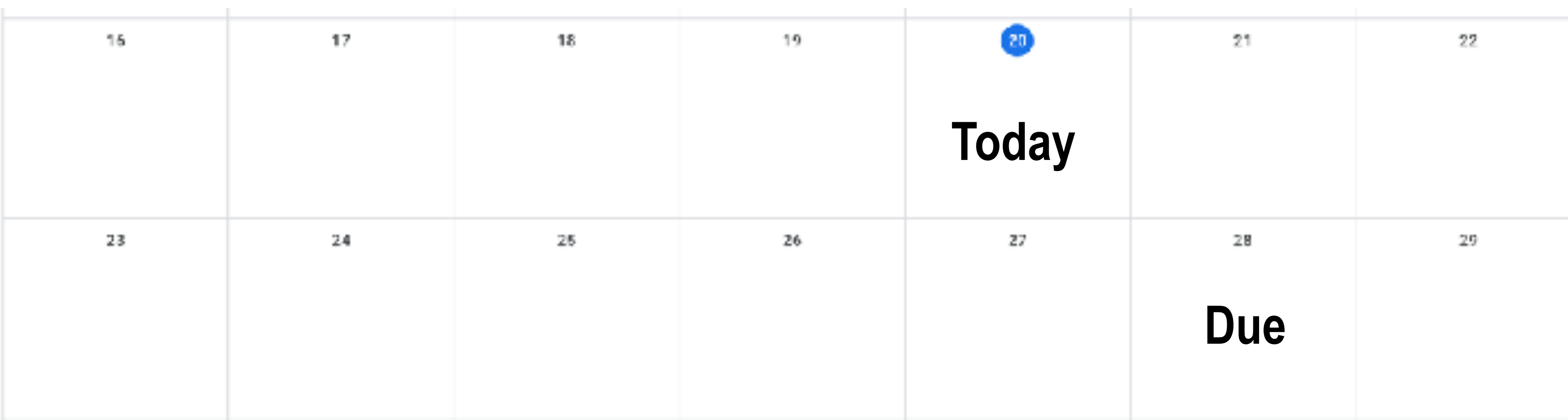
CSC 252: Computer Organization
Spring 2022: Lecture 3

Instructor: Yuhao Zhu

Department of Computer Science
University of Rochester

Announcement

- Programming Assignment 1 is out
  - Details: https://www.cs.rochester.edu/courses/252/spring2022/labs/assignment1.html
  - Due on Jan. 28, 11:59 PM
  - You have 3 slip days



Announcement

- Programming assignment 1 is in C language. Seek help from TAs.
- TAs are best positioned to answer your questions about programming assignments!!!
- Programming assignments do NOT repeat the lecture materials. They ask you to synthesize what you have learned from the lectures and work out something new.

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

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

```
    010
+)  101
-------
    111
```

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

```
    010
+)  101
-------
    111
```

```
      2
+)   -1
-------
     -3
```

Sign-Magnitude Implications

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

```
     010            2
+)   101      +)   -1
---------     --------
     111           -3
```

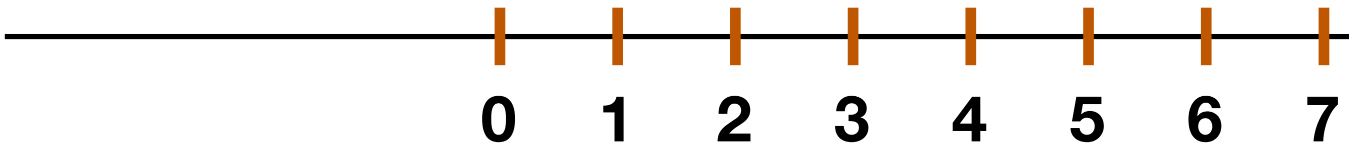
Signed Value	Binary
0	000
1	001
2	010
3	011
-0	100
-1	101
-2	110
-3	111

Encoding Negative Numbers

- Solution 2: Two’s Complement

Encoding Negative Numbers

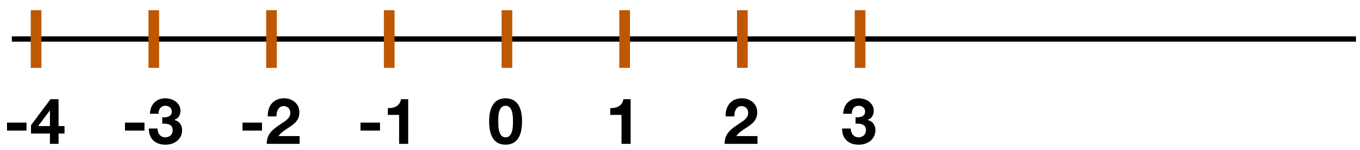
- Solution 2: Two’s Complement



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

Encoding Negative Numbers

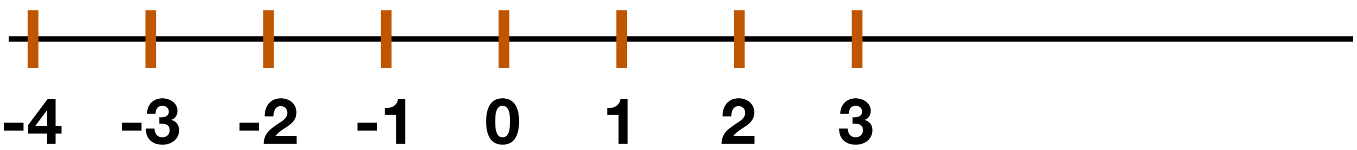
- Solution 2: Two's Complement



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

Encoding Negative Numbers

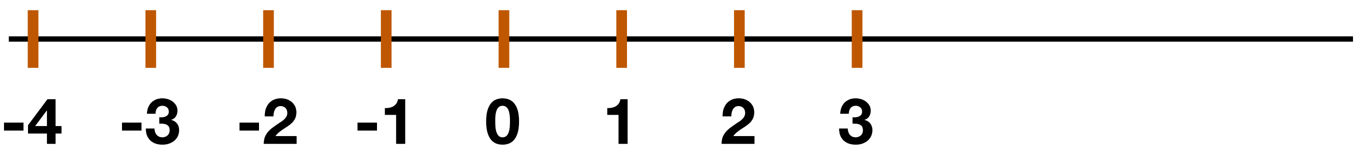
- Solution 2: Two's Complement



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

Encoding Negative Numbers

- Solution 2: Two's Complement

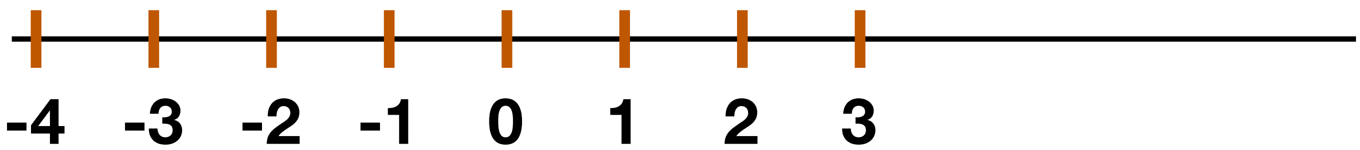


Signed Weight	Unsigned Weight	Bit Position
2^0	2^0	0
2^1	2^1	1
-2^2	2^2	2

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

Encoding Negative Numbers

- Solution 2: Two's Complement

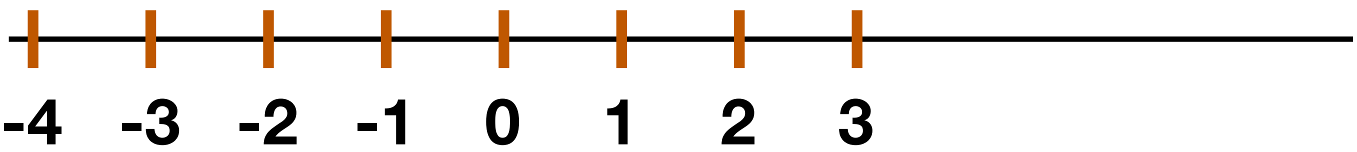


Signed Weight	Unsigned Weight	Bit Position
2^0	2^0	0
2^1	2^1	1
-2^2	2^2	2

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

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

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

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

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 the 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 the 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

```
    010
+)  101
-------
    111
```

Two-Complement Implications

- Only 1 zero
- Usual arithmetic still works
- There is a bit that represents the 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

```
    010
+)  101
-------
    111
```

```
      2
+)   -3
-------
     -1
```

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
`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

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
    - $12.45 = 1*10^1 + 2*10^0 + 4*10^{-1} + 5*10^{-2}$
- $10.01_2 = 1*2^1 + 0*2^0 + 0*2^{-1} + 1*2^{-2} = 2.25_{10}$

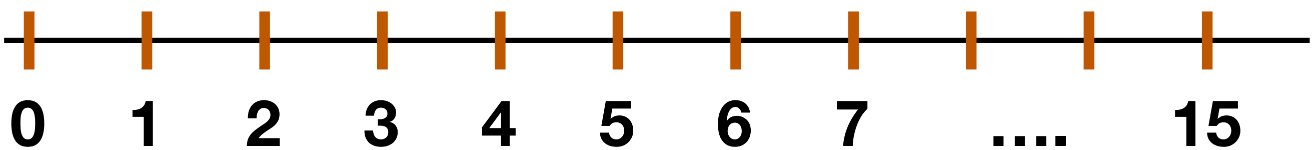
Can We Represent Fractions in Binary?

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

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?

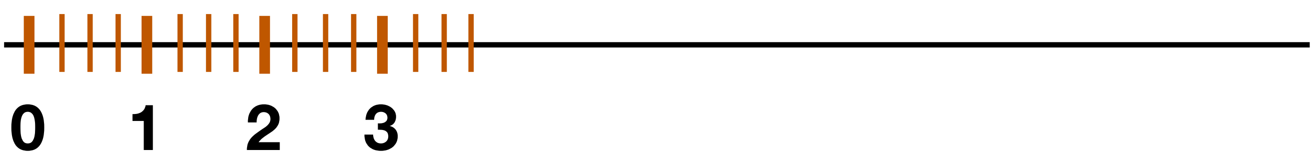
- What does 10.01_2 mean?
- C.f., Decimal
  - $12.45 = 1*10^1 + 2*10^0 + 4*10^{-1} + 5*10^{-2}$
- $10.01_2 = 1*2^1 + 0*2^0 + 0*2^{-1} + 1*2^{-2} = 2.25_{10}$



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?

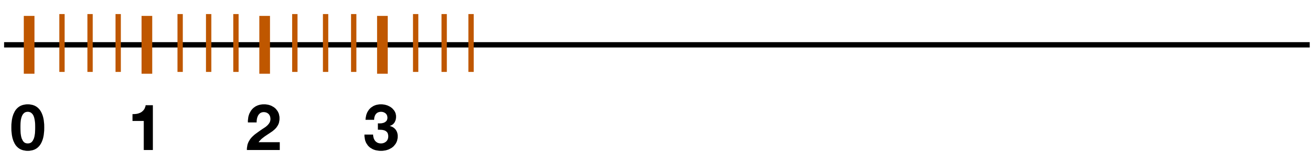
- What does 10.01_2 mean?
- C.f., Decimal
  - $12.45 = 1*10^1 + 2*10^0 + 4*10^{-1} + 5*10^{-2}$
- $10.01_2 = 1*2^1 + 0*2^0 + 0*2^{-1} + 1*2^{-2} = 2.25_{10}$



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
  - $12.45 = 1*10^1 + 2*10^0 + 4*10^{-1} + 5*10^{-2}$
- $10.01_2 = 1*2^1 + 0*2^0 + 0*2^{-1} + 1*2^{-2} = 2.25_{10}$



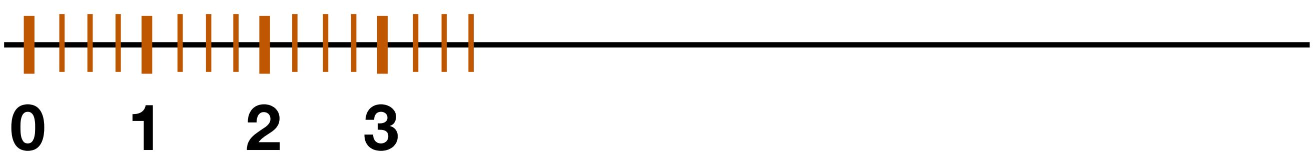

```
  01.10
+ 01.01
-------
  10.11
```

```
  1.50
+ 1.25
------
  2.75
```

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
  - $12.45 = 1*10^1 + 2*10^0 + 4*10^{-1} + 5*10^{-2}$
- $10.01_2 = 1*2^1 + 0*2^0 + 0*2^{-1} + 1*2^{-2} = 2.25_{10}$



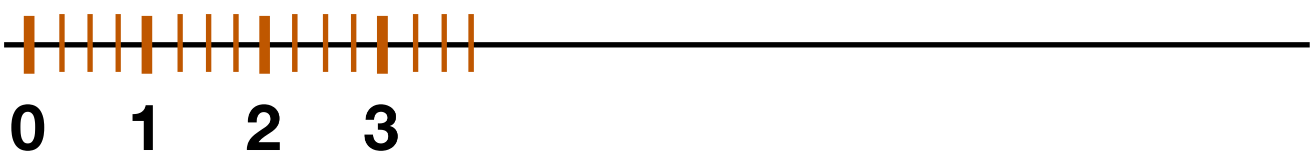
Integer Arithmetic Still Works!

```
   01.10          1.50
 + 01.01        + 1.25
 -------        ------
   10.11          2.75
```

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
  - The interval is 0.25_{10} here
- Fixed-point representation of numbers
  - Integer is one special case of fixed-point




```
  01.10
+ 01.01
-------
  10.11
```

```
  1.50
+ 1.25
------
  2.75
```

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

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

One Bit Sequence, Two Interpretations

- A sequence of bits can be interpreted as either a signed integer or an unsigned integer

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

Signed vs. Unsigned Conversion in C

- What happens when we convert between signed and unsigned numbers?
- Casting (In C terminology)
  - Explicit casting between signed & unsigned

```
int tx, ty = -4;
unsigned ux = 7, uy;
tx = (int) ux; // U2T
uy = (unsigned) ty; // T2U
```

  - Implicit casting
    - e.g., assignments, function calls

```
tx = ux;
uy = ty;
```

Mapping Between Signed & Unsigned

- Mappings between unsigned and two's complement numbers: Keep bit representations and reinterpret

Mapping Between Signed & Unsigned

- Mappings between unsigned and two's complement numbers: Keep bit representations and reinterpret

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

Mapping Signed ↔ Unsigned

Bits	Signed	Unsigned
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	-7	9
1010	-6	10
1011	-5	11
1100	-4	12
1101	-3	13
1110	-2	14
1111	-1	15

Mapping Signed ↔ Unsigned

Bits	Signed	Unsigned
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	-7	9
1010	-6	10
1011	-5	11
1100	-4	12
1101	-3	13
1110	-2	14
1111	-1	15

T2U

Mapping Signed ↔ Unsigned

Bits	Signed	Unsigned
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	-7	9
1010	-6	10
1011	-5	11
1100	-4	12
1101	-3	13
1110	-2	14
1111	-1	15

Signed $\xrightarrow{\text{T2U}}$ Unsigned

Signed $\xleftarrow{\text{U2T}}$ Unsigned

Mapping Signed ↔ Unsigned

Bits	Signed	Unsigned
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	-7	9
1010	-6	10
1011	-5	11
1100	-4	12
1101	-3	13
1110	-2	14
1111	-1	15

