CSC 252: Computer Organization
Fall 2021: Lecture 3

Arithmetic Overflow
Fixed point numbers
Converting between signed and unsigned
Binary arithmetic with logic gates

Instructor: Alan Beadle

Department of Computer Science
University of Rochester

Announcement

Programming assignment 1 is in C language. Seek help from TAs.

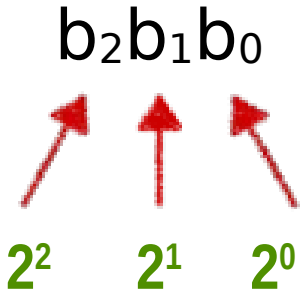
TAs are best positioned to answer your questions about programming assignments!!!

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.

Encoding Negative Numbers

Two’s Complement, give negative weight to MSB

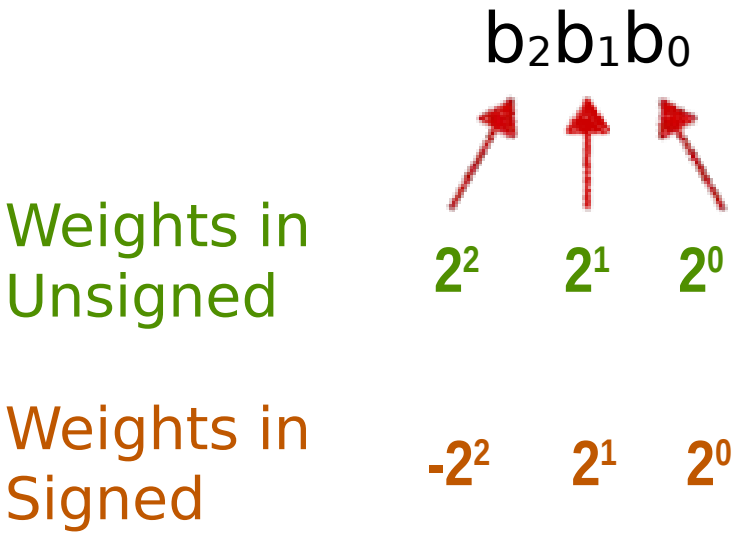
Weights in Unsigned



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

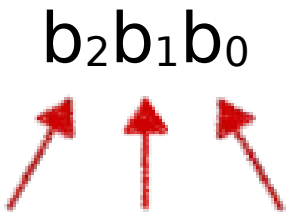
Two’s Complement, give negative weight to MSB



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, give negative weight to MSB



Weights in Unsigned

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

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

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

Two-Complement Implications

Only 1 zero

There is (still) a bit that represents sign!

Unsigned arithmetic still works

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

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

Two-Complement Implications

Only 1 zero

There is (still) a bit that represents sign!

Unsigned arithmetic still works

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

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

With only 3 bits, 3 + 1 becomes -4 because we don't have enough bits to hold the correct answer

(This is called overflow. More on it later.)

```
   011          3
+) 001       +)  1
-----        -----
   100          -4
```

Can We Represent Fractions in Binary?

Yes, in several ways.

Here is “Fixed point” notation:

What does 10.01_2 mean?

$12.45 = 1*10^1 + 2*10^0 + 4*10^{-1} + 5*10^{-2}$

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

Integer Arithmetic Still Works!

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

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

Fixed-Point Representation

Fixed interval between two representable numbers as long as the binary point stays fixed

Each bit represents 0.25_{10}

Fixed-point representation of numbers

Next lecture will cover floating point, which is more useful and actually corresponds to C types

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

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

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

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
- 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)

Select a data type that captures all these attributes:

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)

Select a data type that captures all these attributes:

- **`unsigned char`**

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
- Why bother with unsigned variables?

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
- Why bother with unsigned variables?
  - Sometimes you need that extra bit for larger positive values
  - Even when you don't, it makes things more clear and may prevent strange bugs!

