

COMP311: *COMPUTER ORGANIZATION!*

Lecture 3: Memory Review, Basic Circuits

tinyurl.com/comp311-fa25

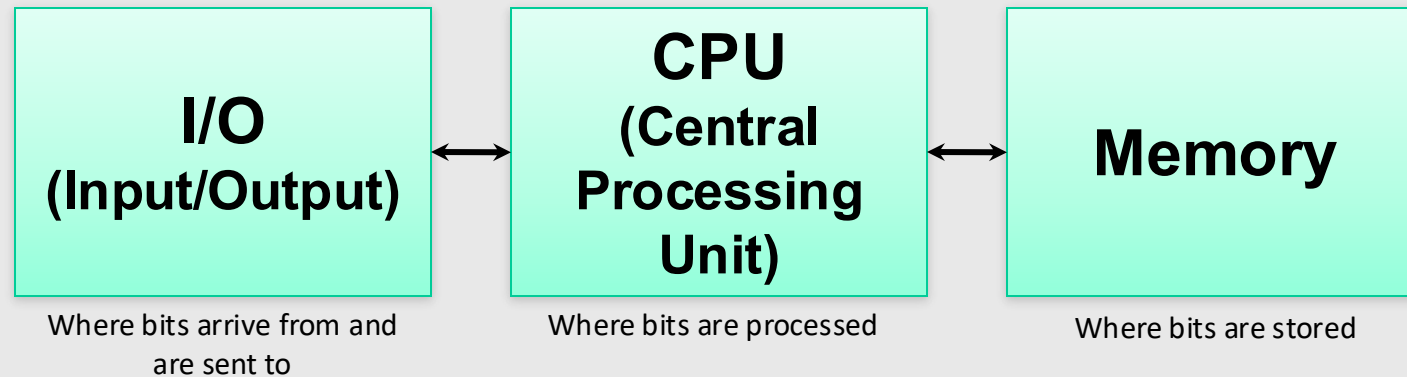
Logistics Updates

- First quiz next Thursday!
- First written assignment will be released tomorrow!
 - *Due a week after that*

Summarizing our review so far...

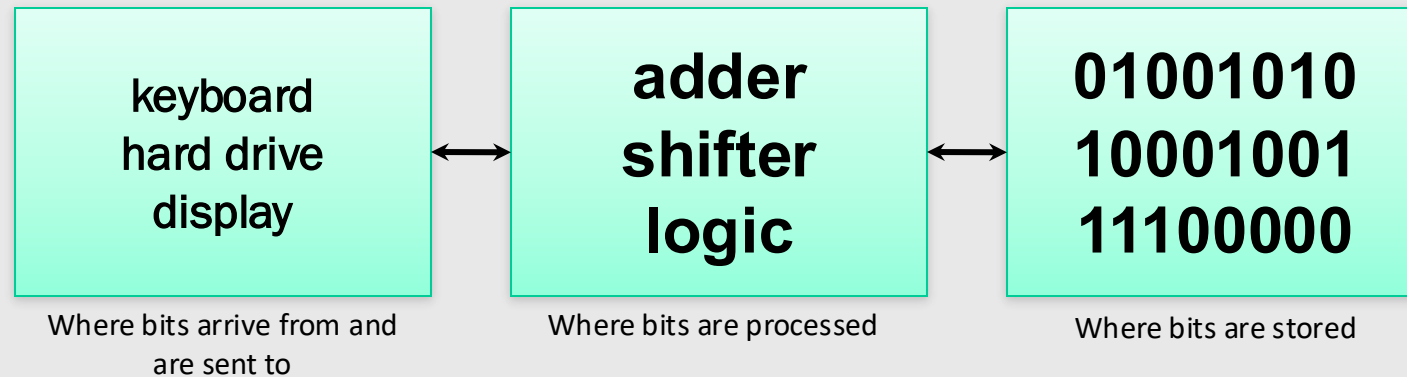
- **ALL** modern computers represent signed integers using a *two's-complement* representation
- **Two's-complement** representations eliminate the need for separate addition and subtraction units
 - Addition is *identical* using either unsigned and two's-complement numbers
- Finite representations of numbers on computers leads to anomalies
 - Overflow!
- Floating-point numbers have separate fraction and exponent components.

Today: Let's Dig into *Computer Organization*!



- Every computer has at least three basic units
 - *Input/Output*
 - where data arrives from the outside world
 - where data is sent to the outside world
 - where data is archived for the long term (i.e. when the lights go out)
 - *Memory*
 - where data is stored (numbers, text, lists, arrays, data structures)
 - *Central Processing Unit*
 - where data is manipulated, analyzed, etc.

Today: Let's Dig into *Computer Organization*!



■ Input/Output

- *converts symbols to bits and vice versa*
- *where the analog “real world” meets the digital “computer world”*
- *must somehow synchronize to the CPU’s clock*

■ Memory

- *stores bits that represent information*
- *every unit of memory has an “address” and “contents”,*

■ Central Processing Unit

- *besides processing, it also coordinates data’s movements between units*

What do we mean by “processing”?

- A CPU performs low-level operations called **INSTRUCTIONS**
- **Arithmetic**
 - *ADD X to Y then put the result in Z*
 - *SUBTRACT X from Y then put the result back in Y*
- **Logical**
 - *Set Z to 1 if X AND Y are 1, otherwise set Z to 0
(AND X with Y then put the result in Z)*
 - *Set Z to 1 if X OR Y are 1, otherwise set Z to 0
(OR X with Y then put the result in Z)*
- **Comparison**
 - *Set Z to 1 if X is EQUAL to Y, otherwise set Z to 0*
 - *Set Z to 1 if X is GREATER THAN OR EQUAL to Y, otherwise set Z to 0*
- **Control**
 - *Skip the next INSTRUCTION if Z is EQUAL to 0*

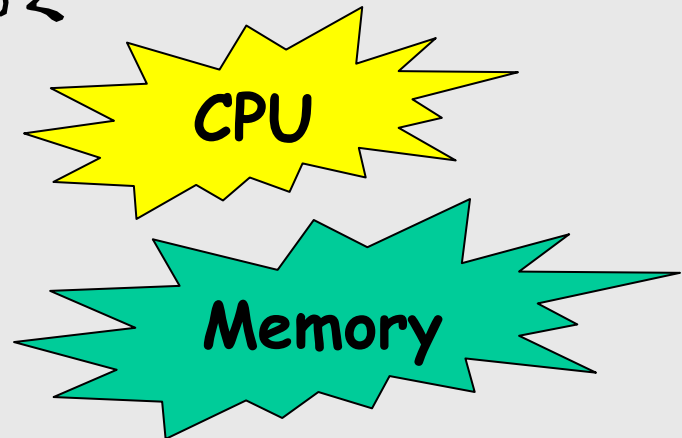
Anatomy of an Instruction

Nearly all instructions can be made to fit a common template...



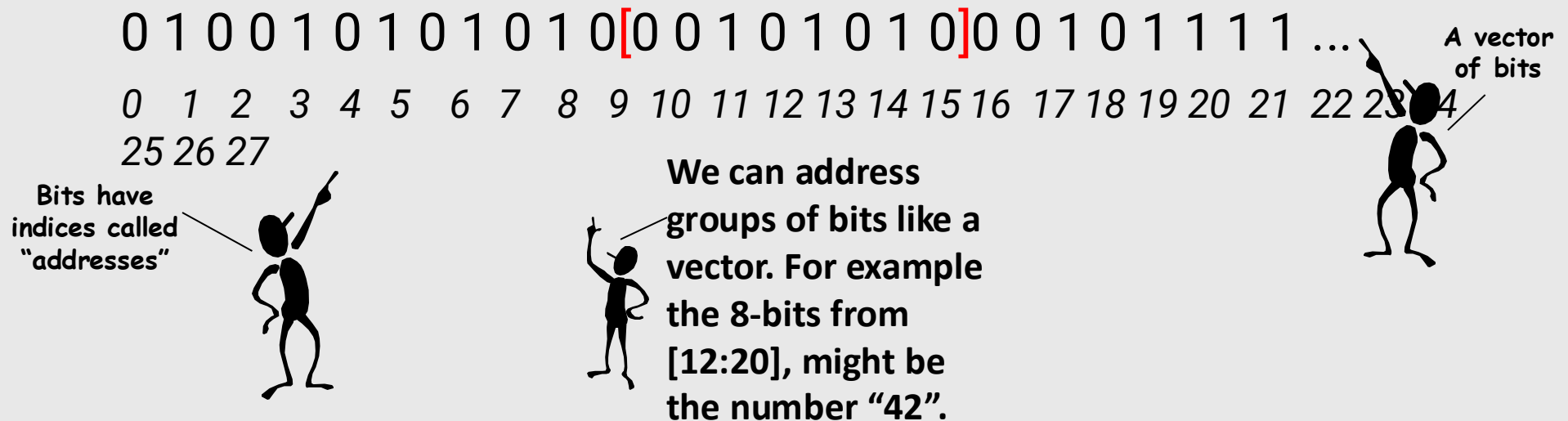
Issues remaining ...


- Which operations to include?
- Where to get variables and constants?
- Where to store the results?



