BITS, BYTES, AND INTEGERS

CS 045

Computer Organization and Architecture

Prof. Donald J. Patterson

Adapted from Bryant and O'Hallaron,
Computer Systems:
A Programmer's Perspective, Third Edition

BOOLEAN ALGEBRA

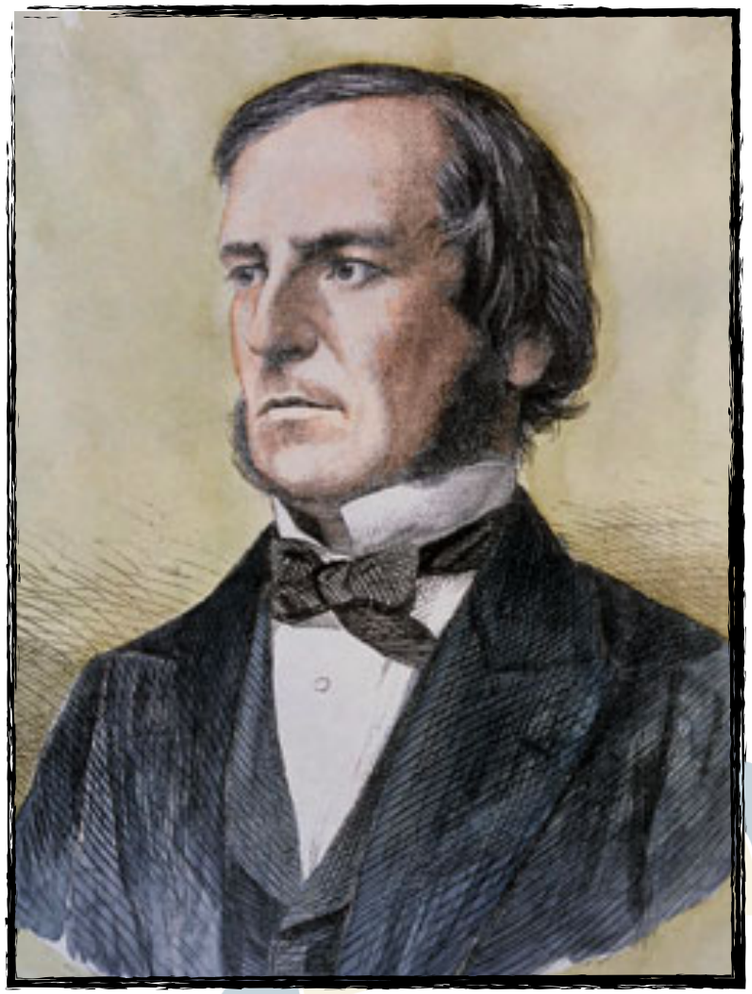
ORIGINS

- Credited to George Boole (1815-1864)
  - Laws of Thought (1854)

BOOLEAN ALGEBRA

ORIGINS

- Credited to George Boole (1815-1864)
  - Laws of Thought (1854)

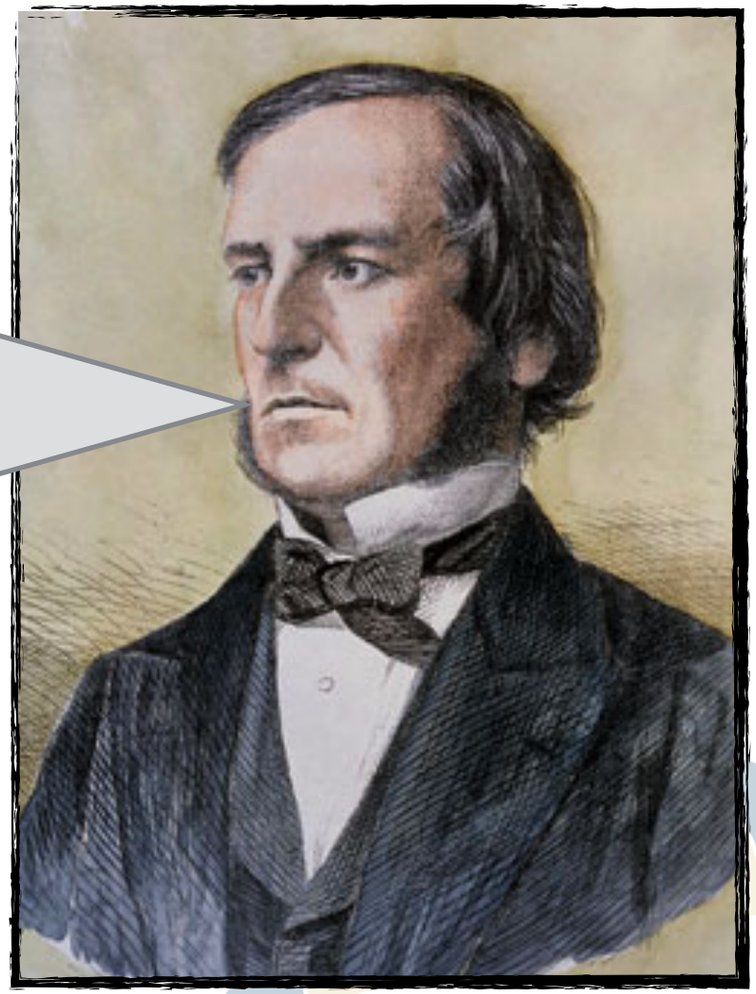


BOOLEAN ALGEBRA

ORIGINS

- Credited to George Boole (1815-1864)
  - Laws of Thought (1854)

