

CSC 252: Computer Organization

Spring 2018: Lecture 2

Instructor: Yuhao Zhu

Department of Computer Science
University of Rochester

Action Items:

- **Programming Assignment 1 is out**
- **Trivia 1 is due on Friday, midnight**

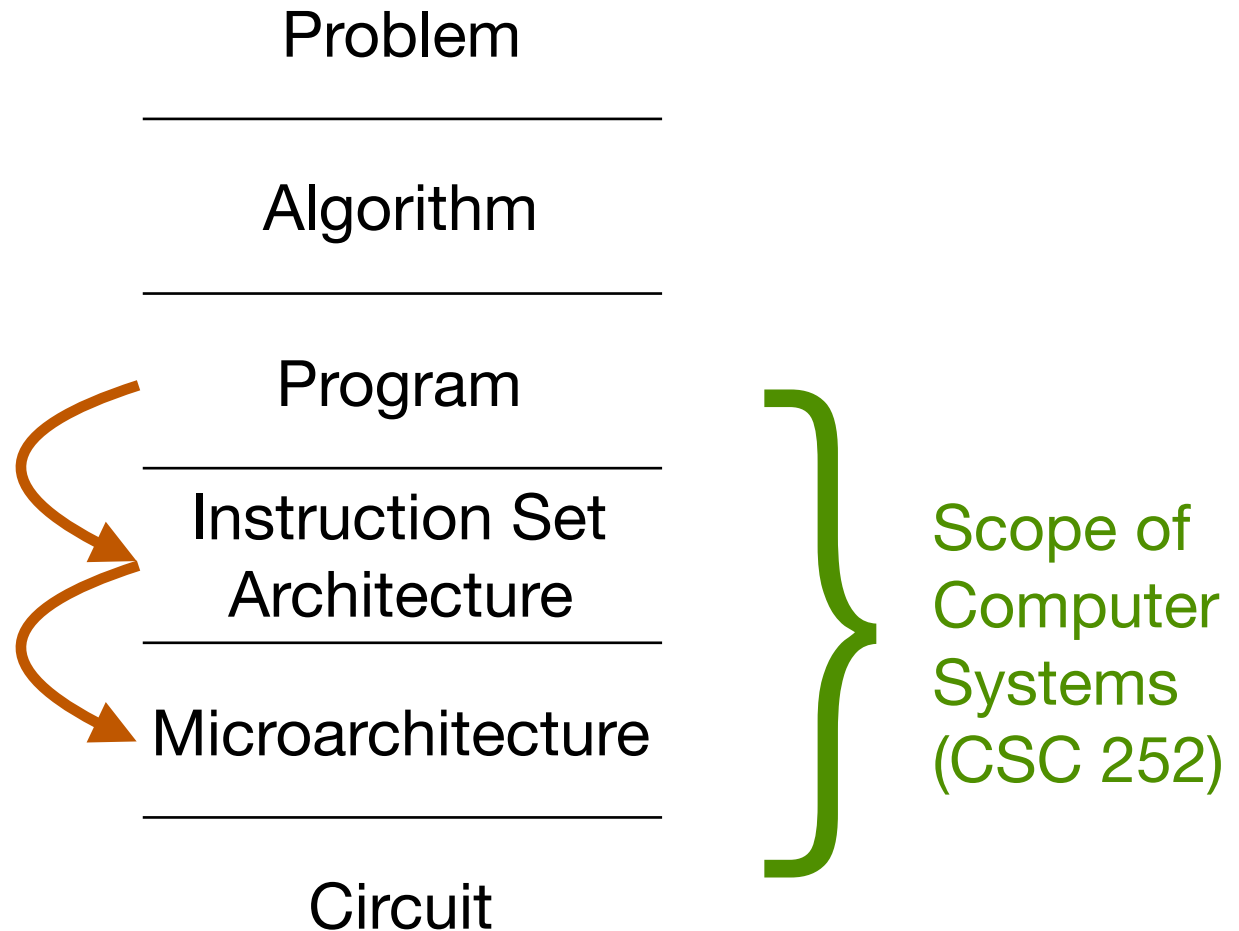
Slide Credits: Randal E. Bryant, David R. O'Hallaron

Announcement

- Programming Assignment 1 is out
 - Details: <http://cs.rochester.edu/courses/252/spring2018/labs/assignment1.html>
 - Due on Feb 2, 11:59 PM
 - Trivia due Friday, 1/26, 11:59 PM
 - You have 3 slip days (not for trivia)
- Ask the TAs if you have any questions regarding programming assignments

Previously in 252...

- How is a human-readable program translated to a representation that computers can understand?
- How does a modern computer execute that program?



Previously in 252...

C Program

```
void add() {  
    int a = 1;  
    int b = 2;  
    int c = a + b;  
}
```

Pre-processor
Compiler



Assembly program

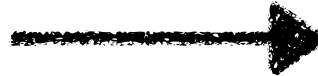
```
movl    $1, -4(%rbp)  
movl    $2, -8(%rbp)  
movl    -4(%rbp), %eax  
addl    -8(%rbp), %eax
```

Previously in 252...

Assembly program

```
movl    $1, -4(%rbp)
movl    $2, -8(%rbp)
movl    -4(%rbp), %eax
addl    -8(%rbp), %eax
```

Assembler
Linker



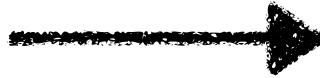
Executable Binary

```
00011001 ...
01101010 ...
11010101 ...
01110001 ...
```

Previously in 252...

Assembly program

```
movl    $1, -4(%rbp)
movl    $2, -8(%rbp)
movl    -4(%rbp), %eax
addl    -8(%rbp), %eax
```



Executable Binary

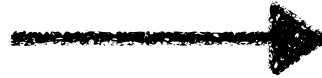
```
00011001 ...
01101010 ...
11010101 ...
01110001 ...
```

- Is ISA referring to assembly or binary?
 - They are the same thing; different representations.
- Instruction = Operator + Operand(s)

Previously in 252...

Assembly program

```
movl $1, -4(%rbp)
movl $2, -8(%rbp)
movl -4(%rbp), %eax
addl -8(%rbp), %eax
```



Executable Binary

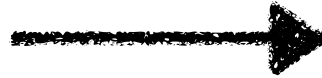
```
00011001 ...
01101010 ...
11010101 ...
01110001 ...
```

- Is ISA referring to assembly or binary?
 - They are the same thing; different representations.
- Instruction = Operator + Operand(s)

Previously in 252...

Assembly program

```
movl  $1, -4(%rbp)
movl  $2, -8(%rbp)
movl  -4(%rbp), %eax
addl  -8(%rbp), %eax
```



Executable Binary

```
00011001 ...
01101010 ...
11010101 ...
01110001 ...
```

- Is ISA referring to assembly or binary?
 - They are the same thing; different representations.
- Instruction = Operator + Operand(s)

Today: Representing Information in Binary

- Why Binary (bits)?
- Bit-level manipulations
- Integers
 - Representation: unsigned and signed
 - Conversion, casting
 - Expanding, truncating
 - Addition, negation, multiplication, shifting
 - Summary
- Representations in memory, pointers, strings

Everything is bits

Everything is bits

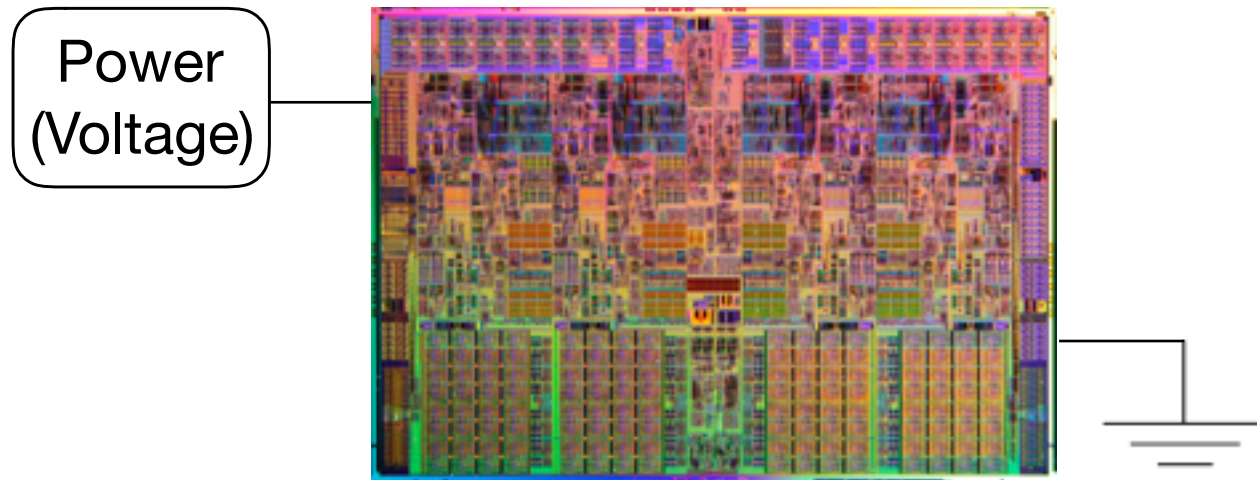
- Each bit is 0 or 1
- By encoding/interpreting sets of bits in various ways
 - Computers determine what to do (instructions)

Everything is bits

- Each bit is 0 or 1
- By encoding/interpreting sets of bits in various ways
 - Computers determine what to do (instructions)
- Why bits? Electronic Implementation

