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

ECE 120: Introduction to Computing

2's Complement Overflow and Boolean Logic

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.

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**…

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!

```
  111
   01110  (14)
 + 10101  (-11; 21 unsigned)
 -------
   00011  (3)
```

But here the answer is still correct!

Carry out ≠ overflow for 2's complement.

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.

Long Definition for Overflow of 2's Complement Addition

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

$$\begin{array}{rl} \texttt{A}\ \texttt{a}_{\texttt{N-2}} \ldots \texttt{a}_0 & \text{(sign bit is A)} \\ +\ \texttt{B}\ \texttt{b}_{\texttt{N-2}} \ldots \texttt{b}_0 & \text{(sign bit is B)} \\ \hline \texttt{S}\ \texttt{s}_{\texttt{N-2}} \ldots \texttt{s}_0 & \text{(sign bit is S)} \end{array}$$

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.

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?

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.**

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)

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	

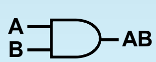
AND: The ALL Function

Let's start with AND.

AND can be written in several ways:
- **AB** | **We usually use these.**
- **A·B**
- **A×B**
- **A^B** (math. conjunction)

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

Note flat input, rounded output.



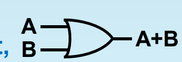
OR: The ANY Function

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.



NOT: Logical Complement

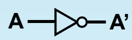
And now NOT.

NOT can also be written in other ways:

- A' | **We usually**
- $\overline{A}$ | **use these.**
- $\neg A$ (math. complement)

A	NOT A
0	1
1	0

Note triangle and inversion bubble.



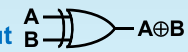
XOR: The ODD Function

And, finally, XOR.

XOR is usually written this way: $A \oplus B$

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.



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	$A \oplus B \oplus C$
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

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,

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

we can write

$C = A \text{ AND } B$

To mean that

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

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.