CS 261
Fall 2019

Mike Lam, Professor

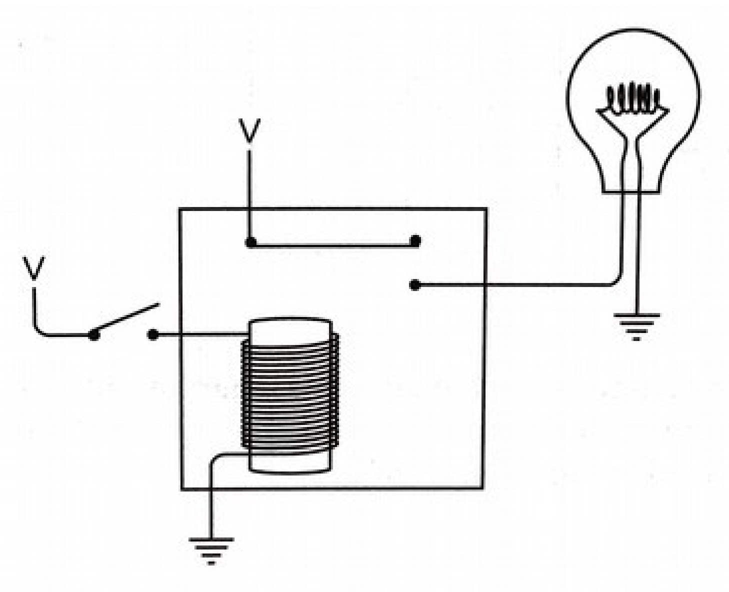
Combinational Circuits

The final frontier

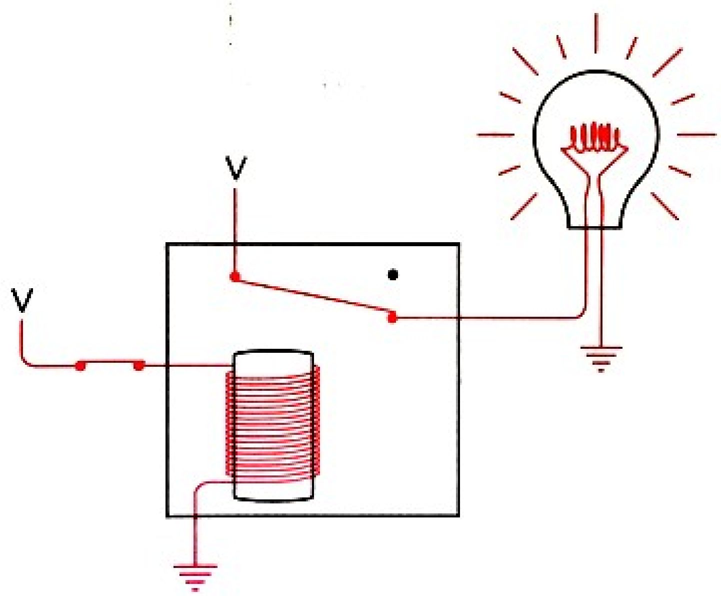
- Java programs running on Java VM
- C programs compiled on Linux
- Assembly / machine code on CPU + memory
- ???
- Switches and electric signals

Aside: Relays

- From “Code” recommended reading:



Relay (off)



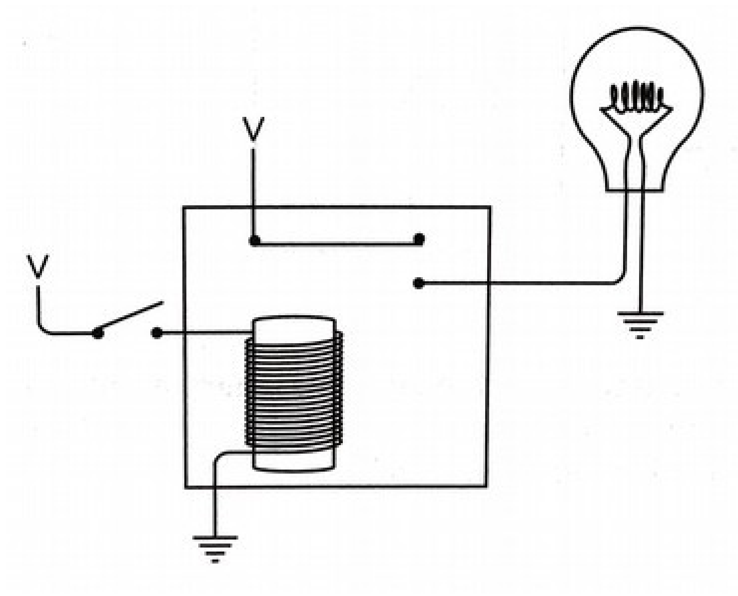
Relay (on)

Light is on when switch is on

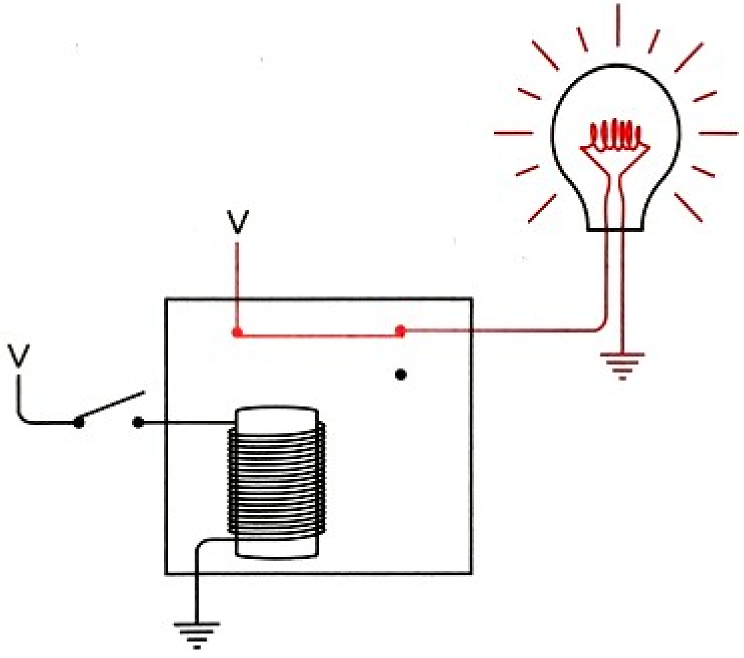
Question: what happens if we connect the light bulb to the other contact?

Aside: Relays

- From “Code” recommended reading:



