Extra Credit Assignment: Talk Analysis

- Intelligent Architectures for Intelligent Machines
- Watch and analyze this short lecture (33 minutes)
  - https://www.youtube.com/watch?v=WxHribseelw (Oct 2022)

Assignment – for 1% extra credit

- Write a good 1-page summary (following our guidelines)
  - What are your key takeaways?
  - What did you learn?
  - What did you like or dislike?
  - Submit your summary to Moodle – deadline April 1
Assignment: Readings

This week
- Combinational Logic
  - P&P Chapter 3 until 3.3 + H&H Chapter 2

Next week
- Hardware Description Languages and Verilog
  - H&H Chapter 4 until 4.3 and 4.5
- Sequential Logic
  - P&P Chapter 3.4 until end + H&H Chapter 3 in full

Within 2-3 weeks, we will be done with
- P&P Chapters 1-3 + H&H Chapters 1-4
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 example chips & platforms we surveyed
… 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

Burks, Goldstein, von Neumann, “Preliminary discussion of the logical design of an electronic computing instrument,” 1946.
What is A Computer?

- Three key components
- Computation
- Communication
- Storage/memory

Burks, Goldstein, von Neumann, “Preliminary discussion of the logical design of an electronic computing instrument,” 1946.
What is A Computer?

- We will cover all three components

Processing
- control (sequencing)
- datapath

Memory
- program and data

I/O
Recall: The Transformation Hierarchy

Computer Architecture (expanded view)

Problem
Algorithm
Program/Language
System Software
SW/HW Interface
Micro-architecture
Logic
Devices
Electrons

Computer Architecture (narrow view)
What We Will Cover (I)

- Combinational Logic Design
- Hardware Description Languages (Verilog)
- Sequential Logic Design
- Timing and Verification
- ISA (MIPS and LC3b as examples)
- Assembly Programming
What We Will Cover (II)

- Microarchitecture Fundamentals
- Single-cycle Microarchitectures
- Multi-cycle and Microprogrammed Microarchitectures
- Pipelining
- Issues in Pipelining: Dependence Handling, State Maintenance and Recovery, ...
- Branch Prediction
- Out-of-Order Execution
- Superscalar Execution
- Other Paradigms: Dataflow, VLIW, Systolic, SIMD/GPUs, ...
What We Will Cover (III)

- Memory Technology and Organization
- Memory Hierarchy
- Caches
- Multi-Core Caches
- Prefetching
- Virtual Memory
DDCA (Spring 2022)

- **Spring 2022 Edition:**

- **Spring 2021 Edition:**

- **Youtube Livestream (Spring 2022):**
  - [https://www.youtube.com/watch?v=cpXdeE3HwyK0&list=PL5Q2soXY2Zi97Ya5DEUpMpo2bbAoaG7c6](https://www.youtube.com/watch?v=cpXdeE3HwyK0&list=PL5Q2soXY2Zi97Ya5DEUpMpo2bbAoaG7c6)

- **Youtube Livestream (Spring 2021):**
  - [https://www.youtube.com/watch?v=LbC0EZY8yw4&list=PL5Q2soXY2Zi_uej3aY39YB5pfW4SJ7LIN](https://www.youtube.com/watch?v=LbC0EZY8yw4&list=PL5Q2soXY2Zi_uej3aY39YB5pfW4SJ7LIN)

- Bachelor’s course
  - 2nd semester at ETH Zurich
  - Rigorous introduction into “How Computers Work”
  - Digital Design/Logic
  - Computer Architecture
  - 10 FPGA Lab Assignments

[https://www.youtube.com/onurmutlulectures](https://www.youtube.com/onurmutlulectures)
Processing Paradigms We Will Cover

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

- Basic building blocks of modern computers
  - Transistors
  - Logic gates

- Boolean algebra

- Combinational logic circuits

- How to use Boolean algebra to represent combinational circuits

- Minimizing logic circuits (if time permits)
# General Purpose vs. Special Purpose Systems

<table>
<thead>
<tr>
<th>General Purpose</th>
<th>Special Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPUs</td>
<td>ASICs</td>
</tr>
<tr>
<td>Flexible: Can execute any program</td>
<td>Efficient &amp; High performance</td>
</tr>
<tr>
<td>Easy to program &amp; use</td>
<td>(Usually) Difficult to program &amp; use</td>
</tr>
<tr>
<td>Not the best performance &amp; efficiency</td>
<td>Inflexible: Limited set of programs</td>
</tr>
</tbody>
</table>

- **CPUs**
  - Apple M1

- **GPUs**
  - Nvidia GTX 1070

- **FPGAs**
  - Xilinx Spartan

- **ASICs**
  - Cerebras WSE-2
General Purpose vs. Special Purpose Systems

**General Purpose**
- Flexible: Can work with any bolt
- Easy to use
- Not the best fit, results or efficiency

**Special Purpose**
- Efficient & High performance
- (Usually) Difficult to use
- Inflexible: Only for fitting bolts

https://www.ubuy.vn/en/product/2MBUW8M-crescent-8-adjustable-wrench-carded-ac28vs
https://capritools.com/shop/bolt-extractor-wrench-set-metric-8-19-mm/
General-Purpose Microprocessors
Modern General-Purpose Microprocessors

5-nanometer process
The first personal computer chip built with this cutting-edge technology.

16 billion transistors
The most we’ve ever put into a single chip.

Source: https://www.apple.com/mac/m1/
Modern General-Purpose Microprocessors

Source: https://www.anandtech.com/show/16252/mac-mini-apple-m1-tested

Apple M1, 2021
Modern General-Purpose Microprocessors

Source: https://www.golem.de/news/m1-pro-max-dieses-apple-silicon-ist-gigantisch-2110-160415.html
Apple M1 Ultra (2022)
Apple M1 Ultra (2022)

- ProRes: Encode and decode
- Thunderbolt 4
- 5 nm process
- 114 billion Transistors
- Silicon interposer with 2.5TB/s interprocessor bandwidth
- 800GB/s Memory bandwidth
- Up to 20-core CPU
- Up to 64-core GPU
- 32-core Neural Engine: 22 trillion operations per second
- Secure Enclave
- Industry-leading performance per watt
- UltraFusion architecture
- Up to 128GB unified memory

https://stadt-bremerhaven.de/apple-neuer-m1-ultra-chip-ist-offiziell/
Modern General-Purpose Microprocessors

Source: https://twitter.com/Locuza_/status/1454152714930331652

Intel Alder Lake, 2021
Modern GPUs

Source: https://en.wikichip.org/wiki/nvidia/tegra/xavier
FPGAs
Modern FPGAs

Source: https://www.mouser.ch/new/xilinx/xilinx-zyng-7000-zc702-eval-kit/
Special-Purpose ASICs (App-Specific Integrated Circuits)
Modern Special-Purpose ASICs

Figure 3. TPU Printed Circuit Board. It can be inserted in the slot for an SATA disk in a server, but the card uses PCIe Gen3 x16.


