Agenda

- Wrap up the Comp Arch Mysteries lectures
  - Bloom Filters
  - Takeaways

- Discuss course logistics (very brief)
  - Slides will be available online

- Combinational Logic Circuits and Design
Required Lecture Video

- Why study computer architecture?
- Why is it important?
- Future Computing Architectures

Required Assignment
- Watch my inaugural lecture at ETH and understand it
  - https://www.youtube.com/watch?v=kgiZlSOcGFM

Optional Assignment
- Write a summary of the lecture
  - What are your key takeaways?
  - What did you learn?
  - What did you like or dislike?
Required Readings

- **This week**
  - Combinational Logic
    - P&P Chapter 3 until 3.3 + H&H Chapter 2
  - Hardware Description Languages and Verilog
    - H&H Chapter 4 until 4.3 and 4.5

- **Next week**
  - Sequential Logic
    - P&P Chapter 3.4 until end + H&H Chapter 3 in full
  - Hardware Description Languages and Verilog
    - H&H Chapter 4 in full

- Make sure you are done with
  - **P&P Chapters 1-3** + **H&H Chapters 1-4**
Recall: Bloom Filters
Recall: Approximate Set Membership

- Suppose you want to quickly find out:
  - whether an element belongs to a set

- And, you can tolerate mistakes of the sort:
  - The element is actually not in the set, but you are incorrectly told that it is → false positive

- But, you cannot tolerate mistakes of the sort:
  - The element is actually in the set, but you are incorrectly told that it is not → false negative

- Example task: You want to quickly identify all Mobile Phone Model X owners among all possible people in the world
  - Perhaps you want to give them free replacement phones
Recall: Example Task

- World population
  - ~8 billion (and growing)
  - 1 bit per person to indicate Model X owner or not
  - $2^{33}$ bits needed to represent the entire set accurately
    - 8 Gigabits $\rightarrow$ large storage cost, slow access

- Mobile Phone Model X owner population
  - Say 1 million (and growing)

Can we represent the Model X owner set approximately, using a much smaller number of bits?
  - Record the ID’s of owners in a much smaller Bloom Filter
Recall: Example Task II

- DRAM row population
  - \( \sim 8 \) billion (and growing)
  - 1 bit per row to indicate refresh often or not
  - \( 2^{33} \) bits needed to represent the entire set accurately
    - 8 Gigabits \( \rightarrow \) large storage cost, slow access

- Refresh-often population
  - Say 1 million

- Can we represent Refresh-often set approximately, using a much smaller number of bits?
  - Record the ID’s of Refresh-Often rows in a much smaller Bloom Filter
Recall: Bloom Filter

- [Bloom, CACM 1970]
- **Probabilistic data structure** that compactly represents set membership (presence or absence of element in a set)

- Non-approximate set membership: Use 1 bit per element to indicate absence/presence of each element from an element space of N elements

- **Approximate set membership**: use a much smaller number of bits and indicate each element’s presence/absence with a subset of those bits
  - Some elements map to the bits other elements also map to

- Operations: 1) insert, 2) test, 3) remove all elements

Bloom Filter Operation Example

Example with 64-128ms bin:

Hash function 1
Hash function 2
Hash function 3

Insert Row 1

Bloom Filter Operation Example

Example with 64-128ms bin:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
</table>

1 & 1 & 1 & =1

Hash function 1
Hash function 2
Hash function 3

Row 1 present? Yes
Bloom Filter Operation Example

Example with 64-128ms bin:

\[
\begin{array}{cccccccccccccccc}
0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]

Hash function 1
Hash function 2
Hash function 3

Row 2 present? No
Bloom Filter Operation Example

Example with 64-128ms bin:

```
0 0 1 0 1 1 0 0 0 1 0 0 1 0 1 0
```

Hash function 1

Hash function 2

Hash function 3

Insert Row 4
Bloom Filter Operation Example

Example with 64-128ms bin:

<table>
<thead>
<tr>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
</table>

Hash function 1

Hash function 2

Hash function 3

Row 5 present?
Yes (false positive)
Bloom Filters

Space/Time Trade-offs in Hash Coding with Allowable Errors

In such applications, it is envisaged that overall performance could be improved by using a smaller core resident hash area in conjunction with the new methods and, when necessary, by using some secondary and perhaps time-consuming test to "catch" the small fraction of errors associated with the new methods. An example is discussed which illustrates possible areas of application for the new methods.

Burton H. Bloom

In this paper trade-offs among certain computational factors in hash coding are analyzed. The paradigm problem considered is that of testing a series of messages one-by-one for membership in a given set of messages. Two new hash-coding methods are examined and compared with a particular conventional hash-coding method. The computational factors considered are the size of the hash area (space), the time required to identify a message as a nonmember of the given set (reject time), and an allowable error frequency.

Bloom Filters: Pros and Cons

- **Advantages**
  + Enables *storage-efficient* representation of set membership
  + Insertion and testing for set membership (presence) are **fast**
  + **No false negatives**: If Bloom Filter says an element is **not** present in the set, the element **must not** have been inserted
  + Enables *tradeoffs* between **time & storage efficiency & false positive rate** (via sizing and hashing)

- **Disadvantages**
  -- **False positives**: An element **may** be deemed to be present in the set by the Bloom Filter even though it **was not** inserted

  Not the right data structure when you cannot tolerate false positives

Benefits of Bloom Filters as Refresh Rate Bins

- **False positives**: a row may be declared present in the Bloom filter even if it was never inserted
  - **Not a problem**: Refresh some rows more frequently than needed

- **No false negatives**: rows are never refreshed less frequently than needed (no correctness problems)

- **Scalable**: a Bloom filter never overflows (unlike a fixed-size table) → You can keep inserting, and above properties are maintained (but false positive rate likely increases)

- **Efficient**: No need to store info on a per-row basis; simple hardware → 1.25 KB for 2 filters for 32 GB DRAM system

Reading: Use of Bloom Filters for Refresh

- Jamie Liu, Ben Jaiyen, Richard Veras, and Onur Mutlu, "RAIDR: Retention-Aware Intelligent DRAM Refresh"

RAIDR: Retention-Aware Intelligent DRAM Refresh

Jamie Liu    Ben Jaiyen    Richard Veras    Onur Mutlu
Carnegie Mellon University
{jamiel,bjaiyen,rveras,onur}@cmu.edu
Recap: Four Mysteries

- Meltdown & Spectre (2017-2018)
- Rowhammer (2012-2014)
- Memory Performance Attacks (2006-2007)
- Memories Forget: Refresh (2011-2012)
Takeaways
Some Takeaways

- It is an exciting time to be understanding and designing computing platforms

- Many challenging and exciting problems in platform design
  - That noone has tackled (or thought about) before
  - That can have huge impact on the world’s future

- Driven by huge hunger for data and its analysis (“Big Data”), new applications, ever-greater realism, ...
  - We can easily collect more data than we can analyze/understand

- Driven by significant difficulties in keeping up with that hunger at the technology layer
  - Three walls: Energy, reliability, complexity
As applications push boundaries, computing platforms will become increasingly strained.

