

CSC 252: Computer Organization

Spring 2020: 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/spring2020/labs/assignment1.html>
 - Due on Jan. 31, 11:59 PM
 - You have 3 slip days

19	20	21	22	23	24	25
				Today		
26	27	28	29	30	31	Feb 1
					Due	

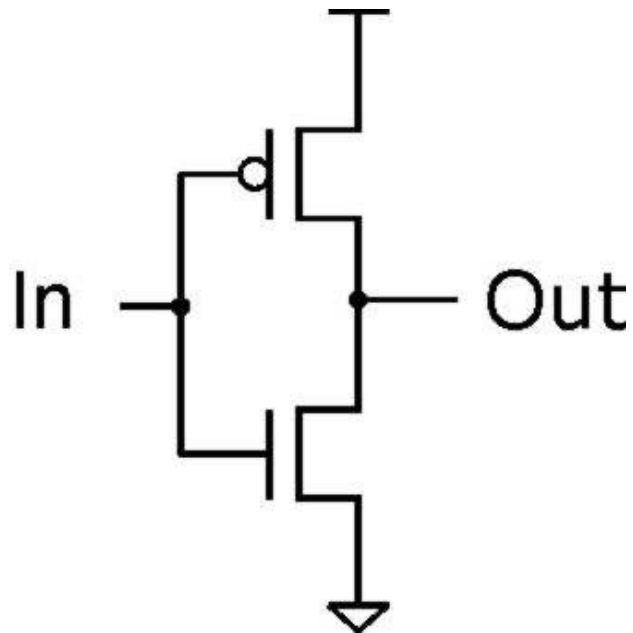
Announcement

- 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.

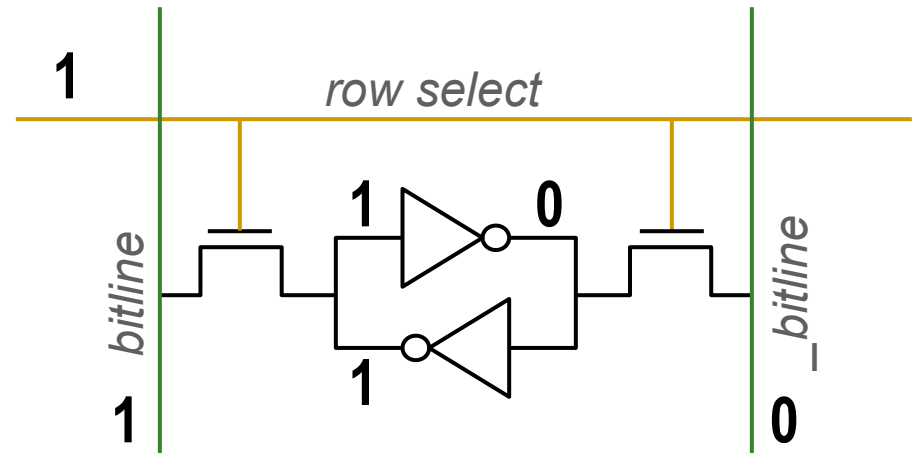
Previously in 252...

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

Transistors



Computation



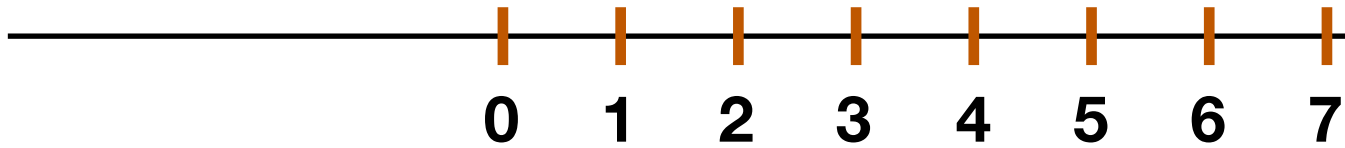
Store/Access Data

Encoding Negative Numbers

- Two's Complement

Encoding Negative Numbers

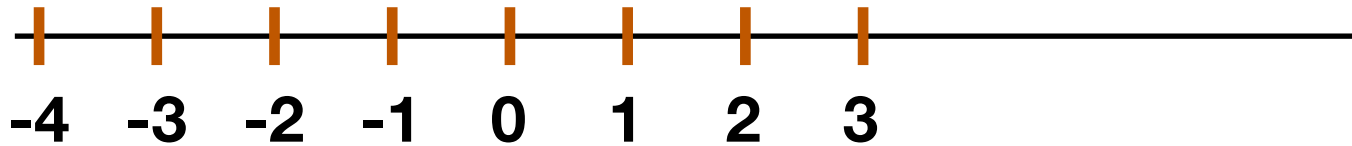
- Two's Complement



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

Encoding Negative Numbers

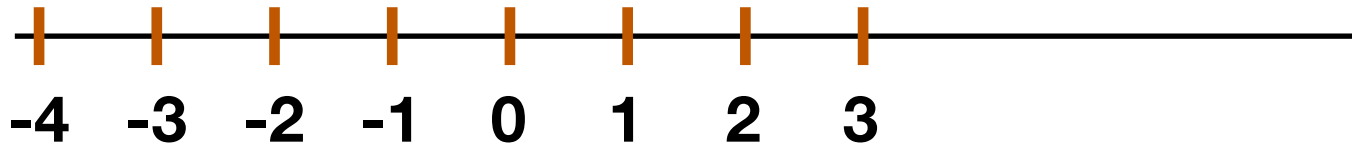
- Two's Complement



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

Encoding Negative Numbers

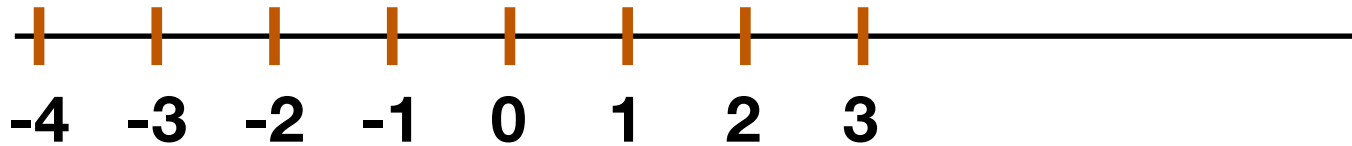
- Two's Complement



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



Weights in
Unsigned

$b_2 b_1 b_0$

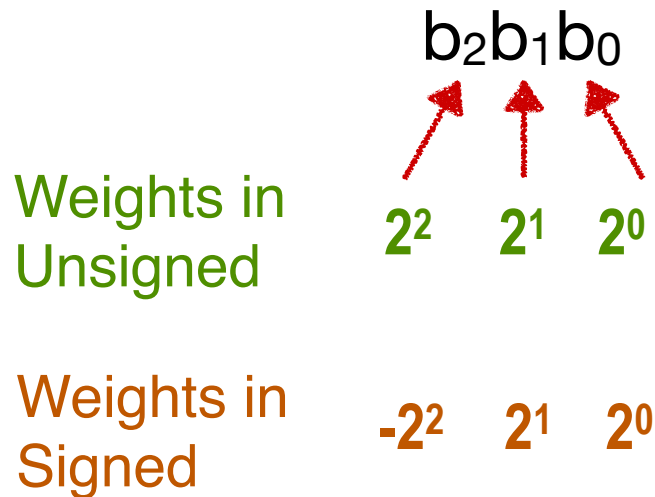
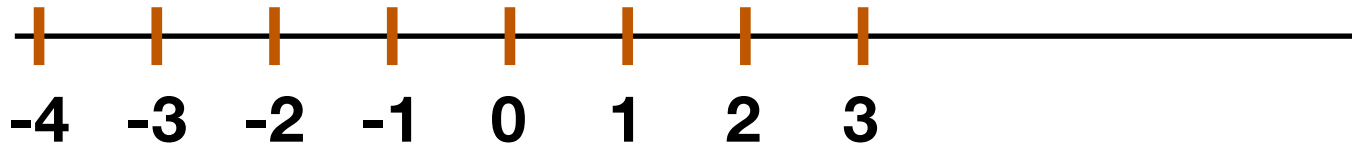
2^2 2^1 2^0

Three red arrows point from the labels 2^2 , 2^1 , and 2^0 to the bits b_2 , b_1 , and b_0 respectively.

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

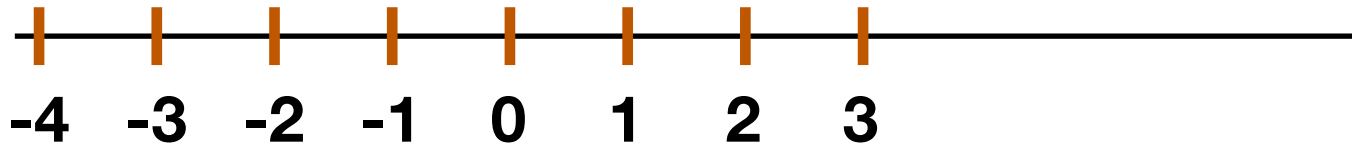
- Two's Complement



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



Weights in
Unsigned

$b_2 b_1 b_0$

$2^2 \quad 2^1 \quad 2^0$

Weights in
Signed

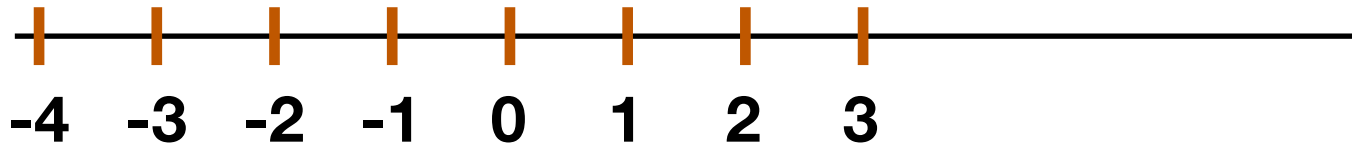
$-2^2 \quad 2^1 \quad 2^0$

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

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



Weights in Unsigned

$b_2 b_1 b_0$

$2^2 \quad 2^1 \quad 2^0$

Red arrows point from the labels 2^2 , 2^1 , and 2^0 to the bits b_2 , b_1 , and b_0 respectively.

Weights in Signed

$-2^2 \quad 2^1 \quad 2^0$

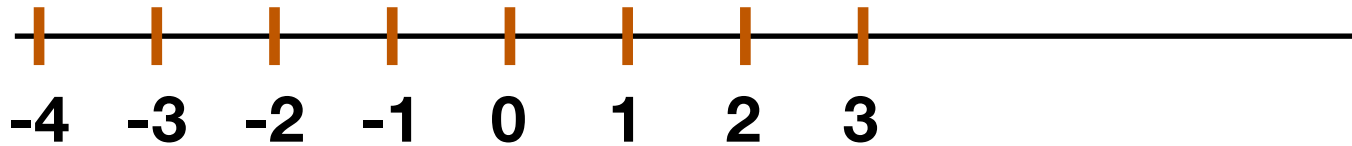
$$101_2 = 1 \cdot 2^0 + 0 \cdot 2^1 + (-1 \cdot 2^2) = -3_{10}$$

A red arrow points to the '1' in the 101_2 term, and a brown box highlights the $1 \cdot 2^0$ term in the equation.

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



Weights in Unsigned

$b_2 b_1 b_0$

$2^2 \quad 2^1 \quad 2^0$

Weights in Signed

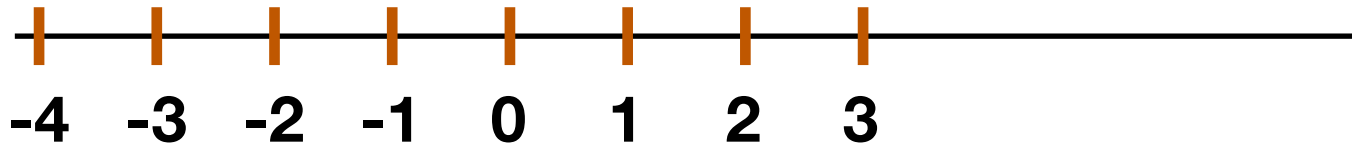
$-2^2 \quad 2^1 \quad 2^0$

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

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



Weights in Unsigned

$b_2 b_1 b_0$

$2^2 \quad 2^1 \quad 2^0$

Diagram showing bit weights for unsigned and signed representations. Red arrows point from the bit labels b_2, b_1, b_0 to the weight values $2^2, 2^1, 2^0$ for unsigned and $-2^2, 2^1, 2^0$ for signed.

Weights in Signed

$-2^2 \quad 2^1 \quad 2^0$

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

A red arrow points to the first bit (1) of the binary number 101 in the equation above.

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

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

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

Two-Complement Implications

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

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

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

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

Two-Complement Implications

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

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

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

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

- 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 char** 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 signed data types (e.g., **int**, **short**, etc.) are for

Data Types (in C)

- What if you want to define a variable that could take negative values?
 - That's what signed data types (e.g., **int**, **short**, etc.) are 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 signed data types (e.g., **int**, **short**, etc.) are 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 signed data types (e.g., **int**, **short**, etc.) are 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 signed data types (e.g., **int**, **short**, etc.) are 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

```
int x = -5, y = 4;  
int z = x + y;  
fprintf(stdout, "%d\n", z);  
fprintf(stdout, "%u\n", z);
```

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;

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 \leftrightarrow 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 \leftrightarrow Unsigned

Bits	Signed		Unsigned
0000	0	→ T2U →	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 \leftrightarrow Unsigned


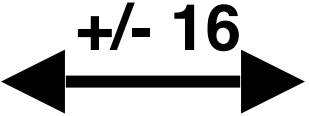
Bits	Signed		Unsigned
0000	0	\rightarrow T2U \rightarrow	0
0001	1		1
0010	2		2
0011	3		3
0100	4		4
0101	5		5
0110	6		6
0111	7		7
1000	-8	\leftarrow U2T \leftarrow	8
1001	-7		9
1010	-6		10
1011	-5		11
1100	-4		12
1101	-3		13
1110	-2		14
1111	-1		15

Mapping Signed \leftrightarrow 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