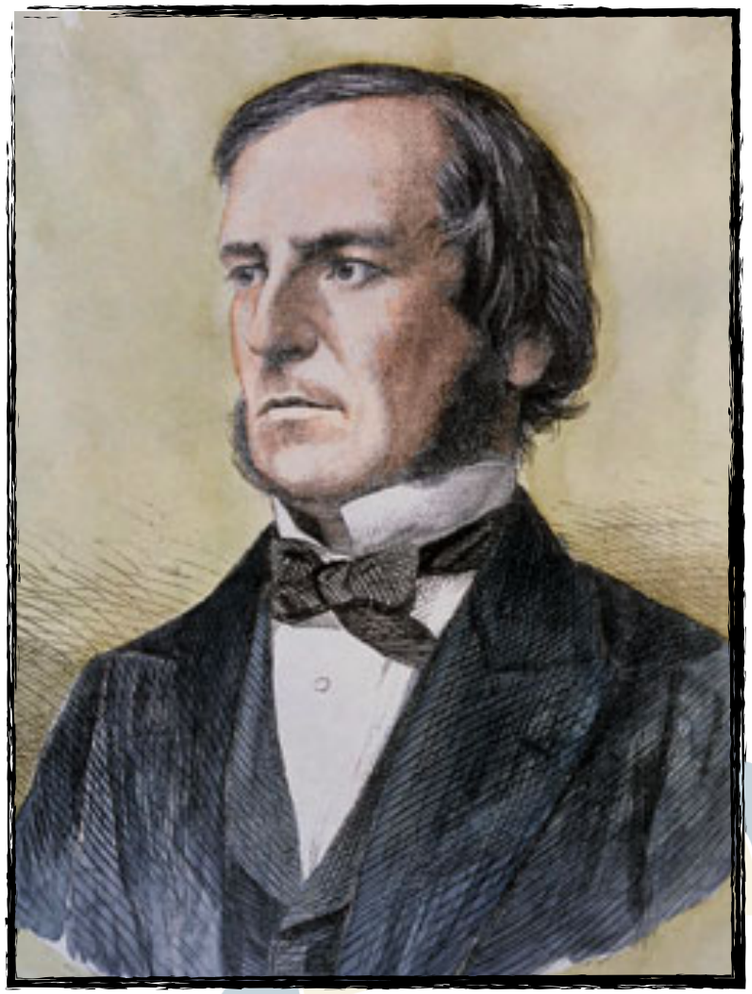
> "No general method for the solution of questions in the theory of probabilities can be established which does not explicitly recognise, not only the special numerical bases of the science, but also those universal laws of thought which are the basis of all reasoning, and which, whatever they may be as to their essence, are at least mathematical as to their form."



BOOLEAN ALGEBRA

ORIGINS

- Credited to George Boole (1815-1864)
  - Laws of Thought (1854)



BOOLEAN ALGEBRA

ORIGINS

- Credited to George Boole (1815-1864)
  - Laws of Thought (1854)

BOOLEAN ALGEBRA

ORIGINS

- Credited to George Boole (1815-1864)
  - Laws of Thought (1854)

- He compared the Trinity to the 3 axes of (x,y,z)

