	7796	7803	7810	7818	7825	7832	7839	7846
61	7853	7860	7868	7875	7882	7889	7896	7903	7910	7917
62	7924	7931	7938	7945	7952	7959	7966	7973	7980	7987
63	7993	8000	8007	8014	8021	8028	8035	8041	8048	8055
64	8062	8069	8075	8082	8089	8096	8102	8109	8116	8122
65	8129	8136	8142	8149	8156	8162	8169	8176	8182	8189
66	8195	8202	8209	8215	8222	8228	8235	8241	8248	8254
67	8261	8267	8274	8280	8287	8293	8299	8306	8312	8319
68	8325	8331	8338	8344	8351	8357	8363	8370	8376	8382
69	8388	8395	8401	8407	8414	8420	8426	8432	8439	8445
70	8451	8457	8463	8470	8476	8482	8488	8494	8500	8506
71	8513	8519	8525	8531	8537	8543	8549	8555	8561	8567
72	8573	8579	8585	8591	8597	8603	8609	8615	8621	8627
73	8633	8639	8645	8651	8657	8663	8669	8675	8681	8686
74	8692	8698	8704	8710	8716	8722	8727	8733	8739	8745
75	8751	8756	8762	8768	8774	8779	8785	8791	8797	8802
76	8808	8814	8820	8825	8831	8837	8842	8848	8854	8859
77	8865	8871	8876	8882	8887	8893	8899	8904	8910	8915
78	8921	8927	8932	8938	8943	8949	8954	8960	8965	8971
79	8976	8982	8987	8993	8998	9004	9009	9015	9020	9025
80	9031	9036	9042	9047	9053	9058	9063	9069	9074	9079
81	9085	9090	9096	9101	9106	9112	9117	9122	9128	9133
82	9138	9143	9149	9154	9159	9165	9170	9175	9180	9186
83	9191	9196	9201	9206	9212	9217	9222	9227	9232	9238
84	9243	9248	9253	9258	9263	9269	9274	9279	9284	9289
85	9294	9299	9304	9309	9315	9320	9325	9330	9335	9340
86	9345	9350	9355	9360	9365	9370	9375	9380	9385	9390
87	9395	9400	9405	9410	9415	9420	9425	9430	9435	9440
88	9445	9450	9455	9460	9465	9469	9474	9479	9484	9489
89	9494	9499	9504	9509	9513	9518	9523	9528	9533	9538
90	9542	9547	9552	9557	9562	9566	9571	9576	9581	9586
91	9590	9595	9600	9605	9609	9614	9619	9624	9628	9633
92	9638	9643	9647	9652	9657	9661	9666	9671	9675	9680
93	9685	9689	9694	9699	9703	9708	9713	9717	9722	9727
94	9731	9736	9741	9745	9750	9754	9759	9763	9768	9773
95	9777	9782	9786	9791	9795	9800	9805	9809	9814	9818
96	9823	9827	9832	9836	9841	9845	9850	9854	9859	9863
97	9868	9872	9877	9881	9886	9890	9894	9899	9903	9908
98	9912	9917	9921	9926	9930	9934	9939	9943	9948	9952
99	9956	9961	9965	9969	9974	9978	9983	9987	9991	9996
N	0	1	2	3	4	5	6	7	8	9

