University of Illinois at Urbana-Champaign
Dept. of Electrical and Computer Engineering

ECE 120: Introduction to Computing

2's Complement Overflow and Boolean Logic

© 2016 Steven S. Lumetta. All rights reserved.

Example: Addition of Unsigned Bit Patterns

A question for you:

What is the overflow condition for addition of two N-bit 2's complement bit patterns?

(That is, when is the sum incorrect?)

Remember that addition works exactly the same way as with **N-bit unsigned** bit patterns, so we can do some base 2 addition to find the answer.

© 2016 Steven S. Lumetta. All rights reserved.

Adding Two Non-Negative Patterns Can Overflow

Let's start with our first example from before:

```
   11
   01110  (14)
+  00100  (4)
-------
   10010  (-14)
```

Oops! We had no carry out, but the answer is wrong (an overflow occurred).

So overflow is different than for **unsigned**…

© 2016 Steven S. Lumetta. All rights reserved.

Carry Out Does Not Indicate 2's Complement Overflow

This example overflowed when the bits were interpreted with an **unsigned** representation.

We have no space for that bit!

```
  (1)11
   01110   (14)
+  10101   (-11; 21 unsigned)
-------
   00011   (3)
```

But here the answer is still correct!

Carry out ≠ overflow for 2's complement.

© 2016 Steven S. Lumetta. All rights reserved.

Adding Non-Negative to Negative Can Never Overflow

Claim:

Addition of two **N-bit 2's complement** bit patterns can not overflow if one pattern is **negative** (starts with 1) and the other pattern is **non-negative** (starts with 0).

Proof: **You do it!**

And THEN you can read the proof in the notes.

© 2016 Steven S. Lumetta. All rights reserved.

Long Definition for Overflow of 2's Complement Addition

Add two **N-bit 2's complement** patterns.

$$\begin{array}{rl} & \mathtt{A}\ a_{N-2} \ \ldots \ a_0 \quad \text{(sign bit is A)} \\ + & \mathtt{B}\ b_{N-2} \ \ldots \ b_0 \quad \text{(sign bit is B)} \\ \hline & \mathtt{S}\ s_{N-2} \ \ldots \ s_0 \quad \text{(sign bit is S)} \end{array}$$

Claim: The addition overflows iff one of the following holds:

1. The two addends are non-negative, and the sum is negative.
2. The two addends are negative, and the sum is non-negative.

© 2016 Steven S. Lumetta. All rights reserved.

Boolean Algebra Gives a More Concise Expression

That's a lot of words!

Boolean algebra gives a more concise form:

OVERFLOW =
[(NOT **A**) AND (NOT **B**) AND **C**] OR
[**A** AND **B** AND (NOT **C**)]

(Remember: **A**, **B**, and **C** were the sign bits.)

But what do these operators (AND, OR, and NOT) mean?

© 2016 Steven S. Lumetta. All rights reserved.

Boolean Operators Were Invented in the mid-19th Century

Boolean operators were invented (by George Boole) to reason about logical propositions.

They originally operated on true/false values.

We use them with … that's right, bits!

0 = false and 1 = true

Be careful not to confuse Boolean operators with English words. **The meanings are not identical.**

© 2016 Steven S. Lumetta. All rights reserved.

We Use Only a Few Boolean Functions

AND: the ALL function
returns 1 iff **ALL inputs are 1** (otherwise 0)

OR: the ANY function
returns 1 iff **ANY input is 1** (otherwise 0)

NOT: logical complement
(NOT 0) is 1; **(NOT 1) is 0**

XOR: the ODD function
returns 1 iff **an ODD number of inputs are 1** (otherwise 0)

© 2016 Steven S. Lumetta. All rights reserved.

A Truth Table Fully Defines a Boolean Function

The drawing to the right is a **truth table**.

A truth table allows us to

- define a Boolean function **C**
- by listing the output value
- for all combinations of inputs (here **A** and **B**, in base 2 order).

Let's write truth tables for our four Boolean functions.

A	B	C
0	0	
0	1	
1	0	
1	1	

© 2016 Steven S. Lumetta. All rights reserved.