Mapping Signed ↔ Unsigned

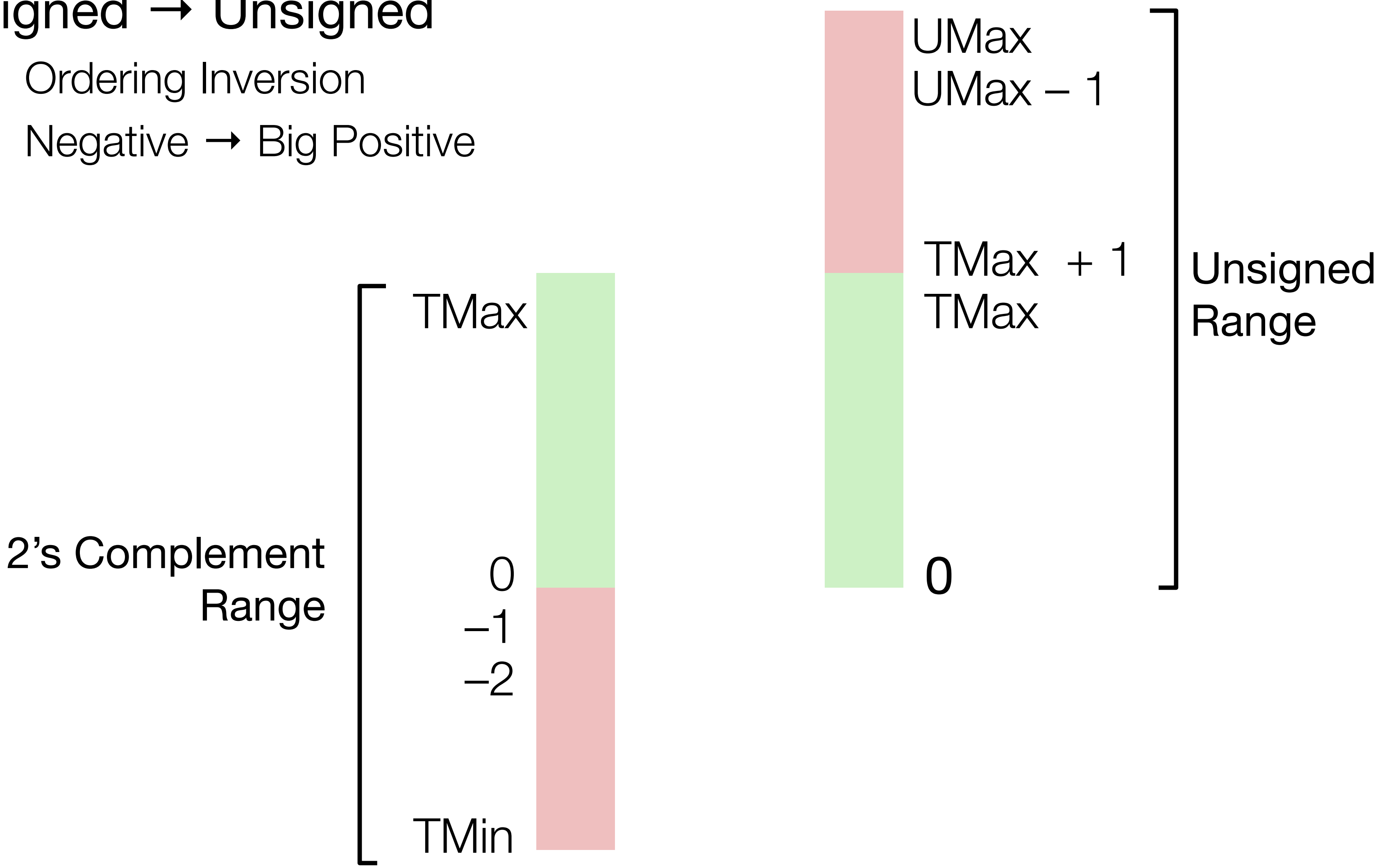
Bits	Signed	Unsigned
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	-8	8
1001	-7	9
1010	-6	10
1011	-5	11
1100	-4	12
1101	-3	13
1110	-2	14
1111	-1	15

=

+/- 16

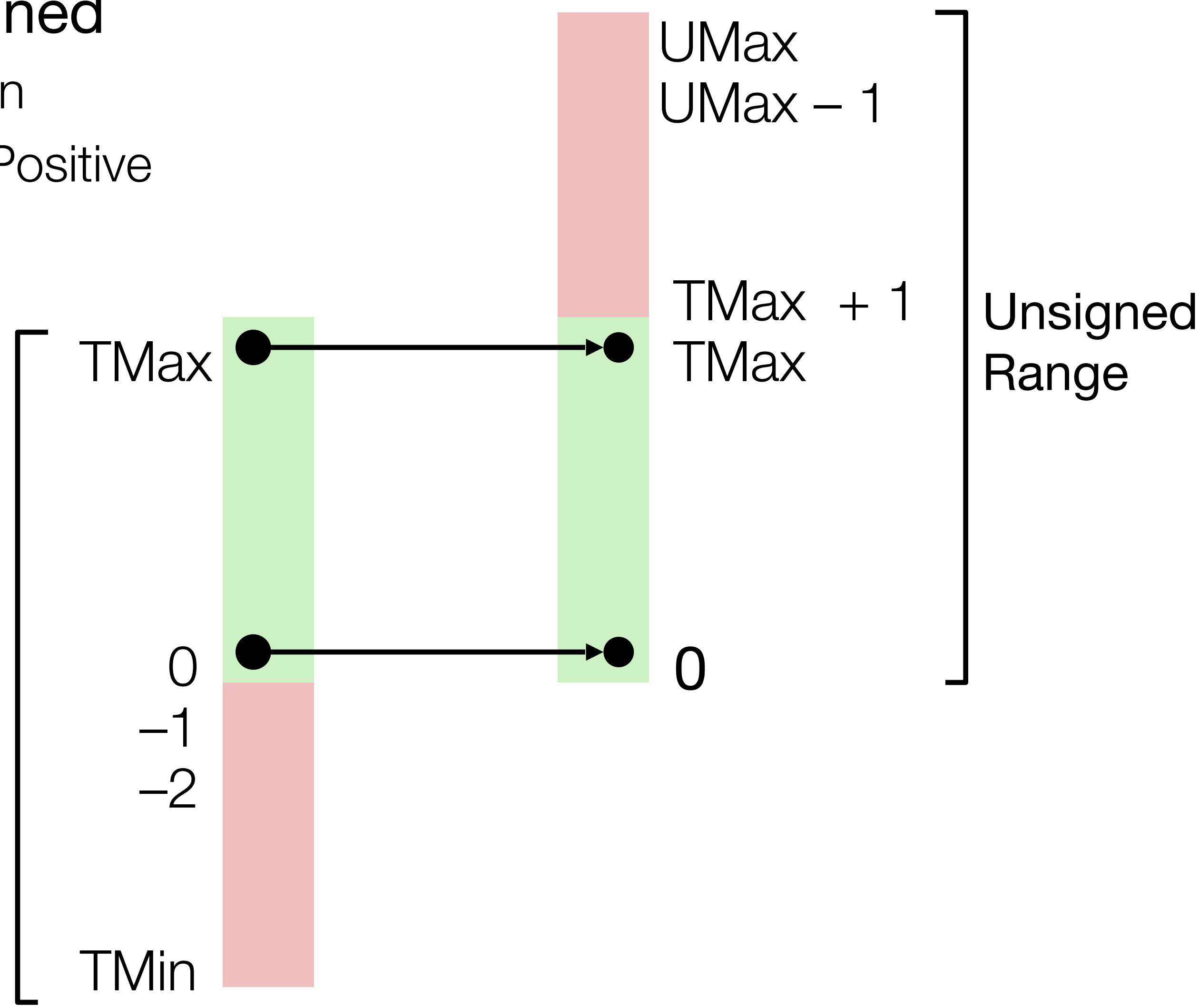
Conversion Visualized

- Signed → Unsigned
  - Ordering Inversion
  - Negative → Big Positive



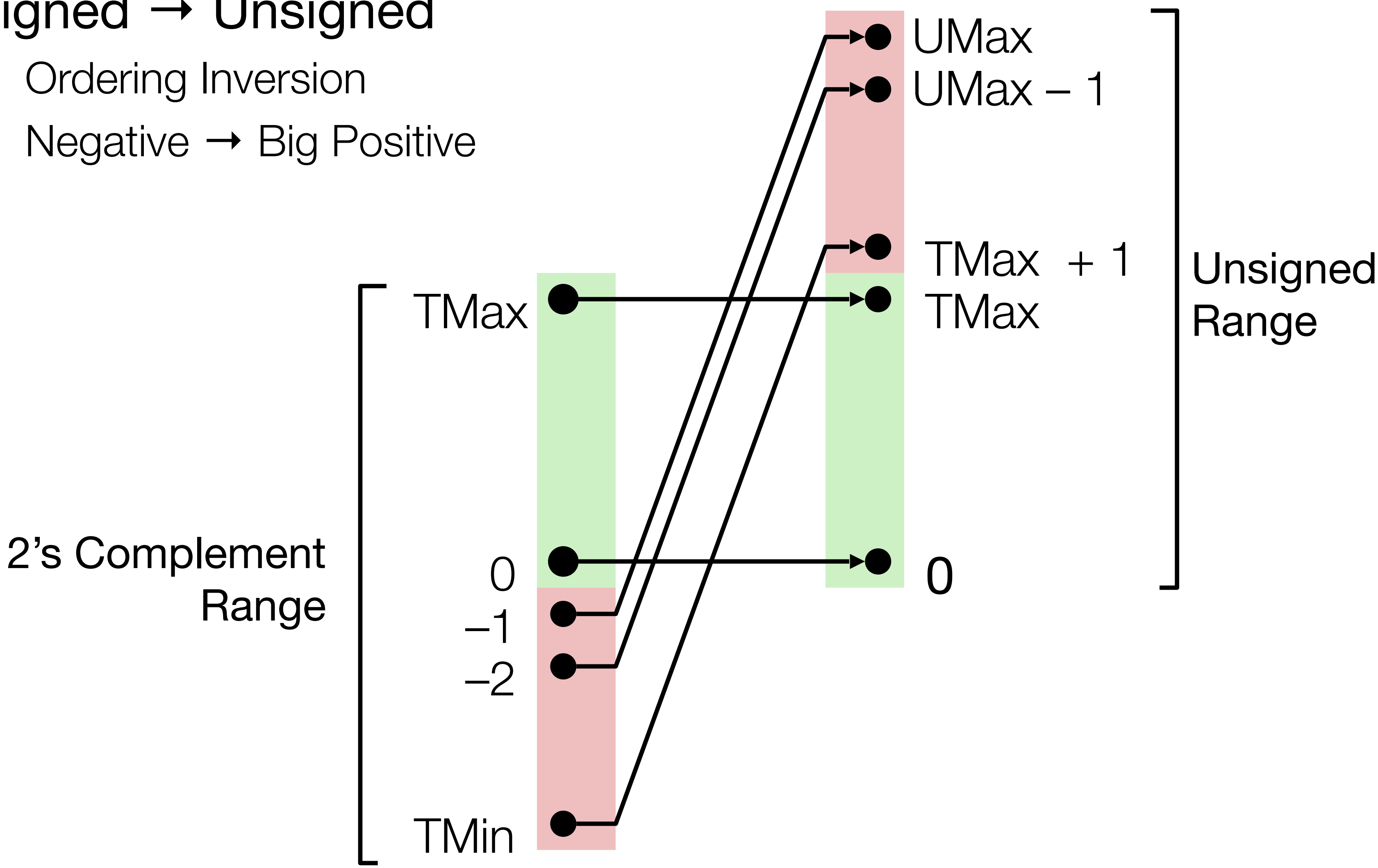
Conversion Visualized

- Signed → Unsigned
  - Ordering Inversion
  - Negative → Big Positive



Conversion Visualized

- Signed → Unsigned
  - Ordering Inversion
  - Negative → Big Positive



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

The Problem

```
short int x =  15213;
int       ix = (int) x;
short int y = -15213;
int       iy = (int) y;
```

C Data Type	# of Bytes
char	1
short	2
int	4
long	8

The Problem

C Data Type	# of Bytes
`char`	1
`short`	2
`int`	4
`long`	8

```
short int x =  15213;
int       ix = (int) x;
short int y = -15213;
int       iy = (int) y;
```

- Converting from smaller to larger integer data type
- Should we preserve the value?
- Can we preserve the value?
- How?

The Problem

```
short int x =  15213;
int       ix = (int) x;
short int y = -15213;
int       iy = (int) y;
```

C Data Type	# of Bytes
`char`	1
`short`	2
`int`	4
`long`	8

- Converting from smaller to larger integer data type
- Should we preserve the value?
- Can we preserve the value?
- How?

	Decimal	Hex	Binary
x	15213	3B 6D	00111011 01101101
ix	15213	00 00 3B 6D	00000000 00000000 00111011 01101101
y	-15213	C4 93	11000100 10010011
iy	-15213	FF FF C4 93	11111111 11111111 11000100 10010011

Signed Extension

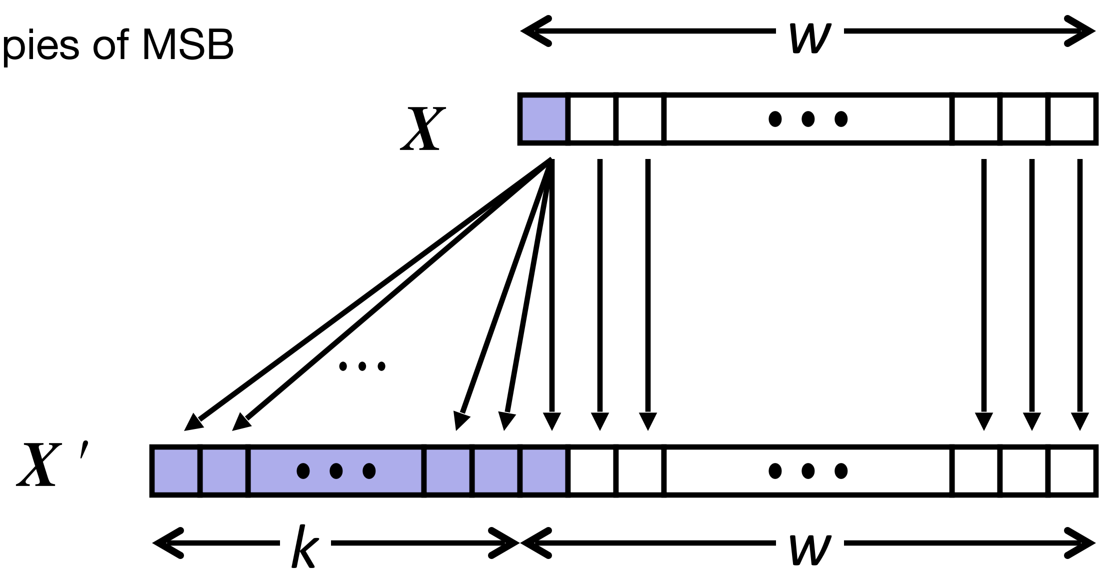
- Task:
  - Given w-bit signed integer x
  - Convert it to $(w+k)$-bit integer with same value

Signed Extension

- Task:
  - Given w-bit signed integer x
  - Convert it to $(w+k)$-bit integer with same value
- Rule:
  - Make k copies of sign bit:
  - $X' = \underbrace{x_{w-1}, \ldots, x_{w-1}}_{k \text{ copies of MSB}}, x_{w-1}, x_{w-2}, \ldots, x_0$

