

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

ECE 120: Introduction to Computing

A Power-of-Two Checker

Powers of Two are Easy to Spot in Binary

Let's do another bit-sliced design.

Can we check whether an unsigned number represents a power of two?

What does a power of two look like in bits?

For **5-bit unsigned**, the powers of 2 are...

00001, 00010, 00100, 01000, 10000

A power of two has exactly one 1 bit
(with place value 2^N for some N , of course!).

The Answers are Not Always Enough

So our design will answer the following:

Is $A = a_{N-1}a_{N-2}\dots a_1a_0$ a power of two?

How many answers are possible?

Two: Yes, and No.

A trick question:

**How many bits do we need to pass
between slices?**

That's right: TWO bits.

What Extra Information Do We Need?

Why not just one? An answer only needs 1 bit!

Say that we pass bits from right to left.

If the bits $\mathbf{a_{N-2}...a_1a_0}$ represent a power of two,
is $\mathbf{a_{N-1}a_{N-2}...a_1a_0}$ be a power of two?

What if $\mathbf{a_{N-2}...a_1a_0}$ does **not** represent a power of two? **Iff $\mathbf{a_{N-1} = 0}$.**

In that case, **we can't tell whether $\mathbf{a_{N-1}a_{N-2}...a_1a_0}$ is a power of two or not!**

What else do we need to know?

For Inductive Step, We Must Know Whether all Bits are 0

Imagine that we have completed $N-1$ bits.

Under what conditions can number A be a power of two?

1. $a_{N-1} = 1$ and the rest is all 0s, or
2. $a_{N-1} = 0$ and the rest is a power of two.

For #2, we need to **know whether the rest of the bits form a power of two.**

But for #1, we also need to **know whether the rest of the bits are all 0.**

There are Three Possible Messages between Bit Slices

The “yes” cases for #1 and #2 do not overlap:
all 0 bits is not a power of two.

The “no” cases need not be further separated:

- all 0s means **no 1 bits**
- a power of two means **one 1 bit**
- **more than one 1 bit** means
“no” to both questions

That’s all we need to know. **Three possible messages** between slices, **so two bits**.

We Need a Representation for Answers

I'll use the following representation.

Others may be better.

C_1	C_0	meaning
0	0	no 1 bits
0	1	one 1 bit
1	0	not used
1	1	more than one 1 bit

We Need a Representation for Answers

Let's build a slice that operates on two bits of **A**.

In the bit slice, we call them **A** and **B**.

Inputs from the previous bit slice are **C₁** and **C₀**.

Outputs to the next bit slice are **Z₁** and **Z₀**.

Direction of our operation doesn't matter. Either will do.

Two Zeroes Do Not Change the Result

Let's fill in a truth table.

We'll start with the case of **A = 0** and **B = 0**.

A	B	C ₁	C ₀	meaning	Z ₁	Z ₀	meaning
0	0	0	0	no 1s	0	0	no 1s
0	0	0	1	one 1	0	1	one 1
0	0	1	0	???	x	x	don't care
0	0	1	1	>one 1	1	1	>one 1

One 1 Input Increments the Count of 1 Bits

Now consider **A = 0** and **B = 1**.

A	B	C ₁	C ₀	meaning	Z ₁	Z ₀	meaning
0	1	0	0	no 1s	0	1	one 1
0	1	0	1	one 1	1	1	>one 1
0	1	1	0	???	x	x	don't care
0	1	1	1	>one 1	1	1	>one 1

One 1 Input Increments the Count of 1 Bits

The case for **A = 1** and **B = 0** is the same.

A	B	C ₁	C ₀	meaning	Z ₁	Z ₀	meaning
1	0	0	0	no 1s	0	1	one 1
1	0	0	1	one 1	1	1	>one 1
1	0	1	0	???	x	x	don't care
1	0	1	1	>one 1	1	1	>one 1

Two 1s in the Number Rules Out Powers of Two

Finally, consider **A = 1** and **B = 1**.