AND: The ALL Function

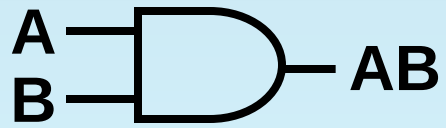
Let's start with AND.

AND can be written in several ways:

- **AB**
- **A·B**
- **A×B**
- **A^B** (math. conjunction)

We usually use these.

A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1



Note flat input, rounded output.

© 2016 Steven S. Lumetta. All rights reserved.

OR: The ANY Function

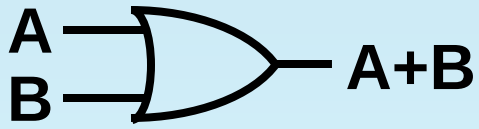
And now OR.

OR can also be written in other ways:

- **A + B** | **We usually use this one.**
- **A∨B** (math. disjunction)

A	B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

Note rounded input, pointed output.



© 2016 Steven S. Lumetta. All rights reserved.

NOT: Logical Complement

And now NOT.

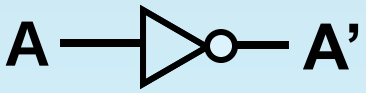
NOT can also be written in other ways:

- A'
- $\overline{A}$
- ¬ A (math. complement)

We usually use these.

A	NOT A
0	1
1	0

Note triangle and inversion bubble.



© 2016 Steven S. Lumetta. All rights reserved.

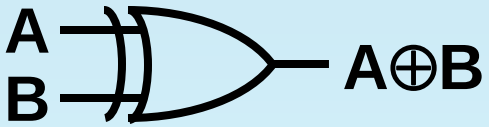
XOR: The ODD Function

And, finally, XOR.

XOR is usually written this way: **A ⊕ B**

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

Note: like OR, but double line for inputs.



© 2016 Steven S. Lumetta. All rights reserved.

Use Definitions to Generalize to More than Two Operands

Generalize to more operands using the definitions given:

- **AND: ALL**
- **OR: ANY**
- **XOR: ODD**

As an example, fill the truth table for a **3-input XOR**.

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

© 2016 Steven S. Lumetta. All rights reserved.

Generalize to Sets of Bits by Pairing Bits

We can also generalize to sets of bits.

For example, if we have two **N**-bit patterns,

$$\mathbf{A=a_{N-1}\ldots a_0} \text{ and } \mathbf{B=b_{N-1}\ldots b_0},$$

we can write

$$\mathbf{C = A \text{ AND } B}$$

To mean that

if $\mathbf{C=c_{N-1}\ldots c_0}$, $\mathbf{c_i = a_i b_i}$ for $\mathbf{0 \le i < N}$.

© 2016 Steven S. Lumetta. All rights reserved.

Don't Mix Algebras: Use AND/OR/NOT for Bitwise Logic

If **A** is a **2's complement** bit pattern, we might also write **-A = (NOT A) + 1**

Be careful about mixing

- algebraic notation for Boolean functions
- with arithmetic operations.

The "+" in the equation above means base 2 addition (and discarding any carry out), not OR.

© 2016 Steven S. Lumetta. All rights reserved.

University of Illinois at Urbana-Champaign
Dept. of Electrical and Computer Engineering

ECE 120: Introduction to Computing

Logical Completeness

© 2016 Steven S. Lumetta. All rights reserved.

Can We Count Functions?

A question for you:

How many different Boolean functions exist for N bits of input?

How can we find the answer?

Start by thinking about small values of **N**.

For example, **N=2**. Given **C = F(A,B)**, **how many choices do we have for F**?

© 2016 Steven S. Lumetta. All rights reserved.

Two Bits of Input Can be Combined into 16 Functions

Write a truth table for **C = F(A,B)**.

But instead of filling in values, call the outputs c_i.

The four c_i values uniquely specify **F**.

If we change any c_i, we get a different function.

A	B	C
0	0	c_0
0	1	c_1
1	0	c_2
1	1	c_3

We thus have **$2 \times 2 \times 2 \times 2 = 2^4$ choices for F**.

© 2016 Steven S. Lumetta. All rights reserved.

Three Bits of Input Can be Combined into 256 Functions

What about **N=3**:
D = G(A,B,C)?

We can again write a truth table.