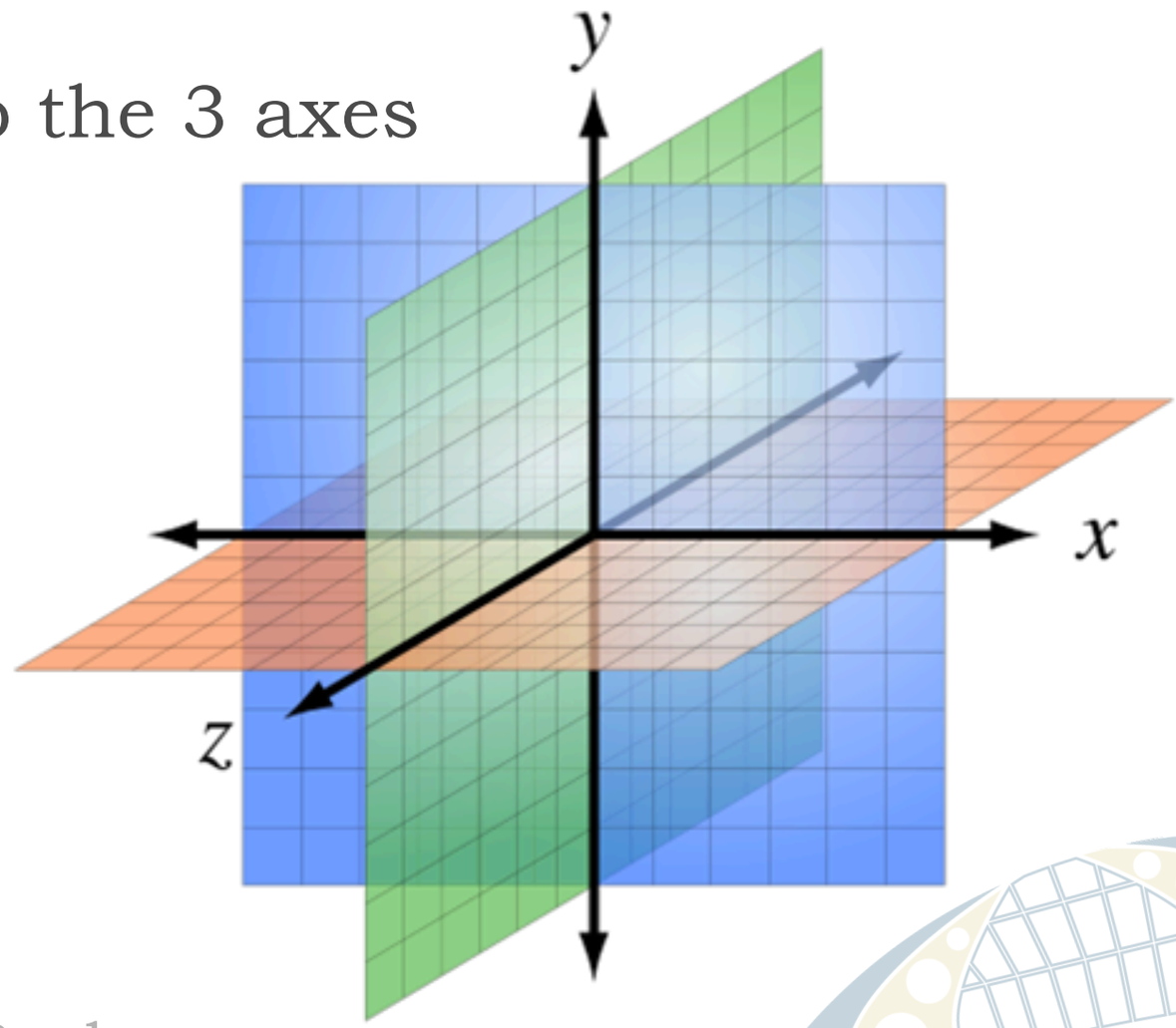
Wikipedia entry on Boole

BOOLEAN ALGEBRA

ORIGINS

- Credited to George Boole (1815-1864)
  - Laws of Thought (1854)

- He compared the Trinity to the 3 axes of (x,y,z)



Wikipedia entry on Boole

BOOLEAN ALGEBRA

AXIOMS

- represent
  - “1” as true
  - “0” as false

BOOLEAN ALGEBRA

FOUR FOUNDATIONAL OPERATIONS

- AND (&)
  - A&B = 1 when both A=1 and B=1

&	0	1
0	0	0
1	0	1

BOOLEAN ALGEBRA

FOUR FOUNDATIONAL OPERATIONS

- OR (|)
  - A|B = 1 when either A=1 or B=1

|	0	1
0	0	1
1	1	1

BOOLEAN ALGEBRA

FOUR FOUNDATIONAL OPERATIONS

- NOT (~)
  - ~A = 1 when A=0

~	
0	1
1	0

BOOLEAN ALGEBRA

FOUR FOUNDATIONAL OPERATIONS

- Exclusive-OR XOR (^)
  - A^B = 1 when either A=1 or B=1, but not both

^	0	1
0	0	1
1	1	0

BOOLEAN ALGEBRA

FOUR FOUNDATIONAL OPERATIONS

&	0	1
0	0	0
1	0	1

|	0	1
0	0	1
1	1	1

~	
0	1
1	0

^	0	1
0	0	1
1	1	0

BOOLEAN ALGEBRA

THESE CAN BE APPLIED TO BIT-VECTORS

- Operations can be applied bit-wise
  - All the properties of Boolean Algebra apply

```
  01101001          01101001          01101001
& 01010101        | 01010101        ^ 01010101
----------        ----------        ----------
  01000001          01111101          00111100

~ 01010101
----------
  10101010
```

BOOLEAN ALGEBRA

THESE CAN BE APPLIED TO BIT-VECTORS

- Operations can be applied bit-wise
  - All the properties of Boolean Algebra apply

```
  01101001
& 01010101
  01000001
```

```
  01101001
| 01010101
  01111101
```

```
  01101001
^ 01010101
  00111100
```

```
~ 01010101
  10101010
```

BOOLEAN ALGEBRA

THESE CAN BE APPLIED TO BIT-VECTORS

- Operations can be applied bit-wise
  - All the properties of Boolean Algebra apply

```
  01101001      01101001      01101001      01101001
& 01010101    | 01010101    ^ 01010101    ~ 01010101
----------    ----------    ----------    ----------
  01000001      01111101      00111100      10101010
```

BOOLEAN ALGEBRA

THESE CAN BE APPLIED TO BIT-VECTORS

- Operations can be applied bit-wise
  - All the properties of Boolean Algebra apply

```
  01101001      01101001      01101001
& 01010101    | 01010101    ^ 01010101
----------    ----------    ----------
  01000001      01111101      00111100
```

```
~ 01010101
----------
  10101010
```