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

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;
```

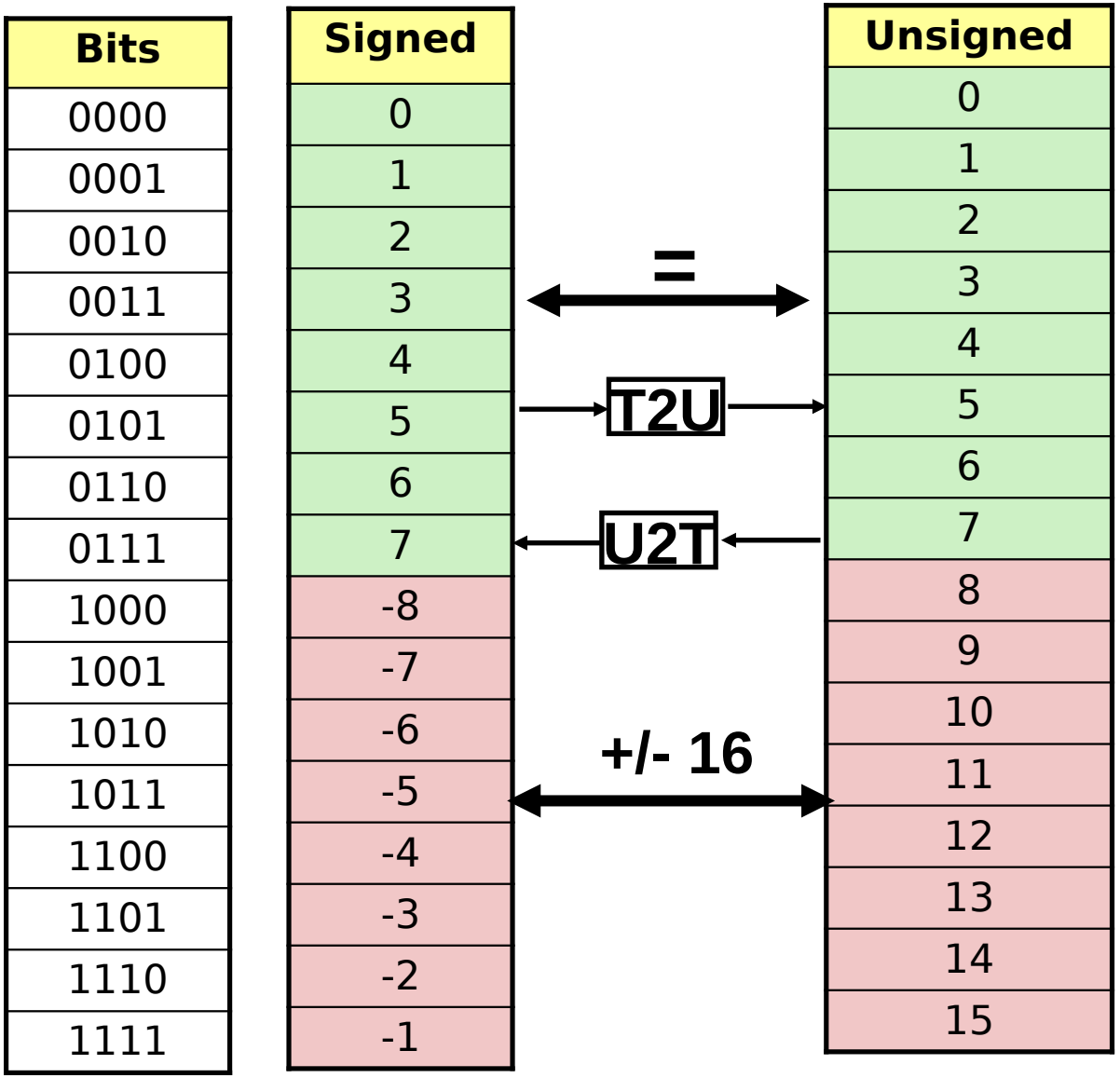
Mapping Between Signed & Unsigned

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

This is "correct" for positive values at least

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

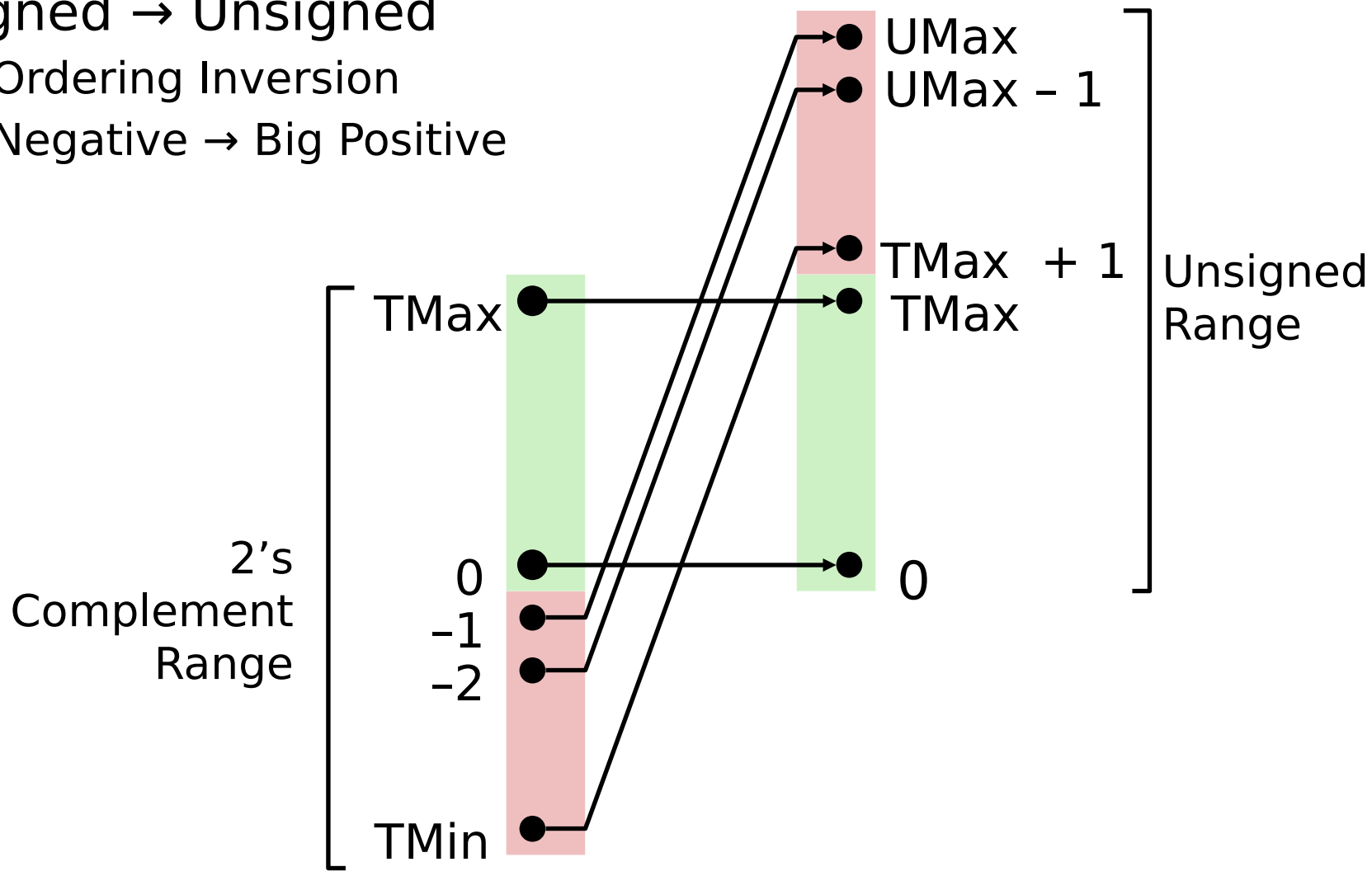
Mapping Signed ↔ Unsigned



Conversion Visualized

Signed → Unsigned

- Ordering Inversion
- Negative → Big Positive



What about converting sizes?