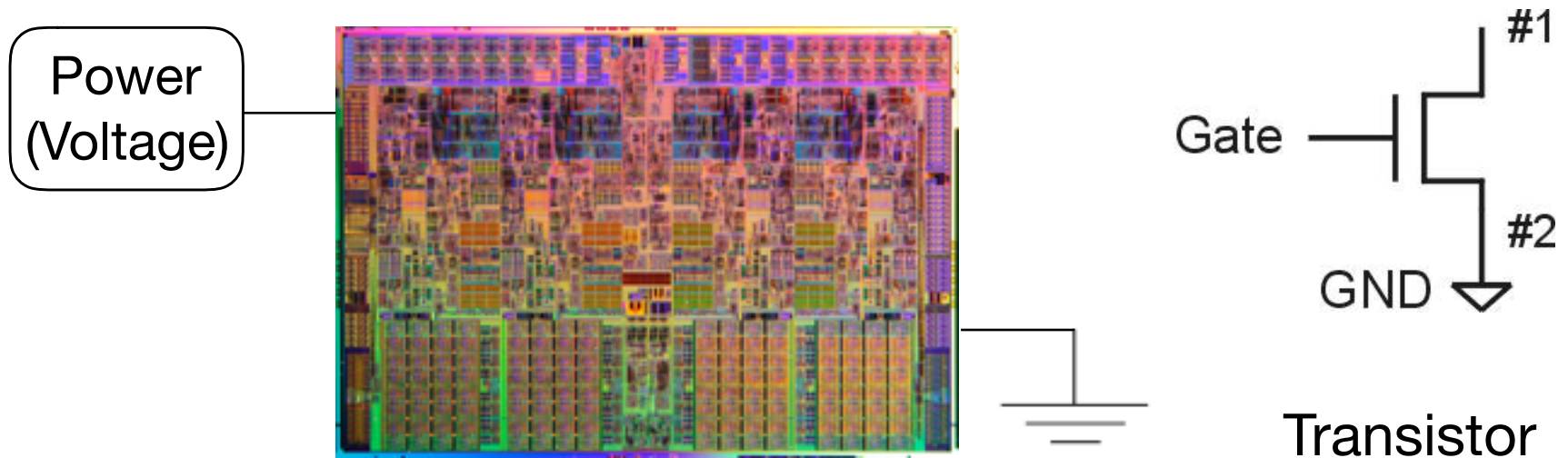
Everything is bits

- Each bit is 0 or 1
- By encoding/interpreting sets of bits in various ways
 - Computers determine what to do (instructions)
- Why bits? Electronic Implementation



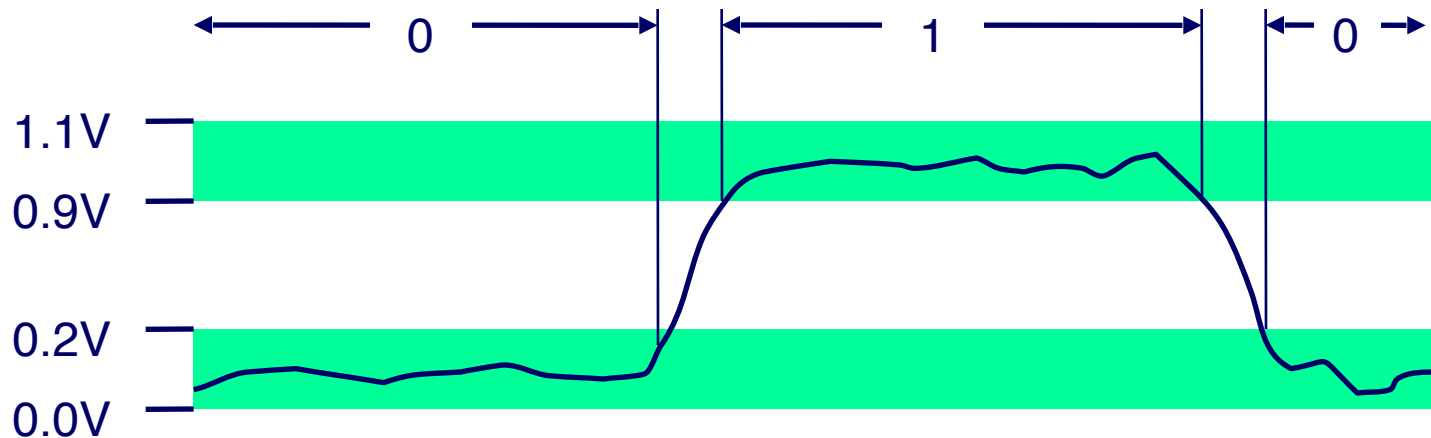
Everything is bits

- Each bit is 0 or 1
- By encoding/interpreting sets of bits in various ways
 - Computers determine what to do (instructions)
- Why bits? Electronic Implementation



Everything is bits

- Each bit is 0 or 1
- By encoding/interpreting sets of bits in various ways
 - Computers determine what to do (instructions)
- **Why bits? Electronic Implementation**
 - Transistor has two states: presence of a high voltage (“1”); presence of a low voltage (“0”)



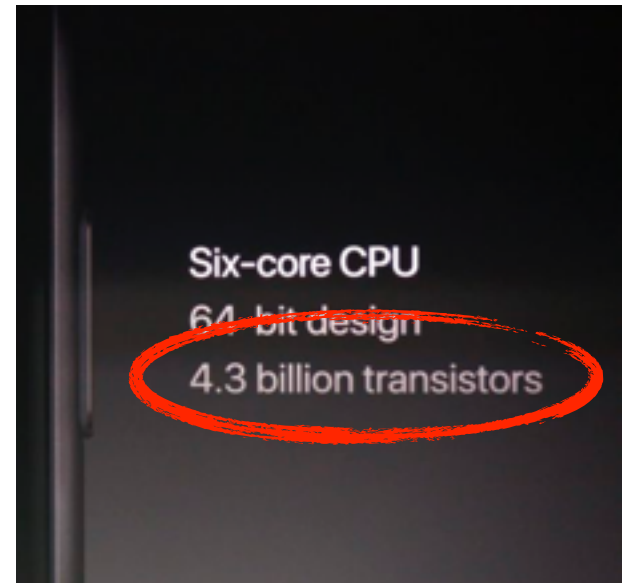
Everything is bits

- Each bit is 0 or 1
- By encoding/interpreting sets of bits in various ways
 - Computers determine what to do (instructions)
- Why bits? Electronic Implementation



Everything is bits

- Each bit is 0 or 1
- By encoding/interpreting sets of bits in various ways
 - Computers determine what to do (instructions)
- Why bits? Electronic Implementation



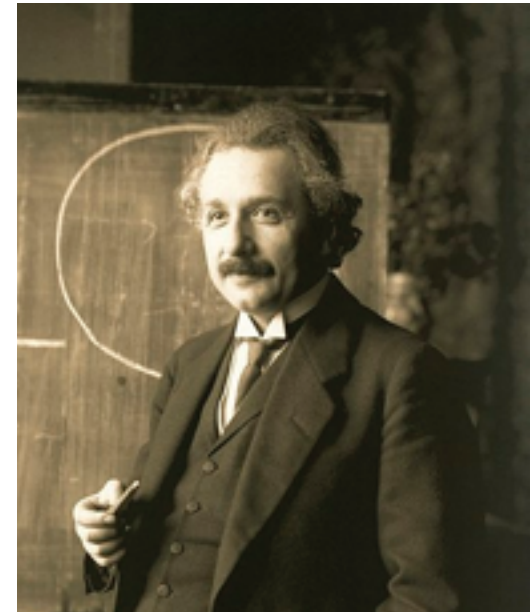
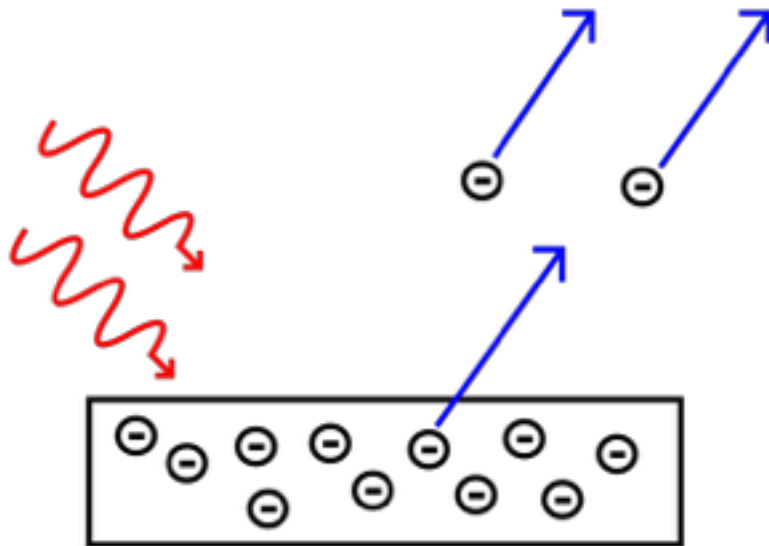
Aside: Why Limit Ourselves Only to Binary?

Aside: Why Limit Ourselves Only to Binary?

- Voltage is continuous. We can interpret it however we want.

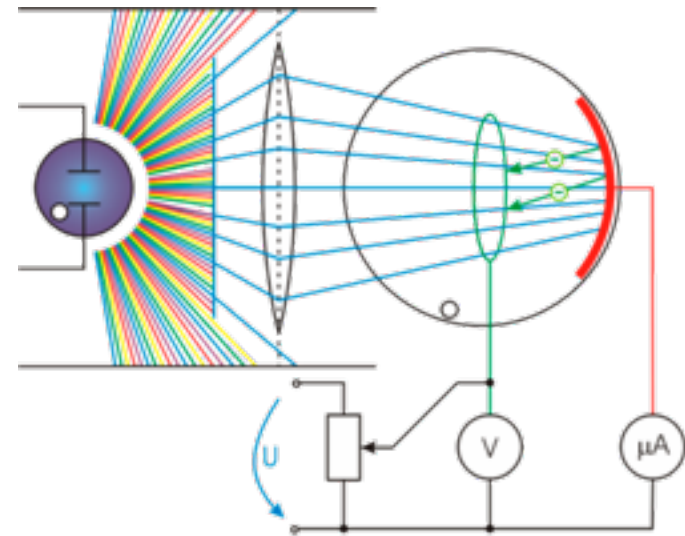
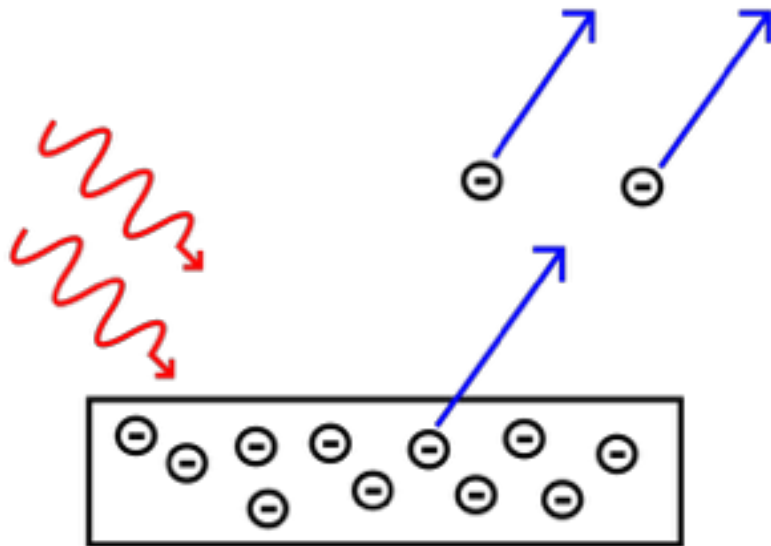
Aside: Why Limit Ourselves Only to Binary?