**Dream**

and, they will come
Dream, and, They Will Come
Dream, and, They Will Come

Source: https://iq.intel.com/5-awesome-uses-for-drone-technology/
Increasingly Diverging/Complex Tradeoffs

Communication Dominates Arithmetic

Dally, HiPEAC 2015

- 64-bit DP 20pJ
- 256-bit buses
- 256-bit access 8 kB SRAM
- 26 pJ
- 256 pJ
- 500 pJ
- 16 nJ
- Efficient off-chip link
- 1 nJ

20mm
Increasingly Diverging/Complex Tradeoffs

Communication Dominates Arithmetic

A memory access consumes ~1000X the energy of a complex addition

- 64-bit DP: 20 pJ
- 256-bit buses
- 256-bit access: 8 kB SRAM
- 26 pJ
- 256 pJ
- 16 nJ
- 500 pJ
- Efficient off-chip link

Dally, HiPEAC 2015
Increasingly Complex Systems

Past systems

Microprocessor  Main Memory  Storage (SSD/HDD)
Increasingly Complex Systems

Modern systems

- FPGAs
- Hybrid Main Memory
- Persistent Memory/Storage
- (General Purpose) GPUs
- Heterogeneous Processors and Accelerators
- (General Purpose) GPUs
Recap: Some Goals of This Course

- **Teach/enable/empower you to:**
  - **Understand** how a processor works: principles & precedents
  - **Implement** a simple microprocessor from scratch on an FPGA
  - **Understand** how decisions made in hardware affect the software/programmer as well as hardware designer
  - **Think critically** (in solving problems)
  - **Think broadly** across the levels of transformation
  - **Understand** how to **analyze** and **make tradeoffs** in **design**
Course Info and Logistics
Course Info: Instructor

Onur Mutlu

- Professor @ ETH Zurich CS, since September 2015 (officially May 2016)
- Strecker Professor @ Carnegie Mellon University ECE/CS, 2009-2016, 2016-...
- PhD from UT-Austin, worked at Google, VMware, Microsoft Research, Intel, AMD
- https://people.inf.ethz.ch/omutlu/
- omutlu@gmail.com (Best way to reach me)
- Office hours: By appointment (email me)

Research and Teaching in:

- Computer architecture, computer systems, bioinformatics
- Memory and storage systems
- Hardware security
- Fault tolerance
- Hardware/software cooperation
- ...
Course Info: PhD Assistants (I)

- Head Assistant
  - Dr. Juan Gomez Luna

- Vice-Head Assistant
  - Hasan Hassan

- (Other) Key Assistants and Guest Lecturers
  - Dr. Arash Tavakkol
  - Jeremie Kim
  - Minesh Patel
  - Giray Yaglikci
Course Info: PhD Assistants (II)

- (Other) Key Assistants
  - Aritra Dhar
  - Daniele Lain
  - David Sommer
  - Francois Serre
  - Mridula Singh
  - Patrick Leu
Course Info: Student Assistants

- Giuseppe Arcuti
- Alexander Breuss
- Salomon Brülisauer
- Josua Cantieni
- Sven Gregorio
- Lukas Gygi
- Leo Horne
- Chris Mnuk
- Johannes Schenk
- Alexander Wälchli
- Patrick Ziegler
- Marco Zeller
Course Info: Lab Assistants

- **Tuesday 15-17**
  - Hasan Hassan
  - Giray Yaglikci
  - David Sommer
  - Daniele Lain
  - Giuseppe Arcuti
  - Salomon Brülisauer
  - Sven Gregorio
  - Lukas Gygi
  - Johannes Schenk
  - Alexander Wälchli
  - Patrick Ziegler

- **Wednesday 15-17**
  - Minesh Patel
  - Patrick Leu
  - Salomon Brülisauer
  - Sven Gregorio
  - Lukas Gygi
  - Leo Horne
  - Patrick Ziegler
Course Info: Lab Assistants

- **Friday 8-10**
  - Juan Gomez Luna
  - Arash Tavakkol
  - Francois Serre
  - Giuseppe Arcuti
  - Josua Cantieni
  - Chris Mnuk
  - Johannes Schenk
  - Marco Zeller

- **Friday 10-12**
  - Jeremie Kim
  - Mridula Singh
  - Aritra Dhar
  - Giuseppe Arcuti
  - Alexander Breuss
  - Josua Cantieni
  - Chris Mnuk
  - Marco Zeller
If You Need Help

- Post your question on Moodle
  - Preferred for technical questions

- Write an e-mail to:
  - digitaltechnik@lists.inf.ethz.ch
  - The instructor and all assistants will receive this e-mail
Where to Get Up-to-date Course Info?

- Website:
  - [https://safari.ethz.ch/digitaltechnik/](https://safari.ethz.ch/digitaltechnik/)
  - Lecture slides and videos
  - Readings
  - Lab information
  - Course schedule, handouts, FAQs
  - Software
  - Plus other useful information for the course
  - Check frequently for announcements and due dates
  - This is your single point of access to all resources

- Your ETH Email
- Lecturers and Teaching Assistants
Lecture and Lab Times and Policies

- Lectures:
  - Thursday and Fridays, 13:15-15:00
  - HG F7 (F5 overflow)
  - Attendance is for your benefit and is therefore important
  - Some days, we will have guest lectures and exercise sessions

- Lab sessions:
  - See online
  - You should definitely attend the lab sessions
  - Labs started this week (March 6 onwards)
  - Lab information and handouts are here:
A Note on Hardware vs. Software

- This course might seem like it is only “Computer Hardware”

- However, you will be much more capable if you master both hardware and software (and the interface between them)
  - Can develop better software if you understand the hardware
  - Can design better hardware if you understand the software
  - Can design a better computing system if you understand both

- This course covers the HW/SW interface and microarchitecture
  - We will focus on tradeoffs and how they affect software

- Recall the four mysteries
- Recall the Transmeta story
What Do I Expect From You?

- **Required background:** Binary numbers/arithmetic, reading material week 1, enthusiasm to learn & think, common sense

- Learn the material thoroughly
  - attend lectures, do the readings, do the exercises, do the labs

- **Work hard:** this will be a hard but fun & informative course

- Ask questions, take notes, participate

- Perform the assigned readings

- **Come to class on time**

- Start early – do not procrastinate

- If you want feedback, come to office hours

- Remember “**Chance favors the prepared mind.**” (Pasteur)
What Do I Expect From You?

- How you prepare and manage your time is very important

- There will be 9 lab assignments
  - They will take time
  - Start early, work hard

- This will be a heavy course
  - However, you will learn a lot of fascinating topics and understand how a microprocessor actually works from the ground up
  - And, it will hopefully change how you look at and think about designs around you
How Will You Be Evaluated?

