Numbers and Arithmetic

Kavita Bala

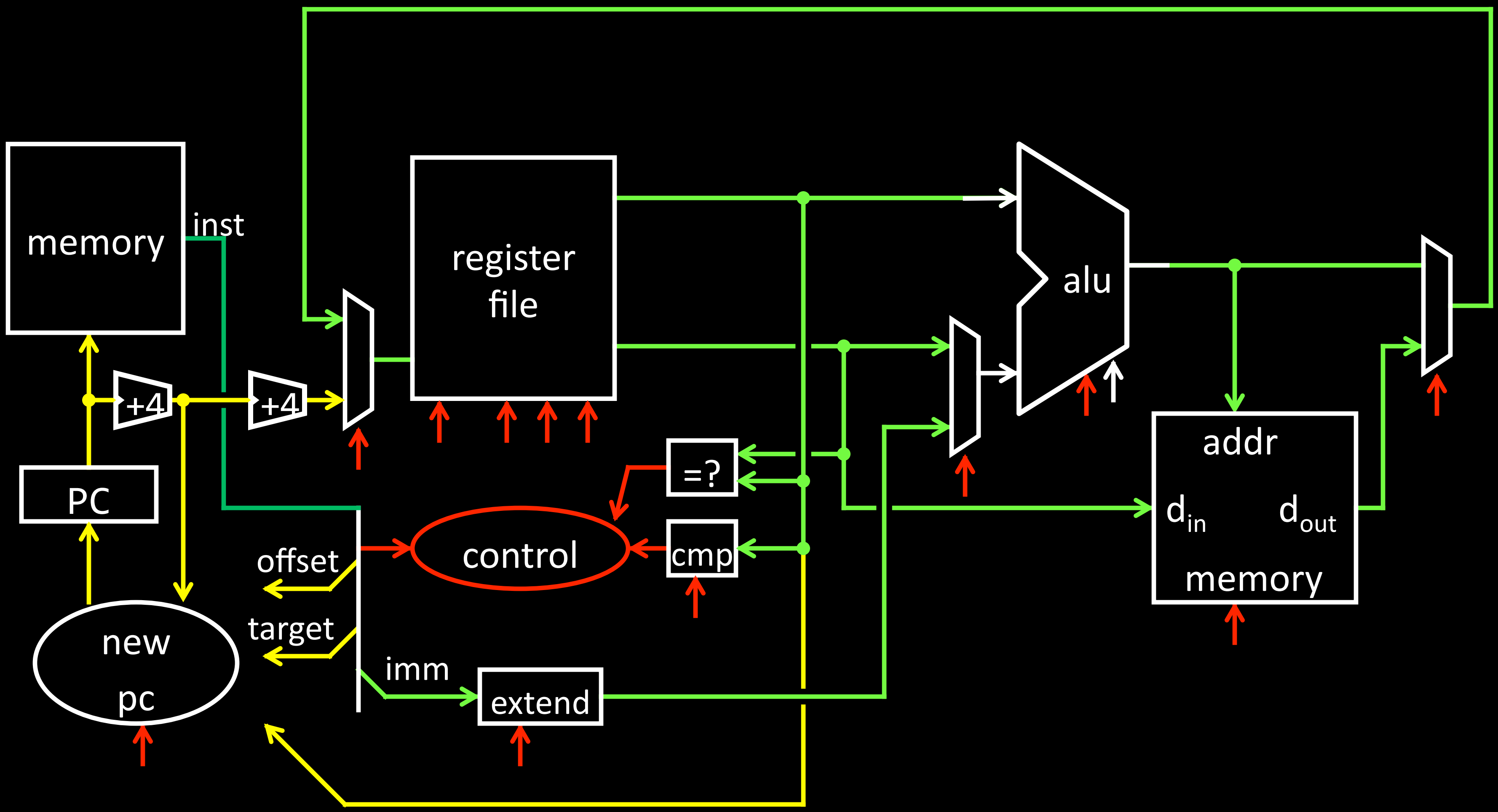
CS 3410, Spring 2014

Computer Science

Cornell University

See: P&H Chapter 2.4, 3.2, B.2, B.5, B.6

Big Picture: Building a Processor



A single cycle processor

Today’s Lecture

Binary Operations

- Number representations
- One-bit and four-bit adders
- Negative numbers and two’s complement
- Addition (two’s complement)
- Subtraction (two’s complement)
- Performance

Number Representations

Recall: binary

- Two symbols (base 2): true and false; 1 and 0
- Basis of logic circuits and all digital computers

How to represent numbers in *binary* (base 2)?

Number Representations

How to represent numbers in *binary* (base 2)?

- Know how to represent numbers in decimal (base 10)
  - E.g. $\underset{10^2}{\underline{6}}\,\underset{10^1}{\underline{3}}\,\underset{10^0}{\underline{7}}$

- Other bases
  - Base 2 — Binary $\quad \underset{2^9}{\underline{1}}\,\underset{2^8}{\underline{0}}\;\;\underset{2^7}{\underline{0}}\,\underset{2^6}{\underline{1}}\,\underset{2^5}{\underline{1}}\,\underset{2^4}{\underline{1}}\;\;\underset{2^3}{\underline{1}}\,\underset{2^2}{\underline{1}}\,\underset{2^1}{\underline{0}}\,\underset{2^0}{\underline{1}}$
  - Base 8 — Octal $\quad 0o\,\underset{8^3}{\underline{1}}\,\underset{8^2}{\underline{1}}\,\underset{8^1}{\underline{7}}\,\underset{8^0}{\underline{5}}$
  - Base 16 — Hexadecimal $\quad 0x\,\underset{16^2}{\underline{2}}\,\underset{16^1}{\underline{7}}\,\underset{16^0}{\underline{d}}$

Number Representations

Dec (base 10)	Bin (base 2)	Oct (base 8)	Hex (base 16)
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	a
11	1011	13	b
12	1100	14	c
13	1101	15	d
14	1110	16	e
15	1111	17	f
16	1 0000	20	10
17	1 0001	21	11
18	1 0010	22	12
.	.	.	.
.	.	.	.
99			
100			

Number Representations

Dec (base 10)	Bin (base 2)	Oct (base 8)	Hex (base 16)
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	a
11	1011	13	b
12	1100	14	c
13	1101	15	d
14	1110	16	e
15	1111	17	f
16	1 0000	20	10
17	1 0001	21	11
18	1 0010	22	12
.	.	.	.
.	.	.	.
99			
100			

0b 1111 1111 =

0b 1 0000 0000 =

0o 77 =

0o 100 =

0x ff =

0x 100 =

Number Representations

Dec (base 10)	Bin (base 2)	Oct (base 8)	Hex (base 16)
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	a
11	1011	13	b
12	1100	14	c
13	1101	15	d
14	1110	16	e
15	1111	17	f
16	1 0000	20	10
17	1 0001	21	11
18	1 0010	22	12
.	.	.	.
.	.	.	.
99			
100			

0b 1111 1111 = 255

0b 1 0000 0000 = 256

0o 77 = 63

0o 100 = 64

0x ff = 255

0x 100 = 256

Number Representations

How to convert a number between different bases?

Base conversion via repetitive division

Divide by base, write remainder, move left with quotient

$$\begin{array}{rl|c|l}
\cline{3-3}
637 \div 8 = 79 & \text{remainder} & 5 & \text{lsb (least significant bit)} \\
79 \div 8 = 9 & \text{remainder} & 7 & \\
9 \div 8 = 1 & \text{remainder} & 1 & \\
1 \div 8 = 0 & \text{remainder} & 1 & \text{msb (most significant bit)} \\
\cline{3-3}
\end{array}$$

$$637 = 0o\ \underset{\text{msb}}{1}17\underset{\text{lsb}}{5}$$

Number Representations

Convert a base 10 number to a base 2 number

Base conversion via repetitive division

- Divide by base, write remainder, move left with quotient
- 637 ÷ 2 = 318 remainder 1 lsb (least significant bit)
- 318 ÷ 2 = 159 remainder 0
- 159 ÷ 2 = 79 remainder 1
- 79 ÷ 2 = 39 remainder 1
- 39 ÷ 2 = 19 remainder 1
- 19 ÷ 2 = 9 remainder 1
- 9 ÷ 2 = 4 remainder 1
- 4 ÷ 2 = 2 remainder 0
- 2 ÷ 2 = 1 remainder 0
- 1 ÷ 2 = 0 remainder 1 msb (most significant bit)

637 = 10 0111 1101 (can also be written as 0b10 0111 1101)

msb lsb

Range of Values

n bits: 0 to 2^n-1

E.g., 4 bits 0000 to 1111 is 0 to 15

$$(x31 \times 2^{31}) + (x30 \times 2^{30}) + (x29 \times 2^{29}) + \ldots + (x1 \times 2^{1}) + (x0 \times 2^{0})$$

Number Representations

Convert a base 2 number to base 8 (oct) or 16 (hex)

Binary to Hexadecimal

Convert each nibble (group of 4 bits) from binary to hex

A nibble (4 bits) ranges in value from 0…15, which is one hex digit

- Range: 0000…1111 (binary) => 0x0 …0xF (hex) => 0…f

- E.g. 0b10   0111   1101
- 0b10 = 0x2
- 0b0111 = 0x7
- 0b1101 = 0xd
- Thus, 637 = 0x27d = 0b10 0111 1101

Similarly for base 2 to base 8

Takeaway

Digital computers are implemented via logic circuits and thus represent *all* numbers in binary (base 2)

We (humans) often write numbers as decimal and hexadecimal for convenience, so need to be able to convert to binary and back (to understand what computer is doing!)