Figure 4. Systolic data flow of the Matrix Multiply Unit. Software has the illusion that each 256B input is read at once, and they instantly update one location of each of 256 accumulator RAMs.
Modern Special-Purpose ASICs

New ML applications (vs. TPU3):
- Computer vision
- Natural Language Processing (NLP)
- Recommender system
- Reinforcement learning that plays Go

250 TFLOPS per chip in 2021
vs 90 TFLOPS in TPU3

1 ExaFLOPS per board

https://spectrum.ieee.org/tech-talk/computing/hardware/heres-how-googles-tpu-v4-ai-chip-stacked-up-in-training-tests
Modern Special-Purpose ASICs

- The largest ML accelerator chip (2021)
- 850,000 cores

Cerebras WSE-2
2.6 Trillion transistors
46,225 mm²

Largest GPU
54.2 Billion transistors
826 mm²

NVIDIA Ampere GA100

https://www.anandtech.com/show/14758/hot-chips-31-live-blogs-cerebras-wafer-scale-deep-learning

https://www.cerebras.net/cerebras-wafer-scale-engine-why-we-need-big-chips-for-deep-learning/
Modern Special-Purpose ASICs

Warehouse-Scale Video Acceleration: Co-design and Deployment in the Wild

(a) Chip floorplan  (b) Two chips on a PCBA

Figure 5: Pictures of the VCU

Source: https://dl.acm.org/doi/pdf/10.1145/3445814.3446723
## 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
## 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
## They All Look the Same

<table>
<thead>
<tr>
<th></th>
<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>
<td></td>
</tr>
<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><strong>Performance</strong></td>
<td>0</td>
<td>+</td>
<td>++</td>
</tr>
</tbody>
</table>
# 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><strong>Performance</strong></td>
<td>0</td>
<td>+</td>
<td>++</td>
</tr>
<tr>
<td><strong>Good for</strong></td>
<td>Ubiquitous Simple to use</td>
<td>Prototyping Small volume</td>
<td>Mass production, Max performance</td>
</tr>
</tbody>
</table>

**SAFARI**
# 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><strong>Performance</strong></td>
<td>o</td>
<td>+</td>
<td>++</td>
</tr>
<tr>
<td><strong>Good for</strong></td>
<td>Ubiquitous</td>
<td>Prototyping</td>
<td>Mass production, Max performance</td>
</tr>
<tr>
<td></td>
<td>Simple to use</td>
<td>Small volume</td>
<td></td>
</tr>
<tr>
<td><strong>Programming</strong></td>
<td>Executable file</td>
<td>Bit file</td>
<td>Design masks</td>
</tr>
<tr>
<td><strong>Languages</strong></td>
<td>C/C++/Java/…</td>
<td>Verilog/VHDL</td>
<td>Verilog/VHDL</td>
</tr>
<tr>
<td><strong>Main Companies</strong></td>
<td>Intel, ARM, AMD, Apple, NVIDIA</td>
<td>Xilinx, Altera</td>
<td>TSMC, Globalfoundries</td>
</tr>
</tbody>
</table>
# Labs: Build A Microprocessor on FPGA

<table>
<thead>
<tr>
<th></th>
<th>Microprocessors</th>
<th>FPGAs</th>
<th>By programming these</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>In short</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Common building block of computers</td>
<td>Reconfigurable hardware, flexible</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Program Development Time</td>
<td>minutes</td>
<td>days</td>
<td>months</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Performance</td>
<td>0</td>
<td>+</td>
<td>++</td>
</tr>
<tr>
<td>Good for</td>
<td>Ubiquitous</td>
<td>Prototyping</td>
<td>Mass production</td>
</tr>
<tr>
<td></td>
<td>Simple to use</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Programming</td>
<td>Executable file</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Languages</td>
<td>C/C++/Java/...</td>
<td>Verilog/VHDL</td>
<td>Verilog/VHDL</td>
</tr>
<tr>
<td>Main Companies</td>
<td>Intel, ARM, AMD, Apple, NVIDIA</td>
<td>Xilinx, Altera</td>
<td>TSMC, Globalfoundries</td>
</tr>
</tbody>
</table>

Want to learn how these work

Using this language

Verilog/VHDL
All Computers are Built Upon the Same Building Blocks
Building Blocks of Modern Computers
Transistors
Transistors

- Computers are built from very large numbers of very small (and relatively simple) structures: transistors
  - Intel 4004, in 1971, had 2300 MOS transistors
  - Intel’s Pentium IV microprocessor, 2000, was made up of more than 42 Million MOS transistors
  - Apple’s M2 Max, offered for sale in 2022, is made up of more than 67 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

Sections 1.6 and 1.7 in Harris and Harris provide lower-level explanations
Different Types of MOS Transistors

- There are two types of MOS transistors: \textit{n-type} and \textit{p-type}.

- They both operate “logically,” very similar to the way wall switches work.
How Does a Transistor 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 (i.e., the circuit is closed).

Depending on the technology, high voltage can range from 0.3V to 3V.

If the gate of the n-type transistor is supplied with zero voltage, the connection between the source and drain is broken (i.e., the circuit is open).
How Does a Transistor Work?

- The **n-type** transistor in a circuit with a battery and a bulb:
  - The circuit is closed when the gate is supplied with 3V.

- 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 0V.
Logic Gates
Now, we know how a MOS transistor works

How do we build logic structures out of MOS transistors?

We construct basic logical units out of individual MOS transistors

These logical units are called logic gates
- They implement simple Boolean functions

George Boole, “The Mathematical Analysis of Logic,” 1847.
Modern computers use both n-type and p-type transistors, i.e. Complementary MOS (CMOS) technology.