BOOLEAN ALGEBRA

THESE CAN BE APPLIED TO BIT-VECTORS

- Operations can be applied bit-wise
  - All the properties of Boolean Algebra apply

```
  01101001      01101001      01101001
& 01010101    | 01010101    ^ 01010101    ~ 01010101
----------    ----------    ----------    ----------
  01000001      01111101      00111100      10101010
```

BIT-VECTORS CAN REPRESENT SETS

- a w-bit bit-vector can encode the membership of w elements in a set
- $a_j = 1$ if $j \in A$

- 01101001 { 0, 3, 5, 6 }
- 76543210

- 01010101 { 0, 2, 4, 6 }
- 76543210

BOOLEAN ALGEBRA

BIT-VECTORS CAN REPRESENT SETS

- 01101001 { 0, 3, 5, 6 }
- 76543210

- 01010101 { 0, 2, 4, 6 }
- 76543210

- Operations
  - & Intersection 01000001 { 0, 6 }
  - | Union 01111101 { 0, 2, 3, 4, 5, 6 }
  - ^ Symmetric difference 00111100 { 2, 3, 4, 5 }
  - ~ Complement 10101010 { 1, 3, 5, 7 }

BOOLEAN ALGEBRA IN C

- Operations &, |, ~, ^ Available in C
  - Apply to any “integral” data type
    - `long`, `int`, `short`, `char`, `unsigned`
  - View arguments as bit vectors
  - Arguments applied bit-wise

BOOLEAN ALGEBRA

BOOLEAN ALGEBRA IN C

- Examples (Char data type)
  - `~0x41 → 0xBE`
  - $\texttt{~01000001}_2 \rightarrow \texttt{10111110}_2$
  - `~0x00 → 0xFF`
  - $\texttt{~00000000}_2 \rightarrow \texttt{11111111}_2$
  - `0x69 & 0x55 → 0x41`
  - $\texttt{01101001}_2 \texttt{ \& } \texttt{01010101}_2 \rightarrow \texttt{01000001}_2$
  - `0x69 | 0x55 → 0x7D`
  - $\texttt{01101001}_2 \texttt{ | } \texttt{01010101}_2 \rightarrow \texttt{01111101}_2$

BOOLEAN ALGEBRA

CONTRAST TO LOGIC OPERATIONS IN C

- Contrast to Logical Operators
  - &&, ||, !
    - View 0 as “False”
    - Anything nonzero as “True”
    - Always return 0 or 1
    - Early termination

BOOLEAN ALGEBRA

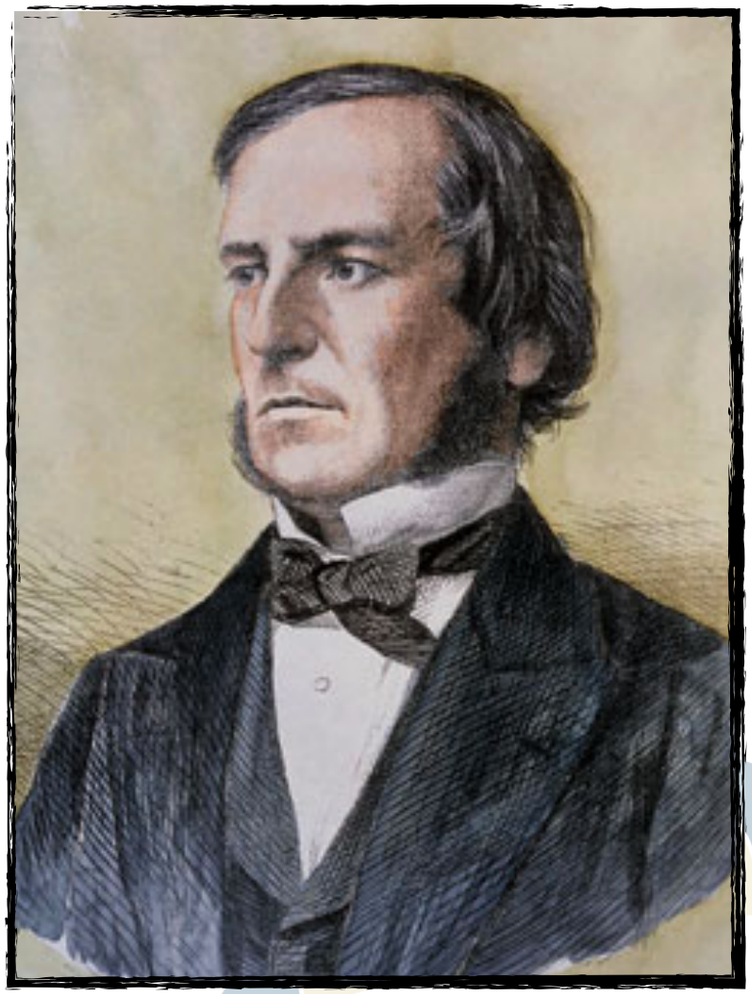
CONTRAST TO LOGIC OPERATIONS IN C

- Examples (char data type)
  - `!0x41  →  0x00`
  - `!0x00  →  0x01`
  - `!!0x41  →  0x01`
  - `0x69 && 0x55  →  0x01`
  - `0x69 || 0x55  →  0x01`
  - `p && *p   (avoids null pointer access)`

