CSC 252: Computer Organization Spring 2019: Lecture 3

Instructor: Yuhao Zhu

Department of Computer Science
University of Rochester

Action Items:

- **Trivia 1 is due tomorrow, midnight**
- **Main assignment due Feb. 1, midnight**

Announcement

- Programming Assignment 1 is out
    - Details: http://cs.rochester.edu/courses/252/spring2019/labs/assignment1.html
    - Due on Feb 1, 11:59 PM
    - Trivia due Friday, 1/25, 11:59 PM
    - You have 3 slip days (not for trivia)

20	21	22	23	24	25	26
				Today	**Trivia**	
27	28	29	30	31	Feb 1	2
					Due	

Announcement

- TA review sessions schedule is posted

Review Session Schedule

Two 1-hour review sessions are offered each week. Review sessions are not mandatory. During review sessions, TAs might review course materials from the past week, go over problem sets and past exams, provide an overview of programming assignments, etc. They will be interactive and you are encouraged to ask questions. It is up to the TAs to decide how to run it.

Tuesday 6 PM - 7 PM, in WH 2506 (Rotating across Olivia, Jessica, Max, Sam, Yawo)

Thursday 7:30 PM - 8:30 PM, in WH 2506 (Rotating across Amir, Minh, Yiyang, Yu)

Announcement

- Check the course website before asking
    - http://www.cs.rochester.edu/courses/252/spring2019/

Announcement

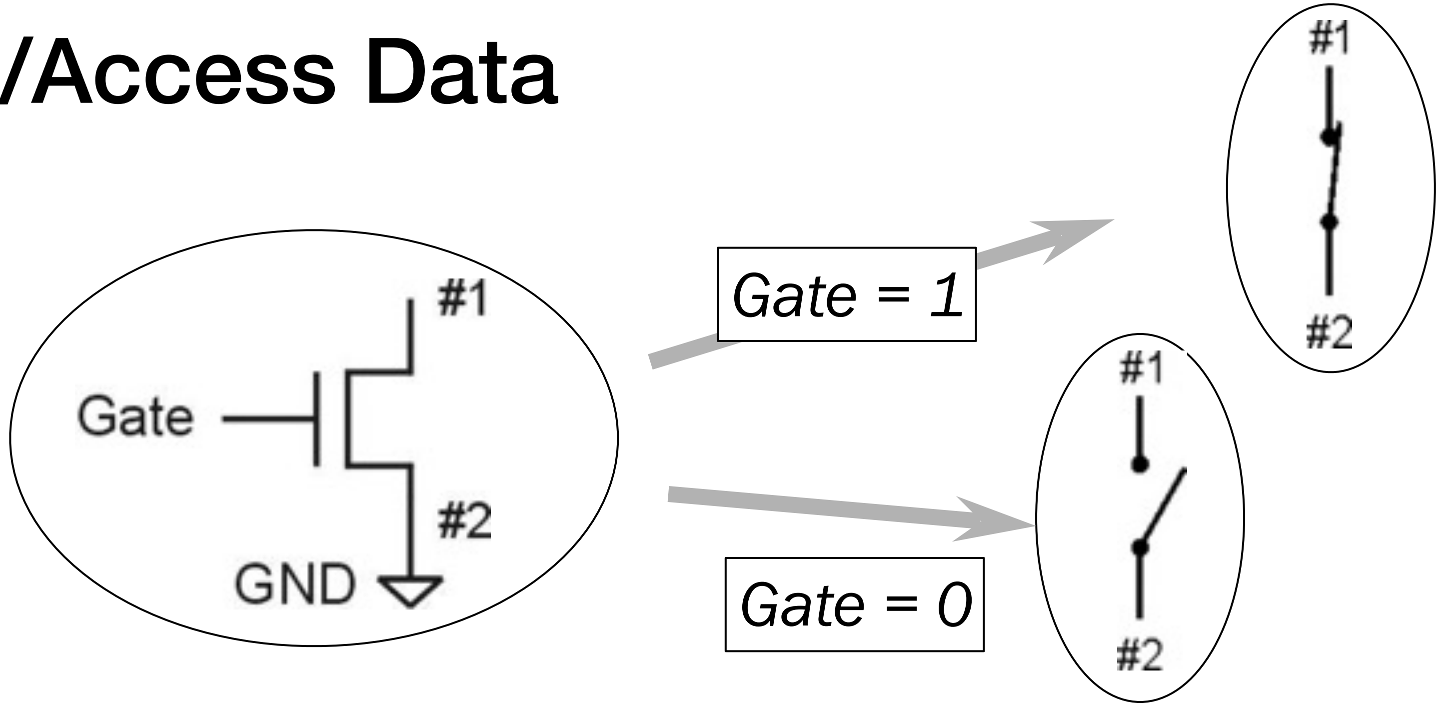
- Check the course website before asking
  - http://www.cs.rochester.edu/courses/252/spring2019/
- Direct ALL questions regarding assignments to the TAs
  - They have done them. They have debugged them. They know them inside out.
  - If one doesn’t know, ask another.
  - If all don’t know, ask me.

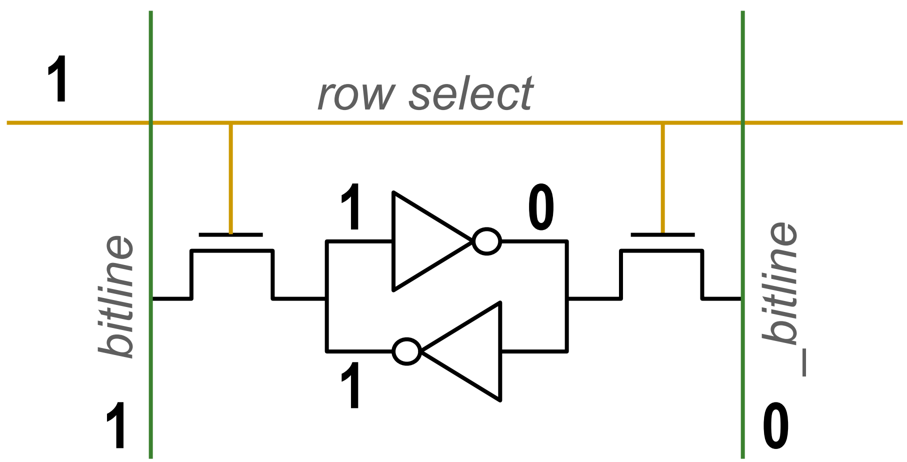
Previously in 252…

- Computers are built to understand bits: 0 and 1
  - 0: low (no) voltage; 1: high voltage
- Integer representations (Fixed-point really)

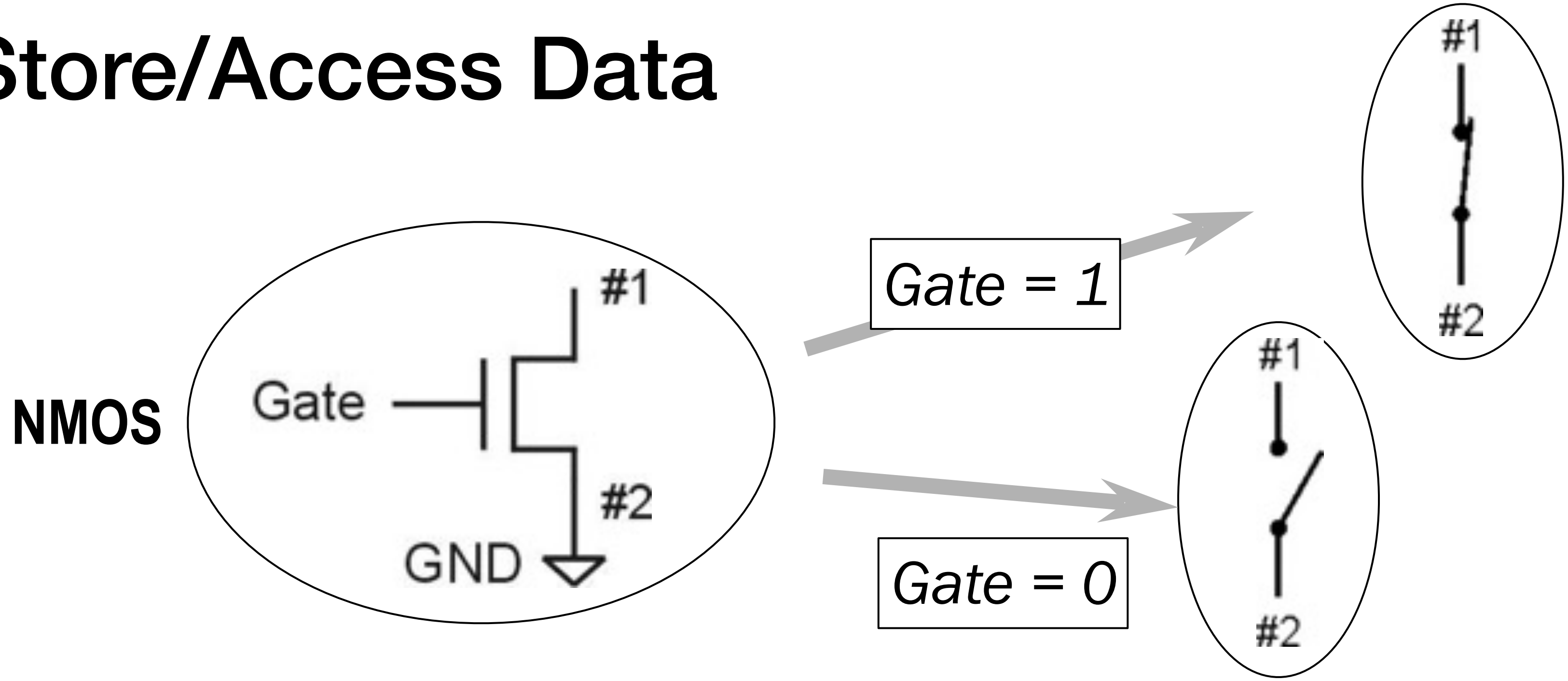
Store/Access Data

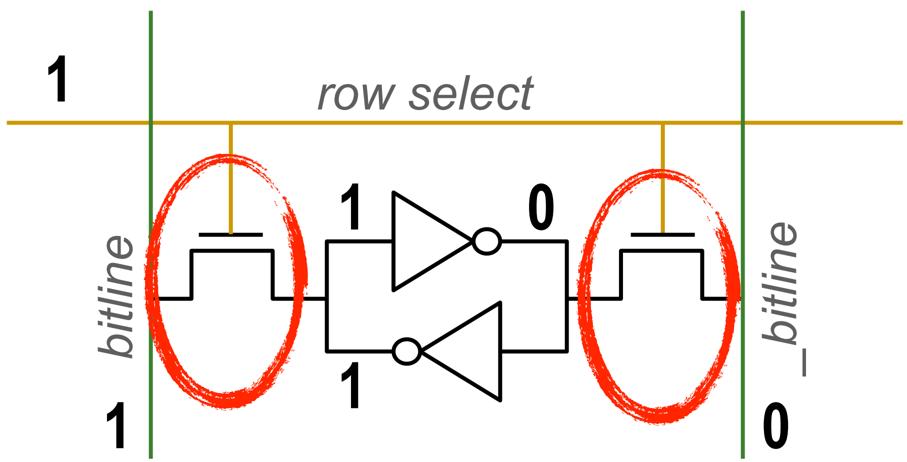
NMOS





Store/Access Data



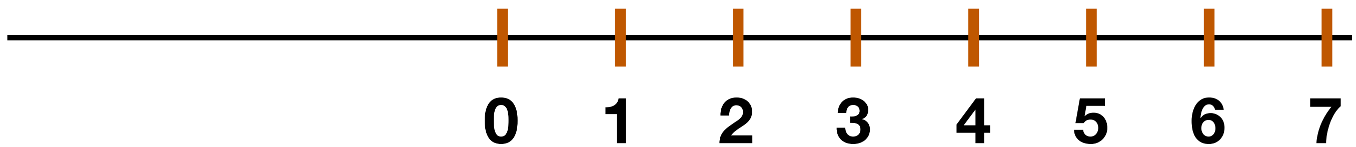


Encoding Negative Numbers

- Two’s Complement

Encoding Negative Numbers

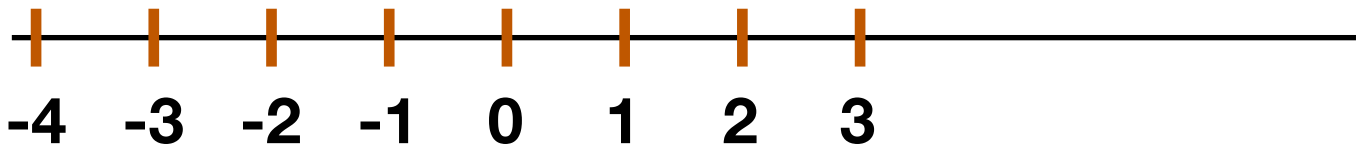
- Two’s Complement



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

Encoding Negative Numbers

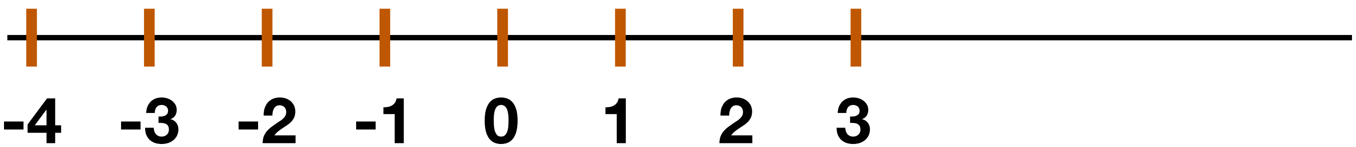
- Two's Complement



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

Encoding Negative Numbers

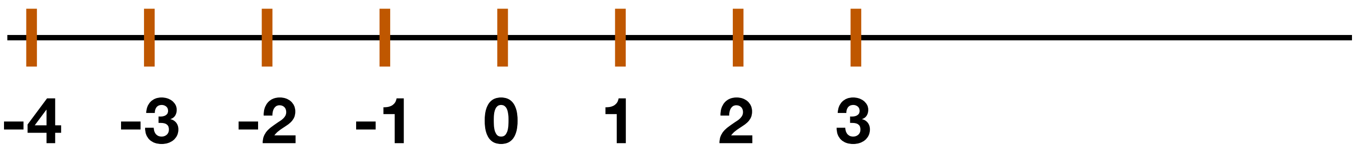
- Two's Complement



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

- Two's Complement



$b_2 b_1 b_0$

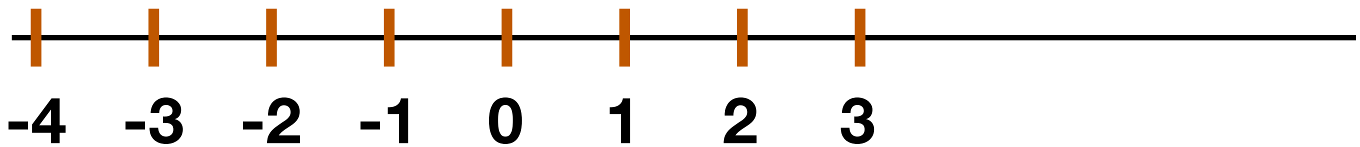
Weights in Unsigned

2^2 2^1 2^0

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

- Two’s Complement



$b_2 b_1 b_0$

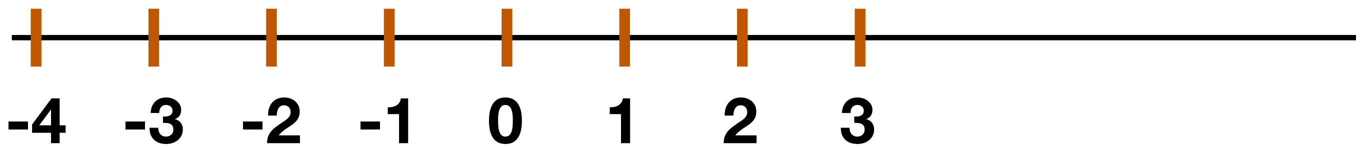
Weights in Unsigned 2^2 2^1 2^0

Weights in Signed -2^2 2^1 2^0

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

- Two's Complement



$b_2 b_1 b_0$

Weights in Unsigned $\quad 2^2 \quad 2^1 \quad 2^0$

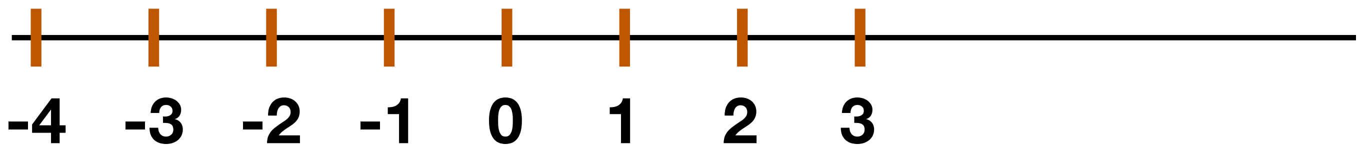
Weights in Signed $\quad -2^2 \quad 2^1 \quad 2^0$

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

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

Encoding Negative Numbers

- Two's Complement



$b_2 b_1 b_0$

Weights in Unsigned: 2^2 2^1 2^0

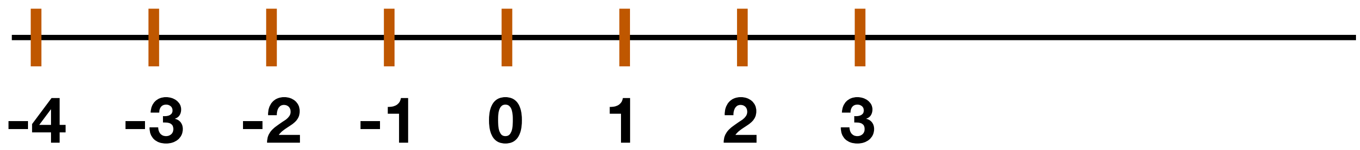
Weights in Signed: -2^2 2^1 2^0

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

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

Encoding Negative Numbers

- Two's Complement



$b_2 b_1 b_0$

Weights in Unsigned: 2^2 2^1 2^0

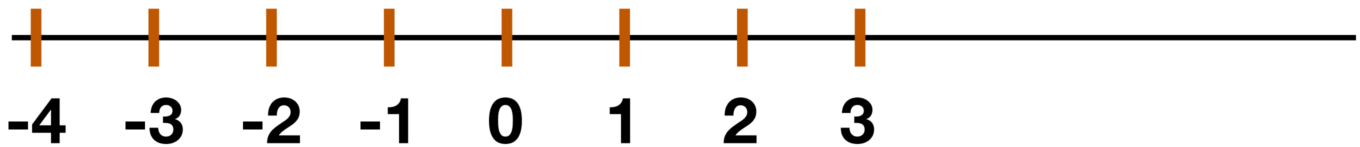
Weights in Signed: -2^2 2^1 2^0

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

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

Encoding Negative Numbers

- Two's Complement



$b_2 b_1 b_0$

Weights in Unsigned 2^2 2^1 2^0

Weights in Signed -2^2 2^1 2^0

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

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 Implications

- Only 1 zero
- There is (still) a bit that represents sign!
- Unsigned arithmetic still works

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

Two-Complement Implications

- Only 1 zero
- There is (still) a bit that represents sign!
- Unsigned arithmetic still works

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

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

Two-Complement Implications

- Only 1 zero
- There is (still) a bit that represents sign!
- Unsigned arithmetic still works

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

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

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

Two-Complement Implications

- Only 1 zero
- There is (still) a bit that represents sign!
- Unsigned arithmetic still works

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

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

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

- 3 + 1 becomes -4 (called overflow. More on it later.)

Data Types (in C)

- Suppose you want to define a variable that represents a person's age. What assumptions can you make about this variable's numerical value?

Data Types (in C)

- Suppose you want to define a variable that represents a person's age. What assumptions can you make about this variable's numerical value?
    - Integer

Data Types (in C)

- Suppose you want to define a variable that represents a person's age. What assumptions can you make about this variable's numerical value?
  - Integer
  - Non-negative

Data Types (in C)

- Suppose you want to define a variable that represents a person’s age. What assumptions can you make about this variable’s numerical value?
  - Integer
  - Non-negative
  - Between 0 and 255 (8 bits)

Data Types (in C)

- Suppose you want to define a variable that represents a person's age. What assumptions can you make about this variable's numerical value?
  - Integer
  - Non-negative
  - Between 0 and 255 (8 bits)
- Define a data type that captures all these attributes: `unsigned char` in C

Data Types (in C)

- Suppose you want to define a variable that represents a person's age. What assumptions can you make about this variable's numerical value?
  - Integer
  - Non-negative
  - Between 0 and 255 (8 bits)
- Define a data type that captures all these attributes: `unsigned char` in C
  - Internally, an unsigned int variable is represented as a 8-bit, non-negative, binary number

Data Types (in C)

- What if you want to define a variable that could take negative values?

Data Types (in C)

- What if you want to define a variable that could take negative values?
  - That's what the **`int`** data type is for

Data Types (in C)

- What if you want to define a variable that could take negative values?
  - That's what the **`int`** data type is for
- How are `int` values internally represented?