How Memory is Organized

- A group of bits!
- **Groups of bits** can represent various types of data
 - *Integers, Signed integers. Floating-point values, Strings, Pixels*
- Memory is organized as a vector of bits with indices called “addresses”



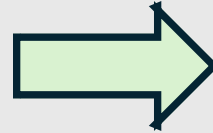


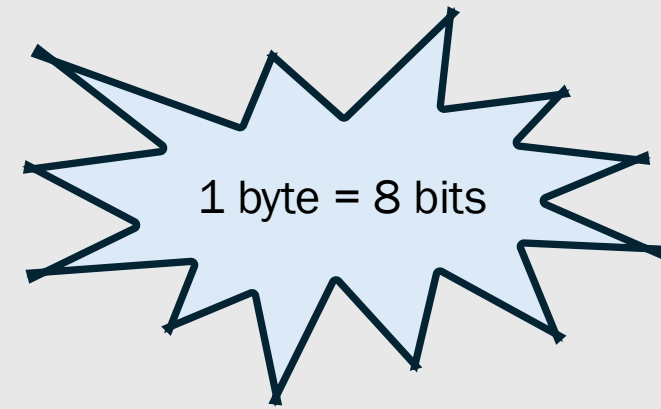
MORE 211 REVIEW: MEMORY



Storing Data in Memory

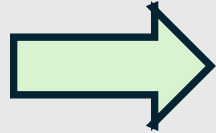
Each of these rows can
store one byte of data





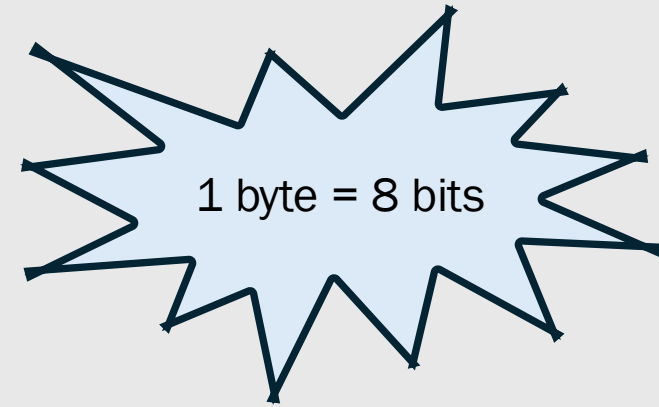
Storing Data in Memory

Each of these rows can
store one byte of data



0b0000 0101
0b0000 0100
0b1111 1110

Let's say we want to store the
number 5 in the top row, 4 in
the next row, and -2 in the
next row.



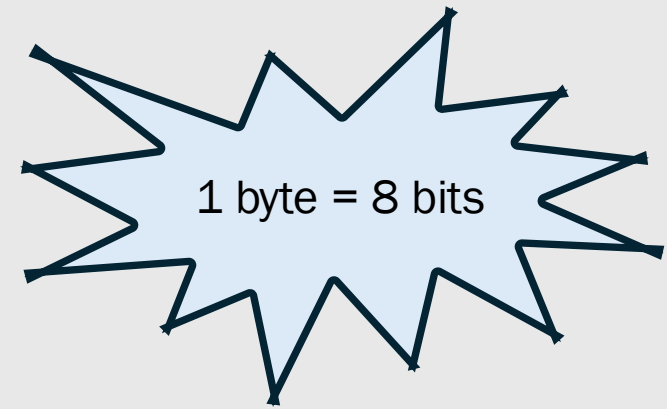
Byte Addresses

Byte
Address

0	0b0000 0101
1	0b0000 0100
2	0b1111 1110
3	
4	
5	
6	
7	
8	
9	
10	
11	

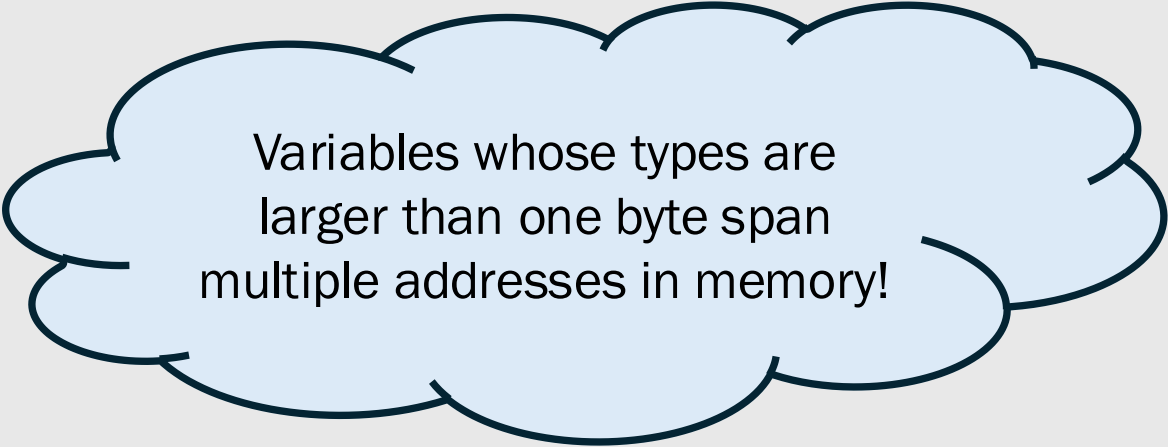
To reference each of these locations, we use something called a byte address.

Note that these addresses are not stored anywhere!



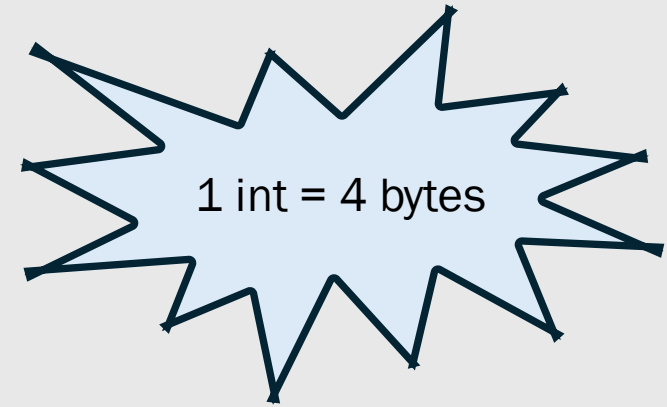
Storing Larger Data

- What if I want to store an integer?
 - *sizeof(int) = 4 bytes*



Variables whose types are
larger than one byte span
multiple addresses in memory!

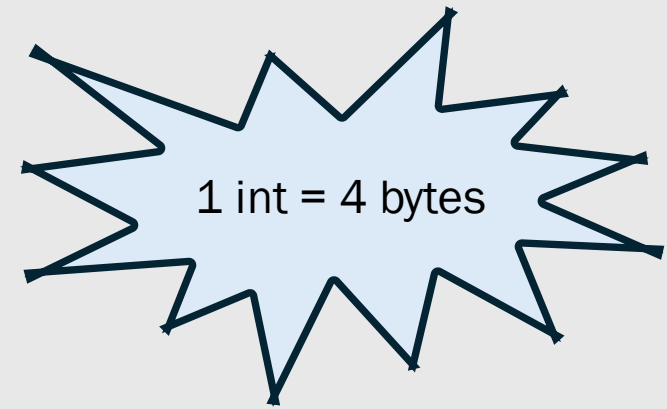
Storing Larger Data



- Let's say I want to store an integer whose value is 329,421

329421 = 0b 0000 0000 0000 0101 0000 0110 1100 1101

Storing Larger Data



- Let's say I want to store an integer whose value is 329,421

329421 = 0b 0000 0000 0000 0101 0000 0110 1100 1101

Byte
Address

Data

0

1

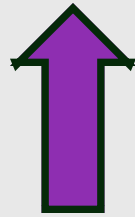
2

3

What order
should we
store the bits?

MSB and LSB

0b 0000 0000 0000 0101 0000 0110 1100 1101



Most Significant Byte
(MSB)

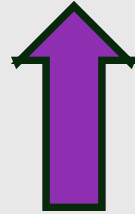


Least Significant Byte
(LSB)

Little Endian Ordering

- The least significant byte of the integer goes in the lowest address

0b 0000 0000 0000 0101 0000 0110 1100 1101



Most Significant Byte
(MSB)



Least Significant Byte
(LSB)

Byte
Address

Data

0

1

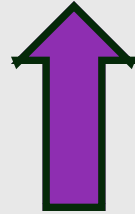
2

3

Little Endian Ordering

- The least significant byte of the integer goes in the lowest address

0b 0000 0000 0000 0101 0000 0110 1100 1101



Most Significant Byte
(MSB)



Least Significant Byte
(LSB)

Byte
Address

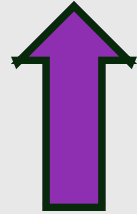
Data

0	1100 1101
1	0000 0110
2	0000 0101
3	0000 0000

Big Endian Ordering

- The least significant byte of the integer goes in the largest address

0b 0000 0000 0000 0101 0000 0110 1100 1101



Most Significant Byte
(MSB)



Least Significant Byte
(LSB)

Byte
Address

Data

0

1

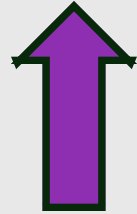
2

3

Big Endian Ordering

- The least significant byte of the integer goes in the largest address

0b 0000 0000 0000 0101 0000 0110 1100 1101



Most Significant Byte
(MSB)



Least Significant Byte
(LSB)

