COMSM1302
Overview of Computer Architecture

Lecture 4

Simple devices

In this lecture

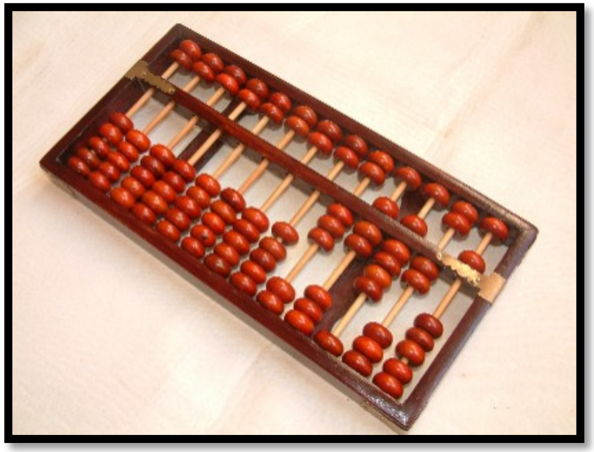
Foundations	• Data representation, logic, Boolean algebra.
Building blocks	• Transistors, transistor based logic, **simple devices**, storage.
Modules	• Memory, simple controllers, FSMs, processors and execution.
Programming	• Machine code, assembly, high-level languages, compilers.
Wrap-up	• Operating systems, energy aware computing.

Today, we learn to add!

- And also…
  - Subtract
  - Select 1 signal from many
  - Distribute 1 signal to many
- The circuits shown hereafter will be drawn in Logisim.
  - You can download it from:

  http://sourceforge.net/projects/circuit/

  - Install Logisim on your own computer and practice.



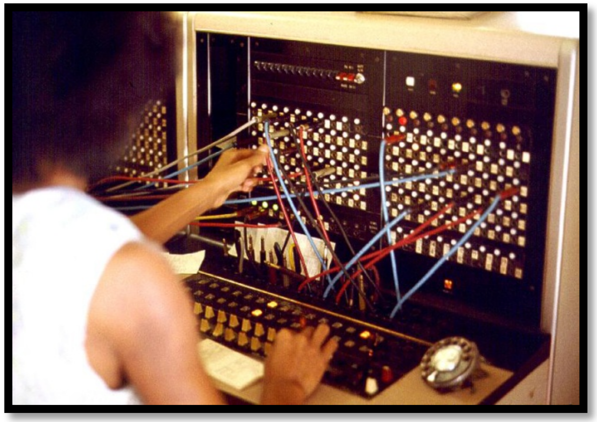


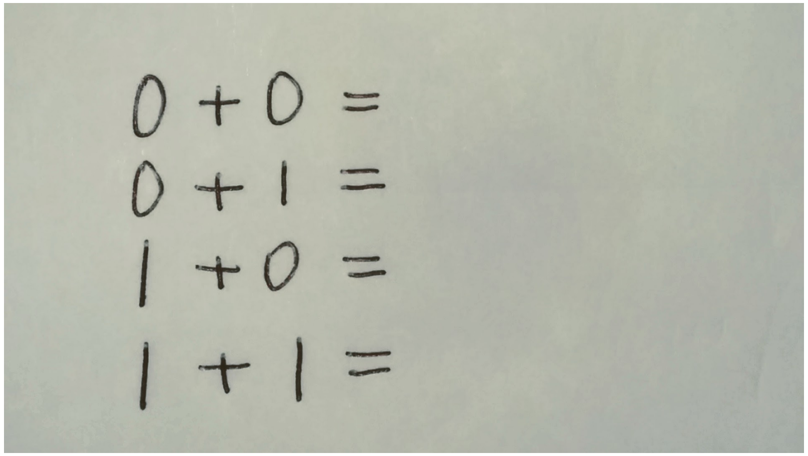
Photo by Joseph A. Carr, 1975

COMSM1302 NAND board kit

- Your task in the NAND labs will involve building some of the circuits that will be introduced today.
- **You should always design with Logisim BEFORE you start building with the NAND board kit.**

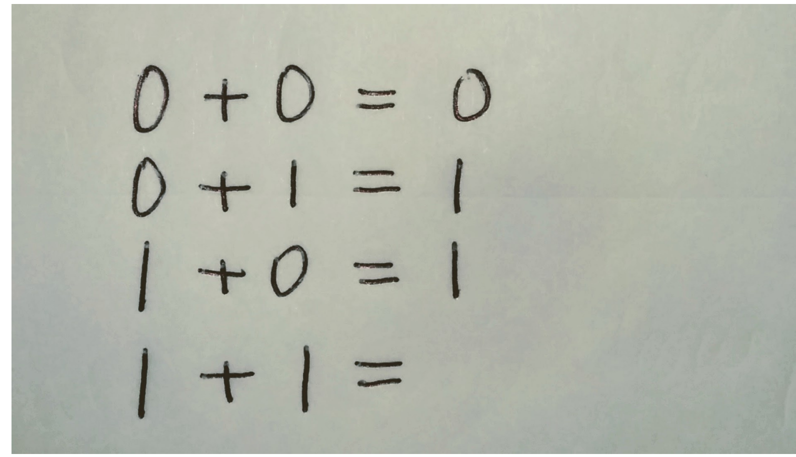
The simplest of binary addition

- How to add two, single-digit binary numbers?
  - Each digit is either 0 or 1
  - What are the possible results?



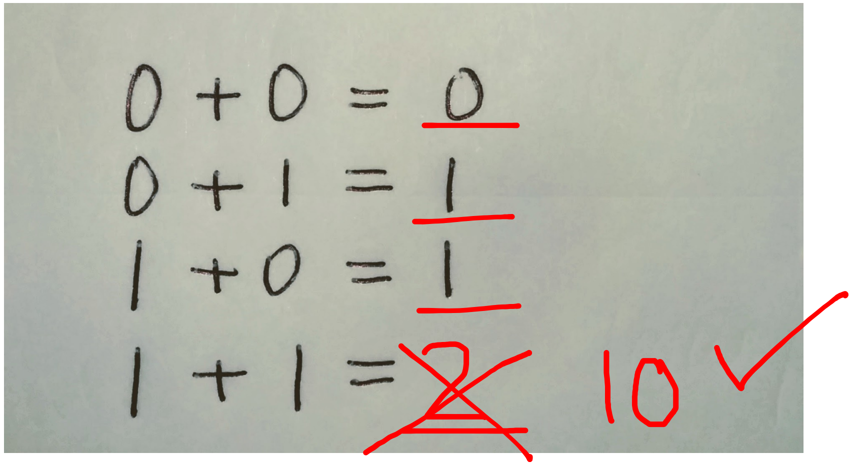
The simplest of binary addition

- How to add two, single-digit binary numbers?
  - Each digit is either 0 or 1
  - What are the possible results?



The simplest of binary addition

- How to add two, single-digit binary numbers?
  - Each digit is either 0 or 1
  - What are the possible results?



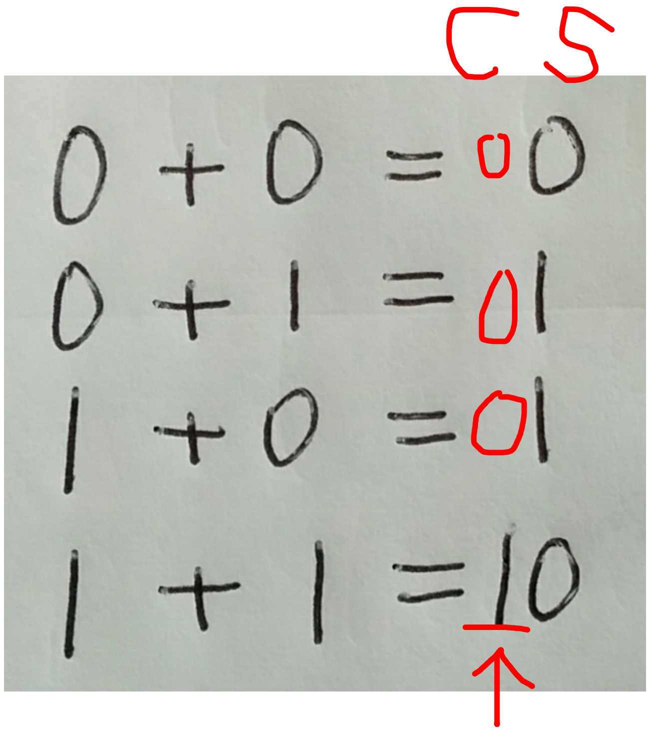
The simplest of binary addition

- How to add two, single-digit binary numbers?
  - Each digit is either 0 or 1
  - There are three possible results
    - 0, 1, 2
    - 0b00, 0b01, 0b10
  - Two inputs
    - A, B
  - Output
    - Sum (S)