Data Types (in C)

- What if you want to define a variable that could take negative values?
  - That's what the **`int`** data type is for
- How are `int` values internally represented?
  - Theoretically could be either signed-magnitude or two's complement

Data Types (in C)

- What if you want to define a variable that could take negative values?
  - That's what the **`int`** data type is for
- How are `int` values internally represented?
  - Theoretically could be either signed-magnitude or two's complement
  - The C language designers chose two's complement

Data Types (in C)

- What if you want to define a variable that could take negative values?
  - That's what the **`int`** data type is for
- How are `int` values internally represented?
  - Theoretically could be either signed-magnitude or two's complement
  - The C language designers chose two's complement
  - This is where math meets computer science

Data Types (in C)

C Data Type	32-bit	64-bit
`(unsigned) char`	1	1
`(unsigned) short`	2	2
`(unsigned) int`	4	4
`(unsigned) long`	4	8

Data Types (in C)

	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
`(unsigned) char`	1	1
`(unsigned) short`	2	2
`(unsigned) int`	4	4
`(unsigned) long`	4	8

Data Types (in C)

	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
`(unsigned) char`	1	1
`(unsigned) short`	2	2
`(unsigned) int`	4	4
`(unsigned) long`	4	8

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

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;
```

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
```

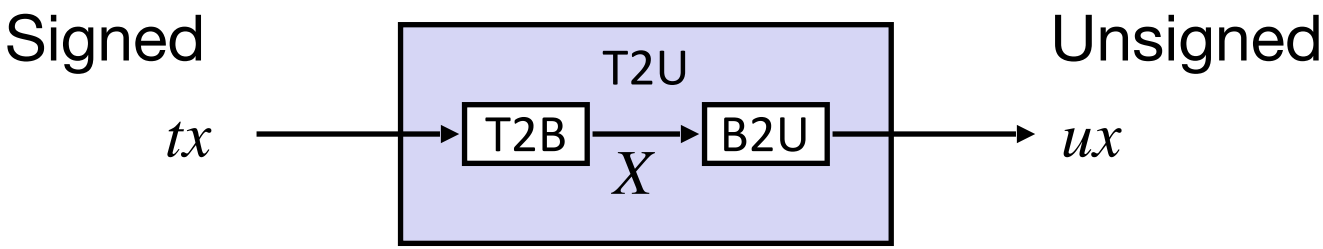
  - Implicit casting
    - e.g., assignments, function calls

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

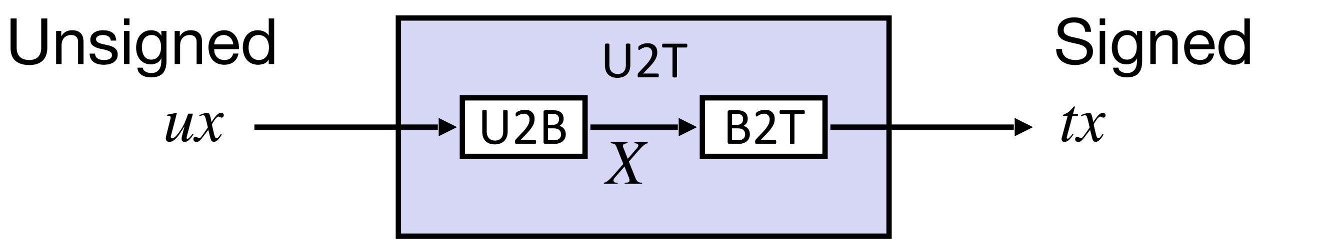
Signed	Unsigned	Binary
0	0	000
1	1	001
2	2	010
3	3	011
-4	4	100
-3	5	101
-2	6	110
-1	7	111

Mapping Between Signed & Unsigned

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



Maintain Same Bit Pattern



Maintain Same Bit Pattern

Mapping Signed ↔ Unsigned

Bits
0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111

Signed
0
1
2
3
4
5
6
7
-8
-7
-6
-5
-4
-3
-2
-1

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

Signed
0
1
2
3
4
5
6
7
-8
-7
-6
-5
-4
-3
-2
-1

=

Unsigned
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
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

Mapping Signed ↔ Unsigned

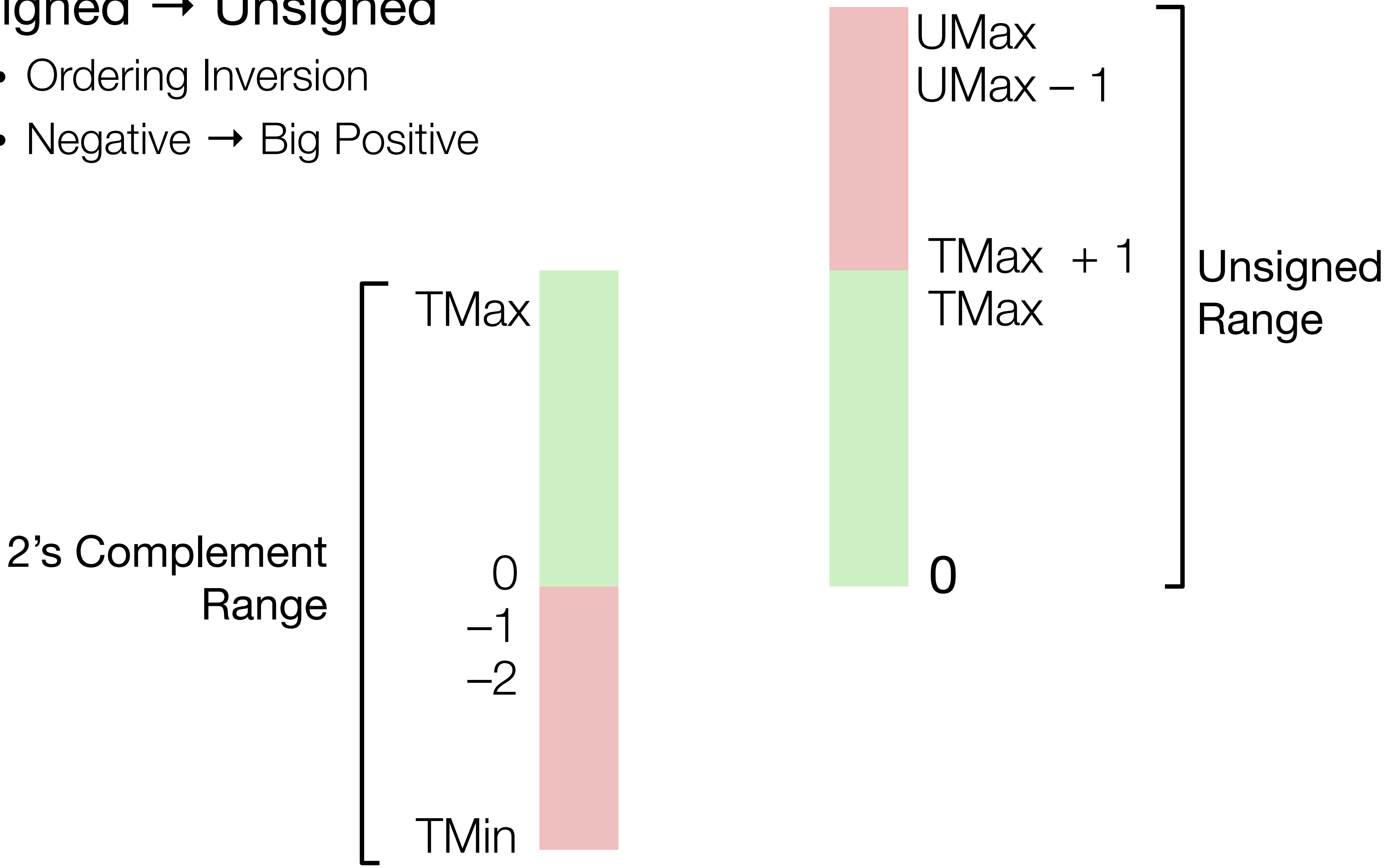
Bits	Signed	Unsigned
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	-8	8
1001	-7	9
1010	-6	10
1011	-5	11
1100	-4	12
1101	-3	13
1110	-2	14
1111	-1	15

=

+/- 16

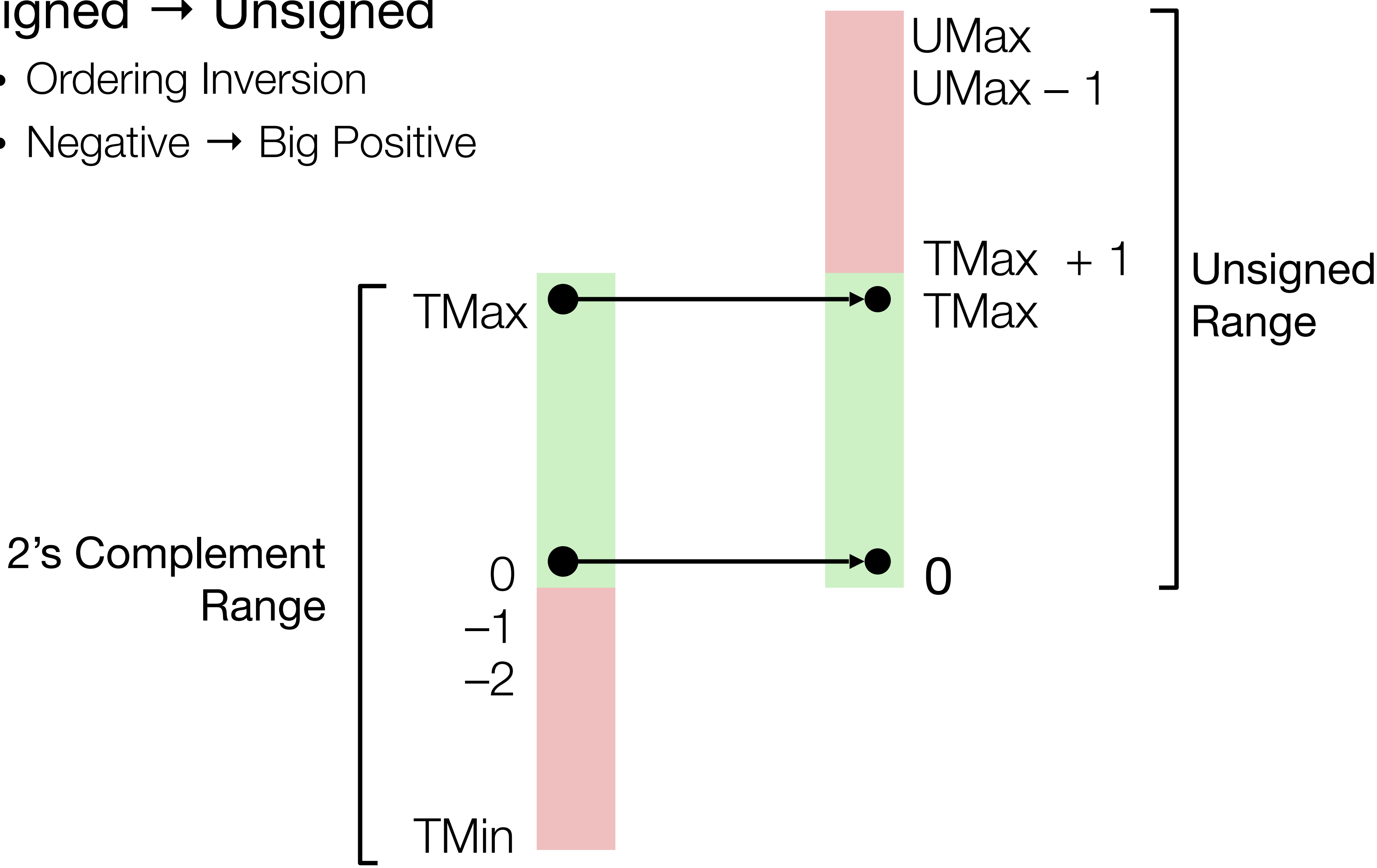
Conversion Visualized

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



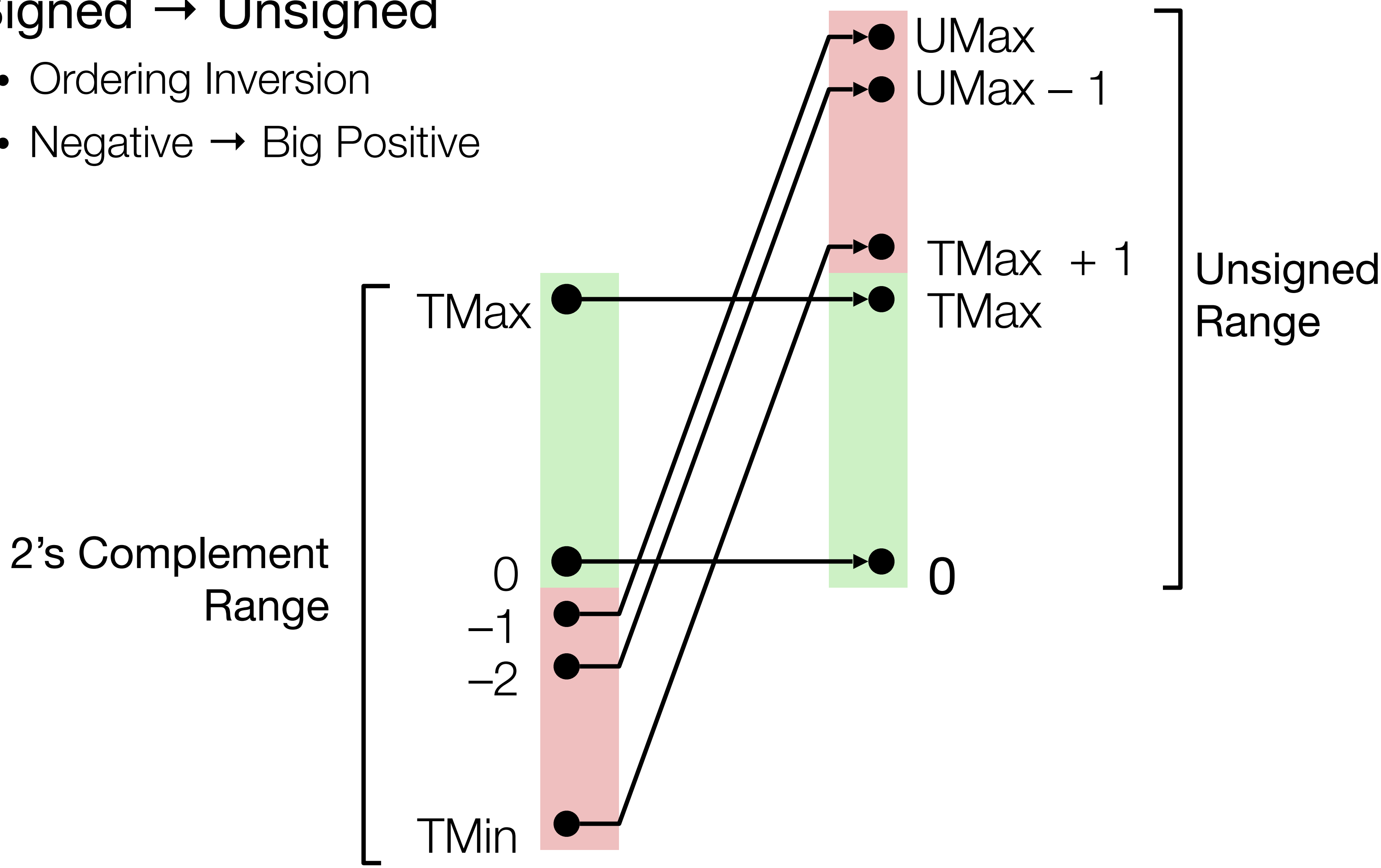
Conversion Visualized

- 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	64-bit
`char`	1
`short`	2
`int`	4
`long`	8

The Problem

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

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

- Converting from smaller to larger integer data type
- Should always be able to preserve the value, but how?

The Problem

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

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

- Converting from smaller to larger integer data type
- Should always be able to preserve the value, but 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

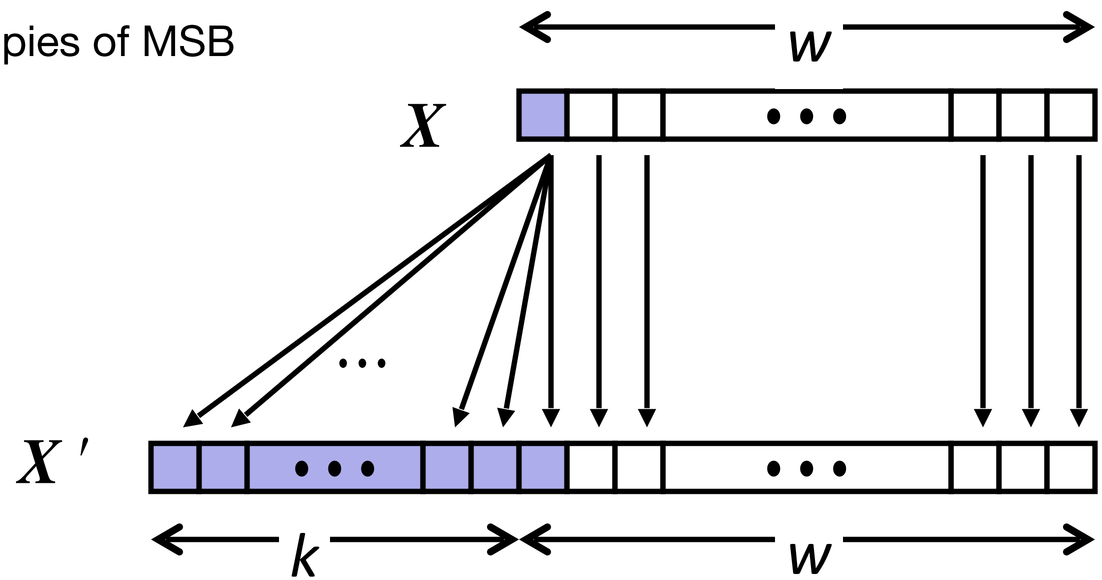
- Task:
  - Given *w*-bit signed integer *x*
  - Convert it to (*w*+*k*)-bit integer with same value

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:
  - $X' = \underbrace{x_{w-1}, \ldots, x_{w-1}}_{k \text{ copies of MSB}}, x_{w-1}, x_{w-2}, \ldots, x_0$



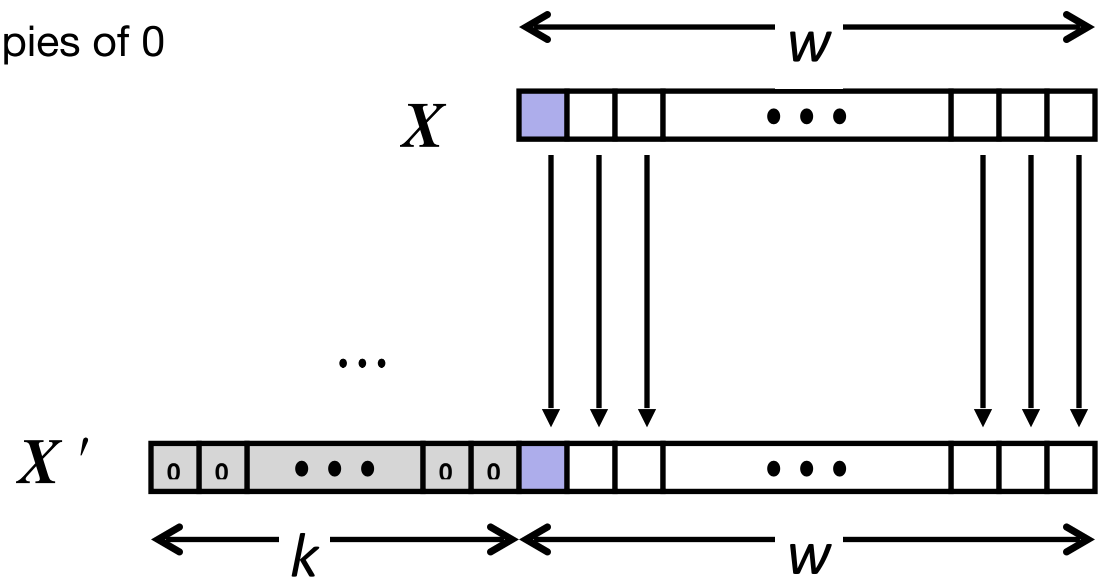
Another Problem

```
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

Questions?

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)
  - Can’t always preserve the numerical value
  - C’s implementation: leading bits are truncated, results reinterpreted

Questions?

Announcement

- Check the course website before asking
  - http://www.cs.rochester.edu/courses/252/spring2019/
- Direct ALL questions regarding assignments to the TAs
  - They have done them. They have debugged them. They know them inside out.
  - If one doesn’t know, ask another.
  - If all don’t know, ask me.

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            2
+)  101       +)   5
---------     --------
    111            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

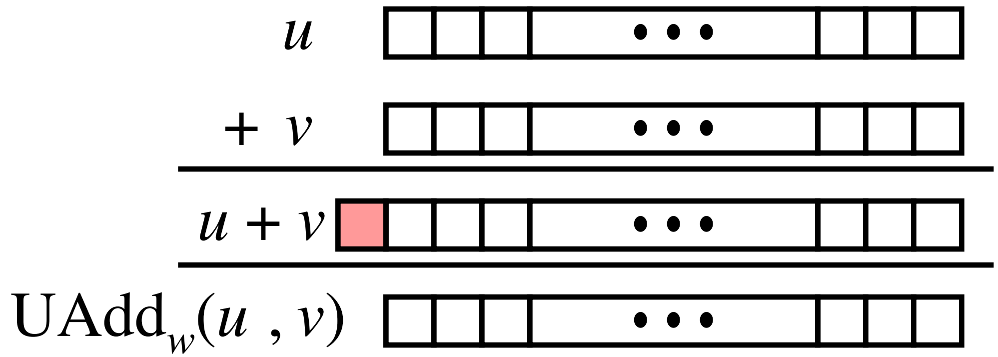
```
    110          6
+)  101     +)   5
-------     ------
   1011         11    <- True Sum
    011          3    <- Sum with same bits
```

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)

Normal Case

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

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)
- 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

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 → 3

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          100          -4
```

Negative Overflow

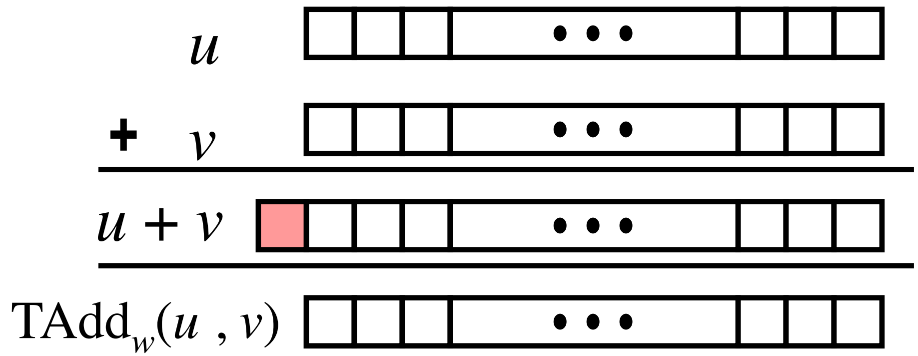
Positive Overflow

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
```

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?

```
    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

Is This Signed Addition an Overflow?