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 transistors are good at pulling up the voltage

p-type transistor pulls the output up

Y = 3V
Functionality of Our CMOS Circuit

What happens when the input is connected to 3V?

n-type transistors are good at pulling down the voltage
CMOS NOT Gate (Inverter)

- 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

\[
Y = \overline{A}
\]

<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>
CMOS NOT Gate (Inverter)

- 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

We call this a **NOT gate** or an **inverter** (bubble indicates inversion)

**Truth table:** shows what is 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>
Another CMOS Gate: What Is This?

- 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 up the output to 3V

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

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

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

- Let’s build more complex gates!

![CMOS NAND Gate Diagram]

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

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

We call this a NAND gate (bubble indicates inversion)
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>

\[ Y = A \cdot B = AB \]

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

Food for thought: Can we not use fewer transistors for the AND gate?
CMOS NOT, NAND, AND Gates

A \rightarrow Y

<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>

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

<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>
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**

pMOS transistors are used for pull-up
nMOS transistors are used for pull-down
General CMOS Gate Structure (II)

- Exactly one network should be ON, and the other network should be OFF at any given time

  - If both networks are ON at the same time, there is a **short circuit** → likely incorrect operation
  
  - If both networks are OFF at the same time, the output is **floating** → undefined

- pMOS transistors are used for pull-up
- nMOS transistors are used for pull-down
Digging Deeper: Why This Structure?

- MOS transistors are **imperfect** switches
- pMOS transistors pass 1’s well but 0’s poorly (holes carry charge)
- nMOS transistors pass 0’s well but 1’s poorly (electrons carry charge)
- pMOS transistors are good at “pulling up” the output
- nMOS transistors are good at “pulling down” the output

See Section 1.7 in H&H
Digging Deeper: Latency

- Which one is faster?
  - Transistors in series
  - Transistors in parallel

- Series connections are slower than parallel connections
  - More resistance on the wire

- How do you alleviate this latency?
  - See H&H Section 1.7.8 for an example: pseudo-nMOS Logic

Used in the past when pMOS transistors could not be fabricated well
Digging Deeper: Power Consumption

- **Dynamic Power Consumption**
  - Power used to charge capacitance as signals change (0 $\leftrightarrow$ 1)
  - $C \times V^2 \times f$
    - $C =$ capacitance of the circuit (wires and gates)
    - $V =$ supply voltage
    - $f =$ charging frequency of the capacitor

- **Static Power consumption**
  - Power used when signals do not change
  - $V \times I_{\text{leakage}}$
    - supply voltage $\times$ leakage current

- **Energy Consumption**
  - $\text{Power} \times \text{Time}$

See more in H&H Chapter 1.8
## Common Logic Gates

<table>
<thead>
<tr>
<th>Buffer</th>
<th>AND</th>
<th>OR</th>
<th>XOR</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Buffer Diagram" /></td>
<td><img src="image" alt="AND Diagram" /></td>
<td><img src="image" alt="OR Diagram" /></td>
<td><img src="image" alt="XOR Diagram" /></td>
</tr>
<tr>
<td><strong>Truth Table</strong></td>
<td><strong>Truth Table</strong></td>
<td><strong>Truth Table</strong></td>
<td><strong>Truth Table</strong></td>
</tr>
<tr>
<td>A</td>
<td>Z</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Inverter</th>
<th>NAND</th>
<th>NOR</th>
<th>XNOR</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Inverter Diagram" /></td>
<td><img src="image" alt="NAND Diagram" /></td>
<td><img src="image" alt="NOR Diagram" /></td>
<td><img src="image" alt="XNOR Diagram" /></td>
</tr>
<tr>
<td><strong>Truth Table</strong></td>
<td><strong>Truth Table</strong></td>
<td><strong>Truth Table</strong></td>
<td><strong>Truth Table</strong></td>
</tr>
<tr>
<td>A</td>
<td>Z</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<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>
</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

Figure 1.35 Three-input NAND gate schematic
Aside: Moore’s Law: Enabler of Many Gates on a Chip
An Enabler: Moore’s Law


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.
Moore’s Law: The number of transistors on microchips doubles every two years.

Moore’s law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important for other aspects of technological progress in computing – such as processing speed or the price of computers.

Transistor count

Data source: Wikipedia (wikipedia.org/wiki/Transistor_count)
OurWorldInData.org – Research and data to make progress against the world’s largest problems. Licensed under CC-BY by the authors Hannah Ritchie and 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: Innovation

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

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

- **Enabling precision manufacturing**
  - Extreme ultraviolet (EUV) light to pattern <10nm structures

- **Creating new device technologies**
  - FinFET, Gate All Around transistor, Single Electron Transistor...
A 5-Minute Video on Transistor Innovation

https://www.youtube.com/watch?v=Z7M8etXUEUU
A 5-Minute Video on Transistor Innovation

https://www.youtube.com/watch?v=Z7M8etXUEUU
Enabling Manufacturing Tech: EUV

https://www.youtube.com/watch?v=Jv40Viz-KTc
Innovation At the Bottom Enables Computing

<table>
<thead>
<tr>
<th>Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algorithm</td>
</tr>
<tr>
<td>Program/Language</td>
</tr>
<tr>
<td>Runtime System</td>
</tr>
<tr>
<td>(VM, OS, MM)</td>
</tr>
<tr>
<td>ISA (Architecture)</td>
</tr>
<tr>
<td>Microarchitecture</td>
</tr>
<tr>
<td>Logic</td>
</tr>
<tr>
<td>Devices</td>
</tr>
<tr>
<td>Electrons</td>
</tr>
</tbody>
</table>
"There's Plenty of Room at the Bottom: An Invitation to Enter a New Field of Physics" was a lecture given by physicist Richard Feynman at the annual American Physical Society meeting at Caltech on December 29, 1959.[1] Feynman considered the possibility of direct manipulation of individual atoms as a more powerful form of synthetic chemistry than those used at the time. Although versions of the talk were reprinted in a few popular magazines, it went largely unnoticed and did not inspire the conceptual beginnings of the field. Beginning in the 1980s, nanotechnology advocates cited it to establish the scientific credibility of their work.

https://en.wikipedia.org/wiki/There%27s_Plenty_of_Room_at_the_Bottom
Feynman considered some ramifications of a general ability to manipulate matter on an atomic scale. He was particularly interested in the possibilities of denser computer circuitry, and microscopes that could see things much smaller than is possible with scanning electron microscopes. These ideas were later realized by the use of the scanning tunneling microscope, the atomic force microscope and other examples of scanning probe microscopy and storage systems such as Millipede, created by researchers at IBM.