A		B		S
0	+	0	=	0
0	+	1	=	1
1	+	0	=	1
1	+	1	=	10

The simplest of binary addition

- How to add two, single-digit binary numbers?
  - Each digit is either 0 or 1
  - There are three possible results
    - 0, 1, 2
    - 0b00, 0b01, 0b10
  - Two inputs
    - A, B
  - Two outputs
    - Sum (S)
    - Carry (C)



The simplest of binary addition

- How to add two, single-digit binary numbers?
  - Each digit is either 0 or 1
  - There are three possible results
    - 0, 1, 2
    - 0b00, 0b01, 0b10
  - Two inputs
    - A, B
  - Two outputs
    - Carry (C)
    - Sum (S)

A	B	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

The simplest of binary addition

- How to add two, single-digit binary numbers?
  - Each digit is either 0 or 1
  - There are three possible results
    - 0, 1, 2
    - 0b00, 0b01, 0b10
  - Two inputs
    - A, B
  - Two outputs
    - Carry (C)
    - Sum (S)

C S

A	B	2^1 2	2^0 1
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

With some Boolean algebra

- Remember our truth tables for the connectives in Boolean algebra, e.g. AND, OR, XOR, etc?
- Which operations can be used to help us generate S and C?

A	B	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

With some Boolean algebra

- Remember our truth tables for the connectives in Boolean algebra, e.g. AND, OR, XOR, etc?
- Which operations can be used to help us generate S and C?

A	B	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

XOR

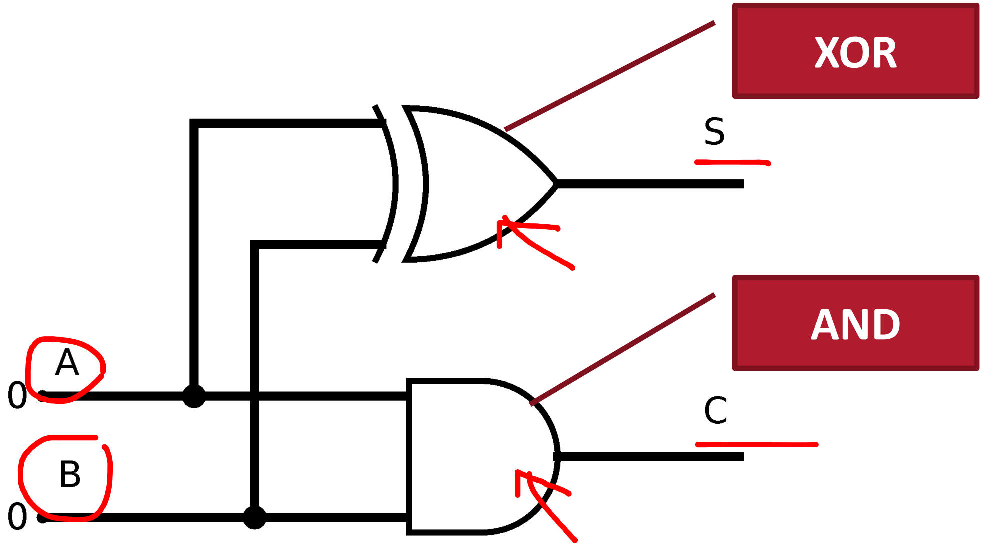
A	B	¬A	A ∧ B	A ∨ B	A ⊕ B	A → B	A≡B
0	0	1	0	0	0	1	1
0	1	1	0	1	1	1	0
1	0	0	0	1	1	0	0
1	1	0	1	1	0	1	1

The half-adder

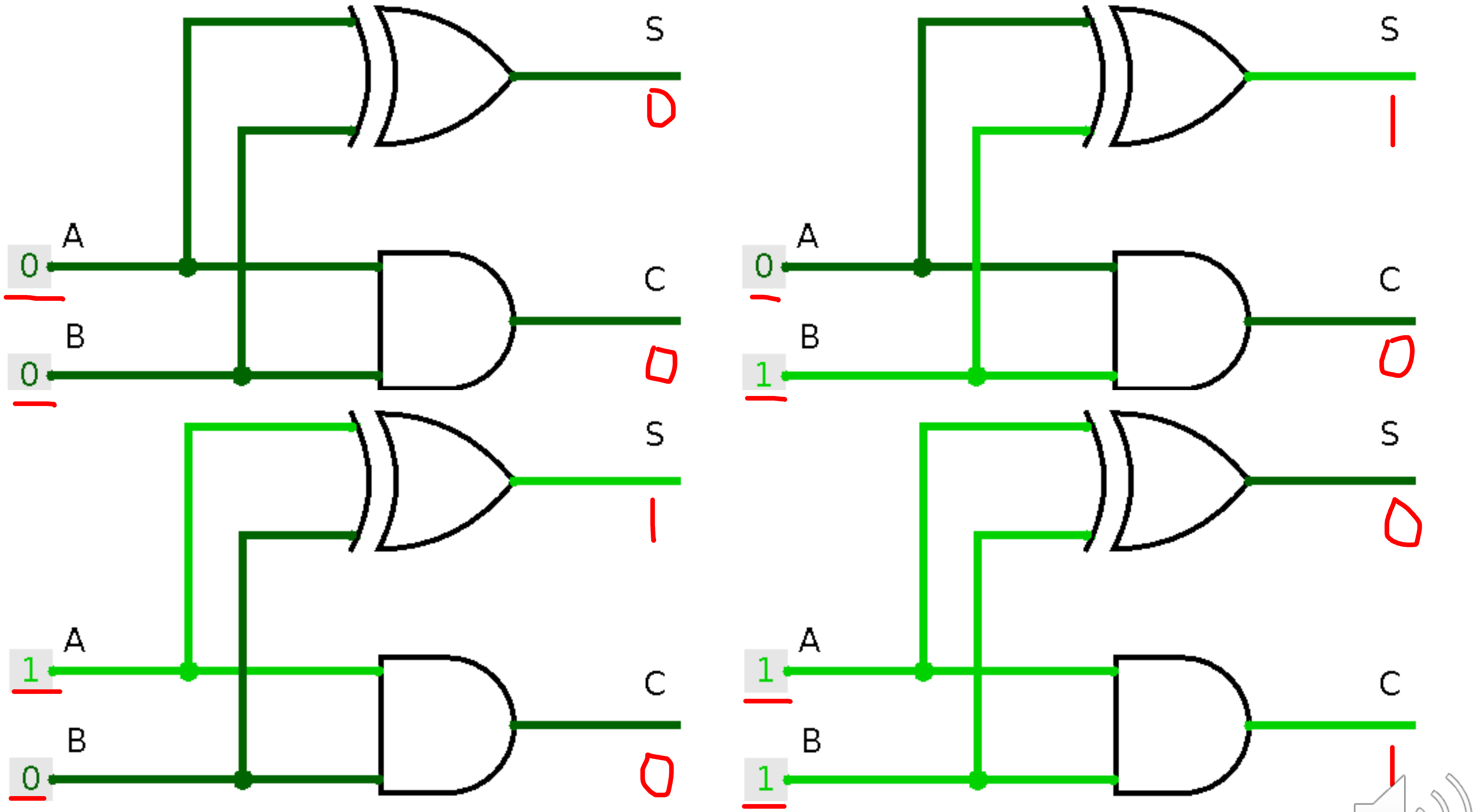
- $S = A \oplus B$
- $C = A \wedge B$

add two bits together, no carry in

- Now let's build it with logic gates.