- **Labs: 30%**
  - Lecture 3 covered what you will do

  **Goal:**
  
  *By the end of the labs (with a little help) design your own processor and make it work!*

- **Examination: 70%**
  - 150-minute exam within the exam period
    - Scheduled by the school, we have no influence on the exam time.
  - Questions related to both labs and lectures
  - Six pages of handwritten notes are allowed
  - Previous exams available on course webpage and via VIS
    - [https://www.vis.ethz.ch/de/services/examcollection/Digitaltechnik/](https://www.vis.ethz.ch/de/services/examcollection/Digitaltechnik/)
Computer Architecture as an Enabler of the Future
Why study computer architecture?

Why is it important?

Future Computing Architectures

Required Assignment

- Watch my inaugural lecture at ETH and understand it
- https://www.youtube.com/watch?v=kgiZlISOcGFM

Optional Assignment

- Write a summary of the lecture
  - What are your key takeaways?
  - What did you learn?
  - What did you like or dislike?
... but, first ...

- Let’s understand the fundamentals...

- You can change the world only if you understand it well enough...
  - Especially the basics (fundamentals)
  - Past and present dominant paradigms
  - And, their advantages and shortcomings – tradeoffs
  - And, what remains fundamental across generations
  - And, what techniques you can use and develop to solve problems
Fundamental Concepts
What is A Computer?

- Three key components
- Computation
- Communication
- Storage (memory)
What is A Computer?

- We will cover all three components

- Processing
  - control (sequencing)
  - datapath

- Memory
  - program and data

- I/O
What We Will Cover (I)

- Combinational Logic Design
- Hardware Description Languages (Verilog)
- Sequential Logic Design
- Timing and Verification
- ISA (MIPS)
- MIPS Assembly Programming
What We Will Cover (II)

- Microarchitecture Basics: Single-cycle
- Multi-cycle and Microprogrammed Microarchitectures
- Pipelining
- Issues in Pipelining: Control & Data Dependence Handling, State Maintenance and Recovery, ...
- Out-of-Order Execution
- Other Processing Paradigms (SIMD, VLIW, Systolic, ...)
- Memory and Caches
Processing Paradigms We Will Cover

- Pipelining
- Out-of-order execution
- Dataflow (at the ISA level)
- Superscalar Execution
- VLIW
- SIMD Processing (Vector and array processors, GPUs)
- Decoupled Access Execute
- Systolic Arrays
Combinational Logic Circuits and Design
What We Will Learn Today?

- Building blocks of modern computers
  - Transistors
  - Logic gates

- Boolean algebra

- Combinational circuits

- How to use Boolean algebra to represent combinational circuits

- Minimizing logic circuits (if time permits)
(Micro)-Processors
FPGAs
Custom ASICs
They All Look the Same

<table>
<thead>
<tr>
<th>Microprocessors</th>
<th>FPGAs</th>
<th>ASICs</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="Microprocessor" /></td>
<td><img src="image2.png" alt="FPGA" /></td>
<td><img src="image3.png" alt="ASIC" /></td>
</tr>
</tbody>
</table>

**In short:**
- Microprocessors: Common building block of computers
- FPGAs: Reconfigurable hardware, flexible
- ASICs: You customize everything
# They All Look the Same

<table>
<thead>
<tr>
<th></th>
<th>Microprocessors</th>
<th>FPGAs</th>
<th>ASICs</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>In short:</strong></td>
<td>Common building block of computers</td>
<td>Reconfigurable hardware, flexible</td>
<td>You customize everything</td>
</tr>
<tr>
<td><strong>Program Development Time</strong></td>
<td>minutes</td>
<td>days</td>
<td>months</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## They All Look the Same

<table>
<thead>
<tr>
<th>Microprocessors</th>
<th>FPGAs</th>
<th>ASICs</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="Image" /></td>
<td><img src="image2.png" alt="Image" /></td>
<td><img src="image3.png" alt="Image" /></td>
</tr>
</tbody>
</table>

**In short:**
- **Microprocessors**: Common building block of computers
- **FPGAs**: Reconfigurable hardware, flexible
- **ASICs**: You customize everything

<table>
<thead>
<tr>
<th>Program Development Time</th>
<th>minutes</th>
<th>days</th>
<th>months</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Performance</strong></td>
<td>0</td>
<td>+</td>
<td>++</td>
</tr>
</tbody>
</table>

---

**SAFARI**
They All Look the Same

<table>
<thead>
<tr>
<th>Microprocessors</th>
<th>FPGAs</th>
<th>ASICS</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="Microprocessor" /></td>
<td><img src="image2.png" alt="FPGA" /></td>
<td><img src="image3.png" alt="ASIC" /></td>
</tr>
</tbody>
</table>

**In short:**
- Microprocessors: Common building block of computers
- FPGAs: Reconfigurable hardware, flexible
- ASICS: You customize everything

**Program Development Time**
- Microprocessors: minutes
- FPGAs: days
- ASICS: months

**Performance**
- Microprocessors: 0
- FPGAs: +
- ASICS: ++

**Good for**
- Microprocessors: Ubiquitous, Simple to use
- FPGAs: Prototyping, Small volume
- ASICS: Mass production, Max performance
## They All Look the Same

<table>
<thead>
<tr>
<th>Microprocessors</th>
<th>FPGAs</th>
<th>ASICs</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="Microprocessors" /></td>
<td><img src="image2.png" alt="FPGAs" /></td>
<td><img src="image3.png" alt="ASICs" /></td>
</tr>
</tbody>
</table>

### In short:
- **Microprocessors**: Common building block of computers
- **FPGAs**: Reconfigurable hardware, flexible
- **ASICs**: You customize everything

<table>
<thead>
<tr>
<th>Program Development Time</th>
<th>minutes</th>
<th>days</th>
<th>months</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>In short:</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Program Development Time</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Performance</th>
<th>0</th>
<th>+</th>
<th>++</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>In short:</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Good for</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Programming</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Languages</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Main Companies</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Performance**: 0
- **Good for**: Ubiquitous, Simple to use
- **Programming**: Executable file
- **Languages**: C/C++/Java/…
- **Main Companies**: Intel, ARM

- **Program Development Time**: minutes
- **Good for**: Prototyping, Small volume
- **Programming**: Bit file
- **Languages**: Verilog/VHDL
- **Main Companies**: Xilinx, Altera, Lattice

- **Program Development Time**: days
- **Good for**: Mass production, Max performance
- **Programming**: Design masks
- **Languages**: Verilog/VHDL
- **Main Companies**: TSMC, UMC, ST, Globalfoundries
## They All Look the Same

<table>
<thead>
<tr>
<th>Microprocessors</th>
<th>FPGAs</th>
<th>ASICs</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>In short:</strong></td>
<td>Common building block of computers</td>
<td>Reconfigurable hardware, flexible</td>
</tr>
</tbody>
</table>