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

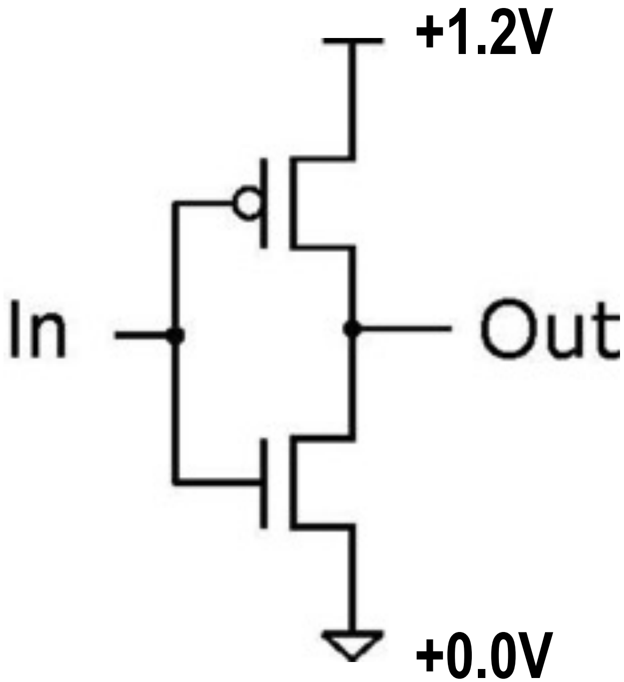
Truncate →

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

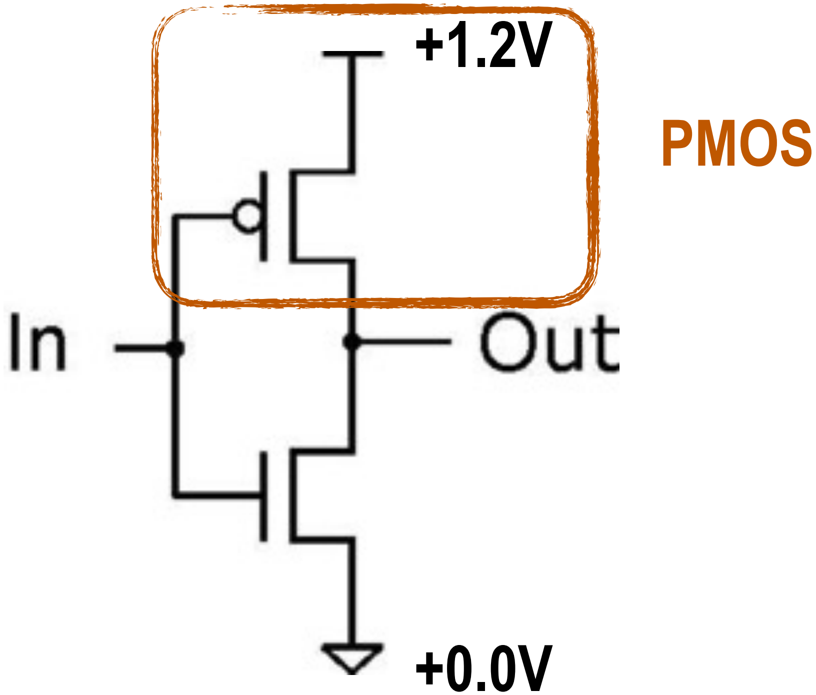
Signed	Binary
0	000
1	001
2	010
3	011
-4	100
-3	101
-2	110
-1	111

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

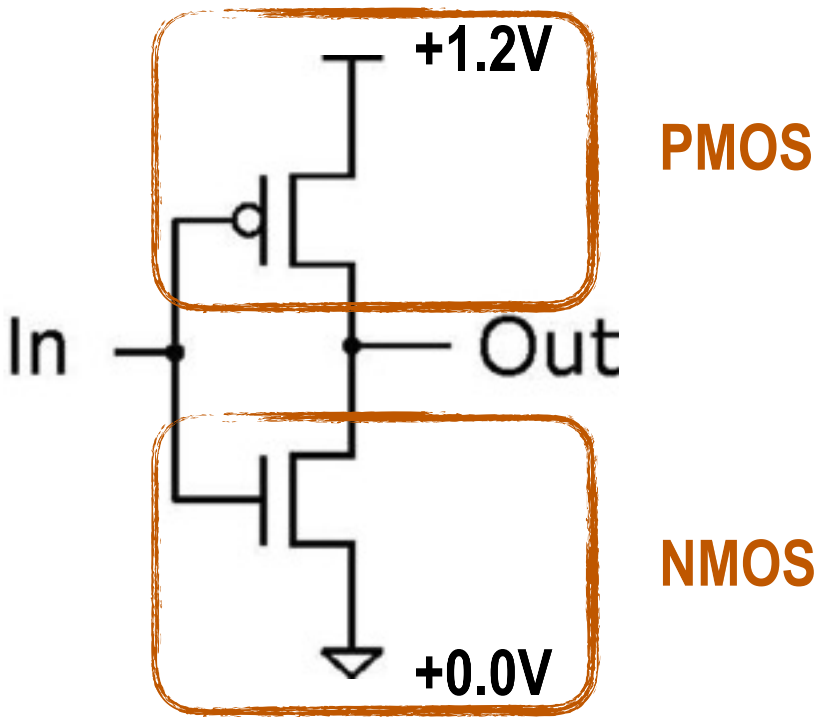
Inverter (NOT Gate)



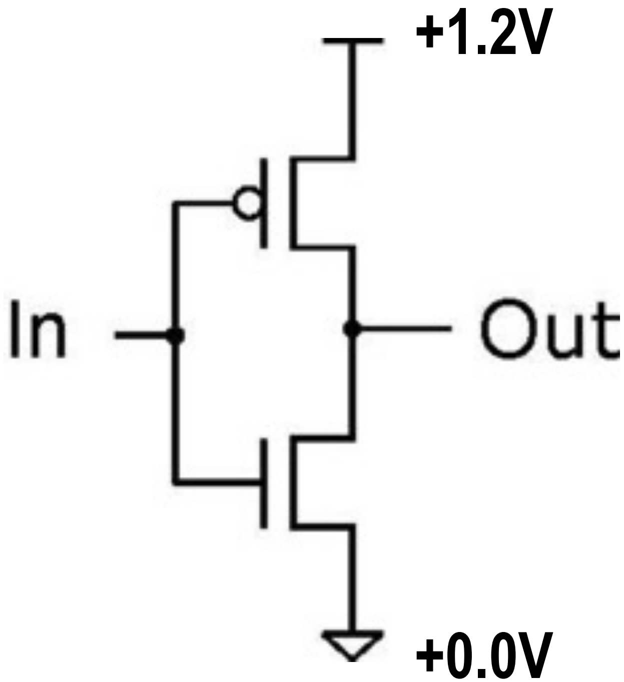
Inverter (NOT Gate)

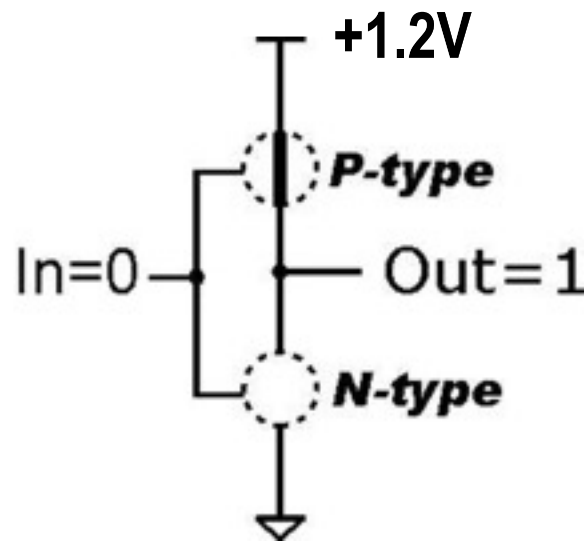


Inverter (NOT Gate)

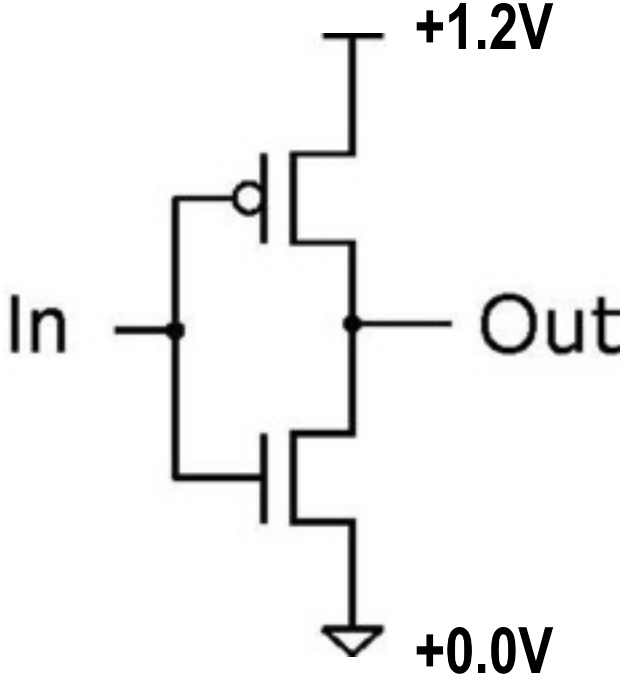


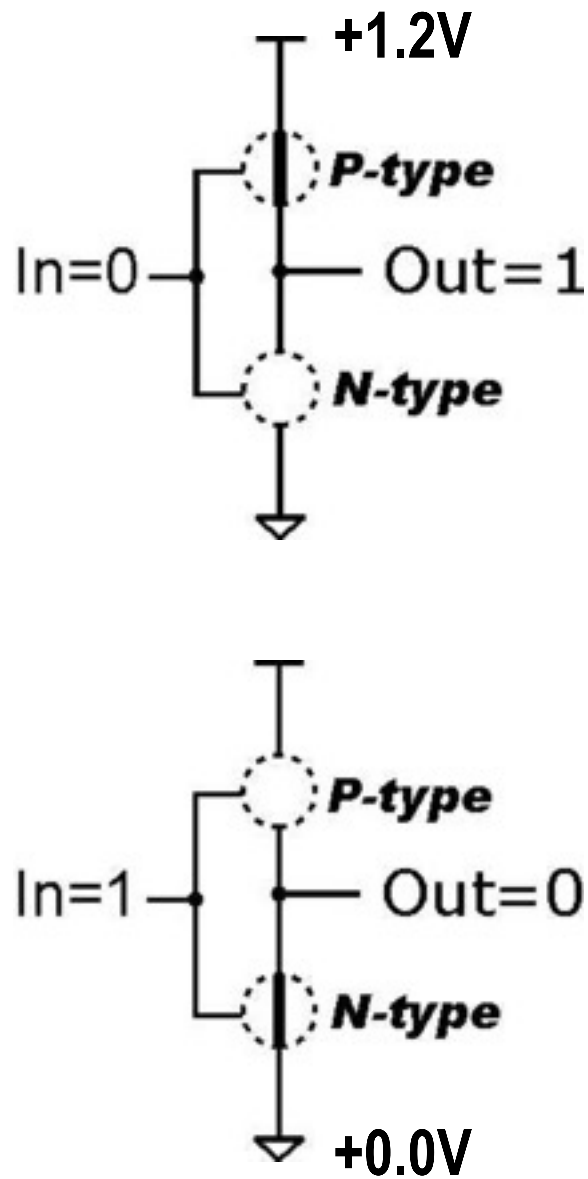
Inverter (NOT Gate)



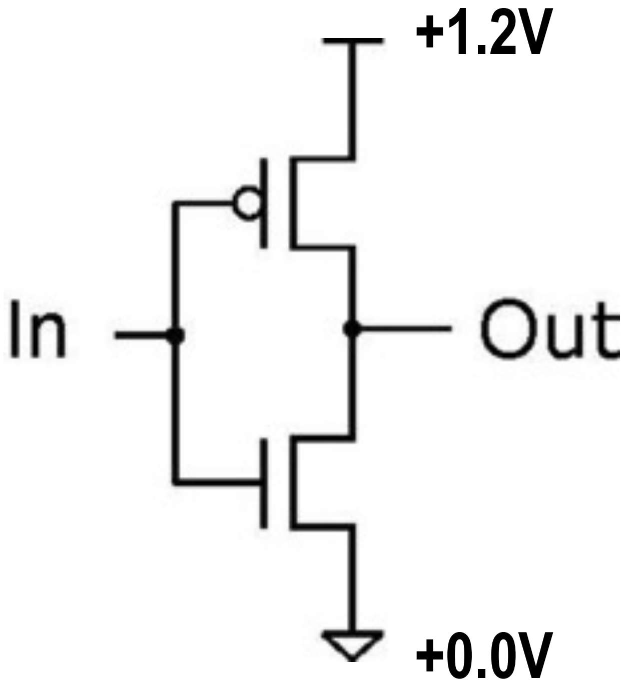


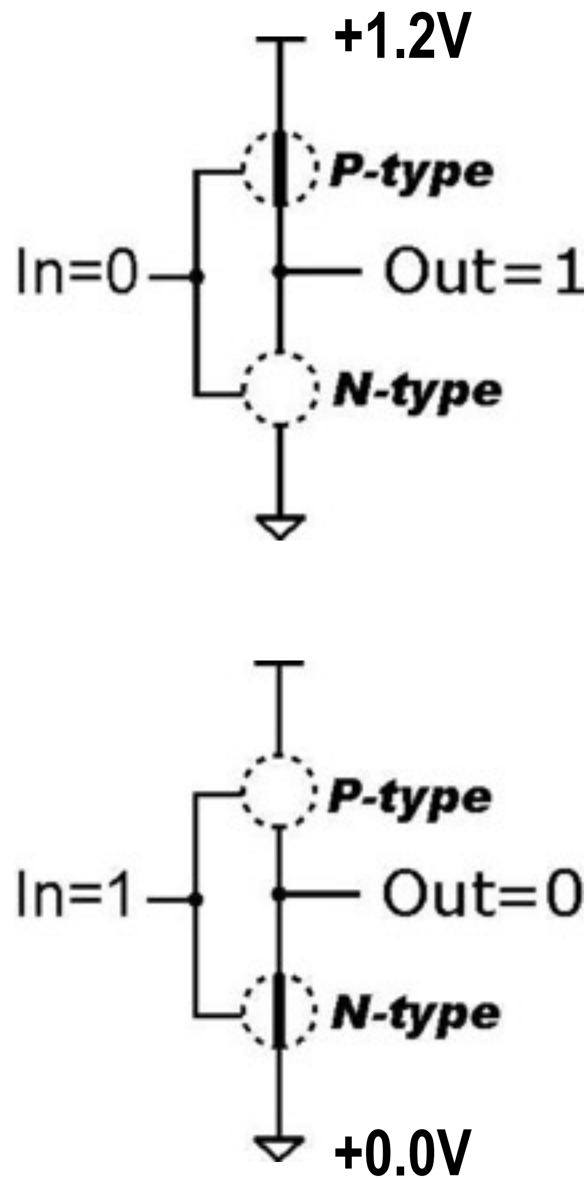
Inverter (NOT Gate)





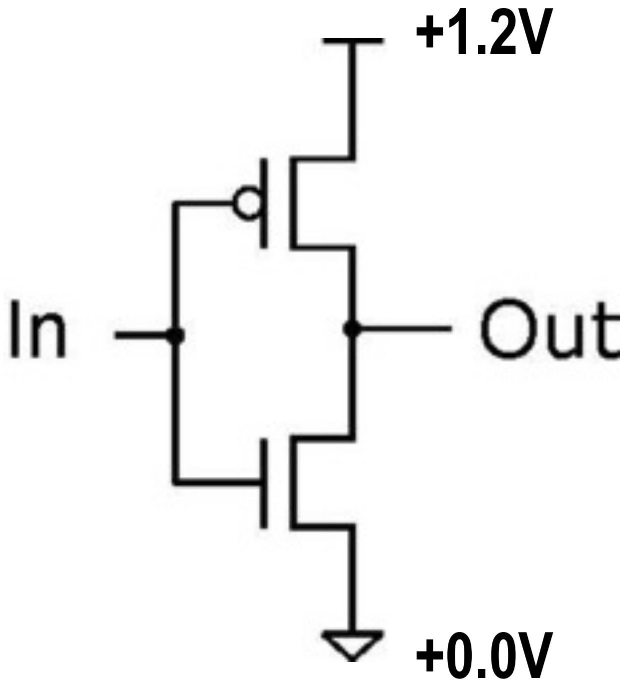
Inverter (NOT Gate)

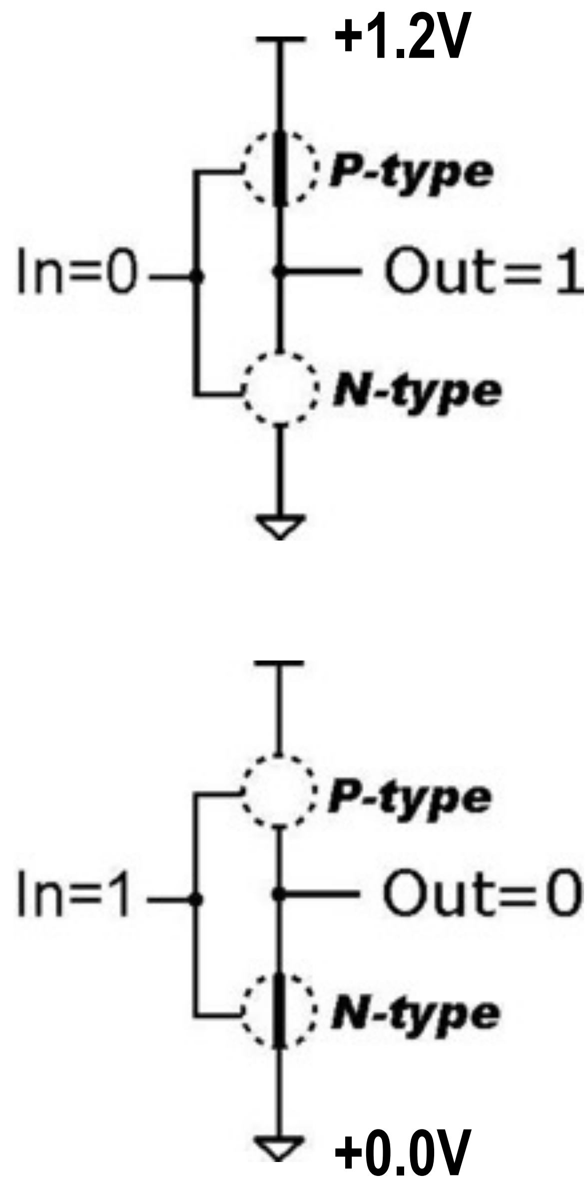




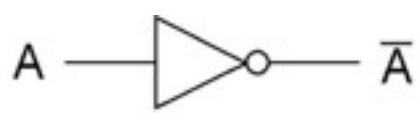
In	Out
0	1
1	0

Inverter (NOT Gate)

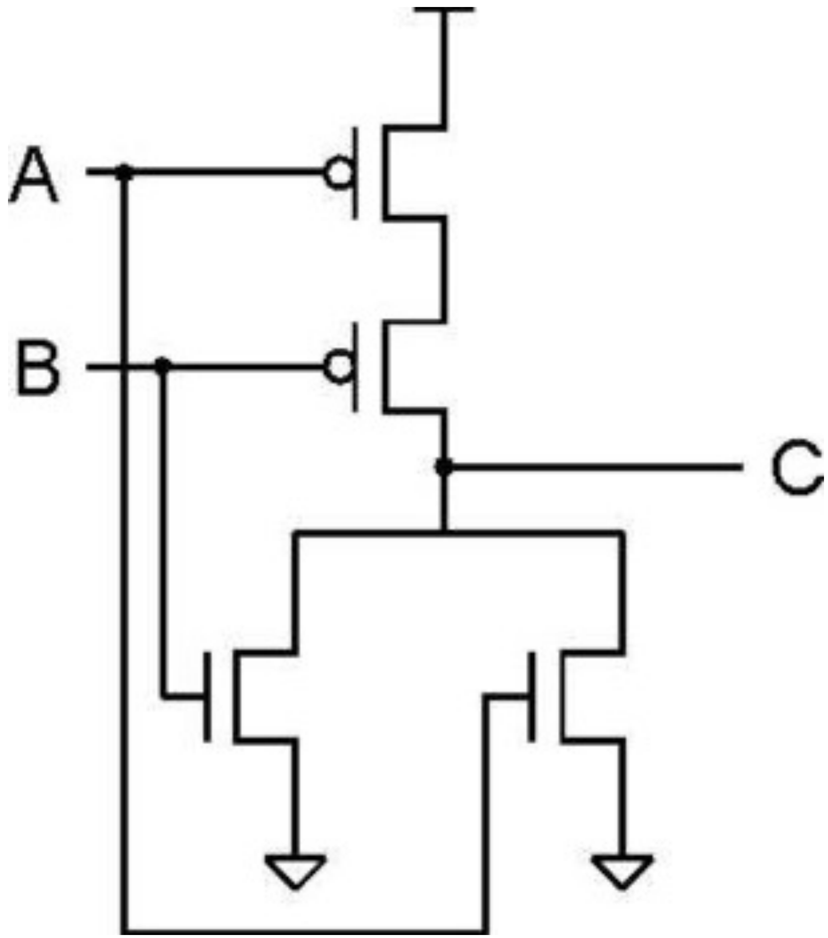


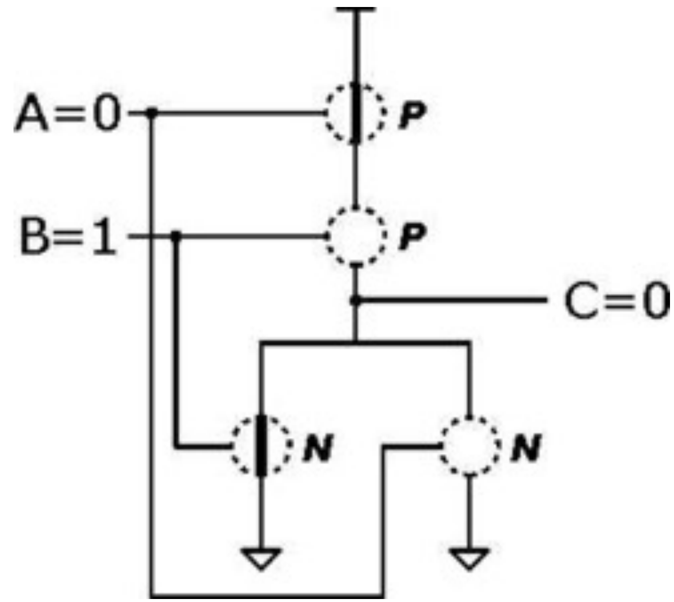


In	Out
0	1
1	0



NOR Gate (NOT + OR)