Next

Binary Arithmetic: Add and Subtract two binary numbers

Binary Addition

How do we do arithmetic in binary?

$$\begin{array}{r} 1 \\ 183 \\ +\ 254 \\ \hline 437 \end{array}$$

Addition works the same way regardless of base

- Add the digits in each position
- Propagate the carry

Carry-in

Carry-out

$$\begin{array}{r} 11\boxed{1} \\ 00\boxed{1}110 \\ +\ 01\boxed{1}100 \\ \hline 10\boxed{1}010 \end{array}$$

Unsigned binary addition is pretty easy

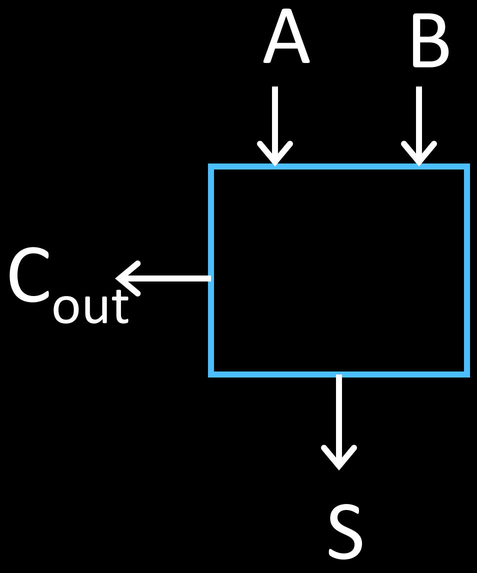
- Combine two bits at a time
- Along with a carry

Binary Addition

Binary addition requires

- Add of ***two bits*** PLUS ***carry-in***
- Also, ***carry-out*** if necessary

1-bit Adder

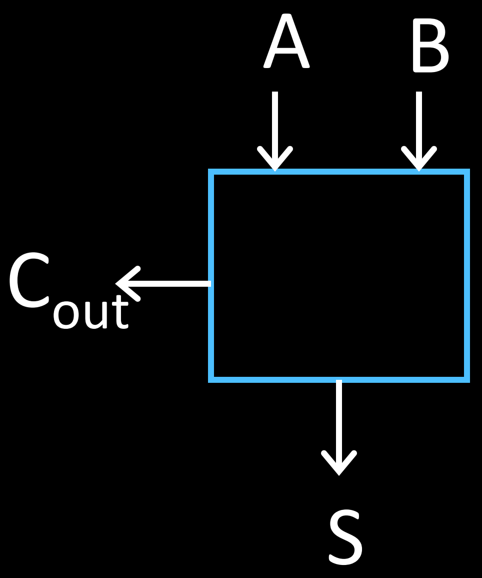


A	B	C_{out}	S
0	0		
0	1		
1	0		
1	1		

Half Adder

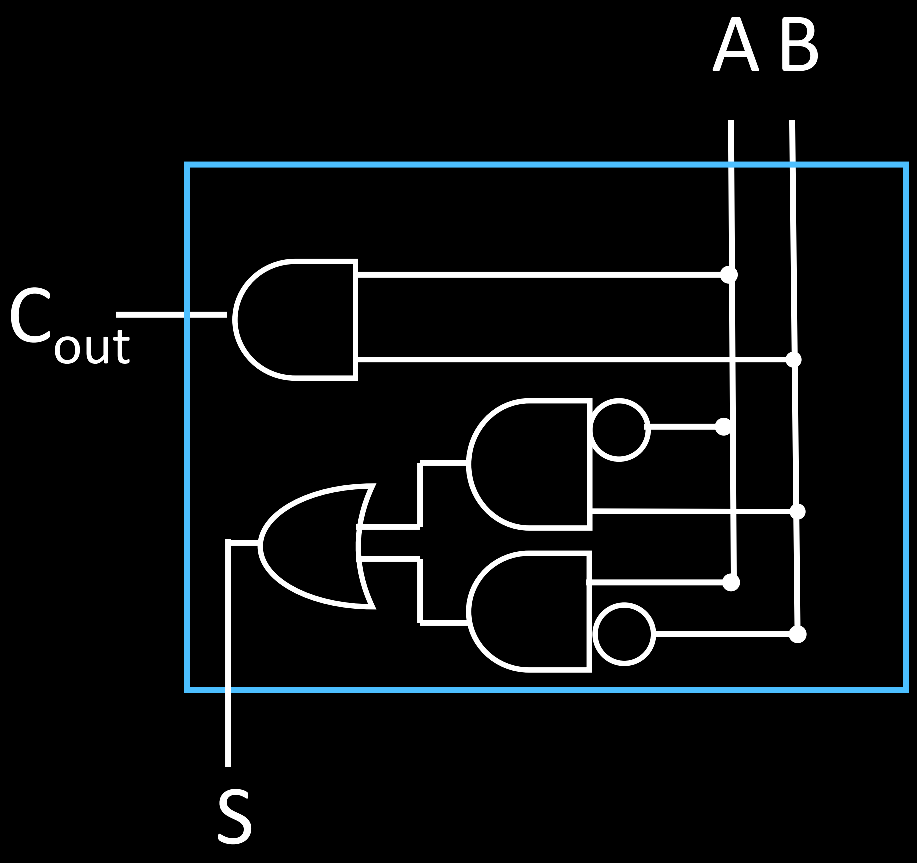
- Adds two 1-bit numbers
- Computes 1-bit result and 1-bit carry
- No carry-in