```
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

What about converting sizes?

```
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

We need to make sure the sign is correct

Signed Extension

Task:

Given *w*-bit signed integer *x*

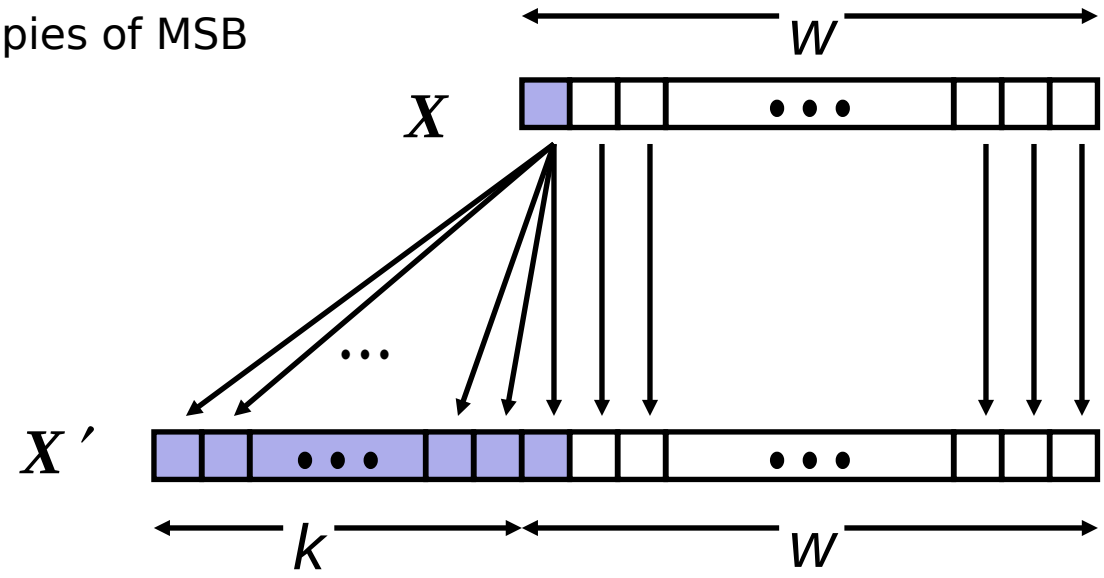
Convert it to (*w*+*k*)-bit integer with same value

Rule:

Make *k* copies of sign bit:

$X' = \underbrace{x_{w-1}, \ldots, x_{w-1}}, x_{w-1}, x_{w-2}, \ldots, x$

k copies of MSB



What about converting sizes?

```
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

We need to make sure the sign is correct

	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

What about extending unsigned numbers?

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

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

Unsigned (Zero) Extension

Task:

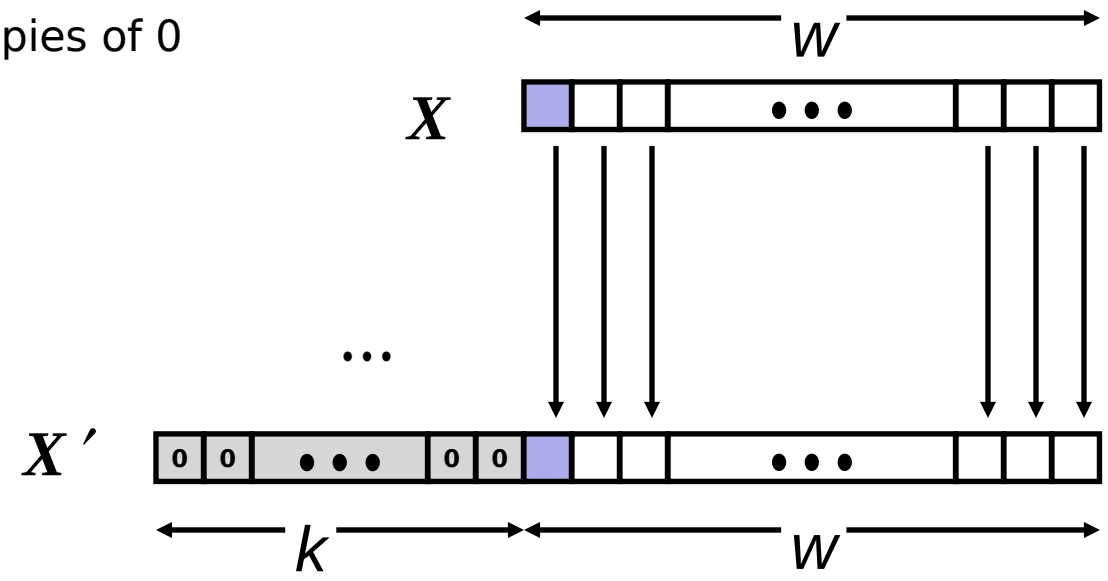
Given *w*-bit unsigned integer *x*

Convert it to (*w*+*k*)-bit integer with same value

Rule:

Simply pad zeros:

$X' = \underbrace{0,\ldots,0}_{k \text{ copies of } 0}, x_{w-1}, x_{w-2}, \ldots, x_0$



What about truncating?

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

What about truncating?