Byte
Address

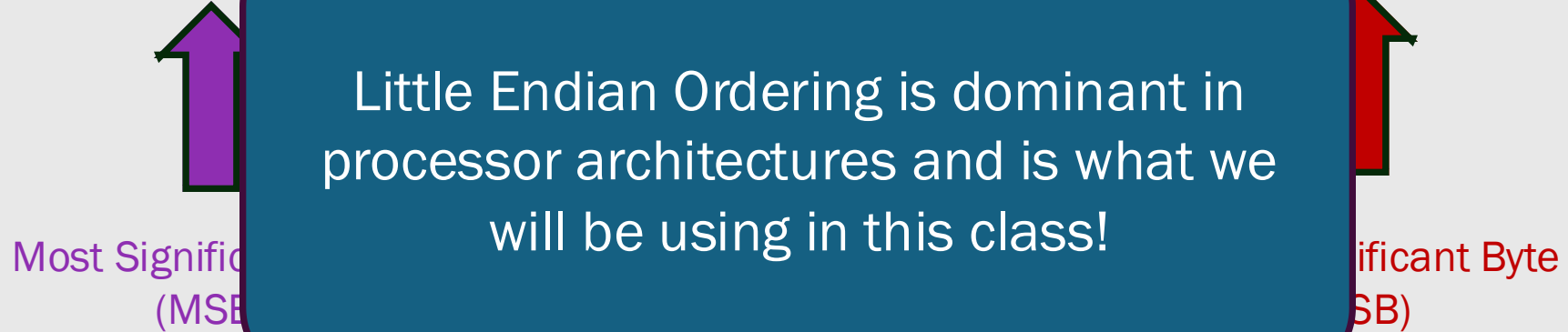
Data

0	0000 0000
1	0000 0101
2	0000 0110
3	1100 1101

Little Endian Ordering

- The least significant byte of the integer goes in the lowest address

0b 0000 0110 1101



Address

0	1100 1101
1	0000 0110
2	0000 0101
3	0000 0000

Working with memory

- How do you know if an individual row should be interpreted as a one-byte value or if it is part of a larger number?
 - *You cannot tell from looking at the memory – you'll have this information stored somewhere else!*
 - *When you send memory requests, you specify the address and the size of data you want to read.*

Integer Array

0	1	2	3	4
5	90	100	40	11

sizeof(int) = 4 bytes

Byte
Address

Data

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	
13	
14	
15	
16	
17	
18	
19	

Byte Address	Data
0	5
1	
2	
3	
4	90
5	
6	
7	
8	100
9	
10	
11	
12	40
13	
14	
15	
16	11
17	
18	
19	

Byte Address	Data
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	
13	
14	
15	
16	
17	
18	
19	

Byte Address	Data
0	5
1	
2	
3	
4	90
5	
6	
7	
8	100
9	
10	
11	
12	40
13	
14	
15	
16	11
17	
18	
19	

Byte Address	Data
0	0b 0000 0101
1	0b 0000 0000
2	0b 0000 0000
3	0b 0000 0000
4	0b 0101 1010
5	0b 0000 0000
6	0b 0000 0000
7	0b 0000 0000
8	0b 0110 0100
9	0b 0000 0000
10	0b 0000 0000
11	0b 0000 0000
12	0b 0010 1000
13	0b 0000 0000
14	0b 0000 0000
15	0b 0000 0000
16	0b 0000 1011
17	0b 0000 0000
18	0b 0000 0000
19	0b 0000 0000

Integer Array

0	1	2	3	4
5	90	100	40	11

`sizeof(int) = 4 bytes`

Assuming this array starts at
address 0, this is how it is stored in
memory

Byte
Address

Data

0	
1	
2	5
3	
4	
5	90
6	
7	
8	
9	100
10	
11	
12	
13	40
14	
15	
16	
17	11
18	
19	

Integer Array

0	1	2	3	4
5	90	100	40	11

The address of each element in the array corresponds to its starting byte in memory.

Since the size of each element in this array is 4 bytes, we can compute the address of any element by taking its index and multiplying it by 4.

Base address = address of the start of the array

Byte
Address

Data

0	5
1	
2	
3	
4	90
5	
6	
7	
8	100
9	
10	
11	
12	40
13	
14	
15	
16	11
17	
18	
19	

Address Calculation

- I have a 10-element integer array whose base address is 20.
What is the address of index 9?

WS 2 Q2

Address Calculation

- I have a 10-element integer array whose base address is 20.
What is the address of index 9?
 - *array's base address + index * size of integer*
 - $20 + 9 * 4 = 56$

Diagram Configurations

- Sometimes memory diagrams will show 4 bytes per row instead of 1 byte per row
 - *See side-by-side comparison on the next slide*
- This provides a more compact way to visualize the data
- Much of the data that we are working with is 4 bytes, so this structure makes it easier to see what is stored in memory

Byte Address	Data
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	
13	
14	
15	
16	
17	
18	
19	

Byte Address	Data
0	
4	
8	
12	
16	
20	
24	
28	
32	
36	
40	
44	
48	
52	
56	
60	
64	
68	
72	
76	

Byte Address	Data
0	5
1	
2	
3	
4	90
5	
6	
7	
8	100
9	
10	
11	
12	40
13	
14	
15	
16	11
17	
18	
19	

Byte Address	Data
0	5
4	90
8	100
12	40
16	11
20	
24	
28	
32	
36	
40	
44	
48	
52	
56	
60	
64	
68	
72	
76	

Byte Address	Data
0	0b 0000 0101
1	0b 0000 0000
2	0b 0000 0000
3	0b 0000 0000
4	0b 0101 1010
5	0b 0000 0000
6	0b 0000 0000
7	0b 0000 0000
8	0b 0110 0100
9	0b 0000 0000
10	0b 0000 0000
11	0b 0000 0000
12	0b 0010 1000
13	0b 0000 0000
14	0b 0000 0000
15	0b 0000 0000
16	0b 0000 1011
17	0b 0000 0000
18	0b 0000 0000
19	0b 0000 0000

Byte Address	Data
0	0b 0000 0000 0000 0000 0000 0000 0101
4	0b 0000 0000 0000 0000 0000 0000 0101 1010
8	0b 0000 0000 0000 0000 0000 0000 0110 0100
12	0b 0000 0000 0000 0000 0000 0000 0010 1000
16	0b 0000 0000 0000 0000 0000 0000 0000 1011
20	
24	
28	
32	
36	
40	
44	
48	
52	
56	
60	
64	
68	
72	
76	

Addresses in Hex

- Addresses are normally written in hex, not decimal
- In this class, we are working in a 32-bit address space, meaning that addresses are 4 bytes
- The rest of the memory diagrams you see in class will look the diagrams on the next slide
- Notice that the compactness of the right diagram allows us to display more data at once than the left diagram

Byte Address	Data
0x00000000	
0x00000001	
0x00000002	
0x00000003	
0x00000004	
0x00000005	
0x00000006	
0x00000007	
0x00000008	
0x00000009	
0x0000000A	
0x0000000B	

Each row stores **one** byte of data

Byte Address	Data
0x00000000	
0x00000004	
0x00000008	
0x0000000C	
0x00000010	
0x00000014	
0x00000018	
0x0000001C	
0x00000020	
0x00000024	
0x00000028	
0x0000002C	

Each row stores **four** bytes of data

Size of Memory Address

- The size of our memory determines the number of bits we need to address it
- For example, if our memory can only hold 8 bytes, we will need $\log_2(8) = 3$ bits to address the memory

Byte Address	Data
0b000	
0b001	
0b010	
0b011	
0b100	
0b101	
0b110	
0b111	

Size of Memory Address

- If the size of our memory is 4 GB, how many bits do we need to address our memory? (1GB = 2^{30} bytes)

WS 2 Q3

Size of Memory Address

- If the size of our memory is 4 GB, how many bits do we need to address our memory? (1GB = 2^{30} bytes)

$$\log_2(4 \text{ GB})$$

$$\log_2(2^2 * 2^{30})$$

$$\log_2(2^{32})$$

$$32$$



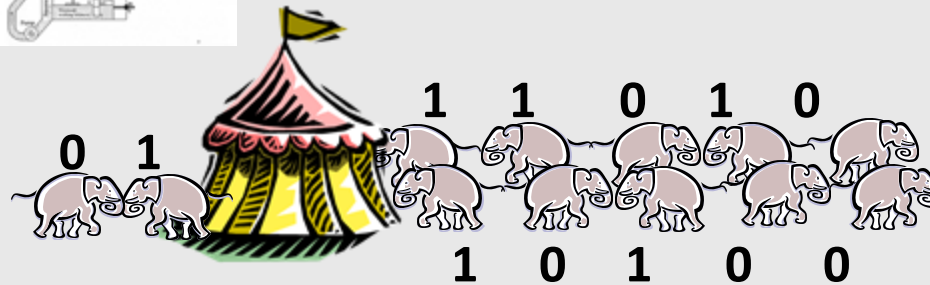
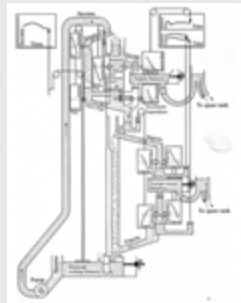
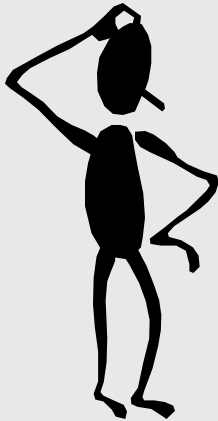
BASIC CIRCUITS



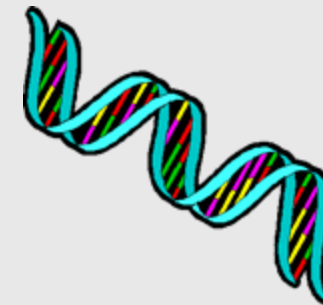
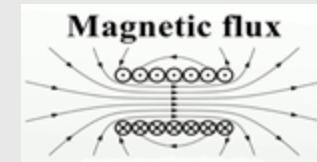
A Substrate for Computation

- We can build devices for processing and representing bits using almost any physical phenomenon

Wait! Some of those
might have potential...