Feynman also suggested that it should be possible, in principle, to make nanoscale machines that "arrange the atoms the way we want", and do chemical synthesis by mechanical manipulation.

He also presented the possibility of "swallowing the doctor", an idea that he credited in the essay to his friend and graduate student Albert Hibbs. This concept involved building a tiny, swallowable surgical robot.

https://en.wikipedia.org/wiki/There%27s_Plenty_of_Room_at_the_Bottom
Extra Assignment: Moore’s Law (I)

- **Paper review**

- **Optional Assignment – for 1% extra credit**
  - Write a 1-page review
  - Upload PDF file to Moodle – Deadline: April 1

I strongly recommend that you follow my guidelines for (paper) review (see next slide)
Extra Assignment 2: Moore’s Law (II)

- Guidelines on how to review papers critically
  - Guideline slides: pdf ppt
  - Video: https://www.youtube.com/watch?v=tOL6FANAJ8c

- Example reviews on “Main Memory Scaling: Challenges and Solution Directions” (link to the paper)
  - Review 1
  - Review 2

- Example review on “Staged memory scheduling: Achieving high performance and scalability in heterogeneous systems” (link to the paper)
  - Review 1
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 Logic Equations
Functional Specification

- **Functional specification** of outputs in terms of inputs
- What do we 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):**

\[
\begin{align*}
S &= F(A, B, C_{in}) \\
C_{out} &= G(A, B, C_{in})
\end{align*}
\]

\[
\begin{align*}
S &= A \oplus B \oplus C_{in} \\
C_{out} &= AB + AC_{in} + BC_{in}
\end{align*}
\]
### Simple Equations: NOT / AND / OR

#### \( \overline{A} \) (reads “not A”) is 1 iff A is 0

<table>
<thead>
<tr>
<th>A</th>
<th>( \overline{A} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

\[
A \rightarrow \overline{A}
\]

#### \( A \cdot B \) (reads “A and B”) is 1 iff A and B are both 1

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>( A \cdot B )</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>

\[
A \rightarrow B \rightarrow A \cdot B
\]

#### \( A + B \) (reads “A or B”) is 1 iff either A or B is 1

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>( A + B )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

\[
A \rightarrow B \rightarrow A + B
\]
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 **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

<table>
<thead>
<tr>
<th>Formal version</th>
<th>English version</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1.</strong> ( B ) contains at least two elements, 0 and 1, such that 0 ( \neq ) 1</td>
<td>Math formality...</td>
</tr>
<tr>
<td><strong>2.</strong> Closure ( a, b \in B ),</td>
<td>Result of AND, OR stays in set you start with</td>
</tr>
<tr>
<td>(i) ( a + b \in B )</td>
<td></td>
</tr>
<tr>
<td>(ii) ( a \cdot b \in B )</td>
<td></td>
</tr>
<tr>
<td><strong>3.</strong> Commutative Laws: ( a, b \in B ),</td>
<td>For primitive AND, OR of 2 inputs, order doesn’t matter</td>
</tr>
<tr>
<td>(i)</td>
<td></td>
</tr>
<tr>
<td>(ii)</td>
<td></td>
</tr>
<tr>
<td><strong>4.</strong> Identities: ( 0, 1 \in B )</td>
<td>There are identity elements for AND, OR, that give you back what you started with</td>
</tr>
<tr>
<td>(i)</td>
<td></td>
</tr>
<tr>
<td>(ii)</td>
<td></td>
</tr>
<tr>
<td><strong>5.</strong> Distributive Laws:</td>
<td>( \cdot ) distributes over +, just like algebra</td>
</tr>
<tr>
<td>(i)</td>
<td>...but + distributes over ( \cdot ), also (!!)</td>
</tr>
<tr>
<td>(ii)</td>
<td></td>
</tr>
<tr>
<td><strong>6.</strong> Complement:</td>
<td>There is a complement element; AND/ORing with it gives the identity elm.</td>
</tr>
<tr>
<td>(i)</td>
<td></td>
</tr>
<tr>
<td>(ii)</td>
<td></td>
</tr>
</tbody>
</table>
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. $X + 1 = 1$

<table>
<thead>
<tr>
<th>Idempotent Law:</th>
<th>Dual</th>
</tr>
</thead>
<tbody>
<tr>
<td>3. $X + X = X$</td>
<td>1D. $X \cdot 1 = X$</td>
</tr>
<tr>
<td>4. $X + 1 = 1$</td>
<td>2D. $X \cdot 0 = 0$</td>
</tr>
</tbody>
</table>

## Idempotent Law:

1. $X + 0 = X$
2. $X + 1 = 1$

<table>
<thead>
<tr>
<th>Involution Law:</th>
<th>Dual</th>
</tr>
</thead>
<tbody>
<tr>
<td>3. $X + X = X$</td>
<td>1D. $X \cdot 1 = X$</td>
</tr>
<tr>
<td>4. $X + 1 = 1$</td>
<td>2D. $X \cdot 0 = 0$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Laws of Complementarity:</th>
<th>Dual</th>
</tr>
</thead>
<tbody>
<tr>
<td>5. $X + \overline{X} = 1$</td>
<td>1D. $X \cdot 1 = X$</td>
</tr>
<tr>
<td>6. $X + Y = Y + X$</td>
<td>2D. $X \cdot 0 = 0$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Commutative Law:</th>
<th>Dual</th>
</tr>
</thead>
<tbody>
<tr>
<td>6. $X + Y = Y + X$</td>
<td>1D. $X \cdot 1 = X$</td>
</tr>
<tr>
<td>7. $X \cdot Y = Y \cdot X$</td>
<td>2D. $X \cdot 0 = 0$</td>
</tr>
</tbody>
</table>

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 (continued)

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

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

**Simplification Theorems:**
9. 
10. 
11. 

7D. \((X \cdot Y) \cdot Z = X \cdot (Y \cdot Z)\)
   \[= X \cdot Y \cdot Z\]

8D. \(X + (Y \cdot Z) = (X + Y) \cdot (X + Z)\)  
   **Axiom**

**Parenthesis order does not matter**

**Actually worth remembering — they show up a lot in real designs…**
**Boolean Algebra: Proving Things**

*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

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

At least one of A, B, C is TRUE --> 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 be useful if you do not have every type of gate... Or, if some types of gates are more desirable to use than others...