A	B	C ₁	C ₀	meaning	Z ₁	Z ₀	meaning
1	1	0	0	no 1s	1	1	>one 1
1	1	0	1	one 1	1	1	>one 1
1	1	1	0	???	x	x	don't care
1	1	1	1	>one 1	1	1	>one 1

We Solve Z_1 as a POS Expression

Let's use a K-map to solve Z_1 . POS looks good.

What are the loops?

$$(C_1 + A + B)$$

$$(C_0 + A)$$

$$(C_0 + B)$$

$$\text{So } Z_1 = (C_1 + A + B) \\ (C_0 + A)(C_0 + B)$$

Z_1

		AB			
		00	01	11	10
$C_1 C_0$	00	0	0	1	0
	01	0	1	1	1
	11	1	1	1	1
	10	X	X	X	X

We Solve Z_0 as an SOP Expression

Now let's solve Z_0 . SOP and POS are the same.

What are the loops for SOP?

C_0

A

B

So $Z_0 = (C_0 + A + B)$

Z_0		AB			
		00	01	11	10
$C_1 C_0$	00	0	1	1	1
	01	1	1	1	1
	11	1	1	1	1
	10	X	X	X	X

We Can Reuse Some Factors with Algebra

Notice that we can reuse factors from Z_1 to calculate Z_0 :

$$Z_1 = (C_1 + A + B)(C_0 + A)(C_0 + B)$$

$$Z_0 = (C_0 + A + B) = (C_0 + A) + (C_0 + B)$$

Let's draw the bit slice, then analyze its area and delay.

Area is $6N$, and Delay is N Gate Delays for N Bits

Here is an implementation of the bit slice using NAND and NOR. Let's find area.

How many literals? **7**

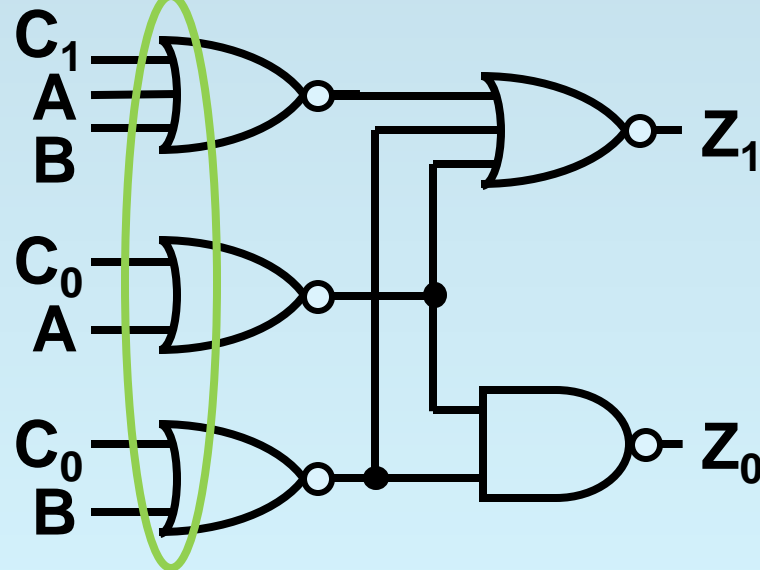
How many operations?

5 (4 NOR, 1 NAND)

And delay?

2 on all paths.

So N gate delays for N bits.



Need One More Gate Delay to Get the Answer

But we don't get an answer!

Our **N-bit** checker,

- composed of **N/2** bit slices,
- produces only a “count” of 1 bits (0, 1, or “many”).

We want yes (**P = 1**) or no (**P = 0**)!

Looking at the representation, the fastest solution is to add an XOR gate at the end.

P = Z₁ ⊕ Z₀ from the last bit slice.

So delay is actually **N + 1** gate delays.