- Voltage is continuous. We can interpret it however we want.
- Classic Example: Camera Sensor
 - Photoelectric Effect



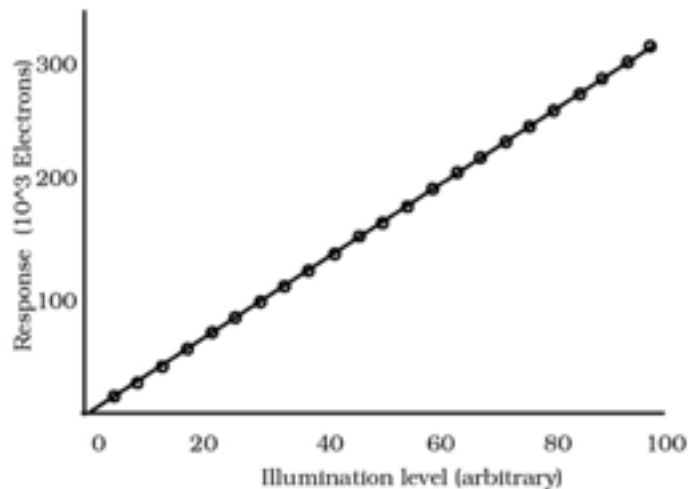
Aside: Why Limit Ourselves Only to Binary?

- Voltage is continuous. We can interpret it however we want.
- Classic Example: Camera Sensor
 - Photoelectric Effect

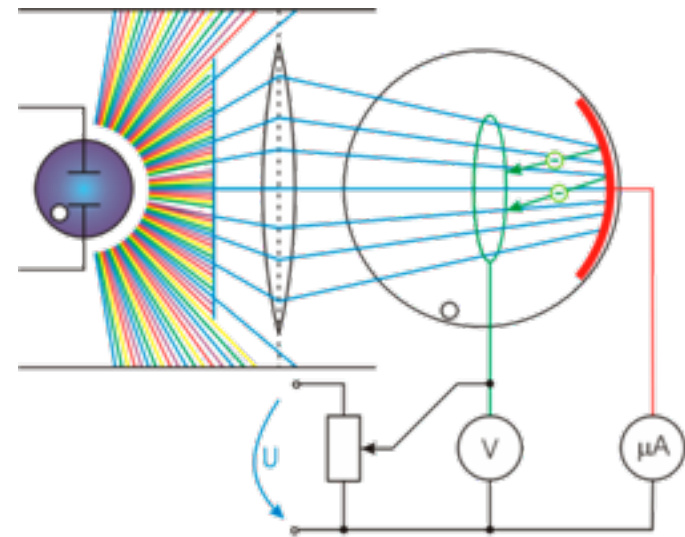


Aside: Why Limit Ourselves Only to Binary?

- Voltage is continuous. We can interpret it however we want.
- Classic Example: Camera Sensor
 - Photoelectric Effect

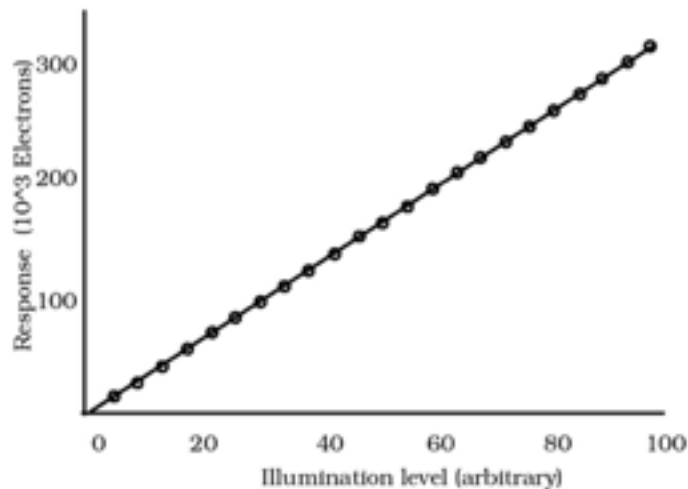


(Epperson, P.M. et al. Electro-optical characterization of the Tektronix TK5 ..., Opt Eng., 25, 1987)



Aside: Why Limit Ourselves Only to Binary?

- Voltage is continuous. We can interpret it however we want.
- Classic Example: Camera Sensor
 - Photoelectric Effect



(Epperson, P.M. et al. Electro-optical characterization of the Tektronix TK5, Opt Eng., 25, 1987)



Binary Notation

Binary Notation

- Base 2 Number Representation
 - e.g., $1011_2 = 11_{10}$

Binary Notation

- **Base 2** Number Representation
 - e.g., $1011_2 = 11_{10}$
- **Weighted Positional Notation**
 - Each bit has a weight depending on its position

Binary Notation

- **Base 2** Number Representation
 - e.g., $1011_2 = 11_{10}$
- Weighted Positional Notation
 - Each bit has a weight depending on its position
- $b_3b_2b_1b_0 = b^0*2^0 + b^1*2^1 + b^2*2^2 + b^3*2^3$

Binary Notation

- **Base 2** Number Representation
 - e.g., $1011_2 = 11_{10}$
- Weighted Positional Notation
 - Each bit has a weight depending on its position
- $b_3b_2b_1b_0 = b^0*2^0 + b^1*2^1 + b^2*2^2 + b^3*2^3$
- $1011_2 = 1*2^0 + 1*2^1 + 0*2^2 + 1*2^3 = 11_{10}$

Binary Notation

- **Base 2** Number Representation
 - e.g., $1011_2 = 11_{10}$
- Weighted Positional Notation
 - Each bit has a weight depending on its position
- $b_3b_2b_1b_0 = b^0*2^0 + b^1*2^1 + b^2*2^2 + b^3*2^3$
- $1011_2 = 1*2^0 + 1*2^1 + 0*2^2 + 1*2^3 = 11_{10}$

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

Binary Notation

- **Base 2** Number Representation
 - e.g., $1011_2 = 11_{10}$
- Weighted Positional Notation
 - Each bit has a weight depending on its position
- $b_3b_2b_1b_0 = b^0*2^0 + b^1*2^1 + b^2*2^2 + b^3*2^3$
- $1011_2 = 1*2^0 + 1*2^1 + 0*2^2 + 1*2^3 = 11_{10}$
- Binary Arithmetic

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

Binary Notation

- **Base 2** Number Representation
 - e.g., $1011_2 = 11_{10}$
- Weighted Positional Notation
 - Each bit has a weight depending on its position
- $b_3b_2b_1b_0 = b^0*2^0 + b^1*2^1 + b^2*2^2 + b^3*2^3$
- $1011_2 = 1*2^0 + 1*2^1 + 0*2^2 + 1*2^3 = 11_{10}$
- Binary Arithmetic

$$\begin{array}{r} 0110 \\ + 0101 \\ \hline 1011 \end{array}$$

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

Binary Notation

- **Base 2** Number Representation
 - e.g., $1011_2 = 11_{10}$
- Weighted Positional Notation
 - Each bit has a weight depending on its position
- $b_3b_2b_1b_0 = b^0*2^0 + b^1*2^1 + b^2*2^2 + b^3*2^3$
- $1011_2 = 1*2^0 + 1*2^1 + 0*2^2 + 1*2^3 = 11_{10}$
- Binary Arithmetic

$$\begin{array}{r} 0110 \\ + 0101 \\ \hline 1011 \end{array}$$

$$\begin{array}{r} 6 \\ + 5 \\ \hline 11 \end{array}$$

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

Hexadecimal (Hex) Notation

- **Base 16** Number Representation
 - Use characters '0' to '9' and 'A' to 'F'
 - Four bits per Hex digit
 - $11111110_2 = FE_{16}$
- Write $FA1D37B_{16}$ in C as
 - `0xFA1D37B`
 - `0xfa1d37b`

Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

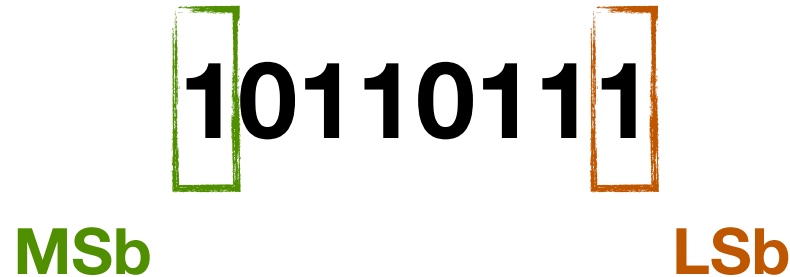
Bit, Byte, Word

- Byte = 8 bits

- Binary 00000000_2 to 11111111_2 ; Decimal: 0_{10} to 255_{10} ; Hex: 00_{16} to FF_{16}
- Least Significant Bit (LSb) vs. Most Significant Bit (MSb)

10110111

MSb **LSb**

The diagram shows the binary sequence '10110111'. The first bit '1' is enclosed in a green rectangular box, and the last bit '1' is enclosed in an orange rectangular box. Below the green box is the label 'MSb' in green text, and below the orange box is the label 'LSb' in orange text.

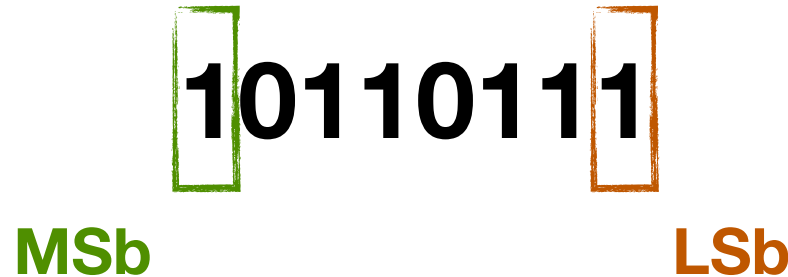
Bit, Byte, Word

- Byte = 8 bits

- Binary 00000000_2 to 11111111_2 ; Decimal: 0_{10} to 255_{10} ; Hex: 00_{16} to FF_{16}
- Least Significant Bit (LSb) vs. Most Significant Bit (MSb)

10110111

MSb **LSb**

A diagram showing the binary sequence '10110111'. The first bit '1' is enclosed in a green rectangular box, and the last bit '1' is enclosed in an orange rectangular box. Below the green box is the label 'MSb' in green text, and below the orange box is the label 'LSb' in orange text.