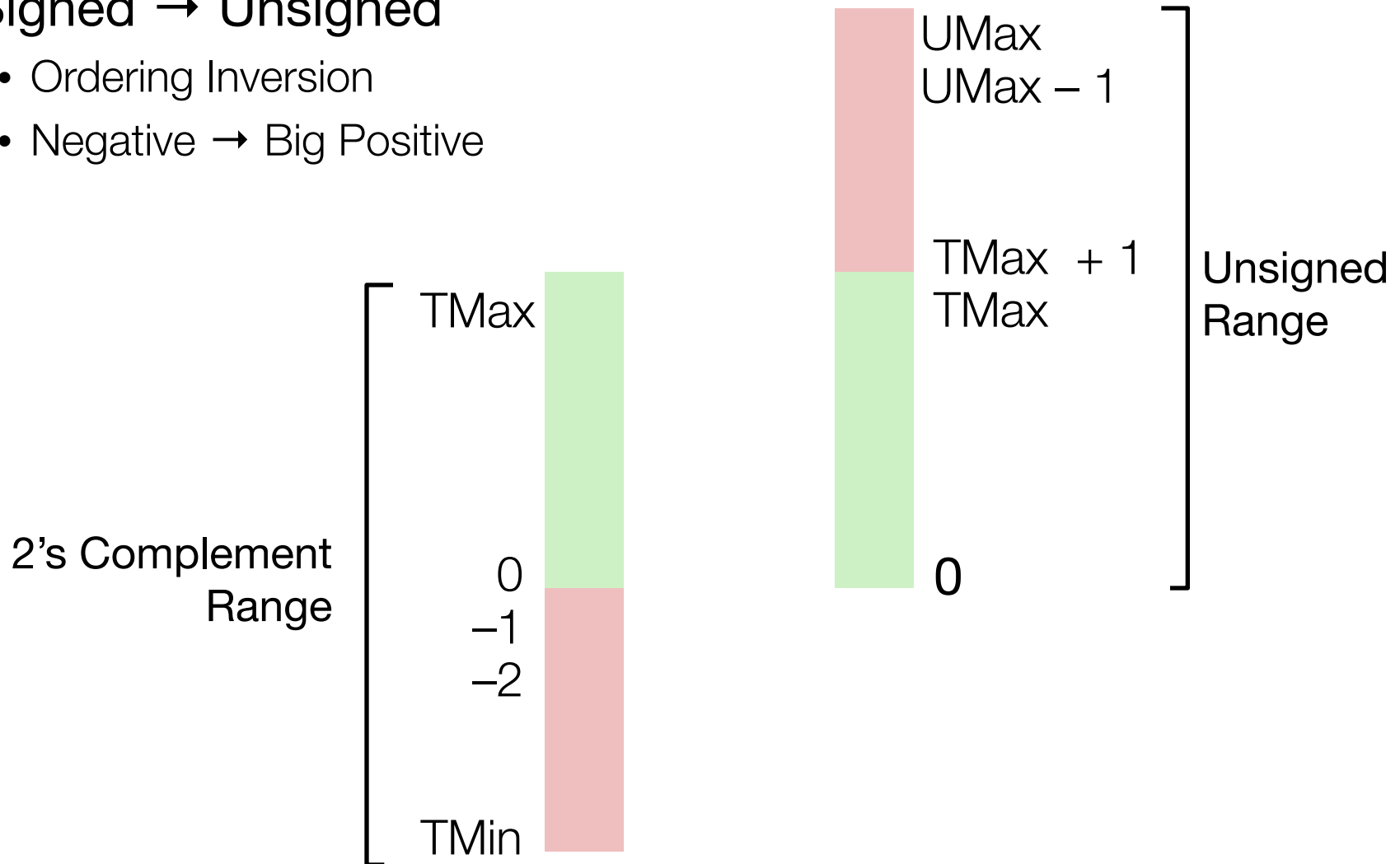


Mapping Signed \leftrightarrow Unsigned

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

Conversion Visualized

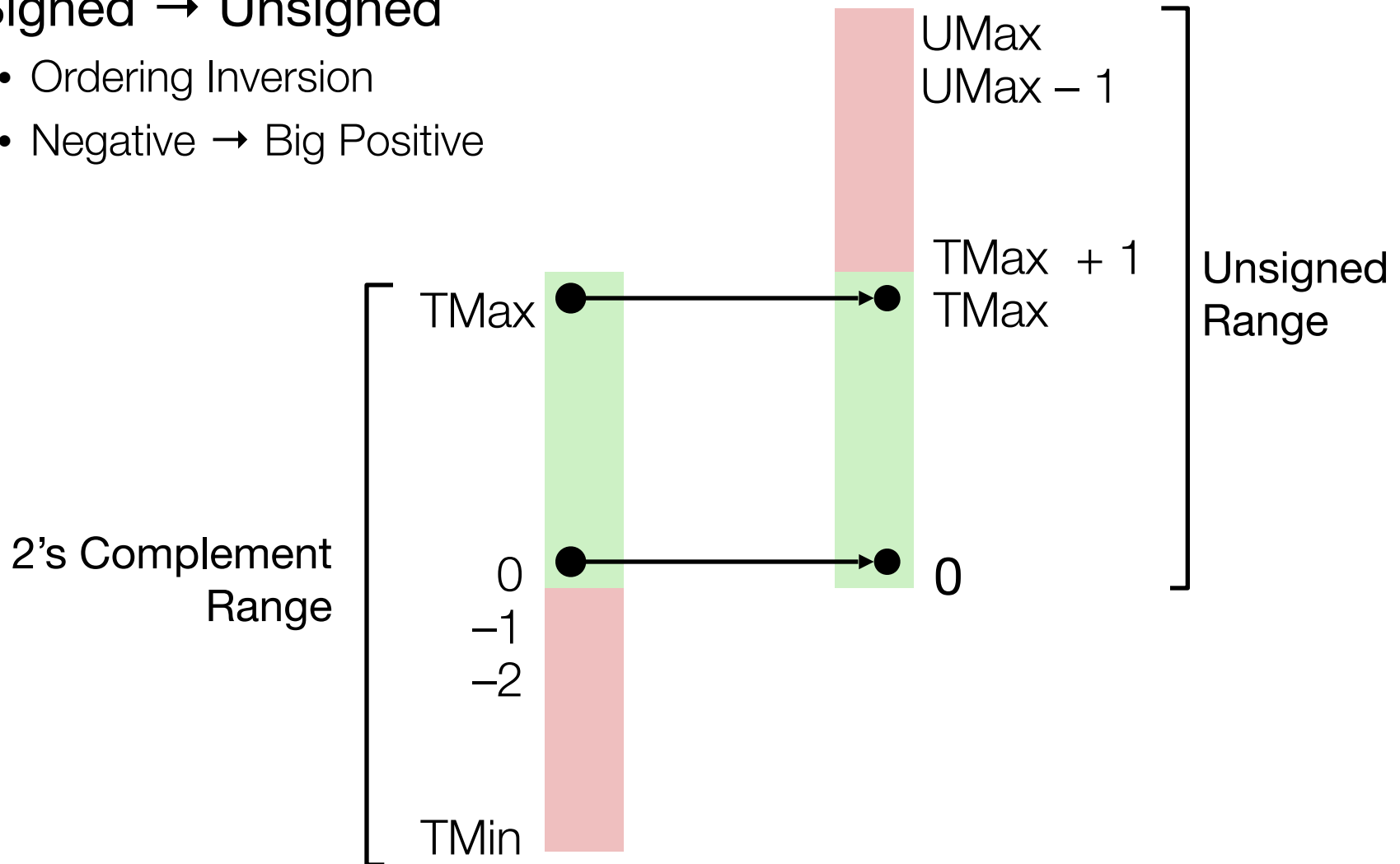
- Signed \rightarrow Unsigned
 - Ordering Inversion
 - Negative \rightarrow Big Positive



Conversion Visualized

- Signed \rightarrow Unsigned

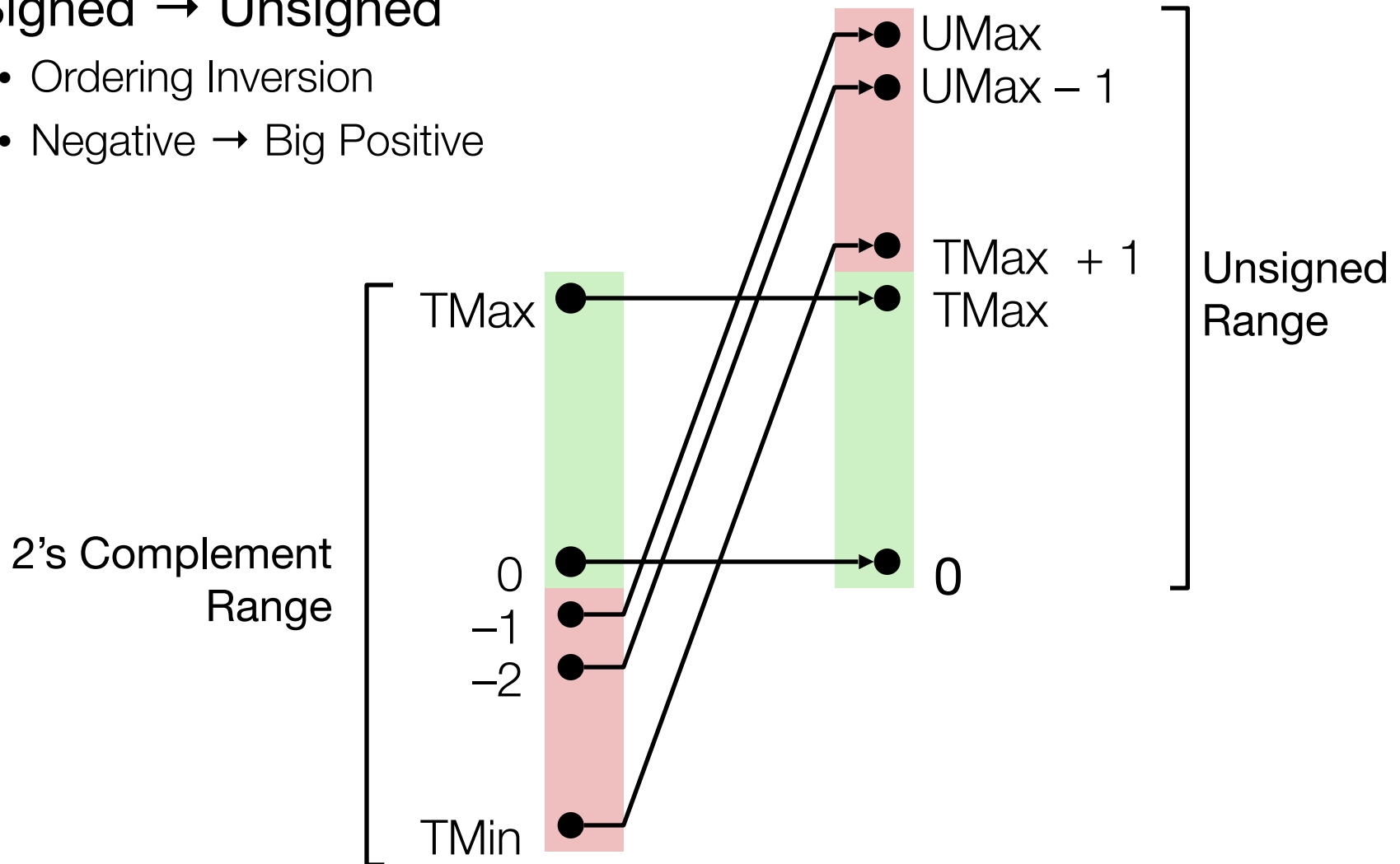
- Ordering Inversion
- Negative \rightarrow Big Positive