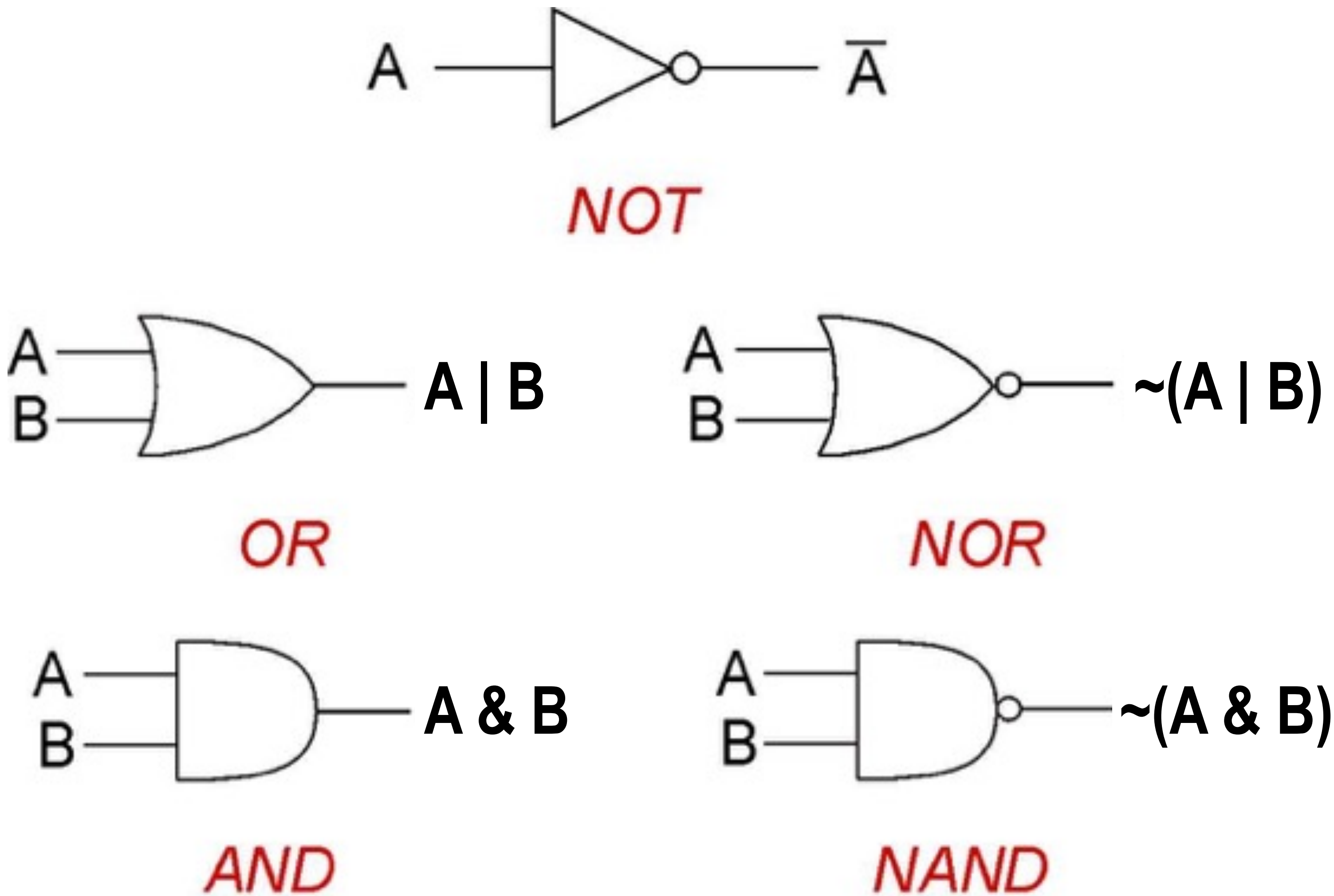




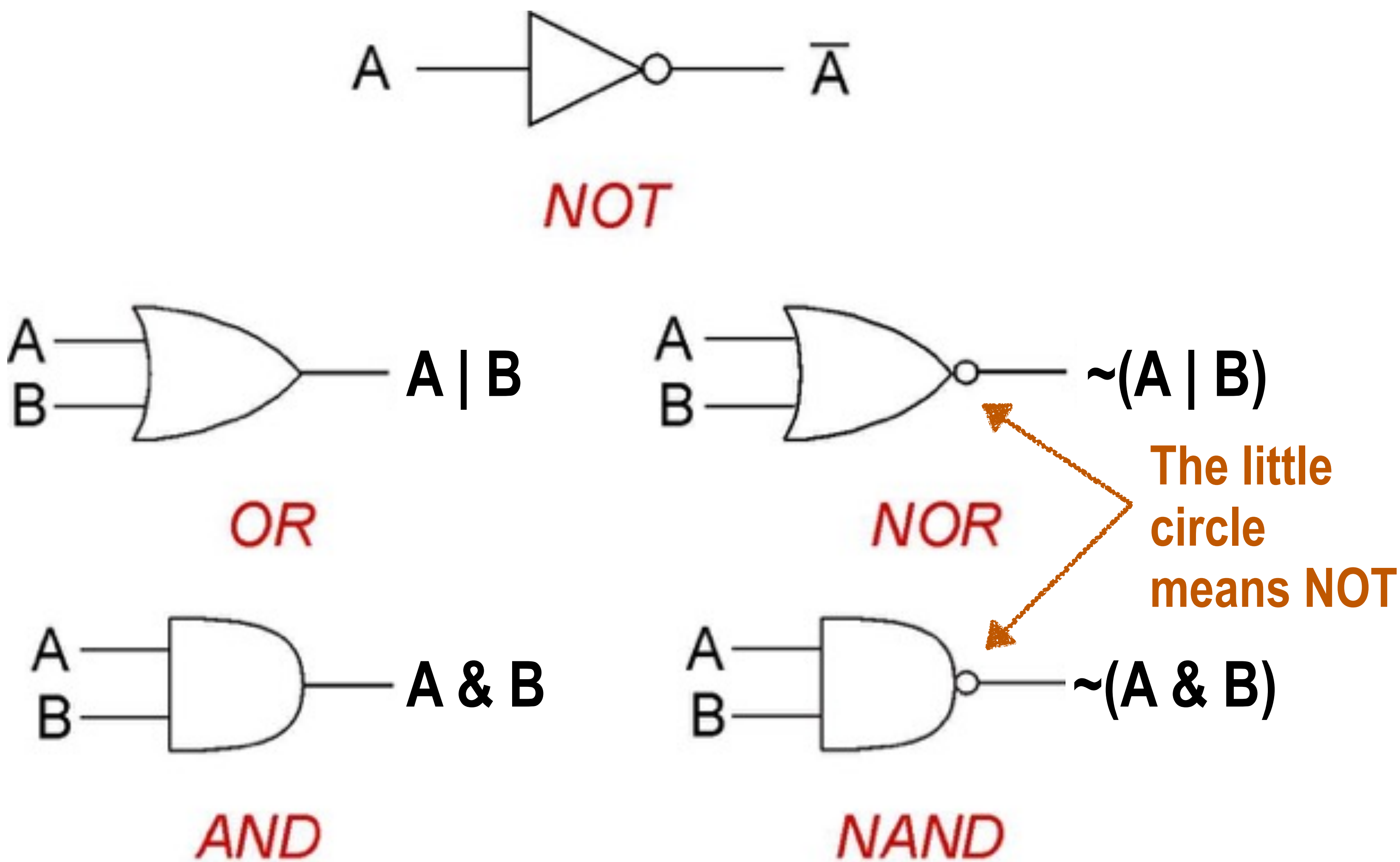
A	B	C
0	0	1
0	1	0
1	0	0
1	1	0

Note: Serial structure on top, parallel on bottom.

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.

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

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

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.

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

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

Truth Table

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

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

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

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

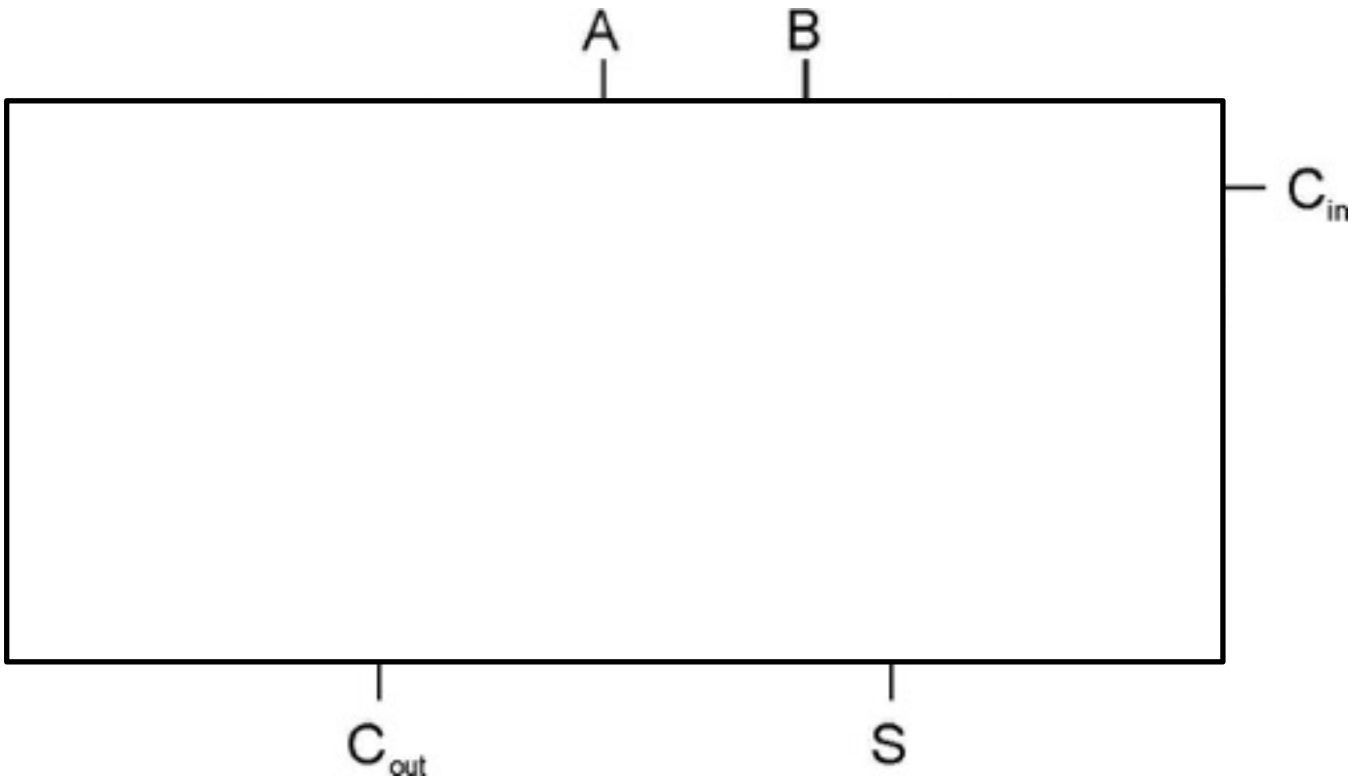
Add two bits and carry-in,
produce one-bit sum and carry-out.

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

Full (1-bit) Adder

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

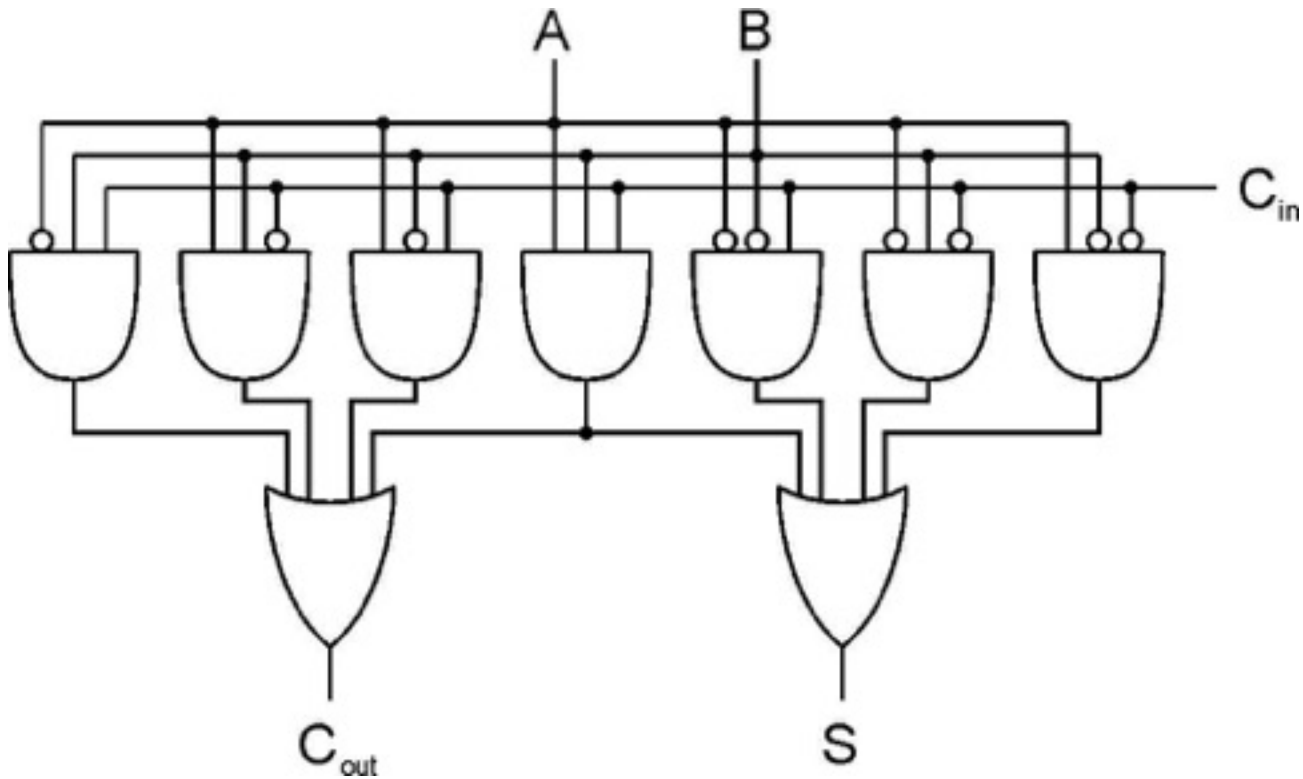
Add two bits and carry-in,
produce one-bit sum and carry-out.



Full (1-bit) Adder

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

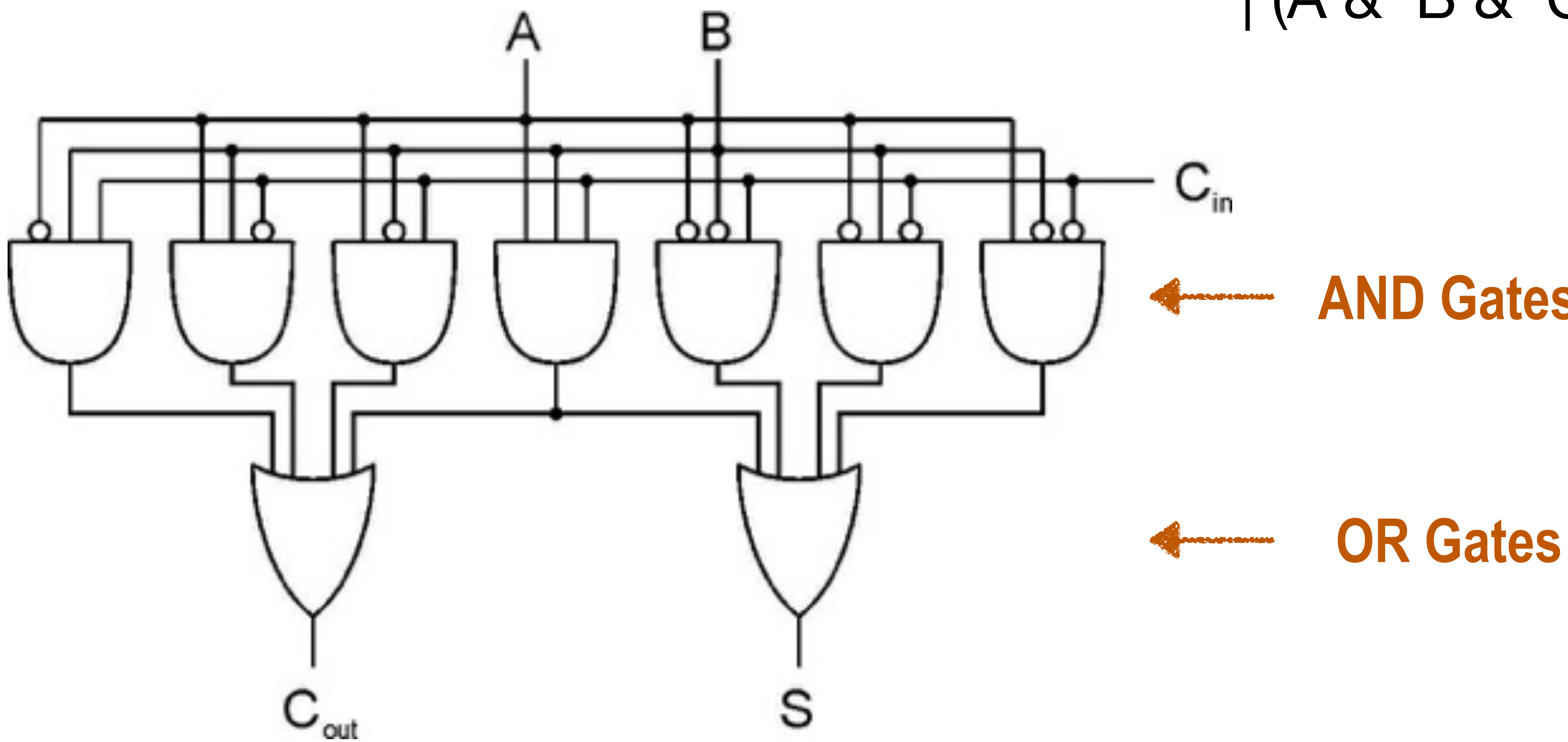
Add two bits and carry-in,
produce one-bit sum and carry-out.



Full (1-bit) Adder

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

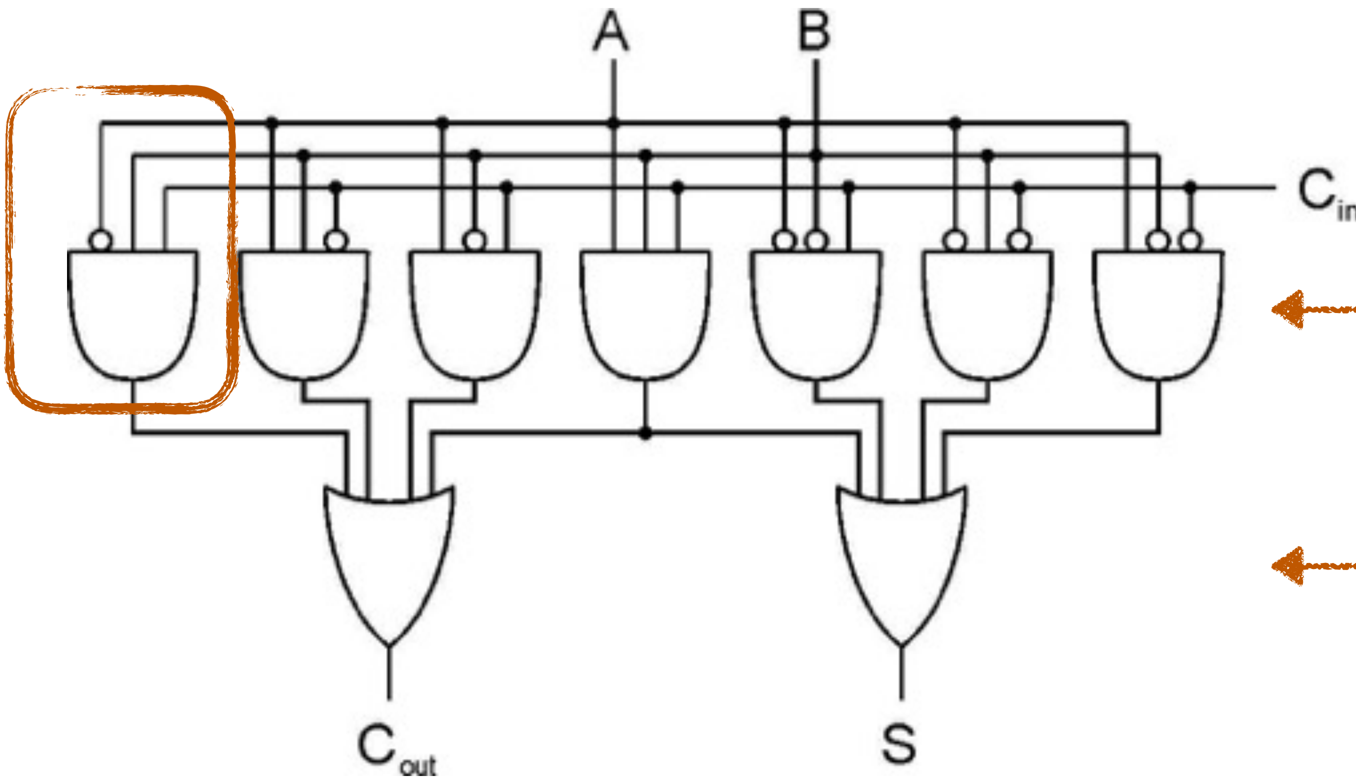
C_{ou} = (~A & B & C_{in})
| (A & ~B & C_{in})
| (A & B & ~C_{in})
| (A & B & C_{in})



Full (1-bit) Adder

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

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



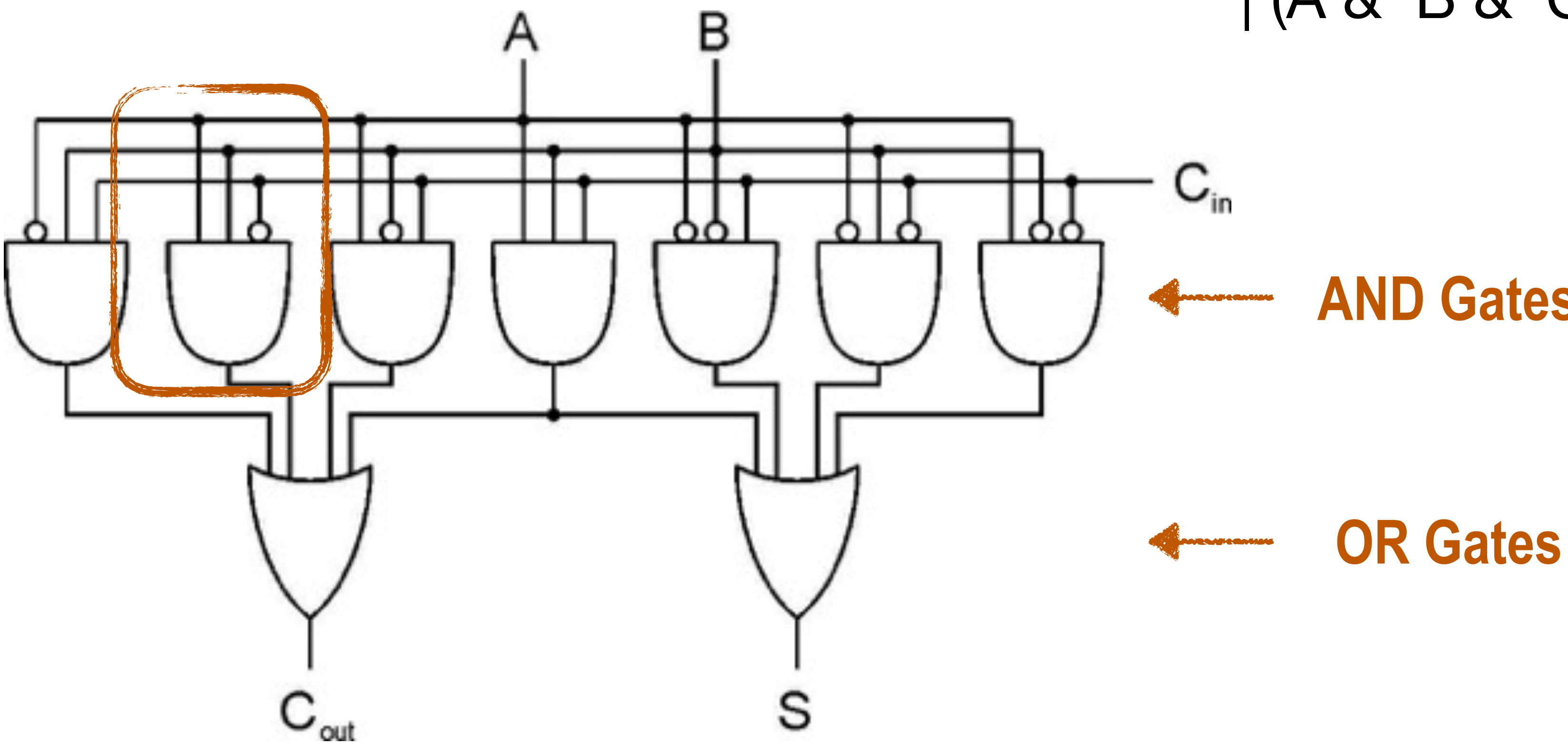
AND Gates

OR Gates

Full (1-bit) Adder

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

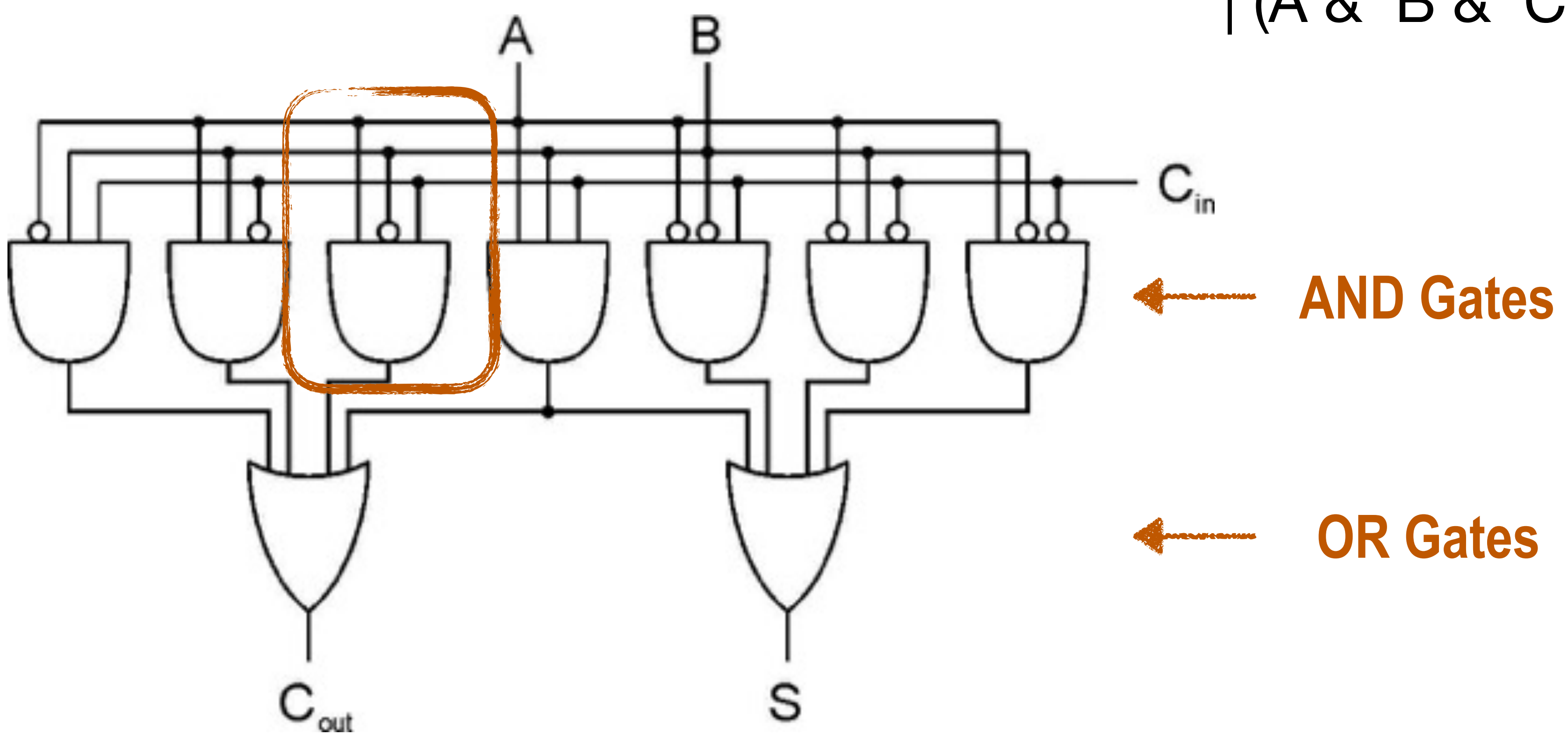
Add two bits and carry-in,
produce one-bit sum and carry-out.



