Physical Bits: Transistors and Logic

- Encoding bits with voltages
- The "Digital" contract
- Digital processing elements
- Gates
- Transistors
- Building gates with transistors

\[ F = \text{XOR}(A, B) \]
WHERE ARE WE?

Things we know so far -
1) Computers process information
2) Information is measured in bits
3) Data can be represented as groups of bits
4) Computer instructions are encoded as bits
5) Computer instructions are just data
6) But, we don’t want to deal with details of bits…
   so we use ASSEMBLY Language
7) Even that is too low-level…
   So we use COMPILERS to generate
   assembly code, and assemblers to
   generate the final bits …

But, how are bits PROCESSED?
**A Substrate for Computation**

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

- neutrino flux
- trained elephants
- engraved stone tablets
- orbits of planets
- DNA sequences
- polarization of a photon

Wait! Some of those might have potential...
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
Representing Information with Voltages

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

- 0 volts: \(\text{BLACK}\)
- 1 volt: \(\text{WHITE}\)
- 0.37 volts: \(37\% \text{ Gray}\)
- 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:

- $v \rightarrow \text{Copy} \rightarrow v$
- $v \rightarrow \text{INV} \rightarrow 1-v$
Let's build a system!

input

Copy → INV

Copy → INV

Copy → INV

Copy → INV

(output)

(Reality)

output
Why Did Our System Fail?

Why doesn’t reality match theory?
1. COPY Operator doesn’t work right
2. INVERSION operator doesn’t work right
3. Theory is imperfect
4. Reality is imperfect
5. Our system architecture stinks

ANSWER: all of the above!

Noise and inaccuracy are inevitable; we can’t reliably reproduce infinite information-- we must design our system to tolerate some amount of error if it is to process information reliably.
The 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 robust...

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

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

Real World

Manufacturing Variations

Volts or Electrons or Ergs or Gallons

Noise

"Ideal" Abstract World

0/1

Bits

Keep in mind, the 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

---

Static Discipline

Output a “1” if at least 2 out of 3 of my inputs are a “1”. Otherwise, output “0”.

I will generate a valid output in no more than 2 minutes after seeing valid inputs.
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

No feedback (yet!)

But, in order to realize digital processing elements we have one more requirement!

A definition for a VALID input
and a VALID output!
Valid = Noise 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 ran

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 (\(T_p\)) after after inputs become valid).
**Digital Processing Elements**

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

1. **AND**: I will output a '1' if all my inputs are '1'.
   - Diagram:
     ```
     A --> Y
     ```

2. **OR**: I will output a '1' if any of my inputs are '1'.
   - Diagram:
     ```
     A --> Y
     B --> Y
     ```

3. **XOR**: I will only output a '1' if an odd number of my inputs are '1'.
   - Diagram:
     ```
     A --> Y
     B --> Y
     ```

4. **Buffer**: I will copy and restore my input to my output.
   - Diagram:
     ```
     A --> Y
     ```

5. **Inverter**: I will output the complement of my input.
   - Diagram:
     ```
     A --> Y
     ```

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

In honor of the richest man in the world we will henceforth refer to digital processing elements as “GATES”
How do we make Gates?

- A controllable switch is the common link of all computing technologies
- How do you control voltages with a switch?
- By creating and opening paths between higher and lower potentials

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

This symbol indicates a “low” or ground potential
**N-Channel Field-Effect Transistors (NFETs)**

Operating regions:

- **Cut-off:**
  \[ V_{GS} < V_{TH} \]
  \[ 0.8V \]

- **Linear:**
  \[ V_{GS} \geq V_{TH} \]
  \[ V_{DS} < V_{Dsat} \]

- **Saturation:**
  \[ V_{GS} \geq V_{TH} \]
  \[ V_{DS} \geq V_{Dsat} \]

When the gate voltage is "high", the switch closes. Good at pulling things "low".
**P-Channel Field-Effect Transistors (PFETs)**

Operating regions:

- **Cut-off:**
  \[ V_{GS} > V_{TH} \]

- **Linear:**
  \[ V_{GS} \leq V_{TH}, \quad V_{DS} > V_{Dsat} \]

- **Saturation:**
  \[ V_{GS} \leq V_{TH}, \quad V_{DS} \leq V_{Dsat} \]

When the gate voltage is "low", the switch closes. Good at pulling things "high".