And call the outputs d_i.

Now we have
2^8 choices for G.

Notice that $2^8 = 2^{(2^3)}$.

A	B	C	D
0	0	0	d_0
0	0	1	d_1
0	1	0	d_2
0	1	1	d_3
1	0	0	d_4
1	0	1	d_5
1	1	0	d_6
1	1	1	d_7

© 2016 Steven S. Lumetta. All rights reserved.

N Bits of Input Can be Combined into Many Functions

Can we generalize to **N** bits?

Without drawing a truth table, please?

N bits means 2^N rows in the truth table.

Thus we need 2^N Boolean values (bits) to specify a function.

Thus **$2^{(2^N)}$ possible functions on N bits**.

© 2016 Steven S. Lumetta. All rights reserved.

We Need More Functions!

So why did we teach you only four functions?

(AND, OR, NOT, and XOR)

Your homework for next time:

Write down and name all functions on 10 bits.

Include a truth table for each function!

© 2016 Steven S. Lumetta. All rights reserved.

Alternate Homework: Understand Logical Completeness

Claim:

With enough 2-input AND, 2-input OR, and NOT functions, one can produce **any function on any number of variables**.

Believe me?

Proof: **by construction**

In other words, I'll show you how to produce an arbitrary function on an arbitrary number of variables.

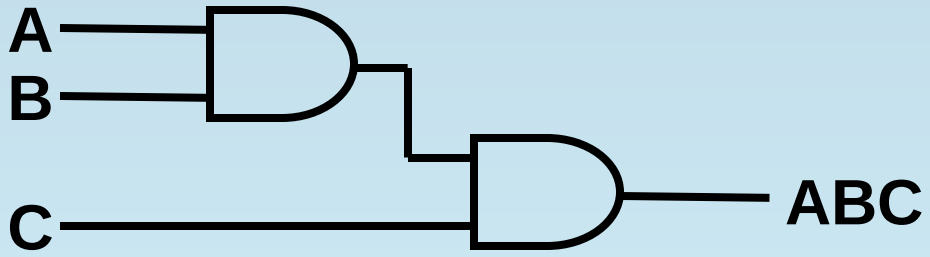
© 2016 Steven S. Lumetta. All rights reserved.

Compose Functions to Produce Functions on More Inputs

Let's start the proof.

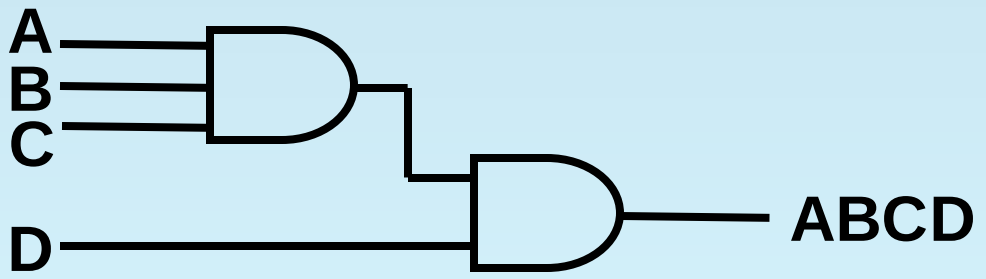
What does this circuit produce?

A **3-input AND**!



What about this one?

A **4-input AND**!



© 2016 Steven S. Lumetta. All rights reserved.

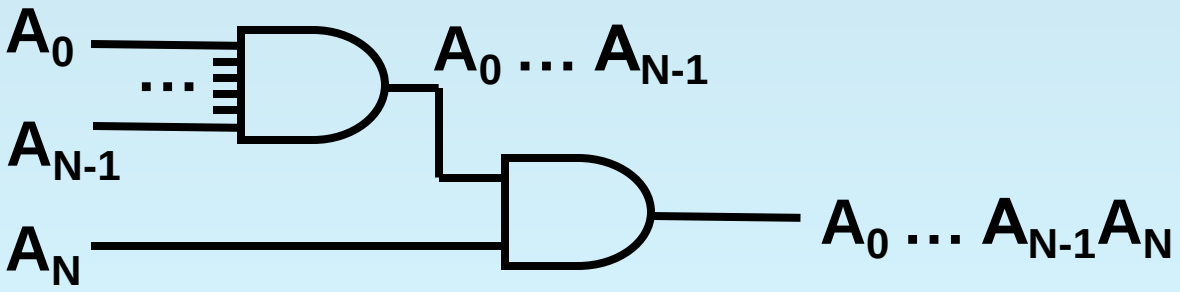
Use 2-Input Gates to Construct N-Input Gates

By induction, we build an **N**-input AND gate.