1-bit Half Adder

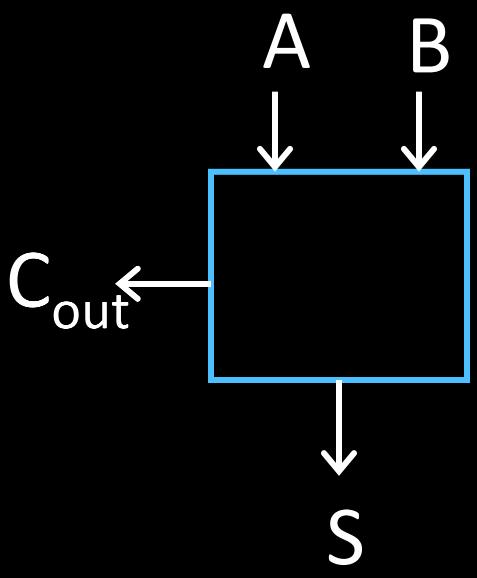


$$C_{out} = AB$$
$$S = A \oplus B = \bar{A}B + A\bar{B}$$

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

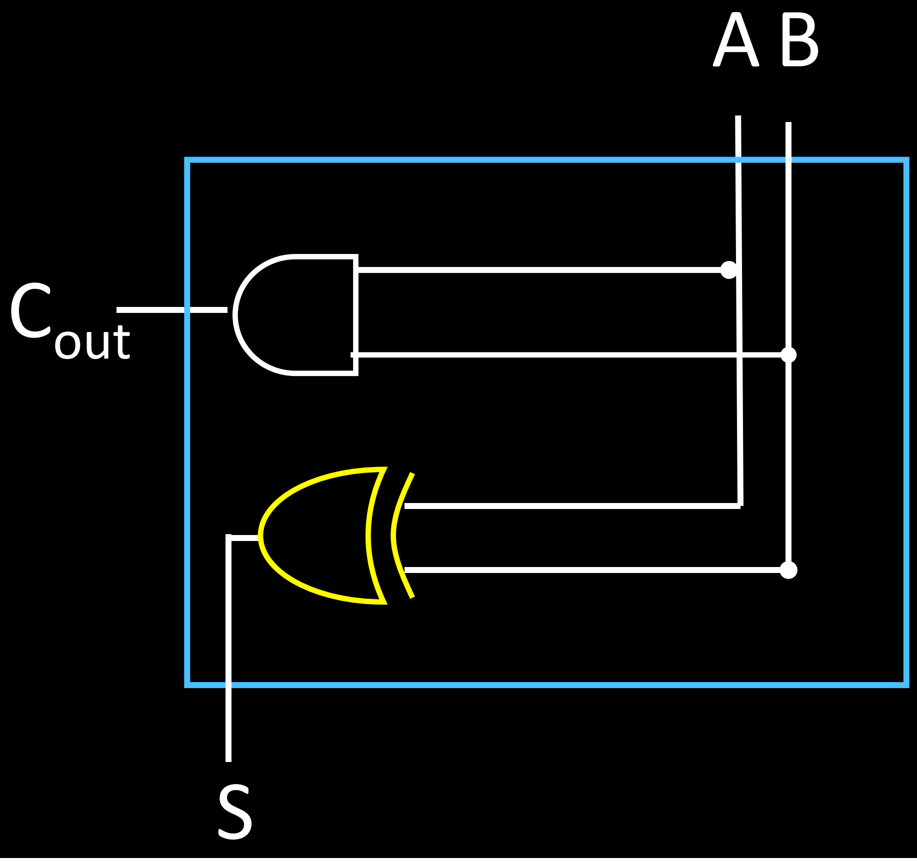


1-bit Half Adder

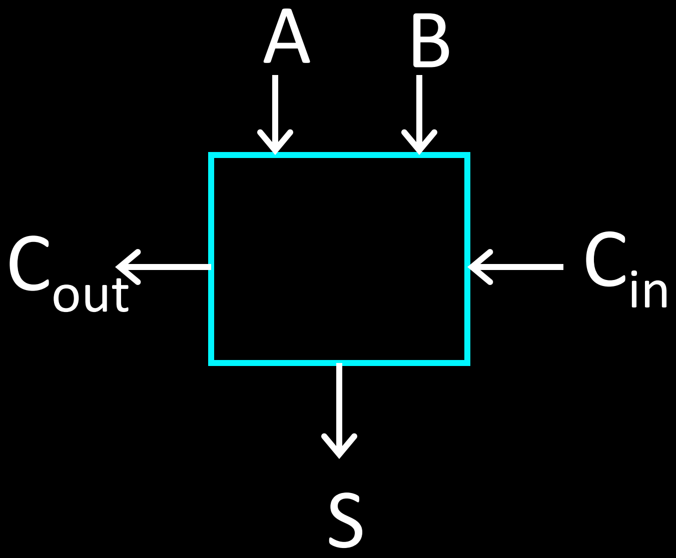


$$C_{out} = AB$$
$$S = A \oplus B = \bar{A}B + A\bar{B}$$

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



1-bit Adder with Carry

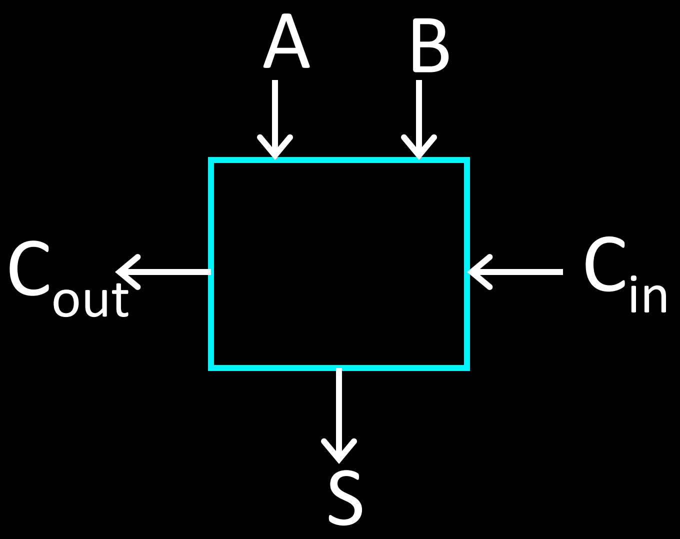


Full Adder

- Adds three 1-bit numbers
- Computes 1-bit result and 1-bit carry
- Can be cascaded

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

1-bit Adder with Carry

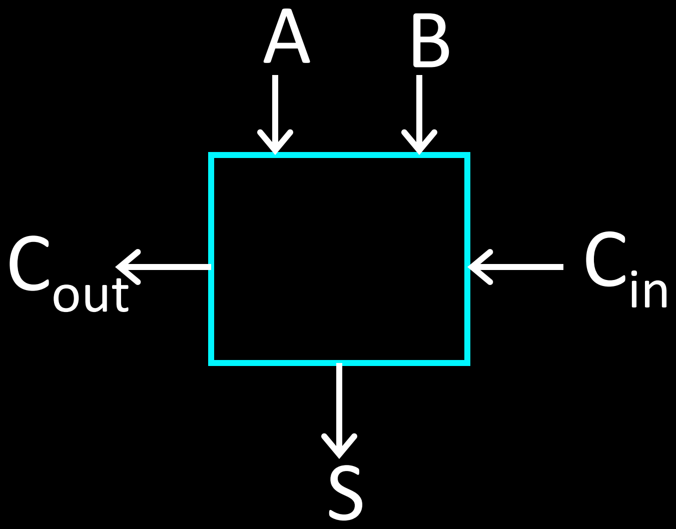


Full Adder

- Adds three 1-bit numbers
- Computes 1-bit result and 1-bit carry
- Can be cascaded

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

1-bit Adder with Carry

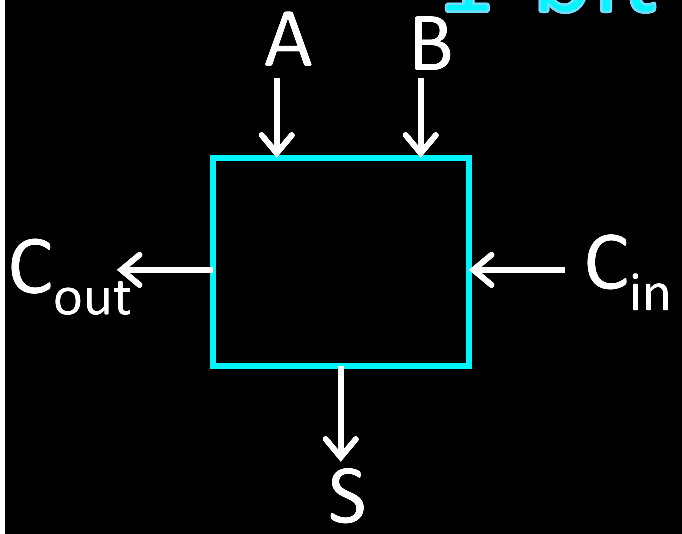


A	B	C_{in}	C_{out}	S
0	0	0	0	0
0	1	0	0	1
1	0	0	0	1
1	1	0	1	0
0	0	1	0	1
0	1	1	1	0
1	0	1	1	0
1	1	1	1	1

$$S = \overline{A}B\overline{C_{in}} + A\overline{B}\,\overline{C_{in}} + \overline{A}\,\overline{B}C_{in} + ABC_{in}$$

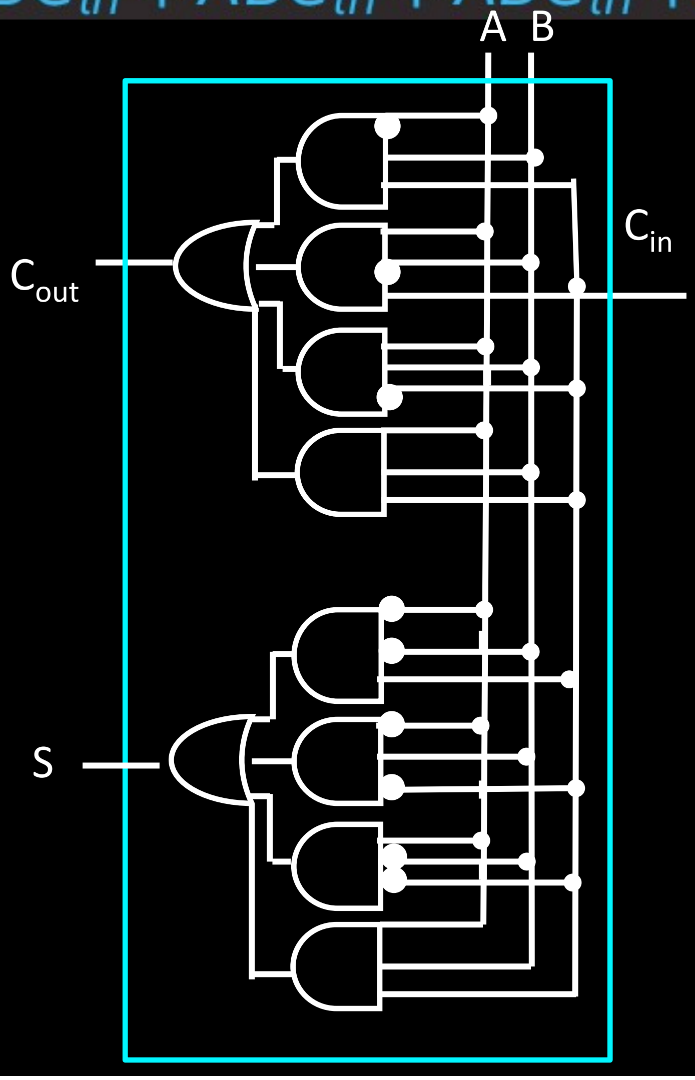
$$C_{out} = AB\overline{C_{in}} + \overline{A}BC_{in} + A\overline{B}C_{in} + ABC_{in}$$

