

CSC 252: Computer Organization

Spring 2018: Lecture 3

Instructor: Yuhao Zhu

Department of Computer Science
University of Rochester

Action Items:

- **Trivia 1 is due tomorrow, midnight**
- **Main assignment due Feb. 2, midnight**

Slide Credits: Bryant, O'Hallaron, Patt, Patel

Announcement

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

Previously in 252...

- Computers are built to understand bits: 0 and 1
 - 0: low (no) voltage; 1: high voltage
- Hexadecimal Notation: 0-9 and A-F

1111	1110
------	------

F

E

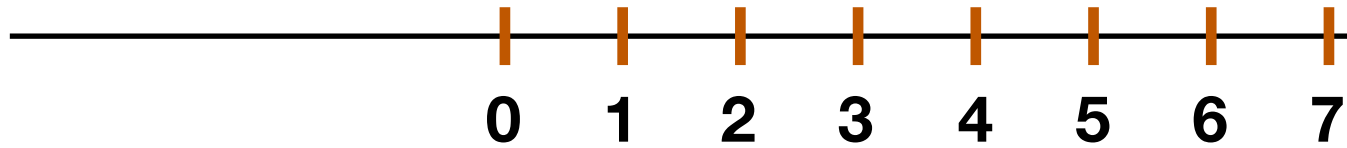
- Integer representations (Fixed-point really)
 - 1111 is really 1111.
 - Unsigned vs. Signed Integers

Encoding Negative Numbers

- Two's Complement

Encoding Negative Numbers

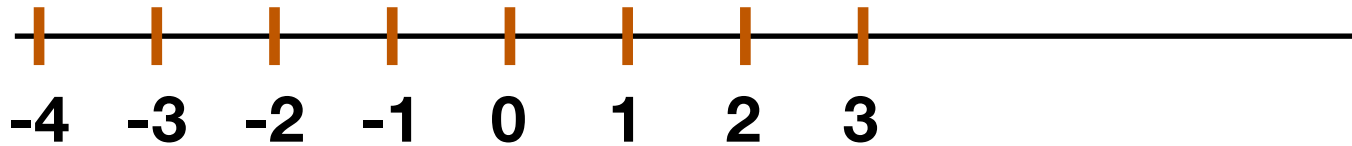
- Two's Complement



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

Encoding Negative Numbers

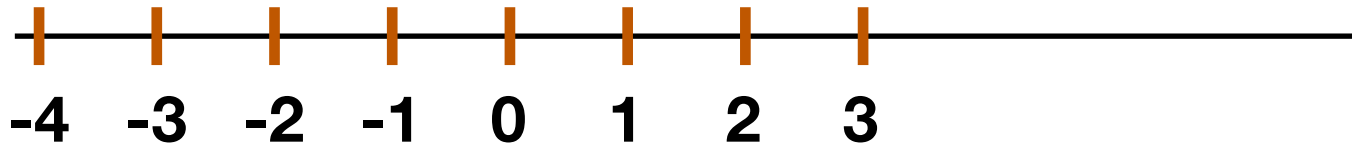
- Two's Complement



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

Encoding Negative Numbers

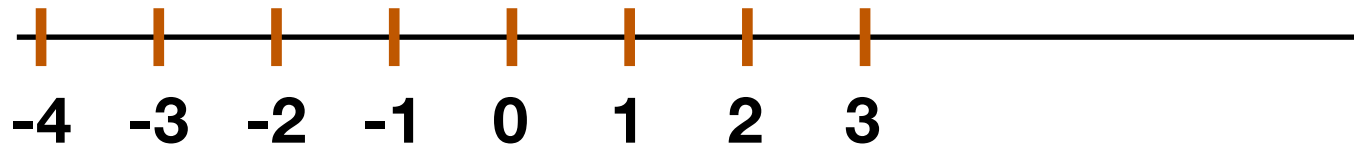
- Two's Complement



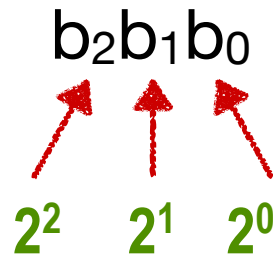
Signed	Unsigned	Binary
0	0	000
1	1	001
2	2	010
3	3	011
-4	4	100
-3	5	101
-2	6	110
-1	7	111

Encoding Negative Numbers

- Two's Complement



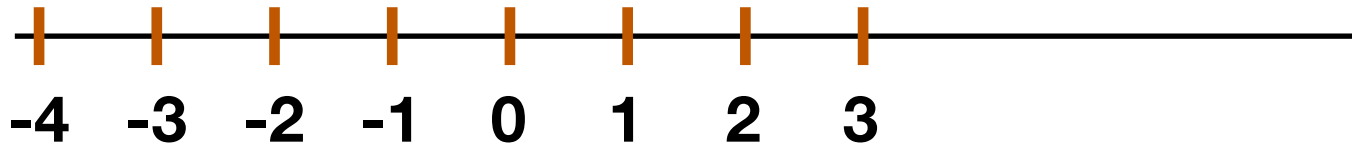
Weights in
Unsigned



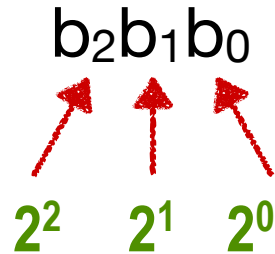
Signed	Unsigned	Binary
0	0	000
1	1	001
2	2	010
3	3	011
-4	4	100
-3	5	101
-2	6	110
-1	7	111

Encoding Negative Numbers

- Two's Complement



Weights in
Unsigned



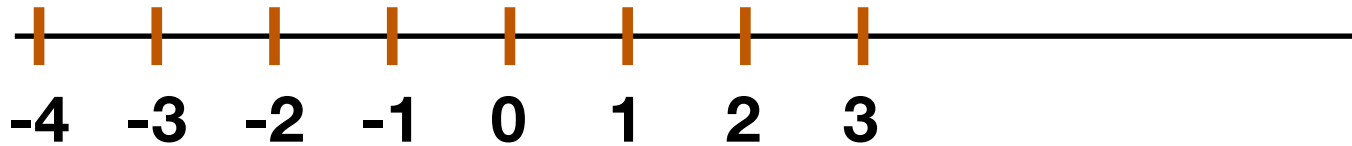
Weights in
Signed

-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



Weights in
Unsigned

$b_2 b_1 b_0$

2^2 2^1 2^0

Diagram showing the weights for unsigned binary representation. 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 2^1 2^0

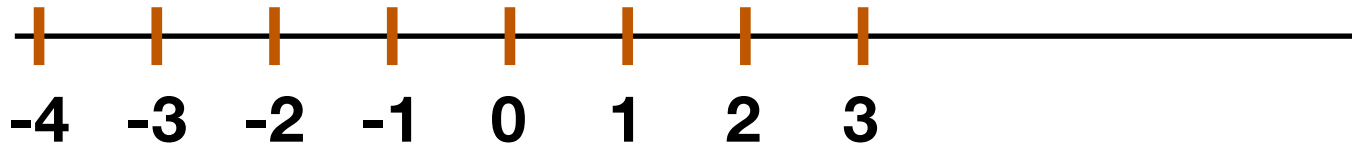
Diagram showing the weights for signed binary representation. The weight for b_2 is -2^2 , while for b_1 and b_0 it is 2^1 and 2^0 respectively.

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

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

$b_2 b_1 b_0$

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

Weights in Signed

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

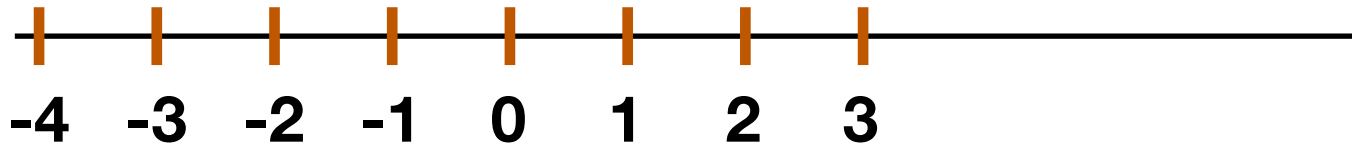
$$101_2 = 1 \cdot 2^0 + 0 \cdot 2^1 + (-1 \cdot 2^2) = -3_{10}$$

↑

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 2^1 2^0

Diagram showing the weights for unsigned binary representation. 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 2^1 2^0

Diagram showing the weights for signed binary representation. Red arrows point from the labels -2^2 , 2^1 , and 2^0 to the bits b_2 , b_1 , and b_0 respectively.

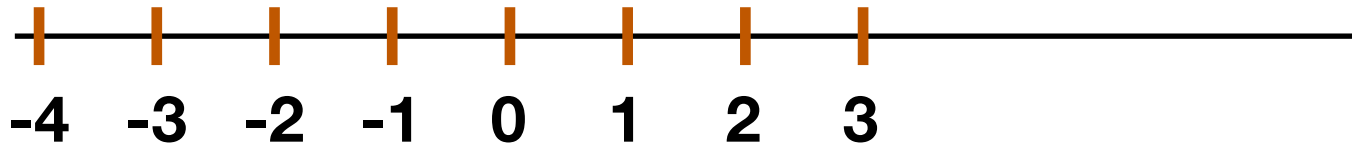
$$101_2 = 1 \cdot 2^0 + 0 \cdot 2^1 + (-1 \cdot 2^2) = -3_{10}$$

A red arrow points to the '1' in the 2^0 term of 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 2^1 2^0

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

Weights in
Signed

-2^2 2^1 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' in the binary number 101_2 .

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

Two-Complement Implications

- Only 1 zero
- Unsigned arithmetic still works
- Almost all C implementations use this
 - If you define a signed variable, it's internally represented using 2's complement

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

Two-Complement Implications

- Only 1 zero
- Unsigned arithmetic still works
- Almost all C implementations use this
 - If you define a signed variable, it's internally represented using 2's complement

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

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

Two-Complement Implications

- Only 1 zero
- Unsigned arithmetic still works
- Almost all C implementations use this
 - If you define a signed variable, it's internally represented using 2's complement

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

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

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

Two-Complement Implications

- Only 1 zero
- Unsigned arithmetic still works
- Almost all C implementations use this
 - If you define a signed variable, it's internally represented using 2's complement

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

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

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

- There is a bit that represents sign!
- 3 + 1 becomes -4 (called **overflow**. More on it later.)

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

Signed vs. Unsigned in C

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

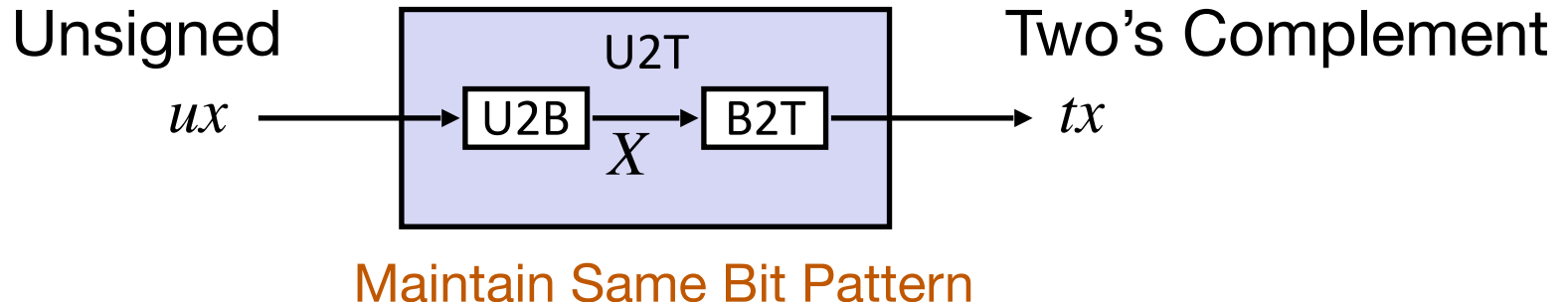
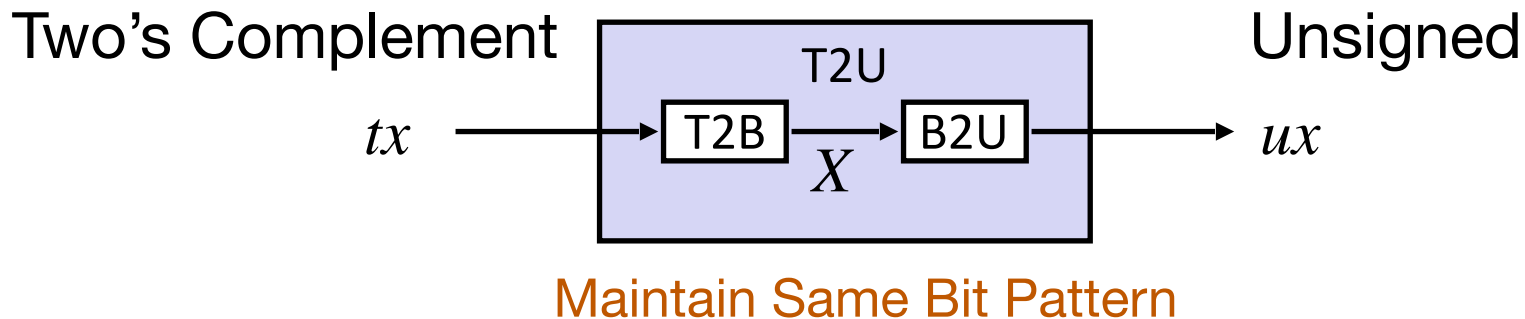
```
int tx, ty;  
unsigned ux, uy;  
tx = (int) ux; // U2T  
uy = (unsigned) ty; // T2U
```

- Implicit casting also occurs via assignments and procedure calls

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

Mapping Between Signed & Unsigned

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



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	\longrightarrow T2U \longrightarrow	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

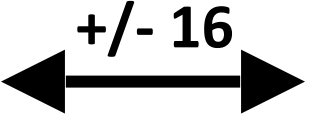
Bits	Signed		Unsigned
0000	0	→ T2U →	0
0001	1		1
0010	2		2
0011	3		3
0100	4		4
0101	5		5
0110	6		6
0111	7		7
1000	-8	← U2T ←	8
1001	-7		9
1010	-6		10
1011	-5		11
1100	-4		12
1101	-3		13
1110	-2		14
1111	-1		15