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

NOR is equivalent to AND with inputs complemented

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

NAND is equivalent to OR with inputs complemented

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>Y</td>
<td>X+Y</td>
<td>X</td>
<td>Y</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>-----</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>0</td>
<td>0</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>
</tr>
<tr>
<td>1</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>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>Y</td>
<td>XY</td>
<td>X</td>
<td>Y</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>----</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<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>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Using Boolean Equations to Represent a Logic Circuit
Boolean Equations Enable Us To…

- Represent the function of a combinational logic block
  - Functional Specification

- Methodically transform the function into simpler functions
  - which lead to different hardware realizations
  - Logic Minimization or Logic Simplification
  - We can automate this process → Computer-Aided Design or Electronic Design Automation

- Different Boolean expressions lead to different logic gate implementations
  → Different hardware area, cost, latency, energy properties
Standardized Function Representations

- Enable a single, universally-agreed-on way of representing a Boolean function starting from its truth table
  - Also called “canonical representations”

- **Sum of Products (SOP) form**

- **Product of Sums (POS) form**
Sum of Products Form: Key Idea

- Assume **we have the truth table of Boolean Function F**

- How do we 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
  - that contains **all** input variable combinations that result in a 1 output
  - If ANY of the combinations of input variables that results in a 1 is TRUE, then the output is 1
  - **F = OR of all input variable combinations that result in a 1**
Some Definitions (for a 3-Input Function)

- **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
  \( (A \cdot B \cdot \overline{C}), (\overline{A} \cdot \overline{B} \cdot C), (\overline{A} \cdot B \cdot \overline{C}) \)

- **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 (Standard) 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)
  - If they all specify the same thing, why do we care?
    - Different Boolean expressions lead to different logic gate implementations \(\rightarrow\) Different cost, latency, energy properties

- **Canonical** form: standard form for a Boolean expression
  - Provides a unique algebraic signature
Two-Level Canonical Forms: SOP

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>

- 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)

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

All Boolean equations can be written in SOP form

Find all the input combinations (minterms) for which the output of the function is TRUE.
SOP Form — Why Does It Work?

- Only the shaded product term — \( A \overline{B} \overline{C} = 1 \cdot 0 \cdot 1 \) — 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 \( 0 + 0 + \ldots + 1 + \ldots + 0 + 0 = 1 \) for output
- If inputs A B C do not correspond to any product term in expression
  - We get \( 0 + 0 + \ldots + 0 = 0 \) for output

The function evaluates to TRUE (i.e., output is 1) if any of the Products (minterms) causes the output to be 1
Standard Notation for SOP Form

- Standard “shorthand” notation
  - 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 \( m_4 \)

111 = decimal 7 so this is minterm #7, or \( m_7 \)

We can write this as a sum of products

\[
f = m_3 + m_4 + m_5 + m_6 + m_7
\]

Or, we can use a summation notation
Shorthand Notation for Minterms of 3 Variables

<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{ABC} = 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{ABC} = 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>

F in canonical form:

\[ F(A, B, C) = \sum m(3, 4, 5, 6, 7) = m3 + m4 + m5 + m6 + m7 \]

canonical form ≠ minimal form

2-Level AND/OR Realization
From SOP to Gates

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

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

SOP form does NOT directly lead to minimal logic
Canonical Sum of Products Form: Key Idea

- Any 1-bit function can be represented as a Sum of Products
- A “Product” is the Boolean AND that includes ALL input variables of the function → minterm
- The 1-bit Output of the Function can be represented as
  - Sum (OR) of all minterms that lead to a 1 in the Output
- Logically
  - The function evaluates to TRUE (i.e., output is 1) if ANY of the Products (minterms) causes the Output to be 1
  - **SOP form** represents the function as the SUM (OR) of all Products (minterms) that cause the Output to be 1
Alternative Canonical Form: POS

DeMorgan of SOP of $\overline{F}$

Find all the input combinations (maxterms) for which the output of the function is FALSE.

Product of Sums (POS)

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

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

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

$A + \overline{B} + C = 0 + 1 + 0$

Anything ANDed with 0 is 0; Output F will be 0

The function evaluates to FALSE (i.e., output is 0)

if any of the Sums (maxterms) causes the output to be 0
Consider $A=0$, $B=1$, $C=0$

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

Therefore, the output is $F = 0$
POS: How to Write It

Maxterm form:
1. Find truth table rows where F is 0
2. 0 in input col $\rightarrow$ true literal
3. 1 in input col $\rightarrow$ 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 $\overline{F}$

<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>

\[ F = (A + B + C)(A + B + \overline{C})(A + \overline{B} + C) \]
### Notation for the Canonical POS Form

**Product of Sums / Conjunctive Normal Form / Maxterm Expansion**

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

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

<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>

Note that you form the maxterms around the “zeros” of the function.

This is not the complement of the function!
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 \( \overline{F} \):**
   \[ E.g., F(A, B, C) = \sum m(3, 4, 5, 6, 7) \quad \longrightarrow \quad \overline{F}(A, B, C) = \sum m(0, 1, 2) \]
   \[ = \prod M(0, 1, 2) \quad \longrightarrow \quad = \prod M(3, 4, 5, 6, 7) \]

4. **Minterm expansion of \( F \) to Maxterm expansion of \( \overline{F} \):**
   rewrite in Maxterm form, using the same indices as \( F \)
   \[ E.g., F(A, B, C) = \sum m(3, 4, 5, 6, 7) \quad \longrightarrow \quad \overline{F}(A, B, C) = \prod M(3, 4, 5, 6, 7) \]
   \[ = \prod M(0, 1, 2) \quad \longrightarrow \quad = \sum m(0, 1, 2) \]
Logic Simplification (or Minimization)

- Using Boolean Algebra, we can simplify the SOP or POS form of any function in a methodical way.