Signed Extension

- Task:
  - Given w-bit signed integer x
  - Convert it to $(w+k)$-bit integer with same value
- Rule:
  - Make k copies of sign bit:
  - $X' = \underbrace{x_{w-1}, \ldots, x_{w-1}}_{k \text{ copies of MSB}}, x_{w-1}, x_{w-2}, \ldots, x_0$



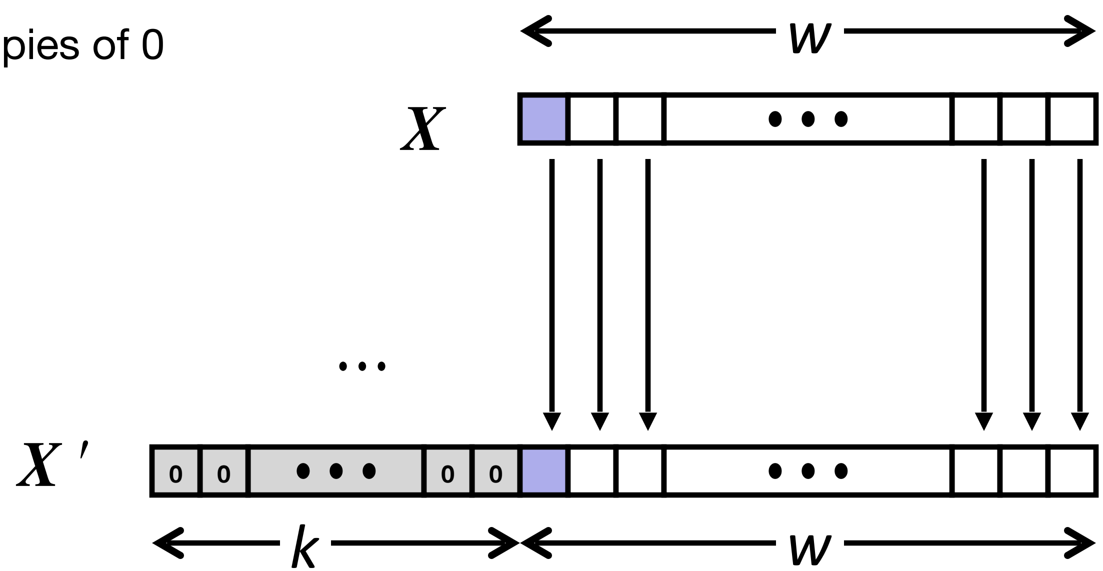
Another Problem

```
unsigned short x = 47981;
unsigned int  ux = x;
```

	Decimal	Hex	Binary
x	47981	BB 6D	10111011 01101101
ux	47981	00 00 BB 6D	00000000 00000000 10111011 01101101

Unsigned (Zero) Extension

- Task:
  - Given w-bit unsigned integer x
  - Convert it to $(w+k)$-bit integer with same value
- Rule:
  - Simply pad zeros:
  - $X' = \underbrace{0, \ldots, 0}_{k \text{ copies of } 0}, x_{w-1}, x_{w-2}, \ldots, x_0$



Yet Another Problem

```
int    x =  53191;
short sx = (short) x;
```

	Decimal	Hex	Binary
x	53191	00 00 CF C7	00000000 00000000 11001111 11000111
sx	-12345	CF C7	11001111 11000111

Yet Another Problem

```
int    x =  53191;
short sx = (short) x;
```

	Decimal	Hex	Binary
x	53191	00 00 CF C7	00000000 00000000 11001111 11000111
sx	-12345	CF C7	11001111 11000111

- Truncating (e.g., int to short)
  - C's implementation: leading bits are truncated, results reinterpreted
  - So can't always preserve the numerical value

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

Unsigned Addition

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

Unsigned Addition

- Similar to Decimal Addition

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

Unsigned Addition

- Similar to Decimal Addition
- Suppose we have a new data type that is 3-bit wide (c.f., **`short`** has 16 bits)

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

Normal Case

```
    010
+)  101
-------
    111
```

```
      2
+)    5
-------
      7
```

Unsigned Addition

- Similar to Decimal Addition
- Suppose we have a new data type that is 3-bit wide (c.f., `short` has 16 bits)
- Might overflow: result can't be represented within the size of the data type

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

Normal Case

```
    010          2
+)  101      +)  5
-------      -----
    111          7
```

Overflow Case

```
    110          6
+)  101      +)  5
-------      -----
   1011         11
```

Unsigned Addition

- Similar to Decimal Addition
- Suppose we have a new data type that is 3-bit wide (c.f., **`short`** has 16 bits)
- Might overflow: result can't be represented within the size of the data type

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

Normal Case

```
    010          2
+)  101      +)  5
-------      -----
    111          7
```

Overflow Case

```
    110          6
+)  101      +)  5
-------      -----
   1011         11
```

True Sum

Unsigned Addition

- Similar to Decimal Addition
- Suppose we have a new data type that is 3-bit wide (c.f., **`short`** has 16 bits)
- Might overflow: result can't be represented within the size of the data type

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

Normal Case

```
    010          2
+)  101     +)   5
-------     ------
    111          7
```

Overflow Case

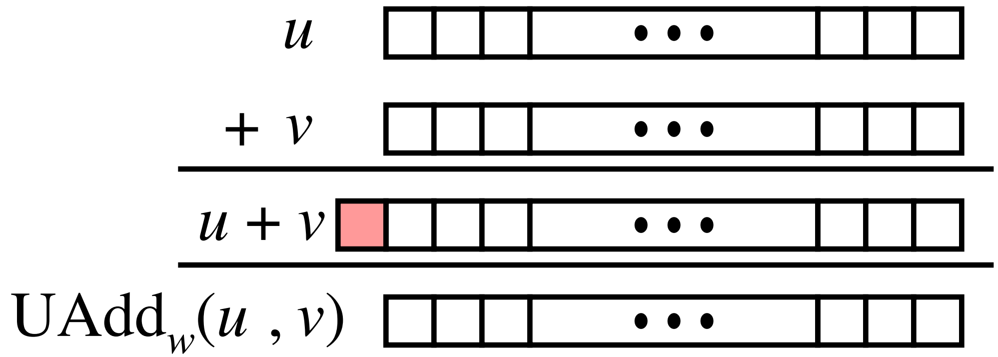
```
    110          6
+)  101     +)   5
-------     ------
   1011         11     True Sum
    011          3     Sum with same bits
```

Unsigned Addition in C

Operands: *w* bits

True Sum: *w*+1 bits

Discard Carry: *w* bits



Two's Complement Addition

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

Two’s Complement Addition

- Has identical bit-level behavior as unsigned addition (a big advantage over sign-magnitude)

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

Two's Complement Addition

- Has identical bit-level behavior as unsigned addition (a big advantage over sign-magnitude)

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

Normal Case

```
    010          2
+)  101     +)  -3
-------     ------
    111         -1
```

Two's Complement Addition

- Has identical bit-level behavior as unsigned addition (a big advantage over sign-magnitude)
- Overflow can also occur

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

Normal Case

```
    010          2
+)  101      +) -3
-------      -----
    111         -1
```

Overflow Case

```
    110         -2
+)  101      +) -3
-------      -----
   1011         -5
```

Two's Complement Addition

- Has identical bit-level behavior as unsigned addition (a big advantage over sign-magnitude)
- Overflow can also occur

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

Normal Case

```
    010          2
+)  101     +)  -3
-------     -------
    111         -1
```

Overflow Case

```
    110         -2
+)  101     +)  -3
-------     -------
   1011         -5
    011          3
```

Two's Complement Addition

- Has identical bit-level behavior as unsigned addition (a big advantage over sign-magnitude)
- Overflow can also occur

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

Min → -4

Normal Case

```
    010          2
+)  101      +) -3
-------      -----
    111         -1
```

Overflow Case

```
    110         -2
+)  101      +) -3
-------      -----
   1011         -5
    011          3
```

Negative Overflow

Two's Complement Addition

- Has identical bit-level behavior as unsigned addition (a big advantage over sign-magnitude)
- Overflow can also occur

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

Min → -4

Normal Case

```
     010          2
+)   101     +)  -3
   -----        ----
     111         -1
```

Overflow Case

```
     110         -2          011           3
+)   101     +)  -3     +)   001     +)    1
   -----        ----       -----        ----
    1011         -5         0100           4
     011          3
```

Negative Overflow

Two's Complement Addition

- Has identical bit-level behavior as unsigned addition (a big advantage over sign-magnitude)
- Overflow can also occur

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

Min → -4 (100)

Normal Case

```
    010          2
+)  101      +) -3
-------      -----
    111         -1
```

Overflow Case

```
    110         -2         011          3
+)  101      +) -3      +) 001      +)  1
-------      -----      -------     -----
   1011         -5        0100          4
    011          3         100         -4
```

Negative Overflow

Two’s Complement Addition

- Has identical bit-level behavior as unsigned addition (a big advantage over sign-magnitude)
- Overflow can also occur

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

Max
Min

Normal Case

```
    010         2
+)  101     +) -3
-------     -----
    111        -1
```

Overflow Case

```
    110        -2         011          3
+)  101     +) -3     +)  001     +)   1
-------     -----     -------     ------
   1011        -5        0100          4
    011         3         100         -4
```

Negative Overflow

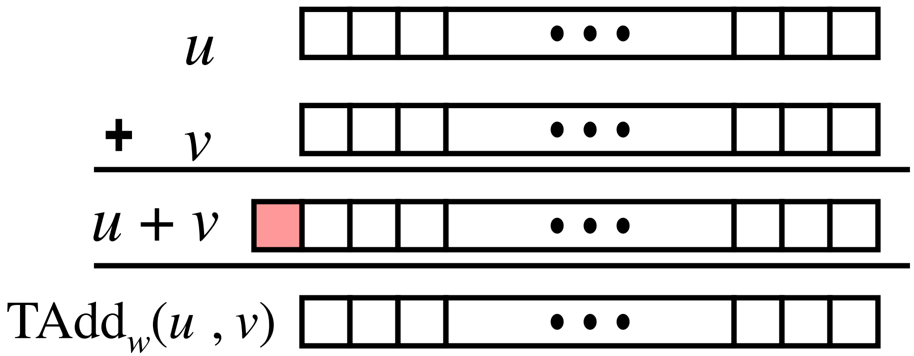
Positive Overflow

Two's Complement Addition in C

Operands: *w* bits

True Sum: *w*+1 bits

Discard Carry: *w* bits



Is This Signed Addition an Overflow?

```
    111
+)  110
-------
   1101
```

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

Is This Signed Addition an Overflow?

```
    111
+)  110
-------
   1101
```

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

Is This Signed Addition an Overflow?

```
    111
+)  110
-------
   1101
```

Truncate →

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

Is This Signed Addition an Overflow?

```
    111
+)  110
-------
   1101
```

$\xrightarrow{\text{Truncate}}$

```
     -1
+)   -2
-------
     -3
```

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

Is This Signed Addition an Overflow?

$$\begin{array}{r} 111 \\ +)\ 110 \\ \hline \boxed{1}101 \end{array} \xrightarrow[\textbf{Truncate}]{} \begin{array}{r} -1 \\ +)\ -2 \\ \hline -3 \end{array}$$

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

- This is not an overflow by definition

Is This Signed Addition an Overflow?

```
    111
+)  110
-------
   1101
```

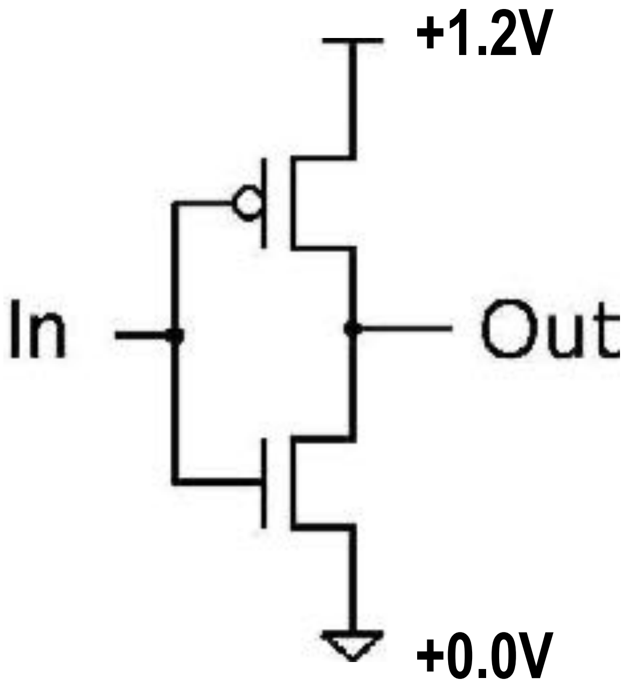
Truncate →

```
     -1
+)   -2
-------
     -3
```

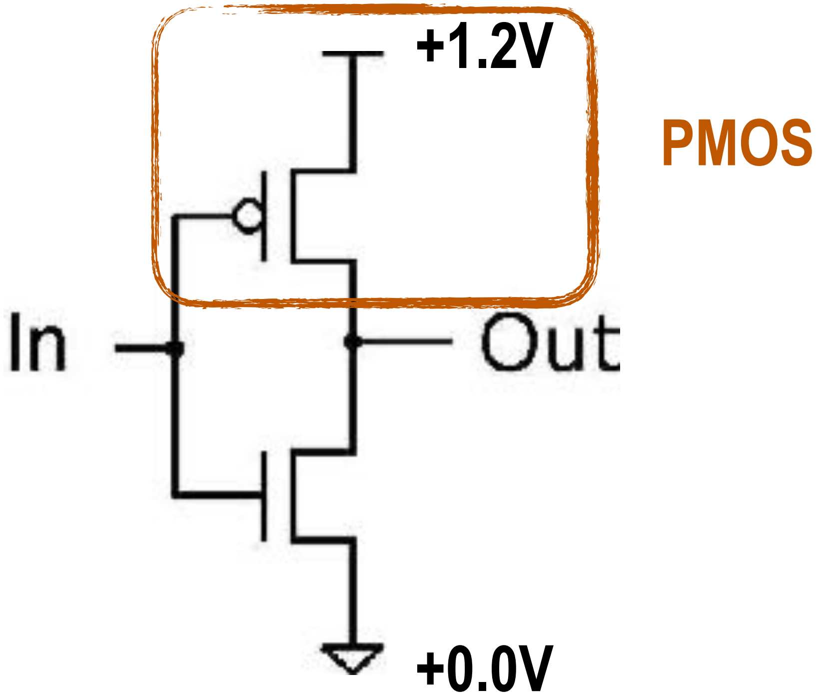
Signed	Binary
0	000
1	001
2	010
3	011
-4	100
-3	101
-2	110
-1	111

- This is not an overflow by definition
- Because the actual result can be represented using the bit width of the datatype (3 bits here)

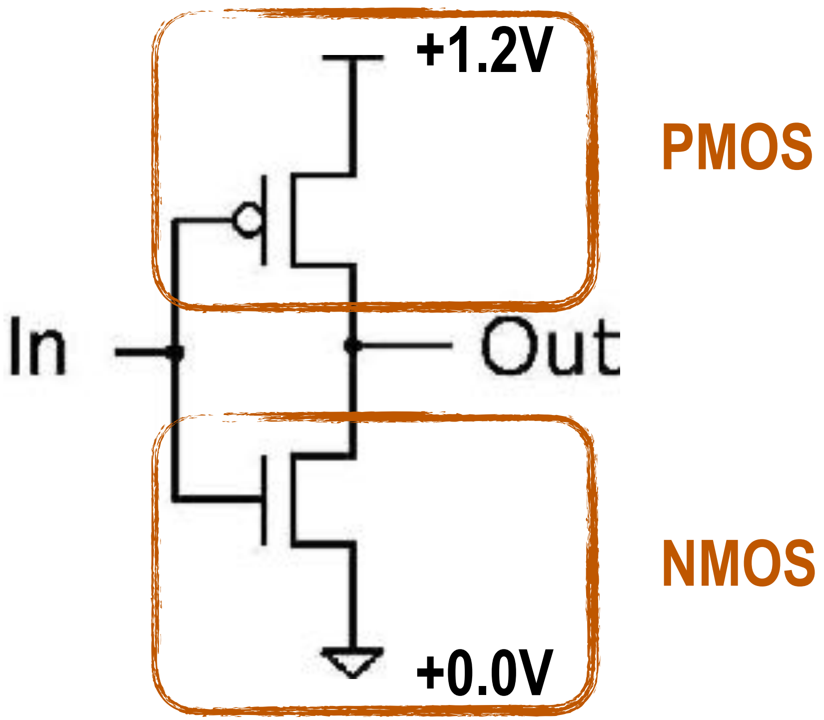
Inverter (NOT Gate)