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**

$$\begin{array}{r} 12345 \\ - \quad 871 \\ \hline \end{array}$$



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**

We have no ~~1~~ space for that digit!

$$\begin{array}{r} 1 \\ 12345 \\ + 99129 \\ \hline 11474 \end{array}$$

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.

$$A - B = A + (\text{NOT } B) + 1$$

(where “NOT” applies to all bits of **B**)

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

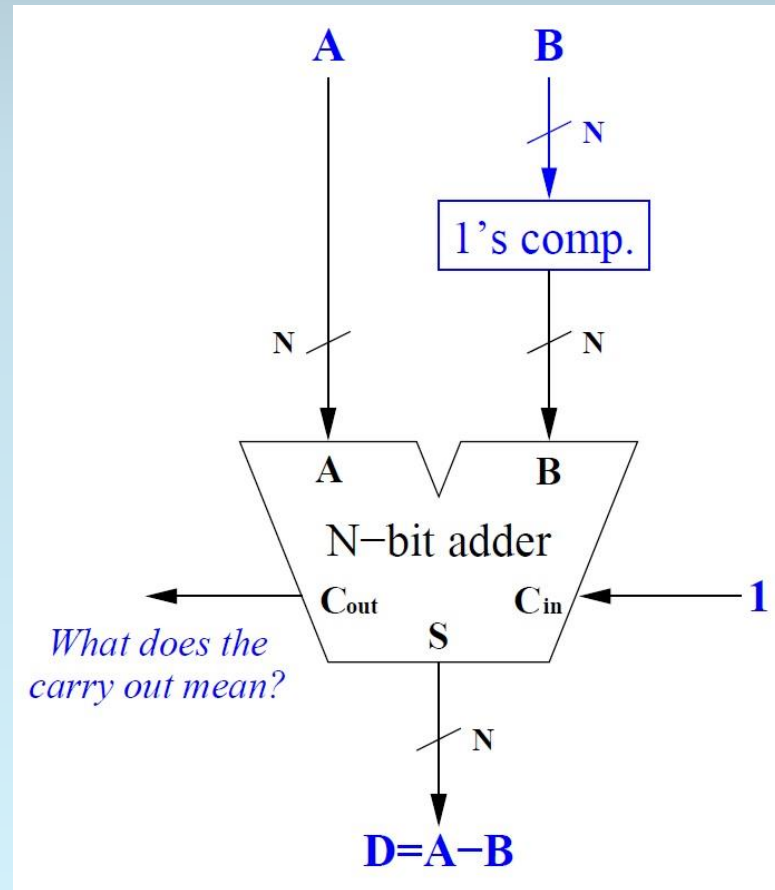
Use an Adder to Implement a Subtractor

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.