Conversion Visualized

- Signed \rightarrow Unsigned

- Ordering Inversion
- Negative \rightarrow 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;
```

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

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 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:
- $X' = \underbrace{X_{w-1}, \dots, X_{w-1}}_{k \text{ copies of MSB}}, X_{w-1}, X_{w-2}, \dots, X_0$

k copies of MSB

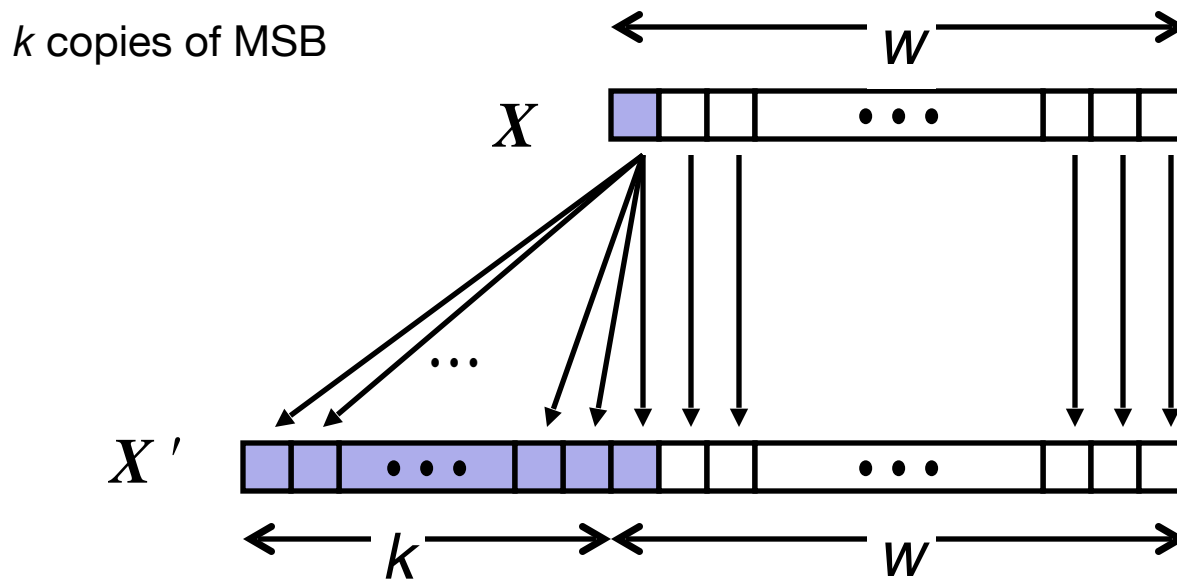
Signed Extension

- Task:

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

- Rule:

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



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

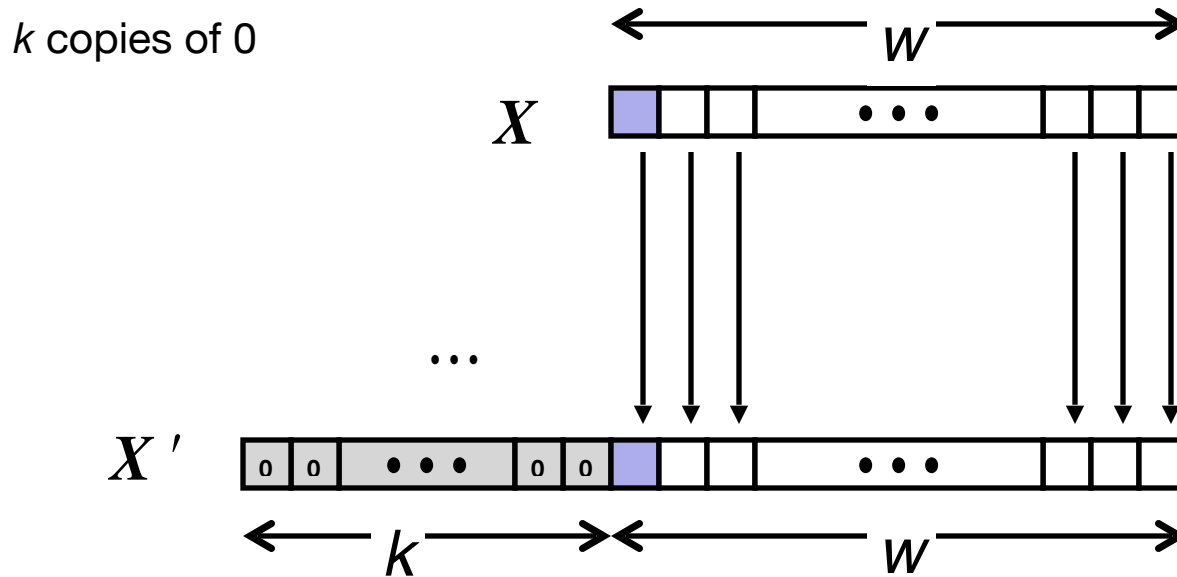
Unsigned (Zero) Extension

- Task:

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

- Rule:

- Simply pad zeros:
- $X' = \underbrace{0, \dots, 0}_{k \text{ copies of } 0}, x_{w-1}, x_{w-2}, \dots, x_0$



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

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)

Normal
Case

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

$$\begin{array}{r} 2 \\ +) 5 \\ \hline 7 \end{array}$$

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)
- Might **overflow**: result can't be represented within the size of the data type

Normal
Case

$$\begin{array}{r} 010 \\ +) 101 \\ \hline 111 \end{array} \qquad \begin{array}{r} 2 \\ +) 5 \\ \hline 7 \end{array}$$

Overflow
Case

$$\begin{array}{r} 110 \\ +) 101 \\ \hline 1011 \end{array} \qquad \begin{array}{r} 6 \\ +) 5 \\ \hline 11 \end{array}$$

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)
- 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

$$\begin{array}{r} 010 \\ +) 101 \\ \hline 111 \end{array} \qquad \begin{array}{r} 2 \\ +) 5 \\ \hline 7 \end{array}$$

Overflow
Case

$$\begin{array}{r} 110 \\ +) 101 \\ \hline 1011 \end{array} \qquad \begin{array}{r} 6 \\ +) 5 \\ \hline 11 \end{array}$$

← 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

$$\begin{array}{r} 010 \\ +) 101 \\ \hline 111 \end{array} \qquad \begin{array}{r} 2 \\ +) 5 \\ \hline 7 \end{array}$$

Overflow
Case

$$\begin{array}{r} 110 \\ +) 101 \\ \hline 1011 \\ 011 \end{array} \qquad \begin{array}{r} 6 \\ +) 5 \\ \hline 11 \\ 3 \end{array}$$



True Sum



Sum with same bits

Unsigned Addition in C

Operands: w bits

u

				...			
--	--	--	--	-----	--	--	--

$+ v$

				...			
--	--	--	--	-----	--	--	--

True Sum: $w+1$ bits

$u + v$

				...			
--	--	--	--	-----	--	--	--

Discard Carry: w bits

$\text{UAdd}_w(u, v)$

				...			
--	--	--	--	-----	--	--	--

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

$$\begin{array}{r} 010 \\ +) 101 \\ \hline 111 \end{array} \qquad \begin{array}{r} 2 \\ +) -3 \\ \hline -1 \end{array}$$

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