- Starting with the canonical SOP or POS form enables convenience and automation.
  - Truth table → SOP/POS form → Boolean Simplification Rules

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

\[
\begin{align*}
S &= F(A, B, C_{\text{in}}) \\
C_{\text{out}} &= G(A, B, C_{\text{in}})
\end{align*}
\]

\[
\begin{align*}
S &= A \oplus B \oplus C_{\text{in}} \\
C_{\text{out}} &= AB + AC_{\text{in}} + BC_{\text{in}}
\end{align*}
\]
Logic Simplification Example: SOP Form

- **SOP (sum-of-products) form of function Y**

- **Example:** \( Y = (\overline{A} \cdot \overline{B} \cdot \overline{C}) + (A \cdot \overline{B} \cdot \overline{C}) + (A \cdot \overline{B} \cdot C) \)

SOP form does NOT directly lead to minimal logic
Logic Simplification Example: Simplified

- SOP (sum-of-products) form of function $Y$
- Example: $Y = (\overline{B} \cdot \overline{C}) + (A \cdot \overline{B})$
Let’s Cover Some Basic Combinational Blocks
Combinational Building Blocks used in Modern Computers
Recall: Common Logic Gates

<table>
<thead>
<tr>
<th>Buffer</th>
<th>AND</th>
<th>OR</th>
<th>XOR</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Buffer" /></td>
<td><img src="image" alt="AND" /></td>
<td><img src="image" alt="OR" /></td>
<td><img src="image" alt="XOR" /></td>
</tr>
<tr>
<td>A</td>
<td>Z</td>
<td>B</td>
<td>Z</td>
</tr>
<tr>
<td>A</td>
<td>Z</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>A</td>
<td>Z</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>A</td>
<td>Z</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>A</td>
<td>Z</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Inverter</th>
<th>NAND</th>
<th>NOR</th>
<th>XNOR</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Inverter" /></td>
<td><img src="image" alt="NAND" /></td>
<td><img src="image" alt="NOR" /></td>
<td><img src="image" alt="XNOR" /></td>
</tr>
<tr>
<td>A</td>
<td>Z</td>
<td>B</td>
<td>Z</td>
</tr>
<tr>
<td>A</td>
<td>Z</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>A</td>
<td>Z</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

SAFARI
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 examine:
  - Decoder
  - Multiplexer
  - Full adder
  - PLA (Programmable Logic Array)
Decoder
Decoder

- “Input pattern detector”
- \( n \) inputs and \( 2^n \) outputs
- Exactly one of the outputs is 1 and all the rest are 0s
- The output that is logically 1 is the output corresponding to the input pattern that the logic circuit is expected to detect
- Example: 2-to-4 decoder

<table>
<thead>
<tr>
<th>( A_1 )</th>
<th>( A_0 )</th>
<th>( Y_3 )</th>
<th>( Y_2 )</th>
<th>( Y_1 )</th>
<th>( Y_0 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>0 0 0 0 1</td>
<td>0 1</td>
<td>0 0 1 0</td>
<td>1 0</td>
<td>0 0 0 0</td>
</tr>
</tbody>
</table>
**Decoder (I)**

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

![Diagram of decoder logic](image_url)
Decoder (II)

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

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

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

- **Selects** one of the $N$ inputs to connect it to the output based on the value of a $\log_2 N$-bit control input called **select**

- **Example:** 2-to-1 MUX

<table>
<thead>
<tr>
<th>$S$</th>
<th>$D_1$</th>
<th>$D_0$</th>
<th>$Y$</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>1</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>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</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>
Multiplexer (MUX), or Selector (II)

- **Selects** one of the $N$ inputs to connect it to the output
  - based on the value of a $\log_2 N$-bit control input called *select*
- Example: 2-to-1 MUX
Multiplexer (MUX), or Selector (III)

- 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

Your task: Draw the schematic for an 4-input (4:1) MUX

- Gate level: as a combination of basic AND, OR, NOT gates
- Module level: As a combination of 2-input (2:1) MUXes
A 4-to-1 Multiplexer
Aside: Logic Using Multiplexers

- Multiplexers can be used as lookup tables to perform logic functions

<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>

\[ Y = AB \]

Figure 2.59 4:1 multiplexer implementation of two-input AND function

Idea: Formulate the truth table as a multiplexer
Aside: Logic Using Multiplexers (II)

- Multiplexers can be used as lookup tables to perform logic functions
Aside: Logic Using Multiplexers (III)

- Multiplexers can be used as lookup tables to perform logic functions

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

\[ Y = \overline{A}B + \overline{B}C + \overline{A}BC \]

Read H&H Chapter 2.8
8-Input Lookup Table (LUT)

- **3-bit input LUT (3-LUT)**

  Data Input

  Output (1 bit)

  **Multiplexer (Mux):**
  Chooses one of the 8 data inputs that corresponds to the 3-bit select input

  Select Input

  Input (3 bits)

  3-LUT can implement any 3-bit input function
An Example of Programming a LUT

- Let’s implement a function that outputs ‘1’ when there are at least two ‘1’s in a 3-bit input

In C:

```c
int count = 0;
for(int i = 0; i < 3; i++) {
    count += input & 1;
    input = input >> 1;
}
if(count > 1) return 1;
return 0;

switch(input){
    case 0:
    case 1:
    case 2:
    case 4:
        return 0;
    default:
        return 1;
}
```
Aside: Logic Using Decoders (I)

- Decoders can be combined with OR gates to build logic functions.

![Diagram of a 2:4 decoder with a logic gate]

Figure 2.65 Logic function using decoder

\[ Y = A \oplus B \]
Full Adder
**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>$\text{carry}_i$</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 (II)

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

```
<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>
```
Creating a **4-bit adder** out of 1-bit full adders

- To add two 4-bit binary numbers A and B

![Diagram of 4-bit adder from full adders](image)

\[
\begin{array}{cccc}
    a_3 & a_2 & a_1 & a_0 \\
+ & b_3 & b_2 & b_1 & b_0 \\
    c_4 & c_3 & c_2 & c_1 \\
\hline
    s_3 & s_2 & s_1 & s_0 \\
\end{array}
\]

\[
\begin{array}{cccc}
    1 & 0 & 1 & 1 \\
+ & 1 & 0 & 0 & 1 \\
\hline
    1 & 0 & 1 & 1 \\
\hline
    0 & 1 & 0 & 0 \\
\end{array}
\]
Adder Design: Ripple Carry Adder

Figure 5.5 32-bit ripple-carry adder
Adder Design: Carry Lookahead Adder

Example of logic specialization:
Specialized logic for fast carry generation
Programmable Logic Array (PLA)
PLA: Recall: SOP Form

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

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

A PLA enables the two-level SOP implementation of any N-input M-output function.
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.