Full (1-bit) Adder

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

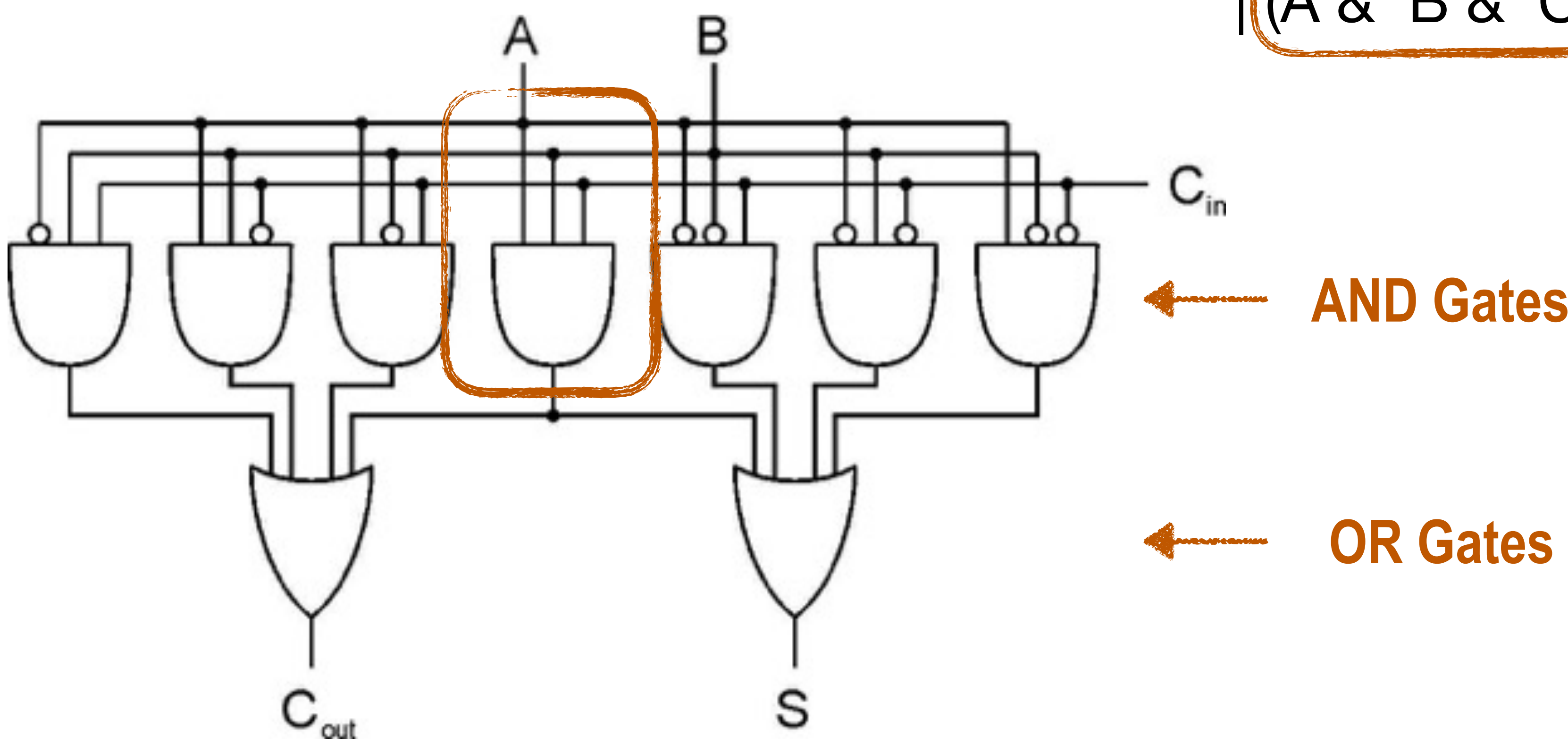
$$C_{ou} = (\sim A \;\&\; B \;\&\; C_{in}) \;|\; (A \;\&\; \sim B \;\&\; C_{in}) \;|\; (A \;\&\; B \;\&\; \sim C_{in}) \;|\; (A \;\&\; B \;\&\; C_{in})$$



Full (1-bit) Adder

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

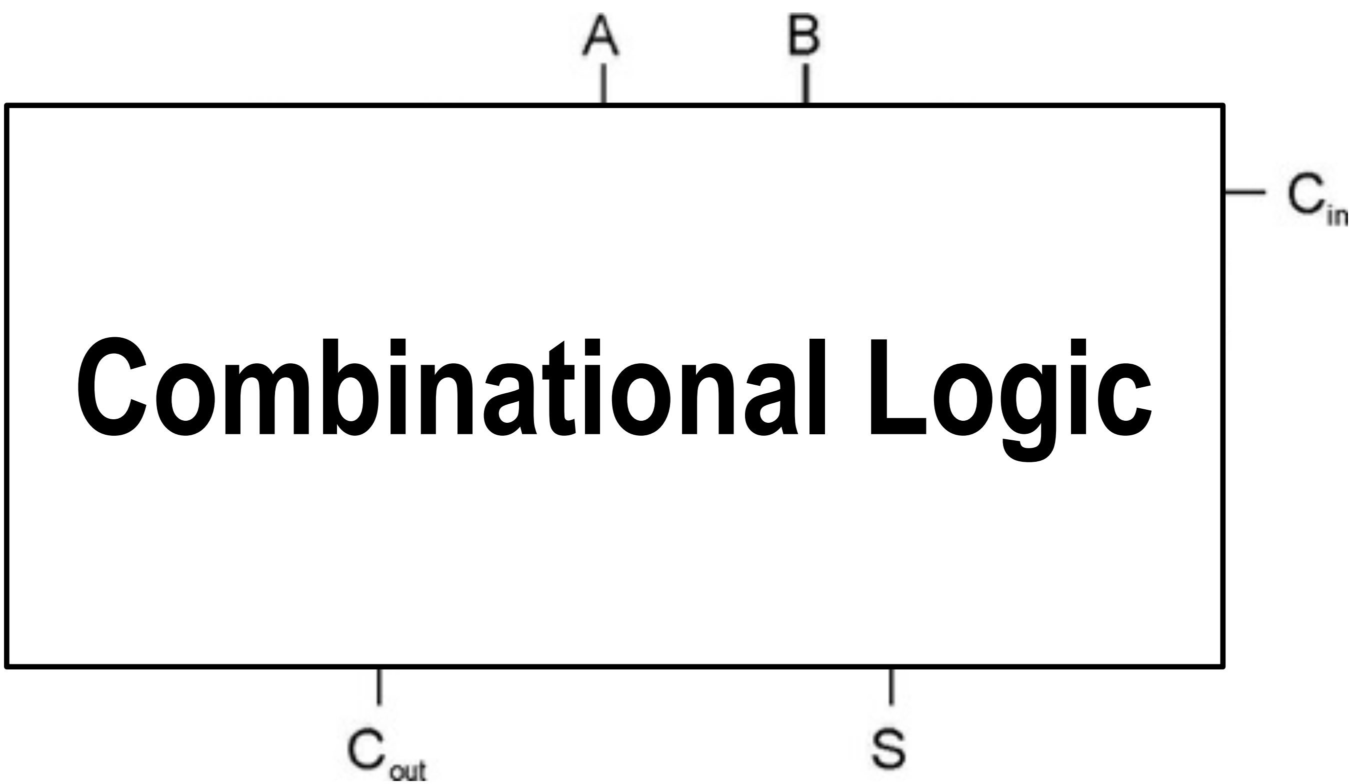
Add two bits and carry-in,
produce one-bit sum and carry-out.



Full (1-bit) Adder

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

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



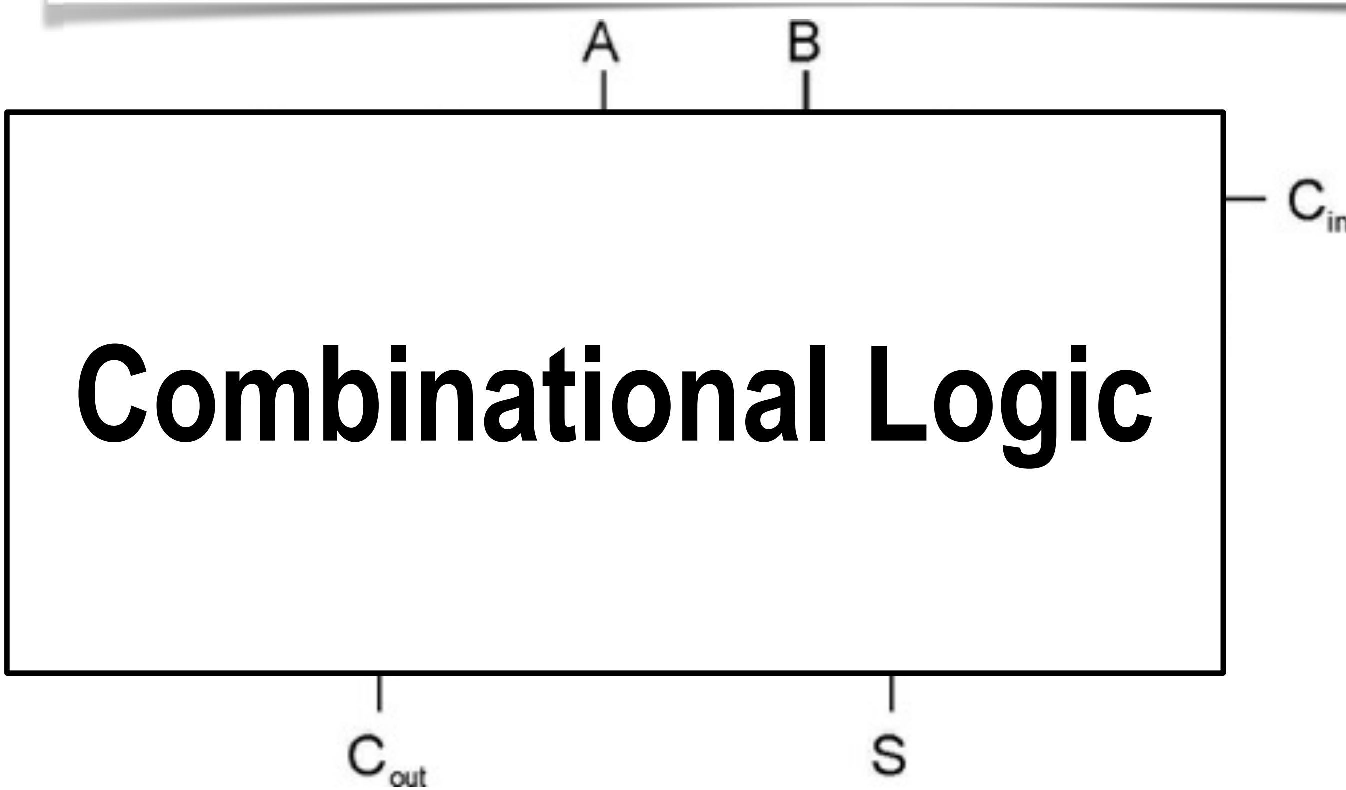
Full (1-bit) Adder

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

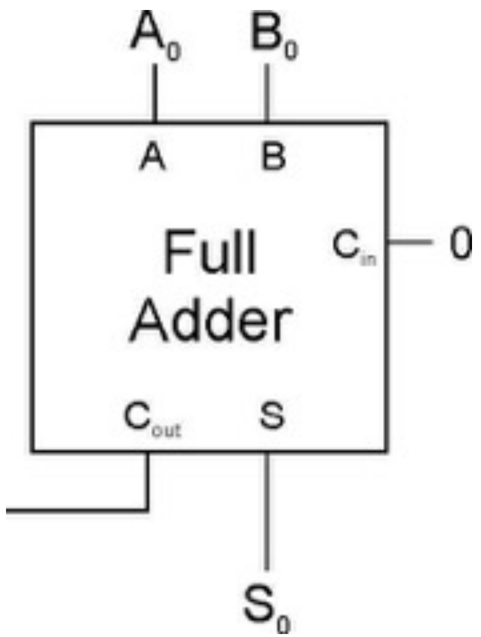
$\quad | \ (A \,\&\, \sim B \,\&\, C_{in})$

$\quad | \ (A \,\&\, B \,\&\, C_{in})$

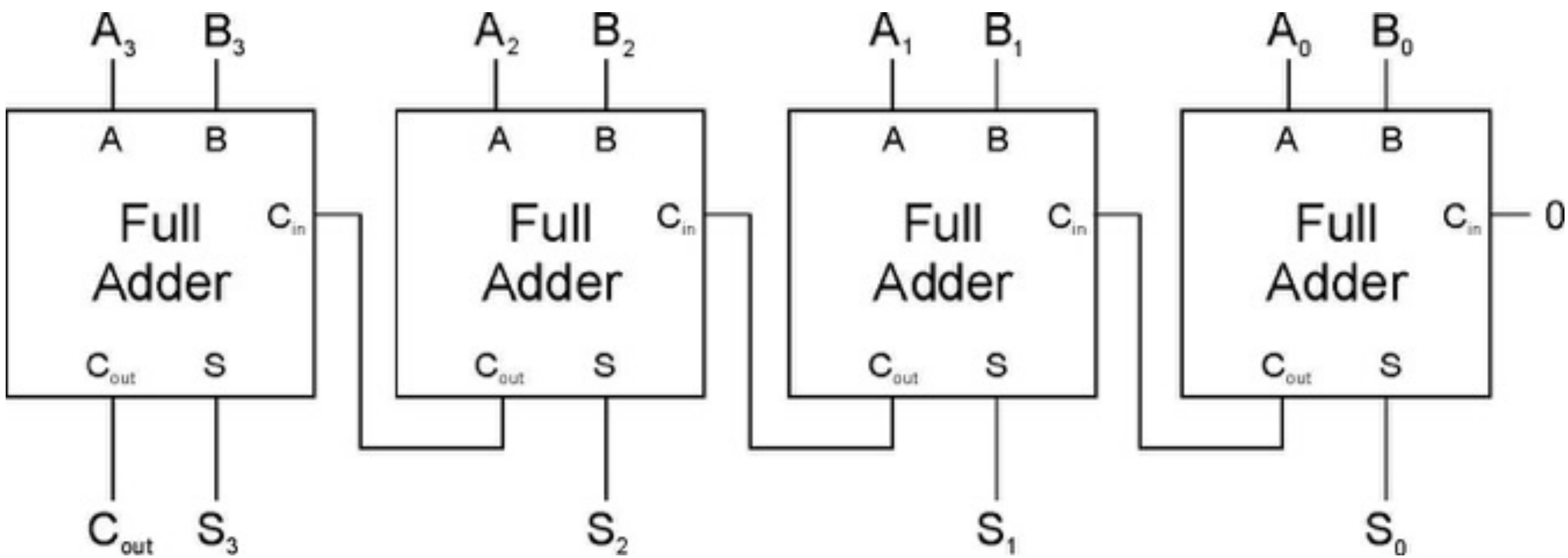
Outputs depend only on current inputs (i.e., not the past inputs), continuously (with some delay)



Four-bit Adder

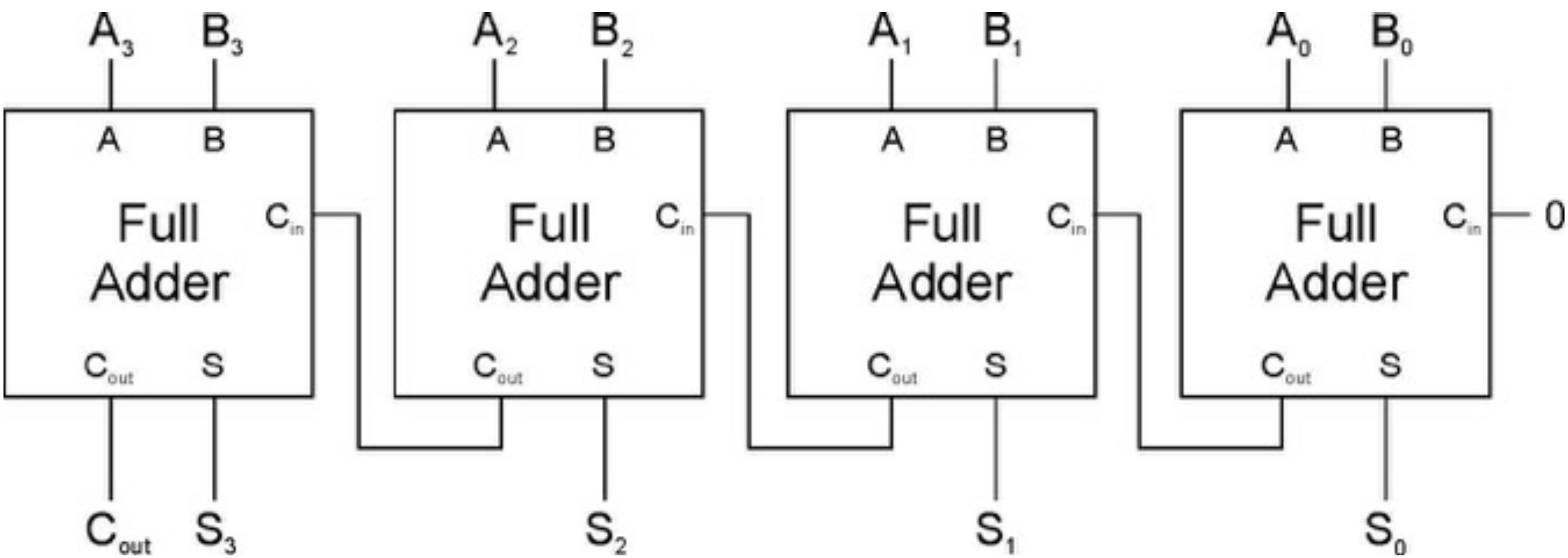


Four-bit Adder



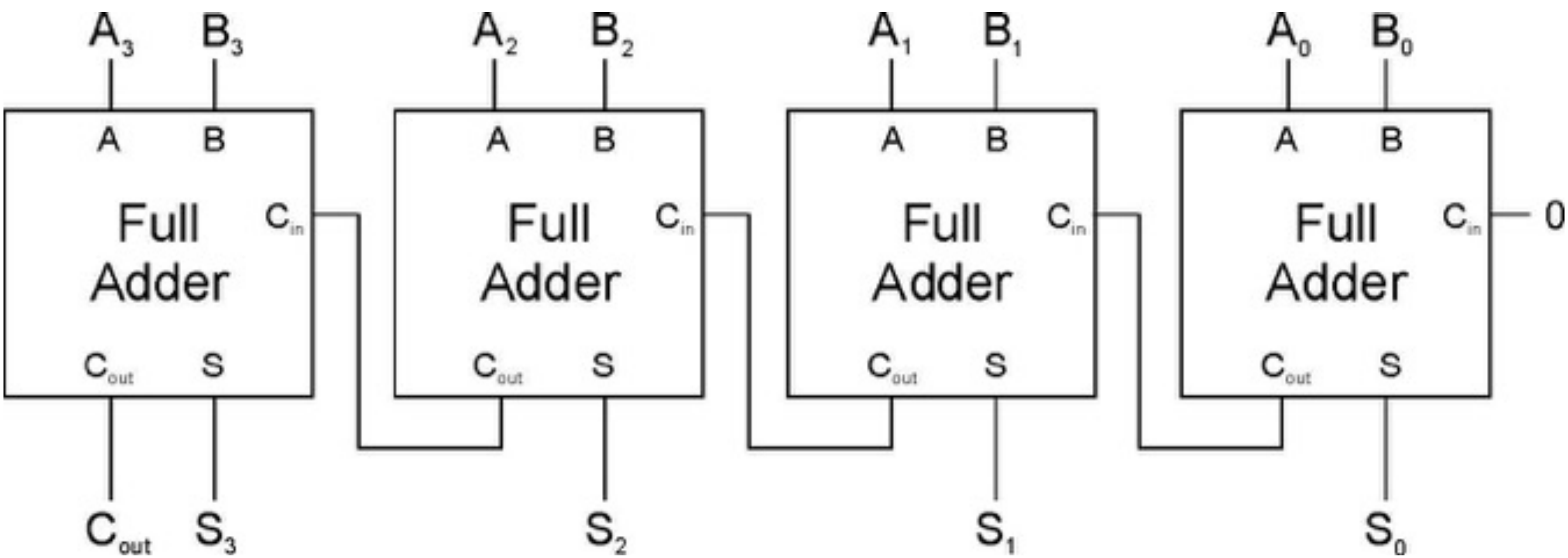
Four-bit Adder

- 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

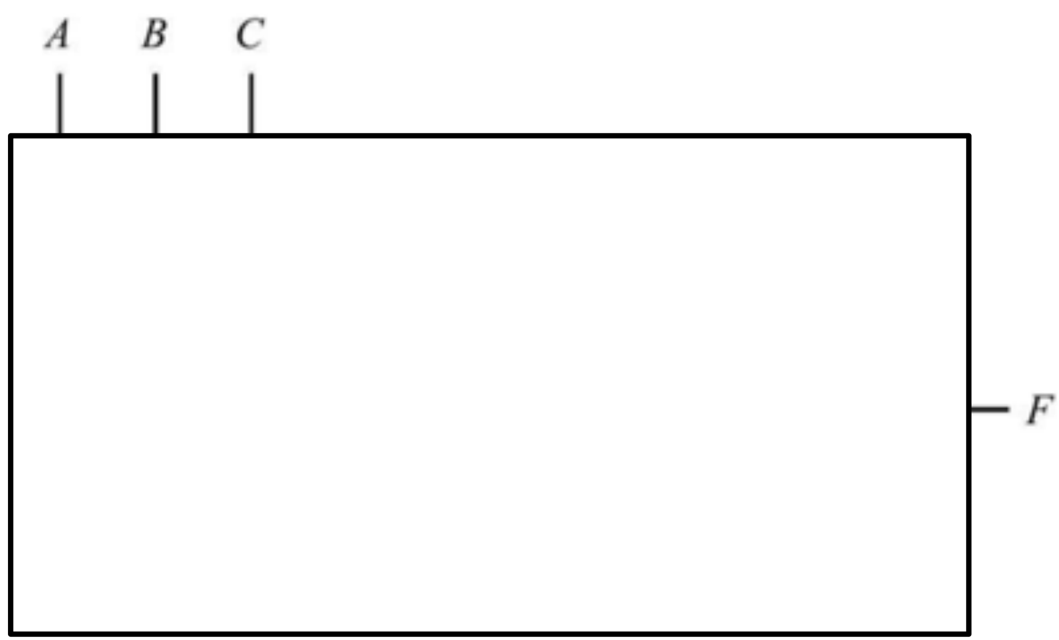
- Design digital components from basic logic gates
- Key idea: use the truth table!