Base case (**N=2**): Use one 2-input AND.

N+1 case (given an **N**-input AND):

- Use one **N**-input AND
- and one 2-input AND
- to produce an (**N+1**)-input AND.



© 2016 Steven S. Lumetta. All rights reserved.

Comments on Functional Form and Practical Value

A couple of comments before we continue…

- Functional form of the inductive proof:
  - base: $\mathbf{AND_2\ (A, B) = AND_2\ (A, B)}$
  - inductive step:
    $\mathbf{AND_{N+1}\ (A_0, \ldots, A_N) =}$
    $\mathbf{AND_2\ (AND_N\ (A_0, \ldots, A_{N-1}), A_N)}$
- This approach is an existence proof, **not a practical way to build bigger gates.**

© 2016 Steven S. Lumetta. All rights reserved.

The Claim is Now Slightly Simpler

Claim:

With enough ~~2-input~~ AND, ~~2-input~~ OR, and NOT functions, I can produce **any function on any number of variables**.

(For OR functions, use the same approach as we did with AND functions, replacing AND with OR.)

Let's first consider functions that

- produce an output of 1
- for exactly one combination of inputs (one row of the function's truth table).

© 2016 Steven S. Lumetta. All rights reserved.

One AND Suffices for Functions that Output One 1

The function **Q(A,B,C)** is an example of such a function.

When is **Q=1**?

Only when **A=1 AND B=0 AND C=1**.

Note that **B=0** when **(NOT B) = 1**.

In other words, **Q = AB'C**.

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

© 2016 Steven S. Lumetta. All rights reserved.

Arbitrary Functions Require Only Two Steps

To produce **an arbitrary function** (which may produce the value 1 for more than one combination of inputs):

1. For each combination of inputs for which the function produces a 1, AND together the corresponding inputs or inverted inputs.*
2. OR together the results of all AND functions.

* The resulting AND is called a **minterm** on the input variables.

© 2016 Steven S. Lumetta. All rights reserved.

A Sum-of-Products Can Express Any Function

The construction described results in a **sum-of-products** form because

- we produce each row of the truth table with an AND (product / multiplication notation)
- we produce the final function by ORing the ANDs (sum / addition notation).

The approach described is **often inefficient**, but it **always works**.

© 2016 Steven S. Lumetta. All rights reserved.

{AND, OR, NOT} is Logically Complete

Definition: The set **{AND, OR, NOT}** is **logically complete** because, as we showed,

any Boolean logic function on any number of inputs can be produced using only AND, OR, and NOT.

To show that another set is logically complete

- You need not construct arbitrary functions.
- You need only show how to construct AND, OR, and NOT.

© 2016 Steven S. Lumetta. All rights reserved.

Why Do You Care? Abstraction!

Imagine working on a new device technology.

- Maybe it's based on DNA.
- Maybe it's based on new semiconductors.
- Maybe it's based on carbon nanotubes.
- Maybe you're still finishing your degree?!

What do you need to be able to build in order to replace the current technology?

AND, OR, and NOT.

Other people can then build higher layers of abstraction!

© 2016 Steven S. Lumetta. All rights reserved.

Example: 3-input XOR

Let's build XOR as an example.

First, write the truth table.

What function produces this row? **A'B'C**

And this row? **A'BC'**

And this one? **AB'C'**

And this one? **ABC**

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

© 2016 Steven S. Lumetta. All rights reserved.

3-input XOR Expressed Using AND, OR, and NOT

Putting these four functions together, we obtain:

A XOR B XOR C =
A'B'C + A'BC' + AB'C' + ABC

Now we are ready to begin building devices such as adders and comparators to manipulate our representations…

© 2016 Steven S. Lumetta. All rights reserved.