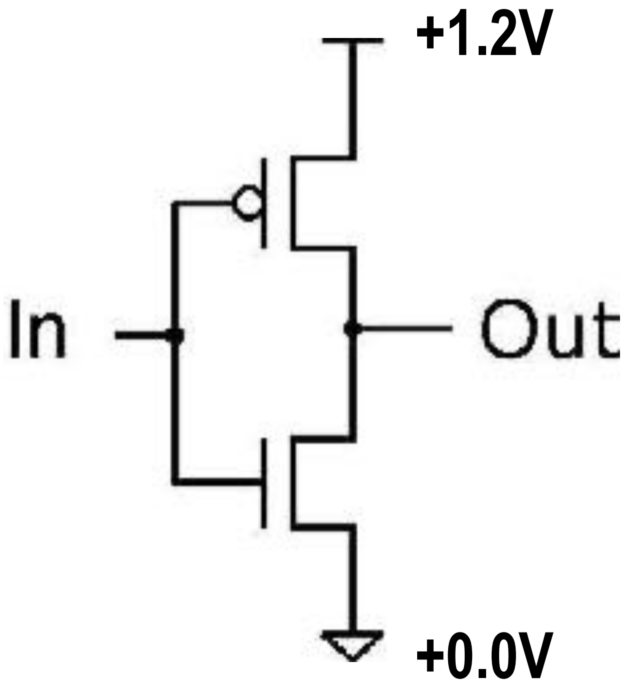
Inverter (NOT Gate)

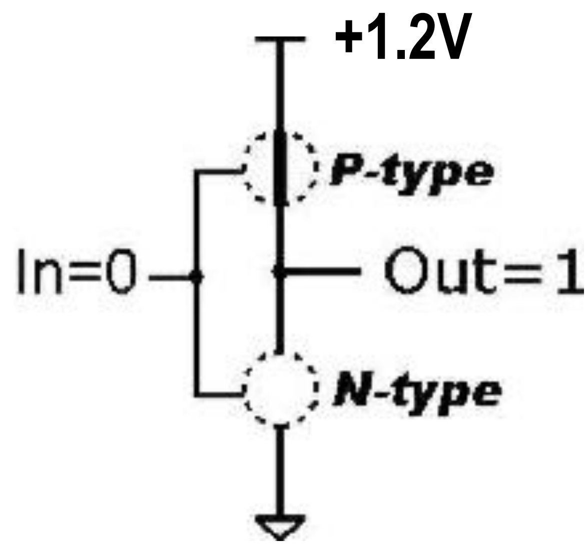


Inverter (NOT Gate)

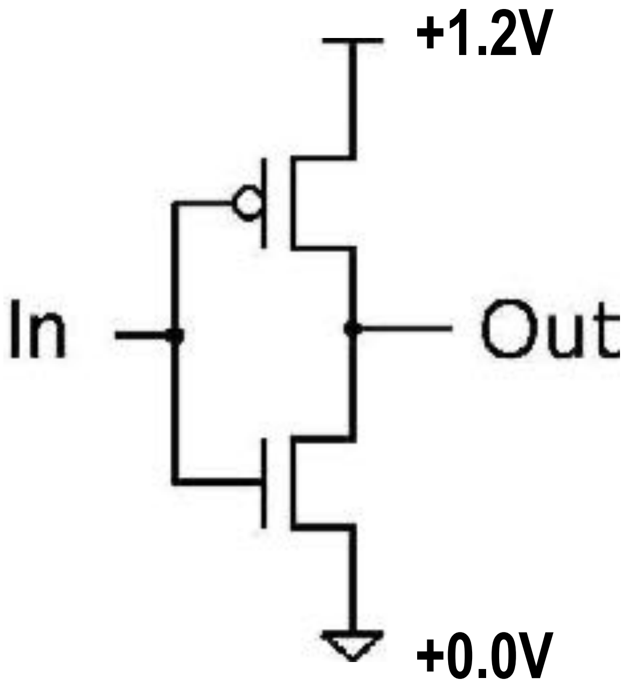


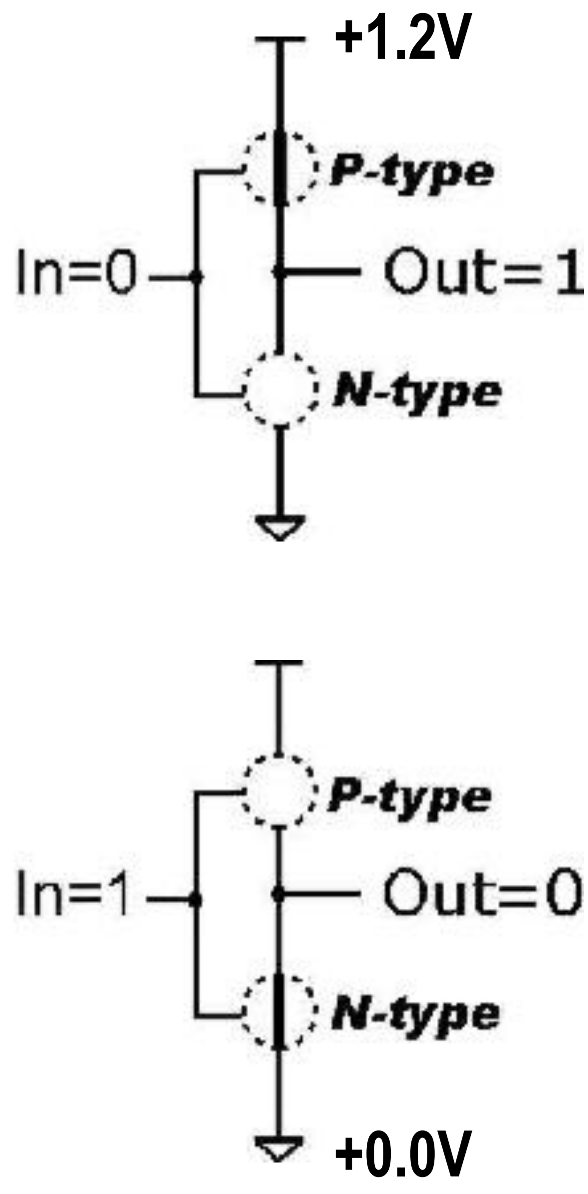
Inverter (NOT Gate)



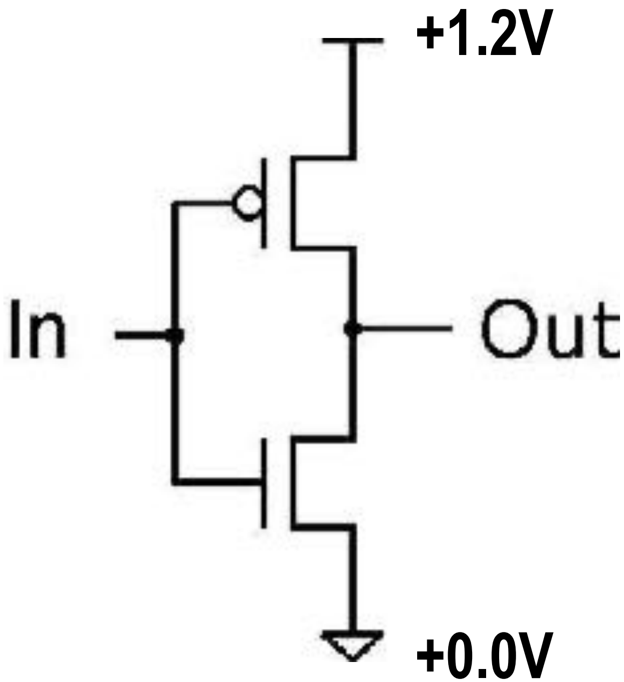


Inverter (NOT Gate)

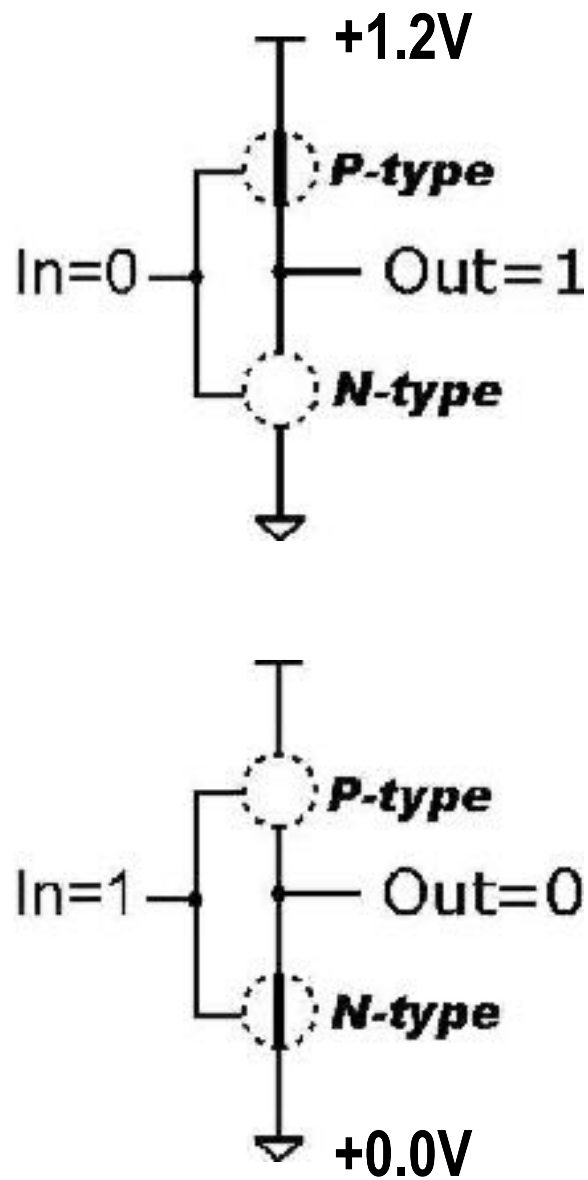




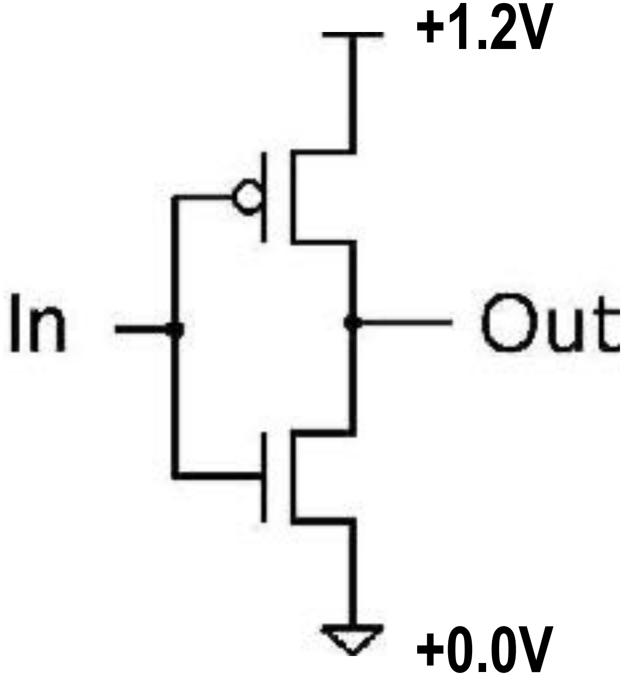
Inverter (NOT Gate)

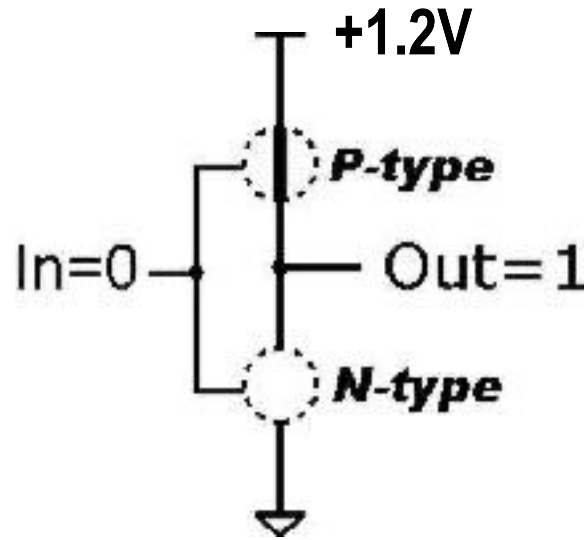


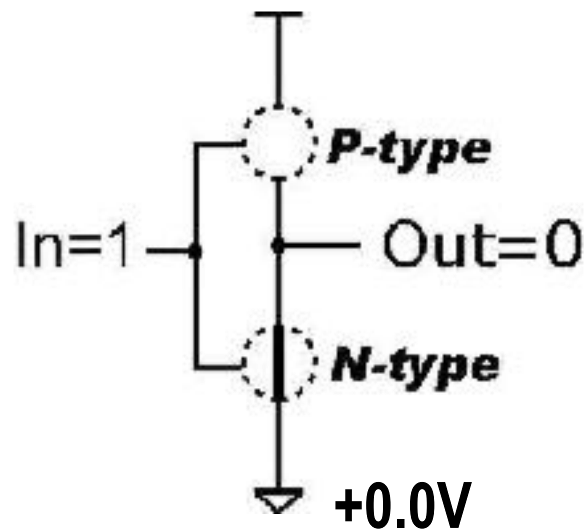
In	Out
0	1
1	0



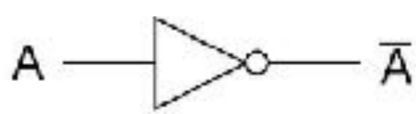
Inverter (NOT Gate)