BOOLEAN ALGEBRA

CONTRAST TO LOGIC OPERATIONS IN C

- Examples (char data type)
  - `!0x41  →  0x00`
  - `!0x00  →  0x01`
  - `!!0x41  →  0x01`
  - `0x69 && 0x55  →  0x01`
  - `0x69 || 0x55  →  0x01`
  - `p && *p   (avoids null pointer a`



BOOLEAN ALGEBRA

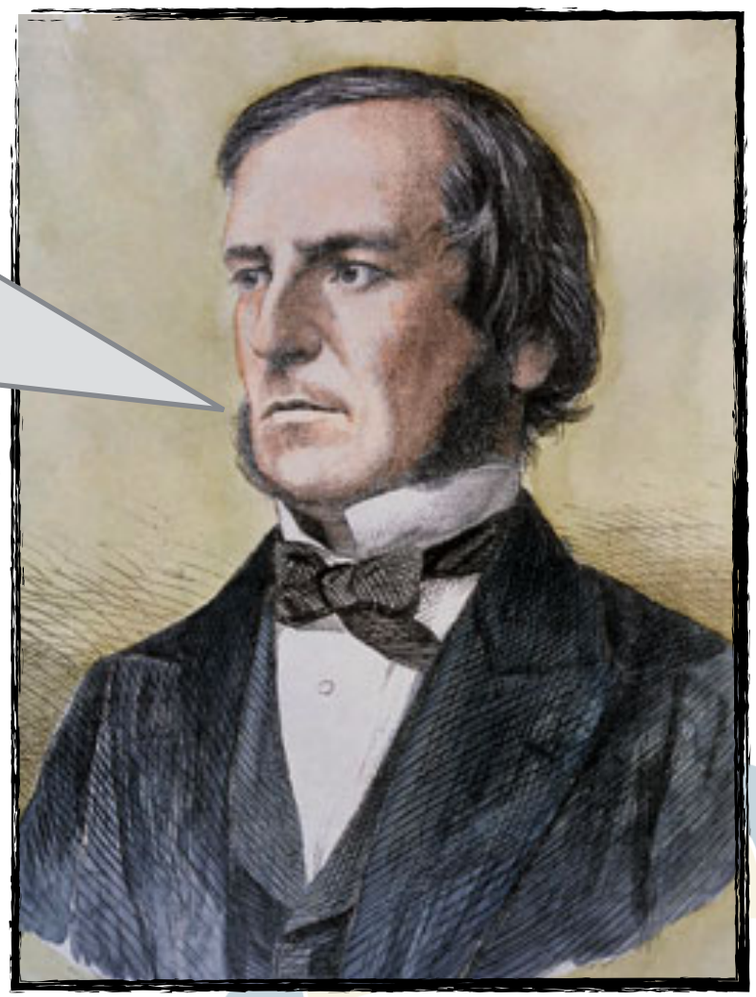
CONTRAST TO LOGIC OPERATIONS IN C

- Examples (char data type)
  - `!0x41  →  0`
  - `!0x00  →  0`
  - `!!0x41  →  `
  - `0x69 && 0x55`
  - `0x69 || 0x55`
  - `p && *p   (avoids null pointer a`

Don't confuse

&&	for	&
||	for	|
!	for	~
pow(x,y)	for	^

common C error



SHIFT OPERATIONS

- Left Shift: x << y
  - Shift bit-vector x left y positions
  - Throw away extra bits on left
  - Fill with 0's on right

SHIFT OPERATIONS

- Right Shift: x >> y
  - Shift bit-vector x right y positions
    - Throw away extra bits on right
  - Logical shift
    - Fill with 0's on left
  - Arithmetic shift
    - Replicate most significant bit on left