Two's Complement Addition

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

Normal Case

$$\begin{array}{r} 010 \\ +) 101 \\ \hline 111 \end{array}$$
$$\begin{array}{r} 2 \\ +) -3 \\ \hline -1 \end{array}$$

Overflow Case

$$\begin{array}{r} 110 \\ +) 101 \\ \hline 1011 \end{array}$$
$$\begin{array}{r} -2 \\ +) -3 \\ \hline -5 \end{array}$$

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

Normal Case

$$\begin{array}{r} 010 \\ +) 101 \\ \hline 111 \end{array}$$
$$\begin{array}{r} 2 \\ +) -3 \\ \hline -1 \end{array}$$

Overflow Case

$$\begin{array}{r} 110 \\ +) 101 \\ \hline 1011 \\ 011 \end{array}$$
$$\begin{array}{r} -2 \\ +) -3 \\ \hline -5 \\ 3 \end{array}$$

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

Normal Case

$$\begin{array}{r}
 010 \\
 +) 101 \\
 \hline
 111
 \end{array}
 \qquad
 \begin{array}{r}
 2 \\
 +) -3 \\
 \hline
 -1
 \end{array}$$

Overflow Case

$$\begin{array}{r}
 110 \\
 +) 101 \\
 \hline
 1011 \\
 011
 \end{array}
 \qquad
 \begin{array}{r}
 -2 \\
 +) -3 \\
 \hline
 -5 \\
 3
 \end{array}$$

Negative Overflow

Min →

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

Normal Case

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

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

Overflow Case

$$\begin{array}{r} 110 \\ +) 101 \\ \hline 1011 \\ 011 \end{array}$$

$$\begin{array}{r} -2 \\ +) -3 \\ \hline -5 \\ 3 \end{array}$$

Min →

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

$$\begin{array}{r} 011 \\ +) 001 \\ \hline 0100 \end{array}$$

$$\begin{array}{r} 3 \\ +) 1 \\ \hline 4 \end{array}$$

Negative Overflow

Two's Complement Addition

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

Normal Case

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

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

Min →

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

Overflow Case

$$\begin{array}{r} 110 \\ +) 101 \\ \hline 1011 \\ 011 \end{array}$$

$$\begin{array}{r} -2 \\ +) -3 \\ \hline -5 \\ 3 \end{array}$$

$$\begin{array}{r} 011 \\ +) 001 \\ \hline 0100 \\ 100 \end{array}$$

$$\begin{array}{r} 3 \\ +) 1 \\ \hline 4 \\ -4 \end{array}$$

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

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

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

Overflow Case

$$\begin{array}{r} 110 \\ +) 101 \\ \hline 1011 \\ 011 \end{array}$$

$$\begin{array}{r} -2 \\ +) -3 \\ \hline -5 \\ 3 \end{array}$$

$$\begin{array}{r} 011 \\ +) 001 \\ \hline 0100 \\ 100 \end{array}$$

$$\begin{array}{r} 3 \\ +) 1 \\ \hline 4 \\ -4 \end{array}$$

Negative Overflow

Positive Overflow

Two's Complement Addition in C

Operands: w bits

u



$+$ v



True Sum: $w+1$ bits

$u + v$



Discard Carry: w bits

$\text{TAdd}_w(u, v)$