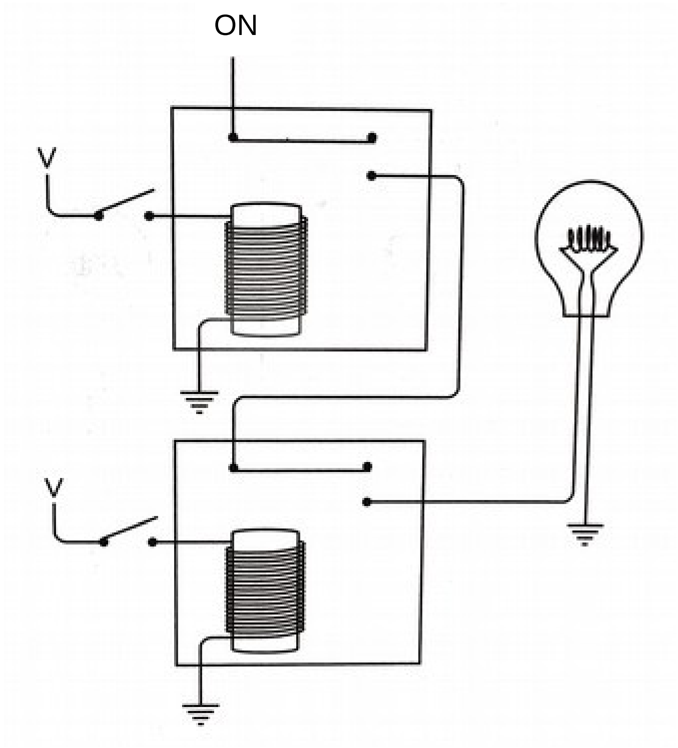
Regular relay



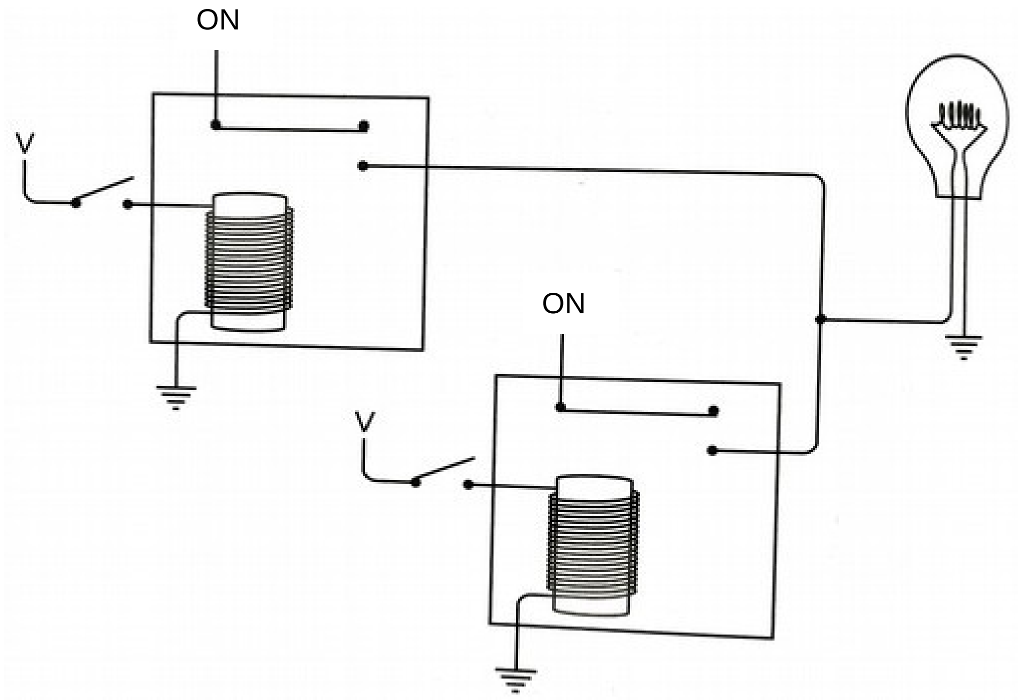
Inverted relay (NOT)

Aside: Relays

- From “Code” recommended reading:



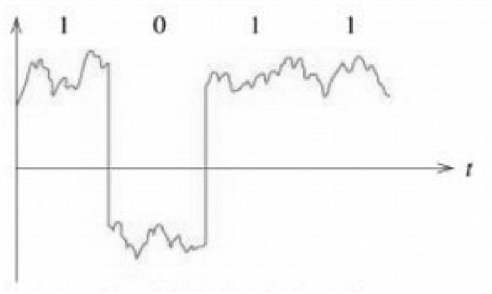
Relays in series (AND)

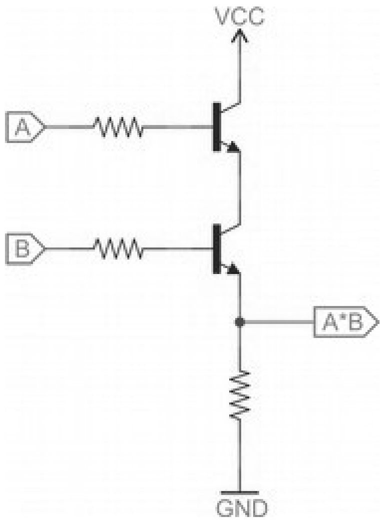


Relays in parallel (OR)

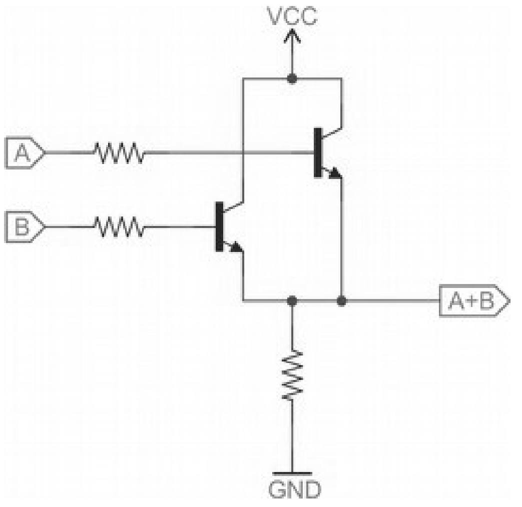
Digital hardware

- Digital signals are transmitted via electric signals by varying voltages
  - 1.0 V (high) = binary 1
  - 0.0 V (low) = binary 0
  - Use a threshold to distinguish





AND

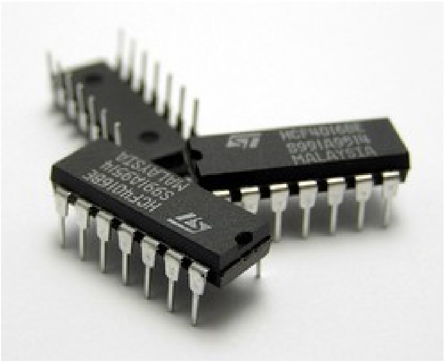


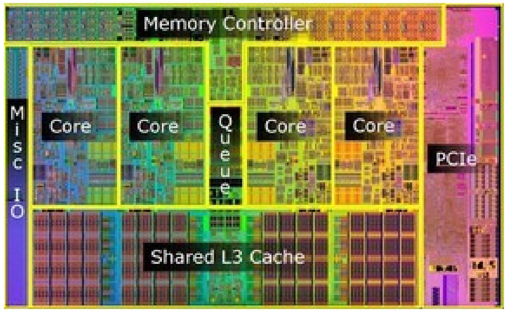
OR

Images from https://learn.sparkfun.com/tutorials/transistors

Transistors

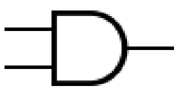
- Transistors are the fundamental hardware component of computing
  - Similar to relays; replaced vacuum tubes
    - Smaller, more reliable, and use less energy
    - Primary functions: switching and amplification
  - Mostly silicon-based semiconductors now
    - Metal–Oxide–Semiconductor Field-Effect Transistor (MOSFET)
    - n-channel ("on" when $V_{gate} = 1V$) vs. p-channel ("off" when $V_{gate} = 1V$)
    - Mass-produced on integrated circuit chips
  - For convenience, we abstract their behavior using logic gates





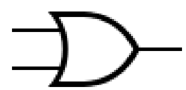
Logic gates

- Primary gates:



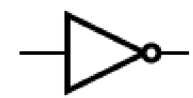
&	0	1
0	0	0
1	0	1

AND



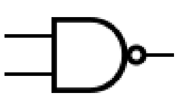
|	0	1
0	0	1
1	1	1

OR



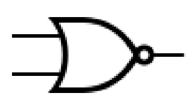
!	
0	1
1	0

NOT



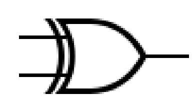
	0	1
0	1	1
1	1	0

NAND



	0	1
0	1	0
1	0	0

NOR



^	0	1
0	0	1
1	1	0

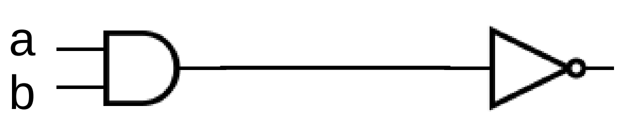
XOR

Basic circuits

- Circuits are formed by connecting gates together
  - Inputs and outputs
    - Link output of one gate to input of another
    - Some circuits have multiple inputs and/or outputs
  - Textbook uses Hardware Description Language (HDL)
  - Equivalent to boolean formulas or functions
    - f(g(x, y)) means “apply f to the result of applying g to x and y”
    - In a diagram: x,y → g → f (i.e., ordering is g first, then f)

Basic circuits

- f(g(x, y)) means "apply f to the result of applying g to x and y"
  - In a diagram: x,y → g → f (i.e., ordering is g first, then f)
- NAND example: (similarly for NOR)
  - Infix/boolean notation: **a NAND b = NOT(a AND b) = !(a & B)**
  - Function notation: **NAND(a, b) = NOT(AND(a, b))**



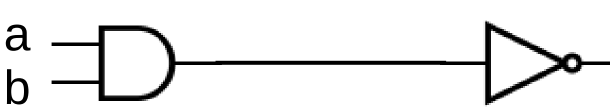
	0	1
0	?	?
1	?	?