### Comparison Table

<table>
<thead>
<tr>
<th>Feature</th>
<th>Microprocessors</th>
<th>FPGAs</th>
<th>ASICs</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Program Development Time</strong></td>
<td>minutes</td>
<td>days</td>
<td>months</td>
</tr>
<tr>
<td><strong>Performance</strong></td>
<td>0</td>
<td>+</td>
<td>++</td>
</tr>
<tr>
<td><strong>Good for</strong></td>
<td>Ubiquitous</td>
<td>Prototyping</td>
<td>Mass production</td>
</tr>
<tr>
<td><strong>Simple to use</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Programming</strong></td>
<td>Executable file</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Languages</strong></td>
<td>C/C++/Java/…</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Main Companies</strong></td>
<td>Intel, ARM</td>
<td>Xilinx, Altera, Lattice</td>
<td>TSMC, UMC, ST, Globalfoundries</td>
</tr>
</tbody>
</table>

*Using this language*

**Verilog/VHDL**

Want to learn how these work

By programming these
Building Blocks of Modern Computers
Transistors
Computers are built from very large numbers of very simple structures

- Intel’s Pentium IV microprocessor, first offered for sale in 2000, was made up of more than 42 million MOS transistors
- Intel’s Core i7 Broadwell-E, offered for sale in 2016, is made up of more than 3.2 billion MOS transistors

This lecture

- How the MOS transistor works (as a logic element)
- How these transistors are connected to form logic gates
- How logic gates are interconnected to form larger units that are needed to construct a computer
MOS Transistor

- By combining
  - Conductors (Metal)
  - Insulators (Oxide)
  - Semiconductors

We get a Transistor (MOS)

- Why is this useful?
  - We can combine many of these to realize simple logic gates
  - The electrical properties of metal-oxide semiconductors are well beyond the scope of what we want to understand in this course
  - They are below our lowest level of abstraction
Different Types of MOS Transistors

- There are two types of MOS transistors: n-type and p-type.

- They both operate “logically,” very similar to the way wall switches work.
In order for the lamp to glow, electrons must flow.

In order for electrons to flow, there must be a closed circuit from the power supply to the lamp and back to the power supply.

The lamp can be turned on and off by simply manipulating the wall switch to make or break the closed circuit.
Instead of the wall switch, we could use an **n-type** or a **p-type** MOS transistor to make or break the closed circuit.

- **If the gate of an n-type transistor is supplied with a high voltage,** the connection from source to drain acts like a piece of wire.

- **Depending on the technology, 0.7V to 3V**

- **If the gate of the n-type transistor is supplied with 0V,** the connection between the source and drain is broken.
How Does a Transistor Work?

- The **n-type** transistor in a circuit with a battery and a bulb...

- The **p-type** transistor works in exactly the opposite fashion from the **n-type** transistor.

The circuit is closed when the gate is supplied with 3V

The circuit is closed when the gate is supplied with 0V
Logic Gates
Now, we know how a MOS transistor works

How do we build logic out of MOS transistors?

We construct basic logic structures out of individual MOS transistors

These logical units are named logic gates
  - They implement simple Boolean functions
Modern computers use both n-type and p-type transistor, i.e. Complementary MOS (CMOS) technology

\[ \text{nMOS} + \text{pMOS} = \text{CMOS} \]

The simplest logic structure that exists in a modern computer

What does this circuit do?
Functionality of Our CMOS Circuit

What happens when the input is connected to 0V?

p-type transistor pulls the output up

Y = 3V
Functionality of Our CMOS Circuit

What happens when the input is connected to 3V?

- The n-type transistor pulls the output down to 0V.

n-type transistor pulls the output down
CMOS NOT Gate

- This is actually the **CMOS NOT Gate**
- **Why do we call it NOT?**
  - If A = 0V then Y = 3V
  - If A = 3V then Y = 0V
- **Digital circuit:** one possible interpretation
  - Interpret 0V as logical (binary) 0 value
  - Interpret 3V as logical (binary) 1 value

<table>
<thead>
<tr>
<th>A</th>
<th>P</th>
<th>N</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ON</td>
<td>OFF</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>OFF</td>
<td>ON</td>
<td>0</td>
</tr>
</tbody>
</table>

\[ Y = \bar{A} \]
**CMOS NOT Gate**

- This is actually the CMOS NOT Gate

- **Why do we call it NOT?**
  - If $A = 0V$ then $Y = 3V$
  - If $A = 3V$ then $Y = 0V$

- **Digital circuit:** one possible interpretation
  - Interpret $0V$ as logical (binary) 0 value
  - Interpret $3V$ as logical (binary) 1 value

---

![Diagrams](image)

We call it a **NOT gate** or an **inverter**

**Truth table:** what would be the logical output of the circuit for each possible input

<table>
<thead>
<tr>
<th>$A$</th>
<th>$Y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
CMOS NAND Gate

- Let’s build more complex gates!
CMOS NAND Gate

- Let’s build more complex gates!

P1 and P2 are in parallel; only one must be ON to pull the output up to 3V

N1 and N2 are connected in series; both must be ON to pull the output to 0V

\[
Y = \overline{A \cdot B} = \overline{AB}
\]

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>P1</td>
<td>P2</td>
<td>N1</td>
<td>N2</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>ON</td>
<td>ON</td>
<td>OFF</td>
<td>OFF</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>ON</td>
<td>OFF</td>
<td>OFF</td>
<td>ON</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>OFF</td>
<td>ON</td>
<td>ON</td>
<td>OFF</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>OFF</td>
<td>OFF</td>
<td>ON</td>
<td>ON</td>
</tr>
</tbody>
</table>
CMOS NAND Gate

Let’s build more complex gates!

\[ Y = \overline{A \cdot B} = \overline{AB} \]

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>
CMOS AND Gate

How can we make an AND gate?

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

We make an **AND** gate using one NAND and one NOT

\[ Y = A \cdot B = AB \]
CMOS NOT, NAND, AND Gates

A \rightarrow Y

\begin{array}{c|c}
A & Y \\
\hline
0 & 1 \\
1 & 0 \\
\end{array}

A \rightarrow Y

\begin{array}{c|c|c}
A & B & Y \\
\hline
0 & 0 & 1 \\
0 & 1 & 1 \\
1 & 0 & 1 \\
1 & 1 & 0 \\
\end{array}

A \rightarrow Y

\begin{array}{c|c|c}
A & B & Y \\
\hline
0 & 0 & 0 \\
0 & 1 & 0 \\
1 & 0 & 0 \\
1 & 1 & 1 \\
\end{array}

3V \rightarrow 0V

Out (Y)

In (A)

P

N

0V

P2

N1

P1

N2

P3

N3

Out (Y)

In (A)

N1

In (B)

N2

In (A)

N1

In (B)

N2

Out (Y)

In (A)

N3

In (B)

N2

Out (Y)

In (A)

N1

In (B)

N2

Out (Y)

In (A)

N3

In (B)

N2

Out (Y)

In (A)

N3

In (B)

N2

Out (Y)

In (A)

N3

In (B)

N2

Out (Y)

In (A)

N3
General CMOS Gate Structure

- The general form used to construct any inverting logic gate, such as: NOT, NAND, or NOR
  - The networks may consist of transistors in series or in parallel
  - When transistors are in parallel, the network is **ON** if one of the transistors is **ON**
  - When transistors are in series, the network is **ON** only if all transistors are **ON**
# Common Logic Gates