The half-adder in action



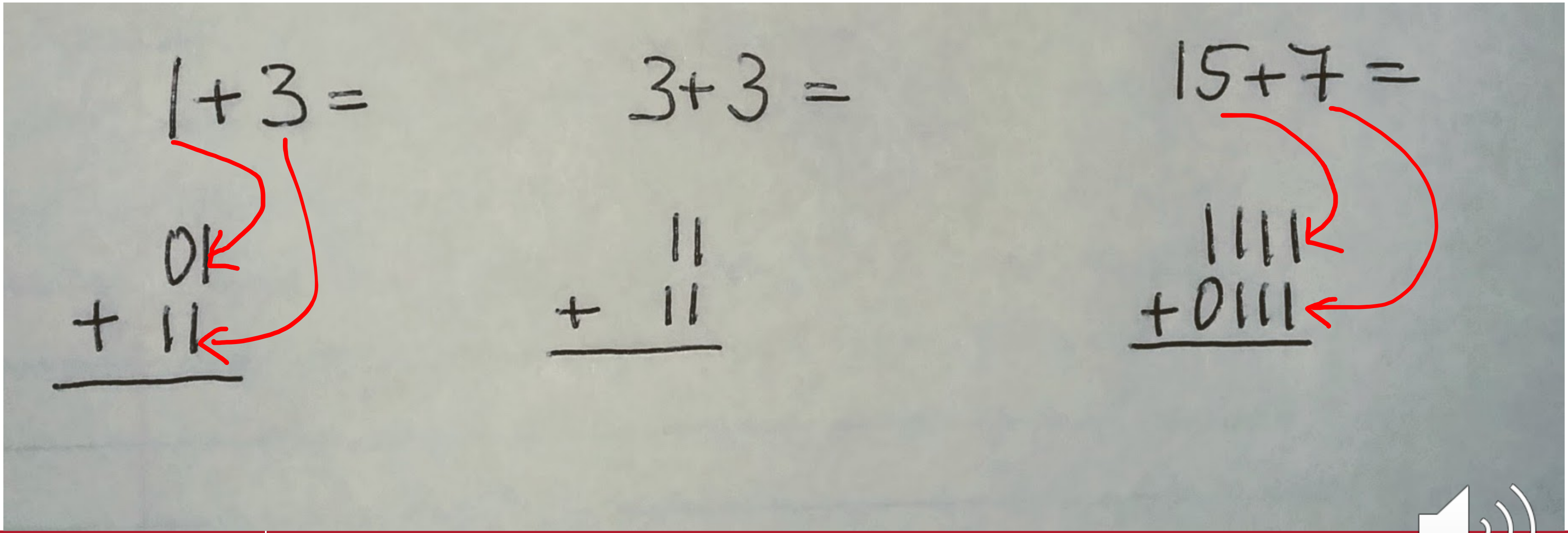
Now what?

- We can add two bits together.
  - By generating a sum and a carry bit.
- How do we add multiple bits together?

$1+3=$ $3+3=$ $15+7=$

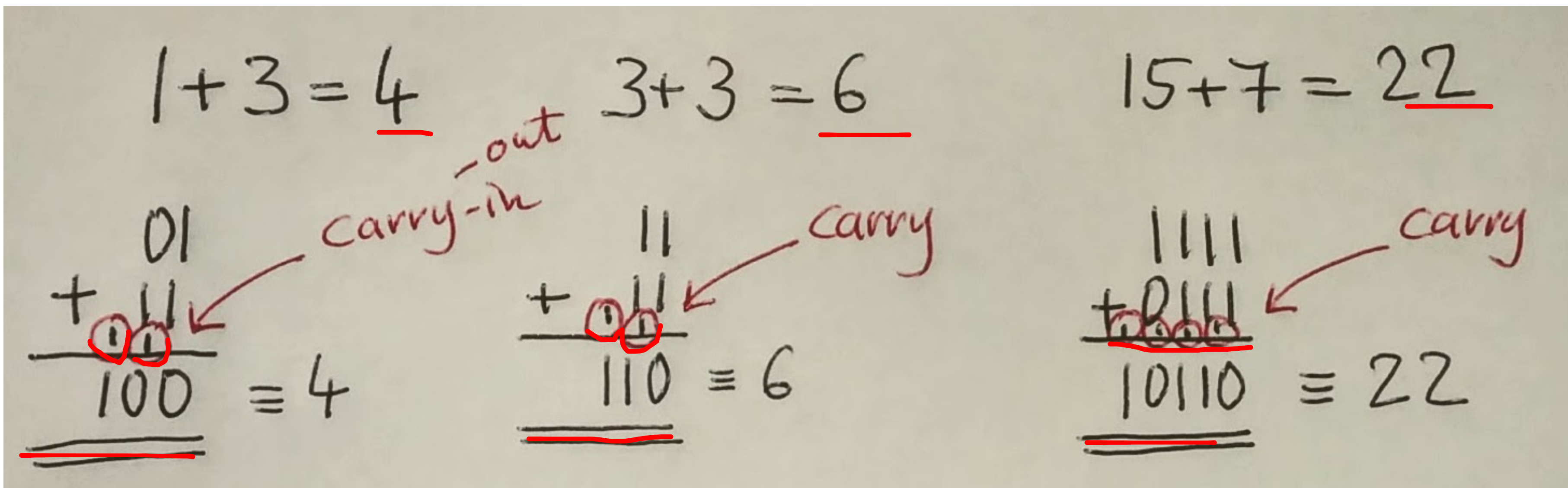
Now what?

- We can add two bits together.
  - By generating a sum and a carry bit.
- How do we add multiple bits together?



Now what?

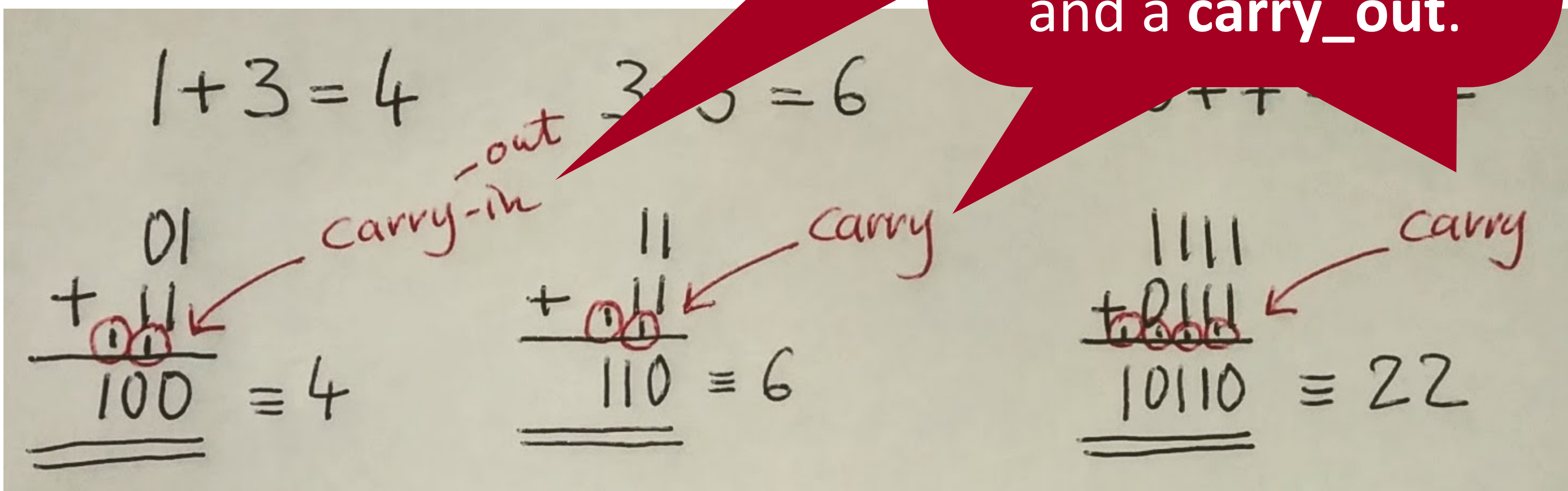
- We can add two bits together.
  - By generating a sum and a carry bit.
- How do we add multiple bits together?



Now what?

- We can add two bits togeth
  - By generating a sum and a
- How do we add multip

For each addition (except for the first) we need to account for two input bits <u>and</u> a **carry_in**, and we produce a **sum** and a **carry_out**.



The full-adder

C_in	A	B	C_out	S
0	0	0	0	0
0	0	1	0	1
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
1	1	1	1	1

From the half-adder:

- $S = A \oplus B$
- $C\_out = A \land B$

(Note that the above covers only the top-half of this table!)

The full-adder

C_in	A	B	C_out	S
0	0	0	0	0
0	0	1	0	1
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
1	1	1	1	1

The full-adder

For multiple arguments, XOR is defined to be true iff an odd number of its arguments is true, and false otherwise.