- Undefined Behavior
  - Shift amount < 0 or ≥ word size

BOOLEAN ALGEBRA

SHIFT OPERATIONS

- x << 3
- x >> 2
- x >> 2

Argument x	01100010
<< 3	00010000
Log. >> 2	00011000
Arith. >> 2	00011000

Argument x	10100010
<< 3	00010000
Log. >> 2	00101000
Arith. >> 2	11101000

ENCODING INTEGERS

UNSIGNED

$$B2U(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i$$

ENCODING INTEGERS

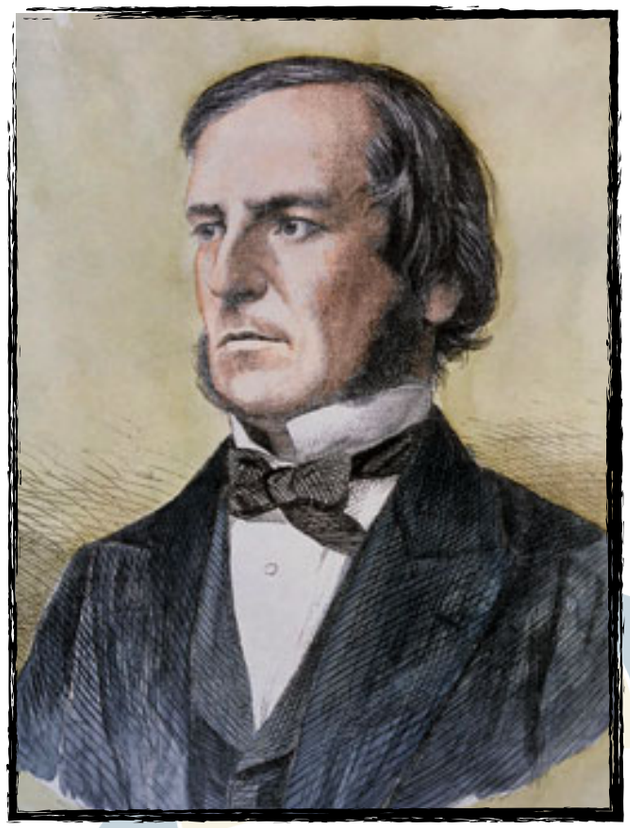
SIGNED - TWO'S COMPLEMENT

$$B2T(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

ENCODING INTEGERS

SIGNED - TWO'S COMPLEMENT

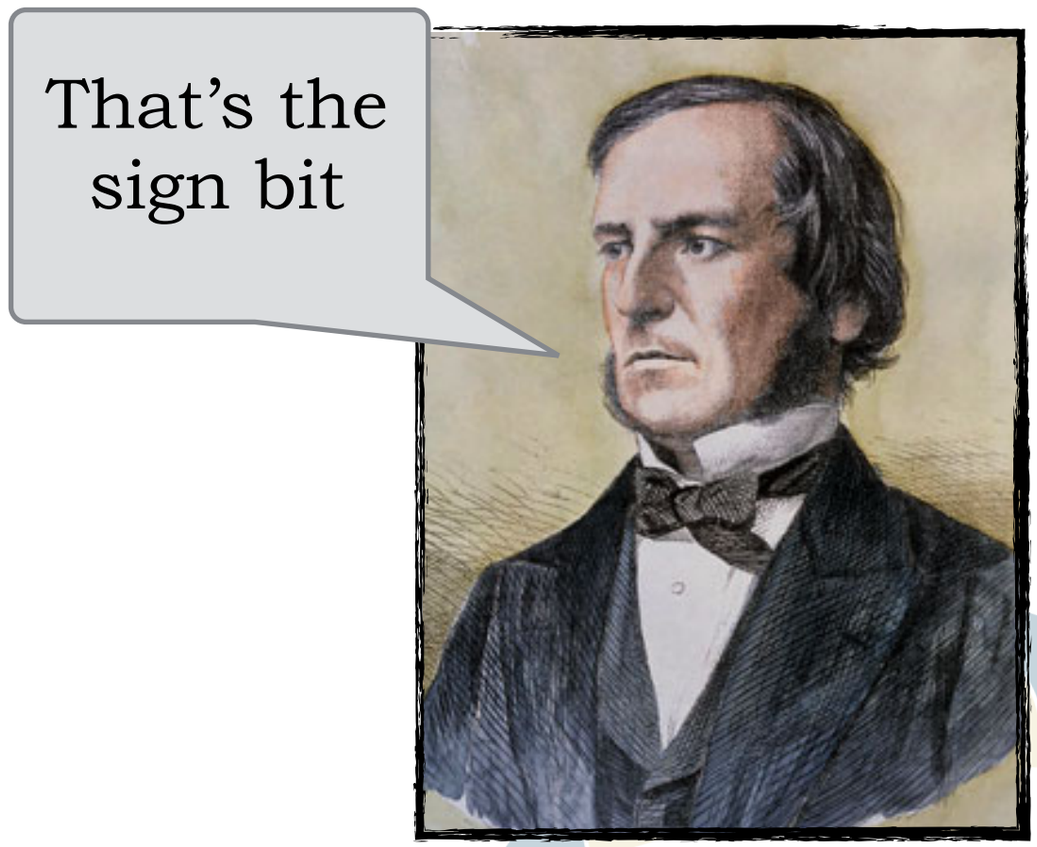
$$B2T(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$