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

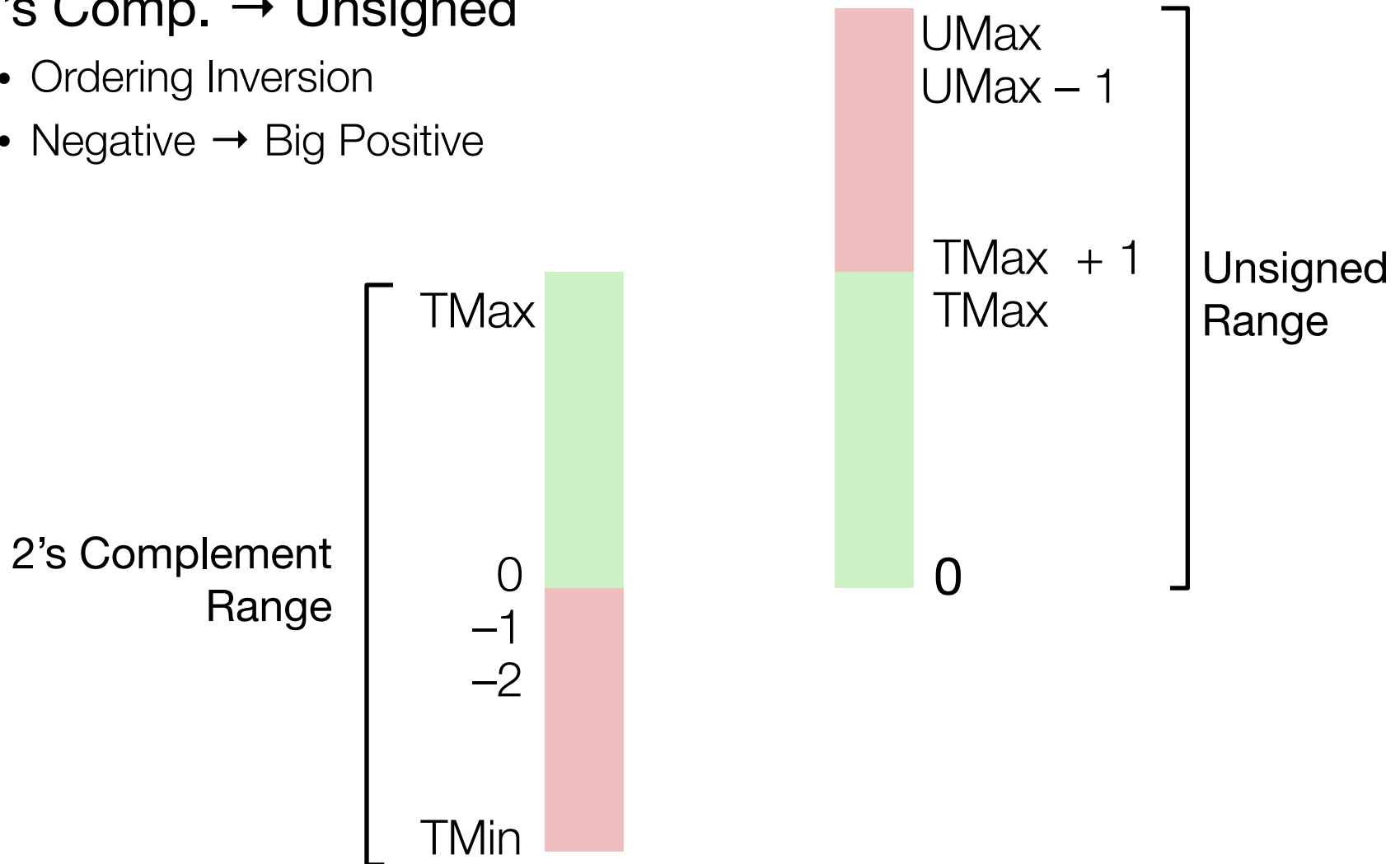
Mapping Signed \leftrightarrow Unsigned

Bits	Signed		Unsigned
0000	0	\longleftrightarrow =	0
0001	1		1
0010	2		2
0011	3		3
0100	4		4
0101	5		5
0110	6		6
0111	7		7
1000	-8	\longleftrightarrow +/- 16	8
1001	-7		9
1010	-6		10
1011	-5		11
1100	-4		12
1101	-3		13
1110	-2		14
1111	-1		15

Conversion Visualized

- 2's Comp. → Unsigned

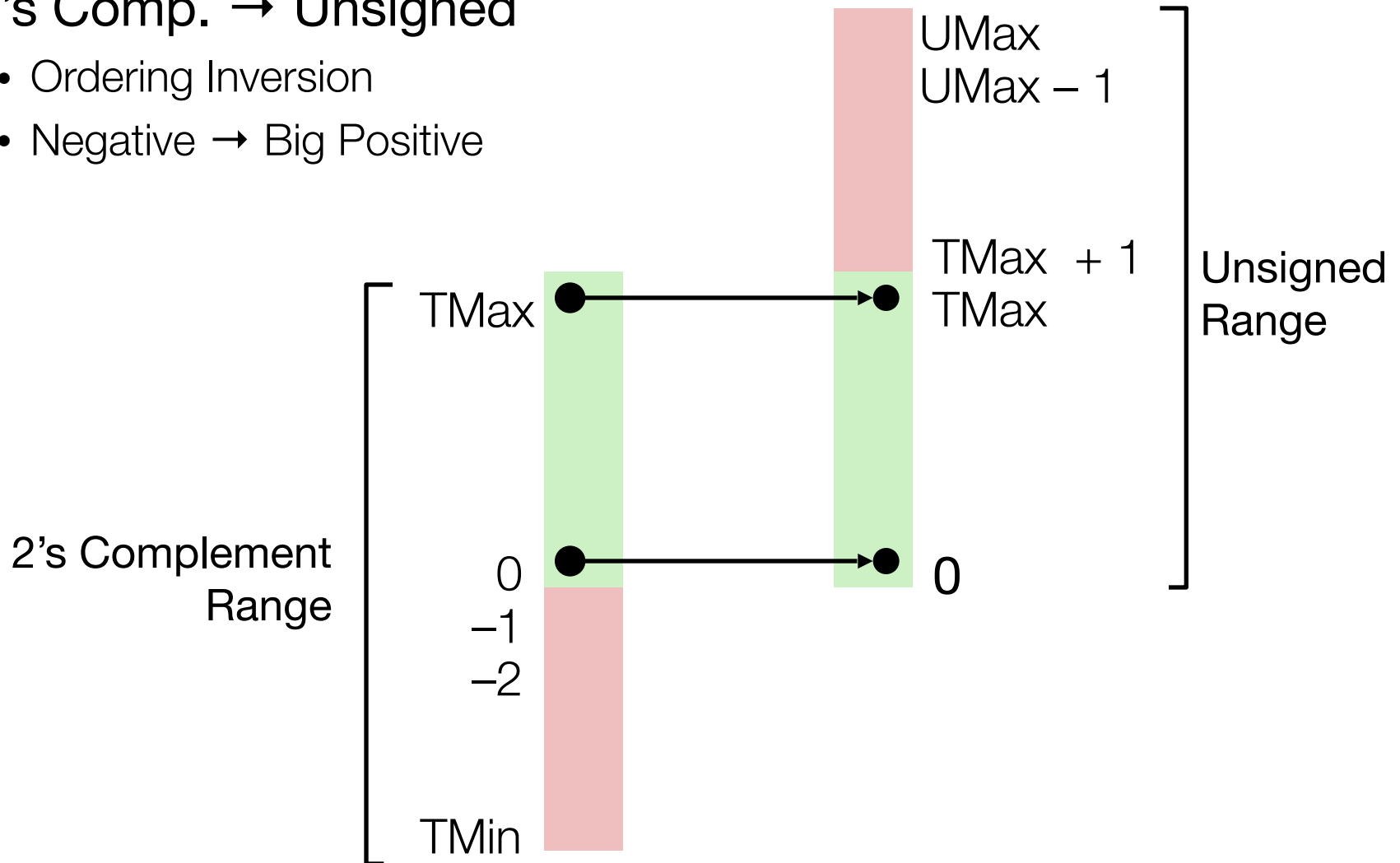
- Ordering Inversion
- Negative → Big Positive



Conversion Visualized

- 2's Comp. \rightarrow Unsigned

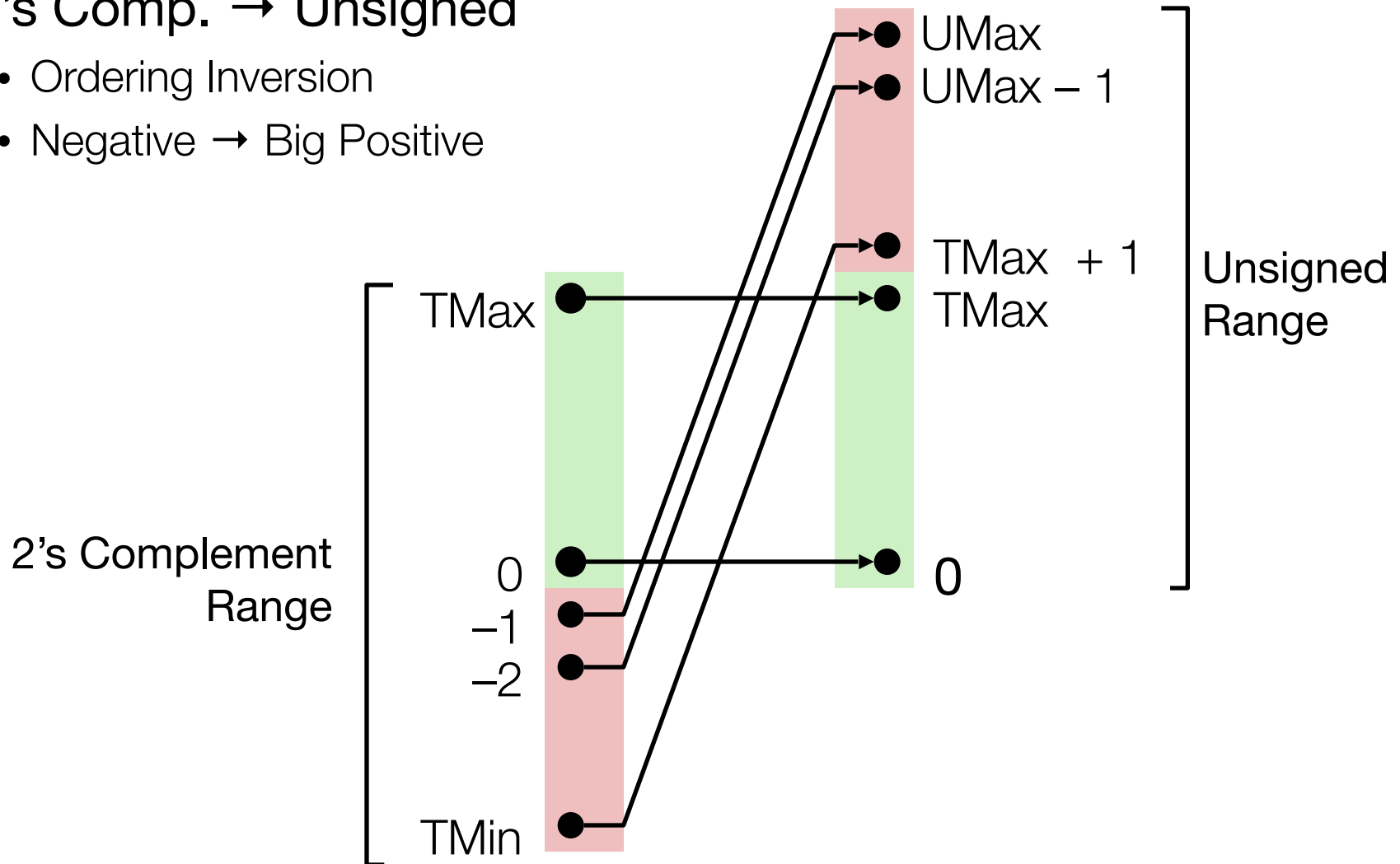
- Ordering Inversion
- Negative \rightarrow Big Positive



Conversion Visualized

- 2's Comp. → 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

Signed Extension

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

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

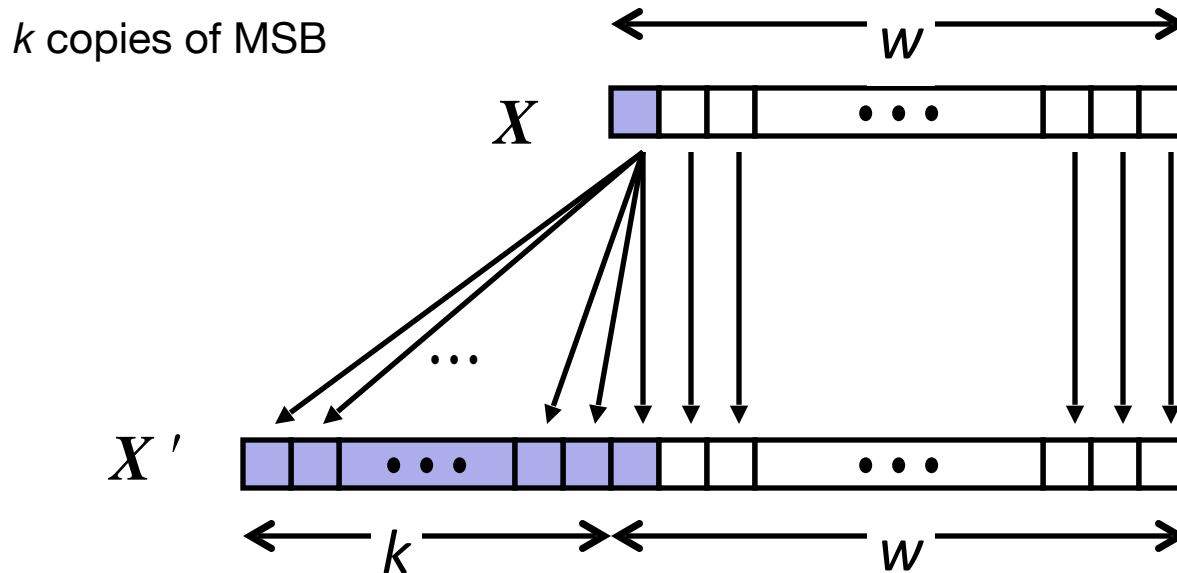
Signed Extension

- Task:

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

- Rule:

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



Signed Extension Example

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

	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

- Converting from smaller to larger integer data type
- C automatically performs sign extension

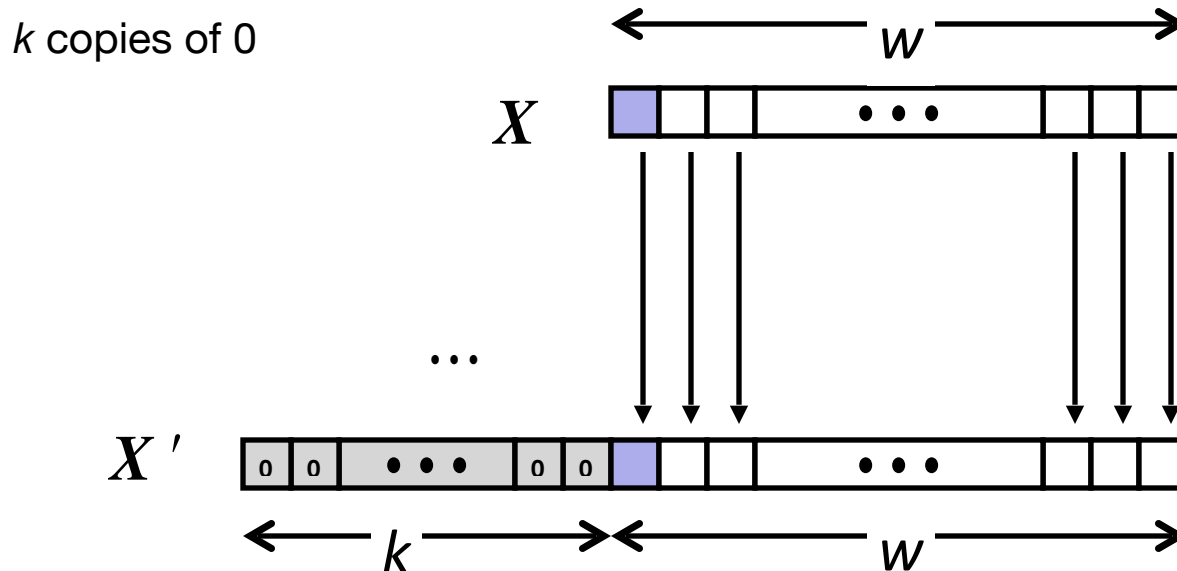
Unsigned Extension

- Task:

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

- Rule:

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



Unsigned Extension Example

```
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 Extension is sometimes also called zero extension

Truncating

- Truncating (e.g., int to short)
 - Leading bits are truncated, results reinterpreted

```
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

- Truncating (e.g., int to short)
 - Leading bits are truncated, results reinterpreted

```
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?

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

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

Unsigned Addition

- Similar to Decimal Addition

Signed	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

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

Signed	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 fit 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

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

Signed	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 fit within the size of the data type

Normal
Case

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

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

Overflow
Case

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

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

← True Sum

Signed	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 fit within the size of the data type

Signed	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

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



How is Addition Implemented in Hardware?

How is Addition Implemented in Hardware?

MOS = Metal Oxide Semiconductor

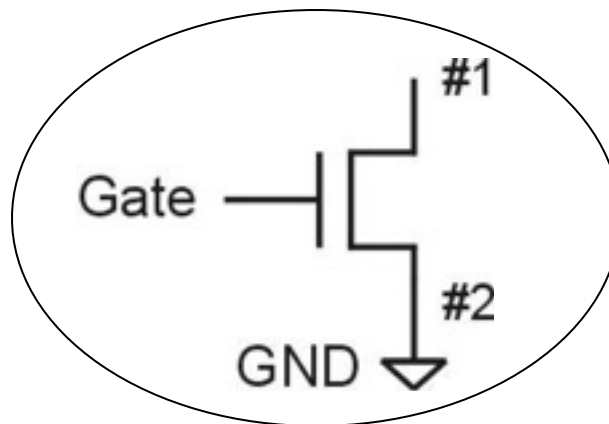
- two types: n-type and p-type

How is Addition Implemented in Hardware?

MOS = Metal Oxide Semiconductor

- two types: n-type and p-type

n-type (NMOS)



Terminal #2 must be connected to GND (0V).

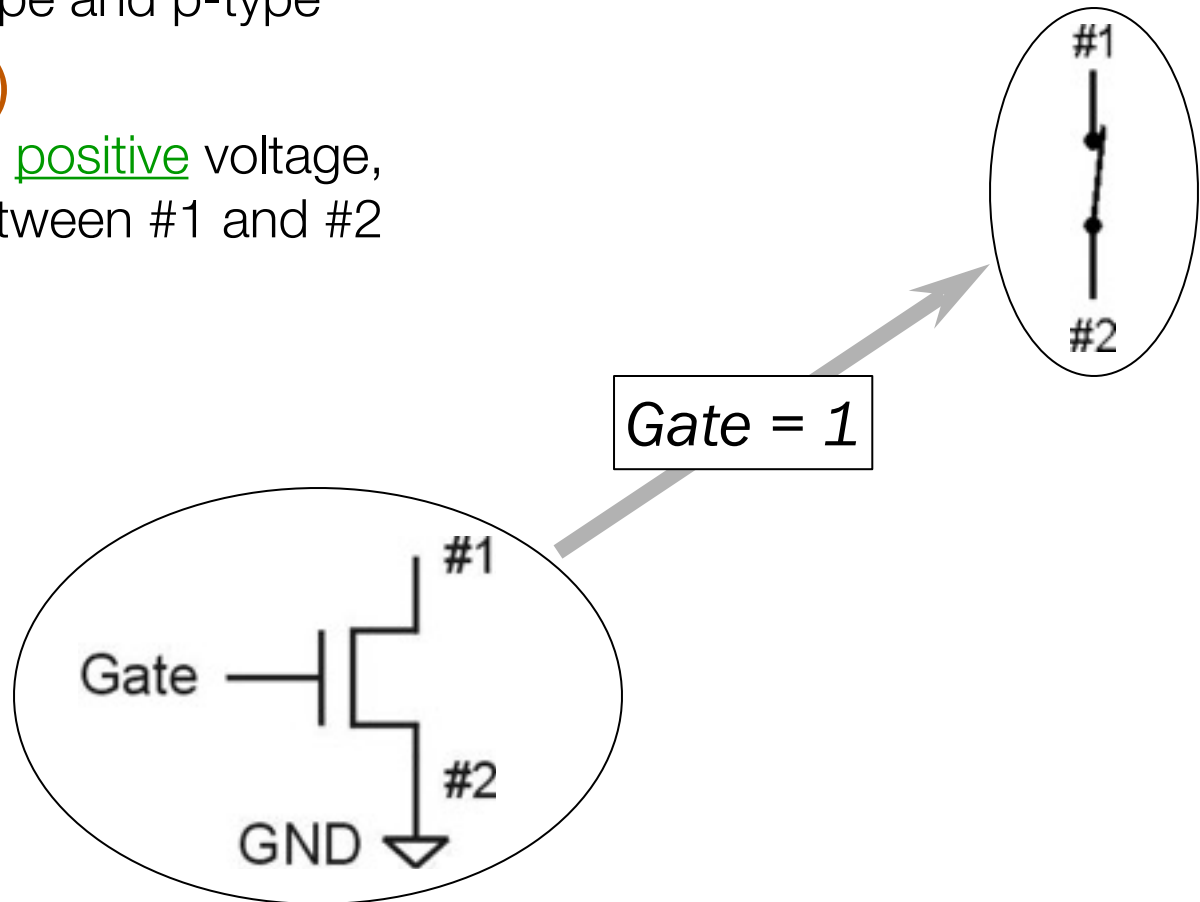
How is Addition Implemented in Hardware?