1-bit Adder with Carry

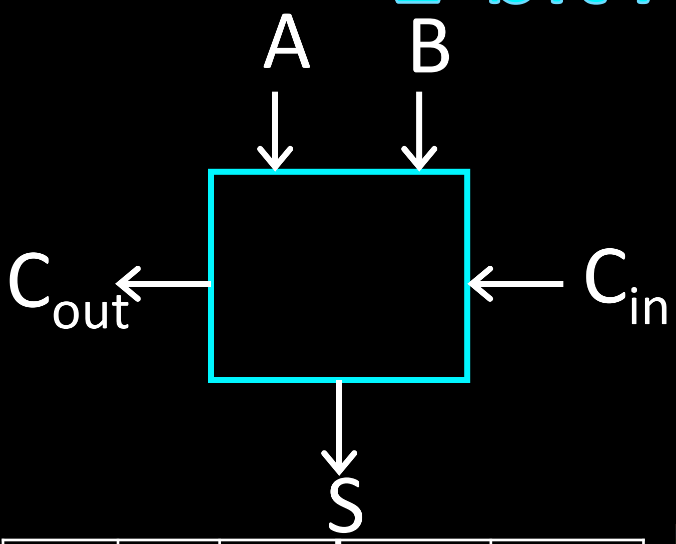


$$S = \overline{A}B\overline{C_{in}} + A\overline{B}\,\overline{C_{in}} + \overline{A}\,\overline{B}C_{in} + ABC_{in}$$
$$C_{out} = AB\overline{C_{in}} + \overline{A}BC_{in} + A\overline{B}C_{in} + ABC_{in}$$

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



1-bit Adder with Carry

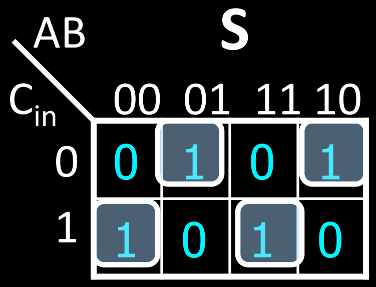


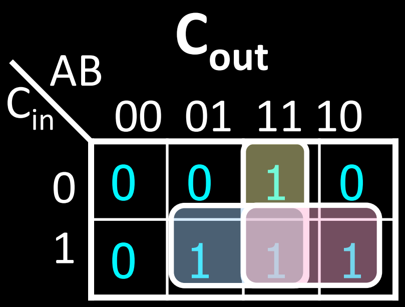
Using Karnaugh maps

$$S = \overline{A}B\overline{C_{in}} + A\overline{B}\,\overline{C_{in}} + \overline{A}\,\overline{B}C_{in} + ABC_{in}$$

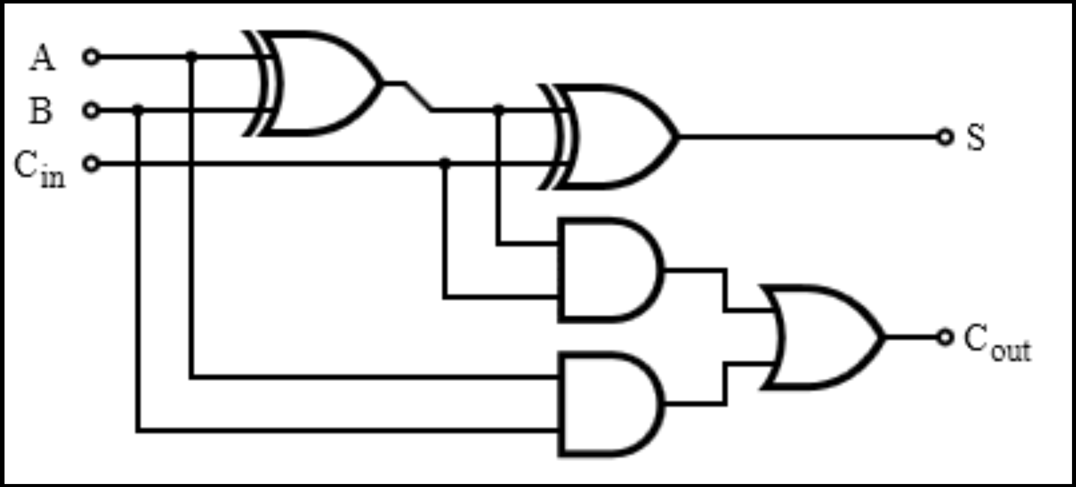
$$C_{out} = AB + AC_{in} + BC_{in}$$

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



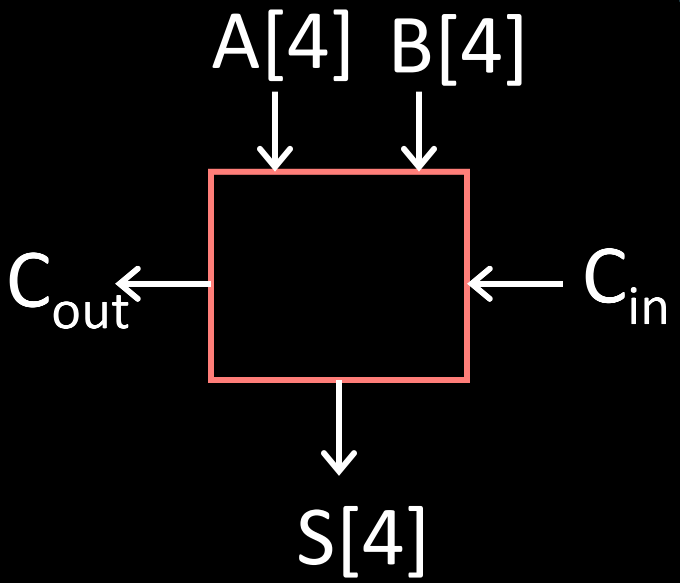


Lab0 1-bit adder



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

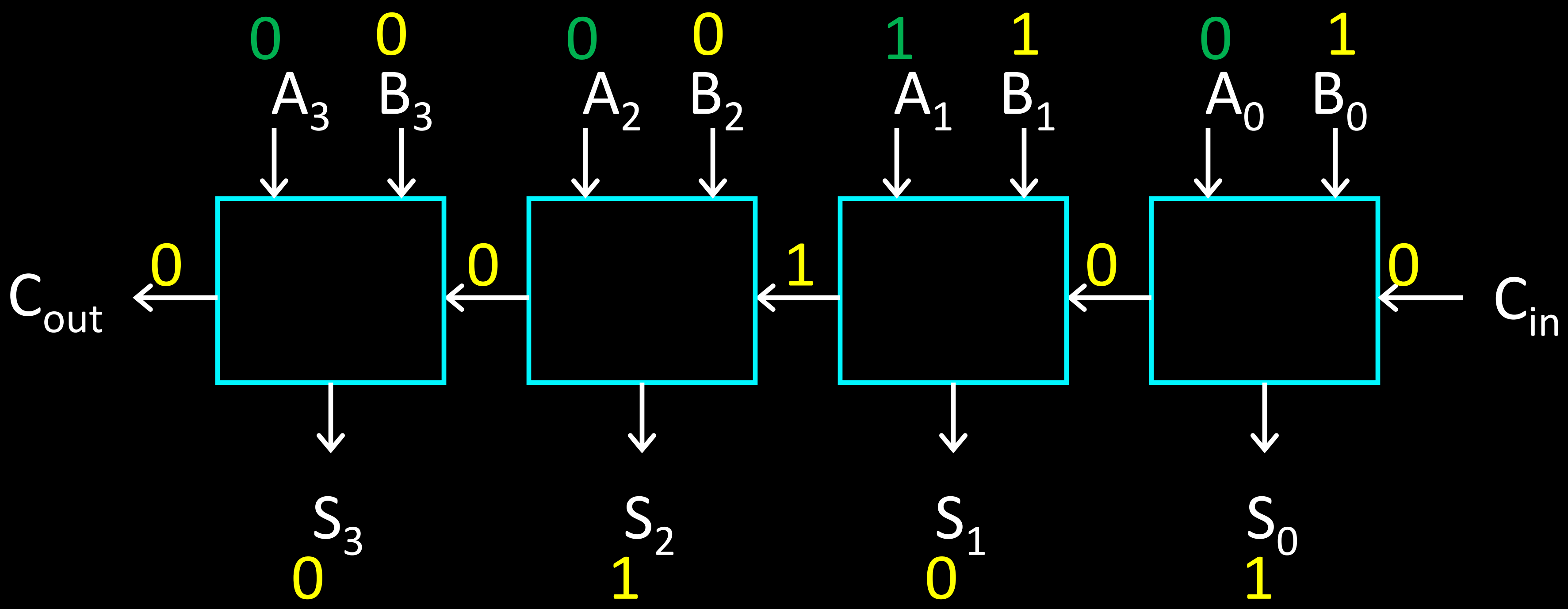
4-bit Adder



4-Bit Full Adder

- Adds two 4-bit numbers and carry in
- Computes 4-bit result and carry out
- Can be cascaded

4-bit Adder



- Adds two 4-bit numbers, along with carry-in
- Computes 4-bit result and carry out

- Carry-out = overflow indicates result does not fit in 4 bits

Takeaway

Digital computers are implemented via logic circuits and thus represent *all* numbers in binary (base 2)

We (humans) often write numbers as decimal and hexadecimal for convenience, so need to be able to convert to binary and back (to understand what computer is doing!)

Adding two 1-bit numbers generalizes to adding two numbers of any size since 1-bit full adders can be cascaded

Next Goal

How do we subtract two binary numbers?

Equivalent to adding with a negative number

How do we represent negative numbers?

First Attempt: Sign/Magnitude Representation

First Attempt: Sign/Magnitude Representation

- 1 bit for sign (0=positive, 1=negative)
- N-1 bits for magnitude

<u>0</u>111 = 7

<u>1</u>111 = -7

Problem?