C_in	A	B	C_out	S
0	0	0	0	0
0	0	1	0	1
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
1	1	1	1	1

- $S = A \oplus B \oplus C\_in$
- $C\_out = (A \wedge B) \vee (C\_in \wedge (A \vee B))$
- Also valid:

$C\_out = (A \wedge B) \vee (C\_in \wedge (A \oplus B))$

  – **Why?**

The full-adder

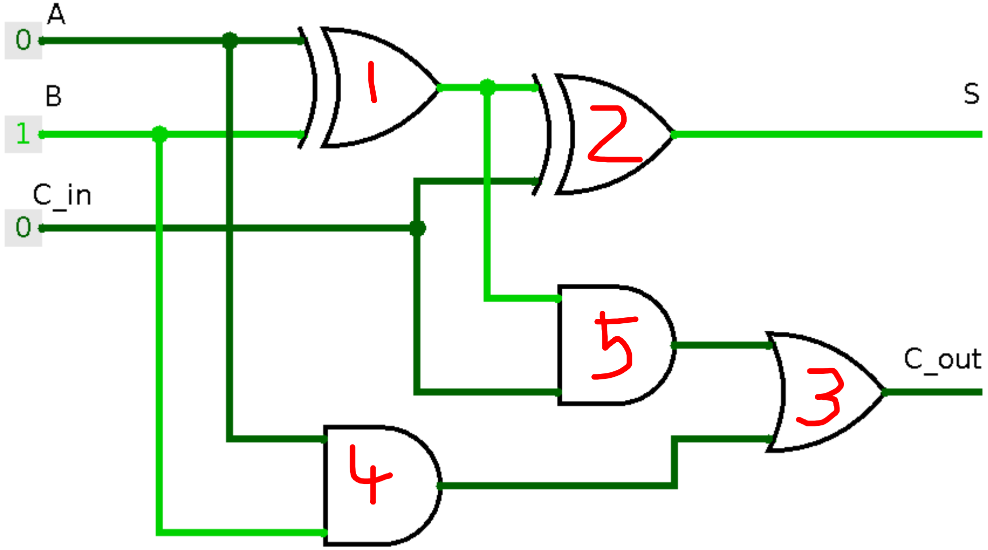
C_in	A	B	C_out	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

- $S = A \oplus B \oplus C\_in$
- $C\_out =$
  $(A \land B) \lor (C\_in \land (A \lor B))$

- Also valid:
  $C\_out = (A \land B) \lor (C\_in \land (A \oplus B))$
  - **Why?**

The full-adder

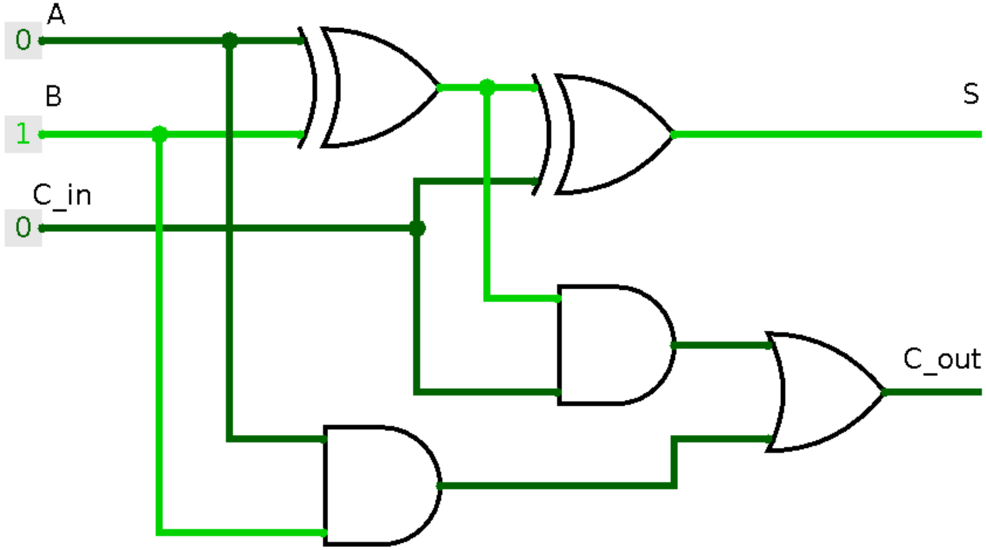
- 8 different combinations of input
- Try them yourself!



- S = A ⊕ B ⊕ C_in
- C_out =(A ∧ B) ∨ (C_in ∧ (A ⊕ B))

The full-adder

- 8 different combinations of input
- Try them yourself!



- To build this with the NAND kits you need to:
  - use Boolean algebra to obtain a design that is based purely on NAND gates,
  - implement this in Logisim to gain confidence your design is correct, then
  - *(and only then)* transfer to NAND boards and test.

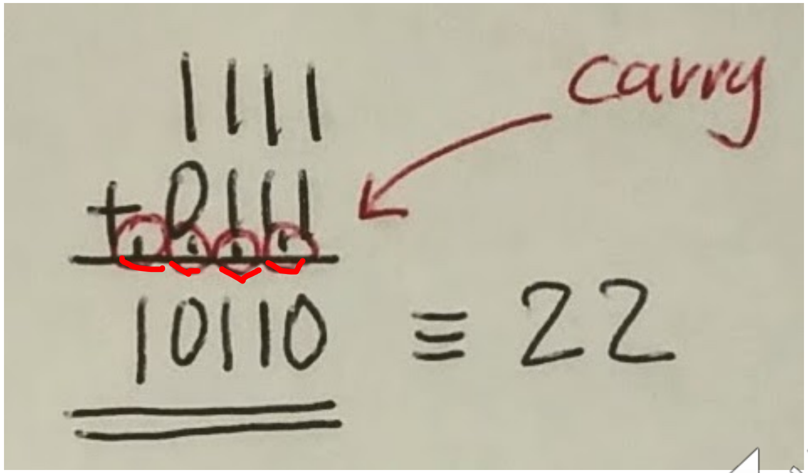
Now what?

- We can add two bits together.
  - By generating a sum and a carry bit.
- We can add three bits together
  - By accommodating a carry-in as well as our regular two inputs.

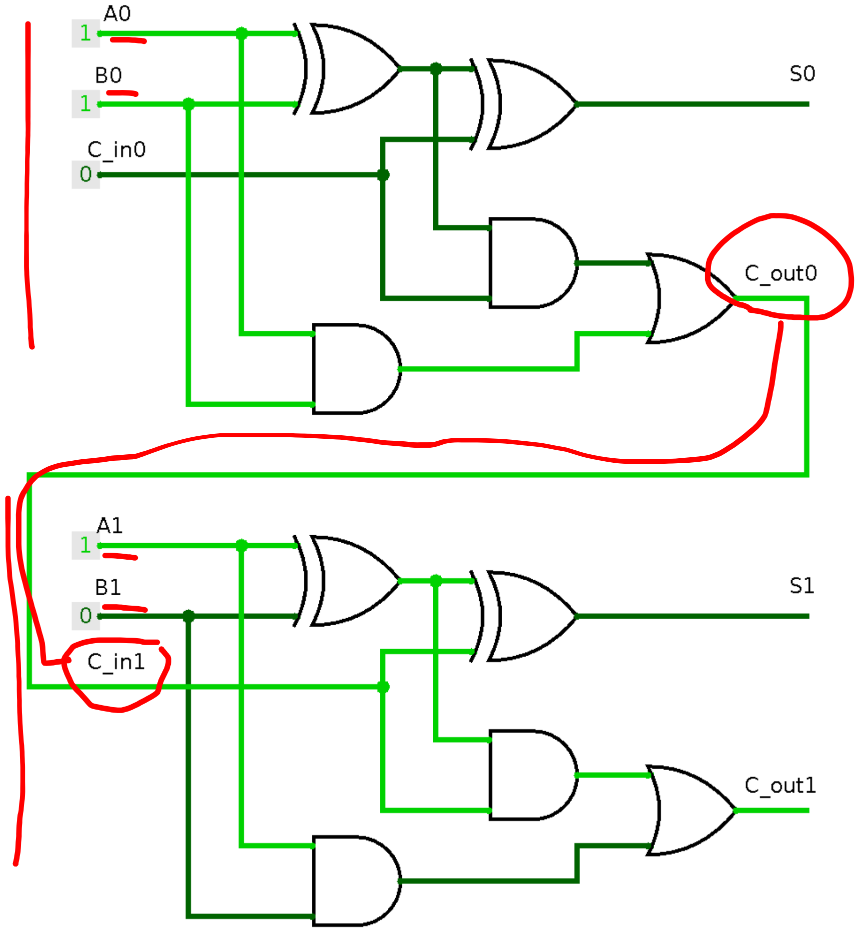
- How do we add multiple bits together?