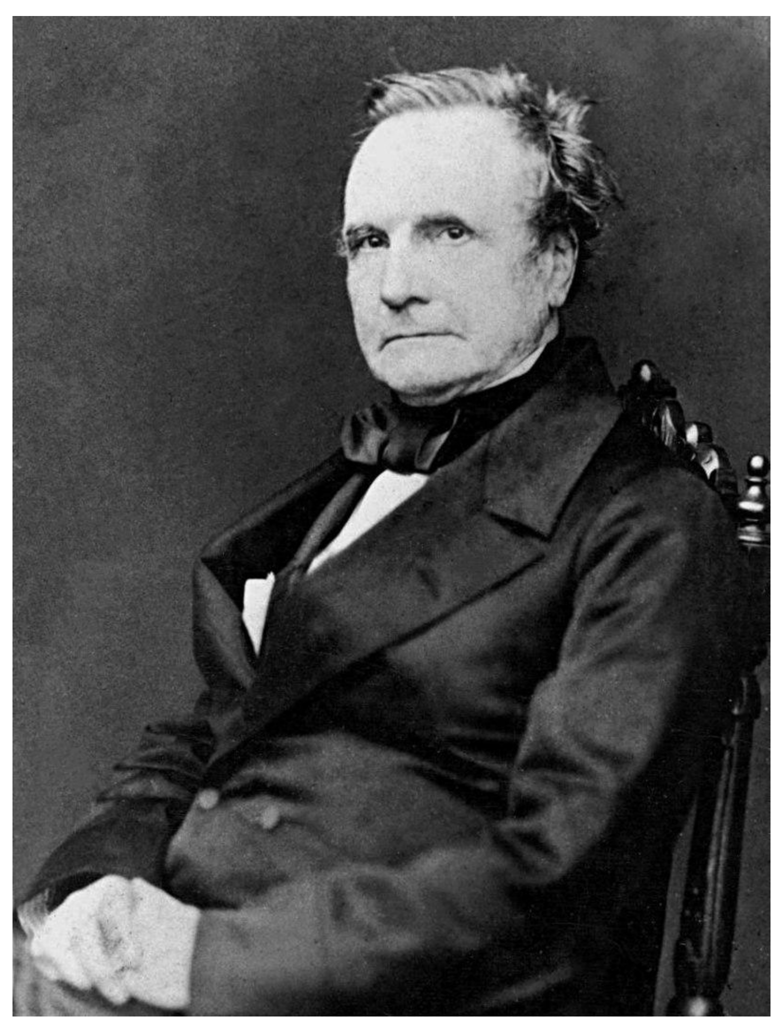
Babbage, inspo by Gaspard De Prony

- Applied division of labor to produce logarithmic tables
- "…manufacture logarithms as one manufactures pins"
- 5 experts, 8 managers, 70 human computers

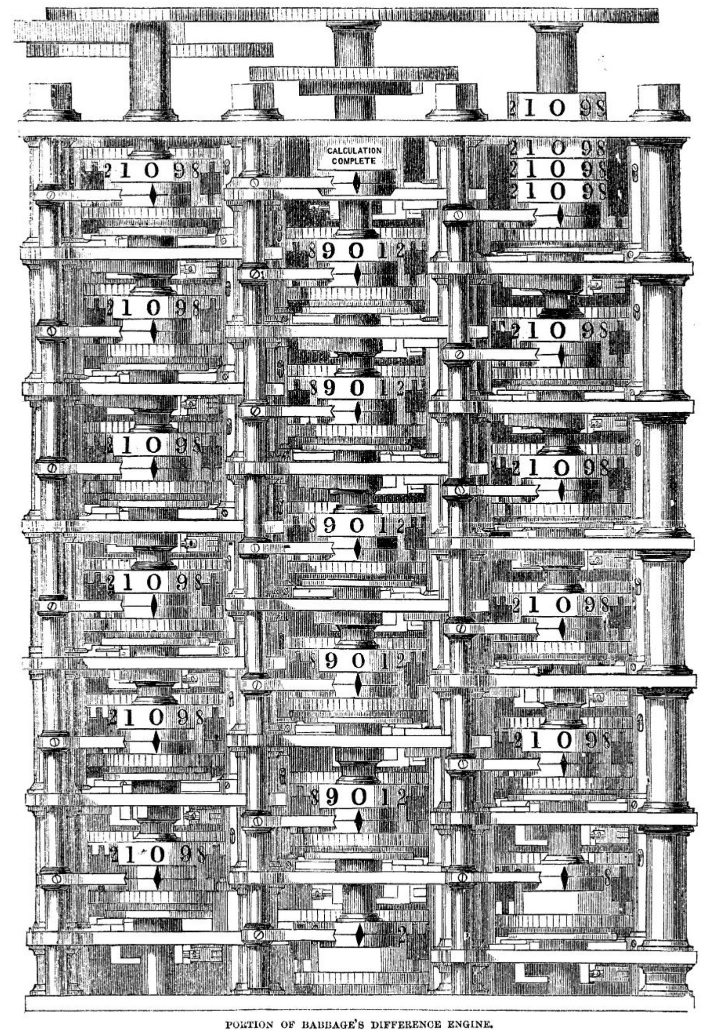
But what if we could automate the human computers?



The “Father” of Computing (~1820)

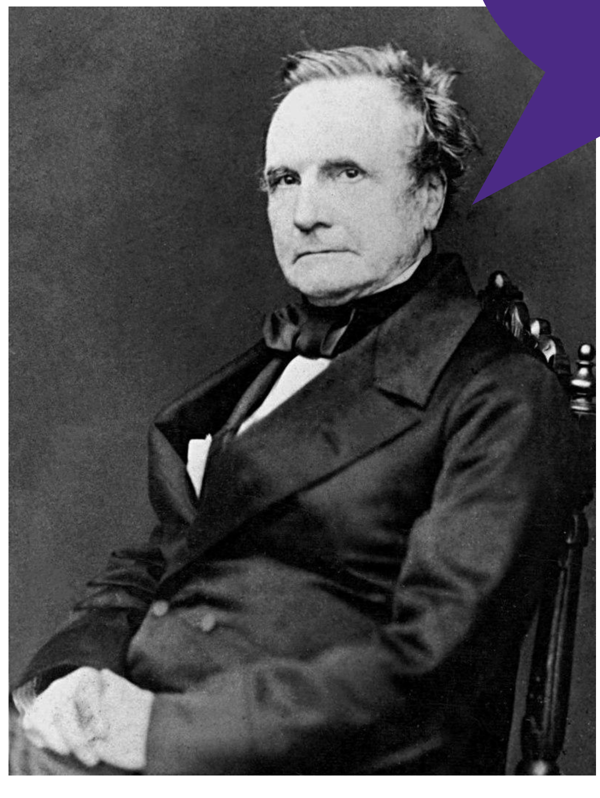


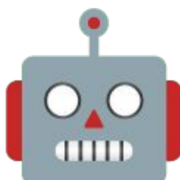
“Great” man



“Great” machine

But what if we could automate the human computers?