- Two zero's: +0 different than -0
- Complicated circuits

<u>0</u>000 = +0

<u>1</u>000 = -0

Others attempts

One's complement

IBM 7090

Two's Complement Representation

What is used: Two's Complement Representation

To negate any number:

- complement *all* the bits (i.e. flip all the bits)
- then add 1

Nonnegative numbers are represented as usual

- 0 = 0000, 1 = 0001, 3 = 0011, 7 = 0111

To negate any number:

- complement *all* the bits (i.e. flip all the bits)
- then add 1
- -1: 1 $\Rightarrow$ 0001 $\Rightarrow$ 1110 $\Rightarrow$ 1111
- -3: 3 $\Rightarrow$ 0011 $\Rightarrow$ 1100 $\Rightarrow$ 1101
- -7: 7 $\Rightarrow$ 0111 $\Rightarrow$ 1000 $\Rightarrow$ 1001
- -0: 0 $\Rightarrow$ 0000 $\Rightarrow$ 1111 $\Rightarrow$ 0000 (this is good, -0 = +0)

Two's Complement Representation

One more example. How do we represent -20?

$$\begin{array}{rl} 20 = & 0001\ \ 0100 \\ \overline{20} = & 1110\ \ 1011 \\ & +1 \\ \hline -20 = & 1110\ \ 1100 \end{array}$$

Two's Complement

Non-negatives (as usual):	Negatives (two's complement: flip then add 1):	
+0 = 0000	flip = 1111	-0 = 0000
+1 = 0001	flip = 1110	-1 = 1111
+2 = 0010	flip = 1101	-2 = 1110
+3 = 0011	flip = 1100	-3 = 1101
+4 = 0100	flip = 1011	-4 = 1100
+5 = 0101	flip = 1010	-5 = 1011
+6 = 0110	flip = 1001	-6 = 1010
+7 = 0111	flip = 1000	-7 = 1001
	flip = 0111	-8 = 1000
+8 = 1000		

Two's Complement Facts

Signed two's complement

- Negative numbers have leading 1's
- zero is unique: +0 = - 0
- wraps from largest positive to largest negative

N bits can be used to represent

- unsigned: range 0…2^N-1
  - eg: 8 bits ⇒ 0…255
- signed (two's complement): -(2^{N-1})…(2^{N-1} - 1)
  - ex: 8 bits ⇒ (1000 000) … (0111 1111)
  - -128 … 127

Sign Extension & Truncation

Extending to larger size

- 1111 = -1
- 1111 1111 = -1
- 0111 = 7
- 0000 0111 = 7

Truncate to smaller size

- 0000 1111 = 15
- BUT, ~~0000~~ 1111 = 1111 = -1

Two’s Complement Addition

Addition with two’s complement signed numbers

Perform addition as usual, regardless of sign
(it just works)

Examples

- 1 + -1 =
- -3 + -1 =
- -7 + 3 =
- 7 + (-3) =

Two's Complement Addition

Addition with two's complement signed numbers

Perform addition as usual, regardless of sign
(it just works)

Examples

- 1 + -1 = 0001 + 1111 = 0000 (0)
- -3 + -1 = 1101 + 1111 = 1100 (-4)
- -7 + 3 = 1001 + 0011 = 1100 (-4)
- 7 + (-3) = 0111 + 1101 = 0100 (4)
- What is wrong with the following additions?
  - 7 + 1, -7 + -3, -7 + -1
  - 1000 overflow, 1 0110 overflow, 1000 fine

Binary Subtraction

Why create a new circuit?

Just use addition using two's complement math

- How?

Binary Subtraction

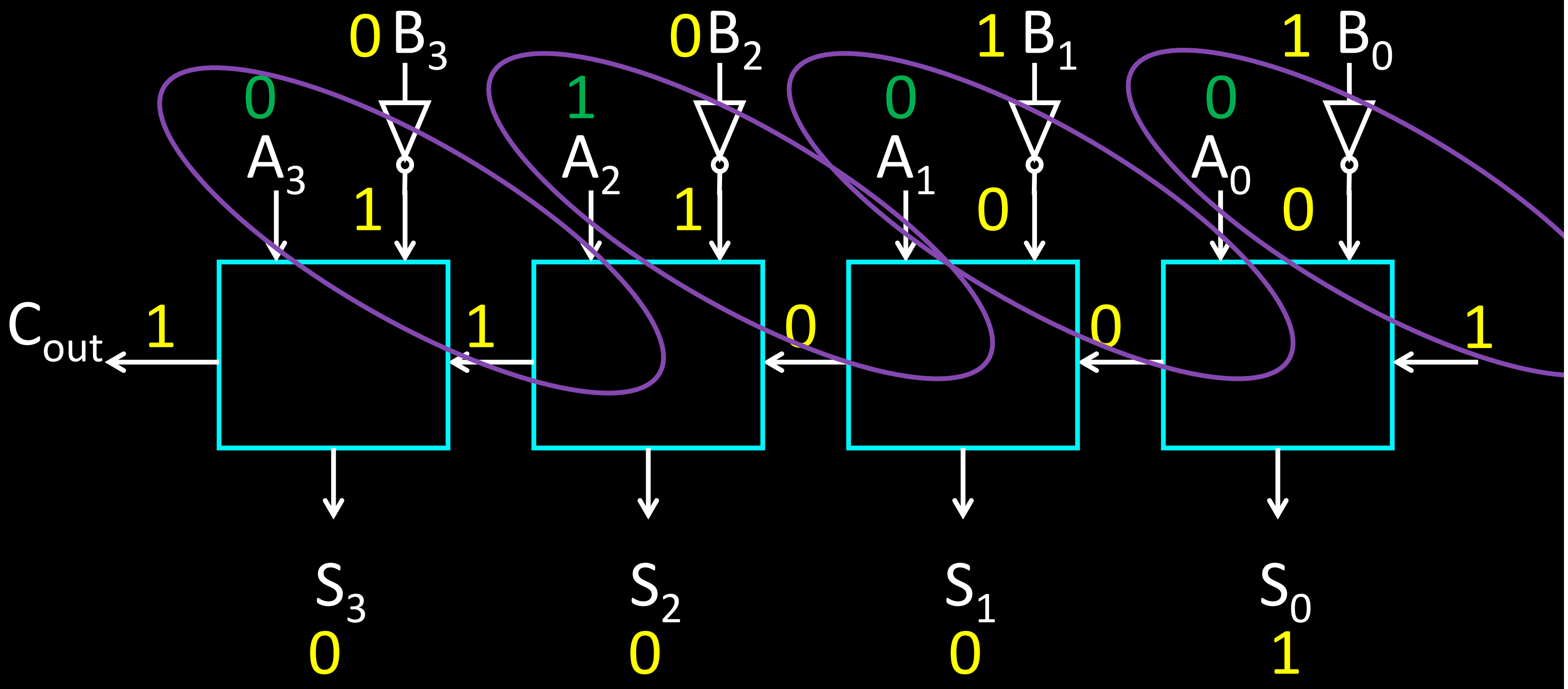
Two's Complement Subtraction

- Subtraction is simply addition,

  where one of the operands has been negated

  – Negation is done by inverting all bits and adding one

    $A - B = A + (-B) = A + (\overline{B} + 1)$



Takeaway

Digital computers are implemented via logic circuits and thus represent *all* numbers in binary (base 2).

We (humans) often write numbers as decimal and hexadecimal for convenience, so need to be able to convert to binary and back (to understand what computer is doing!).

Adding two 1-bit numbers generalizes to adding two numbers of any size since 1-bit full adders can be cascaded.

Using Two's complement number representation simplifies adder Logic circuit design (0 is unique, easy to negate)

Subtraction is simply adding, where one operand is negated (two's complement; to negate just flip the bits and add 1)

Next Goal

In general, how do we detect and handle overflow?

Overflow

When can overflow occur?

- adding a negative and a positive?
- adding two positives?
- adding two negatives?

Overflow

When can overflow occur?

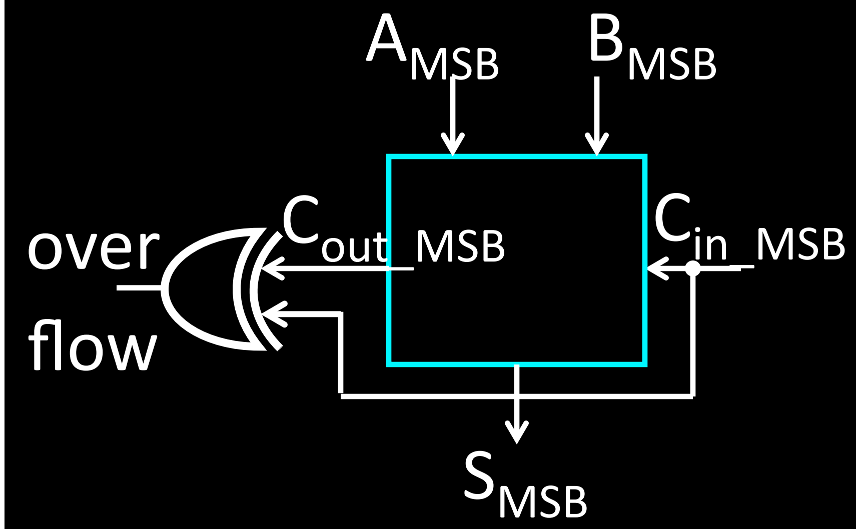
- adding a negative and a positive?
  - Overflow *cannot occur* (Why?)
  - Always subtract larger magnitude from smaller
- adding two positives?
  - Overflow *can occur* (Why?)
  - Precision: Add two positives, and get a negative number!
- adding two negatives?
  - Overflow *can occur* (Why?)
  - Precision: add two negatives, get a positive number!