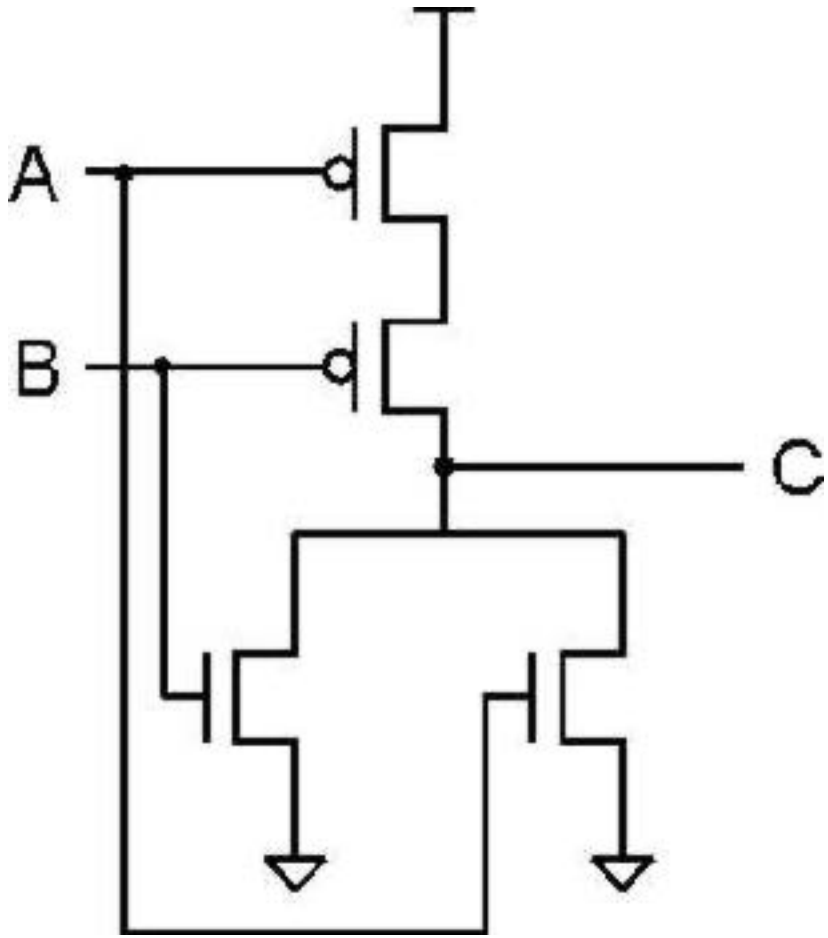


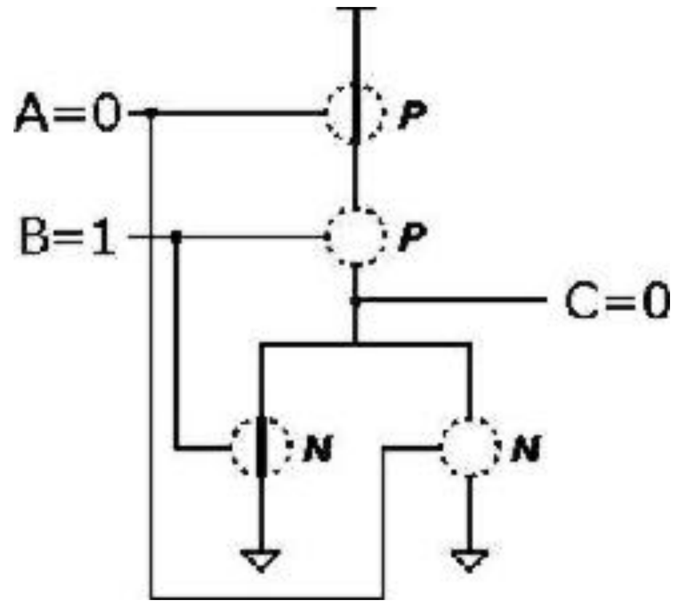


In	Out
0	1
1	0



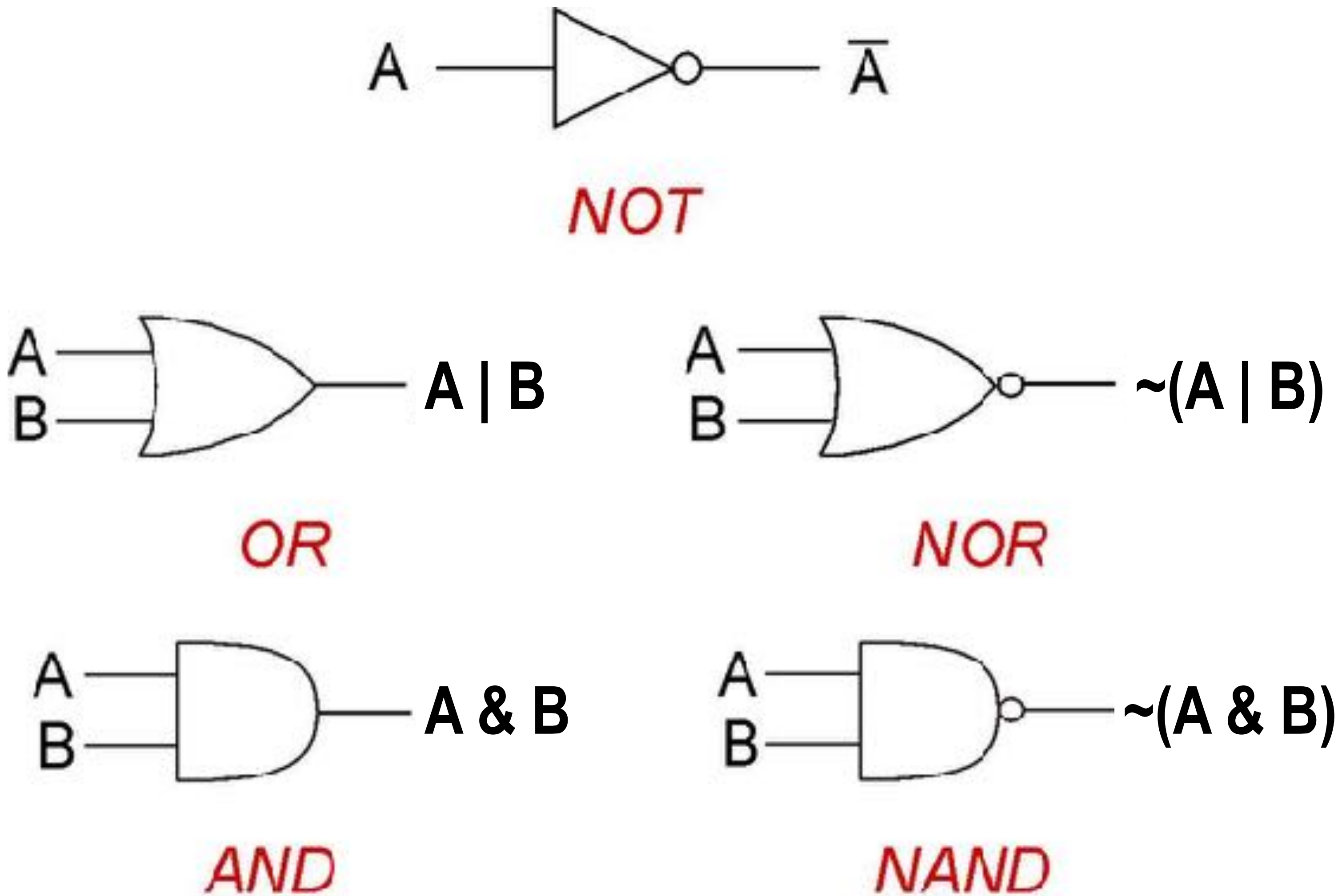
NOR Gate (NOT + OR)



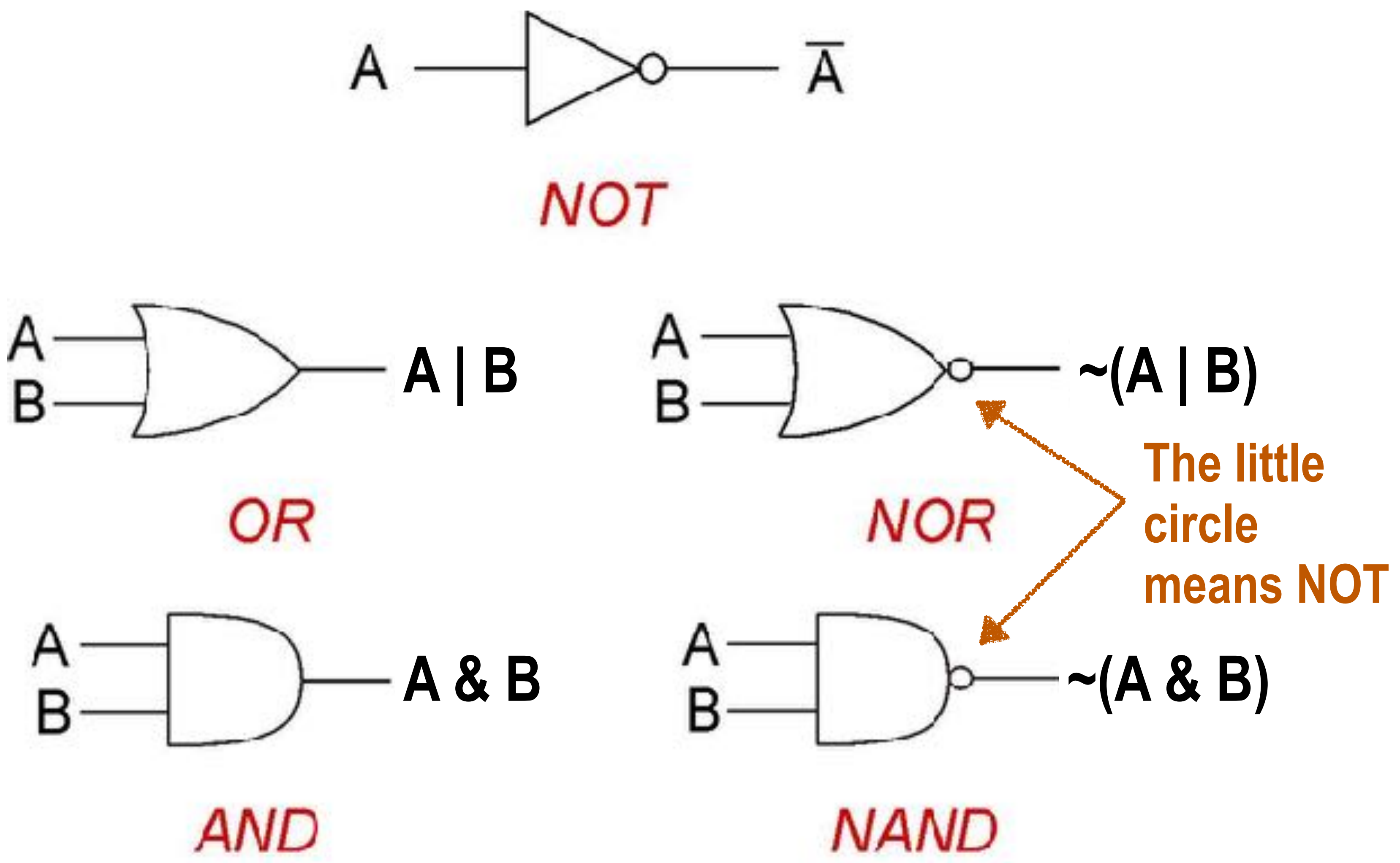


A	B	C
0	0	1
0	1	0
1	0	0
1	1	0

Basic Logic Gates



Basic Logic Gates



Full (1-bit) Adder

Add two bits and carry-in,
produce one-bit sum and carry-out.

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

Full (1-bit) Adder

Truth Table

Add two bits and carry-in,
produce one-bit sum and carry-out.

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

Full (1-bit) Adder

Truth Table

Add two bits and carry-in,
produce one-bit sum and carry-out.

$$S = (\sim A \mathbin{\&} \sim B \mathbin{\&} C_{in})$$

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

Full (1-bit) Adder

Truth Table

Add two bits and carry-in,
produce one-bit sum and carry-out.

$$S = (\sim A \,\&\, \sim B \,\&\, C_{in})$$
$$\mid (\sim A \,\&\, B \,\&\, \sim C_{in})$$

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

Full (1-bit) Adder

Truth Table

Add two bits and carry-in,
produce one-bit sum and carry-out.

$$S = (\sim A \ \& \ \sim B \ \& \ C_{in})$$
$$\mid (\sim A \ \& \ B \ \& \ \sim C_{in})$$
$$\mid (A \ \& \ \sim B \ \& \ \sim C_{in})$$

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

Full (1-bit) Adder

Truth Table

Add two bits and carry-in,
produce one-bit sum and carry-out.

$$S = (\sim A \ \& \ \sim B \ \& \ C_{in})$$
$$| \ (\sim A \ \& \ B \ \& \ \sim C_{in})$$
$$| \ (A \ \& \ \sim B \ \& \ \sim C_{in})$$
$$| \ (A \ \& \ B \ \& \ C_{in})$$

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

Full (1-bit) Adder

Add two bits and carry-in,
produce one-bit sum and carry-out.

$$
\begin{aligned}
S = {} & (\sim A \,\&\, \sim B \,\&\, C_{in}) \\
& \mid (\sim A \,\&\, B \,\&\, \sim C_{in}) \\
& \mid (A \,\&\, \sim B \,\&\, \sim C_{in}) \\
& \mid (A \,\&\, B \,\&\, C_{in})
\end{aligned}
$$

$$
\begin{aligned}
C_{ou} = {} & (\sim A \,\&\, B \,\&\, C_{in}) \\
& \mid (A \,\&\, \sim B \,\&\, C_{in}) \\
& \mid (A \,\&\, B \,\&\, \sim C_{in}) \\
& \mid (A \,\&\, B \,\&\, C_{in})
\end{aligned}
$$

Truth Table

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

Full (1-bit) Adder

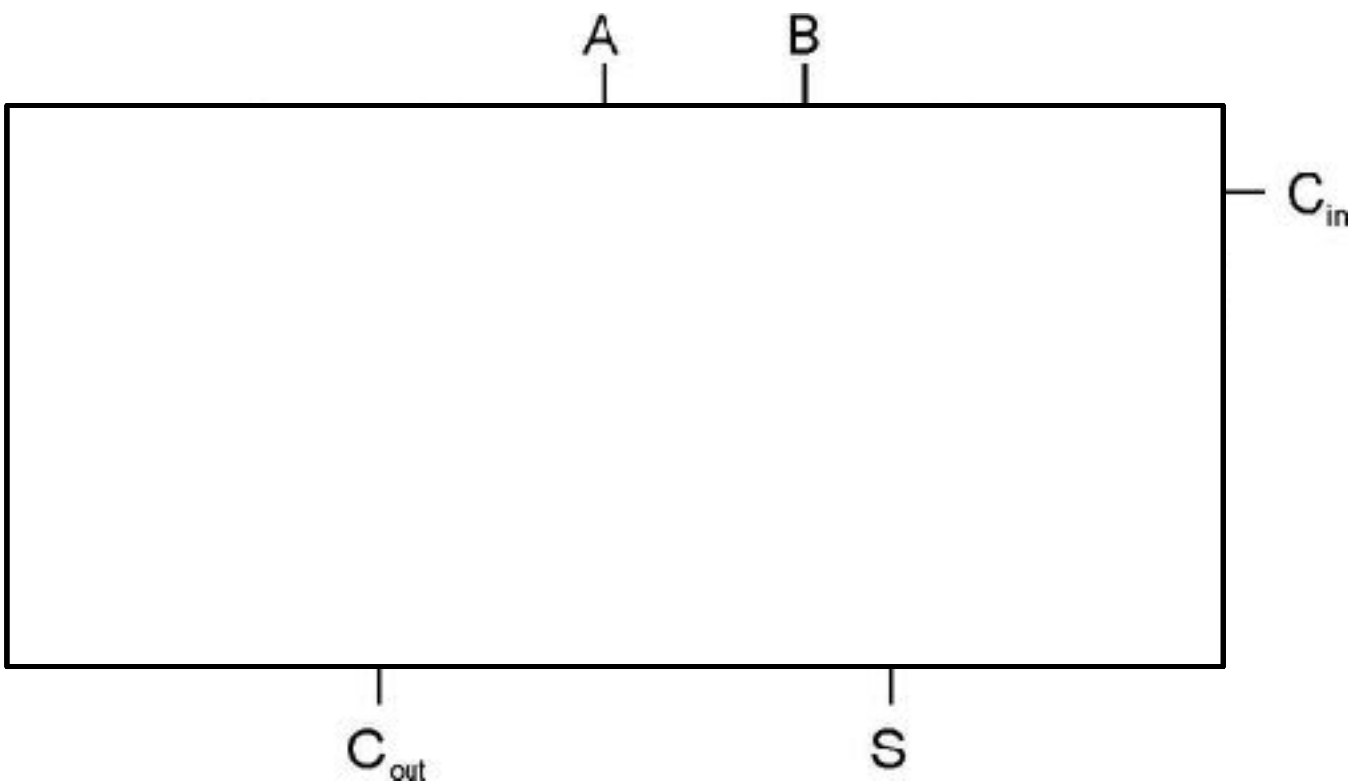
Add two bits and carry-in,
produce one-bit sum and carry-out.

$$C_{ou} = (\sim A \,\&\, B \,\&\, C_{in})$$
$$|\ (A \,\&\, \sim B \,\&\, C_{in})$$
$$|\ (A \,\&\, B \,\&\, \sim C_{in})$$
$$|\ (A \,\&\, B \,\&\, C_{in})$$

Full (1-bit) Adder

Add two bits and carry-in,
produce one-bit sum and carry-out.