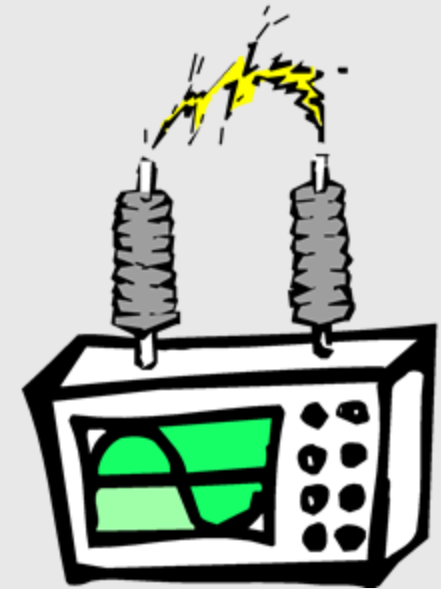


- magnetic flux
- trained elephants
- falling water
- turning gears
- DNA sequences
- polarization of a photon



Using Electromagnetic Phenomena

- Some EM things we could encode bits with:
 - *voltages, phase, currents, frequency*
- With today's technologies **voltages** are most often used.
- Voltage pros:
 - *easy generation, detection*
 - *voltage changes can be very fast*
 - *lots of engineering knowledge*
- Voltage cons:
 - *easily affected by environment*
 - *DC connectivity required?*
 - *R & C effects slow things down*



Voltage and Current

■ Voltage

- *The force that makes electrons flow*
- *Measured in Volts (V)*

■ Current

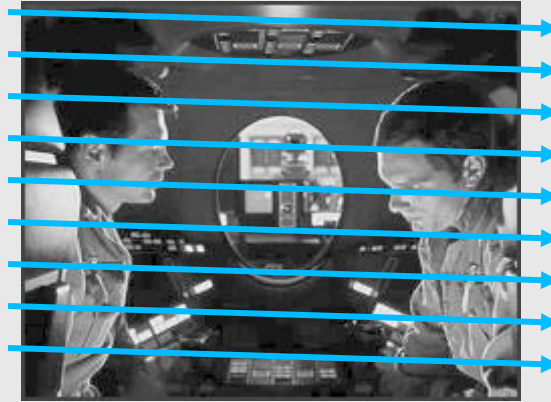
- *The rate of flow of electrons*
- *Measured in Amperes (A)*



Representing Information with Voltages

Representation of each point (x, y) in a B&W Picture:

0 volts: BLACK
1 volt: WHITE
0.37 volts: 37% Gray
etc.



Representation of a picture:

Scan points in some prescribed
raster order... generate voltage
waveform

**How much
information
at each point?**

Information Processing = Computation

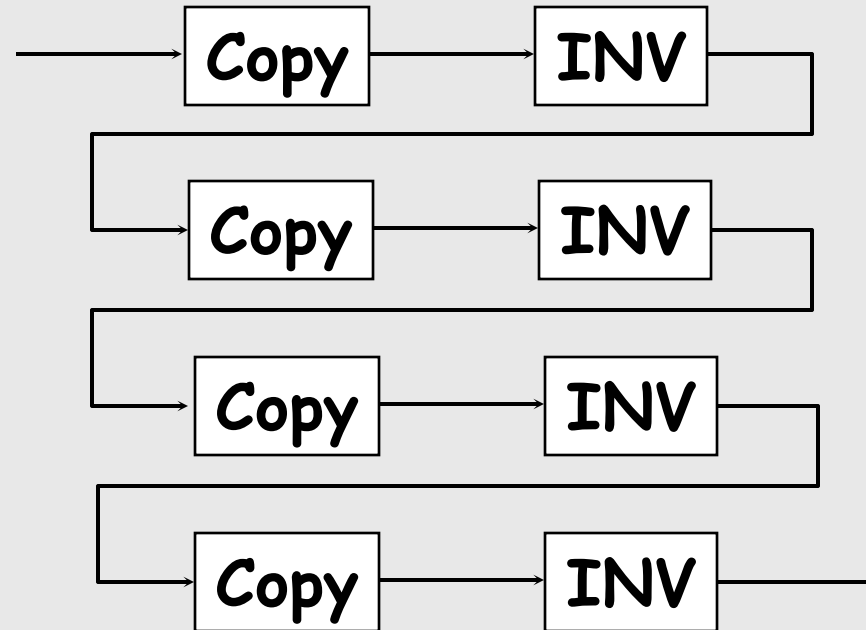
First, let's consider some processing blocks:



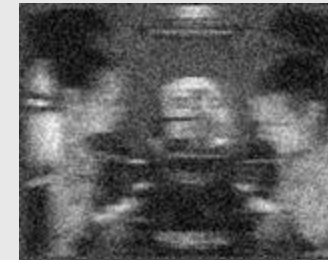
Let's Build a System!



input



(Reality)

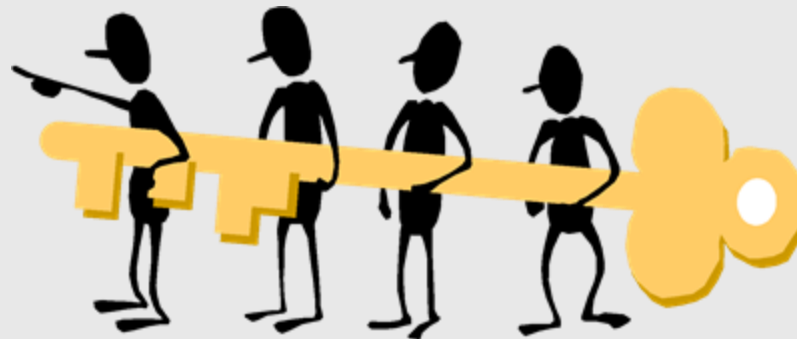


output

Contracts: Key to System Design

A SYSTEM is a structure that is “guaranteed” to exhibit a specified behavior, assuming all of its components obey their specified behaviors.

How is this achieved?



Through Contracts

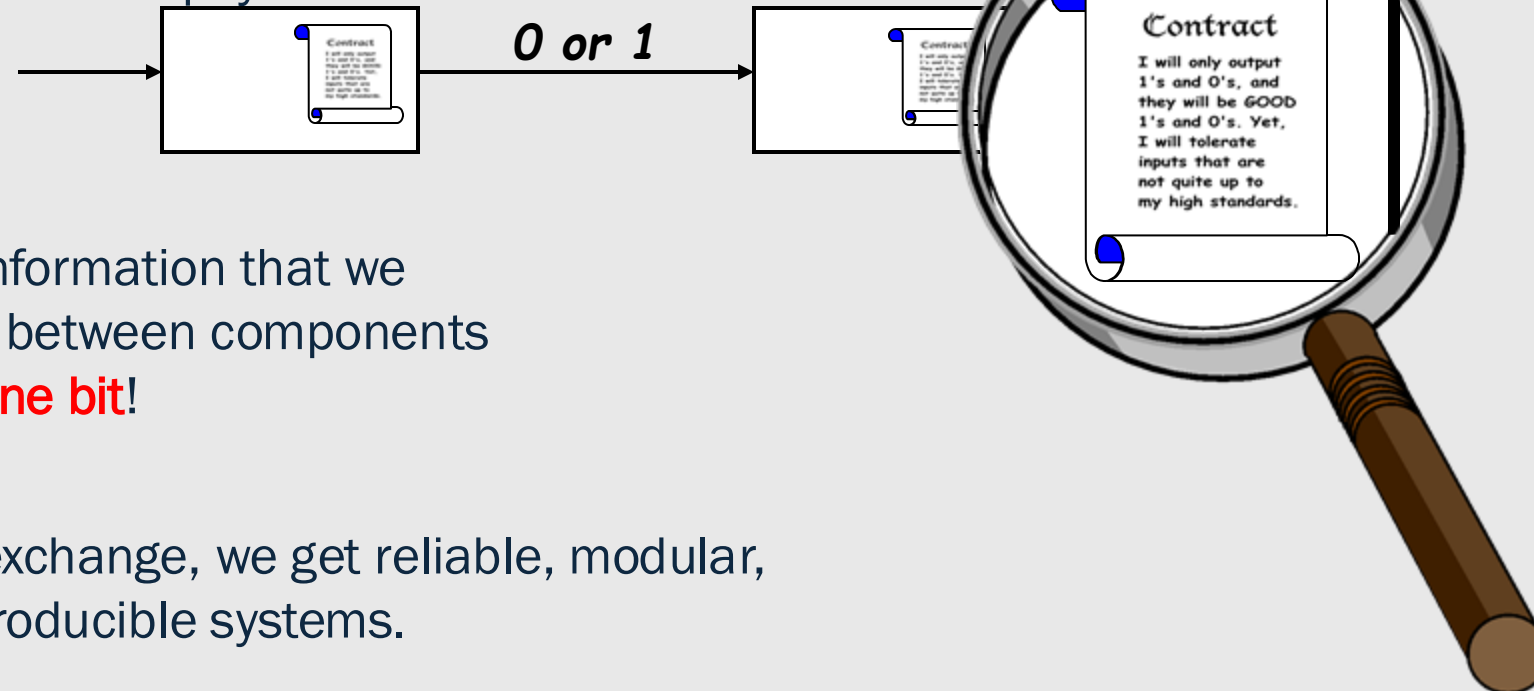
Every system component will have clear obligations and responsibilities. If these are maintained we have every right to expect the system to behave as planned. If contracts are violated all bets are off.