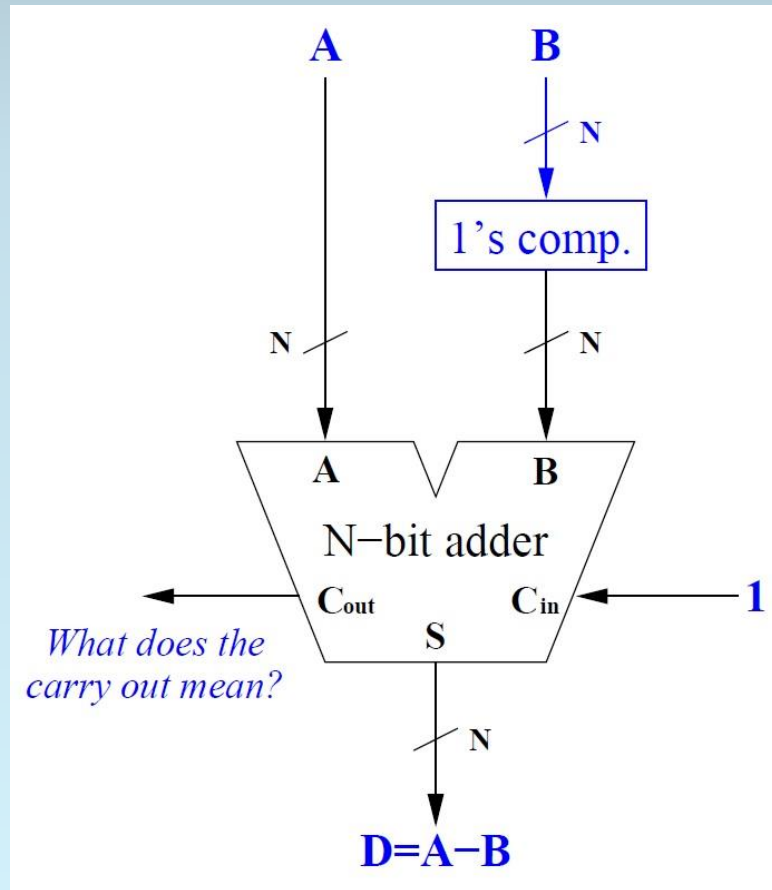
Convert B to its 1's Complement

The input **A** is unchanged.

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

How do we implement 1's complement?

N inverters!



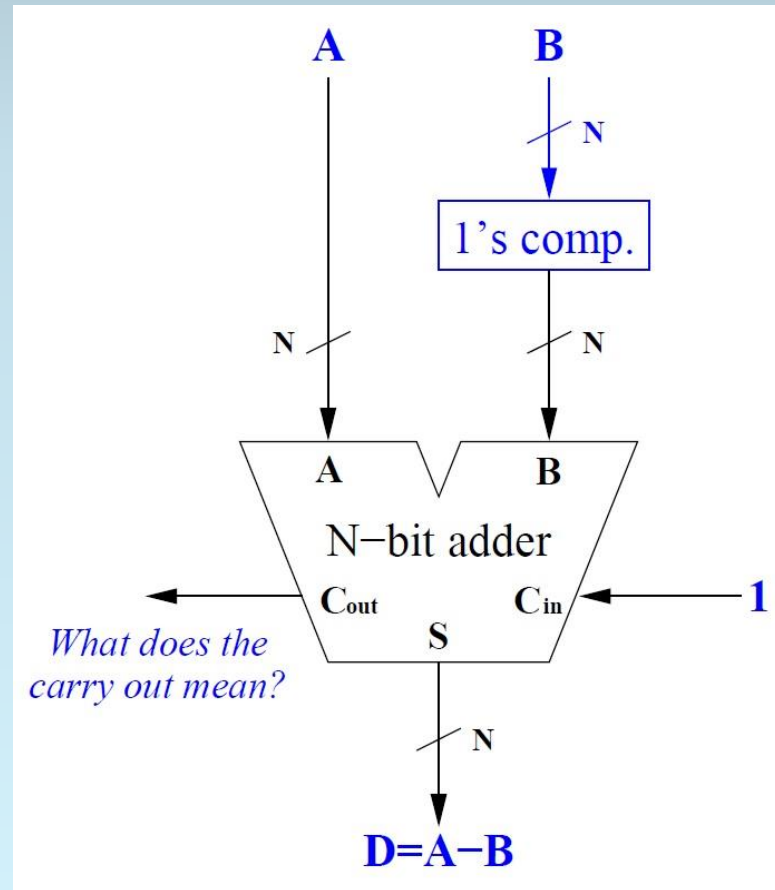
With $C_{in} = 1$, the Adder Produces $A - B$

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

So what does the adder calculate?

$$A + (\text{NOT } B) + 1$$

which is $A - B$,
as desired.