MOS = Metal Oxide Semiconductor

- two types: n-type and p-type

n-type (NMOS)

- when Gate has positive voltage, short circuit between #1 and #2 (switch closed)



Terminal #2 must be connected to GND (0V).

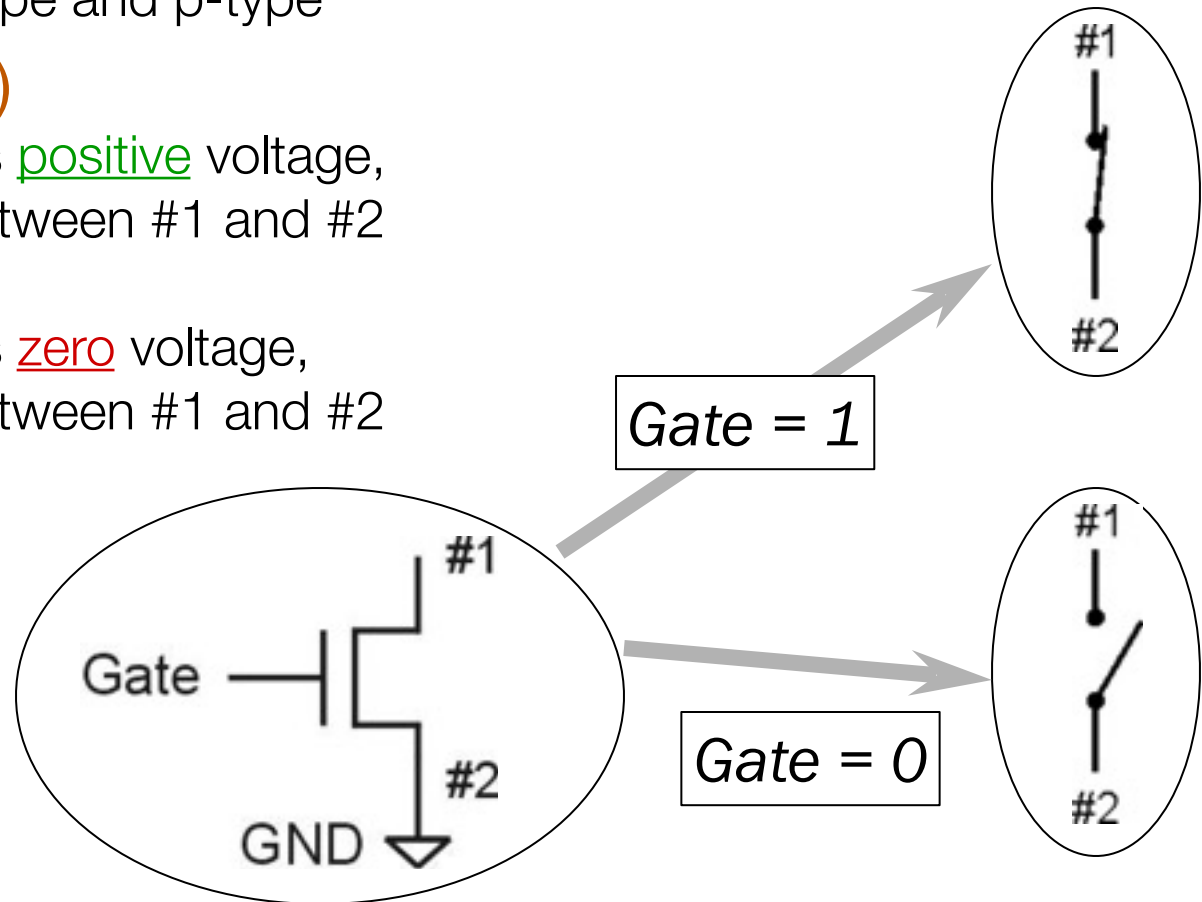
How is Addition Implemented in Hardware?

MOS = Metal Oxide Semiconductor

- two types: n-type and p-type

n-type (NMOS)

- when Gate has positive voltage, short circuit between #1 and #2 (switch closed)
- when Gate has zero voltage, open circuit between #1 and #2 (switch open)

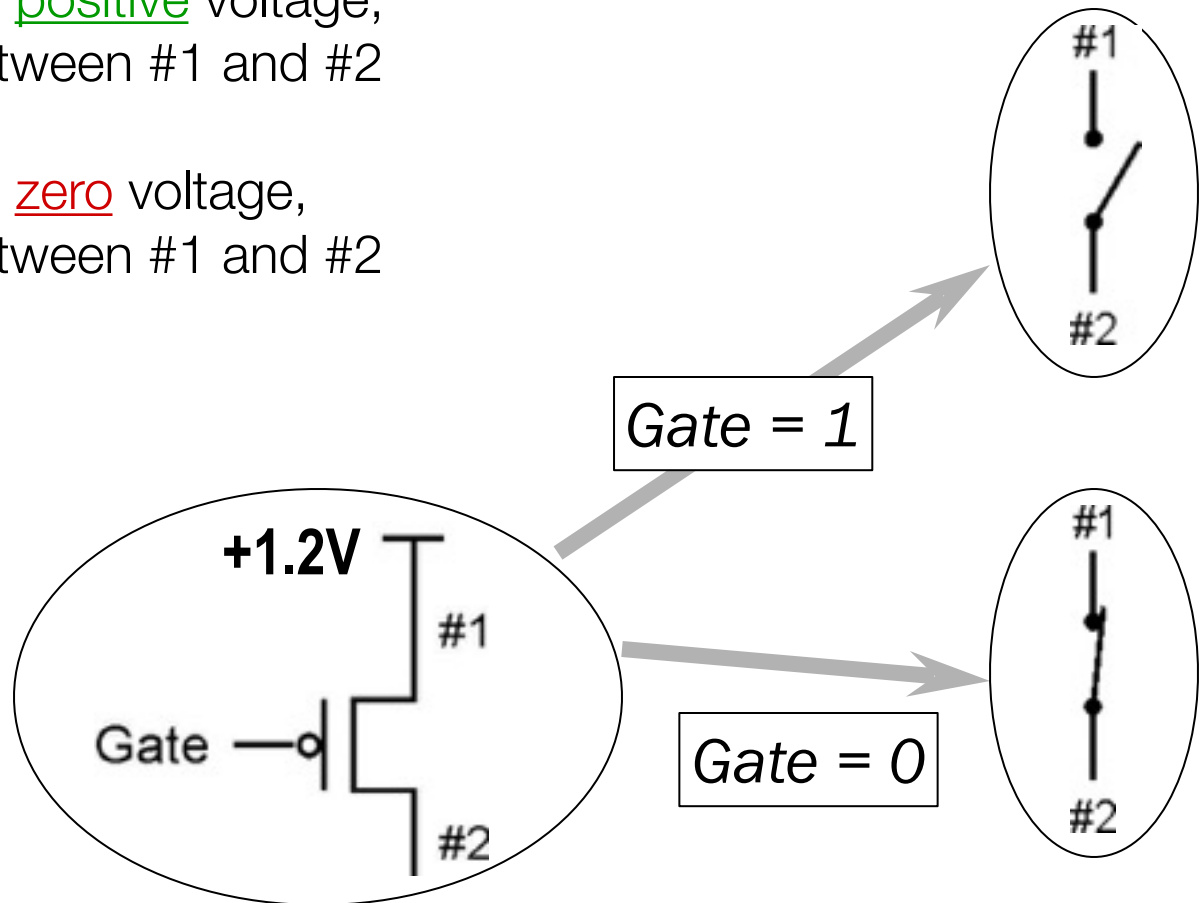


Terminal #2 must be connected to GND (0V).

How is Addition Implemented in Hardware?

p-type is *complementary* to n-type (**PMOS**)

- when Gate has positive voltage, open circuit between #1 and #2 (switch open)
- when Gate has zero voltage, short circuit between #1 and #2 (switch closed)

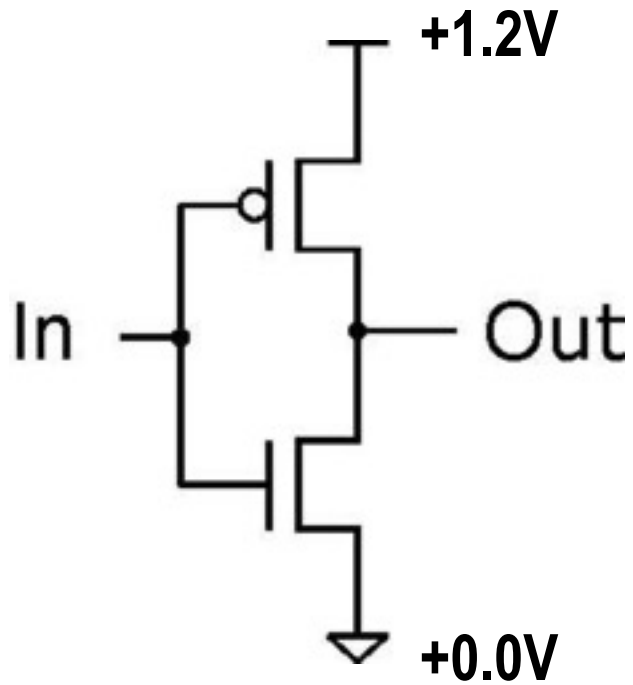


Terminal #1 must be connected to +1.2V

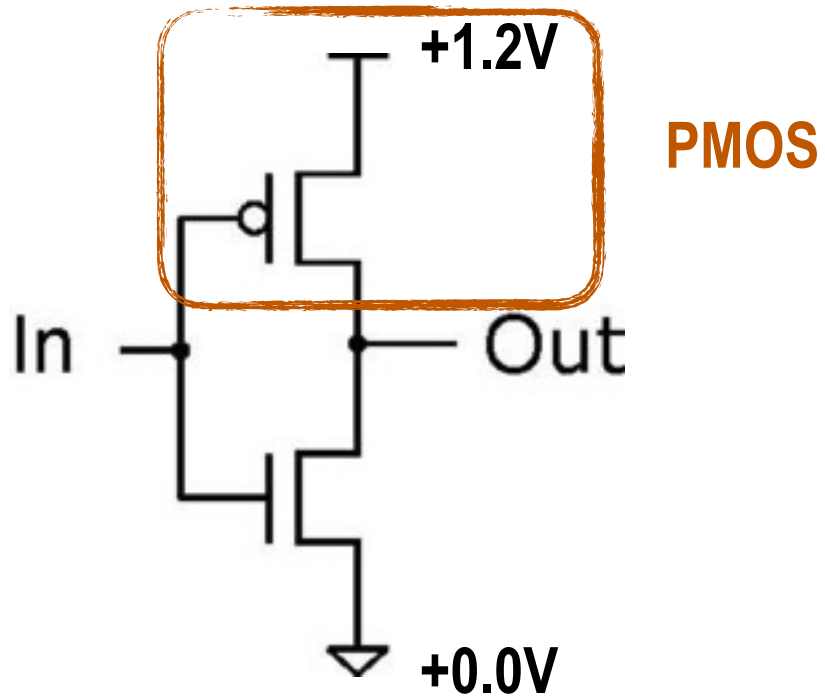
CMOS Circuit

- Complementary MOS
- Uses both n-type and p-type MOS transistors
 - p-type
 - Attached to + voltage
 - Pulls output voltage UP when input is zero
 - n-type
 - Attached to GND
 - Pulls output voltage DOWN when input is one

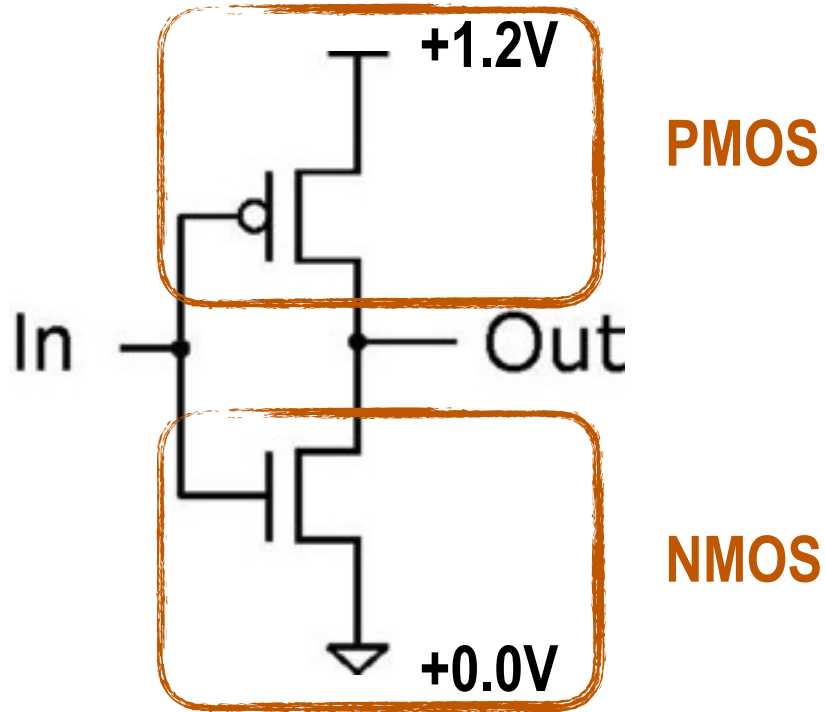
Inverter (NOT Gate)



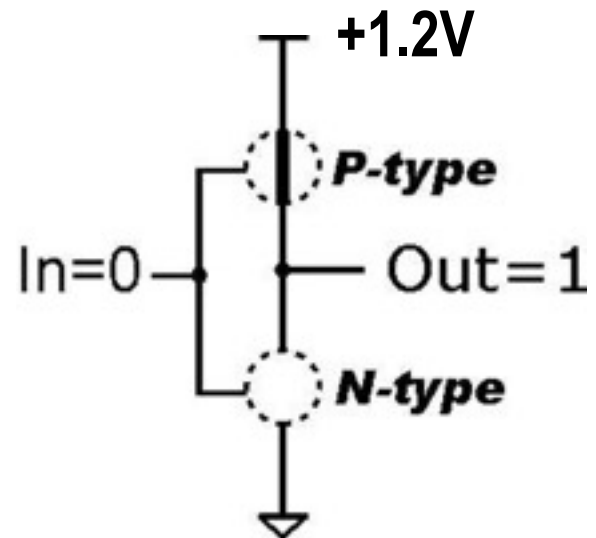
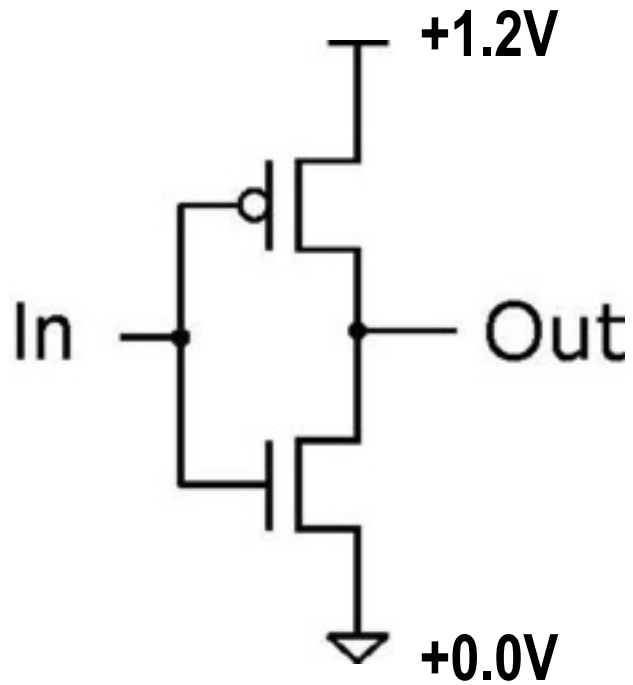
Inverter (NOT Gate)



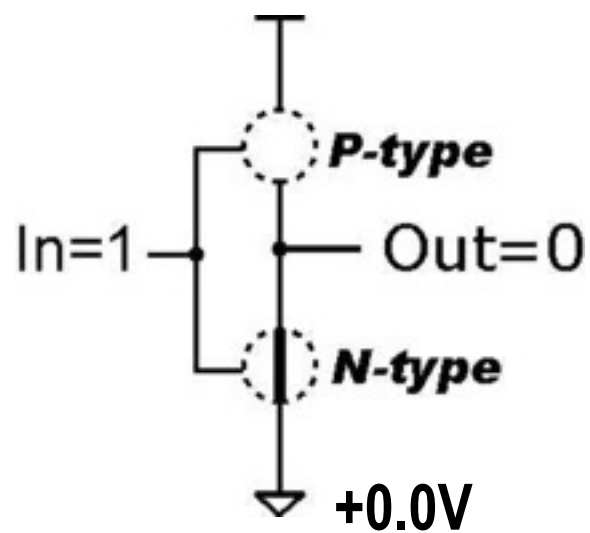
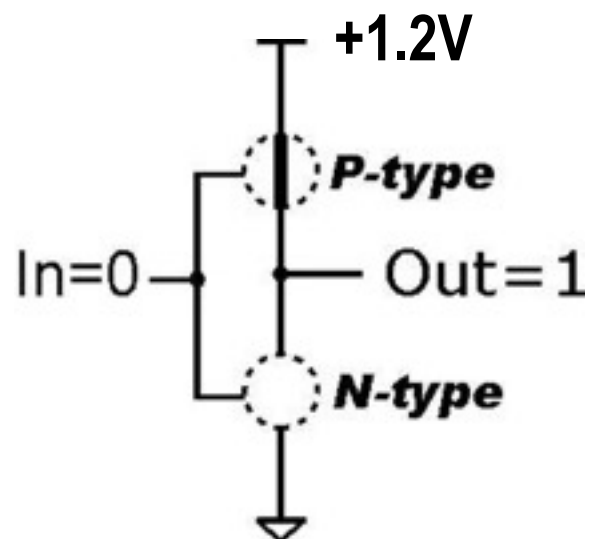
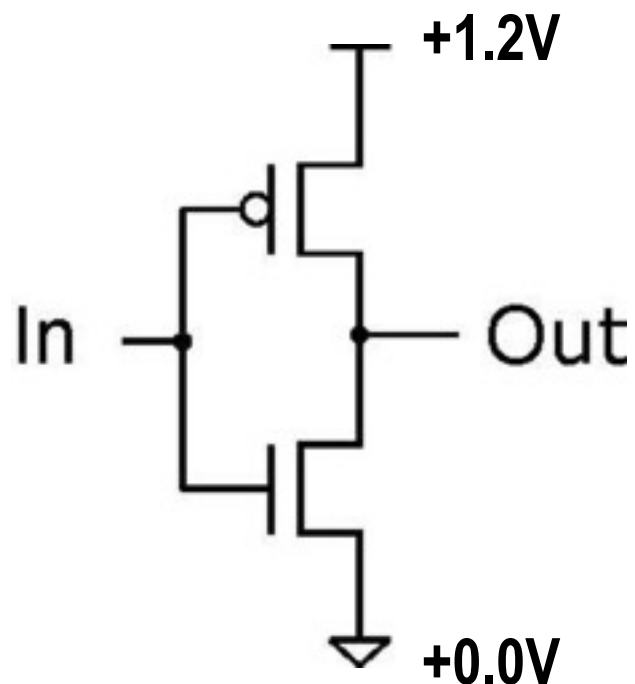
Inverter (NOT Gate)



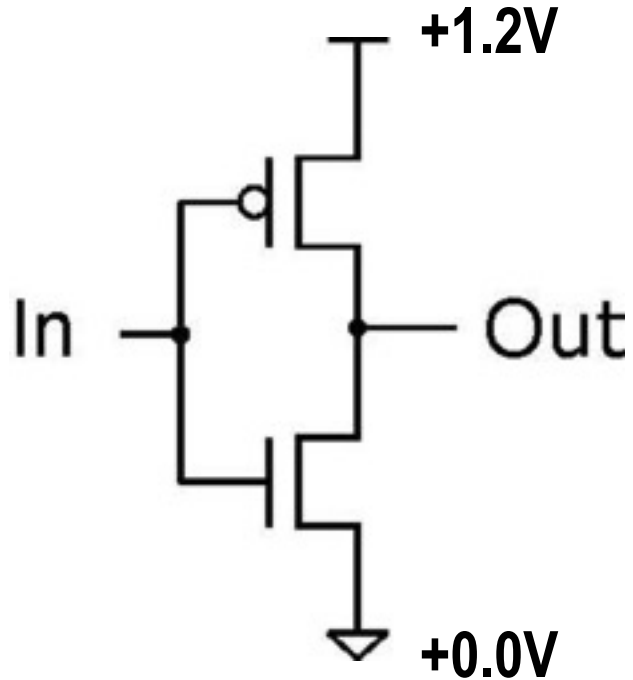
Inverter (NOT Gate)



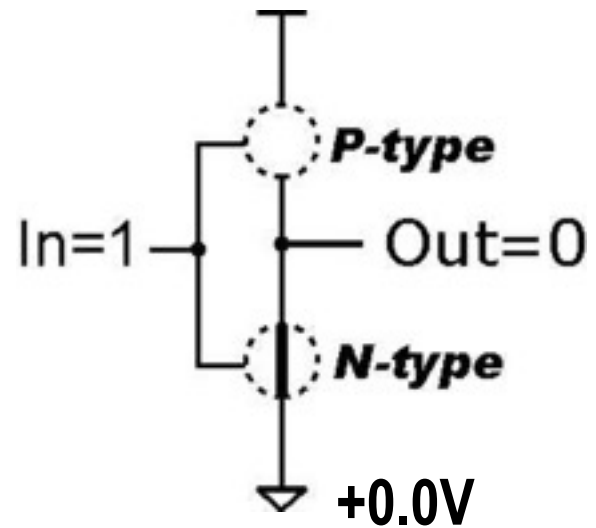
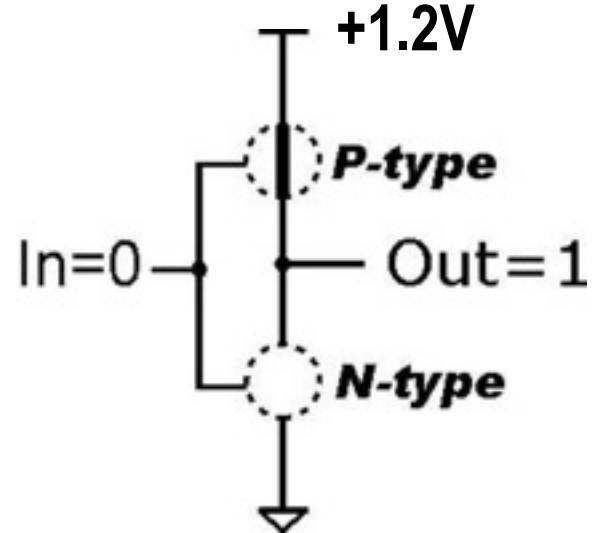
Inverter (NOT Gate)



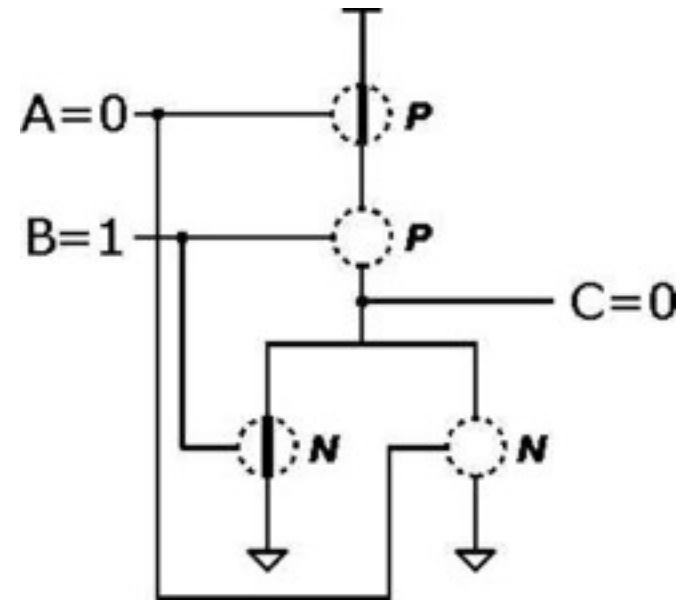
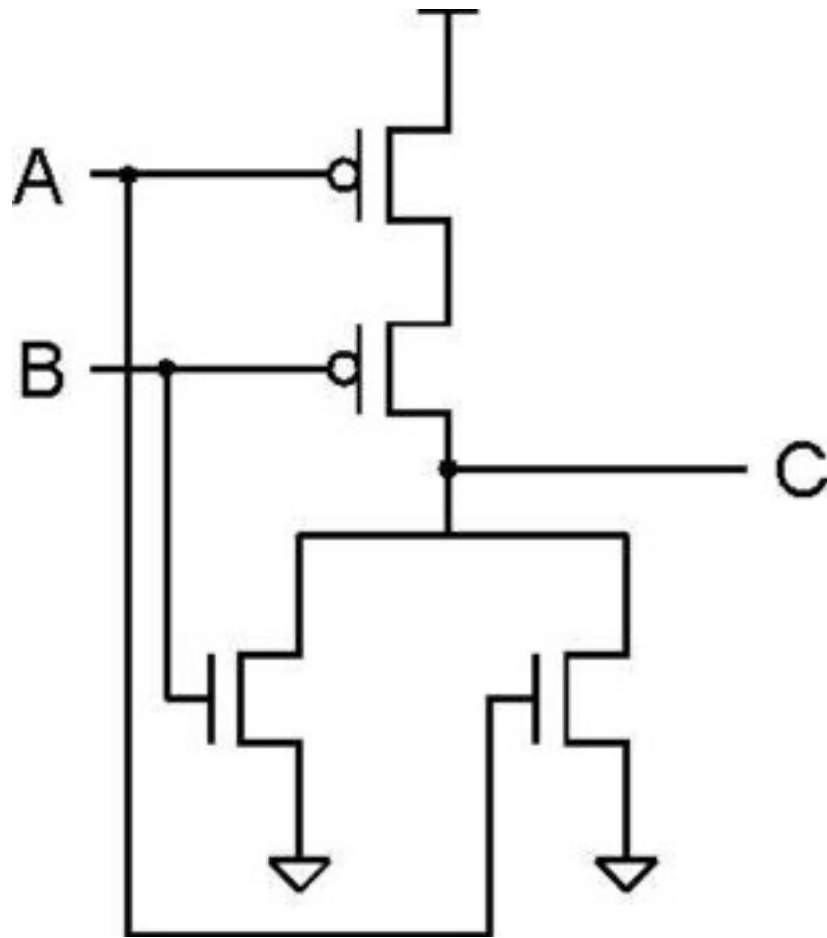
Inverter (NOT Gate)



In	Out
0	1
1	0



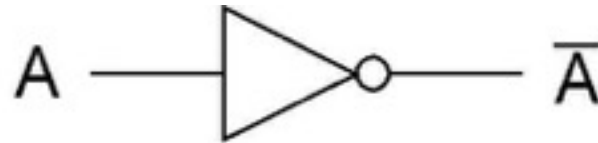
NOR Gate (NOT + OR)



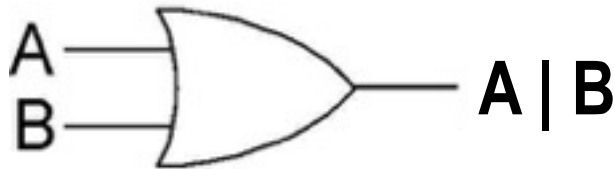
A	B	C
0	0	1
0	1	0
1	0	0
1	1	0

Note: Serial structure on top, parallel on bottom.