&	0	1
0	0	0
1	0	1

a AND b

Basic circuits

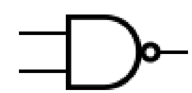
- f(g(x, y)) means “apply f to the result of applying g to x and y”
  - In a diagram: x,y → g → f (i.e., ordering is g first, then f)
- NAND example: (similarly for NOR)
  - Infix/boolean notation: **a NAND b = NOT(a AND b) = !(a & B)**
  - Function notation: **NAND(a, b) = NOT(AND(a, b))**



&	0	1
0	0	0
1	0	1

a AND b

	0	1
0	1	1
1	1	0

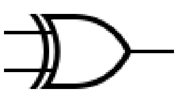


	0	1
0	1	1
1	1	0

a NAND b

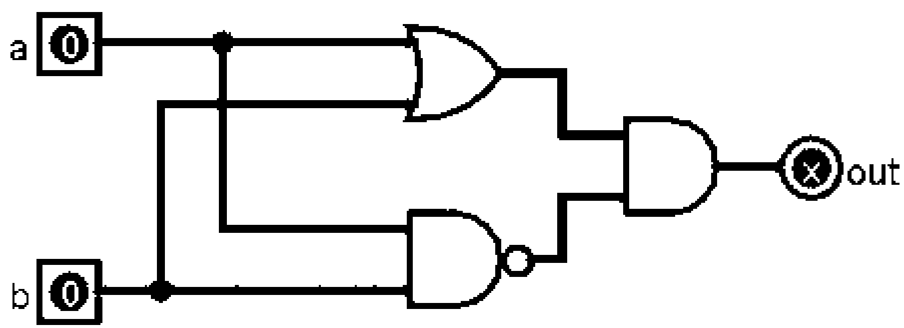
Basic circuits

- Circuits are equivalent if the truth tables are the same
  - **a XOR b = (a OR b) AND (a NAND b)**
  - **XOR(a, b) = AND(OR(a,b), NAND(a,b))**



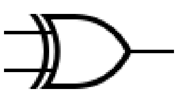
^	0	1
0	0	1
1	1	0

XOR



Basic circuits

- Circuits are equivalent if the truth tables are the same
  - **a XOR b = (a OR b) AND (a NAND b)**
  - **XOR(a, b) = AND(OR(a,b), NAND(a,b))**

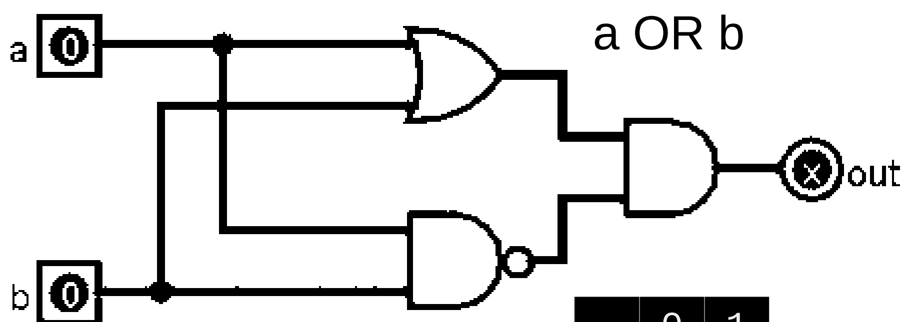


^	0	1
0	0	1
1	1	0

XOR

<table>
<tr><th>|</th><th>0</th><th>1</th></tr>
<tr><td>0</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td>1</td><td>1</td></tr>
</table>

a OR b

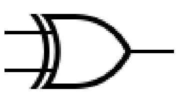


	0	1
0	1	1
1	1	0

a NAND b

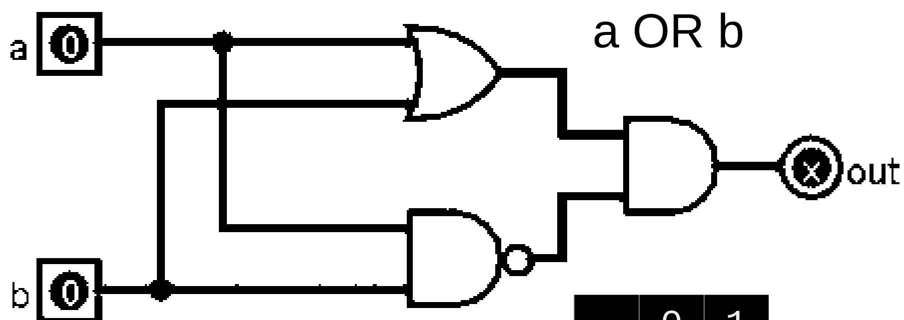
Basic circuits

- Circuits are equivalent if the truth tables are the same
  - **a XOR b = (a OR b) AND (a NAND b)**
  - **XOR(a, b) = AND(OR(a,b), NAND(a,b))**



^	0	1
0	0	1
1	1	0

XOR



|	0	1
0	0	1
1	1	1

a OR b

	0	1
0	1	1
1	1	0

a NAND b

	0	1
0	0	1
1	1	0

(a OR b) AND (a NAND b)

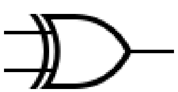
Basic circuits

- Circuits are equivalent if the truth tables are the same
  - **a XOR b = (a OR b) AND (a NAND b)**
  - **XOR(a, b) = AND(OR(a,b), NAND(a,b))**

a	b	a ^ b	f(a,b)
0	0	0	0
0	1	1	1
1	0	1	1
1	1	0	0

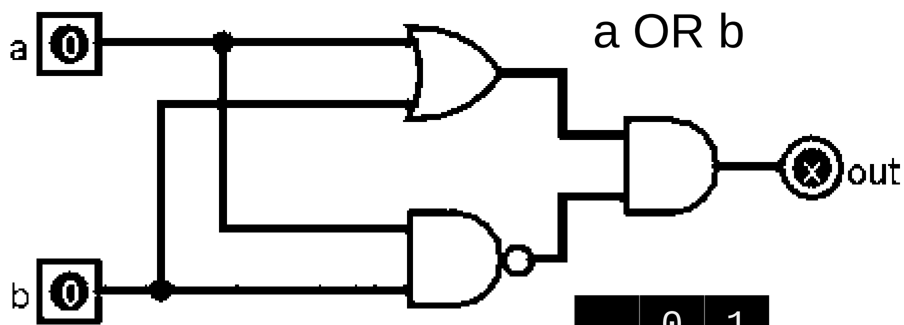
|	0	1
0	0	1
1	1	1

a OR b



^	0	1
0	0	1
1	1	0

XOR



	0	1
0	1	1
1	1	0

a NAND b

	0	1
0	0	1
1	1	0

(a OR b) AND
(a NAND b)

f(a,b)

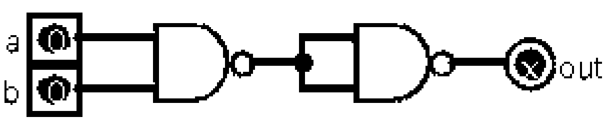
Important properties

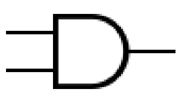
- Identity: **a AND 1 = a** **(a OR 0) = a**
- Constants: **a AND 0 = 0** **(a OR 1) = 1**
  - Also: **a NAND 0 = 1** **(a NOR 1) = 0**
- Inverses: **a NAND 1 = !a** **(a NOR 0) = !a**
  - Also: **a NAND a = !a** **a NOR a = !a**
- Double inverse: **!!a = a**
  - Or: **NOT(NOT(a)) = a**
- De Morgan's law: **!(a & b) = !a | !b**
  - Alternatively: **!(a | b) = !a & !b**

(remember this from CS 227!)