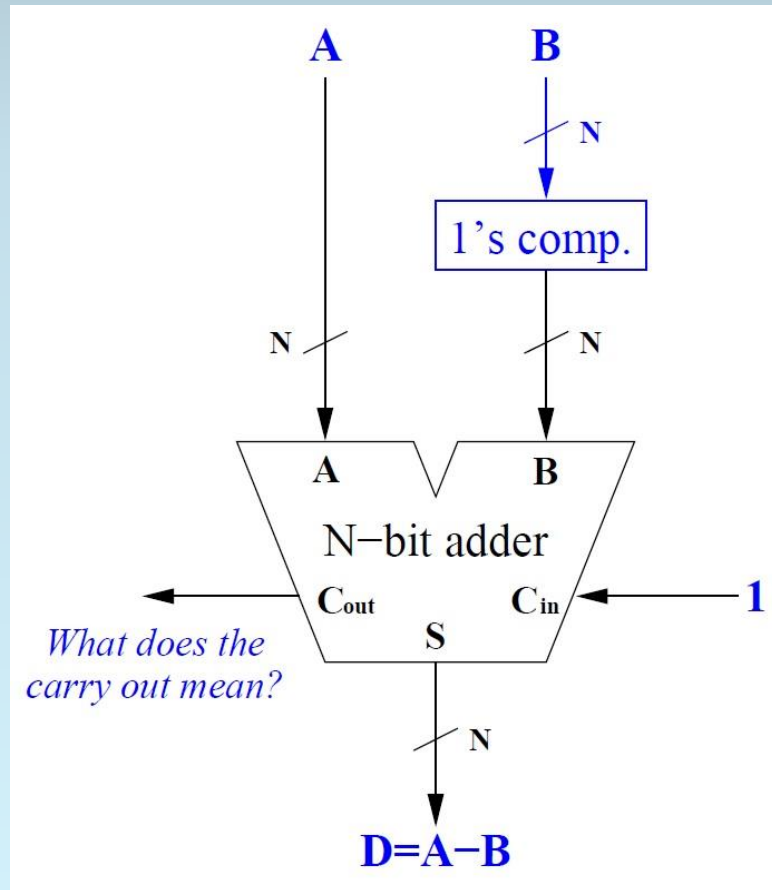


Calculate D to Understand the Carry Out Signal

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

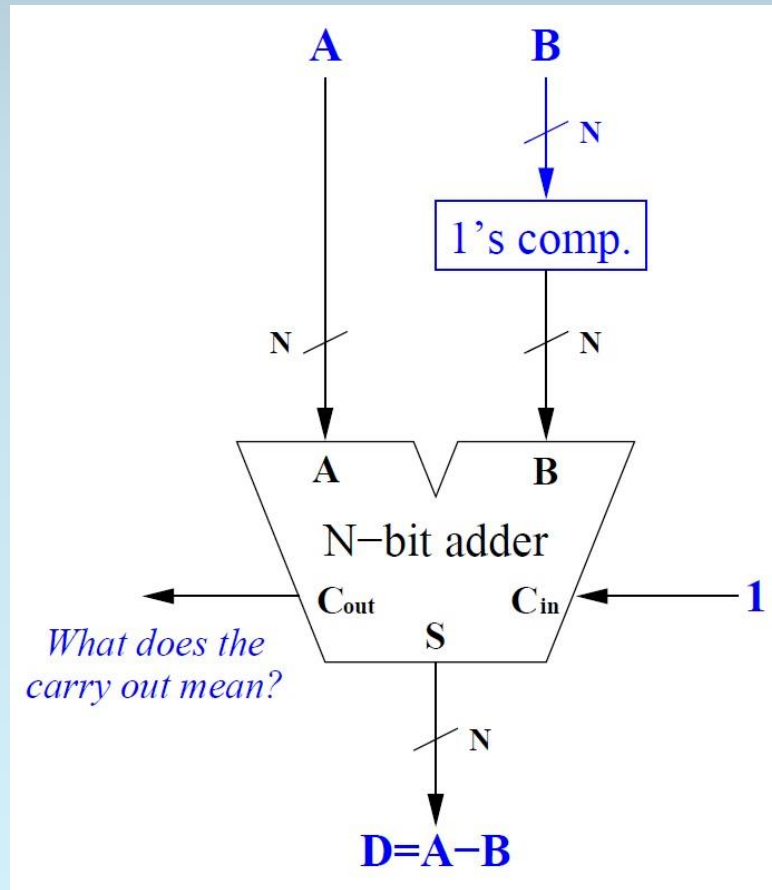
So $D = A - B + 2^N$.