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

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

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

Unsigned Addition

Similar to Decimal Addition

Suppose we have a new data type that is 3-bits wide

Normal Case

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

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-bits wide

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

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

Normal Case

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

Overflow Case

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

True Sum

Unsigned Addition

Similar to Decimal Addition

Suppose we have a new data type that is 3-bits wide

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

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

Normal Case

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

Overflow Case

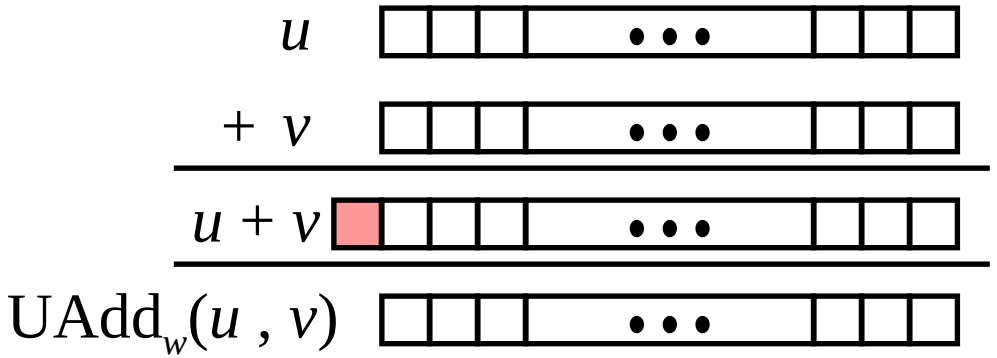
```
   110        6
+) 101     +) 5
-------    -------
  1011       11     True Sum
   011        3     Sum with same bits
```

Unsigned Addition in C

Operands: *w* bits

True Sum: *w*+1 bits

Discard Carry: *w* bits



Two's Complement Addition

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

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

Normal Case

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

Overflow Case

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

Negative Overflow

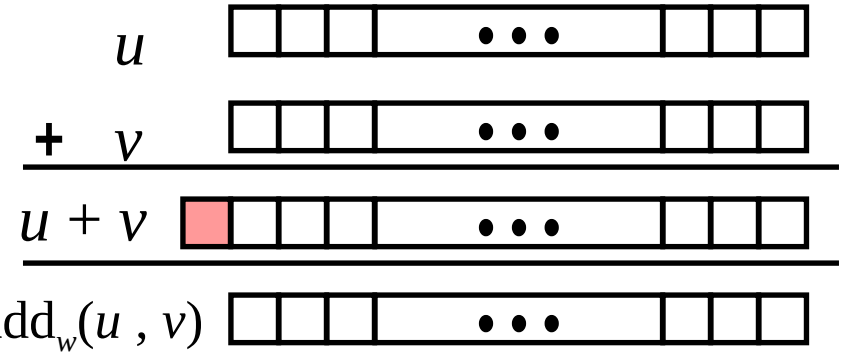
Positive Overflow

Two’s Complement Addition in C

Operands: w bits

True Sum: w+1 bits

Discard Carry: w bits



Is This an Overflow?

(signed addition)

```
   111
+) 110
  1101
```

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

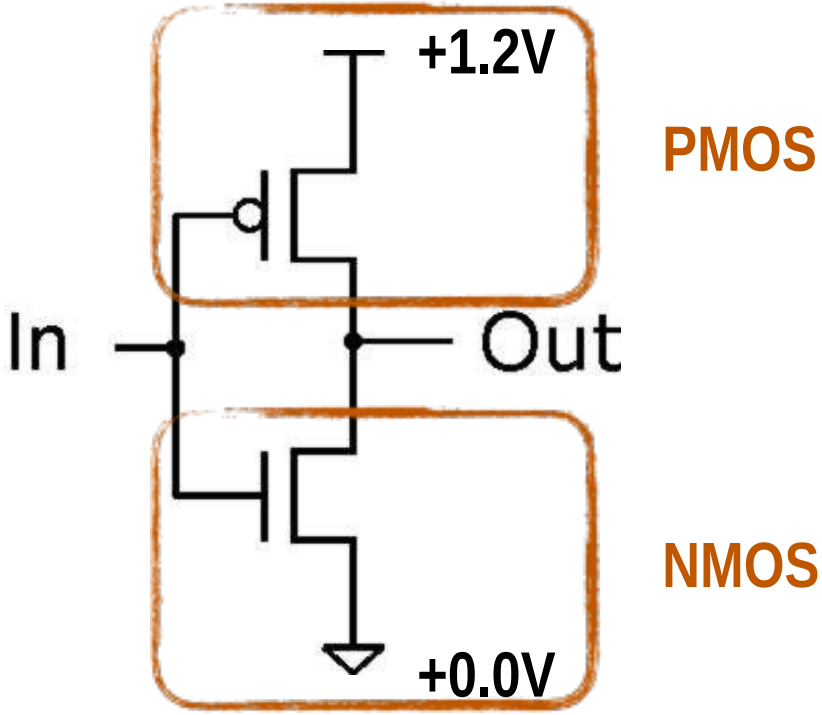
Truncate

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

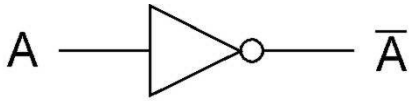
This is not an overflow by definition

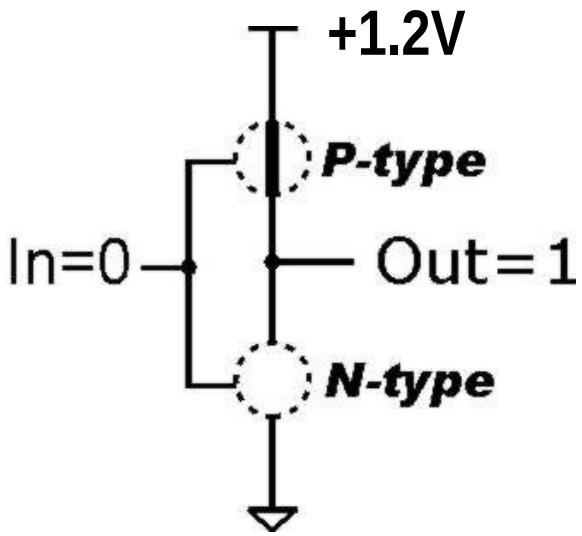
Because the actual result can be represented using the bit width of the datatype (3 bits here)