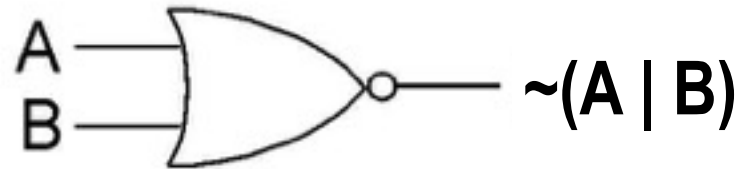
Basic Logic Gates



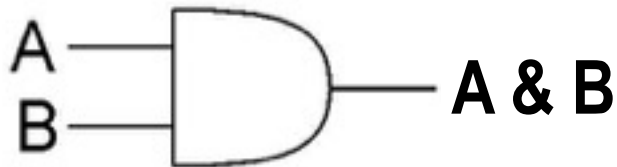
NOT



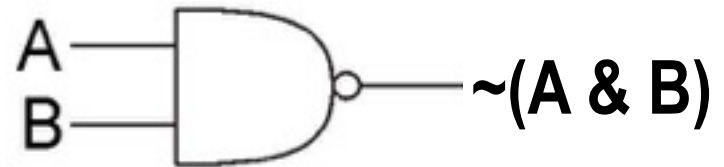
OR



NOR



AND



NAND

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.

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

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.

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

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.

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

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.

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

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.

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

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

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

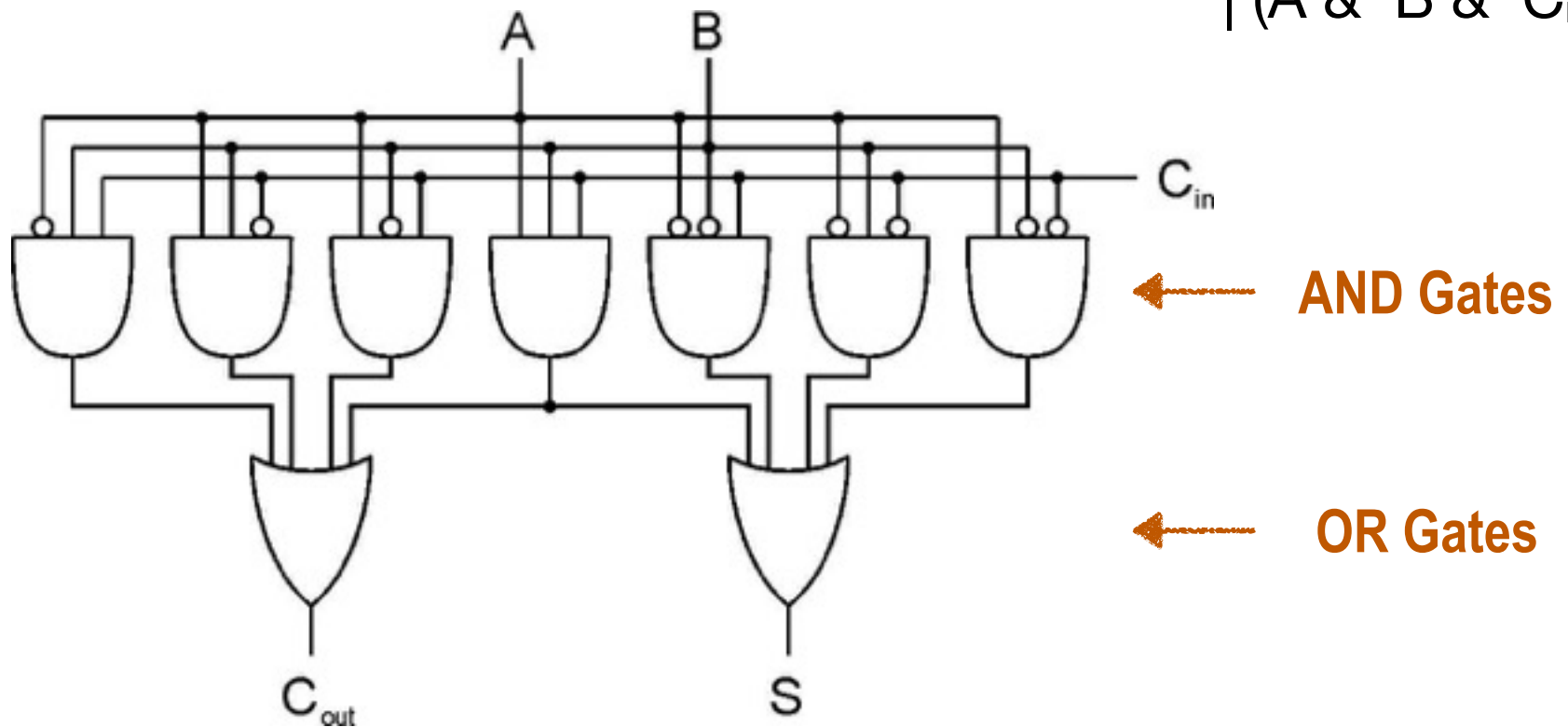
Add two bits and carry-in,
produce one-bit sum and carry-out.

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

Full (1-bit) Adder

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

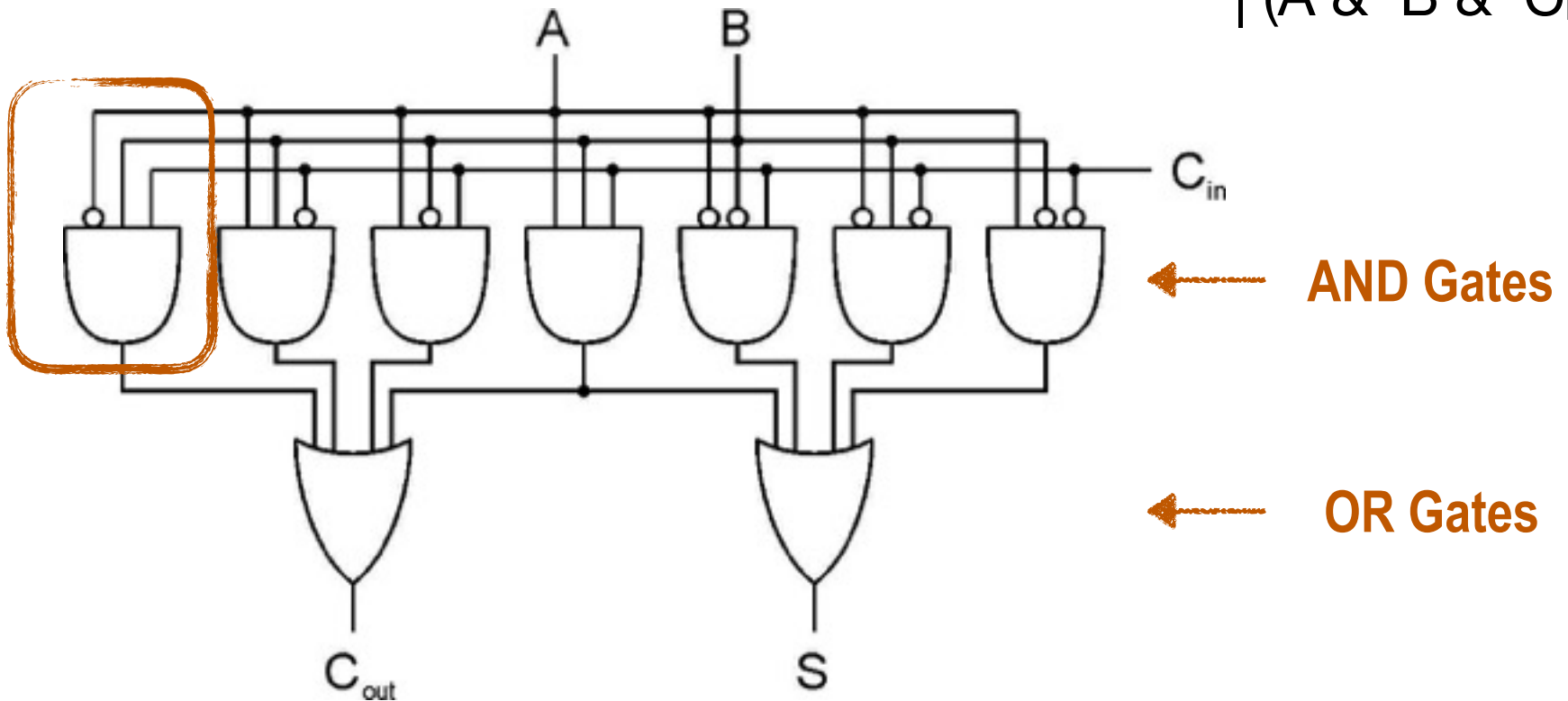
$$\begin{aligned} C_{ou} = & (\sim A \ \& \ B \ \& \ C_{in}) \\ & | (A \ \& \ \sim B \ \& \ C_{in}) \\ & | (A \ \& \ B \ \& \ \sim C_{in}) \\ & | (A \ \& \ B \ \& \ C_{in}) \end{aligned}$$



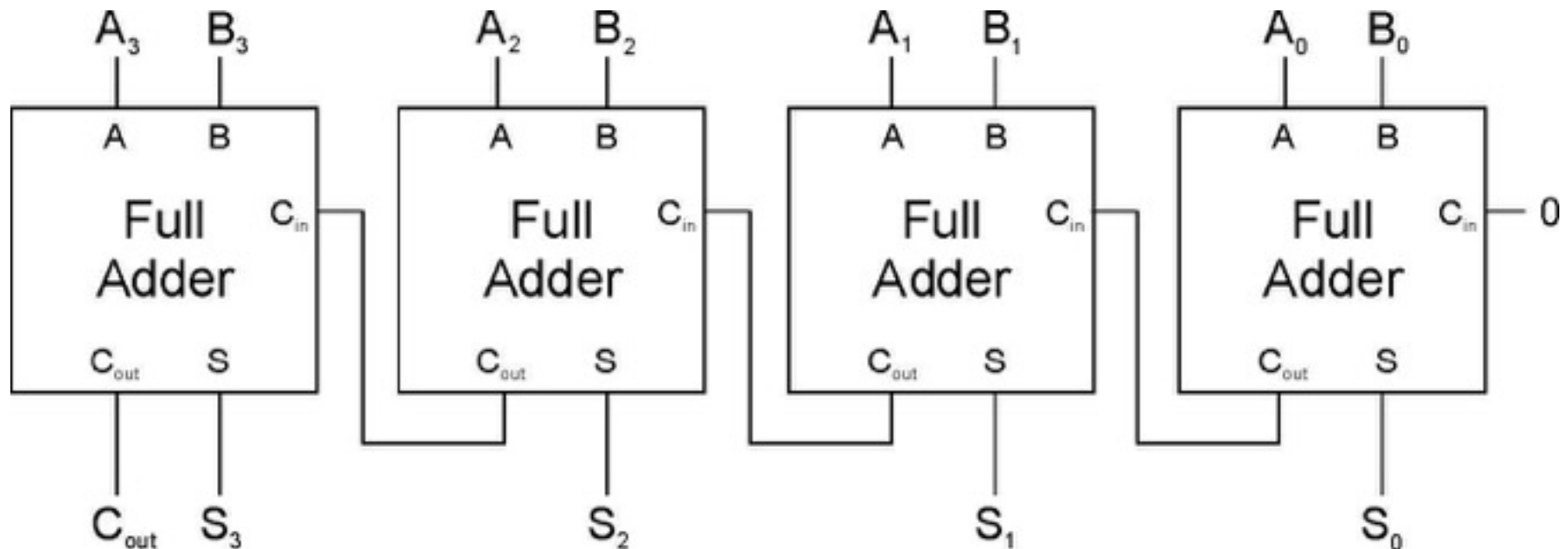
Full (1-bit) Adder

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

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

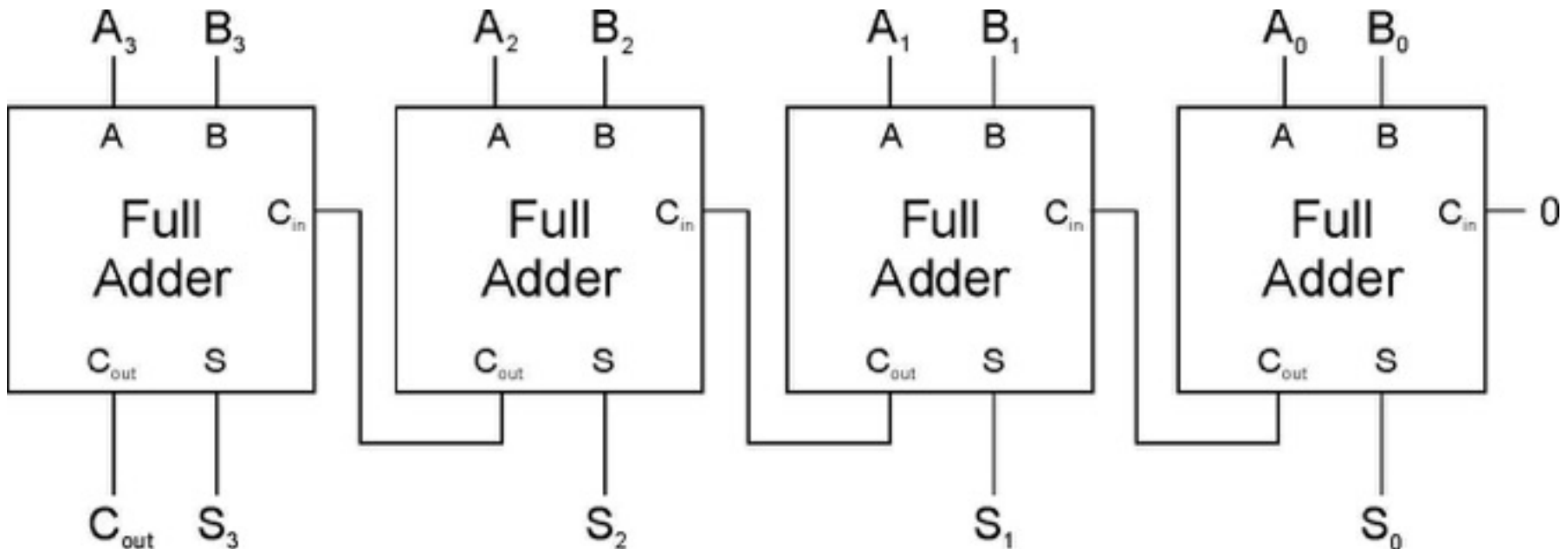


Four-bit Adder



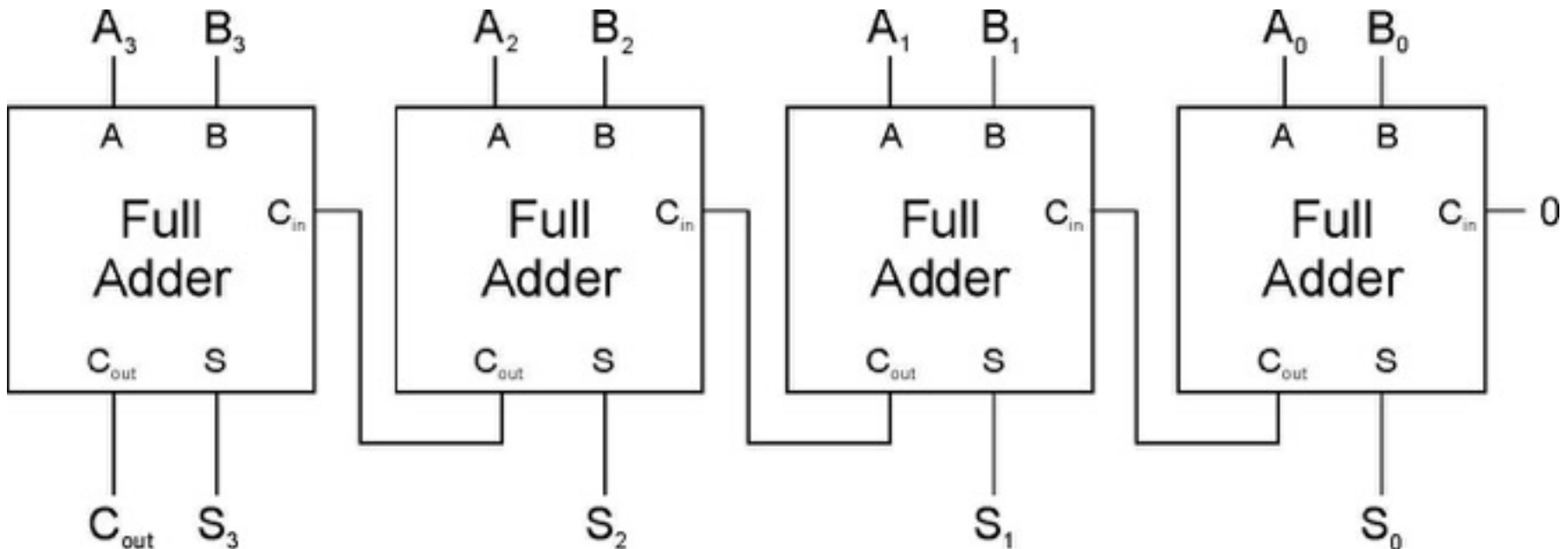
Four-bit Adder

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



Four-bit Adder

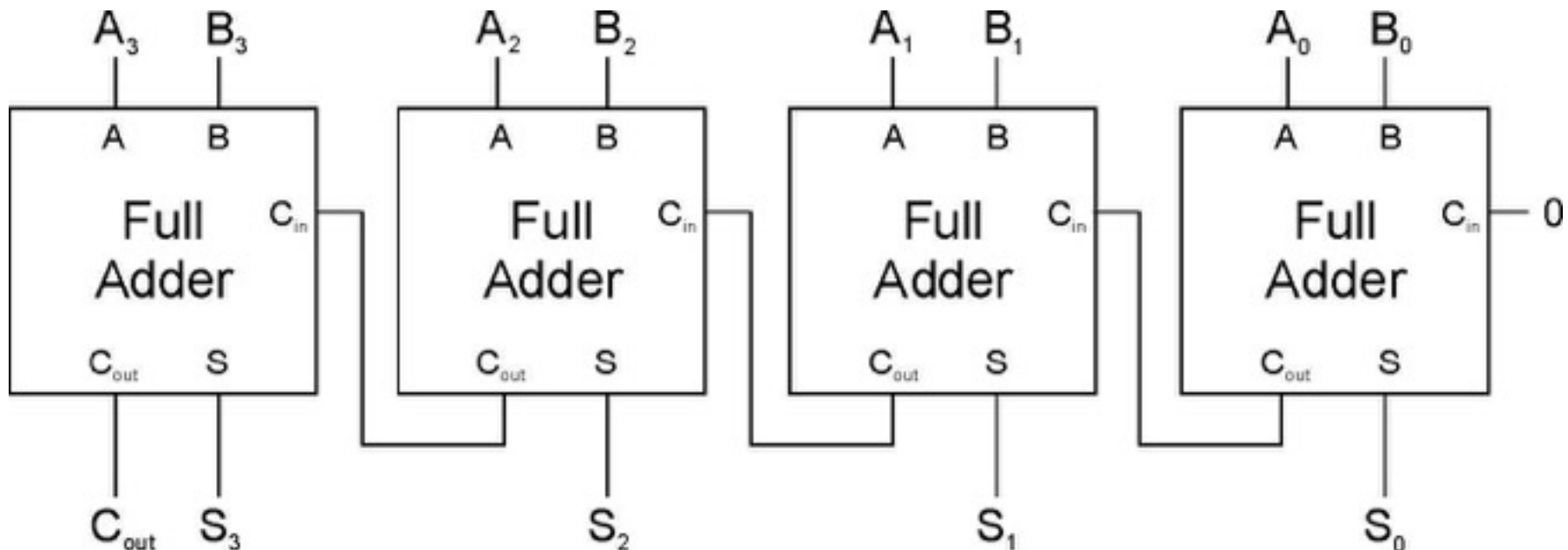
- Ripple-carry Adder
 - Simple, but performance linear to bit width
- Carry look-ahead adder (CLA)
 - Generate all carriers simultaneously



Four-bit Adder

Questions?

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



Multiplication

Multiplication

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