Is This Signed Addition an Overflow?

$$\begin{array}{r} 111 \\ +) 110 \\ \hline 1101 \end{array}$$

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

Is This Signed Addition an Overflow?

$$\begin{array}{r} 111 \\ +) 110 \\ \hline \boxed{1}101 \end{array}$$

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

Is This Signed Addition an Overflow?

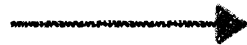
$$\begin{array}{r} 111 \\ +) 110 \\ \hline \boxed{1}101 \end{array}$$

→
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?

$$\begin{array}{r} 111 \\ +) 110 \\ \hline \boxed{1}101 \end{array}$$



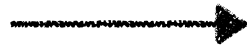
Truncate

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

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

Is This Signed Addition an Overflow?

$$\begin{array}{r} 111 \\ +) 110 \\ \hline \boxed{1}101 \end{array}$$



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?

$$\begin{array}{r} 111 \\ +) 110 \\ \hline \boxed{1}101 \end{array}$$



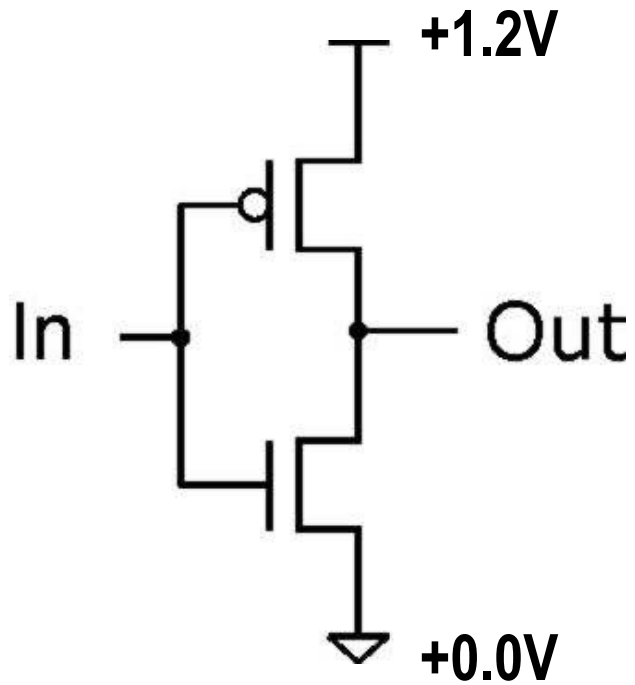
Truncate

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

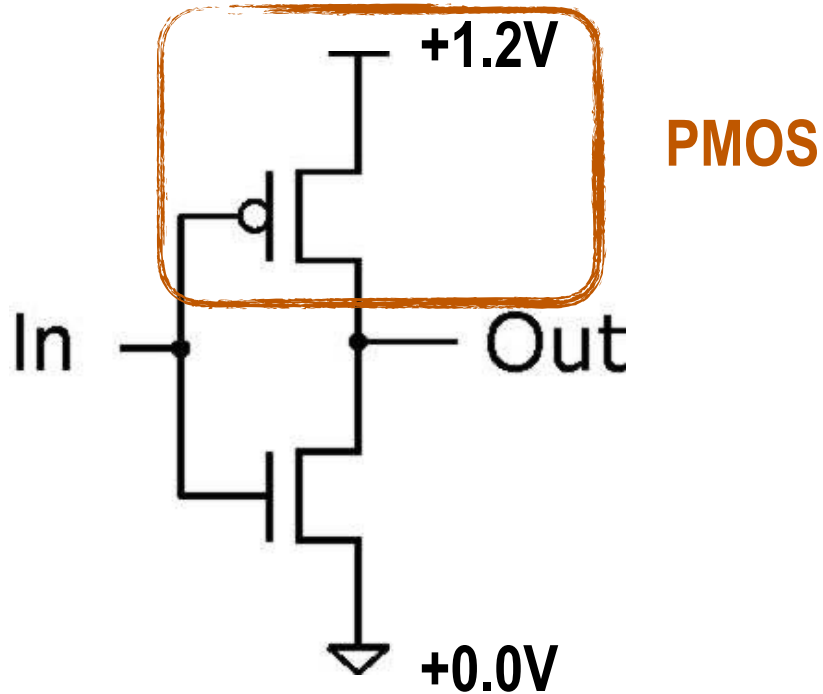
Signed	Binary
0	000
1	001
2	010
3	011
-4	100
-3	101
-2	110
-1	111

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

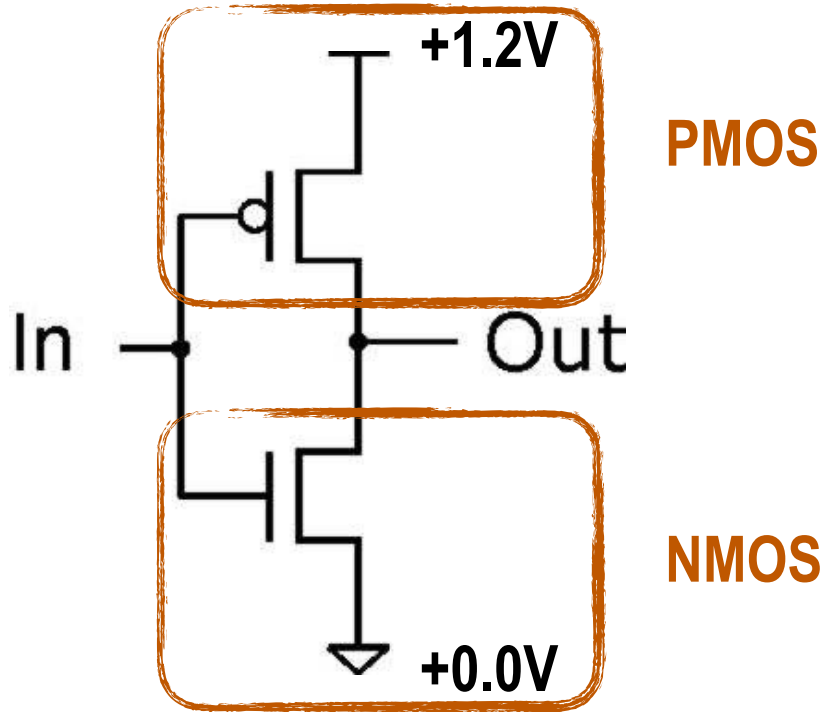
Inverter (NOT Gate)



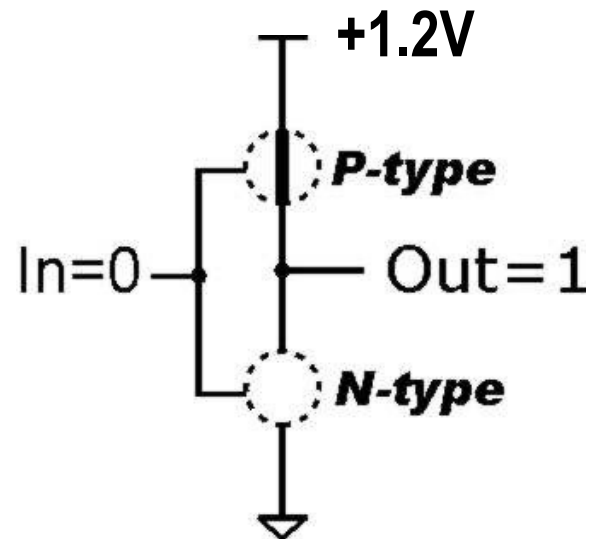
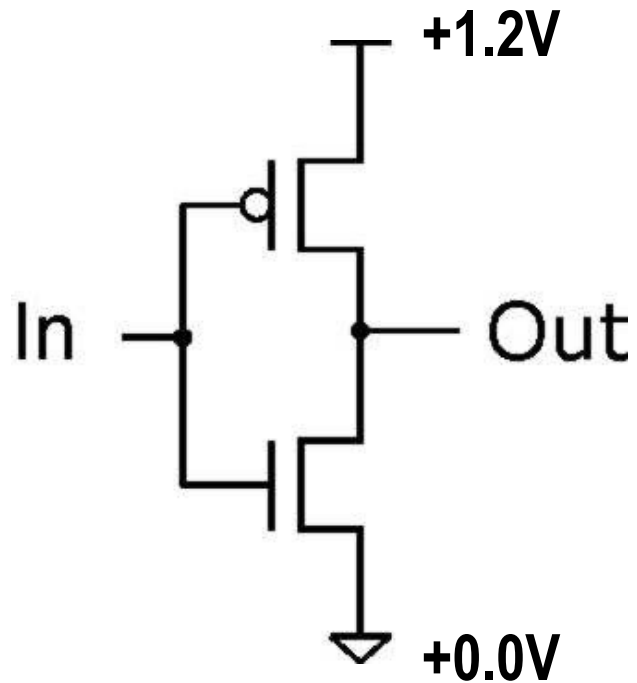
Inverter (NOT Gate)



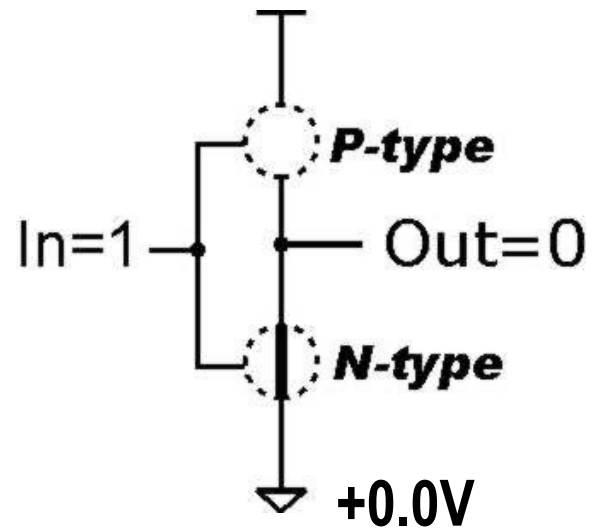
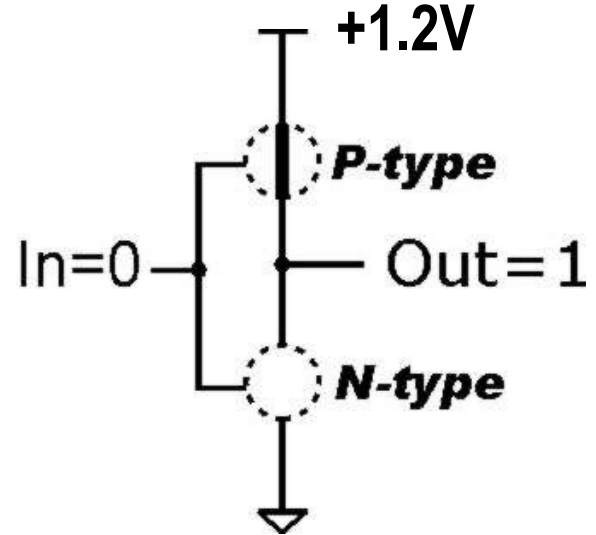
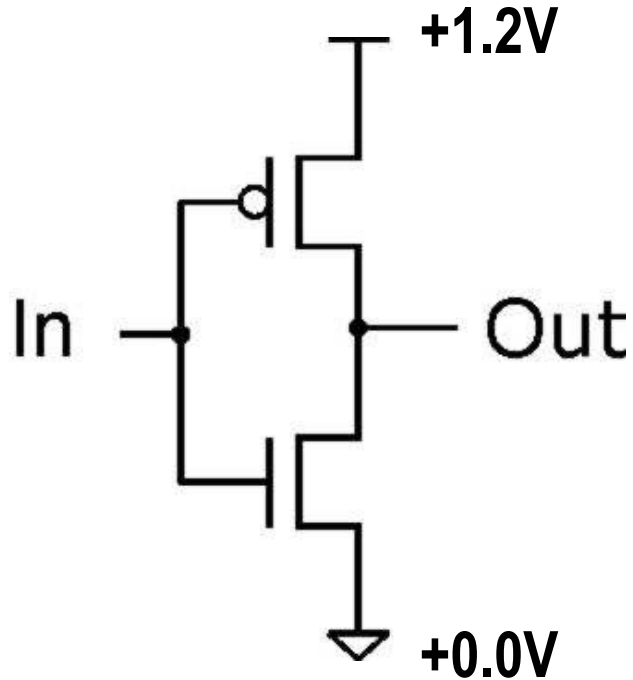
Inverter (NOT Gate)



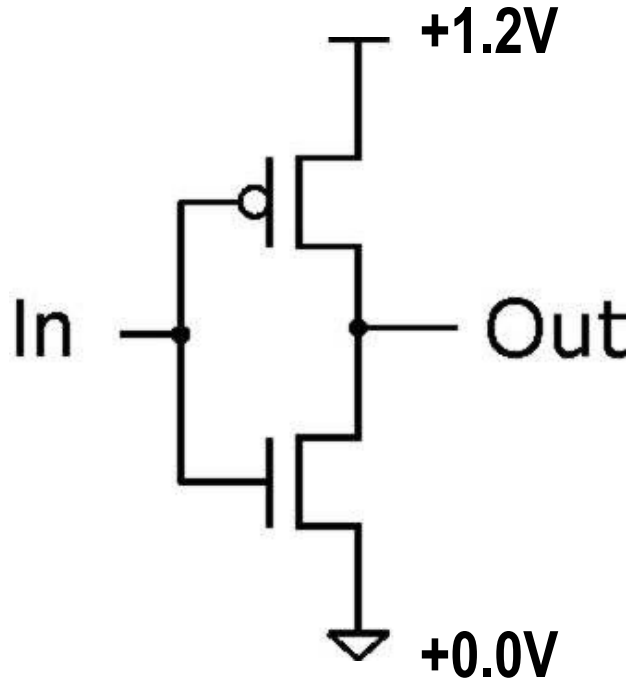
Inverter (NOT Gate)



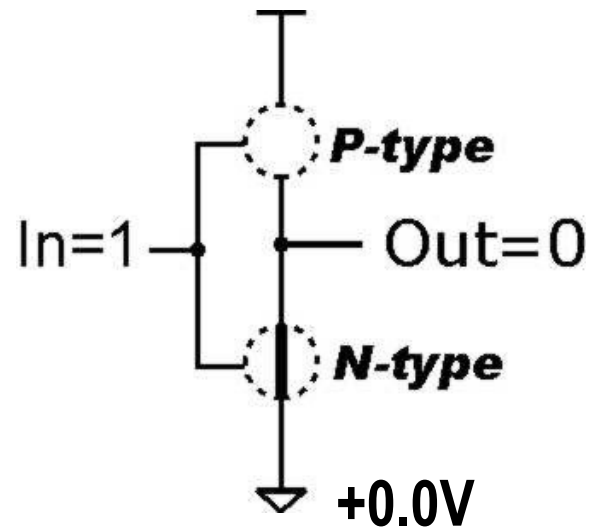
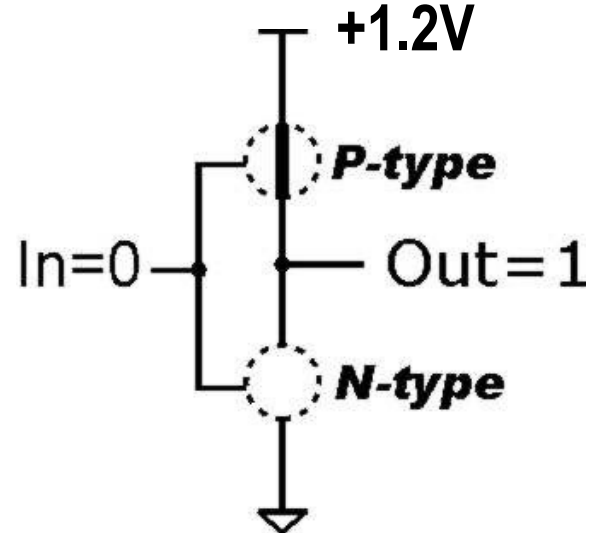
Inverter (NOT Gate)



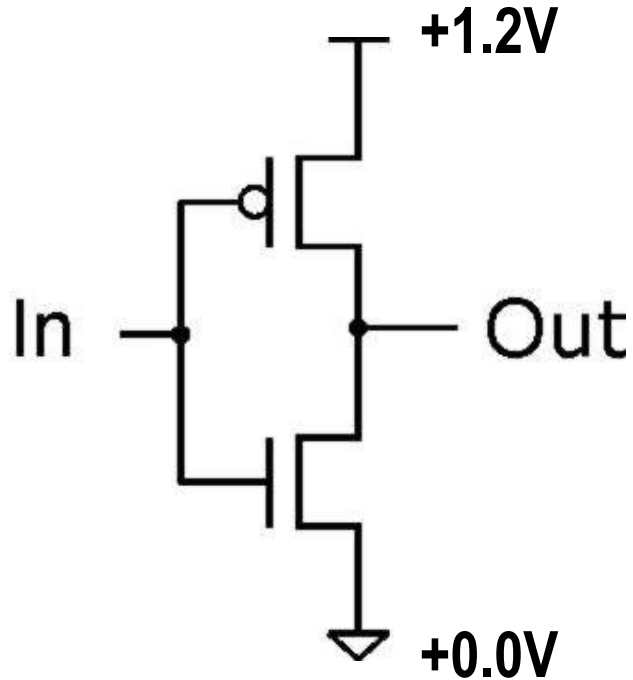
Inverter (NOT Gate)



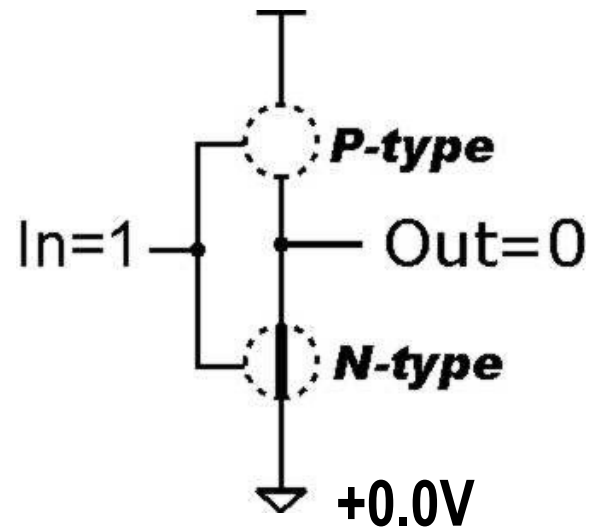
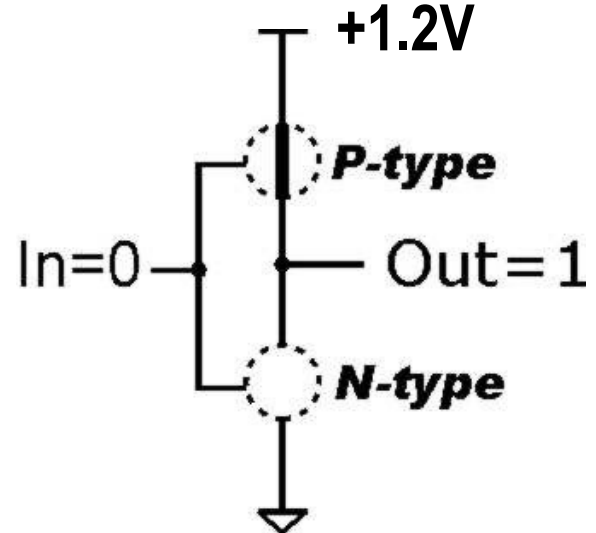
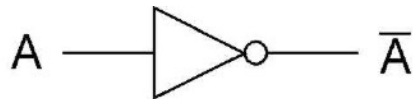
In	Out
0	1
1	0



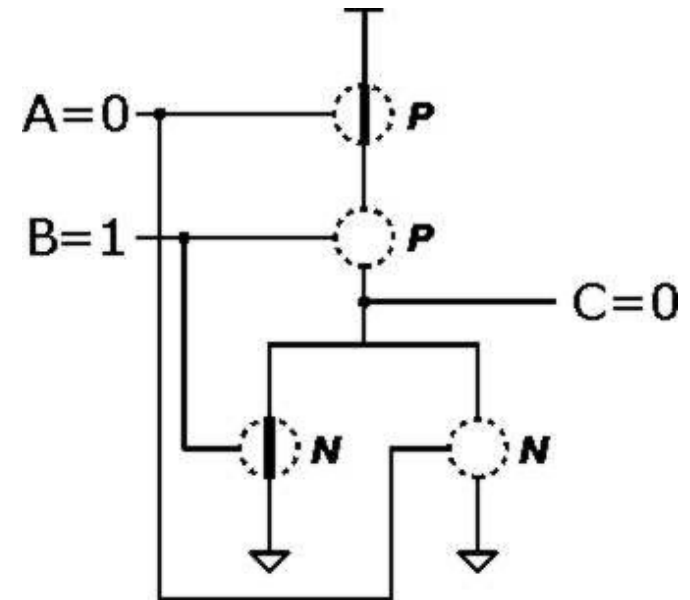
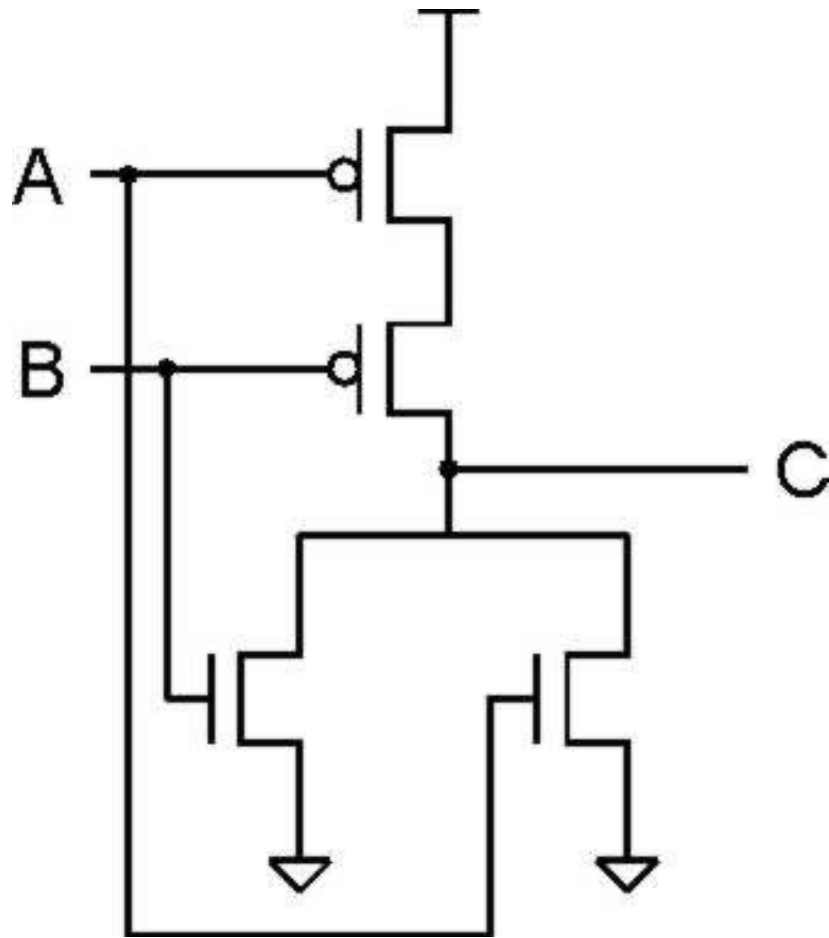
Inverter (NOT Gate)



In	Out
0	1
1	0



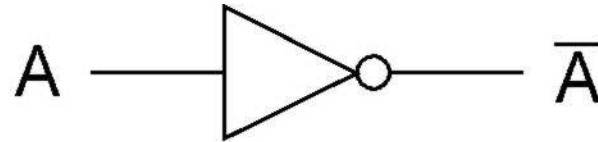
NOR Gate (NOT + OR)



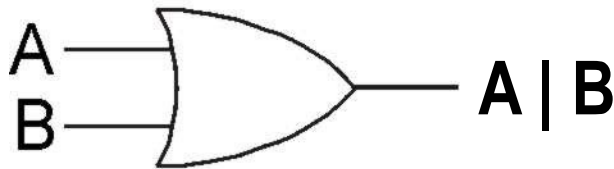
A	B	C
0	0	1
0	1	0
1	0	0
1	1	0

Note: Serial structure on top, parallel on bottom.

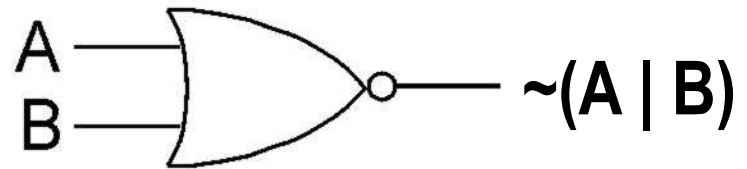
Basic Logic Gates



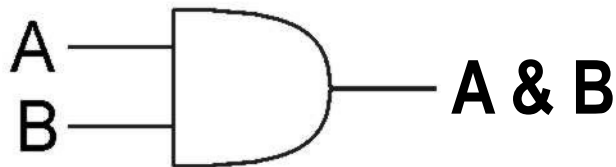
NOT



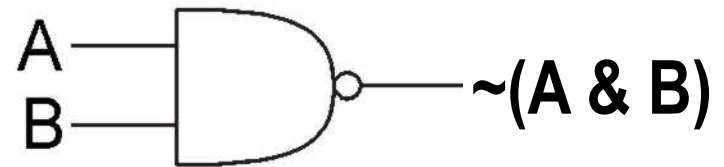
OR



NOR

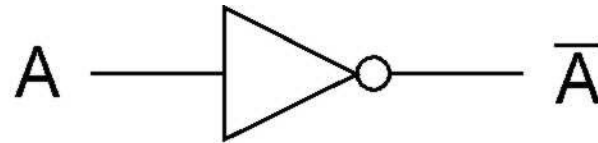


AND

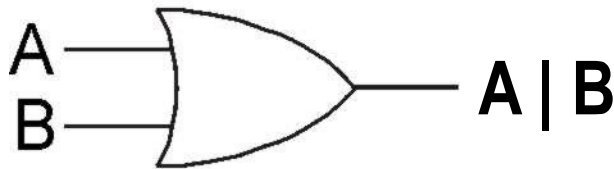


NAND

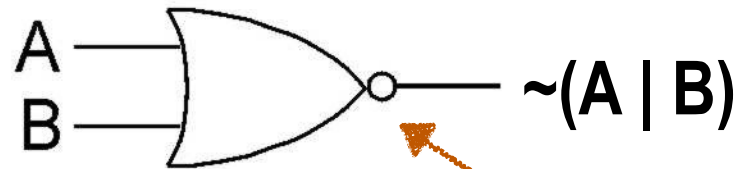
Basic Logic Gates



NOT

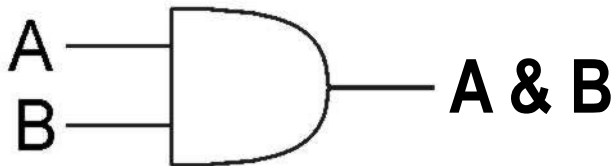


OR

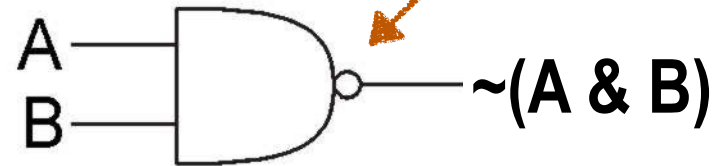


NOR

The little circle means NOT



AND



NAND

Full (1-bit) Adder

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

A	B	C_{in}	S	C_{out}