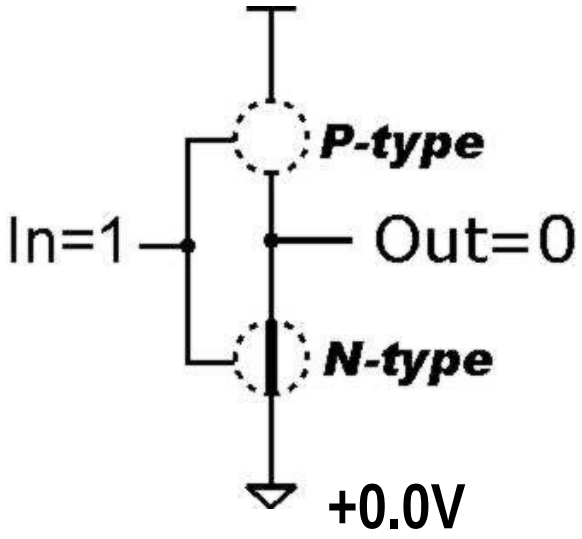
Inverter (NOT Gate)



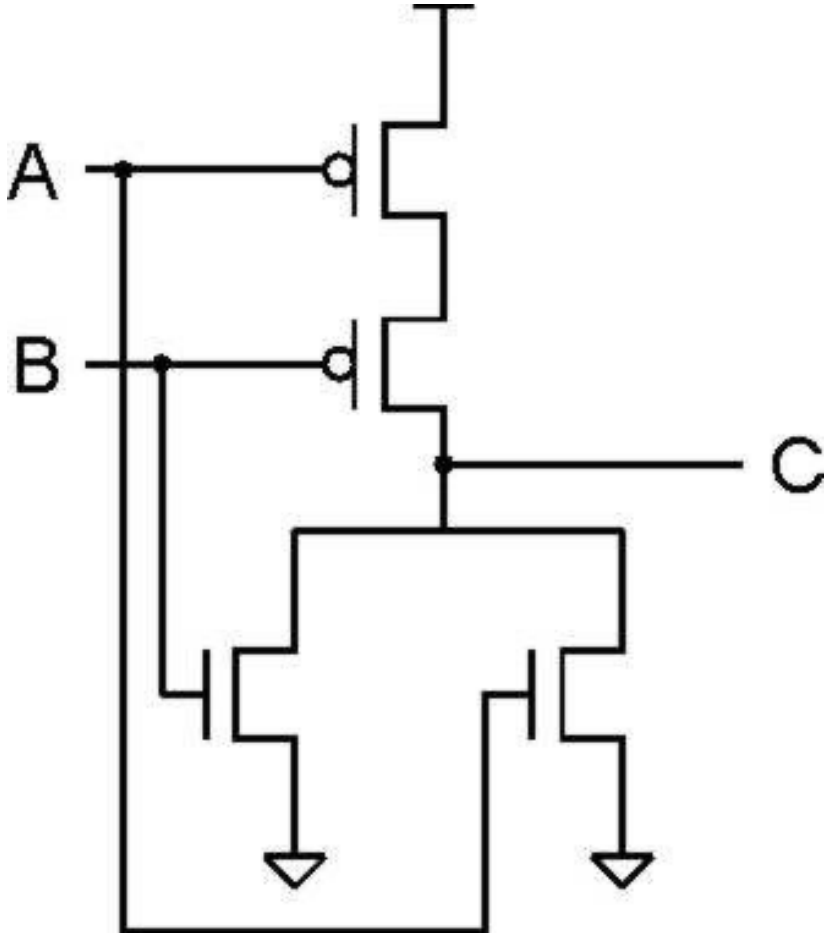
In	Out
0	1
1	0

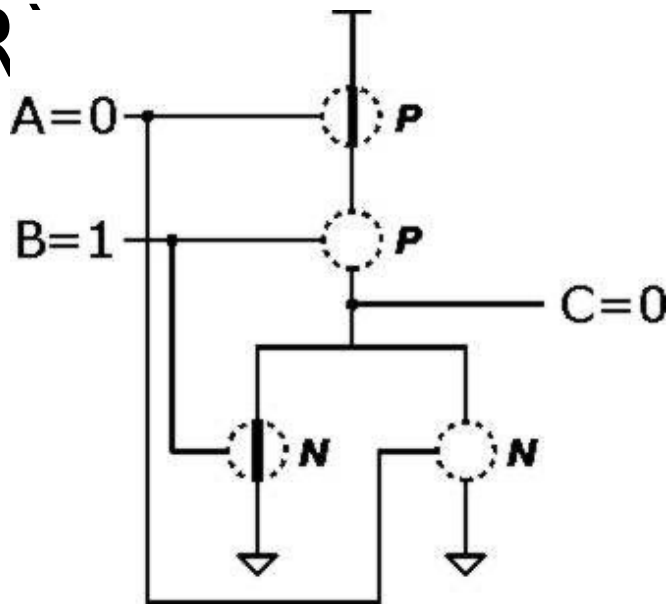






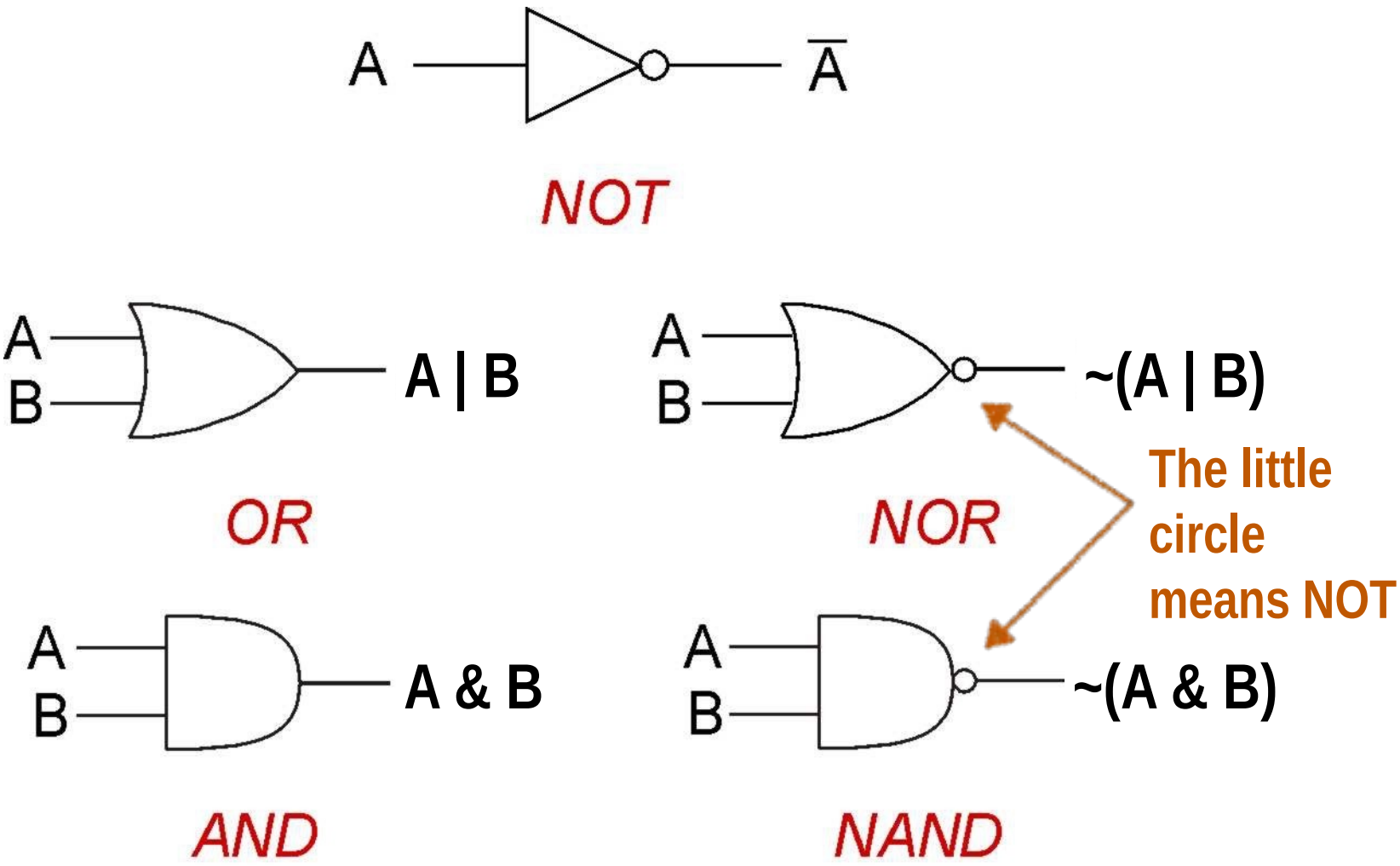
NOR Gate (NOT + OR`





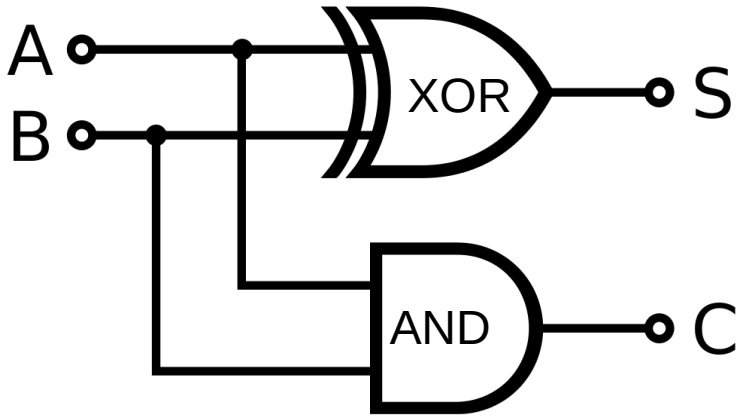
A	B	C
0	0	1
0	1	0
1	0	0
1	1	0

Basic Logic Gates



Half Adder

Add two bits, produce one bit

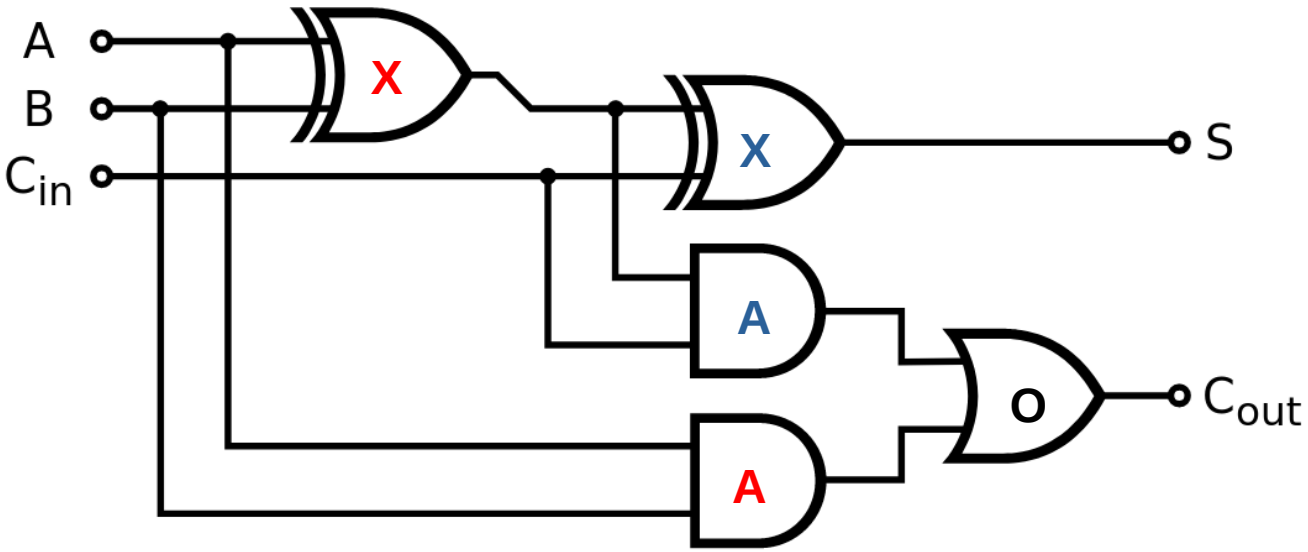


Truth Table

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

Full (1-bit) Adder

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