ENCODING INTEGERS

SIGNED - TWO'S COMPLEMENT

$$B2T(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

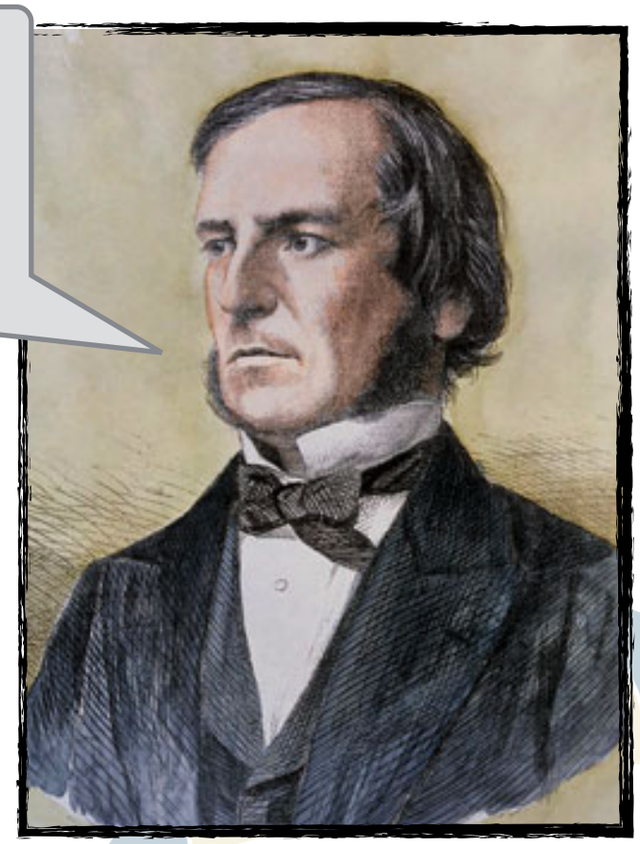


ENCODING INTEGERS

SIGNED - TWO'S COMPLEMENT

$$B2T(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

That's the sign bit



ENCODING INTEGERS

UNSIGNED AND SIGNED INTEGERS

- C `short` 2 bytes long

	Decimal	Hex	Binary
x	15213	**3B 6D**	**00111011 01101101**
y	-15213	**C4 93**	**11000100 10010011**

$$B2U(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i$$

$$B2T(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

- Sign Bit
  - For 2's complement, most significant bit indicates sign
    - 0 for nonnegative
    - 1 for negative

ENCODING INTEGERS

TWO'S COMPLEMENT EXAMPLE

```
x =      15213: 00111011 01101101
y =     -15213: 11000100 10010011
```

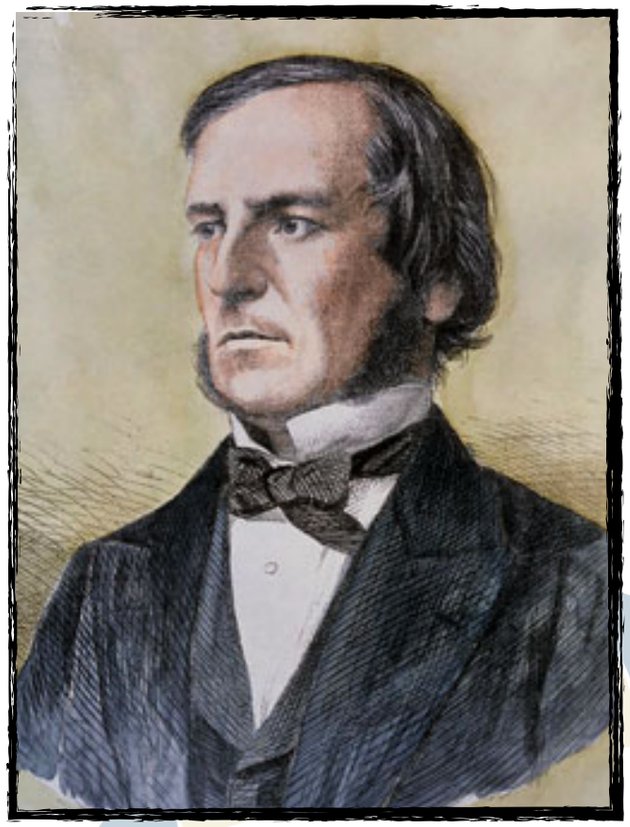
Weight	15213		-15213	
1	1	1	1	1
2	0	0	1	2
4	1	4	0	0
8	1	8	0	0
16	0	0	1	16
32	1	32	0	0
64	1	64	0	0
128	0	0	1	128
256	1	256	0	0
512	1	512	0	0
1024	0	0	1	1024
2048	1	2048	0	0
4096	1	4096	0	0
8192	1	8192	0	0
16384	0	0	1	16384
-32768	0	0	1	-32768
Sum		**15213**		**-15213**

ENCODING INTEGERS

SIGNED - TWO'S COMPLEMENT

ENCODING INTEGERS

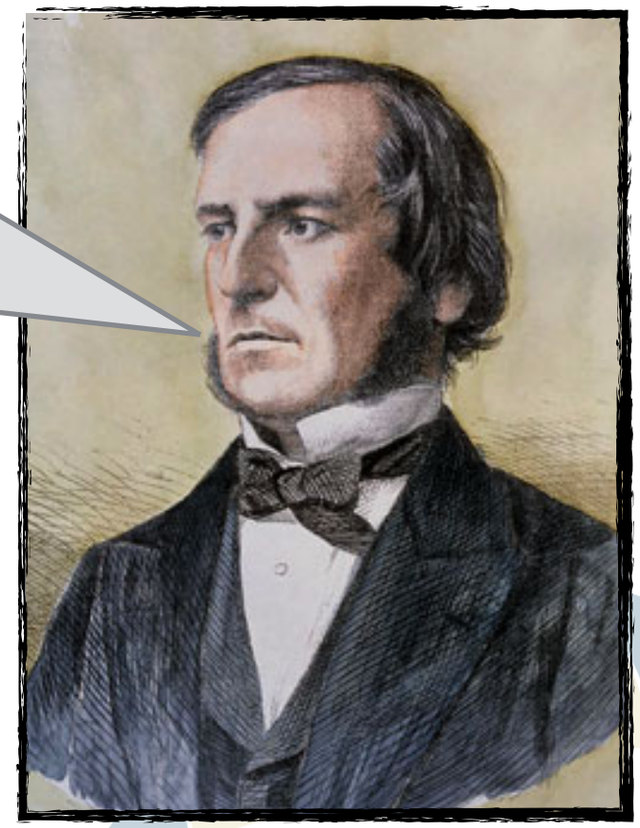
SIGNED - TWO'S COMPLEMENT



ENCODING INTEGERS

SIGNED - TWO'S COMPLEMENT

Why do we use Two's Complement instead of something easier like just a sign bit?