Universal gates

- NAND and NOR gates are universal
  - Each one alone can reproduce all other gates
  - Example: **a AND b** = a & b = !(!(a & b)) = !(a NAND b) = **(a NAND b) NAND (a NAND b)**





	0	1
0	1	1
1	1	0

a NAND b

	0	1
0	0	0
1	0	1

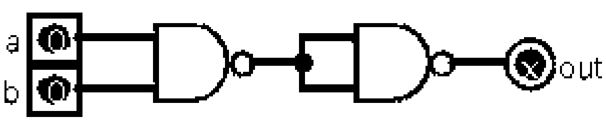
(a NAND b) NAND (a NAND b)

	0	1
0	0	0
1	0	1

a AND b

Universal gates

- NAND and NOR gates are universal
  - Each one alone can reproduce all other gates
  - Example: **a AND b** = a & b = !(!(a & b)) = !(a NAND b) = **(a NAND b) NAND (a NAND b)**
    - Similarly: **a AND b** = !(!(a & b)) = !(!a | !b) = !a NOR !b = **(a NOR a) NOR (b NOR b)**

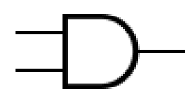


	0	1
0	1	1
1	1	0

a NAND b

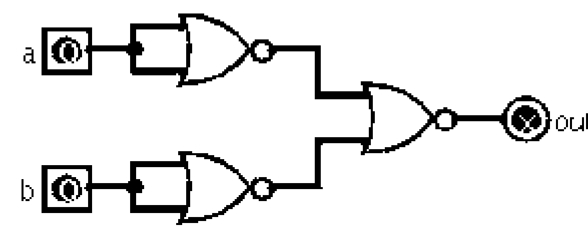
	0	1
0	0	0
1	0	1

(a NAND b) NAND (a NAND b)



	0	1
0	0	0
1	0	1

a AND b



(a NOR a) NOR (b NOR b)

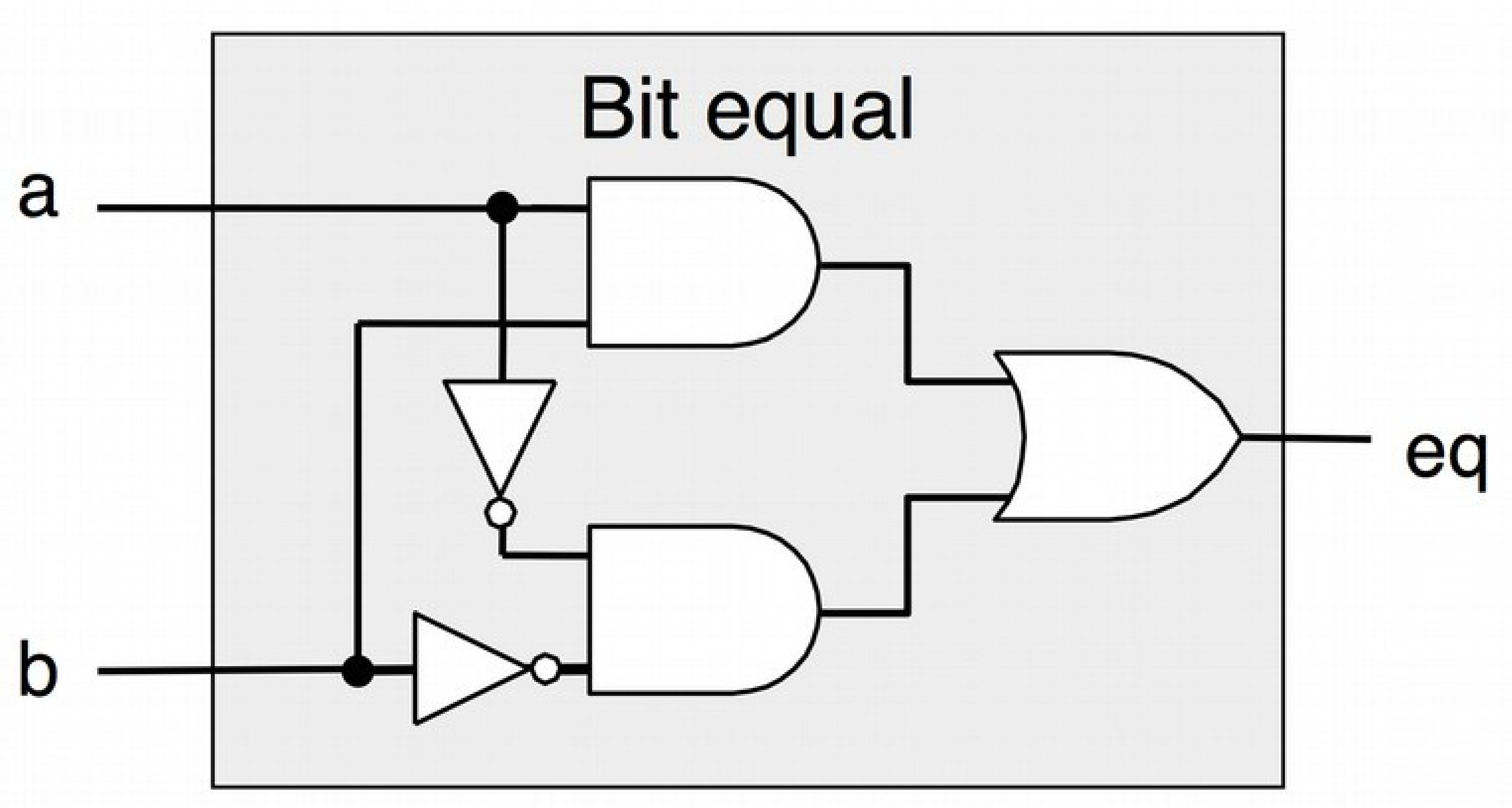
Circuit types

- Two main kinds of circuits:
  - Combinational circuits: outputs are a boolean function of inputs
    - Not time-dependent
    - Used for **computation**
  - Sequential circuits: output is dependent on previous outputs
    - Time-dependent
    - Used for **memory**