Digital Contracts

Why DIGITAL?

... because it keeps the contracts SIMPLE!

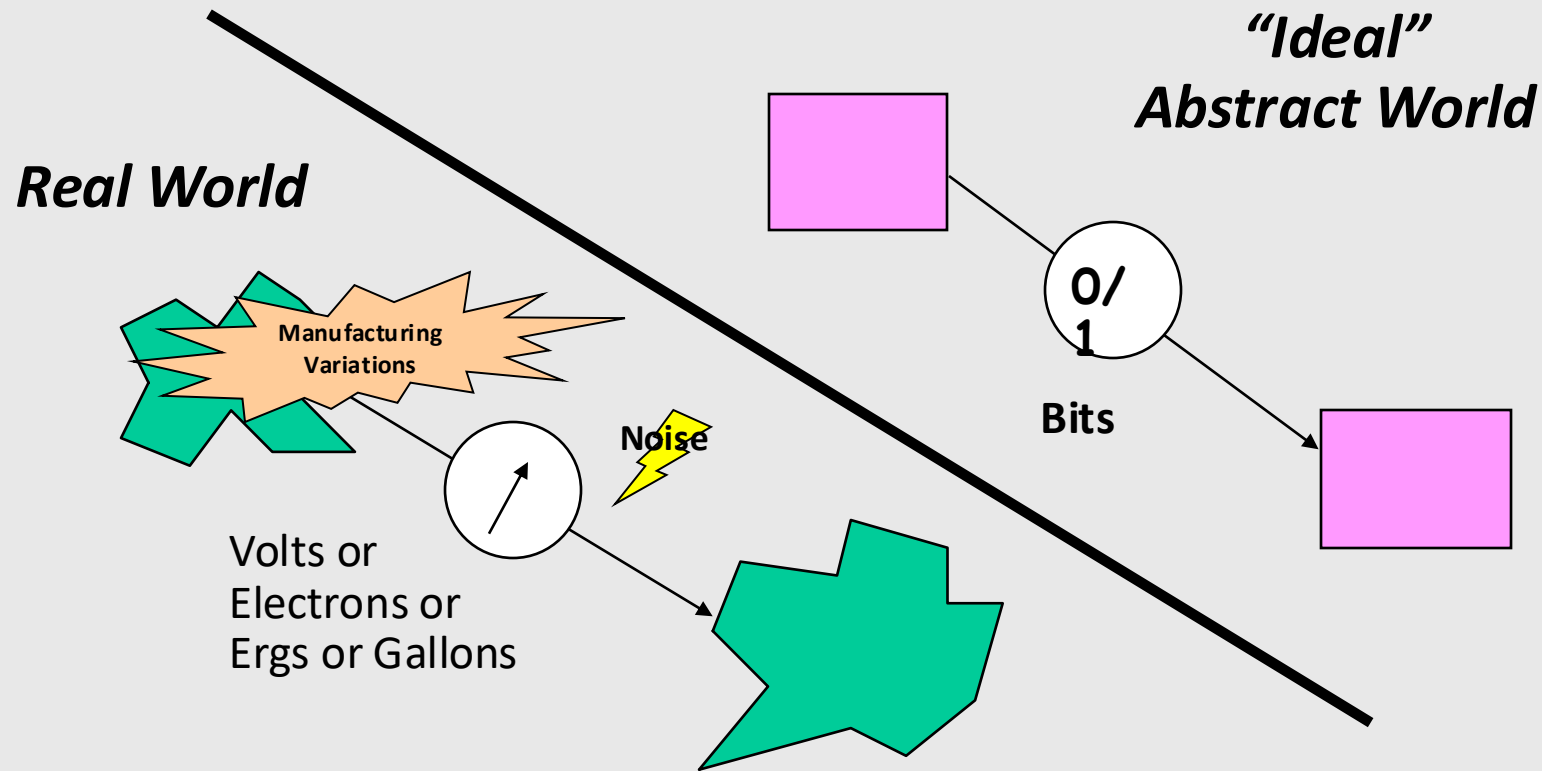
It's the price we pay for this robustness?



All the information that we transfer between components is only **one bit**!

But, in exchange, we get reliable, modular, and reproducible systems.

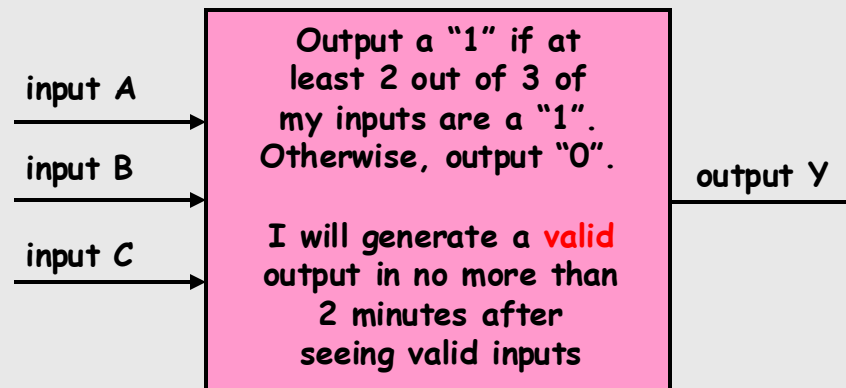
The Digital Abstraction



Keep in mind, **the real world is not digital, we engineer it to behave that way.**
We coerce real physical phenomena to implement digital designs!

A Digital Processing Element

- A **combinational device** is a digital element that has
 - one or more digital inputs
 - one or more digital outputs
 - a functional specification that details the value of each output for **every possible combination of valid input values**
 - a timing specification consisting (at a minimum) an upper bound propagation delay, t_{pd} , on the required time for the device to compute the specified **valid** output values from an arbitrary set of stable, **valid** input values



A Combinational Digital System

A system of interconnected elements is combinational if:

- Each circuit element is combinational
- Every input is connected to exactly one output or directly to some source of 0's or 1's
- The circuit contains no directed cycles
- But, in order to realize digital processing, elements we have one more requirement!

Nose Margins

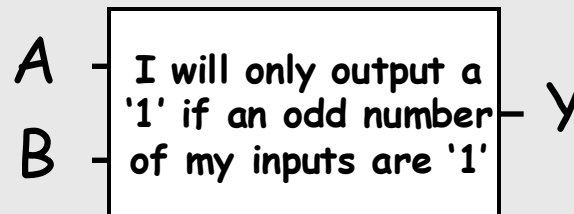
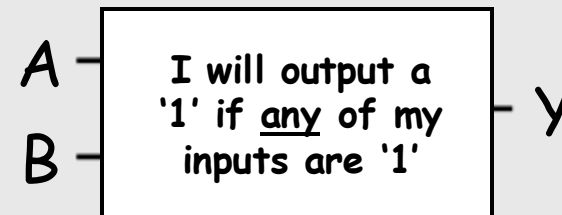
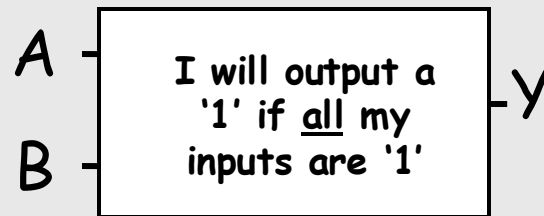
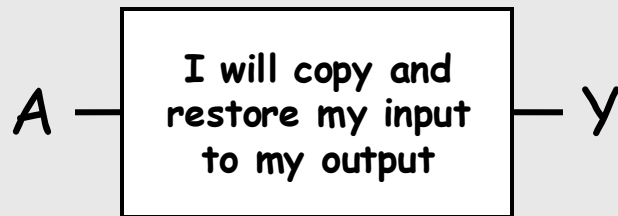
- Key idea:
Don't allow "0" to be mistaken for a "1" or vice versa
- Use the same "uniform bit-representation convention", for every component in our digital system
- To implement devices with high reliability, we outlaw "close calls" via a representation convention which forbids a range of voltages between "0" and "1".
- Ensure the valid input range is more tolerant (larger) than the valid output range



Our definition of valid does not preclude inputs and outputs from passing through invalid values. In fact, they must, but only during transitions. Our specifications allow for this (i.e. outputs are specified sometime (T_{pd}) after after inputs become valid).