				t
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.

Truth Table

A	B	C_{in}	S	C_{out}
				t
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.

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

Truth Table

A	B	C _{in}	S	C _{ou} t
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.

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

Truth Table

A	B	C _{in}	S	C _{ou} t
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.

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

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.

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

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

Truth Table

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

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

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

A	B	C _{in}	S	C _{ou}
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

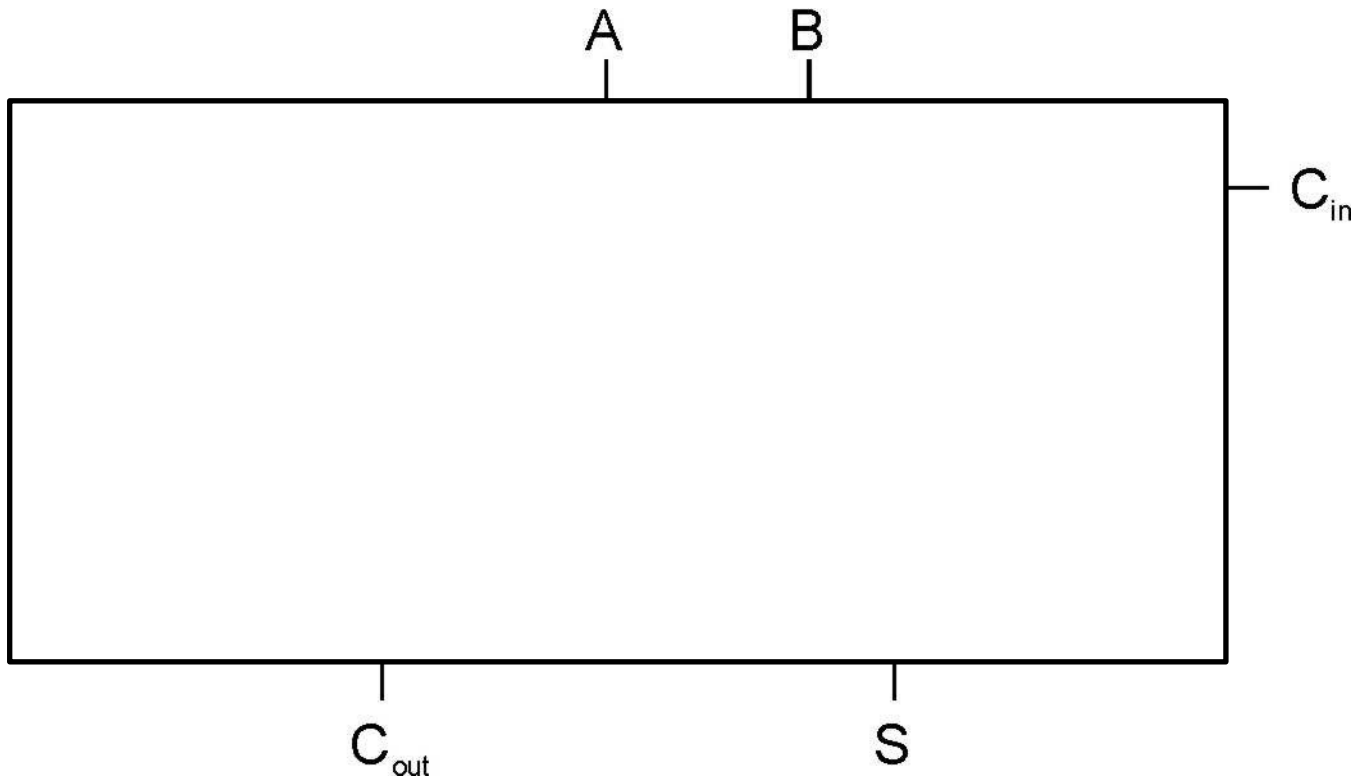
Full (1-bit) Adder

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

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

Full (1-bit) Adder

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

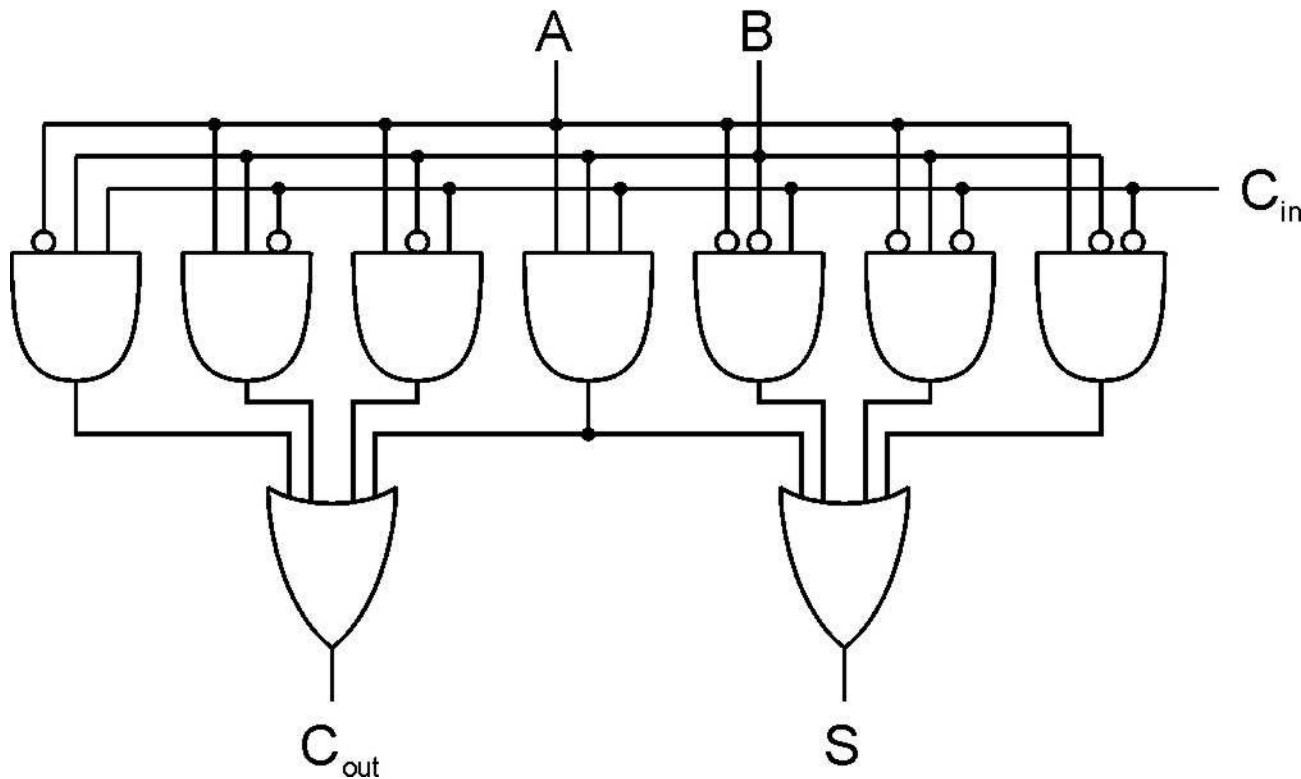


$$\begin{aligned} C_{ou} = & (\sim A \& B \& C_{in}) \\ & | (A \& \sim B \& C_{in}) \\ & | (A \& B \& \sim C_{in}) \\ & | (A \& B \& C_{in}) \end{aligned}$$

Full (1-bit) Adder

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

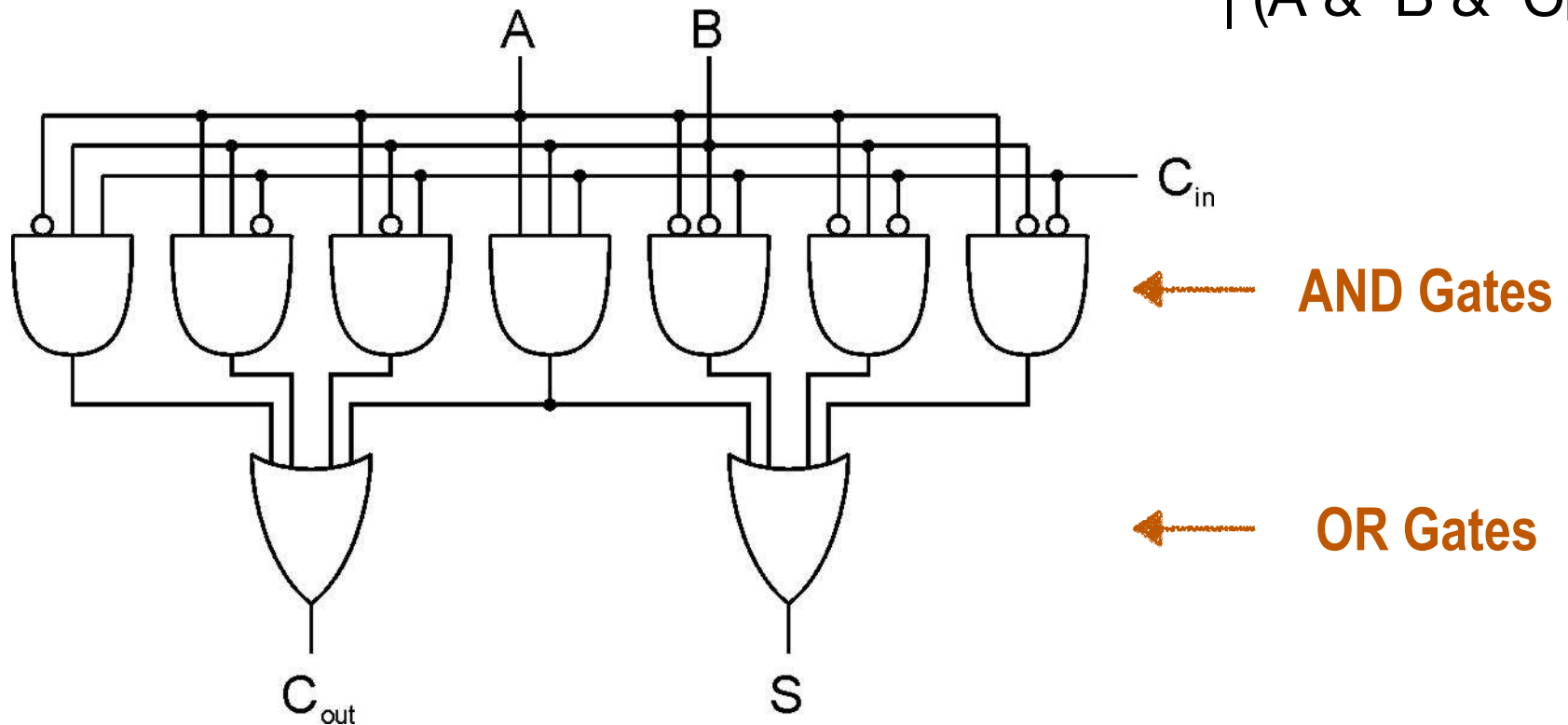
$$\begin{aligned} C_{ou} = & (\sim A \& B \& C_{in}) \\ & | (A \& \sim B \& C_{in}) \\ & | (A \& B \& \sim C_{in}) \\ & | (A \& B \& C_{in}) \end{aligned}$$



Full (1-bit) Adder

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

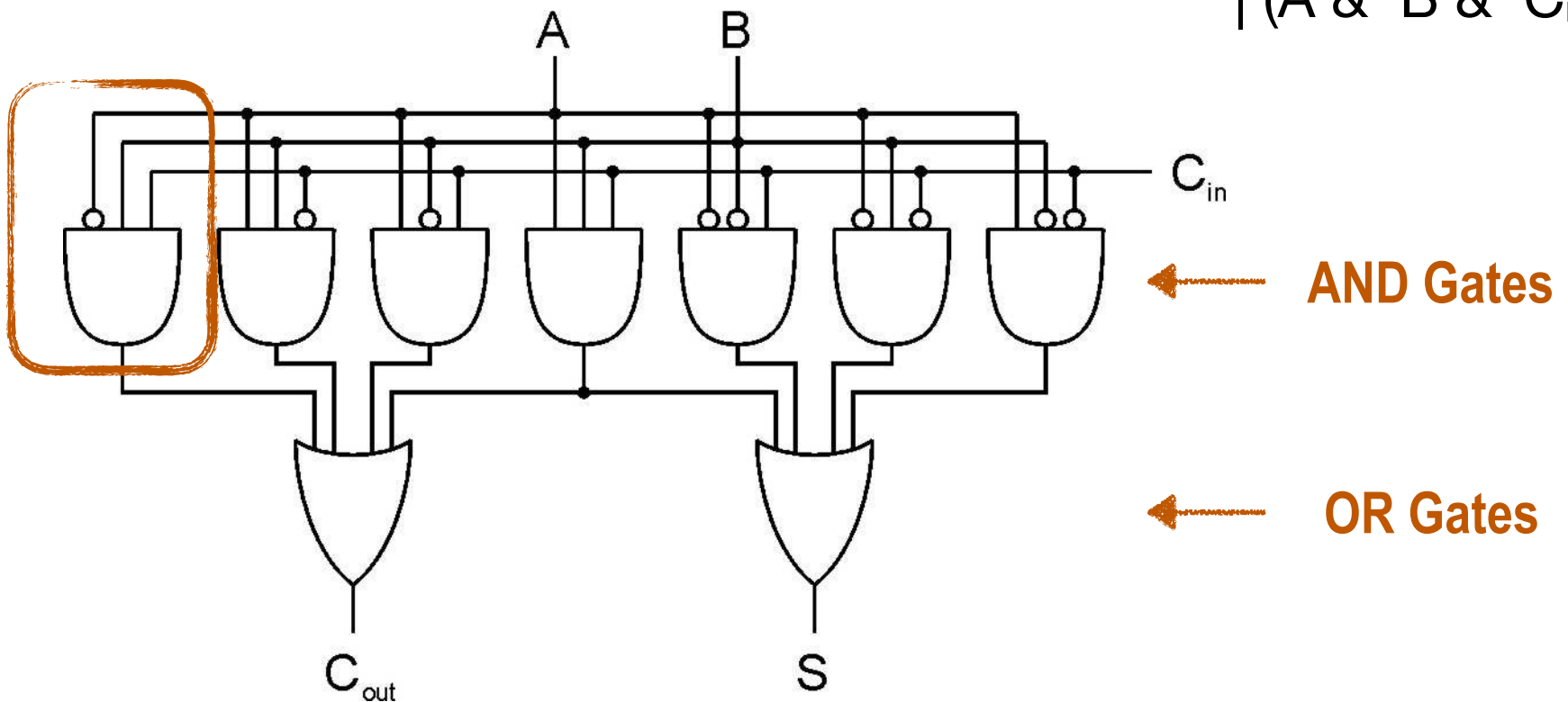
$$\begin{aligned} C_{ou} = & (\sim A \& B \& C_{in}) \\ & | (A \& \sim B \& C_{in}) \\ & | (A \& B \& \sim C_{in}) \\ & | (A \& B \& C_{in}) \end{aligned}$$



Full (1-bit) Adder

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

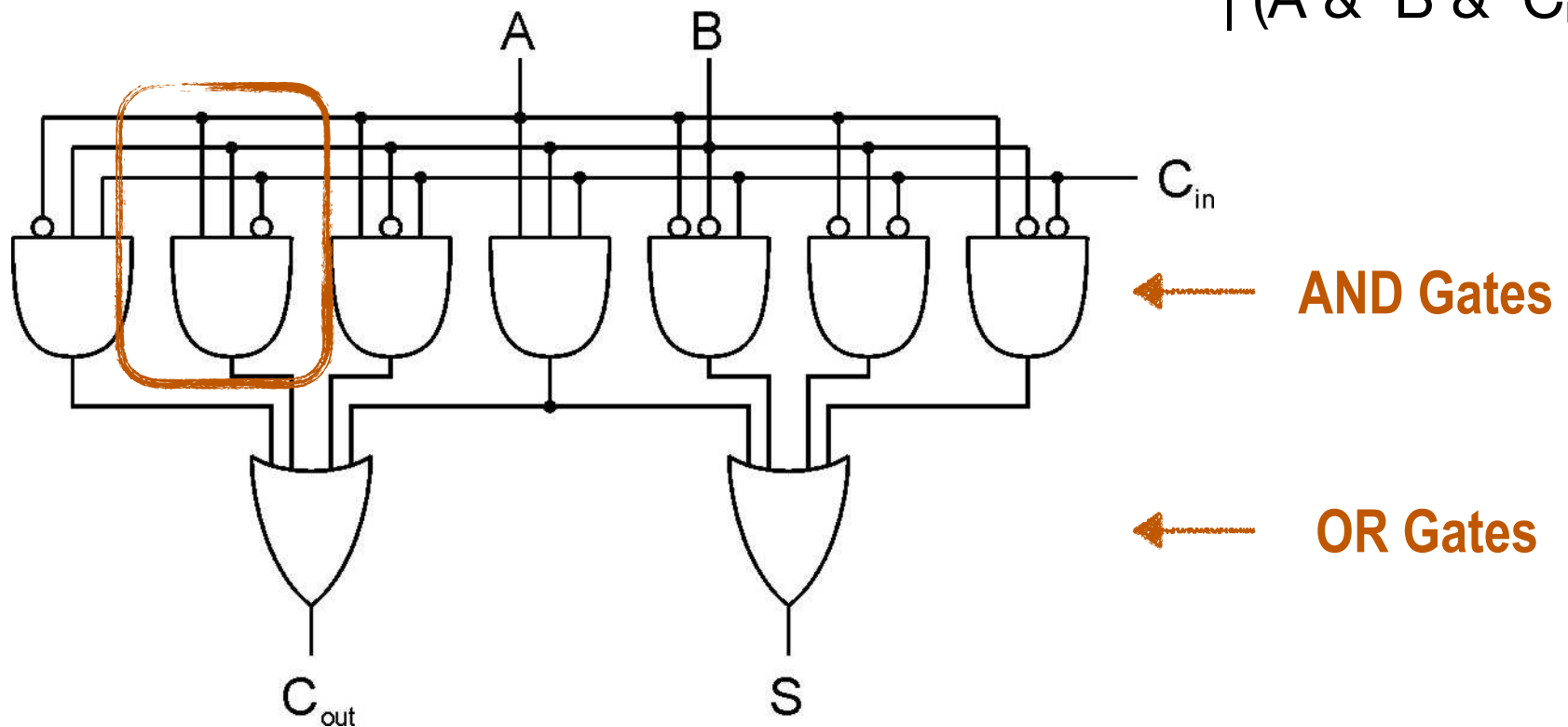
$$C_{ou} = (\sim A \& B \& C_{in}) \mid (A \& \sim B \& C_{in}) \mid (A \& B \& \sim C_{in}) \mid (A \& B \& C_{in})$$



Full (1-bit) Adder

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

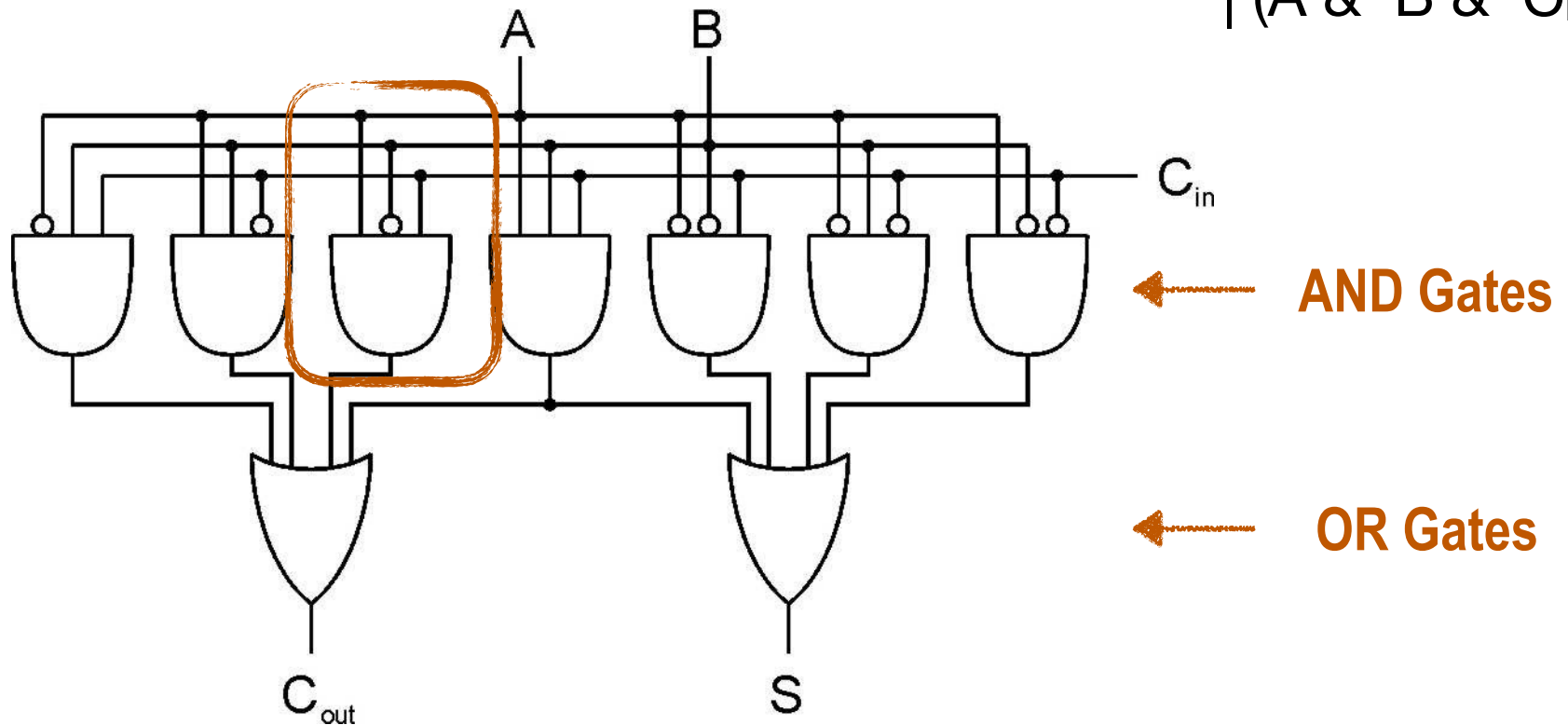
$$\begin{array}{l} C_{ou} = (\sim A \& B \& C_{in}) \\ | (A \& \sim B \& C_{in}) \\ | (A \& B \& \sim C_{in}) \\ | (A \& B \& C_{in}) \end{array}$$



Full (1-bit) Adder

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

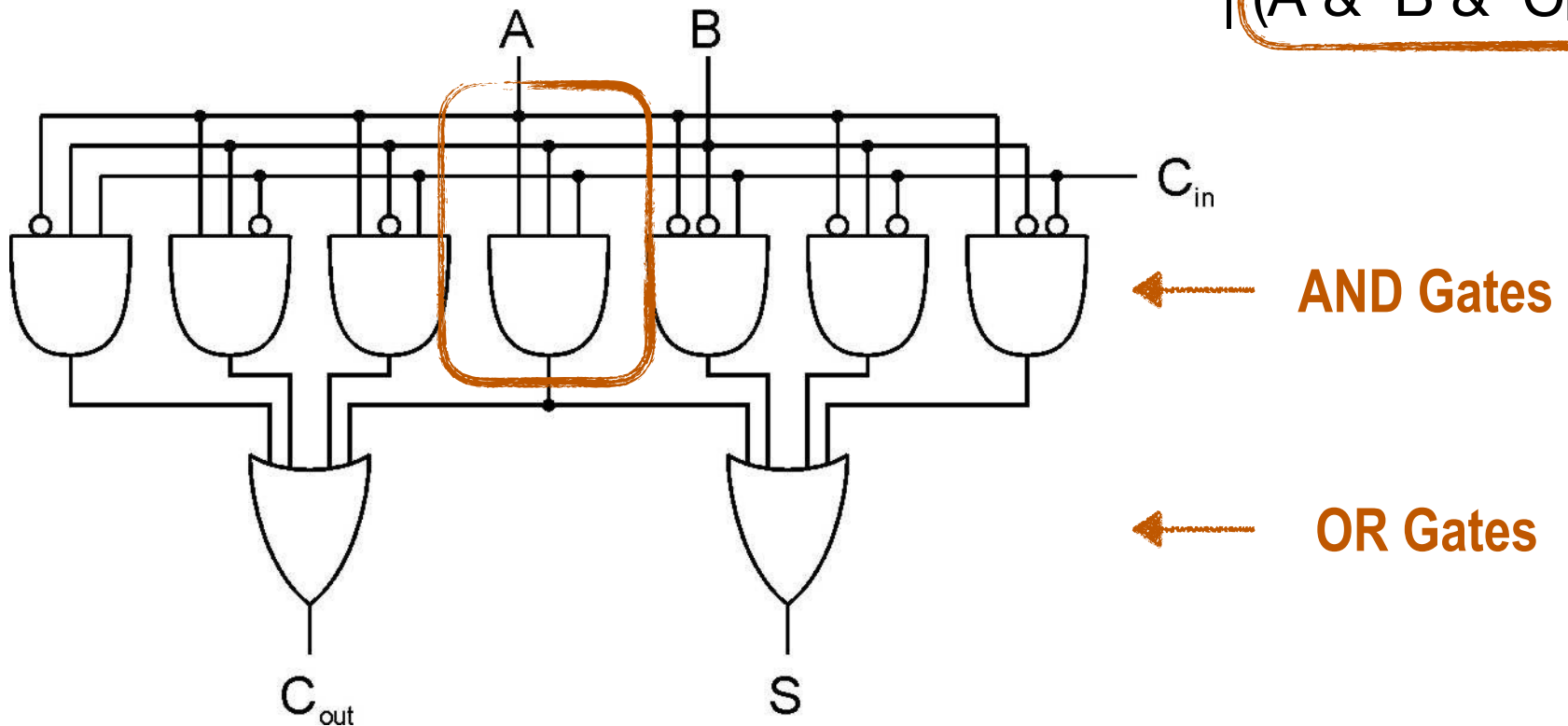
$$C_{ou} = (\sim A \& B \& C_{in}) \mid (A \& \sim B \& C_{in}) \mid (A \& B \& \sim C_{in}) \mid (A \& B \& C_{in})$$



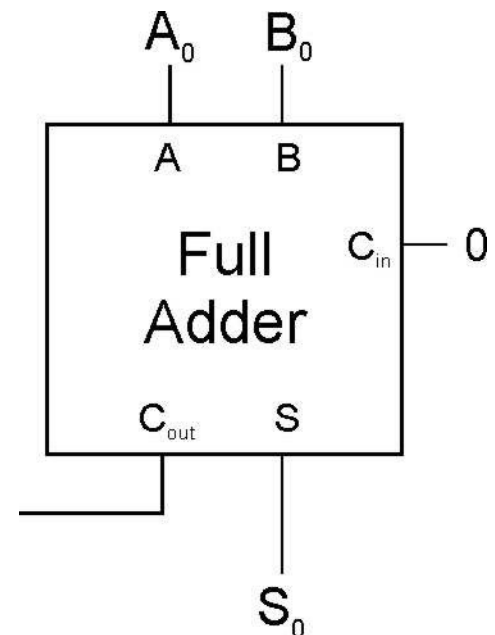
Full (1-bit) Adder

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

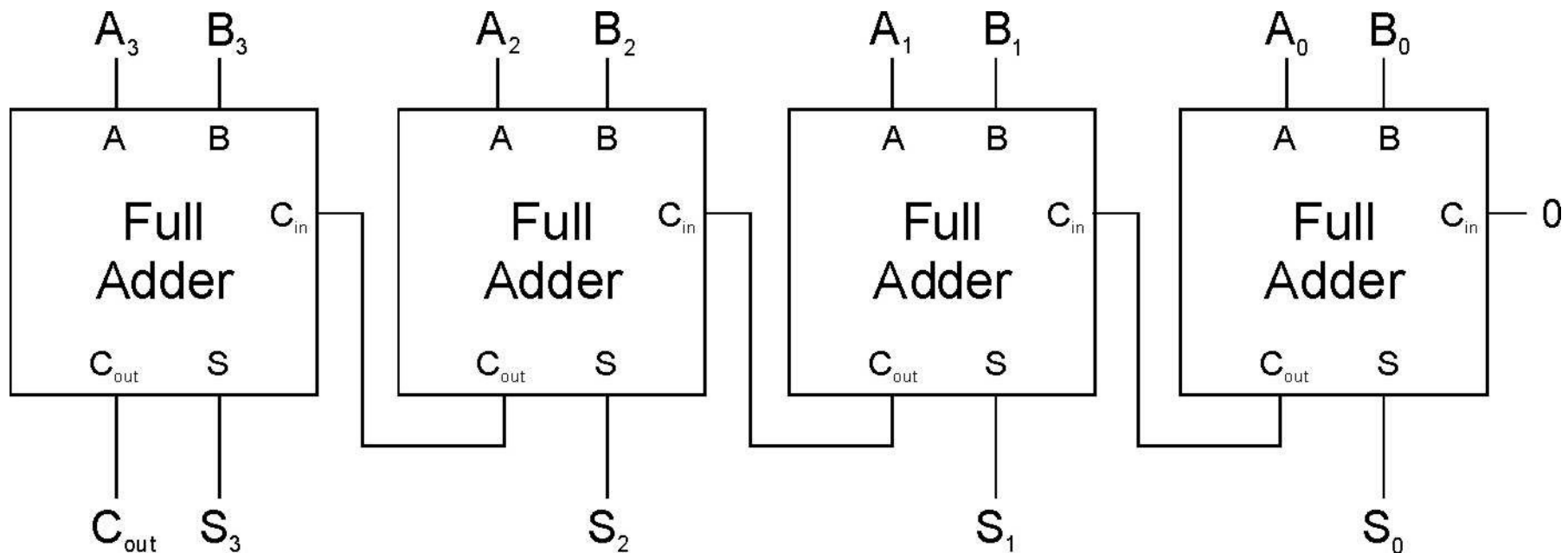