- Word = 4 Bytes (32-bit machine) / 8 Bytes (64-bit machine)
- Least Significant Byte (LSB) vs. Most Significant Byte (MSB)

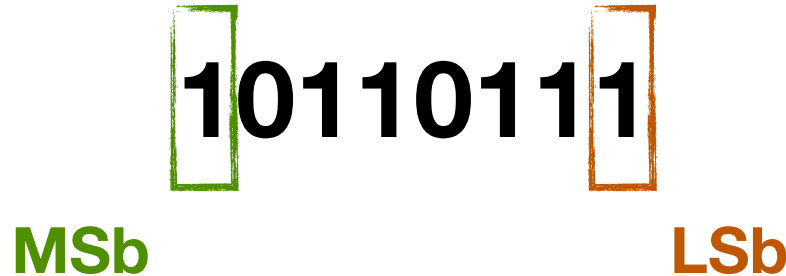
Bit, Byte, Word

- Byte = 8 bits

- Binary 00000000_2 to 11111111_2 ; Decimal: 0_{10} to 255_{10} ; Hex: 00_{16} to FF_{16}
- Least Significant Bit (LSb) vs. Most Significant Bit (MSb)

10110111

MSb **LSb**

A diagram showing the binary sequence '10110111'. The first bit '1' is enclosed in a green rectangular box, and the last bit '1' is enclosed in an orange rectangular box. Below the green box is the label 'MSb' in green text, and below the orange box is the label 'LSb' in orange text.

- Word = 4 Bytes (32-bit machine) / 8 Bytes (64-bit machine)
- Least Significant Byte (LSB) vs. Most Significant Byte (MSB)

Questions?

Today: Representing Information in Binary

- Why Binary (bits)?
- Bit-level manipulations
- Integers
 - Representation: unsigned and signed
 - Conversion, casting
 - Expanding, truncating
 - Addition, negation, multiplication, shifting
 - Summary
- Representations in memory, pointers, strings

Boolean Algebra

- Developed by George Boole in 19th Century
 - Algebraic representation of logic
 - Encode “True” as 1 and “False” as 0

And

- $A \& B = 1$ when both $A=1$ and $B=1$

$\&$	0	1
0	0	0
1	0	1

Or

- $A | B = 1$ when either $A=1$ or $B=1$

$ $	0	1
0	0	1
1	1	1

Not

- $\sim A = 1$ when $A=0$

\sim	
0	1
1	0

Exclusive-Or (Xor)

- $A \wedge B = 1$ when either $A=1$ or $B=1$, but not both

\wedge	0	1
0	0	1
1	1	0

General Boolean Algebras

- Operate on Bit Vectors
 - Operations applied bitwise

01101001	01101001	01101001	
& 01010101	01010101	^ 01010101	~ 01010101
<u> </u>	<u> </u>	<u> </u>	<u> </u>

- All of the Properties of Boolean Algebra Apply

General Boolean Algebras

- Operate on Bit Vectors
 - Operations applied bitwise

$$\begin{array}{cccc} 01101001 & 01101001 & 01101001 & 01101001 \\ \& \underline{01010101} & | \underline{01010101} & ^ \underline{01010101} & \sim \underline{01010101} \\ 01000001 & & & & \end{array}$$

- All of the Properties of Boolean Algebra Apply

General Boolean Algebras

- Operate on Bit Vectors
 - Operations applied bitwise

01101001	01101001	01101001	
& 01010101	01010101	^ 01010101	~ 01010101
<u> </u>	<u> </u>	<u> </u>	<u> </u>
01000001	01111101		

- All of the Properties of Boolean Algebra Apply

General Boolean Algebras

- Operate on Bit Vectors
 - Operations applied bitwise

01101001	01101001	01101001	
& 01010101	01010101	^ 01010101	~ 01010101
<u> </u>	<u> </u>	<u> </u>	<u> </u>
01000001	01111101	00111100	

- All of the Properties of Boolean Algebra Apply

General Boolean Algebras

- Operate on Bit Vectors
 - Operations applied bitwise

01101001	01101001	01101001	
& 01010101	01010101	^ 01010101	~ 01010101
<u> </u>	<u> </u>	<u> </u>	<u> </u>
01000001	01111101	00111100	10101010

- All of the Properties of Boolean Algebra Apply

Bit-Level Operations in C

- Operations $\&$, $|$, \sim , \wedge Available in C
 - Apply to any “integral” data type
 - long, int, short, char, unsigned
 - View arguments as bit vectors
 - Arguments applied bit-wise
- Examples (Char data type)
 - $\sim 0x41 \rightarrow 0xBE$
 - $\sim 01000001_2 \rightarrow 10111110_2$
 - $\sim 0x00 \rightarrow 0xFF$
 - $\sim 00000000_2 \rightarrow 11111111_2$
 - $0x69 \& 0x55 \rightarrow 0x41$
 - $01101001_2 \& 01010101_2 \rightarrow 01000001_2$
 - $0x69 | 0x55 \rightarrow 0x7D$
 - $01101001_2 | 01010101_2 \rightarrow 01111101_2$

Contrast: Logic Operations in C

- Contrast to Logical Operators
 - `&&`, `||`, `!`
 - View 0 as “False”
 - Anything nonzero as “True”
 - Always return 0 or 1
 - Early termination
- Examples (char data type)
 - `!0x41` → `0x00`
 - `!0x00` → `0x01`
 - `!!0x41` → `0x01`
 - `0x69 && 0x55` → `0x01`
 - `0x69 || 0x55` → `0x01`
 - `p && *p` (avoids null pointer access)

Contrast: Logic Operations in C

- Contrast to Logical Operators

- &&, ||, !

- View 0 as “False”

- Anything non-zero

- Always true

- Early termination

- Examples

- !0x41

- !0x00

- !!0x41

- 0x69 && 0x55 → 0x01

- 0x69 || 0x55 → 0x01

- p && *p (avoids null pointer access)

Watch out for && vs. & (and || vs. |)...
one of the more common oopsies in
C programming

Shift Operations

- Left Shift: $x \ll y$
 - Shift bit-vector **x** left **y** positions
 - Throw away extra bits on left
 - Fill with 0's on right
- Right Shift: $x \gg y$
 - 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 \geq word size

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

Argument x	10100010
<< 3	
Log. >> 2	
Arith. >> 2	

Shift Operations

- Left Shift: $x \ll y$
 - Shift bit-vector **x** left **y** positions
 - Throw away extra bits on left
 - Fill with 0's on right
- Right Shift: $x \gg y$
 - 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 \geq word size

Argument x	01100010
<< 3	00010
Log. >> 2	
Arith. >> 2	

Argument x	10100010
<< 3	
Log. >> 2	
Arith. >> 2	

Shift Operations

- Left Shift: $x \ll y$
 - Shift bit-vector **x** left **y** positions
 - Throw away extra bits on left
 - Fill with 0's on right
- Right Shift: $x \gg y$
 - 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 \geq word size

Argument x	01100010
$\ll 3$	00010000
Log. $\gg 2$	
Arith. $\gg 2$	

Argument x	10100010
$\ll 3$	
Log. $\gg 2$	
Arith. $\gg 2$	

Shift Operations

- Left Shift: $x \ll y$
 - Shift bit-vector **x** left **y** positions
 - Throw away extra bits on left
 - Fill with 0's on right
- Right Shift: $x \gg y$
 - 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 \geq word size

Argument x	01100010
<< 3	00010000
Log. >> 2	011000
Arith. >> 2	

Argument x	10100010
<< 3	
Log. >> 2	
Arith. >> 2	

Shift Operations

- Left Shift: $x \ll y$
 - Shift bit-vector **x** left **y** positions
 - Throw away extra bits on left
 - Fill with 0's on right
- Right Shift: $x \gg y$
 - 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 \geq word size

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

Argument x	10100010
<< 3	
Log. >> 2	
Arith. >> 2	

Shift Operations

- Left Shift: $x \ll y$
 - Shift bit-vector **x** left **y** positions
 - Throw away extra bits on left
 - Fill with 0's on right
- Right Shift: $x \gg y$
 - 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 \geq word size

Argument x	01100010
<< 3	00010000
Log. >> 2	00011000
Arith. >> 2	011000

Argument x	10100010
<< 3	
Log. >> 2	
Arith. >> 2	

Shift Operations

- Left Shift: $x \ll y$
 - Shift bit-vector **x** left **y** positions
 - Throw away extra bits on left
 - Fill with 0's on right
- Right Shift: $x \gg y$
 - 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 \geq word size

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

Argument x	10100010
<< 3	
Log. >> 2	
Arith. >> 2	

Shift Operations

- Left Shift: $x \ll y$
 - Shift bit-vector **x** left **y** positions
 - Throw away extra bits on left
 - Fill with 0's on right
- Right Shift: $x \gg y$
 - 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 \geq word size

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

Argument x	10100010
<< 3	00010
Log. >> 2	
Arith. >> 2	

Shift Operations

- Left Shift: $x \ll y$
 - Shift bit-vector **x** left **y** positions
 - Throw away extra bits on left
 - Fill with 0's on right
- Right Shift: $x \gg y$
 - 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 \geq word size

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

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

Shift Operations

- Left Shift: $x \ll y$
 - Shift bit-vector **x** left **y** positions
 - Throw away extra bits on left
 - Fill with 0's on right
- Right Shift: $x \gg y$
 - 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 \geq word size

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

Argument x	10100010
<< 3	00010000
Log. >> 2	101000
Arith. >> 2	

Shift Operations

- Left Shift: $x \ll y$
 - Shift bit-vector **x** left **y** positions
 - Throw away extra bits on left
 - Fill with 0's on right
- Right Shift: $x \gg y$
 - 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 \geq word size

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

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

Shift Operations

- Left Shift: $x \ll y$
 - Shift bit-vector **x** left **y** positions
 - Throw away extra bits on left
 - Fill with 0's on right
- Right Shift: $x \gg y$
 - 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 \geq word size

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

Argument x	10100010
<< 3	00010000
Log. >> 2	00101000
Arith. >> 2	101000

Shift Operations

- Left Shift: $x \ll y$
 - Shift bit-vector **x** left **y** positions
 - Throw away extra bits on left
 - Fill with 0's on right
- Right Shift: $x \gg y$
 - 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 \geq word size

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

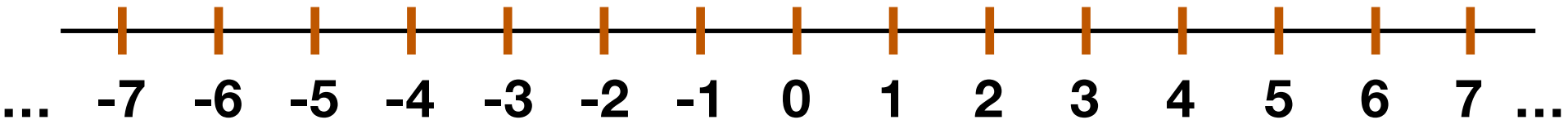
Argument x	10100010
<< 3	00010000
Log. >> 2	00101000
Arith. >> 2	11101000