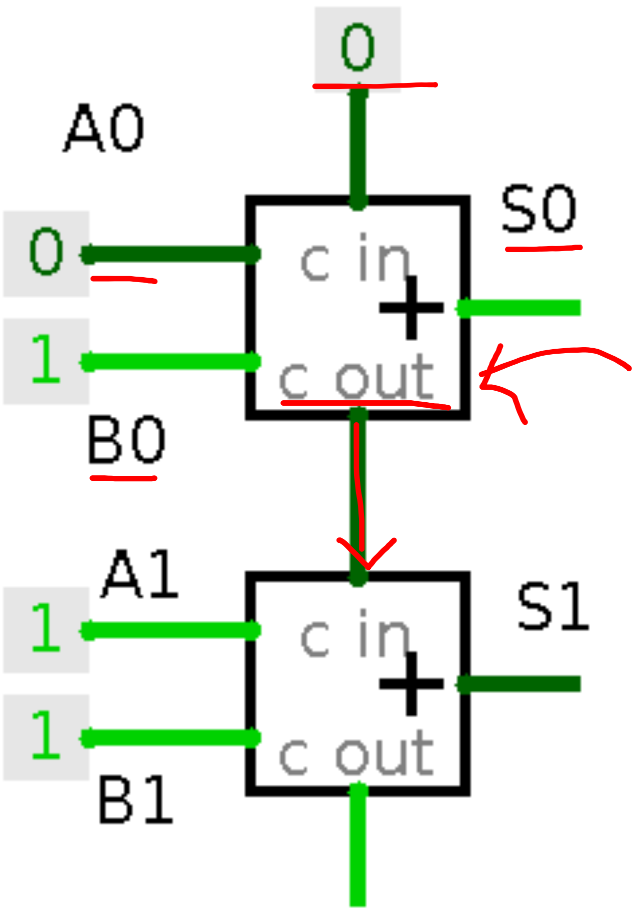
Now what?

- We can add two bits together.
  - By generating a sum and a carry bit.
- We can add three bits together
  - By accommodating a carry-in as well as our regular two inputs.
- How do we add multiple bits together?

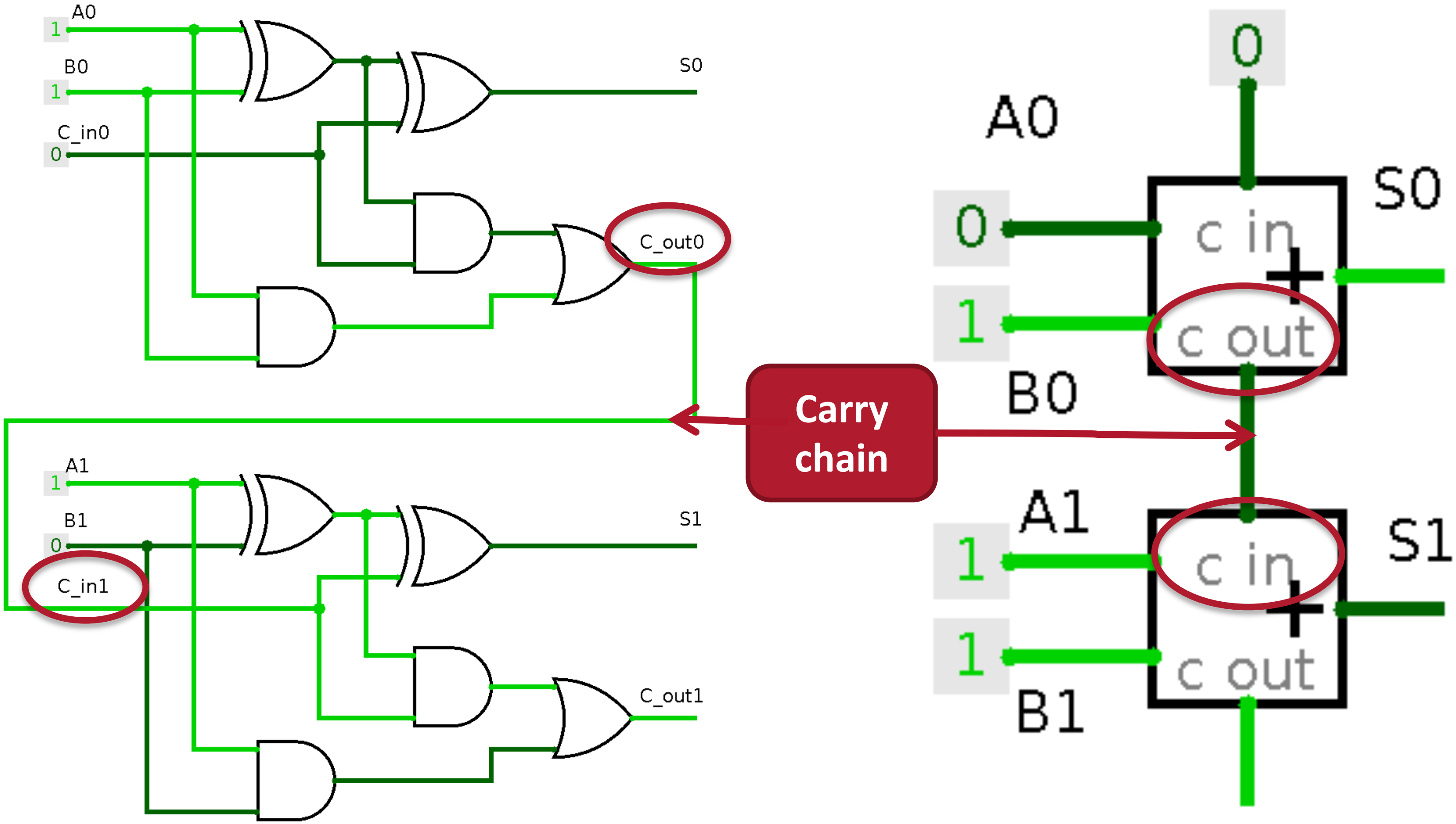


Chaining full-adders

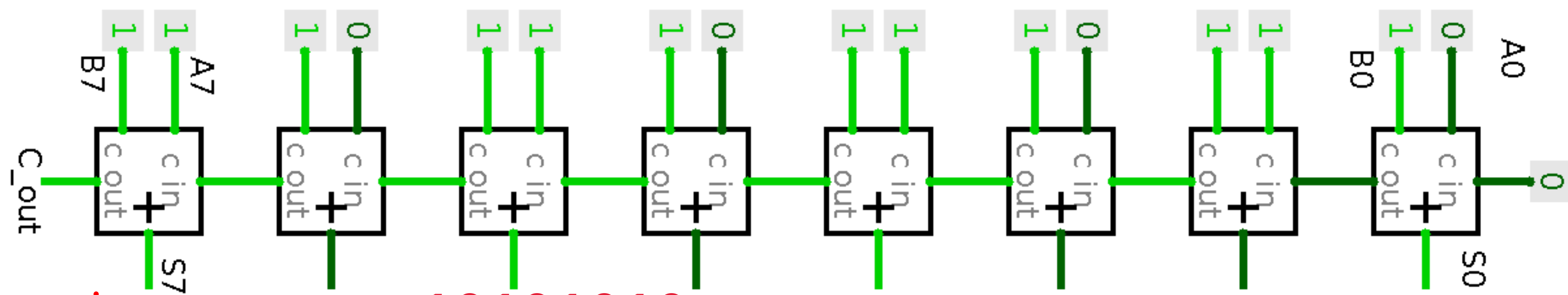




Chaining full-adders



8-bit adder, ripple-carry adder



10101010
+11111111

- Named **ripple-carry** because a carry signal generated at the LSB (Least Significant Bit… bit 0) of the device can affect the result on any/all more significant bits.
- Does this have any implications?

Building blocks

- To recap what we've done:
  – Used Boolean algebra to identify our **building blocks**.
  – Built a unit capable of **adding two bits**.
    - **Half-adder**
  – Extended it to handle **carry-in**.
    - **Full-adder**
  – Chained them together to make an adder of **arbitrary size**.
    - **Ripple-carry adder**
- Now we can add anything!
  – Modern processors typically have adders between 8 and 64 bits.
  – **Why?**

Subtraction

- Subtraction is easy if we think of it as **adding one number to a negative number**.
- So let's represent this subtraction:

  1 – 2 = -1
- As:

  1 + -2 = -1

- How to negate a number?

Subtraction

- Subtraction is easy if we think of it as **adding one number to a negative number**.
- So let's represent this subtraction:

  1 – 2 = -1

- As:

  1 + -2 = -1

- How to negate a number?
  – We use 2s complement!