$$\begin{aligned} C_{ou} = & (\sim A \& B \& C_{in}) \\ & | (A \& \sim B \& C_{in}) \\ & | (A \& B \& \sim C_{in}) \\ & | (A \& B \& C_{in}) \end{aligned}$$



Four-bit Adder

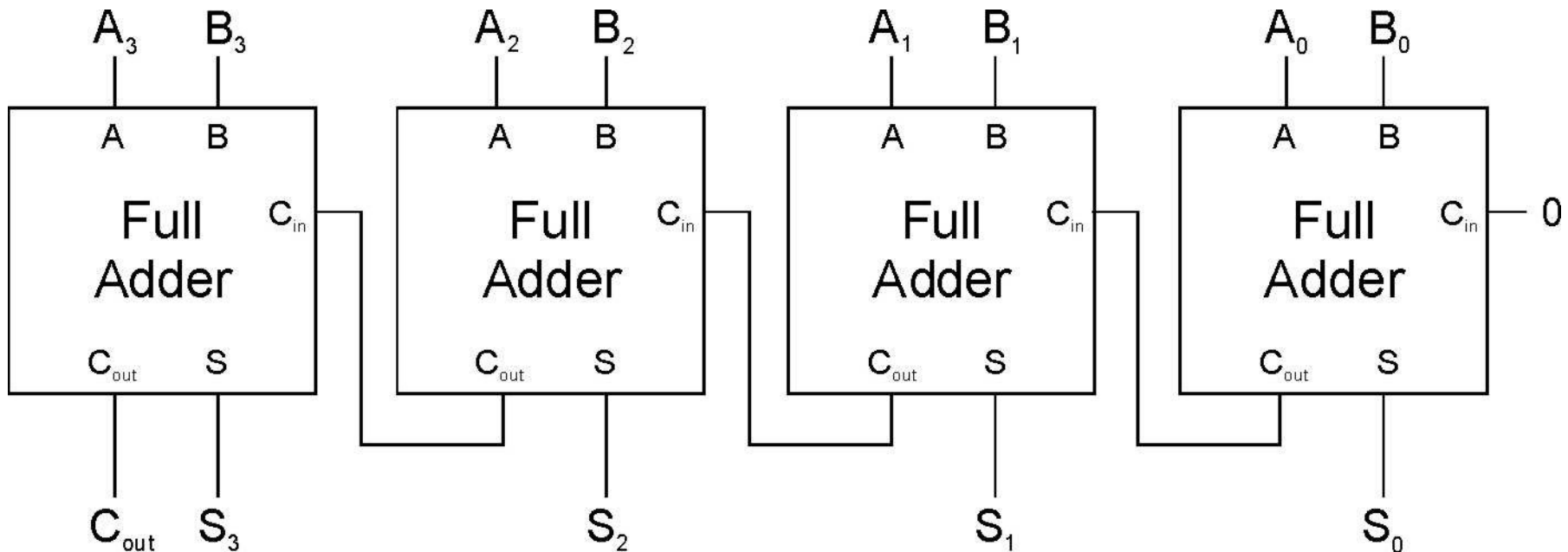


Four-bit Adder



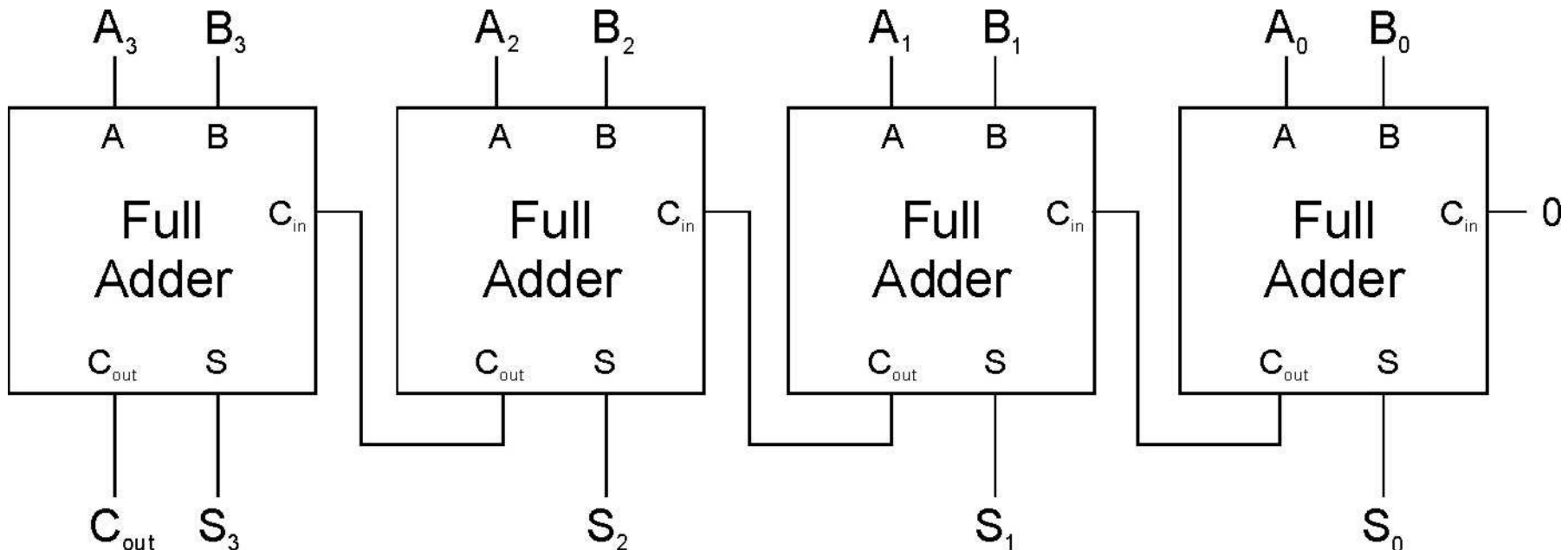
Four-bit Adder

- 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

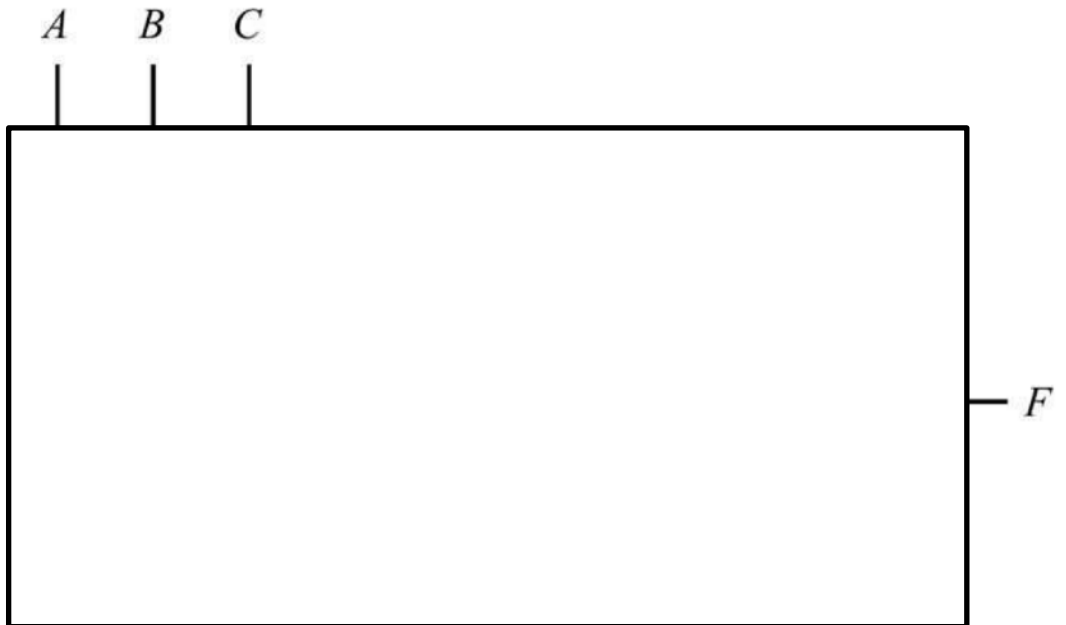
- Design digital components from basic logic gates
- Key idea: use the truth table!

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?

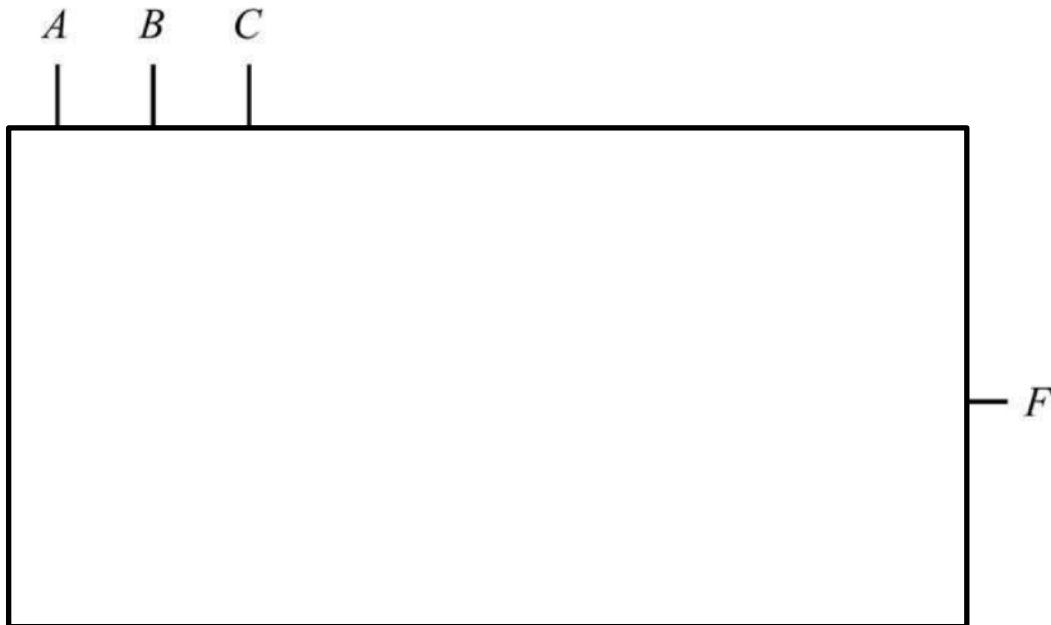
Logic Design

- 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

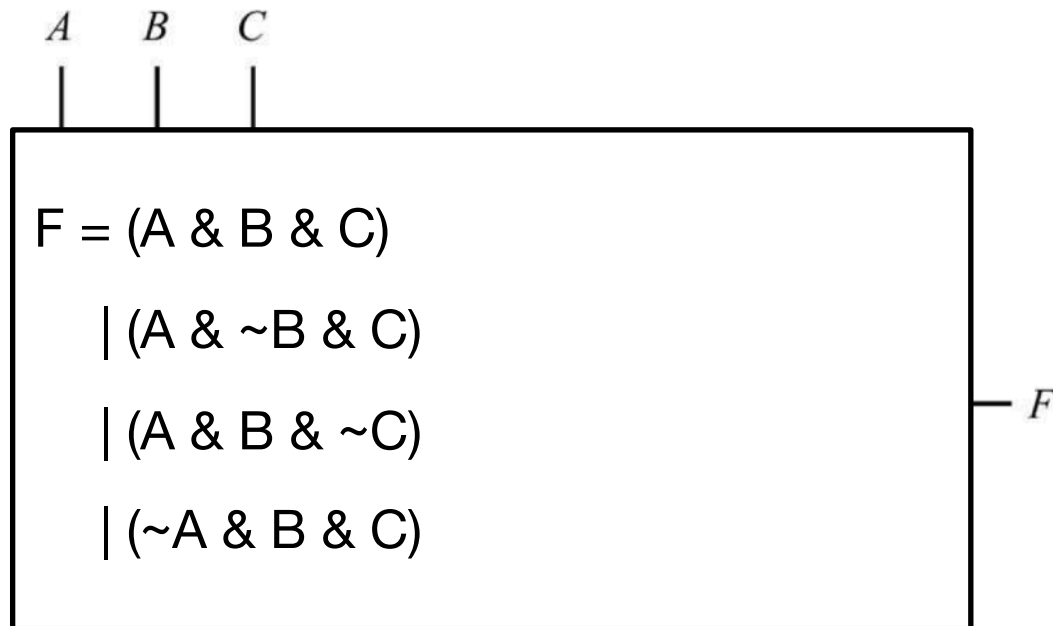
- Design digital components from basic logic gates
- Key idea: use the truth table!
- Example: how to design a piece of circuit that does majority vote?



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

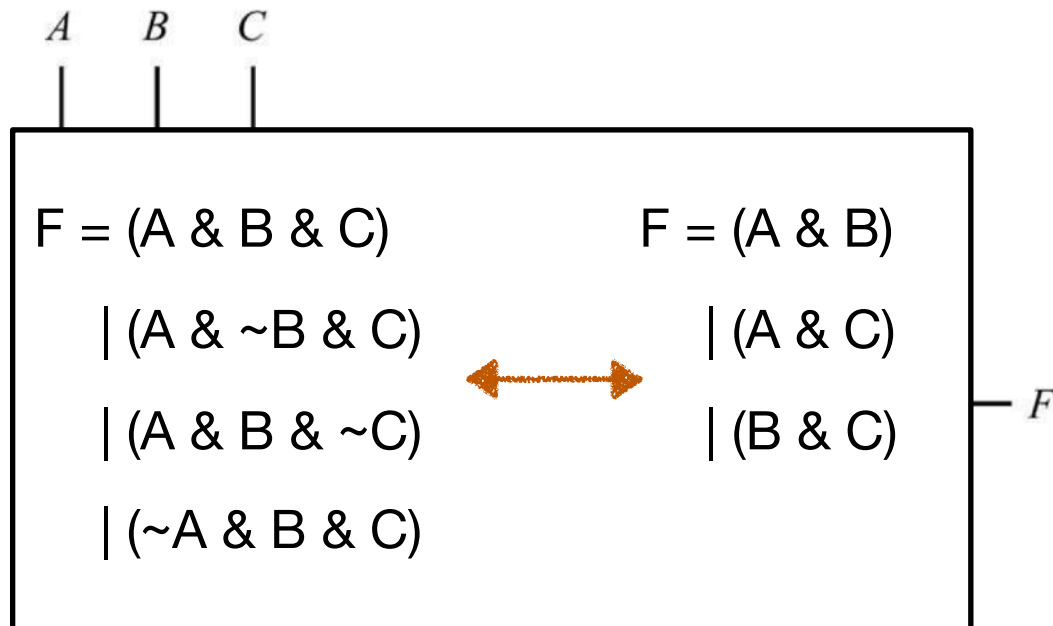
- Design digital components from basic logic gates
- Key idea: use the truth table!
- Example: how to design a piece of circuit that does majority vote?



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

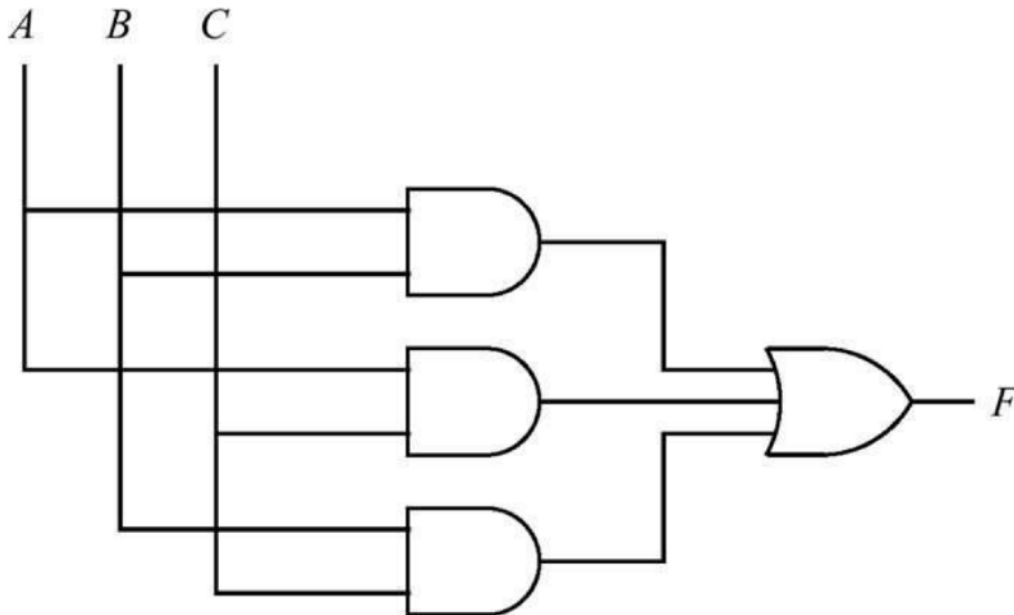
- Design digital components from basic logic gates
- Key idea: use the truth table!
- Example: how to design a piece of circuit that does majority vote?



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

Multiplication

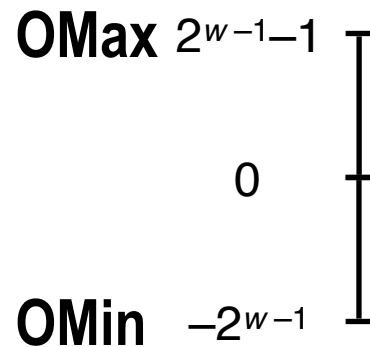
Multiplication

- 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)

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


0

OMin -2^{w-1}



Product

0



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

PMax

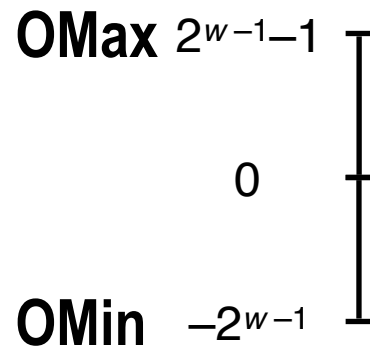
0



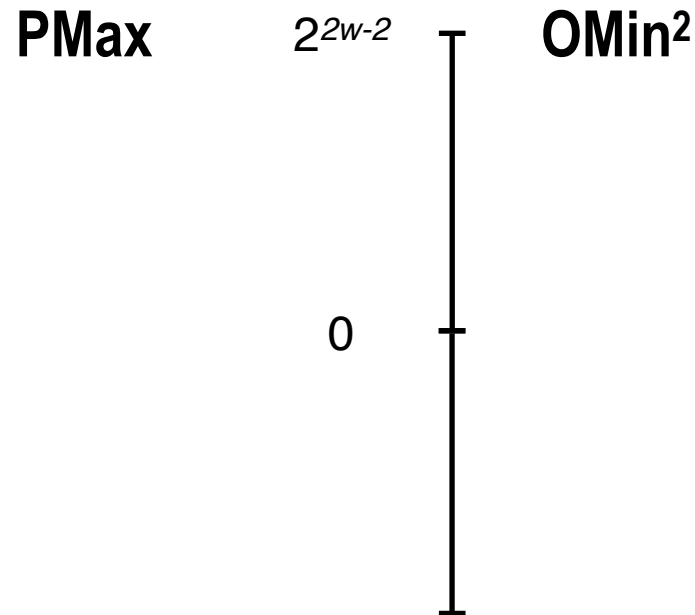
Multiplication

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

Original Number (w bits)



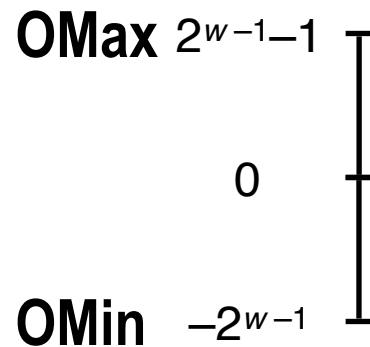
Product



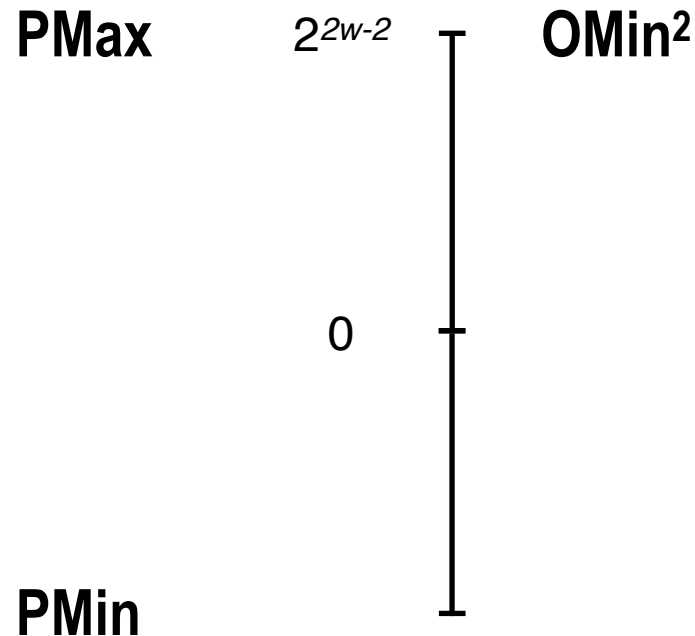
Multiplication

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

Original Number (w bits)



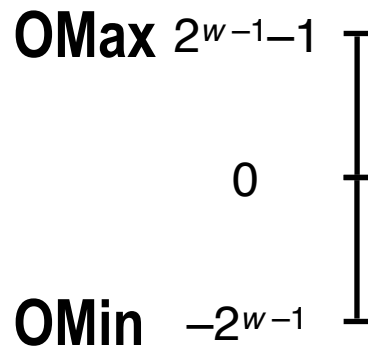
Product



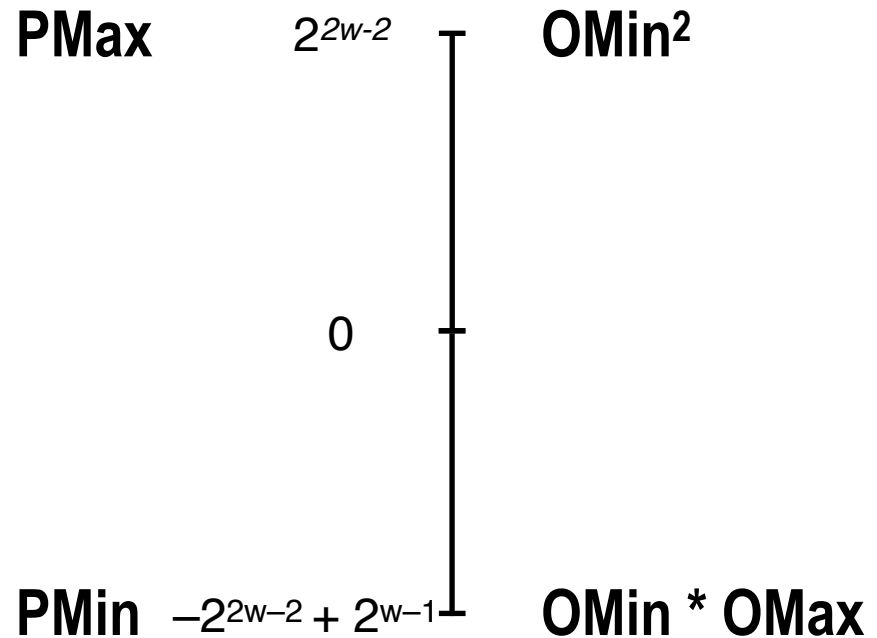
Multiplication

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

Original Number (w bits)



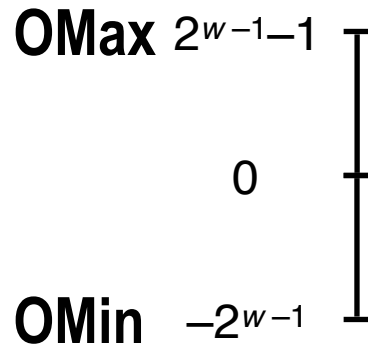
Product



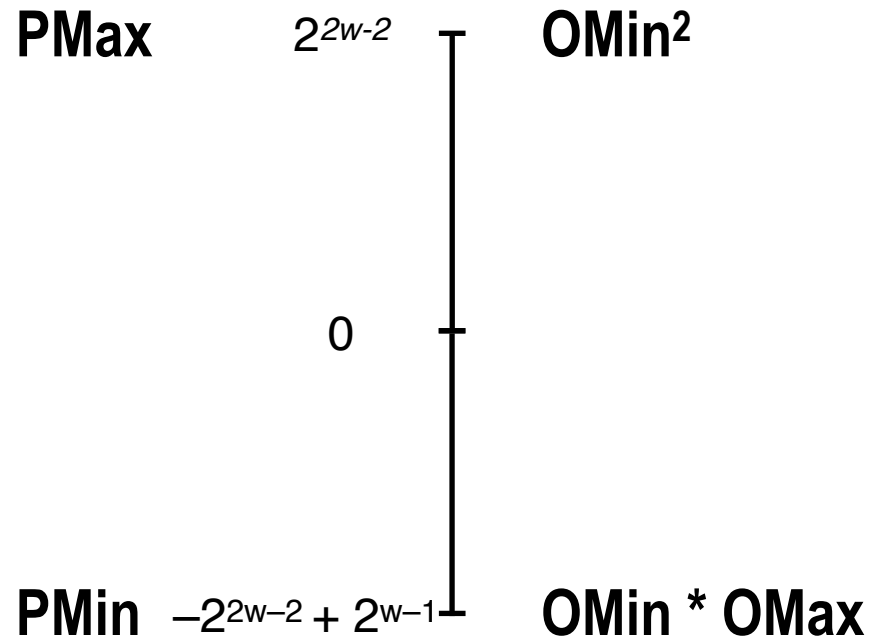
Multiplication

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

Original Number (w bits)



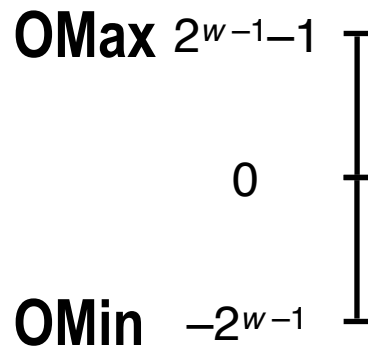
Product ($2w$ bits)



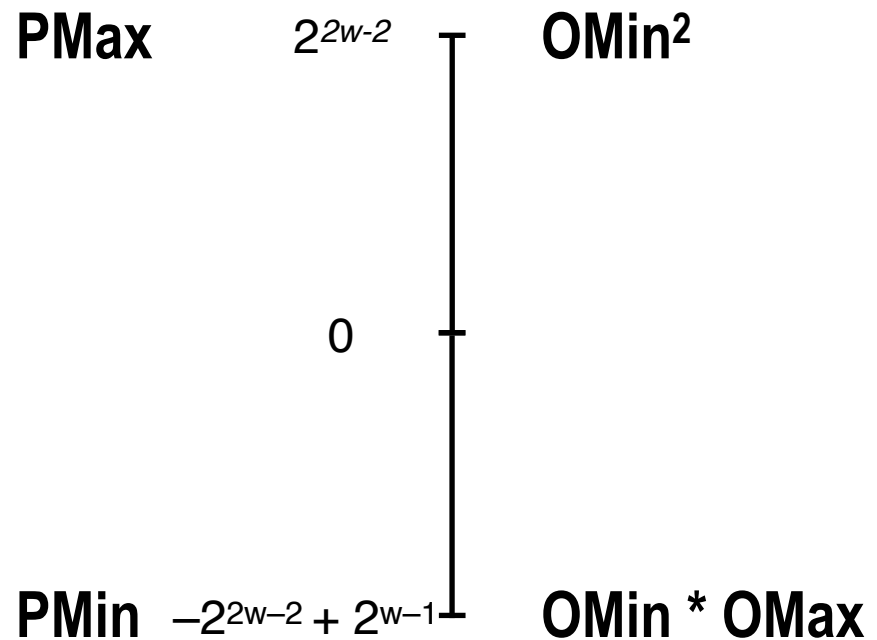
Multiplication

- 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)

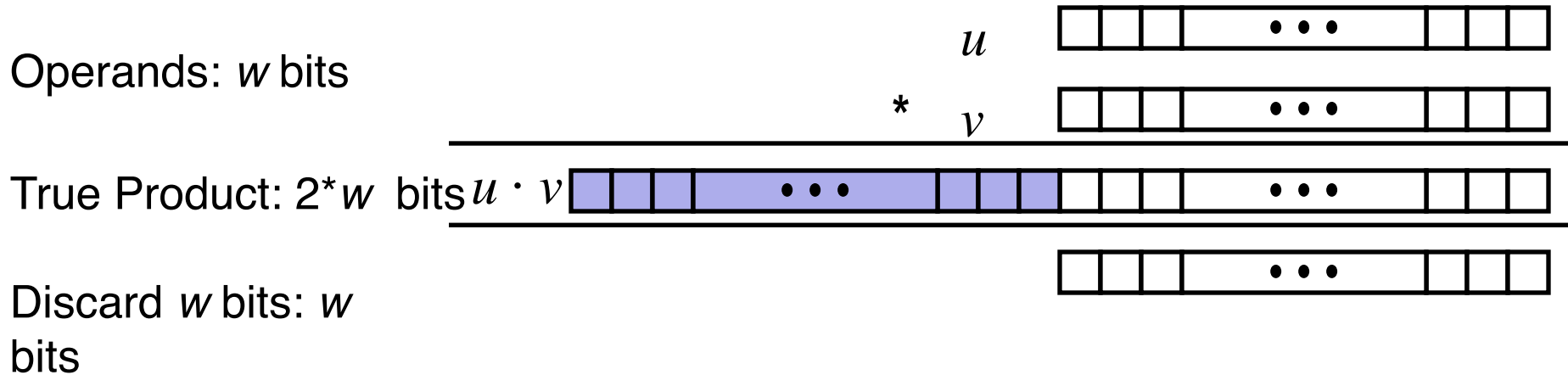
Original Number (w bits)



Product ($2w$ bits)



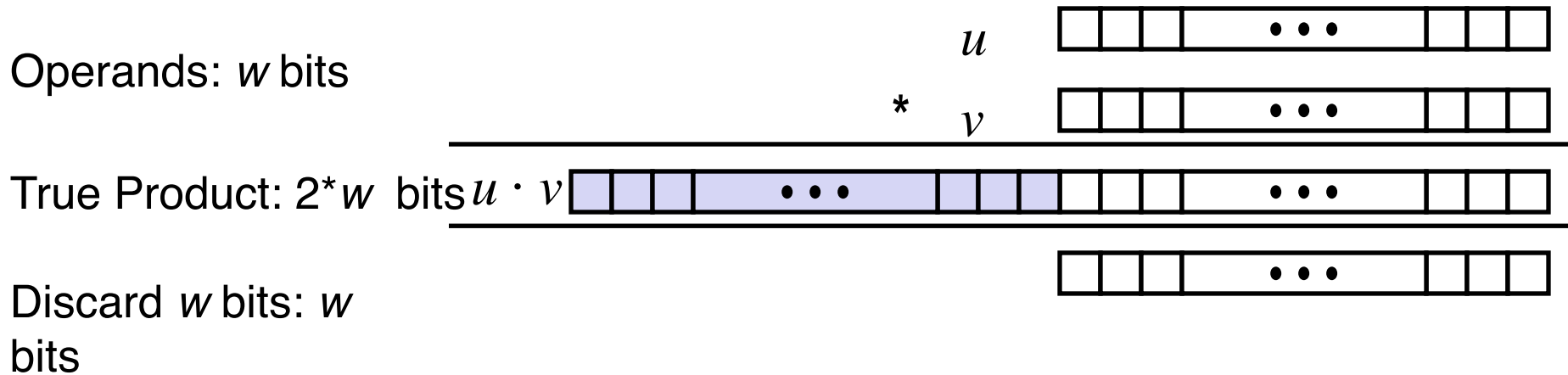
Unsigned Multiplication in C