Digital Processing Elements (Gates)

Some digital processing elements occur so frequently that we give them special names and symbols

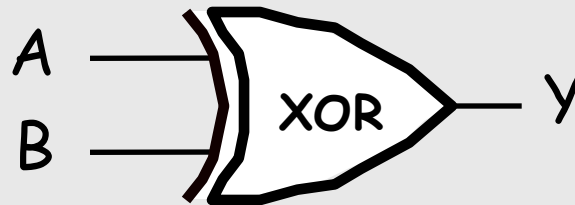
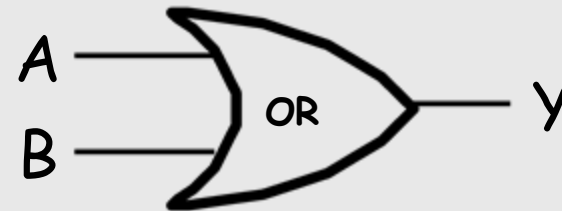
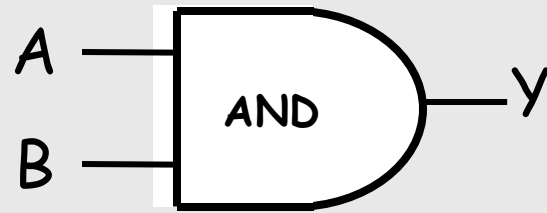
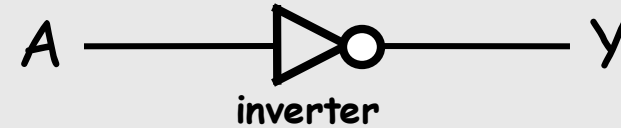


Q: What is the point of a buffer?
Doesn't a wire do the same thing?
A: A buffer restores marginal digital signals, because the output is as good or "better" than the input (i.e. it solves that bad image problem from slide 7).



Digital Processing Elements (Gates)

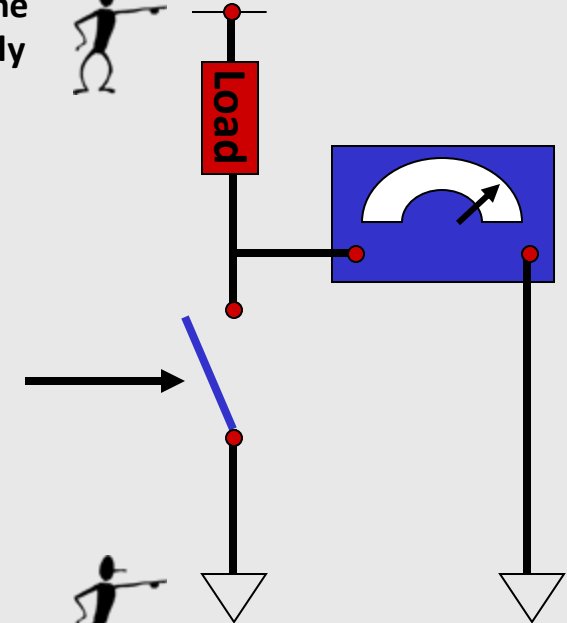
Some digital processing elements occur so frequently that we give them special names and symbols



How do we make gates?

- A controllable switch is the common link of all computing technologies
- Switches control voltages by creating and opening paths between higher and lower potentials
- Closed circuit:
 - *fully connected, allows electricity to flow uninterrupted*
- Open circuit
 - *A circuit that contains a broken connection*
 - *Electricity stops flowing at the point where the connection was lost*

This symbol indicates a “high” potential, or the voltage of the power supply

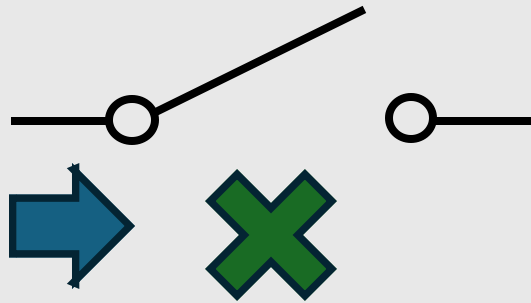


This symbol indicates a “low” or ground potential

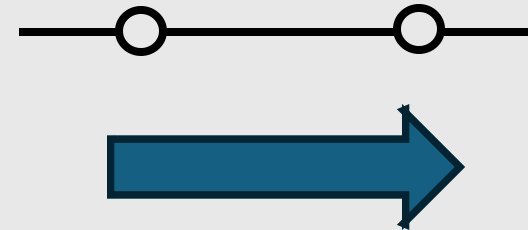


Switches

Open (off)

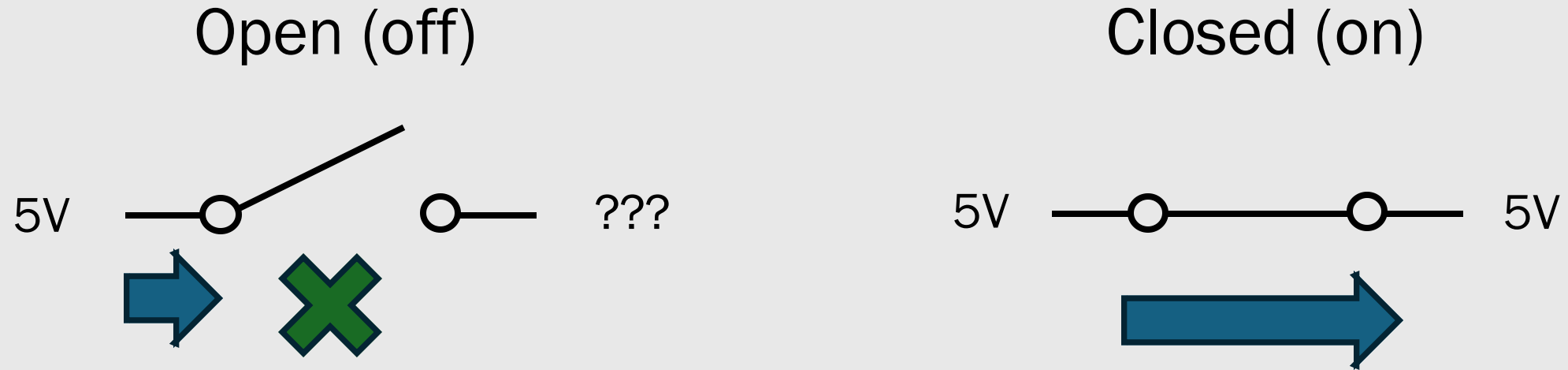


Closed (on)



Electricity can flow through a closed switch, it cannot flow through an open switch

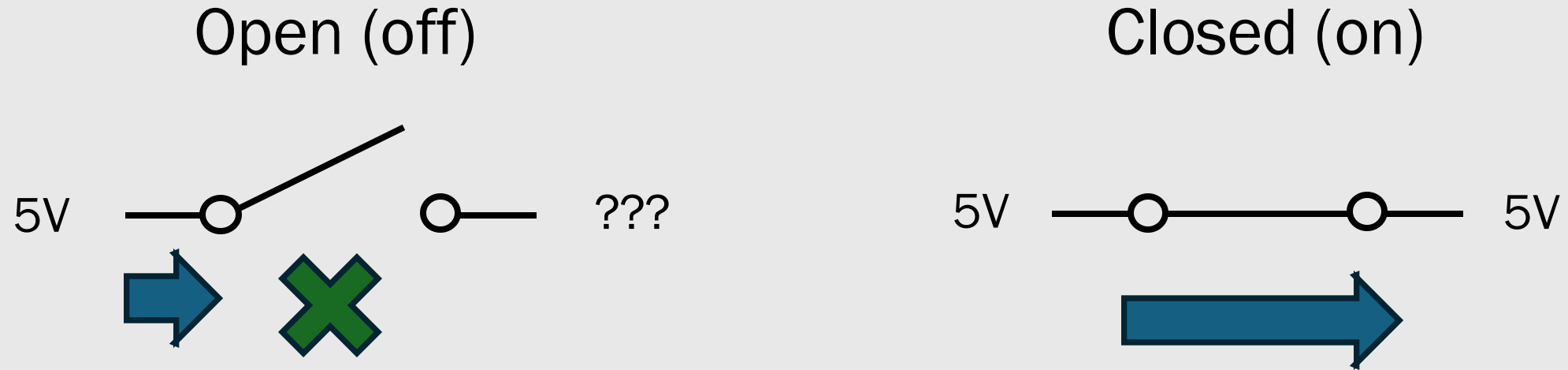
Switches



Let's say we apply 5V to the left side of the switch

- If the switch is open, we don't necessarily know the voltage on the right
 - Would need to see the full circuit to determine the voltage
- If the switch is closed, the voltage on the right is 5V