Reminder: 2s complement

$$Y = -x_{N-1} \cdot 2^{N-1} + \sum_{i=0}^{N-2} x_i \cdot 2^i$$

-128	64	32	16	8	4	2	1	Base 10
0	0	0	0	1	0	1	0	10
0	0	0	0	0	0	0	1	1
1	1	1	1	1	1	1	1	-1
1	1	1	1	1	1	1	0	-2
1	0	0	0	1	0	1	0	-118

Calculating the 2s complement

To calculate the 2's complement of an integer:

1. invert the binary equivalent of the number by changing all of the ones to zeroes and all of the zeroes to ones (also called 1's complement), then
2. add one.

- How do we represent -6?

Calculating the 2s complement

To calculate the 2's complement of an integer:

1. invert the binary equivalent of the number by changing all of the ones to zeroes and all of the zeroes to ones (also called 1's complement), then
2. add one.

- How do we represent -6?

$110 \equiv 6$

Calculating the 2s complement

To calculate the 2's complement of an integer:

1. invert the binary equivalent of the number by changing all of the ones to zeroes and all of the zeroes to ones (also called 1's complement), then
2. add one.

0110 ≡ 6

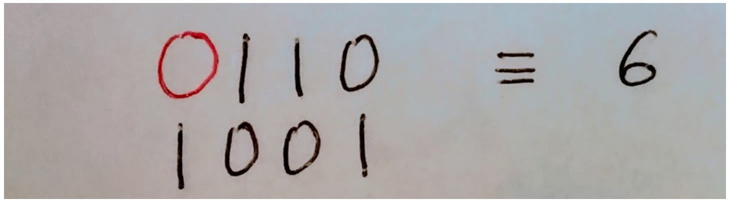
- How do we represent -6?

Calculating the 2s complement

To calculate the 2's complement of an integer:

1. invert the binary equivalent of the number by changing all of the ones to zeroes and all of the zeroes to ones (also called 1's complement), then
2. add one.

- How do we represent -6?

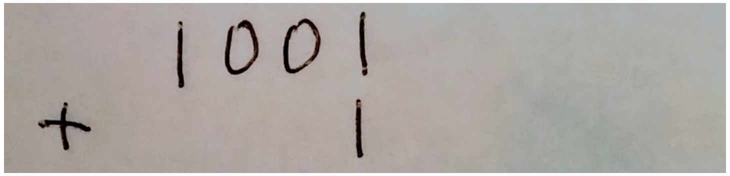


Calculating the 2s complement

To calculate the 2's complement of an integer:

1. invert the binary equivalent of the number by changing all of the ones to zeroes and all of the zeroes to ones (also called 1's complement), then
2. add one.

- How do we represent -6?

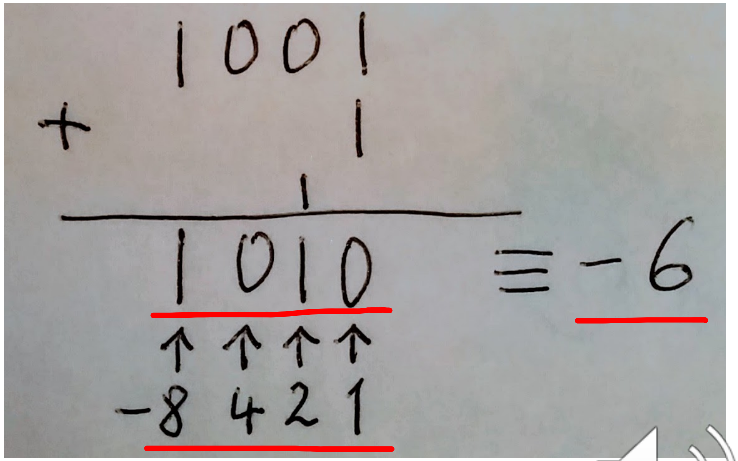


Calculating the 2s complement

To calculate the 2's complement of an integer:

1. invert the binary equivalent of the number by changing all of the ones to zeroes and all of the zeroes to ones (also called 1's complement), then
2. add one.

- How do we represent -6?



Subtraction with 2s complement

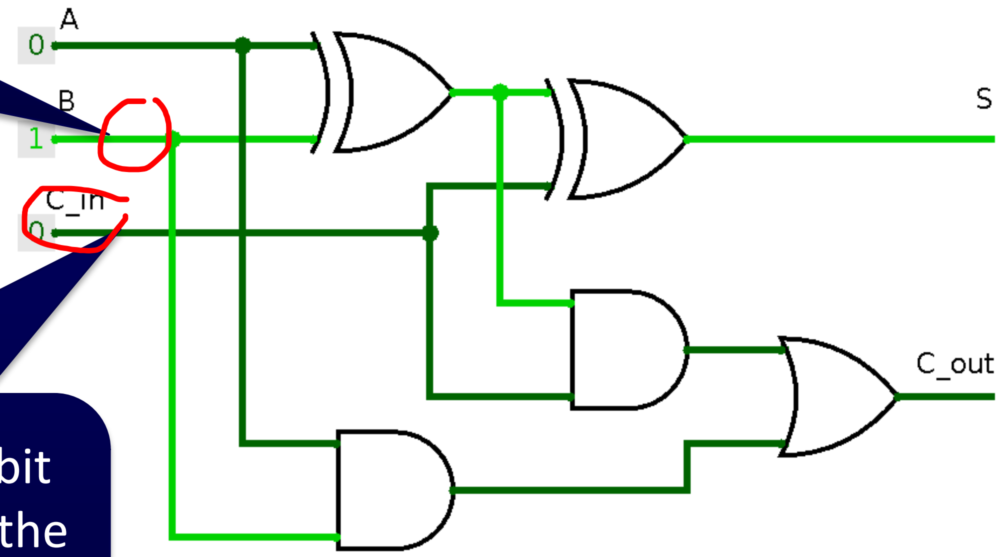
- A – B = A + (-B) = A + (Not(B) + 1)
- We already have all the building blocks we need to implement this!
  - NOT gates to flip bits
  - An unused C_in at the beginning of our adder's carry chain to provide the extra 1.

Subtraction with 2s complement

- A – B = A + (-B) = A + (Not(B) + 1)
- We already have all the building blocks we need to implement this!
  - NOT gates to flip bits
  - An unused C_in at the beginning of our adder's carry chain to provide the extra 1.

From full-adder to subtractor

Add a NOT gate here to obtain the 1s complement of B, i.e. NOT(B)



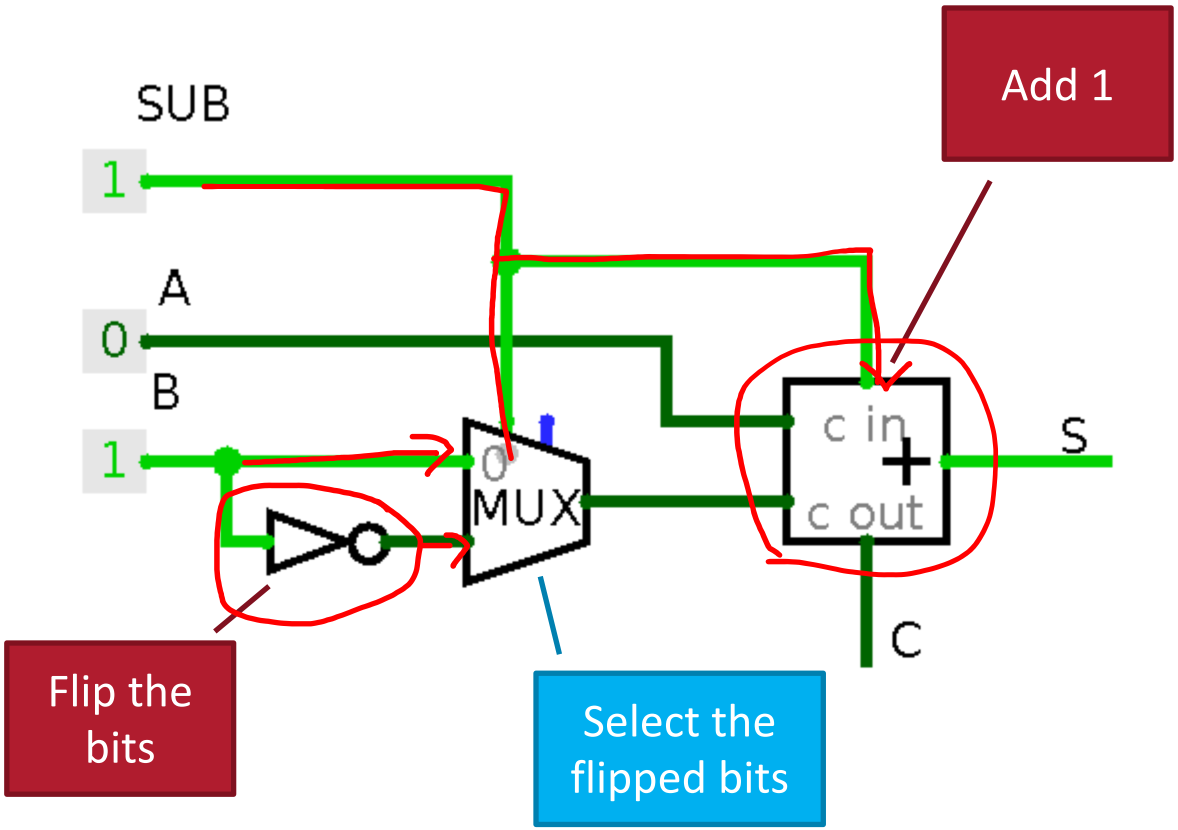
Set the C_in (at the lowest bit position) to 1; this provides the “extra” 1 we need to obtain the 2s complement of B.