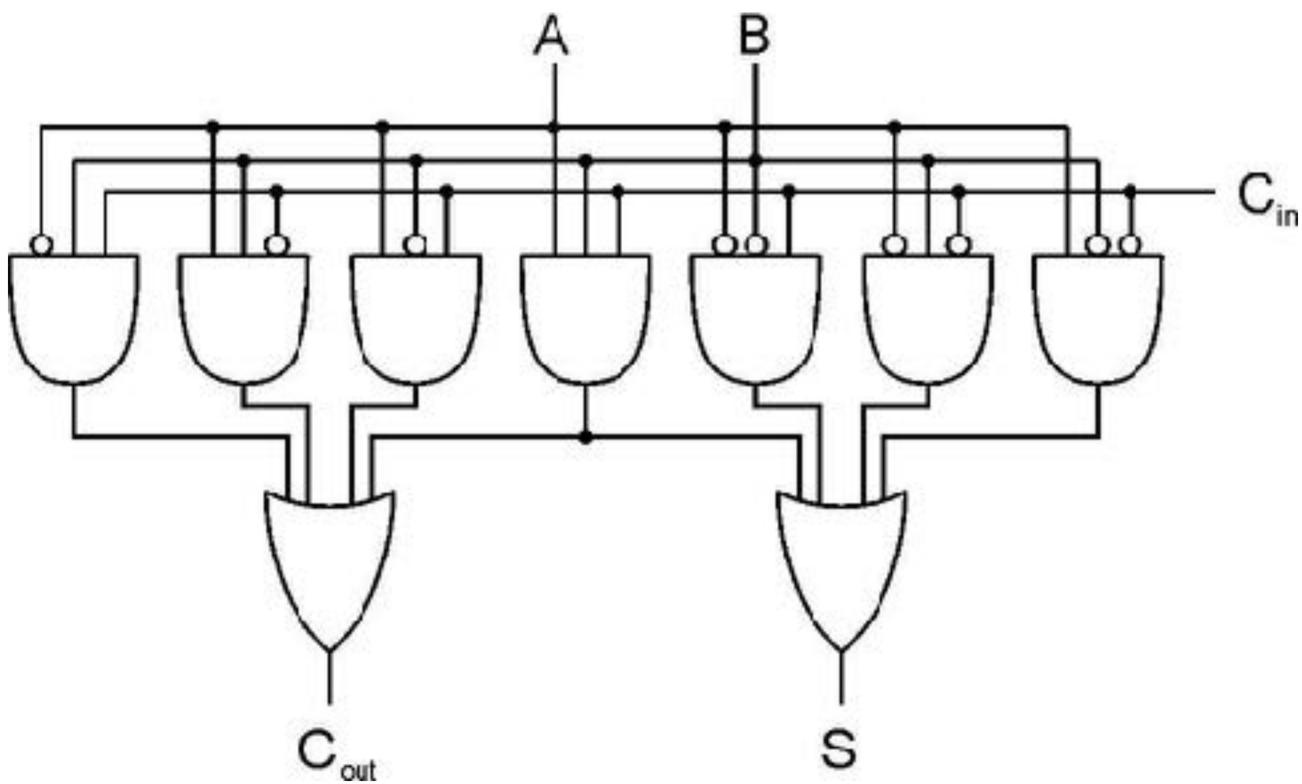
C_{ou} = (~A & B & C_{in})
| (A & ~B & C_{in})
| (A & B & ~C_{in})
| (A & B & C_{in})



Full (1-bit) Adder

$$C_{ou} = (\sim A \ \& \ B \ \& \ C_{in})$$
$$| \ (A \ \& \ \sim B \ \& \ C_{in})$$
$$| \ (A \ \& \ B \ \& \ \sim C_{in})$$
$$| \ (A \ \& \ B \ \& \ C_{in})$$

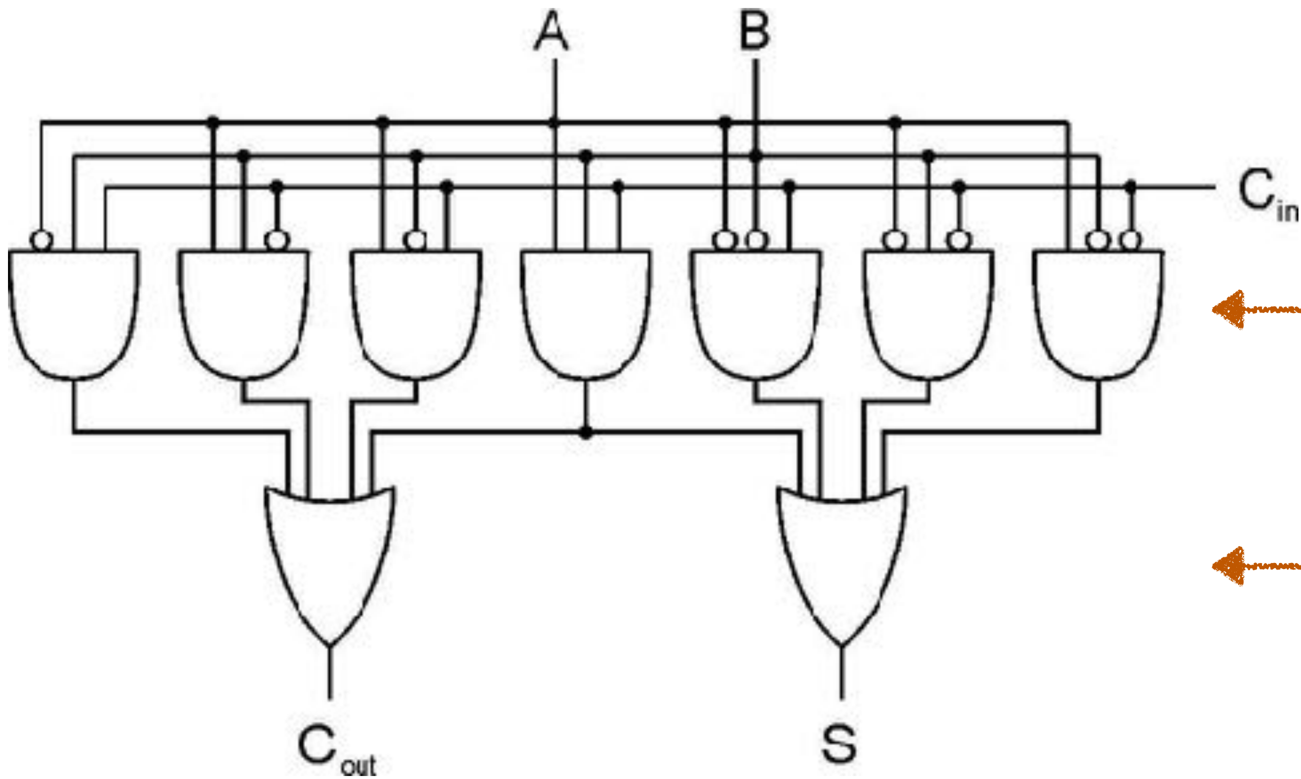
Add two bits and carry-in,
produce one-bit sum and carry-out.



Full (1-bit) Adder

$$C_{ou} = (\sim A \ \& \ B \ \& \ C_{in}) \\ \mid (A \ \& \ \sim B \ \& \ C_{in}) \\ \mid (A \ \& \ B \ \& \ \sim C_{in}) \\ \mid (A \ \& \ B \ \& \ C_{in})$$

Add two bits and carry-in,
produce one-bit sum and carry-out.



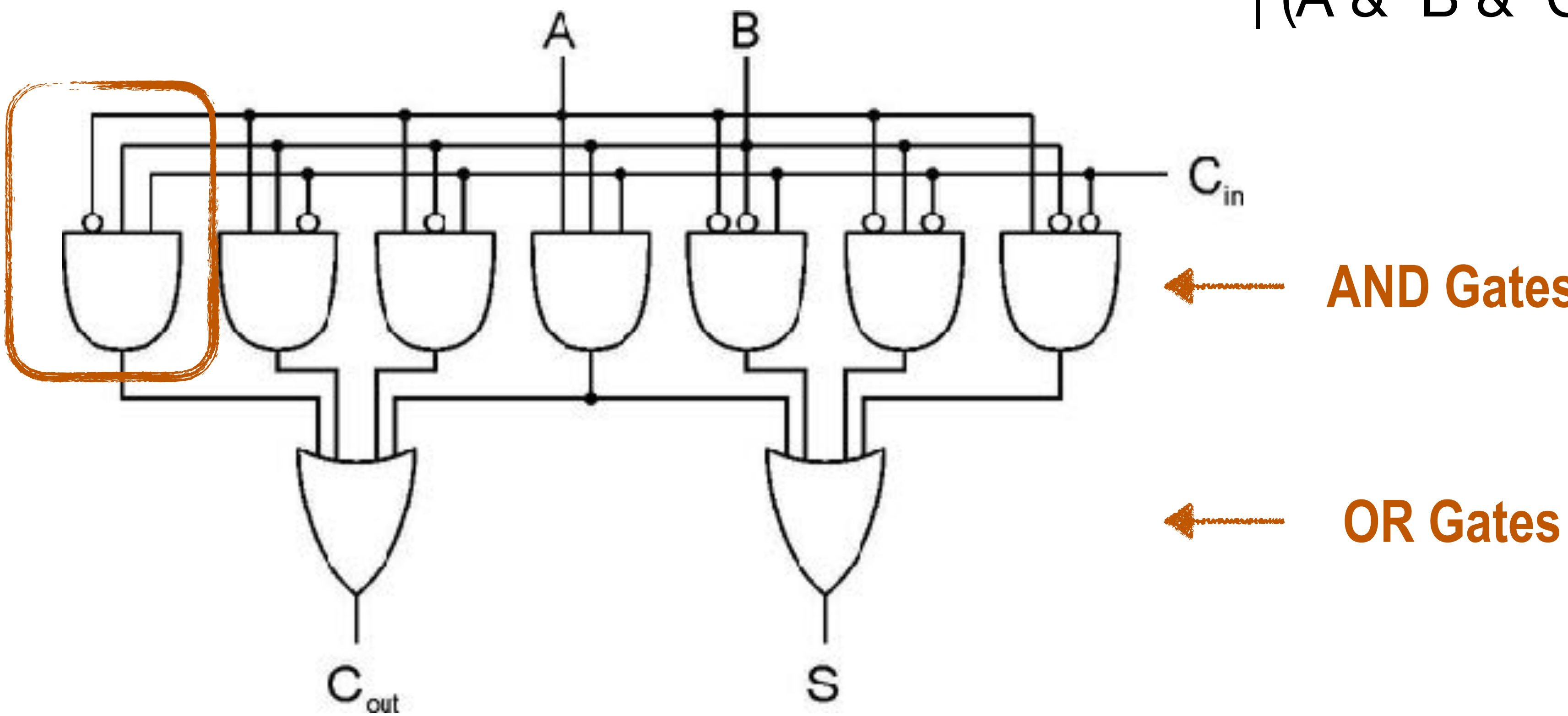
AND Gates

OR Gates

Full (1-bit) Adder

$$
\begin{aligned}
C_{ou} = {} & (\sim A \,\&\, B \,\&\, C_{in}) \\
| \ & (A \,\&\, \sim B \,\&\, C_{in}) \\
| \ & (A \,\&\, B \,\&\, \sim C_{in}) \\
| \ & (A \,\&\, B \,\&\, C_{in})
\end{aligned}
$$

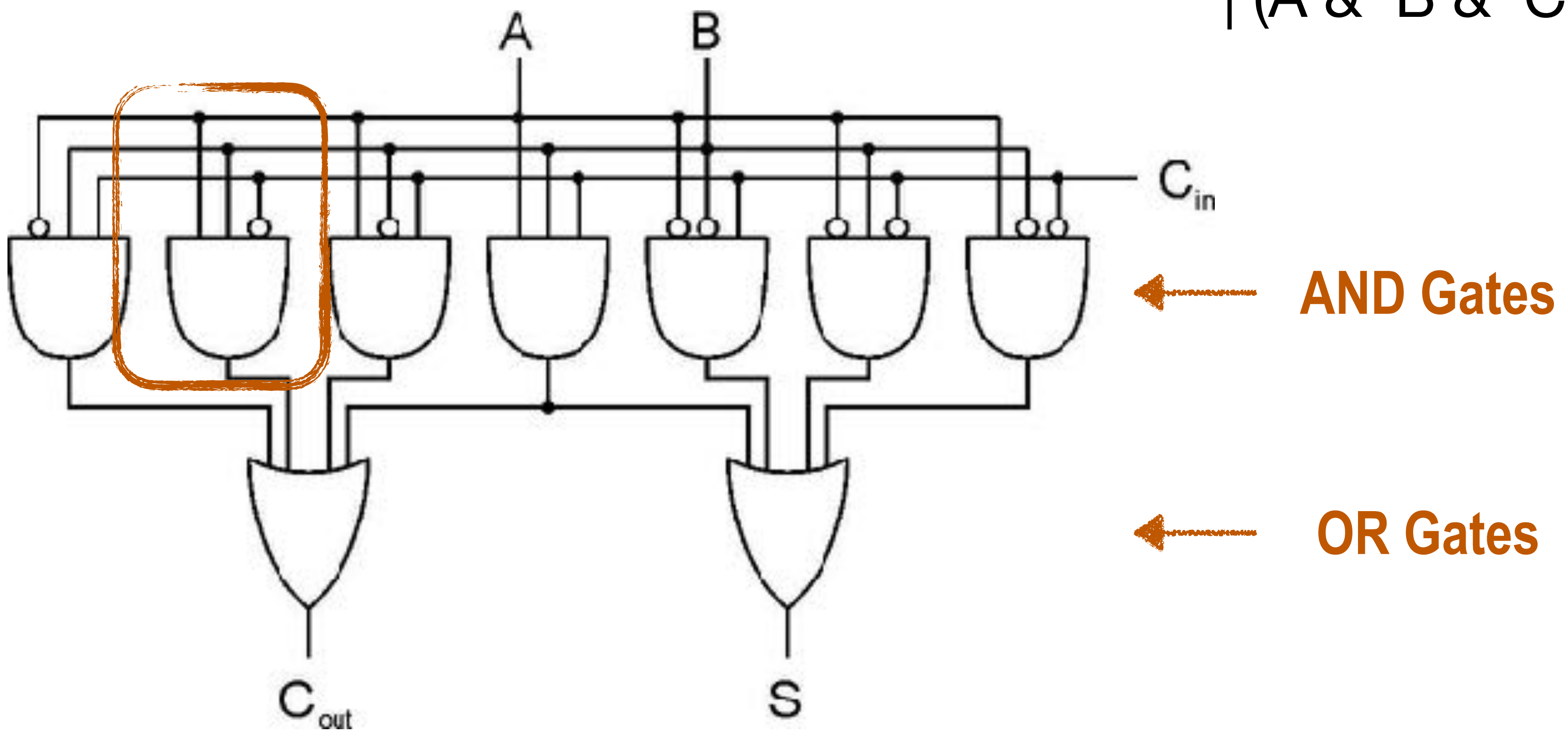
Add two bits and carry-in,
produce one-bit sum and carry-out.



