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

ECE 120: Introduction to Computing

Building with Abstraction and a First Example

Optimization at the Level of Gates is Always Possible

One can always solve a problem by
- developing complete Boolean expressions,
- solving for "good" forms with K-maps (or algebra, with more variables),
- implementing the resulting equations,
- tuning logic to reduce gate sizes (number of inputs) and fanout (the number of gates using a single gate's output).

You can now perform such a process.

But Optimization at Gate Level is Rarely Needed

Such detail is rarely needed for a satisfactory solution.

Instead, humans can
- use abstraction and build with components such as adders and comparators, or
- use extra levels of logic to describe functions more intuitively.

Computer-aided design (CAD) tools
- can help with low-level optimizations.
- In many cases, CAD tools can do better than humans because they explore more options.

Tradeoffs are Always Made in Some Context

Context is important!

If a mechanical engineer produces a 0.5% boost in efficiency for internal combustion engines sized for automobiles, that engineer will probably win a major prize.

In our field, engineers spend a lot of time
- improving the designs of arithmetic units and memory, and
- improving CAD tools' ability to optimize.

But Optimization at Gate Level is Rarely Needed

"Premature optimization is the root of all evil."
– Sir C.A.R. "Tony" Hoare

Don't spend time optimizing
- something that is likely to change, nor
- something that does not contribute much to the overall system goodness (any metric).

The flip side:
- don't ignore scaling issues when choosing algorithms, and
- don't design in a way that prohibits/inhibits optimization.

First Example: Subtraction

Let's start with something simple.

Let's build a subtractor.

How do we subtract as humans?

Example: Subtraction of 5-Digit Numbers

Let's do an example with **5-digit numbers**

```
  12345
-   871
```

Negate by finding the "9's complement" and adding 1.

Example: Subtraction of 5-Digit Numbers

Let's do an example with **5-digit numbers**

```
 11  1
  12345
+ 99129
  11474
```

We have no space for that digit!

Good, we got the right answer (12345 – 871 = 11474)!

Use an Adder to Implement a Subtractor

Ok, maybe your elementary school taught subtraction a different way.

But you probably did use that approach in your ECE120 homework to subtract **unsigned** and **2's complement** values.

A – B = A + (NOT B) + 1

(where "NOT" applies to all bits of **B**)

Instead of mimicking the human subtraction process, let's use an adder to implement a subtractor…

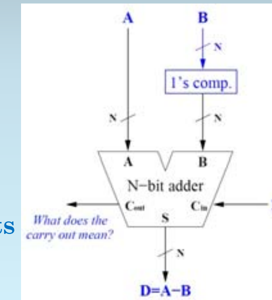
Use an Adder to Implement a Subtractor

Take a look at the design to the right.

The core is an **N-bit adder**.

We want to calculate the difference **D = A – B**.

We **modify the inputs slightly** to perform the subtraction.



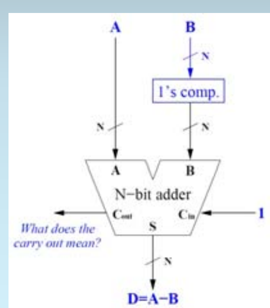
Convert B to its 1's Complement

The input **A** is unchanged.

The input **B** is transformed to its 1's complement.

How do we implement 1's complement?

N inverters!



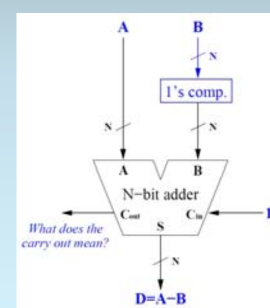
With C_{in} = 1, the Adder Produces A - B

Finally, the C_{in} input is set to 1.

So what does the adder calculate?

A + (NOT B) + 1

which is **A – B**, as desired.

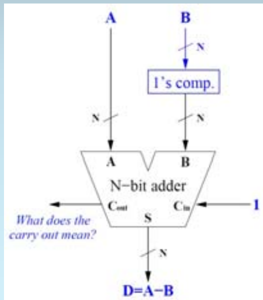


Calculate D to Understand the Carry Out Signal

What does the carry out C_{out} mean?

Remember that the 1's complement is $(2^N - 1) - B$.

So we obtain D
$= A + (2^N - 1) - B + 1$
$= A - B + 2^N$



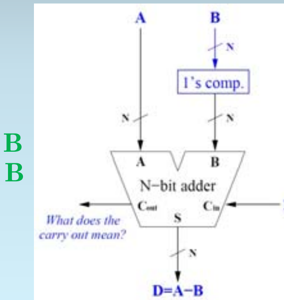
Carry Out Signal (Opposite Sense) Still Means Overflow

So $D = A - B + 2^N$.

But C_{out} is the 2^N term from the adder.

Thus
- $C_{out} = 1$ means $A \geq B$
- $C_{out} = 0$ means $A < B$

So for unsigned subtraction $C_{out} = 0$ indicates overflow!



Carry Out Signal (Opposite Sense) Still Means Overflow

What if we want a device that does both addition and subtraction?

We need a way to choose the operation.

Add a control signal **S**
- **S = 0**: addition
- **S = 1**: subtraction

And the modify the adder inputs with **S**
- **A** … **unmodified**
- **B** … **B XOR S** (bitwise)
- C_{in} … **S**