Rule of thumb:

- Overflow happens iff
  carry in to msb != carry out of msb

Overflow

When can overflow occur?



MSB

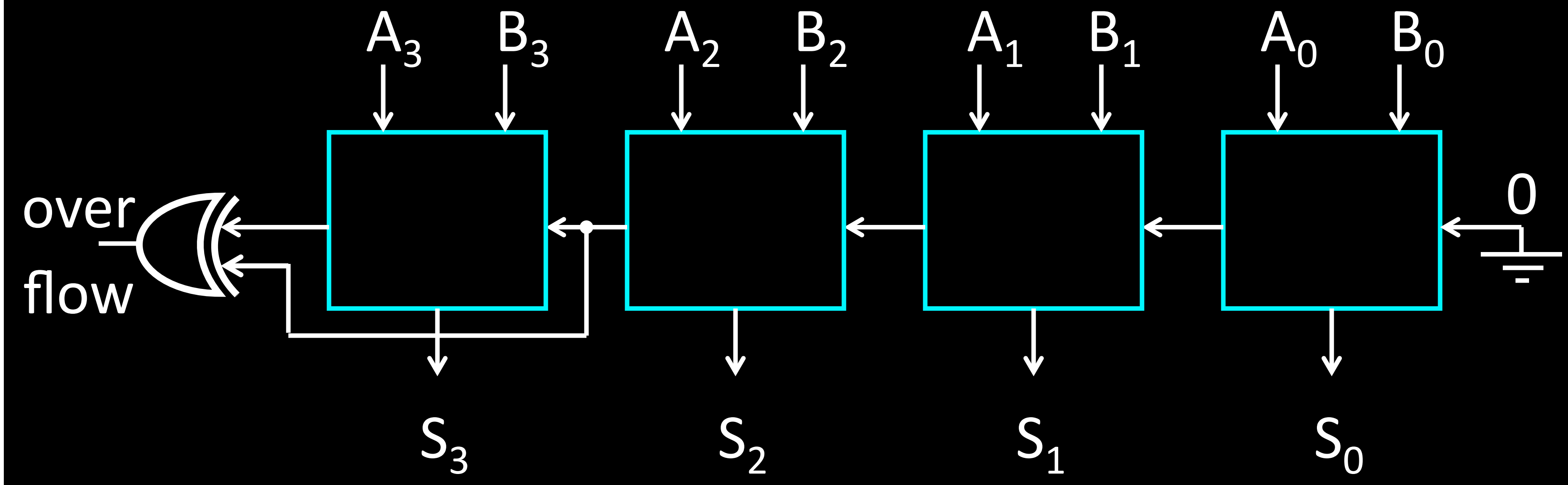
A	B	C_{in}	C_{out}	S	
0	0	0	0	0	
0	0	1	0	1	Wrong Sign
0	1	0	0	1	
0	1	1	1	0	
1	0	0	0	1	
1	0	1	1	0	
1	1	0	1	0	Wrong Sign
1	1	1	1	1	

Rule of thumb:

- Overflow happened iff
  carry into msb != carry out of msb

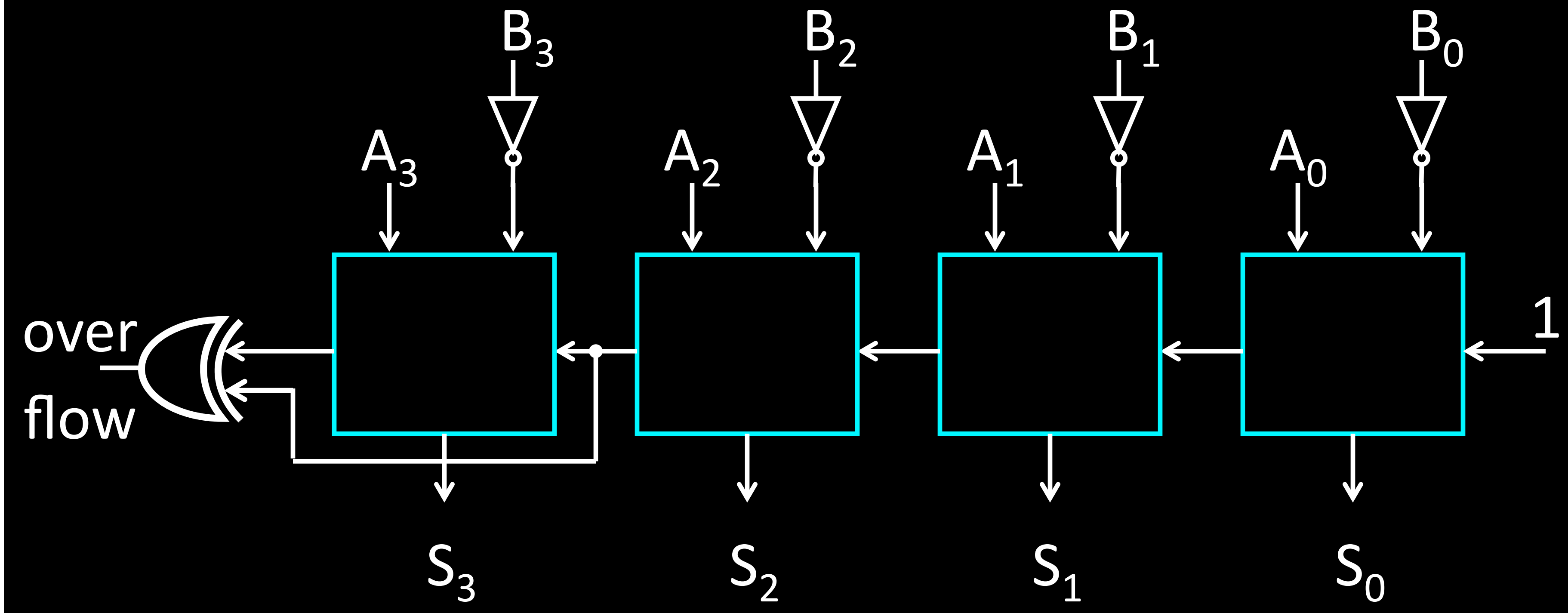
Two's Complement Adder

Two's Complement Adder with overflow detection



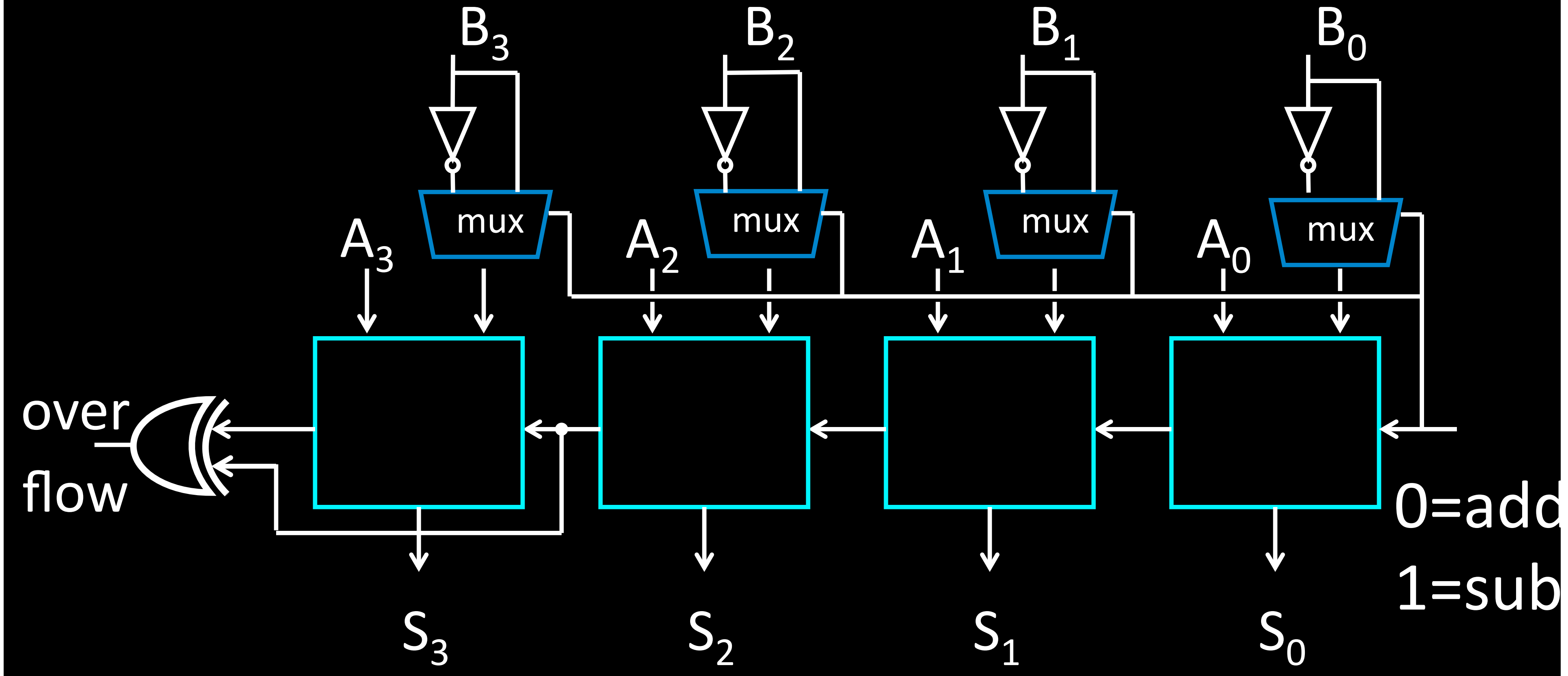
Two's Complement Adder

Two's Complement Subtraction with overflow detection



Two's Complement Adder

Two's Complement Adder with overflow detection



Takeaway

Digital computers are implemented via logic circuits and thus represent *all* numbers in binary (base 2)