Graphs showing the relationship between voltage and current for the different operating regions.
Using Transistors to Build Logic Gates!

Logic Gate recipe:

**pullup**: make this connection when $V_{IN}$ is near 0 so that $V_{OUT} = V_{DD}$

**pulldown**: make this connection when $V_{IN}$ is near $V_{DD}$ so that $V_{OUT} = 0$

We use PFETs here

and, NFETs here
CMOS INVERTER

Only a narrow range of input voltages result in “invalid” output values. This diagram is greatly exaggerated (the invalid input region is actually MUCH smaller)!
"Digital" Transistor Abstraction

- Transistors are extremely flexible, but fickle analog devices.
- If we limit how we use them, (i.e. adopt the following conventions), they can act as robust digital devices.
- Which we can treat as a simple switch abstraction.

![Diagram of N-channel and P-channel FETs]

**N-channel FET**, a 3-input device

- **Convention**: The D terminal of an N-FET *will* be connected to either ground or the D terminal of another N-FET.

**P-channel FET**, a 3-input device

- **Convention**: The S terminal of an N-FET *will* be connected to either the supply (the voltage representing "1") or the S terminal of another P-FET.

<table>
<thead>
<tr>
<th>State</th>
<th>N-FET</th>
<th>P-FET</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;0&quot;</td>
<td>G</td>
<td>D</td>
</tr>
<tr>
<td></td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>&quot;1&quot;</td>
<td>G</td>
<td>D</td>
</tr>
<tr>
<td></td>
<td>S</td>
<td>S</td>
</tr>
</tbody>
</table>
**Complementary Pullups and Pulldowns**

This is what the "C" in CMOS stands for!

We design components with *complementary* pullup and pulldown logic (i.e., the pulldown should be "on" when the pullup is "off" and vice versa).

<table>
<thead>
<tr>
<th>pullup</th>
<th>pulldown</th>
<th>F(I₁,…,Iₙ)</th>
</tr>
</thead>
<tbody>
<tr>
<td>on</td>
<td>off</td>
<td>driven “1”</td>
</tr>
<tr>
<td>off</td>
<td>on</td>
<td>driven “0”</td>
</tr>
<tr>
<td>on</td>
<td>on</td>
<td>driven “X”</td>
</tr>
<tr>
<td>off</td>
<td>off</td>
<td>no connection</td>
</tr>
</tbody>
</table>

**Convention:** In general, let's avoid these last two cases.

When they are used, the resulting device is not strictly following our STATIC DISCIPLINE (e.g., Pass gates and storage devices).

Such devices are only QUASI-DIGITAL!
What a nice $V_{OH}$ you have...

Thanks. It runs in the family...

On when $A$ is “1”

On when $A$ is “0”

Series N connections:  

Parallel N connections:

On when $A$ is “1” and $B$ is “1”: $A \cdot B$

On when $A$ is “0” or $B$ is “0”: $A + B$

On when $A$ is “1” or $B$ is “1”: $A + B$

On when $A$ is “0” and $B$ is “0”: $\overline{A + B}$

Parallel P connections:

Series P connections:
A TWO-INPUT LOGIC GATE

What function does this gate compute?

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
Here's Another...

What function does this gate compute?

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
**General CMOS Gate Recipe**

Step 1. Figure out pulldown network that does what you want (i.e. the set of conditions where the output is ‘0’)

\[ F = A \cdot (B + C) \]

Step 2. Walk the hierarchy replacing nfets with pfets, series subnets with parallel subnets, and parallel subnets with series subnets

Step 3. Combine pfet pullup network from Step 2 with nfet pulldown network from Step 1 to form fully-complementary CMOS gate.

But isn’t it hard to wire it all up?
**ONE LAST EXERCISE**

Let's construct a gate to compute:

\[ F = \overline{A+BC} = \text{NOT}(\text{OR}(A,\text{AND}(B,C))) \]

### Step 1: The pull-down network

### Step 2: The complementary pull-up network

**OBSERVATION:** CMOS gates tend to be **inverting**! Precisely, one or more "0" inputs are necessary to generate a "1" output, and one or more "1" inputs are necessary to generate a "0" output. Why?
Next time

Now that we can see what goes on inside of a single gate, we’ll next use several them to compose larger systems that compute other logic functions.

Android  NANDroid  NOTdroid  ORdroid