Computation

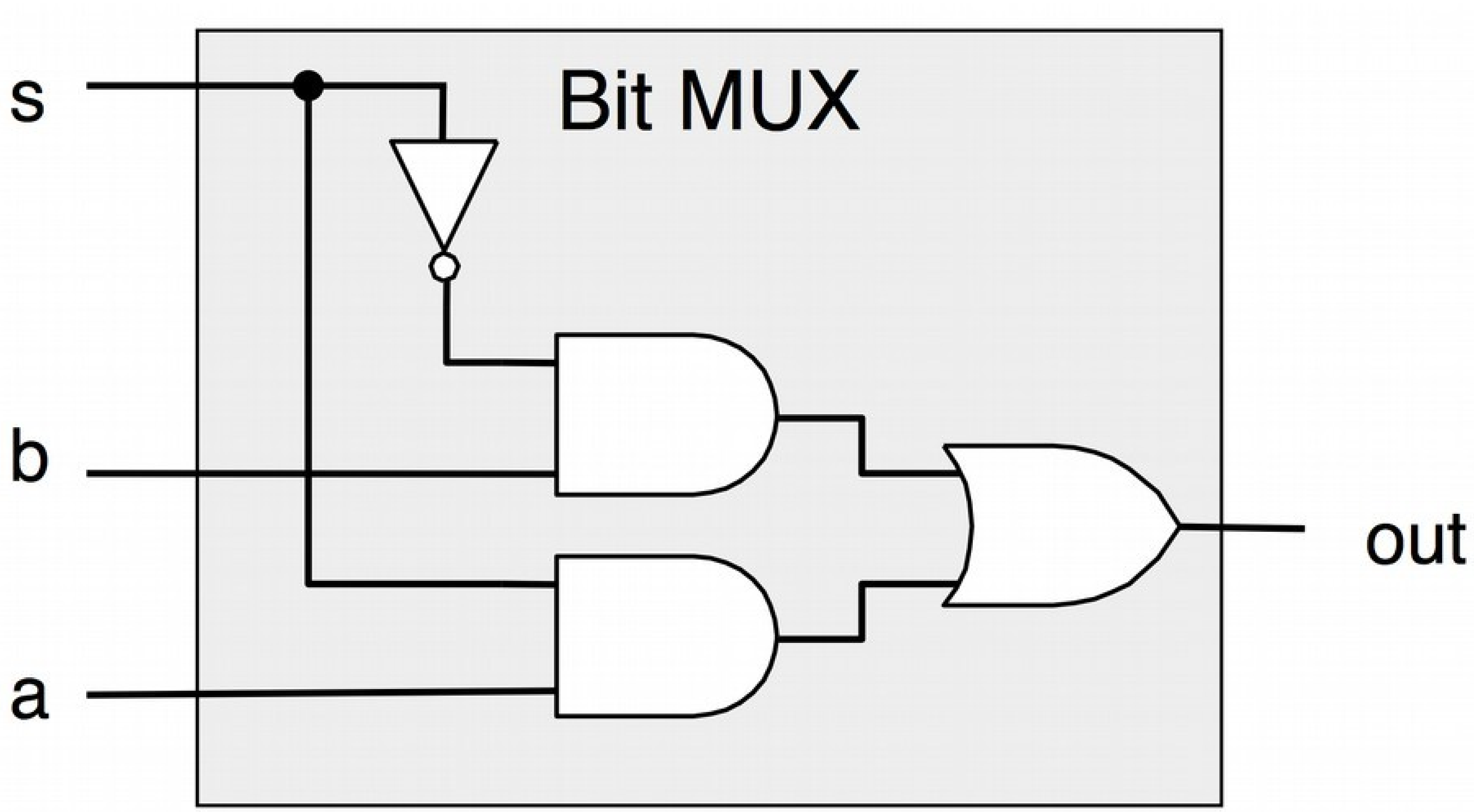
- Goal: identify circuits that perform useful computation
  - Testing bits to see if they're equal
  - Selecting between multiple inputs
  - Adding or subtracting bits
  - Bitwise operations (AND, OR, XOR)
  - Make them work on bytes instead of bits

Equality



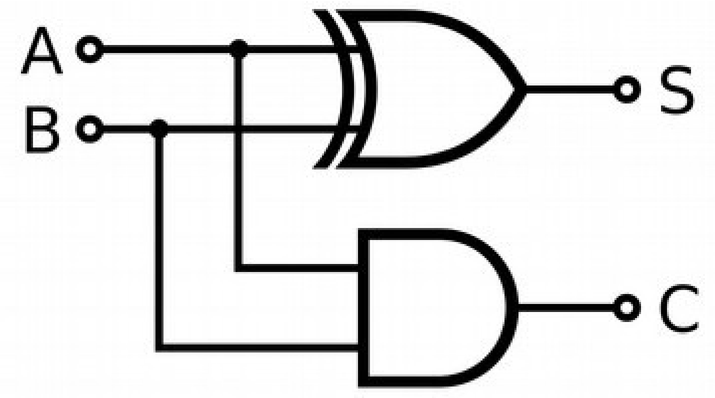
a EQ b = (a & b) | (!a & !b)

Multiplexor ("selector")



MUX (a, b, s) = (s & a) | (!s & b)

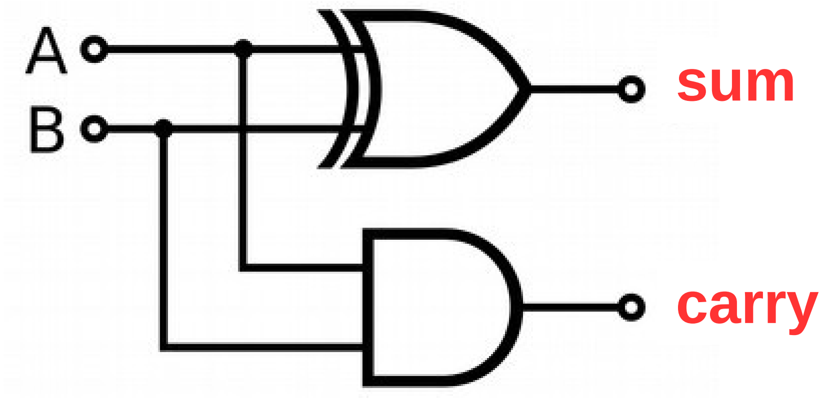
Half adders



Half Adder

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

Half adders



Half Adder

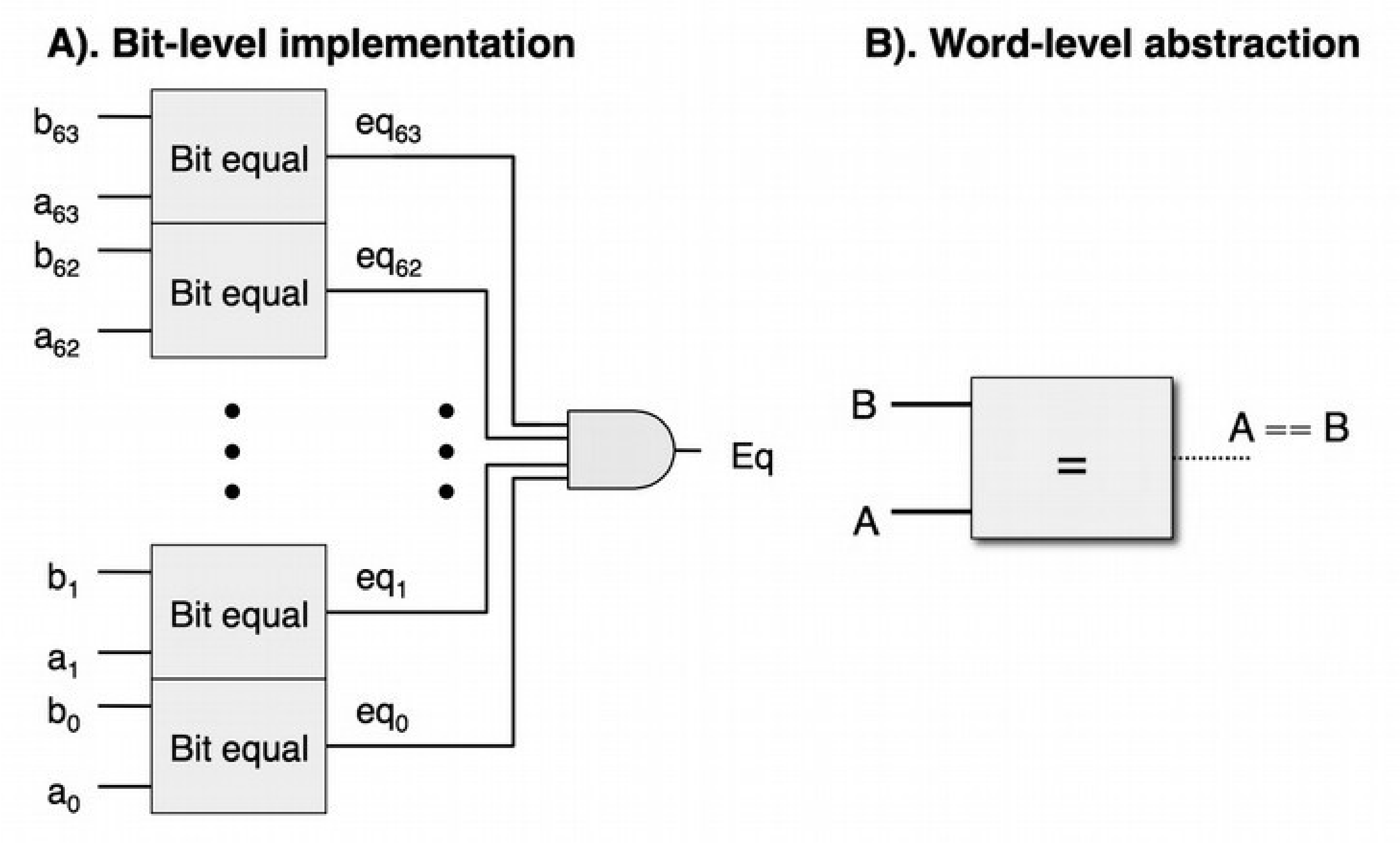
A	B	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