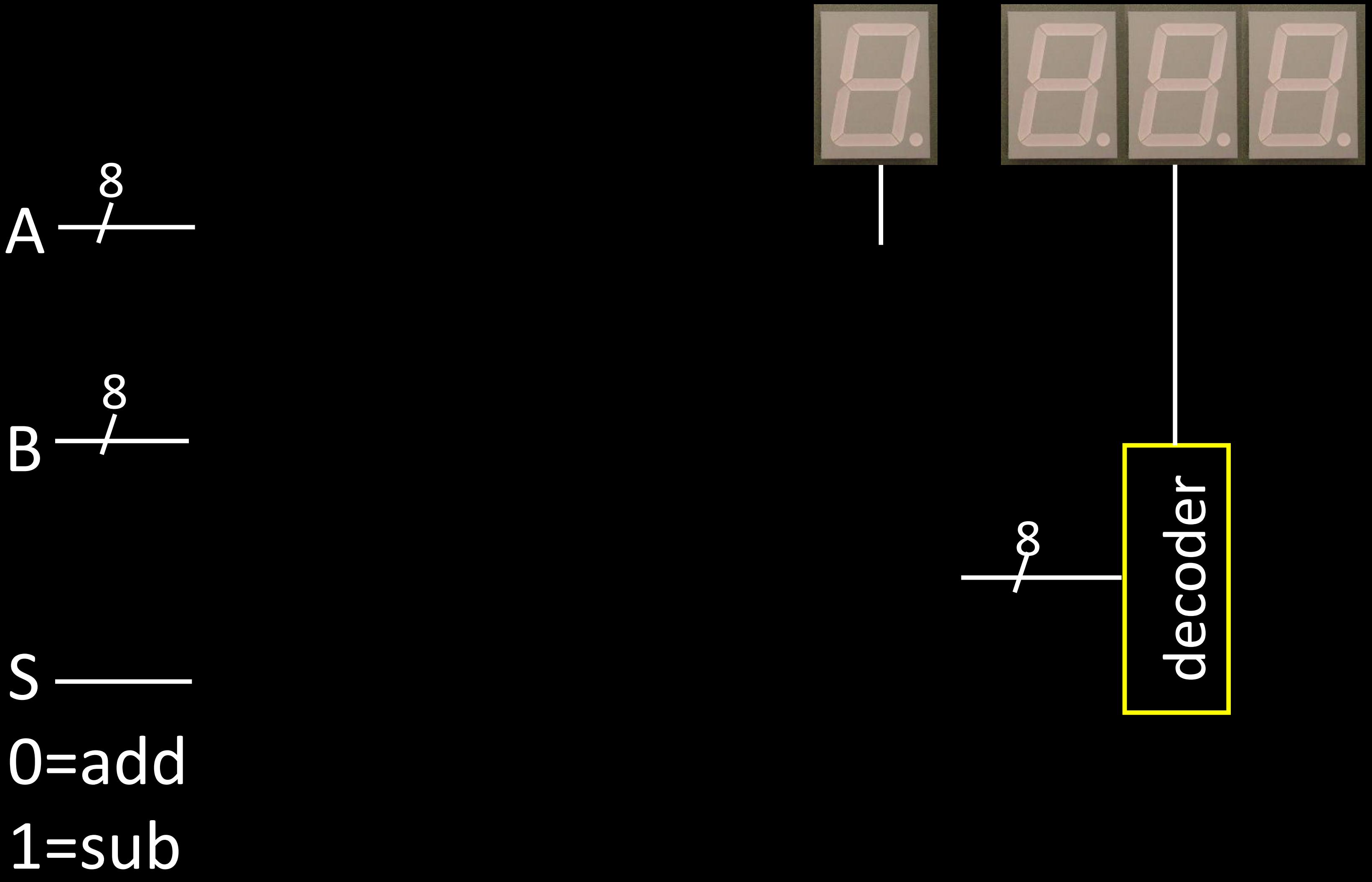
We (humans) often write numbers as decimal and hexadecimal for convenience, so need to be able to convert to binary and back (to understand what computer is doing!)

Adding two 1-bit numbers generalizes to adding two numbers of any size since 1-bit full adders can be cascaded

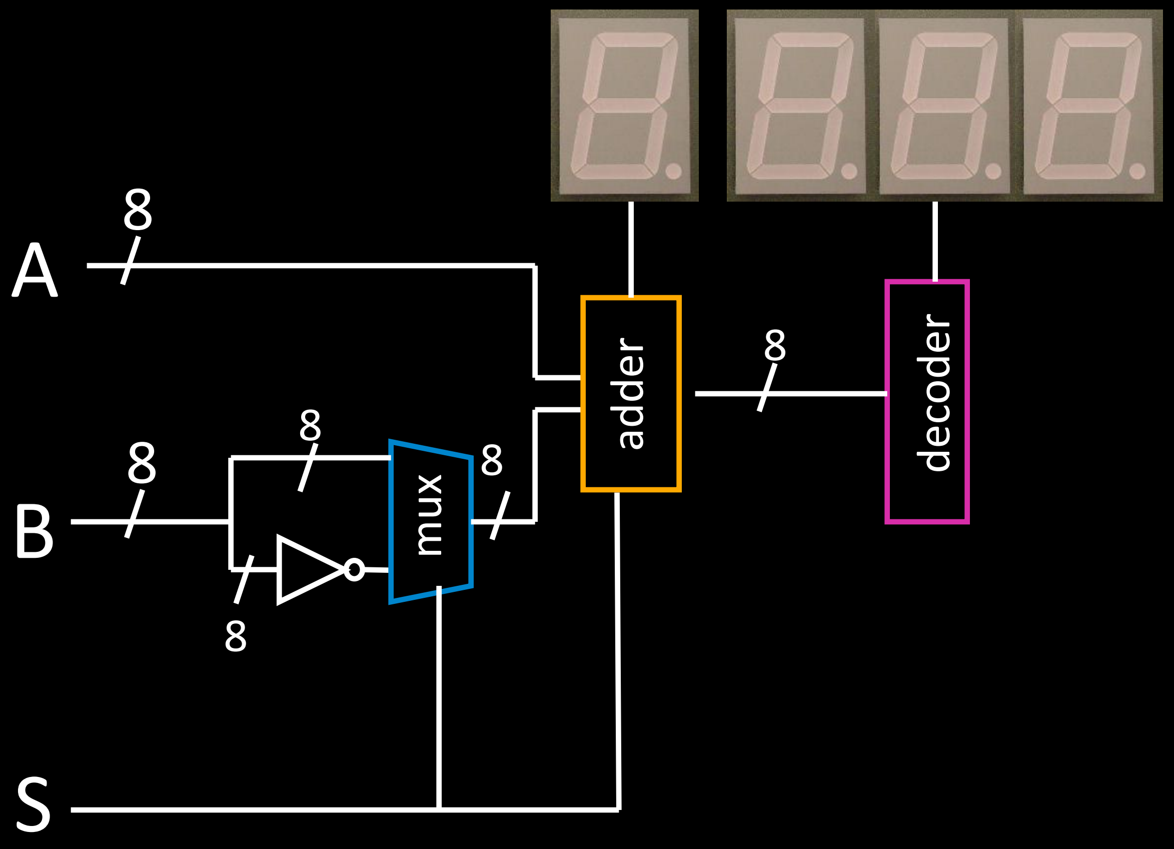
Using two's complement number representation simplifies adder Logic circuit design (0 is unique, easy to negate). Subtraction is simply adding, where one operand is negated (two's complement; to negate just flip the bits and add 1)

Overflow if sign of operands A and B != sign of result S. Can detect overflow by testing C_{in} != C_{out} of the most significant bit (msb), which only occurs when previous statement is true