- Standard Multiplication Function
 - Ignores high order w bits
- Effectively Implements the following:

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

Signed Multiplication in C



- 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

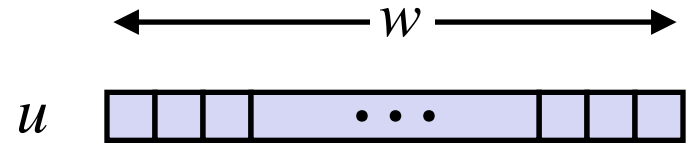
- Operation

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

Power-of-2 Multiply with Shift

- Operation

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

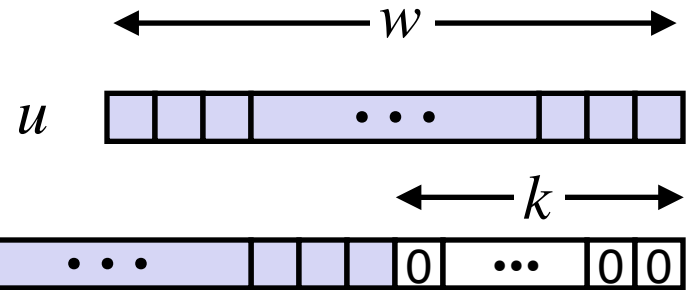


Power-of-2 Multiply with Shift

- Operation

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

True Product: $w+k$ bits $u \cdot 2^k$



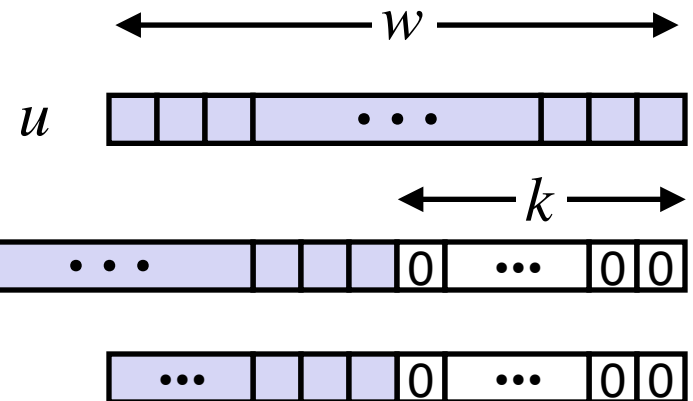
Power-of-2 Multiply with Shift

- Operation

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

True Product: $w+k$ bits $u \cdot 2^k$

Discard k bits (if
overflow)



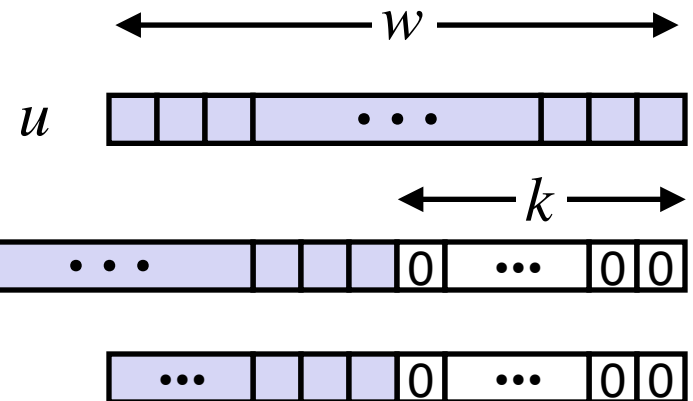
Power-of-2 Multiply with Shift

- Operation

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

True Product: $w+k$ bits $u \cdot 2^k$

Discard k bits (if
overflow)



- Most machines shift and add faster than multiply

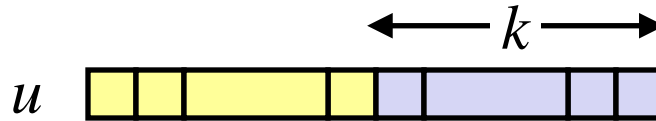
- Compiler generates this code automatically
- $u \ll 3 == u * 8$
- $(u \ll 5) - (u \ll 3) == u * 24$

Unsigned Power-of-2 Divide with Shift

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

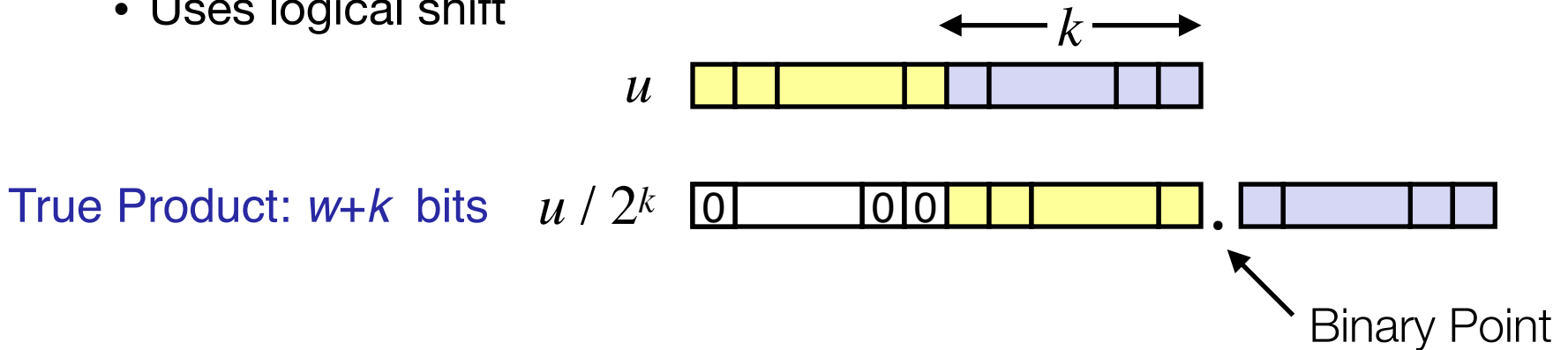
Unsigned Power-of-2 Divide with Shift

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



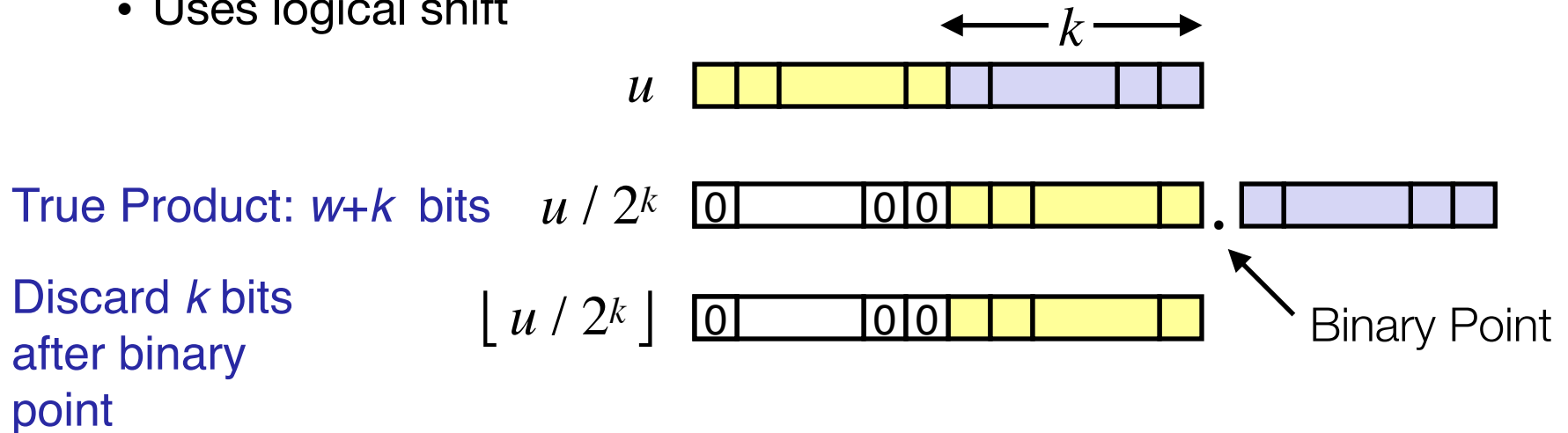
Unsigned Power-of-2 Divide with Shift

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

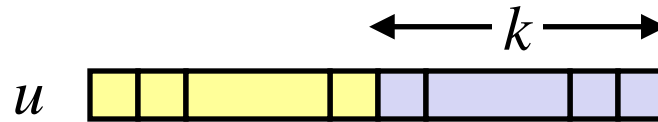


Unsigned Power-of-2 Divide with Shift

- Implement power-of-2 divide with shift

- $u \gg k$ gives $\lfloor u / 2^k \rfloor$ ($\lfloor 2.34 \rfloor = 2$)

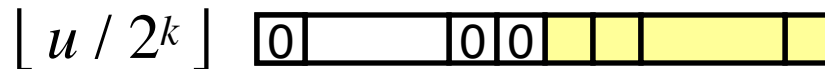
- Uses logical shift



True Product: $w+k$ bits



Discard k bits
after binary
point



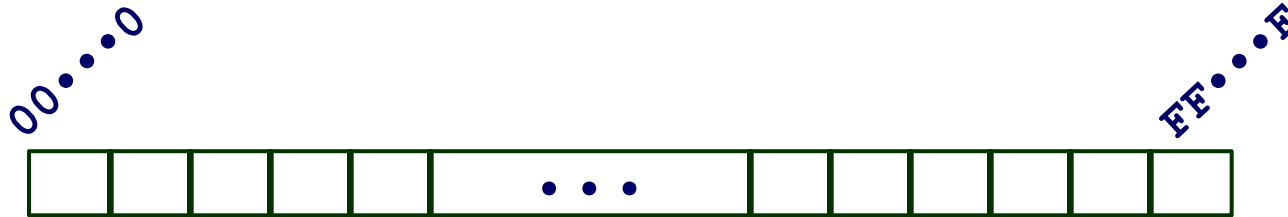
Binary Point

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

Today: Representing Information in Binary

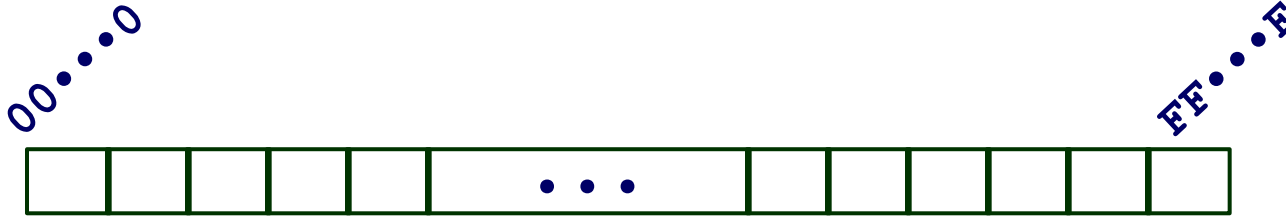
- Why Binary (bits)?
- Bit-level manipulations
- Integers
 - Representation: unsigned and signed
 - Conversion, casting
 - Expanding, truncating
 - Addition, negation, multiplication, shifting
 - Summary
- Representations in memory, pointers, strings

Byte-Oriented Memory Organization



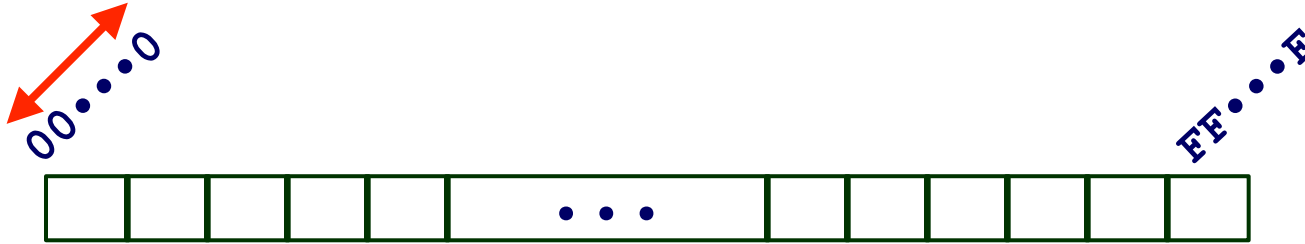
- Programs refer to data by address
 - Conceptually, envision it as a very large array of bytes: **byte-addressable**
 - An address is like an index into that array
 - and, a pointer variable stores an address

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 18.4×10^{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 18.4×10^{18}