The adder-subtractor?

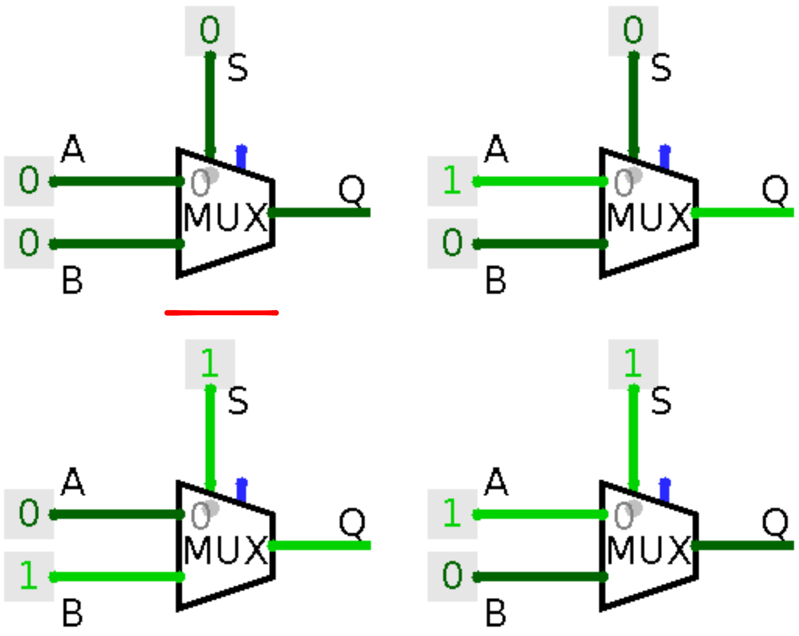
- We can build an adder **or** a subtractor.
- They are **very similar**.
- Can we build one unit that does **both**?

- Almost…

Adder-subtractor



Selecting a signal



S *(Select)*	A	B	Q
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Selecting a signal

- Q = (A ∧ ¬S) ∨ (B ∧ S)
- Consider S = 0
- Consider S = 1

S	A	B	Q
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Selecting a signal

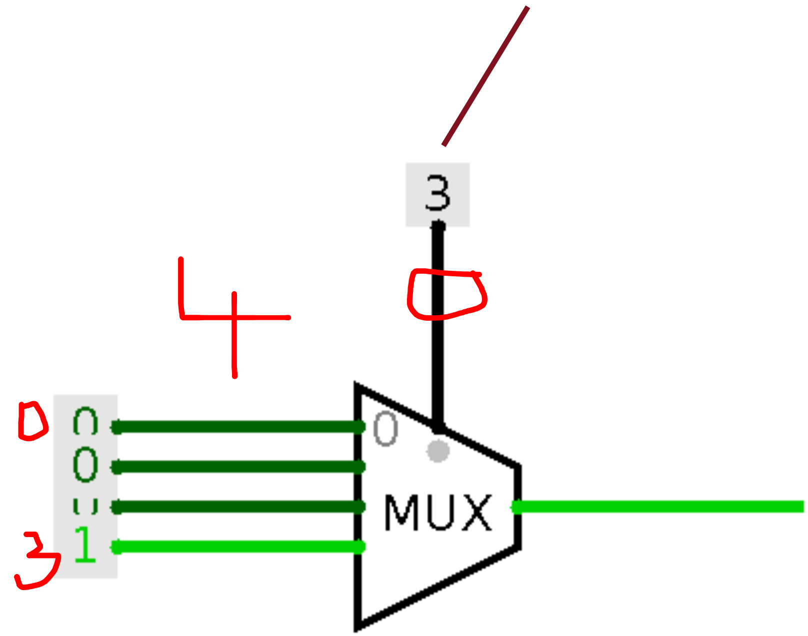
- $Q = (A \land \neg \underline{S}) \lor (B \land \underline{S})$
- Consider $\underline{S = 0}$
  - $Q = (A \land \underline{1}) \lor (B \land \underline{0})$
  - $Q = A \lor 0$
  - $\underline{Q = A}$
- Consider $\underline{S = 1}$
  - $\underline{Q} = (A \land 0) \lor (B \land 1)$
  - $Q = 0 \lor B$
  - $\underline{Q = B}$

S	A	B	Q
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

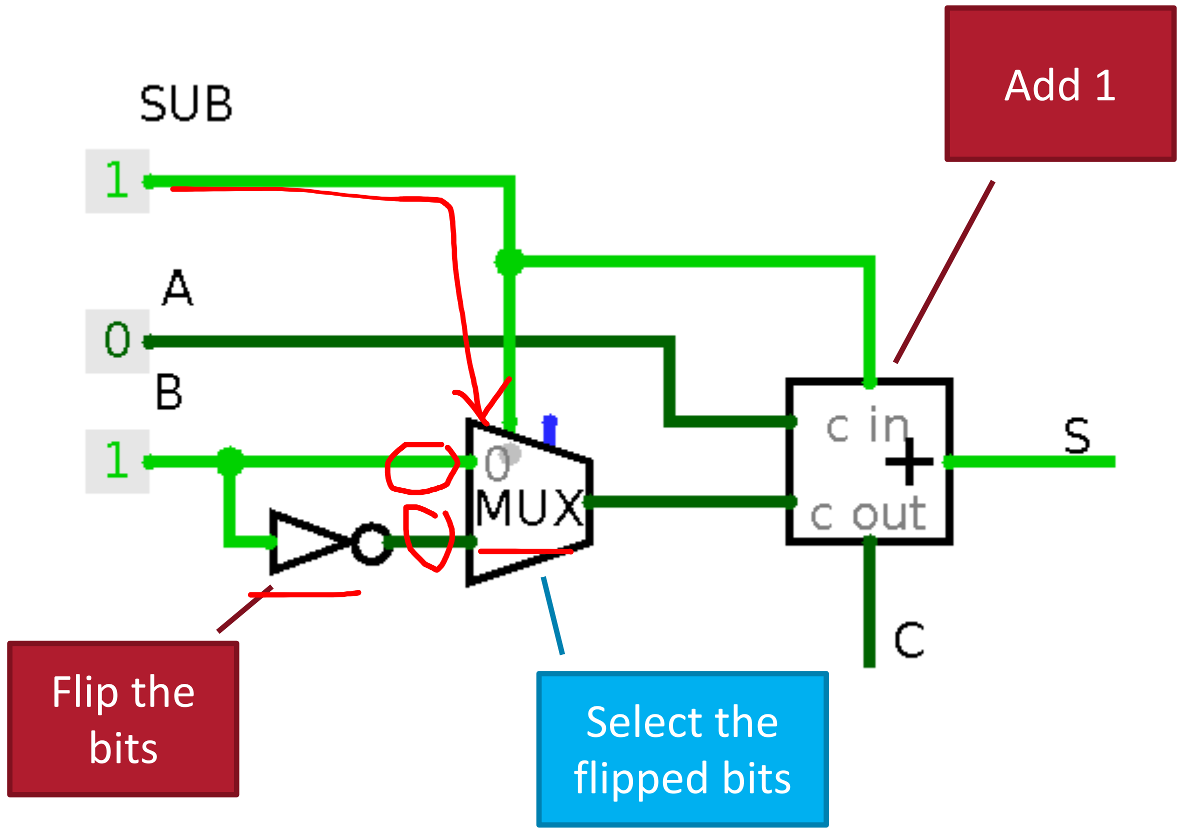
The multiplexer

- S selects which input to propagate to the output.
- 2-to-1 multiplexer
  - 1 select bit
- **N**-to-1 multiplexers are also possible
  - $\log_2(N)$ select bits needed
    - Why?