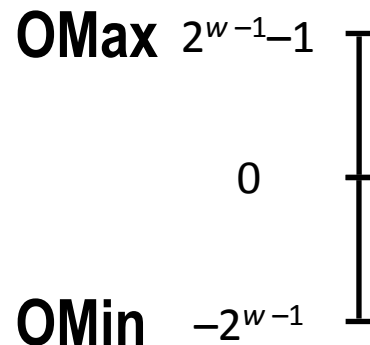
Multiplication

- Goal: Computing Product of w -bit numbers x, y
- But, exact results can be bigger than w bits

Multiplication

- Goal: Computing Product of w -bit numbers x, y
- But, exact results can be bigger than w bits

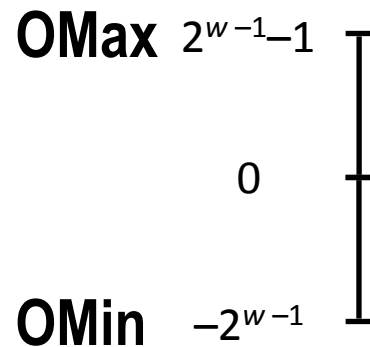
Original Number (w bits)



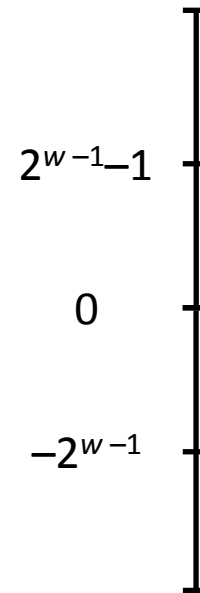
Multiplication

- Goal: Computing Product of w -bit numbers x, y
- But, exact results can be bigger than w bits

Original Number (w bits)



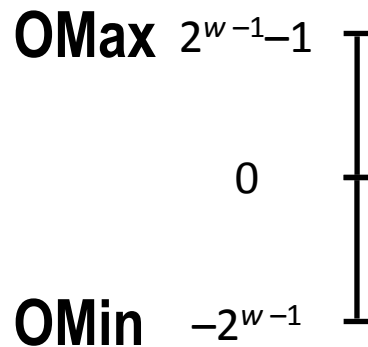
Product



Multiplication

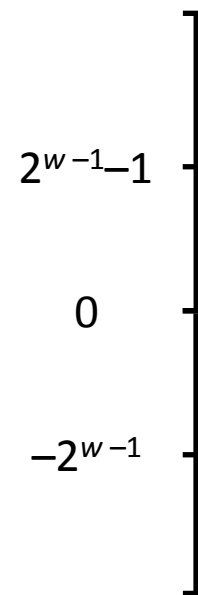
- Goal: Computing Product of w -bit numbers x, y
- But, exact results can be bigger than w bits

Original Number (w bits)



Product

PMax

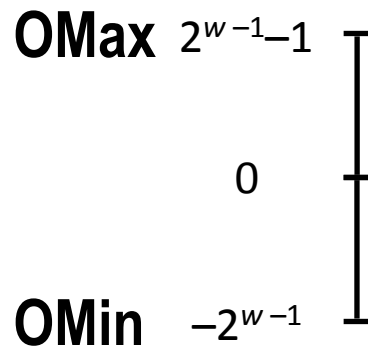


PMin

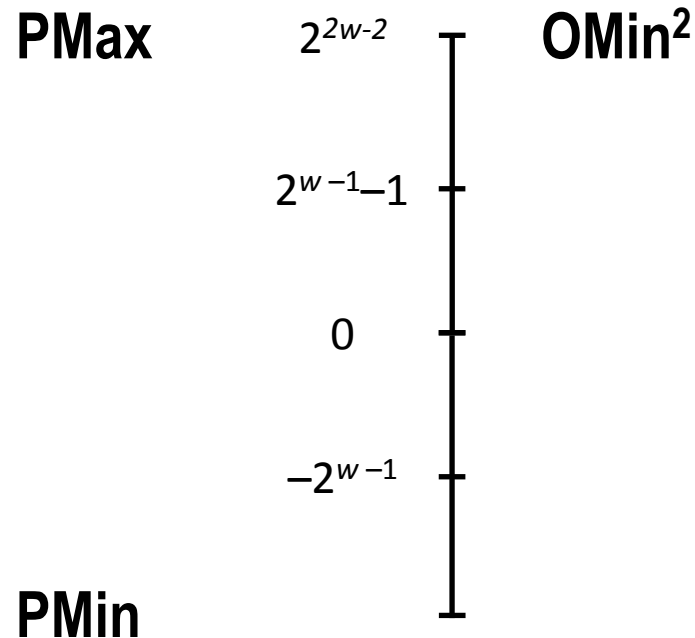
Multiplication

- Goal: Computing Product of w -bit numbers x, y
- But, exact results can be bigger than w bits

Original Number (w bits)



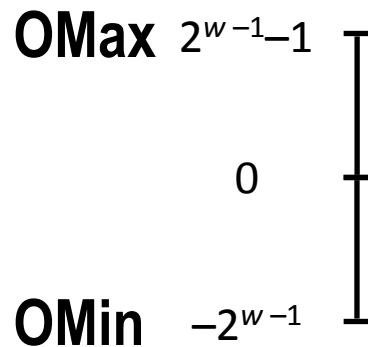
Product



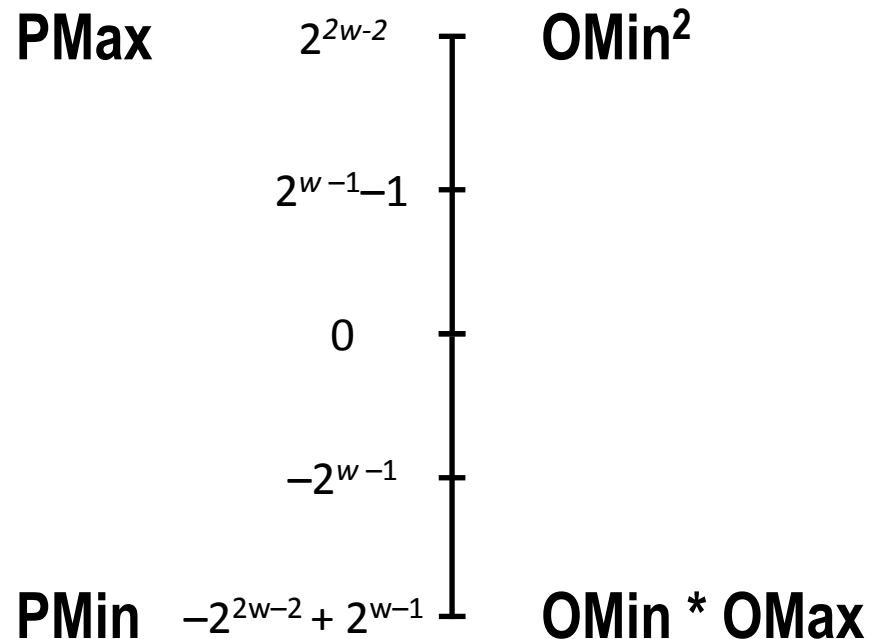
Multiplication

- Goal: Computing Product of w -bit numbers x, y
- But, exact results can be bigger than w bits

Original Number (w bits)



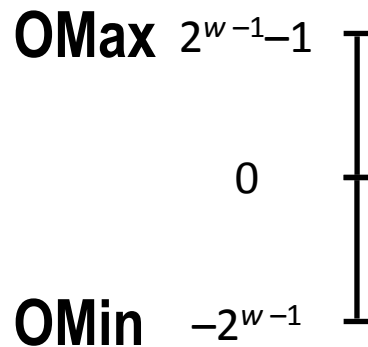
Product



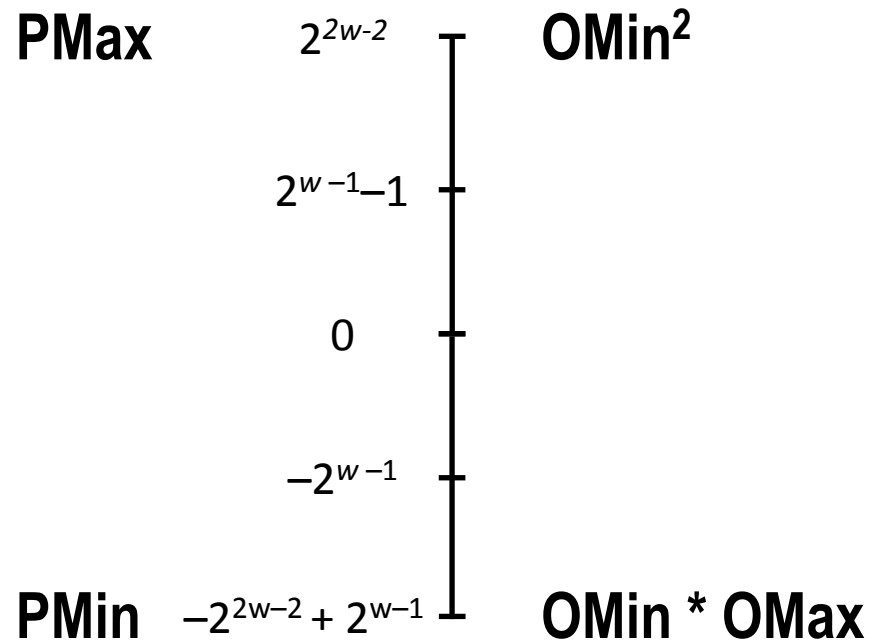
Multiplication

- Goal: Computing Product of w -bit numbers x, y
- But, exact results can be bigger than w bits

Original Number (w bits)



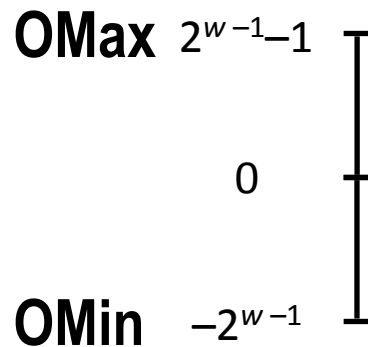
Product ($2w$ bits)



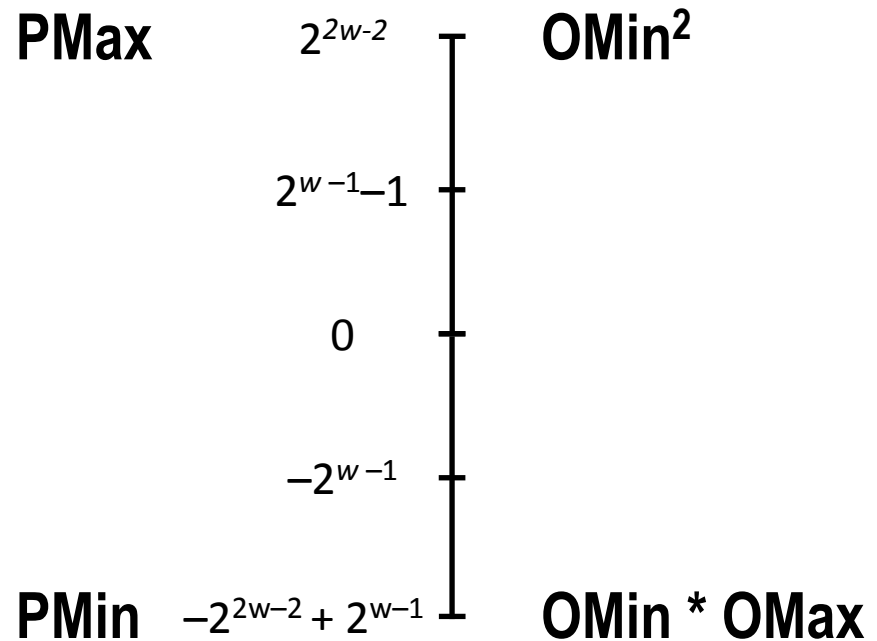
Multiplication

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

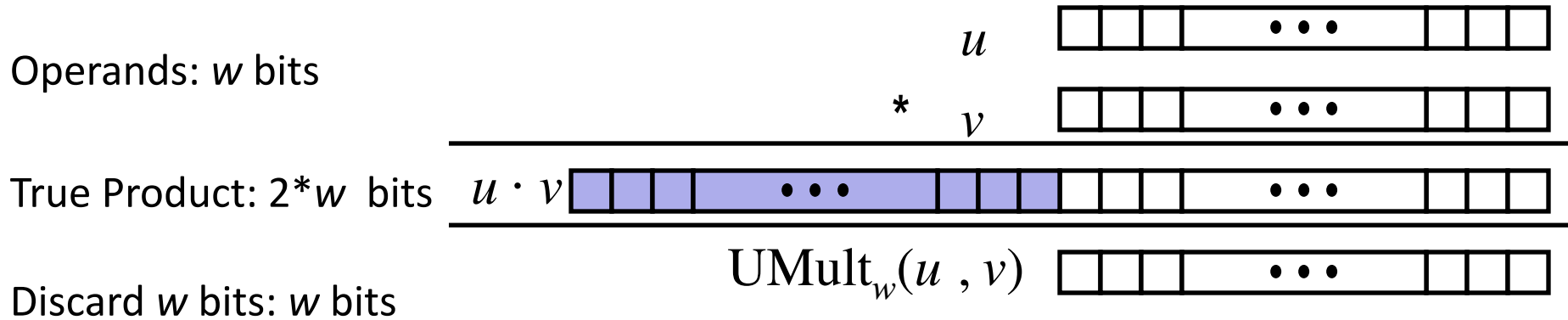
Original Number (w bits)



Product ($2w$ bits)



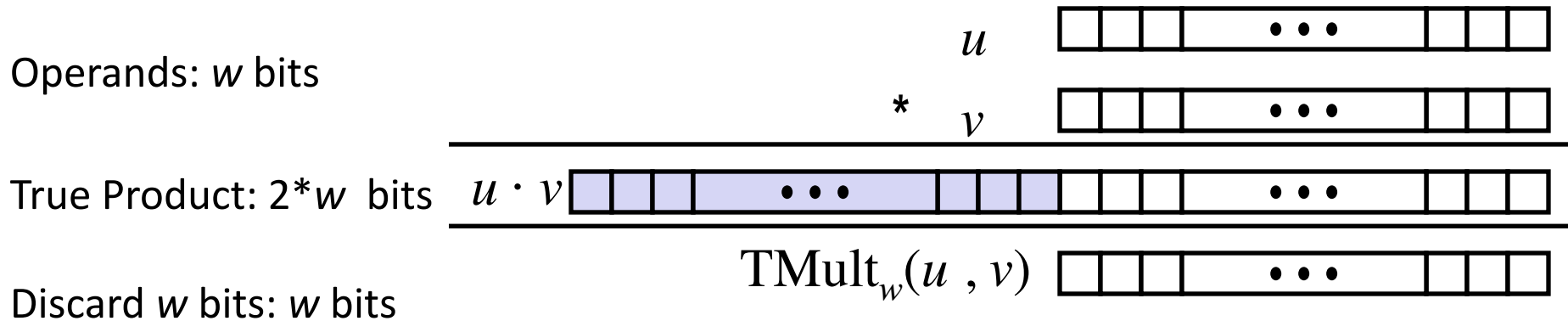
Unsigned Multiplication in C



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

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

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

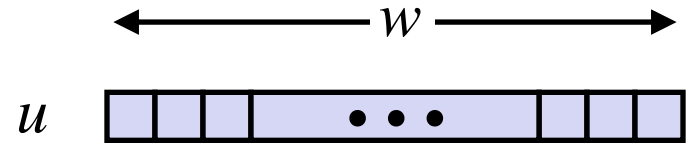