Today: Representing Information in Binary

- Why Binary (bits)?
- Bit-level manipulations
- **Integers**
 - Representation: unsigned and signed
 - Conversion, casting
 - Expanding, truncating
 - Addition, negation, multiplication, shifting
 - Summary
- Representations in memory, pointers, strings

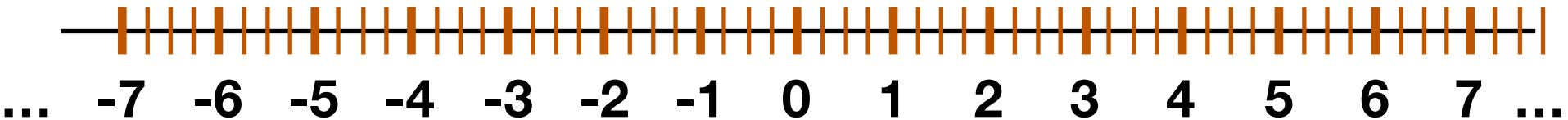
Representing Numbers in Binary

- What numbers do we need to represent in bits?
 - Integer (Negative and Non-negative)
 - Fractions
 - Irrationals



Representing Numbers in Binary

- What numbers do we need to represent in bits?
 - Integer (Negative and Non-negative)
 - Fractions
 - Irrationals



Encoding Negative Numbers

Encoding Negative Numbers

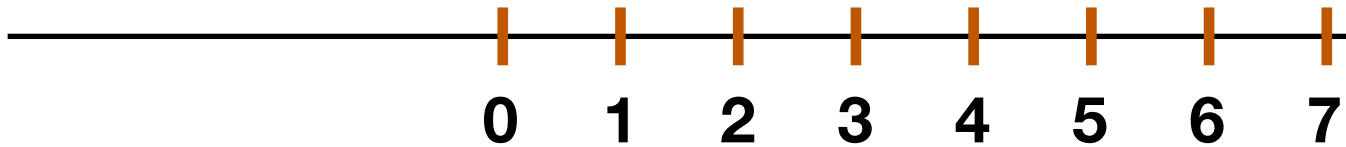
- So far we have been discussing non-negative numbers: so called **unsigned**. How about negative numbers?

Encoding Negative Numbers

- So far we have been discussing non-negative numbers: so called **unsigned**. How about negative numbers?
- Solution 1: Sign-magnitude
 - First bit represents sign; 0 for positive; 1 for negative
 - The rest represents magnitude

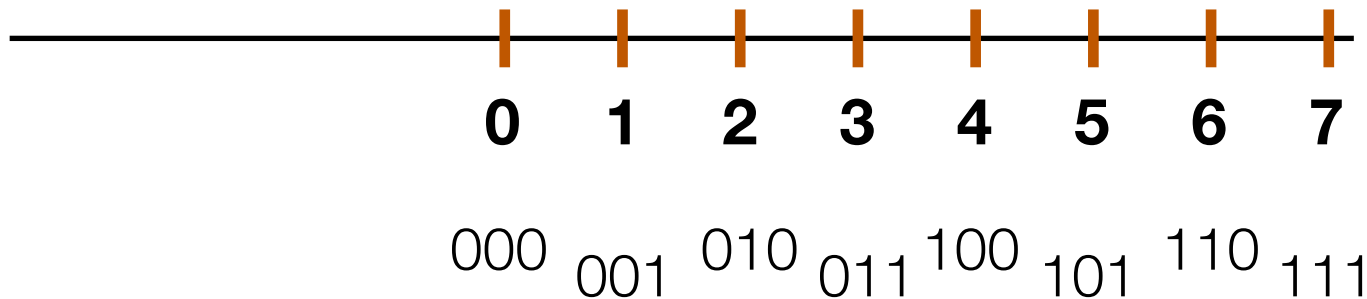
Encoding Negative Numbers

- So far we have been discussing non-negative numbers: so called **unsigned**. How about negative numbers?
- Solution 1: Sign-magnitude
 - First bit represents sign; 0 for positive; 1 for negative
 - The rest represents magnitude



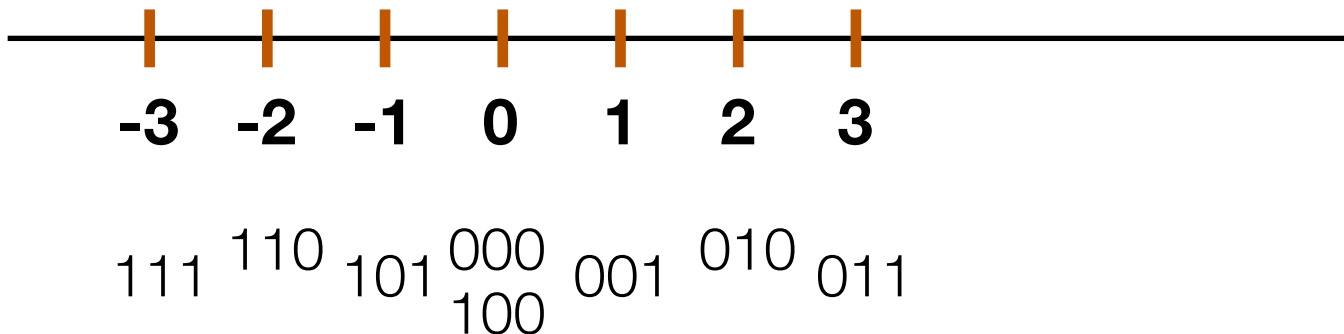
Encoding Negative Numbers

- So far we have been discussing non-negative numbers: so called **unsigned**. How about negative numbers?
- Solution 1: Sign-magnitude
 - First bit represents sign; 0 for positive; 1 for negative
 - The rest represents magnitude



Encoding Negative Numbers

- So far we have been discussing non-negative numbers: so called **unsigned**. How about negative numbers?
- Solution 1: Sign-magnitude
 - First bit represents sign; 0 for positive; 1 for negative
 - The rest represents magnitude



Sign-Magnitude Implications

- Bits have different semantics
 - Two zeros...
 - Normal arithmetic doesn't work
 - Make hardware design harder

Signed Value	Binary
0	000
1	001
2	010
3	011
-0	100
-1	101
-2	110
-3	111

Sign-Magnitude Implications

- Bits have different semantics
 - Two zeros...
 - Normal arithmetic doesn't work
 - Make hardware design harder

$$\begin{array}{r} 010 \\ +) 101 \\ \hline 111 \end{array}$$

Signed Value	Binary
0	000
1	001
2	010
3	011
-0	100
-1	101
-2	110
-3	111

Sign-Magnitude Implications

- Bits have different semantics
 - Two zeros...
 - Normal arithmetic doesn't work
 - Make hardware design harder

$$\begin{array}{r} 010 \\ +) 101 \\ \hline 111 \end{array}$$

$$\begin{array}{r} 2 \\ +) -1 \\ \hline -3 \end{array}$$

Signed Value	Binary
0	000
1	001
2	010
3	011
-0	100
-1	101
-2	110
-3	111

Sign-Magnitude Implications

- Bits have different semantics
 - Two zeros...
 - Normal arithmetic doesn't work
 - Make hardware design harder

The diagram shows two arithmetic operations, each crossed out with a large red X. On the left, the operation is
$$\begin{array}{r} 010 \\ +) 101 \\ \hline 111 \end{array}$$
 with the result 111 written below the line. On the right, the operation is
$$\begin{array}{r} 2 \\ +) -1 \\ \hline -3 \end{array}$$
 with the result -3 written below the line. The red X is drawn over both operations, indicating that such arithmetic is invalid or problematic in sign-magnitude representation.

Signed Value	Binary
0	000
1	001
2	010
3	011
-0	100
-1	101
-2	110
-3	111

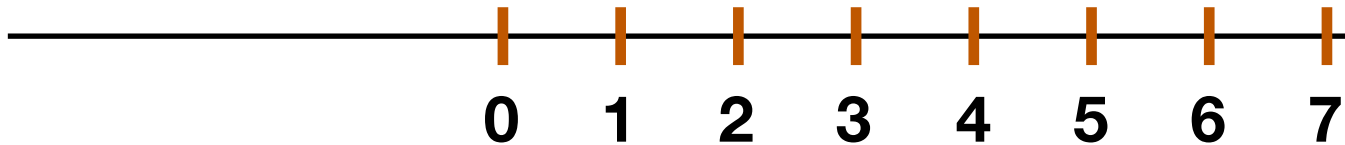
Encoding Negative Numbers

- Solution 2: Two's Complement

$$101_2 = 1 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^{-2} = -3_{10}$$

Encoding Negative Numbers

- Solution 2: Two's Complement

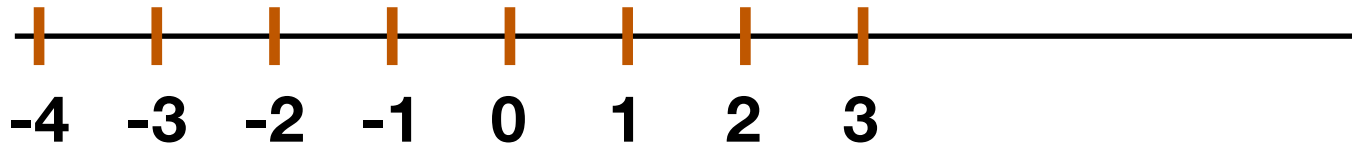


Unsigned	Binary
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

$$101_2 = 1 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^{-2} = -3_{10}$$

Encoding Negative Numbers

- Solution 2: Two's Complement

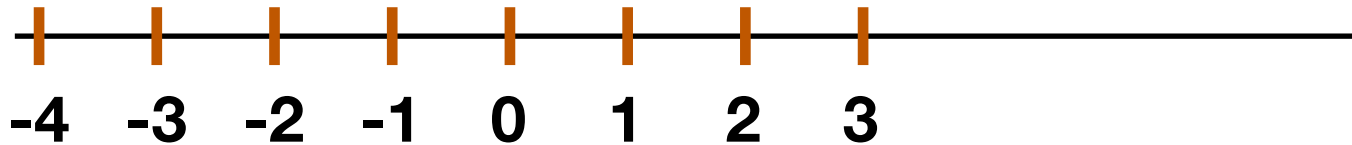


Unsigned	Binary
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

$$101_2 = 1 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^{-2} = -3_{10}$$