<table>
<thead>
<tr>
<th>Buffer</th>
<th>AND</th>
<th>OR</th>
<th>XOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>![Buffer Symbol]</td>
<td>![AND Symbol]</td>
<td>![OR Symbol]</td>
<td>![XOR Symbol]</td>
</tr>
<tr>
<td><strong>A</strong>&lt;br&gt;0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><strong>Z</strong>&lt;br&gt;0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Inverter</th>
<th>NAND</th>
<th>NOR</th>
<th>XNOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>![Inverter Symbol]</td>
<td>![NAND Symbol]</td>
<td>![NOR Symbol]</td>
<td>![XNOR Symbol]</td>
</tr>
<tr>
<td><strong>A</strong>&lt;br&gt;0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td><strong>Z</strong>&lt;br&gt;0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Larger Gates

- We can extend the gates to more than 2 inputs
- Example: 3-input AND gate, 10-input NOR gate
- See your readings
Aside: Moore’s Law: Enabler of Many Gates on a Chip
An Enabler: Moore’s Law

Moore, “Cramming more components onto integrated circuits,” Electronics Magazine, 1965. Component counts double every other year
Number of transistors on an integrated circuit doubles ~ every two years

Moore's Law – The number of transistors on integrated circuit chips (1971-2016)

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important as other aspects of technological progress – such as processing speed or the price of electronic products – are strongly linked to Moore's law.

---

The data visualization is available at OurWorldinData.org. There you find more visualizations and research on this topic. Licensed under CC-BY-SA by the author Max Roser.
Recommended Reading


- Only 3 pages

- A quote:

  "With unit cost falling as the number of components per circuit rises, by 1975 economics may dictate squeezing as many as 65 000 components on a single silicon chip."

- Another quote:

  "Will it be possible to remove the heat generated by tens of thousands of components in a single silicon chip?"
How Do We Keep Moore’s Law

- **Manufacturing smaller transistors/structures**
  - Some structures are already a few atoms in size

- **Developing materials with better properties**
  - Copper instead of Aluminum (better conductor)
  - Hafnium Oxide, air for Insulators
  - Making sure all materials are compatible is the challenge

- **Optimizing the manufacturing steps**
  - How to use 193nm ultraviolet light to pattern 20nm structures

- **New technologies**
  - FinFET, Gate All Around transistor, Single Electron Transistor...
Combinational Logic Circuits
We Can Now Build Logic Circuits

Now, we understand the workings of the basic logic gates

What is our next step?

Build some of the logic structures that are important components of the microarchitecture of a computer!

- A logic circuit is composed of:
  - Inputs
  - Outputs

- **Functional specification** (describes relationship between inputs and outputs)

- **Timing specification** (describes the delay between inputs changing and outputs responding)
Types of Logic Circuits

- **Combinational Logic**
  - Memoryless
  - Outputs are strictly dependent on the combination of input values that are being applied to circuit *right now*
  - In some books called Combinatorial Logic

- **Later we will learn: Sequential Logic**
  - Has memory
    - Structure stores history → Can “store” data values
  - Outputs are determined by previous (historical) and current values of inputs
Boolean Equations
Functional Specification

- **Functional specification** of outputs in terms of inputs
- What do you mean by “function”?
  - Unique **mapping** from input values to output values
  - The **same** input values produce the **same** output value every time
  - **No memory** (does not depend on the history of input values)

- **Example (full 1-bit adder – more later):**

\[
S = F(A, B, C_{\text{in}}) \\
C_{\text{out}} = G(A, B, C_{\text{in}})
\]

\[
S = A \oplus B \oplus C_{\text{in}} \\
C_{\text{out}} = AB + AC_{\text{in}} + BC_{\text{in}}
\]
Simple Equations: NOT / AND / OR

\[ \overline{A} \text{ (reads “not A”)} \text{ is 1 iff } A \text{ is 0} \]

\[ \begin{array}{c|c|c}
A & \overline{A} \\
0 & 1 \\
1 & 0 \\
\end{array} \]

\[ A \cdot B \text{ (reads “A and B”)} \text{ is 1 iff } A \text{ and } B \text{ are both 1} \]

\[ \begin{array}{c|c|c|c}
A & B & A \cdot B \\
0 & 0 & 0 \\
0 & 1 & 0 \\
1 & 0 & 0 \\
1 & 1 & 1 \\
\end{array} \]

\[ A + B \text{ (reads “A or B”)} \text{ is 1 iff either } A \text{ or } B \text{ is 1} \]

\[ \begin{array}{c|c|c|c}
A & B & A + B \\
0 & 0 & 0 \\
0 & 1 & 1 \\
1 & 0 & 1 \\
1 & 1 & 1 \\
\end{array} \]
Boolean Algebra: Big Picture

- An algebra on 1’s and 0’s
  - with AND, OR, NOT operations

- What you start with
  - **Axioms:** basic things about objects and operations you just assume to be true at the start

- What you derive first
  - **Laws and theorems:** allow you to manipulate Boolean expressions
  - ...also allow us to do some simplification on Boolean expressions

- What you derive later
  - More “sophisticated” properties useful for manipulating digital designs represented in the form of Boolean equations

George Boole, “The Mathematical Analysis of Logic,” 1847.
## Boolean Algebra: Axioms

### Formal version

1. **B** contains at least two elements, 0 and 1, such that $0 \neq 1$

2. **Closure** $a, b \in B$,
   - (i) $a + b \in B$
   - (ii) $a \cdot b \in B$

3. **Commutative Laws**: $a, b \in B$,
   - (i)
   - (ii)

4. **Identities**: $0, 1 \in B$
   - (i)
   - (ii)

5. **Distributive Laws**:
   - (i)
   - (ii)

6. **Complement**:
   - (i)
   - (ii)

### English version

1. Math formality...

2. Result of AND, OR stays in set you start with

3. For primitive AND, OR of 2 inputs, order doesn’t matter

4. There are identity elements for AND, OR, that give you back what you started with

5. • distributes over $+$, just like algebra

   …but $+$ distributes over $\cdot$, also (!!)

6. There is a complement element;

   AND/ORing with it gives the identity elm.
Boolean Algebra: Duality

- **Observation**
  - All the axioms come in “dual” form
  - Anything true for an expression also true for its dual
  - So any derivation you could make that is true, can be flipped into dual form, and it stays true

- **Duality — More formally**
  - A dual of a Boolean expression is derived by replacing
    - Every AND operation with... an OR operation
    - Every OR operation with... an AND
    - Every constant 1 with... a constant 0
    - Every constant 0 with... a constant 1
    - But don’t change any of the literals or play with the complements!