ENCODING INTEGERS

NUMERIC RANGES

- Unsigned Values
  - UMin = 0
    - 000…0
  - UMax = $2^w - 1$
    - 111…1
- Two's Complement Values
  - TMin = -2^{w-1}
    - 100…0
  - TMax = $2^{w-1} - 1$
    - 011…1
  - Other Values
    - Minus 1
      - 111…1

ENCODING INTEGERS

VALUES FOR 2 BYTE WORD (W = 16)

	Decimal	Hex	Binary
UMax	**65535**	**FF FF**	11111111 11111111
TMax	**32767**	7F FF	01111111 11111111
TMin	**-32768**	80 00	10000000 00000000
-1	**-1**	FF FF	11111111 11111111
0	**0**	00 00	00000000 00000000

ENCODING INTEGERS

VALUES FOR DIFFERENT WORD SIZES

	W			
	8	**16**	**32**	**64**
UMax	255	65,535	4,294,967,295	18,446,744,073,709,551,615
TMax	127	32,767	2,147,483,647	9,223,372,036,854,775,807
TMin	-128	-32,768	-2,147,483,648	-9,223,372,036,854,775,808

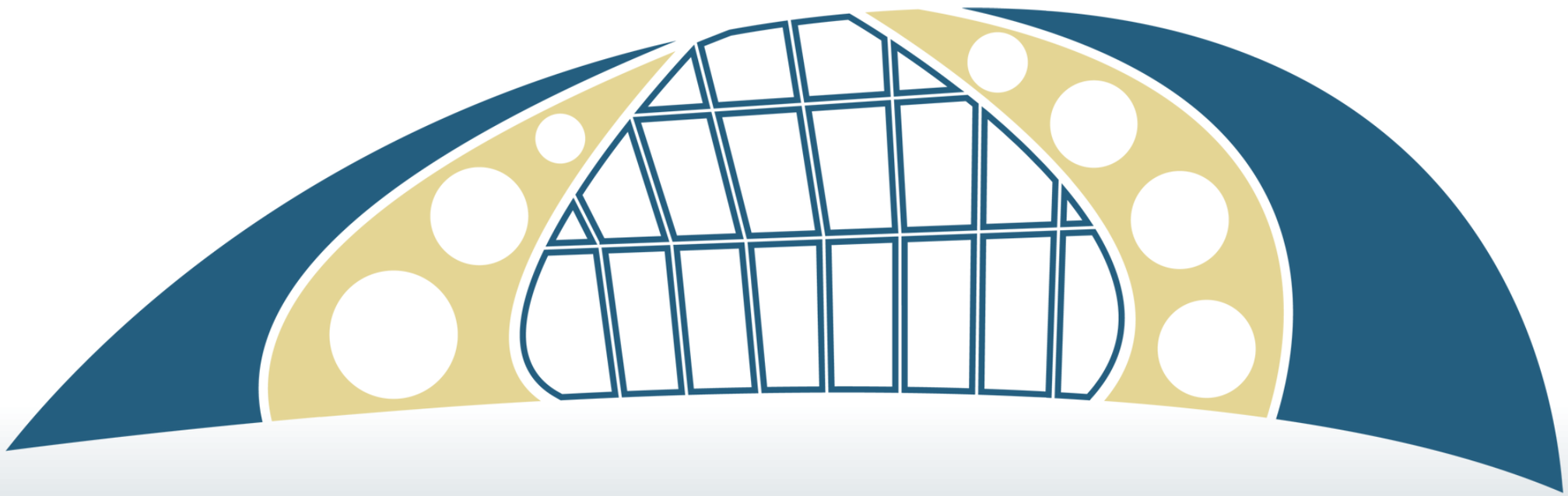
- Observations
  - |TMin | = TMax + 1
    - Asymmetric range
  - UMax = 2 * TMax + 1

ENCODING INTEGERS

UNSIGNED AND SIGNED NUMERIC VALUES

X	B2U(*X*)	B2T(*X*)
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

- Equivalence
  - Same encodings for nonnegative values
- Uniqueness
  - Every bit pattern represents unique integer value
  - Each representable integer has unique bit encoding
- Can Invert Mappings
  - $U2B(x) = B2U^{-1}(x)$
    - Bit pattern for unsigned integer
  - $T2B(x) = B2T^{-1}(x)$
    - Bit pattern for two's comp integer



WESTMONT INSPIRED

COMPUTING LAB