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?

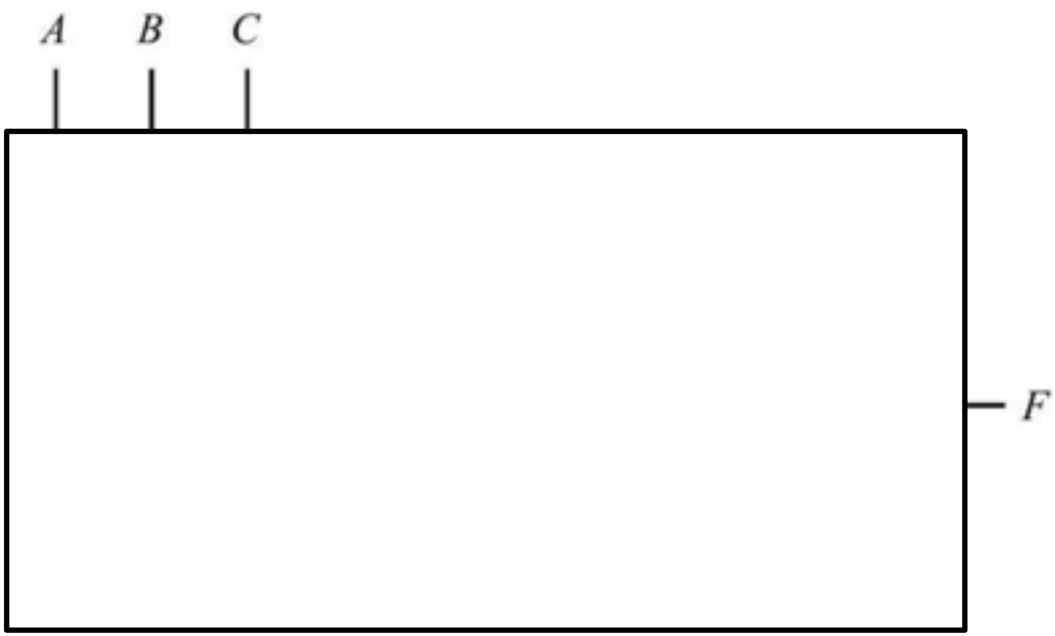
Logic Design

- 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

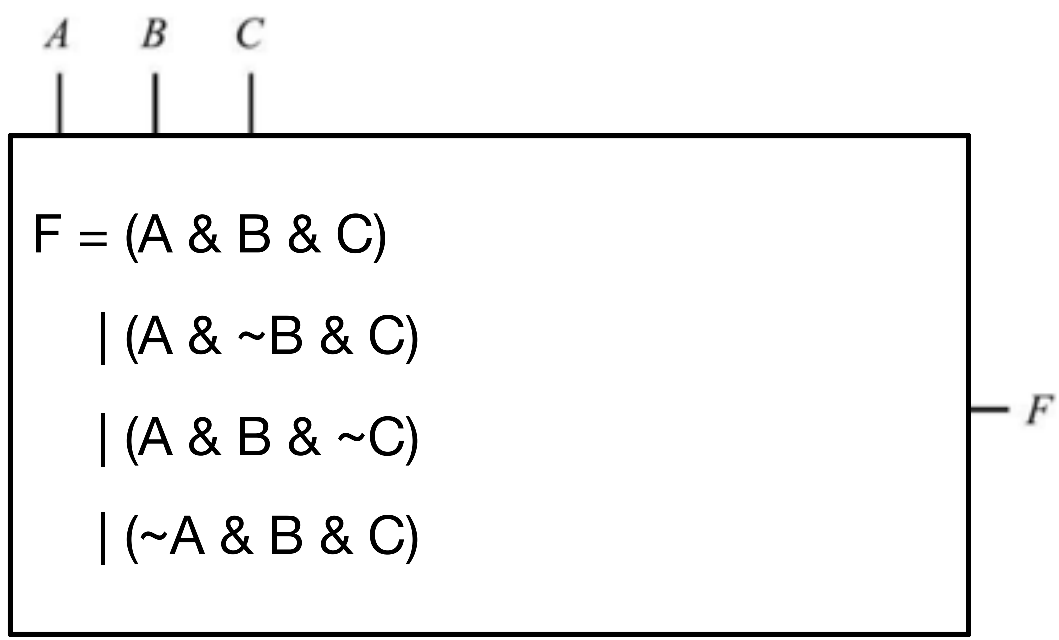
- Design digital components from basic logic gates
- Key idea: use the truth table!
- Example: how to design a piece of circuit that does majority vote?



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

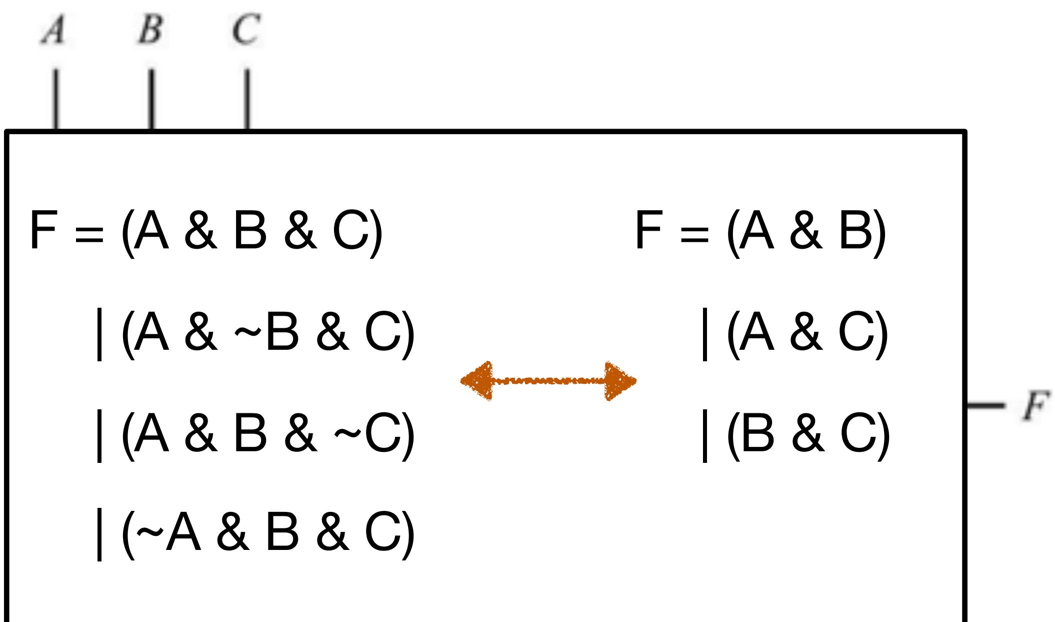
- Design digital components from basic logic gates
- Key idea: use the truth table!
- Example: how to design a piece of circuit that does majority vote?



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

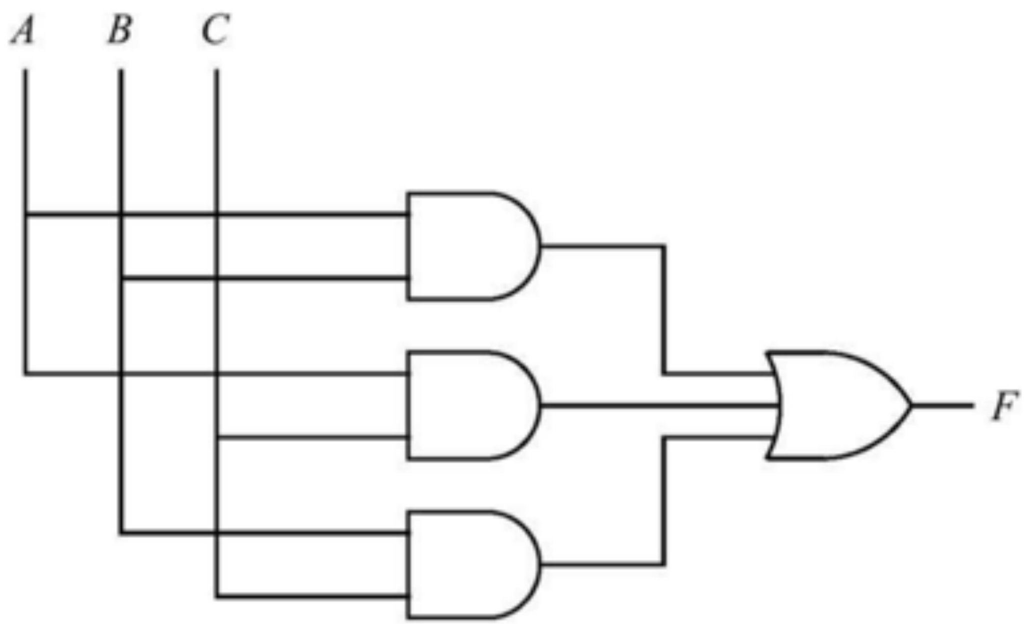
- Design digital components from basic logic gates
- Key idea: use the truth table!
- Example: how to design a piece of circuit that does majority vote?



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?

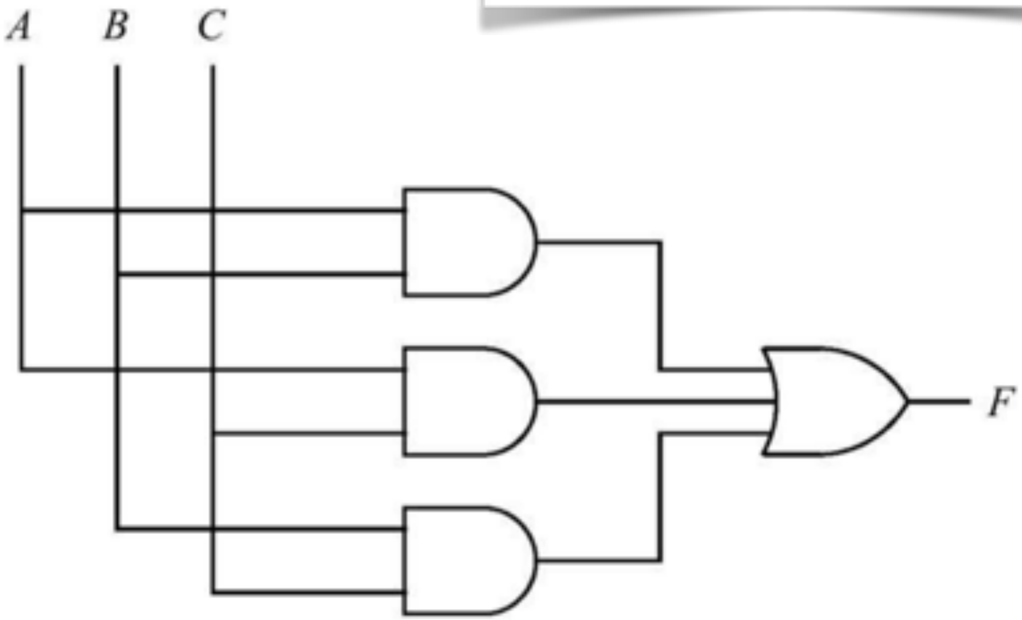


A	B	C	F
0	0	0	**0**
0	0	1	**0**
0	1	0	**0**
0	1	1	**1**
1	0	0	**0**
1	0	1	**1**
1	1	0	**1**
1	1	1	**1**

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?

ECE112 Logic Design



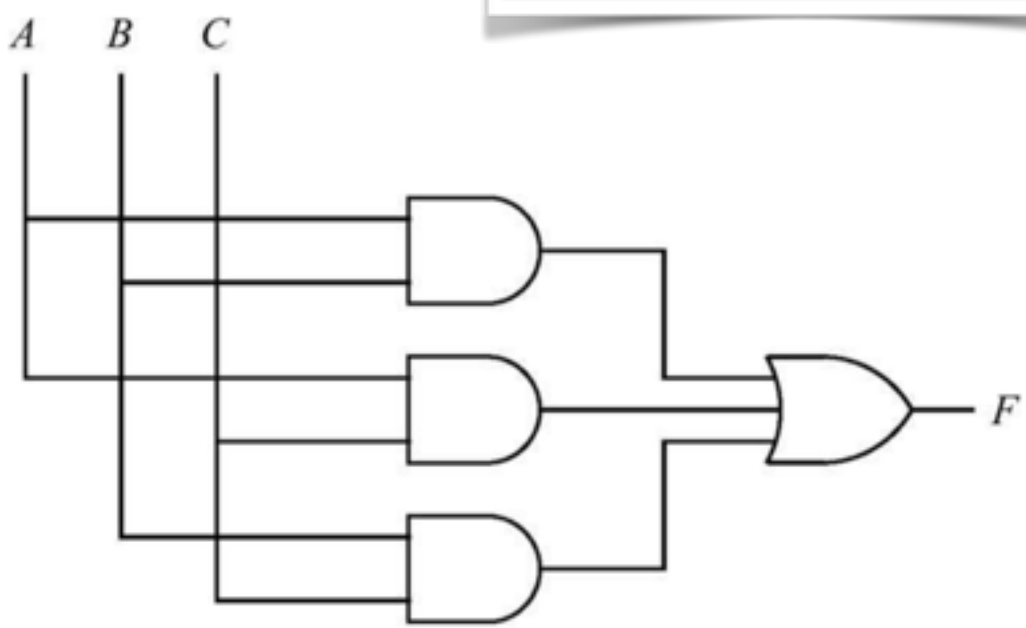
A	B	C	F
0	0	0	**0**
0	0	1	**0**
0	1	0	**0**
0	1	1	**1**
1	0	0	**0**
1	0	1	**1**
1	1	0	**1**
1	1	1	**1**

Logic Design

Questions?

- 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?

ECE112 Logic Design



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**

Multiplication

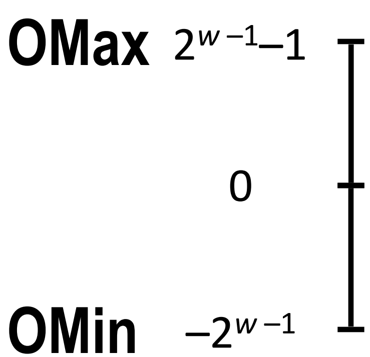
Multiplication

- Goal: Computing Product of w-bit numbers x, y

Multiplication

- Goal: Computing Product of w-bit numbers x, y

Original Number (w bits)

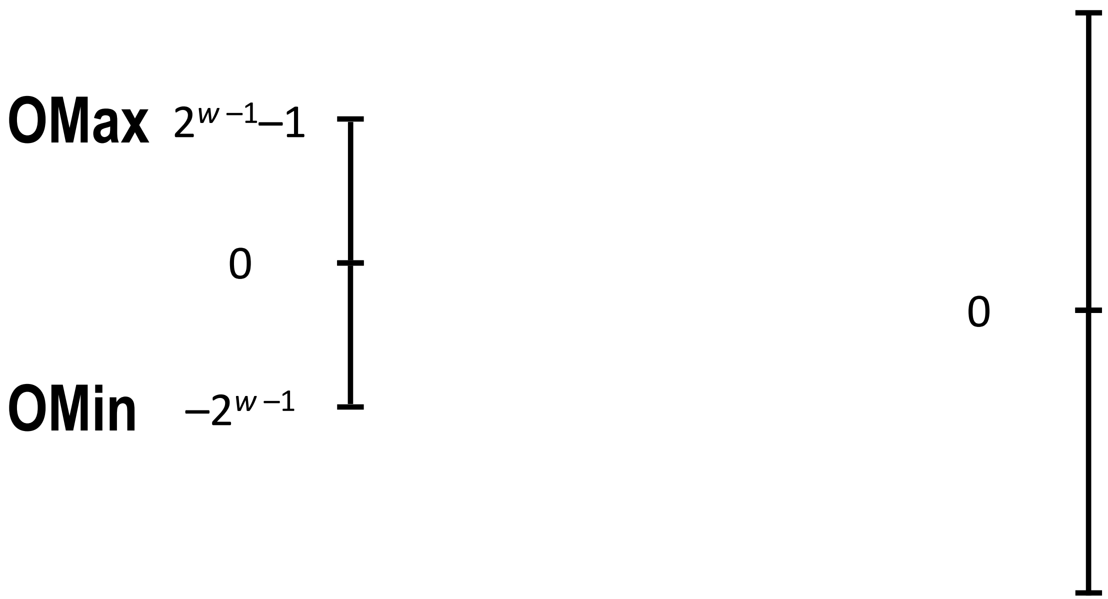


Multiplication

- Goal: Computing Product of *w*-bit numbers *x*, *y*

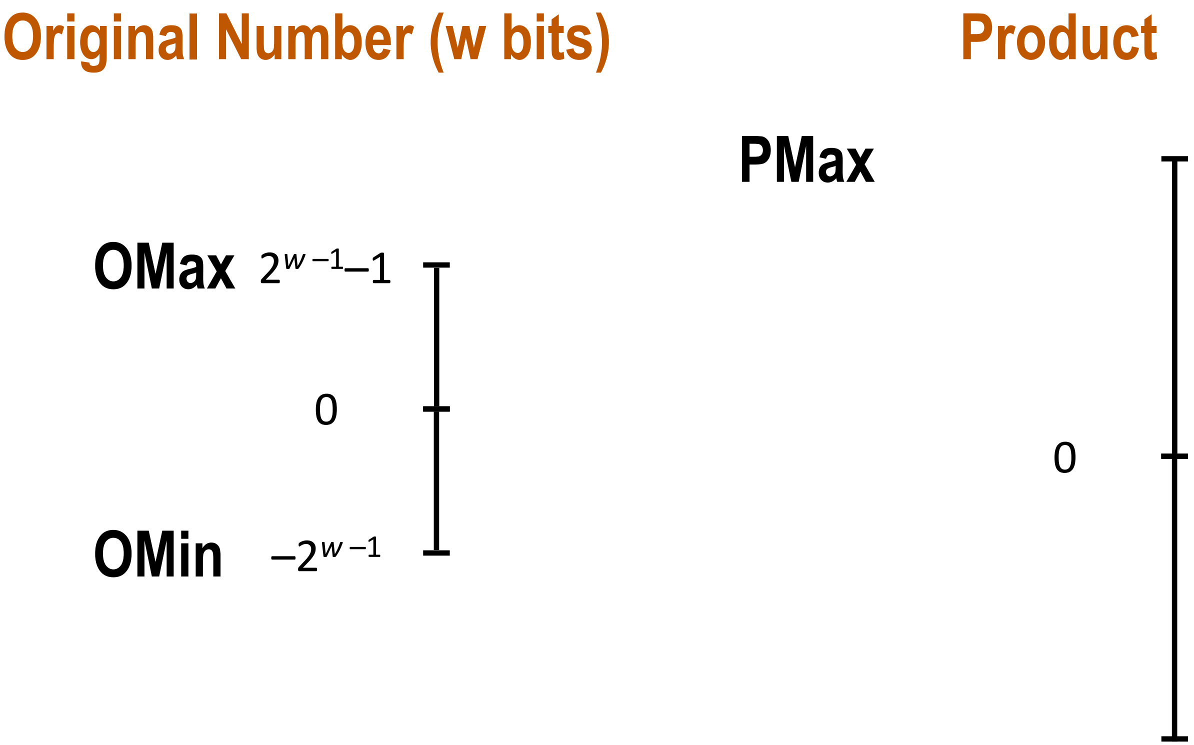
Original Number (w bits)