A PLA enables the two-level SOP implementation of any N-input M-output function.
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 construct.

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 see in the labs.

A PLA enables the two-level SOP implementation of any N-input M-output function.
PLA Example (I)

Read H&H Chapter 5.6.1
PLA Example Function (II)

Read H&H Chapter 5.6.1
PLA Example Function (III)

Read H&H Chapter 5.6.1
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>$cary_i$</th>
<th>$cary_{i+1}$</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>

Connections

This input should not be connected to any outputs

We do not need this output
Logical Completeness
Logical (Functional) Completeness

- **Any logic function** we wish to implement could be accomplished with a 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

- NAND is also logically complete. So is NOR.
  - **Your task**: Prove this.
More Combinational Blocks
More Combinational Building Blocks

- H&H Chapter 2 in full
  - Required Reading
  - E.g., see Tri-state Buffer and Z values in Section 2.6

- H&H Chapter 5
  - Will be required reading soon.

- You will benefit greatly by reading the “combinational” parts of Chapter 5 soon.
  - Sections 5.1 and 5.2
Comparator
Equality Checker (Compare if Equal)

- Checks if two N-input values are exactly the same
- Example: 4-bit Comparator

![Diagram of Equality Checker](image)
ALU (Arithmetic Logic Unit)
ALU (Arithmetic Logic Unit)

- Combines a variety of arithmetic and logical operations into a single unit (that performs only one function at a time)
- Usually denoted with this symbol:

![ALU Symbol](image)

**Figure 5.14 ALU symbol**

**Table 5.1 ALU operations**

<table>
<thead>
<tr>
<th>$F_{2:0}$</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>A AND B</td>
</tr>
<tr>
<td>001</td>
<td>A OR B</td>
</tr>
<tr>
<td>010</td>
<td>A + B</td>
</tr>
<tr>
<td>011</td>
<td>not used</td>
</tr>
<tr>
<td>100</td>
<td>A AND $\overline{B}$</td>
</tr>
<tr>
<td>101</td>
<td>A OR $\overline{B}$</td>
</tr>
<tr>
<td>110</td>
<td>A – B</td>
</tr>
<tr>
<td>111</td>
<td>SLT</td>
</tr>
</tbody>
</table>
## Example ALU (Arithmetic Logic Unit)

### Table 5.1 ALU operations

<table>
<thead>
<tr>
<th>$F_{2:0}$</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>A AND B</td>
</tr>
<tr>
<td>001</td>
<td>A OR B</td>
</tr>
<tr>
<td>010</td>
<td>A + B</td>
</tr>
<tr>
<td>011</td>
<td>not used</td>
</tr>
<tr>
<td>100</td>
<td>A AND $\overline{B}$</td>
</tr>
<tr>
<td>101</td>
<td>A OR $\overline{B}$</td>
</tr>
<tr>
<td>110</td>
<td>A − B</td>
</tr>
<tr>
<td>111</td>
<td>SLT</td>
</tr>
</tbody>
</table>
More Combinational Building Blocks

- See H&H Chapter 5.2 for
  - Subtractor (using 2’s Complement Representation)
  - Shifter and Rotator
  - Multiplier
  - Divider
  - ...

More Combinational Building Blocks

- H&H Chapter 2 in full
  - Required Reading
  - E.g., see Tri-state Buffer and Z values in Section 2.6

- H&H Chapter 5
  - Will be required reading soon.

- You will benefit greatly by reading the “combinational” parts of Chapter 5 soon.
  - Sections 5.1 and 5.2
Tri-State Buffer
Tri-State Buffer

- A tri-state buffer enables gating of different signals onto a wire

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Z</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Z</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>

Figure 2.40 Tristate buffer

- Floating signal (Z): Signal that is not driven by any circuit
  - Open circuit, floating wire
Example: Use of Tri-State Buffers

- Imagine a wire connecting the CPU and memory
  
  - At any time only the CPU or the memory can place a value on the wire, both not both
  
  - You can have two tri-state buffers: one driven by CPU, the other memory; and ensure at most one is enabled at any time
Example Design with Tri-State Buffers
Another Example

Diagram:

- **Processor** en1
  - to bus
  - from bus

- **Video** en2
  - to bus
  - from bus

- **Ethernet** en3
  - to bus
  - from bus

- **Memory** en4
  - to bus
  - from bus

Shared bus
Multiplexer Using Tri-State Buffers

\[ Y = D_0 \bar{S} + D_1 S \]

**Figure 2.56** Multiplexer using tristate buffers
Recall: A 4-to-1 Multiplexer
We Covered Combinational Logic Blocks

- Basic logic gates (AND, OR, NOT, NAND, NOR, XOR)
- Decoder
- Multiplexer
- Full Adder
- Programmable Logic Array (PLA)
- Comparator
- Arithmetic Logic Unit (ALU)
- Tri-State Buffer

- Standard form representations: SOP & POS
- Logical completeness
- Logic simplification via Boolean Algebra
Logic Simplification using Boolean Algebra Rules
Recall: Full Adder in SOP Form Logic

<table>
<thead>
<tr>
<th>$a_i$</th>
<th>$b_i$</th>
<th>$c_{i}^i$</th>
<th>$c_{i+1}$</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>
Goal: Simplified Full Adder

How do we simplify Boolean logic?

How do we automate simplification?

\[ S = A \oplus B \oplus C_{in} \]

\[ C_{out} = AB + AC_{in} + BC_{in} \]

<table>
<thead>
<tr>
<th>( C_{in} )</th>
<th>( A )</th>
<th>( B )</th>
<th>( C_{out} )</th>
<th>( S )</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>

3-input XOR

3-input majority

How do we simplify Boolean logic?