**Example**

\[ a \cdot (b + c) = (a \cdot b) + (a \cdot c) \]

\[ \rightarrow a + (b \cdot c) = (a + b) \cdot (a + c) \]
Boolean Algebra: Useful Laws

**Operations with 0 and 1:**
1. \( X + 0 = X \)
2. \( X + 1 = 1 \)

\[ \begin{align*}
1D. & \quad X \cdot 1 = X \\
2D. & \quad X \cdot 0 = 0 \\
\end{align*} \]

**Idempotent Law:**
3. \( X + X = X \)

\[ 3D. \quad X \cdot X = X \]

**Involution Law:**
4. \( \overline{X} = X \)

**Laws of Complementarity:**
5. \( X + \overline{X} = 1 \)

\[ 5D. \quad X \cdot \overline{X} = 0 \]

**Commutative Law:**
6. \( X + Y = Y + X \)

\[ 6D. \quad X \cdot Y = Y \cdot X \]

AND, OR with identities gives you back the original variable or the identity

AND, OR with self = self
double complement = no complement
AND, OR with complement gives you an identity

Just an axiom…
Useful Laws (cont)

**Associative Laws:**
7. \((X + Y) + Z = X + (Y + Z)\)
\[= X + Y + Z\]
7D. \((X \cdot Y) \cdot Z = X \cdot (Y \cdot Z)\)
\[= X \cdot Y \cdot Z\]

**Distributive Laws:**
8. \(X \cdot (Y + Z) = (X \cdot Y) + (X \cdot Z)\)
8D. \(X + (Y \cdot Z) = (X + Y) \cdot (X + Z)\)

**Simplification Theorems:**
9. 
9D. 

10.  
10D. 

11.  
11D. 

Parenthesis order does not matter

Axiom

Actually worth remembering — they show up a lot in real designs…
Proving theorems via axioms of Boolean Algebra:

**EX: Prove the theorem:** \( X \cdot Y + X \cdot \bar{Y} = X \)

- Distributive (5)
- Complement (6)
- Identity (4)

**EX2: Prove the theorem:** \( X + X \cdot Y = X \)

- Identity (4)
- Distributive (5)
- Identity (2)
- Identity (4)
DeMorgan’s Law: Enabling Transformations

DeMorgan's Law:

12. \( (X + Y + Z + \cdots) = \overline{X} \cdot \overline{Y} \cdot \overline{Z} \cdot \cdots \)

12D. \( (X \cdot Y \cdot Z \cdot \cdots) = \overline{X} + \overline{Y} + \overline{Z} + \cdots \)

Think of this as a transformation

- Let’s say we have:

\[
F = A + B + C
\]

- Applying DeMorgan’s Law (12), gives us

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

At least one of A, B, C is TRUE \(\rightarrow\) It is not the case that A, B, C are all false
DeMorgan’s Law (Continued)

These are conversions between **different types of logic functions**. They can prove useful if you do not have **every type of gate**.

\[ A = \overline{(X + Y)} = \overline{XY} \]

**NOR is equivalent to AND with inputs complemented**

\[ B = \overline{(XY)} = \overline{X} + \overline{Y} \]

**NAND is equivalent to OR with inputs complemented**

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>X+Y</th>
<th>X</th>
<th>Y</th>
<th>XY</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>XY</th>
<th>X</th>
<th>Y</th>
<th>X+Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
We did not cover the following. They are for your preparation.
Using Boolean Equations to Represent a Logic Circuit
Assume we have the truth table of a Boolean Function

How do we canonically express the function in terms of the inputs in a standard manner?

Idea: **Sum of Products** form

Express the truth table as a two-level Boolean expression

- If ANY of the combinations of input variables that results in a 1 is TRUE, then the output is 1
- \( F = \text{OR of all input variable combinations that result in a 1} \)
Some Definitions

- **Complement**: variable with a bar over it
  \[\overline{A}, \overline{B}, \overline{C}\]

- **Literal**: variable or its complement
  \[A, \overline{A}, B, \overline{B}, C, \overline{C}\]

- **Implicant**: product (AND) of literals
  \[(A \cdot B \cdot \overline{C}), (\overline{A} \cdot C), (B \cdot \overline{C})\]

- **Minterm**: product (AND) that includes all input variables
  \[(A \cdot B \cdot \overline{C}), (\overline{A} \cdot B \cdot C), (\overline{A} \cdot B \cdot \overline{C})\]

- **Maxterm**: sum (OR) that includes all input variables
  \[(A + \overline{B} + \overline{C}), (\overline{A} + B + \overline{C}), (A + B + \overline{C})\]
Two-Level Canonical Forms

- **Truth table** is the unique signature of a Boolean function ...
  - But, it is an expensive representation

- A Boolean function can have many alternative Boolean expressions
  - i.e., many alternative Boolean expressions (and gate realizations) may have the same truth table (and function)

- **Canonical form**: standard form for a Boolean expression
  - Provides a unique algebraic signature
  - If they all say the same thing, why do we care?
    - Different Boolean expressions lead to different gate realizations
Two-Level Canonical Forms

Sum of Products Form (SOP)
Also known as disjunctive normal form or minterm expansion

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

\[ F = \overline{A}BC + \overline{A}BC + \overline{A}BC + \overline{ABC} + ABC \]

- Each row in a truth table has a minterm
- A minterm is a product (AND) of literals
- Each minterm is TRUE for that row (and only that row)

All Boolean equations can be written in SOP form

The output of the function is TRUE if for any of the input combinations
SOP Form — Why Does It Work?

Only the shaded product term — \( \overline{A}BC \) — will be 1

No other product terms will “turn on” — they will all be 0

So if inputs \( A \ B \ C \) correspond to a product term in expression,

- We get \( 0 + 0 + ... + 1 + ... + 0 + 0 = 1 \) for output

If inputs \( A \ B \ C \) do not correspond to any product term in expression

- We get \( 0 + 0 + ... + 0 = 0 \) for output
Aside: Notation for SOP

- Standard “shorthand” notation here
  - If we agree on the order of the variables in the rows of truth table...
    - then we can enumerate each row with the decimal number that corresponds to the binary number created by the input pattern

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

100 = decimal 4 so this is minterm #4, or m4

111 = decimal 7 so this is minterm #7, or m7

f = \sum m(3,4,5,6,7)

f = m3 + m4 + m5 + m6 + m7

We can write this as a sum of products

Or, we can use a summation notation
Canonical SOP Forms

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>minterms</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>$\overline{ABC}$ = m0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>$\overline{ABC}$ = m1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>$\overline{A\overline{B}C}$ = m2</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>$\overline{ABC}$ = m3</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>$\overline{A\overline{B}C}$ = m4</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>$\overline{ABC}$ = m5</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>$\overline{ABC}$ = m6</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>$\overline{ABC}$ = m7</td>
</tr>
</tbody>
</table>

Shorthand Notation for Minterms of 3 Variables

F in canonical form:

$F(A,B,C) = \sum m(3,4,5,6,7)$

$= m3 + m4 + m5 + m6 + m7$

canonical form $\neq$ minimal form

2-Level AND/OR Realization
From Logic to Gates

- **SOP (sum-of-products) leads to two-level logic**

- Example: \( Y = (\overline{A} \cdot \overline{B} \cdot C) + (A \cdot \overline{B} \cdot C) + (A \cdot \overline{B} \cdot C) \)
Alternative Canonical Form: POS