- Operation

- $u \ll k$ 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



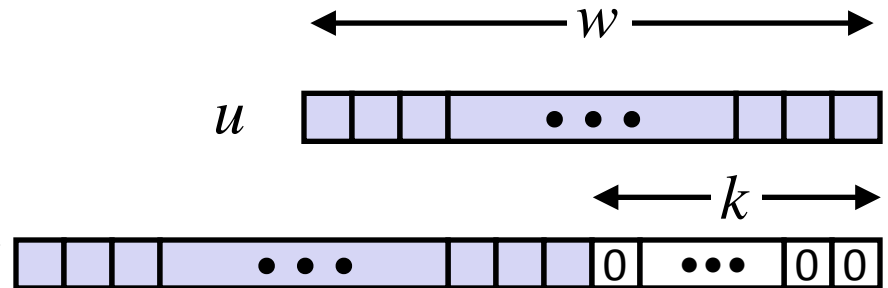
Power-of-2 Multiply with Shift

- 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

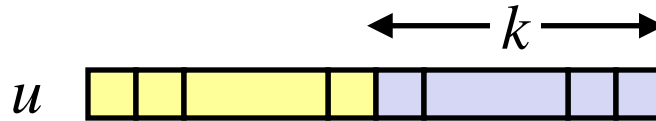
- Compiler generates this code automatically
- $u \ll 3 == u * 8$
- $(u \ll 5) - (u \ll 3) == u * 24$

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

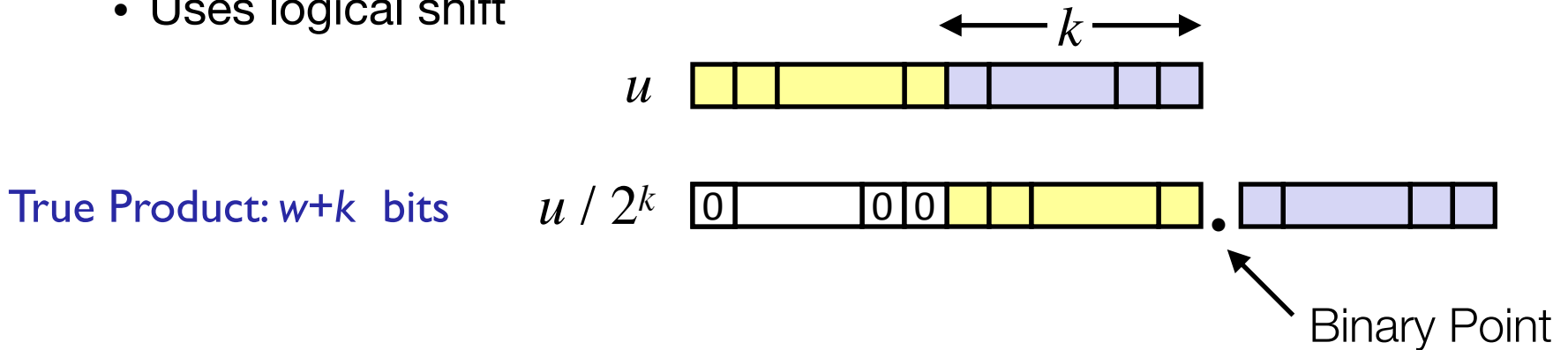
Unsigned Power-of-2 Divide with Shift

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



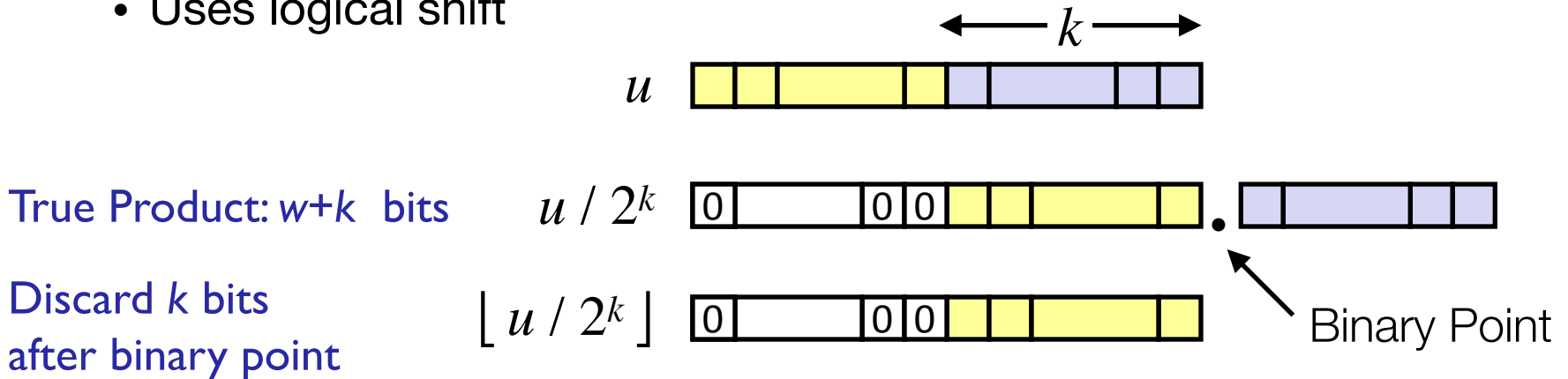
Unsigned Power-of-2 Divide with Shift

- 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

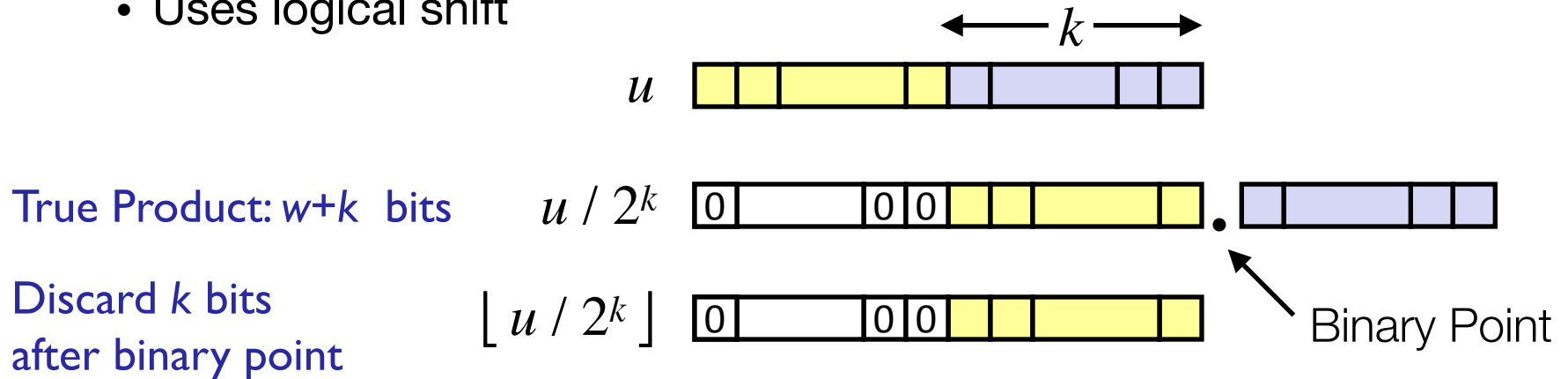


Unsigned Power-of-2 Divide with Shift

- Implement power-of-2 divide with shift

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

- Uses logical shift

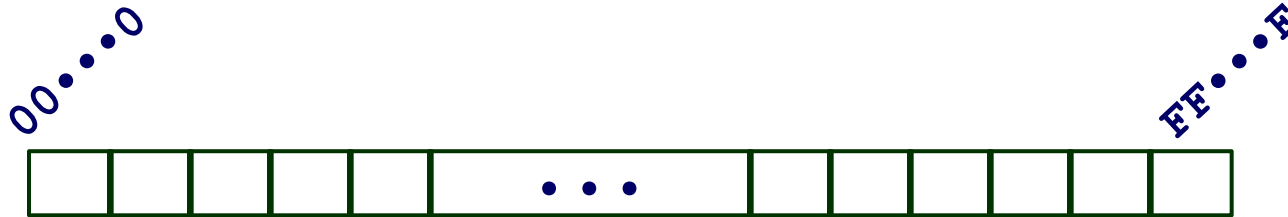


- $234_{10} \gg 2 = 2.34_{10}$, truncated result is 2 ($\lfloor 2.34 \rfloor = 2$)
- $1101_2 \gg 2 = 0011_2$ (true result: 11.01_2 . $\lfloor 13 / 4 \rfloor = 3$)

Today: Representing Information in Binary

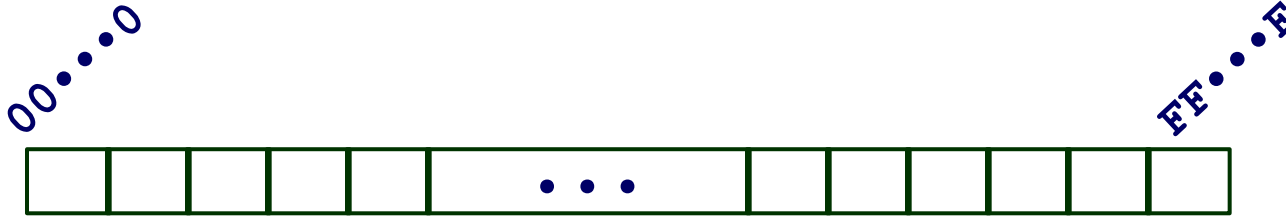
- Why Binary (bits)?
- Bit-level manipulations
- Integers
 - Representation: unsigned and signed
 - Conversion, casting
 - Expanding, truncating
 - Addition, negation, multiplication, shifting
 - Summary
- Representations in memory, pointers, strings

Byte-Oriented Memory Organization



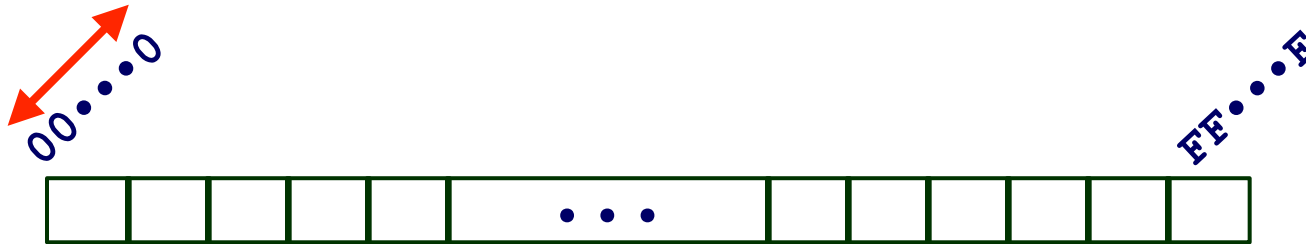
- Programs refer to data by address
 - Conceptually, envision it as a very large array of bytes: **byte-addressable**
 - In reality, it's not, but can think of it that way
 - An address is like an index into that array
 - and, a pointer variable stores an address

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 18.4×10^{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}

Example Data Representations (in Bytes)

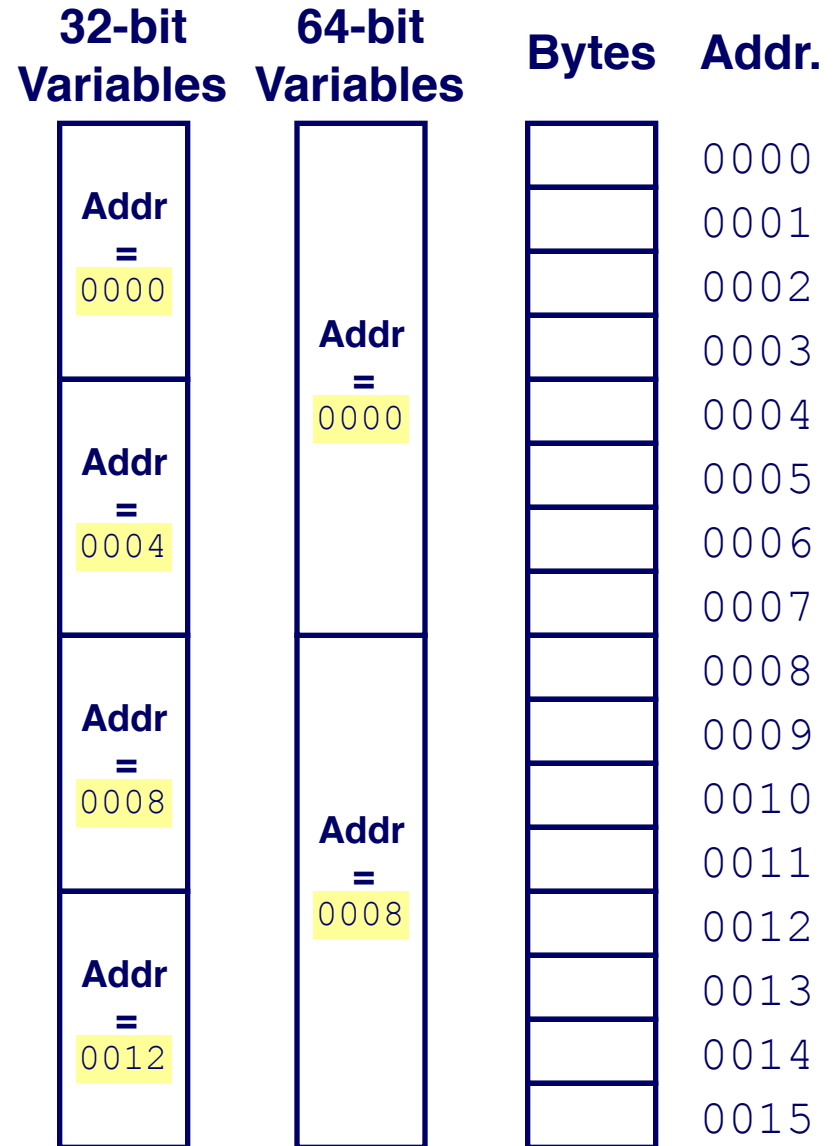
Word Size	4	8