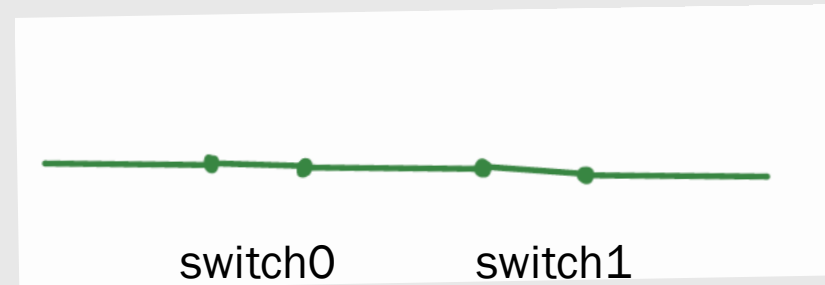
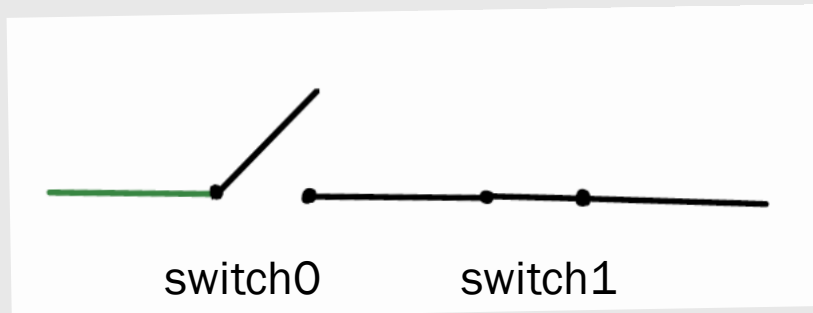
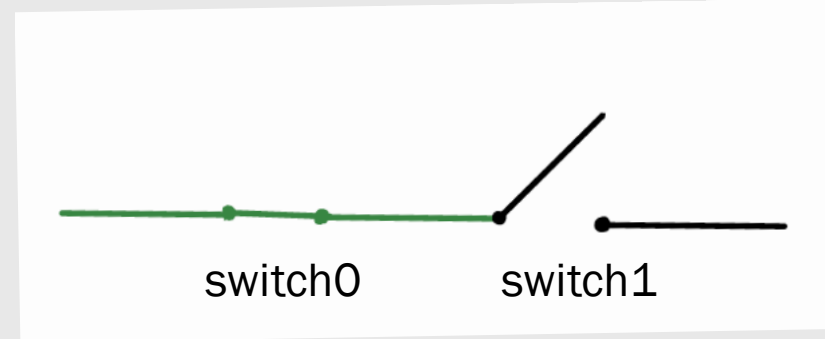
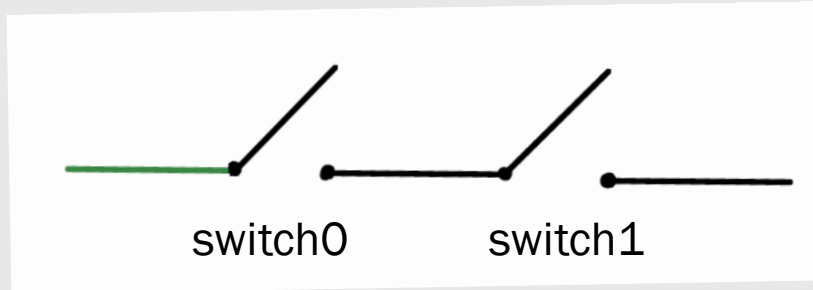
Switches



Let's say we apply 0V to the left side of the switch

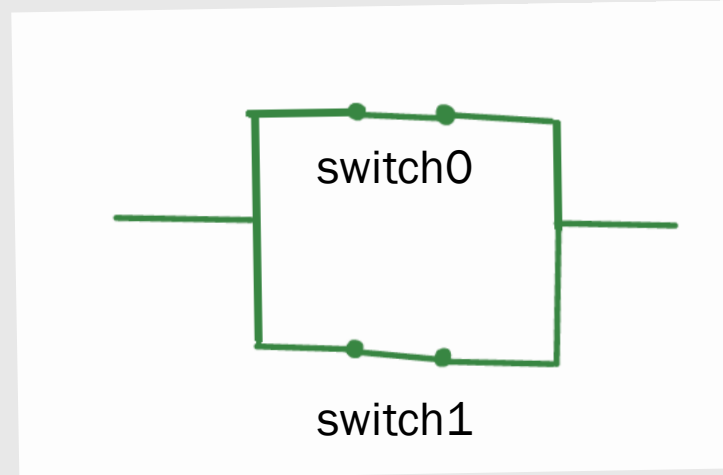
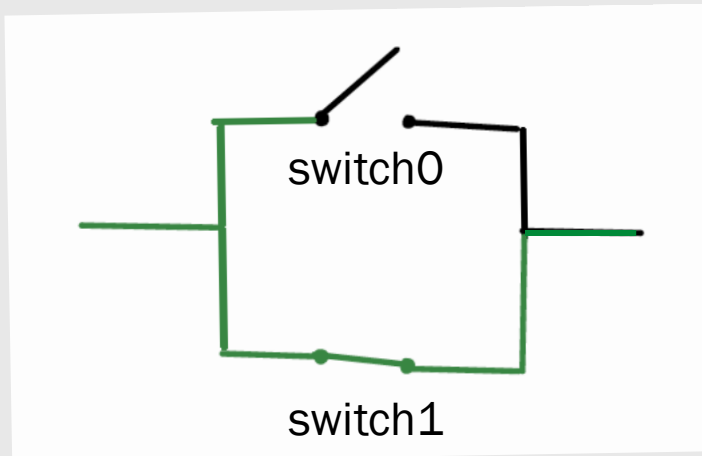
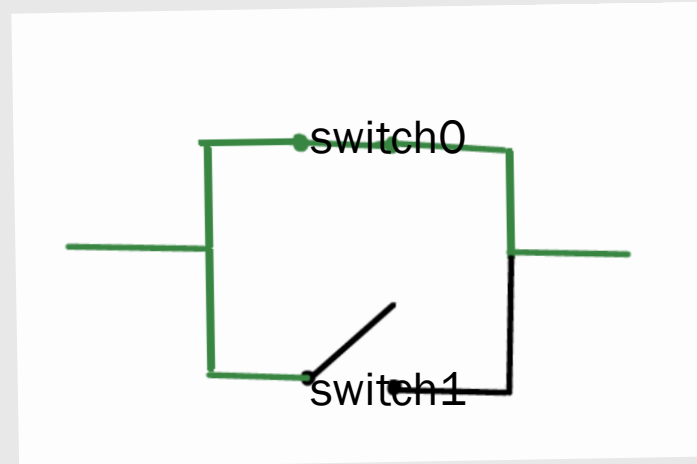
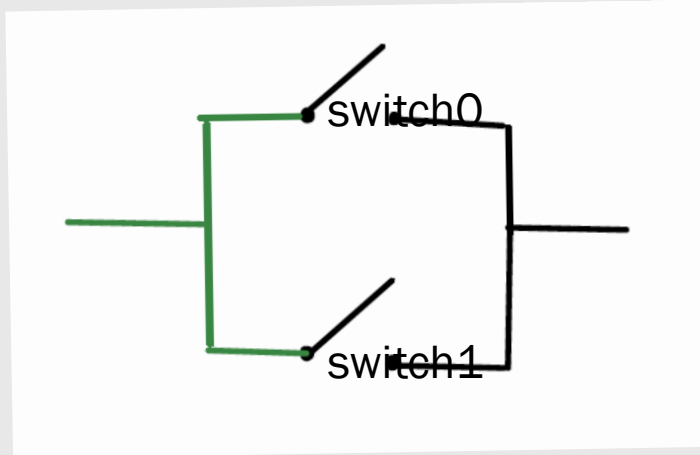
- If the switch is open, we don't necessarily know the voltage on the right
 - Would need to see the full circuit to determine the voltage
- If the switch is closed, the voltage on the right is 0V

Switches in Series



Current can only flow between the two ends if both switches are closed
 $\text{current_flow} = \text{switch}_0 \text{ AND } \text{switch}_1$

Switches in Parallel



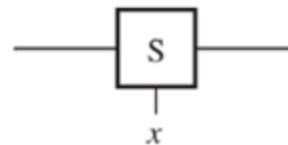
Current can flow between the two ends if either of the switches is closed
 $\text{current_flow} = \text{switch}_0 \text{ OR } \text{switch}_1$

Digital Variables and Functions

- A binary variable, 0 or 1, can represent the state of switch
 - *on or off, open or closed*
- The setting of switches can be used to turn on and off other switches to compose more complicated functions.



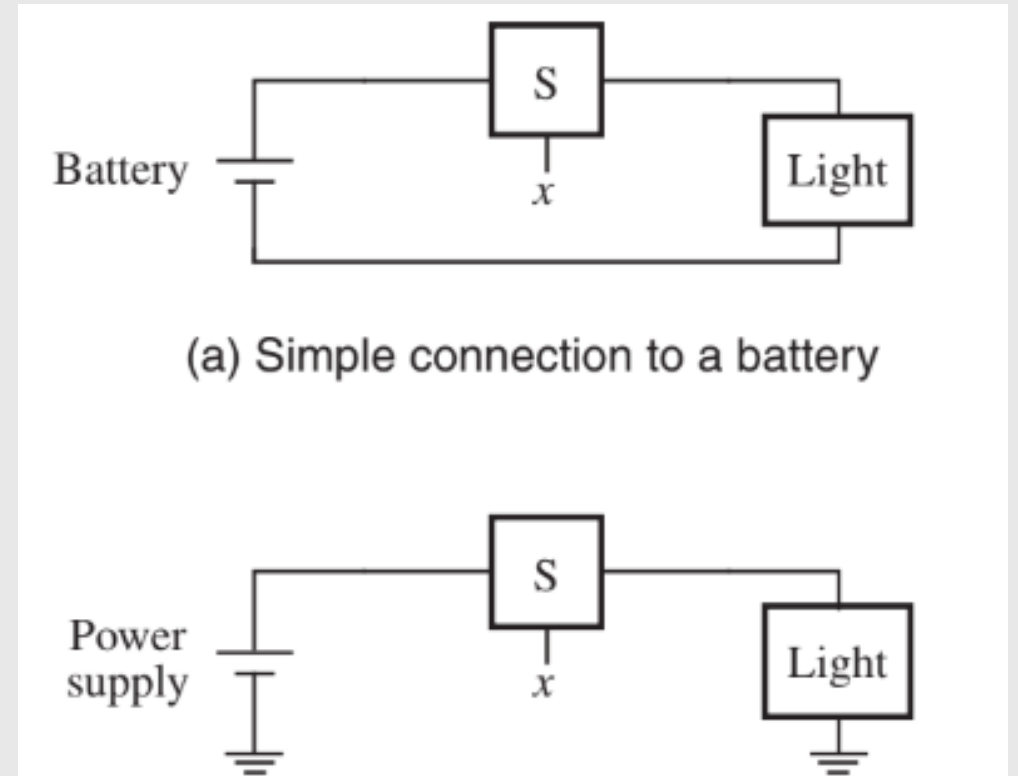
(a) Two states of a switch



(b) Symbol for a switch

A Simple Function: Turning on a Light

- By simply adding a power source and connecting a load (light bulb) our controlled switch can use a binary variable to control a light.
- This simple logic expression describes the output as a function of its inputs.
 - We say that $L(x) = x$ is a logic function and that x is an input variable x .