A product of sums (POS)

Each sum term represents one of the “zeros” of the function

For the given input, only the shaded sum term will equal 0

Anything ANDed with 0 is 0; Output F will be 0
Consider $A=0$, $B=1$, $C=0$

<table>
<thead>
<tr>
<th>$A$</th>
<th>$B$</th>
<th>$C$</th>
<th>$F$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Input $\begin{array}{c}0 \ 1 \ 0\end{array}$

$$F = (A + B + C)(A + B + \overline{C})(A + \overline{B} + C)$$

Only one of the products will be 0, anything ANDed with 0 is 0

Therefore, the output is $F = 0$
Maxterm form:

1. Find truth table rows where F is 0
2. 0 in input col → true literal
3. 1 in input col → complemented literal
4. OR the literals to get a Maxterm
5. AND together all the Maxterms

Or just remember, POS of F is the same as the DeMorgan of SOP of F!!
**Canonical POS Forms**

Product of Sums / Conjunctive Normal Form / Maxterm Expansion

\[ F = (A + B + C)(A + B + \overline{C})(A + \overline{B} + C) \]

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>Maxterms</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>( A + B + C ) = M0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>( A + B + \overline{C} ) = M1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>( A + \overline{B} + C ) = M2</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>( A + \overline{B} + \overline{C} ) = M3</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>( \overline{A} + B + C ) = M4</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>( \overline{A} + B + \overline{C} ) = M5</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>( \overline{A} + \overline{B} + C ) = M6</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>( \overline{A} + \overline{B} + \overline{C} ) = M7</td>
</tr>
</tbody>
</table>

Maxterm shorthand notation for a function of three variables

Note that you form the maxterms around the “zeros” of the function. This is **not** the complement of the function!

\[ \prod M(0, 1, 2) \]
Useful Conversions

1. **Minterm to Maxterm conversion:**
   rewrite minterm shorthand using maxterm shorthand
   replace minterm indices with the indices not already used
   E.g., \( F(A, B, C) = \sum m(3, 4, 5, 6, 7) = \prod M(0, 1, 2) \)

2. **Maxterm to Minterm conversion:**
   rewrite maxterm shorthand using minterm shorthand
   replace maxterm indices with the indices not already used
   E.g., \( F(A, B, C) = \prod M(0, 1, 2) = \sum m(3, 4, 5, 6, 7) \)

3. **Expansion of \( F \) to expansion of \( \overline{F} \):**
   E.g., \( F(A, B, C) = \sum m(3, 4, 5, 6, 7) \rightarrow \overline{F}(A, B, C) = \sum m(0, 1, 2) \)
   \[ = \prod M(0, 1, 2) \rightarrow = \prod M(3, 4, 5, 6, 7) \]

4. **Minterm expansion of \( F \) to Maxterm expansion of \( \overline{F} \):**
   rewrite in Maxterm form, using the same indices as \( F \)
   E.g., \( F(A, B, C) = \sum m(3, 4, 5, 6, 7) \rightarrow \overline{F}(A, B, C) = \prod M(3, 4, 5, 6, 7) \)
   \[ = \prod M(0, 1, 2) \rightarrow = \sum m(0, 1, 2) \]
Combinational Circuits Used in Modern Computers
Combinational Building Blocks

- Combinational logic is often grouped into larger building blocks to build more complex systems.

- Hides the unnecessary gate-level details to emphasize the function of the building block.

- We now look at:
  - Decoders
  - Multiplexers
  - Full adder
  - PLA
Decoder

- Exactly one of the inputs is 1 and all the rest are 0s
- \( n \) inputs and \( 2^n \) outputs
- The one output that is logically 1 is the output corresponding to the input pattern that the logic circuit is expected to detect

\[
\begin{align*}
A = 1 & \quad B = 0 \\
1 & \text{ if } A, B \text{ is 00} \\
1 & \text{ if } A, B \text{ is 01} \\
1 & \text{ if } A, B \text{ is 10} \\
1 & \text{ if } A, B \text{ is 11}
\end{align*}
\]
Decoder

- The decoder is useful in determining how to interpret a bit pattern

  - It could be the address of a row in DRAM, that the processor intends to read from

  - It could be an instruction in the program and the processor has to decide what action to do! (based on instruction opcode)
Multiplexer (MUX), or Selector

- **Selects** one of the \( N \) inputs to connect it to the output
- Needs \( \log_2 N \)-bit control input
- **2:1 MUX:**

![2:1 MUX Circuit Diagram]
Multiplexer (MUX)

- The output C is always connected to either the input A or the input B
  - Output value depends on the value of the select line S

<table>
<thead>
<tr>
<th>S</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A</td>
</tr>
<tr>
<td>1</td>
<td>B</td>
</tr>
</tbody>
</table>

![Diagram of a multiplexer](image-url)
**Full Adder (I)**

- **Binary addition**
  - Similar to decimal addition
  - From right to left
  - One column at a time
  - One sum and one carry bit

- Truth table of binary addition on one column of bits within two n-bit operands

<table>
<thead>
<tr>
<th>$a_i$</th>
<th>$b_i$</th>
<th>carry$_i$</th>
<th>carry$_{i+1}$</th>
<th>$S_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
**Full Adder (II)**

- **Binary addition**
  - N 1-bit additions
  - **Use SOP of one bit addition**

![Full Adder (1 bit)](image)

\[
\begin{align*}
a_{n-1}a_{n-2} \ldots a_1a_0 \\
b_{n-1}b_{n-2} \ldots b_1b_0 \\
c_n \ C_{n-1} \ldots \ C_1 \\
S_{n-1} \ldots S_1S_0
\end{align*}
\]

<table>
<thead>
<tr>
<th>(a_i)</th>
<th>(b_i)</th>
<th>(\text{carry}_i)</th>
<th>(\text{carry}_{i+1})</th>
<th>(S_i)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Full Adder (III)

- Creating a **4-bit adder** out of 1-bit full adders
  - To add two 4-bit binary numbers A and B

```
+  a_3  a_2  a_1  a_0  
    b_3  b_2  b_1  b_0  
    c_4  c_3  c_2  c_1  
---------------------
  s_3  s_2  s_1  s_0  
```

```
+  1  0  1  1  
    1  0  0  1  
    1  0  1  1  
---------------------
  0  1  0  0  0  
```
The Programmable Logic Array (PLA)

- The below logic structure is a very common building block for implementing any collection of logic functions one wishes to.
- An array of AND gates followed by an array of OR gates.
- How do we determine the number of AND gates?
  - Remember SOP: the number of possible minterms.
  - For an n-input logic function, we need a PLA with $2^n$ n-input AND gates.
- How do we determine the number of OR gates? The number of output columns in the truth table.
The Programmable Logic Array (PLA)

- **How do we implement a logic function?**
  - Connect the output of an AND gate to the input of an OR gate if the corresponding minterm is included in the SOP
  - This is a simple programmable logic

- **Programming a PLA:** we program the connections from AND gate outputs to OR gate inputs to implement a desired logic function

- Have you seen any other type of programmable logic?
  - Yes! An FPGA...
  - An FPGA uses more advanced structures, as we saw in Lecture 3
Implementing a Full Adder Using a PLA

Truth table of a full adder

<table>
<thead>
<tr>
<th>$a_i$</th>
<th>$b_i$</th>
<th>carry$_i$</th>
<th>carry$_{i+1}$</th>
<th>$S_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

This input should not be connected to any outputs.