C Data Type	32-bit	64-bit
<code>char</code>	1	1
<code>short</code>	2	2
<code>int</code>	4	4
<code>long</code>	4	8
<code>float</code>	4	4
<code>double</code>	8	8
<code>pointer</code>	4	8

Example Data Representations (in Bytes)

Word Size	4	8
C Data Type	32-bit	64-bit
<code>char</code>	1	1
<code>short</code>	2	2
<code>int</code>	4	4
<code>long</code>	4	8
<code>float</code>	4	4
<code>double</code>	8	8
<code>pointer</code>	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)



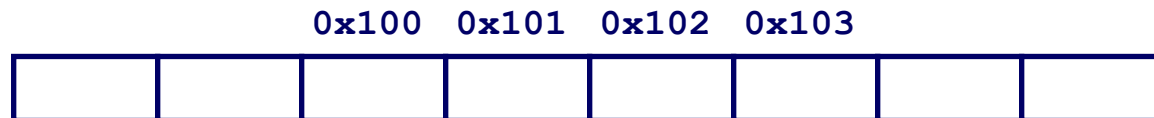
Byte Ordering

Byte Ordering

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

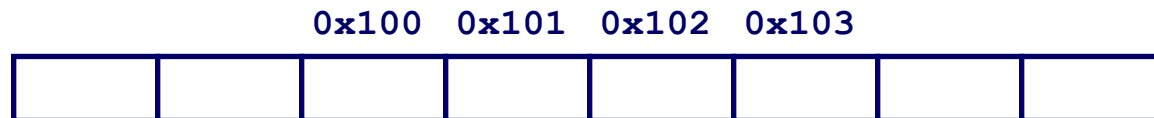
Byte Ordering

- 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

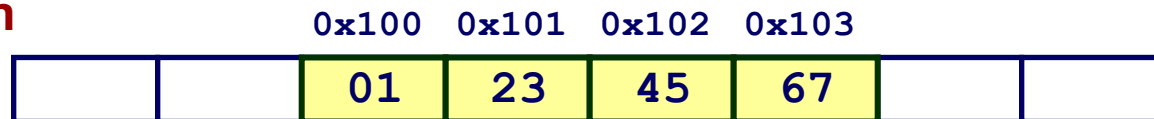
- How are the bytes within a multi-byte word ordered in memory?
- Example
 - Variable x has 4-byte value of 0x01234567
 - Address given by &x is 0x100
- Conventions
 - **Big Endian**: Sun, PPC Mac, IBM z, Internet
 - Most significant byte has lowest address (**MSB first**)
 - **Little Endian**: x86,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

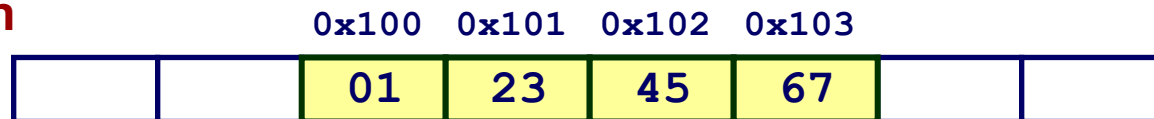
Big Endian



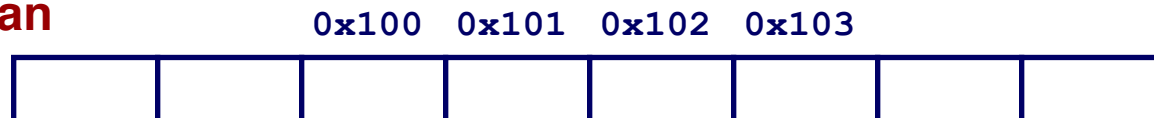
Byte Ordering

- 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**)

Big Endian



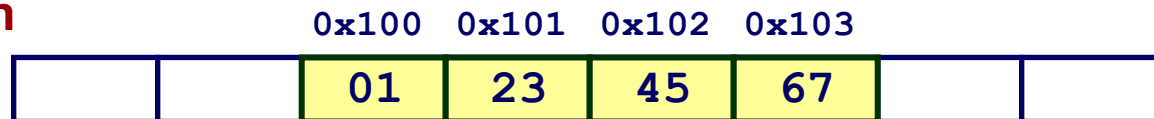
Little Endian



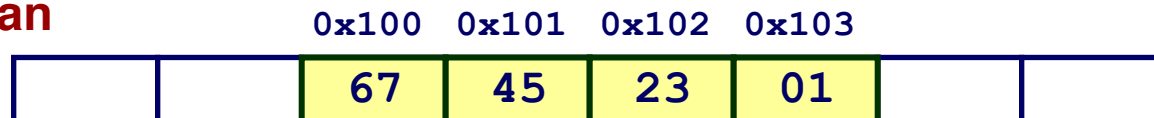
Byte Ordering

- 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**)

Big Endian



Little Endian



Representing Integers

Hex: 00003B6D

Hex: FFFFC493

`int A = 15213;`

`int B = -15213;`

Address Increase
↓

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

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

Representing Integers

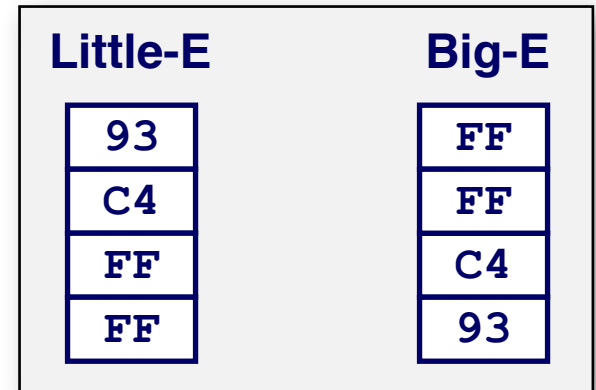
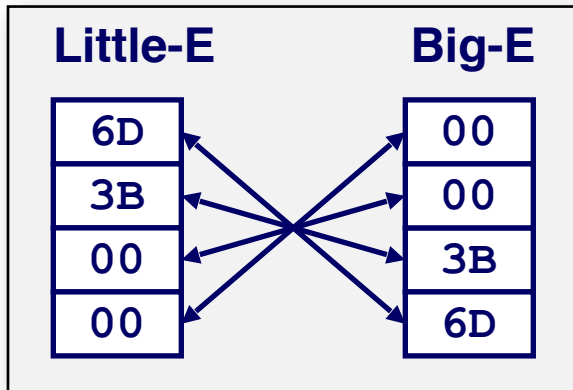
Hex: 00003B6D

Hex: FFFFC493

`int A = 15213;`

`int B = -15213;`

Address Increase
↓



Representing Integers

Hex: 00003B6D

Hex: FFFFC493

`int A = 15213;`

`int B = -15213;`

Address Increase
↓