a + b = a ^ b + a & b

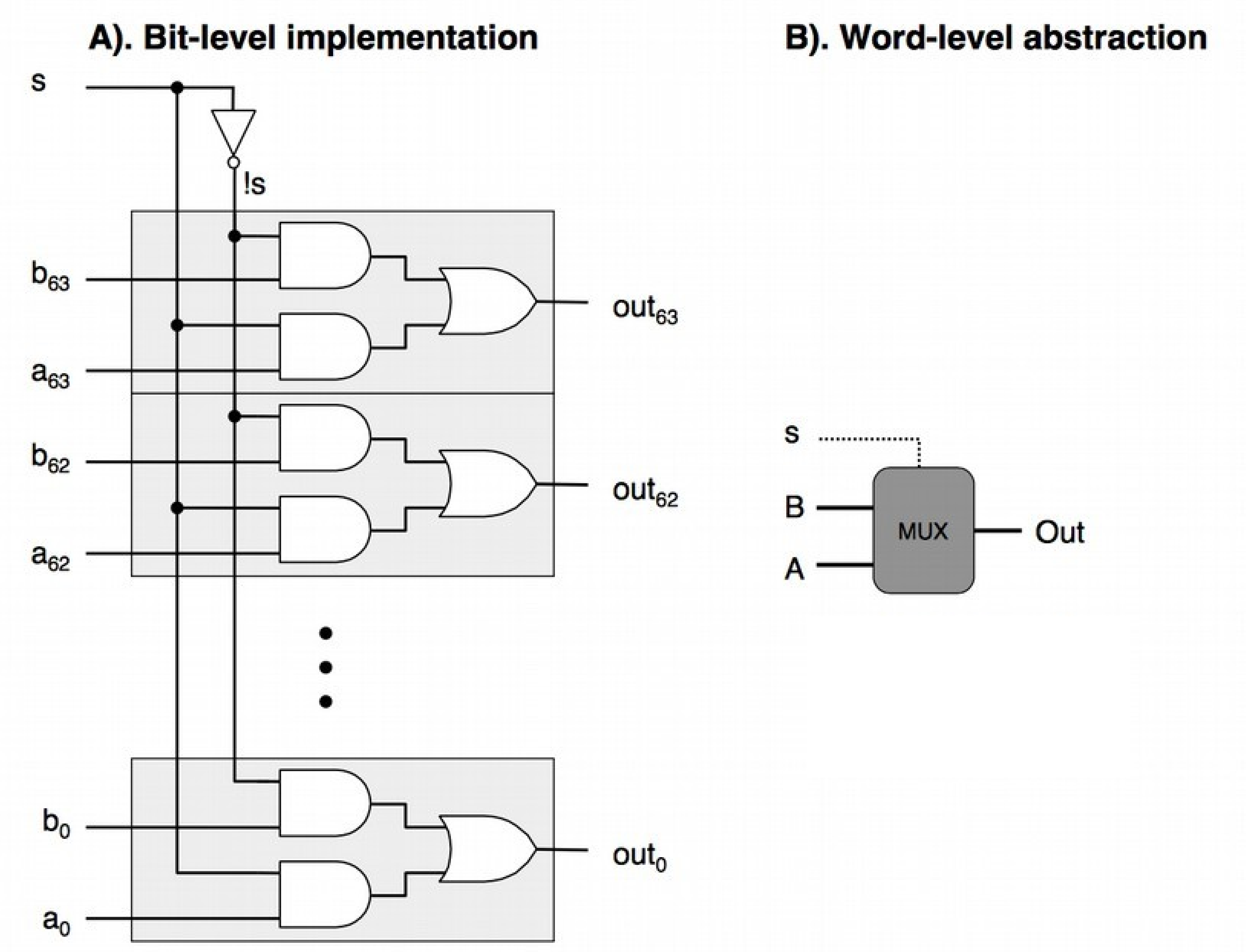
sum **carry**

Abstraction

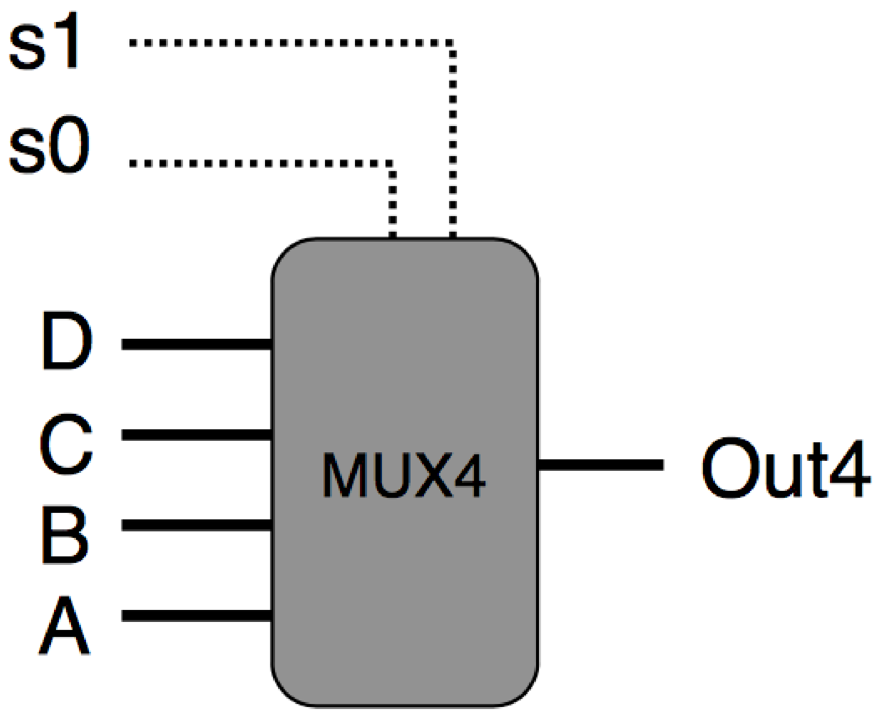
- Name circuits, then use them to build more complex circuits
  - E.g., use bit-level EQ to build a word-level equality circuit:



Word-level 2-way multiplexer



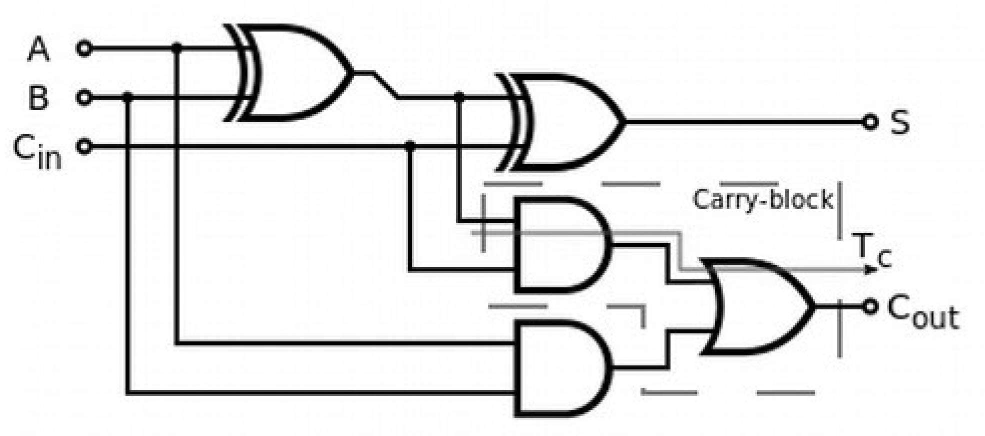
Word-level 4-way multiplexer



How many selector inputs would be required for eight data inputs?