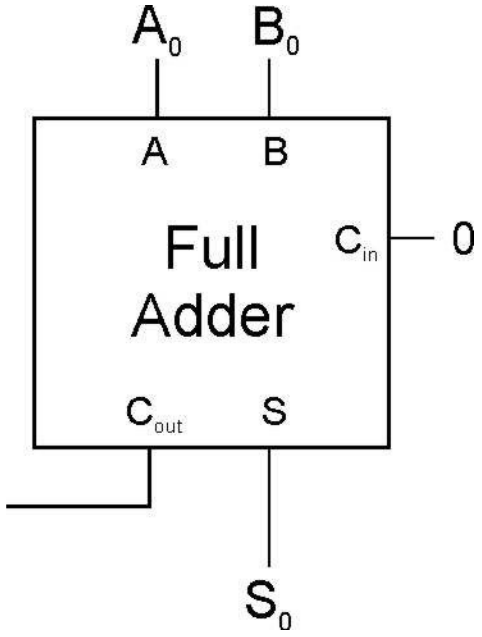


Truth Table

A	B	C_{in}	S	C_{out}
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

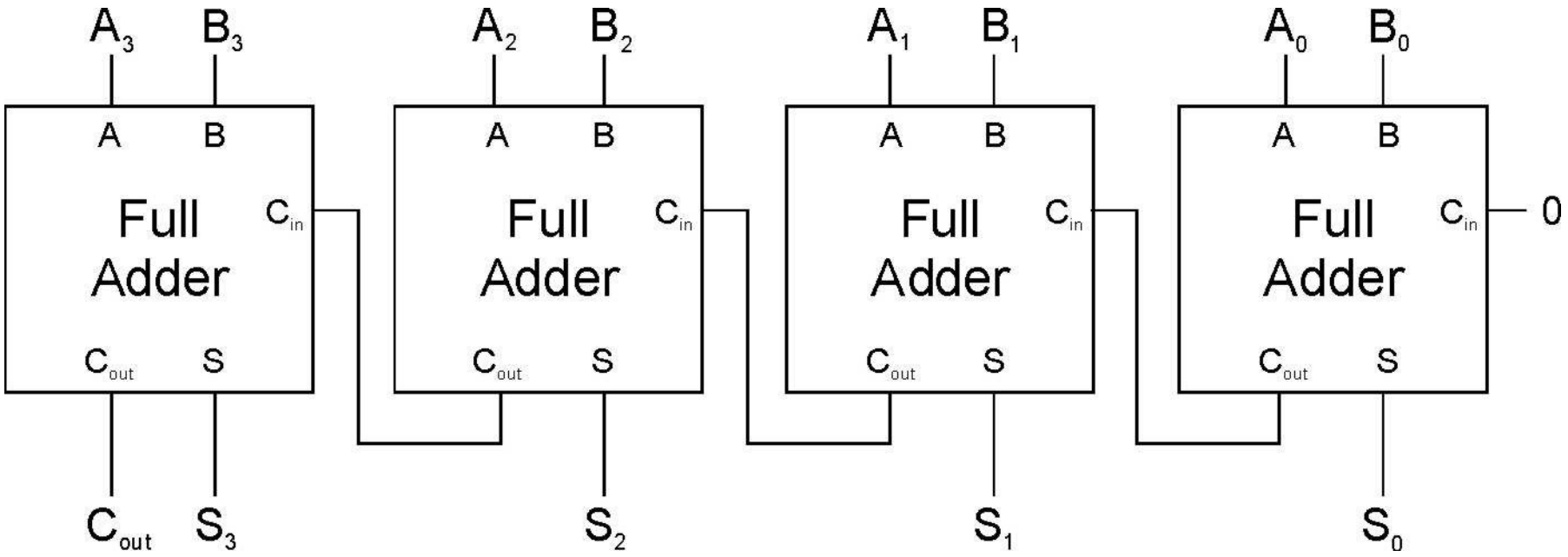
Four-bit Adder

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



Four-bit Adder

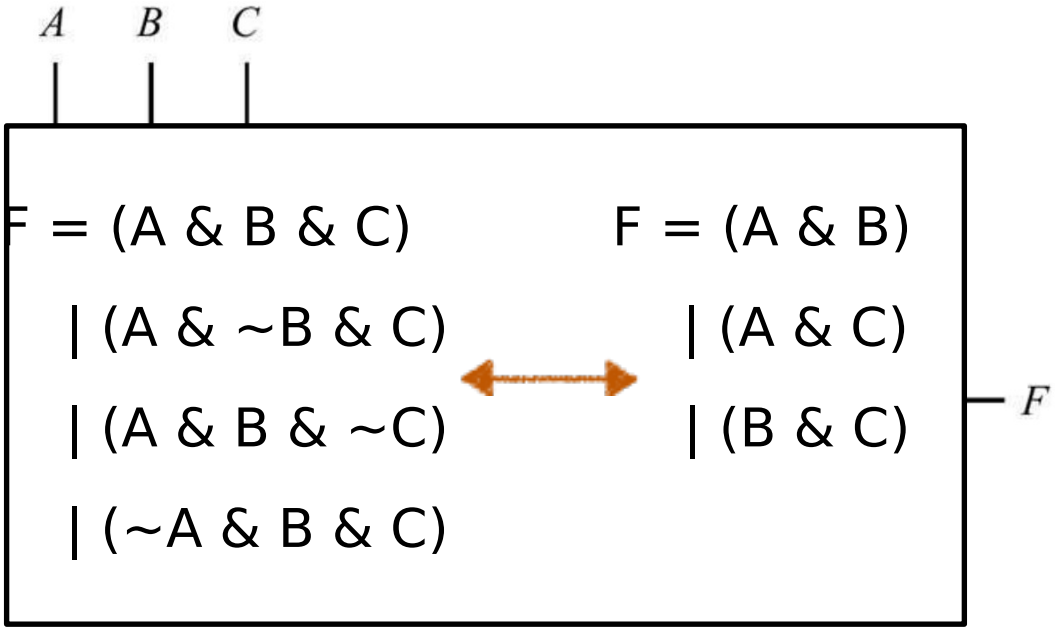
- Ripple-carry Adder
  - Simple, but performance linear to bit width
- Carry look-ahead adder (CLA)
  - Generate all carriers simultaneously



Logic Design

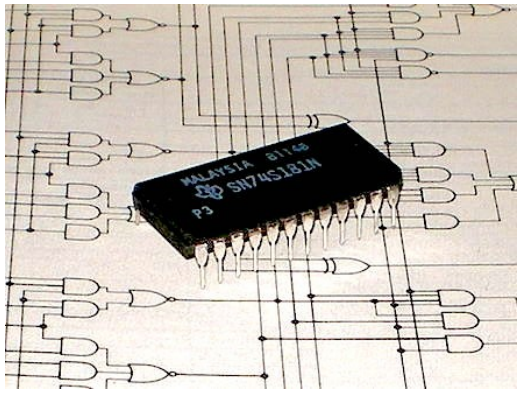
Design digital components from basic logic gates