Full (1-bit) Adder

$$C_{ou} = (\sim A \,\&\, B \,\&\, C_{in})$$
$$|\ (A \,\&\, \sim B \,\&\, C_{in})$$
$$|\ (A \,\&\, B \,\&\, \sim C_{in})$$
$$|\ (A \,\&\, B \,\&\, C_{in})$$

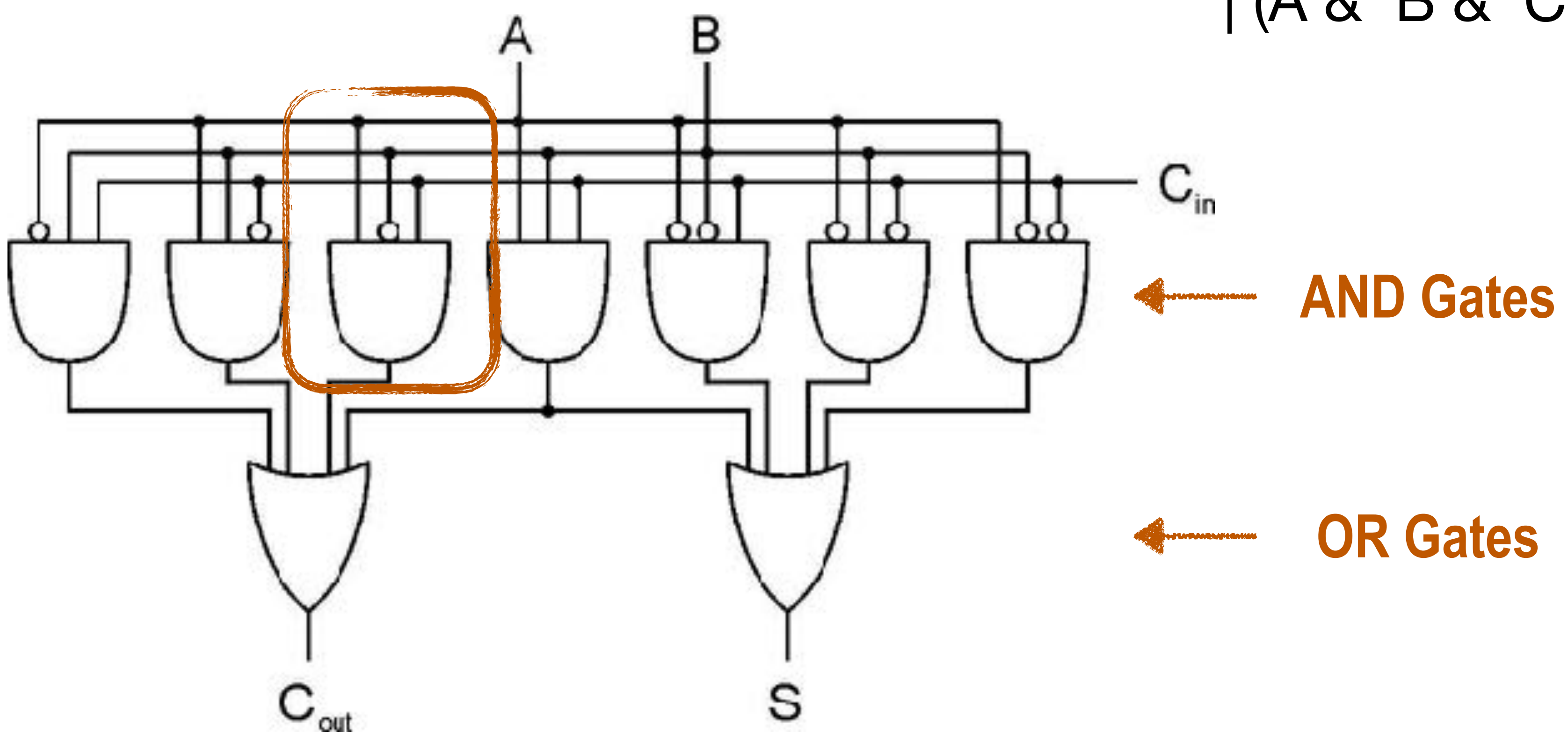
Add two bits and carry-in,
produce one-bit sum and carry-out.



Full (1-bit) Adder

$$
\begin{aligned}
C_{ou} = \; & (\sim A \,\&\, B \,\&\, C_{in}) \\
\mid \; & \boxed{(A \,\&\, \sim B \,\&\, C_{in})} \\
\mid \; & (A \,\&\, B \,\&\, \sim C_{in}) \\
\mid \; & (A \,\&\, B \,\&\, C_{in})
\end{aligned}
$$

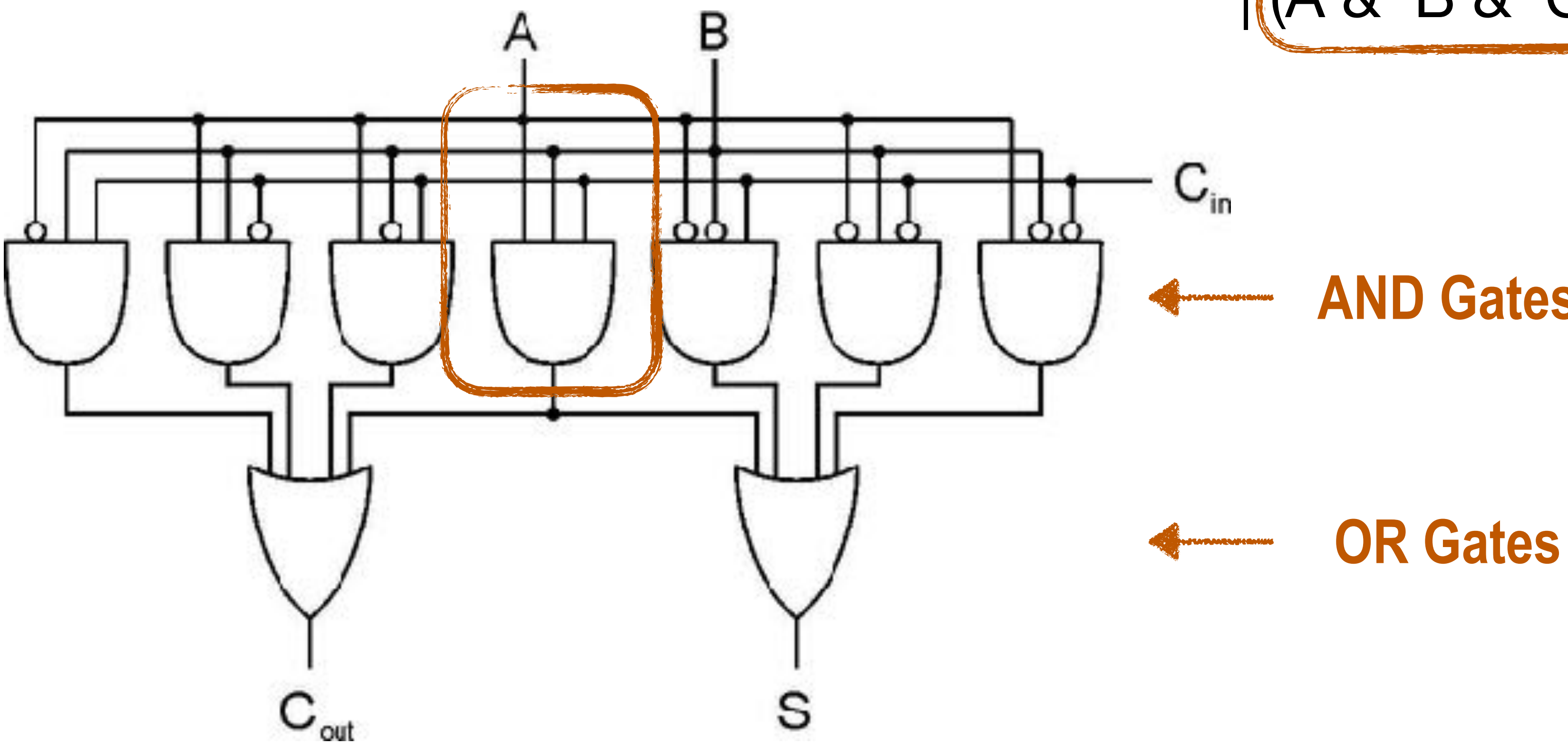
Add two bits and carry-in,
produce one-bit sum and carry-out.



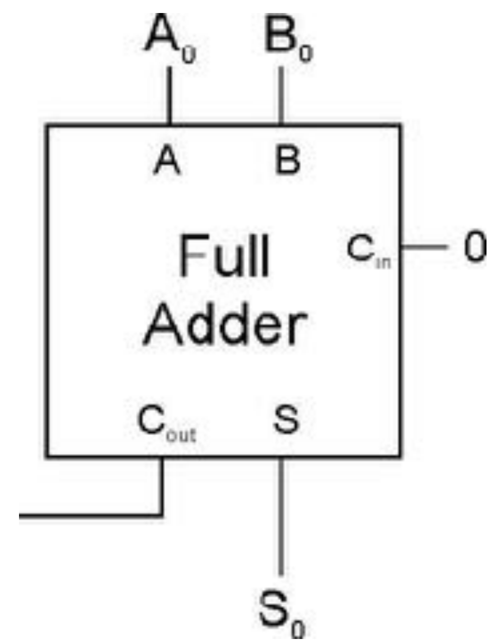
Full (1-bit) Adder

$$
\begin{aligned}
C_{ou} = & (\sim A \ \& \ B \ \& \ C_{in}) \\
& \mid (A \ \& \ \sim B \ \& \ C_{in}) \\
& \mid (A \ \& \ B \ \& \ \sim C_{in}) \\
& \mid \boxed{(A \ \& \ B \ \& \ C_{in})}
\end{aligned}
$$

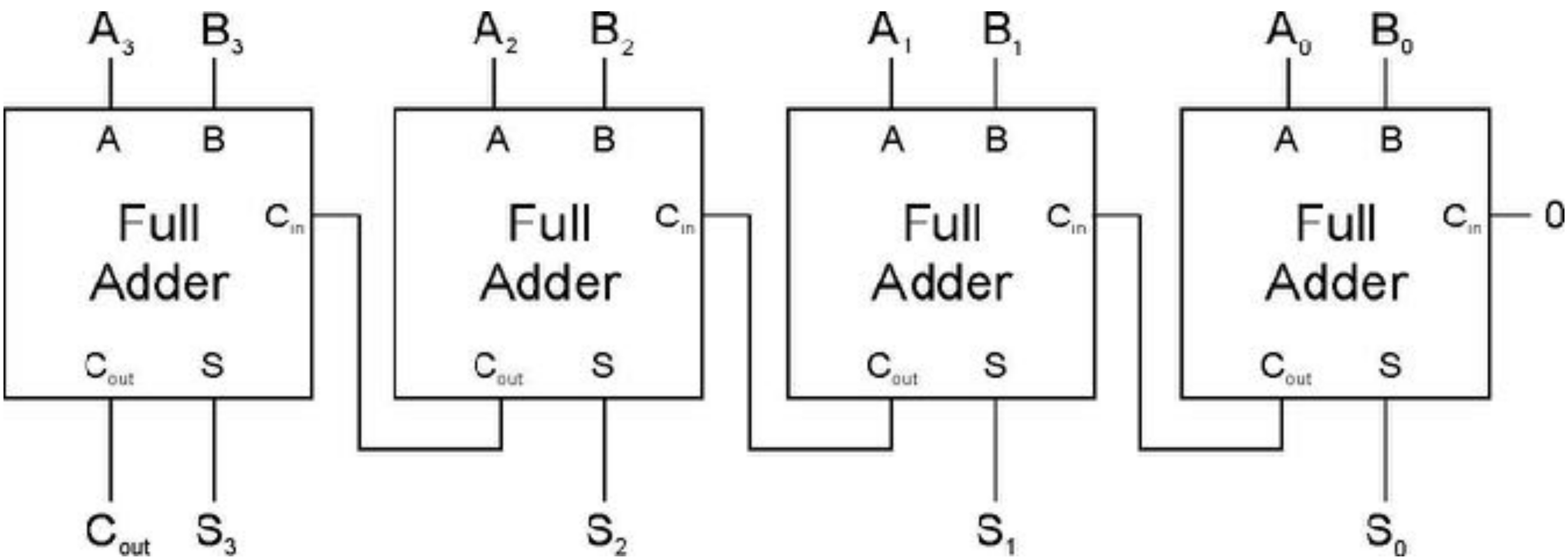
Add two bits and carry-in,
produce one-bit sum and carry-out.