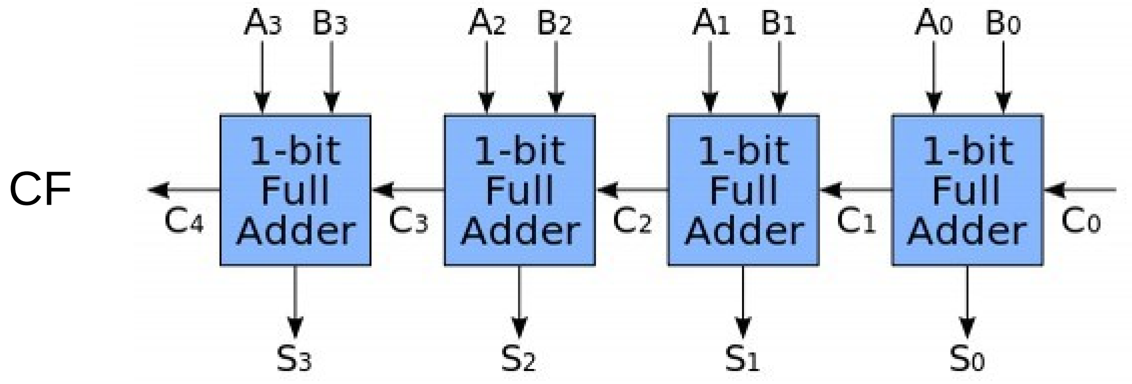
How many data inputs could be supported using four selector inputs?

Full adders

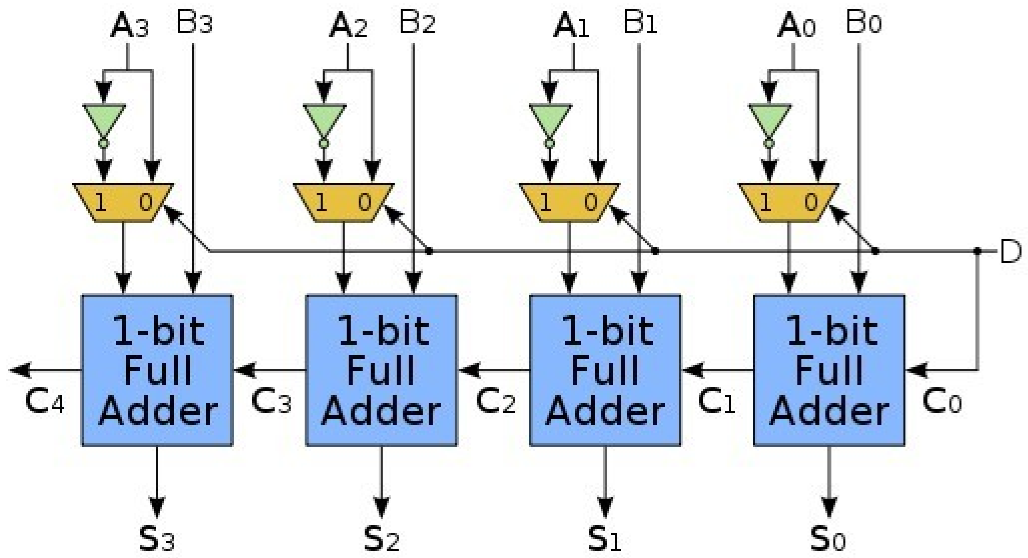


Full Adder

Connect full adders to build a ripple-carry adder that can handle multi-bit addition:



Adder/subtractor

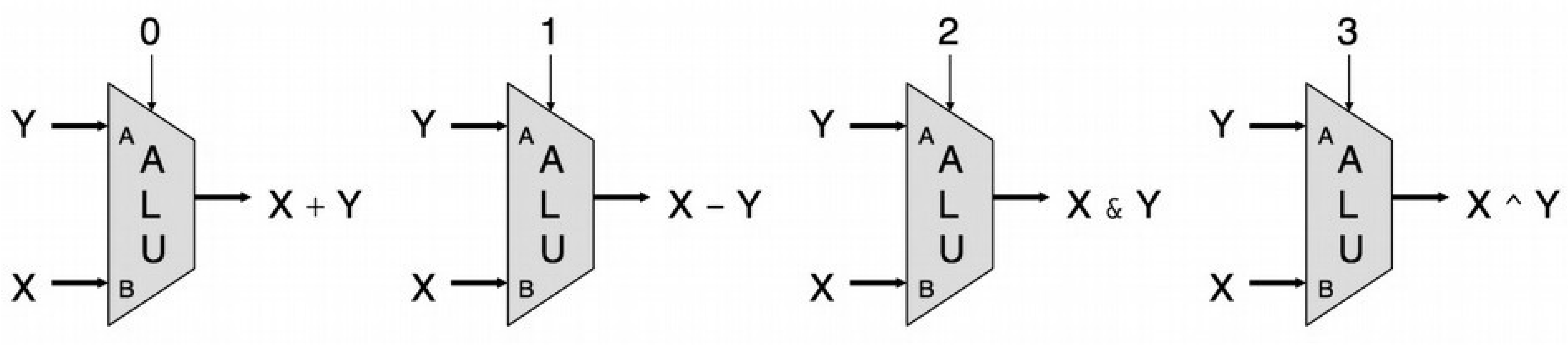


D=0: Addition
D=1: Subtraction

In two's complement: **B – A = B + !A + 1**

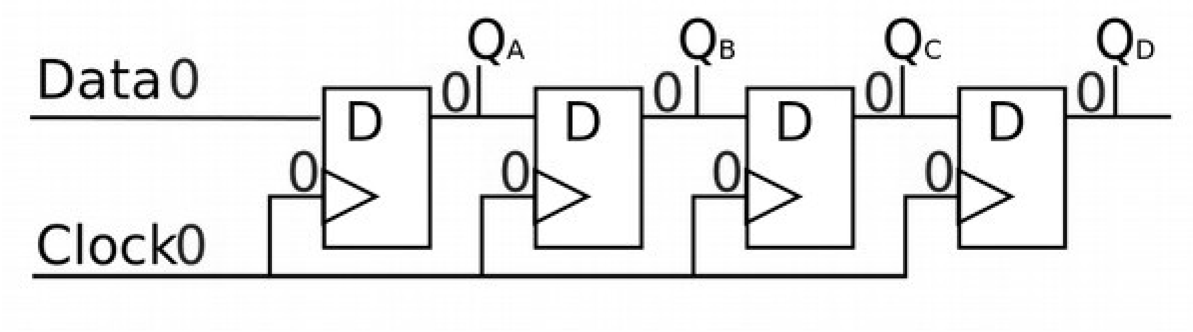
ALUs and memory

- Combine adders and multiplexors to make arithmetic/logic units
- Combine flip-flops to make register files and memory