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!



Use a Control Signal to Select between Operations

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 then modify the adder inputs with **S**

- **A** ... **unmodified**
- **B** ... **B XOR S** (bitwise)
- **C_{in}** ... **S**

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

ECE 120: Introduction to Computing

Checking for Upper-Case Characters

Task: Checking for an Upper-Case Letter

Let's design logic to check whether an **ASCII** character is an upper-case letter.

In **ASCII**, 'A' is **1000001** (0x41),
and 'Z' is **1011010** (0x5A).

Let's say that the **ASCII** character
is in $C = C_6C_5C_4C_3C_2C_1C_0$.

**How can we check whether
C represents an upper-case letter?**

We Will Need a BIG Truth Table!

Should we write a truth table for U(C) ?	C₆	C₅	C₄	C₃	C₂	C₁	C₀	U(C)
Can we skip to the ones that matter?	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	1	0
	0	0	0	0	0	1	0	0
	0	0	0	0	0	1	1	0
	0	0	0	0	1	0	0	0
	0	0	0	0	1	0	1	0
	0	0	0	0	1	1	0	0
	0	0	0	0	1	1	1	0

Let's Break the Truth Table into Eight Pieces

Can we **break the truth table into pieces**?

For example, let's break the truth table

- into eight truth tables
- of 16 rows each.

Each piece represents one value of $C_6C_5C_4$.

We can solve each piece with a K-map on $C_3C_2C_1C_0$.

Some Functions are Quite Simple

Or maybe we don't need a K-map for some.

Remember that 'A' is **1000001** (0x41),
and 'Z' is **1011010** (0x5A).

Think about the table for **$C_6C_5C_4 = 000$** ?

What is the function of **$C_3C_2C_1C_0$** ? **0**

(In other words, no **ASCII** character with
 $C_6C_5C_4 = 000$ is an upper-case letter.)

* This notation means **$C_6 = 0$** AND **$C_5 = 0$** AND **$C_4 = 0$** .

Only Two of Our Functions are Not the 0 Function

For reference: 'A' is **1000001** (0x41), and
'Z' is **1011010** (0x5A).

Which of our eight functions are
not the 0 function?

$C_6C_5C_4 = 100$ Let's call the function **T_4** .

$C_6C_5C_4 = 101$ Let's call the function **T_5** .

Let's solve K-maps for these two.

Solve T_4 Using a Single Loop

Let's solve T_4 . Should we use SOP or POS?

T_4 is a maxterm!

$$T_4 = (C_3 + C_2 + C_1 + C_0)$$

		C_3C_2			
		00	01	11	10
T_4	00	0	1	1	1
	01	1	1	1	1
	11	1	1	1	1
	10	1	1	1	1

Solve T_5 as a POS Expression

Let's solve T_5 . POS is better again.

What are the loops?
for SOP?

$$(C_3' + C_2')$$

$$(C_3' + C_1' + C_0')$$

$$\text{So } T_5 = (C_3' + C_2') \cdot (C_3' + C_1' + C_0')$$

T_5		C_3C_2			
		00	01	11	10
C_1C_0	00	1	1	0	1
	01	1	1	0	1
	11	1	1	0	0
	10	1	1	0	1

Combine T_4 and T_5 to find $U(C)$

How do we combine T_4 and T_5 to find the full upper-case checker function $U(C)$?

Remember:

- T_4 applies when $C_6C_5C_4 = 100$, and
- T_5 applies when $C_6C_5C_4 = 101$.

So ...?