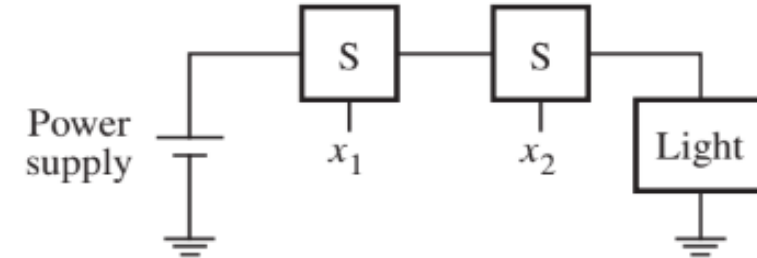


x	light
0	OFF
1	ON

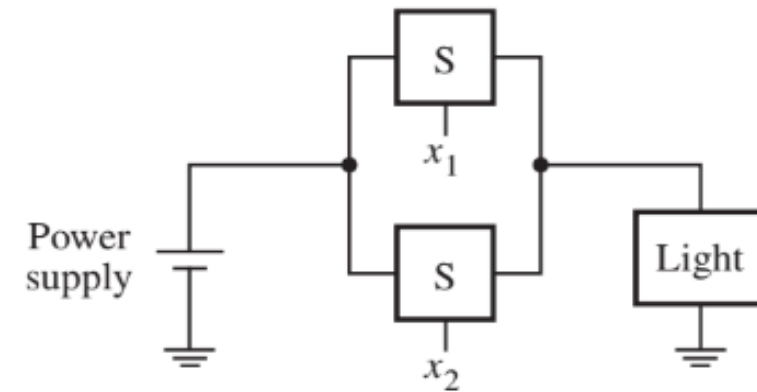
$$L(x) = x$$

Functions with Two Switches

- Two switches can be connected either in **series** or in **parallel**
- Using a **series** connection, the light will be turned on only if **both switches are closed**. If either switch is open, the light will be off.
 - $L(x_1, x_2) = x_1 \cdot x_2$
where $L = 1$ if $x_1 = 1$ and $x_2 = 1$, $L = 0$ otherwise.
- \cdot is logical AND



(a) The logical AND function (series connection)



(b) The logical OR function (parallel connection)

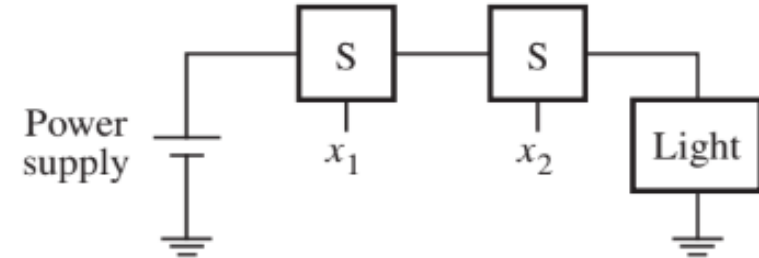
Two basic two-input logic functions

Functions with Two Switches

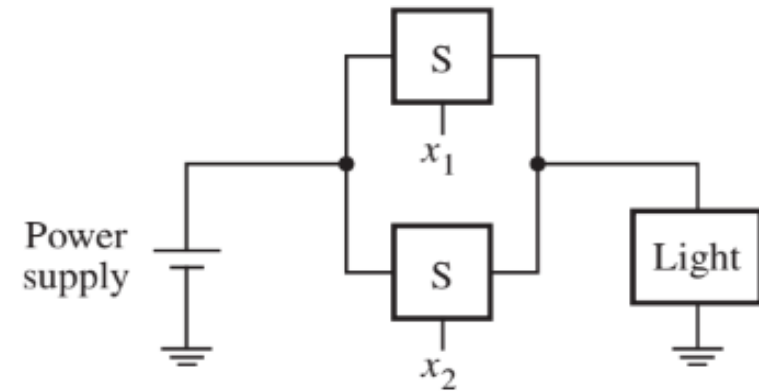
- With a **parallel** connection of the two switches, the light will be on if **either the x_1 or x_2 switch is closed**. The light will also be on if both switches are closed. **The light will be off only if both switches are open.**

- $L(x_1, x_2) = x_1 + x_2$
where $L = 1$ if $x_1 = 1$ or $x_2 = 1$ or if $x_1 = x_2 = 1$,
 $L = 0$ if $x_1 = x_2 = 0$.

- "+" is logical OR



(a) The logical AND function (series connection)



(b) The logical OR function (parallel connection)

Two basic two-input logic functions

A Three-Switch Function

- Three switches can be used to control the light in more complex ways. This series-parallel connection of switches realizes the logic function:
 - $L(x_1, x_2, x_3) = (x_1 + x_2) \cdot x_3$
- The light is on if $x_3 = 1$ and, at the same time, at least one of the x_1 or x_2 are 1.

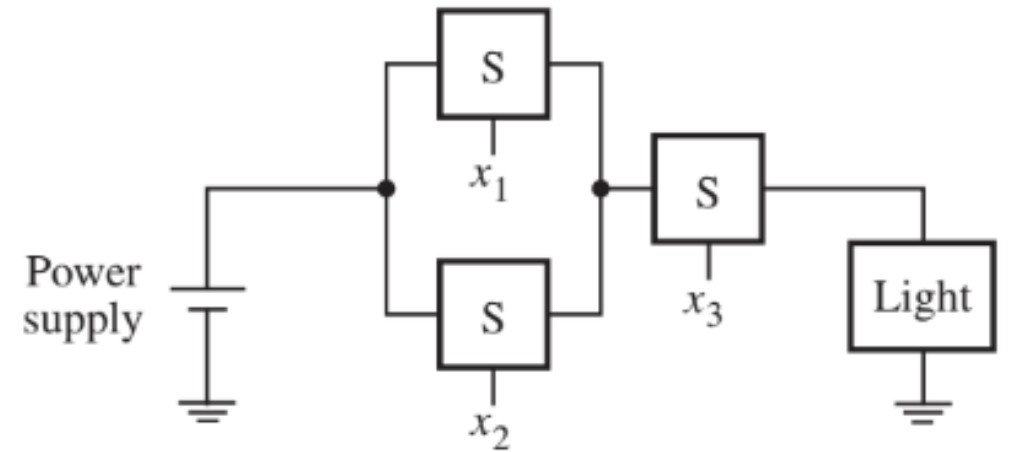


Figure 2.4 A series-parallel connection.

A Three-Switch Function

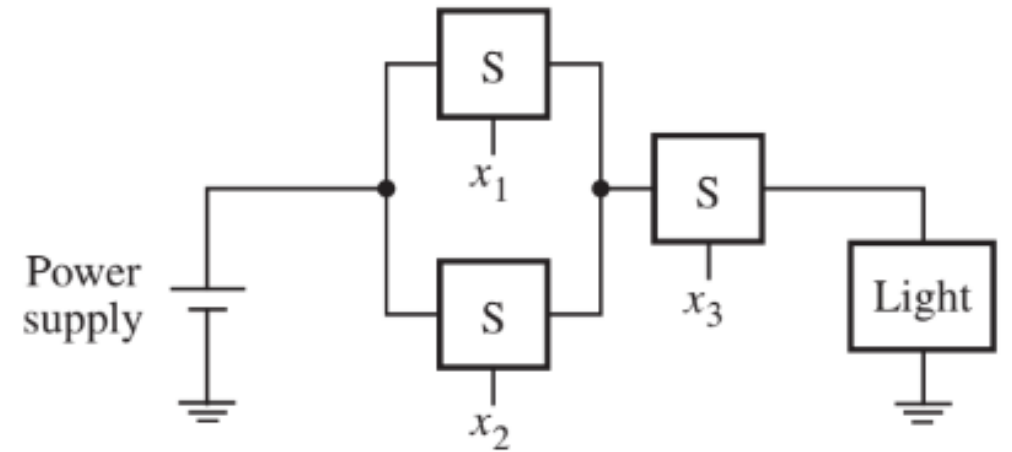


Figure 2.4 A series-parallel connection.

WS 2 Q4