2-bit signal (imagine a bundle of two wires)

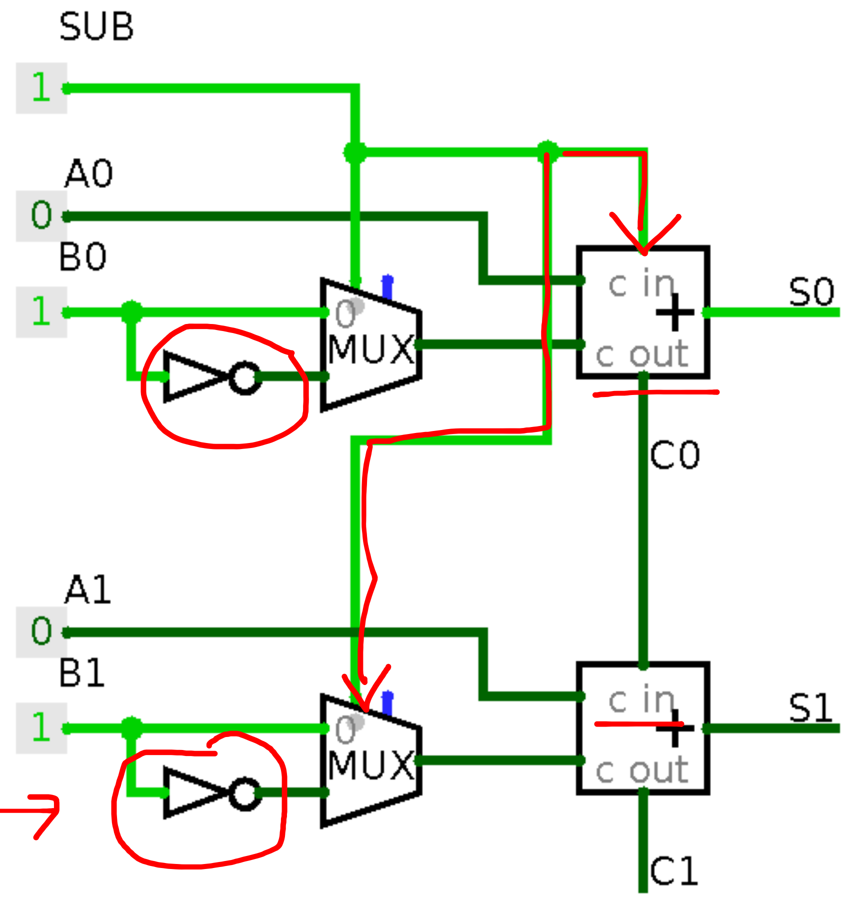


Adder-subtractor



Ripple-carry adder-subtractor

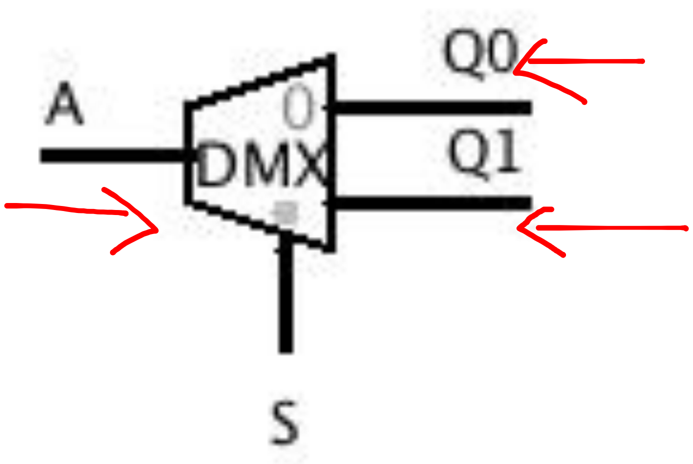
Only a one is needed at the beginning



Multi bits: like 8-bit 64-bit

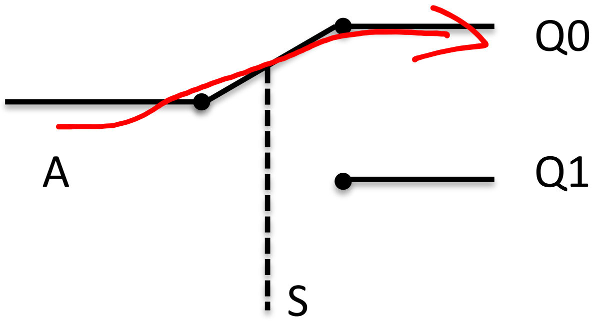
Attention: If we want to add a + b + c, calculate the first item, then add the result and c

Demultiplexer



- 1-to-2 demultiplexer
  - Choose **which of 2 wires** to **propagate** the input signal onto.
  - Q0 = A ∧ ¬S
  - Q1 = A ∧ S
- Instead of choosing which signal to select, we choose **where to send** a signal to, i.e. which output carries the input.

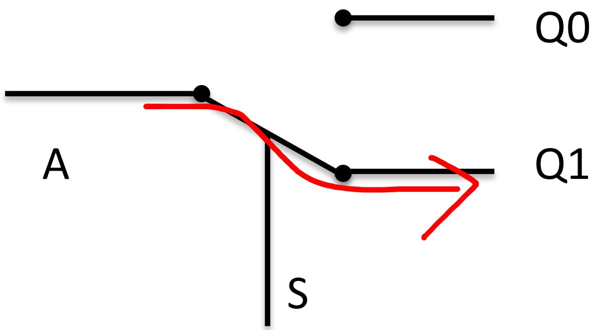
Demultiplexer



The demultiplexer acts like a switch.

- 1-to-2 demultiplexer
  - Choose **which of 2 wires** to **propagate** the input signal onto.
  - **$Q0 = A \wedge \neg S$**
  - $Q1 = A \wedge S$
- Instead of choosing which signal to select, we choose **where to send** a signal to, i.e. which output carries the input.

Demultiplexer



The demultiplexer acts like a switch.

- 1-to-2 demultiplexer
  - Choose **which of 2 wires** to **propagate** the input signal onto.
  - Q0 = A ∧ ¬S
  - **Q1 = A ∧ S**
- Instead of choosing which signal to select, we choose **where to send** a signal to, i.e. which output carries the input.

Demultiplexer

S	A	Q0	Q1
0	**0**	**0**	0
0	**1**	**1**	0
1	0	0	0
1	1	0	1

- 1-to-2 demultiplexer
  - Choose **which of 2 wires** to **propagate** the input signal onto.
  - **Q0 = A ∧ ¬S**
  - Q1 = A ∧ S
- Instead of choosing which signal to select, we choose **where to send** a signal to, i.e. which output carries the input.

Demultiplexer

S	A	Q0	Q1
0	0	0	0
0	1	1	0
1	**0**	0	**0**
1	**1**	0	**1**

- 1-to-2 demultiplexer
  - Choose **which of 2 wires** to **propagate** the input signal onto.
  - $Q0 = A \land \neg S$
  - **$Q1 = A \land S$**
- Instead of choosing which signal to select, we choose **where to send** a signal to, i.e. which output carries the input.

Summary

- Used Boolean algebra to build **four simple devices**:
  - Demux, Mux, Adder, Subtractor.
- Combined Mux, NOT gate and adder to build **adder-subtractor**.
- We can do **basic arithmetic** with a **bunch of NAND gates**!

Imagine if we could **store** the results of that arithmetic, somehow…

In this lecture

Foundations	• Data representation, logic, Boolean algebra.
Building blocks	• Transistors, transistor based logic, **simple devices**, storage.
Modules	• Memory, simple controllers, FSMs, processors and execution.
Programming	• Machine code, assembly, high-level languages, compilers.
Wrap-up	• Operating systems, energy aware computing.

In the next lecture

Foundations	• Data representation, logic, Boolean algebra.
Building blocks	• Transistors, transistor based logic, simple devices, **storage.**
Modules	• Memory, simple controllers, FSMs, processors and execution.
Programming	• Machine code, assembly, high-level languages, compilers.
Wrap-up	• Operating systems, energy aware computing.