- AND T_4 with $C_6C_5'C_4'$,
- AND T_5 with $C_6C_5'C_4$, and
- OR the results together.

A Good Solution, But Maybe We Can Do Less Work?

$$\text{So } U(C) = C_6 C_5' C_4' (C_3 + C_2 + C_1 + C_0) + \\ C_6 C_5' C_4 (C_3' + C_2') (C_3' + C_1' + C_0')$$

That's a pretty small and fast solution.

But we still had to do a fair bit of work.

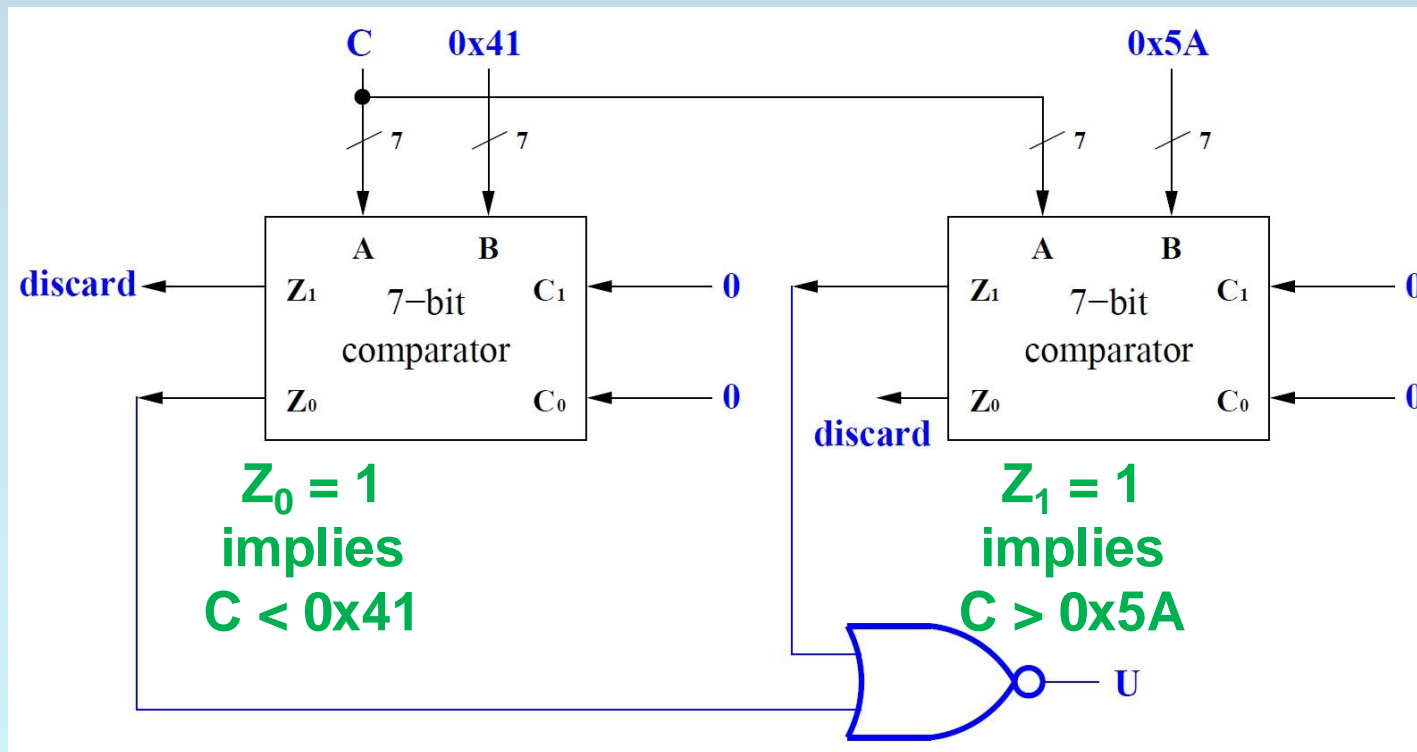
Is there an easier way?