We do not need this output.
Logical Completeness

Any logic function we wish to implement could be accomplished with PLA

- PLA consists of only AND gates, OR gates, and inverters
- We just have to program connections based on SOP of the intended logic function

The set of gates \{AND, OR, NOT\} is logically complete because we can build a circuit to carry out the specification of any truth table we wish, without using any other kind of gate.
LOGIC SIMPLIFICATION:
KARNAUGH MAPS
Quick Recap on Logic Simplification

- The original Boolean expression (i.e., logic circuit) may not be optimal

\[ F = \neg A(A + B) + (B + AA)(A + \neg B) \]

- Can we reduce a given Boolean expression to an equivalent expression with fewer terms?

\[ F = A + B \]

- The goal of logic simplification:
  - Reduce the number of gates/inputs
  - Reduce implementation cost

A basis for what the automated design tools are doing today
Logic Simplification

- Systematic techniques for simplifications
  - amenable to automation

**Key Tool: The Uniting Theorem** — \( F = A\overline{B} + AB \)

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

\[
F = A\overline{B} + AB = A(\overline{B} + B) = A(1) = A
\]

**Essence of Simplification:**
Find two element subsets of the ON-set where only one variable changes its value. This single varying variable can be eliminated!

\[
\rightarrow \text{B is eliminated, A remains}
\]

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

\[
G = \overline{A}B + AB = (\overline{A} + A)\overline{B} = \overline{B}
\]

- B's values stay the same within the ON-set rows
- A's values change within the ON-set rows

\[
\rightarrow \text{A is eliminated, B remains}
\]
Complex Cases

- **One example**
  \[ Cout = \overline{A}BC + \overline{A}BC + ABC + ABC \]

- **Problem**
  - Easy to see how to apply Uniting Theorem...
  - Hard to know if you applied it in all the right places...
  - ...especially in a function of many more variables

- **Question**
  - Is there an easier way to potential simplifications?
  - i.e., potential applications of Uniting Theorem...?

- **Answer**
  - Need an intrinsically *geometric* representation for Boolean f( )
  - Something we can draw, see...
Karnaugh Map

- Karnaugh Map (K-map) method
  - K-map is an alternative method of representing the truth table that helps visualize adjacencies in up to 6 dimensions
  - Physical adjacency ↔ Logical adjacency

2-variable K-map

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>01</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>11</td>
</tr>
</tbody>
</table>

3-variable K-map

<table>
<thead>
<tr>
<th></th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
<td>001</td>
<td>011</td>
<td>010</td>
</tr>
<tr>
<td>1</td>
<td>100</td>
<td>101</td>
<td>111</td>
<td>110</td>
</tr>
</tbody>
</table>

4-variable K-map

<table>
<thead>
<tr>
<th></th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0000</td>
<td>0001</td>
<td>0011</td>
<td>0010</td>
</tr>
<tr>
<td>01</td>
<td>0100</td>
<td>0101</td>
<td>0111</td>
<td>0110</td>
</tr>
<tr>
<td>11</td>
<td>1100</td>
<td>1101</td>
<td>1111</td>
<td>1110</td>
</tr>
<tr>
<td>10</td>
<td>1000</td>
<td>1001</td>
<td>1011</td>
<td>1010</td>
</tr>
</tbody>
</table>

Numbering Scheme: 00, 01, 11, 10 is called a “Gray Code” — only a single bit changes from code word to next code word
### Karnaugh Map Methods

#### Adjacent

Kmap adjacencies go “around the edges”

- Wrap around from first to last column
- Wrap around from top row to bottom row

<table>
<thead>
<tr>
<th></th>
<th>A=0</th>
<th>A=1</th>
</tr>
</thead>
<tbody>
<tr>
<td>B=0</td>
<td>000 001 011 010</td>
<td></td>
</tr>
<tr>
<td>B=1</td>
<td>100 101 111 110</td>
<td></td>
</tr>
</tbody>
</table>
K-map Cover - 4 Input Variables

Strategy for “circling” rectangles on Kmap:

Biggest “oops!” that people forget:

\[ F(A, B, C, D) = \sum_{m(0, 2, 5, 8, 9, 10, 11, 12, 13, 14, 15)} \]

\[ F = B\overline{C}D + A + \overline{B}D \]
K-map Rules

- **What can be legally combined (circled) in the K-map?**
  - Rectangular groups of size $2^k$ for any integer $k$
  - Each cell has the same value (1, for now)
  - All values must be adjacent
    - Wrap-around edge is okay

- **How does a group become a term in an expression?**
  - Determine which literals are constant, and which vary across group
  - Eliminate varying literals, then AND the constant literals
    - constant 1 $\rightarrow$ use $x$, constant 0 $\rightarrow$ use $\overline{x}$

- **What is a good solution?**
  - Biggest groupings $\rightarrow$ eliminate more vars (literals) in each term
  - Fewest groupings $\rightarrow$ fewer terms (gates) all together
  - OR together all AND terms you create from individual groups
K-map Example: Two-bit Comparator

Design Approach:

Write a 4-Variable K-map for each of the 3 output functions

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>F1</th>
<th>F2</th>
<th>F3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
K-map Example: Two-bit Comparator (2)

K-map for F1

\[ F1 = A'B'C'D' + A'BC'D + ABCD + AB'CD' \]
K-map Example: Two-bit Comparator (3)

K-map for F2

F2 = A'C + A'B'D + B'CD

F3 = ? (Exercise for you)
K-maps with “Don’t Care”

- Don’t Care really means *I don’t care what my circuit outputs if this appears as input*

- You have an engineering choice to use DON’T CARE patterns intelligently as 1 or 0 to better *simplify* the circuit

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>F</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- I can pick 00, 01, 10, 11 independently of below
- I can pick 00, 01, 10, 11 independently of above
Example: BCD Increment Function

- BCD (Binary Coded Decimal) digits
  - Encode decimal digits 0 - 9 with bit patterns $0000_2$ — $1001_2$
  - When **incremented**, the decimal sequence is 0, 1, ..., 8, 9, 0, 1

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>W</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

These input patterns **should never be encountered** in practice (hey -- it’s a BCD number!)
So, associated output values are “Don’t Cares”
K-map for BCD Increment Function

Z (without don’t cares) = 

Z (with don’t cares) = 

\[ Z = A'D' + B'C'D' \]
K-map Summary

- **Karnaugh maps** as a formal systematic approach for logic simplification

- 2-, 3-, 4-variable K-maps

- K-maps with "Don’t Care" outputs
Next Lecture: Hardware Description Languages & Verilog