Encoding Negative Numbers

- Solution 2: Two's Complement

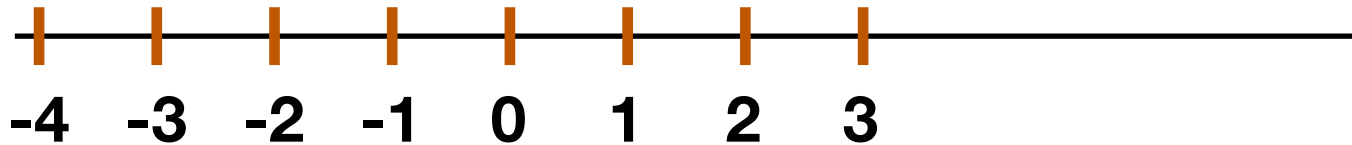


Signed	Unsigned	Binary
0	0	000
1	1	001
2	2	010
3	3	011
-4	4	100
-3	5	101
-2	6	110
-1	7	111

$$101_2 = 1 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^{-2} = -3_{10}$$

Encoding Negative Numbers

- Solution 2: Two's Complement



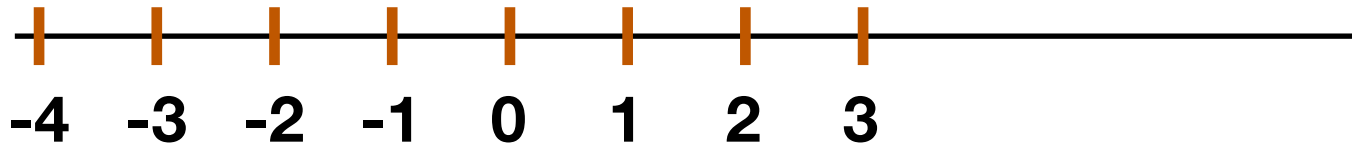
Signed Weight	Unsigned Weight	Bit Position
2^0	2^0	0
2^1	2^1	1
2^{-2}	2^2	2

Signed	Unsigned	Binary
0	0	000
1	1	001
2	2	010
3	3	011
-4	4	100
-3	5	101
-2	6	110
-1	7	111

$$101_2 = 1 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^{-2} = -3_{10}$$

Encoding Negative Numbers

- Solution 2: Two's Complement



Signed Weight	Unsigned Weight	Bit Position
2^0	2^0	0
2^1	2^1	1
2^{-2}	2^2	2

Signed	Unsigned	Binary
0	0	000
1	1	001
2	2	010
3	3	011
-4	4	100
-3	5	101
-2	6	110
-1	7	111

$$101_2 = 1 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^{-2} = -3_{10}$$

Two-Complement Encoding 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	

Two-Complement Implications

- Only 1 zero
- Usual arithmetic still works
- There is a bit that represents sign!
- Most widely used in today's machines

Signed	Binary
0	000
1	001
2	010
3	011
-4	100
-3	101
-2	110
-1	111

Two-Complement Implications

- Only 1 zero
- Usual arithmetic still works
- There is a bit that represents sign!
- Most widely used in today's machines

$$\begin{array}{r} 010 \\ +) 101 \\ \hline 111 \end{array}$$

Signed	Binary
0	000
1	001
2	010
3	011
-4	100
-3	101
-2	110
-1	111

Two-Complement Implications

- Only 1 zero
- Usual arithmetic still works
- There is a bit that represents sign!
- Most widely used in today's machines

$$\begin{array}{r} 010 \\ +) 101 \\ \hline 111 \end{array}$$

$$\begin{array}{r} 2 \\ +) -3 \\ \hline -1 \end{array}$$

Signed	Binary
0	000
1	001
2	010
3	011
-4	100
-3	101
-2	110
-1	111

Numeric Ranges

Numeric Ranges

- Unsigned Values

- $UMin = 0$
000...0

- $UMax = 2^w - 1$
111...1

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

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

Values for $W = 16$

	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

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

Values for $W = 16$

	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

Data Representations in C (in Bytes)

- By default variables are signed
- Unless explicitly declared as unsigned (e.g., `unsigned int`)
- Signed variables use two-complement encoding

C Data Type	32-bit	64-bit
<code>char</code>	1	1
<code>short</code>	2	2
<code>int</code>	4	4
<code>long</code>	4	8

Data Representations in C (in Bytes)

	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

C Data Type	32-bit	64-bit
char	1	1
short	2	2
int	4	4
long	4	8

Data Representations in C (in Bytes)

	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

C Data Type	32-bit	64-bit
char	1	1
short	2	2
int	4	4
long	4	8

- C Language

- `#include <limits.h>`
- Declares constants, e.g.,
 - `ULONG_MAX`
 - `LONG_MAX`
 - `LONG_MIN`
- Values platform specific

Can We Represent Fractions in Binary?

- What does 10.01_2 mean?
- C.f., Decimal
 - $12.45 = 1*10^1 + 2*10^0 + 4*10^{-1} + 5*10^{-2}$
- $10.01_2 = 1*2^1 + 0*2^0 + 0*2^{-1} + 1*2^{-2} = 2.25_{10}$

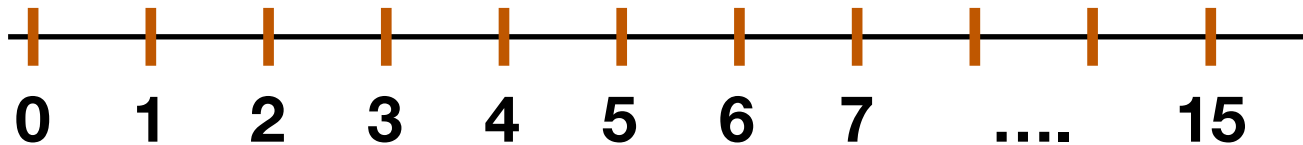
Can We Represent Fractions in Binary?

- What does 10.01_2 mean?
- C.f., Decimal
 - $12.45 = 1 \cdot 10^1 + 2 \cdot 10^0 + 4 \cdot 10^{-1} + 5 \cdot 10^{-2}$
- $10.01_2 = 1 \cdot 2^1 + 0 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} = 2.25_{10}$

Decimal	Binary
0	00.00
0.25	00.01
0.5	00.10
0.75	00.11
1	01.00
1.25	01.01
1.5	01.10
1.75	01.11
2	10.00
2.25	10.01
2.5	10.10
2.75	10.11
3	11.00
3.25	11.01
3.5	11.10
3.75	11.11

Can We Represent Fractions in Binary?

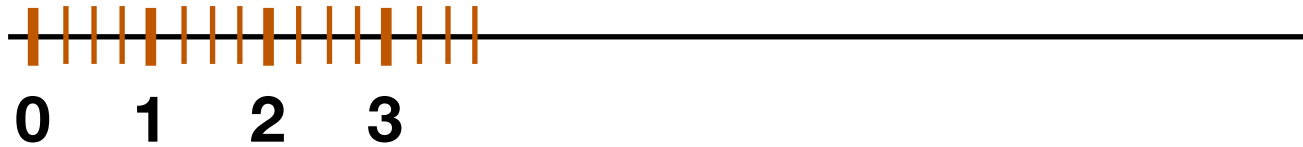
- What does 10.01_2 mean?
- C.f., Decimal
 - $12.45 = 1 \cdot 10^1 + 2 \cdot 10^0 + 4 \cdot 10^{-1} + 5 \cdot 10^{-2}$
- $10.01_2 = 1 \cdot 2^1 + 0 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} = 2.25_{10}$



Decimal	Binary
0	00.00
0.25	00.01
0.5	00.10
0.75	00.11
1	01.00
1.25	01.01
1.5	01.10
1.75	01.11
2	10.00
2.25	10.01
2.5	10.10
2.75	10.11
3	11.00
3.25	11.01
3.5	11.10
3.75	11.11

Can We Represent Fractions in Binary?

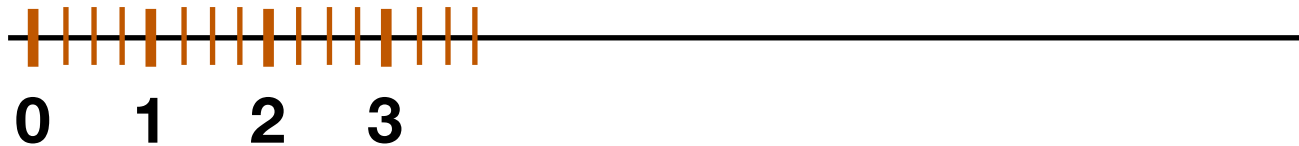
- What does 10.01_2 mean?
- C.f., Decimal
 - $12.45 = 1 \cdot 10^1 + 2 \cdot 10^0 + 4 \cdot 10^{-1} + 5 \cdot 10^{-2}$
- $10.01_2 = 1 \cdot 2^1 + 0 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} = 2.25_{10}$



Decimal	Binary
0	00.00
0.25	00.01
0.5	00.10
0.75	00.11
1	01.00
1.25	01.01
1.5	01.10
1.75	01.11
2	10.00
2.25	10.01
2.5	10.10
2.75	10.11
3	11.00
3.25	11.01
3.5	11.10
3.75	11.11

Can We Represent Fractions in Binary?

- What does 10.01_2 mean?
- C.f., Decimal
 - $12.45 = 1 \cdot 10^1 + 2 \cdot 10^0 + 4 \cdot 10^{-1} + 5 \cdot 10^{-2}$
- $10.01_2 = 1 \cdot 2^1 + 0 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} = 2.25_{10}$



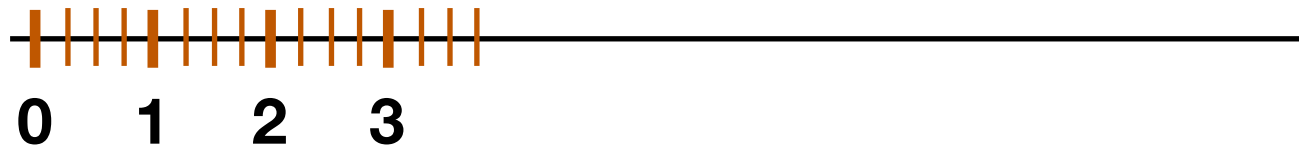
$$\begin{array}{r}
 01.10 \\
 + 01.01 \\
 \hline
 10.11
 \end{array}$$

$$\begin{array}{r}
 1.50 \\
 + 1.25 \\
 \hline
 2.75
 \end{array}$$

Decimal	Binary
0	00.00
0.25	00.01
0.5	00.10
0.75	00.11
1	01.00
1.25	01.01
1.5	01.10
1.75	01.11
2	10.00
2.25	10.01
2.5	10.10
2.75	10.11
3	11.00
3.25	11.01
3.5	11.10
3.75	11.11

Can We Represent Fractions in Binary?