Consider the following: to check for an upper-case letter, we need to know whether

$$C \geq 1000001 \text{ AND } C \leq 1011010$$

Use Two Comparators to Calculate U(C)

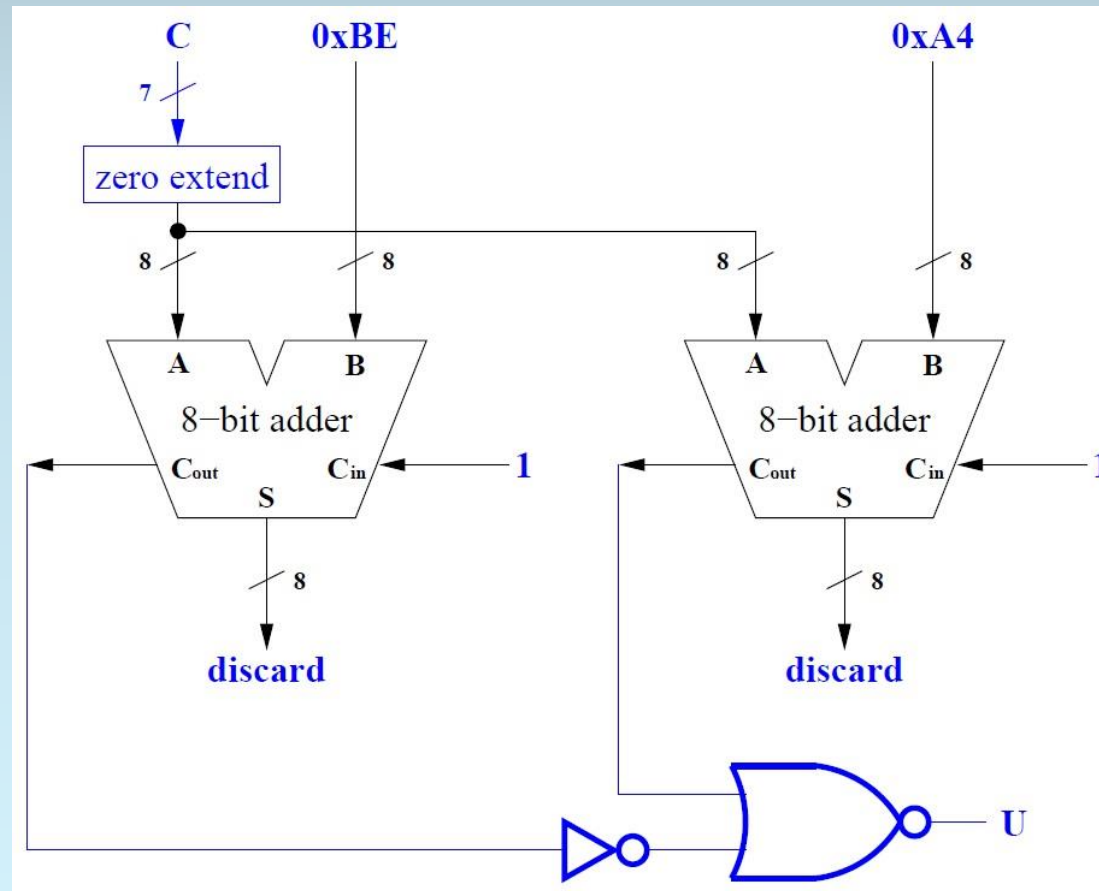
What about this approach?



Or Use Two Adders to Calculate U(C)

Or this approach?

Note that the adders are performing subtractions.



Inefficient, But Simple to Design

Quite large and slow compared with our first solution?

Consider two arguments:

1. CAD tools can optimize away much of the extra overhead.
2. Software executing on data center servers around the world executes the adder design even less efficiently, but it's constantly in use on hundreds of thousands of machines.