Product



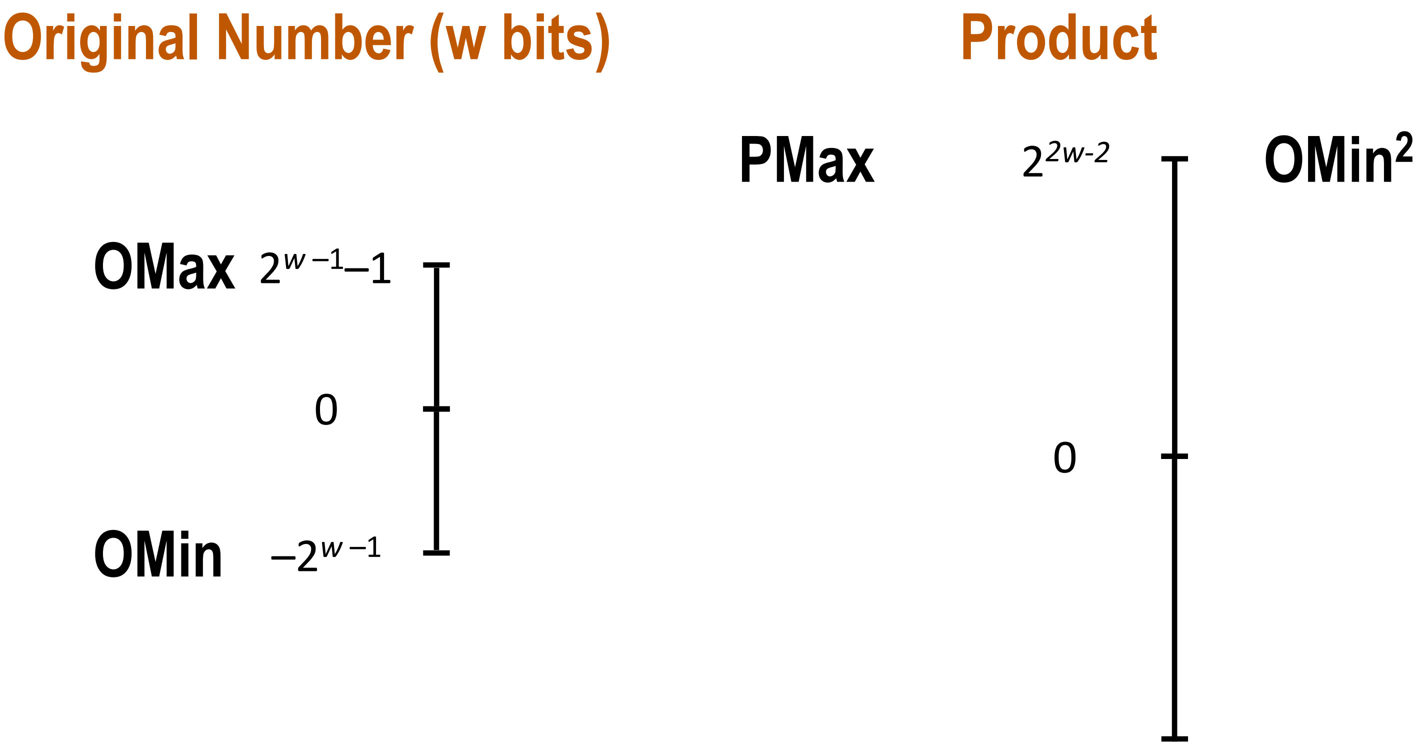
Multiplication

- Goal: Computing Product of *w*-bit numbers *x*, *y*



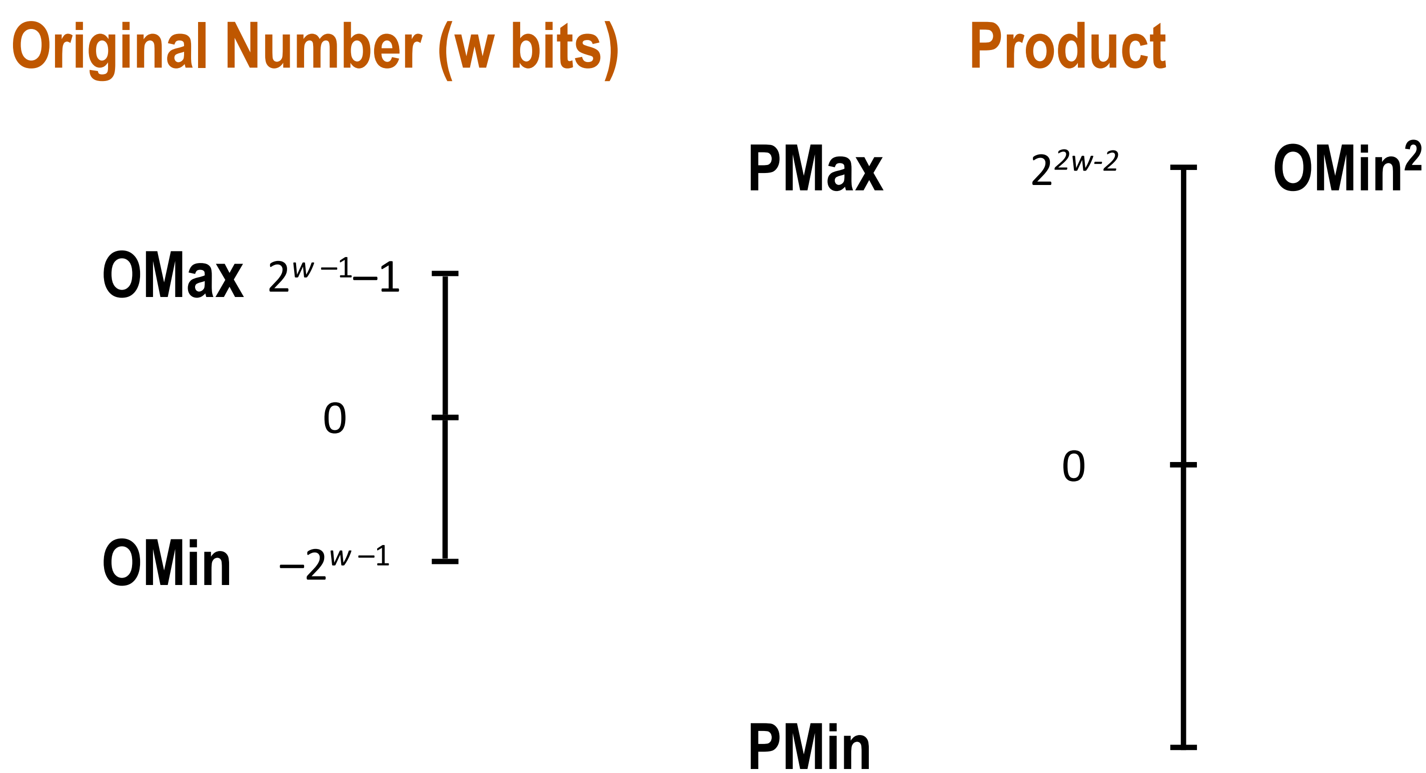
Multiplication

- Goal: Computing Product of *w*-bit numbers *x*, *y*



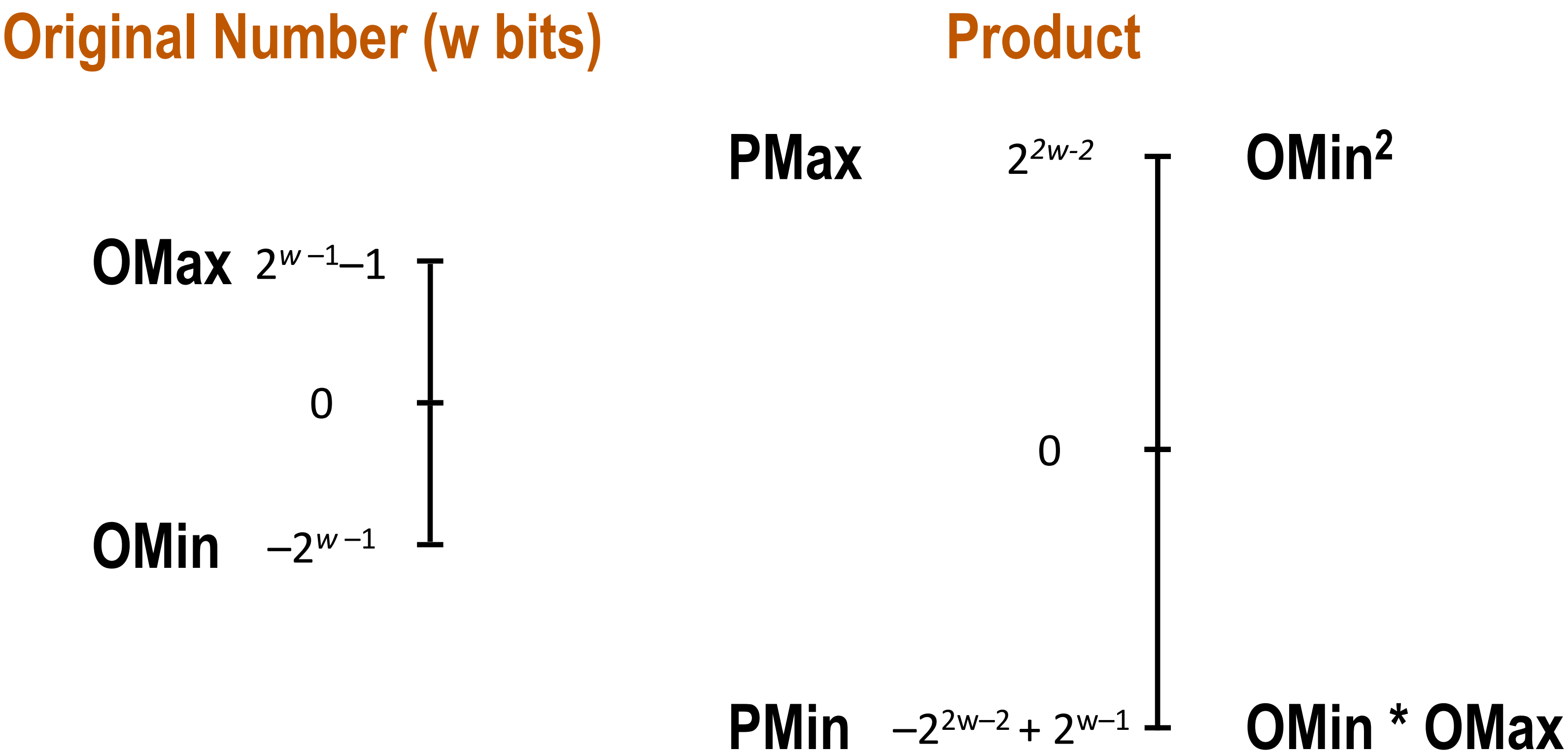
Multiplication

- Goal: Computing Product of *w*-bit numbers *x*, *y*



Multiplication

- Goal: Computing Product of w-bit numbers x, y



Multiplication

- Goal: Computing Product of *w*-bit numbers *x*, *y*

Original Number (w bits)

OMax $2^{w-1}-1$

0

OMin -2^{w-1}

Product (2w bits)

PMax 2^{2w-2} **OMin²**

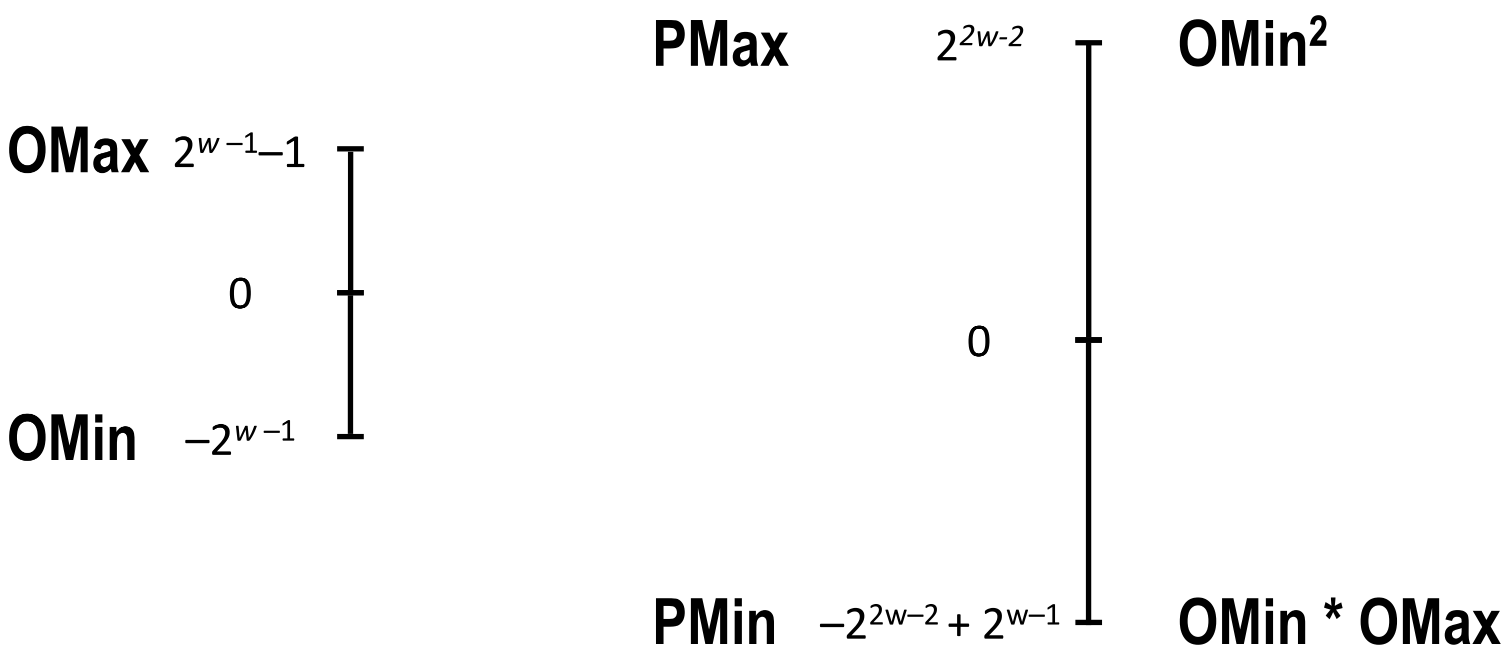
0

PMin $-2^{2w-2}+2^{w-1}$ **OMin * OMax**

Multiplication

- Goal: Computing Product of *w*-bit numbers *x*, *y*
- Exact results can be bigger than *w* bits
  - Up to *2w* bits (both signed and unsigned)

Original Number (w bits) **Product (2w bits)**

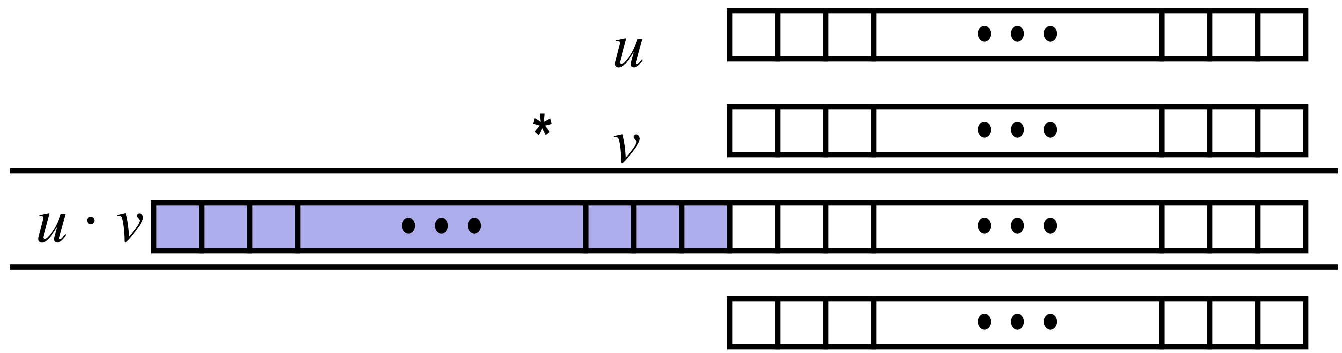


Unsigned Multiplication in C

Operands: *w* bits

True Product: 2**w* bits

Discard *w* bits: *w* bits



- Standard Multiplication Function
  - Ignores high order *w* bits
- Implements Modular Arithmetic

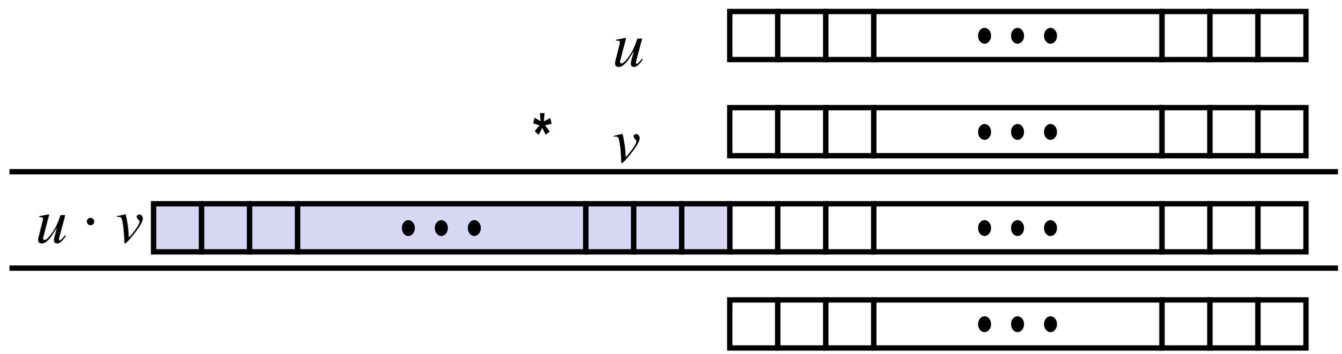
  $\text{UMult}_w(u, v) = u \cdot v \bmod 2^w$

Signed Multiplication in C

Operands: w bits

True Product: $2*w$ bits

Discard w bits: w bits



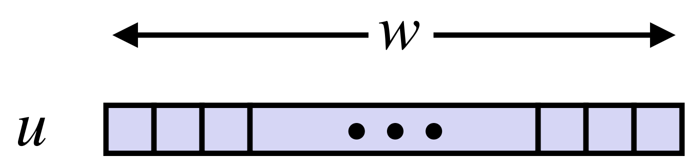
- Standard Multiplication Function
  - Ignores high order w bits
  - Some of which are different for signed vs. unsigned multiplication
  - Lower bits are the same

Power-of-2 Multiply with Shift

- Operation
  - `u << k` gives `u` * *2^k*
  - 001_2 << 2 = 100_2 (1 * 2^2 = 4)
  - Both signed and unsigned

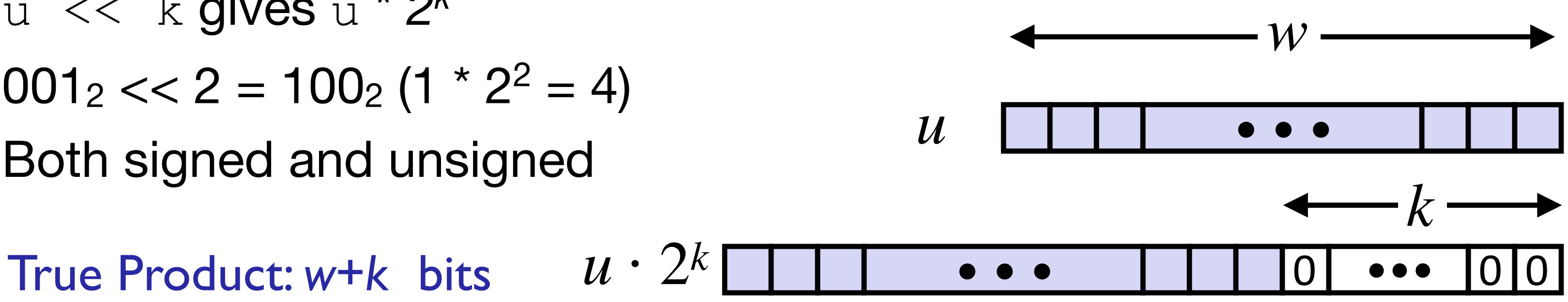
Power-of-2 Multiply with Shift

- Operation
  - `u << k` gives `u` * 2^k
  - $001_2 << 2 = 100_2$ ($1 * 2^2 = 4$)
  - Both signed and unsigned



Power-of-2 Multiply with Shift

- Operation
  - `u << k` gives `u` * 2^k
  - 001_2 << 2 = 100_2 ($1 * 2^2 = 4$)
  - Both signed and unsigned

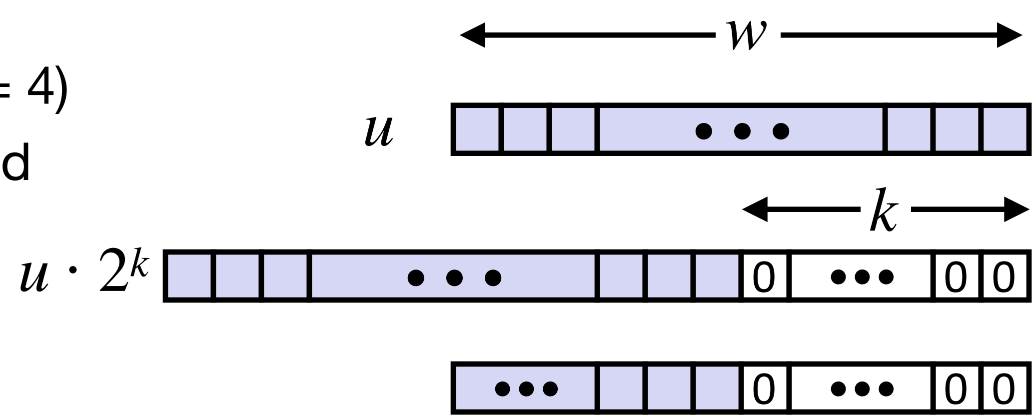


Power-of-2 Multiply with Shift

- Operation
  - `u << k` gives `u` * 2^k
  - $001_2 << 2 = 100_2$ ($1 * 2^2 = 4$)
  - Both signed and unsigned

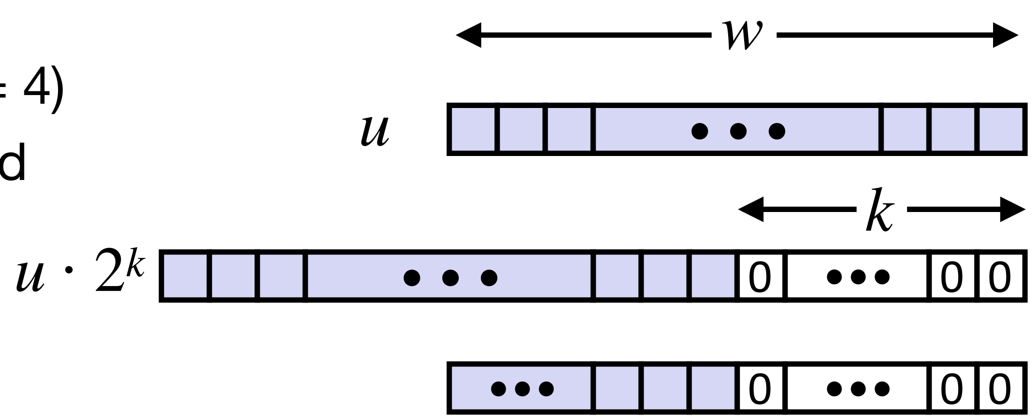
True Product: *w*+*k* bits

Discard *k* bits (if overflow)



Power-of-2 Multiply with Shift

- Operation
  - `u << k` gives `u` * 2^k
  - 001_2 << 2 = 100_2 (1 * 2^2 = 4)
  - Both signed and unsigned



True Product: *w*+*k* bits

Discard *k* bits (if overflow)

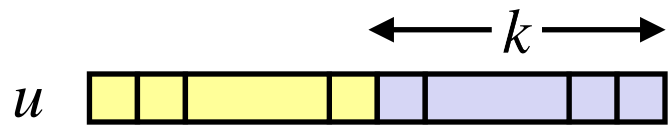
- Most machines shift and add faster than multiply
  - Compiler generates this code automatically
  - `u << 3 == u * 8`
  - `(u << 5) - (u << 3) ==     u * 24`

Unsigned Power-of-2 Divide with Shift

- Implement power-of-2 divide with shift
  - `u >> k` gives $\lfloor \texttt{u} / 2^k \rfloor$ ($\lfloor 2.34 \rfloor = 2$)
  - Uses logical shift