How do we automate simplification?
Quick Recap on Logic Simplification

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

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

- 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 (and potentially latency & power)

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>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>1</td>
</tr>
</tbody>
</table>

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

- B's value changes within the rows where $F=1$ (“ON set”)
- A's value does NOT change within the ON-set rows
- If an input (B) can change without changing the output, that input value is not needed

$\rightarrow B$ 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>

$G = \overline{A}B + A\overline{B} = (\overline{A} + A)\overline{B} = \overline{B}$

- B's value stays the same within the ON-set rows
- A's value changes within the ON-set rows

$\rightarrow A$ is eliminated, $B$ remains
## Logic Simplification

- Systematic techniques for simplifications
  - amenable to automation

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

### 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!*

- \( B \) is eliminated, \( A \) remains

<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>1</td>
<td>1</td>
</tr>
</tbody>
</table>

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

<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>

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

- \( B \)'s value stays the same within the ON-set rows
- \( A \)'s value changes within the ON-set rows
  - \( A \) is eliminated, \( B \) remains
Logic Simplification Example: Priority Circuit

- **Priority Circuit**
  - Inputs: “Requestors” with priority levels
  - Outputs: “Grant” signal for each requestor
  - Example 4-bit priority circuit
  - Real life example: Imagine a bus requested by 4 processors

![Priority Circuit Diagram]

<table>
<thead>
<tr>
<th>A3</th>
<th>A2</th>
<th>A1</th>
<th>A0</th>
<th>Y3</th>
<th>Y2</th>
<th>Y1</th>
<th>Y0</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>0</td>
</tr>
<tr>
<td>0</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>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</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>0</td>
</tr>
<tr>
<td>0</td>
<td>1</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>1</td>
<td>1</td>
<td>0</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>0</td>
<td>0</td>
<td>1</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>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</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>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Simplified Priority Circuit

- **Priority Circuit**
  - Inputs: “Requestors” with priority levels
  - Outputs: “Grant” signal for each requestor
  - Example 4-bit priority circuit

<table>
<thead>
<tr>
<th>( A_3 )</th>
<th>( A_2 )</th>
<th>( A_1 )</th>
<th>( A_0 )</th>
<th>( Y_3 )</th>
<th>( Y_2 )</th>
<th>( Y_1 )</th>
<th>( Y_0 )</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>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</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>0</td>
</tr>
<tr>
<td>0</td>
<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>0</td>
<td>0</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>1</td>
<td>0</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>
<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>
<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>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</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>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</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>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Figure 2.29** Priority circuit truth table with don’t cares (X’s)

\( X \) (Don’t Care) means *I don’t care what the value of this input is*
Logic Simplification:
Karnaugh Maps (K-Maps)
Karnaugh Maps are Fun…

- A pictorial way of minimizing circuits by visualizing opportunities for simplification
- They are for you to study on your own...

- See backup slides
- Read H&H Section 2.7
- Watch videos of Lectures 5 and 6 from 2019 DDCA course:
  - https://youtu.be/0ks0PeaOUjE?list=PL5Q2soXY2Zi8J58xLKBnFQFHRO3GrXxA9&t=4570
  - https://youtu.be/ozs18ARN6s?list=PL5Q2soXY2Zi8J58xLKBnFQFHRO3GrXxA9&t=220
We Are Done with Combinational Logic

- Building blocks of modern computers
  - Transistors
  - Logic gates

- Combinational circuits

- Boolean algebra

- Using Boolean algebra to represent combinational circuits

- Basic combinational logic blocks

- Simplifying combinational logic circuits
Logic Simplification: Karnaugh Maps (K-Maps)
Karnaugh Maps are Fun…

- A pictorial way of minimizing circuits by visualizing opportunities for simplification
- They are for you to **study on your own**...

- See remaining slides
- Read H&H Section 2.7
- Watch videos of Lectures 5 and 6 from 2019 DDCA course:
  - [https://youtu.be/0ks0PeaOUjE?list=PL5Q2soXY2Zi8J58xLKBNFQFHR03GrXxA9&t=4570](https://youtu.be/0ks0PeaOUjE?list=PL5Q2soXY2Zi8J58xLKBNFQFHR03GrXxA9&t=4570)
  - [https://youtu.be/ozs18ARN6s?list=PL5Q2soXY2Zi8J58xLKBNFQFHR03GrXxA9&t=220](https://youtu.be/ozs18ARN6s?list=PL5Q2soXY2Zi8J58xLKBNFQFHR03GrXxA9&t=220)
Backup Slides on Karnaugh Maps (K-Maps)
Complex Cases

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

- 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 find 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>A</th>
<th>B</th>
<th>00</th>
<th>01</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>01</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>10</td>
<td>11</td>
</tr>
</tbody>
</table>

3-variable K-map

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

4-variable K-map

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0000</td>
<td>0001</td>
<td>0011</td>
<td>0010</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td></td>
<td>0100</td>
<td>0101</td>
<td>0111</td>
<td>0110</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>1100</td>
<td>1101</td>
<td>1111</td>
<td>1110</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>1000</td>
<td>1001</td>
<td>1011</td>
<td>1010</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Numbering Scheme: 00, 01, 11, 10 is called a “Gray Code” — only a single bit (variable) changes from one code word and the next code word.
Karnaugh Map Methods

K-map adjacencies go “around the edges”
Wrap around from first to last column
Wrap around from top row to bottom row
Strategy for “circling” rectangles on Kmap:

Biggest “oops!” that people forget:
Logic Minimization Using K-Maps

- Very simple guideline:
  - Circle all the rectangular blocks of 1’s in the map, using the fewest possible number of circles
    - Each circle should be as large as possible
  - Read off the implicants that were circled

- More formally:
  - A Boolean equation is minimized when it is written as a sum of the fewest number of prime implicants
  - Each circle on the K-map represents an implicant
  - The largest possible circles are prime implicants
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 $\bar{x}$

- **What is a good solution?**
  - Biggest groupings $\rightarrow$ eliminate more variables (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)

F1 = A'B'C'D' + A'BC'D + ABCD + AB'CD'

K-map for F1
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>X</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>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>
<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) =

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>01</td>
<td>11</td>
<td>10</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>10</td>
<td>X</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>10</td>
<td>X</td>
</tr>
<tr>
<td>11</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>10</td>
<td>X</td>
</tr>
<tr>
<td>10</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>10</td>
<td>X</td>
</tr>
</tbody>
</table>

W X
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

- H&H Section 2.7