Key idea: use the truth table!



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**

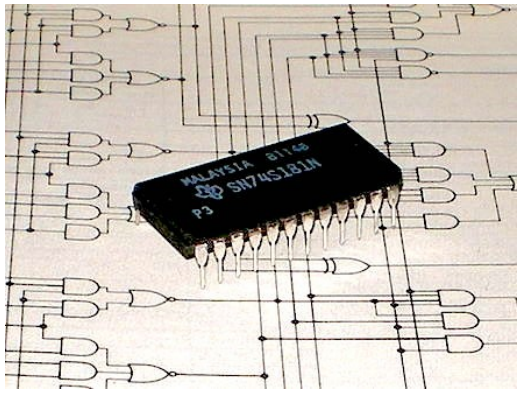
The 74181 ALU



An “Arithmetic Logic Unit”
The part of the CPU that does arithmetic

The first ALU on a chip! (around 1969)

The 74181 ALU

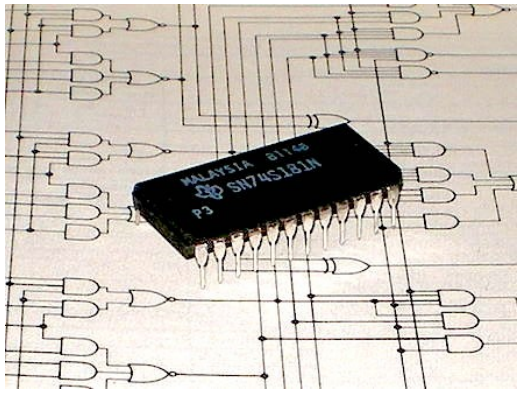


An “Arithmetic Logic Unit”
The part of the CPU that does arithmetic

The first ALU on a chip! (around 1969)

Contains 75 logic gates, and is 4 bits wide (can be chained)

The 74181 ALU



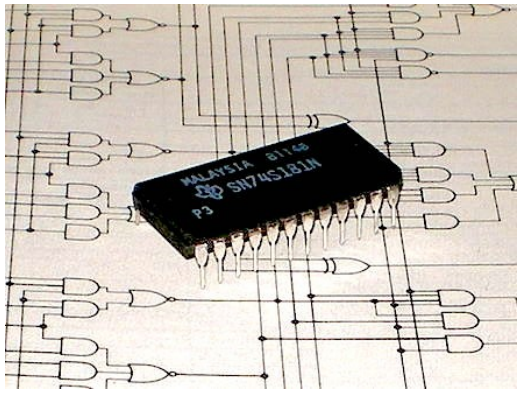
An “Arithmetic Logic Unit”
The part of the CPU that does arithmetic

The first ALU on a chip! (around 1969)

Contains 75 logic gates, and is 4 bits wide (can be chained)

Add/subtract, bitwise operations, shifts

The 74181 ALU



An “Arithmetic Logic Unit”
The part of the CPU that does arithmetic

The first ALU on a chip! (around 1969)

Contains 75 logic gates, and is 4 bits wide (can be chained)

Add/subtract, bitwise operations, shifts

Was used in many historically important computers including the PDP-11, on which Unix was developed

The 74181 ALU