- What does 10.01_2 mean?
- C.f., Decimal
 - $12.45 = 1 \cdot 10^1 + 2 \cdot 10^0 + 4 \cdot 10^{-1} + 5 \cdot 10^{-2}$
- $10.01_2 = 1 \cdot 2^1 + 0 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} = 2.25_{10}$



Integer Arithmetic Still Works!

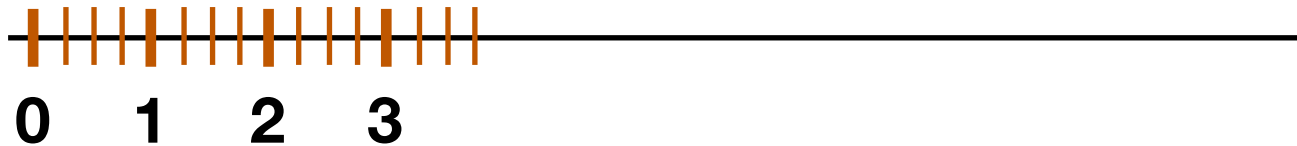
$$\begin{array}{r}
 01.10 \\
 + 01.01 \\
 \hline
 10.11
 \end{array}$$

$$\begin{array}{r}
 1.50 \\
 + 1.25 \\
 \hline
 2.75
 \end{array}$$

Decimal	Binary
0	00.00
0.25	00.01
0.5	00.10
0.75	00.11
1	01.00
1.25	01.01
1.5	01.10
1.75	01.11
2	10.00
2.25	10.01
2.5	10.10
2.75	10.11
3	11.00
3.25	11.01
3.5	11.10
3.75	11.11

Fixed-Point Representation

- Fixed interval between two representable numbers as long as the **binary point stays fixed**
 - Each bit represents 0.25_{10}
- **Fixed-point** representation of numbers
 - Integer is one special case of fixed-point

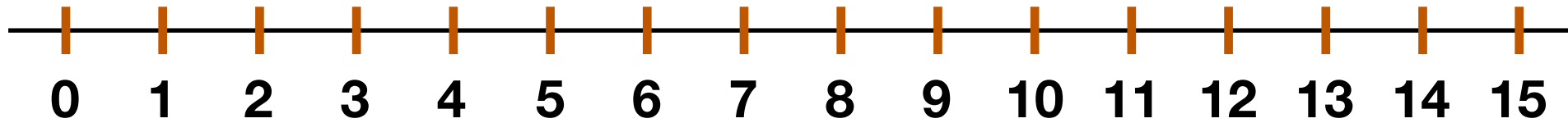


$$\begin{array}{r}
 01.10 \\
 + 01.01 \\
 \hline
 10.11
 \end{array}$$

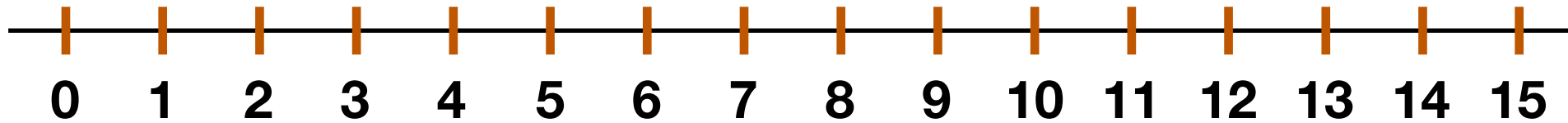
$$\begin{array}{r}
 1.50 \\
 + 1.25 \\
 \hline
 2.75
 \end{array}$$

Decimal	Binary
0	00.00
0.25	00.01
0.5	00.10
0.75	00.11
1	01.00
1.25	01.01
1.5	01.10
1.75	01.11
2	10.00
2.25	10.01
2.5	10.10
2.75	10.11
3	11.00
3.25	11.01
3.5	11.10
3.75	11.11

Aside: Quantization

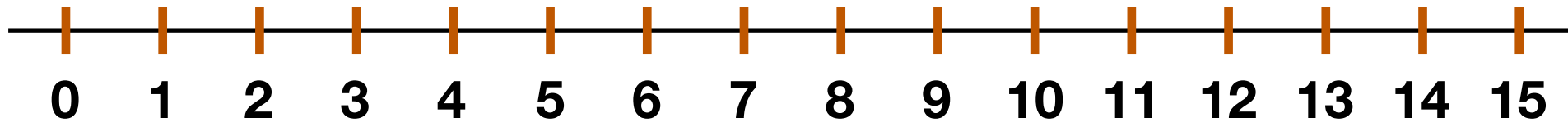


Aside: Quantization



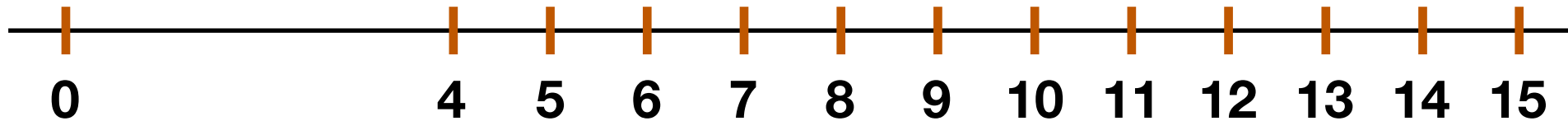
- Representing all integers **precisely** requires 4 bits

Aside: Quantization



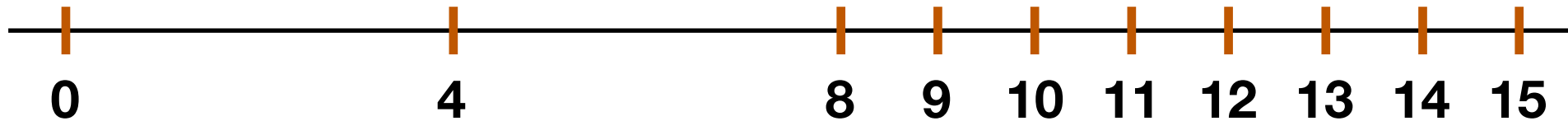
- Representing all integers **precisely** requires 4 bits
- What if we can tolerate some imprecisions
 - 1, 2, 3 are approximated by 0
 - 5, 6, 7 are approximated by 4...
 - We would only need 2 bits

Aside: Quantization



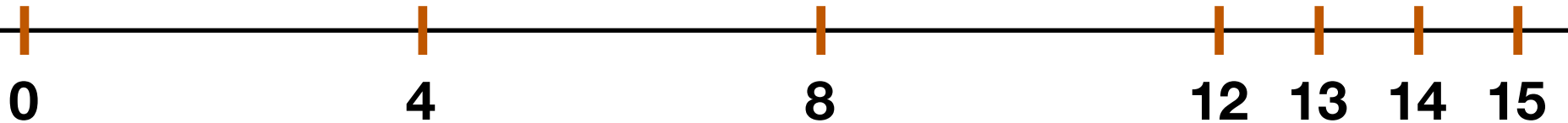
- Representing all integers **precisely** requires 4 bits
- What if we can tolerate some imprecisions
 - 1, 2, 3 are approximated by 0
 - 5, 6, 7 are approximated by 4...
 - We would only need 2 bits

Aside: Quantization



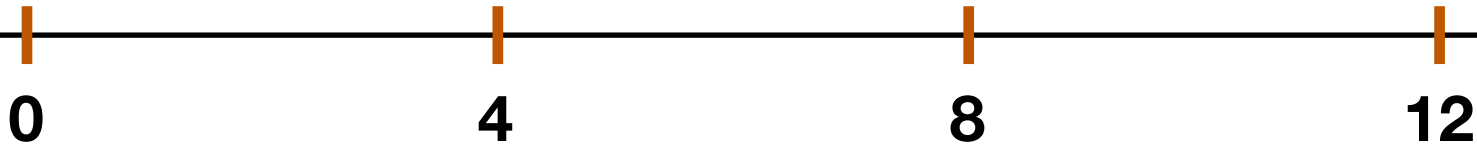
- Representing all integers **precisely** requires 4 bits
- What if we can tolerate some imprecisions
 - 1, 2, 3 are approximated by 0
 - 5, 6, 7 are approximated by 4...
 - We would only need 2 bits

Aside: Quantization



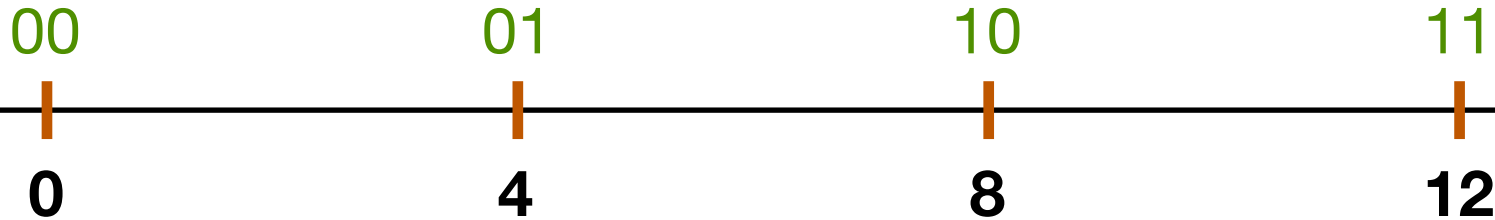
- Representing all integers **precisely** requires 4 bits
- What if we can tolerate some imprecisions
 - 1, 2, 3 are approximated by 0
 - 5, 6, 7 are approximated by 4...
 - We would only need 2 bits

Aside: Quantization



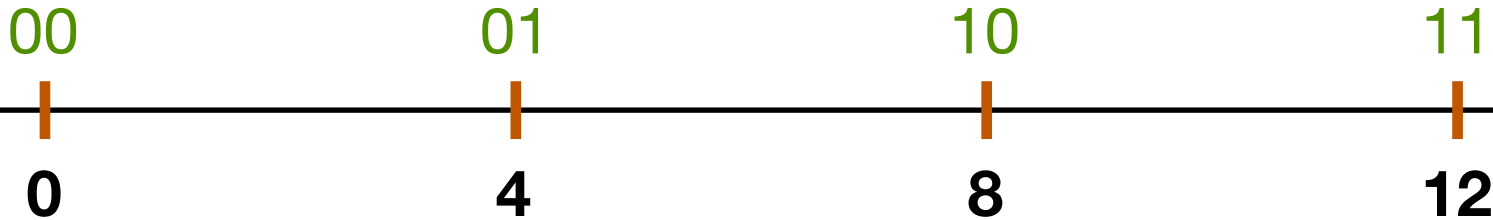
- Representing all integers **precisely** requires 4 bits
- What if we can tolerate some imprecisions
 - 1, 2, 3 are approximated by 0
 - 5, 6, 7 are approximated by 4...
 - We would only need 2 bits