What about the hairdressers, turned human computers?

We've seen this line of thinking before!

Automation/Augmentation

- “Boring, repetitive work” should be automated
  - This tends to eliminate jobs for marginalized folks
- ”Boring, repetitive work” is robot work
  - Performed by non-humans, or ”less humans”
- Thematic through history of computing
  - Even going back to ”the father of computing”
  - Of course, this includes modern incarnations
  - Computing has been “automating” for a while!

UBER
ADVANCED TECHNOLOGIES CENTER

JOHN DEERE

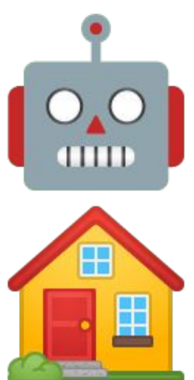
Why are we automating?

- Efficiency of labor – robots don't need healthcare or time off, and are frequently faster
  - Industrialized mechanization: efficiency for profit
- Efficiency of conflict – robots can kill remotely
  - Also, quicker calculation of ballistics, other math
  - Militarization: efficiency for safety (i.e. win faster)

If anyone's studied Marx, feel free to chime in!

Charles Babbage

- Credits De Prony with inspiration
- Coins ***Babbage Principle***
  - ”High Skilled” workers regularly perform tasks below their abilities, yet are paid a “high-skill” wage. We should effectively divide labor so that no one paid a “high-skill” wage was doing “low-skill” work.
- Does this sound familiar to anyone?



CS views on labor

- What work does CS consider “high skill”?
  - Programming, obv.
  - Others?
- **What work does CS consiver “low skill”?**

2021

What values were embedded in the first computing machine?

2021

Efficiency!

Computing tends to value efficiency over humanity!

The Hardware/Software Interface consistently reflects this!

2021

A reminder: your self-worth isn't tied to your efficiency!

BONUS SLIDES

Some examples of using shift operators in combination with bitmasks, which you may find helpful for Lab 1.

- Extract the 2nd most significant byte of an `int`
- Extract the sign bit of a signed `int`
- Conditionals as Boolean expressions

Using Shifts and Masks

- Extract the 2nd most significant *byte* of an `int`:
  - First shift, then mask: `(x>>16) & 0xFF`

x	00000001 00000010 00000011 00000100
x>>16	00000000 00000000 00000001 00000010
0xFF	00000000 00000000 00000000 11111111
(x>>16) & 0xFF	00000000 00000000 00000000 00000010

  - Or first mask, then shift: `(x & 0xFF0000)>>16`

x	00000001 00000010 00000011 00000100
0xFF0000	00000000 11111111 00000000 00000000
x & 0xFF0000	00000000 00000010 00000000 00000000
(x&0xFF0000)>>16	00000000 00000000 00000000 00000010

Using Shifts and Masks

- Extract the *sign bit* of a signed `int`:
  - First shift, then mask: `(x>>31) & 0x1`
    - Assuming arithmetic shift here, but this works in either case
    - Need mask to clear `1`s possibly shifted in

x	**0**0000001 00000010 00000011 00000100
x>>31	00000000 00000000 00000000 0000000**0**
0x1	00000000 00000000 00000000 00000001
(x>>31) & 0x1	00000000 00000000 00000000 00000000

x	**1**0000001 00000010 00000011 00000100
x>>31	11111111 11111111 11111111 1111111**1**
0x1	00000000 00000000 00000000 00000001
(x>>31) & 0x1	00000000 00000000 00000000 00000001

Using Shifts and Masks

- Conditionals as Boolean expressions
  - For `int x`, what does `(x<<31)>>31` do?

`x=!!123`	`00000000 00000000 00000000 00000001`
`x<<31`	`10000000 00000000 00000000 00000000`
`(x<<31)>>31`	`11111111 11111111 11111111 11111111`
`!x`	`00000000 00000000 00000000 00000000`
`!x<<31`	`00000000 00000000 00000000 00000000`
`(!x<<31)>>31`	`00000000 00000000 00000000 00000000`

- Can use in place of conditional:
  - In C: `if(x) {a=y;} else {a=z;}` equivalent to `a=x?y:z;`
  - `a=(((x<<31)>>31)&y) | (((!x<<31)>>31)&z);`