Four-bit Adder

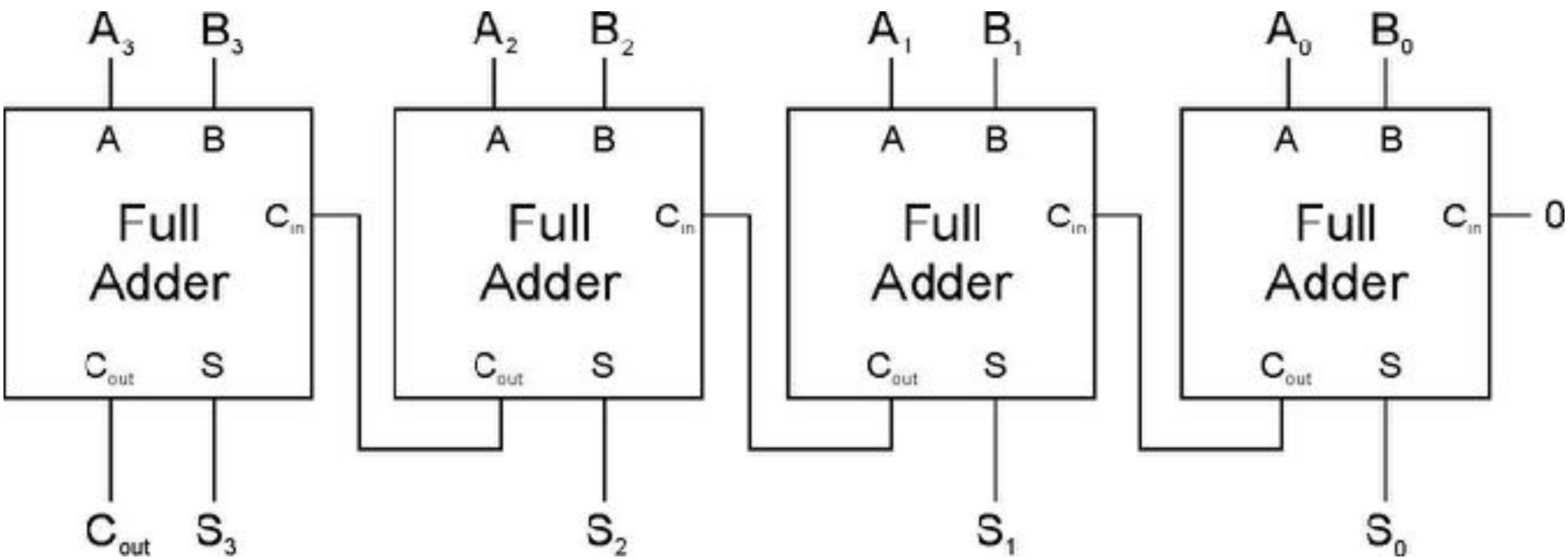


Four-bit Adder



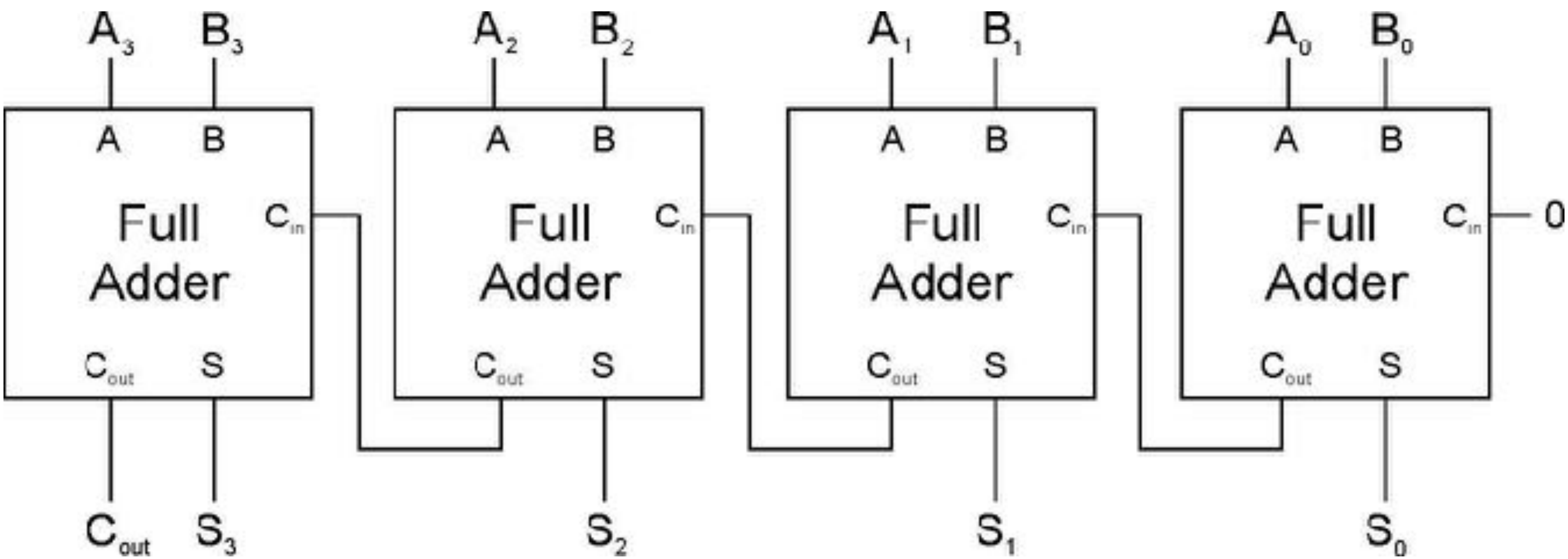
Four-bit Adder

- Ripple-carry Adder
  - Simple, but performance linear to bit width



Four-bit Adder

- Ripple-carry Adder
  - Simple, but performance linear to bit width
- Carry look-ahead adder (CLA)
  - Generate all carriers simultaneously



Logic Design

- Design digital components from basic logic gates

Logic Design

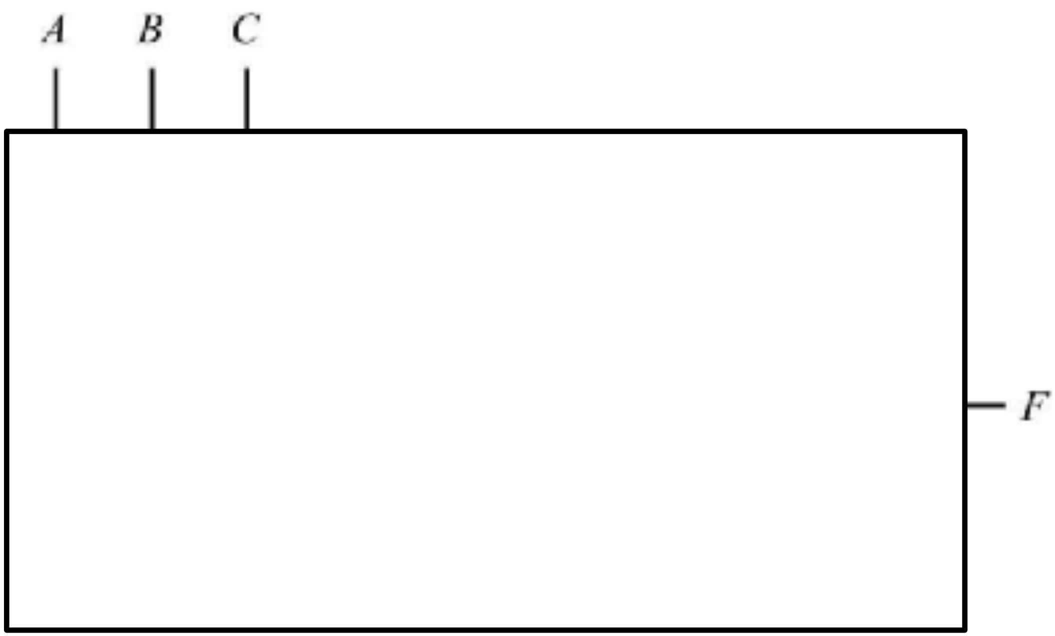
- Design digital components from basic logic gates
- Key idea: use the truth table!

Logic Design

- Design digital components from basic logic gates
- Key idea: use the truth table!
- Example: how to design a piece of circuit that does majority vote?

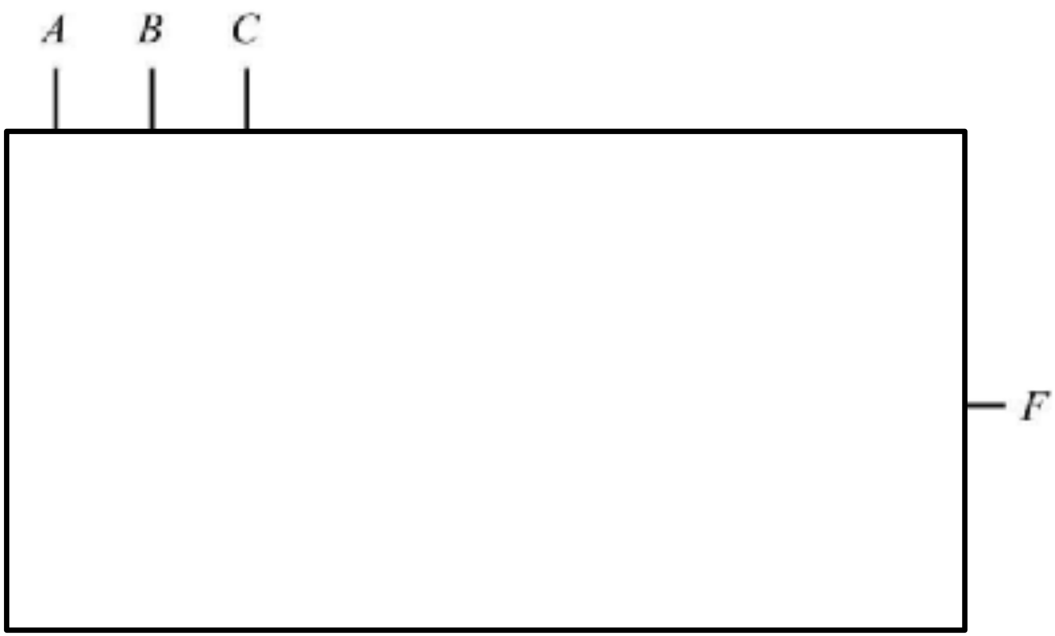
Logic Design

- Design digital components from basic logic gates
- Key idea: use the truth table!
- Example: how to design a piece of circuit that does majority vote?



Logic Design

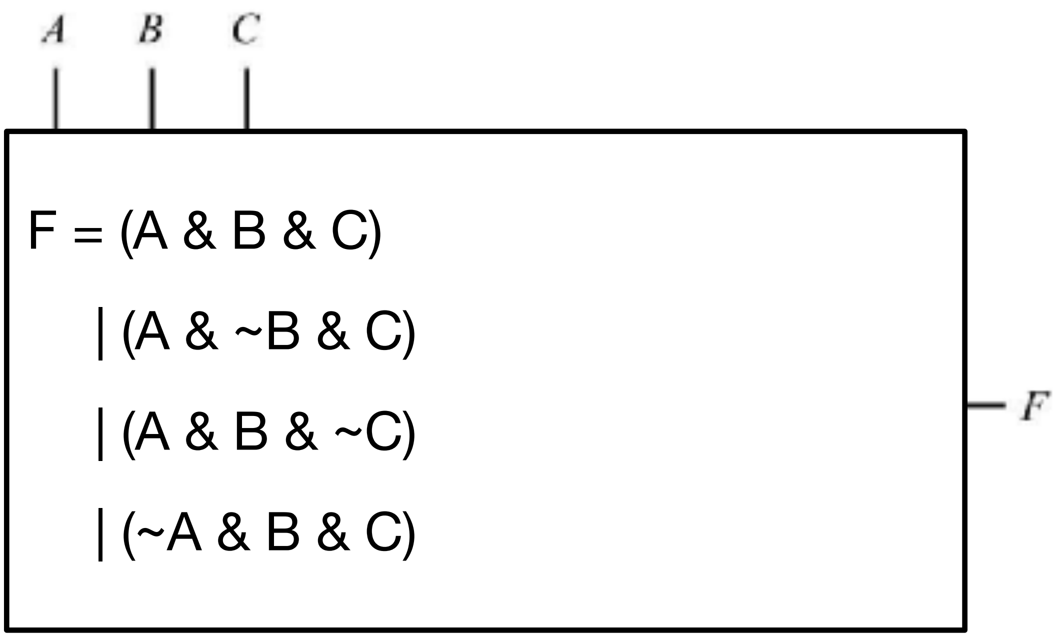
- Design digital components from basic logic gates
- Key idea: use the truth table!
- Example: how to design a piece of circuit that does majority vote?



A	B	C	F
0	0	0	**0**
0	0	1	**0**
0	1	0	**0**
0	1	1	**1**
1	0	0	**0**
1	0	1	**1**
1	1	0	**1**
1	1	1	**1**

Logic Design

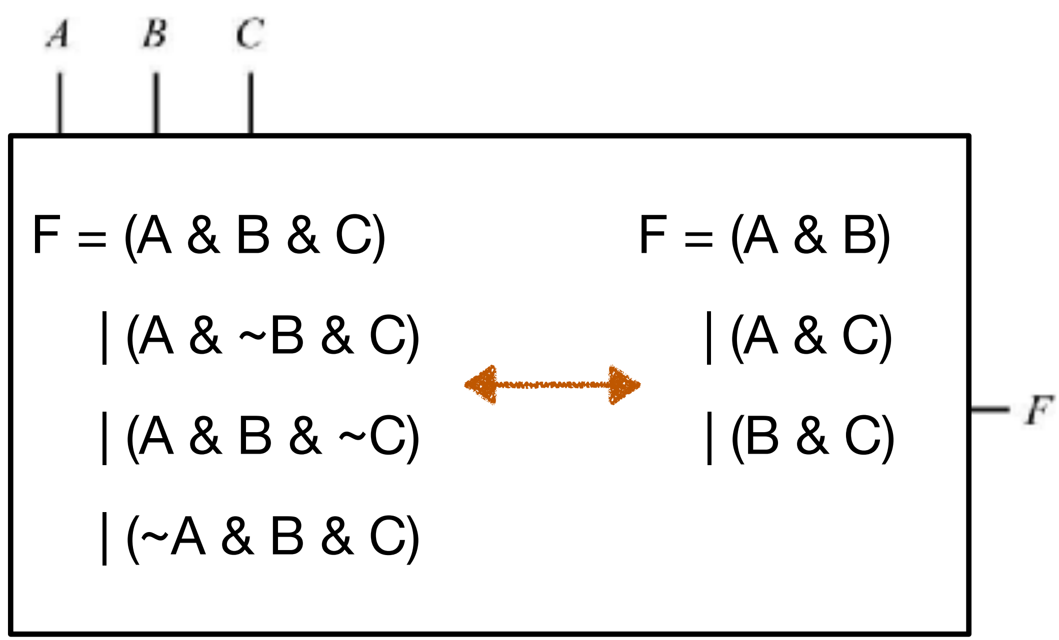
- Design digital components from basic logic gates
- Key idea: use the truth table!
- Example: how to design a piece of circuit that does majority vote?



A	B	C	F
0	0	0	**0**
0	0	1	**0**
0	1	0	**0**
0	1	1	**1**
1	0	0	**0**
1	0	1	**1**
1	1	0	**1**
1	1	1	**1**

Logic Design

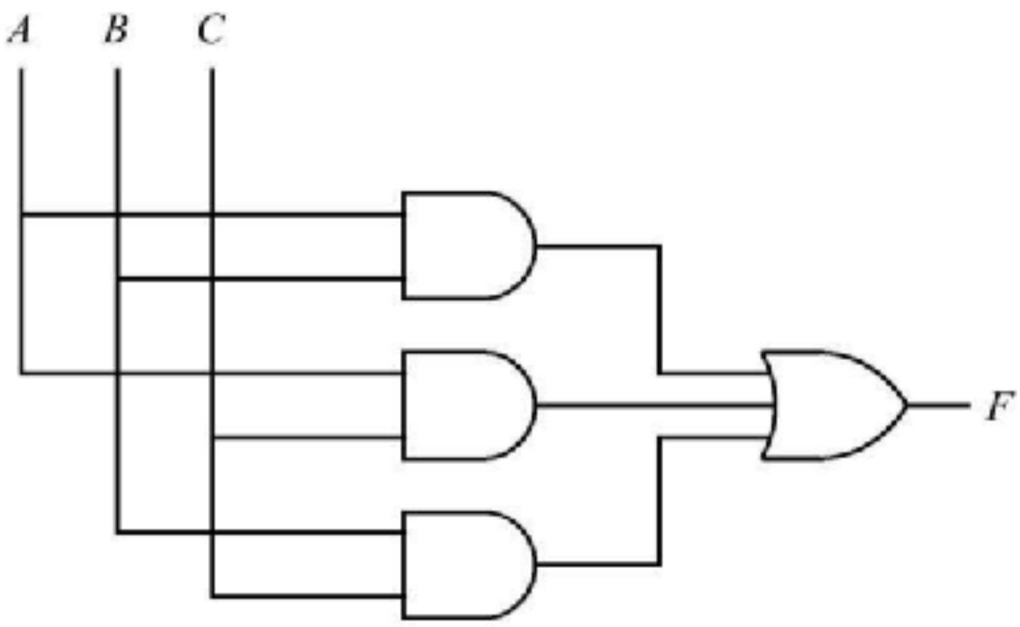
- Design digital components from basic logic gates
- Key idea: use the truth table!
- Example: how to design a piece of circuit that does majority vote?



A	B	C	F
0	0	0	**0**
0	0	1	**0**
0	1	0	**0**
0	1	1	**1**
1	0	0	**0**
1	0	1	**1**
1	1	0	**1**
1	1	1	**1**

Logic Design

- Design digital components from basic logic gates
- Key idea: use the truth table!
- Example: how to design a piece of circuit that does majority vote?



A	B	C	F
0	0	0	**0**
0	0	1	**0**
0	1	0	**0**
0	1	1	**1**
1	0	0	**0**
1	0	1	**1**
1	1	0	**1**
1	1	1	**1**