Aside: Quantization



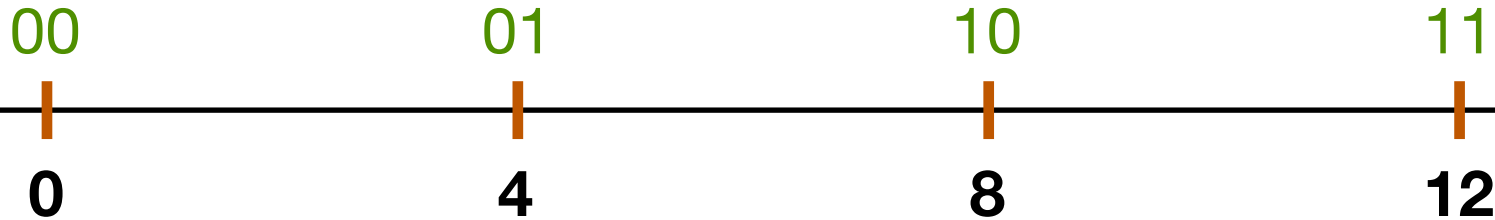
- Representing all integers **precisely** requires 4 bits
- What if we can tolerate some imprecisions
 - 1, 2, 3 are approximated by 0
 - 5, 6, 7 are approximated by 4...
 - We would only need 2 bits

Aside: Quantization



- Representing all integers **precisely** requires 4 bits
- What if we can tolerate some imprecisions
 - 1, 2, 3 are approximated by 0
 - 5, 6, 7 are approximated by 4...
 - We would only need 2 bits
- That is, 1 bit represents 4_{10}
 - 10_2 becomes $4 * (1 * 2^1) = 8$
 - Every time we increment a bit, the value is incremented by 4
 - 1, 2, 3 are represented **approximately** by 10_2

Aside: Quantization



- Representing all integers **precisely** requires 4 bits

- What if

- 1, 2
- 5, 6
- We

Note that this is different from “base 4”

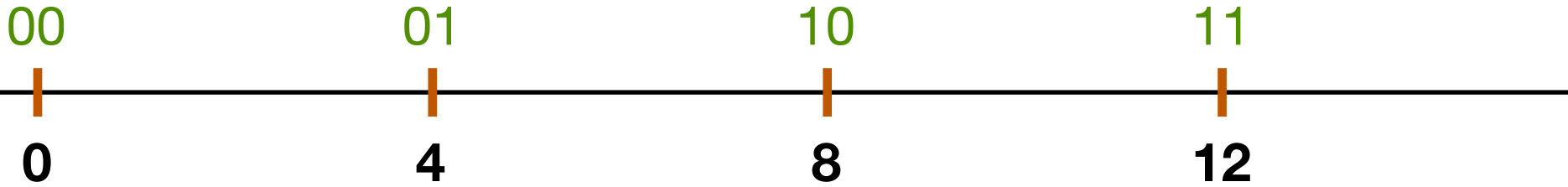
- $10_4 = 1 * 4^1 + 0 * 4^0 = 4$

- Every increment still only increments 1

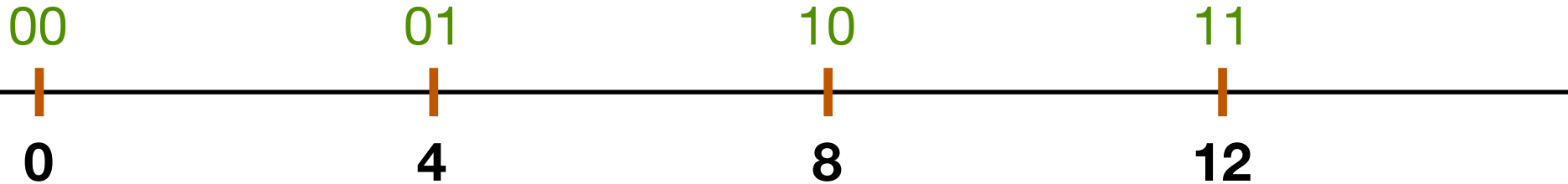
- That is, 1 bit represents 4_{10}

- 10_2 becomes $4 * (1 * 2^1) = 8$
- Every time we increment a bit, the value is incremented by 4
- 1, 2, 3 are represented **approximately** by 10_2

Aside: Quantization

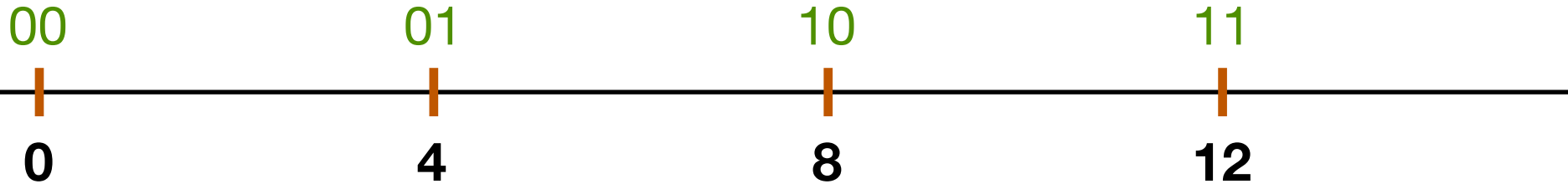


Aside: Quantization



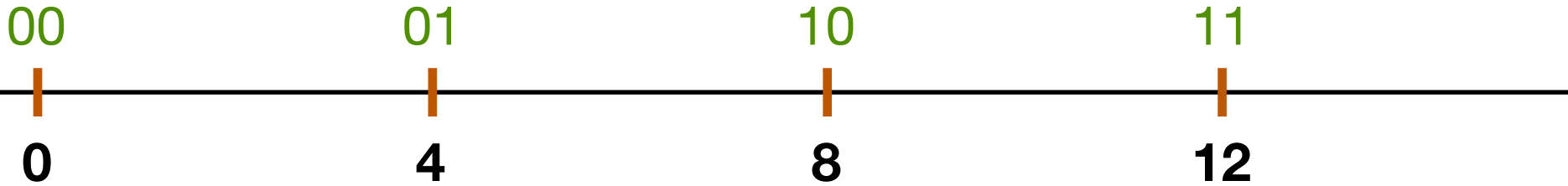
- Saves storage space and improves computation speed
 - 50% space saving
 - 4-bit arithmetic becomes 2-bit arithmetic

Aside: Quantization

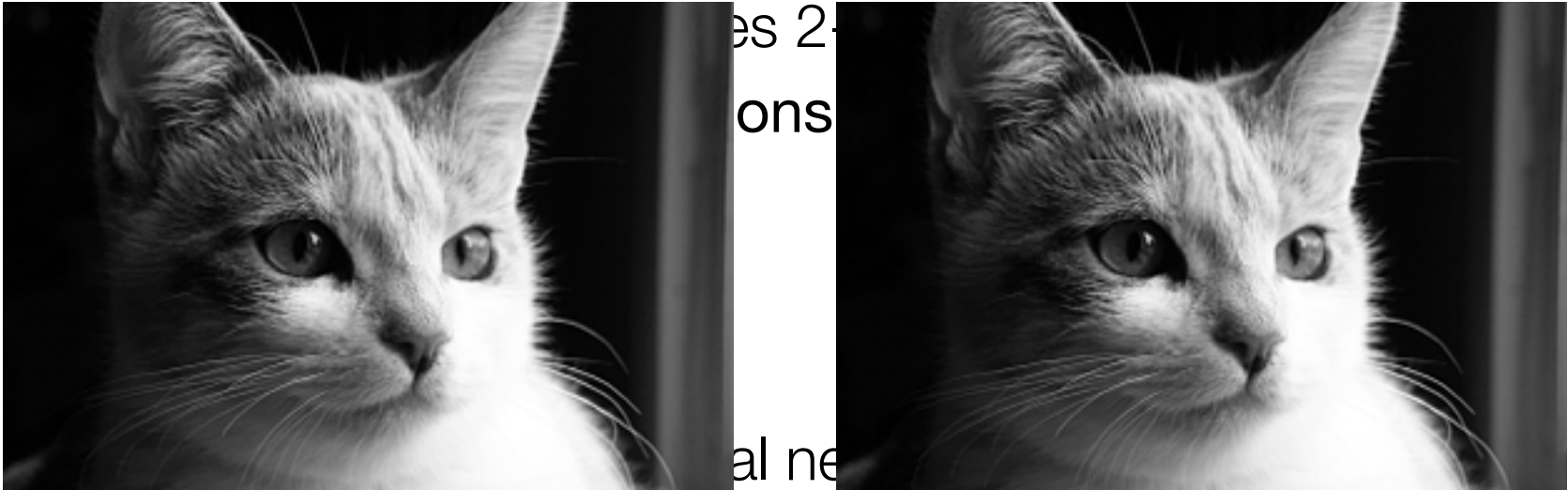


- Saves storage space and improves computation speed
 - 50% space saving
 - 4-bit arithmetic becomes 2-bit arithmetic
- Many real-world applications can tolerate imprecisions
 - Image processing
 - Computer vision
 - Real-time graphics
 - Machine learning (Neural networks)

Aside: Quantization

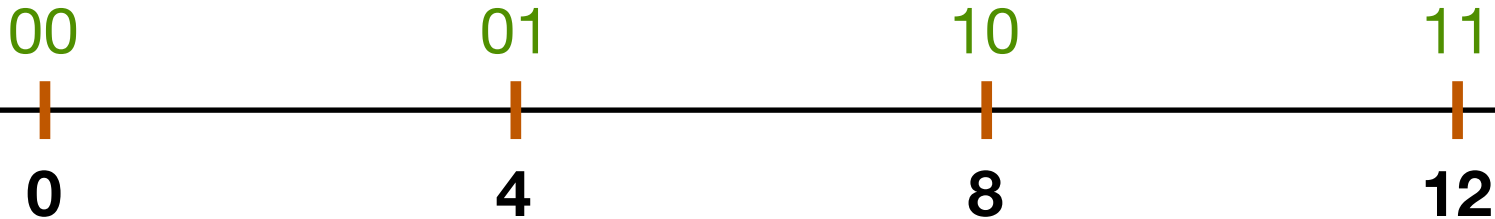


- Saves storage space and improves computation speed
 - 50% space saving



Aside: Quantization

Questions?



- Saves storage space and improves computation speed
 - 50% space saving



es 2
ons

al ne