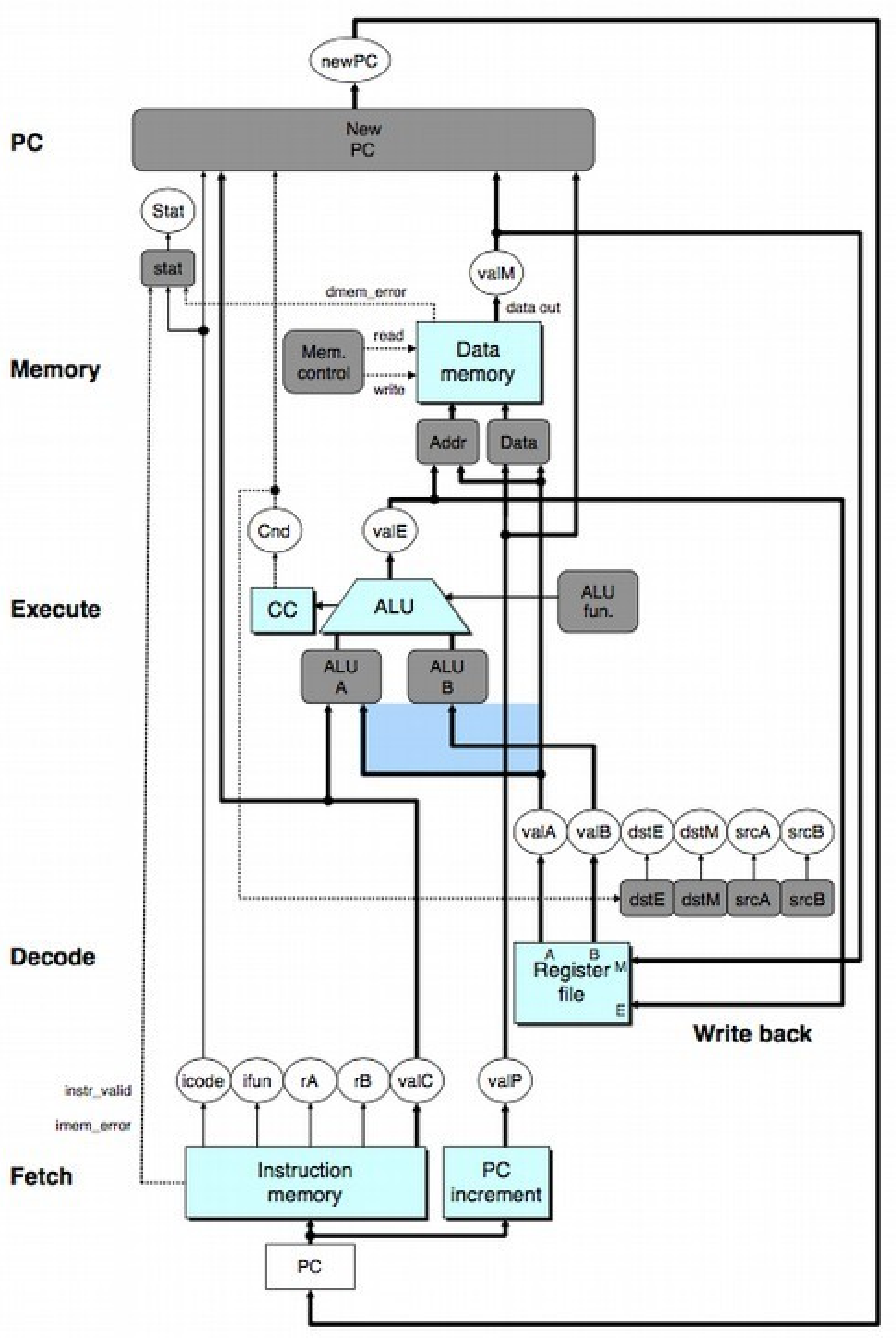
Basic Arithmetic Logic Unit (ALU)

4-bit Register



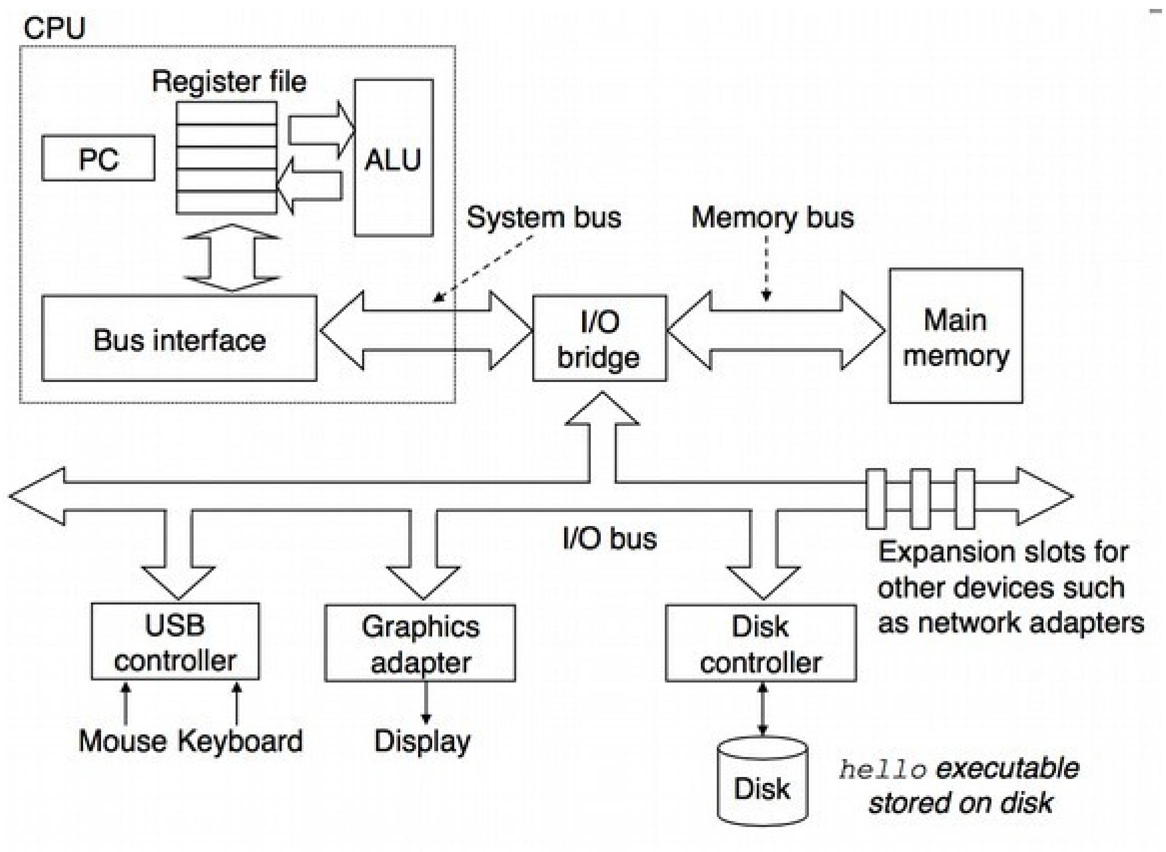
CPUs

- Combine ALU with registers and memory to make CPUs



Computers

- Combine CPU with other electronic components and devices (similarly constructed) communicating via buses to make a computer



Big picture

- Basic systems design approach: exploit abstraction
  - Start with simple components
  - Combine to make more complex components
  - Repeat using the new components as black box “simple components”

- This is true of most areas in systems
  - **CS 261**: transistors → gates → circuits → adders/flip-flops → ALUs/registers → CPUs/memory → computers
  - **CS 261**: machine code → assembly → C code → Java/Python code
  - **CS 361/470**: threads → processes → nodes → networks/clusters
  - **CS 432**: scanner → parser → analyzer → code generator → optimizer
  - **CS 450**: files + processes + I/O → kernel → operating system
  - **CS 456**: multiplexers → primitives → modules → CPUs (on FPGAs)
  - **CS 46?**: byte stream → frames → packets → datagrams → messages

Course status

- We've hit the bottom
  - Or at least as far down as we're going to go (logic gates); from here we go back up!
- Next week
  - Sequential circuits
  - CPU architecture

Suggestion: download **Logisim** (already installed on lab machines) and play around with some circuits!