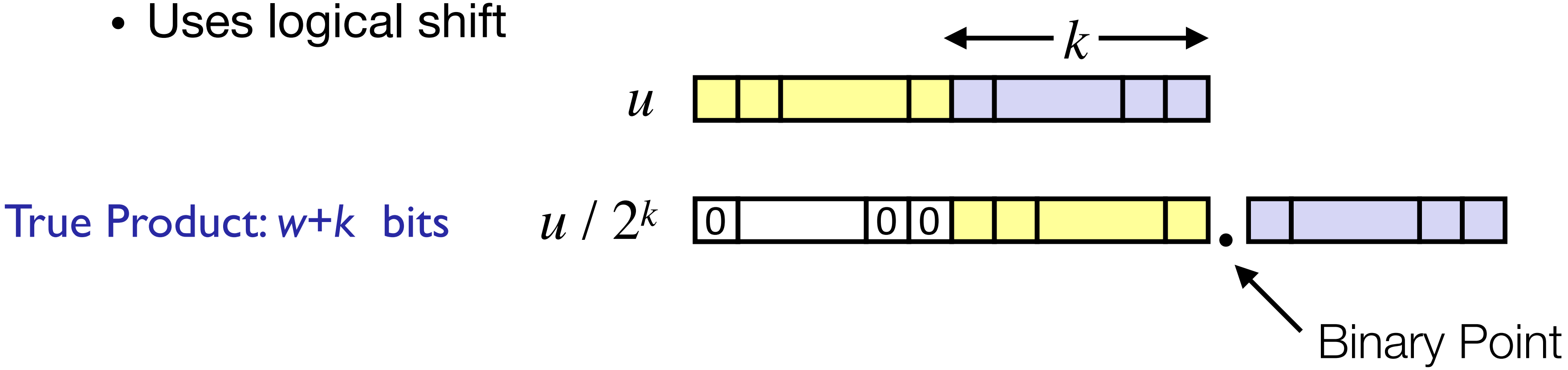
Unsigned Power-of-2 Divide with Shift

- Implement power-of-2 divide with shift
  - `u >> k` gives $\lfloor \texttt{u} / 2^k \rfloor$ ($\lfloor 2.34 \rfloor = 2$)
  - Uses logical shift



Unsigned Power-of-2 Divide with Shift

- Implement power-of-2 divide with shift
  - `u >> k` gives $\lfloor u / 2^k \rfloor$ ($\lfloor 2.34 \rfloor = 2$)
  - Uses logical shift

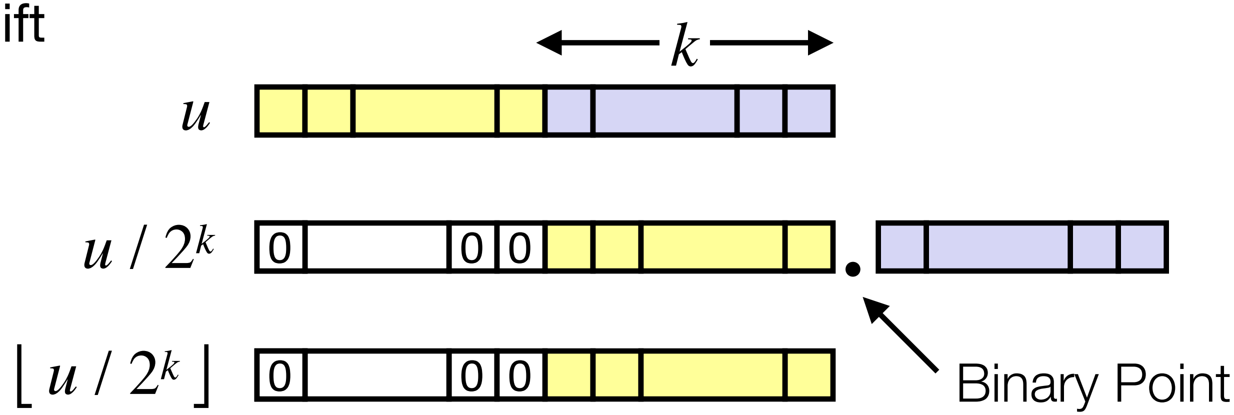


Unsigned Power-of-2 Divide with Shift

- Implement power-of-2 divide with shift
  - `u >> k` gives $\lfloor u / 2^k \rfloor$ ($\lfloor 2.34 \rfloor = 2$)
  - Uses logical shift

True Product: $w+k$ bits

Discard k bits after binary point

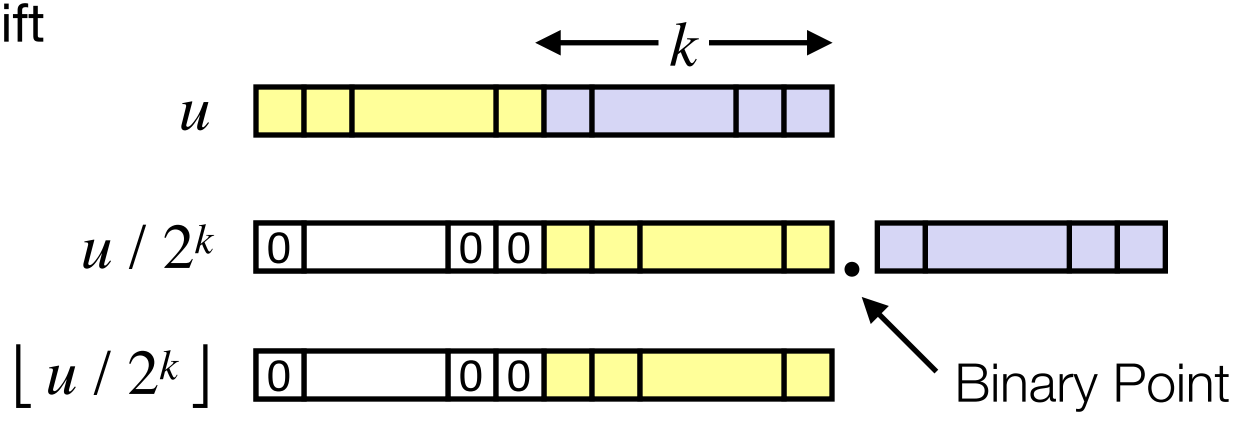


Unsigned Power-of-2 Divide with Shift

- Implement power-of-2 divide with shift
  - `u >> k` gives $\lfloor u / 2^k \rfloor$ ($\lfloor 2.34 \rfloor = 2$)
  - Uses logical shift

True Product: $w+k$ bits

Discard k bits after binary point

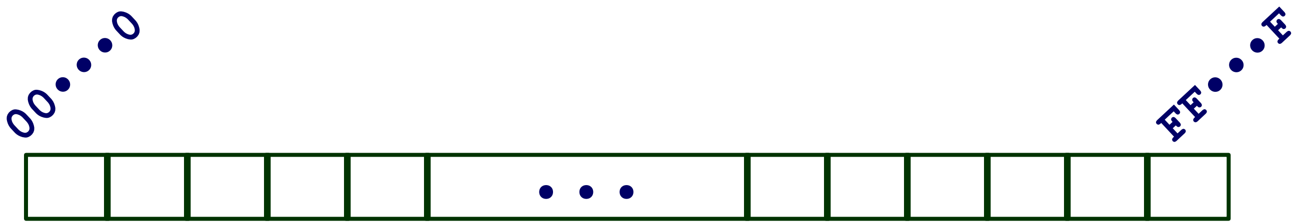


- 234_{10} >> 2 = 2.34_{10}, truncated result is 2 ($\lfloor 2.34 \rfloor = 2$)
- 1101_2 >> 2 = 0011_2 (true result: 11.01_2. $\lfloor 13 / 4 \rfloor = 3$)

Today: Representing Information in Binary

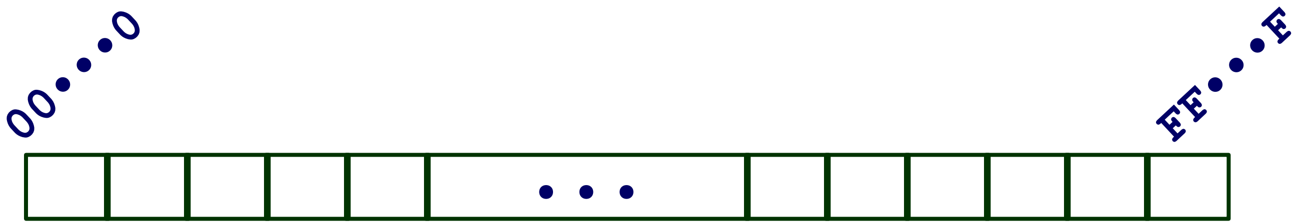
- Why Binary (bits)?
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
  - Summary
- Representations in memory, pointers, strings

Byte-Oriented Memory Organization



- Programs refer to data by address
  - Conceptually, envision it as a very large array of bytes: byte-addressable
    - In reality, it's not, but can think of it that way
  - An address is like an index into that array
    - and, a pointer variable stores an address

Machine Words

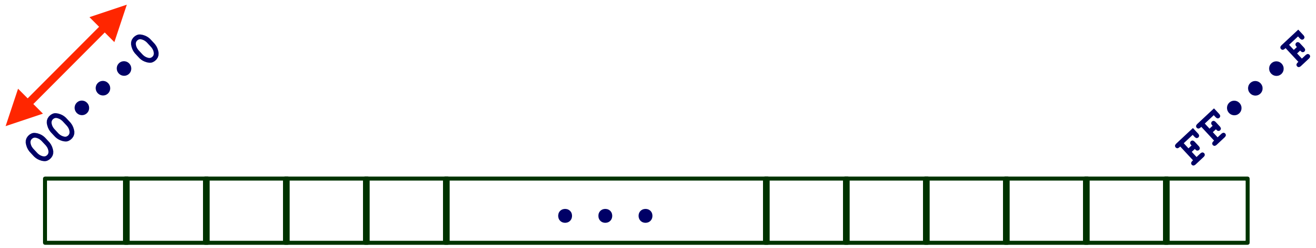


- Any given computer has a "Word Size"
    - Nominal size of a memory address

    - Until recently, most machines used 32 bits (4 bytes) as word size
        - Limits addresses to 4GB (2^{32} bytes)

- Increasingly, machines have 64-bit word size
    - Potentially, could have 18 EB (exabytes) of addressable memory
    - That's 18.4 X 10^{18}

Machine Words



- Any given computer has a "Word Size"
  - Nominal size of a memory address

  - Until recently, most machines used 32 bits (4 bytes) as word size
    - Limits addresses to 4GB (2^{32} bytes)

  - Increasingly, machines have 64-bit word size
    - Potentially, could have 18 EB (exabytes) of addressable memory
    - That's 18.4 X 10^{18}

Example Data Representations (in Bytes)

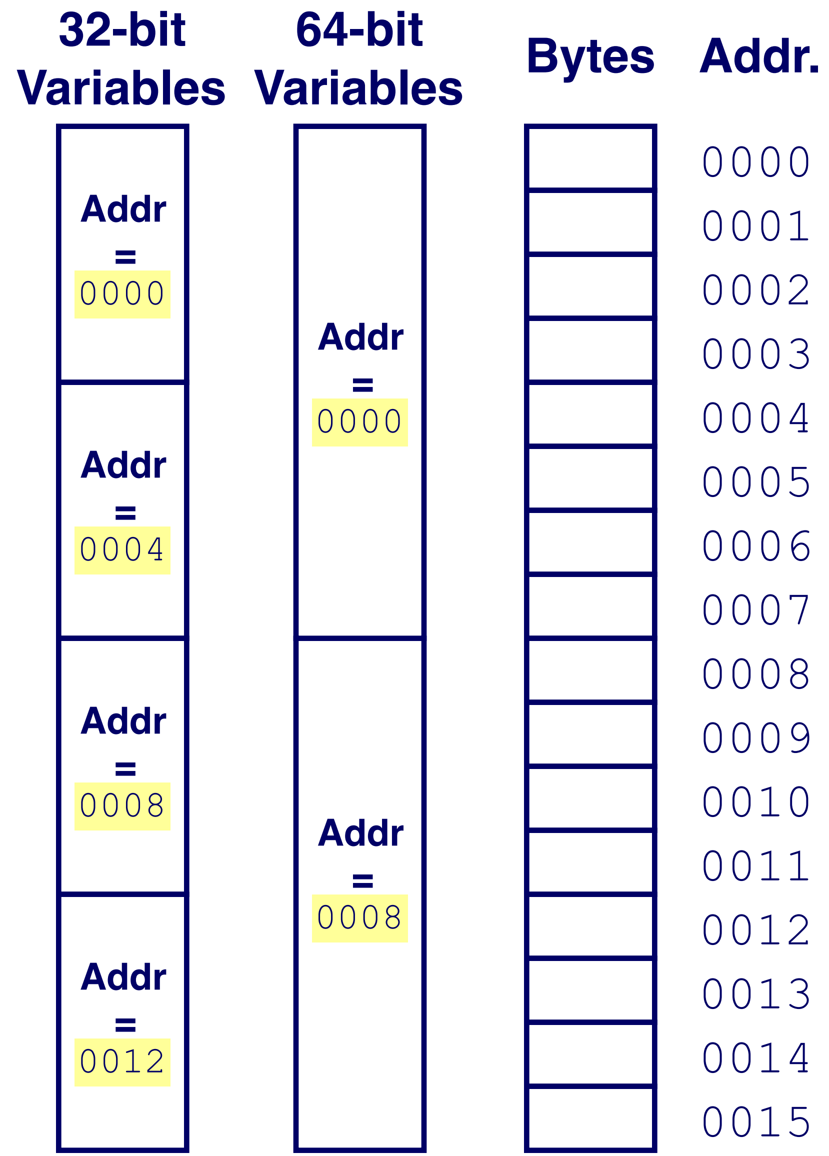
Word Size	4	8
C Data Type	32-bit	64-bit
`char`	1	1
`short`	2	2
`int`	4	4
`long`	4	8
`float`	4	4
`double`	8	8
`pointer`	4	8

Example Data Representations (in Bytes)

Word Size	4	8
C Data Type	32-bit	64-bit
`char`	1	1
`short`	2	2
`int`	4	4
`long`	4	8
`float`	4	4
`double`	8	8
`pointer`	4	8

Word-Oriented Memory Organization

- Addresses Specify Byte Locations
  - Address of first byte in word
  - Addresses of successive words differ by 4 (32-bit) or 8 (64-bit)



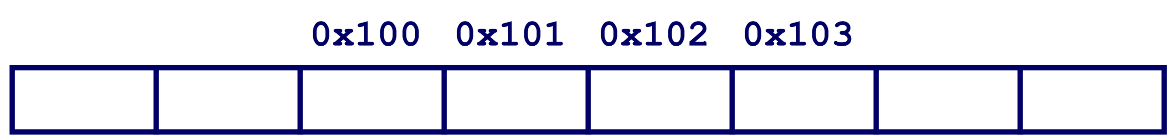
Byte Ordering

Byte Ordering

- How are the bytes within a multi-byte word ordered in memory?

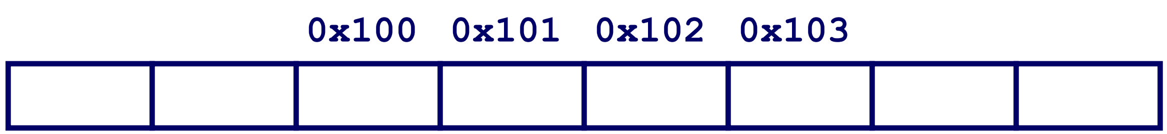
Byte Ordering

- How are the bytes within a multi-byte word ordered in memory?
- Example
  - Variable x has 4-byte value of 0x01234567
  - Address given by &x is 0x100



Byte Ordering

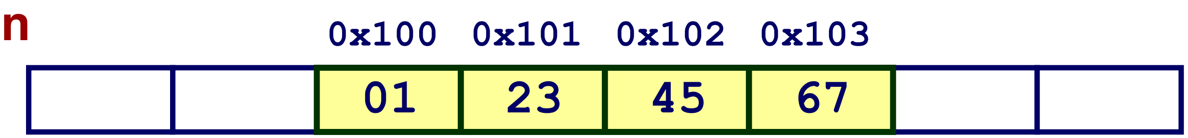
- How are the bytes within a multi-byte word ordered in memory?
- Example
  - Variable x has 4-byte value of 0x01234567
  - Address given by &x is 0x100
- Conventions
  - **Big Endian**: Sun, PPC Mac, IBM z, Internet
    - Most significant byte has lowest address (MSB first)
  - **Little Endian**: x86, ARM
    - Least significant byte has lowest address (LSB first)



Byte Ordering

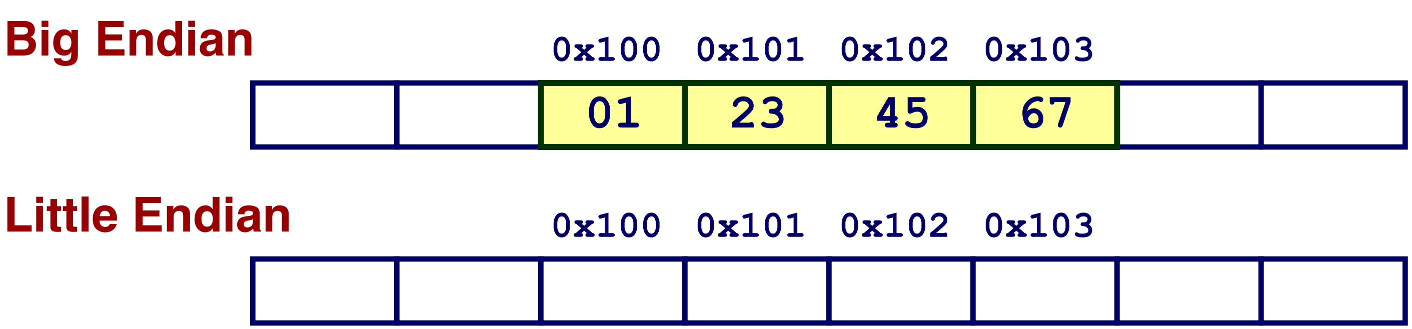
- How are the bytes within a multi-byte word ordered in memory?
- Example
  - Variable x has 4-byte value of 0x01234567
  - Address given by &x is 0x100
- Conventions
  - **Big Endian**: Sun, PPC Mac, IBM z, Internet
    - Most significant byte has lowest address (MSB first)
  - **Little Endian**: x86, ARM
    - Least significant byte has lowest address (LSB first)

Big Endian



Byte Ordering

- How are the bytes within a multi-byte word ordered in memory?
- Example
  - Variable x has 4-byte value of 0x01234567
  - Address given by &x is 0x100
- Conventions
  - **Big Endian**: Sun, PPC Mac, IBM z, Internet
    - Most significant byte has lowest address (MSB first)
  - **Little Endian**: x86, ARM
    - Least significant byte has lowest address (LSB first)



Byte Ordering

- How are the bytes within a multi-byte word ordered in memory?
- Example
  - Variable x has 4-byte value of 0x01234567
  - Address given by &x is 0x100
- Conventions
  - **Big Endian**: Sun, PPC Mac, IBM z, Internet
    - Most significant byte has lowest address (MSB first)
  - **Little Endian**: x86, ARM
    - Least significant byte has lowest address (LSB first)

Big Endian

		0x100	0x101	0x102	0x103		
		01	23	45	67		

Little Endian

		0x100	0x101	0x102	0x103		
		67	45	23	01		

Representing Integers

Hex: `00003B6D`

```
int A = 15213;
```

Address Increase

Little-E	Big-E
6D	00
3B	00
00	3B
00	6D

Hex: `FFFFC493`

```
int B = -15213;
```

Little-E	Big-E
93	FF
C4	FF
FF	C4
FF	93

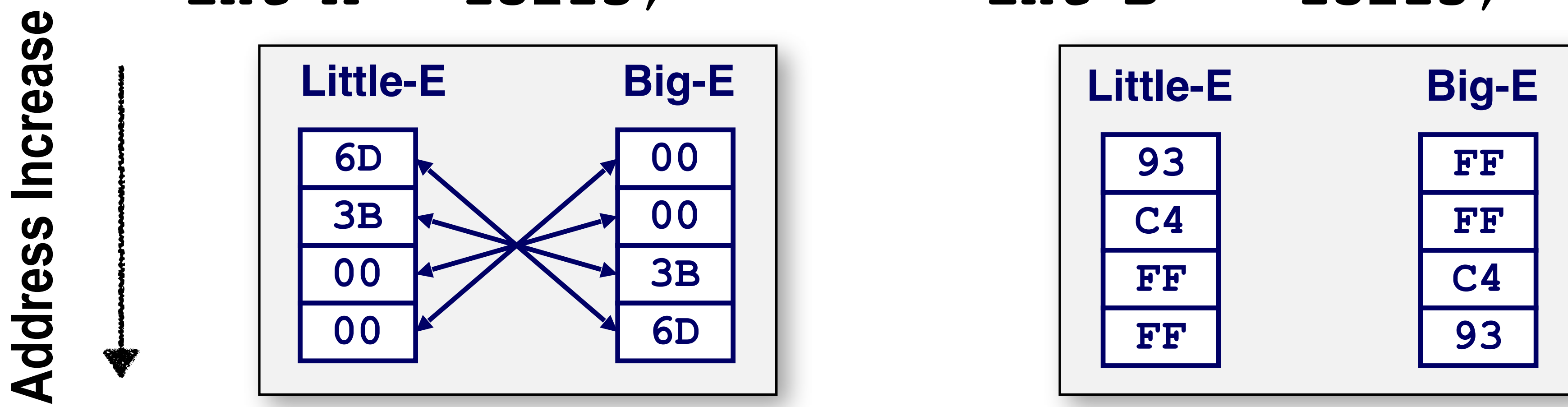
Representing Integers

Hex: `00003B6D`

Hex: `FFFFC493`

`int A = 15213;`

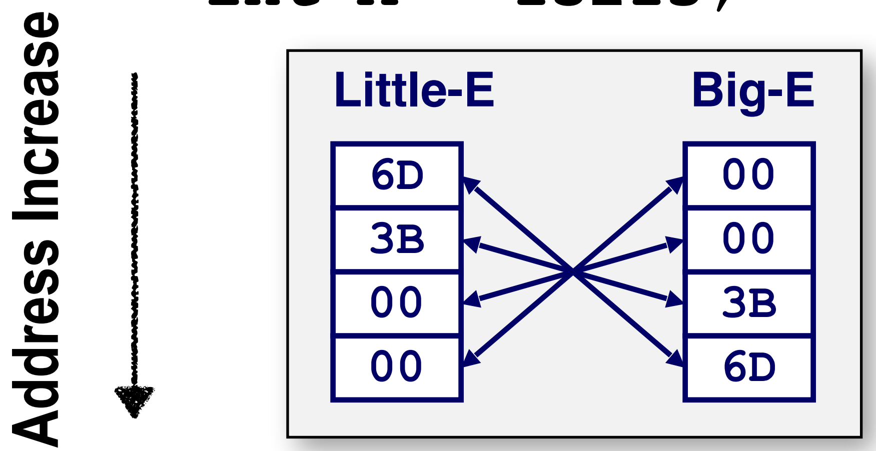
`int B = -15213;`



Representing Integers

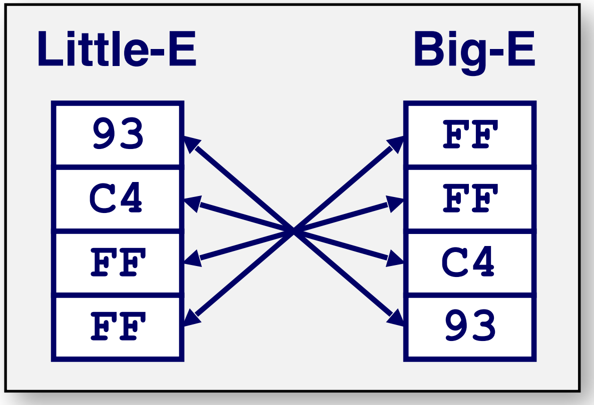
Hex: `00003B6D`

`int A = 15213;`



Hex: `FFFFC493`

`int B = -15213;`



Announcement

- Check the course website before asking
  - http://www.cs.rochester.edu/courses/252/spring2019/
- Direct ALL questions regarding assignments to the TAs
  - They have done them. They have debugged them. They know them inside out.
  - If one doesn’t know, ask another.
  - If all don’t know, ask me.