A Calculator



A Calculator

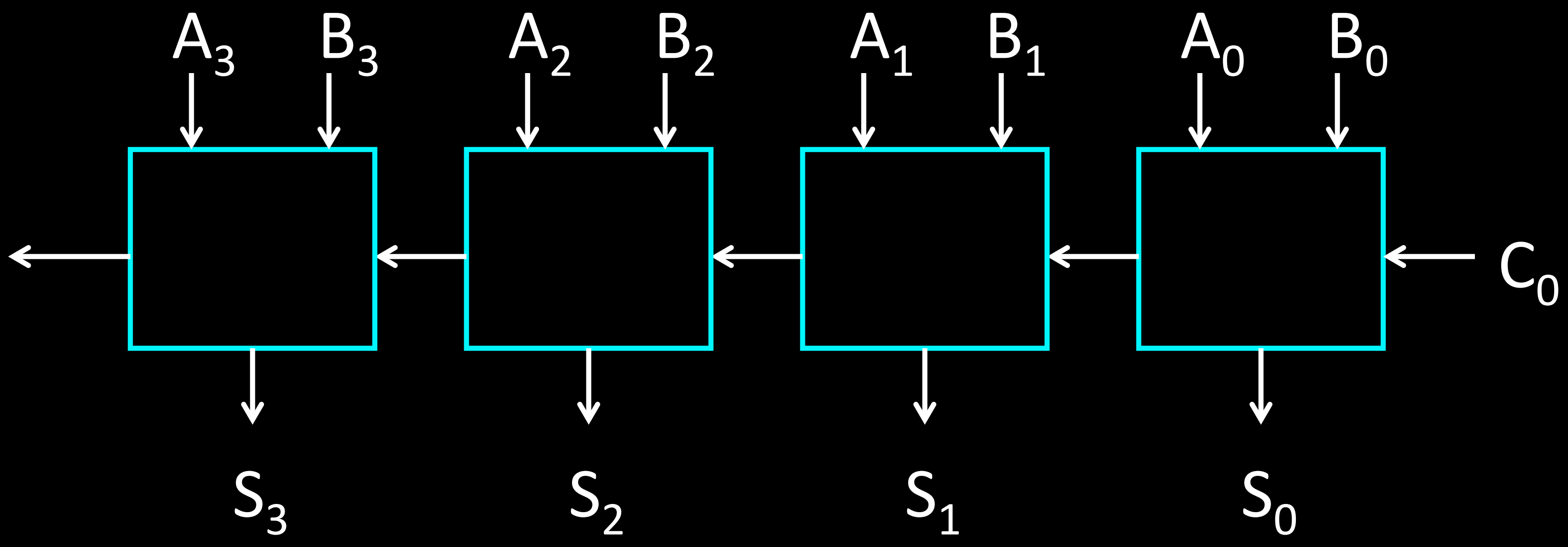


0=add

1=sub

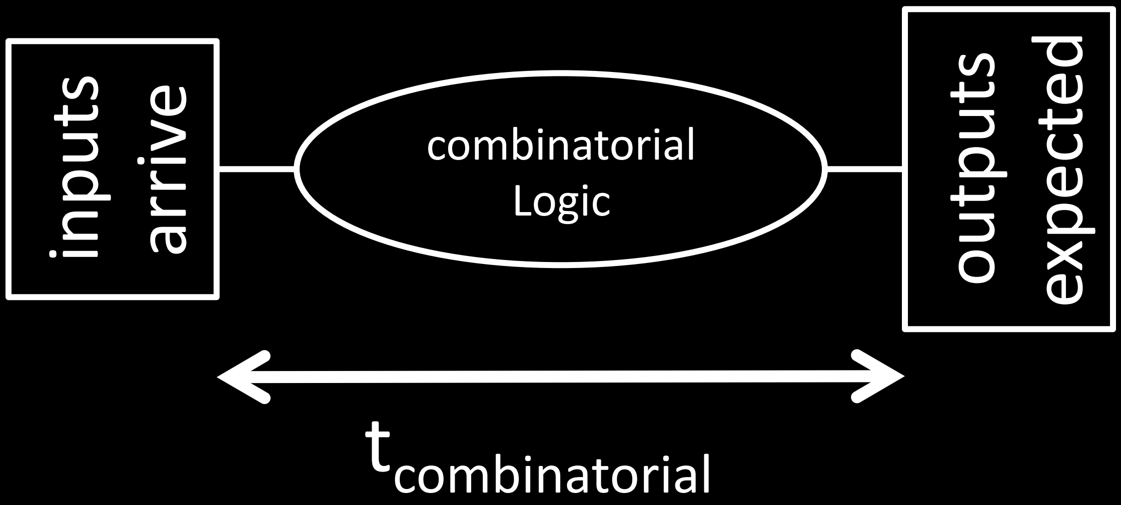
Efficiency and Generality

- Is this design fast enough?
- Can we generalize to 32 bits? 64? more?

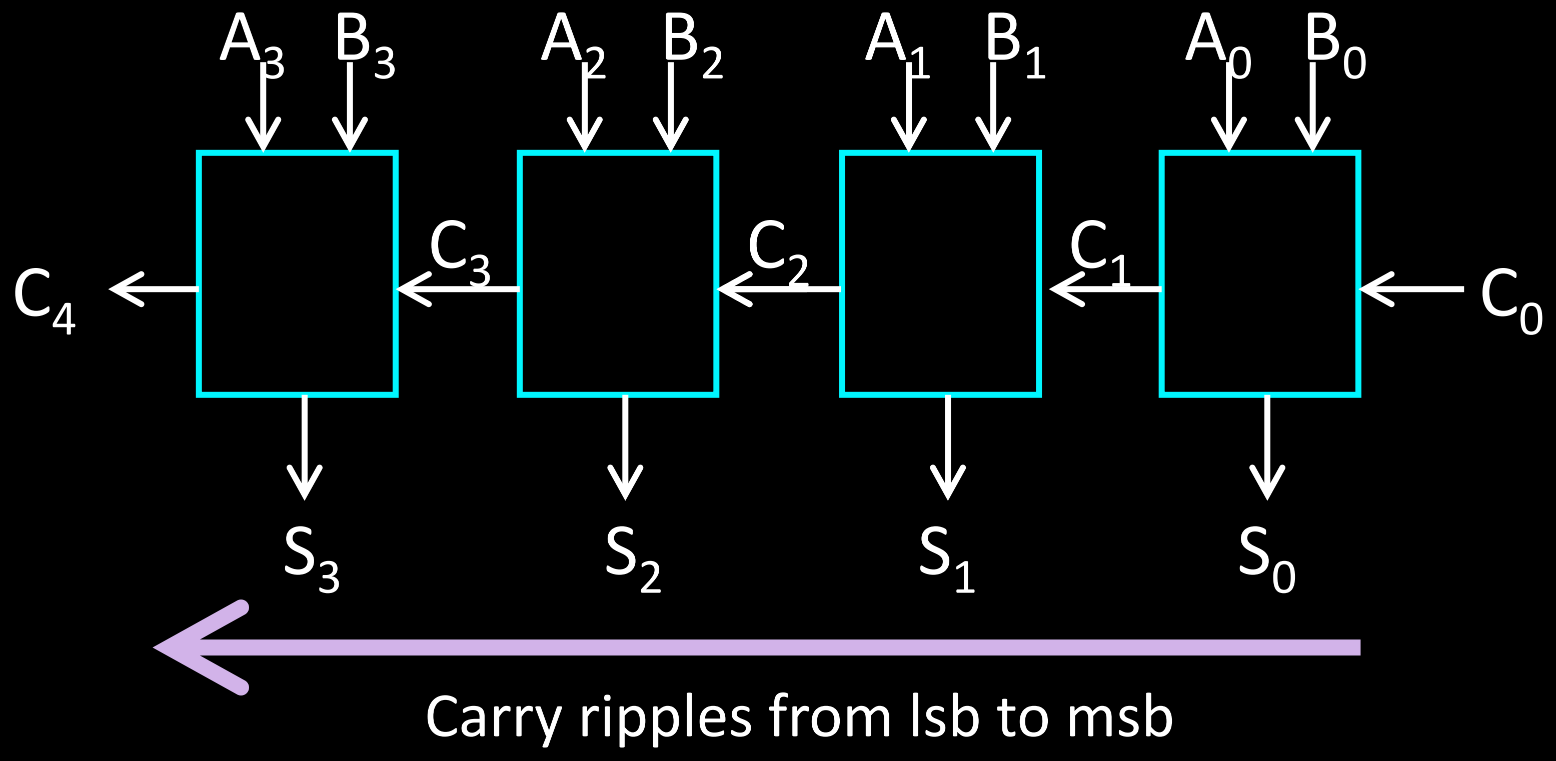


Performance

Speed of a circuit is affected by the number of gates in series (on the *critical path* or the *deepest level of logic*)



4-bit Ripple Carry Adder



- First full adder, 2 gate delay
- Second full adder, 2 gate delay
- …

Today's Lecture

Binary Operations

- Number representations
- One-bit and four-bit adders
- Negative numbers and two's complement
- Addition (two's complement)
- Subtraction (two's complement)
- Performance

Summary

We can now implement combinatorial logic circuits

- Design each block
  - Binary encoded numbers for compactness
- Decompose large circuit into manageable blocks
  - 1-bit Half Adders, 1-bit Full Adders,

    n-bit Adders via cascaded 1-bit Full Adders, ...
- Can implement circuits using NAND or NOR gates
- Can implement gates using use PMOS and NMOS-transistors
- Can add and subtract numbers (in two's complement)!
- Next time, state and finite state machines...

Administrivia

Check online syllabus/schedule

- http://www.cs.cornell.edu/Courses/CS3410/2014sp/schedule.html
- Slides and Reading for lectures
- Office Hours
- Homework and Programming Assignments

Schedule is subject to change